Vergleich und Beurteilung von Java Frameworks für Web Services mit REST

Diplomarbeit

Eingereicht von Andreas Schneider

28. Februar 2010

Datenverarbeitungstechnik
Fakultät für Mathematik und Informatik
FernUniversität in Hagen

Betreuer:
Prof. Dr.-Ing. Bernd J. Krämer
Dipl.-Inf. Daniel Schulte

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und nur mit den angegebenen Hilfsmitteln angefertigt habe. Wörtlich oder inhaltlich übernommene Literaturquellen wurden besonders gekennzeichnet.

Norderstedt, 28. Februar 2010

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation und Ziel

In seiner Dissertation [Fie00] hat Roy Thomas Fielding einen Architekturstil für Web Services beschrieben, dessen Bedeutung in letzter Zeit immer weiter zugenommen hat. Dieser Stil heißt REpresentational State Transfer oder kurz REST. Fielding begann schon früh ein Architekturkonzept für Webprotokolle zu entwickeln, welches ursprünglich den Namen „HTTP Object Model“ trug [Til09, S. 7]. Er war an der Standardisierung von HTTP/1.0 und der Weiterentwicklung zu HTTP/1.1 beteiligt. Während dieser Zeit entwickelte er dieses Konzept stetig weiter. In seiner Dissertation abstrahiert er sein Konzept und nannte es REST. Im engeren Sinne ist REST nicht auf die HTTP-Architektur beschränkt, sondern ein Stil, der eine Stufe abstrakter ist und sich theoretisch „auch mit einem neu erfundenen Satz von Protokollen umsetzen“ [Til09, S. 8] ließe. Das Web ist jedoch tatsächlich die einzige konkrete Ausprägung dieses Architekturstils [Til09, S. 8], weshalb der Begriff REST häufig synonym verwendet wird[1].

Demgegenüber stehen die Web Services auf Basis von SOAP und WSDL. Beide Ansätze sind offensichtlich so unterschiedlich, dass es viele Debatten darüber gab und gibt, welches der bessere Weg sei. Diese Debatte wird häufig mit voreingenommenen und religiösen Argumenten geführt, was nur zu Verwirrung und Erwartungen führt, die nicht eingehalten werden können [PZL08, S. 1]. Obwohl die Diskussionen nicht immer sachlich geführt werden, gibt es Anwendungsfälle, für die SOAP besser geeignet ist und einige, für die sich REST besser eignet. Brian Sletten schreibt in [Sle08], dass SOAP geeigneter ist um Verhaltensweisen zu beeinflussen, während sich REST gut für die Verwaltung von Informationen eignet. Dennoch gibt es Überlappungen in den Einsatzgebieten. So können die meisten Web Services mit beiden Stilen umgesetzt werden. Es haben sich jedoch zwei Lager gebildet, die jeweils einen der beiden Stile vorziehen. Dabei wurde REST bisher von der kleineren der beiden Gruppen bevorzugt. Es ist jedoch ein Trend Richtung REST erkennbar, der durch „eine zunehmende Frustration mit dem Web-Services-Technologiestack“ [Til09, S. 8] bedingt ist. Auch die „immer stärkere Verbreitung von Web-APIs ohne SOAP“ [Til09, S. 8] führt zu einem gestiegenen Interesse an REST bzw. RESTful HTTP.

[1] Auch in dieser Arbeit wird dem allgemeinen Sprachgebrauch gefolgt.

Diesem Trend folgend gibt es inzwischen einige Frameworks, welche die Erstellung von RESTful Web Services vereinfachen wollen. In dieser Arbeit werden ausgewählte Projekte aus dem Javabereich untersucht und verglichen:

- Servlet-API
- JAX-WS Referenz-Implementierung
- Jersey (JAX-RS Referenz-Implementierung)
- RESTEasy
- Restlet
- Spring 3 MVC

Als Grundlage des Vergleichs dienen die von Roy Fielding in [Fie00] erarbeiteten Konzepte des Architekturstils REST. Diese grundlegenden Konzepte werden analysiert und auf dieser Basis ein Fragenkatalog abgeleitet, der für alle Frameworks beantwortet wird. Um einige der Fragen besser beantworten zu können, wird eine Beispielanwendung entworfen und mit den zu untersuchenden Frameworks realisiert.

1.2. Aufbau der Arbeit

Im Kapitel 2 wird zunächst dargelegt, was ein Web Service ist und was es bedeutet, dass ein Web Service RESTful heißt. Die Grundlagen zu REST sind stark auf die Dissertation [Fie00] von Roy Fielding bezogen und bilden die Grundlagen für den Vergleich. Der Fragenkatalog, auf dessen Basis die Frameworks verglichen werden, wird im ersten Abschnitt des Kapitels 3 erarbeitet. In den weiteren Abschnitten wird die Beispielanwendung entworfen, welche exemplarisch mit allen Frameworks implementiert werden soll. Dies beinhaltet die Architektur, einige Details zur Implementierung sowie die Beschreibung des eingesetzten Testclients. Die Anwendung des Fragenkatalogs findet im Kapitel 4 statt. Hier wird jedes Framework kurz vorgestellt, das Vorgehen bei der Realisierung erklärt und die Antworten auf die Fragen gegeben. Jeder Framework-Abschnitt wird durch eine zusammenfassende Beurteilung abgerundet. Die Ergebnisse der Untersuchungen werden in Kapitel 5 gegenübergestellt und verglichen. Mit einem abschließenden Fazit und einem Ausblick in Kapitel 6 wird die Arbeit abgeschlossen.

2. Grundlagen

In diesem Kapitel wird eine Basis für die verwendeten Begriffe und Technologien geschaffen. Die Inhalte sind im Wesentlichen dem Buch *„Web Services mit REST“* von Richardson und Ruby [RR07, S. 20 ff.] entnommen.

Ein Web Service ist eine Software-Anwendung, die über eine definierte Schnittstelle eine bestimmte Funktionalität zur Verfügung stellt. Zwischen Client und Service werden über HTTP Nachrichten ausgetauscht. Der Client schickt einen Request an den Service, welcher meistens auf einem Server im Web zur Verfügung gestellt wird. Der Service bearbeitet den Request, führt die gewünschte Anforderung durch und liefert dem Client eine Response mit den Antwortdaten. Beispiele für Web Services sind das Abrufen der Wetterinformationen an einem bestimmten Ort, die Suche mit einer Suchmaschine oder die Verwaltung von Artikeln eines Onlineshops.

Einige ältere Web Services nutzen XML-RPC[1]. Dabei werden die Standards HTTP und XML miteinander kombiniert. XML-RPC ist ein Datenstrukturformat zum Darstellen und Übermitteln von Funktionsparametern, Funktionsaufrufen und Rückgabewerten. Viele der heutigen Web Services nutzen SOAP[2], ein weiteres Format zur Nachrichtenübertragung wie HTTP. Im Gegensatz zu HTTP ist SOAP XML-basiert. Mit SOAP können beliebige XML-Nachrichten übermittelt werden. Bestehende Web Services nutzen allerdings immer ein Format, das XML-RPC ähnelt [RR07, S. 21]. SOAP wird also darauf beschränkt, als „Umschlag“ für XML-RPC zu dienen. Die verfolgte Philosophie hinter beiden Ansätzen ist identisch, es soll auf dem Server eine Prozedur aufgerufen werden. Über eine XML-Nachricht wird dem Web Service mitgeteilt, welche Prozedur aufgerufen werden soll. Die Nachricht enthält auch erwartete Parameter dieser Prozedur. Der Web Service gibt das Ergebnis des Prozeduraufrufs ebenfalls als XML-Nachricht an den aufrufenden Client zurück.

Roy Fielding beschreibt in [Fie00] einen anderen Ansatz. Dabei erfindet er nichts grundlegend Neues, sondern beschreibt einen Architekturstil namens REST, der bereits in vielen Anwendungen verwendet wird. Angelehnt an HTTP werden mittels einer generischen Schnittstelle Ressourcen[3] angeboten und modifiziert. HTTP ist zwar das ge-

[1] Remote Procedure Call
[2] ursprünglich für Simple Object Access Protocol
[3] Ressourcen sind die *Objekte* der Anwendung (siehe 2.1)

bräuchlichste Protokoll, REST ist allerdings nicht darauf beschränkt. Eine Bedingung für REST ist lediglich die Einheitlichkeit der Protokoll-Schnittstelle, d.h. ein vorgegebener Satz von Operationen (auch Verben genannt) und ihrer Bedeutung. Mit diesen Operationen werden Ressourcen angelegt, verändert, gelesen und gelöscht. Die Ressourcen selbst werden über eine URI[4] identifiziert. Wenn der Aufrufer die URI einer Ressource kennt, kann er aufgrund der einheitlichen Schnittstelle intuitiv Requests an den Server schicken, um diese Ressource zu verwenden. Die folgende Grafik 2.1 verdeutlicht den Unterschied zwischen der Verwendung der einheitlichen Schnittstelle links im Bild und SOAP via HTTP auf der rechten Seite.

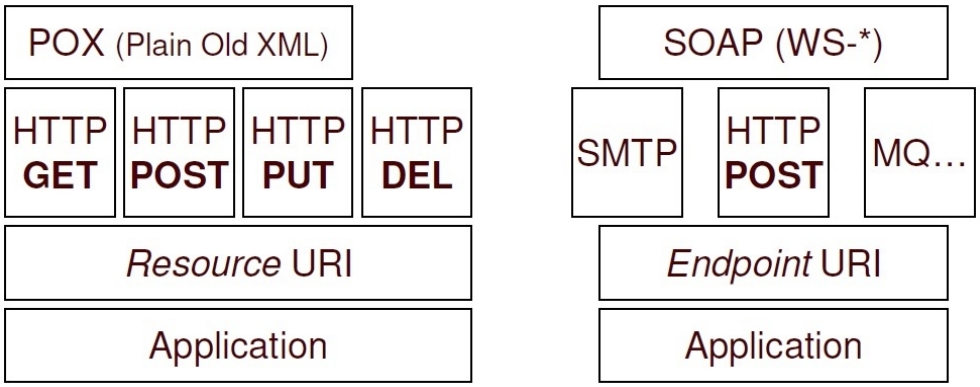


Abbildung 2.1.: REST vs. SOAP
[Pau07, S. 21]

Roy Fielding verfolgt mit REST [Fie00, Kapitel 4.1] im Wesentlichen folgende Architekturziele:

- Das System soll möglichst einfach sein, damit die Hemmschwelle, sich daran zu beteiligen so gering wie möglich ist.
- Das System soll einfach erweiterbar sein, um neue Teilnehmer besser zu integrieren, sowie vorhandene Systeme besser aktualisieren oder ersetzen zu können.
- Das System soll der Darstellung verteilter Hyper-Medien dienen, was den Transfer größerer Datenmengen vom Server zum Konsumenten impliziert. Dabei soll die Antwortzeit so gering wie möglich sein, insbesondere die vom Benutzer beobachtete Latenz, also die Zeit zwischen Senden der Anforderung und Bemerken

[4] ursprünglich für Universal Resource Identifier, später Uniform Resource Identifier
Stefan Tilkov erklärt in [Til09, S. 36 f.], dass der Begriff URL vom allgemeineren Begriff URI abgelöst wurde. In dieser Arbeit wird zumeist der Begriff URI verwendet. Weitere Informationen z.B. zur Syntax von URIs unter [BL94].

einer Reaktion, d.h. die Darstellung der Antwort, auch wenn diese ggf. noch unvollständig sein kann.

- Das System soll den gewaltigen Dimensionen genügen, d.h. es muss die geographische Ausdehnung genauso berücksichtigen wie die Überschreitung von Länder-, Konzern- und Kulturgrenzen. Durch die Verteilung des Systems muss dieses anarchisch erweiterbar sein. Weiterhin soll der Einsatz von Sicherheitsmechanismen unterstützt werden. Verschiedene Protokollversionen müssen einfach erwartet werden, im Idealfall sind höhere Versionen abwärtskompatibel.

Kurz gesagt, sind die Ziele von REST Einfachheit, Erweiterbarkeit, Skalierbarkeit, Performanz und Einheitlichkeit. Brian Sletten beschreibt in einem Artikel in der Javaworld.com die Vorteile von REST. Er schreibt dort, dass in Anbetracht neuer Anforderungen, Technologien und Anwendungsfälle die Systeme so einfach und erweiterbar wie möglich sein sollten [Sle08].

REST bringt einige Vorteile mit sich, ohne die das Web nicht so erfolgreich gewesen wäre, wie es zweifellos war und ist. In den folgenden Abschnitten werden die Konzepte von REST zur Erreichung dieser Ziele kurz vorgestellt. Die Darstellungen sind ebenfalls stark an die Ausführungen von Richardson und Ruby in [RR07, Kapitel 4] angelehnt.

2.1. Ressourcen

Eine Ressource wird von Richardson und Ruby als „... alles, was wichtig genug ist, um als eigenständiges Etwas referenziert zu werden“ [RR07, S. 91] beschrieben. Damit kann eine Ressource nahezu alles sein. Die W3C Technical Architecture Group beschreibt in der Architektur des World Wide Web [Gro04, 2.1] eine Ressource so:

> „A resource should have an associated URI if another party might reasonably want to create a hypertext link to it, make or refute assertions about it, retrieve or cache a representation of it, include all or part of it by reference into another representation, annotate it, or perform other operations on it. Software developers should expect that sharing URIs across applications will be useful, even if that utility is not initially evident.“

Ressourcen können z.B. Abstraktionen physischer Objekte sein, aber auch virtuelle Objekte, die nur als Bytestrom existieren. Ressourcen sind beispielsweise „*Version 1.0.3 des Software-Releases*“, „*die letzte Version des Software-Releases*“ [RR07, S. 92], ein *Stadtplan von Hamburg* oder eine *Kundenliste*.

Nach den Prinzipien von REST müssen Ressourcen eindeutig identifizierbar sein. In dem hier betrachteten Kontext, dem Internet, wurde zu diesem Zweck der Begriff URI

eingeführt. Jede Ressource besitzt mindestens eine eigene URI. Wie der Name schon besagt, wird dadurch eine Ressource eindeutig identifiziert, sie ist gewissermaßen der Name dieser Ressource. Ressourcen können auch mehrere URIs besitzen. Mehrere Ressourcen können zeitweise dieselben Daten liefern. So bedeuten die Ressourcen *Version 1.0.3 des Software-Releases* und *die letzte Version des Software-Releases* einen identischen Stand der Software, wenn die letzte Version gerade Version 1.0.3 ist. Es sind dennoch verschiedene Ressourcen. Idealerweise geht aus der URI bereits hervor, worum es sich bei dieser Ressource handelt. Dies ist allerdings keine Bedingung für REST, lediglich eine Konvention für guten Stil. Eine weitere Empfehlung bei der Vergabe der URIs besagt, dass sie eine Struktur, eine Hierarchie besitzen sollten [RR07, S. 94]. Dadurch werden abhängige Ressourcen ersichtlich.

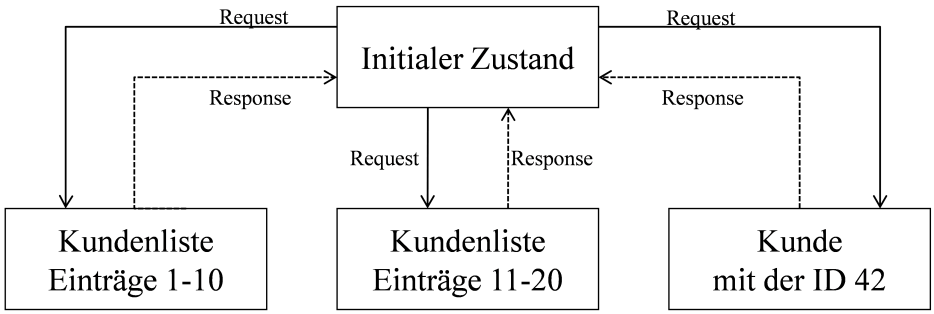
Um Ressourcen vor unbefugten Zugriffen zu schützen, kann für Requests eine Authentifizierung gefordert werden. An dieser Stelle wird auf die beiden in HTTP verankerten Authentifizierungsschemata *Basic Authentication* und *Digest Authentication* kurz eingegangen, die Inhalte wurden [Til07, S. 126 ff.] entnommen.
Für das Schema Basic sendet der Client im Header `Authorization` sogenannte *Credentials* an den Service. Konkret werden Benutzername und Passwort, durch einen Doppelpunkt getrennt, hintereinander geschrieben. Die gesamte Zeichenkette wird nach dem Base64-Verfahren codiert. Dies stellt keine Verschlüsselung dar. Die Zeichenkette wird lediglich auf ein vereinfachtes Alphabet abgebildet, um Probleme zu vermeiden, die durch eine erzwungene Zeichenkonvertierung bei der Übertragung auftreten können. Der Server, aber auch ein Angreifer der die Nachricht mitliest, kann die Credentials wieder dekodieren und mit denen auf dem Server gespeicherten vergleichen. Dieses Schema für sich betrachtet ist noch relativ unsicher. Durch die Kombination mit HTTPS wird es jedoch zu einem Verfahren, welches für die meisten Anwendungen ausreichend ist. Tatsächlich ist dies die am häufigsten verwendete Lösung in existierenden Anwendungen. Das Schema Digest ist bedeutend aufwändiger, bietet jedoch nur wenig mehr Sicherheit und hat sich nicht in der Praxis durchgesetzt. Welches Schema der Server einsetzt, kann der Client der Response entnehmen, die er mit dem Statuscode *401 Unauthorized* erhält. Zusätzlich wird hier vom Server ein Wert namens `realm` vorgegeben, womit der Geltungsbereich der Authentifizierungsdaten gemeint ist.

2.2. Zustandslosigkeit

Die Forderung der Zustandslosigkeit des Servers bedeutet, dass jeder HTTP-Request isoliert betrachtet ausführbar ist. Der Server verwendet keine Informationen aus früheren Requests, um eine Anfrage zu beantworten. Jede Information, die zur Beantwortung

relevant ist, wird in dem Request mitgeschickt. Anders ausgedrückt heißt das, dass jeder mögliche Zustand des Servers eine eigene Ressource mit einer eigenen URI darstellt, die von Clients adressiert werden kann.

Angenommen, ein Client fordert eine Ressource an, die eine Liste von Kunden darstellt. Der Web Service könnte so programmiert sein, dass er die Liste paginiert, wenn sie zu viele Einträge enthält, um sie vollständig aufzuführen. Der Server könnte dann eine Repräsentation liefern, welche die ersten 10 Einträge darstellt. Diese Teilliste könnte einen Verweis auf die nächsten 10 Einträge enthalten. Diese und weitere Teillisten wiederum enthielten dann jeweils einen Verweis auf die vorige und auf die nächste Teilliste. Diese Verweise sind URIs, die genau genommen verschiedene Ressourcen referenzieren, wie „Liste der Kunden ab dem 11. Eintrag“. Clients sollten mithilfe dieser URIs direkt den von ihnen gewünschten Teil der Liste adressieren können und nicht gezwungen sein, zunächst die erste Teilliste, anschließend die nächste Teilliste usw. anfordern zu müssen. Der Server muss also nicht unbedingt in einem bestimmten Zustand sein, bevor er einen bestimmten anderen Zustand annehmen kann. Er bietet einfach Ressourcen an, die von Clients angefordert werden können. Das Zustandsdiagramm in Abbildung 2.2 zeigt, wie ein Client mit verschiedenen Zuständen einer Kundenverwaltung interagiert. Die Zugriffe geschehen völlig unabhängig voneinander, es handelt sich um eine zustandslose Anwendung.



In Anlehnung an: [RR07, Abbildung 4-1]

Abbildung 2.2.: Zustandsloser Web Service

Durch die Zustandslosigkeit ist es allerdings erforderlich, dass der Client ggf. Informationen mehrfach an den Server senden muss. Wenn z.B. der Client durch diverse Filterkriterien die Liste der Kunden eingeschränkt hätte, würde er diese Filterkriterien bei jedem Aufruf erneut übermitteln. Hätte der Server sich beim 1. Aufruf gemerkt, dass sich dieser Client nur für Kunden interessiert, die diesen Kriterien genügen, müsste der Client sie bei weiteren Aufrufen nicht erneut übertragen.

Eine Reihe möglicher Fehlerquellen können durch die Zustandslosigkeit vermieden werden, wie Probleme durch Timeouts oder Fehler bei der Synchronisation zwischen Server und Client. Zusätzlich werden Eigenschaften gewonnen, die ohne Zustandslosigkeit nicht oder nur schwer erreichbar sind, wie ein einfacherer Lastausgleich, da jeder einkommende Request von jedem Server eines Verbunds beantwortet werden kann, ohne dass sich die Server untereinander synchronisieren müssen. Eine Skalierung des Verbunds ist ebenfalls sehr einfach möglich, indem einfach neue Server hinzugeschaltet werden. Durch die isolierte Betrachtung wird auch ein Caching wesentlich einfacher. Wird ein Request im Cache gefunden, kann die dazugehörige Response aus dem Cache verwendet werden, da sie nicht durch einen vorherigen Request des Clients beeinflusst ist.

HTTP ist ein zustandsloses Protokoll, daher sind Web Services von sich aus auch zustandslos. Um dies zu ändern, muss aktiv etwas getan werden. Beispielsweise könnte der Server eine Session-ID zurückliefern, welche der Client bei jedem weiteren Aufruf mitgibt. Dies wäre REST-konform, wenn diese Session-ID den Zustand enthalten würde. Ist die Session-ID allerdings nur ein Schlüssel für einen Status, der vom Server verwaltet wird, widerspricht dies REST.

Richardson und Ruby unterscheiden in ihrem Buch *„Web Services mit Rest“* [RR07, S. 101] zwei verschiedene Arten von Zuständen. Den Zustand von Ressourcen einerseits und den Zustand der Anwendung andererseits. Den Zustand einer Ressource darf und soll sich der Server merken. Clients dürfen diesen durch entsprechende Requests verändern, aber der Server alleine weiß, welchen Zustand eine Ressource gerade hat. Zustandslosigkeit im Sinne von REST betrifft lediglich den Anwendungszustand. Dieser wird vom Client verwaltet und betrifft den Server nur in dem Moment, in dem ein Request eintrifft, um ihn zu beantworten. In der Zeit zwischen zwei Requests weiß der Server nichts von der Existenz der Clients.

2.3. Repräsentationen

Richardson und Ruby bezeichnen eine Ressource als „eine Quelle für Repräsentationen, und eine Repräsentation besteht einfach aus Daten zum aktuellen Zustand einer Ressource“ [RR07, S. 103]. Eine Repräsentation ist also eine Darstellung einer Ressource. Ressourcen können mehrere Darstellungen, also mehrere Repräsentationen besitzen. Beispielsweise könnte eine Liste von Kunden als kommaseparierte Datei, im JSON[5]-, XML- oder HTML-Format bereitgestellt werden. Eine Statistik könnte als Tabelle oder

[5] JavaScript Object Notation, URI: `http://json.org/`

als Diagramm angeboten werden. Ein Bild könnte in unterschiedlichen Qualitätsgraden verfügbar und eine Pressemitteilung könnte in mehreren Sprachen abrufbar sein.

Bietet ein Server für eine Ressource mehrere Repräsentationen an, muss der Client dem Server mitteilen, für welche davon er sich interessiert. Dazu könnte jede Repräsentation eine eigene URI bekommen: `http://example.net/pressemitteilungen/4711.de` für die deutsche und `http://example.net/pressemitteilungen/4711.en` für die englische Mitteilung beispielsweise. Allerdings sieht es nach außen hin so aus, als handelte es sich hierbei um zwei verschiedene Ressourcen.

Roy Fielding beschreibt in seiner Dissertation [Fie00, Abschnitt 6.3.2.7] eine andere Möglichkeit, das sogenannte *Content Negotiation*. Der HTTP-Request enthält bestimmte Header, die der Client angeben kann. Dadurch kann er anzeigen, welche Art von Repräsentationen er akzeptieren wird bzw. welches Format übertragen wird, wenn es sich um modifizierende Requests handelt. Diese Header kennzeichnen verschiedene Metadaten, wie beispielsweise Sprache, Dateiformat oder Authentifizierungsinformationen. REST gibt allerdings nicht vor, dass diese Metadaten im Header gesetzt werden müssen. Es ist ebenso REST-konform, sie in der URI anzugeben. Dies zu tun erscheint auf den ersten Blick etwas seltsam, da einerseits die URI unnötig lang wird und andererseits speziell dafür vorgesehene Header ungenutzt bleiben. Dennoch gibt es einige Vorteile, diese Metainformationen in der URI anzugeben. URIs werden von Mensch zu Mensch weiter weitergegeben und sie werden in Repräsentationen (z.B. auf Webseiten) eingebunden um auf andere Ressourcen zu verweisen. Die Header-Informationen gehen dabei meist verloren. Einige Web Services erwarten URIs als Query Parameter, z.B. der W3C-HTML-Validator. Dieser Service kann nur eine Repräsentation anfordern, auch wenn mit dieser URI mehrere Repräsentationen abrufbar sind. Ohne die jeweiligen Header erhält der Validator immer die Default-Repräsentation.

Es ist allerdings durchaus möglich, beide Varianten miteinander zu verbinden. Die Ressource kann mit einer einheitlichen URI angeboten und die *Content Negotiation* über Header gesteuert werden. Gleichzeitig kann für jede Repräsentation eine explizite URI angeboten werden. Dadurch sind alle Repräsentationen über beide Wege verfügbar.

2.4. Verweise

In der Repräsentation können Verknüpfungen, sogenannte Links, zu weiteren Ressourcen enthalten sein. Folgt der Client einem Link, erhält er eine Repräsentation der zugehörigen Ressource. Der Client ändert also seinen Status. Anders ausgedrückt findet ein Transfer von einem Repräsentations-Status zum nächsten Repräsentations-Status

statt. Diesem Statuswechsel verdankt REST seinen Namen. Die Verknüpfung von Ressourcen ist zwar keine Bedingung von REST, Fielding bezeichnet in seiner Dissertation allerdings „hypermedia as the engine of application state“ [Fie00, Kapitel 5.1.5].

In [RR07, S. 108] wird dieses Axiom derart gedeutet, „dass der aktuelle Zustand einer HTTP-'Session' nicht auf dem Server als Ressourcen-Zustand gespeichert, sonderm vom Client als Anwendungszustand verfolgt und durch den Pfad erzeugt wird, den der Client im Web beschreitet“. Dadurch, dass die Repräsentationen Ressourcen miteinander verbinden, indem sie aufeinander verweisen und somit Beziehungen zwischen ihnen offensichtlicher werden, gibt der Server dem Client eine Hilfestellung, durch den Service zu navigieren.

Stefan Tilkov verwendet in [Til07] innerhalb der Beispielanwendungen das <link>-Element. In [Til07, S. 75 ff.] beschreibt er den Ursprung in der HTML-Spezifikation und die wachsende Bedeutung mit dem zunehmenden Einsatz von RESTful HTTP für andere Zwecke. Auf die Bedeutung von Verweisen in Repräsentationen wird mehrfach hingewiesen. Demnach sollten „lieber zu viele als zu wenig“ [Til07, S. 75] Links in den Repräsentationen enthalten sein. Tilkov bezeichnen sogar Anwendungen, für die Hypermedia keine zentrale Rolle spielt, als nicht RESTful [Til07, S. 79].

2.5. Einheitliche Schnittstelle

Wie bereits in Kapitel 1 erwähnt, abstrahiert Roy Fielding in [Fie00] die bestehende Internet-Architektur. Diese verwendet HTTP als Transport-Protokoll. HTTP genügt den Kriterien einer einheitlichen Methoden-Schnittstelle, die von REST verlangt wird. Die aktuelle Version des Protokolls ist HTTP/1.1 [FGM+99]. Sie unterstützt acht verschiedende Methoden: GET, POST, PUT, DELETE, OPTIONS, HEAD, TRACE und CONNECT.

- GET – Lesen der Ressource.
- POST – Hinzufügen einer Ressource oder Sub-Ressource. Dabei ist die URI dieser neuen Ressource noch nicht bekannt, sie wird vom Server festgelegt und in der Response dem Client mitgeteilt. Eine weitere Funktion der Methode POST ist die Bereitstellung von Daten an einen datenverarbeitenden Prozess.
- PUT – Hinzufügen oder Ändern einer Ressource, deren URI bekannt ist.
- DELETE – Löschen einer Ressource.
- OPTIONS – Abfrage erlaubter Methoden zu der Ressource.

- HEAD – Wie GET, nur Response-Body bleibt leer. Dient der Abfrage von Meta-Information zu der Ressource ohne eine möglicherweise große Repräsentation direkt mit zu übertragen.
- TRACE – Nur für Testzwecke.
- CONNECT – Nur für Proxies, die andere Protokolle weiterleiten.

Die ersten vier Methoden sind die gebräuchlichsten, sie dienen der Abfrage und Manipulation von Ressourcen. OPTIONS und HEAD können nützlich sein, um Metainformationen zu erhalten, während TRACE und CONNECT hier nicht weiter von Bedeutung sind.

Einige Autoren, wie beispielsweise Thomas Bayer in [Bay02, 1.2] oder Sameer Tyagi[6] verwenden die Methoden PUT und POST nicht immer einheitlich, wie es hier beschrieben wurde. Dabei wird zumeist die Methode PUT zum Anlegen und POST zum Ändern der Ressource verwendet. Durch diese Interpretation gleichen die vier zentralen Methoden den Operationen in der Datenbank-Abfragesprache SQL[7] INSERT, SELECT, UPDATE und DELETE. Sie entspricht allerdings nicht der HTTP-Spezifikation.

Viele Web Services nutzen lediglich die POST-Methode. Die Information, welche Aktion mit einem Request durchgeführt werden soll, befindet sich nicht in der Methode, sondern „in dem URI, den HTTP-Headern oder dem Entity-Body“ [RR07, S. 115]. Durch die ausschließliche Verwendung einer Methode kann im Zusammenhang mit REST nicht von einer einheitlichen Schnittstelle gesprochen werden. Diese Art der Verwendung entspricht eher dem RPC-Stil.

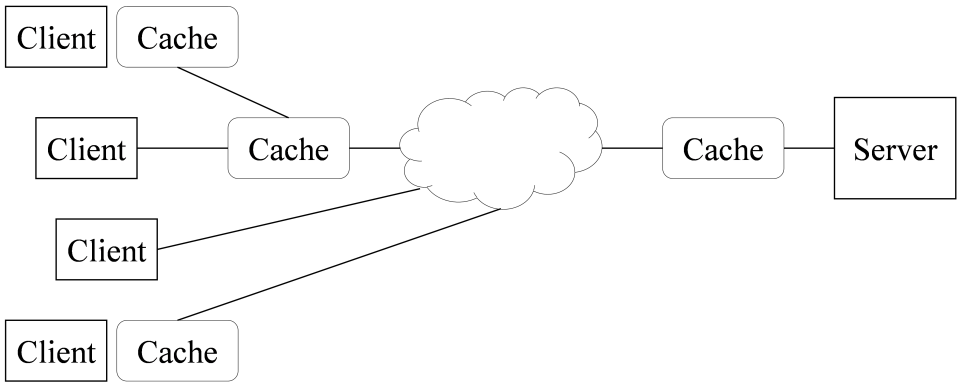
Wichtig an der Verwendung der einheitlichen Schnittstelle ist nicht, dass es die von HTTP bereitgestellten Methoden sind. Sondern es geht darum, dass die Schnittstelle einheitlich ist. Es wäre durchaus möglich, die Schnittstelle zu ändern indem beispielsweise Methoden dazukommen, wegfallen, umbenannt werden oder ähnliches. Hauptsache ist, dass der Gebrauch der Methoden verständlich ist und einheitlich verwendet wird. Jeder Service sollte die Schnittstelle auf die gleiche Art und Weise verwenden. Dadurch müssen nicht bei jedem Service die eingesetzten Methoden und ihre Verwendung gelernt werden. Dies vereinfacht die Bedienung der Schnittstelle durch einen Computer. Wenn dieser die URI einer Ressource und ihre Bedeutung kennt, kann er den Service bedienen, da er weiß, wie die Schnittstelle funktioniert.

Die HTTP-Schnittstelle besitzt außerdem zwei weitere Eigenschaften: Sicherheit und Idempotenz. Sicherheit bedeutet, dass ein Request bedenkenlos abgeschickt werden

[6] URI: `http://java.sun.com/developer/technicalArticles/WebServices/restful/` (Stand: 28.02.2010)
[7] Structured Query Language

kann, wenn die Methode eine Ressource nicht verändert. GET, HEAD und OPTIONS sind solche Methoden. Bei Idempotenz handelt es sich um einen aus der Mathematik stammenden Begriff. Er bedeutet, dass ein Web Service sich immer gleich verhält, unabhängig davon, wie oft ein Request abgeschickt wird. Der Aufruf der DELETE-Methode ist idempotent: Nach dem ersten Aufruf kann die Ressource nicht mehr abgerufen werden, ebensowenig nach dem zweiten oder n-ten Aufruf von DELETE. HEAD und GET sind als sichere Methoden natürlich auch idempotent. Bei richtiger Verwendung ist auch die PUT-Methode idempotent. Beim Aufruf von PUT werden die Attribute der Ressource auf die übermittelten Werte gesetzt. Auch nach einem wiederholten Aufruf erhalten die Attribute genau diese Werte. Services sollten die PUT-Methode nicht verwenden, um eine Operation auszuführen, wie beispielsweise einen Zähler zu inkrementieren. Dadurch hätte die Ressource nach jedem erneuten Aufruf einen veränderten Zustand, dies wäre nicht idempotent.

Eine Möglichkeit, den Web Service zu entlasten, ist der Einsatz von Caches. Dabei werden erzeugte Repräsentationen zusätzlich zur Übertragung an die Client-Anwendung in einem Cache gespeichert. Caches können im Server, im Client oder dazwischen in Infrastrukturelementen wie Proxies eingesetzt werden. Fordert ein Client eine Ressource an, die bereits in einem Cache liegt, so kann die im Cache gespeicherte Kopie an den Client geliefert werden. Die Anwendung muss die Repräsentation also nicht erneut erzeugen. Befindet sich der Cache-Eintrag auf Client-Seite, wird zusätzlich Bandbreite eingespart, da die Repräsentation nicht über möglicherweise große Strecken übertragen werden muss. Dies nützt allerdings nichts, wenn ein weiterer Client dieselbe Ressource anfordert. Eine komplexe Kombination von Caches zeigt Abbildung 2.3.



In Anlehnung an: [Til09, Abbildung 10-4]

Abbildung 2.3.: Cache-Topologie

Caching gilt als eines der Hauptargumente für den Einsatz von REST-konformen Anwendungen. Roy Fielding beschreibt in [Fie00, S. 32] Caching als ein wesentliches Element, welches entscheidend für die Skalierbarkeit des Webs ist. Das Caching-Verhalten kann vom Server, aber auch vom Client beeinflusst werden. Hierfür steht der Header `Cache-Control` sowohl im Request als auch in der Response zur Verfügung. Die Bedeutung ist jeweils eine andere. Im Response-Header gibt der Server an, ob und wenn ja, wie lange die Ressource in einem Cache abgelegt werden darf. Nach dieser Zeitspanne wird sie als *abgelaufen (stale)* gekennzeichnet. Diese Kennzeichnung sollte auch vorgenommen werden, wenn der Cache einen PUT-, POST- oder DELETE-Request für diese Ressource registriert. Web Services, die vorwiegend POST einsetzen, können deren Ergebnisse in den meisten Fällen nicht cachen.
Der Client kann den Request-Header nutzen, um anzugeben, ob er Daten aus einem Cache akzeptiert und wenn ja, welche Bedingungen diese erfüllen müssen. Dies können Angaben zum Alter oder zur Restgültigkeitsdauer sein.

Eine Alternative oder auch Ergänzung zum Caching sind bedingte Requests, wie sie beispielsweise in [Til09, S. 115 f.] vorgestellt werden. In der Response eines GET-Requests gibt der Server kennzeichnende Metadaten zu der Ressource an den Client zurück. Im einfachsten Fall ist dies der Header `Last-Modified`. Diesen kann sich der Client merken und bei einem späteren Request im Header `If-Modified-Since` angeben. Er verknüpft also den Request mit einer Bedingung. Die Anwendung prüft, ob die Ressource in der Zwischenzeit verändert wurde. Ist dies nicht der Fall, antwortet sie mit dem Statuscode *304 Not Modified*. Anderenfalls wird ganz normal die Repräsentation erzeugt und übertragen. Diese Methode entlastet den Service nur, wenn die Erstellung der Repräsentation signifikant lange dauert oder sonstige Systemressourcen in Anspruch nimmt. Jedoch wird Bandbreite eingespart, wenn keine Repräsentation übertragen werden muss.

Die Anwendung kann einen weiteren Header `ETag` verwenden, um eine Repräsentation zu kennzeichnen. Es wird zwischen schwachen und starken ETags unterschieden. Schwache ETags werden nur aus der internen Darstellung der Ressourcen erzeugt, die Repräsentationen können leicht abweichen. Starke ETags hingegen berücksichtigen die komplette Repräsentation, hierbei dürfen gleiche ETags nur entstehen, wenn die Repräsentationen Byte für Byte identisch sind. Analog zum letzten Änderungsdatum überträgt der Client das ETag im Header `If-None-Match`. Eine neue Repräsentation wird dann nur übertragen, wenn die Ressource verändert wurde und sich dadurch auch das ETag geändert hat. Anwendungen, die beide Header unterstützen, sollten auch beide füllen und beide vergleichen.

Die Funktionsweise von bedingten PUT- und DELETE-Requests, wie sie Tilkov in [Til09, S. 164] vorstellt, ist sehr ähnlich. Der Client verwendet jedoch die Header `If-Unmodified-Since` und `If-Match` im Request, um sicherzustellen, dass sich die

Ressource seit dem letzten Request nicht verändert hat. Schlägt die Vorabprüfung fehl, d.h. die Ressource wurde zwischenzeitlich verändert, bekommt der Client den Statuscode *412 Precondition Failed*. Die Überprüfung des Datums ist etwas unsicherer als der Vergleich des ETags. Ursache hierfür ist die relativ grobe Granularität von HTTP, das Datum nur sekundengenau anzugeben, wodurch bei häufig frequentierten Änderungsrequests die Gefahr sogenannter *Lost Updates* besteht.

Eine Vorabprüfung ist prinzipiell auch für POST möglich. In einem RESTful Web Service wird POST meist zum Anlegen neuer (Sub-)Ressourcen verwendet. Es kann jedoch Gründe geben, diese Neuanlage an Bedingungen zu knüpfen.

3. Methodik

Um die Frameworks möglichst objektiv vergleichen und beurteilen zu können, wird in Abschnitt 3.1 ein Fragenkatalog entwickelt. Ziel des Katalogs ist es, die Frameworks auf einer einheitlichen Basis zu untersuchen.

Um die Frameworks im praktischen Einsatz zu untersuchen, wird jeweils exemplarisch ein Web Service implementiert. Aus Gründen der Einheitlichkeit sind die entstandenen Web Services in ihrer Funktionalität weitgehend[1] identisch, in ihrer Implementierung je nach Framework individuell. Diese Funktionalität wird im Abschnitt 3.2 beschrieben. Anschließend wird im Abschnitt 3.3 der eingesetzte Testclient vorgestellt. Mit einer knappen Zusammenfassung in Abschnitt 3.4 wird das Kapitel abgeschlossen.

3.1. Fragenkatalog

In diesem Abschnitt wird der Fragenkatalog zusammengestellt, anhand dessen die Frameworks untersucht und beurteilt werden. Das Hauptaugenmerk dieser Untersuchung liegt auf der Unterstützung der Erstellung von RESTful Web Services. Einen Schwerpunkt bei der Erstellung des Fragenkatalogs bildet die Tauglichkeit der Frameworks, die von Roy Fielding in seiner Dissertation [Fie00] entworfenen Konzepte umzusetzen bzw. zu unterstützen. Die Konzepte von REST wurden im Kapitel 2 aufgeführt. Diese dienen im Folgenden als Gliederung und Stütze bei der Zusammenstellung der Fragen.

3.1.1. Ressourcen und Schnittstelle

Der Architekturstil REST ist ressourcenorientiert. Ressourcen sind die Objekte, die Substantive der Anwendung, mit denen etwas getan werden kann. Welche Tätigkeiten das sind, bestimmt die einheitliche Schnittstelle, die Verben der Anwendung. Diese Verben sind bei den hier betrachteten Web Services durch das Protokoll HTTP vorgegeben. Für die Untersuchung ist es relevant, welche HTTP-Methoden der Web Service anbieten kann. Es wird untersucht, welche Methoden mit Hilfe des Frameworks umgesetzt werden können, um die Einheitlichkeit der Schnittstelle zu gewährleisten.

[1] Ein absolut identisches Verhalten ist nicht mit jedem Framework realisierbar.

Die Ressourcen sind je nach Anwendung verschieden. Sie müssen vom Entwickler erstellt werden. Dabei ist eine wichtige Frage, wie Ressourcen als solche gekennzeichnet, im Framework verwaltet und welche Funktionalität im Umgang mit den Ressourcen angeboten wird.

Mit Hilfe von URIs werden Ressourcen unterschieden und identifiziert. Wie die URIs den Ressourcen zuordnet werden, ist ein weiterer Aspekt der Untersuchung. In diesem Zusammenhang wird auch der Umgang mit Template- und Query-Parametern betrachtet.

Außerdem werden die vom Framework angebotenen Authentifizierungs-Mechanismen betrachtet. Die verschlüsselte Übertragung wurde hierbei bewusst ausgeklammert, da diese Aufgabe im Normalfall vom Applikationsserver umgesetzt werden kann. Eine spezifische Framework-Lösung wäre lediglich für einen Standalone-Einsatz interessant.

Daraus lassen sich folgende Fragen ableiten:
Frage 1: Wie erfolgt die Implementierung von Ressourcen?
Frage 2: Wie geschieht die Zuordnung zwischen Ressourcen und ihren URIs?
Frage 3: Welche HTTP-Methoden können umgesetzt werden?
Frage 4: Werden Sicherheitsmechanismen zur Authentifizierung angeboten?

3.1.2. Statushaltung und Caching

Die Zustandslosigkeit des Servers ist ebenfalls ein wichtiges Merkmal von REST. Vor und nach einem Request soll der Server nichts über den Zustand der Clients wissen. Es wird untersucht, ob die Serverkomponenten des Frameworks vom Zustand der Clients entkoppelt sind.

Eine weitere Frage bezieht sich darauf, ob und wie Caching vom Framework unterstützt wird. Caching betrifft eher den Zustand der Ressourcen als den Zustand der Clients. Je nach Anwendung kann es einen erheblichen Unterschied in der Antwortzeit ausmachen.

Auch bedingte Zugriffe können die Antwortzeiten bzw. die Netzwerklast reduzieren. Hierbei kann das Framework durch Überprüfung der Bedingungsheader unterstützen. Hilfreich ist es auch, wenn das Framework die Kalkulation und den Vergleich von ETags anbietet.

Hieraus ergeben sich diese Fragen:
Frage 5: Verwaltet die Serverkomponente Anwendungsstatus der Clients?
Frage 6: Unterstützt das Framework Caching?
Frage 7: Inwieweit unterstützt das Framework bedingte Requests?

3.1.3. Repräsentationen und Verweise

Clients holen bzw. liefern Repräsentationen von Ressourcen. Mithilfe der Content Negotiation kann der Client über die Art der Repräsentation entscheiden. Ob und wie das Framework Content Negotiation unterstützt, wird untersucht. Außerdem wird geprüft, ob das Framework dem Entwickler die Aufbereitung von Repräsentationsformen abnimmt, so dass diese Darstellung der Ressource nicht explizit implementiert werden muss.

Ein ebenfalls wichtiger Aspekt ist die Unterstützung von Hypermedia. Einige Bestandteile der Repräsentation können und sollten Verweise auf andere Ressourcen sein. Deshalb wird eine besondere Unterstützung von Verweisen in Repräsentationen ebenfalls untersucht.

Es werden daher diese Fragen formuliert:
Frage 8: Müssen alle Repräsentationsformate explizit implementiert werden?
Frage 9: Wie erfolgt die Content Negotiation?
Frage 10: Werden Verweise der Ressourcen untereinander in Repräsentationen unterstützt?

3.1.4. Zusammenfassung und Bewertung

Die Gliederung der Fragen wird auch bei deren Beantwortung beibehalten. So bestehen die Abschnitte zu den Frameworks jeweils aus einem Unterabschnitt zur *API Implementierung*, dann aus den drei Gliederungspunkten *Ressourcen und Schnittstelle*, *Statushaltung und Caching* sowie *Repräsentationen und Verweise*. Eine *Zusammenfassung und Bewertung* schließt jeden Abschnitt ab. Die Fragen des ermittelten Fragenkatalogs werden in der Zusammenfassung jedes Frameworks in einer Tabelle noch einmal abschließend bewertet. Diese Bewertung beruht zu wesentlichen Teilen auf der gesammelten Erfahrung bei der Implementierung. Die Skala bedeutet dabei im Einzelnen:

`--` Mit dem Framework ist eine Umsetzung dieses Aspekts nicht möglich.
`-` Die Realisierung ist möglich, aber umständlich.
`o` Das Framework unterstützt hierbei nicht explizit.
`+` Dieser Aspekt ist gut, aber nicht ideal umsetzbar.
`++` Hierbei bietet das Framework eine sehr gute Unterstützung.

3.2. Beispielanwendung

Bei der mit allen Frameworks exemplarisch erstellten Beispielanwendung handelt es sich um eine Kundenverwaltung. Dies ist ein einfaches, aber gängiges Beispiel. Die umgesetzte Funktionalität ist jeweils identisch, soweit das möglich ist. Mit dem Web Service sollen neue Kunden angelegt, die Daten bereits bestehender Kunden geändern und die Daten eines oder mehrerer Kunden abgerufen werden können.

Normalerweise liegt beim Design der Ressourcen, also der Objekte des Web Services, die Idee des zu erstellenden Services bereits vor. So können mit den zur Verfügung stehenden Verben[2] die Ressourcen festgelegt werden, mit denen die gewünschten Anforderungen realisiert werden können. Sandoval beschreibt in [San09, S.70] vier Schritte beim Entwurf von RESTful Web Services:

1. Sammlung der Anforderungen
2. Ermittlung der Ressourcen
3. Festlegung der Repräsentationen
4. Definition der URIs

Bei der Erstellung des Fallbeispiels wird ein wenig anders vorgegangen. Im Mittelpunkt des Entwurfs steht nicht der Inhalt der fertigen Anwendung, sondern der Wunsch, möglichst verschiedene Situationen zu integrieren, um die jeweiligen Ansätze in den Frameworks besser beurteilen zu können. Nach den ersten Überlegungen soll die Anwendung aus wenigstens einer Primärressource[3] und einer Subressource bestehen, um den Einsatz der verschiedenen HTTP-Methoden zu integrieren. Wenigstens für die Primärressourcen soll es eine zugehörige Listenressource geben. Um die Content Negotiation untersuchen zu können, soll mindestens eine der Ressourcen verschiedene Repräsentationsformate unterstützen.

Zunächst werden die Ressourcen ermittelt, mit denen die genannten Nebenbedingungen erfüllt werden können. Anschließend werden die jeweiligen Repräsentationsformate festgelegt. Danach werden die URIs definiert, mit denen die Ressourcen identifiziert werden können. Schließlich wird auf die konkrete Umsetzung der Web Services mit Hilfe der Frameworks eingegangen.

[2] Bei HTTP: GET, POST, PUT und DELETE

[3] Gemeint ist hier ein Ressourcentyp. Es ist jedoch üblich, den Begriff Ressource sowohl für einen Typ als auch für eine konkrete Instanz zu verwenden.

3.2.1. Ermittlung der Ressourcen

Im Mittelpunkt der Kundenverwaltung steht die zentrale Ressource *Kunde*. Sie soll einige Attribute enthalten, die nicht alle vom Datentyp `String` sind, um mögliche Probleme bei der Typumwandlung zu erkennen. *Kunde* erhält die Attribute *Name*, *Vorname* und *Geburtsdatum*, wobei letzteres vom Datentyp `Date` ist.

Als nächstes soll die Anwendung eine abhängige Subressource enthalten. Hierfür wird die Ressource *Adresse* mit den Attributen *Straße*, *Stadt*, *Bundesland* und *Postleitzahl* bestimmt. Um die Einführung einer eigenen Ressource zu rechtfertigen, sollte *Kunde* mehrere *Adressen* haben können. Bei einer 1:1 Beziehung könnten die Attribute der Ressource *Adresse* in die Ressource *Kunde* mit aufgenommen werden. Die beiden Ausprägungen der *Adresse* sind eine Rechnungs- und eine Lieferadresse. Die IDs der *Adressen* sind dem Client bekannt, es sind die beiden Ausprägungen ≫billing≪ für die Rechnungsadresse und ≫shipping≪ für die Liederadresse erlaubt. Die Kenntnis der IDs erlangt der Client über Verweise in den Repräsentationen, siehe 3.2.2.

Die Methode PUT wird verwendet, wenn dem Client die ID bekannt ist, wie es bei den *Adressen* der Fall ist. Kennt der Client die ID der neuen Ressource nicht, werden Subressourcen mit der Methode POST erzeugt. Um beide Situationen in das Beispiel aufzunehmen, wird eine weitere Subressource *Telefonnummer* geschaffen. Sie enthält lediglich die Attribute *Name*[4] und *Nummer*. Einem *Kunden* können beliebig viele *Telefonnummern* zugefügt werden. Ihre ID wird vom Service vergeben.

Um einen Überblick über bereits erfasste *Kunden* zu bekommen, bietet die Anwendung eine *Kundenliste* an, welche eine eigene Ressource darstellt. Um die Verwendung von Query-Parametern in das Beispiel zu integrieren, soll die *Kundenliste* nach *Name* und *Vorname* gefiltert werden können, d.h. die Ergebnisliste wird eingeschränkt. Eine weitere Ressource ist die *Liste der Telefonnummern eines Kunden*.

Der Schwerpunkt dieser Arbeit ist die Untersuchung der verschiedenen Frameworks. Für diesen Zweck reichen die gewählten Ressourcen aus. Grundsätzlich ist dieses Beispiel jedoch beliebig erweiterbar. So könnten neben den Kunden auch Artikel verwaltet werden, welche in Bestellungen und Warenkörben miteinander verknüpft wären usw.

Die Beispielanwendung enthält somit die fünf in Abbildung 3.1 dargestellten Kern-Ressourcen. Die Bezeichner orientieren sich an der tatsächlichen Umsetzung und sind deshalb in englischer Sprache verfasst. Eine Übersicht der erlaubten Verben für die Ressourcen wird in Abschnitt 3.2.3 im Zusammenhang mit den festgelegten URIs gegeben.

[4] Z.B. ≫Büro≪ oder ≫Privat≪

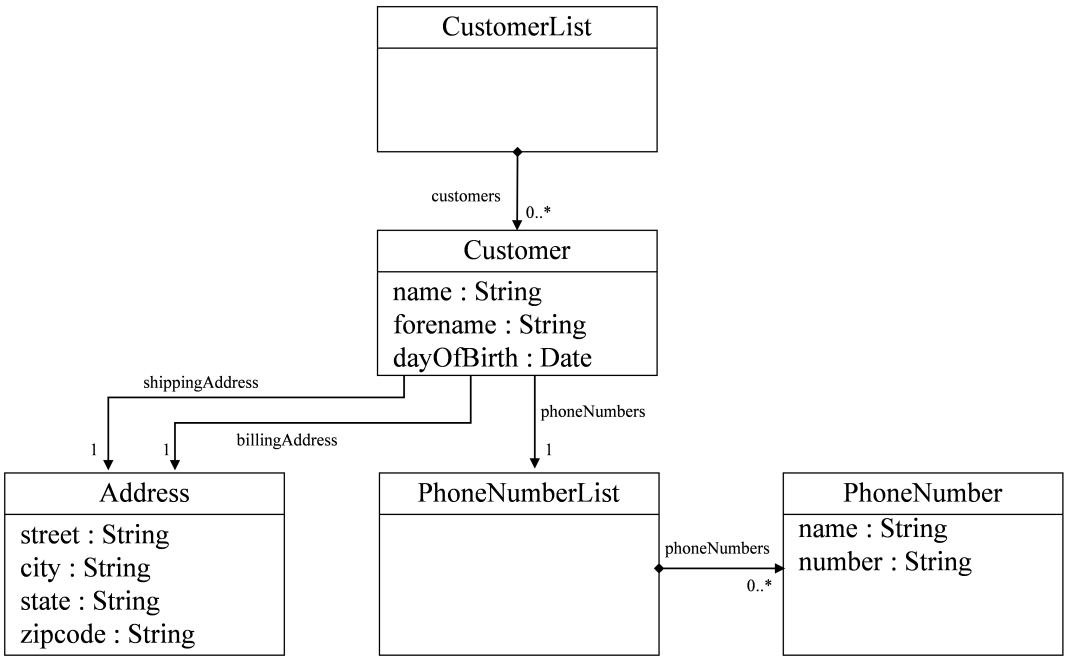


Abbildung 3.1.: Ressourcen der Beispielanwendung

3.2.2. Festlegung der Repräsentationen

Hinter dem Medientyp *application/xml* verbirgt sich das wahrscheinlich verbreitetste Repräsentationsformat [Til09, S. 82]. Mit XML lassen sich Daten gut strukturiert aufbereiten. Es lässt sich von einem Programm gut parsen und weiterverarbeiten und ist dadurch ein sehr geeignetes Repräsentationsformat für Web Services. Deshalb wurde es auch für dieses Beispiel eingesetzt. XML wird für sämtliche hier implementierten Ressourcen akzeptiert und angeboten.

Für den Test der Content Negotiation wurde ein ebenfalls häufig verwendetes und leicht zu erzeugendes Repräsentationsformat eingesetzt: JSON. Da es für die Untersuchung keinen nennenswerten Nutzen gibt, mehrere Repräsentationsformate für alle Ressourcen anzubieten, beschränkt sich dieses zweite Format in der Beispielanwendung auf die Abfrage der Kundenliste mit Hilfe der GET Methode.

Die Repräsentationen enthalten Verweise auf andere Ressourcen. Die Kundenliste verweist auf die enthaltenen Kunden. Die Darstellung des Kunden wiederum enthält Verweise auf die Ressourcen von abhängigen Adressen und Telefonnummern. So wird es einem Client ermöglicht, ohne vorherige Kenntnis der URI-Struktur, Ressourcen anzufordern oder zu verändern.

Die Listings A.1 und B.1 im Anhang zeigen die verfügbaren Repräsentationsformate.

Sie sind das Ergebnis je eines GET-Requests mit den MIME[5]-Typen *application/xml* und *application/json* auf die Ressource Kundenliste. Hierfür wurde exemplarisch das Ergebnis der Umsetzung mit RESTEasy gewählt.

3.2.3. Definition der URIs

Mit welchen URIs die Ressourcen adressiert werden können und welche Auswirkung die einzelnen Methoden haben, wird im Folgenden beschrieben. URIs werden hierbei relativ zu einer Basis-URI angegeben, welche einen Bezeichner für das Framework enthält, um diese während der Tests zu unterscheiden.

Die Ressource *Liste aller Kunden* wird über die URI /customers angesprochen. Ein einzelner Kunde wird über die URI /customers/{custid}[6] adressiert.

Die Methode GET liefert für die URI /customers/{custid} als Response die Repräsentation dieses Kunden. GET für die URI /customers liefert eine Liste mit den Repräsentationen aller Kunden. Diese Liste kann mit Hilfe der Query-Parameter `name` und `forename` eingeschränkt werden.

Für die URI /customers wird mit POST eine neue Subressource Kunde angelegt. Die Daten eines Kunden werden geändert, wenn PUT für die URI /customers/{custid} aufgerufen wird.

Mit derselben URI und der Methode DELETE kann ein Kunde wieder gelöscht werden. Es ist auch möglich, alle Kunden mit einem Aufruf der Methode DELETE für die URI /customers zu löschen. Hierfür muss sich der Client jedoch über das HTTP-Basic-Schema authentifizieren, d.h. im Header `Authorization` müssen entsprechende Authentifizierungsdaten mitgegeben werden.

Tabelle 3.1 zeigt noch einmal zulässige Methoden und ihre Response für die Ressourcen Kundenliste und Kunde.

Methode	/customers	/customers/{custid}
GET	Lese Kundenliste	Lese Kunde
PUT	nicht unterstützt	Aktualisiere Kunde
POST	Lege neuen Kunden an	nicht unterstützt
DELETE	Lösche alle Kunden, wenn authentifiziert	Lösche Kunden

Tabelle 3.1.: Ressourcen *Kundenliste* und *Kunde*

[5] Multipurpose Internet Mail Extensions

[6] {custid} steht hierbei für eine Template-Variable, die als Platzhalter für die tatsächliche Kunden-ID betrachtet wird. Die Spezifikation solcher Templates steht unter folgender URI: `http://bitworking.org/projects/URI-Templates/spec/draft-gregorio-uritemplate-03.html` (Stand: 28.02.2010)

In Kapitel 2.1 wurde eine Empfehlung erwähnt, abhängige Ressourcen durch eine Hierarchie in den URIs sichtbar zu machen. Dieser Empfehlung wird hier gefolgt.

Die Methoden für die Liste der Telefonnummern eines Kunden sind über die URI /customers/{custid}/phones adressierbar. Durch die Hierarchie wird deutlich gemacht, dass die Telefonnummern abhängige Ressourcen sind, die einem speziellen Kunden zugeordnet sind. Die URIs der Telefonnummern enthalten die URI des jeweiligen Kunden. Die Methode GET dient zum Lesen der Liste, POST zum Erstellen einer neuen Telefonnummer. Über die URI /customers/{custid}/phones/{id} wird eine einzelne Telefonnummer angesprochen. Analog zu den Methoden für /customers/{custid} werden mit GET die Daten der Telefonnummer gelesen, mit PUT geändert und mit DELETE gelöscht.

Die zulässigen Methoden und ihre Response für die Ressourcen Telefonnummer und die Liste werden in Tabelle 3.2 noch einmal zusammengefasst.

Methode	/customers/{custid}/phones	/customers/{custid}/phones/{id}
GET	Lese Liste der Telefonnummern für diesen Kunden	Lese Telefonnummer
PUT	nicht unterstützt	Aktualisiere Telefonnummer
POST	Lege neue Telefonnummer für diesen Kunden an	nicht unterstützt
DELETE	nicht unterstützt	Lösche Telefonnummer

Tabelle 3.2.: Ressourcen *Liste der Telefonnummern eines Kunden* und *Telefonnummer*

Für die Adressen eines Kunden steht keine Liste zur Verfügung. Einem Kunden können nur höchstens zwei Adressen zugeordnet werden: die Rechnungs- und die Lieferadresse. Die URIs hierfür stehen fest und werden nicht vom Service dynamisch vergeben. Sie lauten /customers/{custid}/addresses/billing und /customers/{custid}/addresses/shipping. Auf ihnen sind jeweils die Methoden GET, PUT und DELETE zum Lesen, Ändern und Löschen erlaubt. Anders formuliert lautet die URI /customers/{custid}/addresses/{id}, wobei für {id} lediglich die Werte »billing« und »shipping« erlaubt sind.

Methode	/customers/{custid}/addresses/{id}
GET	Lese Adresse
PUT	Aktualisiere Adresse
POST	nicht unterstützt
DELETE	Lösche Adresse

Tabelle 3.3.: Ressource *Adresse*

Diese Hierarchie in den URIs ist nicht selbstverständlich. Das Framework muss dafür

die Verwendung von URI-Templates mit Variablen unterstützen. Diese Bedingung wird nicht von allen Frameworks erfüllt. Daher wurde für diese Frameworks eine alternative URI-Struktur geschaffen. Für diese Struktur werden IDs der Kunden, Adressen und Telefonnummern als Query-Parameter übergeben. Um die Pfade der Listen von denen der Primärressourcen unterscheiden zu können, werden die ansonsten im Plural gehaltenen Elemente für Primärressourcen im Singular angegeben. Beispielsweise lautet die relative URI für die Liste der Kunden /customers und für den Kunden mit der ID 42 /customer?custid=42.
Eine Übersicht der URI-Templates wird in den Tabellen 3.4 und 3.5 gegeben.

Ressource	URI-Template
Kundenliste	/customers
Kunde	/customers/{custid}
Telefonnummernliste	/customers/{custid}/phones
Telefonnummer	/customers/{custid}/phones/{id}
Adresse	/customers/{custid}/addresses/{id}

Tabelle 3.4.: Hierarchische URI-Struktur der Ressourcen

Ressource	URI-Template
Kundenliste	/customers
Kunde	/customer?custid={custid}
Telefonnummernliste	/customer/phones?custid={custid}
Telefonnummer	/customer/phone?custid={custid}&phoneid={id}
Adresse	/customer/address?custid={custid}&addressid={id}

Tabelle 3.5.: Nicht-hierarchische URI-Struktur der Ressourcen

Soweit dies möglich war, wurde die hierarchische Struktur bevorzugt, weil die alternative Struktur einige Probleme mit sich bringt. So speichern einige Cache-Implementierungen keine Response im Cache, wenn die Request-URI Query-Parameter enthält. In einigen Frameworks wird die Request-URI aufgeteilt, so dass auf Pfad und Query separat zugegriffen werden muss. Dadurch ist der Code weniger kompakt, es entsteht eine zusätzliche Fehlerquelle und die Fehlersuche wird erschwert.

Die genaue Dokumentation der Schnittstelle des Web Services ist in Anhang C wiedergegeben. Als Format für die Dokumentation wurde WADL[7] gewählt. Dabei handelt es sich um eine Beschreibungssprache, die explizit auf RESTful HTTP zugeschnitten ist. Stefan Tilkov stellt in [Til09, Kapitel 12.4.2] WADL kurz vor und beschreibt, warum WADL die REST-Prinzipien besser unterstützt als beispielsweise WSDL[8].

[7] Web Application Description Language, URI: `http://www.w3.org/Submission/wadl/` (Stand: 28.02.2010)
[8] Web Services Description Language

Damit der Client tatsächlich nur Kenntnis von einer Start-URI braucht, wird in jedem Projekt eine statische Startressource hinterlegt. Diese enthält eine Liste der Einstiegspunkte als attributierte Links. Diese wird ebenfalls in den Formaten XML und JSON angeboten und via Content Negotiation über Dateiendungen angefordert, d.h. diese Startressource ist unter /start.xml bzw. /start.json zu finden.
Die folgende Darstellung 3.1 zeigt die Startressource im XML-Format für das Framework RESTEasy.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <serviceEntryPoints xml:base="http://localhost:8080/RESTEasy">
3     <link rel="customer list" href="/customers"/>
4 </serviceEntryPoints>
```

Listing 3.1: XML-Repräsentation der Startressource

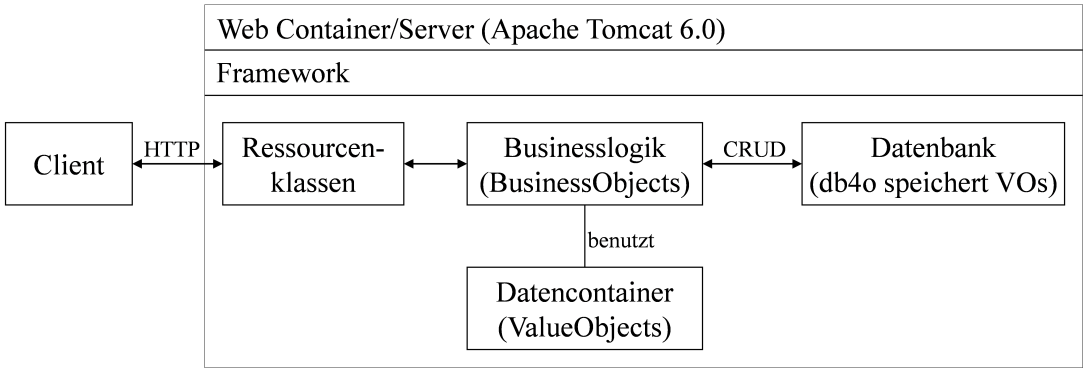
Ein Client muss so lediglich wissen, dass ein Eintrag mit dem Wert `customer list` im Attribut `rel` gesucht werden muss und dass im zugehörigen Attribut `href` die URI einer Kundenliste zu finden ist. Wird diese mit der ebenfalls enthaltenen Basis-URI kombiniert, kann der Client Requests für die Kundenliste absetzen. Auch wenn der Nutzen in diesem Beispiel begrenzt ist, schließlich enthält sie nur einen Link, der dem Client stattdessen direkt mitgeteilt werden könnte, zeigt es doch, wie die Lösung in einer realistischen Anwendung aussehen könnte.

3.2.4. Konkrete Implementierung

Die Implementierung des Fallbeispiels erfolgt pro Framework in einem separaten Eclipse-Projekt. Die Projekte werden zur Ausführung in einen Application Server eingebunden. Dafür wird der Apache Tomcat Application Server in der Version 6.0 verwendet. Die Web Services werden mit einem HTTP-Client bedient, welcher in Kapitel 3.3 ausführlich beschrieben wird. Abbildung 3.2 zeigt die Architektur der implementierten Web Anwendung.

Um die Frameworks besser vergleichen zu können, wird bei der Umsetzung der Beispielanwendung mit den verschiedenen Frameworks vornehmlich das Ziel verfolgt, eine einheitliche Reaktion auf eine bestimmte Requestfolge zu erreichen. Um das einheitliche Verhalten zu erreichen, wird ein von allen Web Services verwendetes Projekt `Common` erstellt. Darin wird die fachliche Verarbeitung implementiert. Lediglich die frameworkspezifischen Teile sind in den anderen Projekten enthalten.

Das Common-Projekt enthält die `ValueObject`s, welche die Daten für die drei Objekte Kunde, Adresse und Telefonnummer enthalten. Sämtliche `ValueObject`s enthalten eine



In Anlehnung an: [San09, S. 91]

Abbildung 3.2.: Architektur der Web Anwendung

ID, eine URI und das Datum der letzten Änderung. Die URI wird für die Angabe in der Repräsentation verwendet, deshalb wird ebenfalls für die beiden Listen eine Ableitung von `ValueObject` implementiert. Das Datum wird von den Ressourcenklassen in einigen Frameworks für die Prüfung bei bedingten Zugriffen verwendet. Eine besondere Bedeutung kommt der Klasse `CustomerService` zu, sie stellt die „äußere Klammer“ einschließlich des Werts `xml:base` in der Repräsentation dar. Dazu beinhaltet sie ein `ValueObject`, welches die darzustellenden Daten enthält.

Wie im Kapitel 4 gezeigt wird, unterstützen einige Frameworks die Erzeugung von Repräsentationen aus JAXB-Objekten, teilweise sowohl für XML als auch für JSON. Aus diesem Grund wurden die `ValueObject`s mit JAXB-Annotationen versehen. Die Annotationen werden von einigen Marshallern[9] verwendet, um relevante Attribute, Typen und ggfs. in der Darstellung abweichende Namen zu ermitteln und aufzubereiten. Da nicht alle Frameworks die Erstellung der Repräsentationen übernehmen, wurde die Erzeugung der XML- und JSON-Repräsentationen zusätzlich im Common-Projekt realisiert. Für die Erstellung der XML-Repräsentation wird die Klasse `Marshaller` der in Java integrierten JAXB-API verwendet. Bei der Erzeugung der JSON-Repräsentation kam der `JSONSerializer` des Projekts JSON-lib[10] zum Einsatz.

Auch die Content Negotiation wird nicht von allen Frameworks umgesetzt, so dass im Common-Projekt die Utility-Klasse `CheckMediaTypeUtil` implementiert wurde. Sie enthält eine Methode, mit der überprüft werden kann, ob das im Header `Content-Type` angegebene Format akzeptiert wird, sowie eine Methode zur Prüfung des Headers

[9] Marshalling bedeutet, ein Datenobjekt in ein Format zu transformieren, das sich für eine Übertragung eignet. Das Wiederherstellen der ursprünglichen Datenstruktur aus dem Transportformat heißt Unmarshalling.

[10] URI: `http://json-lib.sourceforge.net/`

`Accept` für die angebotenen Formate. Gleichzeitig wird das geeignetste Format ermittelt. Schlägt die Prüfung der Header fehl, wird jeweils eine Exception geworfen, die die eigens erstellte Klasse `CommonException` erweitert. Diese Exceptions enthalten einen fest zugeordneten HTTP-Statuscode. Über den Konstruktor wird eine sprechende Fehlermeldung mitgegeben.

Die `ValueObject`s werden mit Hilfe des Objektdatenbanksystem db4o[11] persistiert. Hierbei handelt es sich um eine einfach einzubindende Datenbank, für die eine freie Lizenz für den nicht-kommerziellen Einsatz angeboten wird. In der Datenbank werden ganze Objekte gespeichert, ohne dass diese dafür besonders vorbereitet werden müssen, sei es durch Annotationen oder Implementierung eines Interfaces. Zudem zeichnet sich db4o durch einen sehr geringen Speicherbedarf aus. So ist eine einfache Persistierung der Ressourcen möglich.

Weiterhin beinhaltet das Common-Projekt `BusinessObject`s, welche die Verwaltung (Erzeugen, Persistieren, Laden, Ändern und Löschen sowie Repräsentationen erstellen) der `ValueObject`s realisieren. Die `BusinessObject`s werfen im Fehlerfall eine Exception, die ebenfalls von `CommonException` abgeleitet ist. Die framework-spezifischen Ressourcenklassen fangen diese Exceptions und reagieren entsprechend. In den meisten Fällen gibt der Web Service den zugehörigen Fehler-Statuscode und eine dazu passende Meldung im Response-Body an den Aufrufer zurück.

Wann welcher Statuscode geliefert werden soll, ist im Wesentlichen durch das Protokoll vorgegeben [FGM+99]. Zur Überprüfung der Reaktionen der jeweiligen Implementierung dient der im folgenden Abschnitt 3.3 vorgestellte Testclient.

3.3. Testclient

Um die implementierten Web Services hinreichend testen zu können, wird ein Testclient implementiert. Seine Aufgaben sind das Abschicken von Requestfolgen, Auswerten der Responses und Vergleich mit erwarteten Antworten. Für die technische Umsetzung wird zunächst die Utility-Klasse `HttpRequestUtil` implementiert. Diese ist für die Erstellung des Requests und Übertragung zum Server sowie Auswerten und Ausgabe der Response zuständig. Als Basis dafür dient der Jakarta Commons HttpClient[12] von Apache in der Version 3.1. Die Durchführung der Tests erfolgt mit Hilfe von JUnit[13] 4.8.1. Sämtliche Testfälle sind in der abstrakten Testklasse `TestClient` umgesetzt. Diese wird

[11] URI: `http://www.db4o.com/`
[12] URI: `http://hc.apache.org/httpclient-3.x/`
[13] URI: `http://www.junit.org/`

für jedes Framework spezialisiert, um dort über einen Aufzählungstyp festzulegen, welches Framework getestet wird, also unter welcher URI die Startseite abgerufen werden soll. Die Testklassen sind zusätzlich in einer Suite zusammengefasst, um in einem Testlauf nacheinander ausgeführt zu werden.

Die Testklasse wird sukzessive erweitert, um möglichst viele verschiedene Situationen zu beleuchten und die Reaktionen zu überprüfen. So enthält sie neben den Grund-Funktionen zum Anlegen, Ändern, Lesen und Löschen der Ressourcen auch Negativtestfälle für die jeweils nicht unterstützten Methoden und für ungültige sowie nichtnumerische IDs in den URI-Templates. Außerdem werden die Methoden OPTIONS und HEAD aufgerufen, verschiedene Repräsentationsformate von Ressourcen angefordert, das Löschen aller Kunden mit und ohne Authentifikation sowie bedingte GET- und PUT-Requests mit Änderungsdatum bzw. ETag getestet. Eine umfassende Auflistung der Requests und erwarteter Antworten ist in Anhang D zu finden. Dort werden auch die tatsächlich erhaltenen Antworten der Frameworks gegenübergestellt und verglichen.

Die mit den Frameworks realisierten Web Services sollen sich möglichst gleich verhalten. So war das ursprüngliche Ziel, dass alle für das Fallbeispiel erstellten Anwendungen nicht nur die gleiche Funktionalität bieten, sondern auch ihre Antworten gleiche Statuscodes und gleichen Inhalt, d.h. gleichen Response-Body haben. Während der Untersuchung zeigte sich, dass dieses Ziel nicht immer einzuhalten war. Zum einen ist es so, dass die Frameworks einige Requests automatisch beantworten. Dies betrifft beispielsweise Requests für unzulässige Methoden oder nicht unterstützte Repräsentationsformate. Diese automatisch erzeugten Responses unterscheiden sich teilweise von Framework zu Framework. Für einige Situationen wäre es möglich gewesen, eine selbst implementierte Response anstelle der vom Framework erzeugten zurückzugeben. So hätten auch eigentlich unzulässige Methoden explizit implementiert werden können, so dass diese den Statuscode *405 Method Not Allowed* inklusive gefülltem `Allow` Header liefern. Dies hätte jedoch die Untersuchung verfälscht, da ja das eigentliche Verhalten des Frameworks untersucht werden soll.

Eine weitere Abweichung tritt auf, wenn ein Framework eine bestimmte Funktionalität nicht explizit unterstützt, wie beispielsweise bedingte Requests. Dies hätte durch Auswertung der Header und Implementierung der Vergleichsalgorithmen für alle Web Services realisiert werden können. Einen Nutzen für die Untersuchung hätte dies jedoch nicht gebracht. So werden schließlich Abweichungen im Test akzeptiert, anstatt zu versuchen, Schwächen einzelner Frameworks durch zusätzliche Implementierungen auszugleichen.

In einem typischen Einsatzszenario für Unit-Tests ist es das Ziel, dass sämtliche Tests

jederzeit erfolgreich abschließen. Ist dies nicht der Fall, wird die Ursache gesucht und entweder der Fehler im Prüfling behoben oder das erwartete Ergebnis im Test umformuliert. Da aus den oben genannten Gründen die Responses nicht immer identisch sind, hätten für diese abweichenden Situationen verschiedene Ergebnisse erwartet werden müssen. Dies hätte einerseits durch Fallunterscheidungen in den Tests erreicht werden können, was jedoch zu sehr unübersichtlichen Testfällen geführt hätte. Eine alternative Lösung wäre die Vervielfältigung der Testklasse gewesen, was es wiederum erschwert hätte, die Einheitlichkeit der Tests zu gewährleisten. Keine der Lösungen ist optimal, daher werden Fehler während der Testdurchführung hingenommen und dokumentiert. Dies wird deshalb als unkritisch angesehen, da es sich nicht um Unit-Tests von realen Anwendungen handelt, sondern vielmehr um Funktionstests verschiedener Frameworks.

3.4. Zusammenfassung

In diesem Kapitel wurde die Vorgehensweise bei der Umsetzung der Beispielanwendung und ihrer Untersuchung erläutert. Die Basis der gesamten Untersuchung bildet dabei der Fragenkatalog mit seinen 10 enthaltenen Fragen.

Für die Implementierung wurden framework-übergreifende und -spezifische Teile getrennt. Das Ergebnis sind die Projekte für die Frameworks, die ein gemeinsames Common-Projekt benutzen, welches die `ValueObjects`, `BusinessObjects` und die Persistierung in einer Datenbank enthält. Die Bedienung der Anwendung erfolgt durch automatisierte Tests. Der dafür eingesetzte Testclient besteht aus JUnit-Testklassen, die per Generalisierung identische Testfälle ausführen. Mit ihrer Hilfe wurden alle Frameworks den gleichen Tests unterzogen. Die hierbei aufgetretenen Abweichungen von den erwarteten Ergebnissen wurden überprüft. Einige Abweichungen wurden nicht korrigiert, sondern akzeptiert und in Anhang D dokumentiert.

4. Frameworks

Das Ziel dieser Arbeit ist es, verschiedene Frameworks zur Erstellung RESTful Web Services zu untersuchen und zu vergleichen. Der Schwerpunkt liegt dabei auf der Unterstützung der Frameworks bei der Einhaltung der Merkmale, die einen Web Service RESTful machen. Darüberhinaus werden die allgemeinen Eigenschaften der Frameworks untersucht sowie besondere Kennzeichen, die die Erstellung von Web Services vereinfachen.

Zunächst wird kurz auf den Begriff *Framework* eingegangen. Ein wichtiges Merkmal eines Frameworks ist die *Inversion of Control*. Die Kontrolle umzukehren bedeutet, dass nicht die Anwendung Methoden des Frameworks aufruft, sondern umgekehrt in der Anwendung Methoden implementiert werden, die vom Framework aufgerufen werden. Der Kontrollfluss wird vom Framework gesteuert, die Anwendung implementiert lediglich den individuellen Code. Ein Framework stellt dazu einen Satz wiederverwendbarer oder erweiterbarer Interfaces oder Klassen bereit. Diese werden implementiert bzw. abgeleitet oder die Anwendungsklassen werden in geeigneter Weise an Klassen des Frameworks angemeldet. Das Framework dient somit als Grundlage, eine spezielle Anwendung zu entwickeln. Frameworks abstrahieren häufig tieferliegende Schichten, wodurch die Anwendung effizienter erstellt werden kann. So ist es auch möglich, mit einer API (der des Frameworks) verschiedene Technologien (z.B. verschiedene Übertragungsprotokolle) zu bedienen, ohne deren gekapselten Schnittstellen zu kennen.

4.1. Auswahl der Frameworks

Bevor die Untersuchung beginnt, wird die Auswahl der Frameworks erläutert. Web Services erfreuen sich einer großen Popularität. Entsprechend groß ist die Anzahl an Frameworks, die Entwickler bei deren Erstellung unterstützen sollen. Für diese Arbeit werden lediglich Frameworks für Java betrachtet. Durch die Beschränkung auf eine Programmiersprache lassen sich die Frameworks besser miteinander vergleichen, da Unterschiede nur durch die Frameworks selbst und nicht durch eingesetzte Programmiersprachen auftreten können.

Eine bereits seit 1998 in Java integrierte Möglichkeit Web Services zu erstellen, auch wenn das nicht das ursprüngliche Ziel bei ihrer Einführung war, ist die *Java Servlet-API*. Sie ist leicht zugänglich und wird deshalb in die Auswahl aufgenommen.

Bei der Weiterentwicklung von Java gab und gibt es Tendenzen, die Erstellung von Web Services noch besser zu unterstützen. Im Rahmen des Java Community Process (JCP) werden Java Specification Requests an das von Sun Microsystems betriebene Process Management Office (PMO) gestellt [JP09, 8.]. Ein Java Specification Request (JSR) ist eine Anforderung zur Entwicklung einer neuen Spezifikation oder der signifikanten Änderung einer bestehenden Spezifikation. Der JSR 224 lautet: *Java^{TM}API for XML-Based Web Services (JAX-WS) 2.0* [KG07]. Diese API stellt damit eine weitere, unmittelbar in Java integrierte Möglichkeit zur Erstellung von Web Services dar. Für diesen Vergleich wurde der Kern des Projekts *Metro* betrachtet. Es handelt sich dabei um die Referenz-Implementierung von JAX-WS.

Die beiden genannten APIs sind nicht explizit zur Erstellung von RESTful Web Services gedacht. Deshalb hat SUN 2007 den JSR 311: *JAX-RS: The Java^{TM}API for RESTful Web Services* [HS08] initiiert. Diese REST-Orientierung macht sie zu einem geeigneten Kandidaten für diese Untersuchung. Die Referenz-Implementierung von JAX-RS ist das Open Source Projekt *Jersey*.

Eine weitere Implementierung des JSR 311 ist das Projekt *RESTEasy*. Hierbei handelt es sich um ein Open Source Projekt der JBoss Community. Es bietet viele Erweiterungen, die Jersey nicht bietet. Jose Sandoval betont die Bedeutung des Projekts, indem er RESTEasy als JBoss's Schirm-Projekt bezeichnet [San09, S. 169].

Wegen der fehlenden REST-Unterstützung in Java entstand bereits 2005 die Open-Source-Bibliothek *Restlet*. Sie wurde und wird hauptsächlich von ihrem Gründer und Chef-Entwickler Jérôme Louvel getrieben und setzt auf der Servlet-API auf. Ihre Geschichte und ihre explizite REST-Orientierung macht die API für den Vergleich interessant, weshalb sie mit in diese Arbeit aufgenommen wird.

Als letztes zu betrachtendes Framework wird *Spring* ausgewählt. Von den vielen Frameworks des Spring-Projekts kommen sowohl Spring WS als auch Spring MVC in Frage. Das Spring-Team hat sich entschieden die Einführung von RESTful Web Services von der Entwicklung der SOAP-basierten Web Services in Spring WS zu trennen. So floss REST in Spring MVC ein[1], welches hier untersucht wird.

Es gibt noch weitere interessante Frameworks, die aus verschiedenen Gründen nicht weiter untersucht werden. Einige davon sollen dennoch kurz erwähnt werden.

[1] URI: `http://blog.springsource.com/2009/03/08/rest-in-spring-3-mvc/` (Stand: 18.12.2009)

Struts[2] ist ein umfangreiches Web Framework, welches durch Plugins erweitert werden kann. Seit der Version 2.1.1 wird ein REST Plugin für Struts angeboten, es setzt den Ruby on Rails Stil um. Aufgrund der Komplexität und der Entscheidung, mit Spring MVC bereits ein Web Framework in die Untersuchung aufzunehmen, wurde das Struts Plugin nicht näher betrachtet. Ein weiteres, aus denselben Gründen nicht untersuchtes Web Framework ist *Apache CXF*[3]. Der Schwerpunkt liegt hier deutlich auf klassischen Web Services und SOAP. Es beinhaltet eine Implementierung von JAX-WS 2.2, allerdings ebenso eine Implementierung von JAX-RS 1.1. Auf der Schwelle zum Apache Projekt gibt es eine weitere JAX-RS Implementierung: *Apache Wink*[4]. Dies ist noch Teil des *Incubator*-Projektes[5], in dem sich Projekte beweisen können, um aufgenommen zu werden. Ebenso aufstrebend könnte das leichtgewichtige Projekt *restly*[6] sein. Die Entwicklung scheint allerdings zur Zeit still zu stehen, die aktuellste Version 0.3 ist von Dezember 2008 und noch im Alpha-Stadium.

Die Gliederung der Abschnitte 4.2 bis 4.8 ist für jedes Framework identisch. In einer Einleitung werden Hintergrundinformationen und einige Features des Frameworks aufgeführt. Im Abschnitt *API Implementierung* werden generelle Details zur Implementierung eines Web Services vorgestellt. In den darauffolgenden Abschnitten *Ressourcen und Schnittstelle*, *Statushaltung und Caching* und *Repräsentationen und Verweise* werden die Fragen des im Abschnitt 3.1 ermittelten Katalogs näher beleuchtet und für die jeweilige Implementierung des Fallbeispiels beantwortet. Im Abschnitt *Zusammenfassung und Bewertung* werden Besonderheiten, Stärken und Schwächen noch einmal zusammengefasst. Auch der Vergleich der Frameworks in Kapitel 5 folgt dieser Gliederung.

4.2. Servlet-API 2.5

Ein Servlet ist eine Web-Komponente zur Erzeugung dynamischer Inhalte. Es wird von einem Servlet Container verwaltet, der zumeist in einem Web Server läuft. Der Container kann aber auch selbstständig oder als Teil eines Application Servers eingesetzt werden. Der Web Server reicht einkommende Requests an den Servlet Container weiter. Dieser startet das entsprechende Servlet, welches die Anfrage auswertet und eine passende Antwort erzeugt. Der Servlet Container erzeugt aus der Antwort die HTTP-Response, welche über den Web Server an den Client zurückgeschickt wird

[2] URI: `http://struts.apache.org/2.x/`
[3] URI: `http://cxf.apache.org/`
[4] URI: `http://incubator.apache.org/wink/`
[5] URI: `http://incubator.apache.org/`
[6] URI: `http://code.google.com/p/restly/`

[Mor07, SRV.1.1]. Hauptvorteile von Servlets sind die Plattformunabhängigkeit und die Möglichkeit auf sämtliche Java-Bibliotheken zuzugreifen.

Obwohl die Motivation bei der Einführung von Servlets die dynamische Generierung von Webseiten war, ist es mit Servlets möglich, jegliche Repräsentation an den Client zu schicken. Zur Erstellung dieser Repräsentation kann beliebiger Java-Code ausgeführt werden, wodurch diese für den Einsatz von Web Services prinzipiell gut geeignet sind.

Die Einführung der Servlet-API 1998 sollte zusammen mit *Java Server Pages (JSP)* bei der Erstellung von Servlets unterstützen. Die in dieser Arbeit untersuchte Version 2.5 (September 2005) wurde in JSR 154 [Mor07] spezifiziert.

4.2.1. API Implementierung

Eine Servlet-Klasse wird erstellt, indem das Interface `Servlet` implementiert wird. Dieses ist unabhängig vom eingesetzten Protokoll. Die in der API enthaltene, abstrakte Klasse `HttpServlet` implementiert das Servlet-Interface und stellt eine HTTP-spezifische Schnittstelle bereit. Sie enthält jeweils eine Methode für jedes HTTP-Verb sowie die Methode `service`, in der eine Fallunterscheidung nach aufgerufener HTTP-Methode die jeweilige Servlet-Methode aufruft.

In der Deployment Descriptor Datei `web.xml` werden Servlets URIs zugeordnet. Die Zuordnung kann über verschiedene Wege geschehen. So können Servlets z.B. über die reservierte URI */servlet/<servletname>* bzw. */servlet/<classname>* aufgerufen werden. Die Zuordnung zu einzelnen URIs, dem Anfang eines URI-Pfads oder eines URI-Suffixes ist ebenfalls möglich [Mor07, SRV.11.2]. Hierbei kann das Zeichen `*` als Wildcard verwendet werden. Dieses darf am Ende eines URI-Templates eingesetzt werden, um alle Aufrufe für einen bestimmten URI-Präfix einem Servlet zuzuweisen. Zusätzlich besteht die Möglichkeit, URI-Templates mit einem Suffix (z.B. `*.jsp`) zu definieren. Der Einsatz mehrerer `*` als Wildcard innerhalb eines URI-Templates ist allerdings nicht möglich, wodurch auf die Bildung einer URI-Hierarchie verzichtet werden muss.

Für einfache Web Services, wie die gewählte Beispielanwendung, reicht es, wenn Servlet-Klassen die gewünschten Methoden implementieren. Den Methoden werden zwei Parameter übergeben: `HttpServletRequest` und `HttpServletResponse`. Diese stellen Methoden bereit, mit denen z.B. HTTP-Header, Entity-Body und Statuscode gelesen bzw. geschrieben werden können. Die Servlet-API übernimmt damit zwar das Parsen des Requests, nicht aber das Auswerten der Header und sonstigen Informationen. Die komplette Semantik, wie Request-Header zu deuten sind und welche Response-Header gefüllt werden müssen, bleibt dem Entwickler überlassen.

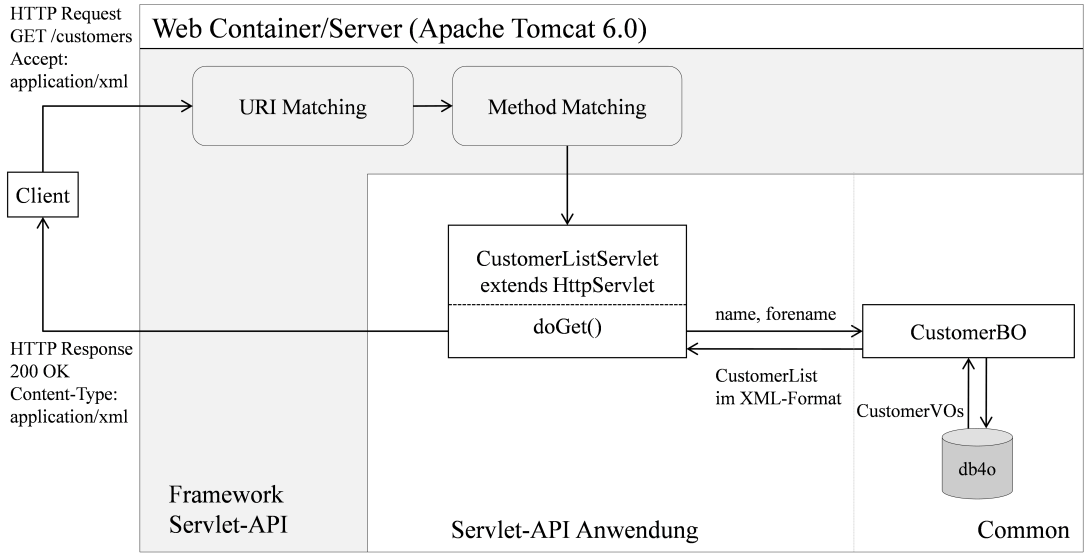


Abbildung 4.1.: Architektur der Realisierung mit der Servlet-API

Die Architektur der Beispiel Web Anwendung für die Umsetzung mit der Servlet-API wird in Abbildung 4.1 aufgezeigt.

4.2.2. Ressourcen und Schnittstelle

Bei der Implementierung des Fallbeispiels wurde für jede der fünf Kern-Ressourcen (siehe Abbildung 3.1) ein Servlet, d.h. eine Java-Klasse erstellt. Die Aufteilung der Ressourcen ist so sehr übersichtlich. Die Servlets leiten von der abstrakten Klasse `HttpServlet` ab. Dadurch steht für jede HTTP-Methode eine Servlet-Methode bereit, die überschrieben werden kann. Der Entwickler kann dementsprechend sämtliche Methoden in ihrer gewünschten Bedeutung umsetzen. Die Anforderung der einheitlichen Schnittstelle ist gewährleistet und wird durch die Bereitstellung der entsprechenden Methoden unterstützt. Anhand der implementierten Methoden erkennt das Framework, welche HTTP-Verben erlaubt sind und beantwortet OPTIONS-Requests automatisch mit entsprechend gesetztem `Allow`-Header. Für Methoden, die nicht im Servlet überschrieben werden, erzeugt das Framework eine Response mit dem Statuscode *405 Method Not Allowed*. Der Header `Allow` wird in diesem Fall allerdings nicht gefüllt, er sollte laut HTTP-Spezifikation die erlaubten Methoden enthalten.
Die Methode für HEAD braucht nicht implementiert zu werden, in diesem Fall wird dieselbe Methode wie bei einem GET-Request aufgerufen und die Response unterdrückt. Dies stellt zwar die korrekte Funktionsweise von HEAD sicher, kann aber unnötig Zeit verbrauchen, wenn die Erzeugung der Entity entsprechend aufwändig ist.

Das Codebeispiel 4.1 zeigt exemplarisch die Implementierung der Methode `doPost` in der Klasse `CustomerListServlet`. Zunächst wird der Header `Content-Type` ausgewertet und überpüft, ob ein unterstütztes Format geliefert wird. Anderenfalls wirft `CheckMediaTypeUtil` eine `UnsupportedMediaTypeException`. Das `BusinessObject` erzeugt anschließend das `ValueObject` aus der vom Client übermittelten Repräsentation. Im Erfolgsfall wird der Statuscode *201 Created* zurückgegeben. Außerdem wird im Header `Location` die URI für die neu erzeugte Ressource gesetzt. Dazu wird wegen der nicht realisierbaren URI-Hierarchie die Request-URI modifiziert, die Endung /customers in /customer geändert und die neue Kunden-ID als Query-Parameter angehängt. Tritt bei der Verarbeitung ein Fehler auf, setzt die Klasse `ExceptionUtil` einen passenden Statuscode und im Response-Body einen Freitext dazu.

```
1
2  public class CustomerListServlet extends HttpServlet {
3
4    ...
5
6    /**
7     * Legt einen neuen Kunden an.
8     */
9    public void doPost(HttpServletRequest request, HttpServletResponse response)
10       throws ServletException, IOException {
11     try {
12       CheckMediaTypeUtil.checkContentTypeHeader(
13           request.getHeader("Content-Type"));
14
15       final CustomerVO customerVO = new CustomerBO(Framework.SERVLETAPI)
16           .create(ServletUtil.readRequestInput(request));
17       response.setStatus(HttpServletResponse.SC_CREATED);
18       final String requestedUrl = request.getRequestURL().toString();
19       response.setHeader("Location", requestedUrl
20           .replace(UriPathParts.CUSTOMERS_HREF, UriPathParts.CUSTOMER_HREF)
21           + "?" + QueryParameter.CUSTID.toParamString(
22               customerVO.getId().toString()));
23     } catch (CommonException e) {
24       ExceptionUtil.handleCommonException(e, response);
25     }
26   }
27
28   ...
29
30 }
```

Listing 4.1: Implementierung der Methode POST der Ressource Kundenliste

Die Zuordnung der URIs zu den Servlets im Deployment Descriptor hat den Nachteil, dass dieser Teil des Services, also die Auswahl des zuständigen Servlets, nicht mit Java-Mitteln, sondern in einer Konfigurationsdatei geschieht. Andererseits ist die URI-Struktur dadurch relativ übersichtlich an einer zentralen Stelle definiert. Wird beispielsweise der Service erweitert, kann so leicht erkannt werden, welche URIs bereits vergeben sind. Wie bereits erwähnt, ist die Verwendung einer hierarchischen URI-Struktur nicht

möglich. Die hierarchische Vergabe der URI-Namen wird zwar empfohlen, ist aber nicht verpflichtend. Eine Möglichkeit, um dennoch eine Hierarchie einzusetzen, kann durch die Verwendung von Servlet-Filtern erreicht werden. Mit Filtern können die Request-URIs verändert und die Requests weitergeleitet werden. Durch die Manipulation von URIs wird die Realisierung jedoch unnötig komplex. Die zusätzlichen Einträge für die Filter machen die Deployment Descriptor Datei unübersichtlicher. Zudem werden URI-Pattern definiert, die kein Client verwendet, sondern die durch Manipulation im Filter entstehen. Außerdem muss in den Servlet-Klassen zwischen den internen URIs, die vom Filter in den Request gesetzt werden, einerseits und externen URIs, die im `Location`-Header und in den Verweisen innerhalb der Repräsentation verwendet werden, andererseits unterschieden werden. Stattdessen wurde die in Abschnitt 3.2.3 beschriebene Lösung gewählt, IDs als Query-Parameter anzugeben.

Die Servlet-API nutzt den Sicherheitsmechanismus, der Servlet Containern durch Konfiguration in der Deployment Descriptor Datei immer zur Verfügung steht. Hierbei kann über URI-Pattern konfiguriert werden, welche Ressourcen geschützt werden sollen und für welche Requests die Einschränkung gilt. Dem geschützten Zugriff wird eine Rolle zugewiesen, die für den Request notwendig ist. Außerdem wird der Login-Mechanismus definiert, also das eingesetzte Authentifizierungsschema und der zugehörige Wert für `realm`. In einer weiteren Datei werden Rollen, Benutzer und die Zuordnung der Rollen zu den Benutzern definiert. Beim eingesetzten Apache Tomcat heißt diese Datei `tomcat-users.xml`. Dieser Mechanismus funktioniert zwar für die Servlet-API gut, kann aber nicht für alle Frameworks eingesetzt werden. Der Grund dafür ist, dass die Konfiguration des eingeschränkten Zugriffs mit Hilfe von URI-Pattern geschieht. Diese erlauben wiederum keine Wildcards inmitten der URI, so dass diese Variante nur für nicht-hierarchische URI-Strukturen eingesetzt werden kann.

Zusätzlich zu dieser generell verfügbaren, sogenannten *Declarative Security* unterstützt die Servlet-API die *Programmatic Security*, die für manche Anwendungen möglicherweise notwendig ist. Hierzu werden im Request Methoden zur Verfügung gestellt, mit denen die hinterlegten Benutzerinformationen abgefragt werden können und anhand derer anschließend überprüft werden kann, ob dem Benutzer eine bestimmte Rolle zugewiesen wurde. Von dieser Möglichkeit wurde für das Fallbeispiel allerdings kein Gebrauch gemacht.

4.2.3. Statushaltung und Caching

Die Servlet-API beinhaltet die Unterstützung von HTTP-Sessions. Die Serverseite darf sich allerdings nicht darauf verlassen, dass der Client seinen nächsten Request innerhalb einer Sitzung sendet. In einer REST-konformen Anwendung gibt es keine Notwendigkeit

für Sitzungen. Jede Information, die zur Bearbeitung eines Requests benötigt wird, ist im Request enthalten. Für die Beispielumsetzung werden dementsprechend keine HTTP-Sessions verwendet, alle Requests erfolgen unabhängig voneinander. Ein weiteres Risiko besteht darin, dass Servletobjekte wiederverwendet werden. Dadurch sind Werte in Klassenattributen, die während einer Requestbearbeitung verwendet werden, für nachfolgende Requests verfügbar. Dies erscheint zunächst hilfreich, verletzt jedoch die Bedingung der Zustandslosigkeit des Servers.

Das Caching-Verhalten kann durch explizites Setzen der entsprechenden Header beeinflusst werden. Die Servlet-API bietet hierbei jedoch keine Hilfe.

Für bedingte GET-Requests bietet die Servlet-API eine zusätzliche Methode `getLastModified` an. Diese erhält als Parameter den Request und erwartet als Rückgabe den Zeitpunkt der letzten Änderung der Ressource, welcher in der Implementierung aus dem `ValueObject` gelesen werden kann. Der zurückgegebene Zeitpunkt wird vom Framework mit dem Header `If-Modified-Since` verglichen, falls dieser im Request gefüllt ist. Ist die Ressource unverändert, wird ohne die Methode `doGet` aufzurufen der Statuscode *304 Not Modified* zurückgegeben. Die zusätzliche Methode wird leider nicht für von GET verschiedene Requests verwendet. Ebensowenig unterstützt die Servlet-API bei dem Vergleich von ETags.

4.2.4. Repräsentationen und Verweise

Die Servlet-Methoden erhalten jeweils den HTTP-Request und die HTTP-Response als Parameter. Der Request wiederum stellt Methoden bereit, mit denen die Header ausgelesen werden können. So kann im Servlet ermittelt werden, welches Repräsentationsformat der Client erwartet. Der Header `Accept` muss allerdings vom Servlet selbst geparsed und ausgewertet werden. Mit Hilfe einer weiteren Methode des Requests kann das Servlet ermitteln, welches Repräsentationsformat der Client übertragen hat. Die Servlet-API bietet darüberhinaus keine Unterstützung bei der Erstellung oder Verarbeitung von Repräsentationen. Dies geschieht unabhängig von der API. Auch bei der Verweiserstellung auf andere Ressourcen in Repräsentationen hilft die API dem Entwickler nicht.

Die Implementierung der Methode `doGet` in der Klasse `CustomerListServlet` wird im Codebeispiel 4.2 aufgezeigt. Zunächst werden die Query-Parameter `name` und `forename` extrahiert. Anschließend wird mit Hilfe der Utility-Klasse `CheckMediaTypeUtil` der `Accept`-Header ausgewertet, um das vom Client erwartete Repräsentationsformat zu ermitteln. Diese wirft auch eine Exception, sollte der Client ein nicht verfügbares Format anfordern. Die Methode liefert dann entweder die Liste der Kunden nach Name und Vorname eingeschränkt im XML- oder JSON-Format mit dem Statuscode *200 OK*

oder, falls der Client ein anderes Format erwartet hatte, den Statuscode *406 Not Acceptable* und im Response-Body einen Freitext dazu. Das gelieferte Format wird im Header `Content-Type` dokumentiert.

```
1
2  public class CustomerListServlet extends HttpServlet {
3
4    ...
5
6    /**
7     * Erstellt eine Kundenliste, filtert diese anhand des Namens
8     * und des Vornames und gibt diese im angeforderten Format zurück.
9     */
10   public void doGet(HttpServletRequest request, HttpServletResponse response)
11       throws ServletException, IOException {
12     try {
13       final PrintWriter out = response.getWriter();
14
15       // Selektionsparameter als Query-String
16       final String name =
17           request.getParameter(QueryParameter.NAME.toString());
18       final String forename =
19           request.getParameter(QueryParameter.FORENAME.toString());
20
21       // Welche Repräsentation? XML und JSON werden unterstützt
22       final ContentType contentType =
23           CheckMediaTypeUtil.checkAcceptHeader(request.getHeader("Accept"));
24       if (ContentType.APPLICATION_XML.equals(contentType)) {
25         out.println(new CustomerBO(Framework.SERVLETAPI).getAllXML(
26             name, forename));
27       } else if (ContentType.JSON.equals(contentType)) {
28         out.println(new CustomerBO(Framework.SERVLETAPI).getAllJSON(
29             name, forename));
30       } else {
31         throw new NotAcceptableException(
32             "Es werden nur XML und JSON unterstützt");
33       }
34
35       response.setStatus(HttpServletResponse.SC_OK);
36       response.setContentType(contentType.toString());
37     } catch (CommonException e) {
38       ExceptionUtil.handleCommonException(e, response);
39     }
40   }
41
42   ...
43
44 }
```

Listing 4.2: Implementierung der Methode GET der Ressource Kundenliste

Die Reaktion auf die unterschiedlichen Repräsentationsformate und ggf. die Verzweigung in der Verarbeitung kann und muss vom Entwickler des Servlets umgesetzt werden. Auch wenn die Verarbeitung nicht explizit durch das Framework unterstützt wird, kann die Umsetzung der Content Negotiation für alle Verben erfolgen.

4.2.5. Zusammenfassung und Bewertung

Die Implementierung von RESTful Web Services ist mit der Servlet-API möglich. URIs und Ressourcen können einander zugeordnet werden. Hierbei muss allerdings auf eine Hierarchie mit mehreren variablen Teilen verzichtet werden. Der Einsatz von Filtern zur Durchsetzung der Hierarchie wird in der Praxis aus den genannten Gründen nur in Ausnahmen geschehen.

Die einheitliche Schnittstelle ist durch eine abstrakte HTTP-Implementierung, die die Methoden bereitstellt, gewährleistet. Verschiedene Repräsentationsformen können verarbeitet und angeboten werden, auch wenn hierfür keine weitere Unterstützung angeboten wird.

Insgesamt gesehen ist die Unterstützungsleistung der Servlet-API eher gering. Viele Aspekte, die für REST wichtig sind, bleiben dem Entwickler überlassen. Es erfolgt keine saubere Trennung zwischen Kommunikations- und eigentlicher Anwendungsschicht. Vielmehr handelt es sich um eine „low-level"-API, die dem Entwickler viele Freiheiten lässt, ihn dadurch aber auch zwingt, viele „Standardaufgaben" selbst zu realisieren.

1. Ressourcen-Implementierung	+	Ressourcen lassen sich gut in Servlets aufteilen, Implementierung eher umständlich, z.B. Auswertung der Header
2. URI-Zuordnung	-	mangelnde Unterstützung von Variablen in URI-Templates
3. Schnittstelle/Methoden	+	alle Methoden sind umsetzbar, abstrakte Klasse bietet sprechende Methoden an, automatisches HEAD und OPTIONS
4. Sicherheit/Authentifizierung	+	leichte Unterstützung erweiterter Sicherheitsabfragen
5. Zustandslosigkeit	-	Servlet-Instanzen werden wiederverwendet
6. Caching	o	keine besondere Unterstützung
7. bedingte Requests	+	bedingtes GET mit Datumsabfrage
8. Repräsentationen	o	keine besondere Unterstützung
9. Content Negotiation	o	keine besondere Unterstützung
10. Verweise auf andere Ressourcen	o	keine besondere Unterstützung

Tabelle 4.1.: Stärken und Schwächen der Servlet-API

4.3. JAX-WS RI 2.1.2

Im Frühjahr 2003 wurde mit der Spezifikation für *JSR 224: Java^{TM}API for XML-Based Web Services (JAX-WS) 2.0* begonnen. Ziel war es, die *JSR 101: Java^{TM}APIs for XML based RPC*[7] so zu erweitern, dass Technologien, die bei der Erstellung dieser Web Services häufig ebenfalls verwendet wurden, besser angebunden werden können. Dies waren unter anderem *Java^{TM}Architecture for XML-Binding (JAXB) 2.0*[8] und *SOAP with Attachments API for Java (SAAJ) 1.3*[9]. JAX-WS ist in Java ab der SE 6 bzw. EE 5 enthalten. Für diese Untersuchung wurde das 2. Maintenance Release der JAX-WS 2.1 (Mai 2007) betrachtet.

Die API enthält sowohl Schnittstellen zur Entwicklung von Web Services als auch von Clients. Betrachtet wurde in dieser Arbeit allerdings nur die API der Serverseite. Die Untersuchung bezieht sich auf die Referenz-Implementierung[10] der API. Diese Referenz-Implementierung liefert den Kern des Metro-Projekts[11] innerhalb der GlassFish Community[12]. JAX-WS enthält außerdem eine Schnittstelle für eine Standalone-Anwendung. Diese wurde hier jedoch nicht verwendet. Die Anwendung lief wie für die anderen Frameworks im Apache Tomcat 6.

4.3.1. API Implementierung

Mit Hilfe von JAX-WS können Web Services auf verschiedene Art und Weise realisiert werden. Eine Möglichkeit ist, die Klasse, die den Web Service bereitstellt, mit der Annotation `@WebService` zu markieren. Ein Client kann dann die Methoden dieser Klasse aufrufen, und bekommt das Ergebnis wie beim Aufruf einer lokalen Methode. Allerdings entspricht diese Art der Implementierung eher RPC als REST.
Eine weitere Möglichkeit ist die Verwendung der Annotation `@WebServiceProvider` und gleichzeitige Implementierung des Interfaces `Provider<T>` mit dessen einziger Methode `invoke`. Der Request wird vom Framework an die `invoke`-Methode durchgereicht, dort verarbeitet und die Response erstellt. Eine Ausnahme hiervon bildet der OPTIONS-Request. Er wird vom Framework beantwortet, wobei als Antwort im Header `Allow` immer GET, HEAD, POST, PUT, DELETE, TRACE, OPTIONS zurückgegeben wird.

[7] URI: `http://jcp.org/en/jsr/detail?id=101`
[8] URI: `http://jcp.org/en/jsr/detail?id=222`
[9] URI: `https://saaj.dev.java.net/`
[10] URI: `https://jax-ws.dev.java.net/`
[11] URI: `https://metro.dev.java.net/`
[12] URI: `https://glassfish.dev.java.net/`

Eine weitere Annotation ist `@BindingType`, mit deren Hilfe man die Bindung[13] festlegen kann. Mit der Annotation `@ServiceMode` wird festgelegt, ob lediglich die Payload oder die gesamte Nachricht verarbeitet werden soll.

JAX-WS nutzt die Common-Annotation `@Resource`. Mit ihr wird ein Klassen-Attribut vom Typ `WebServiceContext` kenntlich gemacht. Bei der Erzeugung der Instanz injiziert JAX-WS den Web Service Kontext. Dieser enthält alle relevanten Informationen zum Request und auch zur Response, wie das Verb, die Header, URI-Pfad und -Query sowie den Statuscode. Die Architektur der Beispiel Web Anwendung für die Umsetzung mit der JAX-WS RI wird in Abbildung 4.2 aufgezeigt.

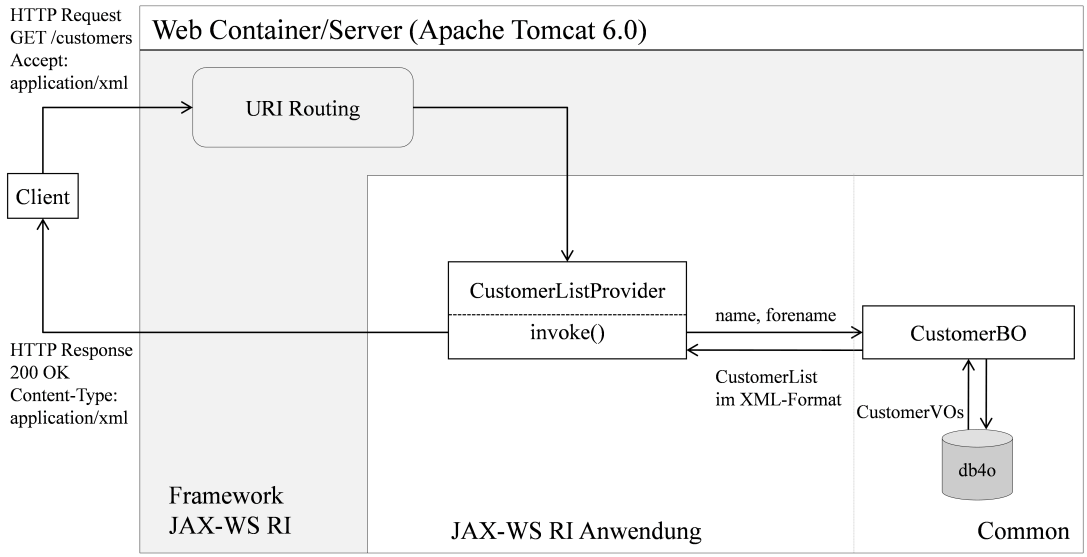


Abbildung 4.2.: Architektur der Realisierung mit der JAX-WS RI

Unabhängig vom Request wird in der Provider-Klasse die Methode `invoke` aufgerufen. Hier wird typischerweise das Verb aus dem Kontext gelesen und jeweils eine passende Methode aufgerufen. Dies ist allerdings kein Zwang, sondern lediglich guter Stil. Für die Unterscheidung der Verben stellt das Framework keine Konstanten bereit. Die Methode `invoke` erhält einen Parameter vom selben Typ, mit der auch das generische Interface `Provider<T>` getypt wird. Derselbe Typ wird auch als Rückgabewert der Methode erwartet. Für das Fallbeispiel wurde der Typ `DataSource` verwendet, welcher den Entity-Body des Requests bzw. der Response enthält. Für diese Typisierung müssen `@BindingType(HTTPBinding.HTTP_BINDING)` und `@ServiceMode(Mode.MESSAGE)` gewählt werden. Der Typ `Source` ist lediglich für XML-Nachrichten geeignet, mit ihm ist die Übertragung der JSON-Repräsentation aus dem Fallbeispiel jedoch nicht

[13] URI: `http://www.w3.org/TR/wsdl20-bindings/`

möglich. Der dritte mögliche Typ `SOAPMessage` ist speziell für SOAP-Nachrichten konzipiert und damit für das Fallbeispiel ebenfalls ungeeignet.

In Fehlersituationen erwartet das Framework, dass der Web Service eine `WebServiceException` wirft. Ein hierbei angebener Meldungstext wird jedoch nicht an den Client weitergegeben. Der Web Service sollte einen entsprechenden Statuscode im Kontext gesetzt haben.

Die Service-Klassen, hier Endpoints genannt, werden nicht in der Deployment Descriptor Datei `web.xml` einzeln zugeordnet. Bei der JAX-WS RI wird lediglich der Servlet-Container `com.sun.xml.ws.transport.http.servlet.WSServlet` einem URI-Pattern zugewiesen. Dieses Pattern stellt so die Wurzel der für den Service verwendeten URIs dar. Stattdessen werden die Endpoint-Klassen in der Datei `sun-jaxws.xml` mit einem URI-Pattern verknüpft. Auch dieses Pattern unterstützt keine Wildcards inmitten der URI, so dass auch für JAX-WS keine Hierarchie für die Ressourcen-URIs gebildet werden kann. Bei der Definition der Endpoints gilt zu beachten, dass nicht der am besten passende Eintrag ermittelt wird, sondern der erste Eintrag zählt, für den das URI-Pattern passt.

4.3.2. Ressourcen und Schnittstelle

Die fünf Kern-Ressourcen wurden jeweils in einer eigenen Endpoint-Klasse mit der Annotation `WebServiceProvider` implementiert. Die Methode `invoke` ermittelt das HTTP-Verb und verzweigt dann in die entsprechende Methode oder gibt als Antwort *405 Method Not Allowed* an den Client, wenn der Request ein ungültiges Verb enthält. Der Header `Allow` bleibt hierbei unbelegt. Die Verzweigung ist nicht sehr komfortabel, da es vom Framework keine Unterstützung bzgl. Methodennamen oder ähnlichem gibt. Andererseits macht es diese Offenheit in der Umsetzung leichter, eigene Verben einzuführen und zu verwenden. Von dieser Möglichkeit ist in Bezug auf REST allerdings eher sparsam Gebrauch zu machen, da hierunter die Einheitlichkeit der Schnittstelle leidet.

Das folgende Codebeispiel 4.3 zeigt die Implementierung der Methode `post` in der Klasse `CustomerListProvider`. Der 1. Parameter der Methode ist `source`, welcher den Entity-Body des Requests enthält. Ein weiterer Parameter ist der Kontext `mc`. Nachdem der Header `Content-Type` überprüft wurde und der Client tatsächlich ein unterstütztes Format gesendet hat, erzeugt das `BusinessObject` aus der vom Client übermittelten Repräsentation das `ValueObject`. Im Header `Location` wird die URI für die neu erzeugte Ressource gesetzt. Wegen der nicht realisierbaren URI-Hierarchie wird wiederum die Request-URI modifiziert und die neue Kunden-ID als Query-Parameter

angehängt. Der Statuscode *201 Created* wird ebenfalls über den Kontext in die Response eingetragen. Tritt ein Fehler auf, setzt die Klasse `ExceptionUtil` den entsprechenden Statuscode im Kontext und gibt eine Repräsentation zurück, die einen passenden Fehlertext enthält.

```
1
2  @WebServiceProvider
3  @BindingType(value = HTTPBinding.HTTP_BINDING)
4  @ServiceMode(value = Mode.MESSAGE)
5  public class CustomerListProvider implements Provider<DataSource> {
6
7    ...
8
9    /**
10    * Legt einen neuen Kunden an.
11    */
12   private DataSource post(DataSource source, MessageContext mc) {
13
14     try {
15       CheckMediaTypeUtil.checkContentTypeHeader(
16           HeaderUtil.getRequestHeader(mc, "Content-Type"));
17
18       final CustomerVO customerVO = new CustomerBO(Framework.JSR224).create(
19           StreamSourceUtil.readRequestInput(source));
20
21       // URI des neu angelegten Kunden in den Header schreiben
22       final HttpServletRequest request =
23           (HttpServletRequest) mc.get(MessageContext.SERVLET_REQUEST);
24       final String requestedUrl = request.getRequestURL().toString();
25       HeaderUtil.setResponseHeader(mc, "Location", requestedUrl
26           .replace(UriPathParts.CUSTOMERS_HREF, UriPathParts.CUSTOMER_HREF)
27           + "?" + QueryParameter.CUSTID.toParamString(
28               customerVO.getId().toString()));
29
30       mc.put(MessageContext.HTTP_RESPONSE_CODE, 201);
31       return new ByteArrayDataSource("".getBytes(), "text/plain");
32     } catch (CommonException e) {
33       return ExceptionUtil.handleCommonException(e, mc);
34     }
35   }
36
37   ...
38
39 }
```

Listing 4.3: Implementierung der Methode POST der Ressource Kundenliste

Für die Authentifizierung wurde wie bei der Servlet-API der über den Deployment Descriptor verfügbare Mechanismus eingesetzt. Darüber hinaus ist bei JAX-WS ebenso analog zur Servlet-API eine programmatische Sicherheitsabfrage möglich. Der injizierte Kontext stellt Methoden zur Verfügung gestellt, mit denen die Benutzerinformationen und Rollenzugehörigkeit abgefragt werden können. Jedoch wurde auch hier von dieser Möglichkeit kein Gebrauch gemacht.

4.3.3. Statushaltung und Caching

JAX-WS unterstützt eine Statusverwaltung für die SOAP- und die HTTP-Bindung. Der Einsatz dieser Möglichkeit wäre eine Verletzung des REST-Prinzips der Zustandslosigkeit. Wie bei der Servlet-API werden außerdem Ressourcenobjekte wiederverwendet. Klassenattribute sind also nicht automatisch bei jedem Request neu initialisiert, sondern behalten ihre Werte für nachfolgende Requests.

Um das Caching-Verhalten zu beeinflussen, müssen die entsprechenden Header-Informationen manuell gesetzt werden. Eine Unterstützung vom Framework diesbezüglich gibt es nicht.

Ebensowenig hilft JAX-WS bei bedingten Requests. ETag, Änderungsdatum oder beides müssen in der Providerklasse selbst ermittelt, verglichen und gesetzt werden.

4.3.4. Repräsentationen und Verweise

Aus dem Namen der API *Java™API for XML-Based Web Services* geht bereits hervor, dass die bevorzugte Repräsentationsform XML ist. So war XML auch die erste Repräsentationsform, die im Fallbeispiel umgesetzt wurde. Bei der Typisierung der Interfaces `Provider<T>` gibt es laut Spezifikation drei Möglichkeiten, welche bereits im Abschnitt 4.3.1 erwähnt wurden. Das Interface `Source` ist für XML gut geeignet. Es gibt einige Implementierungen, welche unter anderem die Verwendung von JAXB-Objekten unterstützen. So ließ sich eine bequeme Realisierung die Ressourcenklassen, zumindest bezüglich der Erzeugung der Repräsentation, für XML leicht umsetzen. Für die Angabe von MIME-Typen, die weder *text/xml* noch *application/xml* sind, kann diese Typisierung jedoch nicht eingesetzt werden.

Die bereits beschriebene Typisierung `DataSource` für die HTTP-Bindung im Message-Modus eignet sich für die Verarbeitung beliebiger Daten. Das Framework liefert hierfür allerdings keinerlei Unterstützung. Die Auswertung der Header für die Content Negotiation, das Setzen von Headern in der Response, das Lesen und Interpretieren sowie das Schreiben der Entity-Bodies muss explizit implementiert werden. Hierbei ist das Lesen und Schreiben von Headern relativ umständlich, da hierfür lediglich das `MessageContext`-Objekt zur Verfügung steht, welches nur eine rudimentäre Schnittstelle dafür anbietet. Aus diesem Grund wurden diese Schritte in Utility-Klassen ausgelagert.

Die Implementierung der Methode `get` in der Klasse `CustomerListProvider` wird im Codebeispiel 4.4 aufgezeigt. Da hierfür kein Request-Body benötigt wird, ist der Kontext `mc` der einzige Parameter. Zuerst werden die Query-Parameter `name` und

`forename` ermittelt, nach denen die Liste gefiltert wird. Wie bei der Servlet-API-Umsetzung wertet die Utility-Klasse `CheckMediaTypeUtil` den `Accept`-Header aus, um das vom Client erwartete Repräsentationsformat zu ermitteln. Es wird dann die Liste der Kunden nach Name und Vorname eingeschränkt entweder im XML- oder JSON-Format erzeugt. Anschließend wird der Statuscode *200 OK* gesetzt und das Ergebnis als `ByteArrayDataSource` zurückgegeben. Diese Implementierung von `DataSource` kann beliebige Bytefelder aufnehmen. Dem Konstruktor wird das MIME-Format mitgeteilt, welches dann vom Framework in den `Content-Type`-Header übernommen wird.

```
1
2  @WebServiceProvider
3  @BindingType(value = HTTPBinding.HTTP_BINDING)
4  @ServiceMode(value = Mode.MESSAGE)
5  public class CustomerListProvider implements Provider<DataSource> {
6
7    ...
8
9    /**
10    * Erstellt eine Kundenliste, filtert diese anhand des Namens
11    * und des Vornames und gibt diese im angeforderten Format zurück.
12    */
13   private DataSource get(MessageContext mc) {
14
15     try {
16       // Selektionsparameter als Query-String
17       final String query = (String) mc.get(MessageContext.QUERY_STRING);
18       final String name = QueryParser.parseQuery(query, QueryParameter.NAME);
19       final String forename =
20           QueryParser.parseQuery(query, QueryParameter.FORENAME);
21
22       // Welche Repräsentation? XML und JSON werden unterstützt
23       final ContentType contentType = CheckMediaTypeUtil.checkAcceptHeader(
24           HeaderUtil.getRequestHeader(mc, "Accept"));
25       String result = "";
26       if (ContentType.APPLICATION_XML.equals(contentType)) {
27         result = new CustomerBO(Framework.JSR224).getAllXML(name, forename);
28       } else if (ContentType.JSON.equals(contentType)) {
29         result = new CustomerBO(Framework.JSR224).getAllJSON(name, forename);
30       } else {
31         throw new NotAcceptableException(
32             "Es werden nur XML und JSON unterstützt");
33       }
34
35       mc.put(MessageContext.HTTP_RESPONSE_CODE, 200);
36       return new ByteArrayDataSource(result.getBytes(),
37           contentType.toString());
38     } catch (CommonException e) {
39       return ExceptionUtil.handleCommonException(e, mc);
40     }
41   }
42
43   ...
44
45 }
```

Listing 4.4: Implementierung der Methode GET der Ressource Kundenliste

Eine Variante ist die Verwendung der JSON-Bindung mittels

`@BindingType(JSONBindingID.JSON_BINDING)`. Dadurch ist allerdings die Rückgabe einer XML-Repräsentation in dieser Ressourcenklasse nicht mehr möglich. Beim Einsatz von Content Negotiation durch Dateiendungen wäre es möglich, zwei Sätze Ressourcenklassen zu implementieren, einen für XML und einen für JSON. Ob dieser Ansatz trägt, wurde für das Fallbeispiel nicht verifiziert.

Eine weitere Möglichkeit ist, XML-Strukturen in Fast Infoset[14] umzuwandeln. Fast Infoset Dokumente sind eine Alternative zu großen XML-Dokumenten, die viel Zeit zum Erstellen oder Interpretieren benötigen. Sie sind inhaltlich äquivalent, aber kleiner und schneller zu serialisieren sowie zu parsen, wodurch die Menge der Übertragungsdaten und damit die Antwortzeit reduziert wird.

4.3.5. Zusammenfassung und Bewertung

Insgesamt gesehen, ist JAX-WS eine stark auf XML fokussierte API und für die Erstellung „klassischer“ SOAP-Web Services geeigneter als für REST-Web Services. Durch den Einsatz als „low-level“-API, d.h. der Verarbeitung der gesamten Nachricht auf Protokollebene, und der händischen Implementierung der REST-Aspekte war es trotzdem möglich, das Fallbeispiel wunschgemäß umzusetzen. Es gibt eine Provider-Klasse pro Ressource. Diese sähe möglicherweise anders aus, wäre der in Abschnitt 4.3.4 angesprochene Ansatz weiterverfolgt worden, je nach Bindung eine Ressourcenklasse zu erstellen. In diesem Fall wären zwei Klassen je Ressource nötig gewesen, oder sogar drei, wenn identische Teile wiederum in eine gemeinsam genutzte Klasse ausgelagert würden. Dies erscheint sinnvoll, um sicherzustellen, dass Header, Statuscodes etc. unabhängig vom Repräsentationsformat einheitlich gehandhabt werden. Die Verwendung hierarchischer URIs ist nicht möglich, da das URI-Pattern Wildcards nicht an beliebigen Stellen in der URI erlaubt.

Die Einheitlichkeit der Schnittstelle kann gewährleistet werden, wird aber vom Framework nur eingeschränkt unterstützt. Die Übertragung der Repräsentation wird für HEAD automatisch unterdrückt, die immer automatisch generierte Antwort für OPTIONS hingegen enthält stets sämtliche Methoden.

Auch wenn der Kontext Instanzen der Ressourcenklassen für jeden Request neu injiziert wird, werden die Objekte für weitere Requests wiederverwendet. Der Entwickler muss also Sorgfalt walten lassen, um die Zustandslosigkeit nicht zu verletzen. Es werden zusätzlich vom Framework Mittel angeboten, Ressourcenklassen explizit zustandsbehaftet werden zu lassen.

[14] URI: `https://metro.dev.java.net/1.4/guide/FastInfoset.html` (Stand: 28.02.2010)

Content Negotiation und von XML verschiedene Repräsentationen werden nur eingeschränkt unterstützt, mit erhöhtem Programmieraufwand können jedoch mehrere Formate vom Web Service angeboten werden.

1. Ressourcen-Implementierung	+	durch „low-level“-Nutzung und Einschränkung auf eine Bindung ist Aufteilung der Ressourcen möglich
2. URI-Zuordnung	-	mangelnde Unterstützung von Variablen in URI-Templates
3. Schnittstelle/Methoden	+	Schnittstelle bietet eine zentrale Methode, alle HTTP-Verben sind umsetzbar, ressourcenunabhängige Standardantwort für OPTIONS
4. Sicherheit/Authentifizierung	+	leichte Unterstützung erweiterter Sicherheitsabfragen
5. Zustandslosigkeit	-	Provider-Instanzen werden wiederverwendet
6. Caching	o	keine besondere Unterstützung
7. bedingte Requests	o	keine besondere Unterstützung
8. Repräsentationen	o	unflexibel, Unterstützung nur bei klassenbezogener Typ-Bindung
9. Content Negotiation	o	keine besondere Unterstützung
10. Verweise auf andere Ressourcen	o	keine besondere Unterstützung

Tabelle 4.2.: Stärken und Schwächen von JAX-WS

4.4. Jersey 1.0.3.1

Sun ruft im Februar 2007 den JSR 311 *JAX-RS: The Java^{TM} API for RESTful Web Services* [HS08] ins Leben. Diese Spezifikation beschreibt eine API, die explizit REST-orientiert ist. Die finale Version 1.0 wurde im Oktober 2008 zur Verfügung gestellt.

Stefan Tilkov fasst in [Til09, Anhang C.4.1] die Ziele von JAX-RS zusammen. Hauptziel ist die REST-Konformität. Aber auch eine einfache Entwicklung und „das Ausnutzen moderner Java-Sprachmittel (insbesondere Annotations)“ [Til09, S. 208] gehören zu den Zielen. JAX-RS abstrahiert nicht das zugrunde liegende Transportprotokoll. Es ist explizit auf HTTP ausgerichtet. Im Wesentlichen vermittelt das Framework zwischen den HTTP-Requests und den zugehörigen Methoden der Ressourcenklassen.

JAX-RS betrachtet bewusst nicht die Clientseite sondern nur die Serverseite der Anwendung. Diese Anwendungen können sowohl in beliebigen Servlet-Containern als auch beispielsweise in einer Java-SE-Umgebung oder einer ganz anderen Umgebung betrieben werden [HS08, Kapitel 6]. Dies ist von der jeweiligen Implementierung der API

abhängig. In diesem Kapitel wird die Referenz-Implementierung der API von SUN *Jersey*[15] in der Version 1.0.3.1 betrachtet.

Über die Spezifikation hinausgehende Features der Jersey-Implementierung sind zusätzliche Provider für einige Repräsentationsformen. Für den Betrieb des Services außerhalb eines Servlet Containers können ein zur Verfügung gestellter „leichtgewichtiger" HTTP Server, der Simple[16] HTTP Web Server oder der Grizzly[17] HTTP Web Server verwendet werden. Ebenfalls zum Jersey-Framework gehört eine Client-API, ein spezielles Test Framework und die Anbindung von Spring.

4.4.1. API Implementierung

Um eine Ressourcenklasse zu implementieren, wird weder ein Interface implementiert noch eine abstrakte Klasse erweitert. Stattdessen erfolgt die Entwicklung in POJOs[18]. Ressourcenklassen werden über Annotationen als solche gekennzeichnet. Das Jersey-Framework kann so konfiguriert werden, dass es automatisch nach Klassen scannt, die bestimmte JAX-RS-Annotationen wie `@Path` enthalten. Es müssen also nicht sämtliche Klassen in der Deployment Descriptor Datei `web.xml` explizit zugeordnet werden. Wahlweise kann aber auch eine explizite Zuordnung in einer durch die Annotation `@Application` markierten Klasse erfolgen.
Für den Betrieb innerhalb eines Servlet Containers wird lediglich die Klasse `com.sun.jersey.spi.container.servlet.ServletContainer` einem URI-Pattern zugewiesen. Dieses Pattern stellt so die Wurzel einer Hierarchie von URIs dar.

Mittels Annotationen können die Requests den entsprechenden Methoden zugeordnet werden. Die bereits erwähnte Annotation `@Path` kann sowohl für die Klasse als auch für jede Methode gesetzt werden. Sie enthält ein URI-Pattern für die Ressource. Dabei gilt die Klassen-Annotation als Wurzel für das ggf. an einer Methode gesetzte Pattern. Diese URI-Pattern dürfen Template-Variablen enthalten, z.B. `{id}` in `/customers/{id}`. Mit Hilfe der Annotation `@PathParam` werden diese Template-Variablen extrahiert und können einem Klassen-Attribut, einem Methoden-Parameter oder einem Bean-Property zugewiesen werden. Der Parameter muss nicht vom Typ `String` sein, hier kann beispielsweise auch ein `Integer` verwendet werden. Das Framework übernimmmt dann die Typumwandlung. Tritt hierbei ein Fehler auf, erhält der Client die Antwort *404 Not Found*.
Ähnliche Annotationen sind `@QueryParam`, `@FormParam`, `@MatrixParam`, `@CookieParam` und `@HeaderParam`, mit denen Werte aus der URI, dem Entity-Body des Requests, aus

[15] URI: `https://jersey.dev.java.net/`
[16] URI: `http://www.simpleframework.org/`
[17] URI: `https://grizzly.dev.java.net/`
[18] Plain Old Java Object

HTTP-Headern oder Cookies extrahiert werden können. Für den Fall, dass Werte nicht gefüllt sind, kann mit der Annotation `@DefaultValue` ein entsprechender Standard-Wert definiert werden. Weitere Informationen zur URI, zu den Headern oder auch zu Authentifizierungsinformationen können mit der Annotation `@Context` erhalten werden.

Der Entity-Body des Requests kann einer Methode zur Verfügung gestellt werden, indem ein Parameter ohne Annotation übergeben wird. Dieser Parameter heißt dann Entity-Parameter. Für Methoden, die ein HTTP-Verb implementieren, ist nur *ein* solcher Entity-Parameter erlaubt.

Die Kennzeichnung, welches HTTP-Verb mit dieser Methode der Ressourcenklasse implementiert wird, erfolgt mit den Annotationen `@GET`, `@POST`, `@PUT`, `@DELETE` und `@HEAD`.[19] Der Name der Java-Methode ist dabei beliebig. In einer Klasse können mehrere Methoden pro HTTP-Verb implementiert werden. Beispielsweise ist es denkbar, verschiedene Repräsentationsformen einer Ressource unterschiedlichen URI-Endungen zuzuordnen. Dann würden mehrere GET-Methoden implementiert, die sich jeweils im Wert der Annotation `@Path` unterscheiden. Eine andere Möglichkeit der Content Negotiation besteht in zwei weiteren Annotationen: `@Consumes` und `@Produces` beschreiben, welche Medientypen diese Methode verarbeiten bzw. zurückgeben kann.

Die Spezifikation erlaubt es, dass Ressource-Methoden verschiedene Return-Typen liefern können. Dies kann `void`, die im Framework enthaltenen Typen `Response` sowie `GenericEntity` oder jeder beliebige andere Typ sein. Für die verschiedenen Fälle ist in [HS08, Kapitel 3.3.3] aufgeführt, was an den Client zurückgeliefert wird. Zusätzlich fängt das Framework Instanzen bestimmter Exceptions, wie die `WebApplicationException`, und erzeugt daraus eine Response.

Die Methoden für HEAD und OPTIONS müssen nicht unbedingt implementiert werden, sie werden auf besondere Weise unterstützt. Existiert keine Implementierung von HEAD wird die passende GET-Methode aufgerufen und anschließend die Entity der Response entfernt. Besteht das Risiko, dass die Response sehr groß wird, kann für HEAD eine eigene Methode erstellt werden. Wenn OPTIONS nicht implementiert ist, wird bei einem solchen Request automatisch eine Response erzeugt. Hierbei werden in der jeweiligen Klasse die Metadaten verwendet, die durch die Annotationen gesetzt sind, um die Response zu generieren. Der Response-Body eines OPTIONS-Requests enthält eine Beschreibung der Ressource im WADL-Format. Eine vollständige WADL-Beschreibung kann durch einen GET-Request auf die Ressource /application.wadl abgerufen werden. Für das Fallbeispiel ist die durch Jersey generierte WADL-Beschreibung in Anhang C aufgelistet.

[19] Diese Aufzählung wird mit JAX-RS 1.1 um *@OPTIONS* ergänzt.

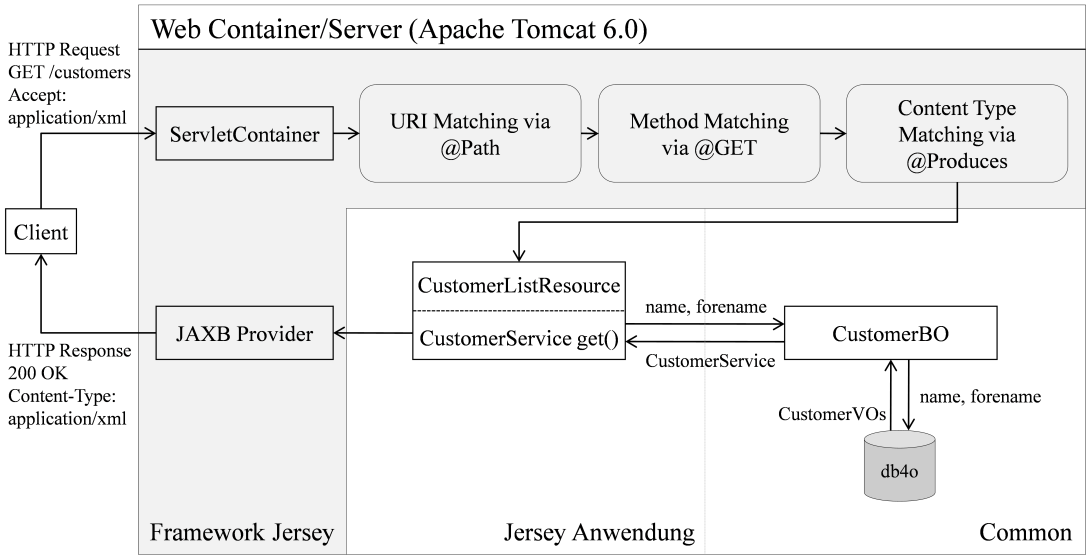


Abbildung 4.3.: Architektur der Realisierung mit Jersey

Die Architektur der Beispiel Web Anwendung für die Umsetzung mit Jersey wird in Abbildung 4.3 aufgezeigt.

JAX-RS stellt noch zwei weitere Annotationen zur Verfügung. Mittels `@HttpMethod` können weitere HTTP-Methoden definiert werden. Selbst erstellte Entity-Provider müssen mit der Annotation `@Provider` gekennzeichnet werden und eines der JAX-RS-Erweiterungs-Interfaces `MessageBodyWriter` oder `MessageBodyReader` implementieren. Dadurch kann das Marshalling und Unmarshalling für weitere Repräsentationen realisiert und integriert werden. Die genauen Beschreibungen der hier beschriebenen Annotationen können [HS08, Kapitel 3 sowie Anhang A] entnommen werden.

4.4.2. Ressourcen und Schnittstelle

Jede der fünf Kern-Ressourcen wurde in einer eigenen Ressourcenklasse realisiert. In ihnen sind jeweils die durch die Beispielanwendung vorgegebenen Methoden implementiert. Da die Spezifikation explizit für das Protokoll HTTP geschrieben wurde, ist die korrekte und intuitive Verwendbarkeit der Schnittstelle sichergestellt. Für HTTP-Verben mit mehreren Repräsentationsformaten können mehrere Methoden implementiert werden. Die Unterscheidung erfolgt im Request durch den Header `Accept` und in der Umsetzung durch die Annotation `@Produces`. Es besteht auch die Möglichkeit, mit einer Methode verschiedene Repräsentationsformate bereitzustellen. Wie dies geschieht, wird im Abschnitt 4.4.4 anhand der Umsetzung der Methode GET gezeigt. Fehlt die

Implementierung eines HTTP-Verbs für eine Ressource, erzeugt Jersey eine Response mit dem Statuscode *405 Method Not Allowed*.

```
1
2  @Path(UriPathParts.CUSTOMERS_HREF)     // <== "'/customers"'
3  @Consumes(MediaType.APPLICATION_XML)  // <== "'application/xml"'
4  public class CustomerListResource {
5
6    ...
7
8    /**
9     * Legt einen neuen Kunden an.
10    */
11   @POST
12   public Response post(CustomerService customerService)
13       throws WebApplicationException {
14     try {
15       final CustomerVO customerVO = new CustomerBO(Framework.JSR311).create(
16           customerService.getCustomer());
17       return Response.created(URI.create("/" + customerVO.getId())).build();
18     } catch (CommonException e) {
19       throw ExceptionUtil.handleCommonException(e);
20     }
21   }
22
23   ...
24
25 }
```

Listing 4.5: Implementierung der Methode POST der Ressource Kundenliste

Die Implementierung der Methode `post` in der Klasse `CustomerListResource` wird im Codebeispiel 4.5 aufgezeigt. Der einzige Parameter der Methode ist der Entity-Parameter ohne Annotation. Das Framework hat das Unmarshalling in das JAXB-Element vom Typ `CustomerService` automatisch durchgeführt. Die Methode liefert den Return-Typ `Response`. In der Beispielanwendung ist dies der gewählte Return-Typ für die Methoden POST, PUT und DELETE. Im Erfolgsfall wird durch Aufruf der Methode `created` in der `Response` der Statuscode *201 Created* gesetzt. Die Methode erhält als Parameter eine relative URI, die die ID des neu erzeugten Kunden enthält. Diese relative URI wird an die URI der aufgerufenen Ressource angehängt. Dies spiegelt wider, dass mit POST Subressourcen angelegt werden. Die URI für die neu erzeugte Ressource wird in den Response-Header `Location` gesetzt. Die Methode `post` selbst hat keine Annotation `@Consumes`, sondern erbt diese von der Definition am Klassenstatement. Dies ist dadurch die erwartete Standard-Repräsentation, wenn an einer Methode nichts anderes definiert ist. Tritt beim Anlegen des Kunden ein Fehler auf, erzeugt die Klasse `ExceptionUtil` eine `WebApplicationException` mit einem passenden Statuscode und im Response-Body einen Freitext dazu.

Da im Beispiel keine Content Negotiation durch URI-Pfad-Endungen stattfindet und jede Ressource in einer eigenen Klasse implementiert wurde, ist die Verwendung der An-

notation `@Path` an Methoden nicht notwendig. Sie wird hier nur als Klassen-Annotation eingesetzt und ordnet jeder Klasse ein URI-Pattern zu. Darin enthaltene Template-Variablen werden Klassen-Attributen zugewiesen. Dadurch stehen diese allen Methoden zur Verfügung. Die Umsetzung von URI-Templates mit mehreren Variablen ist problemlos möglich, was die Verwendung von hierarchischen URIs erlaubt. Optional können die Templates auch reguläre Ausdrücke enthalten.

Im Abschnitt 4.2.2 wurde ein Sicherheitsmechanismus vorgestellt, der beim Betrieb in einem Servlet Container generell zur Verfügung steht, aber nicht unbedingt für hierarchische URI-Strukturen verwendet werden kann. Dies ist abhängig davon, welche Ressourcen geschützt werden sollen. Im Fallbeispiel soll lediglich die Ressource Kundenliste, welche über die URI /customers identifiziert wird, gesichert werden. Hierfür sind keine Wildcards notwendig, insbesondere nicht mitten im URI-Pattern. Sollten beispielsweise sämtliche Ressourcen, die Kunden betreffen, geschützt werden, so wäre dies durch Angabe des URI-Patterns /customers/* ebenfalls mit diesem Mechanismus möglich. Dennoch kann es gewünscht sein, die Authentifizierung detaillierter und je Ressource verschieden festzulegen.

In JAX-RS kann die Annotation `@Context` verwendet werden, um eine Instanz der Klasse `SecurityContext` in eine Ressourcenmethode zu injizieren. Diese bietet Methoden zur Abfrage von Benutzerinformationen, Rollenzugehörigkeit, verwendetes Authentifizierungsschema und Einsatz eines sicheren Protokolls. Eine feinere Gliederung mit hierarchischen URI-Strukturen ist mit Jersey jedoch nicht möglich, da die Konfiguration in der Deployment Descriptor Datei Voraussetzung ist, damit der `SecurityContext` die gewünschten Informationen enthält. Im Jersey User-Guide werden diese Informationen dazu verwendet, verschiedene Repräsentationen für Kunden in unterschiedlichen Rollen zu erzeugen[20].

4.4.3. Statushaltung und Caching

Der Lebenszyklus der Ressourcenklassen unterstützt die Forderung nach Zustandslosigkeit des Servers. Eine Ressourcenklasse wird instanziiert, wenn ihre `@Path`-Definition mit der URI des Requests übereinstimmt. Der exakte Algorithmus, mit dem Requests den Methoden der Ressourcenklassen zugeordnet werden, ist der Spezifikation [HS08, Kapitel 3.7] zu entnehmen. Nach dem Aufruf der Methode wird die Klasse für den Garbage collector freigegeben. Dadurch können die Klassen selbst keinen Zustand des Clients requestübergreifend halten. Werden solche Informationen für den nächsten Request benötigt, muss der Client diese Information wieder mitliefern. Dies ist ganz im

[20] URI: `https://jersey.dev.java.net/nonav/documentation/latest/user-guide.html#d4e466` (Stand: 28.02.2010)

Sinn von REST.
Jersey bietet die Möglichkeit, Ressourcenklassen als Singleton zu definieren. Damit soll die Zeit für das Instanziieren und Zerstören der Objekte gespart werden. Hierbei sollte große Sorgfalt walten, da dies eine Fehlerquelle in Bezug auf die Zustandslosigkeit sein kann.

Um das Caching-Verhalten etwas komfortabler beeinflussen zu können, bietet JAX-RS die Klasse `CacheControl` an, die entsprechende Methoden bereitstellt. Mit der Methode `cacheControl` wird eine solche Instanz an die Klasse `ResponseBuilder` übergeben, die zum Erstellen einer `Response` verwendet kann.

Ebenfalls Unterstützung wird für bedingte Requests geboten. Das Request-Objekt bietet Methoden namens `evaluatePreconditions` an, in die das letzte Änderungsdatum, ein ETag oder beides hineingereicht werden kann und die dann vom Framework gegen die Informationen geprüft werden, die vom Client in den Bedingungs-Headern angegeben wurden. Das Datum und das ETag muss die Anwendung selber erzeugen bzw. bereitstellen, das Framework übernimmt lediglich die Überprüfung und je nach Ergebnis das Erzeugen einer Response.

4.4.4. Repräsentationen und Verweise

Durch die Annotationen `@Consumes` und `@Produces` wird für jede Methode festgelegt, welche Repräsentationsformate sie verarbeitet bzw. erzeugt. Innerhalb einer Methode muss also keine Auswertung der entsprechenden Header geschehen, dies übernimmt das Framework bei der Auswahl der aufzurufenden Methode. Auch eine Verzweigung je nach Repräsentation ist nicht notwendig. Dadurch sind die Methoden übersichtlicher. Die API bietet durch angebotene Provider Hilfe bei der Erstellung oder Verarbeitung von Repräsentationen. Bei der Verweiserstellung auf andere Ressourcen in Repräsentationen hilft die API dem Entwickler hingegen nicht.

Die Realisierung der GET-Methode für die Repräsentationsformate *application/xml* und *application/json* zeigt das Codebeispiel 4.6. Es enthält die Methode `get` der Klasse `CustomerListResource`. Durch die Annotation `@GET` kann das Framework sie diesen HTTP-Requests zuordnen. Die beiden Query-Parameter `name` und `forename` werden vom Framework extrahiert und der Methode als Parameter übergeben. Die Annotation `Produces` enthält in einer Liste die zur Verfügung stehenden Formate. Erwartet der Client ein anderes Repräsentationsformat, erhält er vom Framework die Antwort *406 Not Acceptable*. Das `BusinessObject` erzeugt die Liste der Kunden nach Name und Vorname eingeschränkt und gibt die ermittelten Daten in einem Objekt des Types `CustomerService` zurück. Durch entsprechende Provider, die Jersey bereitstellt,

werden solche für JAXB gekennzeichneten Klassen automatisch in eine Repräsentation umgewandelt. Jersey kann das Marshalling aus diesen JAXB-Ressourcen sowohl in XML als auch in JSON durchführen. Dadurch wird von Jersey automatisch die angeforderte Repräsentation erzeugt. Der Statuscode lautet bei erfolgreicher Erstellung der Repräsentation *200 OK*.

```
1
2  @Path(UriPathParts.CUSTOMERS_HREF)
3  @Consumes(MediaType.APPLICATION_XML)
4  public class CustomerListResource {
5
6    ...
7
8    /**
9     * Erstellt eine Kundenliste, filtert diese anhand des Namens
10    * und des Vornames und gibt diese zurück.
11    */
12   @GET
13   @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
14   public CustomerService get(@QueryParam("name") String name,
15                              @QueryParam("forename") String forename)
16       throws WebApplicationException {
17     try {
18       return new CustomerBO(Framework.JSR311).getCustomerServiceByQueryParams(
19           name, forename);
20     } catch (CommonException e) {
21       throw ExceptionUtil.handleCommonException(e);
22     }
23   }
24
25   ...
26
27 }
```

Listing 4.6: Implementierung der Methode GET der Ressource Kundenliste

Die Spezifikation des JSR 311 sieht vor, dass einige Standard-Provider bereitgestellt werden müssen. Werden eigene Repräsentationsformate benötigt, ist es möglich, passende Provider dafür zu entwickeln, wie bereits in Abschnitt 4.4.1 erwähnt wurde. Jersey enthält Provider für XML, JSON, Fast Infoset, Atom[21] und Multipart. Details zur Erstellung von Providern und bereitgestellten Standard-Providern sind in [HS08, Kapitel 4.2] zu finden.

4.4.5. Zusammenfassung und Bewertung

Die Spezifikation JSR 311 und ihre Referenz-Implementierung Jersey liefern eine solide Grundlage für die Erstellung von Web Services nach den Prinzipien von REST. Die Implementierung der Ressourcen und Zuordnung der URIs ist leicht verständlich

[21] URI: `http://www.atomenabled.org/`

und unproblematisch. Auch die Abbildung der Ressourcen-Hierarchie in entsprechenden URIs ist durch die Angabe beliebiger Templates gut möglich.

Es wird explizit nur das Protokoll HTTP unterstützt. Dessen Verben können intuitiv gemäß ihrer Bedeutung verwendet werden. Ebenfalls möglich ist auch die Definition weiterer Verben.

Die Verarbeitung mehrerer Repräsentationsformate stellt keine Schwierigkeit dar. Das Framework unterstützt hierbei durch die Auswahl der für den Request geeignetsten Methode. Durch die Bereitstellung von Providern und die Möglichkeit eigene Provider zu programmieren, wird dem Entwickler die Erstellung der Repräsentationen sowie umgekehrt die Umwandlung in die interne Darstellung deutlich vereinfacht bzw. weitgehend abgenommen. Lediglich bei den Verlinkung der Ressourcen bietet JAX-RS und auch Jersey keine Unterstützung.

1. Ressourcen-Implementierung	++	gute Aufteilung der Ressourcen, Implementierung in POJOs
2. URI-Zuordnung	++	bequeme Zuordnung per Annotation, Hierarchie lässt sich durch Variablen in URI-Templates abbilden
3. Schnittstelle/Methoden	++	alle Methoden sind umsetzbar, Kennzeichnung per Annotation
4. Sicherheit/Authentifizierung	+	leichte Unterstützung erweiterter Sicherheitsabfragen
5. Zustandslosigkeit	++	Framework hält keinen Anwendungsstatus
6. Caching	+	CacheControl-Objekt abstrahiert von Cache-Control Header
7. bedingte Requests	+	Bedingungsabfrage für alle Methoden möglich, aber kein Automatismus; ETag muss Anwendung selbst bereitstellen
8. Repräsentationen	++	einige Provider für Marshalling/Unmarshalling
9. Content Negotiation	++	bequeme Zuordnung per Annotation, Framework übernimmt Auswahl einer passenden Methode und eines geeigneten Providers
10. Verweise auf andere Ressourcen	o	keine besondere Unterstützung

Tabelle 4.3.: Stärken und Schwächen von Jersey

Die Implementierung ist durch die durchgängige Verwendung von Annotationen einfach und übersichtlich. Stefan Tilkov fasst seine Beschreibung der API so zusammen: „Die Unterstützung für Hypermedia ist eher dürftig, diese Meinung teilen sogar die Spec-

Leads. Dennoch macht die Entwicklung einer REST-Anwendung mit JAX-RS deutlich mehr Spaß als mit dem doch arg in die Jahre gekommenen Servlet-API.“ [Til09, S. 210]

4.5. RESTEasy 1.2.1

RESTEasy[22] ist ein weiteres Framework, welches JAX-RS realisiert, so wie Jersey. RESTEasy ist ein JBoss Projekt, wodurch die Realisierung der API laut Herstellerangaben optimal auf den JBoss Application Server abgestimmt ist. Dennoch kann RESTEasy mit jedem Servlet Container eingesetzt werden.

Wie Jersey liefert RESTEasy eine API für die Erstellung von Clients. Sie basiert auf dem Apache HttpClient 4.x. RESTEasy verwendet auch für den Client die JAX-RS-Annotationen. So ist es in den Clients möglich, ausgehende HTTP-Requests mit den von der Server-API bekannten Annotationen zu mappen. Außerdem wird Caching sowohl auf Client- als auch auf Serverseite unterstützt.

Neben Providern für XML und JSON bietet RESTEasy einige weitere Provider, wie für Fast Infoset, YAML[23], Multipart, Xop[24] oder Atom.

Mit der Servlet 3.0 Spezifikation [Mor09] wird ein asynchroner HTTP-Request-Prozess (auch COMET genannt) auf Basis langlebiger Verbindungen eingeführt, der von RESTEasy bereits unterstützt wird. Mit Apache Tomcat 6 ist diese Technik ebenfalls bereits umgesetzt worden.

Einen anderen Weg, HTTP-Request nach REST-Prinzipien asynchron zu verarbeiten, beschreiben Richardson und Ruby in [RR07, S. 259 ff.], wobei Requests, die voraussichtlich lange dauern werden, vom synchronen Request-Response-Modell entkoppelt werden. RESTEasy hat dieses Konzept des Asynchronen Job Service ebenfalls realisiert.

Durch die von JBoss bekannten Interceptoren ermöglicht RESTEasy eine ansatzweise aspektorientierte Programmierung. Dadurch ist es möglich, die an die Servlet-Filter angelehnte Filterfunktionalität nachzubauen bzw. noch weitergehende Änderungen an den Lebenszyklusphasen einer JAX-RS-Ressource vorzunehmen.

Weiterhin unterstützt RESTEasy bei der Komprimierung und Dekomprimierung von Nachrichten mit GZIP[25]. Eingehende komprimierte Nachrichten mit dem Header

[22] URI: `http://www.jboss.org/resteasy/`
[23] YAML Ain't Markup Language, URI: `http://www.yaml.org/`
[24] XML-binary Optimized Packaging, URI: `http://www.w3.org/TR/xop10/`
[25] GNU zip, URI: `http://www.gzip.org/`

`Content-Encoding: gzip` werden automatisch GZIP-dekomprimiert. Dies betrifft sowohl mit RESTEasy erstellte Services als auch Clients. Mit diesem Header versehene ausgehende Requests bzw. Responses werden vom Framework automatisch komprimiert. Der Header muss nicht manuell gesetzt werden, hierfür steht eine besondere Annotation `@GZIP` zur Verfügung. Zusätzlich werden vom Client Requests immer mit dem Header `Accept-Encoding: gzip, deflate` gesendet, ohne dass dies manuell implementiert werden muss.

RESTEasy unterstützt außerdem bei der Anbindung von EJB 3, Seam, Guice, Spring und Spring MVC. Durch ein Mock-Framework ist die serverseitige Anbindung von JUnit-Tests möglich. Weitere Informationen zu den hier beschriebenen Features werden in [RES09] gegeben.

Die Version, die in dieser Arbeit betrachtet wird, ist RESTEasy 1.2.1.GA.

4.5.1. API Implementierung

Die Implementierung der Ressourcenklassen mit RESTEasy ist nahezu identisch zu der in Abschnitt 4.4.1 vorgestellten Implementierung mit Jersey. Dies ist nicht weiter verwunderlich, da beide Frameworks den JSR 311 implementieren. Auch RESTEasy kann in der Deployment Descriptor Datei `web.xml` so konfiguriert werden, dass nach Ressourcenklassen gescannt wird, so dass diese nicht explizit zugeordnet werden müssen. Der Name der Klasse für den Servlet-Container lautet bei RESTEasy `org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher`. Die Architektur der Beispiel Web Anwendung für die Umsetzung mit RESTEasy wird in Abbildung 4.4 aufgezeigt.

Die Umsetzung der Spezifikation unterscheidet sich teilweise zwischen Jersey und RESTEasy. So liefern die Frameworks in einigen Situationen verschiedene Statuscodes. Dies ist zulässig, da HTTP [FGM+99] der Anwendung einen gewissen Spielraum bei der Interpretation und Reaktion auf Fehler lässt. Für den Fall einer nicht-numerischen Kunden-ID im URI-Pattern /customers/{id} liefert RESTEasy beispielsweise den Statuscode *400 Bad Request*. Jersey liefert in dem Fall, dass die ID nicht auf das Klassen-Attribut `id` vom Typ `Integer` gecastet werden kann, den Statuscode *404 Not Found* (vgl. 4.4.1).

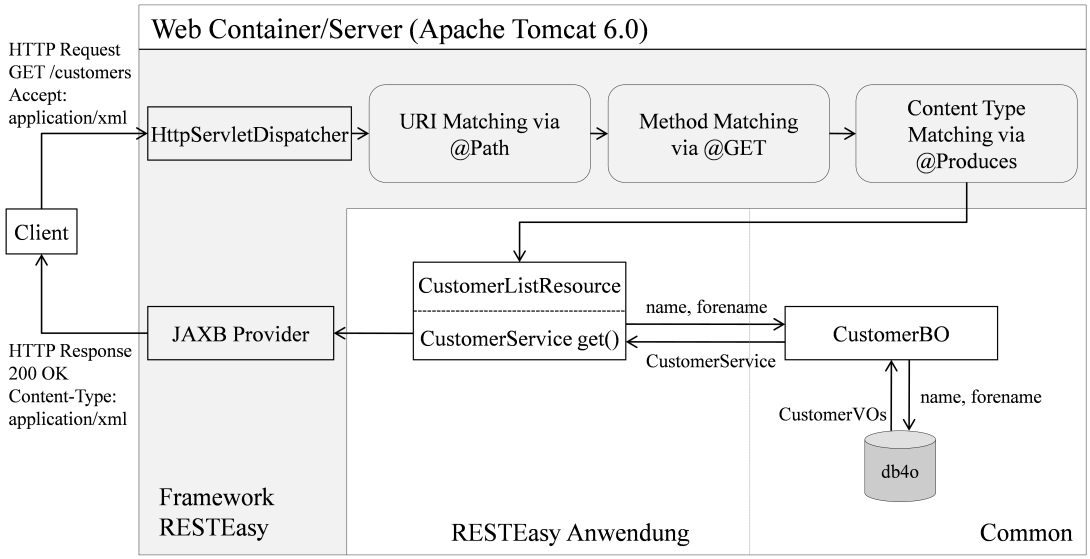


Abbildung 4.4.: Architektur der Realisierung mit RESTEasy

4.5.2. Ressourcen und Schnittstelle

Eine weitere Situation, in der RESTEasy unter bestimmten Umständen einen anderen Statuscode als Jersey liefert, ist der Aufruf einer nicht implementierten HTTP-Methode für eine existierende Ressource. RESTEasy liefert für einige Ressourcen den Statuscode *404 Not Found* und nicht wie Jersey *405 Method Not Allowed* (vgl. 4.4.2). Laut RESTEasy Reference Guide [RES09, Kapitel 27.2] soll das Framework allerdings ebenfalls *405 Method Not Allowed* liefern. Diese Abweichung von der Dokumentation und die Tatsache, dass dies nur für einige Ressourcen auftritt, lässt auf einen Fehler in der Implementierung schließen. Betroffen hiervon sind Subressourcen, in deren URI eine Template-Variable nicht am Ende steht. Im Fallbeispiel betrifft das die Ressourcen Adresse mit der URI /customers/{custid}/address/{id}, Liste der Telefonnummern eines Kunden mit der URI /customers/{custid}/phones und eine Telefonnummer mit der URI /customers/{custid}/phones/{id}. Für dieselben Ressourcen gibt es einen Fehler beim Aufruf der Methode OPTIONS. Statt der Response *200 OK* und der gewünschten Antwort im Header `Allow` antwortet RESTEasy ebenfalls mit *404 Not Found*.[26]

Die Auswirkungen dieses Fehlers sind jedoch in der Praxis kaum relevant. Die Methode OPTIONS wird sehr selten verwendet, insbesondere wenn die Clients die Anwendung kennen. An einem abweichenden Statuscode für Requests mit Methoden, die für diese Ressource nicht erlaubt sind, kann der Client dennoch erkennen, dass der Request

[26]Für die Fehler wurde im Bug Tracking System von RESTEasy ein Ticket mit der URI `https://jira.jboss.org/jira/browse/RESTEASY-363` eingestellt.

fehlgeschlagen ist, auch wenn die Antwort nicht exakt spezifikationsgemäß ist. Abgesehen von diesem Problem mit hierarchischen URIs verhält sich die Implementierung der Ressourcen und die Einhaltung der generischen Schnittstelle genau wie bei Jersey, sie ist übersichtlich und REST-konform.

Die Umsetzung von Sicherheitsmechanismen geht in RESTEasy bedeutend weiter als in Jersey. Hier wurde das Problem erkannt, dass mit der Servlet-Security keine hierarchischen URI-Strukturen abgebildet werden können. Deshalb wird in der Deployment Descriptor Datei der Zugriff auf sämtliche Ressourcen, also /* erlaubt. Die Rollen, die hierbei von den Benutzern eingenommen werden können, müssen ebenfalls explizit definiert werden. Ein weiterer Parameter aktiviert die Sicherheitsüberprüfung durch RESTEasy. In den Ressourcenklassen werden Methoden durch drei weitere Annotationen `@RolesAllowed`, `@PermitAll` und `@DenyAll` markiert. Für alle Rollen, die in der ersten Annotation aufgeführt sind, wird die Servlet-Request-Methode `isUserInRole` aufgerufen. Ist dem Benutzer eine dieser Rollen zugeordnet, wird die Methode ausgeführt. Die anderen beiden Annotationen sorgen dafür, dass alle Requests ausgeführt werden, bzw. keiner.
Wenn darauf vertraut wird, dass RESTEasy die Prüfung exakt durchführt, handelt es sich bei diesem Konzept um eine gute Lösung für die mangelnde Unterstützung hierarchischer URI-Strukturen im Deployment Descriptor. Allerdings sind für die so geschützten Ressourcen keine anonymen Requests mehr möglich. Es muss immer eine korrekte Authentifizierung für irgendeine der konfigurierten Benutzer-Rollen angegeben sein, damit der Request überhaupt an den Servlet Container und damit an das Framework weitergeleitet wird. Als Lösung dafür könnte nur ein Teilbaum der URI-Struktur geschützt werden, auf den nur authentifiziert zugegriffen werden kann oder es wird eine Benutzer-Passwort-Kombination veröffentlicht, deren zugehörige Rolle eingeschränkten Zugriff erlaubt.

4.5.3. Statushaltung und Caching

Der Lebenszyklus einer Ressourcenklasse unterstützt die Zustandslosigkeit des Services. Mit Eintreffen eines Requests wird die zuständige Ressourcenklasse instanziiert und nach dessen Beantwortung für den Garbage collector freigegeben.

Mit Hilfe der zwei Annotationen `@Cache` und `@NoCache` bietet RESTEasy Unterstützung zum Caching von Ressourcen. Diese können Methoden markieren, die gleichzeitig die Annotation `@GET` besitzen. Durch die Annotationen kann gekennzeichnet werden, ob die Response in einem Cache abgelegt werden kann oder nicht. Es werden grundsätzlich nur Responses erfolgreicher Requests im Cache abgelegt, d.h. solche, deren Response-Statuscode *200 OK* lautet. Während `@NoCache` einfach bedeutet, dass eine Response

nicht im Cache abgelegt werden darf, erzeugt RESTEasy aus `@Cache` einen komplexen `Cache-Control` Header. Die dort gesetzten Werte können beeinflusst werden, indem optionale Attribute in der Annotation gesetzt werden, wie z.B. `maxAge` für die Zeit, nach der die Response im Cache als *abgelaufen* markiert wird, oder `isPrivate` um zu verbieten, dass die Response in einem Proxy-Cache abgelegt wird.

Um auf der Serverseite einen Cache einzurichten, werden in der Deployment Descriptor Datei `web.xml` zwei Parameter gesetzt, mit denen die maximale Anzahl Einträge im Cache und das Intervall, in dem nach *abgelaufenen* Einträgen gesucht wird, gesteuert werden. Zusätzlich wird ein weiterer Listener definiert, der dafür sorgt, dass entsprechend markierte Responses in einem speziellen JBossCache abgelegt werden.
Bei der Ablage im Cache erhalten die Einträge automatisch ein ETag, das mit dem MD5[27]-Hash-Algorithmus aus dem Response-Body berechnet wird. Bedingte GET-Requests werden automatisch mit dem Eintrag im Cache verglichen, wodurch die Anwendung entlastet wird. Es handelt sich also um eine Kombination aus Caching und bedingten Requests. Von GET verschiedene Requests werden nicht berücksichtigt. Sie werden weder verwendet, um z.B. bedingte PUT-Requests abzuweisen, noch werden durch sie Cache-Einträge aktualisiert. Dies hat zur Folge, dass ein bedingter GET-Request durch den Vergleich mit der Kopie im Cache die Response *304 Not Modified* erhält, auch wenn die Ressource zwischenzeitlich durch einen PUT-Request manipuliert wurde. Eine genaue Beschreibung der Einrichtung von Caches steht in [RES09, Kapitel 30].

4.5.4. Repräsentationen und Verweise

JAX-RS gibt eine Handvoll Standard-Provider vor, die bereitgestellt werden müssen. Dies schließt den JAXB-Provider ein, der das Marshalling und Unmarshalling der XML-Repräsentationen durchführt. RESTEasy stellt sogar mehrere JAXB-XML-Provider bereit, um die feinen Unterschiede zwischen Klassen mit `@XmlRootElement` Annotation, durch XJC generierte Klassen oder `JAXBElement` Klassen abzubilden. Der bereitgestellte JAXB-JSON-Provider verwendet normalerweise die Jettison JSON Bibliothek[28]. Wahlweise kann stattdessen auch Jackson[29] verwendet werden. Ein weiterer JAXB-Provider wird für Fast Infoset zur Verfügung gestellt. JAXB-Objekte können außerdem in Java-Arrays, `Set`s, `List`s und `Map`s bereitgestellt werden. Das Framework ist in der Lage, diese in eine Repräsentation zu marshallen. Die drei Formate XML, JSON und Fast Infoset können auch für eine Atom-Anbindung eingesetzt werden. RESTEasy enthält hierfür wieder einige Klassen zur Abbildung der Atom-Objekte. Diese enthalten

[27] Message-Digest Algorithm 5
[28] URI: `http://jettison.codehaus.org/`
[29] URI: `http://jackson.codehaus.org/`

entsprechende JAXB-Annotationen. In den Ressourcenklassen wird `atom` den Subtypen vorangestellt, die in den Annotationen `@Consumes` bzw. `@Produces` angegeben werden, z.B. `"application/atom+json"` oder `"application/atom+fastinfoset"`.

Ein weiteres Format, was durch Provider unterstützt wird, ist YAML. Hierfür wird die Jyaml-Bibliothek[30] verwendet. YAML ist eine Auszeichnungssprache, die ursprünglich an XML angelehnt war, die Informationen aber einfacher und für Menschen lesbarer darstellt.

Desweiteren bietet RESTEasy eine Vielzahl an Providern für die Multipart-Typen `"multipart/*"` und insbesondere `"multipart/form-data"`. Multipart-Typen werden verwendet, wenn in einer Nachricht mehrere Bodies mit verschiedenen Formaten enthalten sind. In diesem Zusammenhang wird die Übertragung von Xop-Nachrichten mit dem Format `"multipart/related"` unterstützt. Hierbei werden Binärdaten, die nicht mehr codiert werden sollen, übertragen, was zur Beschleunigung des Transfers beiträgt.

Durch den enormen Umfang an mitgelieferten Providern ist ein Service in der Lage, mit überschaubarem Aufwand viele moderne Repräsentationformate anzubieten. Dies ist zweifellos eine der Stärken des RESTEasy-Frameworks. Teilweise sehr ausführliche Beschreibungen zur Anbindung der aufgeführten Providern können in den Kapiteln 18 bis 26 in [RES09] nachgelesen werden. Eine Hilfe bei die Verlinkung der Ressourcen bietet RESTEasy nicht an.

4.5.5. Zusammenfassung und Bewertung

Da RESTEasy die Spezifikation JSR 311 implementiert, ist die Basis für die Erstellung von RESTful Web Services gegeben. Die Ressourcenklassen sind dadurch denen mit dem Jersey-Framework erstellten sehr ähnlich. RESTEasy bietet jedoch weitere Funktionalität durch weitere Annotationen. Dadurch ist ein Umstieg von Jersey zu RESTEasy relativ einfach möglich, umgekehrt bei Ausnutzung aller Möglichkeiten nicht ohne weiteres.

Das derzeit bestehende Problem einiger Methoden-Mappings bei hierarchischen URIs ist zwar unschön, aber nicht dramatisch. Es ist möglich, funktionierende Web Services zu erstellen. Möglicherweise wird der Fehler in einer zukünftigen Version behoben.

RESTEasy wartet mit einer Vielzahl bereitgestellter Provider auf, durch die eine Verarbeitung vieler Repräsentationsformate sehr einfach möglich ist. Wie bei Jersey kann das Angebot durch Erstellung eigener Provider ergänzt werden. Für die Verlinkung der Ressourcen bietet allerdings auch RESTEasy keine Unterstützung.

[30] URI: `http://www.jyaml.de/`

1. Ressourcen-Implementierung	++	gute Aufteilung der Ressourcen, Implementierung in POJOs
2. URI-Zuordnung	++	bequeme Zuordnung per Annotation, Hierarchie lässt sich durch Variablen in URI-Templates abbilden
3. Schnittstelle/Methoden	++	alle Methoden sind umsetzbar, Kennzeichnung per Annotation
4. Sicherheit/Authentifizierung	++	per Annotationen Rollenprüfung pro Methode möglich
5. Zustandslosigkeit	++	Framework hält keinen Anwendungsstatus
6. Caching	++	Caching-Verhalten durch Annotationen steuerbar
7. bedingte Requests	+	Kombination aus Caching und Bedingungsabfrage für GET per automatisch erstelltem ETag
8. Repräsentationen	++	Vielzahl von Providern für Marshalling/Unmarshalling
9. Content Negotiation	++	bequeme Zuordnung per Annotation, Framework übernimmt Auswahl einer passenden Methode und eines geeigneten Providers
10. Verweise auf andere Ressourcen	o	keine besondere Unterstützung

Tabelle 4.4.: Stärken und Schwächen von RESTEasy

4.6. Restlet 1.1.7

Die Open Source Bibliothek Restlet[31] hat eine vergleichsweise lange Geschichte. Noch bevor Sun JAX-RS spezifiziert hat, beschloss Jérôme Louvel, der Begründer von Restlet, ein eigenes REST-Framework zu entwickeln. Grund dafür war, dass in Java ein seinem Verständnis von REST entsprechendes Framework fehlte.

Die Grundidee von Restlet beruht auf einer einzigen abstrakten Klasse `Uniform`, welche als Basis für alle Applikationen, Connectoren usw. dient. Die Vereinbarung, nach der sich alle Teile des Frameworks richten, ist eine in dieser Klasse zur Verfügung gestellte Methode: `handle(Request request, Response response)`. Dieses Prinzip, einen Request zu verarbeiten und eine Response zu erstellen, kapselt alle Aspekte von Restlet, um die Protokollunabhängigkeit zu wahren. Mit derselben API kann sowohl die Client- als auch die Serverseite entwickelt werden, wodurch Einarbeitungs- und Wartungsaufwände gespart werden können, da nur eine API „erlernt“ werden muss.

[31] URI: `http://www.restlet.org/`

Die Verarbeitung eines Requests erfolgt in zwei Schritten. Zunächst wird eine Kette von sogenannten Restlets vom Root-Restlet bis zur Ressourcenklasse verarbeitet. Im zweiten Schritt erfolgt die Verarbeitung innerhalb der Ressourcenklasse.

Die aktuellste Version ist Restlet 1.1.7, welche auch in diesem Kapitel untersucht wird. Es gibt bereits eine Version 2.0, die allerdings noch als *unstable* bezeicht wird. Dennoch bietet sie einige interessante und vielversprechende Neuerungen, die es wert sind, im Abschnitt 4.7 separat untersucht zu werden.

Restlet Anwendungen können in beliebigen Servlet Containern oder einfach in der JVM[32] betrieben werden. Es gibt eine weitere Restlet 1.1 Edition für das GWT[33], mit der Restlet innerhalb eines Web Browsers läuft.

Das Framework nimmt dem Service einige Arbeit ab. So werden Repräsentationen bei Bedarf nach ihrer Ankunft automatisch dekodiert und entschlüsselt. Dies geschieht für die Anwendung völlig transparent. Sämtliche Requests werden in einer Logdatei nach dem „W3C Extended Log File Format“ protokolliert. Ähnlich wie durch das „Apache Rewrite Modul“ können URIs umgeschrieben werden, wodurch es möglich ist, die interne Adressierung einer Ressource von der/den nach außen bekannten URI(s) zu entkoppeln.

Zusätzlich zu dem HTTP Server Connector von Jetty können auch die des Simple oder des Grizzly Frameworks verwendet werden. Basierend auf dem Jetty Connector ist auch der AJP[34] Server Connector verfügbar.

Analog zu den Standard-Providern für JAX-RS bietet auch Restlet Hilfe bei der Verarbeitung von Repräsentationen, wie beispielsweise XML mittels XPath API und Engine. Auf XML-Repräsentationen können mittels XSLT[35] Transformationen durchgeführt werden. Auch die Unterstützung von JSON und Atom ist integriert. FreeMarker[36] und Velocity[37] sind sogenannte Template Engines, also Lückentext-Ersetzer, die von Restlet unterstützt werden. Zur Verarbeitung großer Dateien und Multipart Forms wird von Restlet der Apache FileUpload[38] verwendet. Weiterhin wird ein erweiterbarer Satz an Kern-Repräsentationen angeboten, die auf BIO Streams (Package `java.io`) oder NIO Channels (Package `java.nio`) basieren.

[32] Java Virtual Machine
[33] Google Web Toolkit
[34] Apache JServ Protocol
[35] Extensible Stylesheet Language Transformation
[36] URI: `http://freemarker.org/`
[37] URI: `http://velocity.apache.org/`
[38] URI: `http://commons.apache.org/fileupload/`

Eine Anbindung des Spring IoC[39] Frameworks ist genauso in Restlet integriert, wie auch eine spezielle Erweiterung für Oracle 11g.

Die Featureliste von Restlet enthält außerdem diverse Sicherheitsmechanismen, wie HTTP Basic und Digest, OAuth auf Serverseite, HTTPS, SMTPS und POPS. Für die Clientseite können zusätzlich die Authentifizierungen von Amazon S3 sowie von Microsoft Shared Key und Shared Key Lite eingesetzt werden.

Über eine sogenannte Extension stellt Restlet seit der Version 1.1 eine Implementierung von JAX-RS zur Verfügung. Diese ist jedoch nicht näher untersucht worden, da in den Kapiteln 4.4 und 4.5 bereits auf zwei JAX-RS Implementierungen eingegangen wurde und nach eigenen Angaben[40] die Restlet-API mächtiger ist als JAX-RS.

4.6.1. API Implementierung

Das Root-Restlet, das die gesamte Anwendung beschreibt, ist eine Klasse, die von `Application` abgeleitet wird. Diese erweitert die Klasse `Restlet`, welche wiederum eine Spezialisierung von `Uniform` ist. In der geerbten Methode `createRoot` wird eine Instanz der Klasse `Router` erzeugt. Diesem Router werden die Zuordnungen von URIs zu den Ressourcenklassen mitgeteilt. Die Zuordnung erfolgt also nicht wie bei der Servlet-API in der Deployment Descriptor Datei `web.xml`. Die URIs können mehrere Template-Variablen enthalten, welche extrahiert und dem Request als Attribute hinzugefügt werden.
Außerdem können dem Router weitere `Restlet`s zugefügt werden. Neben `Application` sind Spezialisierungen von `Restlet` beispielsweise `Connector`, `Redirector`, `Router` oder `Filter`. Letztere sind vergleichbar mit denen von der Servlet-API bekannten Filtern. Einige Spezialisierungen hiervon sind `Encoder` und `Decoder`, `StatusFilter`, `TemplateFilter` oder `Guard`. Dieser letztgenannte überwacht die Authentifizierung für definierte Ressourcen. Sämtliche `Restlet`s können in Ketten verknüpft werden, deren letztes Glied dann beispielsweise erst eine Ressourcenklasse ist. Der Router sorgt für das Leiten eines Requests durch diese Kette bis hin zu den Ressourcen, sowie der Response wieder zurück. Dabei ist es sogar möglich, verschiedene Strategien für die Wahl des Pfades festzulegen. Die Architektur der Beispiel Web Anwendung für die Umsetzung mit Restlet 1.1 wird in Abbildung 4.5 aufgezeigt.

Ressourcenklassen in Restlet erweitern die Klasse `Resource`. Der Konstruktor bekommt als Parameter den `Context`, den `Request` und die `Response` übergeben. Diese drei Werte werden via Aufruf von `super` in geerbten Klassenattributen hinterlegt. Die Klasse

[39] Inversion of Control
[40] URI: `http://www.noelios.com/products/restlet-framework` (Stand: 27.2.2010)

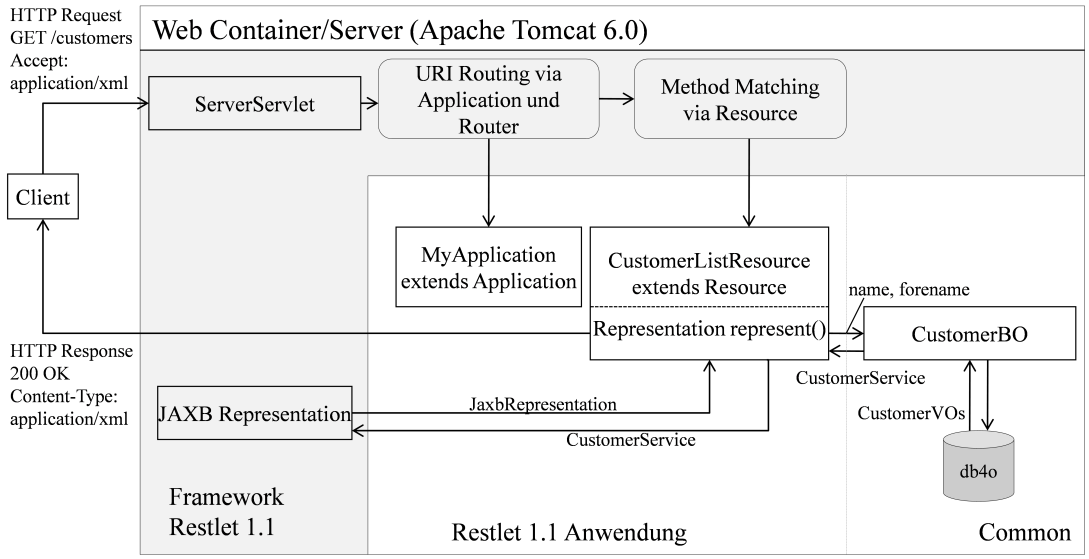


Abbildung 4.5.: Architektur der Realisierung mit Restlet 1.1

`Resource` enthält fünf weitere Attribute, über die das Verhalten der Ressource gesteuert werden kann und die deshalb hier vorgestellt werden.
In der Liste `variants` werden die Repräsentationsformate der Ressource aufgeführt. Diese werden im Konstruktor der Ressourcenklasse gefüllt, also der Liste zugefügt. Hierfür stehen auch Restlet-spezifische Abkürzungen bereit, um ähnliche Typen zusammenzufassen. Beispielsweise werden durch die Angabe von `"xml"` die Typen *text/xml* und *application/xml* vereint. Die Varianten gelten für die Methoden GET und HEAD, sie bedeuten also lediglich **erzeugbare** Repräsentationen. Es gibt kein Pendant für **verarbeitbare** Repräsentationen. In den Fällen PUT und DELETE werden die Repräsentationsformate zwar im Zusammenhang mit bedingten Requests ausgewertet, allerdings nur um die Vorabprüfung durchzuführen und nicht, um sie mit dem Header `Content-Type` zu vergleichen.
Die anderen vier Attribute sind vom Typ `boolean`. Das Attribut `negotiateContent` kennzeichnet, ob das Framework die Content Negotiation übernehmen soll. Falls ja, wird der im Request mitgelieferte Header mit den verfügbaren Repräsentationen in der Liste `variants` abgeglichen.
Mit `available` wird die generelle Verfügbarkeit einer Ressoure gesteuert. Normalerweise kann damit die (Nicht-)Existenz einer Ressource gekennzeichnet werden, beispielsweise abhängig davon, ob eine Datei oder ein Datenbanksatz existiert, in der die Ressource gespeichert ist. Requests für nicht verfügbare Ressourcen werden vom Framework mit *404 Not Found* beantwortet.
Das Attribut `readable` regelt, ob die lesenden Zugriffe GET und HEAD zugelassen

werden sollen. Analog wird mit `modifiable` festgelegt, ob modifizierende Zugriffe wie POST, PUT und DELETE zulässig sind. Auf Requests für unzulässige Methoden reagiert das Framework mit *405 Method Not Allowed*.
Über die jeweiligen Set-Methoden können diese Attribute manipuliert werden, damit Requests vom Framework an die Ressourcenklasse durchgereicht oder entsprechend direkt beantwortet werden können.

Für jedes ressourcenbezogene HTTP-Verb existiert eine Methode `allow<HTTP-Verb>`, mit der die Zulässigkeit geprüft wird. Die Standardimplementierung in der Klasse `Resource` wertet dabei die Attribute `readable` und `modifiable` aus. Um gezielt einzelne Methoden zu verbieten, z.B. PUT für die Ressource Kundenliste oder POST für die Ressource Kunde, können die zugehörigen `allow<HTTP-Verb>`-Methoden überschrieben werden. Dies wirkt sich gleichzeitig auf die Reaktion auf einen OPTIONS-Request aus, den das Framework eben anhand dieser Methoden selbstständig beantwortet. Vorausgesetzt die Methode `allowOptions` wird nicht so überschrieben, dass sie `false` liefert.

Die für eine Ressource erlaubten Methoden müssen ebenfalls noch implementiert werden. Hierfür werden von `Resource` geerbte Methoden überschrieben, deren Standard-Implementierung ansonsten den Statuscode *500 Internal Server Error* an den Client zurückgibt. Diese Methoden heißen `acceptRepresentation` für POST, `storeRepresentation` für PUT, `represent` für GET (und HEAD) und `removeRepresentations` für DELETE. Die Namen der Methoden deuten darauf hin, dass von dem zugrunde liegenden Protokoll HTTP abstrahiert werden soll. In [RR07, S. 398] schreibt Louvel, dass die Schnittstelle maßgeblich von HTTP inspiriert wurde, aber auch mit Protokollen wie FTP und SMTP verwendet werden kann.

Die Methode `acceptRepresentation` enthält einen Parameter vom Typ `Representation`. Darin verbirgt sich die zu akzeptierende Repräsentation inklusive Metadaten, wie Format, Sprache usw. Der Parameter der Methode `storeRepresentation` ist ebenfalls vom Typ `Representation`. Das Framework ruft diese Methode erst auf, nachdem es Vorabbedingungen für das bedingte PUT überprüft hat. Auch `represent` wird erst aufgerufen, nachdem die Prüfungen für bedingtes GET erfolgt sind und vom Framework sichergestellt wurde, dass das angeforderte Repräsentationsformat generell zur Verfügung steht. Sie erhält einen Parameter vom Typ `Variant`. Dies ist die Oberklasse von `Representation`, sie enthält die Metadaten einer Repräsentation. Die Rückgabe ist dann die vollständige `Representation`. Kein Parameter wird der Methode `removeRepresentations` übergeben. Vor dem Aufruf erfolgen die Prüfungen für bedingtes DELETE.
Werden in den Methoden weitere Informationen benötigt, beispielsweise Query-Attribute, Request-Header oder die Werte der Template-Variablen, können diese aus den im Konstruktor gelieferten Werten Kontext, Request und Response ausgelesen werden.

Umgekehrt können so auch zusätzliche Header in der Response oder der Statuscode gesetzt werden. Da vom Framework keine Content Negotiation bezüglich **verarbeitbarer** Repräsentationen durchgeführt wird, muss dies bei Bedarf in den Methoden `acceptRepresentation` und `storeRepresentation` getan werden.

4.6.2. Ressourcen und Schnittstelle

Eine Aufteilung der Ressourcen in zugehörige Ressourcenklassen ist auch mit Restlet problemlos möglich. Die Zuordnung zwischen Ressourcen und URIs findet an zentraler Stelle in der von `Application` abgeleiteten Klasse statt. Hierbei ist der Einsatz von Templates mit mehreren Variablen möglich, wodurch eine hierarchische URI-Struktur realisiert werden kann.

In den Ressourcenklassen sind die für die jeweilige Ressource erlaubten Methoden implementiert. Für nicht erlaubte Methoden wird die passende `allow<HTTP-Verb>`-Methode überschrieben und `false` zurückgeliefert. Dies ist deswegen notwendig, weil die Standard-Implementierung statt *405 Method Not Allowed* den Statuscode *500 Internal Server Error* zurückgibt. Es erscheint etwas umständlich, dennoch ist es möglich, auf Requests für unerlaubte Methoden wie gewünscht zu reagieren. Wie bereits erwähnt, beeinflusst dies ebenfalls die Response auf OPTIONS, welche Restlet automatisch erzeugt. Die einheitliche Schnittstelle kann dementsprechend realisiert werden.

Die Standardimplementierung für HEAD ruft dieselbe Methode wie für GET auf und ignoriert den erstellten Entity-Body in der Response. Auch hier besteht das Risiko Performance einzubüßen, wenn die Repräsentation sehr groß ist.

Das Anlegen eines neuen Kunden mit der Methode `acceptRepresentation` der Klasse `CustomerListResource` wird im Codebeispiel 4.7 gezeigt. Der einzige Parameter der Methode enthält die Repräsentation einschließlich deren Metadaten. Da die Content Negotiation nur für lesende Zugriffe erfolgt, wird der Header `Content-Type` mittels der Hilfsklasse `CheckMediaTypeUtil` überprüft. Wurde ein akzeptiertes Format geliefert, wird dem `BusinessObject` die vom Client übermittelte Repräsentation mittels der Methode `getText` übergeben, um das `ValueObject` zu erzeugen. Dies ist hier möglich, da in dem Fallbeispiel die Entity nicht besonders groß ist. In einer echten Anwendung sollte sicherheitshalber überprüft werden, welchen Umfang die Repräsentation besitzt. Die Response ist in einem Klassenattribut per Konstruktor bereits bekannt, in ihr werden der Statuscode *201 Created* und der Header `Location` gesetzt. Der Rückgabewert der Methode ist `void`. Tritt bei der Anlage des Kunden ein Fehler auf, erzeugt die Klasse `ExceptionUtil` eine `ResourceException` mit einem passenden Statuscode und im Response-Body einen Freitext dazu.

```
1
2  public class CustomerListResource extends Resource {
3
4    ...
5
6    /**
7     * Legt einen neuen Kunden an.
8     */
9    public void acceptRepresentation(Representation entity)
10       throws ResourceException {
11
12     try {
13       CheckMediaTypeUtil.checkContentTypeHeader(
14           entity.getMediaType().toString());
15
16       final CustomerVO customerVO =
17           new CustomerBO(Framework.RESTLET11).create(entity.getText());
18       getResponse().setStatus(Status.SUCCESS_CREATED);
19       getResponse().setLocationRef(
20           getRequest().getResourceRef().getIdentifier() + "/" +
21           customerVO.getId());
22     } catch (CommonException e) {
23       throw ExceptionUtil.handleCommonException(e);
24     } catch (IOException e) {
25       throw ExceptionUtil.handleCommonException(
26           new InternalServerErrorException(
27               "XML konnte nicht ermittelt werden."));
28     }
29   }
30
31   ...
32
33 }
```

Listing 4.7: Implementierung der Methode POST der Ressource Kundenliste

Da für das Fallbeispiel eine Authentifizierung nur für eine Methode einer Ressource notwendig ist, wurde statt einen `Guard` zu erstellen und dem Router bekanntzugeben ein anderer Weg gewählt. Hierbei wurde der `Guard` unmittelbar in der für DELETE zuständigen Methode `removeRepresentations` erzeugt. Dafür wird neben dem Kontext das Authentifizierungsschema und der Wert für `realm` benötigt. Anschließend werden dem `Guard` sämtliche berechtigten Benutzer und deren Passworte mitgeteilt. Dies ist hier akzeptabel, da im Fallbeispiel lediglich ein einziger Benutzer berechtigt ist. Vor Durchführung des Löschvorgangs wurde die `Guard`-Methode `authenticate` aufgerufen, welcher der Request übergeben wird. Lautet das Ergebnis, dass der Request von einem berechtigten und korrekt authentifizierten Benutzer aufgerufen wurde, wird fortgefahren, anderenfalls wird dem Aufrufer ein Challenge Request zusammen mit dem Statuscode *401 Unauthorized* zurückgeliefert.
Um in einer realen Anwendung nicht in jeder zu schützenden Methode sämtliche berechtigten Benutzer inklusive Passwort aufzählen zu müssen, könnte man den `Guard` in einer separaten Klasse erzeugen und bereitstellen. Um ein Rollenmodell zu realisieren, könnte man für jede Rolle jeweils einen `Guard` definieren.

4.6.3. Statushaltung und Caching

Auch in Restlet wird die Zustandslosigkeit des Services durch pro Request instanziierte Ressourcenklassen voll unterstützt. Es wurden sogar die von der Servlet-API bekannten HTTP-Sessions bewusst wieder aus dem Framework entfernt, um die Skalierbarkeit zu vereinfachen und dieses REST-Prinzip sicherzustellen.

Eine besondere Möglichkeit, das Caching-Verhalten der Ressourcen zu steuern ohne explizit den `Cache-Control` Header zu setzen, gibt es in den Ressourcenklassen nicht. Welche Aufgaben genau Restlet übernimmt, wenn es als unabhängige Anwendung betrieben wird, wurde im Rahmen dieser Arbeit nicht näher betrachtet, hier wurde wie für alle Frameworks nur der Betrieb im Apache Tomcat untersucht.

Für die Methoden GET, PUT und DELETE sind bedingte Zugriffe möglich, hierbei ermittelt das Framework automatisch, ob im Request Bedingungsheader angegeben wurden und führt die Prüfungen selbstständig aus. Dazu benötigt das Framework die Repräsentation und darin gesetzte Werte für das Änderungsdatum bzw. das ETag, für beide Werte wurde das Datum der letzten Änderung aus dem `ValueObject` eingesetzt[41]. Bei GET erfolgt die Prüfung, nachdem die Ressourcenmethode die Repräsentation erzeugt hat. Die möglicherweise aufwändige Erstellung der Repräsentation wird dadurch nicht verhindert, wohl aber die Übertragung zum Client. Bei PUT wird vor der Ausführung der Änderung ebenfalls mit Hilfe der Methode `represent` die möglicherweise aufwändige Repräsentation erzeugt und die Vorabbedingungen durchgeführt. Hierbei ist ein Fehler in der Framework-Implementierung aufgefallen. Ein bedingter PUT-Request, wobei im Header `If-Unmodified-Since` das exakte Änderungsdatum eingetragen ist, führt zur Response *412 Precondition Failed*. Ursache hierfür ist, dass bei Bedingung das Datum nicht auf Gleichheit abgefragt wird.[42]
Die Verwendung des ETags bei bedingten PUT-Requests funktioniert jedoch einwandfrei, was in der Praxis vermutlich häufiger verwendet wird als das Änderungsdatum.

4.6.4. Repräsentationen und Verweise

Ob vom Client übermittelte Repräsentationen akzeptiert werden oder nicht, muss vom Entwickler in den Methoden jeweils implementiert werden. Die Content Negotiation wird von Restlet nur für die Methoden GET und HEAD übernommen. Dafür werden

[41] Dies ist ein sehr trivialer Wert für das ETag, für diese Untersuchung aber ausreichend.
[42] Für den Fehler wurde im Bug Tracking System von Restlet ein Ticket mit der URI `http://restlet.tigris.org/issues/show_bug.cgi?id=1038` eingestellt, woraufhin der Fehler inzwischen behoben wurde.

akzeptierte „Varianten“ im Konstruktor festgelegt. Dadurch sind die Repräsentationsformate an einer zentralen Stelle übersichtlich definiert. Das Framework gleicht bei Content Negotiation nicht nur den `Accept`-Header mit dem Format der Variante ab, sondern ebenfalls die Header `Accept-Charset`, `Accept-Encoding` und `Accept-Language`.

Die Methode `represent` ist für die Verarbeitung von GET- und HEAD-Requests zuständig. Als Parameter werden die Metadaten der Repräsentation, wie beispielsweise das Format oder die Sprache, übergeben. Die Erstellung der Repräsentation erfolgt in der Methode abhängig von diesen Metadaten, wobei Restlet durch bereitgestellte Spezialisierungen der Klasse `Representation` unterstützt. Es ist außerdem möglich, durch eigene Ableitungen das Repertoire zu erweitern. Durch eine Fallunterscheidung wird zwischen den verfügbaren Repräsentationen ausgewählt.

```
1
2  public class CustomerListResource extends Resource {
3
4    ...
5
6    /**
7     * Erstellt eine Kundenliste, filtert diese anhand des Namens
8     * und des Vornames und gibt diese zurück.
9     */
10   public Representation represent(Variant variant) throws ResourceException {
11     final Form form = getRequest().getResourceRef().getQueryAsForm();
12     final Parameter name = form.getFirst(QueryParameter.NAME.toString());
13     final Parameter forename = form.getFirst(
14         QueryParameter.FORENAME.toString());
15
16     try {
17       if (MediaType.APPLICATION_XML.equals(variant.getMediaType())) {
18         return new JaxbRepresentation<CustomerService>(
19             new CustomerBO(Framework.RESTLET11).getCustomerServiceByQueryParams(
20                 (name == null ? null : name.getValue()),
21                 (forename == null ? null : forename.getValue())));
22       } else if (MediaType.APPLICATION_JSON.equals(variant.getMediaType())) {
23         return new JsonRepresentation(
24             new CustomerBO(Framework.RESTLET11).getCustomerServiceByQueryParams(
25                 (name == null ? null : name.getValue()),
26                 (forename == null ? null : forename.getValue())));
27       } else {
28         throw ExceptionUtil.handleCommonException(
29             new NotAcceptableException(
30                 "Es werden nur XML und JSON unterstützt"));
31       }
32     } catch (CommonException e) {
33       throw ExceptionUtil.handleCommonException(e);
34     }
35   }
36
37   ...
38
39 }
```

Listing 4.8: Implementierung der Methode GET der Ressource Kundenliste

Die Erzeugung der Repräsentationen für die Ressource Kundenliste wird in Codebei-

spiel 4.8 gezeigt. Es enthält die Methode `represent` der Klasse `CustomerListResource`. Zunächst werden die Query-Parameter `name` und `forename` ermittelt. Für den leichteren Zugriff auf die Parameter stellt Restlet die Klasse `Form` zur Verfügung. Zur Erzeugung der Repräsentationen werden die ebenfalls vom Framework bereitgestellten Klassen `JaxbRepresentation` und `JsonRepresentation` verwendet. Deren Konstruktoren wird jeweils ein JAXB-Objekt vom Typ `CustomerService`, welches vom `BusinessObject` erzeugt wird, übergeben. Die erzeugte Repräsentation wird an das Framework zurückgegeben. Der Statuscode *200 OK* wird von Restlet im Erfolgsfall automatisch gesetzt.

4.6.5. Zusammenfassung und Bewertung

Restlet ist ein recht komplexes Framework mit einer Vielzahl von Klassen und Bibliotheken. Dies macht es einerseits relativ mächtig, andererseits etwas unübersichtlich.

Nichtsdestotrotz ist die Erstellung von Web Services nach den Prinzipien von REST sehr gut möglich. Die Aufteilung der Ressourcenklassen und ihre Zuordnung zu URIs erfolgt übersichtlich an einer zentralen Stelle. Auch die Abbildung der Ressourcen-Hierarchie ist durch die Angabe entsprechender URI-Templates gut möglich.

Restlet unterstützt neben HTTP noch einige weitere Protokolle wie beispielsweise SMTP, was aber nicht näher untersucht wurde. Die Unabhängigkeit vom Protokoll ist jedoch ein weiterer positiver Aspekt bei der Erstellung von Anwendungen nach dem Prinzip von REST.

Die Unterstützung der Content Negotiation ermittelt für lesende Zugriffe das am besten passende Format. Abhängig davon verschiedene Repräsentationen zu erzeugen, bleibt jedoch dem Entwickler überlassen. Für ändernde Requests muss die Content Negotiation komplett selbstständig implementiert werden. Durch die Bereitstellung von Repräsentationsklassen und die Möglichkeit diese um eigene zu ergänzen, wird dem Entwickler die Erstellung der Repräsentationen deutlich vereinfacht bzw. weitgehend abgenommen. Für die Verlinkung der Ressourcen bietet allerdings auch Restlet keine Lösung an.

Die Implementierung wirkt nicht ganz so modern, Restlet ist seine Geschichte teilweise anzumerken. Auch die Methodennamen sind gewöhnungsbedürftig, wenn wie in dem Fallbeispiel lediglich HTTP realisiert wird, wovon die Methoden ja gerade abstrahieren wollen. Insbesondere `acceptRepresentation` und `storeRepresentation` können jedoch leicht zu Verwechslungen führen. Weiterhin sind einige Methoden zu überschreiben, wie z.B. `allowPut`, was nicht unbedingt intuitiv ist. Andererseits können letztlich alle geerbten Methoden überschrieben werden, wodurch bei Bedarf das Verhalten der Ressource sehr feingranular gesteuert werden kann.

Für viele Header bieten die Klassen `Request` und `Response` separate Lese- und Schreib-Methoden an, aber nicht für alle. Bei der Untersuchung wurden beispielsweise die Methoden zur Manipulation von `Cache-Control` vermisst. Im Prinzip ist es jedoch sinnvoll, die Anwendungslogik von protokollspezifischen Details zu trennen.

1. Ressourcen-Implementierung	++	gute Aufteilung der Ressourcen, Spezialisierung einer Frameworkklasse
2. URI-Zuordnung	++	Zuordnung in Root-Restlet, Hierarchie lässt sich durch Variablen in URI-Templates abbilden
3. Schnittstelle/Methoden	++	alle Methoden sind umsetzbar, Standardimplementierung wird überschrieben
4. Sicherheit/Authentifizierung	++	diverse Möglichkeiten der Authentifizierung, keine Angaben in web.xml notwendig
5. Zustandslosigkeit	++	Framework hält keinen Anwendungsstatus
6. Caching	o	keine besondere Unterstützung
7. bedingte Requests	++	wenn in Repräsentation Datum bzw. ETag gesetzt werden, erfolgt automatische Prüfung
8. Repräsentationen	+	einige Standard-Repräsentationen stehen zur Verfügung, eigene können ergänzt werden; Fallunterscheidung nach Format notwendig
9. Content Negotiation	+	Framework übernimmt Auswahl der passendsten „Variante“, aber nur für GET/HEAD
10. Verweise auf andere Ressourcen	o	keine besondere Unterstützung

Tabelle 4.5.: Stärken und Schwächen von Restlet 1.1

4.7. Restlet 2.0

Mit Restlet 2.0 hat das Framework einige Modernisierungen erfahren. Das Ausmaß der Veränderungen spiegelt sich auch in der Versionsnummer wider, das Release hatte zunächst die Nummer 1.2 und ist dann nachträglich in 2.0 geändert worden. Das Framework wurde einem umfangreichen Refactoring unterzogen, viele Klassen und Interfaces wurden umbenannt, in andere Packages verschoben oder mit `@Deprecated` markiert. Einige Klassen, die in Version 1.1 noch mit `@Deprecated` markiert waren, sind inzwischen komplett entfallen. Die zentrale Klasse `Uniform` wurde in ein Interface verwandelt. Analog zu JAX-RS können Ressourcenmethoden nun durch Annotationen als solche markiert werden.

Ein weiteres großes Schlagwort für die Neuerungen in 2.0 lautet „Ready for the Semantic Web (Web 3.0)“ mit voller Unterstützung von RDF[43]. Das Semantische Web ist eine Idee vom Begründer des World Wide Web, Tim Berners-Lee, bei der es darum geht, Ressourcen mit Metadaten bezüglich ihrer Bedeutung anzureichern, um Zusammenhänge zwischen Ressourcen sichtbar zu machen.

Die zur Verfügung gestellten Repräsentationen wurden erweitert, so dass jetzt weitere Formate vom Framework abgebildet werden können. Zu sämtlichen Repräsentationen werden durch das Apache Lucene Tika[44] Metadaten zur Verfügung gestellt. Zusätzlich unterstützt Restlet bei der Erstellung der Repräsentationen durch Konvertierung von beliebigen Objekten.

Ebenfalls erweitert wurde die Unterstützung für Authentifizierung und Autorisierung.

Für die Clientseite wurden einige Connectoren ergänzt, unter anderem HTTPS, SMTP(S), POP(S) v3, JDBC und FILE Connectoren.

Mit Restlet 2.0 wird das Framework in fünf verschiedenen Editionen bereitgestellt:

- Edition für Java SE
- Edition für Java EE, welche für diese Arbeit eingesetzt wurde
- Edition für Google Web Toolkit
- Edition für Google App Engine
- Edition für Android (für Google Android Mobilgeräte)

Der komplette Umfang der Änderungen kann dem Änderungsprotokoll[45] entnommen werden. Die zu diesem Zeitpunkt neueste verfügbare Version, die in den nächsten Abschnitten untersucht wird, ist „Version 2.0 Milestone 6“.

4.7.1. API Implementierung

Von dem Refactoring ist unter anderem auch die Klasse `Application` betroffen. Im Root-Restlet zur Beschreibung der Anwendung und seiner Pfade wird nun statt der Methode `createRoot` die Methode `createInboundRoot` überschrieben. Gleichzeitig wird eine weitere Methode `createOutboundRoot` angeboten, mit der ausgehende Aufrufe bearbeitet werden. Auch die Klassen für Authentifizierung und Autorisierung sind überarbeitet worden. Beispielsweise wurde die Klasse `Guard` von den Klassen `Authenticator`,

[43] Resource Description Framework
[44] URI: `http://lucene.apache.org/tika/`
[45] URI: `http://www.restlet.org/documentation/snapshot/jse/changes` (Stand: 28.02.2010)

`Authorizer` und davon abgeleiteten Klassen abgelöst. Das Prinzip, diese als Ketten von Filtern zu verknüpfen und vor eine Ressource zu „schalten“, ist erhalten geblieben.

Die Architektur der Beispiel Web Anwendung für die Umsetzung mit Restlet 2.0 wird in Abbildung 4.6 aufgezeigt.

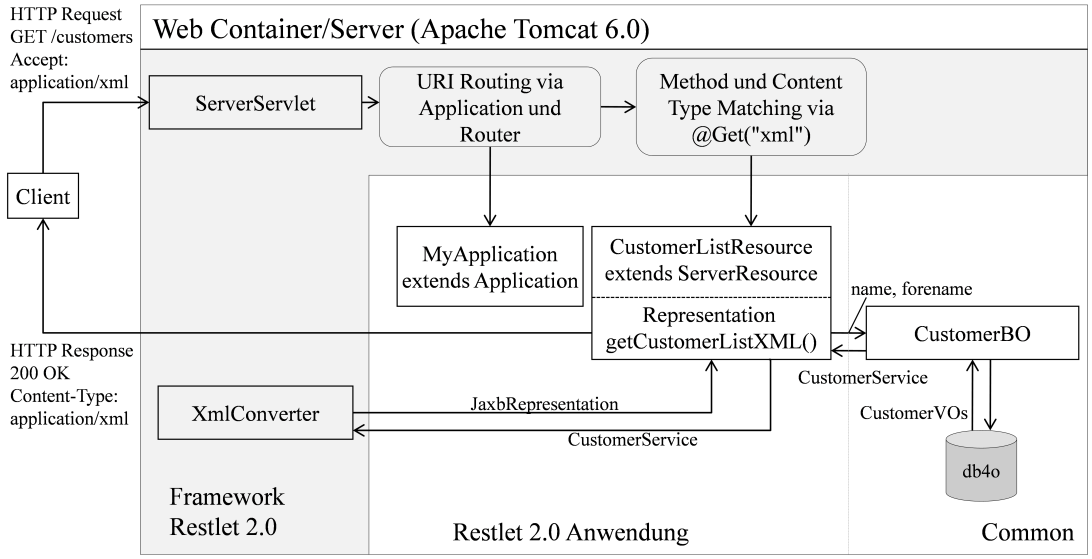


Abbildung 4.6.: Architektur der Realisierung mit Restlet 2.0

Ressourcenklassen in Restlet 2.0 erweitern nun die Klasse `ServerResource`. Der Konstruktor bekommt keine Parameter übergeben. Die Objekte `Context`, `Request` und `Response` werden der Ressource über die geerbte Methode `init` injiziert, die nach Instantiierung der Ressource aufgerufen wird. Dort wird auch die neue Methode `doInit` aufgerufen. Diese Methode ist dafür gedacht, Aufgaben zu übernehmen, die ansonsten in jeder Ressourcenmethode zu Beginn ausgeführt würden, wie das Lesen der Ressource aus einer Datenbank oder einem Dateisystem, oder das Auswerten von Query-Parametern. Nach dem Einlesen der Ressource, kann über das geerbte Attribut `existing` gekennzeichnet werden, ob die Ressource vorhanden ist oder nicht. Bei Setzen des Wertes auf `false` wird die Ressourcenmethode nicht mehr aufgerufen, sondern stattdessen vom Framework der Statuscode *404 Not Found* an den Client übermittelt. Die Klasse `ServerResource` enthält vier weitere, teils von Restlet 1.1 bekannte, Attribute. Das Attribut `annotated` kennzeichnet, ob die neuen Methoden-Annotationen verwendet werden sollen. Die Kennzeichnung, ob das Framework die Content Negotiation übernehmen soll, geschieht mit dem Attribut `negotiated`. Mit dem Attribut `conditional` kann gesteuert werden, ob das Framework die Behandlung bedingter Requests übernehmen soll. Der Standardwert dieser vier Attribute vom Typ `boolean` ist

`true`. Die Liste `variants` steht weiterhin zur Verfügung, wird aber offensichtlich nicht mehr verwendet[46]. Stattdessen kann das Format als Wert der neuen Annotationen `@Get`, `@Post`, `@Put`, `@Delete` und `@Options` angegeben werden. Hierbei ist die Angabe mehrerer Formate möglich, indem diese durch das Zeichen `'|'` getrennt hintereinander angegeben werden. Bei den Annotationen `@Post` und `@Put` ist die Angabe von Formaten für die eingehende und für die ausgehende Repräsentation möglich. Dies erfolgt durch die Angabe des Zeichens `':'` zwischen den Formaten. Die anzugebenden Formate entsprechen jedoch nicht den offiziellen MIME-Typen, sondern den Abkürzungen, die in Restlet 1.1 bereits für die Angabe der Varianten verwendet wurden.

Durch die an JAX-RS angelehnten Namen der Annotationen wird in Restlet 2.0 die Abstraktion vom Protokoll nicht intensiviert. Aufgrund der Möglichkeit, Ressourcenmethoden durch die Annotationen zu kennzeichnen und einem HTTP-Verb zuzuordnen, werden dafür keine Methoden mehr überschrieben. Ebenso ist es möglich, mehrere Methoden pro Verb zu definieren, um damit verschiedene Repräsentationsformate zu behandeln.

Die für Restlet 1.1 im Abschnitt 4.6.1 beschriebenen Methoden `allow<HTTP-Verb>`, mit der die Zulässigkeit von HTTP-Verben gesteuert werden konnte, sind durch die Annotationen überflüssig geworden. Das Framework ermittelt diese Information aus der Existenz der entsprechenden Annotation in einer Ressourcenklasse. Existiert für ein Verb keine Methode mit der jeweiligen Annotation, beantwortet das Framework den Request mit *405 Method Not Allowed*. Dies betrifft auch die Methode OPTIONS, sie wird mit Restlet 2.0 nicht mehr vom Framework automatisch beantwortet, sondern muss explizit implementiert und mit der Annotation `@Options` versehen werden.

Die mit `@Post` und `@Put` annotierten Methoden enthalten einen Parameter vom Typ `Representation`. Darin verbirgt sich die zu akzeptierende Repräsentation inklusive Metadaten, wie Format, Sprache usw. Dieser Typ kann ebenfalls als Rückgabewert der mit `@Get` annotierten Methoden verwendet werden.

4.7.2. Ressourcen und Schnittstelle

An der Aufteilung der Ressourcen in zugehörige Ressourcenklassen hat sich mit Restlet 2.0 gegenüber der Version 1.1 nichts geändert. Das Root-Restlet dient weiterhin als zentrale Stelle für die Zuordnung zwischen Ressourcen und URIs. Auch die hierarchische URI-Struktur kann genauso realisiert werden.

[46] In dem untersuchten Stand führte das Zufügen einer Variante, auch „*/*“, zu einem Statuscode *405 Method Not Allowed*.

In den Ressourcenklassen sind die für die jeweilige Ressource erlaubten Methoden implementiert und durch entsprechende Annotationen als solche gekennzeichnet. Der Wegfall der automatischen Antwort auf OPTIONS Requests ist ein überraschender Rückschritt gegenüber Restlet 1.1. Die Standardimplementierung für HEAD ruft weiterhin dieselbe Methode wie für GET auf und ignoriert den erstellten Entity-Body in der Response, was auch hier, sinnvoll bei großen Repräsentationen, überschrieben werden kann.

```
1
2  public class CustomerListResource extends ServerResource {
3
4    ...
5
6    /**
7     * Legt einen neuen Kunden an.
8     */
9    @Post("xml:xml")
10   public void acceptCustomer(Representation entity) throws ResourceException {
11
12     try {
13       final CustomerVO customerVO =
14           new CustomerBO(Framework.RESTLET20).create(
15               (entity == null ? null : entity.getText()));
16       setStatus(Status.SUCCESS_CREATED);
17       setLocationRef(getRequest().getResourceRef().getIdentifier() +
18           "/" + customerVO.getId());
19     } catch (CommonException e) {
20       throw ExceptionUtil.handleCommonException(e);
21     } catch (IOException e) {
22       throw ExceptionUtil.handleCommonException(
23           new InternalServerErrorException(
24               "XML konnte nicht ermittelt werden."));
25     }
26   }
27
28   ...
29
30 }
```

Listing 4.9: Implementierung der Methode POST der Ressource Kundenliste

Die Methode `acceptCustomer` der Klasse `CustomerListResource` ist ihrem Pendant in Restlet 1.1 immer noch sehr ähnlich, sie wird im Codebeispiel 4.9 aufgezeigt. Als Parameter erhält die Methode unverändert die Repräsentation einschließlich deren Metadaten. Die Überprüfung des Formats ist durch die Angabe in der Annotation nicht mehr notwendig. Für die Erzeugung des `ValueObject`s wird dem `BusinessObject` die vom Client übermittelte Repräsentation mittels der Methode `getText` übergeben. Im Unterschied zu Restlet 1.1 muss zuvor geprüft werden, ob überhaupt eine Repräsentation übergeben wurde. In Restlet 2.0 ist es nun laut Dokumentation möglich, POST und PUT mit einem leeren Entity-Body aufzurufen, was zuvor vom Framework mit einem entsprechenden Statuscode abgelehnt wurde. Tatsächlich ist die Reaktion auf den Versuch, einen leeren Body per PUT zu schicken, ein Statuscode *500 Internal Server*

Error.[47]

Die Methoden zum Setzen des Statuscodes *201 Created* und des Headers `Location` werden in der Klasse `ServerResource` durch eigene Methoden gekapselt, die hier aufgerufen werden. Eine gefangene `CommonException`, welche bei der Anlage des Kunden auftreten kann, wird wie bei Restlet 1.1 von `ExceptionUtil` in eine `ResourceException` mit einem passenden Statuscode und einem Freitext umgewandelt.

Für die Umsetzung des Fallbeispiels mit Restlet 2.0 wurde für die Authentifizierung die Variante, einen Filter im Pfad vor der Ressource Kundenliste zu verketten, gewählt. Hierzu wurde zunächst eine Ableitung der Klasse `ChallengeAuthenticator` erstellt, die die Methode `authenticate` überschreibt. Diese liefert immer `true`, außer für DELETE-Requests, bei denen an die super-Methode delegiert wird. Im Root-Restlet wird zunächst eine Instanz der Klasse `MapVerifier` erzeugt und dieser sämtliche, d.h. im Fallbeispiel genau eine, Benutzer-Passwort-Kombinationen bekanntgemacht. Anschließend wird `MyChallengeAuthenticator` instanziiert, wofür wie beim `Guard` in Restlet 1.1 neben dem Kontext das Authentifizierungsschema und der Wert für `realm` benötigt wird. Die Menge der Verifier wird dem Authenticator übergeben. Außerdem wird dem Authenticator das nächste Glied der Kette, hier die Ressourcenklasse `CustomerListResource` mitgeteilt. Bei der URI-Zuordnung im Router wird die URI /customers anstatt mit der Ressourcenklasse mit dem Authenticator verknüpft.

4.7.3. Statushaltung und Caching

Das Prinzip, Ressourcenklassen pro Request zu instanziieren und dadurch die Zustandslosigkeit des Services zu gewährleisten, bleibt in Restlet 2.0 unverändert.

Die Response stellt jetzt Methoden bereit, mit denen das Cache-Verhalten von Ressourcen gesteuert werden kann. Dies geschieht mit Hilfe der Klasse `CacheDirective`, welche vom Framework in Einträge des Headers `Cache-Control` umgewandelt werden.

Für die Methoden GET, PUT und DELETE sind weiterhin bedingte Zugriffe möglich. Neu hierbei ist, dass dafür nicht mehr zwingend die Repräsentation erzeugt werden muss, wie es in Restlet 1.1 der Fall war. Stattdessen kann in den Ressourcenklassen die Methode `getInfo` überschrieben werden. Die Methode liefert ein Objekt des Typs `RepresentationInfo`, der Superklasse von `Representation`. Dieses Metadaten-Objekt enthält die Attribute Änderungsdatum und ETag, anhand derer die Bedingungsprüfung durchgeführt werden kann. Diese Möglichkeit funktioniert bisher allerdings nur auf dem Papier, da das Framework die Methode während der Umsetzung des Fallbeispiels nicht

[47] Für den Fehler wurde im Bug Tracking System von Restlet ein Ticket mit der URI `http://restlet.tigris.org/issues/show_bug.cgi?id=1043` eingestellt.

aufgerufen hat. Stattdessen wird bei GET die zugehörige Methode aufgerufen, mit der die Repräsentation erzeugt wird.[48] Bei PUT wird tatsächlich ebenfalls die zugehörige Methode aufgerufen, bevor die Prüfung durchgeführt wird. Dies führt dazu, dass die Änderungen durchgeführt werden und der Client außerdem die Antwort *404 Not Found* bekommt.[49] Der Fehler bei der Prüfung des Änderungsdatums war in dem untersuchten Stand noch vorhanden, wurde inzwischen jedoch behoben, nachdem der Fehler gemeldet wurde (vgl. Abschnitt 4.6.3).

Bedingte Zugriffe können durch Zurücksetzen des Attributs `conditional` deaktiviert werden.

4.7.4. Repräsentationen und Verweise

Bei dem in diesem Abschnitt untersuchten Stand „Restlet Version 2.0 Milestone 6“ handelt es sich um eine noch nicht freigegebene Version. Mit dieser war eine Content Negotiation noch nicht befriedigend möglich. Deshalb wird hier die vermutlich beabsichtigte Funktionsweise beurteilt. Auch die Dokumentation ist noch nicht abgeschlossen, worauf im Restlet User Guide explizit hingewiesen wird.[50] Die Annahmen stützen sich teilweise auf Quellcode-Kommentare.

Die Annotationen `@Post` und `@Put` ermöglichen die Angabe einer Liste von Repräsentationsformaten empfangener Entitäten. Dadurch wird Content Negotiation im Vergleich zu Restlet 1.1 nun nicht mehr nur für ausgehende Formate unterstützt. Dies ist eine wertvolle Erweiterung des Frameworks.

In allen fünf Annotionen zur Kennzeichnung von Ressourcenmethoden kann eine Liste akzeptierter Repräsentationsformate angegeben werden. Hierfür müssen offensichtlich die Restlet-spezifischen Abkürzungen verwendet werden. Für die Erstellung des Response-Body müssen die Methoden, nicht unbedingt einen Rückgabewert vom Typ `Representation` liefern, wie es in Restlet 1.1 der Fall ist. Mit Restlet 2.0 kann hier ein beliebiges Objekt zurückgegeben werden. Dies kann ein einfacher `String` oder beispielsweise auch ein JAXB-Objekt sein. Das Framework enthält einige Hilfsklassen, die Converter heißen, und ermittelt automatisch den am besten geeigneten Converter, mit dem das Objekt in den Typ `Representation` transformiert werden kann. In dem betrachteten Stand gab es vier Converter: `XmlConverter` für XML, `JaxbConverter`, der derzeit lediglich XML konvertiert, aber vermutlich für weitere Formate erweitert

[48] Für den Fehler wurde im Bug Tracking System von Restlet ein Ticket mit der URI `http://restlet.tigris.org/issues/show_bug.cgi?id=1047` eingestellt.
[49] Für den Fehler wurde im Bug Tracking System von Restlet ein Ticket mit der URI `http://restlet.tigris.org/issues/show_bug.cgi?id=1044` eingestellt.
[50] URI: `http://wiki.restlet.org/docs_2.0/2-restlet.html` (Stand: 27.2.2010)

wird, `JsonConverter` für JSON, und `DefaultConverter` für alle anderen Formate. Es können jedoch eigene Converter erstellt werden.

Da der JAXB-Converter bisher nur XML konvertiert[51], kann nicht einfach ein Objekt vom Typ `CustomerService` als Rückgabe dienen, wie es angenommen wird und bei JAX-RS der Fall ist. Stattdessen zeigt das folgende Codebeispiel 4.10 die Realisierung durch zwei Methoden `getCustomerListXML` und `getCustomerListJSON` in der Klasse `CustomerListResource`, mit der das gewünschte Verhalten erzielt werden konnte. Die Query-Parameter `name` und `forename` wurden bereits in der Methode `doInit` extrahiert und als Klassenattribute bereitgestellt, ebenso wie das Objekt `customerService`. Die Methoden erzeugen die jeweilige Repräsentation und geben diese zurück. Restlet setzt im Erfolgsfall automatisch den Statuscode *200 OK*.

```
1
2  public class CustomerListResource extends ServerResource {
3
4    ...
5
6    /**
7     * Erstellt eine Kundenliste, filtert diese anhand des Namens
8     * und des Vornames und gibt diese im XML-Format zurück.
9     */
10   @Get("xml")
11   public Representation getCustomerXML() throws ResourceException {
12     try {
13       return new JaxbRepresentation<CustomerService>(
14           new CustomerBO(Framework.RESTLET20)
15               .getCustomerServiceByQueryParams(name, forename));
16     } catch (CommonException e) {
17       throw ExceptionUtil.handleCommonException(e);
18     }
19   }
20
21   /**
22    * Erstellt eine Kundenliste, filtert diese anhand des Namens
23    * und des Vornames und gibt diese im JSON-Format zurück.
24    */
25   @Get("json")
26   public Representation getCustomerJSON() throws ResourceException {
27     try {
28       return new JsonRepresentation(
29           new CustomerBO(Framework.RESTLET20)
30               .getCustomerServiceByQueryParams(name, forename));
31     } catch (CommonException e) {
32       throw ExceptionUtil.handleCommonException(e);
33     }
34   }
35
36   ...
37
38 }
```

Listing 4.10: Implementierung der Methode GET der Ressource Kundenliste

[51] Für den Fehler wurde im Bug Tracking System von Restlet ein Ticket mit der URI `http://restlet.tigris.org/issues/show_bug.cgi?id=1048` eingestellt.

4.7.5. Zusammenfassung und Bewertung

Die Komplexität und Mächtigkeit von Restlet 2.0 hat im Vergleich zur Vorgängerversion sicher nicht abgenommen. Das Framework wurde um einige Funktionalitäten erweitert. Die Erstellung der Ressourcenklassen, welche für die Beispielanwendung im Fokus standen, ist durch die Einführung von Annotationen modernisiert worden. Dank der umfassenden Refactoringmaßnahmen, beispielsweise das Entfernen veralteter Klassen, ist das Framework geordneter und übersichtlicher.

Die Bewertung in Tabelle 4.6 bezieht sich auf die angenommene Arbeitsweise nach Freigabe der Version. Wegen des unfertigen Softwarestands ist eine Beurteilung nicht abschließend möglich.

1. Ressourcen-Implementierung	++	gute Aufteilung der Ressourcen, Spezialisierung einer Frameworkklasse
2. URI-Zuordnung	++	Zuordnung in Root-Restlet, Hierarchie lässt sich durch Variablen in URI-Templates abbilden
3. Schnittstelle/Methoden	+	alle Methoden sind umsetzbar, Annotationen zur Kennzeichnung, allerdings keine automatische Beantwortung von OPTIONS-Requests mehr
4. Sicherheit/Authentifizierung	++	diverse Möglichkeiten der Authentifizierung, keine Angaben in web.xml notwendig
5. Zustandslosigkeit	++	Framework hält keinen Anwendungsstatus
6. Caching	+	Response enthält Methoden zur einfachen Manipulation des Cache-Control Headers
7. bedingte Requests	++	durch zusätzliche Methode ermittelt das Framework Datum bzw. ETag ohne eine Repräsentation erzeugen zu müssen, Prüfung erfolgt automatisch
8. Repräsentationen	++	Standard-Repräsentationen stehen zur Verfügung, Umfang ungewiss, laut Release-Notes aber umfangreich
9. Content Negotiation	++	bequeme Zuordnung per Annotation, Framework übernimmt Auswahl einer passenden Methode und eines geeigneten Converters
10. Verweise auf andere Ressourcen	o	keine besondere Unterstützung

Tabelle 4.6.: Stärken und Schwächen von Restlet 2.0

Die Aufteilung der Ressourcenklassen und ihre Zuordnung zu URIs geschieht übersichtlich an einer zentralen Stelle. Auch die Abbildung der Ressourcen-Hierarchie ist durch die Angabe entsprechender URI-Templates gut möglich. Dieser Aspekt ist gegenüber Restlet 1.1 unverändert geblieben. Die Strukturierung der Methoden in den Ressourcenklassen kann nun etwas freier geschehen. Es gibt keine vorgegebenen Methoden, die überschrieben werden müssen.

Die Content Negotiation wurde erweitert, so dass jetzt auch zu verarbeitende Formate für PUT und POST unterstützt werden. Der Umfang der Converter-Klassen ist zu diesem Zeitpunkt noch sehr eingeschränkt, wahrscheinlich wird das Framework an dieser Stelle noch wachsen. Dadurch wird die Grundlage geschaffen, Objekte möglichst komfortabel in das gewünschte Repräsentationsformat zu konvertieren. Durch die Möglichkeit, beliebige Objekte an das Framework zu übergeben, wird dem Entwickler viel Arbeit abgenommen, auch wenn dies zu diesem Zeitpunkt nur eingeschränkt funktioniert. Die Verlinkung der Ressourcen wird allerdings auch mit Restlet 2.0 nicht explizit unterstützt.

4.8. Spring 3 MVC

Unter dem Begriff *Spring* werden mehrere Projekte vereint. Das Spring-Framework stellt die führende Plattform für verschiedene Schwester-Projekte[52] dar. Beispiele hierfür sind Spring Web Flow, Spring Web Services (Spring WS) und Spring Security. Das Kernprojekt besteht aus mehreren Komponenten. Eine dieser Komponenten ist Spring MVC, ein selbstständiges Framework für Web Anwendungen, vergleichbar mit Struts[53], WebWork[54] oder Tapestry[55].

Auf den ersten Blick erscheint das Framework Spring WS als die geeignete Wahl, um den Web Service der Beispielanwendung zu realisieren. Tatsächlich dachte das Entwickler-Team zunächst darüber nach, die REST-Fähigkeit in Spring WS zu integrieren. Nach der Entwicklung eines Prototypen wurde diese Idee jedoch wieder verworfen und Spring MVC als das passendere Framework für REST erkannt. Zu der Zeit wurde das annotationenbasierte Modell für Spring MVC entworfen, während gleichzeitig die Spezifikation für JAX-RS entwickelt wurde. Deshalb war die nächste Idee, die MVC- und die JAX-RS-Annotationen zu kombinieren, was jedoch aus technischen Gründen ebenfalls wieder verworfen wurde. Letztlich wurde die Spring-REST-Lösung in Spring MVC integriert, ohne JAX-RS zu implementieren. Mit dem Verweis[56] auf die drei bestehenden JAX-RS-

[52] URI: `http://www.springsource.org/projects`
[53] URI: `http://struts.apache.org/`
[54] URI: `http://www.opensymphony.com/webwork/`
[55] URI: `http://tapestry.apache.org/`
[56] URI: `http://blog.springsource.com/2009/03/08/rest-in-spring-3-mvc/` (Stand: 28.02.2010)

Implementierungen Jersey, RESTEasy und Restlet ist nicht geplant, mit Spring eine weitere Implementierung hinzuzufügen.

Das Sub-Projekt Spring MVC wird hier mehr oder weniger losgelöst vom restlichen Spring Framework untersucht und betrachtet. Wie der Name bereits erkennen lässt, implementiert Spring MVC das MVC[57]-Pattern oder exakter gesagt das MVC2-Pattern für Webanwendungen. Der MVC Ansatz ist sehr geeignet für die Verarbeitung von Ressourcen nach REST. Das Model entspricht der internen Darstellung der Ressource. Eine View ist eine Repräsentation der Ressourcen, die Umwandlung geschieht abhängig von der jeweiligen View nach vorgegebenen Regeln. Die Controller sind in diesem Fall die Ressourcenklassen, welche die Requests verarbeiten und Ressourcen manipulieren bzw. für die View bereitstellen.

In das Spring Framework ist REST nicht nur für Web Service-Implementierungen integriert worden, es bietet außerdem eine einfache API zur Erstellung von RESTful Clients. Hierbei handelt es sich um die Klasse `RestTemplate`, welche für jedes Verb eine Methode mit den wichtigsten Parametern für diesen Request enthält.

Der in diesem Kapitel untersuchte Stand des Spring Frameworks ist Version 3.0.0.

4.8.1. API Implementierung

Wie bei einigen anderen Frameworks erfolgt die Entwicklung bei Spring ebenfalls in POJOs. Ressourcenklassen werden bei Spring MVC Controller genannt und durch die Annotation `@Controller` als solche gekennzeichnet. Dadurch ist es dem Framework möglich, nach Ressourcenklassen zu scannen. Für den Betrieb im Servlet Container muss, wie bei den anderen Frameworks auch, die Servlet-Klasse bekannt gemacht und einem URI-Pattern zugewiesen werden. Bei Spring ist dies die Klasse `org.springframework.web.servlet.DispatcherServlet`. Zusätzlich wird für das Servlet eine sogenannte Spring Konfigurationsdatei eingerichtet, welche unter anderem das Basis-Package enthält, in dem rekursiv nach Ressourcenklassen gesucht wird. Hier wird auch die Konfiguration der ViewResolver, Views, Converter, diverser Handler-Klassen etc. vorgenommen. ViewResolver dienen der Ermittlung einer passenden View, während eine View für die Erzeugung der Repräsentation zuständig ist. Converter übernehmen beispielsweise das Unmarshalling von Nachrichten in Model-Klassen. Handler behandeln verschiedene Elemente, wie Exceptions oder Annotationen.

Die Architektur der Beispiel Web Anwendung für die Umsetzung mit Spring MVC wird in Abbildung 4.7 aufgezeigt.

[57] Model-View-Controller

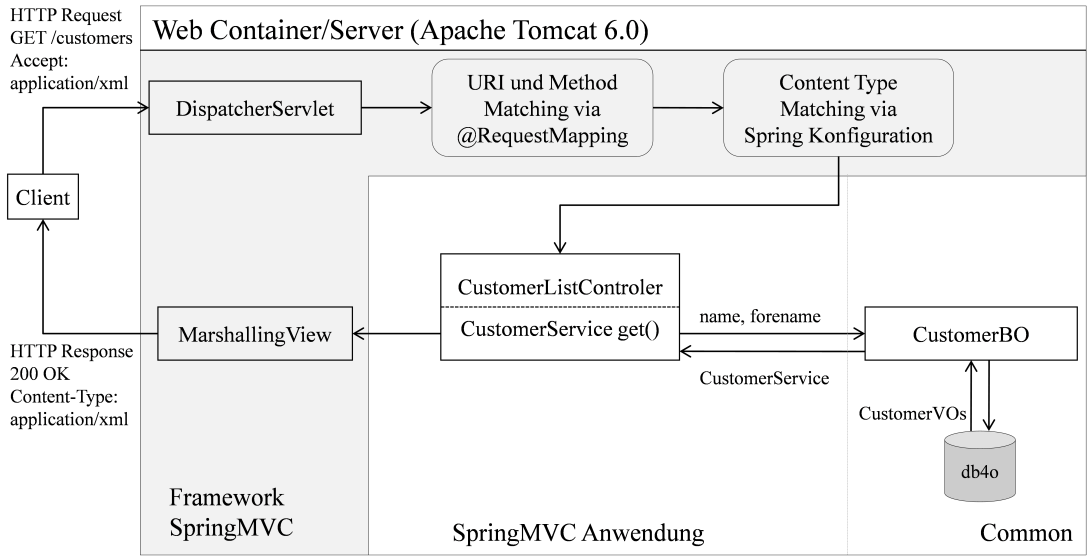


Abbildung 4.7.: Architektur der Realisierung mit Spring MVC

In der Einleitung dieses Abschnitts wurde die Entstehung der REST-Annotationen in Spring MVC erwähnt. Sie sind nicht mit den JAX-RS-Annotationen identisch, zeigen jedoch einige Parallelen. Eine wichtige Annotation ist `@RequestMapping`. Wie der Name bereits sagt, ist ihr Zweck, den Request auf die am besten geeignete Ressourcenklasse bzw. -methode zu mappen. Sie kann an Klassen und an Methoden geschrieben werden und mappt den Request anhand mehrerer Kriterien. Diese Kriterien sind URI-Templates, HTTP-Methoden, Request-Parameter und Request-Header. Wie bei den JAX-RS-Annotationen werden die Klassen-Annotationen an die Methoden-Annotationen vererbt, sie können dort überschrieben werden. URI-Templates an der Methode sind relativ zu denen an der Klasse definierten zu verstehen. Auch bei Spring dürfen die URI-Templates beliebige Wildcards und auch Template-Variablen enthalten. Letztere werden mit Hilfe der Annotation `@PathVariable` für Parameter Methoden bekanntgemacht. Fehler beim Casten auf Werte vom Typ `Integer`, z.B. die Kunden-ID, werden von Spring mit dem Statuscode *400 Bad Request* beantwortet. Weitere Informationen können durch entsprechende Parameter-Annotationen an Methoden übergeben werden, wie `@RequestBody`, `@RequestHeader`, `@RequestParam` oder `@CookieValue`.
Mit dem Rückgabewert von Methoden versucht Spring, die View zu erstellen. Dabei kann die Rückgabe verschiedene Typen annehmen, häufig wird der Typ `ModelAndView` verwendet, der die Daten im Model und wenigstens den Namen der darzustellenden View enthält. Gibt die Methode einen Wert vom Typ `String` zurück, wertet Spring diesen als Namen der View, beispielsweise einer jsp-Datei. Dies zeigt die ursprüngliche und hauptsächliche Verwendung von Spring MVC für Anwendungen mit einem webba-

sierten Benutzerinterface. Soll die Rückgabe nicht als Viewname, sondern als Entity-Body der Response verstanden werden, kann dies durch die Methoden-Annotation `@ResponseBody` gekennzeichnet werden. Eine weitere Methoden-Annotation ist `@ResponseStatus`, womit der Statuscode in der Response gesetzt werden kann. Im Fehlerfall kann stattdessen eine Exception geworfen werden. Zur Vereinfachung des Exception-Handlings können Methoden mit der Annotation `@ExceptionHandler` und der Angabe, welche Exceptions die Methode bearbeiten soll, versehen werden. So werden wiederkehrende try-catch-Blöcke vermieden.
Es gibt jedoch auch Annotationen, die in Hinblick auf REST eher fragwürdig sind, wie `@SessionAttributes`. Mit dieser Annotation werden Attribute der Klassen über mehrere Requests, für eine Sitzung, gespeichert. Dies verletzt jedoch das Prinzip der Zustandslosigkeit des Servers. Auch hieran ist zu erkennen, dass REST nachträglich in Spring MVC aufgenommen wurde.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans ...>
3    <context:component-scan base-package="de.q5050030.spring.controller"/>
4
5    <bean class="org.springframework. ... .ContentNegotiatingViewResolver">
6      <property name="mediaTypes">
7        <map>
8          <entry key="xml" value="application/xml"/>
9          <entry key="json" value="application/json"/>
10       </map>
11     </property>
12
13     <property name="defaultViews">
14     <list>
15       <bean class="org.springframework. ... .json.MappingJacksonJsonView"/>
16       <bean class="org.springframework.web.servlet.view.xml.MarshallingView">
17         <property name="marshaller" ref="marshaller"/>
18       </bean>
19     </list>
20   </property>
21   </bean>
22
23   <oxm:jaxb2-marshaller id="marshaller">
24     <oxm:class-to-be-bound name="de.q5050030.common.model.CustomerService"/>
25   </oxm:jaxb2-marshaller>
26
27   <bean class="org.springframework. ... AnnotationMethodHandlerAdapter">
28     <property name="messageConverters">
29       <list>
30         <ref bean="marshallingHttpMessageConverter"/>
31       </list>
32     </property>
33   </bean>
34
35   <bean id="marshallingHttpMessageConverter" class=
36     "org.springframework.http.converter.xml.MarshallingHttpMessageConverter">
37     <property name="marshaller" ref="marshaller"/>
38     <property name="unmarshaller" ref="marshaller"/>
39   </bean>
40 </beans>
```

Listing 4.11: Auszug aus Spring Konfigurationsdatei

Die Content Negotiation wird durch das Framework durchgeführt und muss nicht selbst implementiert werden. Hierfür könnte beispielsweise folgende Annotation `@RequestMapping(headers="Content-Type=application/xml")` verwendet werden. Für das Fallbeispiel wurde eine andere Realisierung bevorzugt. In der Spring Konfigurationsdatei wurde ein Eintrag für die Klasse `ContentNegotiatingViewResolver` vorgenommen und mit den angebotenen Formattypen verknüpft. Gleichzeitig wurde für jeden Formattyp eine Default-View für die jeweilige Darstellung der Ressource definiert. Für das Unmarshalling empfangener Repräsentationen wurde ein entsprechender Converter konfiguriert. Außerdem wurde das Model, hier die Klasse `CustomerService` eingerichtet und als JAXB-Objekt gekennzeichnet. Diese Konfiguration ermöglicht Spring, jeweils die passende Konvertierung durchzuführen. Ein Auszug der Spring Konfigurationsdatei ist im Listing 4.11 aufgeführt.

4.8.2. Ressourcen und Schnittstelle

Die Abbildung der Ressourcen auf zugehörige Ressourcenklassen ist ohne weiteres möglich. Die Verknüpfung einer Ressourcenklasse mit einem URI-Template geschieht mit der Klassen-Annotation `@RequestMapping`. Die Abbildung einer Ressourcen-Hierarchie ist durch erlaubte Template-Variablen ebenso realisierbar. In derselben Annotation können auch die mit der Klasse bereitgestellten HTTP-Methoden aufgeführt werden. Das Framework entscheidet dann, welche Methode aufgerufen wird. Existiert nur eine Java-Methode, die nicht durch eine eigene Annotation die Klassen-Annotation überschreibt, wird diese verwendet. Dies könnte verwendet werden, um nicht erlaubte HTTP-Verben in einer dafür vorgesehen Methode zu behandeln. Bei mehreren in Frage kommenden Methoden wird beispielsweise der Name in die Entscheidung mit einbezogen. Für die Implementierung des Fallbeispiels wird die explizite Zuordnung der HTTP-Verben in den Methoden-Annotationen bevorzugt. Die Verben GET und HEAD werden hierbei derselben Methode zugewiesen, Spring sorgt automatisch dafür, dass für HEAD die Rückgabe eines Entity-Body unterdrückt wird. Nicht implementierte Methoden werden vom Framework mit dem Statuscode *405 Method Not Allowed* beantwortet.
Eine Methode für OPTIONS kann nicht ohne weiteres implementiert werden. Die Beantwortung wird ebenfalls automatisch vom Framework übernommen. Allerdings werden im Header `Allow` nicht nur die tatsächlich implementierten Methoden aufgeführt, sondern sämtliche HTTP-Methoden mit Ausnahme von CONNECT. Um eine eigene Implementierung für OPTIONS und damit eine den Tatsachen entsprechende Rückgabe zu realisieren, muss die automatische Antwort deaktiviert werden. Dazu muss das zentrale `DispatcherServlet` überschrieben werden. Insgesamt kann für diese Methode das gewünschte Ergebnis erzielt werden, ist allerdings mit etwas Aufwand verbunden.

Wie mit Spring die Anlage eines neuen Kunden umgesetzt wird, zeigt die Methode `post` der Klasse `CustomerListController` im Codebeispiel 4.12. Über Annotationen wird die Methode dem Verb POST mit dem Statuscode *201 Created* im Erfolgsfall zugeordnet. Der erste Parameter der Methode ist der durch eine Annotation gekennzeichnete Entity-Body des Requests, der bereits durch den festgelegten Unmarshaller in die Model-Klasse überführt wurde. Das `BusinessObject` erzeugt das `ValueObject` und persistiert es. Aus der Request-URI und der neu erzeugten Kunden-ID wird die URI der neuen Ressource gebildet und direkt in den Response-Header `Location` geschrieben. Hierfür werden sowohl Request als auch Response als weitere Parameter der Methode benötigt. Eine Rückgabe erfolgt nicht, da für diese Methode keine Repräsentation in der Response erfolgen soll.

```
1
2  @Controller
3  @RequestMapping(value = UriPathParts.CUSTOMERS_HREF)
4  public class CustomerListController extends BaseController {
5
6    ...
7
8    /**
9     * Legt einen neuen Kunden an.
10    */
11   @RequestMapping(method = RequestMethod.POST)
12   @ResponseStatus(value = HttpStatus.CREATED)
13   public void post(@RequestBody CustomerService customerService,
14                    HttpServletRequest request,
15                    HttpServletResponse response)
16       throws CommonException {
17     final CustomerVO customerVO = new CustomerBO(Framework.SPRINGMVC).create(
18         customerService.getCustomer());
19     response.setHeader("Location", request.getRequestURL().toString() + "/"
20         + customerVO.getId());
21   }
22
23   ...
24
25 }
```

Listing 4.12: Implementierung der Methode POST der Ressource Kundenliste

Die in dieser Methode geworfenen Exceptions werden in einer separaten Methode verarbeitet, die dafür mit der Annotation `@ExceptionHandler(CommonException.class)` gekennzeichnet wurde. Da diese Verarbeitung für alle Ressourcen einheitlich erfolgen soll, wird die Methode in der Klasse `BaseController` implementiert, von der alle Ressourcenklassen der Beispielanwendung ableiten. Sie setzt in der Response den der Exception zugewiesenen Statuscode und die Fehlermeldung in den Body. Tritt hierbei ein Fehler auf, wird der Statuscode in *500 Internal Server Error* geändert.

Die Authentifizierung erfolgt für Spring MVC über den normalen Servlet-Security-Mechanismus im Deployment Descriptor. Spring MVC enthält keine Komponenten, die

eine Authentifizierung ermöglichen, hierfür steht das Framework Spring Security zur Verfügung. Dieses wird meist verwendet, wenn das Spring Framework eingesetzt wird. Da es sich hierbei um ein selbstständiges Framework unabhängig von Spring MVC handelt, wurde dies aus Komplexitätsgründen nicht integriert.

4.8.3. Statushaltung und Caching

Im Abschnitt 4.8.1 wurde bereits eine Annotation erwähnt, die in Bezug auf die Zustandslosigkeit des Services fraglich ist. Selbst ohne diese Annotation ist Spring nicht einwandfrei zustandslos, da hier nicht das Prinzip der „Pro-Request-Instantiierung“ verwendet wird. So ist es ohne weiteres möglich, Klassenattribute während einer Requestverarbeitung zu verändern und genau diese Werte während der Verarbeitung eines nächsten Requests wieder auszulesen. Das Framework stellt somit nicht die Zustandslosigkeit des Servers sicher. Dies ist ein weiterer Aspekt der zeigt, dass Spring MVC nicht von vornherein für die Erstellung von RESTful Web Services konzipiert wurde.

Für die Manipulation des Caching-Verhaltens von Ressourcenklassen bietet Spring MVC keine Unterstützung.

Bedingte GET-Requests hingegen können von Spring MVC automatisch behandelt werden. Zur Aktivierung wird in der Deployment Descriptor Datei ein Filter mit der Klasse `org.springframework.web.filter.ShallowEtagHeaderFilter` definiert. Dieser filtert GET-Requests, nutzt den Response-Body zur Berechnung eines MD5-Hashwerts und setzt diesen in den Response-Header `ETag`. Ist der Request-Header `If-None-Match` gefüllt, wird dieser mit dem frisch berechneten ETag verglichen und bei Übereinstimmung wird ein leerer Body mit dem Statuscode *304 Not Modified* zurückgegeben.

4.8.4. Repräsentationen und Verweise

Die Trennung von Model und View, also der internen Ressourcendarstellung und der äußeren Repräsentation, die elementarer Bestandteil von Spring MVC ist, wirkt sich sehr positiv auf diesen Aspekt von REST aus. Innerhalb der Controller-Methoden wird stets mit dem Model gearbeitet. Das Marshalling und Unmarshalling wird vom Framework übernommen. Eintreffende Repräsentationen für PUT und POST werden bei entsprechender Konfiguration in der Spring Konfigurationsdatei angenommen und abhängig vom Request-Header `Content-Type` mit dem passenden Converter in das Model transformiert. In der Beispielanwendung, die lediglich XML-Repräsentationen annimmt, wurde hierfür der `MarshallingHttpMessageConverter` eingesetzt. Die Konfiguration sorgt dafür, dass für das Mapping Springs „Object to XML mapping (OXM)“

Funktionalität benutzt wird, welche die ohnehin vorhandenen JAXB-Annotationen in der Model-Klasse ausnutzt.

Umgekehrt geben Methoden, die eine Repräsentation an den Client übertragen sollen, einfach das Model an das Framework zurück. Der konfigurierte `ContentNegotiatingViewResolver` wertet den Request-Header `Accept` aus und entscheidet dann selbst, welche View jeweils am besten geeignet ist. Hierfür sind zwei Default-Views eingerichtet: `MarshallingView` für XML und `MappingJacksonJsonView` für JSON. Der View-Resolver kann für die Content Negotiation auch Dateiendungen verwenden, da in den meisten Browsern der `Accept`-Header nicht beeinflusst werden kann. Wenn für die akzeptierten Repräsentationsformate keine geeignete View zur Verfügung steht, reagiert Spring MVC mit dem Statuscode *500 Internal Server Error* anstatt mit *406 Not Acceptable*.[58]

Das Codebeispiel 4.13 zeigt die Methode `get` der Klasse `CustomerListController`. Sie verarbeitet GET- und HEAD-Requests, was durch die Annotation gekennzeichnet wird. Der Responsecode für den Fall, dass die Ressource existiert, ist *200 OK*. Die Annotation `@RequestParam` markiert die beiden Query-Parameter `name` und `forename`, die zuvor vom Framework extrahiert wurden. Das Model-Objekt, welches die Liste der Kunden nach Name und Vorname eingeschränkt enthält, wird vom `BusinessObject` erzeugt und an das Framework zurückgegeben.

```
1
2  @Controller
3  @RequestMapping(value = UriPathParts.CUSTOMERS_HREF)
4  public class CustomerListController extends BaseController {
5
6    ...
7
8    /**
9     * Erstellt eine Kundenliste, filtert diese anhand des Namens
10    * und des Vornames und gibt diese zurück.
11    */
12   @RequestMapping(method = {RequestMethod.GET, RequestMethod.HEAD})
13   @ResponseStatus(value = HttpStatus.OK)
14   public CustomerService get(
15       @RequestParam(value = "name", required = false) String name,
16       @RequestParam(value = "forename", required = false) String forename)
17       throws CommonException {
18     return new CustomerBO(Framework.SPRINGMVC)
19         .getCustomerServiceByQueryParams(name, forename);
20   }
21
22   ...
23
24 }
```

Listing 4.13: Implementierung der Methode GET der Ressource Kundenliste

[58] Für die Fehler wurde im Bug Tracking System von Spring ein Ticket mit der URI `http://jira.springframework.org/browse/SPR-6894` eingestellt.

4.8.5. Zusammenfassung und Bewertung

Spring MVC wurde bei der Integration der REST-Fähigkeit deutlich von der JAX-RS Spezifikation geprägt. Dennoch gibt es einige Unterschiede.
Die Aufteilung der Ressourcenklassen erfolgt auch bei Spring intuitiv und gewollt. Die Annotation `@RequestMapping` bündelt in ihrer Bedeutung die komplette Zuordnung eines Requests zu einer Ressourcenklasse und -methode. Durch den Einsatz von URI-Templates ist die Bildung einer Ressourcen-Hierarchie und entsprechender URIs gut möglich.

Durch die Konfiguration eines Servlets erfolgt die Spezialisierung auf HTTP, mit Spring werden aber auch weitere Protokolle unterstützt. Die HTTP-Verben können ebenfalls über die Annotation einer Klasse oder besser einer Methode zugeordnet werden. Die Umsetzung von OPTIONS ist allerdings nicht ohne weiteres möglich. Die Standardimplementierung berücksichtigt nicht die tatsächlich erlaubten Methoden.

Eine Schwäche ist die vom Framework nicht explizit unterstützte Zustandslosigkeit des Services. Ressourcenklassen werden nach ihrer Verwendung nicht für den Garbage collector freigegeben, sondern für den nächsten Request wiederverwendet. Hier muss der Entwickler bewusst darauf achten, dieses REST-Prinzip nicht zu verletzen.

Die namensgebende Einhaltung des MVC-Design Patterns vereinfacht die Erzeugung der Repräsentationen deutlich. So sind Ein- und Rückgabeparameter der Methoden stets ein Model-Objekt, welches je nach Request bearbeitet wird. Das Marshalling und Unmarshalling wird automatisch vom Framework übernommen. Um weitere Repräsentationsformate zu unterstützen, brauchen die Ressourcenklassen nicht verändert werden, es müssen lediglich entsprechende Converter in der Spring Konfigurationsdatei eingerichtet werden.
Die Verlinkung der Ressourcen wird durch Spring nicht besonders unterstützt.

1. Ressourcen-Implementierung	++	gute Aufteilung der Ressourcen, Implementierung in POJOs
2. URI-Zuordnung	++	bequeme Zuordnung per Annotation, Hierarchie lässt sich durch Variablen in URI-Templates abbilden
3. Schnittstelle/Methoden	+	bis auf OPTIONS sind alle Methoden umsetzbar, Kennzeichnung per Annotation
4. Sicherheit/Authentifizierung	o	keine Unterstützung in Spring MVC, sondern durch weiteres Spring Framework
5. Zustandslosigkeit	-	Instanzen der Ressourcenklassen werden wiederverwendet
6. Caching	o	keine besondere Unterstützung
7. bedingte Requests	+	GET-Requests per ETag durch Filter in web.xml aktivierbar
8. Repräsentationen	++	einige Converter zur Erstellung der Repräsentationen, Konfiguration in separater Datei
9. Content Negotiation	+	Framework übernimmt Auswahl eines geeigneten Converters, aber ungeschicktes Verhalten im Fehlerfall
10. Verweise auf andere Ressourcen	o	keine besondere Unterstützung

Tabelle 4.7.: Stärken und Schwächen von Spring MVC

5. Vergleich

In diesem Abschnitt werden die Untersuchungsergebnisse für die einzelnen Frameworks miteinander verglichen. Hierfür werden wie im Kapitel 4 die Fragen des Fragenkatalogs betrachtet und die Umsetzung durch die Frameworks gegenübergestellt. Für den Vergleich wurde neben den Ausführungen des vorangegangenen Kapitels eine Reihe von Testfällen verwendet, die ausführlich in Anhang D beschrieben sind.

5.1. API Implementierung

Die Art und Weise, wie mit den Frameworks Ressourcenklassen implementiert werden, unterscheidet sich teilweise deutlich. So wird bei beiden Restlet-Versionen von einer Klasse abgeleitet, die bereits einen Großteil der Funktionalität abdeckt. Das Verhalten der Ressourcen kann durch Setzen von Attributen der geerbten Klasse und durch Überschreiben von Methoden relativ fein justiert werden. Andererseits wirkt die Fülle an Methoden sehr komplex, was den Einstieg und die Umsetzung erster Ressourcenklassen erschwert. In Restlet ist es notwendig eine zentrale Anwendungsklasse, das Root-Restlet zu implementieren. Ein hier definierter Router sorgt für das Routing von der Wurzel über mögliche Filter hin zu den Ressourcenklassen.

Bei der Servlet-API und JAX-WS wird jeweils ein Interface implementiert. Dieses enthält bei der Servlet-API je HTTP-Verb eine Methode, welche den Request und die Response übergeben bekommt. Die Struktur ist übersichtlich und wird klar vorgegeben, bei der Umsetzung wird jedoch kaum weitere Unterstützung geboten. Von Lesen, Parsen und Interpretieren der Header und Entity bis zum Schreiben des Response-Body und Setzen der Header erfolgt die Umsetzung auf einem niedrigen Abstraktionsniveau. Das Interface von JAX-WS stellt nicht einmal diese Methoden-Struktur bereit, sie enthält lediglich eine einzige Methode. Die Unterstützung, die das Framework bietet, ist so sehr auf XML-Web Services ausgerichtet, dass diese für die Erstellung eines RESTful Web Services nicht in Anspruch genommen wird.

Die drei anderen Frameworks Jersey, RESTEasy und Spring MVC kennzeichnen Ressourcenklassen durch Annotationen, anstatt eine Klasse zu erweitern oder ein Interface

zu implementieren. Durch die Implementierung der JAX-RS-Spezifikation sind die Ressourcenklassen bei Jersey und RESTEasy zunächst sehr ähnlich. Die Verwendung der Annotationen ist sehr intuitiv, wodurch schnell erste Erfolge erzielt werden. RESTEasy ergänzt das Angebot um weitere Annotationen.

Seit der Version 3.0 wird in Spring MVC ebenfalls mit Annotationen gearbeitet. Diese sind nicht identisch mit denen von JAX-RS, aber erkennbar an diese angelehnt. An verschiedenen Aspekten bei der Umsetzung zeigt sich, dass Spring MVC nicht von vornherein für REST konzipiert wurde. Erwähnenswert ist bei der Spring MVC-Umsetzung, dass eine Methode durch eine Annotation für die Verarbeitung bestimmter Exceptions bestimmt werden kann. So entfallen in den Methoden wiederkehrende try-catch-Blöcke, wodurch sie etwas kompakter werden. Außerdem erfolgt ein Teil der Implementierung in der Spring Konfigurationsdatei. Dadurch ist es beispielsweise möglich, weitere Repräsentationsformate anzubieten, ohne Java-Code zu ändern, wodurch das Kompilieren von Klassen und das Erstellen der Archiv-Dateien[1] entfällt. Andererseits wird dadurch die Implementierung in Java-Code und Konfiguration aufgeteilt, anstatt alle Aspekte in den Javaklassen zu bündeln.

Annotationen für die Kennzeichnung der Methoden werden seit der Version 2.0 ebenfalls in Restlet eingesetzt. Außerdem wurde das Framework einem gründlichen Refactoring unterzogen, wodurch die Ressourcenklassen insgesamt moderner und übersichtlicher wirken.

5.2. Ressourcen und Schnittstelle

Frage 1: Wie erfolgt die Implementierung von Ressourcen?
Die Implementierung der Ressourcen in je einer zugehörigen Ressourcenklasse ist mit allen Frameworks möglich. Eine Ausnahme hiervon ist für die Implementierung mit der JAX-WS RI möglich, wenn für verschiedene Bindungstypen eigene Ressourcenklassen erstellt würden. Allerdings wäre in diesem Fall die Herausforderung, mehrere Klassen, die unterschiedliche Repräsentationen umsetzen, so zu gestalten, dass für alle Repräsentationsformate ein identisches Verhalten erreicht wird. Der Fokus läge damit nicht mehr auf der Ressource, sondern auf dem Format.

Das JAX-WS-Framework gibt für die Realisierung der Methoden keinerlei Struktur vor. Das Interface enthält lediglich eine einzelne Methode. Bei der Servlet-API gibt es für jedes Verb eine Methode, die überschrieben wird, wodurch eine unabhängige Betrachtung der unterschiedlichen Requests gefördert wird. Das Konzept, vorgegebene Methoden zu überschreiben, prägt auch die Umsetzung mit Restlet 1.1. Durch die

[1] JAR- bzw. WAR-Dateien

Historie des Frameworks und die vielen Möglichkeiten, das Verhalten der Ressourcen zu steuern, gibt es hier sehr viele Methoden, die überschrieben werden können und teilweise sogar müssen. Die Komplexität des Frameworks existiert auch in Restlet 2.0 weiterhin. Die Kennzeichnung der Methoden durch Annotationen trägt jedoch dazu bei, dass die Ressourcenklassen übersichtlicher und durchschaubarer sind. Die Strukturierung kann etwas freier geschehen, Methodennamen werden nicht vorgegeben, dennoch geben die Annotationen eine Aufteilung nach HTTP-Verben vor. Dieser Ansatz wird auch bei den Implementierungen von Jersey, RESTEasy und Spring MVC verfolgt.

Frage 2: Wie geschieht die Zuordnung zwischen Ressourcen und ihren URIs?
Die Zuordnung der URIs zu den Ressourcen erfolgt bei Jersey, RESTEasy und Spring MVC innerhalb von Annotationen in den Ressourcenklassen. Dadurch entsteht eine enge Bindung der URI an die Ressource. Restlet 1.1 und 2.0 verwenden hierfür das zentrale Root-Restlet. Dies hat den Vorteil, dass die URI-Struktur übersichtlich an einer zentralen Stelle verwaltet wird. Beide Varianten ermöglichen die hierarchische URI-Struktur. Diese kann mit der Servlet-API und JAX-WS nicht realisiert werden, hier erfolgt die URI-Zuordnung in XML-Dateien. Immerhin geschieht die Verwaltung an einer zentralen Stelle und ermöglicht es, die URI-Struktur ohne Änderung des Java-Codes zu ändern. Dieser Vorteil wiegt jedoch das Fehlen der Hierarchie nicht auf.

Frage 3: Welche HTTP-Methoden können umgesetzt werden?
Nahezu alle Frameworks unterstützen die Umsetzung aller HTTP-Methoden, zumindest der sechs in dieser Untersuchung betrachteten. Die Servlet-API bietet für jede Methode eine Java-Methode, die überschrieben werden kann. HEAD und OPTIONS müssen nicht explizit implementiert werden, das Framework erzeugt eine korrekte Response. Die JAX-WS RI leitet mit Ausnahme von OPTIONS alle Requests an die Ressourcenklasse weiter, wodurch für diese Requests eine Umsetzung erfolgen kann. OPTIONS-Requests werden vom Framework beantwortet, allerdings ohne Einschränkung auf angebotene Methoden. Die JAX-RS-Implementierungen Jersey und RESTEasy bieten Annotationen für alle Methoden außer OPTIONS. Dies ist zum einen nicht notwendig, da hierfür automatisch eine korrekte Response erzeugt wird. Bei Bedarf ist es möglich, eigene Methoden-Annotationen zu ergänzen, so dass eine eigene Implementierung für OPTIONS möglich wäre. In JAX-RS 1.1 ist OPTIONS ebenfalls ergänzt worden. Auch Restlet 1.1 beantwortet OPTIONS-Requests automatisch, hierfür kann auch keine Methode überschrieben werden, ebensowenig wie für HEAD. Restlet 2.0 hingegen bietet eine Annotation für OPTIONS, dafür ist die automatische Beantwortung durch das Framework entfallen. Spring MVC beantwortet OPTIONS-Requests automatisch, allerdings ohne die angegebenen Methoden auf tatsächlich implementierte zu beschränken. Eine eigene Implementierung ist nicht möglich.

Methode	Servlet-API 2.5	JAX-WS RI 2.1.2	Jersey 1.0.3.1	RESTEasy 1.2.1	Restlet 1.1.7	Restlet 2.0 M6	Spring 3 MVC
GET	muss	muss	muss	muss	muss	muss	muss
POST	muss	muss	muss	muss	muss	muss	muss
PUT	muss	muss	muss	muss	muss	muss	muss
DELETE	muss	muss	muss	muss	muss	muss	muss
HEAD	kann	muss	kann	kann	auto	auto	muss
OPTIONS	kann	auto	auto	auto	auto	muss	auto

Tabelle 5.1.: Realisierung der Methoden

Tabelle 5.1 zeigt noch einmal die Implementierung der Methoden im Überblick. Dabei bedeutet *muss*, dass die Methode implementiert werden muss, damit Requests bearbeitet werden, *kann* heißt das Framework erzeugt eine automatische Antwort, es kann aber eine eigene Implementierung erfolgen und *auto* drückt aus, dass die automatische Antwort nicht überlagert werden kann.

Frage 4: Werden Sicherheitsmechanismen zur Authentifizierung angeboten?
Die Verwendung der Standard-Servlet-Security ist mit allen Frameworks möglich, wenn der Betrieb wie für diese Untersuchung im Servlet-Container erfolgt. Die zusätzlichen Abfragen in den Ressourcenklassen bei der Servlet-API, der JAX-WS RI und Jersey bringen keinen echten Sicherheitsgewinn. Überzeugender sind hier die Mechanismen von RESTEasy und Restlet. In RESTEasy werden die Methoden über Annotationen für definierte Rollen freigeschaltet. Restlet verfolgt ein komplett eigenes Konzept, bei dem keine Angaben in Konfigurationsdateien erfolgen müssen, sondern die Ressourcen über Guards geschützt werden. Spring MVC verfügt über keinen eigenen Authentifizierungsmechanismus. Das Spring Framework bietet hierfür das Framework Spring Security an.

5.3. Statushaltung und Caching

Frage 5: Verwaltet die Serverkomponente Anwendungsstatus der Clients?
Die Frameworks, die explizit für REST-konforme Anwendungen entworfen wurden, unterstützen aktiv die Zustandslosigkeit des Services. Ressourcenklassen werden für genau einen Request instanziiert und nach erfolgter Verarbeitung für den Garbage collector freigegeben. Außerdem werden keine Sessions unterstützt.

Das genaue Gegenteil ist bei den anderen Frameworks, also der Servlet-API, der JAX-WS RI und Spring MVC der Fall. Klassenattribute bleiben über mehrere Requests erhalten und Sessions werden ausdrücklich ermöglicht.

Frage 6: Unterstützt das Framework Caching?
Diese drei Frameworks Servlet-API, JAX-WS und Spring MVC bieten auch keine besondere Unterstützung, den `Cache-Control` Header zu verändern. Für dynamische Web-Seiten und SOAP-Web Services ist dies auch nicht entscheidend. Die Option, das Cache-Verhalten durch direkte Änderung von Headern zu beeinflussen, besteht jedoch in allen Frameworks.
Restlet 1.1 bietet auch keine explizite Möglichkeit der Steuerung des Verhaltens. Diese Schwäche wird mit Restlet 2.0 behoben. Hier wird eine separate Klasse dafür ergänzt. Mit Jersey wird dieser Aspekt über eine ähnliche Klasse abgedeckt, während RESTEasy eine Lösung durch zusätzliche Annotationen anbietet.

Frage 7: Inwieweit unterstützt das Framework bedingte Requests?
Etwas anders sieht die Verteilung der Frameworks bei der Frage nach bedingten Requests aus. Die Servlet-API bietet für bedingte GET-Requests bezogen auf das Änderungsdatum eine Methode an, die allerdings selbst implementiert werden muss. Den Aufruf der Methode, wenn der entsprechende Request-Header gesetzt ist, übernimmt jedoch das Framework. Jersey bietet eine Methode an, die allerdings explizit in den Ressourcenmethoden aufgerufen werden muss. Dieser können sowohl Änderungsdatum als auch ETag mitgegeben werden, das Framework vergleicht diese mit den Request-Headern. Der Aufruf ist in allen Methoden möglich, wodurch nicht nur bedingte GET-Requests unterstützt werden. Restlet 1.1 bietet Unterstützung für die Methoden GET, PUT und DELETE. Dafür enthalten die Repräsentationen Attribute für Änderungsdatum und ETag. Diese werden ausgewertet und die Requests gegebenenfalls automatisch beantwortet. Allerdings wird dadurch für die Prüfung die Repräsentation erzeugt. Um dies zu vermeiden, wird mit Restlet 2.0 eine separate Methode angeboten, die nur diese Metadaten enthält anstatt der kompletten Repräsentation. Der Mechanismus funktionierte allerdings in dem betrachteten Stand noch nicht. RESTEasy bietet eine Kombination aus Caching und bedingten GET-Requests an. Die Response von erfolgreichen GET-Requests werden in einem Cache gehalten und automatisch mit einem ETag versehen. Bedingte Requests werden mit diesem ETag abgeglichen und ohne Aufruf der Ressourcenmethode beantwortet, wenn dies möglich ist. Eine ähnliche Variante bietet Spring MVC. Hier werden Responses mit einem automatisch berechneten ETag versehen. Bei bedingten GET-Requests wird erneut ein ETag aus der Repräsentation berechnet, mit dem gelieferten ETag verglichen und automatisch beantwortet. Die JAX-WS RI bietet keine Unterstützung für bedingte Requests an.

5.4. Repräsentationen und Verweise

Frage 8: Müssen alle Repräsentationsformate explizit implementiert werden?
Die Servlet-API bietet keine Hilfe bei der Erstellung der Repräsentationen. Diese müssen selbst erzeugt und dann unmittelbar in den Response-Body geschrieben werden. Die Unterstützung von JAX-WS bezieht sich im Wesentlichen auf XML. Durch verschiedene Bindungen werden zwar auch verschiedene Formate unterstützt, jedoch sind die Ressourcenklassen damit buchstäblich an dieses Format gebunden. Die Spezifikation von JAX-RS gibt einige Standard-Provider vor, die angeboten werden müssen, wodurch einige Formate immer unterstützt werden. Jersey bietet darüberhinaus einige weitere Provider. Auch RESTEasy unterstützt mehr Formate als die Spezifikation verlangt, sogar einige mehr als Jersey. Es ist außerdem für beide Implementierungen möglich, eigene Provider zu ergänzen. Ein Schwerpunkt liegt hierbei auf der Erstellung von Repräsentationen aus JAXB-Elementen, die gut dafür geeignet sind, in verschiedene Formate umgesetzt zu werden. Die Erstellung der Repräsentation wird auch bei Restlet 1.1 vom Framework übernommen, allerdings muss hierbei explizit das gewünschte Format angesteuert werden. Durch die automatische Wahl eines geeigneten Converters für das vom Client gewünschte Format ist Restlet 2.0 bedeutend komfortabler. Hier können letztlich beliebige Objekte an das Framework zurückgegeben werden, die automatisch konvertiert werden. Ähnlich komfortabel wird bei Spring MVC über die Wahl eines geeigneten ViewResolvers entschieden. Hierbei ist es sogar möglich, ohne Änderung des Java-Codes weitere Formate zu unterstützen, indem weitere Views in der Spring Konfigurationsdatei ergänzt werden. Der Grund hierfür ist, dass unterstützte Formate einschließlich passender Views und Converter sowie zugehöriger Model-Klassen lediglich in dieser XML-Datei konfiguriert werden.

Frage 9: Wie erfolgt die Content Negotiation?
Die Wahl der richtigen Marshaller und Unmarshaller übernimmt Spring MVC anhand der Konfiguration automatisch. So bekommen die Methoden bereits Model-Objekte übergeben und können ebenso Model-Objekte wieder zurückgeben. Dies entspricht der Implementierung mit Jersey und RESTEasy, deren Methoden-Annotationen ebenfalls das oder die Formate aufnehmen. Die Wahl eines zum Format passenden Providers funktionierte zumindest für die im Fallbeispiel eingesetzten JAXB-Objekte einwandfrei. Die Rückgabe eines Model-Objekts ist ebenfalls bei Restlet 2.0 möglich, da versprochen wird, beliebige Objekte konvertieren zu können. Die Wahl des Converters kann beeinflusst werden, indem für verschiedene Formate separate Methoden implementiert werden, die unterschiedliche Typen zurückgeben. Die Wahl der passenden Methode geschieht durch Angabe des Formats in einer Annotation. Die Überführung von der Repräsentation in ein Model-Objekt bleibt der Ressourcenmethode überlassen. Soll die

Umsetzung abhängig vom Format geschehen, können hier ebenfalls mehrere Methoden implementiert werden, deren Auswahl das Framework automatisch übernimmt. Nicht ganz so komfortabel erfolgt die Umsetzung mit Restlet 1.1. Hier wird die Content Negotiation lediglich für GET-Requests unterstützt. Angebotene Formate werden hierbei im Konstruktor festgelegt, anstatt über Annotationen an der Methode. In der Methode muss eine Fallunterscheidung implementiert werden, um die jeweils gewünschte Repräsentation zu erzeugen. Das Framework übernimmt also nur die Entscheidung, welches das am besten passende Format bezüglich der angegebenen `Accept`-Header ist. Diese Entscheidung muss bei der Servlet-API und JAX-WS komplett selbst implementiert werden, die Frameworks bieten keine Unterstützung bei der Auswahl des Formats.

Frage 10: Werden Verweise der Ressourcen untereinander in Repräsentationen unterstützt?
Eine Unterstützung, in Repräsentationen Verweise auf andere Ressourcen abzubilden, bietet kein Framework. Dies ist ein Indikator dafür, dass es hierfür keine offensichtliche Lösung gibt. Andererseits ist es überraschend, da die Bedeutung der Verweishaftigkeit sowohl von Roy Fielding selbst [Fie00, S. 82] als auch von Stefan Tilkov [Til09, S. 65 bis 80] sowie Leonard Richardson und Sam Ruby [RR07, S. 106 bis 109] betont werden.

5.5. Zusammenfassung und Bewertung

Wie in den vorherigen Abschnitten gezeigt wurde, besitzen die Frameworks verschiedene Stärken und Schwächen. Diese wurden mit Hilfe des in Abschnitt 3.1 eingeführten Fragenkatalogs untersucht und herausgestellt. Die Unterschiede sind teilweise in der Geschichte und den Hintergründen der Frameworks begründet. Die Servlet-API wurde dafür entwickelt, dynamische Web-Inhalte leichter darstellen zu können. JAX-WS wurde für die Unterstützung XML-basierter Web Services entworfen. Beide Frameworks bieten so gut wie keine Unterstützung der REST-Prinzipien. Spring MVC merkt man an, dass die REST-Unterstützung nachträglich integriert wurde. Durch die starke Anlehnung an JAX-RS und die ohnehin vorhandene strikte Trennung von interner Darstellung und Repräsentation ist dies dennoch sehr gelungen.
Restlet 1.1 ist ein mächtiges Framework, was einerseits explizit die REST-Prinzipien unterstützt, andererseits durch die Komplexität gerade für einfache Web Services überladen wirkt. Dieses Manko wurde durch das umfangreiche Refactoring für Restlet 2.0 behoben. Eine abschließende Beurteilung ist aufgrund des unfertigen Stands nicht möglich.
Jersey und RESTEasy zeigen, dass JAX-RS eine gelungene API ist, welche die Konzepte von REST gut unterstützt. Jersey wirkt etwas leichtgewichtiger als RESTEasy.

Neben dem größeren Angebot an mitgelieferten Providern überzeugt RESTEasy vor allem durch ein eigenes Sicherheitskonzept und einen integrierten Cache.

Tabelle 5.2 zeigt noch einmal die Stärken und Schwächen der Frameworks im Überblick.

Vergleichsmerkmal	Servlet-API 2.5	JAX-WS RI 2.1.2	Jersey 1.0.3.1	RESTEasy 1.2.1	Restlet 1.1.7	Restlet 2.0 M6	Spring 3 MVC
1. Ressourcen-Implementierung	+	+	++	++	++	++	++
2. URI-Zuordnung	–	–	++	++	++	++	++
3. Schnittstelle/Methoden	+	+	++	++	++	+	+
4. Sicherheit/Authentifizierung	+	+	+	++	++	++	o
5. Zustandslosigkeit	–	–	++	++	++	++	–
6. Caching	o	o	+	++	o	+	o
7. bedingte Requests	+	o	+	+	++	++	+
8. Repräsentationen	o	o	++	++	+	++	++
9. Content Negotiation	o	o	++	++	+	++	+
10. Verweise auf andere Ressourcen	o	o	o	o	o	o	o

Tabelle 5.2.: Stärken und Schwächen im Vergleich

Die Untersuchung hat gezeigt, dass sämtliche Aspekte prinzipiell in allen Frameworks realisiert werden können, so dass eine Bewertung mit -- nicht notwendig ist. Die einheitliche Vergabe von o für den Aspekt der Verweise auf andere Ressourcen macht deutlich, dass die Frameworks diesbezüglich in der Zukunft noch besser werden können.

6. Fazit und Ausblick

Die Aufgabe dieser Arbeit war die Untersuchung einer Auswahl von Frameworks auf ihre Tauglichkeit zur Erstellung von Web Services nach den Prinzipien von REST. Dazu sollte zunächst ein Fragenkatalog auf Basis der Dissertation von Roy Fielding ermittelt werden, um mit diesem die Frameworks systematisch zu überprüfen. Im Ergebnis enthält dieser Katalog 10 Fragen, die in die drei Themen-Schwerpunkte *Ressourcen und Schnittstelle*, *Statushaltung und Caching* sowie *Repräsentationen und Verweise* gegliedert wurden. Weiterhin wurde ein Fallbeispiel entworfen und entwickelt, welches die Beantwortung der Fragen im praktischen Einsatz unterstützen sollte.

Um eine einheitliche Basis für den Vergleich zu schaffen, wurden hierfür Rahmenbedingungen festgelegt. So wurde die Untersuchung auf Java-Frameworks beschränkt. Es wurde nur die serverseitige Erstellung der Web Services untersucht und eventuelle Client-APIs außer Acht gelassen. Zur Vereinheitlichung der Beispiel-Realisierung wurden allgemeine Teile der Anwendung in das Common-Projekt ausgelagert. Dazu gehörten die Datencontainer, die Businesslogik einschließlich der Datenbank-Anbindung, einige Utility-Klassen und eine Handvoll gemeinsam genutzter Exceptions. Eine Herausforderung bei der Erstellung des Common-Projekts war das Bestreben einerseits, die Reaktionen der Frameworks zu vereinheitlichen. Andererseits sollten Besonderheiten der Frameworks dadurch nicht in den Hintergrund geraten.

Die vorliegende Arbeit hat gezeigt, dass die untersuchten Frameworks in einigen Aspekten ähnlich oder sogar identisch sind, während sie sich in anderen Punkten stark unterscheiden. Die Servlet-API und JAX-WS zeigen deutliche Schwächen bei der Umsetzung von RESTful Web Services, auch wenn sich das Fallbeispiel mit erhöhtem Programmieraufwand realisieren ließ. Die anderen Frameworks haben unterschiedliche Stärken und Schwächen. Eine Schwäche, die alle Frameworks gemeinsam besitzen, ist die fehlende Unterstützung von Verweisen der Ressourcen untereinander innerhalb ihrer Repräsentationen. Zumindest für XML und JSON konnten diese Verweise durch eine eigene Implementierung realisiert werden, jedoch wäre eine automatisierte Unterstützung durch die Frameworks wünschenswert. Ein weiterer Aspekt, der noch nicht in allen Frameworks optimal realisiert wird, ist die Unterstützung von bedingten Requests, indem das Framework die Request-Header mit Metadaten vergleicht, die die Ressource be-

reitstellt. Ebenfalls wünschenswert ist eine optionale Berechnung des ETags durch das Framework.

Die untersuchten Softwarestände der Frameworks haben sich während der Erstellung dieser Arbeit überholt. Für sämtliche Frameworks liegen inzwischen neuere Versionen als die untersuchten vor.
Im Dezember 2009 wurde die Servlet-API 3.0 [Mor09] veröffentlicht, die Anforderungen aus dem JSR 311 (JAX-RS) aufgreift. Eine Tendenz Richtung REST lassen weder die Spezifikation für JAX-WS 2.2 [KG07] noch deren Referenz-Implementierung 2.2, die ebenfalls im Dezember 2009 veröffentlicht wurden, erkennen. Auch für JAX-RS gibt es seit November 2009 eine neue Version 1.1 der Spezifikation [HS08]. Ziele von JAX-RS 1.1 sind die Erhöhung der Anwenderfreundlichkeit, aber auch die Erstellung von RESTful Web Services als Servlets oder JAX-WS-Endpoints. JAX-RS 1.1 wird in Jersey 1.1.5 und RESTEasy 2.0 (beta) implementiert, beide Stände sind von Januar 2010. Ebenfalls im Januar 2010 wurden neue Versionen von Restlet veröffentlicht, die stabile Version 1.1.8 sowie die instabile Version 2.0 Milestone 7. Diese neuen Stände enthalten im Wesentlichen Fehlerkorrekturen. Dies trifft genauso auf die neue Spring MVC Version 3.0.1 zu, die seit Februar 2010 zur Verfügung steht.

Diese aktuellen Veröffentlichungen zeigen, dass die Werkzeuge zur Erstellung von Web Services überhaupt, aber im Besonderen auch nach den Prinzipien von REST, stetig weiter entwickelt werden. Die aufgezeigten Schwächen bieten Potenzial für zukünftige Versionen der Frameworks, um die Prinzipien von REST noch besser zu unterstützen. Eine Tendenz, die unterschiedlichen Technologien enger zu verzahnen ist in den neuen Versionen der Spezifikationen erkennbar. Dies ist eine positive Entwicklung, anstatt die Diskussionen über den richtigen Weg zu erhärten. Insgesamt kann festgestellt werden, dass REST ein nicht neuer, aber dennoch aktueller und moderner Stil zur Umsetzung von Web Services ist, der in jüngster Zeit noch an Bedeutung gewonnen hat.

Literaturverzeichnis

[Bay02] Bayer, Thomas: *REST Web Services*. Version: 2002. http://www.oio.de/public/xml/rest-webservices.htm. – Online–Ressource, Abruf: 28.02.2010

[BL94] Berners-Lee, T.: *Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web*. Version: June 1994. http://www.ietf.org/rfc/rfc1630.txt. RFC 1630 (Informational) (Request for Comments 1630). – Online–Ressource, Abruf: 28.02.2010

[FGM+99] Fielding, R. ; Gettys, J. ; Mogul, J. ; Frystyk, H. ; Masinter, L. ; Leach, P. ; Berners-Lee, T.: *RFC 2616: Hypertext transfer protocol – HTTP/1.1*. Version: 1999. http://www.ietf.org/rfc/rfc2616.txt. – Online–Ressource, Abruf: 28.02.2010

[Fie00] Fielding, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, Diss., 2000. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. – Online–Ressource, Abruf: 28.02.2010

[Gro04] Group, W3C Technical A.: *Architecture of the World Wide Web, Volume One*. Version: Dezember 2004. http://www.w3.org/TR/2004/REC-webarch-20041215. – Online–Ressource, Abruf: 28.02.2010

[HS08] Hadley, Marc ; Sandoz, Paul: *JAX-RS: Java^{TM}API for RESTful Web Services*. Version: September 2008. http://jcp.org/en/jsr/detail?id=311. – Online–Ressource, Abruf: 28.02.2010

[JP09] JCP-PMO: *The Java Community Process(SM) Program - Introduction - FAQ*. Version: 2009. http://www.jcp.org/en/introduction/faq#general. – Online–Ressource, Abruf: 28.02.2010

[KG07] Kohlert, Doug ; Gupta, Arun: *The Java API for XML-Based Web Services (JAX-WS) 2.1*. Version: Mai 2007. http://jcp.org/en/jsr/detail?id=224. – Online–Ressource, Abruf: 28.02.2010

[Mor07] Mordani, Rajiv: *Java^{TM}Servlet Specification Version 2.5 MR6*. Version: First Edition, August 2007. `http://jcp.org/en/jsr/detail?id=154`. – Online–Ressource, Abruf: 28.02.2010

[Mor09] Mordani, Rajiv: *Java^{TM}Servlet 3.0 Specification*. Version: Dezember 2009. `http://jcp.org/en/jsr/detail?id=315`. – Online–Ressource, Abruf: 28.02.2010

[Pau07] Pautasso, Cesare: *SOAP vs. REST*. Version: 2007. `http://www.iks.inf.ethz.ch/education/ss07/ws_soa/slides/SOAPvsREST_ETH.pdf`. – Online–Ressource, Abruf: 28.02.2010

[PZL08] Pautasso, Cesare ; Zimmermann, Olaf ; Leymann, Frank: *RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision*. Version: April 2008. `http://www2008.org/`. – 805–814 S. – Online–Ressource, Abruf: 28.02.2010

[RES09] *RESTEasy JAX-RS RESTFul Web Services for Java 1.2.GA*. Version: November 2009. `http://www.jboss.org/resteasy/docs.html`. – Online–Ressource, Abruf: 28.02.2010

[RR07] Richardson, Leonard ; Ruby, Sam: *Web Services mit REST*. O'REILLY, 2007

[San09] Sandoval, Jose: *RESTful Java Web Services*. PACKT, 2009

[Sle08] Sletten, Brian: *REST for Java developers*. Version: Oktober 2008. `http://www.javaworld.com/javaworld/jw-10-2008/jw-10-rest-series-1.html`. – Online–Ressource, Abruf: 28.02.2010

[Til07] Tilkov, Stefan: *A Brief Introduction to REST*. Version: Dezember 2007. `http://www.infoq.com/articles/rest-introduction`. – Online–Ressource, Abruf: 28.02.2010

[Til09] Tilkov, Stefan: *REST und HTTP*. dpunkt.verlag, 2009

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

A. Anhang - XML-Repräsentation des Fallbeispiels

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<customerService xml:base="http://localhost:8080/RESTEasy">
<customers href="/customers">
    <customer href="/customers/1">
        <name>Mustermann</name>
        <forename>Martin</forename>
        <dayOfBirth>01.01.1970</dayOfBirth>
        <billingAddress href="/customers/1/addresses/billing">
            <city>Musterstadt</city>
            <street>Musterstr. 11</street>
            <zipCode>12345</zipCode>
        </billingAddress>
        <shippingAddress href="/customers/1/addresses/shipping">
            <city>Teststadt</city>
            <state>Testland</state>
            <street>Teststr. 9</street>
            <zipCode>54321</zipCode>
        </shippingAddress>
        <phoneNumbers href="/customers/1/phones">
            <phoneNumber href="/customers/1/phones/2">
                <name>Dienstlich</name>
                <number>09999/9999999</number>
            </phoneNumber>
            <phoneNumber href="/customers/1/phones/1">
                <name>Privat</name>
                <number>0123/12345678</number>
            </phoneNumber>
            <phoneNumber href="/customers/1/phones/3">
                <name>Handy</name>
                <number>11111/22222</number>
            </phoneNumber>
        </phoneNumbers>
    </customer>
    <customer href="/customers/2">
        <name>Musterfrau</name>
        <forename>Maria</forename>
        <dayOfBirth>01.01.1980</dayOfBirth>
        <billingAddress href="/customers/2/addresses/billing">
            <city>Musterdorf</city>
            <street>Dorfstr. 11</street>
            <zipCode>11111</zipCode>
        </billingAddress>
        <shippingAddress href="/customers/2/addresses/shipping">
            <city>Testdorf</city>
            <state>Testland</state>
            <street>Hauptstr. 6</street>
            <zipCode>54321</zipCode>
        </shippingAddress>
        <phoneNumbers href="/customers/2/phones">
            <phoneNumber href="/customers/2/phones/5">
                <name>Handy</name>
                <number>6666/77777777</number>
```

```
53                </phoneNumber>
54                <phoneNumber href="/customers/2/phones/4">
55                    <name>Privat</name>
56                    <number>5555/6666</number>
57                </phoneNumber>
58            </phoneNumbers>
59        </customer>
60    </customerService>
```

Listing A.1: XML-Repräsentation der Ressource Kundenliste

B. Anhang - JSON-Repräsentation des Fallbeispiels

```
{
    "xml:base":"http://localhost:8080/RESTEasy",
    "customers":{
        "href":"/customers",
        "customer":[{
            "href":"/customers/2",
            "dayOfBirth":"01.01.1980",
            "forename":"Maria",
            "billingAddress":{
                "href":"/customers/2/addresses/billing",
                "street":"Dorfstr. 11",
                "city":"Musterdorf",
                "zipCode":"11111"
            },
            "shippingAddress":{
                "href":"/customers/2/addresses/shipping",
                "street":"Hauptstr. 6",
                "city":"Testdorf",
                "zipCode":"54321",
                "state":"Testland"
            },
            "phoneNumbers":{
                "href":"/customers/2/phones",
                "phoneNumber":[{
                    "href":"/customers/2/phones/5",
                    "name":"Handy",
                    "number":"6666/77777777"
                },
                {
                    "href":"/customers/2/phones/4",
                    "name":"Privat",
                    "number":"5555/6666"
                }]
            },
            "name":"Musterfrau"
        },
        {
            "href":"/customers/1",
            "dayOfBirth":"01.01.1970",
            "forename":"Martin",
            "billingAddress":{
                "href":"/customers/1/addresses/billing",
                "street":"Musterstr. 11",
                "city":"Musterstadt",
                "zipCode":"12345"
            },
            "shippingAddress":{
                "href":"/customers/1/addresses/shipping",
                "street":"Teststr. 9",
                "city":"Teststadt",
                "zipCode":"54321",
                "state":"Testland"
```

```
53                  },
54                  "phoneNumbers":{
55                      "href":"/customers/1/phones",
56                      "phoneNumber":[{
57                          "href":"/customers/1/phones/3",
58                          "name":"Handy",
59                          "number":"11111/22222"
60                      },
61                      {
62                          "href":"/customers/1/phones/2",
63                          "name":"Dienstlich",
64                          "number":"09999/9999999"
65                      },
66                      {
67                          "href":"/customers/1/phones/1",
68                          "name":"Privat",
69                          "number":"0123/12345678"
70                      }]
71                  },
72                  "name":"Mustermann"
73              }]
74      }
75  }
```

Listing B.1: JSON-Repräsentation der Ressource Kundenliste

C. Anhang - WADL Beschreibung des Fallbeispiels

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application xmlns="http://research.sun.com/wadl/2006/10">
    <doc xmlns:jersey="http://jersey.dev.java.net/"
        jersey:generatedBy="Jersey: ${project.version} ${buildNumber}"/>
    <resources base="http://localhost:8080/jsr311/">
        <resource path="/customers">
            <method name="GET" id="get">
                <request>
                    <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
                      type="xs:string" style="query" name="name"/>
                    <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
                      type="xs:string" style="query" name="forename"/>
                </request>
                <response>
                    <representation mediaType="application/xml"/>
                    <representation mediaType="application/json"/>
                </response>
            </method>
            <method name="DELETE" id="delete">
                <request>
                    <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
                      type="xs:string" style="header" name="Authorization"/>
                    <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
                      type="xs:string" style="query" name="name"/>
                    <param xmlns:xs="http://www.w3.org/2001/XMLSchema"
                      type="xs:string" style="query" name="forename"/>
                </request>
                <response>
                    <representation mediaType="*/*"/>
                </response>
            </method>
            <method name="POST" id="post">
                <request>
                    <representation mediaType="application/xml"/>
                </request>
                <response>
                    <representation mediaType="*/*"/>
                </response>
            </method>
        </resource>
        <resource path="/customers/{custid}/phones">
            <method name="GET" id="get">
                <response>
                    <representation mediaType="application/xml"/>
                    <representation mediaType="application/json"/>
                </response>
            </method>
            <method name="POST" id="post">
                <request>
                    <representation mediaType="application/xml"/>
                </request>
                <response>
```

```
53                  <representation mediaType="*/*"/>
54              </response>
55          </method>
56      </resource>
57      <resource path="/customers/{custid}/addresses/{id}">
58          <method name="GET" id="get">
59              <response>
60                  <representation mediaType="application/xml"/>
61                  <representation mediaType="application/json"/>
62              </response>
63          </method>
64          <method name="PUT" id="put">
65              <request>
66                  <representation mediaType="application/xml"/>
67              </request>
68              <response>
69                  <representation mediaType="*/*"/>
70              </response>
71          </method>
72          <method name="DELETE" id="delete">
73              <response>
74                  <representation mediaType="*/*"/>
75              </response>
76          </method>
77      </resource>
78      <resource path="/customers/{custid}/phones/{id}">
79          <method name="GET" id="get">
80              <response>
81                  <representation mediaType="application/xml"/>
82                  <representation mediaType="application/json"/>
83              </response>
84          </method>
85          <method name="PUT" id="put">
86              <request>
87                  <representation mediaType="application/xml"/>
88              </request>
89              <response>
90                  <representation mediaType="*/*"/>
91              </response>
92          </method>
93          <method name="DELETE" id="delete">
94              <response>
95                  <representation mediaType="*/*"/>
96              </response>
97          </method>
98      </resource>
99      <resource path="/customers/{id}">
100         <method name="GET" id="get">
101             <response>
102                 <representation mediaType="application/xml"/>
103                 <representation mediaType="application/json"/>
104             </response>
105         </method>
106         <method name="PUT" id="put">
107             <request>
108                 <representation mediaType="application/xml"/>
109             </request>
110             <response>
111                 <representation mediaType="*/*"/>
112             </response>
113         </method>
114         <method name="DELETE" id="delete">
115             <response>
116                 <representation mediaType="*/*"/>
117             </response>
118         </method>
119     </resource>
```

```
120     </resources>
121 </application>
```

Listing C.1: WADL Beschreibung der Anwendung

Die vorliegende WADL-Beschreibung wurde von Jersey durch einen GET-Request auf die Ressource /application.wadl generiert.

D. Anhang - Testfälle und -ergebnisse

Das Verhalten der Frameworks wurde unter anderem mit Hilfe des in Abschnitt 3.3 beschriebenen Testclients untersucht. Dabei wurde eine Serie von Tests entwickelt, die für sämtliche Frameworks durchgeführt wurde, d.h. eine identische Folge von Requests wurde an die unterschiedlichen Web Services gesendet und die erwartete Response mit der erhaltenen Response verglichen.

Die Tests gliedern sich dabei wie folgt:

- fehlerhafte Requests (Negativtest)
  - leerer Entity-Body
  - nicht erlaubte Methoden
  - nicht vorhandene bzw. falsche ID
- Authentifizierung
- Content Negotiation mit exotischem MIME-Type
- OPTIONS Requests
- HEAD Requests
- triviale Requests (Positivtest)
  - Ressourcen aller Typen anlegen
  - Ressourcen aller Typen ändern
  - Ressourcen aller Typen abrufen
  - Kundenliste im JSON-Format abrufen
  - Kundenliste mit Angabe von Query-Parametern
  - Ressourcen aller Typen löschen
- bedingte Requests
  - bedingtes GET
  - bedingtes PUT

Im Folgenden werden die Testfälle kurz beschrieben, das erwartete Ergebnis aufgeführt und die Reaktionen der einzelnen Frameworks erklärt. Ähnliche Testfälle werden hierbei gruppiert und zusammengefasst.

fehlerhafte Requests (Negativtest)

Testfall 1: POST auf eine Listressource mit leerem Body
erwartetes Ergebnis: Statuscode *400 Bad Request* - Es werden nur gültige XML-Repräsentationen akzeptiert, eine leere Repräsentation ist ein ungültiger Zugriff.
tatsächliches Ergebnis: Alle Frameworks liefern den Statuscode *400 Bad Request*.

Testfall 2: PUT auf eine Einzelressource mit leerem Body
erwartetes Ergebnis: Statuscode *400 Bad Request* - Es werden nur gültige XML-Repräsentationen akzeptiert, eine leere Repräsentation ist ein ungültiger Zugriff.
tatsächliches Ergebnis: Alle Frameworks bis auf eines liefern den Statuscode *400 Bad Request*. Restlet 2.0 liefert stattdessen den Statuscode *500 Internal Server Error*.

Testfälle 3-8: Requests für ungültige Methoden
erwartetes Ergebnis: Statuscode *405 Method Not Allowed*, Header `Allow` enthält die erlaubten Methoden
tatsächliches Ergebnis: Das erwartete Ergebnis liefern Jersey und Restlet 1.1. RESTEasy liefert dies ebenfalls, jedoch nur für die Ressourcen Kunde und Kundenliste, für die anderen Ressourcen (mit einer ID mitten in der URI) wird *404 Not Found* geliefert. Die anderen Frameworks liefern zwar den korrekten Statuscode, der `Allow`-Header fehlt jedoch bei der Servlet-API und der JAX-WS RI. Restlet 2.0 liefert diesen Header, er enthält jedoch keine Methoden, während Spring MVC lediglich die Methode OPTIONS nicht aufführt.

Testfälle 9-10: POST auf Einzelressource mit ungültiger ID
erwartetes Ergebnis: Statuscode *404 Not Found*
tatsächliches Ergebnis: Da der Ressourcentyp existiert, d.h. es wird ein passendes URI-Template gefunden, für diese aber keine POST-Methode implementiert ist, antworten fast alle Frameworks nicht mit *404 Not Found* sondern mit *405 Method Not Allowed*. In Restlet 2.0 wird die Existenz der konkreten Ressource mit der angegebenen ID bereits in der Methode `doInit` geprüft, weshalb hier doch mit *404 Not Found* reagiert wird. Dieselbe Reaktion kann bei Jersey, RESTEasy, Restlet 1.1 und Spring MVC erreicht werden, indem die Existenz der ID im Konstruktor geprüft wird. Dies ist zur Unterstützung bedingter Requests ohnehin empfehlenswert, da Metadaten zur Ressource anderenfalls in jeder Methode ermittelt werden müssten. Bei JAX-WS kann die Prüfung in der Methode `invoke` vor der Fallunterscheidung für das HTTP-Verb geschehen. In der Servlet-API steht dem Konstruktor nicht der Request zur Verfügung,

weshalb hier keine Prüfung erfolgen kann.
Wegen des Fehlers bei RESTEasy, auf nicht implementierte Methoden für bestimmte URIs immer mit *404 Not Found* zu antworten, liefert das Framework diesen Statuscode für die Ressourcen Adresse und Telefonnummer.

Testfall 11: POST auf Adresse mit ungültiger Adress-ID
erwartetes Ergebnis: Statuscode *403 Forbidden*
tatsächliches Ergebnis: Für diesen Testfall ist das Ergebnis sehr ähnlich wie in Testfällen 9-10. Restlet 2.0 liefert den Statuscode *403 Forbidden*, RESTEasy liefert auch hier *404 Not Found* und die anderen Frameworks liefern *405 Method Not Allowed*.

Testfälle 12-23: Requests für nicht existierende Ressourcen
erwartetes Ergebnis: Statuscode *404 Not Found*
tatsächliches Ergebnis: Alle Frameworks liefern das erwartete Ergebnis.

Testfälle 24-26: Requests auf Adresse mit ungültiger Adress-ID
erwartetes Ergebnis: Statuscode *403 Forbidden*, Body enthält Freitext mit erlaubten IDs
tatsächliches Ergebnis: Alle Frameworks liefern das erwartete Ergebnis.

Testfälle 27-32: Requests Einzelressource mit nicht-numerischer ID
erwartetes Ergebnis: Statuscode *400 Bad Request* oder *404 Not Found*
tatsächliches Ergebnis: Jersey und RESTEasy liefern *404 Not Found*, die anderen Frameworks liefern *400 Bad Request*.

Authentifizierung

Testfälle 33-34: DELETE auf Kundenliste ohne bzw. mit falschen Authentifizierungsdaten
erwartetes Ergebnis: Statuscode *401 Unauthorized*,
Header `WWW-Authenticate: Basic realm="Customer List"`
tatsächliches Ergebnis: Alle Frameworks liefern das erwartete Ergebnis.

Testfall 35: DELETE auf Kundenliste mit korrekten Authentifizierungsdaten
erwartetes Ergebnis: Statuscode *200 OK*
tatsächliches Ergebnis: Alle Frameworks liefern das erwartete Ergebnis. RESTEasy hält die Ressourcen allerdings weiterhin im Cache, so dass ein anschließender GET auf einen gelöschten Kunden diesen liefert anstatt *404 Not Found*.

Content Negotiation mit exotischem MIME-Type

Testfall 36: GET mit exotischem Format im `Accept`-Header
erwartetes Ergebnis: Statuscode *406 Not Acceptable*, Body enthält Freitext mit erlaubten Formaten
tatsächliches Ergebnis: Restlet 2.0 liefert die XML-Repräsentation mit dem Statuscode *200 OK*, Spring MVC liefert den Statuscode *500 Internal Server Error*. Alle anderen Frameworks liefern *406 Not Acceptable*. Bei der Servlet-API und der JAX-WS RI enthält der Body die erlaubten Formate, während Jersey, RESTEasy und Restlet 1.1 eine standardisierte Fehlermeldung des Apache Tomcat liefern.

Testfall 37: GET mit exotischem Format im `Content-Type`-Header
erwartetes Ergebnis: Statuscode *200 OK*, gefordertes Format im `Content-Type`-Header
tatsächliches Ergebnis: Alle Frameworks liefern das erwartete Ergebnis.

Testfall 38: PUT mit exotischem Format im `Accept`-Header
erwartetes Ergebnis: Statuscode *200 OK*
tatsächliches Ergebnis: Alle Frameworks liefern das erwartete Ergebnis.

Testfall 39: PUT mit exotischem Format im `Content-Type`-Header
erwartetes Ergebnis: Statuscode *415 Unsupported Media Type*, Body enthält Freitext mit erlaubten Formaten
tatsächliches Ergebnis: Restlet 2.0 akzeptiert die Repräsentation und liefert den Statuscode *200 OK*, RESTEasy liefert für einige Ressourcen den Statuscode *404 Not Found*. Alle anderen Frameworks liefern *415 Unsupported Media Type*. Bei der Servlet-API, der JAX-WS RI und Restlet 1.1 enthält der Body die akzeptierten Formate, während Jersey und Spring MVC eine standardisierte Fehlermeldung des Apache Tomcat liefern.

Testfall 40: POST mit exotischem Format im `Accept`-Header
erwartetes Ergebnis: Statuscode *201 Created*
tatsächliches Ergebnis: Alle Frameworks liefern das erwartete Ergebnis.

Testfall 41: POST mit exotischem Format im `Content-Type`-Header
erwartetes Ergebnis: Statuscode *415 Unsupported Media Type*, Body enthält Freitext mit erlaubten Formaten
tatsächliches Ergebnis: Restlet 2.0 akzeptiert die Repräsentation und liefert den Statuscode *200 OK*, RESTEasy liefert für einige Ressourcen den Statuscode *404 Not Found*. Alle anderen Frameworks liefern *415 Unsupported Media Type*. Bei der Servlet-API, der JAX-WS RI und Restlet 1.1 enthält der Body die akzeptierten Formate,

während Jersey und Spring MVC eine standardisierte Fehlermeldung des Apache Tomcat liefern.

Testfall 42: DELETE mit exotischem Format im `Accept`-Header
erwartetes Ergebnis: Statuscode *200 OK*
tatsächliches Ergebnis: Spring MVC liefert den Statuscode *500 Internal Server Error*. Alle anderen Frameworks liefern das erwartete Ergebnis.

Testfall 43: DELETE mit exotischem Format im `Content-Type`-Header
erwartetes Ergebnis: Statuscode *200 OK*
tatsächliches Ergebnis: Alle Frameworks liefern das erwartete Ergebnis.

Testfall 44: HEAD mit exotischem Format im `Accept`-Header
erwartetes Ergebnis: Statuscode *406 Not Acceptable*, kein Body
tatsächliches Ergebnis: Restlet 2.0 liefert den Statuscode *200 OK*, Spring MVC liefert den Statuscode *500 Internal Server Error*. Alle anderen Frameworks liefern *406 Not Acceptable* ohne Entity-Body.

Testfall 45: HEAD mit exotischem Format im `Content-Type`-Header
erwartetes Ergebnis: Statuscode *200 OK*, kein Body
tatsächliches Ergebnis: Alle Frameworks liefern das erwartete Ergebnis.

Testfall 46: OPTIONS mit exotischem Format im `Accept`-Header
erwartetes Ergebnis: Statuscode *200 OK*
tatsächliches Ergebnis: Bis auf RESTEasy, welches auf OPTIONS für einige Ressourcen mit *404 Not Found* antwortet, liefern alle Frameworks das erwartete Ergebnis.

Testfall 47: OPTIONS mit exotischem Format im `Content-Type`-Header
erwartetes Ergebnis: Statuscode *200 OK*
tatsächliches Ergebnis: Bis auf RESTEasy, welches auf OPTIONS für einige Ressourcen mit *404 Not Found* antwortet, liefern alle Frameworks das erwartete Ergebnis.

OPTIONS

Testfälle 48-52: OPTIONS für alle Ressourcen
erwartetes Ergebnis: Statuscode *200 OK*, Header `Allow` enthält die erlaubten Methoden
tatsächliches Ergebnis: Bei der JAX-WS RI und Spring MVC enthält der Header stets alle HTTP-Methoden. Alle anderen Frameworks liefern das erwartete Ergebnis,

bis auf RESTEasy, welches auf OPTIONS für einige Ressourcen mit *404 Not Found* antwortet.

HEAD

Testfälle 53-57: HEAD für alle Ressourcen
erwartetes Ergebnis: Statuscode *200 OK*, kein Body
tatsächliches Ergebnis: Alle Frameworks liefern das erwartete Ergebnis.

triviale Requests (Positivtest)

Testfall 58: umfassender Positivtest, einschließlich Neuanlage, Ändern, Lesen und Löschen für alle Ressource-Typen, XML- und JSON-Repräsentation für Kundenliste
erwartetes Ergebnis: korrekte Verarbeitung der Requests
tatsächliches Ergebnis: Alle Frameworks liefern das erwartete Ergebnis. RESTEasy hält die Ressourcen allerdings weiterhin im Cache, so dass ein anschließender GET auf einen gelöschten Kunden diesen liefert anstatt *404 Not Found*.

bedingte Requests

Testfälle 59-61: bedingtes GET (Datum, ETag, beides) auf Kunde
erwartetes Ergebnis: vor PUT: Statuscode *304 Not Modified*, nach PUT: Statuscode *200 OK*
tatsächliches Ergebnis: Jersey, Restlet 1.1 und Restlet 2.0 liefern für alle Varianten das erwartete Ergebnis. Die Servlet-API unterstützt nur den Vergleich mit dem Änderungsdatum und nicht mit dem ETag. Spring MVC unterstützt umgekehrt den Vergleich mit dem ETag und nicht mit dem Datum. RESTEasy unterstützt ebenfalls nur den Vergleich mit dem ETag, allerdings wirkt sich ein PUT nicht auf die gespeichert Kopie im Cache aus, sodass die Antwort auf den GET nach dem PUT unverändert *304 Not Modified* lautet. JAX-WS bietet keine Unterstützung, dadurch enthält die Response weder Datum noch ETag, wodurch dem Client die Informationen für einen bedingten Request fehlen.

Testfälle 62-64: bedingtes PUT (Datum, ETag, beides) auf Kunde
erwartetes Ergebnis: 1. PUT: Statuscode *200 OK*, 2. PUT: Statuscode *412 Precondition Failed*
tatsächliches Ergebnis: Die Servlet-API liefert zwar das Änderungsdatum in der Response, wertet diesen aber bei einem PUT-Request nicht aus, so dass auch der 2. PUT-Request durchgeführt wird und *200 OK* liefert. JAX-WS bietet keine Unterstützung,

dadurch enthält die Response weder Datum noch ETag, wodurch dem Client die Informationen für einen bedingten Request fehlen. Jersey funktioniert prinzipiell, durch einen Fehler im Vergleich[1] werden Änderungen an der Ressource allerdings erst als solche erkannt, wenn sie wenigstens 2 Sekunden nach der letzten vorherigen Änderung erfolgen. Im Testszenario erfolgt der 1. PUT nur 1 Sekunde nach der Anlage der Ressource, wodurch auch der 2. PUT-Request durchgeführt wird und *200 OK* liefert. Auch bei RESTEasy und Spring MVC wird der 2. PUT-Request durchgeführt und liefert *200 OK*, da der Vergleich mit dem ETag lediglich für GET-Requests erfolgt. Durch einen Fehler beim Vergleich liefert Restlet 1.1 für bedingte PUT-Requests den Statuscode *412 Precondition Failed*, dieser wurde inzwischen jedoch behoben, nachdem der Fehler gemeldet wurde. Der Testfall mit dem ETag ist erfolgreich. Bei Restlet 2.0 ist die Reaktion auf einen bedingten PUT-Request stets *404 Not Found*.

[1] Für den Fehler wurde im Bug Tracking System von Jersey ein Ticket mit der URI `https://jersey.dev.java.net/issues/show_bug.cgi?id=472` eingestellt.

E. Anhang - Inhalt der DVD

Die DVD enthält neben diesem Dokument (Diplomarbeit.pdf) eine vollständig konfigurierte Version von Eclipse für Windows, eine Tomcat 6.0 Installation, den Workspace der Beispielanwendung und die verwendeten Spezifikationen und JAR-Dateien der 7 Frameworks. Die Datei readme.pdf im Ordner Beispielanwendung beschreibt das Vorgehen, die entwickelten WebServices zu starten und zu testen.

Pfad	**Beschreibung**
Beispielanwendung	
eclipse	Eclipse Galileo Installation
javadoc	Dokumentation der Beispielanwendung
jre6	Installationsdatei für JRE 6
Tomcat 6.0	Installation des Tomcats 6.0
workspace	Workspace der Beispielanwendung
Common	Implementierung der allgemeinen Logik
daten	Gespeicherte Daten pro Framework
HttpClient	Testclient zum Test der Beispielanwendung
Jaxwsri	Implementierung mit JAX-WS RI 2.1.2
Jersey	Implementierung mit Jersey 1.0.3.1
RESTEasy	Implementierung mit RESTEasy 1.2.1
Restlet11	Implementierung mit Restlet 1.1.7
Restlet20	Implementierung mit Restlet 2.0 M6
Servers	Serverkonfiguration für Tomcat 6.0
ServletAPI	Implementierung mit Servlet-API 2.5
SpringMVC	Implementierung mit Spring 3 MVC
Common	JARs des Common Projekts
Frameworks	
JAX-WS RI	JARs und Spezifikation JAX-WS RI 2.1.2
Jersey	JARs Jersey 1.0.3.1 und Spezifikation JAX-RS 1.0
RESTEasy	JARs RESTEasy 1.2.1
Restlet	JARs Restlet 1.1.7 und 2.0 M6
ServletAPI	JARs und Spezifikation Servlet-API 2.5
SpringMVC	JARs Spring 3 MVC
HttpClient	JARs des Testclients

Tabelle E.1.: Inhalt DVD

F. Anhang - Download-Links

- Servlet-API 2.5
  `http://java.sun.com/javaee/downloads/previous/index.jsp`
- JAX-WS RI 2.1.2
  `https://jax-ws.dev.java.net/`
- Jersey 1.0.3.1
  `https://jersey.dev.java.net/`
- RESTEasy 1.2.1
  `http://sourceforge.net/projects/resteasy/files/ResteasyJAX-RS/`
- Restlet 1.1.7, Restlet 2.0
  `http://www.restlet.org/downloads/`
- Spring 3 MVC
  `http://www.springsource.org/download`
- db4o 7.12
  `http://www.db4o.com/DownloadNow.aspx`
- JAXB 2.1.12
  `https://jaxb.dev.java.net/`
- Json-lib 2.3 (abhängig von jakarta commons-lang 2.4, jakarta commons-beanutils 1.7.0, jakarta commons-collections 3.2, jakarta commons-logging 1.1.1, ezmorph 1.0.6)
  `http://sourceforge.net/projects/json-lib/files/`
  `http://commons.apache.org/`
  `http://sourceforge.net/projects/ezmorph/`
- JUnit 4.8
  `http://sourceforge.net/projects/junit/files/junit/`
- Commons-httpclient-3.1
  `http://hc.apache.org/downloads.cgi`