Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Stephan Koops

Implementierung des Java Specification Request 311
auf der Basis des Restlet-Frameworks

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Stephan Koops

Implementierung des Java Specification Request 311 auf der Basis des Restlet-Frameworks

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang Informatik
am Studiendepartment Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Friedrich Esser
Zweitgutachter : Prof. Dr. rer. nat. Bettina Buth

Abgegeben am 10. Juli 2008

Stephan Koops

Thema der Masterarbeit

Implementierung des Java Specification Request 311 auf der Basis des Restlet-Frameworks

Stichworte

Representational State Transfer, REST, Ressourcen-orientierte Architektur, ROA, RESTful Web-Service, JSR 311, JAX-RS: Java API for RESTful Web Services, Restlet, HTTP, dienst-orientierte Architektur, SOA

Kurzzusammenfassung

In den letzten Jahren geht der Trend in der Software-Entwicklung immer mehr zu dienst-orientierten Architekturen. Ein leichtgewichtiger Ansatz dafür sind RESTful Web-Services. Sun spezifiziert z. Zt. unter dem Namen JAX-RS (JSR 311) eine API für solche Web-Dienste. Diese Arbeit stellt diese Spezifikation vor, bewertet sie und zeigt Verbesserungsvorschläge auf. Außerdem wird geprüft, ob und wie die JAX-RS-Spezifikation auf der Basis des ebenfalls für RESTful Web-Services entwickelten Frameworks Restlet implementiert werden kann. Dazu werden entsprechende Anforderungen gesammelt und darauf aufbauend ein Entwurf vorgeschlagen und dieser prototypisch umgesetzt.

Stephan Koops

Title of the paper

Implementation of Java Specification Request 311 based on the Restlet framework

Keywords

Representational State Transfer, REST, resource oriented architecture, ROA, RESTful Web Service, JSR 311, JAX-RS: Java API for RESTful Web Services, Restlet, HTTP, service oriented architecture, SOA

Abstract

During the last years a trend in software engeneering is to model business software as service oriented architectures. A lightweight approach to implement a SOA are RESTful Web-Services. Sun ist just specifying a new API for such web services with the name JAX-RS (JSR 311). This paper presents this specification, reviews it and shows some ideas for improvements. It also checks, if and how the JAX-RS specification could be implemented based on the Restlet framework. This paper also collects some requirements for it. It also presents a design and shows the results of a prototypical implementation.

Inhaltsverzeichnis

Danksagung

An dieser Stelle bedanke ich mich bei meinem Betreuer Herrn Prof. Dr. Friedrich Esser für seine lange Geduld bei der Themenfindung und für die Betreuung meiner Masterarbeit, ebenso bei der Zweitbetreuerin Frau Prof. Dr. Bettina Buth.

Weiterhin möchte ich mich bei Jérôme Louvel und Thierry Boileau – dem Team des Restlet-Frameworks – bedanken, dass sie mich bei der Arbeit reichlich und unkompliziert unterstützt haben. Auch für die Erlaubnis der Nutzung einiger Grafiken von der Restlet-Webseite durch die Firma Noelios Technologies gilt der Dank.

Besonders gilt mein Dank auch meinen Eltern Rudolf und Helga Koops, die mir mein Studium ermöglicht und wesentlich finanziert haben. Meiner Mutter gilt besonderer Dank für das Korrektur lesen dieser Arbeit, ebenso meinem Vater und meinem Bruder Simon.

1 Einführung

1.1 Dienstorientierung im Internet

In den letzten Jahren wurden dienst-orientierte oder service-orientierte Architekturen (SOAs) immer populärer. In einer SOA stellen Anwendungen ihre Funktionalität als Dienste zur Verfügung. Diese können dann auch von anderen Anwendungen genutzt werden. Dies vermindert unerwünschte Kopplung zwischen verschiedenen Software-Systemen. SOAs, die (auch) von Anwendungen aus dem Internet genutzt werden sollen – z. B. von anderen Firmen – werden häufig mit Web-Diensten umgesetzt. Meistens werden dafür SOAP-Web-Services verwendet. Es gibt jedoch auch Alternativen zum SOAP. Eine dienst-orientierte Architektur kann beispielsweise auch mit Web-Diensten im Architekturstil *Representational State Transfer* (REST) umgesetzt werden.

1.2 Ziel dieser Arbeit

Für Web-Dienste im REST-Architekturstil gibt es inzwischen einige Frameworks für Java [1]. Die Firma Sun spezifiziert zur Zeit (seit April 2007, bis voraussichtlich September 2008) eine API für solche Web-Dienste. Dies geschieht im Rahmen des Java Community Process [2] als Java Specification Request 311. Die Spezifikation bekommt den Namen „JAX-RS: Java API for RESTful Web Services“ (abgekürzt mit „JAX-RS“) und soll Teil der Java Enterprise Edition (Java EE) in Version 6 werden.

Das Ziel dieser Arbeit ist es, diese Spezifikation zu bewerten, Verbesserungsvorschläge zu entwickeln und diese in den Spezifikationsprozess einzubringen. Außerdem soll ein Entwurf für die Implementierung dieser Spezifikation auf der Basis des ebenfalls für RESTful Web-Services entwickelten Framework *Restlet* gemacht werden, welcher dann prototypisch umgesetzt werden soll.

[1] Java ist ein eingetragenes Markenzeichen der Firma Sun Microsystems, Inc.
[2] Erklärungen finden sich im Glossar ab Seite 123.

1.3 Inhalt und Aufbau der Arbeit

Im Kapitel 2 werden die Grundlagen für diese Arbeit behandelt:

- der Architekturstil Representational State Transfer (Abschnitt 2.1 ab Seite 6),
- das Hypertext Transfer Protocol (HTTP, Abschnitt 2.2 ab Seite 21),
- auf diesen beiden aufbauend RESTful Web-Services (Abschnitt 2.3 ab Seite 28) und
- Implementierungs-Frameworks für RESTful Web-Services (Abschnitt 2.4 ab Seite 31).
- Als letzter Teil dieses Kapitels werden RESTful Web-Services mit Alternativen verglichen (Abschnitt 2.5 ab Seite 39).

Im 3. Kapitel (ab Seite 43) wird die zu implementierende Spezifikation JAX-RS für RESTful Web-Services vorgestellt und bewertet.

Darauf folgend werden im 4. Kapitel (ab Seite 66) Anforderungen für eine Implementierung der JAX-RS-Spezifikation auf Basis des Restlet-Frameworks gesammelt und anhand dieser Anforderungen im Kapitel 5 (ab Seite 79) eine JAX-RS-Laufzeitumgebung auf der Basis des Restlet-Frameworks entworfen. Das Kapitel 6 (ab Seite 105) beschreibt die prototypische Implementation dieses Entwurfes.

Im abschließenden Kapitel 7 (ab Seite 109) wird ein Fazit gezogen und ein Ausblick auf weitere Möglichkeiten im Umfeld dieser Spezifikation gegeben.

Im Anhang werden einige Code-Beispiele für in dieser Arbeit genannte APIs gezeigt, um die praktische Umsetzung kurz zu zeigen (Anhang A ab Seite 111).

1.4 Related Work

Das wichtigste Dokument zum Architekturstil Representational State Transfer ist die Dissertation „Software Architectural Styles for Network-based Applications“ von Roy Thomas Fielding [Fielding 2000]. Im Kapitel 5 definiert er dort diesen Architekturstil. Die Dissertation ist im Internet verfügbar, jedoch nicht als Buch.

Das Buch „RESTful Web Services“ von Leonard Richardson und Sam Ruby aus dem Jahr 2007[3] beschreibt die Konzepte des REST-Architekturstils und zeigt die Entwicklung von Web-Diensten im REST-Architekturstil anhand von Beispielen, meistens in der Programmiersprache Ruby. Das Buch gibt auch eine Einführung in das in dieser Arbeit verwendete Restlet-Framework. – Auch diese Arbeit liefert im Anhang einige Beispiele um zu zeigen, wie die beschriebenen Dinge praktisch umgesetzt werden.

Rohit Khare entwickelte in seiner Dissertation aus dem Jahr 2003[4] verschiedene Derivate des Representational State Transfer für verteilte Systeme, welche allerdings teilweise den Architekturbedingungen von Roy Fielding widersprechen .

Zur JAX-RS-Spezifikation existieren noch keine Bücher, da der Spezifikationsprozess zum Zeitpunkt der Erstellung dieser Arbeit noch nicht abgeschlossen ist. Informationen dazu finden sich im Internet[5]. In etlichen Blogs etc. finden sich Beiträge über diese Spezifikation. Dabei ist zu beachten, dass viele dieser Beiträge nicht mehr ganz aktuell sind, da in der Zwischenzeit – je nach Veröffentlichungsdatum des Beitrags – mehr oder weniger große Details verändert wurden.

Ansonsten gibt es zum Thema REST nicht viele Bücher, weil das Thema relativ neu ist und vieles direkt im Internet behandelt wird. Suchen bei Amazon und der Google Buchsuche (z. B. 6. 3. 2008 und auch am 7. 7. 2008, deutsche und englische Ausgabe) ergab keine Hinweise auf weitere Bücher, die sich im wesentlichen mit dem REST-Architekturstil beschäftigen.

[3] Buch „RESTful Web Services“: [Richardson und Ruby 2007a];
deutsche Übersetzung: „Web Services mit REST“ [Richardson und Ruby 2007b]

[4] „Extending the REpresentational State Transfer Architectural Style for Decentralized Systems“: [Khare 2003]

[5] Infos zum JAX-RS-Spezifikationsprozess: [Hadley und Sandoz 2007b] und [Hadley und Sandoz 2008d]

1.5 Begriffsdefinitionen

Da einige Begriffe in verschiedenen Kontexten verschieden verwendet werden, konkretisiert dieser Abschnitt einige Begriffe , so wie sie in dieser Arbeit verwendet werden. Begriffe, die ggf. nicht so bekannt sind, deren Bedeutung aber fest steht, werden im Glossar (ab Seite 123) erklärt.

Dienst

Ein Dienst (oder auch Service) stellt seine Fähigkeiten anderen Nutzern zur Verfügung. Der Begriff wird in dieser Arbeit völlig unabhängig von Protokollen oder Programmiersprachen verwendet. Ein solcher Dienst kann z. B. als SOAP-Web-Service realisiert werden, aber auch als Dienst direkt mit dem HTTP im REST-Architekturstil, mit EJB, CORBA etc.

Web-Dienst

Ein Web-Dienst ist in dieser Arbeit ein Dienst, der mit Web-Technologien (insbesondere Klartext-Protokollen) implementiert ist. Ein Web-Dienst muss so aufgebaut sein, dass er leicht über Firmengrenzen hinweg funktioniert. Der englische Begriff Web-Service wird häufig mit SOAP-Web-Services gleichgesetzt. Deshalb wird in dieser Arbeit der Begriff Web-Dienst für die allgemeine Bezeichnung solcher Dienste verwendet.

Web-Service

Mit dem Begriff Web-Service ist in der Literatur meistens ein SOAP-Web-Service nach den WS-*-Spezifikationen [6] gemeint, manchmal wird er aber auch allgemeiner im Sinne von Web-Dienst verwendet, wie in der vorigen Erklärung beschrieben. Deshalb wird der Begriff Web-Service in dieser Arbeit nicht weiter verwendet. Entweder wird allgemein der Begriff *Web-Dienst* (als Konzept) oder speziell der Begriff *SOAP-Web-Service* benutzt.

Endanwender

Ein Endanwender nutzt eine fertige Software, z. B. mit Hilfe eines Browsers.

Anwendungsentwickler

Ein Anwendungsentwickler ist im Rahmen dieser Arbeit ein Software-Entwickler, der in sich geschlossene Anwendungen für Endbenutzer oder andere Anwendungen schreibt. Dieser Begriff wird in dieser Arbeit weitläufig verwendet, so dass bspw. auch Software-Architekten gemeint sind.

[6] für SOAP-Web-Services siehe z. B. [W3C 2008]

Entwickler einer Laufzeitumgebung

Eine in dieser Arbeit besonders behandelte Gruppe sind Entwickler von Laufzeitumgebungen. Ihr Produkt wird von anderen Anwendungsentwicklern genutzt. – Eine Laufzeitumgebung alleine ist von Endanwendern nicht nutzbar.

REST-konform, REST-System, REST-Server

Um Sachverhalte kurz formulieren zu können, werden an einigen Stellen Begriffe wie REST-konform, REST-System, REST-Server etc. verwendet. Sie beschreiben Softwaresysteme etc., die den im Architekturstil REST vorgegebenen Architekurbedingungen entsprechen. Diese Begrifflichkeit ist zwar etwas salopp, aber bringt die Dinge prägnant auf den Punkt.

Sprache

Mit einer Sprache ist in dieser Arbeit eine der menschlichen Sprachen wie deutsch, englisch etc. gemeint. Sollte eine Programmiersprache gemeint sein, wird explizit der Begriff Programmiersprache verwendet.

Die in der JAX-RS-Spezifikation definierten Begriffe sind im Abschnitt 3.2 auf Seite 44 dieser Arbeit im Text beschrieben.

Die in dieser Arbeit betrachtete Spezifikation beinhaltet auch eine Behandlung verschiedener Media-Typen / Mime-Typen („text/xml“, „text/html“, „image/jpeg“ etc.). Da es keine richtige Übersetzung ins Deutsche für diese Begriffe gibt, wird in dieser Arbeit wird der Begriff „Datenformat“ verwendet.

Der Begriff HTTP-Protokoll wird so nicht verwendet, weil das P schon Protokoll bedeutet. Abkürzend wird von *dem HTTP* gesprochen.

2 Grundlagen

2.1 Representational State Transfer

Der Architekturstil *Representational State Transfer* (abgekürzt mit REST) wurde von Roy Thomas Fielding entwickelt und im Jahr 2000 in seiner Dissertation veröffentlicht [Fielding 2000]. Dieser Abschnitt beschreibt die theoretischen Grundlagen des REST.

Der Representational State Transfer beschreibt u. a. wohl geformte Web-Anwendungen aus der Sicht von Roy Fielding, einem der Spezifikatoren des HTTP. Dabei stehen besonders verteilte Hypermedia-Anwendungen [1] im Fokus, nicht zwingend alle Web-Anwendungen.

Motiviert wurde dieser Architekturstil dadurch, dass es um 1993 nur informelle Beschreibungen für das World Wide Web gab, aber keine echten Standards, so dass die vorhandenen Implementierungen nicht zueinander kompatibel waren. Gleichzeitig gab es Druck von der Industrie, endlich einen oder nur wenige Standards zu schaffen. Deshalb haben Roy Fielding und andere HTTP/1.0 und später HTTP/1.1 spezifiziert.

REST als Architekturstil wurde entwickelt, um eine Möglichkeit zu zeigen, wie das Web gut funktioniert. Außerdem wurde REST beschrieben, um verschiedene Probleme – u. a. im HTTP – zu identifizieren, Alternativen zu vergleichen und um sicherzustellen, dass Protokollerweiterungen nicht das zerstören, was das Web so erfolgreich gemacht hat. Diese Beschreibung war Teil der Arbeit von IETF und W3C, um Architektur-Standards für HTTP, URI und HTML zu entwickeln. Der REST beschreibt einige Architektur-Bedingungen, u. a. um die Latenz der Anwendungen und die Netzwerkbelastung zu senken, bei gleichzeitig möglichst loser Kopplung der Komponenten.

Der Name Representational State Transfer leitet sich aus dem Netzwerk von Web-Seiten ab, durch die der Benutzer durch das Wählen von Links von einer Seite zur nächsten wechselt. Das Netzwerk von Web-Seiten ist in diesem Modell eine virtuelle Zustandsmaschine. Der Zustand der Anwendung (*state*) wird durch die aktuell ausgewählte Web-Seite (genauer: einer *Repräsentation*) dargestellt. Der Anwender wechselt den Zustand, indem er durch das Wählen von Links von einer Seite zur nächsten wechselt, d. h. den Zustand der Anwendung *transferiert*.

Dieser Abschnitt basiert auf der Dissertation von Roy Fielding [Fielding 2000].

[1] Hypermedia bezeichnet den multimedialen Hypertext.

2.1.1 Ressourcen

Im REST-Architekturstil stehen Subjekte – die Entitäten (Informationsobjekte) – im Zentrum. Jede Information, die auf einem Server identifiziert werden kann, ist im REST-Kontext eine Entität. Eine *Ressource* ist das Mapping auf eine oder auch mehrere Entitäten. Eine entsprechende Architektur wird auch „Resource-Oriented Architecture“ (ROA) genannt. Eine Ressource kann auch eine leere Menge von Entitäten sein, wenn die Entität(en) später einmal angelegt werden soll(en).

Eine Ressource beschreibt immer eine Semantik. Die Semantik einer Ressource kann bei einer Bank bspw. „alle Konten des angemeldeten Benutzers“ sein, so dass sie für verschiedene Benutzer verschiedene Entitäen enthalten kann. Weitere Beispiele für Ressourcen sind Dokumente, Bilder, Videos, Geschäftsobjekte, etc. Auch eine Sammlung von Ressourcen und / oder Links zu anderen Ressourcen ist wieder eine Ressource. Aus praktischer Sicht des Webs betrachtet ist alles das eine Ressource, was das Ziel eines Hypertext-Links sein kann.
Roy Fielding hat eine Ressource formal als Zugehörigkeitsfunktion (Membership function) definiert. Sie beschreibt, welche Entitäten zu einem gegebenen Zeitpunkt zu einer Ressource gehören.

Ressourcen können statisch – also unveränderlich – sein, z. B. fertige Dokumente, Fotos, die Wettervorhersage vom 9. 7. 2008 für den 10. 7. 2008. Sie können aber auch sich ständig ändernd sein, z. B. der aktuelle Status eines Dokumentes, die aktuelle Wettervorhersage für den 10. 7. 2008 oder aktuelle Stau-Informationen. Eine Möglichkeit ist auch, dass Ressourcen nur temporär vorhanden sind, d. h. sie sind nach kurzer Zeit nicht mehr vorhanden / verfügbar.
Eine Entität kann auch auf mehrere Ressourcen gemappt sein. Im Beispiel des Wetterberichtes ist am 9. 7. 2008 die Vorhersage für den 10. 7. 2008 sowohl als „Wettervorhersage für den 10. 7. 2008“ ansprechbar, als auch als „Wettervorhersage für morgen“.

Wie schon kurz erwähnt, können auch Geschäftsobjekte als Ressourcen modelliert werden. Dabei werden immer die Geschäftsobjekte selber angesprochen, nicht der Dienst, über den auf sie zugegriffen wird. Mehr dazu folgt im Abschnitt 2.3 ab Seite 28. – Zur Praxis der Bearbeitung von Ressourcen siehe Abschnitt 2.2.1 ab Seite 22.

Ressource-Identifier

Um auf Ressourcen zugreifen zu können, müssen sie eindeutig referenzierbar sein. Dafür gibt es das Konzept des *Ressource-Identifiers.* Ressource-Identifier sind immer *Nomen*, weil sie *Subjekte* identifizieren. Der Ressource-Identifier repräsentiert dabei die Semantik der angefragten Ressource. Datenformat etc. sind im Konzept der Ressource-Identifier erst einmal unwichtig.

In einem Netz von Ressourcen (z. B. einem Webauftritt) dürfen Clients – abgesehen vom Start-Resource-Identifier – nur die Identifier verwendet werden, die vom Server übertragen wurden. Wenn Clients die Ressource-Identifier selber zusammenbauen sollen, dann müssten sie auch wissen, wie die einzelnen Parameter der Anfrage zu einem Ressource-Identifier zu kombinieren sind. Bei Webauftritten werden die Links immer vom Server vorgegeben. Sonst wäre der Webauftritt praktisch nicht nutzbar. Von Web-Diensten werden die Links in der Praxis häufig nicht vorgegeben, sondern müssen von den Clients selber zusammengebaut werden (vgl. Abschnitt 2.3.2 ab Seite 30). Dies entspricht nicht dem REST-Architekturstil.

Es ist nicht nötig, dass die Clients die vom Server empfangenen Ressource-Identifier verstehen oder interpretieren können. Die Übertragung der Ressource-Identifier durch den Server ermöglicht es auch, dass – wenn nötig – Teile der Anwendung auf einen anderen Server verschoben werden. Solange der zentrale URI gleich bleibt, kann die Anwendung uneingeschränkt genutzt werden. Dies erhöht die Flexibilität der Verteilung der Anwendung.
Wenn die Ressource-Identifier (also die öffentliche Schnittstelle) abstrakt genug sind, dann kann auf dem Server die Software ausgetauscht werden, ohne dass die Schnittstelle geändert werden muss. Dazu sind technikunabhängige [2] Ressource-Identifier sehr vorteilhaft.

Ressource-Identifier dürfen im REST-Architekturstil nicht den Namen des eingeloggten Benutzers o. ä. enthalten, um den Benutzer zu ermitteln. Dazu sind die Authentifizierungsmechanismen des verwendeten Protokolls (HTTP, ...) zu verwenden. Ein wesentlicher Grund dafür ist, dass ein Ressource-Identifier den Zustand der aktuellen Interaktion des Clients mit dem Server beschreibt. Wenn Anwender Anton bspw. einen Link an Anwenderin Berta weitergibt, der Link aber einen Benutzernamen von Anton enthält, dann kann Berta mit dem Link nichts anfangen, da sie als Benutzer Anton auf den Server zugreift, aber sich nicht als Anton authentifizieren kann.
Ausnahmen sind Dienste wie Social Bookmarking. Hier ist der Benutzername manchmal Teil des Ressource-Identifiers, weil jeder Anwender auf die Bookmarks aller Anwender zugreifen können soll. Dafür ist dann der Benutzername des gewünschten Anwenders Teil des Ressource-Identifier, denn hier gehört der Ressource-Identifier zur Semantik der Anfrage.

[2] Technikunabhängigkeit bedeutet hier, dass der Resource Identifier keinen Hinweis auf die verwendete Technik enthalten, d. h. bspw., dass die Ressource-Identifier nicht auf „.php“ o. ä. enden.

Uniform Resource Identifier

Als Ressource-Identifier wird im Internet – unabhängig vom REST-Architekturstil – typischerweise der *Uniform Resource Identifier (URI)*[3] verwendet. Da der Ressource-Identifier ein REST-Konzept ist, wird die typische Umsetzung ebenfalls hier beschrieben.

Nach der Meinung von Tim Berners-Lee – dem Erfinders des Webs – ist der URI das wichtigste Konzept der Web-Architektur überhaupt, weil jede Verknüpfung im Web über einen URI adressiert wird, egal welches Protokoll (HTTP, FTP, ...) verwendet wird. Dieser Meinung schließt sich auch Roy Fielding an[4].

Aufbau des URI Der Aufbau des URI ist aktuell im RFC 3986[5] spezifiziert. An diesem RFC sind u. a. die schon genannten Tim Berners-Lee und Roy Fielding beteiligt.
Der Uniform Resource Identifier soll – wie der Name sagt – uniform, d. h. einheitlich sein.[6] Deshalb ist der Aufbau so allgemein gehalten wie möglich:

<Schema>**:**<Schema-spezifischer Teil>

Beispiele für das Schema sind *http*, *ftp*, *gopher*, *wais*, *file*, *mailto*, *ldap* und *news*.

Der Schema-spezifische Teil kann in jedem Schema anders definiert werden. Im Fall von *http* und *ftp* ist dies der Hostname, gefolgt meist von einem Pfad (häufig Verzeichnisstruktur und ein Dokument-/Dateiname[7]). Der Pfad kann in einzelne Pfad aufgeteilt werden; sie werden typischerweise (aber nicht zwingend) durch „/“ voneinander getrennt. Wenn es nötig ist, können auch Benutzername und Passwort, bei Bedarf eine vom Standard abweichende Portnummer, einen oder mehrere Anfrageparameter (die *Query*) und – hinter einem # – noch ein Fragmentbezeichner (z. B. ein HTML-Anker) im URI angegeben werden. Am Ende des Pfades sind mehrere durch Semikolon abgetrennte Parameter erlaubt, ähnlich den Anfrageparametern in der Query. Sie werden als Matrix-Parameter bezeichnet[8].

Schon in den anfänglichen Überlegungen von Tim Berners-Lee war vorgesehen, dass der Teil des URI hinter dem Host-Namen nicht zwingend auf eine reale Verzeichnisstruktur mit Dateien abgebildet werden muss, sondern vom Server bzw. dessen Administratoren und Entwicklern völlig frei definierbar sein soll und alles ansprechen kann: Den Namen einer

[3] Da die deutsche Übersetzung des Uniform Resource Identifier männlich ist (der Identifizierer), wird auch die Abkürzung URI männlich verwendet.
[4] URI ist das wichtigste Konzept des Webs: [Berners-Lee 1999b], [Fielding 2000, Abschnitt 6.2]
[5] URI-Spezifikation: [RFC-3986 2005]
[6] Tim Berners-Lee hatte ursprünglich *Universal* vorgesehen, womit er sich im Standardisierungsgremium aber nicht durchsetzen konnte.
[7] Anfang der 1990er-Jahre hieß der URI deshalb auch *Uniform Document Identifier* (UDI).
[8] Matrix-Parameter sind in der aktuellen URI-Definition nicht mehr vorhanden. In älteren RFCs sind sie jedoch aufgeführt, siehe [RFC-1808 1995, Abschnitt 2.4.5] oder [Berners-Lee 1996].

Tabelle oder eines Kontos, die Koordinaten auf einer Landkarte oder sonst irgendetwas. Es ist für den Client völlig irrelevant, weil er es über die einheitliche URI-Schnittstelle anfordern und bekommen kann, ohne Detailswissen über den Server zu benötigen.

In der inzwischen veralteten URI-Definition (RFC 2396) waren die Anfrageparameter als ein von der Ressource selber zu interpretierender String definiert. Mehr ist dort nicht festgelegt. Daraus ergab sich u. a., dass die Anfrageparameter nicht zur Identifikation der Ressource selber dienen konnten, da sie ja von ihr interpretiert werden sollten.
Im aktuellen RFC zur Spezifikation des URI (RFC 3986) ist die Abgrenzung zwischen Pfad und Anfrageparametern zur Identifikation der Ressource nicht mehr vorhanden. Jetzt dienen Pfad und Anfrageparameter zusammen zur Identifikation der Ressource.

Der Autor dieser Arbeit empfiehlt für den öffentlichen URI trotzdem die Trennung nach dem RFC 2396, um eine saubere Trennung von Ressource-Identifier und den an die Ressource übergebenen Parametern zu ermöglichen. Ob die Interpretation der Anfrage dann auch nach dieser Trennung erfolgt, oder ob der Web-Server den URI vor dem Mapping auf die konkrete(n) Entität(en) mittels URL-Rewriting [9] umbaut, ist davon unabhängig.

Andere Ressource-Identifier

Außer dem URI gibt es noch andere Ressource-Identifier. Ein Beispiel sind WHERE-Clauses in SQL-Anfragen: Mit ihnen lassen sich Ressourcen (in diesem Fall einzelne Tupel einer Relation) aus der Menge der Tupel der SELECT-FROM-Anfrage identifizieren.

Repräsentationen

Wie in nächsten Abschnitten weiter ausgeführt wird, besteht ein REST-System u. a. aus Clients, die Anfragen an Server stellen. Eine Anfrage liefert als Ergebnis immer eine *Repräsentation* der angefragten Ressource zurück, nicht die Ressource selber. Das bedeutet, dass der Server *immer* die Masterkopie der Entität behält.

Ein wichtiger Punkt, der sich auch im Namen Representational *State Tranfer* wiederfindet ist, dass die Anwendung über vorgegebene Links von einem Zustand (*state*) in den nächsten wechselt (*transfer*). Im Web wird dies durch die Links realisiert, die ein Benutzer im Browser anklicken kann.

Das Datenformat der Repräsentation kann nach Wünschen des Clients und Beschaffenheit der Ressource variieren. Der Server wählt nach den vom Client übertragenen Wünschen

[9] URL-Rewriting ist eine Möglichkeit, dass ein Web-Server beliebige angefragte URLs umbaut, bevor die Anfrage auf Skripte etc. abgebildet wird.

und seinen eigenen Möglichkeiten ein Datenformat aus und gibt eine entsprechende Repräsentation zurück. So kann der Client bspw. angeben, dass er eine Ressource gerne als HTML-Datei bekommen möchte, alternativ im PDF-Format.

Außer der Repräsentation der Ressource bekommt der Client auch Metadaten über die Repräsentation (z. B. das Datenformat oder den Zeitpunkt der letzten Änderung).

Ressourcen vs. Objekte

Relativ leicht drängt sich die Frage der Analogie von Ressourcen und (verteilten) Objekten auf. Ressourcen sind aber keine Objekte.[10]

Objekte werden – häufig von einer Factory – instantiiert. Sie haben einen Lebenszyklus und sind deshalb zustandsbehaftet. Während des Lebenszyklusses kann der Client alle Methoden des Objektes beliebig aufrufen u. ä. Wenn der Client das Objekt nicht mehr braucht, muss er es (explizit oder implizit) freigeben. Bei verteilten Objekten bleiben die Daten vor dem Client versteckt. Über Methoden kann er auf die Originaldaten zugreifen. Je nach Protokoll kann er sogar exklusiven Zugriff auf ein Objekt bekommen.

All diese Dinge existieren im REST-Architekturstil nicht. Der Client bekommt keinen direkten Zugriff auf das auf dem Server liegende Objekt und braucht es auch hinterher nicht freigeben. Er bekommt – wie schon angeführt – nur eine Repräsentation der Ressource, welche natürlich ein Objekt auf dem Server darstellt. Aber er bekommt „nur“ eine Kopie der Daten. Die Repräsentation weiß nicht, wo sie herkommt, sie wird nicht snychronisiert o. ä. In einem REST-Server gibt es auch keine Status-Informationen über verschiedene Aufrufe hinweg (siehe Abschnitt 2.1.3 auf Seite 14), so dass auch keine Objektreferenzen zwischen den beteiligten Prozessen verwaltet werden müssen. Ressourcen sind also leichtgewichtiger als Objekte im Sinne der Objektorientierung.

Verteilte Objekte sind für homogene Umgebungen innerhalb einer Firma geschaffen und funktionieren dort sehr gut. Im Web funktioniert dies, u. a. wegen gesperrten Firewalls, nicht gut. Web-Dienste sind technisch für den Zugriff aus dem ganzen Internet konzipiert. Web-Dienste können vorhandene verteilte Objekt-Systeme teilweise kapseln, so dass sie mit den Web-Dienst-Vorteilen verwendet werden können. Andererseits lässt sich die Kommunikation verteilter Objekte auch in Web-Dienste verpacken, aber dafür sind Object-Request-Broker (ORB) und die dafür entwickelten Protokolle besser geeignet.

[10] Der Inhalt dieses Abschnitts stammt nicht aus der Dissertation von Roy Fieldung, sondern aus [Vogel 2003].

2.1.2 Connectoren

Connectoren sind im REST-Kontext die Bindeglieder aus einem REST-System heraus. Sie kommunizieren mit einem anderen Connector eines (meist) anderen REST-Systems. Connectoren sind eine abstrakte Schnittstelle für die Kommunikation zwischen den Komponenten. Sie sind möglichst einfach und erlauben eine klare Trennung der Verantwortlichkeiten von Anwendungslogik und Kommunikation und auch eine Trennung zwischen dem gewünschten Kommunikationsmechanismus und dessen konkreter Implementierung.

Da die gesamte Kommunikation zustandslos ist, brauchen die Connectoren keine Zustände zwischen verschiedenen Aufrufen zu speichern. Weiterhin können auch parallele Anfragen gestellt werden, ohne dass zwischen diesen beiden irgendein für die Anfrage relevanter Zusammenhang bestehen kann.

Im REST-Architekturstil existieren folgende Connectoren:

Client-Connector Der Client-Connector ist der „Ausgang“ für eine Anfrage aus den Clients (z. B. Web-Browsern) heraus und der „Eingang“ für die Antworten. Bevor der Zugriff auf das Netzwerk stattfinden kann, muss häufig auf den Resolver-Connector (siehe unten) zugegriffen werden, um den angegeben Hostnamen zu der dazugehörigen IP-Adresse aufzulösen.

Server-Connector Der Server-Connector ist der „Eingang“ in den Server für Anfragen von Clients und der „Ausgang“ für die dazugehörigen Antworten. Der Server-Connector wartet ständig, bis eine Anfrage kommt, und wird erst dann aktiv. Ein REST-System kann sowohl Client- als auch Server-Connectoren enthalten.

Cache-Connector Der Cache-Connector kann sowohl in der Client-Connector-Schnittstelle als auch in der Server-Connector-Schnittstelle integriert oder ihnen vorgeschaltet sein. Bevor der Cache-Connector die Anfrage an den Server (auf der Clientseite) stellt bzw. sie auf der Serverseite an die eigentliche Server-Logik weiter gibt, prüft er, ob er die passende Antwort schon gecacht hat und gibt sie ggf. zurück, so dass die Netzwerkanfrage bzw. die Bearbeitung durch den Server entfallen kann.

Weiteres zum Caching findet sich im Abschnitt 2.1.3 ab Seite 15.

Resolver-Connector Der Resolver-Connector löst den Ressource-Identifier oder Teile davon auf, um die für den physikalischen Netzwerkzugriff benötigten Daten zu bekommen. Ein typisches Beispiel ist das Dynamic Name System (DNS), welches die zu einem Hostnamen gehörige IP-Adresse ermittelt. Ein URI kann aber ggf. auch noch weiter aufgelöst / verändert werden.

Tunnel-Connector Der letzte Connector ist der Tunnel: Er tauscht Daten über Bereichsgrenzen hinaus aus, bspw. über Firewalls hinweg. Tunnel wurden in die REST-Architektur aufgenommen, da auch Proxies Tunnel sein können, wenn der Client sich über die HTTP-Methode CONNECT direkt mit einem Server verbindet, bspw. über dann getunneltes HTTPS.

2.1.3 REST-Architektur-Konzepte

Der Architekturstil REST verbindet verschiedene Konzepte miteinander:

1. Client-Server-Modell
2. Zustandslosigkeit des Servers
3. Caching
4. einheitliche Schnittstellen
5. Schichten
6. Code-on-Demand (optional)

Sie werden auf den folgenden Seiten beschrieben.

Client-Server-Modell

Das Hauptmerkmal eines Client-Server-Systems ist, dass es eine Gruppe von Komponenten gibt, die Dienste anbieten (Server, dt. Diener) und andere Komponenten, die diese Dienste nutzen (Clients, dt. Kunden). Die Verantwortungsbereiche sind so klar voneinander getrennt. Der Client kann die Angebote des Servers nutzen und die Antworten entsprechend seinen eigenen Wünschen und Fähigkeiten weiterverarbeiten (z. B. anzeigen). Aktionen gehen in

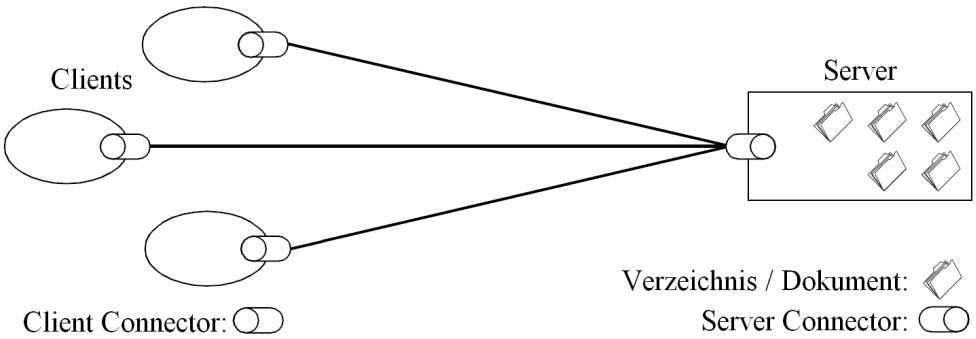

Abbildung 2.1: REST: Client-Server-System (angelehnt an [Fielding 2000, Abb. 5-6])

REST-Systemen *immer* vom Client aus, auf welche der Server *immer* mit genau *einer* Antwort reagiert.

Ein wesentlicher Vorteil eines Client-Server-Systems ist, dass Client und Server unabhängig voneinander entwickelt und auch unabhängig voneinander ausgetauscht werden können, wenn beide Seiten die Schnittstellendefinition einhalten.

Zustandslosigkeit des Servers

Zustandslosigkeit bedeutet in diesem Zusammenhang, dass der Server keinerlei Informationen über zusammenhängende Aufrufe speichert. Daraus folgt, dass der Client *alle* für eine Anfrage benötigten Informationen mitliefern muss. Session-IDs und ähnliches sind nicht erlaubt. So wird die Kopplung zwischen Client und Server kleiner.
Das bedeutet nicht, dass der Server keinerlei Daten speichern darf. Seine Geschäftsdaten muss er selbstverständlich speichern, wie bei anderen Architekturstilen auch. Das Verbot bezieht sich auf Informationen, die (noch) nicht als Geschäftsdaten gespeichert sind, bspw. schon im vorigen Formular eingegebenen Daten, zu denen später weitere Daten folgen.
Der Anwendungsentwickler muss den Zustand der Anwendung als Ressourcen modellieren. Dies ist für jemanden, der es gewohnt war Sessions zu benutzen, gewöhnungsbedürftig.

Ein Vorteil der Zustandslosigkeit ist, dass sich der Server nicht die Session-Informationen vieler oder sehr vieler Clients merken muss. Dies kostet bei vielen Sessions entsprechend viel Arbeitsspeicher. Wenn ein physikalischer Server für die Bearbeitung der Anfragen nicht ausreicht, und der Server-Betreiber mehrere parallele Server-Rechner installiert und sie mit Load-Balancing gleichmäßig auslastet, dann tritt in REST-Systemen auch nicht das Problem auf, dass Sessions synchronisiert werden müssen, da es keine Sessions gibt. Deshalb skaliert ein System ohne Zustandsinformationen wesentlich besser als mit Zustandsinformationen.

Ein weiteres Problem bei zustandsbehafteten Web-Anwendungen ist, dass Server manchmal erwarten, dass der Client an einer bestimmten Stelle im Arbeitsablauf ist. Wenn der Anwender im Browser den „Zurück“-Button verwendet hat, dann löst er als nächstes ggf. eine Anfrage aus, die der Server nicht erwartet und deshalb anders reagiert, als es vorgesehen ist und vom Anwender erwartet wird.
Das Problem abgelaufener Sessions tritt bei zustandslosen Systemen ebenfalls nicht auf.

Der Client ist der einzige Teil in dem Client-Server-System, der Zustandsinformationen speichern darf. Dies muss er in einem System mit Zuständen im Server aber auch, um dem Server mitzuteilen, welche Session „seine“ ist. Deshalb ergibt sich für den Client kein prinzipieller Nachteil.

Caching

Caches speichern schon einmal geladenen Informationen, um sie später bei Bedarf schnell verfügbar zu haben, ohne die eigentliche Datenquelle belasten zu müssen. In der Informatik gibt es Caches auf vielen Ebenen. Hier einige Beispiele:

- Die CPU hat einige Register, um nicht so häufig auf den im Vergleich relativ langsameren Arbeitsspeicher zugreifen zu müssen.
- Um langsame Festplatten-Zugriffe zu vermeiden, speichern Betriebssystem und Datenbankmanagementsysteme geladene Daten im Arbeitsspeicher zwischen.

Caching ist auch im REST-Architekturstil möglich: Bei nur lesenden Operationen können Zwischenergebnisse lokal im Client oder auf einem Server gecacht werden [11].

Beim Caching prüft der Client vor einer Anfrage an die Datenquelle, ob die benötigten Informationen schon im Cache vorhanden sind. Wenn ja, wird geprüft, ob sie noch aktuell (genug) sind. Wenn sie als aktuell angesehen werden, werden sie an den Anfragenden zurückgegeben. Eine erneute Abfrage bei der Datenquelle ist dann nicht nötig. Sind die Daten im Cache nicht vorhanden oder (vielleicht) veraltet, wird der aktuelle Stand bei der Datenquelle abgerufen und im Cache gespeichert bzw. die alte gecachte Version ersetzt.

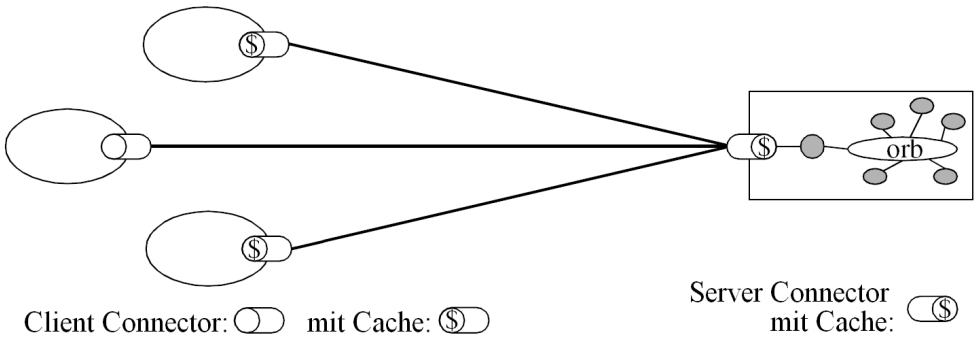

Abbildung 2.2: REST: cachendes Client-Server-System
(angelehnt an [Fielding 2000, Abb. 5-6])

Wenn die Cache-Verwaltung nicht prüfen kann, ob die gespeicherten Informationen noch aktuell sind, kann sie ggf. bei der Datenquelle z. B. den aktuellen Zeitstempel der Ressource erfragen. Wenn dieser neuer ist als der gecachte Wert, dann muss die Information neu übertragen werden. Andernfalls ist der lokal gespeicherte Stand aktuell.

Im Client wird der Cache (wie in der Abbildung im oberen und unteren Client zu sehen) vor den „Ausgang“ in das Netzwerk platziert. Im Server wird ein Cache im „Eingang“ platziert, um eingehende Anfragen daraufhin zu überprüfen, ob das Ergebnis der Anfrage schon einmal

[11] Proxies werden später eingeführt, siehe Seite 18.

erzeugt wurde und im Cache zur Verfügung steht. Dies lohnt sich besonders, wenn der Server die benötigten Daten seinerseits wieder von anderen Rechnern (im Beispiel über einen ORB, Object Request Broker) zusammensammeln muss.
Vorteil aus Sicht der Clients ist eine Geschwindigkeitssteigerung. Außerdem werden Ressourcen des Netzwerkes und der Server gespart, da weniger oder kleinere Anfragen gestellt werden müssen.

Ein allgemeines Problem von Caches ist, dass der Cache-Verwalter ggf. falsch entscheiden könnte, dass eine gecachten Information noch aktuell ist. In HTML-Dateien kann beispielsweise angegeben werden, wie lange die Seite gecacht werden darf. Wenn sich der Cache-Verwalter darauf verlässt, aber die Seite auf dem Web-Server in der Zwischenzeit geändert wurde, dann ist die Seite nicht mehr aktuell, wird aber trotzdem dem Client ausgeliefert.

Ein Server kann von ihm gesendete Anfragen explizit als cachable oder nicht cachable kennzeichnen. Um aktuelle Ergebnisse auch bei folgenden Anfragen sicherstellen zu können, dürfen als nicht cachable gekennzeichnete Antworten dann nicht gecacht werden.
Statische Web-Seiten sind typischerweise über einen relativ langen Zeitraum stabil, und es ist deshalb sinnvoll, sie zu cachen, weil sie sich nicht häufig ändern. Für andere Repräsentationen (z. B. aktuelle Wetterdaten) lohnt sich das Caching häufig nicht, weil sie sich ständig ändern.

Wenn die Antworten des Servers vom Sessionverlauf abhängen würden, müsste der Cache-Verwalter den Sessionverlauf nachverfolgen, bevor er eine Information als aktuell zurück geben kann. Diese Nachverfolgung ist aber sehr schwierig. Dies ist ein wesentlicher Grund, warum der REST-Architekturstil Zustandslosigkeit verlangt.

Auch ein System, welches das Caching nicht nutzt, aber ermöglicht, ist entsprechend dem REST-Architekturstil implementiert. Die Nutzung ist also nicht zwingend, muss aber möglich sein.

Einheitliche Schnittstellen

Ein wesentliches Kennzeichen, in dem sich REST-Systeme von anderen Netzwerk-basierten Architekturen unterscheidet ist die sehr einheitliche Schnittstelle zwischen den Komponenten.
Zum einen gibt es Ressourcen, die über Ressource-Identifier angesprochen werden können. Zum anderen gibt es wenige, universell definierte Methoden mit denen auf die Ressourcen zugegriffen wird. Diese Methoden lassen sich potentiell auf alle Ressourcen anwenden. Andere Methoden sind typischerweise nicht möglich. Der Client braucht also nur den URI wissen und kann dann die Ressource bearbeiten, weil die Methoden eine auf allen Ressourcen festgelegte Semantik haben.

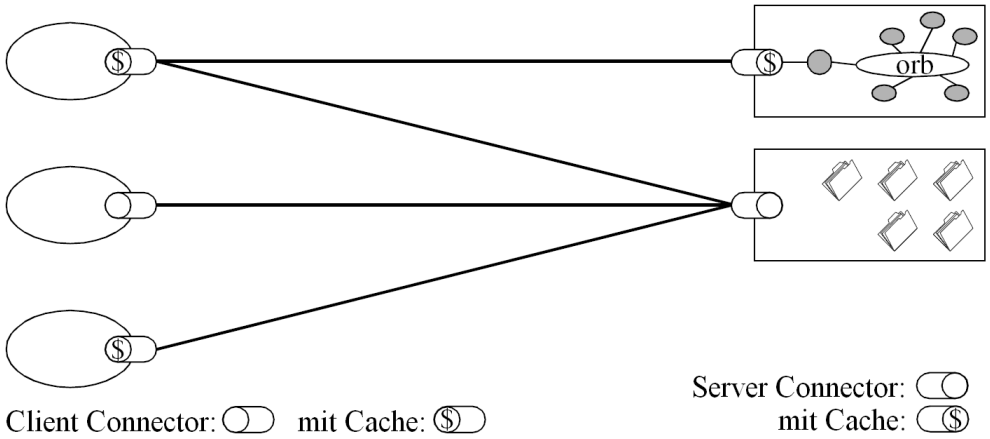

Abbildung 2.3: REST: cachendes Client-Server-System mit einheitlichen Schnittstellen (aus [Fielding 2000, Abb. 5-6], modifiziert)

Die Clients sind nicht an einen bestimmten Server gebunden, sondern sie können wegen der einheitliche Schnittstelle auf beliebige Server zugreifen.
Wie im rechten Teil der Abbildung 2.3 zu sehen, kann hinter der einheitlichen Schnittstelle eine beliebige Technik arbeiten: Dies kann eine aufwendige Anwendungslogik sein, die ggf. über andere Protokolle ihre Informationen zusammen sammelt. Der in der Abbildung untere Server liefert einfach direkt auf der Festplatte liegende Dokumente aus.

Standardisierte Schnittstellen haben aber immer auch den Nachteil, dass sie Overhead verursachen und es prinzipiell immer speziell ausgelegte, effizientere Möglichkeiten der Kommunikation gibt. Da der REST als universeller Architekturstil für Hypermedia-Anwendungen mit typischerweise größeren übertragenen Datenmengen, z. B. Videos, Musik, HTML-Dateien u. ä. konzipiert ist, fallen ein paar 10 Bytes mehr für die einheitliche Schnittstelle nur unwesentlich ins Gewicht.

Schichtung

Schichtung wird in quasi allen nicht-trivialen Software-Systemen eingesetzt. Eine Schicht nutzt die Dienste der unter ihr liegenden Schicht und bietet der über ihr liegenden Schicht Dienste an. Ein bekanntes Schichten-Modell ist das ISO/OSI-Referenzmodell für den Netzwerkverkehr[12]:

7. Anwendungsschicht
6. Darstellungsschicht
5. Sitzungsschicht

[12] ISO/OSI-Referenzmodell: z. B. [Tanenbaum und van Steen 2002]

4. Transportschicht
3. Vermittlungsschicht
2. Sicherungsschicht
1. Bitübertragungsschicht

Die Vermittlungsschicht im Internet bietet z. B. das Internet-Protokoll (IP) als Dienst an. Mit dem IP lassen sich Pakete zu einem bestimmten Ziel-Rechner bringen. Die Dienste UDP und TCP der Transportschicht nutzt diesen Dienst, um verschiedene höherwertige Dienste anzubieten [13].

Auch im REST-Architekturstil ist die Schichtung möglich. In service-orientierten Architekturen ist es erwünscht, dass vorhandene Dienste wieder verwendet werden und z. B. zu komplexeren orchestriert werden. Dabei fungiert ein Server dann auch als Client eines anderen Servers. Im REST-Archtekturstil ist – entgegen dem Beispiel des ISO/OSI-Referenzmodells – nicht festgelegt, wie tief die Schichtung sein soll und welche Schicht was zu erledigen hat. Dieses Modell entspricht auch dem Architekturmodell Pipe-and-Filters, wie es beispielsweise von Unix / Linux-Nutzern auf der Kommandozeile und in Skripten häufig verwendet wird [14].

Der Vorteil von Schichten ist, dass es innerhalb eines Systems wieder klar definierte Schnittstellen gibt, so dass einzelne Module relativ einfach ausgetauscht werden können.

Der Nachteil von Schichten ist, dass das Gesamtsystem – insbesondere aus Benutzersicht – langsamer wird, da eine Anfrage durch mehrere Schichten hindurch muss, die Daten ggf. zwischendurch konvertiert werden u. ä. Dies erhöht auch die Anforderungen an die Hardware, weil mehr CPU-Leistung und mehr Arbeitsspeicher benötigt wird. Wenn die Schichtung intelligent eingesetzt wird, kann sie das Gesamtsystem schneller machen.

Proxies sind dafür ein gutes Beispiel. Sie sind eine typische Kombination aus Schichtung und Caching. Sie sind eine weitere Schicht im REST-Modell, aber sie helfen – je nach Anwendungsfall und Platzierung der Proxies – die durch die Benutzer gefühlte Geschwindigkeit zu erhöhen, da Anfragen und Netzwerkverkehr und damit Zeit gespart werden können. Ein Vorteil des REST-Architekturstils ist in diesem Zusammenhang, dass Proxies aufgrund der einheitlichen Schnittstelle (s. o.) eine generische Software sind, so dass sie universell verwendet werden können, ohne dass für jede Anwendung eine neue Proxy-Software entwickelt werden muss.

Die vier Clients ganz links und oben in der Abildung 2.4 greifen über Proxies auf die verschiedenen Dienste zu. Dabei greift einer der beiden Proxies für einige Zugriffe auf einen weiteren Proxy zu. Auch dies ist möglich und manchmal sinnvoll. Ein Beispiel ist ein Proxy einer Firma, der auf einen Proxy des Internet-Providers zugreift. Der untere Client in der Abbildung greift ohne Proxy direkt auf den Quell-Server rechts zu, welcher in Wirklichkeit ein

[13] Hier bleibt unberücksichtigt, dass die Schichtung der Internetprotokolle teilweise grobkörniger ist. Die Vermittlungs- und die Transportschicht sind aber in beiden Protokollstapeln mit der gleicher Semantik enthalten.

[14] Pipe-and-Filters: z. B. [Meunier 1995]. Beispiel: `cat <Dateiname> | grep <Suchstring>`

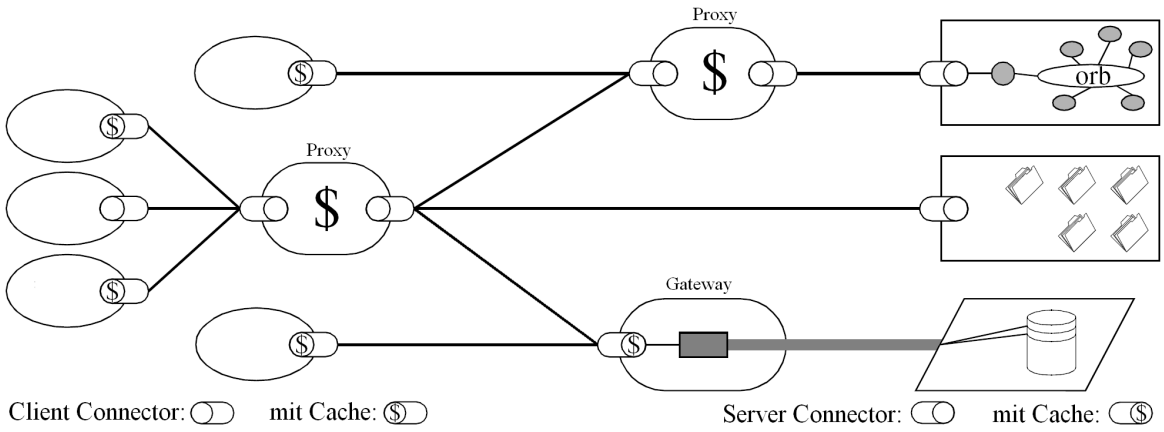

Abbildung 2.4: REST: geschichtetes, cachendes Client-Server-System, einh. Schnittstellen (angelehnt an [Fielding 2000, Abb. 5-7])

Gateway zu einem anderen System oder Netzwerk-Protokoll ist. Dies ist dem Client aber verborgen.

Der Proxy links in der Abbildung steht „nahe“ bei den Clients und bearbeitet für relativ wenige Clients Anfragen an viele Server. der Proxy rechts oben in der Abbildung cacht die Anfragen vieler Clients an eine kleine Gruppe von Servern. Dann werden die zentralen Server mit der Geschäftslogik einer Organisation entlastet. Dies ist bspw. bei Wikipedia so.

Code-on-Demand

Dem Prinzip Code-on-Demand liegt zugrunde, dass der Client ausführbaren Programmcode vom Server runter lädt und dann lokal ausführt.

Mit diesem Prinzip ist es möglich, dem Client auch dann noch Funktionalität hinzuzufügen, wenn er schon fertig implementiert und installiert ist, ohne dass eine neue Version des Clients verteilt werden muss. Prominenteste Beispiele des Webs sind JavaScripts, Java-Applets und Flash-Animationen. Die Programmlogik wird zentral auf einem Server bereitgestellt, und alle Clients können diese dann herunterladen und ausführen.

Dies setzt voraus, dass die Clients eine entsprechende Laufzeitumgebung zur Verfügung stellen. Wenn es verschiedene mögliche Clients (Web-Browser) gibt, dann funktioniert dies nur dann gut, wenn die Laufzeitbibliotheken aller Clients die benötigten Funktionalitäten zur Verfügung stellen, d. h. auch, dass die Laufzeitbibliothek der angeforderten Version installiert sein muss. Innerhalb einer Organisation mit einer zentral organisierten Datenverabeitung ist

dies relativ leicht möglich. Allerdings kann man bei Anwendungen für Endbenutzer im ganzen Internet nicht voraussetzen, dass alle potentiellen Interessenten die benötigte Laufzeitumgebung installiert und aktiviert haben.

Code-on-Demand ist im REST-Architekturstil ein optionales Feature, d. h. es ist nicht vorgeschrieben wie die bisher erwähnten Einschränkungen, aber explizit erlaubt. Eine Anwendung ist auch dann konform zum REST-Architekturstil implementiert, wenn kein auszuführender Code existiert.

2.1.4 Zusammenfassung

Der Representational State Transfer beschreibt einen Architekturstil mit folgenden Kennzeichen:

- Ressourcen-Orientierung,
- Client-Server-Orientierung,
- Zustandslosigkeit des Servers,
- Caching,
- einheitliche Zugriffsschnittstellen,
- geschichtete Systeme und
- explizit erlaubtes Nachladen von ausführbarem Programmcode.

Dies trifft z. B. auf Web-Anwendungen zu, wie sie nach einem der Erfindern des HTTP aussehen sollten. Es lassen sich aber auch andere Systeme damit modellieren.

Die folgende Abbildung zeigt alle vorgestellten Architektur-Elemente:

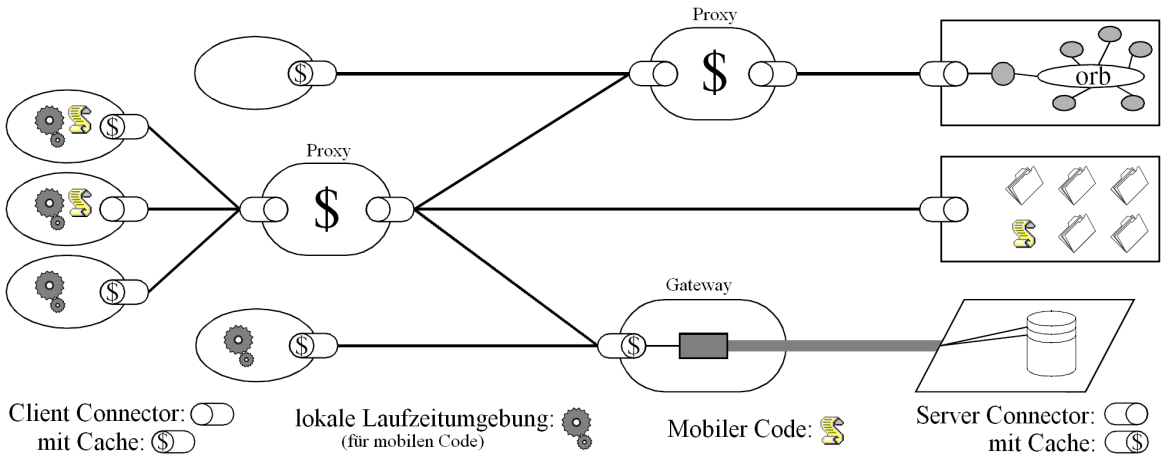

Abbildung 2.5: Representational State Transfer (angelehnt an [Fielding 2000, Abb. 5-8])

2.2 HTTP

Da der Architekturstil REST für das Web entwickelt wurde, wird er meistens mit dem Web-Protokoll HTTP verwendet. Das bedeutet aber nicht, dass REST nur mit dem HTTP realisiert werden kann, auch wenn dies häufiger vom REST behauptet und dann als Nachteil dargestellt wird (siehe dazu auch Kapitel 2.5 ab Seite 39). Es können auch andere Protokolle z. B. FTP, Gopher [15] oder WAIS [16] verwendet werden.
Die in dieser Arbeit zu betrachtende JAX-RS-Spezifikation (JSR 311) basiert ebenfalls auf dem HTTP. Deshalb wird das HTTP hier vorgestellt, wobei der Schwerpunkt auf den für die Implementierung dieser Spezifikation relevanten Punkten liegt.

Das HTTP wurde 1990 von Tim Berners-Lee entwickelt. Er bescheibt die Geschichte u. a. in dem Buch „Weaving the Web“ [17]. Die Version 0.9 des HTTP hat Tim Berners-Lee 1991 festgeschrieben [18]. 1996 folgte Version 1.0, 1997 Version 1.1 [19]. Die Versionsnummern werden bei HTTP üblicherweise direkt mit einem Schrägstrich an „HTTP“ angehängt, z. B. „HTTP/1.0“. Diese Notation wird in dieser Arbeit auch verwendet.
Das HTTP ist ein zustandsloses Request-Response-Protokoll, wie es von Roy Fielding für Anwendungen nach dem REST-Architekturstil gefordert wird (siehe Abschnitt 2.1.3 ab Seite 14). Das bedeutet, dass ein Client (z. B. ein Web-Browser) eine Anfrage an einen HTTP-Server stellt, der Server diese Anfrage dann mit genau einer Repräsentation beantwortet und der HTTP-Stack des Servers dann alle Informationen über diese konkrete Anfrage vergisst.

Auf eine Ressource auf einem HTTP-Server wird mittels einer Methode zugegriffen. Sie gibt an, ob der Server eine Repräsentation der Ressource ausgeben soll (GET), eine neue Ressource anlegen soll (POST oder ggf. PUT) o. ä. Das HTTP ist also ein Anwendungsprotokoll, nicht nur ein reines Transportprotokoll.
Außer der Anfragezeile (sie enthält die Methode und den URI der Ressource) werden seit HTTP/1.0 in einer Anfrage noch verschiedene Kopfzeilen (siehe Abschnitt 2.2.3) und bei Bedarf ein Nachrichtenkörper (die Entität der Anfrage) übertragen. Die Kopfzeilen übertragen Metadaten zur Anfrage, der Nachrichtenkörper enthält die eigentlichen Nutzdaten. Eine Antwort enthält als erste Zeile einen Statuscode als maschinenlesbare Beschreibung der Antwort (siehe Abschnitt 2.2.2) und dann ebenfalls Kopfzeilen und den Nachrichtenkörper.

[15] Das Gopher-Protokoll vereinfacht gegenüber FTP den Zugriff auf Dokumente, weil keine manuellen Befehle wie bei FTP eingegeben werden müssen. Das Navigieren ist deshalb einfacher.
[16] WAIS (Wide Area Information Server System) ist ein Volltextsuchsystem. WAIS durchsucht anhand von eingegebenen Schlüsselwörtern die Texte auf den Servern und listet die Fundstellen auf. WAIS ist also ein Vorläufer der Volltext-Suchmaschinen.
[17] „Weaving the Web“: [Berners-Lee 1999a]
[18] HTTP/0.9: [Berners-Lee 1991]
[19] HTTP/1.0: [RFC-1945 1996]; HTTP/1.1: [RFC-2068 1997], inzwischen ersetzt durch [RFC-2616 1999]

2.2.1 Methoden

Die einheitliche Schnittstelle des HTTP wird neben den URIs zur Identifizierung der Ressourcen durch wenige, allgemein definierte Methoden hergestellt. Alle HTTP-Methoden haben eine wohldefinierte Semantik, die sie auf allen Ressourcen haben, auf denen sie erlaubt sind. Die HTTP-Methoden werden in diesem Abschnitt kurz vorgestellt:

GET Die GET-Methode dient dazu, Repräsentationen von Ressourcen abzurufen. Sie soll für nichts anderes verwendet werden, als um Daten auszuliefern. Das bedeutet auch, dass GET idempotent und seiteneffektfrei sein muss.

HEAD HEAD ist identisch zu GET, außer dass nie ein Body einer Antwort übertragen wird. Diese Methode dient zur Entlastung des Netzwerkes, wenn nur der Nachrichtenkopf interessant ist, z. B. wenn nur geprüft werden soll, ob die Datei existiert, der Inhalt jedoch (erstmal) irrelevant ist.

POST Mit POST werden die übergebenen Informationen an die angegebene Ressource angehängt. Ein Beispiel dafür sind Foren o. ä., wo der angegebenen Ressource (z. B. einem Forums-Thread) ein neuer Artikel hinzugefügt wird.

PUT PUT dient dazu, vorhandene Ressourcen zu ändern oder mit dem angegebenen URI neu anzulegen. Dies setzt – wie bei den anderen Methoden auch – voraus, dass der Benutzer das Recht hat, die Methode auf dieser Ressource auszuführen (siehe Abschnitt 2.2.4, S. 24). Diese Methode sollte idempotent implementiert werden, d. h. ein erneuter Aufruf der Methode auf der gleichen Resource mit der gleichen Entität verändert das Ergebnis nicht.

Der Unterschied zwischen dem Anlegen mit PUT und POST ist, dass bei POST die neue Ressource an die angegebene Ressource angehängt wird (als Kindsknoten), während bei PUT der URI der neuen Ressource direkt vorgegeben wird.

DELETE Mit DELETE können vorhandene Ressourcen gelöscht werden. Auch ein Aufruf von DELETE sollte idempotent implementiert werden.

Die bis hier angeführten Methoden dienen zum Lesen und Bearbeiten der Ressourcen. Die im folgenden genannten Metoden haben nur Hilfsfunktionen. Die letzten der beiden sind für diese Arbeit irrelevant, werden aber der Vollständigkeit halber angeführt:

OPTIONS Mit der Methode OPTIONS kann der Client anfragen, welche Methoden auf der Ressource mit dem angegebenen URI erlaubt sind.

TRACE Wenn der Client die Methode TRACE aufruft, dann gibt der Server als Body der Antwort die gestellte Anfrage zurück. Dies dient bspw. zum Testen / Debuggen.

CONNECT Diese Methode ist reserviert für Proxies, die dynamisch zu einem Tunnel umgeschaltet werden können, z. B. für HTTPS.

2.2.2 Statuscodes

Damit auch die Client-Software die Antworten des Servers verstehen kann, gibt der Server seit HTTP/1.0 auch einen maschinenlesbaren Status zurück. Die Statuscodes sind immer dreistellige Zahlen. Es gibt 5 Gruppen von Statuscodes. Die Gruppen und die in dieser Arbeit benötigten Status werden hier angeführt:

- 100-199: reine Information.
- 200-299: Operation erfolgreich.
  - 200: Anfrage erfolgreich verarbeitet.
  - 201: Ressource angelegt.
  - 202: Die Anfrage wird akzeptiert, die Antwort steht erst später zur Verfügung.
  - 204: Anfrage erfolgreich; die Antwort enthält keine Entität.
- 300-399: Weiterleitung auf eine andere Seite oder ähnliches.
- 400-499: Fehler, den vermutlich der Client verursacht hat.
  - 400: Die Anfrage kann aus diversen Gründen nicht bearbeitet werden.
  - 401: Die Anfrage muss authentifiziert werden. Wenn Authentifikationsdaten angegeben wurden, dann waren diese ungültig.
  - 403: Der authentifizierte Benutzer darf nicht auf die Ressource zugreifen.
  - 404: Die angesprochene Ressource existiert nicht.
  - 405: Die angegebe HTTP-Methode ist auf dieser Ressource nicht erlaubt.
  - 406: Zu den angegebenen Accept-Kopfzeilen kann keine Entität erzeugt werden.
  - 415: Das Datenformat der Anfrage-Entität kann nicht verarbeitet werden.
- 500-599: Fehler, dessen Ursache vermutlich auf der Seite des Servers liegt.
  - 500: Der Server hat einen nicht näher bezeichneten internen Fehler festgestellt.

Eine vollständige Liste findet sich im RFC 2616 [20].

2.2.3 Kopfzeilen

Seit HTTP/1.0 werden neben den reinen Nutzdaten auch Metadaten übertragen. Sie heißen Kopfzeilen (englisch: Header). Dabei wird syntaktisch nicht zwischen den Metadaten der Übertragung, denen der Ressource und denen der konkreten Repräsentation unterschieden, was Roy Fielding in seiner Dissertation bemängelt [21].

[20] HTTP/1.1-Spezifikation: [RFC-2616 1999]
[21] Dissertation von Roy Fielding: [Fielding 2000, Abschnitt 6.3.4.4]

Die Kopfzeilen enthalten bspw. Informationen über

- das in dieser Repräsentation verwendete Datenformat,
- in dieser Übertragung verwendete Zeichensätze und Kodierungen,
- Vorbedingungen der Anfrage (die entweder erfüllt sein müssen oder nicht erfüllt sein dürfen, damit die Anfrage ausgeführt wird),
- Authentifikation (Diese wird im nächsten Abschnitt kurz beschrieben.),
- vorhandene Varianten von Repräsentationen dieser Ressource (siehe Abschnitt 2.1.1 auf Seite 10). Der Client kann diesbezügliche Wünsche in der Anfrage in den Kopfzeilen Accept (für das Datenformat), Accept-Language, Accept-Charset und Accept-Encoding angeben, wobei er jeweils verschiedene Prioritäten für die einzelnen Wünsche vorgeben kann,
- Zeitpunkt der Erzeugung der Repräsentation,
- Zeitpunkt des Verfalls der Gültigkeit der Repräsentation,
- das Entity-Tag (ein kurzer String, welches sich bei jeder Änderung der Repräsentation ändern muss; mit ihm lässt sich schnell prüfen, ob die Repräsentation geändert wurde),
- explizite Angaben zur Caching-Fähigkeit der Repräsentation,
- die Namen der Kopfzeilen, anhand denen der Server entschieden hat, diese Repräsentation auszuliefern. Anhand dieser Information kann ein Cache erkennen, ob er für eine Anfrage eine passende Repräsentation zur Verfügung hat oder ob er wegen anderslautender Kopfzeilen-Werte beim Server anfragen muss. Leider gibt es keinen standardisierten Weg, um die erlaubten Werte der Variationen anzugeben. Diese wären für den Endanwender manchmal interessant.

2.2.4 Authentifikation

Um den Zugriff auf Ressourcen beschränken zu können, besteht seit HTTP/1.0 die Möglichkeit, dass der Server vom Benutzer eine Authentifikation anfordern kann, bevor er die gewünschten Informationen ausliefert. Das HTTP verwendet ein Challenge-Response-Verfahren, d. h. der Server schickt auf die HTTP-Anfrage eine Antwort, in der er Vorgaben für eine erneute Anfrage durch den Client macht.

Wenn die Anwendung einen von der Anwendung selber unabhängige Authentifikationsmechanismus verwendet (bspw. einen in das HTTP integrierte Mechanismus), dann kann dieser leicht ausgetauscht werden. Außerdem dürfen URIs keine Authentifikationsdaten als Teil des Pfades o. ä. enthalten (siehe Abschnitt 2.1.1 ab Seite 9).

Folgende Authentifikationsmechanismen sind für das HTTP definiert [22]:

- *Basic Authentication* ist ein ganz einfacher Mechanismus. Benutzername und Passwort werden base64-kodiert (nicht verschlüsselt) an den Server geschickt.
- *Digest Access Authentication* ist etwas komplizierter. Dabei wird das Passwort verschlüsselt übertragen.

Außer diesen beiden Mechanismen dürfen Entwickler auch eigene Authentifizierungsmechanismen implementieren. Ein Beispiel dafür ist der Microsoft NT LAN Manager (NTLM) [23]. Er verwendet die Identität des im Betriebssystem Windows eingeloggten Benutzers. [Pilgrim 2003] beschreibt anhand eines Beispiels, wie ein eigener Authentifikationsmechanismus implementiert werden kann.

2.2.5 Content Negotiation

Schon Tim Berners-Lee hat bei der Entwicklung des HTTP-Protokolls die Möglichkeit der Aushandlung des übertragenen Datenformats in das Protokoll integriert. Dieses Feature heißt Content Negotiation (etwa: Inhalts-Aushandlung). Damit ist es möglich, dass für eine Ressource verschiedene Repräsentationen unter einem gemeinsamen URI zur Verfügung stehen. Ein Web-Server kann dann die Repräsentation einer angefragten Ressource im HTML-Format für eine traditionelle Web-Anwendung zurückgeben, oder alternativ als PDF oder PostScript zum Durcken etc. Grafiken können je nach Wunsch des Clients bspw. als vektororientierte Grafik (z. B. „image/svg+xml“), „image/png“ oder „image/jpeg“ zurückgeben werden. Dazu übergibt der Client in der Anfrage in der Kopfzeile „Accept“ die von ihm verstandenen Datenformate.

Der Client kann dem Server auch Wünsche bzgl. der Sprache der zurückzugebenden Repräsentation geben. Texte können bspw. in verschiedenen Sprachen vorliegen und werden dann nach den Wünschen des Anwenders zurückgegeben. Besonders in mehrsprachigen Ländern wie z. B. der Schweiz ist dies ein sinnvolles Feature. Technisch wird dies umgesetzt, indem der Client dem Server in der Kopfzeile „Accept-Language“ die Sprachkürzel (de, en, fr, ...) der von ihm bevorzugten Sprachen angibt.

Roy Fielding schreibt allerdings in einem Posting in einer Newsgroup, dass er auf Webauftritten immer verschiedene Pfade für verschiedene Sprachen vorsieht. Content-Negotiation verwendet er nur beim Betreten der Seite zur Weiterleitung zur gewünschten Sprache. Seiner Ansicht nach sollte man es mit der reinen Lehre nicht übertreiben [24].

[22] HTTP-Standard-Authentifikationsmechanismen: [RFC-2617 1999]
[23] Infos zum NT LAN Manager finden sich z. B. unter [Microsoft 2007] und [Tschalär 2003].
[24] Nicht übertreiben mit der Content-Negotiation: [Fielding 2006]

Wenn Sprache und gewünschtes Datenformat im Link enthalten sind, kann ein Anwender solch einen Link auch weitergeben, egal wie der Browser bzgl. Sprache und Format eingestellt ist. Bietet die Web-Seite auch Links zu dem gleichen Dokument in anderen Sprachen, dann kann der Anwender bei Bedarf manuel eine andere Sprache wählen. Die flexibelste Variante ist dabei die Möglichkeit, dass der Server sowohl die in HTTP vorgesehene Variante als auch die Angabe von Format oder Sprache im Link anbietet. Dann verwendet er als Standard die in den HTTP-Kopfzeilen angegebenen Werte, welche der Client bei Bedarf durch im URI angegeben Werte überschreiben kann.

2.2.6 Cookies

Cookies dienen dazu, Zustände Client-seitig speichern zu können. Sie enthalten Schlüssel-Wert-Paare. Ein HTTP-Server kann in einer HTTP-Antwort Cookies an den Client übertragen. Dieser speichert sie lokal und sendet sie beim nächsten Aufruf einer Seite derselben Domain wieder zum Server.

2.2.7 Einschränkungen durch die HTML

Das HTTP wird sehr häufig für das World Wide Web verwendet. Dabei wird meistens die Hypertext Markup Language (HTML) verwendet. Sie unterstützt leider nicht alle Features, die für die spezifikationsgemäße Verwendung des HTTPs nötig sind. Einige dieser Probleme werden hier angeführt.

Problem: HTTP-Methoden

In Version 2.0 der HTML [25] wurden Formulare eingeführt, in die Anwender Informationen eingeben können. Sie werden nach dem Absenden zum Server übertragen, welcher sie dann verarbeitet. In der Definition des Formulars wird u. a. die zu verwendende HTTP-Methode angegeben. Dort dürfen auch bei den aktuellen HTML-Versionen nur die in der damals gültigen Verson 1.0 des HTTP gültigen Methoden GET und POST angegeben werden [26].

Deshalb ist es mit der HTML alleine nicht möglich, Ressourcen mit den spezifikationsgemäß zu verwenden HTTP-Methoden zu ändern oder zu löschen. Mit Hilfe von AJAX ist es möglich, beliebige Methoden anzugeben. Allerdings ist JavaScript nicht in allen Browsern in der gewünschten Version implementiert und aktiviert, so dass sich Entwickler nicht auf die Verfügbarkeit von AJAX verlassen können.

[25] HTML 2.0: [HTML 2.0 1996]

[26] Die Methode HEAD macht im Browser keinen Sinn, da bei HEAD-Anfragen kein Antwort-Body übertragen wird, den der Browser anzeigen könnte.

Problem: Kopfzeilen können nicht angegeben werden

Aus Browsern heraus ist es auch nicht möglich, beim Anklicken eines Links vom Client zu übertragende Kopfzeilen vorzugeben, wie die gewünschte Sprache oder das Datenformat für die Content-Negotiation.

Lösung: Tunneln von Methode und Parametern

Eine auch aus Browsern heraus nutzbare Möglichkeit ist, dass Methode oder Kopfzeilen als Parameter im URI übertragen werden. Der Server kann diese dann extrahieren und sich so verhalten, als ob die jeweilige Methode verwendet worden wäre bzw. die Kopfzeile gesetzt wurde.
Der URI ändert sich zwar, was erstmal nicht gut ist. Das Konzept des Modellierens in Ressourcen bleibt davon jedoch unberührt. Nach einer ändernden Operation sollte der Browser mit dem Redirect-After-Post-Pattern[27] zu dem URI weitergeleitet werden, von dem die Ressource gelesen werden kann. Dann taucht der „verunreinigte“ URI auch nicht in der Browser-History auf.
Bei der Angabe von bspw. Datenformat bei einer Anfrage schadet es auch nicht, wenn in der History oder in einem weitergegebenen Link das Datenformat mit auftaucht, denn der Anwender hat sich hier ja für ein spezielles Datenformat entschieden.

Authentifikation

Aus Gründen des (optischen) Designs wünschen Betreiber von Webauftritten häufig, dass Anwender Benutzername und Passwort auf einer Web-Seite eingegeben sollen, die so aussieht wie der Rest des Webauftritts. Dies ist mit der HTML nicht machbar, wenn die HTTP-Authentifikation verwendet werden soll. Mit AJAX lässt sich dies lösen, allerdings ist AJAX nicht in allen Brosern verfügbar, entweder weil der Browser oder die verwendete Version des Browsers AJAX nicht unterstützt oder weil der Anwender JavaScript deaktiviert hat.

[27] Browser speichern die gesendeten Anfragen typischerweise in der History. Beim Rücksprung werden dann auch POST-Anfragen inklusive ihrer Seiteneffekte auf dem Server wiederholt. Um dies zu verhindern, gibt es das Redirect-after-Post-Pattern: Anstatt nur einen Bestätigungscode an den Client zu senden, sendet der Server eine Weiterleitung auf eine andere Seite. Der Browser ruft die Seite mit GET ab und gibt sie aus. Dieser Eintrag ersetzt dann auch die POST-Anfrage in der Browser-History.

2.3 RESTful Web-Services

Die typischen Aufgaben in einer Geschäftsanwendung bestehen meistens aus 4 Operationen: Ein Geschäftsobjekt anlegen, es ausgeben, es ändern oder es zu löschen. Diese 4 Operationen werden auch als CRUD-Operationen bezeichnet: Create, Read, Update und Delete. Sie lassen sich auch auf die 4 HTTP-Methoden abbilden: POST, GET, PUT und DELETE, wobei Ressourcen außer mit POST auch mit PUT angelegt werden können[28]. Deshalb eignet sich das HTTP mit angewandten REST-Architektur-Bedingungen sehr gut dazu, typische Geschäftsanwendungen zu realisieren.

Da HTTP das Web-Protokoll ist, werden die auf dieser Basis angebotenen Dienste entsprechend Web-Dienste oder Web-Services genannt. Wenn sie nach den REST-Prinzipien implementiert werden, werden sie als *RESTful Web-Services* bezeichnet. Dabei ergibt sich ein Wortspiel: „restful“ heißt auf deutsch „ruhig“ oder „erholsam“.

Roy Fielding hat in seiner Dissertation Web-Dienste nicht angesprochen. Die Entwickler haben die REST-Prinzipien einfach so eingesetzt, obwohl es keine akademische Grundlage explizit für die Übertragung der Idee auf die Verwendung *als Web-Dienst-Schnittstelle* gibt[29]. Roy Fielding hat später in einem Post in einer REST-Newsgroup als Antwort auf eine Frage geäußert, dass es die Hauptaufgabe des REST sei, den Web-Dienst zu modellieren[30]. Dies zeigt, dass er vermutlich nicht gegen die Verwendung des REST-Architekturstils für Web-Dienste ist.

REST-Web-Dienste wurden z. B. im Jahr 2002 in [Prescod 2002d] beschrieben. Ein Jahr später hat Rohit Khare seine Dissertation über REST für verteilte Systeme[31] veröffentlicht. Sie bezieht sich auf die Verwendung von REST für Dienste, wenn auch in einem etwas anderen Kontext.

Auch wenn die Verwendung explizit für Web-Dienste nicht theoretisch behandelt wurde, so haben sich doch einige Punkte in der Praxis bewährt:

- Für Geschäftsobjekte sollte der URI technikunabhängig[32] sein. Schließlich soll das Geschäftsobjekt adressiert werden, nicht das zu einem bestimmten Zeitpunkt gerade verwendete PHP-Skript o. ä., welches die Ausgabe erzeugt. Dies ist keine Forderung von Roy Fielding, aber sie wird im Internet immer wieder genannt, und diesem Vorschlag schließt sich der Autor dieser Arbeit an, denn sie betont die Geschäftsobjekte

[28] Auch die SQL besteht im Kern aus den 4 CRUD-Operationen: INSERT, SELECT, UPDATE und DELETE.
[29] REST als Web-Dienst-Schnittstelle: z. B. [Prescod 2002b]
[30] Details finden sich im Posting [Fielding 2003b].
[31] „Extending the REpresentational State Transfer Architectural Style for Decentralized Systems“: [Khare 2003]
[32] Das heißt z. B.: keine Dateieindung „.php“ o. ä.

(die Ressourcen) anstatt der Implementierung. Wenn der URI technik-unabhängig aufgebaut ist, kann ohne eine Veränderung der Schnittstelle leicht auf eine andere Implementierung, Programmiersprache o. ä. umgestellt werden. Die Techniknabhängigkeit lässt sich z. B. mit URL-Rewriting umsetzen.

- Ein wichtiger Punkt von Roy Fielding ist die Statuslosigkeit. Ein Hauptgrund für die Verwendung von Sessions in Web-Anwendungen ist die Steuerung und Kontrolle, welcher Benutzer eingeloggt ist. Wenn HTTP-Authentifikation verwendet wird, dann ist für viele Fälle die Verwendung von Sessions nicht mehr nötig. Die im Abschnitt 2.2.7 ab Seite 27 genannten Einschränkungen bzgl. des Web-Seiten-Designs sind bei Web-Diensten hinfällig, so dass die HTTP-Authentifikation bei Web-Diensten problemlos verwendet werden kann.
- Roy Fielding fordert in seiner Dissertation, dass in den vom Server versendeten Repräsentation URIs zu den mit der Ressource verbundenen Ressourcen vorhanden sein sollen. Clients sollen ausschließlich diese URIs verwenden. Ansonsten ist die Anwendung nicht im REST-Architekturstil implementiert. Ein typisches Beispiel ist eine Ressource, welche eine Liste von Verweisen zu den in ihr enthaltenen Ressourcen mit Kurzbeschreibungen enthält. Ruft man eine der enthaltenen URIs ab, so erhält man weitere Details dieser Ressource. Der Eintrittspunkt in eine REST-Anwendung ist immer ein zentraler URI. Dies ist der einzige URI, den der Client benötigt, weil er alle weiteren URIs vom Server übertragen bekommt. Die Links zu weiteren Ressourcen werden in der Praxis häufig nicht mitgeliefert.

Die Repräsentationen werden – wie es der Sinn bei Web-Diensten ist – in maschinenlesbaren Formaten übertragen (z. B. XML). Besonders im JavaScript-Umfeld ist auch die JSON (JavaScript Object Notation) beliebt. Mit Hilfe des Konzeptes der Content Negotiation von HTTP ist es auch sehr leicht möglich, unter demselben URI verschiedene Formate anzubieten.

In den folgenden Abschnitten wird ein Protokoll im REST-Architekturstil vorgestellt und dann einige im REST-Architekturstil implementierte Web-Dienste. Im Anhang A.1 ab Seite 111 findet sich ein Beispiel für RESTful-Web-Service-Interaktionen.

2.3.1 Beispiele für RESTful Protokolle

Ein Beispiel für ein populäres, im REST-Architekturstil implementiertes Protokoll ist das Atom Publishing Protokoll [33]. Es wurde für die Verbreitung und Bearbeitung von News-Feeds entwickelt und ist damit ein Nachfolger des RSS-Protokolls.

[33] AtomPub: siehe [RFC-5023 2007]

2.3.2 Beispiele für RESTful Web-Services

Inzwischen bieten viele Dienstleister ihre Dienste (auch) über eine REST-konforme Schnittstelle an. Hier werden jetzt einige wenige Beispiele aufgeführt:

- *Blinksale*[34] ist eine Anwendung, um Rechnungen online zu verschicken und zu verwalten. Blinksale bietet auch eine RESTful Web-Service API an.
- *Amazon S3*[35] stellt Datenspeicher für beliebige Daten zur Verfügung, auf welche über eine REST-konforme Schnittstelle zugegriffen werden kann.
- *stikkit*[36] ist eine Webseite, auf der man eigene Notizen speichern kann. Die Notizen können auch über einen RESTful Web-Service abgerufen oder geändert werden.

Eine Liste weiterer Dienste findet sich z. B. in [Richardson und Ruby 2007a, Anhang A].

Es gibt auch fertige Anwendungen, um RESTful auf Datenbank-Tabellen zuzugreifen:

- *phprestsql*[37] ist ein REST-Frontend für SQL-Datenbanken, welches exakt in Ressourcen modelliert ist.
- *sqlREST*[38] bietet eine ähnliche Funktionalität auf Servlet-Basis.

2.3.3 Häufige Fehler

Es gibt viele – auch große – Anbieter, die direkt über HTTP nutzbare Schnittstellen zur Verfügung stellen. Etliche der Anbieter versuchen, auf den REST-Zug aufzuspringen und geben ihren Diensten an REST erinnernde Namen, auch wenn die Dienste nicht immer RESTful implementiert sind.

Häufig wird dabei der Fehler gemacht, dass nicht Ressourcen modelliert werden, sondern Dienste zur Verfügung gestellt werden. Dies ist vor allem daran zu erkennen, dass die URIs Verben (get, add, delete, ...) als zentrales Element enthalten anstatt Nomen. Die zu verwendenen Verben sind als Methoden in das HTTP integriert. Häufig werden auch die HTTP-Methoden falsch verwendet.

Leider finden sich einige dieser Fehler sogar in Anleitungen und Beispielen zu RESTful WebServices im Internet. Weitere typisch Fehler werden bspw. in [Prescod 2002a] beschrieben.

---

[34] www.blinksale.com
[35] Amazon S3: [Amazon 2007], [Richardson und Ruby 2007a]
[36] www.stikkit.com/api
[37] phprestsql: [Bayer 2007], [PhpRestSql 2005]. Online-Beispiel: http://phprestsql.sourceforge.net/demo/
[38] http://sqlrest.sourceforge.net/

2.4 Java-Frameworks und -APIs für RESTful Web-Services

In diesem Abschnitt werden Frameworks, Spezifikationen u. ä. kurz vorgestellt, die gut geeignet sind, um mit Java RESTful Web-Services zu implementieren.

2.4.1 Restlet

In diesem Abschnitt wird das Restlet-Framework[39] vorgestellt, wobei besonders die Dinge berücksichtigt werden, die für die Umsetzung der JAX-RS-Spezifikation (JSR 311, siehe Kapitel 3 ab Seite 43) von Bedeutung sind.

Restlet ist ein leichtgewichtiges REST-Framework für Java. Es unterstützt sowohl die Server-seitige als auch die Client-seitige Entwicklung. Restlet wurde von Jérôme Louvel entwickelt, als er kein für ihn passendes REST-Framework fand. Restlet ist vollständig an den REST-Architektur-Bedingungen orientiert und stellt deshalb für die REST-Konzepte Ressource, Repräsentation, Anfrage, Antwort etc. entsprechende Klassen zur Verfügung. Außerdem stellt das Framework einen Lebenszyklus für die Server-seitigen Artifakte zur Verfügung.

Anfangs war die Restlet-API auf die Java-Servlet-API aufgesetzt. Es zeigte sich jedoch, dass die Abhängigkeit von der Servlet-API nicht schön war. Deshalb wurde die Restlet-API komplett von der Servlet-API unabhängig; seit dem funktionierte das Konzept nach Aussage von Jérôme Louvel gut. Inzwischen können gegen die Restlet-API entwickelte Anwendungen ganz unabhängig von der Servlet-API auf verschiedenen Java-HTTP-Servern laufen. Es ist aber weiterhin möglich, Restlet-Anwendungen als Teil einer Servlet-Anwendung laufen zu lassen.

Anwendungsentwickler implementieren gegen die Restlet-API. Um Netzwerkzugriffe und ähnliches brauchen sie sich nicht kümmern; sie werden von der Restlet-Engine behandelt. Die Engine steuert auch die Deserialisierung und die Serialisierung[40] der Anfragen und Antworten. Als Referenzimplementierung steht die Noelios Restlet Engine zur Verfügung. Die Restlet-Engine hat ein Service Provider Interface (SPI) und kann deshalb auch von anderen Entwicklern implementiert werden.

Die Restlet-API wird durch Erweiterungen ergänzt, um bspw. Objekte mit JAXB, JIBX oder JSON zu (de-)serialisieren, Atom-Feeds zu erstellen oder Repräsentationen mit einer der Template-Engines Velocity und Freemarker zu erstellen (z. B. HTML-Repräsentationen).

Um einen Eindruck der programmiertechnischen Umsetzung zu bekommen, finden sich in den Anhängen A.2 bis A.4 ab Seite 114 einige Code-Beispiele.

[39] Restlet ist ein eingetragenes Markenzeichen der Firma Noelios Technologies. Webseite: [Noelios 2007b]

[40] Da das Restlet-Framework mit dem Ziel vorgestellt wird, es für eine Server-API zu nutzen, werden Anfragen erst deserialisiert und nach der Bearbeitung die Antworten serialisiert.

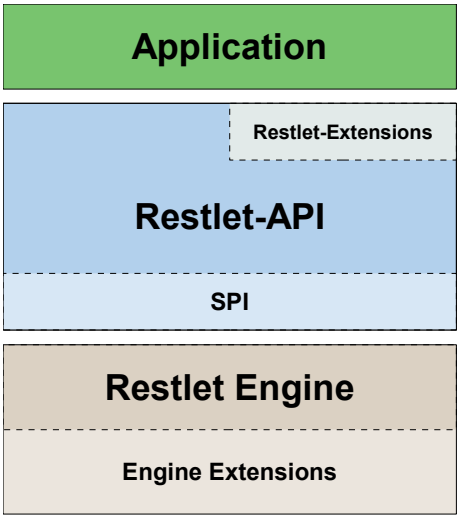

Abbildung 2.6: Restlet-Architektur (aus [Noelios 2008c] © Noelios Technologies)

Restlet-Hierarchie

Um die einheitliche Schnittstelle der REST-Architektur darzustellen, existiert die abstrakte Klasse `Uniform`. Sie enthält die 6 HTTP-Methoden POST, GET, HEAD, PUT, DELETE und OPTIONS[41]. Diese werden an die im nächsten Absatz beschriebene Methode handle(..) weitergeleitet.

Von der Klasse `Uniform` ist die Klasse `Restlet` abgeleitet. Sie entspricht etwa dem Interface `Servlet` aus der Servlet-Spezifikation von Sun. Die Klasse `Restlet` ist der Dreh- und Angelpunkt des Restlet-Frameworks. Als zentrale Methode enthält sie die Methode `handle(Request, Response)`, welche Anfragen bearbeitet. Diese Klasse enthält auch die Methoden für den Lebenszyklus der Restlet-Instanzen.

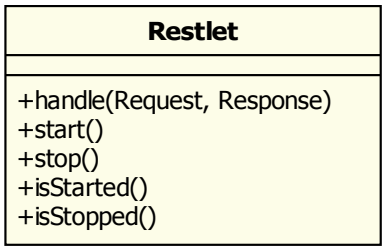

Abbildung 2.7: Die wichtigsten Methoden der Klasse Restlet (eigene Grafik)

[41] Die Methode TRACE ist in der Restlet-Schnittstelle nicht enthalten, da sie zum Debuggen des Connectors konzipiert ist.

Die meisten Klassen, die Anfragen behandeln können, sind von der Klasse `Restlet` abgeleitet. Die folgende Liste beschreibt kurz einige dieser Klassen:

- `Finder` behandeln Anfragen an konkrete Ressourcen. Für die Implementierung des JSR 311 sind sie irrelevant und werden deshalb nicht weiter behandelt.
- Ein `Router` leitet Anfragen abhängig vom URI der Anfrage an andere Restlets weiter.
- `Filter` ermöglichen es, Anfragen oder Antworten vor oder nach der Verarbeitung zu verändern (z. B. die Kodierung). Eine Unterklasse ist `Guard`. Ein Guard überprüft die Authentifizierungsdaten. Bei erfolgreicher Authentifizierung leiten er die Anfrage an ein vorgegebenes Restlet weiter. Andernfalls verweigert er die Anfrage und schickt dem Client die zur Authentifizierung benötigte Informationen.
- Die Klasse `Application` dient zum Sammeln und Initialisieren weiterer Restlets, die zu einer Anwendung gehören. Weiteres dazu folgt auf Seite 36.
- Ein `VirtualHost` ist ein spezieller Router, der es ermöglicht, unter einer IP-Adresse mehrere Domains zu hosten. Jede Restlet-Anwendung ist einem VirtualHost zugeordnet, wobei dies auch der Standard-Host sein kann, der alle Anfragen akzeptiert.
- Ein `Connector` serialisiert oder deserialisiert Anfragen und Antworten und verbindet so die Anwendung über das Netzwerk mit anderen REST-Systemen, entsprechend der Definition von Roy Fielding (siehe Abschnitt 2.1.2). Für weitere Details siehe Seite 36.
- Eine `Component` verwaltet die in einem Server laufenden Anwendungen, welche bei Bedarf verschiedenen virtuellen Hosts zugeordnet sein können.

Die Unterklassen von Restlet sind sehr flexibel verschachtelbar. Ein Restlet reicht die Anfragen dann an eines der enthaltenen Restlets weiter (vgl. Abbildung 2.9 auf Seite 37).

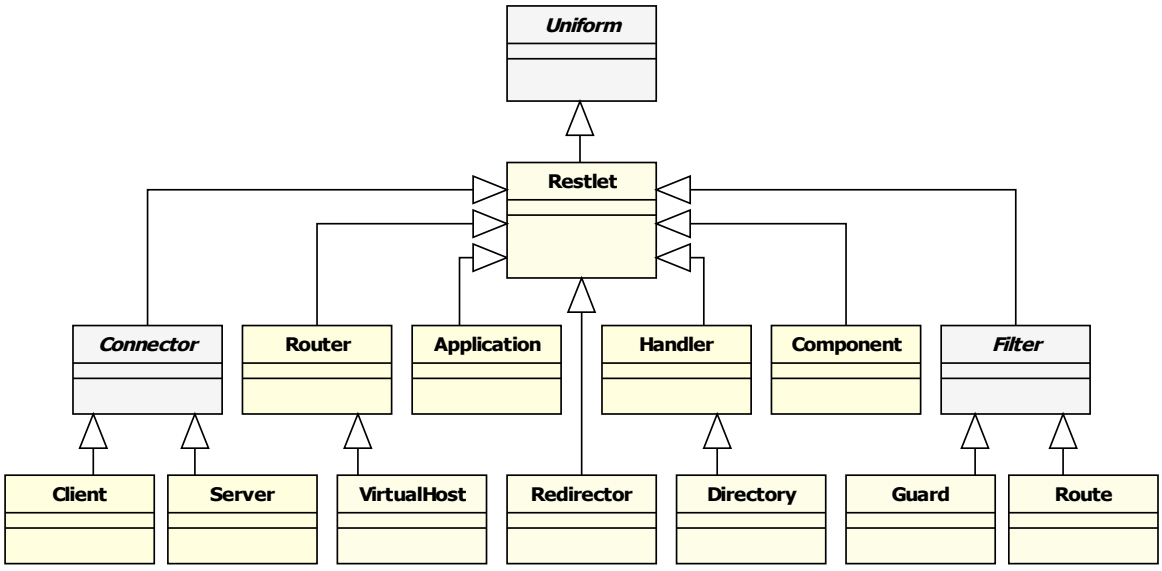

Abbildung 2.8: Restlet: Klassenhierarchie
(aus [Noelios 2008c] © Noelios Technologies, Layout modifiziert)

Request und Response

Die Klassen `Request` und `Response` stellen Methoden zur Verfügung, um auf hohem Abstraktionsniveau auf die in den HTTP-Kopfzeilen übertragenen oder zu übertragenden Werte zugreifen zu können. Auch um die Kodierung oder Dekodierung der Kopfzeilen braucht sich die Anwendung nicht zu kümmern, denn die Restlet-Engine stellt sie dekodiert zur Verfügung und übernimmt beim Senden auch die Kodierung.

Im Restlet-Framework stehen für in Collections gespeicherte Werte in den meisten Klassen Getter und Setter für den Zugriff auf die Collection zur Verfügung. Wenn für jedes Collection-Attribut eine add-Methode, eine remove-Methode etc. vorhanden sein würde, dann würde die Anzahl der Methoden in den Klassen sehr wachsen und die Schnittstelle der Klasse unübersichtlich werden [42]. Die Klassen Request und Response sind ein gutes Beispiel dafür. Um zu einer Collection Elemente hinzuzufügen, greift der Anwendungsentwickler mit dem Getter auf die jeweilige Collection zu und fügt das hinzuzufügende Element mit add() ein.

Ressourcen

Einzelne konkrete Ressourcen werden als Unterklasse von `Resource` implementiert. Aus konzeptioneller Sicht müsste diese Klasse auch von Restlet erben, denn sie kann Anfragen bearbeiten. Aus pragmatischen Gründen erbt diese Klasse jedoch nicht von `Restlet`, sondern von `Handler`. Ein Handler – und damit auch eine Resource – braucht nicht Theradsicher zu sein, im Gegensatz zu den Unterklassen von `Restlet`. Eine `Resource` wird mit einem `Finder` (siehe oben) als Restlet eingebunden.

Die Klasse `Resource` hat ein höheres Abstraktionsniveau als die Restlet-Klassen. Für GET-Anfragen wählt sie aus den von der Ressource unterstützten Varianten automatisch die richtige aus, abhängig von den vom Client gewünschten Datenformaten, Sprachen etc. Dafür muss der Anwendungsentwickler im Konstruktor der Ressource angeben, welche Varianten diese Ressource unterstützt. Die Methode zur Erzeugung der Repräsentation bekommt die Variante übergeben, die am besten zu den Wünschen des Clients passt. Ein Beispiel findet sich im Anhang A.3 auf Seite 117.

Repräsentationen

Repräsentationen werden mit eigenen Klassen verarbeitet. Für häufig verwendete Datenformate stehen spezielle Klassen zur Verfügung, z. B. für die XML- und JSON-Verarbeitung. Für die Erzeugung von HTML-Seiten o. ä. gibt es spezielle Klassen, um die Template-Engines Freemarker oder Velocity [43] einfach einzubinden.

[42] [Louvel 2008d]
[43] http://freemarker.org/, http://velocity.apache.org/

Die Restlet-Engine steuert die Deserialisierung und die Serialisierung der Repräsentationen. Die Repräsentation einer empfangenen Nachricht wird den verarbeitenden Restlets in einer Unterklasse der Klasse `Representation` zur Verfügung gestellt. In dieses Objekt kopiert die Restlet-Engine erst die zur Repräsentation gehörigen Metadaten, und stellt der Repräsentation dann den InputStream oder den ReadableByteChannel (NIO) mit dem Nachrichtenkörper der Anfrage zur Verfügung, je nach dem, was der Server-Connector zur Verfügung stellt.
Für die Verarbeitung wird dieses Repräsentations-Objekt in eine andere Unterklasse von `Representation` verpackt, welche sie weiterverarbeitet, z. B. einer `JsonRepresentation`, von welcher dann das gewünschte (JSON-)Objekt abfragbar ist. Die Daten der HTTP-Anfrage können von der verarbeitenden Klasse als Reader, InputStream oder ReadableByteChannel abgerufen und ausgelesen werden. Die Representation konvertiert sie bei Bedarf automatisch.

Beim Senden einer Nachricht sendet die Restlet-Engine als erstes die Anfrage- bzw. die Statuszeile, gefolgt von den Kopfzeilen. Danach ruft die Engine die Methode write() der zurückgegebenen `Representation` auf, welche den Nachrichtenkörper serialisiert. Die konkrete Klasse delegiert die Serialisierung dann an den jeweiligen Serialisierungsmechanismus – z. B. an einen JAXB-Marschaller oder eine Template-Engine – weiter.

Tunnel-Service

Wie im Abschnitt 2.2.7 auf Seite 26 beschrieben, erlauben Browser und die HTML nur die Methoden GET und POST. Auch die Wünsche für die Content-Negotiation können in Browsern meistens nur relativ umständlich oder gar nicht für eine einzelne HTTP-Anfrage angegeben werden.

Abhilfe schafft der Tunnel-Service. Er erlaubt es für ändernde Anfragen (z. B. PUT und DELETE) die HTTP-Methode als Query-Parameter einer POST-Anfrage zu übergeben. Wenn ein entsprechender Query-Parameter in der URI der Anfrage gesetzt wurde, dann ändert der diese Funktionalität umsetzende `TunnelFilter` die Anfrage so ab, dass die zu verwendende HTTP-Methode an die Anwendung weitergegeben wird; der ausgewertete Query-Parameter wird aus der Anfrage entfernt. Restlet unterstützt ebenso das Tunneln virtueller Dateiendungen für die Content-Negotiation mit Hilfe von Query-Parametern. Der TunnelFilter setzt die angeführten Query-Parameter transparent für die Restlet-Anwendung auf die jeweiligen Accept-Kopfzeilen um, so dass es für die Anwendung so aussieht, als ob die Accept-Kopfzeilen verwendet worden wäre.

Application

Eine `Application` sammelt diverse Restlets und Ressourcen, die zu einer logischen Anwendung zusammengehören und verschachtelt sie beim Starten der Anwendung wie vom Anwendungsentwickler vorgesehen. Hierzu dient die Methode `createRoot()`. Die `Application` stellt auch einige Services zur Verfügung, z. B. das dekodieren oder kodieren von Anfragen, wie es in der HTTP-Spezifikton vorgesehen ist. Auch die Einstellungen für den Tunnel-Servcie (siehe oben) werden hier gesetzt.

Connectoren

Wie im Abschnitt 2.1.2 ab Seite 12 vorgestellt, sind Connectoren die Verbindungen zu anderen REST-Systemen. Dieses Konzept ist auch in der Restlet-Architektur umgesetzt. Die Restlet-Engine verwendet sowohl auf der Client- als auch auf der Server-Seite verschiedene Connectoren. In der Restlet-Engine ist je ein Connector enthalten. Es können aber auch vorhandene Client- oder Server-Bibliotheken verwendet werden, z. B. Servlet-Container, der Grizzly-Server von Sun oder der Apache-HTTP-Client. Zu diesen Bibliotheken existieren jeweils Restlet-Adapter-Klassen, die den einheitlichen Zugriff von der Engine aus ermöglichen. Die Conectoren stehen als eigene jar-Files zur Verfügung und werden je nach Bedarf in den Java-Klassenpfad eingebunden.

Die Servlet-API unterstützt selber auch Authentifizierung und eine Rollenverwaltung. Die hier vorhandenen Möglichkeiten können aus der Restlet-API fast nicht genutzt werden. Bei entsprechender Konfiguration des Servlet-Containers wird der anfragende Benutzer authentifiziert oder die Anfrage zurückgewiesen. Der Benutzername steht in der Restlet-API bisher nicht zur Verfügung, und auch auf die Rollenverwaltung kann nicht zugegriffen werden.

Component

Eine Component ist im REST-Architekturstil eine abstrakte Software-Einheit von Befehlen und internen Zuständen, die die Bearbeitung von Daten über ihre Schnittstelle erlaubt. In der Restlet-Architektur ist eine Component ein Container, der eine oder mehrere Anwendungen und einen oder mehrere virtuelle Hosts als Server verwaltet.

Exception-Behandlung

In Restlet ist kein Mechanismus zur gesonderten Behandlung von Exceptions implementiert, wie z. B. in der Servlet-Spezifikation, wo in den Konfigurationsdateien für spezielle Exceptions spezielle Fehlerseiten zurückgegeben werden können. Checked Exceptions sind in

Restlet-Unterklassen und Resourcen in Version 1.0 nicht erlaubt. Eine spezielle Exception, um Fehler an die Restlet-API zurückzugeben wird in der Version 1.1 implementiert. Sie enthält außer der üblichen Nachricht und einer eventuellen Ursache auch einen HTTP-Status, welcher dann an den Client zurückgegeben wird. Auftretende RuntimeExceptions werden auf den HTTP-Status 500 (Interner Server Fehler) gemappt, wenn sie an die Restlet-API zurückgegeben werden.

Bearbeitung einer Anfrage

Die folgende Abbildung zeigt (vereinfacht), wie eine Anfrage vom Server-Connector durch den Virtual Host und einige Filter zu den Resource-Implementierungen weitergereicht wird.

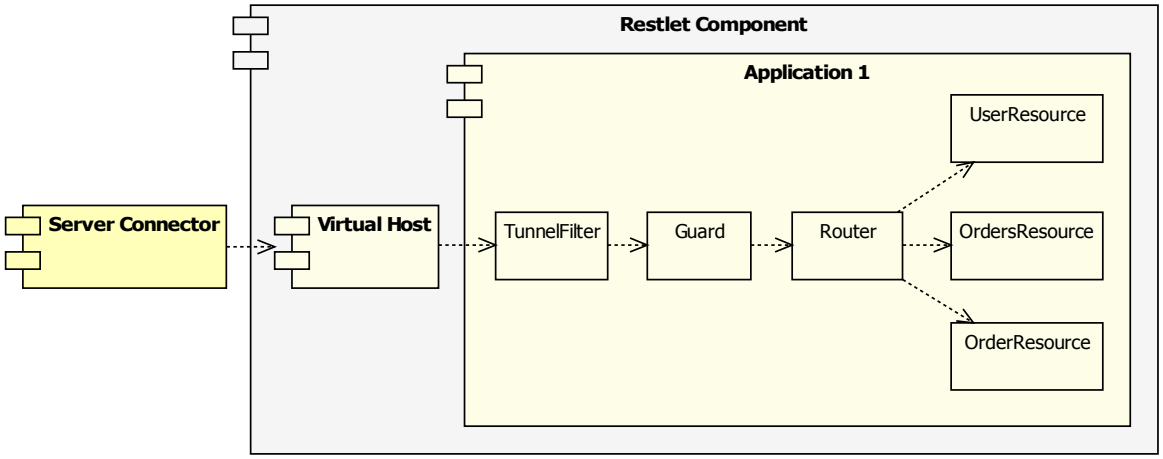

Abbildung 2.9: Restlet: Bearbeitung einer Anfrage (angelehnt an [Noelios 2008c])

2.4.2 JAX-RS / JSR 311

Die Firma Sun spezifiziert z. Zt. im Rahmen des Java Community Process die JAX-RS-Spezifikation. Sun möchte damit eine High-Level-API für RESTful Web-Services mit Java entwickeln. Mit Hilfe der in Java 5.0 eingeführten Annotationen werden Ressourcen von normalen Java-Klassen (POJOs[44]) repräsentiert. Die HTTP-Aufrufe werden mit Hilfe von weiteren Annotationen und den im HTTP-Request vom Client empfangenen Wünschen auf verschiedene Methoden dieser Klassen gemappt.

Diese Spezifikation steht im Zentrum dieser Arbeit. Sie wird im Abschnitt 3.2 ab Seite 44 ausführlich beschrieben.

[44] POJO = Plain Old Java Object; siehe Glossar

2.4.3 Weitere Frameworks

In diesem Abschnitt wird eine Auswahl weiterer Frameworks und APIs aus dem Java-Umfeld aufgeführt, mit denen ebenfalls RESTful Web-Services implementiert werden können. Es wird jedoch nicht weiter auf diese Frameworks eingegangen.

- *Apache CXF* [45] (früher XFire) ist ein Open Source Service Framework für die Entwicklung von Web Services mit SOAP. Inzwischen werden auch RESTful Web-Services unterstützt. Dazu stehen drei Möglichkeiten zur Verfügung:
  - Zum ersten steht JAX-WS zur Verfügung, womit aber nur XML-Nachrichten verschickt werden können.
  - Das HTTP-Binding erlaubt es, Services über Namenskonventionen oder mit Annotationen als RESTful WebService zu nutzen. Auch hiermit können nur XML-Nachrichten empfangen und versandt werden.
  - Im CXF-Projekt wird z. Zt. auch die JAX-RS-Spezifikation implementiert [46]. Hierzu siehe Abschnitt 3.9 auf Seite 64.

  CXF unterstützt – ähnlich wie Restlet – diverse „Transports“ (Restlet-Analogie: Connectoren), von deren Details abstrahiert wird.
- *RESTEasy* [47] ist ein Projekt der Firma JBoss, mit dem einfache Web-Diensten im REST-Architekturstil ermöglichen werden sollen. RESTEasy unterstützt auch Spring, EJB3 Session Beans und Message Driven Beans. Es basiert auf der Servlet-API, so dass immer ein Servlet-Container nötig ist. Seit dem 12. 6. 2008 steht die fünfte Beta-Version von RESTEasy zur Verfügung. Im Rahmen dieses Projektes wird auch die JAX-RS-Spezifikation implementiert.
- *Servlet-basiert:* Auch mit Java-Servlets können RESTful Web-Services implementiert werden. Die Behandlung der verschiedenen HTTP-Methoden ist sehr einfach, da für jede HTTP-Methode eine entsprechende Methode in der Klasse HttpServlet zur Verfügung steht (doGet(..), doPost(..), doPut(..) etc.). Unterstützung für weitere Anforderungen wie Caching, Content-Negotiation etc. ist nicht vorhanden. Außerdem stellt die Servlet-API auch Funktionalität bereit, die für REST-Anwendungen nicht erlaubt ist, z. B. eine Sessions-Verwaltung.
  Für die Erzeugung von HTML-Seiten für Web-Anwendungen stehen JSPs zur Verfügung. Hierfür gibt es inzwischen viele Bibliotheken, so dass auch aufwendige Web-Seiten relativ einfach implementiert werden können.

[45] [CXF 2008]
[46] siehe [Apache 2007a] [Apache 2007b]
[47] RESTEasy bei JBoss: [Burke 2008a], zu JAX-RS: [Burke 2008f]; bei Sourceforge.net: [Burke 2007b].

- *gomba*[48] stellt einige Java-Servlets zur Verfügung, um die Entwicklung von REST-Anwendungen zu vereinfachen. gomba unterstützt auch JDBC-Transaktionen über mehrere Aufrufe hinweg. Damit wird das REST-Kriterium der Zustandslosigkeit verletzt, so dass eine solche Anwendung nicht RESTful ist.

Apache-CXF hat den Nachteil, dass dort (außer mit der noch nicht fertigen JAX-RS-Implementierung) nur XML-Nachrichten versandt werden können. Mit den rein Servlet-basierten Ansätzen lassen sich zwar alle Datenformate verwenden, allerdings ist keine gute Unterstützung für URI-Verarbeitung etc. vorhanden.

2.5 Alternativen zu Restful Web-Services

Dieser Abschnitt vergleicht kurz RESTful Web-Services mit einigen „Konkurrenten“ und zeigt ihre jeweiligen Vor- und Nachteile auf. Betrachtet werden nur Konzepte, mit denen sich mit Web-Technologien (z. B. HTTP) dienst-orientierte Architekturen implementieren lassen. Binäre Protokolle wie CORBA, DCOM, RMI u. ä. werden deshalb nicht betrachtet.

Sowohl mit RPC als auch mit SOAP lassen sich prinzipiell alle Anwendungsfälle für Web-Dienste implementieren. Ohne zusätzliche Bibliotheken lassen sich damit aber keine direkt aus Browsern heraus nutzbare Web-Anwendungen damit entwicklen.

2.5.1 Remote Procedure Call

Beim „Remote Procedure Call“ (RPC) werden Prozeduren auf entfernten Rechnern aufgerufen. Dabei stehen die Prozeduren, also die Funktionalitäten (entsprechend den Verben) im Zentrum. Den Prozedur-Aufrufen werden die benötigten Parameter mitgegeben. Ein Ziel des Remote Procedure Call Konzepts ist, das Netzwerk so weit wie möglich zu verbergen.

Beim XML-RPC[49] werden Methodenaufruf, die dazugehörigen Parameter und der Rückgabewert in XML kodiert übertragen. Damit wird Plattformunabhängigkeit erreicht. Der Aufruf erfolgt grundsätzlich mit der HTTP-Methode POST. Der Client muss wissen, welche Parameter er in welcher Reihenfolge und welchem Format angeben muss.

Da die Paramater in Typ und Reihenfolge genau in dem vom Server verlangten Format vorgegeben werden müssen, sind Client und Server relativ eng gekoppelt.

[48] [Gomba-Team 2007]
[49] [XML-PRC 2003]

Vergleich mit REST und RESTful Web-Services

Bei RESTful WebServices stehen die Ressourcen (Subjekte, angesprochen mit Nomen) im Zentrum, anstatt der Funktionalitäten. Da diese beiden Konzepte verschiedene Dinge in den Mittelpunkt stellen, lässt sich ein Remote Procedure Call nicht direkt mit dem REST-Architekturstil realisieren. Allerdings lassen sich alle für das RPC-Konzept modellierten Anwendungsfälle auch im REST-Architekturstil modellieren.

Die mit RESTful Web-Services übertragenen Daten enthalten mehr Semantik. Im Konzept sind Dinge schon fest eingebaut wie z. B. die Auswahl des Datenformats und andere Metadaten. Der REST-Architekturtil versucht nicht, das Netzwerk einfach zu verstecken, sondern nutzt Optimierungsmöglichkeiten aus (z. B. durch Caching).

Für Anwendungsfälle, in denen wirklich die Funktionalität im Vordergrund steht, eignet sich das RPC-Konzept besser als RESTful Web-Services. Wenn die Funktionalität allerdings aus dem Bearbeiten von auf dem Server gespeicherten Subjekten besteht, dann sind RESTful Web-Services gut geeignet.

2.5.2 SOAP

SOAP ist ein Protokoll einerseits zum Aufruf von entfernten Prozeduren (RPC-style), andererseits zum Übertragen von Dokumenten (document style).

SOAP verwendet Protokolle wie HTTP, SMTP, JMS, aber auch direkt TCP als reine Transport-Protokolle, ohne die speziellen Fähigkeiten der einzelnen Protokolle auszunutzen. Dies erhöht die Flexibilität in der Verwendung einzelner dieser Protokolle, aber auch den Overhead. Im Internet wird meistens nur das HTTP verwendet, so dass die mögliche Flexibilität durch die Protokollunabhängigkeit im Internet selten genutzt wird. Innerhalb von Organisationen werden auch andere Protokolle verwendet.

Für SOAP-Web-Services gibt es die Beschreibungssprache WSDL (Web Service Description Language). Mit entsprechenden Tools ist es sehr einfach möglich, Stub-Klassen zu generieren, um den Zugriff mit wenig Programmierarbeit zu ermöglichen. Dies gilt besonders für den RPC-Stil. Beim RPC-Stil sind Client und Server dann – wie immer beim RPC – relativ eng gekoppelt. Beim Dokumenten-Stil ist die Kopplung nicht so stark; dafür muss die empfangende Software flexibler bei der Interpretation des Dokumentes sein als beim RPC-Stil.

Das SOAP verwendete bis zur Version 1.1 immer die HTTP-Methode POST. Seit SOAP 1.2 darf auch GET verwendet werden, wenn die Operation nur lesend ist. Die URIs beschreiben einen Endpunkt, unter dem viele Operationen und Geschäftsobjekte angesprochene werden können. Welche Operation auf welchem Geschäftsobjekt aufgerufen wird, kann der HTTP-Anfragezeile nicht entnommen werden.

Zum Auffinden von SOAP-Web-Services gibt es das Verzeichnisdienst-Protokoll UDDI [50].

[50] UDDI = Universal Description, Discovery and Integration; siehe z. B. [UDDI 2008]

Vergleich mit REST und RESTful Web-Services

Ein ganz wesentlicher, häufig vergessener Unterschied ist, dass SOAP ein Protokoll ist, der REST hingegen ein Architekturstil. RESTful Web-Services sind eine konkrete Implementierung dieses Stils auf der Basis des Protokolls HTTP. Sowohl mit SOAP als auch mit RESTful Web-Services lassen sich service-orientierte Architekturen implementieren. Der RESTful Web-Services sind am ehesten mit dem Document-Style von SOAP vergleichbar. Bei SOAP-Web-Services müssen die Daten im XML-Format übertragen werden. RESTful Web-Services sind in diesem Punkt flexibler, denn das Datenformat kann vom Anwendungsentwickler festgelegt werden.

RESTful WebServices verwenden das HTTP als Applikationsprotokoll, so wie das HTTP auch konzipiert ist. Vorteil bei der direkten Verwendung des HTTP gegenüber anderen Protokollen ist, dass das HTTP von Browsern per se nativ unterstützt wird. So kann eine im REST-Architekturstil mit dem HTTP implementierte Anwendung direkt aus einem Browser (ggf. mit AJAX) genutzt werden. Wenn der Dienst die JSON verwendet, dann braucht für den Client kein spezieller Parser eingebunden zu werden, denn die populären Browser, die AJAX unterstützen, implementieren auch die JavaScript-Funktion `eval()`, die in der JSON-Notation notierte Objekte direkt in ein JavaScript-Objekt umwandeln kann.

SOAP-Clients müssen außer dem URI des Endpunktes auch den Name der gewünschten Operation kennen. Die möglichen Operationen können sie über die WSDL-Datei abfragen, aber daraus kann die Client-Software nicht entnehmen, welche der dort aufgeführten Operationen welche Semantik hat (lesen, neu anlegen, löschen etc.). RESTful Web-Services haben den Vorteil, dass auf alle Ressourcen über die sehr generische Schnittstelle zugegriffen werden kann. Der Client braucht nur der URI der Anwendung kennen. Die URIs von weiteren Ressourcen bekommt er vom Server übertragen und kann über die generische Schnittstelle auf die Ressourcen zugreifen.

Bei der Verwendung von RESTful Web-Services können vorhandene HTTP-Infrastrukturen weiter verwendet werden: Differenziertes Sperren von Operationen ist mit üblichen HTTP-Firewalls möglich und vorhandene HTTP-Logging-Mechanismen können einfach weiter verwendet werden. Bei Verwendung des SOAP muss zum Feststellen der Operation und des betroffenen Geschäftsobjektes die Nachricht teilweise interpretiert werden. Dies ist mit reinen HTTP-Tools nicht möglich. Auch der Antwort-Status des HTTP-Protokolls wird vom SOAP nicht wesentlich genutzt; auch für das Verstehen der Antwort muss der Body der Nachricht geparst werden.

Ähnlich der WSDL gibt es auch für RESTful Web-Services Bescheibungssprachen. Die bekannteste ist die WADL (Web Application Description Language). Inzwischen können auch mit der WSDL (seit Version 2.0) reine XML-HTTP-Dienste beschrieben werden, so dass auch diese in die SOAP-Welt Einzug halten.

SOAP-Systeme haben gegenüber REST-basierten Systemen den Vorteil, dass sie Transaktionen o. ä. unterstützen können, im Gegensatz zu REST-Systemen, die per Definition zustandslos sind. Im SOAP ist die Zustandslosigkeit nicht zwingend vorgesehen. Für verteilte Transaktionen mit SOAP existieren mehrere Spezifikationen[51]. Wenn verteilte oder über mehrere Aufrufe gehende Transaktionen nötig sind, dann ist die Verwendung des REST-Architekturstils schwierig.

SOAP bietet etwas weitreichendere Standards bei der Sicherheit und Verschlüsselung als RESTful Web-Services, ohne darauf jetzt weiter einzugehen. Da die in SOAP verwendeten Konzepte meistens auf XML aufbauen, können sie auch auf RESTful Web-Services übertragen werden. Für RESTful Web-Services gibt es dafür keine fertigen Spezifikationen. Eine einfache Möglichkeit der Absicherung der Übertragung steht mit HTTPS zur Verfügung.

RESTful Web-Services sind für die meisten Anwendungsfälle einfacher also SOAP, da dort manche Dinge unnötig kompliziert sind. Dies spiegelt sich auch bei den Zugriffen auf die Web-Schnittstelle von Amazon wieder: die meisten Zugriffe (ca. 85%) erfolgen über die RESTful Schnittstelle[52].

Fazit des Vergleichs RESTful Web-Services sind gut für die typischen Anwendungsfälle Geschäftsobjekte anlegen, auslesen, bearbeiten oder löschen geeignet, weil der Kern dieses Architekturstils Ressourcen (d. h. Geschäftsobjekte) sind, auf die lesend und ändernd zugegriffen wird.

RESTful Web-Services sollten verwendet werden, wenn der zu entwicklende Dienst etwa folgende Eigenschaften hat:

- Es gibt viele lesende Anfragen (dann kann Caching genutzt werden)
- HTTP-Features können oder sollen genutzt werden (neben dem Caching z. B. Ablauf der Gültigkeit, ...),
- Die Ausgabe erfolgt in verschiedenen Datenformaten.
- Dienste sollen auch aus Browsern genutzt werden (können).

Wenn eine oder mehrere der folgenden Eigenschaften gefragt sind, dann ist SOAP vermutlich die bessere Wahl:

- Es gibt komplexe Abläufe, ggf. mit verteilten Transaktionen.
- Die Dienste fordern besondere Zuverlässigkeit der Nachrichtenübertragung.

[51] Für weitere Informationen zu verteilten Transaktionen mit SOAP siehe z. B. [Gerlach 2006].
[52] Nutzung der Amazon-Schnittstellen: [O'Reilly 2003]

3 JAX-RS

Mit dem Java Specification Request 311 [1] möchte die Firma Sun eine High-Level-API für die Entwicklung von RESTful Web-Diensten mit Java entwicklen. Nach der Ankündigung auf den Seiten des Java Community Process am 13. 2. 2007 wurden die ersten konkreten Ideen dazu am 14. 2. von einem der Spezifikationsleiter (Marc Hadley) veröffentlich [2]. Dem ursprünglichen Namen „Java API for RESTful Web Services“ wurde die Abkürzung „JAX-RS“ vorangestellt.

Mit Hilfe der in Java 5.0 eingeführten Annotationen repräsentieren normale Java-Klassen (POJOs) Ressourcen. Die JAX-RS-API ist also ressourcenorientiert. Die HTTP-Aufrufe werden anhand der in der HTTP-Anfrage vom Client empfangenen Wünschen auf verschiedene Java-Methoden gemappt.

Anwendungsentwickler brauchen nur die wirkliche Programmlogik implementieren. Viele anderen Dinge werden über Annotationen deklariert oder mit Hilfe von mitgelieferten Klassen erledigt. Die Java-Methoden der Ressourcen-Klassen wissen über Annotationen, welche Anfragen sie beantworten können, also welche HTTP-Methode, welches Datenformat sie verstehen und erzeugen etc. Ein Beispiel findet sich im Anhang A.5 ab Seite 119. Die Anfragen werden von einer JAX-RS-Laufzeitumgebung entgegengenommen und an die jeweils passende Java-Methode weitergegeben. Der Entwickler muss sich nur sehr wenig mit der Antwort und fast nie mit der Anfrage selber beschäftigen, da die API dies dem Entwickler abnimmt.

Die Spezifikation sollte ursprünglich im Mai 2008 endgültig festgeschrieben werden. Inzwischen wurde der Termin auf September 2008 verschoben, so dass für diese Arbeit keine endgültige Version zur Verfügung steht. Deshalb wird der Public Review Draft der JAX-RS-Spezifikation vom 18.04.2008 (Version 0.8 der API) verwendet. Die Version 0.9 wurde am 27.6. veröffentlicht, also nur 2 Wochen vor der Abgabe dieser Arbeit. Deshalb konnte diese Version nicht berücksichtigt werden.

Die API soll Teil der Java Enterprise Edition (Java EE), Version 6 werden [3].

---

[1] Spezifikation des JSR 311: [Hadley und Sandoz 2008a], Webseite: [Hadley und Sandoz 2007b]
[2] Veröffentlichung der Idee der JAX-RS-API: [Hadley 2007a]
[3] Bestandteile der Java EE 6: [Chinnici und Shannon 2007]

3.1 Ziele

Die JAX-RS-API hat folgende Ziele (siehe Abschnitt 1.2 der JAX-RS-Spezifikation):

- POJO-basiert: Ressourcen werden als annotierte Klassen implementiert. Die API bietet Hilfsklassen / -interfaces für weitere Funktionalitäten an.
- HTTP-zentriert: Die Spezifikation nimmt HTTP als das unterliegende Netzwerk-Protokoll an und unterstützt speziell die HTTP- und URI-Konzepte. Auch auf HTTP basierende Protokolle (z. B. WebDAV [4]) sollen leicht implementiert werden können. Andere Protokolle werden nicht speziell unterstützt.
- Format-Unabhängigkeit: Die API soll nicht nur ein spezielles Datenformat (wie typischerweise XML) unterstützen, sondern alle Datenformate.
- Container-Unabhängigkeit: Anwendungen sollen in vielen Web-Containern laufen können. Die Spezifikation definiert einige Möglichkeiten, wie JAX-RS-Laufzeitumgebungen bspw. in Servlet-Container und als JAX-WS-Provider eingebunden werden können.

3.2 Beschreibung

Dieser Abschnitt gibt einen Überblick über die JAX-RS-Spezifikation und -API, im wesentlichen aus Sicht eines Entwicklers, der eine Anwendung implementieren möchte, die diese Spezifikation nutzt. In diesem Abschnitt werden nicht alle Details beschrieben; in der Anforderungsanalyse im Abschnitt 4.1 auf Seite 66 werden bei Bedarf weitere Einzelheiten erläutert.

3.2.1 Ressource-Klassen und -Methoden

Ein Anwendungsentwickler implementiert die Logik seiner Anwendung in *Ressource-Klassen* [5] Eine *Root-Ressource-Klasse* stellt die Wurzel eines Baumes von Ressource-Klassen dar. Eine Root-Ressource-Klasse ist mit @Path annotiert. Diese Annotation enthält

[4] WebDAV ist ein Standard zur Bereitstellung von Dateien über das Internet, um darauf wie über ein normales Dateisystem zugreifen zu können; spezifiziert in [RFC-4918 2007]

[5] Die in diesem Abschnitt hervorgehobenen Begriffe stammen aus dem Abschnitt 1.5 der JAX-RS-Spezifikation. Da diese Arbeit in deutsch verfasst ist, werden die Begriffe meistens ins Deutsche übersetzt. Da es für das englische Wort *locator* keine dem englischen Wort ähnliche deutsche Übersetzung existiert, wird dieser Begriff nicht übersetzt.

den Pfad, unter der sie relativ zum Pfad der Laufzeitumgebung zu erreichen ist. Ansonsten gilt in dieser Arbeit für Root-Ressource-Klassen das, was auch für normale Ressource-Klassen gilt, es sei denn, es ist extra angeführt.

Zur Ausführung von Logik auf einer Ressource-Klasse dienen *Ressource-Methoden.* Diese Java-Methoden sind mit einem *Request-Method-Designator* annotiert. Ein Request-Method-Designator ist eine Annotation, die mit @HttpMethod annotiert ist. Ein Request-Method-Designator definiert die zu verwendende HTTP-Methode für eine Ressource-Methode. In der JAX-RS-API sind 5 Request-Method-Designators enthalten: @GET, @POST, @PUT, @DELETE und @HEAD. In dieser Arbeit wird im Folgenden davon gesprochen, dass eine Java-Methode mit einer HTTP-Methode annotiert ist. Diese Analogie wird auch bei anderen Annotationen verwendet, z. B. wird davon gesprochen, dass eine Root-Ressource-Klasse oder eine Java-Methode mit einem Pfad (@Path) annotiert ist.

Wenn eine zu implementierende Ressource eine untergeordnete Ressource enthält, die wenig Logik benötigt, dann können dafür *Sub-Ressource-Methoden* verwendet werden. Sie ist wie eine normale Ressource-Methode mit einer HTTP-Methode annotiert. Zusätzlich ist sie noch mit @Path annotiert. Dieser Pfad gilt relativ zum Pfad der aktuellen Ressource-Klasse. Außer der zusätzlichen @Path-Annotation werden Sub-Ressource-Methoden und Ressource-Methoden gleich behandelt. Wenn in dieser Arbeit von Ressource-Methoden gesprochen wird, sind deshalb immer auch Sub-Ressource-Methoden gemeint, es sei denn es wird explizit darauf hingewiesen.

Benötigt eine untergeordnete Ressource jedoch viel Logik, so dass die übergeordnete Ressource-Klasse bei Verwendung von vielen Sub-Ressource-Methoden unübersichtlich werden würde, dann kann für die Logik der untergeordneten Ressource eine eigene Ressource-Klasse implementiert werden. Sie ist über einen *Sub-Ressource-Locator* zu erreichen. Ein Sub-Ressource-Locator eine mit @Path, aber ohne eine HTTP-Methode annotierte Java-Methode. Passt der Pfad einer HTTP-Anfrage zum angegeben Pfad des Sub-Ressource-Locators, dann wird die entsprechende Java-Methode ausgeführt. Das von der Methode erzeugte Objekt wird dann erneut als Ressource-Objekt (Instanz einer Ressource-Klasse) behandelt und dort weiter nach zur Anfrage passenden Java-Methoden gesucht.

Zur Auswahl der aufzurufenden Java-Methode wird neben der HTTP-Methode und dem Pfad der Anfrage auch das Datenformat des Anfragekörpers und die gewünschten Datenformate für die HTTP-Antwort berücksichtigt. Dazu können die Ressource-Methoden mit @ConsumeMime und / oder @ProduceMime annotiert werden. Wenn die Datenformate von HTTP-Anfrage und Java-Methode nicht zusammenpassen, wird nach anderen passenden Methoden gesucht. Wenn keine passende Ressource-Methode gefunden wird, dann wird die Anfrage als nicht bearbeitbar mit einem entsprechenden HTTP-Status zurückgewiesen.

Die Laufzeitumgebung ist für die Instantiierung und Initialisierung der Root-Ressource-Klassen und den Aufruf der Methoden verantwortlich.

3.2.2 Parameter von Ressource-Methoden

Die von der Laufzeitumgebung aufgerufenen Methoden und Konstruktoren können auch Parameter haben. Wenn ein Parameter ohne JAX-RS-Annotationen versehen ist, dann wird der Java-Methode dort der Nachrichtenkörper der HTTP-Anfrage übergeben. Dieser kann in verschiedenen Javatypen angefordert werden. Die Laufzeitumgebung ist dafür verantwortlich, den Anfragekörper in den gewünschten Javatyp umzuwandeln.

Alle anderen Parameter außer dem Anfrage-Körper müssen mit (mind.) einer JAX-RS-Annotation versehen sein. Für alle Parameter der Anfrage stehen Annotationen zur Verfügung, um einfach darauf zugreifen zu können:

- @CookieParam: Die Variable enthält beim Aufruf der Methode den Wert, den der Cookie mit dem angegebenen Namen hat. Wenn der Parameter vom Typ `Cookie` ist, dann wird dort der ganze Cookie übergeben.
- @HeaderParam: Die Variable enthält den Wert der Kopfzeile mit dem angegebenen Namen.
- @MatrixParam: Die Variable enthält beim Aufruf der Methode den Wert des Matrix-Parameters [6] mit den in der Annotation angegebenen Namen. Die JAX-RS-Spezifikation erlaubt Matrix-Parameter nicht nur am Ende des Pfades [7], sondern am Ende jedes Pfad-Segmentes.
- @PathParam: Mit @Path angegebene Pfade dürfen auch Variablen enthalten. Diese müssen dort in geschweifte Klammern eingeschlossen werden [8]. Über die Annotation @PathParam auf einem Parameter können diese Variablen abgefragt werden.
- @QueryParam: Die Variable enthält den Wert des Query-Parameters mit den in der Annotation angegebenen Namen.

Da diese Parameter meistens sehr ähnlich behandelt werden, werden sie in dieser Arbeit zusammenfassend mit @*Param bezeichnet. Erlaubt sind bei allen Variablen primititive Typen und alle Klassen, die eine Klassenmethode valueOf(String) haben, oder einen Konstruktor mit dem Typ String als einzigen Parameter. Für in der Anfrage nicht gegebene Werte kann ein Standardwert festgelegt werden. Dazu kann überall, wo @*Param auf einem Parameter erlaubt ist (außer auf @PathParam), zusätzlich mit @DefaultValue ein Standardwert vorgegeben werden.

Standardmäßig werden Parameter aus dem URI dekodiert. Dies lässt sich unterdrücken, indem der Anwendungsentwickler auf einem Methodenparameter, einer Ressource-Methode oder einer Ressource-Klasse die Annotation @Encoding hinzufügt.

[6] Zu Matrix-Parametern siehe Abschnitt 2.1.1 ab Seite 9.
[7] In [RFC-1808 1995] und [Berners-Lee 1996] sind Matrix-Parameter nur am Pfad-Ende vorgesehen.
[8] Siehe als Beispiel die Methode EmployeesResource.getSub() im Beispiel in Anhang A.5 ab Seite 119.

Wenn ein Parameter mit @Context annotiert ist, dann muss der Typ des Parameters eines der folgenden Interfaces sein:

- Das Interface `UriInfo`
  - ermöglicht den Zugriff auf Informationen zum angefragten URI,
  - enthält Methoden, um neue – von der Anfrage abhängige URIs – zu bauen und
  - speichert die bisher für diese Anfrage aufgerufenen Instanzen der Ressource-Klassen und die gematchten URIs zu den bisher durchlaufenen Sub-Ressource-Locatorn.
- Das Interface `HttpHeaders` ermöglicht den Zugriff auf die HTTP-Kopfzeilen der Anfrage. Für häufig verwendete Kopfzeilen stehen spezielle Getter-Methoden – teilweise mit High-Level-Objekten als Rückgabetypen – zur Verfügung.
- Das Interface `Request` bietet die Möglichkeit, Vorbedingungen der Anfrage auswerten zu lassen. Mit den Details der Auswertung braucht sich der Anwendungsentwickler nicht zu beschäftigen.
  Als zweites Feature kann aus einer vorgegebenen Liste von zur Verfügung stehenden Varianten die am besten zur Anfrage passende Variante auszuwählt werden.
- Das Interface `SecurityContext` enthält Informationen zum Benutzer der Anfrage (wenn in der HTTP-Anfrage angegeben) und eine Prüfung, ob ihm eine Rolle zugewiesen wurde. Außerdem kann geprüft werden, welches Authentifikationsverfahren verwendet wurde und ob die Anfrage verschlüsselt war oder nicht.
- Das Interface `MessageBodyWorkers` ist – wie auch das folgende – hauptsächlich für Provider gedacht: Mit ihm kann ein Entity-Provider[9] die Konvertierungsmöglichkeiten anderer Entity-Provider nutzen.
- `ContextResolver`: Context-Resolver ermöglichen es, für bestimmte Klassen besondere Contexte zu verwenden, z. B. für die Konvertierung mit JAXB.
- Laufzeitumgebungen dürfen weitere eigene Typen definieren, die dann der jeweiligen Ressource-Methode etc. übergeben werden.

Um in Root-Ressource-Klassen übersichtliche Parameterlisten von Ressource-Methoden zu ermöglichen, können auch Instanzvariablen und Bean-Setter wie Methoden-Parameter annotiert werden. Die Laufzeitumgebung injiziert dann nach der Instantiierung die entsprechenden Objekte.

[9] Die JAX-RS-Spezifikation verwendet anstatt des Begriff Repräsentation den Begriff Entität; in der HTTP-Spefizikation [RFC-2616 1999] werden beide Begriffe verwendet.

3.2.3 Verarbeitung der Rückgabewerte von Ressource-Methoden

Die Rückgabe-Werte von Ressource-Methode werden wie folgt behandelt:

- Wenn die Methode nichts (`void`) oder `null` zurückgibt, resultiert daraus der Antwort-Status 204 (No Content) und ein leerer Nachrichtenkörper.
- Wenn die Methode ein Objekt vom Typ `Response` zurückgibt, dann wird die enthaltene Antwort direkt zurückgegeben. Die enthaltene Entität wird mit einem Message-Body-Writer (siehe weiter unten) serialisiert als Nachrichtenkörper der Antwort verschickt.
- Für alle anderen Objekte wird der HTTP-Status 200 zurückgegeben. Das Objekt wird als Entität der Antwort an den Client – wie im vorigen Fall beschrieben – in den Nachrichtenkörper umgewandelt.

Die Entität einer zurückgegebenen `Response` und ein direkt von der Ressource-Methode zurückgegebenes Objekt sind immer gleich zu behandeln. Diese Gleichbehandlung wird im Folgenden nicht weiter explizit genannt.

3.2.4 Hilfsklassen

Um die beschriebenen Möglichkeiten leicht verwenden zu können, stehen einige Hilfsklassen zur Verfügung:

- Eine Ressource-Methode kann – wie erwähnt – ein Objekt vom Typ `Response` zurückgeben. Sie speichert den Status der Antwort, die HTTP-Kopfzeilen und die Entität der HTTP-Antwort. Um eine Response zu erzeugen, gibt es die Hilfsklasse `ResponseBuilder`.
- Um einen URI zu erzeugen, gibt es die Klasse `UriBuilder`. Da Repräsentationen häufig Links zu internen Zielen enthalten sollen, lassen sich mit Hilfe des Interfaces `UriInfo` UriBuilder URIs zu Zielen innerhalb der aktuellen Anwendung erzeugen. Wenn ein URI unabhängig von dem URI der aktuell angefragten Ressource erzeugt werden soll, kann eine Instanz des UriBuilders auch mit statischen Methoden dieser Klasse erzeugt werden.
  Der UriBuilder kodiert die übergebenen Werte wie es für einen URI vorgeschrieben ist. Die automatische Kodierung kann auch abgeschaltet werden; dann muss der UriBuilder überprüfen, ob übergebene Werte ungültige Zeichen enthalten und den Methodenaufruf ggf. zurückweisen. Beim UriBuilder ist es – ähnlich wie bei der Annotation @Path – möglich, mit geschweiften Klammern Variablen einzubauen und diese erst beim Erzeugen des URIs mit konkreten Werten zu füllen.

- Wenn es nötig oder sinnvoll ist, direkt in einer Ressource-Methode auf den Output-Stream zu schreiben, dann kann eine Ressource-Methode ein Objekt vom Typ StreamingOutput zurückgeben. Die – typischere anonyme – Klasse implementiert dann die Methode write(OutputStream).

Zum Setzen oder Ändern eines Cookies auf dem Client kann die Klasse `NewCookie` verwendet werden. Dort lassen sich die benötigten Attribute einfach setzen. Die JAX-RS-Laufzeitumgebung ist für die Serialisierung nach der Cookie-Erweiterung [10] für HTTP verantwortlich.
Caching-Informationen können mit Hilfe der Klasse `CacheControl` gesetzt werden.

3.2.5 Provider

Provider ermöglichen es dem Anwendungsentwickler, der Laufzeitumgebung Funktionalität hinzuzufügen.

Entity-Provider

Damit sich der Anwendungsentwickler in der Ressource-Methode nicht um die Deserialisierung oder Serialisierung von Objekten kümmern muss, gibt es Entity-Provider (Message-Body-Reader und Message-Body-Writer), die dies erledigen. Der Entwickler muss in den Ressource-Methoden nur den gewünschten Datentyp angeben, bzw. das Objekt zurückgeben. Die JAX-RS-Laufzeitumgebung sucht nach einem passenden Entity-Provider und wandelt das Objekt entsprechend um.

Jede JAX-RS-Laufzeitumgebung muss einige Entity-Provider zur Verfügung stellen, z. B. für Strings, XML-Daten und mit POST übertragene HTML-Formulare. Ein Anwendungsentwickler kann eigene Entity-Provider implementieren und der Laufzeitumgebung hinzufügen.

Exception-Mapper

Exception-Mapper wandeln von Ressource-Methoden und Sub-Ressource-Locatorn geworfene Exceptions in eine JAX-RS-Response um. Damit können dem Client z. B. sinnvolle Fehlermeldungen zur Verfügung gestellt werden, ohne dass der Quellcode der Ressource-Methoden / Sub-Ressource-Locatorn unübersichtlich wird.

[10] Cookies sind in [RFC-2965 2000] spezifiziert.

Context-Resolver

Für Entity-Provider für bestimmte Datentypen sind in Einzelfällen ggf. besondere Contexte gewünscht, z. B. für JAXB. Um dies zu ermöglichen, kann ein Provider verlangen, dass er einen `ContextResolver` initiiert bekommt. Dieser gibt dann für die ihm bekannten Datentypen einen entsprechenden Kontext zurück.

3.2.6 Extension-Mapping

Der Client hat die Möglichkeit der einfachen Wahl des gewünschten Datentyp. Dies ist außer mit der in das HTTP integrierten Content-Negotiation (siehe Abschnitte 2.2.5 und 2.2.7 ab Seite 25) auch mit Hilfe des URIs möglich. Dazu kann an den URI eine „Dateiendung“ angehängt werden, welche das Datenformat oder die Sprache vorgeben. Die zu mappenden Datenformate und Sprachen werden jeweils mit der korrespondierenden Dateiendung beim Start der Laufzeitumgebung übergeben.

3.2.7 ApplicationConfig

Ein Anwendungsentwickler übergibt seine Root-Ressource-Klassen, seine eigenen Provider und die Informationen zum Extension-Mapping in einer Unterklasse von `ApplicationConfig` an die Laufzeitumgebung. Wie dies geschieht, definiert jede Laufzeitumgebung selber.

3.3 Der Bedarf nach einer API wie JAX-RS

Viele der existierenden APIs wie bspw. die Servlet-API unterstützen in RESTful Web-Services sinnvolle Möglichkeiten nicht; sie stellen nur die benötigten Kopfzeilen der zu verarbeitenden Anfrage als Strings zur Verfügung bzw. Kopfzeilen können als Strings in die Antwort gesetzt werden. Deshalb ist detailiertes Wissen für die Umsetzung nötig, z. B. der genaue Name der Kopfzeile und das genau zu verwendende Format für den Wert der Kopfzeile. Deshalb verleiten diese APIs dazu, entsprechende Kopfzeilen nicht auszuwerten und / oder zu setzen, wie es im HTTP eigentlich vorgesehen ist. Dies betrifft vor allem folgende Punkte:

- Content-Negotiation
- Caching-Unterstützung (Kopfzeilen „Last-Modified“, „Cache-Control“ und „Entity-Tag“)

- Vorbedingungen von Anfragen werden von diesen APIs nicht ohne zusätzlichen Aufwand unterstützt.
- Da die Kopfzeilen nur als String zur Verfügung stehen, müssen diese im Quellcode umgewandelt werden, was den Quellcode unübersichtlich macht. Auch wenn dies nicht in der Geschäftslogik geschieht, enthält der konvertierende Programmcode keine wirkliche Logik, sondern nur Konvertierung und Fehlerbehandlung. Dieser Code muss für jeden Anwendungsfall ganz ähnlich neu implementiert werden.
- Da die APIs meistens nur den direkten Zugriff auf den Input- und OutputStream ermöglichen, verleiten sie dazu, die (De-)Serialisierung direkt in der Geschäftslogik umzusetzen. Auch wenn die Geschäftlogik sauber von der Konvertierung getrennt ist, muss die Konvertierung muss für jede Anwendung neu implementiert werden.

Die Servlet-API und andere APIs bieten Möglichkeiten an, die in im REST-Architekturstil implementierten Anwendungen verboten sind, z. B. die Nutzung von Sessions.

Die Restlet-API unterstützt etliche der aufgezählten Features einfacher als andere APIs:

- Content-Negotiation ist in die Restlet-API integriert, muss aber für jede Ressource doppelt kodiert werden (siehe Abschnitt 2.4.1 auf Seite 34 und Anhang A.3 auf Seite 117). Dafür stehen Objekte mit hohem Abstraktionsniveau zur Verfügung.
- Caching wird bis jetzt noch nicht von der Restlet-API unterstützt. Die Umsetzug ist vorgesehen.
- Vorbedingungen werden mit der Methode Conditions.getStatus() unterstützt.
- Viele Standard-Kopfzeilen stehen als High-Level-Objekt zur Verfügung. Pfade müssen per Programmcode in der Ressourcen-Implementierung vom String in das gewünschte Format konvertiert werden.
- Auf den Input- und OutputStream wird in den Ressource-Klassen normalerweise nicht zugegriffen. Für die Konvertierung sind die Repräsentations-Klassen verantwortlich. Wenn es nötig ist, kann auch aus einer Ressourcen-Implementierung der InputStream der Anfrage ausgelesen werden oder Daten in den OutputStream einer Repräsentation geschrieben werden, ähnlich dem JAX-RS-Interface `StreamingOutput` (siehe Abschnitt 3.2.4 ab Seite 49).

Die JAX-RS-API unterstützt die genannten Punkte direkt:

- Die Content-Negotiation ist mit JAX-RS noch einfacher als mit Restlet, weil sie nur einmal deklarativ implementiert werden muss (als Annotation auf der entprechenden Ressource-Methode oder Ressource-Klasse).
- Vorbedingungen der Anfrage, z. B. bzgl. des Cachings, können mit der Methode `Request.evaluatePredocditions()` einfach ausgewertet werden.

- Die JAX-RS-API erlaubt es, in Ressource-Methoden Objekte mit hohem Abstraktionsniveau aus der Anfrage abzufragen oder in eine Antwort zu setzen. So wird die Anwendungslogik übersichtlich und entsprechend einfach zu pflegen.
- Ein Beispiel für von der Laufzeitumgebung übernommene Aufgaben ist die automatische Konvertierung von Parametern der Anfrage in viele verschiedene Javatypen.
- Mit Hilfe der Message-Body-Reader stellt die Laufzeitumgebung die Objekte in gewünschten Javatypen zur Verfügung. Bei Bedarf kann der InputStream direkt ausgelesen werden. Auch ein Schreiben direkt auf den OutputStream ist möglich (siehe Abschnitt 3.2.4 ab Seite 48 und Anhang A.5 auf Seite 119).

Sessions und andere in RESTful Web-Services nicht erlaubte Features werden von der JAX-RS-API nicht unterstützt.

Die Entwicklung von RESTful Anwendungen wird mit der JAX-RS-API für Entwickler wesentlich vereinfacht. Der Quellcode ist besser zu lesen, da Entwickler nur noch die wirkliche Geschäftslogik implementieren müssen. Die JAX-RS-Spezifikation ermöglicht es dem Anwendungsentwickler, einfache Sachen einfach zu implementieren. Komplizierte Dinge können ebenfalls umgesetzt werden.

3.4 Umsetzung des REST-Architekturstils

Die JAX-RS-Spezifikation setzt die Ressourcen-orientierte Architektur des REST-Architekturstils sehr gut um. Sie verlangt auch, dass Anwendungsentwickler ressourcenorientiert modellieren.

Im folgenden wird die JAX-RS-Spezifikation anhand der Punkte aus der Vorstellung der REST-Architektur-Konzepten aus Abschnitt 2.1.3 (ab Seite 13) bewertet:

1. Das Client-Server-Modell ist inhärent in jeder HTTP-Anwendung implementiert.
2. Der Standard-Lebenszyklus für Root-Ressource-Klassen bildet die Zustandslosigkeit des Servers im REST-Architekturstil ab: Die Instanz der Root-Ressource-Klasse mit allen enthaltenen Informationen wird nach der Bearbeitung der Anfrage zerstört, und damit im Normalfall alle in dieser Anfragebearbeitung verwendeten Informationen. Sessions – wie bspw. in der Servlet-API – werden von JAX-RS explizit nicht unterstützt.
3. Die API unterstützt den Anwendungsentwickler bei der Nutzung des Cachings: Die Auswertung von Vorbedingungen wie in bedingten GET-Anfragen lässt sich mit der Methode Request.evaluatePreconditon(..) sehr leicht implementieren. Auch das Setzen von Einstellungen bzgl. des Cachings lassen sich mit der Klasse `CacheControl` sehr leicht realisieren. Das gleiche gilt für Entity-Tags und das Datum der letzten Änderung einer Repräsentation.

4. Die API unterstützt den Anwendungsentwickler dabei, die im HTTP-Protokoll vorgesehene einheitliche Schnittstelle zu implementieren, indem sie ihn nötigt, nur wohldefinierte (HTTP-)Methoden zu verwenden. Andernfalls müsste er eine neue Annotation für die zu verwendende Methode implementieren. Dies wird er nicht „mal eben“ tun, wenn ihm bewusst ist, dass die korrekte und möglichst ausschließliche Verwendung von standardisierten Methoden ein wichtiger Punkt von mit HTTP implementierten RESTful Web-Services ist.

   Die API verlangt auch nicht die Definition von Java-Interfaces als Schnittstellen. Im REST-Architekturstil ist das verwendete (Netzwerk-)Protokoll – in diesem Fall also das HTTP – die einheitliche Schnittstelle. Diese Architektur der JAX-RS-API verhindert, dass der Architekt oder Anwendungsentwickler in exakten Verträgen oder Java-Interfaces planen, denn dies ist im REST-Architekturstil nicht vorgesehen. Dies entkoppelt den Client vom Server und auch von dem Namen der Java-Methode.

5. Da die JAX-RS-Spezifikation nur die Server-seitige Implementierung von Diensten betrachtet, lässt sich Schichtung mit dieser API alleine nicht implementieren. Dies liegt aber entsprechend der im Abschnitt 3.1 auf Seite 44 genannten Ziele nicht im Bereich dieser Spezifikation.

6. Die Erzeugung von Code-on-Demand wird von der API nicht explizit unterstützt. Dies ist aber auch keine vorgeschriebene REST-Architektur-Bedingung. Die Auslieferung von im Dateisystem liegende Dateien ist mit dem für die Klasse `File` vorgesehenen Entity-Provider sehr einfach, so dass ein Client leicht fertigen Code vom Server herunterladen kann.

Für alle diese Punkte gilt, dass die API dem Anwendungsentwickler hilft, diese Dinge richtig zu entwerfen und zu implementieren, ihn aber nicht dazu zwingen kann. Sie kann z. B. nicht verhindern, dass ein Anwendungsentwickler Sessions oberhalb dieser API nachbildet. Da eine API dies grundsätzlich nicht leisten kann, lässt es sich auch von dieser Spezifikation nicht verlangen.

Die im REST-Architekturstil verlangte Verlinkung der Ressourcen wird durch Links zu weiteren Ressourcen in den Repräsentationen umgesetzt. Der `UriBuilder` (zusammen mit den Informationen aus dem Interface `UriInfo`) unterstützt den Anwender bei der einfachen Erzeugung von URIs.

Die JAX-RS-Architektur mit der kompletten Trennung der (De-)Serialisierung der Entitäten von der eigentlichen Logik in den Ressource-Methoden erleichtert es dem Anwendungsentwickler, verschiedene Datenformate zur Verfügung zu stellen. Damit eine oder auch mehrere Ressourcen neben einem bspw. schon unterstützten XML-Format auch JSON unterstützen, muss einmal ein passender Entity-Provider implementiert werden und die Annotationen @ConsumeMime bzw. @ProduceMime um „application/json“ ergänzt werden. Dann unterstützen diese Ressourcen auch das JSON-Format.

3.5 Anfängliche Kritik

Die Planung einer Spezifikation für RESTful Web-Services wurde am 13. 2. 2007 auf den Seiten des Java Community Process (JCP) veröffentlicht [11]. Über den Sinn einer solchen Spezifikation wurde in den Tagen nach der Veröffentlichung viel diskutiert. Dabei muss berücksichtigt werden, das auf den Webseiten des JCP keinerlei Details aufgeführt sind, so dass über die Umsetzung nur spekuliert werden konnte. Daraufhin hat einer der beiden Spezifikationsleiter (Marc Hadley) einen Blog-Artikel über die Planungen eher veröffentlicht [12] als von ihm geplant. Dieser Artikel enthält auch einige Code-Beispiele. Bei der Betrachtung der Meinungen ist also zu berücksichtigen, dass diese Beispiele vielen Schreibern zum Zeitpunkt der Veröffentlichung ihrer Gedanken noch nicht bekannt waren.
Dieser Abschnitt gibt einen kurzen Überblick über die Meinungen verschiedener Leute.

Roy Fielding bspw. hält nicht viel von solchen oder ähnlichen Spezifikationen [13] und erlaubt Sun nicht, „REST“ als Teil des API-Namens zu verwenden (mehr dazu auf Seite 55). Joe Gregorio und Roy Fielding haben die Angst [14], dass Java-Entwickler in Zukunft vielleicht meinen, dass alles, was mit der JAX-RS-API implementiert werden kann, sei RESTful und alles andere nicht. Dass diese Angst nicht ganz unberechtigt ist, zeigen ähnliche Missverständnisse und Unkenntnisse bzgl. REST, die es schon länger gibt (siehe auch Abschnitt 2.3.3 auf Seite 30). Stefan Tilkov kommentiert, dass in Java lieber erstmal HTTP besser unterstützt werden sollte, bevor da ggf. eine Schicht draufgesetzt wird. Dies wird auch von anderen unterstützt [15].

Einige kritisierten auch, dass die Spezifikation von den Leuten geleitet wird, die auch JAX-RPC und JAX-WS mit spezifiziert haben. Diese Spezifikationen finden viele Anhänger des REST-Architekturstils nicht schön und haben oder hatten Angst, dass die JAX-RS-Spezifikation in ihren Augen ähnlich schlecht wird. Diese Sorge wird bspw. auch von Mark Baker geteilt. Er meint aber auch, dass bessere APIs als die bis dahin vorhandenen benötigt werden [16]. Auch der Name des Java-Packages (siehe auch Seite 56) deutete für viele darauf hin, dass die Spezifikation am Ende ganz ähnlich den SOAP-Web-Service-Spezifikationen sein würde [17].
Einige der Kritiker haben sich zur JSR-Expert-Group angemeldet, um an der Spezifikation mitzuarbeiten.

[11] Veröffentlichung der JSR-311-Planung: [Hadley und Sandoz 2007b]
[12] Veröffentlichung weiterer Details mit erstem Code-Beispiel: [Hadley 2007a]
[13] z. B. [Fielding 2007a], [Fielding 2007c]
[14] z. B. [Gregorio 2007]
[15] z. B. [Tilkov 2007b] und [Semergence 2007]
[16] [Harold 2007b], [Baker 2007]. Einige Kritiker haben ihre Meinung inzwischen geändert: [Harold 2007c].
[17] [Loughran 2007]

Der Restlet-Erfinder Jérôme Louvel hingegen findet den Ansatz gut und arbeitet ebenfalls im Java Community Process (JCP) an der Spezifikation mit. Auch andere reagierten positiv bis euphorisch auf die Ankündigung der JAX-RS-Spezifikation [18].

Einige Blog-Schreiber [19] haben sich zum Zeitpunkt der ersten Veröffentlichung Gedanken über Anforderungen gemacht. Hier folgt eine (nicht abschließende) Zusammenfassung von Gedanken und Anforderungen verschiedener Leute an die Spezifikation:

1. Die API soll ein hohes Abstraktionsniveau haben.
2. Sie soll deklarativ sein.
3. Sie soll das Ressourcen-orientierte Design einer ROA voll unterstützen und die Konzepte des REST klar mappen.
4. Sie sollte keinen sich ständig wiederholenden Quellcode nötig machen.
5. Die API soll es einfach machen, das Richtige zu tun.
6. Sie sollte es ermöglichen, bei Bedarf auf APIs mit einem niedrigeren Level zurückgehen zu können.
7. Die Spezifkation soll auch andere Datenformate als XML unterstützen. Dazu gehören auch große binäre Daten wie gestreamte Videos.
8. Serverseitig soll kein kompletter Anwendungsserver nötig sein. JAX-RS-Anwendungen sollen auch auf „kleinen“ HTTP-Servern laufen, wie z. B. auf Jetty oder dem in Java SE 6 enthaltenen HTTP-Server.
9. Die API soll nicht für (noch) nicht existierende zukünftige Protokolle ausgelegt sein. Dies würde sowohl den Spezifikations-Prozess als auch hinterher die Spezifikation selber komplizierter machen.
10. Die API soll dynamische URIs unterstützen, basierend auf URI-Templates [20].
11. Sie soll alle HTTP-Methoden unterstützen können, bspw. auch die von WebDAV.
12. Die API soll verschiedene Repräsentationen einer einzigen Ressource unterstützen.

Seit Februar 2007 ist die Diskussion in der REST-Community über die JAX-RS-Spezifikation allerdings ziemlich schnell ruhig geworden.

3.5.1 Name

Ursprünglich war für die Spezifikation der Name „Java API for RESTful Web Services“ vorgesehen. Roy Fielding – als Namensgeber des Architekturtstils REST – hatte jedoch den

[18] Zustimmungen: z. B. [Diephouse 2007], [Sandoz 2007b], [Lacey 2007] und [de hÓra 2007]
[19] Anforderungen: z. B. [Law 2007], [Loughran 2007], [Louvel 2007c]. Gesammelte Meinungen: [Tilkov 2007b].
[20] URI-Templates: [Gregorio u. a. 2008]

Wunsch, dass es eine Abkürzung geben soll, die keinen direkten Bezug zum Begriff REST hat, damit der Architekturstil klar von der API unterschieden wird [21]. Deshalb wurde der Name am 5. 4. 2007 zu „JAX-RS: Java API for RESTful Web Services“ geändert [22].

„JAX“ bedeutet in den anderen mit „JAX“ beginnenden Java-Spezifikationen „Java API for XML“, so dass schnell der Eindruck entstehen kann, das auch JAX-RS nur das Datenformat XML unterstützt. Dies ist jedoch nicht der Fall, denn JAX-RS unterstützt alle Datenformate. Dazu sagt Marc Hadley, dass das X stumm sei, also in diesem Fall nichts bedeute [23]. Die Abkürzung JAR-WS war ebenfalls im Gespräch; da JAR im Java-Umfeld ein feststehender Begriff für „Java Archive“ ist, bevorzugten viele doch „JAX-RS“, auch wenn das X keine Bedeutung hat.
Elliotte Rusty Harold schlug vor, die API in „Java HTTP Server API“ umzubenennen, da eine API keine architektonischen Entscheidungen erzwingen kann, was der Namensbestandteil „RESTful“ aber andeute [24].

Der Autor dieser Arbeit ist auch der Meinung, dass das X in JAX-RS nicht gut gewählt ist. Einer Umbenennung, in der mehr deutlich wird, dass JAX-RS eine HTTP-API ist, steht er positiv gegenüber. Dabei sollte aber auch zu erkennen sein, dass die API auf RESTful Web-Services ausgerichtet ist. Die Servlet-API (bzw. dessen HTTP-Anteil) ist auch eine Java-HTTP-Server-API, so dass ein etwas speziellerer Name als der Vorschlag von Elliotte Rusty Harold nach Meinung des Autors dieser Arbeit doch gut ist.

3.5.2 Package-Name

Als Package-Name war ursprünglich javax.ws.rest vorgesehen. Da Roy Fielding aus den in den beiden vorigen Abschnitten genannten Gründen die Bezeichnung „REST“ auch nicht im Package-Namen haben wollte, wurde dieser auf seinen Vorschlag hin zu javax.ws.rs geändert. Direkt nach dieser Änderung wurden noch verschiedene andere Vorschläge gemacht, z. B. javax.net.rws, javax.net.rs, javax.rws, javax.jws.rs, javax.web.rs, javax.rest.ws [25]. Einige Leute hatten auch aufgrund des „ws“ im Package-Namen Angst, dass die Spezifikation am Ende sehr ähnlich den WS-*-Spezifikationen wird [26]. Diese Angst wäre aus Sicht des Autors dieser Arbeit durch eine Umbenennung des Packages in „javax.jws.rs“ genährt worden, denn einige SOAP-Web-Service-Packages (JAX-WS) beginnen mit „javax.jws“.

Letztlich wurde die Änderung zu javax.ws.rs nicht noch einmal geändert.

---

[21] Einwände durch Roy Fielding: [Fielding 2007c]. Erklärungen: [Fielding 2007d], [Fielding 2007e]
[22] Namensänderung: [Hadley und Sandoz 2007b]
[23] Das „X“ in JAX-RS ist stumm: [Hadley 2007d]
[24] [Harold 2007c]
[25] Vorschläge für den Package-Namen: [Prasanna 2007], [Louvel 2007a], [Hadley 2007e], [Louvel 2007b]
[26] javax.ws.rest zu nahe an WS-*: z. B. [Loughran 2007]

3.5.3 Meinung des Autors dieser Arbeit

Nach Meinung des Autors dieser Arbeit ist die Angst vor einer Ähnlichkeit zu den WS-*-Spezifikationen mit Blick auf die damals – nicht – vorhandenen Informationen nachvollziehbar. Inzwischen zeigt sich aber, dass die API es Anwendungsentwicklern leicht macht, die REST-Architekturbedingungen einzuhalten. Auch die ab Seite 55 zusammengestellten Anforderungen werden alle erfüllt, ohne dass dies aus Platzgründen Punkt für Punkt aufgeführt wird. Einige dieser Punkte wurden auf den vorigen Seiten schon behandelt oder werden auf den nächsten Seiten angeführt.

3.6 Diskussionen in der Expert-Group

Im Rahmen einer Spezifikation, an denen viele Leute mitwirken können, gibt es naturgemäß fast immer Diskussionen, was denn genau alles spezifiziert werden soll. Diese Diskussionen finden im Rahmen des Java Community Process im wesentlichen in der für den JSR zuständigen Expert-Group statt. Einige viel diskutierte Punkte werden hier aufgeführt.

3.6.1 Annotationen oder Javatypen

Im ersten Monat der Diskussionen in der Expert-Group wurde besonders viel darüber diskutiert, wie viel der Funktionalität mit Annotationen und wieviel mit Klassen oder Interfaces umgesetzt werden sollte [27]. Annotationen ermöglichen flexibleren Programmcode, dafür wird zur Verarbeitung derselben etwas mehr Zeit benötigt. Dieser Zeitbedarf lässt sich aber durch Caching optimieren. Wenn nur Annotationen verwendet werden, lassen sich nicht alle Anwendungsfälle implementieren, oder die Annotationen würden sehr aufwendig.

Deshalb wurde vorgeschlagen, für die Dinge, die sich nicht mit den High-Level-Annotationen implementieren lassen, auf den verwendeten Container zurückzugreifen. Nach Meinung vieler anderer Mitglieder der Expert-Group und auch des Autors dieser Arbeit sollte es mit der API möglich sein, alle Anwendungsfälle abzudecken. Ein Argument gegen die häufige Verwendung von Klassen ist, dass es schon eine API gibt, die Klassen verwendet, nämlich Restlet. Besonders viel wurde über die am Ende dieser Diskussionen entfernten Typen Representation<T> und Entity<T> diskutiert, in welcher die vom Client empfangene Repräsentation an die Ressource-Methode übergeben wurde, bzw. die Ressource-Methode zurückgab und dann serialisiert werden sollte.

[27] Annotationen oder Javatypen? [JSR311-Expert-Group 2007d], [JSR311-Expert-Group 2007c], [JSR311-Expert-Group 2007a]

3.6.2 Client-API

Von Anfang an war in allen Versionen der Spezifikation[28] explizit angegeben, dass eine Client-API im Rahmen dieser Spezifikation nicht vorgesehen ist.

Dies wurde sowohl im Rahmen des Normierungs-Prozesses in der Expert-Group[29], als auch an anderen Stellen[30] kritisiert. Es sei sinnvoll, da etliche Dinge der JAX-RS-Spezifikation dafür wiederverwendet werden können. Dieser Meinung stimmt der Autor dieser Arbeit ausdrücklich zu: Die Annotationen für die HTTP-Methoden und die diversen Parameter (@*Param), die Entity-Provider, die Klassen für das Datenformat, für Entity-Tags u. ä. könnten vermutlich direkt übernommen werden. Patrick Müller bspw. führt als Argument für eine Client-API an, dass jemand, der Dienste zur Verfügung stellt, auch einen Client dafür zur Verfügung stellen sollte. Außerdem gäbe es viel mehr Client-Implemetierer als Server-Implementierer. Es ist also fast nötiger eine Client-API zu entwickeln, als eine Server-API. Dies würde die Nutzung von RESTful Web-Services fördern[31].

Andere forderten, dass die Spezifikation für eine Server-API und eine ggf. geplante Client-API explizit getrennt erfolgen solle[32]. Ein Grund gegen die Definition einer Client-API ist auch, dass Aufrufe von RESTful Web-Services dann zu einem Remore Procedure Call degradiert würden.

Unter anderem zur Nutzung in einer Client-API war auch vorgesehen, die Annotationen auf Interfaces zu ermöglichen, welche dann auch von den Clients genutzt werden können. Dieses Detail ist aber nach Meinung des Autors dieser Arbeit nicht immer sinnvoll, z. B. wenn HTML-Seiten erzeugt werden. Auch ist es für den Client in den meisten Fällen egal, in welchem Datenformat das Objekt übertragen wird, solange es die gewünschten Daten enthält. Server-Methoden mit einem Namen wie getAsXML() neben bspw. getAsHtml() sind in einem Interface für einen Client nicht sinnvoll.

Anfang Mai 2008 haben die Spezifikationsleiter dann entschieden, dass es keine Client-API geben wird, u. a. weil der ursprünglich vorgesehene Zeitplan schon nach hinten verschoben werden musste, und dies vermutlich eine weitere Verzögerung verursachen würde[33].

Inzwischen ist sowohl im Rahmen der Referenzimplementierung Jersey als auch beim Projekt RESTEasy eine Client-API entstanden. Dazu siehe auch Abschnitt 7.2.2 auf Seite 110.

---

[28] 1. Spezifikationsvorschlag: [Hadley und Sandoz 2007a]

[29] Client-API gewünscht: z. B. [McDonough 2007a], [Mueller 2007a], [Mueller 2007b], [McDonough 2007b] und [Braun 2007]. Diskussion im Oktober 2007: [JSR311-Expert-Group 2007b], vorl. Ende: [Hadley 2007c]; im März 2008: [JSR311-Expert-Group 2008a], endgültiges Ende der Planung: [Hadley 2008b]

[30] für eine Client-API: z. B. [Mueller 2007c], [Burke 2007a] oder [McDonough 2007c]

[31] [Mueller 2007a], [Mueller 2007b], [McDonough 2007b]

[32] Client- unabhängig von der Server-PI entwickeln: [Harold 2007a].

[33] keine Client-API: z. B. [Hadley 2007c], [Louvel 2008a] und [Hadley 2008b]

3.6.3 Ausgabe direkt in den OutputStream

Im ersten API-Vorschlag war in den Ressource-Methoden ein Zugriff auf den OutputStream möglich, um direkt in diesen zu schreiben. Da dieser Zugriff in einer Ressource-Methode nicht weit von der Netzwerkschicht abstrahiert, wurde diese Möglichkeit wieder aus der API entfernt [34].

Bill Burke stellte dann im März 2008 fest, dass ihm dieser Zugriff in den Ressource-Methoden fehlt [35]. Dafür spricht aus seiner Sicht u. a., dass viele Anwendungsentwickler diese Möglichkeit werden nutzen wollen, weil sie es von anderen APIs – z. B. der Servlet-API – so kennen und gewohnt sind.

Hieraus entwickelte sich eine lange Diskussion, in der verschiedene Vorschläge durchdacht wurden: Es gibt einen speziellen OutputStream, in den eine JAX-RS-Response geschrieben wird, und dann die Daten in den OutputStream geschrieben werden können. Eine andere Idee war, dass der OutputStream vom Response-Objekt abfragbar ist. Diese Ansätze haben alle den Nachteil, dass die Laufzeitumgebung ein zweites Modell für die Behandlung des OutputStreams implementieren muss, denn bei dem Modell der Message-Body-Writer wird der OutputStream in der Ressource-Methode überhaupt nicht angefasst. Im Restlet-Modell wäre dieser Ansatz ohne Pufferung gar nicht umzusetzen, da der OutputStream nur beim Serialisieren der zurückgegebenen Repräsentation zur Verfügung steht, aber nicht während der Ausführung der Geschäftslogik.

Später machte Bill Burke dann den Vorschlag, ein Interface zur Verfügung zu stellen, welches genau eine Methode hat, die als Parameter den OutputStream übergeben bekommt. Eine Ressource-Methode kann einen Instanz dieses Interfaces zurückgeben. In dieser Instanz kann auch auf (als final deklarierten) Variablen aus der Ressource-Methode zugegriffen werden [36]. Diese Variante hat – auch aus Sicht des Autors dieser Arbeit – den Vorteil, dass sie in das Entity-Provider-Modell von JAX-RS (und auch von Restlet) passt, da die Serialisierung erst durch den Provider ausgelöst wird, aber auch auf die in der Ressource-Methode verwendeten Variablen Zugriff hat, so dass nach der Ausgabe der Daten bspw. offene Datenbankverbindungen oder ähnliches geschlossen werden können. Für die Erzeugung von längeren Text-Ausgaben (z. B. als Plain-Text oder HTML) ist dieses Modell ebenfalls sinnvoll, da nicht der komplette Text in einen StringBuilder o. ä. geschrieben werden muss, bis alle für die Entität nötigen Daten aus der Datenbank o. ä. gelesen sind, sondern die Daten direkt ins Netz geschrieben werden können.

Dieser Vorschlag wurde umgesetzt. Dafür wurde das Interface `StreamingOutput` mit der Methode `writeTo(OutputStream)` eingeführt.

[34] Zugriff auf OutputStream entfernt: [Hadley 2008e]
[35] fehlender Zugriff auf den OutputStream: [Burke 2008b]
[36] [Burke 2008c]

3.6.4 Platzierbarkeit von @*Param

Bis Anfang Oktober 2007 waren die @*Param-Annotationen auf Parametern von Konstruktoren und Methoden, aber auch auf Instanzvariablen und Bean-Settern erlaubt. Wenn als Lebenszyklus für Root-Ressource-Klassen ein nicht standardmäßiger Lebenszyklus (z. B. Singelton) verwendet wird, kann dies zu Problemen führen. Deshalb wurde die Verwendung dieser Annotationen auf Methoden- und Konstruktorenparameter beschränkt.

Bill Burke hat im März 2008 vorgeschlagen, dies wieder zu erlauben. Dem stimmt der Autor dieser Arbeit zu[37]. Marc Hadley war dagegen, weil nicht klar ist, was beim Singelton-Lebenszyklus passieren soll. Er schlug vor, dass Laufzeitumgebungen entsprechend eigene Annotationen implementieren könnten[38]. So annotierte Root-Ressource-Klassen wären jedoch nicht portierbar, da verschiedene Laufzeitumgebungen verschiedene eigene Annotationen implementieren müssten.

Gegen das Argument von Marc Hadley hat der Autor dieser Arbeit eingewendet, dass die Spezifikation als Standard-Lebenszyklus jedoch vorsieht, für jede Anfrage eines Clients die Root-Ressource-Klasse neu zu instantiieren[39]. Es ist nach Meinung des Autors dieser Arbeit gut, wenn die Spezifikation andere Lebenszyklen als den Standard berücksichtigt, der Fokus sollte jedoch auf den Standard-Lebenszyklus ausgerichtet sein. Wenn eine Laufzeitumgebung den Standard nicht berücksichtigt, dann müsse sie auch mit den daraus resultierenden Problemen umgehen.

Zum Ende der Diskussion wurden die Annotationen @*Param wieder auf Instanzvariablen und auch auf Bean-Settern erlaubt. In die Spezifikation wurde ein Hinweis aufgenommen, dass Laufzeitumgebungen mit anderen Lebenszyklen die daraus resultierenden Problematiken berücksichtigen müssen.

3.7 Bewertung der JAX-RS-Spezifikation

Da die API ein hohes Abstraktions-Niveau hat, brauchen Anwendungsentwickler nur die wirkliche Programmlogik ihrer Anwendungen zu implementieren. Dies ist nach Meinung des Autors dieser Arbeit sehr gut. Deshalb unterstützt er diese Spezifikation. Trotzdem hat der Autor dieser Arbeit etliche Verbesserungsvorschläge zur JAX-RS-Spezifikation gemacht, von denen viele umgesetzt wurden. Die wichtigsten davon werden im folgenden Abschnitt aufgeführt. Einige wurden auch schon im vorigen Abschnitt genannt.

[37] @*Param auch auf Instanzvariablen: [Burke 2008d], [Koops 2008d]
[38] [Hadley 2008f]
[39] [Koops 2008e]

3.8 Verbesserungsvorschläge des Autors

Im Laufe dieser Arbeit sind dem Autor etliche Punkte aufgefallen, die in der Spezifikation oder der API noch nicht genau definiert waren oder nach Meinung des Autors anderweitig verändert oder hinzugefügt werden sollten. Der Autor dieser Arbeit beschäftigt sich erst seit Ende 2007 mit JAX-RS-Spezifikation. Zu diesem Zeitpunkt waren die wesentlichen Design-Entscheidungen schon gefallen. Viele kleinere Punkte waren zu dem Zeitpunkt aber noch nicht geklärt. Hier hat der Autor dieser Arbeit aktiv Vorschläge eingebracht und die Vorschläge anderer mit diskutiert. Einige dieser Vorschläge werden auf den nächsten Seiten angeführt.

Weitere Meinungen zu einzelnen Punkten der Spezifikation wurden in den vorigen Abschnitten bereits behandelt.

3.8.1 Entität in Sub-Resource-Locatorn

Bis zum 28. 03. 2008 war nicht genau definiert, ob auch ein Sub-Ressource-Locator die Entität der Anfrage von der Laufzeitumgebung anfordern darf. Dies ist zum einen semantisch nicht sinnvoll, da Sub-Ressource-Locator nur die zu verwendende Ressource-Klasse zur Verarbeitung der Anfrage finden sollen. Zum anderen kann es zu Problemen führen, weil einige Entitäts-Datentypen – z. B. `InputStream` – nicht mehrfach lesbar sind. Ein entsprechender Einwand vom Autor dieser Arbeit wurde berücksichtigt und die Spezifikation entsprechend angepasst[40], so dass die Entität nur von Ressource-Methoden und Sub-Ressource-Methoden angefordert werden darf.

3.8.2 HTTP-Kopfzeilen und Exceptions in Message-Body-Writern

Entity-Provider deserialisieren bzw. serialisieren[41] die Körper der HTTP-Nachrichten. Message-Body-Writer dürfen vor dem Schreiben der ersten Daten auch die Kopfzeilen der Anfrage verändern oder eine WebApplicationException werfen. Letzteres hat zur Folge, dass eine ganz andere Antwort verschickt wird, als die bis dahin vorgesehene Antwort

Wie der Name des Interfaces schon sagt, schreiben die Entity-Provider die Nachrichten*körper*, nicht die Kopfzeilen. Nach der Meinung des Autors dieser Arbeit sollen Message-Body-Writer deshalb exakt die Repräsentation serialisieren. Dies deckt sich mit der Definition

[40] Entity in Sub-Ressource-Locatorn? Anfrage: [Koops 2008f], Antwort: [Hadley 2008g]

[41] Da der Server erst die ankommende Anfrage deserialisiert und danach die Antwort serialisiert, wird erst die Deserialisierung und dann die Serialisierung genannt.

von Roy Fielding, dass die Repräsentation der Körper der Nachricht ist, die anderen Daten sind Metadaten der Repräsentation.

Der Autor dieser Arbeit hat deshalb versucht, die Möglichkeit, Kopfzeilen zu schreiben oder eine WebApplicationException zu werfen, wieder aus der Spezifikation entfernen zu lassen. Der Vorschlag wurde nicht übernommen [42].

3.8.3 Sinn der Annotation @Provider

Die JAX-RS-Spezifikation definiert im Abschnitt4 für Provider, dass sie mit der Annotation @Provider annotiert sind. Die Provider werden der Laufzeitumgebung in einem Objekt vom Typ ApplicationConfig zur Verfügung gestellt. Dieser Typ wurde am 19.2.2008 spezifiziert. Deshalb liegt seitdem die Frage nahe, wozu die Annotation noch da ist, denn es wird auch nicht definiert, was passieren soll, wenn sie nicht vorhanden ist.

Eine Nachfrage bei den Leitern der Spezifikation ergab, dass sie dazu da sei, um einen Provider zu erkennen. Das kann jedoch auch an der Implementierung eines der betroffenen Interfaces erkannt werden. Darauf kam die Antwort [43], dass die Annotation dafür da sei, um konkrete Klassen erkennen zu können, die zwar mindestens eines der Interfaces implementieren, aber kein Provider sein sollen. Da die Klasse ApplicationConfig nicht von Anfang an existierte, machte das Argument bis dahin Sinn. Nach Einführung der Klasse ApplicationConfig macht diese Annotation in den Augen des Autors dieser Arbeit keinen Sinn mehr: Wenn eine Klasse in ApplicationConfig übergeben wird, dann kann sie nach Auffassung des Autors auch als Provider aufgefasst werden, ansonsten nicht.

3.8.4 Generischer Typ für die Entity-Provider

Die JAX-RS-Spezifikation verlangt in Abschnitt 4.3.4 u. a. einen Entity-Provider, um JAXB-Objekte (u. a. vom Typ javax.xml.bind.JAXBElement<ObjectType>) zu (de-)serialisieren. Dazu benötigt der JAXB-Marshaller bzw. der Unmarshaller Zugriff auf die gewünschte Klasse des Objektes. Beim Serialisieren ist dies kein Problem, da der Message-Body-Writer den Typ des konkreten Objektes abfragen kann.

Beim Deserialisieren war dies bis 21. 2. 2008 nicht möglich, da der Message-Body-Reader nur den nicht-generischen gewünschten Javatyp der Entität als `Class` zur Verfügung gestellt bekam. Auf den Typparameter der Methode, aus der der Message-Body-Reader dies

[42] Im Message-Body-Writer HTTP-Kopfzeilen ändern / ergänzen und WebApplicationException werfen? Anfrage: [Koops 2008c], Antwort: [Hadley 2008d]

[43] Anfrage bzgl. @Provider: [Koops 2008g]; Antwort: [Hadley 2007b]

hätte abfragen können, hatte er keinen Zugriff. Deshalb hat der Autor dieser Arbeit vorgeschlagen, dem Message-Body-Reader auch den generischen Typ zur Verfügung zu stellen. Dieser Vorschlag wurde umgesetzt [44].

3.8.5 Weitere Vorschläge

Neben den genannten Punkten sind dem Autor dieser Arbeit weitere Kleinigkeiten aufgefallen, insbesondere fehlende Definitionen, was in Fehlersituationen passieren soll, z. B.:

- Bis Mitte März 2008 war nicht definiert, was passieren soll, wenn für die Deserialisierung oder die Serialisierung kein passender Entity-Provider gefunden wurde. Dafür hat der Autor dieser Arbeit vorgeschlagen, den Status 415 bzw 406 zurückzugeben. Dieser Vorschlag wurde umgesetzt [45].

  Später stellte der Autor fest [46], dass diese Reaktion bei einem fehlenden Message-Body-Writer nur bei lesenden Anfragen sinnvoll ist. Bei nicht-sicheren Anfragen (POST, PUT, DELETE, ...) ist dies nach Meinung des Autors nicht sinnvoll, da die Hauptaufgabe mit dem erfolgreichen Verlassen der Ressource-Methode erledigt ist. Deshalb sollte eine von der Ressource-Methode als erfolgreich gekennzeichnete Antwort nicht als Fehler an den Client zurückgegeben werden. Die HTTP-Spezifikation erlaubt explizit, dass Server auch andere Datenformate als die angefragten zurückgeben dürfen. Nach Meinung des Autors dieser Arbeit ist diese Reaktion in dem geschilderten Fall sinnvoll, da die zentrale Aufgabe erfüllt wurde. Dazu könnte bspw. eine kurze Klartext-Entität zurückgegeben werden, die dem Client mitteilt, dass die Operation erfolgreich war, die Antwort jedoch nicht wie gewünscht ausgegeben werden kann. Dieser Vorschlag wurde nicht übernommen.

- Auf Methoden-Parameter und an anderen Stellen können Variablen o. ä. mit @*Param annotiert werden. Dann muss der entsprechende Parameter der Anfrage in den passenden Javatyp konvertiert werden. Bis zum 5. 4. 2008 war nicht definiert, wie auf Fehler reagiert werden soll, die während einer solchen Konvertierung auftreten. Der Autor dieser Arbeit hat daraufhin vorgeschlagen, je nach Annotation den Fehler-Status 400 bzw. 404 zurückzugeben, wenn die Exception nicht eine WebApplicationException mit einem eigenen Status ist. Auch dieser Vorschlag wurde umgesetzt.

- In vielen Methoden war nicht definiert, was pasieren soll, wenn bspw. null übergeben wurde oder gewünschte Werte der Anfrage nicht vom Client angegeben wurden [47]. An

[44] Vorschlag bzgl. generischem Typ: [Koops 2008g]; Antwort: [Sandoz 2008b].
[45] Reaktion auf nicht gefundenen Entity-Provider: Anfrage: [Koops 2008j] Antwort: [Hadley 2008i]; siehe auch Abschnitte 4.3.1 und 4.3.2 der JAX-RS-Spezifikation [Hadley und Sandoz 2008c]
[46] [Koops 2008i]
[47] z. B. [Koops 2008f]

etlichen Stellen war es auch sinnvoll zu definieren, dass zurückgegebene Listen nicht veränderbar sind. Die entsprechenden Vorschläge wurden umgesetzt [48].

3.9 Laufende Implementierungen

Da die JAX-RS-Spezifikation noch nicht fertig ist, kann es auch noch keine fertigen Implementierungen geben. Neben der in dieser Arbeit zu entwerfenden Restlet-Erweiterung wird zur Zeit [49] an drei weiteren voneinander unabhängigen JAX-RS-Implementierung gearbeitet. Diese werden hier ganz kurz aufgeführt:

Jersey Jersey [50] ist die Referenz-Implementierung von Sun. Sie läuft in verschiedenen Containern, z. B. einem Servlet-Container oder dem Grizzly-Server, als JAX-WS-Endpoint und auf dem Java-SE-6-HTTP-Stack.

RESTEasy Das Projekt RESTEasy enthält auch eine Implementierung der JAX-RS-Spezifikation implementiert. Eine kurze Übersicht zu diesem Projekt findet sich im Abschnitt 2.4.3 auf Seite 38.

CXF Das Apache-Projekt CXF erhält z. Zt. ebenfalls eine JAX-RS-Implementierung. Eine kurze Übersicht zum CXF Framework findet sich im Abschnitt 2.4.3 auf Seite 38.

3.10 JAX-RS-Laufzeitumgebung als Restlet-Erweiterung

Restlet ist ein Framework, das RESTful Web-Services mit HTTP sehr gut unterstützt, da viele Standard-HTTP-Kopfzeilen etc. in der API mit einem hohen Abstraktionsniveau zur Verfügung stehen [51]. Deshalb ist es sinnvoll, auch andere APIs mit einem ähnlich hohem Abstraktionsniveau im Restlet-Framework zu unterstützen. So ist die Kombination verschiedener Ressourcen, die zum einen direkt mit der Restlet-API und zum anderen mit JAX-RS-API implementiert sind, leicht möglich.

Im vorigen Abschnitt wurden Implementierungen der JAX-RS-Spezifikation angeführt. Eine Möglichkeit, JAX-RS-Anwendungen auf der Basis des Restlet-Frameworks zu nutzen ist es, eine dieser Implementierungen zu verwenden und dafür einen Adapter zu entwickeln. Dies

[48] Vorschläge z. B. [Koops 2008h], [Koops 2008a], [Koops 2008b], Antworten: [Hadley 2008h], [Hadley 2008c].
[49] 1.4.2008, [Hadley 2008a]
[50] [Jersey 2007]
[51] Die noch nicht unterstützten HTTP-Standard-Kopfzeilen sollen ebenfalls umgesetzt werden: [Louvel 2007e].

würde allerdings zu technischen und organisatorischen Abhängigkeiten von diesem Projekt führen. Dies spricht für eine eigene Implementierung der Spezifkation im Rahmen des Restlet-Projektes. Eine weiteres Argument dafür ist, dass jede weitere Implementierung hilft, die JAX-RS-Spezifikation zu validieren, Probleme und auch weitere Verbesserungsvorschläge zu finden, so dass mehr unabhängiges Feedback zur Verfügung steht. Die Abschnitte 3.6 (ab Seite 57) und 3.8 (ab Seite 61) zeigen einige dieser Ergebnisse.

Die Nutzung von Synergie-Effekten ist ebenfalls möglich: Viele Workarounds bzgl. des Tunnelns sind im Restlet-Framework bereits integriert (siehe Abschnitt 2.4.1 auf Seite 35) und können auch von der JAX-RS-Implementierung genutzt werden. Andererseits können vielleicht einige für die JAX-RS-Implemetierung zu entwickelnde Features auch von anderen Restlet-Anwendern genutzt werden.
Wenn es einfach umsetzbar ist, könnten Restlet-Repräsentation direkt an Ressource-Methoden übergeben werden, ebenso die Antwort als Restlet-Repräsentation von einer Ressource-Methode erzeugt und zurückgegeben werden. Dies betrifft z. B. die JiBX [52]- und Atom-Erweiterungen von Restlet.

3.11 Fazit

Aus der Sicht des Autors dieser Arbeit ist die JAX-RS-Spezifikation mit der dazugehörigen API sinnvoll. Auch eine Implementierung auf der Basis des Restlet-Frameworks ist sinnvoll. Deshalb werden in den nächsten beiden Kapiteln Anforderungen für eine entsprechende Implementierung zusammengestellt und ein Entwurf dafür ausgearbeitet.

[52] JiBX ist – ähnlich wie JAXB – ein Framework zur Konvertierung von XML-Daten zu Java-Objekten, siehe http://jibx.sourceforge.net/.

4 Anforderungsanalyse

In diesem Kapitel werden anhand der JAX-RS-Spezifikation und allgemeiner Betrachtungen funktionale und nicht-funktionale Anforderungen für eine Implementierung der JAX-RS-Spezifikation auf der Basis des Restlet-Frameworks formuliert. Wie diese Anforderungen umgestetzt werden können, wird im nächsten Kapitel ab Seite 79 beschrieben.
Vorteile einer Implementierung der JAX-RS-Spezifikation auf der Basis des Restlet-Frameworks wurden im Abschnitt 3.10 ab Seite 64 angeführt.

4.1 Funktionale Anforderungen

Die zentrale Aufgabe der zu entwickelnden Laufzeitumgebung ist es, die eingehenden HTTP-Anfragen auf die jeweils zu verwendende JAX-RS-Ressource-Methoden zu verteilen, bei Bedarf vorher den Nachrichtenkörper der Anfrage entsprechend den Anforderungen der jeweiligen Ressource-Methode zu konvertieren und nach dem Ausführen der Ressource-Methode das von ihr zurückgegebene Objekt wieder in eine HTTP-Nachricht zu konvertieren.

Weiterhin muss das Extension-Mapping implementiert und die Authentifizierung und die Zugriffskontrolle zur Verfügung gestellt werden.

4.1.1 Deployment von Anwendungen

Sämtliche für eine JAX-RS-Anwendung nötigen Informationen sind in einer Unterklasse der JAX-RS-Klasse `ApplicationConfig` zusammengefasst. Die Laufzeitumgebung bekommt für jede JAX-RS-Anwendung eine Instanz dieses Typs übergeben. Wie dies geschieht, ist in der Spezifikation nicht allgemeingültig beschrieben. Sie liefert zwei Beschreibungen für JAX-WS-Endpunkte und Servlet-Container. Diese sind aber nur verbindlich, wenn die JAX-RS-Laufzeitumgebung als Basis Servlets und / oder JAX-WS-Endpunkte verwendet, was im Fall der hier zu entwerfenden Laufzeitumgebung jedoch nicht der Fall ist. Ansonsten definieren JAX-RS-Laufzeitumgebungen selber, wie sie das Deployment umsetzen. Deshalb brauchen die beiden eben genannten Möglichkeiten für die Implementierung mit dem Restlet-Framework als Basis nicht berücksichtigt zu werden.

4.1.2 Extension-Mapping

Das in Abschnitt 3.2.6 auf Seite 50 beschriebene Extension-Mapping gilt jeweils für ganze JAX-RS-Anwendungen. Der Algorithmus dafür ist in Abschnitt 3.7.1 der JAX-RS-Spezifikaton vorgegeben. Er wird vor der Feststellung der zu verwendenden Methode ausgeführt.

4.1.3 Ressourcen

Zur Feststellung, welche Ressource-Methode für eine bestimmte Anfrage verwendet werden soll, ist in der JAX-RS-Spezifikation ein genauer Algorithmus vorgeschrieben (siehe dort, Abschnitt 3.7.2). Er wählt als erstes die Root-Ressource-Klasse aus. Wenn die JAX-RS-Laufzeitumgebung den Standard-Lebenszyklus verwendet, instantiiert sie demgemäß als nächsten Schritt die Root-Ressource-Klasse. Dafür muss der passende Konstruktor ausgewählt werden (siehe nächste Seite).
Direkt nach der Instantiierung müssen in die mit @Context oder @*Param annotierten Instanzvariablen und Bean-Setter die entsprechenden Objekte injiziert werden. Hier gilt das gleiche wie bei den Parametern von Ressource-Methoden (siehe nächste Seite).

Als nächstes wird geprüft, ob der Anfang des bisher nicht bearbeitete Teiles des URI-Pfades der Anfrage leer ist oder als Pfad auf einer Sub-Ressource-Methode annotiert ist. Dann ist die zu verwendende Ressource-Klasse gefunden und darauf kann die zu verwendende Ressource-Methode gesucht werden. Wenn dies nicht der Fall ist, wird geprüft, ob der Anfang des restlichen Teiles des Anfrage-Pfades auf einem Sub-Ressource-Locator als Pfad annotiert ist. Wenn ja, wird dieser Sub-Ressource-Locator aufgerufen und die zurückgegebene Ressource-Klasse-Instanz rekursiv der gleichen Prüfung unterzogen, bis der jeweils überbleibende Pfad der Anfrage zu einer (Sub-)Ressource-Methode passt.

Als letzter Schritt wird die zu verwendende Ressource-Methode gesucht. Dabei wird neben dem restlichen Teil des Pfades und der angegebenen HTTP-Methode auch das gegebene und / oder zu erzeugende Datenformat berücksichtigt.
Vor dem Aufruf der Ressource-Methode müssen die gewünschten Parameter der Java-Methode aus der Anfrage zur Verfügung gestellte werden. Wie die Parameter dieser Methode behandelt werden, ist auf der nächsten Seite beschrieben.
Die von der Ressource-Methode zurückgegebenen Objekte müssen mit Hilfe eines Entity-Providers in den HTTP-Nachrichten-Körper umgewandelt werden. Weiter Details dazu stehen im Abschnitt 4.1.4 auf Seite 71.

Wenn keine passende Ressource-Methode zur Verfügung steht, dann kann die Anfrage von der Laufzeitumgebung nicht beantwortet werden. Sie wird mit einem entsprechenden Fehlerstatus (404, 405, 406 oder 415) beantwortet. In dem Fall, dass die Ressource existiert und die gewünschte HTTP-Methode für die Ressource implementiert ist, jedoch das gewünschte

Datenformat nicht zur Verfügung steht, ist es praktisch für den Client, wenn die Laufzeitumgebung Links zu den verfügbaren Datenformaten im Nachrichtenkörper der Antwort zurückgibt. Mit der Content-Negotiation über Dateieindungen können solche Links erzeugt werden. Dies soll nach Möglichkeit in der Laufzeitumgebung umgesetzt werden, auch wenn die Spezifikation dies nicht verlangt.

Instantiierung der Root-Ressource-Klasse

Die Laufzeitumgebung muss aus den vorhandenen Konstruktoren der Root-Ressource-Klasse denjenigen mit den meisten Parametern auswählen, auf dem alle Parameter mit erlaubten JAX-RS-Annotationen (@Context und @*Param) annotiert sind. Mit diesem Konstruktor muss die Laufzeitumgebung die Root-Ressource-Klasse instantiieren. Dabei gelten die im Folgenden beschriebenen Regeln für Parameter von Methoden (vgl. Abschnitt 3.2.2 auf Seite 46), mit der Einschränung, dass der Konstruktor nicht die Entität der Anfrage abfragen darf. Die Details sind in Abschnitt 3.1.2 der JAX-RS-Spezifikation beschrieben.

Parameter von Ressource-Meth., Konstruktoren, Bean-Settern und Instanzvariablen

Außer Parametern von Ressource-Methoden und Konstruktoren können auch Instanzvariablen und Bean-Setter mit @Context oder @*Param annotiert sein. In die entsprechenden Bean-Setter und Instanzvariablen müssen nach der Instantiierung durch die Laufzeitumgebung den Annotationen entsprechende Objekte mit den Daten der Anfrage injiziert werden.

Für mit @*Param annotierte Methoden-Parameter, Instanzvariablen oder Bean-Setter müssen die Annotationen @DefaultValue und @Encoded berücksichtigt werden. Die erste dient zur Vorgabe eine Wertes, wenn in der Anfrage kein passender Wert vorhanden ist. Wenn @Encoded auf mit @PathParam, @MatrixParam oder @QueryParam annotierten Parametern, Instanzvariablen oder Bean-Settern gesetzt wurde, dann darf der entsprechende Wert nicht URI-dekodiert werden, ansonsten muss er kodiert bleiben. Bei @CookieParam und @HeaderParam soll stattdessen eine Warnung ausgegeben werden, dass die Annotation @Encoded dort nicht sinnvoll ist.

Wenn die aufgerufene Ressource-Methode verlangt, dass sie die Entität der Anfrage übergeben bekommt, dann muss die Laufzeitumgebung ihr diesen in dem gewünschten Javatyp zur Verfügung stellen. Der Nachrichtenkörper der HTTP-Anfrage muss dazu mit Hilfe eines der Entity-Provider (siehe Abschnitt 4.1.4 auf Seite 71) in eine Instanz des gewünschten Javatyps umgewandelt werden.

Für mit @Context versehene Parameter siehe Abschnitt 4.1.5 ab Seite 72.

Verwendung von Restlet-Repräsentationen als Entität

Die Laufzeitumgebung soll auch Restlet-Repräsentationen sowohl als Entität an Ressource-Methoden übergeben können, als auch von ihnen als Rückgabewert oder Entität in einer JAX-RS-Response an den Client zurückgeben können. So lassen sich die als Repräsentation in Restlet integrierten Bibliotheken direkt nutzen, z. B. für das Atom-Format oder JiBX-Repräsentationen. Auch von Anwendungsentwicklern implementierte Repräsentationsklassen lassen sich dann nutzen.

Behandlung von Fehlern

Ressource-Methoden dürfen alle Exceptions werfen. Wie hiermit umzugehen ist, ist im Abschnitt 3.3.4 der JAX-RS-Spezifikation definiert. Dort steht unter anderem, dass geworfene RuntimeExceptions von der Laufzeitumgebung an den Container weitergereicht werden müssen, damit die dafür im HTTP-Container vorhanden Möglichkeiten verwendet werden können. Da Restlet keine spezielle Behandlung für Exceptions vorsieht (außer den Status 500 zu setzen) kann dies auch direkt in der JAX-RS-Laufzeitumgebung geschehen, da sowieso eine Exception-Behandlung zur Verfügung gestellt werden muss (siehe Abschnitt 4.1.4 auf Seite 71). Diese muss dann für nicht anderweitig verarbeitete Laufzeitfehler den Status 500 an den Client zurück geben.

Die Antwort an den Client

Das von einer Ressource-Methode zurückgegebene Objekt wird normalerweise als Entität der HTTP-Antwort mit dem Status 200 („OK“) verwendet. Das Objekt wird mit einem Message-Body-Writer serialisiert. Wenn eine Ressource-Methode `null` zurückgibt oder den Rückgabetyp `void` hat, dann wird eine Antwort mit dem Status 204 („No Content“) und ohne Entität zurückgegeben.

Als dritte Möglichkeit kann eine Ressource-Methode ein Objekt vom JAX-RS-Typ `Response` zurückgeben (siehe Abschnitt 3.2.3 auf Seite 48). Die Klasse `Response` speichert den Status der Antwort, die HTTP-Kopfzeilen und die zu sendende Entität. Diese muss dann für die Antwort an den Client verwendet werden. Die Entität muss wie direkt zurückgegebene Objekte mit einem Message-Body-Writer serialisiert werden.

Vererbung von Annotationen

Die auszuwertenden Annotationen dürfen auch auf Methoden der Oberklassen oder implementierten Interfaces stehen. Wenn eine Methode nicht mit JAX-RS-Annotationen annotiert ist, dann werden die aus den Oberklassen und Interfaces überschriebenen Methoden entsprechend nach Annotationen durchsucht, bis eine Methode mit Annotationen gefunden wurde. Wenn keine Annotationen gefunden werden, dann ist die Methode keine Ressource-Methode und kein Sub-Ressource-Locator und wird deshalb von der JAX-RS Laufzeitumgebung nicht weiter betrachtet. Wenn eine Methode mit JAX-RS-Annotationen versehen ist, werden Annotationen übergeordneter Klassen oder Interfaces ignoriert.

HEAD und OPTIONS

JAX-RS definiert eine automatische Unterstützung für die HTTP-Methoden HEAD und OPTIONS, wenn für die jeweilige HTTP-Methode keine entsprechende Ressource-Methode implementiert ist:

- Wenn eine Anfrage mit der Methode HEAD an die JAX-RS-Laufzeitumgebung gestellt wird und für die angefragte Ressource (inkl. gegebenem und / oder gewünschten Datenformaten) keine mit HEAD annotierte Ressource-Methode zur Verfügung steht, dann muss die entsprechende mit GET annotierte Methode verwendet werden. Der Nachrichtenkörper der Antwort wird dann verworfen.
- Für eine OPTIONS-Anfrage ohne passende Ressource-Methode muss die Laufzeitumgebung die auf der Ressource-Klasse implementierten HTTP-Methoden aus den vorhandenen Metadaten sammeln und als Antwort an den Client schicken.

4.1.4 Provider

Als Standard-Lebenszyklus für Provider ist vorgesehen, dass sie beim Start der Anwendung jeweils einmal instantiiert und dann beliebig häufig – ggf. auch von mehreren Threads gleichzeitig – verwendet werden. Die Regeln für die Auswahl des Konstruktors zur Instantiierung der Provider sind die gleichen wie bei Root-Ressource-Klassen (siehe Abschnitt 4.1.3 auf Seite 68), außer dass die Annotationen @*Param nicht erlaubt sind.

Da die Annotation @Provider keinerei Funktionalität hat (siehe auch Abschnitt 3.8.3 auf Seite 62), soll nur der Entwickler eine entsprechende Warnung erhalten. Ansonsten soll der Provider verwendet werden, als ob die Annotation vorhanden sei.

Die Aufgaben und Typen der Provider sind im Abschnitt 3.2.5 ab Seite 49 beschrieben. Weitere Details zu den Providern finden sich im Kapitel 4 der JAX-RS-Spezifikation.

Entity-Provider

Bei der Auswahl des zu verwendenden Entity-Providers zur Deserialisierung bzw. Serialisierung der HTTP-Anfrage- bzw. -Antwort-Körper (siehe 3.2.5 auf Seite 49) muss außer auf den Javatyp des zu erzeugenden bzw. zu serialisierenden Objektes auch darauf geachtet werden, dass der Provider auch das gegebene bzw. mindestens eines der gewünschten Datenformate bearbeiten kann. Dies wird auf den Entity-Provider mit @ComsumeMime bzw. @ProduceMime festgelegt.

Jede JAX-RS-Laufzeitumgebung muss einige vorgegebene Entity-Provider zur Verfügung stellen. Sie sind im Abschnitt 4.3.4 der JAX-RS-Spezifikation vorgegeben. Im Rahmen der Implementierung auf der Basis des Restlet-Frameworks ist für das Auslesen von mit POST versendeten HTML-Formularen (Datenformat „application/x-www-form-urlencoded“) neben dem standardmäßig dafür vorgesehenen Provider für das JAX-RS-Interface `MultivaluedMap<String, String>` auch ein Entity-Provider für die Klasse `Form` sinnvoll, welche in der Restlet-API zum Auslesen des HTML-Formulares dient. Ein entsprechender Provider soll deshalb ebenfalls implementiert werden.

Exception-Mapper

Von Sub-Ressource-Locatorn, Ressource-Methoden und Entity-Providern geworfene Exceptions müssen mit den vorhandenen Exception-Mappern in eine JAX-RS-`Response` umgewandelt werden. Dafür müssen beim Auftreten einer Exception die vorhandenen Exception-

Mapper nach dem zu verwendenden Exception-Mapper durchsucht werden. Das gleiche gilt für während der Konvertierung zu den gewünschten Javatypen von mit @*Param annotierte Bean-Setter, Instanzvariablen oder Parameter auftretene WebApplicationExceptions.

4.1.5 Kontext

Root-Ressource-Klassen, Sub-Ressource-Locator, Ressource-Methoden und Provider können verlangen, dass die Laufzeitumgebung bestimmte Informationen aus der Umgebung zur Verfügung stellt. Dazu können sie Instanzvariablen, Bean-Setter und ggf. Methoden-Parameter (bei Sub-Ressource-Locatorn und Ressource-Methoden) mit vorgegebenen Typen mit der Annotation @Context versehen, ähnlich wie bei @*Param auch.

Wenn ein Parameter mit @Context annotiert ist, dann muss der Typ des Parameters einer der folgenden Interfaces sein:

- `SecurityContext`: Wie der Name des Benutzers aus der Anfrage gelesen wird, ist in der Spezifikation nicht festgelegt, sondern wird der Laufzeitumgebung überlassen. Das gleiche gilt für die Prüfung, ob der Benutzer eine bestimmte Rolle hat.
  Für die Authentifizierung soll der Standard-Mechanismus in Restlet verwendet werden (`Guard`, siehe Seite 33). Einige Server-Connectoren (z. B. Servlet-Container, siehe Abschnitt 2.4.1 auf Seite 36) bieten ebenfalls eine Authentifizierung an, worauf die Restlet-API bisher keinen Zugriff bietet. Sie sollen – wenn möglich – genutzt werden können.

  Im Restlet-Framework ist bisher keine Rollenverwaltung integriert. Deshalb muss die Einbindung einer Rollenverwaltung ermöglicht werden. Wenn später in das Restlet-Framework eine Rollenverwaltung integriert wird, dann soll diese ebenfalls verwendet werden können. Auch Rollenverwaltungen der HTTP-Server-Connector sollen genutzt werden können.

  Weiterhin lässt sich das Authentifikationsverfahren abfragen und feststellen, ob die Anfrage verschlüsselt ist.

- Für die Interfaces `UriInfo`, `HttpHeaders`, `Request`, `MessageBodyWorkers` und `ContextResolver` sind die Funktionalitäten im Abschnitt 3.2.2 auf Seite 46 beschrieben. Sie müssen entsprechend umgesetzt werden.

Sämtliche so injizierte / übergebene Objekte müssen von mehreren Threads parallel angesprochen werden können, und die zu der vom jeweiligen Thread bearbeiteten HTTP-Anfrage passenden Werte zurückgeben. Weitere Details finden sich im Abschnitt 5 der JAX-RS-Spezifikation.

Die JAX-RS-Spezifkation erlaubt es anderen Laufzeitumgebungen, ebenfalls Javatypen zu definieren, die mit @Context annotiert werden dürfen. Analog zu den Javatypen `HttpServletRequest` und `HttpServletResponse` der Servlet-Spezifikation stehen im Restlet-Framework bei jeder Anfrage Objekte der Typen `Request` und `Response` zur Verfügung. Da in mit @Context annotierte Instanzvariablen etc. verschiedene Threads Informationen zur ihrer jeweiligen Anfrage abrufen können müssen, müssten entsprechend für die jeweiligen Restlet-Typen entsprechende Proxies implementiert werden. Da `Request` und `Response` Klassen sind, ist dies aufwendiger als bei Interfaces. Weiterhin zeigt eine (aus Platzgründen nicht angeführte) Untersuchung, dass dem Aufwand nur ein geringer Mehrwert gegenüberstehen würde. Deshalb werden in diesem Zusammenhang keine zusätzlichen Anforderunen definiert.

4.1.6 Laufzeitumgebung

Im Kapitel 6 der JAX-RS-Spezifikation ist beschrieben, welche Anforderungen zusätzlich an Implementierungen für auf der Servlet-API basierende Container gestellt werden. Da diese Laufzeitumgebung nicht direkt auf der Servlet-API basiert, ist in diesem Zusammenhang nichts zu berücksichtigen.

Das Kapitel beschreibt weiterhin zusätzliche Anforderungen für die Spezifikation implementierende Java-EE-Container. Da die Restlet-API auch keinen Java-EE-Container bereit stellt, müssen diese Anforderungen ebenfalls nicht berücksichtigt werden.

4.1.7 Runtime Delegate

Entwickler einer JAX-RS-Laufzeitumgebung müssen eine konkrete Unterklasse der Klasse `RuntimeDelegate` zur Verfügung stellen. Sie dient als Factory für einige von der JAX-RS-Laufzeitumgebung zu implementierenden Klassen:

- Ein `ResponseBuilder` dient zur Erzeugung einer JAX-RS-`Response`. Die den verschiedenen Methoden des ResponseBuilders übergebenen Werte müssen beim Erzeugen einer Response dieser übergeben werden. Die Laufzeitumgebung muss die abstrakte JAX-RS-Klasse `Response` ebenfalls implementieren.
- Der `UriBuilder` muss die übergebenen Werte kodieren, wie es in der URI-Spezifikation [1] vorgesehen ist. Welche Zeichen zu kodieren sind, hängt davon ab, an welcher Stelle im URI (Host, Pfad, Query, etc.) der Wert eingebaut werden soll. Die automatische Kodierung kann vom Anwendungsentwickler im Einzelfall abgeschaltet werden; dann muss der UriBuilder überprüfen, ob übergebene Werte ungültige Zeichen enthalten und den Aufruf ggf. zurückweisen. Für die Erzeugung einer Instanz der Klasse `java.net.URI` kann der UriBuilder die vorhandene Restlet-Klasse `Template` verwenden, um die Template-Variablen zu füllen.
- Die Klasse `VariantListBuilder` muss entsprechend der Dokumentation in dieser Klasse implementiert werden.
- `HeaderDelegate`s müssen für die Klassen implementiert werden, die als Anfrage- oder Antwortkopfzeile deserialisiert und / oder serialisiert werden müssen. Sie müssen entsprechende Strings deserialisieren und ebenso ein Objekt der jeweiligen Klasse zu einem String gemäß der HTTP- bzw. Cookie-Spezifikation [2] serialisieren.

Da die zu entwerfende JAX-RS-Laufzeitumgebung keine Endpunkte (JAX-WS o. ä.) unterstützt, kann die Methode zur Erzeugung derselben eine UnsupportedOperationException werfen.

Eine JAX-RS-Laufzeitumgebung muss die konkrete Implementierung der Klasse `RuntimeDelegate` entweder über eine Klassenmethode in die Klasse RuntimeDelegate injizieren oder den Klassennamen über eine von mehreren Konfigurationsmöglichkeiten zur Verfügung stellen. Die weiteren Details stehen im Abschnitt 7.1 der JAX-RS-Spezifikation.

[1] Syntax des Uniform Resource Identifier: [RFC-3986 2005]
[2] HTTP-Spezifikation: [RFC-2616 1999]; Cookie-Spezifikation: [RFC-2965 2000].

4.2 Nicht-funktionale Anforderungen

Außer den funktionalen Anforderungen sollte Software noch weiteren Ansprüchen genügen. Im Folgenden werden zu berücksichtigende Kriterien genannt, angelehnt an [Kahlbrandt 2001, Kap. 3]. Die Punkte, die in diesem Buch als „aus Benutzersicht“ gekennzeichnet sind, gelten bei der Entwicklung eines Frameworks oder einer Laufzeitumgebung für *Anwendungsentwickler*, welche die Spezifikation nutzen, um eigene Anwendungen zu entwickeln. Die Punkte, die im Buch als „aus Entwicklersicht“ gekennzeichnet sind, gelten für die *Entwickler einer Laufzeitumgebung*.

Wichtig ist aus der Sicht eines Anwendungsentwicklers:

- Zuverlässigeit / Robustheit (Abschnitt 4.2.1)
- Benutzbarkeit (Abschnitt 4.2.2 auf der nächsten Seite)
- Sicherheit (Abschnitt 4.2.3 auf der nächsten Seite)
- Performance / Effizienz (Abschnitt 4.2.4 auf Seite 77)
- Skalierbarkeit (Abschnitt 4.2.5 auf Seite 77)

Aus der Sicht eines Entwicklers einer Laufzeitumgebung sind folgende Punkte wichtig:

- Wartbarkeit und Erweiterbarkeit (Abschnitt 4.2.6 auf Seite 77)
- Portierbarkeit (Abschnitt 4.2.7 auf Seite 78)
- Testbarkeit (Abschnitt 4.2.8 auf Seite 78)

4.2.1 Zuverlässigkeit / Robustheit

Zuverlässigkeit bedeutet, dass eine entwickelte Software – im Falle dieser Arbeit ein Softwarebaustein, die Laufzeitumgebung – nicht unerwartet aufhört zu funktionieren und auch bei unerwarteten Ereignissen (z. B. fehlerhaften oder fehlenden Eingaben) weiterhin funktioniert. Die Laufzeitumgebung soll dann, wenn irgend möglich, nicht komplett abstürzen, sondern den Fehler sinnvoll und verständlich melden und so gut wie möglich weiterhin ihren Dienst erfüllen.

Allerdings sollte eine Software nicht zu viele Fehler tollerieren, damit Benutzer ihr Fehlverhalten auch merken und korrigieren können. Das bedeutet übertragen auf die Laufzeitumgebung, dass Unstimmigkeiten – wenn möglich und sinnvoll – hingenommen werden, und der Anwendungsentwickler zusätzlich für ihn aufschlussreiche Informationen bekommt.

4.2.2 Benutzbarkeit

Jegliche Software sollte möglichst einfach zu benutzen sein. Das gilt auch für die zu entwickelnde Laufzeitumgebung.

Die JAX-RS-API wird im Rahmen des Java Community Process festgelegt. An der Schnittstelle zur Anwendungsentwicklung kann deshalb im Rahmen dieser Umsetzung nichts verändert werden. Siehe jedoch die Abschnitte 3.6 und 3.8 ab Seite 57.

Die Schnittstelle zur Restlet-API liegt in der Verantwortlichkeit dieser Arbeit. Diese Schnittstelle sollte möglichst einfach sein, und den Anwendungsentwicklern möglichst viel Arbeit abzunehmen. Es ist auch darauf zu achten, dass erfahrene Restlet-Entwickler sich schnell darin orientieren können, d. h. sie sollte – wenn möglich und sinnvoll – eine vergleichbare Schnittstelle haben wie andere Restlet-Erweiterungen. Natürlich soll die Laufzeitumgebung auch gut dokumentiert werden.

4.2.3 Sicherheit

In Anlehnung an [Kahlbrandt 2001] ist ein System sicher, wenn es unter vorgegebenen Bedingungen keine unzulässigen Ereignisse zulässt und in keinen unzulässigen Zustand wechseln kann. Dabei ist zwischen Safety und Security zu unterscheiden (beide Worte werden mit „Sicherheit“ übersetzt):

- *Safety* bezeichnet die Datenintegrität, die auch bei Systemfehlern gewährleistet sein muss. – Da die zu entwerfende Laufzeitumgebung weder Daten selber fest speichert, noch für die Speicherung von Anwendungsdaten zuständig ist, braucht dieser Punkt nicht betrachtet zu werden. Die zur Laufzeit nötigen Daten können jederzeit aus den Java-Klassen neu gelesen werden. Für die Daten der Anwendung ist die Anwendung selber oder deren Persistenzschicht verantwortlich.
- *Security* bezeichnet den Schutz vor unbefugter Nutzung und anderem Missbrauch. – Die Laufzeitumgebung selber hält keine Daten, die zu schützen wären. Der grundlegende Schutz vor Angriffen aus dem Internet ist im Betriebssystem auf TCP/IP-Ebene (z. B. als Firewall) oder im eigentlichen Web-Server / Web-Container besser aufgehoben, weil sie dort generisch für viel mehr Anwendungsfälle implementiert werden kann. Um Anwendungsentwickler bei der Implementierung von Sicherheitsmechanismen zur Verhinderung von unbefugter Benutzung für ihre Anwendung zu unterstützen, definiert die JAX-RS-Spezifikation das Interface `SecurityContext`, welches Informationen speichert, ob die Anfrage authentifiziert wurde, wer der anfragende Benutzer ist u. ä. Dieser Punkt wurde bei den funktionalen Anforderungen behandelt, siehe Abschnitt 4.1.5 auf Seite 72.

4.2.4 Performance / Effizienz

Damit die Endanwender nicht lange auf Antworten warten müssen, und um die Hardware-Ressourcen des Servers zu schonen, soll die Laufzeitumgebung möglichst sparsam mit ihnen umgehen. Dabei muss ein sinnvoller Kompromiss zwischen Optimierung auf wenig Arbeitsspeicher auf der einen Seite und wenig benötigter CPU-Leistung auf der anderen Seite gefunden werden. Eine Definition dieses „sinnvoll“ ist sehr von den konkreten Anforderungen abhängig und erfolgt deshalb hier nicht. Wenn eine Geschwindigkeitssteigerung nur relativ wenig zusätzlichen Speicherplatz benötigt, dann ist es vernünftig, sie umzusetzen. Für Objekte, die während der Laufzeit der Laufzeitumgebung nur einmal instantiiert und dann häufig abgefragt werden, ist dies sinnvoll. Wenn ein Objekt für jede Anfrage neu erzeugt wird, lohnt es sich nicht so sehr, viel Speicherplatz für insgesamt wenig Zeitgewinn zu opfern, speziell, wenn die gecachten Werte vermutlich nicht noch einmal abgerufen werden. Dieser Punkt muss im Einzelfall im Entwurf oder bei der Implementierung geklärt werden.

4.2.5 Skalierbarkeit

Da die zu entwickelnde Laufzeitumgebung Anwendungsentwicklern zur Implementierung von Server-Anwendungen dient, ist darauf zu achten, dass sie gut skaliert, denn Server werden häufig von vielen Clients gleichzeitig genutzt. Anwendungen, die im REST-Architekturstil implementiert sind, skalieren gut (siehe Abschnitt 2.1.3 auf Seite 14), wenn die in der REST-Architektur vorgesehenen Konzepte im Entwurf und bei der Implementierung der Laufzeitumgebung wirklich korrekt umgesetzt werden.

4.2.6 Wartbarkeit und Erweiterbarkeit

Dieser Punkt beinhaltet, dass Fehler im Quellcode schnell lokalisiert und auch mit wenig Aufwand behoben werden können. Außerdem soll die Anwendung so modelliert sein, dass eine Erweiterung leicht möglich ist. Wesentliche Punkte dazu sind:

- vollständige Dokumentation, inklusive sinnvoll und ausreichend kommentiertem Quellcode,
- aussagekräftige Fehlermeldungen,
- Speicherung von Fehlermeldungen, wenn dies zur Behebung von Fehlern hilfreich ist,
- Modularisierung der Anwendung.

Deshalb sollen die einzelnen Klassen, Dateien und Methoden nicht zu groß / lang, gut dokumentiert, die Verantwortlichkeiten klar getrennt und Fehlermeldungen möglichst aussagekräftig sein. Außerdem sollen Variablen, die typischerweise gleiche oder gleichartige Objekte enthalten in verschiedenen Methoden und Klassen sprechende und möglichst gleiche oder zumindest ähnliche Namen bekommen.

Da diese Arbeit etliche Monate vor der endgültigen Version der JAX-RS-Spezifikation begonnen wurde, ist darauf zu achten, dass gegebenenfalls Änderungen der Spezifikation im schon implementierten Quellcode leicht umgesetzt werden können. Deshalb sollen z. B. die einzelnen in der Spezifikation vorgegebenen Algorithmen nicht auseinandergerissen werden, sondern jeweils zusammenhängend implementiert werden, um Änderungen leicht umsetzen zu können.

4.2.7 Portierbarkeit

Portierbarkeit bedeutet, dass eine Anwendung mit möglichst wenig Aufwand auf eine andere Hardware- und / oder Softwareumgebung übertragen werden kann. Für die Nutzer der Laufzeitumgebung ist dies automatisch gegeben, da sie relativ einfach die Laufzeitumgebung wechseln können. Dies setzt vorraus, dass die Spezifikation eingehalten wurde, sowohl vom Anwendungsentwickler als auch von den verschiedenen Entwicklern der Laufzeitumgebungen.

Da die JAX-RS-Spezifikation ausschließlich Java-Klassen verwendet, ist eine grundsätzliche Übertragbarkeit auch auf andere Betriebssysteme und Hardwareumgebungen gegeben, da virtuelle Maschinen für Java für viele Betriebssysteme und Hardwareumgebungen existieren.

Die JAX-RS-Spezifikation nutzt die in Java 1.5 eingeführten Annotationen. Deshalb ist es nicht möglich und in der Spezifikation explizit nicht vorgesehen, dass die Anwendungen auf älteren Java-Versionen funktionieren. Diese Einschränkung liegt außerhalb des Rahmens dieser Arbeit und wird deshalb nicht weiter beachtet.

4.2.8 Testbarkeit

Ein weiteres Kriterium für gute Software ist gute Testbarkeit. Da der zu entwickelnde Software-Baustein exakte und auch leicht überprüfbare Ergebnisse erwarten lässt (im Gegensatz zu bspw. GUI-Anwendungen), lassen sich Tests leicht realisieren. Deshalb sollen möglichst einfach ausführbare und reproduzierbare Tests entwickelt werden, die einen möglichst großen Teil der Funktionalität abdecken.

5 Entwurf

In diesem Kapitel wird gezeigt, wie die im vorigen Kapitel gesammelten Anforderungen auf das Restlet-Framework abgebildet werden können. Die JAX-RS-Laufzeitumgebung wird als Erweiterung der Restlet-API entworfen. Da die Entwicklung dieser JAX-RS-Laufzeitumgebung durch das Restlet-Team unterstützt wird, ist es möglich, interne Funktionalitäten der Restlet-Engine zu nutzen und dafür auch die Schnittstellenklasse zur Restlet-Engine (`Engine`) – nach Absprache – zu erweitern.
Details der Restlet-Architektur sind im Abschnitt 2.4.1 ab Seite 31 beschrieben.

Die im Abschnitt 4.2 ab Seite 75 gesammelten nicht-funktionalen Anforderungen werden in diesem Kapitel nicht alle explizit angeführt. Sie werden nur genannt, wenn eine Design-Entscheidung aufgrund einer dieser Anfoderungen gefällt wurde.

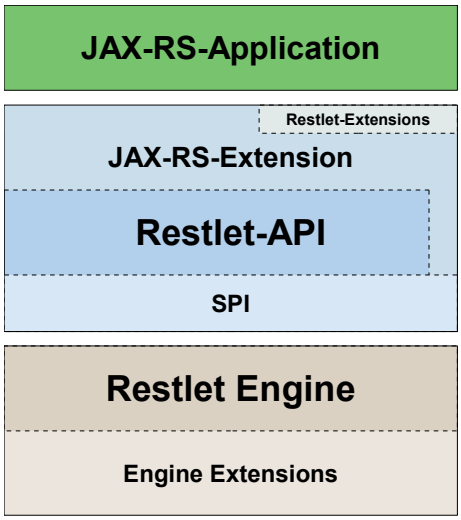

Abbildung 5.1: Die JAX-RS-Erweiterung innerhalb der Restlet-Architektur
(eigene Grafik, vgl. auch Abb. 2.6 auf Seite 32)

5.1 JAX-RS-Restlet

Die zentralen Aufgaben der Laufzeitumgebung sind kurz gefasst folgende:

1. Für jede von einem Client eingehende HTTP-Anfrage muss die zu verwendende Ressource-Methode festgestellt werden.
2. Die ausgewählte Ressource-Methode muss ausgeführt werden. Dazu muss der vom Client empfangene Nachrichtenkörper bei Bedarf nach den Wünschen der Ressource-Methode konvertiert werden und andere Parameter der HTTP-Anfrage zur Verfügung gestellt werden.
3. Nach der Ausführung der Ressource-Methode muss der Rückgabe-Wert entsprechend der JAX-RS-Spezifikation zu einem Nachrichtenkörper konvertiert und an den Client geschickt werden.

Für die Aufgaben der Vor- und Nachbereitung der Anfrage bzw. Antwort wird die Klasse `JaxRsRestlet` vorgesehen [1]. Sie wird die zentrale Klasse dieser Laufzeitumgebung.
Um die Klasse `JaxRsRestlet` übersichtlich zu gestalten (siehe Abschnitt 4.2.6 auf Seite 77), sollen die Funktionalitäten, die nicht direkt mit diesen beiden Aufgaben zu tun haben, in andere Klassen (ggf. Utility-Klassen) ausgelagert werden.

5.2 Ressource-Klassen

Jedes JaxRsRestlet speichert die ihr übergebenen Root-Ressource-Klassen. Die folgende Abbildung zeigt die Klasse `JaxRsRestlet` mit der für die Speicherung der Root-Ressource-Klassen benötigte den Methode und der speichernden Collection. Für eine vollständige Übersicht siehe Abbildung 5.7 auf Seite 96. Sequenzdiagramme über den Ablauf des Starts der Anwendung und der Verarbeitung der Anfragen finden sich auf Seite 102 und 103.

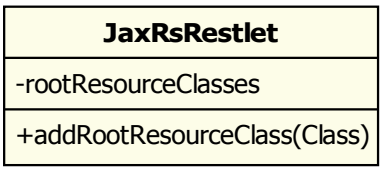

Abbildung 5.2: JaxRsRestlet mit Root-Ressource-Klassen (eigene Grafik)

[1] Im Entwurf werden Klassennamen vorgeschlagen, um die entworfenen Klassen später leicht referenzieren zu können.

5.2.1 Wrapper-Klassen

Viele Vorgaben für die Laufzeitumgebung werden mit Hilfe von Annotationen an Java-Methoden und anderen Stellen im Quellcode der Ressource-Klassen definiert (siehe auch Abschnitt 3.2 auf Seite 44). Um mit sprechenden Getter-Methoden darauf zugreifen zu können, werden entsprechende Wrapper-Klassen vorgesehen:

- `RootResourceClass` für Root-Ressource-Klassen,
- `ResourceClass` für Ressource-Klassen,
- `ResourceMethod` für Ressource-Methoden und Sub-Ressource-Methoden,
- `SubResourceLocator` für Sub-Ressource-Locatorn und

Diese Wrapper bearbeiten von ihrer Semantik her Anfragen. Deshalb ist es naheliegend, sie als Unterklassen der Klasse Restlet zu entwerfen. Sie würden Anfragen entweder weiterleiten (Root-Ressource-Klassen, Ressource-Klassen und Sub-Ressource-Locatorn) oder selber beantworten (Ressource-Methoden und Sub-Ressource-Methoden). Wenn diese Wrapper-Klassen die HTTP-Anfragen selber beantworten würden, müsste der Algorithmus jedoch auf die verschiedenen Klassen verteilt werden, was jedoch nicht gewünscht ist (siehe Abschnitt 4.2.6 ab Seite 77). Da diese Klassen speziell für diese JAX-RS-Implementierung konzipiert werden, lassen sie sich vermutlich in anderen Anwendungsfällen nicht verwenden. Deshalb ist es kein Vorteil, wenn sie als Unterklassen von Restlet implementiert werden. Aus den beiden genannten Gründen halten die Instanzen der genannten Klassen nur die jeweils gewrappten Objekte, ohne eigene Funktionalität zur Verfügung zu stellen.

Die Klasse `RootResourceClass` ist auch für den Lebenszyklus der Root-Ressource-Klasse zuständig, d. h. sie muss entsprechend des Standard-Lebenszyklusses für Root-Ressource-Klassen für jede Anfrage die Klasse instantiieren. Andere Lebenszyklen können in einer späteren Version der Laufzeitumgebung hier ebenfalls implementiert werden.

Da sowohl `ResourceMethod` als auch `SubResourceLocator` einige gemeinsame Funktionalitäten haben (z. B. den Aufruf der Java-Methode und das Laden der dazugehörigen Parameter oder im Fall von Sub-Ressource-Methoden den mit @Path annotierte Pfad auf der Methode), wird hierfür eine gemeinsame Oberklasse vorgesehen. Da eine Root-Ressource-Klasse alles kann, was auch eine normale Ressource-Klasse kann (im wesentlichen die Speicherung von Ressource-Methoden und Sub-Ressource-Locatorn), erbt die Klasse `RootResourceClass` von der Klasse `ResourceClass`. Zusätzlich hat die Root-Ressource-Klasse noch eine @Pfad-Annotation und wird von der Laufzeitumgebung instantiert. Um die @Path-Verarbeitung nur einmal zu implementieren, wird für die Klasse `ResourceClass` als Oberklasse der Klasse `RootResourceClass` und der gemeinsamen Oberklasse der beiden Methoden-Wrappern ebenfalls eine gemeinsame Oberklasse vorgesehen, wobei der @Pfad-Zugriff in der `ResourceClass` nicht öffentlich ist.

Neben den genannten Wrappern mit direkten Entsprechungen in der JAX-RS-Spezifikation werden weitere Wrapper vorgesehen:

- `ResourceObject` für von Sub-Ressource-Locatorn zurückgegebenen Instanzen einer Ressource-Klasse, welche Zugriff auf den Ressource-Klasse-Wrapper bietet und
- `ParameterList` für die Parameter von Konstruktoren von Root-Ressource-Klassen, von Sub-Ressource-Locatorn und Ressource-Methoden.

Die Wrapper für Ressource-Methoden und Sub-Ressource-Locatorn enthalten jeweils eine Parameterliste mit den benötigten Parameterzugriffen. Die `RootResourceClass` erhält dementsprechend eine Parameterliste für den zu verwendenden Konstruktor.

Um innerhalb der Laufzeitumgebung Zeit zu sparen, könnten reflektive Aufrufe von Konstruktoren und Methoden durch Aufrufe von dynamischen Proxies ersetzt werden. Für die Generierung dieser Proxies wird entweder ein Compiler (und damit zur Laufzeit ein Java-JDK / -SDK) oder eine zusätzliche Bibliothek benötigt. Da es nicht das Ziel der zu entwerfenden Laufzeitumgebung ist, höchstmöglich performant zu sein, wird die Generierung von Proxies wegen des zusätzlichen Aufwandes nicht weiter betrachtet. Eine spätere Optimierung ist möglich, da die Objekt-Erzeugungen und Methodenaufrufe alle in den Wrappern gekapselt sind, also relativ leicht ausgetauscht werden können.

Parameter-Liste und Injektion in Instanzvariablen und Bean-Setter

Um nicht bei jeder Anfrage die Liste der Parameter einer Java-Methode oder eines Konstruktors mit der Reflection-API neu feststellen zu müssen, soll dies einmal beim Erzeugen des Wrapper-Objektes (`RootResourceClass`, `ResourceMethod` bzw. `SubResourceLocator`) erledigt und die entsprechenden Ergebisse dann gecacht werden. Dann können die jeweils benötigten Objekte schnell aus der HTTP-Anfrage ausgelesen werden. Hierfür wird ein Interface `ParamGetter` vorgesehen, dessen einzige Methode `getValue()` den jeweils zu injizierenden Wert zurück gibt. Da das Caching für jeden Parameter, Instanzvariable etc. nur einmal pro instantiierter Laufzeitumgebung benötigt wird, wird nur wenig zusätzlicher Speicherplatz benötigt, der bei jeder Anfrage eines Clients Zeit spart. Für die verschiedenen möglichen Parameter-Typen (die verschiedenen mit @Context oder @*Param annotierten Parameter oder die Entität der Anfrage, siehe Abschnitt 5.3.1 ab Seite 87) werden jeweils spezielle Klassen vorgesehen. Dieser Entwurf hat viele kleine Klassen zur Folge, welche dafür übersichtlich sind. Außerdem brauchen nicht bei jedem Aufruf der Ressource-Methode oder des Sub-Ressource-Locators viele Bedingungen bzgl. des zu verwendenden Parameters geprüft zu werden.
Einige dieser Klassen werden auch verwendet, um die Objekte für entsprechend annotierte Instanzvariablen oder Bean-Setter festzustellen.

5.2.2 Identifizierung der zu verwendenden Java-Methode

Der Algorithmus zur Identifizierung der zu verwendenden Ressource-Methode wird wie in der JAX-RS-Spezifikation vorgegeben [2] implementiert. Die einzelnen Teile dieses Alogrithmusses werden in der Implementierung als einzelne Methoden des JaxRsRestlets implementiert. Die Namen der Methoden entsprechen den Überschriften des jeweiligen Teils des Algorithmusses. Dies erleichtert die Übersicht innerhalb dieser Klasse (siehe Abschnitt 4.2.6 auf Seite 77).

Während der Abarbeitung des Algorithmusses müssen die für das Interface `UriInfo` benötigten Informationen (Pfad- und Matrix-Parameter aus der Anfrage-URI, die bisher durchlaufenen Instanzen der Ressource-Klassen und die dazugehörigen Pfad-Segmente) gesammelt und festgehalten werden (siehe dazu Abschnitt 5.4.1 ab Seite 91).

Wenn zu einer Ressource keine zu den gewünschten Datenformaten passende Ressource-Methode vorhanden ist, soll laut JAX-RS-Spezifikation eine WebApplicationException geworfen werden. In der Anforderungsanalyse wurde definiert, dass die Antwort an den Client ergänzend zur JAX-RS-Spezifikation Links zum Abruf der vorhandenen Repräsentationen enthalten soll (siehe Abschnitt 4.1.3 ab Seite 67). Die in einem solchen Fall zu werfende Exception soll von der Klasse `WebApplicationException` erben und Informationen über mögliche Datenformate als Attribut enthalten. Diese stehen in den Wrappern der Ressource-Methoden zur Verfügung. Die Links sollen dann vom Exception-Mapper für die `WebApplicationException` entsprechend umgewandelt werden.
Anwendungsentwickler dürfen eigene Exception-Mapper für die WebApplicationException implementieren. Ein spezieller Exception-Mapper für Unterklassen der Klasse WebApplicationException würde vom Anwendungsentwickler implementierte Exception-Mapper für die jeweilige(n) Exception-Unterklasse(n) blockieren. Deshalb muss die Verarbeitung dieser Unterklassen im Exception-Mapper für die WebApplicationException mit Fallunterscheidung nach der jeweiligen Exception-Klasse durchgeführt werden.

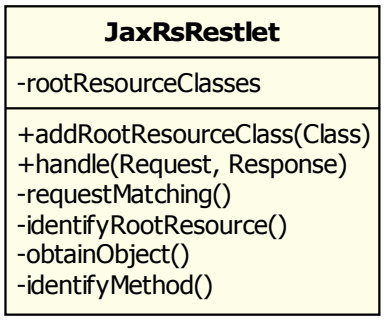

Abbildung 5.3: JaxRsRestlet, Identifizierung der Ressource-Methode
(eigene Grafik; nicht alle Methoden-Parameter angeführt)

[2] JAX-RS-Spezifikation: [Hadley und Sandoz 2008a, Abschnitt 3.7.2]

URI-Templates

Zum Feststellen der Ressource-Methode müssen an mehreren Stellen (Root-Ressource-Klassen, Sub-Ressource-Locatorn und Sub-Ressource-Methoden) Pfade aus Java-Annotation mit Pfaden aus der HTTP-Anfrage verglichen werden. Während dieses Matchings muss jede gefundene Variable gespeichert werden, um mit @PathParam abgefragt werden zu können. Die Variablen-Werte können in dem im Abschnitt 5.4.1 auf Seite 91 entworfenen CallContext gespeichert werden. – Für das Matching steht im Restlet-Framework die Utility-Klasse `Template` zur Verfügung. Sie bietet allerdings nicht die volle Unterstützung für die benötigten Vergleiche, denn sie unterstützt nicht das Speichern ganzer Pfadsegmente (inklusive den Matrix-Parametern), sondern nur das Speichern des gefundenen Variablenwertes, welche ggf. nur ein Teil des Pfad-Segmentes ist.

Details dieses Matching werden z. Zt.– nach der Veröffentlichung des Public Review Draftes der JAX-RS-Spezifikation – neu diskutiert[3]. Deshalb stehen die endgültige Anforderungen noch nicht fest. Darum soll der Einfachheit halber vorerst die in der Restlet-API vorhandene Klasse `Template` verwendet werden. Die Konsequenz ist, dass es vorerst nicht möglich sein wird, mit @PathParam ein `PathSegment` oder eine `List<PathSegment>` anzufordern.
Wenn die genauen Anforderungen in einer späteren JAX-RS-Version feststehen, können auch die endgültigen Anforderungen des Matchings inklusive der Speicherung der Pfadsegmente gesammelt werden. Um ein einfaches Austauschen des Matchings zu ermöglichen, soll dies in einer eigenen Klasse gekapselt werden, welche am Anfang das Matching an die Restlet-Klasse `Template` deligiert. In die wrappende Klasse kann später das Matching direkt integriert werden, ohne dass an anderen Stellen Code-Änderungen nötig sind.

5.2.3 Verarbeitung des Ergebnisses

Nach der Ausführung der Ressource-Methode kann diese eine JAX-RS-Response oder ein anderes Objekt zurückgeben. Wenn eine JAX-RS-Response zurückgegeben wurde und darin das Datenformat vorgegeben ist, muss dies direkt für die Antwort verwendet werden. Ansonsten muss es anhand der von der ausgeführten Ressource-Methode erzeugbaren Datenformaten, der den gegebenen Javatyp unterstützenden Message-Body-Writer und den Wünschen des Clients ausgewählt werden. Der genaue Algorithmus ist in Abschnitt 3.8 der JAX-RS-Spezifikation beschrieben. Er soll in einer eigenen Methode der Klasse JaxRsRestlet implementiert werden.

Die Umwandlung des zurückgegebenen Objektes bzw. der Entität der JAX-RS-Response wird im Abschnitt 5.3.1 ab Seite 88 beschrieben.

[3] Diskussionen zum Matchen des Pfades: [JSR311-Expert-Group 2008d], [JSR311-Expert-Group 2008c]

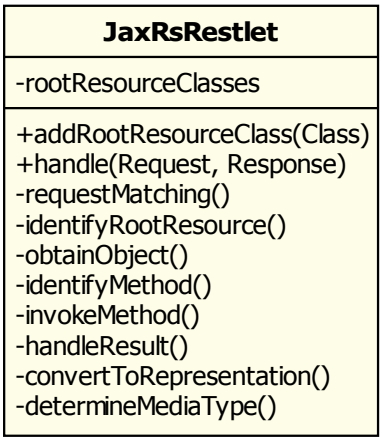

Abbildung 5.4: JaxRsRestlet, mit Verarbeitung der Antwort-Entität (eigene Grafik)

5.3 Provider

Jedes JaxRsRestlet speichert außer den Root-Ressource-Klassen auch die ihm übergebenen Provider. Für letztere werden teilweise spezielle Klassen vorgesehen, die alle in einem JaxRsRestlet vorhandenen Provider der jeweiligen Art (Entity-Provider, Context-Resolver oder Exception-Mapper) speichern. Diese Klassen wählen auch anhand gegebener Parameter (Java-Klasse, Datenformate etc.) den jeweils zu verwendenden Provider aus.

- Für die Entity-Provider (Message-Body-Reader und Message-Body-Writer) wird eine gemeinsame Klasse (`EntityProviders`) vorgesehen. Diese Klasse implementiert das JAX-RS-Interface `MessageBodyWorkers`, so dass die Instanz dieser Klasse bei einer entsprechender Anfrage durch einen Provider oder eine Ressource-Klasse direkt injiziert werden kann.
- Die Klasse `ExceptionMappers` speichert die Exception-Mapper und deligiert die Konvertierung einer Exception in eine JAX-RS-Response an den jeweiligen Exception-Mapper.
- Die `ContextResolver` werden in einer normalen Collection gespeichert, da die Auswahl der zu injizierenden Context-Resolver erst bei der Überprüfung des Parametertyps des zu injizierenden Parameter etc. festgestellt wird.

Beim Hinzufügen eines Providers zu einem JaxRsRestlet wird der Wrapper der Provider-Instanz je nach dem vom Provider implementierten Interface der jeweiligen Instanz einer oder mehrerer der genannten Klassen hinzugefügt.

Da auch Provider einen Lebenszylus haben und mit für die Laufzeitumgebung relevanten Informationen annotatiert sind, werden auch sie in einer dafür implementierten Klasse ge-

wrappt. Hierfür wird die Klasse `Provider` vorgesehen, welche alle Provider-Interfaces implementiert:

- Für Entity-Provider werden die Interfaces `MessageBodyReader` und `MessageBodyWriter` erweitert.
- Exception-Mapper und Context-Resolver sind nur mit @Provider annotiert, welche jedoch keinerlein Funktion hat. Deshalb kann hierfür jeweils direkt das JAX-RS-Interface verwendet werden.

Die Klasse `Provider` übernimmt die Instantiierung des JAX-RS-Providers. Da Provider auch andere Lebenszyklen als Singelton unterstützen dürfen, lässt sich dies später mit einer Erweiterung dieser Klasse entsprechend unterstützen. Die Klasse steuert auch die Injizierung in mit @Context annotierten Instanzvariablen und Bean-Settern. Dazu werden einige der `ParamGetter` aus Abschnitt 5.2.1 auf Seite 82 verwendet, wobei die Annotationen @*Param in Providern nicht erlaubt ist. Für die Entity-Provider stellt die Klasse `Provider` auch den Zugriff auf die mit Annotationen gegebenen Werte über Getter zur Verfügung. Die Provider werden dann je nach implementierten Interfaces in die entsprechenden Collections im JaxRsRestlet gespeichert.

Da die Annotation @Provider keinerlei Funktion hat (vgl. Abschnitt 3.8.3 auf Seite 62), braucht nicht beachtet werden, ob sie vorhanden ist oder nicht. Wenn die Annotation bei einem Provider fehlt, soll beim Laden dieses Providers ein entsprechender Hinweis geloggt werden. Ansonsten kann der Provider wie üblich verwendet werden.

5.3.1 Entity-Provider

Zum Deserialisieren des Nachrichtenkörpers von HTTP-Anfragen (siehe Abschnitt 2.1.1 auf Seite 10) und zum Serialisieren der Antwort-Entität werden Entity-Provider verwendet. Für sie gibt es in der JAX-RS-API zwei Interfaces: `MessageBodyReader` und `MessageBodyWriter`. Sie stellen Methoden für die Deserialisierung bzw. die Serialisierung zu Verfügung. Die Auswahl des konkreten Message-Body-Readers bzw. -Writers hängt außer vom Javatyp des gewünschten bzw. von der Ressource-Methode erzeugten Java-Objekts auch vom gegebenen bzw. gewünschten Datenformat der Anfrage bzw. der Antwort ab. Zur Auswahl des zu verwendenden Message-Body-Readers bzw. -Writers sind im Abschnitt 4.3 der JAX-RS-Spezifikation je ein Algorithmus vorgegeben. Dabei muss zusätzlich berücksichtigt werden, dass es in dieser Laufzeitumgebung auch möglich sein soll, Restlet-Repräsentationen als Entität zu verwenden. Die Restlet-API stellt diese direkt zur Verfügung, so dass keine Konvertierung mit einem Entity-Provider nötig ist.

Die JAX-RS-Spezifikation gibt vor, dass jede Laufzeitumgebung Entity-Provider für einige Kombinationen aus Javatyp und Datenformat zur Verfügung stellt. Die Liste findet sich im

Abschnitt 4.3.4 der JAX-RS-Spezifikation. Diese Provider müssen entsprechend implementiert werden. Die eigentliche (De-)Serialisierung ist dabei meistens entweder trivial (z. B. für Strings) oder wird an eine Bibliothek (z. B. für JAXB) weitergegeben. Im Abschnitt 4.1.4 auf Seite 71 ist ein Provider für die Restlet-Klasse `Form` beschrieben, welcher ebenfalls umgesetzt werden soll.

Message-Body-Reader

Wenn eine Ressource-Methode die Entität der HTTP-Anfrage anfordert, muss beim Erzeugen `ParamGetter`s (siehe Abschnitt 5.2.1 auf Seite 82) für die Ressource-Methode der gewünschten Javatyp der Entität geprüft werden:

- Wenn er `org.restlet.resource.Representation` ist, dann kann bei der Bearbeitung einer Anfrage die im Restlet-Request vorhandene Repräsentation direkt an die Ressource-Methode übergeben werden.
- Ist der Javatyp eine Unterklasse der Restlet-Klasse `Representation`, dann kann eine Instanz injiziert werden, wenn die Unterklasse einen Konstruktor hat, zu dem die Laufzeitumgebung alle Parameter zur Verfügung stellen kann:
  - Der einfachste Fall liegt vor, wenn die angegebene Klasse einen Konstruktor hat, der genau einen Parameter vom Restlet-Typ `Representation` verlangt. Diesem Konstruktor kann die vom Restlet-Request zur Verfügung gestellte Instanz übergeben werden.
  - Einige Unterklassen von Representation haben einen Konstruktor, der außer der Representation noch eine Klasse benötigen, z. B. die Repräsentationen für JAXB und JiBX. Wenn diese Unterklassen generisch sind, kann der Typparameter der Deklaration an die zu erzeugende Instanz der Representations-Unterklasse übergeben werden.

  Bei jeder Anfrage muss dann das mit dem ausgewählten Konstruktor jeweils erzeugte Objekt an die Ressource-Methode übergeben werden.
- Für Unterklassen der Klassse `Representation`, für die in den beiden vorigen Schritten kein Konstruktor gefunden wurde, könnten Annotationen eingeführt werden, die dann Werte für die weiteren Parameter eines der Konstruktoren enthalten. Da in Annotationen nur eine beschränkte Anzahl von Javatypen verwendet werden können, müsste eine Annotation zur Erkennng des Konstruktors verwendet werden und eine für die zu verwendenden Werte der Parameter. Da die Unterstützung von Containerspezifischen Objekten nicht von der JAX-RS-Spezifikation verlangt wird, sollen vorläufig nur die von Representation abgeleitete Klassen direkt unterstützt werden, für die mit den vorigen Schritten ein Konstruktor gefunden wurde.

Für alle Javatypen, für die mit dem beschriebenen Algorithmus kein passender Konstruktor gefunden wird, soll in der Parameterliste der Ressource-Methode ein Objekt gespeichert werden, welches den entsprechenden Message-Body-Reader sucht und ihm bei Anfragen den InputStream der Restlet-Repräsentation übergibt (siehe Abschnitt 2.4.1 auf Seite 34). Der zu verwendende Algorithmus zum Identifizieren des Message-Body-Readers ist im Abschnitt 4.3.1 der JAX-RS-Spezifikation beschrieben.

Message-Body-Writer

Ein von einer Ressource-Methode zurückgegebenes Objekt (oder ein in einer JAX-RS-Response als Entity enthaltenes Objetkt) muss in eine Restlet-Repräsentation konvertiert werden, welche dann der Restlet-Response übergeben wird. Dafür muss der zu verwendende Message-Body-Writer festgestellt werden. Der hierfür zu verwendende Algorithmus ist im Abschnitt 4.3.2 der JAX-RS-Spezifikation beschrieben. Der gefundene Message-Body-Writer wird dann in einer dafür vorgesehenen Unterklasse von `Representation` mitsamt allen benötigten Daten übergeben. Diese Klasse deligiert das Serialisieren (siehe Abschnitt 2.4.1 auf Seite 34) an den gefundenen Message-Body-Writer.

Die Restlet-Engine schickt erst die Statuszeile und die Kopfzeilen der Antwort zum Client. Danach ruft sie die Serialisierung des Nachrichtenkörpers auf. Deshalb kann eine von einem Message-Body-Writer geworfenen WebApplicationException mit einer aktuellen Restlet-Engine nicht gemäß der JAX-RS-Spezifikation verarbeitet werden. Das gleiche gilt, wenn im Message-Body-Writer Kopfzeilen hinzufügt oder geändert werden sollen.
Um eine geworfene WebApplicaionException oder die Veränderung von Kopfzeilen spezifikationsgemäß zu verarbeiten, müsste das im Restlet-Framework verwendete Konzept des Sendens von HTTP-Nachrichten umstrukturiert werden: Die Restlet-Engine dürfte die Status- und Kopfzeilen erst serialisieren und senden, wenn die ersten Daten auf den OutputStream der Antwort geschrieben werden, wobei die Engine die Kopfzeilen der Serialisierungsmethode für den Nachrichtenkörper zur Verfügung stellen müsste. Dies widerspricht dem bisher verwendeten Serialisierungskonzept des Restlet-Frameworks.
Um dies umzusetzen, können für jeden Thread der geplante Status und die vorgesehenen Kopfzeilen gespeichert werden. Sie würden dann beim Schreiben des ersten Zeichens auf den OutputStream abgeschickt. So könnten die Kopfzeilen geändert werden und auch WebApplicationExceptions verarbeitet werden. Die Restlet-Engine müsste dafür in einem sehr zentralen Punkt verändert werden.

Die Konvertierung mit einem Message-Body-Writer ist nicht nötig, wenn die Ressource-Methode ein Objekt vom Restlet-Typ `Representation` zurückgegeben hat. Das Problem mit der Veränderung der Kopfzeilen und der geworfenen WebApplicationException tritt dann nicht auf, da dies in Restlet-Repräsentationen nicht vorgesehen ist.

Kopieren von Kopfzeilen

Ressource-Methoden und die WebApplicationException können eine JAX-RS-`Response` zurückgeben, welche auch HTTP-Kopfzeilen enthält. Diese müssen in die Restlet-Response kopiert werden, wobei zu berücksichtigen ist, dass die Restlet-API die Kopfzeilen auf einem höheren Abstraktionsniveau verarbeitet. Eine eigene Implementierung ist fehlerträchtig. Außerdem müssten bei späteren Erweiterungen der Restlet-API diese auch hier umgesetzt werden, was evtl. vergessen wird.
Die Funktionalität dieses Übertragens ist in der Restlet-Engine schon vorhanden, da beim Deserialisieren einer Antwort im Client auch die Kopfzeilen der Anfrage entsprechend konvertiert werden müssen. Deshalb soll – in Absprache mit dem Restlet-Team – die Zugriffsklasse auf die Restlet-Engine (Klasse `Engine`) um eine Methode erweitert werden, die diese Funktionalität für die Restlet-API und damit auch die JAX-RS-Erweiterung zur Verfügung stellt. In der Noelios Restlet Engine sind dazu nur kleine Veränderungen im Methodenzuschnitt nötig.

Auch die umgekehrte Konvertierung ist nötig: Jeder Message-Body-Writer bekommt die Kopfzeilen der Antwort als MultivaluedMap<String, String> zur Verfügung gestellt. Dafür müssen neben den in JAX-RS-Response-Kopfzeilen gespeicherten Einträge ebenso die von der Restlet-API in die Restlet-Response eingetragenen Kopfzeilen übertragen werden. Auch hierzu ist eine vergleichbare Funktionalität in der Restlet-Engine vorhanden: Beim Serialisieren der Antwort muss der Restlet-Server ebenfalls die Daten aus der Restlet-Response als Strings zur Verfügung stellen. Auch diese Funktionalität kann mit kleinen Änderungen des Methodenzuschnitts und Veröffentlichung in der Klasse `Engine` genutzt werden.

5.3.2 Exception-Mapper

In Ressource-Klassen und Providern geworfene Exceptions müssen gefangen und von dem jeweiligen Exception-Mapper behandelt werden, wenn für die gefangene Exception (oder einer der Ober-Klassen) ein Exception-Mapper existiert.

Wie in Abschnitt 2.4.1 auf Seite 36 beschrieben, werden in Restlet alle auftretenden RuntimeExceptions auf den Status 500 gemappt; ansonsten existiert kein spezielles Verhalten bei Fehlern. Das Setzen des Statusses 500 kann also auch direkt in der JAX-RS-Laufzeitumgebung geschehen. Dies kann einfach umgesetzt werden, indem ein Exception-Mapper für Throwable definiert wird, der deshalb alle nicht von anderen Exception-Mappern behandelten Exceptions abfängt und auf den Status 500 mappt. Bei der Konvertierung für mit @*Param annotierte Parameter etc. auftretende WebApplicationExceptions müssen ebenfalls mit dem Exception-Mapper für WebApplicationExceptions behandelt werden. Andere bei dieser Konvertierung auftretenden Exceptions müssen je nach Annotationstyp auf

den Status 400 bzw. 404 gemappt werden; für Details siehe JAX-RS-Spezifikation, Abschnitt 3.2.

Wie schon angesprochen, benötigt der Exception-Mapper für die WebApplicationException Zugriff auf die gemappten Dateierweiterungen mit den dazugehörigen Datenformaten. Hierfür wird ein Typ `ExtensionBackwardMapping` definiert, welcher ebenfalls in eine mit @Context annotierte Instanzvariable dieses Typs in den Exception-Mapper für die WebApplicationExceptions injiziert wird. Weitere Details finden sich im Abschnitt 5.4.5 auf Seite 94 (im Absatz „Interna“)

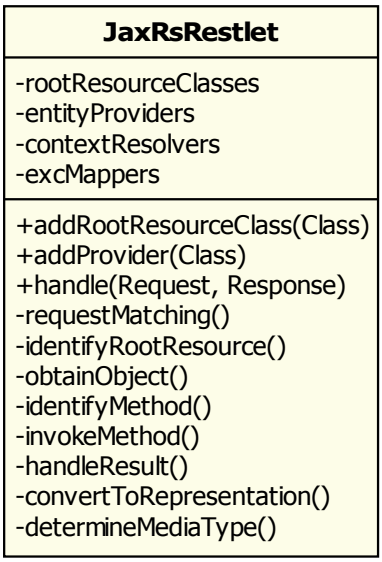

Abbildung 5.5: JaxRsRestlet mit Root-Ressource-Klassen und Providern (eigene Grafik)

5.4 Kontexte

Die JAX-RS-Spezifikation erlaubt für Ressource-Methoden und Sub-Ressource-Locatorn die Annotation @Context für einzelne Parameter, ebenso auf Instanzvariablen und Bean-Settern von Root-Ressource-Klassen und Providern. So annotierte Paramater etc. müssen von einem der Typen `Request`, `UriInfo`, `HttpHeaders`, `SecurityContext`, `MessageBodyWorkers` oder `ContectResolver` sein, wie in Abschnitt 4.1.5 auf Seite 72 beschrieben. Die ersten 4 Interfaces enthalten Informationen über die jeweils bearbeitete Anfrage, die anderen sind unabhängig von der jeweiligen Anfrage.

5.4.1 Request, HttpHeaders, SecurityContext und UriInfo

Die aus diesen Interfaces abfragbaren Informationen beziehen sich alle auf die jeweils zu bearbeitende Anfrage. Dennoch muss auf die injizierten Objekte von mehreren Threads gleichzeitig zugegriffen werden können, wobei jeder Thread die zu der von ihm zu bearbeitenden Anfrage passenden Informationen bekommen muss. Für das Interface `SecurityContext` sind ein paar weitere Zugriffe nötig, welche im nächsten Abschnitt behandelt werden.

Das Interface `UriInfo` stellt auch Zugriff auf die zum Zeitpunkt der jeweiligen Injektion durchlaufenen Instanzen der Ressource-Klassen und den bis dahin abgearbeiteten Teil des Pfades der Anfrage zur Verfügung, welche in der API „Vorläufer“ (ancestor) genannt werden. Bis auf diese Vorläufer bleiben alle Informationen dieser 4 Interfaces für eine Anfrage unverändert. Deshalb wird für diese Informationen eine gemeinsame Klasse vorgesehen (im folgenden `CallContext` genannt), welche die JAX-RS-Interfaces `Request`, `HttpHeaders` und `SecurityContext` implementiert. Sie soll für eine Anfrage alle Informationen speichern und kann schon einmal abgefragte Infos cachen, damit sie bei einem späteren Zugriff schnell verfügbar sind. Da für das Interface `UriInfo` für verschiedene Injektionsstellen für die Vorläufer verschiedene Informationen benötigt werden, muss hier jeweils ein eigenes Objekt injiziert werden. Vor jeder Injektion muss der aktuelle Stand der Vorläufer gespeichert werden, so dass er unverändert bleibt, auch wenn später weitere Pfad-Segmente verarbeitet werden. Die anderen Informationen direkt aus der Anfrage sind jedoch bei jeder Anfrage gleich und werden deshalb auch im `CallContext` gespeichert. In den `CallContext` werden auch die bis zum jetzigen Zeitpunkt der Anfrage verarbeiteten Teile des Pfades und die bisherig durchlaufenen Instanzen der Ressource-Klassen gespeichert (siehe auch Abschnitt 5.2.2 ab Seite 83).

Um ein Objekt von mehreren Threads aus nutzen zu können, muss für jede Anfrage der dazugehörige `CallContext` in einem Proxy gespeichert werden, der für den anfragenden Thread das jeweils benötigte Objekt zur Verfügung stellt. Hierfür wird ein Proxy erzeugt, der mit Hilfe einer Instanz vom Type `ThreadLocal` die Speicherung von Objekten zugeordnet zum jeweiligen Thread ermöglicht. Die Proxy-Klasse implementiert ebenfalls die Interfaces `Request`, `HttpHeaders` und `SecurityContext` und deligiert die Methodenaufrufe an den zum jeweiligen Thread gehörenden CallContext weiter.
Die für `UriInfo` extra vorgesehene Klasse ruft die im `CallContext` vorhandenen Informationen über dessen Proxy dort ab. Für die Vorläufer speichert das UriInfo-Objekt jeweils einen `ThreadLocal`-Proxy. Dieser muss vor der Injektion in die Ressource-Klasse oder die Ressource-Methode den jeweils aktuellen Stand der Vorläufer aus dem jeweiligen CallContext speichern. Bei der Injektion eines `UriInfo` in einen Provider muss jeweils vor dem Aufrufen des Providers der aktuelle Stand aus dem CallContext für den jeweiligen Thread festgeschrieben werden.

Die Klasse `CallContext` arbeitet nur für einen Thread. So sind die Verantwortlichkeiten zwischen den zu bearbeitenden Daten und der Zur-Verfügung-Stellung an verschiedene Threads klar getrennt. Außerdem bleibt die Klasse `CallContext` übersichtlicher. Bei der Verarbeitung im JaxRsRestlet wird diese Proxy-Funktionalität nicht benötigt, da dort jeder Thread den jeweiligen CallContext als lokale Variable nutzen kann.

5.4.2 SecurityContext

Authentifizierung Für die Authentifizierung soll – wie in Restlet üblich – ein `Guard` verwendet werden (siehe Seite 33). Bisher ist in einem Restlet-Request nicht bekannt, ob die Anfrage authentifiziert wurde. Der Anwendungsentwickler muss sicherstellen, dass ein Guard in der Kette der Restlets enthalten ist.
Da das entworfene JaxRsRestlet (und auch die im Abschnitt 5.8 ab Seite 100 beschriebene JaxRsApplication) beim Starten der Anwendung nicht prüfen kann, ob die Anwendung eine Authentifizierung – und damit einen Guard – benötigt wird, ist es nicht sinnvoll, dass die Verwendung eines Guards grundsätzlich erzwungen wird. Der `SecurityContext` darf den Namen des Anwenders jedoch nur dann zurückgeben oder prüfen ob der Anwender eine bestimmte Rolle hat, wenn die Anfrage authentifiziert wurde. Deshalb soll ein `Guard` verwendet werden, der so erweitert wurde, dass er im Falle einer erfolgreichen Authentifizierung speichert, dass der Benutzer authentifiziert wurde. Da die Möglichkeit dieser Prüfung für alle Restlet-Entwickler nützlich ist, ist es sinnvoll, die Restlet-Klassen `Guard` und `ChallengeResponse`[4] so zu erweitern, dass die Information über eine erfolgreiche Authentifizierung als boolsches Attribut direkt aus dem ChallengeResponse abfragbar ist.
Um die Authentifizierungsdaten des Connectors bzw. des darin verwendeten HTTP-Containers (Servlet-API etc.) nutzen zu können[5], muss die Adapter-Klasse des jeweiligen Restlet-Connectors eine zusätzliche Methode bekommen, die aus der jeweils gewrappten Anfrage einen eventuell vorhandenen Benutzernamen o. ä. ausliest und als authentifizierten ChallengeResponse speichert.

Rollenverwaltung Für die Prüfung, ob der die Anfrage stellende Client (bzw. dessen Benutzer) eine bestimmte Rolle hat, wird Zugriff auf eine Rollenverwaltung benötigt. In der Restlet-API ist z. Zt. keine eigene Rollenverwaltung vorhanden.

Wenn der Server-Connector eine Rollenverwaltung zur Verfügung stellt (wie z. B. Servlet-Container), soll diese nutzbar sein. Eine Möglichkeit hierfür ist, das diesen Dienst zur Verfügung stellende Objekt des HTTP-Containers hinter einem Interface zu wrappen, um die API von der konkreten Implementierung des Connectors zu entkoppeln. Dieses Interface hätte genau eine Methode mit einem Parameter (dem Rollennamen). Der Zugriff hierauf könnte

[4] Objekte vom Typ ChallengeResponse speichern die Authentifikationsdaten einer einzelnen Anfrage.
[5] Zur Nutzung der Authentifizierung der Connectoren siehe Abschnitte 2.4.1 (Seite 36) und 4.1.5 (Seite 72).

– ähnlich wie bei der Prüfung in der Servlet-API – mit einer neu anzulegenden Methode `ChallengeResponse.isUserInRole()` geprüft werden. So würde diese Funktionalität allen Restlet-Entwicklern zur Verfügung stehen. Diese Methode würde dann von der Implementierung des `SecurityContext`es aufgerufen werden.

Das Restlet-Team plant, die Authentifikation etc. mit – gegenüber dem aktuellen Stand – erweiterten Anforderungen zu überarbeiten und auch eine Rollenverwaltung zu integrieren[6]. Da die nächste Restlet-Version (1.1) schon im aktuellen Quartal[7] freigegeben werden soll, soll diese umfangreiche Änderung erst in Restlet-Version 1.2 umgesetzt werden. Dies ermöglicht ein ausgiebiges Testen vor der nächsten Version. Deshalb wird die Nutzung der im Connector enthaltenen Authentifizierungsmöglichkeiten in dieser Arbeit nicht detailierter betrachtet. Das im vorigen Absatz entworfene Konzept kann in die Diskussionen über die Veränderungen des Restlet-Guards eingebracht werden.

Damit Anwendungsentwickler einer JAX-RS-Anwendung trotzdem eine Rollenverwaltung verwenden können, wird hierfür ein Interface `RoleChecker` vorgesehen, an welches Anfragen bzgl. einer bestimmten Rolle zusammen mit dem `Principal` des authentifizierten Anwenders delegiert werden sollen. Ein Objekt dieses Typs muss dem JaxRsRestlet beim Start der Anwendung übergeben werden, wenn die Methode `SecurityContext.isUserInRole()` verwendet werden soll. Die Implementierung dieses Interfaces ermöglicht es, die Zugriffsberechtigungen je nach den zu implementierenden Anforderungen bspw. gegen eine Datenbank-Tabelle zu prüfen. Zur einfachen Nutzung während der Entwicklung einer Anwendung werden auch Standard-Implementierungen vorgesehen, die entweder alle Zugriffe erlauben, alle verbieten oder den Status 500 („interner Server Fehler“) an den Client zurückliefern, um zu zeigen, dass die Anwendung nicht korrekt konfiguriert ist. Die letzte der genannten Möglichkeit wird als Standardverhalten vorgesehen, wenn der Anwendungsentwickler keinen RoleChecker gesetzt hat.

Wenn Restlet später eine eigene Rollenverwaltung erhält, lässt sich dieses Interface verwenden, um die Anfragen an diese Rollenverwaltung weiterzuleiten.

Authentifikationsschema Das Authentifikationsschema steht – wie auch der Benutzername – im vom `Request` abfragbaren `ChallengeResponse` zur Verfügung. Die jeweiligen Restlet-Namen der Authentifikationsschemata müssen nur auf die im SecurityContext als Konstanten angegebenen Werte gemappt werden.

Verschlüsselung Die Überprüfung, ob eine Anfrage verschlüsselt ist oder nicht, lässt sich einfach lösen, da eine entsprechende Methode im Restlet-Request zur Verfügung steht, an welche der Methodenaufruf weiterdeligiert werden kann.

[6] Überarbeitung der Authentifikation etc: [Louvel 2008e].
[7] 3. Quartal 2008; siehe Restlet-Roadmap: [Noelios 2008a]

5.4.3 MessageBodyWorkers

Die Ergebnisse des Interfaces `MessageBodyWorkers` sind für alle Anfragen und Klassen, in die sie injiziert werden, gleich. Deshalb kann hier immer das gleiche die Entity-Provider speichernde Objekt vom Typ `EntityProviders` injiziert werden (siehe Abschnitt 5.3 auf Seite 85).

5.4.4 ContextResolver

Die Ergebnisse von Anfragen an `ContextResolver` hängen vom Typparameter des generischen Typs des Parameters, Bean-Setters oder der Instanzvariable ab, ist aber unabhängig von der konkreten Anfrage. Deshalb soll er zusammen mit den Metadaten über die zu injizierenden Instanzvariablen und Methoden gecacht werden.
Was für ein Objekt injiziert wird, hängt von der Anzahl der zu dem Typparameter des Parameters, Bean-Setters oder der Instanzvariable passenden Context-Provider ab:

- Wenn kein passender Context-Resolver existiert, muss ein das Interface `ContextResolver` implementierendes Objekt injiziert werden, dass immer null zurück gibt.
- Wenn es genau einen Context-Resolver dafür gibt, dann kann er direkt injiziert werden.
- Wenn es mehrere passende Context-Resolver gibt, muss ein das Interface `ContextResolver` implementierendes Objekt injiziert werden, welches über die passenden Context-Resolver iteriert, bis einer der enthaltenen Context-Resolver einen Kontext zurückgibt.

5.4.5 Interna

Wenn für keines der gewünschten Datenformate eine passende Ressource-Methode implementiert ist, dann soll die Laufzeitumgebung Links zu den verfügbaren Datenformaten zurückgeben. Dazu muss der Exception-Mapper für die WebApplicationExceptions (siehe Abschnitt 5.3.2 auf Seite 89) die zu den möglichen Datenformaten passenden virtuellen Dateierweiterungen herausfinden. Diese müssen aus dem Mapping der Datenformate geladen werden. Dafür wird ein Typ `ExtensionBackwardMapping` definiert, welcher direkt nach der Instantiierung des Exception-Mapper für die WebApplicationExceptions in eine mit @Context annotierte Instanzvariablen diese Typs injiziert wird.

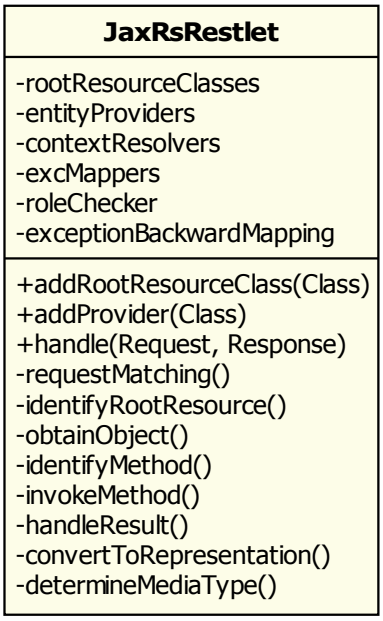

Abbildung 5.6: JaxRsRestlet, mit Instanzen für Kontexte (eigene Grafik)

5.5 Exceptions außerhalb von JAX-RS-Methoden

Innerhalb der Laufzeitumgebung auftretende Exceptions (z. B. Fehler bei der Konvertierung eines Parameters) und andere Fehler (z. B. nicht gefundene Ressource-Methoden) sollen möglichst speziell behandelt werden, um exakte Fehlerbehandlungen und / oder -meldungen zu ermöglichen (siehe Abschnitt 4.2.1 auf Seite 75). Deshalb werden für die einzelnen Fehler spezielle Exception-Klassen vorgesehen.
In den in der JAX-RS-Spezifikation definierten Algorithmen ist an 9 Stellen ein Abbruch mit einer WebApplicationException möglich, und auch während der Konvertierung der Entitäten, der Instantiierung von Root-Ressource-Klassen etc. können Fehler auftreten. Deshalb werden entsprechend viele Fehlerbehandlungsmethoden nötig. Um die Klasse `JaxRsRestlet` übersichtlich und damit wartbar zu gestalten (siehe Abschnitt 4.2.6 auf Seite 77), soll sie von solchen Fehlerbehandlungsmethoden möglichst frei gehalten werden. Deshalb wird für die Fehlerbehandlung eine eigene Klasse vorgesehen, von der jedes JaxRsRestlet eine Instanz hält.

In diesem Abschnitt wurden die letzten der für die Klasse `JaxRsRestlet` wichtigen Attribute und Methoden vorgestellt. Diese werden im folgenden Klassendiagramm gezeigt. Die Methoden sind wieder in einer typischen Aufruf-Reihenfolge angeordnet. Sequenzdiagramme über den Ablauf des Starts der Anwendung und der Verarbeitung der Anfragen finden sich auf Seite 102 und 103.

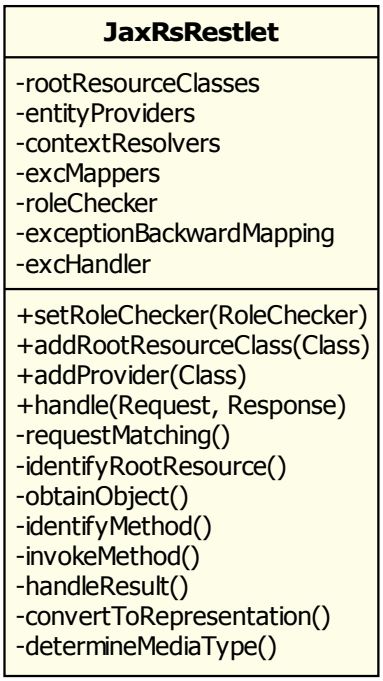

Abbildung 5.7: JaxRsRestlet, vollständig (eigene Grafik)

5.6 RuntimeDelegate

Die von jeder Laufzeitumgebung mitzuliefernde konkrete Unterklasse der Klasse `RuntimeDelegate` dient zur Instantiierung des Interfaces `HeaderDelegate` für einige Abstraktionsklassen von HTTP-Kopfzeilen, zur Instantiierung einiger implementationsspezifischen Klassen und optional zur Erzeugung von Endpunkten, wenn die implementierende Laufzeitumgebung diese unterstützt.

In der JAX-RS-Spezifikation (Abschnitt 7.1) sind verschiedene Möglichkeiten vorgesehen, um die Implementierung des RuntimeDelegates den JAX-RS-API-Klassen bekannt zu machen. Von diesen soll die Möglichkeit der automatischen Erkennung mit Hilfe der Datei META-INF/services/javax.ws.rs.ext.RuntimeDelegate genutzt werden. Sie ist am einfachsten zu realisieren. Die erste Zeile dieser Datei besteht dazu aus dem vollständigen Namen (inklusive dem Package-Namen) der konkreten Implementierung der abstrakten Klasse `RuntimeDelegate`.

5.6.1 HeaderDelegates

Das RuntimeDelegate muss `HeaderDelegate`s für die Serialisierung und Deserialisierung der JAX-RS-Klassen `Cookie`, `CacheControl`, `EntityTag`, `MediaType` und `NewCookie` zur Verfügung stellen. Für die Klassen `Cookie`, `EntityTag`, `MediaType` und `NewCookie` ist die Serialisierung und Deserialisierung der korrespondierenden Restlet-Klassen bereits in der Restlet-API oder in der Restlet-Engine vorhanden. Diese können entweder direkt genutzt oder durch mit dem Restlet-Team abgestimmten Erweiterungen der Schnittstelle der Restlet-Engine nutzbar gemacht werden. Vor der Nutzung müssen die JAX-RS-Objekte in die entsprechenden Restlet-Objekte konvertiert werden.

CacheControl

Die in der JAX-RS-Klasse `CacheControl` vorhandenen Werte müssen direkt auf die entsprechenden Attribute der HTTP-Kopfzeile „Cache-Control“ abgebildet werden. Die Details dazu sind im RFC 2616 spezifiziert. Die Angabe von Caching-Optionen wird von der Restlet-API bisher nicht unterstützt. Dies muss deshalb noch implementiert werden. Da die Möglichkeit der Nutzung der Caching-Optionen für alle Restlet-Nutzer sinnvoll ist, soll dieses Feature direkt in die Restlet-API und -Engine integriert werden. Zur Nutzung aus der JAX-RS-Erweiterung muss dann das JAX-RS-CacheControl-Objekt in das äquivalente Restlet-Objetkt konvertiert werden.

Wenn der Client eine Anfrage mit der HTTP-Version 1.0 gestellt hat, dann versteht er die Kopfzeile „Cache-Control“ vermutlich nicht, da sie erst in der HTTP-Version 1.1 definiert wurde. Statt dessen kann die Kopfzeile „Pragma“ verwendet werden. Für diese Kopfzeile ist nur die Angabe „no-cache“ definiert, welche angibt, ob eine Antwort ohne Prüfung der Aktualität von einem Cache zurückgegeben werden darf oder nicht. Da jede im `CacheControl` gesetzte Angabe eine Einschränkung der Caching-Erlaubnis ist oder sein kann, muss bei HTTP/1.0-Anfragen die Kopfzeile „Pragma“ auf den Wert „no-cache“ gesetzt werden, wenn irgendeines der gegebenen Attribute nicht dem Standard-Wert entspricht. Die Kopfzeile „Cache-Control“ wird für HTTP/1.0-Anfragen nicht gesetzt.

5.6.2 Abstrakte JAX-RS-Klassen

Die Klasse `RuntimeDelegate` dient auch zur Instantiierung der implementationsspezifischen Klassen, welche von in der JAX-RS-API enthaltenen Klassen benötigt werden. Diese Klassen werden im Folgenden aufgeführt:

ResponseBuilder Diese abstrakte Klasse muss entsprechend der Dokumentation in der Klasse implementiert werden. Sie speichert außer dem Status und der Entität auch die Kopfzeilen der zu sendenden Antwort.

Bei den Kopfzeilen ist darauf zu achten, dass `NewCookie`s mit jedem Namen nur einmal vorkommen können; d. h. neuere `NewCookie`s mit einem schon vorhandenen Namen ersetzen schon gesetzte Cookies mit diesem Namen. Deshalb sollen die `NewCookie`s in einer eigenen Map gespeichert und erst beim Erzeugen der Antwort für die Response den anderen Kopfzeilen hinzugefügt werden. In der Methode `header(..)` ist darauf zu achten, dass mit dem Kopfzeilennamen „Set-Cookie“ übergebene Werte ebenfalls als `NewCookie` in dieser Map gespeichert werden.

Die Methode ResponseBuilder.variants(List<Variants>) soll eine Vary-Kopfzeile entsprechend den angegebenen Varianten erzeugen (vgl. Abschnitt 2.2.3 auf Seite 24). In der Restlet-API ist für die Vary-Kopfzeile die Enumeration `Dimension` vorgesehen. Diese Enumeration soll auch hier genutzt werden, auch um bei eventuellen zukünftigen Änderungen / Erweiterungen des Restlet-Frameworks diese einfach mit nutzen zu können. Diese Informationen können direkt in die Restlet-Response geschrieben werden.
Die erzeugte Kopfzeile steht dann aber noch nicht in der JAX-RS-Response zur Verfügung, wo sie aber auch erscheinen sollte. Deshalb müssen die Restlet-Dimensions in einen String umgewandelt werden, welcher dann als VARY-Kopfzeile in die HTTP-Kopfzeilen des JAX-RS-ResponseBuilders eingetragen wird. Da die Funktionalität der Konvertierung in einen String in der Restlet-Engine schon vorhanden ist, soll sie dort – wieder in Absprache mit dem Restlet-Team – in eine eigene Methode ausgelagert und über die Klasse `Engine` genutzt werden können.

UriBuilder Der `UriBuilder` ist dafür verantwortlich, dass ihm übergebenen Werte gültig sind oder – wenn nötig – kodiert werden. Wenn die automatische Kodierung abgeschaltet ist, muss der `UriBuilder` prüfen, ob die übergebenen Werte ohne Kodierung korrekt sind oder Zeichen oder Zeichenkombinationen enthalten, die in URIs nicht erlaubt sind. Ist dies der Fall, dann muss der UriBuilder den Methodenaufruf mit einer Exception zurückweisen.

Wenn mehrere Elemente (z. B. Pfad-Segmente) übergeben werden, ist darauf zu achten, dass nicht erst einzelne dieser Elemente dem UriBuilder hinzugefügt werden und dann festgestellt wird, das andere mit einer Exception zurückgewiesen werden müssen. Deshalb

muss der UriBuilder erst alle hinzuzufügenden Elemente überprüfen und erst danach speichern. Weiterhin muss berücksichtigt werden, dass in verschiedenen Teilen der Anfrage verschiedene Zeichen erlaubt bzw. verboten sind. Im URI-Schema und im Hostnamen sind z. B. viele Zeichen ganz verboten, die bspw. im Pfad oder in der Query erlaubt sind, oder alternativ kodiert werden können. In der Query müssen Parameter anders kodiert werden als in Matrix-Parametern. Deshalb sollen die in der Restlet-Klasse `Reference` vorhandenen Methoden zur Kodierung nicht genutzt werden. Ein zweiter Grund hierfür ist, dass das Restlet-Framework und auch die Klasse `Reference` weiterentwickelt werden, so dass diese Methoden später vielleicht einzelne Zeichen kodiert, die z. Zt. nicht kodiert werden oder umgekehrt.

VariantListBuilder Die abstrakte Klasse `VariantListBuilder` muss entsprechend der Dokumentation in dieser Klasse implementiert werden.

5.6.3 Erzeugung von Endpunkten

Weiterhin dient die Klasse RuntimeDelegate auch zur Erzeugung von Endpunkten, wenn dies von der Laufzeitumgebung unterstützt wird. Da die hier zu entwerfende Implementierung auf der Basis von Restlet keine Endpunkte unterstützt, wirft diese Methode immer eine UnsupportedOperationException.

5.7 Extension-Mapping

Das Mapping der Datei-Erweiterungen ist auch für nicht-JAX-RS-Anwendungen sinnvoll. Das gleiche Feature ist mit Hilfe von Query-Parametern statt virtuellen Datei-Erweiterungen im TunnelService bzw. TunnelFilter schon in Restlet implementiert (siehe Abschnitt 2.4.1 auf Seite 35). Deshalb soll dieses Feature ebenfalls in den TunnelService und den TunnelFilter integriert werden.

5.8 JAX-RS-Application

Die Schnittstelle für die Benutzung der entworfenen JAX-RS-Laufzeitumgebung durch den Anwendungsentwickler soll eine Unterklasse der Restlet-Klasse `Application` (siehe Seite 36) werden. Als Klassename wird `JaxRsApplication` vorgesehen. Diese Klasse hat folgende Aufgaben:

- Sie bekommt eine oder mehrere JAX-RS-ApplicationConfig-Instanzen übergeben. Sie liest die Root-Ressource-Klassen und Provider aus der ApplicationConfig aus und leitet sie an das enthaltenen JaxRsRestlet weiter.
- Die JaxRsApplication nimmt ggf. einen RoleChecker entgegen und leitet ihn an das JaxRsRestlet weiter.
- Sie stellt dem TunnelService der Klasse `Application` die für das Extension-Mapping zu verwendenden Mappings aus der ApplicationConfig zur Verfügung (siehe vorigen Abschnitt).
- Sie verschachtelt den Guard und das JaxRsRestlet, so wie es benötigt wird. Der Guard steht – wenn gesetzt – als Instanz-Variable zur Verfügung. Wenn kein Guard gesetzt wurde, dann wird keiner verwendet.

Die JaxRsApplication kapselt einzelne JAX-RS-Anwendungen, so wie es in der Restlet-API vorgesehen ist. So wird eine der Restlet-API vergleichbare Schnittstelle zur Verfügung gestellt. Dies erhöht die Wiedererkennung und Wartbarkeit durch andere Restlet-Entwickler (vgl. Abschnitt 4.2.2 auf Seite 76). Wenn ein Anwendungsentwickler weitere Filter etc. nutzen möchte, kann er eine Unterklasse der Klasse `JaxRsApplication` erzeugen und die Methode `createRoot()` nach den eigenen Wünschen reimplementieren.

Im folgenden Klassendiagramm sind die wichtigen Attribute und Methoden der Klasse `JaxRsApplication` aufgeführt, sortiert in einer typischen Aufruf-Reihenfolge.

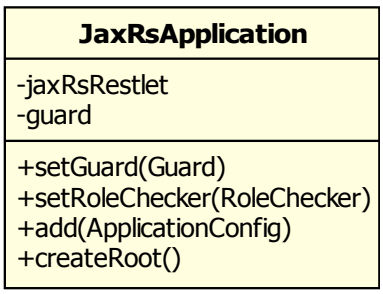

Abbildung 5.8: JAX-RS-Application (eigene Grafik)

In der folgenden Abbildung werden die beiden JAX-RS-Klassen in der Restlet-Klassenhierarchie gezeigt. Dazu vergleiche auch Abbildung 2.8 auf Seite 33.

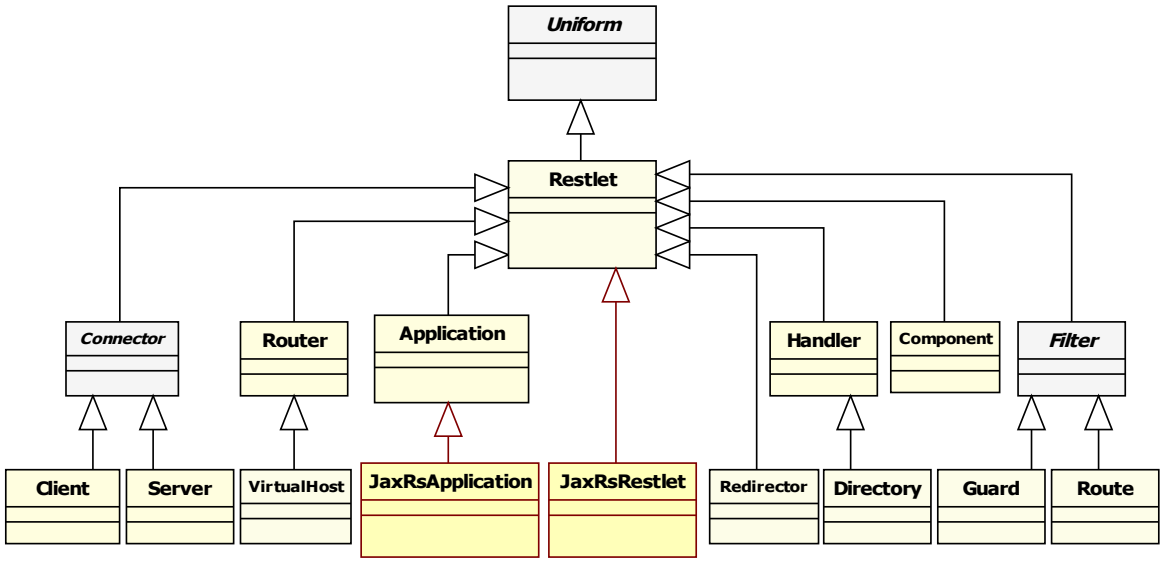

Abbildung 5.9: Restlet mit JAX-RS-Klassen (angelehnt an [Noelios 2008c])

Die nächste Abbildung zeigt, wie eine Anfrage vom VirtualHost durch verschiedene Filter (z. B. den TunnelFilter und den Guard) läuft und dann vom JaxRsRestlet an eine der Ressource-Methoden weitergeleitet wird. Dazu vergleiche auch Abbildung 2.9 auf Seite 37.

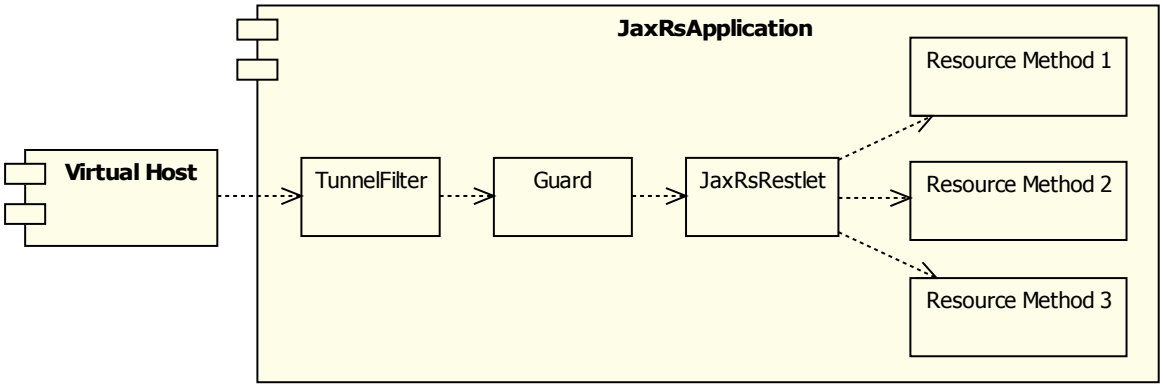

Abbildung 5.10: Restlet: JAX-RS-Application (eigene Grafik, vgl. Abb. 2.9 auf Seite 37)

Dieses Sequenzdiagramm zeigt, wie die JaxRsApplication und das JaxRsRestlet beim Starten der Laufzeitumgebung initialisiert werden.

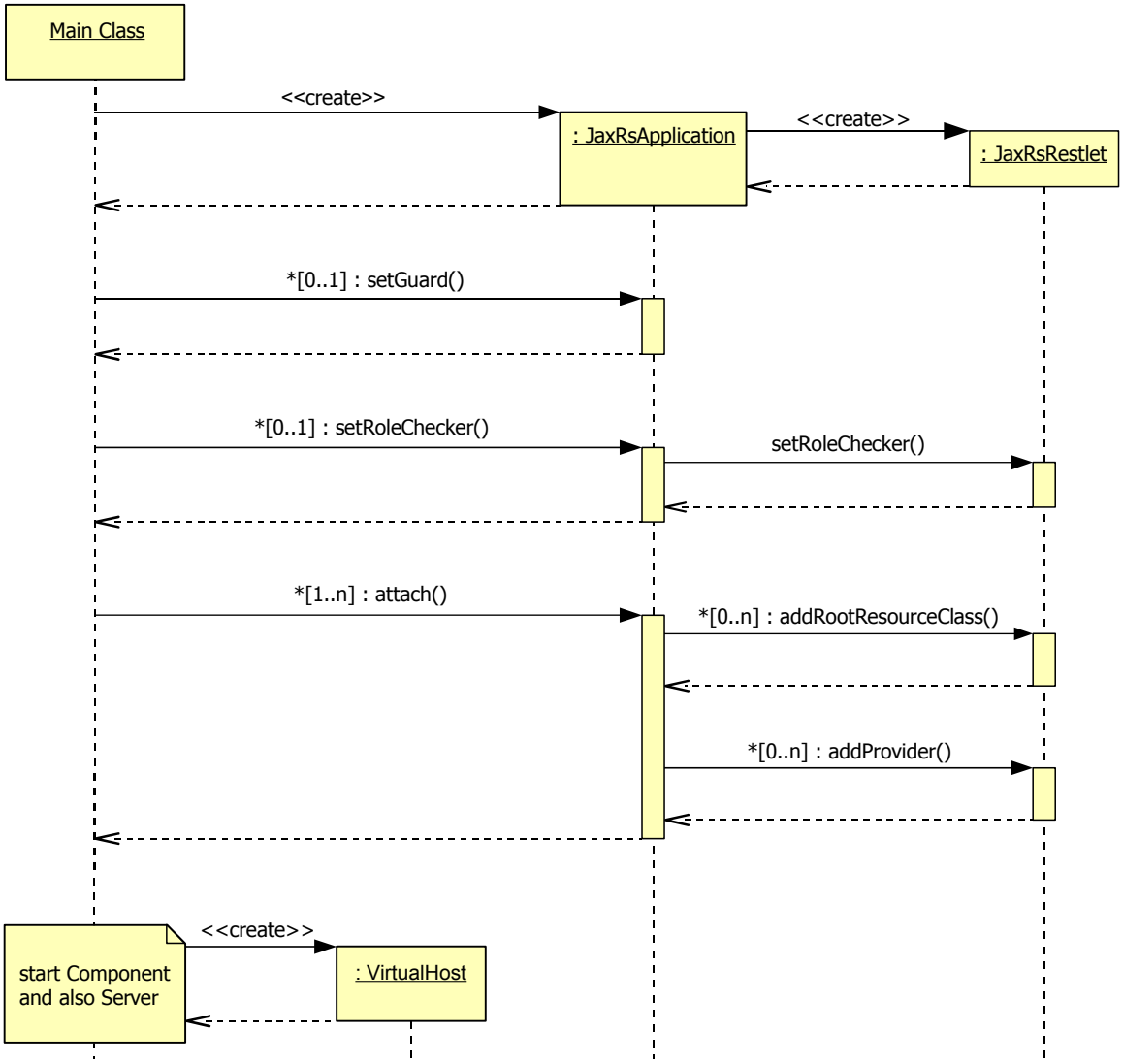

Abbildung 5.11: Start der Laufzeitumgebung (eigene Grafik)

Die Fortsetzung mit der Verarbeitung von Anfragen wird auf der nächsten Seite gezeigt.

Dieses Sequenzdiagramm zeigt, wie Anfragen von Clients im Pinzip abgearbeitet werden. Verwendete Filter (z. B. der TunnelFilter oder der Guard) werden in dieser Abbildung nicht berücksichtigt. Sie gehören zwischen die JaxRsApplication und das JaxRsRestlet.

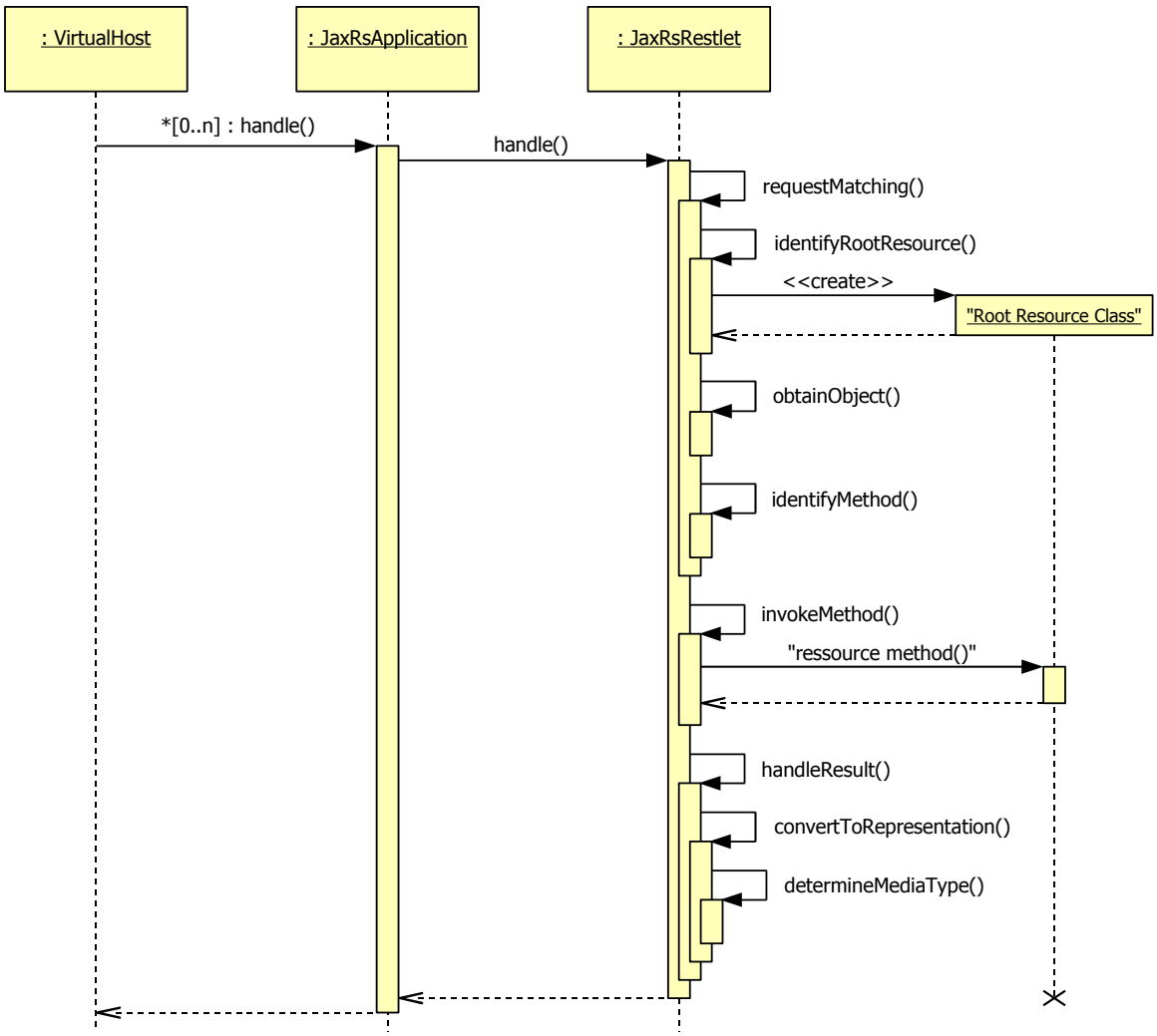

Abbildung 5.12: Bearbeitung einer Anfrage (eigene Grafik)

Wenn die Anfrage noch von einem Sub-Ressource-Locator bearbeitet wird, dann würde dieser von der Methode `obtainObject()` aufgerufen werden, welche eine Instanz einer (meist anderen) Ressource-Klasse zurückgibt. Diese enthält dann die aufzurufende Ressource-Methode.

Wenn eine Ressource-Methode eine `Response` zurückgibt, muss sie in eine Restlet-Response umgewandelt werden. Die Methode `JaxRsRestlet.handleResult()` ruft dazu eine hier nicht angeführte Konvertierungsmethode auf, welche dann die Methode `convertToRepresentation()` auftruft.

5.9 Fazit

Wie gezeigt wurde, kann die JAX-RS-Spezifikation auf Basis des Restlet-Frameworks mit einer Einschränkung implementiert werden: Das Verändern von Kopfzeilen durch Message-Body-Writer und die Berücksichtigung von durch sie geworfenen WebApplicationExceptions ist nicht möglich. Dies ließe sich nur durch eine größere Veränderung der Restlet-Engine umsetzen. Die anderen Anforderungen lassen sich auf die Restlet-API abbilden.

Einige der benötigten Funktionalitäten sind schon in der Restlet-API oder der Restlet-Engine vorhanden und können genutzt werden. Andere für die JAX-RS-Laufzeitumgebung neu zu implementierende Features sind auch für andere Restlet-Entwickler sinnvoll und können direkt in die Restlet-API integriert werden. Das betrifft insbesonders das Extension-Mapping mit virtuellen Dateierweiterungen und die Speicherung der Authentifizierung mit dem Guard.

Da in der Restlet-API bisher keine Rollenverwaltung vorhanden ist, wurde dafür ein eigenes Interface entworfen. Dieses oder ein ähnliches Interface könnte später in die Restlet-API übernomen werden, um dort ebenfalls eine Rollenverwaltung zu implementieren.

Durch das hohe Abstraktionsniveau der Restlet-API müssen die Kopfzeilen von JAX-RS-Antworten nach der Bearbeitung durch die Ressource-Methode erst in die jeweiligen Attribute der Restlet-Anfrage kopiert werden. Zum Serialisieren der Anfrage müssen sie bei Bedarf dem Message-Body-Writer wieder als Map zur Verfügung gestellt werden. Auch die Restlet-Engine muss sie für den Connector wieder serialisieren. Dies erzeugt einen nicht unerheblichen Aufwand bei der Bearbeitung von Anfragen.

6 Implementierung

Die entworfene JAX-RS-Laufzeitumgebung wird mit Unterstützung des Restlet-Teams eine offizielle Erweiterung der Restlet-API. Deshalb wird die Infrastruktur des Restlet-Projekts genutzt: Der Quellcode wird im Restlet-Subversion-Repository als Restlet-Erweiterung abgelegt (Projekt org.restlet.ext.jaxrs_0.8 [1]). Bugs können über das Bug-Tracking-System von Restlet verwaltet werden.

Im Rahmen einer akademischen Arbeit ist es nicht das Ziel, eine bis ins letzte Detail funktionierende Implementierung zu erstellen. Vielmehr soll gezeigt werden, dass das entworfene Konzept wie entworfen funktioniert. Einige der Anforderungen wurden deshalb nur prototypisch umgesetzt.

6.1 Entwicklungsumgebung

Da das Restlet-Projekt mit Eclipse 3.x entwicklet wird, wird für die Software-Entwicklung im Rahmen dieser Arbeit die Entwicklungsumgebung Eclipse in Version 3.3 verwendet.
Da die JAX-RS-Spezifikation auf Annotationen basiert und die Restlet-API die generischen Datentypen ausgiebig nutzt, wird Java in Version 1.5 / 5.0 benötigt.

6.2 Umsetzung

Die Restlet-Erweiterung wurde wie entworfen in den wesentlichen Teilen implementiert. Einige der entworfenen Änderungen der Restlet-Engine wurden vom Restlet-Team nicht ganz, wie vom Autor dieser Arbeit entworfen, unterstützt. Ein bis jetzt nicht umgesetzter Vorschlag wird im Rahmen der Weiterentwicklung der Restlet-API in zukünftige Überlegungen mit einbezogen. Details dazu finden sich im weiteren Verlauf dieses Abschnittes.

Aus zeitlichen Gründen wurden einige Details nur prototypisch implementiert, z. B. das Dekodieren und Kodieren der Anfrage-URI mit `UriInfo` und das Kodieren und Prüfen der URI-Teile mit dem `UriBuilder`. Alle benötigten Methoden, Variablen etc. sind jedoch vorhanden.

[1] Die Version 0.8 gilt zur Zeit der Abgabe dieser Arbeit.

Erweiterungen der Restlet-API und -Engine

Außer dem Restlet-Team entwickeln auch andere Entwickler Erweiterungen für das Restlet-Framework. Diese Entwickler checken ihre Änderungen am Quellcode der eigenen Erweiterungen selber in das Subversion-Repository ein.
Änderungen an anderen Teilen der Restlet-API oder -Engine (in Falle dieser Arbeit alle Änderungen außerhalb der JAX-RS-Erweiterung) werden vom vorschlagenden Entwickler lokal entwickelt und getestet. Danach erzeugt der Entwickler einen Subversion-Patch und schickt diesen an das Restlet-Team, welches die Änderungen nach einer Überprüfung in die Restlet-API übernimmt (oder ggf. ablehnt).

Engine-Erweiterungen Da die Entwicklung dieser JAX-RS-Laufzeitumgebung durch das Restlet-Team unterstützt wird, war es unkompliziert möglich, interne Funktionalitäten der Restlet-Engine zu nutzen. Dazu wurden in der Schnittstellenklasse zur Restlet-Engine (Klasse `Engine`) einige Methoden hinzugefügt oder erweitert. Dies sind im wesentlichen die beiden Methoden `copyResponseHeaders(...)` (mit zwei verschiedenen Signaturen), um die in einer Restlet-`Response` als High-Level-Objekte gespeicherten HTTP-Kopfzeilen in eine Liste von Restlet-`Parameter`n zu kopieren und anders herum (siehe Abschnitt 5.3.1 auf Seite 89). Außerdem wurden einige Methoden hinzugefügt, um in der Restlet-Engine vorhandenen Funktionalitäten zur Konvertierung von `Cookie`s (vom Client übertragene Cookies), `CookieSetting`s (zu setzende JAX-RS-`NewCookie`s) und `Dimension`s (für die VARY-Kopfzeile) aus einem String in ein High-Level-Objekt und / oder umgekehrt zu nutzen. Weiterhin wurde die Methode zum Parsen einer Query[2] so erweitert, dass auch Matrix-Parameter damit geparst werden können. Entsprechend den Anforderungen der JAX-RS-Spezifikation muss auch die automatische Dekodierung abgeschaltet werden können.

Die Möglichkeit der Veränderung der Kopfzeilen und des Werfens einer WebApplication-Exception in einem Message-Body-Writer (siehe Abschnitt 5.3.1 auf Seite 88) ist aus der Sicht des Autors dieser Arbeit kein wichtiges Feature (siehe Abschnitt 3.8.2 auf Seite 61). Aus Zeitgründen wurde es deshalb bisher nicht umgesetzt.

Guard Die Klassen `Guard` und `ChallengeResponse` wurden so erweitert, dass Informationen über eine erfolgreiche Authentifizierung in der ChallengeResponse-Instanz der Anfrage gespeichert werden. Da das Restlet-Team zuerst Vorbehalte hatte, dies direkt im ChallengeResponse zu speichern, hat der Autor dieser Arbeit diese Informationen dann in einer standardmäßig dem Request angehängten Attribut-Map gespeichert. Später wurde der ursprünglich vorgesehene Entwurf vom Restlet-Team übernommen.

[2] Engine.parse(Logger, Form, String, CharacterSet, boolean, char)

TunnelFilter Die Änderungen des `TunnelService`s und des `TunnelFilter`s bzgl. des Extension-Mappings wurden wie geplant (siehe Abschnitt 5.7 auf Seite 99) vom Autor dieser Arbeit umgesetzt und an das Restlet-Team geschickt. Das Restlet-Team wollte das Extension-Mapping nur für die HTTP-Methoden GET und HEAD umsetzen, jedoch nicht für andere Methoden wie z. B. PUT. Dies ist semantisch sinnvoll: Der URI identifiziert die Ressource, nicht eine konkrete Repräsentation. Wenn die zur Änderung angesprochene Ressource eine virtuelle Dateiendung enthält, würde die Anfrage genau genommen nur die Repräsentation dieses Datenformates ersetzen oder löschen dürfen. Die Anfrage ändert / löscht jedoch – wie vorgesehen – die ganze Ressource. Jérôme Louvel vom Restlet-Team schlug stattdessen vor, diese Vorgaben über Query-Parameter zu machen. Über die virtuellen Dateiendungen wird z. Zt. in der JAX-RS-Expert-Group diskutiert [3].

Um die Anforderungen der JAX-RS-Spezifikation bzgl. der Content-Negotiation mit virtuellen Dateiendungen umzusetzen, musste ein Teil des `TunnelFilter`s aus der Noelios Restlet Engine kopiert und als eigener Filter in die JAX-RS-Erweiterung übernommen werden. Da die Klasse `TunnelFilter` ein Teil der Restlet-Engine ist, steht sie in der Restlet-API nicht zur Verfügung, weshalb nicht von ihr geerbt werden kann. Das entsprechende Feature der Restlet-Application muss abgeschaltet werden, damit das Extension-Mapping für GET- und HEAD-Anfragen nicht doppelt ausgeführt wird.

CacheControl Der Autor dieser Arbeit hoffte, dass das Restlet-Äquivalent zur JAX-RS-Klasse `CacheControl` vor dem Ende dieser Arbeit vom Restlet-Team implementiert wird, was jedoch nicht der Fall war. Aus zeitlichen Gründen hat der Autor dieser Arbeit dies bisher nicht umgesetzt.

[3] Diskussion über virtuelle Dateierweiterungen: [JSR311-Expert-Group 2008b]

6.3 Validierung und Tests

Um alle Tests gesammelt auf Knopfdruck ausführen zu können, wurden alle Tests wurden mit Unit-Tests (JUnit [4]) realisiert. So kann einfach geprüft werden, ob Änderungen wie gewünscht funktionieren und keine unerwarteten Seiteneffekte auftreten, so dass bisher implementierte Funktionalitäten weiter wie gewünscht funktionieren.
Die Tests stehen im Restlet-Repository zur Verfügung (im Test-Projekt org.restlet.tests).

Für alle Tests müssen die gleichen Initialisierungsschritte ausgeführt werden (JAX-RS-`ApplicationConfig` laden und Server starten). Auch der Zugriff auf die Ressource-Methode ist immer ähnlich. Deshalb wurde eine abstrakte Unterklasse der JUnit-Klasse `TestCase` mit dem Namen `JaxRsTestCase` implementiert. Sie startet vor dem Ausführen einer Test-Methode einen Restlet-Server mit der / den gewünschten Root-Ressource-Klasse(n). Diese Klasse stellt auch Methoden zur Verfügung, um Anfragen an den Test-Server möglichst einfach stellen zu können. Für die Anfragen werden die Funktionalitäten der Restlet-Klasse `Client` genutzt.

Im Laufe der Entwicklung sind gut 300 Test-Methoden entstanden. Die Ausführung aller Tests dauert mit echter TCP-Kommunikation (ServerSocket starten, Datenaustausch mit TCP, Sockets schließen etc.) entsprechend lange. Deshalb wurde aus der Klasse `JaxRsTestCase` eine Oberklasse `RestletServerTestCase` herausfaktorisiert, die durch das Setzen eines boolschen Schalters entweder ein richtiger TCP-ServerSocket startet oder alterativ die `JaxRsApplication` direkt aufruft.
In diesem Zusammenhang ist es sehr vorteilhaft, dass die zentrale Schnittstelle der Klassen `Client` und der `JaxRsApplication` gleich ist: Beide Klassen können mit der Methode `handle(Request, Response)` Anfragen bearbeiten. Ein `Client` serialisiert die Anfrage und schickt sie an einen Server, die `JaxRsApplication` beantworten die Anfrage mit den enthaltenen Ressource-Methoden selber. Auch Repräsentationen lassen sich unabhängig von der konkreten Unterklasse verarbeiten. So war die Klasse `RestletServerTestCase` einfach zu entwickeln.
Diese Klasse kann auch unabhängig von der JAX-RS-Erweiterung verwendet werden.

Eine zum Testen dieser Laufzeitumgebung taugliche Beispiel-Anwendung wurde im Rahmen dieser Arbeit nicht entwickelt. Hiermit würde im Wesentlichen die JAX-RS-API validiert. Dies ist zwar eine sinnvolle Aufgabe, liegt aber nicht im Zentrum dieser Arbeit. Die Restlet-Distribution (siehe auch nächsten Abschnitt) enthält viele Beispiele für die Nutzung der Restlet-API (Projekt org.restlet.examples). Sie wurde um zwei Beispiele zur Nutzung der JAX-RS-Erweiterung ergänzt. Eines dieser Beispiele ist in Auszügen im Anhang A.5 ab Seite 119 angeführt.

[4] www.junit.org; in Eclipse enthalten.

7 Fazit und Ausblick

7.1 Fazit

Die JAX-RS-Laufzeitumgebung wurde wie entworfen implementiert. Nur kleinere Details wurden aus Zeitgründen nicht fertig gestellt. Es konnte gezeigt werden, dass sich das entworfene Konzept umsetzen lässt und auch funktioniert.

Die JAX-RS-Erweiterung ist seit dem Milestone 3 der Restlet-Version 1.1 Bestandteil der Restlet-Distribution. Die aktuelle Version steht auf der Restlet-Webseite zum Download [1] bereit. Es ist zu berücksichtigen, dass die z. Zt. vorliegende JAX-RS-Laufzeitumgebung nur experimentell verwendet werden sollte, da die JAX-RS-API z. Zt. (Juli 2008) noch nicht endgültig beschlossen ist. Seit der Veröffentlichung des dieser Arbeit zu Grunde liegenden Public Review Draft im April 2008 wurden noch etliche Details geändert, so dass die hier entworfenen Laufzeitumgebung noch an die neuen Anforderungen angepasst werden muss.

7.2 Ausblick

7.2.1 Anpassungen für JAX-RS 1.0

Zwischen dem Public Review Draft vom April 2008 und dem Abschluss dieser Arbeit im Juli 2008 wurde die JAX-RS-Spezifikation in etlichen Details verändert und vermutlich werden noch weitere Änderungen folgen. Diese Änderungen müssen in die Anforderungen in Kapitel 4 übernommen, der Entwurf und dann die Implementierung entsprechend angepasst werden.

Für eine produktionstaugliche Version müssen außerdem die Funktionalitäten implementiert werden, deren Anforderungen beim Entwurf noch nicht ganz klar waren. Ebenso müssen die bisher nicht oder nur prototypischen umgesetzten Funktionalitäten vervollständigt werden:

[1] Download der Restlet-Distribution: http://www.restlet.org/downloads/

- Die Verarbeitung der in @Path enthaltenen Pfad-Teile muss den neuen Anforderungen entsprechend aktualisiert werden. Auch für mit @Context annotierte Parameter etc. vom Typ `PathSegment` und `List<PathSegment>` muss die Unterstützung implementiert werden. Zur Zeit [2] wird gerade diskutiert, ob und wie reguläre Ausdrücke direkt in die @Path-Annotation verwendet werden können. Hier wird also vermutlich noch eine weitere Änderung kommen.
- Die Änderung bzgl. der Möglichkeit der Veränderung der Kopfzeilen und dem Werfen einer WebApplicationException wurden bisher nicht umgesetzt (siehe Abschnitt 6.2 auf Seite 106). Dies muss noch nachgeholt werden.
- Die Restlet-Klasse für die HTTP-Kopfzeile „CacheControl“ und ihre Serialisierung und Deserialisierung muss umgesetzt werden.
- Die teilweise nur prototypisch umgesetzte URI-Dekodierung für das Interface `UriInfo` und die URI-Kodierung im `UriBuilder` muss vervollständigt werden.
- Die Authentifikation und die Rollenverwaltung der Connectoren werden bis jetzt noch nicht unterstützt. Auch wenn dieser Punkt nicht zur Funktionsfähgkeit der JAX-RS-Laufzeitumgebung nötig ist, sollte dies im Rahmen der Fortentwicklung der Restlet-API weiter verfolgt werden.

7.2.2 Client-API

Wenn eine Server-API definiert wird, liegt es immer nahe, auch eine Client-API nach ähnlichem Konzept zu definieren. Im Rahmen der Diskussionen in der Expert-Group wurde eine Client-API lange diskutiert (siehe Abschnitt 3.6.2 auf Seite 58). Da hierfür ebenfalls Zeit benötigt wird, wurde eine Client-API letztendlich vertagt [3].

Im Rahmen der JAX-RS-Referenz-Implementierung Jersey und des Projektes RESTEasy von JBoss wurden unterschiedliche Client-APIs definiert. Die API des RESTEasy-Projektes [4] nutzt dafür die JAX-RS-Annotationen, die Jersey-Client-API [5] traditionelle Methodenaufrufe. Diese beiden Implementierungen sind ein weiterer Hinweis, dass Bedarf für eine entsprechende API vorhanden ist. Diese könnte im Rahmen einer weiteren Arbeit entworfen und / oder implementiert werden.

[2] 3.7.2008, siehe [JSR311-Expert-Group 2008c]
[3] Diskussion zur Client-API im Oktober 2007: [JSR311-Expert-Group 2007b], vorl. Ende: [Hadley 2007c]; im März 2008: [JSR311-Expert-Group 2008a], endgültiges Ende der Planung: [Hadley 2008b]
[4] RESTEasy-Cient-API: [Burke 2008e]
[5] Jersey Client API: [Sandoz 2008a]

A Beispiele

Um eine praktische Vorstellung über den konkreten Ablauf der in dieser Arbeit dargestellten Kommunikationen zu bekommen, werden in diesem Anhang kleine Beispiele für RESTful Web-Dienst-Kommunikation gezeigt.

Danach folgen ab Seite 114 ein paar kurze Quellcode-Beispiele für

- mehrere kleine Restlet-Clients (ab Seite 114)
- einen Rumpf für die Implemetierung einer Ressource (Seite 117)
- einen Mini-Restlet-Server (Seite 118) und
- eine kleine JAX-RS-Anwendung (ab Seite 119).

Als Beispiel dient eine vereinfachte Personalverwaltung.

A.1 RESTful-Web-Service-Kommunikation

Der Eintrittspunkt in eine REST-Anwendung ist immer ein zentraler URI. Dies ist der einzige URI, den der Client kennen muss. In diesen Beispielen wird der URI http://staff.company.com/employees verwendet. Da die HTTPS-Kommunikation Einstellungen benötigen, die ohne Verschlüsselung nicht nötig sind, verwendet dieses Beispiel unverschlüsseltes HTTP.

Um sicherzustellen, dass nur authorisierte Personen Zugriff bekommen, sollte eine HTTP-Authentifizierung verwendet werden. Die Authentifizierung wird jedoch nicht explizit angeführt, um die Beispiele einfach zu halten.

Wenn ein Mitarbeiter der Personalabteilung von der zentralen Ressoure mit der HTTP-Methode GET ein XML-Dokument abruft, dann bekommt er bspw. eine Liste der Mitarbeiter, für die er zuständig ist. Diese würde dann von der Anwendung auf dem Rechner des Mitarbeiters dem Anwender zur Verfügung gestellt. Ein Beispiel für ein übertragenes Dokument wäre z. B. folgendes (nur HTTP-Antwort, ohne HTTP-Kopf):

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<employees>
   <employee>
      <persNo>1234</persNo>
      <firstname>Lucy</firstname>
      <lastname>Smith</lastname>
      <details>http://staff.company.com/employees/1234</details>
   </employee>
   <employee>
      <persNo>3210</persNo>
      <firstname>Jack</firstname>
      <lastname>Jonson</lastname>
      <details>http://staff.company.com/employees/3210</details>
   </employee>
   (...)
</employees>
```

Die Ressource enthält Verweise auf andere Ressourcen, in diesem Fall zu den einzelnen Angestellten. Um weitere Infos zu ihnen zu erhalten, ruft der Cient den zum Angestellten angegebenen URI mit GET auf (bspw. http://staff.company.com/employees/1234). Als Ergebnis bekommt er bspw. folgendes XML-Dokument (Antwort ohne HTTP-Kopf):

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<employee>
   <persNo>1234</persNo>
   <firstname>Lucy</firstname>
   <lastname>Smith</lastname>
   <sex>w</sex>
   <department>research</department>
   <departmentUri>
      http://staff.company.com/departments/research
   </departmentUri>
</employee>
```

Um einen neuen Angestellten in die Personalverwaltung einzutragen, wird ein HTTP-POST-Request an die Collection der Angestellten (http://staff.company.com/employees/) gesendet (Beispiel jetzt mit einem Teil des HTTP-Kopfes der Anfrage):

```
POST /employees/ HTTP/1.0
(...)

<?xml version="1.0" encoding="UTF-8" ?>
<employee>
   <firstname>William</firstname>
   <lastname>Wallace</lastname>
   <sex>w</sex>
   <department>research</department>
</employee>
```

Die übergebene Repräsentation wird dann als Unterpunkt zum angegebenen URI (Liste der Angestellten) angelegt, also als ein Angestellter. Die neue Personalnummer legt die Anwendung selber fest (z. B. 3456). Die Antwort lautet dann etwa:

```
HTTP/1.0 201 Created
Location: http://staff.company.com/employees/3456
(...)
```

Unter der angegebenen Location kann der neue Angestellte abgefragt werden.

Wird William Wallace in eine andere Abteilung versetzt, dann kann der Personalbearbeiter die aktuell gespeicherte Version mit PUT aktualisieren und dabei eine andere Abteilung angeben. Diesmal wird direkt der URL des Angestellten angesprochen, denn der Angestellte soll ja verändert werden:

```
PUT /employees/3456 HTTP/1.0
(...)

<?xml version="1.0" encoding="UTF-8" ?>
<employee>
   (...)
   <department>purchase</department>
</employee>
```

Verlässt ein Angestellter den Betrieb, wird er mit der HTTP-Methode DELETE gelöscht (oder ggf. ins Archiv verschoben):

```
DELETE /employees/3456 HTTP/1.0
(...)
```

A.2 Restlet-Client

Beispielcode A.1: Abfragen und Ausgeben einer Repräsentation mit Restlet

```
1  public class RestletClient
2  {
3     public static void main(String[] args) throws IOException
4     {
5        // Client-Objekt anlegen
6        Client client = new Client(Protocol.HTTP);
7
8        // Representation vom Server laden
9        Response response = client.get("http://www.restlet.org");
10       Representation representation = response.getEntity();
11
12       // Ausgabe der Repräsentation
13       representation.write(System.out);
14    }
15 }
```

Die Beispiele auf den folgenden Seite nutzen das Restlet-Framework, um

- eine XML-Repräsentation von einem Server auszulesen und daraus mit XPath einen URI abzufragen und
- eine Ressource auf einem Server anzulegen.

Beispielcode A.2: Auslesen eines URIs aus einer XML-Representation

```
1
2  static String EMPL_URI = "http://staff.company.com/employees";
3
4  /**
5   * @return URI der Details des/der ersten Angestellten mit dem
6   *         angegebenen Nachnamen.
7   */
8  static String readDetailUri(String lastname) throws Exception
9  {
10    // Anfrage erzeugen
11    Request request = new Request(Method.GET, EMPL_URI);
12    // Authentifikation
13    ChallengeResponse auth = new ChallengeResponse(
14                 ChallengeScheme.HTTP_BASIC, "mamu", "password");
15    request.setChallengeResponse(auth);
16
17    // akzeptiertes Datenformat
18    request.getClientInfo().getAcceptedMediaTypes().add(
19                  new Preference<MediaType>(MediaType.TEXT_XML));
20
21    // Client erzeugen und Anfrage stellen
22    Client client = new Client(Protocol.HTTP);
23    Response response = client.handle(request);
24
25    // DOM-XML-Repräsentation abfragen
26    XmlRepresentation output = response.getEntityAsDom();
27
28    // URI des Angestellten mit XPath auslesen
29    String xPathAnfr = "/employees/employee"
30                 + "[lastname/text()=\"" + lastname + "\"]/details";
31    String uri;
32    uri = (String)output.evaluate(xPathAnfr, XPathConstants.STRING);
33    System.out.println("URI der Details: "+uri);
34                     // -> "http://staff.company.com/employees/1234"
35    return uri;
36 }
```

Beispielcode A.3: Hinzufügen einer Ressource

```
1
2  static String EMPL_URI = "http://staff.company.com/employees";
3
4  /**
5   * Legt einen Angestellten an.
6   * @param auth Authentifikationsdaten
7   * @param employee Daten des Angestellten in einem vom Server
8   *                 verstandenen Datenformat.
9   * @return URI des neu angelegten Benutzers
10  */
11 static Reference createUser(ChallengeResponse auth,
12    Representation employee)
13 {
14    Request postRequ = new Request(Method.POST, EMPL_URI);
15    postRequ.setEntity(employee);
16    postRequ.setChallengeResponse(auth);
17    Client client = new Client(Protocol.HTTP);
18    Response postAntw = client.handle(postRequ);
19
20    if(postAntw.getStatus().isSuccess())
21       System.out.println("Angestellten angelegt.");
22    else
23       System.out.println("Angestellten nicht angelegt.");
24    System.out.println("HTTP-Status: " + postAntw.getStatus());
25                                                  // -> Created (201)
26    Reference employeeUri = postAntw.getLocationRef();
27    System.out.println("URI des Angestellten: " + employeeUri);
28                    // -> "http://staff.company.com/employees/3456"
29    return employeeUri;
30 }
```

A.3 Restlet-Ressource

Dieses Beispiel zeigt den Rumpf einer mit der Restlet-API implementierten Ressource, die die Datenformate XML und HTML unterstützt. [Noelios 2008b] zeigt, wie eine Resource in einem Restlet-Server gestartet wird.

Beispielcode A.4: Beispiel einer Restlet-Resource

```
1  public class EmployeeResource extends Resource{
2     public EmployeeResource(Context c, Request req, Response resp) {
3        super(c, req, resp);
4        getVariants().add(new Variant(MediaType.TEXT_HTML));
5        getVariants().add(new Variant(MediaType.TEXT_XML));
6        getVariants().add(new Variant(MediaType.APPLICATION_XML));
7     }
8
9     /** Returns a full representation for a given variant. */
10    @Override
11    public Representation represent(Variant variant) {
12       Employee employee = ...;
13       if(variant.getMediaType().equals(MediaType.TEXT_HTML)) {
14          StringBuilder html = ...; // build HTML from employee
15          return new StringRepresentation(html, MediaType.TEXT_HTML);
16       }
17       else // could only be XML, because HTML was handled. No other
18       { // than these 2 media types are supported by this resource.
19          return new JaxbRepresentation(employee,
20             variant.getMediaType()); // text/xml or application/xml
21       }
22    }
23
24    /** Update the employee. Is called, if a PUT was received. */
25    @Override
26    public void storeRepresentation(Representation entity)
27          throws ResourceException {
28       if(entity.getMediaType().equals(MediaType.TEXT_XML) ||
29          entity.getMediaType().equals(MediaType.APPLICATION_XML)) {
30          JaxbRepresentation emplRepr =
31                      new JaxbRepresentation(entity, Employee.class);
32          Employee employee = emplRepr.getObject();
33          ...; // save to the database
34       } else {
35          throw new ResourceException(
36                         Status.CLIENT_ERROR_UNSUPPORTED_MEDIA_TYPE);
37       }
38    }
39 }
```

A.4 Restlet-Server

Hier folgt ein kurzes Beispiel für einen ganz einfachen, aber vollständigen Restlet-Server. Er zeigt eine sehr einfache Verarbeitung einer Anfrage mit einem Restlet. – Um eine Ressource zu modellieren, würde man normalerweise eine Unterklasse von `Resource` implementieren (siehe vorige Seite). Dieses Beispiel soll jedoch zeigen, wie die Verarbeitung innerhalb eines Restlets funktioniert. – In den Klassen-Pfad müssen die Restlet-API (org.restlet.jar) und die Noelios Restlet Engine (com.noelios.restlet.jar) aufgenommen werden.

Beispielcode A.5: Beispiel-Restlet

```
1  public class ExampleRestlet extends Restlet {
2     public void handle(Request request, Response response) {
3        if(request.getMethod().equals(Method.GET))
4        { // check, if the HTTP method is GET
5           response.setEntity(new StringRepresentation(
6                              "Hello World!", MediaType.TEXT_PLAIN));
7        }
8        else
9        { // other methods are not allowed -> Return HTTP status 405
10          response.setStatus(Status.CLIENT_ERROR_METHOD_NOT_ALLOWED);
11       }
12    }
13 }
14
15 public class HelloWorldServer {
16    public static void main(String[] args) throws Exception {
17       // Create a new Restlet component
18       Component component = new Component();
19       // add a HTTP server connector to it
20       component.getServers().add(Protocol.HTTP, 8182);
21
22       // Create a new Restlet
23       Restlet restlet = new ExampleRestlet(); // see above
24       // attach it to the default host
25       component.getDefaultHost().attach("/helloWorld", restlet);
26
27       component.start();
28       System.out.println("Press any key to stop the server ...");
29       System.in.read();
30       component.stop();
31    }
32 }
```

Weitere Beispiele finden sich im Restlet-Tutorial [Noelios 2008c].

A.5 JAX-RS

Die beiden folgenden JAX-RS-Ressource-Klassen (erst die Root-Ressource-Klasse, dann die Ressource-Klasse für einzelne Angestellte) implementiert die XML-Kommunikation aus dem Beispiel für die REST-Web-Dienst-Kommunikation aus Abschnitt A.1 auf Seite 111. Da diese Laufzeitumgebung z. Zt. kein Entity-Provider zum Lesen von JSON-Daten enhält, wird dies z. Zt. noch nicht voll unterstützt. Ein Anwendungsentwickler kann einen solchen Provider auch selber implementieren. Diese Klassen zeigen auch, wie die gleichen Ressource-Klassen auch HTML unterstützen können.

Beispielcode A.6: Beispiel für eine JAX-RS-Root-Ressource-Klasse

```
1  @Path("employees")
2  public class EmployeesResource
3  {
4     /** The EmployeeMgr manages the data storage */
5     private EmployeeMgr employeeMgr = EmployeeMgr.get();
6
7     /** the UriInfo is injected after creating */
8     @Context
9     private UriInfo uriInfo;
10
11    /** @return the employees -> will get converted with JAXB to XML */
12    @GET
13    @ProduceMime({"application/xml", "text/xml"})
14    public EmployeeList getEmployees() {
15       EmployeeList employees = employeeMgr.getAll();
16       // set detail URIs
17       UriBuilder uriBuilder = uriInfo.getPlatonicRequestUriBuilder();
18       uriBuilder.path("{staffNo}");
19       uriBuilder.extension(uriInfo.getPathExtension());
20       for (SmallEmployee empl : employees)
21          employee.setDetails(uriBuilder.build(empl.getStaffNo()));
22       return employees;
23    }
24
25    /** Creates a new employee from XML */
26    @POST
27    @ConsumeMime({"application/xml", "text/xml"})
28    public Response createEmployee(Employee employee) {
29       int staffNo = employeeMgr.createEmployee(employee);
30       String uriExts = uriInfo.getPathExtension();
31       URI location = createdLocation(staffNo, uriExts);
32       return Response.created(location).build();
33    }
```

```
34      /**
35       * Creates the URI for the location of an created employee.
36       * @param staffNo the number of the created employee
37       * @param ext the file extension to use for content negotiation
38       * @return the URI for the location of an created employee.
39       */
40      private URI createdLocation(int staffNo, String ext) {
41         // this is needed for JAX-RS 0.8, but improved in 0.9
42         UriBuilder locBuilder = uriInfo.getPlatonicRequestUriBuilder();
43         locBuilder.path("{staffNo}").extension(ext);
44         return locBuilder.build(staffNo);
45      }
46
47      @POST
48      @ConsumeMime("application/x-www-form-urlencoded")
49      public Response createEmployee(
50            MultivaluedMap<String, String> employeeData) {
51         Employee employee = new Employee();
52         employee.setFirstname(employeeData.getFirst("firstname"));
53         employee.setLastname(employeeData.getFirst("lastname"));
54         employee.setSex(employeeData.getFirst("sex"));
55         employee.setDepartment(employeeData.getFirst("department"));
56         int persNo = employeeMgr.createEmployee(employee);
57         URI location = createdLocation(persNo, "html");
58         return Response.seeOther(location).build();
59      }
60
61      /** Create sub resource for one concrete employee. */
62      @Path("{staffNo}")
63      public EmployeeResource getSub(
64            @PathParam("staffNo") int staffNo) {
65         if (!employeeMgr.exists(staffNo))
66            throw new WebApplicationException(Response.Status.NOT_FOUND);
67         return new EmployeeResource(staffNo);
68      }
69
70      @GET
71      @ProduceMime("text/html")
72      public StreamingOutput getListAsHtml() {
73         final EmployeeList employees = getEmployees();
74         // create HTML-Representation with links to the employees
75         return ...; // the code is available in the Restlet Examples.
76      }
77   }
```

Hier folgt die dazugehörige Ressource-Klasse für einen einzelnen Angestellten:

Beispielcode A.7: Beispiel für eine JAX-RS-Ressource-Klasse

```
1  public class EmployeeResource
2  {
3     /** The EmployeeMgr manages the data storage */
4     private EmployeeMgr employeeMgr = EmployeeMgr.get();
5
6     private int staffNo;
7
8     EmployeeResource(int persNr) {
9        this.staffNo = persNr;
10    }
11
12    /** Returns the employee, will be converted to XML. */
13    @GET
14    @ProduceMime({"application/xml", "text/xml"})
15    public Employee get(@Context UriInfo uriInfo) {
16       // load employee with requested id
17       Employee employee = employeeMgr.getFull(staffNo);
18       // set department uri
19       UriBuilder departmentUB = uriInfo.getBaseUriBuilder();
20       departmentUB.path("departments", "{depId}");
21       departmentUB.extension(uriInfo.getPathExtension());
22       String department = employee.getDepartment();
23       employee.setDepartmentUri(departmentUB.build(department));
24       return employee;
25    }
26
27    /** Updates this employee. It will be read from XML. */
28    @PUT
29    @ConsumeMime({"application/xml", "text/xml"})
30    public void update(Employee employee) {
31       employeeMgr.update(staffNo, employee);
32    }
33
34    @DELETE
35    public Object delete(@Context HttpHeaders httpHeaders,
36          @Context UriInfo uriInfo) {
37       employeeMgr.remove(staffNo);
38       if(httpHeaders.getAcceptableMediaTypes().contains(
39             MediaType.TEXT_HTML_TYPE))
40          return Response.seeOther(createEmployeesUri(uriInfo));
41       return null;
42    }
43
```

```
44      /** This method creates a URI to the employees list */
45      private static URI createEmployeesUri(final UriInfo uriInfo) {
46         UriBuilder employeesUri = uriInfo.getBaseUriBuilder();
47         employeesUri.path(uriInfo.getAncestorResourceURIs().get(0));
48         // this is needed for JAX-RS 0.8, but improved in 0.9
49         employeesUri.extension(uriInfo.getPathExtension());
50         URI build = employeesUri.build();
51         return build;
52      }
53
54      /** Returns the employee as HTML document */
55      @GET
56      @ProduceMime("text/html")
57      public StreamingOutput getHtml(@Context final UriInfo uriInfo) {
58         final Employee employee = get(uriInfo);
59         final URI employeesUri = createEmployeesUri(uriInfo);
60         // creates a HTML Repräsentation of the employee with
61         // a link to the list of employees.
62         return new StreamingOutput() {
63            public void write(OutputStream output) throws IOException {
64               PrintStream ps = new PrintStream(output);
65               ps.println("<html><head>");
66               ps.println("<title>Employee"+staffNo+"</title>");
67               ps.println("</head></body>");
68               ps.println("<h2>Employee</h2>");
69               ps.println(employee.getFirstname());
70               ps.println(employee.getLastname());
71
72               // ...
73               // add HTML <form> to update the employee
74               // the full code is available in the Restlet examples.
75               // ...
76
77               // creates a link to the employees collection
78               ps.print("<hr><a href=\"");
79               ps.print(employeesUri);
80               ps.print("\">all employees</a>");
81               ps.println("</body></html>");
82            }
83         };
84      }
85   }
```

Dieses Beispiel findet sich auch in der Restlet-Distribution im Package org.restlet.example.jaxrs.

Glossar

Kürzel	Beschreibung	
Authentifizierung	Die Authentifizierung ist die Überprüfung einer vorgegebenen Identität, z. B. mit Hilfe eines Passwortes.	66
Bean-Setter	Eine Java-Bean ist ein Objekt, das u. a. für seine Attribute setter-Methoden nach der Konvention Attributname = xxx -> Setter ist setXxx(Typ) zur Verfügung stellt. Diese Setter heißen Bean-Setter	47
Expert-Group	Im Rahmen des Java Community Process (siehe dort) finden die entscheidenden Diskussionen in der sogenannten Expert-Group statt.	57
Geschäftsobjekt	Ein Geschäftsobjekt ist ein im fachlichen Betriebsablauf benötigtes Objekt, z. B. ein Kunde, eine Adresse o. ä.	28
IETF	Die IETF (Abkürzung für *Internet Engineering Task Force*) ist eine Organisation, die sich mit der technischen Weiterentwcklung des Internets beschäftigt	6
Java Community Process	Der Java Community Process beschreibt, wie Java Specification Requests bearbeitet werden, siehe http://jcp.org	43
Java Specification Request	Im Java Community Process werden neue Java-Spezifikationen definiert. Eine Anfrage für eine neue Spezifikation heißt 'Java Specification Request'.	43
JCP	Abkürzung für *Java Community Process*, siehe dort.	54
JDBC	JDBC (Abkürzung für *Java Database Connectivity*) ist eine standardisierte Datenbank-Schnittstelle in Java.	38
JSR	Abkürzung für *Java Specification Request*, siehe dort.	43

Kürzel	**Beschreibung**	
Laufzeitumgebung	Eine Laufzeitumgebung stellt sämtliche Logik bereit, die ein Programmteil benötigt, um darin zu laufen. Aufgabe dieser Masterarbeit ist es, eine Laufzeitumgebung für JAX RS-Anwendungen zu entwickeln.	80
Load-Balancing	Load-Balancing (dt. Lastverteilung) ermöglicht es, für einen Dienst mehrere Server zu installieren. Die Lastverteilung sorgst dafür, dass alle Server gleichmäßig ausgelastet werden.	14
Masterkopie	Die Masterkopie ist der authoritative Wert eines Attributs, einer Tabelle, Datenbank o. ä. bei verteilten Systemen. Andere Systeme erhalten Kopien, die ggf. veralten. Änderungen müssen insbesonders bei der Masterkopie erfolgen.	10
orchestrieren	Orchestrieren bedeutet, dass mehrere Dienste zu einem – typischerweise höherwertigeren – Dienst kombiniert werden.	18
POJO	Ein Plain Old Java Object (POJO) ist ein Objekt, welches keine Abhängigkeiten zu Interfaces oder Oberklassen von Frameworks benötigt. Die EJB-2.x-Spezifikation bspw. verlangt die Implementierung bestimmter Interfaces; die Restlet-API verlangt das Erben von Klassen aus der Restlet-API	43
URI-dekodiert	Da in einer URI viele Zeichen eine besondere Bedeutung haben (z. B. der Slash, der Doppelpunkt, das Fragezeichen oder das Gleichheitszeichen; das Leerzeichen ist ganz verboten), aber manchmal trotzdem übertragen werden sollen, werden diese Zeichen URI-kodiert. Dazu wird statt dem Zeichen ein Prozentzeichen und der hexadezimale Wert übertragen. Anstatt eines Leerzeichen wird dann %20 übertragen.	68

Kürzel	**Beschreibung**	
Virtueller Host	Ein Virtueller Host ermöglicht es, dass ein Server unter einer IP-Adresse mehrere Domains hosten kann. Die Unterscheidung der Domains erfolgt über die in HTTP/1.1 zwingend vorgeschriebene HTTP-Kopfzeile „Host“.	33
W3C	Das W3C (Abkürzung für *World Wide Web Consortium*) ist das Gremium, das Technologien des World Wide Web standardisiert. Homepage: www.w3c.org	6
Zugriffskontrolle	Die Zugriffskontrolle ist die Prüfung, ob ein authentifizierter Client eine bestimmte Funktion ausführen darf.	66

Abbildungsverzeichnis

Listings

Literaturverzeichnis

[Amazon 2007] : *Amazon Simple Storage Service (Amazon S3)*. 2007 – URL http://www.amazon.com/S3-AWS-home-page-Money/b/ref=sc_fe_c_0_15763381_5?ie=UTF8&node=16427261&no=15763381&me=A36L942TSJ2AJA (Zugriff: 26.11.2007)

[Apache 2007a] APACHE: *JAX-RS (JSR-311) - Understanding the basics*. September 2007 – URL http://cwiki.apache.org/CXF20DOC/jax-rs-jsr-311.html (Zugriff: 8.1.2008). – War am 8.3.2008 nicht mehr auf aktuellem Stand.

[Apache 2007b] APACHE: *RESTful Services*. September 2007 – URL http://cwiki.apache.org/CXF20DOC/restful-services.html (Zugriff: 8.1.2008)

[Baker 2007] BAKER, Mark: *REST Discussion Mailing List – Re: Sun proposes to apply Web service standardization principles to REST*. Februar 2007 – URL http://tech.groups.yahoo.com/group/rest-discuss/message/7837 (Zugriff: 1.11.2007)

[Batovanja 1998/99] BATOVANJA, Antonio: *HTTP - Hypertext Transfer Protocol, Gegenüberstellung verschiedener Versionen*. 1998/99 – URL http://toni.to/HTTP_seminar.html (Zugriff: 3.7.2007)

[Bayer 2007] BAYER, Thomas: *sqlREST*. März 2007 – URL http://sqlrest.sourceforge.net/ (Zugriff: 18.9.2007)

[Berners-Lee 1991] BERNERS-LEE, Tim: *The Original HTTP as defined in 1991*. 1991 – URL http://www.w3.org/Protocols/HTTP/AsImplemented.html (Zugriff: 28.6.2007). – beschreibt HTTP/0.9

[Berners-Lee 1996] BERNERS-LEE, Tim: *Matrix URIs - Ideas about Web Architecture*. Dezember 1996 – URL http://www.w3.org/DesignIssues/MatrixURIs.html (Zugriff: 20.11.2007)

[Berners-Lee 1999a] BERNERS-LEE, Tim: *Weaving the Web*. Harper San Francisco, Oktober 1999. – ISBN 978-0062515865. – deutsche Übersetzung: [Berners-Lee 1999b]

[Berners-Lee 1999b] BERNERS-LEE, Tim: *Der Web-Report*. Econ, 1999. – ISBN 3-430-11468-3. – deutsche Übersetzung von „Weaving the Web“ [Berners-Lee 1999a]

[Bjorg 2007] BJORG, Steve: *Introduction to REST*. Mai 2007 – URL http://wiki.opengarden.org/@api/deki/files/357 (Zugriff: 13.9.2007)

[Blinksale 2007] : *Blinksale API Reference*. März 2007 – URL http://www.blinksale.com/api (Zugriff: 26.11.2007)

[Bowman 2006] BOWMAN, Eric J.: *REST Discussion Mailing List – Re: Matrix URIs*. September 2006 – URL http://tech.groups.yahoo.com/group/rest-discuss/message/6572 (Zugriff: 20.11.2007)

[Braun 2007] BRAUN, Heiko: *Mailing-List dev@jsr311.dev.java.net – JAX-RS Client API*. Oktober 2007 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=708 (Zugriff: 29.04.2008)

[Burke 2007a] BURKE, Bill: *JSR 311, JAX-RS: Comment and suggestions*. Oktober 2007 – URL http://bill.burkecentral.com/2007/10/01/jsr-311-jax-rs-comment-and-suggestions (Zugriff: 30.10.2007)

[Burke 2007b] BURKE, Bill: *RESTEasy*. Februar 2007 – URL http://sourceforge.net/projects/resteasy/ (Zugriff: 05.04.2008)

[Burke 2008a] BURKE, Bill: *JBoss Resteasy*. März 2008 – URL http://wiki.jboss.org/wiki/RESTeasy (Zugriff: 05.04.2008)

[Burke 2008b] BURKE, Bill: *Mailing-List dev@jsr311.dev.java.net – Response isn't adequate*. Februar 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=899 (Zugriff: 29.04.2008)

[Burke 2008c] BURKE, Bill: *Mailing-List dev@jsr311.dev.java.net – Response isn't adequate*. März 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=1018 (Zugriff: 29.04.2008)

[Burke 2008d] BURKE, Bill: *Mailing-List dev@jsr311.dev.java.net – @Target of FIELD for @*Param annotations*. März 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=999 (Zugriff: 29.04.2008)

[Burke 2008e] BURKE, Bill: *Resteasy Client Framework*. Februar 2008 – URL http://wiki.jboss.org/wiki/RESTeasyClientFramework (Zugriff: 03.07.2008)

[Burke 2008f] BURKE, Bill: *Resteasy JAX-RS*. März 2008 – URL http://wiki.jboss.org/wiki/RESTeasyJAXRS (Zugriff: 05.04.2008)

[Chinnici und Shannon 2007] CHINNICI, Roberto ; SHANNON, Bill: *Java Platform, Enterprise Edition 6 (Java EE 6) Specification*. Juli 2007 – URL http://jcp.org/en/jsr/detail?id=316

[CXF 2008] CXF, Apache: *Apache CXF: An Open Source Service Framework*. April 2008 – URL http://cxf.apache.org/

[Diephouse 2007] DIEPHOUSE, Daniel: *JSR 311 - javax.ws.rest*. Februar 2007 – URL http://netzooid.com/blog/2007/02/14/jsr-311-javaxwsrest (Zugriff: 30.10.2007)

[Dostal u. a. 2005] DOSTAL, Wolfgang ; JECKLE, Mario ; MELZER, Ingo ; ZENGLER, Barbara: *Service-orientierte Architekturen mit Web Services. Konzepte - Standards - Praxis*. 1. Auflage. Spektrum Akademischer Verlag, 2005. – ISBN 3827414571

[Fielding 2003a] FIELDING, Roy: *REST Discussion Mailing List – Re: cgi generating HTML, do you have a REST example ?* Juli 2003 – URL http://tech.groups.yahoo.com/group/rest-discuss/message/3714 (Zugriff: 28.8.2007)

[Fielding 2003b] FIELDING, Roy: *REST Discussion Mailing List – Re: ReST and Representing Mailboxes*. Januar 2003 – URL http://tech.groups.yahoo.com/group/rest-discuss/message/3181 (Zugriff: 28.8.2007)

[Fielding 2006] FIELDING, Roy: *REST Discussion Mailing List – Request Header vs. Query String*. April 2006 – URL http://tech.groups.yahoo.com/group/rest-discuss/message/5857 (Zugriff: 18.9.2007)

[Fielding 2007a] FIELDING, Roy: *Apache – Mailing list archive: Re: JSR 311 - Java API for RESTful Web Services*. Februar 2007 – URL http://mail-archives.apache.org/mod_mbox/www-jcp-open/200702.mbox/%3c7AB99AA1-BEA6-498B-BC14-EB49BE40260B@gbiv.com%3e (Zugriff: 25.10.2007)

[Fielding 2007b] FIELDING, Roy: *REST Discussion Mailing List – Re: REST intro slides*. Mai 2007 – URL http://tech.groups.yahoo.com/group/rest-discuss/message/8377 (Zugriff: 12.9.2007)

[Fielding 2007c] FIELDING, Roy: *REST Discussion Mailing List – Re: Sun proposes to apply Web service standardization principles to REST*. Februar 2007 – URL http://tech.groups.yahoo.com/group/rest-discuss/message/7858 (Zugriff: 25.10.2007)

[Fielding 2007d] FIELDING, Roy T.: *Mailing-List dev@jsr311.dev.java.net – Welcome to JSR 311*. April 2007 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=144 (Zugriff: 29.04.2008)

[Fielding 2007e] FIELDING, Roy T.: *Mailing-List dev@jsr311.dev.java.net – Welcome to JSR 311*. April 2007 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=146 (Zugriff: 29.04.2008)

[Fielding 1999] FIELDING, Roy T.: *Software Architectural Styles for Network-based Applications*. Juli 1999 – URL http://www.ics.uci.edu/~taylor/ics280e/Fielding%20arch_survey.pdf (Zugriff: 8.6.2007). – Draft 1.1

[Fielding 2000] FIELDING, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, Dissertation, 2000 – URL http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf (PDF, Zugriff: 18.03.2005) http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (HTML, Zugriff: 18.03.2005)

[Gerlach 2006] GERLACH, Martin: *Entwicklung eines Transaktionsframeworks für mobile Web Services*. August 2006 – URL http://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/master/gerlach.pdf (Zugriff: 29.3.2007)

[Gomba-Team 2007] GOMBA-TEAM: *gomba - Java RESTful web services*. Mai 2007 – URL http://gomba.sourceforge.net/ (Zugriff: 26.9.2007)

[Gregorio 2007] GREGORIO, Joe: *REST Discussion Mailing List – Re: Sun proposes to apply Web service standardization principles to REST*. Februar 2007 – URL http://tech.groups.yahoo.com/group/rest-discuss/message/7866 (Zugriff: 25.10.2007)

[Gregorio u. a. 2008] GREGORIO, Joe ; CLINTON, DeWitt ; NOTTINGHAM, Mark ; HADLEY, Marc ; ORCHARD, Dave ; SNELL, James: *URI Templates*. März 2008 – URL http://bitworking.org/projects/URI-Templates/spec/draft-gregorio-uritemplate-03.html

[Hadley 2007a] HADLEY, Marc: *JSR 311: Java API for RESTful Web Services*. Februar 2007 – URL http://weblogs.java.net/blog/mhadley/archive/2007/02/jsr_311_java_ap.html (Zugriff: 16.10.2007)

[Hadley 2007b] HADLEY, Marc: *Mailing-List dev@jsr311.dev.java.net – Container Independence*. April 2007 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=73 (Zugriff: 29.04.2008)

[Hadley 2007c] HADLEY, Marc: *Mailing-List dev@jsr311.dev.java.net – JAX-RS Client API*. Oktober 2007 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=714 (Zugriff: 29.04.2008)

[Hadley 2007d] HADLEY, Marc: *Mailing-List dev@jsr311.dev.java.net – Title change to JAX-RS*. April 2007 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=130 (Zugriff: 29.04.2008)

[Hadley 2007e] HADLEY, Marc: *Mailing-List dev@jsr311.dev.java.net – Welcome to JSR 311*. April 2007 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=142 (Zugriff: 29.04.2008)

[Hadley 2008a] HADLEY, Marc: *JAX-RS Implementations*. April 2008 – URL http://weblogs.java.net/blog/mhadley/archive/2008/04/jaxrs_implement.html (Zugriff: 01.04.2008)

[Hadley 2008b] HADLEY, Marc: *Mailing-List dev@jsr311.dev.java.net – Client API*. März 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=1054 (Zugriff: 29.04.2008)

[Hadley 2008c] HADLEY, Marc: *Mailing-List dev@jsr311.dev.java.net – @*Param and List<?>*. März 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=1121 (Zugriff: 29.04.2008)

[Hadley 2008d] HADLEY, Marc: *Mailing-List dev@jsr311.dev.java.net – Problem with StreamedOutput model*. März 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=1094 (Zugriff: 29.04.2008)

[Hadley 2008e] HADLEY, Marc: *Mailing-List dev@jsr311.dev.java.net – Response isn't adequate*. Februar 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=911 (Zugriff: 29.04.2008). – siehe auch https://jsr311.dev.java.net/source/browse/jsr311/trunk/src/jsr311-api/src/javax/ws/rs/core/HttpResponseContext.java?rev=7&view=markup

[Hadley 2008f] HADLEY, Marc: *Mailing-List dev@jsr311.dev.java.net – @Target of FIELD for @*Param annotations*. März 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=1001 (Zugriff: 29.04.2008)

[Hadley 2008g] HADLEY, Marc: *Mailing-List users@jsr311.dev.java.net – more comments to JAX-RS*. März 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=users&msgNo=177 (Zugriff: 22.04.2008)

[Hadley 2008h] HADLEY, Marc: *Mailing-List users@jsr311.dev.java.net – Some proposals / questions for JSR 311*. Januar 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=users&msgNo=43 (Zugriff: 22.04.2008)

[Hadley 2008i] HADLEY, Marc: *Mailing-List users@jsr311.dev.java.net – UriInfo.getAncestorResource*()*. März 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=users&msgNo=149 (Zugriff: 22.04.2008)

[Hadley und Sandoz 2007a] HADLEY, Marc ; SANDOZ, Paul: *JAX-RS: Java API for RESTful Web Services*. März 2007 – URL https://jsr311.dev.java.net/drafts/spec20070330.pdf (Zugriff: 17.6.2008). – Editors Draft

[Hadley und Sandoz 2007b] HADLEY, Marc ; SANDOZ, Paul: *JSR 311: JAX-RS: The JavaTM API for RESTful Web Services*. Oktober 2007 – URL http://jcp.org/en/jsr/detail?id=311 (Zugriff: 29.10.2007)

[Hadley und Sandoz 2008a] HADLEY, Marc ; SANDOZ, Paul: *JAX-RS: Java API for RESTful Web Services*. April 2008 – URL http://jcp.org/aboutJava/communityprocess/pr/jsr311/index.html (Zugriff: 18.04.2008). – JCP Public Review Draft

[Hadley und Sandoz 2008b] HADLEY, Marc ; SANDOZ, Paul: *JAX-RS: Java API for RESTful Web Services*. Januar 2008 – URL https://jsr311.dev.java.net/drafts/spec20080131.pdf (Zugriff: 01.02.2008). – Spezifikation Draft

[Hadley und Sandoz 2008c] HADLEY, Marc ; SANDOZ, Paul: *JAX-RS: Java API for RESTful Web Services*. März 2008 – URL https://jsr311.dev.java.net/drafts/spec20080320.pdf (Zugriff: 20.03.2008). – Editors Draft

[Hadley und Sandoz 2008d] HADLEY, Marc ; SANDOZ, Paul: *JSR 311 – Project Home*. Juni 2008 – URL https://jsr311.dev.java.net/ (Zugriff: 30.6.2008)

[Harold 2007a] HAROLD, Elliotte: *REST Discussion Mailing List – Another potential problem with the 311 proposal*. Februar 2007 – URL http://tech.groups.yahoo.com/group/rest-discuss/message/7871 (Zugriff: 06.05.2008)

[Harold 2007b] HAROLD, Elliotte: *REST Discussion Mailing List – Sun proposes to apply Web service standardization principles to REST*. Februar 2007 – URL http://tech.groups.yahoo.com/group/rest-discuss/message/7830 (Zugriff: 25.10.2007)

[Harold 2007c] HAROLD, Elliotte R.: *Java News from Thursday, November 1, 2007*. November 2007 – URL http://www.cafeaulait.org/oldnews/news2007November1.html

[Hübner 2007] HÜBNER, Martin: *Folien zur Vorlesung Rechnernetze*. 2007 – URL https://users.informatik.haw-hamburg.de/home/pub/prof/huebner/Rechnernetze/RN-SS07-Kap1.pdf (Zugriff: 19.1.2007). – Zugang passwortgeschützt für Mitglieder des Studiendepartments Informatik der HAW Hamburg

[He 2003] HE, Hao: *What Is Service-Oriented Architecture*. September 2003 – URL http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html (Zugriff: 2.5.2007)

[Hess u. a. 2006] HESS, Andreas ; HUMM, Bernhard ; VOSS, Markus: Regeln für serviceorietierte Architekturen hoher Qualitaet. In: *Informatik Spektrum* (2006), Dezember, S. 395–410

[de hÓra 2007] HÓRA, Bill de: *Servlets+JSP design answers*. August 2007 – URL http://www.dehora.net/journal/2007/08/servletsjsp_design_answers.html (Zugriff: 5.11.2007)

[HTML 2.0 1996] CONNOLLY, Daniel W.: *Public Text of the HTML 2.0 Specification*. September 1996 – URL http://www.w3.org/MarkUp/html-spec/html-pubtext.html

[Jacobs und Walsh 2004] JACOBS, Ian ; WALSH, Norman: *Architecture of the World Wide Web, Volume One*. Dezember 2004 – URL http://www.w3.org/TR/2004/REC-webarch-20041215/ (Zugriff: 8.6.2007)

[James 2002] JAMES, Paul: *Representational State Transfer (REST)*. Mai 2002 – URL http://www.peej.co.uk/articles/rest.html (Zugriff: 27.11.2007)

[James 2005] JAMES, Paul: *3 Tiered REST Architecture*. März 2005 – URL http://www.peej.co.uk/articles/3-tiered-rest-architecture.html (Zugriff: 27.8.2007)

[Jersey 2007] : *GlassFish – Jersey*. 2007 – URL https://jersey.dev.java.net/ (Zugriff: 25.10.2007)

[Jouravlev 2004] JOURAVLEV, Michael: *Redirect After Post*. August 2004 – URL http://www.theserverside.com/tt/articles/article.tss?l=RedirectAfterPost (Zugriff: 18.8.2007)

[JSR311-Expert-Group 2007a] JSR311-EXPERT-GROUP: *Container Independence und Dual dispatch*. April 2007 – URL https://jsr311.dev.java.net/servlets/Brow-

seList?listName=dev&from=732959&to=732959&count=22&by=thread&paged=false (Zugriff: 4.6.2008)

[JSR311-Expert-Group 2007b] JSR311-EXPERT-GROUP: *JAX-RS Client API*. Oktober 2007 – URL https://jsr311.dev.java.net/servlets/BrowseList?list=dev&by=thread&from=934559 (Zugriff: 22.4.2008)

[JSR311-Expert-Group 2007c] JSR311-EXPERT-GROUP: *Mapping POJOs*. April 2007 – URL https://jsr311.dev.java.net/servlets/BrowseList?listName=dev&from=724102&to=724102&count=103&by=thread&paged=false (Zugriff: 4.6.2008)

[JSR311-Expert-Group 2007d] JSR311-EXPERT-GROUP: *Representation<T> and Entity<T>*. April 2007 – URL https://jsr311.dev.java.net/servlets/BrowseList?listName=dev&from=723786&to=723786&count=68&by=thread&paged=false (Zugriff: 4.6.2008)

[JSR311-Expert-Group 2008a] JSR311-EXPERT-GROUP: *Client API*. März 2008 – URL https://jsr311.dev.java.net/servlets/BrowseList?list=dev&by=thread&from=1063789 (Zugriff: 22.4.2008)

[JSR311-Expert-Group 2008b] JSR311-EXPERT-GROUP: *Limit extensions preprocessing*. Juli 2008 – URL https://jsr311.dev.java.net/servlets/BrowseList?listName=dev&from=1196535&to=1196535&count=99&by=thread&paged=false (Zugriff: 9.7.2008)

[JSR311-Expert-Group 2008c] JSR311-EXPERT-GROUP: *@Path limited=false templates: (?!/).+(?<!/)*. Juli 2008 – URL https://jsr311.dev.java.net/servlets/BrowseList?listName=users&from=1193001&to=1193001&count=99&by=thread&paged=false (Zugriff: 3.7.2008)

[JSR311-Expert-Group 2008d] JSR311-EXPERT-GROUP: *@PathParam(...) PathSegments*. Mai / Juni 2008 – URL https://jsr311.dev.java.net/servlets/BrowseList?listName=users&from=1146110&to=1146110&by=thread&paged=false (Zugriff: 3.6.2008)

[JSR311-Expert-Group 2008e] JSR311-EXPERT-GROUP: *Problem with StreamedOutput model*. März 2008 – URL https://jsr311.dev.java.net/servlets/BrowseList?listName=dev&by=thread&from=1071672&to=1071672&first=1&count=14 (Zugriff: 4.6.2008)

[JSR311-Expert-Group 2008f] JSR311-EXPERT-GROUP: *Response isn't adequate*. Februar / März 2008 – URL https://jsr311.dev.java.net/servlets/BrowseList?listName=dev&from=1053563&to=1053563&count=75&by=thread&paged=false (Zugriff: 4.6.2008)

[Kahlbrandt 2001] KAHLBRANDT, Bernd: *Software-Engineering mit der Unified Modeling Language*. 2. Auflage. Springer, Berlin, März 2001. – ISBN 978-3540416005

[Khare 2003] KHARE, Rohit: *Extending the REpresentational State Transfer (REST) Architectural Style for Decentralized Systems*, University of California, Irvine, Dissertation, 2003 – URL http://www.ics.uci.edu/~rohit/Khare-Thesis-FINAL.pdf (Zugriff: 14.6.2007). – Kurzversion: [Khare und Taylor 2004] (andere Kapitelnummerierung)

[Khare und Taylor 2004] KHARE, Rohit ; TAYLOR, Richard N.: *Extending the REpresentational State Transfer (REST) Architectural Style for Decentralized Systems*. Mai 2004 – URL http://www.ics.uci.edu/˜rohit/ARRESTED-ICSE.pdf (ausführlich unter [Khare 2003], Zugriff: 14.6.2007)

[Klimek 2005] KLIMEK, Helge S.: *Eine Web Service Schnittstelle für ein Web Service Entwickler-Portal*, Technische Universität Hamburg-Hamburg, Bachelorarbeit, Oktober 2005 – URL http://www.sts.tu-harburg.de/pw-and-m-theses/2005/klim05.pdf (Zugriff: 24.5.2007)

[Koops 2008a] KOOPS, Stephan: *Mailing-List dev@jsr311.dev.java.net – Breadcrumbs*. Februar 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=927 (Zugriff: 29.04.2008)

[Koops 2008b] KOOPS, Stephan: *Mailing-List dev@jsr311.dev.java.net – @*Param and List<?>*. März 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=1120 (Zugriff: 29.04.2008)

[Koops 2008c] KOOPS, Stephan: *Mailing-List dev@jsr311.dev.java.net – Problem with StreamedOutput model*. März 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=1078 (Zugriff: 29.04.2008)

[Koops 2008d] KOOPS, Stephan: *Mailing-List dev@jsr311.dev.java.net – @Target of FIELD for @*Param annotations*. März 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=1000 (Zugriff: 29.04.2008)

[Koops 2008e] KOOPS, Stephan: *Mailing-List dev@jsr311.dev.java.net – @Target of FIELD for @*Param annotations*. März 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=1016 (Zugriff: 29.04.2008)

[Koops 2008f] KOOPS, Stephan: *Mailing-List users@jsr311.dev.java.net – more comments to JAX-RS*. März 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=users&msgNo=176 (Zugriff: 22.04.2008)

[Koops 2008g] KOOPS, Stephan: *Mailing-List users@jsr311.dev.java.net – questions to JSR311*. Februar 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=users&msgNo=69 (Zugriff: 22.04.2008)

[Koops 2008h] KOOPS, Stephan: *Mailing-List users@jsr311.dev.java.net – Some proposals / questions for JSR 311*. Januar 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=users&msgNo=39 (Zugriff: 22.04.2008)

[Koops 2008i] KOOPS, Stephan: *Mailing-List users@jsr311.dev.java.net – status 406 if return object not serializable for POST, PUT, DELETE?* Juni 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=users&msgNo=402 (Zugriff: 18.06.2008)

[Koops 2008j] KOOPS, Stephan: *Mailing-List users@jsr311.dev.java.net – UriInfo.getAncestorResource*()*. März 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=users&msgNo=146 (Zugriff: 22.04.2008)

[Lacey 2007] LACEY, Pete: *New REST JSR*. Februar 2007 – URL http://wanderingbarque.com/nonintersecting/2007/02/14/new-rest-jsr (Zugriff: 30.10.2007)

[Law 2007] LAW, Andrew: *RESTful Web Services: JSR-311 (TS-6411) at JavaOne 2007*. Februar 2007 – URL http://the-music-of-time.blogspot.com/2007/05/restful-web-services-jsr-311-ts-6411-at.html (Zugriff: 25.10.2007)

[Linke 2005] LINKE, Mario: Beispiel für eine XML-RPC-Schnittstelle. In: *JavaSPEKTRUM* (2005) – URL http://www.sigs.de/publications/js/2005/04/linke_JS_04_05.pdf (Zugriff: 27.11.2007)

[Loughran 2007] LOUGHRAN, Steve: *javax.ws.rest*. Februar 2007 – URL http://www.1060.org/blogxter/entry?publicid=8C08746C8C0462CC6FB4E4D69098F1AE (Zugriff: 26.10.2007)

[Louvel 2007a] LOUVEL, Jérôme: *Mailing-List dev@jsr311.dev.java.net – Title change to JAX-RS*. April 2007 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=133 (Zugriff: 29.04.2008)

[Louvel 2007b] LOUVEL, Jérôme: *Mailing-List dev@jsr311.dev.java.net – Title change to JAX-RS*. April 2007 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=151 (Zugriff: 29.04.2008)

[Louvel 2007c] LOUVEL, Jérôme: *New JSR to define a high-level REST API for Java*. Februar 2007 – URL http://blog.noelios.com/2007/02/14/new-jsr-to-define-a-high-level-rest-api-for-java/ (Zugriff: 29.10.2007)

[Louvel 2007d] LOUVEL, Jérôme: *Restlet API and JSR-311 API*. April 2007 – URL http://blog.noelios.com/2007/04/25/restlet-api-and-jsr-311-api (Zugriff: 30.10.2007)

[Louvel 2007e] LOUVEL, Jérôme: *Restlet Issue 282: Support misc HTTP headers*. 2007 – URL http://restlet.tigris.org/issues/show_bug.cgi?id=282 (Zugriff: 28.5.2008)

[Louvel 2008a] LOUVEL, Jérôme: *Mailing-List dev@jsr311.dev.java.net – Client API*. März 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=974 (Zugriff: 29.04.2008)

[Louvel 2008b] LOUVEL, Jérôme: *Re: Arguments for own JAX-RS implementation*. Juni 2008. – Email vom 11.6.2008

[Louvel 2008c] LOUVEL, Jérôme: *Re: JAX-RS comments*. März 2008. – Email vom 28.3.2008

[Louvel 2008d] LOUVEL, Jérôme: *Restlet-Diskussions-Mailing-List – Adding Cookies to a Client Request*. Februar 2008 – URL http://restlet.tigris.org/servlets/ReadMsg?list=discuss&msgNo=4070 (Zugriff: 04.04.2008)

[Louvel 2008e] LOUVEL, Jérôme: *Restlet Issue 505: Refactor authentication and authorization*. 2008 – URL http://restlet.tigris.org/issues/show_bug.cgi?id=505 (Zugriff: 29.5.2008)

[McDonough 2007a] MCDONOUGH, Ryan: *Mailing-List dev@jsr311.dev.java.net – JAX-RS Client API*. Oktober 2007 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=693 (Zugriff: 29.04.2008)

[McDonough 2007b] MCDONOUGH, Ryan: *Mailing-List dev@jsr311.dev.java.net – JAX-RS Client API*. Oktober 2007 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=706 (Zugriff: 29.04.2008)

[McDonough 2007c] MCDONOUGH, Ryan J.: *Some thoughts on a JAX-RS client API*. Oktober 2007 – URL http://www.damnhandy.com/2007/10/11/the-potential-of-a-jax-rs-client-api (Zugriff: 17.06.2008)

[Meunier 1995] MEUNIER, Regine: *The Pipes and Filters Architecture*. Kap. 22. In: *Pattern Languages of Program Design*, Addison-Wesley, 1995

[Microsoft 2007] MICROSOFT: *Microsoft NTLM*. Juli 2007 – URL http://msdn2.microsoft.com/en-us/library/Aa378749.aspx (Zugriff: 21.8.2007)

[Mueller 2007a] MUELLER, Patrick: *Mailing-List dev@jsr311.dev.java.net – JAX-RS Client API*. Oktober 2007 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=694 (Zugriff: 29.04.2008)

[Mueller 2007b] MUELLER, Patrick: *Mailing-List dev@jsr311.dev.java.net – JAX-RS Client API*. Oktober 2007 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=705 (Zugriff: 29.04.2008)

[Mueller 2007c] MUELLER, Patrick: *on JSR 311*. August 2007 – URL http://pmuellr.blogspot.com/2007/08/on-jsr-311.html (Zugriff: 29.03.2008)

[Noelios 2007a] NOELIOS TECHNOLOGIES: *Restlet – Introduction*. September 2007 – URL http://www.restlet.org/about/introduction (Zugriff: 20.9.2007)

[Noelios 2007b] NOELIOS TECHNOLOGIES: *Restlet – Lightweight REST framework for Java*. September 2007 – URL http://www.restlet.org/ (Zugriff: 20.9.2007)

[Noelios 2008a] NOELIOS TECHNOLOGIES: *Restlet – Roadmap*. Juli 2008 – URL http://www.restlet.org/about/roadmap (Zugriff: 4.7.2008)

[Noelios 2008b] NOELIOS TECHNOLOGIES: *Restlet 1.1 – First steps*. Mai 2008 – URL http://www.restlet.org/documentation/1.1/firstSteps (Zugriff: 9.7.2008)

[Noelios 2008c] NOELIOS TECHNOLOGIES: *Restlet 1.1 – Tutorial*. April 2008 – URL http://www.restlet.org/documentation/1.1/tutorial (Zugriff: 15.4.2008). – Herzlichen Dank für die freundliche Genehmigung, Grafiken der Seite nutzen zu dürfen

[O'Reilly 2003] O'REILLY, Tim: *REST vs. SOAP at Amazon*. April 2003 – URL http://www.oreillynet.com/pub/wlg/3005 (Zugriff: 5.9.2007)

[Peltz 2003] PELTZ, Chris: *Applying Design Issues and Patterns in Web Services*. Januar 2003 – URL http://www.devx.com/enterprise/Article/10397/0/page/3 (Zugriff: 7.12.2007)

[PhpRestSql 2005] : *PHP REST SQL – Demo/tutorial*. November 2005 – URL http://phprestsql.sourceforge.net/tutorial.html (Zugriff: 18.9.2007)

[Pilgrim 2003] PILGRIM, Mark: *Atom Authentication*. Dezember 2003 – URL http://www.xml.com/pub/a/2003/12/17/dive.html (Zugriff: 21.8.2007)

[Prasanna 2007] PRASANNA, Dhanji R.: *Mailing-List dev@jsr311.dev.java.net – Welcome to JSR 311*. April 2007 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=dev&msgNo=99 (Zugriff: 29.04.2008)

[Prescod 2002a] PRESCOD, Paul: *Common REST Mistakes*. September 2002 – URL http://www.prescod.net/rest/mistakes/ (Zugriff: 29.8.2007)

[Prescod 2002b] PRESCOD, Paul: *The Emperor's New Tags - The SOAP/REST-Controversy*. Oktober 2002 – URL http://www.prescod.net/rest/soap_rest_short.ppt (Zugriff: 27.8.2007)

[Prescod 2002c] PRESCOD, Paul: *Roots of the REST/SOAP Debate*. 2002 – URL http://www.prescod.net/rest/rest_vs_soap_overview/ (Zugriff: 27.8.2007)

[Prescod 2002d] PRESCOD, Paul: *Second Generation Web Services*. Februar 2002 – URL http://webservices.xml.com/pub/a/ws/2002/02/06/rest.html (Zugriff: 26.7.2007)

[RFC-1738 1994] BERNERS-LEE, Tim ; MASINTER, Larry ; MCCAHILL, Mark: *Request for Comments 1738: Uniform Resource Locators*. Juni 1994 – URL http://www.ietf.org/rfc/rfc1738.txt (plain-text, Zugriff: 20.11.2007) http://tools.ietf.org/html/rfc1738 (HTML, Zugriff: 20.11.2007)

[RFC-1808 1995] FIELDING, Roy: *Request for Comments 1808: Relative Uniform Resource Locators*. Juni 1995 – URL http://www.ietf.org/rfc/rfc1808.txt (plain-text, Zugriff: 20.11.2007) http://tools.ietf.org/html/rfc1808 (HTML, Zugriff: 20.11.2007)

[RFC-1945 1996] BERNERS-LEE, Tim ; FIELDING, Roy ; FRYSTYK, Henrik: *Request for Comments 1945: Hypertext Transfer Protocol – HTTP/1.0*. Mai 1996 – URL http://www.ietf.org/rfc/rfc1945.txt (plain-text, Zugriff: 12.6.2007) http://tools.ietf.org/html/rfc1945 (HTML, Zugriff: 12.6.2007)

[RFC-2068 1997] FIELDING, Roy ; GETTYS, J. ; MOGUL, J. ; FRYSTYK, H. ; BERNERS-LEE, Tim: *Request for Comments 2068: Hypertext Transfer Protocol – HTTP/1.1*.

Januar 1997 – URL http://www.ietf.org/rfc/rfc2068.txt (plain-text, Zugriff: 20.8.2007) http://tools.ietf.org/html/rfc2068 (HTML, Zugriff: 20.8.2007)

[RFC-2396 1998] BERNERS-LEE, Tim ; FIELDING, Roy ; MASINTER, Larry: *Request for Comments 2396: Uniform Resource Identifier: Generic Syntax*. August 1998 – URL http://www.ietf.org/rfc/rfc2396.txt (plain-text, Zugriff: 25.5.2007) http://tools.ietf.org/html/rfc2396 (HTML, Zugriff: 25.5.2007)

[RFC-2616 1999] BERNERS-LEE, Tim ; FIELDING, Roy ; FRYSTYK, Henrik ; GETTYS, Jim ; MOGUL, Jeffrey C. ; LASINTER, Larry ; LEACH, Paul: *Request for Comments 2616: Hypertext Transfer Protocol – HTTP/1.1*. Juni 1999 – URL http://www.ietf.org/rfc/rfc2616.txt (plain-text, Zugriff: 12.6.2007) http://tools.ietf.org/html/rfc2616 (HTML, Zugriff: 12.6.2007)

[RFC-2617 1999] FRANKS, J. ; HALLAM-BAKER, P. ; HOSTETLER, J. ; LAWRENCE, S. ; LEACH, Paul ; LUOTONEN, A. ; STEWART, L.: *Request for Comments 2617: HTTP Authentication: Basic and Digest Access Authentication*. Juni 1999 – URL http://www.ietf.org/rfc/rfc2617.txt (plain-text, Zugriff: 9.6.2007) http://tools.ietf.org/html/rfc2617 (HTML, Zugriff: 9.6.2007)

[RFC-2965 2000] KRISTOL, D. ; MONTULLI, L.: *HTTP State Management Mechanism*. Oktober 2000 – URL http://www.ietf.org/rfc/rfc2965.txt (plain-text, Zugriff: 8.10.2007) http://tools.ietf.org/html/rfc2965 (HTML, Zugriff: 8.10.2007)

[RFC-3986 2005] BERNERS-LEE, Tim ; FIELDING, Roy ; MASINTER, Larry: *Request for Comments 3986: Uniform Resource Identifier: Generic Syntax*. Januar 2005 – URL http://www.ietf.org/rfc/rfc3986.txt (plain-text, Zugriff: 25.5.2007) http://tools.ietf.org/html/rfc3986 (HTML, Zugriff: 25.5.2007)

[RFC-4918 2007] DUSSEAULT, Lisa: *HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV)*. Juni 2007 – URL http://www.ietf.org/rfc/rfc4918.txt (plain-text, Zugriff: 15.1.2008) http://tools.ietf.org/html/rfc4918 (HTML, Zugriff: 15.1.2008)

[RFC-5023 2007] GREGORIO, Joe ; HORA, Bill de: *Request for Comments 5023: The Atom Publishing Protocol*. Oktober 2007 – URL http://www.ietf.org/rfc/rfc5023.txt (plain-text, Zugriff: 12.10.2007) http://tools.ietf.org/html/rfc5023 (HTML, Zugriff: 12.10.2007)

[Richardson und Ruby 2007a] RICHARDSON, Leonard ; RUBY, Sam: *RESTful Web Services*. 1. Auflage. O'Reilly Media, Mai 2007. – deutsche Übersetzung: [Richardson und Ruby 2007b]. – ISBN 978-0596529260

[Richardson und Ruby 2007b] RICHARDSON, Leonard ; RUBY, Sam: *Web Services mit REST*. 1. Auflage. O'Reilly Media, Oktober 2007. – deutsche Übersetzung von [Richardson und Ruby 2007a]. – ISBN 978-3897217270

[Sandoz 2007a] SANDOZ, Paul: *JSR 311 - JAX-RS The Java API for RESTful Web Services*. Dezember 2007 – URL Folien: http://www.javapolis.com/

confluence/download/attachments/32918/C_13_04_05.pdf (PDF, Zugriff: 16.6.2008) Vortrag: http://www.parleys.com/display/PARLEYS/JSR+311+-+JAX-RS+The+Java+API+for+RESTful+Web+Services (HTML, Zugriff: 12.6.2008). – Vortrag auf der Konferenz JavaPolis '07 in Antwerpen, gehalten am 13.12.2007

[Sandoz 2007b] SANDOZ, Paul: *JSR-311 and the republicans*. Februar 2007 – URL http://blogs.sun.com/sandoz/entry/jsr_311_and_the_republicans (Zugriff: 30.10.2007)

[Sandoz 2008a] SANDOZ, Paul: *Jersey Client API*. Februar 2008 – URL http://blogs.sun.com/sandoz/entry/jersey_client_api (Zugriff: 23.6.2008)

[Sandoz 2008b] SANDOZ, Paul: *Mailing-List users@jsr311.dev.java.net – questions to JSR311*. Februar 2008 – URL https://jsr311.dev.java.net/servlets/ReadMsg?list=users&msgNo=73 (Zugriff: 22.04.2008)

[Semergence 2007] : *Java Might Get a Formal API for REST. Even Needed?* Februar 2007 – URL http://www.semergence.com/2007/02/15/java-might-get-a-formal-api-for-rest-even-needed (Zugriff: 30.10.2007)

[SOAP 1.2 2007] GUDGIN, Martin ; HADLEY, Marc ; MENDELSOHN, Noah ; MOREAU, Jean-Jacques ; NIELSEN, Henrik F. ; KARMARKAR, Anish ; LAFON, Yves: *SOAP Version 1.2 Part 2: Adjuncts*. April 2007 – URL http://www.w3.org/TR/soap12-part2 (Zugriff: 7.12.2007)

[Tanenbaum und van Steen 2002] TANENBAUM, Andrew S. ; STEEN, Maarten van: *Distributed Systems - Principles and Paradigms*. Prentice Hall, 2002. – ISBN 0-13-088893-1

[Thelin 2003] THELIN, Jørgen: *A Comparison of Service-oriented, Resource-oriented, and Object-oriented Architecture Styles*. 2003 – URL http://www.thearchitect.co.uk/presentations/arch-styles/3-arch-styles.pdf (Präsentation, Zugriff: 12.6.2007)

[Tilkov 2007a] TILKOV, Stefan: *Interview: Jérôme Louvel about Restlet*. April 2007 – URL http://www.infoq.com/articles/restlet-louvel-interview (Zugriff: 5.11.2007)

[Tilkov 2007b] TILKOV, Stefan: *New JSR Proposed: Java API for RESTful Web Services*. Februar 2007 – URL http://www.infoq.com/news/2007/02/jsr-311-java-rest-api (Zugriff: 25.10.2007)

[Tilkov 2007c] TILKOV, Stefan: *REST - das bessere Web-Services-Modell*. April 2007 – URL http://www.innoq.com/blog/st/presentations/2007/REST_-_das_bessere_Web-Service-Modell__Stefan_Tilkov.pdf (Zugriff: 7.12.2007)

[Tilkov 2008a] TILKOV, Stefan: *Addressing Doubts about REST*. März 2008 – URL http://www.infoq.com/articles/tilkov-rest-doubts (Zugriff: 24.4.2008)

[Tilkov 2008b] TILKOV, Stefan: *Eine kurze Einführung in REST*. 2008 – URL http://www.sigs.de/publications/js/2008/03/tilkov_xml_JS_03_08.pdf (Zugriff: 29.5.2008)

[Tilkov und Ghadir 2006] TILKOV, Stefan ; GHADIR, Phillip: REST – die Architektur des Webs. In: *JavaSPEKTRUM* (2006) – URL http://www.sigs.de/publications/os/2006/05/tilkov_ghadir_OS_05_06.pdf (Zugriff: 27.11.2007)

[Tschalär 2003] TSCHALÄR, Ronald: *NTLM Authentication Scheme for HTTP*. June 2003 – URL http://www.innovation.ch/personal/ronald/ntlm.html (Zugriff: 21.8.2007)

[UDDI 2008] : *OASIS UDDI Specifications TC - Committee Specifications*. 2008 – URL http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm (Zugriff: 14.1.2008)

[Vogel 2003] VOGEL, Werner: *Web Services are not Distributed Objects*. November 2003 – URL http://www.allthingsdistributed.com/historical/archives/000343.html (Zugriff: 12.6.2007). – Werner Vogel ist CTO von amazon.com

[W3C 2008] W3C: *Web Service Activity*. April 2008 – URL http://www.w3.org/2002/ws (Zugriff: 8.5.2008)

[Wikipedia 2007] WIKIPEDIA: *Wikipedia — Wikipedia, Die freie Enzyklopädie*. 2007 – URL http://de.wikipedia.org/w/index.php?title=Wikipedia&oldid=33316850 (Zugriff: 19.6.2007)

[WSDL-2.0-Adjuncts 2007] CHINNICI, Roberto ; HAAS, Hugo ; LEWIS, Amelia A. ; MOREAU, Jean-Jacques ; ORCHARD, David ; WEERAWARANA, Sanjiva: *Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts*. Juni 2007 – URL http://www.w3.org/TR/wsdl20-adjuncts/ (Zugriff: 7.12.2007)

[XML-PRC 2003] : *XML-RPC.com*. 2003 – URL http://www.xmlrpc.com/ (Zugriff: 27.11.2007)

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 10. Juli 2008	
Ort, Datum	Unterschrift