Inhalt

1. Einleitung

Der anhaltende Trend zu Globalisierung und Expansion führt zu ständig wachsenden heterogenen IT-Landschaften der Unternehmen. Anwendungen, die alle möglichen Formen von Schnittstellen aufweisen, werden auf unterschiedlichsten Systemen verteilt betrieben. Auch der Einsatz mobiler Plattformen im Business Umfeld nahm in den letzten drei Jahren spürbar zu.

Komplexe Geschäftsprozesse laufen zunehmend auf unterschiedlichen Systemen verteilt ab. Um dieser Entwicklung Rechnung zu tragen, wurde deshalb nach einem Weg gesucht, heterogenen Systemen einen sicheren und einfachen Kommunikationsaustausch zu ermöglichen.
Den heutigen Quasi-Standard für den Informationsaustausch verteilter Systeme bilden Web Services. Diese gewährleisten den von SOA geprägten Begriff der Interoperabilität wie keine andere Technologie. Viele der bis dato realisierten Web Services verwenden **E**xtensible **M**arkup **L**anguage **R**emote **P**rocedure **C**alls (XML-RPC) oder das **S**imple **O**bject **A**ccess **P**rotocol (SOAP), das den inoffiziellen Standard in der SOA-Welt bildet. Doch in den letzten Jahren gewann ein völlig anderer Web Service Ansatz zunehmend an Bedeutung.

Roy Fielding untersuchte in seiner Doktorarbeit aus dem Jahr 2000 den Erfolg der größten verteilten Anwendung, dem World Wide Web. In dieser Dissertation identifizierte er verschiedene Architekturprinzipien, die **RE**presentational **S**tate **T**ransfer oder kurz REST genannt werden. Dieser Architekturstil stellt neben SOAP und XML-RPC eine weitere, relativ neue Alternative zur Realisierung von Web Services dar, die sich immer größerer Beliebtheit erfreut. Auch Google hat diesen Trend erkannt und verwendet bei den neueren APIs REST.

Diese Arbeit befasst sich mit RESTful Web Services. Es werden die theoretischen Hintergründe in Bezug auf Architektur und dem Zusammenspiel von Server und Client erläutert. Es wird gezeigt, wie man einen Web Service mit der Java API for RESTful Web Services (JAX-RS 1.0) implementiert. Zudem wird auf die mit Java EE 7 ausgelieferte JAX-RS 2.0 Spezifikation vorgegriffen und gezeigt, wie man mithilfe der Client API einen einfachen Client implementiert.

In Kapitel 2 wird zunächst erklärt, was Web Services sind und wozu sie dienen. Der Leser erhält einen kurzen Überblick über die Funktionsweise und das Zusammenspiel zwischen Server und Client.

Kapitel 3 erläutert den Begriff REST und stellt dessen verschiedenen Grundprinzipien vor. Es werden die Architekturprinzipien aufgezeigt, die die Grundpfeiler einer REST-Applikation bilden.

In Kapitel 4 werden zuerst einige Grundlagen vorgestellt, die für die Verwendung der Java API for RESTful Web Services benötigt werden. Im Anschluss daran wird anhand von Code-Beispielen gezeigt, wie man einen Web Service in der Programmiersprache Java erstellt. Um diesen Web Service nutzen zu können, muss dieser deployed werden. Auch dieses Thema wird Teil von Kapitel 4 sein. Abgerundet wird Kapitel 4 mit einer kurzen Einführung in die Client API von JAX-RS 2.0, die Bestandteil von Java EE 7 sein wird.

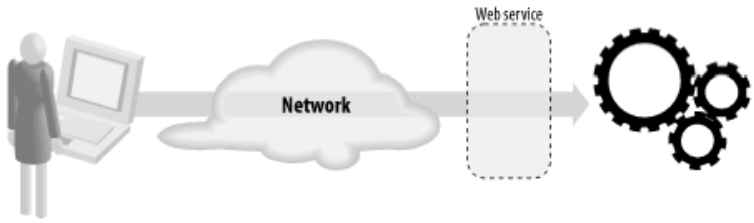
In Kapitel 5 wird der Autor ein persönliches Fazit abgeben. Es werden die wichtigsten Eigenschaften und Features von JAX-RS nochmals hervorgehoben und bewertet.

2. Was ist ein Web Service

Mit der wachsenden Bedeutung verteilter Systeme und komplexer Geschäftsprozesse, die sich zeitgleich mehrerer Systeme bedienen, gewinnt die Kommunikation im Internet immer mehr an Bedeutung. Web Services sind Client-Server Applikationen, die es Systemen ermöglichen – unabhängig von Plattform, Ort oder Programmiersprache – über das Internet oder lokal zu kommunizieren. Dieses Merkmal wird auch Interoperabilität genannt und ist das erklärte Ziel einer jeden SOA-Landschaft.

Das Prinzip eines Web Services ist simpel. Der Anwender greift über die Web Service Schnittstelle lokal oder über das Internet auf die vom Web Service angebotenen Ressourcen zu oder stößt über den Web Service einen Prozess an.
Kann also auf eine Applikation über ein Netzwerk zugegriffen werden und werden dabei Protokolle genutzt wie HTTP, XML oder SMTP, dann ist es ein Web Service.
Abbildung 1 illustriert das zugrunde liegende Konzept aller Web Services[1].

Abbildung 1 - Ein typischer Web Service



Ein Web Service besitzt eine eindeutig identifizierbare Adresse, auch Uniform Resource Identifier(URI) genannt. Der URI ermöglicht eine klare Zuordnung und weltweite Verfügbarkeit des Web Services.

Web Services sind in der Lage, diverse Datenformate zu empfangen und in ein für den Client lesbares Format zu transformieren. Dieser Vorgang wird Datentransformation oder auch Daten Mapping genannt. Bewährte Datenformate sind XML oder auch JSON, kurz für **J**ava**S**cript **O**bject **N**otation[2]. JSON ist ein leichtgewichtiges, einfach zu lesendes und schreibendes Datenformat, das sich immer größerer Beliebtheit erfreut. Als Übertragungsstandard hat sich das Transportprotokoll **H**yper**t**ext **T**ransfer **P**rotocol (HTTP) durchgesetzt.

Web Services können über verschiedene Technologien realisiert werden. Neben SOAP und REST ist noch **E**xtensible **M**arkup **L**anguage **R**emote **P**rocedure Call (XML-RPC) als weitere Implementierungsvariante zu nennen.
REST bildet den Schwerpunkt dieser Arbeit.

[1] Vgl. Snell, James.: Programming Web Services with SOAP, S.10
[2] JSON ist valider JavaScript Code

3. RESTful Web Services

RESTful Web Services liegen im Trend der Zeit. Viele Entwickler und Organisationen bewerben ihre Technologien und Services als RESTful Web Services. Dies entspricht jedoch oftmals nicht der Wahrheit. Damit ein Web Service auch tatsächlich RESTful ist, muss er einige Grundprinzipien erfüllen. Im Folgenden werden die Eckpfeiler eines RESTful Web Service näher erläutert. Im Anschluss werden kurz einige bekannte Vertreter von RESTful Web Services vorgestellt.

3.1 Einführung in REST

Im Jahre 2000 untersuchte Roy Fielding in seiner Dissertation den Erfolg des World Wide Web und identifizierte dabei spezifische Architekturprinzipien, die er **RE**presentational **S**tate **T**ransfer (REST) nannte. Die Anwendung dieser Prinzipien führt bei einem Web Service zu einigen wünschenswerten Eigenschaften wie loser Kopplung oder hoher Skalierbarkeit.
Obwohl REST an kein spezifisches Protokoll gebunden ist, hat sich als Übertragungsstandard das **H**yper**t**ext **T**ransport **P**rotocol (HTTP) durchgesetzt.
Zu beachten ist, dass REST kein Protokoll, sondern ein Architekturstil ist. Der Aufbau eines REST Services ist simpel, aber dennoch wohl definiert. Web Services, die alle Architekturprinzipien von REST erfüllen, werden auch RESTful Web Services genannt.[3]

3.2 RESTful Architekturprinzipien

Ein RESTful Web Service entsteht nicht durch das Nutzen eines Frameworks oder das Einbinden von Bibliotheken. RESTful Web Services entstehen durch das Einhalten strikter Architekturprinzipien. Im Folgenden werden die Architekturprinzipien von REST beschrieben. Ein Web Service ist nur dann ein RESTful Web Service, wenn er all diese Prinzipien befolgt.

3.2.1 Adressierbarkeit

Ressourcen sind das zentrale Konzept in REST. Gelegentlich wird eine REST-Architektur auch als **R**esource-**o**riented **A**rchitecture (ROA) bezeichnet.
Jede Ressource muss durch einen eindeutigen Uniform Resource Identifier (URI) identifizierbar und erreichbar sein. Eine Ressource wird in Java als Objekt repräsentiert, im Zusammenhang mit REST allerdings als Ressource bezeichnet. So wird zum Beispiel ein Arbeitnehmer mit einer bestimmten Personalnummer (hier 12345) durch folgende URI adressiert: *http://www.mycompany.com/employee/12345*.[4]

3.2.2 Einheitliche Schnittstellen

Zur Manipulation von Ressourcen werden die Methoden aus dem Protokoll der Anwendungsschicht, auf dem die Web Services aufsetzen, verwendet. Findet also HTTP Verwendung, so können HTTP-Methoden wie GET, PUT, DELETE, POST, HEAD und OPTIONS auf alle Ressourcen angewendet werden. Ein GET-Aufruf liefert den aktuellen Zustand einer Ressource. Bei einem Arbeitnehmer kann dieser zum Beispiel das aktuelles Gehalt, die Anschrift oder andere personelle Informationen beinhalten. Mit POST oder PUT kann eine Ressource angelegt und verändert, mit DELETE gelöscht werden.[5]

3.2.3 Repräsentationsorientierung

Durch dieses Architekturprinzip wird eine Ressource von seiner Repräsentation entkoppelt. Die unter einem URI adressierbaren Ressourcen können unterschiedliche

[3] Vgl. Burke, B.: RESTful Java with JAX-RS, S. 3
[4] Vgl. Burke, B.: RESTful Java with JAX-RS, S.6f.
[5] Vgl. Burke, B.: RESTful Java with JAX-RS, S. 7 ff.

Repräsentationsformate haben. In welchem Datenformat die entsprechende Ressource repräsentiert wird, wird von Client und Server ausgehandelt und Content Negotiation genannt. Mögliche Repräsentationsformate können zum Beispiel E**x**tensible **M**arkup **L**anguage (XML), **J**ava**S**cript **O**bject **N**otation (JSON) oder das **H**yper**t**ext **M**arkup **L**anguage Format (HTML) sein.[6]

3.2.4 Zustandslose Kommunikation

Zustandslose Kommunikation in RESTful Web Services bedeutet, dass keine Client Session auf dem Server angelegt wird. Der Server verwaltet nur die Ressourcen, die er erhält. Sollten auf einer Client Session basierende Informationen benötigt werden, so werden diese vom Client verwaltet und dem Server bei Bedarf mit dem Request mitgeteilt. Eine Serviceschicht, die keine Client Informationen zu verwalten hat, ist um einiges einfacher skalierbar.[7]

3.2.5 Hypermedia As The Engine Of Application State

Hypermedia **A**s **T**he **E**ngine **O**f **A**pplication **S**tate, kurz HATEOAS, ist ein dokumentenzentrierter Ansatz, der das Einbetten von Hyperlinks zu anderen Services und Informationen ermöglicht. Die Idee dahinter ist die, dass ein Web Service ähnlich wie eine Website funktionieren soll. Der Client kennt nur den Einstiegspunkt, alle folgenden Web Services werden in dem vom Server zurückgelieferten Dokument verlinkt. Der Zustand der repräsentierten Ressource kann also durch das Aufrufen der verlinkten Web Services vollständig abgerufen werden. Will man beispielsweise alle Arbeitnehmer eines Unternehmens aufrufen, so kann man einen GET-Request auf den URI *http://www.mycompany.com/employee* absenden. Listing 1 zeigt die Antwort des Servers (Response) im XML Format, der auf weitere Web Services verweist. Da der Response sehr groß werden und clientseitig zu sehr langen Wartezeiten führen kann, werden hier nur die ersten fünf Arbeitnehmer angezeigt. Zusätzlich wird ein Hyperlink eingebettet, der auf die nächste Menge an Arbeitnehmern verweist.

Listing 1: Response im XML-Format mit einer Verlinkung zu weiteren Web Services

```
<employees>
<link rel="next" href="http://www.mycompany.com/employee?startIndex=6"/>
<employee id="1">
      <name>Max Mustermann</name>
      <salary>5500.0</salary>
      ...
   </employee>
   ...
<employee id="5">
      <name>Heinrich Maier</name>
      <salary>8500.0</salary>
      ...
   </employee>
</employees>
```

Durch Hypermedia und Hyperlinks ist somit das Zusammensetzen von komplexen Mengen an Informationen aus verschiedenen Quellen möglich. Hyperlinks erlauben es, zusätzliche Daten zu referenzieren und zu aggregieren, ohne die Antwort vom Server unnötig aufzublähen. Des Weiteren kann der Server seine URIs ändern, ohne dass dies Auswirkungen auf den Client hat.[8]

---

[6] Vgl. Burke, B.: RESTful Java with JAX-RS, S. 10

[7] Vgl. Burke, B.: RESTful Java with JAX-RS, S. 10 f.

[8] Vgl. Burke, B.: RESTful Java with JAX-RS, S. 11 ff.

3.2.6 RESTful Services

REST gewann in den letzten Jahren zunehmend an Popularität. Große, internationale Unternehmen machen sich den Hype um REST zu Nutze und vermarkten ihre Web Services als RESTful. Nachfolgend seien einige bekannte Vertreter kurz vorgestellt.

3.2.6.1 Google – Google Translate API & Youtube

Der Internetkonzern Google bietet neben seinem Suchmaschinendienst noch zahlreiche weitere Dienste an. Einer dieser Dienste ist die Bereitstellung einer Google Translate API. Die API bietet Entwicklern die Möglichkeit, ihre Texte programmatisch in andere Sprachen zu übersetzen.
Ein weiterer Google-Dienst ist das Internet-Videoportal YouTube (www.youtube.com). Auf der Internetpräsenz befinden sich Film-, Musik- und Fernsehausschnitte. Sogenannte „Video-Feeds“ bzw. „Vlogs“ können in Blogs gepostet oder auch einfach auf Webseiten über die YouTube REST API eingebunden werden. Der Entwicklungsstandard des Google Developer Teams ist vorbildlich. Alle in dieser Arbeit behandelten Architekturprinzipien wurden in beiden APIs strikt befolgt.

3.2.6.2 Twitter – The Twitter REST API

Twitter ist ein Mikro-Blogging-Dienst, der die Verbreitung von Kurznachrichten über das Internet ermöglicht. Twitter wird zudem als Kommunikationsplattform und soziales Netzwerk kommuniziert. Der Twitter Dienst kann über dessen eigene REST API angesprochen werden. Rein technisch gesehen ist diese API allerdings nicht RESTful, da sie gegen einige der REST Architekturprinzipien verstößt. So zeigt beispielsweise die URL http://twitter.com/favorites die Favoritenliste des jeweils authentifizierten Users an. Dieses Verhalten verstößt gegen das Prinzip der Addressierbarkeit (engl. Resource Identification), *Kapitel 3.2.5.1*. Jede Ressource muss eindeutig identifizierbar und erreichbar sein.

4. Java API for RESTful Web Services

Die Java API for RESTful Web Services, kurz JAX-RS, ist die Spezifikation eines Frameworks, die im Oktober 2008 fertiggestellt wurde und darauf fokussiert ist, sogenannte Plain Old Java Objects (POJOs) durch Annotationen als Web Service zu deklarieren. JAX-RS setzt HTTP als verwendetes Protokoll der Anwendungsschicht voraus und bietet Annotationen an, mit denen spezielle URI-Muster und HTTP-Methoden an Java-Klassen und Java–Methoden gebunden werden können. Statt Java-Klassen mit Annotationen zu versehen, kann auch das Interface annotiert werden, um den Web Service von der Business Logik der implementierenden Klasse zu trennen. Durch die sogenannte Parameter Injection können Informationen aus dem HTTP-Request als Parameter der jeweiligen Java-Methode übergeben werden. JAX-RS bietet Message Body Reader und Message Body Writer an, mit denen die Java-Objekte in das jeweilige Repräsentationsformat verpackt werden können und umgekehrt. Auch die verfügbaren Exception Mapper sind bei der Entwicklung eines RESTful Web Services hilfreich. Zusätzlich gibt es noch Unterstützung für die Content Negotiation beim Datenaustausch zwischen Client und Server.
Mit der baldigen Veröffentlichung von Java EE 7 wird es erstmals möglich sein, die JAX-RS eigene Client API zu nutzen. Diese Neuerung bringt viele Vorteile mit sich. Mussten bis dato stets eigene, proprietäre Client Schnittstellen geschrieben werden, so kann man dann auf die JAX-RS eigene Referenzimplementierung Jersey zurückgreifen.

Im Folgenden werden zuerst einige Grundlagen für die Verwendung von JAX-RS erklärt. Danach wird beispielhaft gezeigt, wie man mit JAX-RS einen Web Service in Java implementiert und diesen mit der JAX-RS Client API anspricht.[9]

4.1 JAX-RS Grundlagen

Bevor auf die Verwendung der Java API for RESTful Web Services eingegangen wird, werden nachfolgend noch die Grundlagen für deren Nutzung erläutert. Es wird sowohl die Abbildung der HTTP-Methoden auf die JAX-RS Annotationen, als auch die Bedeutung der verschiedenen **M**ultipurpose **I**nternet **M**ail **E**xtension Typen (MIME-Typen) beschrieben. Anschließend wird der Mechanismus Content Negotiation näher erklärt.

4.1.1 HTTP Methoden

JAX-RS bietet fünf Annotationen an, die spezifische HTTP-Operationen binden. Zu beachten ist, dass jeder Java Methode nur eine dieser Annotation zugeordnet werden darf.[10]

- @javax.ws.rs.GET
- @javax.ws.rs.POST
- @javax.ws.rs.PUT
- @javax.ws.rs.DELETE
- @javax.ws.rs.HEAD
- @javax.ws.rs.OPTIONS

Die am häufigsten verwendeten HTTP Methoden sind hierbei GET, POST, PUT und DELETE. Diese seien im folgenden Abschnitt erklärt. [11]

[9] Vgl. Burke, B.: RESTful Java with JAX-RS, S. xiii und S. 27 ff.
[10] Vgl. JavaTM Platform, Enterprise Edition 6 API Specification, Package javax.ws.rs
[11] Vgl. Burke, B.: RESTful Java with JAX-RS, S.20ff

4.1.1.1 GET

GET dient zum Abruf von Zuständen eines Objekts bzw. einer Ressource. So kann der Client beispielsweise alle Mitarbeiter eines Unternehmens anfragen. Die angeforderten Mitarbeiterdaten werden dann vom Server über die HTTP-Response an den Client übermittelt. Um bei einer großen Anzahl an Mitarbeitern eine Überladung des Clients oder Verletzung der Antwortzeit zu vermeiden, ist es dem Client möglich, seine Anfrage mittels Parametern über den URI zu spezifizieren. Wie das funktioniert, wird in Kapitel 4.2.2.3 @QueryParam beschrieben.

4.1.1.2 POST

Mit POST können Ressourcen angelegt werden. Um zum Beispiel einen neuen Mitarbeiter anzulegen, sendet der Client eine Repräsentation des neuen Objekts zum Ziel-URI, ohne dabei eine eindeutige Objekt ID mitzuliefern. Der Server empfängt die POST-Nachricht, erstellt den neuen Mitarbeiter und weist ihm eine von der Datenbank eindeutige ID zu. Damit der Client den Mitarbeiter editieren, bearbeiten oder löschen kann, benötigt er diese ID. Diese teilt ihm der Server über die Response Nachricht mit.

4.1.1.3 PUT

Mit PUT kann eine Ressource sowohl angelegt als auch aktualisiert werden. Um eine neue Ressource anzulegen, sendet der Client die Repräsentation des Objekts an den exakten URI, der die entsprechende Ressource repräsentiert. Existiert die Ressource bereits unter diesem URI, so wird die Ressource lediglich aktualisiert. Wurde die Ressource erfolgreich angelegt, so sendet der Server den Response Code 201 „Created“ an den Client. Ein potentielles Problem bei PUT ist, dass der Client zur Aktualisierung immer die eindeutige ID der Ressource angeben muss. Dies ist nicht immer leicht, immerhin sollte diese ID von der Datenbank vergeben werden.[12]

4.1.1.4 DELETE

DELETE dient dem Löschen von Ressourcen. Der Client ruft hierzu einfach die DELETE Methode mit Angabe des exakten URI auf, der die Ressource repräsentiert, die er löschen möchte. Wurde die Ressource erfolgreich gelöscht, sendet der Server über die Response Nachricht den Code 200 „OK“ mit Message Body oder Code 204 „NO CONTENT“ ohne Message Body an den Client.

4.1.2 Multipurpose Internet Mail Extension Typ

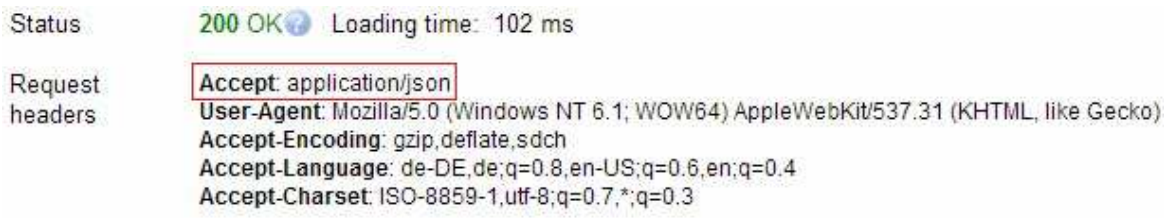
Der Multipurpose Internet Mail Extension Typ, kurz MIME-Type, fand ursprünglich in der Übertragung von Emails Verwendung. Heute definiert er das Format der über den Body einer HTTP-Request oder einer HTTP-Response übermittelten Ressource. Das hierfür verwendete Feld im HTTP-Header wird Content-Type genannt.

Der MIME-Typ setzt sich aus einem Haupt- und einem Untertypen zusammen. Bei JAX-RS wird der MIME-Typ verwendet, um anzugeben, welches Repräsentationsformat eine Java-Methode empfangen, bzw. produzieren kann.[13] Dieser Mechanismus wird auch Content Negotiation genannt. So kann beispielsweise der Client über den Request Header angeben, welches Repräsentationsformat er vom Server im Response akzeptiert.

Abbildung 2 zeigt die typischen Headerinformationen.

[12] Vgl. W3C: Hypertext Transfer Protocol – HTTP/1.1
[13] Vgl. REST Patterns, MIME Type

Abbildung 2 – HTTP Request Header via Advanced Rest Client



Zu den für JAX-RS wichtigsten MIME-Typen gehören unter anderem

- application/xml
- application/json
- application/plain

Die Bedeutung der oben angegebenen MIME-Typen ist hierbei selbsterklärend. Neben diesen gängigen Typen gibt es noch zahlreiche weitere, die im Verlaufe dieser Arbeit nicht mehr erwähnt werden sollen.

4.1.3 Content Negotiation

RESTful Web Services müssen eine Vielzahl von Clients, die auf verschiedenen Plattformen operieren, bedienen. Je nach Plattform benötigt ein Client ein anderes Repräsentationsformat. Zum Beispiel können **A**synchronous **Ja**vaScript and **X**ML-Clients (AJAX-Clients) besser mit dem JSON-Format arbeiten, Ruby Clients bevorzugen **Y**et **A**nother **M**ulticolumn Layout (YAML). HTTP bietet Hilfen an, um diese Integrationsprobleme zu unterstützen. Ein Client kann durch das jeweilige Accept-Feld im Request-Header dem Server mitteilen, welches Format, Encoding und Sprache der Response haben soll. Die Aushandlung dieser Eigenschaften des HTTP-Responses wird Content Negotiation genannt.

Auch JAX-RS bietet Unterstützung für die Content Negotiation. Listing 6 zeigt, wie ein Web Service durch Annotation von @Produces angibt, welche Formate der Web Service für die Antwort erzeugen kann. Als Parameter für die @Produces-Annotation werden die MIME-Types der produzierten Datenformate, wie in Kapitel 4.1.2 Multipurpose Internet Mail Extensions Typen beschrieben, angegeben.
Die Daten werden dann in dem Format zurückgesendet, welches der Server anbietet und der Client im Accept-Feld am höchsten priorisiert hat. Sollte der Client nur Formate akzeptieren, die der Web Service nicht produziert, wird der Response-Code 406 (Not Acceptable) an den Client gesendet. Umgekehrt kann auch der Server durch die @Consumes-Annotation angeben, welche Datenformate er konsumieren kann.[14]

Listing 2: Die Web Service Methode erzeugt je nach Request XML oder JSON

```
@Path("/employee")
public class EmployeeResource {

@GET
@Path("/{id}")
@Produces({"application/xml", "application/json"})
public Employee getEmployee(@PathParam("id") String id) {
...
}
```

[14] Vgl. Burke, B.: RESTful Java with JAX-RS, S.108 ff.

4.2 JAX-RS – Web Service API

In den vorangegangenen Kapiteln wurden bereits die Grundlagen bezüglich der Abbildung der HTTP Methoden auf die JAX-RS Annotationen erläutert. Dieses Kapitel baut auf diesem Wissen auf und zeigt die Implementierung eines RESTful Web Services in Java. Der Leser sollte im Anschluss dazu in der Lage sein, einfache Web Services mit JAX-RS zu implementieren.

4.2.1 Web Service Deklaration

Um eine Java-Klasse als Web Service zu deklarieren, muss diese entsprechend annotiert werden. Dies geschieht durch die Annotation @javax.ws.rs.Path oder – wenn man die Java-Klasse importiert hat – nur durch @Path. Listing 2 zeigt beispielhaft eine Web Service Deklaration. Zu beachten ist, dass der String in der @Path-Annotation einen Teil des URIs zum Web Service darstellt. Ist beispielsweise der absolute Pfad unseres Servers *http://www.mycompany.com* und annotieren wir unsere Mitarbeiterklasse Employee mit @Path, so wird unsere Employee-Ressource unter *http://www.mycompany.com/employee* verfügbar sein. Den Angestellten mit der ID 15 können wir dann durch Aufruf des URIs http://www.mycompany.com/employee/15 abrufen.[15]

4.2.2 Binden von HTTP Methoden

Wurde die EmployeeResource-Klasse als Web Service deklariert, können nun an die einzelnen Java-Methoden die HTTP Methoden gebunden werden. Listing 2 zeigt, wie ein GET-Aufruf auf den URI *http://www.www.mycompany.com/employee/*1 über die @GET-Annotation auf die Methode getEmployee und ein @PUT-Aufruf über denselben URI auf die Methode createEmployee gemappt wird.

Listing 3: Binden von HTTP Methoden

```
package com.mycompany.rest.restservices;

import javax.ws.rs.*;
import javax.ws.rs.core.Response;
import java.net.URI;

@Path("employee")
public class EmployeeResource {

	@GET
	@Path("/{id}")
	@Produces("application/xml")
	public Employee getEmployee(@PathParam("id") String id) {
	//TODO get employee with given id from database
	return employee;
	}

	@PUT
	@Path("/{id}")
	@Consumes("application/xml")
	public Response createEmployee(@PathParam("id") String id, Employee
	employee) {
	// TODO insert into Database
	return Response.created(URI.create("/employee/" +employee.getId())).build();
	}
}
```

[15] Vgl. Burke, B.: RESTful Java with JAX-RS, S.30 und 45f

4.2.3 JAX-RS Parameter Injection

Der Wert der @Path-Annotation ist in der Regel ein einfacher String. Durch eine Modifikation des Strings ist es allerdings möglich, der aufgerufenen Methode diverse Parameter zu übergeben. Dieser Vorgang wird Parameter Injection genannt. Die JAX-Spezifikation stellt folgende Annotationen zur Injection von Parametern bereit:

- @javax.ws.rs.PathParam
- @javax.ws.rs.MatrixParam
- @javax.ws.rs.QueryParam
- @javax.ws.rs.FormParam
- @javax.ws.rs.HeaderParam
- @javax.ws.rs.CookieParam
- @javax.ws.rs.core.Context
- @javax.ws.rs.DefaultValue

Nachfolgend werden diese Annotationen kurz erklärt. Für eine detaillierte Beschreibung der Anwendung dieser Annotationen, sei auf die JAX-RS Spezifikation verwiesen.

4.2.3.1 @PathParam

Mit dieser Annotation kann man, wie in Listing 2 gezeigt, Teile aus dem URI String auslesen und als Parameter in der Resource-Methode verwenden. Zu beachten sei hierbei, dass der Name des Parameters in der @PathParam-Annotation mit dem aus der geschweiften Klammer aus der @Path-Annotation übereinstimmen muss.[16]

4.2.3.2 @Matrix Parameter

Matrix Parameter sind Name/Wert-Paare die im Pfad des URI Strings eingebettet sind. Die Matrix Parameter werden durch Semikolon getrennt und können, wie in folgender URI durch Fettschrift gekennzeichnet, eingebettet werden:

http://www.mycompany.com/employee;firstname=hans;surname=mustermann/20

Listing 3 zeigt eine Abwandlung von Listing 2. Es wird angenommen, dass ein Mitarbeiter eindeutig durch seinen Namen in Kombination mit seinem Alter identifiziert werden kann. Das Alter wird direkt in die Methode injiziert und ist sofort zur weiteren Verwendung verfügbar.[17] Um den Vor- und Nachname des Mitarbeiters zu bekommen, muss zuerst das PathSegment-Objekt im Methodenrumpf ausgelesen werden.

Anstelle des PathSegment-Objekts hätte auch @MatrixParam wie in Listing 4 verwendet werden können. Vorteil wäre, dass der Wert des Parameters bereits in die Methode injiziert wird. Ein Problem würde sich hierbei jedoch ergeben, wenn ein Matrix Parameter mit demselben Namen zweimal in der URI vorkommt.[18]

Listing 3: Zusammenspiel von Matrix Parametern und dem PathSegement-Objekt

```
@GET
@Path("/{name}/{age}")
@Produces("application/xml")
public Employee getEmployee(@PathParam("name")PathSegment name,
@PathParam("age") Integer age) {

String firstName = name.getMatrixParameters().getFirst("firstname");
```

[16] Vgl. Burke B.: RESTful Java with JAX-RS, S.55
[17] Siehe Kapitel 4.3.2.1, @PathParam
[18] Vgl. Burke B.: RESTful Java with JAX-RS, S.57 ff.

```
String surname = name.getMatrixParameters().getFirst("surname");
...
}
```

Listing 4: Verwendung der @MatrixParam-Annotation

```
public Employee getEmployee(@MatrixParam("firstname") String firstname,
@MatrixParam("surname") String surname, @PathParam("age")Integer age) {
…
}
```

4.2.3.3 @QueryParam

Mit dieser Annotation kann man Query Parameter, die auch wieder Name/Wert-Paare sind, aus dem URI extrahieren. Die verschiedenen Paare werden durch ein „&“ voneinander getrennt. Als Beispiel soll uns folgender URI dienen:

http://www.mycompany.com/employees?startIndex=0&size=10

Mit @QueryParam("startIndex") wird der Wert 0 in die Methode injiziert, mit @QueryParam("size") der Wert 10.[19]
Eine beispielhafte Anwendung der @QueryParam Annotation ist in Abschnitt 4.2.3.7 DefaultValue, zu finden.

4.2.3.4 @FormParam

Die Annotation @FormParam wird dazu verwendet, um aus einer HTML-Form, die sich im Body des HTTP-Request befindet, bestimmte Einträge auszulesen. Sendet der Client beispielweise eine HTML-Form mit dem Feld “firstname“, so kann man dies durch @FormParam("firstname") injezieren.[20]

4.2.3.5 @HeaderParam

@HeaderParam ermöglicht das Injizieren von Werten aus dem Header des HTTP-Requests. So kann man zum Beispiel den Referer durch @HeaderParam(„Referer“) auslesen.[21]

4.2.3.6 @Context

Mit der Annotation @Context ist es möglich, diverse von der JAX-RS API angebotene Hilfsobjekte zu injizieren. Dies umfasst Objekte folgender Klassen:

- javax.ws.rs.core.HttpHeaders
- javax.ws.rs.core.UriInfo
- javax.ws.rs.core.Request
- javax.servlet.HttpServletRequest
- javax.servlet.HttpServletResponse
- javax.servlet.ServletConfig,
- javax.servlet.ServletContext
- javax.ws.rs.core.SecurityContext

Listing 5 zeigt beispielhaft, wie diese Hilfsobjekte ausgelesen werden können.

Listing 5: Einsatz der @Context-Annotation

```
@GET
@Path("/{id}")
```

[19] Vgl. Burke, B.: RESTful Java with JAX-RS, S.60 f.
[20] Vgl. Burke, B.: RESTful Java with JAX-RS, S.61 f.
[21] Vgl. Burke, B.: RESTful Java with JAX-RS, S.62.

```
@Produces("application/xml")
public Employee getEmployee(@PathParam("id") String id, @Context
HttpServletRequest httpServletRequest) {
...
}
```

4.2.3.7 @DefaultValue

Bei vielen Typen von JAX-RS Services werden optionale Parameter eingesetzt. Liefert der Client diese optionalen Informationen im HTTP-Request nicht mit, so werden diese mit Standard-werten belegt. Objekte erhalten per Default den Wert null, primitive Typen den Wert 0. Oftmals kann der Server die Anfrage mit den predefinierten Werten nicht richtig verarbeiten. Mithilfe der Annotation @javax.ws.rs.DefaultValue ist es möglich, eigene Standardwerte für optionale Parameter zu definieren. Listing 6 zeigt beispielhaft, wie @DefaultValue eingesetzt werden kann. Ruft der Client die Methode getEmployees auf, so gibt ihm der Web Service eine Menge von Mitarbeitern zurück. Die Menge kann mit den Parametern „start“ (Startindex) und „size“ (Menge der gewünschten Angestellten) spezifiziert werden. Fehlen die beiden Parameter, so liefert der Web Service die ersten 10 Angestellten.

Listing 6: Vergabe von eigenen Standardwerten mithilfe @DefaultValue

```
@GET
@Produces("application/xml")
public String getEmployees(@DefaultValue("0") @QueryParam("start") int
start, @DefaultValue("10") @QueryParam("size") int size) {
...
}
```

4.2.4 Typkonversion

Wir haben bereits in einigen Beispielen die automatische Typkonversion eines Strings zu einem primitiven Typen gesehen. Listing 7 zeigt, wie der mit der @Path-Annotation versehene Teilstring des URIs ausgelesen und in einen Integer-Wert mit dem Variablennamen „id“ injiziert wird.

Listing 7: Automatische String-Konversation zu einem primitiven Typen

```
@GET
@Path("{id}")
public String get(@PathParam("id") int id) {...}
```

Neben dieser primitiven Typkonversation ist es auch möglich, die Daten des String Requests in ein Java-Objekt zu konvertieren, bevor sie in die JAX-RS Methode injiziert werden. Die Klasse des Ziel-Objekts muss hierzu entweder einen Konstruktor oder eine statische Methode mit dem Namen valueOf() enthalten. Als Parameter muss ihnen ein einfaches String-Objekt übergeben werden können.

4.2.5 JAX-RS Content Handler

In Kapitel 4.2.3 JAX-RS Parameter Injection wurde gezeigt, wie Informationen aus dem Header des HTTP-Requests und aus dem URI ausgelesen werden können. In diesem Kapitel soll gezeigt werden, wie aus dem Body des HTTP-Requests Daten ausgelesen werden können.

Der Vorgang des Verpackens eines Java-Objekts in eine XML Repräsentation wird marshalling, das Entpacken unmarshalling genannt. Dazu stellt JAX-RS einige MessageBodyWriter und MessageBodyReader bereit, die einige Java-Typen automatisch in das jeweilige Datenformat verpacken und in den Body des HTTP-Response schreiben und umgekehrt.

Tabelle 1 listet die von JAX-RS bereitgestellten MessageBodyReader und MessageBodyWriter auf. Die Tabelle kann folgendermaßen interpretiert werden: Ein String kann automatisch in jeden Medientyp konvertiert werden und jeder Medientyp kann in einen String konvertiert werden. Ein StreamingOutput kann in jeden Medientyp konvertiert werden, jedoch kann kein Medientyp in einen StreamingOutput konvertiert werden.

Tabelle 1: Bereitgestellte MessageBodyReader und MessageBodyWriter

Unterstützte Medientypen	Java Typ
Alle Medientypen (*/*)	java.lang.String
Alle Medientypen (*/*)	Java.io.InputStream, java.io.Reader
Alle Medientypen (*/*)	Javax.activation.DataSource
XML Medientypen (text/xml, application/xml, application/*+xml)	Javax.xml.transform.Source
Alle Medientypen (*/*)	Java.io.File
Alle Medientypen (*/*)	byte[]
Form Inhalt (aplication/x-www-form-urlencoded)	javax.ws.rs.core.MultivaluedMap<String,String>
Alle Medientypen (*/*), nur für MessageBodyWriter	Javax.ws.rs.core.StreamingOutput

Sollen andere Objekte automatisch in das Repräsentationsdatenformat verpackt und entpackt werden, so kann die in Kapitel 4.2.7 beschriebene Java Architecture for XML Binding verwendet werden.[26]

JAX-RS interpretiert alle Methodenparameter, die nicht mit einer JAX-RS Annotation versehen sind, als Repräsentation des Bodys eines HTTP-Request. Lediglich ein Methodenparameter kann den Body eines HTTP-Requests repräsentieren, alle anderen müssen mit einer JAX-RS Annotation versehen sein. In Listing 8 repräsentiert das Employee-Object “employee” den Body des HTTP-Requests.

4.2.6 Responses & Exception Handling

Bisher sind wir in dieser Arbeit noch nicht auf das Standardverhalten der HTTP Responses von JAX-RS Methoden im Erfolgs- bzw. Fehlerfall eingegangen. Nachfolgend werden zunächst die Default Response Codes erläutert. Anschließend wird gezeigt, wie komplexe Response-Objekte erstellt und wie Exceptions in einer JAX-RS Applikation gehandelt werden können.

4.2.6.1 Default Response Codes

Das Prinzip der Default Response Codes einer JAX-RS Applikation ist leicht zu erfassen. Anhand von Listing 8 soll dieses Prinzip näher erläutert werden.

Listing 8: @GET, @POST, @PUT, @DELETE

```
@Path("{id}")
@GET
@Produces("application/xml")
public Employee getEmployee(@PathParam("id") int id) {...}

@POST
@Produces("application/xml")
@Consumes("application/xml")
public Employee create(Employee newEmployee) {...}

@PUT
@Path("{id}")
```

```
@Consumes("application/xml")
public void update(@PathParam("id") int id, Employee employee) {...}

@Path("{id}")
@DELETE
public void delete(@PathParam("id") int id) {...}
```

Response Codes für erfolgreiche Anfragen des Clients liegen im Wertebereich von 200 bis 399. Die Methoden create() und getEmployee() geben den Response Code 200 (OK) zurück, wenn das Employee Objekt, das sie zurückliefern, nicht null ist.
War die Anfrage des Clients erfolgreich, aber ist das Employee Objekt null, so wird der Response Code 204 „NO CONTENT“ zurückgegeben. Der HTTP Code 204 ist kein „Fehler-Code“. Er teilt dem Client mit, dass sein Request erfolgreich war, die Response aber keinen Body enthält.
Ist der Rückgabewert wie in den letzten beiden Methoden update() und delete() void, so wird ebenfalls der Response Code 204 (No Content) zurückgeliefert.
Zusammenfassend kann man sagen, dass eine erfolgreiche Response, die einen Message Body beinhaltet, den Code 200 (OK) zurückliefert. Ist kein Message Body vorhanden, wird der HTTP Code 204 „NO CONTENT“ zurückgegeben.

Die Response Codes für Fehlermeldungen liegen im Wertebereich zwischen 400 und 599. Fordert der Client beim Aufruf der Methoden getCustomer() und create() in der Response die Repräsentationsformate application/json oder text/html an, liefert die JAX-RS Implementation automatisch den Fehler Code 406 (Not Acceptable). Der HTTP Code 406 bedeutet, dass die aufgerufene JAX-RS Methode das jeweils angeforderte Repräsentationsformat nicht unterstützt.

4.2.6.2 Komplexe Responses

Response-Objekte als Rückgabewert werden vor allem dann genutzt, wenn man den HTTP-Response, der an den Client zurück gesendet wird, noch modifizieren möchte.

Die Klasse Response ist abstrakt und enthält drei simple Methoden.

- **getEntity()**
  liefert das Java-Objekt, das in den HTTP Message Body konvertiert werden soll.

- **getStatus()**
  gibt den HTTP Response Code zurück

- **getMetaData()**
  ist eine MultivaluedMap<Object, String>

Response Objekte können nicht direkt instanziiert werden. Sie werden mithilfe von Instanzen von javax.ws.rs.core.Response.ResponseBuilder erstellt. Auf diese kann über statische Helfer-Methoden der abstrakten Klasse Response zugegriffen werden.
Listing 9 zeigt, wie man ein Response-Objekt mit einer modifizierten Antwort und dem Statuscode 202 (Accepted) erzeugt.

Listing 9: Erzeugung eines modifizierten Response-Objekts

```
@PUT
@Path("/{id}")
@Consumes("application/xml")
public Response createEmployee(@PathParam("id") String id,
Employee employee) {
// TODO Insert into Database
Response.ResponseBuilder builder = Response.ok("Insert into Database
successful", MediaType.TEXT_PLAIN_TYPE);
```

```
builder.status(Response.Status.ACCEPTED);
return builder.build();
}
```

4.2.6.3 Exception Handling

Fehler in der Anwendung können dem Client entweder durch das Erzeugen und Senden eines entsprechenden Response-Objekts mitgeteilt werden oder durch Werfen einer Exception. Die Applikation kann jede geprüfte (alle abgeleiteten Klassen von java.lang.Exception) und ungeprüfte (alle abgeleiteten Klassen von java.lang.RuntimeException) werfen, die sie will. Zur Laufzeit geworfene Exceptions werden nur dann von JAX-RS gehandelt, wenn man einen Exception Mapper registriert hat. Exception Mapper können eine Exception auf einen bestimmten HTTP-Response abbilden. Werden die geworfenen Exceptions nicht vom Exception Mapper gehandelt, übernimmt das Handling der Container, in dem JAX-RS läuft.

Es gibt auch die Möglichkeit eine javax.ws.rs.WebApplicationException zu werfen, die automatisch von der JAX-RS Laufzeit verarbeitet wird. Dieser Exception kann man einen Response-Code im Konstruktor beim Erzeugen übergeben. Wird kein Response-Code angegeben, so wird standardmäßig der Fehlercode 500 (Internal Server Error) an den Client gesendet. Listing 10 zeigt, wie so eine WebApplicationException geworfen werden kann. Wird im Beispiel kein Employee mit der übergbenen ID gefunden, wird der HTTP Code 404 „Not Found“ an den Client gesendet.

Listing 10: Nach Fehlerursache customized WebApplicationException

```
@GET
@Path("/{id}")
@Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
public Employee getEmployeeById(@PathParam("id") int id) {
	Employee employee = DataAccess.getEmployeeById(id);
	if(employee == null) {
		throw new WebApplicationException(Response.Status.NOT_FOUND);
	}
	return employee;
}
```

Will man seine Exceptions über einen Exception Mapper handeln, so muss man eine Klasse implementieren, die das Interface javax.ws.rs.ext.ExceptionMapper implementiert. Die implementierende Klasse muss die Methode toResponse(E exception) implementieren, die als Rückgabewert ein Response-Objekt hat. So kann man verschiedene, in der Anwendung geworfene Exceptions handeln und je nach Exception einen speziellen Response-Code oder eine Nachricht im Response mitzugeben. Die Klasse, die das Interface ExceptionMapper implementiert, muss mit @Provider annotiert sein. Diese Annotation sagt der JAX-RS Laufzeitumgebung, dass es sich hierbei um eine Komponente handelt, die mit der Applikation deployt werden soll.[22]

Listing 11 zeigt einen Exception Mapper, der die vom Framework **J**ava **P**ersistence **A**PI (JPA) geworfene javax.persistence.EntityNotFoundException handelt und sie auf ein Response-Objekt mappt.

Listing 11: ExceptionMapper mappt JPA-Exception auf Response-Objekt

```
@Provider
public class EntityNotFoundMapper
     implements ExceptionMapper<EntityNotFoundException> {
```

[22] Vgl. Burke, B.: RESTful Java with JAX-RS, S. 102 ff.

```
 public Response toResponse(EntityNotFoundException e) {
      return Response.status(Response.Status.NOT_FOUND).build();
   }
}
```

4.2.7 Java Architecture for XML Binding

Die **J**ava **A**rchitecture for **X**ML **B**inding, kurz JAXB, ist eine Spezifikation, bzw. ein Framework, das nicht in der JAX-RS Spezifikation enthalten ist. JAXB bietet dem Entwickler eine einfache Möglichkeit, Java-Klassen mittels Annotationen an XML bzw. ein XML-Schema zu binden. Objekte einer Klasse können somit automatisch in das XML-Repräsentationsformat konvertiert werden. Umgekehrt ist eine automatische Konvertierung von XML in Java-Objekte ebenfalls möglich. Listing 12 zeigt, wie in der Methode createEmployee() aus dem konsumierten XML, durch den JAXB Provider ein Objekt der Klasse Employee erzeugt und in der Methode anschließend als Parameter zur Verfügung gestellt wird.[23]

Listing 12: Automatischer Konvertierung durch den JAXB Provider

```
@POST
@Path("/insert")
@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
@Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public static Response createEmployee(Employee e) {
      DataAccess.insertEmployee(e);
      …
      }
```

Nachfolgend wird auf die wichtigsten Annotationen im Umgang mit JAXB eingegangen.

4.2.7.1 JAXB Annotationen

Um die automatische Konvertierung von einem Java-Objekt in das XML-Format (und umgekehrt) zu initialisieren, muss eine Java-Klasse mit entsprechenden Annotationen versehen werden. Listing 13 zeigt beispielhaft, wie die Klasse Employee annotiert werden muss, damit eine automatische Abbildung auf das XML-Zielformat erfolgen kann. Zu beachten ist, dass die Klasse Employee ein Attribut der Klasse Address beinhaltet. Bis auf die @XmlAttribute-Annotation müssen in der Klasse Address die gleichen Annotationen vorgenommen werden. Die XmlAttribute-Annotation ist in der Klasse Address nicht nötig, da eine Adresse eindeutig einem Angestellten zugeordnet werden kann.
Um die Java-Objekte in das ebenfalls sehr populäre JSON-Format zu konvertieren, werden die gleichen Annotationen verwendet. Es wird jedoch noch ein JAXB bzw. Objektmapper benötigt, wie beispielsweise die Open Source Projekte Jettison oder Jackson.[24] Listing 15 zeigt das in Listing 14 beschriebene XML-Format im JSON-Format. Dort ist zu sehen, warum sich das Repräsentationsformat JSON sich immer größer werdender Beliebtheit erfreut. Es ist einfach zu lesen und zu schreiben und produziert deutlich weniger Overhead als das XML-Format.

Listing 13: JAXB Annotationen in der Employee Klasse

```
…
// Imports

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Employee {
      @XmlAttribute
      private int id;
```

[23] Vgl. Burke, B.: RESTful Java with JAX-RS, S.77

[24] Vgl. Burke, B.: RESTful Java with JAX-RS, S.78

```
        private String firstname;
        private String lastname;
        private int age;
        @XmlElement(name = “wage”)
        private double salary;
        private Address address;

// Getter and Setter
…
}
```

4.2.7.1.1 @XmlRootElement

Die Annotation @XMLRootElement deklariert die Klasse als XML-Element. In Listing 13 wird das name() Attribut verwendet, um das Employee-Objekt auf ein XML-Element namens <employee> zu mappen.[25]

4.2.7.1.2 @XMLAccessorType

Mit der Annotation @AccessorType wird angegeben, wie die Daten eines Objektes an ein XML-Element bzw. XML-Attribut gebunden werden. So werden beispielsweise mit XmlAccessType.PUBLIC_MEMBER nur öffentliche Variablen in XML abgebildet. XmlAccessType.PROPERTY dagegen berücksichtigt alle Objektattribute, die über Getter oder Setter verfügen. Man beachte, dass dies nicht alle Accessortypen sind.[26]

4.2.7.1.3 @XmlAttribute

Mithilfe der @XmlAttribute-Annotation wird das damit versehene Feld als Attribut des XmlRootElements gekennzeichnet.
Im obigen Beispiel teilen wir JAXB mit, dass das annotierte Feld id auf ein gleichnamiges Attribut des Elements <employee> im XML Dokument gemappt werden soll.[27]

4.2.7.1.4 @XmlElement

In der Regel mappt JAXB die Java-Attribute automatisch auf ein gleichnamiges XML- Element. Sollen jedoch noch Kardinalitäten angegeben oder das Attribut auf ein anders heißendes XML-Element gemappt werden, so kann dies mithilfe der Annotation @XmlElement erfolgen.[28]

Listing 14: XML Repräsentation

```
<employee id ="31">
        <firstname>Heinrich</firstname>
        <lastname>Maier</lastname>
        <age>40</age>
        <wage>5550.0</wage>
        <address>
                <street>Lothstr.</street>
                <streetNr>64</streetNr>
                <zipcode>80335</zipcode>
                <place>München</place>
        </address>
</employee>
```

Listing 15: Repräsentation des Employees mit der ID 31 im JSON Format

```
{
```

[25] Vgl. Burke, B.: RESTful Java with JAX-RS, S. 78
[26] Vgl. Burke, B.: RESTful Java with JAX-RS, S. 78
[27] Vgl. Burke, B.: RESTful Java with JAX-RS, S. 78 f.
[28] Vgl. Burke, B.: RESTful Java with JAX-RS, S. 78 f.

```
@id: "31"
firstname: "Heinrich"
lastname: "Maier"
age: "40"
wage: "5550.0"
address:
{
street: "Lothstr."
streetNr: "64"
zipcode: "80335"
place: "München"
}
}
```

4.3 Deployment

Um eine JAX-RS Applikation zu deployen, gibt es zwei Möglichkeiten. Entweder wird die Applikation innerhalb eines eigenständigen Servlet Containers wie Apache Tomcat oder Jetty deployed oder man nutzt den Servlet Container eines Application Servers wie JBoss, Websphere oder Glassfish.
Ein Servlet Container ist vergleichbar mit einem Web Server. Er basiert auf dem HTTP Protokoll und stellt ein low-level Komponenten Modell (das Servlet API) zur Verfügung, um HTTP Requests zu empfangen. Servlet-basierte Applikationen werden zu einem **W**eb **AR**chive, kurz WAR zusammengefasst. Ein WAR ist ein JAR-basiertes Packformat, dass die Java-Bibliotheken und –Klassen, sowie statische Inhalte wie HTML Dateien enthält.
Nachfolgend sei die Dateistruktur des WARs dargestellt.

<statischer Inhalt>
WEB-INF/
    web.xml
    classes/
    lib/

Alle Dateien außerhalb und über dem WEB-INF/ Ordner werden veröffentlicht und sind direkt über HTTP zugänglich. Dort wird der gesamte statische Inhalt wie HTML-Seiten oder Bilder, der nach außen hin preisgegeben werden soll, abgelegt.
Der Ordner WEB-INF hat zwei Unterordner. Im Ordner classes werden die Java-Klassen, in lib alle für die Applikation notwendigen Bibliotheken abgelegt. Der WEB-INF Ordner enthält zusätzlich eine Datei namens web.xml, den sogenannten Deployment Descriptor. Ein Deployment Descriptor beschreibt die Klassen, Ressourcen und die Konfiguration der Applikation und wie der Web Server diese nutzt, um mit Anfragen umzugehen. Listing 16 zeigt, wie so web.xml aussehen kann. Mittels dem `<servlet>` Tag wird das Servlet konfiguriert. Anschließend wird mithilfe von `<servlet-mapping>` angegeben, wie das Servlet erreicht werden kann.

Listing 16: Beispiel einer web.xml. Zu finden im Ordner WEB-INF

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
<display-name>EmployeeRESTService</display-name>
<servlet>
       <servlet-name>EmployeeRESTService</servlet-name>
       <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
       <init-param>
              <param-name>com.sun.jersey.config.property.packages</param-name>
              <param-value>com.mycompany.rest.restservices</param-value>
       </init-param>
```

```
</servlet>
<servlet-mapping>
       <servlet-name>EmployeeRESTService</servlet-name>
       <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
</web-app>
```

Arbeitet man mit einem JEE6-konformen Servlet-Container wie Tomcat 7+ oder Glassfish 3+, so ist die Angabe bzw. Konfiguration des REST-Servlets im Deployment Descriptor nicht mehr „State of the Art“. Stattdessen erfolgt die Konfiguration der Applikation über eine mit JAX-RS Annotationen versehene Klasse. Der Container fügt die Ressourcen-Klassen der Anwendung dann automatisch hinzu. Listing 17 zeigt die Java-Klasse, welche die Konfiguration aus Listing 16 überflüssig macht.

Listing 17: Java-Klasse zur Konfiguration des Web Services

```
package com.mycompany.rest.restservices;

import java.util.Set;
import javax.ws.rs.core.Application;
import javax.ws.rs.ApplicationPath;

@ApplicationPath("/rest")
public class ApplicationConfig extends Application {

          @Override
          public Set<Class<?>> getClasses() {
              return getRestResourceClasses();
          }

          private Set<Class<?>> getRestResourceClasses() {
              Set<Class<?>> resources = new java.util.HashSet<Class<?>>();

resources.add(com.mycompany.rest.restservices.EmployeeResource.class);
              return resources;
          }
}
```

Um die Web Service Konfiguration zu erhalten, muss die Konfigurations-Klasse von javax.ws.rs.core.Application ableiten.
Mit der Annotation `@ApplicationPath("/rest")` geben wir den Pfad der Applikation an. Das heißt, wir leiten alle Anfragen, die nach dem Server URI (*http:www.mycompany.com*) mit *„/rest“* beginnen, auf unser Servlet um. Die Funktion von `@ApplicationPath` ist gleichbedeutend mit der des Elements `<url-pattern>/rest/*</url-pattern>` in der web.xml.
Im Anschluss wird die Resource-Klasse mit resources.add(Resource.class) hinzugefügt.

4.4 JAX-RS – Client API

Im Januar 2011 beauftragte JCP die JSR 339 Expertengruppe, die Java API for RESTful Web Services, kurz JAX-RS Version 1.0 weiterzuentwickeln. JAX-RS 2.0 liegt derzeit in der Version „proposed final draft“[29] vor und wird Teil der im Jahr 2013 veröffentlichten JavaEE 7 Spezifikation sein. Mit JAX-RS 2.0 wird erstmals, neben anderen neuen Features, auch ein Client Framework mit ausgeliefert. Während in Version 1.0 jede einzelne JAX-RS Implementierung ihre eigene proprietäre Schnittstelle schreiben musste, gibt es nun eine flüssige, low-level, request-building API. Dieses Kapitel dient dem Leser, ein grundlegendes Verständnis für die Funktionsweise einer RESTful Client-API aufzubauen. Es sei darauf hingewiesen, dass Änderungen in der Client-API bis zum endgültigen Release durchaus möglich sind.

4.4.1 Instanziierung des Clients

Die Client-API ist im Paket javax.ws.rs.client.Client zu finden. Nach dem Import des Packages kann der Client durch einen Aufruf der statischen Methode newClient() in der Klasse ClientFactory instanziiert werden. [30]

4.4.2 Ressourcenzugriff

Der Client kann auf die entsprechende Web Ressource zugreifen, in dem Methodenaufrufe aneinander gekettet werden. In Listing 18 erfragt der Client über die URL *http://www.mycompany.com/employee/5* die Repräsentation eines Employees im XML-Format.

Listing 18: Anfrage der Employee Ressource im XML-Format

```
Client client = ClientFactory.newClient();
Employee e = client
        .target("http://www.mycompany.com/employee/5")
        .request(MediaType.APPLICATION_XML).get(Employee.class);
```

Zur Durchführung einer erfolgreichen Anfrage sind folgende Schritte nötig:

- Instanziieren des Clients
- Erstellen des WebTargets
- Erstellen des Client Requests auf dem WebTarget
- Versenden des Client Requests

Zu beachten ist, dass die Abfrage nicht sofort versendet werden muss, sondern auch für die spätere Verwendung vorbereitet werden kann. In Abschnitt *5.5 Invocation* wird diese Alternative näher beleuchtet.
Die Aneinanderreihung von Methodenaufrufen ist weit vielschichtiger, als dies in Listing 12 der Fall ist. So kann eine Anfrage beispielsweise durch Header, Cookies oder Query Parameter noch weiter spezifiziert werden. Listing 19 zeigt, wie Informationen mithilfe eines Query Parameters (siehe Kap. 4.2.3.3) bzw. als Headerinformation in das Request eingebettet werden können.

Listing 19: Komplexe Anfrage einer Ressource

```
Client client = ClientFactory.newClient();
Response res = client.target("http://www.mycompany.com/employee")
                        .queryParam("name", "value")
                        .request("text/plain")
                        .header("MyHeader", "...")
                        .get();
```

[29] Stand: 21. Februar 2013
[30] Vgl. Pericas-Geertsen, S.: JAX-RS: Java API for RESTful Web Services, S.33

4.4.3 Targets

Targets eignen sich hervorragend zur Bildung komplexer URIs. So kann beispielsweise der Basis URI mit zusätzlichen Pfadsegmenten oder Templates erweitert werden. Listing 13 zeigt ein Anwendungsbeispiel.

Listing 20: Konkatenation von Targets

```
Target base = client.target("http://www.mycompany.com/employee");
Target division = base.path("division").path("{whom}");
Response res = division.pathParam("whom", "5000").request("...").get();
```

Obiges Beispiel liefert eine Repräsentation der Ressource, die durch den eindeutigen URI *http://www.mycompany.com/employee/division/5000* identifiziert wird.
Man beachte die Nutzung des Template Parameters {whom}.

Instanzen der Klasse Target sind in Bezug auf Ihren URI unveränderlich. Methoden, die den Pfad mittels zusätzlichen Pfadsegmenten und Parametern spezifizieren, liefern eine neue Target-Instanz.
In Bezug auf deren Konfiguration dagegen, sind Targets veränderlich. Dieser Umstand ermöglicht die Bildung komplexer Vererbungshierarchien. In Listing 14 werden zwei Target-Instanzen erstellt. Die Instanz division erbt die Konfiguration der Instanz base und kann nun weiter konfiguriert werden. Änderungen auf der hello-Instanz wirken sich nicht auf die base-Instanz aus.[31]

4.4.4. Typisierte Entities

Der Response eines Requests ist nicht zwangsläufig vom Typ Response. In Listing 21 wird der Mitarbeiter mit der ID 1 in die Abteilung C versetzt. Hierzu wird zuerst die Entität vom Typ Employee des entsprechenden Mitarbeiters angefragt. Anschließend wird die Entität des Mitarbeiters per POST-Methode wieder an den Server geschickt und in die Datenbank geschrieben.

Listing 21: Typisierte Entity im Response Objekt

```
Client client = ClientFactory.newClient();
Employee e = client.target("http://www.mycompany.com/employee/1")
                   .request(MediaType.APPLICATION_XML).get(Employee.class);
String diversion = client.target("http://www.mycompany.com/employee/div-C")
                   .request().post(xml(e), String.class);
```

Zu beachten ist die Methode xml(e) im POST-Aufruf. Die Klasse javax.ws.rs.client.Entity definiert die Varianten für die gängigsten Medientypen, die in JAX-RS genutzt werden. Wie in der Server API sind die JAX-RS Implementierungen auf den Gebrauch von Entity Providern angewiesen.[32]

4.4.5 Invocations

Eine Invocation (dt. Aufruf) ist ein Request, das bereits vorbereitet und bereit für die Ausführung ist. Invocations stellen ein generisches Interface bereit, welches das Prinzip „seperation of concerns“ zwischen dem Ersteller der Invocation und der ausführenden Instanz ermöglicht. Die ausführende Instanz muss nicht wissen, wie die Invocation vorbereitet wurde, sondern nur, ob sie synchron oder asynchron ausgeführt werden soll.[33]

Ein Client Request Invocation Builder bietet entsprechende Methoden, um die Invocation vorzubereiten. Listing 22 zeigt beispielhaft, wie eine solche Invocation vorbereitet und

[31] Vgl. Pericas-Geertsen, S.: JAX-RS: Java API for RESTful Web Services, S.34
[32] Vgl. Pericas-Geertsen, S.: JAX-RS: Java API for RESTful Web Services, S.34 f.
[33] Vgl. Pericas-Geertsen, S.: JAX-RS: Java API for RESTful Web Services, S.35

ausgeführt werden kann. Die vorbereitete Invocation liefert eine Liste von Mitarbeitern zurück. Im Anschluss ist es der ausführenden Instanz überlassen, diese synchron oder asynchron auszuführen. Die Methode buildGet() des Invocation Builders dient der Entkopplung der Invocation vom GET-Request.

Listing 22: Vorbereitung und Ausführung der Invocation

```
Invocation inv; inv=client.target("http://www.mycompany.com/employee")
.request("application/xml").buildGet();

            //synchron
List<Employee> empList1 = inv.invoke(new GenericType<List<Employee>>() {});

            //asynchron
Future<List<Employee>> empList2 = inv.submit(new GenericType<List<Employee>>()
{});
try {
      List<Employee> test = empList2.get();
} catch (InterruptedException | ExecutionException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
}
```

Zu beachten ist, dass die asynchrone Ausführung der Invocation ein Objekt des Typen *Future* zurückliefert. Dieses stellt Methoden zur Verfügung, die für die asynchrone Ausführung benötigt werden. So kann beispielsweise geprüft werden, ob die Ausführung bereits ausgeführt worden ist. Das Ergebnis kann über die Methode get() auf dem Future-Objekt abgerufen werden. Voraussetzung hierfür ist die vollständige Ausführung der Invocation. 4.4.6 Konfigurierbare Typen

Einige Typen der Client API sind tief konfigurierbar. Zu den konfigurierbaren Typen gehören *Client*, *Invocation*, *Invocation.Builder* und Target. Die Konfigurationsmethoden werden vom Interface *Configurable* geerbt und von allen Klassen implementiert.[34]
Das Interface Configurable unterstützt die Konfiguration von

- **Properties** Name/Wert-Paare für die zusätzliche Konfiguration von Features oder anderen Komponenten der JAX-RS Implementation
- **Features** Ein spezieller Typ eines Providers, der das Interface *Feature* implementiert und zur Konfiguration der JAX-RS Implementation dient
- **Providers** Klassen oder Instanzen, die eines oder mehrere Interfaces von Providern Implementieren. Klassische Provider[35] sind
  - ✓ Entity Provider wie Message Body Reader und Writer
  - ✓ Context Provider
  - ✓ Exception Mapping Provider

Die Konfiguration einer Instanz der obenerwähnten Typen wird von anderen Instanzen, die auf deren Basis erstellt werden, geerbt. Zum Beispiel erbt eine Instanz des Typen Target die Client Konfiguration von dem Client, von dem sie instanziiert worden ist. Man beachte, dass diese zwei Instanzen weiterhin unabhängig voneinander existieren. Wird beispielsweise die Target Instanz verändert, so wirkt sich das nicht auf den Client aus und umgekehrt (deep copy).

[34] Vgl. Pericas-Geertsen, S.: JAX-RS: Java API for RESTful Web Services, S.36

[35] Für mehr Informationen bzgl. der Provider Klassen sei auf die JAX-RS 2.0 Spezifikation verwiesen.

5. Fazit

REST ist ein sehr einfacher, leichtgewichtiger Architekturstil, der keine hohe Lernkurve besitzt und daher schnell zu verstehen ist. Auch ohne nennenswerte Erfahrungen mit Web Services war es möglich, schnell eine funktionierende Service Schnittstelle zu implementieren. REST basiert auf Standards wie HTTP, Hypertext und URIs, die sich bereits seit Jahrzenten bewährt haben und macht sich somit die Ideen des World Wide Webs zunutze. Da vor allem das Hypertext Transfer Protokoll stark genutzt wird, ist die Spezifikation eines eigenen Protokolls überflüssig. REST benötigt im Vergleich zu SOAP kein großes XML Format zur Beschreibung der Daten, der Datentypen und der aufzurufenden Methode. Des Weiteren skalieren RESTful Web Services auch bei großer Nutzeranzahl sehr gut und unterstützen die Kommunikation von verteilten Systemen, unabhängig von deren Programmiersprache.

Die Java API for RESTful Web Services bietet Java-Entwicklern einfache, schnelle und komfortable Möglichkeiten zur Entwicklung eines RESTful Web Services. Durch die Verwendung eines JAX-RS Frameworks können Java-Klassen mit geringem Konfigurationsaufwand als Web Service deklariert werden. Die URI zu diesem Web Service, sowie die HTTP-Operation können einfach an die Java-Methode gebunden werden. Dieser annotationsbasierte Ansatz macht es möglich, dass man sich im Vergleich zu SOAP das Schreiben von viel Quellcode erspart und somit die Fehleranfälligkeit des Web Services verringert. Auch die Entwicklungszeit wird erheblich verkürzt. Durch die Parameter Injection muss sich der Entwickler nicht um die Konvertierung von XML oder JSON in ein Java-Objekt (und umgekehrt kümmern), sondern kann einfach die bereitgestellten Message Body Reader und Message Body Writer verwenden. Er bekommt damit direkt fertige Java-Objekte in seinen Web Service injiziert, mit denen sofort gearbeitet werden kann. Auch die Content Negotiation, wodurch den Präferenzen des Clients in Bezug auf das ausgetauschte Datenformat Rechnung getragen wird, sowie das Erzeugen eines modifizierten Response-Objektes werden hervorragend integriert. Anwendungsspezifische Exceptions können automatisiert in spezifische HTTP-Responses umgewandelt werden.
Lediglich das **H**ypermedia **A**s **T**he **E**ngine **O**f **A**pplication **S**tate-Prinzip, also das Verlinken anderer Web Services in der Repräsentation einer Ressource, wird nicht ausreichend unterstützt. Man kann die URIs anderer Web Service zwar programmatisch in den Repräsentationen verlinken, jedoch nicht deklarativ.
Für die Implementierung der Web Service Schnittstelle waren die vielen Tutorials und Beispielprojekte, die es mittlerweile im World Wide Web für die JAX-RS 1.0 Spezifikation zu finden gibt, äußerst nützlich. Zum Testen der Web Service Schnittstelle, die im Rahmen dieser Studienarbeit implementiert wurde, haben sich Tools wie Google „Chrome`s Advanced REST Client“ oder auch „SoapUI“ bewährt. Selbstverständlich ist und bleibt das Schreiben von Testklassen, beispielsweise mit jUnit, das Mittel erster Wahl.
Die in Java EE 7 integrierte JAX-RS 2.0 Spezifikation bietet dem Entwickler erstmals die Möglichkeit, auf einfache Art und Weise die Client Seite zu implementieren. Die gesamte Client API ist einfach verwendbar und doch sehr mächtig. Durch die bereits existierenden Message Body Reader und Writer ist es ohne weiteres möglich, diverse Datenformate bereitzustellen und abzufragen. Das Target-Prinzip erlaubt es dem Entwickler der Clientseite, auch komplexere Anfragen übersichtlich und simpel zu generieren. Die Kapselung der Requests mithilfe des Invocation Builders ist eine weitere sinnvolle Erweiterung von JAX-RS 2.0. Die hierdurch erlangte **S**eparation **o**f **C**oncerns (SoC) ist ein mächtiges Design Prinzip, dass den Entwicklern bei der Modularisierung ihrer Aufgaben entgegenkommen wird. Zu bemängeln sind Umfang und Aktualität der derzeitigen JAX-RS 2.0 Spezifikation sowie die dazugehörigen JAR Files. Während der Implementierung des REST-Clients kamen viele Fragen auf, die nur unzureichend oder überhaupt nicht durch die Spezifikation geklärt werden konnten. Hier half oftmals nur das altbewährte Prinzip „Try and Error“. Dennoch ist die Simplizität von REST unübertroffen. Wurden die Grundprinzipien erst einmal verinnerlicht, so steht einer erfolgreichen Implementation nichts mehr im Weg.

Abbildungsverzeichnis

Listingverzeichnis

Literaturverzeichnis

REST Patterns, MIME Type. (2008). Abgerufen am 04. April 2013 von http://restpatterns.org/Glossary/MIME_Type

redhat: RESTEasy JAX-RS, RESTFul Web Services for Java. (2012). Abgerufen am 13. April 2013 von http://docs.jboss.org/resteasy/docs/2.3.1.GA/userguide/html_single/index.html

Oracle: The Java EE 6 Tutorial, Stand 2012. (02. April 2013). Von http://www.docs.oracle.com/javaee/6/tutorial/doc/gijvh.html abgerufen

Burke, B. (2009). *RESTful Java with JAX-RS.* Sebastapol: O`Reilly.

Oracle: JavaTM Platform, Enterprise Edition 6 API Specification, Package javax.ws.rs, Stand Januar 2013. (kein Datum). Abgerufen am 02. April 2013 von http://docs.oracle.com/javaee/6/api/javax/ws/rs/package-summary.html

Pericas-Geertsen, S. (2013). *JAX-RS: Java API for RESTful Web Services*. Abgerufen am 29. April 2013 von http://download.oracle.com/otn-pub/jcp/jaxrs-2_0-pfd-spec/339-spec.pdf?AuthParam=1367262282_3929e3836cff9fc3fcaad2dae38cd0a8

Snell, J. (2001). *Programming Web Services with SOAP.* Sebastopol: O`Reilly.

W3C: Hypertext Transfer Protocol - HTTP/1.1. (kein Datum). Abgerufen am 04. April 2013 von http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html