RESTful Web Services mit Java

Prinzipiell lassen sich REST-konforme Anwendungen schon seit langem mit dem Servlet API realisieren – schließlich ist die Grundvoraussetzung nur die Unterstützung von Webstandards wie HTTP und URIs. Sun hat aus diesem Grund (und ganz sicher auch wegen des damals beginnenden REST-Hypes) im Februar 2007 den JSR 311 ins Leben gerufen, der sich mit der Spezifikation für ein explizit REST-orientiertes API beschäftigt.

von Stefan Tilkov

Seit Oktober 2008 liegt „JAX-RS: The Java API for RESTful Web Services“ in der finalen 1.0-Version vor, und auch eine Handvoll Implementierungen ist bereits verfügbar. Ein wesentliches Ziel der Spezifikation ist es, das Verfolgen der REST-Prinzipien einfacher zu machen als deren Missachtung. In diesem Artikel wird JAX-RS kurz vorgestellt und beleuchtet, wie weit dieses Ziel erreicht wird. Zu den Zielen von JAX-RS gehört neben der bereits erwähnten REST-Konformität Einfachheit bei der Entwicklung, das Ausnutzen moderner Java-Sprachmittel (insbesondere Annotations), die Unterstützung unterschiedlicher Ablaufumgebungen (Java SE und EE) und vor allem die Möglichkeit, auf alle Konstrukte von HTTP zugreifen zu können. JAX-RS beschreitet damit einen anderen Weg als viele andere Java-Spezifikationen: Auf eine Kapselung bzw. Abstraktion des darunterliegenden technischen Protokolls wird bewusst verzichtet. Wer JAX-RS verwendet, muss sich mit REST und HTTP auseinandersetzen – dann, so die Leitlinie, unterstützt JAX-RS ihn perfekt. Bewusst ausgeklammert wurde die Clientseite des APIs, d.h. JAX-RS beschäftigt sich nur mit der Entwicklung von Serveranwendungen (oder Services). Schließlich wird keine Festlegung auf ein bestimmtes Datenformat getroffen: Natürlich lassen sich XML und die entsprechenden APIs wie JAXB auch in Verbindung mit JAX-RS nutzen, ebenso möglich ist aber der Versand binärer Inhalte oder die Nutzung von einfachem Text oder JSON. Damit unterscheidet sich der bei JAX-RS gewählte Ansatz deutlich von dem populärer Web-Services-Implementierungen wie Apache Axis2, in denen REST (fälschlicherweise) als Implementierungsdetail betrachtet wird.

Grundkonzepte: Ressourcenklassen und Annotationen

Die Aufgabe, die JAX-RS erledigen muss, ist die Vermittlung zwischen den an einzelne Ressourcen adressierten HTTP-Requests und den Methoden entsprechender Ressourcenklassen. Mit

Listing 1: JAX-RS „Hello World“

```
package com.innoq.jersey.helloworld;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Path("/helloworld")
public class HelloWorldResource {

  @GET @Produces("text/plain")
  public String sayHello() {
    return "Hello World\n";
  }

  @GET @Produces("text/html")
  public String sayHelloInHtml() {
    return "<html><title>Hello, world</title>
                              <body><h2>Hi!</h2></body></ht-
ml>";
  }
}
```

diesem Begriff werden in JAX-RS Java-Klassen bezeichnet, die über die uniforme HTTP-Schnittstelle angesprochen werden können. Eine Art von Ressource entspricht dabei jeweils einer eigenen Java-Klasse, die mit Annotationen auf URIs, HTTP-Methoden und MIME-Content-Typen abgebildet wird. Das kanonische „Hello World"-Beispiel ist in Listing 1 dargestellt, den vollständigen Sourcecode finden Sie zum Download unter [2].

Die Klasse *HelloWorldResource* ist eine einfache Java-Klasse, die Methoden beliebigen Namens enthalten kann. Mit der Annotation *@Path* wird sie unter dem URI */helloworld* „verankert". Dieser URI wird mit der Basis-URI des Services verkettet. Die beiden freundlichen Grußmethoden unterstützen als HTTP-Methode ein *GET*. Dies wird durch die *@GET*-Annotation festgelegt. Schließlich wird deklariert, dass die eine Methode den MIME-Typ *text/plain* und die andere *text/html* produziert – welche Methode aufgerufen wird, entscheidet die Laufzeitumgebung anhand des vom Client übermittelten Accept Headers. Hat man eine solche Ressourcenklasse z.B. in einem Servlet-Container auf Port 9090 unter */JerseyTest* deployt, lässt sich mit dem Kommandozeilenwerkzeug *curl* (dem Schweizer Taschenmesser des HTTP-Entwicklers) der Dienst wie folgt aufrufen (*-i* zeigt Header-Informationen an, mit *-H* setzt man einen HTTP-Header):

```
curl -i http://localhost:9090/JerseyTest/
                    helloworld -H 'Accept: text/plain'
```

Als Antwort liefert der Server:

```
HTTP/1.1 200 OK
Content-Type: text/plain

Hello World
```

Fordert man per *curl* eine HTML-Repräsentation an oder kopiert den URI einfach in die Browseradresszeile, sieht die Antwort erwartungsgemäß anders aus:

```
curl -i http://localhost:9090/JerseyTest/
                    helloworld -H 'Accept: text/html'
HTTP/1.1 200 OK
Content-Type: text/html
html>title>Hello, world/title>
  body>h2>Hi!/h2>/body>/html>
```

Die Parameter für *@Path*-Annotationen sind *URI Templates* und entsprechen einem zwar noch nicht offiziellen, aber dennoch recht verbreiteten Standard, mit dessen Hilfe man die variablen Anteile eines URI

Listing 2: Zugriff auf URI-Elemente

```
package com.innoq.jaxrs.params;

import javax.ws.rs.*;

@Path("/params/{segment1}")
public class ParamsResource {

  @GET @Path("/{segment2}")
  @Produces("text/plain")
  public String getParams(
        @PathParam("segment1") String segment1,
        @PathParam("segment2") String segment2,
        @QueryParam("q") String q) {
          return "Segment 1: " + segment1 + "\n" +
                "Segment 2: " + segment2 + "\n" +
                "Query: " + q + "\n";
  }
}
```

Listing 3: Erste Version der Customers-Ressource

```
@Path("/procurement1/customers/")
public class CustomersResource {
 private final static String ns =
              "http://innoq.com/ns/procurement-demo";

 @GET
 @Produces("application/vnd.innoq.customers+xml")
 public String getAsXml() {
   List<Customer> customers = Customer.findAll();
   Element root = new Element("customers", ns);
   for (Customer customer : customers) {
     Element customerElement
       = new Element ("customer", ns);
     customerElement.appendChild(customer.getName());
     root.appendChild(customerElement);
   }
   return elementToXmlString(root);
 }

 @POST
 @Consumes("application/vnd.innoq.customer+xml")
 public Response newCustomer(String body) {
   Builder b = new Builder();
   try {
     System.out.println("Received: " + body);
     Document doc = b.build(body, ".");
     Customer c = new Customer(doc.query
                              ("/i:customer/i:name",
         new XPathContext("i", ns)).get(0).getValue());
     Customer.add(c);
     return Response.ok().build();

   } catch (Exception e) {
     e.printStackTrace();
     return Response.status(400).entity
("Please send well-formed XML\n").
                              type("text/plain").build();
   }

 }
}
```

Listing 4: MessageBodyWriter

```
@Provider
@Produces("application/vnd.innoq.customers+xml")
public class CustomerListWriter
  implements MessageBodyWriter<CustomerList> {
 public long getSize(CustomerList t,
                     Class<?>type,
              Type genericType,
              Annotation[] annotations,
              MediaType mediaType) {
  return -1;
 }

 public boolean isWriteable(
              Class<?> type,
              Type genericType,
              Annotation[] annotations,
              MediaType mediaType) {
    return CustomerList.class == type;
 }

 public void writeTo
     (CustomerList customers, Class<?> type,
      Type genericType, Annotation[] annotations,
      MediaType mediaType, MultivaluedMap
            <String, Object> httpHeaders,
      OutputStream entityStream)
   throws IOException, WebApplicationException {
   Element root = new Element("customers", NAMESPACE);
   for (Customer customer : customers) {
     if (customer != null) {
       Element customerElement =
            new Element("customer", NAMESPACE);
       customerElement.appendChild(
            customer.getName());
       root.appendChild(customerElement);
     }
   }

   writeElementToStream(root, entityStream);
 }
}

}
```

spezifizieren kann. Die einzelnen Elemente aus einem URI lassen sich mithilfe weiterer Annotationen in Methodenparameter übertragen. Pfadannotationen können sowohl für die Klasse als auch für ihre Methoden verwendet werden (Listing 2).

Bislang habe ich nur den einfachsten Fall dargestellt, die Abbildung eines *HTTP GET*. An einem weiteren Beispiel können wir auch die andere Richtung der Datenverarbeitung illustrieren – das Verarbeiten von Daten, die per *POST* oder *PUT* übermittelt werden. Analog zur *@Produces*-Annotation, die definiert, welche Formate eine Methode erzeugen kann, gibt es auch ein *@Consumes*, mit dem definiert wird, welche Datentypen eine Methode im *POST*- oder *PUT*-Request akzeptiert. Als Beispiel wird eine Ressource gewählt, die nicht nur ein einzelnes Objekt, sondern eine ganze Liste kapselt: eine Liste von Kunden, die unter dem URI */consumers* erreichbar ist (Listing 3). Nach gängigem REST-Muster akzeptiert diese Ressource neue Unterelemente (in diesem Fall: *Kunden*) via *POST* und liefert bei *GET* eine Darstellung ihres Inhalts zurück. Für die Darstellung der Liste von Kunden ebenso wie für einen einzelnen Kunden wurde XML gewählt und jeweils ein eigener MIME-Typ zugeordnet: *application/vnd.innoq.customers+xml* bzw. *application/vnd.innoq.customer*s*+xml*. MIME-Typen wie *application/pdf*, *text/html* usw. werden normalerweise offiziell zentral registriert – man sollte sie nicht nach Belieben neu erfinden. Manchmal lässt sich das jedoch nicht vermeiden. Dann sollten sie mit dem Präfix „vnd" als „vendor", d.h. proprietär gekennzeichnet werden.

Eine Implementierung des JAX-RS-APIs nimmt also einen HTTP-Request entgegen, findet zunächst eine geeignete Klasse und gegebenenfalls Methode für den URI, sucht dann nach einer Methode, die auf das HTTP-Verb (*GET*, *PUT*, *POST*, *DELETE*) passt und prüft schließlich, ob sowohl der per Accept-Header geforderte als auch der per Content Type angegebene MIME-Type zu den *@Consumes*- und *@Provides*-Annotationen passen. Unterschiedliche MIME-Typen können dabei auf verschiedene oder auch auf ein und dieselbe Java-Methode abgebildet werden. Bei *POST* und *PUT*, also bei Requests, die einen „Body" enthalten, wird dieser in den einzigen Parameter übertragen, der nicht mit einer Annotation ausgezeichnet ist; für *GET*, *POST* und *PUT* gilt dies entsprechend auch für den Rückgabewert. Wird ein Content Type angefordert oder versandt, den die Implementierung nicht unterstützt, sorgt das Framework für die korrekte HTTP-Antwort (415, Unsupported Media Type).

MessageBodyReader und MessageBodyWriter

Die Signatur der *getAsXML*-Methode in Listing 3 ist Zeichen eines Problems: Man produziert einen String, der einer XML-Darstellung der Kundenliste entspricht. Das ist eine klare Vermischung eigentlich unabhängiger Aspekte. Eigentlich sollte sich die Methode nicht um das Format kümmern, sondern darum, das richtige Objekt zurückzugeben. Das gilt umso mehr, wenn ein Format an mehreren Stellen verwendet wird.

JAX-RS löst dieses Problem mithilfe der Interfaces *MessageBodyReader* und *MessageBodyWriter*, zusammenfassend *EntityProvider* genannt. Klassen, die diese Interfaces implementieren, können bei der JAX-RS Runtime registriert werden und übernehmen die Konversion von einem spezifischen Datenformat in die Java-Objektwelt und zurück. Ist ein *MessageBodyWriter* wie in Listing 4 implementiert (und ein analoger *MessageBodyWriter*, aus Platzgründen nicht dargestellt) können wir unsere Methoden *getAsXml* und *newCustomer* wie folgt vereinfachen:

Java-Typ	MIME-Typ
byte[]	*/*
java.io.InputStream	*/*
java.io.Reader	*/*
java.io.File	*/*
javax.activation.DataSource	*/*
javax.xml.transform.Source	text/xml text/xml application/xml application/*+xml
javax.xml.bind.JAXBElement eigene JAXB-Klassen	text/xml text/xml application/xml application/*+xml
MultiValuedMap<String, String>	application/x-www-form-urlencoded
StreamingOutput	*/*

Tabelle 1: Standard-EntityProvider

```
@GET
public CustomerList get() {
  return Customer.findAll();
}

@POST
public Response newCustomer(Customer c) {
  Customer.add(c);
  return Response.ok().build();
}
```

Interessant am *MessageBodyReader* ist die Annotation *@Provider*. Diese Kennzeichnung ermöglicht es der Runtime, die entsprechenden Klassen automatisch zu finden, ohne dass sie in eine XML-Konfigurationsdatei eingetragen oder mit einem anderem Mechanismus explizit registriert werden müssen. Man muss jedoch in vielen Fällen überhaupt keinen eigenen Mechanismus implementieren, denn JAX-RS schreibt eine Reihe von Standard-*EntityProvider*-Klassen vor, die jede standardkonforme Implementierung mitbringen muss (Tabelle 1). Für Methoden, die einen dieser Typen zurückgeben oder als Eingabe erwarten, ist die entsprechende Abbildung damit bereits erledigt.

Listing 5: Subressourcenklasse

```
public class CustomerResource {
  @GET
  public Response get() {
    if (customer == null) return noSuchCustomer();
    return Response.ok().entity(customer).build();
  }

  @PUT
  public Response put(Customer c) {
    if (customer == null) return noSuchCustomer();
    try {
      customer.setName(c.getName());
      return Response.ok().build();

    } catch (Exception e) {
      e.printStackTrace();
      return Response.status(400)
          .entity("Please send well-formed XML\n")
          .type("text/plain").build();
    }

  }
  //...
```

Flüssige Antworten

HTTP zeichnet sich unter anderem durch eine Vielzahl von Statuscodes aus, deren Bedeutung in der Spezifikation genau beschrieben ist. Darin sind nicht nur bekannte Vertreter wie 200 (OK), 500 (Internal Server Error) oder 404 (Not Found) enthalten, sondern z.B. auch 201 (Created), der versandt wird, wenn eine Ressource erfolgreich erzeugt wurde, oder 409 (Conflict) im Falle eines Lost-Update-Problems. Gleichzeitig kann die Antwort auf einen HTTP-Request nicht nur den Statuscode und eine Payload enthalten, sondern auch diverse Header (z.B. für die Kontrolle des Cachings). JAX-RS kapselt diese vielfältigen Informationen in einem eigenen Response-Objekt, das über ein „fluid API" konstruiert wird. Ein Beispiel dafür haben wir in Listing 4 schon gesehen:

```
return Response.status(400).entity("Please send
    well-formed XML\n").type("text/plain").build();
```

Wurzel- und Subressourcen

In der REST-Dissertation [3] selbst wird nur der allgemeine Begriff „Ressource" verwendet. Dennoch erkennt man im praktischen Einsatz von RESTful HTTP das Muster von übergeordneten und untergeordneten Ressourcen. In unserem Beispiel „enthält" die Ressource *Liste aller Kunden* Verweise auf einzelne Kundenressourcen, die untergeordnet sind. Üblicherweise wird eine Unterressource durch ein *POST* der notwendigen Daten auf die übergeordnete Ressource angelegt. JSR 311 definiert dazu die Konzepte von Wurzel- und Subressourcen. Bei den Beispielen bisher handelte es sich um Root Resources – diese stellen typischerweise eigenständige oder übergeordnete Ressourcen dar. Innerhalb solcher Klassen kann eine Methode an eine Unterressource delegieren:

```
@Path("/procurement3/customers/")
public class CustomersResource {
 @Path("{id}")
 public CustomerResource customerById
                    (@PathParam("id") int id) {
   return new CustomerResource(Customer.get(id));
 }
 //...
}
```

In diesem Fall erkennt die JAX-RS Runtime am Fehlen einer Methodenannotation (*@GET*, *@POST* usw.), dass es sich bei der Methode um einen *sub resource locator* handelt und ordnet die Anfrage der Instanz einer Subressourcenklasse zu, die zurückgegeben wird (Listing 5).

Kontext

Sowohl innerhalb von Ressourcenklassen als auch in EntityProvidern ist es möglich, auf die darunterliegende Protokollebene zuzugreifen. So kann man über die Annotation *@Context* in Verbindung mit einem Parameter vom Typ *UriInfo* alle Elemente einer URI „von Hand" verarbeiten; über *@Context* in Verbindung mit dem Typ *HttpHeaders* erhält man Zugriff auf die HTTP-Header. Ähnliche Mechanismen gibt es für den kompletten Request und für Sicherheitsinformationen.

HEAD und OPTIONS

Zu den weniger bekannten, aber dennoch standardisierten HTTP-Methoden gehören *HEAD* und *OPTIONS*. *HEAD* ähnelt stark einem *GET*, allerdings ohne die Übertragung der eigentlichen Ressourcenrepräsentation. Ein Client kann diese Methode benutzen, um zunächst zu prüfen, ob er ein *GET* wirklich durchführen will (z.B. abhängig von der Größe der zu übertragenden Datenmenge). *OPTIONS* bildet eine Art Reflection-Mechanismus ab: Ein Client kann sich damit über die für diese spezifische Ressource erlaubten HTTP-Methoden informieren. Beide Methoden sind automatisch verfügbar, wenn man seine Ressourcen mit dem JAX-RS-API implementiert: Die

Runtime ruft bei einem *HEAD* einfach die passende *GET*-Methode auf und verwirft den Body der Nachricht. Das Ergebnis von *OPTIONS* wird auf Basis der Metainformationen berechnet, die für die Verteilung der Anfragen ohnehin ermittelt werden. Ist dies nicht gewünscht (z.B. aus Effizienzgründen), kann man auch eigene Implementierungen erstellen und die Methoden entsprechend mit *@HEAD* oder *@OPTIONS* annotieren.

Hypermedia

Das wichtigste aller REST-Prinzipien ist Hypermedia, die Möglichkeit zur Dynamisierung eines Anwendungsablaufs durch Links. Dieser Punkt wird in JAX-RS durch die Klasse *UriBuilder* unterstützt. Damit lässt sich ein URI Template mit einer Menge von Werten „verheiraten" und so einen URI produzieren. Das Template muss dabei nicht explizit angegeben werden, denn damit wäre man gezwungen, die URI-Abbildung an mehr als einer Stelle zu definieren und auch synchron zu halten. Stattdessen kann man sich einen *UriBuilder* auf Basis einer Ressourcenklasse erzeugen lassen:

```
UriBuilder.fromResource(CustomersResource.class)
    .path(CustomersResource.class, "customerById")
    .build(4711)
```

Mit *fromResource* erhält man den Pfad, der per Annotation für die übergebene Klasse definiert ist; bei *path* kann die Klasse und ein Methodenname übergeben werden; *build* schließlich erhält die Werte, die in das Ergebnis-URI-Template eingesetzt werden sollen. Ergebnis dieses recht komplexen Ausdrucks wäre folgender relativer URI, der sich nun in eine Repräsentation einbetten lässt:

```
/procurement3/customers/4711
```

Dieser Aspekt ist zwar besonders wichtig, denn in einer „echten" REST-Anwendung führt der Weg des Clients von einer Ressource zur nächsten immer über Verknüpfungen. Mit einiger Berechtigung kann man aber auch behaupten, dass es sich hier um den am wenigsten elegant gelösten Aspekt von JAX-RS handelt.

Container und Implementierungen

Eine JAX-RS-Anwendung kann in einer Standard-Java-SE-Umgebung, einem beliebigen Servlet-Container, einer JAX-WS Runtime oder anderen, in der Spezifikation nicht näher definierten Umgebungen in Betrieb genommen werden. Wie das geschieht, ist von der eingesetzten Implementierung abhängig. Für erste Experimente mit JAX-RS eignet sich die Referenzimplementierung Jersey [4] von Sun. Mit dieser können sowohl eigenständige Applikationen (mithilfe des in Java 6 mitgelieferten HTTP-Servers) gestartet als auch aufwändigere Servlet- bzw. HTTP-Container eingesetzt werden. Neben Jersey gibt es bereits eine Reihe weiterer Implementierungen des JSR 311, z.B. RESTEasy aus dem JBoss-Umfeld oder die auf der REST-Bibliothek Restlet aufsetzende Implementierung. Zu Jersey gehört auch eine Reihe von Beispielapplikationen, z.B. eine einfache Implementierung des Atom Publishing Protocols. Den vollständigen Code für die in diesem Artikel beschriebenen Beispiele (inklusive Maven-Build-Instruktionen für die Verwendung mit der Jetty Engine) kann ebenfalls als Startpunkt dienen.

Fazit: JAX-RS RESTful?

JAX-RS ist eine gelungene Abbildung der REST-Prinzipien auf aktuelle Java-Sprachmittel. Insbesondere die Verknüpfung von HTTP-Methoden, akzeptierten und versandten MIME-Typen und der Zugriff auf alle HTTP- und URI-Informationen ist nach kurzer Einarbeitung durchaus intuitiv. In einigen Fällen kann es allerdings recht schwierig werden, die Ursache eines Fehles zu finden: Wird die Methode, die man erwartet, nicht aufgerufen, muss man sich durch die Regeln kämpfen. Die Beschreibung des Algorithmus' zur Abbildung von HTTP-Requests auf Java-Methoden nimmt dabei in der Spec immerhin knapp drei Seiten ein. Die Unterstützung für Hypermedia ist eher dürftig, diese Meinung teilen sogar die Spec-Leads [5]. Dennoch macht die Entwicklung einer REST-Anwendung mit JAX-RS deutlich mehr Spaß als mit dem doch arg in die Jahre gekommenen Servlet-API.

Stefan Tilkov ist Geschäftsführer und Principal Consultant bei der innoQ Deutschland GmbH.

Links & Literatur

[1] JSR 311: JAX-RS: The JavaTM API for RESTful Web Services: jcp.org/en/jsr/detail?id=311

[2] www.innoq.com/resources/articles/JM-2009-01/Code

[3] Fielding, Roy Thomas: Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, 2000: www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[4] Jersey, JAX-RS (JSR 311) Reference Implementation: https://jersey.dev.java.net

[5] JSR 311 Final, InfoQ: www.infoq.com/news/2008/09/jsr311-approved

[4] www.netbeans.org/products/platform