6 REST-Services mit JAX-RS und Jersey

Heutzutage ist es weit verbreitet, dass Applikationen als verteilte Systeme realisiert werden und verschiedene Systembestandteile auf unterschiedlichen Rechnern oder virtuellen Maschinen (VMs) laufen. Zur Bereitstellung von Funktionalität über das Netzwerk kann man sogenannte Webservices einsetzen. Anhand des Wortes erkennt man, dass es sich dabei um Dienste handelt, die auf das Web ausgerichtet sind. Im Gegensatz zu den klassischen, in einem Browser laufenden Webapplikationen sind Webservices weniger für menschliche Benutzer gedacht, sondern bieten Funktionalitäten in Form spezieller APIs (wie Twitter, Google usw.) an, die von anderen Programmen zugreifbar sind. Zur Implementierung verteilter Services nutzt man in der Java-EE-Welt oftmals Session Beans. Was unterscheidet Webservices von diesen? Zunächst die Möglichkeiten zum Deployment, aber vor allem die Art der Kommunikation: Zwischen Java-Applikationsservern ermöglichen es Java-spezifische Protokolle zum entfernten Methodenaufruf, gewöhnlich RMI (Remote Method Invocation), Funktionalitäten von Objekten in Form von Methoden aus einer anderen JVM aufzurufen. Für Webservices kommen beim Nachrichtenaustausch standardisierte, textbasierte und dadurch technologieunabhängige Formate und Protokolle wie XML oder JSON über HTTP zum Einsatz (Grundlagen zu HTTP finden Sie in Anhang B.

In diesem Kapitel wollen wir auf REST-Services schauen, die eine Interoperabilität zwischen Systemen erlauben, sodass sogar Unterschiede im eingesetzten Betriebssystem sowie der verwendeten Programmiersprache keine Rolle spielen. Dadurch können Java-Programme auch Funktionalität von C#-Systemen nutzen, oder in JavaScript realisierte Frontends auf in Java implementierte Services zugreifen.

Die Webservice-Varianten SOAP und REST

Für die Kommunikation in verteilten Systemen haben sich in den letzten Jahren zunehmend Webservices durchgesetzt. Hier unterscheidet man zwischen SOAP und REST. SOAP steht für Simple Object Access Protocol und beschreibt einen Standard, um auf Basis von XML entfernte Funktionalität (analog zu Methodenaufrufen) ausführen zu können. REST-Webservices folgen dagegen dem Architekturstil REST (Representional State Transfer), den Roy Fielding im Jahr 2000 in seiner Dissertation[1] beschrieben hat. Die grundsätzliche Idee besteht darin, alle Funktionalität in Form von adressierbaren Ressourcen bereitzustellen – und eben nicht über Methoden.

[1] https://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm

6.1 REST im Kurzüberblick

REST nutzt HTTP für eine zustandslose Client-Server-Kommunikation, d. h., ein Client sendet Anfragen (***Requests***) an einen Server, der diese bearbeitet und dann Antworten (***Responses***) zurücksendet. Man unterscheidet zwischen einem REST-Server, der Ressourcen bereitstellt, und REST-Clients, die auf diese Ressourcen zugreifen. Dazu besitzt jede Ressource eine ID, die in der Regel als URI (Uniform Resource Identifier) modelliert ist.

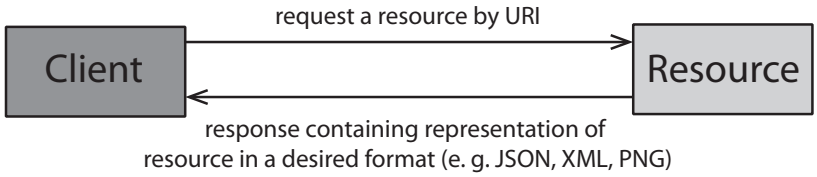

Abbildung 6-1 *Prinzipieller Ablauf bei der Client-Server-Kommunikation und REST*

Zur Adressierung eines ***REST-Service*** (auch ***Ressource*** genannt) dient eine URI, die aus Server und Port sowie einem Basispfad und einem Pfad der Ressource besteht:

```
http://server:port/basePath/resourcePath/
```

Damit wird der darunter registrierte REST-Service angesprochen, der die gewünschte Aktion ausführt. Es werden z. B. neue Datensätze angelegt oder Informationen zu bestehenden Datensätzen abgefragt. Die Kommunikation basiert auf HTTP und stützt sich vor allem auf die vier Operationen `POST`, `GET`, `PUT` und `DELETE`:

- `POST` – Erzeugt neue Daten, d. h. eine neue Ressource.
- `GET` – Definiert einen Lesezugriff auf eine oder mehrere Ressourcen.
- `PUT` – Verändert eine existierende Ressource. Falls diese noch nicht existiert, kann eine Ressource auch neu erzeugt werden.
- `DELETE` – Löscht eine Ressource.

Anforderungen an die Befehle

Damit ein REST-Service korrekt funktionieren kann, müssen die obigen Befehle einige Konventionen einhalten. Falls dies nicht geschieht, kann es unerwartet zu Fehlern kommen. Daher sind folgende Dinge relevant:

- Die `GET`-Methode muss seiteneffektfrei sein. Sie darf serverseitig nur lesen, aber nichts am Zustand der Ressourcen ändern – Logging ist erlaubt. Deshalb lassen sich `GET`-Aufrufe gefahrlos cachen, wodurch Webserver performant arbeiten können.
- Die Methoden `GET`, `PUT` und `DELETE` müssen im Server so implementiert werden, dass sie ***idempotent*** sind. Das bedeutet Folgendes: Der Client muss sie gefahrlos mehrfach mit denselben Parametern ausführen dürfen.

Idempotenz erleichtert die Fehlertoleranz. Wenn sich ein Client wegen einer Störung nicht sicher ist, ob ein gewünschtes Kommando beim Server angekommen ist und dort ausgeführt wurde, kann es gefahrlos nochmals gesendet werden. Lediglich `POST` ist nicht idempotent, da es eine Änderung der Daten vornimmt. `PUT` und `DELETE` sind idempotent, weil sie Daten nur für eine spezifische Ressource ändern. Wenn sie mehrmals aufgerufen werden, ändern sich die Werte nicht mehr bzw. wird keine weitere Ressource gelöscht.

Tabelle 6-1 *Gebräuchliche REST-Kommandos*

HTTP-Befehl	URL-Pfad	Beschreibung
`POST`	`/customers`	Erzeugt einen neuen Kunden mithilfe der Informationen des Bodys.
`GET`	`/customers`	Ermittelt alle Kunden.
`GET`	`/customers/<id>`	Ermittelt den Kunden mit der im Pfad übergebenen `id`.
`PUT`	`/customers/<id>`	Aktualisiert den Datensatz des Kunden mit der übergebenen `id` mithilfe der Informationen des Bodys.
`DELETE`	`/customers/<id>`	Löscht den Kunden mit der Id `id`.

Formate von Antworten

Die vom Server erzeugten Antworten können gewöhnlich verschiedene Repräsentationen besitzen, etwa XML, JSON oder Plain Text – je nach spezifischen Anforderungen von Clients werden Ressourcen dynamisch in unterschiedlichen Formaten zurückgeliefert. Durch Angaben im HTTP-Header kann man das oder die gewünschte(n) Format(e) festlegen, z. B. durch `Accept: application/xml` eine XML-Repräsentation anfordern. Die automatisch ablaufende ***Content Negotiation*** sorgt dafür, dass zwischen Client und Server ein bestmöglich passendes Format ausgehandelt wird.

Nehmen wir an, das später entwickelte Programm StandaloneRESTServer wäre gestartet. Ein `GET` an `http://localhost:7777/rest/greeting/Mike` liefert je nach gewünschtem Format als Antwort etwa die XML-Repräsentation eines Grußes:

```
<greeting>
   <salutation>Hello</salutation>
   <name>Mike</name>
</greeting>
```

Würde JSON präferiert, so sähe eine mögliche Antwort wie folgt aus:

```
{ greeting : [{ "salutation" : "Hello" },{ "name" : "Mike"}] }
```

Eigenschaften von RESTful Webservices

Bevor wir REST-Services in Aktion erleben, möchte ich folgende wichtige Eigenschaften in Erinnerung rufen:

- Adressierung von Ressourcen – Ein Nutzer (Client) kann Ressourcen über URIs direkt adressieren.
- Einheitliches Interface – REST-Services bieten oftmals ein an den CRUD-Operationen (Create, Read, Update und Delete) ausgerichtetes Interface, basierend auf den HTTP-Methoden `POST`, `GET`, `PUT` und `DELETE`.
- Client-Server-Kommunikation – Die Kommunikation bei REST läuft gemäß dem Client-Server-Modell ab und nutzt HTTP zum Datenaustausch.
- Zustandslos – Um Zustandslosigkeit zu erzielen, muss der Client immer alle für die Bearbeitung eines Requests vom Server benötigten Daten mitsenden. Es sollte keinen auf dem Server gehaltenen Zustand für eine Kommunikation geben. Bei einer verteilten Kommunikation erleichtern Zustandslosigkeit und Idempotenz die Skalierung der Lösung.
- Netzwerktransparenz – Verschiedene Netzwerkkomponenten wie Proxyserver, Gateways usw. sollten für die Kommunikation nicht von Relevanz sein.
- Vielfältige Formate – Die von einem Client angesprochenen Ressourcen können Daten in verschiedenen Repräsentationsformen, häufig XML oder JSON, aber auch Plain Text oder gar als PDF oder Bild (PNG, JPEG, ...), liefern.

6.1.1 Einführendes Beispiel eines REST-Service

Wenn man REST-Services mit Java aufrufen oder bereitstellen möchte, dann gibt es dafür den Standard JAX-RS (Java API for RESTful Web Services). Dazu benötigt man eine Implementierung von JAX-RS. Gebräuchlich ist die Bibliothek Jersey, die die Referenzimplementierung darstellt. In Applikationsservern kommen auch andere Implementierungen wie etwa RESTEasy zum Einsatz. Unabhängig von der konkreten Implementierung erlaubt es JAX-RS, REST-Services in einem Applikationsserver oder Servlet-Container bereitzustellen. Nur etwas Zusatzaufwand ist für die Bereitstellung von REST-Services in einem normalen Java-Programm nötig. Bevor wir das vertiefen, ergänzen wir zunächst unseren Build, schauen uns ein einführendes Beispiel an. Doch zuvor werfen wir einen Blick auf zentrale Annotations aus JAX-RS.

Vorbereitungsarbeiten JAX-RS und Jersey

JAX-RS definiert Klassen, Interfaces und Annotations. Damit diese verfügbar sind, müssen wir in unserem Gradle-Build einige Abhängigkeiten ergänzen. Neben JAX-RS als Spezifikation binden wir die auf verschiedene JARs aufgeteilte Bibliothek Jersey als konkrete Implementierung ein:

```
// JAX-RS
compile 'javax.ws.rs:javax.ws.rs-api:2.0'
// REST-Client
compile 'org.glassfish.jersey.core:jersey-client:2.22.1'
// REST-Server
compile 'org.glassfish.jersey.core:jersey-server:2.22.1'
// HTTP-Server
compile 'org.glassfish.jersey.containers:jersey-container-jdk-http:2.22.1'
// JAXB-Support
compile 'org.glassfish.jersey.media:jersey-media-jaxb:2.22.1'
// JSON-Support
compile 'org.glassfish.jersey.media:jersey-media-json-jackson:2.22.1'
```

Hier zeige ich bereits alle Abhängigkeiten, die zwar noch nicht alle sofort, aber später im Verlaufe des Kapitels zur Realisierung der Funktionalitäten benötigt werden.

Definition einer einfachen Ressource

Mithilfe der im JAX-RS definierten Annotations können wir eine Java-Klasse in eine von einem REST-Server bereitgestellte Ressource (auch REST-Service genannt) umwandeln. Schauen wir uns das am Beispiel der Klasse `HelloWorldResource` an:

```
@Path("/greeting")
public class HelloWorldResource
{
    @GET
    @Path("/{name}")
    @Produces(MediaType.TEXT_PLAIN)
    public Response getMsg(@PathParam("name") final String name)
    {
        final String output = "Hello " + name + "\n";
        return Response.status(Response.Status.OK).entity(output).build();
    }
}
```

Die Annotation `@Path` legt fest, unter welcher Adresse die Ressource relativ zum Pfad des REST-Servers zugreifbar ist, hier `greeting`. Die Zuordnung von Methoden zu HTTP-Kommandos geschieht ebenfalls mithilfe von auf Annotations: Durch die Angabe von `@GET` registriert sich die Methode `getMsg()` – deren Name im Kontext von JAX-RS keine Rolle spielt, da er nach außen nicht sichtbar ist – als Handler für HTTP `GET`. Für Methoden kann man weitere Pfadangaben annotieren, um Varianten von Kommandos anbieten oder wie hier Parameter im Pfad angeben zu können. Deren Auswertung geschieht automatisch, wenn man die Annotation `@PathParam` nutzt. Dabei müssen der in `@Path` in geschweiften Klammern notierte Parametername mit demjenigen in der Annotation `@PathParam` übereinstimmen: Im Listing bekommen wir als Parameter einen Namen übergeben und generieren daraus eine einfache textuelle Nachricht als Antwort. Den Typ des Inhalts der Antwort legen wir mithilfe der Annotation `@Produces` und der Angabe eines MIME-Types, hier durch `javax.ws.rs.core.MediaType.TEXT_PLAIN` beschrieben, fest.

Die Aufbereitung der Antwort nutzt die Klasse `Response`, die gemäß dem Entwurfsmuster BUILDER realisiert ist.[2] Über `status()` legen wir den HTTP-Statuscode fest. Der Nutzinhalt des Bodys wird über einen Aufruf von `entity()` definiert. Schließlich wird die Antwort durch einen Aufruf von `build()` erzeugt.

Hinweis: Eingabetypen und Rückgabewerte

In JAX-RS werden als Eingabetypen primitive Datentypen, `List<T>`, `Set<T>` und Typen, die die statische Methode `valueOf()` anbieten, unterstützt. Als Ergebnis wird entweder ein Wert oder aber in komplexeren Fällen ein Datensatz oder eine Menge davon zurückgeliefert. JAX-RS unterstützt die Konvertierung von POJOs in eine entsprechende XML- bzw. JSON-Repräsentation. Dazu kommt das bereits in Abschnitt 1.3 vorgestellte JAXB zum Einsatz.

Zentrale JAX-RS-Annotations im Überblick

Die gebräuchlichsten JAX-RS-Annotations aus dem Package `javax.ws.rs` sind in Tabelle 6-2 aufgeführt.

Tabelle 6-2 *Zentrale JAX-RS-Annotations*

Annotation	Beschreibung
`@Path`	Legt den Pfad fest, unter dem die Ressource ansprechbar ist.
`@POST`	Die annotierte Methode reagiert auf ein HTTP `POST`.
`@GET`	Die annotierte Methode bearbeitet einen HTTP `GET` Request.
`@PUT`	Die annotierte Methode reagiert auf HTTP `PUT`.
`@DELETE`	Die annotierte Methode behandelt HTTP `DELETE` .
`@Produces`	Bestimmt, welche Rückgabeformate von der annotierten Methode produziert werden können.
`@Consumes`	Legt für Eingabeparameter fest, in welchem Format diese von der annotierten Methode entgegengenommen werden können.
`@PathParam`	Beschreibt Parameter, die im Pfad der URL notiert werden.
`@QueryParam`	Beschreibt Parameter, die im Query-Teil der URL angegeben sind, also nach dem `?` als Name-Wert-Paar `name = value` und durch `&` voneinander getrennt.
`@FormParam`	Beschreibt Parameter, die über HTML-Formulare eingegeben werden.

[2] Entwurfsmuster beschreibe ich detailliert in meinem Buch »Der Weg zum Java-Profi« [8].

Für die beiden Annotations `@Produces` und `@Consumes` wird das Format der Daten durch MIME-Types und den Enum `javax.ws.rs.core.MediaType` spezifiziert: Dabei stehen `TEXT_PLAIN` (`"text/plain"`) für Plain Text, `APPLICATION_XML` (`"application/xml"`) für XML und `APPLICATION_JSON` (`"application/json"`) für JSON. Neben den in `MediaType` spezifizierten MIME-Types gibt es noch viele weitere, die dann textuell zu spezifizieren sind.

Bereitstellen des Webservice

Nachdem wir einen REST-Service implementiert haben, sollte dieser auch im Netzwerk zugänglich gemacht werden. Wenn Sie sich im Applikationsserverumfeld befinden, dann wird der REST-Service aufgrund der Annotations automatisch deployt und unter der im `@Path` angegebenen Adresse bereitgestellt. Weil wir unser Beispiel ohne Applikationsserver leichtgewichtiger halten wollen, muss die Bereitstellung selbst implementiert werden. Das ist problemlos möglich: Seit Java 6 ist ein HTTP-Server durch die Klasse `com.sun.net.httpserver.HTTPServer` ins JDK integriert, der für Jersey eine Erweiterung bietet, die JAX-RS-Ressourcen deployt. Einen solchen HTTP-Server erzeugt man durch Aufruf der statischen Methode `JdkHttpServerFactory.createHttpServer()`. Damit kann man Stand-alone-Applikationen wie folgt erstellen:

```
import org.glassfish.jersey.jdkhttp.JdkHttpServerFactory;
import org.glassfish.jersey.server.ResourceConfig;

public class StandAloneRESTServer
{
    private static final String BASE_URI = "http://localhost:7777/rest/";
    private static final String BASE_PACKAGE = "rest.jersey.server.basics";

    public static void main(final String[] args)
                  throws URISyntaxException, IOException
    {
        final ResourceConfig rc = new ResourceConfig().packages(BASE_PACKAGE);
        final HttpServer server = JdkHttpServerFactory.
                                  createHttpServer(URI.create(BASE_URI), rc);

        // ACHTUNG: Der Aufruf von server.start() geschieht als Seiteneffekt
        System.out.println("REST-Server is running at " + BASE_URI);
        System.in.read();
        server.stop(0);
    }
}
```

Listing 6.1 *Ausführbar als* **'STANDALONERESTSERVER'**

Führen wir das Programm STANDALONERESTSERVER aus, so wird der Server nach der Konstruktion durch Aufruf von `createHttpServer()` automatisch auch gestartet und horcht auf Port 7777 auf Verbindungen. Alle im angegebenen Package `rest.jersey.server.basics` gefundenen REST-Services sind als Folge unter dem Basispfad `http://localhost:7777/rest/` zugreifbar.

6.1.2 Zugriffe auf REST-Services

Nachdem wir nun REST-Funktionalität im Netz veröffentlicht haben, wollen wir natürlich auch sehen, wie wir darauf zugreifen können. Dazu gibt es verschiedene Varianten, auf die ich im Anschluss ein wenig genauer eingehen werde.

Zugriffe mit CURL

Wir beginnen mit dem Kommandozeilentool CURL, um HTTP-Abfragen auszuführen.[3] Probieren wir es einfach mal aus, beispielsweise wie folgt:

```
curl http://localhost:7777/rest/greeting/Mike
```

Sofern der zuvor erstellte REST-Server läuft, liefert dieser Aufruf Folgendes:

```
Hello Mike
```

Zugriffe mit dem REST-Client-API

JAX-RS und Jersey bieten ein REST-Client-API. Mit dessen Hilfe kann man Zugriffe auf eine REST-Ressource auf einer höheren Abstraktionsebene formulieren. Dabei spielen folgende Klassen, Interfaces und Methoden aus Jersey eine zentrale Rolle:

- `Client` – Ein solcher REST-Client wird von einem `ClientBuilder` erzeugt.
- `WebTarget` – Um einen REST-Service anzusprechen, benötigt man eine Instanz eines `WebTarget`. Eine solche erhält man von einem `Client` durch Aufruf einer der mehrfach überladenen `target()`-Methoden. Eine Ressource wird über ihren Pfad durch Aufruf der Methode `path()` angesprochen. Um nun einen Request abzusetzen, verwenden wir die Methode `request()`. Das gewünschte HTTP-Kommando wird dann über eine korrespondierende Methode, wie z. B. `get()` oder `post()`, ausgeführt und die Antwort in Form einer `Response` zurückgegeben.
- `Response` – Repräsentiert die Antwort einer HTTP-Anfrage. Eine `Response` bietet Zugriff auf den Statuscode (`getStatus()`) und die Daten (`readEntity()`) sowie darauf, ob es diese überhaupt gibt (`hasEntity()`). Dabei übergibt man der Methode `readEntity()` den gewünschten Typ von Antwortdaten. Die im Body vorliegenden Daten werden automatisch durch das JAX-RS-Framework (genauer durch `javax.ws.rs.ext.MessageBodyReader`) in den Typ konvertiert, sofern dies möglich ist. Wie schon zuvor im Praxishinweis »Eingabetypen und Rückgabewerte« erwähnt, ist dies für primitive Typen sowie `List<T>` und `Set<T>` der Fall.[4]

[3] Es ist ohne weitere Installation unter Mac OS X und Linux verfügbar. Für Windows steht es unter `http://curl.haxx.se/` frei zum Download bereit.

[4] Zudem können eigene `MessageBodyReader` zum Lesen spezieller Typen registriert werden. Zum Schreiben dienen korrespondierende `MessageBodyWriter`.

Mit diesem Wissen können wir die Abfrage mithilfe einer Methode `performGet()` wie folgt implementieren:

```
public static void main(final String[] args) throws Exception
{
    final String basePath = "http://localhost:7777/rest/";
    final String resourcePath = "greeting/Mike";

    final Client client = ClientBuilder.newClient();

    final String result = performGet(client, basePath, resourcePath);
    System.out.println("Response Content: " + result);
}

private static String performGet(final Client client,
                                 final String basePath,
                                 final String resourcePath)
{
    final WebTarget webTarget = client.target(basePath).path(resourcePath);
    System.out.println("\nSending 'GET' request to URL '" + basePath +
                       resourcePath + "'");

    final Response response = webTarget.request().get();
    final int responseCode = response.getStatus();
    System.out.println("Response Code: " + responseCode);

    if (responseCode != Response.Status.OK.getStatusCode())
    {
        throw new RuntimeException("HTTP error code: " + responseCode);
    }

    if (response.hasEntity())
    {
        return response.readEntity(String.class);
    }
    throw new IllegalStateException("response expected to contain data");
}
```

Listing 6.2 *Ausführbar als* **'HELLOCLIENTEXAMPLE'**

Erwähnenswert an der ansonsten recht eingängigen Implementierung dieses Clients ist, das Verhalten für den Fall, dass in der Response keine Daten vorhanden sind. Eine leere Response zeugt von einem Programmierfehler im Server und mündet hier deshalb in einer Exception.

Startet man das Programm HELLOCLIENTEXAMPLE, so erhält man folgende Konsolenausgaben:

```
Sending 'GET' request to URL 'http://localhost:7777/rest/greeting/Mike'
Response Code: 200
Response Content: Hello Mike
```

6.1.3 Unterstützung verschiedener Formate

Wie eingangs erwähnt, unterstützen Webservices in der Regel verschiedene Formate für Eingabeparameter und Rückgaben. Wir betrachten zunächst den Fall, dass wir die Aufbereitung der unterstützten Antwortformate selbst implementieren. Dieses Vorgehen ist bereits für simple Beispiele etwas schwerfällig und führt zu einigen Wiederholungen oder zumindest recht ähnlichen Abschnitten im Sourcecode. Dies akzeptieren wir hier, weil das Beispiel nur der Demonstration dient und wir im Verlauf noch eine elegantere Variante durch den Einsatz von JAXB kennenlernen werden.

Ergänzungen im REST-Service

Um neben XML auch JSON als Format des Grußes zu unterstützen, fügen wir der Ressourcenklasse einige Methoden hinzu. Abhängig von dem in `@Produces` angegebenen Format bereiten wir innerhalb der Methode selbst die Daten adäquat auf. Wir sehen später, wie man das Ganze deutlich besser lösen kann.

```
@Path("/greeting")
public class HelloWorldResource
{
    @GET
    @Path("/{name}")
    @Produces(MediaType.TEXT_PLAIN)
    public Response getGreetingAsText(@PathParam("name") final String name)
    {
        final String output = "Hello " + name;

        return Response.status(Response.Status.OK).entity(output).build();
    }

    @GET
    @Path("/{name}")
    @Produces(MediaType.APPLICATION_XML)
    public Response getGreetingAsXML(@PathParam("name") final String name)
    {
        final String output = "<greeting>\n" +
                              "   <salutation>Hello</salutation>\n" +
                              "   <name>" + name + "</name>\n" +
                              "</greeting>";

        return Response.status(Response.Status.OK).entity(output).build();
    }

    @GET
    @Path("/{name}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getGreetingAsJson(@PathParam("name") final String name)
    {
        final String output = "{ greeting : [{ \"salutation\" : \"Hello\" },{ " +
                                             "{ \"name\" : \"" + name + "\"}] }";

        return Response.status(Response.Status.OK).entity(output).build();
    }
}
```

Abfragen mit dem Client-API

Schauen wir uns nun an, wie wir bei Abfragen mithilfe eines Aufrufs von `accept()` den gewünschten Formattyp für die Antwort vorgeben:

```
private static final String REST_URI  = "http://localhost:7777/rest/";
private static final String HELLO_MIKE_RESOUCRE_PATH = "greeting/Mike";

public static void main(final String[] args)
{
    final Client client = ClientBuilder.newClient();
    final WebTarget target = client.target(REST_URI);
    final WebTarget helloService = target.path(HELLO_MIKE_RESOUCRE_PATH);

    System.out.println("As Response:  " + getOutput(helloService));
    System.out.println("As XML:       " + getOutputAsXML(helloService));
    System.out.println("As JSON:      " + getOutputAsJSON(helloService));
    System.out.println("As Text:      " + getOutputAsText(helloService));
}

private static Response getOutput(final WebTarget service)
{
    return service.request().accept(MediaType.APPLICATION_XML).get();
}

private static String getOutputAsXML(final WebTarget service)
{
    return getOutputOfType(service, MediaType.APPLICATION_XML);
}

private static String getOutputAsJSON(final WebTarget service)
{
    return getOutputOfType(service, MediaType.APPLICATION_JSON);
}

private static String getOutputOfType(final WebTarget service,
                                      final String type)
{
    return service.request().accept(type).get(String.class);
}

private static String getOutputAsText(WebTarget service)
{
    return service.request().accept(MediaType.TEXT_PLAIN).get(String.class);
}
```

Listing 6.3 *Ausführbar als* **'HELLORESTCLIENTWITHFORMATS'**

Führen wir das Programm HELLORESTCLIENTWITHFORMATS aus, so erhalten wir folgende Ausgaben, die die Abfrage der unterschiedlichen Formate demonstrieren:

```
As Response:  InboundJaxrsResponse{context=ClientResponse{method=GET, uri=http:
      //localhost:7777/rest/greeting/Mike, status=200, reason=OK}}
As XML:       <greeting>
   <salutation>Hello</salutation>
   <name>Mike</name>
</greeting>
As JSON:      { greeting : [{ salutation : Hello },{ name : Mike}] }
As Text:      Hello Mike
```

6.1.4 Zugriffe auf REST-Services am Beispiel von MongoDB

Als Abschluss der Einführung in REST möchte ich den in MongoDB integrierten REST-Server einsetzen, um Ihnen verschiedene Zugriffsvarianten zu zeigen.

Viele Applikationen (Twitter, Google usw.) bieten zum externen Zugriff ein REST API. Das gilt ebenso für MongoDB, wenn beim Start des MongoDB-Servers die Option `--rest` angegeben wird. Der zugehörige REST-Service ist auf Port 28017 erreichbar.

Von MongoDB angebotene REST-Ressourcen

Das von MongoDB angebotene REST API bietet nicht allzu viel Funktionalität, sondern vor allem lesenden Zugriff an. Hier folgen einige Beispiele:

- Anzeige aller Datenbanken –
  `http://127.0.0.1:28017/listDatabases`
- Abfragen der Daten einer Collection –
  `http://127.0.0.1:28017/<dbName>/<collectionName>/`
- Abfragen mit Filterung –
  `http://127.0.0.1:28017/<dbName>/<collectionName>/`
  `?filter_<attributeName>=<value>`

Wir schauen uns nachfolgend ein paar Varianten an, wie wir den von MongoDB bereitgestellten REST-Service aufrufen können.

REST-Aufruf per Browser

Probieren wir doch den einen oder anderen der obigen Befehle einmal aus. Das können wir mit dem bereits kennengelernten CURL machen. Mitunter etwas bequemer als die Kommandozeile ist der Einsatz unseres Browsers zur Interaktion mit dem MongoDB-REST-Service (vgl. Abbildung 6-2).

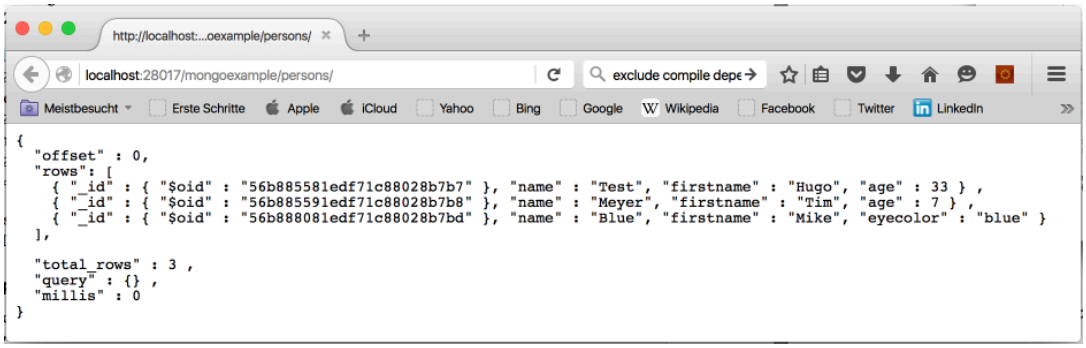

Abbildung 6-2 *REST-Abfrage per Browser an MongoDB*

REST-Aufruf per Postman

Die Aufrufe per CURL bzw. Browser sind für einfaches Ausprobieren und einen schnellen Test durchaus akzeptabel. Wenn Sie Ihre REST-Schnittstelle aber komfortabel testen wollen, sind oft eine Historie der Aufrufe (`GET`, `POST`, `DELETE` usw.) und der transportierten Daten für Sie wichtig. Dafür sollten Sie einen Blick auf das Tool Postman werfen (vgl. Abbildung 6-3). Ursprünglich als Erweiterung zum Chrome-Browser entstanden, ist es nun auch separat als Chrome-App verfügbar. Weitere Details finden Sie unter `https://www.getpostman.com/`.

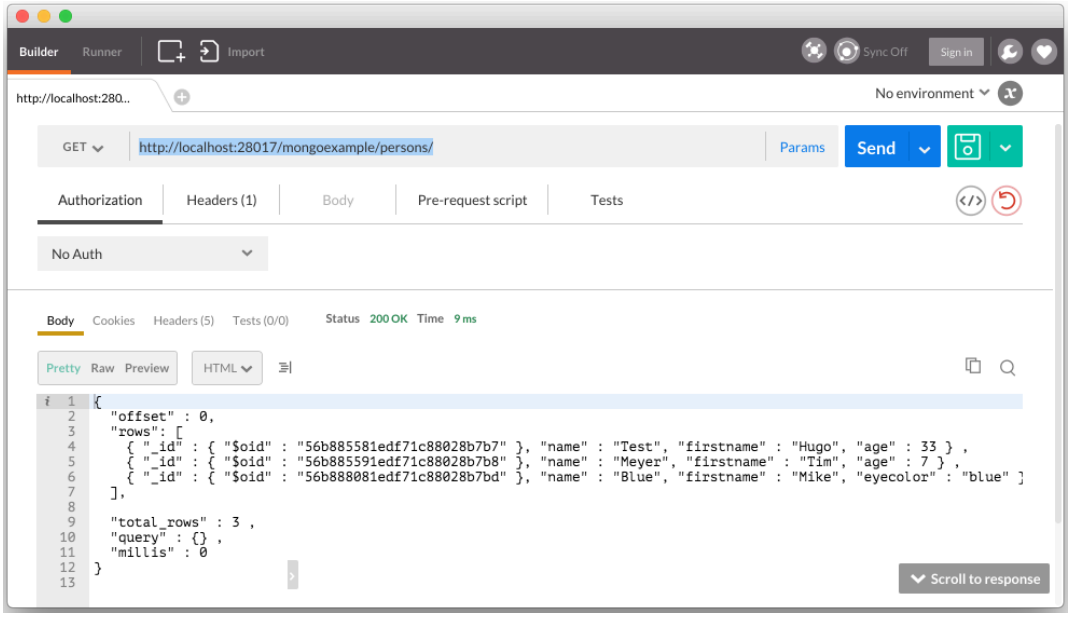

Abbildung 6-3 *Postman und der Aufruf an MongoDB*

Fazit

Wir haben nun schon einige Varianten eingesetzt, um auf REST-Services zuzugreifen. Das manuelle Ausführen ist für einfache Abfragen sicher in Ordnung. Auch für das Kennenlernen von und das Experimentieren mit einem REST-Service kann das nützlich sein – insbesondere kann man über das Tool Postman erste, manuelle Tests der Schnittstelle vornehmen. Zur Automatisierung von Tests und für die Kommunikation zwischen verschiedenen Systemen benötigt man aber eine programmatische Umsetzung mit einem problemangepassteren API. Dafür hatten wir bereits das REST-Client-API kennengelernt. Dieses sollte auch bevorzugt werden, weil es, wie eingangs schon erwähnt, bei Webservices eigentlich nicht um Interaktivität geht, sondern um eine Kommunikation zwischen Programmen.

Bevor wir uns im Anschluss mit weiteren Details zum Implementieren von REST-Services beschäftigen, möchte ich im nachfolgenden Praxishinweis eine Abgrenzung von REST-Services zu anderen Formen der verteilten Kommunikation vornehmen.