JAX-RS

Friedrich Kiltz

November 2019

Inhaltsverzeichnis

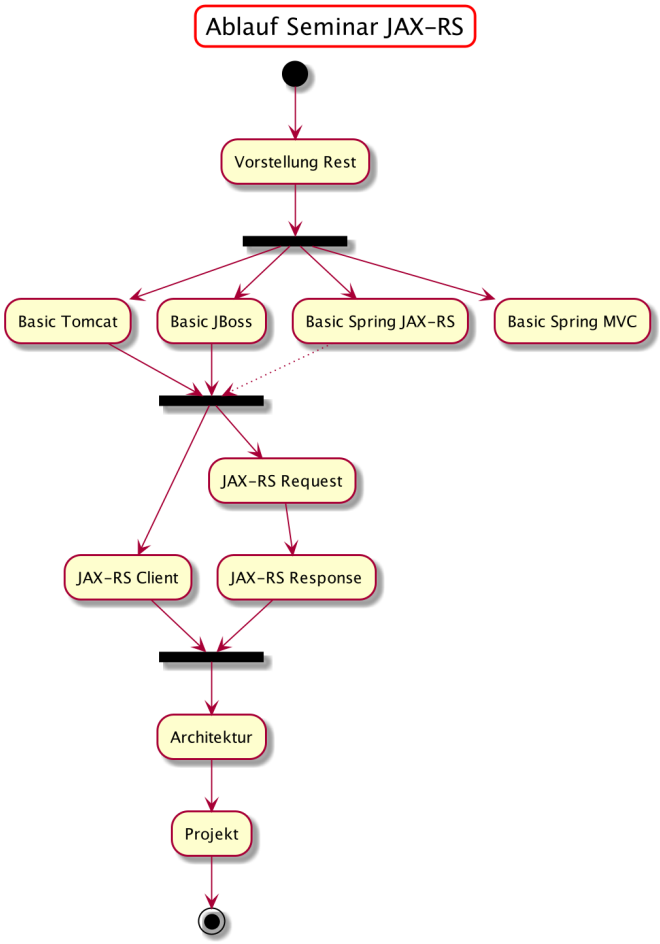
Ablauf des Seminars . . . 1
1. Allgemeiner Ansatz von REST . . . 2
  1.1. Herkunft . . . 2
  1.2. Beschreibung von REST . . . 3
2. JAX-RS mit Jersey . . . 5
  2.1. Nutzung von Jersey im Tomcat . . . 6
  2.2. Unterschied bei Deployment im JBoss . . . 10
  2.3. Application und @ApplicationPath . . . 11
  2.4. JAX-RS und Spring . . . 12
3. Request, Response . . . 15
  3.1. Routing zu einer Java-Methode . . . 15
  3.2. JAX-RS Injection . . . 17
  3.3. Content Handler . . . 18
  3.4. Responses . . . 18
4. Client-API . . . 21
  4.1. JAX-RS Client . . . 21
  4.2. Security . . . 21
  4.3. Weitere REST-Clients . . . 23
5. REST API Design . . . 24
  5.1. Architekturmodelle (ROA, WOA, SOA) . . . 24
  5.2. URLs . . . 24
  5.3. Sortierung, Filterung und Felder-Limitierung . . . 25
  5.4. Paging . . . 26
  5.5. Rückgabe mit HTTP-Responsecodes . . . 26
  5.6. Konventionen zum Benennen von Java-Klassen . . . 27
  5.7. HATEOAS . . . 27
  5.8. REST Dokumentation über WADL . . . 28
  5.9. REST Dokumentation über Swagger . . . 29
A. Anhang . . . 31
  A1. Links & Quellen . . . 31
  A2. Stichwortverzeichnis . . . 32
  A3. Abkürzungsverzeichnis . . . 33

Ablauf des Seminars

Nach einer kurzen Vorstellung was REST ist und was es tut schauen wir uns mehrere Varianten an um einen REST-Service in unterschiedlichen Umgebungen zu erstellen.

Dann suchen wir uns eine der Umgebungen (Tomcat, JBoss, Spring JAX-RS) aus und sehen uns die verschiedenen Möglichkeiten der Parameter und Rückgabewerte an. Parallel dazu bilden wir diese Features mit der Client-API ab.

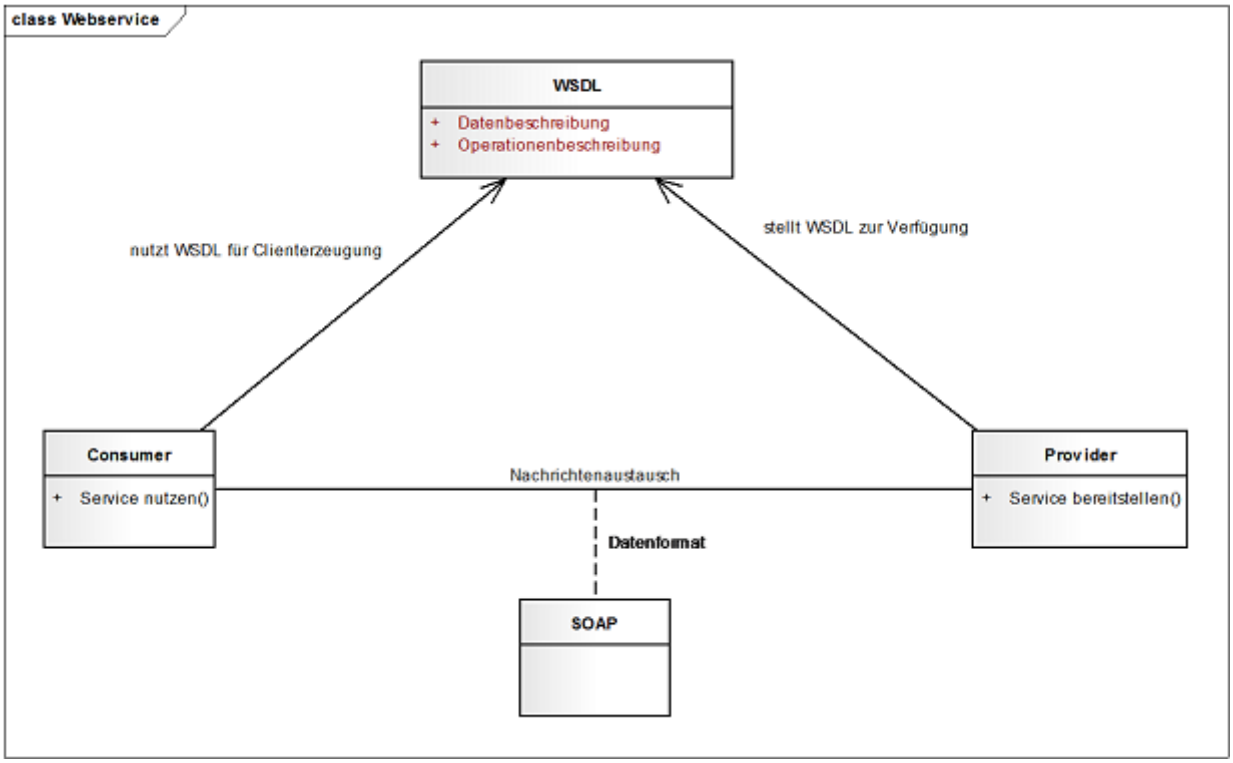
Dann wenden wir uns der Architektur zu und erstellen eine API zu einem kleinen ausbaufähigen Projekt.



1. Allgemeiner Ansatz von REST

1.1. Herkunft

WebServices im engeren Sinne nutzen eine WSDL zur Beschreibung der Services mit ihren Datentypen und SOAP-Nachrichten zum Transport der eigentlichen Aufrufe. Dabei können die unterschiedlichsten Protokolle (http(s), ftp, SMTP etc.) genutzt werden.



Die Vorteile dieses Ansatzes stecken in der sehr aussagekräftigen WSDL die eine starke Entkopplung von Provider und Consumer ermöglicht. Außerdem haben wir durch die SOAP-Nachricht die Möglichkeit im Header zusätzliche Informationen (Security, Adressing etc.) mit zu übertragen. Die unterschiedlichen Protokolle ermöglichen uns auf die Natur der Daten einzugehen.

Roy Fielding stelle diese Basis für eine Reihe von Anwendungsfällen in Frage. Da doch viele Webservices nur mit HTTP(S) arbeiten und Consumer und Provider sich kennen und austauschen können und es wahrlich performantere Möglichkeiten als XML gibt um Daten auszutauschen könnte man die WebServices vereinfachen.

In seiner Dissertation im Jahre 2000 veröffentlichte Fielding den REST-Architekturstil, wobei REST für Representational State Transfer steht.

> Die Bezeichnung „Representational State Transfer“ soll den Übergang vom aktuellen Zustand zum nächsten Zustand (state) einer Applikation verbildlichen. Dieser Zustandsübergang erfolgt durch den Transfer der Daten, die den nächsten Zustand repräsentieren.

(aus: https://de.wikipedia.org/wiki/Representational_State_Transfer)

1.2. Beschreibung von REST

REST lehnt sich an das Prinzip des WWW an und unterstützt nur das Kommunikationsmuster Request/Response. Ein wichtiges Prinzip ist die eindeutige Adressierbarkeit jeder Resource durch einen URI. Hierbei kann der Request per GET oder POST erfolgen und entweder per Query-String oder POST einfache Datentypen, HTML-, XML- oder Binärdaten übertragen. Der Response gibt nur die Nutzdaten zurück. Je nach Anforderung durch den Client kann die Rückgabe in einer unterschiedlichen Repräsentation geschehen, z.B. in HTML, XML, JPG etc. Neben den Zugriffsmethoden POST und GET können die beiden anderen HTTP-Methoden PUT und DELETE für die Änderung oder Löschung von Resourcen benutzt werden.

Im Vergleich zu den Basisfunktionalitäten der Persistenz CRUD (Create, Read, Update und Delete) verhalten sich die HTTP-Methoden folgendermaßen:

GET	Abrufen von Informationen, wie z.B. die Liste der Artikel, einen Kunden, den Status einer Lieferung. Die Daten können eventuell gecached werden (analog zum Read aus CRUD). `GET /kunden/k123` GET-Aufrufe sind nur lesend und idempotent.

POST	Erzeugung einer neuer Resource, wie z.B. der Position einer Bestellung, Bemerkungen zu einem Kunden (analog zum Create aus CRUD).
	
	`POST /bestellungen/0815`
	`<position nr="1" artnr="123" menge="5"/>`
	
	POST-Aufrufe sind **nicht** idempotent.

PUT	Änderung einer Resource mit einer bestimmten ID, wie z.B. den Lieferanten l321 (analog zum Update aus CRUD).
	
	`PUT /lieferanten/l321`
	`<name>Chateau Certan de May</name> <land>Frankreich</land>`
	
	PUT-Aufrufe sind idempotent.

DELETE	Löschen einer Resource, wie z.B. Löschen des Kunden mit der Kd-Nr.: k123 (analog zum Delete aus CRUD). `DELETE /kunden/k123` DELETE-Aufrufe sind idempotent.

Die HTTP-Methoden TRACE, OPTIONS und HEAD werden in der Praxis kaum genutzt.

REST ist genau wie HTTP zustandslos, d.h., jeder Zugriff steht für sich allein. Sessions sollten nur auf der Clientseite verwaltet werden. Dies hat den Nachteil, dass die Daten im Request größer werden und bereits übertragene Daten erneut übertragen werden müssen, da immer alle relevanten Daten übertragen werden müssen. Auf die Daten vorheriger Requests kann nicht zugegriffen werden. Die Zustandslosigkeit bringt die Eigenschaften Sichtbarkeit, Ausfallsicherheit und Skalierbarkeit mit sich. Die Sichtbarkeit ergibt sich daraus, dass ein einzelner Request alle

relevanten Daten enthält und keine Annahmen über vorherige Requests getroffen werden müssen. Die Ausfallsicherheit wird verbessert, da bei einem partiellen Ausfall des Systems nur der letzte Request wiederholt werden muss. Die Skalierbarkeit wird verbessert, weil keine Daten auf dem Server gehalten werden und somit kein Session-Sharing benötigt wird.

2. JAX-RS mit Jersey

In der JSR 311 wird die JAX-RS (Java API for RESTful Webservices)-Spezifikation definiert. JAX-RS hat folgende Ziele:

- Die Grundlage der Entwicklung sind POJO, die per Annotation als Webresource zur Verfügung gestellt werden. Die Spezifikation legt auch den Lebenszyklus und den Sichtsbarkeitsbereich (Scope) der Resource fest.
- JAX-RS basiert auf HTTP. Eine Unabhängigkeit des Protokolls wird nicht angestrebt.
- JAX-RS strebt eine Unabhängigkeit des Formats an. Diese Formate werden per MIME-Type im Header angegeben und definieren so den erwarteten Content-Type.
- JAX-RS ist unabhängig von einem Container. Die Spezifikation unterstützt das Deployment in einem Servlet-Container und in einer JAX-WS-Umgebung. JAX-RS verwendet folgende Terminologie:

Resource class	Eine Java-Klasse, die per Annotations eine Webresource implementiert.
Resource method	Eine Methode einer Resourcenklasse, die einen spezifischen Request verarbeitet.
Provider	Die Implementation eines JAX-RS-Interfaces. Referenz-Implementation: Jersey

Die JSR-311 definiert folgende Annotations:

Annotation	Element	Beschreibung
@Consumes	Klasse oder Methode	Liste der Mediatypen, die konsumiert werden können. `@Consumes("application/x-www-form-urlencoded")`
@Produces	Klasse oder Methode	Liste der Mediatypen, die erzeugt werden können. `@Produces("text/plain")`
@GET @POST @PUT @DELETE @HEAD	Methode	Spezifiziert, dass diese Methode einen entsprechenden Request behandelt.
@Path	Klasse oder Methode	Spezifiziert den relativen Pfad zu dieser Resource. Angabe aus der Klasse und den Methoden werden zusammengesetzt. `@Path("info")`
@PathParam	Parameter, Feld oder Methode	Spezifiziert, dass ein Teil des Pfades als Parameter übergeben wird, z.B. http://beispiel.org/Lieferanten/l321. wird l321 als Lieferantennummer bei `getLieferant(@PathParam("nr") String liefNr)`
@QueryParam	Parameter, Feld oder Methode	Spezifiziert, dass ein bestimmter Query-Parameter zu einer Variablen gemapped wird, z.B. http://beispiel.org/Lieferanten/s=Zypern zu der Methode `getLieferanten(@QueryParam("s") String such)`

Annotation	Element	Beschreibung
@FormParam	Parameter, Feld oder Methode	Wie `@QueryParam` nur für Formular-Elemente. Sollte nur bei Methoden-Parametern genutzt werden. `neu(@FormParam("name") String name, @FormParam("nr") String nr)`

@MatrixParam	Parameter, Feld oder Methode	Wie `@QueryParam` nur für Matrix-Parameter.
		`@MatrixParam("info")`
		`@DefaultValue("Life")`
		`@Encoded`
		`private String info;`

@CookieParam	Parameter, Feld oder Methode	Spezifiziert, dass ein Methoden-Parameter durch einen Cookie- Parameter gefüllt wird.
@HeaderParam	Parameter, Feld oder Methode	Überträgt einen Header-Parameter an einen Methoden-Parameter.
@Encoded	Klasse, Konstruktor, Parameter, Feld oder Methode	Die Parameter werden normalerweise decoded. Sollte dies nicht geschehen, kann diese Standardeinstellung mit `@Encoded` ausgeschaltet werden.
@DefaultValue	Parameter, Feld oder Methode	In Kombination mit den Annotations `@QueryParam`, `@MatrixParam`, `@CookieParam`, `@FormParam` und `@HeaderParam` kann diese Annotation den Vorgabewert spezifizieren.
@Context	Parameter, Feld oder Methode	Definiert ein Ziel für eine Dependency Injection, wie im vorherigen Abschnitt beschrieben wurde. `getUriInfo(@Context UriInfo info)`
@Provider	Klasse	Annotation für eine Klasse, die eine JAX-RS-Extension-Schnittstelle implementiert.

Die Rückgabe des Ergebnisses erfolgt meist in einem Response-Objekt. Mit dieser Methode kann man auch die Response Codes (200 für ok, 404 für Not Found etc.) zurück geben. Eine nette Übersicht für die Codes finden Sie unter http://www.webmaster-eye.de/Status-Codes-beim-HTTP-Response.149.artikel.html das maßgebliche Dokument befindet sich unter http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html .

2.1. Nutzung von Jersey im Tomcat

Jersey ist die Default-Implementation von JAX-RS. Der folgende Anwendungsfall beschreibt die notwendigen Schritte zur Nutzung von Jersey (Version 2.28) mit dem Tomcat (Version 8.5) in Eclipse.

Kurzbeschreibung

Dieser Anwendungsfall erstellt eine Webapplikation mit einer einfachen Resource zur Überprüfung der Kommunikation. Die Webapplikation wird im Tomcat deployed.

Ausgangssituation

Tomcat ist installiert.

Ziel

Ein einfacher REST-Service ist deployed und kann per URI getestet werden.

Ablauf

Schritt 1: Installation von Jersey

Unter der Adresse http://repo1.maven.org/maven2/org/glassfish/jersey/bundles/jaxrs-ri/2.28/jaxrs-ri-2.28.zip kann man das komplette Archiv für Jersey herunterladen. Entpacken Sie das Archiv an einen Ort Ihrer Wahl.

Schritt 2: Erzeugen des Projekts

Erzeugen Sie ein neues Projekt für eine Webapplikation (Dynamic Web Project) in Eclipse. Nutzen Sie als ContextPath `/rs`. Stellen Sie Ihrer Webapplikation die Bibliotheken aus api, ext und lib zur Verfügung. Kopieren Sie die Bibliotheken dafür in das lib-Verzeichnis unter WEB-INF. Nehmen Sie die Bibliotheken in den Build-Path auf (erfolgt automatisch).

Schritt 3: Frontcontroller definieren

Binden Sie in der web.xml den ServletContainer von Jersey (bis Jersey 1.16 `com.sun.jersey.spi.container.servlet.ServletContainer` in der Version 2.28 die Klasse `org.glassfish.jersey.servlet.ServletContainer`) als Servlet ein, sorgen Sie dafür, dass er beim Start der Applikation gleich geladen wird (`<load-on-startup> 1</load-on-startup>`) und mappen Sie ein passendes URL-Pattern zu dem Servlet (in meinem Beispiel nehme ich `/api/*`) .

Schritt 4: Erzeugen der Resource-Class

Erzeugen Sie im Paket `rest.basic` die Resource-Class `KommunikationsRestService` und annotieren Sie die Klasse mit der @Path- Annotation (`@Path("/basic")`).

Schritt 5: Erzeugen der Resource-Method

Erzeugen Sie die Resource-Method `public String ping(String text)` und annotieren Sie diese mit @GET, @Produces("text/plain") und einer @Path("ping")- Annotation. Der Parameter kann noch mit @QueryParam("text") annotiert werden. Treten Sie den Beweis des Aufrufs an, indem Sie den übergebenen String in Großbuchstaben umgewandelt zurückgeben.

Schritt 6: Deployment und Test

Deployen Sie die Webapplikation auf Ihrem Webserver. Testen Sie den Service im Browser durch Eingabe des URL

```
http://localhost:8080/<Context>/<URLMapping>/<Resource-Class>/<Resource-
Method>?text=Test
```

Dabei ist `<context>` der Kontext der Webapplikation (`rs`), `<URL-Mapping>` das Mapping aus der web.xml (`api`) zum ServletContainer-Servlet, `<Resource-Klasse>` der Inhalt der Path- Annotation der

Resource-Class (`basic`) und <Resource-Method> der Inhalt der Path- Annotation der Resource-Method (`ping`). Mit ?text=Test kann noch ein Parameter übergeben werden. Der Browser sollte nun das erwartete Ergebnis zeigen: TEST.

In unserem Beispiel also:

```
http://localhost:8080/rs/api/basic/ping?text=Test
```

Alternativen

Jersey kann auch per Maven installiert werden. Der URL für Jersey und Maven ist http://download.java.net/maven/2/com/sun/jersey/ . Natürlich kann die Ping-Methode auch andere Ausgaben (Repräsentationen) erstellen. Schauen Sie sich die Alternativen mit

- @Produces("text/html")
- @Produces("application/json")
- @Produces("application/xml")

an.

Der Service kann auch per curl getestet werden, was den Vorteil hat, dass man den Medientyp besser angeben kann:

```
curl -i http://localhost:8080/rs/api/basic/ping?text=Test –H "ACCEPT:text/plain"
```

Tipp: Erzeugen Sie für die Rückgabe ein Objekt der Klasse `javax.ws.rs.core.Response` und nutzen Sie für die Medientypen die Klasse `javax.ws.rs.core.MediaType`.

```
@GET
@Path("ping")
@Produces( MediaType.TEXT_PLAIN)
public Response pingPost(@QueryParam("text") String text){
    return Response.ok(text.toUpperCase(), MediaType.TEXT_PLAIN).build();
}
```

Dokumente

Im Deployment Descriptor web.xml wird der ServletContainer eingebunden und mit einem Mapping verbunden:

```
<servlet>
    <display-name>JAX-RS REST Servlet</display-name>
    <servlet-name>REST-Servlet</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <init-param>
        <param-name>jersey.config.server.provider.packages</param-name>
        <param-value>de.kiltz.rest.basic</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>REST-Servlet</servlet-name>
    <url-pattern>/api/*</url-pattern>
</servlet-mapping>
```

Die Resource-Class enthält hier nur die eine Resource-Method mit den entsprechenden Annotations:

```
package rest.basic;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
@Path("basic") // 1
public class KommunikationsRestService {
    @GET // 2
    @Path("ping") //3
    @Produces("text/plain") 4
    public String ping(@QueryParam("text") //5
    String text) {
        return text.toUpperCase();
    }
}
```

Die Resource-Class ist ein normales POJO. Die Path-Annotation bei der Klasse (1) ist das Präfix für jeden weiteren Pfad, der bei den Methoden definiert ist. Die GET-Annotation (2) legt die HTTP-Methode fest. Mit der Path-Annotation (3) wird der Pfad der Klasse erweitert. Die Produces-Annotation (4) bestimmt den Rückgabetyp, der vom Empfänger per Accept erlaubt sein muss. Mit QueryParam kann man die Namen der Query-Parameter mit den Parametern der Methode verknüpfen (5).

Ein Gradle Build-Script für diese Konstellation könnte folgendermaßen ausschauen:

```
apply plugin: 'java'
apply plugin: 'idea'
apply plugin: 'eclipse'
apply plugin: 'war'

sourceCompatibility = 1.8

repositories {
    mavenCentral()
}

dependencies {
    providedCompile group: 'javax.servlet', name: 'servlet-api', version: '2.4'
    providedCompile group: 'javax.servlet', name: 'jsp-api', version: '2.0'

    compile fileTree(dir: 'libs/jaxrs-ri/api', include: ['*.jar'])
    compile fileTree(dir: 'libs/jaxrs-ri/lib', include: ['*.jar'])
    compile fileTree(dir: 'libs/jaxrs-ri/ext', include: ['*.jar'])

    testCompile group: 'junit', name: 'junit', version: '4.12'
}

task deploy (type:Copy, dependsOn: build) {

    from('build/libs'){
        rename 'jax-rs-server.war', 'rs.war'
    }
    into "${project.property('tomcat.home')}/webapps/"
    println "kopiere rs.war nach ${project.property('tomcat.home')}/webapps/"
}
```

Dazu könnte man in einer `gradle.properties` noch `tomcat.home` spezifizieren.

2.2. Unterschied bei Deployment im JBoss

Der wichtigste Unterschied ist, dass JBoss per Default RESTEasy statt Jersey als Implementierung von JAX-RS nutzt. Das WAR-File, dass wir dem JBoss zur Verfügung stellen benötigt also keine weiteren Bibliotheken im `WEB-INF/lib` Verzeichnis.

Das wirkt sich beim Projektaufabau dadurch aus, dass wir nur noch die JAX-RS-API benötigen und die auch nur Provided (also für unsere Entwicklungsumgebung, wird nicht im WAR-File verpackt).

In Gradle wird dann also aus

```
    compile fileTree(dir: 'libs/jaxrs-ri/api', include: ['*.jar'])
    compile fileTree(dir: 'libs/jaxrs-ri/lib', include: ['*.jar'])
    compile fileTree(dir: 'libs/jaxrs-ri/ext', include: ['*.jar'])
```

folgendes:

```
    providedCompile fileTree(dir: 'libs/jaxrs-ri/api', include: ['*.jar'])
```

Einen weiteren Unterschied haben wir in dem Deployment-Descriptor `web.xml`. Hier muss der `HttpServletDispatcher` von ReasEasy eingebunden werden:

```
    <context-param>
        <param-name>resteasy.scan</param-name>
        <param-value>true</param-value>
    </context-param>
    <context-param>
        <param-name>resteasy.servlet.mapping.prefix</param-name>
        <param-value>/api</param-value>
    </context-param>

    <servlet>
        <servlet-name>resteasy</servlet-name>
        <servlet-class>
org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>resteasy</servlet-name>
        <url-pattern>/api/*</url-pattern>
    </servlet-mapping>
```

RestEasy unterscheidet sich von Jersey insbesondere in den Möglichkeiten des Cachings, GZip Kompression und der Test-Plugins. Eine schöne Übersicht dazu findet man unter https://www.genuitec.com/jersey-resteasy-comparison/

Sollte die REST-Applikation nicht als WAR-File sondern als JAR in einem EJB-Container deployed werden muss man die Resource-Class zu Stateless Session Beans machen.

2.3. Application und @ApplicationPath

Alternativ zum Einbinden in der web.xml oder im JEE-Umfeld kann die Klasse javax.ws.rs.core.Application genutzt werden.

```
package rest.basic;
import javax.ws.rs.core.Application;
import javax.ws.rs.ApplicationPath;

@ApplicationPath("/api")
public class RestApplication extends Application {
    @Override
    public Set<Object> getSingletons() {

       HashSet<Object> set = new HashSet();
       set.add(new KommunikationsRestService());
       return set;
   }
}
```

2.4. JAX-RS und Spring

Default: Spring-MVC

Spring kommt von Haus aus mit dem Spring-MVC, mit dem auch REST realisiert werden kann. Die zentrale Annotation hierfür ist `org.springframework.web.bind.annotation.RequestMapping` mit der die wichtigsten Informationen für unsere REST-Services gesetzt werden können.

```
@RestController
@RequestMapping(path = "rs/api/basic")
public class KommunikationsRestService {

    @RequestMapping(method = RequestMethod.GET, path = "ping",
        produces = "text/plain")
    public String pingPlain(@RequestParam("s") String txt) {
        txt = txt == null ? "NULL" : txt;
        return txt.toUpperCase();
    }
}
```

Die Annotation `RestController` ist dabei gar nicht so aufregend. Sie sorgt nur dafür, dass alle Rückgabewerte der REST-Methoden nicht als View-Template sondern als RequestBody angesehen werden.

Spring-MVC und REST mit Spring-MVC ist ein eigenes Thema, das in diesem Rahmen nicht weiter verfolgt wird. Für weitere Informationen empfehle ich die ausführliche und gut gemachte Spring-Dokumentation z.B. https://docs.spring.io/spring/docs/5.2.0.RELEASE/spring-framework-reference/web.html#spring-web

Alternative: Spring mit JAX-RS

Eine Spring-Boot-Applikation kann mit der zusätlichen Abhängigkeit `spring-boot-starter-jersey`

JAX-RS-fähig gemacht werden.

Durch diese Abhängigkeit bekommen wir den Zugriff auf die Klasse org.glassfish.jersey.server.ResourceConfig bei der wir unsere JAX-RS Resource-Classes registrieren können.

```
@Component
public class JerseyConfig extends ResourceConfig {

    public JerseyConfig() {
        register(KommunikationsRestService.class);
    }
}
```

Die Resource-Class wird dann noch mit `@Component` springifiziert und unterscheident sich ansonsten kaum von der Resource-Class in unserem Tomcat-Beispiel.

Da es natürlich ein wenig fehleranfällig ist jede Resource in der JerseyConfig zu registrieren kann man mit einem einfachen Qualifier die RestServices markieren und automatisch zur Registrierung bereit stellen lassen.

Dazu können wir eine kleine Annotation z.B. mit dem Namen `RestService` erzeugen:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.PARAMETER})
@Qualifier
@Component
public @interface RestService {
}
```

und uns in der Config einfach alle RestService mit dieser Annotation injizieren lassen:

```
@Component
public class JerseyConfig extends ResourceConfig {

    public JerseyConfig(@RestService List<Object> restServices) {
        restServices.forEach(e -> {
            register(e.getClass());
        });

    }
}
```

Da wir Component schon in der Annotation eingebunden haben brauchen wir unsere Resource-Class nur noch mit `@RestService` zu annotieren:

```java
@Path("rs/api/basic")
@RestService
public class KommunikationsRestService {

    @GET
    @Path("ping")
    @Produces("text/plain")
    public String pingPlain(@QueryParam("s") String txt) {
        txt = txt == null ? "NULL" : txt;
        return txt.toUpperCase();
    }
}
```

3. Request, Response

Dieses Kapitel befasst sich mit den Mechanismen, mit denen wir Informationen aus dem HTTP-Request in Java verarbeiten können und wie wir aus Java heraus unseren Response gestalten können.

3.1. Routing zu einer Java-Methode

Um einen spezifischen Request von einer bestimmten Java-Methode behandeln zu können benötigen wir hier ein Routing, das hauptsächlich durch drei Komponenten vorgenommen wird:

- Die `Path`-Annotation
- Die Request-Methode
- Der `Content-Type`

Die Path-Annotation

Wie wir in dem Jersey-Kapitel schon kennen gelernt haben, setzt sich unsere Request-URI aus mehreren Teilen zusammen. Nach den Information für Server, Port, Context und URI-Mapping spezifizieren wir mit dem Path die Resource Class und evtl. weiter die Methode.

Normalerweise ist der Value der Path-Annotation ein einfacher String

`@Path("/kunden)`

Der durch Path-Variablen erweitert werden kann

`@Path("images/{image}")`

Auf diese Variable kann dann mit der `PathParam`-Annotation (s.u.) zugegriffen werden.

Path-Variablen können auch mit Requlären Ausdrücken erweitert werden:

```
@Path("/kunden")
public class KundenRestService {

   @GET
   @Path("{id : \\d+}")
   public Kunde getKundePerId(@PathParam("id") int id) {
      //...
   }
}
```

Diese ID darf nur numerische Zeichen enthalten. Es kann vorkommen, dass wir mehrdeutige Path-Ausdrücke definieren:

```
@Path("/kunden")
public class KundenRestService {

   @GET
   @Path("{id : .+}")
   public Kunde getKundePerId(@PathParam("id") String id) {
      //...
   }

   @GET
   @Path("{id : .+}/adressen")
   public List<Adresse> getAddressenEinesKunden(@PathParam("id") String id) {
      //...
   }

}
```

Der Aufruf `/kunden/Hägar/adressen` würde sowohl zu der ersten Definition als auch zur zweiten Definition passen. Hier entscheidet Jersey sich für den Ausdruck, der die meisten fixten Treffer hat, hier also für die Methode `getAddressenEinesKunden`.

Die Request-Methode

Die Request-Methoden haben wir im ersten Kapitel schon kennen gelernt:

GET	GET-Aufrufe sind nur lesend und idempotent.
POST	Erzeugung einer neuen Resource, **nicht** idempotent.
PUT	Änderung einer Resource, idempotent.
DELETE	Löschen einer Resource, idempotent.

Sofern eine Request-Methode nicht definiert ist, wird vom Server der Fehlercode `Method not allowed (405)` zurück gegeben. Ausgenommen hierfür ist es, wenn zu einer Path-Definition gar keine HTTP-Methoden defniert sind. Dann handelt es sich um einen Subresource Locator, der die Resource-Klasse zurück gibt, die eine weitere Verarbeitung des Requests vornehmen soll. Siehe dazu z.B. https://dennis-xlc.gitbooks.io/restful-java-with-jax-rs-2-0-2rd-edition/content/en/part1/chapter4/subresource_locators.html

Der Content-Type

Weiterhin ist für die Auswahl der richtigen Methode die Inhalte der `Accept` und `Content-Type` Header bedeutend.

```
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public RootDatenTypen getDatenPerJSON() {
        // ...
    }

    @GET
    @Produces(MediaType.APPLICATION_XML)
    public RootDatenTypen getDatenPerXML() {
        // ...
    }
```

Bei einem `Accept: application/json` würde nun die Methode `getDatenPerJSON` und bei `Accept: application/xml` die Methode `getDatenPerXML` gewählt. Ein `Accept: text/plain` würde automatisch zu einem 406 "Not Acceptable" führen.

3.2. JAX-RS Injection

Um Informationen aus dem Request in ein Java-Objekt zu übertragen bietet JAX-RS mehrere Möglichkeiten an:

@PathParam	Ein Teil des Pfades wird als Parameter betrachtet.	`@Path("{id}")` `… getPerId(@PathParam("id") int id)`
@MatrixParam	Matrix-Parameter sind den `QueryParam` ähnlich, werden aber mit einem Semikolon getrennt und gehören zu einem Path-Segment, sie sind mit Attributen vergleichbar	`/jacken;farbe=schwarz` `@MatrixParam("farbe")`
@QueryParam	QueryParam übertragen einzelne Parameter, die als Query-String bei der URL mit übergeben werden.	`GET /dinge?start=0&size=10` `get(@QueryParam("start") int start, @QueryParam("size") int size)`
@FormParam	werden mit dem Access-Header `application/x-www-form-urlencoded` aus HTML-Formularen gepostet	analog zu QueryParam, nur aus HTML-Form heraus übertragen.
@HeaderParam	Zugriff auf einen spezifischen Header-Parameter	`get(@HeaderParam("Referer") String referer)`
@CookieParam	Zugriff auf einen Cookie-Parameter der mit NewCookie gesetzt wurde.	`@CookieParam("id") int id)` oder auch `@CookieParam("id") Cookie id`
@BeanParam	Zur Verschlankung der Resource-Method-Signatur kann ein BeanParam angegeben werden, der die Request-, Form- und Header-Paramter kapselt.	
@Context	Mit der Context-Annotation kann man z.B. auf folgende Informationen zugreifen.	ServletContext, HTTPRequest, UriInfo, ResourceInfo, HTttpHeaders, Providers

Die Annotations (außer Context) können noch mit den beiden Annotation `DefaultValue` und `Encoded` kombiniert werden.

Damit eine automatische Umwandlung der Request-Informationen in die entsprechenden Parameter erfolgen kann ist eine der folgende Voraussetzungen zu erfüllen:

1. Der Parameter ist ein primitiver Datentyp
2. Die Java-Klasse hat einen "Ein-String-Konstruktor"
3. Die Klasse hat eine Methode `static <T> valueOf(String s)` (z.B. Enums)
4. Das Ziel ist eine `List<T>`, `Set<T>` oder `SortedSet<T>` wobei `<T>` Die Kriterien aus 2 oder 3 erfüllt.

3.3. Content Handler

Um den Body einer Request-Message zu lesen oder einen Body eines Responses zu erzeugen benötigen wir einen JARX-RS Content Handler.

JAX-RS bringt dazu folgende Content Handler mit:

- StreamingOutput
- InputStream, Reader
- File (Input und Output)
- MultivaluedMap (Form, Input und Output)

Hierzu gibt es genügend Beispiele im Netz wobei ich auf die MultivaluedMap besonders hinweisen möchte, da diese wieder eine elegante Möglichkeit zur Stabilisierung der Schnittstelle bei `@FormParam` zur Verfügung stellt.

Gebräuchlicher ist die Nutzung von JAXB zur Umwandlung von Java-Datenstrukturen zu einem Massage Body. Hierzu muss man sein Projekt durch einen JAXB-Handler erweitern. Eine gute Wahl hierbei ist der Jackson JAXB Provider. JAXB kann sowohl XML- als auch JSON-Formate erstellen. In den meisten Fällen ist das JSON-Format eine gute Wahl, da es performanter und kleiner ist.

Mit dem MessageBodyReader und dem MessageBodyWriter kann man auch seinen eigenen ContentProvider erzeugen.

3.4. Responses

Daten

Mit der Nutzung von JAX-RS i.V. mit JAXB können die Methoden unsere Datenobjekte direkt zurück geben:

```
@GET
@Produces({ MediaType.APPLICATION_JSON })
List<Kunde> getKunden(@QueryParam("s") String suchBegriff);

@GET
@Produces({ MediaType.APPLICATION_JSON })
@Path("{id}")
Kunde getKunde(@PathParam("id") long id);
```

Dabei ist die Klasse `Kunde` eine JAXB-Klasse, die sowohl in JSON als auch in XML zurück gegeben werden kann.

Alternativ kann man die Rückgabe auch in `Response`-Objekt verpacken:

```
@Override
public Response getKunde(long id) {
    Kunde k = service.getKunde(id);
    return Response.ok().entity(k).build();
}
```

Bei komplexeren Entitys kann man diese auch in der Klasse `GenericEntity` verpacken.

```
//...
    List<Kunde> liste = ...
    GenericEntity entity = new GenericEntity<List<Kunde>>(liste) {
    };
    return Response.ok().entity(entity).build();
```

Fehler und Exceptions

Jede Resource-Methode darf eine checked oder unchecked Exption werfen. diese wird dann automatisch in einen HTTP-Error 500 umgewandelt.

Um den Fehler genauer zu kontrollieren bietet sich die Klasse `WebApplicationException`, der man den entsprechenden Fehlerstatus mit übergeben kann:

```
    @GET
    @Path("images/{image}")
    @Produces("image/*")
    public Response getImage(@PathParam("image") String image, +
        @Context ServletContext ctx) {

        File f = new File(ctx.getRealPath("/img/")+image);
        System.out.println(f.getAbsolutePath());
        if (!f.exists()) {
            throw new WebApplicationException(404);
        }

        String mt = new MimetypesFileTypeMap().getContentType(f);
        return Response.ok(f, mt).build();
    }
```

`WebApplicationException` hat interessante Unterklassen, die die typischen Fehlersituationen abbilden. Statt der `new WebApplicationException(404);` hätte man auch eine `new NotFoundException();` werfen können.

Die Hierarchie von `WebApplicationException`:

```
Exception
   -> RuntimeException
       -> WebApplicationException
           -> RedirectionException
           -> ServerErrorException
               -> ServiceUnavailableException
               -> InternalServerErrorException
           -> ClientErrorException
               -> NotAcceptableException
               -> ForbiddenException
               -> NotAuthorizedException
               -> BadRequestException
               -> NotAllowedException
               -> NotFoundException
               -> NotSupportedException
```

4. Client-API

4.1. JAX-RS Client

JAX-RS bietet eine Klasse `Client`, die mit Hilfe eines `ClientBuilder` erstellt wird. Der Client kann wieder verwendet werden, muss aber kontrolliert am Ende geschlossen werden.

Bei der Erzeugung des Clients kann man auch gleich zusätzliche Provider wie z.B. den `JacksonJsonProvider` registrieren.

Aus dem Client kann man dann mit einer URI und möglichen Parametern ein `WebTarget` erstellen, auf das man dann den eigentlichen Request ausführen kann:

```
public class BasicTest {
    private static final String URL = "http://localhost:8081/rs/api/basic/";
    private static Client client;

    @BeforeClass
    public static void init() {
        client = ClientBuilder.newClient().register(new JacksonJsonProvider());

    }

    @AfterClass
    public static void beende() {
        client.close();
    }

    @Test
    public void testPing() {
        String query = "Test";
        String matrix = "JUnit";
        WebTarget target = client.target(URL).path("ping").matrixParam("info", matrix
).queryParam("s", query);
        String resp = target.request().accept(MediaType.TEXT_PLAIN).get(String.class);
        Assert.assertEquals(query.toUpperCase()+" "+matrix, resp);

    }
}
```

4.2. Security

Im Bereich der Sicherheit setzt JAX-RS auf die Mechanismen von HTTP und nutzt damit die Features des Servers. Diese basieren auf SSL und HTTP-Authentifizierung.

Beim Tomcat kann man die Benutzer z.B. in der `conf/tomcat-users.xml` spezifizieren.

```
<tomcat-users xmlns="http://tomcat.apache.org/xml"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://tomcat.apache.org/xml tomcat-users.xsd"
              version="1.0">

        <role rolename="admin"/>
        <user username="Ben" password="geheim" roles="admin"/>

</tomcat-users>
```

und in der `web.xml` der Applikation den Security Constraint definieren:

```
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Test SecurityContext (Ben, geheim)</web-resource-name>
            <url-pattern>/api/context/security-context</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>admin</role-name>
        </auth-constraint>
    </security-constraint>


    <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>Test SecurityContext (Ben, geheim)</realm-name>
    </login-config>

    <security-role>
        <role-name>admin</role-name>
    </security-role>
```

Alternativ dazu kann im JEE-Umfeld auch mit ´@RolesAllowed´, ´@DenyAll´, ´@PermitAll´ und ´@RunAs` gearbeitet werden.

Dann kann man bei der Erzeugung des Client ein `HttpAuthenticationFeature` registrieren:

```
client.register(HttpAuthenticationFeature.basic("Ben", "geheim"));
```

Auch die SSL-Configuration kann man gleich bei der Erzeugung des Clients mit übergeben:

```
SslConfigurator sslConfig = SslConfigurator.newInstance()
        .trustStoreFile("./truststore_client")
        .trustStorePassword("secret-password-for-truststore")
        .keyStoreFile("./keystore_client")
        .keyPassword("secret-password-for-keystore");

SSLContext sslContext = sslConfig.createSSLContext();
Client client = ClientBuilder.newBuilder().sslContext(sslContext).build();
```

Weitere Alternativen findet man sehr anschaulich unter

https://eclipse-ee4j.github.io/jersey.github.io/documentation/latest/client.html#d0e5314

beschrieben.

4.3. Weitere REST-Clients

Ein REST-Client kann eigentlich jeder sein, der einen HTTP-Request absenden und auswerten kann. z. B.:

- eine java.net.URL-Connection
- ein Apache HttpClient
- ein RESTEasy Client Proxy
- AJAX in den verschiedensten Varianten

Zum Testen eignet sich z. B. curl, Insomnia oder Postman (Links s. im Anhang).

5. REST API Design

5.1. Architekturmodelle (ROA, WOA, SOA)

- SOA Service Oriented Architecture
  - Unabhängig vom Protokoll
  - Bietet Funktionalitäten
- WOA Web Oriented Architecture
  - Subset von SOA
  - HTTP(S)-Protokoll
  - Nutzung von W3C
- ROA Resource Oriented Architecture
  - Resource: In den meisten Fällen etwas, was gespeichert werden kann (Dokument, Teile einer Datenbank, Ergebnisse von Berechnungen etc.)
  - Eine Resource kann man mit einer URI Adressieren.
  - Funktionalitäten (holen, erzeugen, ändern, löschen) werden durch die HTTP-Methoden bestimmt.

Aus diesen sich teilweise überlappenden Ansätzen ergeben sich folgende Best-Practices.

5.2. URLs

- URLs sollten im spinal-case definiert werden (alles in Kleinbuchstaben mit einem Bindestrich dazwischen)
  snake_case und CamelCase sind natürlich auch möglich, sollten aber konsequent genutzt werden.
  spinal-case wird von der RFC3986 (URI: Generic Syntax), snake_case wird von den WebGiganten (Google, Facebook, Twitter) bevorzugt.
- URLs sollten im Sinne eines Pfades zu einer Resource gestaltet werden.
  Es sollten keine Verben, sondern Nomen verwendet werden Falls Verben zum Signalisieren von Aktionen doch notwendig sind, sollten sie ans Ende der URL.
  Beispiel: `/datensicherung/execute`

Resource	GET read	POST create	PUT update	DELETE
/kunden	gibt eine Liste von Kunden zurück	Erzeugt einen neuen Kunden	Bulk update für Kunden	Löscht alle Kunden
/kunden/008	gibt einen bestimmten Kunden zurück	Method not allowed (405)	Update eines speziellen Kunden	löscht einen speziellen Kunden

- GET-Methoden sollten keine Änderungen im Datenbestand vornehmen. Hierfür sind die PUT,

POST und DELETE-Methoden zuständig.

- GET, PUT und DELETE sind idempotent, POST ist nicht idempotent.
- Für partielle Updates kann die Http-Methode `PATCH` genutzt werden.
- Mische keine Singular- und Plural-Nomen, der Einfachheit halber nutzt man durchgängig den Plural.
- Relationen werden durch Sub-Resourcen ausgedrückt:
  `GET /kunden/008/umsaetze/2019`
- spezifiziere im Header den Content-Type. `Content-Type` definiert das Request-Format, `Accept` definiert das Rückgabeformat. Sofern nichts dagegen spricht kann man durchgängig mit `MediaType.APPLICATION_JSON` arbeiten.
- Benutze eine Versionierung - von Anfang an.

```
http://beispiel.org/api/v1/kunden
```

5.3. Sortierung, Filterung und Felder-Limitierung

- Biete eine Sortierung nach mehreren Feldern und mit einem einheitlichen Parameter an. Hierzu kann man aufsteigend als Default (oder mit `+`) setzen und absteigend mit einem `-` fest legen.

```
GET /kunden?sort=nachname,-letztesKaufdatum
```

- Zum Filtern der Ergebnisse kann man auf die Eigenschaften der Resource zugreifen.

```
GET /kunden?letztesKaufdatum=last30days
```

- für häufig genutzte Filterungen kann man auch einen benutzerfreundlichen Alias anlegen:

```
GET /artikel/zum_nachbestellen
```

- Die Rückgabe kann auch nach den zurückzugebenden Feldern limitiert werden:

```
GET /kunden?fields=nachname,vorname,umsatz
```

- Bei komplexen suchen kann als Suchparameter `q=⋯` genutzt werden.
  Hierzu kann man die Suchparameter in einem POJO speichern, das man dann URLEncoded per JSON überträgt. Das POJO benötigt eine `toString()`-Methode, die ein JSON-String für die Inhalte übergibt und eine statische `fromString`-Methode die einen JSON-String in ein Objekt des POJOs umwandelt.

5.4. Paging

Wenn für die Suche ein Paging verwendet wird, sollten durchgängige Festlegungen gelten. Z. B.

- In Anlehnung an Spring Data PageRequest können die Attribute als `page` (zero-based page index) und `size` (the size of the page to be returned) bezeichnet werden.
- In der Rückgabe sollte es Informationen zu `TotalPages` und `TotalElements` geben.

5.5. Rückgabe mit HTTP-Responsecodes

In REST sollte man die komplette Vielfalt der HTTP-Responsecodes nutzen. Die wichtigsten Codes sind:

Responsecode	**Wann zurückgeben?**	**Weitere Details**
200 OK	Ausführung der Aktion war erfolgreich.	

201
Created
	Ausführung der Aktion war erfolgreich.	Wird ausschließlich bei den POST Requests nach dem Anlegen eines neuen Datensatzes zurückgegeben. Hierbei ist es üblich die URI und die ID des neue angelegten Datensatzes im zusätzlichen Location-Header zurückzugeben.
		
		`CURL –X POST \`
		`-H "Accept: application/json" \`
		`-H "Content-Type: application/json" \`
		`-d '{"state":"running","id_client":"007"}' \`
		`https://api.fakecompany.com/v1/clients/007/orders`
		`< 201 Created`
		`< Location: https://api.fakecompany.com/orders/1234`

204 No Content	Ausführung der Aktion war erfolgreich.	Es wird jedoch kein / einen leeren Response zurückgegeben (entspricht einer void-Methode in Java)

500
Internal Server Error
	Allgemeiner Serverfehler.	Fallback-Fehlermeldung, wenn nichts anderes definiert ist. Fehler-Payloads sollten im speziellen JSON-Format zurückgegeben werden. Beispiel
		
		`{`
		`"statusCode": "INTERNAL_SERVER_ERROR",`
		`"applicationCode": "32",`
		`"message": "something goes wrong…",`
		`"stackTrace": "java.lang.RuntimeException: das ist die ursache…"`
		`}`

503 Service Unavailable	Fehler, wenn der Service einen weiteren Knoten nicht erreichen konnte.	
400 Bad Request	Allgemeiner Clientfehler.	Faustregel: wenn der Client einen Fehler gemacht hat und seinen Request umformulieren muss damit dieser erfolgreich ist

Responsecode	Wann zurückgeben?	Weitere Details
401 Unauthorized	Fehler, wenn der Client nicht authentifiziert ist.	
403 Forbidden	Fehler, wenn der Client zwar authentifiziert ist, aber ihm fehlen die Rechte	

Siehe auch https://blog.mwaysolutions.com/2014/06/05/10-best-practices-for-better-restful-api/

Jersey stellt uns dafür die Enum `javax.ws.rs.core.Response.Status` zur Verfügung.

5.6. Konventionen zum Benennen von Java-Klassen

Man sollte auf eine einheitliche Benennung der Java-Klassen für REST achten. Welche Konventionen nun genau für Ihr Projekt oder Ihr Unternehmen genutzt werden steht Ihnen frei.

Ein Vorschlag dazu schaut so aus:

Benennung der Services:

- <Resource>RestService - für REST Services
- <Resource>SoapService - für SOAP Services

Es bietet sich an gegen ein Interface zu programmieren und im Interface die JAX-RS (und Swagger) Annotations zu definieren.

Java-Klassen für JSON-Requests / POJOs

- Sollten im selben Package wie der Service abgelegt sein
- Keine pauschale Präfixe oder Suffixe verwenden
- Sinnvolle Suffixe je nach Anfrage vergeben (VO, DTO, Wrapper, Impl sind NICHT sinnvoll)
- Beispiele:
  - `KundenInfo` bei LeseAnfrage
  - `KundenInfoMitHistorie` für Details zum Lesen etc.

Request-Klassen sind nur für die WebService-Schicht gedacht und dürfen im Domänen-Modell nicht verwendet werden Ebenso darf in den Signaturen der Rest-Service-Methoden keine Entität aus dem Domänen-Modell auftauchen

5.7. HATEOAS

HATEOAS (Hypermedia As The Engine Of Application State) ist eine Architektur-Prinzip, das bei einem Request die weiteren Möglichkeiten von Zustandsänderungen oder Zustandsabfragen mit übergibt. Damit können wir ausdrücken, welche Zustände von dem aktuellen Zustand aus

erreichbar sind. Dies erreichen wir, in dem wir in den Response Links einfügen:

```
<customers>
   <link rel="next"
         href="http://example.com/customers?start=2&size=2"
         type="application/xml"/>
   <customer id="123">
      <name>Bill Burke</name>
   </customer>
   <customer id="332">
      <name>Roy Fielding</name>
   </customer>
</customers>
```

(aus RESTful Java with JAX-RS 2.0)

Die Relationen sollten einheitliche und stabile Bezeichnungen haben. Einige Relationsnamen sind Quasi-Standard:

Name	Beschreibung
self	Zeigt den Link zur aktuellen Resource
first	Link zur ersten Seite beim Paging
last	Link zur letzen Seite beim Paging
prev/previous	Vorherige Seite beim Paging
next	Nächste Seite beim Paging
up	Referenziert eine übergeordnete Resource (z.B. von Adresse zu Benutzer)
item	Link zu einem Mitglied einer eingebetteten Liste (z.B. ein Kommentar aus einer Aufgabe)

Diese Links kann man auch sehr gut bei der Umsetzung einer UI des REST-Clients nutzen:

```
if (kunden._links.has("delete")) {
    // Kunde darf gelöscht werden, Lösch-Button anzeigen
}
```

5.8. REST Dokumentation über WADL

In Anlehnung an die WSDL der WebServices bietet REST eine WADL (Web Application Description Language) zur Übersicht der angebotenen Resourcen.

In der WADL werden die Resourcen mit deren Methodnen (GET, POST,...) und den Request und Response-Parametern beschrieben. Dabei werden die Datentypen nicht (wie bei der WSDL) komplett in einem Schema definiert sondern nur der Typ genannt.

In diesem Beispiel wird angezeigt, dass die Resource `daten` mit der Methode `GET` ein Element vom

Typ `datenObjekt` zürück gibt - entweder per XML oder per JSON.

```
<resource path="daten">
    <method id="getDaten" name="GET">
        <response>
            <ns2:representation xmlns:ns2="http://wadl.dev.java.net/2009/02" xmlns=""
element="datenObjekt" mediaType="application/json"/>
            <ns2:representation xmlns:ns2="http://wadl.dev.java.net/2009/02" xmlns=""
element="datenObjekt" mediaType="application/xml"/>
        </response>
    </method>
</resource>
```

Was nun genau in dem `datenObjekt` enthalten ist müssen wir schon bei dem Provider nachfragen.

5.9. REST Dokumentation über Swagger

Alternativ zur WADL kann auch eine Dokumentation per Swagger (https://swagger.io/) vorgenommen werden.

Die Konfiguration von Swagger ist teilweise etwas Umfangreich und von Umgebung zu Umgebung unterschiedlich. Zu den einzelnen Umgebungen gibt es recht ordentliche Anweisungen wie man Swagger in sein Projekt einbinden kann. Z.B Für Swagger in eine Spring-Boot-Application einzubinden ist der Artikel unter https://www.atechref.com/blog/spring-boot/using-swagger-2-with-spring-boot-spring-fox-and-jax-rs-project/ sehr hilfreich.

Swagger bietet einen Satz Annotation, mit denen man seinen REST-Service ausführlich beschreiben kann. Leider beschreibt man manche Sachverhalte damit mehrfach, wie man im folgenden Quelltext z.B. beim Produces sehen kann.

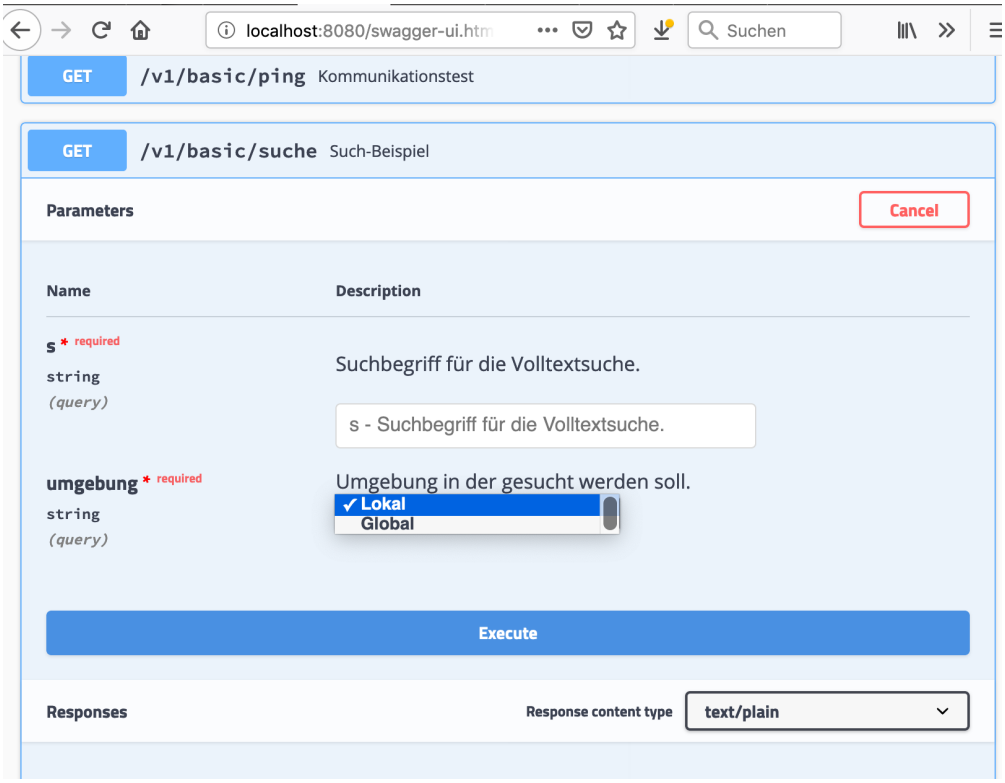
```
    @GET
    @Path("suche")
    @Produces("text/plain")
    @ApiOperation(value = "Such-Beispiel", produces = "text/plain")
    public String suche(
            @ApiParam(value = "Suchbegriff für die Volltextsuche.",
            required = true)
            @QueryParam("s") String suchBegriff,
            @ApiParam(value = "Umgebung in der gesucht werden soll.",
               required = true, allowableValues = "Lokal, Global",
               defaultValue = "Lokal")
            @QueryParam("umgebung") String umgebung) {

        return String.format("suche nach %s in der Umgebung %s",
            suchBegriff, umgebung);
    }
```

Die wichtigsten Annotations zur Beschreibung unseres Services für Swagger sind die Annotation

`io.swagger.annotations.Api` zur Ergänzung der `Path`-Annotation
`io.swagger.annotations.ApiOperation` zur Ergänzung von `GET`, `POST` etc.
`io.swagger.annotations.ApiParam` zur Beschreibung der Parameter.

Mit der Beschreibung aus obigem Quelltext kann man in der Swagger-UI sehr einfach den REST-Service aufrufen und bekommt auch noch vernünftige Hinweise welche Inhalte erwartet werden (z.B. bei der Umgebung werden in der Auswahlliste die Elemente aus `allowableValues` angezeigt.)



A. Anhang

A1. Links & Quellen

API-Design

- https://blog.mwaysolutions.com/2014/06/05/10-best-practices-for-better-restful-api/
- http://blog.octo.com/en/design-a-rest-api/
- https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api
- https://www.youtube.com/watch?v=ybwo_70jpGc Hypermedia APIs and HATEOAS von Volodymyr Tsukur

Jersey

- https://eclipse-ee4j.github.io/jersey.github.io/documentation/latest/
  Jersey User Guide

WADL & Swagger

- https://www.w3.org/Submission/wadl/
- https://swagger.io/

RestEasy

- https://resteasy.github.io RestEasy, ein JBoss-Projekt
- https://docs.jboss.org/resteasy/docs/4.3.1.Final/userguide/
- https://www.genuitec.com/jersey-resteasy-comparison/ Vergleich RestEasy und Jersey

Bücher

- https://www.oreilly.com/library/view/restful-web-services/9780596529260/
  Online-Version des Buches "RESTful Web Services" von Sam Ruby, Leonard Richardson
- Bill Burke: RESTful Java with JAX-RS 2.0
  ISBN-10: 144936134X
  Verlag: O'Reilly Media

Tools

- https://curl.haxx.se/ curl
- https://insomnia.rest Insomnia
- https://www.getpostman.com Postman

A2. Stichwortverzeichnis

C

CRUD, 3
Content-Type, 25

D

DELETE, 3, 16

F

Filtern, 25

G

GET, 3, 16
GenericEntity, 19

H

HATEOAS, 27
HTTP-Authentifizierung, 21
HTTP-Responsecodes, 26

I

idempotent, 25

J

JAXB, 18
JSON, 18

K

Kommunikationsmuster, 3

P

POST, 3, 16
PUT, 3, 16
Paging, 26

R

ROA, 24

S

SOA, 24
SSL-Configuration, 22
Sortierung, 25
Spring-MVC, 12
Subresource Locator, 16
Swagger, 29

V

Versionierung, 25

W

WADL, 28
WOA, 24
WebApplicationException, 20

A3. Abkürzungsverzeichnis

conneg
: HTTP Content Negotiation

CRUD
: Create, Read, Update und Delete

HATEOAS
: Hypermedia As The Engine Of Application State

JAXB
: Java Architecture for XML Binding

JAX-RS
: Java API for RESTful Webservices

JAX-WS
: Java API for XML Web Services

JSON
: JavaScript Object Notation, s. http://www.json.org

JSR-311
: Java Specification Request für JAX-RS

MVC
: Model View Controller

REST
: Representational State Transfer

ROA
: Resource Oriented Architecture

SOA
: Service Oriented Architecture

SOAP
: Format zum Austausch von Nachrichten bei WebServices

SSL
: Secure Sockets Layer

URI
: Uniform Resource Identifier

URL

Uniform Resource Locator

WADL

Web Application Description Language, Beschreibung von REST-Services

WOA

Web Oriented Architecture

WSDL

Web Services Description Language, Bescheibung einer WebService-Schnittstelle