RESTful WebServices in Java

Revision : 708

Chantal Taconet

September 2020
1. Introduction
1.1 REST API examples
2. REST architectural style
3. Marshalling/unmarshalling
4. Hyper Text Transfer Protocol: basics reminder
5. From a Java instance to a neutral representation
6. Java RESTful service
7. Synthethis and go further
1.1 REST API examples

- REST is a “URL friendly” way to retrieve distributed resources
- Well known examples of REST resources
  - Twitter: API
  - Google maps: API
    - where is this place http://maps.googleapis.com/maps/api/geocode/json?latlng=40.714224,-73.961452
    - URL to get an address in Evry with GPS: lat=48.625595, lon=2.443234
  - Open street map API
    - Where is this place http://nominatim.openstreetmap.org/reverse?lat=48.858518&lon=2.294524&addressdetails=1
  - State of bike stations in Lyon, API
    - https://api.jcdecaux.com/vls/v1/stations?contract=lyon&apiKey=91f170cdabb4c3227116c3e871a63e8d3ad148ee

1. Access restricted you need a google account and a key
2 REST architectural style

1. Introduction
2. REST architectural style
   2.1 REST: Representational State Transfer
   2.2 Constraint 1: Client-server architecture
   2.3 Constraint 2: Stateless
   2.4 REST: Architectural style: 6 constraints
   2.5 Uniform interface: CRUD operations
   2.6 Are these operations sufficient to build an application?
   2.7 REST resource
   2.8 URI Naming conventions
   2.9 Skiers example
   2.10 HATEOAS
       Hypermedia as the Engine of Application State
3. Marshalling/unmarshalling
4. Hyper Text Transfer Protocol: basics reminder
5. From a Java instance to a neutral representation
2.1 REST: Representational State Transfer

- **Architectural style** defined by Roy Fielding in a PhD [Fielding, 2000]
- Described by six identified constraints
- World Wide Web conforms to the REST architectural style
- Components that conform to this architectural style are called **RESTful** Web services
- **RESTful** Web services easier to implement than SOAP Web services
2.2 Constraint 1: Client-server architecture

- Allows the components (**client** and **server**) to evolve independently: the link is the API (Application Programming Interface)

- Separate the interface (API) concerns from the data storage concerns (implementation of the **resource**)

- Supported by multiple platforms and multiple languages
2.3 Constraint 2: Stateless

From Roy Fielding dissertation:

- Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server.
- Session state is therefore kept entirely on the client.

Advantage of the stateless constraint:

- **Scalability**: as each request may be handled by a different server, the number of servers may be augmented as necessary.
2.4 REST: Architectural style: 6 constraints

- **Client/server** architecture
- **Stateless**: no client context on the server
- **Cacheable**: clients can cache responses
- **Layered system**: clients and servers may be connected through intermediate layers (e.g. proxies)
- **Code on demand**: the state may include code (e.g. javascript)
- Uniform interface between clients and servers

Main advantages: **scalability, simplicity of interfaces**
2.5 Uniform interface: CRUD operations

- Requests and responses are built around the transfer of representations of resources.
- Requests are one of the four CRUD Operations:
  - Create resource $\mapsto$ POST http method
  - Read resource $\mapsto$ GET http method
  - Update resource $\mapsto$ PUT http method
  - Delete resource $\mapsto$ DELETE http method
2.6 Are these operations sufficient to build an application?

<table>
<thead>
<tr>
<th>Resource</th>
<th>Create</th>
<th>Read</th>
<th>Update</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection</td>
<td>POST</td>
<td>GET</td>
<td>PUT</td>
<td>DELETE</td>
</tr>
<tr>
<td></td>
<td>Create entry</td>
<td>List entries</td>
<td>Replace collection</td>
<td>Delete collection</td>
</tr>
<tr>
<td>Element</td>
<td>/</td>
<td>Get element</td>
<td>Replace element</td>
<td>Delete element</td>
</tr>
</tbody>
</table>
2.7 REST resource

- Any (Web) resource
- Identified by a global identifier (e.g. URI [Uniform Resource Identification])
- State of a resource may be transferred through a representation of this resource
## 2.8 URI Naming conventions

- Collection of resources: e.g., /skiers
- Single resource: e.g., /skiers/{skierid}
  - `{skierid}` is a **parameter path**
- Subcollection: e.g., /skiers/{skierid}/achievements
- Single resource: e.g., /skiers/{skierid}/achievements/{achievementId}
- Controller: e.g., /skiers/{skierid}/change-name/{new-name}
- Find: /skiers?age=41
  - age is a **query parameter**

😊 When resources are named well: an API is intuitive and easy to use.

😢 If done poorly, that same API can feel difficult to use and understand.
2.9 Skiers example

```
<table>
<thead>
<tr>
<th>«Interface»</th>
</tr>
</thead>
<tbody>
<tr>
<td>SkiersInterface</td>
</tr>
<tr>
<td>+ getAllSkiers() : Skiers</td>
</tr>
<tr>
<td>+ searchByAge() : Skiers</td>
</tr>
<tr>
<td>+ searchByName() : Skiers</td>
</tr>
<tr>
<td>+ addVictory(name:String, victory:String)</td>
</tr>
<tr>
<td>+ addSkier(skier:Skier): Skiers</td>
</tr>
<tr>
<td>+ removeByAge(age:int)</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>«REST class»</th>
</tr>
</thead>
<tbody>
<tr>
<td>SkiersREST</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>- age : int</td>
</tr>
<tr>
<td>- name : String</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Skier</th>
</tr>
</thead>
<tbody>
<tr>
<td>- nationalTeam: String</td>
</tr>
<tr>
<td>- achievements: Collection&lt;String&gt;</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Skiers</th>
</tr>
</thead>
</table>
```
2.10 HATEOAS

Hypermedia as the Engine of Application State

What is it: including hypermedia links into a resource state

Objective

- A client of a REST application need only to know a single fixed URL
- Related resources should be discoverable dynamically from that URL

HOW: Hyperlinks included in the representations of returned resources

JSON EXAMPLE

```json
{
    "person": {"name": "Kelly"},
    "nationalTeam": {
        "Norway",
        "_links": {
            "nbskiers": {"href": "http://rest.norway-ski-team.no/nbskiers"}
        }
    },
    "achievements": ["12 Olympic Medals",
                     "9 World Championships"]
}
```
3 Marshalling/unmarshalling

1. Introduction
2. REST architectural style
3. Marshalling/unmarshalling
   3.1 From resource, to remote resource
   3.2 Marshalling and unmarshalling
   3.3 Representation formats
4. Hyper Text Transfer Protocol: basics reminder
5. From a Java instance to a neutral representation
6. Java RESTful service
7. Synthethis and go further
### 3.1 From resource, to remote resource

Client

```
skiers = service.path(addSkier).post(skier)
```

Implementation

```
addSkier(Skier skier){
    return skiers;
}
```

Interface (contract)

```
interface SkiersInterface {
    Skiers addSkier (Skier skier);
}
```
3.2 Marshalling and unmarshalling

- **Marshalling**: e.g. **Java instance** \(\mapsto\) **representation**

- **Unmarshalling**: e.g. one **representation** \(\mapsto\) **Php instance**
3.3 Representation formats

- Resources are distinct from their possible representations
- Format of a representation (i.e. content type) is defined by an Internet media type (previously known as a MIME type)
- Some common formats
  - plain text: text/plain
  - html: text/html
  - xml: text/xml, application/xml
  - code: application/javascript
  - json: application/json
  - image: image/jpeg, image/png, image/*
  - video: video/mpeg
1. Introduction
2. REST architectural style
3. Marshalling/unmarshalling
4. Hyper Text Transfer Protocol: basics reminder
   4.1 HTTP GET Request message
   4.2 HTTP GET Response message
   4.3 HTTP GET give it a try
5. From a Java instance to a neutral representation
6. Java RESTful service
7. Synthethis and go further
4.1 HTTP GET Request message

<table>
<thead>
<tr>
<th>GET</th>
<th>/hello HTTP/1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accept:</td>
<td>text/plain, text/html</td>
</tr>
<tr>
<td>%———empty line: end of header</td>
<td></td>
</tr>
</tbody>
</table>

- Sent to a web server to access one of its web resource
  - Request message (message method, identification of the resource inside the server, HTTP version)
    - For instance: GET /hello HTTP/1.1
  - Request Headers
    - accepted content types (e.g. Accept: text/plain, text/html)
    - accepted charsets (e.g. Accept-Charset: utf-8)
    - cookie (e.g. Cookie: Version=1; Skin=new;)
  - Request body (empty for a get)
### 4.2 HTTP GET Response message

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Return code</td>
<td>HTTP/1.1 200 OK</td>
</tr>
<tr>
<td>2</td>
<td>Date:</td>
<td>Date: Mon, 11 Nov 2013 17:47:24 GMT header (begin)</td>
</tr>
<tr>
<td>3</td>
<td>Server:</td>
<td>Server: Apachee/2.2.3 (Debian GNU/Linux)</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Per1/v5.8.4 PHP/5.2.6</td>
</tr>
<tr>
<td>6</td>
<td>Content-length:</td>
<td>Content-length: 327</td>
</tr>
<tr>
<td>7</td>
<td>Content-type:</td>
<td>Content-type: text/html</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>empty line (end of header)</td>
</tr>
<tr>
<td>9</td>
<td>&lt;HTML&gt;</td>
<td>&lt;HTML&gt; content</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>... document HTML</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>&lt;/HTML&gt;</td>
</tr>
</tbody>
</table>

- **Return code (line 1)**
  - 100 - 199: Information message
  - 200 - 299: Success (e.g., 200 OK)
  - 300 - 399: Redirections
  - 400 - 499: client-side errors (e.g., 404 Not Found, 403 Forbidden)
  - 500 - 599: server-side errors (e.g., 500 Internal Server Error)

- **Header (line 2–7)**

- **Resource content (line 9-11)**
4.3 HTTP GET give it a try

🔍 Give it a try

1. Visualize this simple page on your favourite navigator
   http://checkip.dyndns.org/ and visualize the headers with the network
   inspector of your navigator

2. Visualize the result with the **curl** command

   ```bash
   curl http://checkip.dyndns.org/
   ```

3. Connect to the web server with the **telnet** command

   ```bash
   telnet checkip.dyndns.org 80 % establish a connexion
   GET / HTTP/1.1
   HOST: checkip.dyndns.org
   % empty line
   ```

4. Use a REST client (such as postman)
5 From a Java instance to a neutral representation

1. Introduction
2. REST architectural style
3. Marshalling/unmarshalling
4. Hyper Text Transfer Protocol: basics reminder
5. From a Java instance to a neutral representation
   5.1 Java instance to State representation
   5.2 Json (Javascript Object Notation)
   5.3 Java Instance to XML document
6. Java RESTful service
7. Synthethis and go further
5.1 Java instance to State representation

- Several marshalling/unmarshalling means
  - Java serialization: binary representation
    ```java
    class MyClass implements Serializable {
    }
    instance = new MyClass();
    final FileOutputStream fichier = new FileOutputStream("file.ser");
    ObjectOutputStream oos = new ObjectOutputStream(fichier);
    oos.writeObject(instance);
    ```
  - JAXB: XML Document
  - Json: JavaScript Object Notation
5.1 Serialization

⚠️ Automatic serialization concerns

- Loop: Object graph with cycles
- Multiple references: Object graph with multiple reference paths to the same object

Figure source: Javadoc DataSerialize

readObject() does not maintain referential integrity
5.2 Json (Javascript Object Notation)

- "JSON is a **lightweight data-interchange format**. It is easy for humans to read and write. It is easy for machines to parse and generate." (json.org)
- Native representation of object in JavaScript
- Many programming languages include code to generate and parse JSON-format data
5.2 From Java instance to Json document

Java Class

```java
public class Person {
    private String name;
    private int age;
    private String gender;
}
```

Json Schema

```json
{
    "type": "object",
    "properties": {
        "name": {
            "type": "string"
        },
        "age": {
            "type": "integer"
        },
        "gender": {
            "type": "string"
        }
    }
}
```

Java object

```java
Person p = new Person("Bjoern Daehlie", 41, "Male");
```

Json document

```json
{
    "name": "Bjoern Daehlie",
    "age": 41,
    "gender": "Male"
}
```
5.2 Json with Jackson: a first example

- Many java libraries to serialize/deserialize Json strings, **jackson** is one of them
- Skier Example in ExemplesREST/REST-JSON-in-jackson
- A skier in Json

```java
skier = {
    "nationalTeam": "Norway",
    "achievements": ["12 Olympic Medals",
                     "9 World Championships",
                     "Winningest Winter Olympian",
                     "Greatest Nordic Skier"],
    "name": "Bjoern Daehlie",
    "age": 41,
    "gender": "Male"
}
```
5.2 Skier in json

```java
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.SerializationFeature;

Skier skier = createSkier();
//create ObjectMapper instance
ObjectMapper objectMapper = new ObjectMapper();
//configure Object mapper for pretty print
objectMapper.configure(SerializationFeature.INDENT_OUTPUT, true);
//writing to console, can write to any output stream such as file
String json = objectMapper.writeValueAsString(skier);

System.out.println("The initial skier: " + json);
PrintWriter out = new PrintWriter(FILE_NAME);
out.println(json);
out.close();
// Un—marshal as proof of concept
Skier clone = objectMapper.readValue(json, Skier.class);
json = objectMapper.writeValueAsString(clone);
```

2. Example from REST examples: directory REST-JSON-in-jackson
5.3 JAXB — Java Architecture for XML Binding

- JAXB used to transfer complex java objects in XML structured strings
  - Marshalling: Convert a Java object into an XML document
  - Unmarshalling: Convert an XML document into a Java Object
5.3 JAXB primitive data types

- Java basic types have a representation in xs types

<table>
<thead>
<tr>
<th>Java type</th>
<th>xs type</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.String</td>
<td>xs:string</td>
</tr>
<tr>
<td>int</td>
<td>xs:int</td>
</tr>
<tr>
<td>double</td>
<td>xs:double</td>
</tr>
<tr>
<td>boolean</td>
<td>xs:boolean</td>
</tr>
<tr>
<td>java.util.Date</td>
<td>xs:dateTime</td>
</tr>
</tbody>
</table>

- What about complex type?
5.3 From Java instance to XML document

Java Class

```java
public class Person {
    private String name;
    private int age;
    private String gender;
}
```

XSD schema

```xml
<xs:schema version="1.0"...>
  <xs:complexType name="person">
    <xs:sequence>
      <xs:element name="age" type="xs:int"/>
      <xs:element name="gender" type="xs:string" minOccurs="0"/>
      <xs:element name="name" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Java object

```java
Person p = new Person("Bjoern Daehlie", 41, "Male");
```

XML document

```xml
<person>
  <name>Bjoern Daehlie</name>
  <age>41</age>
  <gender>Male</gender>
</person>
```
### 5.3 JAXB annotations I

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>@XmlRootElement(namespace = &quot;namespace&quot;)</code></td>
<td>Root element for an XML tree</td>
</tr>
<tr>
<td><code>@XmlType(propOrder = &quot;field2&quot;, &quot;field1&quot;, .. )</code></td>
<td>XSD Type, order of fields</td>
</tr>
<tr>
<td><code>@XmlAttribute</code></td>
<td>Translated into an attribute (rather than an element)</td>
</tr>
<tr>
<td><code>@XmlTransient</code></td>
<td>Not translated into XML</td>
</tr>
<tr>
<td><code>@XmlAccessorType(XmlAccessType.FIELD)</code></td>
<td>All attributes translated (by default, only public + getter/setter)</td>
</tr>
<tr>
<td><code>@XmlElementWrapper(name=&quot;&quot;)</code></td>
<td>Add a wrapper XML element</td>
</tr>
<tr>
<td><code>@XmlElement(name = &quot;newName&quot;)</code></td>
<td>Rename a field (element)</td>
</tr>
</tbody>
</table>
5.3 Skier example

The JAXB examples are in the directory REST-JAXB-01

Annotations for the Skier class

```java
import javax.xml.bind.annotation.*;

@XmlElementWrapper(name = "achievements") // Addition of a wrapper for the collection
@XmlElement(name = "achievement") // Name of the elements in the collection
public class Skier extends Person {
    private String nationalTeam;
    @XmlElementWrapper(name = "achievements") // Addition of a wrapper for the collection
    @XmlElement(name = "achievement") // Name of the elements in the collection
    private Collection<String> achievements;

    public Skier() {}
    public Skier(final Person person, final String nationalTeam, final Collection<String> achievements) {
        super(person);
        this.nationalTeam = nationalTeam;
        this.achievements = achievements;
    }
}
```
Annotations for the Person class (not a root document)

```java
import javax.xml.bind.annotation.*;

@XmlElementType(XmlAccessType.FIELD) // All the fields, even the private ones are marshalled in XML
public class Person {
    private String name;
    private int age;
    private String gender;

    public Person() {}
    public Person(final Person person) {
        this(person.name, person.age, person.gender);
    }
    public Person(final String name, final int age, final String gender) {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }
}
```

3. REST examples: directory REST-JAXB-01
5.3 Skier example, XML root document

Example XML Document for a Skier object

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<skier>
  <name>Bjoern Daehlie</name>
  <age>41</age>
  <gender>Male</gender>
  <nationalTeam>Norway</nationalTeam>
  <achievements>
    <achievement>12 Olympic Medals</achievement>
    <achievement>9 World Championships</achievement>
    <achievement>Winningest Winter Olympian</achievement>
    <achievement>Greatest Nordic Skier</achievement>
  </achievements>
</skier>
```
5.3.1 JAXB in action I

JDK commands

- Java Class to XSD: `{schemagen}`

```bash
schemagen Skier.java Person.java
```

- XSD to java class: `{xjc}`

```bash
xjc schema1.xsd
```
5.3.1 JAXB in action II

Using the JAXB API to marshal and unmarshal

```java
import javax.xml.bind.*;

Skier skier = createSkier();

// Create a Marshaller for the skier class
JAXBContext ctx = JAXBContext.newInstance(Skier.class);
Marshaller m = ctx.createMarshaller();
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);

// Marshal and Write on a file
FileOutputStream out = new FileOutputStream(FILE_NAME);
m.marshal(skier, out);
out.close();

// Read from the file and Un—marshal
Unmarshaller u = ctx.createUnmarshaller();
Skier clone = (Skier) u.unmarshal(new File(FILE_NAME));
```
5.3 Handling specific marshalling/unmarshalling

- @XmlTransient attribute not marshalled
- `beforeMarshal` and `afterMarshal`: callbacks called (when defined) before and after marshalling
- `beforeUnmarshal` and `afterUnmarshal`: callbacks called (when defined) before and after unmarshalling
6 Java RESTful service

6.1 RESTful web service architecture
6.2 @path annotation and resource URI
6.3 RestFul class recap table
6.4 Input or output representation format
6.5 JAXB representation or JSON
6.6 Query parameters (GET)
6.7 Path parameters
6.8 Other params
6.9 Hello World in REST
6.10 Java Client example
6.11 Light Grizzly server
6.1 RESTful web service architecture

HTTP

Client

Server

Java

php

GlassFish

JAX-RS

JEE container

Apache HTTP Server

Lightweight Web server

JAX-RS

Java

RESTful WebServices in Java
6.2 @path annotation and resource URI

Each resource is identified by a URI defined by

- The server URL

```
http://localhost:9999/MyServer/
```

- The root resource class @path annotation for a RestFul java class

```
@Path("/hello") // http://localhost:9999/MyServer/hello
public class Hello {
  ...
}
```

- Additionally, a method may have a subidentification

```
@Path("replace") // http://localhost:9999/MyServer/hello/replace
public String replace(...) {
}
```
6.3 RestFul class recap table 1

- It may help to build a recap table for each RestFul java class

- Method name of the method
- Subpath subpath (optional)
- CRUD create, read, update or delete
- Http msg PUT, GET, POST, DELETE
- Parameters additional parameters in the URL
- Presentation format (e.g., json, XML, html, text)

- Example for class Hello, subpath hello, with one method replace (encore incomplet à ce stade du cours)

<table>
<thead>
<tr>
<th>method</th>
<th>SubPath</th>
<th>CRUD</th>
<th>http msg</th>
<th>parameters</th>
<th>presentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>replace</td>
<td>replace</td>
<td>update</td>
<td>TBD</td>
<td>TBD</td>
<td>TBD</td>
</tr>
</tbody>
</table>
6.4 Input or output representation format

- Defined with `@consumes` for input (POST and PUT) and `@produces` for output (GET)
- Defined for a class and/or overloaded on a method
- Client requirement and server representation offers should match
6.4 Input or output representation format II

- Client requirement defined in the GET request

```
GET /hello HTTP/1.1
Host: localhost
Accept: text/html, text/plain
```

- Service offeree

```java
@GET
@Produces("text/html")
public String readHTML() {
    return "<html><body>" + msg + "</body></html>";
}
```

- Recap table

<table>
<thead>
<tr>
<th>method</th>
<th>SubPath</th>
<th>CRUD</th>
<th>http msg</th>
<th>parameters</th>
<th>presentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>readHTML</td>
<td>/</td>
<td>read</td>
<td>GET</td>
<td>/</td>
<td>HTML</td>
</tr>
</tbody>
</table>

- Matching representation defined in the response header

```
HTTP/1.1 200 OK
Content-Type: text/html
<html><body>Hello</body></html>
```
### 6.5 JAXB representation or JSON I

- According to the client request, produces an XML or a JSON string.
- Return type is a class, the marshalling is done automatically by the jaxb or jackson library.

```java
@GET
@Path("searchskier")
//http://localhost:9999/MyServer/skiers/searchskier?name=xxx
@Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
public Skier getSkier(String name){
    ...
    Skier foundSkier= lookup(name);
    return foundSkier; // marshalled in XML with JAXB or in Json with Jackson
}
```

<table>
<thead>
<tr>
<th>method</th>
<th>SubPath</th>
<th>CRUD</th>
<th>http msg</th>
<th>parameters</th>
<th>presentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSkier</td>
<td>searchskier?name=</td>
<td>read</td>
<td>GET</td>
<td>name</td>
<td>XML or Json</td>
</tr>
</tbody>
</table>
6.6 Query parameters (GET) I

Parameters: URL query parameter

- Requested URL

```java
http://localhost:9999/MyServer/calc/add?a=3&b=4
http://localhost:9999/MyServer/calc/add?a=3
```

- Method definition

```java
@Path("/calc")
public class CalcRest {
  @GET
  @Path("/add")
  @Produces(MediaType.TEXT_PLAIN)
  public String addPlainText(@QueryParam("a") double a,
                               @DefaultValue("0") @QueryParam("b") double b) { return (a + b) + ""; }
}
```

<table>
<thead>
<tr>
<th>method</th>
<th>SubPath</th>
<th>CRUD</th>
<th>http msg</th>
<th>parameters</th>
<th>presentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addPlainText</td>
<td>add?a=&amp;b=</td>
<td>read</td>
<td>GET</td>
<td>a,b</td>
<td>TEXT</td>
</tr>
</tbody>
</table>
6.7 Path parameters I

- Parameters included in the path

- Requested URL

  ```
  http://localhost:9999/MyServer/calc/add/3/4
  ```

- Method definition

  ```java
  @Path("/calc")
  public class CalcRest {
      @GET
      @Path("/add/{a}/ {b}")
      @Produces(MediaType.TEXT_PLAIN)
      public String addPlainText(@PathParam("a") double a,
        @DefaultValue("0") @PathParam("b") double b) { return (a + b) + ""; }
  }
  ```

<table>
<thead>
<tr>
<th>method</th>
<th>SubPath</th>
<th>CRUD</th>
<th>http msg</th>
<th>parameters</th>
<th>presentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addPlainText</td>
<td>add/a/b=</td>
<td>read</td>
<td>GET</td>
<td>a,b</td>
<td>TEXT</td>
</tr>
</tbody>
</table>
6.8 Other params

- `@FormParam`, Form parameters (POST)
- `@HeaderParam`, parameter extracted from the header
- `@CookieParam`, parameter extracted from the cookie
import javax.ws.rs.*;

@Path("/hello") // This is the base path, which can be extended at the method level.
public class HelloRest {
    private static String msg = "Hello world";

    public static void setMsg(final String msg) { HelloRest.msg = msg; }

    @GET
    @Produces("text/plain")
    public String read() { return msg + "\n"; }

    @GET
    @Produces("text/html")
    public String readHTML() { return "<html><body>" + msg + "</body></html>"; }

    @GET
    @Produces("text/plain")
    @Path("/{extra}") // http://.../hello/xxx
    public String personalizedRead(final @PathParam("extra") String cus) { return HelloRest.msg + ": " + cus + "\n"; }

    @GET
    @Produces("text/plain")
    @Path("replace") // http://.../hello/replace?newmsg=xxx
    public String replaceAndRead(final @DefaultValue("") @QueryParam("newmsg") String newMsg) {
        System.out.println("replaceAndRead new_msg=" + newMsg);
        HelloRest.msg = newMsg;
        return HelloRest.msg + "\n";
    }
}
### 6. REST examples: directory REST-JAXREST-01

<table>
<thead>
<tr>
<th>method</th>
<th>SubPath</th>
<th>CRUD</th>
<th>http msg</th>
<th>parameters</th>
<th>presentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>/</td>
<td>read</td>
<td>GET</td>
<td>/</td>
<td>TEXT</td>
</tr>
<tr>
<td>readHTML</td>
<td>/</td>
<td>read</td>
<td>GET</td>
<td>/</td>
<td>HTML</td>
</tr>
<tr>
<td>personalized_read</td>
<td>{ extra }</td>
<td>read</td>
<td>GET</td>
<td>extra</td>
<td>TEXT</td>
</tr>
<tr>
<td>replaceAndRead</td>
<td>replace?newmsg=</td>
<td>replace</td>
<td>GET</td>
<td>newmsg</td>
<td>TEXT</td>
</tr>
<tr>
<td>replace</td>
<td>replace</td>
<td>update</td>
<td>PUT</td>
<td>newmsg</td>
<td>TEXT</td>
</tr>
<tr>
<td>delete</td>
<td>delete</td>
<td>delete</td>
<td>DELETE</td>
<td>/</td>
<td>/</td>
</tr>
</tbody>
</table>
6.10 Java Client example

(ExemplesREST/JAXREST-01)

```
restURI = "http://" + properties.getProperty("rest.serveraddress") + "/MyServer";
Client client = ClientBuilder.newClient();
URI uri = UriBuilder.fromUri(restURI).build();
WebTarget service = client.target(uri);
service.path("hello").path("replace").request().put(Entity.text("coucou"));
String getResult = service.path("hello").request().accept(MediaType.TEXT_PLAIN).get(String.class);
service.path("hello").path("delete").request().delete();
```

- **path("hello")**: subpath (or path parameters)
- **request()**: create an http request for the path
- **accept(MediaType.TEXTPLAIN)**: accepted representation format
- **get(String.class)**: message http GET, the return body is converted into a string
### 6.11 Light Grizzly server I

*(ExemplesREST/JAXREST-01)*

```java
/*
public static HttpServer startServer() throws IOException {
    // server address defined in a property file
    Properties properties = new Properties();
    FileInputStream input = new FileInputStream("src/main/resources/rest.properties");
    properties.load(input);
    baseURI = "http://" + properties.getProperty("rest.serveraddress") + "/MyServer/";

    // create a resource config that scans for JAX-RS resources and providers
    // in the server package
    final ResourceConfig rc = new ResourceConfig().packages("server");
    // create and start a new instance of grizzly http server
    // exposing the Jersey application at BASE_URI
    return GrizzlyHttpServerFactory.createHttpServer(URI.create(baseURI), rc);
*/

/*
public static void main(final String[] args) throws IOException {
    final HttpServer server = startServer();
    System.out.println(String.format("Jersey app started with WADL available at " + "%sapplication.wadl\nHit enter to stop it...", baseURI));
    System.in.read();
    server.shutdownNow();
}
```

- **The server will handle requests for all the RestFull classes in the server package**
7 REST Synthesis

- Easy to write and easy to test RESTful WebServices and REST clients
- As a consequence, a high percentage of deployed web services are RESTful services
Microservice architecture
A microservice is a software architectural style that structures an application as a collection of loosely coupled services. Advantages:

Advantages
- modularity
- continuous delivery
- better scalability
Microservices interaction patterns

- Services in a microservice architecture are often processes that communicate over a network
  - For synchronous interactions: REST over HTTP (one of the most popular)
  - For Asynchronous interactions: AMQP and Akka actors are good candidates
7.1 Some links to be studied

- Json 2 java http://www.jsonschema2pojo.org/

- swagger and open API
  https://swagger.io/docs/specification/about/
  - A language to describe API
  - Tools to generate the skeleton of classes from an API description
  - Tools to generate the documentation of an API (example of generated documentation https://www.versasense.com/docs/rest/)
References

Burke, B. (2010).
RESTful Java.
O'Reilly.

REST Architectural Styles and the Design of Network-based Software Architectures.
Doctoral dissertation, University of California, Irvine.

Java Web Services, Up and Running.
O'Reilly.