RESTful WebServices in Java

Revision : 708

Chantal Taconet

September 2019
1 Introduction

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
2 REST architectural style

1. Introduction

2. REST architectural style
   2.1 REST: Representational State Transfer
   2.2 Constraint: Client-server architecture: independance between the client and the server
   2.3 Constraint: Stateless
   2.4 Uniform interface: CRUD operations
   2.5 Are these operations sufficient to build an application?
   2.6 REST resource
   2.7 URI Naming conventions
   2.8 HATEOAS
       Hypermedia as the Engine of Application State

3. Marshalling/unmarshalling

4. Hyper Text Transfer Protocol: basics reminder
2.1 REST: Representational State Transfer

- **Architectural style** defined by Roy Fielding in 2000 [Fielding, 2000]

- Described by six identified constraints
  - **Client/server** architecture: independance between the client and the server
  - **Stateless**: no client context on the server  \(\implies\) client requests include all the necessary context
  - **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

- World Wide Web conforms to the REST architectural style

- Applications that conform to this architectural style are called **RESTful**

- Main advantages: scalability, simplicity of interfaces
2.2 Constraint: Client-server architecture: independence between the client and the server

- Separate the user interface concerns from the data storage concerns
- Portability across multiple platforms
- Allows the components to evolve independently

*Synchronous call*
2.3 Constraint: 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 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 $\leftrightarrow$ POST http method
  - Read resource $\leftrightarrow$ GET http method
  - Update resource $\leftrightarrow$ PUT http method
  - Delete resource $\leftrightarrow$ DELETE http method
2.5 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 Create entry</td>
<td>GET List entries</td>
<td>PUT Replace collection</td>
<td>DELETE Delete collection</td>
</tr>
<tr>
<td>Element</td>
<td>/</td>
<td>GET Get element</td>
<td>Replace element</td>
<td>Delete element</td>
</tr>
</tbody>
</table>
2.6 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
2.7 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.
2.8 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

Interface (contract)

```java
interface SkiersRemoteInterface {
    Skiers addSkier (Skier skier);
}
```

Client

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

Implementation

```java
addSkier(Skier){
return skiers;
   ...}
```

Communication messages
3.2 Marshalling and unmarshalling

- **Marshalling**: Java instance to one representation
- **Unmarshalling**: one representation to Java instance

```
client

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

marshalling arguments (skier)

unmarshalling results (skiers)

http message

network neutral format
XML
JSON
...
implementation

unmarshalling arguments (skier)

marshalling results (skiers)

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

http message
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
4. Hyper Text Transfer Protocol: basics reminder

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
   4.4 HTTP GET vs POST

5. From a Java instance to a neutral representation

6. Java RESTful service
4.1 HTTP GET Request message

```
1. GET /hello HTTP/1.1
2. Accept: text/plain, text/html
3. %--- empty line: end of header
```

- 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

HTTP/1.1 200 OK  return code
Date: Mon, 11 Nov 2013 17:47:24 GMT  header (begin)
Server: Apache/2.2.3 (Debian GNU/Linux)
        Perl/v5.8.4 PHP/5.2.6
Last-Modified: Wed, 28 Apr 2012 15:55:02 GMT
Content-length: 327
Content-type: text/html

    empty line (end of header)

<HTML>  content
    ... document HTML
</HTML>

■ 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

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

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

```
telnet checkip.dyndns.org 80
GET / HTTP/1.1
HOST: checkip.dyndns.org
```

4. Use the REST client plugin on your navigator (e.g., Firefox RestClient addon)
4.4 HTTP GET vs POST

- GET method, gets data, it has no input
- For input, use POST to create or PUT to update
- For HTML forms, which do have inputs, you may see GET with a modified URL but it is not recommended
## 4.4 HTTP GET vs POST II

<table>
<thead>
<tr>
<th>Method</th>
<th>GET</th>
<th>POST</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML</td>
<td><code>&lt;form method=&quot;GET&quot; action=&quot;AfficheGET.php&quot;&gt;</code></td>
<td><code>&lt;form method=&quot;POST&quot; action=&quot;AffichePOST.php&quot;&gt;</code></td>
</tr>
<tr>
<td>HTTP message</td>
<td>GET <code>/AfficheGET.php?Nom=Taconet&amp;Prenom=Chantal</code></td>
<td>POST <code>/AffichePOST.html</code> HTTP/1.1</td>
</tr>
</tbody>
</table>
| header | `Host: www.my.eu` | `Host: www.my.eu`  
`Content-Type: application/x-www-form-urlencoded`  
`Content-length: 27` |
| body   | None | `Nom=Taconet&Prenom=Chantal` |
4.4 HTTP GET vs POST III

- ✚ Give it a try with a curl client

```bash
1 curl --request POST --data 'Nom=Taconet&Prenom=Chantal'
   http://www-public.imtbs-tsp.eu/%7etaconet/REST/AffichePOST.php
```

```bash
```

- ✚ Give it a try with a REST client in your browser
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
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);
```

1. 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</code> (namespace = &quot;namespace&quot;)</td>
<td>Root element for an XML tree</td>
</tr>
<tr>
<td><code>@XmlType</code> (propOrder = &quot;field2&quot;, &quot;field1&quot;,.. )</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.*;

@XmlRootElement // XML Root
@XmlAccessorType(XmlAccessType.FIELD) // All the fields, even the private ones are marshalled in XML
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;
    }
}
```
5.3 Skier example, parent class

Annotations for the Person class (not a root document)

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

@XmlAccessorType(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;
    }
}
```

2. 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>
```

3. **REST examples: directory REST-JAXB-01**
5.3.1 JAXB in action I

JDK commands

- Java Class to XSD: `schemagen`
  ```
schemagen Skier.java Person.java
  ```

- XSD to java class: `xjc`
  ```
xjc schema1.xsd
  ```
5.3.1 JAXB in action II

Using the JAXB API to marshall and unmarshall

```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));
```

4. REST examples: directory REST-JAXB-01
5.3 Handling specific marshalling/unmarshalling

JAXB I

- `@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
5.3.2 Customize unmarshal example 1

See example JAXB-AfterUnmarshal

- A department contains employees

```java
@XmlElement
class Department {
    @XmlAttribute String name;
    @XmlElement(name="employee") List<Employee> employees = new ArrayList<Employee>();
    public Department() {}
    public Department(String name) {this.name=name;}
    public String getName() {return name;}
    public void addEmployee(Employee e) {employees.add(e);}
}
```

- An employee references its department
- Department reference is transient (not marshalled)
5.3.2 Customize unmarshal example II

Employee includes a afterUnmarshal to set the department reference

```java
class Employee {
    @XmlTransient Department department; // reference not marshalled
    @XmlAttribute String name;
    public Employee(String name, Department department) {
        this.name = name;
        this.department = department;
    }
    public Employee(){}
    public Department getDepartment() { return department; }

    public void afterUnmarshal(Unmarshaller u, Object parent) { // after JAXB
        unmarshal
        this.department = (Department)parent; // parent element in the xml is the
        department
    }
}
```
5.3.2 Customize unmarshal example III

```java
JAXBContext ctx = JAXBContext.newInstance(Department.class);
Marshaller m = ctx.createMarshaller();
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
// Marshal a Department object: 1st to stdout, 2nd to file
Department inf = createDepartment();
m.marshal(inf, System.out);
FileOutputStream out = new FileOutputStream(file_name);
m.marshal(inf, out);
out.close();
// Unmarshal as proof of concept
Unmarshaller u = ctx.createUnmarshaller();
Department clone = (Department) u.unmarshal(new File(file_name));

assert inf.employees.get(0).getDepartment().getName().equals(
    clone.employees.get(0).getDepartment().getName());
```
5.3.2 Customize unmarshal example IV

```xml
<department name="inf">
  <employee name="Denis Conan" />
  <employee name="Sophie Chabridon" />
</department>
```

- Result of a marshal
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
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

Client

Server

HTTP

Java

php

GlassFish

JEE container

APACHE HTTP SERVER

Lightweight Web server

Java

JAX-RS

JAX-RS
Each resource is identified by a URI defined by

- The server URL

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

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

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

- Additionally, a method may have a subidentification

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

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

<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>

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

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

- Service offeree

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

```java
HTTP/1.1 200 OK
Content-Type: text/html
<html><body>Hello</body></html>
```
6.5 JAXB representation

- `@produces("application/xml")`
- Return type is a class annotated `@XmlRootElement` or `@XmlElement`

```java
@GET
@Path("searchskier")
//http://localhost:9999/MyServer/skiers/searchskier?name=xxx
@produces("application/xml")
public Skier getSkier(@QueryParam("name") String name){
    Skier foundSkier = lookup(name);
    return foundSkier; // marshalled in XML with JAXB
}
```

<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</td>
</tr>
</tbody>
</table>
6.6 Query parameters (GET) I

Parameters: variables in the URLs

- Requested URL


- Method definition

```
@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 in the core of the URL

  - 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?
    public String replaceAndRead(final @DefaultValue("") @QueryParam("newmsg") String newMsg) {
        System.out.println("replaceAndRead new_msg=" + newMsg);
        HelloRest.msg = newMsg;
        return HelloRest.msg + "\n";
    }
}
6.9 Hello World in REST II

```java
@PUT
@Consumes("text/plain")
@Path("replace")
public void replace(final String newMsg) {
    System.out.println("replace new.msg=" + newMsg);
    HelloRest.msg = newMsg;
}

@DELETE
@Path("/delete")
public void delete() {
    HelloRest.msg = "";
    System.out.println("Message deleted.\n");
}
```

---

<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>read</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>

5. REST examples: directory REST-JAXREST-01
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
The server will handle requests for all the RestFul classes in the server package.
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.