RESTful WebServices in Java

Revision: 708

Chantal Taconet

April 2018
Companion examples

First of all, get the examples for this class (choose):

- download archive from:
  

- copy git repository:
  
  git clone git@gitlab.tem-tsp.eu:denis.conan/tsp-csc-mw-examples.git

- This class examples are in the directory Exemples/ExemplesREST
1 Introduction

1.1 What is REST?
1.2 REST: Representational State Transfer
1.3 REST resource
1.4 Uniform interface: CRUD operations
1.5 Are these operations sufficient to build an application?
1.6 Representation formats
1.7 How to implement a RESTful service?

2. Hyper Text Transfer Protocol: basics reminder

3. Representation of a Java object

4. Java RESTful service
1.1 What is REST?

- REST is a “URL friendly” way to retrieve distributed resources

- Well known examples of REST resources
  
  - ![Twitter Logo] Twitter: API
  
  - ![Google Maps Logo] 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 Logo] Open street map API
    
    Where is this place http://nominatim.openstreetmap.org/reverse?lat=48.858518&lon=2.294524&addressdetails=1
  
  - ![Bike Logo] State of bike stations in Paris, API
    
    https://api.jcdecaux.com/vls/v1/stations?contract=lyon&apiKey=91f170cdabb4c3227116c3e871a63e8d3ad148ee
1.2 REST: Representational State Transfer

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

- Described by six identified constraints
  - **Client/server** architecture: independence 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
1.3 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
1.4 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 $\mapsto$ POST http method
  - Read resource $\mapsto$ GET http method
  - Update resource $\mapsto$ PUT http method
  - Delete resource $\mapsto$ DELETE http method
1.5 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>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>
1.6 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
1.7 How to implement a RESTful service?

- Container of resources (http fluent)
  - Web server (e.g., apache web server, google GWS, NGINX)
  - Application server (e.g. JEE)
  - Light servers (e.g., home made HTTP servers)

- Resources may be implemented with many languages
  - PHP
  - C++
  - Java classes following the JAX-RS specification (several implementations, e.g. Jersey)
  - ...
2 Hyper Text Transfer Protocol: basics reminder

1. Introduction

2. Hyper Text Transfer Protocol: basics reminder
2.1 HTTP GET Request message
2.2 HTTP GET Request message
2.3 HTTP GET Response message
2.4 HTTP GET vs POST

3. Representation of a Java object

4. Java RESTful service
2.1 HTTP GET Request message

- 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)
2.2 HTTP GET Request message

นอกจาก

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)
2.3 HTTP GET Response message

```
HTTP/1.1 200 OK
Date: Mon, 11 Nov 2013 17:47:24 GMT
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

<HTML>
content
...document HTML
</HTML>
```
2.4 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
## 2.4 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 header</td>
<td>GET /AfficheGET.php?Nom=Taconet&amp;Prenom=Chantal HTTP/1.1 Host: <a href="http://www.my.eu">www.my.eu</a></td>
<td>POST /AffichePOST.html HTTP/1.1 Host: <a href="http://www.my.eu">www.my.eu</a> Content-Type: application/x-www-form-urlencoded Content-length: 27</td>
</tr>
<tr>
<td>body</td>
<td></td>
<td>Nom=Taconet&amp;Prenom=Chantal</td>
</tr>
</tbody>
</table>
2.4 HTTP GET vs POST III

- Give it a try with a curl client

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

```bash
```

- Give it a try with a REST client in your browser
3 Representation of a Java object

1. Introduction

2. Hyper Text Transfer Protocol: basics reminder

3. Representation of a Java object
   3.1 Java instance to State representation
   3.2 Serialization
   3.3 JAXB — Java Architecture for XML Binding
   3.4 JAXB data types
   3.5 JAXB annotations
   3.6 JAXB annotations example file
   3.7 Handling specific marshalling/unmarshalling JAXB
   3.8 Json (Javascript Object Notation)
   3.9 Json in Gson: a first example
   3.10 Skier in json
   3.11 Gson custom marshalling/unmarshalling
3.1 Java instance to State representation

- Marshalling: Java instance to one representation
- Unmarshalling: one representation to Java instance
- 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
### 3.2 Serialization

⚠️ Automatic serialization concerns

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

![Infinite Loop](image)

**readObject() does not maintain referential integrity**

Figure source: Javadoc DataSerialize
3.3 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
3.4 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?
### 3.4 JAXB complex data types

#### 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>
```
# 3.5 JAXB annotations

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>@XmlRootElement</code>&lt;br&gt;(namespace = &quot;namespace&quot;)</td>
<td>Root element for an XML tree</td>
</tr>
<tr>
<td><code>@XmlType</code>&lt;br&gt;(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>
3.6 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;
    }
}
```
3.6 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;
    }
}
```

1. REST examples: directory REST-JAXB-01
3.6 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>
```

2. REST examples: directory REST-JAXB-01
3.6.1 JAXB in action

JDK commands

- Java Class to XSD: `schemagen`

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

- XSD to java class: `xjc`

  ```
  xjc schema1.xsd
  ```
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));
```
3.7 Handling specific marshalling/unmarshalling
JAXB

- `@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
3.7.1 Customize unmarshal example I

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)
Employee includes a afterUnmarshall 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
        this.department = (Department)parent; // parent element in the xml is the 
        department
    }
}
```
3.7.1 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());
```
3.7.1 Customize unmarshal example IV

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

- Result of a marshal
3.8 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
Many java libraries to serialize/deserialize Json strings, **google-gson** is one of them

**Skier Example in ExemplesREST/JSON-in-gson**

**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"
}
```
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

Skier skier = createSkier();

Gson gson = new GsonBuilder().setPrettyPrinting().create();

// Marshal a Skier object: 1st to stdout, 2nd to file
String json = gson.toJson(skier);

// Un-marshall
Skier clone = (Skier) gson.fromJson(json, Skier.class);

4. Example from REST examples: directory REST-JSON-in-gson
3.11 Gson custom marshalling/unmarshalling

If necessary (e.g., references, loop) it is possible to register serializers

```java
public class ProductJsonSerializer implements JsonSerializer<Product> {
    @Override
    public JsonElement serialize(final Product product, final Type typeOfSrc, final JsonSerializationContext context) {
        final JsonObject json = new JsonObject();
        json.addProperty("i", product.getId());
        json.addProperty("n", product.getName());
        json.addProperty("p", product.getPrice());
        final JsonArray categoriesArray = new JsonArray();
        json.add("c", categoriesArray);
        for (final Category category : product.getCategories()) {
            categoriesArray.add(context.serialize(category));
        }
        return json;
    }
}

// register the serializer for the Product class
Gson gson = new GsonBuilder().registerTypeAdapter(Product.class, new ProductJsonSerialiser()).create();
```
If necessary, one can register a deserializer

```java
public class ProductJsonDeserializer implements JsonDeserializer<Product> {
    @Override
    public Product deserialize(final JsonElement json, final Type typeOfT, final JsonDeserializationContext context) throws JsonParseException {
        Product product = new Product();
        // Parsing will be done here.
        return product;
    }
}
```

// register the serializer for the Product class
Gson gson = new GsonBuilder().registerTypeAdapter(Product.class, new ProductJsonDeserializer());
4 Java RESTful service

4. Java RESTful service
4.1 REST JAX-RS architecture
4.2 @path annotation and resource URI
4.3 RestFul class recap table
4.4 Input or output representation format
4.5 JAXB representation
4.6 Form parameters (GET)
4.7 Parameters in the URL
4.8 Other params
4.9 Resource life cycle in Jersey
4.10 Example of a RestFull class constructor with injection
4.11 Injection of values
4.12 Hello World in REST
4.13 Java Client example
4.14 Light Grizzly server
4.15 REST Synthesis
4.16 Some links to be studied
4.1 REST JAX-RS architecture

Client

HTTP

Server

Java

Lightweight Web server

JAX-RS

JEE container

GlassFish

GlassFish

JAX-RS
4.2 @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 { ...
}
```

- Additionnally, a method may have a subidentification

```
@Path("/replace")  //http://localhost:9999/MyServer/hello/replace
public String replace(...) {
}
```
4.3 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)
4.4 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
### 4.4 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>
  ```
4.5 JAXB representation

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

```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>
4.6 Form parameters (GET) 1

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>
4.7 Parameters in the URL I

- Parameters in the core of the URL
- Requested URL

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

- Method definition

```
@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=&amp;b=</td>
<td>read</td>
<td>GET</td>
<td>a,b</td>
<td>TEXT</td>
</tr>
</tbody>
</table>
4.8 Other params

- @FormParam, Form parameters (POST)
- @HeaderParam, parameter extracted from the header
- @CookieParam, parameter extracted from the cookie
4.9 Resource life cycle in Jersey

- By default a new resource class instance is created for each request (@RequestScoped by default)
  - Firstly the constructor is called
  - Any requested values are injected (e.g., @QueryParam, @Context, see example below)
  - The appropriate method is invoked
  - Finally the object is made available for garbage collection.

- @Singleton annotation could be used (one instance by application), but it violates the stateless philosophy of REST, and necessitates concurrency protection
4.10 Example of a RestFull class constructor with injection

(from jersey user guide)

- Injection of @QueryParam parameters
- Injection of Request and UriInfo

```java
public SparklinesResource(
    @QueryParam("d") IntegerList data,
    @DefaultValue("0,100") @QueryParam("limits") Interval limits,
    @Context Request request,
    @Context UriInfo ui) {
    if (data == null) {
        throw new WebApplicationException(400);
    }
    this.data = data;
    this.limits = limits;
    if (!limits.contains(data)) {
        throw new WebApplicationException(400);
    }
    this.tag = computeEntityTag(ui.getRequestUri());
}
```
4.11 Injection of values

```java
@Path("resource")
public static class SummaryOfInjectionsResource {
    // injection into a class field
    @QueryParam("query")
    String param;
    @Context
    UriInfo uriInfo;
    @Context
    Request request;

    // injection into a resource method parameter
    @GET
    public String get(@QueryParam("query") String methodQueryParam) {
        return "query param: " + param;
    }

    // injection into a constructor parameter
    public SummaryOfInjectionsResource(@QueryParam("query") String constructorQueryParam) {
    }
}
```
4.12 Hello World in REST

(ExemplesREST/JAXREST-01) ^5

```java
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";
    }
}
```
4.12 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");
}
```
4.13 Java Client example

(ExemplesREST/JAXREST-01)

```java
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
- **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.
4.15 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
4.16 Some links to be studied

- Web browser REST Client plug-in
- retrofit annotations to write REST client
  http://square.github.io/retrofit/
- swagger language-agnostic interface to REST APIs
  http://swagger.io/getting-started/
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.