RESTful WebServices in Java

Revision : 708

Chantal Taconet

Septembre 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=91f170cdaabb4c3227116c3e871a63e8d3ad148ee

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 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 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 $\leftrightarrow$ POST http method
  - Read resource $\leftrightarrow$ GET http method
  - Update resource $\leftrightarrow$ PUT http method
  - Delete resource $\leftrightarrow$ 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 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. Synthetise and go further
3.1 From resource, to remote resource

Interface (contract)

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

```
client
implementation

@path(addSkier) @POST

skiers = service.

post (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>Line</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GET /hello HTTP/1.1</td>
</tr>
<tr>
<td>2</td>
<td>Accept: text/plain, text/html</td>
</tr>
<tr>
<td>3</td>
<td>%−−−−empty line: end of header</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

```plaintext
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>
```

- **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 % establish a connexion
   GET / HTTP/1.1
   HOST: checkip.dyndns.org
   % empty line
   ```

4. Use a REST client (such as postman)
6. Java RESTful service

6.1 RESTful web service architecture

6.2 Java Client example
6.1 RESTful web service architecture

Client

HTTP

Server

Java

GlassFish

APACHE HTTP SERVER

JEE container

Lightweight Web server

JAX-RS

JAX-RS

Client Server

JEE container

GlassFish

Java

Lightweight Web server

JAX-RS
6.2 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 (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
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

- Highly used in the domain of the IoT (mainly on Gateways and IoT platforms)

- Standardized REST thing API: W3C Web of Things API
  https://iot.mozilla.org/wot/
7.1 Some links to be studied


- swagger and open API
  [https://swagger.io/docs/specification/about/](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/](https://www.versasense.com/docs/rest/))
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.