RESTful Web Services

M.I. Capel

ETS Ingenierías Informática y
Telecomunicación
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada
Email: manuelcapel@ugr.es
http://lsi.ugr.es/mcapel/

October, 22nd 2020

Máster Universitario en Ingeniería Informática

1. Http methods and REST architectures
2. Java API for RESTful services
3. Example of simple WS implementation using Jersey

Representational State Transfer (REST)

Historic outline

Initially proposed by Roy Thomas Fielding in his PhD dissertation book: *Architectural Styles and the Design of Network-based Software Architectures*(2000)

Fundamental characteristics

- REST-based notation to be used is mainly based on the 1996 `Http 1.0` standard
- REST Server provides resource access and REST-Client accesses and modifies resources
- Client applications communicate with servers by using *Http verbs*: `GET, POST, DELETE, PUT, PATCH`
- The server can access *resources* that are identified through a `URI`-based (*Uniform Resource Identifier*) common interface and the operations used with HTTP
- Resources have several textual repres.: `XML, JSON, HTML, ...`

REST applications and services characteristics

RESTful Web applications

- Placed in different Java *packages*:
  - Data classes
  - Resources
- Services

Any defined REST-service

- will respond to HTTP operations (PUT, GET, POST, DELETE...)
- it's unambigously identified by a URL
- each one can accept different representations
- negotiation of delivery/acceptance-format of contents and representations

REST applications and services characteristics-II

RESTful Web-services are:

- Services that are based on HTTP operations and notation
- Servicess that are accessed through a previously published base-URL
- Supporting MIME (“Multipurpose Internet Mail Extensions”) types, XML, JSON, APPLICATION_XML, APPLICATION_JSON, etc.
- and operations (also known as ”HTTP verbs”): GET,PUT,PUT,DELETE,POST

REST supported operations

GET

Reading access without any lateral effects to the local resource representation or other resources whatsoever

PUT

With this operation a resource is replaced or created and it is an idempotent operation , as well as the GET operation PUT is most-often utilized for update capabilities, PUT-ing to a known resource URI with the request body containing the newly-updated representation of the original resource. On successful update it returns 200 (or 204 if not returning any content in the body).

PATCH

This request says that we would only send the data that we need to modify without modifying or effecting other parts of the data. Example: if we need to update only the first name, we pass only the first name.

REST supported operations

DELETE

This operation remove resources from the data repository. It is an idempotent operation

POST

This operation creates a new resource. This is not an idempotent operation. The POST verb is mostly utilized to create new resources. In particular, it's used to create subordinate resources. That is, subordinate to some other (e.g. parent) resource. On successful creation, return HTTP status 201, returning a Location header with a link to the newly-created resource with the 201 HTTP status.

Http methods

Recommended return values for primary HTTP methods which are combined with URI resources

No	**Verb**	**CRUD**	**Entire Collection**	**Specific Item**
1	POST	Create	201 (Created)	404 (Not Found), 409 (Conflict) if resource exists.
2	GET	Read	200 (OK)	200 (OK)
3	PUT	Update/Replace	404 (Not Found)	200 (OK) or 204 (No Content). 404 (Not Found *)
4	PATCH	Update/Modify	404 (Not Found)	200 (OK) or 204 (No Content). 404 (Not Found *)
5	DELETE	Delete	404 (Not Found)	200 (OK). 404 (Not Found *)

(*):404 (Not Found), if ID not found or invalid.

Explanation

1. 'Location' header with link to /customers/id containing new ID.
2. List of customers. Use pagination, sorting and filtering to navigate big lists.
3. Single customer. 404 (Not Found), if ID not found or invalid, unless you want to update/replace every resource in the entire collection.
4. if ID not found or invalid, unless you want to modify the collection itself.
5. if ID not found or invalid, unless you want to delete the whole collection—not often desirable.

Definition of the base URL of a resource

Idea fundamental

A SW implemented with RESTful technology must define the *base direction* of each one of the services that offers to its clients

Example:

```
1 com.sun.jersey.api.client.config.ClientConfig
2     config = new DefaultClientConfig();
3 com.sun.jersey.api.client.WebResource
4     service =
5 com.sun.jersey.api.client.Client.create(config).resource(
      getBaseURI());
```

Data exchange between the client and the *service*

`accept` protocol

```
service.accept(MediaType.TEXT_XML).get(String.class));
service.accept(MediaType.APPLICATION_XML).get(String.class));
service.accept(MediaType.APPLICATION_JSON).get(String.class));
```

We have to program with the prior pattern each one of the read(GET()), write(PUT()), update(PATCH(),POST()) operations..., which are going to be supported by the service

JAX-RS

Fundamental concepts

- REST support provided by Java for Web applications and services
- Leverages creation of XML and JSON documents by using JAXB (*Java architecture for XML binding*)
- Assumes Java Specification Request (JSR) 311 standard conformance
- Uses *annotations* for indentifying the *REST part* of Java classes

JAXB

Fundamental idea

- This is about a specific standard (*Java Architecture for XML Binding*) of use for obtaining a correspondence between 'regular' data objects (*POJO*) and their representation in XML
- The associated framework allow us to read/write from/in Java objects and in/from XML documents

JAXB annotations

`@XmlRootElement(namespace = "space_of_names")`	Root element of an "XML tree"
`@XmlType(propOrder = "field1",... )`	writting order for class fields into the XML
`@XmlElement(name = "newName")`	The XML element that is used instead[a]

[a] It only needs to be used if it is different from the name assigned by the JavaBeans framework

JAX-RS II

Definition

- JAX-RS is an API proposed by Java RESTful(JAX-RS) Web services creation
- It is considered a programming language API that provides the necessary support for WS creation, following the REST architectural pattern

JAX-RS uses annotations, introduced in Java SE 5 to make easier the development and deployment of client and server-(*endpoints*) that are needed for setting up RESTful Web services

JAX-RS Annotations

- @PATH(my_path): it is based on the servlet code and the URL disclosed by `web.xml` file
- @POST: will respond to an HTTP POST request
- @GET: will respond to an HTTP GET request
- @PUT: will respond to an HTTP PUT request
- @DELETE: will respond to an HTTP DELETE request
- @Produces(MediaTypes.TEXT_PLAIN[More types]): defines what MIME type returns a @GET annotated method [it can also have been "application_xml" o "application_json"]
- @Consumes(type, [more types]): defines what MIME type is consummed by this method
- @PathParam: for URL address values injection into a method argument

JAX-RS Implementations

- Apache CXF, open source infrastructure for Web services
- Jersey, Oracle's reference implementation
- RESTeasy, JBoss's reference implementation
- Restlet, created by Jorme Louvel, a pioneer of REST-based frameworks
- Apache Wink, Apache Software Foundation Incubator Project, the server module implements JAX-RS
- WebSphere Application Server of IBM
- WebLogic Application Server of Oracle
- Apache Tuscany
- Cuubez framework

Jersey

Fundamentals

- The JAX-RS implementation of reference proposed by Sun/Oracle
- Uses a Java *container–servlet* for WS implementation
- The *servlet* has to be prior registered in `web.xml` to work

What is Jersey actually? Extracted from *Java EE 6 tutorial*

- Jersey is the implementation of reference for quality software production of Sun following the JSR 311: JAX-RS.
- Jersey implements a support for the annotations defined in the JSR-311 standard, which makes easier for developers to create *web RESTful* services with Java and the Java Virtual Machine (JVM)
- Jersey adds supplemental characteristics to the JSR

Jersey II

Servlet part (hidden to client applications)

- The *servlet* is implemented inside Jersey and explores the code of Java classes for RESTful resources identification
- Analyzes the HTTP input request and selects the suitable class/method and the specific response method to attend the HTTP-based call
- Exploration based on annotations written in classes

Example

Domain class

```
1  @XmlRootElement
2  public class Todo{
3    private String id;
4    private String summary;
5    private String description;
6
7    public Todo(){
8    }
9    public Todo (String id, String summary){
10     this.id = id;
11     this.summary = summary;
12   }
13   public String getId() {
14     return id;
15   }
16   public void setId(String id) {
17     this.id = id;
18   }
19  ...
20 }
```

Example–II

Definition of DAO

"Data Access Object" is an object that provides an abstract interface to a DB or any other mechanism for persistence of entities of software Web-applications

- DAO provide us with some operations on specific data without disclosing, however, the supporting DB low-level details to the user applications
- It also provide us a mapping between operation calls performed in an application to the *persistence layer* of a Web service

Example–III

DAO `Todo`

```
1  import java.util.HashMap;
2  import java.util.Map;
3  //import the data domain model
4  public enum TodoDao {
5    INSTANCE;//for singleton.
6    private Map<String, Todo> contentsProvider = new HashMap<
         String, Todo>();
7    private TodoDao() {
8      Todo todo = new Todo("1", "Learn REST");
9      todo.setDescription("Read http://lsi.ugr.es/dsbcs/Documentos
           /Practica/practica3.html");
10     contentsProvider.put("1", todo);
11     todo = new Todo("2", "Learn something about DSBCS");
12     todo.setDescription("Read all the material placed at https
           ://pradoposgrado2021.ugr.es/mod/folder/view.php?id
           =22195");
13     contentsProvider.put("2", todo);  }
14   public Map<String, Todo> getModel(){
15     return contentsProvider;  }
16 }
```

Example–IV

Resource class

```
1  import javax.ws.rs.Consumes;
2  import javax.ws.rs.DELETE;
3  import javax.ws.rs.GET;
4  import javax.ws.rs.PUT;
5  import javax.ws.rs.Produces;
6  import javax.ws.rs.core.Context;
7  import javax.ws.rs.core.MediaType;
8  import javax.ws.rs.core.Request;
9  import javax.ws.rs.core.Response;
10 import javax.ws.rs.core.UriInfo;
11 import javax.xml.bind.JAXBElement;
12 ...
```

Example–V

...Resource class

```
1  import javax.servlet.http.HttpServletResponse;
2  @Path("/todos")//Mapping of resource into URL: todos
3  public class TodosRecurso {
4  //It allows inserting contextual objects in the class,
5  //for instance, ServletContext, Request, Response, UriInfo
6            @Context
7            UriInfo uriInfo;
8            @Context
9            Request request;
10 //Returns the list of all contained elements
11           @GET
12           @Produces(MediaType.TEXT_XML)
13           public List<Todo> getTodosBrowser() {
14             List<Todo> todos = new ArrayList<Todo>();
15             todos.addAll(TodoDao.INSTANCE.getModel().values());
16             return todos;
17           }
```

Example–VI

...Resource class

```
1  ...// To obtain the total number of elements stored in the
       service's data base
2          @GET
3          @Path("cont")
4          @Produces(MediaType.TEXT_PLAIN)
5          public String getCount() {
6            int cont = TodoDao.INSTANCE.getModel().size();
7            return String.valueOf(cont);
8          }
9          @Path("{todo}")
10         public TodoRecurso getTodo(@PathParam("todo") String
               id) {
11           return new TodoRecurso(urilnfo, request, id);
12         }
13 }
```

Example–VII

Configuration and deployment of the Web service

- It is necessary to program a simple *servlet dispatcher*: the `web.xml` configuration file
- The `web.xml` file is considered a service deployment description
- To deploy a WS we can use any container of Web applications, e.g.: Tomcat
- If we use Tomcat, we need to follow the usual steps:
  - To export the *Dynamic Web Project* created in Eclipse IDE to one `.war` file
  - To deploy the file above into the `webapps` folder of Tomcat
  - To create a client class to test the Web service

Example–VIII

web.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://xmlns.jcp.org/xml/ns/javaee" xsi:
       schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://
       xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd" id="WebApp_ID"
        version="3.1">
3    <display-name>Contents server with REST technology</display-
         name>
4    <welcome-file-list>
5      <welcome-file>index.html</welcome-file>
6      <welcome-file>index.htm</welcome-file>
7      <welcome-file>index.jsp</welcome-file>
8      <welcome-file>default.html</welcome-file>
9      <welcome-file>default.htm</welcome-file>
10     <welcome-file>default.jsp</welcome-file>
11   </welcome-file-list>
12   ...
```

Example–IX

...web.xml

```
1   <servlet>
2     <servlet-name>Jersey-implemented REST service</servlet-name>
3     <servlet-class>org.glassfish.jersey.servlet.ServletContainer
          </servlet-class>
4      <!-- It records resources held in mio.jersey.primero -->
5     <init-param>
6         <param-name>jersey.config.server.provider.packages</
              param-name>
7         <param-value>mio.jersey.primero</param-value>
8     </init-param>
9     <load-on-startup>1</load-on-startup>
10  </servlet>
11  <servlet-mapping>
12    <servlet-name>Jersey-implemented REST service</servlet-name>
13    <url-pattern>/rest/*</url-pattern>
14  </servlet-mapping>
15 </web-app>
```

Example–X

Deployment of the server

Client or *test* class for checking the implemented Web service

```
1  public class Tester {
2    public static void main(String[] args) {
3  // TODO Auto-generated method stub
4          ClientConfig config = new DefaultClientConfig();
5          Client client = Client.create(config);
6  WebResource service = client.resource(getBaseURI());
7  //create a third "todo" object, in addition to the other 2
8          Todo todo = new Todo("3","This is the summary of the
               third record");
9  ClientResponse response = service.path("rest").path("todos").
       path(todo.getId()).accept(MediaType.APPLICATION_XML).put(
       ClientResponse.class, todo);
10         System.out.print("Returned code: ");
11 //The code must be: 201 == created
12         System.out.println(response.getStatus());
13 //Shows the contents of the resource "Todos" as XML text
14 System.out.println("To show as XML Plain text");          System.
       out.println(service.path("rest").path("todos").accept(
       MediaType.TEXT_XML).get(String.class));
```

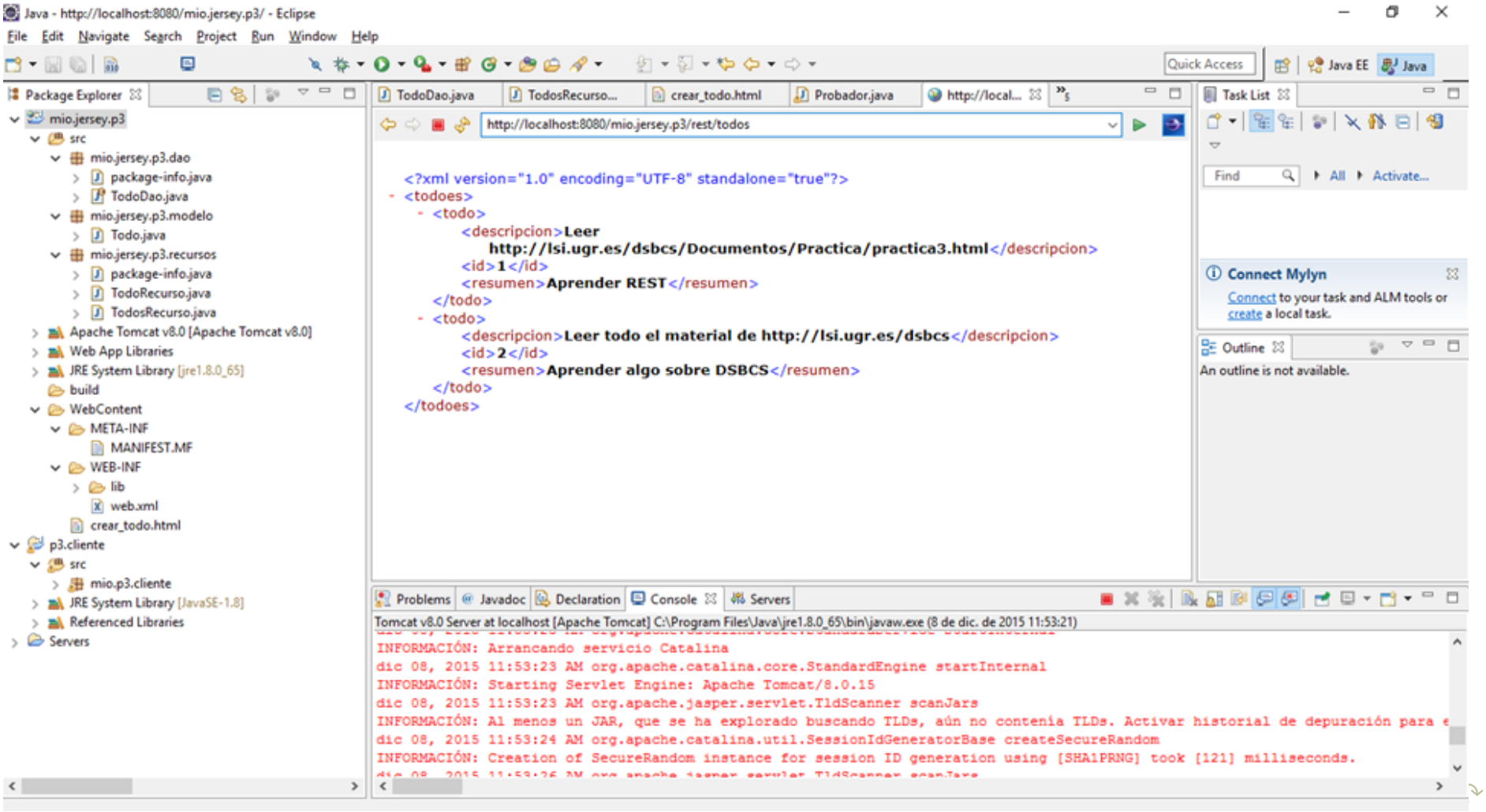
Test class for testing the implemented Web service-II

```
1  // Create a fourth "Todo" resource by a Web form
2  System.out.println("Form creation");
3  Form form = new Form(); form.add("id", "4");
4  form.add("summary", "Demonstration of the form client-library");
5  response = service.path("rest").path("todos").type(MediaType.
       APPLICATION_FORM_URLENCODED).post(ClientResponse.class,
       form);
6  System.out.println("Response with the form" + response.getEntity
       (String.class));
7  // An element with id = 4 must have been created
8  System.out.println("Resource contents, after sending the element
       with id=4");
9  System.out.println(service.path("rest").path("todos").
10 accept(MediaType.APPLICATION_XML).get(String.class));}
11 private static URI getBaseURI() {
12   return UriBuilder.fromUri("http://localhost:8080/mio.jersey.p3
         ").build();  }
13 }//the class ends
```

Program for deleting an object from the resource

```
1 // We are going to delete the "objects" with id=1 from the
      resource
2 service.path("rest").path("todos/1").delete();
3 // We show the contents of the resource "Todos", the element
      with id=1
4 // must have been deleted already
5 System.out.println("The element with id = 1 in the resource has
      been deleted");
6 System.out.println(service.path("rest").path("todos")
7 .accept(MediaType.APPLICATION_XML).get(String.class));
```

CRUD service deployed in a Tomcat server



Fundamental references I

Vogella (2016).

http://www.vogella.com/tutorials/REST/article.html

The Java EE Tutorial (2010).

http://docs.oracle.com/javaee/5/tutorial/doc/bnbpy.html

Legget, R. (2014).

Jersey RESTful WebService JAX-RS with JPA and Derby In Memory Database

http://robertleggett.wordpress.com/2014/01/29/jersey-2-5-1-restful-webservice-jax-rs-with-jpa-2-1-

Fundamental references II

Morales Machuca, C.A. (2010).

Estado del Arte: Servicios Web

http://camoralesma.googlepages.com/articulo2.pdf

“Eclipse can’t find Jersey imports”.

http://stackoverflow.com/questions/24512031/eclipse-cant-find-jersey-imports

Guia breve de servicios Web

http://www.w3c.es/Divulgacion/GuiasBreves/ServiciosWeb

Fundamental references III

Representational State Transfer en wikipedia

https://en.wikipedia.org/wiki/Representational_state_transfer

Murach, J., Urban, M.

“Java Servlets and JSP”. Murach, 2014