Web Services with Java/Jersey

Infrastructure for agile SW deployment

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/

November, 8th 2017

Máster Universitario en Ingeniería Informática

1 RESTful Web Services

2 Java API for RESTful services

REST

Definition

Representational State Transfer (REST) is a Web standard-based architectural style that uses the HTTP protocol

Characteristics

- “Todo” is a resource
- Resource access through a URL-based common interface and the operations used with HTTP
- REST Server that provides resource access and REST Client that accesses and mofidies resources

REST characteristics

RESTful Web applications

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

Any defined resource

- will respond to HTTP operations (PUT, GET, POST, DELETE...)
- is unambigously identified by a URL
- each one can accept differente representations
- contents negotiation and representations

REST supported operations

GET

Acceso en lectura al recurso sin producir efectos laterales
Reading access to the resource without any lateral effects

PUT

With this operation a resource is created anew and it is an idempotent operation , as well as the GET operation

DELETE

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

POST

This operation updates an existing resource or creates a new one

'RESTful' Web Service

RESTful

- Services thata based on HTTP operations and notation
- Servicess are accessed through a previously published base-URL
- Supported MIME ("Multipurpose Internet Mail Extensions") types, XML, JSON, etc.
- and operations: GET,PUT,PUT,DELETE,POST

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

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

JAX-RS III Annotations

- @PATH(my_path): it is based on the servlet code and the URL shown in `web.xml`
- @POST: points out that this method 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[Más tipos]): defines what MIME type returns a @GET annotated method
- @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)
as well Jersey adds supplemental characteristics to the JSR

Jersey II

Server part

- Servlet: explores the code of Java classes for RESTful resources identification
- Analyzes the HTTP input request and selects the suitable class and response method
- Exploration based on annotations written in classes

Example of simple WS using Jersey

Domain class

```
1  package com.mio.jersey.first;
2  import javax.xml.bind.annotation.XmlRootElement;
3
4  @XmlRootElement
5  public class Todo {
6    private String summary;
7    private String description;
8    public String getSummary() {
9      return summary;
10   }
11   public void setSummary(String summary) {
12     this.summary = summary;
13   }
14   public String getDescription() {
15     return description;
16   }
17   public void setDescription(String description) {
18     this.description = description;
19   }
20 }
```

Example of simple WS using Jersey - II

"resources" class

```
1  @Path("/todo")
2  public class TodoResource {
3    @GET
4    @Produces({ MediaType.APPLICATION_XML, MediaType.
         APPLICATION_JSON })
5    public Todo getXML() {
6      Todo todo = new Todo();
7      todo.setsummary("This is my primer todo");
8      todo.setdescription("This is my primer todo");
9      return todo;
10   }
11   @GET
12   @Produces({ MediaType.TEXT_XML })
13   public Todo getHTML() {
14     Todo todo = new Todo();
15     todo.setSummary("This is my primer todo");
16     todo.setDescription("This is my primer todo");
17     return todo;
18   }
19 }
```

Example of simple servlet dispatcher file with Jersey

`web.xml` configuration file

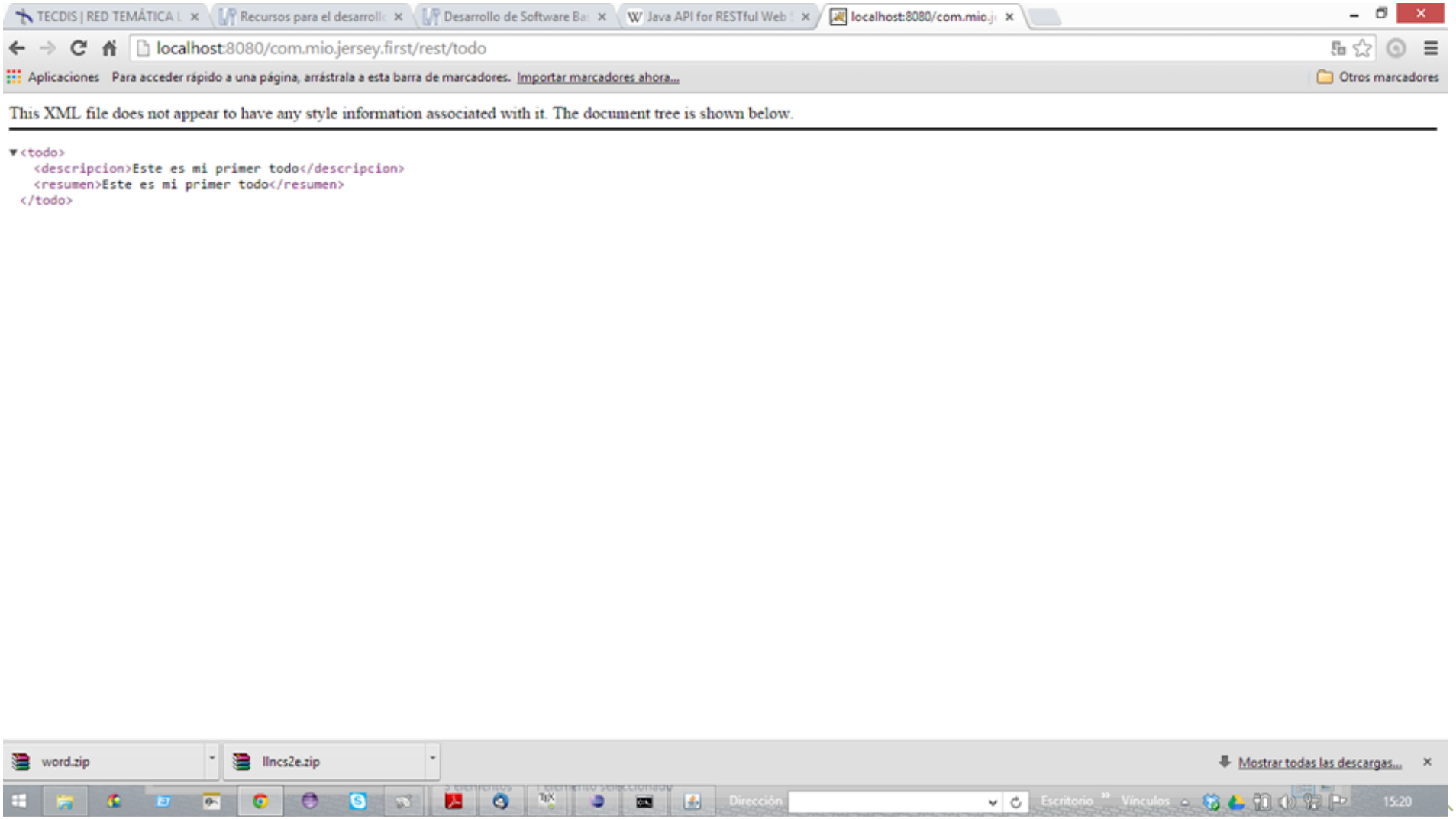
```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://java.sun.com/xml/ns/javaee" <!-- ...-->>
3      <display-name>com.mio.jersey.first</display-name>
4      <servlet>
5          <servlet-name>Jersey REST service</servlet-name>
6          <servlet-class>org.glassfish.jersey.servlet.
               ServletContainer</servlet-class>
7           <!-- To record resources located inside the package -->
8       <init-param>
9   <param-name>jersey.config.server.provider.packages</param-name>
10          <!-- Name of the package where the resources are -->
11          <param-value>com.mio.jersey.first</param-value>
12       </init-param>
13       <load-on-startup>1</load-on-startup>
14      </servlet>
15      <servlet-mapping>
16        <servlet-name>Jersey REST Service</servlet-name>
17        <url-pattern> /rest/ *</url-pattern>
18      </servlet-mapping>
19 </web-app>
```

Example of simple WS using Jersey

Configuration and deployment of the Web service

- 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 service

Deployment of the server



Client

```
1  public class Test {
2    public static void main(String[] args) {
3          ClientConfig config = new ClientConfig();
4      Client client = ClientBuilder.newClient(config);
5      WebTarget service= client.target(getBaseURI());
6      // Get the XML plain text
7      System.out.println(service.path("rest").path("todo").
8        request().accept(MediaType.TEXT_XML).get(String.class));
9      // Get the XML text for the application
10     System.out.println(service.path("rest").path("todo").
11       request().accept(MediaType.APPLICATION_JSON).get(String.
             class));
12     // Get the JSON text for the application
13     System.out.println(service.path("rest").path("todo").
14       request().accept(MediaType.APPLICATION_XML).get(String.
             class));
15   }
16   private static URI getBaseURI() {
17     return UriBuilder.fromUri("http://localhost:8080/com.mio.
           jersey.first").build();
18   }
19 }
```

Client importations

```
1  package com.mio.jersey.first.client;
2  import java.net.URI;
3  import javax.ws.rs.client.Client;
4  import javax.ws.rs.client.ClientBuilder;
5  import javax.ws.rs.client.WebTarget;
6  import javax.ws.rs.core.MediaType;
7  import javax.ws.rs.core.UriBuilder;
8
9  import org.glassfish.jersey.client.ClientConfig;
10 public class Test {
11 ...
12 }
```

Bibliografía Fundamental


Eden, A., Hirshfeld, Y., and Kazman, R. (2006).
Abstraction classes in software design.
IEE Software, 153(4):163–182.


Exposito, D. and Saltarello, A. (2009).
Architecting Microsoft .NET solutions for the enterprise.
Microsoft Press, Redmond, Washington.


Szyperski, C. (1998).
Component Software. Beyond Object-Oriented Programming.
Addison–Wesley, **Básica**.


Verissimo, P. and Rodrigues, L. (2004).
Distributed Systems for System Architects.
Kluwer Academic.