Práctica 1: Programación de servicio Web RESTful con EJB y ORM

Servicios Web `RESTful`

Comienzo de la práctica	10-10-2023
Fecha de entrega recomendada de la práctica	31-10-2023
Documento actualizado en fecha:	14-10-2023

1 Métodos HTTP y arquitecturas REST

El estilo arquitectónico de la Web se denomina "Representational State Transfer" (REST), cuya principal función consiste en proporcionar un conjunto coordinado de restricciones para el diseño de componentes en un sistema hipermedia distribuído que puede ser una base firme para el desarrollo de arquitecturas software más mantenibles y de altas prestaciones.

En la medida en que los sistemas software sean conformes con las restricciones que propugna REST se les podrá denominar "*sistemas RESTful*".

El estilo `REST` fue inicialmente propuesto por Roy Thomas Fielding en la defensa de su tesis doctoral (2000): "Architectural Styles and the Design of Network-based Software Architectures", desarrollándose el mencionado estilo al mismo tiempo que `HTTP 1.1.` (1996-1999), que estaba basado en HTTP 1.0 de 1996.

Estilo arquitectónico `REST`

Los sistemas *RESTful* normalmente, pero no siempre, se comunican utilizando HTTP y los mismos *verbos–HTTP*: `GET`, `POST`, etc. que los servidores–Web usan para abrir páginas y para enviar datos a servidores remotos. En `REST` todo es un recurso, que es accedido a través de una interfaz común invocable mediante los verbos estándar de HTTP: `GET`, `PUT`, `DELETE` y `POST`. Un sistema remoto con una interfaz `REST`, que utiliza recursos identificados mediante URIs (por ejemplo: \`personas\juan`), y permite el acceso a los recursos que se ofrecen mediante los métodos HTTP estándar, por ejemplo: `DELETE` *\personas\juan*eliminaría el campo `juan` del recurso.

De forma general, para desarrollar de acuerdo con este estilo, hemos de tener acceso a un *servidor* `REST`, que nos proporcione acceso a los mencionados *recursos*. Explicar cómo se desarrolla un sencillo servidor REST es el objetivo fundamental de esta documentación. También hemos de contar con un cliente programado conforme con la arquitectura `REST` que accederá y/o modificará dichos recursos, que se desarrollará también.

Los recursos–`REST` son archivos textuales y, por tanto, podrán tener diferentes representaciones; por ejemplo, podrían estar escritos utilizando: TEXT_XML, APPLICATION_XML, JSON, etc.Para encontrar un determinado recurso, el cliente sólo utilizará su identificación y

acceso a través de URLs y es responsabilidad del marco de trabajo “negociar" con el servidor REST el formato de transferencia según la citada representación del recurso. Los clientes pueden realizar una negociación de contenidos con el *servidor REST*. Un cliente conforme al estilo REST ha de poder solicitar, a través del protocolo HTTP, que se atiendan sus peticiones en una representación específica (XML-plano, aplicación–XML, JSON, etc.).

Para ser conforme con esta teconología, todo recurso *RESTful* tendrá que aceptar la invocación de las operaciones mediante los verbos comunes (GET, POST, PUT ...) de `HTTP`, ya sea desde las aplicaciones–cliente o desde otros servicios REST.

Métodos `HTTP`

Las arquitecturas `REST` utilizan normalmente los siguientes métodos (también denominados *verbos* HTTP) para resolver las llamadas a servidores:

- `GET` que sirve para definir un acceso de lectura al recurso sin efectos colaterales. El recurso nunca se ve alterado como consecuencia de una petición de este tipo.
- `PUT` crea un recurso en un URI específico. Si existiese ya el recurso nombrado por dicho URI, esta acción lo reemplazará. Si, por el contrario, no existe ningún recurso con esta identificación, se creará uno. Tal como ocurre con `GET`, `PUT` es también una operación idempotente, se puede repetir sin que produzca otro resultado diferente de la primera vez que se ejecutó. Las respuestas que proporciona esta operación no se guardan en el cache.
- `DELETE` es una operación idempotente para eliminar recursos en la parte servidora.
- `POST` se utiliza para actualizar un recurso existente o crear uno nuevo. Esta operación envía datos a un URI específico y confía en que el recurso ubicado allí se encargue de gestionar tal petición. En el momento de recibir los datos desde un `POST`, un servidor *REST* puede determinar qué corresponde hacer con estos datos en el contexto del recurso que gestiona. La operación `POST` no es idempotente, sin embargo, las respuestas se pueden guardar en el cache siempre que el servidor ajuste las cabeceras de control y expiración de cache apropiadas.

Puesto que los servicios Web “*RESTful*” están basados en el estándar HTTP, un SW de este tipo ha de definir el `URL` de base para cada uno de los servicios que ofrece a sus clientes; por ejemplo mediante la clase y método siguientes:
`UriBuilder.fromUri("http://localhost:8080/mio.jersey.primero").build();`

Por otra parte, los tipos `MIME` que podemos usar para que un cliente pueda entenderse, utilizando un mismo protocolo, con un servidor *REST* son generalmente: XML,JSON o HTML. La forma de programarlo dentro de un clase Java consiste en crearse un *cliente* configurado, al que se hace accesible el servicio Web desde su URL de base:

```
1 import jakarta.ws.rs.core.UriBuilder;
2 import jakarta.ws.rs.client.Client;
3 import jakarta.ws.rs.client.ClientBuilder;
4 import jakarta.ws.rs.client.WebTarget;
5 import org.glassfish.jersey.client.ClientConfig;
6 ...
7 ClientConfig clientConfig = new ClientConfig();
8 Client client = ClientBuilder.newClient(clientConfig);
9 WebTarget webTarget = client.target(getBaseURI());
```

Posteriormente, se establece el protocolo (`request` de la clase `jakarta.ws.rs.client.Invocation;`) de intercambio de datos con el servicio y se llama al método que nos interese:

- `import jakarta.ws.rs.client.Invocation;`

  `Invocation.Builder invocationBuilder = todoWebTargetWithQueryParam.request(MediaType.TEXT_XML);`

  También se han de programar cada una del conjunto de operaciones (`get()`, `put()`, etc.) que vayan a ser soportadas por el servicio que pretendemos desarrollar.

- `import jakarta.ws.rs.core.Response;`
  `Response response = invocationBuilder.get();`

2 Arquitecturas de Java para ligadura XML (JAXB)

(Esta sección no se ha actualizado; sigue utilizándose el framework `javax`, que ha sido sobrepasado por `jakarta`)

La Arquitectura Java para obtener una Ligadura XML (JAXB) es un estándar de Java que define cómo los objetos de Java ("POJOs" o Plain Old Java Objects) se convierten a/desde una notación `XML`. `JAXB` utiliza un conjunto estándar de correspondencias (o "mappings") para definir las citadas conversiones.

Se define un API para leer y escribir de/en objetos de Java y en/desde documentos XML. Como JAXB se define a través de una especificación, existen varias implementaciones de herramientas software comerciales para este estándar. Un buen proveedor de servicios ha de permitirnos el poder elegir una implementación JAXB concreta.

`JAXB` utiliza anotaciones para indicar los elementos centrales o "raíz" de un proyecto que, normalmente, programaremos como paquetes y clases en Java:

`@XmlRootElement(namespace = "espacionombre")`	Elemento raíz de un "árbol XML"
`@XmlType(propOrder = "campo2", "campo1",.. )`	Orden escritura campos en el XML
`@XmlElement(name = "nuevoNombre")`	El elemento XML que será usado (*)

Nota(*): Sólo necesita ser utilizado si es diferente del nombre que le asigna el marco de trabajo JavaBeans

Un ejemplo inicial de aplicación consiste en la implementación de un catálogo de películas que programaremos, utilizando la tecnología `JAXB`, como sigue:

```
1  package mio.xml.jaxb.modelo;
2  import javax.xml.bind.annotation.XmlElement;
3  import javax.xml.bind.annotation.XmlRootElement;
4  import javax.xml.bind.annotation.XmlType;
5
6  @XmlRootElement(name = "libro")
7  // Ahora se va a definir el orden en el cual se escriben los campos en el documento XML
8  @XmlType(propOrder = { "autor", "nombre", "editorial", "isbn" })
9  public class Libro {
10         private String nombre;
11         private String autor;
12         private String editorial;
13         private String isbn;
14         // Si te gusta otro nombre para una variable,se puede cambiar con facilidad
15         // antes de sacarlo hacia la corriente de salida XML:
16           @XmlElement(name = "titulo")
```

```
17            public String getNombre() {
18               return nombre;
19            }
20            public void setNombre(String nombre) {
21               this.nombre = nombre;
22            }
23            public String getAutor() {
24               return autor;
25            }
26            public void setAutor(String autor) {
27               this.autor = autor;
28            }
29            public String getEditorial() {
30               return editorial;
31            }
32            public void setEditorial(String editorial) {
33              this.editorial = editorial;
34            }
35            public String getIsbn() {
36              return isbn;
37            }
38            public void setIsbn(String isbn) {
39              this.isbn = isbn;
40            }
41 }
```

Vamos a programar la biblioteca de películas, que es el elemento raíz de nuestra aplicación–ejemplo:

```
1  package mio.xml.jaxb.modelo;
2  import java.util.ArrayList;
3  import javax.xml.bind.annotation.XmlElement;
4  import javax.xml.bind.annotation.XmlElementWrapper;
5  import javax.xml.bind.annotation.XmlRootElement;
6
7  @XmlRootElement(namespace = "com.mio.xml.jaxb.modelo")
8  public class Libreria {
9  // XmLElementWrapper genera un elemento envoltorio alrededor de una representacion XML
10 @XmlElementWrapper(name = "listaLibros")
11 // XmlElement fija el nombre de las entidades
12 @XmlElement(name = "libro")
13         private ArrayList<Libro> listaLibros;
14         private String nombre;
15         private String ubicacion;
16         public void setListaLibros(ArrayList<Libro> listaLibros) {
17               this.listaLibros = listaLibros;
18         }
19         public ArrayList<Libro> getListaDeLibros() {
20               return listaLibros;
21         }
22         public String getNombre() {
23               return nombre;
24         }
25         public void setNombre(String nombre) {
26               this.nombre = nombre;
27         }
28         public String getUbicacion() {
29               return ubicacion;
30         }
31         public void setUbicacion(String ubicacion) {
32               this.ubicacion = ubicacion;
33         }
34 }
```

Por último, ahora programamos el programa de demostración:

```
package mio.xml.jaxb.test;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;

import mio.xml.jaxb.modelo.Libro;
import mio.xml.jaxb.modelo.Libreria;

public class LibroPrincipal {
        private static final String LIBRERIA_XML = "./libreria-jaxb.xml";
        public static void main(String[] args) throws JAXBException, IOException{
                // TODO Auto-generated method stub
                ArrayList<Libro> listaLibros = new ArrayList<Libro>();
                  // crear los libros
                Libro libro1 = new Libro();
                libro1.setIsbn("978-0060554736");
                libro1.setNombre("El Juego");
                libro1.setAutor("Neil Strauss");
                libro1.setEditorial("Harpercollins");
                listaLibros.add(libro1);
                Libro libro2 = new Libro();
                libro2.setIsbn("978-3832180577");
                libro2.setNombre("Zonas humedas");
                libro2.setAutor("Charlotte Roche");
                libro2.setEditorial("Dumont Buchverlag");
                listaLibros.add(libro2);
```

```
  // crear libreria, asignar libro
                Libreria libreria = new Libreria();
                libreria.setNombre("Frankfurt Bookstore");
                libreria.setUbicacion("Aeropuerto de Frankfurt");
                libreria.setListaLibros(listaLibros);
                // crear el contexto JAXB e instanciar el marshaller
                JAXBContext contexto = JAXBContext.newInstance(Libreria.class);
                Marshaller m = contexto.createMarshaller();
                m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
                // Escribir en la corriente de salida: System.out
                m.marshal(libreria, System.out);
                // Escribir en File
                m.marshal(libreria, new File(LIBRERIA_XML));
                // conseguir las variables de nuestro archivo XML, que hemos creado anteriormente
                System.out.println();
                System.out.println("Salida desde nuestro archivo XML: ");
                Unmarshaller um = contexto.createUnmarshaller();
                Libreria libreria2 = (Libreria) um.unmarshal(new FileReader(LIBRERIA_XML));
                ArrayList<Libro> lista = libreria2.getListaDeLibros();
                    for (Libro libro : lista) {
                      System.out.println("Libro: " + libro.getNombre() + " de "
                          + libro.getAutor());
                    }
                  }
}
```

Finalmente, lo ejecutamos como una aplicación Java. Se ha de mostrar en pantalla el resultado que muestra la figura (**??**).

Se creará el archivo `libreria-jaxb.xml` en el directorio principal de nuestro proyecto:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:libreria xmlns:ns2="com.mio.xml.jaxb.modelo">
    <listaLibros>
        <libro>
            <autor>Neil Strauss</autor>
            <titulo>El Juego</titulo>
            <editorial>Harpercollins</editorial>
            <isbn>978-0060554736</isbn>
        </libro>
        <libro>
            <autor>Charlotte Roche</autor>
            <titulo>Zonas humedas</titulo>
            <editorial>Dumont Buchverlag</editorial>
            <isbn>978-3832180577</isbn>
        </libro>
    </listaLibros>
    <nombre>Frankfurt Bookstore</nombre>
    <ubicacion>Aeropuerto de Frankfurt</ubicacion>
</ns2:libreria>
```

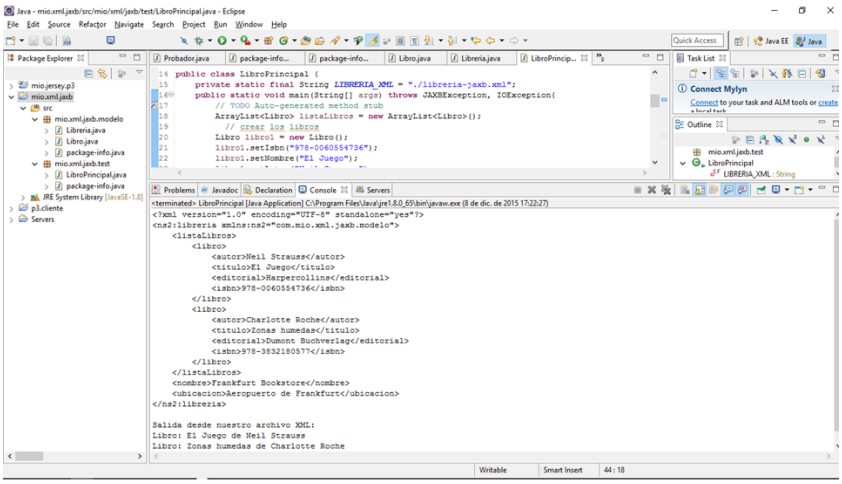


Figura 1: Contenido de Librería mostrado en System.out

3 Java Specification Request (JAX-RS)

Java define un soporte `REST` para las aplicaciones Web de cliente y para servicios Web (SW) a través del estándar JSR-311. Esta especificación se denomina `JAX-RS` de forma técnica y se aplica al API de Java para SW que sean del tipo "*RESTful*". `JAX-RS` utiliza anotaciones, que sirven de soporte para utilizar esta tecnología, para definir la parte que tiene que ver con el estilo `REST` dentro de las clases–Java programadas en nuestras aplicaciones.

3.1 `JAX-RS`

Cuando utilizamos el término *JAX–RS*, además del estándar, nos queremos referir a un conjunto de librerías FOSS[1] que ofrecen el soporte `REST` para aplicaciones y servicios *RESTful* cuando programamos con el lenguaje Java.

Java Specification Request (JSR) 311 utiliza un conjunto de anotaciones (`@XXX`)) para seleccionar la "parte *REST*" del código que programamos en una clase de Java. Se puede decir, por tanto, que utilizamos a Java como un *lenguaje soporte* para programar aplicaciones y servicios *RESTful* La implementación considerada de referencia actualmente para la JSR–311 se la conoce cn el nombre de *Jersey*. *Jersey* es un marco de trabajo abierto y fácil de utilizar para implementar SW que sean auténticamente "*RESTful*" y posteriormente desplegarlos en un contenedor de servlets de Java como *Tomcat* o en uno de aplicaciones Web como *Glassfish*[2].

Utilizando *Jersey*, el propio marco de trabajo asignará internamente un servlet que se encargará de analizar las peticiones HTTP entrantes y seleccionar las clases y métodos adecuados para responder a dicha petición. Esto lo hace de forma totalmente transparente para un programador de SW. Esta selección se basa en las anotaciones que hayamos escrito en la citada clase y en sus métodos programados.

Una aplicación Web conforme al estilo `REST` consistirá en clases de datos (o *recursos*) y *servicios*. Estos 2 tipos de elementos se mantienen normalmente en paquetes distintos del proyecto que nos hayamos creado en nuestro IDE, ya que a través de archivos de configuración (tal como `web.xml`), el servlet de *Jersey* será capaz de explorar los paquetes hasta encontrar las clases que contienen los *recursos RESTful* que se necesitan para resolver las llamadas que se han recibido en el servidor. El servlet ha de ser registrado en el archivo de configuración `web.xml`, para que pueda ser accedido por la aplicación Web que queremos construir.

3.2 Despliegue de un SW con Jersey

Para que funcione correctamente el servicio que hayamos programado y se puedan ejecutar sus métodos "*REST*", tendremos que proporcionar un URL de base al marco de trabajo para ubicar

[1] "Free and Open Source Software"

[2] GlassFish es un proyecto de servidor de aplicaciones de código abierto de la plataforma Jakarta EE iniciado por Sun Microsystems, patrocinado posteriormente por Oracle Corporation, y que ahora vive en la Fundación Eclipse y cuenta con el apoyo de Payara, Oracle y Red Hat

el servlet:
`http://localhost:8080/nombre-del-proyecto/patron-url/camino-para-el-resto-de-la-clase.`

El *patrón–url* se especifica en el archivo de configuración `web.xml`, por ejemplo, si queremos que nuestros SW comiencen con `rest` cuando se desplieguen, escribiremos el siguiente patrón: `<url-pattern>/rest/*</url-pattern>`.

El camino para ubicar el recurso se programa dentro de la clase que lo implementa, utilizando la anotación `@Path`, por ejemplo: `@Path("/todo")`.

Por último, mencionaremos que `JAX-RS` apoya la creación de contenidos XML o JSON a través de la Arquitectura Java para ligadura con XML (`JAXB`).

4 Anotaciones "RESTful"

Las anotaciones más importantes de JAX-RS se pueden ver en la lista siguiente:

- `@PATH(mi_camino)`: asigna el camino a la `URL` de base, al que se le añade `/mi_camino`. La citada `URL` está basada en el nombre que le damos a nuestro proyecto en el IDE, en el del patrón URL que indicamos en el archivo de configuración `web.xml` y en el nombre del recurso (por ejemplo: `@Path("/todo")`).
- `@POST`: indica que el siguiente método va a responder a una petición *HTTP* `POST`.
- `@GET`: indica que el siguiente método va a responder a una petición *HTTP* `GET`.
- `@PUT`: indica que el siguiente método va a responder a una petición *HTTP* `PUT`.
- `@DELETE`: indica que el siguiente método va a responder a una petición *HTTP* `DELETE`.
- `@Produces(TiposMedia.TEXT_PLAIN[Más tipos])`: define qué tipo `MIME` concreto se devuelve por un método que posee la anotación `@GET`. En este caso se produce: "text/plain", pero también podría ser: $\in \{\text{“}application\_xml\text{”}, \text{“}application\_json\text{”}, \text{“}application\_plain\text{”}\}$
- `@Consumes(tipo, [más tipos])`: define qué tipo MIME es consumido por este método.
- `@PathParam`: se utiliza para inyectar valores de la dirección URL en un parámetro de método. De esta manera podremos, por ejemplo, inyectar dinámicamente el `ID` de un recurso en el método llamado, para conseguir así el objeto adecuado.

5 Ejemplo tutorial

Vamos a crearnos un SW *RESTful* con el IDE Eclipse y la tecnología `JAX-RS` que nos proporciona *Jersey* para, de esta manera, demostrar cómo funcionan las aplicaciones Web que utilizan Java y JSR-311 para proporcionar estos servicios. A continuación se describen los pasos que hay que dar para llevar a cabo el proceso de desarrollo y despliegue de un SW correctamente. Los ejemplos que se van a presentar se han programado con el siguiente IDE y entorno de ejecución para Java:

- Eclipse IDE for Enterprise Java and Web Developers (includes Incubating components) Version: 2023-09 (4.29.0) Build id: 20230907-1323
- Jakarta ee 9
- Java Development Kit (JDK) version=11.0.18
- Jersey 3.1.0

5.1 Crearse un proyecto Web Dinámico

Hay que asegurarse que seleccionamos la opción de creación de un descriptor de despliegue (`web.xml`) al aceptar la vista Java del proyecto. Posteriormente, hay que copiar todos los `*.jar` de la distribución de Jersey que nos hayamos instalado en el disco duro en la carpeta `WEB-INF/lib` del proyecto en el IDE. El código que se va a presentar a continuación se supone que ha utilizado la librería con los "jars" adecuados en `/lib`: `mio.jersey.primero/WebContent/WEB-INF/l`

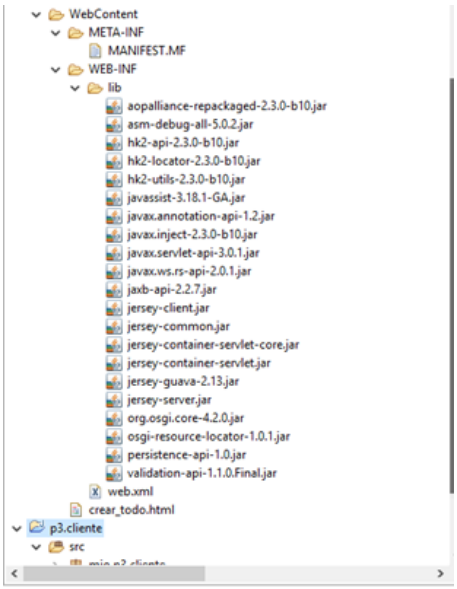


Figura 2: Librerias de *Jersey* necesarias para compilar el proyecto

5.2 Crear la clase que representa el *dominio* de los datos de la aplicación

Esto se hace programando algo similar a:

```java
package mio.jersey.primero.modelo;
import jakarta.xml.bind.annotation.XmlRootElement;
@XmlRootElement
//JAX soporta una correspondencia automatica desde una clase JAXB con anotaciones XML y JSON

public class Todo {
        private String resumen;
        private String descripcion;
        public String getResumen() {
                return resumen;
        }
        public void setResumen(String resumen) {
                this.resumen= resumen;
        }
        public String getDescripcion() {
                return descripcion;
        }
        public void setDescripcion(String resumen) {
                this.descripcion= resumen;
        }

}
```

5.3 Crear la clase que contiene el *recurso*

Programar la siguiente clase, que sólo devolverá una instancia de la clase que representa al dominio de datos "*todo*" de nuestra aplicación:

```java
package mio.jersey.primero.modelo;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;
//Esta clase solo devuelve una instancia de la clase Todo
@Path("/todo")
public class TodoRecurso {
        //Este metodo se llamara si existe una peticion XML desde el cliente
        @GET
        @Produces({MediaType.APPLICATION_ATOM_XML, MediaType.APPLICATION_JSON})
        public Todo getXML() {
                Todo todo = new Todo();
                todo.setResumen("Este␣es␣mi␣primer␣Todo");
                todo.setDescripcion("Este␣es␣mi␣primer␣Todo");
                return todo;
        }
        @GET
        @Produces({MediaType.TEXT_XML})
        public Todo getHTML() {
                Todo todo = new Todo();
                todo.setResumen("Este␣es␣mi␣primer␣Todo");
                todo.setDescripcion("Este␣es␣mi␣primer␣Todo");
                return todo;
        }
}
```

5.4 Archivo de descripción de despliegue

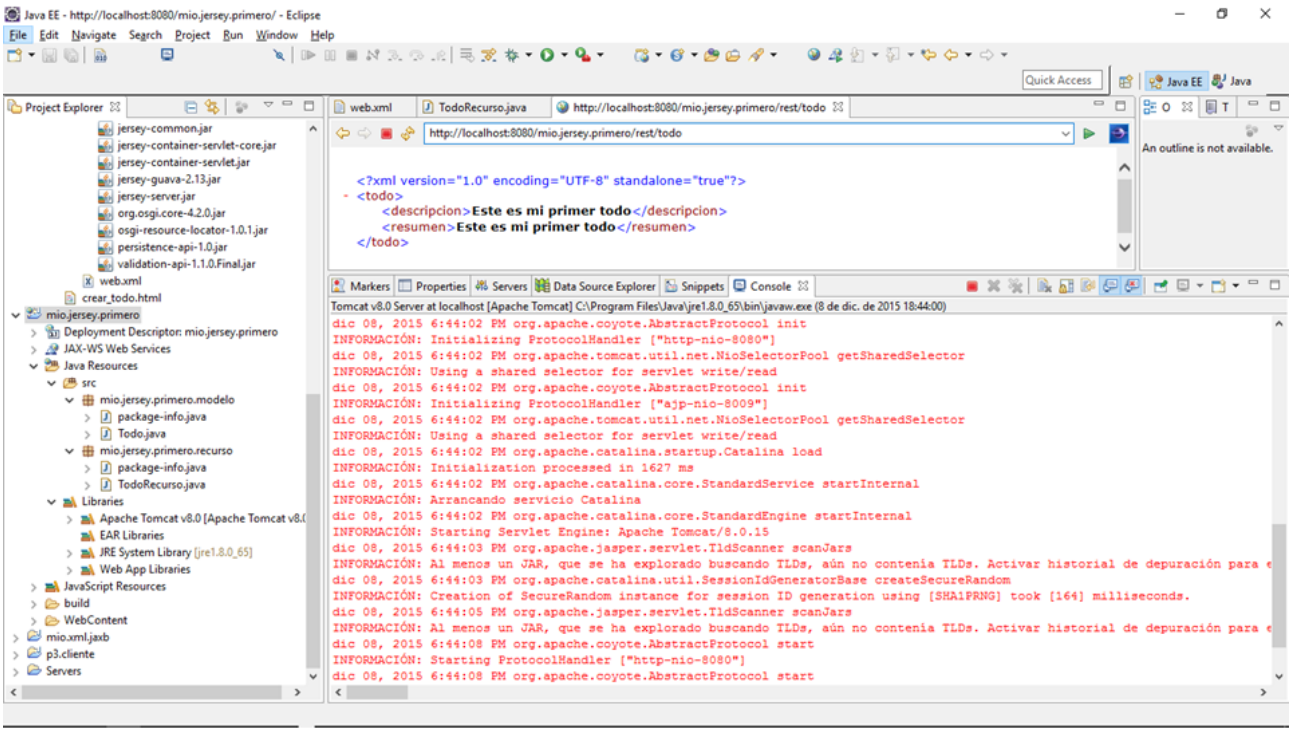
El archivo `web.xml` está ubicado dentro de la estructura del proyecto en el siguiente lugar: `p1-rest/Deployed Resources/webapp/WEB-INF/web.xml`. Todo esto de acuerdo con la estructura de carpetas que nos ha creado por defecto el proyecto Web dinámico. Para que el marco de trabajo encuentre los recursos y las clases adecuadas y haga funcionar nuestra aplicación de demostración sencilla, escribiremos el siguiente `web.xml`:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<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">

<display-name>mio.jersey.primero</display-name>
  <servlet>
    <servlet-name>Servicio REST de Jersey</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <!-- Registra recursos que estan ubicados dentro de mio.jersey.primero -->
    <init-param>
      <param-name>jersey.config.server.provider.packages</param-name>
      <param-value>mio.jersey.primero.modelo</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Servicio REST de Jersey</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>
</web-app>
```

5.5 Ejecutar la aplicación Web

Ahora comprobaremos que podemos ejecutar el servlet de nuestra aplicación y validar que podemos acceder a nuestro SW. La aplicación debería estar disponible bajo la siguiente dirección web: `http://localhost:8080/p1-rest/rest/todo`, tras ejecutarla con la opción *Run as->Run on Server*, obtendremos la siguiente salida en pantalla:



5.6 Crearse el cliente

Vamos a crearnos ahora en nuestro IDE otro proyecto (esta vez será un proyecto Java *regular*) que llamaremos algo así como:
`mio.jersey.primero.cliente`. No olvidar de añadir todos los `*.jar` de Jersey e incluirlos en el camino de construcción ("*build path*") del nuevo proyecto, ya que si no se incluye la librería adecuada, no funcionará (ver "Eclipse can't find Jersey imports" en los créditos de este documento). Por último, programaremos la clase del cliente, de forma parecida a lo siguiente:

```
package mio.jersey.primero.cliente;
import java.net.URI;
import jakarta.ws.rs.core.MediaType;
import jakarta.ws.rs.core.Response;
import jakarta.ws.rs.core.UriBuilder;
import jakarta.ws.rs.client.Client;
import jakarta.ws.rs.client.ClientBuilder;
import jakarta.ws.rs.client.Invocation;
import jakarta.ws.rs.client.WebTarget;
import org.glassfish.jersey.client.ClientConfig;

public class Test {
  public static void main(String[] args) {
    ClientConfig clientConfig = new ClientConfig();
    Client client = ClientBuilder.newClient(clientConfig);
    WebTarget webTarget = client.target(getBaseURI());
    WebTarget todoWebTarget = webTarget.path("rest");
    WebTarget helloworldWebTarget = todoWebTarget.path("todo");
    WebTarget helloworldWebTargetWithQueryParam = helloworldWebTarget.queryParam("greeting", "Hi World!");
    Invocation.Builder invocationBuilder = helloworldWebTargetWithQueryParam.request(MediaType.TEXT_XML);
    invocationBuilder.header("some-header", "true");
    Response response = invocationBuilder.get();
    System.out.println("Mostrar el codigo de respuesta:");
    System.out.println(response.getStatus());
    // Mostrar contenido para aplicaciones TEXT_XML
    System.out.println("Mostrar contenido del recurso como Texto XML Plano");
    System.out.println(response.readEntity(String.class));
    Invocation.Builder invocationBuilder2 = helloworldWebTargetWithQueryParam.request(MediaType.APPLICATION_
    invocationBuilder.header("some-header", "true");
    Response response2 = invocationBuilder2.get();
    System.out.println("Mostrar el codigo de respuesta:");
    System.out.println(response2.getStatus());
    // Mostrar contenido para aplicaciones TEXT_XML
    System.out.println("Mostrar contenido del recurso como Texto JSON de Aplicacion");
    System.out.println(response2.readEntity(String.class));
}
 private static URI getBaseURI(){
   return UriBuilder.fromUri("http://localhost:8080/p1-rest").build();
 }
}
```

En la consola de salida se mostrará, después de ejecutar `Test` como una palicación Java y formatear la salida, el siguiente resultado:

```
Mostrar el codigo de respuesta:
200
Mostrar contenido del recurso como Texto XML Plano
<?xml version="1.0" encoding="UTF-8"?>
<todo>
  <descripcion>Este es mi primer Todo</descripcion>
  <resumen>Este es mi primer Todo</resumen>
</todo>
Mostrar el codigo de respuesta:
200
Mostrar contenido del recurso como Texto JSON de Aplicacion
{"descripcion":"Este es mi primer Todo","resumen":"Este es mi primer Todo"}
```

6 Ejemplo Tutorial–2

Ahora se trata de crear un SW 'CRUD' (Create, Read, Update, Delete). Por supuesto, este servicio ha de ser *RESTful* y servirá para mantener contenedor de "objetos" definidos mediante el siguiente tipo de datos:

```
//Definicion del dominio de datos de la aplicacion
import jakarta.xml.bind.annotation.XmlRootElement;
@XmlRootElement
public class Todo {
  private String id;
  private String resumen;
  private String descripcion;
  public Todo(){
   }
  public Todo (String id, String resumen){
    this.id = id;
    this.resumen = resumen;
   }
  public String getId() {
        return id;
        }
  public void setId(String id) {
          this.id = id;
  }
  public String getResumen() {
          return resumen;
  }
  public void setResumen(String resumen) {
          this.resumen = resumen;
  }
  public String getDescripcion() {
          return descripcion;
  }
  public void setDescripcion(String descripcion) {
          this.descripcion = descripcion;
  }
}
```

6.1 Objeto de Acceso a Datos

Para poder implementarlo, con las tecnologías que hemos introducido en esta práctica, nos crearemos una clase *singleton* de Java basada en enumeración que actuará como el *proveedor de contenidos* (reseñas bibliográficas, catálogo de objetos multimedia, cartera de clientes, etc.) de nuestro SW:

```
import java.util.HashMap;
import java.util.Map;

public enum TodoDao {
          INSTANCE;
          private Map<String, Todo> proveedorContenidos = new HashMap<String, Todo>();

          private TodoDao() {
            //Creamos 2 contenidos iniciales
            Todo todo = new Todo("1", "Aprender␣REST");
            todo.setDescripcion("Leer␣http://lsi.ugr.es/dsbcs/Documentos/Practica/practica3.html");
            proveedorContenidos.put("1", todo);
            todo = new Todo("2", "Aprender␣algo␣sobre␣DSBCS");
            todo.setDescripcion("Leer␣todo␣el␣material␣de␣http://lsi.ugr.es/dsbcs");
            proveedorContenidos.put("2", todo);
          }
          public Map<String, Todo> getModel(){
            return proveedorContenidos;
          }
}
```

Hemos utilizado en concepto de "objeto de acceso a datos" (DAO), que nos proporciona una interfaz abstracta para facilitar el acceso a los datos del contenedor desde una base de datos u otro mecanismo de persistencia.

6.2 Recursos

Tendremos que programar una clase de Java que proporcione los servicios *REST* necesarios para acceder y/o modificar nuestro proveedor de contenidos a las aplicaciones y otros SW. Para hacerlo vamos a utilizar el marco de trabajo *Jersey* para servicios Web *RESTful*, que nos proporciona apoyo para trabajar con una interfaz de aplicaciones JAX y además es considerado actualmente como la implementación de referencia del estándar JSR 311. Conseguimos el acceso al API JAX-RS indicando las siguientes importaciones al comienzo de la clase del recurso:

```
1  package mio.jersey.segundo.modelo;
2  import jakarta.ws.rs.GET;
3  import jakarta.ws.rs.POST;
4  import jakarta.ws.rs.PUT;
5  import jakarta.ws.rs.DELETE;
6  import jakarta.ws.rs.Path;
7  import jakarta.ws.rs.PathParam;
8  import jakarta.ws.rs.Produces;
9  import jakarta.ws.rs.Consumes;
10 import jakarta.ws.rs.FormParam;
11 import jakarta.ws.rs.core.Context;
12 import jakarta.ws.rs.core.MediaType;
13 import jakarta.ws.rs.core.Request;
14 import jakarta.ws.rs.core.Response;
15 import jakarta.ws.rs.core.UriInfo;
16 import java.io.IOException;
17 import java.util.ArrayList;
18 import java.util.List;
19 import jakarta.servlet.http.HttpServletResponse;
20 import jakarta.xml.bind.*;
```

Tal como hicimos en el ejemplo tutorial anterior, dentro del paquete `recursos`, programamos una clase `TodoRecurso`(ver 5.3) que proporcina los métodos `GET` para acceder al recurso desde el navegador o desde una aplicación cliente, `PUT` para incluir contenidos y `DELETE` para suprimirlos. También añadimos la clase `TodosRecurso` en el mismo paquete, que amplia los servicios de la clase anterior incluyendo envío de datos con `POST` y de formularios Web, para las aplicaciones cliente y navegadores. La siguiente clase constituye el núcleo del SW que queremos implementar:

```
1  @Path("/todos")
2  public class TodosRecurso {
3    //Devolvera la lista de todos lo elementos contenidos en el proveedor al
4    //navegador del usuario.
5    @Context
6    UriInfo uriInfo;
7    @Context
8    Request request;
9    String id;
10 //Devolvera la lista de todos lo elementos contenidos en el proveeedor
11 //a las aplicaciones cliente
12   @GET
13   @Produces(MediaType.APPLICATION_JSON)
14   public List<Todo> getTodosBrowser() {
15     List<Todo> todos = new ArrayList<Todo>();
16     todos.addAll(TodoDAO.INSTANCE.getModel().values());
17     return todos;
18   }
```

```
19    @GET
20    @Produces({MediaType.TEXT_HTML,MediaType.APPLICATION_XML})
21    public List<Todo> getTodos(){
22            List<Todo> todos = new ArrayList<Todo>();
23            todos.addAll(TodoDAO.INSTANCE.getModel().values());
24            return todos;
25    }
26    @GET
27    @Path("cont")
28    @Produces(MediaType.TEXT_PLAIN)
29    public String getCount() {
30              int cont = TodoDAO.INSTANCE.getModel().size();
31              return String.valueOf(cont);
32    }
33    @PUT
34    @Consumes(MediaType.TEXT_XML)
35    public Response putTodo(JAXBElement<Todo> todo) {
36        Todo c = todo.getValue();
37        return putAndGetResponse(c);
38    }
39    @DELETE
40    public void deleteTodo() {
41        Todo c = TodoDAO.INSTANCE.getModel().remove(id);
42        if(c==null)
43            throw new RuntimeException("Delete: Todo con identificador " + id +  " no se encuentra");
44    }
45  //Para enviar datos al servidor como un formulario Web
46    @POST
47    @Produces(MediaType.TEXT_HTML)
48    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
49    public void newTodo(@FormParam("id") String id,
50     @FormParam("resumen") String resumen,
51     @FormParam("descripcion") String descripcion,
52     @Context HttpServletResponse servletResponse) throws IOException {
53      Todo todo = new Todo(id, resumen);
54      if (descripcion != null) {
55        todo.setDescripcion(descripcion);
56      }
57      TodoDAO.INSTANCE.getModel().put(id, todo);
58    servletResponse.sendRedirect("../crear_todo.html");
59    }
60    //////////////////////////
61    private Response putAndGetResponse(Todo todo) {
62        Response res;
63        if(TodoDAO.INSTANCE.getModel().containsKey(todo.getId())) {
64            res = Response.noContent().build();
65        } else {
66            res = Response.created(uriInfo.getAbsolutePath()).build();
67        }
68        TodoDAO.INSTANCE.getModel().put(todo.getId(), todo);
69        return res;
70    }
71  //Para poder pasarle argumentos a las operaciones en el servidor
72  //Permite por ejemplo escribir http://localhost:8080/p2-rest/rest/todos/1
73   @Path("{todo}")
74   public TodoRecurso getTodo(@PathParam("todo") String id) {
75      return new TodoRecurso(uriInfo, request, id);
76    }
77  }
```

6.3 Ejecución de la aplicación Web

Ahora podemos comprobar que el recurso y las operaciones *REST* que implementa el servicio funcionan correctamente ejecutado la aplicación Web con la opción *Run as-> Run on Server*. En la clase DAO hemos incluido ya 2 elementos iniciales (ver 6.1) para el proveedor de contenidos que queremos programar. Por tanto, enviando el siguiente URL `http://localhost:8080/p2-rest/rest/todos` desde un navegador se nos mostrará la siguiente información:

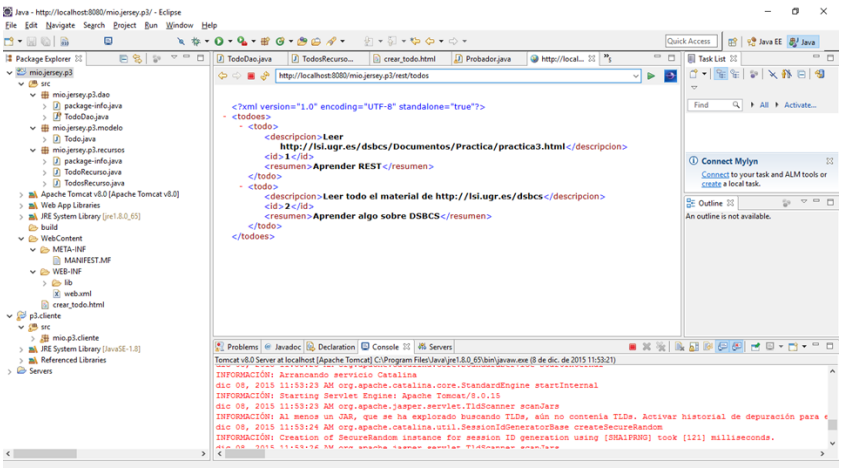


Figura 3: Servicio *REST* de contenidos con operaciones CRUD desplegado correctamente en un servidor Tomcat v8.0

6.4 Creación del cliente

Para mostrar de una manera sencilla toda la funcionalidad que nuestro servicio de provisión de contenidos puede ofrecer, vamos a programar varias versiones del cliente que irán mostrando progresivamente el resultado de las operaciones del servidor.

Primero vamos a programar dentro de un proyecto Java *regular* una clase principal que sirve para mostrar los contenidos del proveedor, después de introducir 2 elementos adicionales con `PUT` y dentro de un formulario, según los formatos que hemos denominado *texto XML plano* y *formulario HTML*:

```
1  package mio.jersey.segundo.cliente;
2  import java.net.URI;
3  import jakarta.ws.rs.core.MediaType;
4  import jakarta.ws.rs.core.Response;
5  import jakarta.ws.rs.core.UriBuilder;
6  import jakarta.ws.rs.client.Client;
7  import jakarta.ws.rs.client.ClientBuilder;
8  import jakarta.ws.rs.client.Invocation;
9  import jakarta.ws.rs.client.WebTarget;
10 import jakarta.ws.rs.client.Entity;
11 import org.glassfish.jersey.client.ClientConfig;
12 import mio.jersey.segundo.modelo.Todo;
13 import jakarta.ws.rs.core.Form;
14 public class Test {
15   public static void main(String[] args) {
16         // TODO Auto-generated method stub
17         ClientConfig clientConfig = new ClientConfig();
18         Client client = ClientBuilder.newClient(clientConfig);
19         WebTarget webTarget = client.target(getBaseURI());
20         //crearse un todo
21         Todo todo = new Todo("99", "Este es el resumen de otro registro");
22         WebTarget todoWebTarget = webTarget.path("rest");
```

```
23          WebTarget helloworldWebTarget = todoWebTarget.path("todos");
24          WebTarget helloworldWebTargetWithQueryParam = helloworldWebTarget.queryParam("greeting", "Hi␣World
25           ////////////////////
26          Invocation.Builder invocationBuilder = helloworldWebTargetWithQueryParam.request(MediaType.TEXT_XMI
27          invocationBuilder.header("some-header", "true");
28          Response response = invocationBuilder.put(Entity.entity(todo, MediaType.TEXT_XML));
29          comprobacion(response,2,helloworldWebTarget);
30          //// Obtener un Todo a partir de su identificador = "99"
31          WebTarget indicadorWebTarget= helloworldWebTarget.path("99");
32          WebTarget indicadorWebTargetWithQueryParam = indicadorWebTarget.queryParam("greeting", "Hi␣World!"
33          Invocation.Builder invocationBuilder4 = indicadorWebTargetWithQueryParam.request(MediaType.TEXT_XMI
34          invocationBuilder4.header("some-header", "true");
35          Response response4 = invocationBuilder4.get();
36          comprobacion(response4,4,indicadorWebTargetWithQueryParam);
37          //////// Eliminar el Todo con identificador 99
38          Response response5 = invocationBuilder4.delete();
39          comprobacion(response5,5,helloworldWebTargetWithQueryParam);
40          ///// Mostrar contenido para aplicaciones APPLICATION_JSON
41          Invocation.Builder invocationBuilder6 = helloworldWebTargetWithQueryParam.request(MediaType.APPLIC
42          invocationBuilder6.header("some-header", "true");
43          Response response6 = invocationBuilder6.get();
44          comprobacion(response6,6,helloworldWebTargetWithQueryParam);
45          ///////Insertar formulario y salvarlo como un html
46          Form forma = new Form();
47          forma.param("i","4");
48          forma.param("resumen", "Demostracion␣del␣cliente␣lib␣para␣formularios");
49          Invocation.Builder invocationBuilder7 = helloworldWebTargetWithQueryParam.request(MediaType.TEXT_HT
50          invocationBuilder7.header("some-header", "true");
51          Response response7 = invocationBuilder7.post(Entity.entity(forma,MediaType.APPLICATION_FORM_URLENCO
52          System.out.println("Formulario␣respuesta␣" + response7.getStatus());
53          System.out.println(response7.readEntity(String.class));
54          }
55    private static void comprobacion(Response response, int i, WebTarget t) {
56          System.out.println("Mostrar␣el␣codigo␣de␣respuesta:"+i);
57          System.out.println(response.getStatus());
58          // Mostrar contenido para aplicaciones JSON
59          WebTarget t1 = t.queryParam("greeting", "Hi␣World!");
60          Invocation.Builder invocationBuilder = t1.request(MediaType.APPLICATION_JSON);
61          invocationBuilder.header("some-header", "true");
62          Response response2 = invocationBuilder.get();
63          System.out.println("Mostrar␣el␣codigo␣de␣respuesta:"+(i+1));
64          System.out.println(response2.getStatus());
65          // Mostrar contenido para aplicaciones APPLICATION_XML
66          System.out.println("Mostrar␣contenido␣del␣recurso␣como␣HTML");
67          System.out.println(response2.readEntity(String.class));
68          }
69          private static URI getBaseURI(){
70          return UriBuilder.fromUri("http://localhost:8080/p2-rest/").build();
71          }
72  }
```

Si ahora ejecutamos el cliente desde nuestro IDE como una aplicación Java, se nos mostrará (después de formatearlo) el siguiente resultado en la consola:

```
Mostrar el codigo de respuesta:2
201
Mostrar el codigo de respuesta:3
200
Mostrar contenido del recurso como HTML
[{"id":"99","resumen":"Este␣es␣el␣resumen␣de␣otro␣registro"},{"descripcion":"Leer␣http://lsi.ugr.es/dsbcs/
Mostrar el codigo de respuesta:4
200
Mostrar el codigo de respuesta:5
200
Mostrar contenido del recurso como HTML
{"id":"99","resumen":"Este␣es␣el␣resumen␣de␣otro␣registro"}
Mostrar el codigo de respuesta:5
204
Mostrar el codigo de respuesta:6
200
Mostrar contenido del recurso como HTML
[{"descripcion":"Leer␣http://lsi.ugr.es/dsbcs/Documentos/Practica/practica3.html","id":"1","resumen":"Apre
Mostrar el codigo de respuesta:6
200
Mostrar el codigo de respuesta:7
200
Mostrar contenido del recurso como HTML
[{"descripcion":"Leer␣http://lsi.ugr.es/dsbcs/Documentos/Practica/practica3.html","id":"1","resumen":"Apre
Formulario respuesta 200
<!doctype html><html lang="es"><head><title>Segundo ejemplo con Jersey 3.1.0</title>
<style type="text/css">body {font-family:Tahoma,Arial,sans-serif;} h1, h2, h3, b {color:white;background-c
h1 {font-size:22px;}
h2 {font-size:16px;}
h3 {font-size:14px;}
p {font-size:12px;} a {color:black;} .line {height:1px;background-color:#525D76;border:none;}
</style></head><body>
<h1>Contenidos</h1><hr class="line" />
<p><b>Id</b> Muerte en el Nilo</p>
<p><b>resumen</b> Se trata de la historia de unos viajeros por el Nilo que han de hacer frente a un asesina
<p><b>Descripcion</b> El El misterio de viajes mas audaz de Agatha Christie. La tranquilidad de un crucero
anicos al descubrir que Linnet Ridgeway ha recibido un disparo en la cabeza. Era joven, elegante y hermosa
tenia todo... hasta que perdio la vida.</p><p><b>Descripcion</b>
<h3>Apache Tomcat/10.1.14</h3>
</body></html>
```

Créditos

- http://www.vogella.com/tutorials/REST/article.html
- http://docs.oracle.com/javaee/5/tutorial/doc/bnbpy.html
- http://robertleggett.wordpress.com/2014/01/29/jersey-2-5-1-restful-webservice-jax-rs-with-jpa-2-1-and-derby-in-memory-database/
- http://camoralesma.googlepages.com/articulo2.pdf
- "Eclipse can't find Jersey imports" http://stackoverflow.com/questions/24512031/eclipse-cant-find-jersey-imports
- http://www.w3c.es/Divulgacion/GuiasBreves/ServiciosWeb
- Representational State Transfer en wikipedia: https://en.wikipedia.org/wiki/Representational_state_transfer