Servicios Web RESTful

Comienzo de la práctica	22-10-2020
Fecha de entrega aconsejada	5-11-2020

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 utilice recursos identificados mediante URIs (por ejemplo: /personas/juan), permite el acceso a los recursos que ofrece 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*. También hemos de contar con un cliente programado conforme con la arquitectura REST, que accederá y/o modificará dichos recursos.

Los recursos–REST podrían tener diferentes representaciones textuales; por ejemplo, utilizando XML, JSON, etc., pero para encontrar un determinado recurso, sólo se utilizará su identificación y acceso a través de URLs.

Para ser conforme con esta teconología, todo recurso *RESTful* tendrá que aceptar la invocación de las operaciones comunes de HTTP desde las aplicaciones–cliente u otros servicios.

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.).

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 pone 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 enviar datos a un URI específico y el recurso existente allí ha de manejar esta operación (o crear uno nuevo). Es decir, en el momento de recibir los datos de una petición 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:
`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 y sus versiones para aplicaciones. 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 URI de base:

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

Posteriormente, se establece el protocolo (accept) de intercambio de datos con el servicio y se llama al método que nos interese:

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

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.

2 Arquitecturas de Java para ligadura XML (JAXB)

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:

```
package mio.xml.jaxb.modelo;
import java.util.ArrayList;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlElementWrapper;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(namespace = "com.mio.xml.jaxb.modelo")
public class Libreria {
// XmLElementWrapper genera un elemento envoltorio alrededor de una representacion XML
@XmlElementWrapper(name = "listaLibros")
// XmlElement fija el nombre de las entidades
@XmlElement(name = "libro")
        private ArrayList<Libro> listaLibros;
        private String nombre;
        private String ubicacion;
        public void setListaLibros(ArrayList<Libro> listaLibros) {
             this.listaLibros = listaLibros;
        }
        public ArrayList<Libro> getListaDeLibros() {
             return listaLibros;
        }
        public String getNombre() {
             return nombre;
        }
        public void setNombre(String nombre) {
             this.nombre = nombre;
        }
        public String getUbicacion() {
             return ubicacion;
        }
        public void setUbicacion(String ubicacion) {
             this.ubicacion = ubicacion;
        }
}
```

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. Para que funcione con un nivel de conformidad del compilador de Java = 1.8 o superior, hay que incluir las siguientes librerías en la carpeta WEB-INF/lib del proyecto (ver tip en `https://stackoverflow.com/questions/54705246/contextfactory-cannot-be-found`

- jaxb-api-2.3.1.jar
- jaxb-runtime-2.3.2.jar
- javax.activation-api-1.2.0.jar
- istack-commons-runtime-3.0.7.jar

Se ha de mostrar en pantalla el resultado que muestra la siguiente figura:

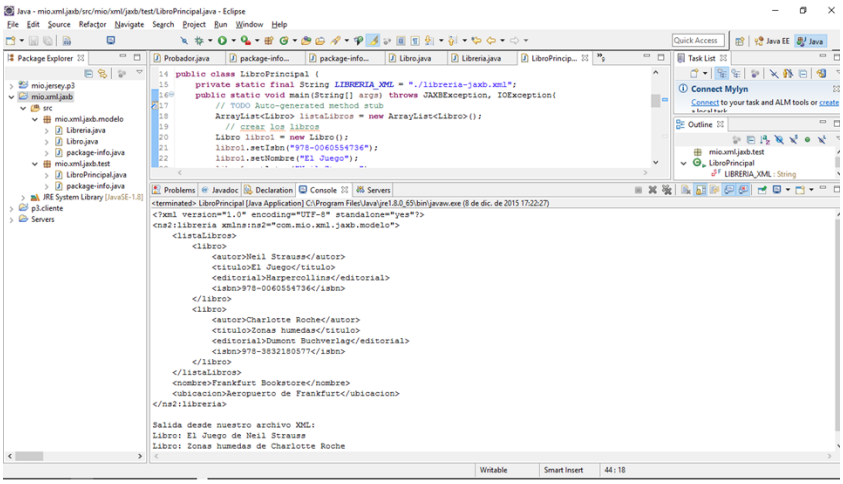


Figura 1: Contenido de Librería mostrado en System.out

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

```xml
<?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>
```

3 Java Specification Request (JAX-RS)

Java define un soporte `REST` para las aplicaciones y SW a través del estándar JSR-311. Esta especificación se denomina `JAX-RS` y se aplica al API de Java para Servicios Web que sean "*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. La implementación considerada de referencia actualmente para esta especificación 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*.

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`),

[1] "Free and Open Source Software"

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 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: $\in \{$"$application/xml$", "$application/json$", "$application\_plain$"$\}$
- `@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 Java EE IDE for Web Developers. Version: Mars.1 Release (4.5.1)
- awt.toolkit=sun.awt.windows.WToolkit
- os.arch=amd64; os.name=Windows 10; os.version=10.0
- osgi.arch=x86_64
- java.version=1.8.0_65
- java.vm.name=Java HotSpot(TM) 64-Bit Server VM

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/lib`.

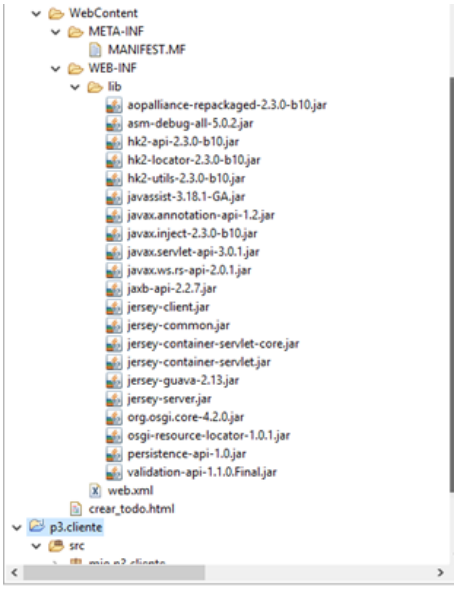


Figura 2: Librerias de *Jersey* necesarias para compilar el proyecto

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

Esto se hace programando algo similar a:

```
1  package mio.jersey.primero.modelo;
2  import javax.xml.bind.annotation.XmlRootElement;
3  @XmlRootElement
4  //JAX apoya una correspondencia automatica desde una clase JAXB con anotaciones a XML y JSON
5  public class Todo {
6    private String resumen;
7    private String descripcion;
8    public String getResumen() {
9      return resumen;
10   }
11   public void setResumen(String resumen) {
12     this.resumen = resumen;
13   }
14   public String getDescripcion() {
15     return descripcion;
16   }
17   public void setDescripcion(String descripcion) {
18     this.descripcion = descripcion;
19   }
20 }
```

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:

```
1  package mio.jersey.primero.modelo;
2  import javax.ws.rs.GET;
3  import javax.ws.rs.Path;
4  import javax.ws.rs.Produces;
5  import javax.ws.rs.core.MediaType;
6  //Esta clase solo devuelve una instancia de la clase Todo
7  @Path("/todo")
8  public class TodoRecurso {
9  // Este metodo se llamara si existe una peticion XML desde el cliente
10   @GET
11   @Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
12   public Todo getXML() {
13     Todo todo = new Todo();
14     todo.setResumen("Este␣es␣mi␣primer␣todo");
15     todo.setDescripcion("Este␣es␣mi␣primer␣todo");
16     return todo;
17   }
18 //Lo que sigue se puede utilizar para comprobar la integracion con el navegador que utilicemos
19   @GET
20   @Produces({ MediaType.TEXT_XML })
21   public Todo getHTML() {
22     Todo todo = new Todo();
23     todo.setResumen("Este␣es␣mi␣primer␣todo");
24     todo.setDescripcion("Este␣es␣mi␣primer␣todo");
25     return todo;
26   }
27 }
```

5.4 Archivo de descripción de despliegue

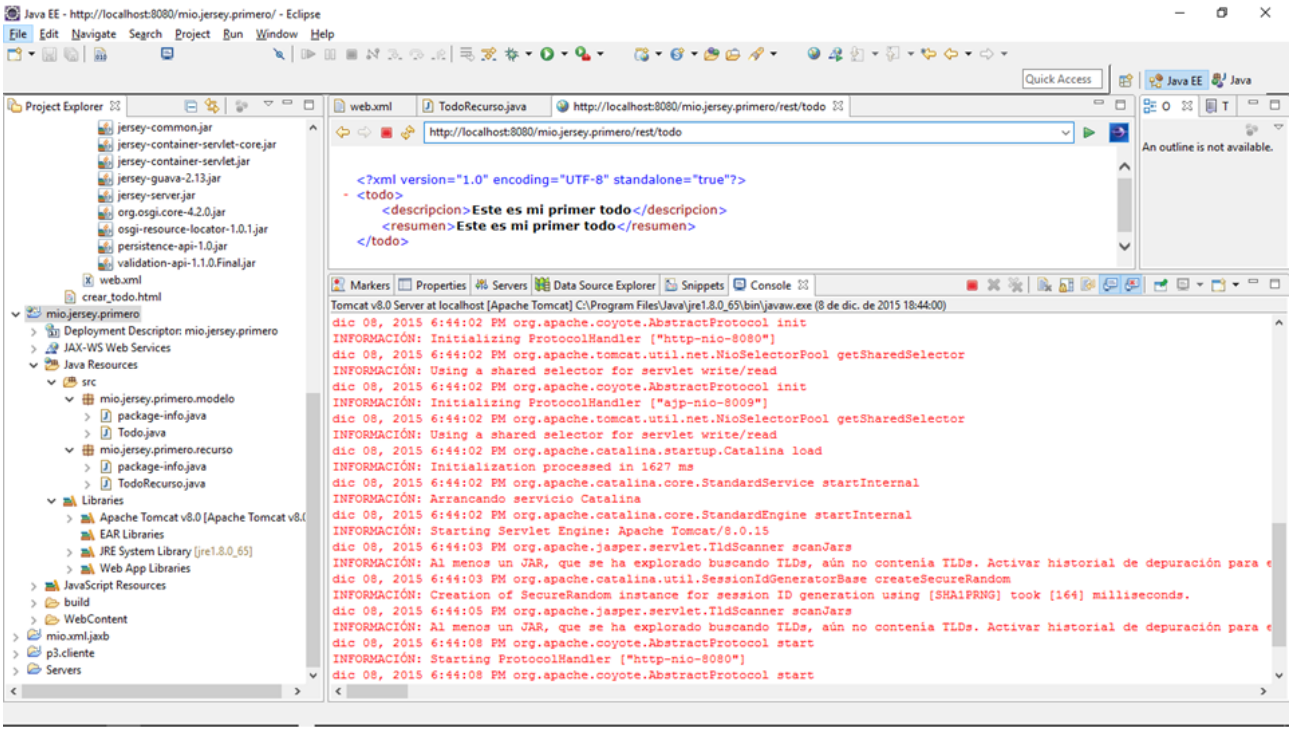
El archivo `web.xml` está ubicado dentro de la estructura del proyecto en el siguiente lugar: `mio.jersey.primero/WebContent/WEB-INF/web.xml`. Todo esto de acuerdo con la estructura de carpetas que nos ha creado por defecto el proyecto Web dinámico. Para que el marco

de trabajo encuentre los recursos y las clases adecuadas y haga funcionar nuestra aplicación de demostración sencilla, escribiremos el siguiente `web.xml`:

```
<?xml 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.recurso</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/mio.jersey.primero/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:

```
1  package mio.jersey.primero.cliente;
2  import java.net.URI;
3  import javax.ws.rs.core.MediaType;
4  import javax.ws.rs.core.UriBuilder;
5  import com.sun.jersey.api.client.Client;
6  import com.sun.jersey.api.client.WebResource;
7  import com.sun.jersey.api.client.config.ClientConfig;
8  import com.sun.jersey.api.client.config.DefaultClientConfig;
9
10 public class Test {
11         public static void main(String[] args) {
12                 // TODO Auto-generated method stub
13                 ClientConfig config = new DefaultClientConfig();
14                 Client cliente = Client.create(config);
15                 WebResource servicio = cliente.resource(getBaseURI());
16                 // Mostrar el contenido del recurso Todos como texto XML
17                 System.out.println("Mostrar␣contenido␣del␣recurso␣como␣Texto␣XML␣Plano");
18                 System.out.println(servicio.path("rest").
19                                 path("todo").accept(MediaType.TEXT_XML).get(String.class));
20                 // Mostrar contenido para aplicaciones XML
21                 System.out.println("Mostrar␣contenido␣del␣recurso␣para␣aplicacion␣XML");
22                 System.out.println(servicio.path("rest").
23                                 path("todo").accept(MediaType.APPLICATION_XML).get(String.class));
24         }
25         private static URI getBaseURI(){
26                 return UriBuilder.fromUri("http://localhost:8080/mio.jersey.primero").build();
27         }
28 }
```

En la consola de salida se mostrará, después de formatearlo, el siguiente resultado:

```
1  Mostrar contenido del recurso como Texto XML Plano
2  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
3  <todo>
4     <descripcion>Este es mi primer todo</descripcion>
5     <resumen>Este es mi primer todo</resumen>
6  </todo>
7  Mostrar contenido del recurso para aplicacion XML
8  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
9  <todo>
10   <descripcion>Este es mi primer todo</descripcion>
11   <resumen>Este es mi primer todo</resumen>
12 </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
@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:

```
import javax.ws.rs.Consumes;
import javax.ws.rs.FormParam;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Request;
import javax.ws.rs.core.UriInfo;
```

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:

```
//El paquete recursos contiene esta clase y TodoRecurso: GET (para el navegador y las aplicaciones),
//PUT y DELETE.
package mio.jersey.segundo.recursos;
...
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import javax.servlet.http.HttpServletResponse;
// Hara corresponder el recurso al URL todos
@Path("/todos")
public class TodosRecurso {
        // e.g. ServletContext, Request, Response, UriInfo
        @Context
        UriInfo uriInfo;
        @Context
        Request request;
        // Devolvera la lista de todos lo elementos contenidos en el proveedor al
        //navegador del usuario
        @GET
        @Produces(MediaType.TEXT_XML)
        public List<Todo> getTodosBrowser() {
           List<Todo> todos = new ArrayList<Todo>();
           todos.addAll(TodoDao.INSTANCE.getModel().values());
           return todos;
        }
        // Devolvera la lista de todos lo elementos contenidos en el proveeedor
        //a las aplicaciones cliente
```

```
28          @GET
29          @Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
30          public List<Todo> getTodos() {
31                    List<Todo> todos = new ArrayList<Todo>();
32                    todos.addAll(TodoDao.INSTANCE.getModel().values());
33                    return todos;
34                  }
35          // Para obtener el numero total de elementos en el proveedor de contenidos
36          @GET
37          @Path("cont")
38          @Produces(MediaType.TEXT_PLAIN)
39          public String getCount() {
40             int cont = TodoDao.INSTANCE.getModel().size();
41             return String.valueOf(cont);
42          }
43          //Para enviar datos al servidor como un formulario Web
44          @POST
45          @Produces(MediaType.TEXT_HTML)
46          @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
47          public void newTodo(@FormParam("id") String id,
48          @FormParam("resumen") String resumen,
49          @FormParam("descripcion") String descripcion,
50          @Context HttpServletResponse servletResponse) throws IOException {
51             Todo todo = new Todo(id, resumen);
52             if (descripcion != null) {
53             todo.setDescripcion(descripcion);
54             }
55          TodoDao.INSTANCE.getModel().put(id, todo);
56          servletResponse.sendRedirect("../crear_todo.html");
57          }
58          //Para poder pasarle argumentos a las operaciones en el servidor
59          // Permite por ejemplo escribir http://localhost:8080/mio.jersey.segundo/rest/todos/1
60          @Path("{todo}")
61          public TodoRecurso getTodo(@PathParam("todo") String id) {
62            return new TodoRecurso(uriInfo, request, id);
63          }
64  }
```

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/mio.jersey.p3/rest/todos` desde un navegador se nos mostrará la siguiente información:

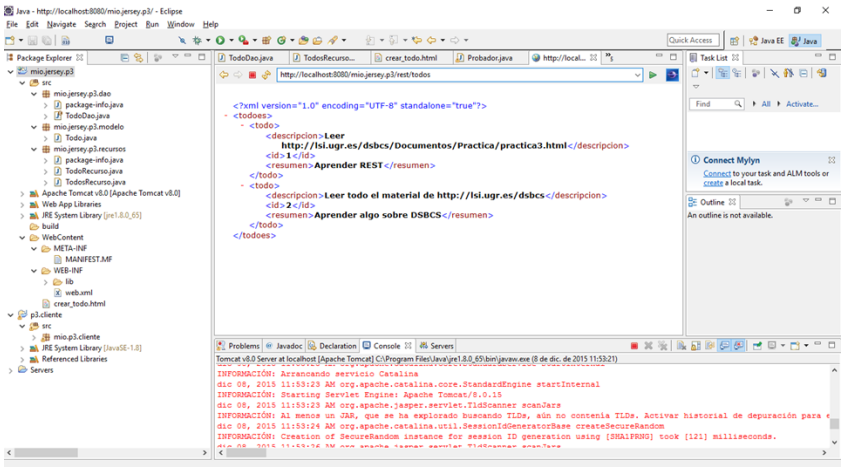


Figura 3: Servicio *REST* de contenidos con operaciones CRUD desplegado correctamente en un servidor Tomcat v8.0

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.test;
2  import java.net.URI;
3  import javax.ws.rs.core.MediaType;
4  import javax.ws.rs.core.UriBuilder;
5  import javax.ws.rs.core.Form;
6  import javax.ws.rs.client.Client;
7  import javax.ws.rs.client.ClientBuilder;
8  import javax.ws.rs.client.Entity;
9  import javax.ws.rs.client.WebTarget;
10 import javax.ws.rs.core.Response;
11 import org.glassfish.jersey.client.ClientConfig;
12 import mio.jersey.segundo.modelo.*;
13
14 public class Probador {
15         public static void main(String[] args) {
16                 ClientConfig config = new ClientConfig();
17             Client cliente = ClientBuilder.newClient(config);
18              WebTarget servicio = cliente.target(getBaseURI());
19             // crearse un tercer "objeto" todo, aparte de los 2 que ya estan creados en TodoDao
20
21             Todo todo = new Todo("3", "Este␣es␣un␣resumen␣de␣otro␣registro");
22         Response respuesta = servicio.path("rest").path("todos").path(todo.getId()).
23                 request(MediaType.APPLICATION_XML).
24                 put(Entity.entity(todo,MediaType.APPLICATION_XML),Response.class);
25
26         // Return code para ver si todo va bien
27         System.out.println(respuesta.getStatus());
28         // Obtener el contenido de Todos
29         System.out.println(servicio.path("rest").path("todos").
30                 request().accept(MediaType.TEXT_XML).get(String.class));
31             // Obtenerlo en formato XML
32         System.out.println(servicio.path("rest").path("todos").
33                 request().accept(MediaType.APPLICATION_XML).get(String.class));
34         // Obtener el objeto Todo con id = 1
35         Response checkDelete = servicio.path("rest").path("todos/1").
36                 request().accept(MediaType.APPLICATION_XML).get();
37         // Eliminar el Todo con id = 1
38         servicio.path("rest").path("todos/1").request().delete();
39         // Volver a obtener todos los objetos 'Todos' pero el id = 1 se ha debido destruir
40         System.out.println(servicio.path("rest").path("todos").
41                  request().accept(MediaType.APPLICATION_XML).get(String.class));
42             // Ahora nos crearemos un recurso Todo utilizando un formulario Web
43             // System.out.println("Creacion de 1 formulario");
44                Form form = new Form();
45                form.param("id", "4");
46                form.param("resumen", "Demostracion␣de␣la␣biblioteca-cliente␣para␣formularios");
47                Response respuesta2 = servicio.path("rest").path("todos").request().
48                post(Entity.entity(form,MediaType.APPLICATION_FORM_URLENCODED),Response.class);
49            System.out.println("Respuesta␣del␣formulario␣" + respuesta2.getStatus());
50         // Mostar el contenido de todos, id = 4 se ha debido de crear
51            System.out.println(servicio.path("rest").path("todos").request().
52            accept(MediaType.APPLICATION_XML).get(String.class));
53     }
54         private static URI getBaseURI() {
55            return UriBuilder.fromUri("http://localhost:8080/mio.jersey.segundo").build();
56         }
57 }
```

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