Desarrollo de ejemplos tutoriales con servicios web y persistencia

1 Introducción

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 distribuido 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.

En este documento se explica cómo desarrollar proyectos con Eclipse ©, que incluyen las librerías adecuadas para obtener SW-*RESTful* y también muestra de manera tutorial la programación de algunas aplicaciones cliente para demostración.

2 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 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.).

3 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:

```
UriBuilder.fromUri("http://localhost:8080/mio.jersey.primero").build();
```

Por otra parte, XML,JSON o HTML son los tipos `MIME` que generalmente podemos utilizar para que un cliente pueda entenderse con un servidor *REST*. La forma de programarlo dentro de un clase Java consiste en crearse un *cliente* configurado (un aplicación Java aparte), que accederá el servicio Web programado anteriormente desde la URI que se ha declarado como su dirección 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 desde nuestra aplicación:

```
- 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 de ls operaciones (`get()`, `put()`, etc.) del conjunto cuya invocación vaya a ser atendida por el servicio que pretendemos desarrollar.

4 Arquitecturas de Java para ligadura XML (JAXB)

La Arquitectura software de Java necesaria para obtener una Ligadura XML (JAXB) es un estándar de Java que define cómo los objetos de Java ("POJOs") 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.

`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[1]

Un ejemplo inicial de aplicación consiste en la implementación de un catálogo de libros 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  @XmlRootElement(name = "libro")
6  // Ahora se va a definir definir el orden en el cual se escriben los campos
7  // 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, se puede cambiar con facilidad
15   // antes de sacarlo hacia la corriente XML de salida:
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 ahora la biblioteca, 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  //La siguiente sentencia significa que la clase "Libreria.java" es el elemento-raiz de
7  //nuestro ejemplo:
8  @XmlRootElement(namespace = "mio.xml.jaxb.modelo")
9  public class Libreria {
10 // XmLElementWrapper genera una entidad-envoltorio alrededor de una representacion XML
11   @XmlElementWrapper(name = "listaLibros")
12 // XmlElement fija el nombre del componente-software "libro" en que se va a convertir este "POJO"
13   @XmlElement(name = "libro")
14   private ArrayList<Libro> listaLibros;
15   private String nombre;
16   private String ubicacion;
17   public void setListaLibros(ArrayList<Libro> listaLibros) {
18     this.listaLibros = listaLibros;
19   }
20   public ArrayList<Libro> getListaDeLibros() {
21     return listaLibros;
22   }
23   public String getNombre() {
24     return nombre;
25   }
26   public void setNombre(String nombre) {
27     this.nombre = nombre;
28   }
29   public String getUbicacion() {
30     return ubicacion;
31   }
32   public void setUbicacion(String ubicacion) {
33     this.ubicacion = ubicacion;
34   }
35 }
```

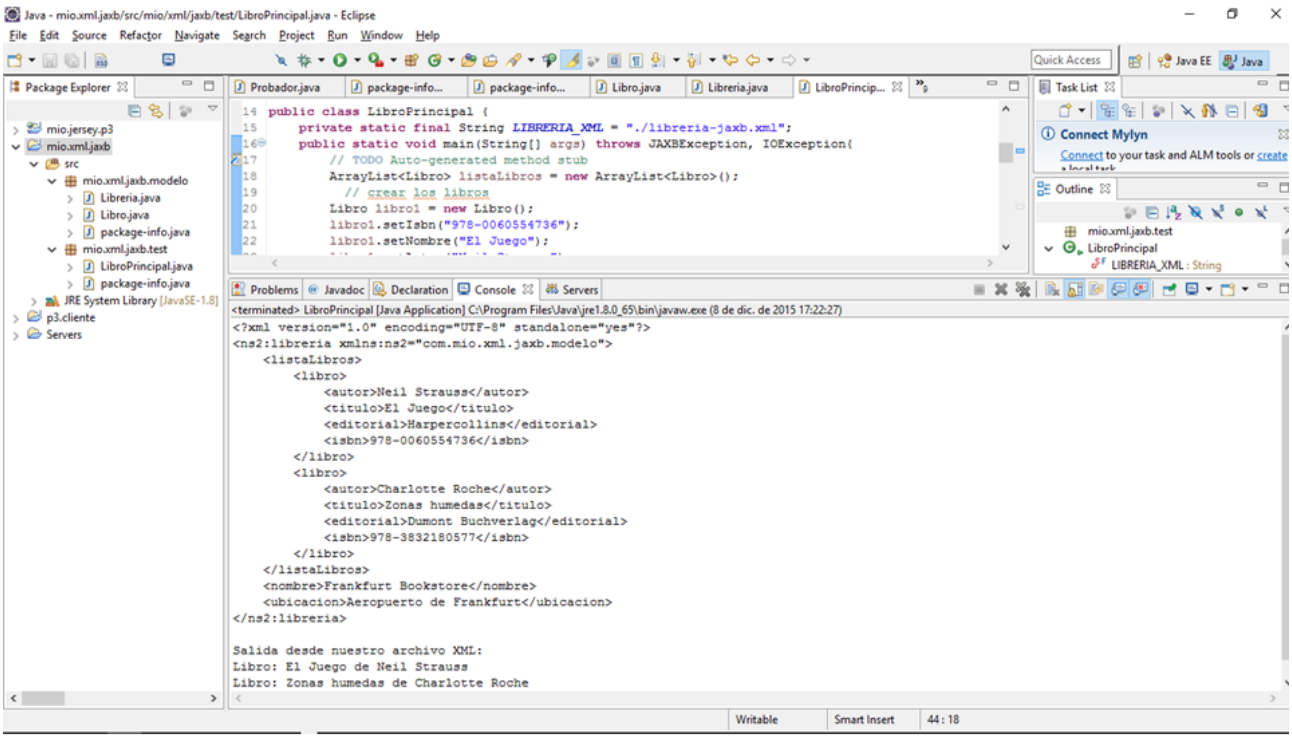


Figura 1: Contenido de Librería mostrado en System.out

Por último, escribimos 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.Libreria;
import mio.xml.jaxb.modelo.Libro;

public class LibroPrincipal {
  private static final String LIBRERIA_XML = "./libreria-jaxb.xml";
  public static void main(String[] args) throws JAXBException, IOException {
    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);
 //otro libro
    Libro libro2 = new Libro();
    libro2.setIsbn("978-3832180577");
    libro2.setNombre("Zonas humedas");
    libro2.setAutor("Charlotte Roche");
    libro2.setEditorial("Dumont Buchverlag");
    listaLibros.add(libro2);
// Ahora: crear la libreria y asignarle los libros
    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 el archivo que hemos ubicado en nuestro directorio: ./libreria-jabx.xml
    m.marshal(libreria, new File(LIBRERIA_XML));
/*****************************************************/
//Ahora vamos a volcar el contenido del archivo xml: libreria-jabx.xml, que
//hemos creado anteriormente
    System.out.println();
    System.out.println("Salida desde el archivo libreria-jabx.xml: ");
    Unmarshaller um = contexto.createUnmarshaller();
//Adapto los bytes que leo del archivo XML al formato de la clase Libreria
    Libreria libreria2 = (Libreria) um.unmarshal(new FileReader(LIBRERIA_XML));
//Ahora puedo obtener la lista de libros en le formato correcto para que
//mi aplicacion pueda imprimir los campos del titulo y del autor de cada libro
//guardado en "listaLisbros"
    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 presentar en pantalla el resultado que muestra la Figura 1. 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>
```

5 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.

5.1 La biblioteca estándar `JAX-RS` de Java

Cuando utilizamos el término *JAX–RS*, además del estándar, nos queremos referir a un conjunto de librerías FOSS[2] 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. Existen varias implementaciones de esta especificación integradas en la plataforma Java (`javax.ws.rs.*`). Para poder utilizarlo en nuestro proyecto Eclipse, sólo hay que incluir las clausulas de importación convenientes en la cabecera: `import javax.ws.rs.GET;`, `import javax.ws.rs.core.UriInfo;`, etc. Esto hace que la programación interna para acceder al *servlet* que internamente soporta a nuestro SW se pueda hacer de forma totalmente transparente para un programador del cliente. Es decir, la programación en el cliente sólo se basa en las anotaciones que hayamos escrito en la clase del SW 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

[2]"Free and Open Source Software"

hayamos creado en nuestro IDE, ya que a través de archivos de configuración (tal como `web.xml`), el servlet 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.

5.2 Despliegue de un SW con Jersey

La implementación considerada de referencia actualmente para la especificación JSR-311 se la conoce con 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.

Progrmando con *Jersey*, para que funcione correctamente el servicio Web 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`) (ver en sección anterior 4).

5.3 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: "`application/xml`", "`application/json`" o "`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.

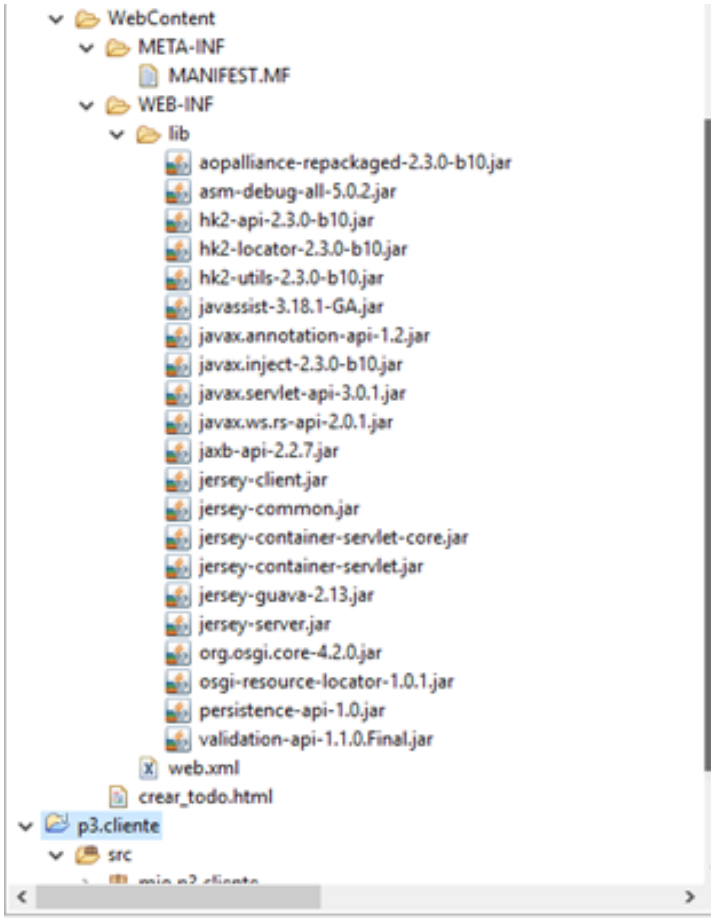


Figura 2: Librerias de *Jersey* necesarias para compilar el proyecto

6 Caso de estudio inicial. Servicio de archivo de resúmenes

Vamos a crearnos un SW *RESTful* con el IDE Eclipse y la tecnología `Jersey`, que ofrecerá a sus clientes la posibilidad de almacenar descripciones y resúmenes en formato XML. Se almacenan en un archivo con formato XML, que podría utilizarse para mantener una base de datos básicos sobre películas, música, videos, etc.

A continuación se describen los pasos que hay que dar para llevar a cabo el proceso de desarrollo y despliegue de un SW correctamente con esta tecnología. Los ejemplos que se van a presentar se han programado con Eclipse Java EE IDE for Web Developers. Version: Mars.1 Release (4.5.1) y la máquina virtual: Java HotSpot(TM) 64-Bit Server VM.

6.1 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` (Figura 2).

6.2 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 soporta una correspondencia automatica desde una clase JAXB
5  //con anotaciones a XML y JSON
6  public class Todo {
7    private String resumen;
8    private String descripcion;
9    public String getResumen() {
10     return resumen;
11   }
12   public void setResumen(String resumen) {
13     this.resumen = resumen;
14   }
15   public String getDescripcion() {
16     return descripcion;
17   }
18   public void setDescripcion(String descripcion) {
19     this.descripcion = descripcion;
20   }
21 }
```

6.3 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 javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
//Esta clase solo devuelve una instancia de la clase Todo
@Path("/todo")
public class TodoRecurso {
//Este metodo se ejecutara si existe una peticion XML desde el cliente
  @GET
  @Produces({ MediaType.APPLICATION_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;
  }
//Lo que sigue se puede utilizar para comprobar la integracion con el navegador
//que utilicemos
  @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;
  }
}
```

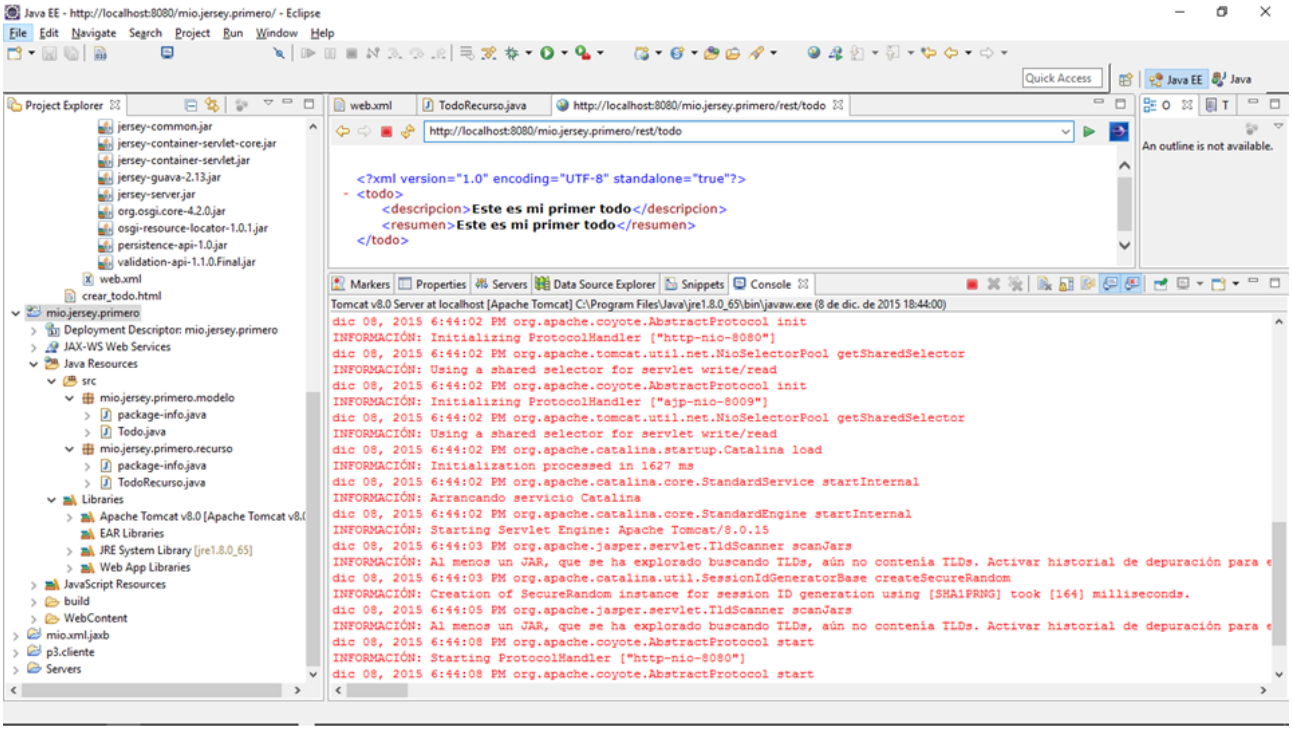
6.3.1 Archivo de descripción de despliegue

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
<?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:sch
  <display-name>Servidor de Contenidos REST</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
  <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</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>
```

6.4 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:



6.4.1 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 añadir los `*.jar` de Jersey e incluirlos en el "*build path*" del nuevo proyecto, ya que si no se incluyen las librerías adecuadas, no funcionará. 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  public class Test {
10   public static void main(String[] args) {
11         // TODO Auto-generated method stub
12         ClientConfig config = new DefaultClientConfig();
13         Client cliente = Client.create(config);
14         WebResource servicio = cliente.resource(getBaseURI());
15         // Conseguir XML        System.out.println(servicio.path("rest").path("todo").accept(MediaType.TEXT_XML).
16         // Conseguir XML para la aplicacion     System.out.println(servicio.path("rest").path("todo").accept(Medi
17         }
18   private static URI getBaseURI(){
19         return UriBuilder.fromUri("http://localhost:8080/mio.jersey.primero").build();
20   }
21 }
```

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

```
Mostrar contenido del recurso como Texto XML Plano
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<todo>
   <descripcion>Este es mi primer todo</descripcion>
   <resumen>Este es mi primer todo</resumen>
</todo>
Mostrar contenido del recurso para aplicacion XML
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<todo>
  <descripcion>Este es mi primer todo</descripcion>
  <resumen>Este es mi primer todo</resumen>
</todo>
```

Créditos

- Package javax.persistence: http://docs.oracle.com/javaee/7/api/javax/persistence/package-summary.html
- Jersey 1.17 User Guide
  https://jersey.java.net/nonav/documentation/1.17/user-guide.html
- 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