UNIVERSIDAD DE OVIEDO

GIITIN – Tecnologías Web

Sesión 11 de Laboratorio - Curso 2013 / 2014

Sesión 11. WebServices (WS-REST)

Índice

Introducción

Las arquitecturas orientadas a servicios (SOA) no son nuevas y existen desde hace ya bastante tiempo. Surgen por la necesidad de las aplicaciones de comunicarse entre ellas para intercambiar información y colaborar. Dada la heterogeneidad de las mismas aplicaciones en el lenguaje de programación o de representación de datos requieren un lenguaje común con el que intercambiar información, para ello se puede emplear un lenguaje de marcas como XML y para el intercambio de mensajes SOAP (Simple Object Access Protocol) sobre el protocolo HTTP. Otras opciones como CORBA los mensajes se transmitían de forma binaria. Pero los servicios web basados en XML y SOAP no están exentos de problemas, los mensajes SOAP son bastante complejos y para algunas necesidades no son fáciles de construir o cómodas de usar, por ejemplo desde un navegador o un dispositivo móvil empleando JavaScript.

Dado el auge u omnipresencia que están tomando los servicios basados en Internet, la web, los navegadores y JavaScript, muchos de los sitios más importantes están evolucionando hacia servicios REST por su simplicidad de ser consumidos y fácil acceso, y muchos sitios ofrecen una API basada en REST para que terceros puedan integrase con ellos, este es el principio para poder construir mashups. Algunas de sus características son que emplean el protocolo HTTP y sus diferentes métodos (GET, POST, DELETE y PUT, etc.) asignando a cada uno de ellos una operación de las denominadas CRUD (create -> PUT/POST, retrieve -> GET, update -> PUT/POST, delete -> DELETE) y pueden devolver datos en cualquier formato pero normalmente se emplea JSON o XML.

En esta práctica partiremos de la versión 9.1 de Gestioneitor que incorporaba el control de seguridad y añadiremos acceso por Web Services usando REST. Esto va a permitir que un programa cliente, escrito para cualquier plataforma y lenguaje, pueda usar de forma remota los servicios de nuestra aplicación simplemente usando un cliente JavaScript o bien directamente el propio navegador para el caso de los servicios GET.

Los pasos que seguiremos para hacer la evolución serán los siguientes:

- Evolución de Gestioneitor v.9.1 a v.9.3 diseñando el acceso por REST.
- Desarrollo de un cliente Javascript para acceso a WS por REST.
- Despliegue y pruebas.

Pasos generales para crear un servicio REST

Para crear un servicio REST se deben seguir los siguientes pasos generales:

- Crear un interface fachada para el servicio que vamos a implementar y en el que anotaremos con notación JAX-RS los diferentes métodos (*AlumnosServicesRs*).
- Anotar los métodos getter de las clases DTO del paquete "com.tew.model" que se van a emplear como parámetros/retorno de los métodos del servicio (*Alumno*).
- Crear una clase servicio que implementará el interfaz fachada *AlumnosServicesRs* definido en el primer paso (*AlumnosServicesRsImpl*).
- Registrar en web.xml el controlador RestEasy asociado al URL pattern "/rest"

- Finalmente crear una clase denominada "*Application*" en la que registraremos las clases de implementación de los servicios que deseemos. En este caso sólo registraremos la clase "*AlumnosServicesRsImpl*".

Desarrollo de la práctica

1. Crear un proyecto nuevo JSF vacío en eclipse seleccionando JSF 2.2 como entorno de ejecución y con nombre tew-gestioneitorv9_1.
2. Descomprimir el zip suministrado tew-gestioneitorv9_1.zip y copiar todas las carpetas a la raíz del proyecto JSF creado en work\tew-gestioneitorv9_1.

Creación del servicio Web

3. Crear un nuevo paquete **com.tew.business.resteasy**[1] que es donde ubicaremos las nuevas clases.
4. Vamos a crear una fachada anotada para nuestros servicios REST. Denominaremos a esta clase **com.tew.business.resteasy.AlumnosServicesRs(AlumnosServicesRs.java)**:

```
package com.tew.business.resteasy;

import java.util.List;
import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;

import com.tew.business.AlumnosService;
import com.tew.business.exception.EntityAlreadyExistsException;
import com.tew.business.exception.EntityNotFoundException;
import com.tew.model.Alumno;

@Path("/AlumnosServicesRs")
public interface AlumnosServicesRs extends AlumnosService{
   @GET
   @Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
   public List<Alumno> getAlumnos();

   @GET
   @Path("{id}")
   @Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
   Alumno findById(@PathParam("id") Long id) throws EntityNotFoundException;

   @DELETE
   @Path("{id}")
   void deleteAlumno(@PathParam("id") Long id) throws EntityNotFoundException;

   @PUT
   @Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
   void saveAlumno(Alumno alumno) throws EntityAlreadyExistsException;

   @POST
   @Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
   void updateAlumno(Alumno alumno) throws EntityNotFoundException;
}
```

Hemos empleado las anotaciones @Path, @GET, @PUT, @POST, @DELETE y @PathParam. Con @Path estamos indicando parte de la URL en la que el web service

[1] Existe a día de hoy dos implementaciones del standard REST, que son la implementación de referencia de Oracle nombre en clave "Jersey" y la implementación de JBoss denominada "RESTeasy". Dado que estamos empleando JBoss como contenedor emplearemos la implementación RESTeasy.

responderá, con @GET indicamos que el método HTTP que llame a esa URL deberá ser GET y con @PathParam recogemos un parámetro indicado en la URL. Además en este interfaz como manejamos datos estructurados (clase Alumno) se debe indicar el formato en que esos datos se reciben y se retornan (@Consumes y @Produces).

5. Debemos anotar los tipos de datos estructurados que vayamos a emplear en los métodos anotados. Para ello modificaremos nuestra clase **com.tew.modelo.Alumno** añadiendo aquellas partes que **aparecen en azul** en el bloque de código que se indica continuación**:**

```
package com.tew.model;

import java.io.Serializable;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "alumno")
public class Alumno  implements Serializable{
        ….
        public Alumno(Long id, String nombre, String apellidos, String iduser, String email) {
          this.id = id; this.nombre = nombre; this.apellidos = apellidos;
          this.iduser = iduser; this.email = email;
        }
        @XmlElement
        public String getNombre() {
                return nombre;
        }
        ….
        @XmlElement
        public String getApellidos() {
                return apellidos;
        }

        @XmlElement
        public String getIduser() {
                return iduser;
        }
        ….
        @XmlElement
        public String getEmail() {
                return email;
        }
        ….

        @XmlElement
        public Long getId() {
                return id;
        }
        …
}
```

Hemos anotado la propia clase como `@XmlRootElement(name = "alumno")` y todos los métodos de lectura (get*) como `@XmlElement`. El propio controlador de Resteasy se encarga de traducir este formato a XML o JSON o el formato que se indique en la notación de los métodos (@Produces/@Consumes) en tiempo de ejecución y en función del formato que indique el cliente que invoca al servicio.

6. Deberás incorporar aquellos "import" que te vaya sugiriendo Eclipse.

7. A continuación vamos a incluir la implementación del interfaz de los servicios con la clase **impl.tew.business.resteasy.AlumnosServicesRsImpl (AlumnosServicesRsImpl.java**).

```
package impl.tew.business.resteasy;

import java.util.List;

import com.tew.business.resteasy.AlumnosServicesRs;
import com.tew.business.exception.EntityAlreadyExistsException;
```

```
import com.tew.business.exception.EntityNotFoundException;
import com.tew.model.Alumno;
import impl.tew.business.classes.*;

public class AlumnosServicesRsImpl implements AlumnosServicesRs {

	@Override
	public List<Alumno> getAlumnos() {
		return new AlumnosListado().getAlumnos();
	}

	@Override
	public void saveAlumno(Alumno alumno) throws EntityAlreadyExistsException {
		new AlumnosAlta().save(alumno);
	}

	@Override
	public void updateAlumno(Alumno alumno) throws EntityNotFoundException {
		new AlumnosUpdate().update(alumno);
	}

	@Override
	public void deleteAlumno(Long id) throws EntityNotFoundException {
		new AlumnosBaja().delete(id);
	}

	@Override
	public Alumno findById(Long id) throws EntityNotFoundException {
		return new AlumnosBuscar().find(id);
	}
}
```

8. Verás que todavía queda un problema que resolver al respecto a la excepción del método getAlumnos(). Eleva la excepción oportunamente tanto en esta clase como el método prototipo del interfaz..

9. Una vez que disponemos del web service debemos añadir al archivo web.xml los elementos necesarios para que RESTEasy recoja las llamadas a las URLs de los servicios web y los invoque, entre ellas el servlet de nombre resteasy de la clase HttpServletDispatcher. Debes añadir lo siguiente a web.xml:

```
….
 <!-- CONFIGURACION DE RESTEASY  -->
       <!-- Es necesario incluir esta variable de contexto para que
       Resteasy reconozca el mapeo /rest/ correspondiente a servicios REST.
       Además de incluir el propio <mapping> y el servelt Resteasy -->
       <context-param>
        <param-name>resteasy.servlet.mapping.prefix</param-name>
        <param-value>/rest</param-value>
      </context-param>

      <!-- Servlet resteasy que se encarga de procesar las peticiones  correspondientes a
servicios REST. Lleva un parámetro de inicialización donde deben estar registradas todas las
clases que implementan servicios REST en esta aplicación web -->
    <servlet>
        <servlet-name>resteasy</servlet-name>
        <servlet-class>
               org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
        </servlet-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>com.tew.business.resteasy.Application</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <!-- Regla de mapeo para todas las peticiones REST -->
    <servlet-mapping>
        <servlet-name>resteasy</servlet-name>
        <url-pattern>/rest/*</url-pattern>
    </servlet-mapping>
```

...

Lo básico a añadir es el servlet de RESTEasy y el correspondiente servlet-mapping, también deberemos indicar el parámetro de contexto "resteasy.servlet.mapping.prefix" con el prefijo de la ruta del servlet-mapping del servlet de RESTEasy. Con esta configuración todas las URLs que lleguen a la aplicación web y con la ruta «/rest/» corresponderán a un web service REST. Una cosa a destacar es que los servicios web pueden convivir con el resto de la aplicación independientemente del framework que utilicemos para desarrollarla. Lo único que deberemos hacer es indicar un prefijo para el servlet de RESTEasy tal y como hemos hecho en el ejemplo.

10. ¿Cómo sabe RESTEasy cuáles son los servicios web de la aplicación? La forma de indicarle a RESTEasy los servicios web es mediante una clase que extiende de javax.ws.rs.core.Application y utiliza el parámetro de inicialización javax.ws.rs.Application para el servlet de RESTEasy que se puede ver en el web.xml. La implementación no tiene más que un Set con las implementaciones de los servicios web. Crear la clase **com.tew.business.resteasy.Application (Application.java)**:

```
//Registro de clases punto final.
//Es una forma de registro estático de clases punto final (servicios rest)

package com.tew.business.resteasy;

import java.util.Collections;
import java.util.HashSet;
import java.util.Set;

@SuppressWarnings("unchecked")
public class Application extends javax.ws.rs.core.Application {

	private Set<Class<?>> classes = new HashSet<Class<?>>();

	public Application() {
		classes.add(AlumnosServicesRsImpl.class);

	}

	@Override
	public Set<Class<?>> getClasses() {
		return classes;
	}

	@Override
	public Set<Object> getSingletons() {
		return Collections.EMPTY_SET;
	}
}
```

11. Recuerda importar las clases que te indique Eclipse.
12. Ahora ya podemos compilar y desplegar el proyecto. Observa la consola de JBoss, que no se produzcan errores.
13. Para consultar los servicios basados en el método http GET simplemente indicando la URL del servicio en el navegador:

    /rest/AlumnosServicesRs Acceso al servicio web AlumnosServicesRs con el método GET. Se corresponde con el método "getAlumnos".

    /rest/AlumnosServiceRs/1 Acceso al servicio web AlumnosServicesRs con el método GET pero con un parámetro numérico (1). Se invoca al método "findById" con el parámetro 1. Prueba con diferentes parámetros, tanto de Ids de alumnos que no existen como de alumnos que existan.

Para probar el resto de métodos HTTP de este servicio deberemos emplear clientes ad-hoc.

Cliente Javascript

14. RESTEasy proporciona un servlet que genera como salida el código de un librería Javascript de acceso a nuestros servicios web REST. Para hacer uso de este script debemos hacer dos cosas:

    a. Configurar este servlet incluyendolo en el archivo web.xml de nuestra aplicación para que sea generado el servlet.

    b. Para usarlo en una página html donde se quiera utilizar, se debe incluir el url-pattern de este servlet como librería Javascript.

    A continuación incluye en web.xml el siguiente fragmento de código al final del archivo.

```
<!--Generación de la librería Javascript para acceso a servicios Web REsteasy-->
<!-- Servlet que atenderá las consultas de clientes Javascript
        Tiene que ser cargado después del servlet restasy.
    -->
    <servlet>
        <servlet-name>resteasy-jsapi</servlet-name>
        <servlet-class>org.jboss.resteasy.jsapi.JSAPIServlet</servlet-class>
        <load-on-startup>2</load-on-startup>
    </servlet>
<!-- Regla de mapeo para la libreria Javascript -->
    <servlet-mapping>
        <servlet-name>resteasy-jsapi</servlet-name>
        <url-pattern>/rest-jsapi</url-pattern>
    </servlet-mapping>
```

    Es importante tener en cuenta que este servlet debe cargarse después del servlet resteasy que es quien soporta los servicios y por ello se indica para el servlet resteasy-jsapi el orden de carga 2.

15. En lo que respecta al código del cliente, debemos incluir la librería jsapi (*/tew-gestioneitorv9_1/rest-jsapi*) generada automáticamente por RESTeasy para el cliente y hacer referencia a los método remotos de una forma transparente (por debajo se trata de peticiones Ajax).

    Aunque en los clientes, estas operaciones son totalmente transparentes para nosotros, en cada cliente JavaScript tendremos disponibles unos métodos que se encargarán de realizar de la forma adecuada las peticiones así como el un/marshalling de los parámetros de los métodos.

    Crea un cliente html con el siguiente código:

```
<html>
<head>
  <title>Ejemplo sencillo de web service con RESTEasy</title>
  <script type="text/javascript"
          src="http://localhost:8080/tew-gestioneitorv9_1/rest-jsapi">
  </script>

  <script type="text/javascript">
    function solicitarDatosAlumnos() {
      var alumnos = AlumnosServicesRs.getAlumnos({"username": 'admin', "password":
'password'});
      // Obtener el elemento HTML donde ir dejando el resultado
      // del servicio
      div = document.getElementById("contenido")
      for (i in alumnos) {
        div.innerHTML += alumnos[i]["nombre"] + alumnos[i]["apellidos"] + alumnos[i]["email"]
+ alumnos[i]["iduser"]+"</br>";
      }
```

```
    }
  </script>
</head>

<body>
  <h1>Listado de alumnos</h1>
  <div>
    Obtener lista de alumnos
    <INPUT TYPE="button" NAME="button" Value="Pulsa" onClick="solicitarDatosAlumnos()">
  </div>
  <div id="contenido" style="height:90%;top:30px">
    Resultado del servicio  REST...</br>
  </div>
</body>
</html>
```

En este cliente hemos definido un elemento div (contenido) donde se dejarán los resultados obtenidos a partir del servicio (OJO con las comillas dobles al copiar y pegar). La función solicitarDatosAlumnos es la encargada de invocar al servicio. En ella definimos las credenciales para poder acceder al servicio.

16. A modo de prueba, prueba a llamar desde el navegador directamente a la librería http://localhost:8080/tew-gestioneitorv9_1/rest-jsapi y analiza en ella AlumnosServicesRs.getAlumnos. Revisa cómo está implementado el sistema de paso de credenciales así como toda la fachada del servicio en Javascript.
17. Prueba a cargar el cliente html desde el disco file://... y ejecutarlo. ¿Qué ocurre? Traza la ejecución con la Herramienta de desarrolladores de Chrome/Firefox. (Herramientas/Herramientas para desarrolladores).

    Lo que está ocurriendo es que ni Chrome ni la mayoría de navegadores permiten ejecutar servicios desde una página cargada localmente (no de un servidor).
18. Sube el cliente Javascript a Jboss y pruébalo. Debes tener en cuenta que esto funcionará correctamente siempre que el cliente esté en el mismo servidor (dominio) que el servicio al que accede. En caso contrario el navegador va a impedir la recepción de la respuesta del servicio. Para evitar esto debes enviar la cabecera desde el servidor antes de prestar el servicio:

    ```
    Access-Control-Allow-Origin: *
    ```

    Esta indicación es a título informativo ya que no es algo trivial de hacer en RESTEasy. Y lo dejamos pendiente como trabajo extra.

Cliente Java (consumo de servicios REST externos)

Para consumir un servicio REST de un tercero vamos a emplear la clase ClientRequest, la cual nos permite obtener el contenido de un recurso publicado vía HTTP.

Los pasos que vamos a seguir para crear un cliente Java de servicios REST son los siguientes:

1. Crear un proyecto de tipo cliente para aplicación empresarial (New/File/Application Client project).
2. Incluir en el "Build Path" del proyecto los jar siguientes que te suministramos:
   a. Todos los jar incluídos en lib/ApacheHttp.zip.
   b. El lib/json_simple.jar.

3. Obtener el URI del servicio que queremos emplear. En este caso será el servicio que vamos a usar para la práctica 02 de la asignatura: http://di002.edv.uniovi.es/~delacal/tew/practica02/servicio.php.

4. A continuación vamos a incluir en el método main() de la clase la invocación al servicio empleando el siguiente código:

```
ClientRequest cr = new
ClientRequest("http://di002.edv.uniovi.es/~delacal/tew/practica02/servicio.php");
String result = cr.get(String.class).getEntity(String.class);
System.out.println(result);
```

5. No olvides incluir los import que el propio Eclipse te indicará.

Fíjate que una vez obtener el objeto ClientRequest conectado con el servicio, tenemos que realizar la invocación del método HTTP que deseamos enviar al URI del servicio. En este caso emplearemos el método get() y como parámetro le hemos pasado la clase en que queremos obtener el resultado. De esta format obtendremos el la salida del servicio en formato cadena JSON. Aunque también es posible parsear el resultado del servicio por uno mismo, será más comodo hacer uso de la librería SimpleJson para que nos parsee los objetos Json a formato Java.

6. A continuación se incluye el código ampliado en que accedemos mediante simplejson a nuestro servicio. Fíjate que el acceso a los atributos se realiza mediante la propiedad @attributes.

```
import org.jboss.resteasy.client.ClientRequest;
import org.json.simple.*;

/*El código para acceder al servicio y luego tratar los datos.
Obviamente, no se debería hacer así, esto es sólo una muestra de cómo acceder.
Los objetos de las clases JSONObjects (mapas) y JSONArray (arrays)
*/

public class Main {
  public static void main(String[] args) throws Exception {
    ClientRequest cr = new
ClientRequest("http://di002.edv.uniovi.es/~delacal/tew/practica02/servicio.php");
    String result = cr.get(String.class).getEntity(String.class);

    System.out.println(result);

    Object obj=JSONValue.parse(result);
    JSONObject p=(JSONObject)obj;
    JSONArray movies = (JSONArray) p.get("Movie");

    System.out.println(movies.get(2));


System.out.println(((JSONObject)((JSONObject)movies.get(1)).get("@attributes")).get("Pictu
re"));

System.out.println(((JSONObject)((JSONObject)movies.get(1)).get("@attributes")).get("Comme
nts"));
  }

}
```

7. Te sugerimos analizar y probar el servicio web publico del ayuntamiento de gijón sobre datos de las líneas de autobuses públicos (http://datos.gijon.es/doc/transporte/busgijontr.json).