Introducción a los Servicios Web RESTful

Índice

El estilo REST (*Representational State Transfer*) es una forma ligera de crear Servicios Web. El elemento principal en el que se basan estos servicios son las URLs. En líneas generales podemos decir que estos servicios consisten en URLs a las que podemos acceder, por ejemplo mediante protocolo HTTP, para obtener información o realizar alguna operación. El formato de la información que se intercambie con estas URLs lo decidirá el desarrollador del servicio. Este tipo de servicios acercan los Servicios Web al tipo de arquitectura de la *web*, siendo especialmente interesantes para su utilización en AJAX.

1. Fundamentos de REST

El término REST proviene de la tesis doctoral de Roy Fielding, publicada en el año 2000, y significa ***REpresentational State Transfer***. REST es un conjunto de restricciones que, cuando son aplicadas al diseño de un sistema, crean un estilo arquitectónico de software. Dicho estilo arquitectónico se caracteriza por:

- Debe ser un sistema cliente-servidor
- Tiene que ser sin estado, es decir, no hay necesidad de que los servicios guarden las sesiones de los usuarios (cada petición al servicio tiene que ser independiente de las demás)
- Debe soportar un sistema de *cachés*: la infraestructura de la red debería soportar *caché* en diferentes niveles
- Debe ser un sistema uniformemente accesible (con una interfaz uniforme): cada recurso debe tener una única dirección y un punto válido de acceso. Los recursos se identifican con URIs, lo cual proporciona un espacio de direccionamiento global para el descubrimiento del servicio y de los recursos.
- Tiene que ser un sistema por capas: por lo tanto debe soportar escalabilidad
- Debe utilizar mensajes auto-descriptivos: los recursos se desacoplan de su representación de forma que puedan ser accedidos en una variedad de formatos, como por ejemplo XML, HTML, texto plano, PDF, JPEG, JSON, etc.

Estas restricciones no dictan qué tipo de tecnología utilizar; solamente definen cómo se transfieren los datos entre componentes y qué beneficios se obtienen siguiendo estas restricciones. Por lo tanto, un sistema RESTful puede implementarse en cualquier arquitectura de la red disponible. Y lo que es más importante, no es necesario "inventar" nuevas tecnologías o protocolos de red: podemos utilizar las infraestructuras de red existentes, tales como la Web, para crear arquitecturas RESTful.

Antes de que las restricciones REST fuesen formalizadas, ya disponíamos de un ejemplo de un sistema RESTful: la Web (estática). Por ejemplo, la infraestructura de red existente proporciona sistemas de *caché*, conexión sin estado, y enlaces únicos a los recursos, en donde los recursos son todos los documentos disponibles en cada sitio web y las representaciones de dichos recursos son conjuntos de ficheros "legibles" por navegadores web (por ejemplo, ficheros HTML). Por lo tanto, la web estática es un sistema construido

sobre un estilo arquitectónico REST.

A continuación analizaremos las abstracciones que constituyen un sistema RESTful: recursos, representaciones, URIs, y los tipos de peticiones HTTP que constituyen la interfaz uniforme utilizada en las transferencias cliente/servidor

1.1. Recursos

Un recurso REST es cualquier cosa que sea direccionable a través de la Web. Por direccionable nos refererimos a recursos que puedan ser accedidos y transferidos entre clientes y servidores. Por lo tanto, un recurso es una correspondencia lógica y temporal con un concepto en el dominio del problema para el cual estamos implementando una solución.

Algunos ejemplos de recursos REST son:

- Una noticia de un periódico
- La temperatura de Alicante a las 4:00pm
- Un valor de IVA almacenado en una base de datos
- Una lista con el historial de las revisiones de código en un sistema CVS
- Un estudiante en alguna aula de alguna universidad
- El resultado de una búsqueda de un ítem particular en Google

Aun cuando el mapeado de un recurso es único, diferentes peticiones a un recurso pueden devolver la misma representación binaria almacenada en el servidor. Por ejemplo, consideremos un recurso en el contexto de un sistema de publicaciones. En este caso, una petición de la "última revisión publicada" y la petición de "la revisión número 12" en algún momento de tiempo pueden devolver la misma representación del recurso: cuando la última revisión sea efectivamente la 12. Por lo tanto, cuando la última revisión publicada se incremente a la versión 13, una petición a la última revisión devolverá la versión 13, y una petición de la revisión 12, continuará devolviendo la versión 12. En definitiva: cada uno de los recursos puede ser accedido directamente y de forma independiente, pero diferentes peticiones podrían "apuntar" al mismo dato.

Debido a que estamos utilizando HTTP para comunicarnos, podemos transferir cualquier tipo de información que pueda transportarse entre clientes y servidores. Por ejemplo, si realizamos una petición de un fichero de texto de la CNN, nuestro navegador mostrará un fichero de texto. Si solicitamos una película flash a YouTube, nuestro navegador recibirá una película flash. En ambos casos, los datos son transferidos sobre TCP/IP y el navegador conoce cómo interpretar los *streams* binarios debido a la cabecera de respuesta del protocolo HTTP *Content-Type*. Por lo tanto, en un sistema RESTful, la representación de un recurso depende del tipo deseado por el cliente (tipo MIME), el cual está especificado en la petición del protocolo de comunicaciones.

1.2. Representación

La representación de los recursos es lo que se envía entre los servidores y clientes. Una representación muestra el estado del dato real almacenado en algún dispositivo de almacenamiento en el momento de la petición. En términos generales, es un *stream* binario, juntamente con los metadatos que describen cómo dicho *stream* debe ser consumido por el cliente y/o servidor (los metadatos también puden contener información extra sobre el recurso, como por ejemplo información de validación y encriptación, o código extra para ser ejecutado dinámicamente).

A través del ciclo de vida de un servicio web, pueden haber varios clientes solicitando recursos. Clientes diferentes son capaces de consumir diferentes representaciones del mismo recurso. Por lo tanto, una representación puede tener varias formas, como por ejemplo, una imagen, un texto, un fichero XML, o un fichero JSON, pero tienen que estar disponibles en la misma URL.

Para respuestas generadas para humanos a través de un navegador, una representación típica tiene la forma de página HTML. Para respuestas automáticas de otros servicios web, la legibilidad no es importante y puede utilizarse una representación mucho más eficiente como por ejemplo XML.

El lenguaje para el intercambio de información con el servicio queda a elección del desarrollador. A continuación mostramos algunos formatos comunes que podemos utilizar para intercambiar esta información:

Formato	Tipo MIME
Texto plano	`text/plain`
HTML	`text/html`
XML	`application/xml`
JSON	`application/json`

De especial interés es el formato JSON. Se trata de un lenguaje ligero de intercambio de información, que puede utilizarse en lugar de XML (que resulta considerablemente más pesado) para aplicaciones AJAX. De hecho, en Javascript puede leerse este tipo de formato simplemente utilizando el método `eval()`.

1.3. URI

Una URI, o **Uniform Resource Identifier**, en un servicio web RESTful es un hiper-enlace a un recurso, y es la única forma de intercambiar representaciones entre clientes y servidores. Un servicio web RESTful expone un conjunto de recursos que identifican los objetivos de la interacción con sus clientes.

El conjunto de restricciones REST no impone que las URIs deban ser hiper-enlaces. Simplemente hablamos de hiper-enlaces porque estamos utilizando la Web para crear servicios web. Si estuviésemos utilizando un conjunto diferente de tecnologías

soportadas, una URI RESTful podría ser algo completamente diferente. Sin embargo, la idea de direccionabilidad debe permanecer.

En un sistema REST, la URI no cambia a lo largo del tiempo, ya que la implementación de la arquitectura es la que gestiona los servicios, localiza los recursos, negocia las representaciones, y envía respuestas con los recursos solicitados. Y lo que es más importante, si hubiese un cambio en la estructura del dispositivo de almacenamiento en el lado del servidor (por ejemplo, un cambio de servidores de bases de datos), nuestras URIs seguirán siendo las mismas y serán válidas mientras el servicio web siga estando "en marcha" o el contexto del recurso no cambie.

Sin las restricciones REST, los recursos se acceden por su localización: las direcciones web típicas son URIs fijas. Si por ejemplo renombramos un fichero en el servidor, la URI será diferente; si movemos el fichero a un directorio diferente, la URI también será diferente.

Por ejemplo, si en nuestra aplicación tenemos información de cursos, podríamos acceder a la lista de cursos disponibles mediante una URL como la siguiente:

```
http://jtech.ua.es/resources/cursos
```

Esto nos devolverá la lista de cursos en el formato que el desarrollador del servicio haya decidido. Hay que destacar por lo tanto que en este caso debe haber un entendimiento entre el consumidor y el productor del servicio, de forma que el primero comprenda el lenguaje utilizado por el segundo.

Esta URL nos podría devolver un documento como el siguiente:

```
<?xml version="1.0"?>
<j:Cursos xmlns:j="http://www.jtech.ua.es"
          xmlns:xlink="http://www.w3.org/1999/xlink">
  <Curso id="1"
         xlink:href="http://jtech.ua.es/resources/cursos/1"/>
  <Curso id="2"
         xlink:href="http://jtech.ua.es/resources/cursos/2"/>
  <Curso id="4"
         xlink:href="http://jtech.ua.es/resources/cursos/4"/>
  <Curso id="6"
         xlink:href="http://jtech.ua.es/resources/cursos/6"/>
</j:Cursos>
```

En este documento se muestra la lista de cursos registrados en la aplicación, cada uno de ellos representado también por una URL. Accediendo a estas URLs podremos obtener información sobre cada curso concreto o bien modificarlo.

1.4. Uniformidad de las interfaces a través de peticiones HTTP

Ya hemos introducido los conceptos de recursos y sus representaciones. Hemos dicho que los recursos son *mappings* de los estados reales de las entidades que son intercambiados entre los clientes y servidores. También hemos dicho que las representaciones son negociadas entre los clientes y servidores a través del protocolo de comunicación en

tiempo de ejecución (a través de HTTP). A continuación veremos con detalle lo que significa el intercambio de estas representaciones, y lo que implica para los clientes y servidores el realizar acciones sobre dichos recursos.

El desarrollo de servicios web REST es similar al desarrollo de aplicaciones web. Sin embargo, la diferencia fundamental entre el desarrollo de aplicaciones web tradicionales y las más modernas es cómo pensamos sobre las acciones a realizar sobre nuestras abstracciones de datos. De forma más concreta, el desarrollo moderno está centrado en el concepto de **nombres** (intercambio de recursos); el desarrollo tradicional está centrado en el concepto de verbos (acciones remotas a realizar sobre los datos). Con la primera forma, estamos implementando un servicio web RESTful; con la segunda un servicio similar a una llamada a procedimiento remoto- RPC). Y lo que es más, un servicio RESTful modifica el estado de los datos a través de la representación de los recursos (por el contrario, una llamada a un servicio RPC, oculta la representación de los datos y en su lugar envía comandos para modificar el estado de los datos en el lado del servidor). Finalmente, en el desarrollo moderno de aplicaciones web limitamos la ambigüedad en el diseño y la implementación debido a que tenemos cuatro acciones específicas que podemos realizar sobre los recursos: *Create, Retrieve, Update, Delete (CRUD)*. Por otro lado, en el desarrollo tradicional de aplicaciones web, podemos tener otras acciones con nombres o implementaciones no estándar.

A continuación mostramos la correspondencia entre las acciones CRUD sobre los datos y los métodos HTTP correspondientes:

Acción sobre los datos	Protocolo HTTP equivalente
CREATE	POST
RETRIEVE	GET
UPDATE	PUT
DELETE	DELETE

En su forma más simple, los servicios web RESTful son aplicaciones cliente-servidor a través de la red que manipulan el estado de los recursos. En este contexto, la manipulación de los recursos significa creación de recursos, recuperación, modificación y borrado. Sin embargo, los servicios web RESTful no están limitados solamente a estos cuatro conceptos básicos de manipulación de datos. Por el contrario, los servicios RESTful pueden ejecutar lógica en al lado del servidor, pero recordando que cada respuesta debe ser una representación del recurso del dominio en cuestión. Deberemos determinar que operación HTTP se ajusta mejor a la manipulación que deseamos realizar sobre los datos. Mención especial merece el método PUT, ya que no se trata simplemente de una actualización de los datos, sino de establecer el estado del recurso, exista previamente o no. A continuación trataremos cada uno de estos métodos con más detalle.

Nota

Una interfaz uniforme centra la atención en los conceptos abstractos que hemos visto: recursos,

representaciones y URIs. Por lo tanto, si consideramos todos estos conceptos en su conjunto, podemos describir el desarrollo RESTful en una frase: utilizamos URIs para conectar clientes y servidores para intercambiar recursos en forma de sus representaciones. O también: en una arquitectura con estilo REST, los clientes y servidores intercambian representaciones de los recursos utilizando un protocolo e interfaces estandarizados.

2. Tipos de peticiones HTTP

A continuación vamos a ver los cuatro tipos de peticiones HTTP con detalle, y veremos cómo se utiliza cada una de ellas para intercambiar representaciones para modificar el estado de los recursos.

2.1. GET/RETRIEVE

El método GET se utiliza para **RECUPERAR** recursos. Antes de indicar la mecánica de la peticion GET, vamos a determinar cuál es el recurso que vamos a manejar y el tipo de representación que vamos a utilizar. Para ello vamos a seguir un ejemplo de un servicio web que gestiona alumnos en una clase, con la URI: *http://restfuljava.com*. Para dicho servicio, asumiremos una representación como la siguiente:

```
<alumno>
    <nombre>Esther</nombre>
    <edad>10</edad>
    <link>/alumnos/Jane</link>
</alumno>
```

Una lista de alumnos tendrá el siguiente aspecto:

```
<alumnos>
  <alumno>
    <nombre>Esther</nombre>
    <edad>10</edad>
    <link>/alumnos/Esther</link>
  <alumno>
  <alumno>
    <nombre>Pedro</nombre>
    <edad>11</edad>
    <link>/alumnos/Pedro</link>
  <alumno>
</alumnos>
```

Una vez definida nuestra representación, asumimos que las URIs tienen la forma: *http://restfuljava.com/alumnos* para acceder a la lista de alumnos, y *http://restfuljava.com/alumnos/{nombre}* para acceder a un alumno específico con el identificador con el valor *nombre*.

Ahora hagamos peticiones sobre nuestro servicio. Por ejemplo, si queremos recuperar la información de una alumna con el nombre *Esther*, realizamos una petición a la URI: *http://restfuljava.com/alumnos/Esther*.

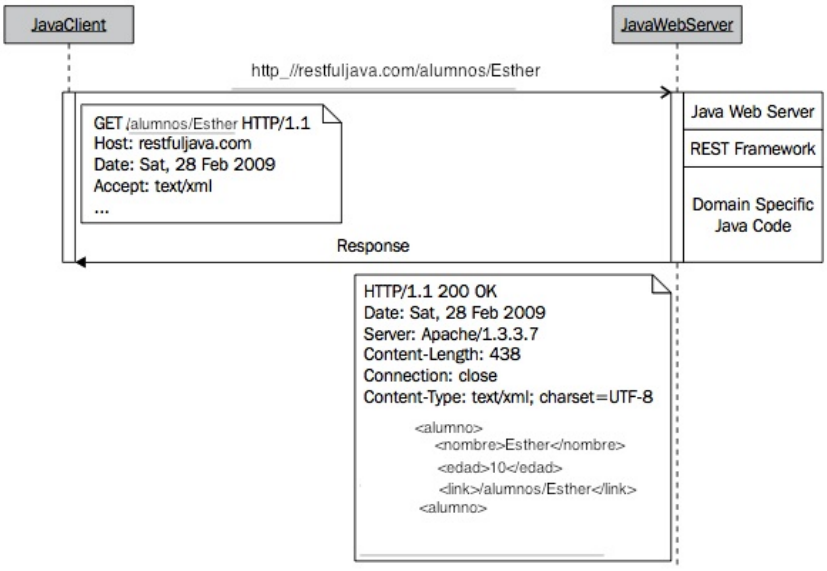
Una representación de *Esther* en el momento de la petición, puede ser ésta:

```
<alumno>
   <nombre>Esther</nombre>
   <edad>10</edad>
   <link>/alumnos/Esther</link>
</alumno>
```

También podríamos acceder a una lista de estudiantes a través de la URI: y *http://restfuljava.com/alumnos* y la respuesta del servicio sería algo similar a ésta (asumiendo que solamente hay dos alumnos):

```
<alumnos>
  <alumno>
    <nombre>Esther</nombre>
    <edad>10</edad>
    <link>/alumnos/Esther</link>
  <alumno>
  <alumno>
    <nombre>Pedro</nombre>
    <edad>11</edad>
    <link>/alumnos/Pedro</link>
  <alumno>
</alumnos>
```

Echemos un vistazo a los detalles de la petición. Una petición para recuperar un recurso *Esther* usa el método GET con la URI: *http://restfuljava.com/alumnos/Esther*. Un diagrama de secuencia de dicha petición sería como el que mostramos a continuación:



Escenario de petición GET/RETRIEVE

¿Qué está ocurriendo aquí?:

1. Un cliente Java realiza una petición HTTP con el método GET y Esther es el identificador del alumno
2. El cliente establece la representación solicitada a través del campo de cabecera

`Accept`

3. El servidor web recibe e interpreta la petición GET como una acción RETRIEVE. En este momento, el servidor web cede el control al *framework* RESTful para gestionar la petición. Remarquemos que los *frameworks* RESTful no recuperan de forma automática los recursos, ése no es su trabajo. La función del *framework* es facilitar la implementación de las restricciones REST. La lógica de negocio y la implementación del almacenamiento es el papel del código Java específico del dominio.
4. El programa del lado del servidor busca el recurso *Esther*. Encontrar el recurso podría significar buscarlo en una base de datos, un sistema de ficheros, o una llamada a otro servicio web.
5. Una vez que el programa encuentra a *Esther*, convierte el dato binario del recurso a la representación solicitada por el cliente.
6. Con la representación convertida a XML, el servidor envía de vuelta una respuesta HTTP con un código numérico de 200 (`Ok`) junto con la representación solicitada. Si hay algún error, el servidor HTTP devuelve el código numérico correspondiente, pero es el cliente el que debe tratar de forma adecuada el fallo. El fallo más común es que el recurso no exista, en cuyo caso se devolvería el código 404 (`Not Found`).

Todos los mensajes entre el cliente y el servidor son llamadas del protocolo estándar HTTP. Para cada acción de recuperación, enviamos una petición GET y obtenemos una respuesta HTTP con la representación del recurso solicitada, o bien, si hay un fallo, el correspondiente código de error (por ejemplo, `404 Not Found` si un recurso no se encuentra; `500 Internal Server Error` si hay un problema con el código Java en forma de una excepción).

En las peticiones de recuperación de datos resulta recomendable también implementar un sistema de caché. Para hacer esto utilizaremos el código de respuesta `304 Not Modified` en caso de que los datos no hubiesen cambiado desde la última petición que realizamos (se podría pasar un parámetro con la fecha en la que obtuvimos la representación por última vez). De esta forma, si un cliente recibe ese código como respuesta, sabe que puede seguir trabajando con la representación de la que ya dispone, sin tener que descargar una nueva.

Solicitar una representación para todos los alumnos funciona de forma similar.

Nota

El método HTTP GET solamente debería utilizarse para recuperar representaciones. Podríamos utilizar una petición GET para actualizar el estado de los datos en el servidor, pero no es recomendable. Una operación GET debe ser segura e idempotente (para más información ver http://www.w3.org/DesingIssues/Axioms. Para que una petición sea **segura**, múltiples peticiones al mismo recurso no deben cambiar el estado de los datos en el servidor. Por ejemplo, supongamos una petición en el instante t1 para un recurso R devuelve R1; a continuación, una petición en el instante t2 para R devuelve R2; suponiendo que no hay más acciones de modificación entre t1 y t2, entonces R1 = R2 = R. Para que una petición sea **idempotente** tiene que ocurrir que múltiples llamadas a la misma acción dejan siempre el mismo estado en el recurso. Por ejemplo, múltiples llamadas para crear un recurso R en los instantes t1, t2, y t3, darían como resultado que el recurso R existe sólo como R, y que las llamadas en los instantes t2 y t3 son ignoradas.

2.2. POST/CREATE

El método POST se utiliza para **CREAR** recursos. Vamos a utilizar el método HTTP POST para crear un nuevo alumno. De nuevo, la URI para añadir un nuevo alumno a nuestra lista será: *http://restfuljava.com/alumnos*. El tipo de método para la petición lo determina el cliente.

Asumamos que el alumno con nombre *Ricardo* no existe en nuestra lista y queremos añadirlo. Nuestra nueva representación XML de *Ricardo* es:

```
<alumno>
   <nombre>Ricardo</nombre>
   <edad>10</edad>
   <link></link>
</alumno>
```

El elemento *link* forma parte de la representación, pero está vacía debido a que éste valor se genera en tiempo de ejecución y no es creado por el cliente cuando envía la petición POST. Esto es solamente una convención para nuestro ejemplo; sin embargo, los clientes que utilizan el servicio web pueden especificar la estructura de las URIs.

En este caso, no mostraremos el escenario, pero los pasos que se siguen cuando se realiza la petición son los siguientes:

1. Un cliente Java realiza una petición HTTP a la URI *http://restfuljava.com/alumnos*, con el método HTTP POST
2. La petición POST incluye una representación en forma de XML de *Ricardo*
3. El servidor web recibe la petición y delega en el *framework* REST para que la gestione; nuestro código dentro del *framework* ejecuta los comandos adecuados para almacenar dicha representación (de nuevo, el dispositivo de almenamiento puede ser cualquiera).
4. Una vez que se ha completado el almacenamiento del nuevo recurso, se envía una respuesta de vuelta: si no ha habido ningún error, enviaremos el código 201 (`Created`); si se produce un fallo, enviaremos el código de error adecuado. Además, podemos devolver en la cabecera `Location` la URL que nos dará acceso al recurso recién creado.

```
Location: http://restfuljava.com/alumnos/Ricardo
```

Las peticiones POST no son idempotentes, por lo que si invocamos una misma llamada varias veces sobre un mismo recurso, el estado del recurso puede verse alterado en cada una de ellas. Por ejemplo, si ejecutamos varias veces la acción POST con los datos del ejemplo anterior, podríamos estar creando cada vez un nuevo alumno de nombre `Ricardo`, teniendo así varios alumnos con el mismo nombre y edad (pero asociados a IDs distintos, por ejemplo: `/Ricardo`, `/Ricardo1`, `/Ricardo2`, etc).

2.3. PUT/UPDATE

El método PUT se utiliza para **ACTUALIZAR** (modificar) recursos, o para crearlos si el recurso en la URI especificada no existiese previamente. Es decir, PUT se utiliza para establecer un determinado recurso, dada su URI, a la representación que proporcionemos, independientemente de que existiese o no. Para actualizar un recurso, primero necesitamos su representación en el cliente; en segundo lugar, en el lado del cliente actualizaremos el recurso con los nuevos valores deseados; y finalmente, actualizaremos el recurso mediente una petición PUT, adjuntando la representación correspondiente.

Para nuestro ejemplo, omitiremos la petición GET para recuperar a *Esther* del servicio web, ya que es el mismo que acabamos de indicar en la sección anterior. Supongamos que queremos modificar la edad, y cambiarla de 10 a 12. La nueva representación será:

```
<alumno>
   <nombre>Esther</nombre>
   <edad>12</edad>
   <link>/alumnos/Esther</link>
</alumno>
```

La secuencia de pasos necesarios para enviar/procesar la petición PUT es:

1. Un cliente Java realiza una petición HTTP PUT a la URI *http://restfuljava.com/alumnos/Esther*, incluyendo la nueva definición XML
2. El servidor web recibe la petición y delega en el *framework* REST para que la gestione; nuestro código dentro del *framework* ejecuta los comandos adecuados para actualizar la representación de *Esther*.
3. Una vez que se ha completado la actualización, se envía una respuesta al cliente. Si el recurso que hemos enviado no existía previamente, se devolverá el código 201 (`Created`). En caso de que ya existiese, se podría devolver 200 (`Ok`) con el recurso actualizado como contenido, o simplemente 204 (`No Content`) para indicar que la operación se ha realizado correctamente sin devolver ningún contenido.

Muchas veces se confunden los métodos PUT y POST. El significado de estos métodos es el siguiente:

- **POST**: Publica datos en un determinado recurso. El recurso debe existir previamente, y los datos enviados son añadidos a él. Por ejemplo, para añadir nuevos alumnos con POST hemos visto que debíamos hacerlo con el recurso lista de alumnos (`/alumnos`), ya que la URI del nuevo alumno todavía no existe. La operación **no es idempotente**, es decir, si añadimos varias veces el mismo alumno aparecerá repetido en nuestra lista de alumnos con URIs distintas.
- **PUT**: Hace que el recurso indicado tome como contenido los datos enviados. El recurso podría no existir previamente, y en caso de que existiese sería sobrescrito con la nueva información. A diferencia de POST, PUT **es idempotente**. Múltiples llamadas idénticas a la misma acción PUT siempre dejarán el recurso en el mismo estado. La acción se realiza sobre la URI concreta que queremos establecer (por ejemplo, `/alumnos/Esther`), de forma que varias llamadas consecutivas con los mismos datos tendrán el mismo efecto que realizar sólo una de ellas.

Podríamos añadir nuevos alumnos de dos formas diferentes. La primera de ellas es

haciendo POST sobre el recurso que contiene la lista de alumnos:

```
POST /alumnos HTTP/1.1
```

También podríamos hacer PUT sobre el recurso de un alumno concreto:

```
PUT /alumnos/Esther HTTP/1.1
```

Si *Esther* existía ya, sobrescribirá sus datos, en casa contrario, creará el nuevo recurso.

Si utilizamos POST de esta última forma, sobre un recurso concreto, si el recurso existiese podríamos realizar alguna operación que modifique sus datos, pero si no existiese nos daría un error, ya que no podemos hacer POST sobre un recurso inexistente.

```
POST /alumnos/Esther HTTP/1.1
```

El caso anterior sólo será correcto si *Esther* existe, en caso contrario obtendremos un error. Para crear nuevos recursos con POST debemos recurrir al recurso del conjunto de alumnos. Una diferencia entre estas dos formas alternativas de crear nuevos recursos es que con PUT podemos indicar explícitamente el identificador del recurso creado, mientras que con POST será el servidor quien lo decida.

2.4. DELETE/DELETE

El método DELETE se utiliza para **BORRAR** representaciones. Para nuestro ejemplo, usaremos la misma URI de las secciones anteriores.

La secuencia de pasos necesarios para enviar/procesar la petición DELETE es:

1. Un cliente Java realiza una petición DELETE a la URI *http://restfuljava.com/alumnos/Esther*
2. El servidor web recibe la petición y delega en el *framework* REST para que la gestione; nuestro código dentro del *framework* ejecuta los comandos adecuados para borrar la representación de *Esther*.
3. Una vez que se ha completado la actualización, se envía una respuesta al cliente. Se podría devolver 200 (`Ok`) con el recurso borrado como contenido, o simplemente 204 (`No Content`) para indicar que la operación se ha realizado correctamente sin devolver ningún contenido.

Hasta aquí hemos visto las principales acciones que podemos realizar con los recursos en un servicio web RESTful. No conocemos cómo el servicio web implementa el almacenamiento de los datos, y no conocemos qué tecnologías se utilizan para implementar el servicio. Todo lo que conocemos es que nuestro cliente y servidor se comunican a través de HTTP, que usamos dicho protocolo de comunicaciones para enviar peticiones, y que nuestras representaciones de los recursos se intercambian entre el cliente y el servidor a través del intercambio de URIs.

3. Clientes de servicios REST

3.1. Invocación de servicios RESTful desde una clase Java

Vamos a ver como crear un cliente RESTful utilizando una sencilla clase Java. Para ello vamos a utilizar el API de mensajes proporcionado por Twitter (http://www.twitter.com). No va a ser necesario disponer de una cuenta de Twitter ni conocer con detalle qué es Twitter para seguir el ejemplo.

Nota

Twitter es una plataforma de *micro-blogging* que permite que múltiples usuarios actualicen su estado utilizando 140 caracteres como máximo cada vez. Además, los usuarios pueden "seguirse" unos a otros formando redes de "amigos". Twitter almacena estas actualizaciones en sus servidores, y por defecto, están disponibles públicamente. Es por ésto por lo que utilizaremos Twitter para crear nuestros ejemplos de clientes REST.

```
public class ClienteTwitter {
  public static void main(String[] args) {
    try {
      URL twitter = new
        URL("http://twitter.com/statuses/public_timeline.xml");

      // Abrimos la conexión
      URLConnection tc = twitter.openConnection();

      // Obtenemos la respuesta del servidor
      BufferedReader in = new BufferedReader(new
                InputStreamReader( tc.getInputStream()));
      String line;

      // Leemos la respuesta del servidor y la imprimimos
      while ((line = in.readLine()) != null) {
           System.out.println(line);
      }
      in.close();
    } catch (MalformedURLException e) {
           e.printStackTrace();
    } catch (IOException e) {
           e.printStackTrace();
    }
  }
}
```

Podemos ver que hemos utilizado el paquete estándar `java.net`. La URI del servicio web es: *http://twitter.com/statuses/public_timeline.xml*. Ésta será la URI de nuestro recurso y apuntará a las últimas 20 actualizaciones públicas.

Para conectar con el servicio web, primero tenemos que instanciar el objeto URL con la URI del servicio. A continuación, "abriremos" un objeto *URLConnection* para la instancia de Twitter. La llamada al método *twitter.openConnection()* ejecuta una petición HTTP GET.

Una vez que tenemos establecida la conexión, el servidor devuelve la respuesta HTTP. Dicha respuesta contiene una representación XML de las actualizaciones. Por

simplicidad, volcaremos en la salida estandar la respuesta del servidor. Para ello, primero leemos el *stream* de respuesta en un objeto *BufferedReader*, y a continuación realizamos un bucle para cada línea del *stream*, asignándola a un objeto *String*. Finalmente hemos incluido nuestro código en una sentencia *try/catch*, y enviamos cualquier mensaje de excepción a la salida estándar.

Éste es el estado público de la última actualización de Twitter de la estructura XML obtenida (sólo mostramos parte de uno de los *tweets*).

```
<?xml version="1.0" encoding="UTF-8"?> <statuses type="array">
  ...
  <status>
    <created_at>Tue Feb 22 11:43:25 +0000 2011</created_at>
    <id>40013788233216000</id>
    <text>Haar doen voor de cam. #ahahah</text>
    <source>web</source>
    <truncated>false</truncated>
    <favorited>false</favorited>
    <in_reply_to_status_id></in_reply_to_status_id>
    <in_reply_to_user_id></in_reply_to_user_id>
    <in_reply_to_screen_name></in_reply_to_screen_name>
    <retweet_count>0</retweet_count>
    <retweeted>false</retweeted>
    <user>
      <id>250090010</id>
      <name>Dani&#235;l van der wal</name>
      <screen_name>DanielvdWall</screen_name>
      <location>Nederland,   Hoogezand</location>
      <description></description>
      <profile_image_url>
http://a0.twimg.com/profile_images/1240171940/Picture0003_normal.JPG
      </profile_image_url>
      <url>http://daniel694.hyves.nl/</url>
      <protected>false</protected>
       <followers_count>50</followers_count>
       ...
      <friends_count>74</friends_count>
      ...
      <following>false</following>
      <statuses_count>288</statuses_count>
      <lang>en</lang>
      <contributors_enabled>false</contributors_enabled>
      <follow_request_sent>false</follow_request_sent>
      <listed_count>0</listed_count>
      <show_all_inline_media>false</show_all_inline_media>
      <is_translator>false</is_translator>
    </user>
    <geo/>
    <coordinates/>
    <place/>
    <contributors/>
  </status>
...
</statuses>
```

No entraremos en los detallos de la estructura XML, podemos encontrar la documentación del API en http://apiwiki.twitter.com/Twitter-API-Documentation

La documentación del API nos dice que si cambiamos la extensión *.xml* obtendremos diferentes representaciones del recurso. Por ejemplo, podemos cambiar *.xml*, por *.json*,

.rss o *.atom*. Así, por ejemplo, si quisiéramos recibir la respuesta en formato JSON (JavaScript Object Notation), el único cambio que tendríamos que hacer es en la siguiente línea:

```
URL twitter =
      new URL("http://twitter.com/statuses/public_timeline.json");
```

En este caso, obtendríamos algo como esto:

```
[{"in_reply_to_status_id_str":null,"text":"THAT GAME SUCKED ASS.",
"contributors":null,"retweeted":false,"in_reply_to_user_id_str"
:null,"retweet_count":0,"in_reply_to_user_id":null,"source":"web",
"created_at":"Tue Feb 22 11:55:17 +0000 2011","place":null,
"truncated":false,"id_str":"40016776221696000","geo":null,
"favorited":false,"user":{"listed_count":0,"following":null,
"favourites_count":0,"url":"http:\/\/www.youtube.com\/user\
/haezelnut","profile_use_background_image":true,...
```

Los detalles sobre JSON se encuentran en la documentación del API. Para "parsear" un documento JSON podemos utilizar varias librerías, como por ejemplo *Jettison* (http://jettison.codehaus.org/), o *gson* (http://code.google.com/p/google-gson/), que podrá ser utilizada también en aplicaciones Android. Hablaremos un poco más adelante sobre las estructuras JSON.

Nota

Aunque Twitter referencia este servicio como servicio RESTful, esta API en particular no es completamente RESTful, debido a una elección de diseño. Analizando la documentación del API vemos que el tipo de representación de la petición forma parte de la URI y no de la cabecera *accept* de HTTP. El API devuelve una representación que solamente depende de la propia URI: *http://twitter.com/statuses/public_timeline.FORMATO*, en donde FORMATO puede ser *.xml*, *.json*, *.rss*, o *.atom*. Sin embargo, esta cuestión no cambia la utilidad del API para utilizarlo como ejemplo para nuestro cliente REST.

Otra posibilidad de implementación de nuestro cliente Java es utilizar la librería para clientes HTTP de Jakarta Commons. Dicha librería ofrece una mayor facilidad para controlar y utilizar objetos de conexión HTTP.

El código de nuestra clase cliente utilizando la librería quedaría así:

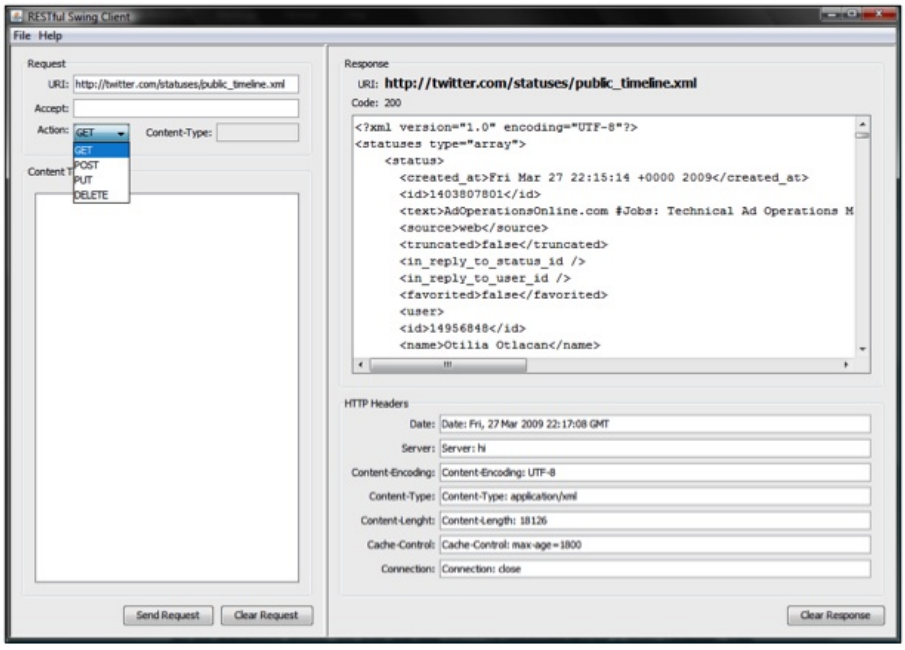
```
public class ClienteTwitter {
  public static void main(String[] args) {
    HttpClient client = new HttpClient();
    GetMethod method = new GetMethod(
        "http://twitter.com/statuses/public_timeline.xml");
    try {
      int statusCode = client.executeMethod(method);
      if (statusCode == HttpStatus.SC_OK) {
        System.out.println(new
             String(method.getResponseBody()));
      }
    } catch (HttpException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
```

```
      method.releaseConnection();
    }
  }
}
```

Observamos que primero instanciamos el cliente HTTP y procedemos a crear un objeto que representa el método HTTP GET. Con el cliente y el método instanciado, necesitamos ejecutar la petición con el método `executeMethod`. Con esta librería tenemos un mayor control y podemos añadir fácilmente una comprobación de errores en nuestro programa. Por ejemplo, podemos mostrar la respuesta sólo si obtenemos un código de estado HTTP 200. Finalmente necesitamos liberar la conexión.

3.2. Invocación de servicios RESTful desde una aplicación Java de escritorio

En esta sección mostraremos un cliente RESTful utilizando una aplicación Swing Java. El aspecto de la aplicación es el siguiente:



Aplicación cliente RESTful desde una aplicación de escritorio

Esta aplicación conecta con cualquier servicio web utilizando el valor del campo de texto URI. Debido a que los servicios RESTful son accesibles a través de HTTP, podemos utilizar esta aplicación para probar cualquier servicio web RESTful (lo implementemos nosotros o no). La imagen anterior muestra la petición y respuesta a la URI de Twitter. Además, podemos indicar qué tipo de método queremos invocar, y la respuesta se muestra en el panel de la derecha. También se visualizan algunas de las respuestas de la cabecera HTTP. Finalmente, podemos "limpiar" cada panel con sus respectivos botones *Clear Request* o *Clear Response*.

El código de la aplicación la proporcionamos en el apartado de *plantillas* y está formada por cuatro ficheros, con la siguiente estructura:

```
/RESTfulSwingClient/
   /RESTfulSwingClientApp.java
   /RESTfulSwingClientView.java
   /resources/
        /RESTfulSwingClientApp.properties
        /RESTfulSwingClientView.properties
```

La mayor parte de este código crea el GUI del cliente, además se incluyen dos ficheros de propiedades de fichero Java que proporcionan valores a mostrar por el GUI en tiempo de ejecución. Evidentemente, no vamos a mostrar todo el código, sino solamente a comentar algunas partes del mismo.

Concretamente, vamos a fijarnos en la clase factoría que ayuda a determinar qué tipo de método de petición crear (dentro del fichero `RESTfulSwingClientView.java`). Para cada tipo de método de petición, necesitamos una implementación concreta que ya hemos utilizado con la instancia del cliente HTTP Commons. La clase es la siguiente:

```
class MethodFactory {
  public MethodFactory() {
    super();
  }
  public HttpMethod getMethod(String methodType, String URI)
                      throws Exception {
    HttpMethod method = null;
    if (methodType.equals("GET")) {
      method = new GetMethod(URI); }
    else if (methodType.equals("POST")) {
      method = new PostMethod(URI);
    } else if (methodType.equals("PUT")) {
      method = new PutMethod(URI);
    } else if (methodType.equals("DELETE")) {
      method = new DeleteMethod(URI);
    }
    if (method != null) { // Con POST y PUT se utiliza "Content-Type"
      if (methodType.equals("POST") ||
          methodType.equals("PUT")) {
        ((EntityEnclosingMethod)
            method).setRequestEntity(new
               StringRequestEntity(jTextAreaReqBody.getText().trim(),
               jTextFieldReqContentType.getText().trim(), "UTF-8"));
      }
      return method;
    }
    return null;
  }
```

Esta factoría devuelve el objeto *HttpMethod* adecuado, junto con la información adicional requerida, dependiendo de si se trata de un método GET, POST, PUT o DELETE. Así, por ejemplo, si se trata de un método POST o PUT, necesitamos el contenido del componente `jTextAreaReqBody`.

Finalmente, el manejo de las peticiones a enviar utiliza un código similar al que utilizamos en nuestro cliente Http anterior. La diferencia es que obtenemos los valores a través del GUI y por lo tanto, cuando actualizamos la vista con los valores resultantes. El

código que gestiona las peticiones también se encuentra en el fichero `RESTfullSwingClientView.java` y es el siguiente:

```
private void handleSendRequest() {
  String requestURI = jTextFieldReqURI.getText().trim();
  if (!requestURI.isEmpty()) {
    try {

      // Limpia la respuesta
      handleClearResponse();

      // Instancia el cliente
      HttpClient client = new HttpClient();

      // Obtiene el tipo de metodo de la factoria
      HttpMethod method = new
         MethodFactory().getMethod(jComboBoxMethod
           .getSelectedItem().toString(), requestURI);

      // Realiza la peticion HTTP
      int statusCode = client.executeMethod(method);

      // Actualiza la interfaz
      jLabelURI.setText(requestURI);
      jLabelResultCode.setText("" + statusCode);
      jTextAreaResBody.setText(
                  method.getResponseBodyAsString());

      // Omitimos el resto de las actualizaciones de los campos

    } catch (Exception ex) {
      ex.printStackTrace();

      // Se muestra el error en el campo URI
      handleClearResponse();
      jLabelURI.setText("Error with URI: " + requestURI);
      jTextAreaResBody.setText(ex.getMessage());
    }
  }
}
```

3.3. Invocación de servicios RESTful desde una aplicación JSP

Para esta sección vamos a utilizar la representación RSS de las últimas 20 actualizaciones. La URI, por lo tanto, será: *http://twitter.com/statuses/public_timeline.rss*. Utilizaremos el navegador para visualizar los resultados.

Ahora codificamos nuestro cliente. En este caso, vamos a utilizar el mismo patrón de código que para nuestro cliente con una clase Java, pero desde nuestra página jsp (*index.jsp*). El contenido, por lo tanto, del fichero *src/main/webapp/index.jsp* será el siguiente:

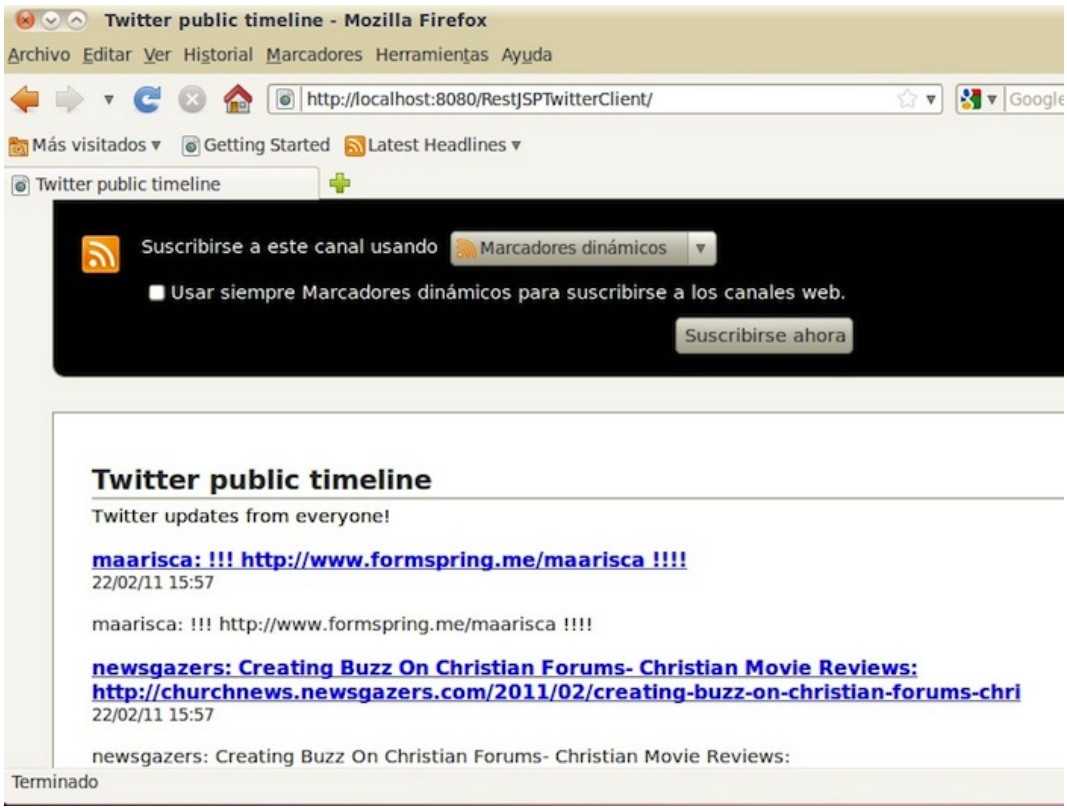
```
<%@ page contentType="text/xml; charset=UTF-8" %><%@ page import=
"java.io.BufferedReader, java.io.IOException, java.io.InputStreamReader,
java.net.MalformedURLException, java.net.URL, java.net.URLConnection" %><%
// Hemos eliminado cualquier CR and LF del código JSP, debido a
// que, cuando el fichero se genera sobre el servidor, guarda estos
// caracteres haciendo que el XML sea inválido
//(esto es un requerimiento de JSP)
```

```
 try {
      URL twitter = new
          URL("http://twitter.com/statuses/public_timeline.rss");
      URLConnection tc = twitter.openConnection();
      BufferedReader in = new BufferedReader(
          new InputStreamReader(tc.getInputStream()));
      String line;
      while ((line = in.readLine()) != null) {
          out.println(line);
      }
      in.close();
 } catch (MalformedURLException e) {
      e.printStackTrace();
 } catch (IOException e) {
      e.printStackTrace();
 }
%>
```

Nota

Cuidado con los espacios en blanco, retornos de carro y finales de línea del código jsp. No tiene que haber ninguno de estos caracteres entre cada par de delimitadores de código jsp: <% ... %>

Podemos observar dos cosas nuevas en el código anterior. Primero, utilizamos la URI correspondiente a una petición de formato *rss*, en lugar de *xml* o *json* (como habíamos hecho anteriormente). La información RSS consiste en un fichero formateado como XML. Este tipo de ficheros se utiliza con frecuencia por varios programas sindicados y servicios (podemos encontrar más detalles sobre RSS en http://www.rssboard.org/). En segundo lugar, consumimos la respuesta escribiendo en la salida estándar de JSP



Resultado de ejecutar el cliente RESTful desde un JSP

Al ejecutar este código, podemos apreciar que la respuesta de este cliente JSP es la misma que si estuviésemos ejecutando la URI directamente desde los servidores de Twitter. Podemos pensar en este cliente JSP como en un *proxy* para la API de Twitter. No obstante, ahora podemos conectarnos con cualquier servicio disponible, manipular la respuesta como queramos, y crear nuestros propios *mashups* de servicios. (Un *mashup* es una aplicación web que combina más de un servicio web).

3.4. Parsing de estructuras JSON

JSON es una representación muy utilizada para formatear los recursos solicitados a un servicio web RESTful. Se trata de ficheros con texto plano que pueden ser manipulados muy fácilmente utilizando JavaScript.

La gramática de los objetos JSON es simple y requiere la agrupación de la definición de los datos y valores de los mismos. En primer lugar, los elementos están contenidos dentro de llaves `{` y `}`; los valores de los diferentes elementos se organizan en pares, con la estructura: `"nombre":"valor"`, y están separados por comas; y finalmente, las secuencias de elementos están contenidas entre corchetes `[` y `]`. Y esto es todo :)! Para una descripción detallada de la gramática, podéis consultar http://www.json.org/fatfree.html

Con la definición de agrupaciones anterior, podemos combinar múltiples conjuntos para crear cualquier tipo de estructura requerido. El siguiente ejemplo muestra una descipción JSON de un objeto con información sobre un cliente:

```
{   "firstName": "John",
     "lastName": "Smith",
     "age": 25,
     "address":
     { "streetAddress": "21 2nd Street",
         "city": "New York",
         "state": "NY",
         "postalCode": "10021"
     },
     "phoneNumber":
     [  {
           "type": "home",
           "number": "212 555-1234"
         },
         {
           "type": "fax",
           "number": "646 555-4567"
         }
      ]
 }
```

Antes de visualizar cualquiera de los valores de una respuesta JSON, necesitamos convertirla en una estructura que nos resulte familiar, por ejemplo, sabemos como trabajar con las jerarquías de objetos Javascript.

Para convertir una cadena JSON en código "usable" utilizaremos la función Javascript nativa *eval()*. En el caso de un *stream* JSON, *eval()* transforma dicho *stream* en un objeto junto con propiedades que son accesibles sin necesidad de manipular ninguna cadenas de caracteres.

Nota

Los *streams* JSON son fragmentos de código Javascript y deben evaluarse utilizando la función *eval()* antes de poder utilizarse como objetos en tiempo de ejecución. En general, ejecutar Javascript a través de *eval()* desde fuentes no confiables introducen riesgos de seguridad debido al valor de los parámetros de las funciones ejecutadas como Javascript. Sin embargo, para nuestra aplicación de ejemplo, confiamos en que el JavaScript enviado desde Twitter es una estructura JSON segura.

Debido a que un objeto JSON evaluado es similar a un objeto DOM, podremos atravesar el árbol del objeto utilizando el carácter "punto". Por ejemplo, un elemento raíz denominado `Root` con un sub-elemento denominado `Element` puede ser accedido mediante `Root.Element`

Nota

Si estamos ante una cadena JSON sin ninguna documentación del API correspondiente, tendremos que buscar llaves de apertura y cierre (`{` y `}`). Esto nos llevará de forma inmediata a la definición del objeto. A continuación buscaremos pares de de nombre/valor entre llaves.

4. Creación de servicios REST con Jersey

En el mundo RESTful encontramos varios *frameworks* Java para servicios RESTful puros disponibles. Uno de ellos es Jersey, la implementación de referencia del API de Sun de Java para servicios web RESTful, más comúmente conocida como JAX-RS. En esta sesión introduciremos el uso de las librerías JAX-RS para la creación de servicios web RESTful.

4.1. JAX-RS y Jersey

La tecnología java siempre ha formado parte de las soluciones para implementar servicios web. Con la llegada de REST, el proyecto JAX-RS fue iniciado por la *Java Community Process* (JCP) con el objetivo de crear un API java para servicios RESTful. El API se conoce también como JAX-311 o JAX-RS. Esta especificación es utilizada por cualquiera que quiera implementar un *framework* Java que se adhiera a las restricciones impuestas por REST y comentadas en la sesión anterior.

Una implementación de referencia es aquella que implementa todos los requerimientos de una especificación particular. Las implementaciones de referencia no comienzan con el rendimiento de la producción en mente, debido a que deben implementar cada detalle de la especificación. No obstante, Jersey ha evolucionado a una opción viable para el desarrollo de servicios web RESTful.

Una de las metas del grupo de JAX-RS es proporcionar una especificación independiente del contenedor, y por este motivo, Jersey funciona con cualquier servidor JEE. Jersey es un proyecto open source y está constantemente siendo actualizado. Podemos consultar información sobre Jersey en http://jersey.java.net

4.2. Anotaciones Jersey

El principal objetivo de la especificación JAX-RS es posibilitar el desarrollo de servicios web RESTful de una forma sencilla. Jersey proporciona los conectores para los servicios web a través de anotaciones Java. Las anotaciones automáticamente generan el código necesaro para las clases que usan dicho código para conectar sin problemas con *frameworks* (librerías) específicos.

El uso de anotaciones nos permite crear recursos Jersey tan fácilmente como desarrollar POJOs (Plain Old Java Objects). En otras palabras, nos olvidamos de la tarea de interceptar las peticiones HTTP y las negociaciones con el *framework* para centrarnos en las reglas de negocio necesarias para resolver nuestro problema.

Los desarrolladores "decoran" los ficheros de clases Java de una aplicación web con anotaciones específicas para definir recursos y las acciones que pueden realizarse sobre dichos recursos. Estos recursos son expuestos a los clientes desplegando la aplicación

web en un servidor de aplicaciones Java EE o en un servidor web.

Antes de codificar nuestro servicio, vamos a analizar las anotaciones que proporciona Jersey.

La siguiente tabla muestra una lista de algunas de las anotaciones java definidas en JAX-RS, junto con una breve descripción de cómo se usa cada una.

Anotación	Descripción
@Path	Es una URI de un path relativo que indica dónde se mapeará el servicio. Por ejemplo, `/helloWorld`
@GET	Es un *request method designator*. El método java anotado con @GET procesa peticiones HTTP GET. El comportamiento del recurso es determinado por el método HTTP al que responde el recurso
@POST	Es un *request method designator*. El método java anotado con @POST procesa peticiones HTTP POST. El comportamiento del recurso es determinado por el método HTTP al que responde el recurso
@PUT	Es un *request method designator*. El método java anotado con @PUT procesa peticiones HTTP PUT. El comportamiento del recurso es determinado por el método HTTP al que responde el recurso
@DELETE	Es un *request method designator*. El método java anotado con @DELETE procesa peticiones HTTP DELETE. El comportamiento del recurso es determinado por el método HTTP al que responde el recurso
@HEAD	Es un *request method designator*. El método java anotado con @HEAD procesa peticiones HTTP HEAD. El comportamiento del recurso es determinado por el método HTTP al que responde el recurso
@PathParam	Es un tipo de parámetro que puede extraerse para utilizarse en la clase del recurso. Estos parámetros se extraen de la URI de la petición. En la plantilla de la ruta especificada mediante @Path se pueden incluir segmentos variables, y mediante @PathParam podemos hacer referencia a estas variables para obtener su valor y utilizarlo en nuestro código

@QueryParam	Es un tipo de parámetro que puede extraerse para utilizarse en la clase del recurso. Los parámetros de la *query* de la URI se extraen de los parámetros de *query* de la URI de la petición
@Consumes	Se utiliza para especificar el tipo MIME de las representaciones que un recurso puede consumir cuando ésta es enviada desde el cliente
@Produces	Se utiliza para especificar el tipo MIME de las representaciones que un recurso puede proporcionar y enviar al cliente, por ejemplo `"text/plain`
@Provider	Se utiliza para especificar cualquier elemento de interés para el *runtime* de JAX-RS, como por ejemplo `MessageBodyReader` y `MessageBodyWriter`. Para peticiones HTTP, el primero de ellos se utiliza para mapear el cuerpo de la entidad de una petición HTTP a parámetros de un método. En la parte de la respuesta, un valor de retorno se mapea a un cuerpo de una entidad de petición HTTP utilizando `MessageBodyWriter`. Si la aplicación necesita devolver metadatos adicionales, tales como cabeceras HTTP o un código de estado diferente, el método puede devolver un objeto de tipo `Response` que *envuelve* a la entidad y puede construirse utilizando `ResponseBuilder`

Denominamos clases raíz de recursos (***Root resource classes***) a POJOs anotados con @Path o tienen al menos un método anotado con @Path o con @GET, @PUT, @POST o @DELETE

Los métodos de recurso (***Resource methods***) son métodos de una clase de recurso anotada con un *request method designator*. A continuación explicaremos cómo anotar clases Java para crear servicios Web RESTful.

4.3. Implementación de aplicaciones JAX-RS

El siguiente ejemplo de código es un ejemplo muy sencillo de una clase raíz de recursos que utiliza anotaciones JAX-RS:

```
package org.especialistajee.rest;

@Path("/holamundo")
public class HolaMundoResource {

    @GET
    @Produces("text/plain")
```

```
    public String getSaludo() {
        return "Hola mundo!";
    }
}
```

En el ejemplo anterior, la clase java `HolaMundoResource` residirá en el *path* relativo `/holamundo`. El método `getSaludo` procesará peticiones HTTP GET, y "producirá" una respuesta que se enviará al cliente en formato texto.

Nota

Las clases raíz de recursos suelen tener como nombre el nombre del recurso seguido del sufijo `Resource`.

Para que la clase anterior pueda ser ofrecida como servicio REST a través de HTTP deberemos declarar y mapear un servlet proporcionado por Jersey, que se encargará de recibir y procesar las peticiones HTTP entrantes, llamar al método de servicio correspondiente, y con el resultado componer y devolver la respuesta HTTP al cliente. Esto lo deberemos configurar en el `web.xml` de la siguiente forma:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
               http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <servlet>
        <servlet-name>Jersey Web Application</servlet-name>
        <servlet-class>
           com.sun.jersey.spi.container.servlet.ServletContainer
        </servlet-class>
        <init-param>
            <param-name>
                com.sun.jersey.config.property.packages
            </param-name>
            <param-value>org.especialistajee.rest</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>Jersey Web Application</servlet-name>
        <url-pattern>/recursos/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Podemos ver que debemos indicar el paquete Java en el que tenemos las clases raíz de recursos y la ruta a la que se mapeará el servlet que nos dará acceso a los servicios REST.

Nota

Podemos ver que cuando trabajamos con servicios REST quien está atendiendo la petición realmente es un servlet. El *framework* Jersey nos evita tener que tratar con los mensajes HTTP directamente.

Por ejemplo, si nuestra aplicación web se encuentra mapeada al contexto `micontexto` en el host `jtech.ua.es`, para acceder al recurso *Hola mundo!* deberemos realizar una

petición a la siguiente URL:

```
http://jtech.ua.es/micontexto/recursos/holamundo
```

Si introducimos dicha URL en el navegador, obtendremos el mensaje `Hola mundo!` en texto plano.

4.3.1. La anotación @Path y plantillas de path para URIs

La anotación @Path identifica la plantilla de path para la URI del recurso al que se accede y se puede especificar a nivel de clase o a nivel de método de dicho recurso. La anotación @Path es relativa a la URI base del servidor en el que se despliega el recurso, a la raiz del contexto de la aplicación, y al patrón URL al que responde el *runtime* de JAX-RS.

La anotación @Path puede incluir variables entre llaves, que serán sustituidas en tiempo de ejecución dependiendo del valor que se indique en la llamada al recurso. Asi, por ejemplo, si tenemos la siguiente anotación:

```
@Path("/alumnos/{nombre}")
```

y el usuario introduce como `nombre` el valor *Pedro*, el servicio web responderá a la siguiente dirección:

```
http://jtech.ua.es/contexto/recursos/alumnos/Pedro
```

Para obtener el valor del nombre de usuario, utilizaremos la anotación @PathParam en los parámetros del método, de la siguiente forma:

```
@Path("/alumnos/{nombre}")
public class AlumnoResource {

   @GET
   @Produces("text/xml")
   public String getAlumno(@PathParam("nombre") String nombre) {
   ...
   }
}
```

Una URI puede tener más de una variable, cada una figurará entre llaves. Por ejemplo, si queremos desplegar un recurso que responda a la plantilla URI de path: `http://jtech.ua.es/contexto/recursos/{nombre1}/{nombre2}`, decoraremos nuestro recurso con la siguiente anotación @Path:

```
@Path("/{nombre1}/{nombre2}/")
public class MiResource {
  ...
}
```

Podemos tener *paths* para cada método, relativos al *path raiz* anotado en la definición de la clase. Por ejemplo, la siguiente clase de recurso sirve peticiones a la URI `/alumnos`

```
@Path("/alumnos")
public class AlumnoResource {
```

```
   @GET public String getAlumnos() { }
}
```

Si quisiéramos proporcionar el servicio en la URI `alumnos/mensajes`, por ejemplo, no necesitamos una nueva definición de clase, y podríamos anotar un nuevo método *getMensajesAlumnos* de la siguiente forma:

```
@Path("/alumnos")
public class AlumnoResource {
   @GET public String getAlumnos() { }

   @GET
   @Path("/mensajes")
   public String getMensajesAlumnos() { }
}
```

Ahora tenemos una clase de recurso que gestiona peticiones para `/alumnos` y `/alumnos/mensajes`

4.3.2. Usos de @Produces y @Consumes

Anotación @Consumes

Esta anotación funciona conjuntamente con @POST y @PUT. Le indica al *framework* (librerías Jersey) a qué método se debe delegar la petición de entrada. Específicamente, el cliente fija la cabecera HTTP *Content-Type* y el *framework* delega la petición al correspondiente método capaz de manejar dicho contenido. Un ejemplo de anotación con @PUT es la siguiente:

```
@Path("/alumnos")
public class AlumnoResource {
   @PUT
   @Consumes("application/xml")
   public void updateAlumno(String representation) { }
}
```

Si @Consumes se aplica a la clase, por defecto los métodos de respuesta aceptan los tipos especificados de tipo MIME. Si se aplica a nivel de método, se ignora cualquier anotación @Consumes a nivel de clase para dicho método.

En este ejemplo, le estamos indicando al *framework* que el método `updateUser` acepta un *input stream* cuyo tipo MIME es `"application/xml"`, y que se almacena en la variable `representation`. Por lo tanto, un cliente que se conecte al servicio web a través de la URI `/alumnos` debe enviar una petición HTTP PUT conteniendo el valor de `"application/xml"` como tipo MIME de la cabecera HTTP *Content-Type*.

Si no hay métodos de recurso que puedan responder al tipo MIME solicitado, se le devolverá al cliente un código HTTP 415 (*"Unsupported Media Type"*). Si el método que consume la representación indicada como tipo MIME no devuelve ninguna representación, se enviará un el código HTTP 204 (*"No content"*). Como por ejemplo sucede en el código siguiente:

```
@POST
@Consumes("text/plain")
public void postMensaje(String mensaje) {
    // Almacena el mensaje
}
```

Podemos ver que el método "consume" una representación en texto plano, pero devuelve `void`, es decir, no devuelve ninguna representación.

Un recurso puede aceptar diferentes tipos de "entradas". Así, podemos utilizar la anotación @PUT con más de un método para gestionar las repuestas con tipos MIME diferentes. Por ejemplo, podríamos tener un método para aceptar estructuras XML, y otro para aceptar estructuras JSON.

```
@Path("/alumnos")
public class AlumnoResource {

   @PUT
   @Consumes("application/xml")
   public void updateAlumnoXml(String representation) { }

   @PUT
   @Consumes("application/json")
   public void updateAlumnoJson(String representation) { }

}
```

Anotación @Produces

Esta anotación funciona conjuntamente con @GET, @POST y @PUT. Indica al *framework* qué tipo de representación se envía de vuelta al cliente.

De forma más específica, el cliente envía una petición HTTP junto con una cabecera *Accept* HTTP que se mapea directamente con el *Content-Type* que el método produce. Por lo tanto, si el valor de la cabecera *Accept* HTTP es `application/xml`, el método que gestiona la petición devuelve un *stream* de tipo MIME `application/xml`. Esta anotación también puede utlizarse en más de un método en la misma clase de recurso. Un ejemplo que devuelve representaciones XML y JSON sería el siguiente:

```
@Path("/alumnos")
public class AlumnoResource {

   @GET
   @Produces("application/xml")
   public String getAlumnosXml() { }

   @GET
   @Produces("application/json")
   public String getAlumnosJson() { }
}
```

Nota

Si un cliente solicita una petición a una URI con un tipo MIME no soportado por el recurso, Jersey lanza la excepción adecuada, concretamente el *runtime* de JAX-RS envía de vuelta un error HTTP 406 (*"Not acceptable"*)

Se puede declarar más de un tipo en la misma declaración @Produces, como por ejemplo:

```
@Produces({"application/xml", "application/json"})
public String getAlumnosXmlOJson() {
   ...
}
```

El método *getAlumnoXmlOJson* será invocado si cualquiera de los dos tipos MIME especificados en la anotación @Produces son aceptables (la cabecera *Accept* de la petición HTTP indica qué representación es aceptable). Si ambas representaciones son igualmente aceptables, se elegirá la primera.

Nota

En lugar de especificar los tipos MIME como cadenas de texto en @Consumes y @Produces, podemos utilizar las constantes definidas en la clase `MediaType`, como por ejemplo `MediaType.APPLICATION_XML` o `MediaType.APPLICATION_JSON`.

4.4. Tipos de datos en la petición y en la respuesta

Hasta el momento hemos estado utilizado el tipo `String` para devolver los datos al cliente, o para recoger el contenido de la petición. Sin embargo, podemos utilizar otros tipos de datos como entrada y salida de nuestras operaciones, como veremos a continuación.

4.4.1. Tipos de datos básicos

Los tipos básicos que podemos utilizar para obtener el contenido de la petición, o devolver el contenido de la respuesta son, además del tipo `String` visto anteriormente, los tipos `char []` y `byte []`, que nos permiten representar cualquier bloque de contenido, bien con codificación de caracteres o binaria.

En este caso tendremos que encargarnos de analizar manualmente el contenido de la petición en nuestro código, y de componer la respuesta.

```
@GET
@Produces("image/jpeg")
byte [] getImagen() {
    byte [] datos = leerFichero("imagen.jpg");
    return datos;
}

@PUT
@Consumes("image/jpeg")
void putImagen(byte [] datos) {
    guardarFichero("imagen.jpg", datos);
}
```

4.4.2. Escritura manual de la respuesta

Realmente, la forma más básica de enviar la respuesta al cliente es escribir directamente sobre el flujo de la misma. Para hacer esto podemos crear una clase que implemente `StreamingOutput`, lo que nos obligará a definir el método `write` en el que deberemos escribir la respuesta:

```
@GET
@Produces("text/plain")
StreamingOutput saluda() {
    return new StreamingOutput() {
        public void write(OutputStream output) throws IOException {
            output.write("Hola mundo!".getBytes());
        }
    };
}
```

Si queremos leer los datos de la petición directamente como flujo de entrada, simplemente utilizaremos como tipo `InputStream`:

```
@PUT
@Consumes("application/java-serialized-object")
public void putObjeto(InputStream is) {
    ObjectInputStream ois = new ObjectInputStream(is);
    Object obj = ois.readObject();
    ...
}
```

4.4.3. Ficheros

En los ejemplos anteriores hemos visto que podemos escribir directamente en el flujo de la respuesta, o leer del flujo de entrada. También hemos visto que con el tipo `byte []` o `char []` podríamos devolver cualquier contenido, por ejemplo contenido leído de ficheros binarios o de texto. Sin embargo, si lo que queremos es devolver el contenido de un fichero, lo más sencillo es abrir un flujo para leer el fichero y devolver dicho flujo como respuesta (o recibirlo como entrada). Por ejemplo, podemos trabajar con imágenes de la siguiente forma:

```
@GET
@Produces("image/jpeg")
InputStream getImagen() {
    return new FileInputStream("imagen.jpg");
}

@PUT
@Consumes("image/jpeg")
void putImagen(InputStream is) {
    guardarFichero("imagen.jpg", is);
}
```

De forma alternativa, también podemos utilizar el tipo `File`:

```
@GET
@Produces("image/jpeg")
File getImagen() {
    return new File("imagen.jpg");
}

@PUT
```

```
@Consumes("image/jpeg")
void putImagen(File file) {
    guardarFichero("imagen.jpg", new FileInputStream(file));
}
```

En el caso de leer un fichero como entrada, realmente estamos leyendo de un fichero temporal que JAX-RS crea con el contenido leído de la petición.

4.4.4. Datos de formularios

En muchas ocasiones nos pueden llegar datos de un formulario HTML, de forma que tendremos un conjunto de parejas *(clave, valor)*. Este conjunto de datos del formulario podemos recogerlo mediante el tipo `MultivaluedMap`, que es similar a `Map`, pero para cada clave nos permite tener varios valores. Es decir, `MultivaluedMap<K,V>` es equivalente a `Map<K,List<V>>`.

```
@POST
@Consumes("application/x-www-form-urlencoded")
public void addAlumno(
            MultivaluedMap<String, String> datosAlumno) {
    String nif = datosAlumno.getFirst("nif");
    String nombre = datosAlumno.getFirst("nombre");
    for(String tlfno: datosAlumnos.get("telefonos")) {
        ...
    }
}
```

Este tipo de datos también se podría utilizar para devolver una respuesta con el formato de los datos de formularios.

4.4.5. Beans JAXB

Como hemos visto anteriormente, es habitual utilizar XML o JSON como representación para los datos que intercambian nuestros servicios. En los ejemplos anteriores hemos utilizado el tipo `String` para los parámetros de entrada y salida, lo que nos forzaría a componer o analizar los mensajes XML y JSON manualmente. Sin embargo, gracias a JAXB podemos conseguir que los objetos Java se mapeen automáticamente a estas representaciones. Por ejemplo, podemos crear un servicio como el siguiente en el que la representación utilizada es JSON, pero como vemos los parámetros de entrada y de salida son nuestros propios objetos de dominio.

```
@Path("/estado")
public class EstadoResource {

    EstadoBean estadoBean = new EstadoBean();

    @GET
    @Produces("application/json")
    public EstadoBean getEstado() {
        return estadoBean;
    }

    @PUT
    @Consumes("application/json")
```

```
    public void setEstado(EstadoBean estado) {
       this.estadoBean = estado;
    }
}
```

Nuestro recurso `EstadoResource` será accedido con la URI `estado`, y responderá a peticiones http GET y PUT (indicado por las anotaciones `@GET,@PUT`, respectivamente). En el primer caso la petición http GET requiere una respuesta con un dato con el tipo MIME *application/json* (indicado por el campo *Accept* de la cabecera de la petición. En el segundo caso, la petición http PUT utilizará el valor *application/json* para el *Content-Type* de la cabecera de la petición, y devuelve una cadena de caracteres como respuesta.

El método *getEstado* será invocado por el *servlet* de Jersey cuando se realice una llamada http GET, y devuelve un objeto de tipo *EstadoBean*, que será "convertido" (*unmarchalled*) a formato JSON utilizando anotaciones de la librería JAXB, para ser enviado como respuesta al cliente.

El método *setEstado* será invocado por el *servlet* de Jersey cuando se realice una llamada http PUT, y requiere como parámetro de entrada un objeto de tipo *EstadoBean*, que habrá sido "generado" (*marshalled*) por la librería JAXB, utilizando la anotaciones correspondientes, a partir del documento JSON que el cliente ha enviado en la petición PUT.

Ahora veremos cómo, utilizando las anotaciones JAXB, nos "despreocuparemos" de las conversiones entre documentos xml/json y clases Java. Primero mostraremos el código de la clase `EstadoBean`

```
@XmlRootElement(name = "estado")
public class EstadoBean {

   public String estado = "Idle";
   public int tonerRestante = 25;
   public List<TareaBean> tareas =
             new ArrayList<TareaBean>();
}
```

La anotación `@XmlRootElement` realiza la serialización de la clase `EstadoBean` en formato xml/json utilizando los campos definidos en la clase (*estado*, *tonerRestante*, y *tareas*). Vemos que el campo *tareas*, a su vez, es una colección de elementos de tipo `TareaBean`, que también necesitan ser serializados. A continuación mostramos la implementación de la clase `TareaBean.java`:

```
@XmlRootElement(name = "tarea")
public class TareaBean {
   public String nombre;
   public String estado;
   public int paginas;

   public TareaBean() {};

   public TareaBean(String nombre, String estado,
                    int paginas) {
      this.nombre = nombre;
```

```
      this.estado = estado;
      this.paginas = paginas;
   }
}
```

JAXB

Para dar soporte al serializado y deserializado XML se utilizan *beans* JAXB.Por ejemplo las clases anteriores `EstadoBean` y `TareaBean` son ejemplos de *beans* JAXB. La clase que se serializará como un recurso XML o JSON se tiene que anotar con *@XmlRootElement*. Si en algún elemento de un recurso se retorna un elemento de esa clase y se etiqueta con los tipos `@Produces({MediaType.APPLICATION_XML,Mediatype.APPLICATION_JSON})`, éste se serializa automáticamente utilizando el tipo de representación aceptada por el cliente. Se puede consultar http://www.vogella.de/articles/JAXB/article.html para más información.

Si accedemos a este servicio, nos devolverá la información sobre el estado de la siguiente forma:

```
{"estado":"Idle","tonerRestante":"25","tareas":
  [{"nombre":"texto.doc","estado":"imprimiendo...","paginas":"13"},
   {"nombre":"texto2.doc","estado":"en espera...","paginas":"5"}]
}
```

Por defecto, cuando etiquetemos un *bean* con `@XmlRootElement`, su etiqueta raíz será el nombre de la clase, y todos sus campos se incluiran como elementos (etiquetas) en el XML. Vamos a ver ahora cómo personalizar la forma en la que se obtendrá la representación.

Podemos indicar mediante etiquetas de JAXB si los campos de la clase deben ser representados mediante elementos (`@XmlElement`) o atributos (`@XmlAttribute`) en XML, e incluso de forma opcional podemos especificar el nombre que tendrán en la representación mediante el parámetro `name` de la anotación (por defecto pone el mismo nombre que el campo). Podemos utilizar este atributo para especificar el nombre del elemento raíz también.

```
@XmlRootElement(name="estado")
public class EstadoBean {

   @XmlAttribute(name="valor")
   public String estado = "Idle";

   @XmlAttribute(name="toner")
   public int tonerRestante = 25;

   @XmlElement(name="tarea")
   public List<TareaBean> tareas =
             new ArrayList<TareaBean>();
}
```

En este caso, el XML resultante quedaría de la siguiente forma:

```
<estado valor="Idle" toner="25">
    <tarea>
        <nombre>texto.doc</nombre>
        <estado>imprimiendo...</estado>
        <paginas>13</paginas>
```

```
    </tarea>
    <tarea>
        <nombre>texto2.doc</nombre>
        <estado>en espera...</estado>
        <paginas>5</paginas>
    </tarea>
</estado>
```

Si no se indica lo contrario, por defecto convierte los campos a elementos del XML.

Nota

En caso de que las propiedades no sean públicas, podemos etiquetar los *getters* correspondiente a ellas.

Hemos visto que en las listas el nombre que especificamos en `@XmlElement` se utiliza para nombrar cada elemento de la lista. Si queremos que además se incluya un elemento que envuelva a toda la lista, podemos utilizar la etiqueta `@XmlElementWrapper`:

```
@XmlRootElement(name="estado")
public class EstadoBean {

   @XmlAttribute(name="valor")
   public String estado = "Idle";

   @XmlAttribute(name="toner")
   public int tonerRestante = 25;

   @XmlElementWrapper(name="tareas")
   @XmlElement(name="tarea")
   public List<TareaBean> tareas =
              new ArrayList<TareaBean>();
}
```

En este caso tendremos un XML como el que se muestra a continuación:

```
<estado valor="Idle" toner="25">
    <tareas>
        <tarea>
            <nombre>texto.doc</nombre>
            <estado>imprimiendo...</estado>
            <paginas>13</paginas>
        </tarea>
        <tarea>
            <nombre>texto2.doc</nombre>
            <estado>en espera...</estado>
            <paginas>5</paginas>
        </tarea>
    </tareas>
</estado>
```

Para etiquetar una lista también podemos especificar distintos tipos de elemento según el tipo de objeto contenido en la lista. Por ejemplo, supongamos que en el ejemplo anterior la clase `TareaBean` fuese una clase abstracta que tiene dos posible subclases: `TareaSistemaBean` y `TareaUsuarioBean`. Podríamos especificar una etiqueta distinta para cada elemento de la lista según el tipo de objeto del que se trate con la etiqueta `@XmlElements`, de la siguiente forma:

```
@XmlElementWrapper(name="tareas")
   @XmlElements({
       @XmlElement(name="usuario",type=TareaUsuarioBean.class},
       @XmlElement(name="sistema",type=TareaSistemaBean.class}})
   public List<TareaBean> tareas =
              new ArrayList<TareaBean>();
```

De esta forma podríamos tener un XML como el siguiente:

```
<estado valor="Idle" toner="25">
    <tareas>
        <usuario>
            <nombre>texto.doc</nombre>
            <estado>imprimiendo...</estado>
            <paginas>13</paginas>
        </usuario>
        <sistema>
            <nombre>texto2.doc</nombre>
            <estado>en espera...</estado>
            <paginas>5</paginas>
        </sistema>
    </tareas>
</estado>
```

Hemos visto que por defecto se serializan todos los campos. Si queremos exluir alguno de ellos de la serialización, podemos hacerlo anotándolo con `@XmlTransient`. Como alternativa, podemos cambiar el comportamiento por defecto de la clase etiquetándola con `@XmlAccessorType`. Por ejemplo:

```
@XmlAccessorType(NONE)
@XmlRootElement(name="estado")
public class EstadoBean {

    ...

}
```

En este último caso, especificando como tipo `NONE`, no se serializará por defecto ningún campo, sólo aquellos que hayamos anotado explícitamente con `@XmlElement` o `@XmlAttribute`. Los campos y propiedades (*getters*) anotados con estas etiquetas se serializarán siempre. Además de ellos, también podríamos especificar que se serialicen por defecto todos los campos públicos y los *getters* (`PUBLIC_MEMBER`), todos los *getters* (`PROPERTY`), o todos los campos, ya sean públicos o privados (`FIELD`). Todos los que se serialicen por defecto, sin especificar ninguna etiqueta, lo harán como elemento.

Por último, si nos interesa que toda la representación del objeto venga dada únicamente por el valor de uno de sus campos, podemos etiquetar dicho campo con `@XmlValue`.

4.4.6. Códigos de respuesta

En todos los casos anteriores se estará devolviendo un código de respuesta 200 (`Ok`), junto con el contenido especificado en el tipo de datos utilizado en cada caso. Si devolvemos `void`, el código de respuesta será 204 (`No Content`).

Sin embargo, puede que nos interese tener más control sobre el código de respuesta que

se envía en cada caso. Por ejemplo, cuando con POST se crea un nuevo recurso deberíamos devolver 201 (`Created`). Para tener control sobre este código podemos devolver el tipo `Response`.

```
@GET
@Produces(MediaType.APPLICATION_XML)
public Response getAlumnos() {
  AlumnosBean alumnos = FactoriaDao.getInstance().getAlumnos();
  return Response.ok(alumnos).build();
}

@POST
@Consumes(MediaType.APPLICATION_XML)
public Response addAlumno(AlumnoBean alumno,
                          @Context UriInfo uriInfo) {
  String id = FactoriaDao.getInstance().addAlumno(alumno);
  URI uri = uriInfo.getAbsolutePathBuilder().path("{id}").build(id);
  return Response.created(uri).build();
}
```

Al crear una respuesta con `Response`, podemos especificar una entidad, que podrá ser un objeto de cualquiera de los tipos vistos anteriormente, y que representa los datos a devolver como contenido. Por ejemplo, cuando indicamos `ok(alumnos)`, estamos creando una respuesta con código 200 (`Ok`) y con el contenido generado por nuestro *bean* JAXB `alumnos`. Esto será equivalente a haber devuelto directamente `AlumnoBean` como respuesta, pero con la ventaja de que en este caso podemos controlar el código de estado de la respuesta.

En algunos casos también podemos indicar una URI. Por ejemplo, cuando creamos un nuevo recurso con POST, debemos devolver código 201 (`Created`) con una cabecera `Location` que indique la URL con la que podremos acceder al recurso que acabamos de crear. Por este motivo, cuando creamos una respuesta de tipo `create` nos obliga a proporcionar la `URI` correspondiente al recurso creado. Podemos aprovechar el objeto inyectado `UriInfo` para construir a partir de la URI actual la URI del nuevo recurso añadiendo su identificador a la ruta (a continuación veremos más detalles sobre la inyección de objetos).

En muchas ocasiones nos interesará devolver determinados códigos de respuesta sin contenido cuando sucede algún error. Por ejemplo, si solicitamos un recurso concreto y dicho recurso no existe, deberemos devolver 404 (`Not Found`). Aunque hemos visto que esto se puede hacer con el tipo `Response`, como alternativa también podemos lanzar una excepción de tipo `WebApplicationException` pasando como parámetro del constructor el código de estado a devolver.

```
@GET
@Path("{isbn}")
@Produces(MediaType.APPLICATION_XML)
public AlumnoBean getAlumno(@PathParam("isbn") String isbn) {
    AlumnoBean alumno = FactoriaDao.getInstance().getAlumno(isbn);
    if(alumno==null) {
        throw new WebApplicationException(Status.NOT_FOUND);
    } else {
        return alumno;
    }
```

```
}
```

Podemos encontrar los distintos códigos de estado como elementos de la enumeración `Status`.