7 IMPLEMENTACIONES

En este capítulo se comentan algunos de los Servicios Web que están actualmente funcionando de acuerdo al modelo de arquitectura REST. Además se proporciona un ejemplo para comprender mejor el diseño de un Servicio Web y se incluye un artículo que enumera los principales errores de implementación.

7.1 Implementaciones REST en la Web

Actualmente ya hay implementados algunos servicios Web que tienen un diseño al estilo REST (creando sus propias API), como ejemplos notables están **eBay**, **Amazon** y **Blogger**. Aunque la realidad es que estas implementaciones nunca han sido catalogadas totalmente como REST, porque analizándolas exhaustivamente, algunas violan las restricciones esenciales de REST según las necesidades que tengan.

Uno de los pasos más grandes que se han dado es la creación de **Restlet**, una API que nos permite implementar los Servlets de Java siguiendo el estilo REST. La principal ventaja de Restlets es que permite programar Servicios Web de carácter general, no está diseñado para un Servicio en particular (como ocurre con las API de Amazon o eBay).

A continuación se comentan más profundamente estas implementaciones.

7.1.1 Amazon

Amazon es una empresa muy potente en el campo de la venta online. Se podría decir que fue uno de los pioneros en el uso de Servicios Web REST.

Su funcionamiento se basa en la esencia de REST. Posee una base de datos con todos los detalles de los productos que vende. Cuando un usuario quiere realizar una búsqueda o acceder a la información de determinados productos en particular, accede directamente a los recursos solicitados y no a métodos remotos.

Se puede ver la API asociada a Amazon.com (disponible en `associates.amazon.com`). Permite hacer una petición asociada a la base de datos de Amazon.com e integrar los resultados obtenidos en el sitio. La petición es un HTTP GET normal y el resultado es un documento XML

El hecho de que una empresa tan importante se haya decantado por el uso de REST apoyó bastante al despegue de esta tecnología. El nombre de Amazon ha estado bastante ligado al debate sobre REST que existe en la red. Muchos de los autores que se han expuesto lo han usado como ejemplo en sus artículos.

Sin embargo, existe un aspecto que ha sido bastante criticado sobre la API de Amazon. Solamente usa uno de los cuatro métodos HTTP, que es GET. Por lo que no se puede decir que use todo el potencial de REST. La crítica a Amazon [12] alega que cuando

Amazon necesite realizar Servicios Web más complejos, no le bastará con usar el resto de métodos, sino que necesitará migrar a SOAP.

7.1.2 eBay

eBay es una famosa empresa dedicada a la venta y subasta de productos en la red. Es uno de los "gigantes" de la Web.

En 2004 eBay sacó a la luz una API basada en REST a disposición de los clientes [24]. La interfaz de programación de aplicación REST (REST API) de eBay es una manera sencilla de integrar la funcionalidad de eBay en una página Web o una aplicación. La API REST permite usar una llamada, ***GetSearchResult***, para obtener información sobre los productos de eBay. La información devuelta por la llamada a *GetSearchResult* está en formato XML. Además de usar *GetSearchResult* para obtener información sobre los productos de eBay, se permite el procesamiento que transforma los resultados de la búsqueda usando ficheros XSL que se suban. Se accede al fichero XSL subido usando un link de la página central de desarrolladores REST (http://developer.ebay.com/rest).

Aquí se expone un ejemplo de llamada que usa la API REST. Los parámetros usados como CallName, son parámetros de entrada específicos de la API REST

```
http://rest.api.ebay.com/restapi?CallName=GetSearchResults&RequestToken
=xyz123&RequestUserId=ebayuser&Query=toy%20boat&Schema=1
```

Como se puede observar, esta API viola una de las restricciones de REST. eBay nos ofrece ***GetSearchResult*** que es un método remoto al que se le pasan argumentos. Esta práctica va en contra de REST. eBay debería proporcionar recursos a los que acceder y no métodos. Dicho de otra manera, debería ofrecer nombres y no verbos.

7.1.3 Restlets

Restlets es una API desarrollada en 2006 cuya misión es acercar la simplicidad y eficiencia de REST a los desarrolladores Java. Actualmente se encuentra en la versión 1.0 beta 14.

El autor de *Restlets* es Jérôme Louvel. A continuación se expone un fragmento de un artículo en el cual explica qué es *Restlets* [26].

Cuando empezamos el desarrollo de un sitio Web, queremos cumplir con el estilo de arquitectura **REST** lo máximo posible. Después de investigar bastante, nos dimos cuenta de que existía la carencia de un marco de trabajo REST para Java. El único proyecto que se había llevado a cabo era *1060 NetKernel* desarrollado por *1060*

Research pero contenía demasiadas características para nuestras necesidades y no soportaba correctamente los conceptos de REST.

Esto nos llevó a desarrollar nuestro propio marco de trabajo REST encima de la API Servlet. Llegó a un punto en el que la API Servlet estaba completamente oculta. Hicimos una separación entre la implementación del protocolo HTTP y el soporte para la API Servlet. Al final, estábamos preparados para desarrollar el primer conector Restlet, un conector de servidor HTTP que permitía llamadas uniformes REST.

También queríamos deshacernos de la separación que existe en Java entre la vista de la parte del cliente y la del servidor. En la Web actual no necesitamos hacer esas diferencias. Nadie va a actuar al mismo tiempo como cliente y servidor Web. En REST, cada componente puede tener tantos conectores de cliente y servidor como necesite, por lo que simplemente desarrollamos un conector HTTP basado en la clase URLConnection. Por supuesto, se pueden proporcionar otras implementaciones, como una basada en *Yakarta Commons HTTP Client*.

Después de varios intentos, estuvo claro que sería beneficioso para los desarrolladores separar el proyecto Restlet en dos partes.

- La primera parte es un conjunto genérico de interfaces llamada ***Restlet API***, que incluye algunas clases de ayuda y mecanismos para registrar una implementación Restlet.

- La segunda parte es una referencia a la implementación, llamada ***Noelios Restlet Engine***, que incluye un conector de servidor HTTP, conectores de clientes HTTP, JDBC y SMTP, un conjunto de representaciones basadas en cadenas, ficheros, streams, channels o FreeMarker, y un directorio Restlet que puede servir ficheros estáticos de un árbol de directorios con negociación automática del contenido basada en las extensiones de los ficheros.

Los desarrolladores de Java necesitan empezar a pensar a la manera de REST cuando desarrollen nuevos Servicios Web. EL proyecto *Restlets* proporciona una manera simple pero a la vez sólida para que comiencen con buen pie en la Web 2.0.

7.2 Ejemplo de uso de REST

A continuación se va a exponer un ejemplo de Roger L. Costello [9] para explicar el uso de REST.

Parts Depot, Inc (una compañía ficticia) ha desarrollado un Servicios Web para permitir a sus clientes;

- Conseguir una lista de partes (productos)
- Conseguir información detallada de una parte en particular
- Emitir una orden de compra (Purchase Order, PO)

Estudiaremos como se implementa cada una de esas partes en REST.

7.2.1 Conseguir una lista de las partes

El servicio Web hace disponible una URL para un recurso de lista de partes. Por ejemplo, un cliente podría usar esta URL para conseguir la lista de partes:

```
http://www.parts-depot.com/parts
```

Debemos darnos cuenta de que la manera en que el servicio Web genera la lista de partes es completamente transparente al cliente. Todos los clientes conocen que si acceden a esa URL, entonces se les devuelve un documento que contiene la lista de las partes. Como la implementación es transparente a los usuarios, Parts Depot es libre de modificar la implementación subyacente de este recurso sin que esto tenga un impacto en los clientes.

Este es el documento que recibe el cliente:

```
<?xml version="1.0"?>
<p:Parts xmlns:p="http://www.parts-depot.com"
        xmlns:xlink="http://www.w3.org/1999/xlink">
    <Part id="00345" xlink:href="http://www.parts-depot.com/parts/00345"/>
    <Part id="00346" xlink:href="http://www.parts-depot.com/parts/00346"/>
    <Part id="00347" xlink:href="http://www.parts-depot.com/parts/00347"/>
    <Part id="00348" xlink:href="http://www.parts-depot.com/parts/00348"/>
</p:Parts>
```

Estamos asumiendo que por medio de la negociación, el servicio ha determinado que el cliente quiere una representación en XML (por un proceso máquina-máquina). Debemos darnos cuenta de que la lista de partes contiene links para conseguir información detallada sobre cada parte. Esto es una característica clave de REST. El cliente se transfiere de un estado a otro examinando y eligiendo de entre las URL alternativas que se entregan en el documento que se ha obtenido como respuesta.

7.2.2 Conseguir datos detallados de una parte

El servicio Web hace disponible una URL para cada recurso parte. Un cliente solicita por ejemplo la parte 00345 de la siguiente manera:

```
http://www.parts-depot.com/parts/00345
```

El documento que recibe el cliente como respuesta a esa petición, es el siguiente:

```
<?xml version="1.0"?>
<p:Part xmlns:p="http://www.parts-depot.com"
       xmlns:xlink="http://www.w3.org/1999/xlink">
     <Part-ID>00345</Part-ID>
     <Name>Widget-A</Name>
     <Description>This part is used within the frap assembly</Description>
     <Specification xlink:href="http://www.parts-depot.com/parts/00345/specification"/>
     <UnitCost currency="USD">0.10</UnitCost>
     <Quantity>10</Quantity>
</p:Part>
```

De nuevo, debemos observar cómo estos datos enlazan a más datos. La especificación para estos datos puede encontrarse a través del hiperlink. Cada documento de respuesta permite al cliente seguir explorando para conseguir información más detallada.

7.2.3 Emitir una PO (orden de compra)

El servicio Web hace disponible una URL para emitir una PO. El cliente crea un documento de instancia de PO que es conforme al esquema de PO que Parts Depot ha diseñado (y ha hecho público en un documento WSDL). El cliente envía Po.xml como la carga de un mensaje HTTP POST.

El servicio de PO responde al HTTP POST con una URL a la PO enviada. Así, el cliente puede recuperar la PO en cualquier momento posterior (para actualizarla o editarla). La PO se ha convertido en un fragmento de información que se comparte entre el cliente y el servidor. A la información compartida (PO), el servidor le asigna una dirección (URL), y se expone como un servicio Web.

7.3 Errores comunes a la hora de diseñar con REST

En este apartado se incluye un artículo de Paul Prescod [10] en el que se enumeran los principales **errores** a la hora de implementar Servicios Web al estilo **REST**.

Cuando se diseña el primer sistema REST, hay varios errores que la gente comete a menudo. En este artículo se van a resumir para que la gente pueda evitarlos.

1. **Usar HTTP no es suficiente**. Algunas personas usan HTTP en un servicio Web sin SOAP o XML-RPC y hacen el equivalente lógico de SOAP o XML-RPC. Si vamos a usar mal HTTP deberíamos dejarlo y hacerlo de una manera estándar.

2. **No sobreusar POST**. POST es en algún sentido el método más flexible de HTTP. Tiene una definición más débil que los otros métodos y soporta el envío y recepción de información al mismo tiempo. Por tanto, hay una tendencia a querer usar POST para todo. En el primer servicio Web que creemos, sólo deberíamos usar POST cuando estemos creando una nueva URI. Cuando vayamos cogiendo soltura, puede que decidamos usar POST para otra clase de mutaciones en un recurso. Una regla útil es preguntarnos si estamos usando POST para hacer algo que realmente es un GET, DELETE o PUT, o puede ser descompuesto como una combinación de esos métodos.

3. **No depender de la estructura interna de las URI**. Algunas personas piensan en el diseño de REST en términos de configuración de un conjunto de URIs. "Pondré las ordenes de compra en /compras y les daré a todas ellas números como /compras/12132 y los documentos de clientes estarán en /clientes…" Esto puede ser una manera útil de pensar si estamos haciendo un borrador, pero no debería ser nuestra interfaz pública final para el servicio. De acuerdo con los principios de la arquitectura Web, la mayoría de las URIs son opacas para el software del cliente la mayor parte del tiempo. En otras palabras, nuestra API pública no debería depender de la estructura de nuestras URIs. En vez de eso, debería haber un único fichero XML que apunte a los componentes de nuestro servicio. Esos componentes deberían tener hiperlinks que apunten a otros componentes y así sucesivamente. Por tanto, podemos introducir a la gente a nuestro servicio con una única URI y podemos distribuir los actuales componentes a través de ordenadores y dominios siempre que queramos. Una regla es que los clientes sólo "fabrican" URIs cuando están construyendo peticiones para buscar datos (usando cadenas de caracteres para formarla). Esas peticiones devuelven referencias a objetos con URIs opacas.

4. **No poner acciones en URIs.** Esto surge naturalmente del punto anterior. Un abuso de las URIs es tener cadenas de búsqueda como "someuri?action=delete". Primeramente, estaríamos usando GET para realizar una operación que no es segura. Segundo, no hay una relación formal entre esta "acción URI" y el objeto "URI". También hay que resaltar que nuestra decisión de convenir "action=" es algo específico para nuestra aplicación. REST trata de conducir fuera del protocolo todas las "convenciones de aplicación" que pueda.

5. **Los servicios son raramente recursos.** En el diseño REST, un servicio que nos proporcione una cuota que indica el stock de unos determinados artículos no es muy interesente. En vez de eso, en un diseño REST deberíamos tener recursos "stock" y un servicio sería un índice de los recursos "stock".

6. **Las sesiones son irrelevantes.** No debería haber necesidad alguna de que un cliente haga el proceso de "login" o comience una conexión. La autenticación HTTP se realiza automáticamente en cada mensaje. Las aplicaciones cliente son consumidoras de recursos, no de servicios. Por tanto, no hay nada sobre lo que

realizar el proceso de "login". Si estamos reservando un billete de avión en un servicio Web REST, no creamos una nueva conexión de sesión para el servicio. Más bien preguntamos por el objeto creador del itinerario para crear un nuevo itinerario. Podemos comenzar rellenando un formulario y después conseguir algunos componentes totalmente diferentes de alguna parte de la red para seguir rellenando otros formularios. No hay sesión, por lo que no existen problemas de migración del estado de la sesión entre clientes, Tampoco hay problemas de "afinidad de sesión" en el servidor.

7. **No inventar identificadores de objetos propietarios**. Debemos usar URIs. Las URIs son importantes porque siempre podemos asociar información con ellas de dos maneras. El método más simple es poner datos en un servidor de manera que la URI pueda ser dereferenciada para conseguir los datos. Debemos darnos cuenta de que esta técnica sólo funciona con URIs que puedan ser derreferenciadas, por lo que esas URIs (HTTP URIs) son bastante más recomendables que el uso de URIs basadas en URN o UUID. Otra manera es usar RDF y otras técnicas para que nos permitan proyectar metadatos en una URI que puede que no esté bajo nuestro control. Si usamos la sintaxis URI con UUID o algo parecido, entonces solamente obtendremos la mitad de los beneficios de las URIs. Tendremos una sintaxis estandarizada pero tendremos una capacidad de dereferenciar que no está estandarizada. Si usamos URIs HTTP obtendremos la otra mitad de los beneficios porque obtendremos un mecanismo estandarizado para dereferenciar.

8. **No debemos preocuparnos por la independencia del protocolo.** Sólo existe un protocolo que soporta una semántica correcta para la manipulación de recursos. Si en el futuro surge otro, será fácil mantener el mismo diseño y simplemente soportar una interfaz de protocolo distinta. Por otra parte, lo que normalmente entiende la gente por "independencia del protocolo" es abandonar el modelo de recursos y por tanto, abandonar REST y la Web.

Lo más importante de todo es tener en la mente que REST trata de exponer recursos por medio de URIs, no servicios a través de interfaces de mensajes.

7.4 Conclusiones

En este capítulo se han expuesto algunos de los Servicios Web que están actualmente funcionando de acuerdo al modelo de arquitectura REST. Además se han proporcionado un ejemplo para comprender mejor el diseño de un Servicio Web y un artículo que enumera los principales errores de implementación.. En el siguiente capítulo se hará un seguimiento de la relevancia que ha tenido REST en la Web desde que salió a la luz. Además se intentará dar una previsión del futuro que le espera a REST.