Arquitectura para servicios REST

Este documento contiene la definición clásica de una arquitectura de servicios REST. **Describe en detalle la tecnología** y presenta con cierto nivel de profundidad los aspectos fundamentales de la misma. No está enfocada a atender las necesidades de un proyecto particular y debe ser utilizada como guía o referente en cada proyecto que emplee este tipo de arquitectura.

También incorpora un conjunto de lineamientos generales para el **diseño de las URL asociadas a cada servicio REST**.

Esta guía se creó pensando en la independencia de lenguaje, dado que la técnica se puede implementar para prácticamente cualquiera.

La Transferencia de Estado Representacional (REST - Representational State Transfer) es una alternativa más simple a la de SOAP y a los servicios web basados en el Lenguaje de Descripción de Servicios Web (Web Services Descripcion Language - WSDL). Grandes proveedores de Web 2.0 están migrando a esta tecnología, incluyendo a Yahoo, Google y Facebook, quienes marcaron como obsoletos a sus servicios SOAP y WSDL y pasaron a usar un modelo más fácil de usar, orientado a los recursos.

Presentando REST

REST define un conjunto de principios arquitectónicos con los cuales se diseñan servicios web haciendo énfasis en los recursos del sistema, incluyendo el cómo se accede al estado de dichos recursos y el cómo se transfieren por HTTP hacia clientes que están escritos en diversos lenguajes. REST se posicionó en los últimos años como el modelo predominante para el diseño de servicios. De hecho, REST logró un impacto tan grande en la web que prácticamente logró desplazar a SOAP y las interfaces basadas en WSDL por tener un estilo bastante más simple de usar.

El impacto de REST no fue grande, cuando Roy Fielding lo presentó por primera vez en el año 2000 en la Universidad de California, durante la charla "Estilos de Arquitectura y el Diseño de Arquitecturas de Software basadas en Redes", la cual analizaba un conjunto de principios arquitectónicos de software para usar a la Web como una plataforma de Procesamiento Distribuido. Ahora, años después de su presentación, comienzan a aparecer varios frameworks REST y se convirtió en una parte integral de Java 6 a través de JSR-311.

Autor: Gustavo Adolfo Arellano Sandoval (arellano.gustavo@gmail.com)

Los 4 principios de REST

Una implementación concreta de un servicio web REST sigue cuatro principios de diseño fundamentales:

- utiliza los métodos HTTP de manera explícita
- no mantiene estado
- expone URIs con forma de directorios
- transfiere XML, JavaScript Object Notation (JSON), o ambos

A continuación vamos a ver en detalle estos cuatro principios, y explicaremos porqué son importantes a la hora de diseñar un servicio web REST.

REST utiliza los métodos HTTP de manera explícita

Una de las características fundamentales de los servicios web REST es el uso explícito de los métodos HTTP, siguiendo el protocolo definido por RFC 2616. Por ejemplo, HTTP GET se define como un método productor de datos, cuyo uso está pensado para que las aplicaciones cliente obtengan recursos, busquen datos de un servidor web, o ejecuten una consulta esperando que el servidor web la realice y devuelva un conjunto de recursos.

REST hace que los desarrolladores usen los métodos HTTP explícitamente de manera que resulte consistente con la definición del protocolo. Este principio de diseño básico establece una asociación uno-a-uno entre las operaciones de crear, leer, actualizar y borrar y los métodos HTTP. De acuerdo a esta asociación:

- se usa POST para crear un recurso en el servidor
- se usa GET para obtener un recurso
- se usa PUT para cambiar el estado de un recurso o actualizarlo
- se usa DELETE para eleminar un recurso

Una falla de diseño poco afortunada que tienen muchas APIs web es el uso de métodos HTTP para otros propósitos. Por ejemplo, la petición del URI en un pedido HTTP GET, en general identifica a un recurso específico. O el string de consulta en el URI incluye un conjunto de parámetros que definen el criterio de búsqueda que usará el servidor para encontrar un conjunto de recursos. Al menos, así como el RFC HTTP/1.1 describe al GET.

Pero hay muchos casos de APIs web poco elegantes que usan el método HTTP GET para ejecutar algo transaccional en el servidor; por ejemplo, agregar registros a una base de datos. En estos casos, no se utiliza adecuadamente el URI de la petición HTTP, o al menos no se usa "a la manera REST". Si el API web utiliza GET para invocar un procedimiento remoto, seguramente se verá algo como esto:

```
GET /agregarusuario?nombre=Zim HTTP/1.1
```

Autor: Gustavo Adolfo Arellano Sandoval (arellano.gustavo@gmail.com)

Este no es un diseño muy atractivo porque ese método expone una operación que cambia estado sobre un método HTTP GET. Dicho de otra manera, la anterior petición HTTP GET tiene efectos secundarios. Si se procesa con éxito, el resultado de la petición es agregar un usuario nuevo (en el ejemplo, Zim) a la base de datos. El problema es básicamente semántico. Los servidores web están diseñados para responder a las peticiones HTTP GET con la búsqueda de recursos que concuerden con la ruta (o el criterio de búsqueda) en el URI de la petición, y devolver estos resultados o una representación de los mismos en la respuesta, y no añadir un registro a la base de datos. Desde el punto de vista del protocolo, y desde el punto de vista de servidor web compatible con HTTP/1.1, este uso del GET es inconsistente.

Más allá de la semántica, el otro problema con el GET es que al ejecutar eliminaciones, modificaciones o creación de registros en la base de datos, o al cambiar el estado de los recursos de cualquier manera, provoca que las herramientas de caché web y los motores de búsqueda (crawlers) puedan realizar cambios no intencionales en el servidor. Una forma simple de evitar este problema es mover los nombres y valores de los parámetros en la petición del URI a tags XML. Los tags resultantes, una representación en XML de la entidad a crear, pueden ser enviados en el cuerpo de un HTTP POST cuyo URI de petición es el padre de la entidad.

Antes:

```
GET /agregarusuario?nombre=Zim HTTP/1.1
```

Después:

```
POST /usuarios HTTP/1.1
Host: miservidor
Content-type: application/xml
<usuario>
   <nombre>Zim</nombre>
</usuario>
```

El método anterior es un ejemplo de una petición REST: hay un uso correcto de HTTP POST y la inclusión de los datos en el cuerpo de la petición. Al recibir esta petición, la misma puede ser procesada para que pueda agregar el recurso contenido en el cuerpo como un subordinado del recurso identificado en el URI de la petición; en este caso el nuevo recurso debería agregarse como hijo de /usuarios. Esta relación de contención entre la nueva entidad y su padre, como se indica en la petición del POST, es análoga a la forma en la que está subordinado un archivo a su directorio. El cliente indica esta relación entre la entidad y su padre y define el nuevo URI de la entidad en la petición del POST.

Luego, una aplicación cliente puede obtener una representación del recurso usando la nueva URI, sabiendo que al menos lógicamente el recurso se ubica bajo /usuarios

```
GET /usuarios/Zim HTTP/1.1
Host: miservidor
```

```
Accept: application/xml
```

Es explícito el uso del GET de esta manera, ya que el GET se usa solamente para recuperar datos. GET es una operación que no debe tener efectos secundarios, una propiedad también conocida como *idempotencia*.

Se debe hacer un refactor similar de un método web que realice actualizaciones a través del método HTTP GET. El siguiente método GET intenta cambiar la propiedad "nombre" de un recurso. Si bien se puede usar el string de consulta para esta operación, evidentemente no es el uso apropiado y tiende a ser problemático en operaciones más complejas. Ya que nuestro objetivo es hacer uso explícito de los métodos HTTP, un enfoque REST sería enviar un HTTP PUT para actualizar el recurso, en vez de usar HTTP GET.

Antes:

```
GET /actualizarusuario?nombre=Zim&nuevoNombre=Dib HTTP/1.1
```

Después:

```
PUT /usuarios/Zim HTTP/1.1
Host: miservidor
Content'Type: application/xml
<usuario>
   <nombre>Dib</nombre>
</usuario>
```

Al usar el método PUT para reemplazar al recurso original se logra una interfaz más limpia que es consistente con los principios de REST y con la definición de los métodos HTTP. La petición PUT que se muestra arriba es explícita en el sentido que apunta al recurso a ser actualizado identificándolo en el URI de la petición, y también transfiere una nueva representación del recurso del cliente hacia el servidor en el cuerpo de la petición PUT, en vez de transferir los atributos del recurso como un conjunto suelo de parámetros (nombre = valor) en el mismo URI de la petición.

El PUT mostrado también tiene el efecto de renombrar al recurso Zim a Dib, y al hacerlo cambia el URI a */usuarios/Dib*. En un servicio web REST, las peticiones siguientes al recurso que apunten a la URI anterior van a generar un error estándar "404 Not Found".

Como un principio de diseño general, ayuda seguir las reglas de REST que aconsejan usar sustantivos en vez de verbos en las URIs. **En los servicios web REST, los verbos están claramente definidos por el mismo protocolo: POST, GET, PUT y DELETE**. Idealmente, para mantener una interfaz general y para que los clientes puedan ser explícitos en las operaciones que invocan, **los servicios web no deberían definir más verbos o procedimientos remotos, como ser */agregarusuario* y */actualizarusuario***. Este principio de diseño también aplica para el cuerpo de la petición HTTP, el cual debe usarse para transferir el estado de un recurso, y no para llevar el nombre de un método remoto a ser invocado.

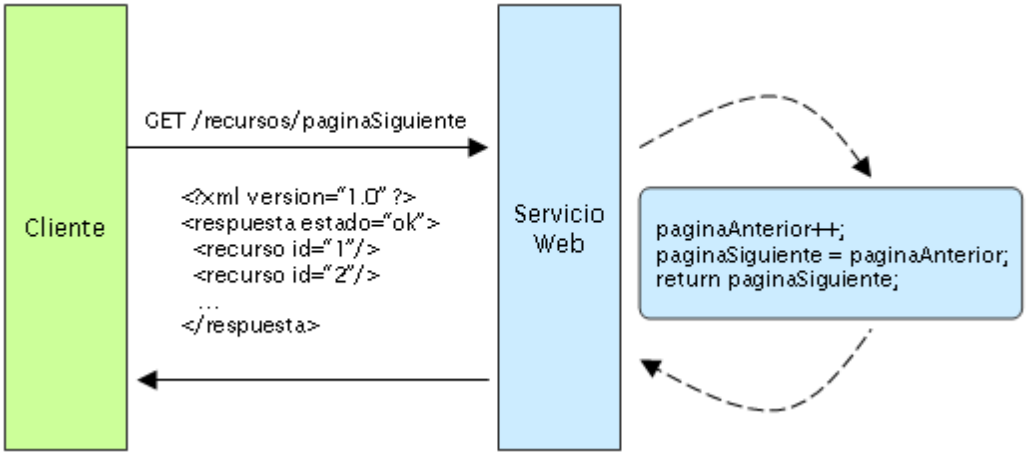
REST no mantiene estado

Los servicios web REST necesitan escalar para poder satisfacer una demanda en constante crecimiento. Se usan clusters de servidores con balanceadores de carga y alta disponibilidad, proxies, y gateways de manera de conformar una topología de servicios, que permita transferir peticiones de un equipo a otro para disminuir el tiempo total de respuesta de una invocación al servicio web. El uso de servidores intermedios para mejorar la escalabilidad hace necesario que los clientes de servicios web REST envíen peticiones completas e independientes; es decir, se deben enviar peticiones que incluyan todos los datos necesarios para cumplir el pedido, de manera que los componentes en los servidores intermedios puedan redireccionar y gestionar la carga sin mantener el estado localmente entre las peticiones.

Una petición completa e independiente hace que el servidor no tenga que recuperar ninguna información de contexto o estado al procesar la petición. Una aplicación o cliente de servicior web REST debe incluir dentro del encabezado y del cuerpo HTTP de la petición todos los parámetros, contexto y datos que necesita el servidor para generar la respuesta. De esta manera, el no mantener estado mejora el rendimiento de los servicios web y simplifica el diseño e implementación de los componentes del servidor, ya que la ausencia de estado en el servidor elimina la necesidad de sincronizar los datos de la sesión con una aplicación externa.

Servicios con estado vs. sin estado

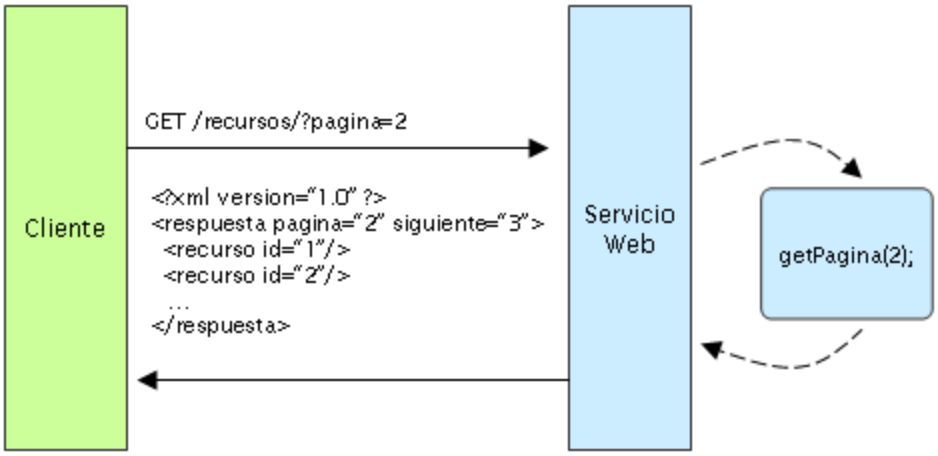
La siguiente ilustración nos muestra un **servicio con estado**, del cual una aplicación realiza peticiones para la página siguiente en un conjunto de resultados múlti-página, asumiendo que el servicio mantiene información sobre la última página que pidió el cliente. En un diseño con estado, el servicio incrementa y almacena en algún lugar una variable *paginaAnterior* para poder responder a las peticiones siguientes.



Los servicios con estado tienden a volverse complicados. En la plataforma Java Enterprise Edition (Java EE), un entorno de servicios con estado necesita bastante análisis y diseño

desde el inicio para poder almacenar los datos eficientemente y poder sincronizar la sesión del cliente dentro de un cluster de servidores. En este tipo de ambientes, ocurre un problema que le resulta familiar a los desarrolladores de servlets/JSP y EJB, quienes a menudo tienen que resolver buscando la causa de una *java.io.NotSerializableException* cuando ocurre la replicación de una sesión. Puede ocurrir tanto en el contenedor de Servlets al intentar replicar la *HttpSession* o en el contenedor de EJB al replicar un EJB con estado; en todos los casos, es un problema que puede costar mucho esfuerzo resolver, buscando el objeto que no implementa *Serializable* dentro de un grafo complejo de objetos que constituyen el estado del servidor. Además, la sincronización de sesiones es costosa en procesamiento, lo que impacta negativamente en el rendimiento general del servidor.

Por otro lado, los **servicios sin estado** son mucho más simples de diseñar, escribir y distribuir a través de múltiples servidores. Un servicio sin estado no sólo funciona mejor, sino que además mueve la responsabilidad de mantener el estado al cliente de la aplicación. En un servicio web REST, el servidor es responsable de generar las respuestas y proveer una interfaz que le permita al cliente mantener el estado de la aplicación por su cuenta. Por ejemplo, en el mismo ejemplo de una petición de datos en múltiples páginas, **el cliente debería incluir el número de página a recuperar** en vez de pedir "la siguiente", tal como se muestra en la siguiente figura:



Un servicio web sin estado genera una respuesta que se enlaza a la siguiente página del conjunto y le permite al cliente hacer todo lo que necesita para almacenar la página actual. Este aspecto del diseño de un servicio web REST puede descomponerse en dos conjuntos de responsabilidades, como una separación de alto nivel que deja claro cómo puede mantenerse un servicio sin estado.

Responsabilidad del servidor

- Genera respuestas que incluyen enlaces a otros recursos para permitirle a la aplicación navegar entre los recursos relacionados. Este tipo de respuestas tiene enlaces embebidos. De la misma manera, si la petición es hacia un padre o un recurso contenedor, entonces una respuesta REST típica debería también incluir enlaces hacia

los hijos del padre o los recursos subordinados, para que de manera, que se mantengan conectados.

- Genera respuestas que indican si son susceptibles de ser guardados en caché o no, para mejorar el rendimiento al reducir la cantidad de peticiones para recursos duplicados, y para eliminar algunas peticiones totalmente. El servidor utiliza los atributos Cache-Control y Last-Modified de la cabecera en la respuesta HTTP para indicarlo.

Responsabilidades del cliente de la aplicación

- Utiliza el atributo Cache-Control del encabezado de la respuesta para determinar si debe cachear el recurso (es decir, hacer una copia local del mismo) o no. El cliente también lee el atributo Last-Modified y envía la fecha en el atributo If-Modified-Since del encabezado para preguntarle al servidor si el recurso cambió desde entonces. Esto se conoce como *GET Condicional*, y ambos encabezados van de la mano con la respuesta del servidor 304 (No Modificado) y se omite al recurso que se había solicitado si no hubo cambios desde esa fecha. Una respuesta HTTP 304 significa que el cliente puede seguir usando la copia local de manera segura, evitando así realizar las peticiones GET hasta tanto el recurso no cambie.
- Envía peticiones completas que pueden ser serviciadas en forma independiente a otras peticiones. Esto implica que el cliente hace uso completo de los encabezados HTTP tal como está especificado por la interfaz del servicio web, y envía las representaciones del recurso en el cuerpo de la petición. El cliente envía peticiones que hacen muy pocas presunciones sobre las peticiones anteriores, la existencia de una sesión en el servidor, la capacidad del servidor para agregarle contexto a una petición, o sobre el estado de la aplicación que se mantiene entre las peticiones.

Esta colaboración entre el cliente y el servicio es esencial para crear un servicio web REST sin estado. Mejora el rendimiento, ya que ahorro ancho de banda y minimiza el estado de la aplicación en el servidor.

REST expone URIs con forma de directorios

Desde el punto de vista del cliente de la aplicación que accede a un recurso, la URI determina qué tan intuitivo va a ser el web service REST, y si el servicio va a ser utilizado tal como fue pensado al momento de diseñarlo. La tercera característica de los servicios web REST es justamente sobre las URIs.

Las URI de los servicios web REST deben ser intuitivas, hasta el punto de que sea fácil adivinarlas. Pensemos en las URI como una interfaz auto-documentada que necesita de muy poca o ninguna explicación o referencia para que un desarrollador pueda comprender a lo que apunta, y a los recursos derivados relacionados.

Una forma de lograr este nivel de usabilidad es definir URIs con una estructura al estilo de

los directorios. Este tipo de URIs es jerárquica, con una única ruta raíz, y va abriendo ramas a través de las subrutas para exponer las áreas principales del servicio. De acuerdo a esta definición, una URI no es solamente una cadena de caracteres delimitada por barras, sino más bien un árbol con subordinados y padres organizados como nodos. Por ejemplo, en un servicio de hilos de discusiones que tiene temas varados, se podría definir una estructura de URIs como esta:

```
http://www.miservicio.org/discusion/temas/{tema}
```

La raíz, */discusión*, tiene un nodo */temas* como hijo. Bajo este nodo hay un conjunto de nombres de temas (como ser tecnología, actualidad, y más), cada uno de los cuales apunta a un hilo de discusión. Dentro de esta estructura, resulta fácil recuperar hilos de discusión al teclear algo después de */temas/*.

En algunos casos, la ruta a un recurso encaja muy bien dentro de la idea de "estructura de directorios". Por ejemplo, tomemos algunos recursos organizados por fecha, que son muy prácticos de organizar usando una sintaxis jerárquica.

El siguiente ejemplo es intuitivo porque está basado en reglas:

```
http://www.miservicio.org/discusion/2008/12/23/{tema}
```

El primer fragmento de la ruta es un año de cuatro dígitos, el segundo fragmento es el mes de dos dígitos, y el tercer fragmento es el día de dos dígitos. Puede resultar un poco tonto explicarlo de esta manera, pero es justamente el nivel de simpleza que buscamos. Tanto humanos como máquinas pueden generar estas estructuras de URI porque están basadas en reglas. Como vemos, es fácil llenar las partes de esta URI, ya que existe un patrón para crearlas:

```
http://www.miservicio.org/discusion/{año}/{mes}/{dia}/{tema}
```

Podemos también enumerar algunas guías generales más al momento de crear URIs para un servicio web REST:

- ocultar la tecnología usada en el servidor que aparecería como extensión de archivos (.jsp, .php, .asp), de manera de poder portar la solución a otra tecnología sin cambiar las URI.
- mantener todo en minúsculas.
- sustituir los espacios con guiones: “-” (no guiones bajos).
- Devolver códigos 2xx (como 200, 203, etc) para indicar el éxito de una operación.
- Devolver códigos 3xx (como 300, 306, 308, etc) para indicar una redirección.
- Devolver códigos 4xx (como 404, etc) para indicar un caso excepcional originado por el usuario.
- Devolver códigos 5xx (como 500, 505, etc) para indicar un caso excepcional originado por el servidor

Autor: Gustavo Adolfo Arellano Sandoval (arellano.gustavo@gmail.com)

- Usar como Guía la lista completa de códigos de respuesta HTTP provista como apéndice a este docuemnto. (HTTP-CODES.pdf)

Las URI deberían ser estáticas de manera que cuando cambie el recurso o cambie la implementación del servicio, el enlace se mantenga igual. Esto permite que el cliente pueda generar "favoritos" o bookmarks. También es importante que la manera de asociar que existirá entre los recursos que están expuestos en las URI, mantenga independencia, con respecto a las relaciones que existen en el almacén de persistencia, en términos de tales recursos.

REST transfiere XML, JSON, o ambos

La representación de un recurso en general refleja el estado actual del mismo y sus atributos al momento en que el cliente de la aplicación realiza la petición. La representación del recurso es una simple "foto" en el tiempo de éste. Podría ser una representación de un registro de la base de datos que consiste en la asociación entre columnas y tags XML, donde los valores de los elementos en el XML contienen los valores de las filas. O, si el sistema tiene un modelo de datos, la representación de un recurso es una fotografía de los atributos de una de los elementos al interior del modelo de datos del sistema.

Una restricción mas al momento de diseñar un servicio web REST tiene que ver con el formato de los datos que la aplicación y el servicio intercambian en las peticiones/respuestas. Aquí es donde realmente vale la pena mantener las cosas simples, legibles por humanos, y conectadas.

Los objetos del modelo de datos generalmente se relacionan de alguna manera, y las relaciones entre los objetos del modelo de datos (los recursos) deben reflejarse en la forma en la que se representan al momento de transferir los datos al cliente. Por ejemplo, en un servicio de “hilos de discusión”, una forma de representación de un recurso conectado podría ser un “tema de discusión raíz con todos sus atributos, y links embebidos a las respuestas al tema”.

```
<discusion fecha="{fecha}" tema="{tema}">
    <comentario>{comentario}</comentario>
    <respuestas>
          <respuesta de="gaz@mail.com" href="/discusion/temas/{tema}/gaz"/>
          <respuesta de="gir@mail.com" href="/discusion/temas/{tema}/gir"/>
    </respuestas>
</discusion>
```

Por último, es bueno construir los servicios de manera que usen el atributo “HTTP Accept” del encabezado, en donde el valor de este campo es un tipo MIME. De esta manera, los clientes pueden pedir por un contenido en particular que mejor pueden analizar. Algunos de los tipos MIME más usados para los servicios web REST son:

MIME-Type	Content-Type
JSON	application/json
XML	application/xml
XHTML	application/xhtml+xml

Esto permite que el servicio sea consumido por distintos clientes escritos en diferentes lenguajes, corriendo en diversas plataformas y dispositivos. El uso de los tipos MIME y del encabezado HTTP es un mecanismo conocido como *negociación de contenido*, el cual le permite a los clientes elegir qué formato de datos puedan leer, y minimiza el acoplamiento de datos entre el servicio y las aplicaciones