Transferencia de Estado Representacional

La **transferencia de estado representacional** (en inglés *representational state transfer*) o **REST** es un estilo de arquitectura *software* para sistemas hipermedia distribuidos como la World Wide Web. El término se originó en el año 2000, en una tesis doctoral sobre la web escrita por Roy Fielding, uno de los principales autores de la especificación del protocolo HTTP y ha pasado a ser ampliamente utilizado por la comunidad de desarrollo.

Índice

Historia

La Web empezó a ser de uso diario en 1993-4, cuando empezaron a estar disponibles sitios web para uso general. En ese momento, solo había una descripción fragmentada de la arquitectura web y había presión en la industria para acordar algún estándar para los protocolos de interfaz web. Por ejemplo, se habían agregado varias extensiones experimentales al protocolo de comunicación (HTTP) para admitir proxies y se estaban proponiendo más extensiones, pero existía la necesidad de una arquitectura web formal con la que evaluar el impacto de estos cambios. [1]

Juntos, los grupos de trabajo del W3C y del IETF comenzaron a trabajar en la creación de descripciones formales de los tres estándares principales de la web: URI, HTTP y HTML. Roy Fielding estuvo involucrado en la creación de estos estándares (específicamente HTTP 1.0 y 1.1, y URI), y durante los siguientes seis años desarrolló el estilo arquitectónico REST, probando sus restricciones en los estándares de protocolo de la web y usándolo como un medio para definir mejoras arquitectónicas y para identificar desajustes arquitectónicos. Fielding definió REST en su disertación de doctorado de 2000 "Estilos arquitectónicos y el diseño de arquitecturas de software basadas en redes" en UC Irvine.

Descripción

Si bien el término *REST* se refería originalmente a un conjunto de principios de arquitectura —descritos más abajo—, en la actualidad se usa en el sentido más amplio para describir cualquier interfaz entre sistemas que utilice directamente HTTP para obtener datos o indicar la ejecución de operaciones sobre los datos, en cualquier formato (XML, JSON, etc) sin las abstracciones adicionales de los protocolos basados en patrones de intercambio de mensajes, como por ejemplo SOAP. Es posible diseñar sistemas de servicios web de acuerdo con el estilo arquitectural REST de Fielding y también es posible diseñar interfaces XMLHTTP de acuerdo con el estilo de llamada a procedimiento remoto (RPC), pero sin usar SOAP. Estos dos usos diferentes del término *REST* causan cierta confusión en las discusiones técnicas, aunque RPC no es un ejemplo de REST.

Roy Fielding charlando durante OSCON 2008

REST afirma que la web ha disfrutado de escalabilidad como resultado de una serie de diseños fundamentales clave:

- Un **protocolo cliente/servidor sin estado**: cada mensaje HTTP contiene toda la información necesaria para comprender la petición. Como resultado, ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre mensajes. Sin embargo, en la práctica, muchas aplicaciones basadas en HTTP utilizan cookies y otros mecanismos para mantener el estado de la sesión (algunas de estas prácticas, como la reescritura de URLs, no son permitidas por REST)
- Un conjunto de **operaciones bien definidas** que se aplican a todos los *recursos* de información: HTTP en sí define un conjunto pequeño de operaciones, las más importantes son **POST**, **GET**, **PUT** y **DELETE**. Con frecuencia estas operaciones se equiparan a las operaciones CRUD en bases de datos (CLAB en castellano: crear,leer,actualizar,borrar) que se requieren para la persistencia de datos, aunque POST no encaja exactamente en este esquema.
- Una **sintaxis universal** para identificar los recursos. En un sistema REST, cada recurso es direccionable únicamente a través de su URI.
- El **uso de hipermedios**, tanto para la información de la aplicación como para las transiciones de estado de la aplicación: la representación de este estado en un sistema REST son **típicamente HTML o XML**. Como resultado de esto, es posible navegar de un recurso REST a muchos otros, simplemente siguiendo enlaces sin requerir el uso de registros u otra infraestructura adicional.

Recursos

Un concepto importante en REST es la existencia de *recursos* (elementos de información), que pueden ser accedidos utilizando un identificador global (un Identificador Uniforme de Recurso). Para manipular estos recursos, los *componentes* de la red (clientes y servidores) se comunican a través de una interfaz estándar (HTTP) e intercambian *representaciones* de estos recursos (los ficheros que se descargan y se envían) - es cuestión de debate, no obstante, si la distinción entre *recursos* y sus *representaciones* es demasiado platónica para su uso práctico en la red, aunque es popular en la comunidad RDF.

La petición puede ser transmitida por cualquier número de *conectores* (por ejemplo clientes, servidores, cachés, túneles, etc.) pero cada uno lo hace sin "ver más allá" de su propia petición (lo que se conoce como *stateless* (sin estado), otra restricción de REST, que es un principio común con muchas otras partes de la arquitectura de redes y de la información). Así, una aplicación puede interactuar con un recurso conociendo el identificador del recurso y la acción requerida, no necesitando conocer si existen cachés, proxys, cortafuegos, túneles o cualquier otra cosa entre ella y el servidor que guarda la información. La aplicación, sin embargo, debe comprender el formato de la información devuelta (la *representación*), que es por lo general un documento HTML o XML, aunque también puede ser una imagen o cualquier otro contenido.

REST frente a RPC

Una aplicación web REST requiere un enfoque de diseño diferente a una aplicación basada en RPC (llamada de procedimiento remoto). En RPC, se pone el énfasis en la diversidad de operaciones del protocolo, o *verbos*; por ejemplo una aplicación RPC podría definir operaciones como:

- getUser()
- addUser()
- removeUser()
- updateUser()
- getLocation()
- addLocation()
- removeLocation()
- updateLocation()
- listUsers()
- listLocations()
- findLocation()
- findUser()

En REST, al contrario, el énfasis se pone en los recursos, o *sustantivos*; especialmente en los nombres que se le asigna a cada tipo de recurso. Por ejemplo, una aplicación REST podría definir algunos tipos de recursos asignándoles estos nombres:

- Usuario {}
- Localización {}

Cada recurso tendría su propio identificador, como `http://www.example.org/locations/us/ny/new_york_city`. Los clientes trabajarían con estos recursos a través de las operaciones estándar de HTTP, como GET para descargar una copia del recurso. Obsérvese cómo cada objeto tiene su propia URL y puede ser fácilmente cacheado, copiado y guardado como marcador. POST se utiliza por lo general para acciones con efectos laterales, como enviar una orden de compra o añadir ciertos datos a una colección.

Por ejemplo, el registro para un usuario podría tener el siguiente aspecto:

```
<usuario>
 <nombre>Maximiliano Alejandro</nombre>
 <sexo>hombre</sexo>
```

```
  <localización href="http://www.example.org/locations/us/ny/new_york_city">Nueva York, NY, US</localización>
</usuario>
```

Para actualizar la localización del usuario, un cliente REST podría primero descargar el registro XML anterior usando GET. El cliente después modificaría el fichero para cambiar la localización y lo subiría al servidor utilizando HTTP PUT.

Nótese, sin embargo, que los verbos HTTP (POST, GET, PUT, DELETE) no proporcionan ningún mecanismo estándar para descubrir recursos -- no hay ninguna operación LIST o FIND en HTTP, que se corresponderían con las operaciones `list*()` y `find*()` en el ejemplo RPC. En su lugar, las aplicaciones basadas en datos REST resuelven el problema tratando una colección de resultados de búsqueda como otro tipo de *recurso*, lo que requiere que los diseñadores de la aplicación conozcan URLs adicionales para mostrar o buscar cada tipo de recurso.

Por ejemplo, una petición GET HTTP sobre la URL `http://www.example.org/locations/us/ny/` podría devolver un enlace a una lista de ficheros en XML con todas las localizaciones posibles en Nueva York, mientras que una petición GET a la URL `http://www.example.org/users?surname=Michaels` podría devolver una lista de enlaces a todos los usuarios con el apellido "Michaels".

REST proporciona algunas indicaciones sobre cómo realizar este tipo de acciones como parte de su restricción "hipermedia como el medio de estado de la aplicación", lo que sugiere el uso de un lenguaje de formularios (tales como un formulario HTML) para especificar consultas parametrizadas.

La iniciativa OpenSearch de A9.com intenta estandarizar las búsquedas usando REST estableciendo especificaciones para descubrir recursos y un formato genérico para utilizar con sistemas basados en REST, incluyendo el RDF, XTM, Atom, RSS (en sus varias formas) y XML con XLink para gestionar los enlaces.

Principios

En su disertación original[1] del año 2000, Roy T. Fielding define REST como un *"...estilo arquitectónico para sistemas distribuidos de hipermedia, describiendo los principios de ingeniería de software que guían REST y las constantes de interacción elegidas para retener esos principios..."*. Estos principios son:

1. **Arquitectura cliente-servidor.** Hace hincapié en la separación de responsabilidades y la portabilidad. Cuanto menos conoce el servidor sobre el cliente más desacoplada está su interacción y más fácil resulta el cambio de componentes. Por definición, REST es una arquitectura diseñada para funcionar con sistemas distribuidos centralizados, en oposición a los que no usan un servidor como nodo principal –por ejemplo, mediante una arquitectura entre iguales o P2P–.
2. **Ausencia de estado**. El estado se guarda y mantiene en el cliente y no en el servidor. Es decir, las solicitudes deben proveer toda la información necesaria para poder realizarse en un servidor que no mantiene estados, por lo que no guarda contexto entre llamadas para un mismo cliente. Esto no significa que el servidor no pueda cambiar de contexto y facilitar esa transición al cliente. Para conseguir esto normalmente se usan redirecciones de peticiones; de forma que el cliente solo realice una petición y el servidor la procese en un endpoint o recurso, y la redirija a otro para continuar su procesamiento allí. Esto ocurre de forma transparente para el cliente, y es habitual por ejemplo en casos de uso relacionados con el registro y posterior inicio de sesión automático: El servidor crea la cuenta, genera un token de sesión y, en vez de devolverlo al cliente para que este llame a la URI de login –lo que aumentaría los tiempos, pues cada petición es costosa para los clientes–, lo reenvía a dicho

endpoint de inicio de sesión él mismo, devolviendo al cliente la respuesta final de la última petición realizada.

3. **Habilitación y uso de la caché.** Todas las solicitudes deben declarar si son o no cacheables, una forma estándar de hacerlo es con los encabezados *cache-control* (https://developer.mozilla.org/es/docs/Web/HTTP/Headers/Cache-Control) de HTTP. De esta forma, se pueden enviar respuestas cacheadas desde cualquier punto de la red sin necesidad de que la petición llegue al servidor. Pese a que en el caso de una API dinámica esto puede parecer no tener sentido, puede ahorrar costes en casos de datos inmutables, como definiciones.
4. **Sistema por capas.** Tiene relación con la separación de responsabilidades anteriormente mencionada, y establece que un cliente debe conocer únicamente la capa a la que le está hablando. Es decir, no debe tener en cuenta aspectos concretos, como particularidades de la base de datos usada; o abstracciones como cachés, proxies o balanceadores de carga implicados. Si se necesita seguridad, esta se debe añadir encima de los servicios Web, permitiendo que la lógica y seguridad permanezcan separadas.
5. **Interfaz uniforme.**
    1. **Identificación de recursos en las peticiones.** Como se señala en la **descripción** de esta página, las solicitudes identifican a recursos individuales. Sin embargo, REST recalca que los recursos están conceptualmente separados de las representaciones que son devueltas por el servidor (HTML, XML, JSON...). El tipo de formato se puede especificar en las cabeceras HTTP y, mediante negociación de contenido, servidor y cliente pueden ponerse de acuerdo en la respuesta que mandará el primero y que espera el segundo.
    2. **Manipulación de recursos a través de representaciones.** La especificación de REST intenta economizar peticiones en todo lo que sea posible. Por ello, cuando un cliente posee la representación de un recurso, incluyendo cualquier metadato adjunto, tiene suficiente información para modificar o eliminar el estado del recurso. Es decir, mediante las herramientas que REST promueve (uso de API auto-documentada, descriptiva, verbos HTTP...), el cliente puede saber predecir el resultado esperado de hacer cualquier operación sobre un recurso recibido primeramente con un GET.
    3. **Mensajes auto-descriptivos.** La idea de un mensaje auto-descriptivo es contener toda la información que el cliente necesita para entenderlo. Por lo tanto, no debería existir información adicional en una documentación separada o en otro mensaje.
    4. **Hipermedia como motor del estado de la aplicación** (*HATEOAS* por sus siglas en inglés, *Hypermedia As The Engine of Application State*). Una vez se ha accedido a la URI inicial de la aplicación, un cliente REST debería ser capaz de usar los enlaces proveídos dinámicamente por parte del servidor para descubrir todos los recursos disponibles que necesita. Según continúa el proceso, el servidor responde con texto que incluye enlaces a otros recursos que están actualmente disponibles. Esto elimina la necesidad de que el cliente tenga información escrita en el código (*hard-codeada)* con respecto a la estructura o referencias dinámicas a la aplicación.

Por la extensión de estas directrices y las particularidades de cada proyecto, dominio de negocio y API, puede resultar complejo aunar en una misma especificación todos los principios que REST describe. Por ello, el modelo de madurez de Richardson describe los niveles por los que pasa una especificación de API REST desde que es creada hasta que se perfecciona adquiriendo controles hipermedia. Estos niveles son:

1. **Nivel 0.** Los servicios cuentan con una sola URI que acepta todo el rango de operaciones admitidas por el servicio, con unos recursos poco definidos. No se considera una API RESTful.
2. **Nivel 1**. Introduce recursos y permite hacer peticiones a URIs individuales para acciones separadas en lugar de exponer un punto de acceso universal. Los recursos siguen estando

generalizados pero es posible identificar un ámbito algo más concreto. El primer nivel sigue sin ser RESTful, pero está más orientado a adquirir la capacidad de serlo.

3. **Nivel 2.** El sistema empieza a hacer uso de los verbos HTTP. Esto permite mayor especialización y generalmente conlleva la división de los recursos en dos: Uno para obtener únicamente datos (GET) y otro para modificarlos (POST), aunque un grado mayor de granularidad también es posible. Una de las desventajas de proveer un sistema distribuido con más de una petición GET y POST por recurso puede ser el aumento de complejidad del sistema, a pesar de que el consumo de datos por clientes de la API se simplifica en gran medida.
4. **Nivel 3.** El último nivel introduce la representación hipermedia, también llamada *HATEOAS*. Esta representación se realiza mediante elementos incrustados en los mensajes de respuesta de los recursos, que permiten al cliente que envía la petición establecer una relación entre entidades de datos individuales. Por ejemplo, una petición GET a un sistema de reservas de un hotel, podría devolver el número de habitaciones disponibles junto con los enlaces que permiten reservar habitaciones específicas.

Diferencias con SOAP

Mientras que una arquitectura REST está fundamentalmente centrada en **datos** (recursos), una arquitectura que use el protocolo SOAP se orienta más hacia los **servicios** que permiten operar con dichos datos. Con frecuencia se puede caer en el error de pensar que SOAP es también otro tipo de arquitectura; sin embargo, como su nombre indica es un **protocolo**, por lo que restringe más su posible implementación y a cambio está más **estandarizado**. Esto tiene especial relevancia en el tipo de mensajes que los servidores envían o reciben: Los clientes REST envían información generalmente en HTML o XML, mientras que una implementación con **SOAP** se decanta normalmente por el último; teniendo un **fuerte tipado**, con un lenguaje más **verboso** que JSON –muy extendido en servicios que implementan arquitecturas REST– y demandando un **mayor ancho de banda** por transmisión.

Por otra parte, los servicios REST deben poder ser probados mediante un **navegador** para poder ser RESTful, por lo que el propio servicio debe controlar si devuelve un contenido solo amigable para las máquinas o una presentación también legible para humanos en función del **contexto**. Para probar una implementación de SOAP, se hace necesario el uso de herramientas *ad-hoc* como la conocida **SoapUI**. Además, el uso de **bibliotecas** está más extendido en esta última opción, quedando REST reservado únicamente para las URIs que representan los recursos como se ha descrito anteriormente.

REST se basa en el principio de **lectura** como operación más frecuente, por lo que la mayor parte de las peticiones serán de tipo GET; mientras que SOAP está más construido con peticiones POST. Debido a esto último y a unos recursos inespecíficos, SOAP puede conllevar el desarrollo de clientes más **complejos** que los que se podrían construir con servicios REST.

Ventajas e inconvenientes

Como toda propuesta arquitectónica, su adopción se debe evaluar de forma holística con respecto al balance de beneficios-costes cuya implementación puede conllevar. Si bien es cierto que a día de hoy gran parte de APIs –en especial las públicas o abiertas a su consumo por terceras partes– se diseñan para ser REST o RESTful y, por lo tanto, hay cierto **consenso** en los programadores sobre qué esperar, la **estandarización** del protocolo SOAP a nivel de implementación hace fácil desentenderse de muchas decisiones que en REST quedan abiertas. Por otro lado, la visibilidad a la que están expuestas las URIs públicas puede suponer un problema de **seguridad**, además de las limitaciones técnicas por la longitud máxima de los parámetros. Esto último se puede solucionar con solicitudes POST, que SOAP recomienda de forma habitual, y es especialmente útil en el caso de enviar gran cantidad de información o datos

binarios. Por lo general, REST puede ser una solución algo más sencilla de implementar en cliente y servidor, con muchos *frameworks* ofreciéndolo por defecto *out-of-the-box*, como el caso de Django. Sin embargo, SOAP puede ahorrar decisiones sobre detalles de implementación y ofrecer operaciones de forma más transparente al cliente, publicando los servicios concretos disponibles en un endpoint dado, en lugar de las abstracciones de los datos.

Implementaciones públicas

Dado que la definición de REST es muy amplia, es posible afirmar que existe un enorme número de aplicaciones REST en la red (prácticamente cualquier cosa accesible mediante una petición HTTP GET). De forma más restrictiva, en contraposición a los servicios web y el RPC, REST se puede encontrar en diferentes áreas de la web:

- La blogosfera -el universo de los blogs- está, en su mayor parte, basado en REST, dado que implica descargar ficheros XML (en formato RSS o Atom) que contienen listas de enlaces a otros recursos.
- Amazon.com ofrece su interfaz para desarrolladores (https://www.amazon.com/gp/aws/landing.html) tanto en formato REST como en formato SOAP (siendo la versión REST la que recibe mayor tráfico).
- eBay ofreció hasta 2008 una interfaz REST para desarrolladores (http://developer.ebay.com/rest/).
- El Proyecto "Seniors Canada On-line" (https://web.archive.org/web/20060428061214/http://www.seniors.gc.ca/index.jsp) del Gobierno de Canadá ofrece una interfaz REST descrito aquí (http://www.megginson.com/blogs/quoderat/archives/2005/03/09/public-rest-application-seniors-canada-online/).
- Bloglines ofrece una API basada en REST para desarrolladores (https://web.archive.org/web/20051228212538/http://www.bloglines.com/services/).
- Yahoo! ofrece una API en REST para desarrolladores (http://developer.yahoo.com).
- El mecanismo de enrutamiento de Ruby on Rails soporta aplicaciones REST utilizando el patrón de diseño MVC.
- Microsoft tiene su implementación en ADO.NET Data Services Framework (anteriormente conocido como “Astoria”) [1] (http://msdn.microsoft.com/en-us/library/cc668792.aspx).
- El mismo mecanismo en Catalyst (http://www.catalystframework.org) también soporta aplicaciones REST mediante MVC.
- El publicador de objetos de Zope.
- Implementación REST para Java: RestLet (https://web.archive.org/web/20110515125242/http://www.restlet.org/).
- Facebook ofrece una API basada en REST.
- Twitter ofrece una API basada en REST.
- MEGA ofrece una API basada en REST.
- MercadoLibre ofrece una API basada en REST para desarrolladores (http://developers.mercadolibre.com/).
- OpenNebula por medio de formato JSON y REST, permite crear, controlar y monitorizar servicios entre máquinas virtuales interconectadas.[2]

Probablemente existan muchas otras implementaciones similares.

En cualquier caso debe tenerse en cuenta que muchas de las implementaciones descritas arriba, no son totalmente RESTful, esto es, no respetan todas las restricciones que impone la arquitectura REST. Sin embargo todas están inspiradas en REST y respetan los aspectos más significativos y restrictivos de su

arquitectura, en particular la restricción de "interfaz uniforme". Estos servicios han sido denominados "Accidentalmente RESTful" (http://www.markbaker.ca/2002/09/Blog/2005/04/14#2005-04-amazon-next).

Referencias

1. Fielding, Roy Thomas (2000). *Architectural styles and the design of network-based software architectures* (https://dl.acm.org/doi/book/10.5555/932295). University of California, Irvine. doi:10.5555/932295 (https://dx.doi.org/10.5555%2F932295). Consultado el 16 de abril de 2021.
2. OneFlow Specification, versión 5.12 (http://docs.opennebula.io/5.12/integration/system_interfaces/appflow_api.html)

- Rodrigo Santamaría. Apuntes Sistemas Distribuidos Universidad de Salamanca. | Tema 3 - Middleware (http://vis.usal.es/rodrigo/documentos/sisdis/teoria/3-middleware.pdf)

Véase también

- XML
- Servicio web
