REST:Representational State Transfer

Escuela Técnica Superior de Ingeniería de Telecomunicación
Universidad Rey Juan Carlos

gsyc-profes (arroba) gsyc.urjc.es

Abril de 2022

REST

REST *Representational State Transfer* es la tecnología más empleada actualmente para intercambiar información *de gestión* entre máquina y máquina

- En rigor, REST es una propuesta académica, no una arquitectura concreta.
- Define una serie de características que debería cumplir una arquitectura.
- Los protocolos que cumplen las restrucciones REST, se les denomina RESTful
- Desarrollado por Roy Fielding a finales de los años 90. Publica REST en su tesis doctoral, año 2000
  - Fielding es uno de los autores de HTTP 1.1
- En la práctica, cuando una aplicación dice ser REST, suele ser ROA (una arquitectura REST concreta desarrollada L.Richardson y S.Ruby)

Características de una arquitectura REST

Una arquitectura es REST si cumple estas 6 caracterísitcas

1. Cliente-servidor
2. Sin estado (*stateless*)
3. Admite el uso de caches (*cacheable*)
4. Sistema basado en capas (*layered system*)
5. Generación (opcional) de código bajo demanda
6. Uso de interfaces uniformes
   - Identificación de recursos
   - Manipulación de recursos mendiante representaciones
   - Mensajes autodescriptivos
   - Hipermedia como motor del estado de la aplicación (*HATEOAS, Hypermedia as the engine of application state*)

1. Cliente-servidor

- Clara separación cliente-sevidor en el API, interfaz robusto que permita desarrollo independiente de cada parte
  - P.e. El cliente no almacena datos
- El servidor ignora al usuario: tanto el interfaz de usuario como el estado de usuario
- Cliente y servidor se pueden desarrollar por separado

2. Sin estado

- En el servidor no se almacena contexto de las peticiones del cliente
- Cada petición del cliente contienen toda la información necesaria para que el servidor responda
- La sesión y el estado del cliente se guardan en el cliente
- Todo esto simplifica la lógica del servidor
  - Garantiza que no llegará a estados inconsistentes (no hay estado)
  - Permite replicar los servidores
  - Facilita el escalado

Ejemplo de protocolo sin estado:

- HTTP

Ejemplo de protocolo que sí tiene estado:

- FTP
  - Hay una sesión, con un usuario autenticado, con directorio de trabajo, con modo de transferencia (texto o binario) preestablecido

3. Admite cachés

- Cada respuesta indica si es susceptible de ser almacenada en un cache o no
  - Si no lo es, los cachés se deshabilitarán, lo que garantiza que no se usará información obsoleta o inadecuada
- Usar caches mejora escalabilidad y el rendimiento

4. Sistema basado en capas

Entre cliente y servidor puede haber proxys, caches o gateways, sin que el cliente lo perciba y sin que produzcan resultados incorrectos

- Un proxy es un intermediario. Recibe la petición y la vuelve a enviar.
  - Por motivos de seguridad, para poner un filtro para niños, para conseguir anonimato, para modificar los documentos sobre la marcha (cambiar formato, idioma..) para forzar a que una red pase por una máquina y haga cache...
- Un gateway es como un proxy, pero cambiando de protocolo. P.e. gateway http-ftp

5. Puede generar código bajo demanda

- Opcionalmente, el servidor puede ampliar la funcionalidad del cliente enviando código, como applets java o JavaScript

6. Interfaz uniforme

Es la característica principal de cualquier sistema REST. Permite sistemas débilmente acoplados

- 6.1 Identificación de recursos
- 6.2 Manipulación de recursos mendiante representaciones
- 6.3 Mensajes autodescriptivos
- 6.4 Hipermedia como motor del estado de la aplicación (*HATEOAS*)

6.1 Identificación de recursos

- Los recursos individuales se identifican en las peticiones, típicamente mediante URIs en sistemas web.
- El recurso es una cosa y la representación es otra
  - Por ejemplo un recurso (un fichero) se puede servir representado en HTML, en XML o JSON. Pero ninguno de estos formatos tiene por qué ser el formato en que el recurso es almacenado en la base de datos

6.2 Manipulación de recursos mediante representaciones

En necesario desacoplar el recurso de la representación del recurso.

- El recurso tiene una única URI para todos los formatos
- En las cabeceras, cliente y servidor negocian qué formatos soportan, prefieren o solicitan de cada recurso.
- No hay una URI diferente para cada formato, cada formato no se entiende como una entidad separada

Recordatorio:

- URI: Uniform Resource Identifier
  Dos tipos
  - URL: Uniform Resource Locator
    Incluye protocolo de acceso
  - URN: Uniform Resource Name
    No indica cómo acceder al recurso

6.3 Mensajes autodescriptivos

Cada mensaje contiene toda la información necesaria para ser procesado.

- Ejemplo: *Internet Media Type* (antiguamente llamado tipo MIME)
- Contraejemplo: Fichero de texto *code page*

6.4 Hipermedia como motor del estado de la aplicación

HATEOAS, Hypermedia as the engine of application state

- Las transiciones entre estados se especifican en un documento HTML, que devuelve el servidor
- El cliente puede descubrir por si mismo los estados a los que puede pasar, mediante información que le pasa en el servidor, en formato hipermedia (enlaces html), sin necesidad de instrucciones adicionales *fuera de banda*

ROA: Resource-Oriented Architecture

L. Richardson y S.Ruby en su libro de 2007 *RESTful Web Services* proponen una arquitectura concreta, RESTful, a la que denominan *ROA: Resource-Oriented Architecture*

- La mayoría de los servicios RESTful actuales siguen esta arquitectura ROA
  - Aunque sus desarrolladores pueden no reconocer este nombre
- Hay arquitecturas que se autodenominan RESTful, pero que no siguen estas pautas, resulta discutible que sean verdaderamente RESTful

Las pautas de Richardson y Ruby para hacer servicios RESTful son muy similares a las características definidas por Fielding, pero un poco más concretas

1. La arquitectura está basada en *recursos* (*resources*)
2. Todo recurso tiene que tener una URI
3. Las aplicaciones son direccionables (*Addressability*)
4. El servidor no tiene estado (es *stateless*)
5. Las aplicaciones usan un interfaz uniforme: GET, PUT, DELETE

1. La arquitectura está basada en recursos

Un recurso *resource* es cualquier cosa con suficiente entidad como para merecer ser nombrado e indentificado por sí mismo:

- La versión 1.0.3 de cierto software
- Un vídeo
- Una entrada en un blog
- Un mapa de Fuenlabrada
- El precio actual del bitcoin en euros en Kraken
- ...

2. Todo recurso tiene que tener una URI

Si no tiene URI, no es un recurso

Y la estructura tiene que ser homogénea y predecible

- http://www.ejemplo.com/software/releases/1.0.3.tgz
- http://www.ejemplo.com/videos/helloworld
- http://www.example.com/blog/2016/04/20
- http://www.example.com/mapas/ES/MAD/fuenlabrada
- http://www.example.com/bitcoin/kraken/BTCEUR

3. Las aplicaciones son direccionables

Addressability
Una aplicación es direccionable si expone todos sus datos relevantes como recursos
Ejemplo:

- http://www.google.com/search?q=fuenlabrada

Esto es muy importante, cuando no existía este concepto, por ejemplo con el protocolo FTP, había que dar indicaciones como *abra una sesión FTP con el usuario anonymous, cambie el directorio a pub/files/ y descargue el fichero file.txt*

Por ser las aplicaciones direccionables, se puede:

- Crear un marcador
- Crear una URI, ponerla por escrito, en un email, en un QR
- Enlazar desde otra página web
- Usar una cache
  - De forma que la segunda vez que se pida el recurso, no se vuelva a consultar la fuente original sino que se sirve la versión en cache (tomando las precauciones necesarias para no servir información anticuada)

- La página que usa Iberia en 2022 para localizar un vuelo, no es RESTful, no es direccionable

  `http://www.iberia.com/es/arrivals-and-departures/`

  No es posible enviar por correo un enlace a tu vuelo
- Para ser REST, debería ser algo así

  `http://www.iberia.com/flights/ib3214/20220329`

No solo las direcciones web son direccionables, por ejemplo también es direccionable

- Un sistema de ficheros

  `/home/jperez/st/ejemplo.txt`

- Las celdas de una hoja de cálculo

Algo tan sencillo como ponerle una URI a un folleto para que la gente lo pueda ver en su navegador no sería posible sin el direccionamiento

- Si un servicio web es direccionable, los clientes pueden usarlos de formas nuevas, no previstas en el diseño original
- Facilita el ser usado desde distintos dispositivos: aplicaciones de escritorio, clientes web, clientes nativos para smartphone (iOS, android, BlueBerry...)

4. El servidor no tienen estado

- Cada petición es independiente de las anteriores
  - Ejemplo: una búsqueda en google devuelve 10 entradas, cada una de ellas contiene toda la información necesaria para hacer la búsqueda
  - Contraejemplo: Cualquier sistema con *directorio actual*
- Cada uno de los posibles estados es un recurso, con su propia URI
- El estado se tienen que guardar en el cliente, nunca en el servidor

El diseño original de HTTP era *stateless*, pero al añadir cookies, este principio suele romperse

- Las cookies que escribe el servidor rompen este principio, no son el estado, son una clave para identificar un estado guardado en el servidor
- Las cookies puden ser RESTful si las genera el cliente y continen toda la información para hacer una petición (posible, pero no habitual)

5. Interfaz uniforme: GET, PUT, POST, DELETE

REST dice que el interfaz debe ser uniforme, pero no especifica ninguno en particular
ROA, sí: Lás únicas acciones posibles son métodos HTTP: GET, PUT, POST, DELETE

- Atención con el término *método*, es el que emplea HTTP pero no tiene nada que ver con los métodos de la programación orientada a objetos

- GET obtiene un recurso
- PUT crea un recurso, creando una nueva URI especificada por el cliente
- PUT modifica un recurso, si la URI ya existía
- POST añade datos a una URI preexistente
  Este añadido puede significar:
  - Modificar un recurso ya existente
  - Crear un nuevo recurso, cuya URI crea el servidor, no el cliente. El cliente solo indica la dirección del recurso *padre* (ejemplo típico: añadir entrada a un blog)
- DELETE borra un recurso

En muchas ocasiones, POST se emplea de forma contraria a REST

- Es el comportamiento de SOAP/WSDL y XML-RPC, es el POST al estilo RPC
  Webber lo llama POST *sobrecargado*
  - El cliente hace todos los POST a la misma URI
  - El servidor analiza el contenido y realiza una acción u otra

Otros métodos de HTTP también pueden resultar útiles, no se usan mucho pero pueden usarse sin romper REST/ROA

- HEAD
  Obtiene los metadatos de un recurso, no los datos
- OPTIONS
  Averigua qué métodos soporta un recurso

Seguridad e idempotencia

- Método seguro (*safety*)
  No modifica nada en el servidor
- Método idempotente
  Aplicarlo una vez es equivalente a aplicarlo varias veces
  - Ejemplo: A=10
  - Contraejemplo: A=A+1

Para que una aplicación sea ROA, GET y HEAD tienen que ser seguros

- No pueden provocar ninguna acción excepto tal vez algún efecto lateral como p.e. actualizar un log
- Por ser seguros, también son idempotentes

Muchas aplicaciones web no cumplen este requisito
Eso provocó el fracaso de la primer versión, de 2005, del *Web accelerator* de Google

- PUT y DELETE tienen que ser idempotentes. Esto no siempre se cumple, entonces la aplicación ya no es REST, o al menos, no es ROA
- A veces los programadores incluso diseñan su sistema de forma que GET provoca cambios en los recursos. Esto va contra todos los principios no solo de REST, sino también de HTTP