REST:Representational State Transfer

Escuela Técnica Superior de Ingeniería de Telecomunicación
Universidad Rey Juan Carlos

gsyc-profes (arroba) gsyc.urjc.es

Abril de 2016

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
- URL: Uniform Resource Locator
- URN: Uniform Resource Name

Un identificador (URI) puede ser o bien una dirección (URL) o bien un nombre (URN) o bien ambas cosas

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 usuarios 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/sowtware/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 2016 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/ib3214/20160329`

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: Las 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

Ajax

Es un conjunto de técnicas para enviar información asíncrona entre servidor web y cliente

- Aunque hay antecedentes desde 1996, el Ajax actual lo crea google en 2004, para gmail y google maps
- Evita que el usuario necesite pulsar *enviar*. La aplicación web recibe información continuamente, resultando una experiencia de usuario similar a la de una aplicación de escritorio
- Originalmente significaba Asynchronous JavaScript and XML, pero cuando empieza a usarse sin JavaScript y sin XML, el acrónimo AJAX se abandona para usar la palabra Ajax

Funcionamiento de ajax

1. El usuario solicita una URL desde su navegador
2. El servidor web devuelve la página, que contiene un script
3. El navegador presenta la página y ejecuta el script
4. El script hace peticiones HTTP asíncronas a una URI del servidor, sin que el usuario intervenga. El servidor suele ser RESTful
5. El script interpreta las respuestas HTTP y modifica el HTML, sin que el usuario intervenga

¿Es Ajax RESTful?

- Ajax es una arquitectura muy similar a la de una aplicación de escritorio
- En una aplicación web ordinaria, el httl solo cambia (la vista solo se refresca) cuando el usuario envía una actualización (pulsa un botón como aceptar, enviar o similar)
- El inconveniente es que toda la aplicación tiene la misma URI
  No es direccionable ni stateless
  El usuario no puede guardar en un bookmark un estado, ni el botón *back* funciona correctamente

- El servicio tal vez sí es REST, pero solo el script javascript lo aprovecha

  En ocasiones hay *atajos* para que el usuario también lo use, pero no está diseñado para usarse así
  Por ejemplo, buscar todos mis correos con el término *fuenlabrada*

  `https://mail.google.com/mail/?q=fuenlabrada&search=query&view=tl`

  A esta arquitectura a veces se denomina *REST híbrido*

Libería Requests

En python hay diversas librerías para manejar HTTP
Una de las más convenientes y populares es Requests

- `http://python-requests.org`

- Desarrollado por Kenneth Reitz con licencia libre Apache2
- Extremadamente fácil de usar
  - El método `get()` hace una petición a la url pasada como parámetro
  - El atributo `status_code` devuelve el status http
  - El métido `json()` decodifica la respuesta json y devuelve un valor python
- Instalación

```
pip install requests
```

Uso básico de Requests

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import requests

# Este servicio de prueba nos devuelve nuestra dirección ip
url="http://httpbin.org/ip"

r=requests.get(url)
if r.status_code==200:
    print r.json() # {u'origin': u'81.37.7.200'}
else:
    print "error "+str(r.status_code)
```

Paso de parámetros

- Para hacer peticiones con parámetros como p.e.

  `https://data.btcchina.com/data/ticker?market=all`

  basta pasar al método get el parámetros `params`, dándole como valor un diccionario
- Otro parámetro habitual es `timeout`, en segundos

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import requests

url="https://data.btcchina.com/data/ticker"
payload={}
payload["market"]="all"
my_timeout=5

r=requests.get(url,params=payload, timeout=my_timeout)
if r.status_code==200:
    print r.json()
else:
    print "error "+str(r.status_code)
```

Valor devuelto:


```
{u'ticker_ltcbtc': {u'sell': u'0.00780000', u'buy': u'0.00770000',
u'last': u'0.00780000', u'vol': u'16.95500000', u'prev_close':
u'0.0077', u'vwap': u'0.0076', u'high': u'0.00790000', u'low':
u'0.00760000', u'date': 1459699123, u'open': u'0.0077'}, u'ticker_ltccny':
{u'sell': u'21.20', u'buy': u'21.05', u'last': u'21.20', u'vol':
u'5976.87000000', u'prev_close': u'21.16', u'vwap': u'21.07',
u'high': u'21.21', u'low': u'21.00', u'date': 1459699123, u'open':
u'21.16'}, u'ticker_btccny': {u'sell': u'2726.98', u'buy': u'2726.65',
u'last': u'2726.79', u'vol': u'17131.95190000', u'prev_close':
u'2725.22', u'vwap': u'2721', u'high': u'2730.49', u'low': u'2712.49',
u'date': 1459699122, u'open': u'2725.02'}}
```

Otras respuestas

- Si la petición devuelve una imagen, un xml o cualquier otro tipo de recurso, basta leer el atributo *content* de la respuesta
  - Si el recurso está comprimido, la librería lo descomprimirá automáticamente
- Las cabeceras están dispoibles en el atributo *headers*
  - Para saber el tipo de contenido (según la norma MIME): `headers["content-type"]`

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import requests,sys
def main():
    url="http://placehold.it/150/15c072"

    r=requests.get(url)
    if r.status_code!=200:
        sys.stderr.write("error"+str(r.status_code))
        raise SystemExit

    print r.headers["content-type"] #   image/png
    fichero=open('mi_imagen','wb')
    fichero.write(r.content)
    fichero.close()
if __name__ == "__main__":
    main()
```

Flask

Flask es un micro *framework* para hacer aplicaciones web en python

- Desarrollado por Armin Ronacher en 2010, ha ido ganando mucha popularidad
- Especialmente útil para servicios REST
- Alternativa a herramientas más completas y complejas como Django
- Ofrece un uso muy sencillo, con la funcionalidad mínima para aceptar peticiones HTTP y ofrecer respuestas, sin integrar gestión de formularios, bases de datos, autenticación de usuarios, etc
  - Aunque todo esto se puede añadir mediante extensiones
- Instalación de flask: `pip install flask`

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import flask
app=flask.Flask(__name__)

@app.route('/')
def index():
    return "hola mundo"

if __name__ == '__main__':
    app.run(debug=True)
```

Para depurar el código, flask incorpora un servidor web, que por omisión atiende peticiones en

`http://127.0.0.1:5000`

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
from flask import Flask
app=Flask(__name__)
@app.route('/')
def index():
    return "hola mundo"

@app.route('/user/<name>')
def user(name):
    return "hola, "+name

if __name__ == '__main__':
    app.run(debug=True)
```

El ejemplo anterior lo invocamos con

```
http://127.0.0.1:5000/user/pedro
```

Obtendremos en el navegador

```
hola, pedro
```

Y en la shell donde lanzamos el script python

```
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 128-685-262
127.0.0.1 - - [13/Apr/2016 20:20:44] "GET /user/pedro HTTP/1.1" 200 -
```

Respuesta json

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
from flask import Flask
import json

app=Flask(__name__)
@app.route('/')
def index():
    return "hola mundo"

@app.route('/user/<name>')
def user(name):
    respuesta=["hola","mundo"]
    return json.dumps(respuesta)

if __name__ == '__main__':
    app.run(debug=True)
```

Query string

- En http, cuando el cliente desea pasar parámetros no jerárquicos, usa la *query string*
- Ejemplo:

```
http://127.0.0.1:5000/user/juanpedro?city=toledo&profile=basic
```

- Flask nos permite acceder a los argumentos de la query string, a través del diccionario `args` del objeto global `request`

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import flask
app=flask.Flask(__name__)

@app.route('/user/<name>')
def user(name):
    query_string= flask.request.args
    respuesta="Solicitas "+name
    respuesta=respuesta+" de la ciudad "+query_string["city"]
    respuesta=respuesta+" con perfil "+query_string["profile"]
    return json.dumps(respuesta)

if __name__ == '__main__':
    app.run(debug=True)
```

Resultado:

```
Solicitas juanpedro de la ciudad toledo con perfil basic
```

El servidor flash puede devolver una tupla, cuyo segundo parámetro es el status http

```
#!/usr/bin/python -tt
# -*- coding: utf-8 -*-
import flask

app=flask.Flask(__name__)

@app.route('/user/<name>')
def user(name):
    if name=="juan" or name =="maria":
        respuesta="Datos de "+name+": ..."
    else:
        respuesta="Usuario desconocido",404
    return respuesta

if __name__ == '__main__':
    app.run(debug=True)
```