REpresentational State Transfer (REST)

PID_00275869

Silvia Llorente Viejo

Tiempo mínimo de dedicación recomendado: 4 horas

Universitat Oberta de Catalunya

Silvia Llorente Viejo

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Joan Manel Marquès Puig

Primera edición: septiembre 2020

Av. Tibidabo, 39-43, 08035 Barcelona
Autoría: Silvia Llorente Viejo
Producción: FUOC

Índice

Introducción

La comunicación entre sistemas ha ido evolucionando muy rápidamente en los últimos años. Hemos pasado de hacer comunicaciones «a medida» entre sistemas a definir mecanismos abiertos basados en estándares y directrices.

El objetivo de este módulo es describir los servicios web basados en REpresentational State Transfer (REST). REST define una arquitectura de cómo implementar servicios web basándose en los métodos del Hypertext Transfer Protocol (HTTP), el protocolo web por excelencia.

Así, pues, se describe el funcionamiento general de REST, su relación con HTTP y cómo se pueden hacer las peticiones y recibir las respuestas. También se comparan los servicios web REST con los servicios web basados en Simple Object Access Protocolo (SOAP), uno de los mecanismos más utilizados para comunicar sistemas heterogéneos antes de la aparición de REST.

Además, se dan algunos detalles de cómo se tendría que implementar un servicio web basado en REST dando algunos ejemplos concretos en diferentes lenguajes de programación como son Java, Node.js y Python.

Objetivos

A través del estudio de este módulo lograréis los siguientes objetivos:

1. Conocer qué es REpresentational State Transfer (REST) y su arquitectura básica.
2. Conocer la relación entre REST y los diferentes métodos del protocolo HTTP.
3. Conocer las bases para definir servicios REST.
4. Conocer otros tipos de servicios web y su relación con REST.
5. Aprender a implementar servicios REST basados en Java y cómo conectarse a ellos.
6. Aprender las bases para implementar clientes y servicios REST basados en otros lenguajes de programación.

1. ¿Qué es REpresentational State Transfer (REST)?

REpresentational State Transfer (REST) define una arquitectura de cómo implementar servicios web distribuidos de forma más simple que otras arquitecturas existentes como pueden ser *Simple Object Access Protocol* (SOAP) o *Remote Method Invocation* (RMI).

Sobre REST, SOAP y RMI

REST: *REpresentational State Transfer* es un estilo de servicios web basados en HTTP.

SOAP: *Simple Object Access Protocolo* es un estilo de servicios web que hace un uso extensivo del lenguaje XML.

RMI: *Remote Method Invoation* es un mecanismo para invocar objetos remotos en lenguaje Java.

REST usa los métodos de *HyperText Transfer Protocol* (HTTP)[1] para invocar las operaciones *Create, Read, Update* y *Delete* CRUD (en español, Crear, Leer, Actualizar y Borrar) en vez de definir operaciones específicas como se hace en otros mecanismos de invocación remota de operaciones (por ejemplo, SOAP o RMI). Esto quiere decir que, cuando en un servicio basado en REST se quiere definir una operación de consulta de un recurso, en vez de definir una operación con el nombre `consultaRecurso` donde se pasa el código del recurso como parámetro, se define una *Uniform Resource Locator* de acceso al recurso con el método HTTP GET, como por ejemplo `www.example.com/recurso/codigo`[2]. En este caso concreto, el parámetro se pasa en la URL, pero se pueden usar diferentes mecanismos y/o formatos para el paso de parámetros. El resultado de esta operación se enviará dentro de un mensaje de respuesta HTTP y puede tener diferentes formatos, como veremos más adelante en este módulo.

(1)**Internet Engineering Task Force (IETF)** (2014) Hypertext Transfer Protocol (HTTP/1.1), RFC 7230 a 7235, <https://tools.ietf.org/html/rfc7230> <https://tools.ietf.org/html/rfc7235>

(2)Nota general: ninguno de los enlaces incluidos en el texto lleva guiones de separación

Finalmente, hay que destacar que REST no es un estándar, a pesar de que está basado en estándares como por ejemplo HTTP, *eXtensible Markup Language* (XML)[3] o *Javascript Object Notation* (JSON)[4]. A lo largo de este módulo veremos cómo usar estos estándares para ofrecer servicios web basados en REST.

(3)**World Wide Web Consortium (W3C)** (2006). Extensible Markup Language (XML) 1.1 (Second Edition), <https://www.w3.org/tr/xml11>

(4)**European Computer Manufacturers Association (ECMA)** (2017). The JSON (Javascript Object Notation) Data Interchange Syntax ECMA – 404 Standard, <http://www.ecma-international.org/publications/files/ecma-st/ecma-404.pdf>

1.1. Visión general

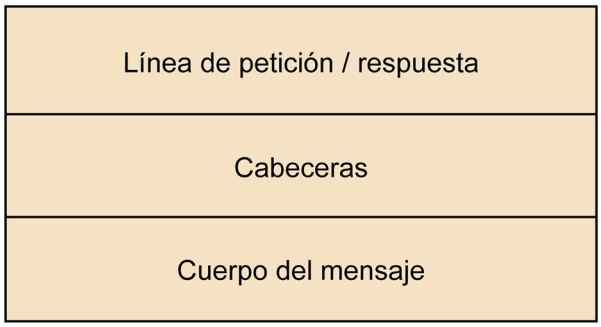
Como ya se ha dicho, REST no es un estándar. Se trata más bien de un estilo arquitectural o patrón de diseño de una Interfaz de Programación de Aplicaciones (API, de sus siglas en inglés *Application Programming Interface*) que funciona sobre HTTP. Lo definió inicialmente Roy Fielding en su tesis doctoral en 2000.

Referencia bibliográfica

Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. [en línia]: <https://www.ics.uci.edu/~fielding/pubs/ dissertation/fielding_dissertation.pdf>.

Para entender mejor el funcionamiento de REST, explicaremos brevemente el funcionamiento del protocolo HTTP. HTTP es un protocolo de tipo Cliente – Servidor donde cliente y servidor se comunican mediante mensajes de petición (del cliente hacia el servidor) y de respuesta (del servidor hacia el cliente) con un formato predeterminado, tal y como se muestra en la figura 1.

Figura 1. Estructura general de los mensajes intercambiados mediante HTTP



La primera línea del mensaje cambia según si el mensaje es de petición o de respuesta. En el caso del mensaje de petición, la línea de petición contiene el método HTTP, la URL del recurso (sin el protocolo ni el nombre del servidor) y la versión de protocolo HTTP que se está usando. A continuación se puede ver un ejemplo de línea de petición, donde se usa el método GET y se pide la página `index.html` con la versión 1.1 del protocolo HTTP.

```
GET index.html HTTP/1.1
```

En el caso del mensaje de respuesta, la línea de respuesta contiene la versión HTTP con la que responde el servidor, el código de estado de la petición y una descripción textual asociada a este código. A continuación se puede ver cómo sería una respuesta exitosa a la petición anterior, donde se indica que la versión del protocolo es 1.1, que el código de respuesta es 200, que corresponde a una respuesta correcta indicando que se ha encontrado la página `index.html` en el servidor y se envía su contenido en el cuerpo del mensaje de respuesta.

```
HTTP/1.1 200 OK
```

El apartado de cabeceras se puede encontrar en los dos tipos de mensajes, tanto petición como respuesta. Algunas cabeceras son comunes a los dos tipos de mensajes y otras son específicas de la petición o la respuesta. A continuación se puede encontrar un ejemplo de cabecera común a los dos tipos de mensajes, `Content-Length`, que nos indica la longitud en bytes del cuerpo del mensaje.

```
Content-Length: 2000
```

Finalmente, el cuerpo del mensaje contiene información referente a la petición, como pueden ser parámetros para poderla realizar, o el contenido de la respuesta. En el caso concreto del método GET, el cuerpo del mensaje de petición tiene que estar obligatoriamente vacío.

Además, HTTP es un protocolo sin estado, es decir, el resultado de las peticiones no depende de peticiones anteriores. En caso de necesitar almacenar el estado es necesario usar mecanismos externos como por ejemplo galletas (*cookies*, en inglés) o sesiones.

Así pues, REST usa los mecanismos proporcionados por HTTP para poder implementar servicios web. El estilo arquitectural de REST tiene algunas características específicas que detallamos a continuación:

- **No tiene estado**: cada petición del cliente hacia el servidor tiene que contener toda la información necesaria para entender la petición y no puede aprovechar el contexto almacenado en el servidor.

- **Memoria caché**: para mejorar la eficiencia de las respuestas, estas tienen que poder indicar si se pueden almacenar o no en las memorias caché de los sistemas intermedios, como por ejemplo representantes (*proxies*, en inglés) o incluso navegadores.

- **Interfaz uniforme**: todos los recursos están accesibles por medio de una interfaz genérica (por ejemplo, la que proporcionan los métodos HTTP, GET, POST, PUT, DELETE, etc.).

- **Recursos con nombre**: el sistema está compuesto por recursos a los que se identifica con una URL (por ejemplo, `www.example.com/recurso/codigo`, damos acceso a un recurso a partir de su código).

- **Representaciones de recursos interconectadas**: los recursos se interconectan por medio de URL y permiten al cliente pasar de un estado a otro «navegando» entre representaciones. Se sigue la misma lógica que en una aplicación web, donde las páginas web están interconectadas entre ellas y podemos navegar siguiendo los enlaces. Por ejemplo, la URL `www.example.com/recurso/lista` permite pedir el listado de todos los recursos. En el resultado se tendría que devolver una URL a cada recurso, para obtener sus datos básicos o incluso más detalles, con URL del tipo `www.example.com/recurso/código` o `www.example.com/recurso/código/detalles`.

En las secciones siguientes se explica con más detalle cómo REST usa HTTP para enviar y recibir datos y gestionar recursos usando las operaciones CRUD.

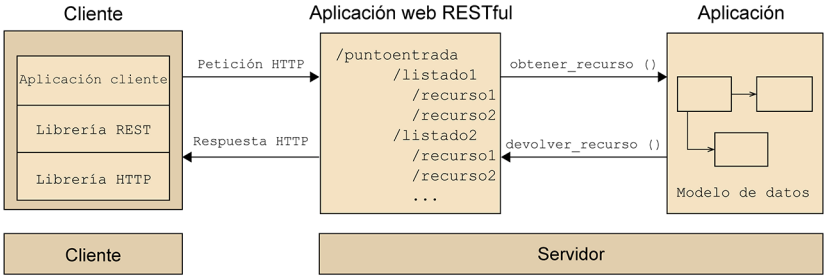
1.2. Relación entre REST y HTTP

Una aplicación web RESTful expone su información a través de recursos. Además, da la posibilidad al cliente de realizar diferentes acciones sobre estos recursos, como por ejemplo crear otros de nuevos (por ejemplo, crear un nuevo usuario o producto) o cambiar los ya existentes (por ejemplo, modificar la descripción de un producto o su precio). Para hacerlo, se usan las órdenes HTTP en vez de definir operaciones específicas. Así, para pedir los datos de un producto, no se define una operación como por ejemplo `pedirDatosProducto(idProducto)`, sino que usaremos el método HTTP GET, pidiendo los datos en la línea de este protocolo `GET /datosProduc-`

`to?idProducto`. El resultado de la operación puede tener cualquier formato, puesto que HTTP permite enviar cualquier tipo de fichero como respuesta a una petición.

El servicio implementado en una aplicación que hay detrás de una aplicación web RESTful puede no tener ningún tipo de interfaz web y su comunicación con la aplicación RESTful se puede hacer por medio de otros mecanismos de comunicación como por ejemplo sockets, Remote Produce Call (RPC), Remote Method Invocation (RMI) o ejecución directa de programas desde la aplicación RESTful.

Figura 2. Arquitectura de un servicio REST



El diagrama de la figura 2 nos muestra la arquitectura general de un servicio basado en REST, donde una aplicación web RESTful ofrece acceso a una aplicación que se ejecuta en el servidor a través del protocolo HTTP. Los clientes se conectarán con la aplicación web RESTful por medio de este protocolo. En el cliente, hay que tener una aplicación cliente, una librería REST (que puede ser opcional) y una librería HTTP, que establezca la comunicación con el servicio.

1.2.1. Métodos HTTP y operaciones REST

Los métodos HTTP que se usan en una API REST son los que aparecen en la tabla 1. En la tabla 1 se presentan los métodos HTTP utilizados en REST, la operación CRUD a la cual corresponde y una pequeña descripción de lo que tiene que hacer la operación que usa cada método.

Tabla 1. Métodos HTTP y su función en REST

Método	Operación CRUD	Descripción
POST	Create – Creación	Se tiene que utilizar para crear un nuevo recurso que se añade a la colección de recursos actual. También se puede utilizar para modificar, a pesar de que no se recomienda el uso[5].
GET	Read – Lectura	Se tiene que utilizar para pedir los datos de un recurso. A pesar de que GET puede recibir parámetros, no se tendría que usar para operaciones de creación y/o modificación.

[5]Una operación *Idempotente* es la que da siempre los mismos resultados dados los mismos parámetros. HTTP, GET, PUT y DELETE son idempotentes.
Por el contrario, una operación *No idempotente* es la que puede dar resultados diferentes con los mismos parámetros.

Método	Operación CRUD	Descripción
PUT	Update – Actualización	Se tiene que utilizar para actualizar un recurso existente, que sustituye el recurso actual por el recurso que se envía en el mensaje de petición. En el supuesto de que el recurso que envía el cliente sea un recurso nuevo, este método lo crearía e informaría convenientemente el cliente.
PATCH	Partial Update – Actualización parcial	Se tiene que utilizar para modificar un recurso, sin enviarlo entero, solo enviando los datos que se tienen que modificar. Es parecido a PUT, pero tiene que contener instrucciones que indiquen qué datos del recurso original se tienen que modificar.
DELETE	Delete - Eliminar	Se tiene que utilizar para borrar un recurso.

En la tabla 2 se describen los posibles códigos de respuesta para cada una de las operaciones de la tabla 1. Estos códigos corresponden a códigos de protocolo HTTP, tal como se definen en Internet Engineering Task Force (2014).

Referencia bibliográfica

Internet Engineering Task Force (IETF) (2014). Hypertext Transfer Protocol (HTTP/1.1), RFC 7230 a 7235. [en línea]: <https://tools.ietf.org/html/rfc7230> y <https://tools.ietf.org/html/rfc7235>.

Tabla 2. Posibles códigos de respuesta HTTP a cada operación REST

Método	Código de respuesta correcta	Código de respuesta errónea
POST	201 Recurso creado	404 Recurso no encontrado 409 Conflicto, el recurso ya existe
GET	200 Correcto	404 Recurso no encontrado
PUT	200 Correcto 201 Recurso creado	204 No más contenido 404 Recurso no encontrado
PATCH	200 Correcto	204 No hay más contenido 404 Recurso no encontrado
DELETE	200 Correcto	404 Recurso no encontrado

1.3. Implementación de servicios basados en REST

Para implementar un servicio basado en REST, es importante definir cómo tiene que ser. De hecho, un servicio basado en REST lo que hace es exponer una serie de operaciones sobre recursos almacenados en un servidor por medio de una interfaz de programación de aplicaciones (API, de ahora en adelante).

Así pues, uno de los conceptos más relevantes dentro de REST es el recurso. Cualquier información a la cual se puede dar un nombre es un recurso: un documento o imagen, un servicio temporal (por ejemplo, «el tiempo durante los próximos tres días en Barcelona»), una colección de otros recursos, un objeto no virtual (por ejemplo, una persona), etc. De este modo, un concepto que se pueda representar como una referencia a un hipertexto puede ser un recurso. Los recursos se pueden agrupar en colecciones, donde cada colección puede contener varios recursos desordenados. Las colecciones también son recursos por si mismas.

Vemos a continuación algunos ejemplos para aclarar el concepto de recurso. `clientes` puede ser un recurso de tipo colección y `cliente` un recurso único (en un dominio como por ejemplo el bancario). Se puede identificar la colección `clientes` con el URI `/clientes`. Y podemos identificar un único cliente con el URI`/clientes/{clientId}`. Además, un recurso puede tener subcolecciones. Por ejemplo, la subcolección `cuentas` de un `cliente` se puede identificar mediante el URI `/clientes/{clientId}/cuentas`. De manera similar a las colecciones, también se puede acceder a un recurso único dentro de las subcolecciones. Así, una `cuenta` concreta dentro de las cuentas de un `cliente` se podría acceder con `/clientes/{clientId}/cuentas/{cuentaId}`.

Como ya hemos visto en los ejemplos, las API REST usan Uniform Resource Identifiers (URI) para acceder a los recursos. El diseñador de una API REST tiene que definir un modelo de recursos que sea comprensible para los desarrolladores de sus clientes potenciales. Cuando a los recursos se les dan nombres con lógica, una API es intuitiva y fácil de utilizar. Si se hace mal, esta misma API puede ser difícil de usar y entender.

A pesar de que no hay un estándar a la hora de definir los nombres de los recursos de una API REST, ponemos aquí una serie de directrices para hacerlo de la manera más comprensible posible.

Directrices para la definición de URI para los recursos de una API REST

1) Usar nombres (cosas) en vez de verbos (acciones) para definir los recursos. Los nombres tienen una característica que los verbos no tienen, como son los atributos. Algunos ejemplos de nombres pueden ser usuarios del sistema, cuentas de usuario, dispositivos de red, etc. Dentro de los nombres podemos distinguir entre recursos únicos (en singular) y colecciones (en plural).

Ejemplos:

```
http://api.com/gestion-dispositivos
http://api.com/
gestion-dispositivos/dispositivos-gestionados
http://api.com/
gestion-dispositivos/dispositivos-gestionados/{ID}
```

2) La consistencia es clave. Se tienen que usar convenciones de nombres consistentes y formatación de URI con la mínima ambigüedad y máxima legibilidad y mantenimiento. Algunos consejos de diseño son: usar la barra (/) para indicar relaciones jerárquicas, no poner la barra (/) al final del URI, no usar mayúsculas y no usar extensiones en los recursos.

Referencia bibliográfica

Internet Engineering Task Force (IETF) (2005). Uniform Resource Identifier (URI): Generic Syntax, RFC 3986, <https://tools.ietf.org/html/rfc3986>.

Ejemplos:

Jerarquía

```
http://api.com/
gestion-dispositivos/dispositivos-gestionados
http://api.com/
gestion-dispositivos/dispositivos-gestionados/{ID}
```

No se recomienda el uso de la barra al final (la segunda opción no sería correcta)

```
http://api.com/
gestion-dispositivos/dispositivos-gestionados
http://api.com/
gestion-dispositivos/dispositivos-gestionados/
```

Uso de mayúsculas. Se pueden usar en el protocolo y el servidor, pero no en la parte del recurso (el tercer URI no sería válido ya que pone Mi en mayúscula)

```
HTTP://API.ORG/mi-carpeta/mi-doc
http://api.org/mi-carpeta/mi-doc
http://api.org/Mi-carpeta/mi-doc
```

No usamos extensiones de ficheros (la segunda opción no sería correcta)

```
http://api.com/
gestion-dispositivos/dispositivos-gestionados
http://api.com/
gestion-dispositivos/dispositivos-gestionados.xml
```

3) No usamos los nombres de las operaciones CRUD en los nombres de los recursos. Lo que se debe hacer es utilizar el método HTTP correspondiente, como GET para obtener los datos, POST para crearlos, etc. Se utiliza el mismo URI, pero diferente método HTTP y, entonces, el efecto sobre los recursos es diferente.

Ejemplos:

Para obtener todos sus dispositivos

```
HTTP GET http://api.com/gestion-dispositivos/dispositi-
vos-gestionados
```

Para crear un nuevo dispositivo

```
HTTP  POST  http://api.com/gestion-dispositivos/disposi-
tivos-gestionados
```

4) Usar el apartado de consulta URI para hacer filtrado u ordenación de los recursos en lugar de crear nuevas entradas de la API.

La sección de consulta es todo lo que está detrás`?` y para poner más de un parámetro se pone `&`. El formato de esta sección es `?clave1=valor1&clave2=valor2`.

Ejemplos:

```
http://api.com/
gestion-dispositivos/dispositivos-gestionados
```

```
http://api.com/
gestion-dispositivos/dispositivos-gestiona-
dos?region=cat
```

```
http://api.com/
gestion-dispositivos/dispositivos-gestiona-
dos?region=cat&fecha=2020-03-18
```

1.3.1. Formato de las peticiones

Las peticiones a un servicio REST tienen que ir en un mensaje de petición HTTP. Según de qué método HTTP se trate, podremos enviar los datos de manera diferente. Primero, describiremos cómo se pueden enviar los datos y después indicaremos con qué métodos HTTP se puede usar cada mecanismo.

URL

Los datos se pueden enviar en la misma URL de petición. Así, cuando hablamos de recursos concretos dentro de colecciones, podemos usar identificadores para referirnos a un recurso concreto. En un ejemplo anterior, hemos usado la colección `clientes` para referirnos a los clientes de una entidad bancaria. El URI `/clientes` identifica la colección entera, es decir, todos los clientes. La manera de identificar un único cliente sería un URI del tipo `/clientes/{clientID}`, donde tenemos que dar un identificador de cliente para hacer la petición en el campo `{clientID}`. El URI resultante sería `/clientes/1111111H`, donde se piden los datos del cliente identificado por el identificador `11111111H`, que corresponde a un documento nacional de identidad (DNI).

Sección query de la URL

También dentro de la misma URL se pueden enviar datos en la sección query de la URL. La sección query empieza con el símbolo `?` y puede contener varias parejas de `clave=valor`, separadas entre ellas por el símbolo `&`. Así, siguiendo con el ejemplo de los clientes de un banco, podríamos pedir los detalles de un usuario de la manera siguiente: `/clientes/11111111H?info=detalles`, donde la clave `info` puede tomar diferentes valores; en este caso concreto se piden todos los detalles del usuario. Si ponemos co-

mo ejemplo la cuenta bancaria de un usuario, podríamos filtrar los movimientos por fecha de la siguiente manera:`/clientes/11111111H/cuentas/12341234?fecha-ini=1-1-2020&fecha-fin=31-03-2020`

Cuerpo del mensaje de petición

Hay diferentes mecanismos para enviar los datos dentro del cuerpo del mensaje de petición:

- Formato `application/x-www-form-urlencoded`, que es el mismo formato que en la sección query, utilizando el formato de `clave=valor`. Con este formato se envian todos los campos necesarios separados por el símbolo `&`. En este caso, un ejemplo de URL podría ser`/clientes/1111111H/cuentas/12341234` y los parámetros dentro del cuerpo del mensaje serían `fecha-ini=1-1-2020&fecha-fin=31-03-2020`.

- Formato `multipart/form-data`, que es un mensaje multiparte de MIME (Multipurpose Internet Mail Extensions), donde se envían los parámetros dentro del cuerpo del mensaje separados por una cadena de caracteres denominada `boundary`. La ventaja de este formato respecto a los anteriores es que permite enviar más datos, incluso datos en formato binario. Este formato está orientado a enviar ficheros con un formulario que se muestra a un usuario en un navegador.

- Contenido de tipo `application/xml` o (`texto/xml`), `application/json` y otros soportados por el servicio REST (que deben indicarse en las cabeceras de servicio, como veremos más adelante).

Sobre MIME

Internet Engineering Task Force (IETF)(2015). Returning Values from Forms: multipart/form-data, RFC 7578, <https://tools.ietf.org/html/rfc7578>.

XML y JSON

XML quiere decir *eXtensible Markup Language* y sus tipos MIME pueden ser application/XML (cuando está dirigido a un servidor) o text/xml (cuando está dirigido a ser entendido por una persona).

JSON quiere decir *JavaScript Object Notation* y su tipo MIME es application/JSON.

Ejemplo de mensaje `multipart/form-data` enviado dentro de un mensaje HTTP de una orden POST:

```
Content-Type: multipart/form-data; boundary=--73532303139
Content-Length: 834
--73532303139
Content-Disposition: form-data; name="text1"
text 123 abc
--73532303139
Content-Disposition: form-data; name="text2"
xyz
--73532303139
Content-Disposition: form-data; name="file1"; filename="a.txt"
Content-Type: text/plain
Contenido fichero a.txt.
--73532303139
Content-Disposition: form-data; name="file2"; filename="a.html"
Content-Type: text/html
DOCTYPE html><title>Contenido a.html.</title>
```

```
--73532303139
Content-Disposition: form-data; name="file3"; filename="fish.jpg"
Content-Type: image/jpeg
Datos binarios aquí en formato base64
--73532303139--
```

En este mensaje se están enviando dos campos de texto (text1 y text2), un fichero de texto plano denominado `a.txt`, un archivo HTML denominado `a.html` y una imagen denominada `fish.jpg`. Cada fichero indica cuál es su tipo MIME con la cabecera `Content-Type`. Cada tipo de datos está separado por el campo `boundary`, que tiene el valor `--73532303139`, tal y como se define al inicio del mensaje en la cabecera `Content-Type` del mensaje completo.

A continuación, pondremos unos ejemplos de cómo pueden ser las peticiones de cada uno de los métodos HTTP. Se incluyen tanto las líneas de petición como las cabeceras del mensaje y el contenido del cuerpo del mensaje, si lo hay.

Ejemplo de solicitud POST

```
POST /client/json HTTP/1.1
Accept: application/json
Content-Type: application/json
Content-Length: 85
Host: exemple.com
{
  "Id": 12345,
  "Cliente": "Arnau Sola",
  "Cantidad": 3,
  "Precio": 10.00
}
```

En este ejemplo usamos el formato JavaScript Object Notation (JSON) (ECMA, 2017) para enviar los datos. En el apartado «JavaScript Object Notation» se explica en detalle este formato, cogiendo este ejemplo de datos como referencia. En este caso, los datos se envían dentro del cuerpo del mensaje.

Además, encontramos varias cabeceras de petición: `Accept`, que nos indica en qué formato nos puede llegar la respuesta, `Content-Type`, que nos indica el tipo de contenido que enviamos, `Content-Length`, que nos indica la longitud del cuerpo del mensaje y `Host`, que nos dice a qué servidor nos tenemos que conectar.

Ejemplo petición GET

```
GET /assigs/json HTTP/1.1
Accept: application/json
```

```
Host: exemple.com
```

En esta petición GET podemos ver que no hay ningún contenido en el mensaje (el método GET no lo permite) y que se acepta que la respuesta sea en formato JSON (`aplicación/json`). También se puede ver el servidor a quien hacemos la petición.

Ejemplo petición PUT

```
PUT /client/json HTTP/1.1
Accept: application/json, application/xml
Content-Type: application/json
Content-Length: 85
Host: exemple.com
{
  "Id": 12345,
  "Cliente": "Arnau Sola",
  "Cantidad": 1,
  "Precio": 10.00
}
```

Esta petición PUT es muy parecida a la petición POST. Vemos que se envían los datos completos del cliente, cambiando el valor 3 que había en POST por un 1. Esto hará que se modifique todo el recurso en el servidor.

Ejemplo petición PATCH

```
PATCH /client/json HTTP/1.1
Accept: application/json, application/xml
Content-Type: application/json
Content-Length: 85
Host: exemple.com
{
  "Id": 12345,
  "Precio": 15.00
}
```

Esta petición PATCH es un subconjunto de las hechas en POST y PUT. Aquí sólo se envía el ID y el precio, que deben ser modificados. El resto de los datos del recurso se mantienen.

Ejemplo petición DELETE

```
DELETE /cliente/json/1234 HTTP/1.1
Accept: application/json, application/xml
Host: exemple.com
```

Esta petición DELETE contiene los datos del recurso que se tiene que borrar en la URL. En este caso, se quiere borrar al cliente con id 1234. No hay contenido del mensaje.

La tabla 3 muestra un resumen de cómo se pueden enviar los datos según el método HTTP utilizado. En algunos casos se pueden combinar varias opciones. Algunos métodos HTTP podrían soportar más maneras de enviar los datos, pero entonces no seguirían las directrices REST. Es por eso que no aparecen en la tabla, a pesar de que la especificación del método en HTTP lo soportaría.

Tabla 3. Cómo enviar las peticiones según el método HTTP utilizado

Método	Opciones de petición
POST	URL, que indica el recurso a crear. Sección query, que permite pasar parámetros. Cuerpo del mensaje de la petición. En este caso, se puede enviar un fichero del tipo soportado por el servicio `application/x-www-form-urlencoded` o `multipart/form-data`.
GET	URL, que indica el recurso a consultar. Sección query, que permite pasar parámetros. En este caso, no se puede enviar nada al cuerpo del mensaje porque el método no lo soporta.
PUT	URL, que indica el recurso a reemplazar o crear. Cuerpo del mensaje de solicitud con un archivo de tipo admitido por el servicio REST.
PATCH	URL, que indica el recurso a modificar parcialmente. Cuerpo del mensaje de solicitud con un archivo de tipo admitido por el servicio REST.
DELETE	URL, que indica el recurso a eliminar.

1.3.2. Javascript Object Notation (JSON)

JavaScript Object Notation (JSON) (European Computer Manufacturers Association, 2017) es un formato ligero de intercambio de datos. Se trata de un formato fácil de generar por parte de las personas y de interpretar por parte de las máquinas. Además, es un formato de texto independiente del lenguaje de programación, pero, a la vez, usa convenciones habituales para lenguajes como C, C++, Java, Python y otros muchos. Estas propiedades lo hacen muy adecuado como lenguaje de intercambio de datos.

JSON se basa en dos estructuras:

- Una colección de parejas nombre/valor. En muchos lenguajes de programación esto se representa como un objeto, registro, estructura, diccionario, tabla de hash o lista con claves, entre otros.

- Una lista ordenada de valores en muchos lenguajes esto se representa como un vector, matriz, lista o secuencia.

```
{
  "Id": 12345,
  "Cliente": "Arnau Sola",
  "Cantidad": 3,
  "Precio": 10.00
}
```

En el ejemplo, se representa un objeto con los datos de un cliente. Las claves ({}) indican el inicio y el final del objeto. Dentro encontramos un conjunto de nombres/valores. El primer nombre que encontramos es «`Id`» con valor 12345. El nombre se separa del valor con el símbolo «:». Cuando un nombre o un valor tiene el símbolo «"», es una cadena de caracteres (por ejemplo, «`Id`» o «`Arnau Sola`»). Para separar parejas nombre/valor entre sí, usamos el símbolo «,». También podemos representar números, tanto valores enteros (por ejemplo, 12345 o 3) como reales (por ejemplo, 10.00). Se pueden incluir objetos dentro de objetos y también usar otros tipos de datos como pueden ser valores booleanos (`true`, `false`). Para una descripción más detallada, consultad European Computer Manufacturers Association (2017).

1.3.3. Formato de las respuestas

Las respuestas proporcionadas por un servicio REST se pueden indicar en las cabeceras de petición (ved el subapartado 1.3.1). Por ejemplo, si queremos que el servicio devuelva un fichero de tipo texto, en la petición lo indicaremos mediante la cabecera `Accept`, donde tenemos que poner como valor `text/plain`. En esta cabecera, la aplicación cliente indica cuáles son los formatos de respuesta del servicio aceptados. Se pueden poner diferentes valores, separados por comas.

Ejemplo de cabecera `Accept` de la petición al servicio REST que indica el formato o formatos soportados por el cliente.

Los más habituales son ficheros en formato JSON o XML, que son los que aparecen en el ejemplo. En la primera línea se indican ambos formatos, en la segunda línea solo JSON, y en la tercera solo XML. En una petición solo se envía una cabecera `Accept`, a pesar de que puede contener más de un valor, como se ve en el primer ejemplo.

```
Accept: application/json, application/xml
Accept: application/json
Accept: application/xml
```

A continuación se muestra un ejemplo de respuesta de un servicio REST para cada método HTTP.

Ejemplo de respuesta POST

```
HTTP/1.1 200 OK
Content-Length: 19
Content-Type: application/json
{"success":"true"}
```

En este ejemplo se responde con el código que indica que la petición ha ido bien. Además, encontramos dos cabeceras que indican la longitud y el tipo de contenido. Finalmente, está el contenido del mensaje, que también indica que la petición ha funcionado correctamente. En este caso los datos se envían en formato JSON, con una única pareja nombre/valor dentro del objeto con un valor de tipo booleano.

Ejemplo de respuesta GET

```
HTTP/1.1 200 OK
Content-Type: application/json
{
    "público": {
        "alumnos": "delegados",
        "asignaturas": "todas",
        "profesores": "todos"
    },
    "privado": {
        "yo": "perfil",
        "asignaturas": "XAI"
    }
}
```

En este ejemplo se responde con el código que indica que la petición ha ido bien. En este caso, se devuelven datos en formato JSON referentes a objetos asociados a asignaturas, profesores y alumnos. En este caso hay dos objetos, uno llamado «público» y otro llamado «privado», que contienen varios pares de objetos nombre/valor. Todos los valores son de tipo cadena de caracteres.

Ejemplo de respuesta PUT, PATCH y DELETE

```
HTTP/1.1 200 OK
Content-Type: application/xml; charset=utf-8
<?xml version="1.0" encoding="utf-8"?>
<Response>
    <ResponseCode>0</ResponseCode>
    <ResponseMessage>Success</ResponseMessage>
</Response>
```

En este ejemplo se responde con el código que indica que la petición ha ido bien, y los datos se devuelven en formato XML. Este formato está explicado con más detalle en el apartado «eXtensible Markup Language (XML)».

1.3.4. eXtensible Markup Language (XML)

eXtensible Markup Language (XML) (World Wide Web Consortium, 2006) es un formato estandarizado por W3C[6], que describe cómo tienen que ser los documentos XML. Estos documentos están formados por unidades de almacenamiento llamadas entidades, que pueden contener diferentes tipos de datos. Los datos formalizados están compuestos por caracteres, algunos de ellos definiendo el marcaje (*markup*, en inglés) y otros definiendo los datos. El marcaje codifica una descripción de la distribución del almacenamiento del documento, como también la estructura lógica. Con XML se proporciona un mecanismo para poder imponer restricciones tanto a la distribución como la estructura lógica de los datos.

(6)World Wide Web Consortium (W3C),<https://www.w3.org/>. [Fecha de consulta: marzo de 2020]

Nota

XML deriva de SGML, que quiere decir *Generalized Markup Language*. Por definición, los documentos XML son conformes a SGML.

```
<?xml version="1.0" encoding="utf-8"?>
<Response>
    <ResponseCode>0</ResponseCode>
    <ResponseMessage>Success</ResponseMessage>
</Response>
```

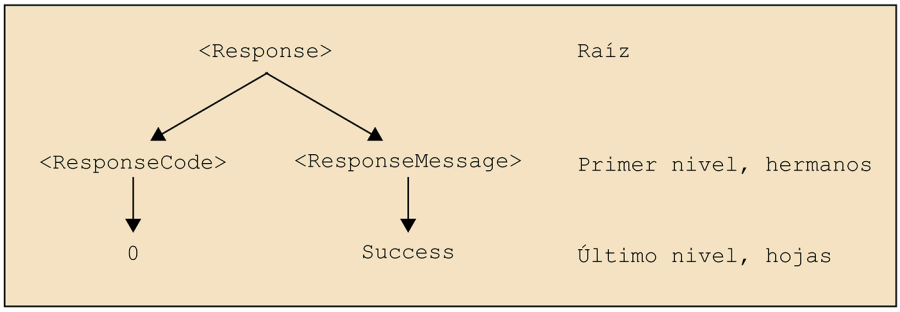
En el ejemplo, se representa un código de respuesta exitoso a una petición PUT, PATCH o DELETE que hemos visto en el apartado 1.3.2. La primera línea indica que se trata de un documento XML de la versión 1.0 y codificado con UTF-8. Esta línea es obligatoria en los documentos XML válidos. Después encontramos el elemento raíz `Response`. El inicio de un elemento se marca con los símbolos «<» y «>» y el final con «</» y «>». Dentro del elemento raíz encontramos los subelementos `ResponseCode` y `ResponseMessage`, que son hijos de `Response` y hermanos entre ellos. Los elementos tienen una relación jerárquica desde la raíz a las hojas. Todos los subelementos de un mismo nivel son hermanos (*siblings*, en inglés). Finalmente, encontramos los valores `0` y `Success`, que corresponden a las hojas, que son elementos finales, que no tienen hijos y que pueden contener valores de tipo cadena de caracteres, números decimales o reales y otros tipos de datos básicos definidos en el estándar XML.

UTF-8

UTF-8 quiere decir *8-bit Unicode Transformation Format*. Es un formato de codificación de caracteres definido por ISO (*Organización Internation for Standardization*).

La figura 3 muestra la relación entre los diferentes elementos XML que aparecen en el ejemplo.

Figura 3. Jerarquía de un documento XML



En este caso, hemos visto que la respuesta tiene formato XML. También sería posible recibir una respuesta en formato JSON como en el ejemplo con el método POST visto en el apartado «Javascript Object Notation (JSON)».

1.4. Comparación con otros tipos de servicios web

Los otros servicios web muy utilizados han sido los basados en el Simple Object Access Protocol (SOAP) [7]. SOAP es un estándar del World Wide Web Consortium (W3C), que define un marco de trabajo basado en XML para comunicar procesos remotos. Tanto los mensajes que se intercambian como el mecanismo para definir el servicio usan XML para el envío y la descripción de los datos. En este apartado hacemos un pequeño resumen de SOAP y otras tecnologías relacionadas y comparamos sus características con los servicios web REST.

(7)World Wide Web Consortium (W3C) (2007). Simple Object Access Protocol (SOAP) Versión 1.2, <https://www.w3.org/TR/soap12/>

1.4.1. Servicios web basados en SOAP

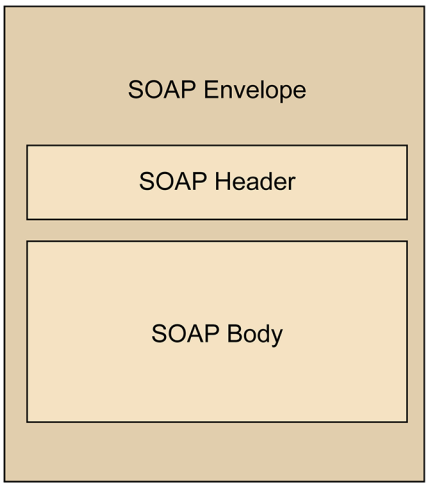
El estándar SOAP es un protocolo de comunicación de aplicaciones que define el formato de los mensajes de petición y respuesta. Este protocolo es independiente de la plataforma y el lenguaje de programación en que están implementados cliente y servidor, puesto que la comunicación se basa en mensajes expresados en formato XML. Estos mensajes se intercambian principalmente sobre el protocolo HTTP, a pesar de que podrían enviarse sobre otros protocolos de nivel de aplicación.

En SOAP los mensajes entre cliente y servicio web siempre se envían en formato XML. A continuación se describen algunas de las características de los servicios SOAP, como el envío de peticiones y respuestas o cómo se definen las operaciones de un servicio.

Mensajes de solicitud y respuesta SOAP

La figura 4 muestra un diagrama del formato de las peticiones y respuestas SOAP. El primer elemento es el SOAP Envelope (`<Envelope>`), que rodea todos los datos de una petición y/o respuesta. Los otros elementos presentes en un mensaje SOAP son el SOAP Header y el SOAP Body.

Figura 4. Diagrama de formato de peticiones y respuestas SOAP



El SOAP Header (`<Header>`) es un elemento opcional del mensaje SOAP y contiene información relacionada con el servicio que tiene que ser procesada por los nodos SOAP a lo largo del flujo del mensaje. La información definida solo depende del servicio web, es decir, los datos que se envían a la cabecera solo los entiende este servicio concreto.

El SOAP Body (`<Body>`) es un elemento obligatorio que contiene información destinada al receptor último del mensaje. Este elemento y sus subelementos se utilizan para intercambiar información entre el emisor SOAP y el receptor último del mensaje SOAP. SOAP define un subelemento, el SOAP Fault (`<Fault>`), que se usa para enviar errores que se hayan producido. El resto de elementos que aparecen en el SOAP Body los define el servicio web y son las operaciones, los parámetros y los resultados de las operaciones que se ofrecen.

A continuación vemos un ejemplo de mensaje de petición SOAP que se envía dentro de una orden POST de HTTP. El cuerpo del mensaje se envía en formato `application/soap+xml`, que indica que se envía en formato XML, siguiendo las directrices del estándar SOAP.

Dentro del mensaje SOAP podemos ver que hay un elemento `<Header>` que contiene un subelemento que indica que el tiempo máximo para responder a la petición es de 10.000 y que todos los nodos que procesen esta petición SOAP tienen que entender esta cabecera, tal como se indica con el atributo `mustUnderstand`. En caso de que un nodo no lo entienda, devolverá un SOAP Fault y no se continuará con el procesamiento del servicio web.

A continuación encontramos el elemento `Body`, donde se envía el nombre de una acción, expresado como cadena de caracteres, a un servicio web que tiene que devolver su precio.

```
POST /acciones HTTP/1.1
Host: www.exemple.org
```

```
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
    <m:tempsMax value="10000" xmlns:m="http://www.exemple.org/accio"
        mustUnderstand="true"/>
  </soap:Header>
  <soap:Body xmlns:m="http://www.exemple.org/accio">
    <m:PrecioAccion>
      <m:NombreAccion>Inditex</m:NombreAccion>
    </m:PrecioAccion>
  </soap:Body>
```

La respuesta viaja dentro de un mensaje de respuesta HTTP, también con formato `application/soap+xml`. El mensaje de respuesta SOAP también contiene un elemento `Envelope` y dentro encontramos un elemento `Body` que lleva la respuesta del servicio web que devuelve el valor de la acción, en este caso con formato de número real.

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope">

  <soap:Body xmlns:m="http://www.exemple.org/accion">
    <m:RespuestaPrecioAccion>
      <m:Preu>34.5</m:Precio>
    </m:RespuestaPrecioAccion>
  </soap:Body>

</soap:Envelope>
```

Ahora que ya hemos visto cómo viajan tanto la petición como la respuesta de un servicio web SOAP dentro de los mensajes del protocolo HTTP, explicaremos con algo más de detalle cómo se tienen que definir las operaciones del servicio, el formato de los parámetros y respuestas del servicio y también cómo se asocia este servicio a un protocolo de nivel de aplicación concreto, en este caso, HTTP.

Definir servicios web con Web Services Description Language (WSDL)

Web Services Description Language (WSDL) 2.0 es una recomendación del W3C que describe un lenguaje basado en XML para describir servicios web. Para hacerlo se define un modelo abstracto de qué es lo que ofrece el servicio. Además, se definen los criterios de conformidad para los documentos descritos en este lenguaje.

Sobre WSDL

World Wide Web Consortium (W3C)(2007). Web Services Description Language (WSDL) Versión 2.0, <https://www.w3.org/TR/wsdl20/>

Los elementos principales de un WSDL son:

- **description**. Es el elemento raíz. El resto de elementos están dentro de él.
- **documentation**. Es un elemento opcional que puede contener una descripción legible por parte de las personas.
- **types**. Contiene la especificación de los tipos de datos intercambiados entre el cliente y el servicio web. Por defecto, estos tipos de datos se definen con un XML Schema[8].
- **interface**. Describe qué operaciones tiene el servicio y qué mensajes se intercambian para cada operación (input / output). También permite definir los posibles mensajes de error.
- **binding**. Define cómo se accede al servicio web a través de la red. Normalmente este elemento define cómo conectar al servicio a través de HTTP (a pesar de que no es la única posibilidad).
- **service**. Define dónde se puede acceder al servicio web en la red. Normalmente contiene una URL donde se puede encontrar el servicio.

[8]**World Wide Web Consortium (W3C)**(2012). XML Schema Definition Language (XSD) 1.1, <https://www.w3.org/TR/xmlschema11-1/>

A continuación, se puede ver un ejemplo de WSDL donde aparecen todos los elementos descritos.

```
<?xml version="1.0" encoding="utf-8" ?>
<description
    xmlns=           "http://www.w3.org/ns/wsdl"
    targetNamespace= "http://www.exemple.org/accion"
    xmlns:tns=       "http://www.exemple.org/accion"
    xmlns:stns =     "http://www.exemple.org/accion/esquema"
    xmlns:wsoap=     "http://www.w3.org/ns/wsdl/soap"
    xmlns:wsdlx=     "http://www.w3.org/ns/wsdl-extensiones" >
  <documentation>
    This is the web service documentation.
  </documentation>
  <types>
    <xs:schema
```

```
        xmlns:xs=        "http://www.w3.org/2001/XMLSchema"
        targetNamespace= "http://www.exemple.org/accion/esquema"
        xmlns=           "http://www.exemple.org/accion/esquema">
      <xs:element name="PeticionPrecioAccion" type="tipoPrecioAccion"/>
      <xs:complexType name="tipoPrecioAccion">
        <xs:sequence>
          <xs:element  name="NombreAccion"   type="xs:string"/>
        </xs:sequence>
      </xs:complexType>
      <xs:element name="RespuestaPrecioAccion" type="tipoRespuestaPrecioAccion"/>
      <xs:complexType name="tipoRespuestaPrecioAccion">
        <xs:sequence>
          <xs:element  name="Precio"   type="xs:float"/>
        </xs:sequence>
      </xs:complexType>
    </xs:schema>
  </types>
  <interface  name = "PrecioAccionInterface" >
    <operation name="PrecioAccion"
            pattern="http://www.w3.org/ns/wsdl/in-out"
            style="http://www.w3.org/ns/wsdl/style/iri"
            wsdlx:safe = "true">
      <input    messageLabel="In"  element="stns:PeticionPrecioAccion" />
      <output   messageLabel="Out" element="stns:RespuestaPrecioAccion" />
    </operation>
  </interface>
  <binding name="PrecioAccionSOAPBinding"
          interface="tns:PrecioAccionInterface"
          type="http://www.w3.org/ns/wsdl/soap"
          wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/"
          wsoap:mepDefault="http://www.w3.org/2003/05/soap/mep/request-response">
    <operation ref="tns:PrecioAccion"/>
  </binding>
  <service
       name     ="ServicioPrecioAccion"
       interface="tns:PrecioAccionInterface">
     <endpoint name ="PrecioAccionEndpoint"
            binding ="tns:PrecioAccionSOAPBinding"
            address ="http://www.exemple.org/accion"/>
  </service>
</description>
```

Los elementos del WSDL están relacionados entre ellos. Por ejemplo, los atributos `interface` y `binding` del elemento `service` son los elementos que `interface` y `binding` (que coinciden con los valores definidos en el atributo `name`), respectivamente. El elemento `operation` dentro de la `binding` (en este caso sólo hay uno, pero podría haber más) coinciden con el

elemento `operation` definido dentro de la `interface`. Por último, el `input` y el `output` definido en el elemento `operation`, deben ser el tipo de datos definidos dentro del elemento `types`.

Los valores `tns` y `stns` que hay delante de algunos de los nombres corresponden a las referencias a los espacios de nombres donde se definen los diferentes elementos. La referencia a los espacios de nombres la encontramos normalmente en el elemento raíz del documento XML, en este caso concreto, en `description`.

A continuación encontramos los espacios de nombres definidos en este WSDL:

```
xmlns=          "http://www.w3.org/ns/wsdl"
xmlns:tns=      "http://www.exemple.org/accion"
xmlns:stns =    "http://www.exemple.org/accion/esquema"
xmlns:wsoap=    "http://www.w3.org/ns/wsdl/soap"
xmlns:wsdlx=    "http://www.w3.org/ns/wsdl-extensions"
```

El primero es el espacio de nombres por defecto de este documento, que hace referencia al espacio de nombres de WSDL. Después de `xmlns` (que quiere decir espacio de nombres XML, xml namespace), no hay ningún nombre cualificado. Esto es porque queremos que los elementos del espacio de nombres WSDL no necesiten ningún prefijo para referenciarlos dentro del documento. El siguiente que encontramos es `xmlns:tns`, que se corresponde con el espacio de nombres que definimos en este documento XML. Así, cualquier elemento que pertenece a este espacio de nombres, habrá que identificarlo como `tns:nombreElemento`. Pasa lo mismo con `xmlns:stns`, que hace referencia a los tipos de datos que se definen en este documento y que se usan en la operación. Finalmente, tenemos otros dos espacios de nombres externos, `xmlns:wsoap` y `xmlns:wsdlx`, que hacen referencia al espacio de nombres de SOAP y al de las extensiones de WSDL, respectivamente. Algunos elementos de estos espacios de nombres son necesarios para definir características específicas del servicio.

Comparación entre servicios web SOAP y REST

Para comparar estos dos tipos de servicios web, nos fijaremos en algunas de sus características principales:

- Formato de peticiones y respuestas
- Protocolo de nivel de aplicación en el cual se envían los datos
- Definición de las operaciones del servicio
- Seguridad

En primer lugar, hablaremos del formato de las peticiones y las respuestas en ambos tipos de servicios. Mientras que SOAP define estructuras complejas para enviar estas peticiones y respuestas, siempre en formato XML, REST es mucho más flexible. En REST, no hay un formato único para enviar las peticiones o

las respuestas. De hecho, en algunos casos no es ni siquiera necesario enviar ningún mensaje de petición y con la misma URL de acceso al recurso se pueden pasar todos los datos necesarios. Veámoslo con un ejemplo:

```
Petición SOAP, pedir los detalles del usuario con ID 12345
 (viajará en una petición POST de HTTP)
<soap:Envelope ...>
   <soap:body pb="http://www.exemple.org/agenda">
      <pb:GetDetallesUsuario>
   <pb:IDUsuario>12345</pb:IDUsuario>
      </pb:GetDetallesUsuario>
   </soap:Body>
</soap:Envelope>

Petición REST, para pedir los datos de este mismo usuario
GET http://www.exemple.org/agenda/DetallesUsuario/12345 HTTP/1.1
```

Como se puede ver, la petición hecha con REST, donde solo es necesaria la URL, es mucho más simple tanto desde el punto de vista de la cantidad de datos enviados como del procesamiento de estas que la de SOAP, donde se envían muchos elementos XML que no son relevantes para el servicio.

Los mensajes de respuesta pueden ser igual de complejos en los dos tipos de servicio web, puesto que REST puede enviar cualquier tipo de fichero, incluso en formato XML. La diferencia principal es que SOAP solo acepta XML, y REST puede aceptar diferentes tipos de datos.

Ambos tipos de servicios web pueden funcionar sobre el protocolo HTTP. De hecho, REST solo funciona sobre HTTP mientras que SOAP podría funcionar sobre otros protocolos, tal como se define en SOAP Versión 1.2 Parte 2: Adjuntos donde se describen diferentes estrategias de envío y recepción de mensajes.

También hemos visto que SOAP define las características de sus servicios (formato de los mensajes de envío y respuesta, tipo de envío utilizado, protocolo de nivel de aplicación, dirección del servicio, etc.) en un WSDL. ¿Se podría hacer esto mismo con REST? La respuesta es que sí, en la versión 2.0 de WSDL. La manera de hacerlo es en el elemento `binding`, donde podemos indicar que usamos el tipo de `binding` correspondiente a HTTP con `http://www.w3.org/ns/wsdl/http` y como operación del servicio un método HTTP, como por ejemplo, GET (`whttp:method="GET"`). A continuación veamos un ejemplo de cómo podría ser un elemento `binding` que defina estos datos para un servicio REST modelo.

```
<wsdl:binding name="ServeiHTTPBinding"
     type="http://www.w3.org/ns/wsdl/http"
     interface="tns:InterfazServicio">
    <wsdl:operation ref="tns:Servicio" whttp:method="GET"/>
```

```
    </wsdl:binding>
```

Desde el punto de vista de la seguridad, REST funciona sobre HTTP, que soporta autenticación básica y seguridad en la comunicación mediante Transport Layer Security (TLS)[9]. En cambio, SOAP tiene el WS-SECURITY[10], que es un estándar que define toda una serie de mecanismos de seguridad para servicios web.

(9)**Internet Engineering Task Force (IETF)**(2018). The Transport Layer Security (TLS) versión 1.3,<https://tools.ietf.org/html/rfc8446>.

(10)**Organization for the Advancement of Structured Information Standards (OASIS)**(2006). Web Services Security: SOAP Message Security 1.1 (WS-Security), <https://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf>.

Conclusiones de los servicios REST y SOAP

En este apartado, hemos visto que los servicios web SOAP y REST comparten algunas características, como por ejemplo el uso de HTTP como protocolo preferido para el envío de sus datos y otras bastante diferentes como son los formatos de envío de las peticiones y respuestas. Este último punto ha hecho que los servicios web basados en REST sean los más utilizados actualmente. La flexibilidad que proporcionan con peticiones mucho más simples o la no obligación de usar un formato como XML han permitido que este tipo de servicios se hayan expandido y que sea posible procesarlos fácilmente con cualquier tipo de dispositivo, especialmente desde dispositivos móviles. Además, la posibilidad de recibir solo una parte de los datos e ir enlazando con el resto a través de URL también les permite gestionar la transferencia de datos más fácilmente.

2. Servicios REST con lenguaje Java

El lenguaje Java proporciona una especificación llamada JAX-RS, definida en el documento JSR 311 (Sun Microsystems, 2008). Esta especificación define cómo el lenguaje Java da soporte a los servicios web basados en REST. En esta especificación se describe cómo acceder a los recursos, definiendo anotaciones Java[11], que apuntan a clases de tipo recurso, a métodos HTTP y a parámetros y resultados a los llamamientos a los métodos HTTP.

[11]Oracle Corporation (2014). The Java Tutorial, Annotations, <https://docs.oracle.com/javase/tutorial/java/annotations/index.html>.

Referencia bibliográfica

Sun Microsystems (2008). JAX-RS: Java™ API for RESTful Web Services. [en línia]: <https://download.oracle.com/otn-pub/jcp/jaxrs-1.0-fr-eval-oth-JSpec/jaxrs-1.0-final-spec.pdf>.

En la tabla 4 se describen algunas de las anotaciones utilizadas en JAX-RS y una pequeña descripción de estas[12].

Tabla 4. Cómo enviar las peticiones según el método HTTP utilizado

Anotación	Descripción
@Path	La clase raíz de acceso a un recurso con lenguaje Java ha de tener esta anotación para indicar en qué URL se puede encontrar este recurso. Ejemplo: `@Path ("/hola")` la dirección base de nuestro recurso será `http://servidor/hola`. Esta anotación también se puede encontrar dentro de los métodos, y entoncesel valor que haya dentro de la anotación se concatenará en el @Path original.
@método_HTTP	Cada método HTTP tiene su propia anotación. Así, podemos encontrar `@POST`, `@GET`, etc. Esta anotación dentro del código del servicio REST indica con qué método HTTP responderemos a una operación concreta.
@tipoParam	Las anotaciones acabadas en Param nos indican cómo se pueden recibir los parámetros. Así, QueryParam indica que se reciben en la parte de la query de la URL, después del símbolo `?`, FormParam que se reciben dentro del cuerpo del mensaje con la clave de formato - valor o PathParam que son parte de la URL (se identifican dentro de corchetes {} en la anotación @Path correspondiente a ese método).
@Consumes, @Produces	Estas dos anotaciones indican qué datos se reciben (`@Consumes`) y cuáles se devuelven (`@Produces`). Dentro de estas anotaciones, se debe colocar el tipo MIME correspondiente. Ejemplos: `application/json` o `text/html`. La clase Java MediaType define algunos de estos tipos de datos.

Las anotaciones Java

Las anotaciones en el lenguaje de programación Java tienen el formato (@nombre), donde el símbolo @ indica al compilador Java que lo que encontrará es una anotación.

Una anotación es un metadato sintáctico que se puede aplicar a una clase, método u otros elementos de este lenguaje.

[12]Oracle Corporation (2013). Java Enterprise Edition 7 Specification APIs, <https://docs.oracle.com/javaee/7/api/overview-summary.html>

Veamos un ejemplo de cómo se puede implementar un servicio REST donde se responde a los métodos GET y POST con diferentes configuraciones de paso de parámetros. Nos centraremos en estos dos métodos porque son los más fáciles de probar directamente desde un navegador (tanto desde la barra de direcciones como desde un formulario HTML).

Lo primero que encontramos en el servicio son las clases JAX-RS que usaremos dentro de nuestra clase Java, que se denomina «hola». Los nombres de los paquetes (*packages*, en inglés) Java empiezan con `javax.ws.rs`, que identifica la biblioteca Java (`javax`) que implementa servicios web (`ws`) basados en REST (`rs`).

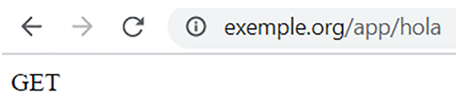
Las clases que incluimos son algunas de las que se corresponden con las anotaciones que han aparecido en la tabla 4. Tenemos el `Path`, los métodos `GET` y `POST`, los tipos de parámetros `FormParam`, `PathParam` y `QueryParam` y `Consumes` y `Produces`. También tenemos la clase del subpaquete `core`, `MediaType`, que permite identificar los tipos de datos dentro de `Consumes` y `Produces`.

```
import javax.ws.rs.Consumes;
import javax.ws.rs.Produces;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.FormParam;
import javax.ws.rs.PathParam;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;
@Path("hola")
public class hola {
   /* Métodos Java que responden a GET y a POST */
}
```

El primer método que implementamos es el `GET`. Este método no recibe ningún dato y devuelve (`Produces`) datos de tipos `MediaType.TEXT_HTML`. La implementación del método es muy sencilla, puesto que devuelve una página HTML que muestra la palabra GET. La URL para llamar a este servicio podría ser `http://exemple.org/app/hola`. En la figura 5 encontramos un ejemplo de respuesta.

```
    @GET
    @Produces(MediaType.TEXT_HTML)
    public String getHtml() {
        return "<html><head/><body>GET</body></html>";
    }
```

Figura 5. Ejemplo de respuesta de la operación getHtml en un explorador



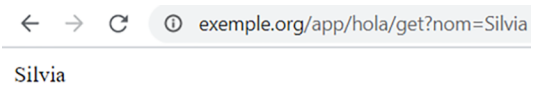
El segundo método que implementamos es también el `GET`. No es posible tener dos métodos que respondan a GET con la misma URL, por lo que en esto agregamos la ruta GET, así que en este hemos añadido el path «get». Además,

pasamos un parámetro por la sección query de la URL. Este parámetro se llama «nom» y se asocia a una variable de tipo String que se denomina nombre. También devuelve (`Produces`) datos del tipo `MediaType.TEXT_HTML`.

La implementación del método es muy sencilla, ya que devuelve una página HTML que muestra el nombre del usuario que se ha pasado como parámetro. La URL para llamar a este servicio podría ser `http://exemple.org/app/hola/get?nom=Silvia`. En la figura 6 encontramos un ejemplo de respuesta.

```
@GET
@Path("/get")
@Produces(MediaType.TEXT_HTML)
public String getQuery(@QueryParam("nom") String nombre) {
    return "<html><head/><body>"+nombre+"</body></html>";
}
```

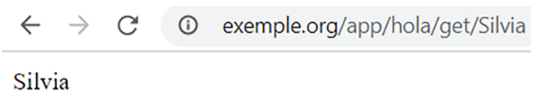
Figura 6. Ejemplo de respuesta de la operación getQuery en un navegador



El tercer método que implementamos es también el `GET`. En este caso, además de añadir el path «get», indicamos que hay un parámetro que se pasa directamente por la URL (PathParam), que da como path resultante «get/{nombre}». Este parámetro se llama nombre y se asocia a una variable de tipo String que también se denomina nombre. También devuelve (`Produces`) datos del tipo `MediaType.TEXT_HTML`. La implementación del método es muy sencilla, puesto que devuelve una página HTML que muestra el nombre del usuario que se ha pasado como parámetro. La URL para llamar a este servicio podría ser `http://exemple.org/app/hola/get/Silvia`. En la figura 7 encontramos un ejemplo de respuesta.

```
@GET
    @Path("/get/{nombre}")
    @Produces(MediaType.TEXT_HTML)
    public String getPath(@PathParam("nombre") String nombre) {
        return "<html><head/><body>"+nombre+"</body></html>";
    }
```

Figura 7. Ejemplo de respuesta de operación getPath en un navegador



El último método que implementamos es el `POST`. En este caso, se reciben datos en el mensaje de petición (`Consumes`) y también se devuelven (`Produces`). Los datos se pasan como datos de formulario HTML (`application/x-www-form-urlencoded`) y se devuelven como `MediaType.TEXT_HTML`. En

el formulario se pasa un parámetro que se denomina nombre como FormParam. Este parámetro se asocia a una variable de tipo String que también se denomina nombre. La implementación del método es muy sencilla, puesto que devuelve una página HTML que muestra el nombre del usuario que se ha pasado como parámetro. La URL para llamar a este servicio podría ser `http://exemple.org/app/hola`, pero en este caso se tiene que enviar la petición en un mensaje POST de HTTP. El servidor sabe a qué método del servicio tiene que llamar a partir de la URL y el método HTTP utilizado, por eso tiene la misma URL de acceso al recurso que el primer método GET que hemos visto.

```
@POST
    @Consumes("application/x-www-form-urlencoded")
    @Produces(MediaType.TEXT_HTML)
    public String postHtml(@FormParam("nombre") String nombre) {
        return "<html><head/><body>"+nombre+"</body></html>";
    }
```

Finalmente, ponemos un extracto de un formulario HTML que permite conectarse con el servicio REST que hemos implementado en este apartado.

```
<form action="http://exemple.org/app/hola" method="POST"
     enctype="application/x-www-form-urlencoded">
  <input type="text" name="nombre"/>
  <input type="submit" value="Enviar"/>
```

En este formulario tenemos que indicar la URL del servicio (atributo *action*, `http://exemple.org/app/hola`), el método de envío (atributo *method*, en este caso tiene que ser POST) el tipo de codificación de los datos (atributos *enctype*, `application/x-www-form-urlencoded`) y dentro del formulario debe haber un campo que se llame `nombre`. En este caso, es un campo de tipo texto y el atributo *name* tiene el valor `nombre`, que es el valor esperado por el servicio. Si no lo definimos exactamente así, el mensaje HTTP enviado no coincidirá con el que espera el servicio y la petición fallará. En las figuras 8 y 9 encontramos el formulario y la respuesta dada por el servicio.

Figura 8. Formulario

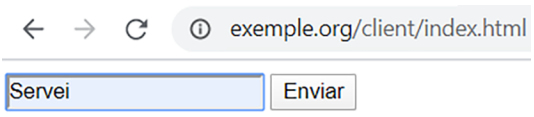
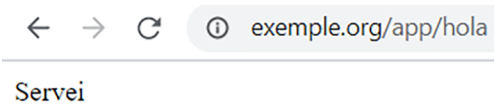


Figura 9. Ejemplo de respuesta de la operación postHtml en un navegador



Un posible error que se podría producir en el llamamiento con POST es que el campo de formulario no se llame `nombre`. En este caso el servicio no encontrará el valor de la variable correspondiente y nos devolverá un valor *null* en la página de respuesta. Otro problema que se podría producir es que, si ponemos como un método GET en vez de POST, se llamará a la primera operación GET de nuestro servicio y nos mostrará el mensaje GET en vez de mostrar los datos que hemos escrito en el formulario. Otro posible error es que llamemos al servicio con una URL indefinida (por ejemplo, `http://exemple.org/app/hola/hola3`) y esto nos daría un error HTTP de recurso no encontrado (código 404). Finalmente, si llamamos al servicio con una URL que no corresponde con el método HTTP, el error que nos daría es método no soportado (código 405).

2.1. Operaciones propias del servidor

Hemos mostrado un ejemplo muy simple de cómo implementar un servicio web REST con lenguaje Java. En este apartado veremos qué más se puede hacer con lenguaje Java para tener un servicio web REST que responda al resto de métodos HTTP, qué otros tipos de formato de datos de peticiones y respuestas podemos tener, etc.

Hay que destacar que lo que hemos visto hasta ahora es independiente del tipo de servidor con el cual se trabaje, puesto que se trata de una especificación Java que los diferentes fabricantes pueden implementar a su manera.

2.1.1. Creación del servicio

Para crear un servicio con JAX-RS, hay que crear una clase que amplíe la clase abstracta `Application` (dentro del paquete `javax.ws.Rs.core`). En esta clase se tiene que definir el recurso raíz del servicio y registrar las clases que responden a los diferentes métodos HTTP. Para el servicio que se ha creado en el apartado 2, la clase derivada de `Application` podría ser la siguiente:

```
import java.util.Set;
import javax.ws.rs.core.Application;
@javax.ws.rs.ApplicationPath("app")
public class ApplicationConfig extends Application {
    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> resources = new java.util.HashSet<>();
        resources.add(hola.class);
        return resources;
    }
}
```

En esta clase se está definiendo la raíz del servicio (`app`) y se añade la clase `hola.class`, que implementa las operaciones del servicio del apartado 2, a los recursos disponibles en dicho servicio.

2.1.2. Definición de operaciones

Ya hemos visto en el apartado 2 cómo se definen las operaciones de un servicio web REST que utiliza JAX-RS. Aquí detallaremos todos los componentes que aparecen en él y su función, así como también las alternativas con las que nos podemos encontrar.

```
@METODO_HTTP
@Path("/URL_Access/{PATH_PARAM}")
@Consumes(TIPOS_MIME_PETICION)
@Produces(TIPOS_MIME_RESPUESTA)
public String nomMetodo(@XParam("nombreParam") String nombreParam) {
    return <MENSAJE EN FORMATO TIPOS_MIME_RESPUESTA>;
}
```

La forma más general que hay de responder a una petición REST con JAX-RS es la de tener una anotación de método HTTP. A continuación, podemos tener una anotación con un `@Path` específico para este método, que puede contener (o no, es opcional) uno o varios parámetros de tipo `@PathParam`, rodeados por «{}». Pueden aparecer en cualquier posición dentro del path, como por ejemplo `{param1}/path/{param2}`. Cada parámetro de este tipo aparecerá como parámetro del método, y será del tipo `@PathParam`. También podemos tener parámetros del tipo `@QueryParam` (los enviados en la sección de consulta de la URL) o `@FormParam`, en caso de que el tipo MIME de solicitud sea `application/x-www-form-urlencoded` o `multipart/form-data`. Hay que decir que podríamos llegar a encontrar los tres tipos de parámetros en un mismo método.

Además de los parámetros antes mencionados, también podemos tener `@Consumes` y `@Produces` con tipos MIME que representan datos estructurados como `application/xml` o `application/jso`. Para tratar con datos estructurados, hay que tener clases Java de soporte que tengan los elementos correspondientes a los datos que se tienen que recibir. Hay diferentes alternativas para implementar estas clases de soporte de manera más o menos automática según las bibliotecas disponibles y las necesidades del servicio que se tiene que implementar (almacenar estos datos en ficheros, bases de datos, etc.). A continuación, presentaremos un ejemplo de implementación de un método que recibe y devuelve datos XML y otro que recibe y devuelve datos JSON. Los dos usan la misma clase Java de soporte, puesto que tienen la misma estructura.

Hemos denominado `Item` a esta clase de soporte. Contiene una serie de anotaciones Java para que se haga el tratamiento automático del lenguaje XML. Las anotaciones son `@XmlRootElement`, que identifica el elemento raíz del

documento, `@XmlAccessorType`, que permite indicar cómo se realiza la conexión entre los elementos de la clase Java y el XML `@XmlRootElement`, que identifica el resto de los elementos. Hay dos campos, uno llamado `id`, de tipo entero y otro llamado `name`, del tipo de cadena de caracteres. Antes del código de clase `Item`, vemos un archivo XML de ejemplo y otro archivo en formato JSON con datos y valores posibles.

```
<?xml version="1.0" encoding="UTF-8"?>
<item>
    <id>4</id>
    <name>Camera</name>
</item>
{
    "id": 4,
    "name": "Camera"
}
```

La clase `Item` contiene las anotaciones ya descritas y algunos métodos de soporte para modificar y consultar los campos de la clase.

```
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlElement;
@XmlRootElement(name="item")
@XmlAccessorType(XmlAccessType.FIELD)
public class Item {
    @XmlElement
    private int id;
    @XmlElement
    private String name;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    @Override
    public String toString(){
        return "Id: " + this.id + " Nombre: " + this.name;
    }
```

```
}
```

A continuación, ponemos dos ejemplos de implementación con el método HTTP PUT donde recibimos y enviamos datos tanto en formato XML como JSON.

En el primer método, el que trabaja con XML, hay que destacar que ahora recibimos un parámetro de tipo `Item` en el método, que convierte automáticamente los datos XML en campos de la clase Java. En este caso, el resultado en XML se genera dentro del mismo método, pero aquí podríamos añadir todo tipo de tratamiento de datos XML, almacenamiento en otros soportes (fichero a disco, base de datos, reenvío a otros servicios, etc.).

```
@PUT
@Path("/xml")
@Consumes("application/xml")
@Produces("application/xml")
public String putXML(Item i) {
    String resultado;
    resultado = "<?xml version=\"1.0\" encoding=\"UTF-8\"?><item><id>"
               +(i.getId()+1)+"</id><name> "
               + i.getName() +" Perez</name></item>";
    return resultado;
}
```

En el segundo método, el que trabaja con JSON, la estrategia es un poco diferente. Como parámetro recibimos un `java.io.InputStream`, que contiene un canal de acceso a los datos JSON enviados por la aplicación cliente. Con este `Inputstream`, utilizando una clase Java de soporte a JSON (`com.google.Gson.Gson`)[13], conseguimos un objeto de tipo `Item`, que después modificamos para enviar la respuesta al cliente. Lo mismo que se ha comentado para el caso XML (bases de datos, disco, etc.) se podría hacer aquí, realizando el tratamiento correspondiente de los datos JSON. Hay muchas librerías Java de apoyo a la creación, la lectura y la escritura de datos JSON. Se ha elegido esta por su simplicidad de uso.

(13) **Biblioteca Java de Gson**, <https://github.com/google/gson> [Fecha de consulta: marzo de 2020]

```
@PUT
@Path("/json")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public String putJSON(java.io.InputStream is) {
    Item test;
    com.google.gson.Gson gson = new com.google.gson.Gson();
    java.io.BufferedReader reader = new java.io.BufferedReader
                                    (new java.io.InputStreamReader(is));
    test = gson.fromJson(reader, Item.class);
    test.setId (test.getId()+1);
```

```
    test.setName (test.getName() + " photo");
    return gson.toJson(test);
}
```

2.1.3. Envío de la petición

El envío de la petición se hará dentro de un mensaje HTTP. A continuación vemos un ejemplo de invocación a los métodos PUT implementados en el apartado anterior con XML y JSON, respectivamente. Después de la línea de petición del servicio encontramos las cabeceras HTTP que identifican el tipo de contenido que se envía, su longitud, el tipo de datos que aceptamos para la respuesta y el nombre del host al cual nos queremos conectar.

```
PUT /app/hola/xml HTTP/1.1
Accept: application/xml
Content-Type: application/xml
Content-Length: 95
Host: www.exemple.com
<?xml version="1.0" encoding="UTF-8"?>
<item>
    <id>4</id>
    <name>Camera</name>
</item>
```

El segundo mensaje de solicitud contiene los mismos datos, pero ahora para el formato JSON para la petición y la respuesta.

```
PUT /app/hola/json HTTP/1.1
Accept: application/json
Content-Type: application/json
Content-Length: 35
Host: www.exemple.com
{
    "id": 4,
    "name": "Camera"
}
```

2.1.4. Envío de la respuesta

El envío de la respuesta se hará también dentro de un mensaje HTTP. A continuación vemos un ejemplo de respuesta al método PUT con XML y JSON, respectivamente. Después de la línea de respuesta del servicio encontramos las cabeceras HTTP que nos identifican el tipo de contenido que se envía y su longitud.

```
HTTP/1.1 200 OK
Content-Length: 95
```

```
Content-Type: application/xml
<?xml version="1.0" encoding="UTF-8"?>
<item>
    <id>5</id>
    <name>Camera photo</name>
</item>
```

El segundo mensaje de solicitud contiene los mismos datos, pero ahora para el formato JSON para la petición y la respuesta.

```
HTTP/1.1 200 OK
Content-Length: 41
Content-Type: application/json
{
    "id": 5,
    "name": "Camera photo"
}
```

2.2. Operaciones propias del cliente

Invocar un servicio web basado en REST desde una aplicación cliente implica abrir una conexión HTTP con el servicio y enviar una petición en el formato esperado por el mismo (método HTTP, cabeceras y parámetros o datos). Esto lo podríamos hacer con una conexión vía *sockets* o usando clases de soporte disponibles en lenguajes de programación como por ejemplo Java

Referencia bibliográfica

March Hermo, M. I. (2020). *Programación de sockets en Java*. Barcelona: UOC.

La aplicación cliente puede ser de cualquier tipo: una aplicación de escritorio, una aplicación web e, incluso, una aplicación móvil. Los requisitos que tiene que cumplir es que sepa enviar el mensaje de petición HTTP con los datos requeridos y después sea capaz de entender la respuesta enviada por el servidor. La complejidad de esta aplicación cliente dependerá sobre todo del formato en que se envíen los datos en estas peticiones y respuestas. Así, cuanto más complejo sea el formato de los datos (por ejemplo, envío de ficheros en formatos XML o JSON), más complicado será tratarlos. Aun así , la existencia de librerías tanto para hacer las conexiones como para tratar los datos facilita mucho el trabajo de implementación.

En el resto del apartado se explicará con un algo más de detalle cómo implementar dos aplicaciones cliente, una aplicación Java de escritorio y una aplicación web basada en J2EE.

Referencia bibliográfica

Java Community Process (2006). JSR 88: Java™ EE (J2EE) Application Deployment, <https://www.jcp.org/en/jsr/detail?id=88&showPrint>

2.2.1. Programación de una aplicación cliente de servicio REST

En este apartado se explicará cómo implementar un cliente de prueba para nuestro servicio web basado en REST, pero la estrategia sería muy similar para conectarse a otro servicio REST ya existente. Primero veremos cómo hacerlo desde una aplicación web con un servlet (`javax.servlet.http.HttpServlet`) y después con una aplicación Java de escritorio.

Referencia bibliográfica

Oracle Corporation (2017). Java Servlets Specification v4.0, <https://javaee.github.io/servlet-spec/downloads/servlet-4.0/servlet-4_0_FINAL.pdf>

Aplicación web basada en J2EE

Un servlet es una clase Java que puede responder a métodos HTTP hechos por un navegador. En este sentido, es un poco como un servicio web REST, pero limitado a GET y a POST y orientado a un usuario final. Ayuda en la recepción de datos de formularios HTML y ficheros. También tiene acceso a la petición HTTP que se recibe desde el cliente (en este caso, un navegador) y a la respuesta HTTP que se tiene que devolver, que tiene que ser en lenguaje HTML para que el navegador lo entienda.

La declaración de una clase de tipo servlet con apoyo para servicios web REST podría ser como sigue:

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.core.MediaType;
@WebServlet(name = "Client", urlPatterns = {"/Client"})
public class Client extends HttpServlet {
    /* Métodos del servlet para contactar con el servicio */
}
```

En el código que vemos hay importación de todas las clases Java necesarias y la declaración de la clase, que deriva de `HttpServlet`. Encontramos también una anotación Java que nos indica que esto es un servlet (`@WebServlet`) y la URL de acceso a este desde la aplicación web.

```
/* Extracto del código de connexión con el servidor */
Cliente cliente;
String REST_URL = "http://www.exemple.org/app/hola";
String resultado;
try {
```

```
    client   = ClientBuilder.newClient();
    resultado = client.target(REST_URL)
               .path("json")
               .request(MediaType.APPLICATION_JSON)
               .put(Entity.json("{\"id\":4,\"name\":\"camera\"}"),
                               String.class);
}
```

Aplicación de escritorio Java

El código de la aplicación de escritorio para conectar con un servicio web basado en REST es muy parecido al código que hemos implementado desde una aplicación web. La diferencia más importante es que tenemos que crear una clase Java con el método `public static void main (String [] args)` para ejecutarlo.

Hay que destacar que el único problema que podemos encontrar en este caso es el de disponer de todas las librerías Java que apoyen a una aplicación cliente de servicio web basado en REST, puesto que la librería que contiene las clases `javax.ws.rs.client` tiene bastantes dependencias con otras librerías Java que cualquier entorno de programación puede resolver sin demasiado problema. No ponemos ninguna versión concreta, puesto que el lenguaje Java evoluciona tan rápidamente que enseguida quedarían obsoletas.

```
/* Código para hacer la connexión desde una aplicación Java de escritorio */
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.core.MediaType;
public class ClientJavaREST {
    public static void main(String[] args) {
        Client client;
        String REST_SERVICE_URL = "http://www.exemple.org/app/hola";
        String resultado;
        try {
            client = ClientBuilder.newClient();
            resultado = client.target(REST_SERVICE_URL)
                 .path("json")
                 .request(MediaType.APPLICATION_JSON)
                 .put(Entity.json("{\"id\":3,\"name\":\"camera\"}"),
                      String.class);
            /* Mostramos el resultado. Hemos recibido datos
            en formato JSON, que podríamos tratar convenientemente  */
            System.out.println (resultado);
        } catch (Exception e)
        {
```

```
                System.out.println (e.getMessage());
            }
        }
    }
```

3. Servicios REST con otros lenguajes de programación (Node. js, Python)

3.1. Node. js

Node.js es un entorno de servidor de código abierto que funciona sobre diferentes sistemas operativos y que usa el lenguaje Javascript en el servidor (el uso habitual de Javascript se encuentra en el lado cliente, incluido en el navegador).

Sobre Node. js

OpenJS Foundation, Node. js, <https://nodejs.org/es/>. [Fecha de consulta: marzo de 2020].

Para poderlo usar como servidor REST, hay que utilizar lo que se conoce como el conjunto de herramientas MEAN, que incluye las herramientas MongoDB, ExpressJS, Angular y Node.JS (MEAN)[14]. Todas estas herramientas se basan en lenguaje Javascript. Como mínimo tenemos que tener Node.js y ExpressJs, pero para tener un servicio REST completo es recomendable tenerlas todas. A continuación las describimos brevemente.

(14)**IBM, MEAN (MongoDB ExpressJS Angular Node.js)** stack explained, <https://www.ibm.com/cloud/learn/mean-stack-explained> [Fecha de consulta: marzo de 2020]

MongoDB es una base de datos no relacional que permite almacenar datos en formato JSON.

ExpressJS permite que Node.js se pueda comportar como un servidor REST. Se trata de un entorno flexible y que proporciona toda una serie de características para dar soporte a aplicaciones web y móviles, incluyendo la creación de API.

Angular permite desarrollar aplicaciones web, móviles y de escritorio usando lenguaje Javascript.

Referencias bibliográficas

MongoDB Inc., **MongoDB**, <https://www.mongodb.com/es>. [Fecha de consulta: marzo de 2020].

StrongLoop, Inc., **ExpressJS**, <https://expressjs.com/es/>. [Fecha de consulta: marzo de 2020].

Google, **Angular**, <https://angular.io/>. [Fecha de consulta: marzo de 2020].

En el subapartado siguiente explicamos con algo más de detalle cómo implementar un servicio web básico basado en REST con Node.js.

3.1.1. Desarrollo de una API REST con Node. JS y ExpressJS

Para desarrollar un servicio REST con Node.js hay que instalarlo desde su web oficial. Una vez instalado, hay que instalar también ExpressJS para poder ofrecer las diferentes operaciones vía web.

Una vez estas dos herramientas están instaladas, implementar una API básica es bastante sencillo.

Lo primero que hay que hacer es declarar las dependencias que tendrá nuestra API. Esto se puede hacer con el código Javascript siguiente:

```
/* Declaración de dependencias y variable app */
```

```
var express = require('express');
var bodyParser = require('body-parser');
var app = express();
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
```

Aquí se indica que hay dos dependencias, `express` y `body-parser`, que usaremos después. A continuación declaramos la variable `app` que será la aplicación web que usará Express.

En el trozo de código siguiente vemos que la aplicación usa algunas características de `body-parser`. Además, declara la variable `puerto` que define el puerto donde escuchará este servidor. Si no se ha definido otro en los ficheros de configuración, el puerto que se usará es el 8080. A continuación, se declara una constante que contendrá el router que proporciona ExpressJS. Con este router se definen las diferentes operaciones que proporciona el servicio.

La primera de todas es responder a GET en la URL base del servicio y devolver los datos `{"mensaje":"API responde a GET!"}`. La siguiente es responder a POST en la URL base del servicio y devolver los datos `{"mensaje":"API responde a POST!"}`.

```
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
var port = process.env.PORT || 8080;
const router = express.Router();
router.get('/', function(req, res) {
   res.json({ message: 'API responde a GET!' });
});
router.post('/', function(req, res) {
   res.json({ message: 'API responde a POST!' });
   res.end("end");
});
```

A continuación vemos una operación que ofrece esta API. Dada una letra, devuelve como resultado la misma letra en mayúsculas. Si ya es mayúscula, devuelve la misma letra que ha escrito el usuario. La URL de acceso (que se tiene que concatenar a la URL del servicio) tiene la forma `/letras/a` y el resultado que devuelve es `{"resultado":"A"}`.

Finalmente, se pone en marcha la aplicación web en la URL `/api` que escuchará por el puerto indicado anteriormente. Así, para acceder a la funcionalidad de la operación letras con esta aplicación tendríamos que usar la URL siguiente: `http://exemple.org/api/letras/a`. La base de URL del servicio sería `http://exemple.org/api` y daría una respuesta diferente a los métodos GET y POST, tal como hemos visto anteriormente.

```
router.route('/letras/:letra').get((req, res) => {
   res.json({resultado: req.params.letra.toUpperCase()})
});
app.use('/api', router);
app.listen(port);
console.log('Escuchando al puerto ' + puerto);
```

El servicio se puede poner en marcha desde una línea de órdenes, indicando el nombre de la orden `node` y, a continuación, el nombre del fichero con extensión `.js` donde tengamos nuestro código. Podemos ver un ejemplo justo debajo.

```
linea_ordenes> node nombre_fichero.js
```

Código de servicio REST completo con Node. js

```
/* Código para hacer una API REST básica con Node.js */
var express = require('express');
var bodyParser = require('body-parser');
var app = express();
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
var puerto = process.env.PORT || 8080;
const router = express.Router();
router.get('/', function(req, res) {
   res.json({ message: 'API responde a GET!' });
});
router.post('/', function(req, res) {
   res.json({ message: 'API responde a POST!' });
   res.end("end");
});
router.route('/letras/:letra').get((req, res) => {
   res.json({resultado: req.params.letra.toUpperCase()})
});
app.use('/api', router);
app.listen(puerto);
console.log('Escuchando en el puerto ' + puerto);
```

3.2. Python

Python es un lenguaje de programación interpretado (necesita un intérprete para poderse ejecutar). Permite implementar muchos tipos de aplicaciones, incluso servicios basados en REST. Para hacerlo, necesita un entorno web, como por ejemplo Flask o Django. En los ejemplos siguientes usaremos Flask para el soporte web.

Flask es un *framework* web escrito en lenguaje Python. Se le denomina microframework porque no necesita herramientas o librerías extra para su funcionamiento. Aun así, se pueden añadir extensiones fácilmente para proporcionar funcionalidad extra.

Django es otro *framework* web escrito en Python. Es gratuito y de código abierto, y sigue el modelo arquitectural Model-Template-View (MTV).

En el subapartado siguiente explicamos con algo más de detalle cómo implementar un servicio web básico basado en REST con Python.

Referencias bibliográficas

Python Software Foundation, Python. [en línea]: <https://www.python.org/> [Fecha de consulta: marzo de 2020]

Pallets. *Flask*. [en línea]: <https://flask.palletsprojects.com/en/1.1.x/> [Fecha de consulta: marzo de 2020]

Django Software Foundation. *Django*. [en línea]: <https://www.djangoproject.com/> [Fecha de consulta: marzo de 2020]

Django Software Foundation. *Model-Template-View* (MTV) *architectural model in Django*. [en línea]: <https://docs.djangoproject.com/en/3.0/ faq/general/> [Fecha de consulta: marzo de 2020]

3.2.1. Desarrollo de una API REST con Python y Flask

Para desarrollar un servicio REST con Python hay que instalarlo desde su web oficial. Una vez instalado, hay que instalar también Flask para poder ofrecer las diferentes operaciones vía web.

Una vez estas dos herramientas están instaladas, implementar una API básica es bastante sencillo.

Lo primero que hay que hacer es declarar las dependencias que tendrá nuestra API. Esto se puede hacer con el código Python siguiente:

```
from flask import Flask, jsonify, request
app = Flask(__name__)
cuentas = [
        {'nombre': "Anna", 'balance':450.0},
        {'nombre': "Pau", 'balance':250.0},
    ]
```

Las dependencias que tenemos son `Flask`, `jsonify` y `request`. Todas estas dependencias provienen de `Flask`. Esto se indica en `from Flask`. Después declaramos la variable `app`, que contendrá nuestra aplicación web. Por último, declaramos la variable `cuentas`, que contendrá los datos de una cuenta bancaria en formato JSON.

Después, declaramos los métodos de nuestra API. Primero implementaremos las funciones que responden a GET. En este caso tenemos dos, una que nos devolverá todas las cuentas que tenemos en nuestra aplicación y otra que nos permitirá saber los datos de una cuenta a partir de su identificador.

Para definir operaciones, usamos la variable `app` declarada anteriormente y llamamos al método `route`, donde definimos la URL de acceso a la operación y el método HTTP al cual responderá. En el primer caso, la URL es `/cuentas` y en el segundo caso, la URL es `/cuentas/<id>` donde `id` es un parámetro de la URL que indica el identificador de la cuenta que se quiere consultar. Después hay que implementar el código de la operación, que devuelve los datos pedidos por el usuario.

```
@app.route("/cuentas", methods=["GET"])
def getCuentas():
    return jsonify(cuentas)
@app.route("/cuentas/<id>", methods=["GET"])
def getCuenta(id):
    id = int(id) - 1
    return jsonify(cuentas[id])
```

La última operación responde al método POST y nos permite dar de alta una nueva cuenta. La URL de acceso será `/cuenta` y los datos se moverán en formato JSON al cuerpo del mensaje HTTP. Por último, la aplicación web se inicia en el puerto 8080.

```
@app.route("/cuenta", methods=["POST"])
def addCuenta():
    nombre = request.json['nombre']
    balance = request.json['balance']
    datos = {'nombre': nombre, 'balance': balance}
    cuentas.append(datos)
    return jsonify(datos)
if __name__=='__main__':
    app.run(port=8080)
```

El servicio se puede poner en marcha desde una línea de órdenes, indicando el nombre de la orden `python3` (correspondiente a la versión 3 de Python) y luego el nombre de archivo con la extensión `.py` donde tenemos nuestro código. Podemos ver un ejemplo justo debajo.

```
linea_ordenes> python3 nombre_fichero.py
```

Código de servicio completo de Python

```
/* Código para hacer una API REST básica con Python */
from flask import Flask, jsonify, request
```

```
app = Flask(__name__)
cuentas = [
        {'nombre': "Anna", 'balance':450.0},
        {'nombre': "Pau", 'balance':250.0},
    ]
@app.route("/cuentas", methods=["GET"])
def getCuentas():
    return jsonify(cuentas)
@app.route("/cuentas/<id>", methods=["GET"])
def getCuenta(id):
    id = int(id) - 1
    return jsonify(cuentas[id])
@app.route("/cuenta", methods=["POST"])
def addCuenta():
    nombre = request.json['nombre']
    balance = request.json['balance']
    datos = {'nombre': nombre, 'balance': balance}
    cuentas.append(datos)
    return jsonify(datos)
if __name__=='__main__':
    app.run(port=8080)
```

Resumen

El módulo ha presentado los servicios web basados en REST (REpresentational State Transfer), que definen una arquitectura de cómo se tiene que acceder y manipular recursos remotos mediante los métodos HTTP (POST, GET, PUT, PATCH y DELETE).

Hemos visto también la comparación entre los servicios web basados en SOAP y REST, incidiendo en las características de cada uno y explicando algunas de las diferencias principales entre ellos.

Finalmente, se ha hablado de cómo desarrollar servicios web en lenguaje Java, dando también algunas pautas de cómo se podría hacer con otros lenguajes de programación como son Node.js y Python.

Bibliografía

Django Software Foundation. *Django*. [en línea]:<https://www.djangoproject.com/> [Fecha de consulta: marzo de 2020].

Django Software Foundation. *Modelo-Plantilla-Vista (MTV) modelo arquitectónico en Django*. [en línea]:<https://docs.djangoproject.com/en/3.0/faq/general/> [Fecha de consulta: marzo de 2020].

European Computer Manufacturers Association (ECMA) (2017). The JSON (JavaScript Object Notation) Data Interchange Syntax ECMA – 404 Standard. [en línea]:<http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>.

Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. [en línea]:<https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf>.

Google. *Angular*. [en línea]:<https://angular.io/> [Fecha de consulta: marzo de 2020].

Gson Java library. [en línea]:<https://github.com/google/gson> [Fecha de consulta: marzo de 2020].

IBM. *MEAN (MongoDB Express Angular Node) stack explained*. [en línea]:<https://www.ibm.com/cloud/learn/mean-stack-explained> [Fecha de consulta: marzo de 2020].

Internet Engineering Task Force (IETF) (2005). Uniform Resource Identifier (URI): Generic Syntax, RFC 3986. [en línea]:<https://tools.ietf.org/html/rfc3986>.

Internet Engineering Task Force (IETF) (2014). Hypertext Transfer Protocol (HTTP/1.1), RFC 7230 a 7235. [en línea]:<https://tools.ietf.org/html/rfc7230> y <https://tools.ietf.org/html/rfc7235>.

Internet Engineering Task Force (IETF) (2015). Returning Values from Forms: multipart/form-data, RFC 7578. [en línea]:<https://tools.ietf.org/html/rfc7578>.

Internet Engineering Task Force (IETF) (2018). The Transport Layer Security (TLS) Protocol Versió 1.3. [en línea]:<https://tools.ietf.org/html/rfc8446>

Introducing JSON. [en línea]:<https://www.json.org/json-en.html> [Fecha de consulta: marzo de 2020].

Java Community Process (2006). JSR 88: Java™ EE (J2EE) Application Deployment. [en línea]:<https://www.jcp.org/en/jsr/detail?id=88&showPrint>.

March Hermo, M. I. (2020). *Programación de Sockets en Java*. Barcelona: UOC.

MongoDB Inc. *Mongodb*. [en línea]:<https://www.mongodb.com/es> [Fecha de consulta: marzo de 2020].

OpenJS Foundation *Node.js* [en línea]:<https://nodejs.org/es/> [Fecha de consulta: marzo de 2020].

Oracle Corporation (2013). Java Enterprise Edition 7 Specification APIs. [en línea]:<https://docs.oracle.com/javaee/7/api/overview-summary.html>

Oracle Corporation (2014). *The Java Tutorial, Annotations*. [en línea]:<https://docs.oracle.com/javase/tutorial/java/annotations/index.html>.

Oracle Corporation (2017). Java Servlets Specification v4.0. [en línea]:<https://javaee.github.io/servlet-spec/downloads/servlet-4.0/servlet-4_0_FINAL.pdf>.

Organization for the Advancement of Structured Information Standards (OASIS) (2006). Web Services Security: SOAP Message Security 1.1 (WS-Security). [en línea]:<https://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf>.

Pallets *Flask*. [en línea]:<https://flask.palletsprojects.com/en/1.1.x/> [Fecha de consulta: marzo de 2020].

Python Software Foundation *Python*. [en línea]:<https://www.python.org/> [Fecha de consulta: marzo de 2020].

StrongLoop, Inc. *ExpressJS*. [en línea]:<https://expressjs.com/es/> [Fecha de consulta: marzo de 2020].

Sun Microsystems (2008). JAX-RS: Java™ API for RESTful Web Services. [en línea]:<https://download.oracle.com/otn-pub/jcp/jaxrs-1.0-fr-eval-oth-JSpec/jaxrs-1.0-final-spec.pdf>.

World Wide Web Consortium (W3C) [en línea]:<https://www.w3.org/> [Fecha de consulta: marzo de 2020].

World Wide Web Consortium (W3C) (2006). Extensible Markup Language (XML) 1.1 (Second Edition). [en línea]:<https://www.w3.org/TR/xml11>.

World Wide Web Consortium (W3C) (2007). Simple Object Access Protocol (SOAP) Versión 1.2. [en línea]:<https://www.w3.org/TR/soap12/>.

World Wide Web Consortium (W3C) (2007). SOAP Version 1.2 Part 2: Adjuncts (Second Edition). [en línea]:<https://www.w3.org/TR/soap12-part2/>.

World Wide Web Consortium (W3C) (2007). Web Services Description Language (WSDL) Versión 2.0. [en línea]:<https://www.w3.org/TR/wsdl20/>.

World Wide Web Consortium (W3C) (2012). XML Schema Definition Language (XSD) 1.1. [en línea]:<https://www.w3.org/TR/xmlschema11-1/>.