# Servicios Web[1]

Los Servicios Web son sistemas software diseñados para proporcionar interacciones ordenador-ordenador utilizando la red, es decir, permiten la interacción directa entre ordenadores utilizando los protocolos de internet. Normalmente están basados en una arquitectura cliente-servidor (petición-respuesta) implementada sobre el protocolo HTTP (Hyper Text Transfer Protocol).
Según describe el *World Wide Web Consortium* (W3C), los servicios web proporcionan un medio estándar de interoperabilidad entre aplicaciones software que se ejecutan en una gran variedad de plataformas y frameworks. Los servicios web se pueden combinar entre sí de forma débilmente acoplada, para lograr operaciones mucho más complejas. De este modo, programas que prestan servicios simples pueden interactuar entre sí para ofrecer servicios sofisticados de valor añadido. Por tanto, su principal característica es su gran interoperabilidad y extensibilidad, así como proporcionar información fácilmente procesable por las máquinas gracias al uso de XML.

### 1. Tipos de Servicios Web

A nivel conceptual, un servicio web es un componente software provisto de un punto de acceso en la red. El consumidor y el proveedor del servicio emplean mensajes para intercambiar información de peticiones y respuestas en la forma de documentos autocontenidos, que incluyen todo lo necesario para prestar su servicio.

A nivel tecnológico, los servicios web se pueden implementar de diferentes modos. Los tres estilos de uso más comunes de Servicios Web son:

- **Servicios Web basados en *Remote Procedure Calls*** (RPC, Llamadas a Procedimientos Remotos). Presentan una interfaz de llamada a procedimientos y funciones distribuidas. Las primeras herramientas para Servicios Web se basaban en este estilo y muchos lo consideran como "la primera generación de Servicios Web". Su principal problema es que se implementan mediante el mapeo de

[1] La documentación de todo este apartado se basa en las siguientes referencias:

- Rafael Navarro Marset. REST vs Web Services. Modelado, Diseño e Implementación de Servicios Web 2006-07. ELP-DSIC-UPV.
http://users.dsic.upv.es/~rnavarro/NewWeb/docs/RestVsWebServices.pdf

- The Java EE 6 Tutorial. Chapter 18 - Introduction to Web Services. Oracle. January 2013. https://docs.oracle.com/javaee/6/tutorial/doc/javaeetutorial6.pdf

- Mª Isabel Alfonso Galipienso. Servicios Rest
expertojava.ua.es/experto/restringido/2015-16/rest/rest01.html

- Koushik Kothagal. Developing RESTful APIS with JAX-RS. *javabrains.koushik.com*
https://www.youtube.com/playlist?list=PLqq6Pq4lTTZh5U8RbdXq0WaYvZBz2rbn

servicios directamente a funciones específicas del lenguaje o llamadas a métodos, por lo que no son débilmente acoplados.

- **Servicios Web SOAP**. Estos servicios web, también conocidos como servicios web "grandes" ("big" Web Services), o simplemente "servicios web", utilizan mensajes XML que siguen el protocolo estándar SOAP (Simple Object Access Protocol). Dicho estándar es un lenguaje XML que define una arquitectura de mensaje y unos formatos de mensaje. Estos sistemas normalmente contienen una descripción de las operaciones ofrecidas por el servicio escritas en WSDL (*Web Services Description Language*), que es un lenguaje XML para definir interfaces sintácticamente.
- **Servicios Web basado en REST** (*REpresentation State Transfer*), conocidos también como servicios "ligeros" o servicios Web RESTful. REST es un estilo de arquitectura software, introducido en la tesis doctoral de Roy Fielding en 2000, quien es uno de los principales autores de la especificación de HTTP. REST en sí no es un estándar, ya que es tan sólo un estilo de arquitectura, pero está basado en estándares: HTTP, URI, XML, JSON, etc. Resulta una alternativa más simple y flexible a los servicios web clásicos, lo que ha hecho que tengan gran aceptación en la actualidad, desbancando en gran medida a los servicios Web SOAP. Los servicios Web RESTful pueden intercambiar mensajes escritos en diferentes formatos y no requieren publicar una descripción de las operaciones que proporcionan.

**2. Características de los Servicios Web**

Podemos distinguir un conjunto de características que cumplen en general los Servicios Web:

- Los servicios intercambian su información a través de la Web, utilizando ***estándares Web***, tales como HTTP. Así, el cliente envía una solicitud HTTP al servidor (*HTTP request*) y el servidor le devuelve una respuesta HTTP (*HTTP Response*).
- Existe un ***protocolo*** que determina cómo es el formato de los mensajes a enviar entre clientes y servidor de un Servicio Web, para que así cliente y servidor puedan entenderse. En algunos tipos de servicios Web, como SOAP, se utiliza un protocolo estandarizado que especifica claramente cómo debe estructurarse el contenido del mensaje. En otros tipos de servicios, como REST, no existe protocolo, de modo que los clientes REST pueden enviar mensajes utilizando XML, JSON o bien simplemente Texto, y siempre y cuando el servidor pueda entender ese tipo de mensaje, el servicio funcionará. JSON (JavaScript Object Notation) es el formato más habitual para el intercambio de información.
- La ***definición del servicio*** implica describir qué hace el servicio. En los servicios Web SOAP la definición del servicio se representa a través del estándar WSDL, que es un documento que define qué hace el servicio, incluyendo cuáles son los métodos, los parámetros, etc. Este documento se facilita a los clientes para que éstos conozcan cómo deben llamar a los métodos disponibles del servicio web. En el caso de los servicios Web REST, no hay ningún estándar para la definición del servicio. En la mayoría de servicios Web REST no existe casi documentación sobre la definición del servicio y la que hay está orientada a ser leída por los desarrolladores (y no por los programas). Por tanto, no se requiere ningún lenguaje de descripción de funcionalidad. No obstante, empiezan a emplearse este tipo de lenguajes, siendo OAS (OpenAPI Specification) el más utilizado actualmente.

- El servicio web puede seguir una ***especificación*** o bien un ***estilo arquitectónico***. Seguir una especificación, como es el caso de los Servicios Web SOAP, implica que existe un conjunto definido de reglas que indican claramente cómo debe ser un servicio web. Por ejemplo, si un "servicio web SOAP" no sigue todas las reglas de la especificación SOAP, entonces no puede ser considerado como un servicio web SOAP.
  Sin embargo, un estilo arquitectónico define unas pautas o guías de diseño a seguir, pero puede ser implementado de forma total o parcial. Por ejemplo, basándonos en el estilo arquitectónico REST, podemos definir servicios Web Completely RESTful (siguen completamente el estilo REST), o bien servicios "Not fully RESTful" (siguen parcialmente el estilo REST). Por tanto, cuando se diseñan Servicios Web REST el objetivo debe ser realizarlos lo más "completamente REST" como sea posible.

## 3. Servicios Web RESTful

Los servicios Web RESTful constituyen una de las tecnologías más importantes para el desarrollo de aplicaciones web. Están disponibles en la inmensa mayoría de lenguajes de programación y frameworks de desarrollo. En la actualidad, existen APIs REST para el acceso a múltiples aplicaciones distribuidas, como por ejemplo Google Drive, Instagram, Gmail, etc. Además, en API Java para servicios Web RESTFul, llamado JAX-RS, permite desarrollar servicios Web RESTful de forma sencilla.

Como se ha explicado anteriormente, REST (*Representational State Transfer*) es un estilo arquitectónico que especifica ciertas restricciones que si se aplican a un servicio web inducen en él propiedades deseables, como escalabilidad, rendimiento, flexibilidad, que permite a los servicios funcionar mejor en la Web.

El estilo arquitectónico REST define los siguientes principios:

- Debe ser un sistema **cliente-servidor**
- **Datos y funcionalidad se consideran recursos,** por lo que se accede a ellos mediante URIs (*Uniform Resource Identifiers* (URIs)[2]. Por tanto, un servicio web RESTful exhibe un conjunto de recursos que representan el punto de interacción con los clientes.

[2] El formato de una URI se estandariza como sigue:
`scheme://host:port/path?queryString#fragment`
donde:
`scheme` es el protocolo para comunicarse con el servidor, que normalmente será http o https
host es un nombre DNS o una dirección IP. Opcionalmente se puede indicar el puerto `(:port)`
`path` es un conjunto de segementos de texto delimitados por `/` (en REST representará el recurso)
Opcionalmente podemos tener una lista de parametros `(?queryString)`, representada como pares nombre/valor y delimitados por carácter `&`.
`#fragment`, también opcional, "apunta" a cierto "lugar" del documento al que estamos accediendo.

Por ejemplo:

`http://administracion.upv.es/alumnos` representa la colección de todos los alumnos de la universidad.

`http://administracion.upv.es/alumnos/123456789A` representa a un alumno con identificador 123456789A

- **Interfaz uniforme**. Los recursos se manipulan utilizando un conjunto de operaciones bien definidas (basadas en HTTP). En concreto, se suelen utilizar cuatro operaciones: GET, PUT, DELETE y POST.
  - **GET:** permite leer un recurso (recupera el estado actual de un recurso en alguna representación). Es una operación solo de lectura, así como idempotente y segura. *Idempotente* significa que no importa cuántas veces invoquemos esta operación, el resultado (que observaremos como usuarios) debe ser siempre el mismo. *Segura* significa que una operación GET no cambia el estado del servidor en modo alguno, es decir, no debe exhibir ningún efecto lateral en el servidor. Por ejemplo, el hecho de "leer" un documento HTML no debería cambiar el estado de dicho documento.

  - **PUT:** solicita al servidor el almacenar el cuerpo del mensaje enviado con dicha operación en la dirección proporcionada en el mensaje HTTP. Normalmente se modela como una modificación de un recurso o bien una inserción idempotente. Enviar el mismo mensaje PUT más de una vez no tiene ningún efecto sobre el servicio subyacente. Por ejemplo, si tenemos el recurso /pedidos que representa una lista de pedidos, y queremos añadir el pedido con identificador 215, podemos utilizar PUT /pedidos/215. Además, varias llamadas consecutivas con los mismos datos (PUT /pedidos/215) tendrán el mismo efecto que realizar sólo una de ellas.

  - **DELETE:** permite eliminar un recurso. También es idempotente.

  - **POST:** es la única operación HTTP que no es idempotente ni segura. Se utiliza para crear un recurso. Por ejemplo, para el recurso /pedidos que representa una lista de pedidos, podemos añadir un nuevo pedido con POST /pedidos, pues la URI del nuevo pedido aún no existe. La operación NO es idempotente, es decir, si añadimos varias veces el mismo pedido aparecerá repetido en nuestra lista de pedidos con URIs distintas.

Como respuesta a estas operaciones, también se utilizan códigos de respuesta HTTP para indicar el estado de finalización de la operación.

| Código HTTP | Significado habitual |
|---|---|
| 200 | Todo correcto |
| 201 | Recurso creado |
| 400 | Solicitud incorrecta |
| 404 | Recurso no encontrado |
| 500 | Fallo en el proveedor del servicio |

- Los **recursos se desacoplan de su representación**, de modo que su contenido pueda ser accedido en una gran variedad de formatos, como HTML, XML, texto plano, PDF, JPEG, JSON u otros. Por tanto, un recurso referenciado por una URI puede tener diferentes formatos (representaciones). Se disponen de metadatos sobre el recurso y pueden utilizarse para, por ejemplo, controlar el almacenamiento en caché, detectar errores de transmisión, negociar el formato de representación adecuado y llevar a cabo control de acceso o autenticación.
  En este ejemplo se muestra el contenido de un determinado recurso, tanto en formato XML como en formato JSON:

| XML | JSON |
|---|---|
| <**Person**><br><**ID**>123456789F</**ID**><br><**Name**>Agustín Espinosa</**Name**><br><br><**Email**>agessa@alumno.upv.es</**Email**><br></**Person**> | {<br>**"ID"**: "123456789F",<br>**"Name"**: "Agustín Espinosa",<br>**"Email"**: "agessa@alumno.upv.es"<br>} |

- Diseñado para utilizar un **protocolo de comunicación sin estado**, es decir, que no hay necesidad de que los servicios guarden las sesiones de los usuarios (cada petición al servicio tiene que ser independiente de las demás).
- En la respuesta se envía un código de estado HTTP, que indica cómo ha finalizado la llamada al servicio.
- Debe soportar un sistema de **cachés**: la infraestructura de la red debería soportar caché en diferentes niveles.
- Los mensajes son **auto-descriptivos**, de modo que cada mensaje debe incluir la suficiente información como para describir cómo procesar el mensaje. Por ejemplo, se puede indicar cómo "parsear" el mensaje indicando el tipo de contenido del mismo (xml, html, texto, ...).

Como hemos comentado, una URI en un servicio web RESTful es un hiper-enlace a un recurso y es la única forma de intercambiar representaciones entre clientes y servidores. Un ejemplo de URI podría ser este: `http://csdcourse.upv.es/resources/students` donde el host viene dado por csdcourse.upv.es, el path o ruta de acceso al recurso es `/resources/students`.

Si por ejemplo dicho recurso representa la información de los estudiantes de la asignatura de CSD, podríamos acceder a la lista correspondiente mediante:
`GET http://csdcourse.upv.es/resources/students`

Esto nos devolverá la lista de estudiantes en el formato que el desarrollador del servicio haya decidido. Por ejemplo, la petición anterior nos podría devolver un documento como el siguiente:

```
<?xml version="1.0"?>
<students>
      <student>http://csdcourse.upv.es/resources/students/1"<student/>
      <student>http://csdcourse.upv.es/resources/students/2"<student/>
      <student>http://csdcourse.upv.es/resources/students/3"<student/>
      <student>http://csdcourse.upv.es/resources/students/6"<student/>
</students>
```

En este documento se muestra la lista de estudiantes registrados en la aplicación, cada uno de ellos representado también por una URL. Accediendo a estas URLS, podremos obtener información sobre cada estudiante concreto o bien modificarlo.

## 4. Cómo crear una interfaz basada en REST

Para el diseño de una interfaz REST debemos tener en cuenta los siguientes aspectos:

- **Recursos**: los recursos son identificados mediante URIS. Por tanto, debemos identificar claramente los tipos de recursos que tendremos y, con ello, sabremos los tipos de URIs a emplear.

  A modo de ejemplo, supongamos un sistema de mantenimiento de una lista de contactos de empleados. En tal sistema cada usuario debería tener su propia URI con una apropiada representación. Además, la colección de recursos (empleados) es también otro recurso.
  Por tanto, hemos identificado dos tipos de recursos, por lo que habrá dos tipos de URIS:
  - `/empleados` (URI que representa al listado de empleados)
  - `/empleados/{id}` (una URI por cada empleado, identificado por su identificación en la empresa)

- **Formato**: el formato o representación del recurso puede ser un documento HTML, XML, una imagen, etc. Para cada uno de los recursos, se debe decidir cuál va a ser su representación. Dicha representación no tiene por qué ser en un único formato (podemos tener la información del recurso en varios formatos) y generalmente se utilizan estándares (como XML y JSON).

  Por ejemplo, el formato de representación de cada empleado podría ser el siguiente, en XML:

```
<employee xmlns='HTTP://example.org/my-example-ns/'>
 <name>  Full name goes here.</name>
 <title> Persons title goes here.</title>
 <phone-number> Phone number goes here.</phone-number>
</employee>
```

Y para el listado de empleados podríamos tener:

```
<employee-list xmlns='HTTP://example.org/my-example-ns/'>
 <employee-ref href="URI of the first employee"/>
    Full name of the first employee goes here.</employee>
  <employee-ref href="URI of employee #2"/>
   Full name</employee>
   .
   .
   <employee-ref href="URI of employee #N"/>
   Full name</employee>
 </employee-list>
```

- **Métodos soportados en cada URI:** debemos definir cómo van a ser referenciados los recursos. El acceso se puede hacer de muchas formas, recibiendo una representación del recurso (GET), añadiendo o modificando una representación (POST o PUT) y eliminando algunas o todas las representaciones (DELETE).

  Siguiendo con el ejemplo, tendríamos:
  - Recurso */empleados/{id}* : métodos GET (obtener la información del empleado); PUT (actualizar su información); DELETE (eliminar un empleado)
  - Recurso */empleados* : métodos GET (obtener el listado de los empleados); POST (añadir un nuevo empleado)

- **Códigos de estado que pueden ser devueltos**: En la respuesta se indica cómo ha finalizado la llamada al servicio utilizando códigos de estado HTTP. Por tanto, para cada método que ofrezcamos para el acceso a los recursos, debemos tener en cuenta los códigos de estado que podemos devolver.

  Los códigos HTTP más usuales son[3]:
  - `2xx: Peticiones correctas`
    - `200`: OK
    - `201`: Recurso creado
    - `204`: Sin contenido
  - `3xx: Redirecciones`. El cliente tiene que tomar una acción adicional para completar la petición.
    - `301`: Movido permanentemente. Estas y todas las peticiones futuras deberían ser dirigidas a la URI dada.
  - `4xx: Errores del cliente.` La solicitud contiene sintaxis incorrecta o no puede procesarse.
    - `400`: Solicitud incorrecta
    - `404`: Recurso no encontrado
    - `410`: Ya no disponible

[3] http://es.wikipedia.org/wiki/Anexo:Códigos_de_estado_HTTP

- `5xx: Errores de servidor`. El servidor falló al completar una solicitud aparentemente válida.
    - `500`: Error interno. Fallo en el proveedor del servicio.

**5. Cómo diseñar un servicio Web basado en REST**

Las pautas que seguir para el diseño de servicios Web basados en REST son las siguientes:

- Identificar todas las entidades conceptuales que se desean exponer como servicio, determinando cuáles son los **recursos**, es decir, los "objetos" de nuestro sistema, sin preocuparnos de las operaciones concretas a realizar sobre dichos objetos.
- Definir las **URIs** para los recursos. Para satisfacer el principio de "direccionabilidad" de los recursos, tendremos que definir las URIs que representarán los "puntos de entrada" de los mismos.
- Definir la **representación de los recursos**, definiendo el formato de los datos que utilizaremos para intercambiar información entre nuestros servicios y clientes.
- Definir los **métodos de acceso** a los recursos. Tendremos que decidir qué métodos HTTP nos permitirán acceder a las URIs que queremos exponer, así como qué hará cada método. Deberemos aquí ceñirnos a las restricciones de los principios RESTful.
    - Por ejemplo, deberemos categorizar los recursos considerando si los clientes pueden obtener una representación del recurso o si pueden modificarlo. Para lo primero, debemos hacer los recursos accesibles utilizando un HTTP GET. Para lo último, debemos hacer los recursos accesibles mediante HTTP POST, PUT y/o DELETE.
    - Todos los recursos accesibles mediante GET no deberían tener efectos secundarios. Es decir, los recursos deberían devolver la representación del recurso. Por tanto, invocar al recurso no debería ser el resultado de modificarlo.
    - Ninguna representación debería estar aislada. Es decir, es recomendable poner hipervínculos dentro de la representación de un recurso para permitir a los clientes obtener más información sobre ese recurso.