# Laboratorio 7 - Servicio Web REST o API REST

Actualmente se escucha bastante hablar de REST, de servicios REST, de aplicaciones REST, pues bien, REST es la abreviatura de REPRESENTATIONAL STATE TRANSFER.

## ¿Qué significa REST?

**REST es una interfaz para conectar varios sistemas basados en el protocolo HTTP** (uno de los protocolos más antiguos) y nos sirve para obtener y generar datos y operaciones, devolviendo esos datos en formatos muy específicos, como XML y JSON.

El formato más usado en la actualidad es el formato JSON, ya que es más ligero y legible en comparación al formato XML. Elegir uno será cuestión de la lógica y necesidades de cada proyecto.

REST deriva de "REpresentational State Transfer", que traducido vendría a ser “transferencia de representación de estado”, lo que tampoco aclara mucho, pero contiene la clave de lo que significa. Porque la clave de REST es que un servicio REST no tiene estado (es stateless), lo que quiere decir que, entre dos llamadas cualesquiera, el servicio pierde todos sus datos. Esto es, que no se puede llamar a un servicio REST y pasarle unos datos (p. ej. un usuario y una contraseña) y esperar que “nos recuerde” en la siguiente petición. De ahí el nombre: el estado lo mantiene el cliente y por lo tanto es el cliente quien debe pasar el estado en cada llamada. Si quiero que un servicio REST me recuerde, debo pasarle quien soy en cada llamada. Eso puede ser un usuario y una contraseña, un token o cualquier otro tipo de credenciales, pero debo pasarlas en cada llamada. Y lo mismo aplica para el resto de información.

El no tener estado es una desventaja clara: tener que pasar el estado en cada llamada es, como mínimo, tedioso, pero la contrapartida es clara: escalabilidad. Para mantener un estado se requiere algún sitio (generalmente memoria) donde guardar todos los estados de todos los clientes. A más clientes, más memoria, hasta que al final podemos llegar a no poder admitir más clientes, no por falta de CPU, sino de memoria. Es algo parecido a lo que ocurre con la web tradicional (que también es stateless). Sea cual sea la tecnología con la que desarrolles para web, seguro que conoces que puedes crear lo que se llama una “sesión”. Los datos de la sesión se mantienen entre peticiones y son por cada usuario. El clásico ejemplo de sesión puede ser un carrito de la compra, entre las distintas peticiones web, la información del carrito de compra se mantiene (¡ojo! hay otras alternativas para implementar carritos de compra).

Por norma general, la sesión se almacena en la memoria RAM del servidor, por lo que es fácil quedarnos sin memoria si introducimos demasiados datos en sesión (o tenemos muchos usuarios simultáneos). Por supuesto, podemos utilizar varios servidores, pero como bien saben los desarrolladores web, mover la sesión contenida en memoria entre servidores es generalmente imposible o altamente ineficiente, lo que penaliza el balanceo de carga, lo que a su vez redunda en menor escalabilidad y menor tolerancia a fallos. Hay soluciones intermedias, tales como guardar la

sesión en base de datos, pero su impacto en el rendimiento suele ser muy grande. De hecho, el consejo principal para desarrollar aplicaciones web altamente escalables es: no usar la sesión.

Así pues, tenemos que el ser stateless es la característica principal de REST, aunque por supuesto no la única. Así, cualquier servicio REST (si quiere ser merecedor de ese nombre) debería no tener estado, pero no cualquier servicio sin estado es REST. Hay otros factores, pero vamos a destacar el que los ingleses llaman "uniform interface" y es lo que diferencia un servicio web clásico (orientado a RPC) de un servicio REST.

## ¿Por qué debemos utilizar REST?

Una diferencia fundamental entre un servicio web clásico (SOAP) y un servicio REST es que el primero está orientado a RPC, es decir, a invocar métodos sobre un servicio remoto, mientras que el segundo está orientado a recursos. Es decir, operamos sobre recursos, no sobre servicios.

En una API REST la idea de "servicio" como tal desaparece. Lo que tenemos son recursos, accesibles por identificadores (URIs). Sobre esos recursos podemos realizar acciones, generalmente diferenciadas a través de verbos HTTP distintos.

**REST se apoya en HTTP**, los verbos que utiliza son exactamente los mismos, con ellos se puede hacer GET, POST, PUT y DELETE. De aquí surge una alternativa a SOAP.

- **Post**: Para crear recursos nuevos.
- **Get**: Para obtener un lichado o un recurso en concreto.
- **Put**: Para modificar.
- **Patch**: Para modificar un recurso que no es un recurso de un dato, por ejemplo.
- **Delete**: Para borrar un recurso, un dato por ejemplo de nuestra base de datos.

Cuando hablamos de SOAP hablamos de una arquitectura divididas por niveles que se utilizaba para hacer un servicio, es más complejo de montar como de gestionar y solo trabajaba con XML.

Ahora bien, **REST llega a solucionar esa complejidad que añadía SOAP, haciendo mucho más fácil el desarrollo de una API REST**, en este caso de un servicio en el cual nosotros vamos a almacenar nuestra lógica de negocio y vamos servir los datos con una serie de recursos URL y una serie de datos que nosotros los limitaremos, es decir, será nuestro BACKEND nuestra lógica pura de negocios que nosotros vamos a utilizar.

Todos **los objetos se manipulan mediante URI**, por ejemplo, si tenemos un recurso usuario y queremos acceder a un usuario en concreto nuestra URI seria /user/identificadordelobjeto, con eso ya tendríamos un servicio USER preparado para obtener la información de un usuario, dado un ID.

Así, en un servicio web clásico (SOAP) tendríamos un servicio llamado BeerService que tendría un método llamado GetAll que me devolvería todas las cervezas. La idea, independientemente de la tecnología usada para consumir el servicio web, es que se llama a un método (GetAll) de un servicio remoto (BeerService). Del mismo modo para obtener una cerveza en concreto

llamaríamos al método GetById() pasándole el id de la cerveza. De ahí que se diga que están orientados a RPC (Remote Procedure Call – Llamada a método remoto).

Por su parte en un servicio REST la propia idea de servicio se desvanece. En su lugar nos queda la idea de un "recurso", llamémosle "Colección de cervezas" que tiene una URI que lo identifica, p. ej. /Beers. Así, si invoco dicha URI debo obtener una representación de dicho recurso, es decir, debo obtener el listado de todas las cervezas.

Para obtener datos de una cerveza, habrá otro recurso (cerveza) con una URI asociada. Además, entre los recursos hay relaciones: está claro que una cerveza forma parte de la "Colección de cervezas" así que parece lógico que a partir de la colección entera (/Beers) pueda acceder a uno de sus elementos con una URI tipo /Beers/123, siendo "123" el ID de la cerveza.

## SOAP y REST

SOAP es el acrónimo de "Simple Object Access Protocol" y es el protocolo que se oculta tras la tecnología que comúnmente denominamos "Web Services" o servicios web. SOAP es un protocolo extraordinariamente complejo pensado para dar soluciones a casi cualquier necesidad en lo que a comunicaciones se refiere, incluyendo aspectos avanzados de seguridad, transaccionalidad, mensajería asegurada y demás. Cuando salió SOAP se vivió una época dorada de los servicios web. Aunque las primeras implementaciones eran lo que se llamaban WS1.0 y no soportaban casi ningún escenario avanzado, todo el mundo pagaba el precio de usar SOAP, ya que parecía claro que era el estándar que dominaría el futuro. Con el tiempo salieron las especificaciones WS-* que daban soluciones avanzadas, pero a la vez que crecían las capacidades de SOAP, crecía su complejidad. Al final, los servicios web SOAP terminan siendo un monstruo con muchas capacidades pero que en la mayoría de los casos no necesitamos.

Por su parte REST es simple. REST no quiere dar soluciones para todo y por lo tanto no pagamos con una demasiada complejidad una potencia que quizá no vamos a necesitar.

### Ventajas de REST

- Nos **permite separar el cliente del servidor**. Esto quiere decir que nuestro servidor se puede desarrollar en Node y Express, y nuestra API REST con Vue por ejemplo, no tiene por qué estar todos dentro de un mismo.
- En la actualidad tiene una **gran comunidad como proyecto** en Github.
- Podemos crear **un diseño de un microservicio orientado a un dominio** (DDD)
- Es totalmente independiente de la plataforma, así que **podemos hacer uso de REST tanto en Windows, Linux, Mac** o el sistema operativo que nosotros queramos.
- Podemos hacer nuestra **API pública**, permitiendo darnos visibilidad si la hacemos pública.
- Nos da **escalabilidad**, porque tenemos la separación de conceptos de CLIENTE y SERVIDOR, por tanto, podemos dedicarnos exclusivamente a la parte del servidor.

# RESTful API

## ¿Qué es un API?

Se conoce como API (del inglés: «Application Programming Interface»), o Interfaz de programación de Aplicaciones al conjunto de rutinas, funciones y procedimientos (métodos) que permite utilizar recursos de un software por otro, sirviendo como una capa de abstracción o intermediario.

Para quienes vienen iniciando en la programación y desarrollo de software, podemos describir un API, según David Berlind, en su artículo: What is an API, Exactly? como un método generalizado de consumir un servicio. Berlind nos presenta la analogía del conector eléctrico en las paredes de nuestro hogar u oficina. Este conector eléctrico que conocemos permite conectar equipos eléctricos que realizan distintas funciones. Este conector eléctrico es un API. El mismo permite consumir un servicio: electricidad sin importar que equipo lo consuma.

En programación y desarrollo de software o aplicaciones, esta analogía describe perfectamente un RESTful API. Creamos un API con el objetivo de que cualquier software pueda «consumir» nuestros recursos, por lo general datos, sin importar en que lenguaje o plataforma en que sea creado.

RESTful API es el sucesor de métodos anteriores como SOAP y WSDL cuya implementación y uso son un poco mas complejos y requieren mayores recursos y especificaciones al ser usados.

## ¿Cómo funciona un API?

Un RESTful API, como describimos hace un momento es un servicio. Y si seguimos la analogía del conector eléctrico, entonces fácilmente identificamos que funciona como un estándar para compartir información, en un sistema de doble vía:Consulta y Respuesta (Request -> Response).

Al consultar una API se deben especificar parámetros de consulta para que el servicio sepa lo que queremos consultar, basado en una estructura previamente definida provista por el API por medio de una documentación.

La arquitectura REST (del inglés: Representational State Transfer) trabaja sobre el protocolo HTTP. Por consiguiente, los procedimientos o métodos de comunicación son los mismos que HTTP, siendo los principales: GET, POST, PUT, DELETE. Otros métodos que se utilizan en RESTful API son OPTIONS y HEAD. Este último se emplea para pasar parámetros de validación, autorización y tipo de procesamiento, entre otras funciones.

Otro componente de un RESTful API es el «HTTP Status Code», que le informa al cliente o consumidor del API que debe hacer con la respuesta recibida. Estos son una referencia universal de resultado, es decir, al momento de diseñar un RESTful API toma en cuenta utilizar el «Status Code» de forma correcta.

Por ejemplo, el código 200 significa OK, que la consulta ha recibido respuesta desde el servidor o proveedor del API. Los código de estado más utilizados son:

- 200 OK
- 201 Created (Creado)
- 304 Not Modified (No modificado)
- 400 Bad Request (Error de consulta)
- 401 Unauthorized (No autorizado)
- 403 Forbidden (Prohibido)
- 404 Not Found (No encontrado)
- 422 (Unprocessable Entity (Entidad no procesable)
- 500 Internal Server Error (Error Interno de Servidor)

Una respuesta de un RESTFul API sería lo siguiente:

```
Status Code: 200 OK
Access-Control-Allow-Methods: PUT, GET, POST, DELETE, OPTIONS
Connection: Keep-Alive
Content-Length: 186
Content-Type: application/json
Date: Mon, 24 May 2016 15:15:24 GMT
Keep-Alive: timeout=5, max=100
Server: Apache/2.4.9 (Win64) PHP/5.5.12
X-Powered-By: PHP/5.5.12
access-control-allow-credentials: true
```

Por lo general y mejor práctica, el cuerpo (Body) de la respuesta de un API es una estructura en formato jSON. Aunque también puede ser una estructura XML o cualquier otra estructura de datos de intercambio, incluso una personalizada. Sin embargo, como el objetivo es permitir que cualquier cliente pueda consumir el servicio de un API, lo ideal es mantener una estructura estándar, por lo que jSON es la mejor opción.

El cuerpo de una respuesta estructurada en jSON se vería así:

```
{
  "error": false,
  "message": "Autos cargados: 2",
  "autos": [
    {
      "make": "Toyota",
      "model": "Corolla",
      "year": "2006",
      "MSRP": "18,000"
    },
    {
      "make": "Nissan",
      "model": "Sentra",
      "year": "2010",
```

```
    "MSRP": "22,000"
   }
  ]
}
```

La respuesta anterior se obtiene al consultar un API por medio de una ruta o método especifico vía URL. Por ejemplo: http://localhost/api/v1/autosutilizando el método HTTP GET.

Una muestra real de un RESTful API gracias a Facebook:

```
https://graph.facebook.com/?ids=http://www.nasa.gov
```

Puedes conocer los shares y comentarios de una URL de un website cualquiera con consultar el API de Facebook desde el navegador.

Y así tenemos una estructura que puede ser «comsumida» desde cualquier plataforma o software sin importar el lenguaje que se utilice, debido a la simple, organizada y estandarizada forma de presentar los datos.

De esta forma funcionan casi todas las aplicaciones en tu smartphone, o servicio como Facebook, Instagram, Snapchat, Messenger, etc. o cualquier otro servicio que permite a terceros utilizar sus recursos.

## Desarrollo del Laboratorio

### 1. PASO 1 (20 puntos)

Crear un Proyecto ASP .NET Web Application (laboratorio 3 y 4), además agregar utilizando la opción WebAPI como se muestra en la imagen 1 y al igual que se

realizó en el laboratorio 4 (punto 2,) se debe Mapear la siguiente

base de datos relacional imagen 2.

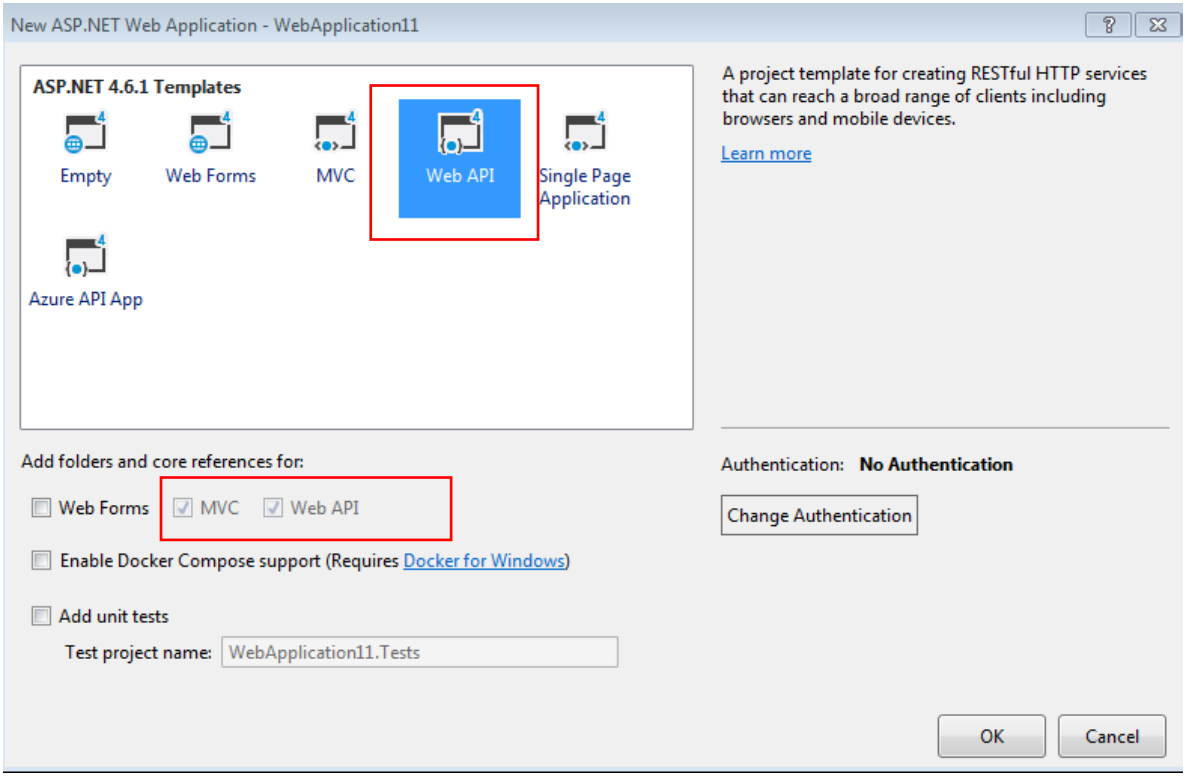


*Imagen 1*

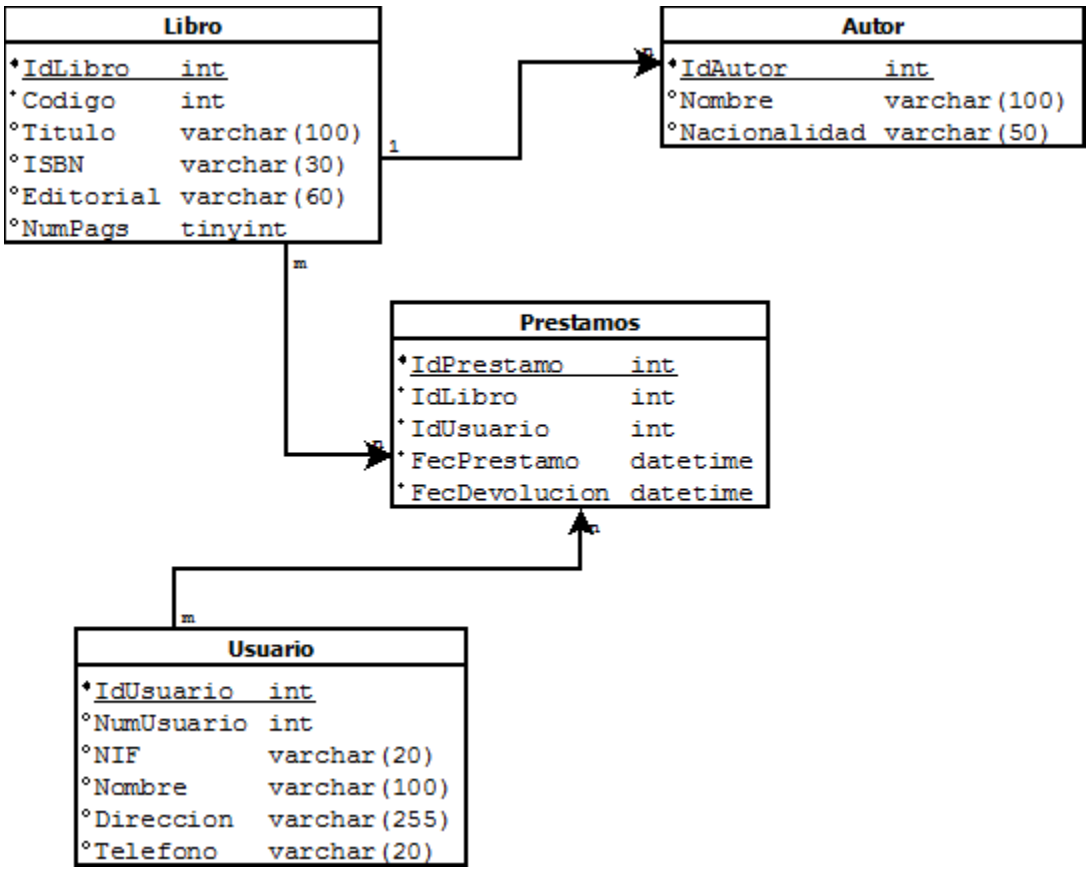


*Imagen 2*

## 2. PASO 2 (20 puntos)

Implementar los siguientes métodos dentro del controlador Rest ValuesController del API (Los métodos siguientes son los mismo utilizados en el laboratorio 5)

- ObtenerTodoslosLibros
- ObtenrUnLibroPorCodigo
- CrearLibro
- ActualizarLibro

**3. PASO 3 (20 punto)**

Crear una aplicación Web, utilizando solo HTML, JS y CSS. Que consuma los servicios creados en el paso 2.  Estos deben llamados con el método Ajax de Jquery (jQuery.ajax())

Nota: Para probar los servicios creados se puede además utilizar el cliente REST “Postman”