CIFP A Carballeira-Marcos Valcárcel
Rúa A Carballeira s/n
Ourense CP 32002
Tfno. 988788470, Fax 988227935

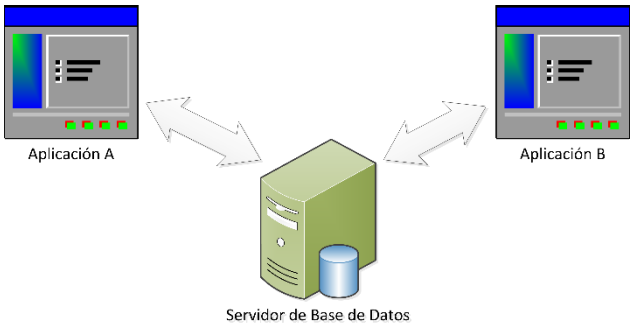
UD7 Servicios web.

7.1 Introducción a los servicios web.

Desde hace mucho tiempo, a la hora de desarrollar o implementar un sistema, ha surgido la necesidad de compartir información entre diferentes aplicaciones del sistema.

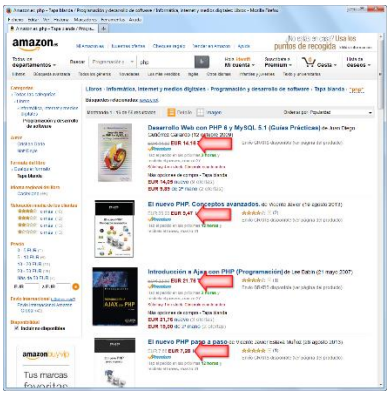
Como es habitual, varios mecanismos han sido diseñados para solucionar este problema, y la tecnología ha evolucionado con el tiempo. Algunas de las soluciones adoptadas fueron las siguientes:

- **Compartir el acceso a la base de datos**: es la técnica más sencilla, ya que se dispone de una misma base de datos con la que pueden trabajar diferentes aplicaciones.



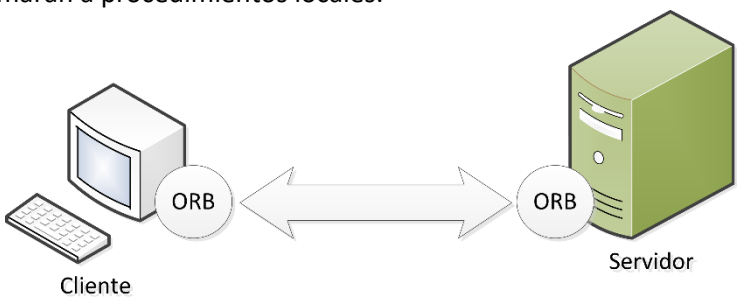
  Esta aproximación, a pesar de ser simple, no se recomienda debido a diversos problemas:
  - A mayor número de aplicaciones trabajando con el mismo conjunto de datos, más probable es que se generen errores de consistencia.
  - Para poner la información a disposición de terceros, sería necesario que estos tuvieran acceso directo a nuestra base de datos, y conocer la estructura o esquema de la misma. Esto plantea problemas graves de seguridad.

- **Acceder a la información publicada por la aplicación**: la idea consiste en acceder a la información de una aplicación directamente de la vista que esta proporciona. Si se trata de una aplicación web, por ejemplo, para obtener esta información habría que programar un procedimiento para buscar dicha información dentro das etiquetas HTML de la página.



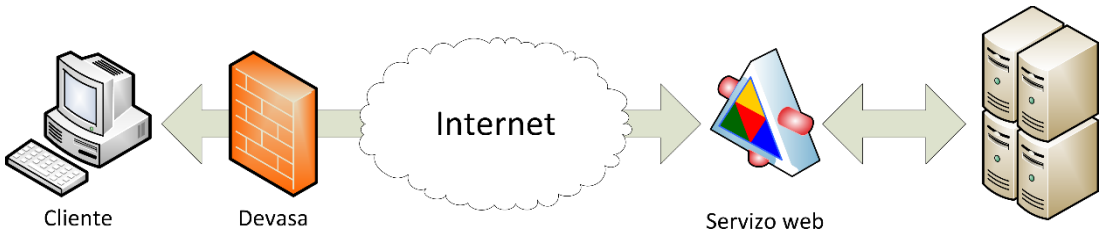
  Esta técnica se conoce como *screen scraper*. Es un procedimiento costoso y requiere rehacer el código cuando se producen cambios tanto en el contenido como en la presentación de la página que contiene la información.

- **Remote Procedure Calls (RPC)**: los primeros mecanismos para compartir información de un modo automatizado y estandarizado utilizaban un protocolo de comunicación llamado RPC. Dicho protocolo permitía realizar llamadas a procesos que se ejecutaban en otros sistemas del mismo modo que si llamaran a procedimientos locales.



Las dos tecnologías más utilizadas fueron DCOM y CORBA.

- **Servicios web**: fueron creados para permitir el intercambio de información, al igual que RPC, pero sobre la base del protocolo HTTP. En lugar de definir un protocolo propio para transportar las peticiones de información, utilizan HTTP para este fin. Por lo tanto, cualquier ordenador que pueda abrir una página web, será capaz también de solicitar información a un servicio web.



Servicios web.

El desarrollo de aplicaciones web moderno requiere de la interacción entre diferentes sistemas. En la actualidad se han desarrollado multitud de frameworks diferentes, tanto para el backend como para el frontend o UI. Este desarrollo en paralelo de diferentes tecnologías para diferentes partes de nuestras aplicaciones necesitará de un mecanismo que permita el intercambio de información entre todos estos diferentes frameworks, sin depender de ninguno en concreto.

Un servicio web es una tecnología que comunica dos aplicacións informáticas a través de da Web, empleando un conxunto de protocolos estándares para posibilitar el intercambio de información.
Los servizos web permiten publicar conjuntos de procedimientos (APIs) independentes de la plataforma, de modo que puedan ser accesibles desde aplicaciones clientes u otros servizos a través de la red.

Se diferencian dos tipos diferentes de actores:

- El proveedor o servidor, que es aquel que crea y publica la API que da acceso a la información.
- El consumidor o cliente, que es el que establece la comunicación con el provedor para obtener dicha información.

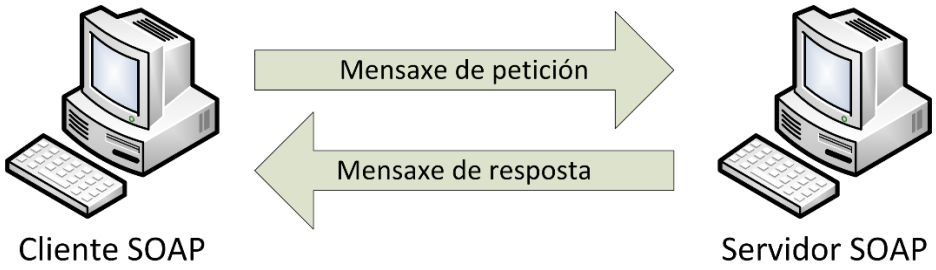
Para el intercambio de la información se utilizarán formatos estándar, siendo los más habituales *xml* y *json*.

Tipos de servicios web.

A pesar de que existe alguna tecnología más, vamos a ver las dos más importantes y ampliamente utilizadas, que son SOAP y REST.

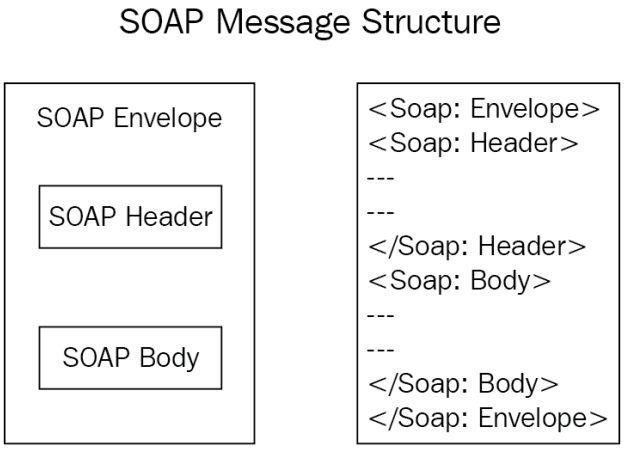
SOAP.

Simple Object Access Protocol (SOAP), es un protocolo para el intercambio de información entre sistemas basado en XML. Se apoya en protocolos de la capa de aplicación, principalmente HTTP, aunque también puede emplear SMTP o FTP. Al funcionar sobre HTTP, con el que puede trabajar por defecto cualquier SO, un servicio web SOAP puede ser llamado desde cualquier plataforma y con cualquier lenguaje de programación.



Estructura de los mensajes SOAP.

El formato SOAP es un estándar creado y mantenido por W3. A continuación se muestra un diagrama de la estructura de un mensaje SOAP:



En un mensaje encontraremos los siguientes elementos:

- **Envelope**: es un elemento obligatorio en un mensaje SOAP. Es el elemento raíz y define el comienzo y el final del mensaje.
- **Header**: la cabecera es opcional y contiene información referente al mensaje. En caso de aparecer debe ser el primer elemento del sobre, antes del cuerpo del mensaje.
- **Body**: parte principal del mensaje, contiene la información que se envía en formato XML.
- **Fault**: un elemento opcional Fault puede añadirse al mensaje para proporcionar información de posibles errores.

A continuación podemos ver un ejemplo de una petición SOAP:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ns1="http://miservicioweb.com/monedas"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:soap-enc="http://schemas.xmlsoap.org/soap/encoding/"
  soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <soap:Body>
      <ns1:getCambio>
     <param0 xsi:type="xsd:int">3</param0>
     <param1 xsi:type="xsd:int">1</param1>
      </ns1:getCambio>
  </soap:Body>
</soap:Envelope>
```

Y un mensaje de respuesta:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ns1="http://miservicioweb.com/monedas"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:soap-enc="http://schemas.xmlsoap.org/soap/encoding/"
  soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <soap:Body>
      <ns1:getCambioResponse>
       <return xsi:type="xsd:float">1.2416</return>
    </ns1:getCambioResponse>
  </soap:Body>
</soap:Envelope>
```

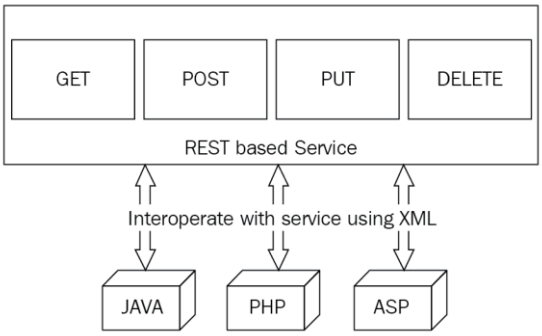
REST.

REST (REpresentational State Transfer) es un conjunto de reglas de referencia para el diseño de arquitecturas distribuidas, de forma que diferentes sistemas puedan comunicarse entre sí de forma sencilla. A los servicios web diseñados conforme a estas reglas se les conoce como servicios RESTful.

Los servicios RESTful son mucho más simples que los servicios SOAP, razón por la cual hoy en día es la tecnología predominante. En un servicio RESTful los recursos se representan por URIs y tienen un tipo concreto de representación como json, xml, pdf, jpg, etc.. Las posibles acciones sobre los recursos se definen empleando los diferentes verbos HTTP.

Sin embargo, no existe una implementación REST si no que esta será diferente dependiendo del desarrollador. Esto no es un problema puesto que todos ellos cumplirán las reglas de referencia.

A continuación podemos ver un diagrama de un servicio RESTful:



Una de las características más importantes de un servicio basado en REST es que puede ser utilizado por aplicaciones desarrolladas en diferentes lenguajes de programación, al tratarse de una arquitectura estándar.

Características de REST.

Los servicios web son sistemas cliente servidor. La información que envía el servidor siempre es en respuesta a una petición de un cliente. Esta respuesta consiste en un recurso en un determinado formato de representación. Los formatos más habituales pueden ser .json, .xml, .pdf, .doc, entre otros.

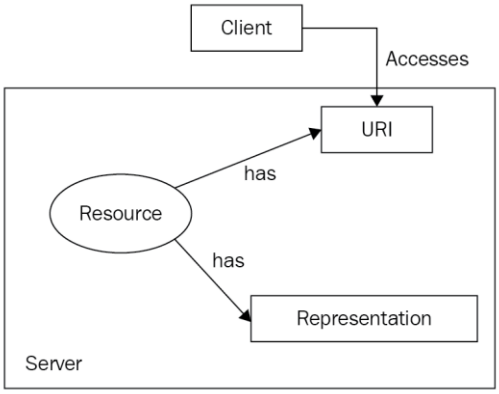
Los servicios RESTful son sin estado, esto significa que cuando una petición llega al servidor, esta se atiende y se olvida. El resultado de la siguiente petición no depende del estado de la anterior, si no que cada petición se maneja de forma independiente.

Las peticiones se realizan por medio de una conexión HTTP, tomando cada una la forma de un URI. Este identificador permite localizar el recurso requerido en el servicio web.

Un servicio web basado en REST puede ser desarrollado en cualquier plataforma o lenguaje de programación, siempre que estos soporten HTTP.

Arquitectura orientada a recursos.

A cada recurso en la web se le asigna un único identificador o URI. Los localizadores uniformes de recursos o URL son el tipo más común de URI utilizado en la web hoy en día. En el siguiente diagrama Podemos ver cómo un cliente obtiene un recurso a través de su URL. El recurso se encuentra disponible en el servidor con una determinada representación, y éste es devuelto al cliente cuando lo solicita:



Como ya sabemos, un URI consiste en una serie de componentes:

```
<protocolo> : <ruta> [ ? <consulta> ] [ # <fragmento> ]
```

Y este un ejemplo de URI:

```
https://en.wikipedia.org/wiki/Packt#PacktLib
```

Arquitectura cliente-servidor.

En la arquitectura REST el cliente o consumidor del servicio no tiene que preocuparse de cómo está implementado el servidor o como hace para almacenar la información. Del mismo modo, el servidor no depende de ningún modo de la implementación del cliente, ni de aspectos como la UI.

Los clientes y los servicios que consumen son entidades independientes que pueden ser diseñadas y evolucionar de forma diferente. Para que el cliente pueda por lo tanto conocer cómo utilizar el servicio, éste proporciona información acerca de cómo debe ser consumido, y las operaciones que se pueden solicitar.

Stateless.

Como ya explicamos anteriormente, el término stateless, o sin estado, se refiere al hecho de que un servicio RESTful no mantiene el estado de la aplicación.

Una petición en un servicio RESTful no depende de otra petición anterior, si no que cada una se atiende de forma independiente. De modo contrario, un servicio stateful o con estado, necesitaría almacenar el estado actual de la aplicación.

Debido a la ausencia de esta complicación, los servicios sin estado son más simples y sencillos de implementar y su mantenimiento es más fácil.

Caching.

Para evitar generar la misma información con cada nueva petición, existe una técnica denominada caching o cacheo, que permite almacenar estos datos, bien en el cliente o en el servidor. Esta información almacenada se utilizará para las peticiones siguientes al mismo recurso.

Esta técnica permite mejorar el rendimiento del servicio pero hay que gestionarlo con mucho cuidado. Estamos almacenando información que no será reemplazada por los datos actualizados del servidor, por lo que es fundamental configurar adecuadamente el tiempo de vida de esta información si no queremos trabajar con datos obsoletos. Además hay que tener en cuenta también la importancia de que los datos estén actualizados, por ejemplo, no es aceptable que la información de un servicio de tipos de cambio de monedas no esté actualizada. Por otro lado, un servicio que devuelva un fichero o una imagen que raramente cambia, puede ser configurado con un mayor tiempo de duración de su caché.

Interfaz uniforme.

Debido a que los servicios y los clientes en un sistema REST son completamente independientes, debemos definir un mecanismo estándar que permita acceder a dichos servicios, sin depender del modo en que han sido implementados.

Por lo tanto, tenemos que seguir determinadas reglas al implementar nuestros servicios para asegurarnos de que un cliente podrá ser capaz de consumirlos. REST define cuatro restricciones que deben cumplirse:

- **Identificación de recursos**: los recursos se identifican de forma única mediante un URI. El siguiente es un ejemplo de un posible identificador para un servicio web:

  ```
  http://omeuservizoweb.com/api/notas/daw/dwes/2014/av02
  ```

- **Manipulación de recursos a través de verbos HTTP**: cuando se accede a un determinado recurso, el servicio debe tener la información que le permita conocer qué hacer con ese recurso, el cliente puede querer acceder a la información, modificarla, o eliminarla. Para ello describiremos el tipo de operación utilizando además de la URI, un método o verbo HTTP determinado.

- **Mensajes auto descriptivos**: los mensajes contienen suficiente información para que tanto el cliente como el servidor sepa como procesar la información. Para ello utilizaremos los tipos MIME, que indicarán el formato de los datos incluidos en el mensaje.

- **Hypermedia as the engine of the application state (HATEOAS)**: la información devuelta por el servidor incluirá futuras acciones en forma de enlaces o URIs. Por ejemplo, una petición para crear un nuevo objeto devolverá la URI para acceder a dicho objeto, una vez creado.

HTTP proporciona un conjunto de métodos, llamados verbos. Estos verbos se utilizan en los servicios web RESTful para indicar diferentes operaciones sobre los recursos. A continuación se listan los verbos junto con las tareas para las que se utilizan en los servicios web:

Método	Operación	Tipo
GET	Leer / obtener un recurso	Seguro
PUT	Crea un nuevo recurso o lo actualiza si ya existe	Idempotente
POST	Crea un nuevo recurso o actualiza uno existente	No idempotente
DELETE	Elimina un recurso	Idempotente
OPTIONS	Obtiene una lista de las operaciones permitidas sobre un recurso	Seguro
HEAD	Devuelve sólo las cabeceras sin ningún cuerpo	Seguro

Un método seguro no tiene ningún efecto sobre el valor original del recurso. Los métodos GET, OPTIONS y HEAD son seguros por que solamente obtienen la información solicitada, sin realizar ninguna modificación sobre la misma.

Un método idempotente obtiene el mismo resultado sin importar el número de veces que este se lleve a cabo. Por ejemplo, un método DELETE al que se llame varias veces sobre el mismo recurso no resultará en un estado diferente del sistema, ya que el estado será el resultante de eliminar ese recurso, de modo que las siguientes llamadas no provocarán otro cambio en el sistema al no existir el elemento a eliminar.

Es importante aclarar la diferencia entre PUT y POST. Ambos métodos pueden utilizarse para crear nuevos recursos o modificar uno existente. La diferencia es que POST, al ser no idempotente, es no repetible. Esto quiere decir que si tratamos de crear un nuevo recurso con POST, cada nueva llamada consecutiva sin especificar una URI determinada crearía un nuevo recurso, modificando por lo tanto el estado del sistema. Con PUT, sin embargo, se comprueba primero si el recurso existe y no se crearía uno nuevo.

Ventajas y desventajas de los servicios RESTful.

A continuación enumeramos algunas de las ventajas e inconvenientes de este tipo de servicios.

Ventajas:

- No dependen de ninguna plataforma ni lenguaje de programación concretos.
- Utilizan métodos estándar HTTP.
- No almacenan el estado.
- Soportan caching.
- Accesibles para cualquier tipo de cliente, como aplicaciones móviles, web, de escritorio, etc..

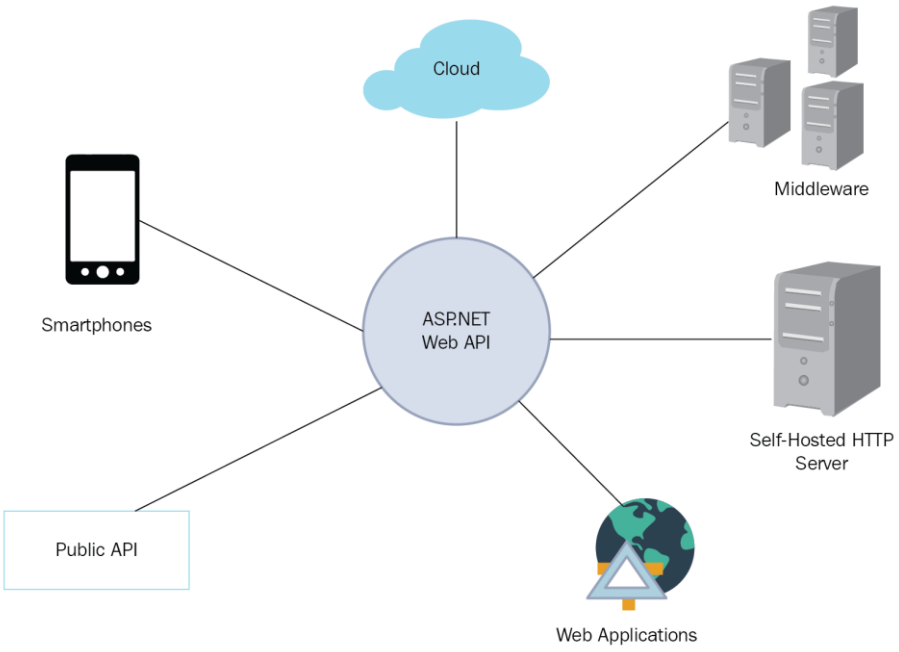
Desventajas:

- Si los estándares no se siguen correctamente, sería complicado para un cliente utilizar dicho servicio.
- Debe implementarse algún tipo de mecanismo para restringir el acceso a los recursos para evitar problemas de seguridad.

Servicios RESTful en ASP.NET Core.

Para crear servicios web en ASP.NET Core debemos crea un proyecto del tipo ASP.NET Web API. Una web API se construye sobre la base de un proyecto ASP.NET MVC, añadiendo algún elemento para simplificar el trabajo con los verbos HTTP. Un proyecto Web API gestionará automáticamente la recepción y envío de los mensajes para que se ajusten a las reglas REST, de modo que lo único que tendremos que hacer es crear controladores que respondan a las peticiones, de la misma forma que lo hemos hecho en nuestras aplicaciones web. Esto es una ventaja porque veremos que desarrollar servicios web es muy parecido a crear una aplicación web ASP.NET Core MVC.

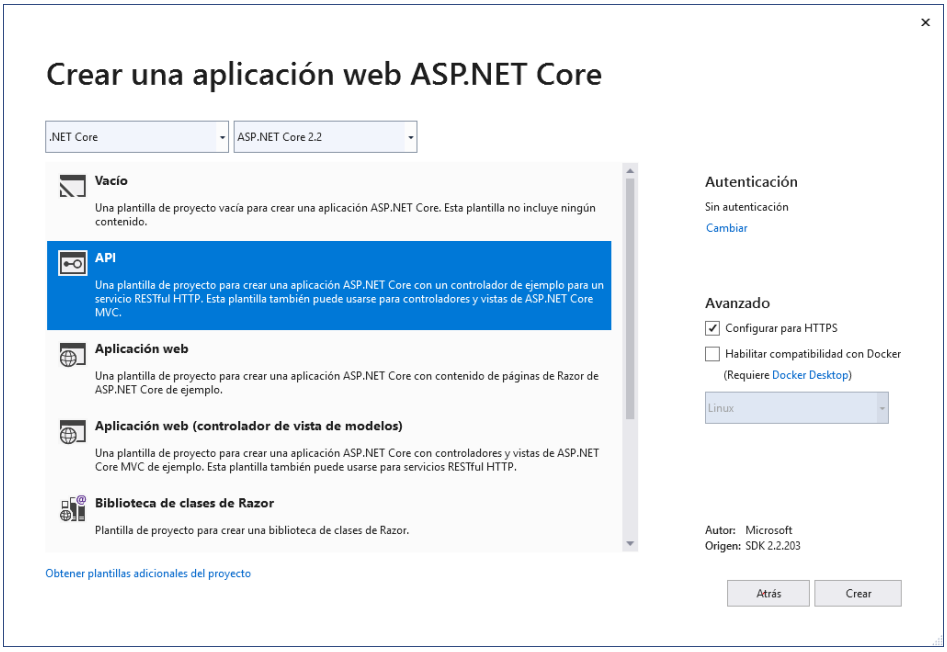
Un proyecto de web API podrá ser consumido por cualquier tipo de cliente desarrollado para utilizar un servicio RESTFul.



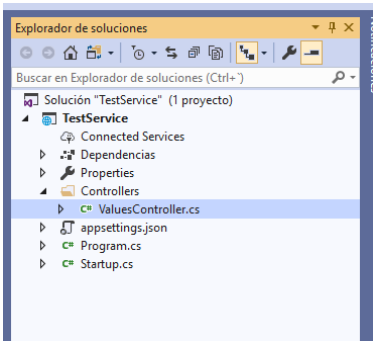
Ejemplo: Creación de un servicio web ASP.NET Core.

Vamos a ver mediante un ejemplo cómo crear un servicio web sencillo con ASP.NET Core utilizando Visual Studio 2019.

1. Comenzamos creando un proyecto de tipo Aplicación web ASP.NET Core, y escogemos crear una Web API.



2. Podemos ver que la estructura del proyecto es muy similar a la de las aplicaciones web MVC que hemos desarrollado hasta ahora. Lógicamente no hay ninguna carpeta *Views* ni *wwwroot*, puesto que no necesitamos interfaz de usuario. Podemos ver una carpeta para nuestros controladores que contiene un controlador por defecto.



3. Si observamos el código de la clase Startup, veremos que la creación de servicios web utiliza la infraestructura de MVC, ya que emplearemos controladores para implementar las operaciones ofrecidas por nuestra API.

```
// This method gets called by the runtime. Use this method to add services to the
container.
public void ConfigureServices(IServiceCollection services)
{
```

```
    services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}

// This method gets called by the runtime. Use this method to configure the HTTP
request pipeline.
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        // The default HSTS value is 30 days. You may want to change this for
production scenarios, see https://aka.ms/aspnetcore-hsts.
        app.UseHsts();
    }

    app.UseHttpsRedirection();
    app.UseMvc();
}
```

4. A continuación, vamos a analizar el código del controlador por defecto:

```
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
    // GET api/values
    [HttpGet]
    public ActionResult<IEnumerable<string>> Get()
    {
        return new string[] { "value1", "value2" };
    }

    // GET api/values/5
    [HttpGet("{id}")]
    public ActionResult<string> Get(int id)
    {
        return "value";
    }

    // POST api/values
    [HttpPost]
    public void Post([FromBody] string value)
    {
    }

    // PUT api/values/5
    [HttpPut("{id}")]
    public void Put(int id, [FromBody] string value)
    {
    }

    // DELETE api/values/5
    [HttpDelete("{id}")]
    public void Delete(int id)
    {
    }
}
```

En este controlador hay muchas cosas interesantes que analizar. En primer lugar podemos ver que se incluye un atributo Route. En lugar de definir el enrutado con un mapeado, como hacíamos en MVC, en una API web indicaremos la ruta de cada servicio mediante el atributo Route. En este caso vemos que la URL de nuestro servicio se formaría con la ruta *api/controller name*. De modo que en este ejemplo, al llamarse el controlador *ValuesController*, la ruta para acceder al servicio sería /api/values. Al igual que en MVC el nombre del controlador se obtiene del nombre de la clase eliminando la parte *Controller* del final.

A continuación se definen una serie de métodos a los que cada uno se asigna un verbo HTTP utilizando para ello un atributo. Es importante tener en cuenta que en este caso los nombres de los métodos no tienen importancia, cada controlador implementa el acceso a un recurso, con una URL única, y se llamará a un método u otro en función del método HTTP que se use en la petición.

Otra cuestión interesante es que tenemos dos diferentes métodos que utilizan el verbo GET, pero en este caso uno recibe un parámetro mientras que el otro no, de modo que la URL sería diferente, por lo que no se incumple la norma REST, ya que las peticiones al primero de los métodos *Get* tendrían la forma */api/values*, mientras que para utilizar el otro método utilizaríamos una ruta como la siguiente */api/values/1*.

5. Vamos a probar a ejecutar la aplicación y ver qué es lo que ocurre.

Nuestro navegador hace una petición a la ruta *api/values*. Como es bien sabido, los navegadores web, por defecto, realizan peticiones GET. Por lo tanto, nuestro servicio localiza el recurso y según el verbo HTTP selecciona la acción a realizar. Nuestro método *Get* devuelve un array de cadenas que el navegador recibe como respuesta, y muestra estos datos en la pantalla. Es importante destacar que los datos devueltos se pasan por defecto con formato .json. Inicialmente los servicios web utilizaban principalmente xml para el intercambio de datos, pero últimamente el formato predominante es json, que es mucho más simple y ligero.

Ya tenemos un servicio web funcionando, en nuestro caso lo hemos probado desde un navegador web, pero podría ser consumido desde cualquier tipo de cliente.

Bibliografía:

- Building RESTful Web Services with .NET Core
  Gaurav Aroraa and Tadit Dash
  Copyright © 2018 Packt Publishing