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

UD3 ASP.NET Core MVC.
3.1 Aplicaciones web y ASP.NET.

En esta unidad comenzaremos a centrarnos en la programación de aplicaciones web, introduciendo primero una serie de conceptos generales sobre las aplicaciones web y la tecnología que utilizan. A continuación nos centraremos en ASP.NET Core, explicando qué es y en que modo podemos desarrollar aplicaciones web utilizando este framework.

Como apoyo al material de la unidad se recomienda el siguiente libro:

https://www.syncfusion.com/ebooks/asp_net_core_succinctly

Además de la documentación oficial de ASP.NET Core:

https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.1

Introducción a las aplicaciones web.

Todas las aplicaciones web, independientemente de la tecnología con la que se desarrollen utilizan como base de su funcionamiento el protocolo HTTP, o en HTTPS, que no es otra cosa que una versión segura de HTTP en la que la información es encriptada antes de ser enviada, para prevenir ataques del tipo man-in-the-middle.

El protocolo HTTP.

Un protocolo no es más que un conjunto de reglas que rigen un proceso de comunicación. HTTP es un protocolo sin estado (stateless) que sigue el patrón request-response (petición-respuesta). HTTP define el formato de los mensajes y la forma en la que estos serán transmitidos.

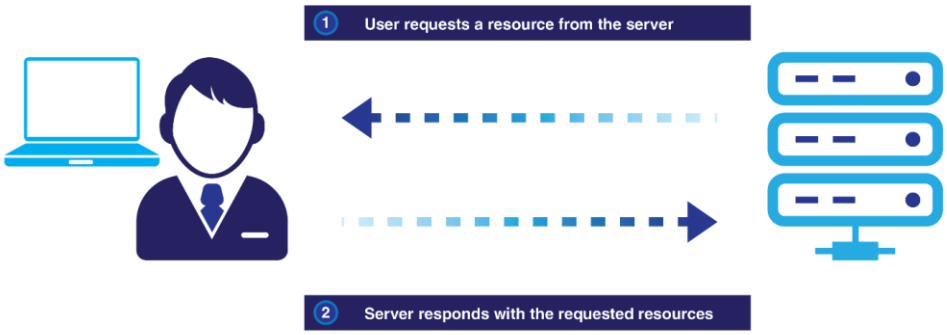
HTTP fue creado por Sir Tim Berners-Lee en 1989. El estándar HTTP fue desarrollado por la **Internet Engineering Task Force (IETF)** y el **World Wide Web Consortium (W3C)**. La versión más común sigue siendo la HTTP 1.1, a pesar de que en 2015 fue estandarizado el protocolo HTTP/2. Esta última versión ofrece una serie de ventajas y mejoras sobre HTTP 1.1, como los mensajes *push* del servidor al cliente, o el soporte de datos binarios, por nombrar algunas. HTTP/2 es soportado por la mayoría de los navegadores web, como Microsoft Edge, Mozilla Firefox o Google Chrome.

El patrón request-response.

Antes de hablar del funcionamiento del patrón, comencemos por definir los dos extremos de la comunicación HTTP, cliente y servidor:

- **Servidor**: Equipo que recibe las peticiones de los clientes y se encarga de servir dichas peticiones. Es típicamente un equipo de gran capacidad de computación y memoria capaz de procesar una gran cantidad de peticiones.
- **Cliente**: Equipo que envía las peticiones al servidor.

Según el patrón request-response, el cliente comienza la comunicación cuando realiza una petición sobre un recurso de un servidor, éste responde entonces enviando de vuelta el recurso solicitado. Un recurso puede ser cualquier objeto, como una página web, un fichero de texto, una imagen, o cualquier dato en cualquier formato.



Cada petición HTTP está compuesto de diversas partes:

- **URI**: La dirección que identifica el servidor y el recurso dentro del mismo.
- **Método (o verbo)**: Define la acción que debe ser llevada a cabo al recibir la petición.
- **Headers o cabeceras**: Un conjunto opcional de datos almacenados como una colección de pares clave-valor y que añaden información a la petición.

La respuesta generada por el servidor, a su vez, también contiene diferentes elementos:

- **Status code**: Un número de tres dígitos que describe el resultado de la petición. Por ejemplo, 200 significa Ok, 500 que se ha producido un error en el servidor o 404 que el recurso solicitado no existe.
- **Body**: Los datos enviados de vuelta al cliente.
- **Headers**: Un conjunto opcional de datos almacenados como una colección de pares clave-valor y que añaden información a la respuesta.

Naturaleza stateless (sin estado) de HTTP.

Cuando un cliente realiza más de una petición sobre un mismo recurso de un servidor, éste responde del mismo modo, sin ningún conocimiento del hecho de que ese mismo recurso ya ha sido servido con anterioridad. El protocolo HTTP por sí solo no posee ningún mecanismo que le permita tener conocimiento del estado de las peticiones y recursos servidos. Existen varios mecanismos para guardar el estado de un servidor entre peticiones, pero son de mayor nivel, proporcionando cada framework de desarrollo sus propias herramientas, pero nunca son proporcionados por HTTP.

Ventajas de HTTP.

A continuación se enumera alguna de las ventajas de utilizar el protocolo HTTP:

- Es un protocolo basado en texto que funciona sobre TCP/IP.
- Es transparente a firewalls.
- Es fácil de depurar al estar basado en texto.

- Es entendido por cualquier navegador. Por lo que puede ser utilizado en cualquier dispositivo y plataforma.
- Utiliza un ciclo request-response estandarizado.

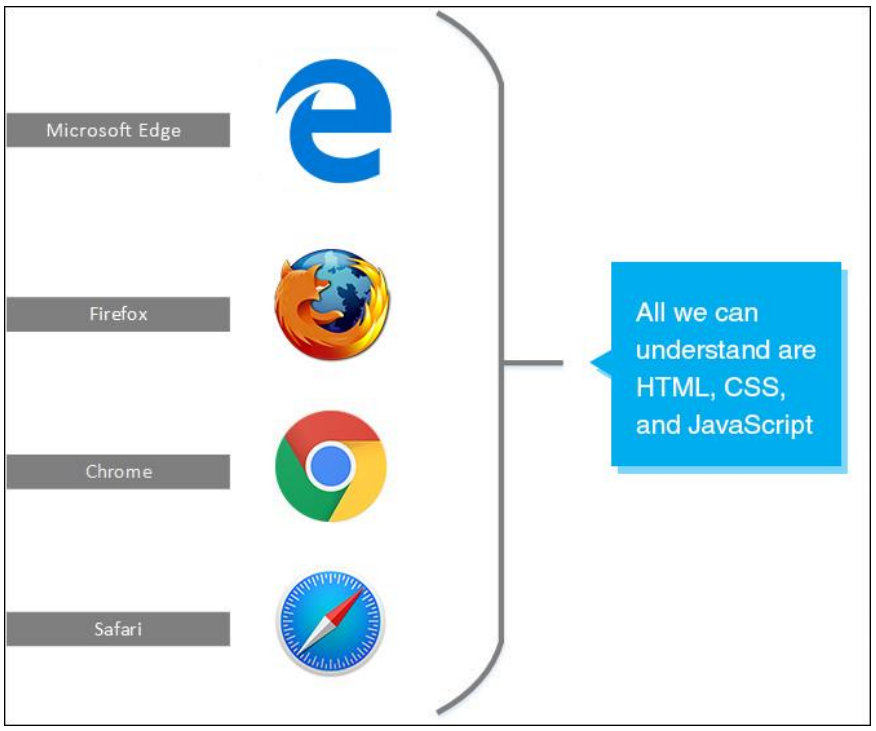
Lado cliente y servidor.

Es importante comprender y diferenciar el lado cliente y servidor de las aplicaciones web y qué es lo tiene lugar en cada uno de ellos. En una aplicación web, el cliente es el navegador y el servidor es el servidor web/servidor de aplicaciones:

- **Lado cliente**: Todo lo que ocurre en el navegador. En este lado se ejecuta código JavaScript y se muestran páginas HTML.
- **Lado servidor**: Las peticiones enviadas desde un navegador llegan al servidor, que ejecutará un determinado código en el mismo y como resultado devolverá la respuesta apropiada. El navegador ignora completamente qué tecnología utiliza el servidor y el lenguaje de programación en el que está escrita la aplicación web. Este será el objeto de este módulo por lo que desarrollaremos el código que se ejecuta del lado servidor, en nuestro caso en C#.

En resumen, es importante tener muy claros los siguientes hechos antes de abordar el desarrollo de una aplicación web:

- Los navegadores web sólo entienden HTML, CSS y JavaScript:
    - Puedes utilizar Firefox, Chrome, Edge o cualquier otro navegador, pero eso no cambiará el hecho de que en el navegador no entenderá C#, Java o Ruby, el navegador es ajeno a la tecnología usada por el servidor, y esta es la razón por la que una aplicación web puede ser utilizada en diferentes navegadores.



- El propósito de cualquier framework de desarrollo web es convertir el código del lado servidor en HTML, CSS y JavaScript:
    - Todas las tecnologías de desarrollo web deben convertir el código de sus aplicaciones en HTML, CSS y JavaScript, para que el navegador sea capaz de mostrar el resultado. Esto es igual estemos desarrollando nuestra aplicación con ASP.NET, J2EE o Ruby on Rails. Eso sí, cada framework llevará a cabo esta tarea de un modo diferente y ofrecerá diferentes capacidades como seguridad o rendimiento. Pero finalmente, todos ellos producirán código HTML como resultado a las peticiones del cliente.

RPC / REST.

Básicamente disponemos de dos modelos diferentes para programar una aplicación web sobre HTTP, RPC (Remote Procedure Calls) y REST (Representational State Transfer):

- **RPC**: Utilizamos HTTP como un medio de transporte. Nuestro servicio proporcionará un conjunto de operaciones que pueden ser llamadas directamente desde nuestro cliente. Es decir, ejecutamos una serie de operaciones como si de métodos locales se tratara, pasando los parámetros requeridos. Para su funcionamiento utiliza otro protocolo basado en XML llamado SOAP (Simple Object Access Protocol). Fue muy popular hace años pero ahora está cayendo en desuso.
- **REST**: Es un estilo de arquitectura para servicios orientados a recursos. Es usado hoy en día por la mayoría de las aplicaciones web para estandarizar las comunicaciones entre el cliente y el servidor. Podemos considerar HTTP como el "lenguaje" de nuestra comunicación, mientras que REST sería un conjunto de reglas aplicadas sobre este lenguaje.

Los servicios web basados en REST, también llamados RESTful APIs, cumplen una serie condiciones que conviene conocer:

- **Interfaz consistente**: En este modelo utilizamos URLs para representar los recursos del servidor, de modo que cada recurso se identifica de forma única. Por ejemplo, una URL como https://api.example.com/students sería un identificador para una colección de estudiantes en el servidor. Esta otra URL https://api.example.com/students/1 se referiría a un estudiante cuyo identificador sea 1.
- **URL y métodos HTTP**: Cuando una URL que identifica un recurso es combinada con un método HTTP, juntos describen una operación que deberá ser llevada a cabo por el servidor. Así por ejemplo, un método GET significa que estamos pidiendo al servidor que nos devuelva el recurso solicitado, POST se utiliza para crear un nuevo recurso, o DELETE para eliminar un recurso existente.
- **Stateless**: Igual que el protocolo HTTP, en los servicios RESTful cada petición es independiente de las anteriores y el resultado de una operación no tiene que ver con las anteriores.
- **Cacheable**: Las respuestas de un servicio pueden ser almacenadas en cache, de modo que si un cliente solicita un recurso 'cacheable', la respuesta se obtendrá de la caché en lugar de tener que ser generada de nuevo por el servidor.

Imaginemos una aplicación web para gestionar estudiantes, podríamos tener servicios como los que se muestran a continuación:

URL	Método HTTP	Descripción
/students	GET	Obtiene todos los estudiantes
/students	POST	Crea un nuevo estudiante
/students/123	GET	Obtiene la información del estudiante cuyo id es 123
/students/123	PUT	Actualiza la información del estudiante con el id 123
/students/123	DELETE	Elimina el estudiante cuyo id es 123

Los datos devueltos pueden formatearse utilizando XML o JSON, dependiendo de la solicitud del cliente, especificando el formato a utilizar en la cabecera de la petición.

Hoy en día la mayoría de frameworks web utilizan REST como mecanismo de comunicación entre el backend y el frontend. Esto es así porque es simple y descansa sobre una serie de métodos estándar. Del mismo modo, aunque hace años era más popular el intercambio de datos con XML, hoy se está imponiendo JSON al ser mucho más ligero y legible.

Métodos HTTP.

Como ya hemos comentado, HTTP define, como parte de su especificación, una serie de métodos, también denominados 'verbos', para indicar una serie de acciones a realizar sobre los recursos especificados. Aunque todos siguen el patrón petición-respuesta, la forma de las peticiones puede variar de unos a otros. La especificación del método define como es enviada la petición al servidor.

Los métodos disponibles en HTTP son GET, HEAD, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT y PATCH, pero de momento nos centraremos en los dos métodos más importantes y ampliamente utilizados en cualquier aplicación web, GET y POST.

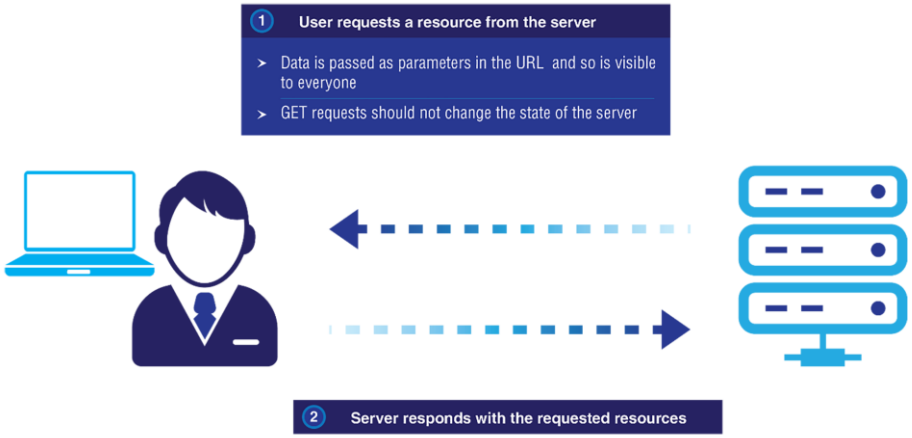
GET.

El método GET se utiliza para obtener un recurso del servidor. Las peticiones que utilizan GET simplemente deben obtener el recurso solicitado pero sin tener ningún efecto secundario en el servidor. Esto quiere decir que si realizamos múltiples peticiones GET sobre el mismo recurso, deberíamos obtener el mismo resultado, sin provocar ningún cambio en el estado del servidor como resultado de la petición GET.

Los parámetros son enviados como una parte de la propia URL de la petición y por lo tanto serán visibles para el usuario. La ventaja de esta aproximación es que el usuario podría marcar la URL como favorito y visitar el mismo recurso en cualquier momento. Ejemplo:

https://dwe.com/?tech=mvc&db=sql

En este caso estamos pasando dos parámetros en esta petición GET, el primero es *tech*, y su valor *mvc*, y el segundo *db*, con el valor *sql*. Un parámetro siempre especifica un nombre y un valor, separados por el signo '=', y utilizaremos '&' para separar diferentes parámetros.



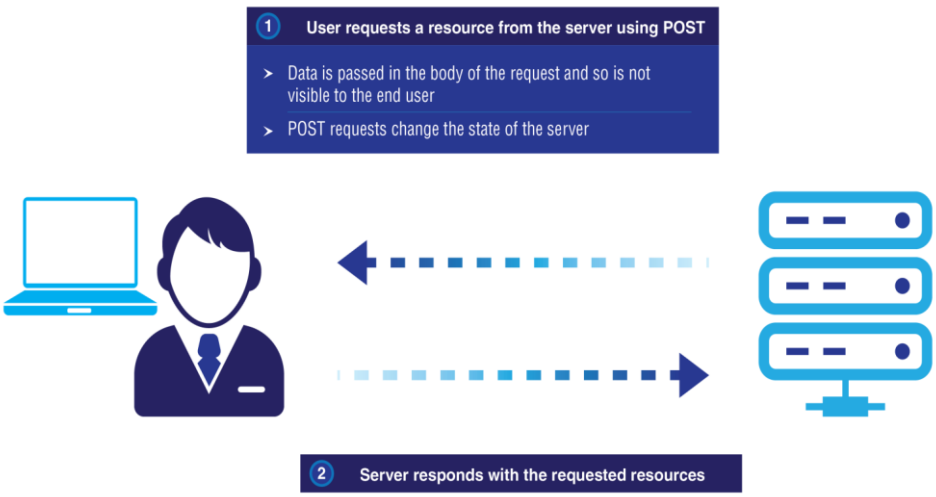
Uno de los problemas de GET es que los parámetros son pasados como texto sin cifrar por lo que no deben ser utilizados para pasar información sensible. Además, muchos navegadores tienen un límite en el número de caracteres de la URL, por lo que no podemos pasar grandes cantidades de información como parámetros de una petición GET.

POST.

El método POST se utiliza normalmente para actualizar o crear recursos en el servidor, además de para enviar información para ser procesada por el servidor, como por ejemplo cuando enviamos los datos de un formulario.

Los datos en este caso son enviados dentro del cuerpo de la petición, lo que tiene las siguientes implicaciones:

- Podemos enviar información sensible, ya que los datos van dentro del cuerpo de la petición y esta no es visible para el usuario. Aún así, esta podría ser interceptada, por lo que lo recomendable es utilizar HTTPS para que toda la información viaje encriptada.
- Como la información no forma parte de la URL de la petición, no tendremos que preocuparnos por la limitación del espacio máximo de la misma.



Características de los métodos HTTP.

Los métodos HTTP se pueden clasificar conforme a 3 diferentes aspectos:

- **Idempotencia**: Se dice que una petición es idempotente cuando si hacemos múltiples peticiones a un servidor, el efecto será el mismo que si sólo hubiéramos hecho una única petición.
- **Seguridad**: Una petición segura es la que no tiene efectos secundarios. Por esto nos referimos a cambios en memoria o una base de datos externa. Por ejemplo, registrar un usuario o realizar un pago son efectos secundarios. Sin embargo, ver los datos de un cliente no lo sería.
- **Cacheabilidad**: El servidor puede guardar en caché las respuestas de modo que se acelere la respuesta a una petición que va a devolver un valor que no ha variado.

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

Introducción a ASP.NET.

ASP.NET es un framework de desarrollo web servidor, desarrollado por Microsoft, que permite a los programadores crear tanto aplicaciones como servicios web. Es muy importante comprender que hay 2 'ramificaciones' o versiones de ASP.NET. La primera es la tradicional ASP.NET for .NET Framework. Este modelo corre sobre la versión clásica de .NET Framework, sólo disponible en sistemas Windows.

En este curso utilizaremos la otra versión, más reciente, denominada ASP.NET Core. Se trata de un rediseño de ASP.NET creado desde cero. Utiliza la nueva plataforma .NET Core, que al ser cross-platform permite que una aplicación pueda ser ejecutada en una gran variedad de plataformas, incluyendo sistemas Linux y macOS. Además está diseñada para ser más ligera y modular que la tradicional y soporta su ejecución sobre contenedores. Actualmente es completamente open source y todo el código fuente se puede encontrar en esta web, mantenida por Microsoft: https://github.com/aspnet

Con ASP.NET Core podremos desarrollar cuatro tipos de proyectos:

- **Web API**: Permite desarrollar servicios web REST.
- **Aplicaciones web**: Permite crear aplicaciones web con un modelo orientado a páginas (Razor Pages). Es un modelo más simple y reciente y es el recomendado por Microsoft.
- **Aplicaciones web MVC**: Permite la creación de aplicaciones web usando el patrón MVC.
- **Single Page Aplications o SPAs**: Permite la creación de páginas web donde la mayor parte del código se ejecuta en la parte cliente, utilizando el servidor para peticiones a una API de servicios.

Como indicamos en la unidad 1, en nuestro caso utilizaremos la plataforma ASP.NET Core y nos decantamos por el modelo MVC, que aún siendo un poco más complejo que Razor Pages ofrece un modelo más maduro y estable y que además es el mismo que se utiliza para la creación de servicios web, lo que aprovecharemos para la última parte del módulo. De todos modos, si deseas informarte más acerca de las diferencias entre un modelo y otro, con una simple búsqueda "razor pages vs mvc" podrás ver que hay una gran controversia y que hay partidarios y detractores de cada uno de los dos modelos.

ASP.NET Core MVC.

ASP.NET Core MVC es la implementación open source y cross-platform del patrón MVC en ASP.NET. Está inspirado en el exitoso Ruby on Rails, y se diseñó para superar las limitaciones del anterior modelo ASP.NET Web Forms, que no ofrecía al desarrollador ningún control sobre el HTML generado por la aplicación web. Hoy en día la parte cliente de las aplicaciones modernas ha ganado en importancia, con librerías/frameworks como jQuery, KnockoutJS, AngularJS, etc.. Esto hace que sea crucial para un framework moderno tener un control total sobre el HTML producido.

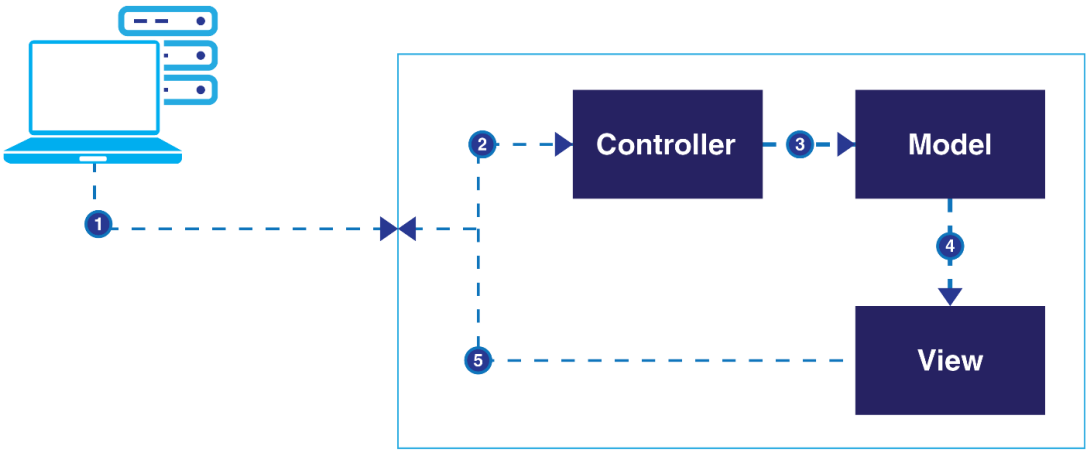
El patrón MVC (Model-View-Controller).

MVC es un patrón de desarrollo de software que ayuda a definir la responsabilidad y tareas de cada uno de los componentes y la forma en las que deben interactuar entre ellos para conseguir desarrollar nuestra aplicación.

Se puede utilizar en diferentes tipos de desarrollo, tanto de escritorio como aplicaciones web, nosotros nos centraremos en la aplicación del patrón en el contexto del desarrollo web.

El patrón MVC identifica tres tipos de componentes:

- **Modelos**: Representan los datos del dominio de la aplicación. Por ejemplo, si estamos desarrollando una aplicación de e-commerce, el modelo podría contener clases como *Product*, *Inventory*, *Customer*, etc..
- **Vistas**: Responsables de presentar la información al usuario. En una aplicación web, se trata normalmente de páginas HTML y hojas de estilo CSS. También puede contener información sobre la disposición o aparencia general de nuestra aplicación (layout).
- **Controladores**: Se encargan de interactuar con el resto de componentes. Las peticiones entrantes son atendidas por un controlador, que consulta el modelo para producir un resultado, que luego es enviado a la vista apropiada para ser mostrado al usuario.



1. User requests the web page by typing the URL in the browser of his computer
2. Based on the URL, the appropriate Controller is selected and the request is forwarded to that Controller
3. Then, the Controller talks to the Model to query/update the data
4. The Controller passes the data to the View and the View is responsible for displaying the data
5. The View is returned to the requested user

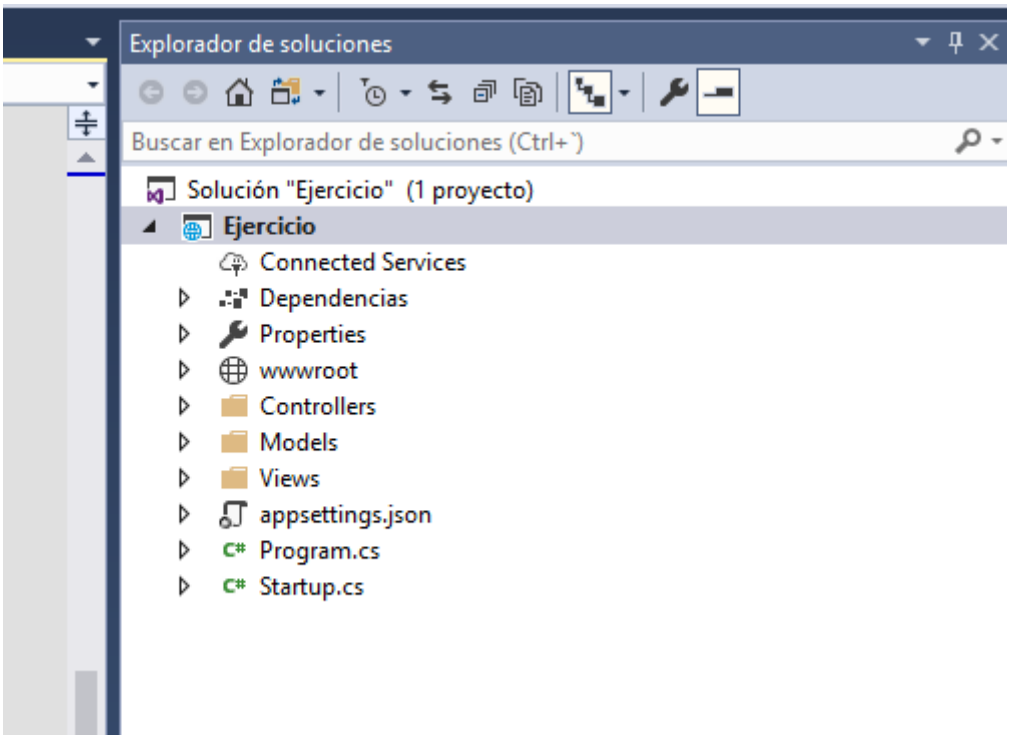
Esta separación de responsabilidades ofrece una gran flexibilidad en el desarrollo de nuestras aplicaciones, permitiendo que cada aspecto pueda ser gestionado y desarrollado de forma separada e independiente.

En ASP.NET Core MVC, cada controlador será una clase que hereda de la clase genérica *Controller*. Cada método público de este controlador representará una *acción* que puede ser ejecutada en respuesta a una petición del cliente.

Modelo basado en convenciones.

ASP.NET Core MVC descansa en una serie de convenciones que hacen más sencillo gestionar los elementos del proyecto sin necesidad de un mecanismo de configuración complejo, a esto se le suele llamar 'convención sobre configuración'. Aunque lo iremos viendo en detalle a lo largo del curso, podemos simplificar diciendo que esto se traduce en que en función del tipo de elemento que añadamos, si seguimos una serie de convenciones, como guardar el elemento en una carpeta determinada o utilizar un nombre concreto, nuestro framework será capaz de identificar y trabajar con ese elemento sin necesidad de especificarlo en ningún fichero de documentación.

A continuación vemos un ejemplo de un proyecto de aplicación web ASP.NET MVC:



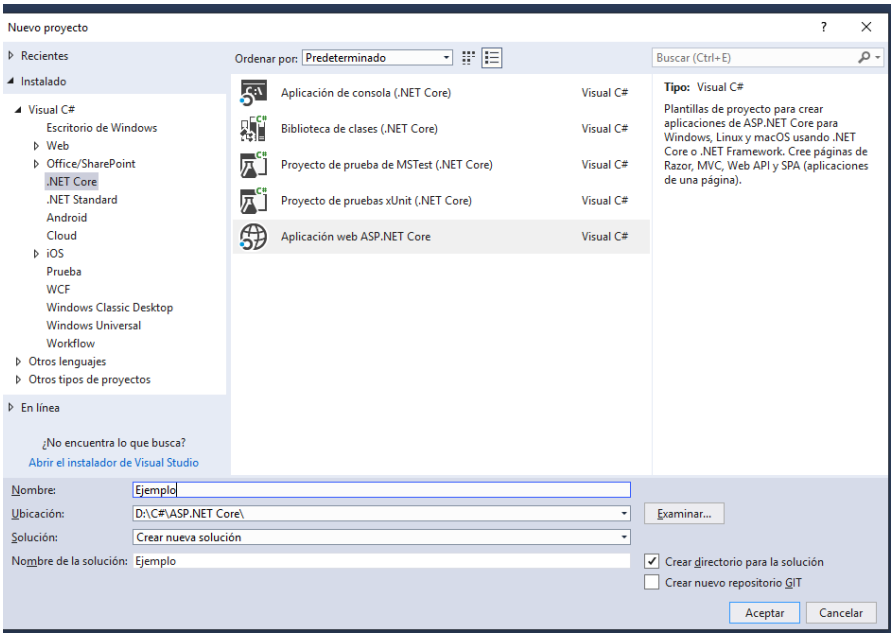
Podemos ver que el proyecto contiene cuatro carpetas de gran importancia. Las carpeta *Controllers*, *Models* y *Views* contienen el código correspondiente a controladores, modelos y vistas respectivamente. La cuarta carpeta que debemos conocer es *wwwroot*, que contendrá todos los recursos estáticos que queramos incluir en nuestra aplicación, como pueden ser imágenes, páginas html estáticas, librerías JavaScript, hojas de estilos, etc..

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

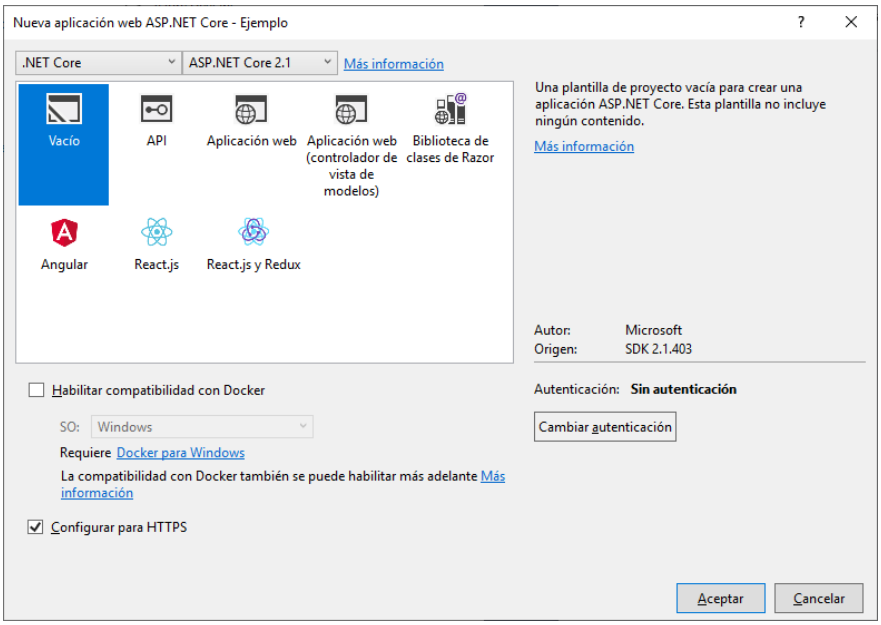
Creación de una aplicación ASP.NET Core MVC.

Hasta ahora hemos utilizado una de las plantillas disponibles para crear nuestros proyectos. Aunque muchas veces será la opción más cómoda ya que realiza gran cantidad de trabajo por nosotros, es importante conocer la función de cada elemento de un proyecto ASP.NET Core, por lo que en este caso comenzaremos de un proyecto vacío e iremos viendo los distintos elementos necesarios para construir una aplicación, lo que nos dará un conocimiento más profundo sobre la estructura del mismo.

1. Comenzamos creando un nuevo proyecto con **Archivo | Nuevo | Proyecto** y seleccionamos **Apliación web ASP.NET Core**.

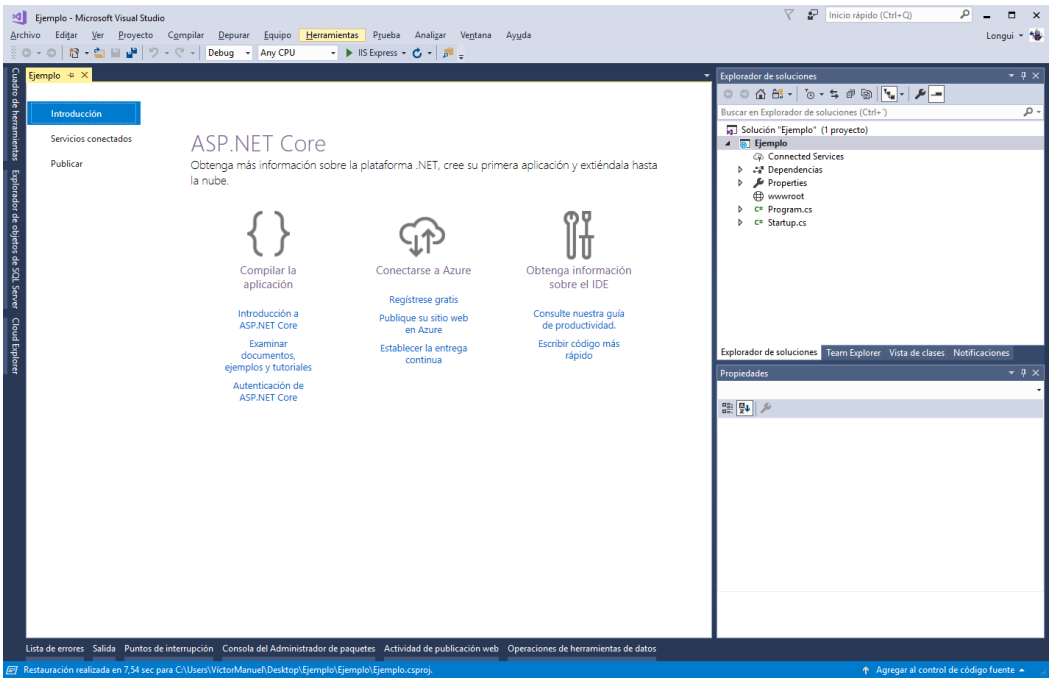


2. A continuación seleccionamos la plantilla de proyecto vacío.



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

3. Cuando aceptemos, nuestro proyecto será creado, y veremos algo similar a lo siguiente:



4. Probaremos su funcionamiento ejecutando la aplicación (F5) y obtendremos un mensaje de **Hello World!**.

En este punto es probable que os encontréis con un error de vuestro navegador, indicando un problema de seguridad. El motivo es que una conexión SSL necesita un certificado para realizar el cifrado. Cuando tenemos una aplicación web en producción debemos disponer de un certificado firmado por alguna de las muchas entidades de certificación disponibles. Sin embargo, para entornos de desarrollo no necesitamos adquirir un certificado y podemos utilizar un certificado autofirmado. Esto es lo que hace ASP.NET Core de forma transparente, crea un certificado denominado ASP.NET Core HTTPS Development Certificate y lo utiliza para las conexiones HTTPS cuando ejecutamos nuestro proyecto.

El problema reside en que los navegadores no confían en este certificado al estar autofirmado, razón por la cual lanzan una excepción de seguridad al usuario indicando que el sitio que estamos visitando no es confiable, que puede decidir si a pesar de este aviso decide seguir adelante.

Para evitar esto debemos indicar a nuestro sistema que confíe en ese certificado. Para hacerlo abriremos una consola y utilizaremos el siguiente comando:

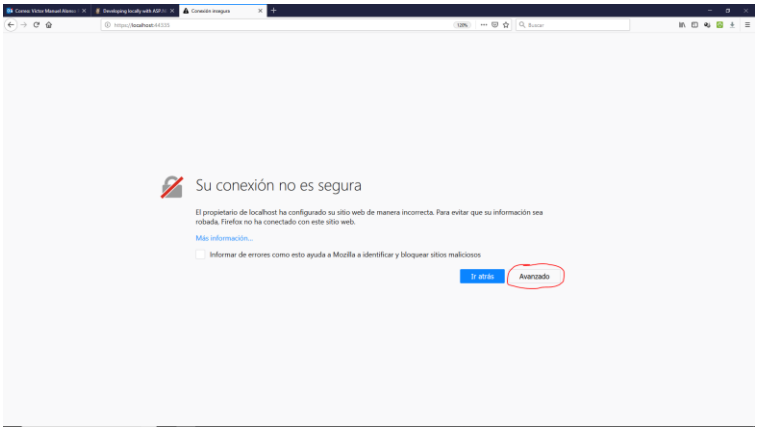
```
> dotnet dev-certs https –trust
```

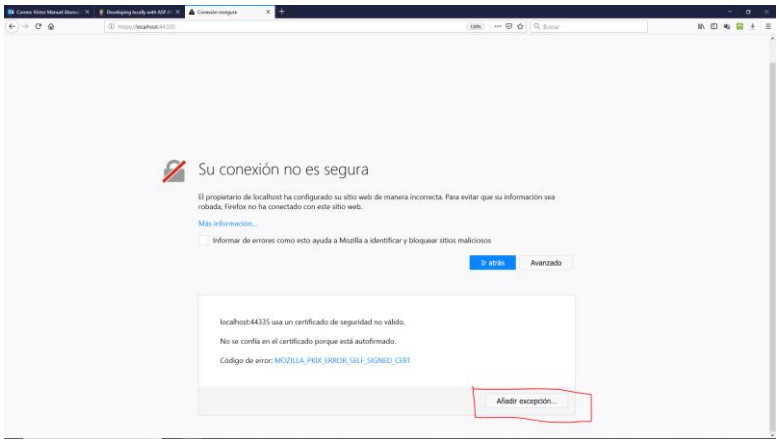
En ese momento se abrirá una ventana indicando que vamos a confiar en un certificado cuya firma no está verificada por ninguna entidad certificadora. Si aceptamos, ya no nos encontraremos con este problema cada vez que ejecutemos nuestros proyectos.

Podéis ver el proceso completo y más información sobre el tema en el siguiente enlace:

https://www.hanselman.com/blog/DevelopingLocallyWithASPNETCoreUnderHTTPSSSLAndSelfSignedCerts.aspx

Desafortunadamente, de un tiempo a esta parte, Firefox no permite ser configurado para confiar en certificados autofirmados, de este modo seguiremos teniendo este problema con este navegador. Para poder ejecutar nuestros proyectos en Firefox debemos añadir una excepción de seguridad que acepte el mismo.

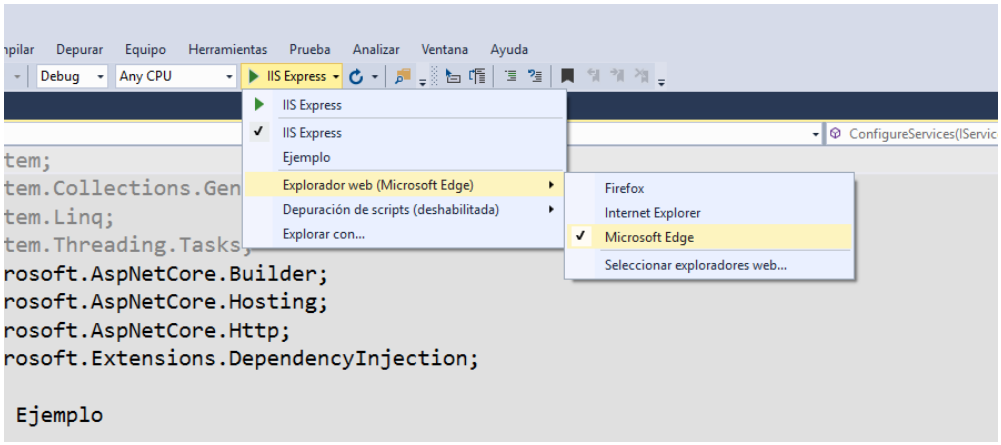




El problema es que cada excepción se configura para un número de puerto determinado, por lo que al ejecutar un proyecto diferente o incluso el mismo proyecto el puerto utilizado sea diferente, lo que nos obligará a añadir nuevas excepciones de seguridad con cada proyecto que ejecutemos.

Con otros navegadores, como Edge, Chrome o Internet Explorer, no deberíamos tener este problema una vez confiado en el certificado, así que podemos configurar nuestro proyecto en

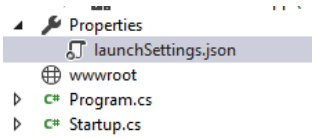
Visual Studio para que utilice por defecto uno de estos navegadores, tal y como se muestra en la imagen:



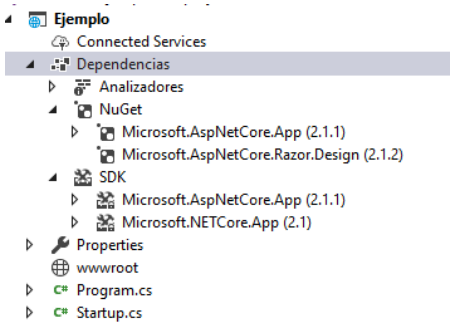
Ficheros por defecto en un proyecto ASP.NET Core.

A pesar de que hemos escogido un proyecto vacío, podemos ver que en realidad nuestro proyecto no está completamente vacío, si no que contiene una serie de elementos que todo proyecto ASP.NET Core debe tener.

- **Program.cs**: Toda aplicación necesita un punto de entrada, que es una clase que contenga un método *Main*() que será el primer método en ser ejecutado. En este caso es la clase *Program* la que se encarga de esto, además arrancar un proceso Web Host por defecto para alojar nuestra aplicación web y cargar la configuración.
- **Startup.cs**: En esta clase se especifica la configuración de los componentes y los servicios utilizados en nuestra aplicación web, y es utilizada por *Program* para arrancar el Web Host con la configuración especificada.
- **Carpeta wwwroot**: Contiene todos los recursos estáticos que la aplicación utilizará, tales como hojas de estilos, ficheros js, imágenes, páginas html estáticas, etc..
- **launchSettings.json**: Fichero que se encuentra dentro de la carpeta *Properties* y que contiene diversos parámetros de configuración que Visual Studio utiliza en el momento de ejecutar la aplicación.



- **Dependencias**: Librerías utilizadas por nuestra aplicación, podemos encontrar los SDK de ASP.NET Core y .NET Core, necesarios para toda aplicación ASP.NET Core, además podemos utilizar tantas librerías de terceros como necesitemos. Para ello disponemos de una herramienta denominada NuGet, que mantiene un repositorio de librerías y que nos permitirá buscar, instalar o actualizar las librerías necesarias.



La clase *Startup*.

La clase Startup es inicializada y ejecutada justo después de arrancar la aplicación web. Es el lugar donde se configuran los servicios y componentes de la aplicación.

Esta clase contiene dos métodos:

- **ConfigureServices**: En este método se añaden a la aplicación todos los servicios necesarios. Ejemplos de servicios pueden ser el de logging o registro y el autenticación. En la plantilla de proyecto vacío no se utiliza inicialmente ningún servicio.
- **Configure**: Aquí es donde los servicios y los componentes de la aplicación son configurados. Siguiendo con el ejemplo anterior, aquí se configurarían los parámetros de los servicios de logging o autenticación.

En nuestro proyecto nos encontraremos con el siguiente código:

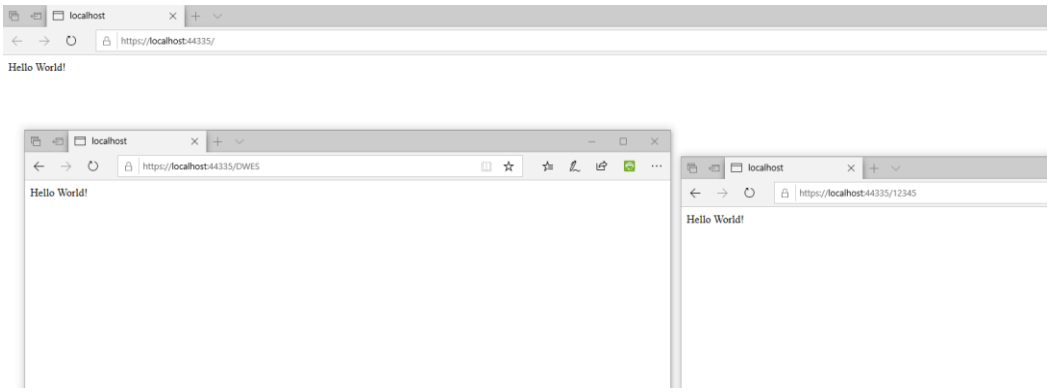
```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
}
```

En primer lugar indicamos que si nos encontramos en un entorno de desarrollo queremos que se muestre la página de excepciones de desarrollador en caso de que se produzca un error al ejecutar nuestra aplicación. Esta página contendrá información que podrá ser muy útil para localizar y depurar errores, pero nunca debe mostrarse en un entorno de producción por motivos de seguridad.

El siguiente fragmento de código puede parecernos más complejo debido al uso de una serie de palabras clave que no hemos visto hasta el momento, como pueden ser *async*, *await*. Sin entrar de momento en detalles nos quedaremos simplemente con que el código de este método captura cualquier petición al servidor y genera una respuesta HTTP en la que escribe directamente el mensaje **Hello World!**.

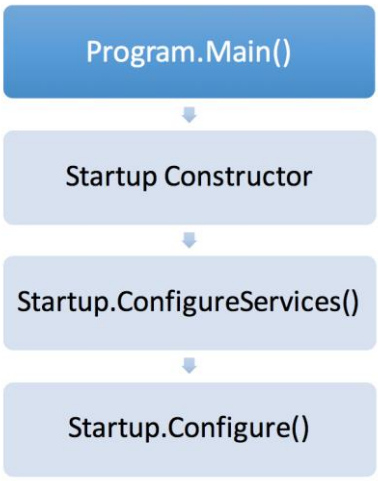
Por supuesto este no es el comportamiento deseado en una aplicación web, ya que independientemente del recurso solicitado la respuesta será la misma. Podemos comprobar esto modificando la URL en nuestro navegador, de modo que indiquemos distintos recursos del servidor y comprobando como el resultado no varía.



Orden de ejecución.

La clase Startup es ejecutada una sola vez durante la vida de la aplicación, justo después de que la aplicación es arrancada. Esta clase no debe contener más código que el relativo a la inicialización y configuración.

A continuación podemos ver el orden de ejecución de los diferentes métodos y las clases implicadas en la inicialización de una aplicación ASP.NET Core:



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

Ficheros estáticos.

En las aplicaciones web, los ficheros estáticos son ficheros que se encuentran en el servidor web y que se sirven al cliente tal cual, sin ninguna manipulación por parte del servidor. Dentro de este tipo de contenido podemos encontrar ficheros HTML, CSS, JavaScript, imágenes, fuentes, videos, etc..

Estos ficheros son especialmente importantes en las SPAs, ya que el código del frontend está basado en ficheros HTML, CSS, y JavaScript estáticos.

Las carpetas web root y content root.

ASP.NET Core define dos tipos de carpetas:

- **Content root**: Directorio raíz de la aplicación. Cualquier fichero fuera de esta carpeta no se considera parte de la misma.
- **Web root**: Lugar en el que se localizan los ficheros estáticos de la aplicación.

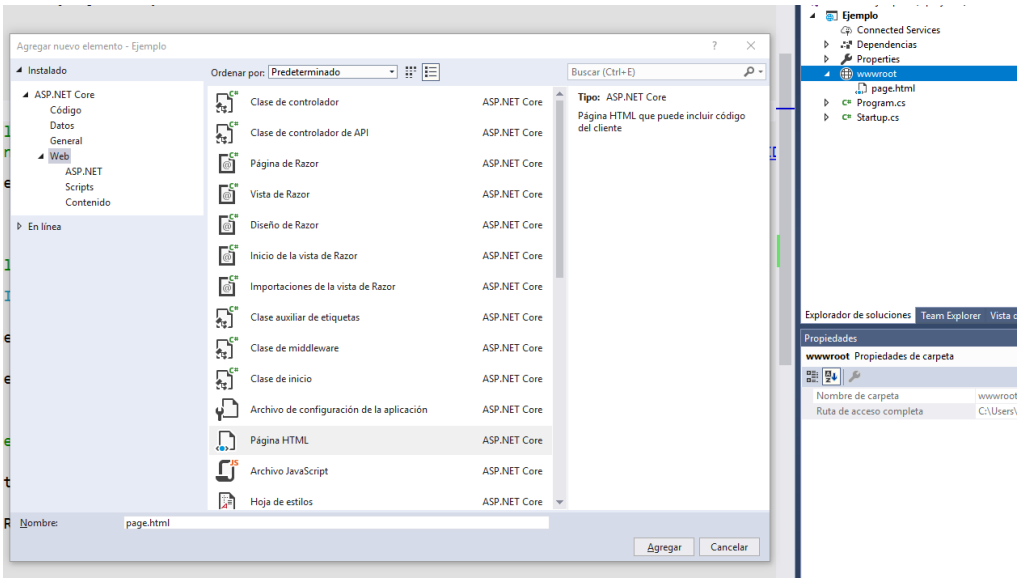
Por defecto, la content root es la carpeta raíz de la aplicación, y la web root sería la carpeta *wwwroot* que se encuentra en ella, es decir *<content root>/wwwroot.*

Por defecto la carpeta wwwroot no está disponible por defecto para el cliente. Esto significa que aunque un fichero se encuentre correctamente guardado en dicha carpeta, el servidor no devolverá el mismo al cliente a no ser que configuremos el proyecto para hacerlo.

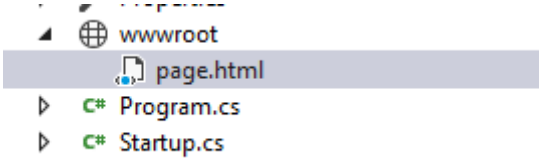
Configurando nuestra aplicación para servir contenido estático.

Vamos a seguir con nuestra aplicación de ejemplo para ilustrar lo que acabamos de comentar sobre el contenido estático.

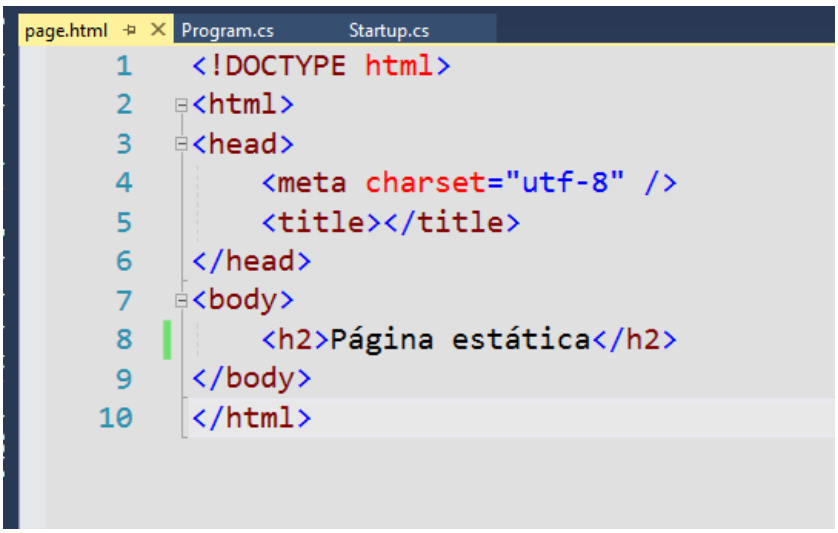
1. Comenzamos creando un nuevo fichero HTML dentro de la carpeta wwwroot, haciendo click derecho en la misma **Agregar | Nuevo Elemento…** y seleccionamos **Página HTML**. La llamaremos *'page.html'*.



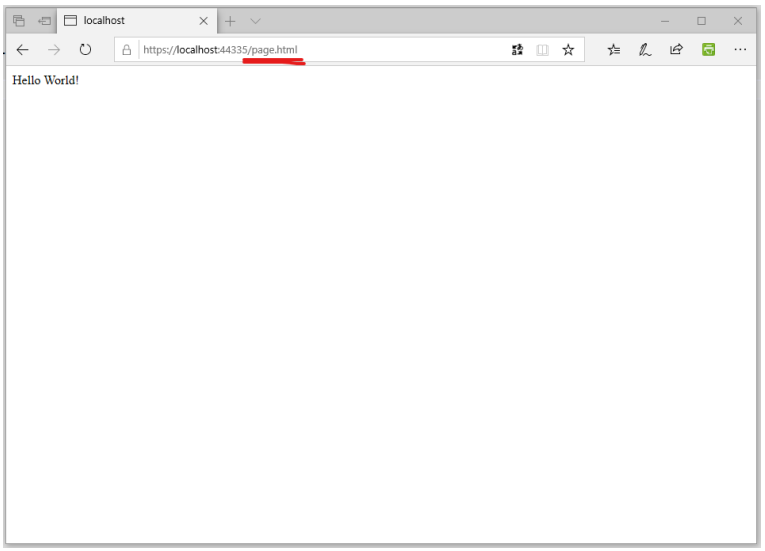
2. Comprobamos que hemos creado nuestro fichero html en la carpeta wwwroot.



3. A continuación editamos la página para que muestre un mensaje.



4. Ahora vamos a ejecutar nuestra aplicación, añadiendo a la url el nombre de la página que queremos que sea mostrada en nuestro navegador, en este caso */page.html*.



Como era de esperar, la página no se muestra a pesar de que se encuentra en la carpeta correcta.

5. Vamos a editar la configuración de nuestra aplicación indicando que permita servir contenido estático. Para ello abrimos el fichero *Startup.cs* y añadimos la línea siguiente al método *Configure()*.

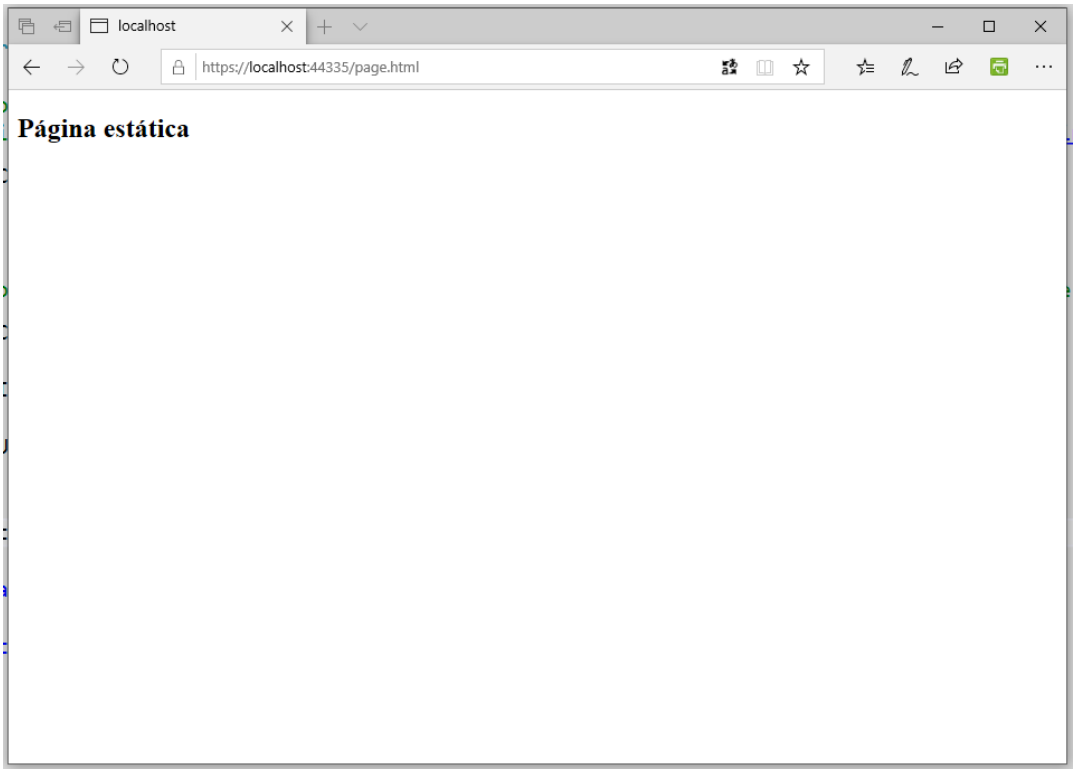
```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseStaticFiles();

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
}
```

Como podemos ver el código es bastante sencillo y autoexplicativo. Como hemos dicho anteriormente, en este método incluiremos código de configuración e inicialización. Con esta línea estamos diciendo a nuestra aplicación '*app*' que utilice ficheros estáticos.

6. Ahora ejecutaremos nuestra aplicación de nuevo, y como antes añadiremos a la url el nombre de la página */page.html*. Ahora vemos que si podemos ver el contenido del fichero estático en nuestro navegador.


localhost
https://localhost:44335/page.html
Página estática

Bibliografía:

- ASP.NET Core 2 Fundamentals.
  OnurGumus, Mugilan T.S. Ragupathi
  Copyright © 2018 Packt Publishing

- Hands-On Full-Stack Web Development with ASP.NET Core
  Tamir Dresher, Amir Zuker and Shay Friedman
  Copyright © 2018 Packt Publishing