UD 2. ASP.NET Core. Características. El lenguaje C#.

El objetivo principal de esta unidad será el de familiarizarnos con el trabajo con el lenguaje de programación C# y sus características principales.

C# es un lenguaje orientado a objetos, fuertemente tipado y cuya sintaxis es muy similar a C, C++ o Java, de modo que cualquiera con conocimientos en estos lenguajes podrá ser capaz de trabajar rápidamente con C#. Se trata de un lenguaje de propósito general, que se puede utilizar para desarrollar todo tipo de aplicaciones, bien sean aplicaciones de consola o escritorio, juegos, aplicaciones móviles o aplicaciones web, tal y como es el propósito de este módulo.

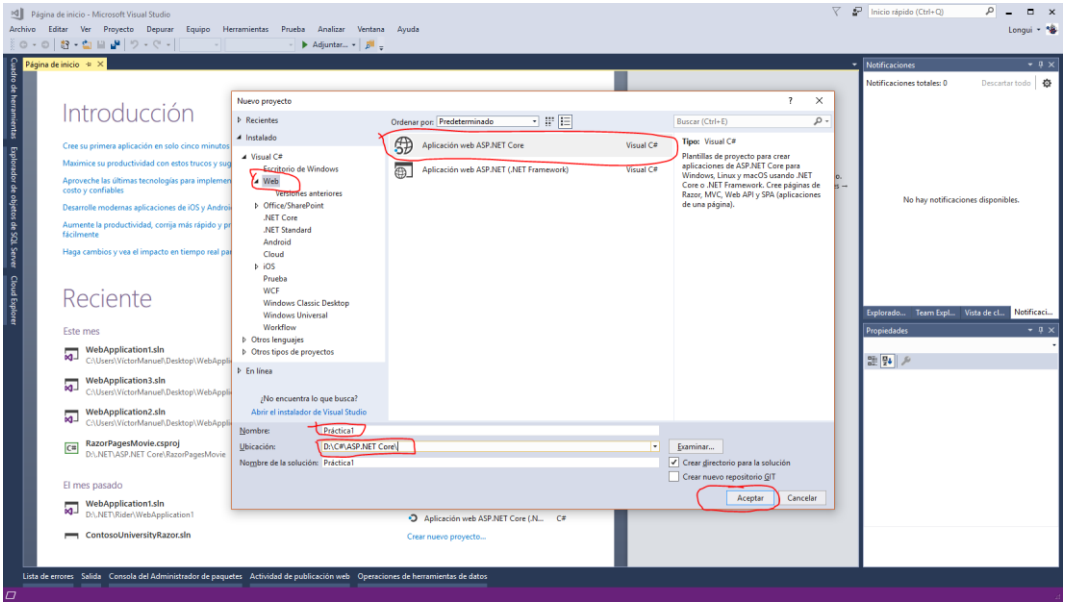
A fecha de hoy se encuentra entre los lenguajes de programación más utilizados, según el índice TIOBE, como se puede comprobar en el siguiente enlace:

https://www.tiobe.com/tiobe-index

Nuestro interés en el lenguaje es de desarrollar aplicaciones web, por lo que utilizaremos C# en el contexto de un proyecto de aplicación web ASP.NET Core. Para ello, en este documento comenzaremos por explicar cómo preparar un proyecto para trabajar con nuestros ejercicios sobre el lenguaje C#, y los mecanismos de ASP.NET Core para generar contenido HTML, sin entrar de momento en detalles sobre la plataforma.

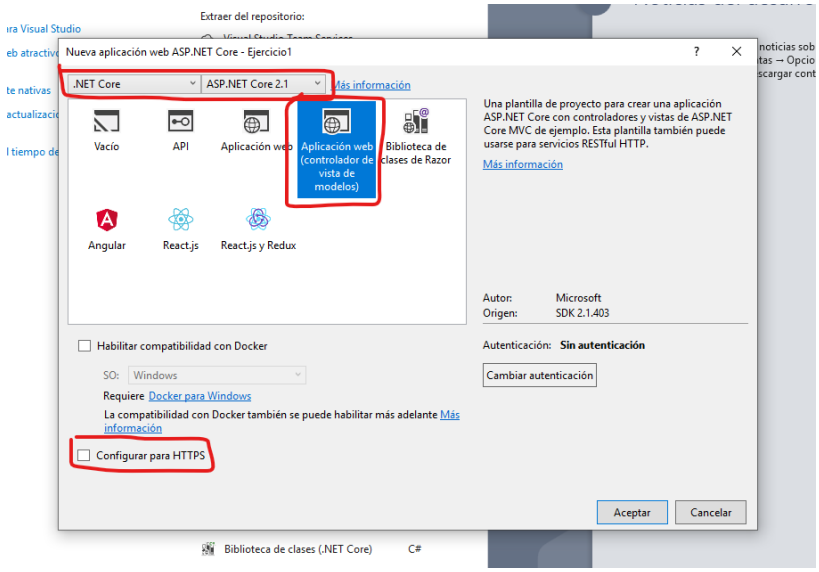
Creación de proyecto de trabajo

Para la creación de un proyecto que nos permita trabajar las características del lenguaje y hacer los ejercicios y tareas de la unidad, comenzaremos creando un nuevo proyecto web, tal y como hicimos en la tarea de la unidad 1.



A continuación, se muestra una nueva ventana para seleccionar una de las plantillas disponibles para aplicaciones web. En nuestro caso es importante que os fijéis en que nuestra aplicación se va a crear para la plataforma .NET Core, y que está seleccionada la última versión de ASP.NET Core, que es la 2.1. Si seleccionamos .NET Framework estaríamos creando una aplicación web en un framework totalmente diferente, que sólo funciona en SO Windows. Por simplicidad también desmarcaremos "Configurar para HTTPS", ya que no es necesario para nuestras tareas.

Por último, y al contrario que para la tarea 1, nos aseguramos de seleccionar la plantilla "Aplicación web (controlador de vista de modelos)" tal y como se indica en la imagen y pulsamos Aceptar.



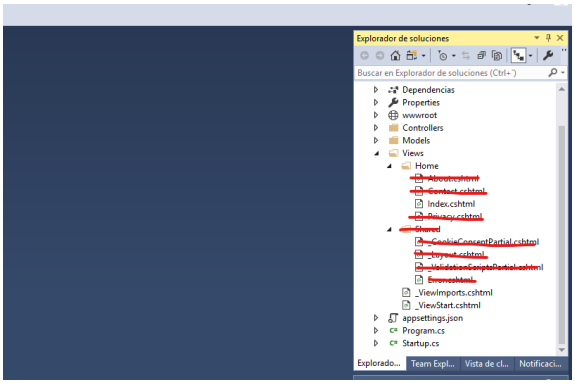
Aunque en la siguiente unidad se profundizará en el tema, os indico que hay dos flujos o modos de desarrollar una aplicación ASP.NET Core: la primera es Razor Pages, que es un modelo basado en páginas y sería la plantilla de aplicación web por defecto y la segunda es la llamada MVC o modelo vista controlador, que es como trabajaremos durante el curso. Existe multitud de documentación al respecto y gran cantidad de discusiones acerca de las ventajas e inconvenientes de un modo u otro de trabajo. Para aplicaciones simples se recomienda el modelo Razor Pages, además de que es el modelo recomendado por Microsoft. En nuestro caso, me he decantado por MVC ya que aún siendo un poco más complejo al principio, tiene la ventaja adicional de que permite tanto desarrollar aplicaciones como servicios web. De este modo sería suficiente para cubrir todo el temario del módulo y no nos obligará a aprender dos marcos de trabajo diferentes, y dado que el tiempo para el módulo es limitado me ha parecido la opción más recomendable.

https://stackify.com/asp-net-razor-pages-vs-mvc

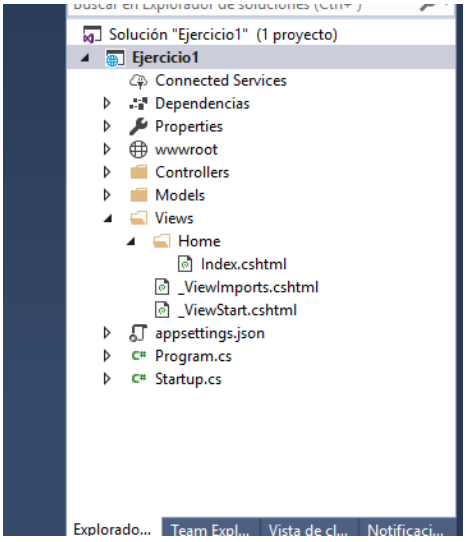
https://jonhilton.net/razor-pages-or-mvc-a-quick-comparison

Preparación del proyecto

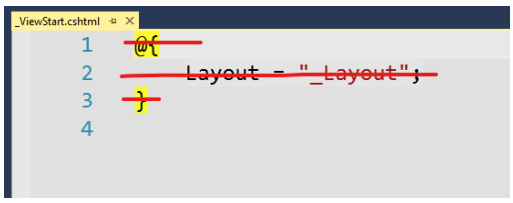
Una vez creado el proyecto, tendremos una aplicación como la obtenida para la tarea 1. Esta plantilla contiene muchos elementos que no necesitamos, por lo que procederemos a eliminar todo lo que no nos hace falta, comenzando por la carpeta Views.

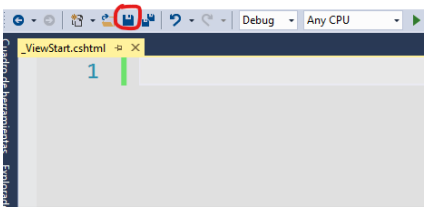


Eliminaremos todos los ficheros excepto Home/Index.cshtml, _ViewImports.cshtml y _ViewStart.cshtml, quedando el proyecto como sigue:



A continuación, eliminamos todo el contenido del fichero _ViewStart.cshtml y lo guardamos.





Hacemos lo mismo con el fichero Views/Home/Index.cshtml.

Y por último, editamos el fichero Controllers/HomeController.cs, dejando únicamente el método Index().

```
 9  namespace Ejercicio1.Controllers
10  {
        0 referencias
11      public class HomeController : Controller
12      {
            0 referencias | 0 solicitudes | 0 excepciones
13          public IActionResult Index()
14          {
15              return View();
16          }
17      }
18  }
19
```

En este momento ya tenemos nuestro proyecto preparado para trabajar con los ejercicios de la unidad. Para comprobar que todo es correcto, ejecutamos nuestra aplicación y nos encontraremos con una página sin contenido en nuestro navegador.

Alguno puede preguntarse el porqué de todo este trabajo cuando tenemos una plantilla de proyecto vacío. El problema es que ASP.NET Core es una plataforma muy flexible pero que requiere que especifiquemos determinadas tareas de configuración en nuestro proyecto. Realizar estas tareas a partir de un proyecto vacío, puede ser más rápido que eliminar los elementos sobrantes, pero requiere un conocimiento más profundo de la plataforma, por lo que dejaremos estas explicaciones para más adelante y trabajaremos de momento de esta forma.

Mecanismos de generación de contenido. Ejemplo inicial.

Antes de pasar al trabajo con C#, vamos a realizar unos pequeños ejemplos para comprobar cómo podemos generar en nuestra aplicación el contenido que será mostrado en nuestro navegador.

Tal y como está configurado nuestro proyecto, al arrancar nuestra aplicación, se comenzará ejecutando el método *Index()* del fichero *HomeController.cs*, así que comenzaremos editando este fichero.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Ejercicio1.Models;

namespace Ejercicio1.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }
    }
}
```

Para los que hayáis trabajado en Java, este código debería resultar sencillo de entender. En C# todo el código debe estar contenido en clases. Un fichero de código en C# tendrá la extensión *.cs*, y aunque no es obligatorio, se recomienda que el nombre coincida con el nombre de la clase que representa, en este caso *HomeController*.

Las sentencias *using* se encargan de importar o incluir clases de otras librerías del lenguaje, y equivalen a los import de java. El *namespace* nos permite empaquetar clases del mismo modo que lo hace package en java.

A continuación vemos que tenemos declarada la clase *HomeController* como pública, y los *:* indican que es nuestra clase va a heredar de la clase padre *Controller.* Por último tenemos un método público *Index()*, que devuelve un objeto de tipo *IActionResult*.

Cuando arrancamos nuestra aplicación, se ejecutará el código contenido en el método *Index()*, y el resultado se mostrará en el navegador.

Comenzaremos nuestros ejemplos mostrando un texto de bienvenida directamente desde el código, para ello editaremos el método *Index()* tal y como sigue:

```
public string Index()
{
    return "Primer ejemplo - DSER";
}
```

Lo que hemos hecho es cambiar el tipo devuelto por una cadena de texto o *string*, y que nuestro método devuelva una cadena de texto. Si ejecutamos nuestro proyecto veremos nuestro mensaje en el navegador. Lo que estamos haciendo en este caso es que nuestra aplicación escriba directamente el resultado de nuestro método como respuesta a la solicitud del navegador. Aunque para este sencillo ejemplo es correcto, nunca se debe generar el contenido de salida directamente desde el código de este modo.

Ejemplo 2.

Deshacemos los cambios realizados en el método *Index()* para dejarlo como estaba inicialmente.

```
public IActionResult Index()
{
    return View();
}
```

En este caso, podemos ver que lo que devuelve nuestro método es el resultado de llamar a otro método, en este caso *View()*. En ASP.NET, este método hace que un controlador devuelva una vista, si no especificamos un nombre, buscará un fichero con el mismo nombre del método que se está ejecutando, en una carpeta del mismo nombre del controlador que lo ejecuta.

Esto parece complejo y lo veremos más adelante en detalle, pero para nuestro caso se traduce en lo siguiente. Nuestra clase es un controlador (hereda de *Controller*) que se llama *HomeController*. ASP.NET se queda con el nombre de nuestra clase sin la parte “Controller”, con lo que buscaremos un fichero en una carpeta *Home*. El método que estamos ejecutando es en

este caso el inicial, llamado *Index()*, por lo que estamos buscando un fichero llamado *Index* en una carpeta Home.

Si nos fijamos, vemos que efectivamente, en la carpeta de vistas (*Views*) tenemos una carpeta *Home*, y dentro un fichero llamado *Index.cshtml*. Este es el fichero que se devolverá al navegador como resultado de la llamada al método *Index()* de nuestro *HomeController*.

Un fichero *cshtml* en realidad no es más que un fichero html que nos permite añadir partes de código ASP.NET incrustado en el html, de un modo similar a un fichero php por ejemplo.

Para nuestro segundo ejemplo editaremos nuestro fichero para incluir código html y que muestre otro mensaje.

```
<html>
<body>
    <h2>Segundo ejemplo - DSER</h2>
</body>
</html>
```

Al ejecutar nuestra aplicación podemos ver como nuestro sencillo código html es devuelto al navegador, mostrando una página con el mensaje.

Ejemplo 3.

Hasta ahora no hemos logrado gran cosa, cualquiera puede darse cuenta de que para devolver una página html estática no necesitamos todo este trabajo. La verdadera utilidad de una aplicación web consiste en la posibilidad de generar código html de forma dinámica, de modo que podamos incrustar el resultado de sentencias escritas en un lenguaje de programación, dentro de una plantilla con código html estático.

Vamos a ver ahora como podemos nos permite ASP.NET Core realizar esta incrustación de código para generar un contenido html de forma dinámica.

Para ello, utilizaremos el símbolo @ para indicar que lo que viene a continuación no es código html ni texto, si no que lo que queremos es insertar un fragmento de código dentro del html.

Podemos insertar una única línea de código a continuación de una @, o incluso un fragmento completo de código, encerrado entre bloques. Vamos a verlo mediante un ejemplo. Editamos nuestro código para dejarlo como sigue:

```
@{
    string message;
    message = "Tercer ejemplo - DSER";
}

<html>
<body>
    <h2>@message</h2>
</body>
</html>
```

En este ejemplo hemos insertado un primer bloque de código que declara e inicializa una variable de tipo string llamada message, con el mensaje que queremos mostrar. Hemos declarado la variable en una sentencia y guardado su valor en otra para ilustrar cómo es posible incluir un bloque completo de código entre llaves.

A continuación, dentro de las etiquetas *h2*, insertamos el código *@message*. La @ indica que lo que viene a continuación es código C# que hay que ejecutar, como en este caso no tenemos una instrucción sino únicamente el nombre de una variable, se obtiene el contenido de esta variable y el resultado es incrustado de forma dinámica en el html que será devuelto al navegador.

Ejemplo 4.

Si bien ya somos capaces de incrustar partes generadas dinámicamente en nuestro html, la forma en la que lo hemos hecho no es la forma recomendada. Los frameworks modernos buscan que las funciones de los distintos elementos de una aplicación web estén bien definidas, separando las responsabilidades de cada uno, de modo que se simplifica cada uno de los elementos de nuestra aplicación, permite la separación de tareas entre los miembros del equipo y facilita el desarrollo y la prueba sobre nuestro código.

Concretamente, el patrón MVC busca esta separación de funciones dividiendo los componentes de nuestra aplicación en modelo, vista y controlador. En la siguiente unidad profundizaremos sobre estos conceptos, pero de momento diremos de forma sencilla que cuando se realiza una petición a nuestra aplicación, esta será atendida por un controlador, que realizará las tareas necesarias, y una vez llevadas a cabo estas tareas seleccionará una vista (en nuestro caso una página cshtml) para mostrar el resultado.

Para que lo entendamos de forma simple, lo que buscamos es separar el código de la aplicación, de lo que se muestra en la pantalla. De este modo una regla fundamental es que no se debe incluir ningún código en nuestra vista salvo el estrictamente necesario para ser mostrado al usuario.

¿Cómo se traduciría esto en nuestro sencillo ejemplo?, muy fácil, nuestro código genera un mensaje que es guardado en una variable, y este mensaje es mostrado en el html resultante. Lo que debemos hacer es generar el mensaje en el código del controlador. Empezamos moviendo ese código al controlador.

```
public IActionResult Index()
{
    string message = "Tercer ejemplo - DSER";

    return View();
}
```

Si hacemos esto nos encontraremos con una desagradable sorpresa, y es que obtendremos un error en el fichero *Index.cshtml*. Esto se debe a que declaramos una variable en nuestro

controlador y pretendemos mostrar el resultado en la vista, pero la vista no conoce esa variable, ya que ha sido declarada en otra parte.

Como veremos, ASP.NET Core dispone de varios mecanismos que le permiten a un controlador comunicar o pasar información a las vistas, pero de momento vamos a ver el más simple, que es el objeto *ViewData*.

ViewData es una colección de objetos, una especie de "bolsa" que nos permite guardar cualquier cosa en ella y que estará disponible para que nuestras vistas puedan coger estos objetos de dicha "bolsa".

```
public IActionResult Index()
{
    string message = "Cuarto ejemplo - DSER";
    ViewData["Message"] = message;

    return View();
}
```

Como queremos que nuestro mensaje esté disponible para ser mostrado al usuario desde una vista, una vez asignada la variable, guardamos su contenido en el *ViewData*. Como en *ViewData* se pueden almacenar tantos objetos como deseemos, tenemos que guardarlos con una etiqueta, de modo que luego podamos sacar el objeto correspondiente haciendo referencia a dicha etiqueta. En este caso guardamos el mensaje contenido en la variable message en nuestro *ViewData* y lo etiquetamos como "Message".

Ahora pasamos a la vista:

```
<html>
<body>
    <h2>@ViewData["Message"]</h2>
</body>
</html>
```

En este caso, queremos incrustar en nuestro código html de salida un valor generado en el controlador. Para ello lo que hacemos es decirle a ASP.NET que busque dentro del *ViewData* un objeto que hemos guardado con la etiqueta "Message", y que incruste el valor de ese objeto dentro del html. Si ejecutamos nuestra aplicación veremos nuestro mensaje en el navegador.