UD3 ASP.NET Core MVC.

3.2 Aplicaciones web y ASP.NET.

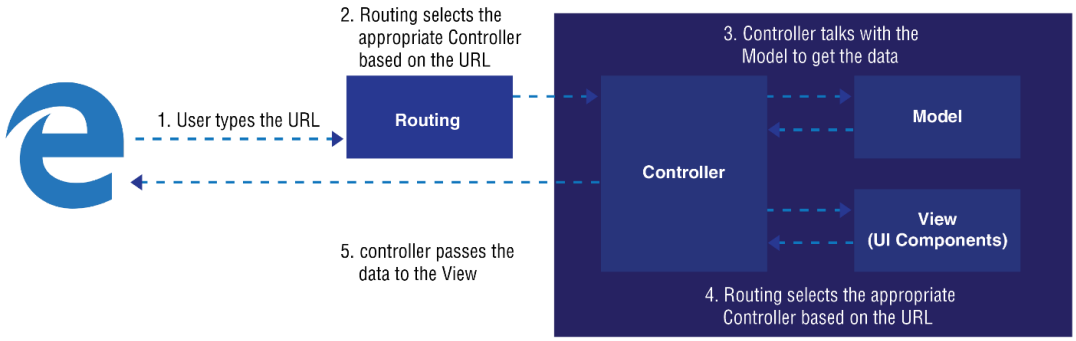
En el apartado anterior, hemos visto como las aplicaciones web reciben peticiones del cliente, produciendo el servidor una respuesta que es enviada de vuelta al cliente. En esta parte de la unidad veremos cuál es la función de los controladores en una aplicación ASP.NET Core MVC y aprenderemos cómo crear controladores y métodos de acción.

Función del controlador en las aplicaciones ASP.NET Core MVC.

La tarea de un controlador ASP.NET Core MVC consiste en recibir una petición del cliente y producir una salida basada en los datos de entrada. Podríamos imaginarnos los controladores como puntos de entrada a nuestra aplicación. Los controladores no deben contener lógica de negocio en su código, en su lugar, los controladores consultarán al modelo, que es el lugar donde se implementará toda la lógica de negocio. De este modo se cumple con la separación de funciones y se favorece la reutilización, al tiempo que se simplifica la verificación del código de cada componente.

Un controlador es capaz de comunicarse con el modelo y la vista, utilizando el modelo para generar los datos de salida, y seleccionando la vista correspondiente a la que pasará los datos obtenidos del modelo, para que estos sean presentados al usuario.

El siguiente diagrama ilustra el proceso que se sigue desde que el servidor recibe una petición hasta que el resultado es devuelto al cliente y cómo el controlador es el encargado de coordinar el funcionamiento de todos los componentes que intervienen en el proceso:



1. El usuario introduce la URL en el navegador.
2. Basándose en el patrón de la URL, el motor de enrutado selecciona el controlador apropiado que se encargará de aceptar la petición.
3. El controlador ejecuta uno de sus métodos de acción y se comunica con el modelo para obtener los datos necesarios,
4. El controlador selecciona la vista que debe ser devuelta como respuesta y le pasa los datos necesarios para que puedan ser mostrados al usuario. La vista habitualmente será una página HTML.
5. Finalmente, la vista es enviada de vuelta para que sea mostrada en el navegador del usuario.

Ejemplo: Creación de un nuevo controlador ASP.NET Core MVC.

Vamos a seguir trabajando sobre el proyecto de ejemplo del punto 3.1. Se trataba de una aplicación creada desde cero con la plantilla vacía, que mostraba un mensaje *Hello World!*.

Como ya hemos comentado, ASP.NET Core permite la creación de diferentes tipos de aplicaciones. La plantilla vacía no añade soporte a ninguno de estos tipos de aplicaciones por defecto (por algo es una aplicación vacía) así que antes de añadir un controlador, que es un objeto específico de una aplicación MVC, debemos añadir soporte para que la aplicación utilice el patrón MVC. Como recordamos, el código de configuración de la aplicación debe escribirse en la clase *Startup*.

1. Configuramos la aplicación para añadir soporte MVC. En ASP.NET Core está implementado como un servicio que debemos añadir a la aplicación, por lo que comenzaremos por añadir este servicio en el método *ConfigureServices()*.

```
0 referencias | 0 excepciones
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
}
```

2. Aunque hemos añadido el servicio MVC, necesita ser configurado. Como hemos explicado anteriormente, la configuración de los servicios tiene lugar en el método *Configure()* de la clase *Startup*.

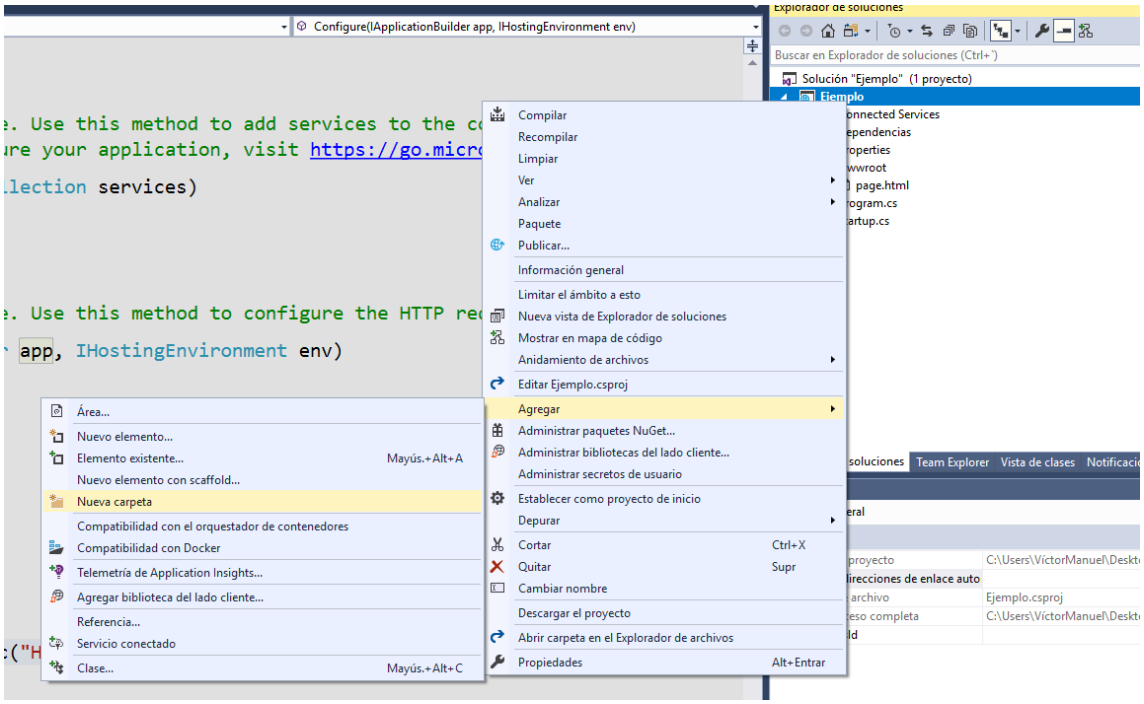
```
    app.UseStaticFiles();
    app.UseMvcWithDefaultRoute();

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

   Hemos configurado el servicio MVC para que utilice la ruta por defecto, de momento dejamos esto de lado y volveremos a explicar en enrutado más adelante, en este mismo apartado.

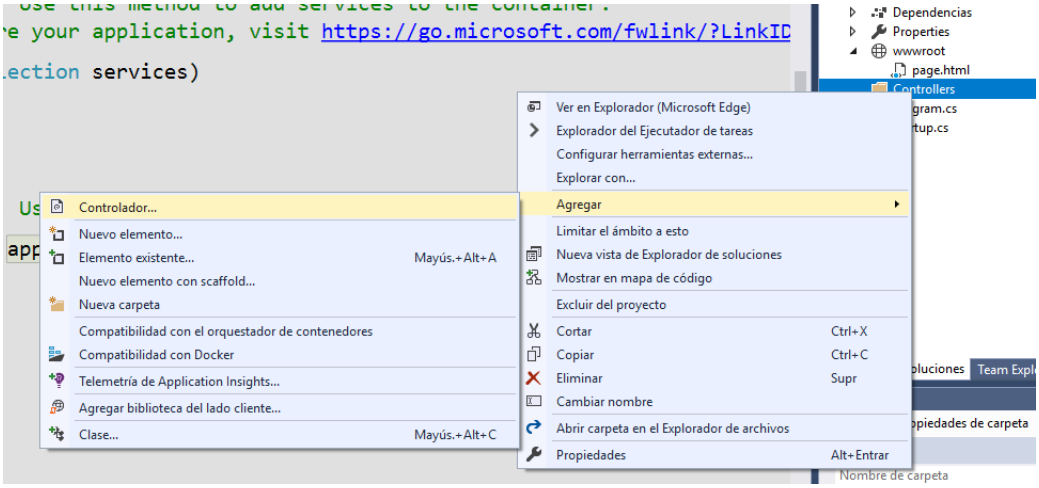
3. En este momento nuestra aplicación captura todas las peticiones respondiendo con el mismo mensaje, como ya pudimos comprobar. Así que antes de crear nuestro controlador eliminaremos el código que captura todas las peticiones.

```
        app.UseStaticFiles();
        app.UseMvcWithDefaultRoute();

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

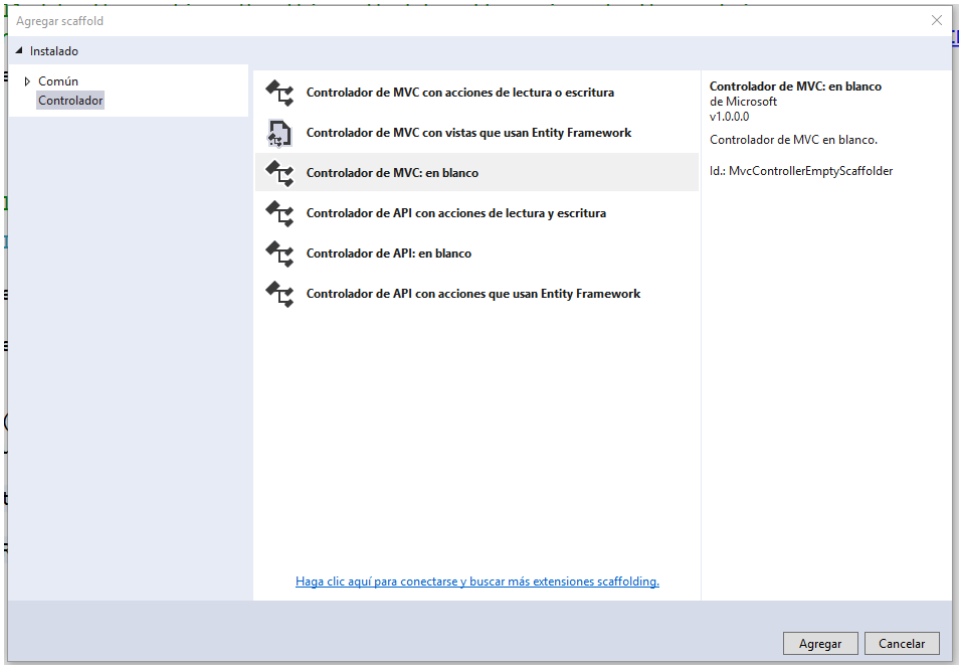
4. Los proyectos ASP.NET Core MVC descansan sobre una serie de convenciones. Como ya hemos explicado, los controladores deben encontrarse en una carpeta de nombre *Controllers*, así que empezamos por crear esa carpeta en el proyecto. Para ello hacemos clic derecho en el proyecto y seleccionamos **Agregar | Nueva carpeta**.



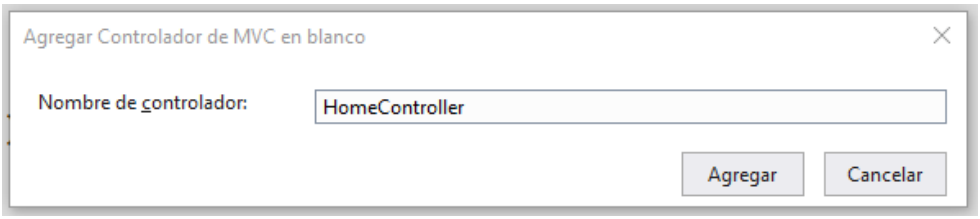
5. Con nuestra carpeta creada, ya podemos añadir el nuevo controlador, haciendo clic derecho en la carpeta *Controllers* y seleccionando **Agregar | Controlador…**



6. En la siguiente ventana seleccionaremos **Controlador de MVC: en blanco** y pulsamos agregar.

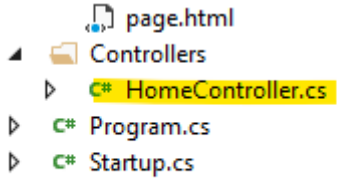


7. Editaremos el nombre del controlador, al que llamaremos *HomeController*. Si antes hablamos de una de las convenciones, que indica que los controladores deben estar situados en una carpeta Controllers, en este caso nos encontramos con otra convención, que esta vez tiene que ver con el nombre de los elementos. En ASP.NET Core MVC todos los controladores son clases C# cuyo nombre debe terminar en la palabra Controller. De este modo el framework puede identificar fácilmente los controladores de nuestra aplicación.



Después de pulsar agregar tarda un rato y entra en funcionamiento el scafolding, aquí explicacmos que es y finalmente obtenemos el controlador y el código.

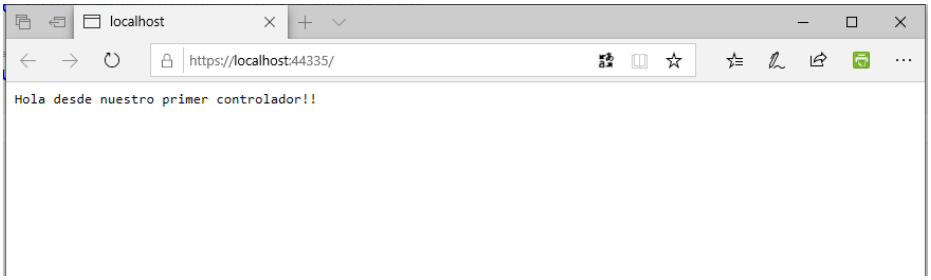
8. En este momento tenemos nuestro controlador creado correctamente en la carpeta *Controllers*, y siguiendo las convenciones su nombre es correcto también, al ser éste *HomeController*, ASP.NET Core MVC lo reconocerá como el controlador *Home*.



9. Con nuestra aplicación configurada para utilizar MVC, y con nuestro controlador añadido correctamente, sólo nos queda editar el código del controlador para que devuelva un mensaje.

```
using Microsoft.AspNetCore.Mvc;

namespace Ejemplo.Controllers
{
    0 referencias
    public class HomeController : Controller
    {
        0 referencias | 0 solicitudes | 0 excepciones
        public string Index()
        {
            return "Hola desde nuestro primer controlador!!";
        }
    }
}
```

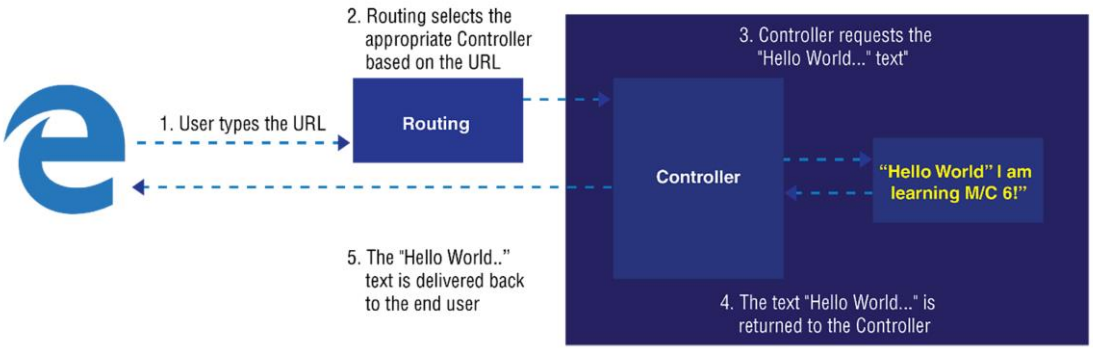
10. Si ejecutamos ahora nuestra aplicación, obtendremos el resultado esperado, con lo que habremos conseguido hacer funcionar nuestro primer controlador.



Como resumen, lo que hemos hecho hasta ahora, a partir de un proyecto vacío es configurar la aplicación para utilizar MVC, añadiendo y configurando el servicio. Hemos eliminado también el código que interceptaba todas las peticiones y mostraba un mensaje de hola mundo. Por último hemos añadido un controlador para que se encargue de recoger la petición del cliente y lo modificamos para que devuelva directamente una cadena de texto como resultado.

No parece gran cosa, pero hay una diferencia fundamental, y es que en este momento estamos utilizando una acción de un controlador para responder a una petición, y por lo tanto estamos listos para añadir tantas acciones y controladores como necesitemos.

A continuación se muestra un esquema de cómo sería la secuencia de acciones que tienen lugar cuando ejecutamos nuestra aplicación (el diagrama está recogido de uno de los libros citados en la bibliografía, por lo que el texto no es el mismo que el de nuestro controlador, aunque si lo es el proceso que tiene lugar en nuestra aplicación cuando una petición es atendida).



Vamos a explicar la secuencia de pasos uno a uno:

1. La aplicación se ejecuta utilizando la URL https://localhost:44335, siendo 44335 el puerto utilizado por el servidor web para escuchar nuestra aplicación, y por lo tanto puede variar en cada proyecto.
2. El motor de enrutado selecciona el controlador que debe responder a la petición y la acción dentro del controlador. En nuestro caso, no hemos especificado ningún controlador ni ninguna acción, y es el mecanismo por defecto el que ha seleccionado el controlador *Home* y la acción *Index*. Más adelante explicaremos cómo funciona el mecanismo de enrutado, que es fundamental para el correcto funcionamiento de la aplicación.
3. Una vez seleccionado el controlador y la acción correspondientes, se crea una instancia de dicho controlador, y a continuación se ejecuta el método que corresponde a la acción. En nuestro caso será el método *Index()* del *HomeController*.
4. En este método, devolvemos como resultado una cadena de texto, que es el valor pasado de vuelta al cliente y mostrado en el navegador.

IActionResult.

En nuestro ejemplo anterior, modificamos el método de acción *Index()* para devolver un mensaje de texto. Originalmente el método devolvía un *IActionResult*, y nosotros lo cambiamos por un string para poder devolver una cadena.

IActionResult es una interface que permite a un método de acción devolver cualquier tipo de dato, desde una simple cadena de texto, un stream de bytes binario, o una compleja colección de objetos formateados como JSON. ASP.NET Core proporciona una serie de objetos *ActionResult* que implementan esta interface y que permiten devolver toda esta variedad de tipos de información.

De este modo, no resulta necesario cambiar el tipo devuelto, tal y como hicimos en nuestro ejemplo por simplicidad. Vamos a modificar de nuevo este método para que siga devolviendo nuestra cadena de texto, pero encapsulada dentro de un *IActionResult*. A partir de ahora dejaremos siempre este tipo de datos devuelto en nuestros métodos de acción, ya que es la forma recomendada de devolver el resultado desde nuestros controladores.

```
public class HomeController : Controller
{
    0 referencias | 0 solicitudes | 0 excepciones
    public IActionResult Index()
    {
        return Content("Hola desde nuestro primer controlador!!");
    }
}
```

En este caso, para poder devolver una cadena como resultado de nuestra acción, utilizamos el método *Content()*, que está definido en la clase base *Controller* (recordemos que nuestra clase *HomeController*, como el resto de controladores que creemos, hereda de *Controller*, y por lo tanto tiene acceso a todos los métodos definidos en ésta). El objetivo del método *Content()* es convertir la cadena de texto que pasamos como parámetro a un objeto que implementa la interface *IActionResult*.

A continuación enumeramos algunos de los tipos de objetos ActionResult disponibles en ASP.NET Core:

1. **ContentResult**: Permite devolver una cadena de texto como resultado.
2. **EmptyResult**: Devuelve un resultado nulo.
3. **FileResult**: Permite devolver un fichero binario en la respuesta.
4. **StatusCodeResult**: Devuelve el código de estado especificado.
5. **JsonResult**: Devuelve un objeto serializado JSON.
6. **RedirectResult**: Redirecciona el resultado a otro método de acción.

En el siguiente enlace se puede consultar más información sobre los ActionResult disponibles en ASP.NET Core:

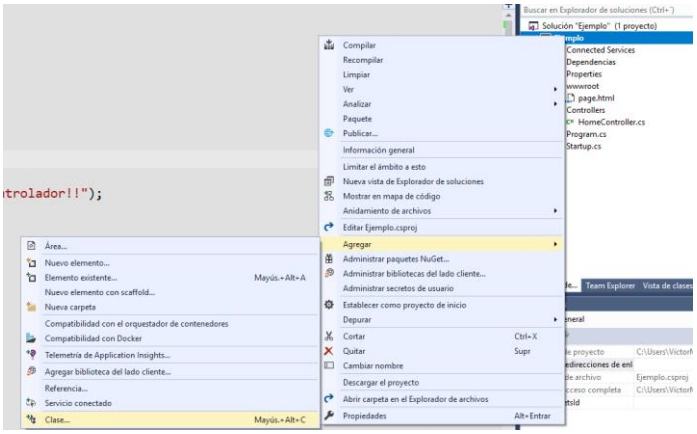
http://hamidmosalla.com/2017/03/29/asp-net-core-action-results-explained

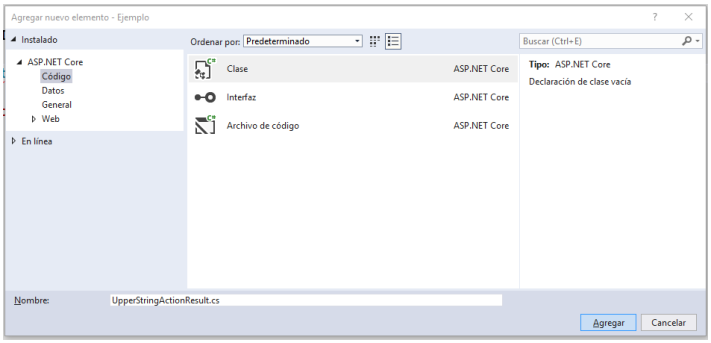
Ejemplo: Implementar un IActionResult personalizado.

Además de todos los tipos de ActionResult disponibles, podemos crear nuestros propios *ActionResult*, en caso de que ninguno se adapte a nuestras necesidades. Para ello tenemos dos opciones, podemos crear una clase que implemente la interface *IActionResult*, o bien podemos aprovechar la clase base *ActionResult* de ASP.NET Core, creando una clase que herede de esta. Esta segunda forma es más sencilla y es como lo haremos en el ejemplo.

A continuación vamos a crear un *ActionResult* que devuelva la cadena de texto pasada como argumento, después de haberla pasado a mayúsculas.

1. Añadimos una nueva clase a nuestro proyecto, a la que llamaremos *UpperStringActionResult*.



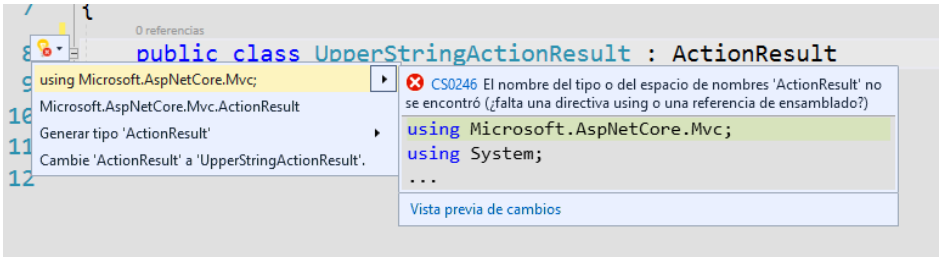


2. Ahora editamos el código de la clase para que nuestra clase herede de ActionResult.

```
namespace Ejemplo
{
    0 referencias
    public class UpperStringActionResult : ActionResult
    {
    }
}
```

Al indicar que nuestra clase hereda de *ActionResult* nos encontraremos con que el nombre de esta clase aparece subrayado en rojo y si compilamos el proyecto nos encontraríamos con un error. Esto es debido a que la clase *ActionResult* se encuentra en un espacio de nombres que no hemos importado / estamos usando. Para ello tenemos que añadir la sentencia *using* correspondiente. En estos casos, como es posible que no recordemos en qué namespace se encuentra, lo mejor es dejar que el entorno de desarrollo nos ayude. Para ello, si mantenemos el puntero del ratón sobre la palabra subrayada, o bien con el cursor en la misma pulsamos Ctrl + Enter, se mostrará un pequeño menú emergente que nos ofrece una serie de recomendaciones para resolver el problema que genera el error. Esto es muy útil en un montón de ocasiones y se puede utilizar siempre que se muestre un error en nuestro código.

En nuestro caso le diremos que queremos añadir la sentencia *using* que nos sugiere y el problema se habrá resuelto.



3. A continuación escribimos el código necesario para que nuestro *ActionResult* formatee el texto recibido convirtiéndolo a mayúsculas.

```
using Microsoft.AspNetCore.Mvc;
using System.Text;

namespace Ejemplo
{
    public class UpperStringActionResult : ActionResult
    {
        private readonly string str;

        public UpperStringActionResult(string str)
        {
            this.str = str;
        }

        public override void ExecuteResult(ActionContext context)
        {
            var upperStringBytes =
            Encoding.UTF8.GetBytes(str.ToUpper());
```

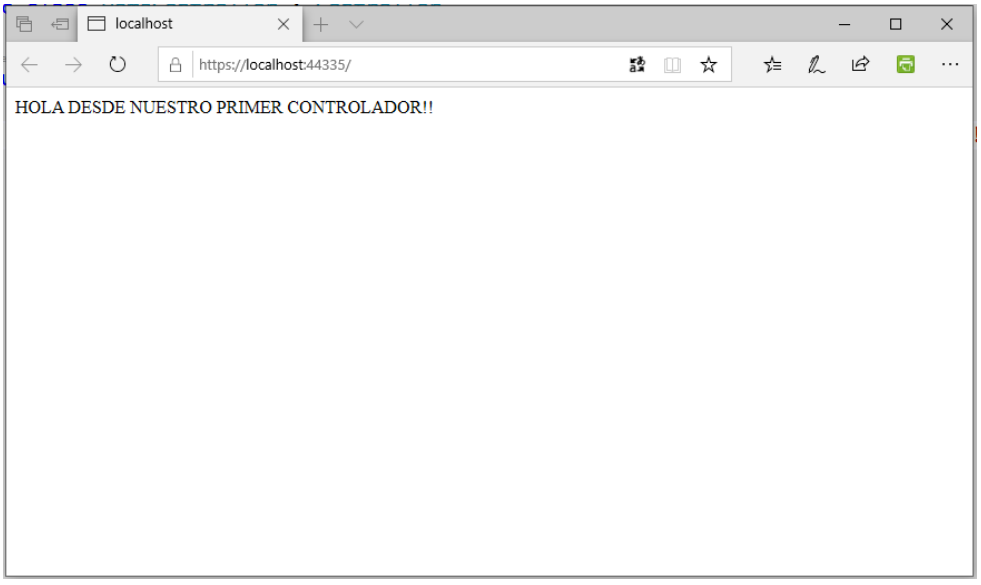
```
                context.HttpContext.Response.Body.Write(
                  upperStringBytes, 0, upperStringBytes.Length);
            }
        }

    }
```

4. Nuestra clase recibe una cadena de texto en el constructor, que almacena en un atributo que será utilizado en el momento de devolver el resultado al cliente. Esto se realiza En el método *ExecuteResult()* del ActionResult. En este caso codificamos nuestra cadena como UTF8, que es el formato de datos que debemos utilizar para la salida y a continuación escribimos el texto en el cuerpo (Body) de la respuesta.

5. Para probar nuestro *ActionResult* personalizado cambiamos el código del método Index del controlador para que lo utilice para formatear la cadena de salida.

```
public IActionResult Index()
{
    return new UpperStringActionResult("Hola desde nuestro primer controlador!!");
}
```

6. Por último ejecutamos nuestro proyecto para ver el resultado.



Introducción al enrutado.

Uno de los componentes fundamentales de ASP.NET Core es el motor de enrutado. Este elemento es el responsable de analizar la petición entrante y enrutar esa petición al controlador apropiado, basándose en el patrón de la URL. Podemos configurar el motor de enrutado para que pueda seleccionar el controlador adecuado. En definitiva, el enrutado es un mapeado que define que método de que controlador será invocado basándose en un patrón URL.

Por convención, ASP.NET Core MVC sigue un patrón como el siguiente: **Controller/Action/Id**. Esto quiere decir que si el usuario introduce la URL http://yourapplication/Hello/Greeting/1, el motor de enrutado seleccionaría *Hello* como controlador, *Greeting* como acción dentro del controlador, y pasaría el valor 1 para el parámetro Id. Siguiendo la convención de nombrado que explicamos anteriormente, ASP.NET sabe que debe buscar, dentro de la carpeta Controllers, una clase de nombre *HelloController* (recordar que se añade 'Controller' al final del nombre de cada controlador), y ejecutar un método público llamado *Greeting()* dentro de dicha clase.

Sin embargo, cuando hemos ejecutado nuestra aplicación de ejemplo hemos visto que se ejecutaba correctamente el código del método *Index()* de nuestro controlador *HomeController*, sin haber indicado el controlador ni la acción en la URL. Esto es posible gracias a los valores por defecto de patrón de enrutado.

Para verlo más claro vamos a Startup.cs y vemos que en la configuración del servicio MVC, hemos indicado que use el mapeado de ruta por defecto:

```
app.UseMvcWithDefaultRoute();
```

Vamos a cambiar este código por otro equivalente, pero utilizando un método que nos permite especificar el mapeado a utilizar:

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

Con este código estamos definiendo un mapeado al que hemos llamado "default" y cuya plantilla es: `{controller=Home}/{action=Index}/{id?}`

Esta es la misma plantilla que utiliza el método UseMvcWithDefaultRoute que teníamos anteriormente. Si observamos la plantilla, vemos que con cada elemento, definimos un valor después de un signo igual. Estos son los valores por defecto que utilizará el motor de enrutado en caso de que no se proporcione ninguno. Por esta razón, cuando ejecutamos nuestro proyecto nuestro código funciona correctamente, a pesar de no haber indicado controlador o acción alguna en la URL. Al no especificarse ninguno seleccionará el controlador por defecto, que es *HomeController*, y la acción por defecto, que es el método *Index()*. En el elemento *id* podemos ver un signo '?', que indica que se trata de un elemento opcional, de modo que en la URL podemos incluir o no un id.

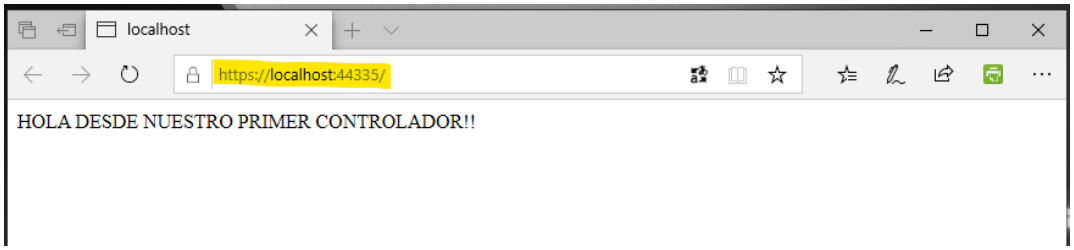
Ejemplo: Prácticas con el motor de enrutado.

Para aclarar el funcionamiento del motor de enrutado analizaremos como se comportaría nuestra aplicación para una serie de URLs, utilizando el mapeado por defecto. En los ejemplos se omiten los números de puerto que utiliza el servidor de desarrollo cuando estamos desarrollando el proyecto, ya que estos varían de un equipo a otro, pero no olvides que debes incluirlo para comprobar los ejemplos en tu aplicación.

URL 1: http://localhost

- **Controlador**: Home.
- **Acción**: Index.
- **Id**: Sin valor.

En este caso, controlador y acción se obtienen de los valores por defecto de la plantilla de mapeado.



URL 2: http://localhost/Hello/Index

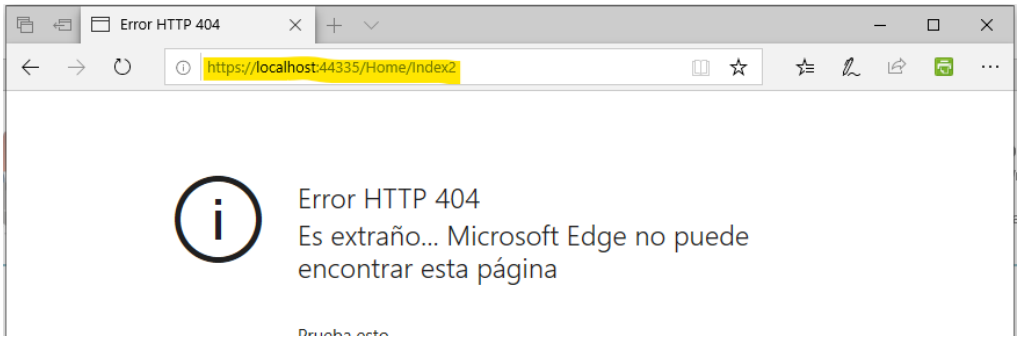
- **Controlador**: Home.
- **Acción**: Index.
- **Id**: Sin valor.

Igual que el caso anterior, excepto por que en esta ocasión si especificamos controlador y acción en la URL.

URL 3: http://localhost/Home/Index2

- **Controlador**: Home.
- **Acción**: Index2.
- **Id**: Sin valor.

De nuevo especificamos controlador y acción en la URL. Es importante entender que los valores por defecto sólo se utilizan cuando no especificamos éstos en la URL, por lo que en este caso el motor de enrutado buscará método de acción *Index2()* que no existe, por lo que obtendremos un error.



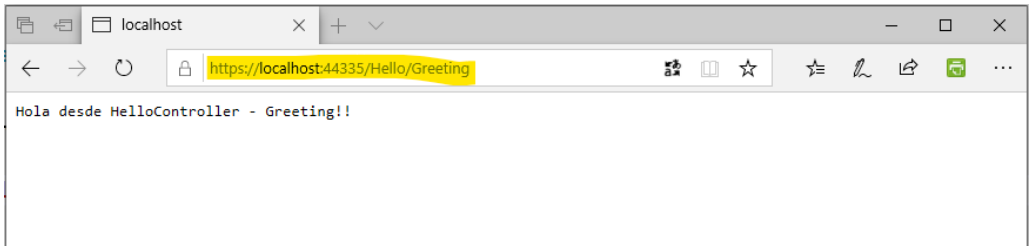
Para solucionarlo, vamos a añadir un nuevo método Index2 en nuestro controlador y volveremos a probar con esta URL:

```
public IActionResult Index()
{
    return new UpperStringActionResult(
         "Hola desde nuestro primer controlador!!");
}

public IActionResult Index2()
{
    return Content("Hola desde HomeController - Index2!!");
}
```

URL 4: http://localhost/Hello/Greeting

- **Controlador**: Hello.
- **Acción**: Greeting.
- **Id**: Sin valor.

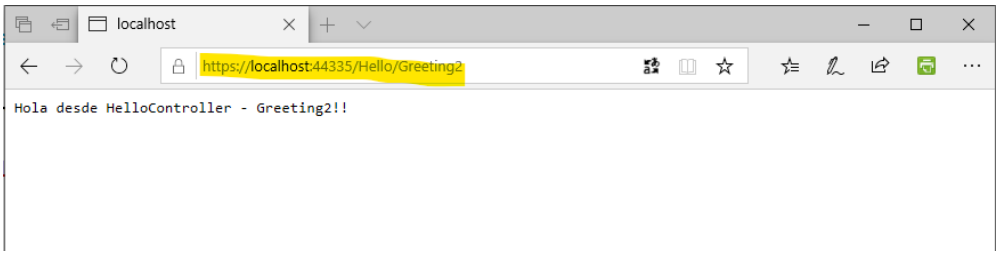
En este caso estamos indicando un controlador diferente, por lo que no bastará con añadir un método a nuestro controlador HomeController. Con lo aprendido hasta ahora deberías ser capaz de crear un nuevo controlador que contenga un método de acción que responda a esta petición correctamente, de modo que te aconsejo que intentes hacerlo como ejercicio.



URL 5: http://localhost/Hello/Greeting2

- **Controlador**: Hello.
- **Acción**: Greeting2.
- **Id**: Sin valor.

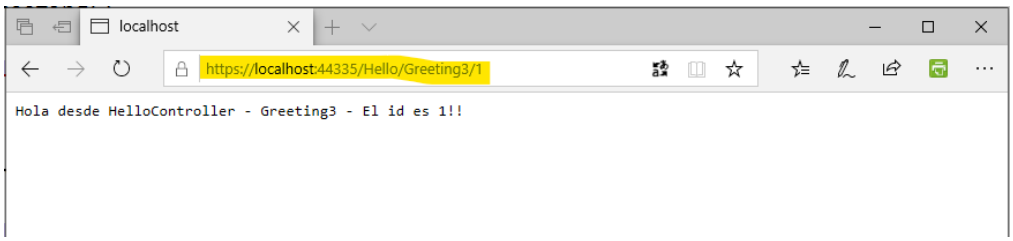
Como ejercicio intenta que tu aplicación responda a esta URL y que el resultado sea similar al siguiente:



URL 6: http://localhost/Hello/Greeting3/1

- **Controlador**: Hello.
- **Acción**: Greeting2.
- **Id**: 1.

Como ejercicio intenta que tu aplicación responda a esta URL y que el resultado sea similar al siguiente. Fíjate en que en este caso pasamos un id, por lo que debes declarar el método correspondiente para que reciba un parámetro entero de nombre id, para que reciba este valor de la URL:



Añadir vistas.

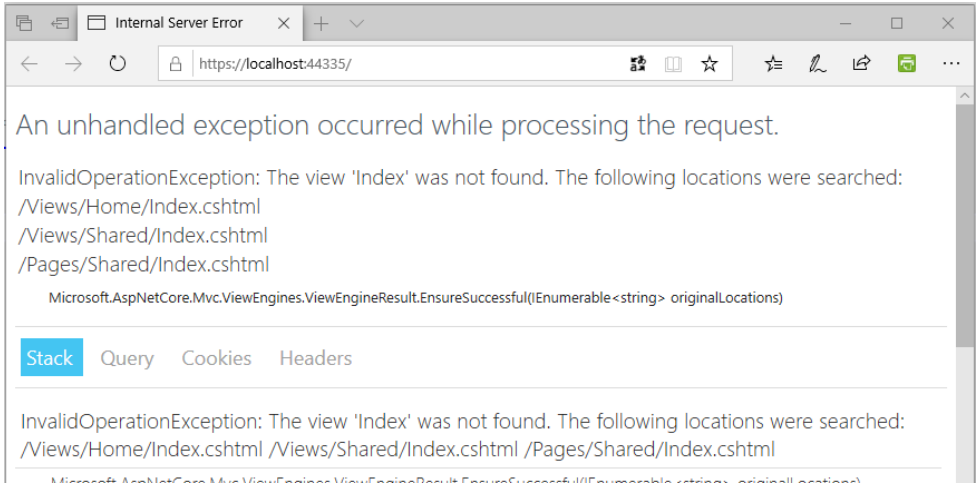
Hasta ahora, las acciones de nuestros controladores se limitan a devolver simples cadenas de texto. Aunque para explicar los conceptos referentes a los controladores y sus métodos de acción es suficiente, no es de mucha utilidad en una aplicación web.

Si recordamos, cuando nuestro controlador fue creado, el código del método Index por defecto era como sigue:

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

Esto será lo más habitual en la mayoría de los casos, y dejar que la respuesta del servidor al cliente descanse sobre una vista, que como ya explicamos es el componente encargado de la presentación de datos al usuario.

Vamos a editar nuestro método *Index* para que quede como inicialmente, devolviendo una vista, y ejecutaremos nuestro proyecto.



Como era de esperar, obtenemos un error, ya que en nuestro proyecto no tenemos todavía ninguna vista. Pero si nos fijamos bien en el error nos da una serie de pistas interesantes.

En primer lugar vemos que ASP.NET ha intentado buscar una vista de nombre Index, exactamente igual que el nombre de la acción, esta sería la primera convención: El nombre una vista será el mismo que el del método de acción del controlador.

En segundo lugar, vemos que ha buscado la vista en una serie de lugares, de momento nos interesa el primero, ya que el resto ya los explicaremos más adelante. Vemos que ha buscado nuestra vista en una carpeta Home, dentro de otra carpeta *Views*. Al igual que los controladores se almacenan en una carpeta *Controllers*, es lógico que las vistas se guarden en una carpeta *Views*. Como mecanismo para organizar la multitud de vistas que puede haber en una aplicación web, se estableció la convención de que las vistas de almacenan en una carpeta cuyo nombre sea el del controlador, sin el sufijo 'Controller', en nuestro caso esta sería *Home*.

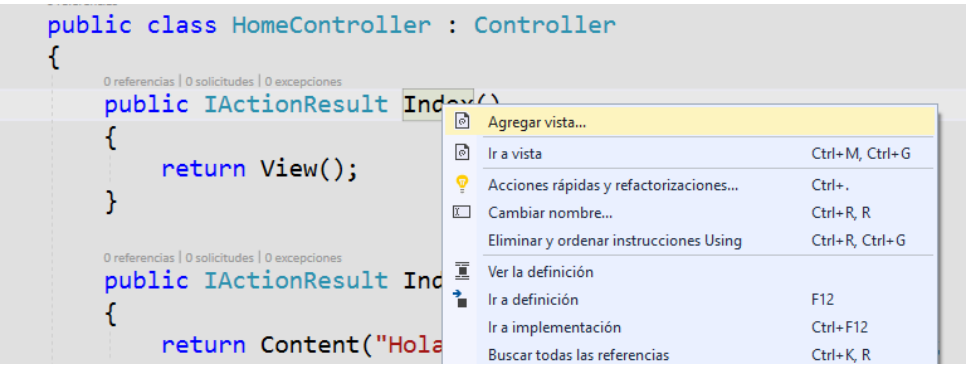
Esta es la razón por la que el método View() no necesita especificar ningún nombre de vista, ya que utilizará las convenciones mencionadas para buscar la vista correspondiente a la acción en

la que se ejecuta. Sin embargo el método si permite especificar un nombre específico de vista, aunque no se recomienda salvo es casos muy concretos, que veremos más adelante. En la mayor parte de las ocasiones la recomendación es descansar en las convenciones de nombrado y utilizar el método sin parámetros.

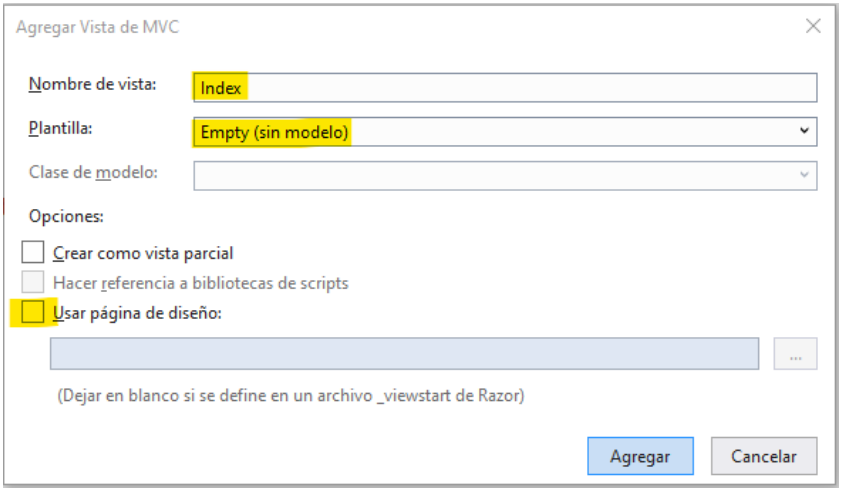
Ejemplo: Añadir una vista.

Queremos añadir una vista para el método de acción *Index* del controlador *Home*. Esto puede hacerse de varias formas, pero personalmente, como soy un poco vago, prefiero dejar que Visual Studio me ayude un poco y se encargue de las tareas más aburidas, así que vamos a ver como hacerlo del modo más simple en mi opinión.

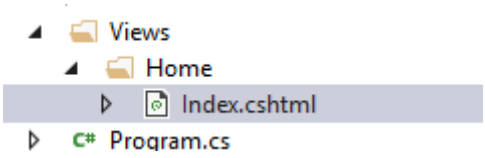
1. Abrimos el código de nuestro controlador y hacemos clic derecho dentro de los límites del método Index. Esto último es muy importante, porque Visual Studio detectará el método para el que queremos crear la vista y mostrará la opción en el menú contextual que si pinchamos en otro sitio no aparecería, o se referiría a otra acción. En el menú pinchamos, lógicamente, en **Agregar vista…**



2. En la ventana que aparece a continuación, vemos como el nombre que sugiere para la vista, es el mismo que el del método, lo que nos da una pista de que lo hemos hecho bien. Dejamos ese nombre, siguiendo las convenciones y escogemos la plantilla vacía. Desmarcamos también la opción página de diseño, que explicaremos en el apartado relativo a las vistas.

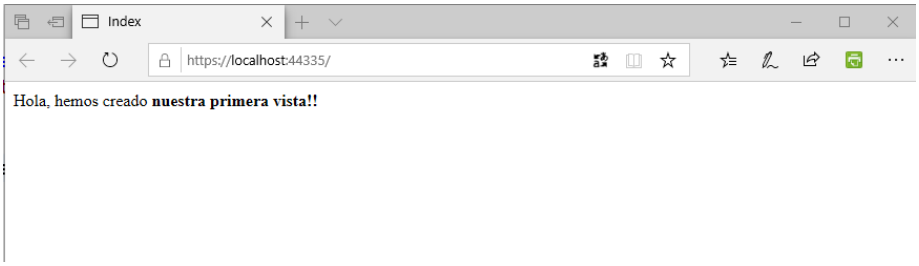


3. Cuando aceptemos, y después de unos instantes, habrá creado la vista, que tendremos abierta en el editor, pero lo más interesante es que si nos fijamos en el explorador de soluciones, veremos que ha creado por nosotros la estructura de carpetas siguiendo las convenciones, colocando nuestra vista en *Views/Home*.

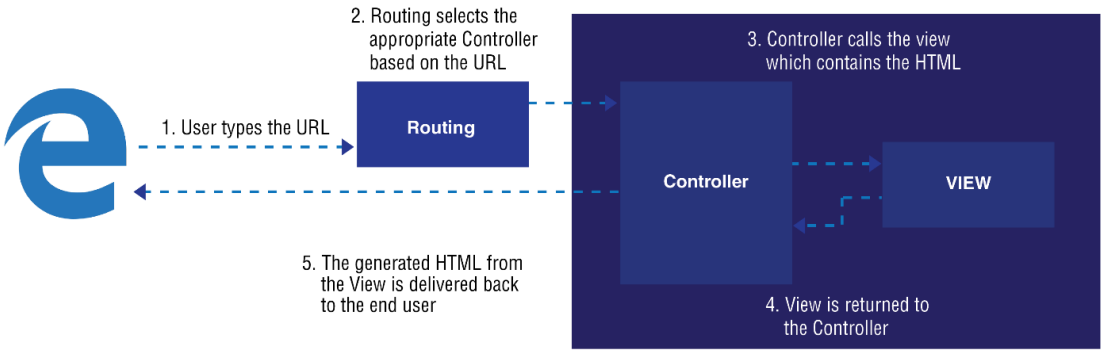


4. Editamos nuestra vista para que muestre un mensaje y ejecutamos el proyecto.

```
<body>
    Hola, hemos creado <b>nuestra primera vista!!</b>
</body>
```



El siguiente diagrama muestra el flujo de la petición y cómo es generada la respuesta utilizando la vista:



Añadir modelos.

Los modelos representan clases del dominio de negocio, es decir, modelan elementos del mundo real con los que nuestra aplicación trabajará. A continuación vamos a ver cómo trabajar con los modelos, creando una clase *Student* que contendrá los datos de un estudiante.

Comenzamos creando una carpeta donde guardar los modelos en nuestro proyecto, si los controladores están en *Controllers*, las vistas en *Views*, los modelos estarán.. efectivamente, los modelos estarán en la correspondiente carpeta *Models*.

Una vez creada la carpeta, agregamos dentro una nueva clase *Student.cs* y a continuación la editamos para que contenga la información relevante:

```
public class Student
{
    public int Id { get; set; }
    public string Nombre { get; set; }
    public string Apellidos { get; set; }
}
```

Es una clase muy sencilla, que únicamente sirve para almacenar información sobre nuestros alumnos, en este caso nombre y apellidos. Si alguno no entiende el código, os aconsejo repasar la unidad 2, en concreto en la parte que hablamos de las propiedades en C#. En este caso tenemos 3 propiedades autoimplementadas, por lo tanto tendremos 3 atributos que el compilador genera automáticamente, y las 3 son de lectura y escritura.

A continuación vamos a añadir un nuevo controlador llamado *Students*, ya que es un controlador que se va a encargar de ofrecer acciones relacionadas con los estudiantes. Una vez creado cambiaremos el nombre del método *Index* por *Student*, en una aplicación real utilizaríamos este método para obtener los datos de un estudiante en concreto, pasándole la id del estudiante que queremos ver. En este caso al ser un ejemplo devolveremos siempre el mismo estudiante. Para ello es código quedará como sigue:

```
using Ejemplo.Models;
using Microsoft.AspNetCore.Mvc;

namespace Ejemplo.Controllers
{
    public class StudentsController : Controller
    {
        public IActionResult Student()
        {
            Student student = new Student
            {
                Id = 1,
                Nombre = "Jean-Luc",
                Apellidos = " Picard"
            };

            return View();
        }
    }
}
```

Lo que estamos haciendo en nuestro método es simplemente crear un nuevo objeto de tipo Student. En C# se permite inicializar un objeto al mismo tiempo que es creado, sin necesidad de utilizar un constructor que reciba los datos como parámetros. Para ello, entre bloques, vamos asignando valores a las propiedades que nos interese, de modo que el objeto será creado y a continuación se irá pasando los valores a estas propiedades. Es equivalente a una tarea habitual como es crear un objeto y establecer su estado pasando valores a sus campos. Este código sería equivalente a hacer en java algo como:

```
Student student = new Student();
Student.SetId(1);
Student.SetNombre("Jean-Luc");
Student.SetApellidos(" Picard"};
```

Ahora añadimos una vista para nuestra acción, del mismo modo que hicimos antes. Con el puntero dentro del método *Student*, hacemos clic derecho y pinchamos en **Agregar vista…**

Ya tenemos nuestro modelo, un controlador que hace uso del mismo para obtener los datos a ser devueltos al cliente y una vista para mostrar los datos al usuario. Sólo nos falta el último ingrediente, saber cómo pasar estos datos desde el controlador a la vista para que pueda incrustarlos en el código HTML.

Comunicación entre el controlador y la vista.

Hasta ahora habíamos utilizado el objeto ViewData como mecanismo de paso de datos a nuestras vistas. Este método, sin embargo, tiene varias desventajas y no es el mecanismo recomendado para el paso de los datos generados desde el controlador.

Una de las mayores ventajas de ASP.NET es el uso de vistas tipadas, o lo que es lo mismo, podemos determinar en la página qué tipo de datos esperamos recibir, de modo que luego podemos insertar la información del modelo de un modo tipado, es decir, tendremos asistencia de Intellisense, que nos facilitará la edición de código, ya que se mostrará información de las propiedades del objeto al escribir el operador '.', además que permitirá detectar errores antes de la ejecución, y evitará tener que hacer castings para convertir a un tipo de datos concreto, como ocurre al usar ViewData.

Vamos a ilustrarlo continuando con nuestro ejemplo. Volvemos a nuestro controlador y vamos a pasar el objeto generado como parámetro del método View, como se muestra a continuación:

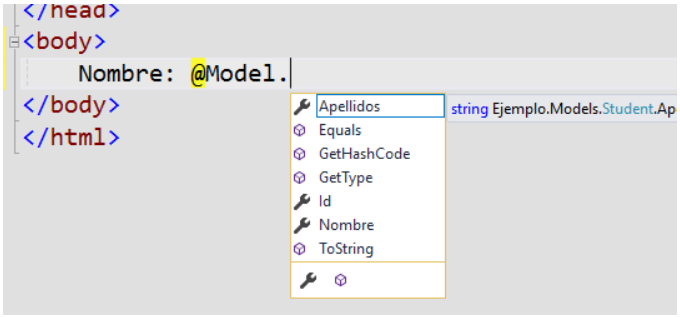
```
return View(student);
```

De este modo lo que hacemos es seleccionar la vista que se va a devolver al cliente, pero además le pasamos el modelo de datos para que la vista pueda formatearlo y mostrarlo al usuario.

Ahora pasamos a la vista, en este caso vamos a indicarle a la página que va a recibir desde el controlador un objeto de tipo *Student*. De este modo obtenemos todas las ventajas comentadas hace un momento. Esto se hace con la instrucción *model*. Añadimos la siguiente línea al inicio de la página:

```
@model Ejemplo.Models.Student
```

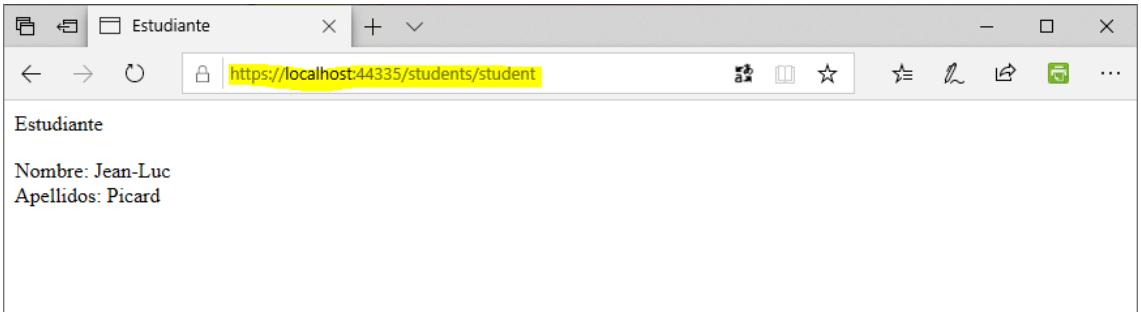
Ahora vamos a mostrar la información del usuario en el cuerpo de nuestra página. En el lugar en que queramos mostrar algún dato contenido en el modelo, utilizaremos el alias *Model*, que se es una referencia al objeto que la página ha recibido. Como podremos comprobar, al escribir el operador '.', se nos ofrecerán las propiedades accesibles de nuestro modelo. Esto es posible porque nuestra página conoce el tipo de dato que está recibiendo, gracias a la instrucción @model que agregamos con anterioridad:



Terminamos de editar el código de la vista para que se muestren nombre y apellidos, quedando la vista como sigue:

```
@model Ejemplo.Models.Student
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Estudiante</title>
</head>
<body>
    Estudiante<br/><br/>
    Nombre: @Model.Nombre<br/>
    Apellidos: @Model.Apellidos
</body>
</html>
```

Ahora ejecutamos el proyecto y especificamos en la URL *students/student*.



Si en la vista intentáramos mostrar información no existente, como @Model.Age, obtendríamos un error de compilación. Esto supone una gran ventaja ya que permite solucionar el problema antes de que ocurra en tiempo de ejecución, siendo mucho más complicado de localizar.

ViewData (ViewBag).

El paso de datos desde el controlador a la vista utilizando el objeto *Model* es el modo recomendable y habitual de realizar esta tarea. Sin embargo, esto no quiere decir que dejemos de utilizar a partir de ahora el objeto ViewData. Hay ocasiones en las que además del modelo principal, puede que queramos añadir información adicional a la vista, que bien no pertenece al modelo, o por que resulta más cómodo hacerlo de este modo. Suele utilizarse con tipos de datos simples, especialmente cadenas de texto, que no haya que parsear para ser mostrados en la vista.

Ya explicamos en la anterior unidad que el objeto *ViewData* es un diccionario de datos que almacena pares de datos, la clave con la que se etiquetan y el propio valor. Por comodidad existe otro objeto llamado *ViewBag*, que utiliza una representación dinámica de los mismos valores. Es exactamente lo mismo utilizar uno u otro objeto, lo único que varía es la sintaxis. Por ejemplo, para referirnos a un objeto etiquetado como 'texto' usando ViewData tendríamos que escribir *ViewData["texto"]*, mientras que el equivalente con ViewBag sería *ViewBag.Texto.*

Vamos a agregar código para pasar a la vista información sobre el ciclo y el centro del estudiante.

```
public IActionResult Student()
{
    Student student = new Student
    {
        Id = 1,
        Nombre = "Jean-Luc",
        Apellidos = " Picard"
    };

    ViewData["ciclo"] = "Desarrollo de aplicaciones web";
    ViewBag.Centro = "CIFP A Carballeira";

    return View(student);
}
```

Y a continuación vamos a mostrar esta información en nuestra página. Hemos declarado cada uno de los datos usando un objeto, ViewData y ViewBag. En la vista los utilizaremos del modo contrario, es decir, usaremos ViewBag para el elemento definido con ViewData y al revés con el declarado con ViewBag. De este modo quedará claro que el uso de uno u otro es indistinto, ya que internamente se refieren a la misma colección de objetos.

```
<body>
    Estudiante<br/><br />
    Nombre: @Model.Nombre<br />
    Apellidos: @Model.Apellidos<br />
    Ciclo: @ViewBag.Ciclo<br />
    Centro: @ViewData["centro"]
</body>
```

Filtros.

Los filtros en ASP.NET Core MVC permiten ejecutar código, antes o después de cada una de las etapas que se llevan a cabo desde que se recibe una petición hasta que la respuesta es enviada al cliente. Pueden configurarse de modo global, por controlador o por acción. Se pueden considerar como una serie de "interceptores".

Hay varios tipos diferentes de filtros, a continuación se enumeran algunos de los tipos predefinidos de filtros disponibles en ASP.NET Core:

- **Filtros de autorización**: Usados para la autorización del usuario, de modo que sea posible determinar si el usuario actual está autorizado para la petición realizada.
- **Filtros de recursos**: Trabajan sobre la petición justo antes de ser procesadas por los controladores. Permiten por ejemplo implementar cachés de resultados, etc..
- **Filtros de acción**: Trabajan por encima de las llamadas a un método de acción, pudiendo manipular tanto los argumentos que se pasan a la acción como el resultado devuelto por ésta.
- **Filtros de excepción**: Son utilizados para gestionar excepciones sin manejar.
- **Filtros de resultado**: Empaquetan los resultados devueltos por las acciones y sólo son ejecutados cuando el método de acción ha sido ejecutado con éxito.

Ejemplo: Aplicación de un filtro.

En este ejemplo veremos cómo utilizar un filtro de acción (uno de los tipos de filtro vistos) y ver cómo funciona.

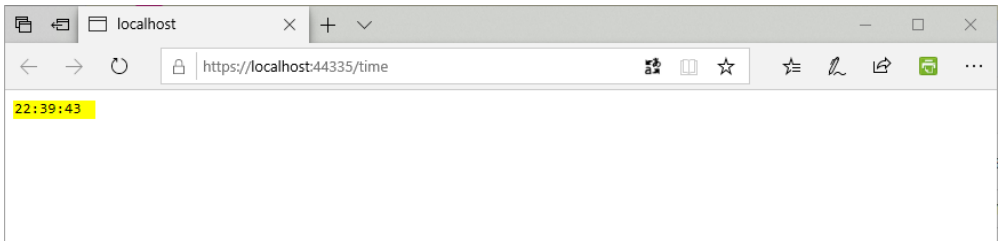
1. Comenzamos creando un nuevo controlador, TimeController, cuyo resultado será simplemente mostrar la hora actual. Por simplicidad, el valor será devuelto directamente como una cadena de texto, sin necesidad de crear una vista.

```
using System;
using Microsoft.AspNetCore.Mvc;

namespace Ejemplo.Controllers
{
    public class TimeController : Controller
    {
        public IActionResult Index()
        {
            return Content(DateTime.Now.ToShortTimeString());
        }
    }
}
```

2. A continuación, vamos a utilizar un filtro predeterminado disponible en ASP.NET Core, llamado *ResponseCache*, cuya función es la de almacenar en caché la respuesta por un tiempo especificado en segundos, en nuestro ejemplo 300 segundos, que hará que la duración de la caché sea de 5 minutos. Al tratarse de un filtro de acción, se declara justo antes del comienzo del método que utilizará el filtro, entre corchetes.

```
[ResponseCache(Duration = 600)]
public IActionResult Index()
```

3. Ejecutamos ahora nuestra aplicación, y especificamos el controlador en la URL añadiendo /time (como repaso recordamos que el motor de enrutado añade los valores por defecto cuando no se indica ningún valor en la URL, al no indicar la acción, el motor selecciona automáticamente la acción por defecto, que no es otra que *Index*).



4. Si abrimos una nueva pestaña u otro navegador y escribimos la misma URL, veremos como la respuesta es la misma que la anterior, a pesar de que la hora ya no es la misma. Esto es así lógicamente porque al estar el resultado en caché este no se recalcula si no que se reenvía el valor guardado. Ojo, si hacemos F5 forzamos a refrescar, evitando la caché.

Este filtro puede ser muy útil en algunas situaciones, ya que si un resultado que está en caché no tiene que ser calculado de nuevo, haciendo más rápida la respuesta. El mayor problema es que la respuesta podría no estar actualizada, por lo que en algunos casos no es aceptable. Hay que tener muy presente esto antes de decidir establecer una caché para una acción.

Ejemplo: Creación de un filtro personalizado.

Cuando necesitamos alguna funcionalidad que no está disponible por defecto, podemos crear nuestro filtro personalizado para cualquier tipo de tarea. En este ejemplo queremos que una determinada acción pueda ser ejecutada cualquier día menos en domingo (nuestra acción también quiere tomarse un descanso de vez en cuando).

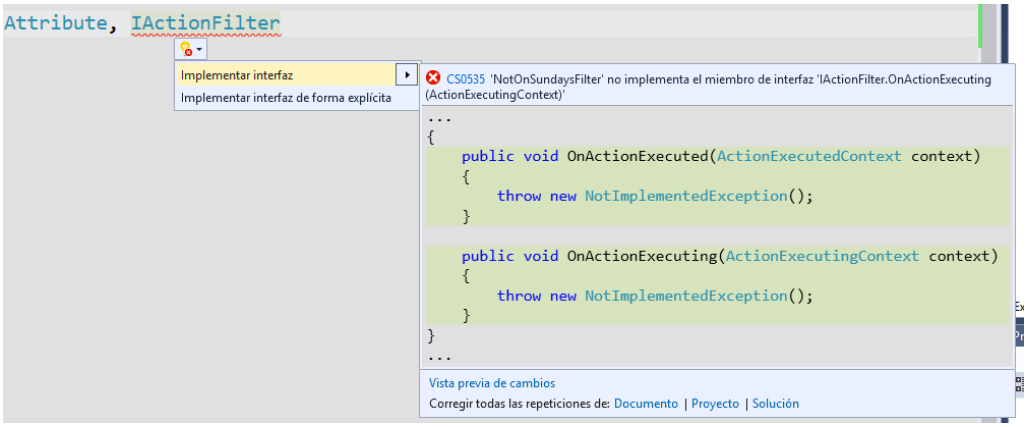
1. En primer lugar agregamos una clase al proyecto, a la que llamaremos *NotOnSundaysFilter*. De nuevo, una convención de nombrado es la de añadir Filter al final del nombre de una clase que actúe como filtro.

2. Para que una clase pueda trabajar como filtro, debe implementar una interface, en nuestro caso queremos un filtro de acción, así que nuestra clase implementará *IActionFilter*. Fijarse de nuevo que en C#, añadiremos una 'I' antes del nombre de una interface, para distinguirlas de una clase normal. Por simplicidad, queremos declarar nuestro filtro delante del nombre de la acción sobre la que se aplicará, utilizando corchetes, del mismo modo que lo hicimos en el ejemplo anterior. Para esto nuestra clase debe heredar de la clase base *Atribute*.

```
public class NotOnSundaysFilter : Attribute, IActionFilter
```

3. Cuando hagamos esto, veremos que tenemos un error en *IActionFilter*. Como ya vimos en ejemplos anteriores, esto es debido a que necesitamos añadir un using con el namespace que contiene esta interface, en este caso Microsoft.AspNetCore.Mvc.Filters, ya vimos como hacer esto de forma simple con la ayuda del entorno.

4. Una vez añadida la sentencia using, vemos que seguimos con un error en *IActionFilter*. En este caso, el problema es bien diferente, cuando una clase implementa una interface, tiene que ofrecer una implementación de todos sus métodos públicos, para así cumplir con el contrato que define esta interface. Sin implementar estos métodos no podríamos decir que nuestra clase es, en este caso, un *IActionFilter*. Una vez más el entorno nos será de gran ayuda, ya que si no conocemos cuáles son estos métodos, tendríamos que buscar la documentación de la interface, y ver cuáles son y sus firmas.
   Podemos evitarnos todo este trabajo, ya que Visual Studio conoce esta información por nosotros, así que lo mejor es dejar que se encargue. Mantenemos el puntero encima del texto que marca el error y se mostrará de nuevo un menú emergente, y en este caso encontraremos la opción **Implementar interfaz**.



5. Ahora ya tenemos en nuestra clase los métodos que tenemos que implementar, aunque el único código en cada método consiste en lanzar una excepción de método no implementado (lógico, bastante ha hecho el editor por nosotros con añadir los cuerpos de los métodos que debemos implementar, no vamos a pretender que el editor sepa también que es lo que queremos hacer..).
   Añadimos el siguiente código, de modo que nuestro filtro quede finalmente como se muestra:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using System;

namespace Ejemplo
{
    public class NotOnSundaysFilter : Attribute, IActionFilter
    {
        public void OnActionExecuted(ActionExecutedContext context) { }

        public void OnActionExecuting(ActionExecutingContext context)
        {
            if (DateTime.Now.DayOfWeek == DayOfWeek.Sunday)
                context.Result = new ContentResult()
                {
                    Content = "Lo siento, no trabajo los domingos!!"
                };
        }
    }

}
```

6. En un filtro de acción podemos realizar tareas antes o después de que la acción se ejecute. En nuestro caso, queremos comprobar si el domingo antes de llegar a ejecutar la acción. Nuestro código, por lo tanto, debe ir en el método *OnActionExecuting*. Lo que hacemos es comprobar si estamos en domingo, devolvemos un resultado directamente desde el filtro (en el nuestro avisando de que nos negamos a trabajar..). De esta forma 'saltamos' la acción, ya que el filtro asume la tarea de devolver una respuesta, sin que el método de acción llegue a hacer su trabajo.
   El método *OnActionExecuted* se ejecutaría después de que el método de acción haya hecho su trabajo, por lo que no es de interés para lo que deseamos, y por lo tanto dejamos el método vacío.

7. Nuestro filtro ya está listo, sólo nos queda utilizarlo. Para ello abrimos nuestro viejo *HomeController* y aplicamos el filtro a la acción *Index()*.

```
[NotOnSundaysFilter]
public IActionResult Index()
{
    return View();
}
```

8. Sólo nos queda ejecutar la aplicación y comprobar que la acción funciona correctamente de lunes a sábado, pero protesta si lo intentamos en domingo.