

UD3 ASP.NET Core MVC.

3.4 Modelos.

Los datos son el corazón, el elemento central de cualquier aplicación. Un usuario introduce datos, los modifica o los busca. Incluso podría decirse que una aplicación es simplemente una interfaz que nos proporciona las operaciones que realizamos con los datos. Por esta razón, es absolutamente necesario para cualquier framework, proporcionar un mecanismo para simplificar el trabajo con datos. Los modelos son utilizados en ASP.NET MVC para representar los objetos del dominio de negocio.

Introducción a los modelos.

En ASP.NET MVC, los modelos son simples objetos POCO (Plain Old C# Objects). Básicamente su función es la de modelar entidades del mundo real.

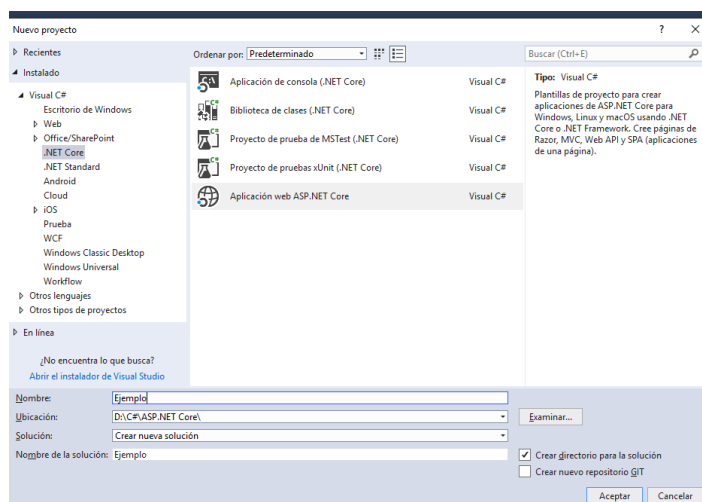
Por ejemplo, en una aplicación de e-commerce, las clases del modelo podrían ser Producto, Pedido o Inventario. En una aplicación para un centro educativo serían sin embargo clases como Estudiante, Profesor o Asignatura.

Los modelos representan el modelo de negocio de nuestra aplicación y no son conscientes ni dependen en ningún caso del almacén de datos utilizado para persistir estos datos. Para los modelos de negocio es igual si los datos son almacenados en una base de datos, en un fichero xml, o utilizan cualquier otro mecanismo.

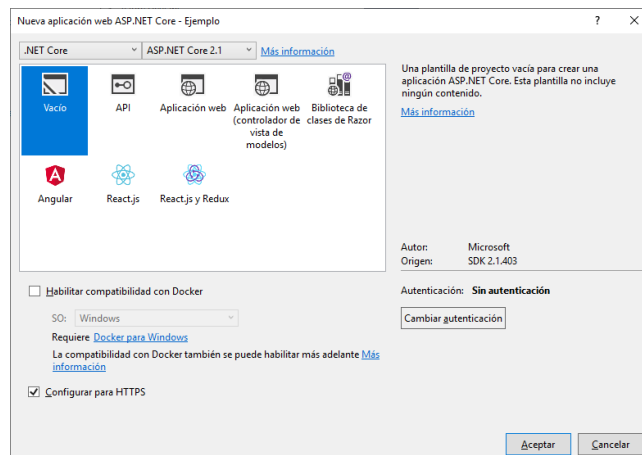
Ejemplo: Trabajo con modelos.

Esta vez, vamos a crear un nuevo proyecto para nuestro ejemplo. De este modo repasaremos todas las tareas necesarias para empezar a trabajar desde una plantilla vacía.

1. Comenzamos creando un nuevo proyecto con **Archivo | Nuevo | Proyecto** y seleccionamos **Aplicación web ASP.NET Core**. En esta ocasión le llamaremos **EjemploModelos**.



2. A continuación seleccionamos la plantilla de proyecto vacío.



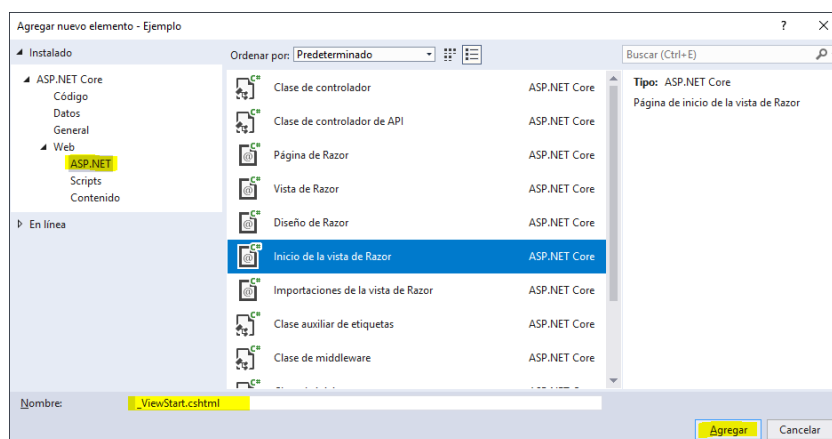
3. Editaremos nuestro fichero Startup.cs, para que no devuelva un mensaje “Hola mundo!!” por defecto, para que sirva contenido estático y lo configuramos para que utilice MVC.

```
public class Startup
{
    // This method gets called by the runtime. Use this method to add
    // services to the container.
    // For more information on how to configure your application, visit
    // https://go.microsoft.com/fwlink/?LinkID=398940
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
    }

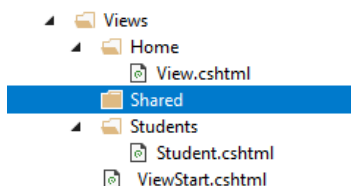
    // This method gets called by the runtime. Use this method to configure
    // the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

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

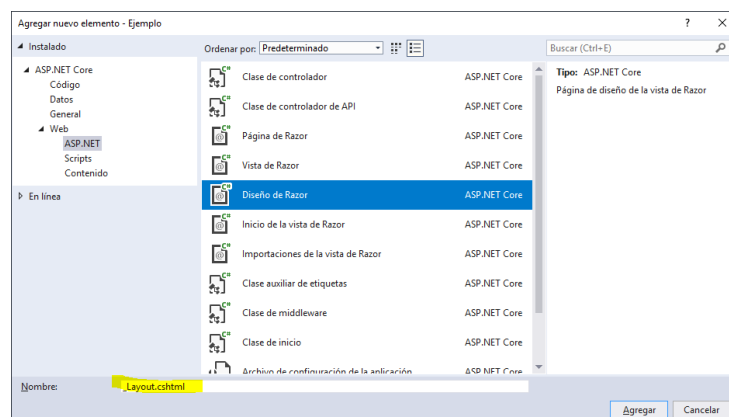
4. Creación de un fichero **_ViewStart.cshtml**. Este es un fichero especial que nos permite definir código común que será ejecutado al comienzo de todas nuestras vistas. Gracias a este fichero podemos definir comportamiento común a las páginas en un único lugar, lo que simplifica el código y facilita el trabajo al evitarnos repetir una y otra vez el mismo código. Si queremos aplicar un layout determinado a todas nuestras páginas, este es por lo tanto el lugar ideal donde especificarlo.
Este fichero, al igual que el resto de vistas debe estar en una carpeta llamada **Vistas**, por lo que lo primero que haremos es crear esta carpeta. Desde esta carpeta hacemos click derecho para agregar el fichero, para ello seleccionamos **Agregar | Nuevo elemento...**:



- Añadir un fichero de layout. Para ello creamos primero en la carpeta de vistas una carpeta de nombre *Shared*:



Ahora, dentro de esa carpeta agregamos un fichero de layout (Diseño de Razor).



- Crear un fichero de vista que utilice el layout. En este caso, nuestra vista será una vista como cualquier otra, con la única particularidad de que al no tratarse ya de una página completa, sólo debe contener el código específico del contenido que queremos mostrar. No necesitaremos que contenga ninguna etiqueta `<html>`, `<body>` o `<title>`, ya que estas ya están especificadas en el layout.
En nuestro ejemplo, en lugar de crear una nueva vista, que ya hemos hecho en múltiples ocasiones, vamos a modificar la vista *Index.cshtml* del controlador *Home*, que ya tenemos, para que utilice correctamente el layout.

```
@{
    Layout = null;
}

<!DOCTYPE html>
```

```
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Index</title>
</head>
<body>
  Hola, hemos creado <b>nuestra primera vista!!</b>
</body>
</html>
```

Vamos a editar el código tal y como se indicó. La primera parte indica que esta página no utiliza layout. Lógicamente ya no queremos este código, y además tampoco necesitamos

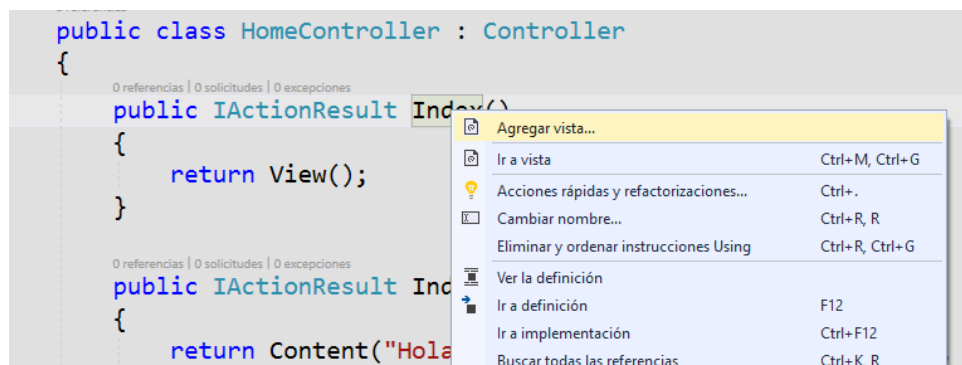
indicar explícitamente que si lo vamos a utilizar, ya que lo hemos hecho en la página de inicio, así que simplemente eliminamos esa parte. tampoco necesitamos nuestras etiquetas html, que ya contiene el layout, si no simplemente el código que es específico de la página. De este modo nuestra página sería simplemente algo así:

```
@{ViewBag.Title = "Home - Index";}
```

Hola, hemos creado nuestra primera vista, esta vez con layout!!

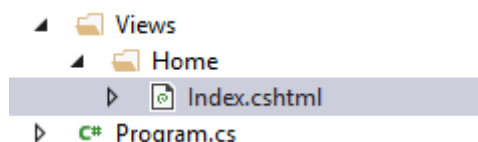
Como podemos comprobar, este mecanismo simplifica enormemente la creación de vistas. Simplemente definimos el valor que será utilizado por el layout para mostrar como título y el contenido específico de la vista.

7. Vamos ahora a añadir un controlador para nuestra aplicación. Empezamos por crear una carpeta llamada **Controllers**. En esta carpeta agregamos un controlador haciendo clic derecho y seleccionando **Agregar | Nuevo controlador....** Llamaremos *HomeController* a nuestro controlador.
8. Abrimos el código de nuestro controlador y hacemos clic derecho dentro de los límites del método Index. En el menú pinchamos en **Agregar vista...**



9. 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. Dejamos marcada la opción página de diseño, que indica que nuestra página utilizará el layout.

10. Cuando aceptemos, y después de unos instantes, habrá creado la vista, que tendremos abierta en el editor. Nuestra vista estará guardada en *Views/Home*.



11. Ya tenemos una aplicación web configurada para utilizar MVC que contiene un controlador y una vista, además de un layout general para todas nuestras vistas, y una vista inicial para incluir el código común a todas las vistas. Es el momento de añadir un objeto del modelo de negocio.

Para ello empezamos por crear una carpeta **Models**. Dentro de ella crearemos una clase llamada *Producto.cs*. Esta clase representa un producto para un sistema de gestión de pedidos. Para nuestro ejemplo utilizaremos una clase *Producto* sencilla, que contiene únicamente un *Id*, el nombre y el precio del producto:

```
namespace EjemploModelos.Models
{
    public class Producto
    {
        public int Id { get; set; }
        public string Nombre { get; set; }
        public decimal Precio { get; set; }
    }
}
```

Hemos escogido el tipo de datos *decimal* para la propiedad *Precio*, en lugar de *float* o *double*, ya que, aunque estos tipos de datos son más eficientes, en ocasiones se realiza algún tipo de redondeo sobre estos datos. El tipo *decimal* es más apropiado para valores de moneda, mientras que los tipos *double* o *float* para cálculos científicos o matemáticos.

12. Actualizamos ahora el método *Index()* para que cree una lista de productos y la pase a la vista. Recordamos que el método principal y recomendado de pasar datos a la vista, es a través del modelo, dejando el *ViewData* / *ViewBag* para datos adicionales y accesorios. De este modo nos aprovechamos de las ventajas de trabajar con datos tipados y ayudas como el completado de código o la detección de errores en tiempo de compilación.

```
public IActionResult Index()
{
    List<Producto> productos = new List<Producto> {
        new Producto {
            Nombre = "Teléfono móvil",
            Precio = 300
        },
        new Producto {
            Nombre = "Ordenador portátil",
            Precio = 1000
        },
        new Producto {
            Nombre = "Tablet",
            Precio = 600
        }
    };

    return View(productos);
}
```

13. Ahora que nuestra vista recibe una lista de productos del controlador, vamos a hacer que muestre estos productos en forma de lista HTML.

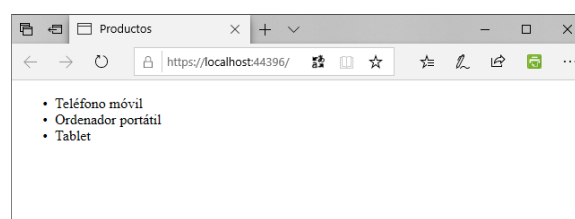
```
@model List<Producto>

@{
    ViewData["Title"] = "Productos";
}

<ul>
    @foreach (var producto in Model)
    {
        <li>@producto.Nombre</li>
    }
</ul>
```

Es importante distinguir en este código la diferencia entre nuestro objeto Model y la instrucción model. La instrucción model del comienzo del fichero especifica los llamados metadatos del modelo, que en la práctica se traduce en que estamos indicando a la vista el tipo de dato que vamos a recibir, de forma que el compilador conoce el tipo del dato esperado y por lo tanto conocerá las propiedades y métodos disponibles. Más adelante en el código, nos referimos a Model (con mayúscula), que es una referencia al dato en sí, el objeto que ha recibido la vista del controlador. En nuestro caso, al ser una lista de productos podemos recorrerlo con un bucle, y al conocer el tipo del dato el compilador, no tenemos ningún error. Además, el bucle reconoce que está recorriendo una lista de objetos de tipo Product, por lo que cuando escribimos @producto. dentro de la lista automáticamente obtenemos las propiedades disponibles del objeto, lo que permite ser más productivo al escribir más rápido gracias al autocompletado, y menos propenso a errores.

14. Ahora es momento de ejecutar nuestra aplicación para ver el resultado, que es el esperado.



15. Ahora nuestra aplicación consta de un controlador, que recoge la petición del usuario, simula la obtención de datos del modelo (lo normal no es crear una lista directamente en el código sino obtener los datos de un origen de datos como una bd) y pasa estos datos a la vista para que sean mostrados al usuario. Todos los elementos trabajan conjuntamente, cada uno con su función, para conseguir nuestro objetivo.

Vamos ahora a complicar un poco nuestro modelo, utilizando objetos relacionados entre si. Creamos una clase *Pedido.cs*, que contiene una lista de productos:

```
using System.Collections.Generic;

namespace EjemploModelos.Models
{
    public class Pedido
    {
        public int Id { get; set; }
        public List<Producto> Productos { get; set; }
        public decimal Total { get; set; }
    }
}
```

16. Modificamos nuestro método *Index()* para que cree un pedido con una lista de productos y calcula el total. Recordamos de nuevo que lo habitual es que esto no se haga directamente en el controlador, en una aplicación real obtendríamos estos datos de un origen de datos. Para ello añadiremos el siguiente código a continuación de la creación de la lista de productos.

```
Pedido pedido = new Pedido();
pedido.Productos = productos;
pedido.Total = productos.Sum(product => product.Precio);

return View(pedido);
```

17. Lo que toca ahora es modificar la vista para que se adapte al nuevo modelo que le pasamos, y muestre los datos del pedido, por ejemplo en una tabla.

```
@model Pedido

@{
    ViewData["Title"] = "Pedido";
}


<table border="1">

    <tr>
        <th>Producto</th>
        <th>Precio</th>
    </tr>

    @foreach (var Product in Model.Productos)
    {
        <tr>
            <td>@Product.Nombre</td>
            <td>@Product.Precio</td>
        </tr>
    }

    <tr>
        <td><b>Total</b></td>
        <td><b>@Model.Total</b></td>
    </tr>
</table>
```

18. Podemos observar que es posible acceder a todas las propiedades del modelo, y recorren la lista de objetos que contiene, obteniendo a su vez acceso a las propiedades anidadas. Todo esto es posible gracias a que especificamos el uso de un modelo de datos tipado en la primera línea de la vista.
Ejecutamos nuestro proyecto y obtenemos el resultado previsto.



The screenshot shows a web browser window with the address bar displaying 'https://localhost:44396/'. The browser tabs are labeled 'Productos' and 'Pedido'. The main content area displays a table with the following data:

Producto	Precio
Teléfono móvil	300
Ordenador portátil	1000
Tablet	600
Total	1900

Modelos específicos para una vista.

Existen escenarios en los que en una aplicación podríamos tener una vista en la que sólo nos interesa ofrecer la posibilidad de mostrar o modificar algunas propiedades de un modelo más grande, o podríamos tener una vista que tenga que mostrar datos de más de un único objeto del modelo de datos. En estos casos, lo recomendable es crear un modelo específico para esa vista.

Por ejemplo, imaginemos que necesitamos una vista en la que podamos actualizar el precio de un producto. Esta página podría contener solamente el código del producto, su nombre y su precio, ya que el resto de características no es relevante. El modelo, sin embargo, podría tener decenas de propiedades que describen el producto, como fabricante, color, tamaño, material, etc... En lugar de enviar el modelo completo a la vista, lo cual sería claramente ineficiente, podemos crear un nuevo modelo, específico para esa vista concreta, que contenga sólo las propiedades que necesitamos.

ViewModels.

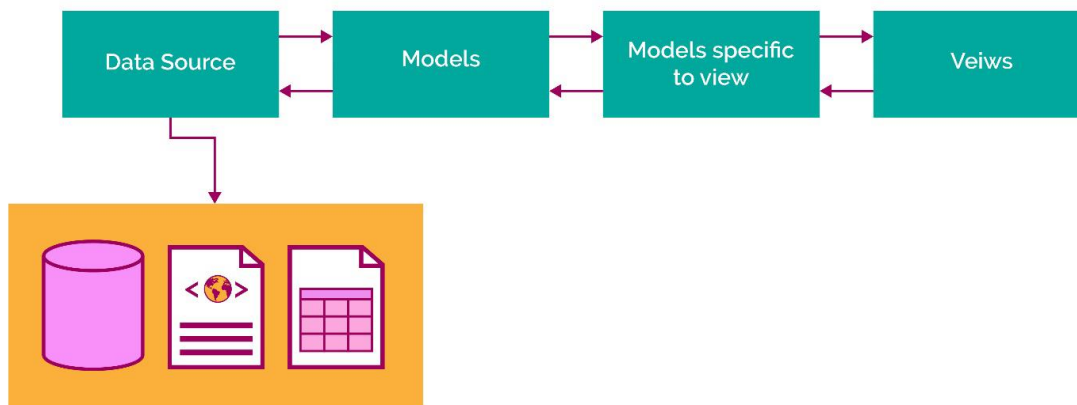
Los ViewModels son entidades específicas para una vista, de modo que pasemos a la vista un objeto que contenga solamente las propiedades que esta necesita.

El uso de ViewModels es completamente opcional, ya que siempre podemos utilizar únicamente los modelos de negocio de nuestra aplicación, pero es altamente recomendable ya que reduce el acoplamiento y mejora la separación de responsabilidades.

Hay sin embargo inconvenientes asociados al uso de ViewModels. En primer lugar, debemos crear una nueva clase para nuestro ViewModel. En segundo lugar, necesitamos escribir código que transfiera los datos de nuestro ViewModel a un modelo y viceversa. Existen frameworks para automatizar este proceso, como AutoMapper. Veremos en más detalle el uso de ViewModels en unidades posteriores.

Flujo de datos con respecto al modelo.

El siguiente diagrama muestra el flujo de los datos en una aplicación ASP.NET MVC:



Los dos elementos más importantes en cuanto al flujo de datos son:

- **Origen de datos:** Representa los datos de nuestra aplicación. Estos datos pueden residir en cualquier medio, desde una compleja base de datos relacional, pasando por una simple hoja de Excel o un fichero xml..
- **Modelos:** Representan las entidades del dominio de negocio de nuestra aplicación y deben ser independientes del origen de datos. El mismo modelo podría ser utilizado si cambiamos nuestro origen de datos.

Podemos usar nuestros modelos tal cual en nuestras vistas para obtener los datos del usuario o presentarlos al cliente. Pero, como indicamos anteriormente, en ocasiones los datos que necesita la vista no coinciden con los del modelo, pueden ser una pocas propiedades de un gran objeto, o podemos necesitar parte de varios modelos diferentes. En estos casos, la mejor opción es crear un modelo específico para esa vista.

Esta es, a alto nivel, la secuencia de eventos que suceden cuando se almacena un registro en ASP.NET Core, utilizando el modelo:

1. El usuario introduce la información en un formulario. Los campos del formulario no tienen porqué representar el modelo completo, siendo posible que sólo necesitemos algunas de las propiedades del modelo.
2. Los datos introducidos son recibidos por el controlador por un proceso denominado Model Binding, que veremos en detalle en la próxima unidad. Este proceso mapea los datos del formulario a un objeto de nuestro modelo o ViewModel.
3. Si los datos se reciben en un ViewModel, debemos convertir éste a un modelo.
4. Finalmente, el modelo es almacenado en el origen de datos.