ASP.NET Core MVC

Desarrollo Web en Entorno Serevidor

Índice

1.Introducción

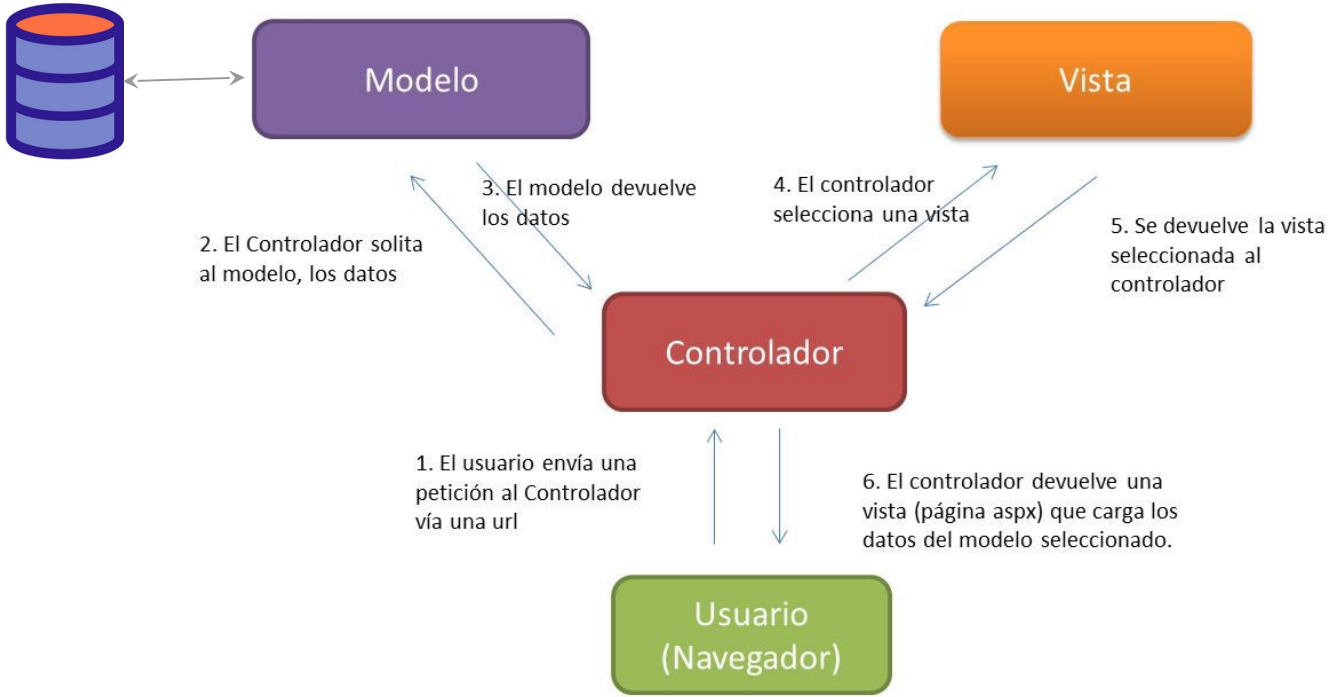
MVC

MVC es un patrón de arquitectura software que separa una aplicación en tres componentes principales: modelos, vistas y controladores.

Esto permite la separación de intereses en cada uno de estos elementos (MVC) y repartir las responsabilidades de una formas más fácil de codificar, depurar y probar, ya que cada una de estas partes tiene solo un trabajo.

En una aplicación MVC, la vista solo muestra información. El controlador controla la entrada y la interacción del usuario y responde a sus peticiones. El modelo se encarga de validar los datos de un modelo e interactuar con la base de datos.

MVC


Modelo
Vista
3. El modelo devuelve los datos
4. El controlador selecciona una vista
2. El Controlador solita al modelo, los datos
5. Se devuelve la vista seleccionada al controlador
Controlador
1. El usuario envía una petición al Controlador vía una url
6. El controlador devuelve una vista (página aspx) que carga los datos del modelo seleccionado.
Usuario (Navegador)

MVC

Modelos

- Son clases que representan los datos de la aplicación.
- Las clases de modelo usan lógica de validación para aplicar las reglas de negocio para esos datos.
- Normalmente, los objetos de modelo recuperan y almacenan el estado del modelo en una base de datos.
- Los datos actualizados se escriben en una base de datos.
- El modelo actualiza los datos en una base de datos, ya sea para crear, leer, actualizar o eliminar estos datos.
- Estas son las funciones básicas de la persistencia de información en bases de datos, a las que nos referiremos con el acrónimo CRUD (Create-Read-Update-Delete)

MVC

Vistas

- Son los componentes que muestran la interfaz de usuario (IU) de la aplicación.
- Por lo general, esta interfaz de usuario muestra los datos del modelo.
- Utilizaremos la sintaxis Razor para generar las páginas dinámicas con los datos del modelo.

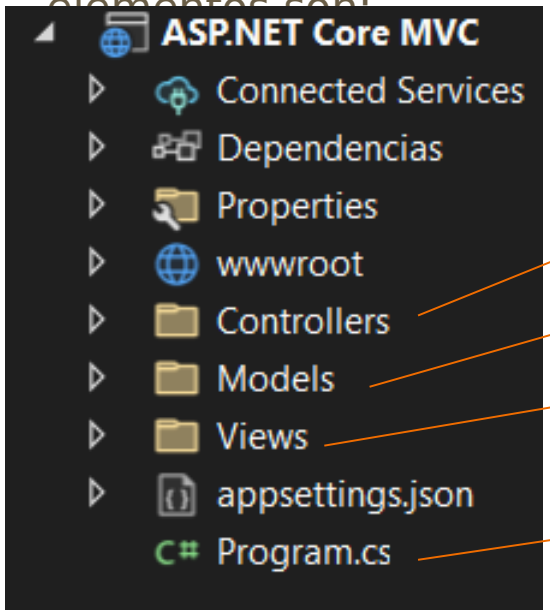
MVC

Controladores

- Son clases que controlan las solicitudes que el usuario realiza a través del navegador.
- Recuperan datos del modelo.
- Llaman a plantillas de vista que devuelven una respuesta.

Proyecto ASP.NET Core MVC: Estructura

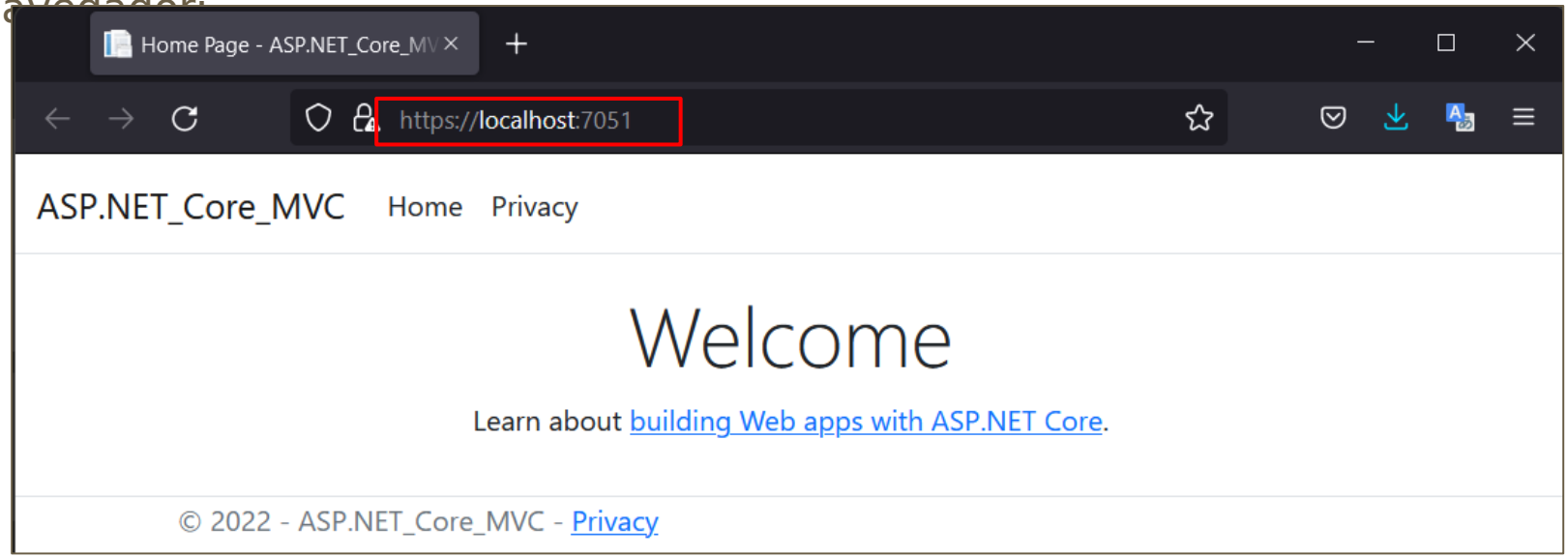
Tras crear un nuevo proyecto con la plantilla de ASP.NET Core MVC, nos encontramos con la siguiente estructura de directorios. Algunos de sus elementos son:



- Controladores del proyecto
- Modelos del proyecto
- Vistas del proyecto
- Inicializador del proyecto

Proyecto ASP.NET Core MVC: Navegación

Al ejecutar el proyecto por primera vez se carga la siguiente página en el navegador:



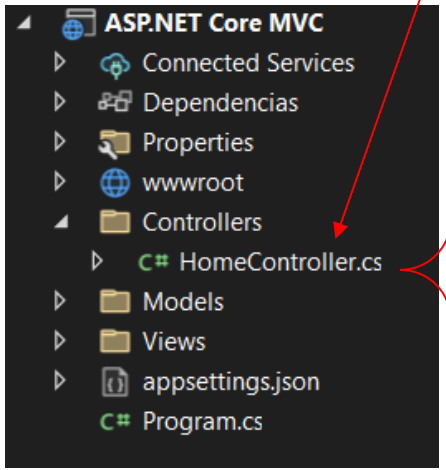
Pero ... ¿Cómo hemos llegado hasta ahí?

Proyecto ASP.NET Core MVC: Navegación

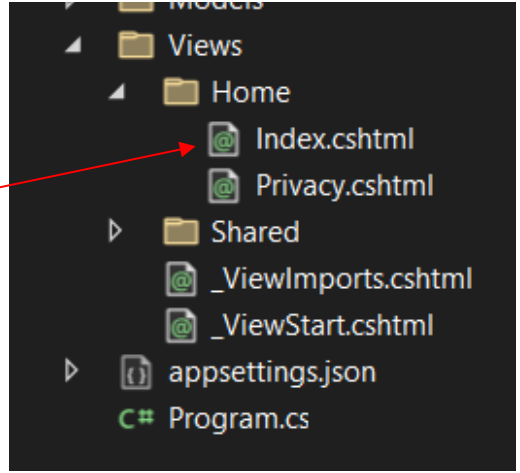
La primera vez se carga la **ruta por defecto**, que indica la acción del controlador definido en “MapControllerRoute” de “Program.cs”

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

Parámetro opcional

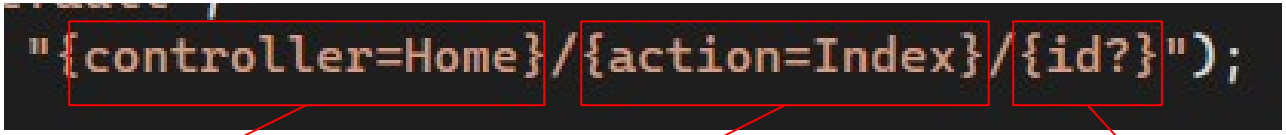



```
public class HomeController : Controller
{
    private readonly ILogger<HomeController>

    0 referencias
    public HomeController(ILogger<HomeContro

    0 referencias
    public IActionResult Index()
    {
        return View();
    }

    0 referencias
    public IActionResult Privacy()...

    [ResponseCache(Duration = 0, Location =
    0 referencias
    public IActionResult Error()...
}
```



Proyecto ASP.NET Core MVC: Navegación

La estructura de una URL se ha de entender de la siguiente manera:



Controlador que gestiona la petición del usuario

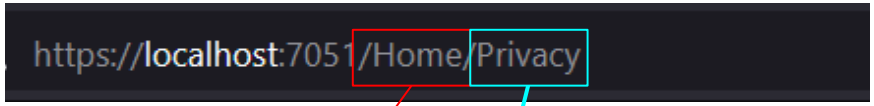
Acción del controlador anterior que se ejecutará

Parámetro opcional (lo es por estar seguido del símbolo ?) indicado para contener un id y ser recibido en la acción con ese mismo nombre

Esto se puede entender mejor navegando a la página “Privacy del menú superior de la página y analizando su URL

Proyecto ASP.NET Core MVC: Navegación

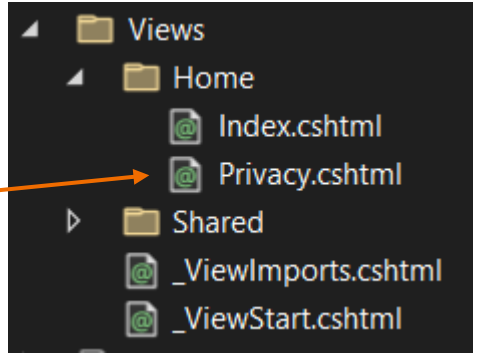
La estructura de una URL se ha de entender de la siguiente manera:



Se invoca a la función/acción Privacy del controlador HomeController

```
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;

    0 referencias
    public HomeController(ILogger<HomeController> logger)...

    0 referencias
    public IActionResult Index()...

    0 referencias
    public IActionResult Privacy()
    {
        return View();
    }
```

Se carga la vista Privacy, que tiene el nombre de la acción y está en el directorio con el nombre del controlador



Ejercicio

Realiza las siguientes acciones:

- Prueba a escribir estas 3 URLs en el navegador (el port es el que corresponda en cada caso):
  - https://localhost:<port>/
  - https://localhost:<port>/Home
  - https://localhost:<port>/Home/Index

  ¿Qué diferencia hay entre ellas? ¿Por qué sucede esto? Reflexiona tu respuesta.

- Modifica tu proyecto para que por defecto se cargue la página de “Privacy Policy” al iniciar.

Proyecto ASP.NET Core MVC: Navegación

¿Cómo se ha indicado que había que navegar a esa ruta (URL)?

Esto se produce gracias a los enlaces que hay en el header de la página, definidos en "_Layout.cshtml", que indican el controlador y la acción que han de figurar en la barra de navegación y que se usarán para poder cargar la vista que corresponda.

```
<div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
    <ul class="navbar-nav flex-grow-1">
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Index">Home</a>
        </li>
        <li class="nav-item">
            <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
        </li>
    </ul>
</div>
```

Proyecto ASP.NET Core MVC: Controladores

Ya hemos visto cómo los controladores están formados por unas **funciones públicas** que llamamos **acciones**.

Estas acciones son llamadas según lo especificado en el segmento de la URL.

Los controladores que hemos visto hasta el momento devuelven una llamada al la función View(), la cual retorna un tipo de dato “IActionResult”.

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

Esta función devuelve un tipo de dato IActionResult

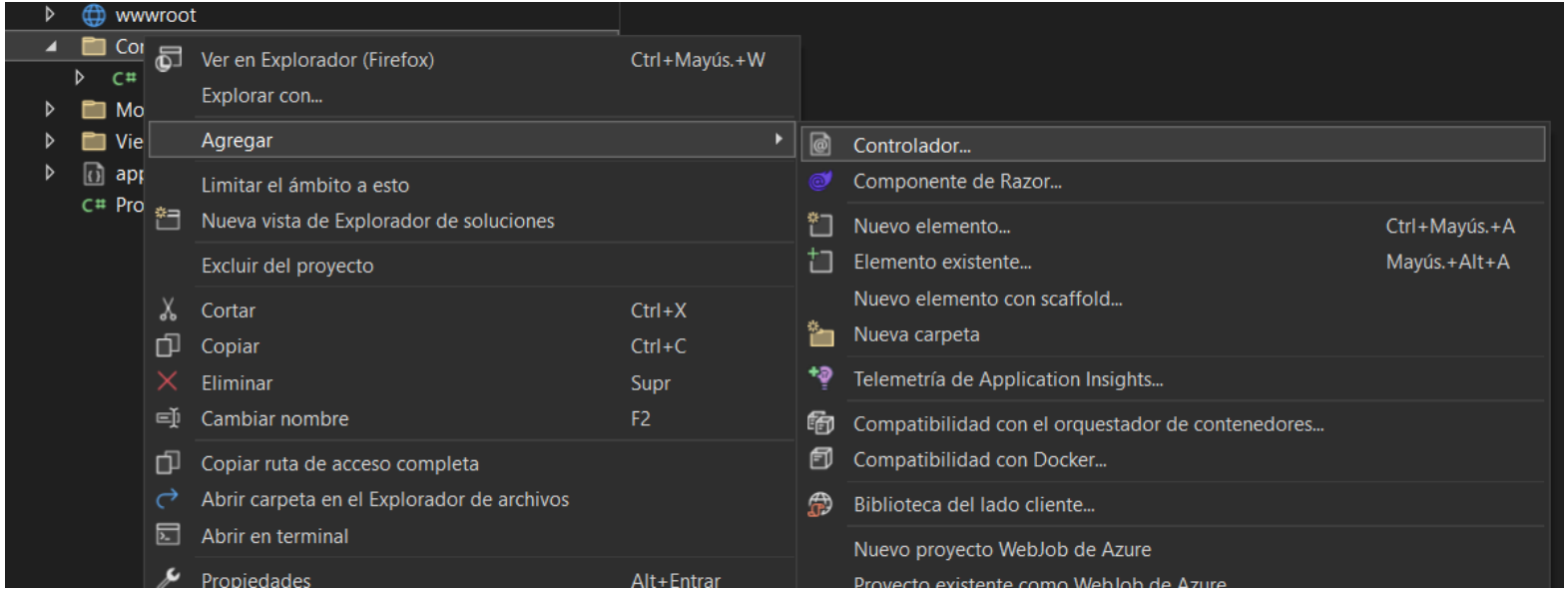
De esta forma, se carga la vista que comparte nombre con la acción y que se encuentra en una carpeta con el nombre del controlador.

Las acciones de un controlador también pueden devolver tipos de datos simples, de forma que lo que se mande al navegador sea una cadena de texto. Veámoslo con un ejemplo…

Proyecto ASP.NET Core MVC: Controladores

Probemos esto en un nuevo controlador.

Para ello agregamos un controlador a nuestra carpeta “Controllers”

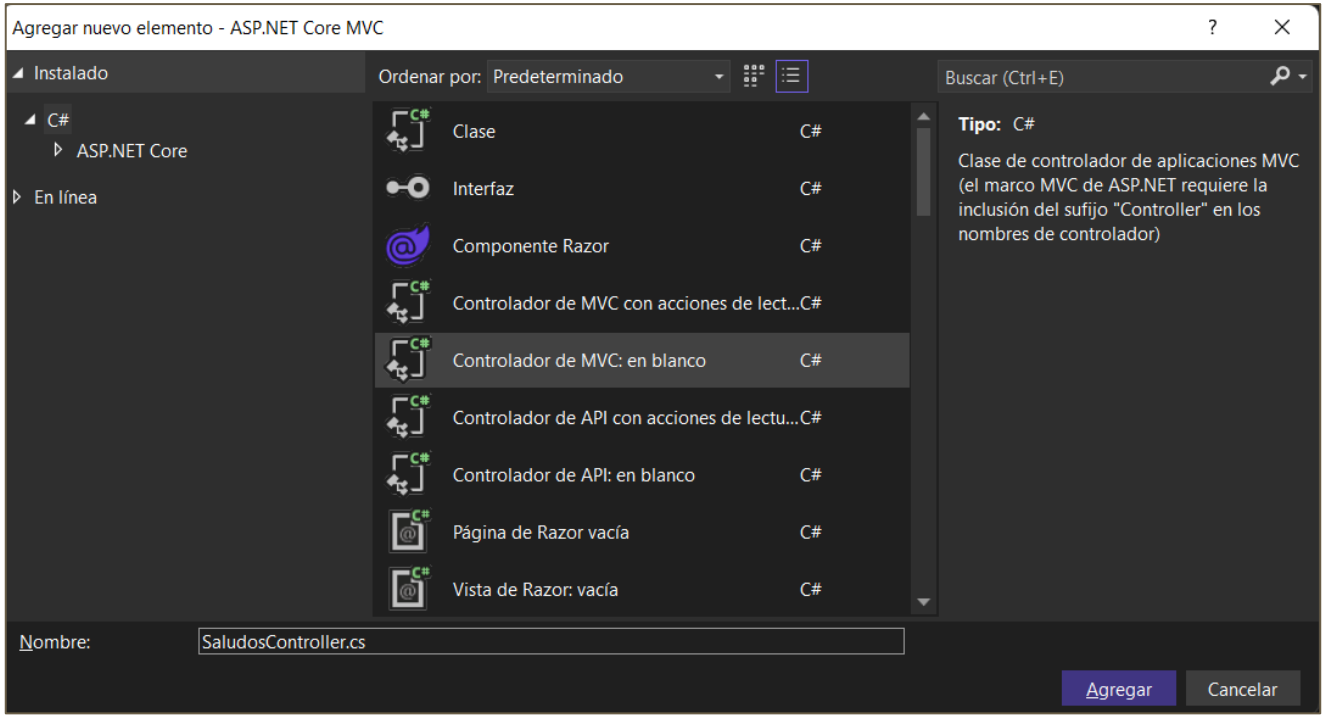


Proyecto ASP.NET Core MVC: Controladores

Elegimos la opción “Controlador MVC: en blanco” y pulsamos “Agregar”

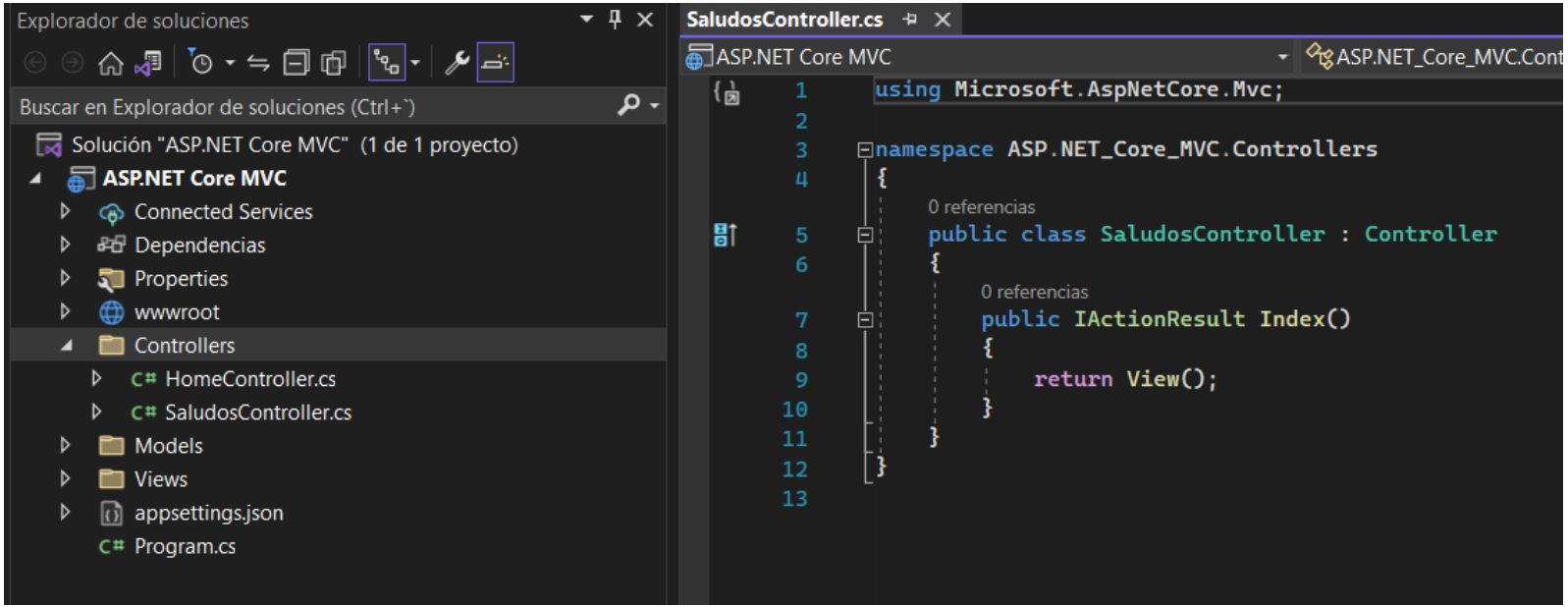
Proyecto ASP.NET Core MVC: Controladores

Le damos el nombre de “SaludosController.cs” y pulsamos “Agregar”



Proyecto ASP.NET Core MVC: Controladores

¡Ya lo hemos creado!



Ejercicio

Realiza las siguientes acciones en el nuevo controlador “SaludosController” creado:

- Modifica el controlador recién creado para que por defecto devuelva una cadena de texto saludando a todo el mundo... Verifica que funciona lanzando el proyecto.
- Crea una nueva acción en el controlador que se despida de todo el mundo. El nombre de esta acción será “Despedida”

Proyecto ASP.NET Core MVC: Controladores

Obtención de Datos de la URL

Los controladores pueden tomar datos de la URL de dos formas distintas:

- Mediante el tercer segmento de la URL {id?}
- Mediante la cadena de consulta.

Se pueden usar estas estrategias por separado o combinadas.

En los dos casos, será necesario que la acción del controlador tenga definidos como parámetros de entrada los valores que desea tomar de la URL, teniendo que coincidir los nombres de los parámetros y de las variables.

Proyecto ASP.NET Core MVC: Controladores

Obtención de Datos de la URL: Tercer segmento de la URL {id?}

Como se dijo anteriormente, este parámetro es opcional, es por ello que tiene un símbolo ?

Un ejemplo de URL que use este segmento sería el siguiente (los nombres del controlador y la acción no son relevantes en este ejemplo):

```
https://localhost:7051/controllerName/actionName/5
```

La acción invocada en esta URL tendrá que declarar en su cabecera un parámetro de nombre ID y de tipo entero (*int*), tal y como muestra la siguiente imagen:

```
public string actionName(int ID)
{
    return "Se ha recibido el parámetro ID con valor: "+ID;
}
```

El parámetro ID tomará el valor de 5 en ese ejemplo.

Proyecto ASP.NET Core MVC: Controladores

Obtención de Datos de la URL: Cadena de Consulta

La cadena de consulta es un conjunto de parámetros, en formato campo-valor, que pueden figurar en la URL y ser usados por la acción del controlador correspondiente.

Esta cadena de consulta se ha de escribir al final del último segmento de la URL y precedida del símbolo ?

Los elementos cambo-valor que compongan la cadena de consulta han de estar separados entre ellos por el símbolo & → ?name1=val1&name2=val2&name3=val3.....


```
public string actionName(string name, int age)
{
    return $"Mi Amigo {name} tiene {age} años";
}
```

¡OjO! Se ha utilizado interpolación para incluir las variables en la cadena de texto

Proyecto ASP.NET Core MVC: Controladores

Obtención de Datos de la URL: Combinando estrategias

Es posible combinar las dos estrategias anteriores.

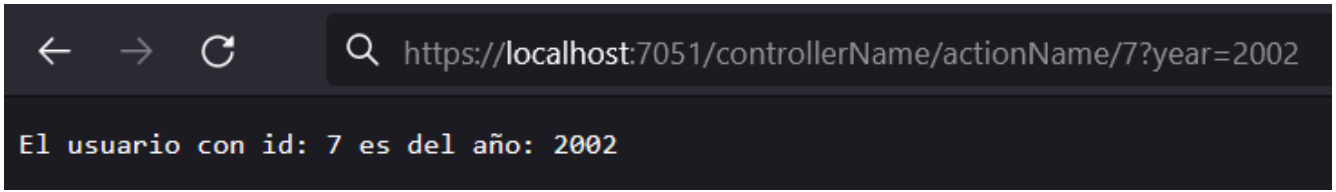
Para ello no hay que hacer nada especial, sólamente respetar los nombres de los parámetros y segmento ID.

```
https://localhost:7051/controllerName/actionName/7?year=2002
```

```
public string actionName(string year, int ID)
{
    return $"El usuario con id: {ID} es del año: "+year;
}
```

El orden de los parámetros no es relevante

Se ha combinado interpolación y concatenación



Ejercicio

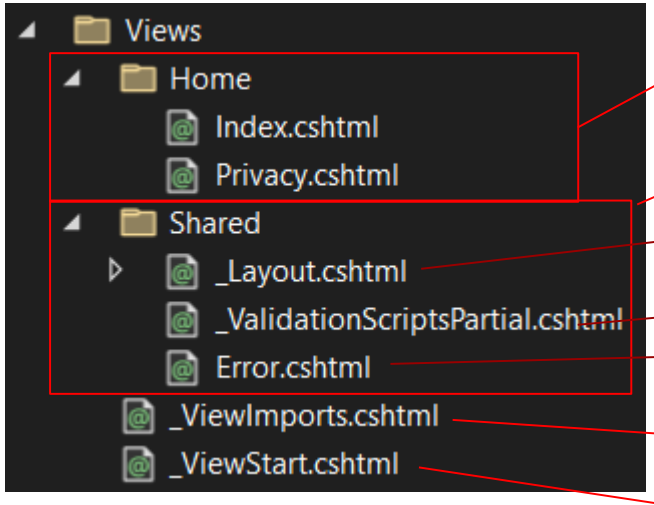
Modifica el controlador anteriormente creado con las siguientes indicaciones:

- Edita la acción por defecto del controlador para que acepte el nombre de una localidad como parámetro en la URL y sólamente salude a esa localidad. (Ejemplo de salida: “Hola a Tacoronte”)
- Edita la acción “Despedida” para que tome un valor de ID (por el tercer segmento de la cadena URL) y dos parámetros más en la cadena de consulta, que serán nombre y edad. Con esta información se mostrará un texto del tipo: “El usuario con ID: 5 y nombre Manuel se despide a sus 35 años”

Proyecto ASP.NET Core MVC: Vistas

Las páginas cargadas en el navegador son una composición de distintas vistas parciales que se combinan para formar el conjunto de la página.

Vamos a conocer primero la estructura del directorio “Views” en primer lugar:



Contiene todas las vistas parciales del controlador “HomeController

“Shared” contiene vistas que son compartidas por todas las páginas.

- Es el contenedor principal de la página web cargada en el navegador
- Define los *scripts* que contendrán las páginas
- Es la vista parcial de la página de error

Define las librerías y helpers disponibles en las vistas

Indica cuál es el *layout* inicial de las páginas

Proyecto ASP.NET Core MVC: Vistas

En nuestros proyectos con ASP.NET Core MVC usaremos archivos de vista Razor.

De esta forma se encapsula el proceso de generación del HTML que se envía al cliente.

Las plantillas de las vistas se crean usando Razor. Estas vistas basadas en Razor:

- Tienen una extensión .cshtml
- Proporcionan una forma elegante de crear una salida HTML con C#

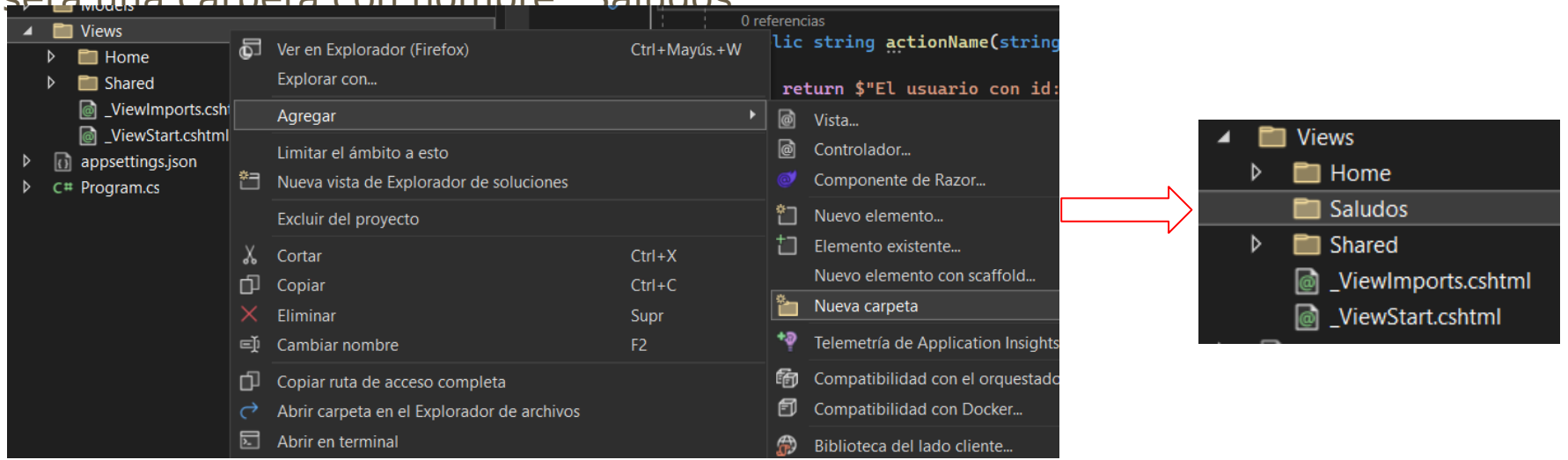
En los ejercicios anteriores se retorna un *string* en los métodos de acción de los controladores, sin embargo, y de forma general, las acciones de los controladores retornan un tipo *IActionResult* o una clase derivada de *ActionResult*, no un string.

Vamos a crear una vista para el controlador “Saludos” usado anteriormente.

Proyecto ASP.NET Core MVC: Vistas

Para crear una vista para la acción de un controlador, el primer paso es asegurarse de que existe una carpeta con el nombre del controlador en el directorio *Views*.

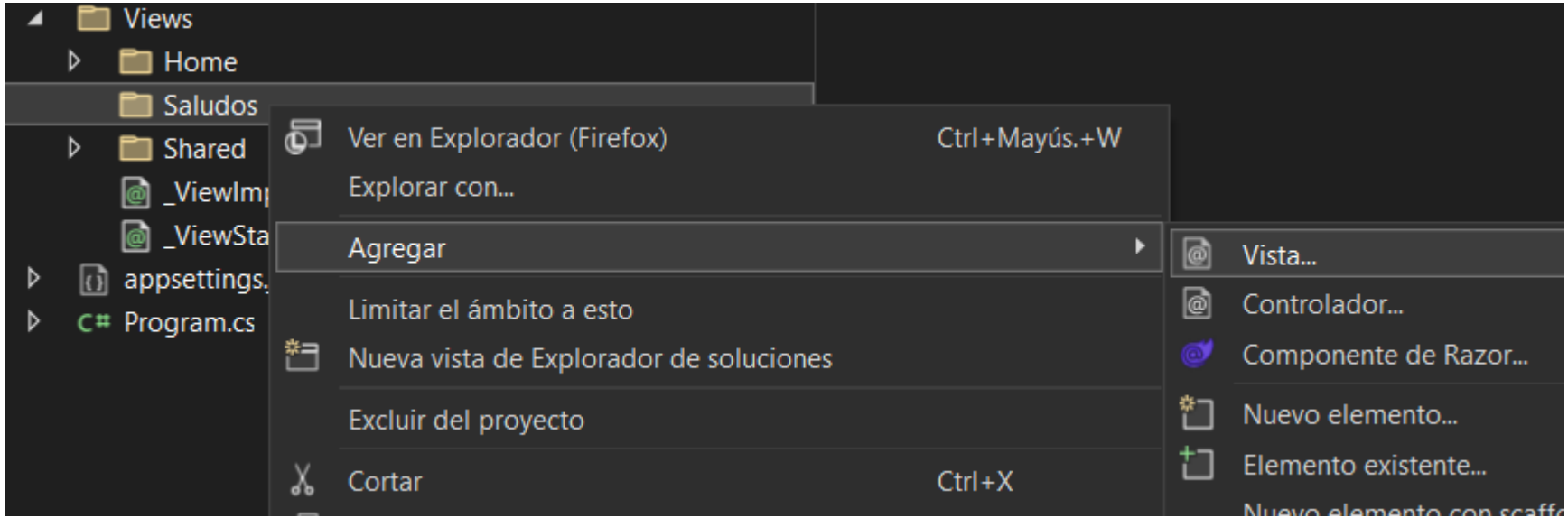
Si no existe tenemos que crearla. En este caso, y siguiendo el ejemplo anterior, será una carpeta con nombre “Saludos”:



Proyecto ASP.NET Core MVC: Vistas

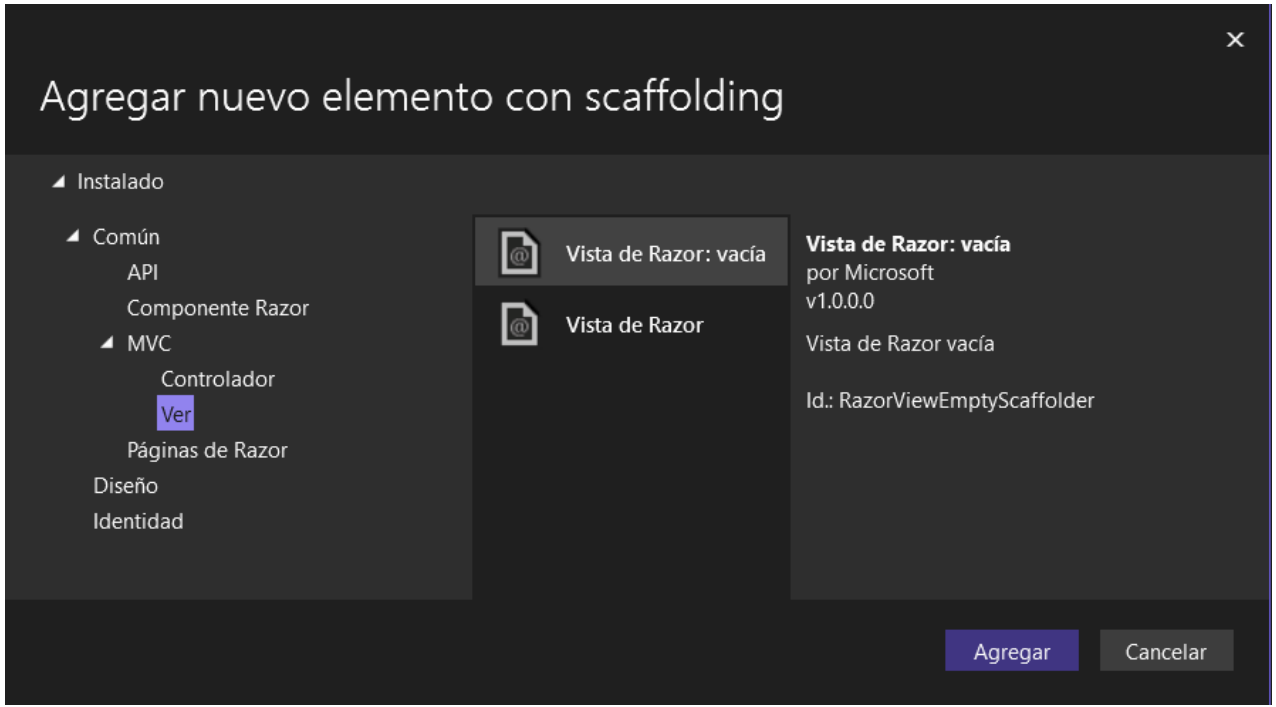
El paso siguiente será agregar una vista en el directorio Views/<controllerName>

Como será la vista por defecto del controlador, se quedará con nombre index.cshtml.



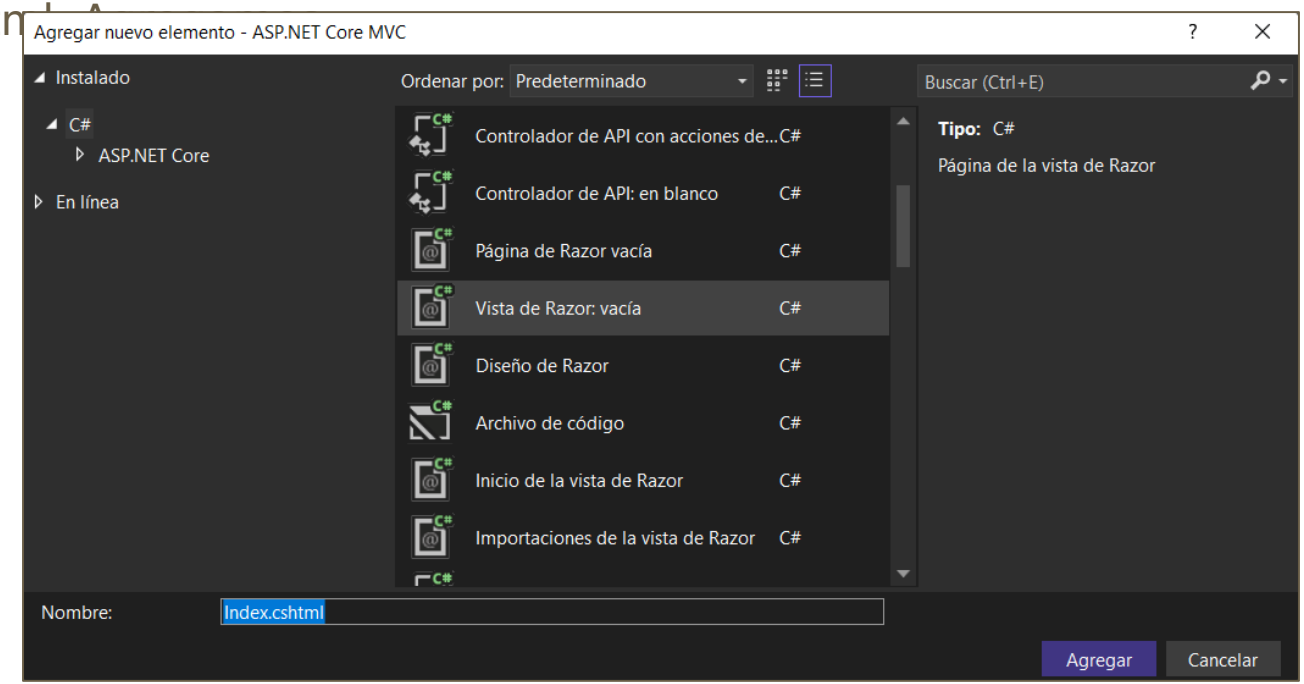
Proyecto ASP.NET Core MVC: Vistas

Seleccionamos “Vista de Razor: vacía” y pulsamos “Agregar”



Proyecto ASP.NET Core MVC: Vistas

En el siguiente paso nos aseguramos que estemos creando “Vista de Razor: vacía” y que el nombre es el adecuado. Como es vista por defecto dejamos index.cshtml.



Proyecto ASP.NET Core MVC: Vistas

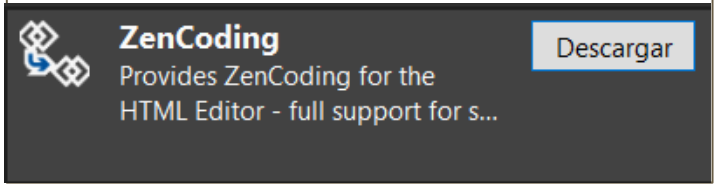
Modificamos el archivo creado “index.cshtml” para que tenga el siguiente contenido:

Index.cshtml

```
1  @{
2      ViewData["Title"] = "Página Saludos";
3  }
4
5  <h2>Index Action - Saludos Controller</h2>
6
7  <p>¡Hola a todo el mundo!</p>
```

Elemento Razor que incluye un título a la página mostrada

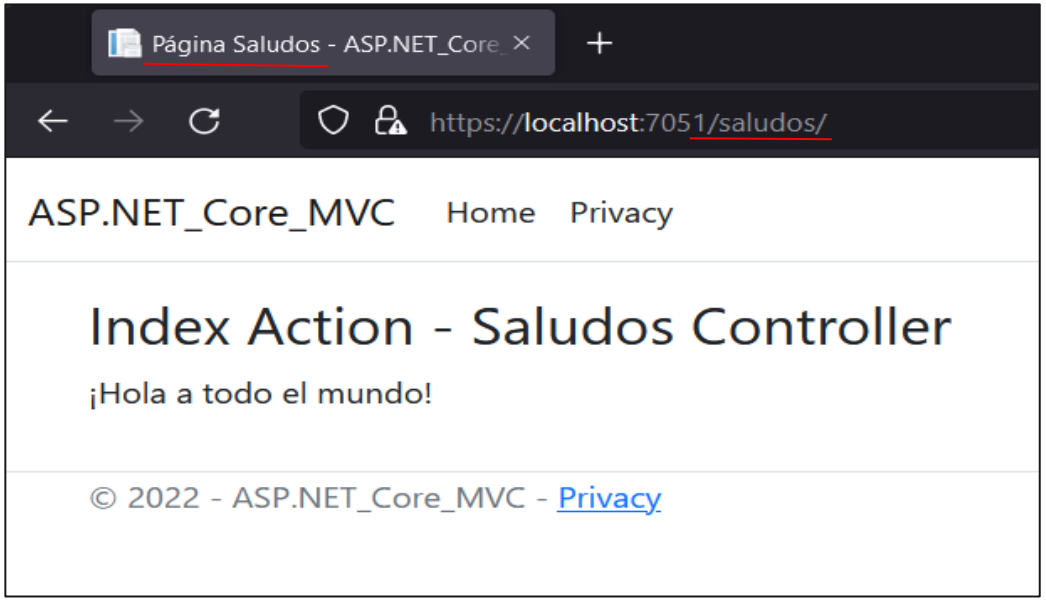
CONSEJO:
Incluye la extensión ZenCoding para poder utilizar la sintaxis Emmet en las plantillas de las vistas.



Proyecto ASP.NET Core MVC: Vistas

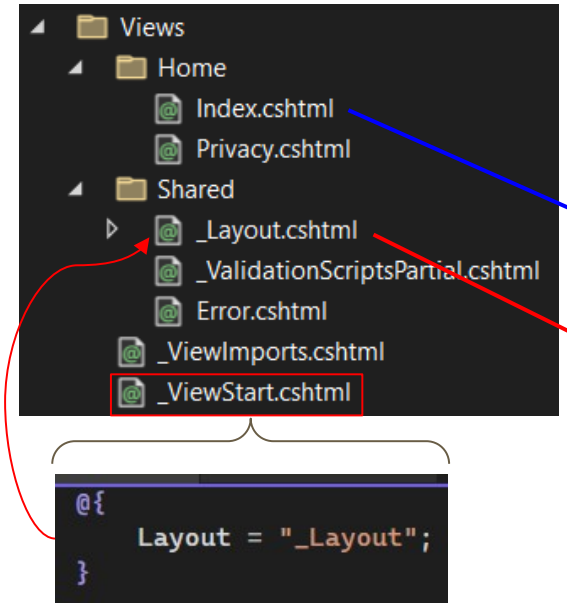
Modificamos la acción del controlador relacionada con esta vista para que devuelva una llamada al método *View()* que devuelve el tipo de dato *IActionResult* y lanzamos.

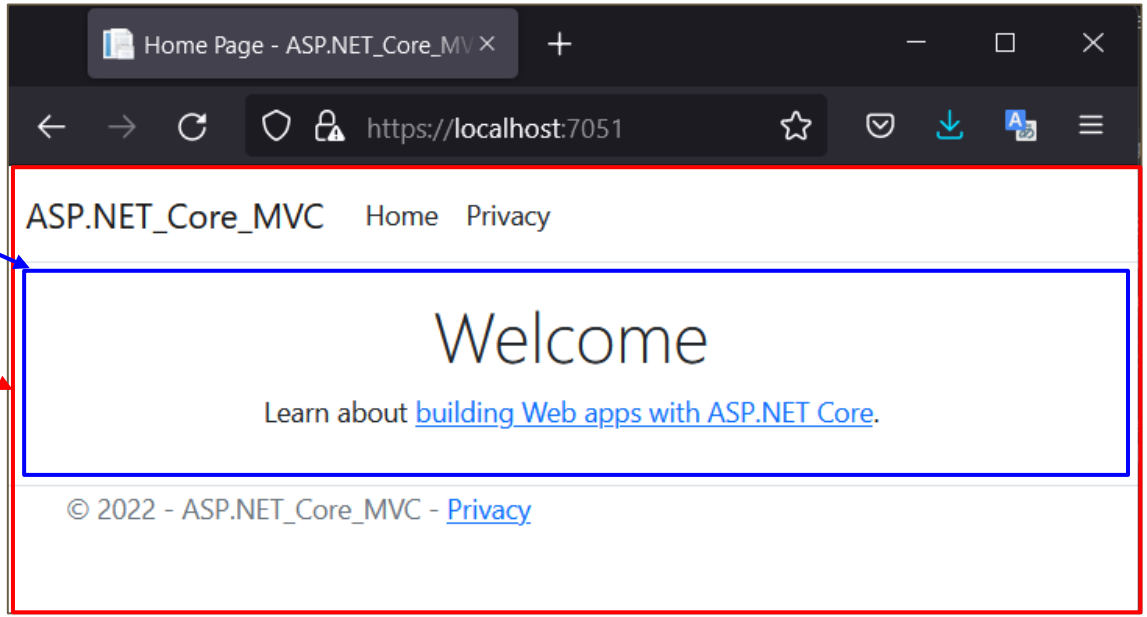
```
public class SaludosController : Controller
{
    0 referencias
    public IActionResult Index()
    {
        return View();
    }
```



Proyecto ASP.NET Core MVC: Vistas (Estructura)

La página anterior está compuesta por más elementos de los definidos en la vista.
Esto está relacionado con el concepto de *layout*. Tratemos de entenderlo
El pintado de la la página “Home” de nuestro ejemplo la forman estos parciales:





Proyecto ASP.NET Core MVC: Vistas (Estructura)

- _ViewStart.cshtml: Indica cuál será el layout principal de las páginas
- _Layout.cshtml:
  - Define la estructura principal de las páginas
  - Es el contenedor principal de la página
  - Es usado para contener todas las vistas parciales de nuestro sitio.
  - Incluye una llamada a **@RenderBody()**, que es el lugar donde se cargan las vistas específicas (los parciales) que quedan incluidas dentro del layout principal
  - Contiene los elementos principales de una página HTML (doctype, head, body, enlaces a librerías, importación de scripts, etc) así como la definición del *header* y *footer*

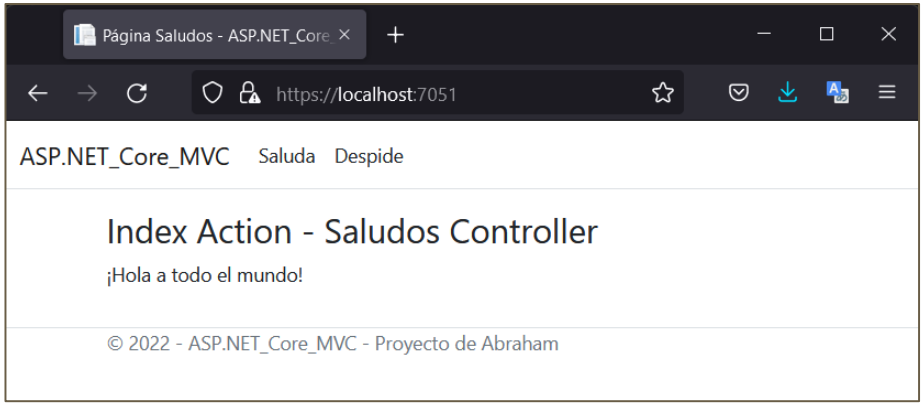
```
<div class="container">
    <main role="main" class="pb-3">
        @RenderBody()
    </main>
</div>
```

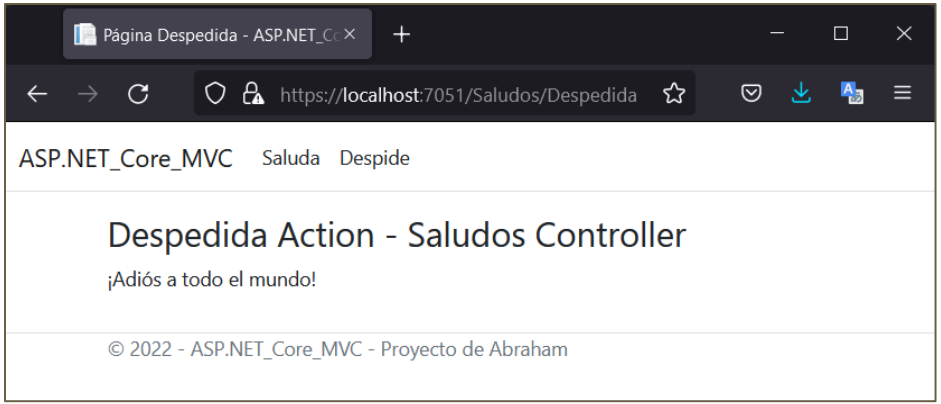
Ejercicio

Realizar las siguientes acciones:

- Crear la vista que se acaba de explicar
- Modificar la ruta por defecto de la aplicación para que navegue a esta vista.
- Crear una vista para la acción “Despedida” que despida a todo el mundo y modifica su acción correspondiente para que pueda ser mostrada.
- Edita el Layout principal para que se produzca navegación entre estas dos vistas.
- Modifica el Layout principal para que muestre algo similar a lo siguiente:

Ejercicio


Página Saludos - ASP.NET_Core
https://localhost:7051
ASP.NET_Core_MVC
Saluda
Despide
Index Action - Saludos Controller
¡Hola a todo el mundo!
© 2022 - ASP.NET_Core_MVC - Proyecto de Abraham


Página Despedida - ASP.NET_C
https://localhost:7051/Saludos/Despedida
ASP.NET_Core_MVC
Saluda
Despide
Despedida Action - Saludos Controller
¡Adiós a todo el mundo!
© 2022 - ASP.NET_Core_MVC - Proyecto de Abraham

Proyecto ASP.NET Core MVC: Vistas

Pasar datos del Controlador a la Vista

Los controladores son responsables de proporcionar los datos requeridos a una vista para que genere y de formato a un HTML como respuesta para el navegador.

Una Vista nunca debería:

- Realizar lógica de negocio [1], [2]
- Interactuar directamente con la base de datos

Una vista (View Template) sólo debería trabajar con los datos que le proporciona el controlador. Esta separación de conceptos nos ayuda a mantener un código: Limpio, Testable y Mantenible

Este framework nos ofrece 2 formas de pasar datos desde el controlador a la vista:

- El diccionario ViewData
- Mediante un modelo (se verá más adelante)

Proyecto ASP.NET Core MVC: Vistas

Pasar datos del Controlador a la Vista: ViewData

El objeto ViewData es un diccionario que permite el paso de datos desde el controlador a la vista de forma dinámica.

Gracias a este tipo de mecanismos, las vistas pueden generar contenido dinámicamente, ya sea obteniendo estos datos de ViewData o de un Modelo.

ViewData es un objeto dinámico, lo que significa que se puede usar cualquier tipo de datos.

Se usa de la siguiente forma:

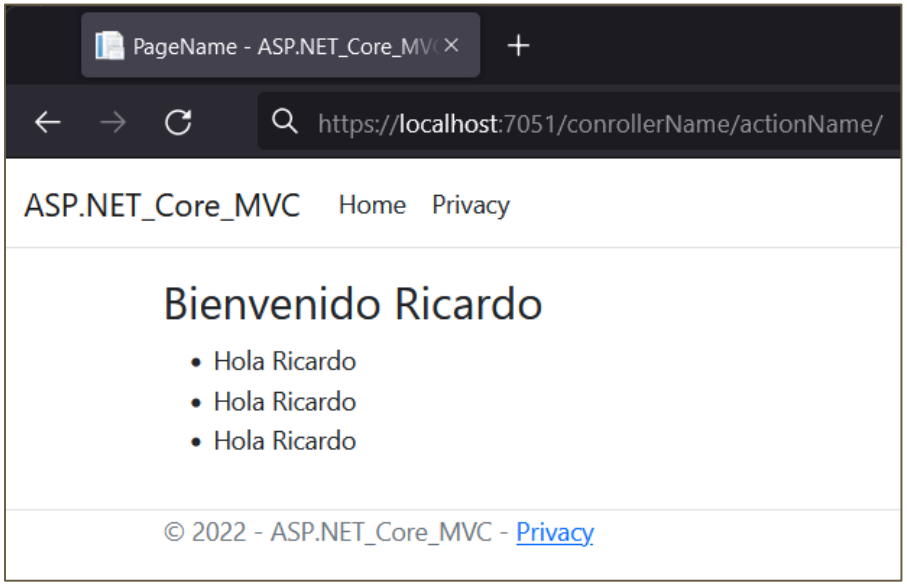
```
public IActionResult actionName()
{
    ViewData["name"] = "Ricardo";
    ViewData["numTimes"] = 3;

    return View();
}
```

Proyecto ASP.NET Core MVC: Vistas

Pasar datos del Controlador a la Vista: ViewData

La forma en que una *View* puede usar los datos puestos en *ViewData* es la siguiente:

```
@{
    ViewData["Title"] = "PageName";
}

<h2>Bienvenido @ViewData["name"]</h2>

<ul>
    @for(int i=0; i< (int)ViewData["numTimes"]; i++){
        <li>Hola @ViewData["name"]</li>
    }
</ul>
```



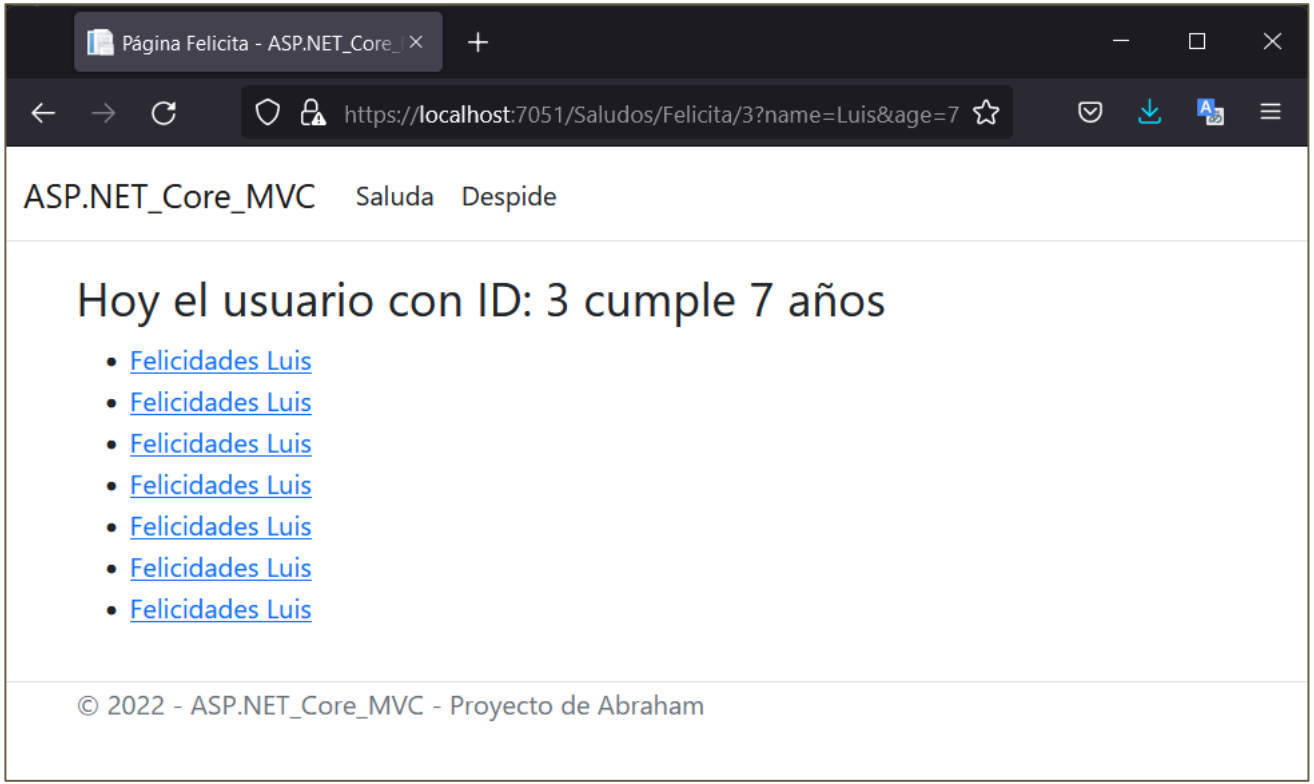
Ejercicio

Realizar las siguientes acciones:

- Crea una nueva acción, en tu controlador “Saludos”, llamada “Felicita”
- Esta nueva acción ha de recibir los siguientes datos:
  - ID de un usuario (en el tercer segmento de la URL)
  - Nombre de un usuario
  - Edad del usuario
- Crea una nueva vista, relacionada con la acción anterior, que reciba los tres datos mediante el objeto *ViewData* y felicite al usuario tantas veces como años tenga.
- Todos esas felicitaciones tendrán que ser enlaces que naveguen a la vista de la acción por defecto del controlador “Saludos”

La apariencia de vista creada será la siguiente:

Ejercicio


Página Felicita - ASP.NET_Core_
https://localhost:7051/Saludos/Felicita/3?name=Luis&age=7
ASP.NET_Core_MVC
Saluda
Despide
Hoy el usuario con ID: 3 cumple 7 años
Felicidades Luis
Felicidades Luis
Felicidades Luis
Felicidades Luis
Felicidades Luis
Felicidades Luis
Felicidades Luis
© 2022 - ASP.NET_Core_MVC - Proyecto de Abraham

ActionResult

- Determinan el tipo de retorno de un Action Method
- Tipos de retorno de los métodos del controller

Name	Behavior
ContentResult	Returns a string
FileContentResult	Returns file content
FilePathResult	Returns file content
FileStreamResult	Returns file content
EmptyResult	Returns nothing
JavaScriptResult	Returns script for execution
JsonResult	Returns JSON formatted data
RedirectToResult	Redirects to the specified URL
HttpUnauthorizedResult	Returns 403 HTTP Status code
RedirectToRouteResult	Redirects to different action/different controller action
ViewResult	Received as a response for view engine
PartialViewResult	Received as a response for view engine

Proyecto ASP.NET Core MVC: Modelos

En la arquitectura MVC que estamos usando, los modelos se corresponden con entidades (tablas) de nuestra base de datos.

MVC ofrece la capacidad de pasar objetos de modelos a una vista.

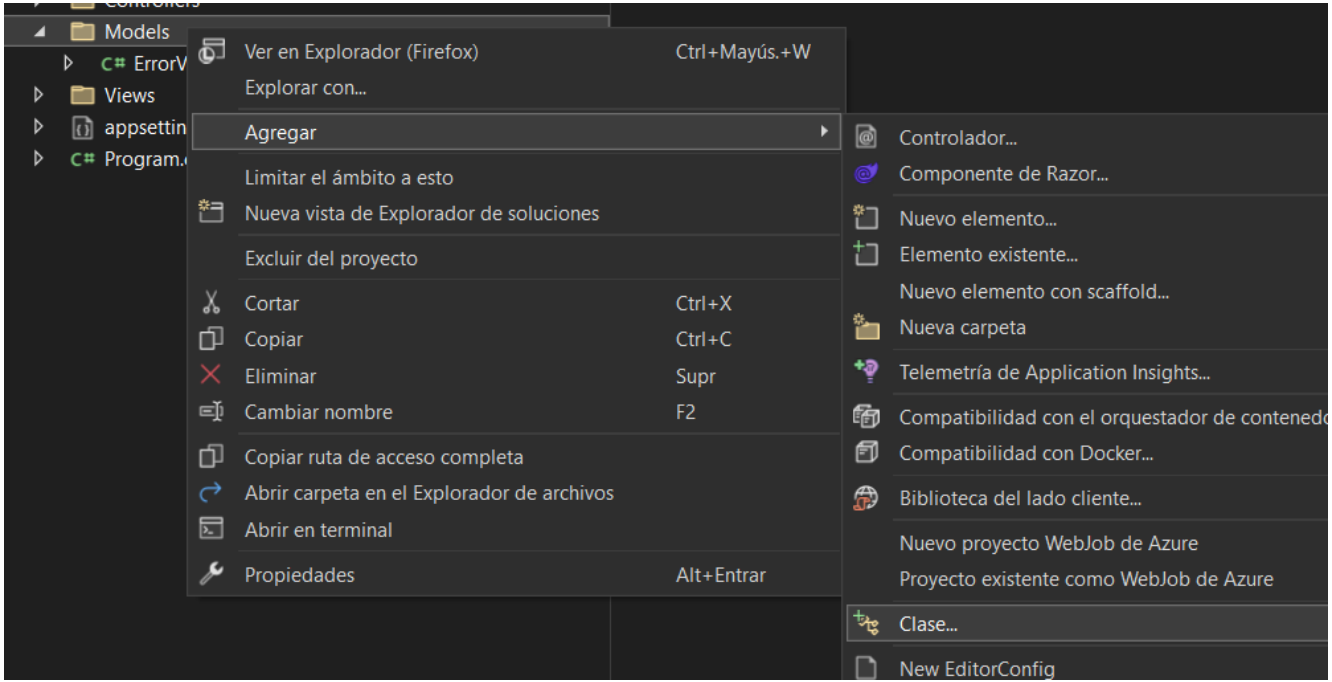
Los modelos serán quienes se comuniquen directamente con la Base de Datos e implementen los métodos CRUD.

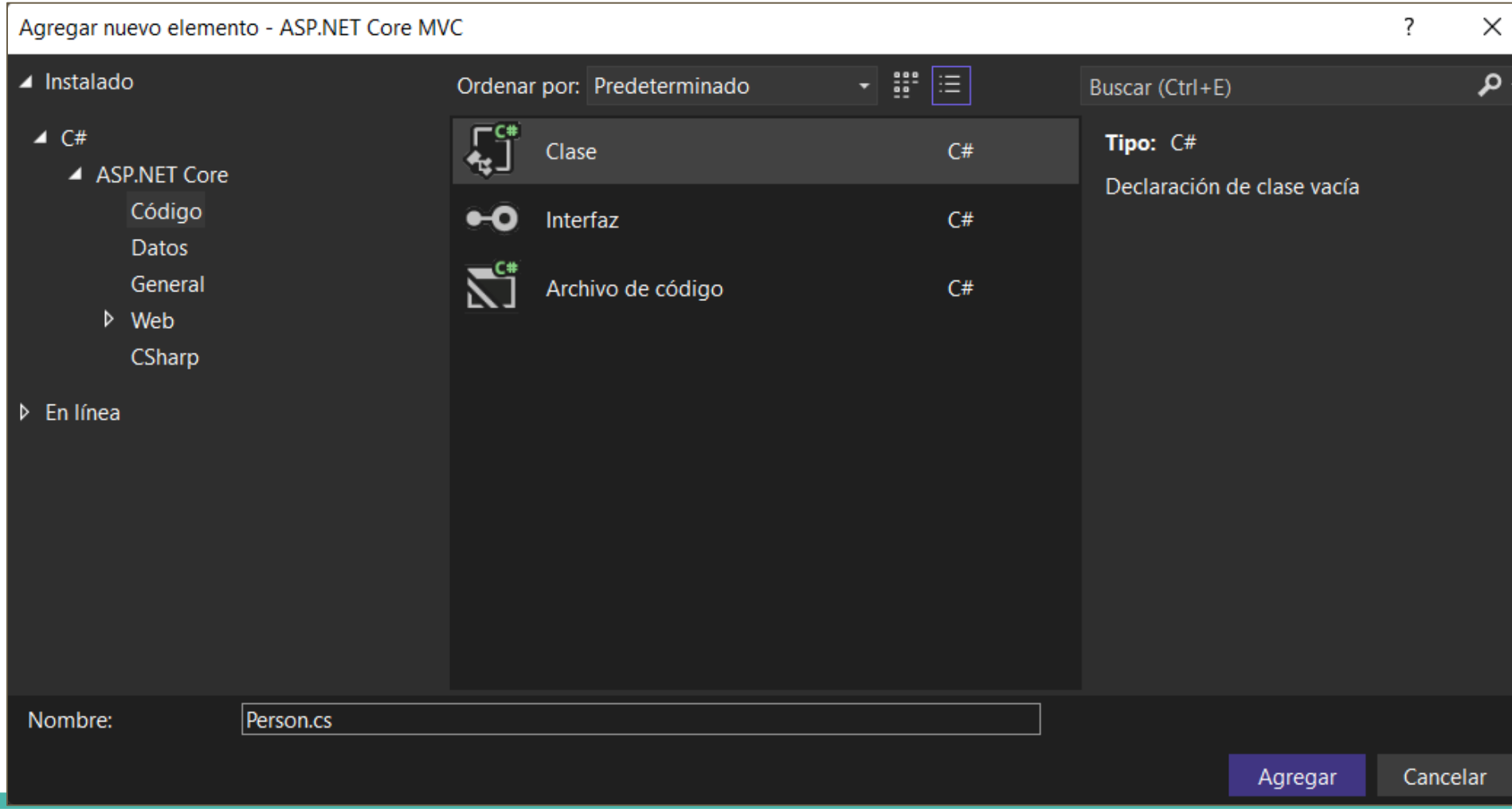
En esta primera aproximación que haremos sobre los modelos, los manejaremos sin usar una base de datos.

Vamos a crear un modelo:

Proyecto ASP.NET Core MVC: Modelos

Agregamos una nueva clase en el directorio “Models”




Agregar nuevo elemento - ASP.NET Core MVC
Instalado
C#
ASP.NET Core
Código
Datos
General
Web
CSharp
En línea
Ordenar por: Predeterminado
Clase C#
Interfaz C#
Archivo de código C#
Buscar (Ctrl+E)
Tipo: C#
Declaración de clase vacía
Nombre: Person.cs
Agregar
Cancelar

Proyecto ASP.NET Core MVC: Modelos

Definimos los atributos/campos que tendrá el nuevo modelo.

Todos estos campos han de ser públicos.

```
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;

namespace ASP.NET_Core_MVC.Models
{
    5 referencias
    public class Person
    {
        2 referencias
        public int Id { get; set; }

        [DisplayName("Nombre")]
        3 referencias
        public string Name { get; set; }
        3 referencias
        public int Age { get; set; }

        [EmailAddress]
        3 referencias
        public string? Email { get; set; }

    }
}
```

El campo Id se usará como clave principal en la BD

DataAnnotation que define el texto que se quiere mostrar para este campo

Define métodos *getter* y *setter* para ese campo

La ? junto al tipo de dato del campo indica que admite un valor NULL

Proyecto ASP.NET Core MVC: Modelos

Enviar Modelo a la Vista

En este punto, podemos crear un objeto del tipo de un modelo determinado y pasarlo a una vista para que muestre su información.

```
public IActionResult Felicita()
{
    Person persona = new Person
    {
        Id = 1,
        Name = "Roberto",
        Age =22,
        Email = "roberto@dominio.com"
    };

    return View(persona);
}
```

Creación de un objeto del modelo Person

Se invoca a la vista pasando el objeto del modelo como parámetro.

Proyecto ASP.NET Core MVC: Modelos

Mostrar Campos del Modelo en la Vista

Para mostrar y utilizar los campos de un modelo, utilizaremos un elemento del framework llamado Tag Helper, concretamente un HTML Helper.

En principio usaremos dos:

- @Html.DisplayNameFor → Muestra el nombre de un campo del modelo. El valor mostrado será el nombre del campo, o el valor definido en la anotación *DisplayName* antes comentada.
- @Html.DisplayFor → Muestra el valor del campo al que se hace referencia.

Para usar un modelo en una vista es necesario especificar el modelo en el archivo de la vista (al principio):

- @model <projectName>.Models.<modelName>

Ejemplo para un supuesto campo *year* de un modelo:

```
<p>@Html.DisplayNameFor(model => model.Year): @Html.DisplayFor(model => model.Year)</p>
```

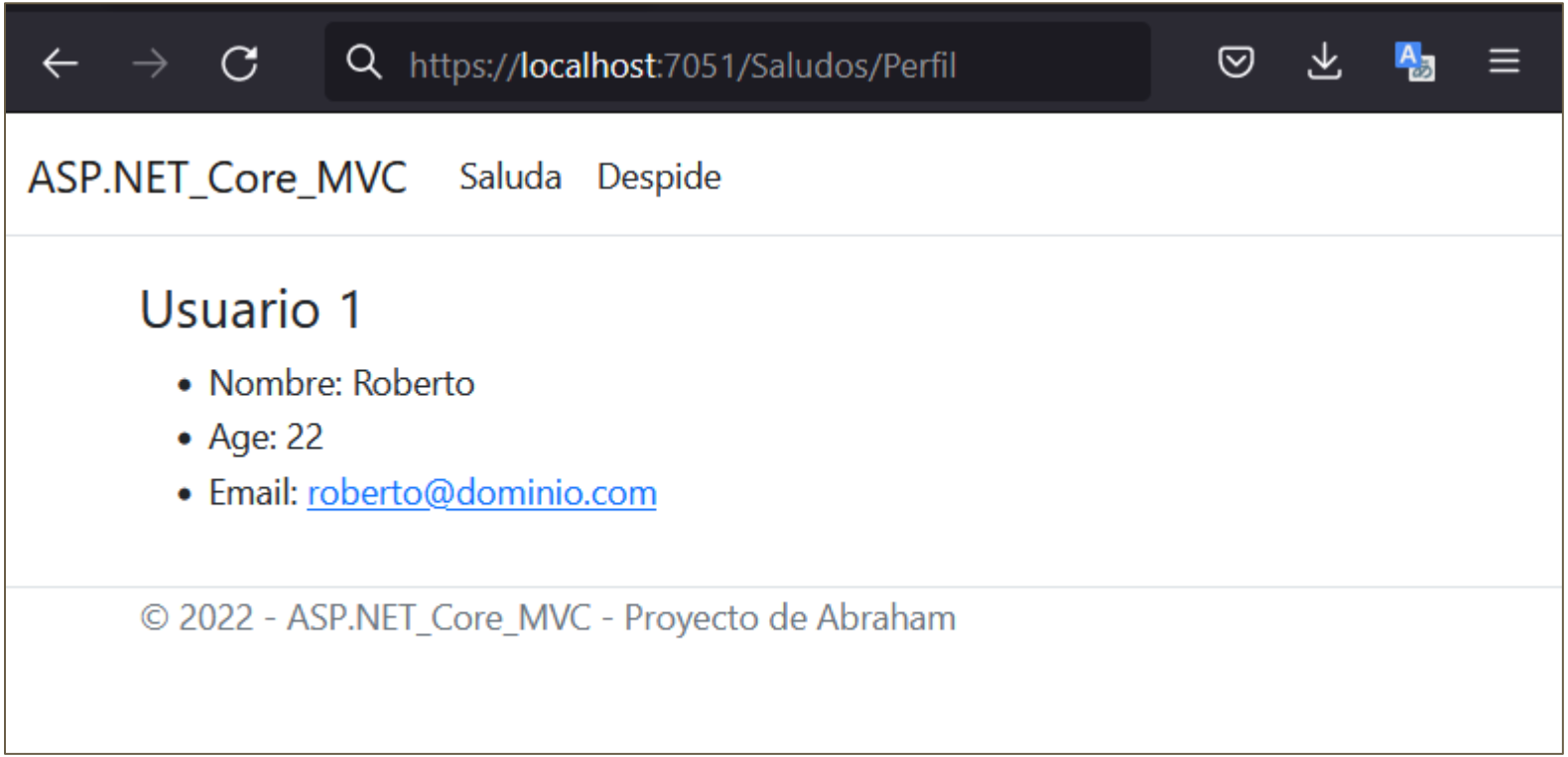
Ejercicio

Realizar las siguientes acciones:

- Crea un modelo como el mostrado en el ejemplo, pero incluyendo nombres para mostrar en cada uno de los campos.
- Crea una acción llamada “Perfil”, en el controlador “Saludos”, que se encargue de crear un objeto del modelo y se lo pase a una vista con el mismo nombre que la acción.
- Modifica la vista para mostrar la información del modelo.

La apariencia de vista creada será similar a la siguiente:

Ejercicio


https://localhost:7051/Saludos/Perfil
ASP.NET_Core_MVC
Saluda
Despide
Usuario 1
Nombre: Roberto
Age: 22
Email: roberto@dominio.com
© 2022 - ASP.NET_Core_MVC - Proyecto de Abraham

Proyecto ASP.NET Core MVC: Modelos

Crear Nuevo Objeto del Modelo

De forma general, y especialmente en las arquitecturas MPA, cuando se pretende crear un nuevo elemento de un determinado tipo de modelo, se sigue el siguiente flujo:

- El cliente comunica al servidor su intención de crear un elemento mediante una petición GET
- El servidor responde a esa petición ofreciendo una página con un formulario que contiene todos los campos necesarios para crear un elemento del modelo.
- El cliente “envía”, mediante una petición POST, todo el formulario cumplimentado
- El servidor crea el objeto del modelo solicitado, siempre y cuando todos los campos del formulario superen la validación pertinente.

Este flujo se podría representar de forma gráfica de la siguiente manera:

CREAR UN NUEVO ELEMENTO DE UN MODELO

Petición **GET**: Solicita página con formulario

Respuesta: Página con el formulario del modelo

Petición **POST**: Formulario cumplimentado

Respuesta: Mensaje de OK o Redirección

2 Acciones del Controlador involucradas en el proceso

Proyecto ASP.NET Core MVC: Modelos

Crear Nuevo Objeto del Modelo

Por tanto, son 2 acciones las involucradas en el proceso de crear un nuevo elemento de un modelo.

- 1ª Acción - GET
  - Su función es la de generar la vista con el formulario completo para crear el modelo
  - Carga una vista que utilizará el modelo para relacionar los elementos label/input con los campos del modelo
- 2ª Acción - POST
  - Recibe la información del formulario en forma de instancia del modelo con sus campos con valores
  - Se encarga de gestionar el modelo recibido → puede persistirlo en la BD o alguna otra función
  - Puede verificar que los datos recibidos sean válidos

Por convenio se utiliza el mismo nombre para estas dos acciones.

Proyecto ASP.NET Core MVC: Modelos

Crear Nuevo Objeto del Modelo - 1ª Acción

Esta acción se encarga de cargar la vista con el formulario correspondiente.

Si la acción es para crear no es necesario pasarle ningún modelo a la vista.

```
[HttpGet]
0 referencias
public IActionResult Create()
{
    return View();
}
```

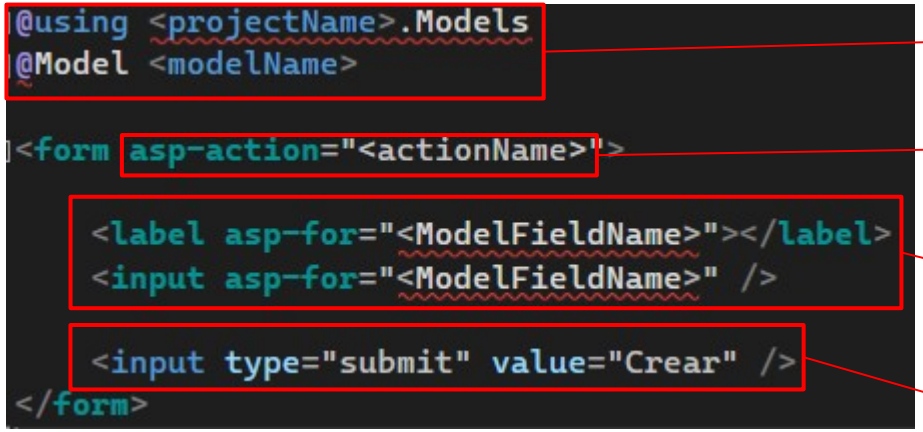
Hacemos explícito que esta acción responde a una petición de tipo GET

Proyecto ASP.NET Core MVC: Modelos

Crear Nuevo Objeto del Modelo - 1ª Acción

Esta acción se encarga de cargar la vista con el formulario correspondiente.

Un esquema de la estructura de un formulario para esta función sería:



Importación del modelo a crear

Acción que se llamará al enviar el form

label e *input* para un campo del modelo. Se relacionan con el atributo “asp-for”

Botón para enviar el formulario

Proyecto ASP.NET Core MVC: Modelos

Crear Nuevo Objeto del Modelo - 2ª Acción

Esta acción recibe los datos del formulario en forma de modelo y realiza las acciones pertinentes en función de lo que se quiera conseguir.

```
[HttpPost]
0 referencias
public IActionResult Create(<ModelClassName> varName)
{
    /*Realizar las acciones pertinentes segun el caso:
        - Almacenar en base de datos
        - Usar el modelo para cualquier otro fin
        - Etc.
    */

    return RedirectToAction("<actionName>");
}
```

Hacemos explícito que esta acción responde a una petición de tipo POST

Se recibe una instancia del modelo creada con los datos del formulario

En este caso se realiza por redireccionar a una acción determinada, pero podría ser otra cosa

Proyecto ASP.NET Core MVC: Modelos

Validación del Modelo en su Creación o Edición

Este proceso involucra de diferente forma a varios archivos del proyecto que participan en la creación/modificación de un elemento de un modelo.

- Clase que define el modelo
- Vista del formulario para crear el modelo
- Acción llamada al hacer *submit* del formulario

Cada uno de estos elementos tiene una labor determinante en la validación del modelo para su creación o modificación.

Proyecto ASP.NET Core MVC: Modelos

Validación del Modelo en su Creación o Edición - Modelo

Como se comentó anteriormente, los campos de un modelo pueden ir acompañados de atributos de validación que definen restricciones a validar sobre cada uno de ellos.

Estas anotaciones se denominan *DataAnnotations* y se pueden consultar en la documentación oficial.

```
[StringLength(60, MinimumLength = 3)]
[Required]
public string? Title { get; set; }
```

Se pueden crear atributos de validación personalizados

```
[StringLength(8, ErrorMessage = "Name length can't be more than 8.")]
```

Proyecto ASP.NET Core MVC: Modelos

Validación del Modelo en su Creación o Edición - Formulario

La validación del formulario se realiza en el lado cliente, y este framework utiliza dos tecnologías para realizar su funcionalidad de validación por defecto.

Estas tecnologías son:

- **Bootstrap**: se utiliza para maquetar los elementos del formulario y habilitar espacios donde mostrar los mensajes de error pertinentes.
- **JQuery**: permite la comprobación de los datos y mostrado de mensajes de error de forma dinámica.

Veamos como queda un formulario de creación de un modelo según indica la documentación oficial del framework.

Proyecto ASP.NET Core MVC: Modelos

Validación del Modelo en su Creación o Edición - Formulario

Estructura de GRID de Bootstrap

Clases de Bootstrap para dar estilos a los elementos del formulario

```
<div class="row">
    <div class="col-md-4">
        <form asp-action="<actionName>">

            <div>
                <label asp-for="<ModelFieldName>" class="control-label"></label>
                <input asp-for="<ModelFieldName>" class="form-control" />
                <span asp-validation-for="<ModelFieldName>" class="text-danger"></span>
            </div>

            <input type="submit" value="Crear" />

        </form>
    </div>
</div>
```

Elemento para mostrar mensaje de error en la validación si lo hubiera

Proyecto ASP.NET Core MVC: Modelos

Validación del Modelo en su Creación o Edición - Acción POST

La validación de datos recibidos en el servidor es una acción imprescindible, ya que es posible saltarse las validaciones del lado cliente.

En la acción (de tipo POST) que recibe el modelo con los datos del formulario se pueden realizar dos tipos de comprobaciones:

- Que los datos que trae el modelo generado con los datos del formulario cumple con todas las restricciones definidas en el modelo mediante los atributos de validación
- Que los datos recibidos vienen de un formulario legítimo, generado en el servidor y procedente del mismo dominio.

Veamos un ejemplo:

Proyecto ASP.NET Core MVC: Modelos

Validación del Modelo en su Creación o Edición - Acción POST

```
<input name="__RequestVerificationToken" type="hidden"
value="CfDJ8Pk9KgjAcJ1Iq9pANN2zlb2sRxKDa_X_9LfKK7hY6BSQK1fPek0eRrTI…yTv-
a9j_a7wCaNHWA4hT2hfuTrw1lPXr1HriJ7IZ1ZX49B6mx-pd1aax2fo4">
```

```
[HttpPost]
[ValidateAntiForgeryToken]
0 referencias
public IActionResult Create(<ModelClassName> model)
{
    if (ModelState.IsValid)
    {
        /*Realizar las acciones pertinentes
        cuando el modelo recibido es válido*/
    }

    return View(model);
}
```

Verifica la recepción de un token válido, el cual fue incluido en el formulario en un elemento *input* oculto

Comprueba la validez de los datos recibidos

Se retorna a la vista en caso de que exista un error de validación en los datos recibidos

Convention over Configuration

- Concepto popularizado por Ruby On Rails
- Significa que si seguimos unas reglas preestablecidas, nos vamos a ahorrar mucho trabajo de configuración, dado que si nos atenemos a las reglas muchas cosas funcionarán (se enlazarán) de forma automática
- En MVC tenemos tres directorios base
  - Controllers
  - Models
  - Views
- No los definimos en ningún fichero de configuración, vienen por defecto y siguiendo las convenciones de .Net MVC podemos hacer muchas cosas sin necesidad de configuración

Convenios aplicables a controladores y vistas

- Convenios de ASP.Net MVC:
  - ➔Cada nombre de clase de controlador lleva el sufijo Controller: Product**Controller**, Home**Controller**… y se ubican en el directorio Controllers
  - ➔Existe un directorio Views para todas las vistas de la aplicación
  - ➔Las vistas de un controlador se ubican en un subdirectorio de Views, cuyo nombre es el del controlador (quitando el sufijo -Controller)
  - ➔Por ejemplo las vistas de **Product**Controller se encuentran en **/Views/Product**

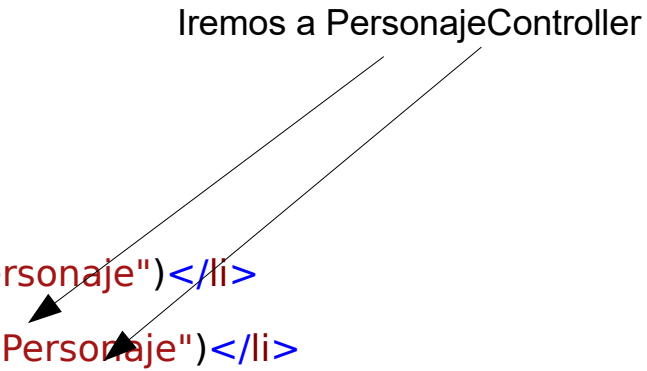
Comenzamos con un nuevo proyecto. AUT02_02_ApellidoNombre_ModernFamily

- En este nuevo proyecto desarrollaremos una actividad entregable, evaluable y calificable.
- La vista inicial personalizada. Y que se parezca a :

Data Model

- Vamos a añadir unos link al HomeController, en su vista de Index (si no la hemos creado nos ponemos en el método Index y agregar vista)

```
@{
    ViewBag.Title = "Index";
}
<h2>Menú</h2>
<ul>
<li>@Html.ActionLink("Home", "Index", "Home")</li>
<li>@Html.ActionLink("Crear personaje", "Create", "Personaje")</li>
<li>@Html.ActionLink("Lista de personajes", "Index", "Personaje")</li>
</ul>
```



Y si queremos lidiar con productos...llega el momento de **MODEL**

Modelo: Personaje

- Creamos un POCO para almacenar productos (POCO: Plain Old CLR Object) qué **poco estilo** tiene .Net ;-)

```
namespace AUT02_02_MartinezI_Listas.Models
{
    public class Personaje
    {
        [Key]
        public int Id { get; set; }
        public string Name { get; set; }
        public string Family { get; set; }
        public int NChildren { get; set; }

    }
```

Inicializar en el constructor del controlador lista de “personajes”

- Creamos una lista de Personajes; variable personajes
- Definir el método index, que devolverá la lista de personajes de Modern Family
- Añadir la vista y decir que es de tipo list y que utiliza el modelo Personaje
- Agregar luego un método Create
- El Create ¿qué hace? Queremos que nos cree un formulario y que nos recoja los datos.
- Muestra el formulario de creación o inserta un objeto.
- Hombre nos interesan dos Create, uno para la primera vez y otro para el PostBack…
- Solución

Añadimos el personaje

- Si has llegado aquí, el fin del mundo ha llegado: .Net ha triunfado y no quedan programadores en la tierra, sólo nanobots que repiten continuamente: *¿Qué deseas hacer hoy?*
- Estoooo, si hemos llegado hasta aquí y hemos insertado el producto. Una vez insertado le decimos un RedirectAction a la acción index, para que nos muestre el nuevo personaje insertado
- Oye, pero ¿veis el nuevo personaje insertado en el listado?

Proyecto ASP.NET Core MVC: Modelos

https://learn.microsoft.com/es-es/aspnet/core/mvc/models/validation?view=aspnetcore-6.0

https://learn.microsoft.com/es-es/aspnet/core/tutorials/first-mvc-app/validation?view=aspnetcore-6.0