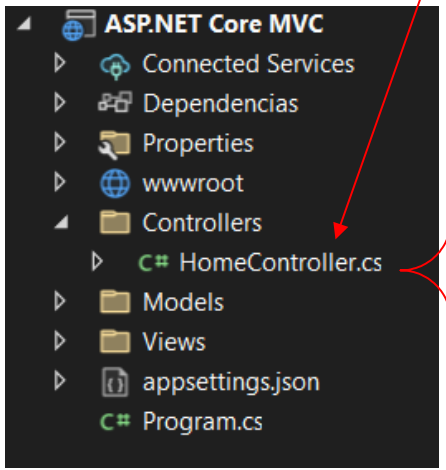
Proyecto ASP.NET Core MVC: Navegación

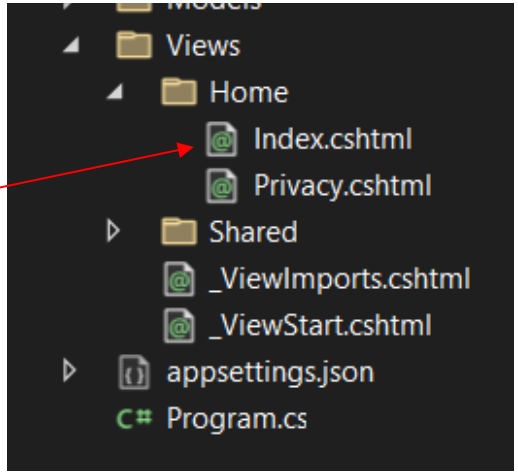
La primera vez se carga la **ruta por defecto**, que indica la acción del controlador definido en “MapControllerRoute” de “Program.cs”

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

Parámetro opcional

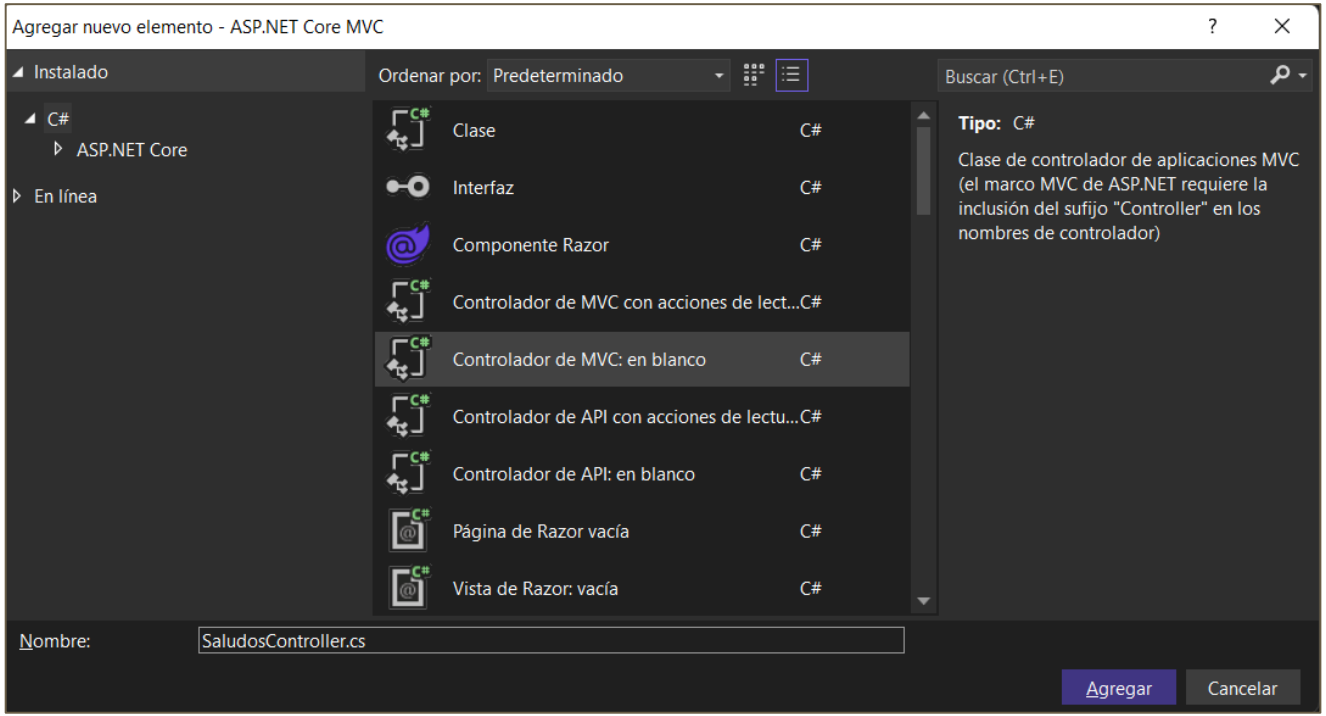



```
public class HomeController : Controller
{
    private readonly ILogger<HomeController>

    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: Controladores

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



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.

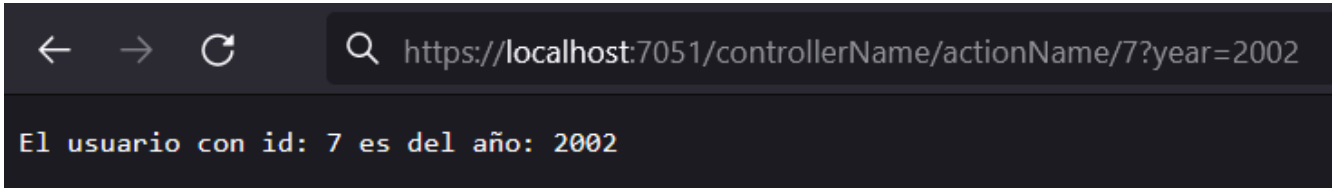
Para ello no hay que hacer nada especial, sólamente respetar los nombres de los parámetros y segmento ID.

```
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

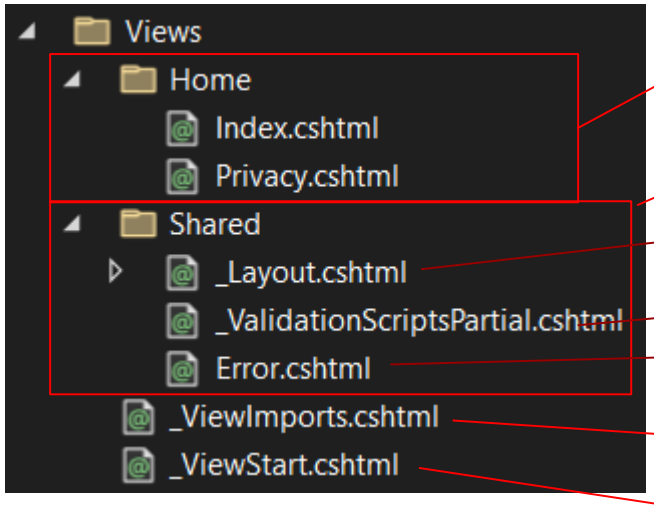
Se ha combinado interpolación y concatenación



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

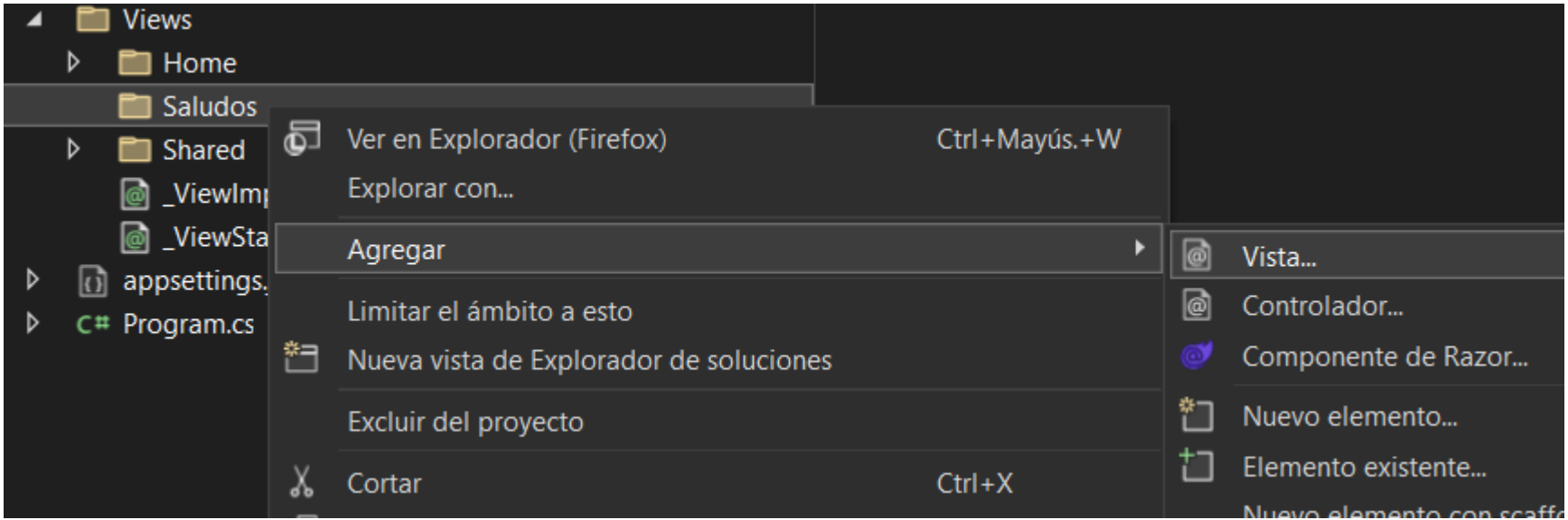
Define las librerías y helpers disponibles en las vistas

Indica cuál es el *layout* inicial de las páginas

Proyecto ASP.NET Core MVC: Vistas

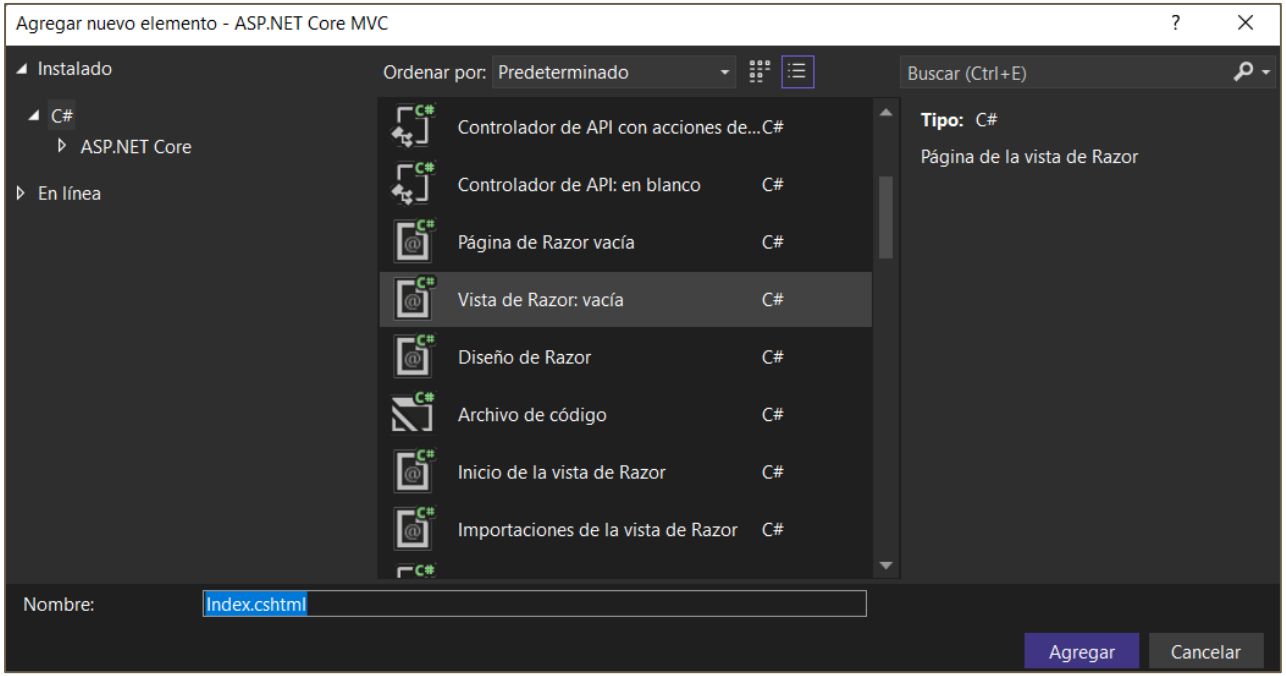
El paso siguiente será agregar una vista en el directorio Views/<controllerName>

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



Proyecto ASP.NET Core MVC: Vistas

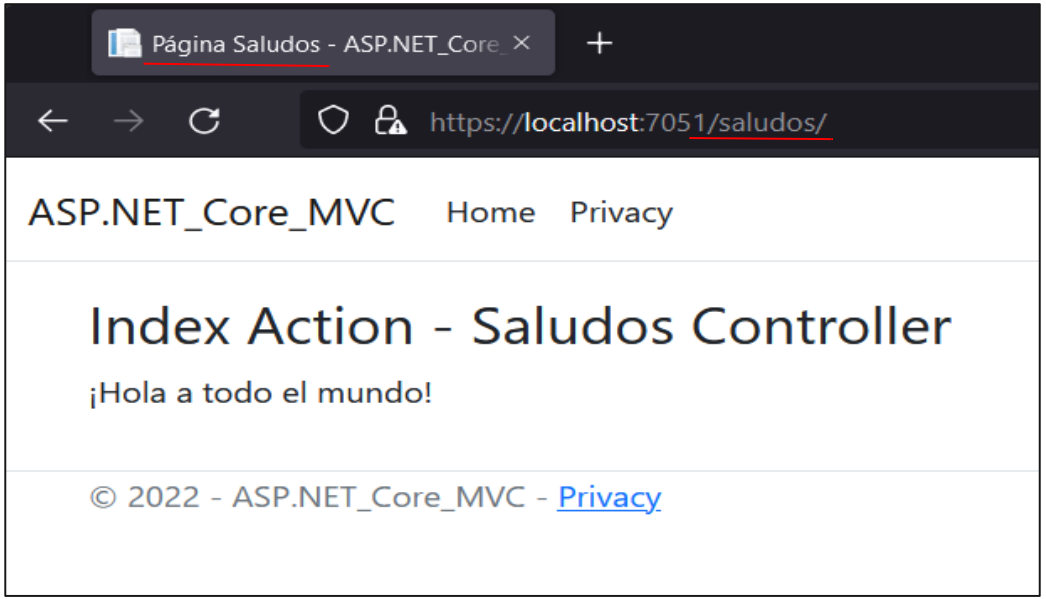
En el siguiente paso nos aseguramos que estemos creando “Vista de Razor: vacía” y que el nombre es el adecuado. Como es vista por defecto dejamos index.cshtml. Agregamos.



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

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:

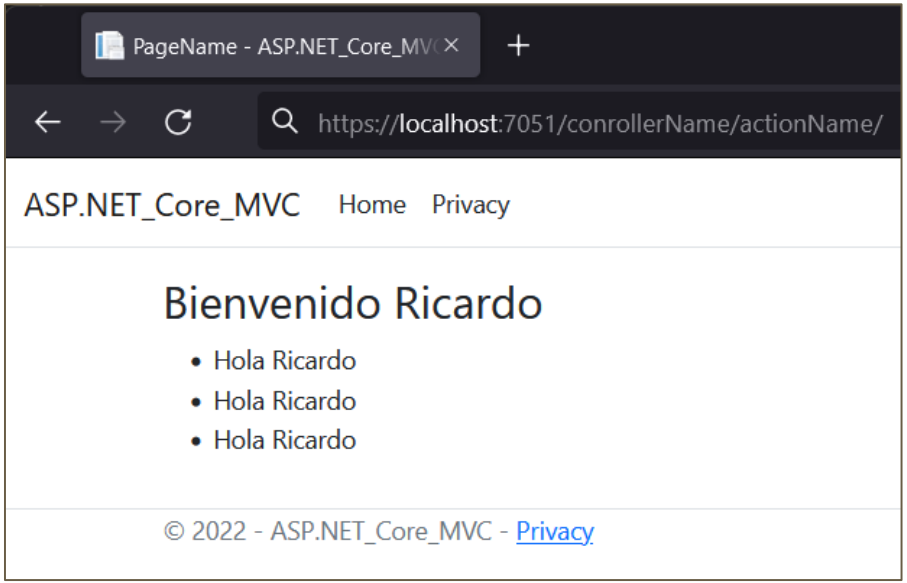
```
public IActionResult actionName()
{
    ViewData["name"] = "Ricardo";
    ViewData["numTimes"] = 3;

    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:

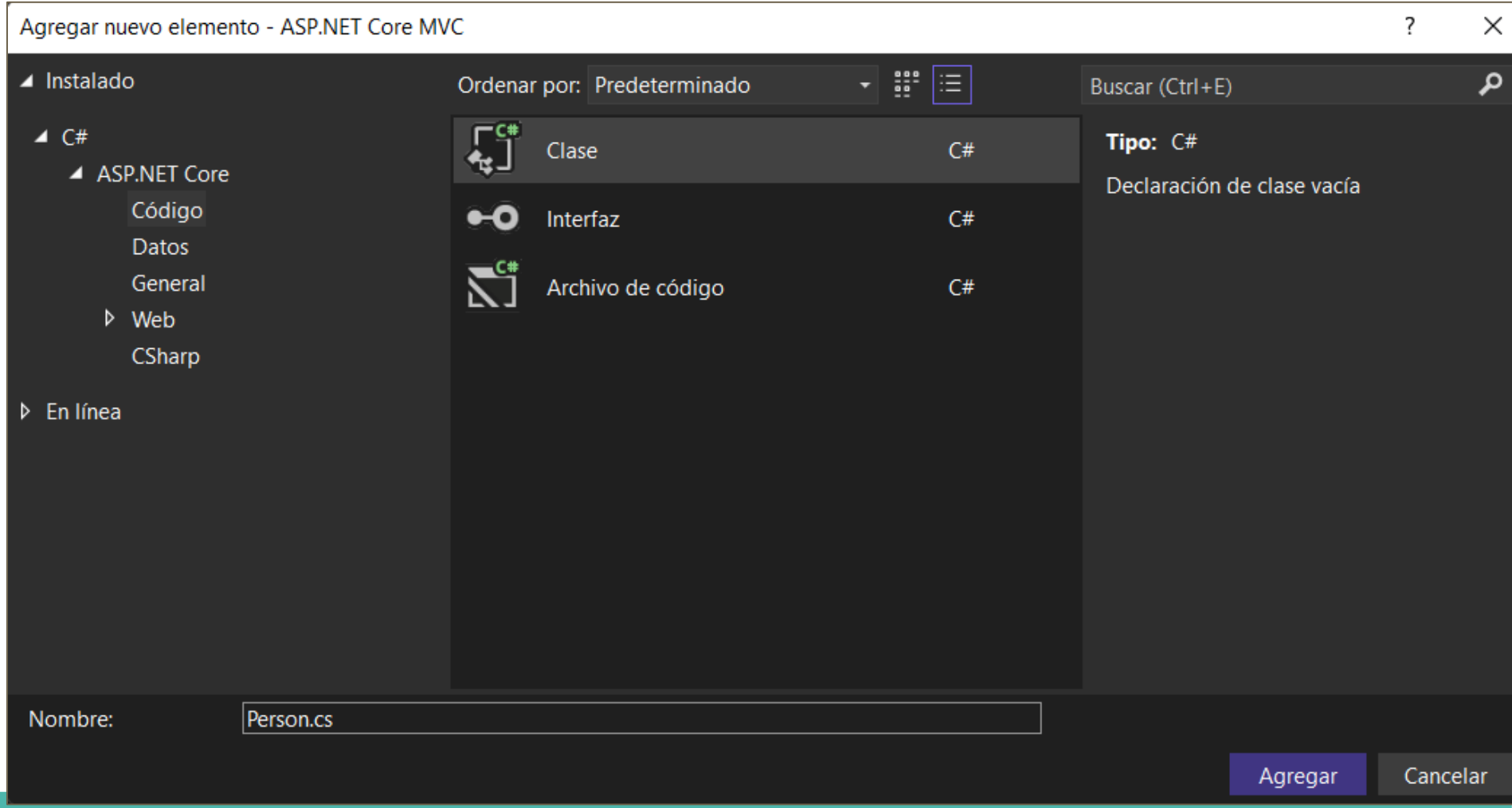
```
@{
    ViewData["Title"] = "PageName";
}

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

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



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


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

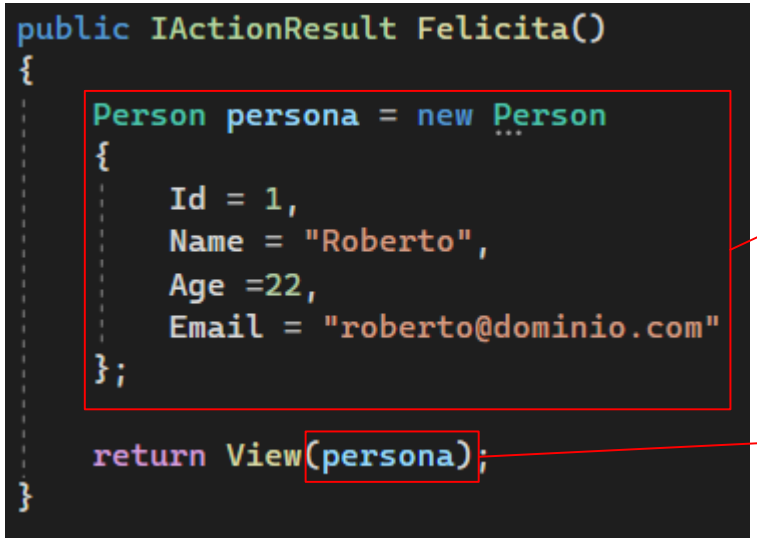
Define métodos *getter* y *setter* para ese campo

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.



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> @model Webapp.Models.NombreDelObjeto

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

Nombre del atributo de la clase Valor del atributo

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

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:

```
@using <projectName>.Models
@Model <modelName>

<form asp-action="<actionName>">

    <label asp-for="<ModelFieldName>"></label>
    <input asp-for="<ModelFieldName>" />

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

- 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

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

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

```
<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

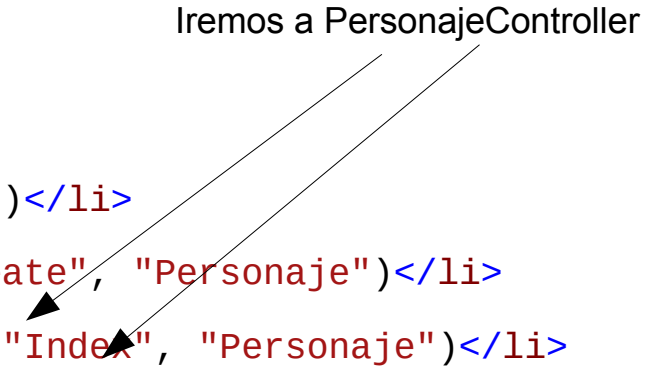
Convenios aplicables a controladores y vistas

- 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**

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; }

    }
```