

FACULTAD DE INFORMÁTICA Y CIENCIAS APLICADAS.

ESCUELA DE INFORMÁTICA.

CATEDRA DE PROGRAMACIÓN



PROGRAMACIÓN II

Proyecto de Investigación de Cátedra – Clase

CICLO: 02 – 2024

SAN SALVADOR, septiembre de 2024

Carrera	Ingeniería, Licenciatura, Técnicos		
Asignatura	Programación II	Sección:	02
Docente	Juan José Santos Escobar		
Fecha de entrega			
Unidad de aprendizaje 3: Diseño de aplicaciones web, utilizando base de datos y el modelo de Capas con MVC.			

Vistas en ASP.NET MVC.

Los desarrolladores invierten mucho tiempo enfocándose en creación de controladores y modelos de objetos bien hechos y por una buena razón, porque el código limpio y bien escrito en estas áreas forma la base de una aplicación web sostenible.

Pero cuando un usuario visita la aplicación web en un navegador, nada de este trabajo es visible. La primera impresión e interacción del usuario con la aplicación empieza con la vista. Por tanto, la vista es efectivamente el primer embajador de la aplicación ante el usuario.

Evidentemente si el resto de la aplicación tiene errores no hay cantidad de palabras ni maquillaje que compense esas deficiencias. De la misma manera, construir una vista poco atractiva y difícil de usar provoca que los usuarios no le den a la aplicación la posibilidad de demostrar la riqueza de características y que se encuentre libre de errores.

La vista es responsable de proporcionar la interface de usuario. Después que el controlador ha ejecutado la lógica apropiada para la URL solicitada, delega la presentación a la vista. A diferencia de frameworks basados en archivos (como ASP.NET y PHP), las vistas no son directamente accesibles: no se puede decir al navegador que vaya a una vista y obtenerla. En lugar de eso, una vista es siempre generada por un controlador, el cual proporciona los datos que la vista mostrará.

En algunos casos sencillos la vista necesita poca o ninguna información del controlador. Más a menudo, el controlador necesita proporcionar alguna información a la vista, por lo que le pasa un objeto de transferencia de datos llamado modelo. La vista transforma el modelo en un formato listo para ser presentado al usuario. En ASP.NET MVC, la vista logra esto examinando el objeto modelo entregado por el controlador y transforma el contenido en HTML.

Nota: No todas las vistas producen HTML. HTML es ciertamente el caso más común cuando se construye aplicaciones web. Pero las vistas también pueden producir una amplia variedad de otros tipos de contenido.

Entendiendo las vistas.

ASP.NET MVC no incluye nada que corresponda directamente a una página. En una aplicación ASP.NET MVC no hay una página en el disco que corresponda a la ruta en el URL que usted escribe en la barra de direcciones del navegador. **Lo más cercano a una página en ASP.NET MVC es algo llamado vista.**

En una aplicación ASP.NET MVC, las solicitudes recibidas en el navegador son mapeadas hacia los métodos de acción de los controladores que podrían devolver

una vista. Sin embargo, un método de acción también puede ejecutar algún otro tipo de acción tal como redireccionar hacia otro método de acción.

Una vista es un documento HTML o XHTML estándar que puede contener scripts, los cuales se usan para agregar contenido dinámico a la vista.

Crear una vista.

A continuación, se modificará el método *Index()* de la clase *DemoController* para que utilice archivos de vista Razor, lo que encapsula limpiamente el proceso de generar respuestas HTML a un cliente.

Las plantillas de vista son creadas usando Razor, y tienen las características siguientes:

- Tienen la extensión de archivo .cshtml.
- Proporcionan una forma elegante de crear salidas HTML con C#.

Actualmente, el método *Index()* devuelve una cadena con un mensaje, por lo cual se deberá reemplazar con el código siguiente:

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

El código anterior:

- Llama la vista del método controlador.
- Utiliza una plantilla de vista para generar una respuesta HTML.

Métodos del controlador:

- Se refiere a los métodos de acción, por ejemplo, el método de acción *Index()* en el código anterior.
- Generalmente devuelve un tipo de datos *IActionResult* o una clase derivada de *ActionResult*, no una cadena.

Adicionalmente, en el archivo Program.cs modifique la ruta de acceso para que el método *Index()* sea invocado al iniciar la aplicación.

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

Haga clic derecho sobre el método *Index()* de la clase *DemoController* y seleccione la opción *Agregar vista...*, como se muestra en la fig. 1.

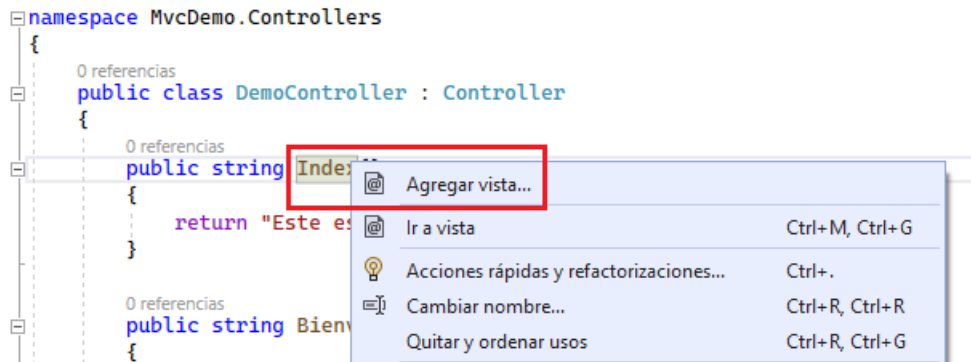


Fig. 1 – Insertar una vista.

En el cuadro de diálogo *Agregar nuevo elemento con scaffolding* seleccione la opción *Vista de Razor: vacía* y presione el botón *Agregar*, como se muestra en la fig. 2.

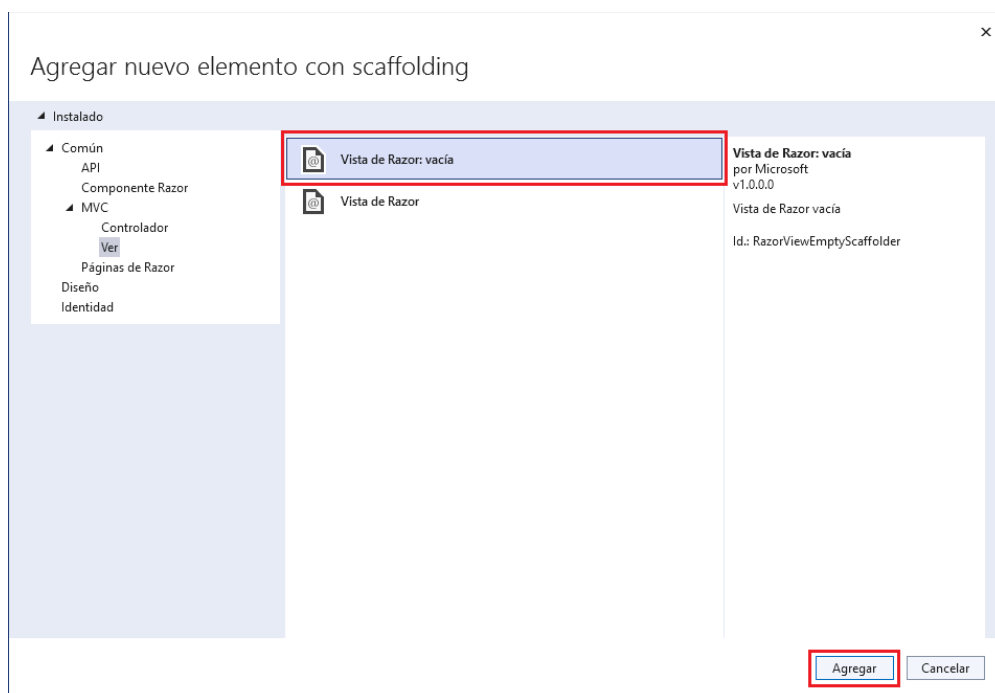


Fig. 2 – El cuadro de diálogo *Agregar nuevo elemento con scaffolding*.

Ahora aparece el cuadro de diálogo *Agregar nuevo elemento* con la opción *Vista de Razor: vacía* ya seleccionada. Mantenga el nombre sugerido y haga clic sobre el botón *Agregar*, como se muestra en la fig. 3.

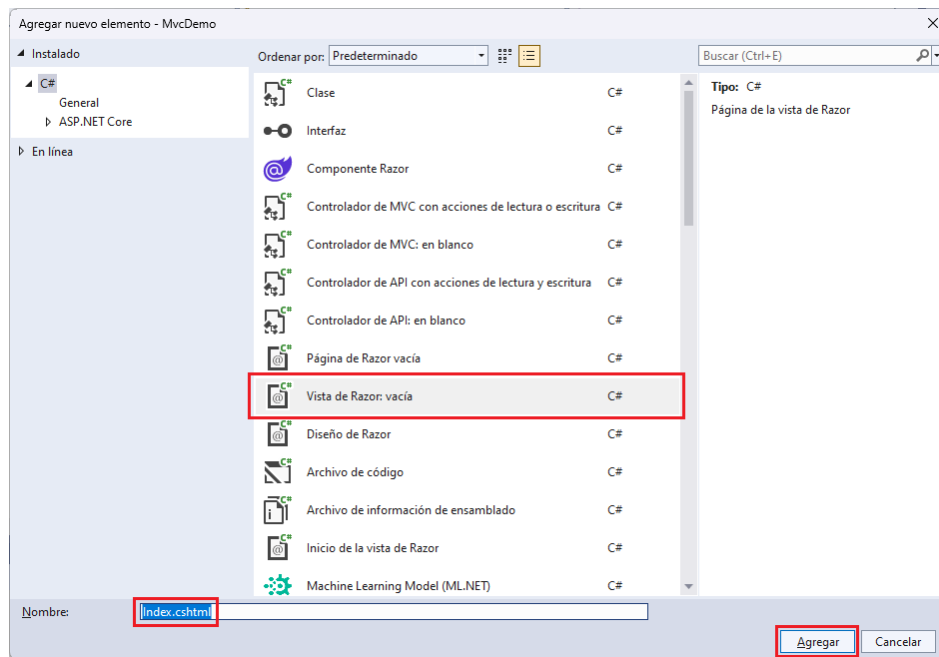


Fig. 3 – El cuadro de diálogo Agregar nuevo elemento.

Dentro de la carpeta *Views* del *Explorador de soluciones* aparece la carpeta *Demo* con el archivo *Index.cshtml*, como se muestra en la fig. 4.

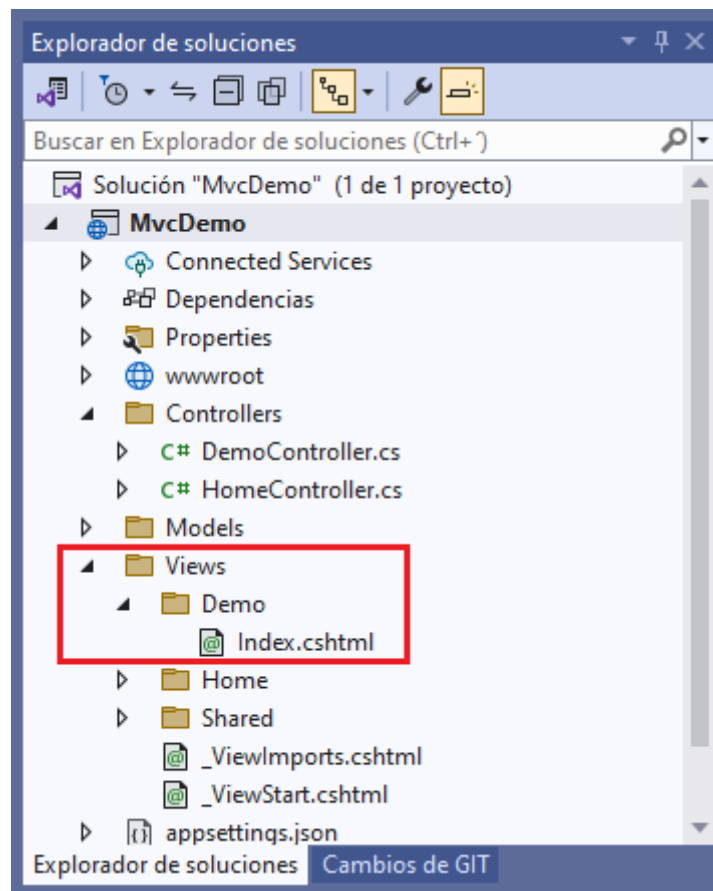


Fig. 4 – El contenido de la carpeta Views.

Reemplace el código del archivo *Index.cshtml* con el código siguiente:

```
@{  
    ViewData["Title"] = "Index";  
}  
  
<h2>Index</h2>  
  
<p>Hola desde la plantilla View! </p>
```

Ejecute la aplicación. Debería obtener una pantalla similar a la fig. 5.

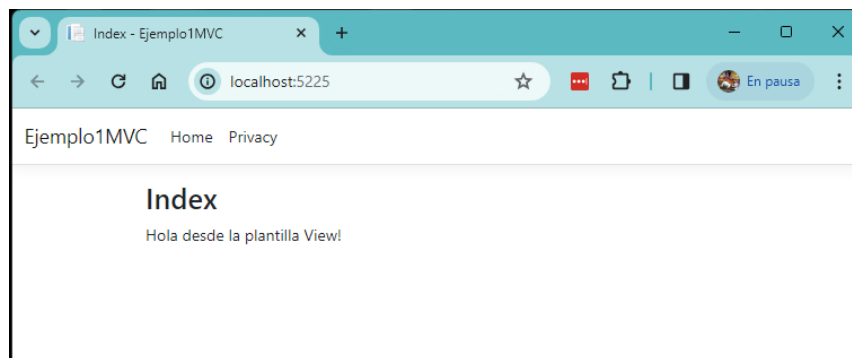


Fig. 5 – Pantalla mostrada al ejecutar la aplicación.

Enviar datos desde el controlador hacia la vista.

Las acciones del controlador son invocadas en respuesta a una solicitud URL de entrada. Una clase controladora es donde el código escrito maneja las solicitudes provenientes de un navegador web. El controlador recupera datos de un origen de datos y decide el tipo de respuesta regresará al navegador. Las plantillas de vista pueden ser usadas desde un controlador para generar y dar formato a una respuesta HTML para el navegador.

Los controladores son responsables de proporcionar los datos requeridos para que una plantilla genere una respuesta.

Las plantillas de vista **NO** deben:

- Ejecutar la lógica del negocio.
- Interactuar directamente con la base de datos.

Una plantilla de vista debería trabajar sólo con los datos proporcionados por el controlador. Mantener esta “separación de áreas de interés” ayuda a mantener el código:

- Limpio.
- Comprobable.
- Mantenible.

Actualmente, el método *Bienvenida* de la clase *DemoController* toma como parámetros un nombre y un id para mostrar los valores directamente en el navegador.

En lugar de hacer que el controlador genere esta respuesta como una cadena, se cambiará el código para que use una plantilla de vista que genera una respuesta dinámica, lo que significa que los datos apropiados deben ser enviados desde el controlador hacia la vista para generar una respuesta. Esto se logra haciendo que el controlador coloque los datos dinámicos (parámetros) que la plantilla de vista necesita en un diccionario *ViewData*. La plantilla de vista puede entonces acceder a los datos dinámicos.

El diccionario *ViewData* es un objeto dinámico, lo que significa que se puede usar cualquier tipo, y no tiene propiedades definidas hasta que se agrega algo. El sistema de enlace de modelos MVC asigna automáticamente los parámetros de la cadena de consulta a los parámetros del método.

Coloque el siguiente código en el método *Bienvenida*:

```
public IActionResult Bienvenida(string nombre, int id = 1)
{
    ViewData["Mensaje"] = "Hola " + nombre;
    ViewData["Veces"] = id;
    return View();
}
```

El objeto diccionario *ViewData* contiene datos que se pasarán a la vista.

Construya una plantilla de vista para el método de acción *Bienvenida()* y coloque un ciclo que muestre "Hola" la cantidad de veces indicada, para lo cual reemplace el contenido [Bienvenida.cshtml](#) con lo siguiente:

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

<h2>Bienvenida</h2>

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

Ejecute la aplicación.

Use el URL `http://localhost:5078/Demo/Bienvenida`. Obtendrá la pantalla mostrada en la fig. 6.

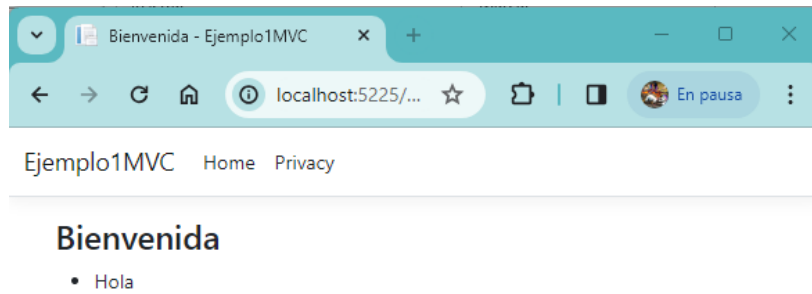


Fig. 6 – Pantalla mostrada al ejecutar la aplicación y el método `Bienvenida()`.

Ahora use el URL `http://localhost:5078/Demo/Bienvenida?nombre=Juan&id=4`. Obtendrá la pantalla mostrada en la fig. 7.

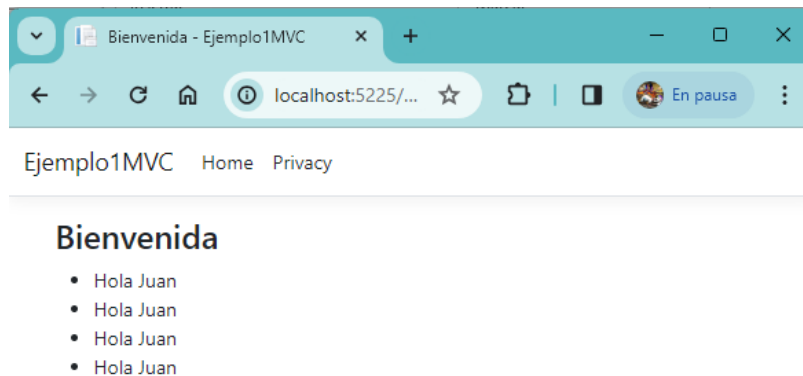


Fig. 7 – Pantalla mostrada al ejecutar la aplicación, usar el método `Bienvenida()` y enviar parámetros.

En la clase `DemoController` agregue el método `Países()` y coloque el código siguiente:

```
public List<string> Países()
{
    return new List<string>()
    {
        "Belize",
        "Guatemala",
        "El Salvador",
        "Honduras",
        "Nicaragua",
```

```

    "Costa Rica",
    "Panama"
  };
}

```

Cambie la ruta de acceso definida en el archivo Program.cs con el código siguiente:

```

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

```

Al ejecutar la aplicación se obtiene un resultado similar al mostrado en la fig. 8.

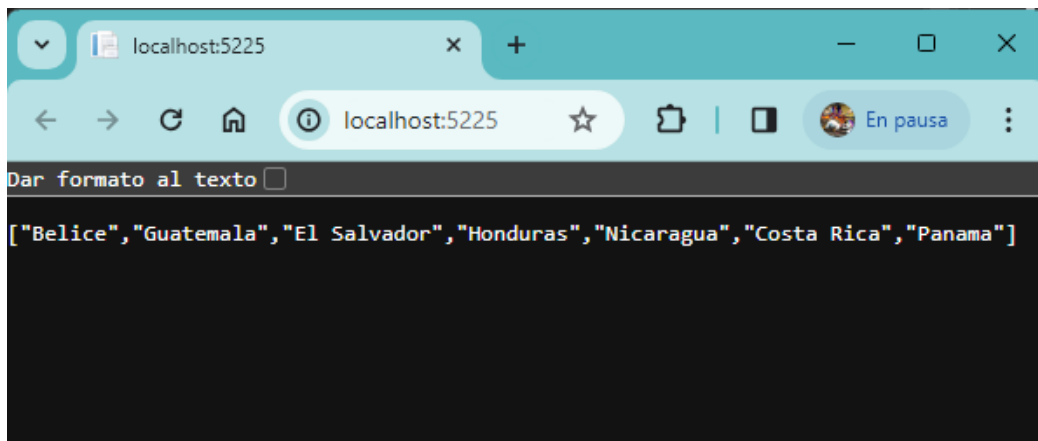


Fig. 8 – Mostrar el contenido de la lista.

Active la opción *Pretty-print*, en caso de que el navegador tenga esta opción para obtener una pantalla similar a la fig. 9.

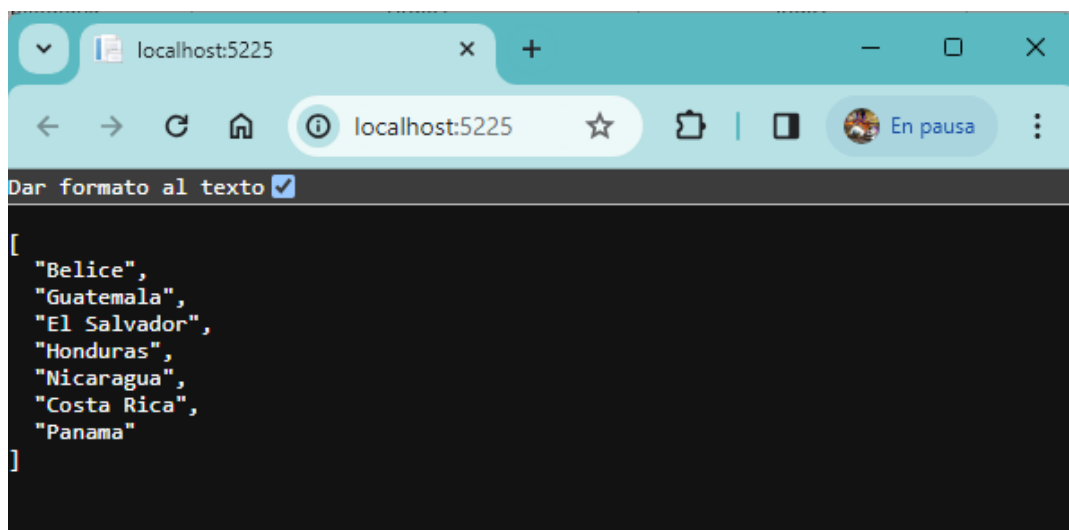


Fig. 9 – Mostrar el contenido de la lista aplicando la opción *Pretty-print*.

Claramente no es el resultado esperado. Para cambiar esta situación, modifique el método **Países** del controlador **DemoController** con el código mostrado:

```

public IActionResult Paises()
{
    ViewBag.Paises = new List<string>()
    {
        "Belice",
        "Guatemala",
        "El Salvador",
        "Honduras",
        "Nicaragua",
        "Costa Rica",
        "Panama"
    };

    return View();
}

```

Agregue una vista de nombre **Paises()** y coloque el código siguiente:

```

@{
    ViewBag.Title = "Listado de paises";
}

<h2>Listado de paises</h2>
<ul>
    @foreach (string pais in ViewBag.Paises)
    {
        <li>pais</li>
    }
</ul>

```

Ejecute la aplicación. Ahora se espera obtener el listado de países, pero se obtiene una pantalla similar a la fig. 10.

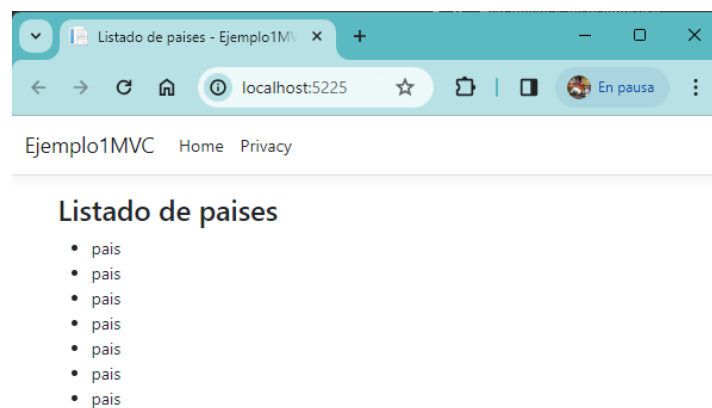


Fig. 10 – Pantalla mostrada al ejecutar la aplicación y el método Paises().

Modifique el código de la vista de la manera siguiente:

```
@{
    ViewBag.Title = "Listado de países";
}

<h2>Listado de países</h2>
<ul>
    @foreach (string pais in ViewBag.Paises)
    {
        <li>@pais</li>
    }
</ul>
```

Ejecute la aplicación. El resultado es mostrado en la fig. 11:

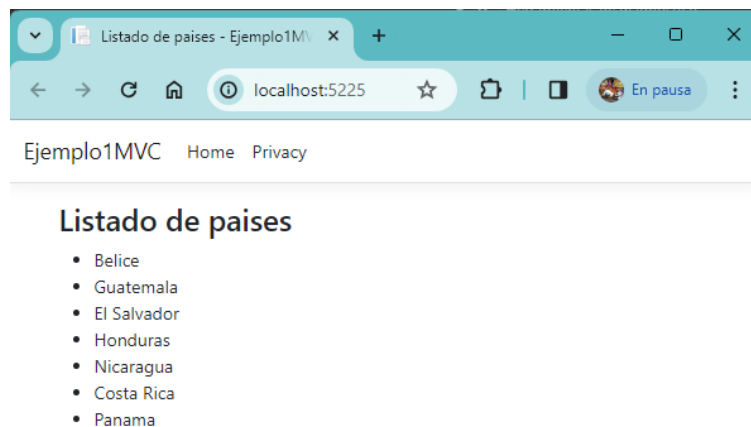


Fig. 11 – Pantalla con el código modificado.

En el código de la vista se detectan dos situaciones:

- Se está declarando la entrada Title en el diccionario ViewBag, pero no se está usando.
- Si el diccionario está disponible en el controlador, ¿es posible declarar ahí la entrada Title en lugar de hacerlo en la vista?

Para atender estas situaciones, en el código de la vista realice los siguientes cambios:

- Sustituya el texto de la etiqueta <h2>.
- Elimine la entrada Title que se encuentra al inicio del código.

El código debería ser similar al mostrado:

```
<h2>@ViewBag.Title</h2>
<ul>
    @foreach (string pais in ViewBag.Paises)
    {
        <li>@pais</li>
    }
</ul>
```

En el método países del controlador Demo agregue la entrada Title al diccionario (y también coloque el acento a Panamá), como se muestra:

```
public IActionResult Paises()
{
    ViewBag.Title = "Listado de países";
    ViewBag.Paises = new List<string>()
    {
        "Belice",
        "Guatemala",
        "El Salvador",
        "Honduras",
        "Nicaragua",
        "Costa Rica",
        "Panamá"
    };
    return View();
}
```

El motor de vistas Razor.

ASP.NET MVC incluye dos diferentes motores de vista: el nuevo motor de vistas Razor y el viejo motor de vistas Web Forms. Debido a que Web Forms es eminentemente código HTML, se creó el motor de vistas Razor que posee las siguientes características:

1. Es un motor de vistas sencillo, limpio y ligero que no contiene sintaxis que propicie la generación de código mal diseñado, innecesariamente complicado o no deseado.
2. Proporciona una sintaxis coordinada que minimiza la cantidad de sintaxis y caracteres extra al entender la estructura de marcación de forma tal que la transición entre código y marcación sea lo más suave posible.

Expresiones de código.

El carácter clave de transición en Razor es **el signo arroba (@)**. Este único carácter **es usado para hacer la transición entre el lenguaje de marcación y el código**, e incluso a veces también para hacer una transición inversa. Los dos tipos básicos de transiciones son expresiones de código y bloques de código. Las expresiones son evaluadas y escritas en la respuesta. Por ejemplo, en el siguiente fragmento:

```
<h1>Mostrando @Países.Length elementos.</h1>
```

La expresión @items.length es evaluada como expresión de código implícito y el resultado es mostrado en la salida. Una cosa a observar es que no fue necesario demarcar el final de la expresión de código. En contraste, con una vista Web Forms que soporta sólo expresiones de código explícitas habría quedado de la siguiente manera:

```
<h1>Mostrando <% Países.Length %> elementos.</h1>
```

Razor sabe que el carácter de espacio después de la expresión no es un identificador válido, por lo que realiza una suave transición de vuelta hacia el lenguaje de marcación.

Observe que el carácter después de la expresión de código **@item** es un carácter de código válido, por lo cual Razor revisa el carácter siguiente y encuentra un paréntesis angular que no es un carácter válido y automáticamente hace una transición a lenguaje de marcación.

Esta habilidad de hacer transiciones automáticas entre código C# y lenguaje de marcación es uno de los mayores recursos de Razor para mantener la sintaxis compacta y limpia.

Sintaxis de Razor.

Las principales reglas de sintaxis para Razor con C# son las siguientes:

- Los bloques de código Razor se encuentran encerrados con @{ ... }.
- Las expresiones inline (variables y funciones) inician con @.
- Las sentencias de código terminan con punto y coma.
- Las variables son declaradas con la palabra reservada var.
- Las cadenas están encerradas con comillas dobles.
- El código C# es sensible a mayúsculas y minúsculas.
- Los archivos C# tienen la extensión .cshtml.

Modelos en ASP.NET MVC.

La palabra modelo en el desarrollo de software está sobrecargada para cubrir cientos de diferentes conceptos. Hay modelos de madurez, modelos de diseño, modelos de amenaza y modelos de procesos. El asistir a una reunión de desarrollo sin hablar sobre un modelo de uno u otro tipo es raro. Aun cuando limita el alcance del término modelo al contexto del patrón de diseño MVC se puede todavía debatir los méritos de tener un modelo de objetos orientado a negocios versus un modelo de objetos específico de vistas. Este material habla sobre modelos como los objetos que se usan para enviar información a la base de datos, desarrollar cálculos de negocios e incluso mostrarlos en una vista. En otras palabras, estos objetos representan el dominio sobre el cual se enfoca la aplicación y los modelos son los objetos que se desean mostrar, guardar, crear, actualizar y borrar.

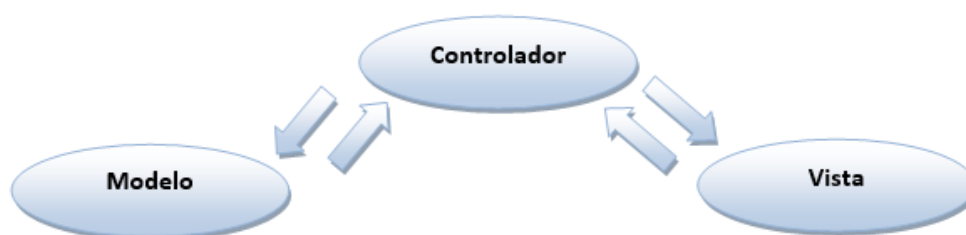
ASP.NET MVC proporciona un conjunto de herramientas y opciones para construir características de una aplicación usando solamente la definición del modelo de objetos. Se puede pensar en el problema que se desea resolver y escribir clases C# para representar los objetos primarios involucrados. A continuación, se pueden usar herramientas proporcionadas por MVC para construir los controladores y las vistas para los escenarios estándar índice, crear, editar y borrar para cada uno de los objetos del modelo. El trabajo de construcción es llamado scaffolding.

Los modelos representan el dominio específico de datos y lógica de negocios en la arquitectura MVC, ya que mantiene los datos de la aplicación. Los objetos modelo recuperan y almacenan estados del modelo en un almacén de persistencia tal como una base de datos.

Una clase de tipo Modelo guarda datos en las propiedades públicas. Todas las clases Modelo residen en la carpeta **Model** en la estructura de carpetas de MVC.

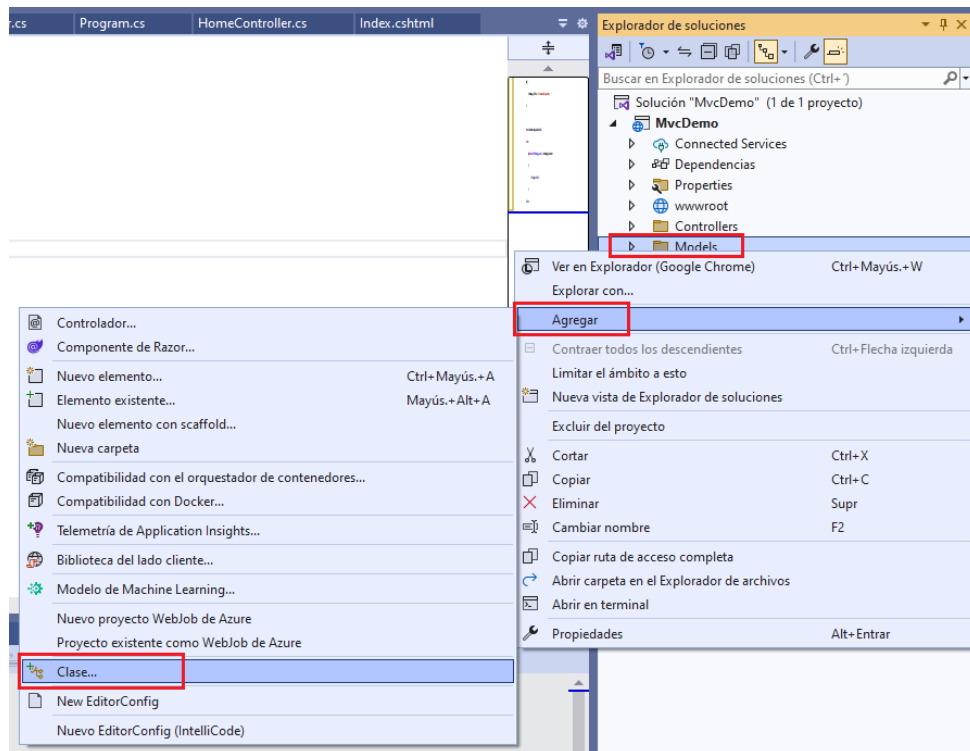
Interacción entre el modelo, la vista y el controlador.

El siguiente diagrama muestra la interacción entre los tres componentes del modelo MVC en ASP.NET:

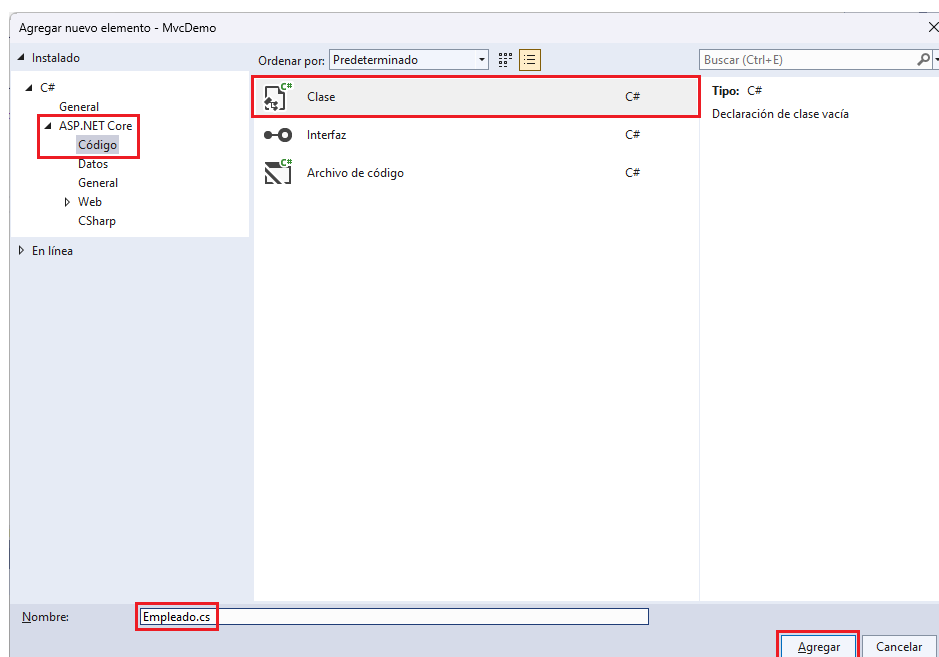


Procedimiento.

- Inicie Visual Studio y abra el proyecto *MVCDemo*.
- Agregue un modelo a la aplicación, para lo cual haga clic derecho sobre la carpeta **Models** y seleccione las opciones **Agregar, Clase**.



- Coloque el nombre *Empleado* a la nueva clase y haga clic en el botón **Add**.



- Digite el código siguiente:

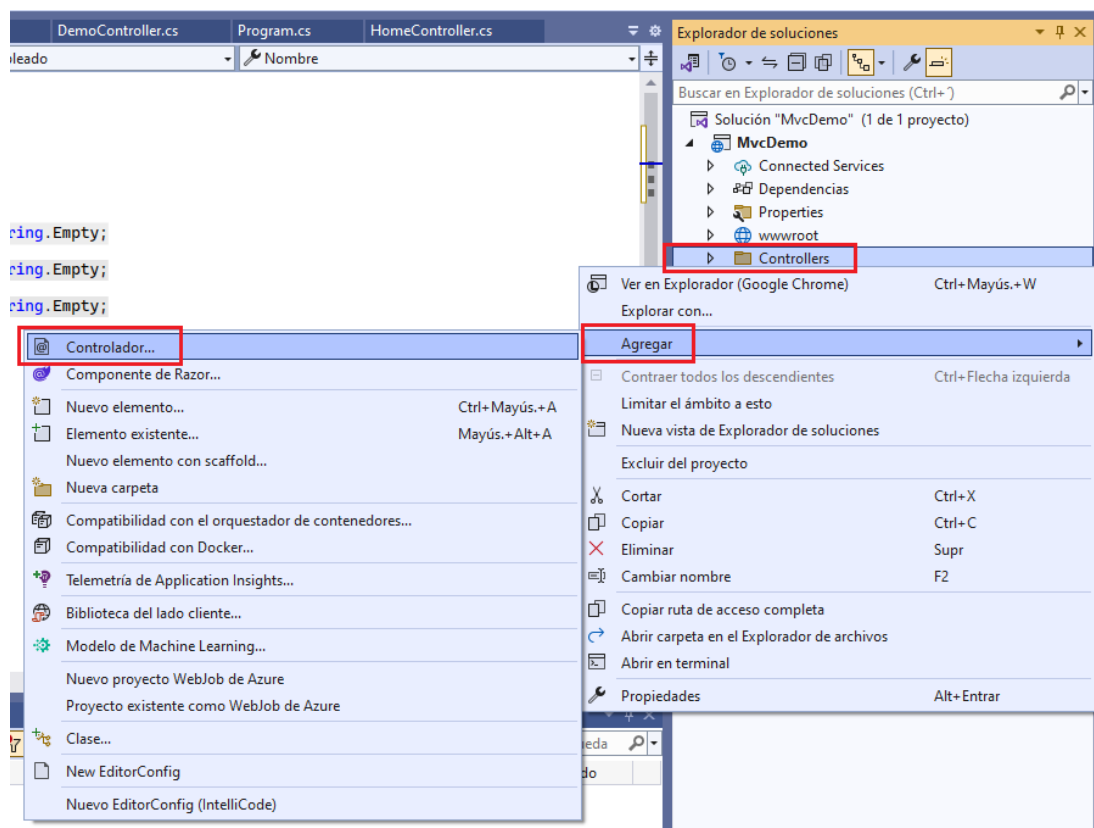
```

namespace MvcDemo.Models
{
    public class Empleado
    {
        public int IDEmpleado { get; set; }
        public string Nombre { get; set; } = string.Empty;
        public string Genero { get; set; } = string.Empty;
        public string Ciudad { get; set; } = string.Empty;
    }
}

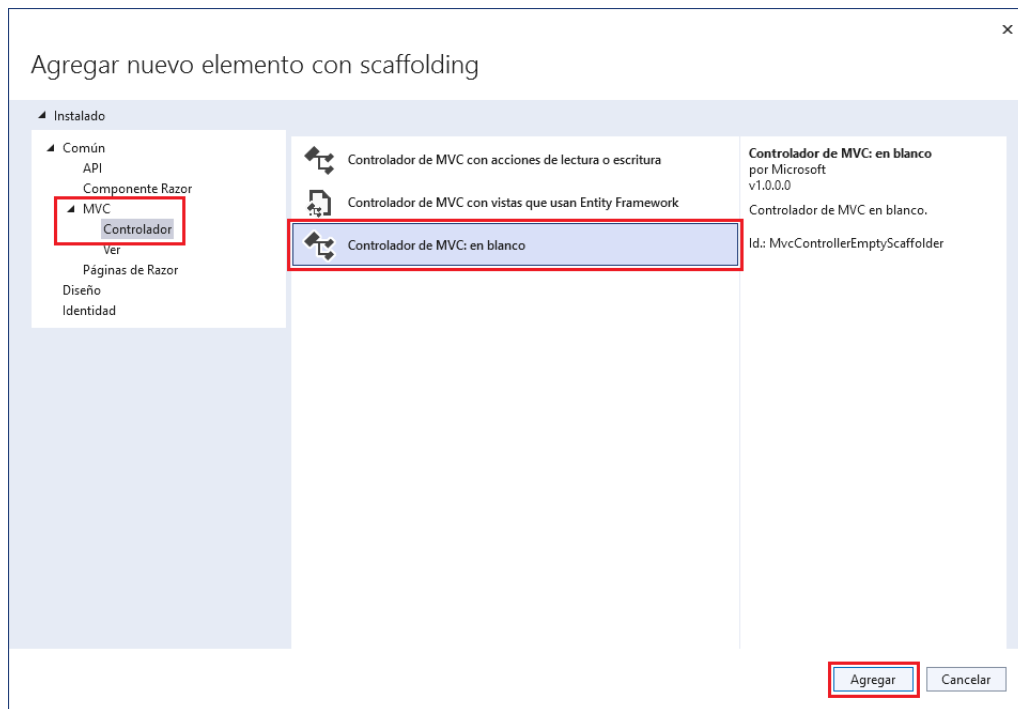
```

SUGERENCIA: Puede generar más rápidamente las propiedades si digita la palabra **prop** y a continuación presiona la tecla **TAB** dos veces.

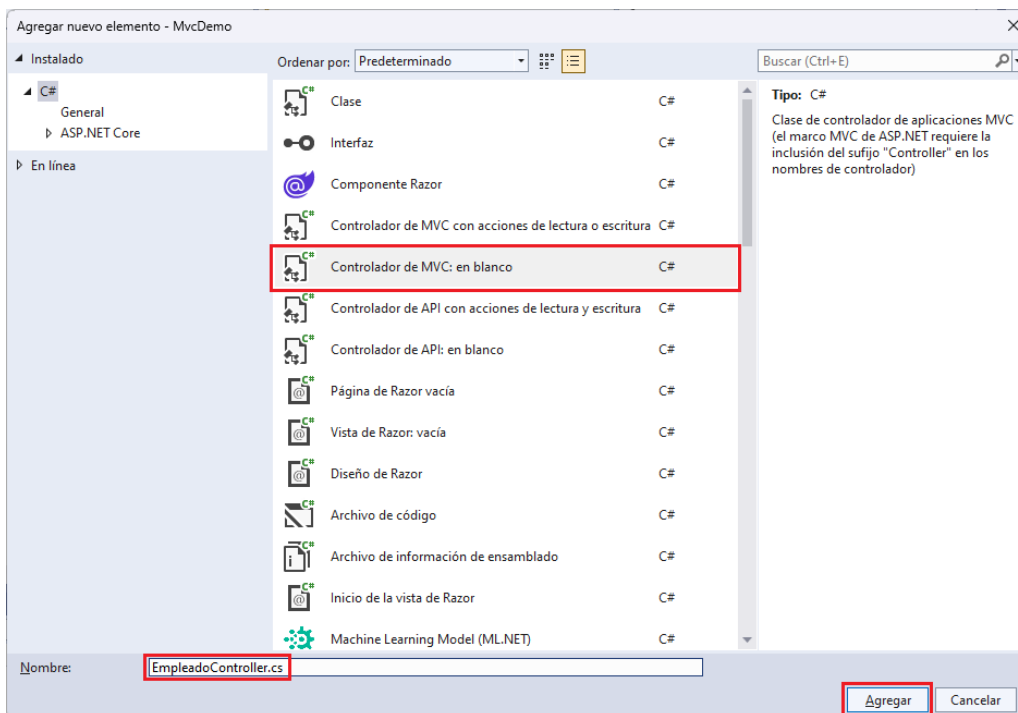
- Agregue un controlador a la aplicación, para lo cual haga clic derecho sobre la carpeta **Controllers** y seleccione las opciones **Agregar**, **Controlador**.



- Seleccione la opción **Controlador de MVC: en blanco** y haga clic sobre el botón **Agregar**.



- Coloque el nombre EmpleadoController.



- Cambie el nombre del método de acción **Index()** a **Detalle()**.

using Microsoft.AspNetCore.Mvc;

namespace MvcDemo.Controllers

{

public class EmpleadoController : Controller

```

{
    // GET: Empleado
    public IActionResult Detalle()
    {
        return View();
    }
}

```

- En este momento se debería escribir el código necesario para recuperar información de una tabla. Para efectos didácticos en este ejercicio se usará "código quemado". Escriba el código siguiente:

```

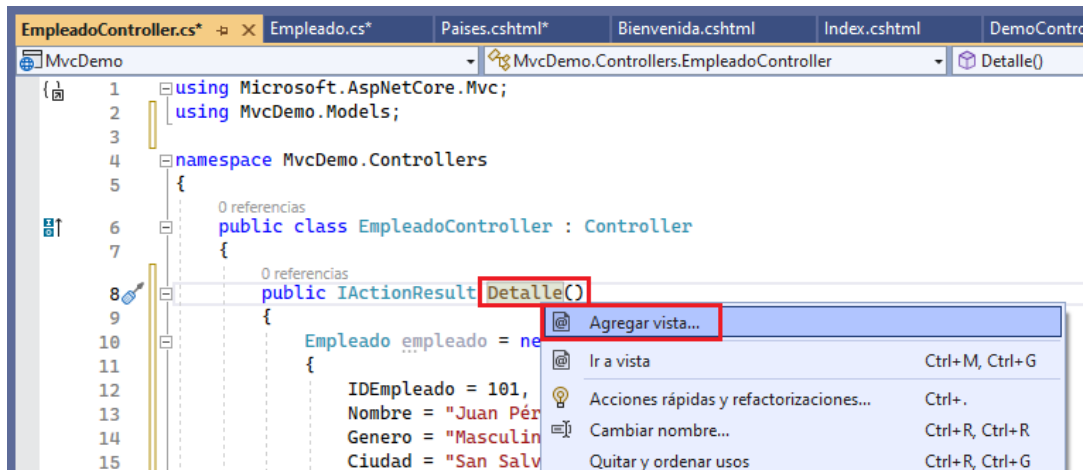
using Microsoft.AspNetCore.Mvc;
using MvcDemo.Models;

namespace MvcDemo.Controllers
{
    public class EmpleadoController : Controller
    {
        public IActionResult Detalle()
        {
            Empleado empleado = new Empleado()
            {
                IDEmpleado = 101,
                Nombre = "Juan Pérez",
                Genero = "Masculino",
                Ciudad = "San Salvador"
            };

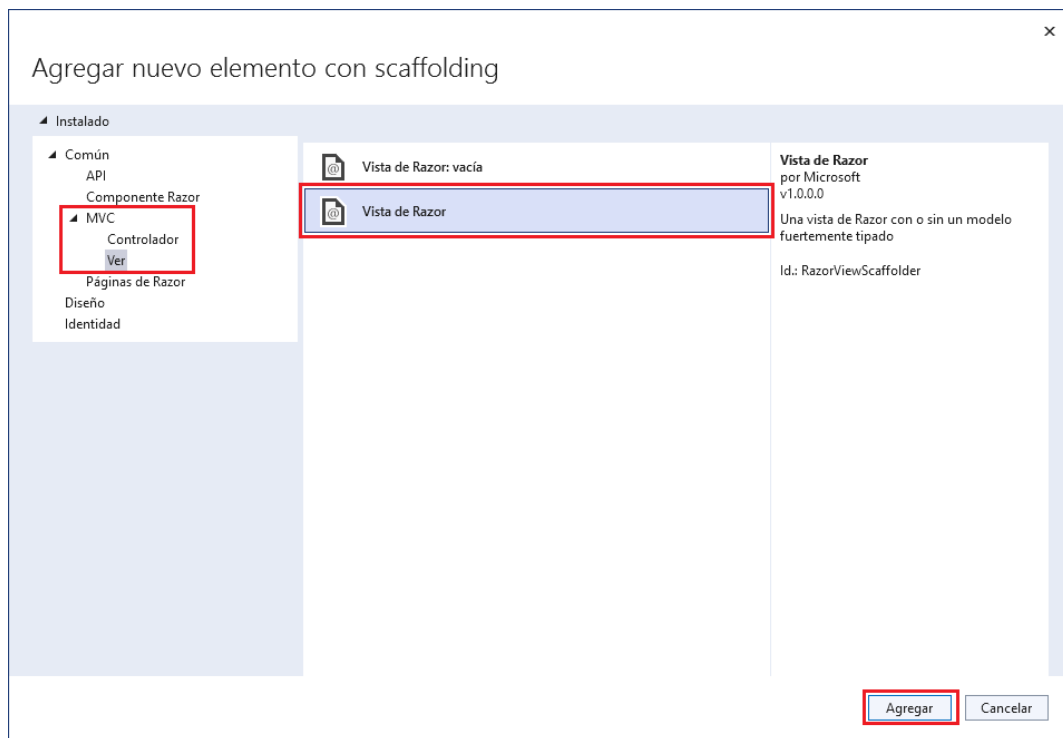
            return View();
        }
    }
}

```

- Agregue una vista. Haga clic derecho sobre el método de acción **Detalle()** y seleccione la opción **Add View**.



- En el cuadro de diálogo **Agregar nuevo elemento con scaffolding** seleccione la plantilla **Vista de Razor** y haga clic en el botón **Agregar**.



- En el cuadro de diálogo **Agregar Vista de Razor** seleccione la plantilla **Details**, la clase de modelo **Empleado (MVCPaises.Models)**, quite las marcas de las casillas de opciones y haga clic en el botón **Agregar**.

Agregar Vista de Razor

Nombre de vista:

Plantilla: Details

Clase de modelo: Empleado (MvcDemo.Models)

Opciones:

- ☐ Crear como vista parcial
- ☐ Hacer referencia a bibliotecas de scripts
- ☐ Usar página de diseño

(Dejar en blanco si se define en un archivo _viewstart de Razor)

Agregar Cancelar

- Modifique el código generado con el siguiente:

@model Ejemplo1MVC.Models.Empleado

```
@{
    ViewBag.Title = "Detalles del empleado";
}
```

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
    <meta name="viewport" content="width=device-width" />
```

```
    <title>Detalle</title>
```

```
</head>
```

```
<body>
```

```
<div>
```

```
    <h4>@ViewBag.Title</h4>
```

```
    <hr />
```

```
    <dl class="row">
```

```
        <dt class = "col-sm-2">
```

```
            @Html.DisplayNameFor(model => model.IDEmpleado)
```

```
        </dt>
```

```
        <dd class = "col-sm-10">
```

```
            @Html.DisplayFor(model => model.IDEmpleado)
```

```
        </dd>
```

```
        <dt class = "col-sm-2">
```

```
            @Html.DisplayNameFor(model => model.Nombre)
```

```
        </dt>
```

```

<dd class = "col-sm-10">
    @Html.DisplayFor(model => model.Nombre)
</dd>
<dt class = "col-sm-2">
    @Html.DisplayNameFor(model => model.Genero)
</dt>
<dd class = "col-sm-10">
    @Html.DisplayFor(model => model.Genero)
</dd>
<dt class = "col-sm-2">
    @Html.DisplayNameFor(model => model.Ciudad)
</dt>
<dd class = "col-sm-10">
    @Html.DisplayFor(model => model.Ciudad)
</dd>
</dl>
</div>
<div>
    @Html.ActionLink("Edit", "Edit",
        new { /* id = Model.PrimaryKey */ }) |
    <a asp-action="Index">Back to List</a>
</div>
</body>
</html>

```

- Si ejecuta la aplicación no se mostrarán los datos porque no se ha enviado la información correspondiente a la vista, para lo cual se debe modificar el código de la forma siguiente:

```

using Microsoft.AspNetCore.Mvc;
using MvcDemo.Models;

namespace MvcDemo.Controllers
{
    public class EmpleadoController : Controller
    {
        public IActionResult Detalle()
        {
            Empleado empleado = new Empleado()
            {
                IDEmpleado = 101,
                Nombre = "Juan Pérez",
                Genero = "Masculino",
            }
        }
    }
}

```

```

        Ciudad = "San Salvador"
    };

    return View(empleado);
}
}
}

```

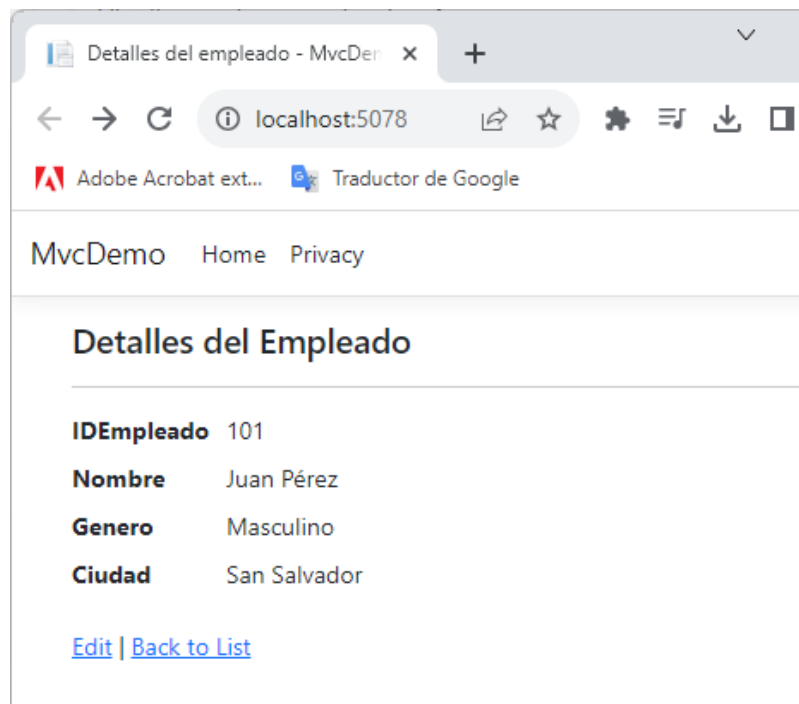
- Finalmente, cambie la configuración de la ruta de acceso, como se muestra:

```

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

```

- Ejecute la aplicación.



- Evidentemente, si hace clic sobre la opción **Edit** o sobre la opción **Back to List** obtendrá un mensaje de error debido a que no están asociados con nada, pero en este material no se trabajará con estas opciones, de modo que deberá modificar parte del código contenido en el archivo Detalle.cshtml, como se muestra:

```
@* @Html.ActionLink("Edit", "Edit", new { /* id = Model.PrimaryKey */
}) |

<a asp-action="Index">Back to List</a> *@
```

- Ejecute nuevamente la aplicación.

