

TECNICATURA
UNIVERSITARIA
EN PROGRAMACIÓN
UTN-FRC



UTN 
Facultad Regional Córdoba

TECNICATURA UNIVERSITARIA EN PROGRAMACIÓN

PROGRAMACIÓN III

Unidad Temática 3: Programación del lado del Servidor

Material de Estudio

2^{do} Año - 3^{er} Cuatrimestre

2020



V.0.1

Índice

Introducción.....	2
Patrón MVC.....	2
Creando nuestro primer proyecto.....	2
Creando modelos, vistas y controladores.	19
Programación de vistas con páginas cshtml.	25
Enumerando arreglos y colecciones	29
Definiendo el espacio de nombre namespace.....	31
Importante	32
Bibliografía	33

Introducción

Según Nick Harrison (2015), “MVC es la última oferta de Microsoft para diseñar y construir aplicaciones web, y también es un patrón arquitectónico de diseño. Como patrón arquitectónico, MVC y sus variantes han estado dando vueltas por algún tiempo pero el framework es relativamente nuevo, teniendo la ventaja de haber aprendido de otros frameworks de otros lenguajes. (...)”

En esta unidad vamos a aprender a crear páginas web con el lenguaje de programación C#. Para eso, utilizaremos el framework MVC ASP .Net.

Patrón MVC

Para entender mejor framework, tenemos que el entender el patrón de diseño en el cual está basado. MVC y sus variantes separan a una aplicación web en tres áreas distintivas: el modelo, la vista y el controlador. el modelo es el conjunto de datos que usa nuestra aplicación, la vista es la interfaz de usuario (UI por User Interface) y el controlador aloja la mayor parte de la lógica de la aplicación que une las tres piezas juntas. (Harrison, 2015)

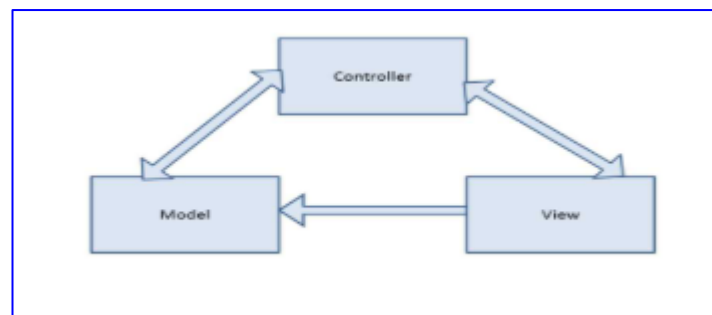


Imagen 1: Recuperado de Nick Harrison (2015), ASP.NET MVC Succinctly, Syncfusion, USA.

Esta forma de división es para separar el funcionamiento de cualquier aplicación. Es más, el controlador no debe saber cómo los datos del modelo son manipulados por las vistas. Tampoco el controlador debe saber nada acerca de cómo los datos son persistidos en la base de datos. A su vez las vistas no saben nada del controlador.

Creando nuestro primer proyecto

Para crear nuestro proyecto vamos a utilizar Visual Studio. Podríamos utilizar un Editor de texto como Visual Studio Code y la línea de comandos. La elección es indistinta.

En este caso vamos a mostrar los pasos para crear un proyecto y ejecutarlo para verlo en el navegador.

Por ser nuestro primer proyecto, no vamos a escribir ninguna línea de código sino que analizaremos lo que el framework nos provee por defecto.

Para empezar, seleccionaremos del menú archivo la opción Nuevo y a continuación Proyecto...

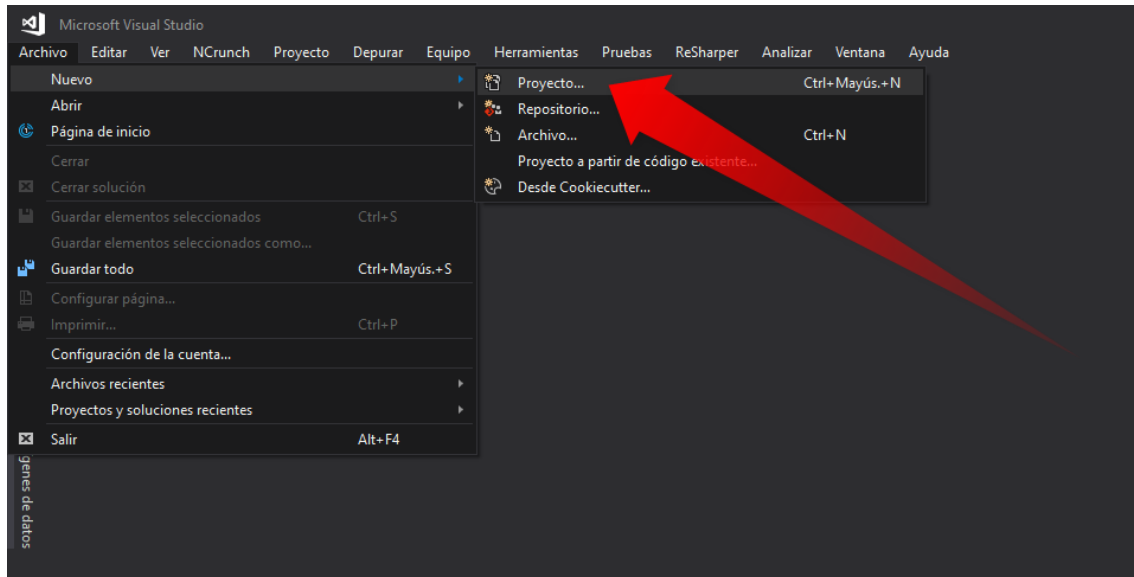


Imagen 2: fuente autoría propia

Luego seleccionamos del panel izquierdo, dentro de la categoría visual C#, Cloud y la opción Aplicación web ASP .Net

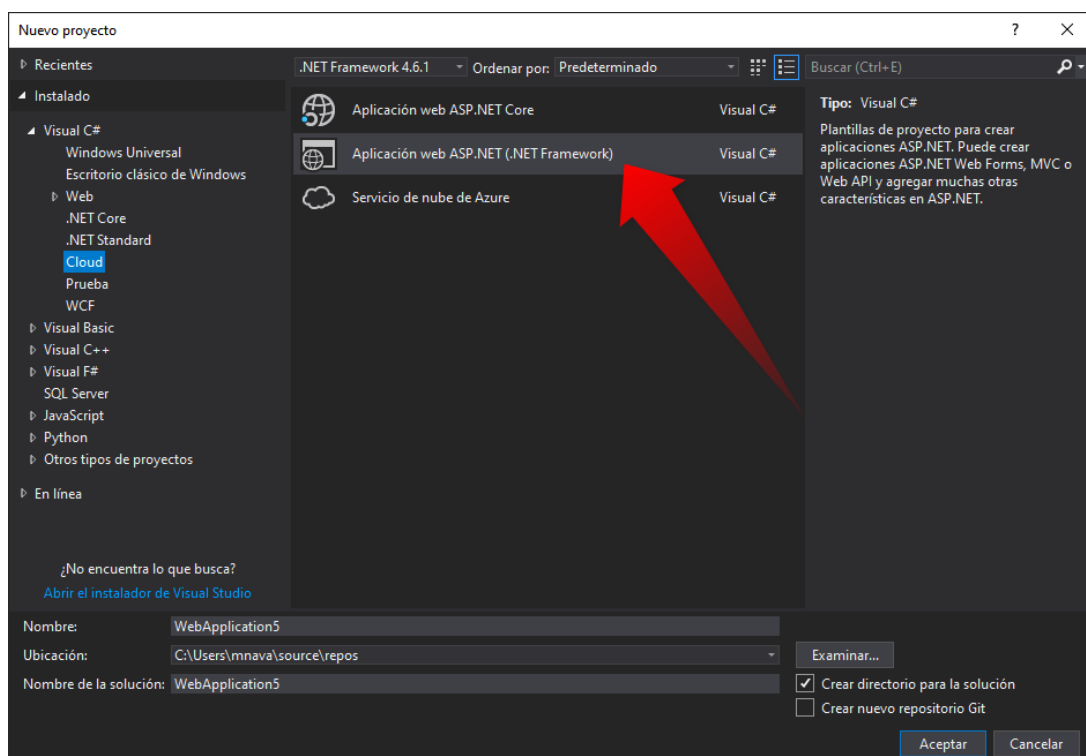


Imagen 3: fuente autoría propia

Por último de las plantillas de ASP.Net elegimos MVC.

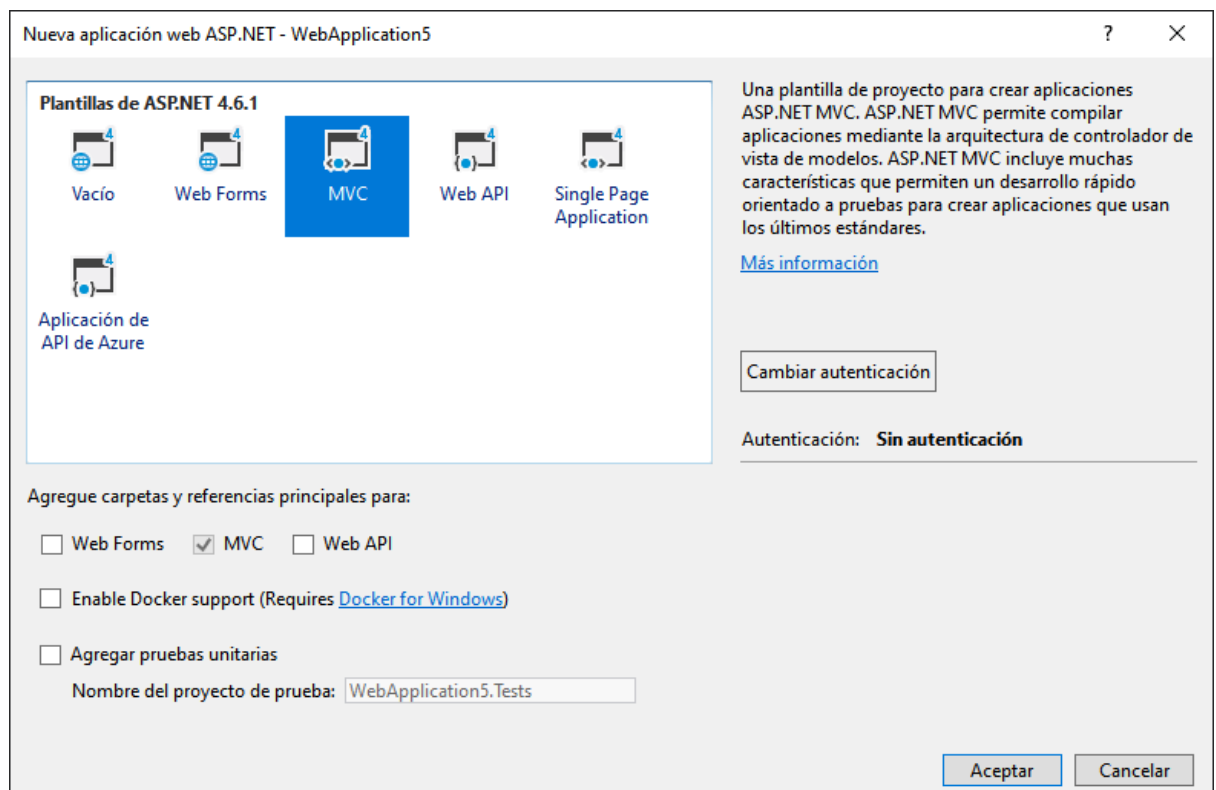


Imagen 4: fuente autoría propia

Una vez creado el proyecto ya podemos ver la estructura de directorios desde el panel Explorador de soluciones.

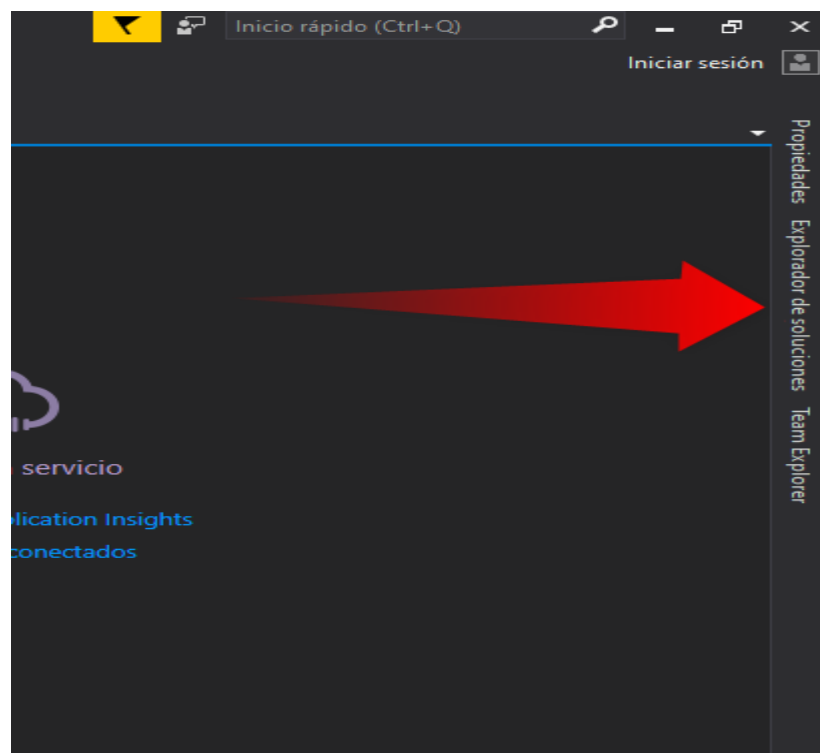


Imagen 5: fuente autoría propia

Para trabajar más cómodos presionaremos el pin Superior para fijar el panel.

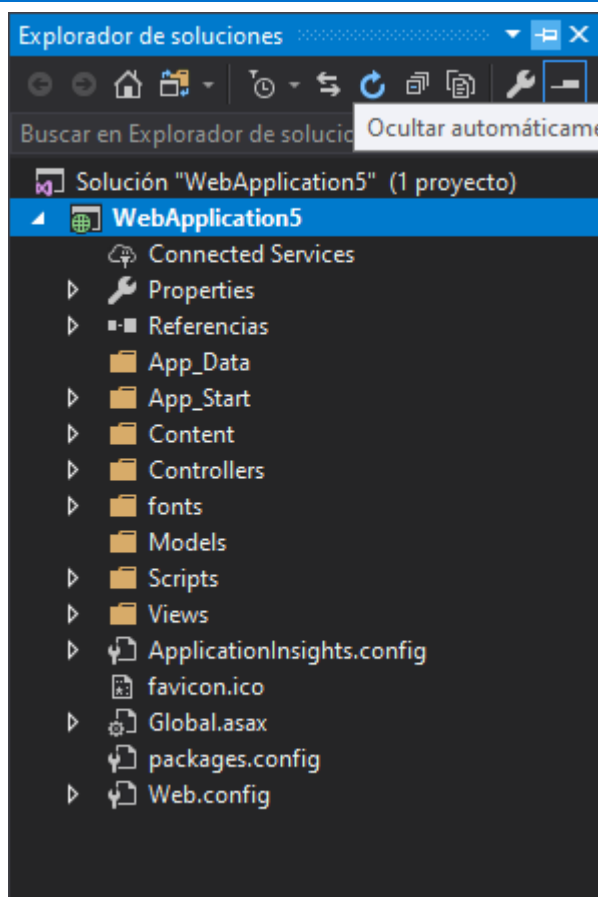


Imagen 6: fuente autoría propia

Cómo podemos observar en la imagen, tenemos la estructura de directorios y los archivos correspondientes al proyecto.

Dentro de la carpeta **Referencias** estarán las vinculaciones a otras librerías. En nuestro caso no la utilizaremos.

Global.asax y **Web.config** se encargan de la configuración del proyecto para ser instalado en el servidor. No los modificaremos en este curso.

Los directorios que utilizaremos son:

App_Start. En este directorio encontramos la configuración de nuestra aplicación web en los referidos el ruteo, es decir, la asociación de una dirección url a un controlador específico.

Content. En **Content** estará nuestra hoja de estilo y el contenido que queramos mostrar en las páginas como lo son imágenes, etc.

Controllers. En esta carpeta se encuentran los controladores.

Models. en esta carpeta se encuentran todas las clases que componen el modelo y los gestores de bases de datos.

Scripts. aquí pondremos nuestros archivos javascript.

Views. son las vistas, o sea, las páginas web que mostraremos al usuario. están escritas en un lenguaje llamado Razor que es similar a html pero con caracteres especiales que las mezclan con C#. Recordemos que el navegador web sólo entiende html por lo tanto al ser enviadas al navegador por el servidor, se reemplazarán por código html únicamente.

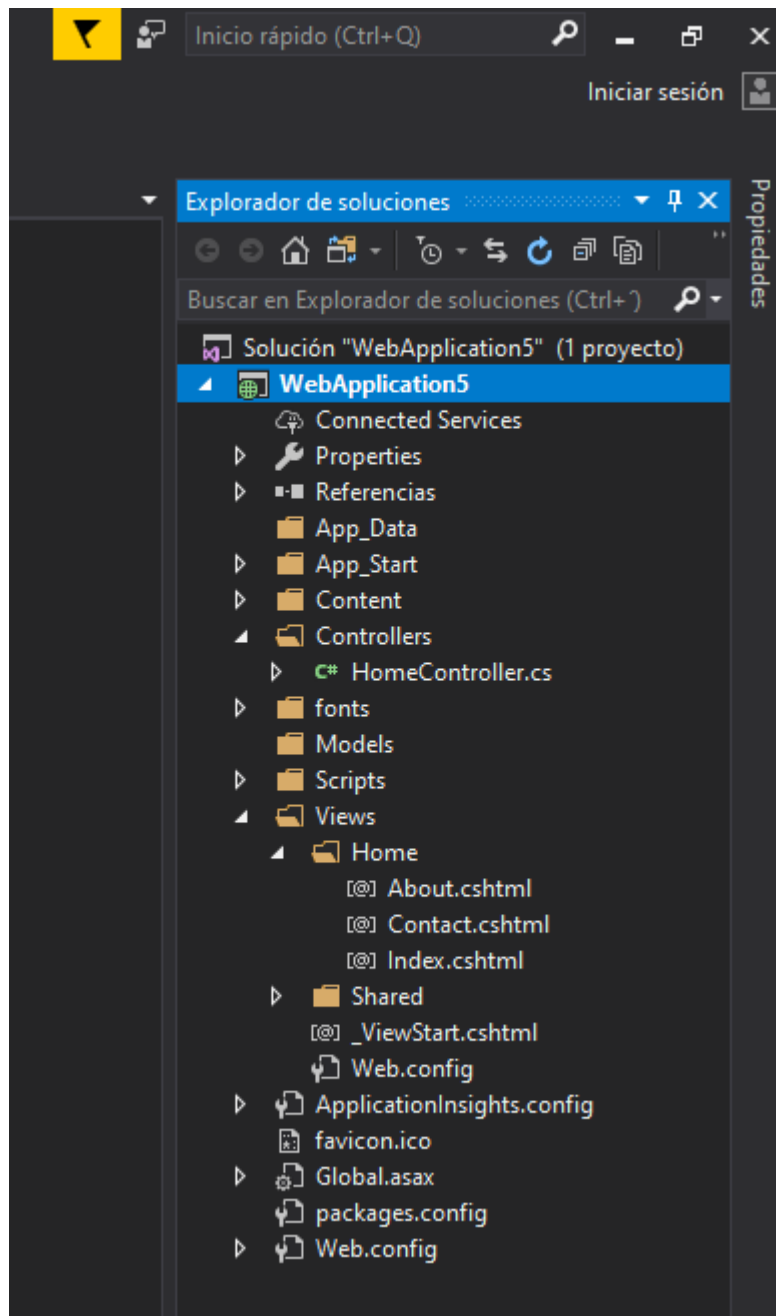


Imagen 7: fuente autoría propia

La carpeta Controllers contiene un archivo HomeController.cs. Este Archivo es el controlador principal de nuestra página que nos provee el framework por defecto. Lo podemos cambiar pero lo haremos más adelante o crearemos otro archivo y lo reemplazaremos como principal.

A su vez en la carpeta Views, se encuentran tres archivos About.cshtml, Contact.cshtml e Index.cshtml. Los dos primeros corresponden a unas páginas creadas por la plantilla. Index corresponde a la vista que responde a la acción con el mismo nombre en el controlador. Es muy importante observar que la vista y la acción dentro del controlador tienen el mismo nombre.

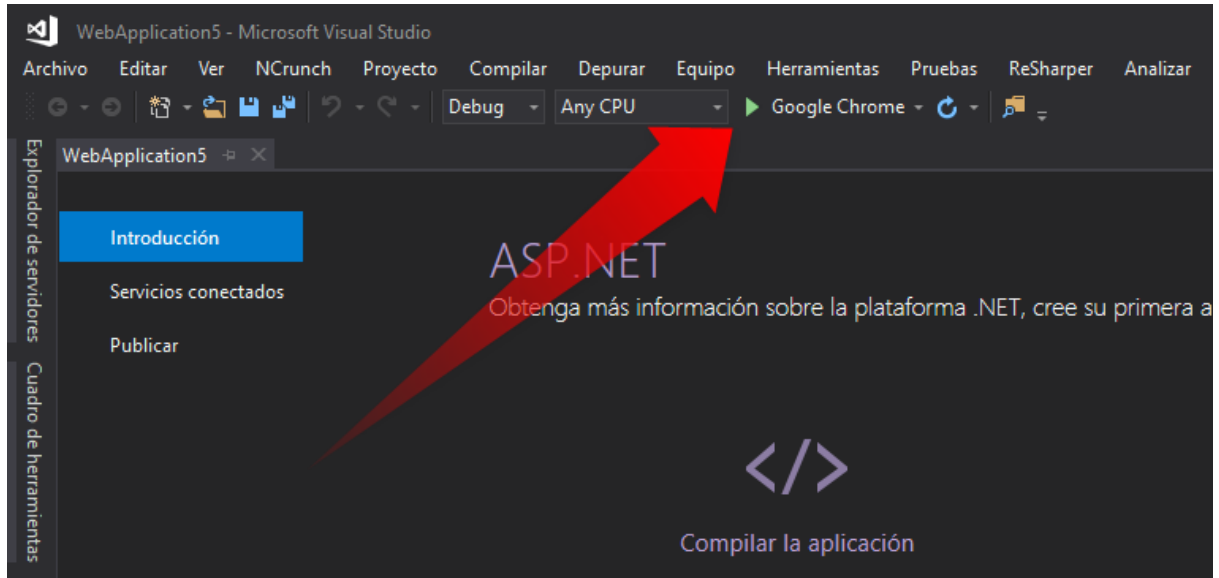


Imagen 8: fuente autoría propia

Para ver la página web propuesta presionamos en el Triángulo verde que indica que se compilará el proyecto, Se abrirá el navegador web (en este caso Google Chrome). Y navegará automáticamente a la página index como se muestra a continuación.

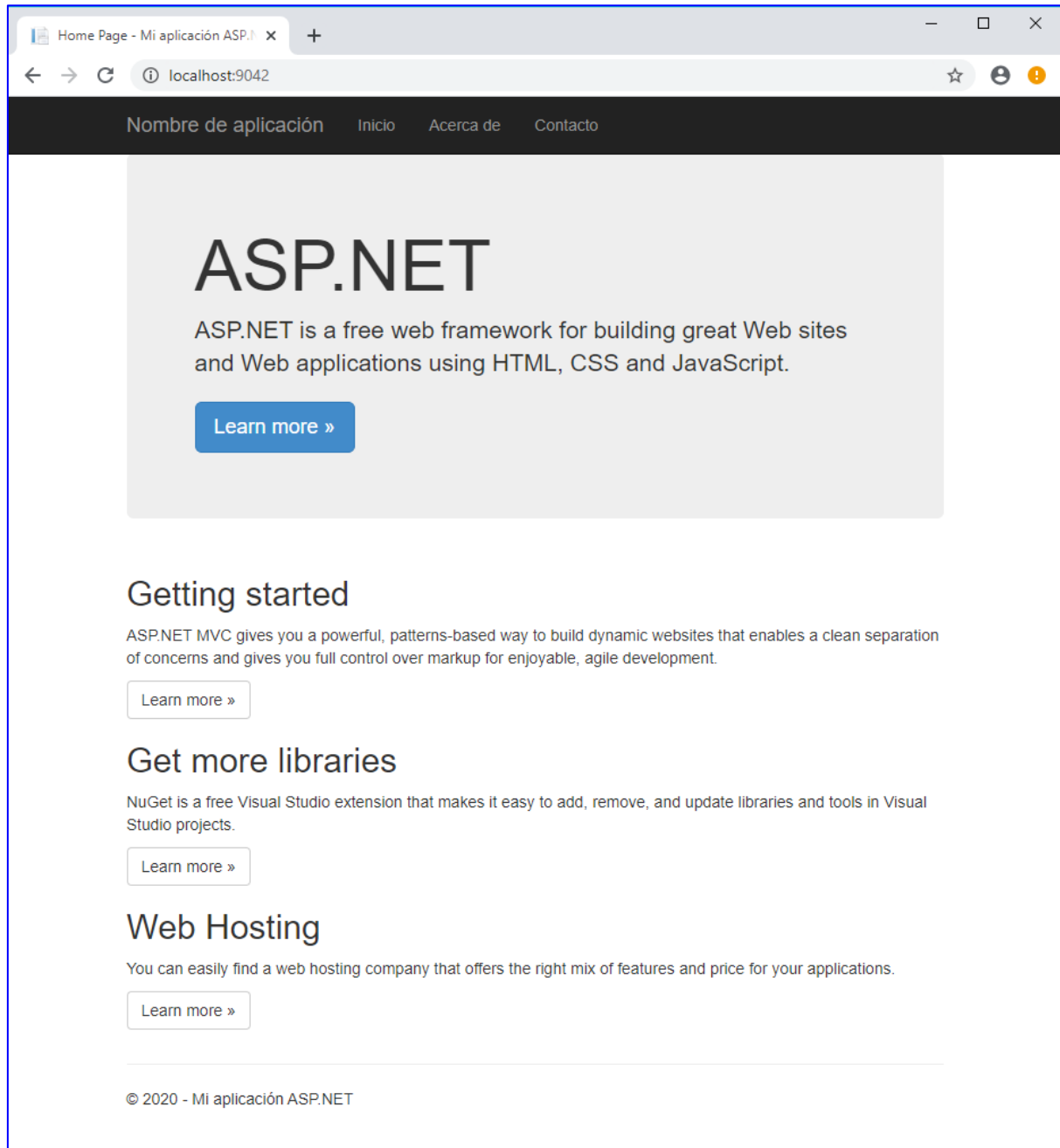


Imagen 9: fuente autoría propia

Como se muestra en la imagen anterior, vemos una página totalmente funcional con encabezado y cuerpo, todo esto con estilo.

A continuación veremos con más detalle al controlador qué es el “cerebro” de nuestra aplicación.

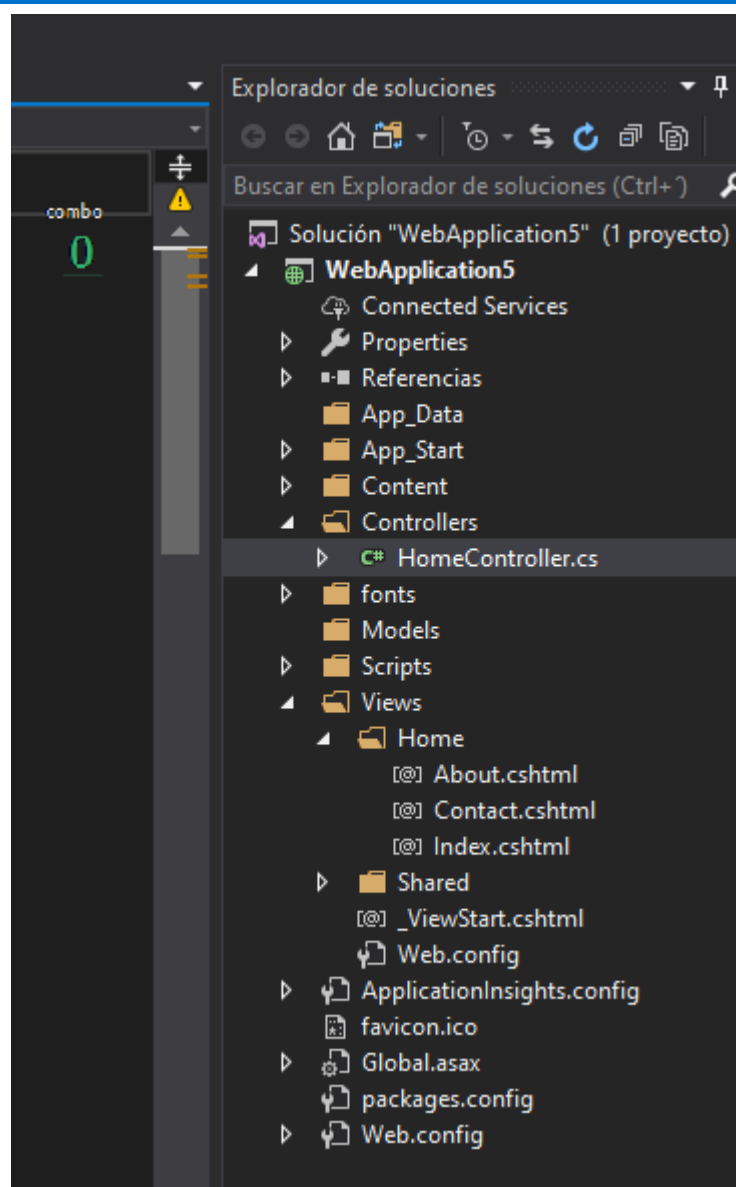


Imagen 10: fuente autoría propia

El contenido del archivo HomeController.cs es el siguiente:

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace WebApplication5.Controllers
{
    public class HomeController : Controller
```

```
{  
  
    public ActionResult Index()  
    {  
        return View();  
    }  
  
    public ActionResult About()  
    {  
        ViewBag.Message = "Your application  
description page.";  
  
        return View();  
    }  
  
    public ActionResult Contact()  
    {  
        ViewBag.Message = "Your contact page.";  
  
        return View();  
    }  
}
```

Podemos observar que HomeController deriva o hereda de Controller. Si bien el sufijo controller en el identificador, nombre, de la clase se da por convención, es muy importante respetarlo pues por detrás el framework los une a las vistas automáticamente.

Vemos tres métodos públicos que responden ActionResult. Esto indica que son “acciones”, en la jerga MVC, y que devuelven una vista. Si bien las tres líneas `return View()` son iguales, por la convención en el nombre del controlador, sabe que la vista enviar al navegador.

El objeto global estático ViewBag sirve para “poner” cosas y “sacarlas” en las vistas. Como es un objeto global, no lo utilizaremos más que para pasar cosas simples, como el título de la página.

Para pasarle a las páginas entidades del modelo, utilizaremos otra técnica (@Model)

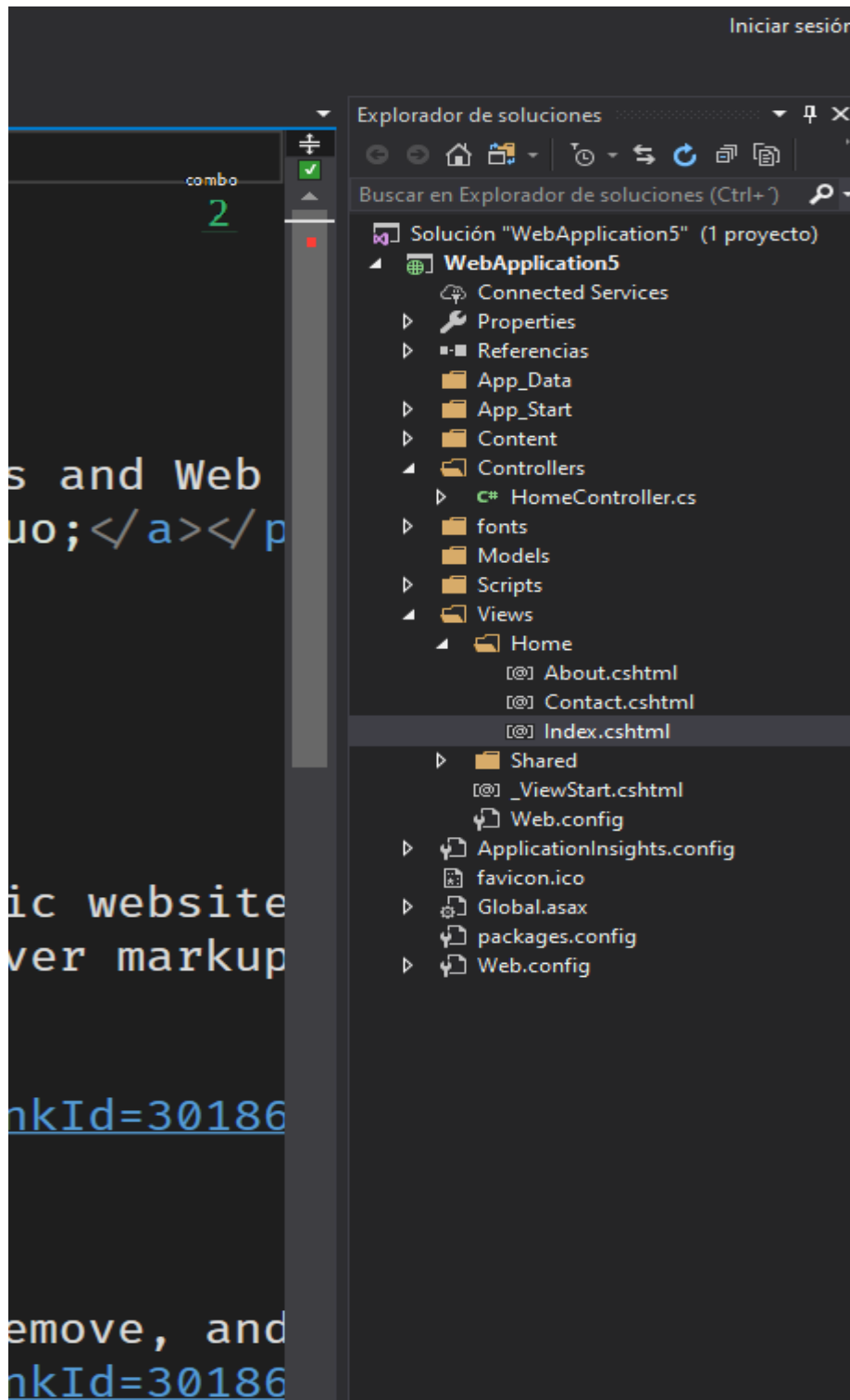


Imagen 11: fuente autoría propia

Veamos el contenido ahora de Index.cshtml

```
@{
```

```
    ViewBag.Title = "Home Page";
```

```
}

<div class="jumbotron">

    <h1>ASP.NET</h1>

    <p class="lead">ASP.NET is a free web framework for
building great Web sites and Web applications using HTML, CSS
and JavaScript.</p>

    <p><a href="https://asp.net" class="btn btn-primary
btn-lg">Learn more &raquo;</a></p>

</div>

<div class="row">

    <div class="col-md-4">

        <h2>Getting started</h2>

        <p>

            ASP.NET MVC gives you a powerful, patterns-
based way to build dynamic websites that

            enables a clean separation of concerns and
gives you full control over markup

            for enjoyable, agile development.

        </p>

        <p><a class="btn btn-default"
href="https://go.microsoft.com/fwlink/?LinkId=301865">Learn
more &raquo;</a></p>

    </div>

    <div class="col-md-4">

        <h2>Get more libraries</h2>

        <p>NuGet is a free Visual Studio extension that
makes it easy to add, remove, and update libraries and tools
in Visual Studio projects.</p>

        <p><a class="btn btn-default"
href="https://go.microsoft.com/fwlink/?LinkId=301866">Learn
more &raquo;</a></p>

    </div>

</div>
```

```
<div class="col-md-4">

    <h2>Web Hosting</h2>

    <p>You can easily find a web hosting company that
offers the right mix of features and price for your
applications.</p>

    <p><a class="btn btn-default"
href="https://go.microsoft.com/fwlink/?LinkId=301867">Learn
more &raquo;</a></p>

</div>

</div>
```

La parte de ViewBag.Title sirve para “sacar” cosas de esa “bolsa” ya mencionada. En éste caso, el título de la página.

Como podemos apreciar, todo el código se trata de divs, títulos y párrafos. No presentan ningún tipo de dificultad.

Sin embargo, ¿de dónde salen esas clases de los divs? Esas clases están definidas en las hojas de estilo que están en el directorio **Content**. Es más, pertenecen a una librería que nos provee mejoras visuales, o sea, estilos ya predefinidos. Esta librería se llama **Bootstrap**

Pero hay algo muy importante que quizás se escape a nuestro razonamiento. Si hemos dicho que estas vistas son transformadas en html y enviadas desde el servidor al navegador, dónde está la parte fuera del body? los tags de head y body? cómo hace para vincular el estilo?

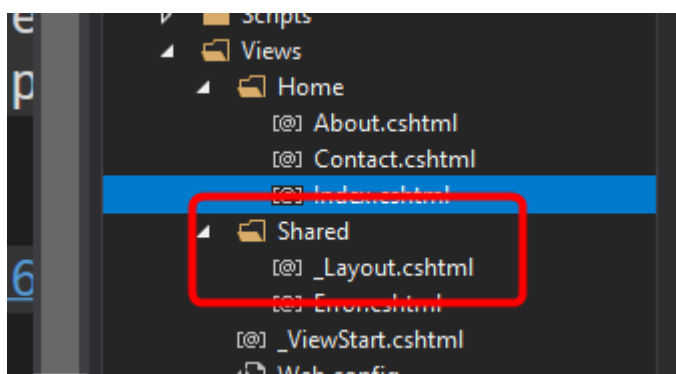


Imagen 12: autoría propia

Todo lo anterior lo hace a través del **_Layout.cshtml** que está en la carpeta **Shared**. Es la forma que tiene el lenguaje Razor cshtml (C# dentro de HTML) para reutilizar partes de páginas.

Lo que se hace es armar un layout o maqueta principal, con el encabezado de la página y en otros archivos (las vistas) el contenido.

Veamos cuál es el código de `_Layout.cshtml`

```
<!DOCTYPE html>

<html>

<head>

  <meta http-equiv="Content-Type" content="text/html;
charset=utf-8"/>

  <meta charset="utf-8" />

  <meta name="viewport" content="width=device-width,
initial-scale=1.0">

  <title>@ViewBag.Title - Mi aplicación ASP.NET</title>

  @Styles.Render("~/Content/css")

  @Scripts.Render("~/bundles/modernizr")

</head>

<body>

  <div class="navbar navbar-inverse navbar-fixed-top">

    <div class="container">

      <div class="navbar-header">

        <button type="button" class="navbar-
toggle" data-toggle="collapse" data-target=".navbar-collapse">

          <span class="icon-bar"></span>

          <span class="icon-bar"></span>

          <span class="icon-bar"></span>

        </button>

        @Html.ActionLink("Nombre de aplicación",
"Index", "Home", new { area = "" }, new { @class = "navbar-
brand" })

      </div>

      <div class="navbar-collapse collapse">

        <ul class="nav navbar-nav">

          <li>@Html.ActionLink("Inicio",
"Index", "Home")</li>

          <li>@Html.ActionLink("Acerca de",
"About", "Home")</li>
```

```

        <li>@Html.ActionLink("Contacto",
"Contact", "Home")</li>
    </ul>
</div>
</div>
</div>
<div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
        <p>&copy; @DateTime.Now.Year - Mi aplicación
ASP.NET</p>
    </footer>
</div>

    @Scripts.Render("~/bundles/jquery")
    @Scripts.Render("~/bundles/bootstrap")
    @RenderSection("scripts", required: false)
</body>
</html>

```

Podemos ver que la mayoría de las líneas son habituales menos algunas:

- @Styles.Render("~/Content/css")
- @Html.ActionLink("Inicio", "Index", "Home")
- @Scripts.Render("~/bundles/jquery")
- @RenderBody()

Styles.Render luego será reemplazado por el tag <link> que vincula las hojas de estilo. Por qué utilizamos esta directiva de Razor en vez de el tag? porque se encarga de manejar los paths o caminos a donde están los archivos en el servidor.

Lo mismo con Scripts.Render que asocia los archivos .js a los tags <scripts>.

Observar que no hace falta poner las extensiones de los archivos.

A continuación se encuentra `Html.ActionLink` que se convertirá en un link de html ``. Tiene varios parámetros y está sobrecargado. En la versión de tres parámetros, el primero es el nombre que se muestra, el segundo la acción a invocar en el controlador y tercero el controlador.

Por último, `@RenderBody()` es el lugar en donde se incrusta el código de la vista. Aquí se puede entender cómo la página tiene anidadas las vistas.

Volviendo al tema de estilos, dijimos que los estilos salen de una librería llamada Bootstrap. Como vemos a continuación existen clases predefinidas en la librería.

Por ejemplo "row" o "col-md-4" son algunas de ellas.

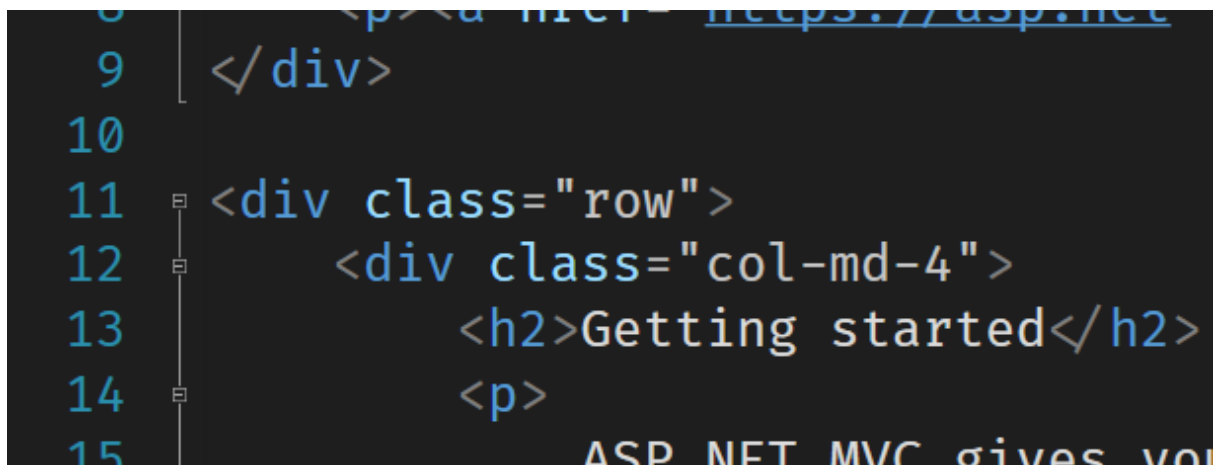
A screenshot of a code editor showing HTML code. Line 9: `</div>`. Line 10: (empty). Line 11: `<div class="row">`. Line 12: `<div class="col-md-4">`. Line 13: `<h2>Getting started</h2>`. Line 14: `<p>`. Line 15: `ASP.NET MVC gives you`. The code is highlighted in blue and green on a dark background.

Imagen 13: fuente autoría propia

Está fuera del alcance de este curso revisar el contenido de la documentación de la librería. Para más información consultar <https://getbootstrap.com/> sección Documentation.

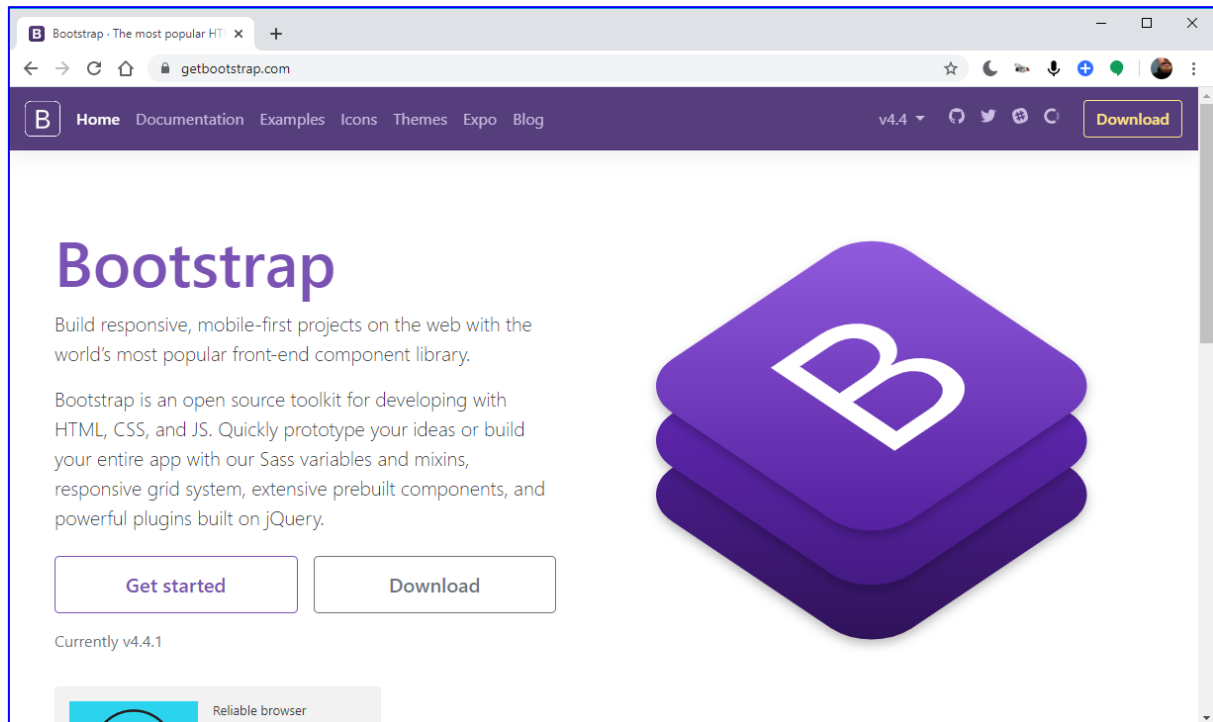


Imagen 14: autoría propia

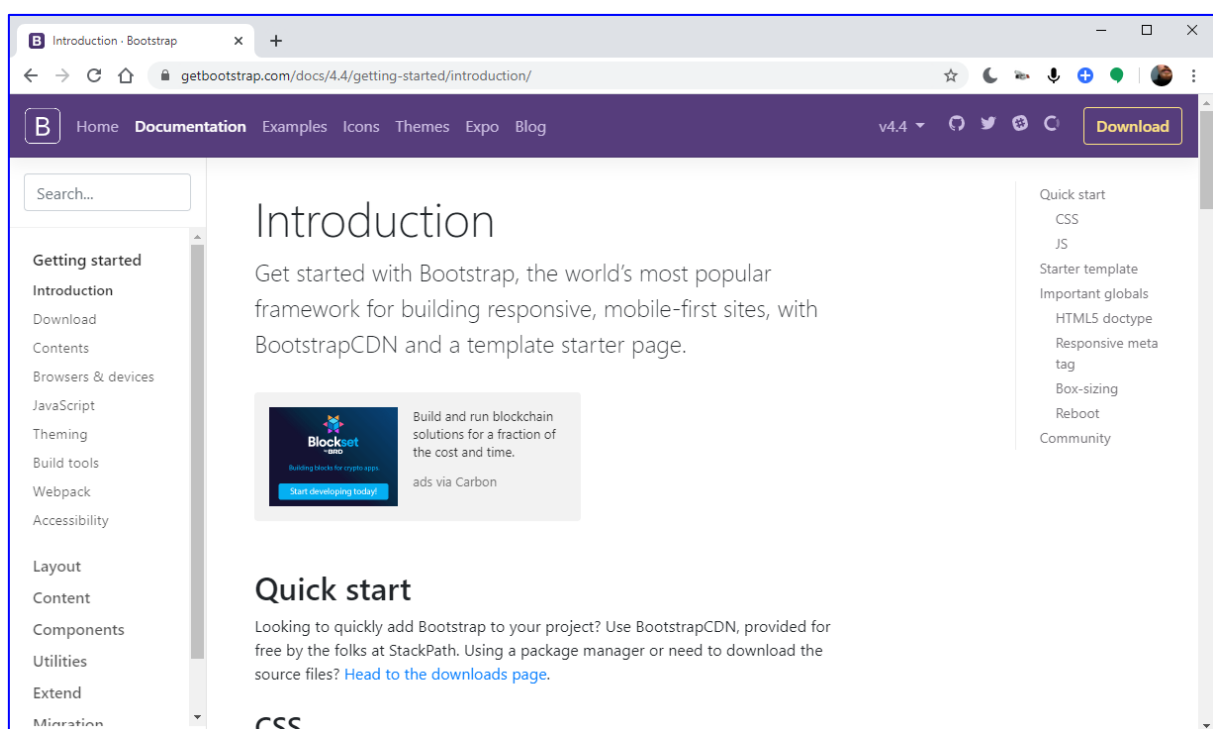


Imagen 15: autoría propia

Si nosotros queremos incorporar nuevas clases de estilo podemos hacerlo en el archivo Site.css que está dispuesto para ello:

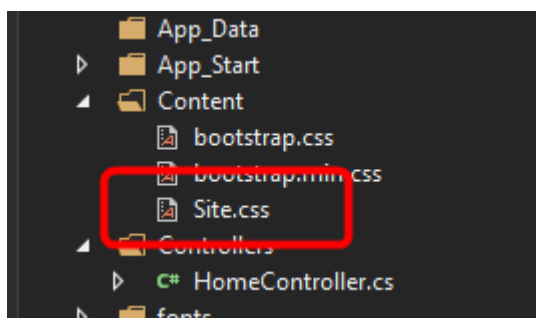


Imagen 16: autoría propia

Ahora bien, veamos el contenido de otro archivo importante que está en la carpeta **App_Start**. Este archivo es el **RouteConfig.cs**

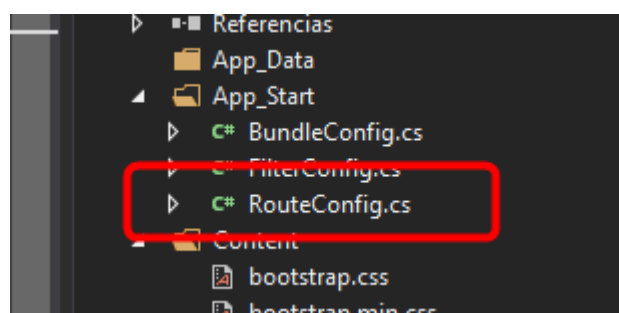


Imagen 17: autoría propia

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using System.Web.Routing;

namespace WebApplication5
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection
routes)
        {
```

```
routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

routes.MapRoute(
    name: "Default",
    url: "{controller}/{action}/{id}",
    defaults: new { controller = "Home",
action = "Index", id = UrlParameter.Optional }
);
}
```

Route significa ruta y hace referencia a la vinculación de lo que ingresamos en la URL del navegador y qué controladores y acciones son asociadas.

Todos los frameworks para crear páginas web tienen la noción de enrutamiento de la URL que asocia paths o caminos en la URL a recursos (controllers).

Aquí podremos cambiar qué pagina es cargada al levantar el servidor (defaults) y cómo son los enrutamientos (controller/action/id). Para más información ver el ejemplo de ABM de la unidad siguiente.

Por último, en este ejemplo, vemos que la carpeta Models se encuentra vacía.

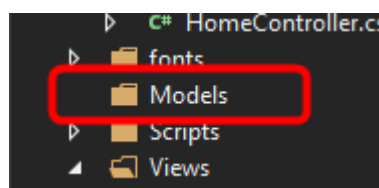


Imagen 18: autoría propia

Aquí es donde pondremos nuestras clases pertenecientes al dominio.

Creando modelos, vistas y controladores.

Empecemos por el controlador.

Los controladores son entidades que controlan algún recurso en la aplicación y poseen, por cada acción por lo general, una vista asociada.

Por ejemplo podríamos tener un controlador llamado PersonaController que permite:

- Mostrar la lista de personas
- Agregar una persona
- Modificar una persona
- Eliminar una persona

A estas acciones en conjunto se las denominan CRUD (en inglés) por create, read, update, delete o en español como ABMC (Alta, baja, modificación y consulta) o simplemente ABM.

Para agregar un controlador debemos hacer click derecho sobre la carpeta Controllers en el Explorador de soluciones y luego seleccionar Agregar > Controlador...

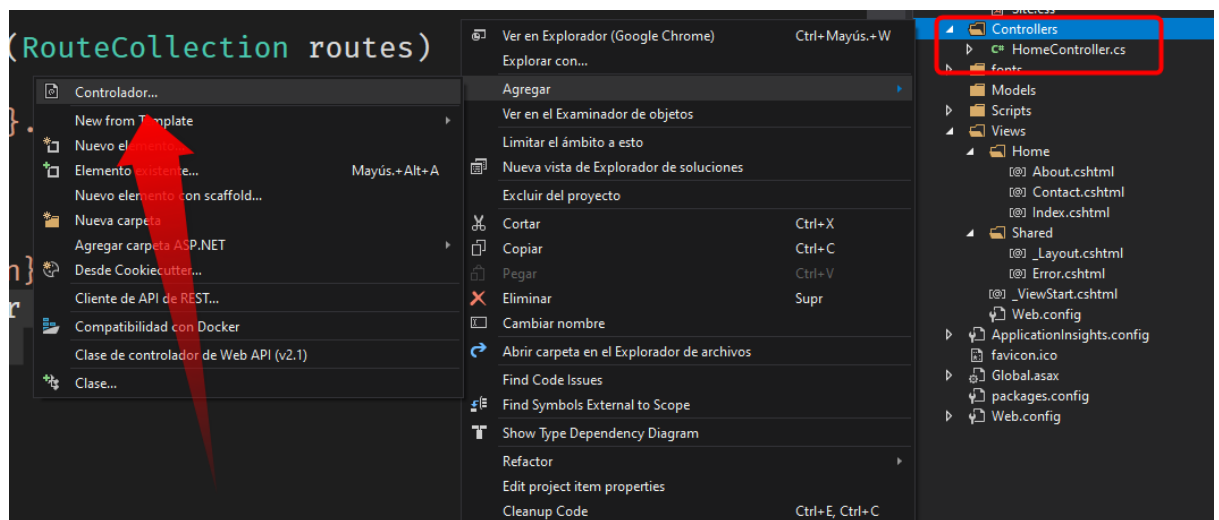


Imagen 19: autoría propia

Si queremos un controlador vacío seleccionamos Controlador de MVC 5: en blanco.

Puede usted probar con el controlador de MVC 5 con acciones de lectura y escritura para ver las convenciones REST de acciones. Ello queda fuera del alcance de este curso.

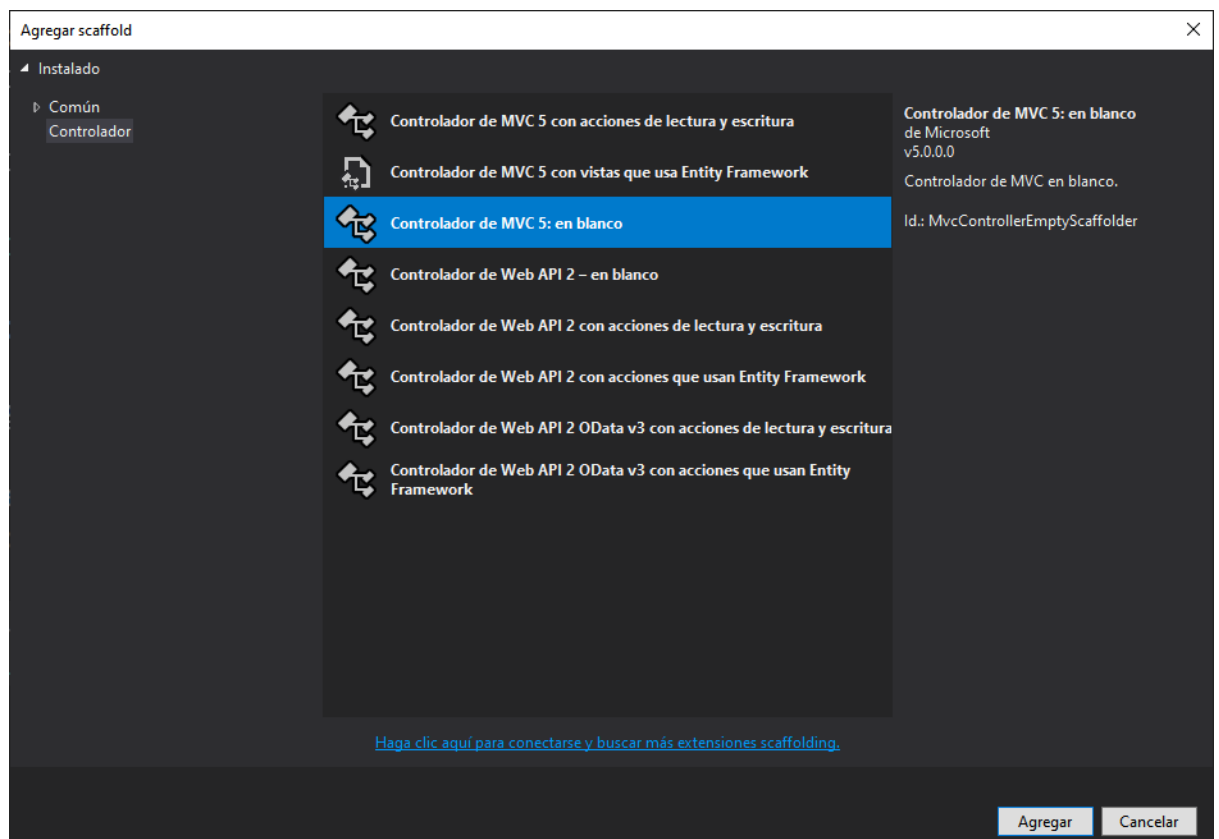


Imagen 20: autoría propia

Es muy importante respetar la convención de nombre. Se deberá mantener el sufijo Controller siempre. A Default lo podemos cambiar por cualquier nombre, dependiendo del recurso a manejar (por ejemplo PersonasController)

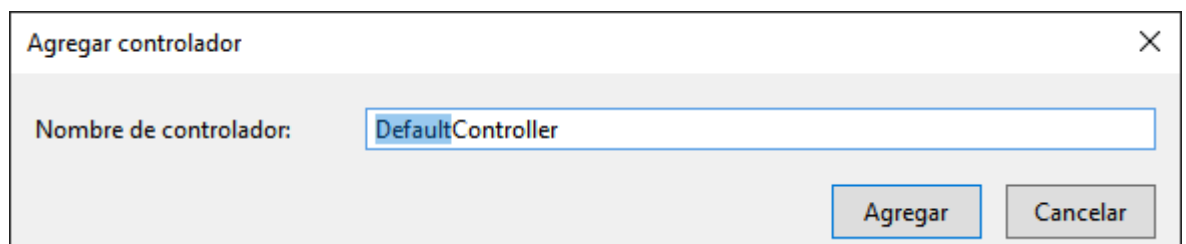


Imagen 21: autoría propia

Ello nos dará el siguiente código fuente:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Web;  
using System.Web.Mvc;
```

```
namespace WebApplication5.Controllers
{
    public class DefaultController : Controller
    {
        // GET: Default
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

Vemos que empezamos con la acción Index().

En este ejemplo mantendremos el nombre de DefaultController para nuestro controlador.

Ahora bien, vemos que Index() retorna una vista. Vamos a crearla.

Dentro de la carpeta Vistas entramos a la carpeta creada con el nombre del controlador. Ahí tenemos que crear una vista con el nombre de la acción del controlador. En este caso la vista se deberá llamar Index.

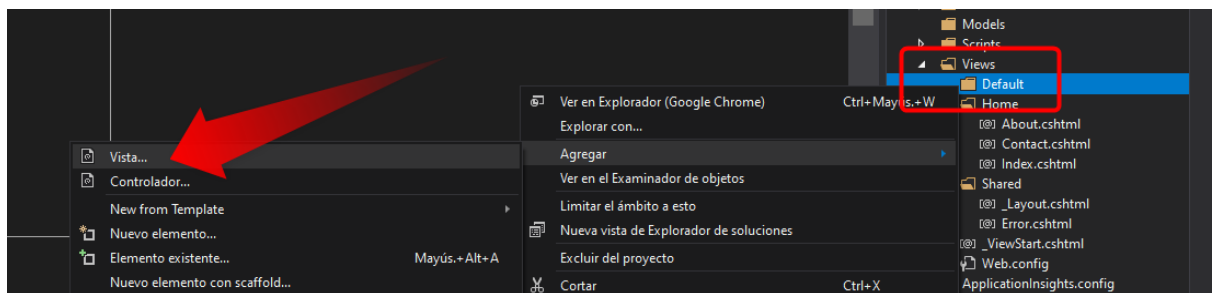


Imagen 22: autoría propia

Del cuadro de diálogo siguiente cambiamos el nombre de View por Index.

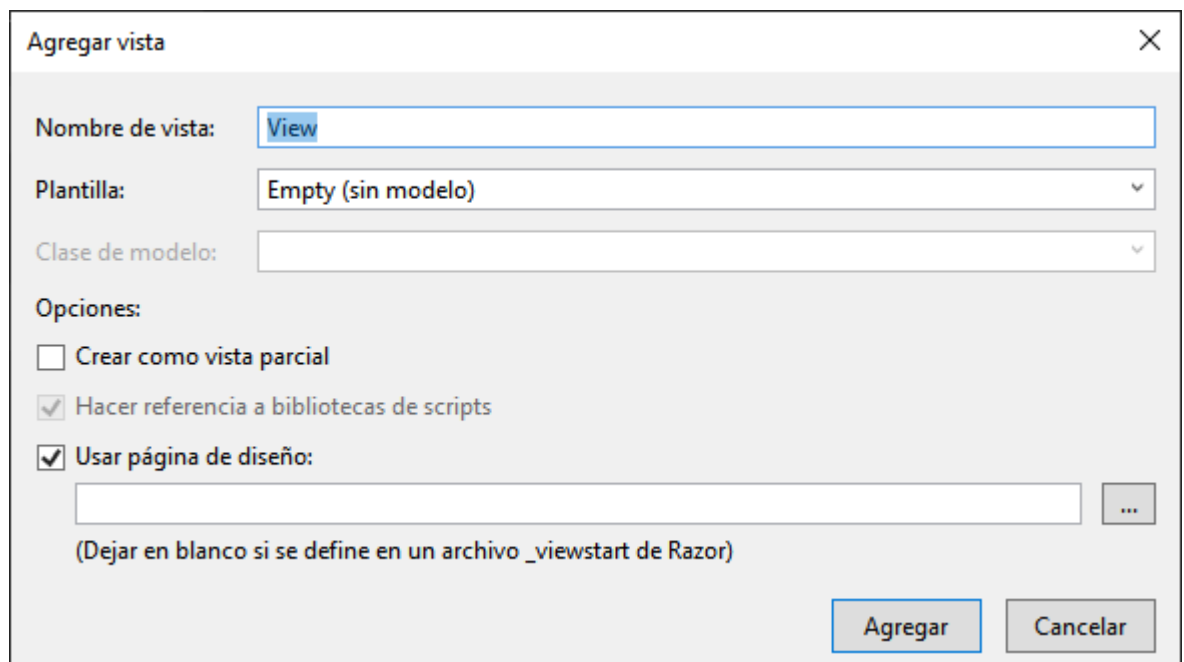


Imagen 23: fuente autoría propia

Que nos dará el siguiente código

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

```
<h2>Index</h2>
```

Por último para crear un modelo para utilizar en los controladores y vistas, hacemos click en la carpeta Models con el botón derecho del mouse y seleccionamos Agregar > Nuevo elemento...

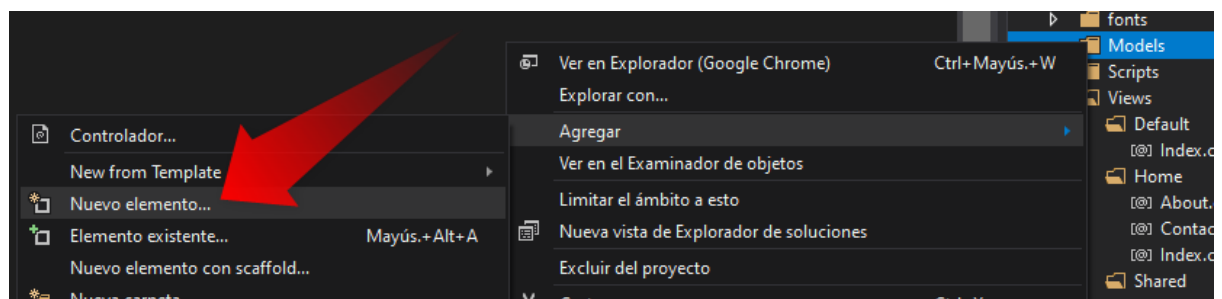


Imagen 24: autoría propia

A continuación buscamos la opción en Visual C# > Código > Clase y le ponemos el nombre de la clase que queremos crear.

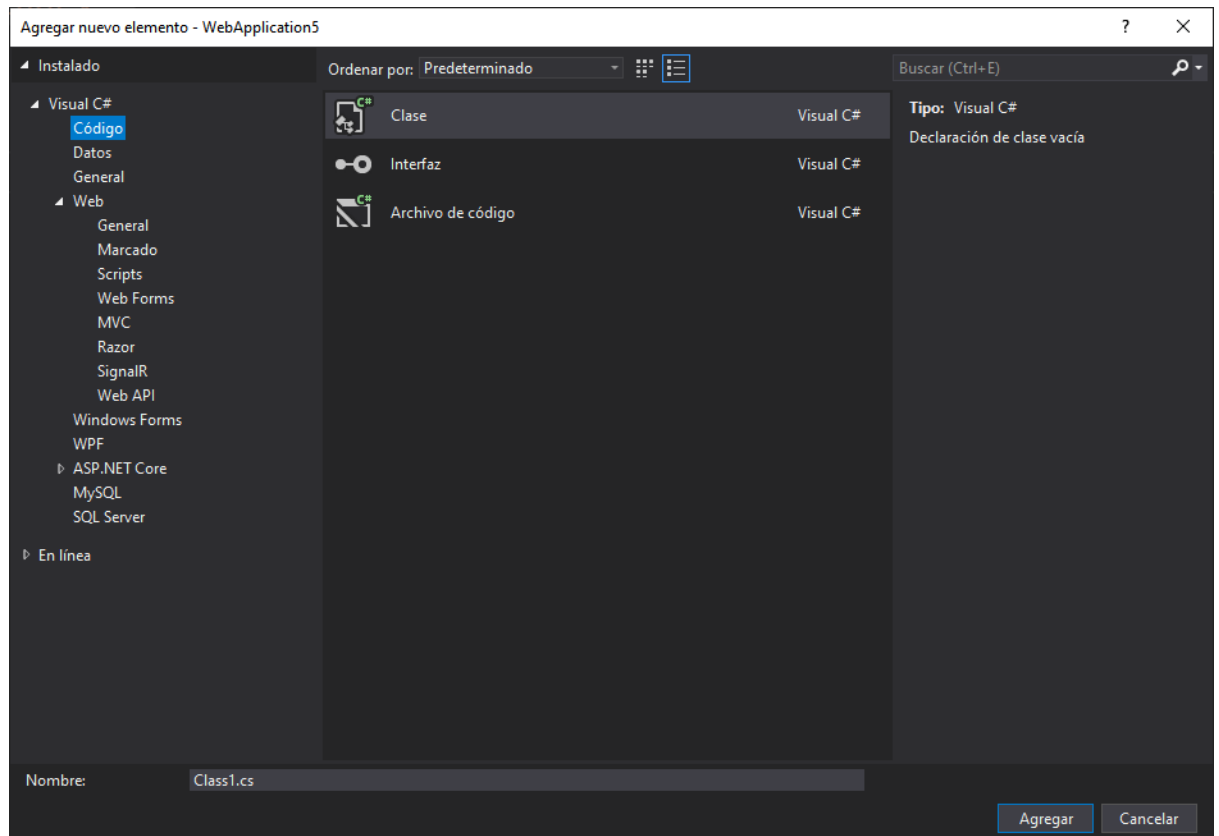


Imagen 25: autoría propia

Nos dará el siguiente código pre completado.

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace WebApplication5.Models
{
    public class Class1
    {
    }
}
```

El espacio de nombres namespace dice “WebApplication5”. Este es el nombre de la aplicación. Tu aplicación seguramente tendrá otro nombre.

Programación de vistas con páginas cshtml.

El motor de vistas procesa el contenido de ASP.Net y busca instrucciones, generalmente para insertar contenido dinámico en la salida enviada al servidor. Ese motor de vistas se llama Razor.

En esta sección veremos un recorrido rápido sobre la sintaxis de Razor para reconocer sus expresiones.

Para practicar sobre las vistas deberemos preparar un modelo de prueba. Vamos a definir el modelo Product.

En la carpeta Models crearemos un archivo Product.cs

```
namespace Razor.Models {  
    public class Product {  
        public int ProductID { get; set; }  
        public string Name { get; set; }  
        public string Description { get; set; }  
        public decimal Price { get; set; }  
        public string Category { set; get; }  
    }  
}
```

Definiremos luego el controlador.

```
using System.Web.Mvc;  
using Razor.Models;  
namespace Razor.Controllers {  
    public class HomeController : Controller {  
        Product myProduct = new Product {  
            ProductID = 1,  
            Name = "Kayak",  
            Description = "A boat for one person",  
            Category = "Watersports",  
            Price = 275M  
        };  
        public ActionResult Index() {
```

```
        return View(myProduct) ;  
    }  
}
```

Como vemos inicializamos un objeto Product directamente en controlador y lo mandamos a la vista por el parámetro de View(myProduct)

Por último nos falta crear la vista. En el ejemplo siguiente no usaremos el Layout.

```
@model Razor.Models.Product  
  
@{  
    Layout = null;  
}  
  
<!DOCTYPE html>  
<html>  
    <head>  
        <meta name="viewport" content="width=device-width"  
    />  
    <title>Index</title>  
    </head>  
    <body>  
        <div>  
        </div>  
    </body>  
</html>
```

Agreguemos entonces la primera línea de código en la vista

```
...  
  
@model Razor.Models.Product  
  
...
```

Las sentencias de Razor comienzan con un caracter de “@”. En este caso, la sentencia @model declara el tipo de objeto del modelo que vamos a pasar a la vista desde la acción index del controlador. De esta forma podemos referirnos a los métodos, atributos y propiedades del modelo a través de la instrucción @Model.

```
@model Razor.Models.Product

@{
    Layout = null;
}

<!DOCTYPE html>

<html>

<head>

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

<title>Index</title>

</head>

<body>

<div>

    @Model.Name

</div>

</body>

</html>
```

Es importante observar la distinción entre cómo se declara el modelo usando `@model` con la letra m minúscula y cómo se accede a las propiedades del objeto con la letra m mayúscula.

Razor es capaz de procesar sentencias condicionales lo que significa que podemos modificar la salida dependiendo de los valores del modelo.

```
@model Razor.Models.Product

@{
    ViewBag.Title = "DemoExpression";
    Layout = "~/Views/_BasicLayout.cshtml";
}

<table>

<thead>

<tr><th>Property</th><th>Value</th></tr>

</thead>

<tbody>
```

```

<tr><td>Name</td><td>@Model.Name</td></tr>
<tr><td>Price</td><td>@Model.Price</td></tr>
<tr>
<td>Stock Level</td>
<td>
@switch ((int)ViewBag.ProductCount) {
case 0:
@: Out of Stock
break;
case 1:
<b>Low Stock (@ViewBag.ProductCount)</b>
break;
default:
@ViewBag.ProductCount
break;
}
</td>
</tr>
</tbody>
</table>

```

Observación: Si bien en el ejemplo utilizamos @ViewBag para pasarle datos a la vista esta no es una buena práctica. Utilizaremos entonces @model.

Pasar datos por el ViewBag, que es una entidad global, nos permite probar más fácilmente la vista. Luego reemplazaremos todas estas instrucciones por los correspondientes @Model.<atributo público>

Otro ejemplo en donde utilizamos un condicional simple:

```

@model Razor.Models.Product
@{
ViewBag.Title = "DemoExpression";
Layout = "~/Views/_BasicLayout.cshtml";
}

```

```

<table>

<thead>

<tr><th>Property</th><th>Value</th></tr>

</thead>

<tbody>

<tr><td>Name</td><td>@Model.Name</td></tr>

<tr><td>Price</td><td>@Model.Price</td></tr>

<tr>

<td>Stock Level</td>

<td>

@if (ViewBag.ProductCount == 0) {

@:Out of Stock

} else if (ViewBag.ProductCount == 1) {

<b>Low Stock (@ViewBag.ProductCount)</b>

} else {

@ViewBag.ProductCount

}

</td>

</tr>

</tbody>

</table>

```

Enumerando arreglos y colecciones

Cuando escribimos aplicaciones MVC a menudo necesitamos enumerar el contenido de un arreglo o de algún tipo de colección de objetos.

En el siguiente ejemplo veremos cómo hacerlo. Primero definimos una nueva acción en el controlador.

```

using System.Web.Mvc;

using Razor.Models;

namespace Razor.Controllers {

public class HomeController : Controller {

    Product myProduct = new Product {

```

```
ProductID = 1,
Name = "Kayak",
Description = "A boat for one person",
Category = "Watersports",
Price = 275M
};

// ...other action methods omitted for brevity...

public ActionResult DemoArray() {
    Product[] array = {
        new Product {Name = "Kayak", Price = 275M},
        new Product {Name = "Lifejacket", Price = 48.95M},
        new Product {Name = "Soccer ball", Price = 19.50M},
        new Product {Name = "Corner flag", Price = 34.95M}
    };
    return View(array);
}
}
```

Esta acción crea un vector de productos que contiene valores simples y son pasados a la vista.

Para mostrar el contenido del vector debemos hacer lo siguiente :

```
@model Razor.Models.Product[]
@{
    ViewBag.Title = "DemoArray";
    Layout = "~/Views/_BasicLayout.cshtml";
}
@if (Model.Length > 0) {
<table>
<thead><tr><th>Product</th><th>Price</th></tr></thead>
<tbody>
    @foreach (Razor.Models.Product p in Model) {
```

```

<tr>

<td>@p.Name</td>

<td>$@p.Price</td>

</tr>

}

</tbody>

</table>

} else {

<h2>No product data</h2>

}

```

Definiendo el espacio de nombre namespace

Como se aprecia en el ejemplo anterior nos referimos al producto con el nombre completo, es decir:

```

...

@foreach (Razor.Models.Product p in Model) {

...

```

Para evitar este tipo de práctica deberemos incluir la expresión @using como se ve en el siguiente ejemplo:

```

@using Razor.Models

@model Product[]

@{

    ViewBag.Title = "DemoArray";

    Layout = "~/Views/_BasicLayout.cshtml";

}

@if (Model.Length > 0) {

<table>

<thead><tr><th>Product</th><th>Price</th></tr></thead>

<tbody>

@foreach (Product p in Model) {

<tr>

<td>@p.Name</td>

```



```
<td>${p.Price}</td>
</tr>
}
</tbody>
</table>
} else {
<h2>No product data</h2>
}
```

Una vista puede contener varias expresiones @using.

Importante

Queda fuera del alcance de este curso, el tema de validaciones del lado del servidor.

Es un tema muy importante si se hará un despliegue del servidor en producción.

Se recomienda leer <https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/introduction/adding-validation>

Bibliografía

Adam Freeman (2013), Pro ASP.NET MVC 5, ISBN 978-1-4302-6530-6, Apress

Nick Harrison (2015), ASP.NET MVC Succinctly, Syncfusion, USA.
Recuperado de: https://www.syncfusion.com/ebooks/aspnet_mvc_succinctly

Rick Anderson (2018) Microsoft .Net Docs, Getting started with ASP.NET MVC 5, Recuperado de <https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/introduction/getting-started>



Atribución-NoComercial-SinDerivadas

Se permite descargar esta obra y compartirla, siempre y cuando no sea modificado y/o alterarse su contenido, ni se comercializarse. Referenciarlo de la siguiente manera:

Universidad Tecnológica Nacional Regional Córdoba (2020). Material para la Tecnicatura en Programación Semipresencial de Córdoba. Argentina.