



Formación  
Profesional  
FOMENTO

# Modelo-Vista-Controlador (MVC)

CS Desarrollo de Aplicaciones Web

# Introducción

- En este capítulo veremos aspectos como:
  - Creación del Modelo
  - Creación de las rutas
  - Creación del Controlador
  - Creación de las Vistas

# Introducción

- El objetivo de lo que vamos a ver en este tema es **ampliar la aplicación añadiéndole un motor simple de visualización de artículos.**
- Las operaciones que permitiremos son las siguientes:
  - Visualizar las 10 últimas entradas publicadas
  - Visualizar la relación de artículos publicados un mes y año concreto
  - Visualizar un artículo determinado



# CREACIÓN DEL MODELO

# Creación del modelo

## El Modelo

- Representa los datos del dominio, son responsables de aportar la lógica de negocio y deben aportar los mecanismos de persistencia del sistema.
- En nuestro ejemplo, utilizaremos las siguientes tecnologías:
  - Almacenaremos los posts de nuestro blog en una base de datos SQL Server
  - Utilizaremos Entity Framework para el mapeo E/R y la gestión de los datos (podríamos utilizar cualquier micro-ORM que conozcamos)

# Creación del modelo

- Para crear las bases de datos utilizaremos un enfoque denominado "**code-first**".
- Esto nos permitirá definir las entidades de datos que necesitemos, y, con muy poca información adicional, Entity Framework se encargará de crear la base de datos por nosotros y gestionar automáticamente la persistencia.
- Necesitaremos definir:
  - **Las entidades de datos que necesitamos.** En nuestro caso, sólo necesitaremos de momento la entidad "Post".
  - **El contexto de datos de Entity Framework**, que es la clase que se encarga de la persistencia de información. La persistencia la trataremos con más calma más adelante.

# Creación del modelo

## Creación de las entidades

- Crearemos entidades básicas donde les definiremos los atributos necesarios.
- Utilizaremos clases POCO (Plain Old CLR Objects, u objetos planos).
- Lo guardaremos en la carpeta **Models**

```
public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Code { get; set; }
    public string Text { get; set; }
    public DateTime Date { get; set; }
    public string Author { get; set; }
}
```



# **Actividad 1**

*Vamos a crear una aplicación web que se conecte a la base de datos Pokemon y nos muestre qué Pokémons hay y sus características.*

*Para ello, empezaremos creando una entidad **Pokemon** y la guardaremos en la carpeta correspondiente.*

*Crea una acción para agregar pokémon y otra para ver un Pokémon*

**10 MINUTOS**



# Creación del modelo

## Creación del contexto de datos

- El contexto de datos es la clase que nos permitirá **acceder fácilmente al almacén de persistencia para leer o grabar entidades**.
- Un contexto de datos simple para nuestro proyecto podría ser el siguiente:



```
public class BlogContext : DbContext
{
    public BlogContext(DbContextOptions<BlogContext> options) : base(options)
    { }
    public DbSet<Post> Posts { get; set; }
}
```

# ▶ Creación del modelo

## Creación del contexto de datos



- Para trabajar con la persistencia de los datos necesitaremos instalar los siguientes paquetes, utilizando el Administrador de paquetes NuGet:

- EntityFrameworkCore

**Microsoft.EntityFrameworkCore**  por Microsoft, **661M** descargas 7.0.4

Entity Framework Core is a modern object-database mapper for .NET. It supports LINQ queries, change tracking, updates, and schema migrations. EF Core works with SQL Server, Azure SQL Database, SQLite, Azure Cosmos DB, MySQL, PostgreSQL, and other databases t...

- EntityFrameworkCore.SqlServer

**Microsoft.EntityFrameworkCore.SqlServer**  por Microsoft, **314M** descargas 7.0.4

Microsoft SQL Server database provider for Entity Framework Core.

# Creación del modelo

## Creación del contexto de datos

- Este contexto de datos contiene un conjunto de objetos de tipo Post (`DbSet<Post>`) al que accederemos utilizando la propiedad llamada `Posts` del contexto.
- Los `DbSet` podemos consultarlas mediante sentencias LINQ y manipular su contenido usando métodos como `Add()` o `Remove()`, veremos más adelante cómo trabajar con ellos.



## ***Vídeo: Iniciación al LINQ***

<https://www.youtube.com/watch?v=q0XoxNKeB3Q>

# Creación del modelo

## Creación del contexto de datos

- Para indicar cuál es la base de datos a la que tenemos que acceder, y con qué usuario, utilizamos las **cadena de conexión**, definidas en el fichero **appsettings.json**
- En nuestro ejemplo, definimos la cadena de conexión “ConexionMontecastelo”, dentro de la sección “ConnectionStrings”

```
"ConnectionStrings": {  
  "ConexionMontecastelo": "Data Source=localhost;Initial Catalog=BD_Articulos;User  
    ID=sa;Password=root;Trusted_Connection=True;MultipleActiveResultSets=true;TrustServerCertificate=True"  
},
```

# Creación del modelo

## Creación del contexto de datos

- Por último, en nuestro fichero Program.cs, debemos especificar el contenido del valor **options**, que pasamos en el constructor del BlogContext:

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddDbContext<BlogContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("ConexionMontecastelo")));

builder.Services.AddControllersWithViews();
```

# Creación del modelo

## Creación del componente de gestión

- Para finalizar con la creación del modelo, necesitamos crear el componente que nos permita **definir las operaciones que queremos ejecutar desde el resto de la aplicación:**
- En el ejemplo, crearemos la clase **BlogManager**, dentro de la carpeta Models

# Creación del modelo

## Creación del componente de gestión

- Los pasos que debemos dar son:
  - Instanciar al contexto para acceder a las entidades de datos
  - Implementar los métodos necesarios:
    - Método para obtener los últimos posts
    - Método para obtenerlos según el mes y año de publicación
    - Método para obtener un post dado su código.
  - El implementar la interfaz IDisposable permitirá la liberación de los recursos usados por el contexto



# Creación del modelo

```
public class BlogManager : IDisposable
{
    private readonly BlogContext _data;

    public BlogManager(BlogContext contexto)
    {
        _data = contexto;
    }

    public IEnumerable<Post> GetLatestPosts(int max)
    {
        var posts = from post in _data.Posts
                     orderby post.Fecha descending
                     select post;
        return posts.Take(max).ToList();
    }

    public IEnumerable<Post> GetPostsByDate(int year, int month)
    {
        var posts = from post in _data.Posts
                     where post.Fecha.Month == month &&
                        post.Fecha.Year == year
                     orderby post.Fecha descending
                     select post;
        return posts.ToList();
    }

    public Post GetPost(string code)
    {
        return _data.Posts.FirstOrDefault(post => post.Codigo == code);
    }

    public void Dispose()
    {
        _data.Dispose();
    }
}
```



## Actividad 2

*Vamos a realizar las siguientes acciones en nuestra web:*

- *Crea el contexto de datos con la información necesaria para conectarse a la BD Pokemon.*
- *Crea un componente de gestión para poder trabajar con los Pokemon. Implementa:*
  - *Método que nos devuelva todos los Pokémon*
  - *Método al que le pasemos un ID y nos devuelva info de ese Pokémon en específico*
  - *Método al que le pasemos un peso y una altura y nos devuelva los Pokemon que correspondan*

**15 MINUTOS**



## ***Vídeo: Creación del modelo desde cero***

<https://youtu.be/kmhy3MhuFBU>



# CREACIÓN DE RUTAS

# Creación de rutas

- Hemos visto que el **sistema de routing actúa como un controlador frontal al que llegan las peticiones para**, en función de la información contenida en la tabla de rutas, **delegar el proceso** de las mismas **a controladores más específicos** de nuestra aplicación.
- Para ello, deberemos trabajar con el fichero **Program.cs**

# Creación de rutas

- Las rutas que habíamos planificado para acceder a las funcionalidades del sistema son las siguientes:

Patrón de URL	Ejemplos de petición	Acción a realizar
<code>{controller}/{action}/{id}</code>	<code>/blog</code>	Muestra el listado con un resumen de las últimas entradas publicadas, ordenadas cronológicamente.
<code>blog/archive/{year}/{month}</code>	<code>/blog/archive/2005/12</code> <code>/blog/archive/2006/8</code>	Muestra el resumen de las entradas publicadas en el mes y año indicado en
<code>blog/{code}</code>	<code>/blog/welcome-to-aspnet-mvc</code>	Muestra el artículo cuyo código coincide con el especificado en la

# Creación de rutas

**Ruta:** {controller}/{action}/{id}

- Esta ruta coincide con la ruta por defecto, por lo que no tendremos que registrarla de nuevo.
- En concreto, la ruta por defecto se define en la siguiente porción de código

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

# Creación de rutas

**Ruta:** {controller}/{action}/{id}

- Los parámetros usados para registrar la ruta son:
  - **name:** Indicamos el nombre único que daremos a esta entrada de la tabla de rutas.
  - **pattern:** Indica el patrón de URL al que hace referencia esta ruta.
  - Le definimos **Home** e **Index** como valores por defecto, de manera que si no especificamos ningún controlador o acción, accede al por defecto.



# Creación de rutas

**Ruta:** blog/archive/{year}/{month}

- Este tipo de direcciones no encajarían en el patrón de ruta anterior, por lo que será necesario crear una ruta específica para ello:

```
app.MapControllerRoute(  
    name: "archive",  
    pattern: "Blog/Archive/{year}/{month}",  
    defaults: new { controller = "Blog", action = "Archive" });
```

- Como queremos que solamente entre si es del controlador **Blog** y la acción **Archive**, los establecemos directamente, mientras que dejamos los parámetros **year** y **month** para que reciban valores variables.

# Creación de rutas

**Ruta:** blog/{codigo}

- Esta ruta coincide con la ruta por defecto, por lo que no tendremos que registrarla de nuevo.
- En concreto, la ruta por defecto se define en la siguiente porción de código

```
app.MapControllerRoute(  
    name: "post",  
    pattern: "Blog/{code}",  
    defaults: new { controller = "Blog", action = "ViewPost" });
```

# Creación de rutas

**Ruta:** `blog/{codigo}`

- Concluimos que necesitamos crear una regla en la tabla de rutas para procesar las peticiones de este tipo.
- Además, **tenemos que registrar la ruta en primer lugar, para que se compruebe antes que las demás**

# Creación de rutas

```
app.MapControllerRoute(
    name: "post",
    pattern: "Blog/{code}",
    defaults: new { controller = "Blog", action = "ViewPost" });

app.MapControllerRoute(
    name: "archive",
    pattern: "Blog/Archive/{year}/{month}",
    defaults: new { controller = "Blog", action = "Archive" });

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



## Actividad 3

*Configura las rutas correspondientes para cada una de estas funcionalidades:*

- *Método que nos devuelva todos los Pokémon: **Pokemon/Index***
- *Método al que le pasemos un ID y nos devuelva info de ese Pokémon en específico: **Pokemon/Info/{id}***
- *Método al que le pasemos un peso y una altura y nos devuelva los Pokemon que correspondan: **Pokemon/GetPokemons/{peso}/{altura}***

**10 MINUTOS**



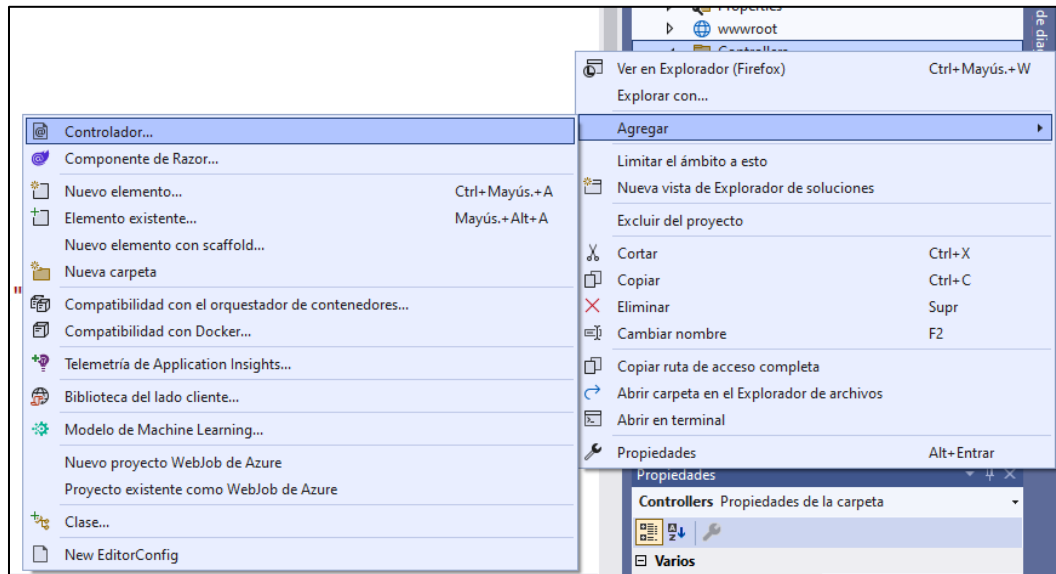
# CREACIÓN DEL CONTROLADOR

# Creación del controlador

- Necesitamos implementar un controlador con estos tres métodos, uno para cada acción definida:
  - **Index()**, que debe retornar al cliente una página mostrando una relación con los últimos artículos publicados.
  - **Archive()**, para visualizar los artículos publicados en un año y mes concretos.
  - **ViewPost()**, cuya misión será mostrar un artículo concreto.

# ► Creación del controlador

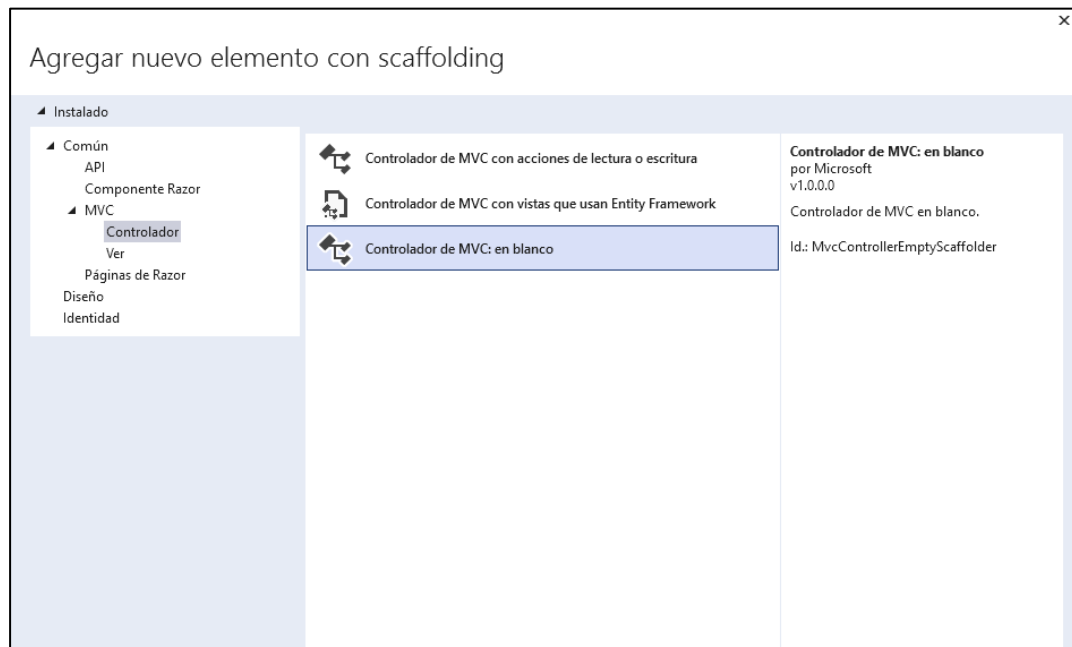
- Para crearlo, vamos a la carpeta /Controllers del proyecto y seleccionamos "Agregar Controlador":





# ► Creación del controlador

- De las opciones que nos salen, seleccionaremos “Controlador de MVC en blanco”



# Creación del controlador

- Le damos un nombre al controlador, y se nos habrá creado esta clase:

```
public class BlogController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
}
```

# Creación del controlador

## Acción Index()

- Esta acción retornará al usuario una página con un resumen de los últimos artículos publicados en el blog.

```
public ActionResult Index()
{
    var manager = new BlogManager(_contexto);
    var posts = manager.GetLatestPosts(10); // 10 ultimos posts
    return View("Index", posts);
}
```

# Creación del controlador

## Acción Archive()

- Esta acción permite obtener artículos publicados un año y mes determinado.
- Como necesitamos obtener la información del año y mes desde la URL, debemos introducir el concepto de los **binders**, que son los encargados de recoger los valores de los parámetros formales y hacer las conversiones de tipo necesarias.
- Por ejemplo, obtendría el año 2023 y el mes 1 de esta URL:

**/blog/archive/2023/1**

# Creación del controlador

## Acción Archive()

```
public ActionResult Archive(int year, int month)
{
    var manager = new BlogManager(_contexto);
    var posts = manager.GetPostsByDate(year, month);
    return View(posts);
}
```

# Creación del controlador

## Acción ViewPost()

- Esta acción retornará al usuario una página con el contenido del artículo cuyo código se incluye en la ruta:

```
public ActionResult ViewPost(string code)
{
    var manager = new BlogManager(_contexto);
    var post = manager.GetPost(code);
    if (post == null)
        return NotFound();
    else
        return View(post);
}
```



## ***Actividad 4***

*Crea el controlador Pokemon correspondiente para que nos redirija cada una de las acciones a las vistas correspondientes*

**10 MINUTOS**



# CREACIÓN DE LAS VISTAS



# Creación de las vistas

- Necesitamos implementar las vistas, es decir, los componentes encargados de generar la interfaz de usuario.
- Necesitamos tres vistas:
  - **ViewPost**, que muestra el contenido completo de un artículo.
  - **Index**, destinada a mostrar una lista con los últimos artículos publicados.
  - **Archive**, que permitirá el acceso al archivo, mostrando los posts creados en un año y mes indicado por el usuario.

# Creación de las vistas

## Vista ViewPost

- En la acción ViewPost() del controlador, obteníamos un post desde el Modelo y, si no había ningún problema, devolvíamos una vista con el objeto de tipo Post a visualizar.
- Para implementar la vista, podemos hacerlo de dos formas:
  - **Forma manual**, poniéndole el mismo nombre que la acción que la va a mostrar
  - **Forma automática**, utilizando las facilidades que nos da Visual Studio

# ► Creación de las vistas

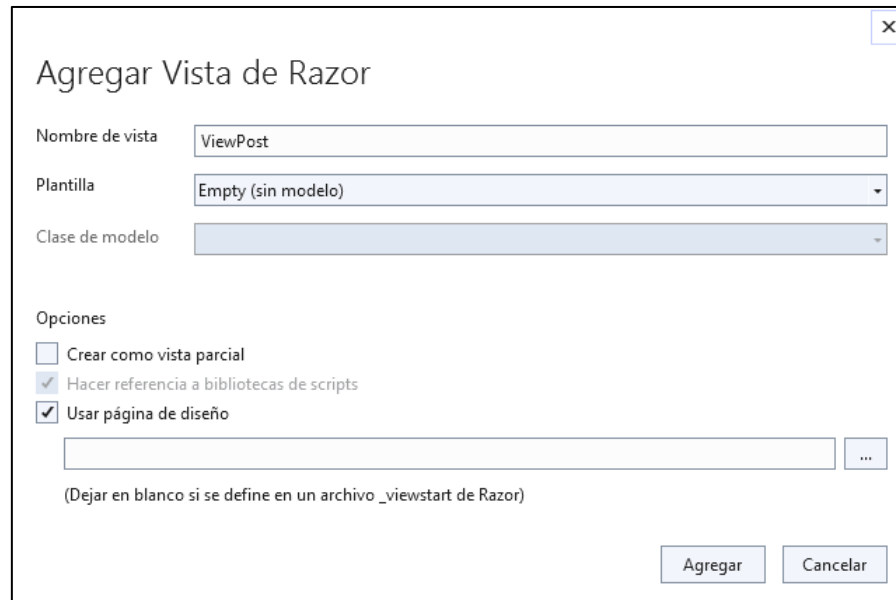
## Vista ViewPost

- Si utilizamos la forma automática, Visual Studio nos facilita mucho la vida, permitiéndonos agregar una vista para cada una de las acciones que tengamos creadas.
- Solamente tenemos que clicar con el botón derecho sobre la acción y seleccionar la opción “Agregar vista”:



# Creación de las vistas

- Nos solicitarán los siguientes datos:



Agregar Vista de Razor

Nombre de vista

Plantilla

Clase de modelo

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)

# Creación de las vistas

- Nos solicitarán los siguientes datos:
  - El **nombre de la vista**. Siguiendo la convención, debería ser **ViewPost**.
  - La **plantilla**: indica qué tipo de contenido deseamos generar automáticamente para la vista, son:
    - **Empty**, el IDE no genera código
    - **Create**, que implica que la vista es un formulario para la creación de una entidad del tipo especificado como "model class".
    - **Edit**, muy similar al anterior, salvo que se refiere a la edición de una instancia existente del tipo especificado.
    - **Details**, que hace que Visual Studio genere código de visualización de la entidad, mostrando el contenido de sus propiedades.
    - **Delete**, generando una vista de confirmación de la eliminación de la entidad.
    - **List**, indicando que se pretende mostrar una lista de entidades. El entorno de desarrollo generará automáticamente código para visualizar en forma de tabla esta información.

# Creación de las vistas

- Nos solicitarán los siguientes datos:
  - **Clase de modelo:** nos permite indicar el tipo de datos concreto (la clase) que contiene los datos que el controlador enviará a la vista.
  - **Checkbox “Crear como vista parcial”:** Nos permite crear vistas que representan el contenido de una porción de la página, lo que permite reutilizar código de vistas y evitar duplicidades.
  - **Checkbox “Hacer referencia a bibliotecas de scripts”:** indica que se deben añadir a la vista referencias (tags <script>) hacia las bibliotecas utilizadas en la misma, como jQuery y algunos plugins necesarios para realizar validaciones de formularios en cliente.
  - Por último, podemos seleccionar el layout o página maestra que utilizará la vista. Si simplemente marcamos la casilla, indicaremos que la nueva vista utilizará el Layout establecido por defecto.

# ► Creación de las vistas

## Vista ViewPost

- El contenido básico de la vista que hayamos decidido generar será este:

```
@model Ejemplo.Models.Post

@{
    ViewBag.Title = "ViewPost";
}

<h2>ViewPost</h2>
```

# ► Creación de las vistas

## Vista ViewPost

- **@model**, especifica el tipo de datos que estamos recibiendo desde el controlador. Es decir, utilizando la propiedad Model podremos acceder a propiedades como **Model.PostId**, **Model.Titulo** o **Model.Texto**

```
@model Ejemplo.Models.Post
```



# Creación de las vistas

## Vista ViewPost

- El bloque @{ } es una sintaxis Razor que nos permite separar código cliente de código servidor.
- **ViewBag** es un contenedor donde podemos almacenar información para compartirla entre varios componentes.

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

# Creación de las vistas

## Vista ViewPost

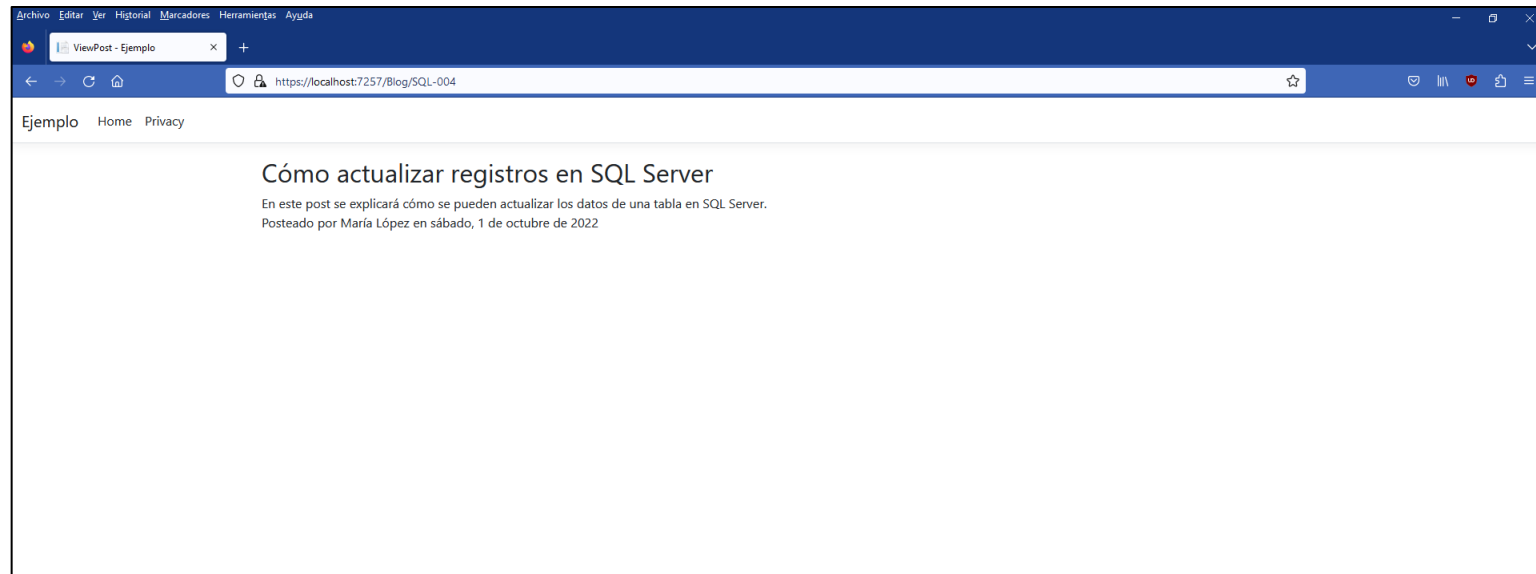
- La vista final nos podría quedar de esta forma:
- En la siguiente diapositiva vemos el resultado de la ejecución:

```
@model Ejemplo.Models.Post

@{
    ViewBag.Title = "ViewPost";
}

<h2>@Model.Titulo</h2>
<div class="post-body"> @Model.Texto
</div>
<p>
    Postado por @Model.Autor
    en @Model.Fecha.ToLongDateString()
</p>
```

# Creación de las vistas



# Creación de las vistas

## Vista Index

- La vista **Index** tiene que mostrar un listado con los últimos posts publicados en nuestro blog.
- Como queremos mostrar una lista de artículos, vamos a utilizar la generación automática de Visual Studio con la opción correspondiente.

# Creación de las vistas

## Agregar Vista de Razor

Nombre de vista

Plantilla

Clase de modelo

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)

# Creación de las vistas

## Vista Index

- La vista final nos podría quedar de esta forma:
- En la siguiente diapositiva vemos el resultado de la ejecución:

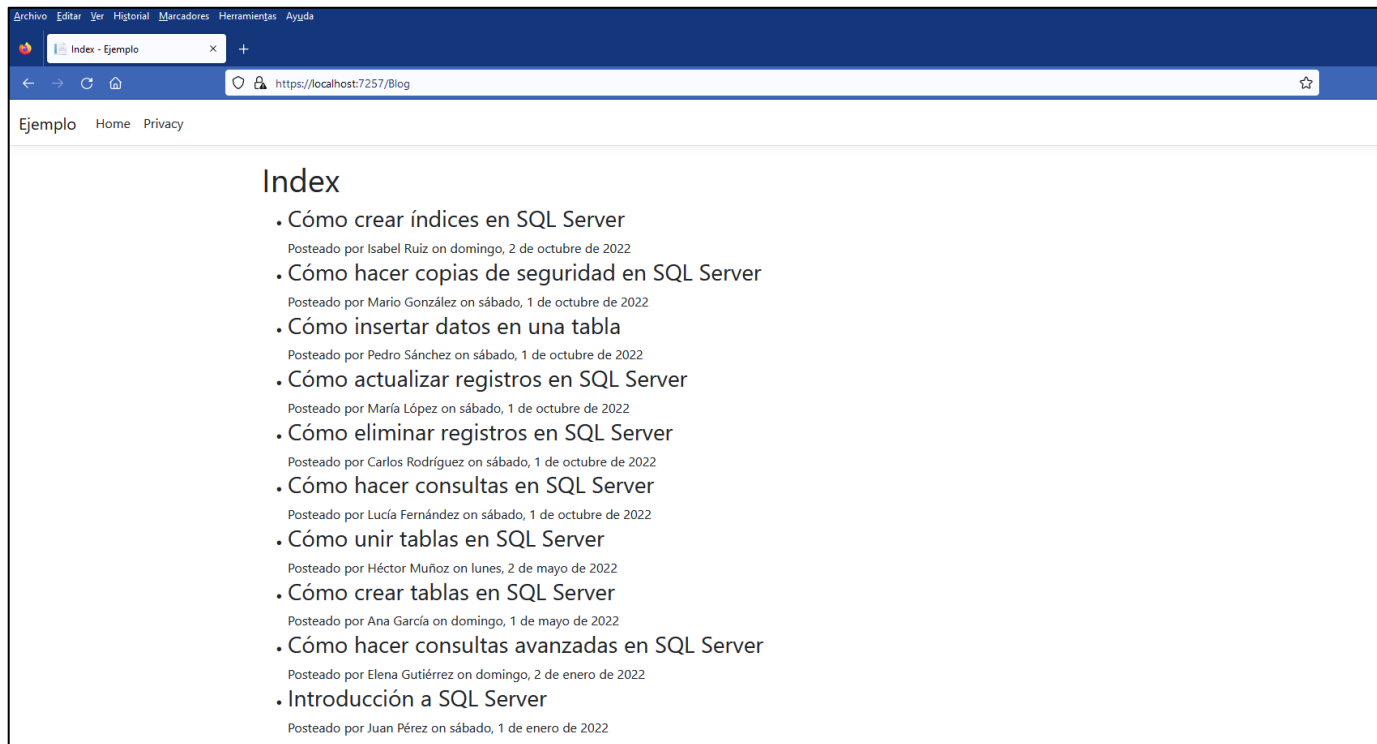
```
@model IEnumerable<Ejemplo.Models.Post>

@{
    ViewData["Title"] = "Index";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h1>Index</h1>

<ul>
    @foreach (var post in Model)
    {
        <li>
            <h3>@post.Titulo</h3>
            <div>
                Posteado por @post.Autor on @post.Fecha.ToLongDateString()
            </div>
        </li>
    }
</ul>
```

# Creación de las vistas



# Creación de las vistas

## Vista Archive

- Esta vista la crearemos igual que la anterior, pues también queremos mostrar una lista de elementos.
- Mostraremos el periodo para el que queremos mostrar la información y los posts publicados.



# Creación de las vistas

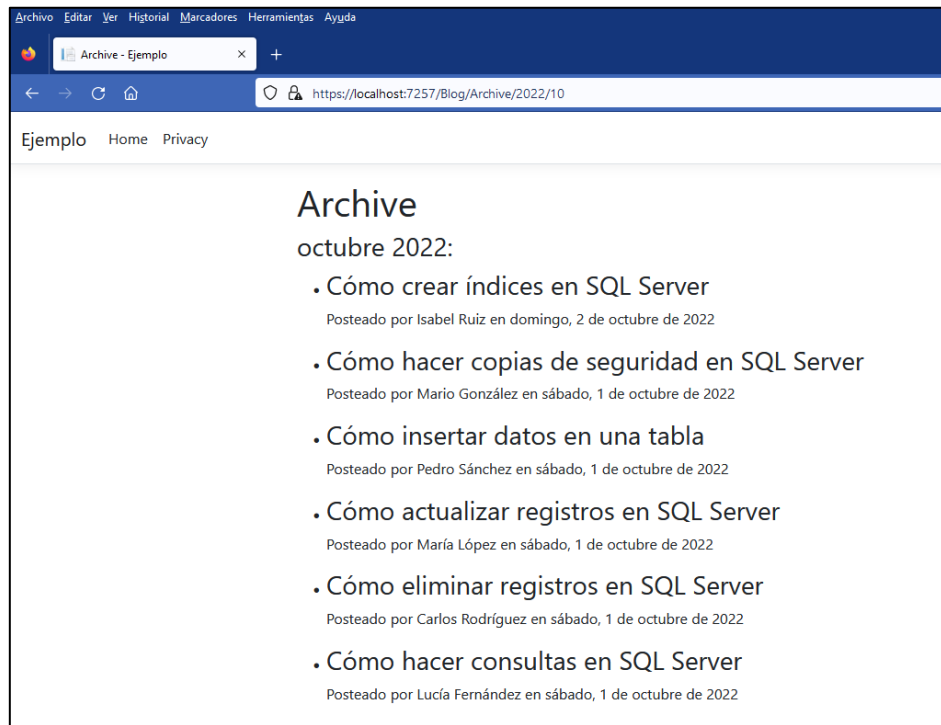
```
@model IEnumerable<Ejemplo.Models.Post>

@{
    ViewData["Title"] = "Archive";
    Layout = "~/Views/Shared/_Layout.cshtml";
}

<h1>@ViewBag.Title</h1>

@{
    var firstPost = Model.FirstOrDefault();
    if (firstPost != null)
    {
        <h3>@firstPost.Fecha.ToString("MMMM yyyy");</h3>
        <ul>
            @foreach (var post in Model)
            {
                <li>
                    <h3>@post.Titulo</h3>
                    <p>Posteado por @post.Autor en @post.Fecha.ToLongDateString()</p>
                </li>
            }
        </ul>
    }
}
```

# Creación de las vistas



# Creación de las vistas

## Helper HTML

- Un helper HTML es una **clase que proporciona métodos para generar elementos HTML en una vista Razor.**
- Estos métodos pueden generar HTML para elementos comunes como formularios, botones, enlaces, etiquetas de texto, etc.

# Creación de las vistas

## Helper HTML

- Son muy útiles porque **reducen la cantidad de código HTML** que se escribe directamente en las vistas y permiten a los desarrolladores crear elementos HTML de forma más rápida y fácil.
- Además, los helpers HTML **generan HTML compatible con los estándares** y aseguran que los nombres de los campos y otros atributos se ajusten correctamente al modelo de datos subyacente.

# Creación de las vistas

## Helper HTML

- Algunos de los helpers HTML más comunes en ASP.NET Core MVC son:
  - **Html.BeginForm**
  - **Html.LabelFor**
  - **Html.TextBoxFor**
  - **Html.CheckBoxFor**
  - **Html.DropDownListFor**

# Creación de las vistas

## Html.BeginForm

- Crea un formulario HTML y establece la acción y el controlador que se ejecutarán cuando se envíe el formulario.
- Veremos un ejemplo a continuación:

# Creación de las vistas

- En este ejemplo, veremos cómo crea un formulario HTML y establece la acción y el controlador que se ejecutarán cuando se envíe el formulario.
- Vemos como la acción es **Crear** y el controlador es **Empleado**. El parámetro `FormMethod.Post` indica que el formulario se enviará utilizando el método POST HTTP.

```
@using (Html.BeginForm("Crear", "Empleado", FormMethod.Post))
{
    <div class="form-group">
        <label for="Name">Nombre:</label>
        <input type="text" class="form-control" name="Name" id="Name" />
    </div>

    <div class="form-group">
        <label for="Email">Email:</label>
        <input type="email" class="form-control" name="Email" id="Email" />
    </div>

    <button type="submit" class="btn btn-primary">Guardar</button>
}
```

# Creación de las vistas

## Html.LabelFor y Html.TextBoxFor

- **Html.LabelFor:** Crea una etiqueta label para una propiedad de modelo
- **Html.TextBoxFor:** Crea un elemento input de tipo texto para una propiedad de modelo.
- Veremos un ejemplo a continuación:



# Creación de las vistas

- En este ejemplo, el helper **Html.LabelFor** crea una etiqueta label para la propiedad Titulo del modelo. El método `p => p.Titulo` se utiliza para obtener el nombre de la propiedad del modelo.
- El helper **Html.TextBoxFor** crea un elemento input de tipo texto para la propiedad Titulo del modelo.

```
@model Ejemplo.Models.Post

<div class="form-group">
  @Html.LabelFor(p => p.Titulo)
  @Html.TextBoxFor(p => p.Titulo, new { @class = "form-control" })
</div>
```

# Creación de las vistas

## Html.CheckBoxFor

- Crea un elemento input de tipo casilla de verificación para una propiedad de modelo booleana.
- Veremos un ejemplo a continuación:

# Creación de las vistas

- En este ejemplo, el helper **Html.CheckBoxFor** crea un elemento input de tipo casilla de verificación para la propiedad booleana `IsSubscribed` del modelo.
- El método `p => p.EsPublicado` se utiliza para obtener el nombre de la propiedad del modelo.

```
@model Ejemplo.Models.Post
<div class="form-group">
  @Html.LabelFor(p => p.EsPublicado)
  @Html.CheckBoxFor(p => p.EsPublicado)
</div>
```

# Creación de las vistas

## Html.DropDownListFor

- Crea un elemento select para una propiedad de modelo y llena el elemento con opciones de una lista de selección.
- Veremos un ejemplo a continuación:

# Creación de las vistas

- En este ejemplo, el helper **Html.DropDownListFor** crea un elemento select para la propiedad Autor del modelo. El segundo parámetro `new SelectList(ViewBag.Autores, "Value", "Text")` se utiliza para llenar el elemento select con opciones de una lista de selección.
- El tercer parámetro `-- Selecciona un autor --` se utiliza para agregar una opción predeterminada con un valor vacío al elemento select.
- El cuarto parámetro `new { @class = "form-control" }` se utiliza para agregar la clase CSS form-control al elemento select.

```
@model Ejemplo.Models.Post
<div class="form-group">
  @Html.LabelFor(p => p.Autor)
  @Html.DropDownListFor(p => p.Autor,
    new SelectList(ViewBag.Autores, "Value", "Text"), "-- Selecciona un autor --", new { @class = "form-control" })
</div>
```



## ***Actividad 5***

*Crea las vistas necesarias para cada una de las acciones implementadas en el*

*Controlador y comprueba que aplicación funciona correctamente.*

*Puedes hacerlas de forma manual o aprovecharte de las funcionalidades de*

*Visual Studio.*

**15 MINUTOS**



# ***Tutorial: Introducción a C# y ASP.NET Core***

<https://learn.microsoft.com/es-es/visualstudio/get-started/csharp/tutorial-aspnet-core?view=vs-2022>



# ***Práctica guiada***

## ***Práctica guiada 2: Añadiendo funcionalidades***

**15 MINUTOS**





**KEEP  
CALM  
IT'S  
KAHOOT  
TIME**



Formación  
Profesional  
FOMENTO

# Modelo-Vista-Controlador (MVC)

CS Desarrollo de Aplicaciones Web