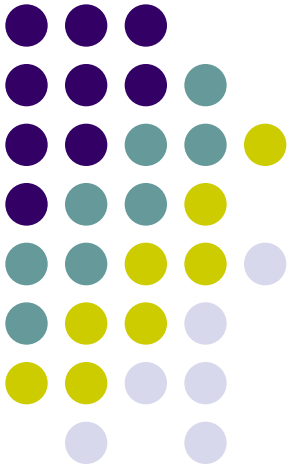
Desarrollo de Aplicaciones Web

Desarrollo Web en Entornos de Servidor



Tema 6

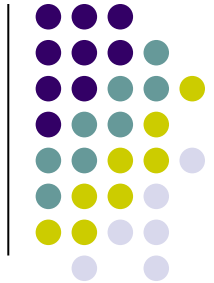
Desarrollo de Aplicaciones Web MVC (I)

Vicente J. Aracil Miralles

vj.aracilmiralles@edu.gva.es

03/09/2024

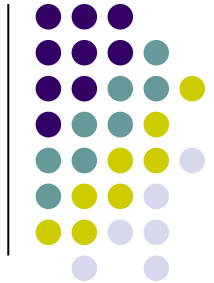
Tema 6
Desarrollo de Aplicaciones Web MVC (I)



Objetivos

- Reconocer los diversos modelos de desarrollo de aplicaciones Web
- Comprender el patrón arquitectónico Modelo-Vista-Controlador (MVC), así como los elementos que lo forman y las interacciones entre ellos
- Identificar las ventajas que aporta un bajo nivel de acoplamiento entre interfaz y lógica
- Identificar y comprender el funcionamiento de los componentes que actúan como modelo, controlador y vista en las aplicaciones basadas en ASP.NET Core MVC

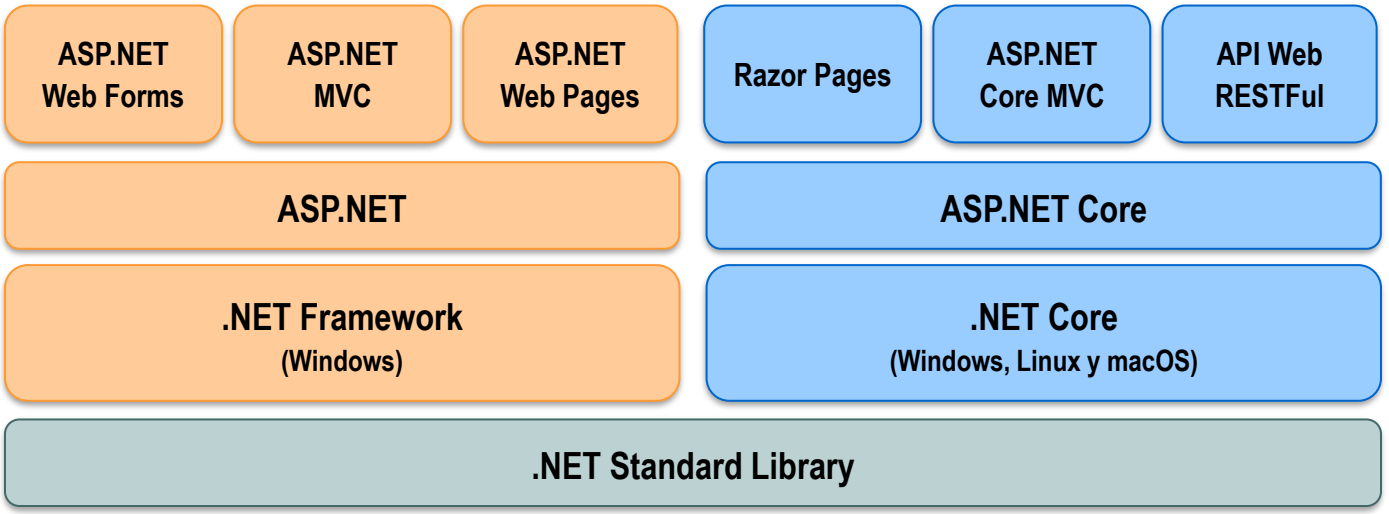
Tema 6
Desarrollo de Aplicaciones Web MVC (I)



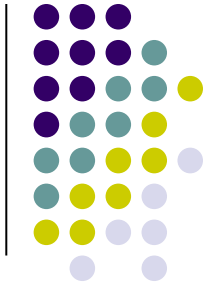
Contenidos

Marco de desarrollo de ASP.NET

- **El Marco de desarrollo de ASP.NET *(ASP.NET Development Framework)***
  - Está concebido para la crear aplicaciones Web interactivas que emplean tecnologías web estándar (HTML, CSS, etc.) y secuencias de código lógico que procesa el servidor Web
  - Admite **diversos modelos de desarrollo de aplicaciones Web**, lo que permite desarrollar diferentes tipos de aplicaciones Web interactivas

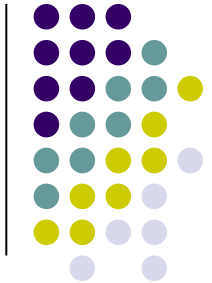


  - La existencia de diversos modelos de desarrollo no implica la substitución entre ellos, sino que coexisten diversos enfoques de implementación que emplean patrones arquitectónicos diferentes
  - Este planteamiento amplía las capacidades de desarrollo de las aplicaciones Web



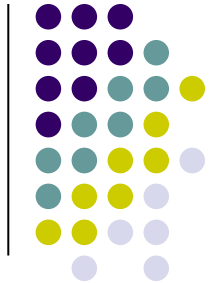
Marco de desarrollo de ASP.NET

- **Algunos tipos de aplicaciones Web basadas en ASP.NET Core**
  - **Razor Pages.** Es el tipo de aplicación Web más simple. La aplicación Web está formada por páginas independientes en las que se combina el contenido estático y el código lógico, utilizando la sintaxis del lenguaje Razor, para crear contenido Web dinámico
  - **ASP.NET Core MVC.** Utiliza ASP.NET Core para poder desarrollar aplicaciones Web basada en el patrón arquitectónico Modelo-Vista-Controlador (MVC). El controlador genera la respuesta que corresponde con las acciones solicitadas por el usuario, de modo que la interfaz no está tan estrechamente relacionada con el resto de los elementos de la aplicación Web
  - **API Web RESTful.** Facilita la creación de servicios HTTP de tipo RESTful
- **Algunos tipos de aplicaciones Web basadas en ASP.NET**
  - **ASP.NET Web Forms.** Es el tipo de aplicación Web originaria de ASP.NET. Se basa en el diseño de la interfaz mediante controles de servidor. El comportamiento de estos controles y del Web Form se especifica a través de código lógico asociado a eventos
  - **ASP.NET MVC.** Utiliza la infraestructura de ASP.NET para poder desarrollar aplicaciones Web basada en el patrón arquitectónico Modelo-Vista-Controlador (MVC)
  - **ASP.NET Web Pages.** Es un tipo de aplicación Web simple y productiva que está basada en páginas. Genera aplicaciones Web similares a las de Razor Pages, pero utilizando ASP.NET



Patrones de software

- **Al desarrollar software es habitual utilizar patrones de software**
  - Son la base para la búsqueda de soluciones a problemas comunes en el ámbito del desarrollo de software
  - Los patrones de software también se denominan, sencillamente, patrones
  - Se trata de esquemas lógicos que tienen un objetivo concreto aplicable a diversas situaciones de desarrollo de software, con independencia de:
    - El tipo de proyecto de software
    - Los lenguajes empleados
    - Las herramientas de desarrollo que se utilicen
  - Los patrones de arquitectura de software, también denominados patrones arquitectónicos, son un tipo de patrones que no afectan a un componente de software concreto sino que **afectan a la estructura general de un proyecto de software**
    - A este grupo de patrones pertenece el patrón arquitectónico MVC (Modelo-Vista-Controlador)

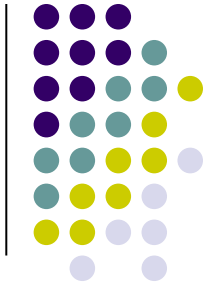


Evolución histórica

- **Orígenes del patrón de arquitectura de software MVC**
  - El patrón MVC fue introducido por **Trygve Reenskaug** en 1979. Este científico de ciencias de la computación y profesor de la Universidad de Oslo, es autor del primer documento[1] en que se formula una separación de los elementos de un sistema de software en: un modelo, un software controlador y una vista manipulable por el usuario para el diseño de una interfaz gráfica
  - En 1983, se realizó la primera implementación MVC como patrón arquitectónico para el lenguaje de programación Smalltalk-80
  - Y posteriormente, el patrón MVC se expresó formalmente como concepción general
  - A partir de ese momento comenzó la extensión y popularidad de su uso como patrón arquitectónico que puede aplicarse para la construcción de aplicaciones informáticas independientemente del lenguaje de programación utilizado

Nota de Referencia al documento:

[1] http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html



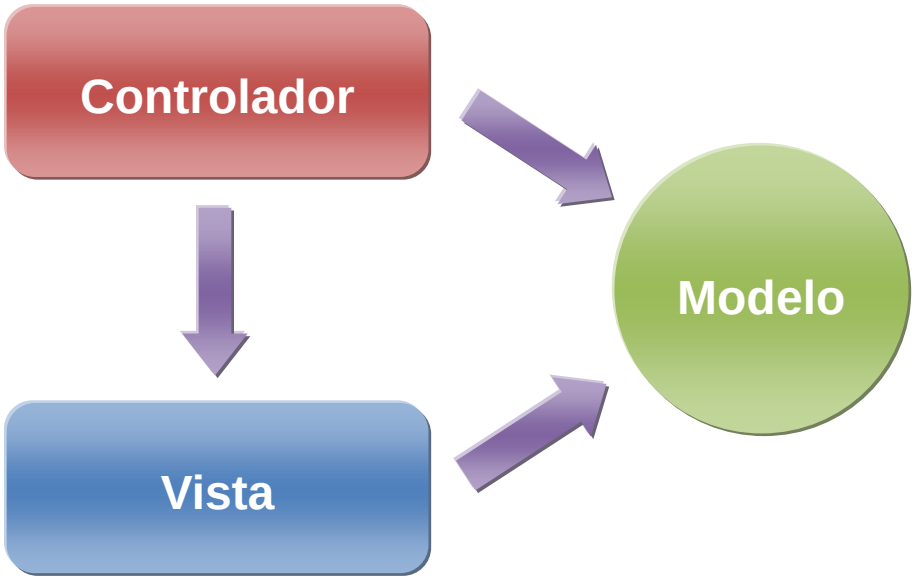
Algunos *Frameworks* MVC utilizados actualmente

Framework MVC	Lenguaje	Licencia
Ruby on Rails	Ruby	MIT
Spring MVC	Java	Apache
Struts	Java	Apache
JSR	Java	Oracle
AngularJS	Javascript	MIT
Symfony	PHP	MIT
CakePHP	PHP	MIT
Cocoa	ObjectiveC	Apple
Django	Python	BSD
ASP.NET MVC	C#, VB, etc. (.NET Framework)	Microsoft
ASP.NET Core MVC	C# (.NET Core)	Microsoft

Arquitectura de software Modelo-Vista-Controlador

- **El patrón arquitectónico Modelo-Vista-Controlador (MVC) se basa en separar la lógica de la aplicación de la interfaz de usuario**
  - Por un lado se definen componentes para la representación de la información
  - Y, por otro lado se definen los componentes para la interacción del usuario

Para ello, el patrón MVC propone que la arquitectura de una aplicación esté compuesta por tres partes bien diferenciadas que son: el Modelo, la Vista y el Controlador



Arquitectura de software Modelo-Vista-Controlador

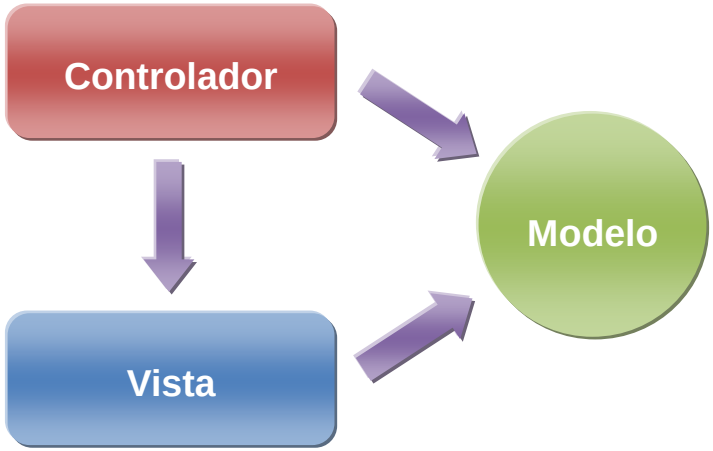
- **Elementos fundamentales**

Modelo. Representa el estado de la aplicación (información que maneja la aplicación en un momento dado) y proporciona los medios para consultar y actualizar (insertar, eliminar y modificar) dicha información. Puede estar constituido por una serie de clases que representan las entidades de una base de datos, junto con sus propiedades y los métodos para actuar sobre ellas

Controlador. Procesa la interacción del usuario. Se encarga de responder a las acciones del usuario, traduciéndolas en alteraciones sobre la vista o sobre el modelo. Se podría decir que hace de intermediario entre la vista y el modelo. Es habitual que exista un controlador por cada funcionalidad de la aplicación. Los controladores:

- Manejan la interacción con el usuario
- Actúan sobre el modelo
- Seleccionan la vista a desplegar

Un controlador puede estar constituido por una clase que incluye una serie de métodos

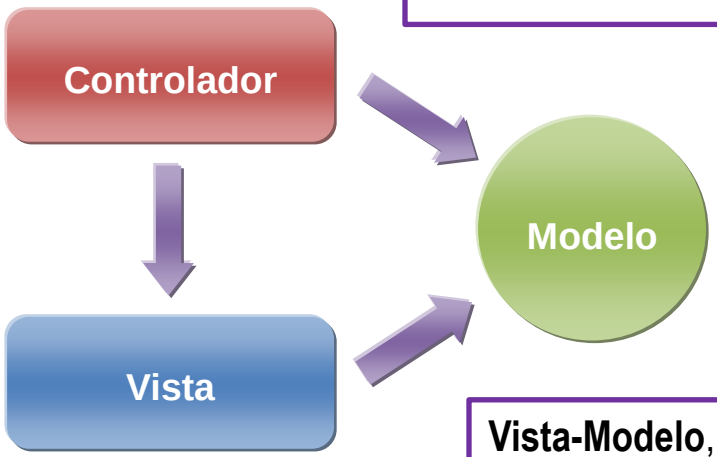


Vista. Genera o "pinta" la interfaz de usuario a partir de la información obtenida del modelo. La vista cambiará a demanda del controlador, cuando una acción de usuario así lo requiera. Habitualmente, se utiliza un lenguaje o motor de vistas para generar las representaciones visuales del estado de la aplicación

Arquitectura de software Modelo-Vista-Controlador

- **Conexiones entre los elementos**

Controlador-Modelo. Se emplea, principalmente, para que el controlador actúe sobre el estado de la aplicación. El acceso a la información que maneja la aplicación es responsabilidad del modelo, de manera que el modelo es el único responsable de establecer las conexiones sobre una base de datos y de ejecutar las consultas o modificaciones de datos cuando el controlador lo demande



Controlador-Vista. El controlador se comunica con la vista para establecer la interfaz de usuario

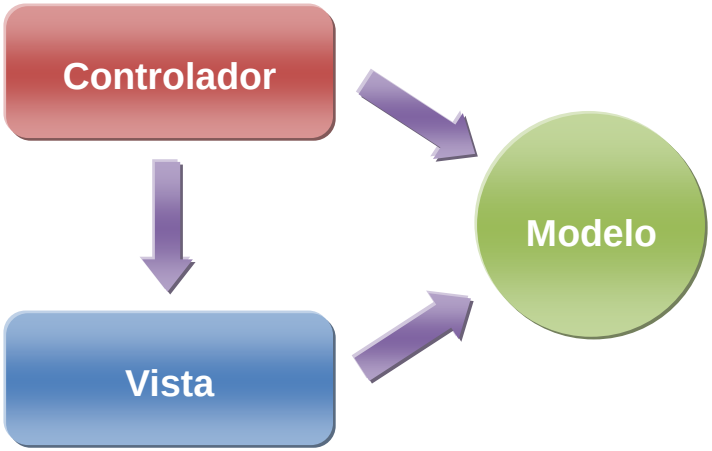
Vista-Modelo, la vista usará la conexión con el modelo para obtener información desde el modelo, con la finalidad de poder componer la interfaz de usuario

Arquitectura de software Modelo-Vista-Controlador

- **Bajo acoplamiento entre componentes**

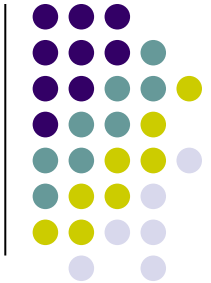
  Es decir, los componentes de la aplicación son bastante independientes entre sí:

  - En una aplicación MVC ni el controlador ni la vista tienen acceso directo al modelo, no conocen las clases concretas que lo implementan, sino que se comunican con él a través de una interfaz pública expuesta por el modelo
  - Igualmente, el controlador se comunica con la vista a través de una interfaz pública



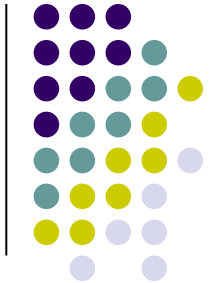
- **Ámbitos de aplicación de la arquitectura MVC**
  - Aunque originalmente el patrón arquitectónico MVC fue desarrollado para aplicaciones de escritorio, su uso ha sido ampliamente aceptado como arquitectura para diseñar e implementar aplicaciones Web interactivas. Se han desarrollado multitud de *Frameworks* Web que implementan este patrón
  - Es necesario tener en cuenta que el patrón MVC es aplicable exclusivamente a sistemas software en los que **se precisa interacción por parte del usuario**, no teniendo sentido fuera de dicho contexto

Características y ventajas del patrón de arquitectura MVC



- **El patrón arquitectónico MVC favorece el diseño de sistemas de software que permiten obtener un bajo nivel de acoplamiento y, al mismo tiempo, un alto nivel de cohesión entre los componentes de software que los forman**
  - El bajo nivel de acoplamiento entre los componentes del software se refiere a que **cada componente puede resolver su funcionalidad completamente**, sin necesidad de recurrir a otros componentes
    - En este tipo de aplicaciones se reduce la dependencia entre los diferentes componentes
  - La alta cohesión entre los componentes se refiere a que **cada componente ofrece una funcionalidad clara y específica** sobre una estructura de datos concreta.
    - En este tipo de aplicaciones solo el modelo representa y gestiona el estado de la información que se maneja, solo la vista genera representaciones visuales de dicho estado sobre la interfaz y, solo el controlador gestiona la interacción del usuario y selecciona la vista a desplegar

  Habitualmente, se considera que conseguir estas características simultáneamente suele ser uno de los objetivos principales de los procesos de ingeniería del software

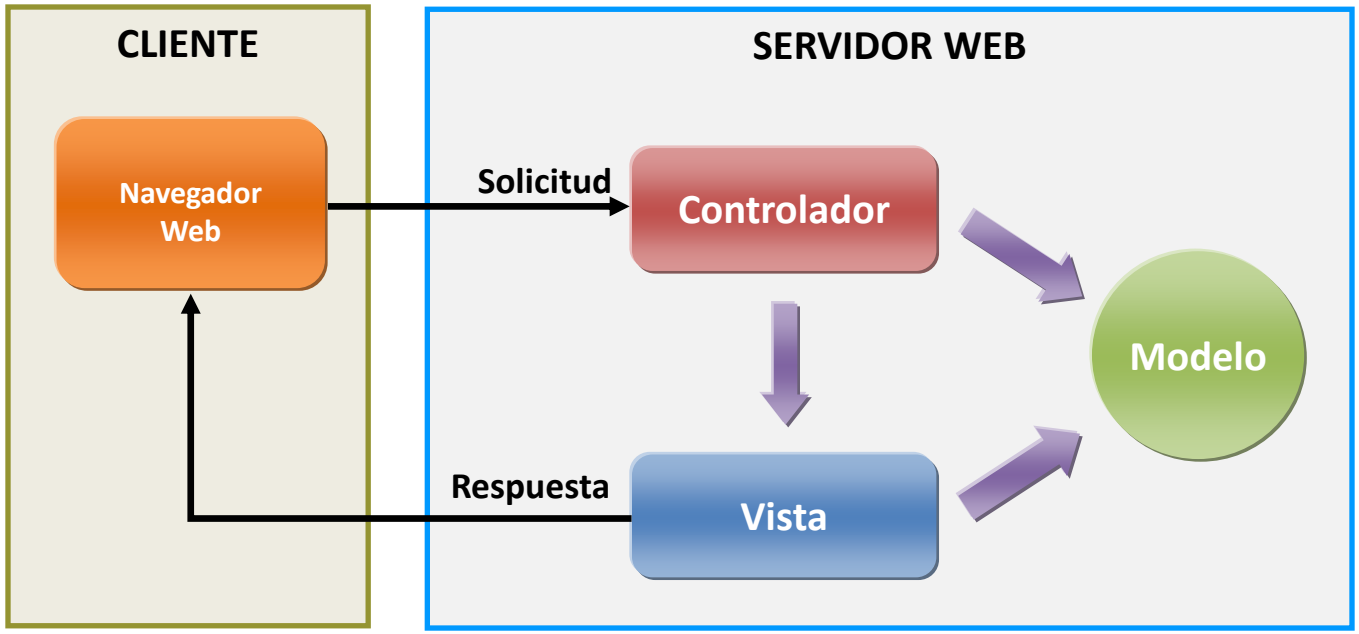


Características y ventajas del patrón de arquitectura MVC

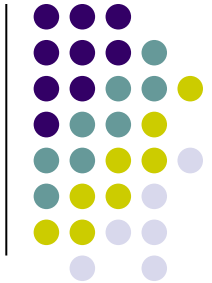
- **Algunas de las ventajas más importantes:**
  - **Ayuda a manejar la complejidad del desarrollo de una aplicación Web**, ya que permite centrarse en cada uno de los aspectos del desarrollo
  - **Posibilita la división del proyecto de desarrollo de software en partes** de forma que cada una de ellas puedan ir desarrollándose en paralelo, por varios equipos de desarrolladores diferentes. Incluso cabe la posibilidad de abordar el desarrollo de la vista, el controlador y el modelo como tres proyectos diferentes. En estos casos, lo único que precisan conocer los diferentes equipos de desarrollo son las interfaces públicas que harán posible la comunicación entre los componentes
  - **Evita la duplicación de código en distintas partes de la aplicación**, porque cada componente tiene asignada una funcionalidad o responsabilidad clara y concreta. Se facilita la reutilización de código
  - **Simplifica la realización de las pruebas del software**, ya que es posible realizar pruebas completas sobre el modelo, la vista y el controlador de manera independiente
  - **Facilita el mantenimiento y los cambios futuros en la aplicación**, al posibilitar la modificación y la sustitución de cualquier componente sin que esto afecte al resto de componentes de la aplicación. Así, por ejemplo, para actualizar la interfaz de usuario empleando nueva tecnología Web, solamente habría que trabajar sobre la vista, sin que ello afecte a las demás partes de la aplicación

La utilización del patrón arquitectura del software MVC puede proporcionar ventajas que se traducen en un incremento de la productividad durante el proceso del desarrollo del software

Dinámica del proceso de interacción en una Aplicación Web basada en ASP.NET Core MVC



1. La solicitud realizada en el cliente desencadena la ejecución de una acción en el controlador. El controlador puede acceder al modelo para consultar o actualizar (añadir, eliminar o modificar) la información requerida
2. A continuación, es responsabilidad del controlador activar la vista que corresponda, en función de los parámetros asociados a la solicitud
3. Finalmente, la vista activada por el controlador genera la página HTML que se envía como respuesta al cliente. Las vistas también pueden acceder al modelo para obtener información del estado y poder componer la interfaz resultante del procesamiento



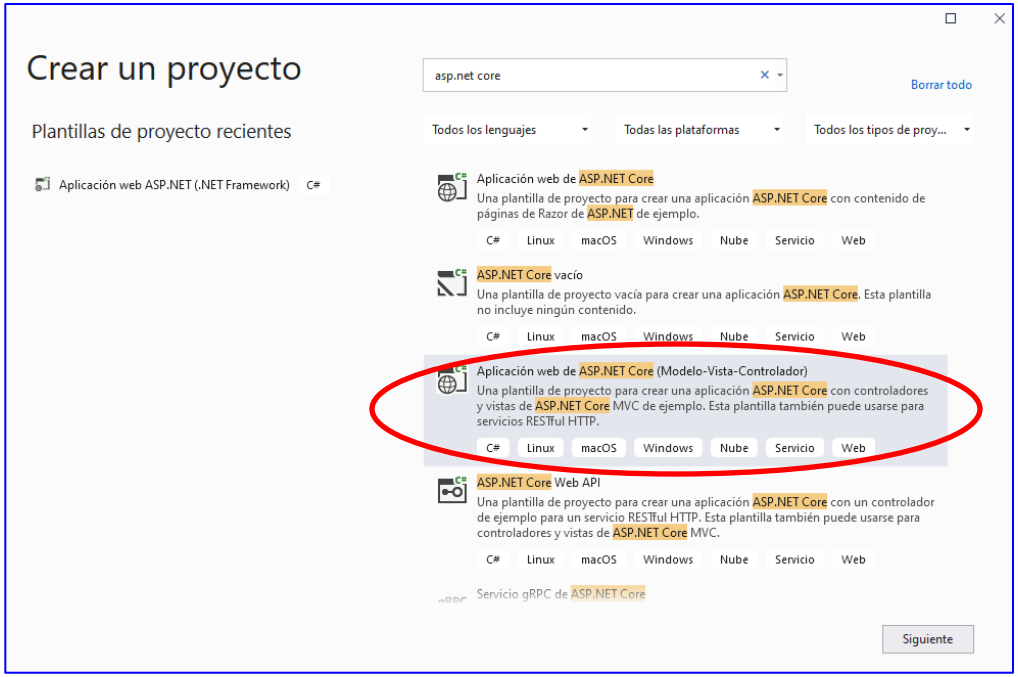
Diferencias entre la aplicaciones Web basadas en Web Forms y las basadas en ASP.NET Core MVC

- Diferencias más significativas:
  - La diferencia principal reside en el **modelo de arquitectura del software** que se emplea
    - En las aplicaciones Web basada en ASP.NET Web Forms, cada página de ASP.NET (.aspx) va forzosamente unida a un archivo de código subyacente (.aspx.cs). De manera que la definición de la interfaz se encuentra directamente asociada con el código lógico que se gestiona a través de eventos. El resultado es un alto nivel de acoplamiento entre la interfaz y la lógica de la aplicación
    - Las aplicaciones Web basadas en ASP.NET Core MVC tienen un bajo nivel de acoplamiento entre sus componentes, porque estos se comunican entre sí a través de una interfaz pública
  - Las aplicaciones Web basadas ASP.NET Web Forms utilizan herramientas RAD *(Rapid Application development)*, mientras que con ASP.NET Core MVC **el trabajo de desarrollo consiste en escribir código** asociado al modelo, a los controladores y a las vistas
  - En cuanto a la **tecnología empleada**, ASP.NET Core MVC propone un enfoque tecnológicamente avanzado y proporciona las ventajas del desarrollo multiplataforma

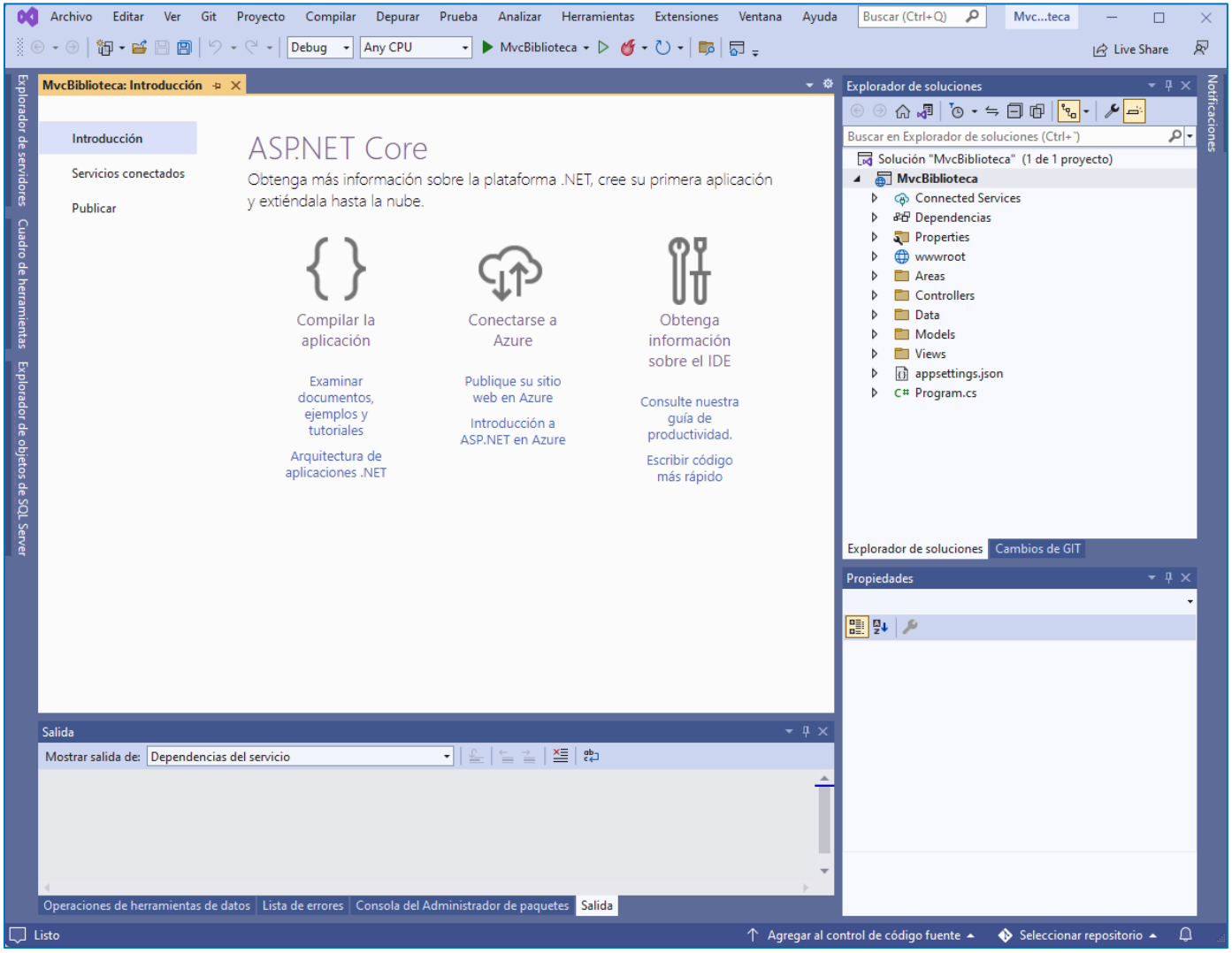
La concepción y el funcionamiento de las aplicaciones Web basadas en ASP.NET Core MVC son totalmente diferentes a los de las aplicaciones Web basadas en Web Forms

ASP.NET Core MVC y Visual Studio

- **La tecnología ASP.NET Core MVC es una capa de software que facilita el desarrollo de aplicaciones Web basadas en el patrón arquitectónico Modelo-Vista-Controlador**
- **Visual Studio utiliza plantillas de proyecto predeterminadas para crear las aplicaciones Web**
  - Al crear un nuevo proyecto de ASP.NET Core MVC, un asistente genera un prototipo básico de aplicación Web que incluye: varios controladores y vistas predeterminados, la página de inicio, el menú de la aplicación, el sistema de autentificación de usuarios, etc.
  - Por razones de productividad, una aplicación Web basada en ASP.NET Core MVC se creará siempre utilizando la plantilla de proyecto correspondiente, por lo que es necesario conocer su estructura y el funcionamiento de sus componentes

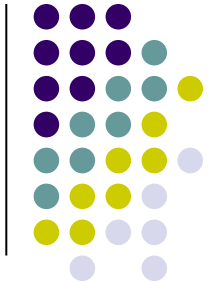


Estructura de aplicación Web basada en ASP.NET Core MVC



Al crear un nuevo proyecto de Aplicación de ASP.NET Core MVC:

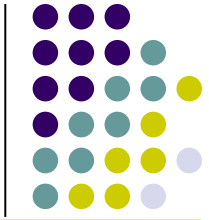
- Se crea una estructura básica de carpetas y archivos que forman el Proyecto de aplicación Web
- Dependiendo del tipo de autentificación que se seleccione, también pueden incluirse varios componentes que implementan la seguridad de acceso a la aplicación Web, mediante autentificación de cuentas de usuario



Estructura de aplicación Web basada en ASP.NET Core MVC

- Los nombres de las carpetas que forman un Proyecto de ASP.NET Core MVC, así como los nombres de los archivos que se almacenan en estas carpetas, de forma predeterminada, no son arbitrarios
  - El funcionamiento de una aplicación Web basada en ASP.NET Core MVC se basa en esa convención de nombres o nomenclatura para poder localizar a cada componente de la aplicación Web
  - La funcionalidad de cada una de las carpetas de un Proyecto de ASP.NET Core MVC es:

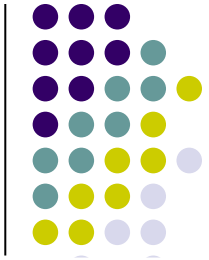
Carpeta	Funcionalidad básica
/Dependencias	Contiene las referencias a los paquetes y las librerías preinstaladas que son necesarias para la solución
/Properties	Contiene algunos archivos de configuración del proyecto
/wwwroot	Incluye el contenido estático y visual de la aplicación Web: archivos de hojas de estilo externas; archivos y librerías Javascript (*.js) que se ejecutan en el cliente (jQuery, Bootstrap, etc./, archivos de imágenes que utiliza la interfaz y otros archivos estáticos



Estructura de aplicación Web basada en ASP.NET Core MVC

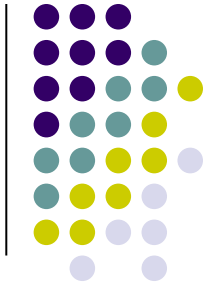
Carpeta	Funcionalidad básica
/Areas	Permite definir diferentes áreas en una aplicación Web para organizar el enrutamiento. Cada área es una parte funcional de una aplicación que incorpora la misma estructura de archivos y carpetas que una aplicación Web basada en ASP.NET Core MVC. Las áreas suelen usarse para estructurar funcionalidades en aplicaciones Web complejas.
/Controllers	Contiene los controladores destinados a procesar las acciones. Cada controlador está representado por una clase que incorpora diversos métodos. Cada uno de estos métodos especifica las acciones a ejecutar cuando son invocados, por este motivo reciben la denominación de **métodos de acción** o **acciones**. Lo más habitual es que los controladores devuelvan la ejecución de una vista, sobre la que se inyecta información proveniente del modelo, aunque también pueden devolver otras acciones. La denominación estándar de los controladores finaliza con el sufijo Controller, por ejemplo: HomeController.cs
/Data	Contiene la definición de las clases de los contextos de datos de la aplicación Web.
/Models	Incluye la definición de las clases de datos que representan el modelo. Estas clases definen las entidades de datos con la que trabaja la aplicación Web
/Migrations	Contiene los archivos necesarios para la realización de migraciones de datos

Estructura de aplicación Web basada en ASP.NET Core MVC



Carpeta	Funcionalidad básica
/Views	Contiene las vistas que son archivos (.cshtml) donde se especifica código HTML estático entremezclado con áreas de código ejecutadas en el servidor. Este código se escribe empleando un lenguaje de vistas, que suele ser Razor. De manera que **una vista genera dinámicamente la una página de respuesta en HTML** Las vistas se agrupan en subcarpetas denominadas zonas que se corresponden con los nombres de los controladores pero sin la terminación Controller. A su vez, en cada zona se define un archivo de vista por cada método de acción del controlador que devuelve la ejecución de una vista. El nombre del archivo de la vista y del método de acción correspondiente debe coincidir, de modo que la ejecución de ese método de acción devolverá la ejecución de la vista correspondiente La subcarpeta **/Views/Shared** no corresponde a ningún controlador. Contiene las plantillas de diseño comunes y las vistas parciales que van a ser reutilizadas por otras vistas. Además, puede alojar vistas compartidas por distintos controladores, lo que evita tener que duplicar vistas en distintas zonas de la carpeta /Views cuando controladores distintos activan la misma vista

- Además de las carpetas anteriores, la estructura predeterminada de un proyecto de ASP.NET Core MVC incluye los archivos ***appsettings.json*** y ***Programs.cs*** que especifican diversos aspectos relativos a la configuración y el inicio de la aplicación Web

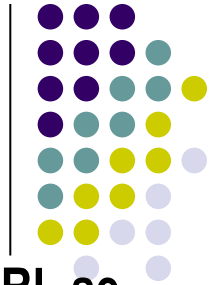


Configuración del enrutamiento y de la página de inicio

- **Formato del enrutamiento MVC de las solicitudes URL**
  - **La ejecución de un método de acción de un controlador se desencadena cuando el servidor Web recibe una solicitud URL concreta**
  - La tecnología ASP.NET Core MVC utiliza un formato URL predeterminado para especificar el método de acción cuyo código se va a ejecutar. Este formato define la lógica del enrutamiento MVC de las direcciones URL entrantes o solicitudes URL
  - El formato predeterminado del **enrutamiento MVC o mapeo MVC** es el siguiente:

    /[NombreControlador]/[NombreMétodoDeAcción]/[Parámetros]

  - **Ejemplo**. Al solicitar la dirección URL http://localhost:xxxx/Libros/Index se especifica que:
    - Se ejecutará el método de acción ***Index()*** del controlador ***Libros*** que se encuentra implementado en la clase ***LibrosController.cs*** de la carpeta **/Controllers**
    - Además, si el método *Index()* devolviera la ejecución de una vista como resultado, que es el caso más habitual, entonces, al finalizar la ejecución de la acción, se ejecutará la vista denominada ***Index.cshtml*** que está ubicada en la subcarpeta o zona de vistas **/Views/Libros**, generándose la página de respuesta HTML resultante



Configuración del enrutamiento y de la página de inicio

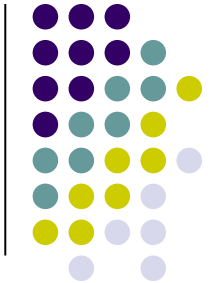
- **La configuración del formato del enrutamiento MVC de las solicitudes URL se establece al iniciar la aplicación Web, cuando se ejecuta el archivo *Program.cs***

```
. . .

app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

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

app.Run();
```

La propiedad *pattern* establece el formato URL y la ruta URL por defecto del enrutamiento MVC

- Además del formato del enrutamiento, el método *MapControllerRoute()* también define la **ruta URL predeterminada** que se desencadenará cuando un usuario acceda a la aplicación Web. Se puede apreciar que la ruta URL predeterminada conducirá a la **acción *Index()*** del **controlador *Home***
  - La solicitud URL http://localhost:xxxx/ produce el mismo resultado que http://localhost:xxxx/Home/Index
  - El enrutamiento por defecto de la acción Index afecta a todos los controladores. La solicitud de la dirección URL http://localhost:xxxx/Libros/Index producirá el mismo efecto que http://localhost:xxxx/Libros. En ambos casos, se procesará la acción *Index()* del controlador *Libros*

Concepto. Clases de datos y Clase del contexto de datos



- **Es una representación en objetos de la información que maneja la aplicación**
  - En las aplicaciones Web de ASP.NET Core MVC se utiliza ***Entity Framework Core***
    - EF Core es un framework ORM *(Object-Relational Mapping)* que se utiliza para administrar y simplificar el acceso a una base de datos desde un modelo conceptual de datos de alto nivel
- El modelo se define mediante archivos de clase que pueden ser de dos tipos:
  - **Clases de datos** o clases de entidad de datos. Son las clases del modelo que actúan como entidades de datos. Se almacena en la carpeta /Models. Cada clase de datos expone sus propiedades, aunque también puede exponer métodos y atributos. De manera que:
    - Cada instancia de un objeto de una clase de datos corresponde con una fila de una tabla de la base de datos que maneja la aplicación Web
    - Cada propiedad de una clase de datos se asignará a una columna de la fila correspondiente
  - **Clase del contexto de datos**. Es una clase del modelo que se encarga de buscar, almacenar y actualizar las instancias de las clases de datos en la base de datos que maneja la aplicación Web. Se almacena en la carpeta /Data. Se trata de una clase derivada de la clase base **DbContext** proporcionada por EF Core
    - Los nombres de las propiedades **DbSet** se usan como nombres de las tablas en la base de datos

Ejemplo: MvcBiblioteca. Clases de datos y Clase del contexto

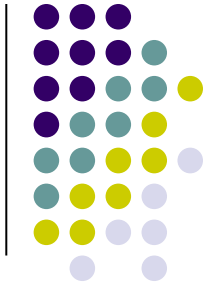
```
public class Libro
{
    public int Id { get; set; }
    public string? Titulo { get; set; }
    public DateTime? FechaEdicion { get; set; }
    public int? NumeroPaginas { get; set; }
    public int AutorId { get; set; }
    public int GeneroId { get; set; }

    public Autor? Autor { get; set; }
    public Genero? Genero { get; set; }
}
```

```
public class Genero
{
    public int Id { get; set; }
    public string? Descripcion { get; set; }

    public ICollection<Libro>? Libros { get; set; }
}
```

```
public class Autor
{
    public int Id { get; set; }
    public string? Nombre { get; set; }

    public ICollection<Libro>? Libros { get; set; }
}
```

```
public class MvcBibliotecaContexto : DbContext
{
    public MvcBibliotecaContexto(DbContextOptions<MvcBibliotecaContexto> options)
        : base(options)
    {
    }

    public DbSet<Autor>? Autores { get; set; }
    public DbSet<Genero>? Generos { get; set; }
    public DbSet<Libro>? Libros { get; set; }
}
```

Clase del Contexto de datos



Asociar el contexto de datos con una base de datos

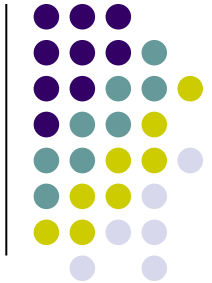
- **Para acceder y poder trabajar con los datos almacenados en una base de datos desde el modelo, es necesario establecer una asociación entre ambos**
  - En primer lugar, se debe **registrar el contexto de datos** en el contenedor de dependencias cada vez que se inicie la ejecución de la aplicación Web. Para realizar este registro, se especifica el código que se muestra resaltado a continuación en archivo de inicio *Program.cs*

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using MvcBiblioteca.Data;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(connectionString));

// Registro del contexto de datos
builder.Services.AddDbContext<MvcBibliotecaContexto>(options =>
    options.UseSqlServer(connectionString));

builder.Services.AddDatabaseDeveloperPageExceptionFilter();

. . .
```

Nombre de la Cadena de conexión

Nombre de la Clase del contexto de datos

Asociar el contexto de datos con una base de datos



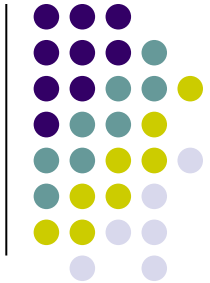
- Y, en segundo lugar, se debe **especificar las características de la cadena de conexión** que enlaza la aplicación Web con la base de datos correspondiente en el archivo *appsettings.json*

Nombre de la Cadena de conexión

Nombre de la Base de datos

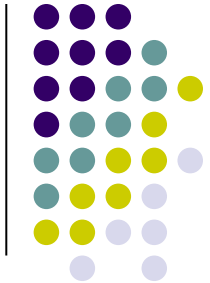
```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=(localdb)\\MSSQL15;Database=aspnet-MvcBiblioteca-DF851F3D-3794-4722-99A1-18DE4AB21DC7;Trusted_Connection=True;MultipleActiveResultSets=true"
  },
"Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

- Dependiendo de cómo se haya creado la aplicación Web, la cadena de conexión *DefaultConnection* puede aparecer configurada de manera predeterminada. Se recomienda utilizar la cadena de conexión predeterminada *DefaultConnection*, si aparece configurada en el archivo *appsettings.json*
- El **archivo de base de datos se ubicará en la carpeta raíz del usuario actual** */Users/{nombre_usuario}* y tendrá el mismo **nombre especificado en el atributo *Database*** del archivo *appsettings.json*



Convenciones de EF Core para especificar el modelo

- **A la hora de especificar el código de las clases del modelo de una aplicación Web, es necesario tener en cuenta las convenciones que establece *EF Core***
  - La existencia de estas convenciones permite minimizar la cantidad de código a escribir
  - El comportamiento predeterminado de estas convenciones se puede modificar
  - Algunas de las convenciones de EF Core más importantes son las siguientes:
    - Los nombres de las propiedades *DbSet* de la clase del contexto de datos se usan como nombres de **tabla.** En el ejemplo anterior, se dispondrá de las tablas: *Autores*, *Generos* y *Libros*
    - Los nombres de las propiedades de una clase de datos se usan como nombres de **columna**. Por ejemplo, la tabla *Libros* dispondrá de la columna o campo *FechaEdicion*
    - Una propiedad de una clase de datos que se denomina ***ID*** o ***Id*** se reconoce que actuará como **clave principal**. Por ejemplo, la propiedad *Id* en la clase de datos *Libro* será interpretada por EF Core como una columna de clave principal
    - Una propiedad de una clase de datos se interpreta como **clave ajena** si se denomina ***<nombre de la propiedad de navegación><nombre de la propiedad de clave principal>***. Por ejemplo, la propiedad *AutorId* de la entidad *Libro* se reconoce como clave ajena para la propiedad de navegación *Autor*, dado que la clave principal de la entidad *Autor* es *Id*

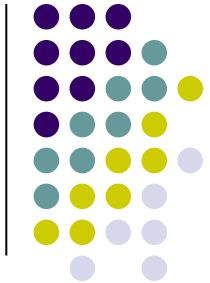


Validación de datos en aplicaciones Web basadas en ASP.NET Core MVC

- **Las condiciones de validación de los datos están vinculadas al modelo**
  - Se emplea un esquema declarativo basado en atributos para establecer las anotaciones de validación en el código del modelo. Los atributos que se pueden emplear para establecer las anotaciones de validación se definen en el espacio de nombres *System.ComponentModel.DataAnnotations*

Atributo	Descripción de la regla de validación
Compare	Proporciona un atributo que compara dos propiedades
CreditCard	Valida un formato de un número de tarjeta de crédito
DataType	Especifica el nombre de un tipo adicional asociado con un campo de datos
Display	Permite especificar el texto de los campos asociados a la propiedad
EmailAddress	Valida un formato de una dirección de correo electrónico
MaxLength	Especifica la longitud máxima del campo de datos
MinLength	Especifica la longitud mínima del campo de datos
Range	Especifica las limitaciones numéricas de rango para el valor de un campo
RegularExpression	El valor del campo debe coincidir con la expresión regular establecida
Required	Especifica que se requiere un valor en el campo de datos
StringLength	Establece la longitud máxima del campo de datos
UrlAttribute	Valida un formato de una dirección URL

Validación de datos en aplicaciones Web basadas en ASP.NET Core MVC

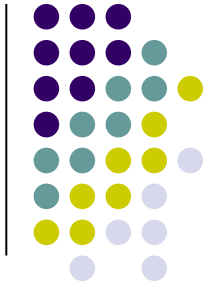


- Las anotaciones de validación se escriben en el código de una clase de datos
  - Las anotaciones de validación se escriben inmediatamente antes de la declaración de la propiedad a la que afectan

```
public class Libro
{
    public int Id { get; set; }
    [Display(Name = "Título")]
    [Required(ErrorMessage = "El título del libro es un campo requerido.")]
    public string? Titulo { get; set; }
    [Display(Name = "Fecha de Edición")]
    [DataType(DataType.Date)]
    public DateTime? FechaEdicion { get; set; }
    [Display(Name = "Núm.Páginas")]
    public int? NumeroPaginas { get; set; }
    [Display(Name = "Autor Principal")]
    public int AutorId { get; set; }
    [Display(Name = "Género")]
    public int GeneroId { get; set; }

    public Autor? Autor { get; set; }
    public Genero? Genero { get; set; }
}
```

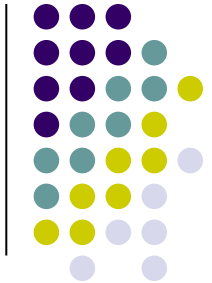
Anotaciones de validación que afectan a la propiedad FechaEdicion

Una vez definidas las anotaciones de validación de una propiedad, las condiciones de validación declaradas se aplican inmediatamente a todos los campos de los formularios asociados al valor de esa propiedad



Operaciones de migración

- **¿Qué ocurrirá si una vez que se ha creado la base de datos se agregan nuevas propiedades a una entidad de datos, o se cambia el tipo de una propiedad o se elimina una propiedad?**
  - Al ejecutar el proyecto se produciría un error, porque existirán inconsistencias entre el esquema de la base de datos y el modelo actual que usa la aplicación
  - **En todo momento, debe existir una correspondencia entre la especificación del modelo de la aplicación Web y la estructura del esquema de la base de datos**
    - Cuando se inicia el desarrollo de una aplicación a partir de una base de datos ya existente suele ser poco habitual que se produzcan cambios en el modelo
    - Sin embargo, puede ocurrir que se produzcan cambios en el modelo a medida que se avanza en el trabajo de desarrollo y en la implementación de la aplicación Web
  - La solución a este potencial problema se encuentra en aplicar las denominadas **operaciones de migración**
    - Una migración es un mecanismo de *Entity Framework Core (EF Core)* que se encarga de:
      1. Comparar la estructura del esquema de la base de datos y la especificación del modelo de la aplicación
      2. Y, si procede, generar un script para actualizar la estructura del esquema de la base de datos a partir de la especificación del modelo, salvaguardando el contenido de los datos almacenados

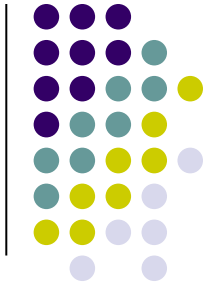


Operaciones de migración

- Las operaciones de migración son una característica de *EF Core* que permiten crear y actualizar el esquema de una base de datos para que corresponda exactamente con la definición de las clases del modelo de la aplicación Web
  - Las operaciones de migración pueden ser de dos tipos:
    - Migración inicial. Produce la creación de la base de datos a partir de la especificación del modelo
    - Migración de datos. Debe realizarse cada vez que se introduzcan cambios en el modelo
  - Ambas se realizan de la misma forma, ejecutando los siguientes comandos:
    1. Se utiliza la Consola del Administrador de paquetes para introducir los comandos de las **operaciones de migración de *EF Core Code First*** sobre la base de datos
    2. En la consola, ejecutar la siguiente instrucción para comparar el esquema de la base de datos con la especificación del modelo y, a continuación, generar un script con las operaciones necesarias para actualizar el esquema de la base de datos que se almacena en la carpeta /Migrations

       PM> **Add-Migration <nombre_migración> -context <nombre_del_contexto>**
    3. En la consola, ejecutar la siguiente instrucción para actualizar la base de datos con la ejecución del script de migración más reciente disponible en la carpeta /Migrations

       PM> **Update-Database -context <nombre_del_contexto>**



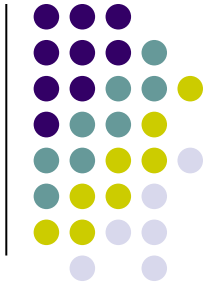
Concepto. Características básicas

- Su función es procesar la interacción del usuario, de manera que se encarga de responder a las acciones del usuario
- Características básicas:
  - Un controlador es una clase derivada de la clase base Controller
  - Las clases que actúan como controlador:
    - Se ubican en la carpeta /Controllers
    - Han de utilizar necesariamente el sufijo *Controller* en su nombre
  - La característica esencial de un controlador consiste en definir uno o más **métodos de acción**, que también suelen denominarse **acciones**
    - Las acciones son métodos públicos que se especifican en un controlador
  - **Una acción de un controlador se ejecuta mediante una solicitud URL**, siguiendo un formato de enrutamiento MVC concreto
  - **Las acciones suelen devolver la ejecución de una vista**, aunque también pueden devolver otros resultados, dependiendo la respuesta que se desee proporcionar

Ejemplo. Controlador *HomeController*

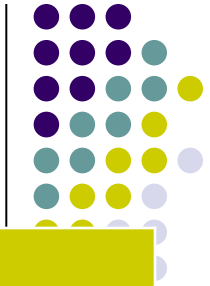
- **El resultado más frecuente de una acción consiste en devolver la ejecución de una vista**
  - En el código, las acciones *Index()* y *Privacy()* del controlador *Home* devuelven una llamada al método *View()*, lo que **hace que se active la vista cuyo nombre coincide** con el nombre de la propia acción
  - Los archivos de vista correspondientes, se encuentran ubicados en la subcarpeta de vistas /Views/Home
  - Esta sencilla regla de convención de nombres constituye el mecanismo por el cual se establece la vista que le corresponde activar a un determinado método de acción

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

    public HomeController(ILogger<HomeController> logger)
    {
        _logger = logger;
    }

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

    public IActionResult Privacy()
    {
        return View();
    }

    [ResponseCache(Duration = 0, Location =
ResponseCacheLocation.None, NoStore = true)]
    public IActionResult Error()
    {
        return View(new ErrorViewModel { RequestId =
Activity.Current?.Id ?? HttpContext.TraceIdentifier });
    }
}
```



El resultado de una acción: la clase IActionResult

- **Los métodos de acción se caracterizan por implementarse como funciones que devuelven un valor de tipo IActionResult**
  - La clase ActionResult representa el resultado de un método de acción
- **El resultado más habitual de una acción consiste en devolver la ejecución de la vista que le corresponde**
  - En estos casos, se emplea el método **View()** de la clase IActionResult para iniciar la ejecución de la vista que activa esa acción, de acuerdo con una convención de nombres
- **Sin embargo, las acciones pueden devolver otros resultados de acción diferentes, dependiendo de la respuesta que se desee proporcionar**
  - En estos casos, pueden utilizarse **otros métodos** de la clase IActionResult para generar resultados de acción diferentes al que proporciona el método View()
  - La clase IActionResult es la base de los resultados de los métodos de acción o acciones de ASP.NET Core MVC

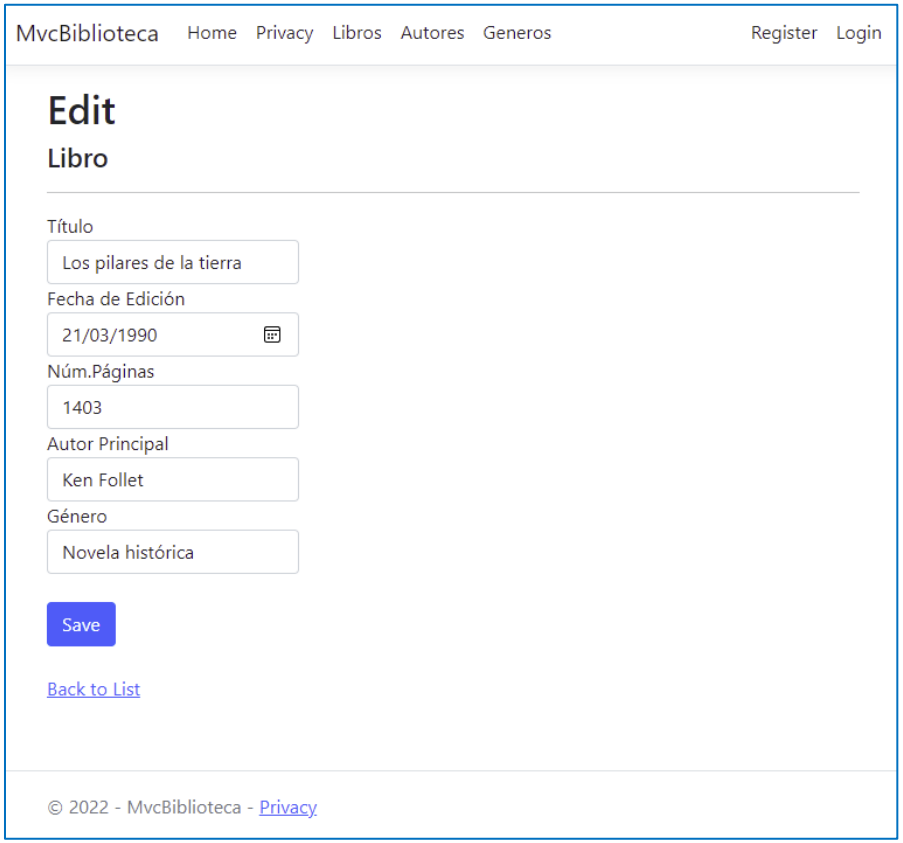


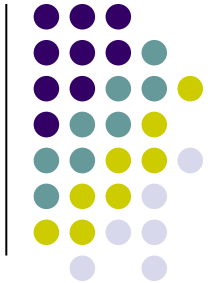
El resultado de la acción: la clase IActionResult

Resultado de la acción	Método	Descripción
ViewResult	**View()**	Representa una vista en la respuesta
RedirectToActionResult	**RedirectToAction()**	Redirecciona a otro método de acción del mismo controlador o de otro controlador
RedirectResult	**Redirect()**	Redirecciona a otro método de acción utilizando su dirección URL
RedirectToRouteResult	**RedirectToRoute()**	Redirecciona a una ruta especificada
ContentResult	**Content()**	Devuelve un tipo de contenido definido por el usuario
ObjectResult	**Object()**	Devuelve un objeto
JsonResult	**Json()**	Devuelve un objeto JSON serializado
StatusCodeResult	**StatusCode()**	Devuelve una respuesta HTTP con el código de estado de respuesta especificado

Acciones GET y POST

- **En los procesamientos que utilizan formularios, suelen definirse dos acciones con el mismo nombre en el mismo controlador**
  - La primera de las acciones se invoca para las **peticiones GET** y se le aplica el atributo **[HttpGet]**, aunque suele quedar implícito al ser este el valor predeterminado
    - Las acciones GET se asocian con la obtención de información desde el servidor hacia el cliente. Por ejemplo, cuando se edita la información sobre un determinado libro
  - La segunda de las acciones se invoca solo para las **peticiones POST** y se le aplica el atributo **[HttpPost]**
    - Las acciones POST se asocian con el envío de información desde el cliente para que sea procesada en el servidor. Por ejemplo, cuando se solicita la actualización de los datos de un libro concreto después de su edición





Acciones GET y POST. Ejemplo acción GET *Edit()*

- **Ejemplo. A continuación se muestran las acciones GET *Edit()* y POST *Edit()* del controlador *Libros* de la aplicación Web *MvcBiblioteca*. La acción GET:**

```
...

// GET: Libros/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null || _context.Libros == null)
    {
        return NotFound();
    }

    var libro = await _context.Libros.FindAsync(id);
    if (libro == null)
    {
        return NotFound();
    }
    ViewData["AutorId"] = new SelectList(_context.Autores, "Id", "Nombre", libro.AutorId);
    ViewData["GeneroId"] = new SelectList(_context.Generos, "Id", "Descripcion", libro.GeneroId);
    return View(libro);
}

...
```

Búsqueda de la entidad *libro* a editar por su *Id*

Activa la vista que gestiona el formulario de edición y le pasa la entidad *libro* para editar

Acciones GET y POST. Ejemplo acción POST *Edit()*

```
...
// POST: Libros/Edit/5
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id,
     [Bind("Id,Titulo,FechaEdicion,NumeroPaginas,AutorId,GeneroId")] Libro libro)
{
    if (id != libro.Id) {
        return NotFound();
    }
    if (ModelState.IsValid) {
        try {
            _context.Update(libro);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException) {
            if (!LibroExists(libro.Id)) {
               return NotFound();
            }
            else {
               throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    ViewData["AutorId"] = new SelectList(_context.Autores, "Id", "Nombre", libro.AutorId);
    ViewData["GeneroId"] = new SelectList(_context.Generos, "Id", "Descripcion", libro.GeneroId);
    return View(libro);
 }
...
```

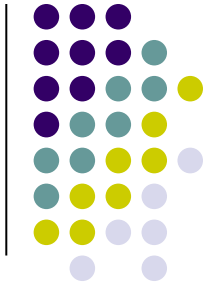
Acción POST

Compone la entidad *libro* con los valores recibidos a través del formulario

Modifica la entidad libro en el modelo

Produce reflejo de los cambios realizados sobre la Base de datos

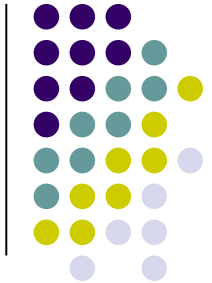
Redirecciona a la acción Index()



Filtros de acción

- **Un filtro de acción es un atributo que puede asociarse a una acción de un controlador, o al controlador en su totalidad, para modificar la forma en la que se ejecutan las acciones**
  - Por ejemplo, los atributos [HttpPost] y [HttpGet], estudiados en punto anterior, son filtros de acción

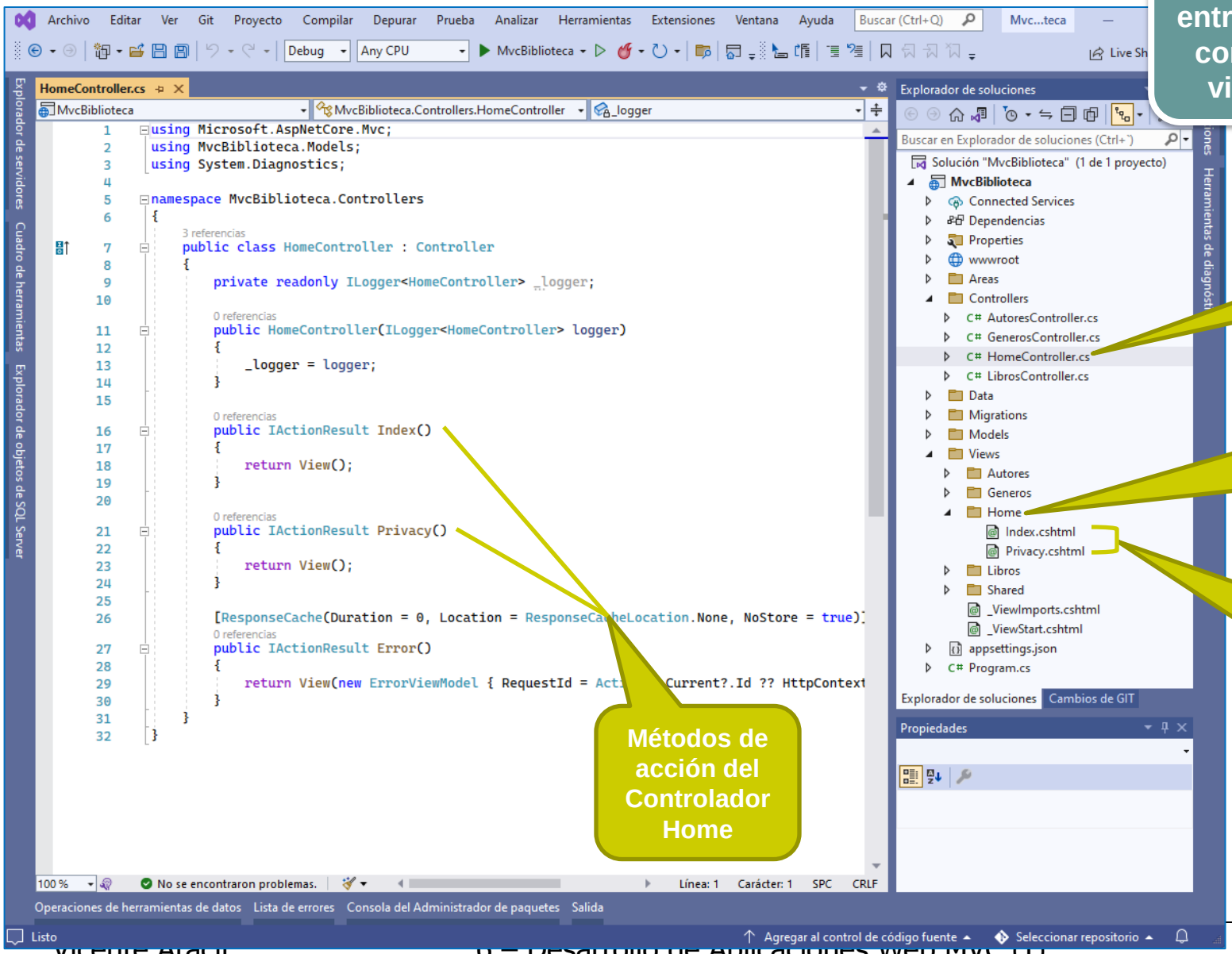
Atributos	Descripción
Authorize	Restringe una acción para los roles o usuarios especificados
HandleError	Permite especificar una vista que se mostrará en caso de excepción no controlada
HttpGet	Filtra la invocación del método de acción solo para la peticiones GET
HttpPost	Filtra la invocación del método de acción solo para la peticiones POST
OutputCache	Almacena en caché la salida de un controlador
ValidateAntiForgeryToken	Facilita un mecanismo de seguridad de accesos indebidos. Ayuda a evitar peticiones de falsificación de solicitud (pishing)
ValidateInput	Desactiva las validaciones de solicitud. Se recomienda no hacer uso de este filtro, ya que puede poner en peligro la seguridad de la aplicación

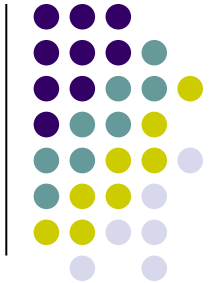


Concepto. Características básicas

- **La ejecución de una vista genera dinámicamente en el servidor una página de respuesta HTML que se envía al cliente**
  - Los archivos de vista permiten encapsular el procesamiento que genera las respuestas HTML hacia un cliente
- **Características básicas de las vistas:**
  - Los nombres de los archivos de vista tienen una extensión **.cshtml**
  - Para crear los archivos de vista se suele emplear el **lenguaje de vistas Razor** que facilita la creación de documentos HTML utilizando el lenguaje de programación C#
  - En las aplicaciones Web basadas en ASP.NET Core MVC el trabajo de desarrollo de las vistas consiste, básicamente, en escribir el código
    - No existe un diseñador visual ni controles de interfaz que se agreguen mediante arrastrar y soltar
  - El proceso de creación de vistas resulta sencillo. Visual Studio proporciona un asistente y ayudas para facilitar la creación de vistas al desarrollador
  - Se emplea una **convención de nombres** sencilla para establecer la correspondencia entre una acción y la vista a ejecutar

Convención de nombres para las vistas





Plantilla de diseño del Proyecto de ASP.NET Core MVC

- **El archivo *_ViewStart.cshtml***
  - Establece el **nombre de la plantilla de diseño** común para todas las vistas de la aplicación Web de ASP.NET Core MVC. Está alojado en la carpeta **/Views**. Su contenido es:

```
@{
    Layout = "_Layout.cshtml";
}
```

    - La marca @ pertenece a la sintaxis del lenguaje o motor de vistas Razor. Especifica el inicio de una sección de código Razor que queda delimitado entre llaves
    - La línea de código asigna a la propiedad *Layout* el nombre del archivo de vista que define la plantilla de diseño común del Proyecto de ASP.NET Core MVC
- **El archivo *_Layout.cshtml***
  - Es un archivo de vista que constituye la **plantilla de diseño predeterminada** que define el código común que ejecutarán todas las vistas. Se aloja en la carpeta **/Views/Shared**
    - Al establecerse una plantilla de diseño común, cada archivo de vista contendrá únicamente el código que define el contenido específico de esa vista
    - La carpeta /Views/Shared aloja archivos de código compartido para las vistas, como: plantillas de diseño, vistas parciales reutilizadas por otras vistas y vistas compartidas por varios controladores

Ejemplo de plantilla de diseño: *_Layout.cshtml*

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - MvcBiblioteca</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true" />
    <link rel="stylesheet" href="~/MvcBiblioteca.styles.css" asp-append-version="true" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light bg-white
                    border-bottom box-shadow mb-3">
            <div class="container-fluid">
                <a class="navbar-brand" asp-area="" asp-controller="Home"
                   asp-action="Index">MvcBiblioteca</a>
                <button class="navbar-toggler" type="button" data-bs-toggle="collapse"
                        data-bs-target=".navbar-collapse" aria-controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-controller="Home"
                               asp-action="Index">Home</a>
                        </li>
. . .
```

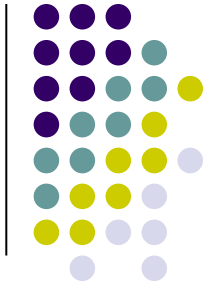
Ejemplo de plantilla de diseño: *_Layout.cshtml*

```
. . .
                    <li class="nav-item">
                        <a class="nav-link text-dark" asp-area="" asp-controller="Home"
                           asp-action="Privacy">Privacy</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link text-dark" asp-area="" asp-controller="Libros"
                           asp-action="Index">Libros</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link text-dark" asp-area="" asp-controller="Autores"
                           asp-action="Index">Autores</a>
                    </li>
                    <li class="nav-item">
                        <a class="nav-link text-dark" asp-area="" asp-controller="Generos"
                           asp-action="Index">Generos</a>
                    </li>
                </ul>
                <partial name="_LoginPartial" />
            </div>
        </div>
    </nav>
</header>
<div class="container">
    <main role="main" class="pb-3">
        @RenderBody()
    </main>
</div>
. . .
```

Ejemplo de plantilla de diseño: *_Layout.cshtml*

```
. . .
    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2022 - MvcBiblioteca –
            <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
        </div>
    </footer>
    <script src="~/lib/jquery/dist/jquery.min.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

- La instrucción ***@RenderBody()*** define la parte de contenido reemplazable
  - En una página de diseño, esta instrucción es un marcador de posición que se reemplazará por el contenido específico definido en cada vista que utilice la plantilla de diseño *_Layout.cshtml*
- Se pueden crear varios archivos de plantilla de diseño diferentes en un mismo Proyecto de ASP.NET Core MVC
  - En estos casos, se asigna la ruta y el nombre de una de las plantillas de diseño existentes en el proyecto, al valor de la propiedad *Layout* en cada vista de manera específica, añadiendo el código correspondiente al inicio de cada archivo de vista



Generar los enlaces para la solicitud URL de las acciones

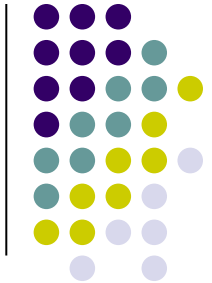
- **Se utiliza un elemento enlace, mediante una etiqueta <a>, para generar sobre la página de respuesta, el enlace HTML de la solicitud URL hacia la acción deseada**
  - Para solicitar una acción determinada desde una vista, es necesario generar el enlace correspondiente, considerando el formato del enrutamiento MVC establecido. Por ejemplo:

```
<a asp-controller="Libros" asp-action="Edit" asp-route-id="@item.Id">Editar</a>
```

    - El **controlador, la acción y el valor del parámetro** a solicitar se especifican mediante el uso de los atributos de Razor: ***asp-controller***, ***asp-action*** y ***asp-route-id***, respectivamente
      - Considerando que el valor de la propiedad *Id* de la entidad *item* sea 3, se generará en la página de respuesta un enlace hacia la solicitud http://localhost:xxxx/Libros/Edit/3 con el texto "Editar" y cuyo código HTML será:

```
<a href="/Libros/Edit/3">Editar</a>
```

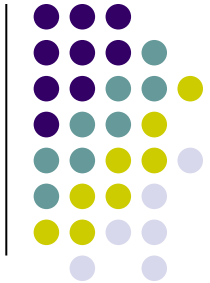
      - Al hacer clic sobre este enlace en la página de respuesta, se ejecutará la acción GET ***Edit(int? id)*** del controlador ***Libros*** y se asignará el valor 3 al parámetro ***id*** de esta acción
    - Si se solicita una acción del mismo controlador, no es necesario especificar ***asp-controller***. Si se solicita una acción que no tiene parámetro, no es necesario especificar el argumento ***asp-route-id***
    - El elemento <a> se utiliza como **asistente de etiquetas** *(Tag helper)*. Los asistentes de etiquetas permiten asociar el código de Razor que se desea ejecutar en las etiquetas de HTML de las vistas



El lenguaje o motor de vistas Razor

- **El lenguaje de vistas Razor permite integrar el lenguaje C# en las vistas**
  - La especificación del código de una vista es una mezcla de sintaxis de C# y código de marcado HTML
  - Los nombres de los archivos de vista tienen la extensión: **.cshtml**
  - El motor de Razor se encargará de la ejecución de los bloques de código C#
    - Mientras que las etiquetas HTML se añadirán directamente al flujo de salida de la página de respuesta hacia el cliente, para que sean interpretadas por el navegador
  - El carácter **@** sirve de marcador *(tag)* al motor de vistas para distinguir las **secuencias de código de Razor** de las etiquetas de HTML estático
    - Si se necesitara escribir un carácter @ sobre la página, por ejemplo, para presentar en la página una dirección de correo electrónico, puede doblarse (@@) para que el motor de Razor inhiba su comportamiento predefinido
  - Para trabajar con las vistas, además de la sintaxis de Razor, es necesario comprender el uso de los denominados **Asistentes** o ***Helpers***
    - Facilitan el trabajo de desarrollo de las vistas, porque permiten que el código de Razor, que se ejecuta en el lado servidor, participe en la creación y representación de etiquetas a la hora de generar los elementos de HTML sobre las páginas de respuesta que se envían al cliente

El lenguaje o motor de vistas Razor



- **Los Asistentes o *Helpers* se usan en las vistas para poder generar etiquetas o código HTML sobre la interfaz de la página de respuesta**
  - Pueden distinguirse dos tipos de Asistentes:
    - **Asistentes de etiquetas o *Tag helpers***. Se usan en una vista para representar etiquetas de HTML sobre la página de respuesta. Más utilizados actualmente. Por ejemplo, para representar un cuadro de texto que permita editar el valor del título del libro sobre la página de respuesta, se haría:

```
<input asp-for="Titulo" class="form-control" />
```

    - **Asistentes de HTML o *HTML helpers***. Se usan en una vista para representar contenido HTML sobre a página de respuesta. En la mayoría de los casos, solo es un método que devuelve una cadena HTML. Siguiendo con el ejemplo, para generar el mismo código HTML resultante, sería:

```
@Html.TextBoxFor(m => m.Titulo, new { @class = "form-control" })
```

    En el primer caso, la definición de la propiedad del modelo que se está editando se hace mediante el atributo **asp-for**, mientras que, en el segundo caso, se usa una función anónima con operador Lambda
  - Más información sobre la sintaxis de Razor para ASP.NET Core:
    - https://learn.microsoft.com/es-es/aspnet/core/mvc/views/razor?view=aspnetcore-8.0