Tema 6: El Controlador

Asignatura: Desarrollo web en entorno servidor

CS Desarrollo de Aplicaciones Web

Introducción

- En este capítulo veremos aspectos como:
  - Características del controlador
  - Configuración de rutas
  - Comunicación del controlador con la capa cliente
  - Variables de sesión, aplicación y caché

Introducción

- El Controlador gestiona las relaciones entre la Vista y el Modelo:
  - **Recibe los datos** del usuario desde distintas fuentes, como la query string, campos de formulario, o parámetros de ruta,
  - **Realiza las conversiones** de tipo necesarias para poder procesar la peticíón de forma correcta
  - **Invoca al modelo para enviar u obtener información** necesaria para realizar la acción solicitada por el usuario,
  - **Decide qué vista enviar al cliente** en función de la acción que ha realizado,
  - **Envia a la vista** la información que necesita para componerse apropiadamente.

FORMACIÓN PROFESIONAL MONTECASTELO

CONFIGURACIÓN DE RUTAS

Sistema de routing

- El sistema de routing actúa como front-controller, al que **llegan las peticiones para**, en función de la información contenida en la tabla de rutas, **delegárselas a controladores específicos** de nuestra aplicación.
- Por lo tanto, tiene vital importancia la correcta definición de la tabla de rutas. Además, la misma información contenida en esta tabla se utiliza cuando deseamos generar hipervínculos o URLs de acceso a acciones definidas en nuestros controladores.

Sistema de routing

Definición de rutas

- Las rutas se registran durante el arranque de la aplicación, en el fichero **Program.cs**

- Las rutas que van a ser procesadas mediante acciones de un controlador se definen utilizando el método **MapControllerRoute**()

Sistema de routing

Definición de rutas

- Existen diversas sobrecargas de **MapControllerRoute**() mediante las que es posible suministrar la siguiente información:
  - nombre de la ruta,
  - patrón de URL al que se aplica la regla,
  - valores por defecto para los parámetros de ruta,
  - restricciones,
  - y por último, espacios de nombre donde localizar los controladores.

Sistema de routing

Definición de rutas

- **Nombre de la ruta:**
  - Debe ser un nombre único para la regla en la tabla de rutas, pues se utilizará como referencia para generar direcciones URL.
  - Por ejemplo, generando un enlace en una Vista con el helper RouteLink, hacia la ruta definida.

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

```
@Html.RouteLink(
    "Netbooks catalog", //Texto del enlace
    "ProductsByCategory", //Nombre de la ruta
    new
    {
        category = "netbooks" //Parámetros
    }
)
```

Sistema de routing

Definición de rutas

- **Patrón de URL:**
  - **Define el formato de la URL a la que se aplica la regla**, ya sea para enviar una petición a una acción (Primer ejemplo de la página anterior) o para direccionar dentro del sistema (Segundo ejemplo)
  - El patrón de URL puede contener partes fijas, que obligatoriamente deben formar parte de la URL de la petición, y, opcionalmente, partes variables o parámetros de ruta, cuyos valores serán obtenidos por el sistema de routing y pasados al controlador y acción encargados de procesar las peticiones.

Sistema de routing

Definición de rutas

- **Patrón de URL:**
  - Los parámetros deben encontrarse definidos sobre la URL de la forma {nombre_parámetro}, por ejemplo así:

```
catalog/{family}/{subfamily}/{id}
```

  - Por ejemplo, una petición entrante hacia la dirección **catalog/computers/netbooks/sony-vaio** generaría tres pares clave/valor como parámetros de petición:
    - family="computers“
    - subfamily="netbooks“
    - id="sony-vaio".

Sistema de routing

Definición de rutas

- **Valores por defecto para los parámetros:**
  - Para que la dirección de una petición coincida con el patrón de URL establecido en una regla no es necesario que aparezcan valores para todos los parámetros.
  - Por ejemplo, en este caso los parámetros **controller** y **action** tienen definido un valor por defecto, e **id** tiene carácter opcional.

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

Sistema de routing

Definición de rutas

- **Restricciones:**
  - Permiten añadir cierta lógica al proceso de selección de rutas, permitiendo descartar una regla de la tabla de rutas si no cumple determinados criterios.
  - Existen dos formas de establecer restricciones en las rutas:
    - **Definir expresiones regulares a comprobar sobre el valor de los parámetros**
    - **Crear una clase que implemente el interfaz IRouteConstraint, donde implementaremos el código de una restricción personalizada**

Sistema de routing

Definición de rutas

- Por ejemplo, si el parámetro "id" no contiene entre uno y cinco dígitos, la regla será descartada, el sistema de routing pasará a evaluar la siguiente entrada de la tabla de rutas.

```
app.MapControllerRoute(
    name: "Product",
    pattern: "Products/{id}",
    defaults: new
    {
        controller = "Products",
        action = "Details"
    },
    constraints: new { id = @"\d{1,5}" }
    );
```

Sistema de routing

Definición de rutas

- Por ejemplo, creamos una clase que implemente el interfaz **IRouteConstraint**, donde implementaremos una restricción que impide que una ruta sea seleccionada si el usuario no se ha autenticado previamente:

```
public class AutenticacionUsuarioRestriccionPersonalizada : IRouteConstraint
{
    public bool Match(HttpContext? httpContext, IRouter? route, string routeKey, RouteValueDictionary values, RouteDirection routeDirection)
    {
        return httpContext.User.Identity.IsAuthenticated;
    }
}
```

Sistema de routing

Definición de rutas

- Para utilizar esta restricción, se la asignaremos a alguna propiedad del objeto anónimo.

```
app.MapControllerRoute(
    name: "Default",
    pattern: "admin/{controller}/{action}/{id?}",
    defaults: new
    {
        controller = "HomeAdmin",
        action = "Index"
    },
    constraints: new { auth = new AutenticacionUsuarioRestriccionPersonalizada() }
    );
```

Sistema de routing

Definición de rutas

- **Espacios de nombre donde localizar los controladores:**
  - A veces, para evitar conflictos entre controladores con el mismo nombre, es interesante indicar al framework qué espacios de nombres debe considerar a la hora de localizar la clase que procesará una petición:

```
app.MapControllerRoute(
    name: "Default",
    pattern: "admin/{controller}/{action}/{id?}",
    defaults: new
    {
        controller = "HomeAdmin",
        action = "Index"
    },
    new[] { "MyAplication.Controllers" }
    );
```

Sistema de routing

MUY IMPORTANTE: ¡EL ORDEN IMPORTA!

- **Cuando llega una petición, el sistema de routing examina la tabla de rutas en el mismo orden en que han sido registradas las distintas reglas, y la primera cuyo patrón de URL encaje con la misma y supere las restricciones, será utilizada para procesarla.**

FORMACIÓN PROFESIONAL MONTECASTELO

Actividad 1

*Practicaremos el **sistema de routing** con estas actividades. Cada una debe tener la acción Index.*

- *Controlador "**ProductoController**" con las acciones "**Listar**", "**Detalle**" y "**Buscar**".*
  - *La acción "Listar" debería mostrar una lista de productos, la acción "Detalle" debería mostrar los detalles de un producto específico y la acción "Buscar" debería buscar productos por nombre.*
- *Controlador "**BlogController**" con las acciones "**Listar**", "**Detalle**", "**Crear**", "**Editar**" y "**Eliminar**".*
  - *La acción "Listar" debería mostrar una lista de publicaciones de blog, la acción "Detalle" debería mostrar los detalles de una publicación de blog específica, la acción "Crear" debería permitir crear nuevas publicaciones de blog, la acción "Editar" debería permitir editar publicaciones existentes y la acción "Eliminar" debería permitir eliminar publicaciones de blog.*
- *Controlador llamado "**TiendaController**" con las acciones "**Listar**", "**Detalle**", "**AgregarAlCarrito**" y "**RealizarPedido**".*
  - *La acción "Listar" debería mostrar una lista de productos en la tienda, la acción "Detalle" debería mostrar los detalles de un producto específico, la acción "AgregarAlCarrito" debería permitir agregar productos al carrito de compras y la acción "RealizarPedido" debería permitir al usuario realizar un pedido.*
- *Controlador llamado "**RecursoController**" con las acciones "**Listar**", "**Detalle**" y "**Descargar**".*
  - *La acción "Listar" debería mostrar una lista de recursos, la acción "Detalle" debería mostrar los detalles de un recurso específico y la acción "Descargar" debería permitir al usuario descargar el recurso.*
- *Controlador llamado "**UsuarioController**" con las acciones "**Registro**", "**InicioSesion**" y "**Perfil**".*
  - *La acción "Registro" debería permitir al usuario registrarse, la acción "InicioSesion" debería permitir al usuario iniciar sesión y la acción "Perfil" debería mostrar la información del usuario y permitir al usuario editar su perfil.*

15 MINUTOS

FORMACIÓN PROFESIONAL MONTECASTELO

RUTADO POR ATRIBUTOS (ATTRIBUTE ROUTING)

Rutado por atributos

Definición de rutas

- El rutado por atributos nos permite definir la ruta en el controlador, en lugar de en el Program.cs, lo cual facilita su legibilidad y mantenibilidad.
- **Las rutas las definiremos utilizando el atributo Route** sobre la acción a la que queremos facilitar el acceso.

```
[Route("product/{productId}/{productTitle}")]
public ActionResult Show(int productId, string productTitle)
{
    return View();
}
```

Rutado por atributos

Definición de rutas

- **El nombre de los parámetros de la ruta y del método deben ser iguales**, para que el binder pueda detectar esta coincidencia y mapearlos sin problema.
- Además, **podemos asignar un nombre a la regla**, de forma que podremos referirnos a ella más adelante para generar URLs o enlaces hacia una acción.

```
[Route("product/{productId}/{productTitle}", Name = "CategoryProduct")]
public ActionResult Show(int productId, string productTitle)
{
    return View();
}
```

Rutado por atributos

Definición de rutas

- **Todos los parámetros de ruta son inicialmente obligatorios.**
- Si queremos indicar su opcionalidad debemos añadir al nombre del parámetro un carácter de interrogación "?", como en el siguiente ejemplo:

```
[Route("calculator/sum/{a}/{b?}")]
public ActionResult Sum(int a, int? b)
{
    return View();
}
```

Rutado por atributos

Definición de rutas

- Podemos **indicar valores por defecto para los parámetros** de ruta introduciendo una igualdad tras el nombre del parámetro:

```
[Route("calculator/sum/{a=5}/{b=10}")]
public ActionResult Sum(int a, int? b)
{
    return View();
}
```

Rutado por atributos

Especificación de restricciones

- Con attribute routing, la introducción de restricciones en las rutas es mucho más sencilla con el enfoque tradicional de definición de rutas.
- Por ejemplo, la siguiente ruta sólo se aplicará cuando el parámetro productId en la URL de la petición sea un entero:

```
[Route("product/edit/{id:int}")]
public ActionResult GetProductById(int id)
{
    return View();
}
```

Rutado por atributos

Especificación de restricciones

- Si el mismo parámetro está sometido a más de una constraint, se pueden concatenar utilizando esta misma sintaxis.
- Por ejemplo, el pin debe ser de exactamente 4 caracteres alfabéticos:

```
[Route("user/confirm/{pin:alpha:length(4}")]
public ActionResult Confirm(string pin)
{
    return View();
}
```

Rutado por atributos

Especificación de restricciones

- Las restricciones que podemos aplicar de serie son las siguientes:
  - **alpha**: Secuencia de caracteres alfabéticos latinos (a-z, A-Z)
  - **bool**: Valor booleanos (true o false)
  - **datetime**: Valor DateTime (en formato aaaa-mm-dd)
  - **decimal:** Valor decimal (usando el punto como separador
  - **double:** Valor en punto flotante de 64-bits
  - **float:** Valor en punto flotante de 32-bits

Rutado por atributos

Especificación de restricciones

- Las restricciones que podemos aplicar de serie son las siguientes:
  - **guid:** GUID
  - **Int:** Valor entero de 32 bits
  - **Length:** Cadena de caracteres con una longitud específica, o en un rango
  - **Long:** Valor entero de 64 bits
  - **Max :** Entero, como máximo el valor indicado

Rutado por atributos

Especificación de restricciones

- Las restricciones que podemos aplicar de serie son las siguientes:
  - **maxlength:** Cadena de caracteres con un tamaño máximo
  - **min:** Entero, como mínimo el valor indicado
  - **minlength:** Cadena de caracteres con un tamaño mínimo
  - **range:** Entero que se encuentre en un rango de valores
  - **regex:** Valor que cumpla la expresión regular indicada

Rutado por atributos

Especificación de restricciones

- Si un parámetro es opcional o tiene un valor por defecto además de restricciones, hay que indicarlo tras las constraints, como en el siguiente ejemplo:

```
[Route("test/{data:alpha:length(3)=car}")]
public ActionResult Test(string data)
{
    return View();
}
```

FORMACIÓN PROFESIONAL MONTECASTELO

Actividad 2

*Practicaremos el **attribute routing** con estas actividades.*

- *Crea una ruta que apunte a una acción "**Index**" en un controlador llamado "**TiendaController**", que tenga una restricción en el parámetro "id" que solo permita valores numéricos enteros positivos.*
- *Crea una ruta que apunte a una acción "**Detalles**" en un controlador llamado "**ProductoController**", que acepte dos parámetros: "id" y "slug". El parámetro "id" debe ser un número entero de cuatro dígitos, mientras que el parámetro "slug" debe ser una cadena alfanumérica. Los valores de los parámetros deben estar separados por un guión.*
- *Crea una ruta que apunte a una acción "**Listado**" en un controlador llamado "**ProductoController**", que tenga un prefijo en la URL de "productos" y acepte dos parámetros opcionales: "categoria" y "marca".*
- *Crea una ruta que apunte a una acción "**Buscar**" en un controlador llamado "**ProductoController**", que tenga un sufijo en la URL de "buscar" y acepte un parámetro "q" que represente la consulta de búsqueda. El parámetro "q" debe estar incluido en la URL y no en la cadena de consulta.*
- *Crea una ruta que apunte a una acción "**Perfil**" en un controlador llamado "**UsuarioController**", que tenga un prefijo en la URL de "usuario" y acepte un parámetro "nombre" que represente el nombre de usuario. El nombre de usuario debe tener una longitud mínima de 3 caracteres y una longitud máxima de 20 caracteres.*

15 MINUTOS

Práctica guiada

Práctica guiada 4: Sistema de routing

FORMACIÓN PROFESIONAL MONTECASTELO

CONTROLADORES

Controladores

El controlador

- Las clases controlador, o simplemente controladores, son las **clases en las que el sistema de routing delega el proceso de las peticiones entrantes.**
- Según la convención de ubicación de archivos, debemos situarlas en la carpeta **/Controllers** del proyecto ASP.NET MVC.
- Además, todas las clases deben nombrarse de la forma "NombreController“. Por ejemplo, al controlador “Blog”, lo implementaremos en “BlogController”

Controladores

El controlador

- Cuando **el controlador recibe el control** desde el sistema de routing, **localiza la clase**, basándose en el valor del parámetro de ruta "Controller" **y la instancia.**
- Para ello, recorre el ensamblado en busca de clases que cumplan las siguientes condiciones:
  - **debe ser pública,**
  - **su nombre debe finalizar con "Controller"**,
  - **no puede ser abstracta**
  - **debe implementar el interfaz IController.**

Controladores

El controlador

- Por tanto, el formato habitual para nuestros controladores será:

```
public class FriendController : Controller
{
        public ActionResult Action1() { ... }
        public ActionResult Action2() { ... }
}
```

Controladores

- En resumen, **un controlador es un contenedor en los que implementaremos las acciones.**

Controladores

Acciones

- Las acciones son **los métodos que implementan el proceso de peticiones concretas**, es decir, **donde se encuentra el comportamiento del controlador**.
- Una vez localizado el controlador, éste busca el "Action" y método asociado que cumpla los siguientes requisitos:
  - **debe ser público,**
  - **de instancia, es decir, no estático,**
  - **no contener parámetros genéricos,**
  - **y debe poder ser invocable directamente.**

Controladores

Acciones

- Por tanto, el formato habitual para nuestros controladores será:

```
public ActionResult MyAction(type1 param1, type2 param2, ...)
{
    // Obtiene y transforma parámetros de entrada
    // Valida los parámetros
    // Utiliza el Modelo para enviar o recibir información
    // Selecciona una vista
    // Envía a la vista la información que necesita
    // Devuelve la vista
}
```

Controladores

Acciones

- Las acciones deben cumplir una serie de buenas prácticas:
  - Deben tener un **propósito claro,** debe quedar claro qué es lo que hacen
  - Deben seguir un **principio de responsabilidad única** (SRP), deben encargarse únicamente de una acción
  - Deben cuidar la **extensión de las acciones,** no debe ocupar más de doce líneas

Controladores

Atributos de selección de acciones

- Veremos una serie de atributos que utiliza ASP.NET Core MVC a la hora de determinar qué acción procesará una petición determinada:
  - **HttpDelete, HttpGet, HttpPost, HttpPut**
  - **NonAction**
  - **ActionName**

Controladores

HttpDelete, HttpGet, HttpPost, HttpPut

- Permite indicar que el **método de acción** al que se aplica **deberá ser seleccionado sólo si el método HTTP utilizado en la petición actual coincide** con el indicado
- Esto permite utilizar el mismo nombre de acción para métodos distintos, siendo ejecutado el correcto en función del tipo de petición.

Controladores

HttpDelete, HttpGet, HttpPost, HttpPut

- Los posibles métodos utilizados serían:
  - DELETE
  - POST
  - GET
  - PUT

Controladores

HttpDelete, HttpGet, HttpPost, HttpPut

```
// POST /Friends/Create
[HttpPost]
public IActionResult Create(Friend friend)
{
    // Validate and update the model
    return View();
}

// DELETE /Friends/Delete/665
[HttpDelete]
public IActionResult Delete(int id)
{
    // Delete the friend
    return View();
}
```

Controladores

HttpDelete, HttpGet, HttpPost, HttpPut

- **OJO:** Si no se indica nada, no se tendrá en cuenta el método HTTP a la hora de determinar si una acción va a ser la encargada de procesar una petición.

Controladores

NonAction

- Este atributo hace que MVC ignore el método sobre el que se aplica.
- Es decir, que este método no podrá ser llamado utilizando el sistema de routing.

```
// GET /Controller/GetSomething
[NonAction]
public string GetSomething()
{
    return string.Empty;
}
```

Controladores

ActionName

- Permite asignar un nombre de acción a un método distinto al propio
- Por ejemplo, el método Remove() responde a las peticiones hacia la acción "Delete":

```
[ActionName("Delete")]
public IActionResult Remove(int id)
{
    return View(); //Se devuelve la vista Delete.cshtml
}
```

FORMACIÓN PROFESIONAL MONTECASTELO

Actividad 3

*Practicaremos el uso de **atributos de selección de acciones** con estas actividades.*

- *Crea una ruta que apunte a una acción "**Buscar**" en un controlador llamado "**ProductoController**" que acepte un parámetro "terminoBusqueda". Si el parámetro "terminoBusqueda" no se proporciona en la URL, la acción debería redirigir a la acción "Index".*

- *Crea una ruta que apunte a una acción "**Agregar**" en un controlador llamado "**ProductoController**". Esta acción debería aceptar un modelo de producto como parámetro. Si el modelo de producto es nulo, la acción debería mostrar un formulario para agregar un nuevo producto. Si el modelo de producto no es nulo, la acción debería agregar el producto a la base de datos y redirigir a la acción "Index".*

- *Crea una ruta que apunte a una acción "**Editar**" en un controlador llamado "**ProductoController**". Esta acción debería aceptar un parámetro "id" y un modelo de producto. Si el parámetro "id" es nulo, la acción debería redirigir a la acción "Index". Si el modelo de producto es nulo, la acción debería mostrar un formulario para editar el producto. Si el modelo de producto no es nulo, la acción debería actualizar el producto en la base de datos y redirigir a la acción "Index".*

- *Crea una ruta que apunte a una acción "**Eliminar**" en un controlador llamado "**ProductoController**". Esta acción debería aceptar un parámetro "id". Si el parámetro "id" es nulo, la acción debería redirigir a la acción "Index". Si el parámetro "id" no es nulo, la acción debería eliminar el producto correspondiente de la base de datos y redirigir a la acción "Index".*

10 MINUTOS

Controladores

Acciones con parámetros simples

- Hasta ahora, en los ejemplos que hemos visto, han aparecido varias veces acciones con parámetros cuyos valores eran tomados del contexto de petición, y concretamente, desde los parámetros incluidos en la URL de la petición e identificados en el patrón de ruta

Controladores

Acciones con parámetros simples

- Por ejemplo:

```
// Default route: {controller}/{action}/{id}

// GET /product/edit/884
public IActionResult Edit(int id)
{
    // "id" contains 884
    return View();
}
```

Controladores

Acciones con parámetros simples

- El mecanismo de **binding de ASP.NET MVC permite que,** una vez ha seleccionado el método a ejecutar, analiza sus parámetros formales y obtiene utiliza sus nombres para obtener un valor para cada uno ellos en:
  - valores de campos de formulario,
  - parámetros de ruta,
  - parámetros en la query string,
  - archivos adjuntos a la petición.

Controladores

Acciones con parámetros simples

- De esta forma, **para un parámetro llamado "id"**, primero **se buscará** si existe **en la petición** el valor de **algún campo de formulario con dicho nombre**; en caso negativo, se intentará localizar **un parámetro de ruta denominado "id"** que tenga algún valor, **y así sucesivamente**.

Controladores

Acciones con parámetros simples

- Por ejemplo, podríamos implementar el método View tal que así:

```
public ActionResult View(string id, bool fullDetail, int currentPage)
{
    return Content("id=" + id +
    ", fullDetail=" + fullDetail + ", currentPage=" + currentPage);
}
```

- Funcionaría ante una petición como:

/Product/View/?id=999&fullDetail=true&currentPage=2

Controladores

Acciones con parámetros simples

- En el paso de parámetros, podemos:
  - Declarar los parámetros como anulables

```
public IActionResult View(string id, bool? fullDetail, int? currentPage)
```

  - Crear parámetros opcionales, a los que se les asigna un valor por defecto

```
public IActionResult View(string id, bool fullDetail = false, int currentPage = 1)
```

  - Especificar valores por defecto mediante el atributo DefaultParameterValue

```
public IActionResult View(string id,
                          [DefaultParameterValue(false)] bool fullDetail,
                          [DefaultParameterValue(1)] int currentPage)
```

Controladores

Upload de ficheros

- El envío de archivos al servidor es una tarea realizada muy frecuentemente en aplicaciones web, y ASP.NET MVC ofrece mecanismos específicos para facilitar la obtención de esta información desde el lado servidor.

Controladores

Upload de ficheros

- Por ejemplo, utilizando esta vista damos la opción de subir ficheros:

```
<form action="/home/upload" method="post" enctype="multipart/form-data">
    <input type="file" name="myFile" />
    <input type="submit" value="Upload" />
</form>
```

- **Importante:** Los formularios para subir ficheros **deben establecer el atributo enctype a multipart/form-data.**

Controladores

Upload de ficheros

- Para obtener los archivos enviados desde el controlador debemos incluir un parámetro de tipo **IFormFile** con el nombre del campo utilizado en el control **<input type="file">** del formulario:

```
public ActionResult Upload(IFormFile myFile)
{
    string path = "C:/"; // Ruta donde queremos guardar el fichero
    return Content("Ok, file uploaded");
}
```

Controladores

Acciones con parámetros complejos

- Desde el binding, **mecanismo encargado de buscar valores en los parámetros**, podemos trabajar con tipos complejos como podría ser una entidad de datos.
- Por ejemplo:

```
public IActionResult Update(Friend friend)
{
    return Content(friend.Name + ", " +
    friend.Age + ", " + "Tlf: " + friend.Description);
}
```

Controladores

Acciones con parámetros complejos

- En este último ejemplo, el binder realiza las siguientes acciones:
  - **Analiza el tipo del parámetro** (en este caso, Friend), **recorriendo sus propiedades y buscando valores para ellas** en el contexto de la petición.
  - **Si encuentra valores para alguna propiedad, instanciará un objeto** y lo poblará con dicha información
  - **Si no los encuentra, enviará null al método de acción.**

Controladores

Acciones con parámetros complejos

- Por ejemplo, la siguiente petición llegaría correctamente a nuestra acción:

/Friends/Update/?name=John&Age=25&Description=Bored

Controladores

- También se aceptaría la petición si se tratase de un formulario de actualización de datos como el siguiente:

```
<form method="post" action="/friends/update">
    <label for="name">Name:</label>
    <input type="text" name="name" />
    <br />

    <label for="age">Age:</label>
    <input type="text" name="age" />
    <br />

    <label for="description">Description:</label>
    <input type="text" name="description" />
    <br />

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

Controladores

Propiedades complejas

- Pueden darse casos en que la clase que se está poblando contenga a su vez propiedades de tipo complejo, con lo que se vuelve a realizar sobre ella la misma operación, es decir, se instancia y se intentan poblar sus propiedades
- Vemos a continuación un ejemplo completo:

Controladores

Propiedades complejas

- Tenemos las siguientes clases Student y Address:

```
public class Student
{
    public string Name { get; set; }
    public int Age { get; set; }
    public int Phone { get; set; }
    public Address Address { get; set; }
}

public class Address
{
    public string Street { get; set; }
    public string ZipCode { get; set; }
}
```

Controladores

Propiedades complejas

- Trabajamos con la siguiente vista:

```
<form method="post" action="/friends/update">
    <label for="name">Name:</label>
    <input type="text" name="name" />
    <br />

    <label for="age">Age:</label>
    <input type="text" name="age" />
    <br />

    <label for="phone">Phone:</label>
    <input type="text" name="phone" />

    <!-- Address editor -->
    <hr />
    <label for="adress.street">Street:</label>
    <input type="text" name="address.street" />
    <br />

    <label for="adress.zipcode">Zip code: </label>
    <input type="text" name="address.zipcode" />
    <br />

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

Controladores

Propiedades complejas

- **El envío de este formulario** por parte del usuario provocará que el navegador **lance una petición, de tipo HTTP POST, que será dirigida hacia la acción.**
- **El binder detectará el parámetro de tipo Friend e instanciará el objeto** que enviará al método en su invocación, poblando cada una de sus propiedades
- **Cuando le toca el turno a la propiedad Address, instancia un objeto de dicha clase** y busca en el contexto parámetros de la forma "address.street" y "address.zipcode" para asignar a sus propiedades.

Controladores

Propiedades complejas

- Con lo cual, la acción en el controlador quedará de la siguiente forma:

```
public IActionResult Update(Student student)
{
    return Content(student.Name + ", " +
    student.Age + ", " +
    "Phone: " + student.Phone + ", " +
    "Street: " + student.Address.Street + ", " + "Zip code: " + student.Address.ZipCode);
}
```

Controladores

Enlace automático de listas

- El Model Binder también es capaz de enlazar automáticamente parámetros de lista como un array o un List<T>.
- Por ejemplo:

```
public class TestController : Controller
{
    public ActionResult Array(string[] text)
    {
        return Content(string.Join(", ", text));
    }
}
```

Controladores

Enlace automático de listas

- Si se produce una petición como **/Test/Array?text=one&text=two&text=three,** el método será invocado correctamente y el parámetro text tendrá los valores "one", "two" y "three" en las posiciones 0, 1 y 2 del array, retornando al usuario el texto "one, two, three“.
- El comportamiento será idéntico con parámetros de tipo lista, colecciones o enumerables que actúen sobre tipos simples, como un string o int.

FORMACIÓN PROFESIONAL MONTECASTELO

Actividad 4

*Practicaremos el trabajo con **propiedades complejas** con estas actividades.*

- *Crea un modelo de vista "**Pedido**" que contenga las propiedades "**Cliente**", "**Direccion**", "**Ciudad**" y "**Productos**". La propiedad "**Productos**" debe ser una lista de objetos "**Producto**" que contienen las propiedades "**Nombre**" y "**Precio**". Crea una acción "**CrearPedido**" en un controlador "**PedidoController**" que acepte un objeto "**Pedido**" como parámetro. Crea una vista para la acción "**CrearPedido**" que muestre un formulario para ingresar los datos del pedido.*

- *Crea un modelo "**Busqueda**" que contenga las propiedades "**Nombre**", "**Descripcion**", "**PrecioMinimo**" y "**PrecioMaximo**". Crea una acción "**BuscarProductos**" en un controlador "**ProductoController**" que acepte un objeto "**Busqueda**" como parámetro. La acción debería buscar los productos que coincidan con los criterios de búsqueda y mostrar los resultados en una vista. Crea una vista para la acción "**BuscarProductos**" que muestre un formulario para ingresar los criterios de búsqueda.*

- *Crea un modelo "**Registro**" que contenga las propiedades "**Nombre**", "**Apellido**", "**CorreoElectronico**" y "**Direccion**". La propiedad "**Direccion**" debe ser un objeto "**Direccion**" que contenga las propiedades "**Calle**", "**Ciudad**" y "**CodigoPostal**". Crea una acción "**RegistrarUsuario**" en un controlador "**UsuarioController**" que acepte un objeto "Registro" como parámetro. Crea una vista para la acción "**RegistrarUsuario**" que muestre un formulario para ingresar los datos de registro.*

15 MINUTOS

FORMACIÓN PROFESIONAL MONTECASTELO

BINDING

Controladores

Binding

- **Para modificar el** comportamiento por defecto del **binding**, el **mecanismo que nos permite enlazar los parámetros de la petición con los del método de acción y con propiedades de entidades del Modelo, utilizaremos el atributo [Bind]**

Controladores

Atributo [Bind]

- Con este atributo podemos **modificar determinados aspectos del proceso de binding en el parámetro del método de acción**
- Por ejemplo:

```
public IActionResult Update(int id, [Bind(options)] Friend friend)
{
    return View();
}
```

Controladores

Atributo [Bind]

- En el ejemplo anterior, podemos ver la diferencia:
  - El parámetro id sería enlazado utilizando el comportamiento por defecto del Model Binder
  - Para el segundo parámetro se indicarán una serie de opciones que lo modifican y que pueden ser:
    - **Especificación de prefijo**
    - **Inclusión de propiedades concretas**

Controladores

Atributo [Bind]

- **Especificación de prefijo**
  - **Permite indicar que los parámetros se encuentran en la petición con una denominación de la forma "Prefijo.Parametro".**
  - En el siguiente ejemplo, la propiedad Name de Friend podrá encontrarse como "Person.Name":

```
public ActionResult Update([Bind(Prefix = "person")] Friend friend)
{
    return View();
}
```

Controladores

Atributo [Bind]

- **Inclusión de propiedades concretas**
  - Esta opción **permite especificar qué propiedades concretas de la entidad deseamos actualizar**.
  - Es interesante para evitar la inyección de valores no deseados en propiedades importantes, simplemente apareciendo en el contexto de petición un parámetro con su nombre.
  - Esta técnica de hacking se suele denominar "overposting".

Controladores

Atributo [Bind]

- **Inclusión de propiedades concretas**
  - Por ejemplo, imagina el siguiente modelo y la siguiente vista:

```
public class User
{
    public string Username { get; set; }
    public string Password { get; set; }
    public bool IsAdmin { get; set; }
}
```

```
<form method="post" action="/user/changepassword">
    Username: <input type="text" name="Username"><br>
    Password: <input type="password" name="Password"><br>
</form>
```

Controladores

Atributo [Bind]

- **Inclusión de propiedades concretas**
  - En el controlador podríamos definir una acción semejante a este código:

```
public IActionResult ChangePassword(User user)
{
    // Validar y actualizar el modelo
    return View();
}
```

  - Sin embargo, **el problema de este código es que**, **aunque** en el formulario **no se encuentre el campo IsAdmin** y por tanto este valor no viaje en la petición, **un usuario malintencionado podría introducirlo** deliberadamente en el querystring **con un valor true y hacer que el método funcione**.

Controladores

Atributo [Bind]

- **Inclusión de propiedades concretas**
  - ¿La solución? **Indicar** con el atributo Bind **la lista de propiedades de la entidad a las que se desea buscar un valor en el contexto, ignorando el resto**:

```
public ActionResult Update([Bind("Username,Password")] User u)
{
    return View();
}
```

  - Esta técnica es conocida como **white list binding**, o enlace por lista blanca.

FORMACIÓN PROFESIONAL MONTECASTELO

Actividad 5

*Practicaremos el trabajo con el **atributo Bind** con estas actividades.*

- *Crea una acción de controlador que acepte los datos de un formulario HTML mediante POST y use el atributo Bind para incluir solo las propiedades necesarias del modelo en el binding. Por ejemplo, si el modelo tiene cinco propiedades y solo se necesitan dos en el binding, usa Bind para incluir solo esas dos propiedades.*
- *Crea un modelo de vista que tenga una propiedad de lista de objetos y usa el atributo Bind para incluir solo las propiedades necesarias de los objetos de la lista en el binding. Por ejemplo, si cada objeto tiene cinco propiedades y solo se necesitan dos en el binding, usa Bind para incluir solo esas dos propiedades de cada objeto en la lista.*
- *Crea una vista que muestre solo un subconjunto de las propiedades del modelo de vista y usa el atributo Bind para incluir solo esas propiedades en el binding cuando se envía el formulario. Por ejemplo, si el modelo de vista tiene cinco propiedades y solo se muestran dos en la vista, usa Bind para incluir solo esas dos propiedades en el binding.*
- *Crea una acción de controlador que acepte datos de un formulario HTML mediante POST y use el atributo Bind con la opción Exclude para excluir una o más propiedades del modelo del binding. Por ejemplo, si el modelo tiene cinco propiedades y se desea excluir una de ellas del binding, usa Bind con la opción Exclude para excluir esa propiedad.*

15 MINUTOS

FORMACIÓN PROFESIONAL MONTECASTELO

VALIDACIÓN DEL MODELO

Controladores

Validación del modelo

- Ya vimos cómo el controlador utiliza el **mecanismo de binding** para **instanciar y poblar automáticamente entidades complejas.**

- Con lo cual, **cuando a un método de acción llega un parámetro de tipo complejo, el binder ya habrá comprobado si cumple las condiciones** impuestas por el Modelo**.**

Controladores

Validación del modelo

- Tras la comprobación, hay dos elementos fundamentales para la validación:
  - La **colección ModelState** de nuestros controladores, **donde se almacena el conjunto de errores detectados**
  - La **propiedad ModelState.IsValid**, que **indicará si podemos considerar válidos los datos recibidos**. Además esta propiedad se utiliza también desde la Vista, entre otras cosas, para mostrar mensajes de error al usuario.

Controladores

Validación del modelo

- El ejemplo típico de una acción que recibe datos desde un formulario puede ser:

```
[HttpPost]
public ActionResult Update(Friend friend)
{
    if (!ModelState.IsValid)
    {
        return View("Edit", friend);
    }
    // Actualiza el modelo y devuelve
    // una vista al usuario
    return View();
}
```

- Si la instancia de Friend no cumple las restricciones definidas, se retorna al usuario de nuevo a la vista de edición, enviándole como parámetro el objeto Friend.

Controladores

Validaciones de control

- Desde el controlador se pueden realizar comprobaciones sobre los datos recibidos y su contexto, como si el usuario actual está logueado:

```
[HttpPost]
public ActionResult Update(Friend friend)
{
    if (!User.Identity.IsAuthenticated)
    {
        ModelState.AddModelError("", "Operation is not authorized");
    }

    if (!ModelState.IsValid)
    {
        return View("Edit", friend);
    }
    // Actualiza el modelo y devuelve
    // una vista al usuario
    return View();
}
```

Controladores

Validaciones de control

- En este ejemplo, lo único que estamos haciendo es añadir a la colección ModelState un error, donde el primer parámetro es el nombre de la propiedad causante del problema (en este caso, ninguna) y el segundo es el mensaje que se mostrará al usuario.

- **ModelState.IsValid** será true cuando la colección ModelState esté limpia de errores.

Práctica guiada

Práctica guiada 5: Binding y validaciones

FORMACIÓN PROFESIONAL MONTECASTELO

ACCESO A SERVICIOS DEL MODELO

Controladores

Acceso a servicios del Modelo

- Una de las **principales misiones de los controladores** es el **acceso a servicios del Modelo.**

- En general obtendremos una instancia de los componentes de dicha capa, e invocaremos en ella los métodos necesarios para conseguir los objetivos pretendidos por la acción, como en el siguiente ejemplo:

Controladores

Acceso a servicios del Modelo

```
private readonly BlogContext _contexto;

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

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

Controladores

Acceso a servicios del Modelo

- Este código, sin embargo, tiene un fuerte inconveniente, **la instanciación del Modelo debemos repetirla en todas las acciones que lo necesitemos.**

- Veremos una forma de mejorarlo, por ejemplo, **instanciar el Modelo a nivel de controlador**.

Controladores

Acceso a servicios del Modelo

```
BlogManager manager = new BlogManager();

public IActionResult Index()
{
    var posts = manager.GetLatestPosts(10); // 10 ultimos posts
    return View("Index", posts);
}

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

FORMACIÓN PROFESIONAL MONTECASTELO

VARIABLES DE SESIÓN, APLICACIÓN Y CACHÉ

Controladores

Variables de sesión, aplicación y caché

- Las **variables de sesión, aplicación y caché son mecanismos proporcionados por la plataforma para salvar el hecho de que la web es stateless, y mantener el estado entre peticiones.**
- Describiremos a continuación cada una de ellas.

Controladores

Variables de sesión, aplicación y caché

- IMPORTANTE: Para poder utilizar estas variables, deberemos especificarlo en el **Program.cs**

```
builder.Services.AddSession();
```

```
app.UseSession();
```

Controladores

Variable de sesión

- **Un usuario en la web dispone de un área de memoria donde puede almacenar pares clave-valor, y cuyo contenido es persistente entre llamadas.**

- Mientras el usuario está activo, el contenido de este diccionario se mantendrá en memoria; transcurrido un tiempo determinado desde su última interacción con el sistema, se considerará desconectado, y su contenido de sesión será eliminado.

Controladores

Variable de sesión

- Un usuario sólo podrá acceder a su propio conjunto de variables de sesión.

- **Cada una de las entradas del diccionario, en las que puede almacenarse o de las que puede recuperarse información**, se denomina **variable de sesión.**

Controladores

Variable de sesión

- **OJO:** Si queremos **guardar algún valor que no sea un entero o un string en la variable de sesión**, **debemos serializarlo** previamente. Es decir, convertirlo en un array de bytes.

- Para serializarlo, utilizaremos la clase **JsonSerializer**, junto con los **métodos Serialize y Deserialize.**

Controladores

Variable de sesión

- **Serializar un objeto:**

```
string jsonString = JsonSerializer.Serialize(lista);
return Encoding.UTF8.GetBytes(jsonString);
```

- **Deserializar un objeto:**

```
string jsonString = Encoding.UTF8.GetString(bytes);
return JsonSerializer.Deserialize<List<string>>(jsonString);
```

Controladores

Variable de sesión

- Establecer un byte[] en la variable de sesión:

```
HttpContext.Session.Set("LISTA", arrayBytes);
```

- Recuperar un byte[] de la variable de sesión:

```
arrayBytes = HttpContext.Session.Get("LISTA");
```

Controladores

Variable de sesión

- **Establecer un valor string en la variable de sesión:**

```
HttpContext.Session.SetString("[CLAVE]", "[VALOR]");
```

- **Recuperar un valor string de la variable de sesión:**

```
string? VALOR = HttpContext.Session.GetString("[CLAVE]");
```

Controladores

Variable de aplicación

- Las **variables de aplicación** son muy similares, **almacenan información usando un diccionario mediante el cual asociaremos una clave a un valor determinado, pero, a diferencia de las variables de sesión, el contenido es compartido por todos los usuarios del sistema, y persistirá en memoria mientras la aplicación se encuentre activa.**

Controladores

Variable de aplicación

- **Establecer un valor en la variable de aplicación:**

```
HttpContext.Items["[CLAVE]"] = "[VALOR]";
```

- **Recuperar un valor de la variable de aplicación:**

```
string? VALOR = HttpContext.Items["[CLAVE]"].ToString();
```

Controladores

Caché

- La caché es un **mecanismo bastante similar a las variables de aplicación, puesto que es información compartida**.
- **La principal diferencia** con ellas **es que**, en este caso, **cada entrada del diccionario llevará asociado un tiempo o condiciones de caducidad** que definirá hasta qué momento estará disponible la información.

Controladores

Caché

- Para almacenar una respuesta de un controlador en caché, utilizamos el atributo **[ResponseCache]** y le asignamos un parámetro:

  - **Duration:** especifica la duración en segundos durante la cual se almacenará en caché la respuesta.
  - **Location**: especifica dónde se almacenará en caché la respuesta
  - **NoStore**: indica si se deben almacenar en caché las respuestas o no

Controladores

Caché

- Para almacenar una respuesta de un controlador en caché, utilizamos el atributo **[ResponseCache]** y le asignamos un parámetro:
  - **VaryByHeader**: especifica los encabezados de la solicitud que deben tenerse en cuenta para determinar si la respuesta almacenada en caché es adecuada para una solicitud posterior
  - **VaryByQueryKeys**: especifica las claves de consulta que se deben tener en cuenta para determinar si la respuesta almacenada en caché es adecuada para una solicitud posterior.
  - **VaryByRoute**: especifica las partes de la ruta que deben tenerse en cuenta para determinar si la respuesta almacenada en caché es adecuada para una solicitud posterior.

Controladores

Caché

- **Por ejemplo, [ResponseCache(Duration = 10)] establece una duración de caché de 10 segundos.**

```
[ResponseCache(Duration = 10)]
public IActionResult Index()
{
    // Obtener los datos de la base de datos
    var datos = _dbContext.Datos.ToList();

    // Retornar la vista con los datos
    return View(datos);
}
```

Controladores

IMPORTANTE:

SÓLO EL CONTROLADOR PUEDE LEER Y ESTABLECER VARIABLES DE SESIÓN Y APLICACIÓN

KEEP CALM IT'S KAHOOT TIME

Tema 6: El Controlador

Asignatura: Desarrollo web en entorno servidor

CS Desarrollo de Aplicaciones Web