

UD4 Formularios y enlace de datos.

En la cuarta unidad abordaremos un aspecto fundamental para cualquier aplicación web, que es el trabajo con formularios. Hasta ahora la interacción del usuario con la aplicación se reducía a pasar determinados parámetros por medio de la url, a partir de ahora mejoraremos esta interacción permitiendo que el usuario pueda introducir datos que la aplicación podrá recuperar.

Formularios web.

Los formularios web son la forma natural de permitir introducir datos al usuario desde el navegador. Un formulario web o HTML consiste en un conjunto de controles (botones, cuadros de texto, casillas de verificación, etc..) que permiten al usuario introducir datos y enviarlos al servidor para que sean procesados en nuestra aplicación.

Un formulario va contenido en una etiqueta `<form>`. Dentro de esta etiqueta incluiremos los controles necesarios que permitan la interacción del usuario. Esta etiqueta puede contener varios atributos, siendo 2 especialmente importantes:

- **action:** indica la URL del recurso de la aplicación web al que se enviarán los datos del formulario. En el caso de una aplicación ASP.NET MVC se corresponde con un controlador.
- **method:** establece el verbo HTML que se utilizará cuando se envíe el formulario, por defecto es *post*. En una petición POST los datos se envían en el cuerpo de la petición.

Los controles de un formulario pueden contener una serie de atributos comunes, de los cuales algunos de los más importantes son los siguientes:

- **id:** identifica el control de forma única.
- **name:** este atributo es muy importante porque especifica la etiqueta o nombre con la que se guardará el dato que contiene el control, cuando el formulario se envíe al servidor.
- **value:** permite establecer un valor inicial para el control.
- **disabled:** permite deshabilitar el control.
- **readonly:** permite evitar que el control sea editable.
- **tabindex:** permite controlar el orden en el que el foco pasa de un elemento a otro del formulario utilizando el tabulador. Un orden correcto es muy importante para facilitar al usuario el trabajo con el formulario.

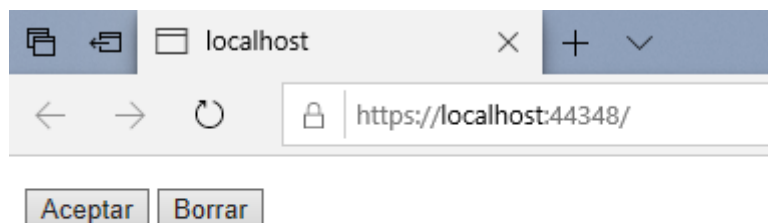
A continuación se muestran algunos de los controles más habituales.

Botón de envío y de reset.

Dentro de un formulario hay dos tipos de botones de gran importancia:

- **submit (botón de envío):** cuando se pulsa este botón los datos son enviados al servidor. Se utiliza una etiqueta `<input>` con el atributo *type* con el valor *submit*.
- **reset:** limpia los datos del formulario, volviéndolo a su estado inicial. En este caso el valor del atributo *type* será *reset*.

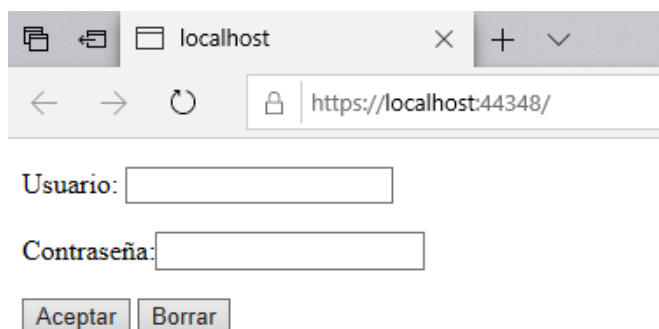
```
<form>
  <p>
    <input type="submit" value="Aceptar">
    <input type="reset" value="Borrar">
  </p>
</form>
```



Textbox.

Este control permite al usuario introducir un texto en una única línea. Utiliza también la etiqueta `<input>` pero indicando el valor `text` en el atributo `type`. Si queremos que el texto que se introduce se oculte utilizaremos el valor `password` como `type`.

```
<form>
  <p>Usuario:<input type="text" name="login"></p>
  <p>Contraseña:<input type="password" name="password"></p>
  <p>
    <input type="submit" value="Aceptar">
    <input type="reset" value="Borrar">
  </p>
</form>
```

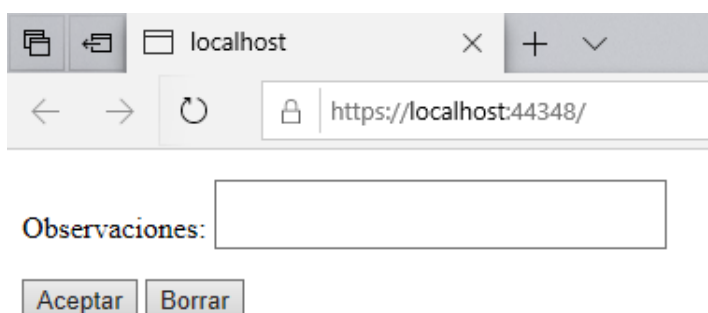


Textarea.

Similar al anterior, con la diferencia de que el texto introducido puede contener varias líneas. Utiliza la etiqueta `<textarea>` y podemos especificar el número de líneas y los caracteres de cada línea con los atributos `rows` y `cols`, respectivamente.

No tiene atributo value, por lo que si queremos tener un valor por defecto, este debe ir contenido entre las etiquetas de apertura y cierre.

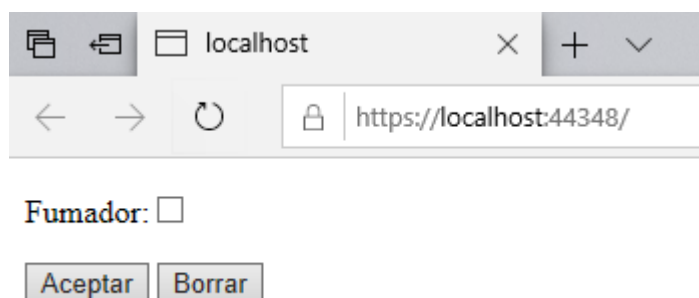
```
<form>
  <p>
    Observaciones:
    <textarea rows="2" cols="30" name="observaciones"></textarea>
  </p>
  <p>
    <input type="submit" value="Aceptar">
    <input type="reset" value="Borrar">
  </p>
</form>
```



Checkbox.

El típico control de casilla de verificación se puede conseguir con una etiqueta `<input>` y el valor `checkbox` para el atributo `type`. Podemos utilizar el atributo `checked` para que esté marcado por defecto.

```
<form>
  <p>Fumador:<input type="checkbox" name="Fumador"></p>
  <p>
    <input type="submit" value="Aceptar">
    <input type="reset" value="Borrar">
  </p>
</form>
```

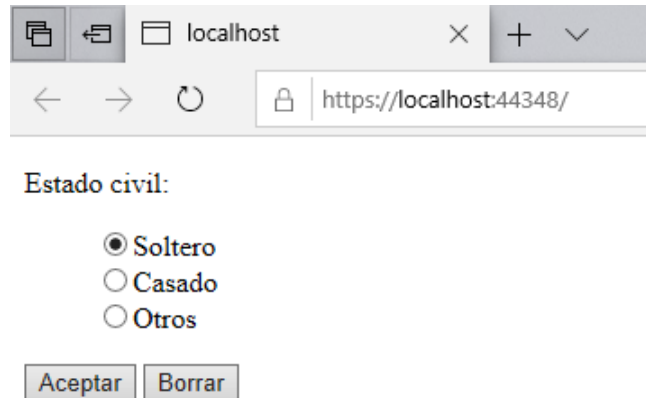


Controles de selección (radio button).

Permiten escoger entre un grupo de opciones de las que sólo podemos seleccionar una. Cada una de las opciones del grupo se define con una etiqueta `<input>` con el valor `radio` para `type`. El

atributo *name* permite agrupar a todas las que contengan el mismo valor. El atributo *checked* permite indicar la opción seleccionada por defecto.

```
<form>
  <p>Estado civil:</p>
  <ul style="list-style: none; ">
    <li><input type="radio" name="est" value="eso"
checked="checked">Soltero
    <li><input type="radio" name="est" value="bac">Casado
    <li><input type="radio" name="est" value="cfm">Otros
  </ul>
  <p>
    <input type="submit" value="Aceptar">
    <input type="reset" value="Borrar">
  </p>
</form>
```



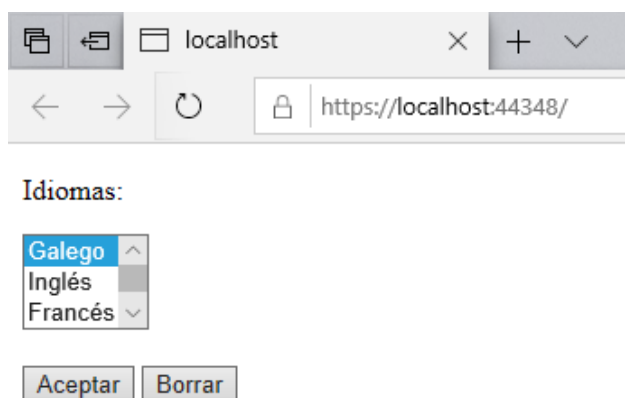
Lista de selección.

Muestra varias opciones, de las que podemos escoger una o varias. Utiliza un elemento `<select>` para definir la lista de opciones, especificando cada opción con un elemento `<option>`.

La lista puede utilizar el atributo *size* para indicar el número de elementos que se muestran al ser desplegada, y un atributo *multiple* para especificar que se puede seleccionar más de un elemento.

Para que una opción esté marcada como seleccionada por defecto, utilizaremos el atributo *selected*.

```
<form>
  <p>Idiomas:</p>
  <select name="idioma[]" size="3" multiple="multiple">
    <option selected="selected" value="g">Galego</option>
    <option value="i">Inglés</option>
    <option value="p">Francés</option>
    <option value="a">Alemán</option>
  </select>
  <p>
    <input type="submit" value="Aceptar">
    <input type="reset" value="Borrar">
  </p>
</form>
```



Ejemplo: Creación de un formulario.

Vamos a crear un ejemplo de aplicación que utilice un formulario para recibir los datos introducidos por el usuario. En nuestro caso se trata de simular una aplicación para comprobar si un usuario está registrado en el censo y por lo tanto puede votar. Para ello introduciremos el nombre completo y el DNI del usuario en un formulario.

1. Comenzamos creando un nuevo proyecto con **Archivo | Nuevo | Proyecto** y seleccionamos **Aplicación web ASP.NET Core**. Seleccionamos la plantilla proyecto vacío y añadimos soporte para MVC en la clase *Startup*, tal y como hemos hecho en los ejemplos de la anterior unidad.
2. Creamos un nuevo controlador de nombre *CensoController*, este controlador tendrá un único método de acción *Comprobar*.
3. Crearemos una vista para este método de acción (recordar que creando la vista desde el código del método, con clic derecho y desde el menú que se muestra, se creará de forma automática la estructura de carpetas de vistas y colocará la vista en el lugar que corresponde). Añadiremos también las vistas *_ViewStart.cshtml* y *_Layout.cshtml*, del mismo modo que en los ejemplos de la unidad 3.
4. Editamos el código de la vista para que muestre un formulario sencillo que recoge el nombre y el DNI para quien queremos comprobar si figura en el censo.

```
@{
    ViewData["Title"] = "Comprobar datos de censo";
}

<form action="/censo/comprobar" method="post">
    <table>
        <tr>
            <td>
                <label for="txtNombre">Nombre</label>
            </td>
            <td>
                <input type="text" id="txtNombre" name="nombre" />
            </td>
        </tr>
    </table>
</form>
```

```
<td>
    <label for="txtDni">DNI</label>
</td>
<td>
    <input type="text" id="txtDni" name="dni" />
</td>
</tr>
<tr>
<td colspan="2">
    <input type="submit" />
</td>
</tr>
</table>
</form>
```

5. Ejecutamos y añadimos `/censo/comprobar` a la url para abrir la página con nuestro formulario.

En este punto es importante tener claras una serie de cuestiones. Al indicar en nuestra URL `/censo/comprobar`, nuestra aplicación buscará un controlador que atienda a dicha petición. Siguiendo el esquema de enrutado por defecto, sería un controlador llamado *CensoController* y se ejecutaría un método de acción llamado *Comprobar()*.

Si vemos nuestra aplicación, vemos que nuestro controlador no realiza ningún trabajo, y lo único que hace es devolver la vista correspondiente, que es la que contiene el formulario.

Las vistas que contienen formularios son especiales en ASP.NET MVC en el sentido de que necesitamos dos métodos de acción para trabajar con ellas. El primero responde a una petición *HttpGet*, y es la que se envía cuando introducimos la dirección de la página que contiene el formulario. El controlador correspondiente recibe dicha petición y devuelve la vista adecuada, mostrando de este modo el formulario al usuario.

El problema es que cuando enviamos los datos del formulario se realizará una nueva petición, en este caso de tipo *HttpPost*, que contiene la información del formulario en el cuerpo de la petición, pero no tenemos un controlador que se encargue de esto. En ASP.NET MVC se suele hacer con un controlador con el mismo nombre, de modo que tendremos dos controladores con el mismo nombre para atender peticiones sobre un mismo recurso, sólo que uno atenderá las peticiones *HttpGet*, mostrando el formulario al usuario y otro recogerá la información del mismo atendiendo la petición *HttpPost*.

Ejemplo: Establecer un controlador para recibir los datos del formulario.

Siguiendo con nuestro ejemplo, vamos a editar nuestro *CensoController*, para añadir el código necesario para procesar los datos del formulario.

1. Editamos el controlador *CensoController* y añadimos el siguiente código:

```
[ActionName("Comprobar")]
[HttpPost]
public IActionResult ComprobarPost()
{
    var nombre = this.HttpContext.Request.Form["name"];
    var dni = this.HttpContext.Request.Form["dni"];

    return Content($"Nombre: {nombre}, DNI: {dni}");
}
```

En este código hay varias cosas que no hemos visto hasta ahora, así que nos pararemos un momento en analizar el código.

Como indicamos antes, queremos utilizar el mismo recurso para atender la petición get que el post de los datos del formulario (recordad que el atributo *action* del formulario indicaba que enviaría los datos al recurso */censo/comprobar*).

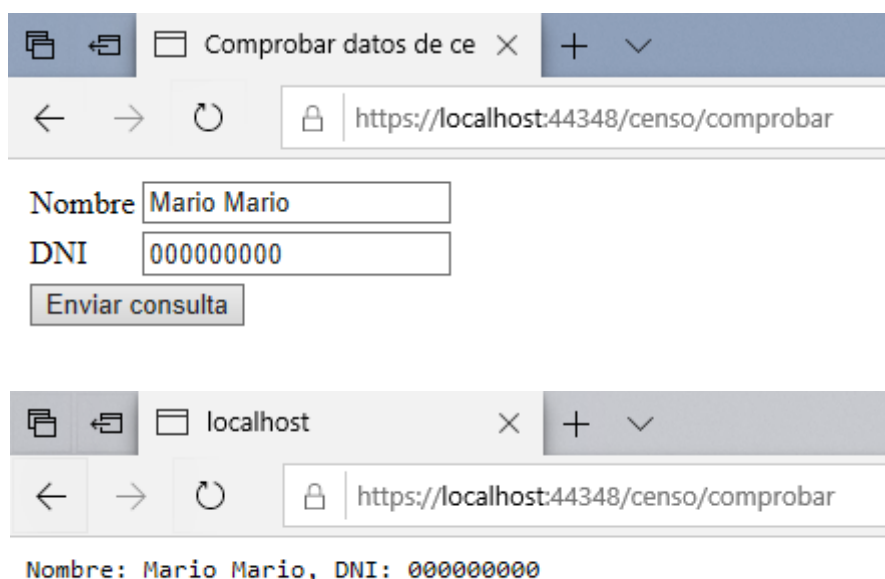
El problema es que como sabéis, no pueden existir dos métodos con el mismo nombre y la misma firma, por lo tanto no podríamos tener al mismo tiempo dos métodos llamados *Comprobar*. La solución aportada por ASP.NET MVC consiste en “decorar” nuestro método con una serie de atributos que permiten modificar su comportamiento.

El primer atributo, *ActionName*, permite especificar el nombre del método de acción que contiene el atributo, de modo que ASP.NET MVC no buscará por el nombre de método, que es el mecanismo por defecto, si no que buscará por el nombre especificado en el atributo. De esta forma, aunque hemos llamado al método *ComprobarPost*, para evitar que la firma sea la misma que nuestro método *Comprobar*, en la práctica, tanto uno como otro se mapearán para atender peticiones que lleguen a la URL */censo/comprobar*.

El problema ahora es que tenemos dos métodos de acción que están asignados al mismo recurso URL, la diferencia es que queremos que cada uno atienda un tipo de petición diferente. Para ello podemos indicar que un método de acción atienda peticiones de un tipo específico. Eso es lo que hacemos al añadir el atributo *HttpPost* a nuestro nuevo método, de modo que *ComprobarPost* atenderá las peticiones post, como era nuestra intención, procesando los datos de entrada, mientras que el otro método recibirá las peticiones get por defecto. De este modo el problema está resuelto y tenemos nuestras acciones configuradas para atender las peticiones correspondientes.

A continuación vemos que nuestro método recoge los valores recibidos del formulario y devuelve un texto con los valores capturados.

2. Para comprobar que todo funciona correctamente ejecutaremos nuestra aplicación, nos iremos a */censo/comprobar*, introduciremos los datos en el formulario y pulsaremos en el botón para enviarlos. Si todo funciona correctamente veremos un mensaje con dichos valores.



The first screenshot shows a web browser window with the title 'Comprobar datos de ce'. The address bar shows 'https://localhost:44348/censo/comprobar'. The form contains two input fields: 'Nombre' with the value 'Mario Mario' and 'DNI' with the value '000000000'. Below the fields is a button labeled 'Enviar consulta'.

The second screenshot shows the same browser window after the form is submitted. The address bar remains the same. Below the browser window, the text 'Nombre: Mario Mario, DNI: 000000000' is displayed.

Model Binding.

En nuestro ejemplo anterior hemos recogido los datos recibidos de nuestro formulario para trabajar con ellos. Para hacerlo, hemos tenido que declarar una serie de variables e ir obteniendo los datos uno a uno a partir de la colección *Form*, donde se reciben en un diccionario en el que cada dato está identificado con una etiqueta determinada.

Esta es la forma clásica de trabajar en diversos frameworks de desarrollo web, pero esto hace que el trabajo con formularios se convierta en una tarea repetitiva y tediosa, y propensa a errores. Imaginemos que tenemos un formulario con decenas de controles, tendríamos que hacer lo mismo control por control, además de que necesitamos conocer el nombre de cada etiqueta, y al estar contenidas en una cadena de texto, no tendremos ayuda del compilador y si nos equivocamos con el nombre de una etiqueta podremos provocar un error en nuestra aplicación.

Una de las características más interesantes y potentes de ASP.NET MVC es la llamada Model Binding. Es un proceso por el cual ASP.NET MVC es capaz de mapear automáticamente los datos recibidos en la petición, ya sea a través de parámetros url o datos de un formulario, a un modelo o ViewModel en nuestro controlador. Cualquier clase que contenga propiedades que coincidan en nombre con los datos del formulario puede ser mapeada automáticamente gracias al Model Binding.

Este mecanismo elimina la necesidad de leer los datos del formulario de forma manual y asignarlos a un objeto, descargándonos de las tareas tediosas y propensas a errores. El Model Binding simplifica también el proceso ya que se encarga de realizar conversiones de tipos de datos automáticamente, ya que los datos pasados en una petición web viajan siempre como texto.

Imaginemos un ejemplo en el que tenemos un formulario con dos campos, nombre y email. Si tenemos en nuestro controlador un modelo o ViewModel definido con dos propiedades llamadas Nombre y Email, el Model Binding será capaz de mapear automáticamente los datos recibidos en la petición a esas propiedades,

Ejemplo: Utilizar el Model Binding para recibir los datos del formulario.

En esta ocasión vamos a modificar nuestro ejemplo para que utilice el Model Binding y reciba los datos directamente en una clase del modelo de negocio.

1. Creamos una carpeta Models y dentro de ella una clase llamada Persona, que contenga las propiedades Nombre y Dni:

```
namespace Formularios.Models
{
    public class Persona
    {
        public string Dni { get; set; }
        public string Nombre { get; set; }
    }
}
```

2. A continuación editaremos nuestro método de acción *ComprobarPost* para que reciba como parámetro un objeto de tipo *Persona*:

```
[ActionName("Comprobar")]
[HttpPost]
public IActionResult ComprobarPost(Persona persona)
{
    var nombre = this.HttpContext.Request.Form["nombre"];
    var dni = this.HttpContext.Request.Form["dni"];

    return Content($"Nombre: {nombre}, DNI: {dni}");
}
```

3. Si observamos el código, al recibir un parámetro de tipo *Persona*, la firma de nuestro método ya no coincidiría con la del otro método *Comprobar* (esto es lo que sucede habitualmente en los formularios, ya que el método que maneja el post siempre recibe datos de algún tipo). De este modo podemos simplificar el código y cambiar de forma segura el nombre del método, dejando de necesitar el atributo *ActionName*.

```
[HttpPost]
public IActionResult Comprobar(Persona persona)
{
    var nombre = this.HttpContext.Request.Form["nombre"];
    var dni = this.HttpContext.Request.Form["dni"];

    return Content($"Nombre: {nombre}, DNI: {dni}");
}
```

4. Por último, eliminaremos el código que obtiene los datos del formulario, ya que gracias al Model Binding, ASP.NET MVC es capaz de detectar que el método de acción que va recibir los datos del formulario espera un objeto del tipo persona, por lo que creará por nosotros un objeto de este tipo y almacenará en sus propiedades los valores del formulario cuyo nombre coincida, realizando las conversiones de tipo necesarias, en su caso.

```
[HttpPost]
public IActionResult Comprobar(Persona persona)
{
    return Content($"Nombre: {persona.Nombre}, DNI: {persona.Dni}");
}
```

- Podemos ver que el código resultante es mucho más simple, pero si ejecutamos la aplicación comprobar que el resultado es idéntico.

Uso de Tag Helpers en formularios.

Si revisamos el código de nuestro formulario, veremos enseguida que crear un formulario es una tarea que nos llevará bastante tiempo y es propensa a errores. Por ejemplo, en nuestro formulario las etiquetas `<label>` y los elementos `<input>` correspondientes deben referirse al mismo elemento. Al escribirlo sin ayuda, podemos cometer un error de escritura que haga que estas ya no se asocien.

El uso de Tag Helpers en formularios nos ayudará en gran medida a evitar posibles errores y a ahorrar tiempo gracias a que podemos aprovechar las ventajas del uso de Intellisense (módulo de completado automático de código de Visual Studio).

Como siempre, lo veremos mejor mediante un ejemplo.

Ejemplo: Utilizar Tag Helpers en nuestro formulario.

Vamos a completar nuestro formulario para que emplee Tag Helpers.

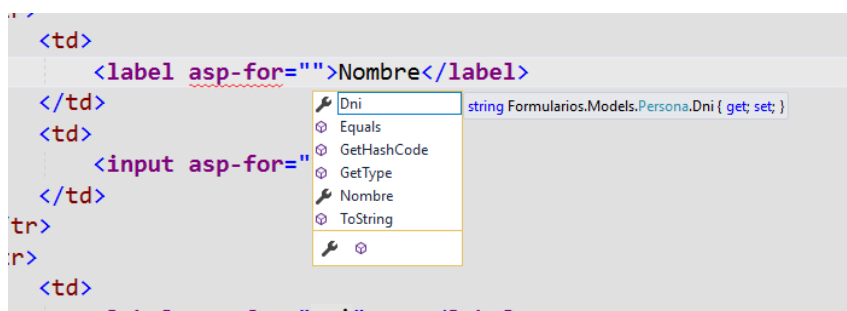
- Añadimos un elemento `_ViewImports.cshtml` en la carpeta `Views`, tal y como aprendimos en la unidad anterior. Añadimos a este fichero la importación necesaria para trabajar con Tag Helpers:

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

- A continuación editaremos el formulario para indicar que vamos a trabajar con un modelo de datos, lo que hará que nuestros helpers puedan aprovecharse del completado de código automático. Añadimos para ello la siguiente línea al principio del fichero:

```
@model Formularios.Models.Persona
```

- Ahora modificamos el formulario utilizando los helpers. Podremos observar que en el momento de introducir el atributo `asp-for`, se nos ofrecerán las distintas propiedades de nuestro modelo, simplificando la escritura de código ya que evitamos errores y además no tenemos que recordar de memoria los nombres exactos para nuestros controles del formulario:



4. El código resultante quedaría como se muestra:

```
@model Formularios.Models.Persona
@{
    ViewData["Title"] = "Comprobar datos de censo";
}

<form asp-controller="censo" asp-action="comprobar" method="post">
    <table>
        <tr>
            <td>
                <label asp-for="">Nombre</label>
            </td>
            <td>
                <input asp-for="Nombre" />
            </td>
        </tr>
        <tr>
            <td>
                <label asp-for="Dni">DNI</label>
            </td>
            <td>
                <input asp-for="Dni" />
            </td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" />
            </td>
        </tr>
    </table>
</form>
```

Si nos fijamos bien, en los controles `<input>` no hemos especificado el atributo `type`. Esto es porque los helpers son capaces de detectar el tipo de dato del modelo, y generar el código HTML adecuado en función de ese tipo. Como en nuestro caso son strings utilizará campos de texto. Esta es otra ayuda que nos ofrecen los helpers para hacer el trabajo con formularios más sencillo.

Ejemplo: Redirección a vista de resultados.

Hasta ahora, en nuestro ejemplo, una vez recuperamos los datos del formulario, los mostramos como texto directamente mediante una sentencia `return Content`. Esto no será de utilidad en una aplicación, y en su lugar lo propio sería devolver la información en una vista. El problema que nos encontramos es que en este caso no queremos devolver la vista asociada al controlador *Censo*, si no otra vista diferente, en este caso una que muestre los datos. Esto es el caso habitual en los controladores que reciben datos de un formulario, ya que la vista asociada al controlador es el propio formulario, y una vez procesado nos interesará cambiar a otra vista diferente.

Esto podemos conseguirlo de dos maneras diferentes, la más simple consiste en devolver una vista distinta, especificando su nombre como parámetro del método `View()`. La otra consistiría en una redirección a otro action method del controlador. Vamos a ver cómo utilizar las dos opciones.

1. Añadimos una nueva vista llamada *Informacion.cshtml*, en la carpeta *Views/Censo*. La vista mostrará los datos de la persona que acabamos de introducir, quedando algo como esto:

```
@model Formularios.Models.Persona

@{
    ViewData["Title"] = "Información";
}

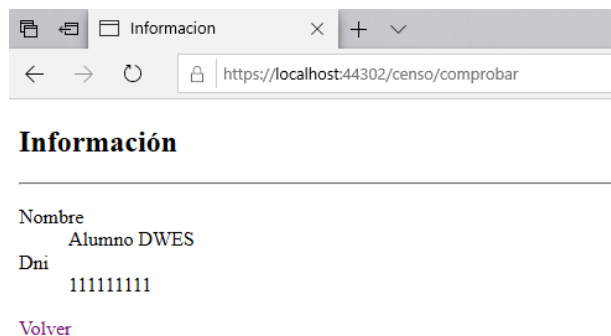
<h2>Información</h2>

<div>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            <label asp-for="Nombre">Nombre</label>
        </dt>
        <dd>
            @Model.Nombre
        </dd>
        <dt>
            <label asp-for="Dni">Dni</label>
        </dt>
        <dd>
            @Model.Dni
        </dd>
    </dl>
</div>
<div>
    <a asp-action="comprobar">Volver</a>
</div>
```

2. Ahora modificamos el método para que en lugar de devolver los datos como texto, devuelva la vista que acabamos de crear, pasando los datos de la persona a la misma. Para ello utilizamos una firma del método View que recibe el nombre de la vista a mostrar y el modelo que pasamos a la vista:

```
return View("Informacion", persona);
```

3. El resultado sería el siguiente, una vez introducimos los datos en el formulario:



Esto sería mucho más correcto, ya que mostramos los resultados en una vista, que puede utilizar la plantilla de nuestra aplicación y a la que podemos dar la apariencia que deseemos. Sin embargo esta aproximación, aunque es simple tiene un problema. Al devolver la vista desde

nuestra misma acción del controlador, la dirección url no ha cambiado, y si actualizamos la página nos preguntará si queremos reenviar el formulario, cosa que parece extraña, ya que ya no estamos viendo un formulario en el navegador.

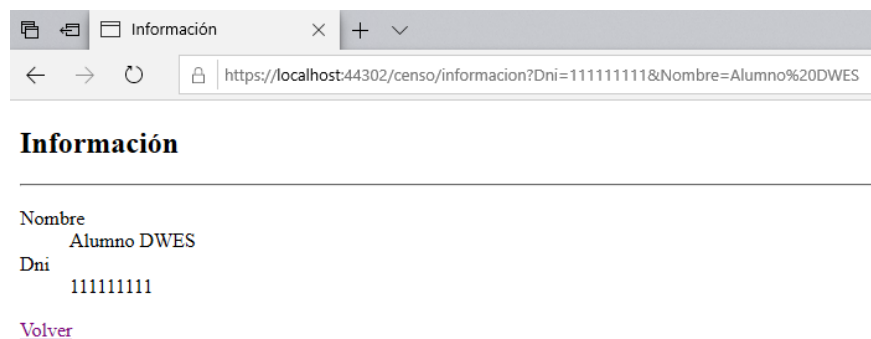
4. Vamos a modificar nuestro ejemplo para realizar una redirección a un nuevo método de acción dentro del controlador, y que sea este el encargado de devolver la vista correspondiente. Para ello creamos un nuevo método de acción llamado *Informacion*, dentro de nuestro controlador:

```
public ActionResult Informacion(Persona p)
{
    return View(p);
}
```

5. La nueva acción es muy sencilla, simplemente recibe los datos de una persona y se encarga de devolver la vista correspondiente, que muestra la información de dicha persona. Ahora lo único que nos queda es modificar el método Comprobar para que en lugar de mostrar directamente la otra vista, pase el control de la aplicación al método *Informacion*, para que sea él quien se encargue de mostrar la información deseada:

```
return RedirectToAction("informacion", persona);
```

De esta forma pasamos la información recogida en el formulario a una nueva acción, cuya misión es la de mostrar información sobre una persona. Si volvemos a probar la aplicación veremos como ahora si cambia la url al cambiar del formulario a la nueva vista.



Podemos observar que al esperar la acción un objeto persona, y al no tratarse de un formulario, ASP.NET MVC pasa los datos de una acción a otra por medio parámetros de la url. Es un poco extraño que se pase a una página todo lo que debe mostrar a través de parámetros, pero hay que tener en cuenta que nuestro ejemplo es simplemente para ilustrar el mecanismo de redirección. En una aplicación real lo más lógico sería pasar únicamente una clave o id del objeto a buscar al nuevo controlador, encargándose éste de obtener los datos completos del objeto y mostrándolo en la vista correspondiente.

Bibliografía:

- ASP.NET Core 2 Fundamentals.
OnurGumus, Mugilan T.S. Ragupathi
Copyright © 2018 Packt Publishing
- Hands-On Full-Stack Web Development with ASP.NET Core
Tamir Dresher, Amir Zuker and Shay Friedman
Copyright © 2018 Packt Publishing