UD5 Acceso a datos con ASP.NET Core.

Entidades relacionadas, cambios de modelo con Migrations y ViewModels.

Ahora que ya conocemos Entity Framework Core, y cómo utilizarlo en nuestros proyectos, vamos a desarrollar otro pequeño proyecto de prueba para ampliar conocimientos centrándonos esta vez en nuevas cuestiones. De nuevo, se trata de un proyecto "de juguete", que nos servirá para resolver determinados problemas sin la complejidad de una aplicación real, de modo que podremos prestar toda nuestra atención a las técnicas que se introducen. Al mismo tiempo, nos servirá para afianzar lo visto en la primera parte de la unidad.

En el primer ejemplo que hemos desarrollado, se utilizaba un modelo de datos muy simple, formado por una única unidad. En este caso utilizaremos dos entidades, para mostrar cómo se definen y funcionan las relaciones en EFC. En una aplicación real lo más común será encontrarnos con multitud de entidades relacionadas entre sí, pero la forma de definir dichas relaciones es la misma.

Veremos también como afrontar un problema muy habitual cuando desarrollamos una aplicación, aunque no por ello menos molesto. Este consiste en el cambio en los requerimientos de la aplicación, que requiera modificaciones en el modelo de datos, y por lo tanto en el almacén de datos correspondiente. Gracias a las Migrations la solución al problema es mucho más sencilla para un desarrollador, como comprobaremos en el ejemplo.

Por último, cuando presentamos los ViewModels hablamos de que hay casos en los que un modelo no se adapta correctamente a la plantilla de datos que necesita una vista. Hasta el momento, en nuestros ejemplos no era así, ya que las vistas mostraban información contenida únicamente en nuestro modelo, por lo que hemos utilizado estos en nuestras vistas. Vamos a ver un ejemplo claro en el que no es posible utilizar una entidad del modelo para nuestra vista y tendremos que definir un ViewModel específico que contenga la información necesaria.

Ejemplo: Creación del proyecto y el modelo.

Creamos un nuevo proyecto de ASP.NET MVC Core a partir de la plantilla MVC. Llamaremos a nuestro proyecto **Libros**, ya que consistirá en una simple página que nos permita introducir una colección de libros, que será guardada en una base de datos.

Modelo de datos.

Comenzamos, una vez más, por definir nuestro modelo de datos. En este ejemplo crearemos dos entidades, que representen a los objetos del modelo de negocio, que en nuestro caso serán libros y sus autores. Es fácil ver que estas entidades estarán relacionadas, de modo que un libro tendrá siempre un autor.

Vamos a crear estas entidades, para ello, en la carpeta *Models*, añadimos una clase *Autor*, con las siguientes propiedades:

- **Id**: por norma, utilizaremos una propiedad Id en todas nuestras entidades para identificarlas.
- **Nombre**: nombre del autor.
- **Apellidos**: apellidos del autor.

```
using System.ComponentModel.DataAnnotations;

namespace Books.Models
{
    public class Autor
    {
        public int Id { get; set; }

        [Required(ErrorMessage = "Debe introducir el nombre del autor")]
        public string Nombre { get; set; }

        [Required(ErrorMessage = "Debe introducir el/los apellidos del autor")]
        public string Apellidos { get; set; }

        public string NombreCompleto
        {
            get { return Nombre + " " + Apellidos; }
        }
    }
}
```

A continuación, vamos con la clase Libro:

- **Id**.
- **Título**: título del libro.
- **Autor**: autor del libro.

```
using System.ComponentModel.DataAnnotations;

namespace Books.Models
{
    public class Libro
    {
        public int Id { get; set; }

        [Required(ErrorMessage = "Debe introducir el título del libro")]
        public string Titulo { get; set; }

        public int AutorId { get; set; }
        public Autor Autor { get; set; }
    }
}
```

Para establecer una relación entre entidades, debemos incluir el identificador de la entidad relacionada. De este modo, dado que cada libro estará escrito por un autor (en este modelo no se contempla que un libro esté escrito por varios autores, por simplicidad), necesitamos almacenar el Id de dicho autor en los datos del libro. En EFC, por defecto, se reconoce como clave foránea un campo cuyo nombre sea “Entidad” + Id. De esta forma, la propiedad *AutorId* sería la clave externa que asocia un libro con un autor. La propiedad Autor es lo que se llama una propiedad de navegación, y permite acceder desde una entidad a una entidad asociada, esta concretamente permitiría acceder a la información de un autor desde el libro que lo contiene.

Contexto y cadena de conexión.

Una vez que tenemos nuestro modelo de datos, añadiremos el resto de elementos necesarios para trabajar con EFC. En la primera parte de la unidad vimos como utilizar el gestor de paquetes NuGet para instalar los elementos necesarios de EFC, en el caso de que necesitemos añadirlos, por haber creado nuestro proyecto con una plantilla que no los incluya por defecto. Si hemos creado un proyecto ASP.NET Core MVC con la plantilla de Visual Studio, no necesitaremos añadir estos paquetes ya que estos proyectos ya incluyen el soporte para EFC.

1. Comenzamos por el contexto de datos. Creamos una carpeta Data, y en ella una nueva clase a la que llamaremos LibrosContext. Recordamos que por convención los contextos de datos serán clases que hereden de DbContext, y cuyo nombre termine en Context,
   Añadimos un constructor que permita pasarle los parámetros de conexión, y incluimos todos los conjuntos de entidades de nuestro modelo. Para este ejercicio tendremos las colecciones Libros y Autores, de forma que nuestra base de datos contendrá dos tablas, relacionadas a través de la clave externa AutorId.

```
using Microsoft.EntityFrameworkCore;
using Books.Models;

namespace Books.Data
{
    public class LibrosContext : DbContext
    {
        public LibrosContext(DbContextOptions<LibrosContext> options) :
            base(options)
        { }

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

2. Seguidamente editamos el fichero de configuración *appsettings.json*, añadiendo la cadena de conexión:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
  "LibrosContext":
       "Server=(localdb)\\mssqllocaldb;Database=Libros;Trusted_Connection=True;Multi
       pleActiveResultSets=true"
  }
}
```

3. Por último, inicializamos e incluimos dentro de los servicios de la aplicación nuestro contexto de datos. Para ello modificamos la clase *Startup.cs*:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
```

```
            // This lambda determines whether user consent for non-essential cookies is
    needed for a given request.
            options.CheckConsentNeeded = context => true;
            options.MinimumSameSitePolicy = SameSiteMode.None;
        });

        services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);

        services.AddDbContext<LibrosContext>(options =>
            options.UseSqlServer(Configuration.GetConnectionString("LibrosContext")));
    }
```

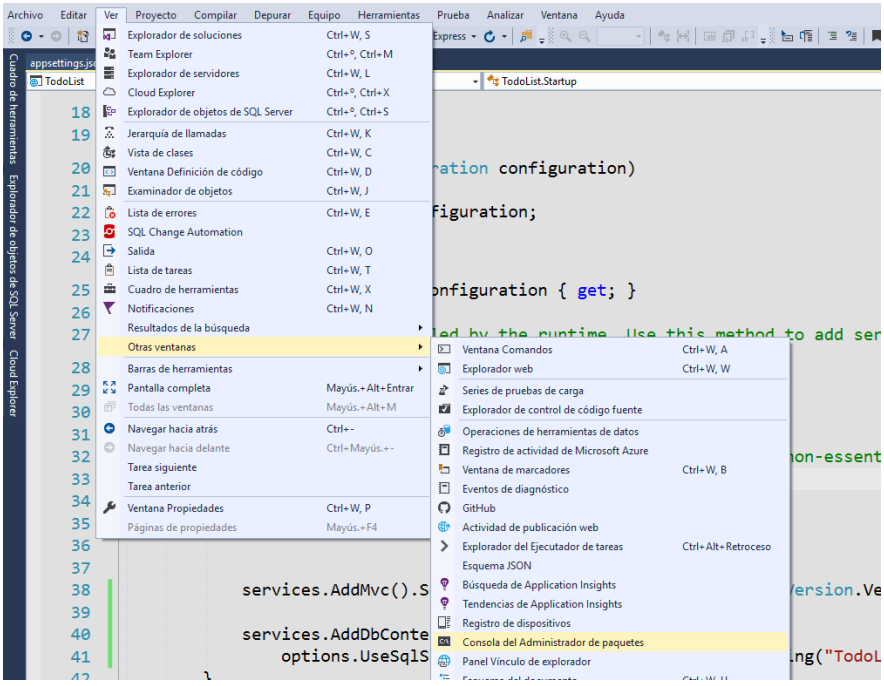
La última línea del método *ConfigureServices* añade un contexto de datos, concretamente un *TodoListContext*, a la lista de servicios de la aplicación, de modo que esté disponible cuando sea necesario. Como opciones de configuración, se especifica que utilizaremos una base de datos SqlServer, y que la cadena de conexión la obtendremos de nuestro fichero de configuración de la aplicación, guardada con el nombre *TodoListContext*.

Creación de la base de datos. Migrations.

Ya tenemos todo lo necesario para trabajar con nuestro modelo de datos, así que vamos a generar el esquema para el almacén de datos y crear la base de datos en la que se almacenará la información de nuestra aplicación.

Comenzamos por añadir una migración inicial, que capture el modelo de datos y genere el esquema correspondiente a un almacén de datos concreto, que en nuestro caso es SQL Server LocalDB.

1. Si no está visible, comenzamos por mostrar la **Consola del Administrador de paquetes** en Visual Studio.



2. Desde la consola, escribimos el siguiente comando para crear la primera migración:

```
Add-Migration InitialCreate
```

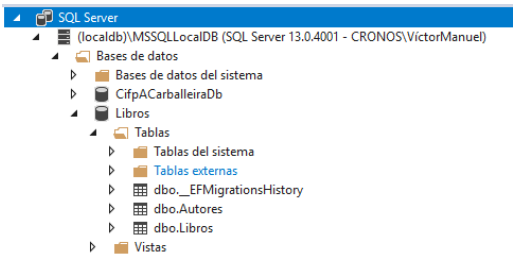
De nuevo recordamos que el nombre InitialCreate no es obligatorio, y podríamos utilizar cualquier otro para identificar nuestra migración.

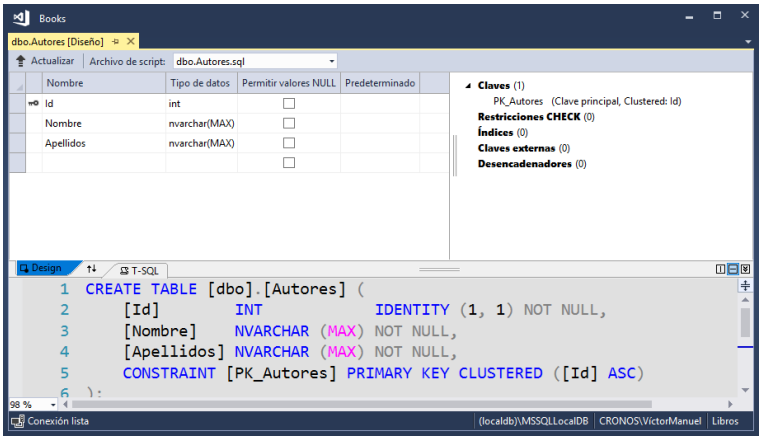
3. Si todo es correcto, se creará una carpeta Migrations en nuestro proyecto, con el código que permite actualizar la base de datos para sincronizar la misma con el modelo de datos actual.

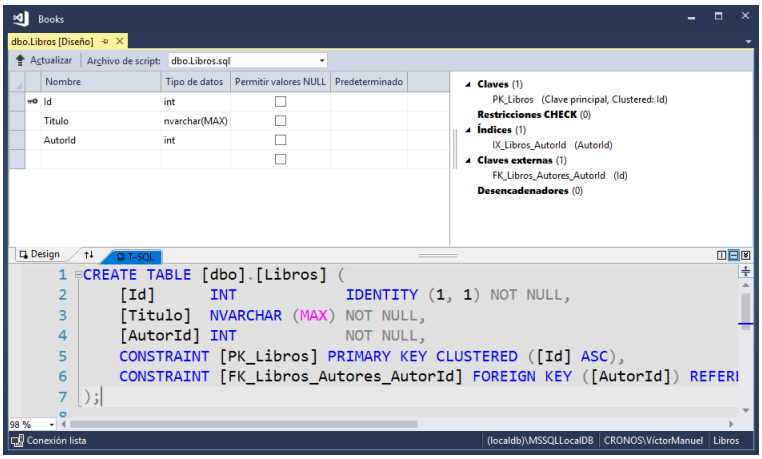
4. El último paso será, como hemos dicho, actualizar la base de datos para sincronizarla con el modelo. Como en este momento no existe dicha BD, para estar sincronizada con el proyecto se tendrá que crear de nuevo, incluyendo las tablas y relaciones necesarias:

```
Update-Database
```

5. Vamos a comprobar ahora cómo la BD ha sido creada correctamente consultando el explorador de objetos de SQL Server.
En el nodo SQL Server, abrimos la BD Libros, y comprobamos que contiene 3 tablas, una que utiliza internamente para las migraciones y las dos tablas para nuestras entidades.





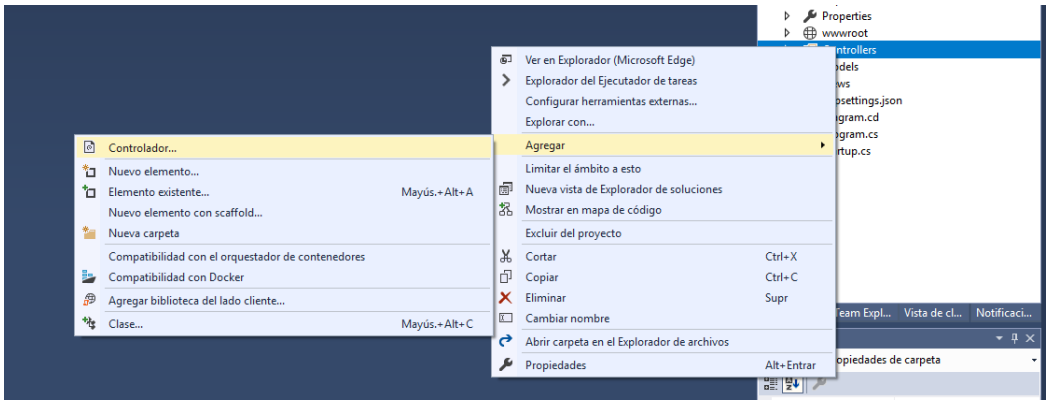


Ejemplo: Creación del controlador y las vistas.

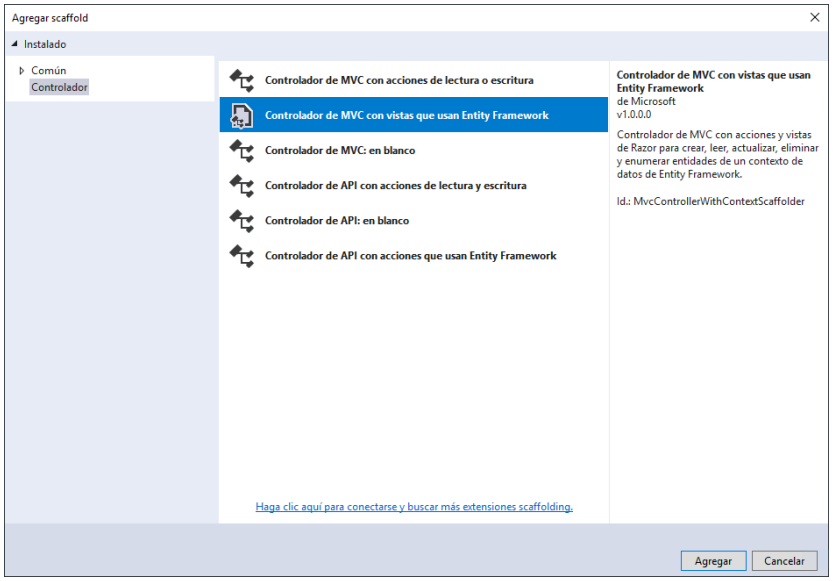
Para el siguiente paso, en esta aplicación lo normal sería crear dos controladores diferentes, cada uno con sus vistas, de modo que un controlador sería el encargado de atender todas las peticiones referentes a tareas relacionadas con los autores, tales como listados, creación, modificación o eliminación, mientras que otro controlador se encargaría de lo propio en lo referente a los libros. Del mismo modo utilizaríamos el nombre *LibrosController* y *AutoresController*, de forma que las URLs de las vistas apuntarían a /libros y a /autores, de modo que la propia ruta es explicativa de la tarea que estamos llevando a cabo.

Para simplificar el ejemplo, nos limitaremos a crear un único controlador para gestionar los libros, e introduciremos algunos autores directamente en la table de la base de datos, para que podamos asignarlos a los libros que introduzcamos.

1. Comenzamos haciendo click derecho en nuestra carpeta Controllers, y seleccionando la opción **Agregar | Controlador...**



2. En el asistente, seleccionamos el controlador de MVC con vistas que usan Entity Framework.



3. Como clase de modelo seleccionaremos Libro y en el segundo desplegable LibrosContext. Llamaremos al controlador *LibrosController*. Finalmente pulsamos en **Agregar**.

4. Una vez terminado el asistente podemos comprobar como en la carpeta *Controllers* se encuentra nuestro nuevo controlador *LibrosController*, mientras que en la carpeta *Views* tenemos una serie de vistas para dicho controlador.

5. Antes de empezar a trabajar con las tareas, vamos a preparar el resto del proyecto, eliminando aquello que sea innecesario y haciendo que la ruta por defecto sea manejada por nuestro nuevo controlador.
   Empezamos por eliminar el controlador *HomeController*, así como la carpeta *Home* de *Views*, ya que no necesitaremos ni este controlador ni sus vistas.

6. Lo siguiente será editar nuestro *_Layout.cshtml*. Lo primero que haremos es editar la cabecera, cambiando el texto del primer enlace a "Colección de libros" y modificándolo para que apunte al controlador Tareas en lugar de a Home. Eliminaremos también el resto de enlaces de la cabecera, quedando finalmente un código como el siguiente:

```
<header>
    <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-light
        bg-white border-bottom box-shadow mb-3">
        <div class="container">
            <a class="navbar-brand" asp-area="" asp-controller="Libros"
                asp-action="Index">Colección de libros</a>
            <button class="navbar-toggler" type="button" data-toggle="collapse"
                data-target=".navbar-collapse"
                aria-controls="navbarSupportedContent"
                aria-expanded="false" aria-label="Toggle navigation">
                <span class="navbar-toggler-icon"></span>
            </button>
        </div>
    </nav>
</header>
```

   En el pie eliminaremos también el enlace a la página de privacidad:

```
<footer class="border-top footer text-muted">
    <div class="container">
        &copy; 2019 - Libros
    </div>
</footer>
```

   Y añadimos la siguiente línea a la cabecera para incluir soporte para Font Awesome:

```
<link  rel="stylesheet"  href="https://use.fontawesome.com/releases/v5.6.3/css/all.css"
integrity="sha384-UHRtZLI+pbxtHCWp1t77Bi1L4ZtiqrqD80Kn4Z8NTSRyMA2Fd33n5dQ8lWUE00s/"
crossorigin="anonymous">
```

7. El último paso será cambiar la ruta por defecto de nuestra aplicación web, para que nuestro controlador inicial sea Tareas:

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Libros}/{action=Index}/{id?}");
});
```

8. Si ejecutamos en este momento nuestra aplicación, obtendremos una página con una lista de libros vacía, con un enlace que permite añadir un nuevo libro.

Ejemplo: Añadir libros.

Para nuestra aplicación nos interesa tener una página principal en la que se muestre la lista de libros de nuestra colección, además de un formulario que permita añadir un nuevo libro. El cliente prefiere esta forma de trabajar porque le parece más cómodo, así que tendremos que realizar varios cambios en nuestras plantillas.

A pesar de que lo que se nos pide no es lo que tenemos, sí que es cierto que podemos aprovechar parte del código de nuestras plantillas. Básicamente necesitamos juntar en una sola vista el formulario de la vista *Create*, con el listado de libros que se proporciona en la vista *Index*. Como esta vista será la vista principal, lo que haremos será copiar el código de *Create* en la última.

El problema que nos encontramos es que, cuando nos fijamos, vemos que el modelo de datos de la página Create es un objeto de tipo Libro, que es enlazado con el formulario para poder introducir un nuevo libro. Por el contrario, la vista *Index* muestra una lista con los libros que ya tenemos, por lo tanto su modelo de datos será un objeto de tipo lista. Está claro que en este caso las clases del modelo no nos sirven, porque para nuestra nueva vista necesitaremos un Libro que introducir en el formulario, además de la lista de libros disponibles. Para poder construir nuestra nueva vista debemos por lo tanto crear un nuevo ViewModel, específico para esta vista (de ahí el nombre de ViewModel, ya que se trata de modelos de datos para vistas específicas) que incluya la información necesaria.

1. Empezamos por crear una carpeta *ViewModels*, y dentro de la misma creamos una nueva clase *NuevoLibroViewModel*. Esta clase contendrá un objeto de tipo libro que sea capaz de recibir los datos del formulario, además de una lista de libros para la tabla:

```
using Books.Models;
using System.Collections.Generic;

namespace Books.ViewModels
{
    public class NuevoLibroViewModel
    {
        public Libro NuevoLibro { get; set; }
        public List<Libro> Libros { get; set; }
    }
}
```

2. Ahora copiamos el formulario que ha creado el asistente para la vista *Create* en la vista *Index*, inmediatamente antes que la lista de libros. Cambiamos el modelo que recibe la vista por el *ViewModel* que acabamos de crear. Debemos añadir también una sentencia using en *_ViewImports.cshtml*, para que todas las vistas tengan acceso a los ViewModels de nuestro proyecto. La vista *Index.cshtml* quedaría como sigue:

```
@model NuevoLibroViewModel

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

<br />
<h4>Nuevo libro</h4>
<hr />
<form asp-action="Index">
    <div class="form-group row">
        <label asp-for="NuevoLibro.Titulo" class="col-sm-1 col-form-
label">Título</label>
        <div class="col-sm-8">
```

```
            <input asp-for="NuevoLibro.Titulo" class="form-control">
        </div>
        <span asp-validation-for="NuevoLibro.Titulo" class="col-sm-3 text-
danger"></span>
    </div>
    <div class="form-group row">
        <label asp-for="NuevoLibro.AutorId" class="col-sm-1 col-form-
label">Autor</label>
        <div class="col-sm-8">
            <select asp-for="NuevoLibro.AutorId" class="form-control" asp-
items="ViewBag.AutorId"></select>
        </div>
    </div>
    <div class="form-group">
        <input type="submit" value="Añadir libro" class="btn btn-primary" />
    </div>
</form>

<br />
<hr />

<h4>Libros en la colección</h4>

<table class="table">
    <thead>
        <tr>
            <th>
                Título
            </th>
            <th>
                Autor
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.Libros)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Titulo)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Autor.NombreCompleto)
                </td>
                <td>
                    <a asp-action="Details" asp-route-id="@item.Id">
                      <i class="fas fa-search" title="Detalle"></i></a> |
                    <a asp-action="Delete" asp-route-id="@item.Id">
                      <i class="fas fa-trash-alt" title="Eliminar"></i></a>
                </td>
            </tr>
        }
    </tbody>
</table>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

En este código hay algunas cosas que conviene explicar. En primer lugar comenzamos por el formulario. Si observamos los controles del mismo, en lugar de estar ligados a propiedades de un

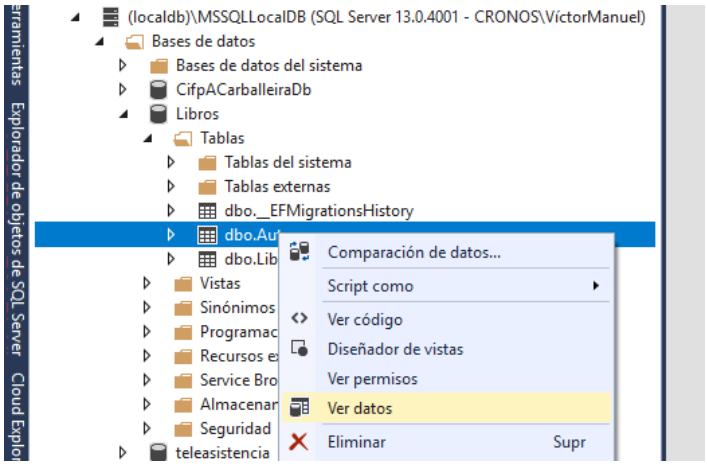
libro, como título, etc.., vemos como los hemos asociado a una propiedad NuevoLibro, seguida de una de las propiedades del libro. Es importante tener claro que en este caso estamos pasando como modelo a nuestra vista un objeto *NuevoLibroViewModel*, que contiene un libro llamado *NuevoLibro* y una lista de libros llamada *Libros*. De este modo, lo que hacemos es asociar los controles del formulario a las propiedades del libro de nuestro ViewModel, que se llama *NuevoLibro*. Así por un lado el formulario estará asociado a una de las partes del ViewModel, que almacena el nuevo libro a guardar.
La tabla, por el contrario, recorrerá los elementos contenidos dentro de la colección *Libros* del ViewModel, por lo que el bucle foreach recorre precisamente la lista *Model.Libros*.

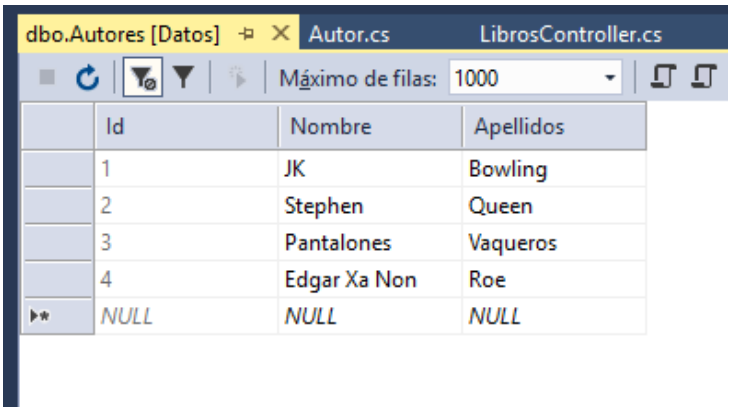
3. Si tratamos de probar nuestra aplicación obtendremos un error. Esto es debido a que nuestro método de acción Index pasa una lista de libros a la vista, y esta espera un NuevoLibroViewModel. Vamos a modificar este método para que pase el ViewModel a la vista, conteniendo un libro vacío para el formulario, además de la lista de libros que hay en la BD:

```
// GET: Libros
public async Task<IActionResult> Index()
{
    var nuevoLibroViewModel = new NuevoLibroViewModel();

    nuevoLibroViewModel.NuevoLibro = new Libro();
    nuevoLibroViewModel.Libros =
        await _context.Libros.Include(l => l.Autor).ToListAsync();

    ViewData["AutorId"] = new SelectList(_context.Autores,
        "Id", "NombreCompleto", nuevoLibroViewModel.NuevoLibro.AutorId);

    return View(nuevoLibroViewModel);
}
```

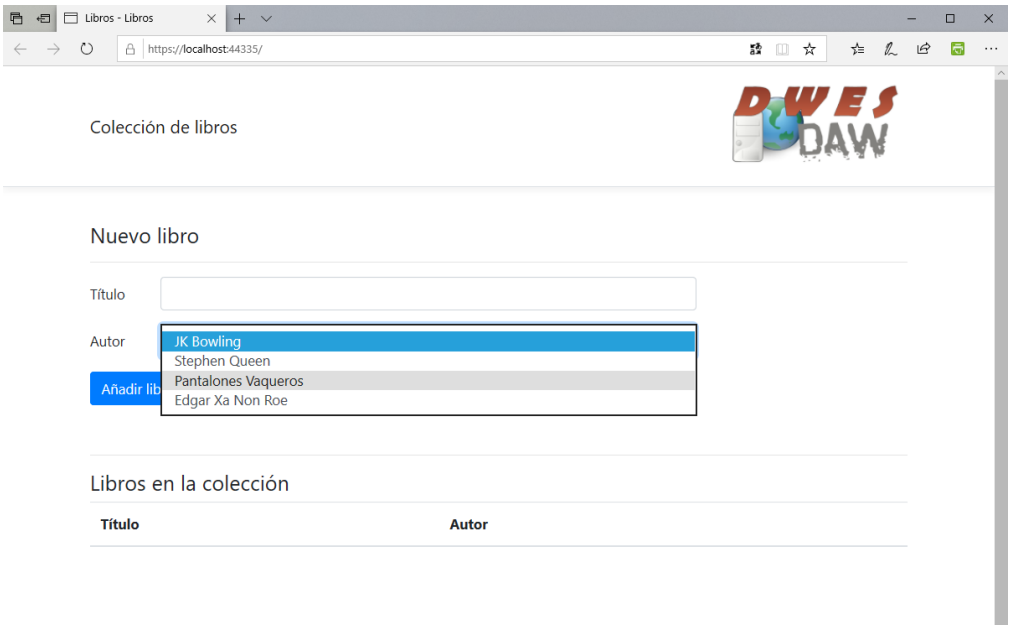
El código del método hace lo siguiente, cuando recibe una petición para mostrar la vista principal, creamos el objeto *NuevoLibroViewModel* que la vista utilizará como modelo. Para el formulario utilizamos un libro vacío, y para la lista obtenemos los libros que nos proporciona el contexto.
Algo nuevo es el método Include que se utiliza al solicitar la lista de libros al contexto. Este método lo que hace es que cuando obtiene los libros, pide que se cargue en cada uno de ellos la información del autor relacionado. Esto es equivalente a un join de SQL, pero es EFC el que se encarga de todo el trabajo, mientras que nosotros simplemente podremos acceder al autor de un libro a través de la propiedad de navegación, Autor en este caso.
Por último, para introducir de una forma más sencilla el autor de cada libro, en lugar de tener que introducir el Id, queremos ver una lista desplegable con los posibles autores. Para ello debemos pasar dicha lista como información adicional a nuestra página. Esto lo hacemos utilizando el *ViewData* / *ViewBag*. El objeto *SelectList* es un objeto especial que permite cargar un control select de modo que muestre unos datos, pero envíe otros diferentes con el formulario. En nuestro ejemplo la *SelectList* contendrá la lista de autores de la BD, guardando el valor de su Id en el campo *AutorId* de nuestro ViewModel, pero mostrando el nombre completo en la lista desplegable.

4. Para poder comprobar que la vista se muestra correctamente, y la lista desplegable se carga con los datos de los autores, necesitamos introducir primero algunos. Como indicamos anteriormente, no vamos a crear la parte de la aplicación para trabajar con autores por simplicidad, así que abrimos desde Visual Studio la tabla autores, y con el botón derecho hacemos clic en **Ver datos**.
Se abrirá la tabla en modo edición y podremos añadir algunos autores para poder probar la aplicación.





5. Ahora ya podemos ejecutar nuestro proyecto y ver la vista inicial, aunque de momento no funcionará podremos ver si se han cargado los autores correctamente.



6. Para completar nuestra tarea, debemos añadir un nuevo método de acción que reciba la petición post del formulario:

```
// POST: Libros/Index
// To protect from overposting attacks, please enable the specific properties you
want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Index([Bind("NuevoLibro")]
    NuevoLibroViewModel viewModel)
{
    if (ModelState.IsValid)
    {
        _context.Add(viewModel.NuevoLibro);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }

    ViewData["AutorId"] = new SelectList(_context.Autores, "Id",
        "NombreCompleto", viewModel.NuevoLibro.AutorId);
    return View(viewModel);
}
```

En este momento deberíamos de ser capaces de introducir nuevos libros en nuestra BD. El código es el mismo que hemos visto en anteriores ejemplos, comenzamos validando el modelo para ver que se cumplen los requisitos marcados para las propiedades. Si todo está correcto se añadirá al contexto el libro contenido en el ViewModel y se guardarán los cambios, si no es así se volverá a mostrar el formulario con los errores correspondientes.

7. En este momento nuestra aplicación ya es capaz de mostrar los datos de los libros que tenemos, además de permitir añadir nuevos libros. Para simplificar no habrá opción de editar libros, así que podemos eliminar los controladores Edit, así como su vista correspondiente. Como hemos implementado la opción de añadir nuevos libros desde la vista principal podemos prescindir también de la vista Create y los métodos con el mismo nombre.

Ejemplo: Eliminar un libro.

La eliminación de libros es una tarea relativamente simple, y en este caso podremos aprovechar la mayor parte del código generado por el asistente.

1. Comenzamos analizando el método Delete que atiende la petición get:

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

    var libro = await _context.Libros
        .Include(l => l.Autor)
        .FirstOrDefaultAsync(m => m.Id == id);
    if (libro == null)
    {
        return NotFound();
    }

    return View(libro);
}
```

Lo primero que se hace es comprobar que hemos recibido un valor para el Id dl libro a eliminar, devolviendo en este caso un resultado de NotFound(). En caso de haber recibido un Id, buscamos el libro por su identificador, y de nuevo, si no se encuentra devolvemos un error. Si el libro está disponible, se cargará junto a los datos de su autor, esto es necesario ya que la vista de confirmación para eliminar el libro mostrará la información completa del mismo. Si decidiéramos que no es necesario mostrar el autor en la vista podríamos eliminar el Include, de modo que sólo se obtendrían los datos del libro, sin su autor asociado.

2. Vamos a hacer algunos cambios en la vista *Delete.cshtml,* principalmente por cuestiones de presentación:

```
@model Books.Models.Libro

@{
    ViewData["Title"] = "Eliminar libro";
}

<h3>Eliminar libro</h3>

<h4>¿Está seguro de eliminar el libro?</h4>
<div>
    <hr />
    <dl class="row">
        <dt class="col-sm-1">
            Título
        </dt>
        <dd class="col-sm-11">
            @Html.DisplayFor(model => model.Titulo)
        </dd>
        <dt class="col-sm-1">
            Autor
        </dt>
        <dd class="col-sm-10">
            @Html.DisplayFor(model => model.Autor.NombreCompleto)
        </dd>
    </dl>

    <form asp-action="Delete">
        <input type="hidden" asp-for="Id" />
        <input type="submit" value="Eliminar" class="btn btn-danger" /> |
        <a asp-action="Index"><i class="fas fa-undo-alt"></i> Volver a lista de
libros</a>
    </form>
</div>
```

Como podemos comprobar, la página contiene un pequeño formulario con un único elemento oculto que almacena el Id del libro, de modo que si enviamos los datos del formulario al pulsar el botón Eliminar, el método Delete correspondiente recibirá dicho identificador, confirmando de este modo el libro a ser borrado.

3. El código del método de acción que recibe la confirmación de eliminación es el siguiente:

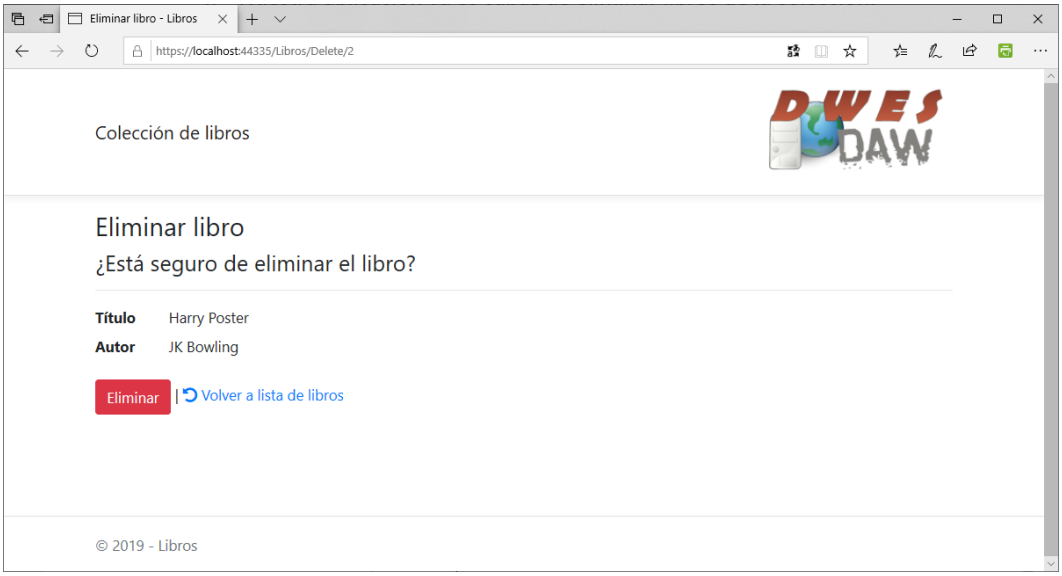
```
// POST: Libros/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var libro = await _context.Libros.FindAsync(id);
```

```
        _context.Libros.Remove(libro);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
```

Al ser la firma del método igual que la de la versión que recibe la petición get, debemos utilizar un nombre diferente para el método, e indicar que reciba peticiones para la acción Delete.

La eliminación es como siempre muy simple, primero recuperamos el libro por su Id, una vez tenemos el libro lo eliminamos de la colección Libros, y por último guardamos los cambios.

4. Nuestra aplicación ya es capaz de eliminar libros de la colección:



Ejemplo: Detalle de libro.

La vista de detalle nos permite ver toda la información de un libro. Se trata de la parte más simple de la aplicación, y no tendremos que modificar más que parte de la vista.

1. Editamos la vista para cambiar ciertos detalles de presentación:

```
@model Books.Models.Libro

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

<div>
    <br />
    <h4>Información del libro</h4>
    <hr />
    <dl class="row">
        <dt class = "col-sm-1">
            Título
        </dt>
        <dd class = "col-sm-11">
            @Html.DisplayFor(model => model.Titulo)
        </dd>
        <dt class = "col-sm-1">
```

```
            Autor
        </dt>
        <dd class = "col-sm-11">
            @Html.DisplayFor(model => model.Autor.NombreCompleto)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Index"><i class="fas fa-undo-alt"></i> Volver a lista de
libros</a>
</div>
```

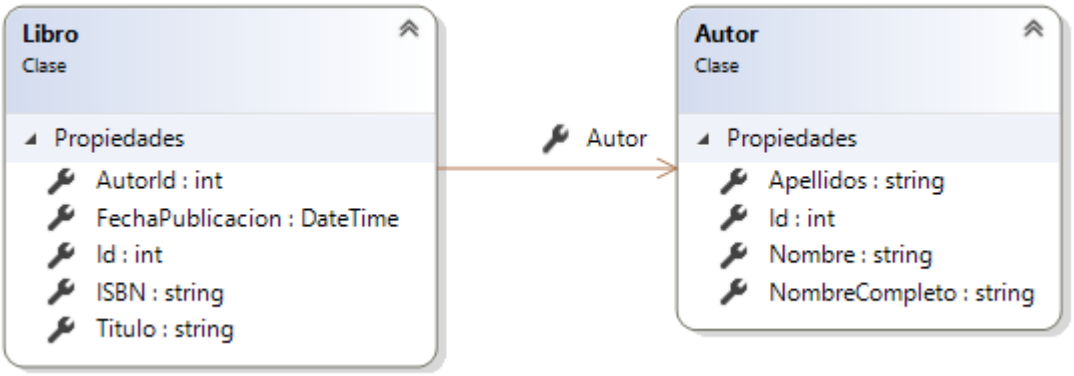
Ejemplo: Modificación del modelo y la BD usando Migrations.

Desgraciadamente, un problema habitual al que hay que enfrentarse cuando desarrollamos una aplicación de cualquier tipo tiene que ver con posibles cambios en el modelo de datos, que pueden ser producidos por malentendidos al comunicar los requerimientos del cliente, o bien por funcionalidad no incluida inicialmente, y que queremos añadir en un momento posterior.

Para nuestro ejemplo, supongamos que nos hemos dado cuenta de que hemos olvidado incluir el ISBN y la fecha de nuestros libros. Vamos a describir el proceso que seguiríamos para realizar este cambio en nuestra aplicación.

Modelo de datos.

El primer paso siempre empieza por el modelo de datos. Añadimos las nuevas propiedades a la clase libro.



De esta forma la clase quedaría como sigue:

```
using System.ComponentModel.DataAnnotations;

namespace Books.Models
{
    public class Libro
    {
        public int Id { get; set; }

        [Required(ErrorMessage = "Debe introducir el título del libro")]
        public string Titulo { get; set; }

        [Required(ErrorMessage = "Debe introducir el ISBN del libro")]
        public string ISBN { get; set; }
```

```
        public DateTime FechaPublicacion { get; set; }

        public int AutorId { get; set; }
        public Autor Autor { get; set; }
    }
}
```

Actualización de la base de datos con Migrations.

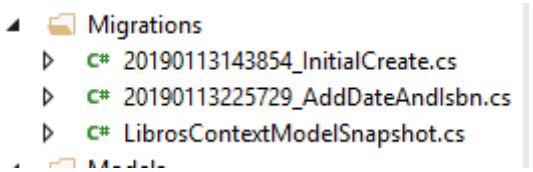
Cuando realizamos un cambio en el modelo de datos de la aplicación, uno de los mayores problemas suele tener que ver con la BD que estamos utilizando. En un entorno de desarrollo es sencillo eliminar esta BD y crearla de nuevo, pero cuando esa aplicación está en explotación esa no es una opción. Debemos mantener los datos contenidos hasta el momento y aplicar las modificaciones necesarias para que el modelo de datos y la base de datos estén sincronizadas.

1. Vamos a crear una nueva migración introduciendo el siguiente comando en la **Consola del Administrador de paquetes**.

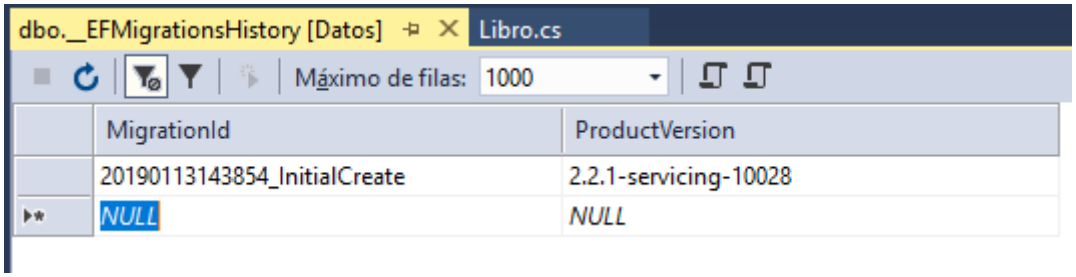
   ```
   Add-Migration AddDateAndIsbn
   ```

   El nombre de la migración puede ser cualquiera pero es de ayuda indicar los cambios que realiza cada nueva migración.

2. En este momento dispondremos en la carpeta Migrations de dos migraciones, cada una de ellas con el nombre que hemos indicado y un número que se a partir de la fecha de creación al revés. De esta forma, cada nueva migración siempre estará después de la anterior. Migrations utiliza este número para poder comprobar en qué versión se encuentra la BD, y de este modo aplica las migraciones que se hayan creado con posterioridad.

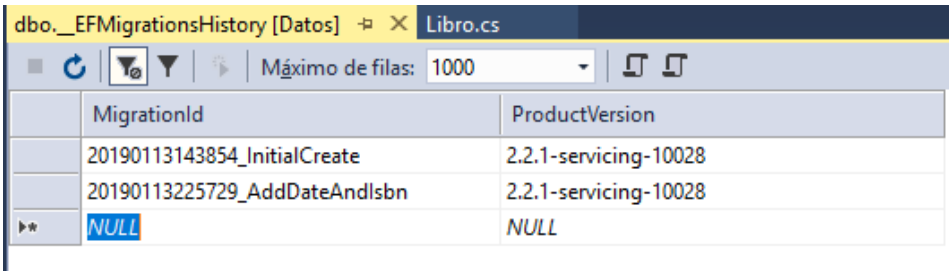
   

   Si abrimos la tabla __EFMigrationsHistory podemos ver hasta que número de migración se ha aplicado a la base de datos.

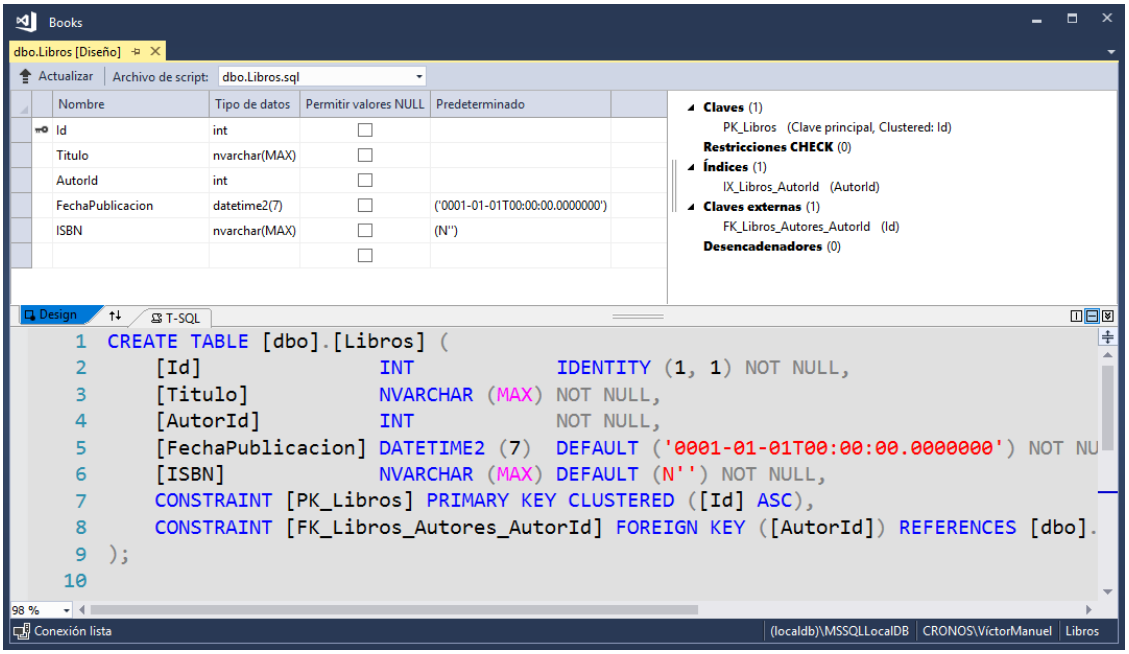
   

3. Como la base de datos ha aplicado únicamente la primera migración, al solicitar una actualización de la base de datos, se aplicarán todas aquellas migraciones creadas más tarde, en nuestro ejemplo sólo una:

   ```
   Update-Database
   ```



4. Vamos a comprobar ahora cómo la BD ha sido modificada añadiendo las dos nuevas columnas en la tabla Libros, equivalentes a las propiedades añadidas.



Vista Index.

Una vez modificado el modelo de datos y sincronizada la BD, sólo nos queda cambiar el código para incluir las nuevas propiedades. Lo haremos vista por vista, comenzando por la vista principal Index.

1. Una de las grandes ventajas de utilizar un ORM y trabajar con objetos es que simplifica mucho las tareas de modificación del código en caso de cambios en el modelo. Si observamos los métodos Index de nuestro controlador, podemos ver que lo que hacemos es crear un nuevo objeto libro que pasamos a la vista. En el formulario se recogen los datos que son enlazados con las propiedades de dicho objeto y luego simplemente añadimos este libro a la colección y guardamos los cambios. Gracias a esto no necesitamos hacer ningún cambio en el código de los métodos de acción. Si utilizáramos consultas SQL tendríamos que modificar cada una de ellas para incluir las nuevas columnas, pero al utilizar EFC, éste se encarga de hacerlo por nosotros.

   Deberemos, eso sí, modificar nuestra vista para que incluya los nuevos campos en el formulario. A continuación se muestra solamente el código de la parte del formulario, que será lo único que cambiemos. Podríamos también añadir más información a la tabla de libros pero no lo haremos por simplicidad. De la misma forma lo ideal habría sido especificar una expresión regular para el ISBN del libro.

```
<form asp-action="Index">
    <div class="form-group row">
        <label asp-for="NuevoLibro.Titulo"
          class="col-sm-1 col-form-label">Título</label>
        <div class="col-sm-8">
            <input asp-for="NuevoLibro.Titulo" class="form-control">
        </div>
        <span asp-validation-for="NuevoLibro.Titulo"
          class="col-sm-3 text-danger"></span>
    </div>
    <div class="form-group row">
        <label asp-for="NuevoLibro.ISBN"
          class="col-sm-1 col-form-label">ISBN</label>
        <div class="col-sm-8">
            <input asp-for="NuevoLibro.ISBN" class="form-control">
        </div>
        <span asp-validation-for="NuevoLibro.ISBN"
          class="col-sm-3 text-danger"></span>
    </div>
    <div class="form-group row">
        <label asp-for="NuevoLibro.FechaPublicacion"
          class="col-sm-1 col-form-label">Fecha</label>
        <div class="col-sm-8">
            <input asp-for="NuevoLibro.FechaPublicacion"
              class="form-control" type="date">
        </div>
    </div>
    <div class="form-group row">
        <label asp-for="NuevoLibro.AutorId"
          class="col-sm-1 col-form-label">Autor</label>
        <div class="col-sm-8">
            <select asp-for="NuevoLibro.AutorId"
              class="form-control" asp-items="ViewBag.AutorId"></select>
        </div>
    </div>
    <div class="form-group">
        <input type="submit" value="Añadir libro" class="btn btn-primary" />
    </div>
</form>
```

2. Simplemente añadiendo los controles necesarios al formulario, el resto del código sigue funcionando sin ningún problema, convirtiendo una tarea compleja y engorrosa en algo muy sencillo.

Vistas de detalle y eliminación.

Al igual que ocurrió con la vista principal, todo nuestro código del controlador sigue funcionando correctamente sin ningún cambio, así que únicamente adaptaremos las vistas para que reflejen la nueva información añadida.

1. Empezamos con la vista de detalle:

```
@model Books.Models.Libro

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

<div>
    <br />
```

```
    <h4>Información del libro</h4>
    <hr />
    <dl class="row">
        <dt class="col-sm-1">
            Título
        </dt>
        <dd class="col-sm-11">
            @Html.DisplayFor(model => model.Titulo)
        </dd>
        <dt class="col-sm-1">
            ISBN
        </dt>
        <dd class="col-sm-11">
            @Html.DisplayFor(model => model.ISBN)
        </dd>
        <dt class="col-sm-1">
            Fecha
        </dt>
        <dd class="col-sm-11">
            @Model.FechaPublicacion.ToShortDateString()
        </dd>
        <dt class="col-sm-1">
            Autor
        </dt>
        <dd class="col-sm-11">
            @Html.DisplayFor(model => model.Autor.NombreCompleto)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Index"><i class="fas fa-undo-alt"></i> Volver a lista de
libros</a>
</div>
```

2. Y por último los cambios en la vista de eliminación:

```
@model Books.Models.Libro

@{
    ViewData["Title"] = "Eliminar libro";
}

<h3>Eliminar libro</h3>

<h4>¿Está seguro de eliminar el libro?</h4>
<div>
    <hr />
    <dl class="row">
        <dt class="col-sm-1">
            Título
        </dt>
        <dd class="col-sm-11">
            @Html.DisplayFor(model => model.Titulo)
        </dd>
        <dt class="col-sm-1">
            ISBN
        </dt>
        <dd class="col-sm-11">
            @Html.DisplayFor(model => model.ISBN)
        </dd>
        <dt class="col-sm-1">
            Fecha
```

```
            </dt>
            <dd class="col-sm-11">
                @Model.FechaPublicacion.ToShortDateString()
            </dd>
            <dt class="col-sm-1">
                Autor
            </dt>
            <dd class="col-sm-10">
                @Html.DisplayFor(model => model.Autor.NombreCompleto)
            </dd>
        </dl>

        <form asp-action="Delete">
            <input type="hidden" asp-for="Id" />
            <input type="submit" value="Eliminar" class="btn btn-danger" /> |
            <a asp-action="Index"><i class="fas fa-undo-alt"></i> Volver a lista de
libros</a>
        </form>
</div>
```

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