

UD5 Acceso a datos con ASP.NET Core.

En esta quinta unidad nos vamos a centrar en uno de los aspectos fundamentales de cualquier aplicación, el acceso a datos. Realmente los datos son el elemento central de toda aplicación, y en un altísimo porcentaje de casos nos encontraremos con la necesidad de acceder a la información almacenada en algún almacén de datos, procesando la misma y almacenándola de nuevo en dicho almacén. Nuestra aplicación web podría verse como un simple mecanismo que proporcione una interface al usuario, para que pueda trabajar con dichos datos.

Uno de los problemas con los que nos vamos a enfrentar es que existen multitud de tecnologías de almacenamiento o gestores de bases de datos disponibles. Esto hace que sea crucial la elección del mismo ya que el código de acceso a los datos va a ser diferente dependiendo del almacén de datos utilizados. Si en un momento dado quisiéramos utilizar un gestor de bases de datos diferente, tendríamos que modificar una gran cantidad de código, razón por la que habitualmente se divide el código en una serie de capas, que ayudan a mejorar la portabilidad del código, el mantenimiento y la separación de tareas.

Por simplicidad, en esta unidad, no haremos uso de esta división en capas e implementaremos nuestro acceso a datos directamente en los controladores, de este modo nos centraremos en aprender a trabajar con un SGBD, sin tener que preocuparnos por otras cuestiones, que serán tratadas más adelante.

Como siempre, intentaremos introducir los conceptos del modo más práctico posible, en este caso, a través de la creación de una sencilla aplicación de ejemplo, que nos permitirá mantener una lista de tareas.

Modelo primero.

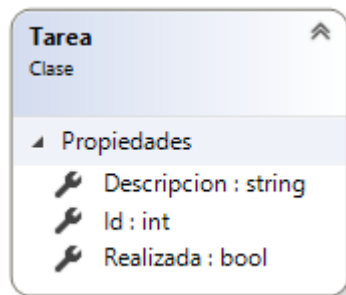
Existen diferentes aproximaciones a la hora de abordar el desarrollo de una aplicación. Hay casos en los que se parte de una base de datos existente a la que debemos adaptarnos. En otras ocasiones, hay equipos de desarrollo que prefieren por definir una base de datos en una tecnología concreta y luego desarrollar la aplicación a partir de la misma. En nuestro caso guiaremos el desarrollo por el modelo de datos, lo que nos da libertad sobre el almacén de datos concreto, de modo comenzaremos por definir el modelo de negocio de la aplicación, analizando el problema que queremos resolver e identificando las entidades que intervienen y sus posibles relaciones.

Para nuestro ejemplo vamos a partir de unos requerimientos muy simplificados, que nos permitan adquirir los nuevos conceptos, sin introducir más dificultades de las necesarias. De este modo, nuestra aplicación se limitará a mantener una lista de tareas pendientes, pudiendo añadir nuevas tareas a esta lista, además de poder marcar como realizadas las tareas que ya hemos terminado. Una tarea consistirá únicamente en un texto que describa lo que tenemos que hacer.

Del análisis de los requerimientos podemos identificar una única entidad para nuestro modelo de aplicación, esta entidad sería la tarea. De este modo vamos a crear una clase que modele dicha entidad, nuestra clase Tarea, contendrá las siguientes propiedades:

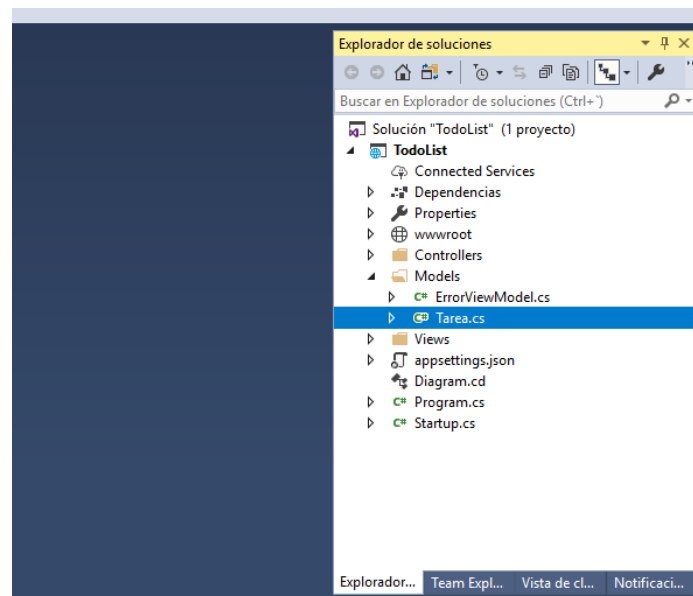
- **Id:** aunque en los requisitos no se habla de ningún tipo de identificador, es obvio que cada tarea debe poder ser identificada de forma independiente, por lo que utilizaremos una propiedad para este fin. Por defecto, utilizaremos una propiedad Id en todas nuestras entidades.

- **Descripción:** como su nombre indica, será un campo de texto con la explicación de lo que tenemos que hacer.
- **Realizada:** utilizaremos una propiedad booleana que nos permite especificar si una tarea ha sido finalizada o no, tal y como se especifica en los requerimientos.



Ejemplo: Creación del proyecto y el modelo.

Creamos un nuevo proyecto de ASP.NET MVC Core a partir de la plantilla MVC, tal y como se explicó en la unidad anterior. A continuación creamos la clase *Tarea* en la carpeta *Models*, ya que es nuestro modelo de datos.



El almacén de datos.

En este momento ya tenemos nuestro modelo de datos, si bien en este ejemplo es extremadamente sencillo, los conceptos son los mismos que si nos encontráramos ante un complejo modelo real.

Si hacemos memoria, recordaremos que para un modelo MVC, las responsabilidades están repartidas, de modo que el controlador debe funcionar como nexo de unión entre modelo y vista, siendo el encargado de recibir las peticiones del usuario, realizar las operaciones necesarias con el modelo de datos, y enviar los datos a mostrar a la vista adecuada.

El siguiente paso será elegir qué tipo de almacén de datos utilizar. La opción más clásica y habitual es la de utilizar un SGBD relacional, aunque los últimos años han proliferado tecnologías no relacionales, como por ejemplo las BD NoSQL como MongoDB o Cassandra. Aunque las bases de datos no relacionales no son siempre la mejor solución para cada tipo de problema, sí que es cierto que son adecuadas para la mayor parte de las aplicaciones. Todas están basadas en el uso del lenguaje SQL, tanto para la definición de la estructura de la BD, como para realizar todo tipo de consultas sobre la misma. El lenguaje SQL es un estándar bien conocido, y además encontramos una gran cantidad de herramientas disponibles. Entre algunos de los SGBD relacionales más utilizados podemos encontrar Oracle, SQL Server, MySQL o PostgreSQL, entre otros.

ORMs.

Cuando trabajamos con bases de datos relacionales, como indicamos en el apartado anterior, necesitamos trabajar con el lenguaje SQL. Esto obliga al desarrollador a conocer dicho lenguaje, y escribir complejas consultas que pueden resultar en ocasiones engorrosas.

Otro problema es que existe una incompatibilidad entre el modo en el que se almacenan los datos en una base de datos relacional, que guarda los datos como filas o registros en una colección de tablas, con la forma en la que modelamos los datos en nuestra aplicación, como clases o entidades que simulan objetos del modelo de negocio.

Un programador a menudo desea poder trabajar de forma cómoda y rápida con la tecnología que utiliza, por lo que la clásica forma de trabajo consistente en definir consultas SQL sobre la BD, y obtener estos como tablas que luego deben ser convertidas a los objetos de nuestra aplicación enseguida se convierte en proceso engorroso.

Para evitar todos estos problemas, surgió un modo diferente de abordar el trabajo con bases de datos relacionales, los Object Relational Mappers o ORMs. El objetivo de un ORM es permitir al desarrollador ignorar por completo los detalles de la base de datos relacional en la que se almacenan los datos de la aplicación, permitiéndole trabajar con objetos, del modo que hace habitualmente, haciendo que pueda centrarse en el código de su aplicación, dejando que el ORM se encargue de escribir las consultas por él, obteniendo además los datos como objetos de aplicación.

Entity Framework Core.

Entity Framework Core es la versión ligera y cross-platform del ORM de Microsoft, Entity Framework. Es el ORM recomendado para trabajar con proyectos de .NET, y permite a los desarrolladores trabajar con una base de datos utilizando simples objetos .NET, eliminando la necesidad de escribir código de acceso a datos específico para una base de datos relacional.

Otra de las ventajas es que soporta gran cantidad de SGBD, de modo que utilizando Entity Framework Core podemos trabajar con diferentes bases de datos relacionales, sin necesidad de escribir código específico para cada una de ellas, simplemente utilizando el proveedor adecuado. De este modo también se facilita una posible migración de un SGBD a otro.

En el siguiente enlace se pueden consultar todos los motores de bases de datos soportados:

<https://docs.microsoft.com/en-us/ef/core/providers>

Otra de las ventajas es que soporta gran cantidad de SGBD, de modo que utilizando Entity Framework Core podemos trabajar con diferentes bases de datos relacionales, sin necesidad de escribir código específico para cada una de ellas, simplemente utilizando el proveedor adecuado. De este modo también se facilita una posible migración de un SGBD a otro.

Como veremos más adelante, mediante el uso de Entity Framework Core, no necesitaremos utilizar ninguna consulta SQL, encargándose el ORM de establecer una conexión con la base de datos y generar las consultas necesarias de forma automática.

En el siguiente enlace podemos consultar la documentación de Entity Framework Core:

<https://docs.microsoft.com/en-us/ef/core>

SQL Server LocalDB.

Aunque vayamos a trabajar con un ORM como es Entity Framework Core, seguimos necesitando un SGBD para almacenar nuestros datos. Es importante recordar que el ORM es un simple sistema intermedio que nos permite trabajar de un modo más cómodo, mediante el uso de objetos, pero el ORM traducirá nuestras peticiones a una consulta que será ejecutada en una base de datos relacional.

Para los ejemplos se utilizará SQL Server LocalDB como motor de bases de datos, pero es posible utilizar cualquier otro sistema soportado por Entity Framework Core de un modo similar.

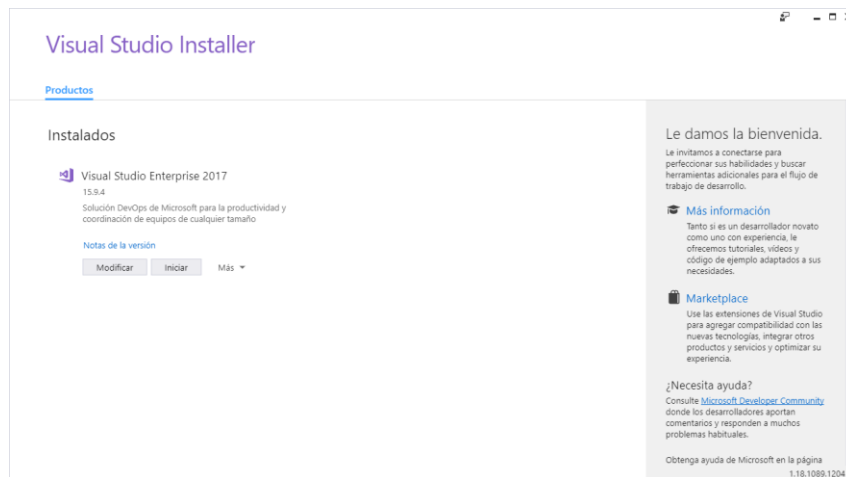
SQL Server es el SGBD de Microsoft, y uno de los más utilizados en entornos de producción. Además de las versiones empresariales, está disponible una versión Express gratuita.

Para equipos de desarrollo, Microsoft creó SQL Server LocalDB. Esta es una versión ligera del motor de bases de datos, no indicada para producción, pero sí para el desarrollo de aplicaciones. Esta herramienta se instala con Visual Studio 2017 y puede ser administrada desde el mismo entorno, siempre y cuando se haya seleccionado la carga de trabajo “Almacenamiento y procesamiento de datos” en el momento de la instalación.

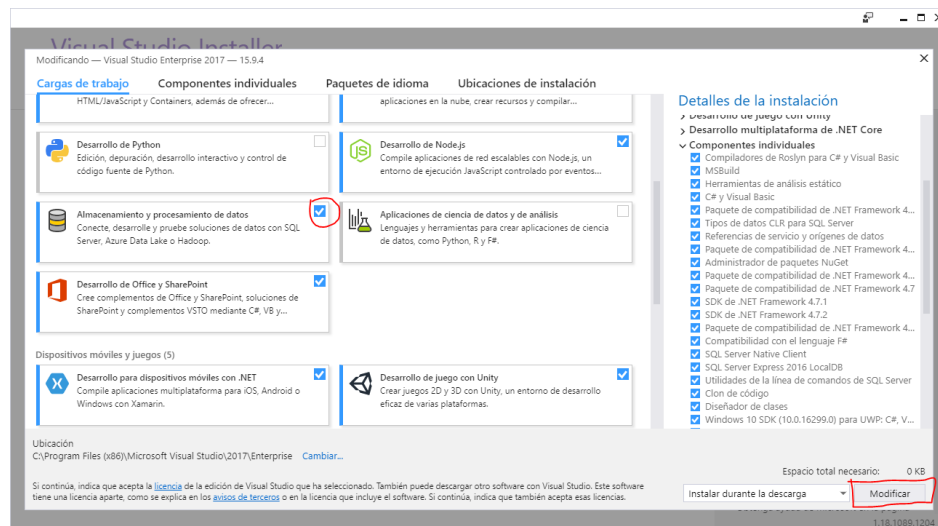
Ejemplo: Comprobar si SQL Server LocalDB está instalado.

Como acabamos de comentar, SQL Server LocalDB viene incluido en la instalación de Visual Studio 2017, si seleccionamos los componentes para trabajo con datos. Para comprobar si esto es así abriremos el instalador de Visual Studio 2017:

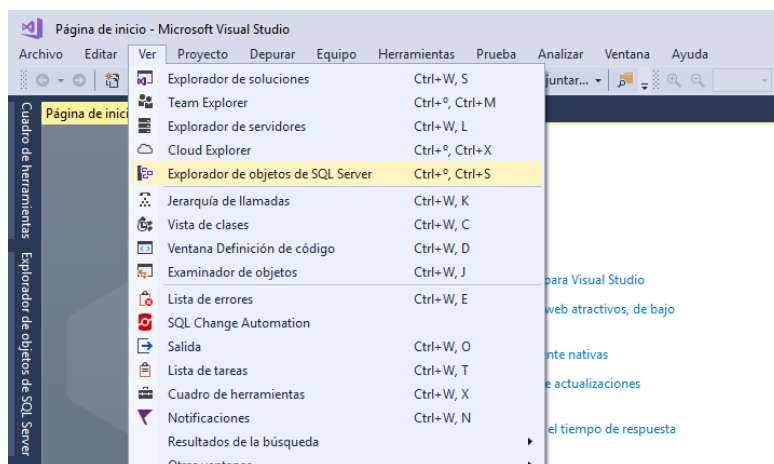
1. Pulsamos **Modificar** para ver los componentes instalados:

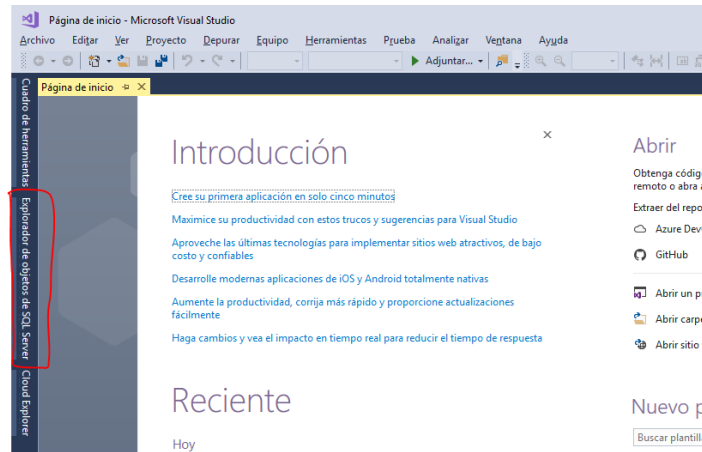


- Comprobamos que está seleccionado el módulo **Almacenamiento y procesamiento de datos**. Si no es así lo marcamos y pulsamos **Modificar**:

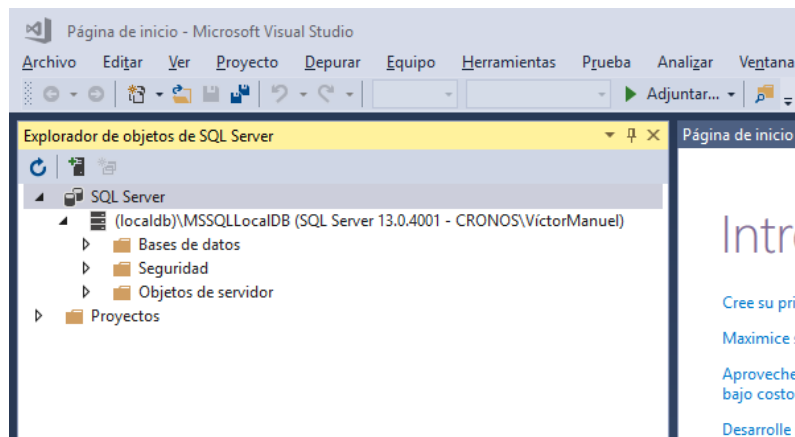


- Una vez comprobado, abriremos Visual Studio y vamos a buscar la instancia instalada. Para ello abrimos el **Explorador de objetos de SQL Server**:



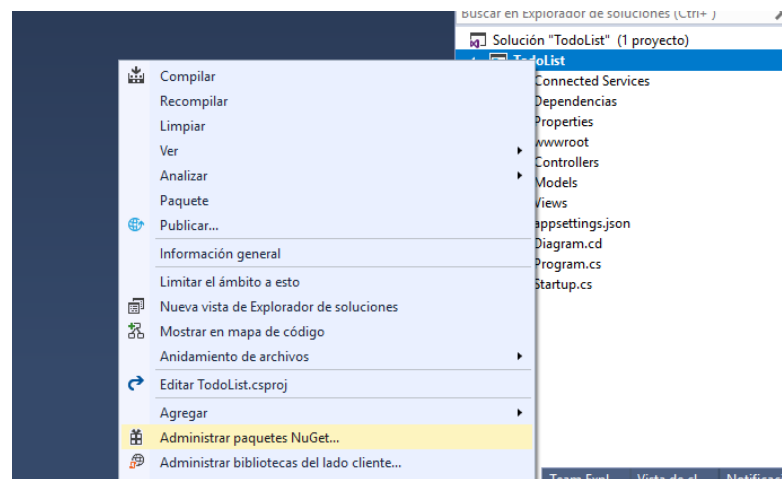


4. Con el **Explorador de objetos de SQL Server** abierto, deberíamos encontrar un nodo SQL Server, en el que encontraremos nuestra instancia del motor de bases de datos:

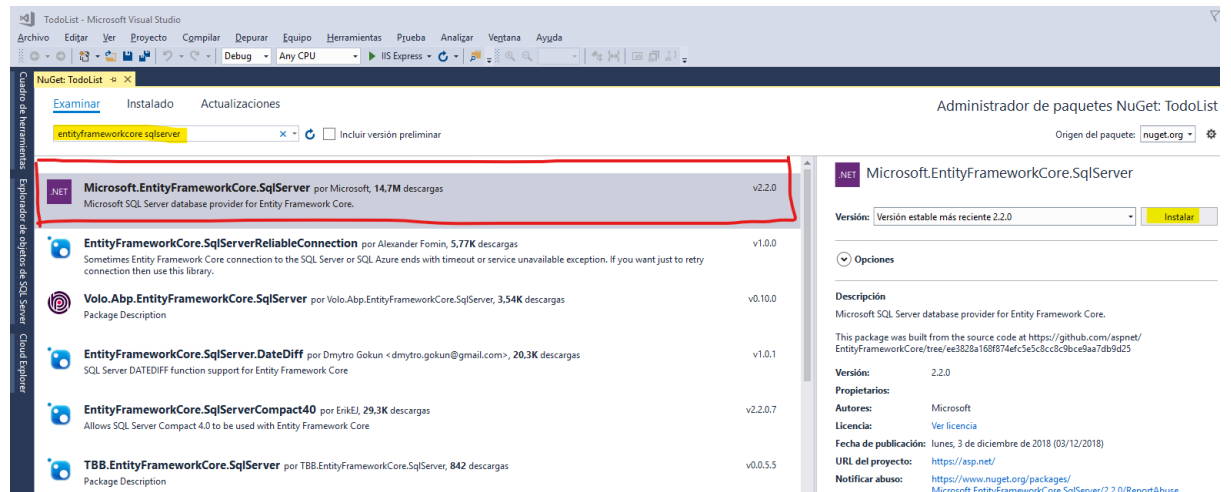


Ejemplo: Instalación de Entity Framework Core en el proyecto.

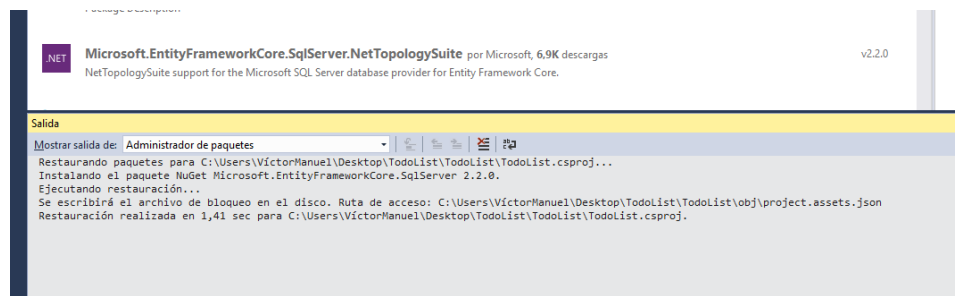
1. Abrimos el gestor de paquetes NuGet haciendo click derecho en nuestro proyecto:



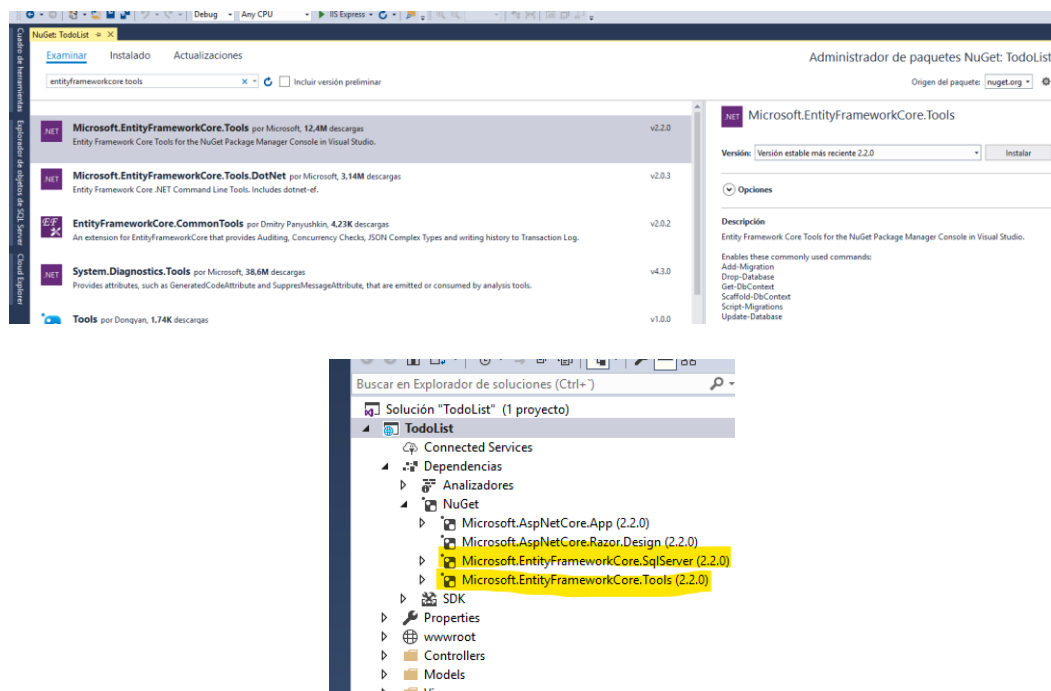
- En el cuadro de búsqueda buscaremos el proveedor de Entity Framework Core para SQL Server, escribiendo entityframeworkcore:



- Una vez encontrado, lo seleccionamos y pulsamos **Instalar**:



- Haremos lo mismo con el paquete Entity Framework Core Tools:



Entidades y contexto de datos.

Para trabajar con Entity Framework Core necesitamos definir un modelo de datos con el que trabajaremos y que será lo que se almacene en el SGBD correspondiente. Este modelo de datos está compuesto por las entidades del modelo y sus relaciones, y el contexto de datos.

Las entidades son simples clases llamadas POCO (Plain Old Class Objects), y aunque el nombre pueda parecer extraño, se refiere a que las entidades se definen como simples objetos normales de C#, cuyas propiedades serán los campos que se almacenen en las diferentes tablas de la BD.

Por convención, si una de nuestras entidades del modelo contiene un campo llamado *Id*, o “Nombre de clase” + *Id*, se considerará que es una clave primaria cuando se cree la tabla para almacenar dicha entidad en la BD. Por ejemplo, si tenemos una clase llamada *Estudiante*, se tomaría como clave primaria una propiedad llamada *Id* o una llamada *EstudiantId*.

Nuestra clase, además de la clave primaria puede contener otras propiedades, que pueden ser de dos tipos:

- **Propiedades de valor:** Contienen valores que serán almacenados en la BD.
- **Propiedades de navegación:** Representan relaciones entre entidades.

Además de definir las entidades que formarán nuestro modelo, tenemos que crear un elemento fundamental, llamada contexto de datos. El contexto de datos es una clase que hereda de la clase base `Microsoft.EntityFrameworkCore.DbContext`, y representa una sesión de conexión con la base de datos, y permite configurar las entidades que formarán parte del mismo y su mapeo a las correspondientes tablas de la BD.

En esta clase especificaremos cuáles son las entidades que vamos a almacenar en la BD, incluyendo las mismas como colecciones de tipo `DbSet`. A continuación se muestra un ejemplo de contexto de datos para un modelo formado por objetos de tipo `Blog`:

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

    public DbSet<Blog> Blogs { get; set; }
}
```

Este contexto de datos crearía una base de datos con una única tabla denominada `Blogs`, que almacenaría objetos de tipo `Blog`, siendo los campos de la tabla las propiedades definidas en la clase `Blog`.

Una vez más, Entity Framework Core utiliza una serie de convenciones a la hora de mapear las entidades a objetos de la base de datos, si bien estas convenciones pueden ser modificadas y podremos especificar diferentes opciones de mapeo de nuestras entidades. Estas convenciones son las siguientes:

- Los nombres de tabla serán la forma en plural del nombre de nuestras clases. Por ejemplo, para la entidad `Blog` del ejemplo, se crearía una tabla en la BD llamada `Blogs`. Hay que tener cuidado con esto ya que el plural se forma añadiendo siempre una ‘s’ al final, lo que no es siempre adecuado. Para estos casos, podemos especificar el nombre de la tabla de forma explícita.

- Los nombres de los campos de la tabla serán los mismos que los de cada propiedad que se vaya a almacenar.
- La clave primaria de nuestra tabla será una propiedad llamada Id o [NombreDeEntidad]Id, como ya se explicó anteriormente.
- Las propiedades que hagan referencia a otra entidad del modelo, o a una colección de ellas, se denominan propiedades de navegación, y establecen relaciones entre tablas, y claves externas.

Para especificar de forma explícita las opciones de mapeo tenemos dos mecanismos:

- **Anotaciones (Data Annotations):** Permite definir los tipos de datos a los que se mapearán nuestras propiedades mediante atributos en la propia entidad.
- **Fluent API:** Especificamos los detalles del mapeo mediante código en el contexto de datos, concretamente en el método *OnModelCreating()*.

Cada opción tiene sus ventajas e inconvenientes. En principio las anotaciones son mucho más simples y claras, y será la opción que utilizemos en esta unidad. Sin embargo, tienen una cantidad importante de detractores, ya que los más puristas dicen que si decoramos nuestras clases con atributos que especifiquen datos concretos sobre el mapeo a una BD, estamos haciendo que nuestro modelo de datos deje de ser portable, al violar una de las normas básicas de separación de responsabilidades, ya que el modelo de datos debe ser independiente del almacén de datos que se utilice. Veremos más sobre esto en unidades posteriores.

En el siguiente enlace podemos ver más información sobre como configurar el mapeo de nuestro modelo de datos:

<https://docs.microsoft.com/en-us/ef/core/modeling>

Una vez que tenemos nuestro modelo de datos definido, y el contexto, que establece el puente entre nuestras entidades y la base de datos en la que se almacenarán, necesitamos un último elemento, que especifica la BD con la que trabajaremos. Este elemento se llama cadena de conexión, y especifica, entre otros, el servidor de BD utilizado, el nombre de la BD, el usuario, etc..

La cadena de conexión es utilizada por el contexto para poder establecer conexiones con la base de datos y realizar las consultas necesarias.

Para más información sobre las cadenas de conexión podemos consultar en el enlace a continuación:

<https://docs.microsoft.com/en-us/ef/core/miscellaneous/connection-strings>

Ejemplo: Preparar el modelo y contexto al proyecto.

Vamos a ver en la práctica cómo crear todos los elementos necesarios, utilizando para ello nuestro proyecto de lista de tareas.

1. Lo primero que necesitamos son las entidades de nuestro modelo de datos. En este caso ya habíamos definido nuestra clase Tarea, que es la única del modelo. De este modo el primer paso ya está realizado.

2. A continuación vamos a crear el contexto de datos, para ello comenzamos por crear una carpeta Data, y en ella creamos una nueva clase a la que llamaremos `TodoListContext`. Por convención los contextos de datos serán clases que hereden de `DbContext`, y cuyo nombre termine en `Context`, de este modo es sencillo identificarlos. Lo hemos situado en la carpeta Data porque se trata de código específico de acceso a un almacén de datos, por lo que no forma parte del modelo en sí mismo, si no que es una clase que nos permitirá trabajar con las entidades de nuestro modelo sobre una base de datos:

```
using Microsoft.EntityFrameworkCore;
using TodoList.Models;

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

        public DbSet<Tarea> Tareas { get; set; }
    }
}
```

Hemos creado nuestra clase con un constructor que permite pasarle los parámetros de conexión, como veremos más adelante. Hemos definido un único conjunto de entidades de tipo `Tarea`, a la que llamamos `Tareas`, de esta forma, nuestra base de datos contendrá una única tabla que almacenará tareas, y podremos acceder a los datos de dicha tabla mediante el contexto, a través de su propiedad `Tareas`.

3. Una vez definido el contexto de datos, debemos especificar una cadena de conexión, que le permita establecer una conexión con una base de datos concreta. En una aplicación ASP.NET esta cadena se almacena en el fichero de configuración, llamado `appsettings.json`. Este fichero almacena diferentes parámetros de configuración en notación Json. Lo que debemos hacer es añadir un elemento `ConnectionStrings`, y especificar la cadena o cadenas necesarias. En nuestro caso definimos una cadena de conexión para una base de datos SQLServer LocalDB, cuyo nombre es `TodoList`. Hemos nombrado esta cadena de conexión como `TodoListContext`, de modo que utilizaremos más adelante este nombre para recuperar dicha cadena:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "TodoListContext":
      "Server=(localdb)\\mssqllocaldb;Database=TodoList;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

4. Ya tenemos todos nuestros elementos, lo único que nos queda es añadir el siguiente código en nuestra clase `Startup.cs`, para que inicialice el contexto de datos y haga que esté disponible para nuestra aplicación:

```
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<TodoListContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("TodoListContext")));
}
```

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*.

Migraciones.

Entity Framework Core nos permite crear y modificar la base de datos y su esquema desde nuestro código, para hacer más sencillo el trabajo con la misma. Para ello, mantiene la estructura de la BD, y los cambios que se vayan realizando, en clases generadas por las Entity Framework Core Tools. Una vez hemos definido nuestro modelo y la clase de contexto, podremos generar estas clases, siempre y cuando hayamos instalado previamente en nuestro proyecto el paquete Nuget *Microsoft.EntityFrameworkCore.Tools*.

Llamamos migración al proceso de guardar un registro de todos los cambios de nuestra base de datos, de este modo nos aseguramos de que nuestra base de datos evoluciona junto con los cambios en nuestra aplicación. Para ello, cada vez que haya un cambio en nuestro modelo, añadiremos una nueva migración al proyecto.

Especialmente importante es la migración inicial, que guardará el esquema inicial de nuestra BD, y nos permitirá crear la misma desde nuestro proyecto, sin necesidad de tener que trabajar directamente con ningún SGBD.

Cada vez que añadamos una nueva migración, debida a algún cambio en el modelo de datos, tendremos también la posibilidad de modificar la base de datos desde nuestro proyecto, para que los cambios del modelo se vean reflejados en la BD.

Para más información acerca de las migraciones de Entity Framework Core, podréis consultar el siguiente enlace:

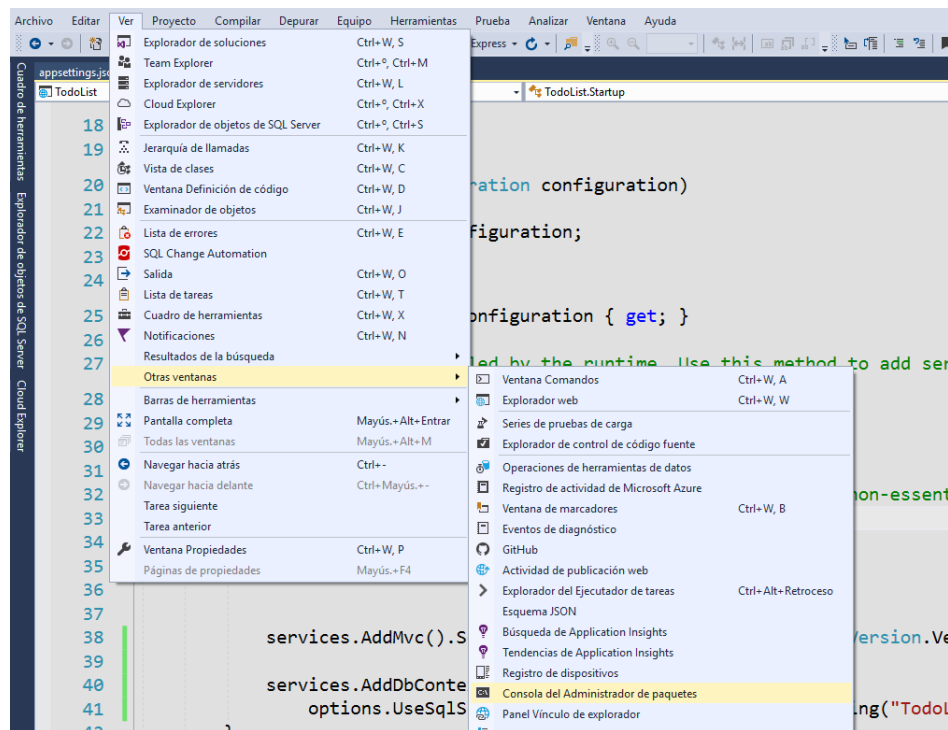
<https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations>

Ejemplo: Creación de la migración inicial y de la base de datos.

En este momento, tenemos todos los elementos necesarios en nuestro proyecto para trabajar con Entity Framework Core, que como recordamos son las clases que representan a las entidades del modelo, y que guardaremos en la carpeta *Models*, el contexto de datos, que es una clase que hereda de *DbContext* y se guarda en la carpeta *Data*, y por último la cadena de conexión que especifica la base de datos que utilizaremos, almacenada en el fichero de configuración de la aplicación, *appsettings.json*.

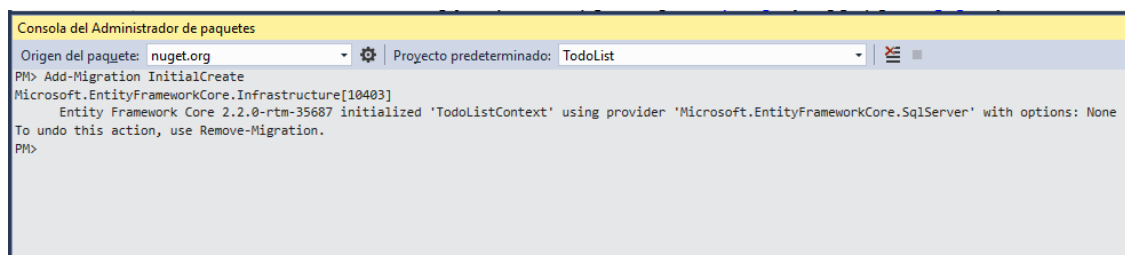
Lo que haremos ahora es crear una migración que capture el estado inicial del modelo de datos y crearemos la BD a partir de esta migración.

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



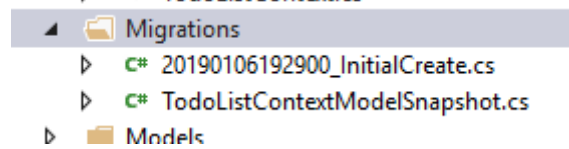
2. Desde la consola, escribimos el siguiente comando para crear nuestra migración inicial:

Add-Migration InitialCreate



Si todo está correcto obtendremos un mensaje similar al anterior. El valor *InitialCreate* es simplemente el nombre que damos a nuestra migración para identificarla y podría ser cualquier otro, aunque es el nombre que se da habitualmente a la migración inicial.

- Podemos comprobar que en nuestro proyecto se ha creado una carpeta Migrations, que contiene el estado actual del modelo de datos y una clase que contiene el código necesario para crear la base de datos conforme al estado inicial del modelo de datos.



Si observamos el código podremos ver que contiene dos métodos, Up y Down. El método Up se ejecuta cuando ejecutamos una actualización de la base de datos desde un estado anterior, mientras que el método Down se ejecutaría si regresáramos a un estado anterior de la base de datos. Para nuestro ejemplo tendrían código para crear nuestra tabla de tareas y eliminar la misma, respectivamente:

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.CreateTable(
        name: "Tareas",
        columns: table => new
        {
            Id = table.Column<int>(nullable: false)
                .Annotation("SqlServer:ValueGenerationStrategy",
                    SqlServerValueGenerationStrategy.IdentityColumn),
            Descripcion = table.Column<string>(nullable: true),
            Realizada = table.Column<bool>(nullable: false)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_Tareas", x => x.Id);
        });
}

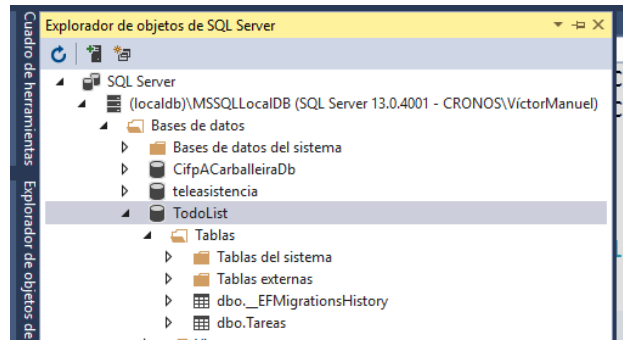
protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropTable(
        name: "Tareas");
}
```

- En este momento ya tenemos todos los elementos necesarios para crear nuestra base de datos. Para ello, vamos a pedir que la base de datos sea actualizada a las migraciones disponibles, que en nuestro caso es sólo la migración de creación inicial. Para ello utilizamos el siguiente comando:

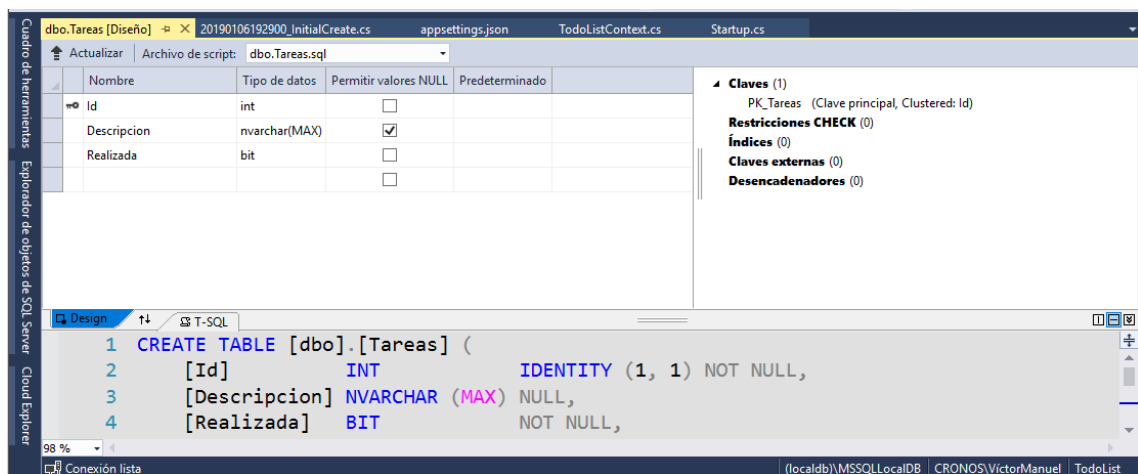
Update-Database

Si todo es correcto se mostrarán las acciones que se han llevado a cabo para la actualización de la BD y el mensaje *Done*.

- Vamos a comprobar ahora cómo la BD ha sido creada correctamente consultando de nuevo el explorador de objetos de SQL Server. En el nodo SQL Server, encontraremos nuestra base de datos, de nombre TodoList, y si exploramos la misma podemos comprobar que contiene 2 tablas, una tabla llamada Tareas, tal y como esperábamos, que almacenará nuestras tareas, y otra llamada _EFMigrationsHistory, que servirá para mantener las migraciones aplicadas sobre nuestra BD, para que cada vez que utilicemos el comando Update-Database, se sepa qué migraciones se han de aplicar.



- Si hacemos doble click en la tabla de tareas podemos ver su esquema, así como el código SQL utilizado para su creación:



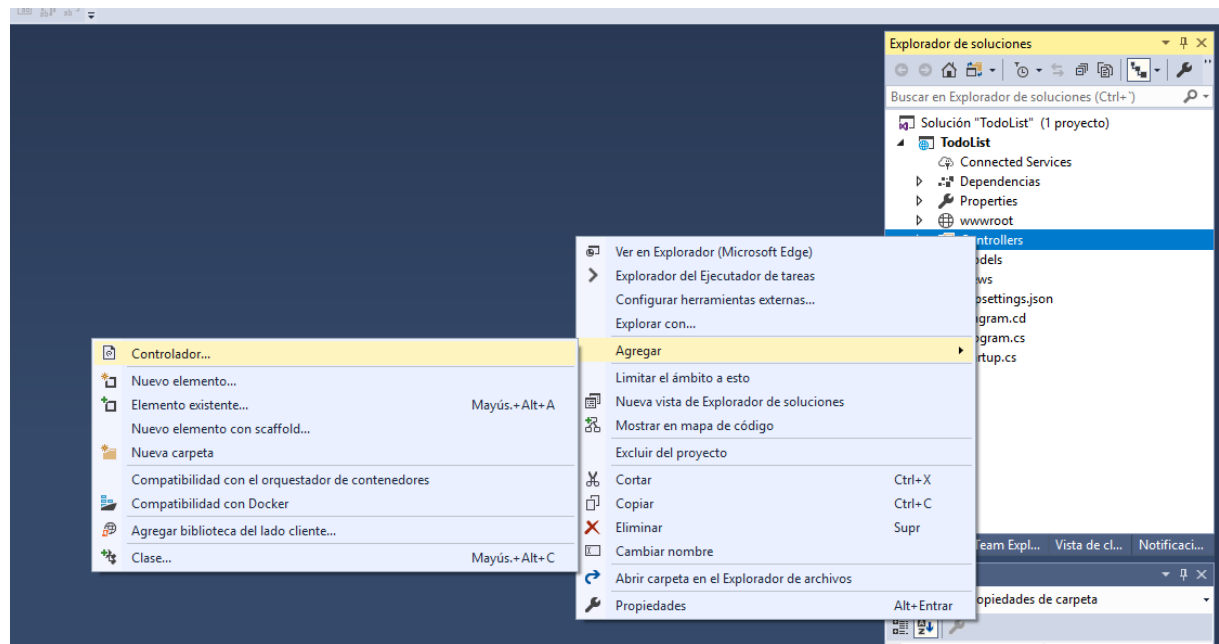
- Llegados a este punto hemos sido capaces de preparar nuestro modelo de datos para trabajar con Entity Framework Core, y pudimos crear nuestra base de datos sin necesidad de utilizar ni una sola sentencia SQL.

Ejemplo: Creación del controlador y las vistas.

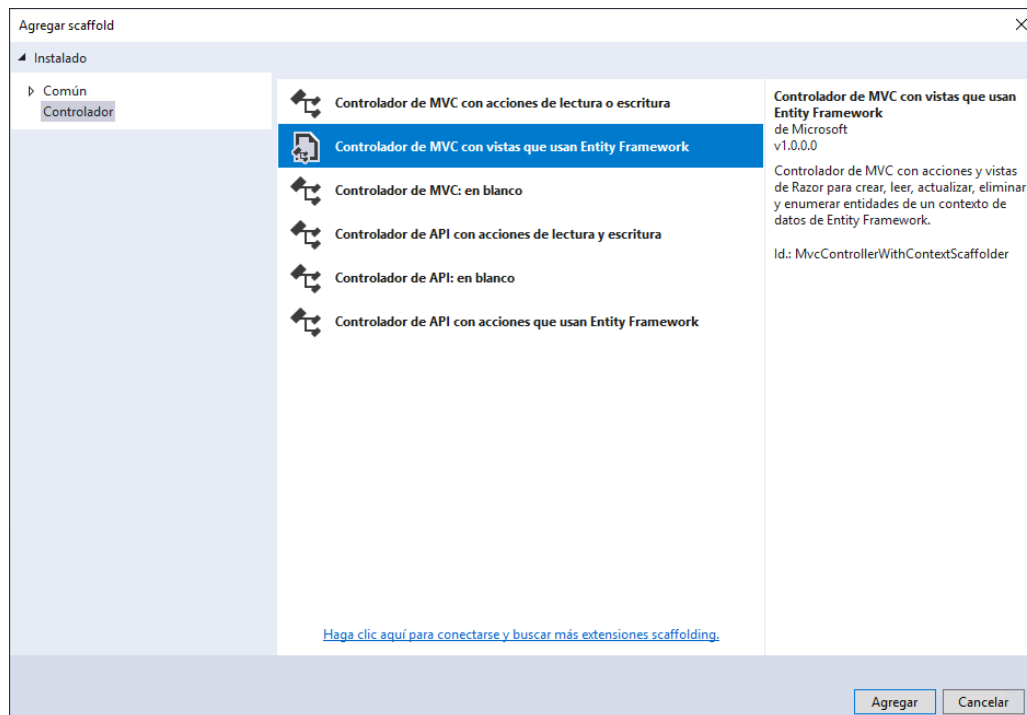
Con todo el modelo preparado y la base de datos creada, llega el momento de trabajar en nuestra aplicación web, creando un controlador para atender a las peticiones sobre las tareas, además de las vistas apropiadas.

En esta ocasión vamos a ver cómo utilizar uno de los asistentes de Visual Studio para la creación de un controlador que proporcionará métodos de acción para trabajar con una entidad de datos, utilizando Entity Framework Core, además de generar las vistas correspondientes.

- 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. A continuación, seleccionaremos en el primer desplegable la entidad para la que crearemos controlador y vistas, en nuestro caso *Tarea*. En el segundo desplegable seleccionamos el contexto de datos que será en nuestra aplicación *ToDoListContext*. Debemos escribir un nombre para nuestro controlador, como se encargará de gestionar las peticiones sobre operaciones relacionadas con tareas, el nombre más apropiado será *TareasController*. Finalmente pulsamos en **Agregar**.

Agregar Controlador de MVC con vistas que usan Entity Framework

Clase de modelo: **Tarea (TodoList.Models)**

Clase de contexto de datos: **TodoListContext (TodoList.Data)** +

Vistas:

☒ Generar vistas

☒ Hacer referencia a bibliotecas de scripts

☒ Usar página de diseño:

...

(Dejar en blanco si se define en un archivo _viewstart de Razor)

Nombre de controlador: **TareasController**

Agregar Cancelar

- Una vez terminado el asistente podemos comprobar como en la carpeta *Controllers* se encuentra nuestro nuevo controlador *TareasController*, mientras que en la carpeta *Views* tenemos una serie de vistas para dicho controlador.
- 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.
- Lo siguiente será editar nuestro *_Layout.cshtml*. Lo primero que haremos es editar la cabecera, cambiando el texto del primer enlace a "Lista de tareas" 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="Tareas"
        asp-action="Index">Lista de tareas</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>
```

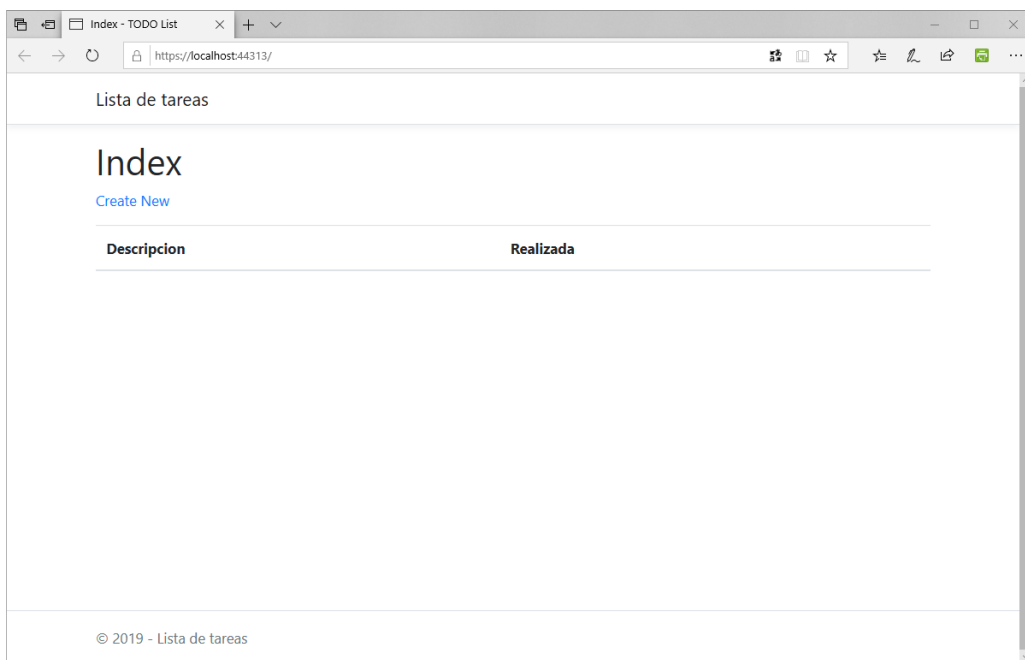
En el pie eliminaremos también el enlace a la página de privacidad:

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


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=Tareas}/{action=Index}/{id?}");
});
```

8. Si ejecutamos en este momento nuestra aplicación, obtendremos una página con una lista de tareas vacía, con un enlace que permite añadir una nueva tarea, tal y como se muestra en la imagen:



Ejemplo: Creación de tareas.

A pesar de que las plantillas generadas automáticamente resultan de gran ayuda, lo más habitual es que no se ajusten completamente a nuestras necesidades, por lo que tendremos que realizar una serie de modificaciones para conseguir que nuestra aplicación se comporte como queremos.

Vamos a comenzar preparando la parte de la aplicación que nos permite la creación de nuevas tareas. Esta funcionalidad está implementada en los métodos *Create* de nuestro controlador y en la vista con el mismo nombre.

1. Comenzamos el controlador. Si observamos, el constructor recibe un objeto de tipo *ToDoListContext*, es decir, el contexto de datos de nuestra aplicación. Este objeto es suministrado por ASP.NET en el momento de crear el controlador, mediante un mecanismo denominado inyección de dependencias, que será explicado en la siguiente unidad. Los diferentes métodos del controlador utilizarán este contexto para interactuar con la base de datos. A continuación analizamos el código de los métodos *Create*:

```
// GET: Tareas/Create
public IActionResult Create()
{
    return View();
}

// POST: Tareas/Create
// 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>
    Create([Bind("Id,Descripcion,Realizada")] Tarea tarea)
{
    if (ModelState.IsValid)
    {
        _context.Add(tarea);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    return View(tarea);
}
```

El primer método es muy sencillo, recibe la petición get, que es cuando se solicita mostrar el formulario de creación de tareas. El controlador simplemente devuelve la vista correspondiente para que se muestre la página que contiene el formulario de introducción de datos.

El segundo método tiene muchas más cosas que analizar. En primer lugar vemos que responde a la petición post, que tiene lugar cuando el usuario ha introducido una nueva tarea y envía el formulario.

Lo primero que nos puede llamar la atención es el atributo [ValidateAntiForgeryToken]. Éste se utiliza para evitar ataques CSRF y es recomendable incluir dicho atributo en los métodos de acción que reciben datos de un formulario. Para más información se puede consultar el siguiente enlace:

<https://docs.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-2.2>

El método está definido como asíncrono (async) por lo que en lugar de devolver un objeto de tipo IActionResult devolverá una Task<IActionResult>. No vamos a entrar ahora en detalles sobre métodos asíncronos en C#, pero de un modo sencillo comentaremos simplemente que los métodos asíncronos pueden mejorar el rendimiento de nuestra aplicación porque en lugar de esperar a terminar su ejecución, quedan a la espera de que la tarea que realizan, liberando recursos del servidor, y en el momento que los datos están disponibles, reanudan su ejecución.

El siguiente elemento que conviene explicar es el atributo Bind que encontramos antes del objeto Tarea que recibimos como parámetro. Este atributo permite detallar las propiedades de un objeto que recibiremos de un formulario. Si no recibimos todas las propiedades que contiene un objeto del formulario donde el usuario introduce los datos, es conveniente especificar cuáles de estas propiedades debemos leer, para evitar que un usuario malicioso fuerce la introducción de datos no deseados. Podemos consultar más información en el enlace que se proporciona en los comentarios del método.

En nuestro ejemplo, si analizamos la situación, cuando un usuario crea una nueva tarea, lo lógico será que el único dato que introduzca es la descripción de la misma, ya que el id no se obtiene hasta que el objeto es introducido en la BD, y una tarea recién creada, por defecto tendría que

estar sin realizar. Por lo tanto vamos a modificar este código para recibir del formulario únicamente la propiedad Descripción.

```
public async Task<IActionResult> Create([Bind("Descripcion")] Tarea tarea)
```

El código que encontramos a continuación es la parte más importante del método. En primer lugar comprobamos si el modelo es válido, es decir, cumple las posibles restricciones que hayamos especificado a las propiedades del mismo. En caso de ser correcto, añadimos nuestro objeto Tarea al contexto. Este método es capaz de detectar el tipo de entidad de la que se trata, en caso de que el modelo de datos conste de más de una, y lo añade a la colección correspondiente.

Por último, pedimos al contexto que guarde los cambios, que en este caso consisten en añadir una nueva tarea. Esto hará que el contexto realice las consultas necesarias para añadir nuestra tarea a la BD. A continuación, una vez insertado, redirigiremos el control de nuestra aplicación al método de acción *Index*, que volverá a mostrar la lista de tareas.

En caso de que el modelo de datos no sea válido, se volvería a mostrar el formulario, donde se indicarían los errores de validación, tal y como ya vimos en la unidad anterior.

2. A continuación vamos a pasar a la vista, donde tendremos que hacer algunos pequeños ajustes. En primer lugar cambiaremos el título y el texto que se muestra encima del formulario para dejarlo de la siguiente forma:

```
@{
    ViewData["Title"] = "Nueva tarea";
}

<h4>Nueva tarea</h4>
<hr />
```

En el propio formulario, vemos que la plantilla ha creado controles de edición para cada propiedad de la tarea, excepto el Id, siendo este el comportamiento habitual. En nuestro caso no tiene sentido editar la propiedad Realizada en una nueva tarea, por lo que la eliminaremos. También cambiaremos algunos textos que por defecto están en inglés y el control de descripción para que permita introducción de texto multilínea.

```
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Descripcion" class="control-label">
                    Descripción</label>
                <textarea asp-for="Descripcion" class="form-control" rows="4">
                </textarea>
                <span asp-validation-for="Descripcion" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Crear" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>
```

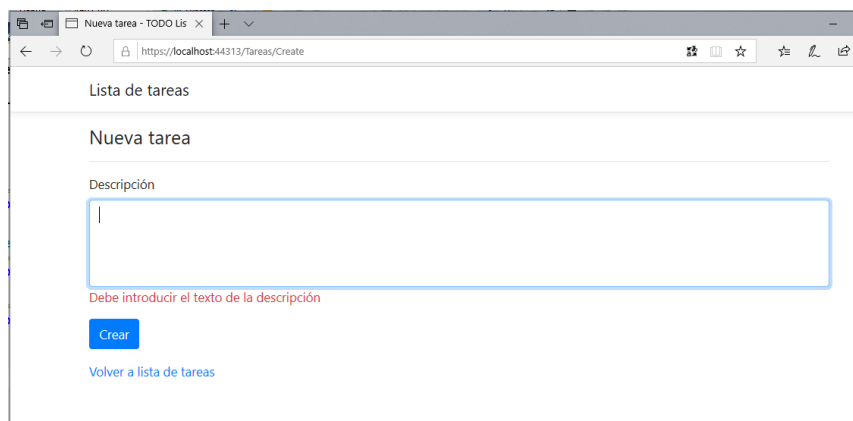
3. Nuestro formulario está mucho mejor ahora, pero nos falta un pequeño detalle. A pesar de que tanto el controlador como la vista incluyen código capaz de comprobar la validez de los datos introducidos, si introducimos una tarea sin texto, esta sería aceptada. Para evitar que esto siga siendo así debemos añadir una anotación en el modelo para marcar la propiedad como requerida:

```
public class Tarea
{
    public int Id { get; set; }

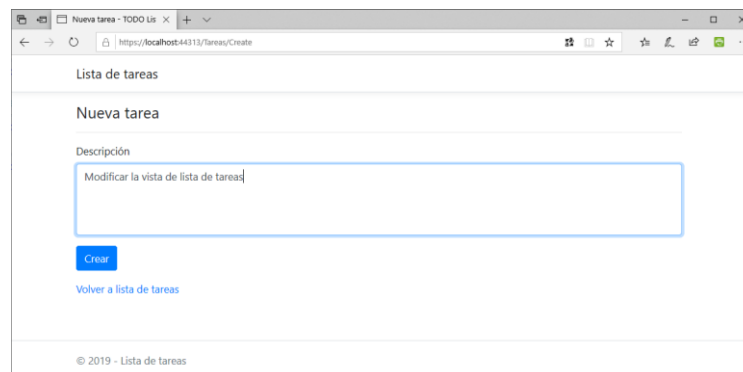
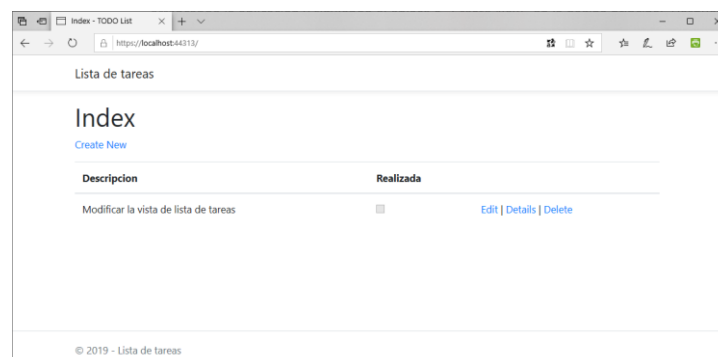
    [Required(ErrorMessage = "Debe introducir el texto de la descripción")]
    public string Descripcion { get; set; }

    public bool Realizada { get; set; }
}
```

En este momento si intentamos introducir una tarea vacía obtendríamos el mensaje de error correspondiente, incluyendo también validación del lado cliente.

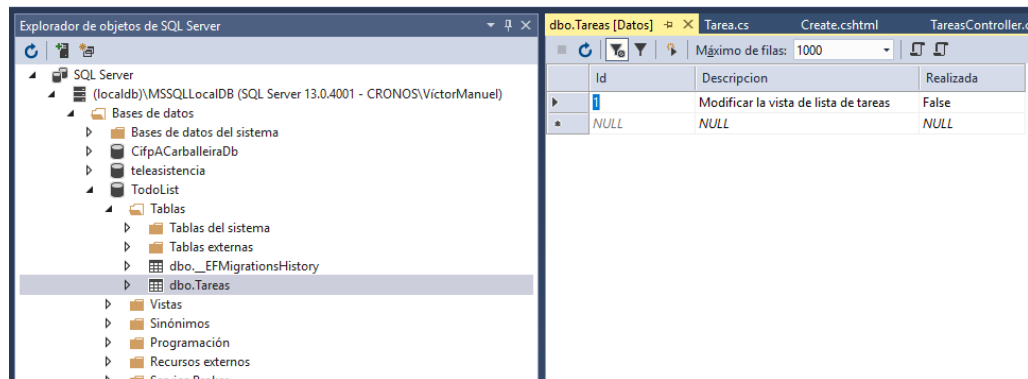


- Nuestra aplicación ya está preparada para permitir añadir nuevas tareas. Vamos a probarla arrancando la aplicación y siguiendo el enlace a “Create New” en la página principal. Después de introducir nuestra tarea volveremos a la página principal, donde aparecerá en la lista, junto con el resto de tareas.

Descripcion	Realizada	
Modificar la vista de lista de tareas	<input type="checkbox"/>	Edit Details Delete

- Ahora que hemos introducido nuestra primera tarea, vamos de nuevo a comprobar nuestra base de datos, donde exploraremos el contenido de la tabla Tareas.



Id	Descripción	Realizada
NULL	NULL	NULL

Podemos comprobar como hemos introducido nuestra tarea en la tabla correspondiente de la base de datos sin una sola sentencia SQL, con dos sencillas líneas de código, simplemente añadiendo un objeto Tarea al contexto de datos y guardando los cambios.

Ejemplo: Marcar una tarea como realizada.

Como ya hemos indicado en varias ocasiones, tratamos de mantener este primer ejemplo lo más sencillo posible, ya que el objetivo es el de familiarizarnos con Entity Framework Core, por lo que no conviene complicarse en demasiados detalles.

En nuestra aplicación, una vez creada una tarea, no permitiremos eliminarla ni modificarla. La única acción permitida será marcarla como realizada, de modo que dejaría de ser una lista pendiente, y para el usuario sería como si dejara de existir, aunque seguirá estando en nuestra base de datos. De este modo estamos utilizando un borrado lógico de las tareas, basándonos en el estado de la propiedad Realizada.

- El procedimiento que vamos a seguir para marcar una tarea como realizada es similar al que se utiliza en la plantilla de eliminar, mostrando primero una página de confirmación, y en caso de que el usuario acepte, se procederá finalmente a marcar la tarea como realizada. Por esta razón, vamos a partir del código de eliminación en lugar de partir de cero.

Comenzamos analizando el método Delete que atiende la petición get:

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

    var tarea = await _context.Tareas
        .FirstOrDefaultAsync(m => m.Id == id);
    if (tarea == null)
    {
        return NotFound();
    }

    return View(tarea);
}
```

Comenzamos por el parámetro `Id` que recibe el método. Nuestro mapeo de URLs por defecto incluye un controlador, una acción, y un parámetro `Id` opcional, de modo que si introducimos una URL como <https://xxx.xxx.xxx.xxx/Tareas/Delete/5>, estaríamos llamando al método y pasando un valor de 5 como parámetro `Id`, lo que es mucho más claro que tener que definir dicho parámetro dentro de la URL. El símbolo `?` tras el tipo de datos indica lo que en C# se llama un tipo “nullable”, es decir, que aunque esperamos un parámetro entero, podría darse que el valor sea nulo, esto ocurriría con una petición que no incluyera el número correspondiente en la URL.

Si no especificamos el `Id` de la tarea, lógicamente esta no se encontraría, por lo que se devuelve un resultado de `NotFound()`. En caso de haber recibido un `Id`, buscamos la tarea con dicho identificador en la BD, y de nuevo, si no se encuentra devolvemos un error. Sólo en caso de que se haya pasado un `Id` correcto para una tarea mostraremos la vista que nos pedirá la confirmación para marcarla como realizada. Además pasaremos los datos de dicha tarea a la vista para que pueda mostrar la descripción al usuario.

El código más importante del método es el que busca la tarea por su `Id`, lo que tiene lugar en la siguiente línea:

```
var tarea = await _context.Tareas.FirstOrDefaultAsync(m => m.Id == id);
```

Lo que estamos haciendo es, de nuevo, realizar nuestra consulta utilizando el contexto de datos. Dentro del contexto, buscamos en la colección de tareas, y pedimos a dicha colección la primera aparición de un elemento tal que la propiedad `Id` de dicho elemento sea igual al `id` que ha recibido el parámetro. Esta consulta se hace de forma declarativa, indicando que es lo que debemos buscar, en este caso un elemento con un `id` determinado, y el contexto se encarga de generar la consulta SQL necesaria para obtener ese objeto.

Podemos reutilizar todo el código del método ya que queremos hacer lo mismo, buscar la tarea con un `id` determinado, y una vez encontrada, mostrar una vista que permita confirmar que la hemos realizado. Por lo tanto, lo único que haremos es cambiar el nombre del método, de `Delete` a `Do`.

```
// GET: Tareas/Do/5
public async Task<IActionResult> Do(int? id)
```

- Al cambiar el nombre del método, deberemos también cambiar el de la vista correspondiente, también por `Do.cshtml`. En el código realizaremos también algunos cambios:

```
@model TodoList.Models.Tarea

@{
    ViewData["Title"] = "Realizar tarea";
}

<h2>Realizar tarea</h2>

<h4>¿Está seguro de marcar la tarea como realizada?</h4>
<div>
    <hr />
    <dl class="row">
        <dt class = "col-sm-2">
            Descripción
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Descripcion)
```

```

        </dd>
    </dl>

    <form asp-action="Aceptar">
        <input type="hidden" asp-for="Id" />
        <input type="submit" value="Realizar tarea" class="btn btn-danger" /> |
        <a asp-action="Index">Volver a la lista de tareas</a>
    </form>
</div>

```

Los cambios se limitan básicamente a modificar los textos, además de eliminar la información de la propiedad Realizada, que incluye la plantilla por defecto.

3. Ya tenemos la primera parte, pero si el usuario decide confirmar la realización de la tarea, estaremos llamando de nuevo al mismo método de acción, esta vez utilizando una petición post. Por lo tanto deberemos cambiar también el nombre del método Delete que recibe dicha petición, de la siguiente manera:

```

// POST: Tareas/Do/5
[HttpPost, ActionName("Do")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DoConfirmed(int id)
{
    var tarea = await _context.Tareas.FindAsync(id);
    _context.Tareas.Remove(tarea);
    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 Do, tal y como se explicó en la unidad 4.

Es interesante estudiar el código del método, que se encarga de eliminar una tarea, para ver cómo se hace en Entity Framework Core, aunque obviamente no es lo que deseamos por lo que lo eliminaremos. La idea es sencilla, en primer lugar obtenemos una tarea cuyo id sea el pasado. En este caso sabemos que el id será correcto ya que el método post sólo se lanza desde el formulario de confirmación, para el que el id ya había sido comprobado. Una vez obtenida la tarea, pedimos al contexto que la elimine, y por último guardamos los cambios para que se haga definitivo, enviando la consulta a la BD.

Lo que necesitamos para nuestra aplicación es modificar el valor de la propiedad Realizada, poniéndola a true, y guardar los cambios de la tarea en la BD. Para ello veremos que, gracias a Entity Framework Core, de nuevo es muy sencillo.

```

var tarea = await _context.Tareas.FindAsync(id);
tarea.Realizada = true;
await _context.SaveChangesAsync();

```

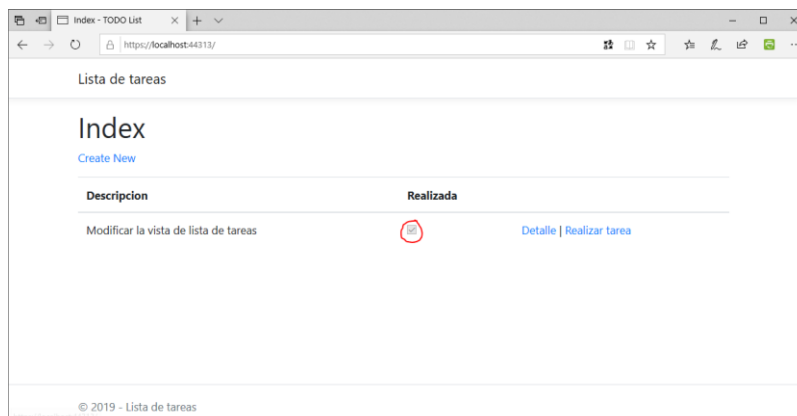
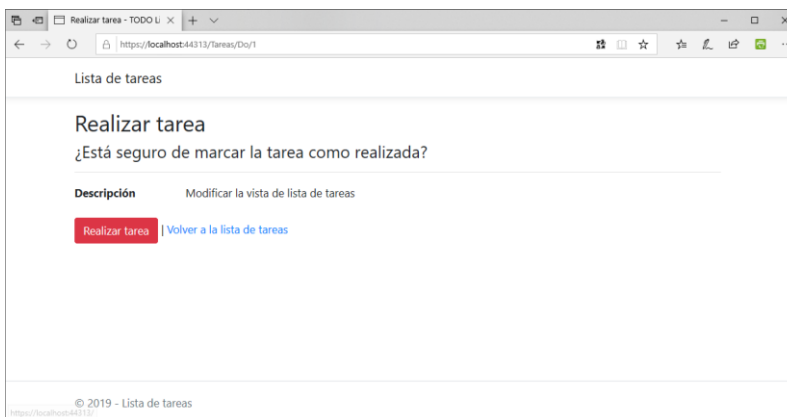
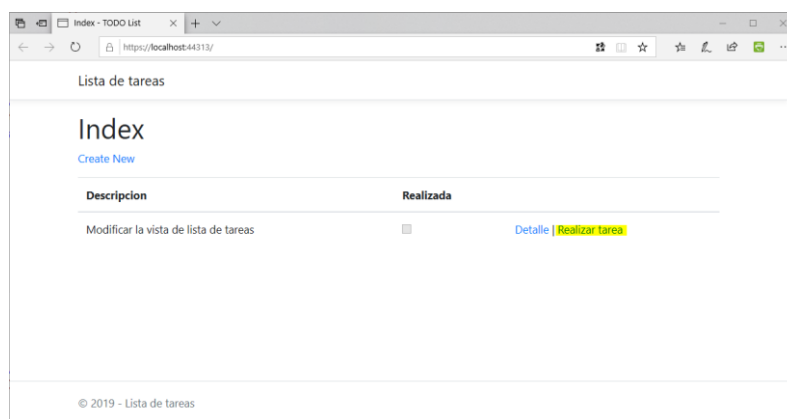
El procedimiento es muy simple, empezamos obteniendo la tarea cuyo id conocemos. Una vez tenemos la tarea, ponemos el valor de la propiedad Realizada a true, y por último persistimos los cambios en la BD. De nuevo, hemos hecho una consulta de modificación, sin necesidad de escribir código SQL gracias a nuestro contexto de datos.

- Para poder probar lo que hemos hecho, debemos modificar los enlaces de la lista de tareas, que en estos momentos apuntan a las acciones Edit y Delete, para que en su lugar apunten a la acción Do, y aprovechamos para cambiar también los textos de los enlaces:

```
<td>
  <a asp-action="Details" asp-route-id="@item.Id">Detalle</a> |
  <a asp-action="Do" asp-route-id="@item.Id">Realizar</a>
</td>
```

- Antes de ejecutar la aplicación para intentar marcar una tarea como realizada, eliminaremos los métodos de acción Edit, que no utilizaremos, así como su vista.

- Ahora, por fin, vamos a marcar nuestra tarea como realizada:



Ejemplo: Detalle de la tarea.

La vista de detalle nos permite ver todo el texto de la descripción de la tarea. Se trata de la parte más simple de la aplicación, y no tendremos que modificar demasiado en esta ocasión.

1. El método de acción que permite ver el detalle de una tarea es muy sencillo, básicamente consiste en buscar una tarea por su id, y mostrar la misma en una vista de detalle. No necesitamos hacer ningún cambio en el método, así que nos limitaremos a cambiar algunos textos de la vista, además de eliminar de la misma la visualización de la propiedad Realizada:

```
@{
    ViewData["Title"] = "Detalle de tarea";
}

<div>
    <h4>Detalle de tarea</h4>
    <hr />
    <dl class="row">
        <dt class = "col-sm-2">
            Descripción
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Descripcion)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Index">Volver a lista de tareas</a>
</div>
```



Ejemplo: La lista de tareas.

Nuestra aplicación está casi terminada, sólo nos queda realizar una serie de cambios en la vista principal, donde se muestra la lista de tareas.

1. El método de acción Index es el que responde a la ruta por defecto y es el encargado de obtener todas las tareas de la tabla de la BD, y mostrarlas en una tabla.

```
// GET: Tareas
public async Task<IActionResult> Index()
{
    return View(await _context.Tareas.ToListAsync());
}
```

En esta ocasión, el código es todavía más simple si cabe. A través del contexto de datos, accedemos a la colección Tareas, que contiene todas las tareas de la tabla de la BD, y la devuelve como una lista a la vista para que las muestre al usuario.

Tal y como hemos planteado nuestra aplicación, siendo un recordatorio de tareas pendientes, no tiene ningún sentido mostrar las tareas ya realizadas. Por esta razón, debemos modificar nuestra consulta, de forma que obtengamos solamente las tareas cuyo valor de Realizada es false.

```
return View(await _context.Tareas.Where(t => t.Realizada == false).ToListAsync());
```

Lo que estamos haciendo es utilizar una clausula Where, que nos permite especificar una condición que deben cumplir los elementos que obtendremos. Aquí estamos obteniendo todas las tareas para las que se cumple, que dada la tarea, su propiedad Realizada tendrá el valor false.

Entity Framework Core utiliza un lenguaje de consultas llamado LINQ, se puede consultar más información en el siguiente enlace:

<https://docs.microsoft.com/es-es/ef/core/querying>

- Una vez que hemos modificado la consulta, sólo nos queda realizar algunos cambios en la vista, que consistirán de nuevo en traducir algunos textos, y eliminar de la tabla la columna que muestra si la tarea está realizada, ya que al mostrar sólo tareas no realizadas, este campo no tiene sentido.

```
@model IEnumerable<TodoList.Models.Tarea>

@{
    ViewData["Title"] = "Lista de tareas";
}

<h2>Tareas pendientes</h2>

<p>
    <a asp-action="Create">Nueva tarea</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                Descripción
            </th>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model) {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Descripcion)
                </td>
                <td>
                    <a asp-action="Details" asp-route-id="@item.Id">Detalle</a> |
                    <a asp-action="Do" asp-route-id="@item.Id">Realizar tarea</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

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