# CRUD básico con Entity Framework

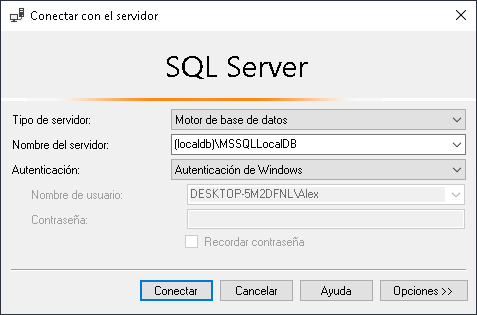
Antes de tener acceso a datos en nuestra aplicación vamos a conectarnos y a definir una nueva base de datos en SQL Server.

Nuestra base de datos de ejemplo se va a llamar Demo y contendrá un modelo de datos sencillo basado en dos tablas relacionadas entre sí. Personas y Gastos.

Para administrar nuestra BD emplearemos la aplicación “SQL Server Management Studio”. Recuerda que este cliente no contiene un motor de bases de datos, deberemos instalarlo aparte o conectarnos con un servidor de bases de datos.

En nuestro caso, para labores de desarrollo lo recomendado es instalar SQL Local DB.

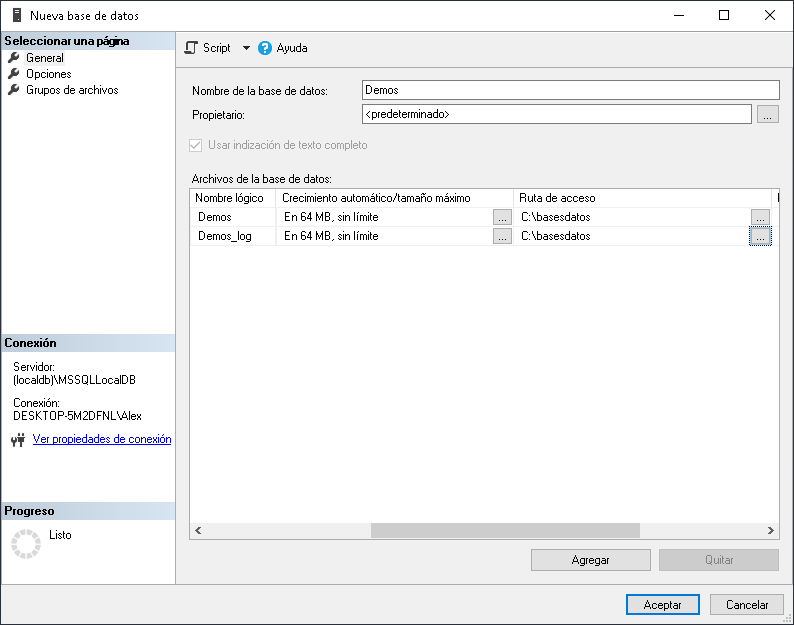
En la siguiente imagen vemos como conectarnos a una instancia de local db. Fíjate en el nombre del servidor, la primera parte es el nombre del motor “(localdb)” y la segunda parte es el nombre de la instancia “MSSQLLocalDB”.



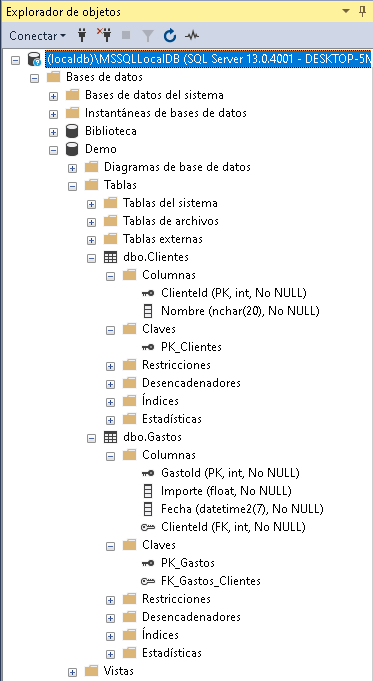
Una vez conectados vamos a crear una nueva BD. Para ello localizamos el nodo de bases de datos en el servidor y pulsamos en crear una nueva base de datos.

Se abrirá un asistente y en la pestaña “General” deberemos indicar los siguientes parámetros:

1. Nombre de la BD, en este caso “Demo”
2. Ruta de acceso para el fichero de datos, cuidado con las rutas relativas al nombre del usuario, en este caso emplearemos “c:\basesdatos”.
3. Ruta de acceso para el fichero de logs, emplearemos la misma ruta que en el punto anterior. Ten cuidado porque si pierdes el fichero de log no podrás leer los datos.



Ahora vamos a crear las dos tablas. Personas (personaid, nombre), Gastos (gastoid, importe, fecha, personaid). Ya deberías de ser capaz de identificar las claves tanto primarias como externas, el aspecto final es el siguiente.



### Preparar el proyecto

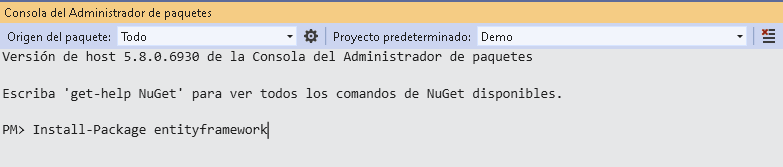
Visual Studio cuenta con un gestor de paquetes o librerías externas que nos facilita mucho la labor a la hora de integrar herramientas de terceros en nuestros proyectos. Este gestor se denomina Nuget y si lo buscas en internet verás que incluso cuenta con una página web del estilo a una tienda de aplicaciones ([www.nuget.org](http://www.nuget.org)).

A través de Nuget vamos a instalar en nuestro proyecto un ORM para el acceso a datos. Existen muchos ORM que nos pueden ser utilidad, y dependiendo del tipo de proyecto incluso puede que sea más aconsejable usar uno u otro tipo. Microsoft tiene su propio ORM que se conoce como “Entity Framework”, este ORM es muy fácil de usar y trata de dar el mayor número de funcionalidades disponibles en la base de datos. Para fines didácticos es muy aconsejable, pero en aplicaciones de alto rendimiento o en entornos no Microsoft puede ser aconsejable emplear otros como puedan ser Dapper, PetaPOCO, etc…

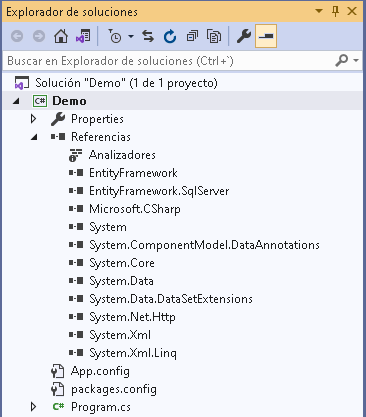
Lo primero es instalar el ORM en nuestro proyecto, para ello debemos abrir la “Consola de Administrador de paquetes” que se encuentra en “Herramientas -> Administrador de paquetes Nuget”. Y escribimos el siguiente comando

* Install-package entityframework

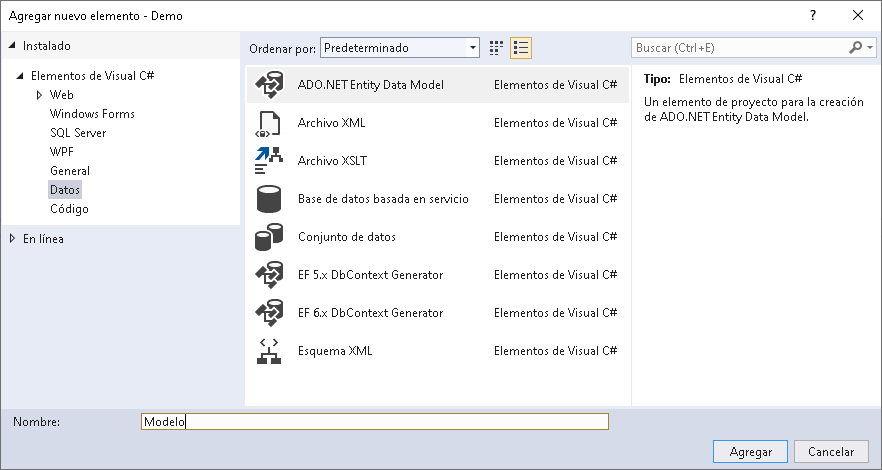
Es posible que dependiendo del tipo de proyecto ya se encuentre referenciado y que no haga falta instalarlo.



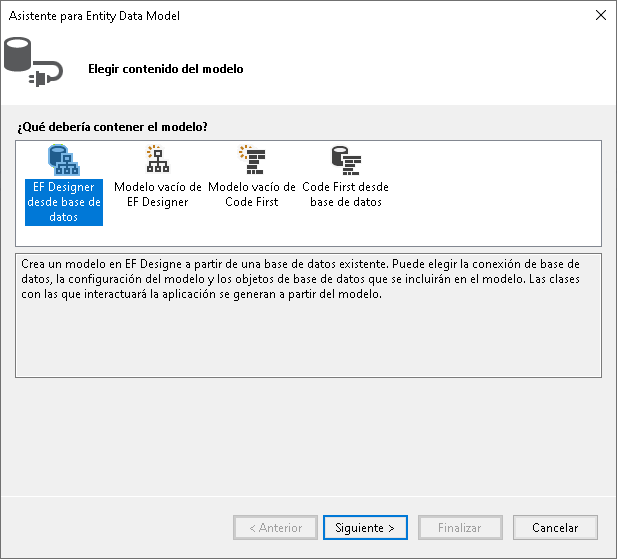
Una vez instalado veras que las referencias del proyecto han aumentado considerablemente.



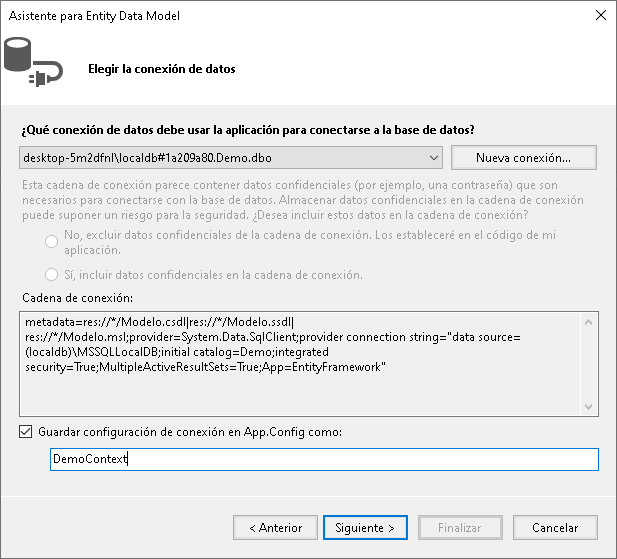
Ahora agregamos al proyecto un archivo con el modelo de datos de Entity Framework.



Y elegimos la primera opción “EF Designer desde base de datos”

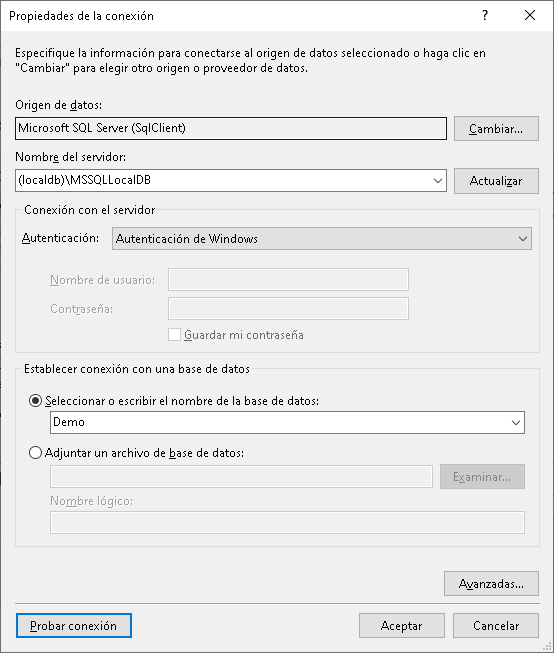


Ahora es el momento de crear una nueva cadena de conexión.



Fíjate en que está indicado que se guarde los datos de conexión en el fichero “App.config” además le hemos dado un nombre más apropiado a la conexión. **Esto del nombre es importante porque se creará una clase con el mismo nombre a través de la cual nos comunicaremos con la BD**.

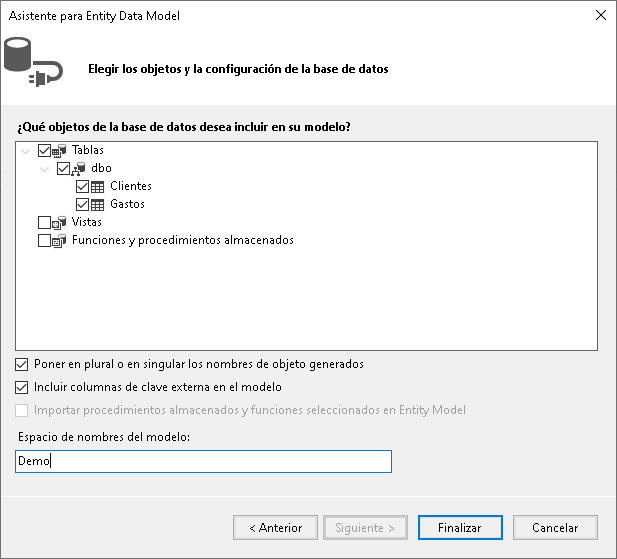
Si en el desplegable no tenemos una conexión a nuestra base de datos, en este caso “Demo” pulsamos en “Nueva conexión…” y nos conectamos como muestra la siguiente imagen.



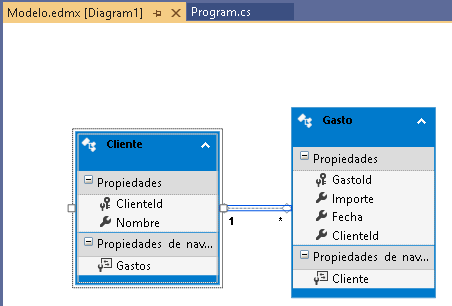
Indicamos con que tablas, vista, procedimientos almacenados vamos a trabajar, normalmente con marcar las únicamente las tablas es suficiente.

Marcamos la opción “**Poner en plural o en singular los nombres de los objetos generados**”, de esta forma por ejemplo si la tabla es “Clientes” la clase de código generada se llamará “Cliente”.

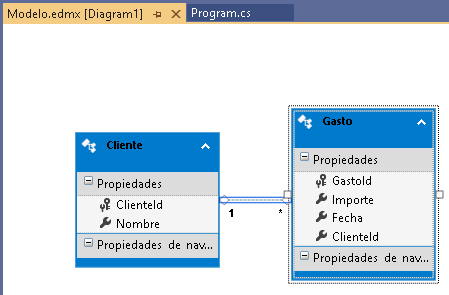
**Y muy importante**, ajustar el “**espacio de nombres del modelo**” al nombre del espacio de nombres del proyecto. Si no lo cambiamos nuestras clases estarán en otro espacio de nombres y tendremos que andar importando dicho espacio cada vez que trabajemos con alguna clase del modelo.



Al finalizar se nos abrirá el modelo de datos de entity. Fíjate que los nombres de las tablas están en singular, que tenemos una relación entre los modelos y que se han añadido unas nuevas propiedades denominadas “propiedades de navegación” que básicamente contienen instancias de los objetos relacionados.

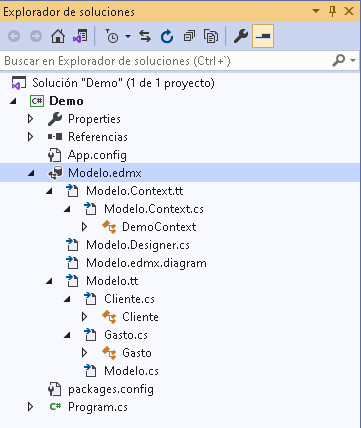


**Las propiedades de navegación las borramos**, son una fuente de problemas, imagínate que cargamos un cliente sin cargar sus gastos, al darle a actualizar sin haber cargado los datos relacionados puede entender que los hemos borrado. Tampoco es del todo funcional que para cargar un dato tenga que cargar todos los datos relacionados.



Es importante que recuerdes que **cada vez que cambies algo en este fichero y pulses en el botón de guardar automáticamente se reescribirán las clases con el código asociado al modelo**. El contexto y las clases de negocio, en este caso “DemoContext, Cliente y Gasto”.

En el explorador del proyecto verás como se han creado las distintas clases, a modo de resumen se basa en plantillas t4 que no son más que scripts que en base a ciertos datos escriben código. Colgando de estas plantillas tenemos las clases. Recuerda que todo lo que cuelga del archivo de modelo puede ser sobrescrito en cualquier momento por lo que no es bueno modificar estos archivos a mano.



Vamos a ver el código generado, fíjate que te avisa de que el código es autogenerado y no se recomienda su edición manual.

//------------------------------------------------------------------------------

// <auto-generated>

// Este código se generó a partir de una plantilla.

//

// Los cambios manuales en este archivo pueden causar un comportamiento inesperado de la aplicación.

// Los cambios manuales en este archivo se sobrescribirán si se regenera el código.

// </auto-generated>

//------------------------------------------------------------------------------

**namespace** Demo

**{**

**using** System**;**

**using** System**.**Collections**.**Generic**;**

**public** **partial** class Cliente

**{**

**public** int ClienteId **{** **get;** **set;** **}**

**public** string Nombre **{** **get;** **set;** **}**

**}**

**}**

Y la clase Gasto igual.

//------------------------------------------------------------------------------

// <auto-generated>

// Este código se generó a partir de una plantilla.

//

// Los cambios manuales en este archivo pueden causar un comportamiento inesperado de la aplicación.

// Los cambios manuales en este archivo se sobrescribirán si se regenera el código.

// </auto-generated>

//------------------------------------------------------------------------------

**namespace** Demo

**{**

**using** System**;**

**using** System**.**Collections**.**Generic**;**

**public** **partial** class Gasto

**{**

**public** int GastoId **{** **get;** **set;** **}**

**public** double Importe **{** **get;** **set;** **}**

**public** System**.**DateTime Fecha **{** **get;** **set;** **}**

**public** int ClienteId **{** **get;** **set;** **}**

**}**

**}**

Y el contexto, a través de esta clase accedemos a la base de datos. Es importante que recuerdes que hereda de “DbContext” y que si fuera necesario podemos manipular el acceso a los datos en esta clase contamos con las funciones que nos ayudarán.

De momento recordar dos cosas sobre la clase DbContext:

* La cadena que recibe el constructor es el nombre de la cadena de conexión definida en App.config. Por defecto el nombre de la cadena de conexión es igual que el del contexto, aunque podemos cambiarlo.
* Cuenta con un método “SaveChanges” que sincronizará los datos creados o modificados por el contexto con la base de datos. Vamos, el guardar cambios de toda la vida…

**namespace** Demo

**{**

**using** System**;**

**using** System**.**Data**.**Entity**;**

**using** System**.**Data**.**Entity**.**Infrastructure**;**

**public** **partial** class DemoContext **:** DbContext

**{**

**public** DemoContext**()**

**:** **base(**"name=DemoContext"**)**

**{**

**}**

**protected** **override** void OnModelCreating**(**DbModelBuilder modelBuilder**)**

**{**

**throw** **new** UnintentionalCodeFirstException**();**

**}**

**public** **virtual** DbSet**<**Cliente**>** Clientes **{** **get;** **set;** **}**

**public** **virtual** DbSet**<**Gasto**>** Gastos **{** **get;** **set;** **}**

**}**

**}**

Para usar el contexto lo único que tenemos que hacer es instanciarlo y recorrer sus propiedades, veras que sin hacer nada ya tienen los datos de la base de datos.

### Cadenas de conexión

El fichero App.config, que se muestra a continuación, contiene los parámetros de configuración de nuestro proyecto. En este caso, un proyecto de consola el fichero es bastante reducido por lo que vamos a aprovechar para describir su contenido de una forma muy general.

En primer lugar tenemos una sección para indicar que se cargue el paquete de “Entity Framework”, fíjate que se indica además la versión. Seguido tenemos la versión del framework de .Net, recuerda que se ofrece al crear un nuevo proyecto. Como tenemos definido un acceso a datos nos encontramos con el proveedor de SQL, es este caso para sql server, si tuviéramos otro motor este apartado indicaría el proveedor para el motor especifico. Y por último tenemos el apartado de las cadenas de conexión. Fíjate que tienen un nombre y luego los datos de conexión, si quisiéramos cambiar la dase de datos o el servidor es aquí donde modificar los datos. Dentro del apartado de cadenas de conexión podemos definir múltiples conexiones añadiendo nuevos nodos hijos.

<?xml version=**"1.0"** encoding=**"utf-8"**?>

<configuration>

<configSections>

<!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlink/?LinkID=237468 -->

<section name=**"entityFramework"** type=**"System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework, Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"** requirePermission=**"false"** />

</configSections>

<startup>

<supportedRuntime version=**"v4.0"** sku=**".NETFramework,Version=v4.7.2"** />

</startup>

<entityFramework>

<providers>

<provider invariantName=**"System.Data.SqlClient"** type=**"System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer"** />

</providers>

</entityFramework>

<connectionStrings>

<add name=**"DemoContext"** connectionString=**"metadata=res://\*/Modelo.csdl|res://\*/Modelo.ssdl|res://\*/Modelo.msl;provider=System.Data.SqlClient;provider connection string=&quot;data source=(localdb)\MSSQLLocalDB;initial catalog=Demo;integrated security=True;MultipleActiveResultSets=True;App=EntityFramework&quot;"** providerName=**"System.Data.EntityClient"** />

</connectionStrings>

</configuration>

### Acceso a datos mediante código

Insertar, en este caso debemos crear un nuevo objeto, añadirlo a la colección correspondiente y guardar los cambios.

Al guardar los cambios los identificadores de los objetos se actualizarán automáticamente.

**TRUCO**: si quisiéramos crear múltiples objetos con datos relacionados podemos asignar identificadores con valores negativos, al guardar estas relaciones se mantendrán y se les asignarán los identificadores correctos.

**private** static void InsertarDatos**()**

**{**

DemoContext bd **=** **new** DemoContext**();**

Cliente nuevoCliente **=** **new** Cliente**()** **{** Nombre **=** "Juan" **};**

bd**.**Clientes**.**Add**(**nuevoCliente**);**

bd**.**SaveChanges**();**

//fijate como el nuevo cliente ya tiene un id asignado

Gasto nuegoGasto **=** **new** Gasto**()** **{** ClienteId **=** nuevoCliente**.**ClienteId**,** Fecha **=** **new** DateTime**(**2020**,** 12**,** 1**),** Importe **=** 50.50 **};**

bd**.**Gastos**.**Add**(**nuegoGasto**);**

nuegoGasto **=** **new** Gasto**()** **{** ClienteId **=** nuevoCliente**.**ClienteId**,** Fecha **=** **new** DateTime**(**2020**,** 12**,** 5**),** Importe **=** 70.50 **};**

bd**.**Gastos**.**Add**(**nuegoGasto**);**

//confirmo los cambios

bd**.**SaveChanges**();**

nuevoCliente **=** **new** Cliente**()** **{** ClienteId **=** **-**1**,** Nombre **=** "Pepe" **};**

bd**.**Clientes**.**Add**(**nuevoCliente**);**

nuegoGasto **=** **new** Gasto**()** **{** ClienteId **=** **-**1**,** Fecha **=** **new** DateTime**(**2020**,** 11**,** 20**),** Importe **=** 10.50 **};**

bd**.**Gastos**.**Add**(**nuegoGasto**);**

nuegoGasto **=** **new** Gasto**()** **{** ClienteId **=** **-**1**,** Fecha **=** **new** DateTime**(**2020**,** 11**,** 25**),** Importe **=** 20.50 **};**

bd**.**Gastos**.**Add**(**nuegoGasto**);**

//confirmo los cambios

bd**.**SaveChanges**();**

**}**

Selección, en este caso debemos emplear LINQ. Con LINQ podemos indicar una variable, normalmente x, seguida de una flecha => donde la x representa cada objeto de la colección. La variable x automáticamente toma el tipo de la colección por lo que podemos acceder a sus propiedades y establecer criterios de filtrado. A esta notación se la conoce como expresiones lambda.

Ten en cuenta que las consultas con LINQ normalmente devuelven objetos de tipo consulta por lo que hasta que no se leen o se accede a los valores no tenemos un objeto de tipo colección. Para forzar a realizar la consulta podemos emplear el .ToList()

Fíjate que la sintaxis de LINQ se parece mucho a los términos empleados por SQL. Para empezar con .Where(lambda) para filtrar múltiples datos y .FirstOrDefault(lambda) para buscar un único dato será más que suficiente.

**private** static void ConsultasSeleccion**()**

**{**

DemoContext bd **=** **new** DemoContext**();**

//Todos los clientes

List**<**Cliente**>** clientes **=** bd**.**Clientes**.**ToList**();**

//Uso de una funcion agregada

int maximoId **=** bd**.**Clientes**.**Max**(**x **=>** x**.**ClienteId**);**

//Ultimo cliente insertado, mayor id

Cliente cliente **=** bd**.**Clientes**.**FirstOrDefault**(**x **=>** x**.**ClienteId **==** maximoId**);**

//gastos del cliente anterior

List**<**Gasto**>** gastosCliente **=** bd**.**Gastos**.**Where**(**x **=>** x**.**ClienteId**.**Equals**(**cliente**.**ClienteId**)).**ToList**();**

//Gastos inferiores a 11 o superiores a 70

List**<**Gasto**>** gastos **=** bd**.**Gastos**.**Where**(**x **=>** x**.**Importe **<** 11 **||** x**.**Importe **>** 70**).**ToList**();**

**}**

Borrado, en este caso buscamos el objeto con una consulta de selección, lo eliminamos de la colección y guardamos los cambios.

**private** static void ConsultaBorrado**()**

**{**

DemoContext bd **=** **new** DemoContext**();**

//busco el objeto a eliminar

List**<**Gasto**>** gastosBorrar **=** bd**.**Gastos**.**Where**(**x **=>** x**.**Importe **>** 70**).**ToList**();**

//elimino los objetos de la lista

bd**.**Gastos**.**RemoveRange**(**gastosBorrar**);**

//confirmo los cambios

bd**.**SaveChanges**();**

**}**

Modificación, lo recomendable, y además de ser la manera tradicional, es buscar el objeto por si no existiera, actualizar sus valores y guardar los cambios.

**private** static void ConsultaModificacion**()**

**{**

DemoContext bd **=** **new** DemoContext**();**

//es interesante asegurarse de que el objeto existe antes de actualizar

//para no crearlo

Cliente cliente **=** bd**.**Clientes**.**FirstOrDefault**(**x **=>** x**.**Nombre**.**Equals**(**"Juan"**));**

cliente**.**Nombre **=** "Maria"**;**

bd**.**SaveChanges**();**

**}**

Modificación con un objeto no perteneciente al contexto, visto lo anterior ahora también es posible actualizar un objeto sin comprobar su existencia. Antiguamente se producía un error en el que se indicaba que el objeto no pertenecía al contexto actual.

**private** static void ConsultaModificacionExterno**()**

**{**

DemoContext bd **=** **new** DemoContext**();**

//esto en versiones antiguas producía un error

Cliente cliente **=** **new** Cliente**()** **{** ClienteId **=** 2**,** Nombre **=** "Maria2" **};**

bd**.**Clientes**.**Add**(**cliente**);**

bd**.**SaveChanges**();**

**}**