# Creating applications with Entity Framework

This chapter will describe the process of creating applications with a Firebird database, using the Entity Framework access components in the Visual Studio 2015 environment.

**ADO.NET Entity Framework** (EF) combines an object-oriented data access technology with an object-relational mapping (ORM) solution for the Microsoft .NET Framework. It enables interaction with objects by means of both LINQ in the form of LINQ to Entities and with Entity SQL.

Entity Framework assumes three possible methods for interacting with the database:

* **Database first**: Entity Framework creates a set of classes that reflect the model of an existing database
* **Model first**: the developer creates a database model that Entity Framework later uses to create an actual database on the server.
* **Code first**: the developer creates a class for the model of the data that will be stored in a database and then Entity Framework uses this model to generate the database and its tables

Our sample application will use the Code First approach, but you could use one of the others just as easily.

|  |
| --- |
| **Note**  As we already have a database, we will just write the code that would result in creating that database. |

## Setting Up for Firebird in Visual Studio 2015

To prepare for working with Firebird, you will need to install the following:

* FirebirdSql.Data.FirebirdClient.dll
* EntityFramework.Firebird.dll
* DDEX Provider for Visual Studio

There is nothing difficult in installing the first two. They are currently distributed and installed into a project by means of the [NuGet package manager](https://www.nuget.org/). The DDEX Provider library, designed for operating Visual Studio wizards, is not so easy to install and may take more time and effort.

Efforts have been made to automate the installation process and include all components in a single installer package: <http://sourceforge.net/projects/firebird-4-8-0-ddex-installer/>. However, you might need to install all components manually under some conditions. If so, you can download the following:

* FirebirdSql.Data.FirebirdClient-4.10.0.0.msi (<http://sourceforge.net/projects/firebird/files/firebird-net-provider/4.10.0.0/FirebirdSql.Data.FirebirdClient-4.10.0.0.msi/download>)
* EntityFramework.Firebird-4.10.0.0-NET45.7z (<http://sourceforge.net/projects/firebird/files/firebird-net-provider/4.10.0.0/EntityFramework.Firebird-4.10.0.0-NET45.7z/download>)
* DDEXProvider-3.0.2.0.7z (<http://sourceforge.net/projects/firebird/files/firebird-net-provider/DDEX%203.0.2/DDEXProvider-3.0.2.0.7z/download>)
* DDEXProvider-3.0.2.0-src.7z (<http://sourceforge.net/projects/firebird/files/firebird-netprovider/DDEX%203.0.2/DDEXProvider-3.0.2.0-src.7z/download>)

### The Installation Process

**Important!**

Because the installation involves operations in protected directories, you will need administrator privileges to do it.

#### Steps

1. Install FirebirdSql.Data.FirebirdClient-4.10.0.0.msi
2. Unpack EntityFramework.Firebird-4.10.0.0-NET45.7z to the folder with the installed Firebird client. In my case, it is the folder c:\Program Files (x86)\FirebirdClient\.
3. You need to install a Firebird build into the GAC. For your convenience, specify the path to the gacutil utility for .NET Framework 4.5 in the environment variable %PATH%. In my case, the path is c:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools\
4. Run the command shell cmd.exe as the administrator and go to the directory with the installed client.

chdir "c:\Program Files (x86)\FirebirdClient"

1. Now make sure that FirebirdSql.Data.FirebirdClient is installed into the GAC by typing the following command:

gacutil /l FirebirdSql.Data.FirebirdClient

If FirebirdSql.Data.FirebirdClient has not been installed into the GAC, use the following command to do it now:

gacutil /i FirebirdSql.Data.FirebirdClient.dll

1. Now install EntityFramework.Firebird into the GAC

gacutil /i EntityFramework.Firebird.dll

1. Unpack DDEXProvider-3.0.2.0.7z to a directory convenient for you. Mine was unpacked to c:\Program Files (x86)\FirebirdDDEX\
2. Unpack the contents of the /reg\_files/VS2015 subdirectory from the archive DDEXProvider-3.0.2.0-src.7z there as well.

**Author's remark**

For some strange reason these files are absent from the archive with the compiled dll libraries, but they are present in the source code archive.

1. Open the FirebirdDDEXProvider64.reg file in Notepad. Find the line that contains %path% and change it to the full path to the file FirebirdSql.VisualStudio.DataTools.dll

e.g.,

"CodeBase"="c:\\Program Files (x86)\\FirebirdDDEX\\FirebirdSql.VisualStudio.DataTools.dll"

1. Save this Registry file and run it. Click YES to the question about adding the information to the Registry.
2. Now you need to edit the machine.config file. In my installation, the path is as follows: C:\Windows\Microsoft.NET\Framework\v4.0.30319\Config

Open this file in Notepad. Find the following section:

<system.data>

<DbProviderFactories>

Add the following lines to this section:

<add name="FirebirdClient Data Provider"

invariant="FirebirdSql.Data.FirebirdClient"

description=".Net Framework Data Provider for Firebird"

type="FirebirdSql.Data.FirebirdClient.FirebirdClientFactory,

FirebirdSql.Data.FirebirdClient, Version=4.10.0.0, Culture=neutral,

PublicKeyToken=3750abcc3150b00c" />

**Note**

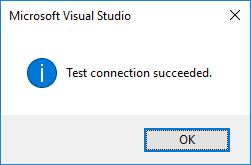
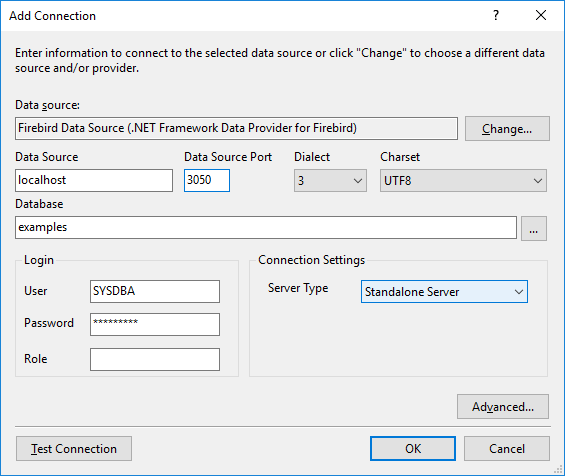
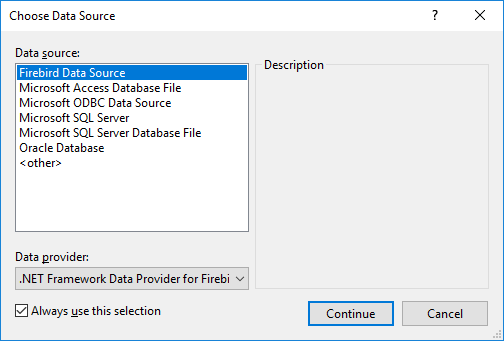
The settings we have configured here are valid for version 4.10.0.

Do the same for machine.config located at c:\Windows\Microsoft.NET\Framework64\v4.0.30319\Config\

This completes the installation.

##### Testing the Installation

To make sure that everything has been successfully installed, start Visual Studio 2015. Find the server explorer and try to connect to an existing Firebird database.



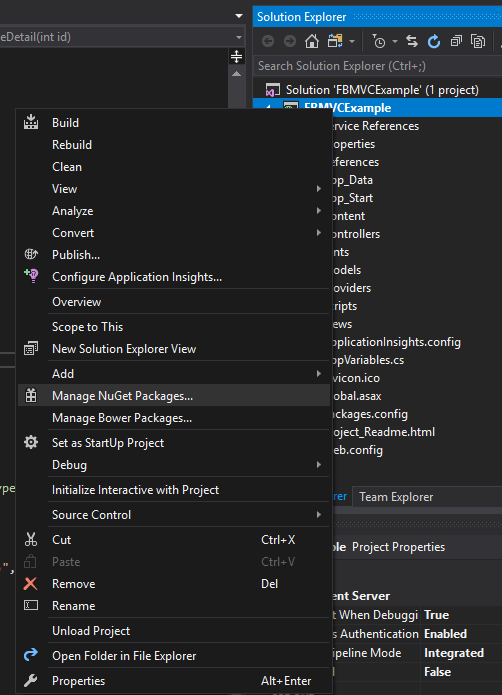
# Creating a Project

For our example in thischapter, we will create a Windows Forms application. Other types of applications differ from it, but the principles of working with Firebird via Entity Framework remain the same.

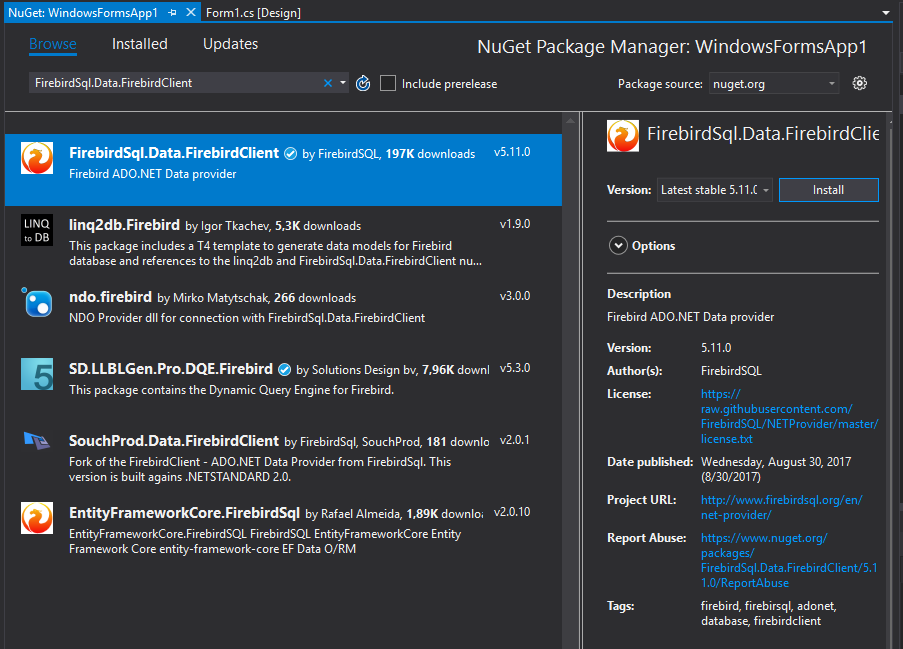
The first thing to do after creating a Windows Forms project is to add the following packages to it, using the NuGet package manager:

* FirebirdSql.Data.FirebirdClient
* EntityFramework
* EntityFramework.Firebird

Right-click the project name in Solution Explorer and select Manage NuGet Packages from the drop-downlist.



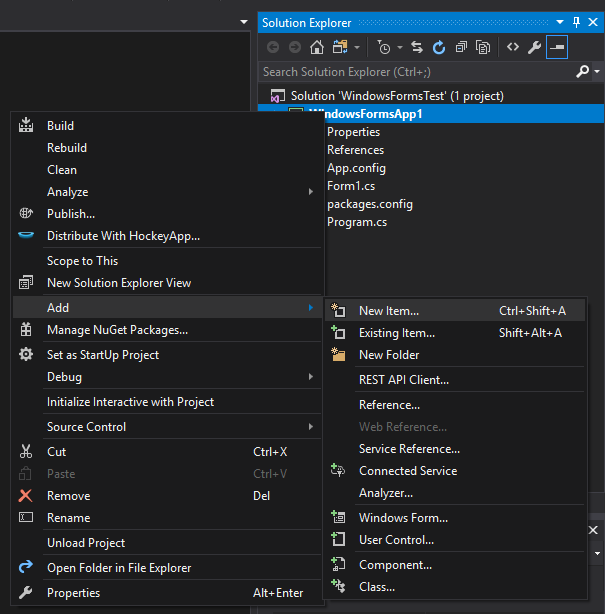
Find the packages listed above in the Nuget catalogue and install them in the package manager.



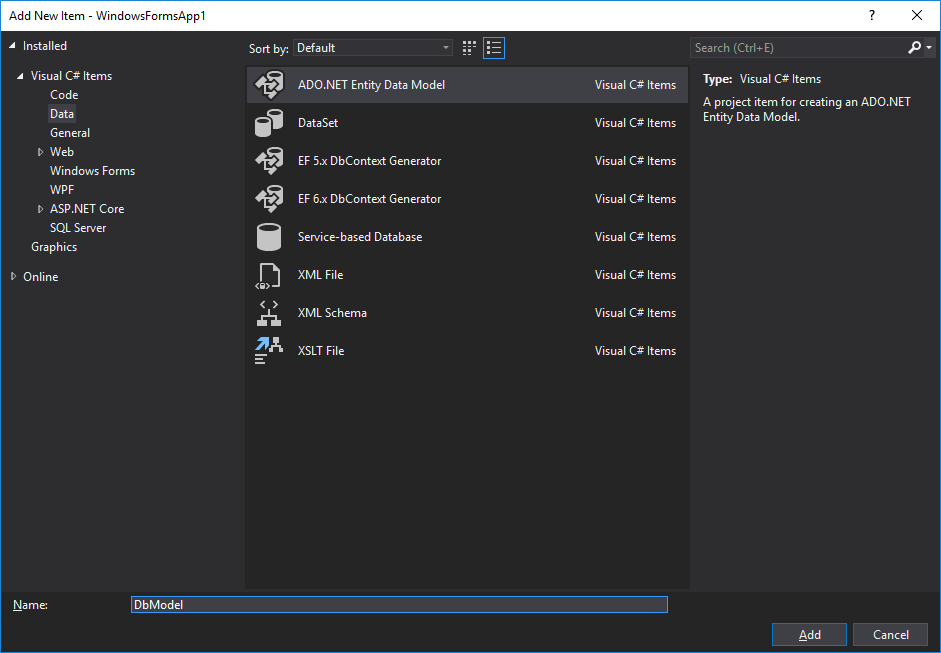
## Creating an Entity Data Model (EDM)

In our application, we will use the Code First approach.

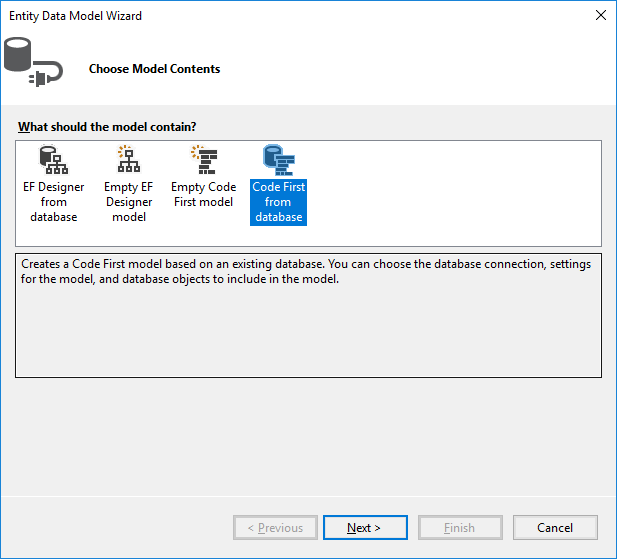
To create an EDM, right-click the project name in Solution Explorer and select the Add -> New Item menu item.



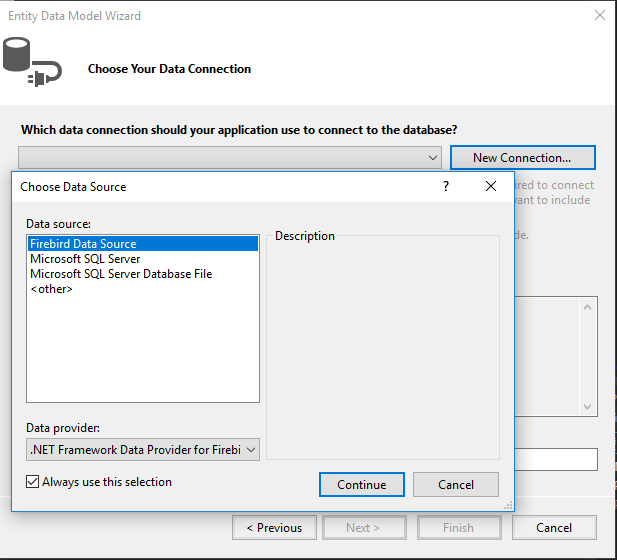
After that, select the ADO.NET Entity Data Model item in the Add New Item wizard.



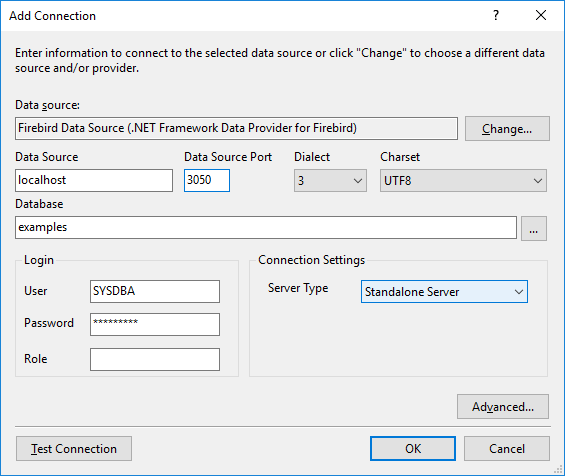
Since we already have a database, we will generate the EDM from the database. Select the icon captioned *Code First from database.*



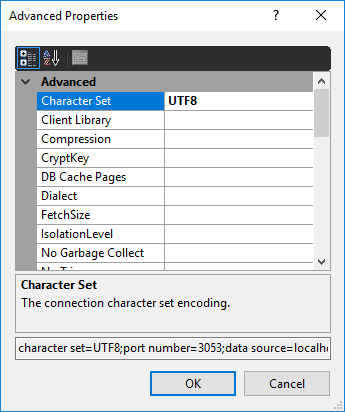
Now we need to select the connection the model will be created from. If the connection does not exist, it will have to be created.



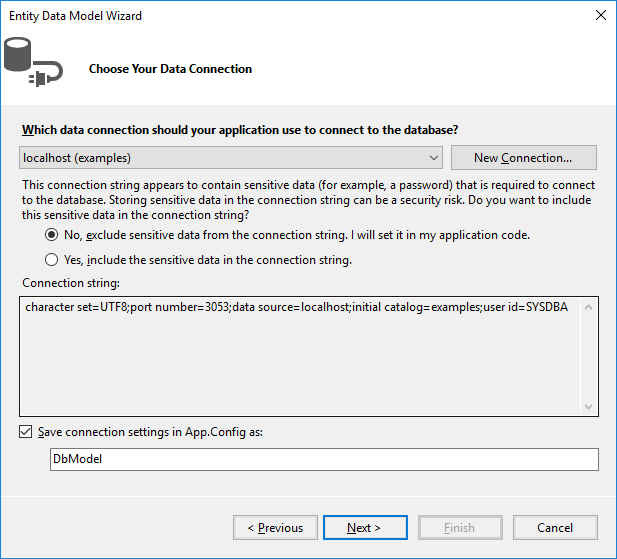
You might need to specify some advanced properties in addition to the main connection properties. You might want to set the transaction isolation to a level different from the default *Read Committed*, or to specify connection pooling, or something else that differs from defaults.



TIP Snapshot is the recommended isolation level because Entity Framework and ADO.NET both use disconnected data access--where each connection and each transaction is active for a very short time--.



The Entity Data Model wizard will ask you how to store the connection string.



For a web application or another three-tier architecture, where all users will be working with the database using a single account, select Yes. If your application is going to request authentication for connecting to the database, select No.

TIP It is much more convenient to work with wizards if you select Yes for each property. You can always change the isolation level in the application when it is ready for testing and deployment by just editing the connection string in the *<AppName>*.exe.conf application configuration file. The connection string will be stored in the **connectionStrings** section and will look approximately like this:

<add name="DbModel"

connectionString="character set=UTF8; data source=localhost;

initial catalog=examples; port number=3050;

user id=sysdba; dialect=3; isolationlevel=Snapshot;

pooling=True; password=masterkey;"

providerName="FirebirdSql.Data.FirebirdClient" />

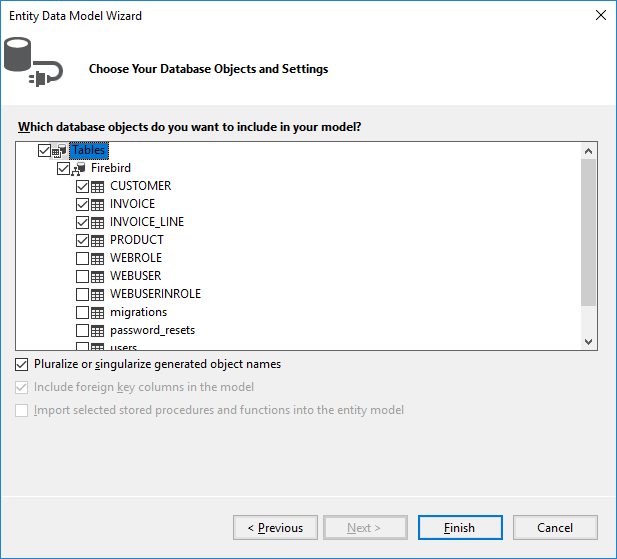
For the configuration file to stop storing the confidential information, just delete the following from the connection string: password=masterkey;

**Firebird 3.0 Notes**

Unfortunately, the current ADO.Net provider for Firebird (version 5.9.0.0) does not support network traffic encryption (Wirecrypt=Enabled by default in Firebird 3.0). If you want to work with Firebird 3.0, you need to change some settings in firebird.conf (or in databases.conf for a specific database) for Firebird to work without using network encryption. To do it, change the following settings:

WireCrypt = Disabled

Next, you will be asked which tables and views should be included in the model.



For our project, select the four tables that are checked in the screenshot.

The basic EDM is now ready.

### The Generated EDM Files

When the wizard's work is finished, you should have five new files: a model file and four files each describing an entity in the model.

#### An Entity File

Let's take a look at the generated file describing the INVOICE entity:

[Table("Firebird.INVOICE")]

**public partial class** INVOICE

{

[System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",

"CA2214:DoNotCallOverridableMethodsInConstructors")]

**public** INVOICE()

{

INVOICE\_LINES = **new** HashSet<INVOICE\_LINE>();

}

[Key]

[DatabaseGenerated(DatabaseGeneratedOption.None)]

**public int** INVOICE\_ID { **get**; **set**; }

**public int** CUSTOMER\_ID { **get**; **set**; }

**public** DateTime? INVOICE\_DATE { **get**; **set**; }

**public decimal**? TOTAL\_SALE { **get**; **set**; }

**public short** PAYED { **get**; **set**; }

**public virtual** CUSTOMER CUSTOMER { **get**; **set**; }

[System.Diagnostics.CodeAnalysis.SuppressMessage("Microsoft.Usage",

"CA2227:CollectionPropertiesShouldBeReadOnly")]

**public virtual** ICollection<INVOICE\_LINE> INVOICE\_LINES { **get**; **set**; }

}

The class contains properties for each field of the INVOICE table. Each of these properties has attributes that describe constraints. You can study the details of the various attributes in the Microsoft document, [Code First Data Annotations](https://msdn.microsoft.com/en-us/data/jj591583).

Navigation Properties and "Lazy Loading"

Two navigation properties are generated: CUSTOMER and INVOICE\_LINES. The first one contains a reference to the customer entity. The second contains a collection of invoice lines. It is generated because the INVOICE\_LINE table has a foreign key to the INVOICE table. Of course, you can remove this property from the INVOICE entity, but it is not really necessary. The CUSTOMER and INVOICE\_LINES properties use "lazy loading" which means that loading is not performed until the first access to an object. That way, the loading of related data is avoided unless it is actually needed. Once the data are accessed via the navigation property, they will be loaded from the database automatically.

Important

If lazy loading is in effect, classes that use lazy loading must be public and their properties must have the **public** and **virtual** keywords.

#### The DbModel Class

Next, we examine the DbModel.cs file that describes the overall model.

public partial class DbModel : DbContext

{

public DbModel()

: base("name=DbModel")

{

}

public virtual DbSet<CUSTOMER> CUSTOMERS { get; set; }

public virtual DbSet<INVOICE> INVOICES { get; set; }

public virtual DbSet<INVOICE\_LINE> INVOICE\_LINES { get; set; }

public virtual DbSet<PRODUCT> PRODUCTS { get; set; }

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

modelBuilder.Entity<CUSTOMER>()

.Property(e => e.ZIPCODE)

.IsFixedLength();

modelBuilder.Entity<CUSTOMER>()

.HasMany(e => e.INVOICES)

.WithRequired(e => e.CUSTOMER)

.WillCascadeOnDelete(false);

modelBuilder.Entity<PRODUCT>()

.HasMany(e => e.INVOICE\_LINES)

.WithRequired(e => e.PRODUCT)

.WillCascadeOnDelete(false);

modelBuilder.Entity<INVOICE>()

.HasMany(e => e.INVOICE\_LINES)

.WithRequired(e => e.INVOICE)

.WillCascadeOnDelete(false);

}

}

The properties coded here describe a dataset for each entity, along with advanced properties that are specified for creating a model with Fluent API. A complete description of the Fluent API can be found in the Microsoft document entitled [Configuring/Mapping Properties and Types with the Fluent API](https://msdn.microsoft.com/en-us/data/jj591617.aspx).

We will use the Fluent API to specify precision (and scale) for properties of type DECIMAL in the OnModelCreating method, by adding the following lines:

modelBuilder.Entity<PRODUCT>()

.Property(p => p.PRICE)

.HasPrecision(15, 2);

modelBuilder.Entity<INVOICE>()

.Property(p => p.TOTAL\_SALE)

.HasPrecision(15, 2);

modelBuilder.Entity<INVOICE\_LINE>()

.Property(p => p.SALE\_PRICE)

.HasPrecision(15, 2);

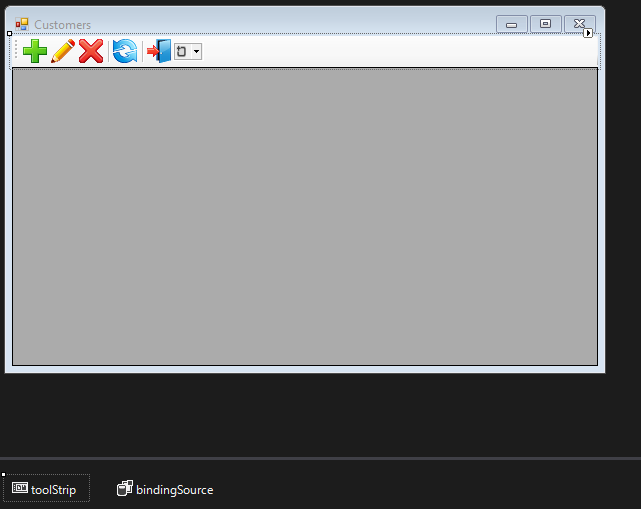
modelBuilder.Entity<INVOICE\_LINE>()

.Property(p => p.QUANTITY)

.HasPrecision(15, 0);

## Creating a User Interface

In our application, we will create user interfaces for two primary entities: a form each for the product and the customer entities. Each form contains a DataGridView grid, a ToolStrip toolbar with buttons and also a BindingSource component that is used to bind data to the controls on the form.



Since both forms are similar in function and implementation, we will describe just one.

### Getting a Context

To work with our model, we will need the method for getting a context (or a model). The following statement is sufficient for that purpose:

DbModel dbContext = **new** DbModel();

If no confidential data are stored in the connection string – for example, the password is absent because it will be captured during the authentication process when the application is started – we will need a special method for storing and recovering the connection string or for storing the previously created context. For that, we will create a special class containing some application-level global variables, along with a method for getting a context.

A context might be the start and end dates of a work period, for example.

**static class** AppVariables

{

**private static** DbModel dbContext = **null**;

/// <summary>

/// Start date of the working period

/// </summary>

**public static** DateTime StartDate { **get**; **set**; }

/// <summary>

/// End date of the working period

/// </summary>

**public static** DateTime FinishDate { **get**; **set**; }

/// <summary>

/// Returns an instance of the model (context)

/// </summary>

/// <returns>Model</returns>

**public static** DbModel CreateDbContext() {

dbContext = dbContext ?? **new** DbModel();

**return** dbContext;

}

}

The connection string itself is applied after the authentication process completes successfully during the application launch. We will add the following code to the Load event handler of the main form for that.

**private void** MainForm\_Load(**object** sender, EventArgs e) {

**var** dialog = **new** LoginForm();

**if** (dialog.ShowDialog() == DialogResult.OK)

{

**var** dbContext = AppVariables.getDbContext();

**try**

{

**string** s = dbContext.Database.Connection.ConnectionString;

var builder = **new** FbConnectionStringBuilder(s);

builder.UserID = dialog.UserName;

builder.Password = dialog.Password;

dbContext.Database.Connection.ConnectionString = builder.ConnectionString;

*// try connect*

dbContext.Database.Connection.Open();

}

**catch** (Exception ex)

{

*// display error*

MessageBox.Show(ex.Message, "Error");

Application.Exit();

}

}

**else**

Application.Exit();

}

Now, to get a context, we use the static CreateDbContext method:

**var** dbContext = AppVariables.getDbContext();

### Working with Data

The entities in the model definition contain no data. The easiest way to load data is to call the Load method. For example,

**private void** LoadCustomersData()

{

dbContext.CUSTOMERS.Load();

**var** customers = dbContext.CUSTOMERS.Local;

bindingSource.DataSource = customers.ToBindingList();

}

**private void** CustomerForm\_Load(**object** sender, EventArgs e)

{

LoadCustomersData();

dataGridView.DataSource = bindingSource;

dataGridView.Columns["CUSTOMER\_ID"].Visible = **false**;

}

However, this approach has a few drawbacks:

1. The Load method loads all data from the CUSTOMER table to memory at once.
2. Although lazy properties (INVOICES) are not loaded immediately, but only once they are accessed, they will be loaded anyway when the records are shown in the grid and it will happen each time a group of records is shown.
3. Record ordering is not defined.

To get around these drawbacks, we will use a feature of the LINQ (Language Integrated Query) technology, **LINQ to Entities**. LINQ to Entities offers a simple and intuitive approach to getting data using C# statements that are syntactically similar to SQL query statements. You can read about the LINQ syntax in [LINQ to Entities](https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/language-reference/linq-to-entities).

#### LINQ Extension Methods

The LINQ extension methods can return two objects: **IEnumerable** and **IQueryable**. The IQueryable interface is inherited from IEnumerable so, theoretically, an IQueryable object is also an IEnumerable. In reality, they are distinctly different.

The IEnumerable interface is in the **System.Collections** namespace. An IEnumerable object is a collection of data in memory that can be addressed only in a forward direction. During the query execution, IEnumerable loads all data. Filtering, if required, is done on the client side.

The IQueryable interface is in the System.Linq namespace. It provides remote access to the database and allows movement through the data to be bi-directional. During the process of creating a query that returns an IQueryable object, the query is optimized to minimise memory usage and network bandwidth.

The Local property returns the IEnumerable interface, through which we can create LINQ queries.

**private void** LoadCustomersData()

{

var dbContext = AppVariables.getDbContext();

dbContext.CUSTOMERS.Load();

**var** customers =

**from** customer **in** dbContext.CUSTOMERS.Local

**orderby** customer.NAME

**select new** customer;

bindingSource.DataSource = customers.ToBindingList();

}

However, as this query will be executed on the data in memory, it is really useful only for small tables that do not need to be filtered beforehand.

For a LINQ query to be converted into SQL and executed on the server, we need to access the dbContext.CUSTOMERS directly instead of accessing the dbContext.CUSTOMERS.Localproperty in the LINQ query. The prior call to dbContext.CUSTOMERS.Load();to load the collection to memory is not required.

##### IQueryable and BindingList

IQueryable objects present a small problem: they cannot return BindingList. BindingList is a base class for creating a two-way data-binding mechanism. We can use the IQueryable interface to get a regular list by calling ToList but, this way, we lose handy features such as sorting in the grid and several more. The deficiency was fixed in .NET Framework 5 by creating a special extension. To do the same thing in FW4, we will create our own solution.

**public static class** DbExtensions

{

*// Internal class for map generator values to it*

**private class** IdResult

{

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

}

*// Cast IQueryable to BindingList*

**public static** BindingList<T> ToBindingList<T>

(**this** IQueryable<T> source) **where** T : **class**

{

**return** (**new** ObservableCollection<T>(source)).ToBindingList();

}

*// Get the next value of the sequence*

**public static int** NextValueFor(**this** DbModel dbContext, **string** genName)

{

**string** sql = String.Format(

"SELECT NEXT VALUE FOR {0} AS Id FROM RDB$DATABASE", genName);

**return** dbContext.Database.SqlQuery<IdResult>(sql).First().Id;

}

*// Disconnect all objects from the DbSet collection from the context*

*// Useful for updating the cache*

**public static void** DetachAll<T>(**this** DbModel dbContext, DbSet<T> dbSet)

**where** T : **class**

{

**foreach** (var obj **in** dbSet.Local.ToList())

{

dbContext.Entry(obj).State = EntityState.Detached;

}

}

*// Update all changed objects in the collection*

**public static void** Refresh(**this** DbModel dbContext, RefreshMode mode,

IEnumerable collection)

{

**var** objectContext = ((IObjectContextAdapter)dbContext).ObjectContext;

objectContext.Refresh(mode, collection);

}

*// Update the object*

**public static void** Refresh(**this** DbModel dbContext, RefreshMode mode,

**object** entity)

{

**var** objectContext = ((IObjectContextAdapter)dbContext).ObjectContext;

objectContext.Refresh(mode, entity);

}

}

##### Other Extensions

There are several more extensions in the iQueryable interface:.

NextValueFor is used to get the next value from the generator.

dbContext.Database.SqlQuery allows SQL queries to be executed directly and their results to be displayed on some entity (projection).

DetachAll is used to detach all objects of the DBSet collection from the context. It is necessary to update the internal cache, because all retrieved data are cached and are not retrieved from the database again. However, that is not always useful because it makes it more difficult to get the latest version of records that were modified in another context.

**Note**

In web applications, a context usually exists for a very short period of time. A new context has an empty cache.

Refresh is used to update the properties of an entity object. It is useful for updating the properties of an object after it has been edited or added.

##### Code for Loading the Data

Our code for loading data will look like this:

**private void** LoadCustomersData()

{

var dbContext = AppVariables.getDbContext();

*// disconnect all loaded objects*

*// this is necessary to update the internal cache*

*// for the second and subsequent calls of this method*

dbContext.DetachAll(dbContext.CUSTOMERS);

**var** customers =

**from** customer **in** dbContext.CUSTOMERS

**orderby** customer.NAME

**select** customer;

bindingSource.DataSource = customers.ToBindingList();

}

**private void** CustomerForm\_Load(**object** sender, EventArgs e)

{

LoadCustomersData();

dataGridView.DataSource = bindingSource;

dataGridView.Columns["INVOICES"].Visible = **false**;

dataGridView.Columns["CUSTOMER\_ID"].Visible = **false**;

dataGridView.Columns["NAME"].HeaderText = "Name";

dataGridView.Columns["ADDRESS"].HeaderText = "Address";

dataGridView.Columns["ZIPCODE"].HeaderText = "ZipCode";

dataGridView.Columns["PHONE"].HeaderText = "Phone";

}

###### Adding a Customer

This is the code of the event handler for clicking the Add button:

**private void** btnAdd\_Click(**object** sender, EventArgs e) {

**var** dbContext = AppVariables.getDbContext();

*// creating a new entity instance*

**var** customer = (CUSTOMER)bindingSource.AddNew();

*// create an editing form*

**using** (CustomerEditorForm editor = **new** CustomerEditorForm()) {

editor.Text = "Add customer";

editor.Customer = customer;

*// Form Close Handler*

editor.FormClosing += **delegate** (**object** fSender,

FormClosingEventArgs fe) {

**if** (editor.DialogResult == DialogResult.OK) {

**try** {

*// get next sequence value*

*// and assign it*

customer.CUSTOMER\_ID = dbContext.NextValueFor("GEN\_CUSTOMER\_ID");

*// add a new customer*

dbContext.CUSTOMERS.Add(customer);

*// trying to save the changes*

dbContext.SaveChanges();

*// and update the current record*

dbContext.Refresh(RefreshMode.StoreWins, customer);

}

**catch** (Exception ex) {

*// display error*

MessageBox.Show(ex.Message, "Error");

*// Do not close the form to correct the error*

fe.Cancel = **true**;

}

}

**else**

bindingSource.CancelEdit();

};

*// show the modal form*

editor.ShowDialog(**this**);

}

}

While adding a new record, we used the generator to get the value of the next identifier. We could have done it without applying the value of the identifier, leaving the BEFORE INSERT trigger to fetch the next value of the generator and apply it. However, that would leave us unable to update the added record.

###### Editing a Customer

The code of the event handler for clicking the Edit button is as follows:

**private void** btnEdit\_Click(**object** sender, EventArgs e) {

var dbContext = AppVariables.getDbContext();

*// get instance*

var customer = (CUSTOMER)bindingSource.Current;

*// create an editing form*

**using** (CustomerEditorForm editor = **new** CustomerEditorForm()) {

editor.Text = "Edit customer";

editor.Customer = customer;

*// Form Close Handler*

editor.FormClosing += **delegate** (**object** fSender, FormClosingEventArgs fe) {

**if** (editor.DialogResult == DialogResult.OK) {

**try** {

*// trying to save the changes*

dbContext.SaveChanges();

dbContext.Refresh(RefreshMode.StoreWins, customer);

*// update all related controls*

bindingSource.ResetCurrentItem();

}

**catch** (Exception ex) {

*// display error*

MessageBox.Show(ex.Message, "Error");

*// Do not close the form to correct the error*

fe.Cancel = **true**;

}

}

**else**

bindingSource.CancelEdit();

};

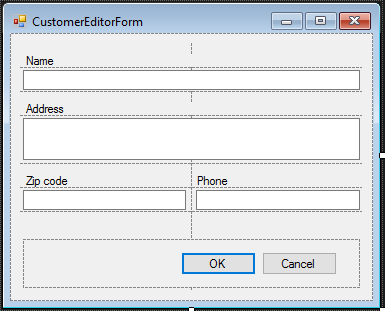
*// show the modal form*

editor.ShowDialog(**this**);

}

}

The form for editing the customer looks like this.



The code for binding to data is very simple.

**public** CUSTOMER Customer { **get**; **set**; }

**private void** CustomerEditorForm\_Load(**object** sender, EventArgs e)

{

edtName.DataBindings.Add("Text", **this**.Customer, "NAME");

edtAddress.DataBindings.Add("Text", **this**.Customer, "ADDRESS");

edtZipCode.DataBindings.Add("Text", **this**.Customer, "ZIPCODE");

edtPhone.DataBindings.Add("Text", **this**.Customer, "PHONE");

}

Editing a CustomerThe code of the event handler for clicking the Delete button is as follows:

**private void** btnDelete\_Click(**object** sender, EventArgs e) {

var dbContext = AppVariables.getDbContext();

var result = MessageBox.Show("Are you sure you want to delete the customer?",

"Confirmation",

MessageBoxButtons.YesNo,

MessageBoxIcon.Question);

**if** (result == DialogResult.Yes) {

*// get the entity*

var customer = (CUSTOMER)bindingSource.Current;

**try** {

dbContext.CUSTOMERS.Remove(customer);

*// trying to save the changes*

dbContext.SaveChanges();

*// remove from the linked list*

bindingSource.RemoveCurrent();

}

**catch** (Exception ex) {

*// display error*

MessageBox.Show(ex.Message, "Error");

}

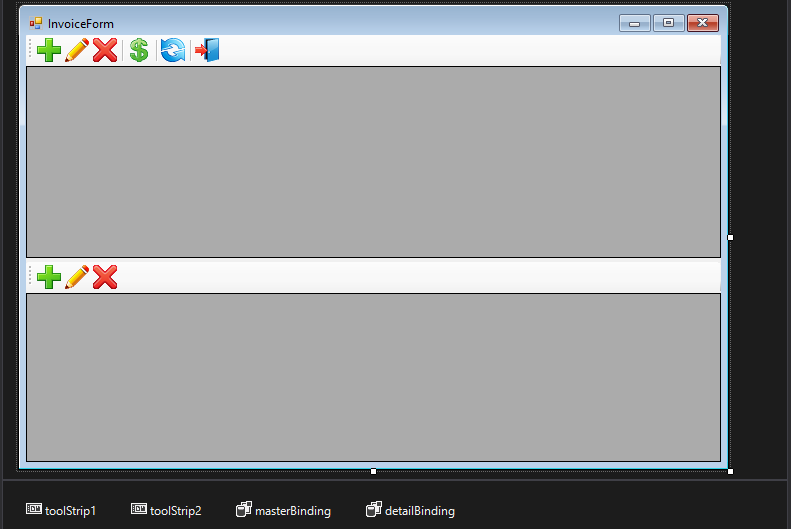
}

}

### Secondary Modules

Our application will have only one secondary module, named "Invoices". Secondary modules typically contain larger numbers of records than primary ones and new records are added to them frequently.

An invoice consists of a title where some general attributes are described (number, date, customer …) and invoice lines with the list of products, their quantities, prices, etc. It is convenient to have two grids for such documents: the main one showing the invoice header data and the detail one for the list of products sold. We will need a DataGridView component for each entityon the document form, binding the appropriate BindingSource to each.



#### Filtering Data

Most secondary entities contain a field with the document creation date. In order to reduce the amount of retrieved data, the concept of a work period is often introduced to filter the data sent to the client. A work period is a range of dates for which the work documents are required. Since the application can have more than one secondary entity, it makes sense to add variables containing the start and end dates of a work period to the global AppVariables data module (see ) that is used by all modules working with the database in one way or another. Once the application is started, the work period is usually defined by the dates when the current quarter starts and ends, although of course, other options are possible. While working with the application, the user can change the work period.

#### Loading the Invoice Data

Since the most recent records are the most requested, it makes sense to sort them by date in reverse order. As with the primary modules, we will use LINQ to retrieve data. The following method loads the invoice headers:

**public void** LoadInvoicesData() {

var dbContext = AppVariables.getDbContext();

var invoices =

**from** invoice **in** dbContext.INVOICES

**where** (invoice.INVOICE\_DATE >= AppVariables.StartDate) &&

(invoice.INVOICE\_DATE <= AppVariables.FinishDate)

**orderby** invoice.INVOICE\_DATE descending

**select new** InvoiceView

{

Id = invoice.INVOICE\_ID,

Cusomer\_Id = invoice.CUSTOMER\_ID,

Customer = invoice.CUSTOMER.NAME,

Date = invoice.INVOICE\_DATE,

Amount = invoice.TOTAL\_SALE,

Payed = (invoice.PAYED == 1) ? "Yes" : "No"

};

masterBinding.DataSource = invoices.ToBindingList();

}

To simplify type casting, we define an InvoiceView class, rather than use some anonymous type. The definition is as follows:

**public class** InvoiceView {

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

**public int** Cusomer\_Id { **get**; **set**; }

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

**public** DateTime? Date { **get**; **set**; }

**public decimal**? Amount { **get**; **set**; }

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

**public void** Load(**int** Id) {

**var** dbContext = AppVariables.getDbContext();

**var** invoices =

**from** invoice **in** dbContext.INVOICES

**where** invoice.INVOICE\_ID == Id

**select new** InvoiceView

{

Id = invoice.INVOICE\_ID,

Cusomer\_Id = invoice.CUSTOMER\_ID,

Customer = invoice.CUSTOMER.NAME,

Date = invoice.INVOICE\_DATE,

Amount = invoice.TOTAL\_SALE,

Payed = (invoice.PAYED == 1) ? "Yes" : "No"

};

InvoiceView invoiceView = invoices.ToList().First();

**this**.Id = invoiceView.Id;

**this**.Cusomer\_Id = invoiceView.Cusomer\_Id;

**this**.Customer = invoiceView.Customer;

**this**.Date = invoiceView.Date;

**this**.Amount = invoiceView.Amount;

**this**.Payed = invoiceView.Payed;

}

}

The Load method allows us to update one added or updated record in the grid quickly, instead of completely reloading all records.

Here is the code of the event handler for clicking the Add button

**private void** btnAddInvoice\_Click(**object** sender, EventArgs e) {

var dbContext = AppVariables.getDbContext();

var invoice = dbContext.INVOICES.Create();

**using** (InvoiceEditorForm editor = **new** InvoiceEditorForm()) {

editor.Text = "Add invoice";

editor.Invoice = invoice;

*// Form Close Handler*

editor.FormClosing += **delegate** (**object** fSender, FormClosingEventArgs fe) {

**if** (editor.DialogResult == DialogResult.OK) {

**try** {

*// get next sequence value*

invoice.INVOICE\_ID = dbContext.NextValueFor("GEN\_INVOICE\_ID");

*// add a record*

dbContext.INVOICES.Add(invoice);

*// trying to save the changes*

dbContext.SaveChanges();

*// add the projection to the grid list*

((InvoiceView)masterBinding.AddNew()).Load(invoice.INVOICE\_ID);

}

**catch** (Exception ex) {

*// display error*

MessageBox.Show(ex.Message, "Error");

*// Do not close the form to correct the error*

fe.Cancel = **true**;

}

}

};

*// show the modal form*

editor.ShowDialog(**this**);

}

}

In our primary modules, the similarly-named method called dbContext.Refresh but, here, a record is updated by calling the Load method of the InvoiceView class. The reason for the difference is that dbContext.Refresh is used to update entity objects, not the objects that can be produced by complex LINQ queries.

The code of the event handler for clicking the Edit button:

**private void** btnEditInvoice\_Click(**object** sender, EventArgs e) {

var dbContext = AppVariables.getDbContext();

*// find entity by id*

var invoice = dbContext.INVOICES.Find(**this**.CurrentInvoice.Id);

**if** (invoice.PAYED == 1) {

MessageBox.Show("The change is not possible, the invoice has already been paid.",

"Error");

**return**;

}

**using** (InvoiceEditorForm editor = **new** InvoiceEditorForm()) {

editor.Text = "Edit invoice";

editor.Invoice = invoice;

*// Form Close Handler*

editor.FormClosing += **delegate** (**object** fSender, FormClosingEventArgs fe) {

**if** (editor.DialogResult == DialogResult.OK) {

**try** {

*// trying to save the changes*

dbContext.SaveChanges();

*// refresh*

CurrentInvoice.Load(invoice.INVOICE\_ID);

masterBinding.ResetCurrentItem();

}

**catch** (Exception ex) {

*// display error*

MessageBox.Show(ex.Message, "Error");

*// Do not close the form to correct the error*

fe.Cancel = **true**;

}

}

};

editor.ShowDialog(**this**);

}

}

Here we needed to find an entity by the identifier provided in the current record. The CurrentInvoice is used to retrieve the invoice selected in the grid. This is how we code it:

**public** InvoiceView CurrentInvoice {

**get** {

**return** (InvoiceView)masterBinding.Current;

}

}

Using the same approach, you can implement deleting the invoice header yourself.

#### Paying an invoice

Besides adding, editing and deleting, we want one more operation for invoices: payment. Here is code for a method to implement this operation:

**private void** btnInvoicePay\_Click(**object** sender, EventArgs e) {

var dbContext = AppVariables.getDbContext();

var invoice = dbContext.INVOICES.Find(**this**.CurrentInvoice.Id);

**try** {

**if** (invoice.PAYED == 1)

**throw new** Exception("The change is not possible, the invoice has already been paid.");

invoice.PAYED = 1;

*// trying to save the changes*

dbContext.SaveChanges();

*// refresh record*

CurrentInvoice.Load(invoice.INVOICE\_ID);

masterBinding.ResetCurrentItem();

}

**catch** (Exception ex) {

*// display error*

MessageBox.Show(ex.Message, "Error");

}

}

#### Showing the Invoice Lines

We have two choices for displaying the invoice lines:

1. Getting data for each invoice from the INVOICE\_LINE navigation property and displaying the contents of this complex property in the detail grid, probably with LINQ transformations.
2. Getting the data for each invoice with a separate LINQ query that will be re-executed when the cursor moves to another record in the master grid

Either way has its advantages and drawbacks.

The first one assumes that we want to retrieve all invoices at once for the specified period together with the bound data from the invoice lines when the invoice form is opened. Although it is done with one SQL query, it may take quite a while and requires a large amount of random-access memory. It is better suited to web applications where records are usually displayed page by page.

The second one is a bit more difficult to implement, but it allows the invoice form to be opened quickly and requires less resource. However, each time the cursor in the master grid moves, an SQL query will be executed, generating network traffic, albeit with only a small volume of data.. .

For our application we will use the second approach. We need an event handler for the BindingSource component for editing the current record;

**private void** masterBinding\_CurrentChanged(**object** sender, EventArgs e) {

LoadInvoiceLineData(**this**.CurrentInvoice.Id);

detailGridView.DataSource = detailBinding;

}

Now, the method for loading the invoice data:

**private void** LoadInvoiceLineData(**int**? id) {

var dbContext = AppVariables.getDbContext();

var lines =

**from** line **in** dbContext.INVOICE\_LINES

**where** line.INVOICE\_ID == id

**select new** InvoiceLineView

{

Id = line.INVOICE\_LINE\_ID,

Invoice\_Id = line.INVOICE\_ID,

Product\_Id = line.PRODUCT\_ID,

Product = line.PRODUCT.NAME,

Quantity = line.QUANTITY,

Price = line.SALE\_PRICE,

Total = Math.Round(line.QUANTITY \* line.SALE\_PRICE, 2)

};

detailBinding.DataSource = lines.ToBindingList();

}

We use the InvoiceLineView class as an extension:

**public class** InvoiceLineView {

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

**public int** Invoice\_Id { **get**; **set**; }

**public int** Product\_Id { **get**; **set**; }

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

**public decimal** Quantity { **get**; **set**; }

**public decimal** Price { **get**; **set**; }

**public decimal** Total { **get**; **set**; }

}

Note

Unlike the InvoiceView class, this one has no method for loading one current record. In our example, the speed of reloading the detail grid it is not crucial, because one document does not contain thousands of items. Implementing this method is optional.

Now we will add a special property for retrieving the current line of the document selected in the detail grid.

**public** InvoiceLineView CurrentInvoiceLine {

**get** {

**return** (InvoiceLineView)detailBinding.Current;

}

}

The methods we will use for adding, editing and deleting illustrate how to work with stored procedures in Entity Framework. As an example, this is the method for adding a new record:

**private void** btnAddInvoiceLine\_Click(**object** sender, EventArgs e) {

var dbContext = AppVariables.getDbContext();

*// get current invoice*

var invoice = dbContext.INVOICES.Find(**this**.CurrentInvoice.Id);

**if** (invoice.PAYED == 1) {

MessageBox.Show("The change is not possible, the invoice has already been paid.", "Error");

**return**;

}

*// create invoice position*

var invoiceLine = dbContext.INVOICE\_LINES.Create();

invoiceLine.INVOICE\_ID = invoice.INVOICE\_ID;

*// create the position editor of the invoice*

**using** (InvoiceLineEditorForm editor = **new** InvoiceLineEditorForm()) {

editor.Text = "Add invoice line";

editor.InvoiceLine = invoiceLine;

*// Form Close Handler*

editor.FormClosing += **delegate** (**object** fSender, FormClosingEventArgs fe) {

**if** (editor.DialogResult == DialogResult.OK) {

**try** {

*// create SP parameters*

var invoiceIdParam = **new** FbParameter("INVOICE\_ID",

FbDbType.Integer);

var productIdParam = **new** FbParameter("PRODUCT\_ID",

FbDbType.Integer);

var quantityParam = **new** FbParameter("QUANTITY", FbDbType.Integer);

*// initial parameters values*

invoiceIdParam.Value = invoiceLine.INVOICE\_ID;

productIdParam.Value = invoiceLine.PRODUCT\_ID;

quantityParam.Value = invoiceLine.QUANTITY;

*// execute stored procedure*

dbContext.Database.ExecuteSqlCommand(

"EXECUTE PROCEDURE SP\_ADD\_INVOICE\_LINE("

+ "@INVOICE\_ID, @PRODUCT\_ID, @QUANTITY)",

invoiceIdParam,

productIdParam,

quantityParam);

*// refresh grids*

*// reload current invoice record*

CurrentInvoice.Load(invoice.INVOICE\_ID);

*// reload all record in detail grid*

LoadInvoiceLineData(invoice.INVOICE\_ID);

*// refresh all related data*

masterBinding.ResetCurrentItem();

}

**catch** (Exception ex) {

*// display error*

MessageBox.Show(ex.Message, "Error");

*// Do not close the form to correct the error*

fe.Cancel = **true**;

}

}

};

editor.ShowDialog(**this**);

}

}

With our example, an update of the master grid record will be needed because one of its fields (TotalSale) contains aggregated information derived from the detail lines. This is how we do that:**private void** btnEditInvoiceLine\_Click(**object** sender, EventArgs e) {

var dbContext = AppVariables.getDbContext();

*// get current invoice*

var invoice = dbContext.INVOICES.Find(**this**.CurrentInvoice.Id);

**if** (invoice.PAYED == 1) {

MessageBox.Show("The change is not possible, the invoice has already been paid.",

"Error");

**return**;

}

*// get current invoice position*

var invoiceLine = invoice.INVOICE\_LINES

.Where(p => p.INVOICE\_LINE\_ID == **this**.CurrentInvoiceLine.Id)

.First();

*// create invoice position editor*

**using** (InvoiceLineEditorForm editor = **new** InvoiceLineEditorForm()) {

editor.Text = "Edit invoice line";

editor.InvoiceLine = invoiceLine;

*// form close handler*

editor.FormClosing += **delegate** (**object** fSender, FormClosingEventArgs fe) {

**if** (editor.DialogResult == DialogResult.OK) {

**try** {

*// create parameters*

var idParam = **new** FbParameter("INVOICE\_LINE\_ID", FbDbType.Integer);

var quantityParam = **new** FbParameter("QUANTITY", FbDbType.Integer);

*// initial parameters values*

idParam.Value = invoiceLine.INVOICE\_LINE\_ID;

quantityParam.Value = invoiceLine.QUANTITY;

*// execute stored procedure*

dbContext.Database.ExecuteSqlCommand(

"EXECUTE PROCEDURE SP\_EDIT\_INVOICE\_LINE("

+ "@INVOICE\_LINE\_ID, @QUANTITY)",

idParam,

quantityParam);

*// refresh grids*

*// reload current invoice record*

CurrentInvoice.Load(invoice.INVOICE\_ID);

*// reload all records in detail grid*

LoadInvoiceLineData(invoice.INVOICE\_ID);

*// refresh all related controls*

masterBinding.ResetCurrentItem();

}

**catch** (Exception ex) {

*// display error*

MessageBox.Show(ex.Message, "Error");

*// Do not close the form to correct the error*

fe.Cancel = **true**;

}

}

};

editor.ShowDialog(**this**);

}

}

The method for deleting a record is implemented as follows:

**private void** btnDeleteInvoiceLine\_Click(**object** sender, EventArgs e) {

var result = MessageBox.Show(

" Are you sure you want to delete the invoice item?",

"Confirmation",

MessageBoxButtons.YesNo,

MessageBoxIcon.Question);

**if** (result == DialogResult.Yes) {

var dbContext = AppVariables.getDbContext();

*// get current invoice*

var invoice = dbContext.INVOICES.Find(**this**.CurrentInvoice.Id);

**try** {

**if** (invoice.PAYED == 1)

**throw new** Exception("It is not possible to delete the entry, the invoice is paid.");

*// create parameters*

var idParam = **new** FbParameter("INVOICE\_LINE\_ID", FbDbType.Integer);

*// initialize parameters values*

idParam.Value = **this**.CurrentInvoiceLine.Id;

*// execute stored procedure*

dbContext.Database.ExecuteSqlCommand(

"EXECUTE PROCEDURE SP\_DELETE\_INVOICE\_LINE(@INVOICE\_LINE\_ID)",

idParam);

*// update grids*

*// reload current invoice*

CurrentInvoice.Load(invoice.INVOICE\_ID);

*// reload all records in detail grids*

LoadInvoiceLineData(invoice.INVOICE\_ID);

*// refresh related controls*

masterBinding.ResetCurrentItem();

}

**catch** (Exception ex) {

*// display error*

MessageBox.Show(ex.Message, "Error");

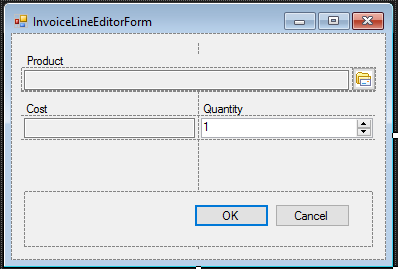
}

}

}

### Showing Products for Selection

In the methods for adding and editing invoice lines we used the form. a control



A click on the button next to the TextBox will open a modal form with a grid for selecting products. The same modal form created for displaying the products is used for selecting them. The click handler code for the embedded button that initiates the form is:**public partial class** InvoiceLineEditorForm : Form {

**public** InvoiceLineEditorForm() {

InitializeComponent();

}

**public** INVOICE\_LINE InvoiceLine { **get**; **set**; }

**private void** InvoiceLineEditorForm\_Load(**object** sender, EventArgs e) {

**if** (**this**.InvoiceLine.PRODUCT != **null**) {

edtProduct.Text = **this**.InvoiceLine.PRODUCT.NAME;

edtPrice.Text = **this**.InvoiceLine.PRODUCT.PRICE.ToString("F2");

btnChooseProduct.Click -= **this**.btnChooseProduct\_Click;

}

**if** (**this**.InvoiceLine.QUANTITY == 0)

**this**.InvoiceLine.QUANTITY = 1;

edtQuantity.DataBindings.Add("Value", **this**.InvoiceLine, "QUANTITY");

}

**private void** btnChooseProduct\_Click(**object** sender, EventArgs e) {

GoodsForm goodsForm = **new** GoodsForm();

**if** (goodsForm.ShowDialog() == DialogResult.OK) {

InvoiceLine.PRODUCT\_ID = goodsForm.CurrentProduct.Id;

edtProduct.Text = goodsForm.CurrentProduct.Name;

edtPrice.Text = goodsForm.CurrentProduct.Price.ToString("F2");

}

}

}

### Working with Transactions

Whenever we call the SaveChanges() method while adding, updating or deleting, Entity Framework starts and ends an implicit transaction. Since we use disconnected data access, all operations are carried out within one transaction. Entity Framework starts and ends a transaction automatically for each data retrieval. We will take the following example to illustrate how automatic transactions work.

Suppose we need to make a discount on goods selected in the grid. Without explicit transaction management, the code would be as follows:

var dbContext = AppVariables.getDbContext();

**foreach** (DataGridViewRow gridRows **in** dataGridView.SelectedRows) {

**int** id = (**int**)gridRows.Cells["Id"].Value;

*// here there is an implicit start and the completion of the transaction*

var product = dbContext.PRODUCTS.Find(id);

*// discount 10%*

**decimal** discount = 10.0m;

product.PRICE = product.PRICE \* (100 - discount) /100;

}

*// here there is an implicit start and the completion of the transaction*

*// all changes occur in one transaction*

dbContext.SaveChanges();

Let's say we select 10 products. Ten implicit transactions will be used for finding the products by their identifiers. One more transaction will be used to save the changes.

If we control transactions explicitly, we can use just one transaction for the same piece of work. For example:

var dbContext = AppVariables.getDbContext();

*// explicit start of a default transaction*

**using** (var dbTransaction = dbContext.Database.BeginTransaction()) {

**string** sql =

"UPDATE PRODUCT " +

"SET PRICE = PRICE \* ROUND((100 - @DISCOUNT)/100, 2) " +

"WHERE PRODUCT\_ID = @PRODUCT\_ID";

**try** {

*// create query parameters*

var idParam = **new** FbParameter("PRODUCT\_ID", FbDbType.Integer);

var discountParam = **new** FbParameter("DISCOUNT", FbDbType.Decimal);

*// create a SQL command to update records*

var sqlCommand = dbContext.Database.Connection.CreateCommand();

sqlCommand.CommandText = sql;

*// specify which transaction to use*

sqlCommand.Transaction = dbTransaction.UnderlyingTransaction;

sqlCommand.Parameters.Add(discountParam);

sqlCommand.Parameters.Add(idParam);

*// prepare query*

sqlCommand.Prepare();

*// for all selected records in the grid*

**foreach** (DataGridViewRow gridRows **in** dataGridView.SelectedRows) {

**int** id = (**int**)gridRows.Cells["Id"].Value;

*// initialize query parameters*

idParam.Value = id;

discountParam.Value = 10.0m; *// discount 10%*

*// execute sql statement*

sqlCommand.ExecuteNonQuery();

}

dbTransaction.Commit();

}

**catch** (Exception ex) {

dbTransaction.Rollback();

MessageBox.Show(ex.Message, "error");

}

}

Our code starts the transaction with the default parameters. To specify your own parameters for a transaction, you should use the UseTransaction method.

**private void** btnDiscount\_Click(**object** sender, EventArgs e) {

DiscountEditorForm editor = **new** DiscountEditorForm();

editor.Text = "Enter discount";

**if** (editor.ShowDialog() != DialogResult.OK)

**return**;

**bool** needUpdate = **false**;

var dbContext = AppVariables.getDbContext();

var connection = dbContext.Database.Connection;

*// explicit start of transaction*

**using** (var dbTransaction = connection.BeginTransaction(IsolationLevel.Snapshot)) {

dbContext.Database.UseTransaction(dbTransaction);

**string** sql =

"UPDATE PRODUCT " +

"SET PRICE = ROUND(PRICE \* (100 - @DISCOUNT)/100, 2) " +

"WHERE PRODUCT\_ID = @PRODUCT\_ID";

**try** {

*// create query parameters*

var idParam = **new** FbParameter("PRODUCT\_ID", FbDbType.Integer);

var discountParam = **new** FbParameter("DISCOUNT", FbDbType.Decimal);

*// create a SQL command to update records*

var sqlCommand = connection.CreateCommand();

sqlCommand.CommandText = sql;

*// specify which transaction to use*

sqlCommand.Transaction = dbTransaction;

sqlCommand.Parameters.Add(discountParam);

sqlCommand.Parameters.Add(idParam);

*// prepare statement*

sqlCommand.Prepare();

*// for all selected records in the grid*

**foreach** (DataGridViewRow gridRows **in** dataGridView.SelectedRows) {

**int** id = (**int**)gridRows.Cells["PRODUCT\_ID"].Value;

*// initialize query parameters*

idParam.Value = id;

discountParam.Value = editor.Discount;

*// execute SQL statement*

needUpdate = (sqlCommand.ExecuteNonQuery() > 0) || needUpdate;

}

dbTransaction.Commit();

}

**catch** (Exception ex) {

dbTransaction.Rollback();

MessageBox.Show(ex.Message, "error");

needUpdate = **false**;

}

}

*// refresh grid*

**if** (needUpdate) {

*// for all selected records in the grid*

**foreach** (DataGridViewRow gridRows **in** dataGridView.SelectedRows) {

var product = (PRODUCT)bindingSource.List[gridRows.Index];

dbContext.Refresh(RefreshMode.StoreWins, product);

}

bindingSource.ResetBindings(**false**);

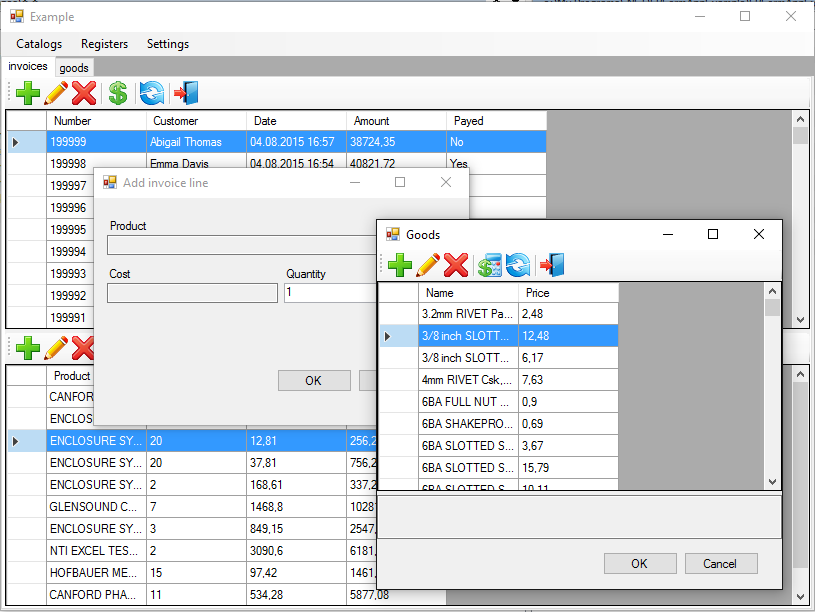
}

}

That's it. Now only one transaction is used for the entire set of updates and there are no unnecessary commands for finding data.

All that is left to do is to add a dialog box for entering the value of the discount and code to update the data in the grid. Try to do it on your own.

# The Result



## Source Code

You can get the source code for the sample application using this link: <https://github.com/sim1984/FBFormAppExample>