Databinding with WinForms

2/20/2020 • 13 minutes to read • Edit Online

This step-by-step walkthrough shows how to bind POCO types to Window Forms (WinForms) controls in a "master-detail" form. The application uses Entity Framework to populate objects with data from the database, track changes, and persist data to the database.

The model defines two types that participate in one-to-many relationship: Category (principal\master) and Product (dependent\detail). Then, the Visual Studio tools are used to bind the types defined in the model to the WinForms controls. The WinForms data-binding framework enables navigation between related objects: selecting rows in the master view causes the detail view to update with the corresponding child data.

The screen shots and code listings in this walkthrough are taken from Visual Studio 2013 but you can complete this walkthrough with Visual Studio 2012 or Visual Studio 2010.

Pre-Requisites

You need to have Visual Studio 2013, Visual Studio 2012 or Visual Studio 2010 installed to complete this walkthrough.

If you are using Visual Studio 2010, you also have to install NuGet. For more information, see Installing NuGet.

Create the Application

- Open Visual Studio
- **File -> New -> Project….**
- Select **Windows** in the left pane and **Windows FormsApplication** in the right pane
- Enter **WinFormswithEFSample** as the name
- Select **OK**

Install the Entity Framework NuGet package

- In Solution Explorer, right-click on the **WinFormswithEFSample** project
- Select **Manage NuGet Packages…**
- In the Manage NuGet Packages dialog, Select the **Online** tab and choose the **EntityFramework** package
- Click **Install**

> **NOTE**
> In addition to the EntityFramework assembly a reference to System.ComponentModel.DataAnnotations is also added. If the project has a reference to System.Data.Entity, then it will be removed when the EntityFramework package is installed. The System.Data.Entity assembly is no longer used for Entity Framework 6 applications.

Implementing IListSource for Collections

Collection properties must implement the IListSource interface to enable two-way data binding with sorting when using Windows Forms. To do this we are going to extend ObservableCollection to add IListSource functionality.

- Add an **ObservableListSource** class to the project:

  - Select **Add -> New Item**
  - Select **Class** and enter **ObservableListSource** for the class name
- Replace the code generated by default with the following code:

This class enables two-way data binding as well as sorting. The class derives from ObservableCollection<T> and adds an explicit implementation of IListSource. The GetList() method of IListSource is implemented to return an IBindingList implementation that stays in sync with the ObservableCollection. The IBindingList implementation generated by ToBindingList supports sorting. The ToBindingList extension method is defined in the EntityFramework assembly.

```
    using System.Collections;
    using System.Collections.Generic;
    using System.Collections.ObjectModel;
    using System.ComponentModel;
    using System.Diagnostics.CodeAnalysis;
    using System.Data.Entity;

    namespace WinFormswithEFSample
    {
        public class ObservableListSource<T> : ObservableCollection<T>, IListSource
            where T : class
        {
            private IBindingList _bindingList;

            bool IListSource.ContainsListCollection { get { return false; } }

            IList IListSource.GetList()
            {
                return _bindingList ?? (_bindingList = this.ToBindingList());
            }
        }
    }
```

Define a Model

In this walkthrough you can chose to implement a model using Code First or the EF Designer. Complete one of the two following sections.

Option 1: Define a Model using Code First

This section shows how to create a model and its associated database using Code First. Skip to the next section (**Option 2: Define a model using Database First)** if you would rather use Database First to reverse engineer your model from a database using the EF designer

When using Code First development you usually begin by writing .NET Framework classes that define your conceptual (domain) model.

- Add a new **Product** class to project
- Replace the code generated by default with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace WinFormswithEFSample
{
    public class Product
    {
        public int ProductId { get; set; }
        public string Name { get; set; }

        public int CategoryId { get; set; }
        public virtual Category Category { get; set; }
    }
}
```

- Add a **Category** class to the project.
- Replace the code generated by default with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace WinFormswithEFSample
{
    public class Category
    {
        private readonly ObservableListSource<Product> _products =
                new ObservableListSource<Product>();

        public int CategoryId { get; set; }
        public string Name { get; set; }
        public virtual ObservableListSource<Product> Products { get { return _products; } }
    }
}
```

In addition to defining entities, you need to define a class that derives from **DbContext** and exposes **DbSet<TEntity>** properties. The **DbSet** properties let the context know which types you want to include in the model. The **DbContext** and **DbSet** types are defined in the EntityFramework assembly.

An instance of the DbContext derived type manages the entity objects during run time, which includes populating objects with data from a database, change tracking, and persisting data to the database.

- Add a new **ProductContext** class to the project.
- Replace the code generated by default with the following code:

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Text;

namespace WinFormswithEFSample
{
    public class ProductContext : DbContext
    {
        public DbSet<Category> Categories { get; set; }
        public DbSet<Product> Products { get; set; }
    }
}
```

Compile the project.

Option 2: Define a model using Database First

This section shows how to use Database First to reverse engineer your model from a database using the EF designer. If you completed the previous section (**Option 1: Define a model using Code First**), then skip this section and go straight to the **Lazy Loading** section.

Create an Existing Database

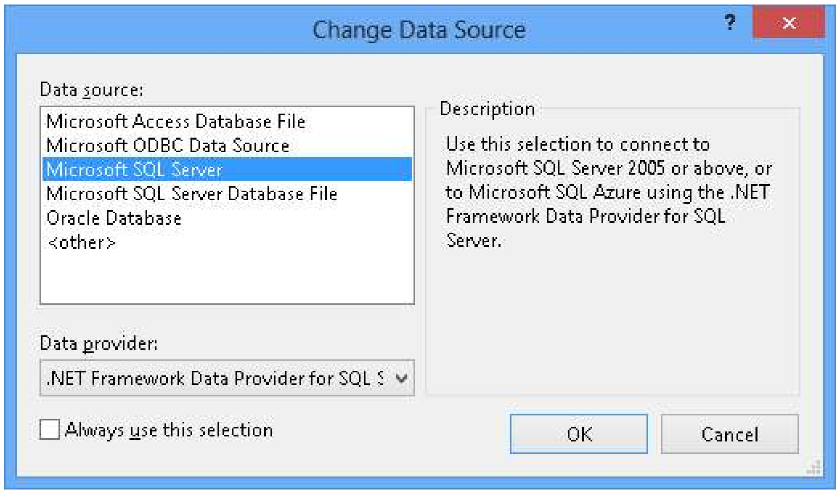
Typically when you are targeting an existing database it will already be created, but for this walkthrough we need to create a database to access.

The database server that is installed with Visual Studio is different depending on the version of Visual Studio you have installed:

- If you are using Visual Studio 2010 you'll be creating a SQL Express database.
- If you are using Visual Studio 2012 then you'll be creating a LocalDB database.

Let's go ahead and generate the database.

- **View -> Server Explorer**
- Right click on **Data Connections -> Add Connection…**
- If you haven't connected to a database from Server Explorer before you'll need to select Microsoft SQL Server as the data source

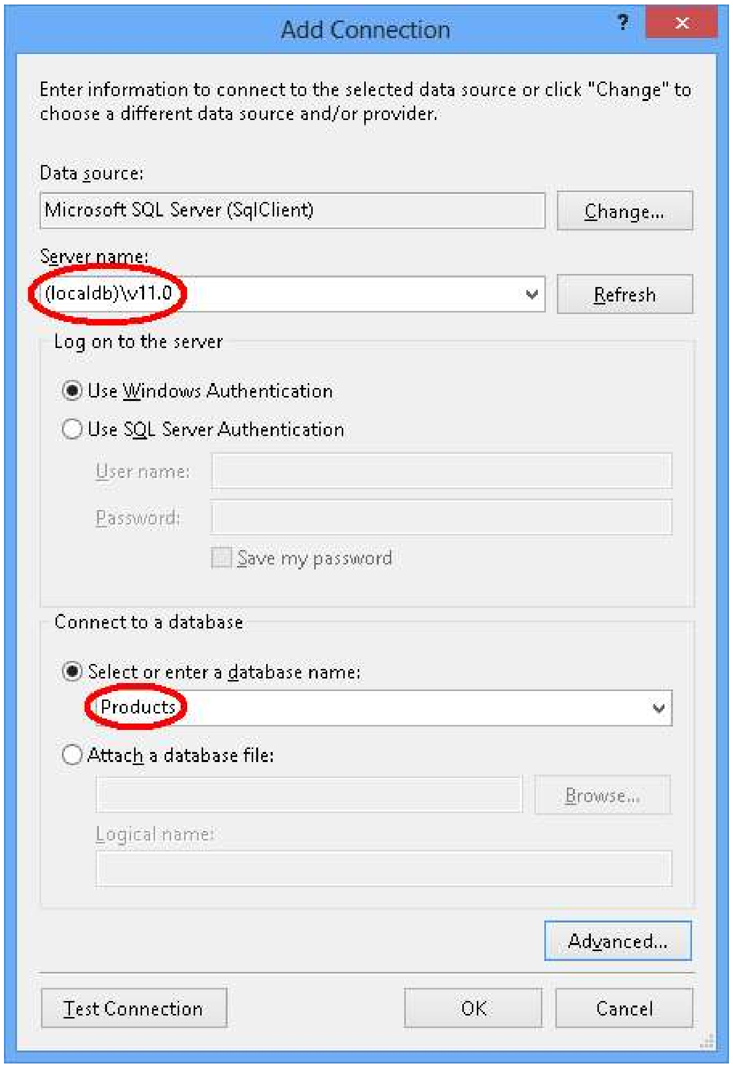

Change Data Source

Data source:
Microsoft Access Database File
Microsoft ODBC Data Source
Microsoft SQL Server
Microsoft SQL Server Database File
Oracle Database
<other>

Description
Use this selection to connect to Microsoft SQL Server 2005 or above, or to Microsoft SQL Azure using the .NET Framework Data Provider for SQL Server.

Data provider:
.NET Framework Data Provider for SQL S

- Connect to either LocalDB or SQL Express, depending on which one you have installed, and enter **Products** as the database name



Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

Data source: Microsoft SQL Server (SqlClient)

Server name: (localdb)\v11.0

Log on to the server

(●) Use Windows Authentication

( ) Use SQL Server Authentication

User name:

Password:

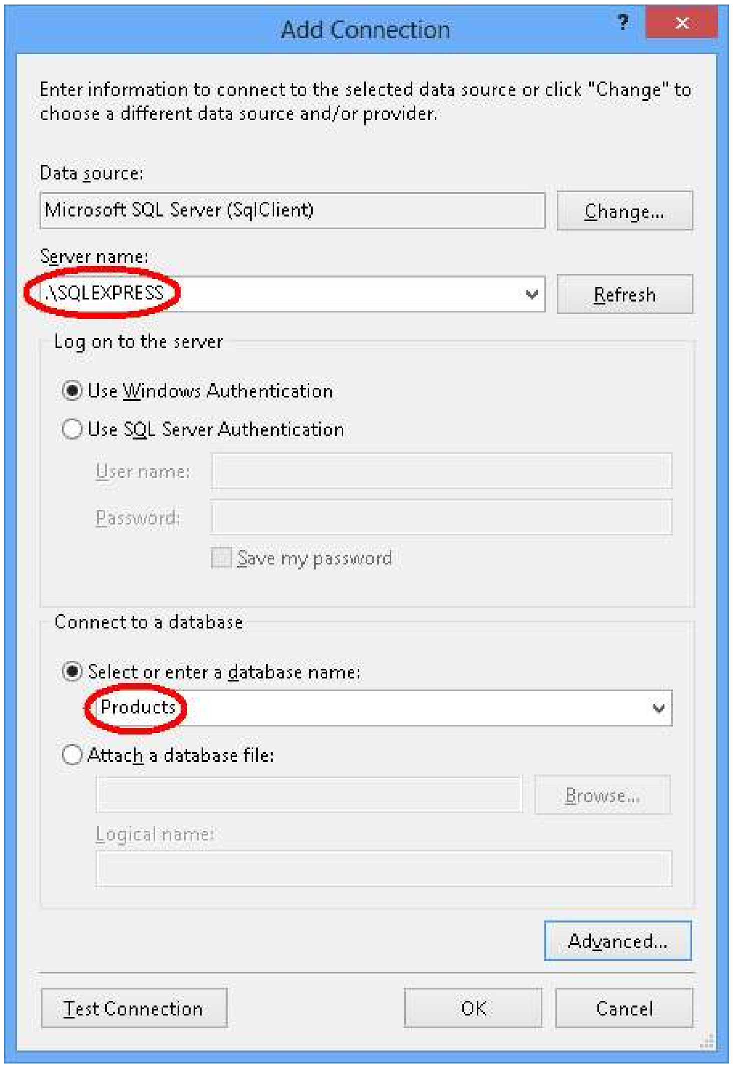
☐ Save my password

Connect to a database

(●) Select or enter a database name: Products

( ) Attach a database file:

Logical name:


Add Connection

Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

Data source: Microsoft SQL Server (SqlClient)

Server name: .\SQLEXPRESS

Log on to the server

● Use Windows Authentication

○ Use SQL Server Authentication

User name:

Password:

☐ Save my password

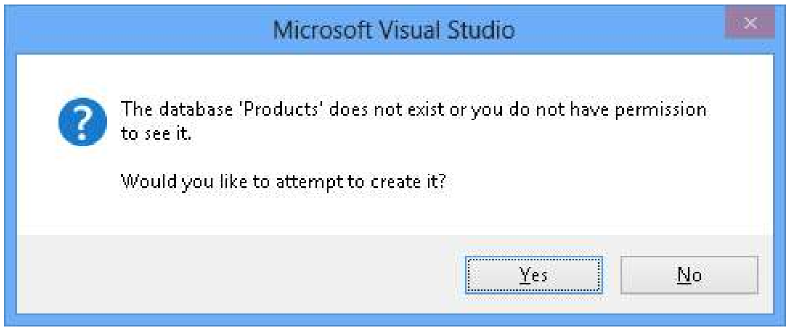
Connect to a database

● Select or enter a database name: Products

○ Attach a database file:

Logical name:

- Select **OK** and you will be asked if you want to create a new database, select **Yes**


Microsoft Visual Studio

The database 'Products' does not exist or you do not have permission to see it.

Would you like to attempt to create it?

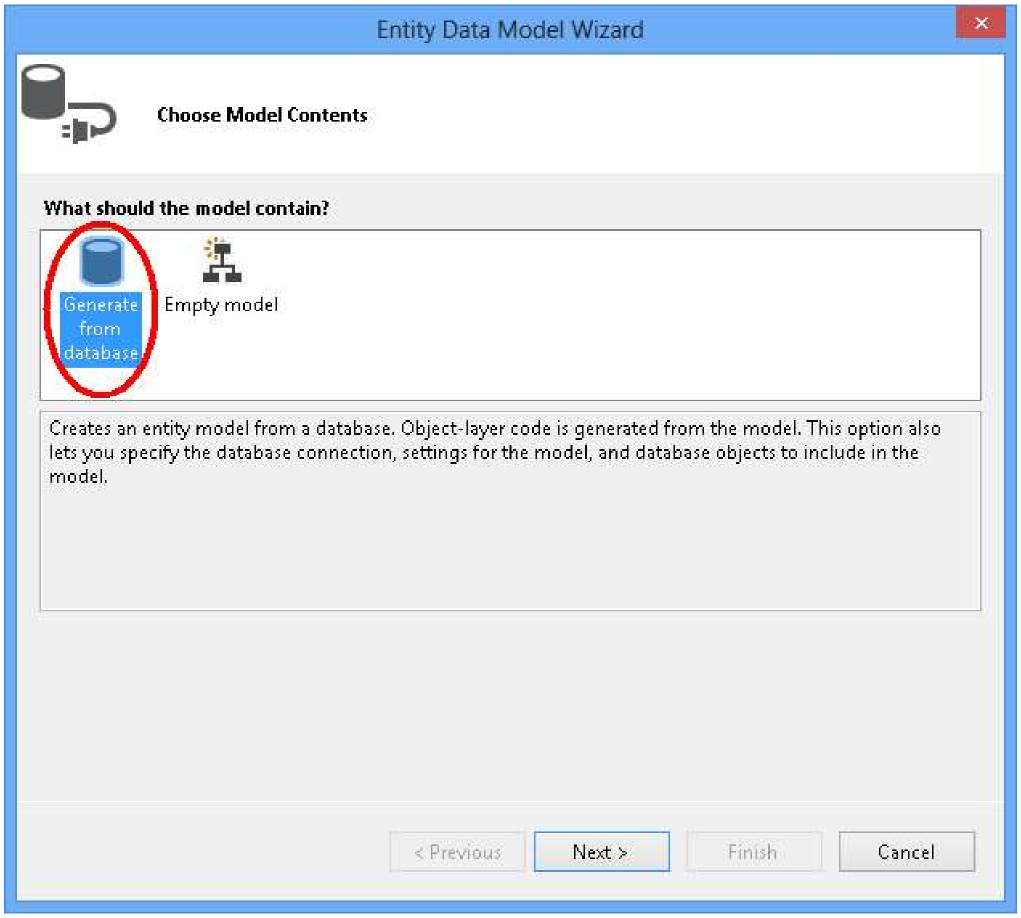
- The new database will now appear in Server Explorer, right-click on it and select **New Query**
- Copy the following SQL into the new query, then right-click on the query and select **Execute**

```
    CREATE TABLE [dbo].[Categories] (
        [CategoryId] [int] NOT NULL IDENTITY,
        [Name] [nvarchar](max),
        CONSTRAINT [PK_dbo.Categories] PRIMARY KEY ([CategoryId])
    )

    CREATE TABLE [dbo].[Products] (
        [ProductId] [int] NOT NULL IDENTITY,
        [Name] [nvarchar](max),
        [CategoryId] [int] NOT NULL,
        CONSTRAINT [PK_dbo.Products] PRIMARY KEY ([ProductId])
    )

    CREATE INDEX [IX_CategoryId] ON [dbo].[Products]([CategoryId])

    ALTER TABLE [dbo].[Products] ADD CONSTRAINT [FK_dbo.Products_dbo.Categories_CategoryId] FOREIGN KEY
([CategoryId]) REFERENCES [dbo].[Categories] ([CategoryId]) ON DELETE CASCADE
```

Reverse Engineer Model

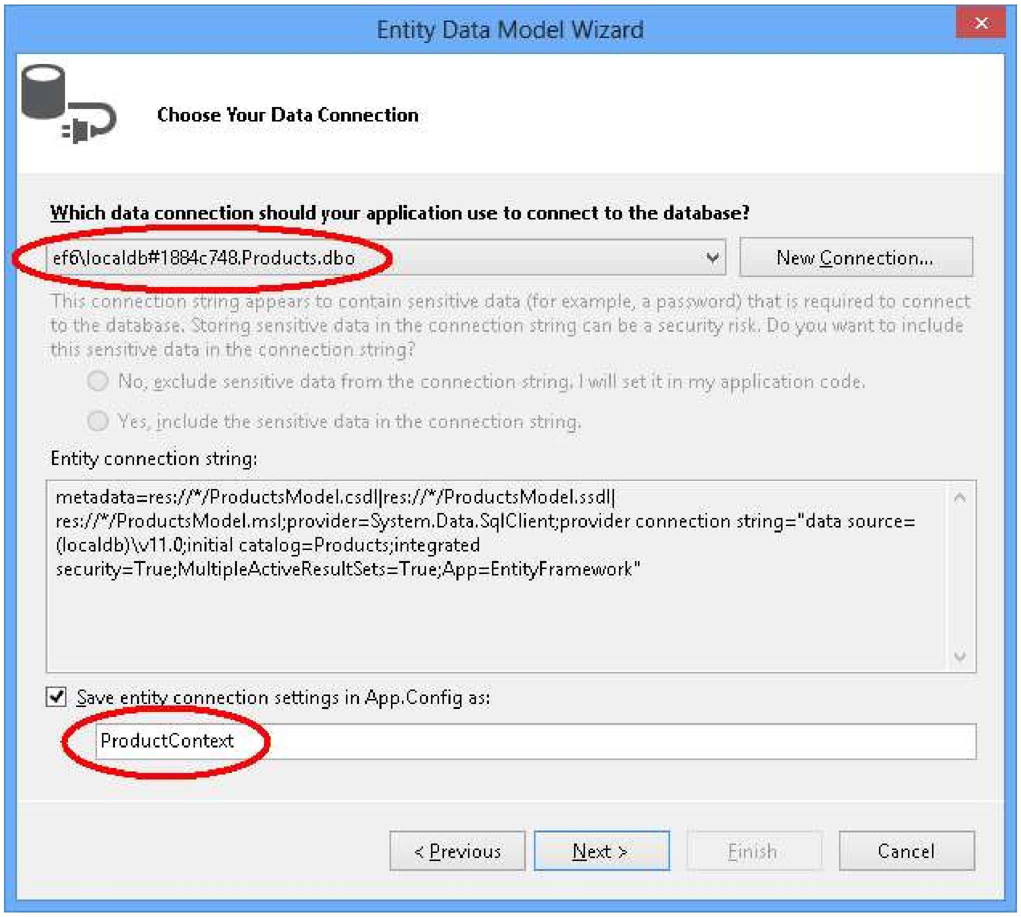
We're going to make use of Entity Framework Designer, which is included as part of Visual Studio, to create our model.

- **Project -> Add New Item…**
- Select **Data** from the left menu and then **ADO.NET Entity Data Model**
- Enter **ProductModel** as the name and click **OK**
- This launches the **Entity Data Model Wizard**
- Select **Generate from Database** and click **Next**



Creates an entity model from a database. Object-layer code is generated from the model. This option also lets you specify the database connection, settings for the model, and database objects to include in the model.

- Select the connection to the database you created in the first section, enter **ProductContext** as the name of



Which data connection should your application use to connect to the database?

ef6\localdb#1884c748.Products.dbo

This connection string appears to contain sensitive data (for example, a password) that is required to connect to the database. Storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

○ No, exclude sensitive data from the connection string. I will set it in my application code.

○ Yes, include the sensitive data in the connection string.

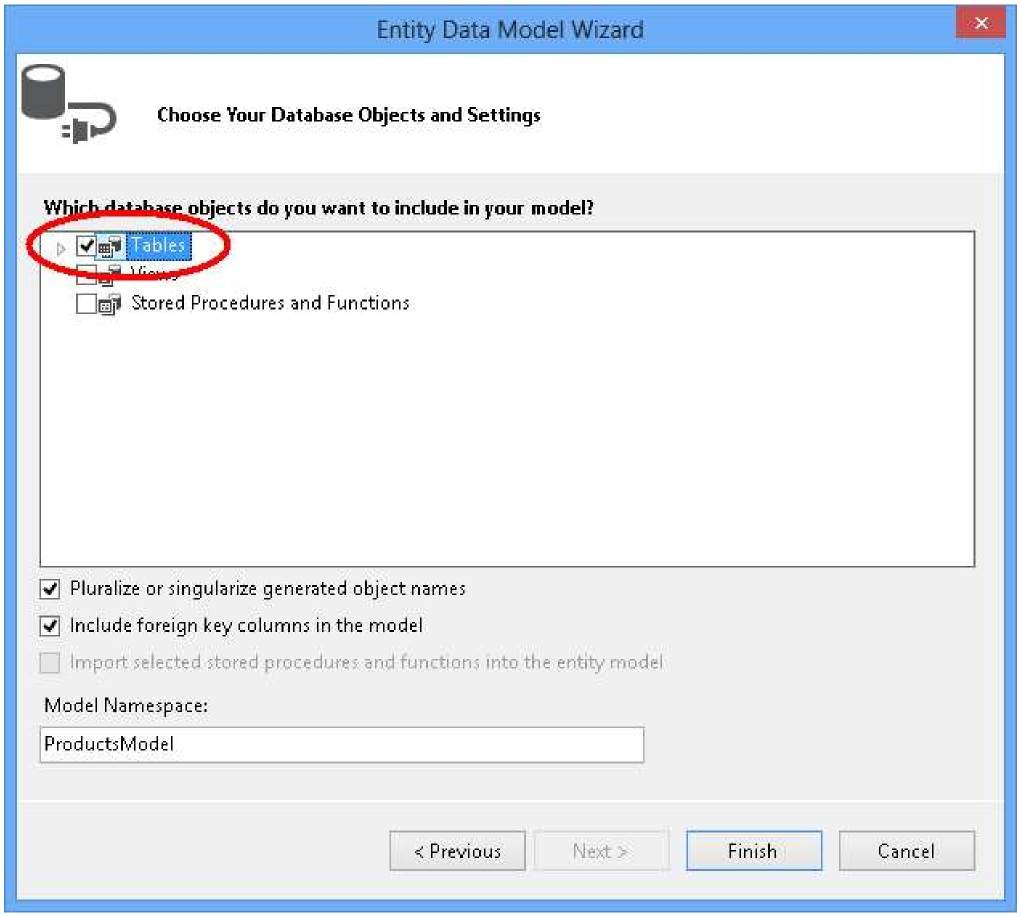
Entity connection string:

```
metadata=res://*/ProductsModel.csdl|res://*/ProductsModel.ssdl|
res://*/ProductsModel.msl;provider=System.Data.SqlClient;provider connection string="data source=
(localdb)\v11.0;initial catalog=Products;integrated
security=True;MultipleActiveResultSets=True;App=EntityFramework"
```

☑ Save entity connection settings in App.Config as:

ProductContext

● Click the checkbox next to ‘Tables’ to import all tables and click ‘Finish’



Which database objects do you want to include in your model?

☑ Tables
☐ Views
☐ Stored Procedures and Functions

☑ Pluralize or singularize generated object names

☑ Include foreign key columns in the model

☐ Import selected stored procedures and functions into the entity model

Model Namespace:

ProductsModel

Once the reverse engineer process completes the new model is added to your project and opened up for you to view in the Entity Framework Designer. An App.config file has also been added to your project with the connection

Additional Steps in Visual Studio 2010

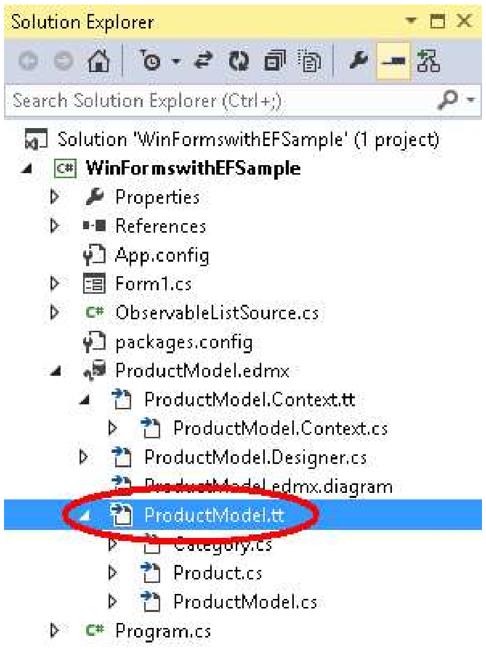
If you are working in Visual Studio 2010 then you will need to update the EF designer to use EF6 code generation.

- Right-click on an empty spot of your model in the EF Designer and select **Add Code Generation Item...**
- Select **Online Templates** from the left menu and search for **DbContext**
- Select the **EF 6.x DbContext Generator for C#,** enter **ProductsModel** as the name and click Add

Updating code generation for data binding

EF generates code from your model using T4 templates. The templates shipped with Visual Studio or downloaded from the Visual Studio gallery are intended for general purpose use. This means that the entities generated from these templates have simple ICollection<T> properties. However, when doing data binding it is desirable to have collection properties that implement IListSource. This is why we created the ObservableListSource class above and we are now going to modify the templates to make use of this class.

- Open the **Solution Explorer** and find **ProductModel.edmx** file
- Find the **ProductModel.tt** file which will be nested under the ProductModel.edmx file



- Double-click on the ProductModel.tt file to open it in the Visual Studio editor
- Find and replace the two occurrences of “**ICollection**” with “**ObservableListSource**”. These are located at approximately lines 296 and 484.
- Find and replace the first occurrence of “**HashSet**” with “**ObservableListSource**”. This occurrence is located at approximately line 50. **Do not** replace the second occurrence of HashSet found later in the code.
- Save the ProductModel.tt file. This should cause the code for entities to be regenerated. If the code does not regenerate automatically, then right click on ProductModel.tt and choose “Run Custom Tool”.

If you now open the Category.cs file (which is nested under ProductModel.tt) then you should see that the Products collection has the type **ObservableListSource<Product>**.

Compile the project.

Lazy Loading

The **Products** property on the **Category** class and **Category** property on the **Product** class are navigation types.

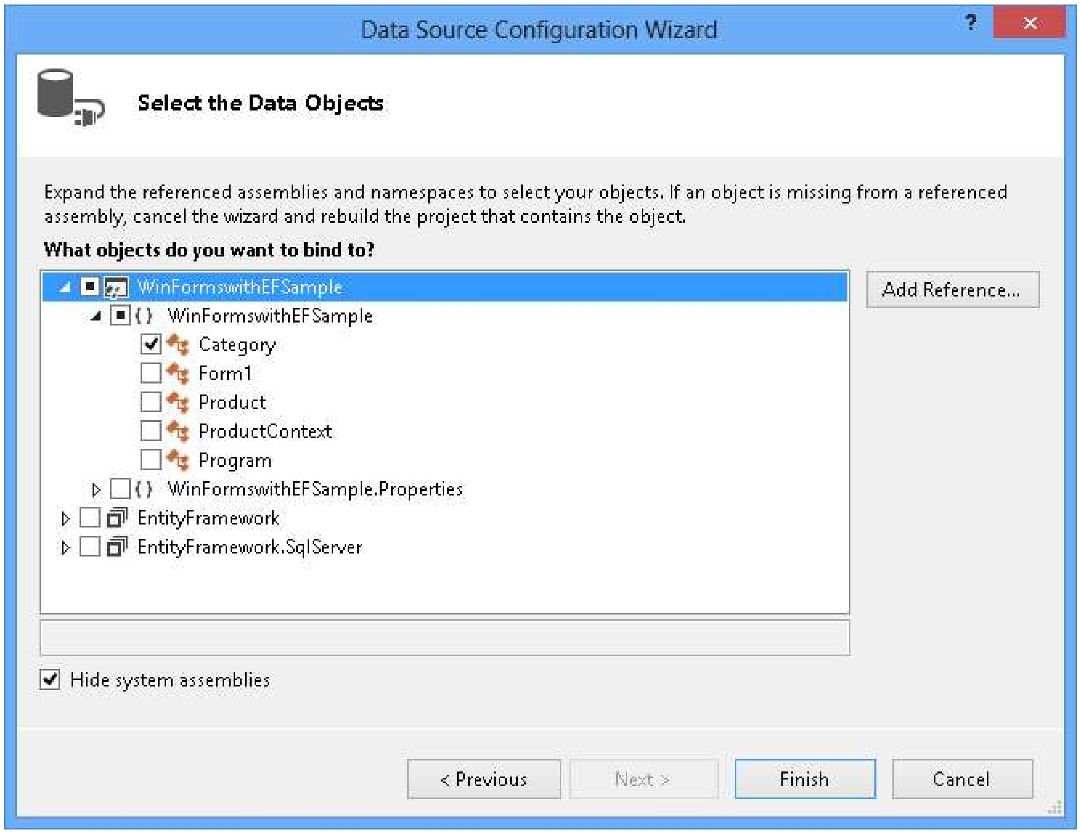
EF gives you an option of loading related entities from the database automatically the first time you access the navigation property. With this type of loading (called lazy loading), be aware that the first time you access each navigation property a separate query will be executed against the database if the contents are not already in the context.

When using POCO entity types, EF achieves lazy loading by creating instances of derived proxy types during runtime and then overriding virtual properties in your classes to add the loading hook. To get lazy loading of related objects, you must declare navigation property getters as **public** and **virtual** (**Overridable** in Visual Basic), and you class must not be **sealed** (**NotOverridable** in Visual Basic). When using Database First navigation properties are automatically made virtual to enable lazy loading. In the Code First section we chose to make the navigation properties virtual for the same reason

Bind Object to Controls

Add the classes that are defined in the model as data sources for this WinForms application.

- From the main menu, select **Project -> Add New Data Source …** (in Visual Studio 2010, you need to select **Data -> Add New Data Source…**)
- In the Choose a Data Source Type window, select **Object** and click **Next**
- In the Select the Data Objects dialog, unfold the **WinFormswithEFSample** two times and select **Category** There is no need to select the Product data source, because we will get to it through the Product's property on the Category data source.



Data Source Configuration Wizard

Select the Data Objects

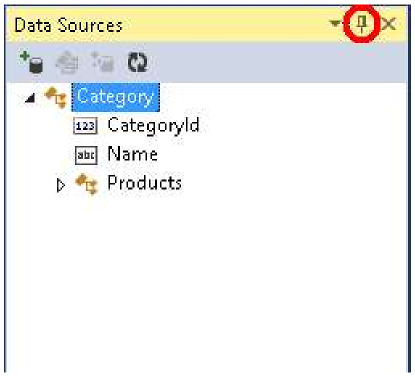
Expand the referenced assemblies and namespaces to select your objects. If an object is missing from a referenced assembly, cancel the wizard and rebuild the project that contains the object.

What objects do you want to bind to?

- WinFormswithEFSample
  - WinFormswithEFSample
    - [✓] Category
    - [ ] Form1
    - [ ] Product
    - [ ] ProductContext
    - [ ] Program
  - [ ] WinFormswithEFSample.Properties
- [ ] EntityFramework
- [ ] EntityFramework.SqlServer

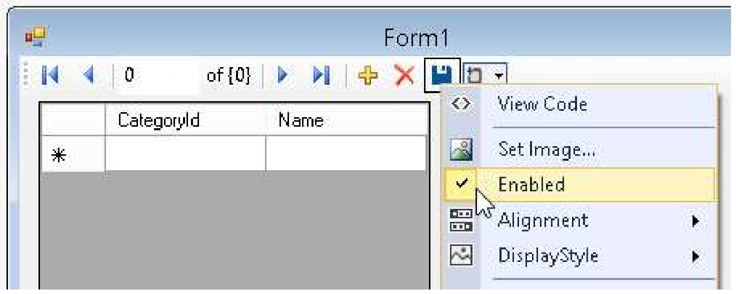
[✓] Hide system assemblies

- Click **Finish.** If the Data Sources window is not showing up, select **View -> Other Windows-> Data Sources**
- Press the pin icon, so the Data Sources window does not auto hide. You may need to hit the refresh button if the window was already visible.



- In Solution Explorer, double-click the **Form1.cs** file to open the main form in designer.
- Select the **Category** data source and drag it on the form. By default, a new DataGridView (**categoryDataGridView**) and Navigation toolbar controls are added to the designer. These controls are bound to the BindingSource (**categoryBindingSource**) and Binding Navigator (**categoryBindingNavigator**) components that are created as well.
- Edit the columns on the **categoryDataGridView**. We want to set the **CategoryId** column to read-only. The value for the **CategoryId** property is generated by the database after we save the data.
  - Right-click the DataGridView control and select Edit Columns...
  - Select the CategoryId column and set ReadOnly to True
  - Press OK
- Select Products from under the Category data source and drag it on the form. The productDataGridView and productBindingSource are added to the form.
- Edit the columns on the productDataGridView. We want to hide the CategoryId and Category columns and set ProductId to read-only. The value for the ProductId property is generated by the database after we save the data.
  - Right-click the DataGridView control and select **Edit Columns...**.
  - Select the **ProductId** column and set **ReadOnly** to **True**.
  - Select the **CategoryId** column and press the **Remove** button. Do the same with the **Category** column.
  - Press **OK**.

  So far, we associated our DataGridView controls with BindingSource components in the designer. In the next section we will add code to the code behind to set categoryBindingSource.DataSource to the collection of entities that are currently tracked by DbContext. When we dragged-and-dropped Products from under the Category, the WinForms took care of setting up the productsBindingSource.DataSource property to categoryBindingSource and productsBindingSource.DataMember property to Products. Because of this binding, only the products that belong to the currently selected Category will be displayed in the productDataGridView.
- Enable the **Save** button on the Navigation toolbar by clicking the right mouse button and selecting **Enabled**.



and bring you to the code behind for the form. The code for the **categoryBindingNavigatorSaveItem_Click** event handler will be added in the next section.

Add the Code that Handles Data Interaction

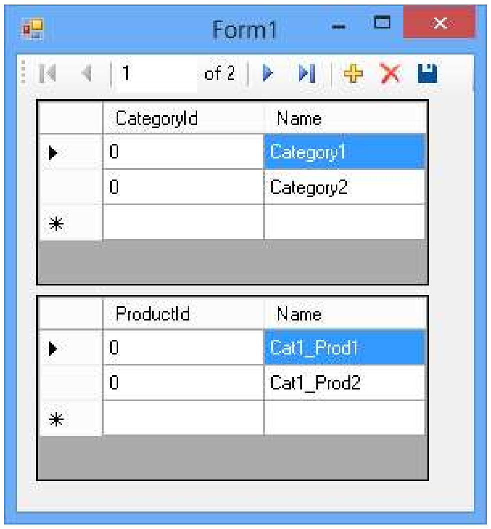
We'll now add the code to use the ProductContext to perform data access. Update the code for the main form window as shown below.

The code declares a long-running instance of ProductContext. The ProductContext object is used to query and save data to the database. The Dispose() method on the ProductContext instance is then called from the overridden OnClosing method. The code comments provide details about what the code does.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Data.Entity;

namespace WinFormswithEFSample
{
    public partial class Form1 : Form
    {
        ProductContext _context;
        public Form1()
        {
            InitializeComponent();
        }

        protected override void OnLoad(EventArgs e)
        {
            base.OnLoad(e);
            _context = new ProductContext();

            // Call the Load method to get the data for the given DbSet
            // from the database.
            // The data is materialized as entities. The entities are managed by
            // the DbContext instance.
            _context.Categories.Load();

            // Bind the categoryBindingSource.DataSource to
            // all the Unchanged, Modified and Added Category objects that
            // are currently tracked by the DbContext.
            // Note that we need to call ToBindingList() on the
            // ObservableCollection<TEntity> returned by
            // the DbSet.Local property to get the BindingList<T>
            // in order to facilitate two-way binding in WinForms.
            this.categoryBindingSource.DataSource =
                _context.Categories.Local.ToBindingList();
        }

        private void categoryBindingNavigatorSaveItem_Click(object sender, EventArgs e)
        {
            this.Validate();

            // Currently, the Entity Framework doesn’t mark the entities
            // that are removed from a navigation property (in our example the Products)
            // as deleted in the context.
            // The following code uses LINQ to Objects against the Local collection
            // to find all products and marks any that do not have
            // The ToList call is required because otherwise
            // the collection will be modified
            // by the Remove call while it is being enumerated.
            // In most other situations you can do LINQ to Objects directly
            // against the Local property without using ToList first.
            foreach (var product in _context.Products.Local.ToList())
            {
                if (product.Category == null)
                {
                    _context.Products.Remove(product);
                }
            }

            // Save the changes to the database.
            this._context.SaveChanges();

            // Refresh the controls to show the values
            // that were generated by the database.
            this.categoryDataGridView.Refresh();
            this.productsDataGridView.Refresh();
        }

        protected override void OnClosing(CancelEventArgs e)
        {
            base.OnClosing(e);
            this._context.Dispose();
        }
    }
}
```

Test the Windows Forms Application

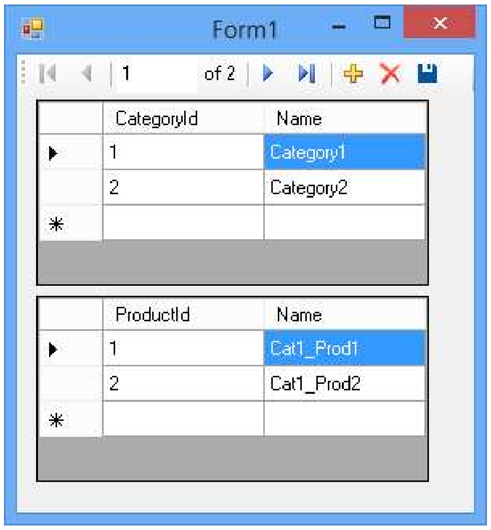
- Compile and run the application and you can test out the functionality.



	CategoryId	Name
▶	0	Category1
	0	Category2
*		

	ProductId	Name
▶	0	Cat1_Prod1
	0	Cat1_Prod2
*		

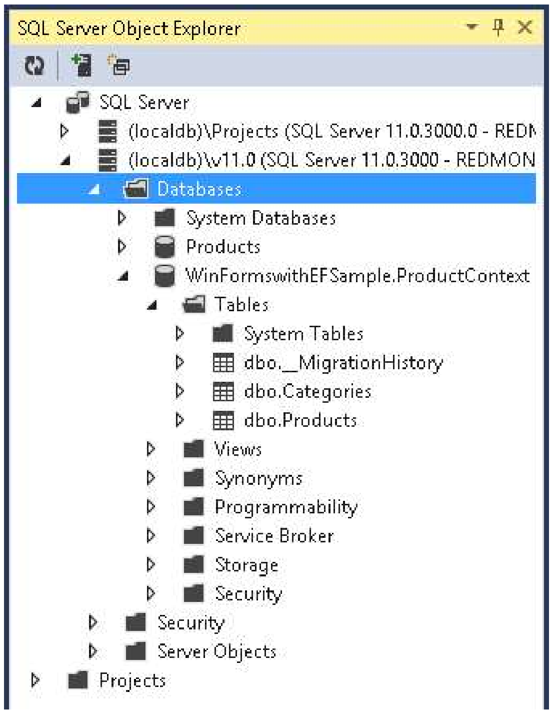
- After saving the store generated keys are shown on the screen.



	CategoryId	Name
▶	1	Category1
	2	Category2
*		

	ProductId	Name
▶	1	Cat1_Prod1
	2	Cat1_Prod2
*		

- If you used Code First, then you will also see that a **WinFormswithEFSample.ProductContext** database is created for you.


SQL Server Object Explorer
SQL Server
(localdb)\Projects (SQL Server 11.0.3000.0 - REDM
(localdb)\v11.0 (SQL Server 11.0.3000 - REDMON
Databases
System Databases
Products
WinFormswithEFSample.ProductContext
Tables
System Tables
dbo.__MigrationHistory
dbo.Categories
dbo.Products
Views
Synonyms
Programmability
Service Broker
Storage
Security
Security
Server Objects
Projects