ComponentOne

# DataSource for Entity Framework

**GrapeCity US**

GrapeCity
201 South Highland Avenue, Suite 301
Pittsburgh, PA 15206
**Tel:** 1.800.858.2739 | 412.681.4343
**Fax:** 412.681.4384
**Website:** https://www.grapecity.com/en/
**E-mail:** us.sales@grapecity.com

## Trademarks



## Warranty

ComponentOne warrants that the media on which the software is delivered is free from defects in material and workmanship, assuming normal use, for a period of 90 days from the date of purchase. If a defect occurs during this time, you may return the defective media to ComponentOne, along with a dated proof of purchase, and ComponentOne will replace it at no charge. After 90 days, you can obtain a replacement for the defective media by sending it and a check for $2 5 (to cover postage and handling) to ComponentOne.

Except for the express warranty of the original media on which the software is delivered is set forth here, ComponentOne makes no other warranties, express or implied. Every attempt has been made to ensure that the information contained in this manual is correct as of the time it was written. ComponentOne is not responsible for any errors or omissions. ComponentOne's liability is limited to the amount you paid for the product. ComponentOne is not liable for any special, consequential, or other damages for any reason.

## Copying and Distribution

# Table of Contents

# DataSource for Entity Framework Overview

**DataSource for Entity Framework** adds ease-of-use and performance enhancements to the Entity Framework and RIA Services. It improves and simplifies data binding with these frameworks by solving common problems related to loading, paging, filtering and saving data. It also provides performance enhancements such as fast loading and transparent scrolling over large datasets with Virtual Mode.

# Help with WinForms Edition

**Getting Started**

For information on installing any of the **ComponentOne Studio WinForms Edition**, licensing, technical support, namespaces and creating a project with the control, please visit Getting Started with WinForms Edition.

## Key Features

The following are some of the main features of **DataSource for Entity Framework**, that you may find useful:

> **Note**: This version of **DataSource for Entity Framework** requires Entity Framework 6 or above, .NET Framework 4.5 or above, and Visual Studio 2012 or above.

- **Entity Framework Made Easy with Design-time Components**
  C1DataSource allows you to set up your data sources directly on the designer surface, with easy-to-use property dialogs and very little code to write. Configure the C1DataSource control and apply server-side filter, sort and group descriptors quickly at design-time. Of course, if you prefer to do everything in code you have the freedom of doing so using the rich data class libraries.
- **Live Views with LiveLinq**
  LINQ is the perfect tool for transforming raw data into custom views. C1DataSource makes LINQ even more powerful by making LINQ query statements live. C1DataSource includes ComponentOne LiveLinq, an extension library which augments the functionality of LINQ to speed up queries and provide live views. With LiveLinq you can shape your view however you want using LINQ operators without losing full updatability and bindability. "Bindability" means that your views are not just static snapshots of their source data. They are "live" and automatically reflect changes in the data. With LiveLinq your query results are kept up-to-date without re-populating every time changes in your data occur.
- **Simplifies MVVM**
  C1DataSource can simplify programming with the widely-adopted Model-View-ViewModel pattern known as MVVM. You have to write a lot more code to develop MVVM applications because of the additional code layer, the ViewModel, and the synchronization between the Model and ViewModel data members. With DataSource you can use live views as your ViewModel and not have to worry about writing any synchronization code. Live views are synchronized automatically with their sources and are much easier to create. You can use C1DataSource with any MVVM framework.
- **Virtual Mode for Large DataSets**
  Handle infinitely large datasets with C1DataSource's Virtual Mode. Virtual Mode allows you to navigate through large datasets asynchronously. It works like paging on the data layer but the user can scroll through the data as if all rows were on the client. As the user scrolls, chunks of data are retrieved from the source page by page and disposed of as necessary. You can use Virtual Mode with any GUI control, such as a **DataGrid**, **C1FlexGrid**, or any control you want. Data can also be modified. This feature is transparent to the developer; you can turn on Virtual Mode with one simple property setting.
- **Paging with Data Modification**
  For applications that prefer a paging interface, C1DataSource also supports paging without any limitations on data modification. That means users can make changes on multiple pages in one session before having to push the changes back to the database. This is a substantial improvement over other paging implementations such as with the Microsoft RIA Services DomainDataSource. Paging with C1DataSource is also supported in WinForms where paging is not provided.
- **Smart Client-side Caching**
  The key to most of **DataSource for Entity Framework**' features is its built-in client-side data cache. C1DataSource maintains a cache of entities on the client. When new queries are executed, it does not necessarily go to the server. It checks the client-side cache first and will not go to the server if it can find the result in the cache. This significantly improves performance and speed because it minimizes the number of trips to and from the server.
- **Context Management**
  With C1DataSource you can use a single data context for your entire application. This allows you to forget the troubles of programming against multiple contexts across multiple views. Without C1DataSource you might have multiple contexts which can be very difficult to write code when you need to use entities from different contexts together. This feature is made possible by the smart client-side cache.
- **Memory Management**
  C1DataSource is optimized for both performance and memory consumption. It manages memory resources for

you releasing old entity objects in the cache when necessary to prevent memory leaks. In doing so, it fully preserves consistency by maintaining required entities and entities modified by the user.

- **Server-side Filtering**
  Data brought from the server to the client usually needs to be filtered, or restricted in some way, to avoid moving large amounts of data over the wire and heaping it on the client. C1DataSource makes this common task very simple. Instead of doing it manually in code, you can specify server-side filters as simple property settings on C1DataSource.
- **Client-Side Transactions**
  **DataSource for Entity Framework** gives developers a simple and powerful mechanism for rolling back (cancelling) and accepting changes on the client without involving the server. C1DataSource makes it easy to implement **Cancel**/**Undo** and **OK**/**Accept** buttons anywhere, even in nested (child) dialog boxes and forms, modal and modeless.
- **Saving Modified Data**
  C1DataSource's smart client caching makes it easy to write code that saves modified data back to the server. With just one line of code and one trip to the server, you can save multiple entities. DataSource does all of the heavy lifting; it maintains changed entities in the cache, and ensures the cache is always consistent, while also optimizing memory usage and performance.
- **Code-First Support at Design Time**
  C1DataSource can also be used in code-first scenarios, without generating code from a model. If you need the C1DataSource "live" features, ensure that your entity classes implement the **INotifyPropertyChanged** interface and if their collection navigation properties exist, they use **ObservableCollection** interface.
- **Cross-Platform Support**
  **DataSource for Entity Framework** includes a C1DataSource component which allows you to combine multiple client view sources using Entity Framework or RIA Services. C1DataSource is supported in WinForms (.NET 4.5 or above), WPF and Silverlight 4.

# DataSource for Entity Framework Introduction

In designing the Entity Framework, Microsoft set out to create a platform agnostic means to allow developers to communicate easily with the database that underpins their desktop or server applications, and with WCF RIA Services, this principle was further extended to the Silverlight platform. Each of these frameworks provides developers with an almost ideal solution to promote data persistence (the controlled retrieval and return of data to an underlying database) but falls short of providing them with an easy way to create the application logic and interaction with bound GUI controls that forms such an intrinsic part of most applications that are being built nowadays. **DataSource for Entity Framework** has been specifically designed to meet this shortfall. It enhances both frameworks by providing additional functionality and improving overall application performance. In short, it provides developers with the means to speed up the development of typical line of business applications and write less code to achieve their goal.

The following topics will examine these performance-enhancing features of the C1DataSource in more detail, but we'll begin by examining the critical improvements it makes to the two core frameworks.

Following is the list of assemblies that are used in a WinForms application using a C1DataSource control.

| Assembly | Description |
|---|---|
| C1.Data.Entity.4.dll | Contains classes and objects that support Entity Framework 6. |
| C1.LiveLinq.4.dll | Contains assemblies that can be used while implementing Linq features. |
| C1.Win.Data.Entity.4.dll | Contains classes and objects that support Entity Framework 6. |

# Unified Data Context

In both the Entity Framework and WCF RIA Services, the developer is solely responsible for managing the contexts (sessions), creating them explicitly and often creating individual ones for each form in an application. Objects retrieved by one context bear no relation to those retrieved by another, even if they are essentially similar. This can create severe problems when different forms (or even different tabs on a tab control in a single form) need to interact. Up until now, the traditional way to solve these problems has been to repeatedly save data back to the underlying database and continually refresh the GUI to reflect any changes that have occurred. The result - a lot of repeated code and degradation in overall application performance brought about through repeated trips to the database server.

**DataSource for Entity Framework** provides a better solution by combining a unified data context and intelligent client cache. This means that an application needs only one data context which is available to all forms and controls therein. The intelligent client cache keeps track of all changes made to the context's objects, in essence enabling client-side transactional functionality, removing the need to continually save and refresh views, so application performance is improved and code simplified. Local data cache also enables client-side queries with full LINQ capabilities (filtering, sorting, grouping, and more) which would otherwise be impossible.

> **Note: DataSource for Entity Framework** supports both kinds of Entity Framework contexts, the newer **DbContext** (the default context in Visual Studio) and the legacy **ObjectContext**.

# Virtual Data Access

With its unique virtual mode feature, **DataSource for Entity Framework** supports access and data binding to large data sets, ranging from thousand to millions of rows, without awkward paging. In this mode, C1DataSource automatically retrieves rows from the database when they are needed for display in bound controls and then disposes of them when they are no longer required. That way the large data set appears to be in memory and ready for data binding without additional code and without consuming more memory resources than is necessary for displaying the rows currently shown by the bound controls.

The most common way of dealing with large data sets is to use paging. Paging usually provides for a poor user experience compared with data grids and other GUI controls directly bound to the data source. Virtual mode makes it possible to bind rich GUI controls directly to large data sets without paging.

## Enhanced Data Binding

Out-of-the-box data binding support, both in Entity Framework and in RIA Services, is limited to binding to the same query result that was originally retrieved from the server. In other words, you can neither bind to a subset of, nor reshape in any way, the original query. Also, an out-of-the-box data source control (**DomainDataSource**) is available only for RIA Services in Silverlight, and it has serious limitations such as disallowing paging and filtering unless all data changes are committed to the database.

**DataSource for Entity Framework** provides data source controls for direct access to Entity Framework, for both WPF and for WinForms, as well as for RIA Services (free from the standard DomainDataSource limitations).

Apart from the data source controls, C1DataSource supports creating live views that allow easy declarative reshaping of collections for two-way live data binding, which includes not just sorting and filtering but also calculated fields and any LINQ operations. Using live views, developers can express a significant part, if not all, of application logic with declarative data binding producing shorter and more reliable code. Also, by defining live views over entity collections, a developer can create a view model layer in the application making it follow the popular MVVM (model-view-view model) pattern with very little code for creating a view model and no code at all for synchronizing it with the model, a task that without C1DataSource would require extensive manual coding.

# Quick Start: C1DataSource for WinForms

The following quick start guides will show you how to get started with C1DataSource for WinForms. This guide walks you through the steps to add a data source, adding a grid control, and connecting it with the data source to display the data.

You start by adding a data source to your project, connect it to a C1DataSource component, and adding a grid to display data. This quick start also illustrates the essential steps for data binding. For a more detailed explanation of the process, see the Simple Binding topic in this documentation.

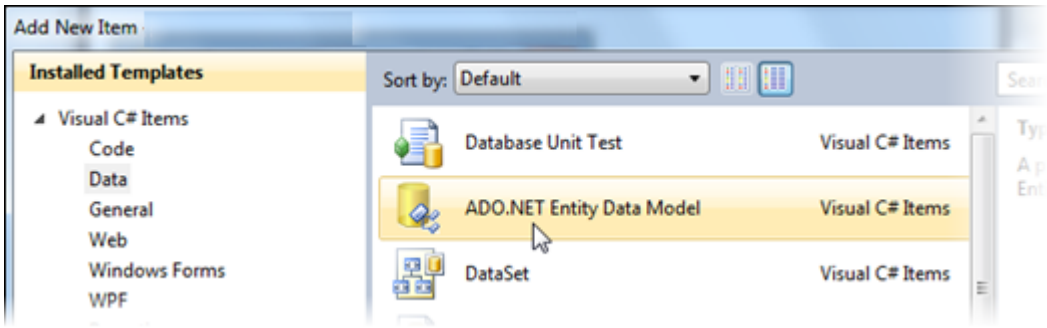
This quick start tutorial uses the c1nwind database (NORTHWIND) that is installed with the product in the **Common** folder at the following location:

**Documents\ComponentOne Samples\Common**

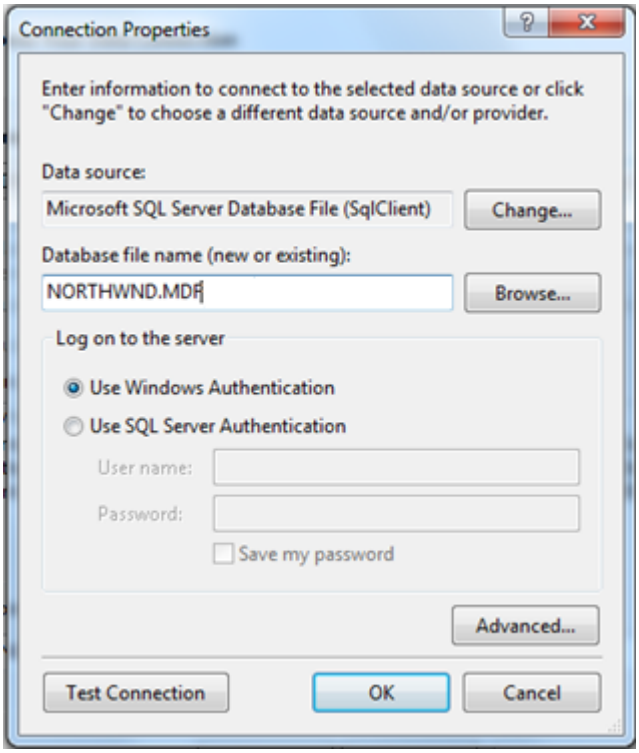
# Step 1: Adding a Data Source

Begin by creating a new Windows Forms project in Visual Studio. Then you will add a connection to the Northwind database.

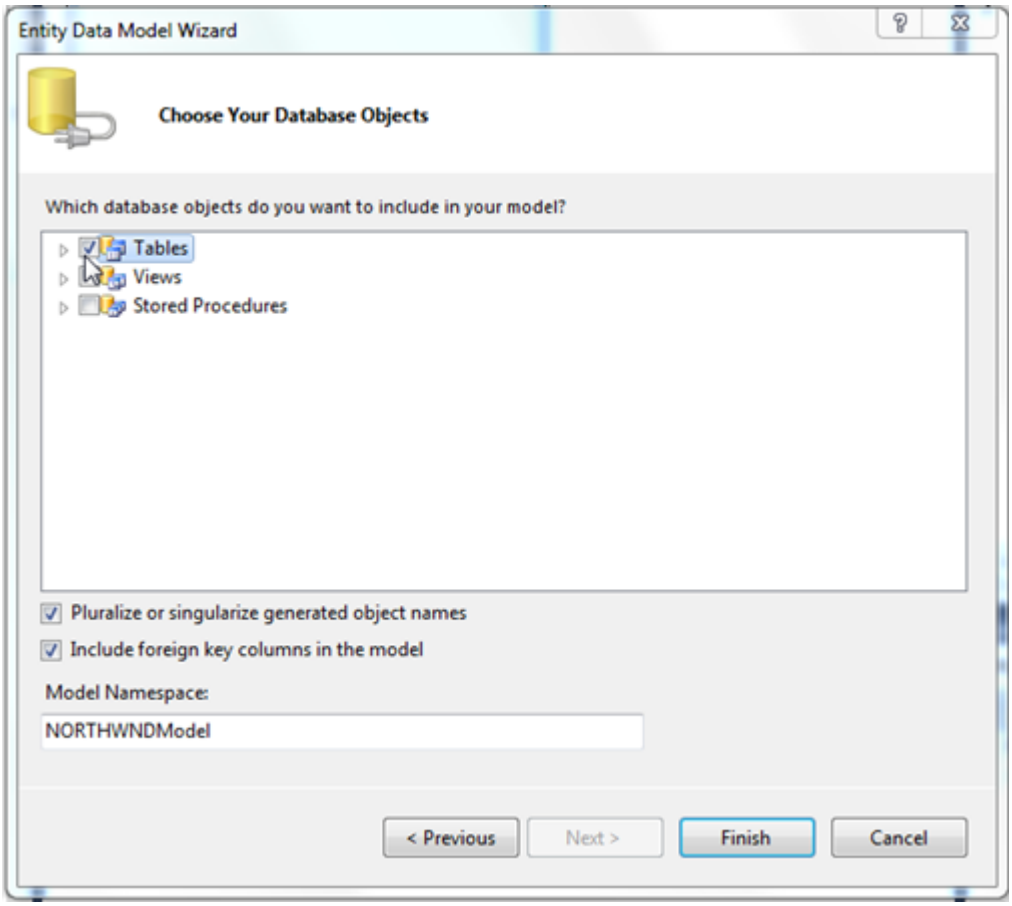
1. In Visual Studio, choose **File | New | Project.**
2. Select the **Windows Forms Application** template and click **OK**.
   Next, add an Entity Data Model based on the Northwind database. This model will provide the data for the entire application.
3. In the **Solution Explorer**, right-click the project name and select **Add | New Item**.
4. Choose the **ADO.NET Entity Data Model** item and then click **Add**.




5. In the **Entity Data Model Wizard**, select **Generate from database** to generate the model from the Northwind database, and then click **Next**.
6. Click the **New Connection** button.
7. In the **Choose DataSource** dialog box, select **Microsoft SQL Server Database File** and click **Continue**.
8. Browse to find the NORTHWND.MDF, select it, and click **Open**. The NORTHWND.MDF is installed with the product in the **Common** folder within **Documents\ComponentOne Samples\Common** .




9. Click **OK** and then click **Next**.
10. In the **Choose Your Database Objects** window, select **Tables** and click **Finish**.




11. In the Solution Explorer, expand the *model1.edmx* node.
12. Delete the *Model1.Context.tt* file.

13. Delete the *Model1.tt* file.
14. Right click the model diagram and select **Add Code Generation Item** from the context menu.
15. In the Add Code Generation Item Dialog box, select '**ComponentOne EF 6.x DbContext Generator'**.

> **Note:** When you install **DataSource for Entity Framework**, **ComponentOne EF6.x DBContext Code Generation Templates** will be added to each version of Visual Studio that C1DataSource supports. These templates ensure that the DBContext models that you create, provide entities that support **INotifyPropertyChanged**.

Now build the project so the new Entity Data Model classes are generated and become available throughout the project. Once they are available, you will be able to connect the data to the C1DataSource component.

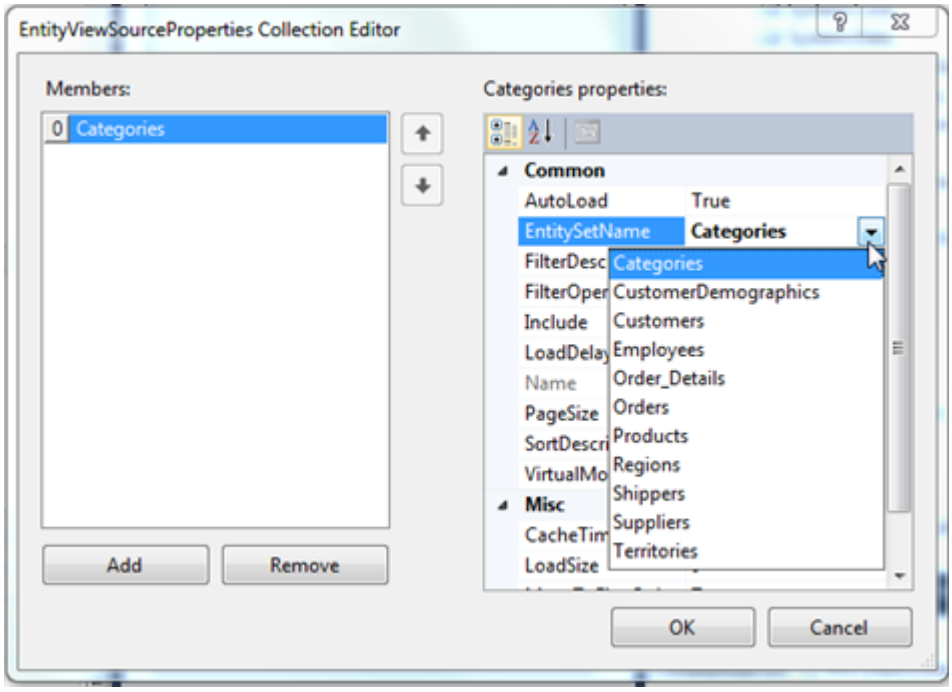
## Step 2: Connecting the Data to C1DataSource

In this step, you'll add a C1DataSource component to the form and connect it to the *Categories* table of the data source.

1. Drag a C1DataSource component from the Toolbox onto the form. This is a non-visual component, so it will appear on the tray below the form area rather than on the form itself.
2. In the **Properties** window, set the C1DataSource's ContextType property to the available item in the drop-down list. It should be something similar to *AppName.NORTHWNDEntities*.

> **Note:** The C1DataSource.ContextType Property accepts both **DbContext** and **ObjectContext** derived types, as its values.

3. Click the **ellipsis** button next to the **ViewSourceCollection** property to open the **EntityViewSourceProperties Collection Editor**.
4. Click **Add** and set the **EntitySetName** property to *Categories*.



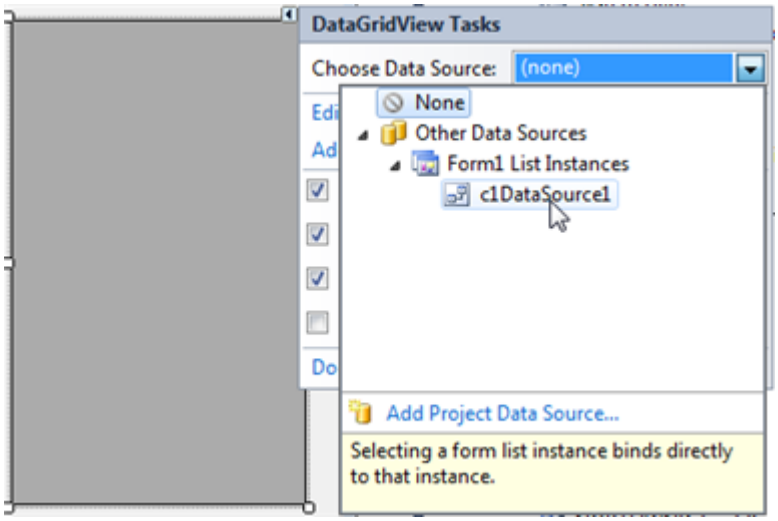

5. Click **OK** to close the editor.

With the database connected to C1DataSource, all you need to do is add a grid to display the data.

## Step 3: Adding a Grid

In this step you will add a grid that will be used to display the data in the *Categories* table of the Northwind database. You can use **C1FlexGrid** or any grid you are familiar with, but in this example, we'll use a **DataGridView** control.

1. Drag a **DataGridView** control from the **Toolbox** onto your form.
2. Click the **DataGridView**'s smart tag and set the **DataSource** to the c1DataSource1 control we added to the form.

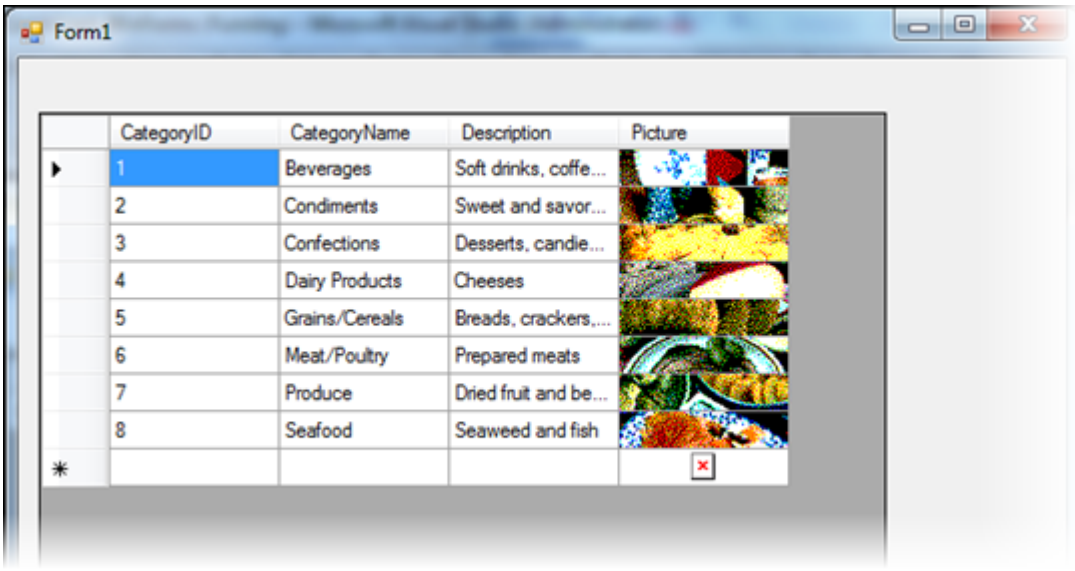



3. In the Visual Studio Properties window, set the **DataMember** property to *Categories*. The grid will automatically generate columns for all the fields in the *Categories* type.

   Now simply run the project to view the grid!

## Step 4: Running the Project

Press **F5** to run the project. The data from the *Categories* table of the Northwind database is displayed.



## Design-Time Features

**DataSource for Entity Framework** includes a C1DataSource component which facilitates Rapid Application Development by allowing developers to specify data sources on a rich designer surface that requires little or no additional coding. In addition, the C1DataSource includes classes that provide support for the same features that can be controlled through the designer surface, plus many extra features that provide greater control over it through code.

We'll begin our exploration of the C1DataSource component by looking at how we can control it via the designer surface. From there, we explore how it can be controlled dynamically at runtime. Not all features available at runtime will be represented here. You can consult the Programming Guide section of this documentation and the API Reference for more runtime features, including client-side transaction support and more.

## Data Binding

We'll begin by creating a new Windows Forms project in Visual Studio.

1. In Visual Studio, choose **File | New Project.**
2. Select the **Windows Forms Application** template and click **OK**.

Next, add an Entity Data Model based on the Northwind database. This model will provide the data for the entire application:

1. In the **Solution Explorer**, right-click the project name and select **Add | New Item**.
2. In the **Data** category, select the "ADO.NET Data Model" item and then click **Add**.
3. Use the **Entity Data Model Wizard** to select the database you want to use. In this example, you will use "NORTHWND.MDF", which should be installed in the **ComponentOne Samples\Common** folder.
4. In the Solution Explorer, expand the mode next to *model1.edmx*.
5. Delete the *Model1.Context.tt* file and the *Model1.tt* file.
6. Right click the model diagram and select **Add Code Generation Item** from the context menu. Select '**ComponentOne EF 6.x DbContext Generator' from the Add Code Generation Item Dialog box**.

   > **Note:** When you install **DataSource for Entity Framework**, **ComponentOne EF6.x DBContext Code Generation Templates** will be added to each version of Visual Studio that C1DataSource supports. These templates ensure that the DBContext models that you create, provide entities that support **INotifyPropertyChanged**.

Now build the project so the new Entity Data Model classes are generated and become available throughout the project.

Next, add a C1DataSource component to the application and connect it to the Entity Data Model:

1. Drag a C1DataSource component from the Toolbox onto the form. This is a non-visual component, so it will appear on the tray below the form area rather than on the form itself.
2. Select the new component and choose **View | Properties Window**.
3. In the Properties window, set the **ContextType** property to the type of object context you want to use. In this case, there should be only one option in the drop-down list, something similar to "AppName.NORTHWINDEntities".

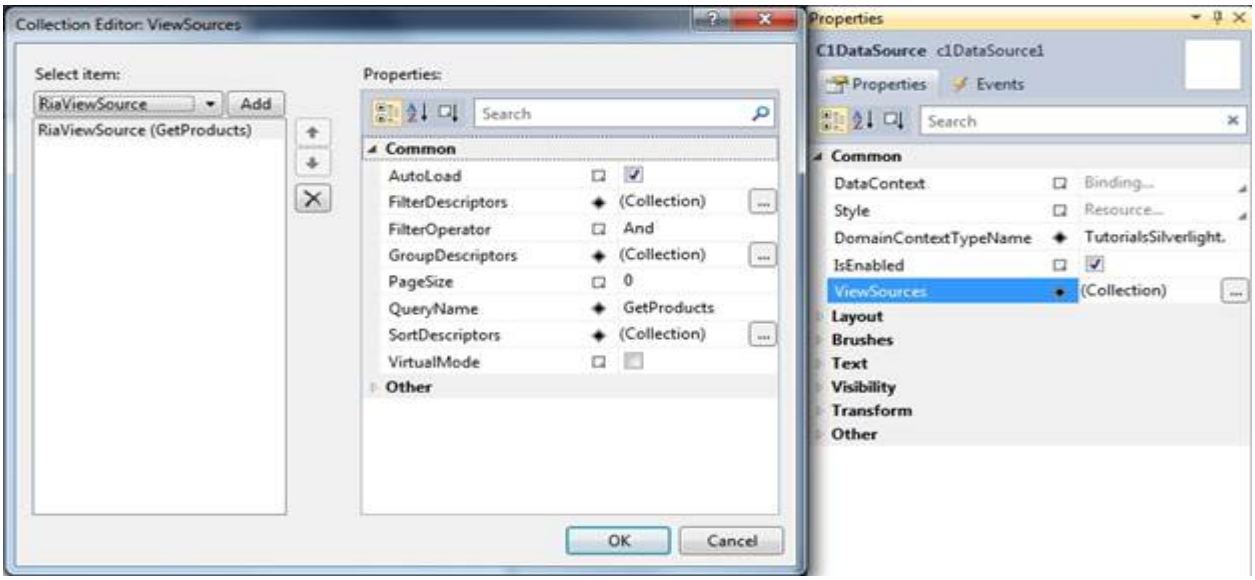
At this point, the C1DataSource has created an application-wide object (an **EntityDataCache**) that represents the Northwind database and has application scope. Additional C1DataSource objects on other forms will share that same object. As long as they are part of the same application, all C1DataSource objects share the same **ObjectContext**.

This unified object context is one of the main advantages C1DataSource provides. Without it, you would have to create multiple object contexts throughout the application, and each would have to be synchronized with the others

and with the underlying database separately. This would be a non-trivial task, and any errors could compromise the integrity of the data. The unified object context handles that for you transparently. It efficiently caches data and makes it available to all views in a safe, consistent way.

Now that our C1DataSource has an **ObjectContext** to work with, we will go on to specify the entity sets it will expose to the application through its **ViewSources** collection. Note that if you are familiar with ADO.NET, you can think of the C1DataSource as a **DataSet** and the **ViewSources** collection as **DataView** objects.

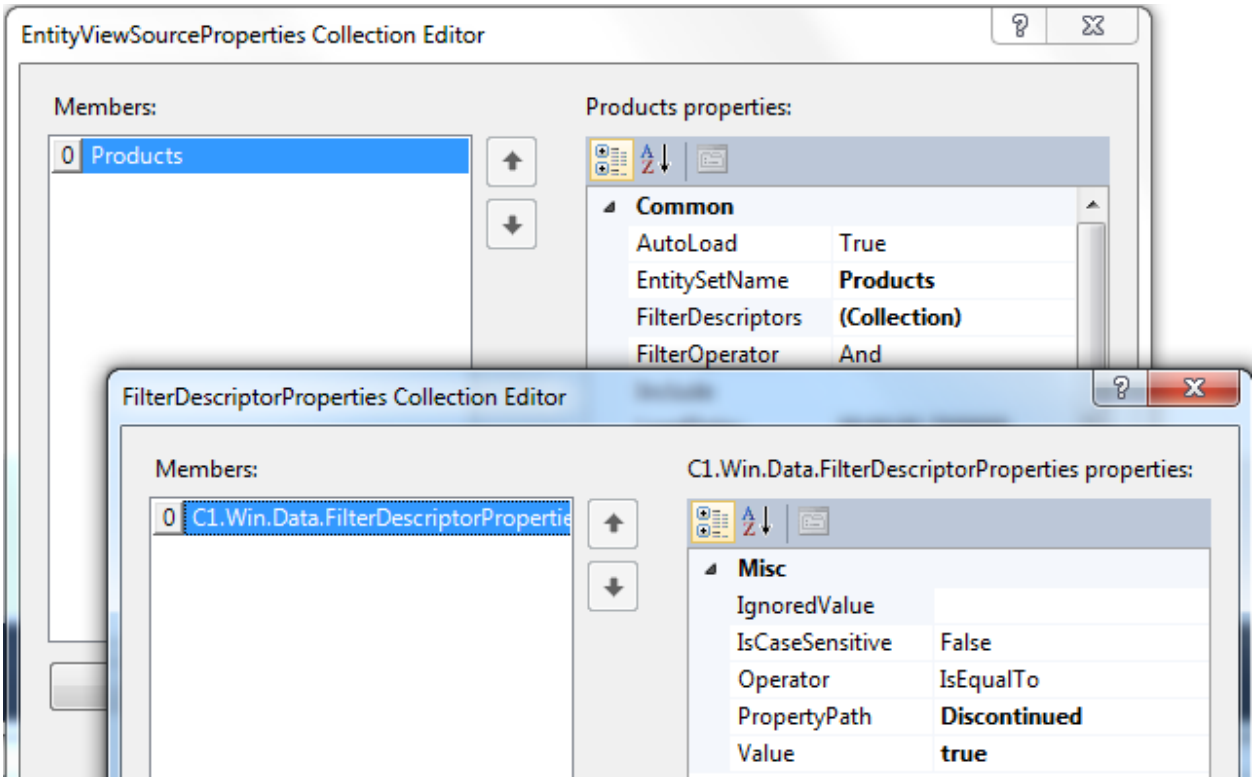
In the properties of the C1DataSource, locate the **ViewSourcesCollection** property and open its editor dialog. Click **Add** and then select *Products* from the **EntitySetName** drop-down list.




For this simple example, *Products* is all that is really necessary, but we could continue to create **ViewSources** within this C1DataSource in exactly the same way. We might, for example, create a **ViewSource** based on the *Categories* entity set allowing us to have a form that would be used to show the master-detail relationship between *Categories* and their *Products*. Conversely, there is no need to define all of the **ViewSources** that you might need in one single C1DataSource. You could have a separate C1DataSource for each **ViewSource** that you need. All you need to do is ensure that the C1DataSource components that you use utilize the same **ContextType**.

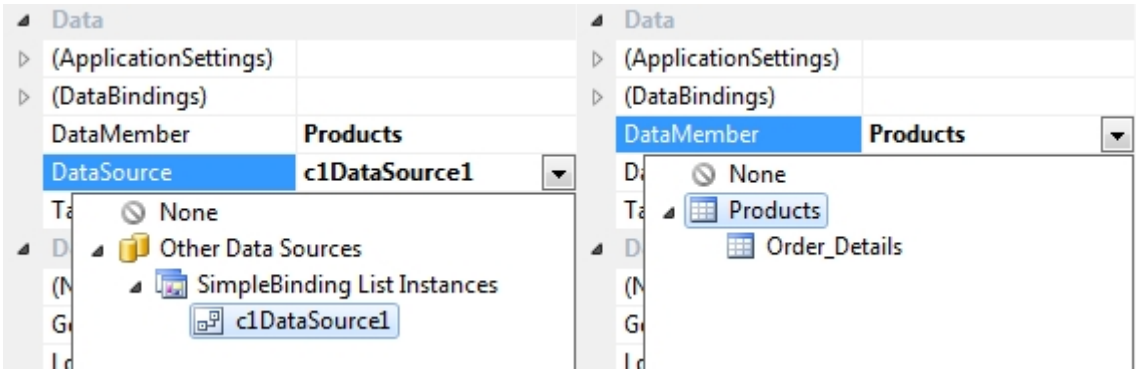
In reality we'll only want to bring a small subsection of the data contained within our database back to the client at any one time. This avoids overloading the client and network and also ensures that we are only presenting data that is relevant to the task our end user is engaged in. The traditional approach to this has been to create code (typically SQL queries) that we would run against the database to achieve our desired results. With C1DataSource, we can make use of the designer surface to achieve our goal without the need to write code by specifying server-side filters as property settings.

From the **ViewSourceCollection** editor, open the **FilterDescriptor** collection editor, add a filter descriptor and type the property name and a value for it for server-side filtering. If you need to filter more than one property, you can add additional filter descriptors.



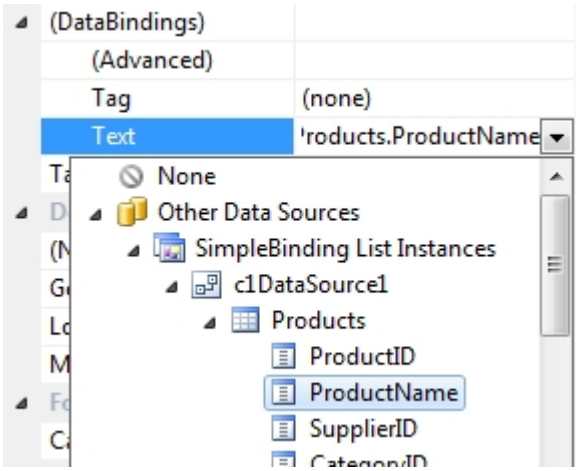

Using exactly the same methodology, you can add **SortDescriptors** provide sorting on the data you retrieve.

With our C1DataSource configured, add a grid control to the form. This can be **DataGridView**, a **C1FlexGrid** or indeed any grid that you are familiar with. Set the grid's DataSource property to the name of the C1DataSource (if you haven't specifically named the **C1DataSource,** then this will be *c1DataSource1*) and its **DataMember** property to *Products*. The **DataMember** property will, in fact, display a drop-down list of all the **ViewSources** (or Entity Sets) that we defined for the C1DataSource.




At this point, the grid will automatically generate columns for all the fields in the *Product* type, and most grids will allow you to further customize these columns and their layout via their built-in designers. Once you are happy with the grid's layout, save, build, and run the application. Notice that the data loads automatically and that you can sort, add and remove items as you'd expect to be able to do. All this has been achieved by adding just two items to your form (a **C1DatSource** and a data grid) and setting a few properties. You did not have to write a single line of code!

You could continue to add more controls to this form and bind them to specific items in the *Products* collection. To illustrate this point, add a **TextBox** control to the form. From the Properties window, expand the **DataBindings** section and bind its **Text** property to the *ProductName*, as illustrated.




Save, build and run the application again, and this time notice how the name of the product currently selected in the grid appears in the **TextBox** that you have just added to the form. If you then edit the product name in either control, the change will be immediately reflected in the other.

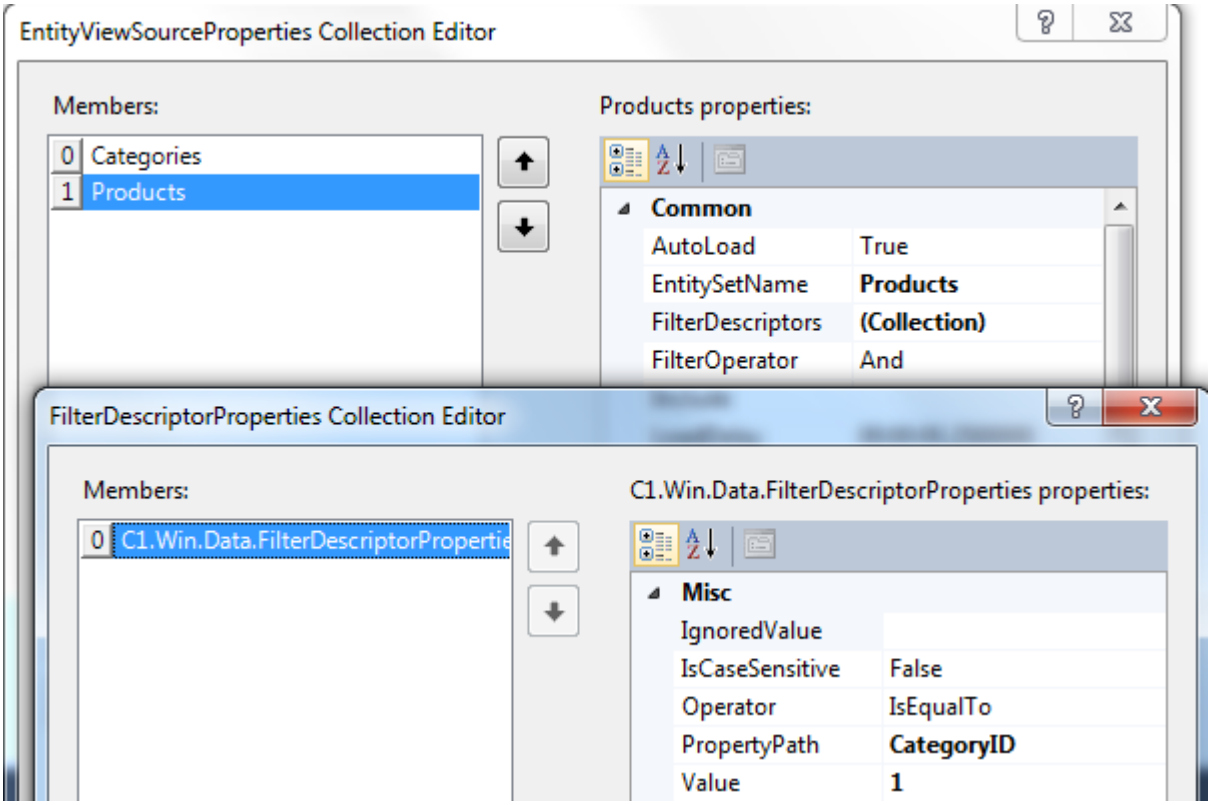
## Server-Side Filtering

We already mentioned that it is generally desirable to restrict the data returned from the server to the client, and we demonstrated how C1DataSource facilitates this with the use of the **FilterDescriptor Collection Editor**. Now we'll demonstrate how to provide the end user with the means to achieve server-side filtering.

The user will select a *Product Category* from a combo box, for example, although other GUI controls can be used, and that will load a **DataGrid** with new data from the server.

To implement server-side filtering, follow these steps:

1. Add a new form with a C1DataSource component to the project used in Simple Binding. You can make this form the project's start up form to save time when you run the project.
2. Establish the C1DataSource as before, but this time define two view sources: *Categories* and *Products*. Click the **Add** button twice in the **ViewSourceCollection**. Enter *Categories* next to the EntityViewSourceProperties.EntitySetName property for the first member (0) and enter *Products* for the second member (1). For *Products*, we'll define a filter as shown in the following picture:




- Add a **ComboBox** control to the form and set the following properties:
  - DataSource = C1DataSource  Name (typically C1DataSource1)
  - DisplayMember = *Categories.CategoryName*
  - ValueMember = *Categories.CategoryID*
- Add a **DataGrid** control to the form and set the following properties:
  - DataSource = C1 DataSource Name (typically C1DataSource1)
  - DataMember = *Products*
- Add the following code to the form to handle the combo box's *SelectedValueChanged* event:

**To write code in Visual Basic**

```
Visual Basic
Private Sub comboBox1_SelectedIndexChanged(sender As Object, e As EventArgs)
   c1DataSource1.ViewSources("Products").FilterDescriptors(0).Value = comboBox1.SelectedValue
End Sub
```

**To write code in C#**

```
C#
private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
{
   c1DataSource1.ViewSources["Products"].FilterDescriptors[0].Value = comboBox1.SelectedValue;
}
```

- Add the following code to the **Form_Load** event, to set the **AutoGenerateColumns** property, of the dataGridView control  to 'true'.

**To write code in Visual Basic**

| Visual Basic |
| --- |
| dataGridView1.AutoGenerateColumns = True |

**To write code in C#**

| C# |
| --- |
| dataGridView1.AutoGenerateColumns = true; |

- Save, build and run the application. Select a category in the combo box and notice how products related to that category are displayed in the grid. Switch between categories and notice how the data is returned. As before, all the items in the grid are editable.

## Client-Side Caching

The Server-Side Filtering example shows how the C1DataSource improves and simplifies working with the Entity Framework in applications, made possible by its client-side data cache, a key feature of C1DataSource. It enables several important enhancements and makes writing application code much easier.

Let's start with a performance enhancement that can be seen in the Server-Side Filtering example. When the user switches between categories, the grid loads products associated with them. The first time a category is chosen, there is a slight delay when the relevant data is retrieved. On subsequent occasions, when the same category is chosen, data retrieval is virtually instantaneous. This is because the data is being retrieved from the client memory data cache (EntityDataCache) rather than the server.

> **Note:** The EntityDataCache is actually smarter still, determining that a return trip to the server can be avoided, even in more complex cases where the queries may not be identical but can be fulfilled from the results of other queries already contained therein.

This performance enhancement may not be obvious if you are working on a single machine where no network interaction is required, but in real world applications it can make all the difference between a sluggish application that calls the server on every user action and a crisp interface with no delays.

The second important enhancement is in memory management. You might think based on what you've read and observed to date that the EntityDataCache continually accumulates data throughout its existence. If this were the case you would very quickly witness severe performance degradation. In reality the EntityDataCache is keeping track of what is stored within it and as data is no longer required is releasing it performing a self-cleansing operation at the same time. All of this is being done without the need for you to add extra code, and more importantly still it is ensuring that data integrity is being preserved at all times. Data won't be released if required by other data, nor will data that has been altered in some way without those alterations having being saved. This also relates to performance, because getting rid of unnecessary objects in memory improves performance; and vice versa, keeping large numbers of obsolete objects in memory leads to performance degradation.

We mentioned before how C1DataSource simplifies context management by eliminating the need to create multiple data contexts thanks to the client cache. Now we should explain what the EntityDataCache actually is and how we can further use this information to our advantage.

The EntityDataCache is essentially the context. In terms of C1DataSource namespaces, the cache is the **C1.Data.Entities.EntityClientCache** class, and it is in one-to-one correspondence with **ObjectContext** through its **ObjectContext** property. Both the cache and its underlying **ObjectContext** are created for you automatically if you use the C1DataSource component ; however, you can create them explicitly and set the **C1DataSource.ClientCache** property in code, if necessary.

To see how simple application code becomes thanks to the client-side cache, let's add the functionality of saving modified data to the Server-Side Filtering project.

1. Simply add a button, **btnSaveChanges**, to the form and add a handler for the code:

**To write code in Visual Basic**

```
Visual Basic
Private Sub btnSaveChanges_Click(sender As System.Object, e As System.EventArgs)
C1DataSource1.ClientCache.SaveChanges()
End Sub
```

**To write code in C#**

```
C#
private void btnSaveChanges_Click(object sender, EventArgs e)
 {
    c1DataSource1.ClientCache.SaveChanges();
 }
```

2. Save, build and run the application.
    - Select a category and then make some changes to the product information in the grid.
    - Select another category (and possibly a third) and again makes changes to the product details in the grid.
    - Click the button you added, close the application, reopen it and select the same categories as before.
    - Observe how the changes you made have been saved. C1DataSource has provided, via the EntityDataCache, a way to alter the product details of several categories' products without the need to save those changes each time different category is selected. To achieve this effect without C1DataSource, you would either need to write a lot of code, or you'd need to keep the entities (the product details) from different categories in the same context. This would waste memory without releasing it, creating a memory leak. C1DataSource simplifies all of this for you while optimizing memory usage and performance.

Client-side caching also makes possible other important features of C1DataSource, such as client-side queries and, especially, live views. Live Views is a feature that allows you to replace much of the complex application coding with simple data binding, which we'll learn more about later.

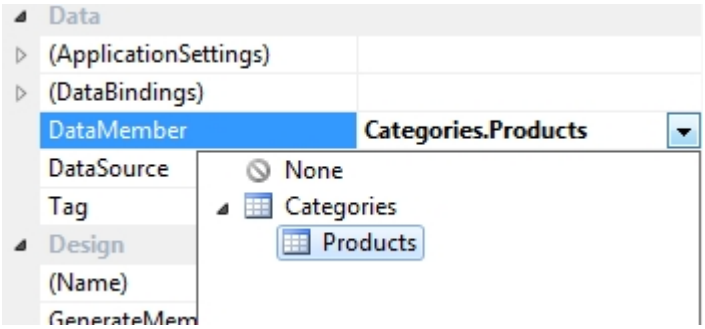
# Master-Detail Binding

As we have already seen in the Server-Side Filtering example, C1DataSource supports master-detail binding. With very large datasets, server-side filtering is by far the best solution, but with smaller datasets, client-side filtering can be just as efficient. This scenario demonstrates client-side master-detail binding using a grid, instead of a combo box used in the previous Server-Side Filtering to select categories.

To implement master-detail binding, follow these steps:

1. Using the project we created to demonstrate Server-Side Filtering, add a new form with a C1DataSource component using the same **ContextType** as before and create a ViewSource based on *Categories*. Note that you can make this the startup form to save time when you run the project.
2. Add what will become the 'master' grid to the form and set its DataSource property to the C1DataSource and its **DataMember** property to *Categories*.

   Now add a second grid to the form below the one you've just configured. Its C1DataSource will again be the C1DataSource, but set its **DataMember** property to the *Products* node which you'll find underneath *Categories* as shown in the following picture:




3. Save, build and run the application.

Selecting a category in the master grid causes the products linked to it to be displayed in the details grid (which as before is fully editable). Running this on a single machine, as you probably are, you won't notice any significant time lapse as you select new categories in the master grid and their respective products are displayed in the details grid. In the background, the C1DataSource is making use of an Entity Framework feature, implicit lazy loading, meaning that products are only being summoned for new categories as they are selected. For many scenarios, this is perfectly acceptable, but we began this section by specifically referring to master-detail relationships in small datasets. We might just as well fetch all of the products for all of the categories when the form loads and then display will be instantaneous whether on a single machine or across a network. To achieve this behavior, open the **ViewSourceCollection** editor and type *Products* in the **Include** property of the *Categories* view source.

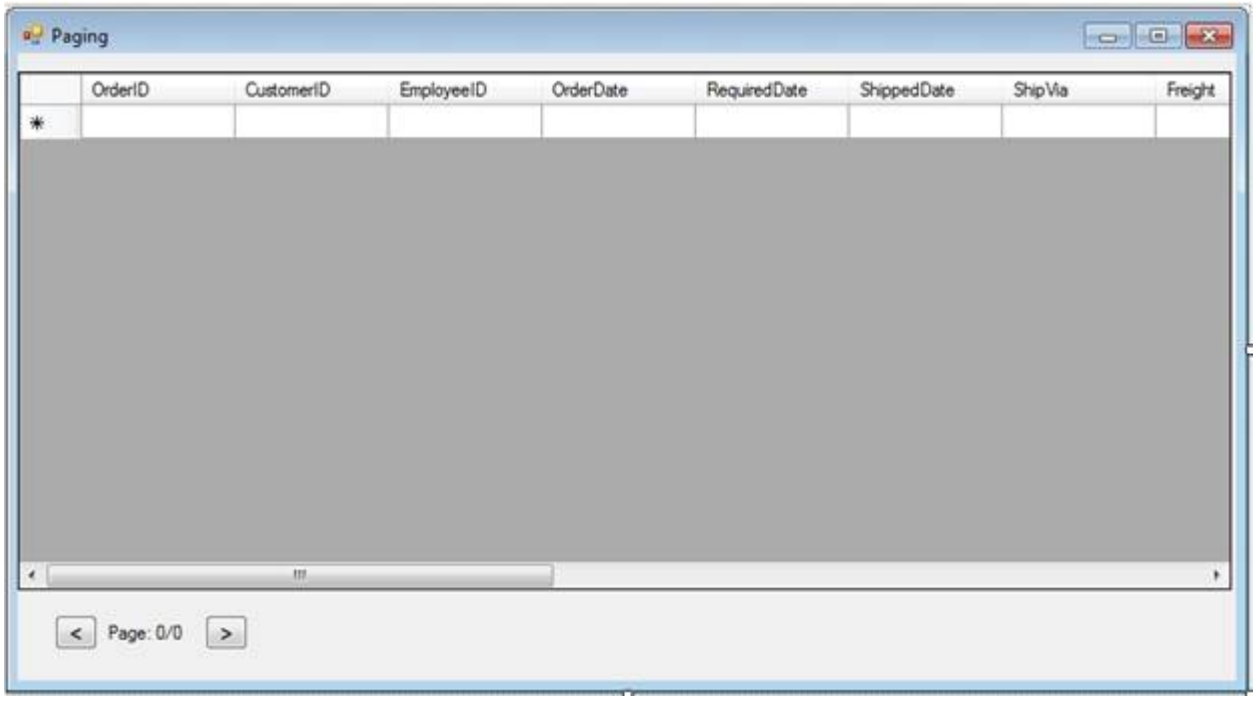
## Large Datasets: Paging

To show large amounts of data without bringing it all to the client at once, applications have traditionally used paging. Paging is not an ideal solution; it complicates the interface and makes it less convenient for the user, but it is preferred in some applications. For these cases, C1DataSource supports paging as expected, but it does so without the traditional limitations on data modification. The user can make changes in multiple pages in one session without being forced to send the changes from one page to the database before moving to the next page. That's a substantial enhancement compared to other paging implementations, such as the one in DomainDataSource in Microsoft RIA Services.

> **Note: DataSource for Entity Framework** does offer a solution to the drawbacks of paging; we'll cover Virtual Mode later in this documentation.

To implement paging, follow these steps:

1. Using the project we created to demonstrate Master-Detail Binding, add a new form with a C1DataSource component using the same **ContextType** as before. Note that you can make this the startup form to save time when you run the project.
2. Create a **ViewSource** in the **ViewSourceCollection** editor, entering *Orders* as the EntitySetName.
3. To enable paging, set the **PageSize** property to 10 for now, but you can choose any reasonable value for this property. It's simply determining the number of data rows that will be displayed on a page.
4. Add a **DataGrid** to the form and set its DataSource property to the C1DataSource and its **DataMember** property to *Orders*.
5. To facilitate easy movement between pages, add two buttons, **btnPrevPage** and **btnNextPage**, and a label, **labelPage**, as shown in the following image.




6. Add the following code containing a **RefreshPageInfo** handler for the **PropertyChanged** event used to show the current page number and page count, and handlers for button **Click** events used to move to the next and previous pages:

**To write code in Visual Basic**

Visual Basic

```
Imports C1.Data.DataSource

 Public Class Paging
    Private _view As ClientCollectionView

    Public Sub New()

        ' This call is required by the designer.
        InitializeComponent()

        ' Add any initialization after the InitializeComponent() call.
        _view = C1DataSource1("Orders")

        RefreshPageInfo()

        AddHandler _view.PropertyChanged, AddressOf RefreshPageInfo

    End Sub

    Private Sub RefreshPageInfo()
        labelPage.Text = String.Format("Page: {0} / {1}", _view.PageIndex + 1, _view.PageCount)
    End Sub

    Private Sub btnPrevPage_Click(sender As System.Object, e As System.EventArgs) Handles
btnPrevPage.Click
        _view.MoveToPreviousPage()
    End Sub

    Private Sub btnNextPage_Click(sender As System.Object, e As System.EventArgs) Handles
btnNextPage.Click
        _view.MoveToNextPage()
```

```
    End Sub
End Class
```

**To write code in C#**

C#

```
namespace TutorialsWinForms
{
    public partial class Paging : Form
    {
        ClientCollectionView _view;

        public Paging()
        {
            InitializeComponent();

            _view = c1DataSource1["Orders"];

            RefreshPageInfo();
            _view.PropertyChanged += delegate { RefreshPageInfo(); };
        }

        private void RefreshPageInfo()
        {
            labelPage.Text = string.Format("Page: {0} / {1}", _view.PageIndex + 1, _view.PageCount);
        }

        private void btnPrevPage_Click(object sender, EventArgs e)
        {
            _view.MoveToPreviousPage();
        }

        private void btnNextPage_Click(object sender, EventArgs e)
        {
            _view.MoveToNextPage();
        }
    }
}
```

7. Save, build and run the application. Page through the Orders. While you are moving between pages, try changing some data in the grid. Try changing data in one page, and then move to another page and try changing data there. Notice that C1DataSource allows you to do this without forcing you to save data to the database before leaving the page. This is an important enhancement compared with other paging implementations, including the one supported by DomainDataSource in Microsoft RIA Services (which is for Silverlight only, whereas C1DataSource supports this, as other features, for all three platforms: WinForms, WPF, Silverlight).

Try also to delete some orders. This is also allowed without restrictions, and moreover, the current page will automatically complete itself to keep the number of rows in the page unchanged. You can also add new rows. They are added to the end of the dataset.

And if you add a **Save Changes** button, using the following code as we did previously, then you will be able to save changes made in multiple pages by pressing that button when you are done.

**To write code in Visual Basic**

Visual Basic

```
Private Sub btnSaveChanges_Click(sender As System.Object, e As System.EventArgs)
C1DataSource1.ClientCache.SaveChanges()
End Sub
```

**To write code in C#**

C#

```
private voidbtnSaveChanges_Click(object sender, EventArgs e)
 {
   c1DataSource1.ClientCache.SaveChanges();
 }
```

All this functionality is what you would expect from a paging implementation that supports unrestricted data modification. Unfortunately, it is not easy to implement. For example, think of all possible cases where changing data on one page interferes with what should be shown on other pages, and so on. That is why paging usually imposes severe restrictions on data modifications, if they are at all allowed. For example, MS DomainDataSource requires you to save all changes before changing pages. Fortunately, C1DataSource supports paging without restricting data modifications in any way.

As with many other features, unlimited modifiable paging is made possible by the client-side cache, see The Power of Client Data Cache. That also means that paging implementation in C1DataSource is optimized both for performance and for memory consumption. The cache provides that optimization. It keeps recently visited pages in memory, so re-visiting them is usually instantaneous. And it manages memory resources, releasing old pages when necessary, to prevent memory leaks.

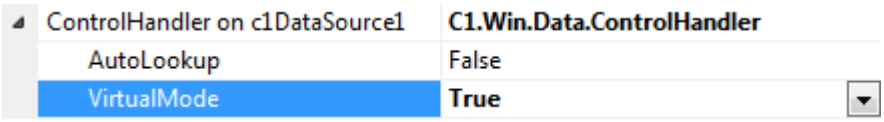
## Large Datasets: Virtual Mode

As mentioned in the Large Datasets: Paging topic, C1DataSource has an even better solution than paging for dealing with large data and large numbers of rows.

What if using large datasets with thousands, or even millions, of rows was no longer a problem? What if you could use the same controls to display massive datasets as you would for smaller datasets, with no paging, no code changes, and all by setting one Boolean property? With C1DataSource you can, thanks to the magical **VirtualMode** property.

To implement virtual mode, follow these steps:

1. Using the project we created to demonstrate Large Datasets: Paging, add a new form with a C1DataSource component using the same **ContextType** as before. Note that you can make this the startup form to save time when you run the project.
2. Create a **ViewSource** in the **ViewSourceCollection** editor. Use the largest table in our sample database: *Order_Details*.
3. Set the **VirtualMode** property to *Managed*. Another possible value is *Unmanaged*, but that is an advanced option that should be used with caution and only when necessary. The *Managed* option means that getting data from the server is managed by a grid control. With the *Managed* option, C1DataSource supports C1FlexGrid and Microsoft DataGridView. It is optimized for performance with those specific grid controls. The *Unmanaged* option means that virtual mode is not driven by a specific control, will work with any bound controls but is subject to some limitations described here.
4. Add a **DataGrid** to the form and set its **DataSource** property to the C1DataSource and its **DataMember** property to *Order_Details*.
5. Now, since we selected the *Managed* option, we need to specify which grid control is driving the data. Select the grid in the designer and find the property called "**ControlHandler on c1DataSource1**" in the Properties window, as shown in the following picture:




This is an extender property and will be available only if there is a C1DataSource component on the form. Set the **VirtualMode** property there to *True*.

6. Save, build and run the application. You'll see nothing earth-shattering, simply a grid you can navigate, scroll and modify as you see fit. It looks and behaves like any conventional data grid, which is exactly the point. You can use large datasets without the drawbacks of paging and without code. And as an added benefit, you can use any GUI control you like so long as it has a DataSource property. This example uses a relatively modest-sized dataset, but C1DataSource running in virtual mode would be just as responsive with a much larger dataset; it does not depend on the number of rows. To further prove the point, look at the **OrdersDemo** sample installed with this product. The sample uses a larger database, with roughly 65,000 additional rows, but the responsiveness is the same as our example here. Again, it does not depend on the number of rows in the dataset.

How does this magic work? It works much like paging, only hiding its inner workings from the GUI controls, with paging occurring under the hood. The GUI controls see the data as if it is fetched to the client and ready for them. When GUI controls request data, C1DataSource, or **ClientViewSource** if it is used in code without a C1DataSource control, first checks whether it can serve the data from memory, from the same client-side cache it uses for all features. If it can't find it in memory, it transparently fetches the required data from the server. As with other features using the client-side cache, C1DataSource does not store the fetched data indefinitely, which would be a memory leak. It knows which parts of the data are needed for serving the GUI controls and which parts should be kept because they are modified or related to modified parts, and so on. It releases old, unnecessary parts of data as needed. No code is required and any GUI controls can be used.

## Automatic Lookup Columns in Grids

A common scenario in data binding is for data classes to contain references to other data classes. For example, a **Product** object may contain references to **Category** and **Supplier** objects.

In ADO.NET, the references usually appear as foreign keys that map to other tables (e.g., **Product.CategoryID** and **Product.SupplierID**).

In the Entity Framework, you still get the key columns, but you also get the actual objects. So you have **Product.CategoryID** (usually an integer) and **Product.Category** (an actual Category object).

Displaying foreign keys in a grid is not very useful, because it is unlikely that users will remember that category 12 is "Dairy Products" or that supplier 15 is "ACME Imports". Allowing users to edit these keys would be even worse. A common way to work around this problem is to remove the related entity columns from any bound grids and, optionally, to replace them with custom columns that use combo boxes for editing the values, so called lookups. The combo boxes have to be bound to the related tables and have their properties set so they display relevant values (e.g., **Category.Name** or **Supplier.CompanyName**) and so they are synchronized with the values being displayed on the grid. This is not terribly hard to do, but it is a tedious and error-prone task that makes projects harder to create and maintain.

C1DataSource can do this tedious work for the developer; it can automatically change related entity columns so that they show combo box lookups. It can do this for several types of data grids, those that it supports. Currently supported WinForms grids are: C1FlexGrid and Microsoft DataGridView. Here we will show how to do this for **C1FlexGrid**.

C1DataSource provides an *extender* property called **ControlHandler**. If you place a **C1FlexGrid** control on a form that contains a C1DataSource, the grid will get an additional **ControlHandler** property. A ControlHandler is an object containing (at present) a single boolean property **AutoLookup**. This property set to *True* causes the C1DataSource to configure grid columns that contain references to other entities so that they show lookup combos.

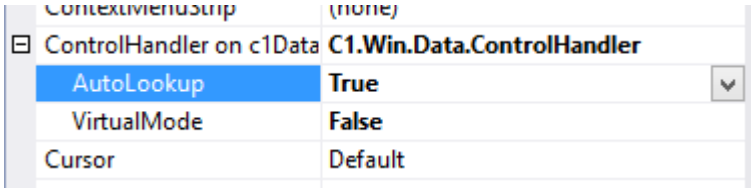
To see how it works, follow these steps:

1. Using the project used in Simple Binding, add a new form with a C1DataSource component using the same **ContextType** as before.
2. Create a **ViewSource** in the **ViewSourceCollection** editor, entering *Products* as the **EntitySetName**.
3. Add a **C1FlexGrid** to the form and set its DataSource property to the C1DataSource and its **DataMember** property to *Products*.
4. Save, build and run the application. It will look like this:

| Product ID ▲ | Product Name | Category |
|---|---|---|
| 5 | Chef Anton's Gumbo Mix | Category : 2 |
| 9 | Mishi Kobe Niku | Category : 6 |
| 17 | Alice Mutton | Category : 6 |
| 24 | Guaraná Fantástica | Category : 1 |
| 28 | Rössle Sauerkraut | Category : 7 |
| 29 | Thüringer Rostbratwurst | Category : 6 |
| 42 | Singaporean Hokkien Fried Me | Category : 5 |
| 53 | Perth Pasties | Category : 6 |

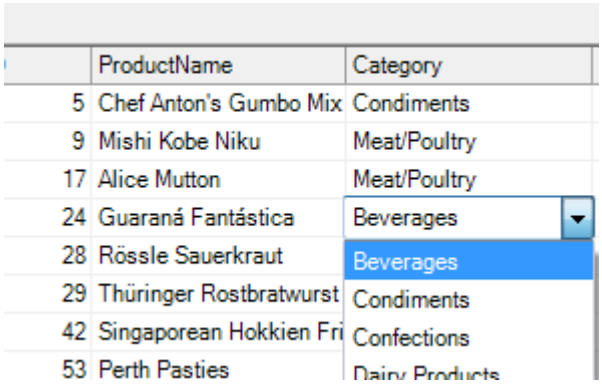
   As you can see, the **Category** and **Supplier** columns are not useful at all. You could remove them or customize the grid by writing some code to create new columns, but there's an easier way.

5. Select the grid in the designer and find the property called "**ControlHandler on c1DataSource1**" in the Properties window, as shown in the following picture:




   Remember, this is an extender property and will be available only if there is a C1DataSource component on the form. Set the **AutoLookup** property there to *True*.

6. When you are done, run the project again and look at the **Category** and **Supplier** columns. Notice that now the grid shows the category name and supplier's company name instead of the generic strings. Also notice that you can edit the product's category and the product's supplier by picking from a drop-down list, complete with auto search functionality.




7. Finally, click the column headers to sort the grid by **Category** or **Supplier** and notice that the sorting is performed based on the value displayed on the grid. This is what anyone would expect, but surprisingly it is not easy to achieve using regular data binding.

The string value (name) shown by the combo box is determined following these rules:

1. If the entity class overrides the **ToString** method, then the string representation of the entity is obtained using the overridden **ToString** method. This should return a string that uniquely represents the entity. For example, it could be the content of a **CompanyName** column, or a combination of **FirstName**, **LastName**, and **EmployeeID**. This is the preferred method because it is simple, flexible, and easy to implement using partial classes (so your implementation will not be affected if the entity model is regenerated).
2. If the entity class does not override the **ToString** method, but one of its properties has the **DefaultProperty** attribute, then that property is used as the string representation of the entity.
3. If the entity class does not override the **ToString** method and has no properties marked with the **DefaultProperty** attribute, then the first column that contains the string "Name" or "Description" in its name will be used as a string representation for the entities.
4. If none of the above applies, then no lookup is created for the given entity type.

## Customizing View

In many situations, you may want to use custom views that do not correspond directly to the tables and views provided by the database. LINQ is the perfect tool for these situations, allowing you to transform raw data using your language of choice using query statements that are flexible, concise, and efficient. C1DataSource makes LINQ even more powerful by making LINQ query statements live. That's why its LINQ implementation is called LiveLinq. We will show the power of LiveLinq in Live Views. For now, suffice it to say that you can transform your view to shape it whatever way you want using LINQ operators without losing full updatability and bindability.

Let's try, for example, one of the LINQ operators, **Select** (also called projection), to customize the fields (properties) of our view.

To customize a view, follow these steps:

1. Using the project we created to demonstrate Paging, add a new form with a C1DataSource component using the same **ContextType** as before. Note that you can make this the startup form to save time when you run the project.
2. Create a **ViewSource** in the **ViewSourceCollection** editor. Use the *Products* table in our sample database.
3. Add a **DataGrid** to the form, but unlike previous examples, we won't actually bind its C1DataSource at design time. We'll do this at run time in code because we want to implement a custom view.
4. Add the following directive statements to the form's code:

   **To write code in Visual Basic**

   ```
   Visual Basic
   Imports C1.LiveLinq.LiveViews
   ```

   **To write code in C#**

   ```
   C#
   using C1.LiveLinq.LiveViews;
   ```

5. In the form's Load event, add the following code to create a custom live view and bind it to the grid:

   **To write code in Visual Basic**

   ```
   Visual Basic
   dataGridView1.DataSource = _
   ```

```
(From p In C1DataSource1("Products").AsLive(Of Product)()
                    Select New With
                    {
                        p.ProductID,
                        p.ProductName,
                        p.CategoryID,
                        p.Category.CategoryName,
                        p.SupplierID,
                        .Supplier = p.Supplier.CompanyName,
                        p.UnitPrice,
                        p.QuantityPerUnit,
                        p.UnitsInStock,
                        p.UnitsOnOrder
                    }).AsDynamic()
```

**To write code in C#**

C#

```
dataGridView1.DataSource =
    (from p in c1DataSource1["Products"].AsLive<Product>()
        select new
            {
                p.ProductID,
                p.ProductName,
                p.CategoryID,
                CategoryName = p.Category.CategoryName,
                p.SupplierID,
                Supplier = p.Supplier.CompanyName,
                p.UnitPrice,
                p.QuantityPerUnit,
                p.UnitsInStock,
                p.UnitsOnOrder
            }).AsDynamic();
```

Here **c1DataSource1**["Products"] is a ClientCollectionView object. It is the view that is created by the view source that we set up in the designer. Note that if you need to access the view source itself in code, it is also available, as **c1DataSource.ViewSources["Products"]**. The AsLive() extension method call is needed to specify the item type of the view (*Product*) so LiveLinq operators can be applied to it. The result, c1DataSource1["Products"].AsLive<Product>(), is a View<Product>. The C1.LiveLinq.LiveViews.View is the main class of LiveLinq used for client-side live views. LINQ operators applied to live views preserve their updatability and bindability. They are the same usual LINQ operators, but the fact that they are applied to a live view gives them these additional features that are critically important for data binding applications.

**Note:** AsDynamic() must be applied to this view because its result selector (the select new... code) uses an anonymous class. This is a minor LiveLinq limitation, only for anonymous classes. If you forget to add AsDynamic() to such view, you will be reminded by an exception.

6. Save, build and run the application. The grid now shows the columns we defined in the 'select' clause of our LiveLinq view. Note also that all columns are modifiable. It may not seem like a big deal; however, it is an important feature that would be difficult to implement on your own for this customized view, especially when adding and deleting rows, which, as you can see here, is also supported. This is what we were referring to by stating that LiveLinq operators preserve updatability.

Bindability is achieved because the views are always 'live'; they aren't simple snapshots of static data. To prove the point, construct an exact replica of the form that we've just built. At this stage, you might find it easier to add a menu to your application to make accessing the forms you've created easier. Save, build and run the application. This time, open two instances of the form you just created and change some data in the grid on one of the forms. Notice how the corresponding data in the grid on the other form is changed automatically. Remember, you have not had to write a single line of code for the grids in these forms to synchronize the changes that you've just made.

You will see more of what LiveLinq can do in the Live Views topic.

## Working with DataSources in Code

Up to this point, we have been setting up data sources directly on the designer surface with very little code. **DataSource for Entity Framework** has made it very easy, but sometimes you want or need to do everything in code. C1DataSource makes this possible as well. Everything we did previously can be done at run time in code.

An obvious way to go about this would be to use the **ClientViewSource** object that we have, in effect, been setting up in the designer as elements of the **ViewSourceCollection** of a C1DataSource, given that it can be created on its own without a C1DataSource. We could, however, take a step further back and use a lower level class ClientView<T>. This would provide full control over loading data from the server and, since it is derived from C1.LiveLinq.LiveViews.View<T>, we can apply any LiveLinq operators to it. The ability to bind this to any GUI control whose datasource can be set to a View<T> also means that we'll end up with a fully editable view of our data.

Server-side filtering is, perhaps, the most common operation, because no one usually wants entire database tables brought to the client unrestricted. Earlier we saw how C1DataSource made it simple to perform without code, but now we'll try it in run-time code.

To begin using C1DataSource in run-time code without a C1DataSource, add a few lines to our project's main class to create a global client-side data cache. When we used C1DataSource, it was created for us behind the scenes. Now we can create it explicitly using the following code:

**To write code in Visual Basic**

Visual Basic

```
Imports C1.Data.Entities

Public Class Program
    Public Shared ClientCache As EntityClientCache
    Public Shared ObjectContext As NORTHWNDEntities

    <STAThread()> _
    Shared Sub Main()

        ObjectContext = New NORTHWNDEntities
        ClientCache = New EntityClientCache(ObjectContext)

        Application.EnableVisualStyles()
        Application.SetCompatibleTextRenderingDefault(False)
        Application.Run(New MainForm())
    End Sub
End Class
```

**To write code in C#**

C#

```
using C1.Data.Entities;
```

```
static class Program
{
    public static EntityClientCache ClientCache;
    public static NORTHWNDEntities ObjectContext;

    [STAThread]
    static void Main()
    {
        ObjectContext = new NORTHWNDEntities();
        ClientCache = new EntityClientCache(ObjectContext);

        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new MainForm());
}
```

This code creates a single application-wide (static) **ObjectContext** and associates it with **EntityClientCache**. As noted previously in The Power of Client Data Cache topic, the ability to have a single context (and cache) for the entire application is a great simplification made possible by C1DataSource.

To perform server-side filtering in run-time code, follow these steps:

1. Add a new form using the project created to demonstrate Customizing View.
2. Add a grid (**dataGridView1**), a combo box (**comboBox1**), and a button (**btnSaveChanges**) to the form.
3. Add the following code to the form class:

**To write code in Visual Basic**

Visual Basic

```
Imports C1.Data.Entities
 Imports C1.Data

 Public Class DataSourcesInCode

    Private _scope As EntityClientScope

    Public Sub New()

        InitializeComponent()

        _scope = Program.ClientCache.CreateScope()

        Dim viewCategories As ClientView(Of Category) = _scope.GetItems(Of Category)()

        comboBox1.DisplayMember = "CategoryName"
        comboBox1.ValueMember = "CategoryID"
        comboBox1.DataSource = viewCategories

        BindGrid(viewCategories.First().CategoryID)
    End Sub

    Private Sub BindGrid(categoryID As Integer)
        dataGridView1.DataSource =
                (From p In _scope.GetItems(Of Product)().AsFiltered(Function(p As Product) p.CategoryID.Value =
categoryID)
                 Select New With
                 {
                    p.ProductID,
                    p.ProductName,
```

```
                p.CategoryID,
                p.Category.CategoryName,
                p.SupplierID,
                .Supplier = p.Supplier.CompanyName,
                p.UnitPrice,
                p.QuantityPerUnit,
                p.UnitsInStock,
                p.UnitsOnOrder
            }).AsDynamic()
    End Sub

    Private Sub btnSaveChanges_Click(sender As System.Object, e As System.EventArgs) Handles
btnSaveChanges.Click
        Program.ClientCache.SaveChanges()
    End Sub

    Private Sub comboBox1_SelectedIndexChanged(sender As System.Object, e As System.EventArgs) Handles
comboBox1.SelectedIndexChanged
        If comboBox1.SelectedValue IsNot Nothing Then
            BindGrid(CType(comboBox1.SelectedValue, Integer))
        End If
    End Sub
 End Class
```

**To write code in C#**

C#

```
namespaceTutorialsWinForms
 {
    using C1.Data.Entities;
    using C1.Data;

    public partial class DataSourcesInCode : Form
    {
        private EntityClientScope _scope;

        public DataSourcesInCode()
        {
            InitializeComponent();

            _scope = Program.ClientCache.CreateScope();

            ClientView<Category> viewCategories =_scope.GetItems<Category>();

            comboBox1.DisplayMember = "CategoryName";
            comboBox1.ValueMember = "CategoryID";
            comboBox1.DataSource = viewCategories;

            BindGrid(viewCategories.First().CategoryID);
        }

        private void comboBox1_SelectedValueChanged(object sender, EventArgs e)
        {
            if (comboBox1.SelectedValue != null)
```

```
            BindGrid((int)comboBox1.SelectedValue);
        }

        private void BindGrid(int categoryID)
        {
            dataGridView1.DataSource =
                (from p in _scope.GetItems<Product>().AsFiltered(
                    p => p.CategoryID == categoryID)
                 select new
                 {
                    p.ProductID,
                    p.ProductName,
                    p.CategoryID,
                    CategoryName = p.Category.CategoryName,
                    p.SupplierID,
                    Supplier = p.Supplier.CompanyName,
                    p.UnitPrice,
                    p.QuantityPerUnit,
                    p.UnitsInStock,
                    p.UnitsOnOrder
                 }).AsDynamic();
        }

        private void btnSaveChanges_Click(object sender, EventArgs e)
        {
            Program.ClientCache.SaveChanges();
        }

    }
}
```

4. Save, build and run your application. You should see similar results to those you saw in the Server-Side Filtering example, the only difference being that this time we've implemented it all in code.

Let's take a closer look at some of the code we've just written.

The private field **_scope** is the form's gateway to the global data cache. It is a pattern we recommend you follow in all the forms where you do not employ a C1DataSource component directly, as that does this for you automatically. It ensures that the entities the form needs stay in the cache while the form is alive, and that they are automatically released when all forms (scopes) holding them are released.

Creating a view showing all categories for the combo box is simple:

**To write code in Visual Basic**

| Visual Basic |
| --- |
| Dim viewCategories As ClientView(Of Category) = _scope.GetItems(Of Category)() |

**To write code in C#**

| C# |
| --- |
| ClientView<Category> viewCategories = _scope.GetItems<Category>(); |

To create the view to which the grid is bound that only provides those products associated with the chosen category in the combo box required one additional operator; AsFiltered(<predicate>).

**To write code in Visual Basic**

```
Visual Basic
From p In _scope.GetItems(Of Product)().AsFiltered(Function(p As Product) p.CategoryID.Value = categoryID)
```

**To write code in C#**

```
C#
from p in _scope.GetItems<Product>().AsFiltered(p => p.CategoryID == categoryID)
```

Note that when this query is executed, the result does not necessarily require a round trip to the server to retrieve the products requested. The cache is examined first to see if it already contains the requested data, either because the required data has already been requested once before within this form or from another form in the application. Or, possibly a completely separate query run elsewhere in the application had requested that all products be returned, so the cache would already have all the product data. Again, this is a fundamental strength of C1DataSource. By providing your application with a global cache of data, its performance is continually improved throughout its lifetime.

Here we chose to create a new view, and bind the grid to it, every time the user selects a new category in the combo box (see the combo box's **SelectedValueChanged** event). However, we could have avoided the need to create new views all the time and instead created one single view using a special **BindFilterKey**, which we'll learn more about in the Simplifying MVVM topic.

So, in summary, we replicated in code what we did on the design surface with C1DataSource in Server-Side Filtering. We have even thrown in a little extra; we customized the fields shown in the grid columns as we did in Customizing View by adding a **Select** to our LiveLinq statement:

**To write code in Visual Basic**

```
Visual Basic
Select New With
{
    p.ProductID,
    p.ProductName,
    p.CategoryID,
    p.Category.CategoryName,
    p.SupplierID,
    Supplier = p.Supplier.CompanyName,
    p.UnitPrice,
    p.QuantityPerUnit,
    p.UnitsInStock,
    p.UnitsOnOrder
}
```

**To write code in C#**

```
C#
select new
```

```
{
    p.ProductID,
    p.ProductName,
    p.CategoryID,
    CategoryName = p.Category.CategoryName,
    p.SupplierID,

   Supplier = p.Supplier.CompanyName,

   p.UnitPrice,
    p.QuantityPerUnit,
    p.UnitsInStock,
    p.UnitsOnOrder
};
```

Had we just wanted the raw product data returned from the table without any special formatting, we could have simply said;

```
    select p;
```

## Live Views

Live views is a powerful feature, so let's take a little more time to see what live views can do. Live views are designed to make data binding more powerful, so powerful, in fact, that you can develop virtually entire applications with just LINQ (in its LiveLinq form) and data binding.

Live views, called **LiveLinq**, a part of **DataSource for Entity Framework**, is a client-side, in-memory feature applicable not only to Entity Framework and RIA Services data, but to any observable collections in memory, including XML (LINQ to XML object model) and ADO.NET DataSets.

So, for example, you can use live views over Entity Framework data and some other data (for example, XML retrieved from some web service) to integrate that data and to provide easy full-featured data binding to the integrated data. This is a powerful tool for building applications that get data from various sources. For now we're going to concentrate on how we can use it with the Entity Framework, but if you'd like to explore **LiveLinq** in greater depth, see the ComponentOne LiveLinq documentation.

In fact, we already saw an example of such customization in Customizing View. But there we only changed the properties (fields) of the view and only applied one LINQ operator, **Select**. Let's apply some more LINQ operators to transform the view. What we do here will be similar to what we did in Customizing View, but instead of using C1DataSource, we'll be doing everything in code.

To use live views, follow these steps:

1. Add a new form using the project created to demonstrate Working with DataSources in Code, and add a data grid, **dataGridView1**, to the form.
2. Add the following code to the form class. Here we are following the pattern recommended in Working with DataSources in Code.

   **To write code in Visual Basic**

   Visual Basic

   ```
   Private _scope As EntityClientScope = Program.ClientCache.CreateScope()
   ```

   **To write code in C#**

   C#

```
private EntityClientScope _scope = Program.ClientCache.CreateScope();
```

3. Getting *Products* data from the scope, we create a live view and bind the grid to that view in the form's constructor:

**To write code in Visual Basic**

```
Visual Basic
_viewProducts =
    (From p In _scope.GetItems(Of Product)()
     Where Not p.Discontinued And p.UnitPrice >= 30
     Order By p.UnitPrice
     Select New With
     {
        p.ProductID,
        p.ProductName,
        p.CategoryID,
        p.Category.CategoryName,
        p.SupplierID,
        .Supplier = p.Supplier.CompanyName,
        p.UnitPrice,
        p.QuantityPerUnit,
        p.UnitsInStock,
        p.UnitsOnOrder
     }).AsDynamic()
 dataGridView1.DataSource = _viewProducts
```

**To write code in C#**

```
C#
_viewProducts =
(from p in _scope.GetItems<Product>()
                        where !p.Discontinued && p.UnitPrice >= 30
                        orderby p.UnitPrice
                        select new
                        {
                            ProductID = p.ProductID,
                            ProductName = p.ProductName,
                            CategoryID = p.CategoryID,
                            CategoryName = p.Category.CategoryName,
                            SupplierID = p.SupplierID,
                            Supplier = p.Supplier.CompanyName,
                            UnitPrice = p.UnitPrice,
                            QuantityPerUnit = p.QuantityPerUnit,
                            UnitsInStock = p.UnitsInStock,
                            UnitsOnOrder = p.UnitsOnOrder
                        }).AsDynamic();

dataGridView1.DataSource = _viewProducts;
```

In this example, we applied several **LiveLinq** operators: *Where*, *OrderBy*, and *Select*. We defined our view as containing products that aren't discontinued and have a unit price of at least 30, and we sorted our view by

unit price.

We chose to store the view in a private field **_viewProducts** here:

**To write code in Visual Basic**

```
Visual Basic
Private _viewProducts As View(Of Object)
```

**To write code in C#**

```
C#
private View<dynamic> _viewProducts;
```

That is only because we will need it later. If we did not, we could use a local variable for the view.

Syntactically, the query that we wrote for **_viewProducts** is just standard LINQ. It could be done without C1DataSource, with standard LINQ to Objects, and the code would be the same, only instead of **_scope.GetItems<Product>()**, you would use something like **ObjectContext.Products**. In fact, we will try to do just that in a moment, once we run the project, to compare what we get from standard LINQ with what we get from **LiveLinq**.

4. Now run the project. You see that the grid shows the filtered set of products, "expensive" products that aren't discontinued, in the order that we specified with the columns that we specified. Note also that all columns are modifiable, and you can even add and delete rows in the grid. Also note that you can sort the grid at run time by clicking column headers.

To appreciate this full data binding support, compare it with what you would get if you did not use C1DataSource, but if you used standard LINQ to Objects instead. It's easy to compare; just replace **_scope.GetItems<Product>()** in the code with **Program.ObjectContext.Products**. Note that you will also need to remove the type **C1.LiveLinq.LiveViews.View** and use the 'var' keyword instead for it to compile, because it will no longer be a live view. The difference is obvious: with standard LINQ, the data in the grid is read-only, and the grid does not support sorting.

But live views offer even more great features. Standard LINQ to Objects produces snapshots of the data that cannot reflect changes in the source data, except some simple property changes, and even then under the strict proviso that you don't utilize a custom **Select** in your LINQ statement. Live Views, on the other hand, provide dynamic 'live' views that automatically reflect changes in their source data. As such, they simplify application development because you can, in most cases, rely on data binding to automate 'live' changes in views without the need to synchronize the changes in different parts of the application with code.

To see that the views are indeed 'live', open two forms side-by-side. Run your application and open up the Custom Columns form, which we built in Customizing View, and the Client Side Querying form, which we just built here. Make some changes to a product in the **CustomColumns** form and observe how they are reflected in the other form. If, for example, you were to increase the **UnitCost** of a product to above **30**, then it would automatically appear in the second form.

To see another example of how live views automatically synchronize themselves with changes in underlying data, follow these steps:

1. Add a live view member of the user control class:

   **To write code in Visual Basic**

   ```
   Visual Basic
   Private _seafoodProductsView As ClientView(Of Product)
   ```

**To write code in C#**

```
C#
private ClientView<Product> _seafoodProductsView;
```

2. Add the following code to the form's constructor:

**To write code in Visual Basic**

```
Visual Basic
_seafoodProductsView = _scope.GetItems(Of Product)().AsFiltered(Function(p) p.CategoryID.Value = 8)
```

**To write code in C#**

```
C#
_seafoodProductsView = _scope.GetItems<Product>().AsFiltered(p => p.CategoryID == 8);
```

3. Add two buttons, named **btnRaise** and **btnCut**, to the form and add the following handlers to the form's code :

**To write code in Visual Basic**

```
Visual Basic
Private Sub raiseButton_Click(sender As System.Object, e As System.EventArgs)
                                For Each p In _seafoodProductsView
                                    p.UnitPrice *= 1.2
                                Next
                            End Sub
     Private Sub cutButton_Click(sender As System.Object, e As System.EventArgs)
                                For Each p In _seafoodProductsView
                                    p.UnitPrice /= 1.2
                                Next
                            End Sub
```

**To write code in C#**

```
C#
private void raiseButton_Click(object sender, EventArgs e)
{
    foreach (var p in _seafoodProductsView)
        p.UnitPrice *= 1.2m;
}
private void cutButton_Click(object sender, EventArgs e)
{
    foreach (var p in _seafoodProductsView)
        p.UnitPrice /= 1.2m;
}
```

4. Save, build and run the application. As you press the buttons, notice how seafood products appear in the grid because their unit price is now either greater than or equal to **30** or how they disappear when their unit price falls below **30**. All of this happens automatically. You did not have to write any special code to refresh or synchronize the grid.
5. To see how live views can make almost any GUI-related code easier to write (and less error-prone), let's add a

label that shows the current row count. Without C1DataSource, this would usually be done in a method counting the rows, and that method would have to be called in every place in the code where that count can change. In this example, there would be three such places: on initial load, and in the two methods called when the buttons are pressed, raiseButton_Click and cutButton_Click. So it is not very easy to synchronize display with changing data even in this simple example, not to speak of a real application. Live views make all this synchronization code unnecessary. We already saw how it works for a view containing filtering, ordering, and projection (Where, OrderBy, and Select). It works as well for views containing aggregation operations, such as Count. A little difference here is that regular LINQ Count returns a single number, to which you can't bind a control, so C1DataSource provides a special operation LiveCount that returns a live view instead of a single number, so you can use it in data binding. We can create this binding with a single line of code:

```
labelCount.DataBindings.Add(new Binding("Text",
                _viewProducts.LiveCount(), "Value"));
```

## Simplifying MVVM

The Model-View-View-Model (**MVVM**) pattern is gaining in popularity as developers realize the benefits it gives their applications, making them easier to maintain and test and, particularly in the case of WPF and Silverlight applications, allowing a much clearer division of labor between the designer of the UI and the creator of the code that makes it work. However, without effective tools to aid program development based on **MVVM** patterns, programmers can actually find themselves working harder because of the need to implement an extra layer (the View Model) and ensure that data is properly synchronized between that and the Model. This extra burden isn't onerous when you have a relatively small application with just a few simple collections (and best practice with MVVM is to use **ObservableCollection** as the datasource), but the bigger it becomes and the more collections it spawns, the worse it becomes. **DataSource for Entity Framework** can ease the burden.

C1DataSource lets you use live views as your view model. Just create live views over your model collections and use them as your view model. Live views are synchronized automatically with their sources (model), so you don't need *any* synchronization code - it's all automatic. And live views are much easier to create than to write your own view model classes using **ObservableCollection** and filling those collections manually in code. You have all the power of LINQ at your disposal to reshape model data into live views. So, not only does synchronization code disappear, but the code creating view models is dramatically simplified.

To demonstrate how easy it is to follow the **MVVM** pattern using live views, let's create a form combining all features from two previous examples: the **Category-Products** master-detail from Working with DataSources in Code and the reshaping/filtering/ordering of data from Live Views. It will be a **Category-Products** view, showing non-discontinued products whose unit price is at least **30**, ordered by unit price, and displaying a customized set of product fields in a master-detail form where the user can select a category to show the products of that category. We'll follow the **MVVM** pattern. The form (view), called **CategoryProductsView**, only hosts GUI controls (a combo box and a grid) and does not have any code except what is used to set the data source to the view model, like this:

**To write code in Visual Basic**

Visual Basic

```
PublicClassCategoryProductsView

    PrivateviewModel AsCategoryProductsViewModel= NewCategoryProductsViewModel()

    PublicSubNew()
        InitializeComponent()

        comboBox1.DisplayMember = "CategoryName"
        comboBox1.ValueMember = "CategoryID"
        comboBox1.DataSource = viewModel.Categories
        dataGridView1.DataSource = viewModel.Products

    EndSub
```

```
EndClass
```

**To write code in C#**

```
C#
publicpartialclassCategoryProductsView: Form
{
    CategoryProductsViewModelviewModel = newCategoryProductsViewModel();
    publicCategoryProductsView()
    {
        InitializeComponent();
        comboBox1.DisplayMember = "CategoryName";
        comboBox1.ValueMember = "CategoryID";
        comboBox1.DataSource = viewModel.Categories;
        dataGridView1.DataSource = viewModel.Products;
    }
}
```

All logic is in the view model class, separate from the GUI. More exactly, there are three classes: **CategoryViewModel**, **ProductViewModel**, and **CategoryProductsViewModel**. The first two are simple classes defining properties with no additional code:

**To write code in Visual Basic**

```
Visual Basic
PublicClassCategoryViewModel
    PublicOverridablePropertyCategoryID AsInteger
    PublicOverridablePropertyCategoryName AsString
 EndClass


 Public ClassProductViewModel
    PublicOverridablePropertyProductID AsInteger
    PublicOverridablePropertyProductName AsString
    PublicOverridablePropertyCategoryID AsInteger?
    PublicOverridablePropertyCategoryName AsString
    PublicOverridablePropertySupplierID AsInteger?
    PublicOverridablePropertySupplierName AsString
    PublicOverridablePropertyUnitPrice AsDecimal?
    PublicOverridablePropertyQuantityPerUnit AsString
    PublicOverridablePropertyUnitsInStock AsShort?
    PublicOverridablePropertyUnitsOnOrder AsShort?
 EndClass
```

**To write code in C#**

C#

```
public classCategoryViewModel
{
    public virtual int CategoryID { get; set; }
    public virtual string CategoryName { get; set; }
}
public classProductViewModel
{
    public virtual int ProductID { get; set; }
    public virtual string ProductName { get; set; }
    public virtual int? CategoryID { get; set; }
    public virtual string CategoryName { get; set; }
    public virtual int? SupplierID { get; set; }
    public virtual string SupplierName { get; set; }
    public virtual decimal? UnitPrice { get; set; }
    public virtual string QuantityPerUnit { get; set; }
    public virtual short? UnitsInStock { get; set; }
    public virtual short? UnitsOnOrder { get; set; }
}
```

And here is the code for the **CategoryProductsViewModel** class:

**To write code in Visual Basic**

Visual Basic

```
PublicClassCategoryProductsViewModel
    Private_scope AsEntityClientScope
    Private_categories AsBindingSource
    PublicPropertyCategories AsBindingSource
        Get
            Return_categories
        EndGet
        PrivateSet(value AsBindingSource)
            _categories = value
        EndSet
    EndProperty
    Private_products AsBindingSource
    PublicPropertyProducts AsBindingSource
        Get
            Return_products
```

```
        EndGet
        PrivateSet(value AsBindingSource)
            _products = value
        EndSet
    EndProperty

    PublicSubNew()
        _scope = Program.ClientCache.CreateScope()

        DimCategoriesView AsObject=
            Fromc In_scope.GetItems(OfCategory)()
            SelectNewCategoryViewModelWith
            {
                .CategoryID = c.CategoryID,
                .CategoryName = c.CategoryName
            }
        Categories = NewBindingSource(CategoriesView, Nothing)

        DimProductsView AsObject=
            Fromp In_scope.GetItems(OfProduct)().AsFilteredBound(Function(p)
p.CategoryID.Value).BindFilterKey(Categories, "Current.CategoryID").Include("Supplier")
            SelectNewProductViewModelWith
            {
                .ProductID = p.ProductID,
                .ProductName = p.ProductName,
                .CategoryID = p.CategoryID,
                .CategoryName = p.Category.CategoryName,
                .SupplierID = p.SupplierID,
                .SupplierName = p.Supplier.CompanyName,
                .UnitPrice = p.UnitPrice,
                .QuantityPerUnit = p.QuantityPerUnit,
                .UnitsInStock = p.UnitsInStock,
                .UnitsOnOrder = p.UnitsOnOrder
            }
        Products = NewBindingSource(ProductsView, Nothing)

    EndSub
EndClass
```

**To write code in C#**

C#

```
public classCategoryProductsViewModel

{
    private C1.Data.Entities.EntityClientScope _scope;

    publicBindingSource Categories { get; private set; }

    public BindingSource Products { get; private set; }

    public CategoryProductsViewModel()

    {
```

```
        _scope = Program.ClientCache.CreateScope();
        Categories = new BindingSource(
            from c in _scope.GetItems<Category>()
            select new CategoryViewModel()
            {
                CategoryID = c.CategoryID,
                CategoryName = c.CategoryName
            }, null);
        Products = new BindingSource(
            from p in _scope.GetItems<Product>().AsFilteredBound(p =>
            p.CategoryID).BindFilterKey(Categories,
                "Current.CategoryID").Include("Supplier")
            select new ProductViewModel()
            {
                ProductID = p.ProductID,
                ProductName = p.ProductName,
                CategoryID = p.CategoryID,
                CategoryName = p.Category.CategoryName,
                SupplierID = p.SupplierID,
                SupplierName = p.Supplier.CompanyName,
                UnitPrice = p.UnitPrice,
                QuantityPerUnit = p.QuantityPerUnit,
                UnitsInStock = p.UnitsInStock,
                UnitsOnOrder = p.UnitsOnOrder
            }, null);
        }
}
```

Basically, it contains just two LiveLinq statements, nothing more. The statements (creating live views, see Live Views) are wrapped in **BindingSource** constructors to add currency, current item, support to the **Categories** and **Products** collections exposed by the view model class. Note that using **BindingSource** is only necessary in WinForms, because the WinForms platform is not as well suited for MVVM as WPF or Silverlight. Note also that because we use **BindingSource**, you need to add the following statement to the code file (in WinForms only):

```
using System.Windows.Forms;
```

Similar to what we saw in Working with DataSources in Code, using **AsFilteredBound**() gives us server-side filtering. We also connected the filter key, which is the **Product.CategoryID** property, to the **CategoryID** selected by the user using a combo box event. Here we can't do this because we must keep our code independent of the GUI. So we use a BindFilterKey method to bind the filter key to the **Category.CategoryID** property of the item currently selected in the **Categories** collection. This is one reason why we need to support currency in the **Categories** collection and why we wrapped it in a **BindingSource** to get currency support in WinForms.

The **Include** ("Supplier") operator is not strictly necessary; it is used here for performance optimization. Without it,

**Entity Framework** will fetch **Supplier** objects lazily, one-by-one, every time the user accesses an element of the **Products** collection whose **Supplier** was not yet fetched. This can cause delays, and it's generally much less efficient to fetch data in single rows rather than in batches, so we opted out of lazy loading here using **Include**("Supplier"), which tells the **Entity Framework** to fetch supplier information in the same query with products.

# Using C1DataSource in MVVM with other MVVM framework

C1DataSource can be used to build Model-View-View-Model (**MVVM**) applications with any other MVVM frameworks.

C1DataSource offers several features to make your MVVM development easier:

- **Simplifies MVVM programming**

  Given that C1DataSource can be used to simplify MVVM programming, as seen in the Simplifying MVVM topic, it's clearly a tool that can be used for MVVM.

- **Helps create view model classes and alleviate code bloating**

  There are a plethora of tools and frameworks that developers can use to aid them in their work with MVVM, but very few that can help them create view model classes. The majority are designed to help with tasks such as passing commands and messages between the view and view model. Creating view model classes and then synchronizing them with model data is left almost entirely to manual coding. This is the primary cause of code bloating in most MVVM applications and precisely the one that C1DataSource is designed to alleviate in a way that is entirely compatible with other frameworks.

- **Allows you to use any framework and live views**

  You can use any framework you like to assist your MVVM application development and simply call on C1DataSource to provide live views to help you create view model classes.

To demonstrate these important points, we provide a sample based on the code from the well-known article by Josh Smith, one of the authors of MVVM, "WPF Apps With The Model-View-ViewModel Design Pattern" (http://msdn.microsoft.com/en-us/magazine/dd419663.aspx).

Please refer to the pre-installed product samples through the following path:

### Documents\ComponentOne Samples\WinForms

Essentially all the files in our modified sample are the same as the originals (bar a few cosmetic changes) except one (ViewModels\OrdersViewModel.cs).

In this file, we build the view model class the C1DataSource way, using live views. You can see how many re-shaping functions are applied to model data to construct a view model, all done exclusively through LINQ. This made it easy and required little code. The best part is that it synchronizes automatically with model data when data in either of the two layers is changed - no synchronization code was necessary.

The fact that we only changed the way that the view model classes themselves were created (they are still derived from the original base class 'ViewModelBase') and made no other changes to the framework code that Josh Smith had employed in his original sample should serve as an example that C1DataSource is entirely compatible with other frameworks. You can continue to use your preferred frameworks when working with MVVM, but now you have an additional tool to make your MVVM development even easier.

# Programming Guide

The following sections provide information concerning C1DataSource programming. For additional information about LiveLinq-specific features see the LiveLinq Programming Guide.

# Working with Entities in Code

**DataSource for Entity Framework** is non-intrusive in the sense that you can do whatever you want with entities in code using the regular Entity Framework (or RIA) methods and properties and that will work well with C1DataSource. You can add, delete and modify entities using the regular methods (no special C1DataSource object model is needed for that), and C1DataSource collections will automatically reflect changes that you make to the entities. For example, to add a new entity, you don't need to use some special C1DataSource method; therefore, none exists. Just add a new entity as you always do in EF (or RIA) and it will automatically appear in corresponding C1DataSource collections. If an C1DataSource collection has a *Where* condition, it will appear there only if it satisfies the condition. Same with deleting and modifying entities: do it the regular EF (or RIA) way, and C1DataSource reflects the changes automatically, so they are automatically reflected in bound controls.

However, there is one important restriction that must be strictly observed. It does not limit what you can do; it only requires you in some (relatively rare) cases to add a method call to your code notifying C1DataSource of what you are doing:

**CAUTION**

**Never fetch entities from the server by directly querying the context without notifying DataSource**.

All entities must be either fetched by one of the C1DataSource's **GetItems** methods, or, if you want to fetch them by querying an object context directly, you must call the **ClientScope.AddRef** method to notify C1DataSource that you fetched entities from the server.

If you use the C1DataSource control or **ClientViewSource**, you fetch entities either implicitly if **AutoLoad** = true, or explicitly when you call the **ClientViewSource.Load** method. In both cases it is done through C1DataSource, so C1DataSource is aware of the newly fetched entities. When you need to fetch entities in code without using ClientViewSource, the standard way to do it is by using one of the C1DataSource's **GetItems** methods (EntityClientScope.GetItems|keyword=EntityClientScope.GetItems method, RiaClientScope.GetItems|keyword=RiaClientScope.GetItems method). Here too, C1DataSource is aware of the fetched entities. However, occasionally you may need to retrieve entities by issuing a query directly to the server without involving C1DataSource. For example, in Entity Framework you can take an ObjectContext that is used by C1DataSource and create queries:

```
ObjectContext context = scope.ClientCache.ObjectContext;
```

// or

```
ObjectQuery query = ((NORTHWNDEntities)context).Customers;
```

// or

```
query = context.CreateObjectSet<Customer>();
```

// or

```
query = context.CreateQuery<Customer>("...");
```

In RIA Services such code can look like this:

```
DomainContext context = scope.ClientCache.DomainContext;
var query = ((DomainService1)context).Customers;

// or

var entities = context.Load(
((DomainService1)context).GetCustomersQuery()).Entities;
```

Having a query, you can get entities directly from the server by enumerating the query result

foreach (Customer c in query) /* do something */

or by binding a control to it:

```
dataGrid.ItemsSource = query;
```

If you do this without calling **ClientScope.AddRef**, you can get entities from the server without C1DataSource knowing about it. Those entities will be in the same cache with other entities, indistinguishable from them, so C1DataSource will manage them as other entities, including possibly releasing them, evicting them from the cache when it does not need them. That will make the entities inaccessible, but your program may still need them! So, very bad things can happen if you fetch entities to a C1DataSource-governed context without using C1DataSource and without notifying it that entities are fetched. Fortunately, adding that notification is easy, just call **ClientScope.AddRef** for all fetched entities like this:

```
foreach (Customer c in query)
{
   scope.AddRef(c);
   // do something
}
```

It will tell C1DataSource that the entities should not be released as long as the scope is alive.

## Using ClientView in Code

**DataSource for Entity Framework** supports both visual and "all code" style of programming:

- Use the C1DataSource control if you want point-and-click, RAD-style application development.
- Use the **ClientViewSource** class if you want to separate your code from GUI but keep the ease of use of the C1DataSource. The ClientViewSource class is independent of GUI; it can be used in code with any of the GUI platforms (WPF, Silverlight, WinForms). It can be used completely separately from GUI, including in the view model layer of an MVVM application. At the same time, the **ClientViewSource** is the same object C1DataSource uses, so you can keep the ease of use characteristic of C1DataSource (but code-only, without visual designers). In fact, C1DataSource is just a collection of **ClientViewSource** objects plus visual designer support for them.
- For the most complete control over your code, you can use the third, lowest level of the C1DataSource object mode: the **ClientView** class. If you prefer pure code, especially (but not only) using the MVVM pattern, this is the recommended level. It is still easy to program with, it is mostly based on LINQ which promotes a functional style of programming, intuitive and expressive.

The rest of this section is devoted to programming using the **ClientView** class.

## Creating Client Views

ClientView objects represent queries that can be executed on the server. In that regard, they provide the same

functionality as queries of Entity Framework and RIA Services. When you start by creating the first ClientView in a newly created context, that's exactly what you get: an EF (or RIA) query that is executed on the server; the resulting entities are fetched to the client and made available for data binding and/or programmatic access on the client. When you continue working on the client, that is, modify some entities on the client and query for more entities, your client views start exhibiting richer behavior that you would not get from mere EF (or RIA) queries:

- Client views are *live*. When you modify (or add, or delete) entities on the client, client views are automatically updated to reflect changed data.
- Client views make use of the C1DataSource client-side data cache. When you create a new client view or change an existing one, querying it does not necessarily require a round-trip to the server. C1DataSource looks in the cache first and uses the cache if the query can be satisfied without going to the server. That happens when a query is repeated, but not only in that case. C1DataSource is smart enough to detect various cases where a query can be satisfied without going to the server.

The starting point for creating client views are the **GetItems** methods of **EntityClientScope** (**RiaClientScope**):

```
ClientView<Product> products = _scope.GetItems<Product>();
```

or, in RIA Services (Silverlight):

```
ClientView<Product> products = _scope.GetItems<Product>("GetProducts");
```

(in RIA Services you need to specify the name of a query method in your domain service).

Having thus started with a base query, you can then apply filtering and paging to the view. Filtering and paging are operations that are performed on the server (if the data is not found in the cache), so applying them to a **ClientView** results in another **ClientView**. You can also apply other LINQ operators to a client view, such as grouping, sorting, and so on. That results in a **View**, which is the base class of **ClientView**. It is also a live view but it does not need ClientView functionality because those operators don't need the server; they can be performed entirely on the client.

## Server-Side Filtering Views

To apply filtering to a client view, use the **AsFiltered** method method returning **FilteredView**, a class derived from **ClientView**:

```
FilteredView<Product> productsByCategory = products.AsFiltered(p => p.CategoryID == categoryID);
```

Here **categoryID** is a variable or parameter that is assigned a value somewhere else in the code. When you create this view, with a certain value of **categoryID**, say 1, it will retrieve products with **p.CategoryID = 1**. It will do it using the cache, without a trip to the server, if possible; if not, it will fetch those entities from the server (that's the cache-awareness feature of client views). Unlike a standard EF (or RIA) query, it will also take into account entities that are added on the client and don't yet exist on the server, and entities modified on the client so they now satisfy the filter condition (that's the *live* feature of client views).

Unlike a standard EF (or RIA) query, this view will remain up to date (live) after it was first used. If you modify data on the client and you now have more (or less) entities satisfying the condition, the view will be automatically updated to reflect the changed data. And if you have GUI controls bound to it, those will be updated too.

If you use the **AsFiltered** method with key selector function

```
FilteredView<Product> productsByCategory = products.AsFilteredBound(p => p.CategoryID);
```

you will be able to set/change the filter condition without creating a new view every time:

```
productsByCategory.FilterKey = categoryID;
```

There is also a convenience method, **BindFilterKey**, you can use to bind the filter key of a view to a property (or property path) of an object:

```
FilteredView<Product> productsByCategory = products.AsFilteredBound(p => p.CategoryID)
        .BindFilterKey(comboBox1, "SelectedValue")
```

## Server-Side Paging Views

The other server-side operation, paging, is specified with the **Paging** method returning **PagingView**, a class derived from **ClientView**:

```
PagingView<Product> productsPaged = products.Paging(p => p.ProductName, 20);
```

A paging view contains a segment (page) of data at any given time. To create a paging view, you need to specify the page size (20 in the example above) and a key selector by which to sort, because paging can only be done over sorted data.

Since a paging view is a ClientView, it supports the two fundamental features of client views: it is cache-aware and live. The performance-enhancing cache-awareness of a paging view allows it to avoid round-trips to the server when a page is requested repeatedly. And since a paging view is live (as any ClientView), it reflects changes that you make to the data on the client. For example, if you delete an entity that is currently in a paging view, it will automatically adjust itself to the changed data, including, if needed, fetching some entities from the server to occupy vacant places on the page.

### Other Client View Operators

**Progressive loading**

If you have a client view that is too big to load all entities at once without a delay, you can make it load incrementally, progressively, using the **ProgressiveLoading** method:

```
ProgressiveView<Product> moreResponsiveView = productsByCategory.ProgressiveLoading(p => p.ProductName, 100);
```

Here we intentionally used productsCategory (a **FilteredView**) instead of products, to show that any client view can be made progressive (except paging view where it does not make sense).

A progressive view loads data in portions, batches making data available on the client immediately after each portion has been loaded.

To create a progressive view, you need to specify the load size (100 in the example above) and a key selector by which to sort, because data must be sorted on the server to perform this kind of loading.

**Include**

Working with Entity Framework, you sometimes need to specify that related entities must be fetched together with your entity query. For example, querying for orders, you may want to get customer information for the orders to the client in the same query that fetches orders, to avoid fetching them one-by-one later (which is considerably slower than fetching all in one query). You can do this in Entity Framework by using the **Include** method of **ObjectQuery**. The same functionality is available in DataSource ClientView.Include method:

```
ClientView<Order> ordersWithCustomerInfo = orders.Include("Customer");
```

## Live Views

Client views are live; they are kept automatically up-to-date with changing data. But live views functionality is more general, live views (objects of class C1.LiveLinq.LiveViews.View from which **ClientView** is derived) can be defined on data of any kind, including but not limited to entities, and they support more LINQ query operators, such as grouping, sorting, joins and more (see ComponentOne LiveLinq).

All such operations (of which the most popular are grouping and sorting) can be applied to client views (because **ClientView** is derived from View). For example:

```
View productsByCategory = products
    .OrderBy(p => p.ProductName).GroupBy(p => p.CategoryID);
```

or in LINQ query syntax:

```
View productsByCategory =
    from p in products
    orderby p.ProductName
    group p by p.CategoryID into g
    select new { Category = g.Key, Products = g };
```

The resulting views are live, but they are not client views. This is not a defect of some sort; it is simply because such views don't need ClientView functionality. Sorting and grouping can be performed entirely on the client without resorting to the server. However, developers must be aware of this fact to avoid confusion. Once you applied a LINQ query operation to your view, it becomes a View, not a **ClientView** (however, it remains live, automatically reflects all changes you make to the data on the client). So, for example, if you need server-side filtering, use the **AsFilter** method, not the LINQ *Where* method. Use the LINQ *Where* method when you want filtering on the client.

## Client-Side Transactions

**DataSource for Entity Framework** gives developers a powerful mechanism for canceling changes on the client without involving the server. It is called *transaction* because it is similar to the common concept of database transaction in that it allows you to ensure that a certain group of changes (unit of work) is either made in its entirety or canceled in its entirety-that your code never makes incomplete or inconsistent changes to entities in memory. It is important to understand that these transactions have no affect  in any way and are completely independent from database transactions. To distinguish them from database transactions, we sometimes call them *client-side transactions*.

Client-side transactions are especially useful in implementing **Cancel/Undo** buttons/commands. Doing this without C1DataSource requires cancelling all changes in the object context. C1DataSource client-side transactions make partial canceling of changes possible. And the transactions can even be nested, so you can have, for example, a dialog box with a **Cancel** button (which cancels only the changes made in that dialog box, not all changes in the object context made elsewhere in the application), and from that dialog box you can open another dialog box with its own **Cancel** button. Using a nested (child) transaction in the child dialog ensures that its **Cancel** button cancels (rolls back) only the changes made in the child dialog box, so the user can return to editing data in the parent dialog box and then accept or cancel changes in it using the parent transaction.

The easiest way to work with client-side transactions is by associating them with live views. For example, if we have a data grid bound to a live view

```
var ordersView = from o in customer.Orders.AsLive()
                 select new OrderInfo
                 {
                     OrderID = o.OrderID,
                     OrderDate = o.OrderDate,
                 };
dataGrid1.ItemsSource = ordersView;
```

We can create a transaction and associate it with the view like this:

```
var transaction = _scope.ClientCache.CreateTransaction();
   ordersView.SetTransaction(transaction);
```

To create a child (nested) transaction, instead of calling the method **ClientCacheBase,CreateTransaction**, use the **ClientTransaction** constructor by passing it the parent transaction as a parameter:

```
var transaction = new ClientTransaction(parentTransaction);
```

Once a transaction is associated with a view by calling **View.SetTransaction**; it tracks all changes made through that view, via data binding (for example, changes made by the end user in the grid bound to the view as in the example

above) as well as programmatically in code. Rolling back the transaction (calling **ClientTransaction.Rollback**) cancels the changes tracked by that transaction.

It is often necessary to bind GUI controls to a single object (as opposed to binding to a collection of objects). A single object can't be represented by a live view, so we don't have the convenience of View.SetTransaction|keyword=SetTransaction method, but in this case we can use the **ClientTransaction.ScopeDataContext** method. In WPF and Silverlight, you can use this method to set the **DataContext** so it will be used for data bindings specified in XAML:

```
DataContext = transaction.ScopeDataContext(order);
```

The resulting **DataContext** wraps the original one and performs the same data binding but with the additional benefit of all changes made through that data binding being tracked by the 'transaction', so they can be rolled back if necessary.

In WinForms, you can use the same **ScopeDataContext** and bind to the resulting object, for example, like this:

```
var dataContext = transaction.ScopeDataContext(order);
    textBox.DataBindings.Add(new Binding("Text", dataContext, "OrderDate"));
```

Finally, sometimes you need to change (or add or delete) some entities in code, not through a live view or data binding, and want those changes to be tracked by a transaction. You can do it using the **ClientTransaction.Scope()** method. That method opens a *scope* for a transaction. When you modify entities while in the scope of a transaction, those changes are tracked by that transaction. That method is designed to be used with the 'using' construct that conveniently closes the scope on exit, like this:

```
using (transaction.Scope())
{
    Customer.Orders.Add(order);
}
```

All three methods of using transactions described above are demonstrated in the **Transactions** sample project that comes with C1DataSource. It also demonstrates how a form with a **Cancel** button can be implemented inside another form that also has a **Cancel** button using child (nested) transactions.

## Virtual Mode

Virtual mode allows **DataSource for Entity Framework** to provide data binding of GUI controls to very large data sets with no delays, no code, and without resorting to paging. This is done by simply setting the **ClientViewSource.VirtualMode** property. It has three possible values:

- **None:** Virtual mode is not enabled. This is the default value.
- **Managed:** Virtual mode (fetching data from the server) is driven by a grid control. The grid control driving it is specified by setting an extender (attached) property on the grid. Most popular grids are supported, but not all grids (see the list below). If you need virtual mode with one of the supported grids, use the **Managed** option, do not use **Unmanaged**. Performance is optimal with the **Managed** option because it is specifically optimized for the grid in question. In addition to the grid control there can be other simple controls bound to the same data source (for example, TextBox controls), but no "complex" bound control (for example, another grid or list) in addition to the driving grid should be used.

  **Managed** mode is currently supported for the following grid controls:

  - **ComponentOne:** C1FlexGrid (WinForms, WPF, Silverlight), C1DataGrid (WPF, Silverlight).
  - **Microsoft:** DataGridView (WinForms), DataGrid (WPF), DataGrid (Silverlight; but see below about a problem with Microsoft DataGrid for Silverlight).

- **Unmanaged:** Virtual mode (fetching data from the server) is driven by the data source itself, regardless of

what type of controls are bound to it. Use this option if you need virtual mode with a grid/list/etc. control that does not belong to the list of supported grids below (but note that "simple" controls such as TextBox that are only bound to the current record and not to the entire data set can be used without limitations with either **Managed** or **Unmanaged** mode). Performance in **Unmanaged** is almost as good as in **Managed**, but it is subject to some limitations. See Unmanaged virtual mode limitations for more information. Those limitations are not very restricting, but they cannot be checked automatically, so, before deciding to use the **Unmanaged** option, make sure you do not break the limitation rules. Also, in **Unmanaged** mode, the **PageSize** property must be set to a number greater than your GUI control will show at any given time (PageSize is ignored in Managed mode).

## Unmanaged Virtual Mode Limitations

The difference between **Managed** and **Unmanaged** options is seen when data is retrieved from the server. In **Managed** mode, data is retrieved when the "driving" grid needs it, for example, when the user scrolls or navigates in the grid. In **Unmanaged** mode data is fetched from the server (if it is not already present in the cache) on every data request a bound control makes, without taking into consideration which control makes the request and for what purpose. The data source fetches ClientViewSource.PageSize rows around every row that is requested from it. But it does not keep/hold those rows because (unlike in **Managed** mode) it does not know what rows are visible in bound controls. It holds only the current row and the last requested row (both with ClientViewSource.PageSize rows before and ClientViewSource.PageSize rows after them). All other rows are not held, that is, they can be released, evicted from cache, when **Entity Framework DataSource** needs to cleanup/compact the cache. In most common scenarios this cannot happen, and in general you can prevent it from happening by making sure that you

**Never bind more than one scrollable control to a data source in Unmanaged virtual mode.** You can bind any number of simple (bound to a single row) controls like TextBox. You can bind any grid or list or other "scrollable" control (bound to the entire data set as opposed to a single row) as well. But you should never bind two scrollable controls that can be scrolled to two different locations (rows) in the data set that are more than ClientViewSource.PageSize apart from one another, because only one (the latest requested) of these rows will be held, the other can be released by **Entity Framework DataSource** (at unpredictable time).

## Other Limitations of Virtual Mode

- Binding two independently scrollable controls (such as grids or lists that can be scrolled to different, independent locations in the data set) to a single data source in virtual mode is not supported in **Managed** mode as well as in **Unmanaged**, and for the same reason. The difference is just that in **Managed** mode it is less dangerous because you will immediately see the effect; there is no element of unpredictability in the behavior. If in addition to the main "driving" bound control there is another scrollable control bound to the same data source, that control is not "driving", that is, scrolling in it does not fetch data; therefore, if you scroll beyond the rows that are visible in the main control, you will see empty rows.
- In both **Managed** and **Unmanaged** modes, grids, lists, and other controls bound to the entire data set (as opposed to a single record), must not do anything with all rows of the data source at once. In other words, a control should not perform actions in its code that involve going in a loop over all rows of the data source and doing something for each row. That is for the simple reason that the number of rows in virtual mode can be very large, many thousands or even millions of rows. It is, in fact, unlimited. Properly designed controls that do not rely on an assumption that the number of rows in its data source is small, will do fine in **DataSource for Entity Framework** virtual mode. But those that were designed specifically for small data sources, those that loop through all their rows, can cause long delays with very large number of rows. C1FlexGrid and C1DataGrid (for WinForms/WPF/Silverlight) are OK, as are Microsoft DataGridView (WinForms) and Microsoft DataGrid for WPF. But Microsoft DataGrid for Silverlight (in its current version, Silverlight 4) is not recommended for C1DataSource virtual mode because it loops through all rows to compute its height.

# ComponentOne LiveLinq

Make LINQ faster and get live views with **ComponentOne LiveLinq**. This unique class library augments the functionality of LINQ using indexing and other optimizations to speed up LINQ queries up to 100 times faster or more. And with live view, your LINQ query result is kept up-to-date without re-populating it every time its base data changes.

# LiveLinq Overview

LiveLinq is a class library that augments the functionality of LINQ in two related directions:

- **It makes LINQ faster**

  LiveLinq uses indexing and other optimizations to speed up LINQ queries in memory. Speed gains can be hundreds and even thousands of times on queries with high selectivity conditions. Typical/average gains are lower but still significant; a 10-50 times speedup can be considered typical.

- **It adds support for live views to LINQ**

  A *live view* is a LINQ query result that is kept constantly up-to-date without re-populating it every time its base data changes. This makes LiveLinq extremely useful in common data-binding scenarios where objects are edited and may be filtered in or out of views, have their associated subtotals updated, and so on. In old jargon, one could say that LINQ queries correspond to snapshots, while LiveLinq views correspond to dynasets. Since live views automatically react to changes, they greatly widen the sphere of declarative programming, not only in data binding and GUI but in many other programming scenarios as well.

## The scope of LiveLinq

LiveLinq implements three LINQ varieties:

- LINQ to Objects
- LINQ to XML
- LINQ to DataSet

In other words, its scope is LINQ in memory. The current LiveLinq version does not offer an alternative to LINQ to databases. However, this does not prevent you from using LiveLinq with data retrieved to memory from a database. For example, you can retrieve data from a database to a dataset in memory using ADO.NET and then use LiveLinq to DataSet. You can also retrieve objects from a database using LINQ to SQL or Entity Framework and then operate on that data using LiveLinq and send changes to the database again using Entity Framework or another framework of your choice.

## LiveLinq and LINQ

LiveLinq has the same syntax as the standard LINQ. The only change necessary is to wrap your data source by applying to it an extension method **AsIndexed()** (for indexing) or **AsLive()** (for live views).

**How the two parts of LiveLinq are related to each other**

The two areas of LiveLinq functionality, indexing and live views can interoperate but are independent. It is not necessary to define indexes if you need to create a live view. A view is populated initially by executing a query, so, if the query can be optimized using indexing, the view will be populated faster. But after its initial population, changes to the view are made using special optimization techniques (known as Incremental View Maintenance) that don't require the user to explicitly define indexes.

## Faster LINQ with Indexing

LiveLinq contains an indexing framework that it uses for optimizing query performance (not available in Silverlight). For example, by defining an index by ProductID, we can dramatically speed up queries like

```
from p in products where p.ProductID == 100
```

because it will use the index to access the requested product directly instead of going through the entire large collection in search of a single product.

Same indexes can also be used programmatically, in code, even without LINQ, for various kinds of searches, including range search and others.

## Declarative programming with Live Views

**Live (real) data binding**

LiveLinq adds the concept of ***view*** to LINQ. A view is a query with a resultset that remains live, dynamic after it is initially populated by executing the query. A standard LINQ query is a snapshot in the sense that its result list does not change when you change the underlying (base) data. So it can't be used for full-featured data binding. A live view is automatically kept in sync with base data, so it enables full data binding to LINQ queries.

Moreover, many views are updatable themselves; you can modify properties in their objects and add and delete objects directly in the view, see Updatable Views. So, a view can be modifiable in both directions: changes in base data propagate to the view and changes in the view propagate to base data (the latter pursuant to usual conditions of updatability).

This is full-featured two-way data binding.

**Reactive = "View-Oriented" = Declarative Programming**

Data binding, which is mostly used in GUI, is a very important part of live views functionality, but not the only one. More generally, live views enable a declarative style of programming which we tentatively call "view-oriented programming". Non-GUI, batch processing code can also be made declarative using live views.

**Enabling technology: Incrementality**

When a change occurs in base data, live views update themselves in a smart, fast way, not simply repopulating themselves from scratch. The changes are made locally, ***incrementally***, calculating the delta in the view from the delta in the base data. In most cases, it allows you to propagate the change from base data to the view very fast. This is a key component of LiveLinq: its enabling technology, based on an area of Computer Science known as Incremental View Maintenance. But the user does not need to do anything to enable this optimization, as it is done entirely beneath the hood, transparent to the user.

# What is LINQ?

LINQ, or Language Integrated Query, is a set of features in .NET 3.5 used for writing structured type-safe queries over local object collections and remote data sources.

LINQ enables you to query any collection implementing the **IEnumerable** interface, including arrays, lists, XML documents, as well as remote data sources such as tables in SQL Server.

LINQ offers the following important benefits:

- Compile-time type-checking
- Language integration (including IntelliSense support)
- Uniformity across different data sources

- Flexible, powerful, and expressive queries

In order to use **LiveLinq** effectively, you must be reasonably proficient in LINQ. A deep coverage of LINQ is beyond the scope of this document, but there are several excellent resources available on the subject. We recommend the following:

- "C# 3.0 in a nutshell", by Joseph and Ben Albahari. O'Reilly, 2007.
- "LINQ in Action", by Fabrice Marguerie, Steve Eichert and Jim Wooley. Manning, 2008.
- "Programming Microsoft LINQ Developer Reference", by Paolo Pialorsi and Marco Russo. Microsoft Press, 2008.

# What is LiveLinq?

**LiveLinq** is a set of extensions to LINQ that add two important capabilities to standard LINQ queries:

1. **LiveLinq optimizes LINQ queries**
   **LiveLinq** uses indexing and other optimizations to speed up LINQ queries. Speed gains of 10 to 50 times are typical for non-trivial queries. The overall performance gains can be dramatic for data-centric applications that rely heavily on LINQ.
2. **LiveLinq turns LINQ query results into Live Views**
   Live views are query results that are kept up-to-date with respect to the base data. Live views are essential to data binding scenarios where objects may be added, removed, or changed while bound to controls in the UI. Using old jargon, one could say that plain LINQ queries correspond to *snapshots*, while **LiveLinq** views correspond to *dynasets*.

# How does LiveLinq work?

**LiveLinq** implements "Incremental View Maintenance" techniques. Unlike standard LINQ, **LiveLinq** does not discard all query information after executing. Instead, it keeps the data it processes in indexed lists which are incrementally updated and synchronized as the underlying data changes.

The overhead involved is relatively small, since the data is already in memory to begin with (**LiveLinq** only operates with in-memory data). The benefits are huge: **LiveLinq** not only accelerates typical queries by orders of magnitude, but also enables the use of LINQ in data binding scenarios that would not be possible with standard LINQ.

# Getting Started with LiveLinq

We will show several samples that illustrate **LiveLinq**. All the samples presented here use the **Northwind** database. You can use the Access version (**C1NWind.mdb**) or the SQL Server version (**NORTHWND.MDF**). If you don't have Northwind and would like to try the samples, you can download the SQL Server version of the database from this URL:

http://www.microsoft.com/downloads/details.aspx?FamilyID=06616212-0356-46A0-8DA2-EEBC53A68034

The **LiveLinq** distribution package includes more samples and demos that show details not covered in the "Getting Started" part of the documentation.

# Optimizing LINQ Queries with LiveLinq

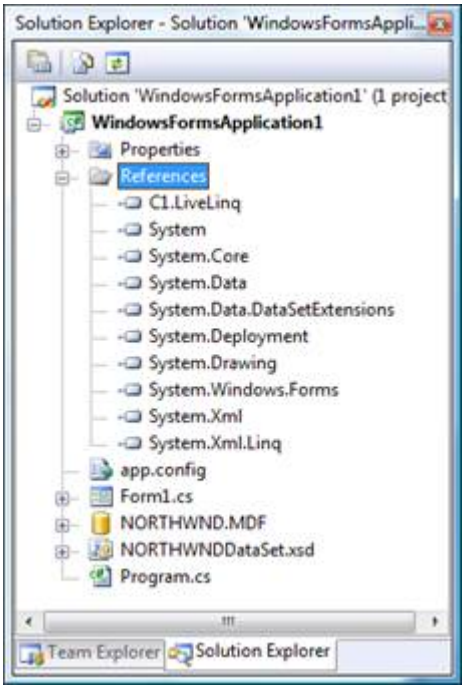
In this section, we will show a sample that illustrates how **LiveLinq** can optimize queries that use the **where** operator to filter data.

To start, follow these steps:

1. Create a new **WinForms** project

2. Add a reference to the **C1.LiveLinq.dll** assembly
3. Use the **Data | Add New DataSource** menu and add a reference to the **NORTHWND.MDF** database. Accept all the default options offered by the wizard, and pick all the tables in the database.

At this point, your project should look like this




Next, double-click the form and add the following code:

```
// declare northwind DataSet
NORTHWNDDataSet _ds = new NORTHWNDDataSet();
private void Form1_Load(object sender, EventArgs e)
{
  // load data into DataSet
  new NORTHWNDDataSetTableAdapters.CustomersTableAdapter()
    .Fill(_ds.Customers);
  new NORTHWNDDataSetTableAdapters.OrdersTableAdapter()
    .Fill(_ds.Orders);
}
```

This code declares an ADO.NET **DataSet** and loads some data into it.

Now that we have some data, let's do something with it.

Add a button to the form, double-click it, and then enter the following code:

```
private void button1_Click(object sender, EventArgs e)
{
  // get reference to source data
  var customers = _ds.Customers;
  var orders = _ds.Orders;
  // find all orders for the first customer
  var q =
    from o in orders
    where o.CustomerID == customers[0].CustomerID
```

```
    select o;
  // benchmark the query (execute 1000 times)
  var start = DateTime.Now;
  int count = 0;
  for(int i = 0; i < 1000; i++)
  {
    foreach (var d in q)
      count++;
  }
  Console.WriteLine("LINQ query done in {0} ms",
     DateTime.Now.Subtract(start).TotalMilliseconds);
}
```

The code creates a simple LINQ query that enumerates all orders for the first customer in the database, then executes the query 1000 times and reports how long the process took.

If you run the project now and click the button, the Visual Studio output window should show something like this:

**LINK query done in 262 ms**

Now let us optimize this query with **LiveLinq**.

Start by adding the following **using** statements to the top of the code:

```
using C1.LiveLinq;
using C1.LiveLinq.AdoNet;
```

Next, add a second button to the form, double-click it and enter the following code:

```
private void button2_Click(object sender, EventArgs e)
{
  // get reference to source data
  var customers = _ds.Customers;
  var orders = _ds.Orders;
  // find all orders for the first customer
  var q =
    from o in orders.AsIndexed()
    where o.CustomerID.Indexed() == customers[0].CustomerID
    select o;
  // benchmark the query (execute 1000 times)
  var start = DateTime.Now;
  int count = 0;
  for(int i = 0; i < 1000; i++)
  {
    foreach (var d in q)
    count++;
  }
  Console.WriteLine("LiveLinq query done in {0} ms",
     DateTime.Now.Subtract(start).TotalMilliseconds);
}
```

The code is almost identical to the plain LINQ version we used before. The only differences are:

- We use the **AsIndexed** method to convert the **orders** table into a **LiveLinq** indexed data source,
- We use the **Indexed** method to indicate to **LiveLinq** that it should index the data source by customer id,

AND

- The output message changed to show we are using **LiveLinq** now.

If you run the project again and click both buttons a few times, you should get something like this:

**LINQ query done in 265 ms**

**LINQ query done in 305 ms**

**LINQ query done in 278 ms**

**LiveLinq query done in 124 ms**

**LiveLinq query done in 7 ms**

**LiveLinq query done in 2 ms**

Notice how the plain LINQ query takes roughly the same amount of time whenever it executes. The **LiveLinq** query, on the other hand, is about twice as fast the first time it executes, and about **one hundred** times faster for all subsequent runs. This level of performance gain is typical for this type of query.

This happens because **LiveLinq** has to build the index when the query is executed for the first time. From then on, the same index is reused and performance improves dramatically. Note that the index is automatically maintained, so it is always up-to-date even when the underlying data changes.

In this example, the index was created when the **Indexed** method executed for the first time. You can also manage indexes using code if you need the extra control. For example, the code below declares an indexed collection and explicitly adds an index on the **CustomerID** field:

```
var ordersIndexed = _ds.Orders.AsIndexed();
ordersIndexed.Indexes.Add(o => o.CustomerID);
```

Note that the indexes are associated with the source data, and are kept even if the collection goes out of scope. If you executed the code above once, the index would remain active even after the **ordersIndexed** collection went out of scope.

Note also that the **AsIndexed** method is supported only for collections that provide change notifications (**LiveLinq** has to monitor changes in order to maintain its indexes). This requirement is satisfied for all common collections used in WinForms and WPF data binding. In particular, **AsIndexed** can be used with **DataTable**, **DataView**, **BindingList<T>**, **ObservableCollection<T>**, and LINQ to XML collections.

The **LiveLinq** installation includes a sample called **LiveLinqQueries** that contains many other examples with benchmarks, and includes queries against plain CLR objects, ADO.NET, and XML data.

Summarizing, **LiveLinq** can significantly optimize queries that search for any type of data that can be sorted. For example, searching for a specific customer or product, or listing products within a certain price range. These are typical queries that can be easily optimized to perform about a hundred times faster that they would using plain LINQ.

## Basic LiveLinq Binding

Besides accelerating typical queries, **LiveLinq** also enables the use of LINQ in data binding scenarios.

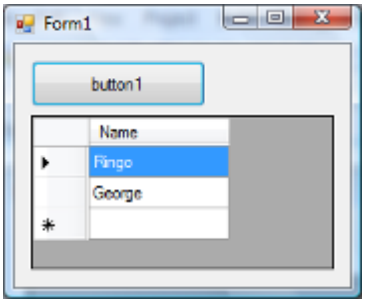
To illustrate this, let us start with a simple example.

1. Create a new **WinForms** project
2. Add a reference to the **C1.LiveLinq.dll** assembly
3. Add a button and a **DataGridView** to the main form
4. Double-click the button and add this code to the form:

C#

```
using C1.LiveLinq;
using C1.LiveLinq.AdoNet;
using C1.LiveLinq.LiveViews;
using C1.LiveLinq.Collections;

private void button1_Click(object sender, EventArgs e)
{
  // create data source
  var contacts = new IndexedCollection<Contact>();

  // bind list to grid (before adding elements)
  this.dataGridView1.DataSource =
    from c in contacts.AsLive()
    where c.Name.Contains("g")
    select c;

  // add elements to collection (after binding)
  var names = "Paul,Ringo,John,George,Robert,Jimmy,John Paul,Bonzo";
  foreach (string s in names.Split(','))
  {
    contacts.Add(new Contact() { Name = s });
  }
}

public class Contact : IndexableObject
{
  private string _name;
  public string Name
  {
    get { return _name; }
    set
    {
      OnPropertyChanging("Name");
      _name = value;
      OnPropertyChanged("Name");
    }
}
```

If you run the project and click the button, you should see two names on the grid: “Ringo” and “George”:




This may not seem surprising, since the data source is a query that returns all names that contain the letter “g”.

The interesting part is that all the contacts were added to the list *after* the query was assigned to the grid’s

DataSource property. This shows that the **LiveLinq** query actually returned a live list, one that (a) notified the grid when elements were added to it, and (b) honored the **where** operator by showing only the names that contain "g".

The differences between this **LiveLinq** query and a regular one are:

1. The object collection is of type **IndexedCollection<T>**. This is a class provided by **LiveLinq** that supports the required change notifications.
2. The **Contact** class is derived from a **LiveLinq** class **IndexedObject** so it can provide change notifications when its properties are set.
3. The query itself contains an **AsLive** call that tells **LiveLinq** we want the result to remain active and issue change notifications that will be handled by bound controls.

You can test that the filter condition remains active by editing the grid and changing "Ringo" into "Ricky". The row will be filtered out of the view as soon as you finish editing.

You can also check the effect of the **AsLive** call by commenting it out and running the sample again. This time, the grid will not receive any notifications as items are added to the list, and will remain empty.

## Hierarchical LiveLinq Binding

The previous example showed how **LiveLinq** provides basic data binding against plain CLR objects.

In this section, we will show how **LiveLinq** supports more advanced scenarios including master-detail data binding and currency management (data cursors). We will use an ADO.NET data source, but the principles are the same for all other in-memory data sources, including plain CLR objects and XML data.

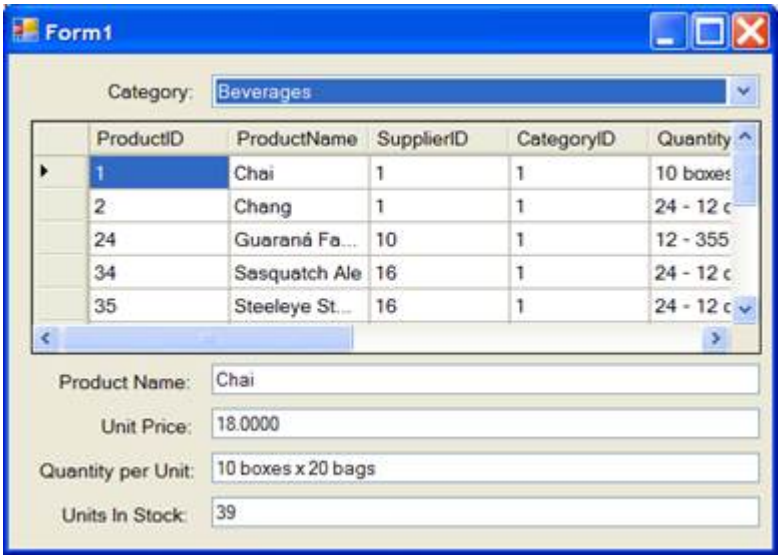
We will start with a simple application using traditional data binding. We will then show how you can easily convert that application to take advantage of **LiveLinq**. Finally, we will create WinForms and WPF versions of the application using **LiveLinq** from the start.

## Traditional WinForms Implementation

Our sample application will consist of the following elements:

- A **ComboBox** listing the NorthWind product categories.
- A **DataGridView** showing all products for the currently selected category.
- Some **TextBox** controls bound to properties of the currently selected product.

This is what the final application will look like:




The basic implementation is simple, since Visual Studio handles most of the data binding related tasks automatically. In fact, the entire application can be written without a single line of code.

Here are the steps:

1. Create a new WinForms application
2. Use the **Data | Add New DataSource** menu and add a reference to the **NORTHWND.MDF** database. Accept all the default options offered by the wizard, and pick all the tables in the database.
3. Add the controls to the form as shown on the image above: one **ComboBox**, one **DataGridView**, four **TextBox** controls, and a few **Label** controls.

At this point, the application has access to the data and it has controls to show and edit the data. To connect the controls to the data, follow these steps:

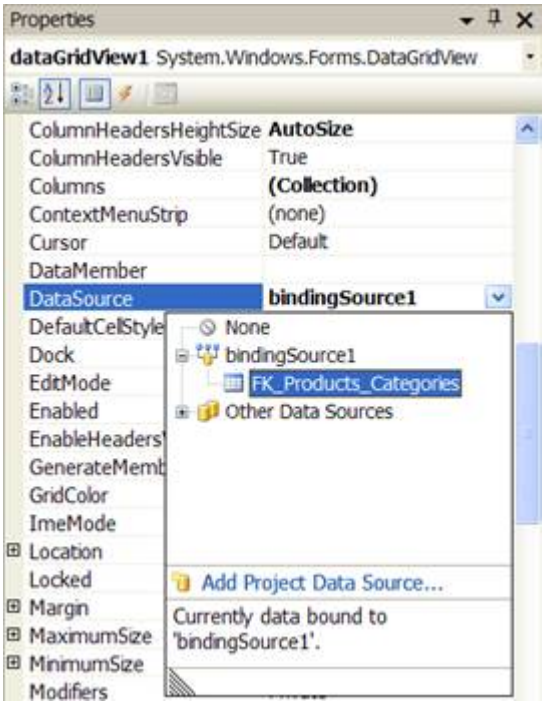
1. Add a new **BindingSource** component to the form
2. Select the new **BindingSource** component, select its DataSource property in the property window, and use the drop-down editor to select the **NORTHWINDDataSet** data set.
3. Still with the **BindingSource** component selected, use the drop-down editor set the **DataMember** property to "Categories".

The final step is to select each data bound control and bind it to the **BindingSource** component:

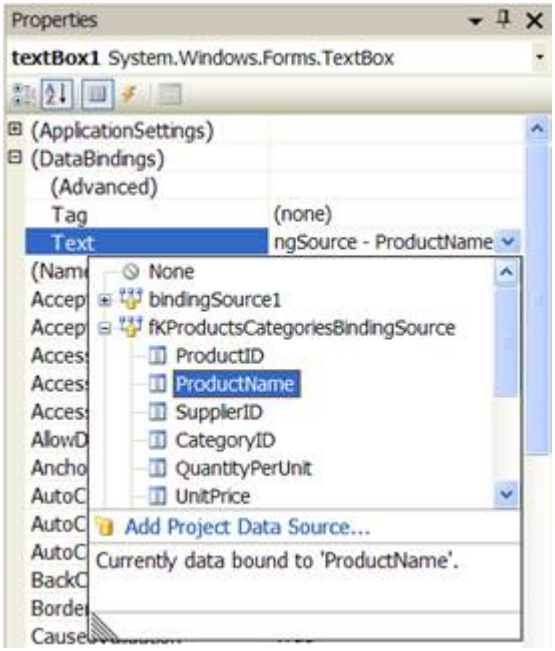
For the **ComboBox**, set the following properties:

C1DataSource bindingSource1
**DisplayMember** CategoryName
**ValueMember** CategoryID

For the **DataGridView**, use the drop-down editor in the property grid to set the DataSource property to **FK_Products_Categories**, the item that appears under bindingSource1 and represents the products under the currently selected category. The image below shows what the drop-down editor looks like just before the selection is made:




Finally, select each of the TextBox controls and use the drop-down editor in the property window to bind the Text property to the corresponding element in the currently selected product. For example:




Repeat this step to bind the other **TextBox** controls to the **UnitPrice**, **QuantityPerUnit**, and **UnitsInStock** fields.

The application is now ready. Run it and notice the following:

- When you select a category from the **ComboBox**, the corresponding products are displayed on the grid below, and the product details appear in the **TextBox** controls.
- When you select a product on the grid, the product details are automatically updated.
- The values shown on the grid and in the text boxes are synchronized. If you change the values in one place, they also change in the other.
- If you change the value of a product's **CategoryID** in the grid, the product no longer belongs to the currently

selected category and is automatically removed from the grid.

This is the traditional way of doing data binding in **WinForms**. Visual Studio provides rich design-time support and tools that make it easy to get applications started. Of course, real applications typically require you to add some code to implement specific logic.

## Migrating to LiveLinq

The application we just described relies on a **bindingSource** object that exposes an ADO.NET **DataTable** as a data source. Migrating this application to **LiveLinq** is very easy. All you need to do is have the **bindingSource** object expose a **LiveLinq** view instead of a regular **DataTable**.

Here are the steps required:

1. Add a reference to the **C1.LiveLinq.dll** assembly to the project.
2. Add a few **using** statements to make the code more readable:

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using C1.LiveLinq;
using C1.LiveLinq.AdoNet;
using C1.LiveLinq.LiveViews;
```

3. Create a **LiveLinq** query and assign it to the DataSource property of the **bindingSource** object, replacing the original reference to a **DataTable** object:

C#

```
private void Form1_Load(object sender, EventArgs e)
{
  // generated automatically
  this.productsTableAdapter.Fill(this.nORTHWNDDataSet.Products);
  this.categoriesTableAdapter.Fill(this.nORTHWNDDataSet.Categories);

  // Create a live view for Categories.
  // Each category contains a list with the products of that category.
  var categoryView =
    from c in nORTHWNDDataSet.Categories.AsLive()
    join p in nORTHWNDDataSet.Products.AsLive()
      on c.CategoryID equals p.CategoryID into g
    select new
    {
```

```
      c.CategoryID,
      c.CategoryName,
      FK_Products_Categories = g
    };

  // replace DataSource on the form to use our LiveLinq Query
  this.bindingSource1.DataSource = categoryView;
}
```

The code starts by creating the **LiveLinq** query that will serve as a data source for all controls on the form.

The query is 100% standard LINQ, except for the **AsLive** statements which turn the standard LINQ query into a live view suitable for binding. Without them, the code would not even compile.

The query uses a **join** to obtain all products for each category and store them in a group, and then selects the category ID, category name, and the group of products associated with the category.

The group of products is named **FK_Products_Categories**. We did not name it something simple and intuitive like "Products" because the binding code created behind the scenes by Visual Studio relies on this specific name.

If you look at the **Form1.Designer.cs** file, you will notice that Visual Studio created a **bindingSource** object named **fKProductsCategoriesBindingSource** which is initialized as follows:

C#

```
//
// fKProductsCategoriesBindingSource
//
this.fKProductsCategoriesBindingSource.DataMember = "FK_Products_Categories";
this.fKProductsCategoriesBindingSource.DataSource = this.bindingSource1;
```

This code assumes that the original binding source contains a property named "FK_Products_Categories" that exposes the list of products for the current category. To ensure that the bindings still work, our query needs to use the same name.

If you run the project now, you will see that it works exactly as it did before. But now it is fully driven by a LINQ query, which means we have gained a lot of flexibility. It would be easy to rewrite the LINQ statement and display additional information such as supplier names, total sales, etc.

## LiveLinq implementation in WinForms

The previous section described how to migrate a traditional data-bound application to use **LiveLinq**. The resulting application used **BindingSource** objects and code that was generated by Visual Studio at design time.

If we wanted to create a new **LiveLinq** application from scratch, we could make it even simpler.

Here are the steps:

1. Create a new WinForms application
2. Use the **Data | Add New DataSource** menu and add a reference to the **NORTHWND.MDF** database. Accept all the default options offered by the wizard, and pick all the tables in the database.
3. Add the controls to the form as before: one **ComboBox**, one **DataGridView**, four **TextBox** controls, and a few **Label** controls.
4. Add a reference to the **C1.LiveLinq.dll** assembly to the project.
5. Double-click the form add the following code:

C#

```
using C1.LiveLinq;
using C1.LiveLinq.AdoNet;
using C1.LiveLinq.LiveViews;

private void Form1_Load(object sender, EventArgs e)
{
  // get the data
  var ds = GetData();

  // create a live view with Categories and Products:
  var liveView =
    from c in ds.Categories.AsLive()
    join p in ds.Products.AsLive()
      on c.CategoryID equals p.CategoryID into g
    select new
    {
      c.CategoryID,
      c.CategoryName,
      Products = g
    };

  // bind view to controls
  DataBind(liveView);
}
```

The code is straightforward. It calls a **GetData** method to load the data into a **DataSet**, creates a **LiveLinq** view using a LINQ statement similar to the one we used earlier, and then calls **DataBind** to bind the controls on the form to the view.

Here is the implementation of the **GetData** method:

C#

```
NORTHWNDDataSet GetData()
{
  NORTHWNDDataSet ds = new NORTHWNDDataSet();
  new NORTHWNDDataSetTableAdapters.ProductsTableAdapter()
    .Fill(ds.Products);
  new NORTHWNDDataSetTableAdapters.CategoriesTableAdapter()
    .Fill(ds.Categories);
  return ds;
}
```

**GetData** creates a **NORTHWNDDataSet**, fills the “Products” and “Categories” tables using the adapters created by Visual Studio at design time, and returns the data set.

Here is the implementation of the **DataBind** method:

C#

```
void DataBind(object dataSource)
{
  // bind ComboBox
```

```
  comboBox1.DataSource = dataSource;
  comboBox1.DisplayMember = "CategoryName";
  comboBox1.ValueMember = "CategoryID";

  // bind DataGridView
  dataGridView1.DataMember = "Products";
  dataGridView1.DataSource = dataSource;

  // bind TextBox controls
  BindTextBox(textBox1, dataSource, "ProductName");
  BindTextBox(textBox2, dataSource, "UnitPrice");
  BindTextBox(textBox3, dataSource, "QuantityPerUnit");
  BindTextBox(textBox4, dataSource, "UnitsInStock");
}

void BindTextBox(TextBox txt, object dataSource, string dataMember)
{
    var b = new Binding("Text", dataSource, "Products." + dataMember);
    txt.DataBindings.Add(b);
}
```

**DataBind** sets the data binding properties on each control to bind them to our **LiveLinq** view. This is exactly the same thing we did before using the property window editors, except this time we are doing it all in code and binding the controls directly to the **LiveLinq** view instead of going through a **BindingSource** component.

If you run the project now, you will see that it still works exactly as before.

Writing the data binding code usually takes a little longer than using the design time editors and letting Visual Studio write the code for you. On the other hand, the result is usually code that is simpler, easier to maintain, and easier to port to other platforms (as we will do in the next section).

Either way, you can use **LiveLinq** views as binding sources for the controls on the form.

## LiveLinq and Declarative Programming

The live views provided by **LiveLinq** are not restricted to data binding scenarios.

Live views can be used to combine data from multiple tables, group and aggregate this data according to business rules, and make it available to the application at all times. The views are always synchronized with the data, so there's no need to call methods to update the views when you need the data. This greatly simplifies application logic and improves efficiency.

To illustrate this point, imagine a NorthWind application that exposes the following services:

**ProcessOrder**: This service bills the customers, ships the products, and updates the information in the database. It is used by the sales force and by the company web store.

**SalesInformation**: This service returns summaries of sales per product and product category. It is used by management and marketing.

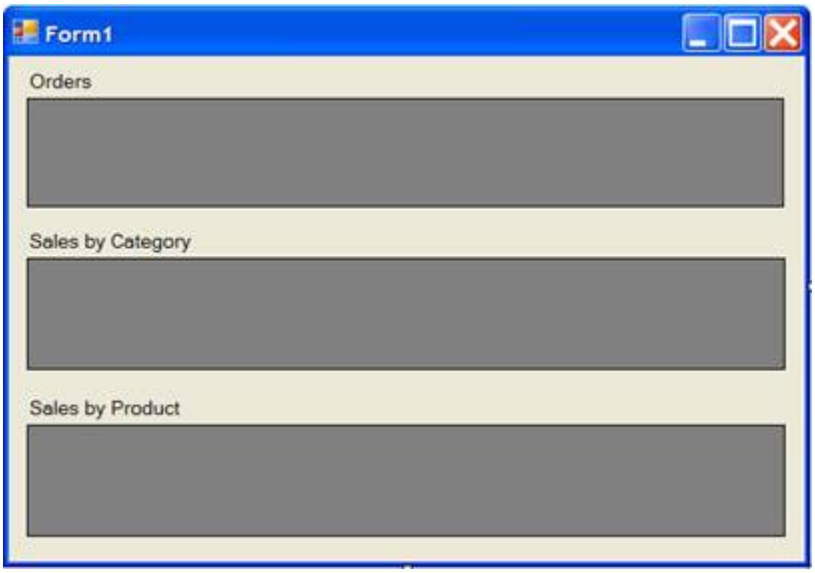
You could implement the application by having the **ProcessOrder** service write orders directly into the database and having the **SalesInformation** service run a stored procedure that would return the latest sales summaries. This would work, but the **SalesInformation** service would be relatively expensive since it would go to the database every time and would have to scan all the orders in the database.

Another approach would be to load the data into live views, where the sales summaries would be kept constantly up to date as orders are processed. Calls to the **ProcessOrder** method would automatically update the summaries provided by **SalesInformation**. Calls to **SalesInformation** would be processed in zero time, without touching the database at all.

To illustrate this, let us create another simple WinForms application using the NorthWind data once again. The application will have three grids. The first corresponds to the **ProcessOrder** service, allowing users to edit orders. The others correspond to the **SalesInformation** service, showing sales summaries that are always synchronized with the orders.

Here are the steps:

1. Create a new WinForms application
2. Use the **Data | Add New DataSource** menu and add a reference to the **NORTHWND.MDF** database. Accept all the default options offered by the wizard, and pick all the tables in the database.
3. Add a reference to the **C1.LiveLinq.dll** assembly to the project.
4. Add three **DataGridView** controls to the form, with labels above each one as shown in the image below.




Now, right-click the form and enter the following code:

C#

```
using C1.LiveLinq;
using C1.LiveLinq.AdoNet;
using C1.LiveLinq.LiveViews;

public Form1()
{
  InitializeComponent();

  // get the data
  NORTHWNDDataSet ds = GetData();

  // create a live view to update order details
  this.dataGridView1.DataSource = GetOrderDetails(ds);
```

```
  // create live views that provide up-to-date order information
  this.dataGridView2.DataSource = GetSalesByCategory(ds);
  this.dataGridView3.DataSource = GetSalesByProduct(ds);
}
```

As before, the first step is loading the relevant data from the database:

C#

```
NORTHWNDDataSet GetData()
{
  var ds = new NORTHWNDDataSet();
  new NORTHWNDDataSetTableAdapters.ProductsTableAdapter()
    .Fill(ds.Products);
  new NORTHWNDDataSetTableAdapters.Order_DetailsTableAdapter()
    .Fill(ds.Order_Details);
  new NORTHWNDDataSetTableAdapters.CategoriesTableAdapter()
    .Fill(ds.Categories);
  return ds;
}
```

Next, we use **LiveLinq** to implement the live view that will be exposed through the **SalesInformation** service. This is a standard LINQ query, only slightly more sophisticated than the ones we used in earlier samples, with a couple of **AsLive** clauses that turn the standard query into a live view:

C#

```
object GetSalesByCategory(NORTHWNDDataSet ds)
{
  var products = ds.Products;
  var details = ds.Order_Details;
  var categories = ds.Categories;

  var salesByCategory =
    from p in products.AsLive()
    join c in categories.AsLive()
      on p.CategoryID equals c.CategoryID
    join d in details.AsLive()
      on p.ProductID equals d.ProductID

    let detail = new
    {
      CategoryName = c.CategoryName,
      SaleAmount = d.UnitPrice * d.Quantity
        * (decimal)(1f - d.Discount)
    }

    group detail
      by detail.CategoryName into categorySales

    let total = categorySales.Sum(x => x.SaleAmount)
    orderby total descending
```

```
    select new
    {
      CategoryName = categorySales.Key,
      TotalSales = total
    };

  return salesByCategory;
}
```

The query starts by joining the three tables that contain the information on products, categories, and orders. It then creates a temporary **detail** variable that holds the product category and total amount for each order detail. Finally, the details are ordered by sales totals and grouped by category name.

The result is a live view that is automatically updated when the underlying data changes. You will see this a little later, when we run the app.

The next live view is similar, except it provides sales information by product instead of by category.

C#

```
object GetSalesByProduct(NORTHWNDDataSet ds)
{
  var products = ds.Products;
  var details = ds.Order_Details;
  var categories = ds.Categories;

  var salesByProduct =
    from p in products.AsLive()
    join c in categories.AsLive()
      on p.CategoryID equals c.CategoryID
    join d in details.AsLive()
      on p.ProductID equals d.ProductID
    into sales
    let productSales = new
    {
      ProductName = p.ProductName,
      CategoryName = c.CategoryName,
      TotalSales = sales.Sum(
         d => d.UnitPrice * d.Quantity *
              (decimal)(1f - d.Discount))
    }
    orderby productSales.TotalSales descending
    select productSales;

  return salesByProduct;
}
```

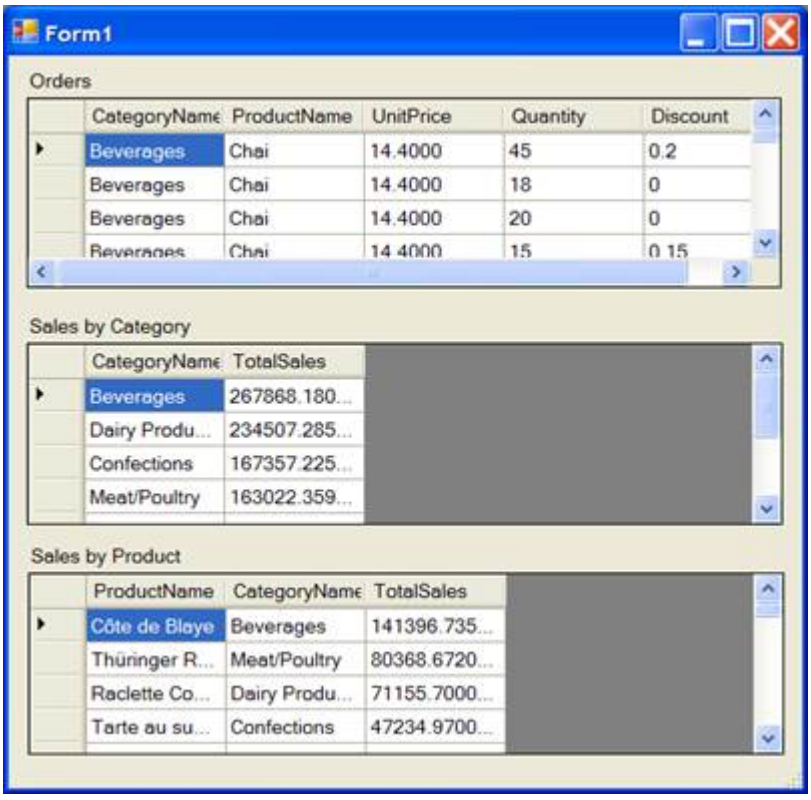
The last view shows the order details. We will bind this view to an editable grid so we can simulate orders being created or modified, and how the changes affect the previous views:

C#

```
object GetOrderDetails(NORTHWNDDataSet ds)
```

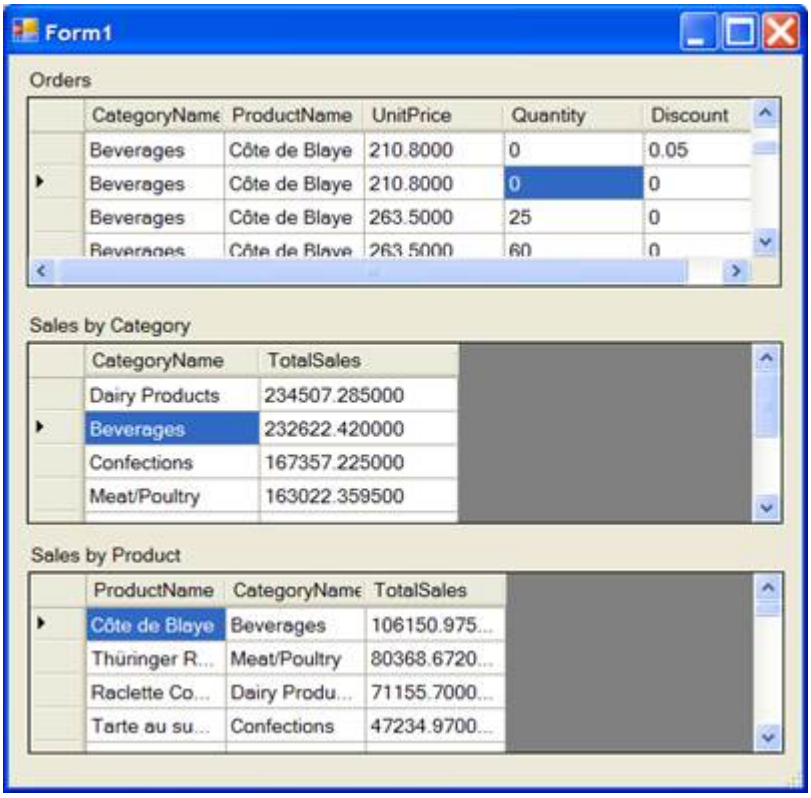
```
{
  var products = ds.Products;
  var details = ds.Order_Details;
  var categories = ds.Categories;

  var orderDetails =
    from d in details.AsLive().AsUpdatable()
    join p in products.AsLive()
      on d.ProductID equals p.ProductID
    join c in categories.AsLive()
      on p.CategoryID equals c.CategoryID
    select new
    {
      c.CategoryName,
      p.ProductName,
      d.UnitPrice,
      d.Quantity,
      d.Discount
    };

  return orderDetails;
}
```

If you run the application now, you should see a window like this one:




The application shows the total sales by category and by product, sorted by amount. The best-selling category is “Beverages”, and the best-selling product is “Cote de Blaye”.

Now click the “ProductName” column header on the top grid and scroll down to find the entries for “Cote de Blaye”. Once you’ve found them, try making a few changes to the orders and see how the summary data is immediately updated. For example, if you change the quantities to zero for a few “Cote de Blaye” orders, you will see that “Beverages” quickly falls behind the “Diary Products” category:




This simple example illustrates the power of LINQ-based live views. They bridge the gap between data and logic and can make data-centric applications much simpler and more efficient.

## How to Use LiveLinq

The following sections explain how to use **LiveLinq**.

## Query Collection

To use LiveLinq in Visual Studio, start by adding the LiveLinq assembly, C1.LiveLinq.dll, to your project's References. Then add the 'using' directives to the source file that will enable you to use LiveLinq in code:

C#

```
using C1.LiveLinq;
using C1.LiveLinq.Indexing;
using C1.LiveLinq.Collections;
```

(Not all of them are always necessary, but let's put all of them there at once for simplicity)

In order to query a collection in LiveLinq, we need to wrap it in an interface **IIndexedSource<T>** that will tell LiveLinq to take over. Otherwise, standard LINQ will be used. This wrapping is done with a call to the **AsIndexed** extension method, for example:

C#

```
from c in source.AsIndexed() where c.p == 1 select source
```

However, not every collection can be wrapped this way. For instance, if we try this with a **List<T>** source, we'll get a compilation error. To be usable in LiveLinq, a collection must support change notifications, it must notify LiveLinq when changes are made to its objects and when objects are added to or deleted from the collection. This requirement is satisfied for collections used for data binding, that is, implementing either **IBindingList** (WinForms data binding) or **INotifyCollectionChanged**/**INotifyPropertyChanged** (WPF data binding)

In particular, **AsIndexed()** is applicable to ADO.NET collections (**DataTable**, **DataView**) and to LINQ to XML collections.

## Using built-in IndexedCollection Class (LiveLinq to Objects)

Suppose first that we don't care what collection we use for our objects. We have a **Customer** class, something like

```
public class Customer
{
    public string Name { get; set; }
    public string City { get; set; }
}
```

and we rely on LiveLinq to supply the collection class we need.

Then we need look no further than the **C1.LiveLinq.Collections.IndexedCollection** class that is supplied by LiveLinq and is specifically optimized for LiveLinq use:

```
var customers = new IndexedCollection<Customer>();
customers.Add(cust1);
customers.Add(cust2);
...
var query =
  from c in customers where c.City == "London" select c;
```

Note that we can simply use **customers** instead of **customers.AsIndexed()**. It is because the **IndexedCollection<T>** class already implements the **IIndexedSource<T>** interface that LiveLinq needs, there is no need to use the **AsIndexed()** extension method to wrap it into that interface.

There is an important consideration to be kept in mind using your own classes such as **Customer** above for collection elements. The **Customer** class above is so basic that it does not support property notifications. If you set one of its properties in code, nobody will notice it, including any indexes and live views|tag=How to create a live view you may create over that collection. Therefore it is highly necessary to provide property change notifications in such classes. Property change notifications is a standard .NET feature recommended for a variety of reasons, LiveLinq just adds

another reason to do that. You can support property change notifications in your class by implementing the **INotifyPropertyChanged** interfaces. Or you can use LiveLinq for that, by deriving your class from IndexableObject Class and calling **OnPropertyChanging**/**OnPropertyChanged** like this:

C#

```
public class Customer : IndexableObject
{
    private string _name;
    public string Name
    {
        get {return _name} ;
        set
        {
            OnPropertyChanging("Name");
            _name = value;
            OnPropertyChanged("Name");
        }
    }
    private string _city;
    public string City
    {
        get {return _city};
        set
        {
            OnPropertyChanging("City");
            _city = value;
            OnPropertyChanged("City");
        }
    }
}
```

## Using ADO.NET Data Collections (DataSet)

ADO.NET **DataTable** can also be used in LiveLinq, for example:

C#

```
CustomersDataTable customers = ...
var query =
  from c in customers.AsIndexed() where c.City == "London" select c;
```

For that, you'll need to add

C#

```
using C1.LiveLinq.AdoNet;
```

to your source file. The **AsIndexed()** used will be C1.LiveLinq.AdoNet.AdoNetExtensions.AsIndexed. It is specifically optimized for data in ADO.NET DataSets.

## Using XML Data

LiveLinq can query data directly from XML in memory (stored in a LINQ to XML **XDocument** class). It dramatically speeds up LINQ to XML performance by supporting XML indexing (not supported by the regular LINQ to XML).

To use LiveLinq to XML, you need to add

```
using C1.LiveLinq.LiveViews.Xml;
```

to your source file.

Then you need to create some live views over your XML data. Unlike LiveLinq to Objects and LiveLinq to DataSet, where you can query data without live views, in LiveLinq to XML queries and live views are always used together. You are using LiveLinq to XML versions of LINQ query operators if you apply them to a live view. Otherwise, they will be the standard, not LiveLinq query operators. To start creating live views, simply apply the extension method **AsLive()** to an **XDocument**:

```
XDocument doc = …

View<XDocument> docView = doc.AsLive();
```

Once you have a live view, you can use the familiar (from LINQ to XML) query operators **Elements, Descendants** and others (see **XmlExtensions Class**) as well as all Standard Query Operators Supported in Live Views to define a live view. For example,

```
View<XElement> orders = docView.Descendants("Orders");
```

defines the collection of orders in the document. This collection is live; it is automatically kept up to date with the data in the **XDocument** that can be changed by your program. As a result, you can work with data in this collection in the same way as you would work with any dynamic collection, including ADO.NET DataTable or DataView. In particular, you can create indexes over it and use them for speeding up queries and for fast programmatic searches as shown in other sections of this documentation. It means that LiveLinq to XML makes it possible to work with XML data in memory without losing performance, so it is no longer necessary to create custom collection classes or use ADO.NET or other frameworks working with XML data. All you need is LINQ to XML with addition of LiveLinq.

Once you have some live views defined over XML data, you can query them. For example,

```
var query = from Order in orders.AsIndexed()

where (string)Order.IndexedAttribute("CustomerID") == "ALFKI"
```

gives you a query for orders of a particular customer.

> **Note:** It is important to distinguish between live views and queries in LiveLinq to XML. Since you always start with a live view, your query will be a live view by default, unless you specify otherwise. In the example above, we used **AsIndexed()** to specify that we only need a query, don't need that query to define a live view. Live views extend query functionality, so they can be used instead of queries. However, live views have performance overhead, so you should avoid using a live view where a simple query would suffice. As a general rule, avoid creating live views and throwing them away after using them just once to get results. Live views are designed to remain active (hence they are called *live*) and show up-to-date data.

## Using Bindable Collection Classes (Objects)

Any bindable collection class (a class implementing change notifications for data binding) can be used in LiveLinq queries after it is wrapped in an **IndexedCollection<T>**-derived adapter class, which is done by applying a **ToIndexed()** extension method to it.

Using **ToIndexed()**, you can apply LiveLinq to various common data collections. For example, since the **EntityCollection** and **ObjectResult** classes of the ADO.NET Entity Framework are bindable, you can use ADO.NET Entity Framework data in LiveLinq queries and views.

## LiveLinq to Objects: IndexedCollection and other collection classes

Querying general collection classes in LiveLinq is called LiveLinq to Objects, to distinguish it from LiveLinq to DataSet and LiveLinq to XML that query such specific data sources as DataSet and XML. So use LiveLinq to DataSet if you need to query data in a DataSet and LiveLinq to XML if your data is in XML. For other cases, we already saw two options in LiveLinq to Objects:

a. Use **IndexedCollection<T>** if you don't have preexisting collection classes and rely on LiveLinq to supply them.
b. **Apply ToIndexed()** to preexisting collection classes that support data binding.

   And there is also a more advanced option that is not frequently needed:

c. Define your own collection class, usually derived from **IndexedCollection<T>**, if you want some non-standard functionality.

   And finally, an option that can also be considered advanced, but, actually, is not particularly difficult, and allows you to bring virtually any collection into the LiveLinq orbit:

d. You can make any existing collection class **C** usable in LiveLinq provided that it somehow lets you know when changes occur. Then you can create an adapter class implementing the **C1.LiveLinq.IObservableSource<T>** interface and delegating most of its functionality to that preexisting collection class. Having an **IObservableSource<T>** implementation, you can apply the ToIndexed() extension method to it, it will return **IIndexedSource<T>**. LiveLinq extension methods implementing query operators take **IIndexedSource<T>** as their argument, see **IndexedQueryExtensions**.

## Creating Indexes

Now that we can query our collections in LiveLinq, we need to create some indexes for them, otherwise LiveLinq won't query them faster than the standard LINQ does.

Suppose we have a collection implementing **IIndexedSource<Customer>**, for example, like this:

```
var customers = new IndexedCollection<Customer>();
```

or like this:

```
var customers = CustomersDataTable.AsIndexed();
```

The **IIndexedSource<T>** interface has an Indexes|tag=P_C1_LiveLinq_Indexing_IIndexedSource1_Indexes collection (which is actually the only thing it has), so we use that collection to create an index like this:

```
var indexByCity = customers.Indexes.Add(x => x.City);
```

That creates an index of customers by city. Once the index is created, it will be automatically maintained on every change made to the **customers** collection. This maintenance comes with a performance cost. The cost is not high; index maintenance is fast and you can usually ignore it as long as you don't have too many indexes, but it may become a concern if you modify the collection very intensively.

To avoid heavy index maintenance cost, you can use the **BeginUpdate**/**EndUpdate** methods while making massive changes to a collection that has indexes attached to it, for example, while populating the collection.

This index will make queries such as

```
from c in customers where c.City == "London"
```

execute very fast because LiveLinq will use the index to go directly to the required items instead of searching for them by traversing the entire collection. Indexes can speed up many queries, not just this simple one, they can also speedup range conditions (with inequalities), joins, grouping, and other LINQ operators. See LiveLinq Query Performance: Tuning Indexing Performance for the full description of what classes of queries benefit from indexes.

Explicitly creating indexes by adding them to the collection as shown above is only needed if you want direct control over their lifetimes and/or direct access to the indexes (**Index<T>** objects) to use them programmatically (see How to use indexes programmatically). If all you need is to optimize a few LiveLinq queries, you can use the alternative, implicit method of creating indexes, using so called *hints* in LiveLinq queries. A hint .Indexed()|keyword=Indexed(T) Method (T) is an extension method that can be applied to a property in a query. It does not change the value of that property; it only tells LiveLinq to create an index on that property (if possible). So, instead of creating an index by city explicitly as shown above, you could write the query like this:

```
from c in customers where c.City.Indexed() == "London"
```

That would tell LiveLinq to create an index by city if it has not been already created before. This hint does not affect the value of the property, that is, the value of **c.City.Indexed()** is the same as the value of **c.City**.

## Using Indexes Programmatically

Indexes are used by LiveLinq to optimize query execution, but they are also accessible for programmatic use. You can use indexes directly in code, calling methods of the **Index<T>** class to perform a variety of fast searches. This makes LiveLinq indexes useful even outside the framework of LINQ, outside queries.

For example, searching for a particular value can look like this:

```
indexByCity.Find("London")
```

or even like this:

```
indexByCity.FindStartingWith("L")
```

The **Index<T>** class also has other methods for performing fast searches and fast joins and groupings that can be useful in any code, not just in queries: FindGreater, FindBetween, **Join**, **GroupJoin**, and a few others.

Examples of programmatic searches (without LINQ) can be found in the LiveLinq indexing demo, see Query Performance sample application (LiveLinqQueries). In fact, every query that is shown in that demo has an alternative implementation via direct programmatic search in code without LINQ.

## Creating Live View

Consider a simple query

```
from a in A where a.p == 1 select a
```

In standard LINQ, the result of this query is a snapshot. The result collection is formed at the time when the query is executed and it does not change after that. If one of its objects changes so it no longer satisfies the condition, that object will not be removed from the result collection. And if an object in **A** changes so it now satisfies the condition,

that object will not be added to the result collection. The result of even such a simple query is not live, not dynamic, does not change automatically when the base collection **A** changes, not kept in sync automatically with the base data.

In LiveLinq, if we create a view based on that query, it will be live, dynamic, will change automatically when the base data changes, will be automatically kept in sync with the base data.

All we need to do to make a view out of this query is to use the extension method **.AsLive()**:

```
var view = from a in A.AsLive() where a.p == 1 select a;
```

The **AsLive()** extension method is the analog of AsIndexed() Method/**ToIndexed()**, it can be used everywhere where the AsIndexed() Method/**ToIndexed()** extension methods can be used. So, live views are supported in all cases in LiveLinq to Objects, LiveLinq to XML and LiveLinq to DataSet (with some restrictions on the supported query operators, see Query Operators Supported in Live Views). The difference between AsIndexed() Method/**ToIndexed()** and **AsLive()** is that **AsLive()** creates a view, thus enabling LiveLinq both to query to populate the view and to maintain the view after it has been initially populated. The AsIndexed() Method/**ToIndexed()** extension methods do only the first part, enable LiveLinq querying.

The example above is a very simple one, it's just one simple Where condition. There are other tools that can do the same, for example, **DataView** in ADO.NET or **CollectionView** in WPF. The power of LiveLinq is that it supports most of the LINQ operators, including joins and others. So you can create a live view based not just on a simple condition, but on virtually any query you need.

## Binding GUI Controls to Live View

LiveLinq views can be used as data sources for GUI controls because they implement the data binding interfaces so you can simply bind any control to it as to any other data source. For example, in WinForms:

```
View<T> view = from a in A.AsLive() join b in B.AsLive() on a.k equals b.k
        where a.p == 5 select new {a, b};

dataGridView1.DataSource = view;
```

Data binding has long been the tool of choice for most developers creating GUI, but its scope was limited to simple cases only. You could bind to your base data, like your base collections, tables of a data set, etc, but not to your derived data, not to data shaped by some query. Some shaping was supported by some tools, but it was very limited, mostly just filtering and sorting. If you had data derived/shaped in any more complex way, for example, with joins (a very common case), you could not use data binding (more exactly, you could only use it for one-time snapshot binding, show data once, no changes allowed).

LiveLinq makes data binding extremely powerful by removing these limitations. Now you have the full power of LINQ to bind to.

With live views, you can create entire GUI, sophisticated applications, virtually without procedural code, using declarative data binding alone. An example of such application is the LiveLinqIssueTracker demo.

## Using Live Views in Non-GUI Code

Using live views is not limited to GUI. In a more general way, live views enable a declarative style of programming which we tentatively call ***view-oriented programming***. Non-GUI, batch processing code can also be made declarative using live views.

Generally speaking, any application operates on some data, and any data can be seen as either based or derived data. Base data contains the main objects your application is dealing with, like Customers, Orders, Employees, et cetera.

Those objects (collections containing all objects of a certain class, sometimes called the *extent* of a class) are the objects that your application modifies when it needs to make some change in the base data. But applications rarely operate on this base data directly, they rarely directly operate on the entire extent of a class. Usually they filter and shape those extents and combine them together to get to a slice of data they need for a particular operation. This shaping/filtering/joining/slicing of data is the process of getting ***derived*** data (as opposed to ***base,*** underlying data). Before the emergence of LINQ, it was done by purely procedural code, with all the ensuing complexity. LINQ made it declarative, which is a big step forward. But, although declarative, it is not live, not dynamic, and does not react to changes in base data automatically. Accordingly, its reaction to change is not declarative; the programmer needs to react to changes in procedural, imperative code. Complexity is diminished by LINQ, but reacting to change remains complex. And reacting to change is pervasive because change is everywhere.

LiveLinq live views make reacting to change declarative too, thus closing the circle of declarativeness. Now entire applications, not just GUI, can be made virtually entirely declarative.

To give just a small example, we can consider a sample in the LiveLinqIssueTracker demo. It has two operations performed on some issue data:

1. Assign as many unassigned issues to employees as possible, using information on pending issues, product features (every issue belongs to a feature) and assignments.
2. Collect information on open issues for given employees.

Each of these two operations (and in a real application there is, of course, many more operations like this) works on data shaped by a query with joins (see the actual queries in Live Views How To: Use Live Views to Create Non-GUI Applications as a Set of Declarative Rules Instead of Procedural Code). Both operations can be performed more than once during program execution. Data is changing while these and other operations on data are performed. Live views used to implement these operations change automatically with that changing data, so the operations can be kept simple and completely declarative, and as a result of that, robust, reliable and flexible, easily modifiable when business requirements change.

There is nothing else needed for this style of programming in addition to what we already know about how to create live views.

## LiveLinq Programming Guide

The following sections provide information concerning **LiveLinq** programming.

## Query Operators Supported in Live Views

LiveLinq supports all LINQ query operators in its queries. Not all operators have LiveLinq-specific implementation benefiting from indexes and other query execution optimization techniques, but such operations simply use the standard LINQ to Objects (or LINQ to XML) implementations, so it is transparent to the user.

However, not all query operations can be used in live views. This is because not all query operations have incremental view maintenance algorithms, as some would require re-populating from scratch (requerying) every time they are maintained. If your query contains such operations, you won't be able to use that query to create a live view. An attempt to do that will cause a compilation error.

Here is the list of query operators allowed in live views:

| Operator | Notes |
|---|---|
| **Select** | Overload with **selector** depending on the index is not allowed. |
| **Where** | Overload with **predicate** depending on the index is not allowed. |
| **Join** | Overload with **comparer** is not allowed. |

| | |
|---|---|
| **GroupJoin** | Overload with **comparer** is not allowed. |
| **OrderBy** | Overload with **comparer** is not allowed. |
| **OrderByDescending** | Overload with **comparer** is not allowed. |
| **GroupBy** | Overload with **comparer** is not allowed. |
| **SelectMany** | Overloads with **selector** and **collectionSelector** depending on the index are not allowed. |
| **Union** | |
| **Concat** | |
| **Aggregate** | Use LiveAggregate method if you want to optimize performance with incremental view maintenance. |
| **Count** | Use LiveCount method if you want to optimize performance with incremental view maintenance. |
| **Min/Max** | Use LiveMin methods if you want to optimize performance with incremental view maintenance. |
| **Sum** | Use LiveSum method if you want to optimize performance with incremental view maintenance. |
| **Average** | Use LiveAverage method if you want to optimize performance with incremental view maintenance. |

## Query Expressions Supported in Live Views

Apart from the limitations on query operators, a query must satisfy an additional condition in order to be used as a live view. Fortunately, this condition is satisfied in most cases, so you should care about it only if your query contains some special expressions, which is relatively rare (except that all classes used in LiveLinq to Objects must satisfy the property notification condition, but that was already mentioned in Using the built-in collection class IndexedCollection (LiveLinq to Objects). Unfortunately, this condition is not verified by LiveLinq automatically, so it is your responsibility to make sure it is satisfied. If this condition is not satisfied, your view will not react to changes in its base data. We give two descriptions of this condition: one short, for basic understanding, and another detailed, for advanced usage.

Short Description

The short answer is, basically, that in some cases your classes need to provide property notifications. There are two such cases where you need to care about that:

(1) **LiveLinq to Objects.**

There your views' arguments are collections of your own classes, so you must make sure that your classes provide property notifications, otherwise your views will not react to changes of those properties,(see Using the built-in collection class IndexedCollection (LiveLinq to Objects).

(2) **Views over views and indexes over views** (applies to LiveLinq to DataSet and LiveLinq to XML as well as to LiveLinq to Objects).

If you have a view

```
View<T> v;
```

and want to create another view over view **v** (use **v** as its argument) or create an index over **v**, then

make sure that the **T** class has property notifications, otherwise the view (or index) you define over **v** will not react to changes of those properties.

> **Note for LiveLinq to DataSet and LiveLinq to XML:** If the view result class **T** is (or is derived from) DataRow (for LiveLinq to DataSet) or XNode (for LiveLinq to XML), then property notifications there are not needed (so there are no conditions or restrictions in this case, it just works), LiveLinq gets the notifications it needs from the standard events.

Also, it can be mentioned here that you don't need to care about these conditions if the properties of your objects are not changed by your code in other ways than automatically by LiveLinq itself; that is, if you don't have code setting properties of the objects that are elements in the source collections of your views. In particular, if your classes have read-only properties, which includes all anonymous classes, that are often used in LINQ. It is also the case if you use structs, that is, value types (as opposed to classes, reference types), because value types are not references, so you can't change the elements of your collection. In all these cases there is no need in change notifications.

And finally, here is what you need to avoid: A possible source of errors is using property chains, such as for an **Order** class with a **Customer** property:

```
o => o.Customer.City
```

You can do it if you do not modify the **City** property in **Customer** objects. But if you have code changing the **City** property, **LiveLinq** will not reflect the change, because the **Order** object usually does not notify it of changes in **Customer** objects. Note that you can usually modify the query so it has the same effect without using property chains. For example, instead of

```
...Select(o => o.Customer.City)
```

use

```
...Select(o => o.Customer).Select(c => c.City)
```

**Detailed Description**

***Observable and non-observable functions***

Our condition limits the choice of functions (such as result selector, key selector, et cetera) that you can use in your query expression depending on whether or not your classes support property notifications, see Using the built-in collection class IndexedCollection (LiveLinq to Objects).

The property notifications we are talking about are those in the element classes of your view's arguments, not in the class of your view's result (but, of course, one view's result can be another view's argument, since views can be created based on views).

We distinguish two types of functions: observable and non-observable. Observable functions are allowed, non-observable functions must be avoided.

1. Observable functions. A function (expression) is observable if the only non-constant sub-expressions it contains are property or method calls whose values are observable in the sense that any change of that value is always accompanied by the change notification events.

   *Examples of observable functions:*

   x => x.P1

   x => x.P1 + x.P2 + x.M(c1)

   (x, y) => new {x.P1, x.P2, M = y.M(), y.P3}

   Here **P1**, **P2**, **P3**, **M** are properties/methods with change notifications, and **c1** is a value that never changes.

**Note for advanced usage:** Property/method call is understood here to include not only one applied directly to one of this function's parameters, but also, recursively, to the parameters of a function preceding this function in the query, if they can be reached through (possibly a chain of) simple references in object initializers. For example,

....Select((x, y) => new { P1 = x, P2 = y }).Select(z => new { z.P2.A, z.P1.B })

is allowed (if, of course, properties **A** and **B** are observable).

1a. Constants are also observable.

Note that we specifically excluded constant values from the condition above. Any constant values/objects are allowed in a function and don't break its observability. By "constant value", we mean that it remains the same object for every given parameter value. In other words, it does not change with time. The most common example of a constant is the identity function:

x => x

Although the state of the object can change with time, the object itself, as well as the reference to it, stays the same (for a given parameter object, of course, which in this case is the same as the result object), therefore it is a constant. Other examples of constants are:

x => c1

(x, y) => new {x, y}

(x, y) => x + y

x => x == c1 ? c2 : c3

where **c1**, **c2**, **c3** are some constant values.

Constant values can be combined in the same function with observable values, and the resulting function remains observable, for example:

x => new {x, x.P1, P = y.P2 + y.P3 }

where **P1** and **P2** are properties with change notifications.

1b. Object initializers and constructors that don't depend on the arguments are allowed.

The previous examples included only new with anonymous classes, but, in fact, user-defined classes and constructor calls are also allowed without breaking observability, as long as you don't use the function arguments in the constructor, using only constant expressions or nothing at all (a parameterless constructor or an object initializer). So, the following functions are observable (where **c1** is a constant value and **P** and **Q** are observable properties):

(x, y) => new C { x.P, y.Q}

(x, y) => new C(c1) { X = x, P = y.P }

and the following are non-observable:

(x, y) => new C(y) { x.P, y.Q}

(x, y) => new C(c1, x) { x, y.P }

2. Non-observable functions.

   As the name suggests, any function that does not satisfy the condition above is considered non-observable. Note that this condition of observability depends not only on the function itself but also on the class of the argument of that function; that class must have property change notifications for everything that is used in the function that is not constant. So, even the simplest functions, such as **x => x.P** can be non-observable if **P** is non-observable (that is, the class does not issue change notifications for **P**). So, the first and most common examples of non-observable functions are the same as above but in situations where at least one of the properties **P1**, **P2** is not observable, that is, the class does not provide change notifications for it:

```
x => x.P1
(x, y) => new {x.P1, y.P2 }
```

This can occur if you simply forgot to add the code providing notifications (see Using the built-in collection class IndexedCollection<T> (LiveLinq to Objects)|tag=Using_the_built_in_collection_class_IndexedCollectionT_LiveLinq_to_Objects).

Another possible cause is a property returning a calculated value, like in

```
public class Customer
{
   public List<Order> orders;
   public int OrderCount { get {return orders.Count;} }
}
```

unless you specifically take care to issue a property change notification in the **Customer** class every time its **orders.Count** changes.

Yet another possible cause is using a chain of properties, like, for an **Order** class with a **Customer** property:

```
o => o.Customer.City
```

Note that any expression, including the two above, can be made observable if you take special care to trigger property change notification every time its value changes, but it requires code written specifically for that purpose. For example, with the last function, you can trigger property change notification events for the **Orders** collection every time the **City** property in the **Customer** class changes.

## View Maintenance Mode

Live views can be used in different kinds of applications. They can be used in GUI, interactive applications and in non-GUI, batch processing-style applications (see How to use live views in non-GUI code). Live views are optimized for both modes, GUI (interactive) and non-GUI (batch). They distinguish between these modes and operate accordingly. In GUI, live views react to a change immediately when the change happens. They do it fast, using incremental algorithms without re-populating themselves, (see Maintaining the view: Incremental View Maintenance), but still it is not always suitable for batch, non-interactive processing. In GUI, interactive programs, with data binding, immediate reaction to change is what is usually needed because the user needs to see the change on the screen. In batch processing, on the other hand, a view may be accessed long after the change occurred or even not at all. So updating that view before it is actually accessed by the program may be an unnecessary drain on resources. By default, live views distinguish between these two modes automatically, but the programmer has an option to control that using the **MaintenanceMode** property. In **Immediate** mode, which is the default for GUI, the view is maintained immediately on every change. In **Deferred** mode, which is the default in a non-GUI case (when there are no listeners attached to the view), the view is maintained on demand. It is not kept in sync with base data until it is needed or until there is a request for data from that view. When such a request arrives, the view sees that it is in a "dirty" unsynchronized state, so it automatically updates (maintains) itself to come in sync with the changed base data.

## Updatable Views

Live views are bi-directionally updatable. The first direction is from base data to the view: the base (source) data can be modified and the view will be automatically updated and then synchronized with the changed base data. That is what makes the views *live*. There is also the second, opposite direction: data can be changed directly in the view. This can be done programmatically (using the ViewRow), and it can also be done via data binding. Updating data directly in the view is especially common if you bind a GUI control, such as a grid, to a view. An example of updating data in a

view via data binding can be seen in one of the first samples, Basic LiveLinq Binding, among others.

We call a view *updatable* if data can be modified directly in the view. This term only concerns the second direction of updating data. The first direction mentioned above, updating base data, is always possible without restrictions for any live views. So, when we say that a view is not updatable (is *read-only*), it does not mean that the data in the view cannot change. It can change to reflect changes in base data because every view in LiveLinq is live. It merely means that data cannot be changed directly in the view,  you need to change base data if you want to modify it.

Not all live views are updatable, and even if the view as a whole is updatable, some properties of it may be read-only. These are properties that don't have direct correspondence to a property in the view's source data. Updating a view ultimately means updating that view's base data, that is, one of the view's sources, because the view itself is only virtual, does not have data of its own, shows (filtered and shaped) data of its sources. This is why a view field (property) can be updated only if it can be put into a direct correspondence with a property in the source. For example, in the following view

```
for c in customers where c.City == "London"
select new
{
   c.City,
   c.CompanyName,
   FullName = c.FirstName + " " + c.LastName
}
```

**City** and **CompanyName** are updatable because they directly correspond to the **City** and **CompanyName** fields of the **customers** source. But **FullName** is not updatable, because there is no single field (property) in the source to which it corresponds, it is a combination of two source properties.

Any update of a view, which can be adding, deleting or modifying a view item, is performed by performing the corresponding operation, adding, deleting or modifying a single item in one of the view's sources. Although the effect is usually obvious and it usually does what is intended, it is important to know this simple rule, to understand it formally and exactly, because the result can sometimes be unexpected if you don't take this rule into account. For example, if you modify an item (or add a new one) in the above view so its **City** value is not "London", that item will disappear from the view.

The above rule stating that updating a view item is equivalent to updating an item of one of the view's sources, implies that only one of the view's sources can be updatable. A **Join** view has two sources, so we must determine which of them is updatable. By default, a join view is read-only. If you need to make one of its two parts updatable, use the extension method AsUpdatable(), for example:

```
for o in orders.AsUpdatable()
join c in customers on o.CustomerID equals c.CustomerID
select new
{
   o.OrderDate,
   o.Amount,
   c.CompanyName
}
```

Here order data (date and amount) will be updatable and customer data (**CustomerName**) read-only.

To find out if a view is updatable, use the property View.IsReadOnly. The list of all properties of a view accessible to data binding and programmatic access, with full information including their updatability, can be obtained using the property ViewRowCollection.Properties.

Updating data directly in a view in code is done through a special class ViewRow, each view row representing a view item for programmatic access and data binding purposes. For details of using this class in code see reference documentation for ViewRowCollection and ViewRow.

# Live View Performance

First performance consideration with live views is that, naturally, live view functionality has a price. Maintaining the view in sync with base data is fast and optimized, but still it consumes some resources when base data changes and the view is maintained. And it consumes some additional resources when it is first populated as well. That additional cost is not high, but it exists, manifesting itself mostly in additional memory consumption: when a live view is populated, it creates some internal data in memory to help it maintain itself faster on base data changes.

This additional memory consumption is moderate—roughly equivalent to the amount of memory needed for the resulting list itself. So, although additional resources needed for live view functionality are light to moderate, the obvious suggestion is to use live views only where you are really interested in keeping the result of a query live, or, in other words, where you need that result more than once and the base data is changing so that the result is changing too.

Other than this, there are two different aspects to live view performance:

## Populating the view: query performance

Query performance is how long it takes to populate the view for the first time by executing the query (or re-populate by re-executing the query, see **View.Rebuild**). This is covered in LiveLinq query performance: logical optimization and LiveLinq Query Performance: Tuning Indexing Performance.

## Maintaining the view: Incremental View Maintenance

Maintaining live views on base data changes is optimized using techniques known as Incremental View Maintenance. It means that a view does not simply re-populate itself, it adjusts to accommodate base data changes in a more smart way. The changes to the view are made locally, incrementally, calculating the delta in the view from the delta in the base data. In most cases, it is very fast and does not require any special performance tuning from the user.

As a guideline to estimating the time of view maintenance, we can say that it is roughly proportional to the number of changed items (including added and deleted ones) in the view's resulting collection. So, if only few items change as a result of a change in base data, the view maintenance time is very small and probably negligible, but it can be considerable if a considerable part of the view result changes. It can even become more costly to maintain a view than to re-query it from scratch, if, for example, half of the resulting list changes.

Finally, it is worth noting that maintenance performance for a view does not depend on its query performance. It does not matter how long it takes to populate the view initially; maintenance time is roughly the same and depends on the size of the change in (delta) and not on the size of the overall data.

# Logical Optimization in LiveLinq Query

Standard LINQ to Objects does not perform any logical optimization, it executes queries exactly as they are written. And, of course, standard LINQ to Object does not use indexes for optimization. In comparison, LiveLinq performs physical optimization (using indexes, if they are present) and logical optimization (re-writing the query in a more efficient form before processing it).

However, LiveLinq does not contain a full-scale query optimizer like one in a relational database such as SQL Server or

Oracle, so it can still matter how a query is written. But it is largely limited to join ordering. LiveLinq does not try to re-order your joins. Joins are executed always in the same order as specified in the query. So, you should avoid writing queries with obviously inefficient join order. But it must be noted that it is a relatively rare problem (and the same consideration, of course, applies to the standard LINQ to Objects anyway). It can be an issue in a query like

C#

```
from a in A
join b in B on a.k equals b.k                    (1)
where b.p == 1
```

if there are, for example, only 10 elements in **B** with **b.p == 1** and only 100 elements in **A** that satisfy **a.k == b.k** and **b.p == 1**, but the overall number of elements in **A** is many times higher, for example, 10000. Then the query above will require 10000 "cycles", whereas rewritten with the right join order,

C#

```
from b in B
join a in Aon b.k equals a.k                     (2)
where b.p == 1
```

this query will require only 100 cycles to complete. Note that the position of **where** is not important. The above query has the same performance as

C#

```
from b in B
where b.p == 1                                    (3)
join a in A on b.k equals a.k
```

That's because of LiveLinq query optimization: LiveLinq re-writes (2) internally to (3) when it executes it, because it is preferable to check conditions before performing other operations and to exclude elements that don't contribute to the result. This is one of the logical optimizations performed by LiveLinq internally.

## Tuning Indexing Performance in LiveLinq Query

Query speedup achieved on any particular query by LiveLinq using indexes for optimization depends on how that query is written. This dependence is usually not dramatic. LiveLinq does a fair job of recognizing opportunities to use indexes for optimization regardless of the way the query is written. For example, it will still execute a query efficiently using indexes even if the condition consists of multiple terms connected with logical operators.

If you want to ensure that your indexes are used effectively by LiveLinq, consider the following guidelines:

Elementary predicates that can benefit from indexing

LiveLinq can use an index by a property **P** in an elementary predicate (a condition without boolean operations) that has one of the forms (patterns) (1) – (6) listed below. The simplest and perhaps the most common of such elementary predicates is a **where** condition with an equality involving a property, as in

C#

```
from x in X where x.P == 1 select x
```

It will use the index by **x.P** provided that such index exists, which, as described in How to create indexes, can be ensured either by creating this index explicitly

C#
```
X.Indexes.Add(x => x.P);
```

or by using the **Indexed()** method hint:

C#
```
from x in X where x.P.Indexed() == 1 select x
```

In fact, LiveLinq supports a more general indexing. It does not necessarily have to be a property of **x**, it can be any expression depending on **x** (and only on **x**). This can come in handy, for example, if we are querying an untyped ADO.NET DataTable, so, because it is untyped, it does not have properties that we can index and query by their names and we need to use a query like this:

C#
```
from c in customersTable.AsIndexed() where c.Field<string>("CustomerID") == "ALFKI"
select x
```

Then we can use an index by the following expression:

C#
```
X.Indexes.Add(c => c.Field<string>("CustomerID"));
```

We will quote only the most common case of a simple property, **x.P** in the list below, but keep in mind that it can be replaced everywhere with any expression depending on **x** (and only on **x**).

Following is the list of patterns recognized by LiveLinq as allowing optimization with indexes:

1. Equality:

   **x.P == Const** (**Const** is a value constant in the given query operator, which means that it can be an expression depending on external parameters and variables, but it must have the same value for all elements that are tested by this **where** condition).

   Example:

   C#
   ```
   from o in Orders.AsIndexed() where o.OrderID.Indexed() == 10011
   ```

2. Inequality:

   **x.P op Const**, where **op** is one of the comparison operators: >, >=, <, <=

   Example:

   C#
   ```
   from o in Orders.AsIndexed() where o.OrderID.Indexed() > 10011
   ```

3. Left and right parts of an equality or inequality are interchangeable (commutative).

   Example: The following query will use indexing exactly as the one in (1):

```
C#
from o in Orders.AsIndexed() where 10011 == o.OrderID.Indexed()
```

4. StartsWith:

   **x.P.StartsWith(Const)**, if **x.P** is of type **string**.

   Example:

```
C#
from o in Orders.AsIndexed() where o.CustomerID.StartsWith("A")
```

5. Belongs to (in, is an element of):

   **ConstColl.Contains(x.P)**, if **ConstColl** implements **IEnumerable<T>** where **T** is the type of **x.P**.

   Example:

```
C#
from o in Orders.AsIndexed()
 where (new int[]{"ALFKI", "ANATR", "ANTON"}).Contains(o.CustomerID)
```

6. Year comparison:

   **x.P.Year op Const**, where **x.P** is of type **DateTime** and **op** is any of the comparison operators ==, >, >=, <, <=

   Example:

```
C#
from o in Orders.AsIndexed()
 where o.Date.Indexed().Year == 2008
```

Other elementary predicates ***don't*** use indexes. Examples where indexing will not be used:

```
C#
from o in Orders.AsIndexed() where o.Freight > 10
(if there is no index defined for the Freight property)
```

```
C#
from o in Orders.AsIndexed() where o.OrderID.Indexed() < o.Freight
(comparison must be with a constant, not with a variable value)
```

```
C#
from o in Orders.AsIndexed() where o.OrderID.Indexed() != 10011
(only certain comparisons are used, ! (not equal) is not one of them)
```

**Boolean operators**

Conjunction (&&) and disjunction (||) are handled by LiveLinq optimizer, including their arbitrary combinations and including possible parentheses. Other boolean operators, such as negation (!) are not handled by the optimizer; they block its use of indexing.

***Conjunction:***

Conjunction does not prevent the use of indexes. For example,

```
C#
from o in Orders.AsIndexed() where o.Freight > 10 && o.Lines.Count > 5
```

will use the index by **Freight** property (supposing such index exists) even though the second condition cannot use indexes. LiveLinq will simply check the other condition for each item that is found using the index from the first condition.

***Conjunction of conditions using the same property***

Moreover, conjunctions of conditions with the same property will be optimized to render the best execution plan. For example, if the **Freight** property is indexed,

```
C#
from o in Orders.AsIndexed() where o.Freight > 10 && o.Freight < 20
```

will not go through all orders with **Freight > 10** and check if it is less than 20 for each of them but will use the index to go directly to the orders between 10 and 20 and won't check any other orders.

***Conjunction of conditions with different properties: where subindexes come into play***

Conjunctions using different properties, for example

```
C#
from o in Orders.AsIndexed() where o.CustomerID == "ALFKI" && o.Freight < 20
```

can utilize subindexes, see **Subindex(T, TKey) Class**. If the index by **CustomerID** has a subindex by **Freight**, LiveLinq will not check all orders with **CustomerID** equal to "ALFKI" (which can be quite numerous). It will go directly to the subindex corresponding to the value "ALFKI" (only one such subindex exists, and it can be accessed directly, without any search) and enumerate the items in that subindex whose **Freight** is less than 20. Both operations (finding a subindex and enumerating a range of items in it) are direct access operators, without search, so the query is executed in the fastest way possible, without spending any time on checking items that  don't contribute to the result.

***Disjunction:***

```
C#
(a)     For an indexed property x.P,
```

```
x.P == Const1 || x.P == Const2
is handled by re-writing the condition as
(new ...[]{Const1, Const2}).Contains(x.P)
or build the help system.
```

C#

```
(b)     If two different properties x.P and x.Q are indexed, then
x.P op Const1 || x.Q op Const2 (op is ==, <, >, <=, etc)
is handled by re-writing the query as a union of two separate queries with
corresponding elementary conditions.
```

**Join**

If either of the two key selectors in a join, that is, either **x.K** or **y.K** in

C#

```
from x in X
join y in Y on x.K equals y.K
```

is indexed, LiveLinq will use that index to perform the join.

Ideally, both **x.K** and **y.K** are indexed, and then LiveLinq will execute the join in the fastest way possible (using the so called *merge join* algorithm, which is very fast; basically, it takes the same time to build as to traverse the result of such join, there is no time lost on anything).

If there are no indexes on **x.K** or **y.K**, LiveLinq will still try to find the best algorithm for the join, which is sometimes merge join (if both parts of the join are ordered by the merge key) and sometimes the *hash join* algorithm used by the standard LINQ to Objects.

As a general guideline, it should be noted that defining indexes only for optimizing **Join** operations in your query will rarely lead to dramatic speedups. That's because the hash join algorithm (the one used in the standard LINQ, it does not need indexes) is already quite fast. It does make sense to define indexes for join keys sometimes, but consider it when you are fine-tuning your query, not when you need the first quick and dramatic speedup. For dramatic speedups, first consider optimizing your **Where** conditions, that can speed up your query often hundreds and even thousands times (depending on the query, of course).

## Creating News Views Based on Existing Views

Multiple queries with parameters can often be replaced with a single “big” live view. That can help making code clear and simple and can often make it faster. This technique is based on two features of live views:

- A view can be based on other views in the same way as it can be based on base data.
- Indexes can be created for a view in the same way as indexes are created for base data.

So, instead of issuing many queries by varying a parameter value, you can create a single live view, create an index for it, and, when you need the list of items corresponding to a particular value of the parameter, you simply get those items from the index for that particular value.

It is illustrated in the LiveLinqIssueTracker demo:

The **Assigned Issues** form uses multiple views, a separate view for every employee: views depending on a parameter

**employeeID** (we are giving LiveLinq to DataSet versions of the views here, Objects and XML versions are similar):

C#

```
from i in _dataSet.Issues.AsLive()
join p in _dataSet.Products.AsLive()
    on i.ProductID equals p.ProductID
join f in _dataSet.Features.AsLive()
    on new { i.ProductID, i.FeatureID }
        equals new { f.ProductID, f.FeatureID }
join e in _dataSet.Employees.AsLive()
    on i.AssignedTo equals e.EmployeeID
where i.AssignedTo == employeeID
select new Issue
{
    IssueID = i.IssueID,
    ProductName = p.ProductName,
    FeatureName = f.FeatureName,
    Description = i.Description,
    AssignedTo = e.FullName
};
```

The demo application also contains an alternative implementation of the same functionality, in the form **Assigned Issues 2**. That form uses a single view containing data for all employees, instead of multiple views depending on a parameter. This single view does not have parameters:

C#

```
_bigView =
    from i in _dataSet.Issues.AsLive()
    join p in _dataSet.Products.AsLive()
        on i.ProductID equals p.ProductID
    join f in _dataSet.Features.AsLive()
        on new { i.ProductID, i.FeatureID }
            equals new { f.ProductID, f.FeatureID }
    join e in _dataSet.Employees.AsLive()
        on i.AssignedTo equals e.EmployeeID
    select new Issue
    {
        IssueID = i.IssueID,
        ProductName = p.ProductName,
        FeatureName = f.FeatureName,
        Description = i.Description,
        AssignedToID = e.EmployeeID,
        AssignedToName = e.FullName
    };
```

That view is indexed by the employee id field:

```
_bigView.Indexes.Add(x => x.AssignedToID);
```

so we can retrieve the items for a particular employee id at any time very fast.

Moreover, we can create a live view of that data given an employee id value (which is **comboAssignedTo.SelectedIndex** in this demo), simply by creating a view over the big view:

```
from i in _bigView where i.AssignedToID == comboAssignedTo.SelectedIndex select i;
```

## Creating Non-GUI Applications using Live Views

In addition to making GUI applications declarative, LiveLinq also enables a programming style (which we tentatively call ***view-oriented programming***) that makes non-GUI, batch processing declarative too. See How to use live views in non-GUI code for the introductory description of this technique.

The LiveLinqIssueTracker demo application contains a **Batch Processing** form where this technique is demonstrated by implementing two actions:

1. Assign as many unassigned issues to employees as possible. Looking at the feature to which a particular issue belongs, find an employee assigned to that feature, and, if that employee does not have other issues for that feature, assign that issue to that employee.
2. Collect information on all open (not fixed) issues for a given employee (perhaps for the purpose of emailing that list to that employee).

We perform action (1) by defining the following view (we are giving LiveLinq to DataSet versions of the views here, Objects and XML versions are similar):

C#

```
_issuesToAssign =
    from i in issues
        where i.AssignedTo == 0
    join a in _dataSet.Assignments.AsLive()
        on new { i.ProductID, i.FeatureID }
            equals new { a.ProductID, a.FeatureID }
    join i1 in issues
        on new { i.ProductID, i.FeatureID, a.EmployeeID }
            equals new { i1.ProductID, i1.FeatureID, EmployeeID = i1.AssignedTo }
        into g
    where g.Count() == 0
    join e in _dataSet.Employees.AsLive()
        on a.EmployeeID equals e.EmployeeID
    select new IssueAssignment
    {
        IssueID = i.IssueID,
        EmployeeID = a.EmployeeID,
        EmployeeName = e.FullName
    };
```

and performing the operation with simple code based on that view:

C#

```
foreach (IssueAssignment ia in _issuesToAssign)
    _dataSet.Issues.FindByIssueID(ia.IssueID).AssignedTo = ia.EmployeeID;
```

Action (2) is performed by defining the following view:

C#

```
from i in _dataSet.Issues.AsLive()
join p in _dataSet.Products.AsLive()
    on i.ProductID equals p.ProductID
join f in _dataSet.Features.AsLive()
    on new { i.ProductID, i.FeatureID }
        equals new { f.ProductID, f.FeatureID }
join em in _dataSet.Employees.AsLive()
    on i.AssignedTo equals em.EmployeeID
    where i.AssignedTo == employeeID && !i.Fixed
select i.IssueID;
```

If we need to perform the operations (1) and (2) just once, then there is no point in using live views. But suppose we are writing a program that needs to perform those actions (1) and (2) many times as steps of the overall algorithm it is executing. This is a common case, especially in server-side programming.

Without live views, we would either need to requery each time, which is very costly, or we would need to create and maintain some collections throughout our processing algorithm. Those collections would need to be maintained by manual code, often complicated, and written by different programmers and often containing bugs. Different people write different functions (actions (1) and (2) are, of course, just two small examples of such functions), they need to know what others are doing if they want to keep it consistent. All this is hard to maintain. When a new action or function is added a year after that, the logic that connects everything is by that time forgotten, the new action breaks that logic, and so the vicious cycle of too complicated programming goes on.

**Compare this with how it can be done with live views:**

Actions (1) and (2) are not actually procedures, they are declarative rules. Rule (1) states that this view contains the assignments to be made. Whatever changes are made to our data, this view will always contain the needed assignments, regardless of anything we add a year from now or at any other time. The logic is guaranteed to be correct because it is a declarative rule, not procedural logic.

And that rule (1) works fast. We can perform that action a thousand times over data that is always changing, and every time it will recompute only the required amount of data, only the amount that is affected by changes. It will perform only the needed ***incremental*** recomputations.

Same with rule (2): it is a declarative rule, and it executes fast without repeating calculations, recalculating only those parts that are affected by changes.

If a substantial part of our algorithm is programmed in this ***view-oriented*** way, as a set of rules like (1) and (2), then they all work together, their consistency is automatically maintained. Ideally, we can represent our entire algorithm with views/rules like that. If not, a part of it can be implemented like that and the rest can be done with the usual procedural code.

# Samples

This section allows you to get access to Getting Started Samples as well Sample Applications.

# Getting Started Samples

These are the sample projects described in the Getting Started part of the LiveLinq documentation. You can create those projects yourself following the complete step-by-step instructions. The pre-created projects are included in the

LiveLinq distribution package for your convenience.

## HowTo Samples

The following sections describe **LiveLinq**'s HowTo samples.

## Indexing samples

HowTo Indexing samples demonstrate basic use of the first of the two areas of LiveLinq functionality: using indexes to make LINQ queries faster. They don't use live views, except for LiveLinqToXml, where live views are necessary to define base data collections.

Every sample uses two queries: one the most basic and the other a little more complex, with a join. The queries are identical in functionality in all three samples, differing only in the nature of the underlying data: user object collections (LiveLinqToObjects), ADO.NET DataSet (LiveLinqDataSet) or XML (LiveLinqToXml).

Every sample shows two alternative ways of creating indexes: explicitly, by adding to the **Indexes** collection, or implicitly, by using hints in queries.

**LiveLinqToObjects**

This sample shows how to use the **IndexedCollection** class to create and query data collections using indexes to speed up query performance.

**LiveLinqToDataSet**

This sample shows how to use LiveLinq to query data residing in an ADO.NET DataSet, how to create indexes over that data that make querying faster than regular LINQ to DataSet queries. Both strongly typed and untyped datasets are demonstrated in the sample.

**LiveLinqToXml**

This sample shows how to use LiveLinq to query XML data. It creates indexes on 'customers' and 'orders' live views defined directly over XML data. This easy XML indexing allows to dramatically speed up LINQ queries over XML data, often make them hundreds of times faster than regular LINQ to XML, see Query Performance sample application (LiveLinqQueries) for performance metrics.

## Live view samples

HowTo LiveViews samples show the second and main part of LiveLinq functionality: LINQ queries as live views, automatically reflecting changes in the data on which they are based, so they can be used to build applications declaratively, minimizing procedural, manual coding.

Every sample uses two views over the same data: one the most basic (filtered view) and the other a little more complex, with a join. The functionality is identical in all three samples, differing only in the nature of the underlying data: user object collections (LiveLinqToObjects), ADO.NET DataSet (LiveLinqDataSet) or XML (LiveLinqToXml).

To see live views in action, you can try, for example:

- Change the **ShipCity** value from "London" to "Colchester" or vice versa in any of the two grids and leave the current row (to commit the change). Observe that the value in the other grid changes accordingly.
- Change the **ShipCity** value to any string other than "London" and "Colchester" in any of the two grids.

Observe that the row disappears from both grids.

See also the Live views sample application (LiveLinqIssueTracker) for more advanced functionality.

**LiveViewsObjects**

This sample demonstrates basic live view functionality for views based on user-defined object collections (LiveLinq to Objects).

**LiveViewsDataSet**

This sample demonstrates basic live view functionality for views based on ADO.NET DataSet (LiveLinq to DataSet). This sample uses a typed data set, but untyped data sets can be used as well, as shown in the LiveLinqToDataSet sample.

**LiveViewsXml**

This sample demonstrates basic live view functionality for views based on XML (LiveLinq to XML).

## Sample Applications

The following sections describe **LiveLinq**'s sample applications.

## Query Performance sample application (LiveLinqQueries)

The querying demo shows all three LiveLinq varieties: LiveLinq to Objects, LiveLinq to DataSet and LiveLinq to XML, side by side, performing the same queries on the same data. The data are the good old Northwind **Customers** and **Orders** tables, in the three incarnations: custom object collections, a data set and XML, respectively.

The demo contains various queries, some with a parameter. For each query it shows the code, how that query is formulated in each of the three LiveLinq varieties. Every query is implemented in two different forms:

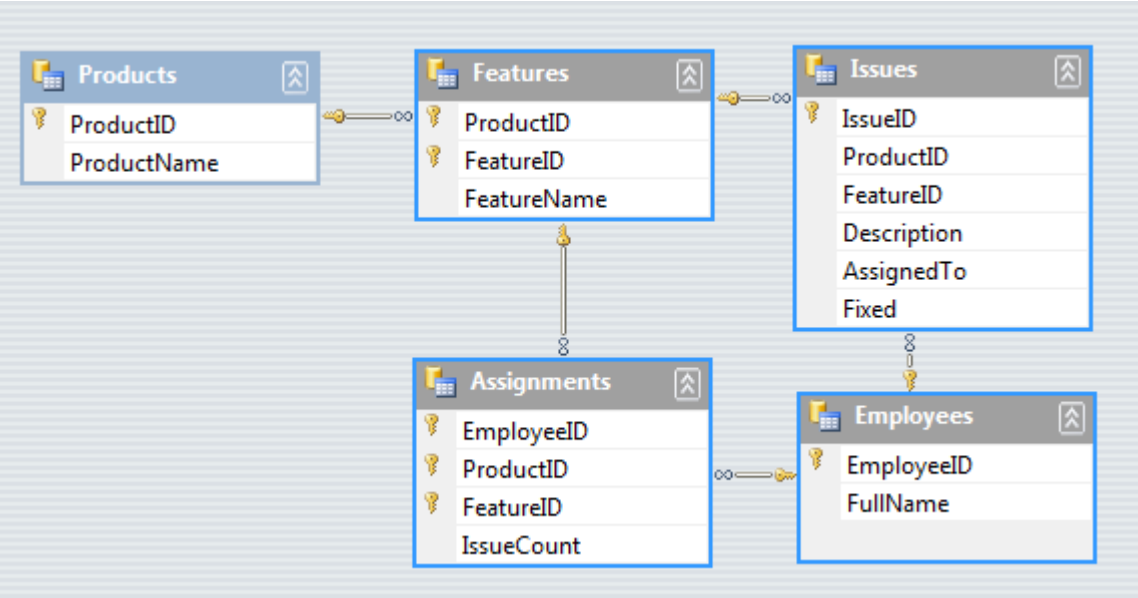
- Programmatic index search; that is, code without LINQ, directly using indexes for search, and
- LINQ query syntax.

These two LiveLinq implementations are compared with a standard LINQ implementation shown along with them. For each query, the demo shows the speedup factor: how much faster the LiveLinq implementation is compared with the standard LINQ one.

## Live views sample application (LiveLinqIssueTracker)

This demo shows how live views can be used to construct entire GUI applications based almost entirely on declarative queries/views and data binding, with little procedural code. It also contains an example of how live views can be used in non-GUI, batch, perhaps server-side processing.

It is a mockup bug/issue tracking application in a fictitious company that produces Shakespeare's plays. The demo comes in three variations, each built on top of a different data store: Collections (uses LiveLinq to Objects), ADO.NET (uses LiveLinq to DataSet) and XML (uses LiveLinq to XML). It uses WinForms data binding, but WPF data binding could be used instead just as well and the live views would be exactly the same.




The data schema contains **Products** (each product is a Shakespeare's play). Each product has features (which, incidentally, are roles in the play). And there are **Issues**, in other words, bugs (such as bad acting in a certain scene). Also, there are **Employees** who are assigned those issues (incidentally, those employees are actors playing those roles), and, an important part of this schema, there are **Assignments**: every employee is assigned a certain number of features to take care of. These assignments can overlap, meaning that more than one employee can be assigned to a feature.

When we start the application and open the **Assigned Issues** form, we can see issues assigned to any given employee. The view representing it is (we are giving LiveLinq to DataSet versions of the views here, Objects and XML versions are similar):

C#

```
from i in _dataSet.Issues.AsLive()
                    join p in _dataSet.Products.AsLive()
                        on i.ProductID equals p.ProductID
                    join f in _dataSet.Features.AsLive()
                        on new { i.ProductID, i.FeatureID }
                        equals new { f.ProductID, f.FeatureID }
                    join e in _dataSet.Employees.AsLive()
                        on i.AssignedTo equals e.EmployeeID
                    where i.AssignedTo == employeeID
                    select new Issue
                    {
                        IssueID = i.IssueID,
                        ProductName = p.ProductName,
                        FeatureName = f.FeatureName,
                        Description = i.Description,
                        AssignedTo = e.FullName
                    };
```

It is a non-trivial LINQ query, with joins and filtering, and the demo shows that it is live, dynamic, automatically reacts to changes in data, and that GUI controls such as **DataGridView** can be bound to it.

For example, if we add an issue in the **Add Issue** form and assign it to the same employee that is shown in the **Assigned Issues** form, the list of issues for that employee is automatically updated to include the newly created issue.

The **Assigned Issues** form also demonstrates the difference between **Immediate** and **Deferred** maintenance mode,

allowing switching between them with two radio buttons. If we switch to the **Deferred** mode, meaning update on demand, and add an issue, that change is not immediately reflected in the view; it is reflected only when we request data from that view, for example, by changing the employee and then changing it back.

There is also a **(Re)assign Issue** form that uses basically the same view, so, without writing code using only live views and data binding, we can do things like assign and reassign issues.

**Assigned Issues** and **(Re)assign Issue** forms can be opened in any number of instances side by side. You can experiment with opening several forms like that, re-assigning issues in one of them, and then observe the changes automatically reflected in all others.

The demo also shows some actions more complicated than that, such as how live views can change more than just one row at a time. There is a form called **All Issues Concerning an Employee** that shows all issues for all features assigned to a particular employee using the following view:

C#

```
from a in _dataSet.Assignments.AsLive()
                  join p in _dataSet.Products.AsLive()
                      on a.ProductID equals p.ProductID
                  join f in _dataSet.Features.AsLive()
                      on new { a.ProductID, a.FeatureID }
                          equals new { f.ProductID, f.FeatureID }
                  join i in _dataSet.Issues.AsLive()
                      on new { a.ProductID, a.FeatureID }
                          equals new { i.ProductID, i.FeatureID }
                  join e in _dataSet.Employees.AsLive()
                      on i.AssignedTo equals e.EmployeeID
                  where a.EmployeeID == employeeID
                  select new Issue
                  {
                      IssueID = i.IssueID,
                      ProductName = p.ProductName,
                      FeatureName = f.FeatureName,
                      Description = i.Description,
                      AssignedTo = e.FullName
                  };
```

Assignments of features to an employee can be changed in a **(Re)assign Features** form. You can experiment with changing the set of features assigned to an employee and see how the changes, which can be quite massive, automatically occur in the **All Issues Concerning an Employee** form. And although it's not easy to see with such a small data set (unless you add thousands of issues to it, which likely would be the case in a real-life application), the changes to the views shown in the grids are fast, without delays for re-querying the views and refreshing the entire grid, thanks to the incremental view maintenance algorithms implemented in LiveLinq.

The demo also shows how to create views based on other views and how to create indexes on views, see Live Views How To: Create Views Based on Other Views and Create Indexes on Views, and how to use live views to create non-GUI applications as a set of declarative rules instead of procedural code, see Live Views How To: Use Live Views to Create Non-GUI Applications as a Set of Declarative Rules Instead of Procedural Code.