

Getting Started with ComponentOne LiveLinq

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 (**NWIND.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.

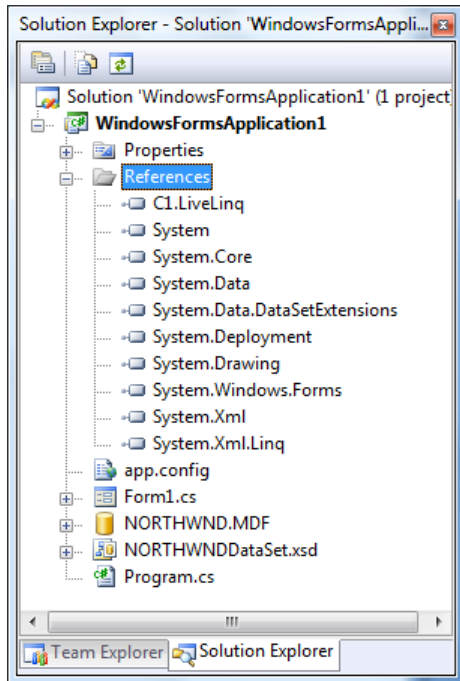
Optimizing LINQ Queries with LiveLinq

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 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:

LINQ 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 Cl.LiveLinq;
using Cl.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 than 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:

```

using Cl.LiveLinq;
using Cl.LiveLinq.AdoNet;
using Cl.LiveLinq.LiveViews;
using Cl.LiveLinq.Collections;

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

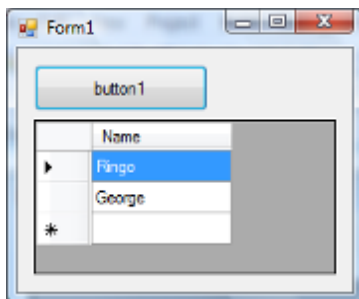
    // bind list to grid (before adding elements)
    this.dataGridView1.DataSource =
        from c in contacts.AsLive()
        where c.Name.Contains("g")
        select c;

    // 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 **IndexableObject** 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:

ProductID	ProductName	SupplierID	CategoryID	Quantity
1	Chai	1	1	10 boxes
2	Chang	1	1	24 - 12 c
24	Guaraná Fa...	10	1	12 - 355
34	Sasquatch Ale	16	1	24 - 12 c
35	Steeleye St...	16	1	24 - 12 c

Product Name: Chai

Unit Price: 18.0000

Quantity per Unit: 10 boxes x 20 bags

Units In Stock: 39

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:

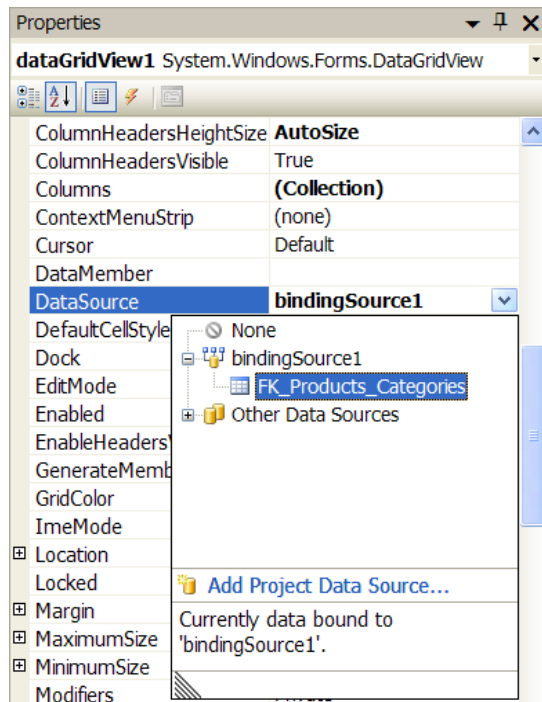
1. Add a new **BindingSource** component to the form
2. Select the new **BindingSource** component, select its **DataSource** property in the property window, and use the drop-down editor to select the **NORTHWINDDataSet** data set.
3. Still with the **BindingSource** component selected, use the drop-down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

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

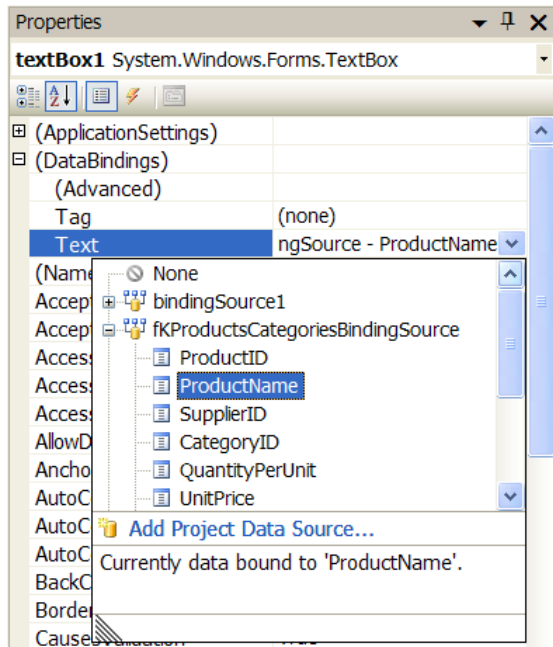
For the **ComboBox**, set the following properties:

DataSource 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.

Switching 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:

```

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 Cl.LiveLinq;
using Cl.LiveLinq.Adonet;
using Cl.LiveLinq.LiveViews;

```

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

```

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:

```

//
// 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:

```
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:

```
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:

```

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).

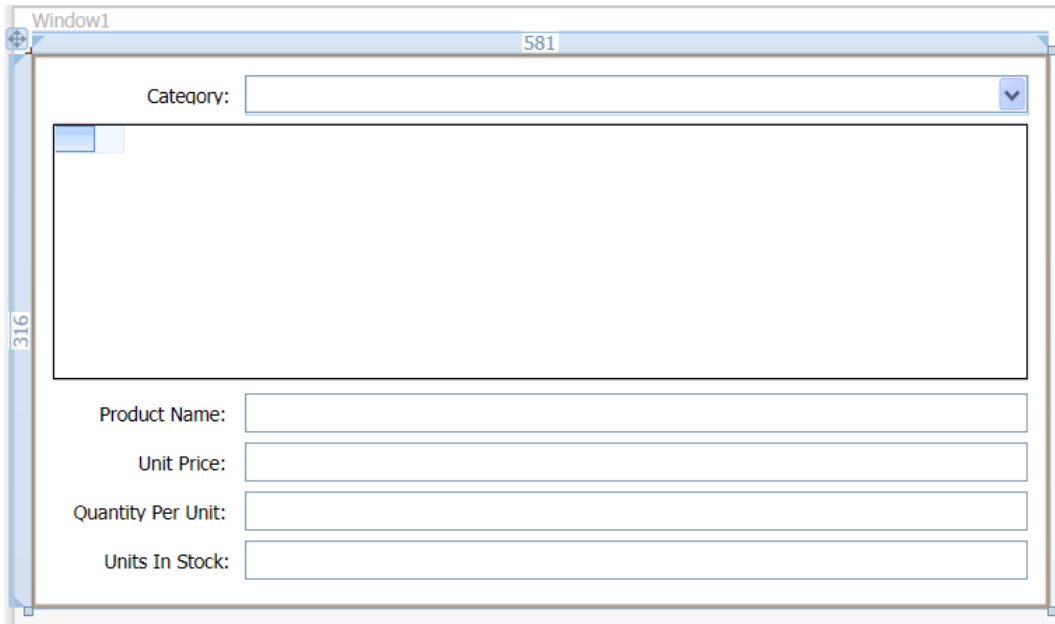
Either way, you can use **LiveLinq** views as binding sources for the controls on the form.

LiveLinq implementation in WPF

In this section, we will create another version of the same application, this time using WPF.

Here are the steps:

1. Create a new WPF 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 a reference to a grid control such as the **C1.WPF.C1DataGrid**, which you can download from www.componentone.com.
5. Add the controls to the main window: one **ComboBox**, one **C1DataGrid**, four **TextBox** controls, and a few **Label** controls. Adjust the control layout so it looks like the previous version of our application, similar to the image below:



Now right-click the window, select **View Code**, and add the following code to the project:

```

using System;
using System.Linq;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using Cl.LiveLinq;
using Cl.LiveLinq.AdoNet;
using Cl.LiveLinq.LiveViews;

namespace LiveLinqWPF
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class Window1 : Window
    {
        public Window1()
        {
            // designer-generated code
            InitializeComponent();

            // get 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
                {
                    CategoryID = c.CategoryID,
                    CategoryName = c.CategoryName,
                    Products = g
                };

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

```

The code is identical to the version we wrote earlier for the WinForms version of the application. It calls **GetData** to load the SQL data into a **DataSet**, then creates a **LiveLinq** view that exposes the data to the application, and finally calls the **DataBind** method to bind the controls to the **LiveLinq** view.

The **GetData** method is also identical to the one we used in the WinForms version of the application:

```

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

```

The **DataBind** method is similar to the one we wrote earlier, but it is not identical. The WPF data binding mechanism is slightly different from the one in WinForms, and that is reflected here:

```

void DataBind(System.Collections.IEnumerable dataSource)
{
    // show categories in ComboBox
    comboBox1.ItemsSource = dataSource;
    comboBox1.DisplayMemberPath = "CategoryName";
    comboBox1.SelectedIndex = 0;

    // show products in ClDataGrid
    var b = new Binding("SelectedValue.Value.Products");
    b.Source = comboBox1;
    clDataGrid1.SetBinding(Cl.WPF.ClDataGrid.ClDataGrid.ItemsSourceProperty, b);

    // show product details in TextBox controls
    BindControl(textBox1, "ProductName");
    BindControl(textBox2, "UnitPrice");
    BindControl(textBox3, "QuantityPerUnit");
    BindControl(textBox4, "UnitsInStock");
}

void BindControl(TextBox txt, string dataMember)
{
    var b = new Binding("SelectedGridItem.Row.DataItem." + dataMember);
    b.Source = clDataGrid1;
    txt.SetBinding(TextBox.TextProperty, b);
}

```

The WPF version of the **DataBind** method starts by assigning our live view to the **ItemsSource** property of the **ComboBox** control. It also sets the **DisplayMemberPath** property of the **ComboBox** to “CategoryName”, which is the field we want to show in the **ComboBox**.

Next, we use the **SetBinding** method to bind the grid’s **ItemsSource** property to the **ComboBox** selection. The string “SelectedValue.Value.Products” selects the “Products” field of the item that is currently selected in the **ComboBox**.

Finally, we use **SetBinding** to bind the **Text** property of each **TextBox** to the corresponding field on the grid selection. This time, we use strings like “SelectedGridItem.Row.DataItem.ProductName” which select specific properties of the product that is currently selected on the grid.

That concludes the WPF version of the application. If you run it now, you will see that it behaves exactly like the WinForms versions. Select categories from the top **ComboBox**, see the corresponding products on the grid below, and the product details in the **TextBoxes** at the bottom of the window.

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:

```
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:

```

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:

```

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.

```

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:

```

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 screenshot shows a Windows application window titled "Form1" with three data grids. The top grid, "Orders", lists individual items with columns for CategoryName, ProductName, UnitPrice, Quantity, and Discount. The middle grid, "Sales by Category", shows the total sales for each category. The bottom grid, "Sales by Product", shows the total sales for each product.

CategoryName	ProductName	UnitPrice	Quantity	Discount
Beverages	Chai	14.4000	45	0.2
Beverages	Chai	14.4000	18	0
Beverages	Chai	14.4000	20	0
Beverages	Chai	14.4000	15	0.15

CategoryName	TotalSales
Beverages	267868.180...
Dairy Produ...	234507.285...
Confections	167357.225...
Meat/Poultry	163022.359...

ProductName	CategoryName	TotalSales
Côte de Blaye	Beverages	141396.735...
Thüringer R...	Meat/Poultry	80368.6720...
Raclette Co...	Dairy Produ...	71155.7000...
Tarte au su...	Confections	47234.9700...

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:

Form1

Orders

CategoryName	ProductName	UnitPrice	Quantity	Discount
Beverages	Côte de Blaye	210.8000	0	0.05
Beverages	Côte de Blaye	210.8000	0	0
Beverages	Côte de Blaye	263.5000	25	0
Beverages	Côte de Blaye	263.5000	60	0

Sales by Category

CategoryName	TotalSales
Dairy Products	234507.285000
Beverages	232622.420000
Confections	167357.225000
Meat/Poultry	163022.359500

Sales by Product

ProductName	CategoryName	TotalSales
Côte de Blaye	Beverages	106150.975...
Thüringer R...	Meat/Poultry	80368.6720...
Raclette Co...	Dairy Produ...	71155.7000...
Tarte au su...	Confections	47234.9700...

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.