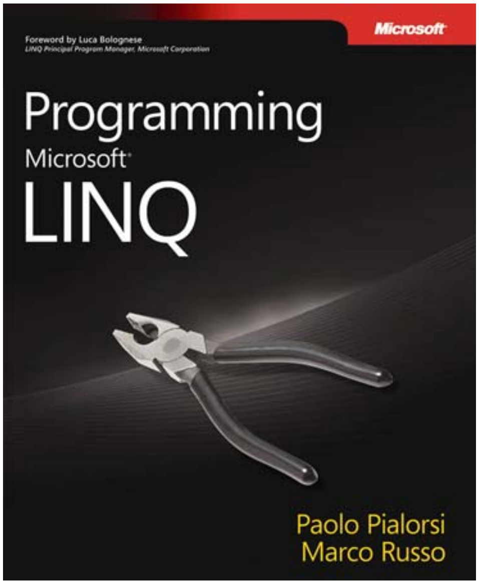

Foreword by Luca Bolognese
LINQ Principal Program Manager, Microsoft Corporation
Microsoft
Programming
Microsoft
LINQ
Paolo Pialorsi
Marco Russo

Programming Microsoft® LINQ

Paolo Pialorsi
Marco Russo

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/10827.aspx

9780735624009

Microsoft Press

Table of Contents

Part I LINQ FOUNDATIONS

2 LINQ Syntax Fundamentals 23

Part II LINQ to Relational Data

Part III LINQ and XML

Part IV Advanced LINQ

Part V Applied LINQ

Part VI Appendixes

Chapter 16

LINQ and ASP.NET

ASP.NET 3.5 is one of the areas of the .NET Framework in which LINQ has been deeply embedded. In this chapter, we will focus on the new features available in ASP.NET 3.5 targeting LINQ—in particular, for LINQ to SQL and LINQ to Objects.

ASP.NET 3.5

In this section, we will cover some of the new features and capabilities available in ASP.NET 3.5 and Microsoft Visual Studio 2008. These features are not all related to LINQ itself, but we will use them during the remaining parts of this chapter. If you are already familiar with ASP.NET 3.5, you can skip or read quickly through this section.

ASP.NET 3.5 extends the ASP.NET 2.0 control and class framework by providing new controls such as *ListView* and *DataPager* and a set of new AJAX-enabling controls. Other improvements involve the IDE of Visual Studio 2008, with new HTML and cascading style sheets (CSS) designers, improved Microsoft IntelliSense, and the capability to design nested master pages.

ListView

ListView is a new control that takes the place of the *DataGrid*, *GridView*, *DataList*, and *Repeater* controls. It allows full control over the data template used to render data binding contents, including the container. This new control allows you to define data binding editing, insertion, deletion, selection, paging, and sorting with full control over the HTML generated. It should be used in place of many of the other controls available prior to ASP.NET 3.5. In fact, using *ListView* you can bind items as you do with *DataGrid*, *GridView*, *DataList*, or *Repeater*, or you can define a freer layout by declaring a set of user-defined HTML/ASPX templates.

The *ListView* control allows you to define a rich set of templates. *LayoutTemplate* and *ItemTemplate* are mandatory, while all the others are optional. Let's start with an ASPX page used to render a set of Northwind customers, shown in Listing 16-1.

Listing 16-1 A sample ASPX page using a *ListView* control instance.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Listing16-1.aspx.cs"
    Inherits="Listing16_1" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Listing 16-1</title>
</head>
```

```
<body>
    <form id="form1" runat="server">
    <div>
    <asp:ListView ID="customersList" runat="server"
        DataSourceID="customersDataSource">
        <LayoutTemplate>
            <ul>
                <asp:PlaceHolder ID="itemPlaceholder" runat="server" />
            </ul>
        </LayoutTemplate>
        <ItemTemplate>
            <li>
                <b><asp:Label runat="server" Text='<%# Eval("CustomerId") %>' /></b> -
                <asp:Label runat="server" Text='<%# Eval("ContactName") %>' /><br />
                <asp:Label runat="server" Text='<%# Eval("CompanyName") %>' />
                [<asp:Label runat="server" Text='<%# Eval("Country") %>' />]
            </li>
        </ItemTemplate>
    </asp:ListView>
    <asp:SqlDataSource ID="customersDataSource" runat="server"
    ConnectionString="<%$ ConnectionStrings:NorthwindConnectionString %>"
    SelectCommand="SELECT [CustomerID], [CompanyName], [ContactName], [Country]
                   FROM [Customers]" />
        </div>
    </form>
</body>
</html>
```

As you can see, we have defined an *<asp:ListView />* element, bound to a *SqlDataSource* control, with a couple of templates in it. The first one, named *LayoutTemplate*, declares how to render the full layout of the container of data items. It could be a TABLE, DIV, SPAN, UL, OL, or any other element you want to use to wrap your set of data. Any data-bound content you are going to render through the *ListView* is rendered as defined in the *ItemTemplate* element. As you can see, the *LayoutTemplate* contains an *<asp:PlaceHolder />* control with an ID *"itemPlaceholder"*. This control will be used as the placeholder for each data-bound item to render. The value of *"itemPlaceholder"* for the ID of the placeholder can be configured, but the key point here is that *ItemTemplate* defines how to render each data-bound item, placing the result of its rendering inside the *LayoutTemplate* result, where a placeholder control has been defined.

The following is the full list of the templates available while defining a *ListView* control:

- *ItemSeparatorTemplate*: As its name implies, this template defines the content to render between data-bound items.
- *AlternatingItemTemplate*: Defines alternate content to distinguish between consecutive items.
- *SelectedItemTemplate*: Defines how to render a selected item to differentiate it from the others.

- *EditItemTemplate*: Defines the content to render in place of the *ItemTemplate*, for an item that has editing status.
- *InsertItemTemplate*: Defines the content to render to insert a new item. By configuring the *InsertItemPosition* property of the *ListView* control, you can decide to render the *InsertItemTemplate* at the top or at the bottom of the rendered list.
- *GroupTemplate*: Defines the content to render to wrap a set of *ItemTemplate* or *EmptyItemTemplate* items. This template is useful for defining output such as multicolumn lists, where output is grouped in rows made of columns. The *GroupTemplate* defines how to wrap a set of data-bound items, where the number of items for each group is defined by the property *GroupItemCount*.
- *GroupSeparatorTemplate*: Defines the content to render between each group of data-bound items.
- *EmptyItemTemplate*: Rendered whenever there is an empty item to render in a group. Imagine a situation in which you have to render a set of items grouped in rows of three columns each, but your data source is not arranged in multiples of three. You will end your data-bound list with one or two empty columns. This template declares how to render those empty columns.
- *EmptyDataTemplate*: Rendered when the data source is empty.

All the template elements in the preceding list use the common and well-known ASP.NET data-binding expressions to define which fields or properties to display.

In Listing 16-2, you can see an example of a *ListView* control rendering a list of items in a grid with multiple rows and columns.

Listing 16-2 A sample ASPX page using a *ListView* control instance to render a list with multiple rows and columns.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Listing16-2.aspx.cs"
    Inherits="Listing16_2" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
    <asp:ListView ID="customersList" runat="server"
        ItemPlaceholderID="dataItemPlaceholder"
        GroupPlaceholderID="groupPlaceholder"
        DataSourceID="customersDataSource" GroupItemCount="5">
        <LayoutTemplate>
            <table cellpadding="2" cellspacing="3">
                <asp:PlaceHolder ID="groupPlaceholder" runat="server" />
            </table>
        </LayoutTemplate>
        <GroupTemplate>
            <tr>
                <asp:PlaceHolder ID="dataItemPlaceholder" runat="server" />
            </tr>
        </GroupTemplate>
        <ItemTemplate>
            <td align="center" style="background-color: LightGreen;">
                <b><asp:Label ID="Label1" runat="server"
                              Text='<%# Eval("CustomerId") %>' /></b> -
                <asp:Label ID="Label2" runat="server"
                              Text='<%# Eval("ContactName") %>' /><br />
                <asp:Label ID="Label3" runat="server"
                              Text='<%# Eval("CompanyName") %>' />
                [<asp:Label ID="Label4" runat="server"
                               Text='<%# Eval("Country") %>' />]
            </td>
        </ItemTemplate>
        <EmptyItemTemplate>
            <td align="center" style="background-color: LightGreen;">&nbsp;</td>
        </EmptyItemTemplate>
    </asp:ListView>
    <asp:SqlDataSource ID="customersDataSource" runat="server"
    ConnectionString="<%$ ConnectionStrings:NorthwindConnectionString %>"
    SelectCommand="SELECT [CustomerID], [CompanyName], [ContactName],  [Country]
                   FROM [Customers]" />
    </div>
    </form>
</body>
</html>
```

In Listing 16-2, you can see how to declare a custom value for the ID of the element to use as the data-bound item placeholder, using the *ItemPlaceholderID* attribute. One more thing to notice in Listing 16-2 is the attribute *GroupingPlaceholderID*, which is used to declare the ID of the placeholder describing each group of data items.

ListView Data Binding

The *ListView* control can be bound to any data source control available in ASP.NET, including the *LinqDataSource* control, which we will cover in detail later in this chapter, or any data source that implements the *IEnumerable* interface.

To bind *ListView* to a data source control, you simply need to assign the ID of the data source control to the *DataSourceID* property of the *ListView* instance. You can see this in Listing 16-1 and Listing 16-2, where we bind a *ListView* control to a *SqlDataSource* control instance.

To bind *ListView* to a data source that implements the *IEnumerable* interface, you need to programmatically set the *DataSource* property, referencing the enumeration of items and invoking the *DataBind* method of the control instance.

In Listing 16-3, you can see a sample ASPX page using this technique, while in Listing 16-4 you can see the corresponding C# code, which sets the *DataSource* property and invokes the *DataBind* method.

Listing 16-3 A sample ASPX page excerpt using a *ListView* control bound to a data source by user code.

```
<asp:ListView ID="customersList" runat="server">
    <LayoutTemplate>
        <ul>
            <asp:PlaceHolder ID="itemPlaceholder" runat="server" />
        </ul>
    </LayoutTemplate>
    <ItemTemplate>
        <li>
            <b><asp:Label ID="Label1" runat="server"
                          Text='<%# Eval("CustomerId") %>' /></b> -
            <asp:Label ID="Label2" runat="server"
                          Text='<%# Eval("ContactName") %>' /><br />
            <asp:Label ID="Label3" runat="server"
                          Text='<%# Eval("CompanyName") %>' />
            [<asp:Label ID="Label4" runat="server"
                           Text='<%# Eval("Country") %>' />]
        </li>
    </ItemTemplate>
</asp:ListView>
```

Listing 16-4 The code-behind class of the ASPX page shown in Listing 16-3.

```
public partial class Listing16_3 : System.Web.UI.Page {
    protected void Page_Load(object sender, EventArgs e) {
        if (!this.IsPostBack) {
            NorthwindDataContext dc = new NorthwindDataContext();
            customersList.DataSource = dc.Customers;
            customersList.DataBind();
        }
    }
}
```

One interesting feature of the *ListView* control is its capability, while it is bound to a data source, to manage a set of data keys to identify each data-bound item. This is useful for data sources where each item has multiple values for its primary key or identifier. To leverage this feature, you can set the *DataKeyNames* property of the *ListView* instance with an array of *String* variables, representing the fields that are part of the item identifier (the record primary key fields in the case of a data base record set). Later in your code, you can get the identifier of each data-bound item from the *ListView* control instance, retrieving the value from the read-only indexer property *DataKeys* of the *ListView* or retrieving the value of the *SelectedDataKey* property. In Listing 16-5, you can see the code-behind class of a sample page that uses this feature to extract the identifier of the item currently selected by the user by leveraging the *ListView* control's selection events. The data source of this sample is the Northwind's Order Details table, where each order detail is identified by the *OrderID* and *ProductID* fields.

Listing 16-5 The code-behind class of the ASPX page using the *DataKeyNames* and *DataKeys* properties.

```
public partial class Listing16_6 : System.Web.UI.Page {
    protected void orderDetailsList_SelectedIndexChanged(Object sender, EventArgs e) {
        Int32? selectedItemOrderID =
                   ((Int32?)orderDetailsList.SelectedDataKey.Values[0]) ?? null;
        Int32? selectedItemProductID =
                   ((Int32?)orderDetailsList.SelectedDataKey.Values[1]) ?? null;

        selectedItemId.Text = String.Format("Selected item ID: {0} - {1}",
            selectedItemOrderID,
            selectedItemProductID);
    }
}
```

In Listing 16-6, you can see the ASPX page corresponding to the code shown in Listing 16-5.

Listing 16-6 The ASPX code of the page using the *DataKeyNames* property.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Listing16-6.aspx.cs"
    Inherits="Listing16_6" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
    <asp:ListView ID="orderDetailsList" runat="server"
        DataKeyNames="OrderID, ProductID"
        DataSourceID="orderDetailsDataSource"
        OnSelectedIndexChanged="orderDetailsList_SelectedIndexChanged">
        <LayoutTemplate>
            <table border="1" cellpadding="2" cellspacing="3">
                <tr>
                    <th style="text-align: center;">Action Command</th>
                    <th style="text-align: center;">OrderID + ProductID</th>
                    <th style="text-align: center;">UnitPrice</th>
                    <th style="text-align: center;">Quantity</th>
                    <th style="text-align: center;">Total</th>
                </tr>
                <asp:PlaceHolder ID="itemPlaceholder" runat="server" />
            </table>
        </LayoutTemplate>
        <ItemTemplate>
            <tr>
```

```
                <td style="text-align: center;">
                    <asp:Button CommandName="Select"
                    Text="Select" runat="server" />
                </td>
                <td style="text-align: center;">
                    <b><asp:Label runat="server"
                    Text='<%# Eval("OrderID") %>' /> -
                    <asp:Label runat="server"
                    Text='<%# Eval("ProductID") %>' /></b><br />
                </td>
                <td style="text-align: center;">
                    <asp:Label runat="server"
                    Text='<%# Eval("UnitPrice", "{0:c}") %>' />
                </td>
                <td style="text-align: center;">
                    <asp:Label runat="server"
                    Text='<%# Eval("Quantity") %>' />
                </td>
                <td style="text-align: right;">
                    <asp:Label runat="server"
                    Text='<%# Eval("Total", "{0:c}") %>' />
                </td>
            </tr>
        </ItemTemplate>
        <SelectedItemTemplate>
            <tr style="background-color: LightGreen;">
                <td style="text-align: center;">
                    <asp:Button ID="Button1" CommandName="Select"
                        Text="Select" runat="server" />
                </td>
                <td style="text-align: center;">
                    <b><asp:Label ID="Label1" runat="server"
                    Text='<%# Eval("OrderID") %>' /> -
                    <asp:Label ID="Label2" runat="server"
                    Text='<%# Eval("ProductID") %>' /></b><br />
                </td>
                <td style="text-align: center;">
                    <asp:Label ID="Label3" runat="server"
                    Text='<%# Eval("UnitPrice", "{0:c}") %>' />
                </td>
                <td style="text-align: center;">
                    <asp:Label ID="Label4" runat="server"
                    Text='<%# Eval("Quantity") %>' />
                </td>
                <td style="text-align: right;">
                    <asp:Label ID="Label5" runat="server"
                    Text='<%# Eval("Total", "{0:c}") %>' />
                </td>
            </tr>
        </SelectedItemTemplate>
    </asp:ListView>

    <asp:SqlDataSource ID="orderDetailsDataSource" runat="server"
        ConnectionString="<%$ ConnectionStrings:NorthwindConnectionString %>"
        SelectCommand="SELECT TOP 20 [OrderID], [ProductID], [UnitPrice],
        [Quantity], [UnitPrice] * [Quantity] AS [Total]
        FROM [Order Details]" />
    </div>
    <div>
       <asp:Label ID="selectedItemId" runat="server" />
    </div>
    </form>
</body>
</html>
```

DataPager

One more new control available in ASP.NET 3.5 is *DataPager*, which is useful for automating paging tasks in data-bound controls. This new control can be applied only to controls implementing the *IPageableItemContainer* interface, like the new *ListView* control we just described. This interface provides a couple of properties—one to set the current page size (*MaximumRows*) and one to set the starting record index (*StartRowIndex*)—one method (*SetPageProperties*) to set those properties, and an event (*TotalRowCountAvailable*) to notify you of the availability of that information as well as the total number of rows. In Listing 16-7, you can see the definition of the *IPageableItemContainer* interface.

Listing 16-7 The *IPageableItemContainer* interface definition.

```
public interface IPageableItemContainer {
    // Events
    event EventHandler<PageEventArgs> TotalRowCountAvailable;

    // Methods
    void SetPageProperties(int startRowIndex, int maximumRows, bool databind);

    // Properties
    int MaximumRows { get; }
    int StartRowIndex { get; }
}
```

The *DataPager* control uses this interface to talk with its paged control.

> **Note** The *DataPager* control by itself does not paginate the data source; instead, it informs the bound control to paginate it. In the case of smart data sources, such as the *LinqDataSource*, which we will cover later in this chapter, the result is efficient and the query extracts exactly the required fields. However, with controls not designed or configured for paging, the paging task occurs in memory—thus, eventually creating performance issues on large sets of data.

To map a *DataPager* instance to a data-binding control, you can set the *PagedControlID* property of the pager control or, in the case of a *ListView* control, you can define the *DataPager* element inside the *LayoutTemplate* of the paged control. In Listing 16-8, you can see an ASPX page using a *DataPager* instance to paginate a list of records bound to a *ListView* control.

Listing 16-8 A sample ASPX page using a *ListView* control paged by a *DataPager* control.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Listing16-8.aspx.cs"
    Inherits="Listing16_8" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
    <asp:ListView ID="customersList" runat="server"
        DataSourceID="customersDataSource">
        <LayoutTemplate>
            <table cellpadding="2" cellspacing="3">
                <tr>
                    <th style="text-align: center">CustomerId</th>
                    <th style="text-align: center">CompanyName</th>
                    <th style="text-align: center">ContactName</th>
                    <th style="text-align: center">Country</th>
                </tr>
                <asp:PlaceHolder ID="itemPlaceholder" runat="server" />
            </table>
        </LayoutTemplate>
        <ItemTemplate>
            <tr>
                <td style="text-align: center">
                    <asp:Label runat="server" Text='<%# Eval("CustomerId") %>' />
                </td>
                <td style="text-align: center">
                    <asp:Label runat="server" Text='<%# Eval("CompanyName") %>' />
                </td>
                <td style="text-align: center">
                    <asp:Label runat="server" Text='<%# Eval("ContactName") %>' />
                </td>
                <td style="text-align: center">
                    <asp:Label runat="server" Text='<%# Eval("Country") %>' />
                </td>
            </tr>
        </ItemTemplate>
    </asp:ListView>

     <asp:DataPager ID="customersListPager" runat="server"
        PagedControlID="customersList" PageSize="5">
        <Fields>
            <asp:NumericPagerField ButtonCount="20" />
        </Fields>
    </asp:DataPager>
    <asp:SqlDataSource ID="customersDataSource" runat="server"
        ConnectionString="<%$ ConnectionStrings:NorthwindConnectionString %>"
        SelectCommand="SELECT [CustomerID], [CompanyName], [ContactName],
        [Country] FROM [Customers]" />

    </div>
    </form>
</body>
</html>
```

As you can see from Listing 16-8, the *DataPager* control provides a *PageSize* property to define the number of records to show for each page. Under the covers, this property sets the *MaximumRows* property of the *DataPager* itself and also for the paged control, using the corresponding property of the *IPageableItemContainer* interface.

The pager control renders the paging UI using a set of fields that can be defined within the *DataPager* element. These fields describe what kind of paging buttons to provide to the user. For instance, there is a *NextPreviousPagerField* that describes the appearance of the buttons to be used to move to the next page, previous page, first page, and last page. This field also enables you to define the CSS style, the images used, and the visibility of the paging buttons. There is also a *NumericPagerField* that describes the appearance, CSS style, and visibility of the numeric paging buttons. In Listing 16-9, you can see an example of using *DataPager* with a custom set of fields defined.

Listing 16-9 A sample ASPX page showing an excerpt of a *DataPager* control with configured pager fields.

```
<asp:DataPager ID="customersListPager" runat="server"
    PagedControlID="customersList" PageSize="5">
    <Fields>
        <asp:NextPreviousPagerField ButtonType="Link"
            FirstPageText="&lt;&lt;" PreviousPageText="&lt;"
            ShowFirstPageButton="true" ShowLastPageButton="false"
            ShowPreviousPageButton="true" ShowNextPageButton="false"  />
        <asp:NumericPagerField ButtonCount="20" ButtonType="Link" />
        <asp:NextPreviousPagerField ButtonType="Link"
            LastPageText="&gt;&gt;" NextPageText="&gt;"
            ShowFirstPageButton="false" ShowLastPageButton="true"
            ShowPreviousPageButton="false" ShowNextPageButton="true" />
    </Fields>
</asp:DataPager>
```

If you need to customize the paging fields to a great extent, you can define a *TemplatePagerField*, which enables you to freely define the layout of the paging UI. In such a situation, the template will be able to manage the properties of the *DataPager* to be aware of its properties—such as *StartRowIndex*, *PageSize*, and *TotalRowCount*—so that it can generate the appropriate HTML output. In Listing 16-10, you can see an example of using a custom field template to render pages in a drop-down list.

Listing 16-10 A sample ASPX page excerpt using a *DataPager* control with a custom template for pager fields.

```
<asp:DataPager ID="customersListPager" runat="server"
    PagedControlID="customersList" PageSize="5">
    <Fields>
      <asp:TemplatePagerField>
        <PagerTemplate>
            Go to page:
            <uc:PagerDropDownList runat="server" AutoPostBack="true"
                TotalPageCount="<%# Math.Ceiling
                    ((double)Container.TotalRowCount / Container.PageSize) %>" />
        </PagerTemplate>
      </asp:TemplatePagerField>
    </Fields>
</asp:DataPager>
```

In the code in Listing 16-10, the *<uc:PagerDropDownList />* element refers to a custom user control that we defined to handle page numbers that are rendered inside a common drop-down list. In Listing 16-11, you can see the source code of this custom user control.

Listing 16-11 The code of a custom user control to render page numbers in a drop-down list.

```
public class PagerDropDownList : DropDownList {

    public Int32 TotalPageCount { get; set; }

    protected override void OnPreRender(EventArgs e) {
        base.OnPreRender(e);

        for (Int32 c = 1; c <= TotalPageCount; c++) {
            this.Items.Add(new ListItem(c.ToString("00")));
        }

        DataPager pager = ((DataPagerFieldItem)this.Parent).Pager;
        Int32 currentPageIndex = (Int32)Math.Ceiling(
            (double)pager.StartRowIndex / pager.PageSize);

        this.Items[currentPageIndex].Selected = true;
    }

    protected override void OnSelectedIndexChanged(EventArgs e) {
        base.OnSelectedIndexChanged(e);

        Int32 currentPageIndex = Int32.Parse(this.SelectedValue);
        DataPager pager = ((DataPagerFieldItem)this.Parent).Pager;
        pager.SetPageProperties((currentPageIndex - 1) * pager.MaximumRows,
            pager.MaximumRows, true);
    }
}
```

Note that all the pager fields inherit from the abstract base class *DataPagerField*; thus, you can define your own fields if you need to.

One last thing to note about the *DataPager* control is that it provides you with the capability to use a querystring field to receive the current page number to show. By default, the *DataPager* uses HTTP POST commands to navigate through the pages. By the way, sometimes it can be useful to make paging available through querystring parameters. Think about a Web site publishing products grouped by pages. With HTTP POST–based paging, which is the default for *DataPager*, search engines crawl just the first page of each set of products. Using an HTTP GET–based rendering pattern allows for the indexing of contents, based on different URLs for each single page, thereby improving the efficiency of content crawling. To leverage this feature, you need to set the value of the *QueryStringField* property of the *DataPager* control. In the case of a null or empty string value for this property, the *DataPager* control uses HTTP POST; otherwise, it looks for a querystring field declaring the page to show. The name of this querystring field will be the value assigned to the *QueryStringField* property of the *DataPager* instance. In the case of an ASPX page with multiple data sources, data bound controls, and *DataPager* controls, you should set different values of this property for each *DataPager* instance; otherwise, unexpected paging behavior could occur. Consider also that using a querystring-based paging technique forces the *DataPager* fields to render paging buttons as *HyperLink* controls, ignoring any explicit setting of the *ButtonType* property of the fields. In Listing 16-12, you can see an example of using the *QueryStringField* property.

Listing 16-12 A sample ASPX page excerpt with a *DataPager* control using querystring paging.

```
<asp:DataPager ID="customersListPager" runat="server"
    PagedControlID="customersList" PageSize="5"
    QueryStringField="page">
    <Fields>
        <asp:NextPreviousPagerField ButtonType="Link"
            FirstPageText="&lt;&lt;" PreviousPageText="&lt;"
            ShowFirstPageButton="true" ShowLastPageButton="false"
            ShowPreviousPageButton="true" ShowNextPageButton="false"  />
        <asp:NumericPagerField ButtonCount="20" ButtonType="Link" />
        <asp:NextPreviousPagerField ButtonType="Link"
            LastPageText="&gt;&gt;" NextPageText="&gt;"
            ShowFirstPageButton="false" ShowLastPageButton="true"
            ShowPreviousPageButton="false" ShowNextPageButton="true" />
    </Fields>
</asp:DataPager>
```

Note If you want to delve deeper into ASP.NET development techniques, we suggest you read the great book *Essential ASP.NET 2.0* by Fritz Onion with Keith Brown (Addison-Wesley Professional, 2006). To improve your knowledge about ASP.NET 3.5 in particular, we suggest you read the book *Microsoft ASP.NET 3.5 Developer Reference* by Dino Esposito (Microsoft Press, 2008).

LinqDataSource

The biggest news concerning ASP.NET 3.5 from a LINQ point of view is the *LinqDataSource* control. This new control implements the *DataSource* control pattern and allows you to use LINQ to SQL or LINQ to Objects to data bind any bindable rendering control using an ASPX declarative approach. For instance, you can use the *LinqDataSource* control to bind a *DataGrid*, *GridView*, *Repeater*, *DataList*, or *ListView* control. This control resembles the *SqlDataSource* and *ObjectDataSource* from previous versions of ASP.NET, but it targets a LINQ to SQL data model or a custom set of entities using LINQ to Objects.

If you have ever used the *SqlDataSource* and *ObjectDataSource* controls, you probably know that you have to define custom queries or methods, respectively, to provide data querying, inserting, editing, and deleting features. With the *LinqDataSource* control, you can leverage the out-of-the-box LINQ environment to get all of these functionalities, without having to write specific code. Of course, you can always decide to customize this behavior, but it is important to know that it is your choice and it is not mandatory.

For instance, imagine that you have the list of customers from the Northwind database. In Figure 16-1, you can see the complete LINQ to SQL data model schema.

The DBML file can be directly added to the *App_Code* folder of the ASP.NET Web project, or it can be defined in a separate and dedicated class library project. In the case of a sample prototypal solution or a simple Web application, you can declare the data model within the Web project itself. If you are developing an enterprise solution, it is better to divide the layers of your architecture into dedicated projects, thus defining a specific class library for the data model definition.

Note For an enterprise-level solution, read Chapter 15, “LINQ in a Multitier Solution,” which discusses how to deal with architectural and design matters related to LINQ adoption.

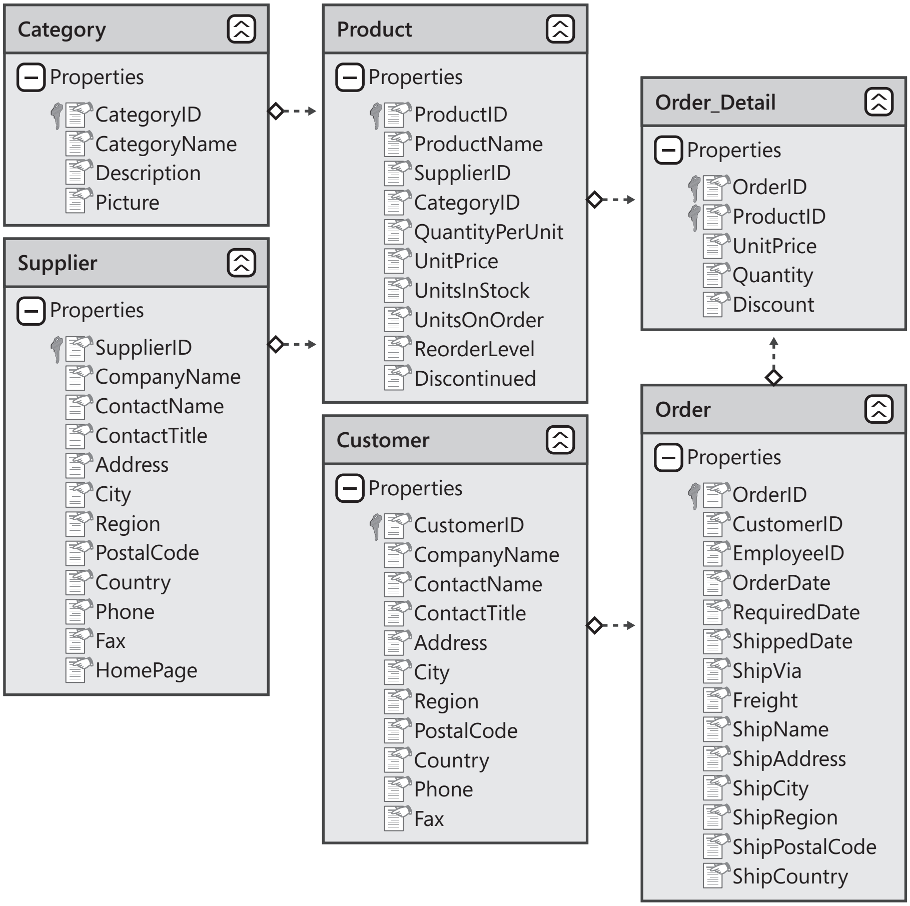


Figure 16-1 The LINQ to SQL Database Model (DBML) schema used in the samples of this chapter

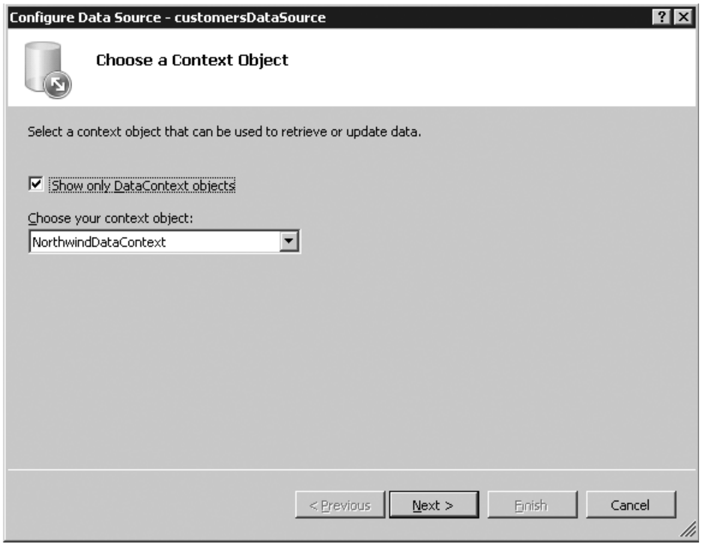
Consider the ASPX page shown in Listing 16-13, where we retrieve a set of customers and bind it to a *GridView* control. As you can see, the *LinqDataSource* control declares a *ContextTypeName* property that maps to the previously defined *NorthwindDataContext* type, and a *TableName* property that corresponds to the *Customers* property of the *DataContext* instance, representing a class of type *Table<Customer>* of LINQ to SQL.

Listing 16-13 A sample ASPX page using a *LinqDataSource* control to render the list of Northwind customers into a *GridView* control.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Listing16-13.aspx.cs"
    Inherits="Listing16_13" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Listing 16-13</title>
</head>
```

```
<body>
    <form id="form1" runat="server">
    <div>
        <asp:GridView runat="server" DataSourceID="customersDataSource" />
        <asp:LinqDataSource ID="customersDataSource" runat="server"
            ContextTypeName="NorthwindDataContext"
            TableName="Customers" />
    </div>
    </form>
</body>
</html>
```

Using Microsoft Visual Studio 2008, you can define these and many other properties using the graphical page designer. In Figure 16-2 and Figure 16-3, you can see the steps for graphically configuring a *LinqDataSource* control by using a designer wizard. To start the wizard you simply need to insert a *LinqDataSource* control into an ASPX page, click on the control task menu, and choose the Configure Data Source task activity. In the first step, presented in the Choose A Context Object screen, you can define the main source for the control.



Configure Data Source - customersDataSource

Choose a Context Object

Select a context object that can be used to retrieve or update data.

☑ Show only DataContext objects

Choose your context object: NorthwindDataContext

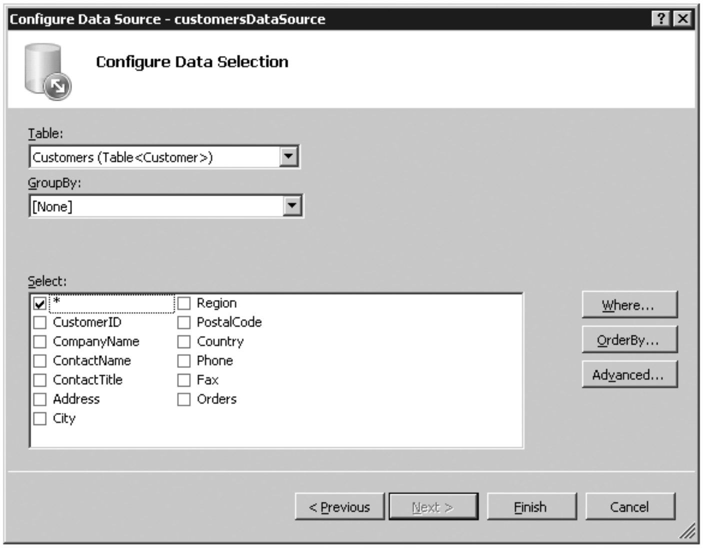
Figure 16-2 The Choose A Context Object screen of the *LinqDataSource* control configuration designer

If you keep the Show Only DataContext Objects option selected, the drop-down list will show only objects inheriting from the LINQ to SQL *DataContext* type, like the *NorthwindDataContext* shown in Figure 16-1. If you clear that option, the designer shows you any .NET type available in the project or in the user-defined projects referenced, including custom entities that can eventually be queried by using LINQ to Objects or any instance of *IEnumerable<T>*.

Note When referencing user-defined class libraries, remember that the *LinqDataSource* designer works with compiled assemblies. Therefore, you need to compile your solution to access newly created or modified types.

In this last case, the drop-down list will also show all the entity types defined in the LINQ to SQL data model, not only the *DataContext* type. This is useful to query a property of type *EntitySet*<*T*> offered by a specific LINQ to SQL entity—for example, the set of orders of a specific customer instance in a master-detail rendering page.

After clicking the Next button in the designer wizard, you are presented with the Configure Data Selection configuration panel, shown in Figure 16-3.



Configure Data Source - customersDataSource

Configure Data Selection

Table: Customers (Table<Customer>)

GroupBy: [None]

Select:

☑ *
☐ CustomerID
☐ CompanyName
☐ ContactName
☐ ContactTitle
☐ Address
☐ City
☐ Region
☐ PostalCode
☐ Country
☐ Phone
☐ Fax
☐ Orders

Figure 16-3 The Configure Data Selection screen of the *LinqDataSource* control configuration designer

Here you can define the whole set of configuration parameters of the *LinqDataSource* control. The Table drop-down list allows you to select the *Table*<*T*> to query, in the case of a *DataContext*, or the *IEnumerable* property of a custom type to query with LINQ to Objects. The GroupBy drop-down list is used to define a grouping rule, eventually with an inner ordering for each group. The Select section provides a list that allows you to choose the fields or properties to select while querying the data source. By default, the *LinqDataSource* control projects all the fields (*), but you can define a custom projection predicate. There are also a set of buttons to define selection rules (the Where button) and ordering conditions (the OrderBy button). Through those buttons, you can define static filtering and ordering rules, but you can also define dynamic parameters, mapping their values to other controls, cookie values, form input elements, ASP.NET *Profile* variables, querystring parameters, or ASP.NET *Session* variables. This behavior is exactly the same as all the other *DataSource* controls, such as *SqlDataSource* and *ObjectDataSource*.

There is one last command, named Advanced, that determines whether you allow automatic inserts, deletes, or updates. By default, the *LinqDataSource* control simply selects data in a read-only manner. Keep in mind that automatic data modification is allowed only when the following conditions are satisfied: the projection returns all the data source items (having *Select* of "*" enabled), there are no grouping rules, the Context Object is set to a class inheriting from *DataContext,* and consequently the queried collection is of type *Table<T>*. If any of these conditions is not true, a custom LINQ query will occur, and the projected results will be an *IEnumerable<T>*, where *T* is an anonymous type, which is a read-only type by design.

> **Note** Think carefully about the previous conditions. Whenever you need to query an updatable set of items using the automatic engine of the *LinqDataSource* control, you need to query its full set of fields/properties (SELECT * FROM ...) even if you need to change a few of them. It is a behavior that is acceptable in very simple solutions with only one full table mapped directly to the UI, but in more complex and common solutions it is better to use custom selection and data updating rules.

After configuring the *LinqDataSource* control, you are ready to bind it to any bindable control, as we did in Listing 16-13.

All the parameters available through the UI designer can be defined in markup inside the ASPX source code of the page. In fact, the result of using the designer affects the markup definition of the *LinqDataSource* control instance. In Listing 16-14, you can see an example of a *LinqDataSource* control bound to the list of *CompanyName*, *ContactName*, and *Country* for Northwind's customers. The list is filtered by *Country*, mapped to a drop-down list with automatic post-back enabled, and ordered by *ContactName* value.

Listing 16-14 A sample ASPX page using a *LinqDataSource* control to render the list of Northwind's customers into a *GridView* control with some filtering and ordering rules.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Listing16-14.aspx.cs"
    Inherits="Listing16_14" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Listing 16-14</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <br />Country:&nbsp;
        <asp:DropDownList ID="ddlCountries" runat="server"
            AutoPostBack="True" DataSourceID="countriesDataSource"
            DataTextField="Country" DataValueField="Country" />
        <br />
```

```
            <asp:GridView ID="customersGrid" runat="server"
                DataSourceID="customersDataSource" AutoGenerateColumns="False">
                <Columns>
                    <asp:BoundField DataField="CompanyName"
                        HeaderText="CompanyName" ReadOnly="True"
                        SortExpression="CompanyName" />
                    <asp:BoundField DataField="ContactName"
                        HeaderText="ContactName" ReadOnly="True"
                        SortExpression="ContactName" />
                    <asp:BoundField DataField="Country"
                        HeaderText="Country" ReadOnly="True"
                        SortExpression="Country" />
                </Columns>
            </asp:GridView>
            <asp:LinqDataSource ID="customersDataSource" runat="server"
                ContextTypeName="NorthwindDataContext"
                Select="new (CompanyName, ContactName, Country)"
                TableName="Customers" Where="Country == @Country">
                <WhereParameters>
                    <asp:ControlParameter ControlID="ddlCountries"
                        Name="Country" PropertyName="SelectedValue"
                        Type="String" />
                </WhereParameters>
            </asp:LinqDataSource>
            <br />
            <asp:LinqDataSource ID="countriesDataSource" runat="server"
                ContextTypeName="NorthwindDataContext" GroupBy="Country"
                OrderGroupsBy="key" Select="new (key as Country)"
                TableName="Customers">
            </asp:LinqDataSource>
        </div>
        </form>
    </body>
    </html>
```

Under the covers of this page, the *LinqDataSource* control converts its configuration to a dynamic query expression executed against the source context.

Paging Data with *LinqDataSource* and *DataPager*

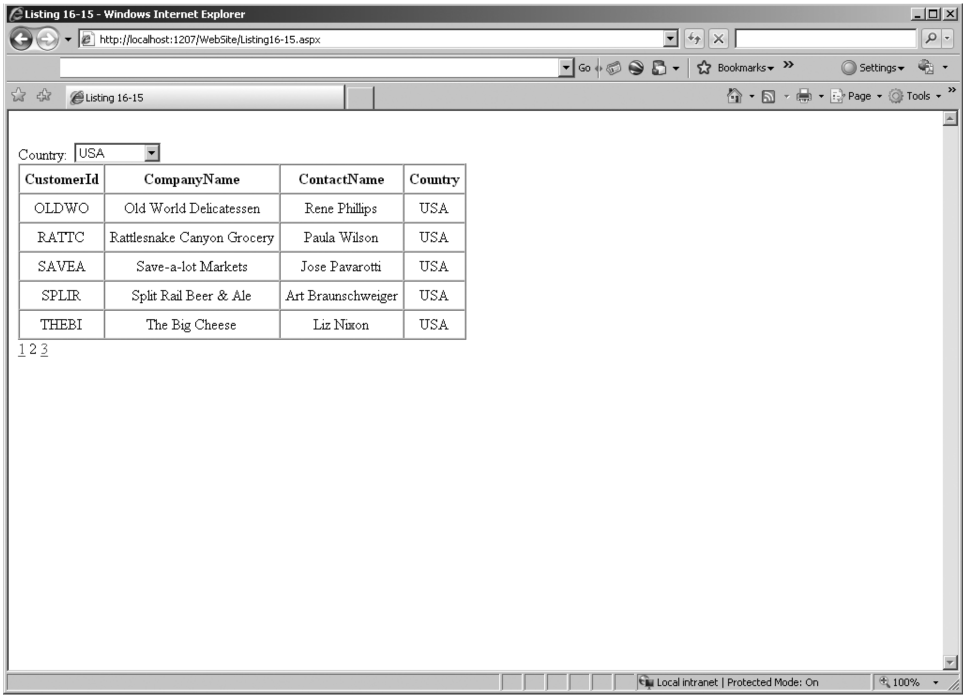
If you define a *DataPager* control that is applied to a rendering control such as a *ListView* and mapped to a *LinqDataSource* control instance, you will be able to paginate the data source at the level of the LINQ query. In fact, the *LinqDataSource* control will be configured to select a maximum number of records (*MaximumRows*) starting from a start record index (*StartRowIndex*), depending on the *DataPager* configuration. Internally, the *LinqDataSource* control will translate these parameters into a query ending with a *.Skip(StartRowIndex).Take(MaximumRows)* expression.

In Listing 16-15, you can see a sample ASPX page querying the list of Northwind's customers, filterable by country using a drop-down list, and paged with a page size of five customers per page.

Listing 16-15 A sample ASPX page using a *LinqDataSource* control to render the list of Northwind's customers into a *ListView* control with filtering and paging through a *DataPager* control.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Listing16-15.aspx.cs"
    Inherits="Listing16_15" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Listing 16-15</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <br />Country:&nbsp;
        <asp:DropDownList ID="ddlCountries" runat="server"
            AutoPostBack="True" DataSourceID="countriesDataSource"
            DataTextField="Country" DataValueField="Country" />
        <br />
        <asp:ListView ID="customersList" runat="server"
            DataSourceID="customersDataSource">
            <LayoutTemplate>
                <table cellpadding="5" cellspacing="0" border="1">
                    <tr>
                        <th style="text-align: center">CustomerId</th>
                        <th style="text-align: center">CompanyName</th>
                        <th style="text-align: center">ContactName</th>
                        <th style="text-align: center">Country</th>
                    </tr>
                    <asp:PlaceHolder ID="itemPlaceholder" runat="server" />
                </table>
            </LayoutTemplate>
            <ItemTemplate>
                <tr>
                    <td style="text-align: center">
                        <asp:Label ID="Label1" runat="server"
                            Text='<%# Eval("CustomerId") %>' />
                    </td>
                    <td style="text-align: center">
                        <asp:Label ID="Label2" runat="server"
                            Text='<%# Eval("CompanyName") %>' />
                    </td>
                    <td style="text-align: center">
                        <asp:Label ID="Label3" runat="server"
                            Text='<%# Eval("ContactName") %>' />
                    </td>
                    <td style="text-align: center">
                        <asp:Label ID="Label4" runat="server"
                            Text='<%# Eval("Country") %>' />
                    </td>
                </tr>
            </ItemTemplate>
        </asp:ListView>
        <asp:LinqDataSource ID="customersDataSource" runat="server"
            ContextTypeName="NorthwindDataContext"
            Select="new (CustomerID, CompanyName, ContactName, Country)"
            TableName="Customers" Where="Country == @Country">
            <WhereParameters>
                <asp:ControlParameter ControlID="ddlCountries"
                    Name="Country" PropertyName="SelectedValue"
                    Type="String" />
            </WhereParameters>
        </asp:LinqDataSource>
        <asp:DataPager ID="customersPager" PagedControlID="customersList"
            runat="server" PageSize="5">
            <Fields>
                <asp:NumericPagerField ButtonCount="5" ButtonType="Link" />
            </Fields>
        </asp:DataPager>
        <br />
        <asp:LinqDataSource ID="countriesDataSource" runat="server"
            ContextTypeName="NorthwindDataContext" GroupBy="Country"
            OrderGroupsBy="key" Select="new (key as Country)"
            TableName="Customers">
        </asp:LinqDataSource>
    </div>
    </form>
</body>
</html>
```

In Figure 16-4, you can see the HTML output of the page shown in Listing 16-15.


Listing 16-15 - Windows Internet Explorer
http://localhost:1207/WebSite/Listing16-15.aspx
Go
Bookmarks
Settings
Listing 16-15
Page
Tools
Local intranet | Protected Mode: On
100%

Country: USA

CustomerId	CompanyName	ContactName	Country
OLDWO	Old World Delicatessen	Rene Phillips	USA
RATTC	Rattlesnake Canyon Grocery	Paula Wilson	USA
SAVEA	Save-a-lot Markets	Jose Pavarotti	USA
SPLIR	Split Rail Beer & Ale	Art Braunschweiger	USA
THEBI	The Big Cheese	Liz Nixon	USA

1 2 3

Figure 16-4 The HTML output of the page shown in Listing 16-15.

You can take a closer look at this behavior by subscribing to the *Selecting* event offered by each *LinqDataSource* control instance. This event allows you to inspect the actual selecting, filtering, ordering, grouping, and paging parameters of the executing query. Within the event code, you can also personalize the values of these parameters by customizing the query result before its execution. In Listing 16-16, you can see an example of using the *Selecting* event to define a custom filtering parameter. The *Arguments* property of the *LinqDataSourceSelectEventArgs* instance, received by the event handler, describes the paging configuration.

Note The technique used for pagination by combining a *DataPager* and a *LinqDataSource* is a good approach from an ASP.NET point of view because it requests only the data displayed in one page from the data source. However, if the query is made on a large table without filters in a relational database such as a SQL Server database and the user selects the last page, the resulting query requires a complete scan of the table. Even with an existing index corresponding to the desired order, this operation consumes resources of the database server and time to complete.

Additionally, if the query is the result of complex filters on a large table, each movement in the pages of the *DataPager* control could still require long execution times, because the query is executed every time the user requests a new page. In scenarios where the table size is large and/or the filter operation requires a particular amount of time, you might need to use a more complex architecture with temporary data caching of the results of such a queries.

Listing 16-16 An example of using the *Selecting* event of a *LinqDataSource* control.

```
public partial class Listing16_16 : System.Web.UI.Page {
    protected void customersDataSource_Selecting(Object sender,
        LinqDataSourceSelectEventArgs e) {
        // Forces sorting by ContactName
        e.Arguments.SortExpression = "ContactName";
    }
}
```

Handling Data Modifications with *LinqDataSource*

Whenever you need to modify data through a *LinqDataSource* instance, you need to configure it with a type inherited from *DataContext* as its *ContextTypeName*, a *TableName* corresponding to the name of a property of type *Table<T>* belonging to the *DataContext* type, a null or empty value for the *Select* property, and a null value for the *GroupBy* property. You also need to set to true the *EnableDelete*, *EnableInsert*, or *EnableUpdate* flags to support deletion, insertion, or modification of data items, respectively. As you have already seen, you can enable these flags using the designer interface or the ASPX markup. Regardless of the way in which you configure the *LinqDataSource* instance, after enabling data modifications you will be able to change the data source by using a data-bound control that supports editing, such as *GridView*, *ListView*, or *DetailsView*. In Listing 16-17, you can see an example of an ASPX page that uses an editable *GridView* to show the list of Northwind's customers.

Listing 16-17 A sample ASPX page excerpt using an editable *LinqDataSource* control.

```
<form id="form1" runat="server">
    <div>
        <asp:GridView ID="customersGrid" runat="server"
            DataKeyNames="CustomerID" DataSourceID="customersDataSource"
            AutoGenerateEditButton="true" AutoGenerateColumns="true" />

        <asp:LinqDataSource ID="customersDataSource" runat="server"
            ContextTypeName="NorthwindDataContext" TableName="Customers"
            OrderBy="ContactName" EnableUpdate="true" />

    </div>
</form>
```

If you need to modify or examine the values of the fields before effectively executing data modification tasks, you can subscribe to the events *Deleting*, *Inserting*, *Selecting*, and *Updating*. When you want to examine field values after data modification, you can handle the corresponding post-events, such as *Deleted*, *Inserted*, *Selected*, and *Updated*. Keep in mind that if you want to specify a default value for empty fields, during modification of the data source you can add parameters to the *InsertParameters*, *UpdateParameters*, and *DeleteParameters* of the control.

The events occurring before the data modification receive event arguments specific to each modification operation. For instance, a *Selecting* event handler receives a *LinqDataSourceSelectEventArgs* object, allowing you to customize the query parameters as well as the result, as you will see in the next section.

A *Deleting* event handler receives an argument of type *LinqDataSourceDeleteEventArgs*. This type provides an *OriginalObject* property that contains the data item that will be deleted. It also provides a property named *Exception*, of type *LinqDataSourceValidationException*, that describes any exception related to data validation that occurred within the entity before its deletion. If you want to handle such an exception by yourself and do not want to throw it again, you can set to true the *ExceptionHandled* Boolean property of the event argument.

An *Inserting* event handler receives an argument of type *LinqDataSourceInsertEventArgs*. This type provides the data item that is going to be inserted as the value of its *NewObject* property, while the *Exception* and *ExceptionHandled* properties work just like those for the *Deleting* event argument, obviously referencing validation exceptions related to data item insertion.

An *Updating* event handler receives an event argument of type *LinqDataSourceUpdateEventArgs*, describing the original and actual state of the data item through the *OriginalObject* and *NewObject* properties, respectively. Again, the handling of any kind of validation exception is based on the *Exception* property and *ExceptionHandled* flag.

Maybe you are wondering how the *LinqDataSource* control keeps track of the original values of entities between page postbacks. Under the covers, the *LinqDataSource* control saves each entity field's original value into the page *ViewState*, except for fields that are marked as

UpdateCheck.Never in the data model. For more details about *UpdateCheck* configuration, refer to Chapter 6, "Tools for LINQ to SQL." Using this technique, the *LinqDataSource* control is able to transparently handle data concurrency checks.

In Listing 16-18, you can see an example of code validating the updating of customer information, before any real data modification occurs.

Listing 16-18 The code-behind class of a page handling a custom validation rule while updating customer information through an editable *LinqDataSource* control.

```
protected void customersDataSource_Updating(object sender,
    LinqDataSourceUpdateEventArgs e) {
    if (((Customer)e.OriginalObject).Country != (((Customer)e.NewObject).Country)) {
            e.Cancel = true;
    }
}
```

The interesting part of this approach is that all the plumbing is handled transparently by the *LinqDataSource* control. Later in this chapter, you will see how to manually handle data selections based on custom LINQ queries.

While each pre-event handler receives a specific event argument type instance, all the post-event handlers receive an event argument of type *LinqDataSourceStatusEventArgs*, which allows you to examine the result of the data modification task. In fact, this event argument type provides the resulting data item in its *Result* property in the case of a successful modification. In the case of data modification failure, the post-event argument will have a null value for the *Result* property and will provide the exception that occurred in its *Exception* property. As with pre-events, you can set an *ExceptionHandled* Boolean property of the event argument to not throw the exception again. In Listing 16-19, you can see an excerpt of code handling a concurrency exception while updating a data item.

Listing 16-19 Sample code handling a concurrency exception while editing a data item through a *LinqDataSource* control.

```
protected void customersDataSource_Updated(object sender,
    LinqDataSourceStatusEventArgs e) {
    if ((e.Result == null) && (e.Exception != null)) {
        if (e.Exception is ChangeConflictException) {
            // Handle data concurrency issue
            // TBD ...

            // Stop exception bubbling
            e.ExceptionHandled = true;
        }
    }
}
```

There are three final events that are useful while handling the *LinqDataSource* control: *ContextCreating*, *ContextCreated*, and *ContextDisposing*. The first two occur when the *DataContext* is going to be created or has been created, respectively. When *ContextCreating* occurs, you can specify a custom *DataContext* instance, setting the *ObjectInstance* property of the event argument of type *LinqDataSourceContextEventArgs* you receive. If you ignore this event, the *LinqDataSource* control will create a *DataContext* by itself, based on the type name provided through the *ContextTypeName* property. You can use this event if you want to customize the *DataContext* creation—for instance, to provide a custom connection string, a user-defined *SqlConnection* instance, or a custom *MappingSource* definition. The *ContextCreated* event, on the other hand, allows you to customize the *DataContext* instance already created. It receives a *LinqDataSourceStatusEventArgs*, like post-event data modification events, but in this case the *Result* property of the event argument contains the *DataContext* type instance. In the case of errors while creating the *DataContext*, you will find an exception in the *Exception* property of the event argument and a value of null for the *Result* property. Listing 16-20 shows an example of handling the *ContextCreated* event to define a custom data shape for querying data.

Listing 16-20 Sample code customizing the *DataContext* of a *LinqDataSource* control, just after its creation.

```
protected void customersDataSource_ContextCreated(object sender,
    LinqDataSourceStatusEventArgs e) {
    NorthwindDataContext dc = e.Result as NorthwindDataContext;
    if (dc != null) {
        // Instructs the DataContext to load orders of current year with
        // each customer instance
        DataLoadOptions dlo = new DataLoadOptions();
        dlo.AssociateWith<Customer>(c => c.Orders
            .Where(o => o.OrderDate.Value.Year == DateTime.Now.Year));

        dc.LoadOptions = dlo;
    }
}
```

The *ContextDisposing* event is useful for handling custom or manual disposing of the *DataContext* type instance and receives an event argument of type *LinqDataSourceDisposeEventArgs*. This event occurs during the *Unload* event of the *LinqDataSource* control and provides the *DataContext* that is going to be disposed of in the *ObjectInstance* property of the event argument.

Note Because the *LinqDataSource* control is hosted by an ASP.NET environment, internally the *DataContext* used to query the data source is created and released for each single request with a pattern similar to the one shown later in this chapter and in Chapter 18, "LINQ and the Windows Communication Foundation." In the case of an uneditable *LinqDataSource* control, the *ObjectTrackingEnabled* property of the *DataContext* instance is set to false; otherwise, it is left to its default value of true.

Using Custom Selections with *LinqDataSource*

Sometimes you need to select data by using custom rules or custom user-defined stored procedures. Whenever you are using a *LinqDataSource* in these situations, you can subscribe to the *Selecting* event of the data source control. The *Selecting* event handler receives a *LinqDataSourceSelectEventArgs* instance that can be used to change the selection parameters, as you saw in the previous section. However, it can also be used to return a completely customized selection by setting the *Result* property of the event argument to a custom value. In Listing 16-21, you can see an example of code handling the *Selecting* event to return the result of a custom stored procedure instead of a standard LINQ to SQL query.

Listing 16-21 The code-behind class of a page handling a custom selection pattern for a *LinqDataSource* control using a stored procedure.

```
protected void customersOrdersDataSource_Selecting(object sender,
    LinqDataSourceSelectEventArgs e) {
    NorthwindDataContext dc = new NorthwindDataContext();
    e.Result = dc.CustOrdersOrders("ALFKI");
}
```

In the case of an editable *LinqDataSource* control, the custom selection code should return a set of items with the same type *T* referenced by the *Table<T>* type of the *TableName* property of the data source control. The code to select the custom result could be any code returning a set of items consistent with the configured *TableName*; thus, you can use a stored procedure as well as customized explicit LINQ to SQL queries. Listing 16-22 shows a *Selecting* event implementation that uses a custom LINQ query to select a customized data source.

Listing 16-22 The code-behind class of a page handling a custom selection pattern for a *LinqDataSource* control using an explicit LINQ query.

```
protected void customersDataSource_Selecting(object sender,
    LinqDataSourceSelectEventArgs e) {
    NorthwindDataContext dc = new NorthwindDataContext();

    e.Result =
        from   c in dc.Customers
        where  c.Country == "USA" && c.Orders.Count > 0
        select c;
}
```

One more thing to notice is that in the case of data paging, the custom result will be paged too. This paging occurs because the *Result* property of the *Selecting* event argument will be managed by the *LinqDataSource* engine anyway, applying any paging rule to it. In the case of an *IQueryable<T>* result, as in Listing 16-22, the paging will occur on the LINQ query expression tree, preserving performance and efficiency.

When you programmatically set a custom result during the *Selecting* event, the *ContextCreated* event of the *LinqDataSource* control is not raised.

Using *LinqDataSource* with Custom Types

A common usage of the *LinqDataSource* control is querying data available through LINQ to SQL, but it can also be used to query custom types and entities. If you define a *ContextTypeName* that does not correspond to a type inherited from the LINQ to SQL *DataContext*, the *LinqDataSource* will switch from LINQ to SQL to LINQ to Objects queries. Consider the code in Listing 16-23, which describes user-defined *Customer* and *Order* types and is not related to a LINQ to SQL data model.

Listing 16-23 User-defined *Customer* and *Order* entities not related to a LINQ to SQL data model.

```
public class Customer {
    public Int32 CustomerID { get; set; }
    public String FullName { get; set; }
    public List<Order> Orders { get; set; }
}

public class Order {
    public Int32 OrderID { get; set; }
    public Int32 CustomerID { get; set; }
    public Decimal EuroAmount { get; set; }
}
```

Now consider a *CustomerManager* class offering a property named *Customers* and of type *List<T>*, where *T* is a *Customer* type, as shown in Listing 16-24.

Listing 16-24 A *CustomerManager* type providing a property of type *List<Customer>*.

```
public class CustomerManager {

    public CustomerManager() {
        this.Customers = new List<Customer> {
            new Customer { CustomerID = 1, FullName = "Paolo Pialorsi",
              Orders = new List<Order> {
                new Order { OrderID = 1, CustomerID = 1, EuroAmount = 100},
                new Order { OrderID = 2, CustomerID = 1, EuroAmount = 200}
              }
            },
            new Customer { CustomerID = 2, FullName = "Marco Russo",
              Orders = new List<Order> {
                new Order { OrderID = 3, CustomerID = 2, EuroAmount = 150},
                new Order { OrderID = 4, CustomerID = 2, EuroAmount = 250},
                new Order { OrderID = 5, CustomerID = 2, EuroAmount = 130},
                new Order { OrderID = 6, CustomerID = 2, EuroAmount = 220}
              }
            },
            new Customer { CustomerID = 3, FullName = "Andrea Pialorsi",
              Orders = new List<Order> {
                new Order { OrderID = 7, CustomerID = 3, EuroAmount = 900},
                new Order { OrderID = 8, CustomerID = 3, EuroAmount = 2500}
              }
            }
        };
    }

    public List<Customer> Customers { get; set; }
}
```

You can use the *CustomerManager* type as the value for the *ContextTypeName* of a *LinqDataSource* control, while the "Customers" string could be the value for the *TableName* property of the *LinqDataSource* control. Internally, the data source control queries the collection of items returned from the *CustomerManager* class, allowing your code to query a custom set of entities instead of a LINQ to SQL data model. Nevertheless, keep in mind that using LINQ to Objects instead of LINQ to SQL requires you to load into memory the whole set of data—before any filtering, sorting, or paging task. Be careful using this technique because it could be very expensive for your CPU and memory if your code is processing a large amount of data or many page requests. In Listing 16-25, you can see an ASPX page using this kind of configuration.

Listing 16-25 A sample ASPX page using a *LinqDataSource* control linked to a set of user-defined entities.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Listing16-25.aspx.cs"
    Inherits="Listing16_25" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Listing 16-25</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

        <asp:GridView ID="customersGrid" runat="server"
            DataSourceID="customersDataSource">
        </asp:GridView>

        <asp:LinqDataSource
            ID="customersDataSource" runat="server"
            ContextTypeName="DevLeap.Linq.Web.DataModel.CustomerManager"
            TableName="Customers" />

    </div>
    </form>
</body>
</html>
```

Binding to LINQ queries

At this point, we have worked with LINQ using the new *LinqDataSource* control. However, in ASP.NET you can bind a bindable control to any kind of data source that implements the *IEnumerable* interface. Every LINQ query, when it is enumerated, provides a result of type *IEnumerable<T>*, which internally is also an *IEnumerable.* Therefore, any LINQ query can be used as an explicit data source in user code. In Listing 16-26, you can see an example of a page using a LINQ query in its *Page_Load* event to bind a custom list of Northwind products to a *GridView* control.

Listing 16-26 A sample page code based on a user-explicit LINQ query in the *Page_Load* event.

```
public partial class Listing16_26 : System.Web.UI.Page {
    protected void Page_Load(object sender, EventArgs e) {
        NorthwindDataContext dc = new NorthwindDataContext();

        var query =
          from   c in dc.Customers
          where  c.Country == "USA" && c.Orders.Count > 0
          select new {
            c.CustomerID, c.ContactName,
            c.CompanyName, c.Country,
            OrdersCount = c.Orders.Count };

        customersGrid.DataSource = query;
        customersGrid.DataBind();
    }
}
```

These considerations enable you to use the complete LINQ query syntax, particularly LINQ to SQL and LINQ to XML, to query custom data shapes and many different content types, binding the results to an ASP.NET control. For instance, Listing 16-27 shows the code-behind class of an ASPX page that renders the list of posts from one blog via LINQ to XML applied to the blog RSS feed.

Listing 16-27 The code-behind class of a page reading an RSS feed using a LINQ to XML query.

```
public partial class Listing16_27 : System.Web.UI.Page {
    protected void Page_Load(object sender, EventArgs e) {
        XElement feed = XElement.Load(
            "http://introducinglinq.com/blogs/MainFeed.aspx");

        var query =
            from   f in feed.Descendants("item")
            select new {
              Title = f.Element("title").Value,
              PubDate = f.Element("pubDate").Value,
              Description = f.Element("description").Value
            };
        blogPostsGrid.DataSource = query;
        blogPostsGrid.DataBind();
    }
}
```

Querying data sources of any kind and shape using the many different flavors of LINQ is a challenging activity. However, the most common use of user-defined explicit LINQ queries is in the field of LINQ to SQL. Sooner or later, almost every Web application will need to query a set of records from a database. What is of most interest occurs when the data needs to be updated. Using the *LinqDataSource* control, we saw that everything is automated and that the data source control handles updates, insertions, deletions, and selections by itself, eventually raising a concurrency exception when data concurrency issues occur.

When querying data manually, you need to take care of many details by yourself. First of all, consider that ASP.NET is an HTTP-based development platform. Therefore, every request you handle can be considered independent from any other, even if it comes from the same user/browser. Given this, you need to keep track of changes applied to your data between multiple subsequent requests, possibly avoiding use of *Session* variables or shared (static) objects, because this can reduce the scalability of your solution. On the other hand, you should not keep an in-memory instance of the *DataContext* used to query a LINQ to SQL data model. You should instead create, use, and dispose of the *DataContext* for each request, as the *LinqDataSource* control does. In Listing 16-28, you can see an example of this technique for explicitly querying the list of Northwind's customers.

Listing 16-28 The code-behind class of a page explicitly querying Northwind's customers via LINQ to SQL.

```
public partial class Listing16_28 : System.Web.UI.Page {
    protected void Page_Load(object sender, EventArgs e) {
        NorthwindDataContext dc = new NorthwindDataContext();

        var query =
          from   c in dc.Customers
          select new {
            c.CustomerID, c.ContactName, c.Country, c.CompanyName};

        customersGrid.DataSource = query;
        customersGrid.DataBind();
    }
}
```

In this sample, we use the query result to bind a *GridView* control, configured to be editable. Listing 16-29 shows the corresponding ASPX page source code.

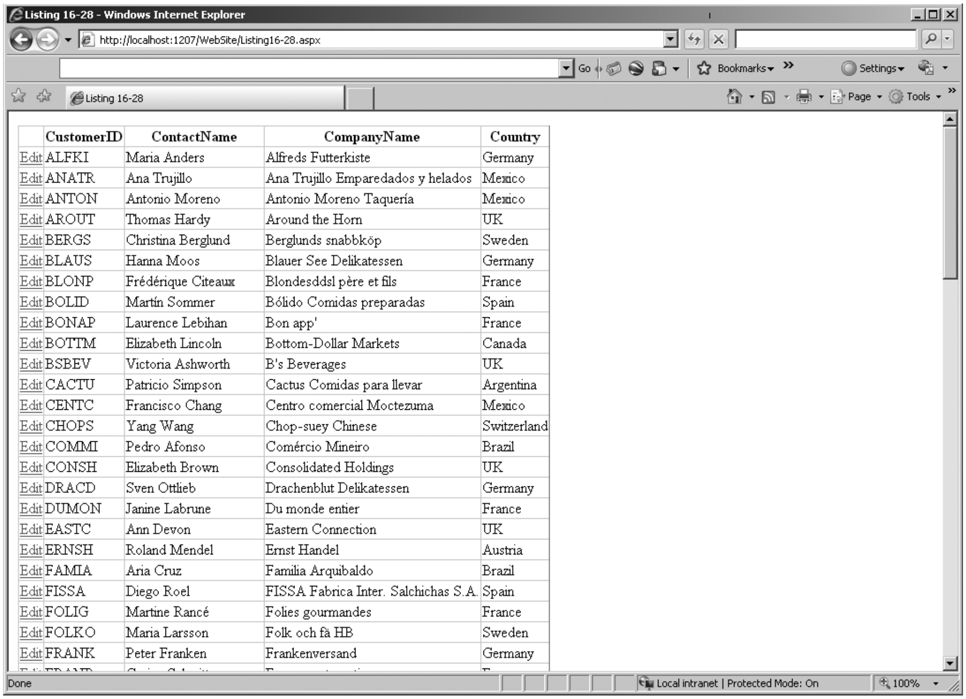
Listing 16-29 The ASPX page code excerpt based on the code-behind class shown in Listing 16-28.

```
<body>
    <form id="form1" runat="server">
    <div>

        <asp:GridView ID="customersGrid" runat="server"
            AutoGenerateEditButton="true"
            AutoGenerateColumns="true" />

    </div>
    </form>
</body>
```

Now here's the interesting part. Imagine that your user decides to change some fields for the currently selected customer, using an editable *GridView* like the one shown in Figure 16-5.


Listing 16-28 - Windows Internet Explorer
http://localhost:1207/WebSite/Listing16-28.aspx
Go
Bookmarks
Settings
Listing 16-28
Page
Tools
Done
Local intranet | Protected Mode: On
100%

	CustomerID	ContactName	CompanyName	Country
Edit	ALFKI	Maria Anders	Alfreds Futterkiste	Germany
Edit	ANATR	Ana Trujillo	Ana Trujillo Emparedados y helados	Mexico
Edit	ANTON	Antonio Moreno	Antonio Moreno Taquería	Mexico
Edit	AROUT	Thomas Hardy	Around the Horn	UK
Edit	BERGS	Christina Berglund	Berglunds snabbköp	Sweden
Edit	BLAUS	Hanna Moos	Blauer See Delikatessen	Germany
Edit	BLONP	Frédérique Citeaux	Blondesddsl père et fils	France
Edit	BOLID	Martín Sommer	Bólido Comidas preparadas	Spain
Edit	BONAP	Laurence Lebihan	Bon app'	France
Edit	BOTTM	Elizabeth Lincoln	Bottom-Dollar Markets	Canada
Edit	BSBEV	Victoria Ashworth	B's Beverages	UK
Edit	CACTU	Patricio Simpson	Cactus Comidas para llevar	Argentina
Edit	CENTC	Francisco Chang	Centro comercial Moctezuma	Mexico
Edit	CHOPS	Yang Wang	Chop-suey Chinese	Switzerland
Edit	COMMI	Pedro Afonso	Comércio Mineiro	Brazil
Edit	CONSH	Elizabeth Brown	Consolidated Holdings	UK
Edit	DRACD	Sven Ottlieb	Drachenblut Delikatessen	Germany
Edit	DUMON	Janine Labrune	Du monde entier	France
Edit	EASTC	Ann Devon	Eastern Connection	UK
Edit	ERNSH	Roland Mendel	Ernst Handel	Austria
Edit	FAMIA	Aria Cruz	Familia Arquibaldo	Brazil
Edit	FISSA	Diego Roel	FISSA Fabrica Inter. Salchichas S.A.	Spain
Edit	FOLIG	Martine Rancé	Folies gourmandes	France
Edit	FOLKO	Maria Larsson	Folk och fä HB	Sweden
Edit	FRANK	Peter Franken	Frankenversand	Germany

Figure 16-5 The HTML output of the page defined by Listings 16-28 and 16-29.

What you get back when the user presses the Update button on the page is the index of the selected/edited item in the *GridView*, as well as the values of the controls rendering the editable row. To update the data source, you need to create a new *DataContext* instance, which will be used during the entire unit of work that handles this single page request, and query the *DataContext* to get the entity corresponding to the data item that is going to be updated.

After you have retrieved the data item entity from the original store, you can explicitly change its properties, handling the user modifications, and then you can submit changes to the persistence layer, using the *SubmitChanges* method of the *DataContext* object. In Listing 16-30, you can see an example of the required code.

Listing 16-30 The code-behind class of a page explicitly updating a Northwind's customer instance with LINQ to SQL.

```
protected void UpdateCustomerInstance(String customerID,
        String contactName, String country, String companyName) {
    NorthwindDataContext dc = new NorthwindDataContext();

    Customer c = dc.Customers.First(c => c.CustomerID == customerID);
    if (c != null) {
        c.ContactName = contactName;
        c.Country = country;
        c.CompanyName = companyName;
    }

    dc.SubmitChanges();
}
```

The preceding example is missing an important part of the process: the concurrency check. In fact, we simply use the input coming from the user to modify the persistence layer. However, we have no guarantees about the exclusiveness of the operation we are performing. When the user decides to update a data item, you can leverage a specific *DataContext* method, called *Attach*. This method allows you to attach an entity to a *DataContext* instance, eventually providing its original state to determine whether any modification happened or simply to notify the *DataContext* that the entity has been changed. If the entity type has an *UpdateCheck* policy defined (see Chapter 6 for further details), the *DataContext* will be able to reconcile the entity with the one actually present in the database. In Listing 16-31, you can see an example of a code-behind class using this technique.

Listing 16-31 The code-behind class of a page updating a Northwind customer instance with LINQ to SQL, tracking the original state of the entity.

```
protected void AttachAndUpdateCustomerInstance(String customerID,
    String contactName, String country, String companyName) {

    NorthwindDataContext dc = new NorthwindDataContext();

    Customer c = new Customer();
    c.CustomerID = customerID;
    c.ContactName = contactName;
    c.Country = country;
    c.CompanyName = companyName;

    // The Boolean flag indicates that the item has been changed
    dc.Customers.Attach(c, true);
    dc.SubmitChanges();
}
```

Summary

This chapter showed you how to leverage the new features and controls available in ASP.NET 3.5 to develop data-enabled Web applications, using LINQ to SQL and LINQ in general. Consider that what you have seen is really useful for rapidly defining Web site prototypes and simple Web solutions. On the other hand, in enterprise-level solutions you will probably need at least one intermediate layer between the ASP.NET presentation layer and the data persistence one, represented by LINQ to SQL. In real enterprise solutions, you usually also need a business layer that abstracts all business logic, security policies, and validation rules from any kind of specific persistence layer. And you will probably have a Model-View-Controller or Model-View-Presenter pattern governing the UI. In this more complex scenario, chances are that the *LinqDataSource* control will be tied to entities collections more often than to LINQ to SQL results.