30

ASP.NET Dynamic Data

WHAT'S IN THIS CHAPTER?

- Building an ASP.NET Dynamic Data application
- Using dynamic data routes
- Handling your application's display

ASP.NET offers a feature that enables you to dynamically create data-driven Web applications. ASP.NET Dynamic Data is more than purely a code generator, and although it can provide a base application, it is completely modifiable by you. This feature allows you to quickly and easily create data entry or applications that allow your end users to work with and view the backend database.

The data-driven applications require a few components in order to work: the template system, a database from which to drive the application, and the ability to work from an object model such as LINQ to SQL or LINQ to Entities.

This chapter illustrates how to build and customize an ASP.NET Dynamic Data application.

CREATING YOUR BASE APPLICATION WITH VISUAL STUDIO 2010

ASP.NET Dynamic Data's capabilities were first introduced with the .NET Framework 3.5 SP1 and have been enhanced with the release of the .NET Framework 4 and Visual Studio 2010. Figure 30-1 shows the two projects that are available in this area from Visual Studio 2010: Dynamic Data Linq to SQL Web Site and Dynamic Data Entities Web Site.

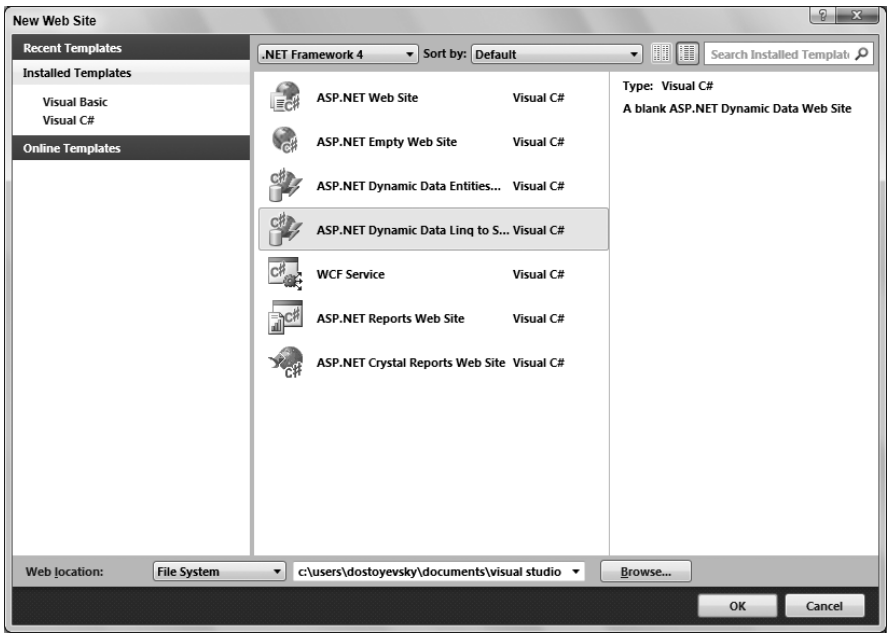


FIGURE 30-1

Which of the two application types you choose depends on how you plan to approach the object model. If you want to work with LINQ to SQL as your object model, then you will choose the Dynamic Data Linq to SQL Web Site option. If you want to work with LINQ to Entities (as discussed in the previous chapter), then you will choose the Dynamic Data Entities Web Site project. Note that if you were to reverse the choices, then you would experience various expected errors in working with your data-driven application.

For an example of working through the creation of your first data-driven application, select the Dynamic Data Linq to SQL Web Site option, as this example will work with LINQ to SQL. Name your project `MyNorthwind`.

Visual Studio 2010 will create a base application that is not connected to any database or object model from the start. It will be your job to make these connections. Before doing this, however, take a look at what Visual Studio has created for you.

Looking at the Core Files Created in the Default Application

Before you even assign the pre-generated dynamic data application to a database, much of the core application is created for you through the Visual Studio process just mentioned.

When you create the application, you will find a lot of new pieces to your ASP.NET application in the Visual Studio Solution Explorer. Figure 30-2 presents these extra items.

The items that are generated for you and what is presented here in the Visual Studio Solution Explorer are generally referred to as *scaffolding*. Even though a lot of code seems to be generated for you, do not be alarmed thinking that you are locked into a specific data-driven application. What is generated is termed "scaffolding" because it is a framework that can be taken holistically or modified and extended for any purpose. This framework is the presentation and database layer support that you will need for the auto-generation of your application. You are in no way locked into a specific set of models, looks and feels, or even an approach that you are unable to modify to suit your specific needs.

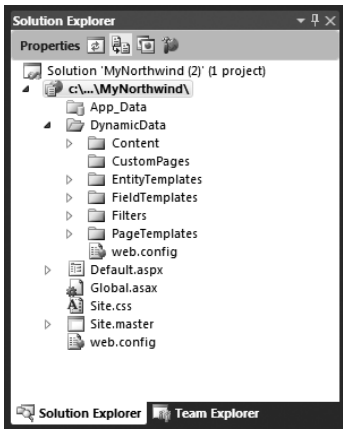


FIGURE 30-2

Even though you will find a lot of pre-generated code in your Solution Explorer, you are not even required to use this code to work with ASP.NET Dynamic Data. In fact, you can even add ASP.NET Dynamic Data to a pre-existing application.

Next, this chapter looks at the pre-generated application that enables you to work with your backend database.

The Dynamic Data Application

One of the biggest additions to this application operation that is dramatically different from the standard ASP.NET application is a folder called DynamicData. This folder contains the pre-generated ASP.NET application that enables you to work with your database through a browser.

The goal of this application is to enable you to work with your database through the entire CRUD process (Create, Read, Update, and Delete). Again, you can limit the amount of interactivity you provide from your application.

To view how this application works against a database, you must dig further into the controls and pages that make up the application. Expanding the DynamicData folder, you find the following folders:

- Content
- CustomPages
- EntityTemplates
- FieldTemplates
- Filters
- PageTemplates

In addition to these folders, you will find a `web.config` file that is specific to this application.

The Content folder in this part of the application includes a user control that is used in the page templates, as well as the underlying images that are used by the style sheet of the application.

The CustomPages folder is a separate folder that allows you to put any custom pages that you might include in the data-driven Web application. When you create an application from scratch, you will not find any file in this folder. It is intentionally blank.

The EntityTemplates folder is a new folder provided in ASP.NET 4 that makes getting the layout you want quite easy, thereby not requiring you to build a custom page. Initially, there is a `Default.ascx` (user control), and the edit and insert versions of this control are found in the folder.

The FieldTemplates folder is interesting because it has some of the more granular aspects of the application. The entire application is designed to work off a database, but it really does not have any idea what type of database it is going to be working from. The FieldTemplates folder is a way that the application can present any of the underlying data types that are coming from the database. Figure 30-3 shows the data types as presented in the Visual Studio Solution Explorer.

In this case, you will find data types for date/time values, integers, text, foreign keys, and more. They are represented as ASP.NET user controls, or `.ascx` files, which makes them easy for you to modify.

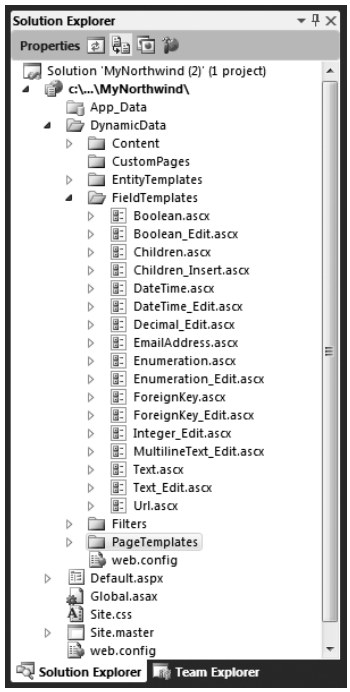


FIGURE 30-3

As an example, open the `DateTime.ascx` file, and you will get the bit of code presented in Listing 30-1.

LISTING 30-1: The DateTime.ascx file

VB

```
<%@ Control Language="VB" CodeBehind="DateTime.ascx.vb"
    Inherits="DateTimeField" %>

<asp:Literal runat="server" ID="Literal1"
 Text="<%# FieldValueString %>" />
```

C#

```
<%@ Control Language="C#" CodeBehind="DateTime.ascx.cs"
    Inherits="DateTimeField" %>

<asp:Literal runat="server" ID="Literal1"
 Text="<%# FieldValueString %>" />
```

You can see that there isn't much to the code shown in Listing 30-1. There is a simple Literal server control and nothing more. The `Text` property of the control gets populated with the variable `FieldValueString` from the code-behind file. Listing 30-2 presents this file.

LISTING 30-2: The code-behind file for DateTime.ascx

VB

```
Imports System.ComponentModel.DataAnnotations
Imports System.Web.DynamicData
Imports System.Web

Class DateTimeField
    Inherits FieldTemplateUserControl


    Public Overrides ReadOnly Property DataControl As Control
        Get
            Return Literal1
        End Get
    End Property


End Class
```

C#

```
using System;
using System.Collections.Specialized;
using System.ComponentModel.DataAnnotations;
using System.Web.DynamicData;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class DateTimeField :
   System.Web.DynamicData.FieldTemplateUserControl
 {
     public override Control DataControl
     {
         get
         {
             return Literal1;
         }
     }
}
```

The data field templates inherit from `System.Web.DynamicData.FieldTemplateUserControl`, as you can see from Listing 30-2. This user control provides a read version of the data field for your application.

In addition to the data field `.ascx` files, there is an *edit* version of the control that is represented as a user control in addition to the *read* version of the data field. The edit versions have a filename ending in `_Edit`, as in the `DateTime_Edit.ascx` file.

Listing 30-3 presents the contents of the `DateTime_Edit.ascx` control.

LISTING 30-3: The DateTime_Edit.ascx file

VB

```
<%@ Control Language="VB" CodeBehind="DateTime_Edit.ascx.vb"
    Inherits="DateTime_EditField" %>

<asp:TextBox ID="TextBox1" runat="server" CssClass="DDTextBox"
 Text='<%# FieldValueEditString %>' Columns="20"></asp:TextBox>

<asp:RequiredFieldValidator runat="server" ID="RequiredFieldValidator1"
 CssClass="DDControl DDValidator" ControlToValidate="TextBox1"
 Display="Dynamic" Enabled="false" />
<asp:RegularExpressionValidator runat="server"
 ID="RegularExpressionValidator1" CssClass="DDControl DDValidator"
 ControlToValidate="TextBox1" Display="Dynamic" Enabled="false" />
<asp:DynamicValidator runat="server" ID="DynamicValidator1"
 CssClass="DDControl DDValidator" ControlToValidate="TextBox1"
 Display="Dynamic" />
<asp:CustomValidator runat="server" ID="DateValidator"
 CssClass="DDControl DDValidator" ControlToValidate="TextBox1"
 Display="Dynamic" EnableClientScript="false" Enabled="false"
 OnServerValidate="DateValidator_ServerValidate" />
```

C#

```
<%@ Control Language="C#" CodeBehind="DateTime_Edit.ascx.cs"
    Inherits="DateTime_EditField" %>

<asp:TextBox ID="TextBox1" runat="server" CssClass="DDTextBox"
 Text='<%# FieldValueEditString %>' Columns="20"></asp:TextBox>

<asp:RequiredFieldValidator runat="server" ID="RequiredFieldValidator1"
 CssClass="DDControl DDValidator" ControlToValidate="TextBox1"
 Display="Dynamic" Enabled="false" />
<asp:RegularExpressionValidator runat="server"
 ID="RegularExpressionValidator1" CssClass="DDControl DDValidator"
 ControlToValidate="TextBox1" Display="Dynamic" Enabled="false" />
<asp:DynamicValidator runat="server" ID="DynamicValidator1"
 CssClass="DDControl DDValidator" ControlToValidate="TextBox1"
 Display="Dynamic" />
<asp:CustomValidator runat="server" ID="DateValidator"
 CssClass="DDControl DDValidator" ControlToValidate="TextBox1"
 Display="Dynamic" EnableClientScript="false" Enabled="false"
 OnServerValidate="DateValidator_ServerValidate" />
```

As you can see, the date will be populated into a TextBox server control. In addition to this representation, a series of validation server controls validate the input before applying an edit to the database.

Listing 30-4 shows the code-behind of these pages.

LISTING 30-4: The code-behind pages from the DateTime_Edit.ascx file

VB

```
Imports System.ComponentModel.DataAnnotations
Imports System.Web.DynamicData
Imports System.Web
```

continues

LISTING 30-4 *(continued)*

```
Class DateTime_EditField
    Inherits FieldTemplateUserControl

    Private Shared DefaultDateAttribute As DataTypeAttribute =
       New DataTypeAttribute(DataType.DateTime)

    Public Overrides ReadOnly Property DataControl As Control
        Get
            Return TextBox1
        End Get
    End Property

    Protected Sub Page_Load(ByVal sender As Object,
     ByVal e As EventArgs)
        TextBox1.ToolTip = Column.Description
        SetUpValidator(RequiredFieldValidator1)
        SetUpValidator(RegularExpressionValidator1)
        SetUpValidator(DynamicValidator1)
        SetUpCustomValidator(DateValidator)
    End Sub

    Private Sub SetUpCustomValidator(ByVal validator As CustomValidator)
        If Column.DataTypeAttribute IsNot Nothing Then
            Select Case (Column.DataTypeAttribute.DataType)
                Case DataType.Date,
                     DataType.DateTime,
                     DataType.Time

                    validator.Enabled = True
                    DateValidator.ErrorMessage =
                     HttpUtility.HtmlEncode(
      Column.DataTypeAttribute.FormatErrorMessage(Column.DisplayName))
            End Select
        ElseIf Column.ColumnType.Equals(GetType(DateTime)) Then
            validator.Enabled = True
            DateValidator.ErrorMessage = HttpUtility.HtmlEncode(
              DefaultDateAttribute.FormatErrorMessage(Column.DisplayName))
        End If
    End Sub

    Protected Sub DateValidator_ServerValidate(ByVal source As Object,
      ByVal args As ServerValidateEventArgs)
        Dim dummyResult As DateTime
        args.IsValid = DateTime.TryParse(args.Value, dummyResult)
    End Sub

    Protected Overrides Sub ExtractValues(ByVal dictionary As
     IOrderedDictionary)
        dictionary(Column.Name) = ConvertEditedValue(TextBox1.Text)
    End Sub

End Class
```

C#

```
<%@ Control Language="C#" CodeFile="DateTime_Edit.ascx.cs"
    Inherits="DateTime_EditField" %>

using System;
using System.Collections.Specialized;
using System.ComponentModel.DataAnnotations;
```

```
using System.Web.DynamicData;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class DateTime_EditField :
   System.Web.DynamicData.FieldTemplateUserControl {
    private static DataTypeAttribute DefaultDateAttribute =
        new DataTypeAttribute(DataType.DateTime);
    protected void Page_Load(object sender, EventArgs e) {
        TextBox1.ToolTip = Column.Description;

        SetUpValidator(RequiredFieldValidator1);
        SetUpValidator(RegularExpressionValidator1);
        SetUpValidator(DynamicValidator1);
        SetUpCustomValidator(DateValidator);
    }

    private void SetUpCustomValidator(CustomValidator validator) {
        if (Column.DataTypeAttribute != null) {
            switch (Column.DataTypeAttribute.DataType) {
                case DataType.Date:
                case DataType.DateTime:
                case DataType.Time:
                    validator.Enabled = true;
                    DateValidator.ErrorMessage =
                      HttpUtility.HtmlEncode(
     Column.DataTypeAttribute.FormatErrorMessage(Column.DisplayName));
                    break;
            }
        }
        else if (Column.ColumnType.Equals(typeof(DateTime))) {
            validator.Enabled = true;
            DateValidator.ErrorMessage = HttpUtility.HtmlEncode(
          DefaultDateAttribute.FormatErrorMessage(Column.DisplayName));
        }
    }

    protected void DateValidator_ServerValidate(object source,
      ServerValidateEventArgs args) {
        DateTime dummyResult;
        args.IsValid = DateTime.TryParse(args.Value, out dummyResult);
    }

    protected override void ExtractValues(IOrderedDictionary
      dictionary) {
        dictionary[Column.Name] = ConvertEditedValue(TextBox1.Text);
    }

    public override Control DataControl {
        get {
            return TextBox1;
        }
    }

}
```

You can see that every time you edit something in the database that includes a `DateTime` data type, it will also appear in a textbox HTML element for that purpose. In the code-behind of the file, you assign a tooltip to the textbox element, and assign four separate validation server controls to the control.

Again, these user controls are the most granular example of how you can modify the output of the application. Another option, which you will review shortly, is to work with the page templates where these granular user controls are used.

The Filters folder is used to create drop-down menus for Booleans (true/false values), foreign keys, and enumerations. These menus enable the end user to filter tables based upon keys within the database. This folder is new for ASP.NET 4.

The PageTemplates folder contains the core pages that you use to bring the application together. Notice that pages exist for many of the core constructs that you will use in representing your tables in the application. The PageTemplates folder includes the following pages:

- `Details.aspx`
- `Edit.aspx`
- `Insert.aspx`
- `List.aspx`
- `ListDetails.aspx`

You use the `List.aspx` page for the tables in your connected database. You use the `Details.aspx` pages when you are examining a single row from the table, and you use the `ListDetails.aspx` page for examining master details views of the table and row relationships. You use the `Edit.aspx` and `Insert.aspx` pages, in turn, for the types of operations that they describe. Listing 30-5 shows a partial listing of the `List.aspx` page, specifically how it represents a table.

LISTING 30-5: A partial code example from the List.aspx page

```
<%@ Page Language="C#" MasterPageFile="~/Site.master"
    CodeBehind="List.aspx.cs" Inherits="List" %>

<%@ Register src="~/DynamicData/Content/GridViewPager.ascx"
      tagname="GridViewPager" tagprefix="asp" %>

<asp:Content ID="Content1" ContentPlaceHolderID="ContentPlaceHolder1"
 Runat="Server">
    <asp:DynamicDataManager ID="DynamicDataManager1" runat="server"
     AutoLoadForeignKeys="true">
        <DataControls>
            <asp:DataControlReference ControlID="GridView1" />
        </DataControls>
    </asp:DynamicDataManager>

    <h2 class="DDSubHeader"><%= table.DisplayName%></h2>

    <asp:UpdatePanel ID="UpdatePanel1" runat="server">
        <ContentTemplate>

           <!-- Code removed for clarity -->

        </ContentTemplate>
    </asp:UpdatePanel>
</asp:Content>
```

Again, these controls and templates do not have any built-in thoughts of what the table is going to be like. These pages are built oblivious to the underlying tables to which this application will connect. This is evident through the use of the code `<%= table.DisplayName%>`. The next step is to actually tie this application to a database.

Incorporating the Database

The next step required to build your first ASP.NET Dynamic Data application is to incorporate a database that you are able to work with. For this example, you need to include the Northwind database. The previous chapter explained how to get a copy of it or the AdventureWorks database, both of which are SQL Server Express Edition databases. Include the `Northwind.mdf` database file in the App_Data folder of your solution for this example.

After the database is in place, the next step is to establish a defined entity data model layer that will work with the underlying database. Because this example is working with the LINQ to SQL approach rather than the LINQ to Entities approach, you must add a LINQ to SQL class to your application.

To accomplish this task, right-click on your project within the Visual Studio Solution Explorer and select Add ➪ New Item from the provided menu. Then add a LINQ to SQL class by selecting that option from the middle section of the Add New Item dialog, as illustrated in Figure 30-4.

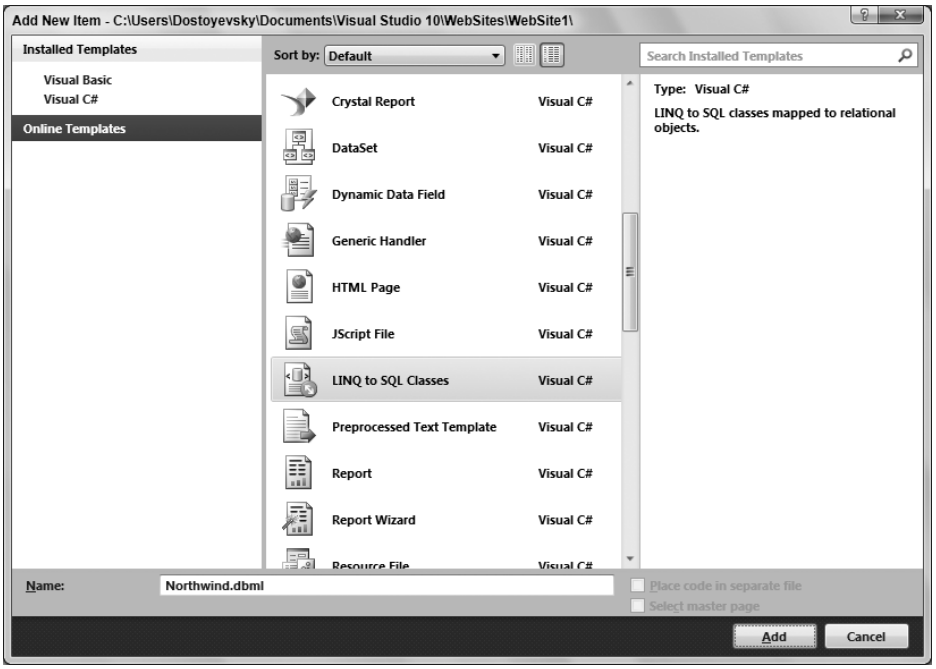


FIGURE 30-4

Name your `.dbml` file `Northwind.dbml`, as shown in Figure 30-4. The O/R (*object relational*) Designer for working with your LINQ to SQL classes then appears. It is a visual representation of the `Northwind.dbml` file (which currently does not contain any references to the database tables).

The O/R Designer has two parts. The first part is for data classes, which can be tables, classes, associations, and inheritances. Dragging such items onto the design surface gives you a visual representation of the object that you can work with. The second part (on the right side of the dialog) is for methods, which map to stored procedures within a database.

Working with the Northwind database is quite simple — it's really a matter of opening the database in the Visual Studio Server Explorer and dragging and dropping all the tables onto the design surface of the O/R Designer. Figure 30-5 shows the results.

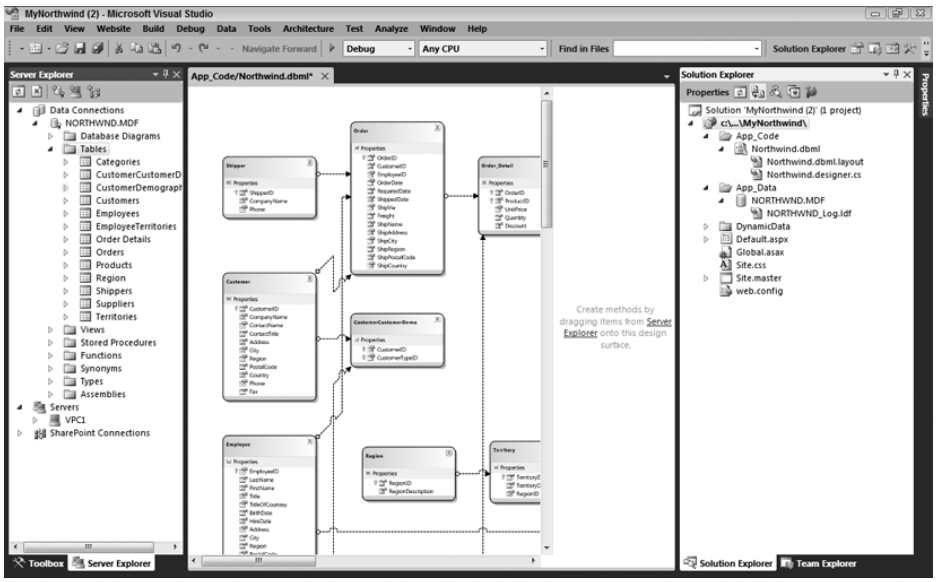


FIGURE 30-5

With this action, a bunch of code is added to the designer files of the `Northwind.dbml` file on your behalf. These classes will now give you strongly typed access to the tables of the database, and more importantly, they will be able to tie themselves to the data-driven application that has been the focus of this chapter.

You will find the code that Visual Studio generated by opening the `Northwind.designer.vb` or `Northwind.designer.cs` file within Visual Studio. Here you can find the programmatic representation of the database and the tables that you chose to add to the data model. Listing 30-6 shows a partial view of the generated code of this class.

LISTING 30-6: A partial look at the generated LINQ to SQL class

Available for download on Wrox.com

VB

```
Option Strict On
Option Explicit On

Imports System
Imports System.Collections.Generic
Imports System.ComponentModel
Imports System.Data
Imports System.Data.Linq
Imports System.Data.Linq.Mapping
Imports System.Linq
Imports System.Linq.Expressions
Imports System.Reflection

<Global.System.Data.Linq.Mapping.DatabaseAttribute(Name:="NORTHWND")> _
Partial Public Class NorthwindDataContext
   Inherits System.Data.Linq.DataContext

   ' Code removed for clarity

End Class
```

C#

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
```

```
using System.Data;
using System.Data.Linq;
using System.Data.Linq.Mapping;
using System.Linq;
using System.Linq.Expressions;
using System.Reflection;

[global::System.Data.Linq.Mapping.DatabaseAttribute(Name="NORTHWND")]
public partial class NorthwindDataContext : System.Data.Linq.DataContext
      {

      // Code removed for clarity

}
```

Filenames Northwind.designer.vb and Northwind.designer.cs

This code creates an implementation of the `DataContext` object called `NorthwindDataContext`, which manages the transactions that occur with the database when you are working with LINQ to SQL. The `DataContext`, in addition to managing the transactions with the database, also contains the connection string, takes care of any logging, and manages the output of the data.

There is a lot to this file. For example, you will find the tables represented as classes within the generated code. Listing 30-7 shows the class that represents the Customers table from the database.

LISTING 30-7: The code representation of the Customers table

VB

```
<Global.System.Data.Linq.Mapping.TableAttribute(Name:="dbo.Customers")>  _
Partial Public Class Customer
   Implements System.ComponentModel.INotifyPropertyChanging,
     System.ComponentModel.INotifyPropertyChanged

   ' Code removed for clarity

End Class
```

C#

```
[global::System.Data.Linq.Mapping.TableAttribute(Name="dbo.Customers")]
public partial class Customer : INotifyPropertyChanging,
   INotifyPropertyChanged
{

   // Code removed for clarity

}
```

Filenames Northwind.designer.vb and Northwind.designer.cs

Now that the database is residing within the application and is ready to use, the next step is to wire the database to the overall application.

Registering the Data Model Within the Global.asax File

The next step is to register the `NorthwindDataContext` object within the overall solution. The `NorthwindDataContext` is the data model that was built using the LINQ to SQL class that you just created.

One of the files that was included when you built the ASP.NET Dynamic Data application from the start was a `Global.asax` file. This file is normally not included with your standard ASP.NET application by Visual Studio. When looking at this file, you will notice that there is a lot more to this file than the normal `Global.asax` file you might be used to (if you ever included one). Listing 30-8 shows the `Global.asax` file that is provided to you (minus some of the comments).

LISTING 30-8: The Global.asax file

Available for download on Wrox.com

VB

```
<%@ Application Language="VB" %>
<%@ Import Namespace="System.ComponentModel.DataAnnotations" %>
<%@ Import Namespace="System.Web.Routing" %>
<%@ Import Namespace="System.Web.DynamicData" %>

<script RunAt="server">
Private Shared s_defaultModel As New MetaModel
Public Shared ReadOnly Property DefaultModel() As MetaModel
    Get
        Return s_defaultModel
    End Get
End Property

Public Shared Sub RegisterRoutes(ByVal routes As RouteCollection)

  ' DefaultModel.RegisterContext(GetType(YourDataContextType),
  '    New ContextConfiguration() With {.ScaffoldAllTables = False})

    routes.Add(New DynamicDataRoute("{table}/{action}.aspx") With {
        .Constraints = New RouteValueDictionary(New With {.Action =
        "List|Details|Edit|Insert"}),
        .Model = DefaultModel})
End Sub

Private Sub Application_Start(ByVal sender As Object,
   ByVal e As EventArgs)
    RegisterRoutes(RouteTable.Routes)
End Sub

</script>
```

C#

```
<%@ Application Language="C#" %>
<%@ Import Namespace="System.ComponentModel.DataAnnotations" %>
<%@ Import Namespace="System.Web.Routing" %>
<%@ Import Namespace="System.Web.DynamicData" %>

<script RunAt="server">
    private static MetaModel s_defaultModel = new MetaModel();
    public static MetaModel DefaultModel {
        get {
            return s_defaultModel;
        }
    }

    public static void RegisterRoutes(RouteCollection routes)
    {
        //DefaultModel.RegisterContext(typeof(YourDataContextType),
        //      new ContextConfiguration()
        //      { ScaffoldAllTables = false });

        routes.Add(new DynamicDataRoute("{table}/{action}.aspx")
        {
            Constraints = new RouteValueDictionary(new
               { action = "List|Details|Edit|Insert" }),
            Model = DefaultModel
        });
    }

    void Application_Start(object sender, EventArgs e)
    {
        RegisterRoutes(RouteTable.Routes);
    }

</script>
```

If you are familiar with the standard `Global.asax` file, then you will recognize the standard `Application_Start()` method that is part of the file. This method fires whenever the ASP.NET application starts up for the first time or is recycled in some manner. The `Application_Start()` method here calls a method named `RegisterRoutes()`, which does the wiring between the generated application and the data model that you created earlier.

You must register the data model that you created, the instance of the `DataContext` object. In this example, you must wire up the `NorthwindDataContext` object. Looking over the code from Listing 30-8, you can see that one of the lines of code is commented out. This is the `RegisterContext()` method call. Uncomment this line of code, and instead of having a reference to an object of type `YourDataContextType`, change it to `NorthwindDataContext`. In the end, your line of code will be as shown in Listing 30-9.

LISTING 30-9: Wiring the NorthwindDataContext object

VB
```
model.RegisterContext(GetType(NorthwindDataContext), _
    New ContextConfiguration() With {.ScaffoldAllTables = True})
```

C#
```
model.RegisterContext(typeof(NorthwindDataContext),
    new ContextConfiguration() { ScaffoldAllTables = true });
```

So, here the model is registered as a `DataContext` object of type `NorthwindDataContext`, and the `ScaffoldAllTables` property is set to `True` (the default is set to `False`), signifying that you want all the table representations in the model to be included in the generated application.

Styles and Layout

With the data model wired to the application, you are actually ready to run the application and see what it produces. However, before this operation, you must be aware of two more pieces of this application.

When you created the application, a master page and a stylesheet were also included. The master page, `Site.master`, is rather simple but does include the overall page framework that allows you to navigate the tables. There is also a simple stylesheet called `Site.css` that provides the overall style of the page.

Results of the Application

As you run the application, notice that the first page allows you to see all the tables that you made a part of your data model, as illustrated in Figure 30-6.

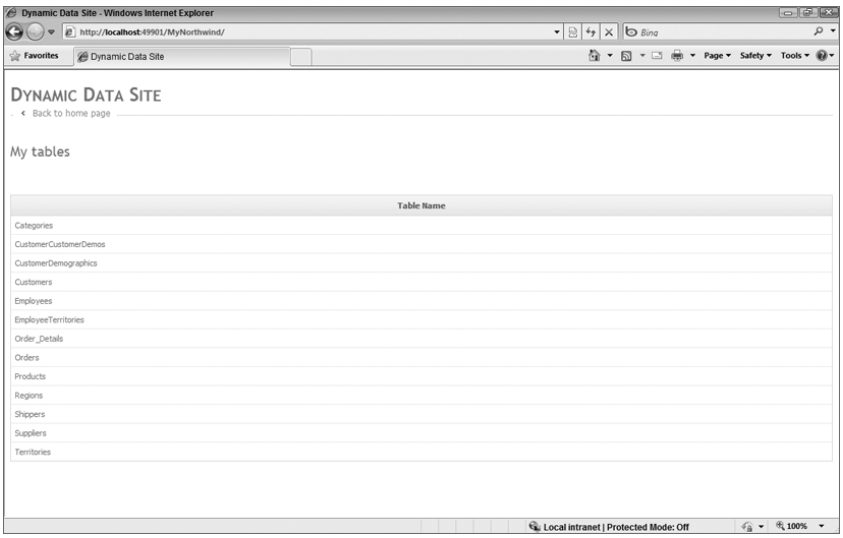


FIGURE 30-6

As an application that reads the contents of your database, it works quite simply. Clicking on a table name (which is a hyperlink in the application) provides the contents of the table. Figure 30-7 shows this view.

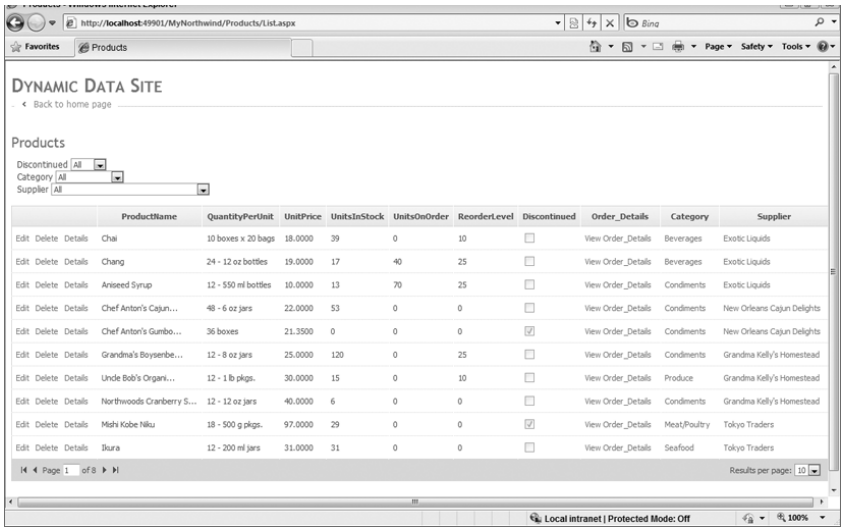


FIGURE 30-7

The table view is nicely styled and includes the ability to edit, delete, or view the details of each row in the database. In cases where a one-to-many relationship exists, you can drill down deeper into it. Another interesting part of the page is the navigation through the table. Pagination appears through the table, as shown at the bottom of the table in Figure 30-7.

In addition to being able to use the edit, delete, or detail view of the row of information in the table, you can insert new rows into the database by clicking on the Insert New Item link below the table. A view similar to that shown in Figure 30-8 appears.

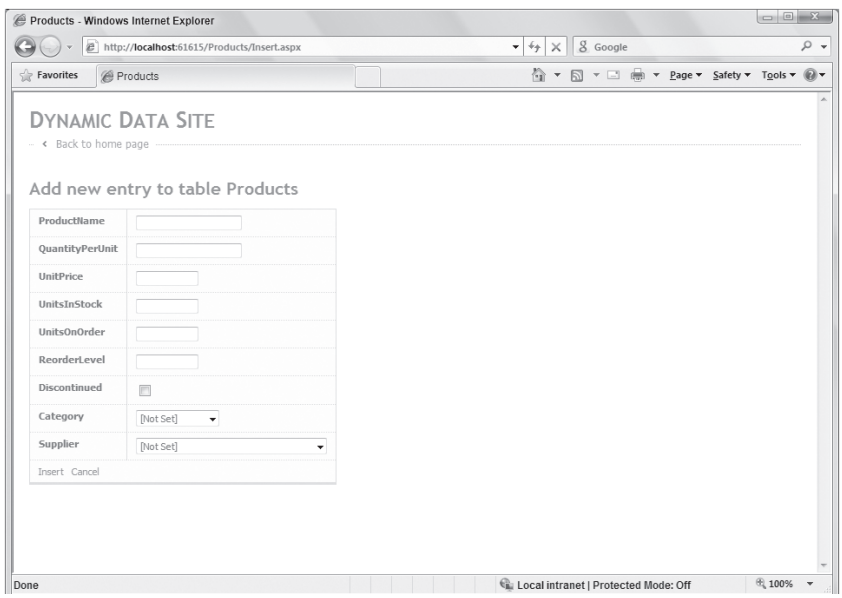


FIGURE 30-8

Editing a row makes the same type of page appear, as shown in Figure 30-9. This page resulted from clicking the Edit link next to one of the rows in the Products table.

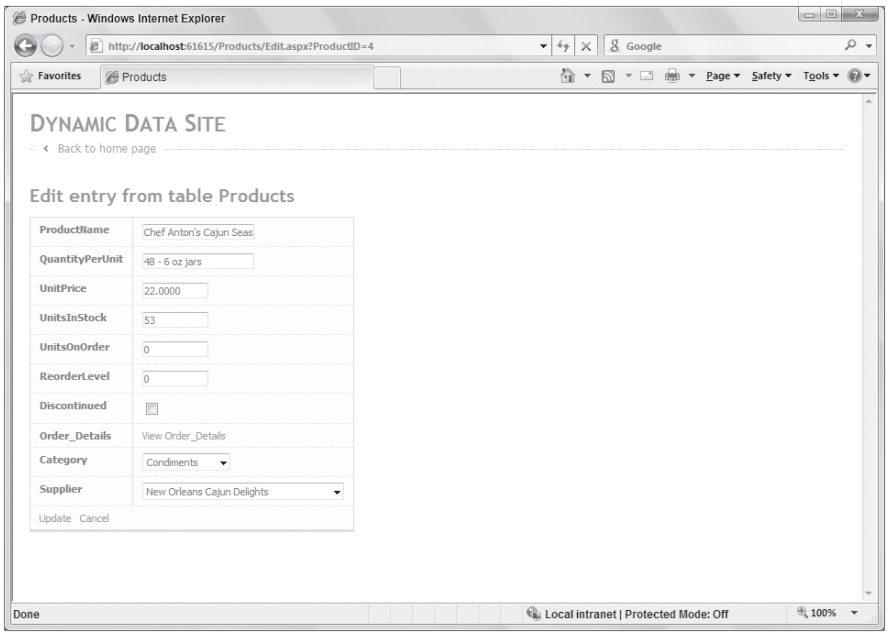


FIGURE 30-9

Another interesting aspect of this application is how it works with the one-to-many relationships of the elements in the data model. For example, clicking the Orders table link produces the view shown in Figure 30-10.

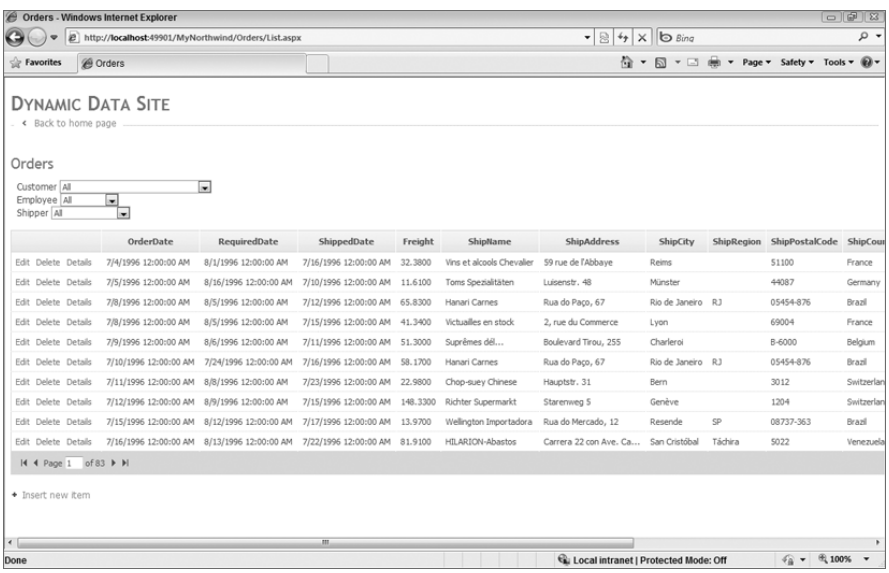


FIGURE 30-10

Here, in addition to the table and its contents, you are presented with a filtering capability via the various drop-down menus at the top of the page. In this case, you can filter the orders by customer, employee, or shipper. You can even use a combination of these elements to filter items. The other aspect to notice in Figure 30-10 about the contents of the table is that instead of a `CustomerID`, an `EmployeeID`, or a `ShipperID`, you see the names of these items, and the application is making the reference to the identifier for these items when drilling further into the views.

The final aspect to understand about this application is that because it is an ASP.NET 4 application, it makes proper use of AJAX. For example, when filtering items in the table, notice that the page makes partial refreshes without refreshing the entire page.

WORKING WITH DYNAMIC DATA ROUTES

As you can see, the application created was quite easy to implement and took only a couple of steps to accomplish. An interesting aspect of this page is how easily configurable it is overall. In particular, you can configure how the URLs are generated and what pages the routing uses.

Going back to the `Global.asax` page that you set up in the previous example, you will find that underneath the `RegisterContext()` call is a method call that sets up the page construct and URL that you will use for the application. Listing 30-10 presents what is established as the default.

LISTING 30-10: Viewing the default routing URL

VB
```
routes.Add(New DynamicDataRoute("{table}/{action}.aspx") With {
        .Constraints = New RouteValueDictionary(New With {.Action =
        "List|Details|Edit|Insert"}),
        .Model = DefaultModel})
```

C#
```
routes.Add(new DynamicDataRoute("{table}/{action}.aspx")
{
   Constraints = new RouteValueDictionary(new { action =
      "List|Details|Edit|Insert" }),
      Model = DefaultModel
});
```

If you go back to the URL that was used in something like an Edit mode operation, you will find a URL like the following:

```
http://localhost:2116/Orders/Edit.aspx?OrderID=10249
```

After the default application, you will find `Orders/Edit.aspx?OrderID=10249`. Here, `Orders` is the name of the table, `Edit` is the operation, and the `OrderID` is the filter on the order that is being worked on.

Returning to Listing 30-10, you can see this model for the URL defined directly in the code through the use of the string:

```
{table}/{action}.aspx
```

Now, if you want to change this model, you can! These are all dynamic URLs and not actual URLs as you know them from the past. For instance, there isn't an Orders folder present in the application. It is simply there as a dynamic route. This is evident when you change the dynamic route path yourself; for example, change the `routes.Add()` call so that it now says the following (shown here in VB as a partial code sample):

```
Routes.Add(New DynamicDataRoute("{action}/{table}.aspx") ...
```

Now, when you go to edit an item, you get the following construct:

```
http://localhost:2116/Edit/Orders.aspx?OrderID=10249
```

From this view, you can see that the action and the table reference have been reversed.

Looking deeper in the `Global.asax` file, you can see that two more pre-established models are at your disposal. Listing 30-11 presents both of them.

LISTING 30-11: The other two routing models

Available for download on Wrox.com

VB

```
'routes.Add(New DynamicDataRoute("{table}/ListDetails.aspx") With { _
' .Action = PageAction.List, _
' .ViewName = "ListDetails", _
' .Model = DefaultModel})

'routes.Add(New DynamicDataRoute("{table}/ListDetails.aspx") With { _
' .Action = PageAction.Details, _
' .ViewName = "ListDetails", _
' .Model = DefaultModel})
```

C#

```
//routes.Add(new DynamicDataRoute("{table}/ListDetails.aspx") {
//    Action = PageAction.List,
//    ViewName = "ListDetails",
//    Model = DefaultModel
//});

//routes.Add(new DynamicDataRoute("{table}/ListDetails.aspx") {
//    Action = PageAction.Details,
//    ViewName = "ListDetails",
//    Model = DefaultModel
//});
```

The example previously presented in this chapter showed each of the operations (for inserting, editing, and so on) on its own separate page. Commenting this previous `routes.Add()` method out of the `Global.asax` file and uncommenting out the second and third options from this file changes the page that is output, and a master/details view now appears on the same page. Figure 30-11 shows one such view of the tables using this new routing.

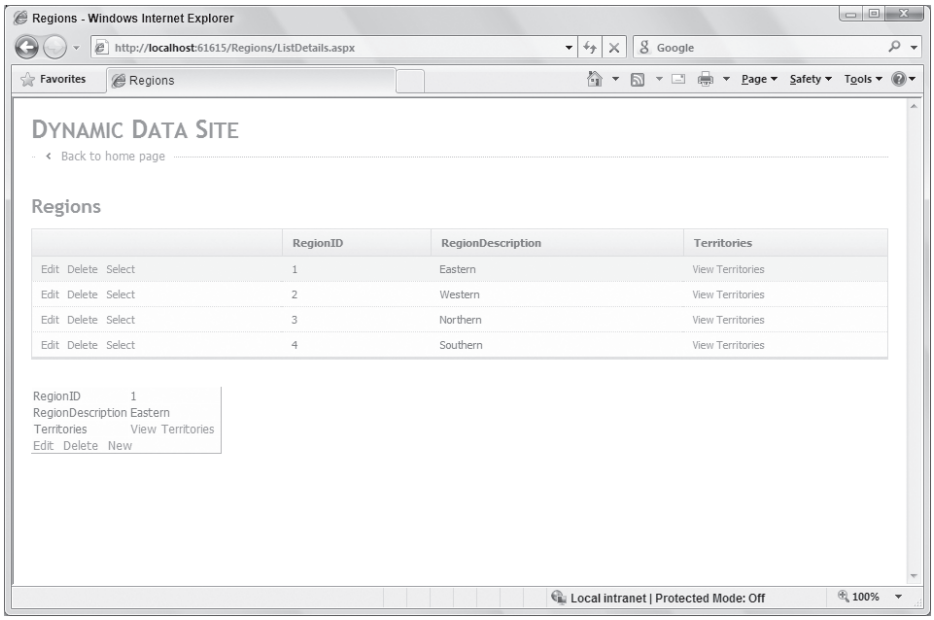


FIGURE 30-11

Now when you select an item, the details of that item also appear on the same page. Also, you are now working on a single page, so when you click the Edit link in the table, you are not sent to another page, but instead, you can now edit the contents of the row directly in the table, as shown in Figure 30-12.

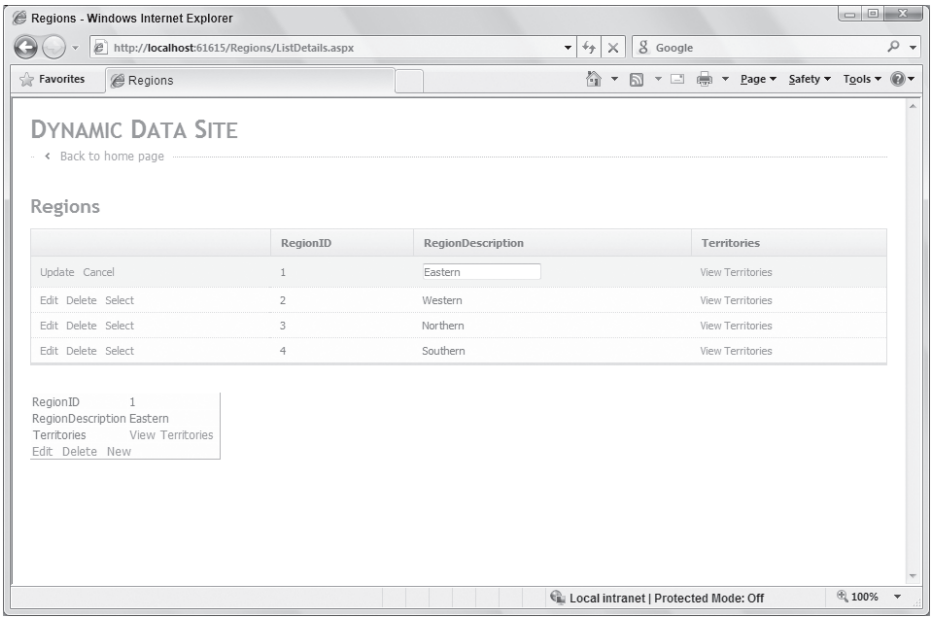


FIGURE 30-12

You need to pay attention to a couple of things when you make these routing changes. If you return to the original routing definition, you can then include the following:

VB
```
routes.Add(New DynamicDataRoute("{table}/{action}.aspx") With { _
        .Constraints = New RouteValueDictionary(New With {.Action = _
        "List|Details|Edit|Insert"}), _
        .Model = DefaultModel})
```

C#
```
routes.Add(new DynamicDataRoute("{table}/{action}.aspx")
{
   Constraints = new RouteValueDictionary(new { action =
      "List|Details|Edit|Insert" }),
      Model = DefaultModel
});
```

From this code, it is true that the route is declared using the table/action options, but just as important, notice that actions are also defined.

This code example basically identifies this route for the List, Details, Edit, and Insert actions. You could have also pointed to other options for each of the action types. The second and third entries in the `Global.asax` file do just that:

VB
```
'routes.Add(New DynamicDataRoute("{table}/ListDetails.aspx") With { _
' .Action = PageAction.List, _
' .ViewName = "ListDetails", _
' .Model = DefaultModel})
```

C#
```
//routes.Add(new DynamicDataRoute("{table}/ListDetails.aspx") {
//    Action = PageAction.List,
//    ViewName = "ListDetails",
//    Model = DefaultModel
//});
```

In this partial code example, you can see that this dynamic data route is defined only for the List operations and does not include the Edit, Insert, or Details operations. In addition, instead of going to `List.aspx` or `Details.aspx`, this operation will go to a specific page — `ListDetails.aspx`. The `ListDetails.aspx`

page provides you the ability to perform everything from a single page, which is why you don't need to define the Edit, Insert, and Details operations in this case.

In addition to everything this chapter has presented thus far, another interesting task you can perform is table-specific routing.

Suppose you want to use the `ListDetails.aspx` page for all the tables *except* the Customers table. For this example, assume that you want to return to the base `List.aspx`, `Details.aspx`, and `Edit.aspx` just when the end user is working with the contents of the Customers table. In this case, you need to establish your routing as illustrated in Listing 30-12.

LISTING 30-12: Routing for specific tables

VB
```
routes.Add(New DynamicDataRoute("Customers/{action}.aspx") With { _
   .Constraints = New RouteValueDictionary(New With {.Action = _
     "List|Details|Edit|Insert"}), _
   .Model = DefaultModel, _
   .Table = "Customers"})

routes.Add(New DynamicDataRoute("{table}/ListDetails.aspx") With { _
   .Action = PageAction.List, _
   .ViewName = "ListDetails", _
   .Model = DefaultModel})

routes.Add(New DynamicDataRoute("{table}/ListDetails.aspx") With { _
   .Action = PageAction.Details, _
   .ViewName = "ListDetails", _
   .Model = DefaultModel})
```

C#
```
routes.Add(new DynamicDataRoute("Customers/{action}.aspx")
{
      Constraints = new RouteValueDictionary(new { action =
        "List|Details|Edit|Insert" }),
      Model = DefaultModel,
      Table = "Customers"
});

routes.Add(new DynamicDataRoute("{table}/ListDetails.aspx")
{
      Action = PageAction.List,
      ViewName = "ListDetails",
      Model = DefaultModel
});

routes.Add(new DynamicDataRoute("{table}/ListDetails.aspx")
{
      Action = PageAction.Details,
      ViewName = "ListDetails",
      Model = DefaultModel
});
```

In this case, a dynamic data route of `Customers/{action}.aspx` is provided and will be used when the `Table` reference is equal to the Customers table through the `Table = "Customers"` declaration in the code.

With these three routing statements, the Customers table will use the `List.aspx`, `Details.aspx`, `Edit.aspx`, and `Insert.aspx` pages, whereas all the other tables will use the `ListDetails.aspx` page.

CONTROLLING DISPLAY ASPECTS

ASP.NET Dynamic Data has a wealth of features — more than a single chapter can possibly cover. One final aspect to note is how easy removing tables and columns from the overall application is.

As shown in this chapter, you provided a data model that ended up controlling which parts of the database the data-driven application would work with. In this way, you can create new data models or even use pre-existing data models that might be present in your application.

Even though something is defined within your data model, you might not want that item present in the application. For this reason, you can use a couple of attributes to control whether a table or a column is present and can be worked with by the end user of the application.

For example, suppose you did not want the Employees table to appear in the data-driven application even though this table is present in the data model that you created. In this case, you should use the `ScaffoldTable` attribute on the class that defines the table and set it to `False`. For this method to work, open the `Northwind.designer.vb` or `Northwind.designer.cs` file and make the addition illustrated in Listing 30-13.

LISTING 30-13: Removing the Employees table from view

VB
```
<ScaffoldTable(False)> _
<Global.System.Data.Linq.Mapping.TableAttribute(Name:="dbo.Employees")> _
Partial Public Class Employee
    Implements System.ComponentModel.INotifyPropertyChanging,
       System.ComponentModel.INotifyPropertyChanged

   ' Code removed for clarity

End Class
```

C#
```
[System.ComponentModel.DataAnnotations.ScaffoldTable(false)]
[global::System.Data.Linq.Mapping.TableAttribute(Name="dbo.Employees")]
public partial class Employee : INotifyPropertyChanging,
   INotifyPropertyChanged
{

   // Code removed for clarity

}
```

To use the `ScaffoldTable` attribute, reference the `System.ComponentModel.DataAnnotations` namespace. With this reference in place, running the page will now show a list of tables that excludes the Employees table (see Figure 30-13).

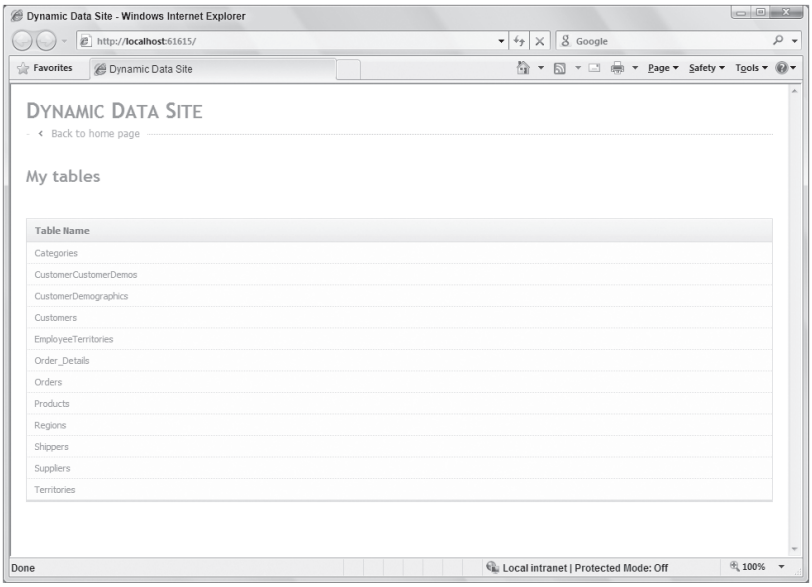


FIGURE 30-13

Similar to the `ScaffoldTable` attribute, the `ScaffoldColumn` attribute is at your disposal (along with many other attributes). When this attribute is applied to a column, that column will be removed from view in the data-driven application.

For example, suppose that you want the Employees table to be viewable, but you do not want users to be able to see employees' home telephone numbers. In this case, you can remove that column by using the `ScaffoldColumn` attribute to set the column's visibility to `False`, as illustrated in Listing 30-14.

LISTING 30-14: Setting the visibility of a column to False

VB
```
<ScaffoldColumn(False)> _
<Global.System.Data.Linq.Mapping.ColumnAttribute(Storage:="_HomePhone",
   DbType:="NVarChar(24)")> _
Public Property HomePhone() As String
      Get
         Return Me._HomePhone
      End Get
      Set(ByVal value As String)
         If (String.Equals(Me._HomePhone, value) = False) Then
            Me.OnHomePhoneChanging(value)
            Me.SendPropertyChanging()
            Me._HomePhone = value
            Me.SendPropertyChanged("HomePhone")
            Me.OnHomePhoneChanged()
         End If
      End Set
End Property
```

C#
```
[System.ComponentModel.DataAnnotations.ScaffoldColumn(false)]
[global::System.Data.Linq.Mapping.ColumnAttribute(Storage="_HomePhone",
   DbType="NVarChar(24)")]
public string HomePhone
{
   get
   {
      return this._HomePhone;
   }
   set
   {
      if ((this._HomePhone != value))
      {
         this.OnHomePhoneChanging(value);
         this.SendPropertyChanging();
         this._HomePhone = value;
         this.SendPropertyChanged("HomePhone");
         this.OnHomePhoneChanged();
      }
   }
}
```

With this code in place, users would see the table and all the columns of the table, except for the `HomePhone` column.

ADDING DYNAMIC DATA TO EXISTING PAGES

When ASP.NET Dynamic Data was first introduced with the .NET Framework 3.5 SP1, it took a bit of setup in order to get dynamic aspects on your pages. With the release of the .NET Framework 4, you will find that it is a lot easier to add dynamic portions to your Web pages.

This is now possible by using the new DynamicDataManager server control. For an example of this in action, take a look at Listing 30-15:

LISTING 30-15: Using the DynamicDataManager server control with an existing GridView control

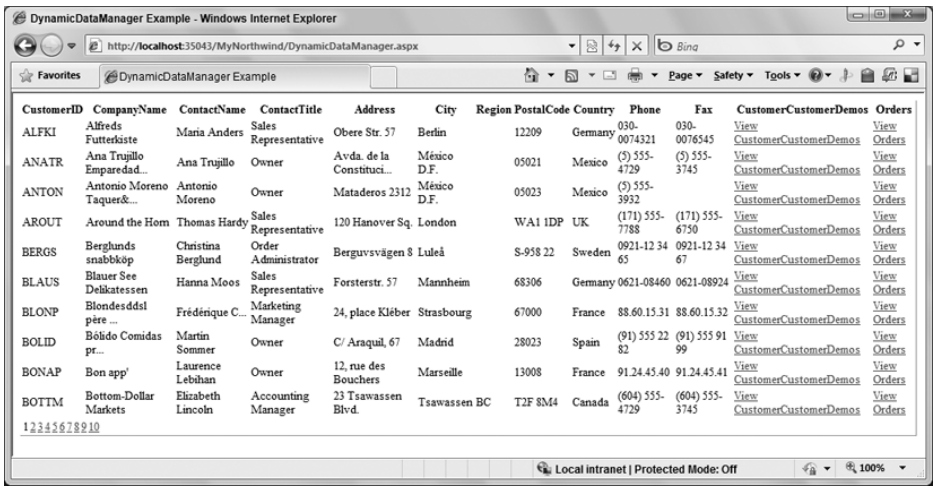
```
<%@ Page Language="C#" %>

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

        <asp:GridView ID="GridView1" runat="server" AllowPaging="True"
            AutoGenerateColumns="True" DataKeyNames="CustomerID"
            DataSourceID="LinqDataSource1">
        </asp:GridView>
        <asp:LinqDataSource ID="LinqDataSource1" runat="server"
            ContextTypeName="NorthwindDataContext" EntityTypeName=""
            OrderBy="CompanyName"
            TableName="Customers">
        </asp:LinqDataSource>
        <asp:DynamicDataManager ID="DynamicDataManager1"
         runat="server">
            <DataControls>
                <asp:DataControlReference ControlID="GridView1" />
            </DataControls>
        </asp:DynamicDataManager>

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

Filename DynamicDataManager.aspx

Using the same Northwind data context from the previous examples, you will find a basic LinqDataSource control has been added to the page to work with the `GridView1` server control. The LinqDataSource has also been assigned to work off the Customers table in this example. What you want is a grid that shows a list of the customers.

The next step in building this example is to add a DynamicDataManager control to the page and to have it work with the `GridView1` control as well. This is done by adding a `<DataControls>` section and making a control reference using the `ControlID` attribute of the DataControlReference control.

Running this page will produce the following results as demonstrated here in Figure 30-14:



CustomerID	CompanyName	ContactName	ContactTitle	Address	City	Region	PostalCode	Country	Phone	Fax	CustomerCustomerDemos	Orders
ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57	Berlin		12209	Germany	030-0074321	030-0076545	View CustomerCustomerDemos	View Orders
ANATR	Ana Trujillo Emparedad...	Ana Trujillo	Owner	Avda. de la Constituci...	México D.F.		05021	Mexico	(5) 555-4729	(5) 555-3745	View CustomerCustomerDemos	View Orders
ANTON	Antonio Moreno Taquer&...	Antonio Moreno	Owner	Mataderos 2312	México D.F.		05023	Mexico	(5) 555-3932		View CustomerCustomerDemos	View Orders
AROUT	Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq.	London		WA1 1DP	UK	(171) 555-7788	(171) 555-6750	View CustomerCustomerDemos	View Orders
BERGS	Berglunds snabbköp	Christina Berglund	Order Administrator	Berguvsvägen 8	Luleå		S-958 22	Sweden	0921-12 34 65	0921-12 34 67	View CustomerCustomerDemos	View Orders
BLAUS	Blauer See Delikatessen	Hanna Moos	Sales Representative	Forsterstr. 57	Mannheim		68306	Germany	0621-08460	0621-08924	View CustomerCustomerDemos	View Orders
BLONP	Blondesddsl père ...	Frédérique C...	Marketing Manager	24, place Kléber	Strasbourg		67000	France	88.60.15.31	88.60.15.32	View CustomerCustomerDemos	View Orders
BOLID	Bólido Comidas pr...	Martín Sommer	Owner	C/ Araquil, 67	Madrid		28023	Spain	(91) 555 22 82	(91) 555 91 99	View CustomerCustomerDemos	View Orders
BONAP	Bon app'	Laurence Lebihan	Owner	12, rue des Bouchers	Marseille		13008	France	91.24.45.40	91.24.45.41	View CustomerCustomerDemos	View Orders
BOTTM	Bottom-Dollar Markets	Elizabeth Lincoln	Accounting Manager	23 Tsawassen Blvd.	Tsawassen	BC	T2F 8M4	Canada	(604) 555-4729	(604) 555-3745	View CustomerCustomerDemos	View Orders

1 2 3 4 5 6 7 8 9 10

FIGURE 30-14

From this example, you can see that the hierarchical content associated with the customers is interpreted and displayed as links in the table. All of this is dynamic.

Another control to be aware of is the DynamicControl server control. This is utilized when you have a templated control and want ASP.NET to make the smart decision on how to render the appropriate HTML for this control. For an example of this, redo the previous example from Listing 30-15 and change the GridView server control to a ListView server control. You will also need to change the reference in the DynamicDataManager control for this example to work.

Dragging and dropping the ListView server control onto the design surface of Visual Studio, you will notice that it now has the ability to enable the control for Dynamic Data, as is shown when configuring the ListView control in Figure 30-15.

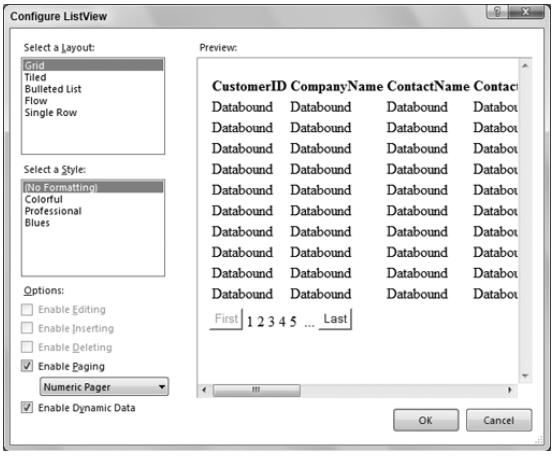


FIGURE 30-15

Checking the Enable Dynamic Data checkbox in this dialog will generate a lot of controls for all the different views provided by the template. With the code generated by Visual Studio, you will see a lot of use of the DynamicControl control. An example of one of the control instances generated (in an edit mode) is presented here:

```
<asp:DynamicControl runat="server" DataField="CompanyName" Mode="Edit" />
```

With the combination of these new controls, you can now quickly and easily add Dynamic Data to any of your ASP.NET pages.

SUMMARY

This chapter explored the capabilities in ASP.NET for building Dynamic Data-driven applications quickly and easily. These capabilities, in conjunction with Visual Studio, enable you to build a reporting application that provides full CRUD capabilities in less than five minutes.

At the same time, you can delve deep into the application and modify how the application presents the contents of your database.