# Table of Contents

**Chapter 13. Managing Views of a Record** ................................................................. 1  
The DetailsView Control ................................................................................................................. 1  
The FormView Control .................................................................................................................. 25  
Conclusion ................................................................................................................................ 33
Chapter 13
Managing Views of a Record

In this chapter:
The DetailsView Control ........................................... 613
The FormView Control ........................................... 637
Conclusion .......................................................... 645

Many applications need to work on a single record at a time. Old versions of ASP.NET had no built-in support for this scenario. For example, creating a single record view is possible in ASP.NET 1.x, but it requires some coding and, possibly, a custom control. You have to fetch the record, bind its fields to a data-bound form, and optionally provide paging buttons to navigate between records. Displaying the contents of a single record is a common and necessary practice when you build master/detail views. Typically, the user selects a master record from a list or a grid, and the application drills down to show all the available fields. In ASP.NET 2.0 and newer versions, the DetailsView control fulfills this role and is the ideal complement to the DataGrid, GridView, and ListView controls that we examined in Chapter 11 and Chapter 12.

The DetailsView control deliberately doesn’t support templates. A fully templated details-view control is the FormView control, which we’ll also cover in this chapter.

The DetailsView Control

The DetailsView is a data-bound control that renders a single record at a time from its associated data source, optionally providing paging buttons to navigate between records. It is similar to the Form View of a Microsoft Access database and is typically used for updating and inserting records in a master/detail scenario.

The DetailsView control binds to any data source control and executes its set of data operations. It can page, update, insert, and delete data items in the underlying data source as long as the data source supports these operations. In most cases, no code is required to set up any of these operations. You can customize the user interface of the DetailsView control by choosing the most appropriate combination of data fields and styles in much the same way that you do with the GridView.

Finally, note that although the DetailsView is commonly used as an update and insert interface, it does not perform any input validation against the data source schema, nor does...
it provide any schematized user interface such as foreign key field drop-down lists or made-to-measure edit templates for particular types of data.

**The DetailsView Object Model**

The *DetailsView* is to a single record what a *GridView* is to a page of records. Just as the grid lets you choose which columns to display, the *DetailsView* allows you to select a subset of fields to display in read-only or read/write fashion. The rendering of the *DetailsView* is largely customizable using templates and styles. The default rendering consists of a vertical list of rows, one for each field in the bound data item. *DetailsView* is a composite data-bound control and acts as a naming and binding container. Much like the *GridView*, the *DetailsView* control also supports out-of-band calls for paging through the `ICallbackContainer` and `ICallbackEventHandler` interfaces. Here’s the declaration of the control class:

```csharp
public class DetailsView : CompositeDataBoundControl, IItemContainer, ICallbackContainer, ICallbackEventHandler, INamingContainer
```

The control is formed by a few main areas—a header, field rows, a pager bar, a command bar, and a footer.

**FIGURE 13-1** A DetailsView control in action.
Properties of the DetailsView

The DetailsView layout supports several properties that fall into the following categories: behavior, appearance, style, state, and templates. Table 13-1 lists the behavioral properties.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllowPaging</td>
<td>Indicates whether the control supports navigation.</td>
</tr>
<tr>
<td>AutoGenerateDeleteButton</td>
<td>Indicates whether the command bar includes a Delete button. The default is false.</td>
</tr>
<tr>
<td>AutoGenerateEditButton</td>
<td>Indicates whether the command bar includes an Edit button. The default is false.</td>
</tr>
<tr>
<td>AutoGenerateInsertButton</td>
<td>Indicates whether the command bar includes an Insert button. The default is false.</td>
</tr>
<tr>
<td>AutoGenerateRows</td>
<td>Indicates whether the control auto-generates the rows. The default is true—all the fields of the record are displayed.</td>
</tr>
<tr>
<td>DataMember</td>
<td>Indicates the specific table in a multimember data source to bind to the control. The property works in conjunction with DataSource. If DataSource is a DataSet object, it contains the name of the particular table to bind.</td>
</tr>
<tr>
<td>DataSource</td>
<td>Gets or sets the data source object that contains the values to populate the control.</td>
</tr>
<tr>
<td>DataSourceID</td>
<td>Indicates the bound data source control.</td>
</tr>
<tr>
<td>DefaultMode</td>
<td>Indicates the default display mode of the control. It can be any value from the DetailsViewMode enumeration (read-only, insert, edit).</td>
</tr>
<tr>
<td>EnablePagingCallbacks</td>
<td>Indicates whether client-side callback functions are used for paging operations.</td>
</tr>
<tr>
<td>PagerSettings</td>
<td>Gets a reference to the PagerSettings object that allows you to set the properties of the pager buttons.</td>
</tr>
<tr>
<td>UseAccessibleHeader</td>
<td>Determines whether to render <code>&lt;th&gt;</code> tags for the column headers instead of default <code>&lt;td&gt;</code> tags.</td>
</tr>
</tbody>
</table>

The DefaultMode property determines the initial working mode of the control and also the mode that the control reverts to after an edit or insert operation is performed.

The output generated by the DetailsView control is a table in which each row corresponds to a record field. Additional rows represent special items such as the header, footer, pager, and new command bar. The command bar is a sort of toolbar where all the commands available on the record are collected. Auto-generated buttons go to the command bar.

The user interface of the control is governed by a handful of visual properties, which are listed in Table 13-2.
TABLE 13-2  DetailsView Appearance Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackImageUrl</td>
<td>Indicates the URL to an image to display in the background</td>
</tr>
<tr>
<td>Caption</td>
<td>The text to render in the control’s caption</td>
</tr>
<tr>
<td>CaptionAlign</td>
<td>Alignment of the caption</td>
</tr>
<tr>
<td>CellPadding</td>
<td>Gets or sets the space (in pixels) remaining between the cell’s border and the embedded text</td>
</tr>
<tr>
<td>CellSpacing</td>
<td>Gets or sets the space (in pixels) remaining, both horizontally and vertically, between two consecutive cells</td>
</tr>
<tr>
<td>EmptyDataText</td>
<td>Indicates the text to render in the control when bound to an empty data source</td>
</tr>
<tr>
<td>FooterText</td>
<td>Indicates the text to render in the control’s footer</td>
</tr>
<tr>
<td>Gridlines</td>
<td>Indicates the gridline style for the control</td>
</tr>
<tr>
<td>HeaderText</td>
<td>Indicates the text to render in the control’s header</td>
</tr>
<tr>
<td>HorizontalAlign</td>
<td>Indicates the horizontal alignment of the control on the page</td>
</tr>
</tbody>
</table>

The properties listed in the table apply to the control as a whole. You can program specific elements of the control’s user interface by using styles. The supported styles are listed in Table 13-3.

TABLE 13-3  DetailsView Style Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlternatingRowStyle</td>
<td>Defines the style properties for the fields that are displayed for each even-numbered row</td>
</tr>
<tr>
<td>CommandRowStyle</td>
<td>Defines the style properties for the command bar</td>
</tr>
<tr>
<td>EditRowStyle</td>
<td>Defines the style properties of individual rows when the control renders in edit mode</td>
</tr>
<tr>
<td>EmptyDataRowStyle</td>
<td>Defines the style properties for the displayed row when no data source is available</td>
</tr>
<tr>
<td>FieldHeaderStyle</td>
<td>Defines the style properties for the label of each field value</td>
</tr>
<tr>
<td>FooterStyle</td>
<td>Defines the style properties for the control’s footer</td>
</tr>
<tr>
<td>HeaderStyle</td>
<td>Defines the style properties for the control’s header</td>
</tr>
<tr>
<td>InsertRowStyle</td>
<td>Defines the style properties of individual rows when the control renders in insert mode</td>
</tr>
<tr>
<td>PagerStyle</td>
<td>Defines the style properties for the control’s pager</td>
</tr>
<tr>
<td>RowStyle</td>
<td>Defines the style properties of the individual rows</td>
</tr>
</tbody>
</table>

The DetailsView control can be displayed in three modes, depending on the value—ReadOnly, Insert, or Edit—of the DetailsViewMode enumeration. The read-only mode is the...
default display mode in which users see only the contents of the record. To edit or add a new record, users must click the corresponding button (if any) on the command bar. Such buttons must be explicitly enabled on the command bar through the AutoGenerateXxxButton properties. Each mode has an associated style. The current mode is tracked by the CurrentMode read-only property.

Other state properties are listed in Table 13-4.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BottomPagerRow</td>
<td>Returns a DetailsViewRow object that represents the bottom pager of the control.</td>
</tr>
<tr>
<td>CurrentMode</td>
<td>Gets the current mode for the control—any of the values in the DetailsViewMode enumeration. The property determines how bound fields and templates are rendered.</td>
</tr>
<tr>
<td>DataItem</td>
<td>Returns the data object that represents the currently displayed record.</td>
</tr>
<tr>
<td>DataItemCount</td>
<td>Gets the number of items in the underlying data source.</td>
</tr>
<tr>
<td>DataItemIndex</td>
<td>Gets or sets the index of the item being displayed from the underlying data source.</td>
</tr>
<tr>
<td>DataKey</td>
<td>Returns the DataKey object for the currently displayed record. The DataKey object contains the key values corresponding to the key fields specified by DataKeyNames.</td>
</tr>
<tr>
<td>DataKeyNames</td>
<td>An array specifying the primary key fields for the records being displayed. These keys are used to uniquely identify an item for update and delete operations.</td>
</tr>
<tr>
<td>Fields</td>
<td>Returns the collection of DataControlField objects for the control that was used to generate the Rows collection.</td>
</tr>
<tr>
<td>FooterRow</td>
<td>Returns a DetailsViewRow object that represents the footer of the control.</td>
</tr>
<tr>
<td>HeaderRow</td>
<td>Returns a DetailsViewRow object that represents the header of the control.</td>
</tr>
<tr>
<td>PageCount</td>
<td>Returns the total number of items in the underlying data source bound to the control.</td>
</tr>
<tr>
<td>PageIndex</td>
<td>Returns the 0-based index for the currently displayed record in the control. The index is relative to the total number of records in the underlying data source.</td>
</tr>
<tr>
<td>Rows</td>
<td>Returns a collection of DetailsViewRow objects representing the individual rows within the control. Only data rows are taken into account.</td>
</tr>
<tr>
<td>SelectedValue</td>
<td>Returns the value of the key for the current record as stored in the DataKey object.</td>
</tr>
<tr>
<td>TopPagerRow</td>
<td>Returns a DetailsViewRow object that represents the top pager of the control.</td>
</tr>
</tbody>
</table>
If you’re not satisfied with the default control rendering, you can use certain templates to better adapt the user interface to your preferences. Table 13-5 details the supported templates.

**TABLE 13-5 DetailsView Template Properties**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EmptyDataTemplate</td>
<td>The template for rendering the control when it is bound to an empty data source. If set, this property overrides the EmptyDataText property.</td>
</tr>
<tr>
<td>FooterTemplate</td>
<td>The template for rendering the footer row of the control.</td>
</tr>
<tr>
<td>HeaderTemplate</td>
<td>The template for rendering the header of the control. If set, this property overrides theHeaderText property.</td>
</tr>
<tr>
<td>PagerTemplate</td>
<td>The template for rendering the pager of the control. If set, this property overrides any existing pager settings.</td>
</tr>
</tbody>
</table>

As you can see, the list of templates is related to the layout of the control and doesn’t include any template that influences the rendering of the current record. This is by design. For more ambitious template properties, such as `InsertTemplate` or perhaps `ItemTemplate`, you should resort to the `FormView` control, which is the fully templated sibling of the DetailsView control.

The DetailsView control has only one method, `ChangeMode`. As the name suggests, the `ChangeMode` method is used to switch from one display mode to the next.

```csharp
public void ChangeMode(DetailsViewMode newMode)
```

This method is used internally to change views when a command button is clicked.

**Events of the DetailsView**

The DetailsView control exposes several events that enable the developer to execute custom code at various times in the life cycle. The event model is similar to that of the GridView control in terms of supported events and because of the pre/post pair of events that characterize each significant operation. Table 13-6 details the supported events.

**TABLE 13-6 Events of the DetailsView Control**

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ItemCommand</td>
<td>Occurs when any clickable element in the user interface is clicked. This doesn’t include standard buttons (such as Edit, Delete, and Insert), which are handled internally, but it does include custom buttons defined in the templates.</td>
</tr>
<tr>
<td>ItemCreated</td>
<td>Occurs after all the rows are created.</td>
</tr>
<tr>
<td>ItemDeleting, ItemDeleted</td>
<td>Both events occur when the current record is deleted. They fire before and after the record is deleted.</td>
</tr>
</tbody>
</table>
### Event Description

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ItemInserting, ItemInserted</td>
<td>Both events occur when a new record is inserted. They fire before and after the insertion.</td>
</tr>
<tr>
<td>ItemUpdating, ItemUpdated</td>
<td>Both events occur when the current record is updated. They fire before and after the record is updated.</td>
</tr>
<tr>
<td>ModeChanging, ModeChanged</td>
<td>Both events occur when the control switches to a different display mode. They fire before and after the mode change occurs.</td>
</tr>
<tr>
<td>PageIndexChanging,PageIndexChanged</td>
<td>Both events occur when the control moves to another record. They fire before and after the display change occurs.</td>
</tr>
</tbody>
</table>

The **ItemCommand** event fires only if the original click event is not handled by a predefined method. This typically occurs if you define custom buttons in one of the templates. You do not need to handle this event to intercept any clicking on the Edit or Insert buttons.

### Simple Data Binding

Building a record viewer with the **DetailsView** control is easy and quick. You just drop an instance of the control onto the Web form, bind it to a data source control, and add a few decorative settings. The following listing shows the very minimum that’s needed:

```xml
<asp:DetailsView runat="server" id="RecordView" DataSourceID="MySource"HeaderText="Employees"></asp:DetailsView>
```

When the **AllowPaging** property is set to **true**, a pager bar is displayed for users to navigate between bound records. As you will see in more detail later, this works only if multiple records are bound to the control. Here’s a more realistic code snippet—the code behind the control in Figure 13-2:

```xml
<asp:ObjectDataSource ID="RowDataSource" runat="server" TypeName="Core35.DAL.Employees" SelectMethod="LoadAll"> </asp:ObjectDataSource>
<asp:DetailsView ID="RecordView" runat="server" DataSourceID="RowDataSource" AllowPaging="true"HeaderText="Northwind Employees"AutoGenerateRows="false"> <Fields> <asp:BoundField DataField="firstname" HeaderText="First Name" /> <asp:BoundField DataField="lastname" HeaderText="Last Name" /> <asp:BoundField DataField="title" HeaderText="Title" HtmlEncode="false" DataFormatString="{0:d}" /> </Fields> </asp:DetailsView>
```
Binding Data to a `DetailsView` Control

A `DetailsView` control is formed by data-bindable rows—one for each field in the displayed data item. By default, the control includes all the available fields in the view. You can change this behavior by setting the `AutoGenerateRows` property to `false`. In this case, only the fields explicitly listed in the `Fields` collection are displayed. Just as grids do, the `DetailsView` control can have both declared and auto-generated fields. In this case, declared fields appear first and auto-generated fields are not added to the `Fields` collection. The `DetailsView` supports the same variety of field types as the `GridView`. (See Chapter 11.)

If no data source property is set, the `DetailsView` control doesn't render anything. If an empty data source object is bound and an `EmptyDataTemplate` template is specified, the results shown to the user have a more friendly look:

```html
<asp:DetailsView runat="server" datasourceid="MySource">
  <EmptyDataTemplate>
    <asp:label runat="server">
      There's no data to show in this view.
    </asp:label>
  </EmptyDataTemplate>
</asp:DetailsView>
```

The `EmptyDataTemplate` property is ignored if the bound data source is not empty. If you simply plan to display a message to the user, you can more effectively resort to the `EmptyDataText` property. Plain text properties, in fact, are faster than templates.
Fields can be defined either declaratively or programmatically. If you opt for the latter, instantiate any needed data field objects and add them to the `Fields` collection, as shown in the following code snippet:

```csharp
BoundField field = new BoundField();
field.DataField = "companyname";
field.HeaderText = "Company Name";
detailView1.Fields.Add(field);
```

Rows in the control's user interface reflect the order of fields in the `Fields` collection. To statically declare your columns in the .aspx source file, you use the `<Fields>` tag.

**Note** If you programmatically add fields to the control, be aware of the view state. The field is not automatically added to the view state and won't be there the next time the page posts back. (This is the same snag we found in a previous chapter for the columns of a GridView or DataGrid control.) If some fields have to be added programmatically all the time, you put the code in the `Page_Load` event handler. If field insertion is conditional, after adding fields you write a custom flag to the view state. In `Page_Load`, you then check the view state flag and, if it is set, you add fields as expected.

### Controlling the Displayed Fields

Just as grid controls can display only a selected range of columns, the DetailsView control can display only a subset of the available fields for the current record. As mentioned, you disable the automatic generation of all fields by setting the `AutoGenerateRows` property to `false`. Then you declare as many fields as needed under the `<Fields>` element, as shown here:

```csharp
<asp:detailsview>
...
<fields>
    <asp:boundfield datafield="firstname" headertext="First Name" />
    <asp:boundfield datafield="lastname" headertext="Last Name" />
    <asp:boundfield datafield="title" headertext="Position" />
</fields>
</asp:detailsview>
```

The `HeaderText` attribute refers to the label displayed alongside the field value. You can style this text using the `FieldHeaderStyle` property. The following code makes field labels appear in boldface type:

```csharp
<FieldHeaderStyle Font-Bold="true" />
```

To improve the readability of displayed data, you select the field type that best suits the data to display. For example, Boolean data is better displayed through `CheckBoxField` rows, whereas URLs render the best via `HyperLinkField`. Admittedly, the list is not exhaustive, but the main issues won’t show up until you turn on the record in edit mode. By default, in fact, in edit or insert mode the content of the field is displayed using a text box, which is great for
many data types but not for all. For example, what if your users need to edit a date? In this case, the Calendar control would be far more appropriate. However, you can’t use templates to modify the default rendering because the DetailsView control doesn’t support data-bound templates on rows. You should resort to the FormView control if template support is an unavoidable necessity.

Paging Through Bound Data

The DetailsView control is designed to display one record at a time, but it allows you to bind multiple records. In a master/detail scenario, you really need to bind a single record. In a record-viewer scenario, you might find it useful to bind the whole cached data source and have the control to page through. The following paragraph details the rules for paging in the DetailsView control.

No paging is allowed if AllowPaging is set to false (the default setting). If AllowPaging is turned on, paging is allowed only if more than one record is bound to the control. When paging is possible, the pager is displayed to let users select the next record to view. Just as for grids, the pager can provide numeric links to the various records (the first, the third, the last, and so forth) as well as relative hyperlinks to the first, previous, next, or last record. The PagerSettings type determines the attributes and behavior of the pager bar. PagerStyle, on the other hand, governs the appearance of the pager.

The DetailsView paging mechanism is based on the PageIndex property, which indicates the index of the current record in the bound data source. Clicking any pager button updates the property; the control does the data binding and refreshes the view. PageCount returns the total number of records available for paging. Changing the record is signaled by a pair of events—PageIndexChanging and PageIndexChanged.

The PageIndexChanging event allows you to execute custom code before the PageIndex actually changes—that is, before the control moves to a different record. You can cancel the event by setting the Cancel property of the event argument class to true:

```csharp
void PageIndexChanging(object sender, DetailsViewPageEventArgs e)
{
    e.Cancel = true;
}
```

Note that when the event fires you don’t have much information about the new record being displayed. You can read everything about the currently displayed record, but you know only the index of the next one. To retrieve details of the current record, you proceed as you would with GridViews and use the DataKey property:

```csharp
DataKey data = DetailsView1.DataKey;
string country = (string) data.Values["country"];
if (country == "Mexico" || country == "USA" || country == "Brazil")
{
    ...
}
```
To be able to use the DataKey property within data-bound events, you must set the DataKeyNames property to the comma-separated list of fields you want to be persisted in the view state and exposed by the DataKey structure later:

```html
<asp:DetailsView ID="DetailsView1" runat="server"
    DataKeyNames="id, country"
    ...
/></asp:DetailsView>
```

It is essential that DataKeyNames contains public properties of the bound data type. In other words, id and country must be record fields if the DetailsView control is bound to a DataSet or DataTable. They must be property names if the DetailsView control is bound to a custom collection via ObjectDataSource.

There's no easy way to look up the next record from within the PageIndexChanging event. The simplest thing you can do is cache the data set bound to the DetailsView, get a reference to the cached data, and select in that list the record that corresponds to the index of the next page.

**Note** Paging with the DetailsView control is subject to the same paging issues for GridView and DataGrid that we examined in the previous chapter. If you bind the control to SqlDataSource, you're better off caching the data source; if you bind to ObjectDataSource, it is preferable that you use business objects that page themselves through the data source.

### Paging via Callbacks

Paging is normally implemented through a server-side event and requires a full page refresh. The DetailsView control provides the EnablePagingCallbacks property to specify whether paging operations are performed using client-side callback functions.

Paging callbacks are based on ASP.NET script callbacks, and when they are enabled, they prevent the need to post the page back to the server. At the same time, new data for the requested page is retrieved through an out-of-band call. The control is responsible for grabbing the server data and refreshing its own user interface on browsers that support a Dynamic HTML–compliant document object model.

For a developer, turning on the client paging feature couldn’t be easier. You just set the EnablePagingCallbacks property to true and you’re done.

**Note** As mentioned for the GridView control in Chapter 11, script callbacks are a functionality completely replaced by AJAX and partial rendering. We’ll cover this in Chapter 18.
Creating Master/Detail Views

In versions prior to ASP.NET 2.0, implementing master/detail views is not particularly hard to do, but it's certainly not automatic and codeless. In ASP.NET 2.0 and beyond, combining the DetailsView control with another data-bound control such as the GridView or DropDownList greatly simplifies the creation of master/detail views of data. The master control (such as the GridView) selects one particular record in its own data source, and that record becomes the data source for a DetailsView control in the same form. Let's see how.

Drill Down into the Selected Record

A typical master/detail page contains a master control (such as a GridView) and a detail control (such as a DetailsView), each bound to its own data source. The trick is in binding the detail control to a data source represented by the currently selected record. The following code snippet shows the configuration of the “master” block. It consists of a GridView bound to a pageable ObjectDataSource:

```xml
<asp:ObjectDataSource ID="CustomersDataSource" runat="server"
    EnablePaging="true"
    StartRowIndexParameterName="firstRow"
    MaximumRowsParameterName="totalRows"
    TypeName="Core35.DAL.Customers"
    SelectListMethod="LoadAll">
</asp:ObjectDataSource>
<asp:GridView ID="GridView1" runat="server"
    DataSourceID="CustomersDataSource"
    DataKeyNames="id"
    AllowPaging="True"
    AutoGenerateSelectButton="True"
    AutoGenerateColumns="False">
    <PagerSettings Mode="NextPreviousFirstLast" />
    <Columns>
        <asp:BoundField DataField="CompanyName" HeaderText="Company" />
        <asp:BoundField DataField="Country" HeaderText="Country" />
    </Columns>
</asp:GridView>
```

The grid shows a Select column for users to select the record to drill down into. However, you don't need to handle the corresponding SelectedIndexChanged event for the details view to kick in. The following code shows the “detail” block of the master/detail scheme:

```xml
<asp:ObjectDataSource ID="RowDataSource" runat="server"
    TypeName="Core35.DAL.Customers"
    SelectListMethod="Load">
    <SelectParameters>
        <asp:ControlParameter Name="id" ControlID="GridView1"
            PropertyName="SelectedValue" />
    </SelectParameters>
</asp:ObjectDataSource>
```
The DetailsView control is bound to the return value of the Load method on the Customer data access layer (DAL) class. The Load method requires an argument to be the ID of the customer. This parameter is provided by the grid through its SelectedValue property. Whenever the user selects a new row in the grid, the SelectedValue property changes (as discussed in Chapter 11), the page posts back, and the DetailsView refreshes its user interface accordingly. No code should be written in the code-behind class for this to happen.

Figure 13-3 shows the page in action when no row is selected in the grid. This is a great example for understanding the importance of the empty data row template.

Figure 13-4 shows the two controls in action when a record is selected.
FIGURE 13-4 The DetailsView control shows the details of the selected customer.

Note that the internal page mechanics places a call to the Load method at all times, even when the page first loads and there’s no record selected in the grid. Even when there’s no record selected, the Load method is passed the value of the SelectedValue property on the grid—which is null. What happens in this case? It depends on the implementation of the Load method. If Load can handle null input values and degrades gracefully, nothing bad happens and the page displays the empty data template. Otherwise, you typically get a runtime exception from the ADO.NET infrastructure in charge of retrieving data because of the invalid parameter you provided to the method. Here’s a good code sequence to use for methods with data source controls:

```csharp
public static Customer Load(string id)
{
    if (String.IsNullOrEmpty(id))
        return null;
    ...
}
```

Caching Issues

The preceding scheme for master/detail pages is easy to understand and arrange. Furthermore, you can design it through a full point-and-click metaphor directly in the Microsoft Visual Studio integrated development environment (IDE) without writing a single line of code. Could you possibly ask for more? Actually, what you should do is ensure that it works the way you want it to. Let’s delve a bit deeper into this type of automatic master/detail binding.

The grid is bound to a list of customers as returned by the data source control. As you would expect, this list is cached somewhere. If you use SqlDataSource, you can control caching to some extent through a bunch of properties. If you use ObjectDataSource as in the previous
example, you have no caching at all unless you instruct the Load method or, more generally, you instruct your DAL and business layer to cache data. All the data bound to the grid is retrieved from the database whenever the grid is paged or sorted. But there's more.

When the user selects a given record, the DetailsView gets bound to a particular record whose details are retrieved through another query. This repeated query might or might not be necessary. It might not be necessary if you're building a master/detail scheme on a single table (Customers, in this case) and if the "master" control already contains all the data. In the previous example, the LoadAll method that populates the grid returns a collection based on the results of SELECT [fields] FROM customers. In light of this, there would be no need for the DetailsView to run a second query to get details that could already be available, if only they were cached.

In summary, ObjectDataSource doesn't inherently support caching unless you use ADO.NET data containers. Generally speaking, caching is a performance booster if the overall size of cached data is limited to hundreds of records. If you can't get caching support from the data source control, build it in the business objects you use. If you use SqlDataSource, or ObjectDataSource with ADO.NET objects, enable caching, but keep an eye on the size of the cached data. And in all cases, use the SQL Server profiler (or similar tools if you use other database management systems) to see exactly when data is being retrieved from the database.

Working with Data

A detailed view like that of the DetailsView control is particularly useful if users can perform basic updates on the displayed data. Basic updates include editing and deleting the record, as well as inserting new records. The DetailsView command bar gathers all the buttons needed to start data operations. You tell the control to create those buttons by setting to true the following properties: AutoGenerateEditButton (for updates), AutoGenerateDeleteButton (for deletions), and AutoGenerateInsertButton (for adding new records).

Editing the Current Record

As with the GridView, data operations for the DetailsView control are handled by the bound data source control, as long as the proper commands are defined and a key to identify the correct record to work on is indicated through the DataKeyNames property. Let's test SqlDataSource first:

```xml
<asp:SqlDataSource runat="server" id="SqlDataSource1"
    ConnectionString="@ConnectionStrings:NWind %"
    SelectCommand="SELECT * FROM customers"
    UpdateCommand="UPDATE customers SET
companyname=@companyname, contactname=@contactname,
city=@city, country=@country
WHERE customerid=@original_customerid"
    DeleteCommand="DELETE customers WHERE customerid=@original_customerid"
/>
Part II Adding Data in an ASP.NET Site

```xml
<asp:DetailsView ID="DetailsView1" runat="server"
    DataKeyNames="customerid"
    DataSourceID="SqlDataSource1"
    AllowPaging="True"
    AutoGenerateRows="False"
    HeaderText="Customers"
    AutoGenerateEditButton="True"
    AutoGenerateDeleteButton="True">
    <PagerSettings Mode="NextPreviousFirstLast" />
    <Fields>
        <asp:BoundField DataField="CompanyName" HeaderText="Company" />
        <asp:BoundField DataField="ContactName" HeaderText="Contact" />
        <asp:BoundField DataField="City" HeaderText="City" />
        <asp:BoundField DataField="Country" HeaderText="Country" />
    </Fields>
</asp:DetailsView>
```

The SqlDataSource must expose SQL commands (or stored procedures) for deleting and updating records. Once this has been done, the DetailsView control does all the rest. Users click to edit or delete the current record and the control ultimately calls upon the underlying data source to accomplish the action.

Figure 13-5 shows the changed user interface of the DetailsView control when it works in edit mode. Note that in edit mode, the default set of buttons in the command is replaced by a pair of update/cancel buttons.

![Figure 13-5 A DetailsView control working in edit mode.](image-url)
If you attach the DetailsView control to an ObjectDataSource control, make sure you properly bind the update and delete methods of the business object:

```xml
<asp:ObjectDataSource runat="server" id="ObjectDataSource1"
    TypeName="Core35.DAL.Customers"
    SelectMethod="LoadAll"
    DeleteMethod="Delete"
    UpdateMethod="Save"
    DataObjectTypeName="Core35.DAL.Customer"/>
```

The DataKeyNames property must be set to the name of the public property that represents the key for identifying the record to delete or update.

### Deleting the Current Record

Although the delete operation can be pre- and post-processed by a pair of events such as ItemDeleting/ItemDeleted, there's not much a page author can do to give users a chance to recall an inadvertently started delete operation. The bad news is that unlike other data-bound controls, the DetailsView doesn't offer easy-to-use events and properties for you to override default behaviors. You might think that the ItemCreated event is the right place to handle the interception of the command bar creation and add some script code to the delete button. ItemCreated is still the right (the only, actually) entry point in the control's machinery, but adding a client-side message box to the delete button is a difficult challenge. ItemCreated fires whenever a DetailsView row is being created, but it doesn't supply additional information about the newly created row. Furthermore, the command row is not exposed through a direct property as is the case with a pager, header, and footer. A trick is needed to access the row representing the command bar. If you turn tracing on and snoop into the contents of the Rows collection while debugging, you can easily figure out that the Rows collection contains as many elements as there are data rows in the control, plus one. The extra row is actually the command bar. You get it with the following code:

```csharp
protected void DetailsView1_ItemCreated(object sender, EventArgs e)
{
if (DetailsView1.FooterRow != null)
{
    int commandRowIndex = DetailsView1.Rows.Count - 1;
    DetailsViewRow commandRow = DetailsView1.Rows[commandRowIndex];
    ...
}
}
```
To be sure that your code kicks in when the command bar exists, you check the FooterRow property for nullness. The footer row is always created regardless of whether it is displayed or not; in addition, it is always created after all the data rows have been created. The command bar is the last row in the Rows collection and is an object of type DetailsViewRow—a special type of table row. The row contains a cell—an internal object of type DataControlFieldCell—which in turn contains edit, delete, and insert buttons. The tracing tool reveals that buttons are not plain buttons, but instances of the internal DataControlLinkButton class, a class derived from LinkButton. You get the cell with the following code:

```csharp
DataControlFieldCell cell;
cell = (DataControlFieldCell) commandRow.Controls[0];
```

At this point, you're pretty much done. What remains for you to do is loop through all the child controls of the cell and get a reference to all link buttons. How can you distinguish the delete button from the edit button? What if one of these controls is not enabled? Link buttons have the CommandName property, which assigns them a characteristic and unique name—Delete, Edit, or New for the data operations we're interested in here. Have a look at the following code:

```csharp
protected void DetailsView1_ItemCreated(object sender, EventArgs e)
{
    if (DetailsView1.FooterRow != null)
    {
        int commandRowIndex = DetailsView1.Rows.Count-1;
        DetailsViewRow commandRow = DetailsView1.Rows[commandRowIndex];
        DataControlFieldCell cell;
cell = (DataControlFieldCell) commandRow.Controls[0];
        foreach (Control ctl in cell.Controls)
        {
            LinkButton link = ctl as LinkButton;
            if (link != null)
            {
                if (link.CommandName == "Delete")
                {
                    link.ToolTip = "Click here to delete";
                    link.OnClientClick = "return confirm('Do you really want to delete this record?');";
                }
                else if (link.CommandName == "New")
                {
                    link.ToolTip = "Click here to add a new record";
                }
                else if (link.CommandName == "Edit")
                {
                    link.ToolTip = "Click here to edit the current record";
                }
            }
        }
    }
}
```
Once you have received a valid reference to the link button that represents, say, the delete button, you can do whatever you want—for example, add a ToolTip and a JavaScript confirm popup. (See Figure 13-6.)

**FIGURE 13-6** A popup asking for confirmation before you delete the current record.

Inserting a New Record

The process of adding a new record is much like the process for editing or deleting. You add an insert command in the bound data source control and enable the insert button on the DetailsView control. Here is a valid insert command:

```xml
<asp:SqlDataSource ID="SqlDataSource1" runat="server"
     EnableCaching="true"
     ConnectionString='<%$ ConnectionStrings:NWind %>'
     SelectCommand="SELECT employeeid, firstname, lastname, title, hiredate FROM employees"
     InsertCommand="INSERT INTO employees (firstname, lastname, title, hiredate) VALUES (@firstname, @lastname, @title, @hiredate)"
 />
<asp:DetailsView ID="DetailsView1" runat="server"
     AllowPaging="true"
     DataSourceID="SqlDataSource1"
     AutoGenerateInsertButton="true"
     HeaderText="Employee Details"
     <PagerSettings Mode="NextPreviousFirstLast" />
</asp:DetailsView>
```
Figure 13-7 shows how it works.

When you implement the insert command, you should pay attention to primary keys. In particular, the preceding command doesn’t specify the primary key (employeeid) because, in this example, the underlying database auto-generates values for the field. Generally, for a database that accepts user-defined keys, you should provide a validation mechanism in the page before you push the new record. Once again, all this code is best placed in the DAL and bound to DetailsView through an ObjectDataSource control. I’ll say more on input data validation in a moment.

**Templated Fields**

The DetailsView control doesn’t support edit and insert templates to change the layout of the control entirely. When editing the contents of a data source, you either go through the standard layout of the user interface—a vertical list of header/value pairs—or resort to another control, such as the FormView or a custom control. Designed to be simple and effective, the DetailsView turns out to be somewhat inflexible and hard to hook up. As seen earlier, walking your way through the internal object model of the DetailsView control is not impossible. The real problem, though, is forcing the control to play by rules that it hasn’t set.

You can change, instead, the way in which a particular field is displayed within the standard layout. For example, you can use a Calendar control to render a date field, or a DropDownList control to display multiple options. To do this, you employ the TemplateField class, as we did for grid controls. By using a TemplateField class to render a data field, you are free to use any layout you like for view, edit, and insert operations, as shown here:

```<asp:TemplateField HeaderText="Country">
  <ItemTemplate>
    <asp:literal runat="server" Text='<%# Eval("country") %>' />
  </ItemTemplate>
</asp:TemplateField>```
The field **Country** is rendered through a literal in view mode, and turns to a data-bound drop-down list control in edit mode. The bound data source control is responsible for providing all the displayable countries. The **Bind** operator is like **Eval** except that it writes data back to the data source—this is the power of ASP.NET two-way data binding. (See Chapter 10.) Figure 13-8 shows a sample page.

![Templated fields in a DetailsView control.](image)

**FIGURE 13-8** Templated fields in a DetailsView control.

### Adding Validation Support

By using templated fields, you can also add any validator control you need and add it where you need it. What if you don’t want templated fields? Limited to validator controls, there’s an alternate approach. In this way, you still use **BoundField** controls to render fields, but you attach validators to them programmatically.
You start by adding an `ItemCreated` event handler to the `DetailsView` control in the page, as follows:

```csharp
protected void DetailsView1_ItemCreated(object sender, EventArgs e)
{
    if (DetailsView1.CurrentMode == DetailsViewMode.ReadOnly)
        return;
    if (DetailsView1.FooterRow == null)
        return;
    AddRequiredFieldValidator(0, "First name required");
    AddRequiredFieldValidator(1, "Last name required");
}
```

First you ensure the control is in edit mode and all the data rows have been created. Next, you assume you know the ordinal position of the fields you want to modify. (This is a reasonable assumption, as we’re not designing a general-purpose solution, but simply adjusting a particular ASP.NET page that we created.)

The `AddRequiredFieldValidator` method takes the index of the field you want to validate and the message to display in case the field is left blank. It instantiates and initializes a validator, and then adds it to the corresponding cell, as in the following code:

```csharp
void AddRequiredFieldValidator(int rowIndex, string msg)
{
    // Retrieve the data row to extend
    const int DataCellIndex = 1;
    DetailsViewRow row = DetailsView1.Rows[rowIndex];
    // Get the second cell—the first contains the label
    DataControlFieldCell cell = (DataControlFieldCell) row.Cells[DataCellIndex];
    // Initialize the validator
    RequiredFieldValidator req = new RequiredFieldValidator();
    req.Text = String.Format("<span title='{0}'>*</span>", msg);
    // Get the ID of the TextBox control to validate
    string ctlID = cell.Controls[0].UniqueID;
    int pos = ctlID.LastIndexOf("$");
    if (pos < 0)
        return;
    string temp = ctlID.Substring(pos + 1);
    req.ControlToValidate = temp;
    // Insert the validator
    cell.Controls.Add(req);
}
```

You retrieve the data row to extend with a validator control and get a reference to its second cell. A `DetailsView` row has two cells—one for the field header and one for the field value. In edit/insert mode, the second cell contains a `TextBox` control.
The validator control—a RequiredFieldValidator in this example—requires some behavior settings (say, the message to display). More importantly, it requires the ID of the control to validate. Nobody knows the ID of the dynamically generated TextBox control. However, you can get a reference to the control and read the UniqueID property, and here's why that's important.

The DetailsView is a naming container, which means that it prefixes the names of contained controls. For example, an internal TextBox named, say, ctl01 is publicly known as DetailsView1$ctl01, where DetailsView1 is the ID of the DetailsView control. You need to pass the real control’s ID to the validator. That’s why the preceding code locates the last occurrence of the $ symbol and discards all that precedes it. The equivalent of ctl01 is finally assigned to the ControlToValidate property of the validator, and the validator is added to the cell.

You have added a new control with its own behavior, and you have no need to interact with the remainder of the host control. In this case, it works just fine, as shown in Figure 13-9.

![Figure 13-9 Validation support added to a DetailsView control.](image-url)
The preceding code always displays an asterisk to signal an incomplete field. The actual text is wrapped by a `<span>` tag to include a ToolTip. This is arbitrary; you can configure the validator control at your leisure.

**Validating Without Validators**

So far we considered two different scenarios for validating data manipulated with the DetailsView control. In the first, you employ templates and explicitly add validator controls. In the second, you stick to nontemplated bound fields but use a slick piece of code to add validators programmatically. It is important to mention that there’s also a simpler, and perhaps more natural, way of approaching the problem of validating data—using events.

You write a handler for the ItemUpdating event (ItemInserting or ItemDeleting for insert and delete operations, respectively), check the new values and cancel the operation if there’s something wrong. The following code ensures that the title field contains one of two hard-coded strings:

```csharp
void DetailsView1_ItemUpdating(object sender, DetailsViewUpdateEventArgs e)
{
    string title = (string) e.NewValues["title"];
    if (!title.Equals("Sales Representative") &&
        !title.Equals("Sales Manager"))
    {
        e.Cancel = true;
    }
}
```

The `NewValues` dictionary you get through the event data contains new values as edited by the user; the `OldValues` dictionary contains the original data. What’s the difference between this approach and validators? ItemUpdating (and similar events) are run on the server during the postback event. Validators can catch patently invalid input data already on the client. However, a golden rule of validation states that you should never rely on client-side validation only. You should always do some validation on the server, so performance is not the issue here. The event-based approach is easier to set up and is ideal for quick pages where you don’t bother using a more advanced and templated user interface. Validators are a more complete toolkit for validation and, of course, include a control for server-side validation as we saw in Chapter 4. The validation-without-validators scheme can be applied for any view controls—DetailsView, FormView, and GridView.
The **FormView Control**

*FormView* is a new data-bound control that can be considered the templated version of the *DetailsView*. It renders one record at a time, picked from the associated data source and, optionally, provides paging buttons to navigate between records. Unlike the *DetailsView* control, *FormView* doesn't use data control fields and requires the user to define the rendering of each item by using templates. The *FormView* can support any basic operation its data source provides.

Note that the *FormView* requires you to define everything through templates, and not just the things you want to change. The *FormView* has no built-in rendering engine and is limited to printing out the user-defined templates.

The **FormView Object Model**

Two functional aspects mark the difference between *FormView* and *DetailsView*. First, the *FormView* control has properties such as *ItemTemplate*, *EditItemTemplate*, and *InsertItemTemplate* that—as we've seen thus far—the *DetailsView* lacks entirely. Second, the *FormView* lacks the command row—that is, a sort of toolbar where available functions are grouped. The graphical layout of a *FormView* control is completely customizable using templates. Therefore, each template will include all command buttons needed by the particular record.

The control's definition is shown in the following code:

```csharp
public class FormView : CompositeDataBoundControl, IDataItemContainer, INamingContainer
```

As you can see, *FormView* has the same root and implements the same interfaces as *DetailsView* except for the interfaces related to ASP.NET script callbacks.

Members of the **FormView Control**

The *FormView* control exposes many of the properties that we've seen already for the *DetailsView* control. This aspect is no surprise, as the two controls serve as two sides of the same coin—a record viewer control—one with and one without templates. Only the templates and related styles mark the difference between *FormView* and *DetailsView*. You can refer to Table 13-1 through Table 13-6 for the complete list of properties and events supported by the *FormView* control.
Part II Adding Data in an ASP.NET Site

Supported Templates

The output of the FormView control is exclusively based on templates. This means that you always need to specify the item template at a very minimum. Table 13-7 shows the list of data-bound supported templates.

**TABLE 13-7 Templates of the FormView Control**

<table>
<thead>
<tr>
<th>Template</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EditTextTemplate</td>
<td>The template to use when an existing record is being updated</td>
</tr>
<tr>
<td>InsertTextTemplate</td>
<td>The template to use when a new record is being created</td>
</tr>
<tr>
<td>ItemTemplate</td>
<td>The template to use when an existing record is rendered for viewing only</td>
</tr>
</tbody>
</table>

It’s not a coincidence that the FormView templates match the three feasible states of the control—ReadOnly, Edit, and Insert. You use ItemTemplate to define the control’s layout when in view mode. You use EditTextTemplate to edit the contents of the current record, and you use InsertTextTemplate to add a new record.

In addition to these templates, the control features the same set of templates offered by the DetailsView—that is, HeaderTemplate, FooterTemplate, and the other templates listed in Table 13-5.

Supported Operations

Because the user interface of the control is largely defined by the page author, you cannot expect a FormView control to understand the click on a particular button and act accordingly. For this reason, the FormView control exposes a few publicly callable methods to trigger common actions, such as those listed in Table 13-8.

**TABLE 13-8 Methods of the FormView Control**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChangeMode</td>
<td>Changes the working mode of the control from the current one to any of the modes defined in the FormViewMode type—ReadOnly, Edit, or Insert.</td>
</tr>
<tr>
<td>DeleteItem</td>
<td>Deletes the current record in the FormView control from the data source.</td>
</tr>
<tr>
<td>InsertItem</td>
<td>Inserts the current record in the data source. The FormView control must be in insert mode when this method is called; otherwise, an exception is thrown.</td>
</tr>
<tr>
<td>UpdateItem</td>
<td>Updates the current record in the data source. The FormView control must be in edit mode when this method is called; otherwise, an exception is thrown.</td>
</tr>
</tbody>
</table>

Both InsertItem and UpdateItem require a Boolean indicating whether or not input validation should be performed. In this context, performing validation simply means that any validator
controls you might have in the template will be called. If no validators are found, no other form of validation ever occurs. The InsertItem and UpdateItem methods are designed to start a basic operation from within controls in any of the supported templates. You don’t have to pass the record to insert, the values to update, or the key of the record to delete. The FormView control knows how to retrieve that information internally in much the same way the DetailsView does.

The DeleteItem, InsertItem, and UpdateItem methods let you define your own delete, insert, and edit user interface and attach it to the standard data-binding model of ASP.NET controls. In the DetailsView control, this association is implicit because the user interface is relatively static and fixed; in the FormView, the same association must be explicitly defined in light of the totally customizable user interface.

**Binding Data to a FormView Control**

Let’s see how to use templates to configure and run a FormView control in a sample ASP.NET Web page. All templates must contain everything needed to accomplish tasks—user interface elements and command buttons. The control itself provides the pager bar and the surrounding table.

**Header, Footer, and Pager**

The final output of the FormView control takes the form of an HTML table with a header and footer row, plus an optional row for the pager. Just like the DetailsView, the FormView control provides templates for the header and footer; unlike the DetailsView, though, it doesn’t provide simpler and handy text properties such asHeaderText and FooterText.

```xml
<asp:FormView ID="FormView1" runat="server" AllowPaging="true" DataSourceID="CustomersDataSource">
    <PagerSettings Mode="NextPreviousFirstLast" />
    <HeaderTemplate>
        <h1>Customer Viewer</h1>
    </HeaderTemplate>
    <FooterTemplate>
        <h3>Courtesy of "Programming ASP.NET"</h3>
    </FooterTemplate>
</asp:FormView>
```

The pager is dual in the sense that you can have the control to render it as the settings established through PagerSettings and PagerStyle properties dictate, or create it from scratch via the PagerTemplate property.
Displaying Data

The following code snippet shows the typical layout of the code you write to embed a FormView in your pages:

```csharp
<asp:FormView ID="FormView1" runat="server"
    DataSourceId="CustomersDataSource" AllowPaging="true">
    <ItemTemplate>
        ...
    </ItemTemplate>
    <EditItemTemplate>
        ...
    </EditItemTemplate>
    <InsertItemTemplate>
        ...
    </InsertItemTemplate>
</asp:FormView>

The following code generates the page shown in Figure 11-9:

```csharp
<asp:FormView runat="server" id="FormView1"
    DataKeyNames="employeeid"
    DataSourceID="MySource" AllowPaging="true">
    <ItemTemplate>
        <table style="border:solid 1px black;" width="100%">
            <tr>
                <td bgcolor="yellow" width="50px" align="center">
                    <b><%# Eval("id") %></b></td>
                <td bgcolor="lightyellow">
                    <b><%# Eval("companyname") %></b></td>
            </tr>
            <table style="font-family:Verdana;font-size:8pt;">
                <tr>
                    <td><b>Contact</b></td>
                    <td><%# Eval("contactname") %></td>
                </tr>
                <tr>
                    <td><b>City</b></td>
                    <td><%# Eval("city") %></td>
                </tr>
                <tr>
                    <td valign="top"><b>Country</b></td>
                    <td><%# Eval("country") %></td>
                </tr>
            </table>
        </table>
    </ItemTemplate>
</asp:FormView>
```
FIGURE 13-10 A FormView control in action.

All the markup you place in the ItemTemplate is rendered in a table cell that spans two columns. As mentioned, the overall layout of the FormView is a table.

```html
<td colspan="2">
...
</td>
```

If you want to obtain a tabular output, feel free to define an inner table, as in the preceding code.

The Edit button is added using a classic `<asp:Button>` button with the Edit command name. The command name will cause the FormView to automatically switch from the read-only mode to edit mode and display using the edit item template, if any is defined. You can use any button control with whatever command name and caption you like. If it doesn’t change mode automatically, you call `ChangeMode` and the other methods supported by the FormView control.

**The Eval Function**

How can you insert data fields in a template? You resort to data-binding expressions and, in particular, use the `Eval` function:

```html
<td><%# Eval("city") %></td>
```

As mentioned in Chapter 10, `Eval` exists in two forms, one of which is also supported in ASP.NET 1.x. The two forms are functionally equivalent, as one of them is implemented in terms of the other. The first form you can use is the following:

```html
<%# DataBinder.Eval(Container.DataItem, "city")%>
```
Part II Adding Data in an ASP.NET Site

The static function `Eval` on the `DataBinder` class uses reflection to parse and evaluate a data-binding expression against an object at run time. The object it works with is the data item object from the bound data source that corresponds to the record being rendered. Most of the time, the data-binding expression will be the name of a property on the data item bound to a row. The `Eval` function has a third overload to specify a format string.

In ASP.NET 2.0 and beyond, a similar function is available that has a more compact syntax—the `Eval` function defined on the `TemplateControl` class and inherited by all ASP.NET pages. `Eval` is an instance function and accepts only the data-binding expression, and optionally a format string. The `Eval` function ends up calling into the `DataBinder.Eval` function.

The `Eval` is useful only in read-only, one-way data-binding scenarios. For implementing real two-way data binding, an extension to `Eval` is required—the `Bind` function, which we’ll discuss in a moment.

Editing Data

To edit bound records, you define an ad hoc edit template through the `EditItemTemplate` property. You can place on the form any combination of input controls, including validators. You are not limited to using text boxes and can unleash your imagination to build the user interface.

How do you retrieve values to update the bound record? You enable two-way binding by using the newest `Bind` function in lieu of `Eval`.

The Edit Template

The following code snippet shows a sample `TextBox` control bound to the `companyname` property of the data source. This is the key difference between item and edit item templates.

```html
<asp:TextBox runat="server" Text='<%# Bind("companyname") %>' />
```

The following code snippet shows a sample edit template. It contains quite a few standard text boxes but also a data-bound drop-down list.

```html
<EditItemTemplate>
<table style="border:solid 1px black;" width="100%">
  <tr>
    <td bgcolor="yellow" align="center"><b><%# Eval("id") %></b></td>
    <td bgcolor="lightyellow">
      <asp:textbox runat="server" text='<%# Bind("companyname") %>' />
    </td>
  </tr>
</table>
<table style="font-family:Verdana;font-size:8pt;">
  <tr>
    <td bgcolor="yellow" align="center"><b><%# Eval("id") %></b></td>
    <td bgcolor="lightyellow">
      <asp:textbox runat="server" text='<%# Bind("companyname") %>' />
    </td>
  </tr>
</table>
```

You use `Eval` to populate control properties not involved in the update process. Wherever you need two-way data binding—that is, read/write capabilities—you use the `Bind` function instead of `Eval`, with the same syntax. For text boxes, you bind the `Text` property; for drop-down lists, you typically bind the `SelectedValue` property. Other controls would bind to their respective data properties. For example, when using a `Calendar` control you would bind to its `SelectedDate` property.

How would you populate a data-bound drop-down list? You would do it by using another data source control, properly configured and parameterized to retrieve its data based on any input that proves necessary. In the sample code, you bind the drop-down list with all possible countries. Similarly, you might bind an employee ID field to the list of all employees from an external, foreign data source.

At last, bear in mind that the edit template must contain buttons to save changes. These are ordinary buttons with specific command names—`Update` to save and `Cancel` to abort. Buttons trigger update commands whose details are stored in the associated data source object. You can choose any text for the captions as long as you don’t change the command names. If you want to modify the command names, be prepared to handle the `ItemCommand` event on the `FormView` and call the `UpdateItem` method in response.

Figure 13-11 demonstrates the output of this code.
Part II Adding Data in an ASP.NET Site

For the update command to work, the DataKeyNames property must be set on the FormView to identify the key field. For deleting a record, just add a button with the Delete command name and configure the underlying data source control.

The Bind Function

How does the Bind function work? The function stores the value of the bound control property into a collection of values that the FormView control automatically retrieves and uses to compose the parameter list of the edit command.

The argument passed to Bind must match the name of a parameter in the update command or method or one of the properties on the type used as an argument to the update method. This is the case in the example shown earlier, where the update method takes an instance of the Core35.DAL.Customer class. An exception is raised if no parameter match is found.

The Insert Template

The InsertItemTemplate property allows you to define the input layout when a new record is being added. To avoid confusion, an insert template should not be much different from an edit template. At the same time, you should be aware that edit and insert are distinct operations with different requirements. For example, an insert template should provide default values to controls wherever that is acceptable, and it should display neutral or null values elsewhere.

To start an insert operation, you also need a button with a command name of New. Clicking on this button will force the FormView control to change its mode to Insert and render the
When the Function Is Not Supported

Both DetailsView and FormView controls expose some predefined operations—such as Insert, Edit, and Delete. As we’ve seen thus far, these operations are implemented inside the data source control bound to the view control. What if, say, the Edit function is enabled on the FormView control but not supported by the underlying data source control? When this happens, a NotSupportedException exception is thrown and the application fails.

It’s hard to imagine a team of developers who release some production code with an Edit button associated with a non-updateable data source. However, checking if a requested function is available is a good measure to make any application more robust and stable.

The following code demonstrates this feature:

```csharp
if (e.CommandName == "Edit")
{
    IDataSource obj = (IDataSource) FindControl(FormView1.DataSourceID);
    DataSourceView view = obj.GetView("DefaultView");
    if (!view.CanUpdate)
    {
        Response.Write("Sorry, you can't update");
        return;
    }
    else
        FormView1.UpdateItem();
}
```

The code retrieves the data source control and obtains a reference to its default data source view object. At this point, it checks whether the requested functionality is available. You can place this code in the ItemCommand event handler of any view controls—DetailsView, FormView, or GridView. Note that all ASP.NET built-in data source controls have only one view, named DefaultView.

Conclusion

Starting with ASP.NET 2.0, the developer’s toolbox for data-binding operations is definitely richer and more complete than in previous versions. You not only have a new and radically revised grid control, but you also have two other controls to manage views of a single record. There’s nothing like this in ASP.NET 1.x.

The DetailsView and FormView controls are two sides of the same coin. Both offer a user interface to see the contents of a single record. In both cases, the user interface is largely customizable and associated with predefined data operations such as delete, update, and insert. Bound to a data source control, both DetailsView and FormView can manage an underlying data source effectively without forcing developers to write ad hoc code. (Well, this is not...
entirely true. If you expose your data source through a DAL—as recommended for large systems—you have to write that code at least.)

The key difference between DetailsView and FormView lies in the support for templates. The former is perhaps a richer control with good basic support for templates limited to individual fields, and it has a relatively rich set of styles and visual properties. If you want to create your own form to edit and insert records, you should use the FormView control. If you do so, though, forget about standard rendering—a form view, in fact, is 100-percent templated and requires you to specify every single byte of markup.

With this chapter, we conclude the second part of the book—the part devoted to data access and related tools. Starting with the next chapter, we’ll begin a new trip in the ASP.NET infrastructure, one that shows you how to make pages and applications run.

Just The Facts

- Both the DetailsView and FormView controls render a single record at a time from the associated data source, optionally providing paging buttons to navigate between records.
- Both the DetailsView and FormView controls lend themselves very well to implement master/detail views.
- The DetailsView control has a fixed tabular layout and is formed by a few main areas—header, field rows, pager bar, command bar, and footer.
- The FormView control has areas such as header, footer, pager plus a completely templated item area. You can define a custom form to render the contents of a single record.
- The DetailsView control typically uses text boxes to render fields. If a particular field defines a template, any markup can be displayed. This feature is useful for representing dates and foreign keys through ad hoc controls such as calendars and drop-down lists.
- Both the DetailsView and FormView controls support basic I/O operations such as insert, delete, and update. If bound to data source controls such as ObjectDataSource or SqlDataSource, they leverage the capabilities of the bound data source control to execute data-binding operations.
- Both the DetailsView and FormView controls support two-way data binding, through which data can not only be automatically read from a bound data source but also written back.
- During updates and insertions, validation is possible in either of two ways. If templates are used, you can insert validator controls to sanitize the input both on the client and the server. If templates are not used, you can intercept events fired by DetailsView and FormView as well as control values being passed, and modify them at will or just cancel the operation.