



<http://www.devx.com>

Printed from <http://www.devx.com/vb2themax/Article/19908/1954>

Dynamic Templates for the Repeater, DataList and DataGrid Controls

This article explains why a single template hard-coded at design-time is not always the best option when using DataGrid, DataList and Repeater controls that must generate different outputs according to the logged-in user, a user's choice, or any other condition. It then provides detailed solutions to implement dynamic templates, by writing external user controls, or classes that implement the ITemplate interface.

by Marco Bellinaso

Most of the time, when you add a Repeater, DataList or DataGrid control on the page, you hardcode their templates: you know the data you want to show, and to do this you declare Literal, Label, CheckBox and any other type of control with bindable properties, or simply add static HTML and binding expressions.

This is fine as long as you have to render only one view of the data, but what happens when you want to use the same template-based control (i.e. one of the controls mentioned above) for showing different data, according to the currently logged-in user? Suppose for example that you have to build a report page for the Northwind's Employees table, and want to show the complete employee's information (name, full mailing address, salary etc.) if the current user is an Administrator, or just the employee's name and title otherwise, so to protect their privacy from non-authorized people. Another example: you may want to provide Edit and Delete buttons to each row when the user is an Administrator, but of course not for the normal user.

Yet another example: it may be the users themselves to decide whether they want to see some data in the report. Maybe the user (even if an administrator) doesn't want to see the employees' sensitive information, because she wants to print the report and distribute it among other people not authorized to see that data, or simply because she wants to keep the report as short and clean as possible, so it's easier to read.

A possible solution is to bind the Visible property of the control declared inside the template to a function that returns True or False according to the current user's roles, to the state of a checkbox/option button on the page, or to any other custom rule. For example, here's how to make a delete button and the employee's address visible to administrators only (users of the Windows' BUILTIN\Administrators group):

```
<asp:DataList Runat="server" ...>
  <ItemTemplate>
    <asp:Button runat="server" Text="Edit" CommandName="Edit"
      Visible='<%= ShowAdminControls() %>' />
    <%= Container.DataItem("TitleOfCourtesy") %>
    <%= Container.DataItem("LastName") %>,
    <%= Container.DataItem("LastName") %>
    <asp:Label runat="Server"
      Text='<%= <br> & Container.DataItem("LastName") %>'
      Visible='<%= ShowAdminControls() %>' />
  </ItemTemplate>
</asp:DataList>
```

in the code-behind file, or the server-side script section

```
Function ShowAdminControls() As Boolean
  Return User.Identity.IsAuthenticated AndAlso _
    User.IsInRole("BUILTIN\Administrators")
End Function
```

Although using many binding expressions can slow down the page's execution makes the code in the ASPX file more difficult to read, this solution can be ok if you have to handle just a couple of possible report's view and you don't have to dynamically hide/show many controls, but just a couple of buttons or labels, as in the last example.

However, if you need to render different outputs for more than two groups of users, and also according their options, this solution will soon become an hell to manage! You'll end up having a single huge template section with lots of controls and lots of binding expressions to decide what must be shown and what must be hidden not an ideal solution, also with regards to maintainability and readability. And don't forget the worse performances: the binding expressions will be evaluated for every single data source's item bound to the control, and this can be thousands of times if you have a few hundred records but dozens of controls that must be visible according to dynamic rules!

A much better solution does fortunately exist, and it is probably simpler than what you may think, especially considering the technique explained above.

Dynamically loading templates from external files

If you need different templates for different users and options, you can easily define the content for these templates in multiple separate file, and then load them dynamically at runtime, according to your own business rules! What you put in these files is almost the same you'd put into the control's `xxxTemplate` section (it works not only for the `ItemTemplate`, but also for the `HeaderTemplate` etc.), what changes is only the way you refer to the row's data item. First, this is how to declare the `DataList` control that will be used for the template files we'll see shortly:

```
<asp:DataList ID=Datalist1 runat="server" Width="100%"
  AlternatingItemStyle-ForeColor="Maroon"
  AlternatingItemStyle-BackColor="LightYellow"
  ItemStyle-ForeColor="DarkBlue"
  ItemStyle-BackColor="LightCyan"
  ...other style formatting...>
  <HeaderTemplate>Employees</HeaderTemplate>
  <ItemTemplate></ItemTemplate>
  <FooterTemplate></FooterTemplate>
</asp:DataList>
```

As you see, the only particular thing is that the `ItemTemplate` section is completely empty. In reality, we could even avoid to declare it, but without it the Visual Studio .NET's visual designer would not be able to render the control, and would show a gray box instead, as it does when adding user control. At this point we have to write the template files. In the first template file, `DataList_SimpleTemplate.ascx`, we paste some of the code of the original `DataList`'s `ItemTemplate` template: the three simple binding expressions that render the employee's title of courtesy, last and first names, as shown in the first example. We should add a `@Control` directive, as this is actually a user control file. We must also cast the `Container` object to `DataListItem`, because the template is defined outside the `DataList`, and thus the `Container` reference is of a general object type without the required `DataItem` property. After making these changes, you will have the following code:

```
<%@ Control %>
<%# DataBinder.Eval( _
  CType(Container, DataListItem).DataItem, "TitleOfCourtesy") %>
<b>
<%# DataBinder.Eval(CType(Container, DataListItem).DataItem, "LastName") %>
</b>,
<%# DataBinder.Eval(CType(Container, DataListItem).DataItem, "FirstName") %>
```

The code above uses Visual Basic .NET. If we were using C# we should specify this in the `@Control` directive, as we would do with a page. The C# code would be as follows:

```
<%@ Control Language="C#" %>
<%# DataBinder.Eval(
  ((DataListItem)Container).DataItem, "TitleOfCourtesy") %>
<b>
<%# DataBinder.Eval(((DataListItem)Container).DataItem, "LastName") %>
</b>,
<%# DataBinder.Eval(((DataListItem)Container).DataItem, "FirstName") %>
```

It looks similar, but of course there is a different syntax for type casting. The second template (we'll only see the VB.NET from now on), saved in `DataList_TemplateEx.ascx`, is similar: it merely adds some data, namely the content of the `Address`, `City`, and `Region` fields of the current record. Here's its complete code:

```
<%@ Control %>
<font face="Verdana">
<%# DataBinder.Eval( _
  CType(Container, DataListItem).DataItem, "TitleOfCourtesy") %>
<b>
<%#DataBinder.Eval(CType(Container, DataListItem).DataItem, "LastName") %>
</b>,
<%# DataBinder.Eval(CType(Container, DataListItem).DataItem, "FirstName") %>
<br>
<%# DataBinder.Eval(CType(Container, DataListItem).DataItem, "Title") %>
<br>
<%# DataBinder.Eval(CType(Container, DataListItem).DataItem, "Address") %> -
<%# DataBinder.Eval(CType(Container, DataListItem).DataItem, "City") %>,
<%# DataBinder.Eval(CType(Container, DataListItem).DataItem, "Region") %>
</font>
```

Once the template files are ready, we can load them when the page loads, according to some rules. In this example, we'll load and associate to the `DataList` the `DataList_TemplateEx.ascx` template if the current user is in the Window's Administrators

group, and the other template otherwise. As you should already know, to being asked a username and password and using Windows security you must first disable the anonymous access and enable the Basic Authentication options in the IIS virtual directory's Properties page. Once we have the virtual path of the template file, we load it by calling the Page's LoadTemplate method, and assign the returned object to the DataList's ItemTemplate property. The method below shows all this:

```
Private Sub Page_Load() Handles MyBase.Load
    ' load the proper templates according to the logged-in user
    If User.IsInRole("BUILTIN\Administrators") Then
        Datalist1.ItemTemplate = Page.LoadTemplate("DataList_TemplateEx.ascx")
    Else
        Datalist1.ItemTemplate = Page.LoadTemplate( _
            "DataList_TemplateSimple.ascx")
    End If
    BindControls()
End Sub
```

This is all you need for a basic use of external template files, and the figure below shows how the two datalists look like in admin and normal user mode, respectively:

Employees
Ms. Davolio , Nancy Sales Representative 507 - 20th Ave. E. Apt. 2A - Seattle, WA
Dr. Fuller , Andrew Vice President, Sales 908 W. Capital Way - Tacoma, WA
Ms. Leverling , Janet Sales Representative 722 Moss Bay Blvd. - Kirkland, WA
Mrs. Peacock , Margaret Sales Representative 4110 Old Redmond Rd. - Redmond, WA
Mr. Buchanan , Steven Sales Manager 14 Garrett Hill - London,
Mr. Suyama , Michael Sales Representative Coventry House Miner Rd. - London,
Mr. King , Robert Sales Representative Edgeham Hollow Winchester Way - London,
Ms. Callahan , Laura Inside Sales Coordinator 4726 - 11th Ave. N.E. - Seattle, WA
Ms. Dodsworth , Anne Sales Representative 7 Houndstooth Rd. - London,

Employees
Ms. Davolio , Nancy
Dr. Fuller , Andrew
Ms. Leverling , Janet
Mrs. Peacock , Margaret
Mr. Buchanan , Steven
Mr. Suyama , Michael
Mr. King , Robert
Ms. Callahan , Laura
Ms. Dodsworth , Anne

Suppose that you want to show different data for five different groups of users, and you'll immediately understand the power and ease of use of this solution, in comparison with hardcoding within the same template all the controls and binding expressions for showing all the data, and then showing/hiding them in various combinations to produce the wanted output for the current user.

However, there is still a thing you may argue about: with this method the template is read from file every time the page loads. Even if this I/O operation shouldn't impact on performances, we can improve the way the template is loaded, by loading it from file just the first time the template is required, and storing it into the cache for future uses. Here's a function that does what described:

```
Private Function GetTemplate(ByVal templateFile As String) As ITemplate
    ' if the file has been already loaded and saved into the cache,
    ' load it from there
    If Not Cache(templateFile) Is Nothing Then
        Trace.Write("Template " & templateFile & " loaded from cache")
        Return CType(Cache(templateFile), ITemplate)
    End If

```

```
Else
    ' otherwise load the template file...
    Dim template As ITemplate = Page.LoadTemplate(templateFile)
    ' ...and save it into the cache, with a dependency to the original
    ' file, so that if the file is modified the cache item is removed
    Cache.Insert(templateFile, template, _
        New Caching.CacheDependency(Server.MapPath(templateFile)))
    Trace.Write("Template " & templateFile & " loaded from file")
    Return template
End If
End Function
```

When inserting the loaded template into the Cache collection we add a dependency to the source template file, so that if it is updated, the cached template is discarded, and the fresh version is loaded at the next page load. The Page.LoadMethod call used above can now be replaced as follows:

```
Datalist1.ItemTemplate = GetTemplate("DataList_TemplateEx.ascx")
```

To check whether the template is actually loaded from the cache from the second time the page is loaded, do the following:

- 1) Enable the tracing by setting to true the enabled attribute of the trace tag in the Web.Config file.
- 2) Refresh the page a couple of times
- 3) Point the browser to trace.axd, and you'll get a list of links to see the tracing details of each page execution. In these details pages you'll actually see the trace messages that confirm the fact that the template has been cached and reused since the second page execution.

The figure below shows two partial screenshot from the two details pages in question:

Request Details

Request Details

Session Id:	azztr1552tyvn255ed0wsraq	Request Type:	GET
Time of Request:	3/10/2003 11:57:29 AM	Status Code:	200
Request Encoding:	Unicode (UTF-8)	Response Encoding:	Unicode (UTF-8)

Trace Information

Category	Message	From First(s)	From Last(s)
aspx.page	Begin Init		
aspx.page	End Init	0.000089	0.000089
	Template Repeater_TemplateSimple.ascx loaded from file	0.539975	0.539886
	Template DataList_TemplateSimple.ascx loaded from file	1.087741	0.547766
	Template DataGrid_TemplateSimple.ascx loaded from file	1.622305	0.534564
aspx.page	Begin PreRender	1.685107	0.062802
aspx.page	End PreRender	1.685267	0.000160
aspx.page	Begin SaveViewState	1.703134	0.017867
aspx.page	End SaveViewState	1.703914	0.000780
aspx.page	Begin Render	1.704005	0.000091
aspx.page	End Render	1.727779	0.023774

Request Details

Request Details

Session Id:	q3n00qycbezbt045xxf11q55	Request Type:	GET
Time of Request:	3/11/2003 1:23:11 AM	Status Code:	200
Request Encoding:	Unicode (UTF-8)	Response Encoding:	Unicode (UTF-8)

Trace Information

Category	Message	From First(s)	From Last(s)
aspx.page	Begin Init		
aspx.page	End Init	0.000104	0.000104
	Template Repeater_TemplateSimple.ascx loaded from cache	0.006595	0.006491
	Template DataList_TemplateSimple.ascx loaded from cache	0.006694	0.000099
	Template DataGrid_TemplateSimple.ascx loaded from cache	0.006738	0.000044
aspx.page	Begin PreRender	0.018138	0.011400
aspx.page	End PreRender	0.018298	0.000160
aspx.page	Begin SaveViewState	0.093878	0.075580
aspx.page	End SaveViewState	0.094572	0.000694
aspx.page	Begin Render	0.094663	0.000091
aspx.page	End Render	0.117333	0.022670

So far I've shown examples with the DataList only. Working with the Repeater and the DataGrid is very similar though. To dynamically load templates in a DataGrid, you must use TemplateColumn columns, left empty as shown in the following code:

```
<asp:DataGrid runat="server" ID="Datagrid1"
  AutoGenerateColumns="False" Width="100%"
  ItemStyle-BackColor="LightCyan"
  ItemStyle-ForeColor="DarkBlue"
  AlternatingItemStyle-BackColor="LightYellow"
  AlternatingItemStyle-ForeColor="Maroon"
  ...other style formatting...>
  <Columns>
    <asp:BoundColumn DataField="EmployeeID" HeaderText="ID"
      ItemStyle-Width="20" ItemStyle-BackColor="LightGreen" />
    <asp:TemplateColumn HeaderText="Employee Info" />
  </Columns>
</asp:DataGrid>
```

All you would need to change in the template file is the cast: in this case you must cast the Container object to the DataGridItem type not to DataListItem because that is the type of the DataGrid's items. The same would be for a Repeater, you would have to cast to a RepeaterItem. In the sample code provided as a separate download for this article, you can find full examples for all

these controls. The code below instead is the complete implementation for the Page_Load event, that show how it's actually the same to load a template for a Repeater and a DataList, and also for a DataGrid, once we have a reference to its template column that we want to use as container for the template:

```
Private Sub Page_Load() Handles MyBase.Load
    ' show the welcome message with the current username
    UserName.Text = User.Identity.Name.Substring( _
        User.Identity.Name.IndexOf("\") + 1)
    ' get a reference to the DataGrid's template column
    Dim templCol As TemplateColumn = CType( _
        Datagrid1.Columns(1), TemplateColumn)
    ' load the proper templates according to the logged-in user
    If User.IsInRole("BUILTIN\Administrators") Then
        Repeater1.ItemTemplate = GetTemplate("Repeater_TemplateEx.ascx")
        Datalist1.ItemTemplate = GetTemplate("DataList_TemplateEx.ascx")
        templCol.ItemTemplate = GetTemplate("DataGrid_TemplateEx.ascx")
    Else
        Repeater1.ItemTemplate = GetTemplate("Repeater_TemplateSimple.ascx")
        Datalist1.ItemTemplate = GetTemplate("DataList_TemplateSimple.ascx")
        templCol.ItemTemplate = GetTemplate("DataGrid_TemplateSimple.ascx")
    End If
    BindControls()
End Sub
```

The BindControls method takes care of retrieving the data to show from the database, and bind it to the template controls. It uses a DataSet instead of a DataReader, because the data must be bound to three controls, so it makes perfect sense to retrieve the data only once and then use it for multiple controls:

```
Private Sub BindControls()
    ' fill a DataSet's DataTable with all the data from
    ' the Northwind's Employees table
    Dim cn As New SqlConnection( _
        ConfigurationSettings.AppSettings("NWConnString"))
    Dim cmd As New SqlCommand("SELECT * FROM Employees", cn)
    Dim ds As New DataSet()
    Dim da As New SqlDataAdapter(cmd)
    da.Fill(ds, "Employees")
    ' bind that table to all the template-based controls
    Repeater1.DataSource = ds.Tables("Employees")
    Repeater1.DataBind()
    Datalist1.DataSource = ds.Tables("Employees")
    Datalist1.DataBind()
    Datagrid1.DataSource = ds.Tables("Employees")
    Datagrid1.DataBind()
End Sub
```

Custom Template Classes

The template technique seen so far is great if you want to dynamically apply the same template to all the rows of a Repeater, DataList or DataGrid, but there yet one more situation you may face. Say for example that you want to use a red foreground color for employees that have a salary greater than a certain amount, and instead use green for employees that make less than another amount. Or that you want to render with different background colors the male, female, and gender-neutral employees, according to their title of courtesy. Or again, you may want to hide some information only for employees that explicitly required this (the e-mail or mailing address, for example), but not for everybody.

Some of these row-by-row customizations, such as showing/hiding certain data, can be done by binding the Visible property of a Label to an expression. Others, regarding the visual customization of the item's layout and style or the data being shown, can be done by handling the control's ItemCreated and ItemDataBound events. However, if you need the same customized template for more than one control, possibly in different pages, you'll end up repeating the same template definition and the same code-behind for its customization over and over again, in many places. This is not the best for maintainability and code-reuse of course.

In these cases, you can create a class that implements the ITemplate interface, and that programmatically creates all the controls for the template column or item, and takes care of binding the required data to the proper controls. In practice, it does by code everything you would declare in the ASPX file and in the code-behind. Then, you can create an instance of this class and associate it to the Repeater's, DataList's or DataGrid's templates, as we did before with the templates loaded from file. This is a great option, because in a single class you can have everything is required for your row-by-row custom output, and you achieve a great code reuse because you'll be able to use the class from everywhere in your project, or in any other project if you compile the class in its own redistributable Class Library assembly. As an example, we're going to see how to render the employees with different background colors, according to their title of courtesy.

We declare a class that implements ITemplate, add any custom property (I use public fields in the example below, as I don't have to validate the value, but the concept it's the same) and a constructor that takes in input the values to initialize these properties:

```
Class MyCustomTemplate
    Implements ITemplate

    Public ForeColor As Color
    Public FemaleColor As Color
    Public MaleColor As Color
    Public NeutralColor As Color
    Dim WithEvents pan As Panel

    ' the constructor takes the color for female/male/neutral empl. titles
    Sub New(ByVal femaleColor As Color, ByVal maleColor As Color, _
        ByVal neutralColor As Color, ByVal foreColor As Color)
        Me.FemaleColor = femaleColor
        Me.MaleColor = maleColor
        Me.NeutralColor = neutralColor
        Me.ForeColor = foreColor
    End Sub
```

The ITemplate interface has a single method to implement, InstantiateIn, called when the container's item is being created, and where we must instantiate the controls that will show the data. In this example we need three labels, for the employee's first and last names, and the title of courtesy. However, we first create a panels that covers the whole container's item's size, and then add the labels inside it. We use a container panel control so that's easier to set the item's background color. So, the labels are added to the panel's Controls collection, and the panel itself is added to the container control's Controls collection. Here's the code for this method:

```
Public Sub InstantiateIn(ByVal container As System.Web.UI.Control) _
    Implements System.Web.UI.ITemplate.InstantiateIn

    ' create a panel with the specified forecolor
    pan = New Panel()
    pan.ForeColor = Me.ForeColor

    ' if the parent control is a WebControl...
    If TypeOf (container) Is WebControl Then
        ' get a strongly-typed reference to the containing item
        Dim webCtl As WebControl = DirectCast(container, WebControl)
        ' set the panel's size so that it fully cover's its containers
        pan.Width = webCtl.Width
        pan.Height = webCtl.Height
    End If

    ' add 3 label controls, and two literals for the <b></b> tags
    pan.Controls.Add(New Label()) ' this is the TitleOfCourtesy label
    pan.Controls.Add(New LiteralControl("<b>"))
    pan.Controls.Add(New Label()) ' this is the LastName label
    pan.Controls.Add(New LiteralControl("</b>, "))
    pan.Controls.Add(New Label()) ' this is the FirstName label

    ' add the Panel control to the DataListItem control
    container.Controls.Add(pan)
End Sub
```

At this point we've only declared the controls for the template, but haven't yet specified what they should show. To bind the data to these controls, we must react to their DataBinding events, raised when a row in the Repeater / DataList / DataGrid, and so the inner controls in its templates, are being bound to a data item. If you look again to the above, you'll see that the Panel control was declared at class level, with the WithEvents keyword, so that's easy to handle its events.

The next method we have to write is actually an handler for the panel's DataBinding event, where we get a reference to the container's data item, extract the data we want (the first and last names, and the title of courtesy), and show it in the three labels created in the InstantiateIn method. Let's first see the complete code:

```
' this event fires once for each item in the DataList,
' as it is being bound to the data source
Private Sub pan_DataBinding(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles pan.DataBinding

    Dim title, firstName, lastName As String

    ' get a reference to the container item, and then to the data item
    Dim data As Object
    If TypeOf (pan.NamingContainer) Is DataListItem Then
        data = DirectCast(pan.NamingContainer, DataListItem).DataItem
```

```

ElseIf TypeOf (pan.NamingContainer) Is DataGridItem Then
    data = DirectCast(pan.NamingContainer, DataGridItem).DataItem
ElseIf TypeOf (pan.NamingContainer) Is RepeaterItem Then
    data = DirectCast(pan.NamingContainer, RepeaterItem).DataItem
Else
    Return
End If

' cast the data item to the proper type:
' - DataRowView if the datasource is a DataTable/DataView)
' - DbDataRecord if the datasource is a DataReader
' and the retrieve the values we need to show in the template
If TypeOf (data) Is DataRowView Then
    Dim drv As DataRowView = DirectCast(data, DataRowView)
    title = drv("TitleOfCourtesy").ToString()
    firstName = drv("FirstName").ToString()
    lastName = drv("LastName").ToString()
Else
    Dim dbr As DbDataRecord = DirectCast(data, DbDataRecord)
    title = dbr("TitleOfCourtesy").ToString()
    firstName = dbr("FirstName").ToString()
    lastName = dbr("LastName").ToString()
End If

' set the pan's BackColor according to the TitleOfCourtesy value
Select Case title
    Case Is = "Ms.", "Mrs.", "Lady"
        pan.BackColor = FemaleColor
    Case Is = "Mr."
        pan.BackColor = MaleColor
    Case Else
        pan.BackColor = NeutralColor
End Select

' display the values in the Panel child controls.
Dim lblTitle As Label = DirectCast(pan.Controls(0), Label)
lblTitle.Text = title
Dim lblLastName As Label = DirectCast(pan.Controls(2), Label)
lblLastName.Text = lastName
Dim lblFirstName As Label = DirectCast(pan.Controls(4), Label)
lblFirstName.Text = firstName
End Sub
End Class

```

First of all, as we earlier did in the template files, we must cast the generic panel's container to the right type (DataListItem, for example), so that we can access its properties, such as DataItem. However, while the template files were tailor-made for a specific template control, we want to make this class working with either the Repeater, the DataList or DataGrid. With the template files this wasn't important because in case we wanted to switch from a DataList to a DataGrid we had just to make a little modification to a couple of text files.

In this case however things would be much more difficult, because we'd also have to recompile the code and this could be even impossible if we don't have the source code but are referencing the class from a compiled assembly developed by someone else. Therefore, you'd likely want to write a single class that can work with any of these three controls. For this reason, we check the type of the container with the TypeOf function, cast the container to the proper type and get an Object reference from the DataItem property, that represents the data item being bound to this item.

Before being able to read the values we need from the retrieved data item, we must follow a similar process though, because the data item can be of type DbDataRecord or DataRowView, according to whether the container Repeater / DataList or DataGrid's DataSource is a DataReader or a DataTable/DataView, respectively. In the code above you see that we use again the TypeOf function to know the right type for the cast, do the cast and can finally read the wanted values and save them to local variables.

We're near the end. At this point, we use the retrieved TitleOfCourtesy value to decide which background color we have to use to render the employee, and use it for the panel's BackColor property. Finally, we get the references to the inner Label controls we added to the panel, by casting from the general Control returned by the panel's Controls collection to Label, and then assigning their Text properties with the values read from the data item.

We're done with the custom template class, and we can use it. We first create a new instance of the class, passing to its constructor method the colors to be used for the different titles of courtesy, and then assign this instance to the ItemTemplate property of a Repeater or DataList, as shown below:

```

' create a new instance of the custom template class,
' and load it in the 2 template-based controls
Dim mct As New MyCustomTemplate(Color.Pink, Color.Cyan, _

```

```

    Color.LightGreen, Color.Black)
Repeater2.ItemTemplate = mct
Datalist2.ItemTemplate = mct

```

For the DataGrid we first retrieve a reference to a TemplateColumn, and then set its ItemTemplate property:

```

' get a reference to the DataGrid's template column, and set its
' template to the MyCustomTemplate instance
Dim templCol2 As TemplateColumn = CType( _
    Datagrid2.Columns(1), TemplateColumn)
templCol2.ItemTemplate = mct

```

The following screenshot shows how the developed custom template class actually works with all these different controls:

Employees

Ms. Davolio, Nancy
Dr. Fuller, Andrew
Ms. Leverling, Janet
Mrs. Peacock, Margaret
Mr. Buchanan, Steven
Mr. Suyama, Michael
Mr. King, Robert
Ms. Callahan, Laura
Ms. Dodsworth, Anne

Employees

Ms. Davolio, Nancy
Dr. Fuller, Andrew
Ms. Leverling, Janet
Mrs. Peacock, Margaret
Mr. Buchanan, Steven
Mr. Suyama, Michael
Mr. King, Robert
Ms. Callahan, Laura
Ms. Dodsworth, Anne

ID Employee Info

1	Ms. Davolio, Nancy
2	Dr. Fuller, Andrew
3	Ms. Leverling, Janet
4	Mrs. Peacock, Margaret
5	Mr. Buchanan, Steven
6	Mr. Suyama, Michael
7	Mr. King, Robert
8	Ms. Callahan, Laura
9	Ms. Dodsworth, Anne

Try to change the BindControls method written before so that it uses DataReaders instead of a DataSet as a data source, and you'll see that the page will continue to display correctly.

In this article I mentioned quite a few situations where you may need custom templates, but you may surely think about others. Templates defined in separate files are great when you need to use different data representation basing on the current user or on other options, while custom template classes are best used when you also have to change the output on a row-by-row basis, and when you want to achieve the maximum code encapsulation and reusability.

Note: You can download the sample code from the link in the left column.

Marco Bellinaso is VB2TheMax's technical editor, and works as a software developer and trainer for Code Architects Srl, an Italian company that specializes in .NET. He's been working with the .NET Framework since the Beta 1, and is now particularly interested in e-commerce design and implementation solutions with SQL Server, ASP.NET, and web services, with both C# and VB.NET. Marco is a co-author for a number of Wrox Press books, among which are [ASP.NET Website Programming](#), [Fast Track ASP.NET](#) and [Visual C# .NET: a Guide for VB6 Developers](#). He is also a technical reviewer for other books, a contributing editor for two leading Italian programming magazines: *Computer Programming* and *Visual Basic & .NET Journal* (Italian licensee for Visual Studio Magazine), and co-author of [VBMaximizer](#) add-in for VB6.

DevX is a division of Jupitermedia Corporation
© Copyright 2007 Jupitermedia Corporation. All Rights Reserved. [Legal Notices](#)