

# Visual Basic Live Techniques

Mike Wills  
Microsoft Corporation

Revised December 1998

**Summary:** Discusses common Web site development techniques discovered during the development of the Visual Basic® Live Web site, created to aid developers in building their own Internet applications using the Microsoft® Visual Basic development system. (11 printed pages) Techniques discussed include:

- Avoiding frames.VBLive\_frames
- Outputting Web pages using multiple templates.VBLive\_output
- Including the cascading style sheet in each page.VBLive\_css
- Using custom webitems instead of HTML template webitems.VBLive\_webitems
- Creating a list of records using Join.VBLive\_join
- Using Redirect vs. set NextItem.VBLive\_redirect
- Encapsulating [data access code in standard modules.](#)

## Introduction

This document explains techniques discovered and used while developing the Visual Basic Live Web site (<http://msdn.microsoft.com/vbasic/downloads/default.asp>). Visual Basic Live is a Web site created using the Visual Basic WebClass Designer and is intended to provide developers with a jump-start for building their own Internet applications using the Visual Basic development system. Each technique provides a solution to common Web site development needs. Therefore, understanding these issues will prepare developers using Visual Basic to create Web sites with good performance and easy maintainability.

To help explain design considerations, this document reports the results of many performance tests. Before reviewing these reports, the reader should understand that statistics from these performance tests should not be compared to any other test results. Some tests were performed with a prerelease version of Visual Basic 6.0 for the DEC Alpha. The prerelease version might not contain optimizations that are in the final release of Visual Basic 6.0. Also, because the server was used for a development environment, debug components may have been on the machine. To compare the performance of the WebClass Designer with any other technology, like Active Server Pages (ASP), new tests should be designed. The tests performed while developing the Visual Basic Live Web site are only intended to compare the performance impact of different architectural choices.

Now that the appropriate uses of the performance statistics have been explained, let's discuss the techniques used to create Visual Basic Live.

## Avoiding Frames

Frames can be an easy way to separate Web site navigation from Web site content. However, certain drawbacks of frames make using them undesirable:

- Frames increase the impact of latency—that is, the frame content of a Web page is not requested until the browser receives the parent page.

- Using frames causes any shortcuts or bookmarks to specific pages on the site to point to the site's home page instead. Pages other than the home page cannot be bookmarked successfully.
- When viewing frames some older browsers, such as Netscape Navigator 2.0, will disable the Back button.
- A framed Web site may cause problems with search engines.

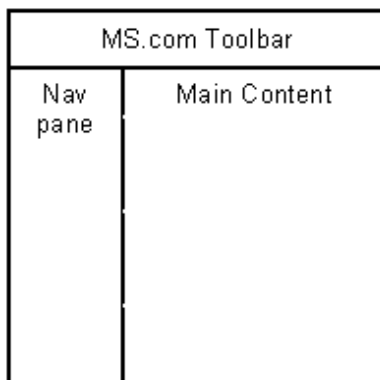
If you can avoid using frames, the potential number of concurrent connections will be reduced and, therefore, TCP timeouts and retries will be less likely, particularly for users of slow modems.

**Note** For more information on frames, refer to the article "Getting Framed" by Robert Hess (<http://msdn.microsoft.com/workshop/author/hess/hess1223.asp>).

Because using frames is undesirable, a technique that offers a similar benefit without the drawbacks was discovered—a technique of using multiple HTML templates to create a single Web page.

## Outputting Web Pages Using Multiple Templates

Even though frames were not used to create Visual Basic Live, an important benefit of using frames was still desired. Frames allow Web page content to be divided into logical reusable units. For example, most of the Web pages in Visual Basic Live have three main sections: the Microsoft.com toolbar, the left navigation pane, and the main content (see Figure 1).



**Figure 1.**

The Microsoft.com toolbar and navigation panes are identical on most of the pages. Therefore, having a copy of the HTML code necessary to create these sections in every page creates redundancy. A change to the navigation pane would cause every page to require the same editing. Not only would this cause a lot of tedious maintenance work, it would cause the work to be error-prone. Frames are commonly used to solve this redundancy problem, but by using the WebClass Designer the same problem was solved by separating Web pages into multiple WebClass templates, as shown in the following code example:

```
<HTML>
<HEAD>
  <TITLE>Visual Basic live</TITLE>
  <!--Styles-->
  <@WRITETEMPLATE>
</HEAD>
<BODY>
  <!--Pop-up Menu-->
  <@WRITETEMPLATE>
  <TABLE>
```

```

<TR>
  <!--MS Toolbar-->
  <TD height=20>
    <@WRITETEMPLATE></TD>
</TR>
<TR>
  <TD>
    <TABLE>
      <!-- Body Row Start-->
      <TR>
        <TD vAlign=top>
          <TABLE>
            <TR>
              <!-- Left Pane Start -->
              <TD>
                <@WRITETEMPLATE></TD>
              <!-- Body Pane Start -->
              <TD VALIGN=top>
                <@WRITETEMPLATE></TD>
              <!-- Body Pane End -->
            </TR>
          </TABLE></TD>
        </TR>
      <!-- Body Row End-->
    </TABLE></TD>
  </TR>
</TABLE>

```

In the Visual Basic Live site, HTML templates are combined with each other to create complete Web pages. Some of the templates are standard sections that are used in most of the Web pages. These templates include LeftPane.htm, MStoolbar.htm, PopupMenus.htm, and Styles.htm. Combining these templates to create one HTML response is accomplished using a technique that involves the use of an HTML resource to define the structure of the Web site's pages and the use of the Visual Basic for Applications (VBA) new string functions **Split** and **Join**. The HTML resource (the structure resource) is simply an HTML file compiled into the Visual Basic Live project's RES file. The HTML file contains all the tags necessary to define how the templates are combined to make one page. The structure resource contains very little content. It mainly contains HTML tables with the token <@WRITETEMPLATE> inserted wherever a template needs to be inserted. Refer to the preceding code example for a simplified demonstration of what the structure resource looks like. To write each of the Web pages, a procedure exists in the Visual Basic Live project that accepts parameters, identifying the webitem and the template for creating the page's main content. Outputting a Web page, the procedure **WriteHTMLUsingStructure** loads the structure resource and separates it into six pieces using the **Split** function. The **Split** function divides the structure into pieces separated by the token <@WRITETEMPLATE> and returns a string array containing them. Continuing, **WriteHTMLUsingStructure** writes each structure piece to the ASP page buffer and between each structure piece the procedure writes an appropriate webitem template. When each webitem template is written using its **WriteTemplate** method, its replacement tokens are processed. Examine the code example to follow for a simplified version of the **WriteHTMLUsingStructure** procedure. To see a more complex version, download the Visual Basic Live source code from <http://msdn.microsoft.com/vbasic/downloads/default.asp>.

```

Private Sub WriteHTMLUsingStructure(oBodyItem As WebItem, _
                                   sBodyTemplate As String)

  Dim sStructPieces() As String
  'Get the page structure from the resource and
  'split it into sections.
  'Between each section, the contents of a WebItem
  'will be written.
  sStructPieces = Split(GetResData(RES_STRUCTURE, RES_HTML_TYPE), _
    WRITETEMPLATE_DELIMITER)
  With Response
    .Write sStructPieces(0)
    'Write styles template
    Styles.WriteTemplate TPLT_STYLES
    .Write sStructPieces(1)
  End With

```

```

'Write Popup Menus template
'Use the LeftPane item to process the popup menu template
LeftPane.WriteTemplate TPLT_POPUPMENUS
.Write sStructPieces(2)
'Write MS Toolbar template
MSToolbar.WriteTemplate TPLT_MSTOOLBAR
.Write sStructPieces(3)
'Write Left Pane template
LeftPane.WriteTemplate TPLT_LEFTPANE
.Write sStructPieces(4)
'Write Body Pane template
oBodyItem.WriteTemplate sBodyTemplate
.Write sStructPieces(5)
End With
End Sub

```

Using the technique demonstrated by **WriteHTMLUsingStructure** provides a clean way to separate HTML content in easy-to-maintain units without using frames. However, maintainable code is not the only issue—on a Web server performance is critical. To test the performance of this design choice, two performance comparisons were conducted to measure the cost of using multiple webitem's per Web page. The first test compared the performance of the server returning two different Web pages. Both pages were the same size and used four replacement tags on the server side, but the first page consisted of one webitem and the second page consisted of five webitems. With 35 client threads across five machines polling the server for five minutes, the server was able to satisfy 21,185 requests for the first page at 70.62 requests per second. When using the second page, the server was able to satisfy 17,876 requests at 59.59 requests per second.

The comparison just made tests the impact of the multitemplate technique on a simple Web page. To test the technique's impact in a more realistic case, the test was repeated except for a change in both Web pages. The new versions queried a database to return an HTML table embedded on each page. With this change, the server was able to satisfy 1,134 requests for the first page at 3.78 requests per second. When using the second modified page, the server was able to satisfy 3,293 requests at 10.98 requests per second. The second page's superior performance was surprising, but it can be explained. The first page used a replacement tag and the ProcessTag event to insert the HTML table into the Web page. The second page did not use a replacement token. Because the page was already segmented into multiple pieces that were put together by sequential calls to the **Write** or **WriteTemplate** methods, the HTML table was inserted by writing it to the **Response** object in the correct sequence between two other page segments. It may be the cost of using replacement tokens is high, when the replacement string is large. However, the point is that the multiple template per page technique, which relies on VBA's new **Split** and **Join** functions, provides excellent performance.

## Including the Cascading Style Sheet in Each Page

Using a linked or included style sheet is often recommended for its maintenance benefits. It allows the look of several pages to be changed quickly and for creating a consistent look across several pages to be easily managed. This is because all styles for several pages are defined in one file. However, using a linked style sheet instead of an inline style sheet has performance problems similar to using frames. According to Gerald Cermak and Kenneth Lassesen in an article they co-authored, using the "LINK [a linked style sheet] is not performance beneficial unless the LINK file is large, probably 20KB or more. Consider the following example: A page gets download with a LINK file, we have 5 seconds for IE turnaround, 4 seconds for Sicily authentication and then 21.4 seconds to get the CSS file. Net result, the page renders 30.4 seconds later than if: no cascading style sheet was used, or the style sheet was placed in line." Because of the performance benefits, the Visual Basic Live Web site uses inline style sheets. However, by using the technique of creating one Web page with multiple webitem templates, all the benefits of linked style sheets are realized. In the section earlier titled "[Outputting Web Pages Using Multiple Templates](#)," the fact that Styles.htm is a template included in all Web pages is mentioned. Styles.htm contains the cascading style sheet applied to all of the Visual Basic Live Web pages. Therefore, the maintenance benefits of linked style sheets are obtained, but because the Styles.htm template is combined with the other webitem templates on the server side, the

performance benefits of inline style sheets are provided as well.

## Using Custom Webitems Instead of HTML Template Webitems

As a result of choosing an architecture that uses multiple webitems per Web page, using custom webitems became preferable over using HTML template webitems. HTML template webitems are webitems that have a design-time association with an HTML template. Because of this design-time association, an HTML template is opened in the WebClass Designer, parsed, and edited to link HTML elements with webitem events. The component the WebClass Designer uses to parse and edit HTML automatically adds <BODY> and <HTML> tags, if they are not already in the HTML template. This poses a problem with using the multiple webitem per page technique. Because each HTML template is combined with other HTML templates to make a complete Web page, it cannot have its own <HTML> or <BODY> tags. The resolution is to use custom webitems instead of HTML template webitems. Custom webitems do not have design-time associations to HTML templates. Therefore, using them did not require loading a template in the WebClass Designer. Instead, using custom webitems required linking HTML elements with webitem events at run time. This is accomplished by inserting a replacement token into a template wherever it needs to be linked to a webitem event. At run time, the token is replaced with the URL generated with the **URLFor** function. Using custom webitems instead of HTML template webitems requires very little extra work but provides the control necessary to use the multiple webitem per Web page architecture.

## Creating a List of Records Using Join

A common requirement of a Web site project is to have data-driven lists. The Visual Basic Live Web site has two data-driven lists: the **Highlights** list on the home page and the **Search Result** list. The code that generates these lists uses a string concatenation technique that provides excellent performance. However, before choosing this technique, tests were performed to measure the performance difference between it and two other techniques.

There were three possibilities considered for writing many records to the response object: 1) Append each formed HTML table row to a string one record at a time. 2) Write each row to the response object as it is formed without appending it to a string. 3) Create an array dimensioned to hold one string for each record and assign each record to an element of the array as it is formed; when the array is complete, use the Visual Basic for Applications **Join** method to create one string out of the array. It was expected that the first option would perform the worst. However, it still provided an interesting comparison.

The first test compared the options by logging how long the list-generating procedure took. On the same machine it took an average of 1056 milliseconds to create a response using option one, but only took an average of 414 milliseconds using option three. The comparisons between option two and option three were close. When testing the two options under light load conditions, option two took an average of 1508 milliseconds and option three took an average of 1466 milliseconds. Because the results were so close, the Microsoft Web Capacity Analysis Tool (WebCat) was used to measure performance on the server when under heavy load. Using option two the server was able to satisfy 3.24 HTTP requests per second, for a total of 971 requests. Using option three the server was able to satisfy 3.88 requests per second for a total of 1,164 requests. Option three became the preferable choice and, fortunately, it offered the cleanest code and best encapsulation.

```
Private Function GetSampleHighlightRecords() As String
    Dim sStructPieces() As String
    Dim sRecords() As String
    Dim lRecCount As Long
    Dim lIndex As Long
    Dim oRecordset As ADODB.Recordset

    'Get the structure for displaying the Highlight
    'records and split it into sections.
    'Between each section, the value of a field
```

```

'will be inserted
sStructPieces = Split(GetResData(RES_HIGHLIGHTRECORD, RES_HTML_TYPE), _
                    WRITEFIELD_DELIMITER)

Set oRecordset = GetHighlightSamples
With oRecordset
    lRecCount = .RecordCount
    'Create result message
    If lRecCount > 0 Then
        'Allocate an array that has an element for every
        'highlight record.
        ReDim sRecords(1 To lRecCount)
        For lIndex = 1 To lRecCount
            sRecords(lIndex) = sStructPieces(0) & _
                GetURLForSample(.Fields.Item(FLD_SAMPLES_SAMPLEID).Value) & _
                sStructPieces(1) & .Fields.Item(FLD_SAMPLES_TITLE).Value & _
                sStructPieces(2) & .Fields.Item(FLD_SAMPLES_DESCRIPTION).Value & _
                sStructPieces(3)
            .MoveNext
        Next lIndex
        'Join all the elements of the array to create one
        'large block of HTML text.
        GetSampleHighlightRecords = Join(sRecords, vbCrLf)
    End If
End With
End Function

```

To understand how to use option three, the option that uses **Join** to concatenate strings, look at the preceding code sample. This example provides a simplified version of the procedure **GetSampleHighlightRecords**. The purpose of this procedure is to return the contents for an HTML table—contents based on a query's result set. To create the table contents, a loop is used to navigate through each record of the result set. In each loop, one row of the HTML table is created. After creating all of the table rows, one string that contains all of the table contents needs to be returned. A typical approach to this procedure would be to append each new row of the HTML table to a string variable, during every increment of the loop. Such a string concatenation would cause Visual Basic to reallocate memory for the string every time a new row is generated. This is because when a string increases in length a whole new block of memory is allocated to store it. Because reallocating memory is expensive, the typical technique of appending each HTML row to the same string variable is very costly on performance. The new Visual Basic for Applications **Join** function offers a better performing alternative, demonstrated in **GetSampleHighlightRecords**. Before entering the HTML row-creating loop, a string array is created with enough elements to store each new HTML row. When a new row is created in each increment of the loop, the row is stored in its own element of the array. This causes only minimal memory allocation to be needed in each loop—only enough to store the new HTML row. After all the HTML rows are created, **Join** is used to convert the string array into one string containing all of the HTML rows. This causes memory allocation needed to store all the rows in one string variable to occur only once, instead of each time a row is created. Using the new **Join** function to concatenate a large number of strings is easy and provides excellent performance.

## Using Redirect vs. Set NextItem

When creating a Web application, it is common for server-side code to select a Web page for output based on application state. Redirecting a client's request for a member-only Web page to a login page is a simple example of this need. One way to cause a Web page selection is to use the Active Server Pages **Response.Redirect** method. Using the **Redirect** method adds an extra network round trip to processing the client request. This occurs because calling **Redirect** causes the server to send a message back to the client, a message that directs the client to request a new URL. The Visual Basic WebClass Designer offers a new way to cause server-side Web page selection, the **NextItem** property. Setting this property equal to a webitem causes that webitem's Respond event to fire. Using Visual Basic Live as an example, there is a Submit event on the Login webitem that processes the click of the **Log in** button. If the Login fails, the Login page must be shown again. To accomplish this, code in the Submit event will set the **NextItem** property equal to the Login webitem. After the Submit event is finished, the WebClass engine checks the **NextItem** property and discovers that it is set equal to the Login webitem. It then

fires the webitem's Respond event, which causes the Login page to be written back to the client. The **NextItem** is an efficient and clean way to select Web page content to be written back to the client. However, there are scenarios where using the less efficient **Redirect** method causes a better experience for the user of the Web application.

The **Redirect** method may be preferable to using the **NextItem** property in an event procedure that handles the submission of HTML form data. Using the **Redirect** method causes the Submit event of the form to be removed from the browser's navigation history and replaced with the URL that the request was redirected to. Take the Visual Basic Live Login page, for example. When the user clicks **Log in** on that page, HTML form data is submitted and handled by the Submit event of the Login webitem. After the Submit event validates the form data, the home page must be displayed to the user. If the Submit event used the **NextItem** property to cause the home page to be displayed, here is the URL that would be in the browser's address box and navigation history for the home page:

```
http://vblive.rte.microsoft.com/Default.asp?WCI=LogIn&WCE=Submit
```

Notice that the preceding URL contains the query string necessary to cause the Submit event to be fired. Therefore, if the user clicked **Refresh** while viewing the home page, the Submit event would fire again and the user would be prompted with a message box asking if the Form data associated to the URL should be resubmitted. Also, if the user navigated away from the URL and navigated back to it after the URL had expired, the user would again be given the form data warning and the Submit event would be fired. Another problem with having the Submit URL associated to the home page occurs when the user bookmarks the home page. The user could activate that bookmark hours or days later, after state data is lost. The user would be expecting to see the home page again, but because the Submit event is fired, he would likely receive a log-in validation message instead.

After considering the difficulties caused by using the **NextItem** property in a webitem event that handles HTML form data, using **Redirect** was used instead. So instead of using "Set NextItem = Home" in the Login webitem's Submit event, "Response.Redirect URLFor(Home)" is used. This causes the following URL to be displayed in the browser's address box and to be saved in the browser's history when the home page is displayed by the Submit event:

```
http://vblive.rte.microsoft.com/Default.asp?WCI=Home
```

Notice that the new URL contains the query string that causes the Home webitem's Respond event to be fired. Using the **Redirect** method solves all of the problems associated to using the **NextItem** property in an event that handles form data. Because the URL for the Submit event is literally replaced with the URL for the home page, the user will experience intuitive results when clicking **Refresh**, creating a shortcut, or navigating back to the home page using the browser's history.

## Encapsulating Data Access Code in Standard Modules

Encapsulating data access code is a common goal when creating a multitier application. Usually this is accomplished by creating a data access layer of code encapsulated in a separate component library or in a set of private class modules. When designing the Visual Basic Live Web site, performance tests were conducted that demonstrated encapsulating the data access code in class modules significantly impacted scalability. In the following test explanations, option one refers to WebClass code that uses Microsoft ActiveX® Data Objects (ADO) without such encapsulation, and option two refers to the same code that uses Visual Basic-built classes to encapsulate code using ADO. When testing the options under a light load, the performance difference was difficult to detect. On an Alpha 500 with 128 megabytes (MB) of RAM, option one required an average of 287 milliseconds and option two took an average of 296 milliseconds to generate an HTML table based on the results of a query.

When testing option one and two under heavy server load, the differences were dramatic. Using option one, the server was able to satisfy 9.80 requests per second for a total of 2,939 requests.

Using option two, the server only satisfied 2.99 requests per second for a total of 898 requests. Both of these tests used 35 client threads, which were distributed across five machines. The tests ran for five minutes. The performance differences listed in the following table show a dramatic difference in processor usage.

	Using ADO w/out class wrapping		Using ADO with class wrapping	
	Average	Avg per Page	Average	Avg per Page
System\% Total Processor Time	92	9.39	90	30.07
Processor(0)\% Processor Time	92	9.39	90	30.07
Processor(0)\DPC Rate	5	0.51	6	2.01
Processor(0)\Interrupts/sec	974	99.42	1048	350.11
Processor(0)\DPCs Queued/sec	700	71.45	825	275.61
Process(inetinfo)\% Processor Time	82	8.37	79	26.39
Process(inetinfo)\% Privileged Time	24	2.45	17	5.68
Process(inetinfo)\% User Time	57	5.82	62	20.71
System\Context Switches/sec	10163	1037.39	5652	1888.20
System\System Calls/sec	19404	1980.67	11133	3719.27
Process(inetinfo)\Thread Count	67	6.84	46	15.37

The performance differences discovered in the preceding tests occur because of the cost of object creation. However, this does not mean the data access code in a Web application like Visual Basic Live cannot be encapsulated. A logical data tier can be achieved by putting all data access code in well-designed procedures. The data access procedures should then be grouped together in a standard module or a set of standard modules. This design achieves the goals of creating a separate data access tier well. The data access code is separated from the business rules and client code; it is easy to identify units of data access code for maintenance, and the data access code is in reusable units (procedures). Furthermore, because all the data access code resides in the WebClass application, there is only one copy of the code on the server, which allows deploying a new version of the code to be easy. The source code of Visual Basic Live demonstrates how to create a logical data access tier in a Visual Basic standard module without the cost of unneeded object creation.

For more information regarding the design choices and techniques used in Visual Basic Live, go to the Visual Basic Live Web site at <http://msdn.microsoft.com/vbasic/downloads/default.asp>. There the source code to Visual Basic Live can be downloaded and used as a guide for implementing any of the techniques explained in this document. Furthermore, Visual Basic Live has a list of samples and documents, which is growing regularly. Each of them is designed to guide developers in creating powerful Web sites using the Visual Basic development system.

---

*Send feedback* to MSDN. [Look here](#) for MSDN Online resources.