

Popups without Prejudice

by Dino Esposito

Banner ads are common today on the Web. Many people hate them; some people probably love them (though we don't know anyone who does). But while annoying, banner ads prove growing interest in the commercial aspects of the Web. The proliferation of advertising popups demonstrates that there's life (and money) beyond Web sites. Good news.

Typically, banner ads display in three ways: static or animated images, popup windows, or popunder windows. In the first case, the image is linked to the page and might be rotated on the server when the host page posts back. Since the advertising banner is part of the page, there's nothing the user can do about it, except ignore it.

Popup and popunder windows are a more interactive way of displaying advertisements over the Web. Because popups and popunders are regular windows, users can close or minimize them if not interested. Still, to most users, a new unsolicited browser window popping up for each page request is pretty annoying. That's why popup blockers such as the MSN Toolbar and the Google Toolbar are very popular downloads these days.

Reassessing Popups

But while popups have a poor reputation now, they're really a neutral programming resource, not necessarily a bad one. What people probably object to most about popups are the content they display. For popups, contents do matter!

Popup windows are just one more developer tool—neither the holy grail of the Web UI nor a feature to avoid at all costs. And popups are useful to Web developers who need to show relatively static and read-only data in response to a specific user click. Popups don't affect the site navigation mechanism: the caller page is still there, and the viewed page is not added to the history and is therefore not subject to the Back button.

Popup windows are made of pure client-side code; typically, a Javascript function calls into the HTML object model of the browser. You get no specific support from ASP.NET for injecting popup windows into applications, and it looks like that won't change in ASP.NET 2.0. But creating popups isn't exactly rocket science. It might even be a bit boring because of the script code required and because of the string-based interface for styling the dialog box.

In this article, I'll build a UI-less Web control that assists you to generate the Javascript code required for popup windows. A control is a reusable class that supplies an object-oriented programming model and abstracts most of the typical difficulties of managing Javascript blocks. Drop one instance of this control in an ASP.NET page and you can generate a popup call to fire at startup or in response to a client-side click.

Script-based Windows

In the HTML object model of most browsers, the window object contains a couple of methods to create modal and modeless dialog boxes. The methods, `showModalDialog` and `showModelessDialog`, share the same signature, as shown below.

```
window.showModalDialog(url [, params] [, style]);  
window.showModelessDialog(url [, params] [, style]);
```

Both methods takes a URL and a couple of optional arguments. The URL indicates the page to display in the popup window. The `params` argument represents a Javascript object that carries over external values for the page to process. The `style` argument is a semicolon-separated string that contains visual settings for the window like scrollbars, resize grips, help button, status bar and so forth.

To create a popup window from within ASP.NET pages, you inject some script code that makes a call to any of the preceding methods. The script code can be attached to any client-side buttons (i.e., non-submit input fields) or invoked at startup.

```
<script language="javascript">  
function ShowPopup()  
{  
    var url = "...";  
    window.showModalDialog(url, "", "");  
}  
ShowPopup();  
&lt;/script&gt;
```

As a Web developer, I try to use Javascript code as much as possible. I also make a point of showing customers the importance of structuring the client code used in Web pages and, wherever possible, retrieving it from a sort of centralized dispenser. Padding pages with repeated loops that accumulate predefined blocks of script code into a string is not really a neat and elegant approach. In the long run, it also represents a significant limitation for effective code maintenance.

In ASP.NET 2.0, most of the script code you find in pages—there is much more of it than in ASP.NET 1.x—is injected through a system HTTP handler named `webresource.axd`. This HTTP handler takes a code and some query string parameters and uses this information to select a script block out of an in-memory data store and complete it with external arguments. An alternative approach to smart scripting in ASP.NET entails developing a bunch of custom controls. This technique is more helpful when the script you're going to pack into a control identifies a specific function and allows for a good deal of customization.

At a minimum, a popup Web control would supply a style object plus a couple of properties for the URL and modality. The generated Javascript source code will be inserted in the form's tag or attached to a client-side button. Enter the new Wintellect.Controls.Popup Web control.

The Popup Server Control

The Popup control is a server control that provides no UI elements, but adds some script code to the response markup. The control class is declared as follows:

```
[DefaultProperty("Url"),  
ToolboxData("<{0}:Popup runat=\"server\" />")]  
public class Popup : Control  
{  
    :  
}
```

The control inherits from Control because it doesn't need any of the UI-specific properties that WebControl—another good potential base class—adds up. When you write a server control, it is a good practice to qualify the class with a couple of design-time attributes—DefaultProperty and ToolboxData. The former sets the control's property that receives the focus when the control is selected in the designer. The latter attribute sets the default markup that is inserted in the page when the control is dropped onto the Web form from the toolbox.

The Popup control provides four properties—ControlToBind, ShowModal, Url, and WindowStyle.

Property	Type	Description
ControlToBind	String	Name of another control in the page that, if clicked, triggers the popup window. If this property is null, the popup is displayed at startup.
ShowModal	Boolean	Indicates if the popup window must be modal (default) or modeless.
Url	String	Name of the URL to display in the popup window.
WindowStyle	PopupStyleObject	that represents the style of the popup.

Three of these properties—Url, ShowModal, and ControlToBind—are persisted in the viewstate.

```
public bool ShowModal
{
    get { return
    Convert.ToBoolean(ViewState["ShowModal"].ToString()); }
    set { ViewState["ShowModal"] = value; }
}
public string Url
{
    get { return ViewState["Url"].ToString(); }
    set { ViewState["Url"] = value; }
}
public string ControlToBind
{
    get { return ViewState["ControlToBind"].ToString(); }
    set { ViewState["ControlToBind"] = value; }
}
```

The ShowModal property influences the method on the HTML window object that is called to show the popup. By default, the Popup control uses a modal dialog box to display the content at the specified URL when the page completes the initialization of the form. If you want to display the popup window when the user clicks on a certain HTML element, assign the name of this control to the ControlToBind property. The control will do the rest by attaching some Javascript code to the bound control's onclick event.

Now let's review the rendering process of the Popup control. I'll start with the style properties.

Generating the Script

Any server control is primarily a class; as such, you can instantiate it and use it at will through the set of public members. My goal here is to build a dual-purpose component: a drag-and-drop control from the Visual Studio .NET toolbox and a reusable class to create and inject multiple calls to popups in the same page. I don't want to force users to use multiple instances of the control if they need multiple popups.

To accomplish the first mission, you need to override the Render method so that the following scenario works as expected:

1. Drop the control onto the Web form
2. Set its properties at design time
3. Run the page

The Popup control has no graphical UI but injects some script code in the page. The RegisterPopupFunction method shows how to generate the Javascript code and inserts it in the page.

```
public void RegisterPopupFunction(string ctlID, string funcName)
{
    RegisterPopupFunction("", funcName);
}

public void RegisterPopupFunction(string ctlID, string funcName)
{
    ControlToBind = ctlID;
    string js = BuildScript(funcName);
    string name = GetFunctionName(funcName);
    if (!Page.IsStartupScriptRegistered(name))
        Page.RegisterStartupScript(name, js);
}
```

The function takes the name of the client control to bind to (if any) and the name of the Javascript function to create. The name of the function is prefixed with the ID of the popup control to create a sort of naming container to guarantee name uniqueness. Once the Javascript string is ready, it gets added to the page collection of script through the RegisterStartupScript page method.

If the name of the control to bind to is empty, the popup is displayed at startup. The function name defaults to Show. Here's the Render method of the control. In the default case, the popup shows up at startup and is given a fixed name.

```
protected override void Render(HtmlTextWriter writer)
{
    base.Render(writer);
    RegisterPopupFunction("Show");
}
```

At this point, I have two methods doing similar things—Render and RegisterPopupFunction. The Render method is always called because it is part of the .NET Framework. What happens if you call RegisterPopupFunction explicitly? Two identical Javascript functions are injected in the page with different names. There's no conflict between the two, yet unneeded code is downloaded to the client. The goal of RegisterPopupFunction is enabling one popup control to generate as many popup windows as needed with different function names and pointing to different URLs. It should be clear that without a method like RegisterPopupFunction, you need N different Popup controls for N popup windows.

Is there a way to make the two methods mutually exclusive? One way is introducing another Boolean property—EnableRender. The property defaults to true; set it to false if you plan to call RegisterPopupFunction yourself. Setting EnableRender to false disables the Render method; only its base class can then be called.

```
protected override void Render(HtmlTextWriter writer)
{
    base.Render (writer);
    if (EnableRender)
        RegisterPopupFunction("Show");
}
```

The BuildScript internal function creates the set of Javascript instructions.

```
private string BuildScript(string funcName)
{
    string fName = GetFunctionName(funcName);

    StringBuilder scriptBuilder = new StringBuilder(
        "<script language=javascript>\r\n");
    scriptBuilder.AppendFormat("function {0}()", fName);
    scriptBuilder.Append(" {");
    scriptBuilder.AppendFormat("window.{0}(\\"{1}\", \\"{2}\",
        \\"{3}\");",
        (ShowModal ? "showModalDialog" : "showModelessDialog"),
        Url,
        "",
        WindowStyle.ToString());
    scriptBuilder.Append("}\r\n");

    // Startup script?
    if (ControlToBind == "")
        scriptBuilder.AppendFormat("{0}()\r\n", fName);
    else
    {
        // Find the associated control
        Control ctl = FindControl(ControlToBind);
        if (ctl != null)
        {
            if (ctl is WebControl)
            {
                WebControl webctl = (WebControl) ctl;
                webctl.Attributes["onclick"] = String.Format("{0}()",
                    fName);
            }
            if (ctl is HtmlControl)
            {
                HtmlControl htmctl = (HtmlControl) ctl;
                htmctl.Attributes["onclick"] = String.Format("{0}()",
                    fName);
            }
        }
    }
}
```

```
    }  
  }  
  
  scriptBuilder.Append(@"</script>");  
  
  // Return  
  string script = scriptBuilder.ToString();  
  return script;  
}  
  
private string GetFunctionName(string fname)  
{  
    string funcName = String.Format("{0}_{1}", UniqueID,  
    fname);  
    return funcName;  
}
```

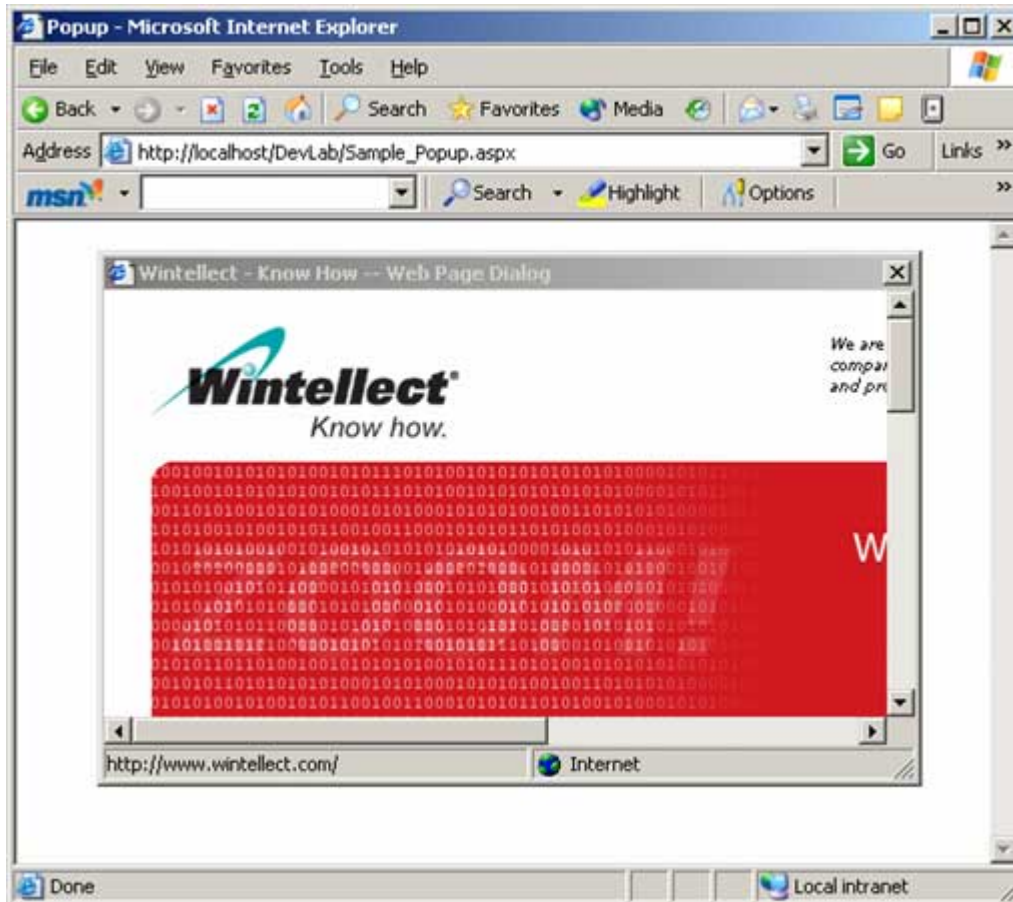
If ControlToBind is not the empty string, you retrieve the control with that ID and append a function call to its onclick event. The click event is arbitrary; you can opt for other events in alternative or in addition to onclick. To get a reference to the instance of the control, use the FindControl method.

Testing the Popup Control

Testing the popup control is as easy as adding the control to the Visual Studio .NET toolbox and dropping an instance onto a Web form. At this point, all that you have to do is set the Url property and configure the popup as required. Options include the modality of the dialog box, the use of the Render method, and the time at which the popup displays (startup or upon clicking). Here's a quick example:

```
<%@ Register TagPrefix="cc1" Namespace="Expoware.Controls"  
    Assembly="MySystem" %>  
:  
<form id="Form1" method="post" runat="server">  
    <cc1:Popup id="Popup1" runat="server"  
    url="www.wintellect.com" />  
</form>
```

The sample page is shown below.



The page contains the following code.

```
<body>
  <form name="Form1" method="post"
action="Sample_Popup.aspx">
  :
  <script language=javascript>
    function Popup1_Show() {
      window.showModelessDialog("http://www.wintellect.com",
        "", "");
    }
    Popup1_Show();
  </form>
</body>
```

To bind the popup to the click action on a client-side control (i.e., a non-submit button), add the following code to the code-behind class:

```
Popup1.ControlToBind = "MyBtn";
```

MyBtn is the name of a server control like the one below.

```
<button runat="server" id="MyBtn">Popup</button>
```

As mentioned, the control must be a client-side one because you don't want postbacks. In addition, it has to be marked `runat=server` because its ID must be retrieved using `FindControl`. The code in the final page changes as follows.

```
<body>
  <form name="Form1" method="post"
  action="Sample_Popup.aspx">
    :
    <button id="MyBtn" onclick="Popup1_Show();">Popup</button>
    <script language=javascript>
      function Popup1_Show() {
        window.showModelessDialog("http://www.wintellect.com",
          "", "");
      }
    </script>
  </form>
</body>
```

In this case, the call to the `Popup1_Show` function is delayed until the client-side button is clicked.

Styling the Popup Window

If you read the documentation for the browser's window object, you saw that the dialog boxes support a bunch of visual styles—scrollability, status bar, help button and a few more. All style properties must be grouped and serialized to a string that will be passed as the third argument to the call. Since I'm building a true ASP.NET server control, it goes without saying that I have to define a style object. Here's an example.

```
public class PopupStyle
{
  public PopupStyle()
  {
    Resizable = false;
  }
}
```

```
ShowStatus = true;
Centered = true;
ShowHelp = false;
Scrollable = true;
Width = Unit.Pixel(600);
Height = Unit.Pixel(400);
LeftPos = -1;
TopPos = -1;
}

// Members
public bool Resizable;
public bool ShowStatus;
public bool Centered;
public bool ShowHelp;
public bool Scrollable;
public Unit Width;
public Unit Height;
public int LeftPos;
public int TopPos;

// ToString
public override string ToString()
{
    StringBuilder builder = new StringBuilder("");

    if (LeftPos > 0)
        builder.AppendFormat("dialogLeft: {0};", LeftPos);
    if (TopPos > 0)
        builder.AppendFormat("dialogTop: {0};", LeftPos);
    builder.AppendFormat("dialogWidth: '{0}';", Width);
    builder.AppendFormat("dialogHeight: '{0}';", Height);
    builder.AppendFormat("center: {0};", (Centered ? "yes"
: "no"));
    builder.AppendFormat("scroll: {0};", (Scrollable ? "yes"
: "no"));
    builder.AppendFormat("resizable: {0};", (Resizable ? "yes"
: "no"));
    builder.AppendFormat("help: {0};", (ShowHelp ? "yes"
: "no"));
    builder.AppendFormat("status: {0};", (ShowStatus ? "yes"
: "no"));
    return builder.ToString();
}
}
```

The `PopupStyle` class sports a bunch of UI properties, each of which corresponds to a style attribute recognized by the HTML window object. The `ToString` method of the class is overridden to produce a valid string based on the values of the properties.

The `WindowStyle` property is just an instance of this class. It is implemented as follows:

```
private PopupStyle m_windowStyle = null;
public PopupStyle WindowStyle
{
    get
    {
        if (m_windowStyle == null)
            m_windowStyle = new PopupStyle();
        return m_windowStyle;
    }
}
```

In theory, you could set any of the style properties in the Visual Studio .NET page designer. In practice, this feature requires some specific design-time support the `Popup` control doesn't include. As a result, in this example the style attributes can only be set programmatically.

```
Popup1.WindowStyle.Resizable = true;
Popup1.WindowStyle.Width = Unit.Pixel(200);
Popup1.WindowStyle.ShowStatus = false;
```

When the internal `BuildScript` member constructs the Javascript call, it renders all the style settings to a string using a rather familiar syntax:

```
WindowStyle.ToString();
```

You now have a `Popup` server control that can easily inject some popup calls into your Web pages. You can control the style of the window, URL, and modality using a familiar and object-based syntax. In addition, you can use the control to generate a single popup call (default) or use its `RegisterPopupFunction` public method to generate as many popup calls as needed.

Can you ask for more?

The `Popup` control gives you a better way to introduce popups inside Web-based applications. The real questions are, should you use popups? In what scenarios? To do what?

Problems with Popups

Using popups may now be easy thanks to the Popup control, but it is not free of problems. The first issue is popup blockers. Before you implement a certain function through popup windows, you should find out about the browser's capabilities and possible blocker tools. You can't just ask users of a Internet application to enable popups, or otherwise your pages won't work as expected.

In addition, there's no standard for the behavior of popup windows. As a result, the behavior of a popup window may change (even significantly) from browser to browser. For fun, take an online tour of **MyIE2**. However, you may need to worry even if you limit your pages to Microsoft browsers. The Windows XP SP2 changes quite a few ways in which Internet Explorer works on up-to-date Windows XP systems. A couple changes just regard popup windows. In brief, Internet Explorer will attempt to block any modal and modeless windows, plus windows opened from scripts. Overlapping DHTML elements will be accepted only if they're created using `window.createPopup`, and with the limitation of at most one popup element per page. Internet Explorer under Windows XP SP2 won't block any popup dialog that is the direct result of a user action (such as clicking a client button). Read more here.

The final issue with popups is accessibility. Popups are not exactly friendly to people using assistive software and should be limited (if there's no other better way) to open drill-down pages that expand on a feature.

Popups' Opportunities

Popups are not the software impersonation of Evil. There's at least one scenario in which popups fit in very well. Imagine you're presenting a list of records for your users to select. When the user clicks, you need to drill down into the data and provide a sort of details view. In this case, a modal dialog might offer the same functionality without impacting the navigation system of the site. Alan Cooper—the guru of UI—is also a big fan of this **approach**. By using popups you don't ask users to navigate away from the main window to see some details of the current page.

While using popups to drill down into available data is definitely a good option, and might result in improved design and usability, implementing the feature might not be trivial. There are two key drawbacks: popup blockers and developmental issues.

To work around blockers, you can limit the implementation of the feature to intranet sites where you can more easily control the client browsers used.

Popups' Developmental Issues

Popups are simple to develop as long as you display relatively readonly data. If you display links and submit buttons, be prepared to control the target of those links. My favorite trick consists in wrapping the target page in an intermediate page that simply contains an `IFRAME` tag with width and height set to 100%.

Another developmental problem to solve involves input parameters and return values. Popup windows support a parameter that you can set with any Javascript object, thus passing any data. The issue is this is client-side data!

A better approach in an ASP.NET application is perhaps instructing the viewed page to retrieve its data from Session or Cache. In other words, you come up with your own protocol for passing arguments and configure child pages. This feature doesn't come for free, either.

Summary

Sometimes popups are good and sometimes they're bad. They're good for intranet applications because users can interact with the application without navigating away from the main page. They're bad for Internet applications not so much because of their inherent functionality but because of the popup blockers or tabbed browsers that are growing in popularity.

Whatever reason you decide to opt for popups, the Wintellect Popup control allows you to create that code using an easier object-based approach.

Note #1: In this article, I didn't cover DHTML-based solutions for building fake popups. Most of these tricks will be blocked and disabled by Internet Explorer under Windows XP SP2.

Note #2: I want to thank all the blog-people who participated in an online survey on the subject of popups. To see posts and comments just join **my blog**.