**Introduction to ASP.NET WebForms**





[**http://msdn.microsoft.com/rampup**](http://msdn.microsoft.com/rampup)

Contents

[Introduction 3](#_Toc223525199)

[Page Model 4](#_Toc223525200)

[Page Directives 8](#_Toc223525201)

[Code behind (Server) 9](#_Toc223525202)

[Server Controls 11](#_Toc223525203)

[Client Controls 13](#_Toc223525204)

[Adding Controls Programmatically 14](#_Toc223525205)

[Web User Controls 16](#_Toc223525206)

## Introduction

Web Forms are the heart and soul of ASP.NET. Web Forms are the User Interface (UI) elements that give your Web applications their look and feel. Web Forms are similar to Windows Forms in that they provide properties, methods, and events for the controls that are placed onto them. However, these UI elements render themselves in the appropriate markup language required by the request, e.g. HTML. If you use Microsoft Visual Studio® .NET, you will also get the familiar drag-and-drop interface used to create your UI for your Web application.

Web Forms are made up of two components: the visual portion (the ASPX file), and the code behind the form, which resides in a separate class file.

Web Forms and ASP.NET were created to overcome some of the limitations of ASP. These new strengths include:

* Separation of HTML interface from application logic
* A rich set of server-side controls that can detect the browser and send out appropriate markup language such as HTML
* Less code to write due to the data binding capabilities of the new server-side .NET controls
* Event-based programming model that is familiar to Microsoft Visual Basic® programmers
* Compiled code and support for multiple languages, as opposed to ASP which was interpreted as Microsoft Visual Basic Scripting (VBScript) or Microsoft Jscript®
* Allows third parties to create controls that provide additional functionality

On the surface, Web Forms seem just like a workspace where you draw controls. In reality, they can do a whole lot more. But normally you will just place any of the various controls onto the Web Form to create your UI. The controls you use determine which properties, events, and methods you will get for each control. There are two types of controls that you can use to create your user interface: HTML controls and Web Form controls.

## Page Model

Each request for a Microsoft® ASP.NET page that hits Microsoft® Internet Information Services (IIS) is handed over to the ASP.NET HTTP pipeline. The HTTP pipeline is a chain of managed objects that sequentially process the request and make the transition from a URL to plain HTML text happen. The entry point of the HTTP pipeline is the **HttpRuntime** class. The ASP.NET infrastructure creates one instance of this class per each AppDomain hosted within the worker process (remember that the worker process maintains one distinct AppDomain per each ASP.NET application currently running).

The **HttpRuntime** class picks up an **HttpApplication** object from an internal pool and sets it to work on the request. The main task accomplished by the HTTP application manager is finding out the class that will actually handle the request. When the request is for an .aspx resource, the handler is a page handler—namely, an instance of a class that inherits from **Page**. The association between types of resources and types of handlers is stored in the configuration file of the application. More exactly, the default set of mappings is defined in the **<httpHandlers>** section of the machine.config file. However, the application can customize the list of its own HTTP handlers in the local web.config file. The line below illustrates the code that defines the HTTP handler for .aspx resources.

<add verb="\*" path="\*.aspx" type="System.Web.UI.PageHandlerFactory"/>

An extension can be associated with a handler class, or more in general, with a handler factory class. In all cases, the **HttpApplication** object in charge for the request gets an object that implements the **IHttpHandler** interface. If the association resource/class is resolved in terms of a HTTP handler, then the returned class will implement the interface directly. If the resource is bound to a handler factory, an extra step is necessary. A handler factory class implements the **IHttpHandlerFactory** interface whose **GetHandler** method will return an IHttpHandler-based object.

How can the HTTP run time close the circle and process the page request? The **IHttpHandler** interface features the **ProcessRequest** method. By calling this method on the object that represents the requested page, the ASP.NET infrastructure starts the process that will generate the output for the browser.

Once the HTTP page handler class is fully identified, the ASP.NET run time calls the handler's **ProcessRequest** method to process the request. Normally, there is no need to change the implementation of the method as it is provided by the **Page** class.

This implementation begins by calling the method **FrameworkInitialize,** which builds the controls tree for the page. The method is a protected and virtual member of the **TemplateControl** class—the class from which **Page** itself derives. Any dynamically generated handler for an .aspx resource overrides **FrameworkInitialize**. In this method, the whole control tree for the page is built.

Next, **ProcessRequest** makes the page transit various phases: initialization, loading of view state information and postback data, loading of the page's user code and execution of postback server-side events. After that, the page enters in rendering mode: the updated view state is collected; the HTML code is generated and then sent to the output console. Finally, the page is unloaded and the request is considered completely served.

During the various phases, the page fires a few events that Web controls and user-defined code can intercept and handle. Some of these events are specific for embedded controls and subsequently can't be handled at the level of the .aspx code.

A page that wants to handle a certain event should explicitly register an appropriate handler. However, for backward compatibility with the earlier Visual Basic programming style, ASP.NET also supports a form of implicit event hooking. By default, the page tries to match special method names with events; if a match is found, the method is considered a handler for the event. ASP.NET provides special recognition of six method names. They are **Page\_Init**, **Page\_Load**, **Page\_DataBind**, **Page\_PreRender**, and **Page\_Unload**. These methods are treated as handlers for the corresponding events exposed by the **Page** class. The HTTP run time will automatically bind these methods to page events saving developers from having to write the necessary glue code. For example, the method named **Page\_Load** is wired to the page's **Load** event as if the following code was written.

this.Load += new EventHandler(this.Page\_Load);

The automatic recognition of special names is a behavior under the control of the **AutoEventWireup** attribute of the **@Page** directive. If the attribute is set to false, any applications that wish to handle an event need to connect explicitly to the page event. Pages that don't use automatic event wire-up will get a slight performance boost by not having to do the extra work of matching names and events. You should note that all Microsoft Visual Studio® .NET projects are created with the **AutoEventWireup** attribute disabled. However, the default setting for the attribute is true, meaning that methods such as **Page\_Load** are recognized and bound to the associated event.

The execution of a page consists of a sequence of stages listed in the following table and is characterized by application-level events and/or protected, overridable methods.

**Key events in the life of an ASP.NET page**

|  |  |  |
| --- | --- | --- |
| **Stage** | **Page Event** | **Overridable method** |
| Page initialization | **Init** |   |
| View state loading |   | **LoadViewState** |
| Postback data processing |   | **LoadPostData** method in any control that implements the **IPostBackDataHandler** interface |
| Page loading | **Load** |   |
| Postback change notification |   | **RaisePostDataChangedEvent** method in any control that implements the **IPostBackDataHandler** interface |
| Postback event handling | Any postback event defined by controls | **RaisePostBackEvent** method in any control that implements the **IPostBackEventHandler** interface  |
| Page pre-rendering phase | **PreRender** |   |
| View state saving |   | **SaveViewState** |
| Page rendering |   | **Render** |
| Page unloading | **Unload** |   |

Some of the stages listed above are not visible at the page level and affect only authors of server controls and developers who happen to create a class derived from **Page**. **Init**, **Load**, **PreRender**, **Unload**, plus all postback events defined by embedded controls are the only signals of life that a page sends to the external world.

##### Stages of Execution

The first stage in the page lifecycle is the initialization. This stage is characterized by the **Init** event, which fires to the application after the page's control tree has been successfully created. In other words, when the **Init** event arrives, all the controls statically declared in the .aspx source file have been instantiated and hold their default values. Controls can hook up the **Init** event to initialize any settings that will be needed during the lifetime of the incoming Web request. For example, at this time controls can load external template files or set up the handler for the events. You should notice that no view state information is available for use yet.

Immediately after initialization, the page framework loads the view state for the page. The view state is a collection of name/value pairs, where controls and the page itself store any information that must be persistent across Web requests. The view state represents the call context of the page. Typically, it contains the state of the controls the last time the page was processed on the server. The view state is empty the first time the page is requested in the session. By default, the view state is stored in a hidden field silently added to the page. The name of this field is **\_\_VIEWSTATE**. By overriding the **LoadViewState** method—a protected overridable method on the **Control** class—component developers can control how the view state is restored and how its contents are mapped to the internal state.

Methods like **LoadPageStateFromPersistenceMedium** and its counterpart **SavePageStateToPersistenceMedium** can be used to load and save the view state to an alternative storage medium—for example, Session, databases, or a server-side file. Unlike **LoadViewState**, the aforementioned methods are available only in classes derived from **Page**.

Once the view state has been restored, the controls in the page tree are in the same state they were the last time the page was rendered to the browser. The next step consists of updating their state to incorporate client-side changes. The postback data-processing stage gives controls a chance to update their state so that it accurately reflects the state of the corresponding HTML element on the client. For example, a server **TextBox** control has its HTML counterpart in an **<input type=text>** element. In the postback data stage, the **TextBox** control will retrieve the current value of <input> tag and use it to refresh its internal state. Each control is responsible for extracting values from posted data and updating some of its properties. The **TextBox** control will update its **Text** property whereas the **CheckBox** control will refresh its **Checked** property. The match between a server control and a HTML element is found on the ID of both.

At the end of the postback data processing stage, all controls in the page reflect the previous state updated with changes entered on the client. At this point, the **Load** event is fired to the page.

There might be controls in the page that need to accomplish certain tasks if a sensitive property is modified across two different requests. For example, if the text of a textbox control is modified on the client, the control fires the **TextChanged** event. Each control can take the decision to fire an appropriate event if one or more of its properties are modified with the values coming from the client. Controls for which these changes are critical implement the **IPostBackDataHandler** interface, whose **LoadPostData** method is invoked immediately after the **Load** event. By coding the **LoadPostData** method, a control verifies if any critical change has occurred since last request and fires its own change event.

The key event in the lifecycle of a page is when it is called to execute the server-side code associated with an event triggered on the client. When the user clicks a button, the page posts back. The collection of posted values contains the ID of the button that started the whole operation. If the control is known to implement the **IPostBackEventHandler** interface (buttons and link buttons will do), the page framework calls the **RaisePostBackEvent** method. What this method does depends on the type of the control. With regard to buttons and link buttons, the method looks up for a **Click** event handler and runs the associated delegate.

After handling the postback event, the page prepares for rendering. This stage is signaled by the **PreRender** event. This is a good time for controls to perform any last minute update operations that need to take place immediately before the view state is saved and the output rendered. The next state is **SaveViewState**, in which all controls and the page itself are invited to flush the contents of their own **ViewState** collection. The resultant view state is then serialized, hashed, Base64 encoded, and associated with the \_\_VIEWSTATE hidden field.

The rendering mechanism of individual controls can be altered by overriding the **Render** method. The method takes an HTML writer object and uses it to accumulate all HTML text to be generated for the control. The default implementation of the **Render** method for the **Page** class consists of a recursive call to all constituent controls. For each control the page calls the **Render** method and caches the HTML output.

The final sign of life of a page is the **Unload** event that arrives just before the page object is dismissed. In this event you should release any critical resource you might have (for example, files, graphical objects, database connections).

Finally, after this event the browser receives the HTTP response packet and displays the page.

##    Page Directives

Directives specify settings that are used by the page and user-control compilers when the compilers process ASP.NET Web Forms pages (.aspx files) and user control (.ascx) files.

ASP.NET treats any directive block (<%@ %>) that does not contain an explicit directive name as an @ Page directive (for a page) or as an @ Control directive (for a user control).

For syntax information and descriptions of the attributes that are available for each directive, use the links that are listed in the following table.

|  |  |
| --- | --- |
| **Directive**  | **Description**  |
| [@ Assembly](http://msdn.microsoft.com/en-us/library/d864zc1k.aspx)  | Links an assembly to the current page or user control declaratively.  |
| [@ Control](http://msdn.microsoft.com/en-us/library/d19c0t4b.aspx)  | Defines control-specific attributes used by the ASP.NET page parser and compiler and can be included only in .ascx files (user controls).  |
| [@ Implements](http://msdn.microsoft.com/en-us/library/cbsf6k72.aspx)  | Indicates that a page or user control implements a specified .NET Framework interface declaratively.  |
| [@ Import](http://msdn.microsoft.com/en-us/library/eb44kack.aspx)  | Imports a namespace into a page or user control explicitly.  |
| [@ Master](http://msdn.microsoft.com/en-us/library/ms228176.aspx)  | Identifies a page as a master page and defines attributes used by the ASP.NET page parser and compiler and can be included only in .master files.  |
| [@ MasterType](http://msdn.microsoft.com/en-us/library/ms228274.aspx)  | Defines the class or virtual path used to type the [Master](http://msdn.microsoft.com/en-us/library/system.web.ui.page.master.aspx) property of a page.  |
| [@ OutputCache](http://msdn.microsoft.com/en-us/library/hdxfb6cy.aspx)  | Controls the output caching policies of a page or user control declaratively.  |
| [@ Page](http://msdn.microsoft.com/en-us/library/ydy4x04a.aspx)  | Defines page-specific attributes used by the ASP.NET page parser and compiler and can be included only in .aspx files.  |
| [@ PreviousPageType](http://msdn.microsoft.com/en-us/library/ms228169.aspx)  | Creates a strongly typed reference to the source page from the target of a cross-page posting.  |
| [@ Reference](http://msdn.microsoft.com/en-us/library/w70c655a.aspx)  | Links a page, user control, or COM control to the current page or user control declaratively.  |
| [@ Register](http://msdn.microsoft.com/en-us/library/c76dd5k1.aspx)  | Associates aliases with namespaces and classes, which allow user controls and custom server controls to be rendered when included in a requested page or user control.  |

## Code behind (Server)

Code-behind refers to code for your ASP.NET page that is contained within a separate class file. This allows a clean separation of your HTML from your presentation logic. The following sample illustrates an ASP.NET code-behind page:

**MyCodebehind.aspx**

<%@ Language="C#" Inherits="MyStuff.MyClass" %>

<HTML>

 <body>

 <form id="MyForm" runat="server">

 <asp:textbox id="MyTextBox" text="Hello World" runat="server"></asp:textbox>

 <asp:button id="MyButton" text="Echo Input" Onclick="MyButton\_Click" runat="server"></asp:button>

 <asp:label id="MyLabel" runat="server" />

 </form>

 </body>

</HTML>

**Mycodebehind.cs**

using System;

using System.Web;

using System.Web.UI;

using System.Web.UI.WebControls;

namespace MyStuff

{

 public class MyClass : Page

 {

 protected System.Web.UI.WebControls.Label MyLabel;

 protected System.Web.UI.WebControls.Button MyButton;

 protected System.Web.UI.WebControls.TextBox MyTextBox;

 public void MyButton\_Click(Object sender, EventArgs e)

 {

 MyLabel.Text = MyTextBox.Text.ToString();

 }

 }

}

In the preceding sample, you can use the following syntax to compile Mycodebehind.cs:

**csc.exe /out:mycodebehind.dll /t:library mycodebehind.cs**

When you use the following code, the code-behind page inherits from the **Page** class. The **Page** class resides in the **System.Web.UI** namespace:

public class MyClass : Page

Inheriting from the **Page** class gives the code-behind page access to the ASP.NET intrinsic objects, such as **Request** and **Response**. In addition, inheriting from the **Page** class provides a framework for handling events for controls within the ASP.NET page.

In the preceding sample, the code-behind page is compiled before ASP.NET runs. Alternatively, you can reference the code-behind class by using an SRC tag as follows:

<%@ Language="C#" Inherits="MyStuff.MyClass" src="MyCodebehind.cs" %>

In this case, ASP.NET compiles the code-behind page on the fly. Notice that this compilation step only occurs when the code-behind file is updated (which is detected through a timestamp change).

##### Code-Behind Support in Visual Studio .NET

When you use Microsoft Visual Studio .NET to create ASP.NET Web Forms, code-behind pages are the default method. In addition, Visual Studio .NET automatically performs precompilation for you when you build your solution. Note that code-behind pages that are created in Visual Studio .NET include a special page attribute, **Codebehind**, which Visual Studio .NET uses.

##     Server Controls

The ASP.NET page framework includes a number of built-in server controls that are designed to provide a more structured programming model for the Web. These controls provide the following features:

* Automatic state management.
* Simple access to object values without having to use the **Request** object.
* Ability to react to events in server-side code to create applications that are better structured.
* Common approach to building user interfaces for Web pages.
* Output is automatically customized based on the capabilities of the browser.

In addition to the built-in controls, the ASP.NET page framework also provides the ability to create user controls and custom controls. User controls and custom controls can enhance and extend existing controls to build a much richer user interface.

ASP.NET Web server controls are objects on ASP.NET Web pages that run when the page is requested and that render markup to the browser. Many Web server controls are similar to familiar HTML elements, such as buttons and text boxes. Other controls encompass complex behavior, such as a calendar controls, and controls that you can use to connect to data sources and display data.

#####  Standard Toolbox controls

There are a myriad of standard controls available to you in the standard toolbox controls. These include controls that enable you to display buttons, lists, images, boxes, hyperlinks, labels, tables, as well as more complicated controls that work with static and dynamic data or controls that act as containers for other controls. The following diagram lists these controls and displays where the base classes they’re inherited from.



##   Client Controls

In addition to using Web server controls in your ASP.NET Web pages, you can create your own custom, reusable controls using the same techniques you use for creating ASP.NET Web pages. These controls are called user controls.

A user control is a kind of composite control that works much like an ASP.NET Web page—you can add existing Web server controls and markup to a user control, and define properties and methods for the control. You can then embed them in ASP.NET Web pages, where they act as a unit.

##### HTML controls

HTML server controls are HTML elements (or elements in other supported markup, such as XHTML) containing attributes that make them programmable in server code. By default, HTML elements on an ASP.NET Web page are not available to the server. Instead, they are treated as opaque text and passed through to the browser. However, by converting HTML elements to HTML server controls, you expose them as elements you can program on the server.

The object model for HTML server controls maps closely to that of the corresponding elements. For example, HTML attributes are exposed in HTML server controls as properties.

Any HTML element on a page can be converted to an HTML server control by adding the attribute runat="server". During parsing, the ASP.NET page framework creates instances of all elements containing the runat="server" attribute. If you want to reference the control as a member within your code, you should also assign an id attribute to the control.

The page framework provides predefined HTML server controls for the HTML elements most commonly used dynamically on a page: the form element, the input elements (text box, check box, Submit button), the select element, and so on. These predefined HTML server controls share the basic properties of the generic control, and in addition, each control typically provides its own set of properties and its own event.

HTML server controls offer the following features:

* An object model that you can program against on the server using familiar object-oriented techniques. Each server control exposes properties that enable you to manipulate the control's markup attributes programmatically in server code.
* A set of events for which you can write event handlers in much the same way you would in a client-based form, except that the event is handled in server code.
* The ability to handle events in client script.
* Automatic maintenance of the control's state. When the page makes a round trip to the server, the values that the user entered into HTML server controls are automatically maintained and sent back to the browser.
* Interaction with ASP.NET validation controls so you can verify that a user has entered appropriate information into a control.
* Data binding to one or more properties of the control.
* Support for styles if the ASP.NET Web page is displayed in a browser that supports cascading style sheets.
* Pass-through of custom attributes. You can add any attributes you need to an HTML server control and the page framework will render them without any change in functionality. This enables you to add browser-specific attributes to your controls.

## Adding Controls Programmatically

Just as you can programmatically create an instance of any server control on an ASP.NET Web page, you can do the same with a user control.

To create an instance of a user control programmatically

1. In the user control, be sure that the @ Control directive contains a ClassName attribute that assigns a class to the user control.

The following example sets the ClassName attribute to strongly type a user control.

<%@ Control className="MyUserControl" %>

1. In the page where you want to work with the user control, create a reference to the user control with the @ Reference directive.

When you create the user control programmatically, the strong type for your user control is available to the ASP.NET Web page only after you have created a reference to it. For example, the following code creates a reference to a user control created in the MyUserControl.ascx file.

<%@ Reference Control="MyUserControl.ascx" %>

1. Create an instance variable for the user control, using the control's class name. The class will be part of the ASP namespace.

For example, if you want to create an instance of the user control declared as class Spinner, you use syntax such as the following:

C#

Protected ASP.Spinner Spinner1;

1. Create an instance of the user control in code by calling the LoadControl method.
2. Assign property values as necessary, and then add the control to the ControlCollection collection of a container on the page, such as a PlaceHolder control.

The following example shows an ASP.NET Web page that loads a user control programmatically. The page includes an @ Reference directive to specify the control's file. The LoadControl method reads the file and instantiates it as a control that can be added to the page.

[C#]

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

<%@ Reference Control="~/Controls/Spinner.ascx" %>

<script runat="server">

private ASP.Spinner Spinner1

protected void Page\_Load(object sender, EventArgs e)

{

 Spinner1 = (ASP.Spinner)LoadControl("~/Controls/Spinner.ascx");

}

protected void Button1\_Click(object sender, EventArgs e)

{

 PlaceHolder1.Controls.Add(Spinner1);

}

</script>

<html>

<head id="Head1" runat="server">

 <title>Load User Control Programmatically</title>

</head>

<body>

 <form id="form1" runat="server">

 <div>

 <asp:PlaceHolder runat="server" ID="PlaceHolder1" />

 <br />

 <asp:Button ID="Button1" runat="server"

 Text="Click to Add User Control"

 OnClick="Button1\_Click" />

 <br />

 <br />

 <asp:Label ID="Label1" runat="server" Text=""></asp:Label>

 </div>

 </form>

</body>

</html>

##   Web User Controls

At times, you might need functionality in a control that is not provided by the built-in ASP.NET Web server controls. In those cases, you can create your own controls. You have two options. You can create:

* User controls. User controls are containers into which you can put markup and Web server controls. You can then treat the user control as a unit and define properties and methods for it.
* Custom controls. A custom control is a class that you write that derives from Control or WebControl.

User controls are substantially easier to create than custom controls, because you can reuse existing controls. They make it particularly easy to create controls with complex user interface elements.

##### clear User Control Structure

An ASP.NET Web user control is similar to a complete ASP.NET Web page (.aspx file), with both a user interface page and code. You create the user control in much the same way you create an ASP.NET page and then add the markup and child controls that you need. A user control can include code to manipulate its contents like a page can, including performing tasks such as data binding.

A user controls differs from an ASP.NET Web page in these ways:

* The file name extension for the user control is .ascx.
* Instead of an @ Page directive, the user control contains an @ Control directive that defines configuration and other properties.
* User controls cannot run as stand-alone files. Instead, you must add them to ASP.NET pages, as you would any control.
* The user control does not have **html**, **body**, or **form** elements in it. These elements must be in the hosting page.

You can use the same HTML elements (except the **html**, **body**, or **form** elements) and Web controls on a user control that you do on an ASP.NET Web page. For example, if you are creating a user control to use as a toolbar, you can put a series of Button Web server controls onto the control and create event handlers for the buttons.



##### Control Directives

The user control looks much like an ASP.NET page —for example it may contain several controls (e.g. a TextBox control and two Button controls) and code that handles the buttons' Click events and the page’s Load event. However, the control contains no markup except for the controls, and instead of an @ Page directive it contains an @ Control directive.

<% @ Control Language="C#" ClassName="UserControl1" %>

<script runat="server">

 protected int currentColorIndex;

 protected String[] colors = {"Red", "Blue", "Green", "Yellow"};

 protected void Page\_Load(object sender, EventArgs e)

 {

 if (IsPostBack)

 {

 currentColorIndex =

 Int16.Parse(ViewState["currentColorIndex"].ToString());

 }

 else

 {

 currentColorIndex = 0;

 DisplayColor();

 }

 }

 protected void DisplayColor()

 {

 textColor.Text = colors[currentColorIndex];

 ViewState["currentColorIndex"] = currentColorIndex.ToString();

 }

 protected void buttonUp\_Click(object sender, EventArgs e)

 {

 if(currentColorIndex == 0)

 {

 currentColorIndex = colors.Length - 1;

 }

 else

 {

 currentColorIndex -= 1;

 }

 DisplayColor();

 }

 protected void buttonDown\_Click(object sender, EventArgs e)

 {

 if(currentColorIndex == (colors.Length - 1))

 {

 currentColorIndex = 0;

 }

 else

 {

 currentColorIndex += 1;

 }

 DisplayColor();

 }

</script>

<asp:TextBox ID="textColor" runat="server"

 ReadOnly="True" />

<asp:Button Font-Bold="True" ID="buttonUp" runat="server"

 Text="^" OnClick="buttonUp\_Click" />

<asp:Button Font-Bold="True" ID="buttonDown" runat="server"

 Text="v" OnClick="buttonDown\_Click" />

##### Converting ASPX file to ASCX file

If you have developed an ASP.NET Web page and would like to access its functionality throughout your application, you can make some minor alterations to the page to change it to a user control.

###### To convert a single-file ASP.NET Web page into a user control

1. Rename the control so the file name extension is .ascx.
2. Remove the html, body, and form elements from the page.
3. Change the [@ Page](http://msdn.microsoft.com/en-us/library/ydy4x04a.aspx) directive to an [@ Control](http://msdn.microsoft.com/en-us/library/d19c0t4b.aspx) directive.
4. Remove all attributes of the @ Control directive except Language, AutoEventWireup (if present), CodeFile, and Inherits.
5. Include a className attribute in the @ Control directive. This allows the user control to be strongly typed when it is added to a page.

###### To convert a code-behind ASP.NET Web page into a user control

1. Rename the .aspx file so the file name extension is .ascx.
2. Rename the code-behind file to have the file name extension .ascx.vb or .ascx.cs, depending on what programming language the code-behind file is in.
3. Open the code-behind file and change the class from which it inherits from [Page](http://msdn.microsoft.com/en-us/library/system.web.ui.page.aspx) to [UserControl](http://msdn.microsoft.com/en-us/library/system.web.ui.usercontrol.aspx).
4. In the .aspx file, do the following:
	1. Remove the html, body, and form elements from the page.
	2. Change the [@ Page](http://msdn.microsoft.com/en-us/library/ydy4x04a.aspx) directive to an [@ Control](http://msdn.microsoft.com/en-us/library/d19c0t4b.aspx) directive.
	3. Remove all attributes of the @ Control directive except Language, AutoEventWireup (if present), CodeFile, and Inherits.
	4. In the @ Control directive, change the CodeFile attribute to point to the renamed code-behind file.
5. Include a className attribute in the @ Control directive. This allows the user control to be strongly typed when it is added to a page.

##### C# ASPX Page

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

<html>

<script runat=server>

void EnterBtn\_Click(Object sender, EventArgs e)

{

 Label1.Text = "Hi " + Name.Text + " welcome to ASP.NET!";

}

</script>

<body>

<h3> <u>Web Forms Page</u> </h3>

<form>

Enter Name: <asp:textbox id="Name" runat=server/>

<asp:button Text="Enter" OnClick="EnterBtn\_Click"

 runat=server/>

<br>

<br>

<asp:label id="Label1" runat=server/>

</form>

</body>

</html>

##### C# Converted control

<%@ Control Language="C#" ClassName="SampleUserControl" %>

<h3> <u>User Control</u> </h3>

<script runat=server>

void EnterBtn\_Click(Object Sender, EventArgs e)

{

 Label1.Text = "Hi " + Name.Text + " welcome to ASP.NET!";

}

</script>

Enter Name: <asp:textbox id="Name" runat=server/>

<asp:button Text="Enter" OnClick="EnterBtn\_Click"

runat=server/>

<br>

<br>

<asp:label id="Label1" runat=server/>

#####  Registering ASCX files on hosting page

To use a user control, you include it in an ASP.NET Web page. When a request arrives for a page and that page contains a user control, the user control goes through all of the processing stages that any ASP.NET server control performs. For more information about these processing stages, see [ASP.NET Page Life Cycle Overview](http://msdn.microsoft.com/en-us/library/ms178472.aspx).

To include a user control in a Web Forms page

1. In the containing ASP.NET Web page, create an [@ Register](http://msdn.microsoft.com/en-us/library/c76dd5k1.aspx) directive that includes:
	* A TagPrefix attribute, which associates a prefix with the user control. This prefix will be included in opening tag of the user control element.
	* A TagName attribute, which associates a name with the user control. This name will be included in the opening tag of the user control element.
	* A Src attribute, which defines the virtual path to the user control file that you are including.
2. In the body of the Web page, declare the user control element inside the form element.
3. Optionally, if the user control exposes public properties, set the properties declaratively.

The following example shows an ASP.NET Web page that contains a user control. The user control is in the file Spinner.ascx in the Controls folder. In the page, the control is registered to use the prefix uc and the tag name Spinner. The user control properties MinValue and MaxValue are set declaratively.

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

<%@ Register TagPrefix="uc" TagName="Spinner"

 Src="~\Controls\Spinner.ascx" %>

<html>

<body>

<form runat="server">

 <uc:Spinner id="Spinner1"

 runat="server"

 MinValue="1"

 MaxValue="10" />

</form>

</body>

#####  Create instance of user controls programmatically

To Create an instance of a user control programmatically

1. In the user control, be sure that the @ Control directive contains a ClassName attribute that assigns a class to the user control.

The following example sets the ClassName attribute to strongly type a user control.

<%@ Control className="MyUserControl" %>

1. In the page where you want to work with the user control, create a reference to the user control with the @ Reference directive.

When you create the user control programmatically, the strong type for your user control is available to the ASP.NET Web page only after you have created a reference to it. For example, the following code creates a reference to a user control created in the MyUserControl.ascx file.

<%@ Reference Control="MyUserControl.ascx" %>

1. Create an instance variable for the user control, using the control's class name. The class will be part of the ASP namespace.

For example, if you want to create an instance of the user control declared as class Spinner, you use syntax such as the following:

C#

Protected ASP.Spinner Spinner1;

1. Create an instance of the user control in code by calling the LoadControl method.
2. Assign property values as necessary, and then add the control to the ControlCollection collection of a container on the page, such as a PlaceHolder control.

The following example shows an ASP.NET Web page that loads a user control programmatically. The page includes an @ Reference directive to specify the control's file. The LoadControl method reads the file and instantiates it as a control that can be added to the page.

[C#]

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

<%@ Reference Control="~/Controls/Spinner.ascx" %>

<script runat="server">

private ASP.Spinner Spinner1

protected void Page\_Load(object sender, EventArgs e)

{

 Spinner1 = (ASP.Spinner)LoadControl("~/Controls/Spinner.ascx");

}

protected void Button1\_Click(object sender, EventArgs e)

{

 PlaceHolder1.Controls.Add(Spinner1);

}

</script>

<html>

<head id="Head1" runat="server">

 <title>Load User Control Programmatically</title>

</head>

<body>

 <form id="form1" runat="server">

 <div>

 <asp:PlaceHolder runat="server" ID="PlaceHolder1" />

 <br />

 <asp:Button ID="Button1" runat="server"

 Text="Click to Add User Control"

 OnClick="Button1\_Click" />

 <br />

 <br />

 <asp:Label ID="Label1" runat="server" Text=""></asp:Label>

 </div>

 </form>

</body>

</html>