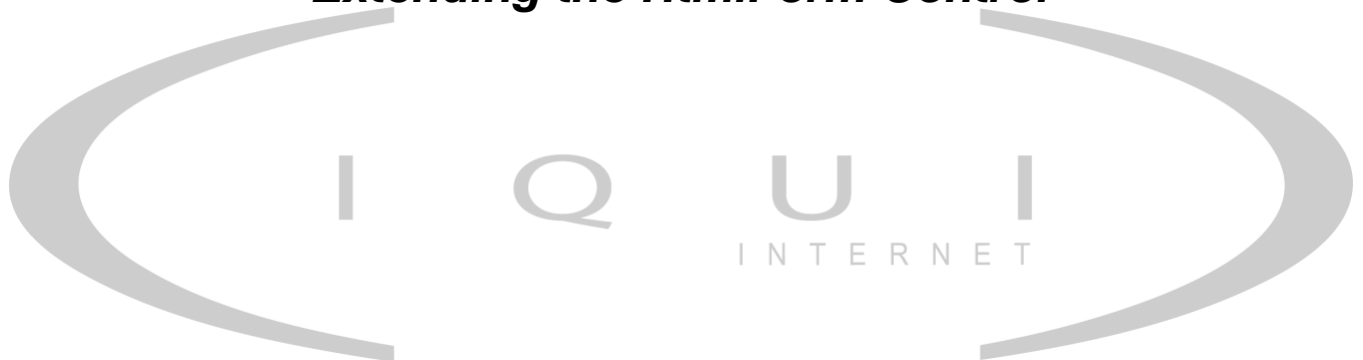


Valid XHTML within .NET

Extending the HtmlForm Control



Article Information

Author: Kevin Brown
Company: Liquid Internet Ltd
Date Created: 20th August 2003
Last Modified: 10th March 2006

Copyright

This document is distributed under the GNU Free Documentation License.
Copyright © 2003 Liquid Internet Limited.

Permission is granted to copy, distribute and / or modify this document under the terms of the GNU Free Documentation License, version 1.2 as published by the Free Software Foundation - <http://www.gnu.org/licenses/fdl.html>

Source code within this document is released under the terms of the GNU General Public License, version 2 as published by the Free Software Foundation - <http://www.gnu.org/licenses/gpl.html>

This document is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Free Documentation License for more details.

Liquid Internet Limited has made every effort to ensure the accuracy of this material. If you have any questions or comments, please contact:

E-mail: info@liquid-internet.co.uk
Website: <http://www.liquid-internet.co.uk/>

Document History

Version	Author	Comments	Date
1.0.3	Kevin Brown	Minor Text Amendments	10-03-2006
1.0.2	Kevin Brown	Added an Index and a Glossary	04-09-2003
1.0.1	Kevin Brown	Added additional screenshots	27-08-2003
1.0.0	Kevin Brown	Document created	20-08-2003

Contents

Valid XHTML within .NET	1
Extending the HtmlForm Control	1
Article Information	1
Copyright	2
Document History.....	3
Version.....	3
Author	3
Comments	3
Date	3
Contents	4
Series Abstract	5
Article Abstract.....	5
Who Should Read This Article?	5
Document Conventions.....	5
System Requirements.....	5
Assumptions.....	5
Limitations	6
The Problem.....	7
Suggested Solution.....	8
Step 1: Creating a Custom Assembly	8
Step 2: Creating a Custom Page	9
Step 3: Fixing the Name Attribute	10
Step 4: Fixing the View-State Field	11
Step 5: Adding Some Error Checking	13
Summary.....	14
Downloads	15
Appendix A: Resources	16
Appendix B: Glossary.....	17
Index.....	18

Series Abstract

Prior to the release of ASP.NET 2.0 none of Microsoft's ASP.NET (version 1.0 / 1.1) controls intrinsically supported the strict W3C XHTML standards (currently 1.0, 1.1 and 2.0). This can cause problems when using ASP.NET controls on pages that need to conform to one of the XHTML standards. In order to use Microsoft's supplied controls some modifications will be required. This series of articles intends to focus on the more widely used controls and the steps required to make them output valid code.

Article Abstract

The first article in this series will concentrate on the `HtmlForm` control and its child controls, deciding how we can best modify the default implementation and extend its capabilities so that it will validate to the W3C's XHTML 1.0 and 1.1 strict standards. Additionally we'll look at how the default instance uses client-side validation and how this too can be improved.

Who Should Read This Article?

This article is targeted at developers of the Microsoft .NET Framework who are using it to create XHTML Web sites containing ASP.NET Web controls.

Document Conventions

File names and file system directory names are written in an italic typeface e.g.

c:\mydirectory\myfile.txt

Code examples and function names are written in a typewriter mono-spaced typeface. Additionally program variables are italicised e.g.

```
000010  Procedure Code
000020  Main
000030  Move X to Y
```

Notes that require special attention are printed in a bold typeface enclosed in a surrounding box e.g.

This is a sample note

System Requirements

In order to test and run sample code included in this document, a text editor and C# compiler will be required along with an ASP.NET compatible Web server. All sample code has been developed and tested using Windows 2000 Professional running IIS 5.0 and version 1.1 of the .NET Framework. There should be no issues using later versions of Windows operating system. It is also possible to use the "Mono" project's compiled source-code running on most Linux distributions.

Assumptions

The source-code in this article is written in C# and ASP.NET. It is assumed that you have a basic understanding of both C# and ASP.NET as well as the essentials of the CLR (or equivalent runtime) and its associated development tools. The code listed is CLS-compliant so you can easily replace the C# code with an alternate CLI-compatible language. It is also assumed that the requirements for creating XHTML compliant Web pages are understood.

Limitations

This article makes no provisions for the XHTML-Basic standard as this is targeted at mobile devices and thus covered by Microsoft's ASP.NET mobile controls.

XHTML 2.0 is currently unsupported by most major browsers or has yet to make recommended status and is thus out of the scope of this article. The solutions discussed here can be applied to this standard if required.

The Problem

In their original out-of-the-box format Microsoft's ASP.NET `HtmlControls` and `WebControls` do not produce XHTML compliant code. Microsoft has thoughtfully (and thankfully) allowed developers to extend the controls to add user-defined custom functionality. It will be through this added functionality that we will be able to output valid code.

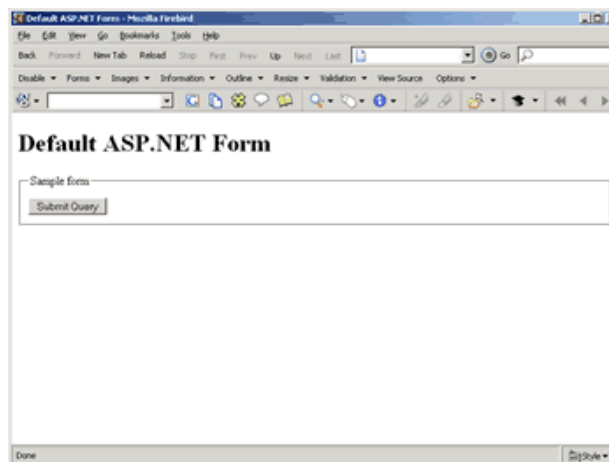
To start with, let us look at the code that the current default implementation produces. Create a new ASP.NET page in the root folder of your IIS development machine; call the page *currentform.aspx* and copy the code in **Listing 01** (you'll notice we're using the "XHTML 1.0 Strict" DTD in this example).

Listing 01

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-GB">
  <head>
    <meta http-equiv="Content-Type"
content="application/xhtml+xml; charset=utf-8"/>
    <title>Default ASP.NET Form</title>
  </head>
  <body>
    <h1>Default ASP.NET Form</h1>
    <form runat="server">
      <fieldset>
        <legend>Sample form</legend>
        <input type="submit"/>
      </fieldset>
    </form>
  </body>
</html>
```

Point your browser at *currentform.aspx*, you should see something similar to **Figure 1a**.

Figure 1a



If you view the source code you should see the following HTML (formatted here for brevity):

...

```
<form name="_ctl0" method="post" action="currentform.aspx"
id="_ctl0">
<input type="hidden" name="__VIEWSTATE" value="**random
characters**" />
    <fieldset>
        <legend>Sample form</legend>
        <input type="submit"/>
    </fieldset>
</form>
...
```

As you can see ASP.NET has added the name, method, action and id attributes as well as a hidden field called `__VIEWSTATE` containing the forms encrypted triplet view-state values. If we try and validate this document by uploading a copy of the source-code to the W3C's validating service available at - <http://validator.w3.org/> you'll see we get the following response:

“This page is not Valid XHTML 1.0 Strict!”

The validating service goes on to tell us we have two errors:

1. There is no attribute called name within a form tag.
2. The document type does not allow element input here because input is an inline tag when a block-level tag was expected.

The remainder of this document will focus on fixing these two errors.

Suggested Solution

Although there are potentially many ways to solve issues like the one detailed above, this document will concentrate on inheriting and extending the base ASP.NET controls to produce the results we desire.

Step 1: Creating a Custom Assembly

To start with we'll create two new classes inside a namespace that we can use to begin building our assembly based solution. The first class will extend `System.Web.UI.Page` and the second will extend `System.Web.UI.HtmlControls.HtmlForm`. Create a new C# file called *Liquid.Tutorial.cs* and save it somewhere outside of your root IIS directory (we'll compile this file into a library class a little later on) and copy the code in **Listing 02** into the newly created file.

Listing 02

```
using System;
using System.IO;
using System.Web;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;

namespace Liquid.Tutorial
{
    public class CustomPage : Page
    {
        protected Literal inputViewState;
        protected Literal outputText;
    }
}
```

```

        public CustomPage()
        {
        }
    }
    public class CustomForm : HtmlForm
    {
        public CustomForm()
        {
        }
    }
}

```

First this code imports all the namespaces that we'll be using, then it extends two default classes and accesses the `asp:Literal` controls in our ASP.NET page. Compile the newly created file into a library *.dll* file using the following command line syntax:

```
csc /t:library Liquid.Tutorial.cs
```

Copy the *.dll* file that is created into the */bin* directory of your Web server. Now we have a base assembly in place we can create a new ASP.NET page that utilises our new classes.

Step 2: Creating a Custom Page

Create a new ASP.NET page in the root folder of your IIS development machine; call the page *replacementform.aspx* and copy the code from **Listing 03** into it.

Listing 03

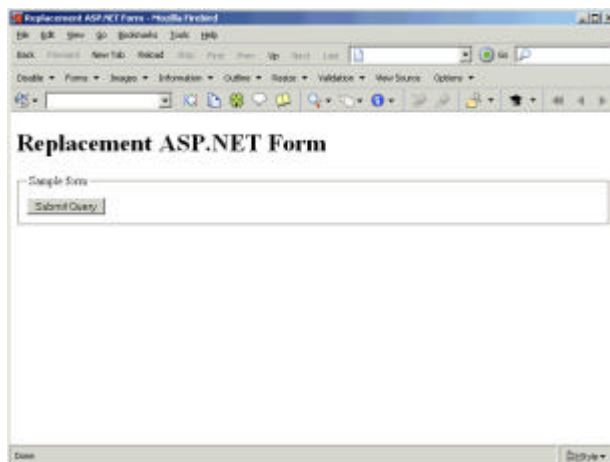
```

<%@ Page Inherits="Liquid.Tutorial.CustomPage" AutoEventWireup="True"
%>
<%@ Register TagPrefix="Liquid" Namespace="Liquid.Tutorial"
Assembly="Liquid.Tutorial" %>
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-GB">
  <head>
    <meta http-equiv="Content-Type"
content="application/xhtml+xml; charset=utf-8"/>
    <title>Replacement ASP.NET Form</title>
  </head>
  <body>
    <h1>Replacement ASP.NET Form</h1>
    <Liquid:CustomForm
      id="myId"
      method="post"
      runat="server">
      <fieldset>
        <legend>Sample form</legend>
        <input type="submit"/>
        <asp:Literal
          id="inputViewState"
          runat="server"/>
      </fieldset>
    </Liquid:CustomForm>
    <p><asp:Literal
      id="outputText"
      runat="server"/></p>
  </body>
</html>

```

You'll recognise most of this code from the *currentform.aspx* file we created earlier, notice the new `@Page` and `@Control` registration lines at the top of the code as well as the new `Liquid:CustomForm` tag that replaces the `form runat="server"` tag. Also notice the `asp:Literal` controls that act as placeholders for the soon to be created view-state field and response text. If you now point your browser to *replacementform.aspx* you should see something similar to **Figure 1b**.

Figure 1b



Step 3: Fixing the Name Attribute

Now that we have a working assembly let's take another look at the first error we encountered:

1. There is no attribute called name within a form tag.

ASP.NET writes the `form` tag, the correct attributes and the offending attribute to the page when the `HtmlForm.Render()` method is called. As we're clearly going to need to make some changes to this particular method it is worth taking a moment to see roughly what goes on when this is called:

- `HtmlForm.Render()`
 - Validate that page is not null
 - Write the smartnavigation scripts
 - Write the start of the form tag
- `HtmlForm.RenderAttributes()`
 - Write the attribute tags
- `HtmlForm.RenderChildren()`
- `Page.OnFormRender()`
 - Ensure no more than one form exists on a page
 - Write hidden fields
 - Write the view-state hidden field
- `HtmlContainerControl.RenderChildren()`
 - Render all code between form tags
- `Page.OnFormPostRender()`
 - Write out additional script blocks
- Write out the end form tag

As you can see the first method that we're likely to be interested in is `HtmlForm.RenderAttributes()` as that's the place where our problematic name attribute is written out. Luckily for us that method happens to be declared `virtual` so it's pretty straightforward to override it and replace it with our own code found in **Listing 04**.

Listing 04

```
protected override void RenderAttributes(HtmlTextWriter output)
{
    output.WriteAttribute("id", this.ID);
    output.WriteAttribute("method", this.Method);
    output.WriteAttribute("action", HttpContext.Current.Request.Path);
}
```

The code simply writes the attributes we are actually interested in and ignores the others. This could easily be extended should additional attributes be required (client-side form validation for example) but to keep it simple we'll stick with those three. Add this code to the `CustomForm` class in our *Liquid.Tutorial.cs* file, compile and copy the assembly file to the Web server's `/bin` directory then run the *replacementform.aspx* page. If we view the page source code we can see the problematic name attribute has disappeared.

Step 4: Fixing the View-State Field

Now that we've fixed our first problem let's take another look at the second error we encountered:

2. The document type does not allow element `input` here because `input` is an inline tag when a block-level tag was expected.

ASP.NET writes the hidden view-state field when the `HtmlForm.Render()` method is called. Looking back over the rendering process we can see that the view-state field is written when the `HtmlForm.RenderChildren()` call the `Page.OnFormRender()` method. The problem here is that the `Page.OnFormRender()` is declared `private` so we can't override as before. Luckily the `HtmlForm.RenderChildren()` is declared as a `virtual` method so we'll start by overriding that as with code from **Listing 05**.

Listing 05

```
protected override void RenderChildren(HtmlTextWriter output)
{
    foreach(Control c in base.Controls)
    {
        c.RenderControl(output);
    }
}
```

This overridden function simply gets the code between the `form` tags and writes it to the page. If you add this code to the `CustomForm` class in our *Liquid.Tutorial.cs* file, compile and copy the *.dll* to the Web server's `/bin` directory then run the *replacementform.aspx* you'll find that it outputs valid XHTML. Only now the form doesn't actually post back properly, this is because there is an issue with all current versions of the .NET runtime that **require** the hidden `input` form field to be present, even if the value is blank.

Before we continue there is one additional point worth bringing up and that is the issue of client-side ECMAScript (JavaScript). As we are developing XHTML code for XHTML capable browsers I'm going to assume that we're also writing correct ECMAScript 1.2 code, in that we're using the `document.getElementById("myId")` method to access document objects. This is important because the minute we remove the "name" attribute from the page `document.forms.formName` will fail to work. Additionally given the problems involved with getting inline script to work correctly with XHTML valid documents I'm also going to assume **all** ECMAScript will be contained in a separate document. Therefore for the sake of this article I'm going to ignore the `Page.OnFormPostRender()` method that writes Microsoft's proprietary JScript to the page. Should you be inclined you can easily apply the same principles shown here to access these properties. I should also add that any hidden fields written to the page using the `RegisterHiddenFields()` method will need to be manually added from now on.

OK, back to the solution. Now we're outputting valid XHTML we need to put the view-state hidden field back onto the page. To get and set the hidden form field we need to override the `Page.LoadPageStateFromPersistenceMedium()` and `Page.SavePageStateToPersistenceMedium()` methods to get and set the hidden form field value.

Earlier you'll have noticed that we placed an `asp:Literal` control within the `fieldset` tag of our ASP.NET page. Normally the `HtmlHidden` field would be used to write hidden form fields but as this particular hidden field has to have a name equal to `__VIEWSTATE` and as ASP.NET derives the field's name from its `id` value we'd have to have an `id` called `__VIEWSTATE`. The problem here is that variable names beginning with a double underscore are not CLS compliant. In order to get around that we'll use the `asp:Literal` control to write out the populated input tag when needed. This way we can call the literal `id` what we like. Add the lines of code in **Listing 06** to the derived `CustomPage` class in our *Liquid.Tutorial.cs* file.

Listing 06

```
protected override Object LoadPageStateFromPersistenceMedium()
{
    LosFormatter format = new LosFormatter();
    String viewState =
    HttpContext.Current.Request.Form["__VIEWSTATE"].ToString();
    return format.Deserialize(viewState);
}
protected override void SavePageStateToPersistenceMedium(Object
viewState)
{
    LosFormatter format = new LosFormatter();
    StringWriter writer = new StringWriter();
    format.Serialize(writer, viewState);
    this.inputViewState.Text = "<input type=\"hidden\"
name=\"__VIEWSTATE\" value=\"" + writer.ToString() + "\"/>";
}
```

`LosFormatter` is a somewhat undocumented class that is used to write the view-state object to a Base64 string and back again. It's worth pointing out that in .NET runtime version 1.1 the `LosFormatter` can be created by passing in an MD5 checksum; this is recommended for additional security but beyond the scope of this article.

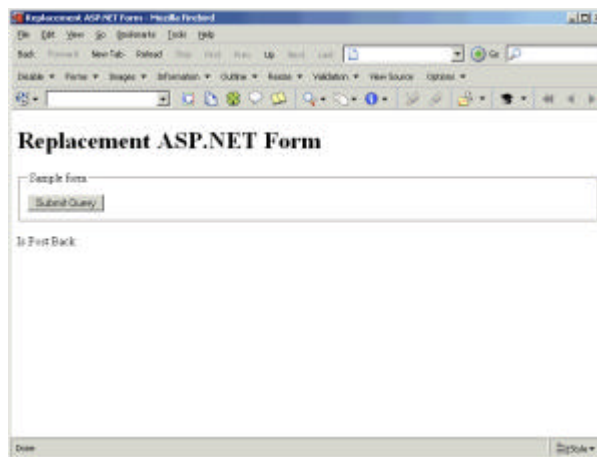
Finally, in order to prove that the form is in fact submitting and firing events correctly we add the final piece of code from **Listing 07**.

Listing 07

```
protected void Page_Load(Object sender, EventArgs e)
{
    if(Page.IsPostBack)
    {
        outputText.Text = "Is post back";
    }
}
```

The code writes an output string to the final `asp:Literal` control when the form is submitted correctly. When the form is viewed and submitted in a browser you should see something similar to **Figure 1c**.

Figure 1c



The final test is to view the page source code and submit it to the W3C's validating service for final validation. Hopefully you should see the following message.

“This Page Is Valid XHTML 1.0 Strict!”

Step 5: Adding Some Error Checking

Assuming you've been paying attention so far you'll have probably realised the one function we overwrote but didn't replace when we overrode the `HtmlForm.RenderChildren()` method, was the check to see if a form already exists on the page. If you're a solo developer working with simple pages this may not be any issue. You can probably remember not to add more than one form to each page. If however the project is larger or involves a team of developers you may require this helpful functionality. The problem is the functionality to perform such a check is locked away in private functions so we'll have to create our own solution. To start with, add the private field and internal accessor code found in **Listing 08** to the `CustomPage` class.

Listing 08

```
private Boolean _fOnFormRenderCalled = false;
internal void OnFormRender()
{
```

```

    if(this._fOnFormRenderCalled)
    {
        throw new HttpException("A page can have only one visible
        server-side CustomForm tag");
    }
    else
    {
        this._fOnFormRenderCalled = true;
    }
}

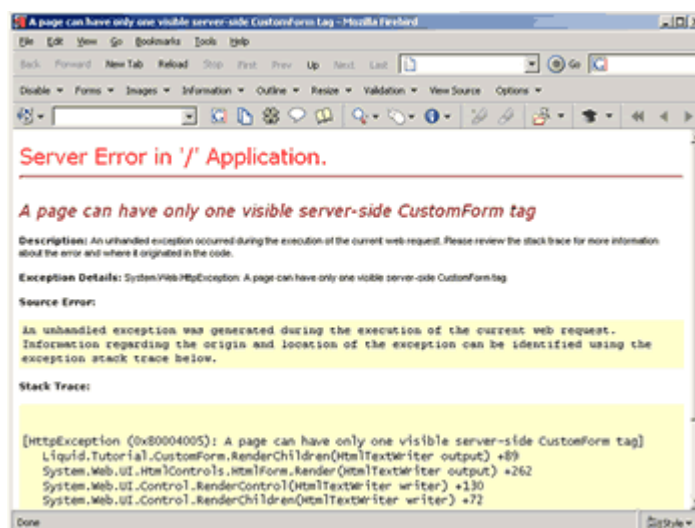
```

The Boolean field will allow us to set and get a flag stating if a form already exists on a page. Add the following code to the `RenderChildren()` method of the `CustomForm` class to set and get the property via the accessor property:

```
((CustomPage)this.Page).OnFormRender();
```

This line of code simply casts the base page class as the derived page class so that it can read and write the flag field. Compile the library assembly, copy it to the `/bin` directory and run the code. If you try and add a second `CustomForm` to the `replacementform.aspx` page, you should see the error message shown in **Figure 1d**.

Figure 1d



Remove the second `CustomForm` tag and you're done. Aside from the aforementioned JScript code inclusions we've managed to replace all of the functionality found in the `HtmlForm` control into a handy XHTML compliant version.

Summary

The goal of this article was to provide an insight into the way Microsoft have structured their `HtmlControls` and how you can go about making them more XHTML friendly. We first looked at a basic ASP.NET `HtmlForm` control and saw how it failed to validate to W3C standards. We then looked into one possible solution that overrides the rendering of the `HtmlForm` control and `Page` view-state handling. This was then validated using the W3C Validator.

Feel free to download the source code using the link provided, should you have any questions or comments please send them via support@liquid-internet.co.uk.

Downloads

Sample source code to accompany this article can be downloaded from <http://www.liquid-internet.co.uk/content/static/media/downloads/series1article1.zip>

Please note that some of the source code includes additional code improvements not covered in this article, where necessary these are covered by inline code comments and additional help files.

Appendix A: Resources

W3C's technical report on the XHTML 1.0 recommendation -

<http://www.w3.org/TR/xhtml1/>

W3C's technical report on the XHTML 1.1 recommendation -

<http://www.w3.org/TR/xhtml11/>

W3C's technical report on the XHTML 2.0 working draft - <http://www.w3.org/TR/xhtml2/>

Microsoft ASP.NET developer information - <http://msdn.microsoft.com/net/aspnet/>

Microsoft C# developer information - <http://msdn.microsoft.com/vcsharp/>

Microsoft .NET developer information - <http://msdn.microsoft.com/netframework/>

The Mono Project home page - <http://www.go-mono.org/>

The ECMA C# specification – <http://www.ecma-international.org/publications/Standards/ecma-334.htm>

The ECMA CLI specification – <http://www.ecma-international.org/publications/Standards/ecma-335.htm>

Appendix B: Glossary

.NET Framework – A group of technologies used to write CLR-based applications.

Assembly – A collection of modules that are deployed together to create a single file.

Common Language Infrastructure (CLI) – A set of ECMA ratified standards that can be used to implement a custom runtime.

Common Language Runtime (CLR) – Implementation of the CLI for the Microsoft Windows operating system.

Common Language Specification (CLS) – All CLI compatible languages must support this specification.

Document Type Definition (DTD) – Defines the legal building blocks of a XML document.

ECMAScript – International Web standard for scripting languages in a host environment.

Extensible Hyper Text Mark-up Language (XHTML) – A W3C standard mark-up language.

Hyper Text Mark-up Language (HTML) – A W3C standard mark-up language.

Internet Information Server (IIS) – Server developed by Microsoft that parses ASP.NET Web pages.

Mono project – An open source implementation of the .NET framework.

W3C – The World Wide Web Consortium develops interoperable technologies and standards for the Internet.

Index

.NET Framework, 5, 17
@Control, 10
@Page, 10
action, 8, 11
ASP.NET, 5, 6, 7, 8, 9, 10, 11, 12, 14, 16
assembly, 8, 9, 10, 11, 14
C#, 5, 8, 16
CLI, 5, 16, 17
CLR, 5, 17
CLS, 5, 12, 17
DTD, 7, 9
ECMAScript, 12, 17
form, 7, 8, 9, 10, 11, 12, 13, 14
HTML, 7, 17
HtmlControls, 7, 8, 14
HtmlForm, 1, 5, 8, 9, 10, 11, 13, 14
IIS, 5, 7, 8, 9
input, 7, 8, 9, 11, 12
Linux, 5
method, 8, 9, 10, 11, 12, 13, 14
Mono, 5, 16
View-State, 11
VIEWSTATE, 8, 12
W3C, 5, 7, 8, 9, 13, 14, 16, 17
WebControls, 7, 8
XHTML, 1, 5, 6, 7, 8, 9, 11, 12, 13, 14, 16, 17