Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview

Welcome to the ASP.NET QuickStart Tutorial

The ASP.NET QuickStart is a series of ASP.NET samples and supporting commentary designed to quickly acquaint developers with the syntax, architecture, and power of the ASP.NET Web programming framework. The QuickStart samples are designed to be short, easy-to-understand illustrations of ASP.NET features. By the time you have completed the QuickStart tutorial, you will be familiar with:

- **ASP.NET Syntax.** While some of the ASP.NET syntax elements will be familiar to veteran ASP developers, several are unique to the new framework. The QuickStart samples cover each element in detail.
- **ASP.NET Architecture and Features.** The QuickStart introduces the features of ASP.NET that enable developers to build interactive, world-class applications with much less time and effort than ever before.
- **Best Practices.** The QuickStart samples demonstrate the best ways to exercise the power of ASP.NET while avoiding potential pitfalls along the way.

What Level of Expertise Is Assumed in the QuickStart?

If you have never developed Web pages before, the QuickStart is not for you. You should be fluent in HTML and general Web development terminology. You do not need previous ASP experience, but you should be familiar with the concepts behind interactive Web pages, including forms, scripts, and data access.

Working with the QuickStart Samples

The QuickStart samples are best experienced in the order in which they are presented. Each sample builds on concepts discussed in the preceding sample. The sequence begins with a simple form submittal and builds up to integrated application scenarios.

Copyright 2001 Microsoft Corporation. All rights reserved.

Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications

A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview

What is ASP.NET?

ASP.NET is a programming framework built on the common language runtime that can be used on a server to build powerful Web applications. ASP.NET offers several important advantages over previous Web development models:

- **Enhanced Performance.** ASP.NET is compiled common language runtime code running on the server. Unlike its interpreted predecessors, ASP.NET can take advantage of early binding, just-in-time compilation, native optimization, and caching services right out of the box. This amounts to dramatically better performance before you ever write a line of code.

- **World-Class Tool Support.** The ASP.NET framework is complemented by a rich toolbox and designer in the Visual Studio integrated development environment. WYSIWYG editing, drag-and-drop server controls, and automatic deployment are just a few of the features this powerful tool provides.

- **Power and Flexibility.** Because ASP.NET is based on the common language runtime, the power and flexibility of that entire platform is available to Web application developers. The .NET Framework class library, Messaging, and Data Access solutions are all seamlessly accessible from the Web. ASP.NET is also language-independent, so you can choose the language that best applies to your application or partition your application across many languages. Further, common language runtime interoperability guarantees that your existing investment in COM-based development is preserved when migrating to ASP.NET.

- **Simplicity.** ASP.NET makes it easy to perform common tasks, from simple form submission and client authentication to deployment and site configuration. For example, the ASP.NET page framework allows you to build user interfaces that cleanly separate application logic from presentation code and to handle events in a simple, Visual Basic - like forms processing model. Additionally, the common language runtime simplifies development, with managed code services such as automatic reference counting and garbage collection.

Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications

- **Manageability.** ASP.NET employs a text-based, hierarchical configuration system, which simplifies applying settings to your server environment and Web applications. Because configuration information is stored as plain text, new settings may be applied without the aid of local administration tools. This "zero local administration" philosophy extends to deploying ASP.NET Framework applications as well. An ASP.NET Framework application is deployed to a server simply by copying the necessary files to the server. No server restart is required, even to deploy or replace running compiled code.

- **Scalability and Availability.** ASP.NET has been designed with scalability in mind, with features specifically tailored to improve performance in clustered and multiprocessor environments. Further, processes are closely monitored and managed by the ASP.NET runtime, so that if one misbehaves (leaks, deadlocks), a new process can be created in its place, which helps keep your application constantly available to handle requests.

- **Customizability and Extensibility.** ASP.NET delivers a well-factored architecture that allows developers to "plug-in" their code at the appropriate level. In fact, it is possible to extend or replace any subcomponent of the ASP.NET runtime with your own custom-written component. Implementing custom authentication or state services has never been easier.

- **Security.** With built in Windows authentication and per-application configuration, you can be assured that your applications are secure.

The remainder of the QuickStart presents practical examples of these concepts.

Copyright 2001 Microsoft Corporation. All rights reserved.

Languages

Declare a variable
Issue a statement
Comment my code
Declare a simple property
Use an indexed property
Declare an indexed property
Declare an enumeration
Enumerate a collection or array
Declare and use a method
Define custom attributes
Declare and use an array
Statically initialize a variable
Write an if statement
Write a case statement
Write a for loop
Write a while loop
Handle exceptions
Concatenate a string
Declare an event
Declare an event handler
Add an event handler
Cast a variable to a type
Convert a variable to type
Declare a class or interface
Inherit from a base class
Implement an interface
Class with a main method
Write a standard module

Back to Index

Get URL for this page

Language Support

The Microsoft .NET Platform currently offers built-in support for three languages: C#, Visual Basic, and JScript.

The exercises and code samples in this tutorial demonstrate how to use C#, Visual Basic, and JScript to build .NET applications. For information regarding the syntax of the other languages, refer to the complete documentation for the .NET Framework SDK.

The following table is provided to help you understand the code samples in this tutorial as well as the differences between the three languages:

Variable Declarations

```
Dim x As Integer
Dim s As String
Dim s1, s2 As String
Dim o 'Implicitly Object
Dim obj As New Object()
Public name As String
```

VB

Statements

```
Response.Write("foo")
```

VB

Comments

```
' This is a comment

' This
' is
' a
' multiline
' comment
```

VB

Accessing Indexed Properties

```
Dim s, value As String
s = Request.QueryString("Name")
value = Request.Cookies("Key").Value

'Note that default non-indexed properties
'must be explicitly named in VB
```

VB

Declaring Indexed Properties

```
' Default Indexed Property
Public Default ReadOnly Property DefaultProperty(Name As String) As String
    Get
        Return CStr(lookuptable(name))
    End Get
End Property
```

VB

Declaring Simple Properties

```
Public Property Name As String

  Get
    ...
    Return ...
  End Get

  Set
    ... = Value
  End Set

End Property
```

VB

Declare and Use an Enumeration

```
' Declare the Enumeration
Public Enum MessageSize

    Small = 0
    Medium = 1
    Large = 2
End Enum

' Create a Field or Property
Public MsgSize As MessageSize

' Assign to the property using the Enumeration values
MsgSize = small
```

VB

Enumerating a Collection

```
Dim S As String
For Each S In Coll
 ...
Next
```

VB

Declare and Use Methods

```
' Declare a void return function
Sub VoidFunction()
 ...
End Sub

' Declare a function that returns a value
Function StringFunction() As String
 ...
    Return CStr(val)
End Function

' Declare a function that takes and returns values
Function ParmFunction(a As String, b As String) As String
 ...
    Return CStr(A & B)
End Function

' Use the Functions
VoidFunction()
Dim s1 As String = StringFunction()
Dim s2 As String = ParmFunction("Hello", "World!")
```

VB

Custom Attributes

```
' Stand-alone attribute
<STAThread>

' Attribute with parameters
<DllImport("ADVAPI32.DLL")>

' Attribute with named parameters
<DllImport("KERNEL32.DLL", CharSet:=CharSet.Auto)>
```

VB

Arrays

```
    Dim a(2) As String
    a(0) = "1"
    a(1) = "2"
    a(2) = "3"

    Dim a(2,2) As String
    a(0,0) = "1"
    a(1,0) = "2"
    a(2,0) = "3"
```

VB

Initialization

```
Dim s As String = "Hello World"
Dim i As Integer = 1
Dim a() As Double = { 3.00, 4.00, 5.00 }
```

VB

If Statements

```
If Not (Request.QueryString = Nothing)
  ...
End If
```

VB

Case Statements

```
Select Case FirstName
  Case "John"
    ...
  Case "Paul"
    ...
  Case "Ringo"
    ...
  Case Else
    ...
End Select
```

VB

For Loops

```
  Dim I As Integer
  For I = 0 To 2
    a(I) = "test"
  Next
```

VB

While Loops

```
Dim I As Integer
I = 0
Do While I < 3
  Console.WriteLine(I.ToString())
  I += 1
Loop
```

VB

Exception Handling

```
Try
    ' Code that throws exceptions
Catch E As OverflowException
    ' Catch a specific exception
Catch E As Exception
    ' Catch the generic exceptions
Finally
    ' Execute some cleanup code
End Try
```

VB

String Concatenation

```
' Using Strings
Dim s1, s2 As String
s2 = "hello"
s2 &= " world"
s1 = s2 & " !!!"

' Using StringBuilder class for performance
Dim s3 As New StringBuilder()
s3.Append("hello")
s3.Append(" world")
s3.Append(" !!!")
```

VB

Event Handler Delegates

```
Sub MyButton_Click(Sender As Object,
                   E As EventArgs)
...
End Sub
```

VB

Declare Events

```
' Create a public event
Public Event MyEvent(Sender as Object, E as EventArgs)

' Create a method for firing the event
Protected Sub OnMyEvent(E As EventArgs)
    RaiseEvent MyEvent(Me, E)
End Sub
```

VB

Add or Remove Event Handlers to Events

```
AddHandler Control.Change, AddressOf Me.ChangeEventHandler
RemoveHandler Control.Change, AddressOf Me.ChangeEventHandler
```

VB

Casting

```
Dim obj As MyObject
Dim iObj As IMyObject
obj = Session("Some Value")
iObj = CType(obj, IMyObject)
```

VB

Conversion

```
Dim i As Integer
Dim s As String
Dim d As Double
```

```
i = 3
s = i.ToString()
d = CDbl(s)

' See also CDbl(...), CStr(...), ...
```

VB

Class Definition with Inheritance

```
Imports System

Namespace MySpace

  Public Class Foo : Inherits Bar

    Dim x As Integer

    Public Sub New()
      MyBase.New()
      x = 4
    End Sub

    Public Sub Add(x As Integer)
      Me.x = Me.x + x
    End Sub

    Overrides Public Function GetNum() As Integer
       Return x
    End Function

  End Class

End Namespace

' vbc /out:libraryvb.dll /t:library
' library.vb
```

VB

Implementing an Interface

```
Public Class MyClass : Implements IEnumerable
 ...

    Function IEnumerable_GetEnumerator() As IEnumerator Implements
IEnumerable.GetEnumerator
         ...
    End Function
End Class
```

VB

Class Definition with a Main Method

```
Imports System

Public Class ConsoleVB

  Public Sub New()
    MyBase.New()
    Console.WriteLine("Object Created")
  End Sub

  Public Shared Sub Main()
```

```
      Console.WriteLine("Hello World")
      Dim cvb As New ConsoleVB
    End Sub

  End Class

  ' vbc /out:consolevb.exe /t:exe console.vb
```

VB

Standard Module

```
  Imports System

  Public Module ConsoleVB

    Public Sub Main()
      Console.WriteLine("Hello World")
    End Sub

  End Module

  ' vbc /out:consolevb.exe /t:exe console.vb
```

VB

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview

Introducing Web Forms

- What is ASP.NET Web Forms?
- Writing Your First Web Forms Page
- Using ASP <%%> Render Blocks
- Introduction to Server Controls
- Handling Server Control Events
- Using Custom Server Controls
- Lists, Data, and Databinding
- Form Validation Controls
- Code-Behind Web Forms
- Section Summary

What is ASP.NET Web Forms?

The ASP.NET Web Forms page framework is a scalable common language runtime programming model that can be used on the server to dynamically generate Web pages.

Intended as a logical evolution of ASP (ASP.NET provides syntax compatibility with existing pages), the ASP.NET Web Forms framework has been specifically designed to address a number of key deficiencies in the previous model. In particular, it provides:

- The ability to create and use reusable UI controls that can encapsulate common functionality and thus reduce the amount of code that a page developer has to write.
- The ability for developers to cleanly structure their page logic in an orderly fashion (not "spaghetti code").
- The ability for development tools to provide strong WYSIWYG design support for pages (existing ASP code is opaque to tools).

This section of the QuickStart provides a high-level code walkthrough of some key ASP.NET Web Forms features. Subsequent sections of the QuickStart drill down into more specific details.

Writing Your First Web Forms Page

ASP.NET Web Forms pages are text files with an .aspx file

Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

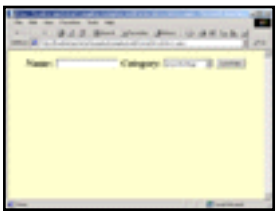
Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications

name extension. They can be deployed throughout an IIS virtual root directory tree. When a browser client requests .aspx resources, the ASP.NET runtime parses and compiles the target file into a .NET Framework class. This class can then be used to dynamically process incoming requests. (Note that the .aspx file is compiled only the first time it is accessed; the compiled type instance is then reused across multiple requests).

An ASP.NET page can be created simply by taking an existing HTML file and changing its file name extension to .aspx (no modification of code is required). For example, the following sample demonstrates a simple HTML page that collects a user's name and category preference and then performs a form postback to the originating page when a button is clicked:



VB Intro1.aspx

[Run Sample] | [View Source]

Important: Note that nothing happens yet when you click the Lookup button. This is because the .aspx file contains only static HTML (no dynamic content). Thus, the same HTML is sent back to the client on each trip to the page, which results in a loss of the contents of the form fields (the text box and drop-down list) between requests.

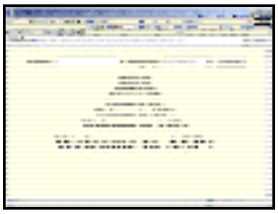
Using ASP <% %> Render Blocks

ASP.NET provides syntax compatibility with existing ASP pages. This includes support for <% %> code render blocks that can be intermixed with HTML content within an .aspx file. These code blocks execute in a top-down manner at page render time.

The below example demonstrates how <% %> render blocks can be used to loop over an HTML block (increasing the font size each time):

A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page



VB Intro2.aspx

[Run Sample] | [View Source]

Important: Unlike with ASP, the code used within the above <% %> blocks is actually compiled--not interpreted using a script engine. This results in improved runtime execution performance.

ASP.NET page developers can utilize <% %> code blocks to dynamically modify HTML output much as they can today with ASP. For example, the following sample demonstrates how <% %> code blocks can be used to interpret results posted back from a client.

VB Intro3.aspx

[Run Sample] | [View Source]

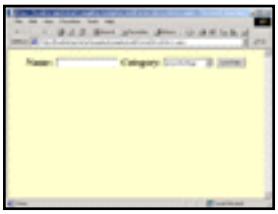
Important: While <% %> code blocks provide a powerful way to custom manipulate the text output returned from an ASP.NET page, they do not provide a clean HTML programming model. As the sample above illustrates, developers using only <% %> code blocks must custom manage page state between round trips and custom interpret posted values.

Introduction to ASP.NET Server Controls

In addition to (or instead of) using <% %> code blocks to program dynamic content, ASP.NET page developers can use ASP.NET server controls to program Web pages. Server controls are declared within an .aspx file using custom tags or intrinsic HTML tags that contain a **runat="server"** attribute value. Intrinsic HTML tags are handled by one of the controls in the **System.Web.UI.HtmlControls** namespace. Any tag that doesn't explicitly map to one of the controls is assigned the type of

System.Web.UI.HtmlControls.HtmlGenericControl.

The following sample uses four server controls: **<form runat=server>**, **<asp:textbox runat=server>**, **<asp:dropdownlist runat=server>**, and **<asp:button runat=server>**. At run time these server controls automatically generate HTML content.



VB Intro4.aspx

[Run Sample] | [View Source]

Important: Note that these server controls automatically maintain any client-entered values between round trips to the server. This control state is not stored on the server (it is instead stored within an **<input type="hidden">** form field that is round-tripped between requests). Note also that no client-side script is required.

In addition to supporting standard HTML input controls, ASP.NET enables developers to utilize richer custom controls on their pages. For example, the following sample demonstrates how the **<asp:adrotator>** control can be used to dynamically display rotating ads on a page.

VB Intro5.aspx

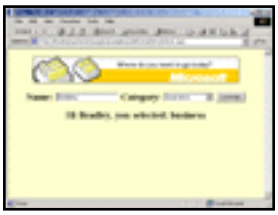
[Run Sample] | [View Source]

Important: A detailed listing of all built-in server controls can be found in the Web Forms Control Reference section of this QuickStart.

Handling Server Control Events

Each ASP.NET server control is capable of exposing an object model containing properties, methods, and events. ASP.NET developers can use this object model to cleanly modify and interact with the page.

The following example demonstrates how an ASP.NET page developer can handle the **OnClick** event from the **<asp:button runat=server>** control to manipulate the **Text** property of the **<asp:label runat=server>** control.



VB Intro6.aspx

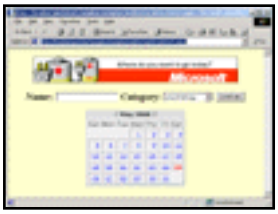
[Run Sample] | [View Source]

This simple sample is functionally equivalent to the "Intro3" sample demonstrated earlier in this section. Note, however, how much cleaner and easier the code is in this new server-control-based version.

Using Custom Server Controls

ASP.NET ships with 45 built-in server controls that can be used out of the box (for details, see Web Forms Controls Reference). In addition to using the built-in ASP.NET controls, developers also can use controls developed by third-party vendors.

The following sample shows a simple calendar control. The **Calendar** control is declared within the page using an **<acme:calendar runat=server>** tag. Note that the **<% Register %>** directive at the top of the page is responsible for registering the "Acme" XML tag prefix with the "Acme" code namespace of the control implementation. The ASP.NET page parser will then utilize this namespace to load the **Calendar** control class instance at run time.



VB Intro7.aspx

[Run Sample] | [View Source]

The **Calendar** control in this sample has been designed to perform "uplevel-like" processing on Internet Explorer 5.5 and "downlevel" processing on all other browsers. This browser sniffing is nowhere near as complex as that provided by the ASP.NET built-in server controls. For Internet Explorer 5.5 browsers it generates DHTML output. This DHTML output does not require round trips back to the server when doing day selections and month navigations. For all other browsers the control generates standard HTML 3.2. This HTML 3.2 does require round trips back to the server to handle client-side user interactions.

Important: The code that a page developer writes is identical regardless of whether an "uplevel" or "downlevel" browser is used to access the page. The **Calendar** control itself encapsulates all of the logic required to handle the two scenarios.

Lists, Data, and Data Binding

ASP.NET ships with a built-in set of data grid and list controls. These can be used to provide custom UI driven from queries against a database or other data source. For example, the following sample demonstrates how a **<asp:datagrid runat=server>** control can be used to databind book information collected using a SQL database query.

VB Intro8.aspx

[Run Sample] | [View Source]

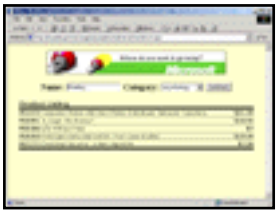
The **<asp:datagrid runat=server> DataGrid** control provides an easy way to quickly display data results using a traditional grid-control UI. Alternatively, ASP.NET developers can use the **<asp:DataList runat=server> DataList** control and a custom **ItemTemplate** template to customize data information, as in the following sample.

VB Intro9.aspx

[Run Sample] | [View Source]

Note that the **<asp:datalist runat=server>** control enables end users to exactly control the structure and layout of each item within the list (using the **ItemTemplate** template property). The control also automatically handles the two-column wrapping of content (users can control the number of columns using the **RepeatColumns** property on the data list).

The following sample provides an alternate view of the **<asp:datalist runat=server>** control.



VB Intro10.aspx

[Run Sample] | [View Source]

Note that the control, data model, and page user in this example are exactly the same as those in the previous sample. The only difference is that, here, alternative templates are declaratively supplied to the code.

Form Validation Controls

The ASP.NET Web Forms page framework provides a set of validation server controls that provide an easy-to-use but

powerful way to check input forms for errors, and, if necessary, display messages to the user.

Validation controls are added to an ASP.NET page like other server controls. There are controls for specific types of validation, such as range checking or pattern matching, plus a **RequiredFieldValidator** that ensures that a user does not skip an entry field.

The following example demonstrates how to use two **<asp:requiredfieldvalidator runat=server>** controls on a page to validate the contents of the **TextBox** and **DropDownList** controls.

VB Intro11.aspx

[Run Sample] | [View Source]

Note that the validation controls have both uplevel and downlevel client support. Uplevel browsers perform validation on the client (using JavaScript and DHTML) and on the server. Downlevel browsers perform the validation only on the server. The programming model for the two scenarios is identical.

Note that ASP.NET page developers can optionally check the **Page.IsValid** property at run time to determine whether all validation server controls on a page are currently valid. This provides a simple way to determine whether or not to proceed with business logic. For example, the following sample performs a **Page.IsValid** check before executing a database lookup on the specified category.

VB Intro12.aspx

[Run Sample] | [View Source]

Code-Behind Web Forms

ASP.NET supports two methods of authoring dynamic pages. The first is the method shown in the preceding samples, where the page code is physically declared within the originating .aspx file. An alternative approach--known as the code-behind method--enables the page code to be more cleanly separated from the HTML content into an entirely separate file.

The following sample demonstrates the use of the code-behind method of writing ASP.NET page code.

VB Intro13.aspx

[Run Sample] | [View Source]

Section Summary

1. ASP.NET Web Forms provide an easy and powerful way to build dynamic Web UI.
2. ASP.NET Web Forms pages can target any browser client (there are no script library or cookie requirements).
3. ASP.NET Web Forms pages provide syntax compatibility with existing ASP pages.
4. ASP.NET server controls provide an easy way to encapsulate common functionality.
5. ASP.NET ships with 45 built-in server controls. Developers can also use controls built by third

parties.

6. ASP.NET server controls can automatically project both uplevel and downlevel HTML.
7. ASP.NET templates provide an easy way to customize the look and feel of list server controls.
8. ASP.NET validation controls provide an easy way to do declarative client or server data validation.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview

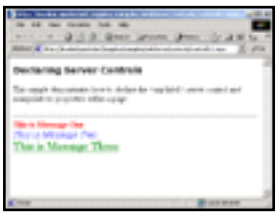
Working with Server Controls

- Declaring Server Controls
- Manipulating Server Controls
- Handling Control Action Events
- Handling Multiple Control Action Events
- Performing Page Navigation (Scenario 1)
- Performing Page Navigation (Scenario 2)

This section of the QuickStart illustrates some common core concepts and common actions performed by end users when using ASP.NET server controls within a page.

Declaring Server Controls

ASP.NET server controls are identified within a page using declarative tags that contain a **runat="server"** attribute. The following example declares three **<asp:label runat="server">** server controls and customizes the text and style properties of each one individually.



VB Controls1.aspx

[Run Sample] | [View Source]

Manipulating Server Controls

You can programmatically identify an individual ASP.NET server control within a page by providing it with an **id** attribute. You can use this **id** reference to programmatically manipulate the server control's object model at run time. For example, the following sample demonstrates how a page developer could programmatically set an **<asp:label runat="server">** control's **Text** property within the **Page_Load** event.

Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

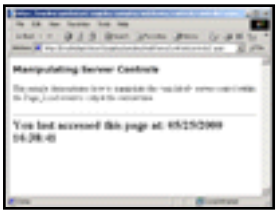
Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications

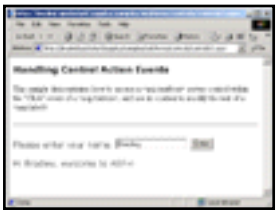


VB Controls2.aspx

[Run Sample] | [View Source]

Handling Control Action Events

ASP.NET server controls can optionally expose and raise server events, which can be handled by page developers. A page developer may accomplish this by declaratively wiring an event to a control (where the attribute name of an event wireup indicates the event name and the attribute value indicates the name of a method to call). For example, the following code example demonstrates how to wire an **OnClick** event to a button control.



VB Controls3.aspx

[Run Sample] | [View Source]

Handling Multiple Control Action Events

Event handlers provide a clean way for page developers to structure logic within an ASP.NET page. For example, the following sample demonstrates how to wire and handle four button events on a single page.

A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

VB Controls4.aspx

[Run Sample] | [View Source]

Performing Page Navigation (Scenario 1)

Page navigation among multiple pages is a common scenario in virtually all Web applications. The following sample demonstrates how to use the **<asp:hyperlink runat=server>** control to navigate to another page (passing custom query string parameters along the way). The sample then demonstrates how to easily get access to these query string parameters from the target page.

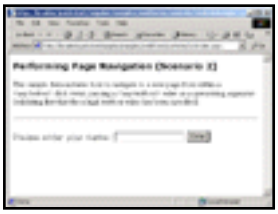
VB Controls5.aspx

[Run Sample] | [View Source]

Performing Page Navigation (Scenario 2)

Not all page navigation scenarios are initiated through hyperlinks on the client. Client-side page redirects or navigations can also be initiated from the server by an ASP.NET page developer by calling the **Response.Redirect(url)** method. This is typically done when server-side validation is required on some client input before the navigation actually takes place.

The following sample demonstrates how to use the **Response.Redirect** method to pass parameters to another target page. It also demonstrates how to easily get access to these parameters from the target page.



VB Controls6.aspx

[Run Sample] | [View Source]

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding

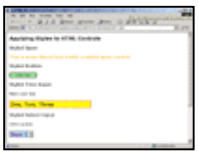
Applying Styles to Controls

- Applying Styles to HTMLControls
- Applying Styles to Web Controls
- Section Summary

The Web is a flexible environment for user interfaces, with extreme variations in the look and feel of different Web sites. The widespread adoption of cascading style sheets (CSS) is largely responsible for the rich designs encountered on the Web. All of ASP.NET's HTML server controls and Web server controls have been designed to provide first-class support for CSS styles. This section discusses how to use styles in conjunction with server controls and demonstrates the very fine control over the look and feel of your Web Forms that they provide.

Applying Styles to HTML Controls

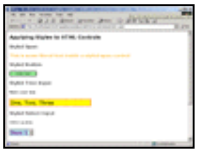
Standard HTML tags support CSS through a style attribute, which can be set to a semicolon-delimited list of attribute/value pairs. For more information about the CSS attributes supported by the Internet Explorer browser, see MSDN Web Workshop's CSS Attributes Reference page. All of the ASP.NET HTML server controls can accept styles in exactly the same manner as standard HTML tags. The following example illustrates a number of styles applied to various HTML server controls. If you view the source code on the page returned to the client, you will see that these styles are passed along to the browser in the control's rendering.



VB Style1.aspx

[Run Sample] | [View Source]

CSS also defines a class attribute, which can be set to a CSS style definition contained in a <style>...</style> section of the document. The class attribute makes it easy to define styles once and apply them to several tags without having to redefine the style itself. Styles on HTML server controls also can be set in this manner, as demonstrated in the following sample.



VB Style2.aspx

[Run Sample] | [View Source]

When an ASP.NET page is parsed, the style information is populated into a **Style** property (of type **CssStyleCollection**) on the **System.Web.UI.HtmlControls.HtmlControl** class. This property is essentially a dictionary that exposes the control's styles as a string-indexed collection of values for each style-attribute key. For example, you could use the following code to set and subsequently retrieve the **width** style attribute on an **HtmlInputText** server control.

```
<script language="VB" runat="server" >

    Sub Page_Load(Sender As Object, E As EventArgs)
        MyText.Style("width") = "90px"
        Response.Write(MyText.Style("width"))
    End Sub

</script>

<input type="text" id="MyText" runat="server"/>
```

VB

Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

This next sample shows how you can programmatically manipulate the style for an HTML server control using this **Style** collection property.

VB Style3.aspx

[Run Sample] | [View Source]

Applying Styles to Web Server Controls

Web server controls provide an additional level of support for styles by adding several strongly typed properties for commonly used style settings, such as background and foreground color, font name and size, width, height, and so on. These style properties represent a subset of the style behaviors available in HTML and are represented as "flat" properties exposed directly on the **System.Web.UI.WebControls.WebControl** base class. The advantage of using these properties is that they provide compile-time type checking and statement completion in development tools such as Microsoft Visual Studio .NET.

The following sample shows a **WebCalendar** control with several styles applied to it (a calendar without styles applied is included for contrast). Note that when setting a property that is a class type, such as **Font**, you need to use the subproperty syntax PropertyName-SubPropertyName .

VB Style4.aspx

[Run Sample] | [View Source]

The **System.Web.UI.WebControls** namespace includes a **Style** base class that encapsulates common style attributes (additional style classes, such as **TableStyle** and **TableItemStyle**, inherit from this common base class). Many Web server controls expose properties of this type for specifying the style of individual rendering elements of the control. For example, the **WebCalendar** exposes many such style properties: **DayStyle**, **WeekendDayStyle**, **TodayDayStyle**, **SelectedDayStyle**, **OtherMonthDayStyle**, and **NextPrevStyle**. You can set individual properties of these styles using the subproperty syntax PropertyName-SubPropertyName, as the following sample demonstrates.

VB Style5.aspx

[Run Sample] | [View Source]

A slightly different syntax allows each **Style** property to be declared as a child element nested within Web server control tags.

```
<ASP:Calendar ... runat="server">
    <TitleStyle BorderColor="darkolivegreen" BorderWidth="3"
            BackColor="olivedrab" Height="50px" />
</ASP:Calendar>
```

The following sample shows alternative syntax but is functionally equivalent to the preceding one.

VB Style6.aspx

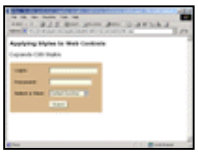
[Run Sample] | [View Source]

As with HTML server controls, you can apply styles to Web server controls using a CSS class definition. The **WebControl** base class exposes a **String** property named **CssClass** for setting the style class:

VB Style7.aspx

[Run Sample] | [View Source]

If an attribute is set on a server control that does not correspond to any strongly typed property on the control, the attribute and value are populated in the **Attributes** collection of the control. By default, server controls will render these attributes unmodified in the HTML returned to the requesting browser client. This means that the style and class attributes can be set on Web server controls directly instead of using the strongly typed properties. While this requires some understanding of the actual rendering of the control, it can be a flexible way to apply styles as well. It is especially useful for the standard form input controls, as illustrated in the following sample.



VB Style8.aspx

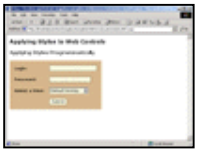
[Run Sample] | [View Source]

Web server control styles can also be set programmatically, using the **ApplyStyle** method of the base **WebControl** class, as in the following code.

```
<script language="VB" runat="server">
    Sub Page_Load(Src As Object, E As EventArgs)
        Dim MyStyle As New Style
        MyStyle.BorderColor = Color.Black
        MyStyle.BorderStyle = BorderStyle.Dashed
        MyStyle.BorderWidth = New Unit(1)

        MyLogin.ApplyStyle (MyStyle)
        MyPassword.ApplyStyle (MyStyle)
        MySubmit.ApplyStyle (MyStyle)
    End Sub
</script>

Login: <ASP:TextBox id="MyLogin" runat="server" />/<p/>
Password: <ASP:TextBox id="MyPassword" TextMode="Password" runat="server" />
View:  <ASP:DropDownList id="MySelect" runat="server">  ...  </ASP:DropDownList>
```

VB

The following sample demonstrates the code above.



VB Style9.aspx

[Run Sample] | [View Source]

Section Summary

1. ASP.NET's HTML server control and Web server control families provide first-class support for CSS styles.
2. Styles may be applied by setting either the style or the class attributes of a control. These settings are accessible programmatically through the control's **Attributes** collection. In the case of HTML server controls, individual values for style-attribute keys can be retrieved from the control's **Style** collection.
3. Most commonly used style settings are exposed on Web server controls as strongly typed properties of the control itself.
4. The **System.Web.UI.WebControls** namespace includes a **Style** base class that encapsulates common style attributes. Many Web server controls expose properties of this type to control individual rendering elements.
5. Styles may be set programmatically on Web server controls using the **ApplyStyle** method of the **WebControl** base class.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Server Control Form Validation

- Introduction to Validation
- Types of Validation Controls
- Client-Side Validation
- Displaying Validation Errors
- Working with CompareValidator
- Working with RangeValidator
- Working with Regular Expressions
- Performing Custom Validation
- Bringing It All Together
- Section Summary

Introduction to Validation

The Web Forms framework includes a set of validation server controls that provide an easy-to-use but powerful way to check input forms for errors and, if necessary, display messages to the user.

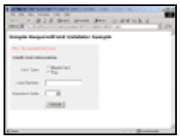
Validation controls are added to a Web Forms page like other server controls. There are controls for specific types of validation, such as range checking or pattern matching, plus a **RequiredFieldValidator** that ensures that a user does not skip an entry field. You can attach more than one validation control to an input control. For example, you might specify both that an entry is required and that it must contain a specific range of values.

Validation controls work with a limited subset of HTML and Web server controls. For each control, a specific property contains the value to be validated. The following table lists the input controls that may be validated.

Control	Validation Property
HtmlInputText	Value
HtmlTextArea	Value
HtmlSelect	Value
HtmlInputFile	Value
TextBox	Text
ListBox	SelectedItem.Value
DropDownList	SelectedItem.Value
RadioButtonList	SelectedItem.Value

Types of Validation Controls

The simplest form of validation is a required field. If the user enters any value in a field, it is valid. If all of the fields in the page are valid, the page is valid. The following example illustrates this using the **RequiredFieldValidator**.



VB Validator1.aspx

[Run Sample] | [View Source]

There are also validation controls for specific types of validation, such as range checking or pattern matching. The following table lists the validation controls.

Control Name	Description
RequiredFieldValidator	Ensures that the user does not skip an entry.
CompareValidator	Compares a user's entry with a constant value or a property value of another control using a comparison operator (less than, equal to, greater than, and so on).
RangeValidator	Checks that a user's entry is between specified lower and upper boundaries. You can check ranges within pairs of numbers, alphabetic characters, or dates. Boundaries can be expressed as constants.

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

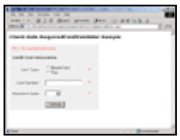
RegularExpressionValidator	Checks that the entry matches a pattern defined by a regular expression. This type of validation allows you to check for predictable sequences of characters, such as those in social security numbers, e-mail addresses, telephone numbers, postal codes, and so on.
CustomValidator	Checks the user's entry using validation logic that you code yourself. This type of validation allows you to check for values derived at run time.
ValidationSummary	Displays the validation errors in summary form for all of the validators on a page.

Client-Side Validation

The validation controls always perform validation checking in server code. However, if the user is working with a browser that supports DHTML, the validation controls can also perform validation using client script. With client-side validation, any errors are detected on the client when the form is submitted to the server. If any of the validators are found to be in error, the submission of the form to the server is cancelled and the validator's **Text** property is displayed. This permits the user to correct the input before submitting the form to the server. Field values are revalidated as soon as the field containing the error loses focus, thus providing the user with a rich, interactive validation experience.

Note that the Web Forms page framework always performs validation on the server, even if the validation has already been performed on the client. This helps prevent users from being able to bypass validation by impersonating another user or a preapproved transaction.

Client-side validation is enabled by default. If the client is capable, uplevel validation will be performed automatically. To disable client-side validation, set the page's **ClientTarget** property to "Downlevel" ("Uplevel" forces client-side validation).



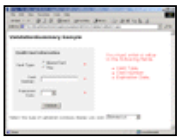
VB Validator2.aspx

[Run Sample] | [View Source]

Displaying Validation Errors

When the user's input is processed (for example, when the form is submitted), the Web Forms page framework passes the user's entry to the associated validation control or controls. The validation controls test the user's input and set a property to indicate whether the entry passed the validation test. After all validation controls have been processed, the **IsValid** property on the page is set; if any of the controls shows that a validation check failed, the entire page is set to invalid.

If a validation control is in error, an error message may be displayed in the page by that validation control or in a **ValidationSummary** control elsewhere on the page. The **ValidationSummary** control is displayed when the **IsValid** property of the page is false. It polls each of the validation controls on the page and aggregates the text messages exposed by each. The following example illustrates displaying errors with a **ValidationSummary** control.



VB Validator3.aspx

[Run Sample] | [View Source]

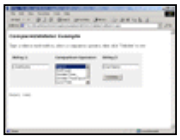
Working with **CompareValidator**

The **CompareValidator** server control compares the values of two controls. **CompareValidator** uses three key properties to perform its validation. **ControlToValidate** and **ControlToCompare** contain the values to compare. **Operator** defines the type of comparison to perform--for example, Equal or Not Equal. **CompareValidator** performs the validation by evaluating these properties as an expression, as follows:

```
   ( ControlToValidate <Operator> ControlToCompare )
```

`If the expression evaluates true, the validation result is valid.`

`The following sample shows how to use the` **`CompareValidator`** `control.`



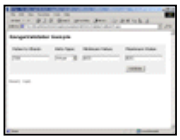
VB Validator4.aspx

[Run Sample] | [View Source]

Working with **RangeValidator**

The **RangeValidator** server control tests whether an input value falls within a given range.
RangeValidator uses three key properties to perform its validation. **ControlToValidate** contains the value
to validate. **MinimumValue** and **MaximumValue** define the minimum and maximum values of the valid
range.

This sample shows how to use the **RangeValidator** control.



VB Validator5.aspx

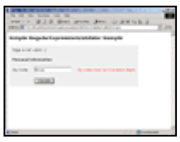
[Run Sample] | [View Source]

Working with Regular Expressions

The **RegularExpressionValidator** server control checks that the entry matches a pattern defined by a
regular expression. This type of validation allows you to check for predictable sequences of characters,
such as those in social security numbers, e-mail addresses, telephone numbers, postal codes, and so on.

RegularExpressionValidator uses two key properties to perform its validation. **ControlToValidate** contains
the value to validate. **ValidationExpression** contains the regular expression to match.

These samples illustrates using the RegularExpressionValidator control.



VB Validator6.aspx

[Run Sample] | [View Source]

VB Validator7.aspx

[Run Sample] | [View Source]

Performing Custom Validation

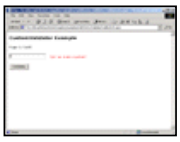
The **CustomValidator** server control calls a user-defined function to perform validations that the standard
validators can't handle. The custom function can execute on the server or in client-side script, such as JScript or VBScript.
For client-side custom validation, the name of the custom function must be identified in the **ClientValidationFunction**
property. The custom function must have the form

```
   function myvalidator(source, arguments)
```

Note that **source** is the client-side **CustomValidator** object, and **arguments** is an object with two properties,
Value and **IsValid**. The **Value** property is the value to be validated and the **IsValid** property is a
Boolean used to set the return result of the validation.

For server-side custom validation, place your custom validation in the validator's **OnServerValidate**
delegate.

The following sample shows how to use the **CustomValidator** control.

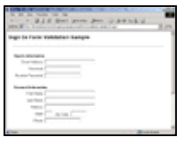


VB Validator8.aspx

[Run Sample] | [View Source]

Bringing It All Together

This sample shows a typical registration form, using the variations of validation controls discussed in this topic.



VB Validator9.aspx

[Run Sample] | [View Source]

Section Summary

1. Validator controls can be used to validate input on any Web Forms page.
2. More than one control can be used on a given input field.
3. Client-side validation may be used in addition to server validation to improve form usability.
4. The **CustomValidator** control lets the user define custom validation criteria.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles

Web Forms User Controls

- Introduction to User Controls
- Exposing User Control Properties
- Encapsulating Events in a User Control
- Creating User Controls Programmatically
- Section Summary

Introduction to User Controls

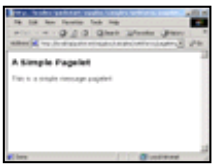
In addition to the built-in server controls provided by ASP.NET, you can easily define your own controls using the same programming techniques that you have already learned for writing Web Forms pages. In fact, with just a few modifications, almost any Web Forms page can be reused in another page as a server control (note that a user control is of type **System.Web.UI.UserControl**, which inherits directly from **System.Web.UI.Control**). A Web Forms page used as a server control is named a user control for short. As a matter of convention, the .ascx extension is used to indicate such controls. This ensures that the user control's file cannot be executed as a standalone Web Forms page (you will see a little that there are a few, albeit important, differences between a user control and a Web Forms page). User controls are included in a Web Forms page using a **Register** directive:

```
<%@ Register TagPrefix="Acme" TagName="Message" Src="pagelet1.ascx" %>
```

The **TagPrefix** determines a unique namespace for the user control (so that multiple user controls with the same name can be differentiated from each other). The **TagName** is the unique name for the user control (you can choose any name). The **Src** attribute is the virtual path to the user control--for example "MyPagelet.ascx" or "/MyApp/Include/MyPagelet.ascx". After registering the user control, you may place the user control tag in the Web Forms page just as you would an ordinary server control (including the **runat="server"** attribute):

```
<Acme:Message runat="server"/>
```

The following example shows a user control imported into another Web Forms page. Note that the user control in this case is just a simple static file.



VB Pagelet1.aspx

[Run Sample] | [View Source]

Exposing User Control Properties

When a Web Forms page is treated as a control, the public fields and methods of that Web Form are promoted to public properties (that is, tag attributes) and methods of the control as well. The following example shows an extension of the previous user control example that adds two public **String** fields. Notice that these fields can be set either declaratively or programmatically in the containing page.

User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

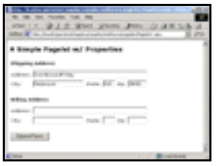
Get URL for this page

VB Pagelet2.aspx

[Run Sample] | [View Source]

In addition to promoting public fields to control properties, the property syntax may be used. Property syntax has the advantage of being able to execute code when properties are set or retrieved. The following example demonstrates an **Address** user control that wraps the text properties of **TextBox** controls within it. The benefit of doing this is that the control inherits the automatic state management of the **TextBox** control for free.

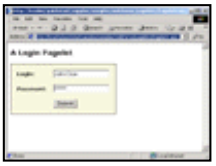
Notice that there are two **Address** user controls on the containing Web Forms page that set the **Caption** property to "Billing Address" and "Shipping Address", respectively. The real power of user controls is in this type of reusability.



VB Pagelet3.aspx

[Run Sample] | [View Source]

Another useful user control is a **Login** control for collecting user names and passwords.



VB Pagelet4.aspx

[Run Sample] | [View Source]

In this example, form validation controls are added to the **Login** user control.

VB Pagelet5.aspx

[Run Sample] | [View Source]

Encapsulating Events in a User Control

User controls participate in the complete execution lifecycle of the request, much the way ordinary server controls do. This means that a user control can handle its own events, encapsulating some of the page logic from the containing Web Forms page. The following example demonstrates a product-listing user control that internally handles its own postbacks. Note that the user control itself has no wrapping **<form runat="server">** control. Because only one form control may be present on a page (ASP.NET does not allow nested server

forms), it is left to the containing Web Forms page to define this.

VB Pagelet6.aspx

[Run Sample] | [View Source]

Creating User Controls Programmatically

Just as ordinary server controls can be created programmatically, so user controls can be. The page's **LoadControl** method is used to load the user control, passing the virtual path to the user control's source file:

```
Dim c1 As Control = LoadControl("pagelet7.ascx")
CType(c1, (Pagelet7VB)).Category = "business"
Page.Controls.Add(c1)
```

VB

The type of the user control is determined by a **ClassName** attribute on the **Control** directive. For example, a user control saved with the file name "pagelet7.ascx" is assigned the strong type "Pagelet7CS" as follows:

```
<%@ Control ClassName="Pagelet7CS" %>
```

Because the **LoadControl** method returns a type of **System.Web.UI.Control**, it must be cast to the appropriate strong type in order to set individual properties of the control. Finally, the user control is added to the base page's **ControlCollection**.

VB Pagelet7.aspx

[Run Sample] | [View Source]

Important The strong type for a user control is available to the containing Web Forms page only if a **Register** directive is included for the user control (even if there are no user control tags actually declared).

Section Summary

1. User controls allow developers to easily define custom controls using the same programming techniques as for writing Web Forms pages.
2. As a matter of convention, an .ascx file name extension is used to indicate such controls. This ensures that a user control file cannot be executed as a standalone Web Forms page.
3. User controls are included into another Web Forms page using a **Register** directive, which specifies a **TagPrefix**, **TagName**, and **Src location**.
4. After the user control has been registered, a user control tag may be placed in a Web Forms page as an ordinary server control (including the **runat="server"** attribute).
5. The public fields, properties, and methods of a user control are promoted to public

properties (tag attributes) and methods of the control in the containing Web Forms page.

6. User controls participate in the complete execution lifecycle of every request and can handle their own events, encapsulating some of the page logic from the containing Web Forms page.
7. User controls should not contain any form controls but should instead rely on their containing Web Forms page to include one if necessary.
8. User controls may be created programmatically using the **LoadControl** method of the **System.Web.UI.Page** class. The type of the user control is determined by the ASP.NET runtime, following the convention filename_extension.
9. The strong type for a user control is available to the containing Web Forms page only if a **Register** directive is included for the user control (even if there are no user control tags actually declared).

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Data Binding Server Controls

- Data Binding Overview and Syntax
- Binding to Simple Properties
- Binding to Collections & Lists
- Binding Expressions or Methods
- DataBinder.Eval()
- Section Summary

Data Binding Overview and Syntax

ASP.NET introduces a new declarative data binding syntax. This extremely flexible syntax permits the developer to bind not only to data sources, but also to simple properties, collections, expressions, and even results returned from method calls. The following table shows some examples of the new syntax.

Simple property	Customer: <%# custID %>
Collection	Orders: <asp:ListBox id="List1" datasource='<%# myArray %>' runat="server">
Expression	Contact: <%# (customer.First Name + " " + customer.LastName) %>
Method result	Outstanding Balance: <%# GetBalance(custID) %>

Although this syntax looks similar to the ASP shortcut for **Response.Write** -- <%= %> -- its behavior is quite different. Whereas the ASP **Response.Write** shortcut syntax was evaluated when the page was processed, the ASP.NET data binding syntax is evaluated only when the **DataBind** method is invoked.

DataBind is a method of the **Page** and all server controls. When you call **DataBind** on a parent control, it cascades to all of the children of the control. So, for example, `DataList1.DataBind()` invokes the **DataBind** method on each of the controls in the **DataList** templates. Calling **DataBind** on the **Page** -- `Page.DataBind()` or simply `DataBind()` -- causes all data binding expressions on the page to be evaluated. **DataBind** is commonly called from the **Page_Load** event, as shown in the following example.

```
Protected Sub Page_Load(Src As Object, E As EventArgs)
    DataBind()
End Sub
```

VB

You can use a binding expression almost anywhere in the declarative section of an .aspx page, provided it evaluates to the expected data type at run time. The simple property, expression, and method examples above display text to the user when evaluated. In these cases, the data binding expression must evaluate to a value of type **String**. In the collection example, the data binding expression evaluates to a value of valid type for the **DataSource** property of **ListBox**. You might find it necessary to coerce the type of value in your binding expression to produce the desired result. For example, if `count` is an integer:

```
Number of Records: <%# count.ToString() %>
```

Binding to Simple Properties

The ASP.NET data binding syntax supports binding to public variables, properties of the **Page**, and properties of other controls on the page.

The following example illustrates binding to a public variable and simple property on the page. Note that these values are initialized before `DataBind()` is called.

VB DataBind1.aspx

[Run Sample] | [View Source]

The following example illustrates binding to a property of another control.

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

VB DataBind2.aspx

[Run Sample] | [View Source]

Binding to Collections and Lists

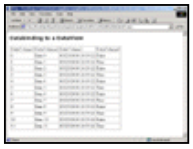
List server controls like **DataGrid**, **ListBox** and **HTMLSelect** use a collection as a data source. The following examples illustrate binding to usual common language runtime collection types. These controls can bind only to collections that support the **IEnumerable**, **ICollection**, or **IListSource** interface. Most commonly, you'll bind to **ArrayList**, **Hashtable**, **DataView** and **DataReader**.

The following example illustrates binding to an **ArrayList**.

VB DataBind3.aspx

[Run Sample] | [View Source]

The following example illustrates binding to a **DataView**. Note that the **DataView** class is defined in the **System.Data** namespace.



VB DataBind4.aspx

[Run Sample] | [View Source]

The following example illustrates binding to a **Hashtable**.

VB DataBind5.aspx

[Run Sample] | [View Source]

Binding Expressions or Methods

Often, you'll want to manipulate data before binding to your page or a control. The following example illustrates binding to an expression and the return value of a method.

VB DataBind6.aspx

[Run Sample] | [View Source]

DataBinder.Eval

The ASP.NET framework supplies a static method that evaluates late-bound data binding expressions and optionally formats the result as a string. **DataBinder.Eval** is convenient in that it eliminates much of the explicit casting the developer must do to coerce values to the desired data type. It is particularly useful when data binding controls within a templated list, because often both the data row and the data field must be cast.

Consider the following example, where an integer will be displayed as a currency string. With the standard ASP.NET data binding syntax, you must first cast the type of the data row in order to retrieve the data field, `IntegerValue`. Next, this is passed as an argument to the **String.Format** method.

```
<%# String.Format("{0:c}", (CType(Container.DataItem, DataRowView)("IntegerValue")))
%>
```

VB

This syntax can be complex and difficult to remember. In contrast, **DataBinder.Eval** is simply a method with three arguments: the naming container for the data item, the data field name, and a format string. In a templated list like **DataList**, **DataGrid**, or **Repeater**, the naming container is always `Container.DataItem`. **Page** is another naming container that can be used with **DataBinder.Eval**.

```
<%# DataBinder.Eval(Container.DataItem, "IntegerValue", "{0:c}") %>
```

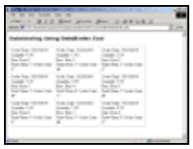
VB

The format string argument is optional. If it is omitted, **DataBinder.Eval** returns a value of type object, as shown in the following example.

```
<%# CType(DataBinder.Eval(Container.DataItem, "BoolValue"), Boolean) %>
```

VB

It is important to note that **DataBinder.Eval** can carry a noticeable performance penalty over the standard data binding syntax because it uses late-bound reflection. Use **DataBinder.Eval** judiciously, especially when string formatting is not required.



VB DataBind7.aspx

[Run Sample] | [View Source]

Section Summary

1. The ASP.NET declarative data binding syntax uses the <%# %> notation.
2. You can bind to data sources, properties of the page or another control, collections, expressions, and results returned from method calls.
3. List controls can bind to collections that support the **ICollection**, **IEnumerable**, or **IListSource** interface, such as **ArrayList**, **Hashtable**, **DataView**, and **DataReader**.
4. **DataBinder.Eval** is a static method for late binding. Its syntax can be simpler than the standard data binding syntax, but performance is slower.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Server-Side Data Access

- Introduction to Server-Side Data
- Connections, Commands, and DataSets
- Accessing SQL-based Data
- Binding SQL Data to a DataGrid
- Performing a Parameterized Select
- Inserting Data in a SQL Database
- Updating Data in a SQL Database
- Deleting Data in a SQL Database
- Sorting Data from a SQL Database
- Working with Master-Detail Relationships
- Writing and Using Stored Procedures
- Accessing XML-based Data
- Section Summary

Introduction to Server-Side Data

Data access is the heart of any real-world application, and ASP.NET provides a rich set of controls that are well-integrated with the managed data access APIs provided in the common language runtime. This section walks through several iterations of a sample that uses the ASP.NET **DataGrid** control to bind to the results of SQL queries and XML data files. This section assumes some familiarity with database fundamentals and the SQL query language.

Server-side data access is unique in that Web pages are basically stateless, which presents some difficult challenges when trying to perform transactions such as inserting or updating records from a set of data retrieved from a database. As you'll see in this section, the **DataGrid** control can help manage these challenges, allowing you to concentrate more on your application logic and less on the details of state management and event handling.

Connections, Commands, and Datasets

The common language runtime provides a complete set of managed data access APIs for data-intensive application development. These APIs help to abstract the data and present it in a consistent way regardless of its actual source (SQL Server, OLEDB, XML, and so on). There are essentially three objects you will work with most often: connections, commands, and datasets.

- A connection represents a physical connection to some data store, such as SQL Server or an XML file.
- A command represents a directive to retrieve from (select) or manipulate (insert, update, delete) the data store.
- A dataset represents the actual data an application works with. Note that datasets are always disconnected from their source connection and data model and can be modified independently. However, changes to a dataset can be easily reconciled with the originating data model.

For a more detailed walkthrough of the managed data access solution in the common language runtime, please read the ADO.NET Overview section of this tutorial.

Accessing SQL-based Data

An application typically needs to perform one or more select, insert, update, or delete queries to a SQL database. The following table shows an example of each of these queries.

Query	Example
Simple Select	SELECT * from Employees WHERE FirstName = 'Bradley';
Join Select	SELECT * from Employees E, Managers M WHERE E.FirstName = M.FirstName;
Insert	INSERT into Employees VALUES ('123-45-6789','Bradley','Millington','Program Manager');
Update	UPDATE Employees SET Title = 'Development Lead' WHERE FirstName = 'Bradley';
Delete	DELETE from Employees WHERE Productivity < 10;

To give your page access to the classes you will need to perform SQL data access, you must import the **System.Data** and **System.Data.SqlClient** namespaces into your page.

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

```
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
```

To perform a select query to a SQL database, you create a **SqlConnection** to the database passing the connection string, and then construct a **SqlDataAdapter** object that contains your query statement. To populate a **DataSet** object with the results from the query, you call the command's **Fill** method.

```
Dim myConnection As New
SqlConnection("server=(local)\NetSDK;database=pubs;Trusted_Connection=yes")
Dim myCommand As New SqlDataAdapter("select * from Authors", myConnection)

Dim ds As New DataSet()
myCommand.Fill(ds, "Authors")
```

VB

As mentioned earlier in this section, the benefit of using a dataset is that it gives you a disconnected view of the database. You can operate on a dataset in your application, and then reconcile your changes with the actual database later. For long-running applications this is often the best approach. For Web applications, you are usually performing short operations with each request (commonly to simply display the data). You often don't need to hold a **DataSet** object over a series of several requests. For situations like these, you can use a **SqlDataReader**.

A **SqlDataReader** provides a forward-only, read-only pointer over data retrieved from a SQL database. To use a **SqlDataReader**, you declare a **SqlCommand** instead of a **SqlDataAdapter**. The **SqlCommand** exposes an **ExecuteReader** method that returns a **SqlDataReader**. Note also that you must explicitly open and close the **SqlConnection** when you use a **SqlCommand**. After a call to **ExecuteReader**, the **SqlDataReader** can be bound to an ASP.NET server control, as you'll see in the next section.

```
Dim myConnection As SqlConnection = New
SqlConnection("server=(local)\NetSDK;database=pubs;Trusted_Connection=yes")
Dim myCommand As SqlCommand = New SqlCommand("select * from Authors", myConnection)

myConnection.Open()

Dim dr As SqlDataReader = myCommand.ExecuteReader()

...

myConnection.Close()
```

VB

When performing commands that do not require data to be returned, such as inserts, updates, and deletes, you also use a **SqlCommand**. The command is issued by calling an **ExecuteNonQuery** method, which returns the number of rows affected. Note that the connection must be explicitly opened when you use the **SqlCommand**; the **SqlDataAdapter** automatically handles opening the connection for you.

```
Dim myConnection As New
SqlConnection("server=(local)\NetSDK;database=pubs;Trusted_Connection=yes")
Dim myCommand As New SqlCommand( _
                  "UPDATE Authors SET phone='(800) 555-5555' WHERE au_id = '123-45-
6789'",  _
                   myConnection)

myCommand.Connection.Open()
myCommand.ExecuteNonQuery()
myCommand.Connection.Close()
```

VB

Important: Always remember to close the connection to the data model before the page finishes executing. If you do not close the connection, you might inadvertently exhaust the connection limit while waiting for the page instances to be handled by garbage collection.

Binding SQL Data to a DataGrid

The following sample shows a simple select query bound to a **DataGrid** control. The **DataGrid** renders a table containing the SQL data.

VB DataGrid1.aspx

[Run Sample] | [View Source]

Like the **DropDownList** shown in the Data Binding section, the **DataGrid** control supports a **DataSource** property that takes an **IEnumerable** or **ICollection**, as well as a **DataSet**. You can use a **DataSet** by assigning the **DefaultView** property of a table contained in the **DataSet** to the name of the table you wish to use within the **DataSet**. The **DefaultView** property represents the current state of a table within a **DataSet**, including any changes which have been made by application code (row deletions or value changes, for example). After setting the **DataSource** property, you call `DataBind()` to populate the control.

```
MyDataGrid.DataSource=ds.Tables("Authors").DefaultView
MyDataGrid.DataBind()
```

VB

An alternative syntax is to specify both a **DataSource** and a **DataMember**. In this case, ASP.NET automatically gets the **DefaultView** for you.

```
MyDataGrid.DataSource=ds
MyDataGrid.DataMember="Authors"
MyDataGrid.DataBind()
```

VB

You can also bind directly to a **SqlDataReader**. In this case you are only displaying data, so the forward-only nature of the **SqlDataReader** is perfectly suited to this scenario, and you benefit from the performance boost that **SqlDataReader** provides.

VB DataGrid1.1.aspx

[Run Sample] | [View Source]

Note: For the remainder of this section, only the **DataSet** model of data access is shown; however, any of these samples could be re-written to take advantage of **SQLDataReader** as well.

Performing a Parameterized Select

You can also perform a parameterized select using the **SqlDataAdapter** object. The following sample shows how you can modify the data selected using the value posted from a select **HtmlControl**.

VB DataGrid2.aspx

[Run Sample] | [View Source]

The **SqlDataAdapter** maintains a **Parameters** collection that can be used to replace variable identifiers (denoted by an "@" in front of the name) with values. You add a new **SqlParameter** to this collection that specifies the name, type, and size of the parameter, and then set its **Value** property to the value of the select.

```
myCommand.SelectCommand.Parameters.Add(New SqlParameter("@State", SqlDbType.NVarChar,
```

```
2))
myCommand.SelectCommand.Parameters("@State").Value = MySelect.Value
```

VB

Important: Note that the **DataGrid**'s **EnableViewState** property has been set to **false**. If the data will be populated in each request, there is no benefit to having the **DataGrid** store state information to be sent through a round trip with form posts. Because the **DataGrid** stores all of its data when maintaining state, it is important to turn it off when appropriate to improve the performance of your pages.

DataGrid2.aspx statically populates the values of the select box, but this will not work well if those values ever change in the database. Because the select **HtmlControl** also supports an **IEnumerable DataSource** property, you can use a select query to dynamically populate the select box instead, which guarantees that the database and user interface are always in sync. The following sample demonstrates this process.

VB DataGrid3.aspx

[Run Sample] | [View Source]

Inserting Data in a SQL Database

To insert a row into the database, you can add a simple input form to the page, and execute an insert command in the form submit event handler. Just as in the previous two samples, you use the command object's Parameters collection to populate the command's values. Note that you also check to make sure the required values are not null before attempting to insert into the database. This prevents an accidental violation of the database's field constraints. You also execute the insert command inside of a try/catch block, just in case the primary key for inserted row already exists.

VB DataGrid4.aspx

[Run Sample] | [View Source]

Instead of explicitly checking the input values, you could have just as easily used the validator controls provided with ASP.NET. The following sample shows how to do that. Note that using the RegEx Validator provides the additional benefit of checking the format for the author ID, zip code, and phone number fields.

VB DataGrid5.aspx

[Run Sample] | [View Source]

Updating Data in a SQL Database

Updating a database can often be tricky in Web applications. The **DataGrid** control provides some built-in support for this scenario that makes updates easier. To allow rows to be edited, the **DataGrid** supports an integer **EditItemIndex** property, which indicates which row of the grid should be editable. When this property is set, the **DataGrid** renders the row at that index as text input boxes instead of simple labels. A value of -1 (the default) indicates that no rows are editable. The page can enclose the **DataGrid** in a server-side form and get access to the edited data through the **DataGrid**'s object model.

To figure out which row should be editable, you need a way to accept some input from the user about which row they would like to edit. The **DataGrid** can contain an **EditCommandColumn** that renders links for firing three special events: **EditCommand**, **UpdateCommand**, and **CancelCommand**. The **EditCommandColumn** is

declaratively added to the **DataGrid**'s Columns collection, as shown in the following example.

```
<ASP:DataGrid id="MyDataGrid" runat="server"
  ...
  OnEditCommand="MyDataGrid_Edit"
  OnCancelCommand="MyDataGrid_Cancel"
  OnUpdateCommand="MyDataGrid_Update"
  DataKeyField="au_id"
>

  <Columns>
    <asp:EditCommandColumn EditText="Edit" CancelText="Cancel" UpdateText="Update" />
  </Columns>

</ASP:DataGrid>
```

On the **DataGrid** tag itself, you wire event handlers to each of the commands fired from the **EditCommandColumn**. The **DataGridCommandEventArgs** argument of these handlers gives you direct access to the index selected by the client, which you use to set the **DataGrid**'s **EditItemIndex**. Note that you need to re-bind the **DataGrid** for the change to take effect, as shown in the following example.

```
Public Sub MyDataGrid_Edit(sender As Object, E As DataGridCommandEventArgs)
    MyDataGrid.EditItemIndex = E.Item.ItemIndex
    BindGrid()
End Sub
```

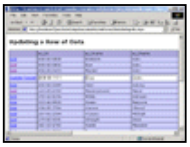
VB

When a row of the **DataGrid** is being edited, the **EditCommandColumn** renders the **Update** and **Cancel** links. If the client selects **Cancel**, you simply set the **EditItemIndex** back to -1. If the client selects **Update**, however, you need to execute your update command to the database. Performing an update query requires that you know the primary key in the database for the row you wish to update. To support this, the **DataGrid** exposes a **DataKeyField** property that you can set to the field name for the primary key. In the event handler wired to the **UpdateCommand**, you can retrieve the key name from the **DataGrid**'s DataKeys collection. You index into this collection using the **ItemIndex** of the event, as shown in the following example.

```
myCommand.Parameters("@Id").Value = MyDataGrid.DataKeys(CType(E.Item.ItemIndex,
Integer))
```

VB

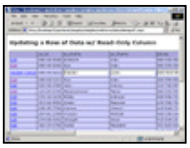
At the end of the Update event handler, you set the **EditItemIndex** back to -1. The following sample demonstrates this code in action.



VB DataGrid6.aspx

[Run Sample] | [View Source]

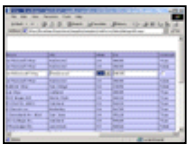
One problem with the preceding example is that the primary key field (au_id) also renders as a text input box when a row is editable. You don't want the client to change this value, because you need it to determine which row to update in the database. Fortunately, you can disable this column from rendering as a text box by specifying exactly what each column looks like for the editable row. You do this by defining each row in the **DataGrid**'s Columns collection, using the **BoundColumn** control to assign data fields with each column. Using this technique gives you complete control over the order of the columns, as well as their **ReadOnly** properties. For the au_id column you set the **ReadOnly** property to **true**. When a row is in edit mode, this column will continue to render as a Label. The following sample demonstrates this technique.



VB DataGrid7.aspx

[Run Sample] | [View Source]

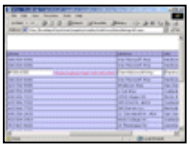
BoundColumn controls aren't the only controls you can set in the **DataGrid**'s Columns collection. You can also specify a **TemplateColumn**, which gives you complete control over the contents of the column. The template is just arbitrary content; you can render anything you like, including server controls, inside the **DataGrid**'s columns. The following sample demonstrates using the **TemplateColumn** control to render the "State" column as a drop-down list and the "Contract" column as a check box HtmlControl. The ASP.NET data binding syntax is used to output the data field value within the template. Note that there is a bit of tricky logic to make the drop-down list and check box reflect the state of the data inside the row.



VB DataGrid8.aspx

[Run Sample] | [View Source]

Just as you can place a drop-down list or check box HtmlControl in a **TemplateColumn**, you can also place other controls there. The following sample adds **Validator** controls to the columns to check the client input before attempting to perform the update.

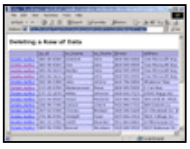


VB DataGrid9.aspx

[Run Sample] | [View Source]

Deleting Data in a SQL Database

Deleting from a database is very similar to an update or insert command, but you still need a way to determine the particular row of the grid to delete. Another control that can be added to the **DataGrid**'s Columns collection is the **ButtonColumn** control, which simply renders a button control. **ButtonColumn** supports a **CommandName** property that can be set to **Delete**. On the **DataGrid**, you wire an event handler to the **DeleteCommand**, where you perform the delete operation. Again, you use the DataKeys collection to determine the row selected by the client. The following sample demonstrates this process.



VB DataGrid10.aspx

[Run Sample] | [View Source]

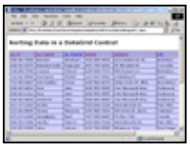
Sorting Data from a SQL Database

A common requirement for any grid is the ability to sort the data it contains. While the **DataGrid** control doesn't explicitly sort its data for you, it does provide a way to call an event handler when the user clicks a column header, which you can use to sort the data. When the **DataGrid**'s **AllowSorting** property is set to **true**, it renders hyperlinks for the column headers that fire a **Sort** command back to the grid. You set the **OnSortCommand** property of the **DataGrid** to the handler you want to call when the user clicks a column link. The name of the column is passed as a **SortExpression** property on the **DataGridSortCommandEventArgs** argument, which you can use to set the **Sort** property of the **DataView** bound to the grid. The following example demonstrates this process.

```
<script>
    Protected Sub MyDataGrid_Sort(Src As Object, E As DataGridSortCommandEventArgs)
        ...
        DataView Source = ds.Tables("Authors").DefaultView
        Source.Sort = E.SortExpression
        MyDataGrid.DataBind()
    End Sub
</script>

<form runat="server">
    <ASP:DataGrid id="MyDataGrid" OnSortCommand="MyDataGrid_Sort" AllowSorting="true"
runat="server" />
</form>
```

VB

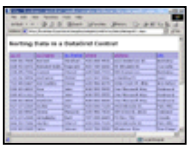
The following sample shows this code in action.



VB DataGrid11.aspx

[Run Sample] | [View Source]

When using **BoundColumn** controls, you can explicitly set the **SortExpression** property for each column, as demonstrated in the following sample.



VB DataGrid12.aspx

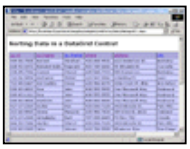
[Run Sample] | [View Source]

Working with Master-Detail Relationships

Often your data model will contain relationships that cannot be represented using just a single grid. A very common Web-based interface is one in which a row of data can be selected that navigates the client to a "details" page, which displays detailed information about the selected row. To accomplish this using the **DataGrid**, you can add a **HyperLinkColumn** to the Columns collection, which specifies the details page to which the client will navigate when the link is clicked. You use the format string syntax to substitute a field value in this link, which is passed as a querystring argument. The following example demonstrates this process.

```
<ASP:DataGrid id="MyDataGrid" runat="server">

  <Columns>
      <asp:HyperLinkColumn
        DataNavigateUrlField="au_id"
        DataNavigateUrlFormatString="datagrid13_details.aspx?id={0}"
        Text="Get Details"
      />
  </Columns>

</ASP:DataGrid>
```

On the details page, you retrieve the querystring argument and perform a join select to obtain details from the database. The following sample demonstrates this scenario.



VB DataGrid13.aspx

[Run Sample] | [View Source]

Writing and Using Stored Procedures

In general, performing ad hoc queries comes at the expense of performance. Using stored procedures can reduce the cost of performing heavy database operations in an application. A stored procedure is easy to create, and can even be done using a SQL statement. The following code example creates a stored procedure that simply returns a table.

```
CREATE Procedure GetAuthors AS
    SELECT * FROM Authors
    return
GO
```

You can create stored procedures that accept parameters as well. For example:

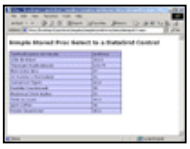
```
CREATE Procedure LoadPersonalizationSettings (@UserId varchar(50)) AS
    SELECT * FROM Personalization WHERE UserID=@UserId
    return
GO
```

Using a stored procedure from an ASP.NET page is just an extension of what you've learned so far about the **SqlCommand** object. The **CommandText** is just the name of the stored procedure instead of the ad hoc query text. You indicate to the **SqlCommand** that the **CommandText** is a stored procedure by setting the **CommandType** property.

```
myCommand.SelectCommand.CommandType = CommandType.StoredProcedure
```

VB

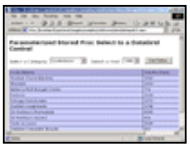
The following sample demonstrates a call to a stored procedure to fill the **DataSet**.



VB DataGrid14.aspx

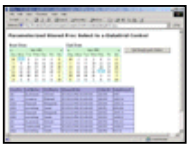
[Run Sample] | [View Source]

Parameters to stored procedures are passed just as they are for ad hoc queries, as shown in the following samples.



VB DataGrid15.aspx

[Run Sample] | [View Source]



VB DataGrid16.aspx

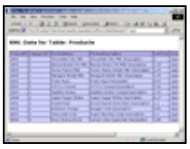
[Run Sample] | [View Source]

Accessing XML-based Data

At the beginning of this section it was mentioned that the **DataSet** was designed to abstract data in a way that is independent of the actual data source. You can see this by changing the focus of your samples from SQL to XML. The **DataSet** supports a **ReadXml** method that takes a **FileStream** object as its parameter. The file you read in this case must contain both a schema and the data you wish to read. The **DataSet** expects data to be in the form, as shown in the following example.

```
<DocumentElement>
    <TableName>
        <ColumnName1>column value</ColumnName1>
        <ColumnName2>column value</ColumnName2>
        <ColumnName3>column value</ColumnName3>
        <ColumnName4>column value</ColumnName4>
    </TableName>
    <TableName>
        <ColumnName1>column value</ColumnName1>
        <ColumnName2>column value</ColumnName2>
        <ColumnName3>column value</ColumnName3>
        <ColumnName4>column value</ColumnName4>
    </TableName>
</DocumentElement>
```

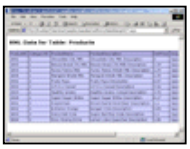
Each **TableName** section corresponds to a single row in the table. The following sample shows reading schema and data from an XML file using the **ReadXml** method of the **DataSet**. Note that after the data is read into the **DataSet** it is indistinguishable from SQL data -- the **DataGrid** binds to it just the same, as shown in the following sample.



VB DataGrid17.aspx

[Run Sample] | [View Source]

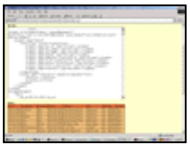
You can also read the data and schema separately, using the **ReadXmlData** and **ReadXmlSchema** methods of the **DataSet**, as shown in the following sample.



VB DataGrid18.aspx

[Run Sample] | [View Source]

Just as the **DataSet** supports reader methods for XML data, it also supports writing the data. The following sample implements a tool for selecting data from SQL and writing the result as XML data or schema text.



VB XMLGen.aspx

[Run Sample] | [View Source]

Section Summary

1. The common language runtime's managed data access APIs abstract data and present it in a consistent way regardless of its actual source (SQL Server, OLEDB, XML, and so on).
2. To give your page access to the classes you will need to perform SQL data access, you must import the **System.Data** and **System.Data.SqlClient** namespaces into your page.
3. Populating a dataset from a SQL query involves creating a **SqlConnection**, associating a **SqlDataAdapter** object with the connection that contains your query statement, and filling the dataset from the command.
4. The **DataGrid** control supports a **DataSource** property that takes an **IEnumerable** (or **ICollection**) type. You can set this to the result of a SQL query by assigning the **DataSet**'s **DefaultView** property, which is of type **DataView**.
5. The **SqlDataAdapter** maintains a Parameters collection that can be used to replace variable identifiers (denoted by an "@" in front of the name) with values.
6. When performing commands that do not require data to be returned, such as inserts, updates, and deletes, you use a **SqlCommand** instead of the **SqlDataAdapter**. The command is issued by calling an **ExecuteNonQuery** method, which returns the number of rows affected.
7. The **SqlConnection** must be explicitly opened when you use the **SqlCommand** (the **SqlDataAdapter** automatically handles opening the connection for you). Always remember to close the **SqlConnection** to the data model before the page finishes executing. If you do not close the connection, you migh inadvertantly exhaust the connection limit while waiting for the page instances to be released to garbage collection.
8. To allow rows to be edited, the **DataGrid** supports an integer **EditItemIndex** property, which indicates which row of the grid should be editable. When this property is set, the **DataGrid** renders the row at that index as text input boxes instead of simple labels.
9. The **DataGrid** exposes a **DataKeyField** property that you can set to the field name for the primary key. In the event handler wired to the **UpdateCommand**, you can retrieve the key name from the **DataGrid**'s DataKeys collection.
10. Using **BoundColumn** controls in the **DataGrid** gives you complete control over the order of the columns, as well as their **ReadOnly** properties.
11. Using **TemplateColumn** controls in the **DataGrid** gives you complete control over the contents of the column.
12. The **ButtonColumn** control can be used to simply render a button control in each row for that column, which can be associated with an event.
13. A **HyperLinkColumn** can be added to the **DataGrid**'s Columns collection, which supports navigating to another page when the link is clicked.
14. When the **DataGrid**'s **AllowSorting** property is set to **true**, it renders hyperlinks for the column headers that fire a **Sort** command back to the grid. You set the **OnSortCommand** property of the **DataGrid** to the handler you want to call when the user clicks a column link.
15. The **DataSet** supports **ReadXml**, **ReadXmlData**, and **ReadXmlSchema** methods that take a **FileStream** as a parameter, which can be used to populate a **DataSet** from an XML file.
16. Using stored procedures can reduce the cost of performing heavy database operations in an application.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Data Access and Customization

- Introduction to Templated Controls
- Handling Postbacks from a Template
- Using Select and Edit Templates
- Finding a Control Inside a Template
- Section Summary

Introduction to Templated Controls

While the **DataGrid** server control demonstrated in the previous section is suitable for many Web application scenarios where a grid-like representation of data is appropriate, many times the presentation of data needs to be much richer. ASP.NET offers two controls, **DataList** and **Repeater**, that give you greater flexibility over the rendering of list-like data. These controls are template-based, and so have no default rendering of their own. The way data is rendered is completely determined by the your implementation of the control's templates, which describe how to present data items.

Like the **DataGrid** control, **DataList** and **Repeater** support a **DataSource** property, which can be set to any **ICollection**, **IEnumerable**, or **IListSource** type. The data in this **DataSource** is bound to the control using its **DataBind** method. Once the data is bound, the format of each data item is described by a template.

The **ItemTemplate** property controls the rendering of each item in the DataSource collection. Inside an **ItemTemplate**, you can define any arbitrary presentation code (HTML or otherwise). Using the ASP.NET data binding syntax, you can insert values from the data bound to the **DataList** or **Repeater** control, as shown in the following example.

```
<ASP:Repeater id="MyRepeater" runat="server">

    <ItemTemplate>
        Hello <%# DataBinder.Eval(Container.DataItem, "name") %> !
    </ItemTemplate>

</ASP:Repeater>
```

The **Container** represents the first control in the immediate hierarchy that supports the **System.Web.UI.INamingContainer** marker interface. In this case, the **Container** resolves to an object of type **System.Web.UI.WebControls.RepeaterItem**, which has a **DataItem** property. As the **Repeater** iterates over the DataSource collection, the **DataItem** contains the current item in this collection. For example, if the data source is set to an **ArrayList** of Employee objects, the **DataItem** is of type Employees. When bound to a **DataView**, the **DataItem** is of type **DataRowView**.

The following example demonstrates a **Repeater** control bound to a **DataView** (returned from a SQL query). **HeaderTemplate** and **FooterTemplate** have also been defined and render at the beginning and end of the list, respectively.

VB DataList1.aspx

[Run Sample] | [View Source]

The **Repeater** control just iterates over the bound data, rendering the **ItemTemplate** once for each item in the DataSource collection. It does not render anything besides the elements contained in its templates. While the **Repeater** is a general purpose iterator, the **DataList** provides some additional features for controlling the layout of the list. Unlike the **Repeater**, **DataList** renders additonal elements, like table rows and cells and spans containing style attributes, outside of the template definition to enable this richer formatting. For example, **DataList** supports **RepeatColumns** and **RepeatDirection** properties that specify whether data should be rendered in multiple columns, and in which direction (vertical or horizontal) the data items should be rendered. **DataList** also supports style attributes, as shown in the following example.

```
<ASP:DataList runat="server" DataSource="<%#MyData%>"
    RepeatColumns="2"
    RepeatDirection="Horizontal"
    ItemStyle-Font-Size="10pt"
    ItemStyle-Font-Name="Verdana"
>
```

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

```
    ...
</ASP:DataList>
```

Note: The remainder of this section concentrates on the many features of the **DataList** control. For more information about the **Repeater** control, refer to the Repeater topic in the Web Forms Controls Reference section of this tutorial.

The following sample demonstrates the use of the **DataList** control. Note that the look of the data items has been changed from the previous example, simply by changing the contents of the control's **ItemTemplate** property. The **RepeatDirection** and **RepeatColumns** properties determine how the **ItemTemplates** are laid out.

VB Datalist2.aspx

[Run Sample] | [View Source]

The following example further demonstrates the infinite flexibility of templates by changing the **ItemTemplate** yet again. This time, one of the **DataItem** values has been substituted for the "src" attribute of an **<img>** tag. The format **String** parameter of **DataBinder.Eval** has also been used to substitute a **DataItem** value in the query string for a URL.

VB Datalist3.aspx

[Run Sample] | [View Source]

Handling Postbacks from a Template

As in the **DataGrid**, you can fire a command from inside a **DataList** template that is passed to an event handler wired to the **DataList** itself. For example, a **LinkButton** inside the **ItemTemplate** might fire a **Select** command. By setting the **OnSelectedIndexChanged** property of the **DataList**, you can call an event handler in response to this command. The following example demonstrates this process.

```
<ASP:DataList id="MyDataList" OnSelectedIndexChanged="MyDataList_Select"
runat="server">

    <ItemTemplate>

        <asp:linkbutton CommandName="Select" runat="server">
            <%# DataBinder.Eval(Container.DataItem, "title") %>
        </asp:linkbutton>

    </ItemTemplate>

</ASP:DataList>
```

The following sample demonstrates this code in action. In the `MyDataList_Select` event handler, you populate several other server controls with the details about the particular selected item.

VB Datalist4.aspx

[Run Sample] | [View Source]

Note that while the **DataList** recognizes a few special commands such as **Select** and **Edit/Update/Cancel**, the

command string fired inside a template can be any arbitrary string. For all commands, the **DataList**'s **OnItemCommand** is fired. You can wire this event to a handler as in the previous example; the following example shows how to do this.

```
<script runat="server">

    Protected Sub MyDataList_ItemCommand(Sender As Object, E As
DataListCommandEventArgs)
        Dim Command As String = E.CommandName

        Select Case Command
            Case "Discuss"
                ShowDiscussions(E.Item.DataItem)
            Case "Ratings"
                ShowRatings(E.Item.DataItem)
        End Select
    End Sub

</script>

<ASP:DataList id="MyDataList" OnItemCommand="MyDataList_ItemCommand" runat="server">

    <ItemTemplate>

        <asp:linkbutton CommandName="Ratings" runat="server">
            View Ratings
        </asp:linkbutton>
        |
        <asp:linkbutton CommandName="Discuss" runat="server">
            View Discussions
        </asp:linkbutton>

    </ItemTemplate>

</ASP:DataList>
```

VB

Note that because more than one command can fire this event handler, you must employ a switch statement to determine the particular command that was fired. The following sample demonstrates this code in action.

VB Datalist5.aspx

[Run Sample] | [View Source]

Using Select and Edit Templates

In addition to handling the **Select** command using a page-level event handler, the **DataList** can respond to this event internally. If a **SelectedItemTemplate** is defined for the **DataList**, the **DataList** renders this template for the item that fired the **Select** command. The following example uses the **SelectedItemTemplate** to make the title of the selected book bold.

VB Datalist6.aspx

[Run Sample] | [View Source]

DataList also supports an **EditItemTemplate** for rendering an item whose index is equal to the **DataList**'s **EditItemIndex** property. For details about how editing and updating works, refer to the Updating Data topic of the Data Access section of this tutorial.

VB Datalist7.aspx

[Run Sample] | [View Source]

Finding a Control Inside a Template

Sometimes it is necessary to locate a control contained inside a template. If a control is given an ID in a template, that control can be retrieved from its container (the first control in the parent hierarchy that supports **INamingContainer**). In this case, the container is the **DataListItem** control. Note that even though there are several controls with the same ID (by virtue of the **DataList**'s repetition), each is contained logically in the namespace of the **DataListItem** container control.

You can go through the **DataList**'s **Items** collection to retrieve the **DataListItem** for a given index, and then call the **DataListItem**'s **FindControl** method (inherited from the base **Control** class) to retrieve a control with a particular ID.

```
<script runat="server">

    Public Sub Page_Load(sender As Object, E As EventArgs))
        ' set datasource and call databind here

        For I=0 To MyDataList.Items.Count-1
           Dim IsChecked As String =
MyDataList.Items(i).FindControl("Save").Checked.ToString()
           If IsChecked = "True" Then
             ...
           End If
        Next
    End Sub
</script>

<ASP:DataList id="MyDataList" runat="server">

    <ItemTemplate>
         <asp:CheckBox id="Save" runat="server"/> <b>Save to Favorites</b>
    </ItemTemplate>

</ASP:DataList>
```

VB

The following sample demonstrates this code in action.

VB Datalist8.aspx

[Run Sample] | [View Source]

Section Summary

1. The **DataList** and **Repeater** controls provide developers fine-tuned control over the rendering of data-bound lists.
2. Rendering of bound data is controlled using a template, such as the **HeaderTemplate**, **FooterTemplate**, or **ItemTemplate**.
3. The **Repeater** control is a general-purpose iterator, and does not insert anything in its rendering that is not contained in a template.
4. The **DataList** control offers more control over the layout and style of items, and outputs its own rendering code for formatting.
5. The **DataList** supports the **Select**, **Edit/Update/Cancel**, and **Item Command** events, which can be handled at the page level by wiring event handlers to the **DataList**'s **Command** events.

6. **DataList** supports a **SelectedItemTemplate** and **EditItemTemplate** for control over the rendering of a selected or editable item.
7. Controls can be programmatically retrieved from a template using the **Control.FindControl** method. This should be called on a **DataListItem** retrieved from the **DataList**'s Items collection.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Working With Business Objects

- The Application /Bin Directory
- Importing Business Objects
- A Simple Two-Tier Web Forms Page
- A Simple Three-Tier Web Forms Page
- Section Summary

Encapsulating logic in business components is an essential part of any real-world application, Web-based or otherwise. In ASP.NET, business objects are the building blocks for multi-tiered Web applications, such as those with a layer for data access or common application rules. This section demonstrates how to write some simple components and include them in your application's Web Forms pages.

The Application /Bin Directory

A problem with using the COM model for Web application components is that those components must be registered (typically using the regsvr32 tool) before they can be used from a traditional ASP application. Remote administration of these types of applications is often not possible, because the registration tool must be run locally on the server. To make matters more difficult, these components remain locked on disk once they are loaded by an application, and the entire Web server must be stopped before these components can be replaced or removed.

ASP.NET attempts to solve these problems by allowing components to be placed in a well-known directory, to be automatically found at run time. This well-known directory is always named **/bin**, and is located immediately under the root directory for the application (a virtual directory defined by Internet Information Services (IIS)). The benefit is that no registration is required to make components available to the ASP.NET Framework application -- components can be deployed by simply copying to the /bin directory or performing an FTP file transfer.

In addition to providing a zero-registration way to deploy compiled components, ASP.NET does not require these components to remain locked on disk at run time. Behind the scenes, ASP.NET duplicates the assemblies found in /bin and loads these "shadow" copies instead. The original components can be replaced even while the Web server is still running, and changes to the /bin directory are automatically picked up by the runtime. When a change is detected, ASP.NET allows currently executing requests to complete, and directs all new incoming requests to the application that uses the new component or components.

Importing Business Objects

At its most basic level, a business component is just a class for which you can create an instance from a Web Forms page that imports it. The following example defines a simple HelloWorld class. The class has one public constructor (which is executed when an instance of the class is first created), a single **String** property called FirstName, and a SayHello method that prints a greeting using the value of the FirstName property.

```
Imports System
Imports System.Text

Namespace HelloWorld
  Public Class HelloObj
    Private _name As String

    Public Sub New
      MyBase.New()
      _name = Nothing
    End Sub

    Public Property FirstName As String
      Get
        Return(_name)
      End get
      Set
        _name = value
       End Set
    End Property

    Public Function SayHello() As String
      Dim sb As New StringBuilder("Hello ")
      If (_name <> Nothing) Then
         sb.Append(_name)
      Else
         sb.Append("World")
      End If
      sb.Append("!")
      Return(sb.ToString())
    End Function
```

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

```
    End Class
End Namespace
```

VB

To compile this class, the C# compiler (Csc.exe) is run from the command line. The **/t** option tells the compiler to build a library (DLL), and the **/out** option tells the compiler where to place the resulting assembly. In this case, the /bin directory for the application is directly under the "aspplus" vroot of this tutorial, and it is assumed this command is being run from the sample directory, that is, ...\QuickStart\AspPlus\Samples\WebForms\Busobjs.

```
csc /t:library /out:..\..\..\..\bin\HelloObj.dll HelloObj.cs
```

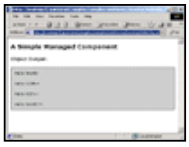
For Visual Basic, the equivalent compilation command is:

```
vbc /t:library /out:..\..\..\..\bin\HelloObjVB.dll HelloObj.vb
```

For JScript, the equivalent compilation command is:

```
jsc /out:..\..\..\..\bin\HelloObjJS.dll HelloObj.js
```

The component is now available to any Web Forms page in the application that needs to use it. The following HelloObj.aspx example illustrates this functionality.



VB HelloObj.aspx

[Run Sample] | [View Source]

Note the **Import** directive at the top of the page that specifies the namespace to include. Once the namespace is included using this directive, the class can be used from within the Web Forms page. Because the assembly is pre-loaded by the ASP.NET runtime, only a simple namespace import is required to make the component available. The following code example the **Import** directive.

```
<%@ Import Namespace="HelloWorld" %>
```

By default, ASP.NET loads all assemblies from the /bin directory when the application is started. The assemblies to load are specifed through the configuration system. For details, see the Configuration Overview section. Additional assemblies can be imported into an application using configuration as well. For example:

```
<configuration>
    <compilation>
        <assemblies>
            <!--The following assemblies are loaded explicitly from the global cache--
>
            <add assembly="System.Data"/>
            <add assembly="System.Web.Services"/>
            <add assembly="System.Drawing"/>
            <!--This tells ASP.NET to load all assemblies from /bin-->
            <add assembly="*"/>
        </assemblies>
    </compilation>
</configuration>
```

Note: Each assembly loaded from /bin is limited in scope to the application in which it is running. This means that peer applications could potentially use different assemblies with the same class or namespace names, without conflicting.

A Simple Two-Tier Web Forms Page

The classic use for an external component is to perform data access. This simplifies the code in your page, improving readability and separating your user interface (UI) logic from the system functionality. The following example demonstrates a simple two-tiered Web Forms page that uses a data access component to retrieve product information.

VB TwoTier.aspx

[Run Sample] | [View Source]

The data access component takes a single parameter to its constructor specifying the connection string to the product database. The Web Forms page calls the component's **GetCategories** method to populate a drop-down list, and calls the component's **GetProductsForCategory** method to display the products for the category selected by the user.

A Simple Three-Tier Web Forms Page

A three-tiered application model extends the two-tiered scenario to include business rules between the UI and data access logic. This model allows UI developers to work with a higher level of abstraction rather than directly manipulating data through low-level data access component APIs. The middle business component typically enforces business rules and ensures that the relationships and primary key constraints of the database are honored. The following example uses the middle component to calculate a discount based on a two-digit Vendor ID entered by the client.

VB ThreeTier.aspx

[Run Sample] | [View Source]

Section Summary

1. The ASP.NET runtime finds business objects (local assemblies) in a well-known /bin directory, directly under the application root. The /bin directory offers the following advantages:
    - **No registration required.** No registration is required to make an assembly available to pages in the application. It is available by virtue of its location in the /bin directory. Compiled code can be deployed by simply copying or FTPing to this location.
    - **No server restart required.** When any part of an ASP.NET Framework application is changed (for example, when a DLL in /bin is replaced), new requests immediately begin execution against the changed file or files. Currently executing requests are allowed to complete before the old application is gracefully torn down. The Web server does not require a restart when you change your application, even when replacing compiled code.
    - **No namespace conflicts.** Each assembly loaded from /bin is limited in scope to the application in which it is running. This means that peer applications could potentially use different assemblies with the same class or namespace names, without conflicting.
2. Classes in an assembly are made available to a page in the application using an **Import** directive within the .aspx file.
3. Two-tiered applications simplify the code in a page, improving readability and separating user interface (UI) logic from system functionality.
4. Three-tiered applications extend the two-tiered model to enable UI developers to work with a higher level of abstraction. The middle business component typically enforces business rules and ensures that the relationships and primary key constraints of the database are honored.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Authoring Custom Controls

- Developing a Simple Custom Control
- Defining Simple Properties
- Defining Class Properties
- Retrieving Inner Content
- Developing a Composite Control
- Handling Events in a Composite Control
- Raising Events from a Composite Control
- Maintaining State
- Developing a Custom (Non-Composite) Control that Handles Post-back Data
- Generating Client-side JavaScript for Custom Post-back
- Developing a Templated Control
- Developing a Templated Databound Control
- Overriding Control Parsing
- Defining a Custom Control Builder

This section of the QuickStart demonstrates how advanced developers can write their own ASP.NET server controls that work within the ASP.NET page framework. By writing your own custom ASP.NET server controls, you can encapsulate custom user interface and other functionality in controls that can be reused on ASP.NET pages. The QuickStart provides an introduction to authoring custom controls through hands-on examples. For more information about control authoring, see Developing ASP.NET Server Controls in the Microsoft .NET Framework SDK documentation.

Note: The controls described in this section might not work correctly at design time in a forms designer such as Microsoft Visual Studio .NET, although they work properly at run time on ASP.NET pages. To work in a designer, a control needs to apply design-time attributes not described here. For details about the design-time attributes you need to apply, see Design-Time Attributes for Components in the SDK documentation.

Developing a Simple Custom Control

It is easy to start authoring your own ASP.NET server controls. To create a simple custom control, all you have to do is to define a class that derives from **System.Web.UI.Control** and override its **Render** method. The **Render** method takes one argument of type **System.Web.UI.HtmlTextWriter**. The HTML that your control wants to send to the client is passed as a string argument to the **Write** method of **HtmlTextWriter**.

The following example demonstrates a simple control that renders a message string.

VB Simple.aspx

[Run Sample] | [View Source]

Defining Simple Properties

Properties are like "smart" fields that have accessor methods. You should expose properties instead of public fields from your controls because properties allow data hiding, can be versioned, and are supported by visual designers. Properties have get/set accessor methods that set and retrieve properties, and allow additional program logic to be performed if needed.

The following sample shows how to add simple properties that correspond to primitive data types such as integer, Boolean, and string. The sample defines three properties - `Message` is of type string, `MessageSize` is of type enumeration, and `Iterations` is of type integer. Note the page syntax for setting simple and enumeration properties.

VB SimpleProperty.aspx

[Run Sample] | [View Source]

Defining Class Properties

If a class A has a property whose type is class B, then the properties of B (if any) are called subproperties of A. The following sample defines a custom control `SimpleSubProperty` that has a property of type `Format`. `Format` is a class that has two primitive properties - `Color` and `Size`, which in turn become subproperties of `SimpleSubProperty`.

VB SimpleSubProperty.aspx

[Run Sample] | [View Source]

Note that ASP.NET has a special syntax for setting subproperties. The following code example shows how to declaratively set the `Format.Color` and `Format.Size` subproperties on `SimpleSubProperty`. The "-" syntax denotes a subproperty.

```
<SimpleControlSamples:SimpleSubProperty Message="Hello There" Format-Color="red"
Format-Size="3" runat=server/>
```

Retrieving Inner Content

Every control has a **Controls** property that it inherits from **System.Web.UI.Control**. This is a collection property that denotes the child controls (if any) of a control. If a control is not marked with the **ParseChildrenAttribute** or marked with **ParseChildrenAttribute(ChildrenAsProperties = false)**, the ASP.NET page framework applies the following parsing logic when the control is used declarartively on a page. If the parser encounters nested controls within the control's tags, it creates instances of them and adds them to the **Controls** property of the control. Literal text between tags is added as a **LiteralControl**. Any other nested elements generate a parser error.

The following sample shows a custom control, `SimpleInnerContent`, that renders text added between its tags by checking if a **LiteralControl** has been added to its Controls collection. If so, it retrieves the **Text** property of the **LiteralControl**, and appends it to its output string.

VB SimpleInnerContent.aspx

[Run Sample] | [View Source]

Note: If your custom control derives from **WebControl**, it will not have the parsing logic described in the sample, because **WebControl** is marked with **ParseChildrenAttribute(ChildrenAsProperties = true)**, which results in a different parsing logic. For more information about the **ParseChildrenAttribute**, see the SDK documentation.

Developing a Composite Control

You can author new controls by combining existing controls using class composition. Composite controls are equivalent to user controls that are authored using ASP.NET page syntax. The main difference between user controls and composite controls is that user controls are persisted as .ascx text files, whereas composite controls

are compiled and persisted in assemblies.

The key steps in developing a composite control are:

- Override the protected **CreateChildControls** method inherited from **Control** to create instances of child controls and add them to the Controls collection.
- If new instances of your composite control will repeatedly be created on a page, implement the **System.Web.UI.INamingContainer** interface. This is a tagging interface that has no methods. When it is implemented by a control, the ASP.NET page framework creates a new naming scope under that control. This ensures that the child controls will have unique IDs in the hierarchical tree of controls.

You do not have to override the **Render** method because child controls provide rendering logic. You can expose properties that synthesize properties of the child controls.

The following sample defines a composite control, `Composition1`, that combines a **System.Web.UI.LiteralControl** and a **System.Web.UI WebControls.TextBox**. `Composition1` exposes a custom property, `Value`, of type integer, that maps to the **Text** property of **TextBox**.

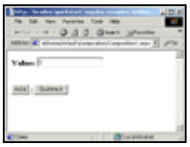
VB Composition1.aspx

[Run Sample] | [View Source]

Handling Events in a Composite Control

A composite control can handle events raised by its child controls. This is accomplished by providing event handling methods and attaching delegates to the events raised by the child controls.

The following sample shows a composite control, `Composition2`, that adds two button controls (named `Add` and `Subtract`) to the composite control from the previous example and provides event handling methods for the **Click** events of the buttons. These methods increment and decrement the `Value` property of `Composition2`. The **CreateChildControls** method of `Composition2` creates instances of event handlers (delegates) that reference these methods, and attaches the delegates to the **Click** events of the **Button** instances. The end result is a control that does its own event handling - when the `Add` button is clicked, the value in the text box is incremented, and when the `Subtract` button is clicked, the value is decremented.



VB Composition2.aspx

[Run Sample] | [View Source]

Raising Custom Events from a Composite Control

A composite control can define custom events that it raises in response to events raised by its child controls.

The following example shows a composite control, `Composition3`, that raises a custom event, `Change`, in response to the **TextChanged** event of the **TextBox** child control.

This is accomplished as follows:

- The custom `Change` event is defined using the standard event pattern. (This pattern includes the definition of a protected `OnChange` method that raises the `Change` event.)

```
Public Event Change(Sender as Object, E as EventArgs)
Protected Sub OnChange(e As EventArgs)
    Change(Me, e)
End Sub
```

VB

- An event-handling method is defined for the **TextChanged** event of **TextBox**. This method raises the `Change` event by invoking the `OnChange` method.

```
Private Sub TextBox_Change(sender As Object, e As EventArgs)
    OnChange(EventArgs.Empty)
End Sub
```

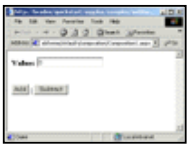
VB

- The **CreateChildControls** method creates an instance of an event handler that references the above method, and attaches the event handler to the **TextChanged** event of the **TextBox** instance.

```
Protected Overrides Sub CreateChildControls()
  ...
  Dim box As New TextBox()
  AddHandler Box.TextChanged, AddressOf TextBox_Change
    ...
End Sub
```

VB

The `Change` event can be handled by a page that hosts the control, as shown in the following sample. In the sample, the page provides an event-handling method for the `Change` event that sets the `Value` property to zero if the number entered by the user is negative.



VB Composition3.aspx

[Run Sample] | [View Source]

Maintaining State

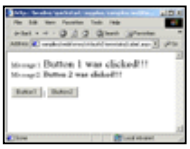
Every Web Forms control has a **State** property (inherited from **Control**) that enables it to participate in **State** management. The type of **State** is **Sytem.Web.UI.StateBag**, which is a data structure equivalent to a hashtable. A control can save data in **State** as key/value pairs. **State** is persisted to a string variable by the ASP.NET page framework and makes a round trip to the client as a hidden variable. Upon postback, the page framework parses the input string from the hidden variable and populates the **State** property of each control in the control hierarchy of a page. A control can restore its state (set properties and fields to their values before postback) using the **State** property. Control developers should be aware that there is a performance overhead in sending data by round trip to the client, and be judicious about what they save in **State**.

The following code example shows a property that is saved in **State**.

```
Public Property Text As String
  Get
    Return CType(State("Text"), String))
  End Get
  Set
    State("Text") = Value
  End Set
End Property
```

VB

The following sample shows a custom control, `Label`, that has two properties, `Text` and `FontSize`, that are saved in **State**. The ASP.NET page that uses **Label** contains buttons that have event handlers to increase the font size of the text in **Label** when a button is clicked. Thus the font size increases every time a button is clicked. This is possible only due to state management - `Label` needs to know what the font size was before postback in order to render the next larger font size after postback.



VB Label.aspx

[Run Sample] | [View Source]

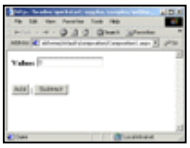
Developing a Custom (Non-Composite) Control that Handles Postback Data

You authored a simple custom control at the beginning of this QuickStart. The following example demonstrates a custom control that does something more meaningful - it renders an input box, and reads in data entered by the user. A control that examines postback (input) data must implement the **System.Web.UI.IPostBackDataHandler** interface. This signals to the ASP.NET page framework that a control should participate in postback data handling. The page framework passes input data to the **LoadPostData** method of this interface as key/value pairs. In its implementation of this method, the control can examine the input data and update its properties as shown below.

```
Private _value As Integer = 0
Public Function LoadPostData(postDataKey As String, values As NameValueCollection) As
Boolean
    _value = Int32.Parse(values(Me.UniqueID))
    Return(False)
End Function
```

VB

The following sample defines a custom control, `NonComposition1`, that implements **IPostBackDataHandler** and has one property, `Value`. The control renders an HTML input box whose text attribute is the string representation of `Value`. The property is set by examining postback input data. The page that uses `NonComposition1` also has two buttons that have event handlers to increment and decrement the `Value` property of `NonComposition1`.



VB NonComposition1.aspx

[Run Sample] | [View Source]

Generating Client-side JavaScript for Custom Postback

If a control wants to capture postback events (form submissions from a client), it must implement the **System.Web.UI.IPostBackEventHandler** interface. This signals to the ASP.NET page framework that a control wants to be notified of a postback event. The **RaisePostBackEvent** method allows the control to handle the event, and to raise other events. Additionally, the ASP.NET page framework has a custom event architecture that allows a control to generate client-side JavaScript that initiates custom postback. Normally, postback is initiated by only a few elements such as a Submit button or an Image button. However, by emitting client-side JavaScript, a control can also initiate postback from other HTML elements.

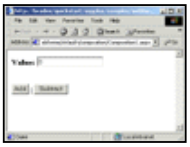
The following example defines a custom control, `NonComposition2`, that builds on the previous example, `NonComposition1`. In addition to the interface provided by `NonComposition1`, it renders two **HtmlButtons** that generate client-side JavaScript to cause postback when clicked. The name attributes of these buttons are Add and Subtract. The name attribute is passed as a string argument to **RaisePostBackEvent** by the page framework. `NonComposition2` implements **RaisePostBackEvent** to increment the `Value` property if Add is clicked and to decrement `Value` if Subtract is clicked, as shown below.

```
Public Sub RaisePostBackEvent(eventArgument As String)
    If eventArgument = "Add" Then
        Me.Value = Me.Value + 1
    Else
        Me.Value = Me.Value - 1
    End If
```

```
End Sub
```

VB

The user interface that is presented to the client is identical to that in the previous example; however, the entire UI is rendered by one custom control that also handles the postback events. The page developer can simply add `NonComposition2` to the page, without providing any event handling logic. The following sample presents this code in action.



VB NonComposition2.aspx

[Run Sample] | [View Source]

Developing a Templated Control

The ASP.NET page framework allows control developers to author controls that separate the user interface from the control logic through the use of templates. Page developers can customize the presentation of the control by providing the UI as parameters between template tags.

Templated controls have one or more properties of type **System.Web.UI.ITemplate**, as shown in the following example.

```
Public Property <TemplateContainer(GetType(Template1VB))> MessageTemplate As
ITemplate
```

VB

The attribute (in square brackets above) specifies the type of the container (parent) control.

The **ITemplate** interface has one method, **InstantiateIn**, that creates a control instance dynamically. This is invoked on the **ITemplate** property in the **CreateChildControls** method, as shown in the following example.

```
Protected Overrides Sub CreateChildControls()
    If MessageTemplate <> Null Then
        MessageTemplate.InstantiateIn(Me)
    End if
    ...
End Sub
```

VB

The following sample shows a simple templated control and an ASP.NET page that uses it.

VB Template1.aspx

[Run Sample] | [View Source]

Developing a Templated Databound Control

The following sample shows a more complex use of templates to create a databound control. The Repeater control defined in this example is similar to the **System.Web.UI.WebControls.Repeater** control.

VB Repeater1.aspx

[Run Sample] | [View Source]

The following sample modifies the preceding sample so that a page consumer can walk its Items collection during postback to pull out values from it.

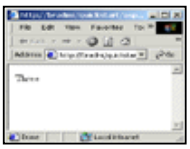
VB Repeater2.aspx

[Run Sample] | [View Source]

Overriding Control Parsing

As you saw in Retrieving Inner Content , if a control A has nested controls within its control tags on a page, the page parser adds instances of those controls to A's Controls collection. This is done by invoking the **AddSubParsedObject** method of A. Every control inherits this method from **Control**; the default implementation simply inserts a child control into the control hierarchy tree. A control can override the default parsing logic by overriding the **AddSubParsedObject** method. Note that this discussion is somewhat simplified; more details are given in the next example.

The following sample defines a custom control, `CustomParse1`, that overrides the default parsing logic. When a child control of a certain type is parsed, it adds it to a collection. The rendering logic of `CustomParse1` is based on the number of items in that collection. A simple custom control, `Item`, is also defined in the sample.



VB CustomParse1.aspx

[Run Sample] | [View Source]

Note: If your custom control derives from **WebControl**, it will not have the parsing logic described in the sample, because **WebControl** is marked with **ParseChildrenAttribute(ChildrenAsProperties = true)**, which results in a different parsing logic. For more information about the **ParseChildrenAttribute**, see the SDK documentation. The Retrieving Inner Content topic also describes this issue in more detail.

Defining a Custom Control Builder

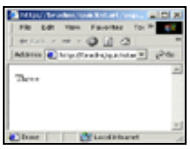
The ASP.NET page framework uses classes called control builders to process the declarations within control tags on a page. Every Web Forms control is associated with a default control builder class, **System.Web.UI.ControlBuilder**. The default control builder adds a child control to the Controls collection for every nested control that it encounters within control tags. Additionally, it adds **Literal** controls for text between nested control tags. You can override this default behavior by associating a custom control builder class with your control. This is done by applying a control builder attribute to your control, as shown in the following example.

```
Public Class <ControlBuilderAttribute(GetType(CustomParse2ControlBuilderVB))> _
  CustomParse2VB : Inherits Control
```

VB

The element in square brackets above is a common language runtime attribute that associates the `CustomParse2ControlBuilder` class with the `CustomParse2` control. You can define your own custom control builder by deriving from **ControlBuilder** and overriding its methods.

The following sample defines a custom control builder that overrides the **GetChildControlType** method inherited from **ControlBuilder**. This method returns the type of the control to be added and can be used to decide which controls will be added. In the example, the control builder will add a child control only if the tag name is "customitem". The code for the control is very similar to the previous example, except for the addition of the custom attribute.



VB CustomParse2.aspx

[Run Sample] | [View Source]

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Web Forms Controls Reference

- System.Web.UI.HtmlControls
- System.Web.UI.WebControls

System.Web.UI.HtmlControls

HTML server controls are HTML elements exposed to the server so you can program against them. HTML server controls expose an object model that maps very closely to the HTML elements that they render.

HtmlAnchor	HtmlButton	HtmlForm	HtmlGenericControl
HtmlImage	HtmlInputButton (Button)	HtmlInputButton (Reset)	HtmlInputButton (Submit)
HtmlInputCheckBox	HtmlInputFile	HtmlInputHidden	HtmlInputImage
HtmlInputRadioButton	HtmlInputText (Password)	HtmlInputText (Text)	HtmlSelect
HtmlTable	HtmlTableCell	HtmlTableRow	HtmlTextArea

System.Web.UI.WebControls

Web server controls are ASP.NET server controls with an abstract, strongly-typed object model. Web server controls include not only form-type controls such as buttons and text boxes, but also special-purpose controls such as a calendar. Web server controls are more abstract than HTML server controls, in that their object model does not necessarily reflect HTML syntax.

AdRotator	Button	Calendar	CheckBox
CheckBoxList	CompareValidator	CustomValidator	DataGrid
DataList	DropDownList	HyperLink	Image
ImageButton	Label	LinkButton	ListBox
Panel	PlaceHolder	RadioButton	RadioButtonList
RangeValidator	RegularExpressionValidator	Repeater	RequiredFieldValidator
Table	TableCell	TableRow	TextBox
ValidationSummary	XML		

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation

Web Forms Syntax Reference

- ASP.NET Web Forms Syntax Elements
- Rendering Code Syntax
- Declaration Code Syntax
- ASP.NET Server Control Syntax
- ASP.NET Html Server Control Syntax
- Databinding Syntax
- Object Tag Syntax
- Server Side Comment Syntax
- Server Side Include Syntax

ASP.NET Web Forms Syntax Elements

An ASP.NET Web Forms page is a declarative text file with an .aspx file name extension. In addition to static content, you can use eight distinct syntax markup elements. This section of the QuickStart reviews each of these syntax elements and provides examples demonstrating their use.

Rendering Code Syntax: <% %> and <%= %>

Code rendering blocks are denoted with <% ... %> elements, allow you to custom-control content emission, and execute during the render phase of Web Forms page execution. The following example demonstrates how you can use them to loop over HTML content.

```
<% For I=0 To 7 %>
     <font size="<%=i%>"> Hello World! </font> <br>
<% Next %>
```

VB

VB Reference1.aspx

[Run Sample] | [View Source]

Code enclosed by <% ... %> is just executed, while expressions that include an equal sign, <%= ... %>, are evaluated and the result is emitted as content. Therefore `<%="Hello World" %>` renders the same thing as the C# code `<% Response.Write("Hello World"); %>`.

Note: For languages that utilize marks to end or separate statements (for example, the semicolon (;) in C#), it is important to place those marks correctly depending on how your code should be rendered.

C# code	
`<% Response.Write("Hello World"); %>`	A semicolon is necessary to end the statement.
`<%="Hello World"; %>`	Wrong: Would result in `"Response.Write("Hello World";);"`.
`<%="Hello World" %>`	A semicolon is not necessary.

Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

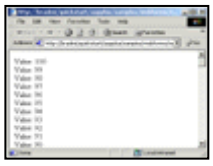
Get URL for this page

Declaration Code Syntax: <script runat="server">

Code declaration blocks define member variables and methods that will be compiled into the generated **Page** class. These blocks can be used to author page/navigation logic. The following example demonstrates how a **Subtract** method can be declared within a **<script runat="server">** block, and then invoked from the page.

```
<script language="VB" runat=server>
Function Subtract(num1 As Integer, num2 As Integer) As Integer
  Return(num1 - num2)
End Function
</script>

<%
  ...
  number = subtract(number, 1)
  ...
%>
```

VB



VB Reference2.aspx

[Run Sample] | [View Source]

Important: Unlike ASP -- where functions could be declared within <% %> blocks -- all functions and global page variables must be declared in a **<script runat=server>** tag. Functions declared within <% %> blocks will now generate a syntax compile error.

ASP.NET Server Control Syntax

Custom ASP.NET server controls enable page developers to dynamically generate HTML user interface (UI) and respond to client requests. They are represented within a file using a declarative, tag-based syntax. These tags are distinguished from other tags because they contain a **"runat=server"** attribute. The following example demonstrates how an **<asp:label runat="server">** server control can be used within an ASP.NET page. This control corresponds to the **Label** class in the **System.Web.UI.WebControls** namespace, which is included by default.

By adding a tag with the ID "Message", an instance of **Label** is created at run time:

```
<asp:label id="Message" font-size=24 runat="server"/>
```

The control can then be accessed using the same name. The following line sets the **Text** property of the control.

```
Message.Text = "Welcome to ASP.NET"
```

VB

VB Reference3.aspx

[Run Sample] | [View Source]

ASP.NET HTML Server Control Syntax

HTML server controls enable page developers to programmatically manipulate HTML elements within a page. An HTML server control tag is distinguished from client HTML elements by means of a "**runat=server**" attribute. The following example demonstrates how an HTML **** server control can be used within an ASP.NET page.

As with other server controls, the methods and properties are accessible programmatically, as shown in the following example.

```
<script language="VB" runat="server">
  Sub Page_Load(sender As Object, e As EventArgs)
    Message.InnerHtml = "Welcome to ASP.NET"
  End Sub
</script>
...
<span id="Message" style="font-size:24" runat="server"/>
```

VB

VB Reference4.aspx

[Run Sample] | [View Source]

Data Binding Syntax: <%# %>

The data binding support built into ASP.NET enables page developers to hierarchically bind control properties to data container values. Code located within a <%# %> code block is only executed when the **DataBind** method of its parent control container is invoked. The following example demonstrates how to use the data binding syntax within an **<asp:datalist runat=server>** control.

Within the datalist, the template for one item is specified. The content of the item template is specified using a data binding expression and the `Container.DataItem` refers to the data source used by the datalist `MyList`.

```
<asp:datalist id="MyList" runat=server>
  <ItemTemplate>
    Here is a value: <%# Container.DataItem %>
  </ItemTemplate>
</asp:datalist>
```

In this case the data source of the `MyList` control is set programmatically, and then `DataBind()` is called.

```
Sub Page_Load(sender As Object, e As EventArgs)
  Dim items As New ArrayList()

  items.Add("One")
  items.Add("Two")
  items.Add("Three")

  MyList.DataSource = items
  MyList.DataBind()
End Sub
```

VB

Calling the **DataBind** method of a control causes a recursive tree walk from that control on down in the tree; the **DataBinding** event is raised on each server control in that hierarchy, and data binding expressions on the control are evaluated accordingly. So, if the **DataBind** method of the page is called, then every data binding expression within the page will be called.

VB Reference5.aspx

[Run Sample] | [View Source]

Object Tag Syntax: <object runat="server" />

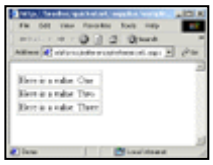
Object tags enable page developers to declare and create instances of variables using a declarative, tag-based syntax. The following example demonstrates how the object tag can be used to create an instance of an **ArrayList** class.

```
<object id="items" class="System.Collections.ArrayList" runat="server"/>
```

The object will be created automatically at run time and can then be accessed through the ID "items".

```
Sub Page_Load(sender As Object, e As EventArgs)
  items.Add("One")
  items.Add("Two")
  items.Add("Three")
  ...
End Sub
```

VB



VB Reference6.aspx

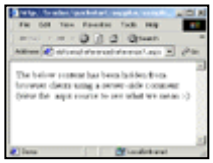
[Run Sample] | [View Source]

Server-Side Comment Syntax: <%-- Comment --%>

Server-side comments enable page developers to prevent server code (including server controls) and static content from executing or rendering. The following sample demonstrates how to block content from executing and being sent down to a client. Note that everything between <%-- and --%> is filtered out and only visible in the original server file, even though it contains other ASP.NET directives.

```
<%--
  <asp:calendar id="MyCal" runat=server/>
    <% For I=0 To 44 %>
            Hello World <br>
    <% Next %>
--%>
```

VB



VB Reference7.aspx

[Run Sample] | [View Source]

Server-Side Include Syntax: <-- #Include File="Locaton.inc" -->

Server-side #Includes enable developers to insert the raw contents of a specified file anywhere within an ASP.NET page. The following sample demonstrates how to insert a custom header and footer within a page.

```
<!-- #Include File="Header.inc" -->
...
<!-- #Include File="Footer.inc" -->
```

VB Reference8.aspx

[Run Sample] | [View Source]

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security

Introducing XML Web services

The Internet is quickly evolving from today's Web sites that just deliver user interface pages to browsers to a next generation of programmable Web sites that directly link organizations, applications, services, and devices with one another. These programmable Web sites become more than passively accessed sites - they become reusable, intelligent Web Services.

The common language runtime provides built-in support for creating and exposing Web Services, using a programming abstraction that is consistent and familiar to both ASP.NET Web Forms developers and existing Visual Basic users. The resulting model is both scalable and extensible, and embraces open Internet standards (HTTP, XML, SOAP, WSDL) so that it can be accessed and consumed from any client or Internet-enabled device.

ASP.NET Web Services

ASP.NET provides support for Web Services with the .asmx file. An .asmx file is a text file that is similar to an .aspx file. These files can be part of an ASP.NET application that includes .aspx files. These files are then URI-addressable, just as .aspx files are.

The following example shows a very simple .asmx file.

```
<%@ WebService Language="VB" Class="HelloWorld" %>

Imports System
Imports System.Web.Services

Public Class HelloWorld :Inherits WebService

     <WebMethod()> Public Function SayHelloWorld() As String
          Return("Hello World")
     End Function

End Class
```

VB

This file starts with an ASP.NET directive **WebService**, and sets the language to C#, Visual Basic, or JScript. Next, it imports the namespace **System.Web.Services**. You must include this namespace. Next, the class HelloWorld is declared. This class is derived from the base class **WebService**; note that deriving from the **WebService** base class is optional. Finally, any methods that will be accessible as part of the service have the attribute **[WebMethod]** in C#, **<WebMethod()>** in Visual Basic, or **WebMethodAttribute** in JScript, in front of their signatures.

To make this service available, we might name the file **HelloWorld.asmx** and place it on a server called **SomeDomain.com** inside a virtual directory called **someFolder**. Using a Web browser, you could then enter the URL **http://SomeDomain.com/someFolder/HelloWorld.asmx**, and the resulting page would show the public methods for this Web Service (those marked with the **WebMethod** attribute), as well as which protocols (such as SOAP, or HTTP GET) you can use to invoke these methods.

Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

Entering the address:
http://SomeDomain.com/someFolder/HelloWorld.asmx?WSDL into the browser returns a Web Service Description Language (WSDL) document. This WSDL document is very important, and is used by clients that will access the service.

Accessing Web Services

In addition to the ASP.NET server side technology that allows developers to create Web Services, the .NET Framework provides a sophisticated set of tools and code to consume Web Services. Because Web Services are based on open protocols such as the Simple Object Access Protocol (SOAP), this client technology can also be used to consume non-ASP.NET Web Services.

Within the SDK, there is a tool called the Web Services Description Language tool (WSDL.exe). This command-line tool is used to create proxy classes from WSDL. For example, you could enter:

```
WSDL http://someDomain.com/someFolder/HelloWorld.asmx?WSDL
```

to create a proxy class called HelloWorld.cs.

This class would look very similar to the class created in the previous section. It would contain a method called SayHelloWorld that returns a string. Compiling this proxy class into an application and then calling this proxy class's method results in the proxy class packaging a SOAP request across HTTP and receiving the SOAP-encoded response, which is then marshaled as a string.

From the client perspective, the code would be simple, as shown in the following example.

```
Dim myHelloWorld As New HelloWorld()
Dim sReturn As String = myHelloWorld.SayHelloWorld()
```

VB

The return would be "Hello World".

The rest of this section deals with more advanced Web Services topics, such as sending and receiving complex data types. There is also a section on Text Pattern Matching, a technology that addresses any URI that returns text as if it were a Web Service. You can also perform data binding operations with Web Services; this topic is discussed in the Data section.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Write a Simple Web Service

You can write a simple XML Web service in a few minutes using any text editor. The service you will create in this section, MathService, exposes methods for adding, subtracting, dividing, and multiplying two numbers. At the top of the page, the following directive identifies the file as a XML Web service in addition to specifying the language for the service (C#, in this case).

```
<%@ WebService Language="C#" Class="MathService" %>
```

In this same file, you define a class that encapsulates the functionality of your service. This class should be public, and can optionally inherit from the **WebService** base class. Each method that will be exposed from the service is flagged with a **[WebMethod]** attribute in front of it. Without this attribute, the method will not be exposed from the service. This is sometimes useful for hiding implementation details called by public **Web Service** methods, or in the case where the **WebService** class is also used in local applications (a local application can use any public class, but only **WebMethod** classes are remotely accessible as XML Web services).

```
Imports System
Imports System.Web.Services

Public Class MathService : Inherits WebService

   <WebMethod()> Public Function Add(a As Integer, b As Integer) As Integer
       Return(a + b)
   End Function

End Class
```

VB

XML Web service files are saved under the .asmx file extension. Like .aspx files, these are automatically compiled by the ASP.NET runtime when a request to the service is made (subsequent requests are serviced by a cached precompiled type object). In the case of MathService, you have defined the **WebService** class in the .asmx file itself. Note that if an .asmx file is requested by a browser, the ASP.NET runtime returns a XML Web service Help page that describes the Web Service.

VB MathService.asmx

[Run Sample] | [View Source]

VB MathService.asmx?wsdl

[View Sample]

Precompiled XML Web services

If you have a precompiled class that you want to expose as a XML Web service (and this class exposes methods marked with the **[WebMethod]** attribute), you can create an .asmx file with only the following line.

```
<%@ WebService Class="MyWebApplication.MyWebService" %>
```

`MyWebApplication.MyWebService` defines the **WebService** class, and is contained in the \bin subdirectory of the ASP.NET application.

Consuming a XML Web service from a Client Application

To consume this service, you need to use the Web Services Description Language command-line tool (WSDL.exe) included in the SDK to create a proxy class that is similar to the class defined in the .asmx file. (It will contain only the **WebMethod** methods.) Then, you compile your code with this proxy class included.

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

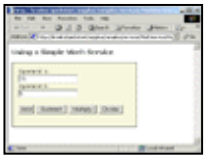
Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

WSDL.exe accepts a variety of command-line options, however to create a proxy only one option is required: the URI to the WSDL. In this example, we are passing a few extra options that specify the preferred language, namespace, and output location for the proxy. We are also compiling against a previously saved WSDL file instead of the URI to the service itself:

```
wsdl.exe /l:CS /n:MathService /out:MathService.cs MathService.wsdl
```

Once the proxy class exists, you can create objects based on it. Each method call made with the object then goes out to the URI of the XML Web service (usually as a SOAP request).



VB MathServiceClient.aspx

[Run Sample] | [View Source]

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

XML Web service Type Marshaling

This section illustrates that various data types can be passed to and returned from **Web Service** methods. Because the XML Web services implementation is built on top of the XML Serialization architecture, it supports a significant number of data types. The following table lists the supported data types for **Web Service** methods when using the SOAP protocol (for example, using the proxy generated by the Web Services Description Language tool, WSDL.exe).

Type	Description
Primitive Types	Standard primitive types. The complete list of supported primitives are String, Char, Byte, Boolean, Int16, Int32, Int64, UInt16, UInt32, UInt64, Single, Double, Guid, Decimal, DateTime (as XML's timeInstant), DateTime (as XML's date), DateTime (as XML's time), and XmlQualifiedName (as XML's QName).
Enum Types	Enumeration types, for example, "public enum color { red=1, blue=2 }"
Arrays of Primitives, Enums	Arrays of the above primitives, such as string[] and int[]
Classes and Structs	Class and struct types with public fields or properties. The public properties and fields are serialized.
Arrays of Classes (Structs)	Arrays of the above.

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

DataSet	ADO.NET DataSet Types (see the next section for an example). DataSets can also appear as fields in structs or classes. **Note:** Microsoft Visual Studio .NET and the XSD.EXE SDK utility have support for "strong-typing" a DataSet. These tools generate a class that inherits from DataSet to produce DataSet1, adding several methods/properties/etc that are specific to a particular XML schema. If you pass DataSet, XML Web services always transmits the schema along with the data (so it knows what tables and columns you are passing), and their types (for example, int, string). If you pass a subclass of DataSet (for example, DataSet1), XML Web services assumes you are adding tables/columns in the constructor, and assumes that those tables/columns represent your schema.
Arrays of DataSet	Arrays of the above.
XmlNode	XmlNode is an in-memory representation of an XML fragment (like a lightweight XML document object model). For example, "<comment>This ispretty neat</comment>" could be stored in an XmlNode. You can pass XmlNodes as parameters, and they are added to the rest of the XML being passed to the XML Web service (the other parameters) in a SOAP-compliant manner.

Get URL for this page

	The same is true for return values. This allows you to pass or return XML whose structure changes from call to call, or where you may not know all the types being passed. XmlNode can also appear as fields in structs or classes.
Arrays of XmlNode	Arrays of above.

Return values:

Whether calling a XML Web service using SOAP or HTTP GET/POST, all the above types are supported for return values.

Parameters:

Both by-value and by-reference (in/out) parameters are supported when using the SOAP protocol. By-reference parameters can send the value both ways: up to the server, and back to the client. When passing input parameters to a XML Web service using HTTP GET/POST, only a limited set of data types are supported, and they must be by-value parameters. The supported types for HTTP GET/POST parameters are listed below:

Type	Description
Primitive Types (limited)	Most standard primitive types. The complete list of supported primitives are Int32, String, Int16, Int64, Boolean, Single, Double, Decimal, DateTime, UInt16, UInt32, UInt64, and Currency. From the client's perspective, all these types turn into string.

Enum Types	Enumeration types, for example, "public enum color { red=1, blue=2 }". From the client's perspective, enums become classes with a static const string for each value.
Arrays of Primitives, Enums	Arrays of the above primitives, such as string[] and int[]

The following example demonstrates the use of the types listed above, using a SOAP proxy generated from WSDL.exe. Note that because there is more than one public class defined in the .asmx file, you must specify which is to be treated as the **WebService** class using the "Class" attribute of the **WebService** directive:

```
<%@ WebService Language="C#" Class="DataTypes" %>
```

VB DataTypes.asmx

[Run Sample] | [View Source]

VB DataTypes.asmx?wsdl

[View Sample]

- The **SayHello** method shows returning a String from a service.
- The **SayHelloName** method returns a String, and also takes a String as a parameter.
- The **GetIntArray** method shows how to return an array of integers.
- The **GetMode** method returns an enum value.
- The **GetOrder** method returns a class (which is almost the same as a struct here).
- The **GetOrders** method returns an array of **Order** objects.

Using the WSDL.exe command line proxy generation tool, the marshaling of these data types is transparent to the consuming client application. A sample client application for the above XML Web service follows:

VB DataTypesClient.aspx

[Run Sample] | [View Source]

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

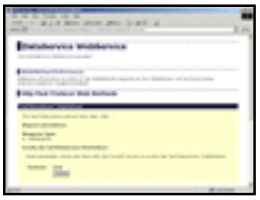
Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Use Data in XML Web services

This sample shows how DataSets, a powerful new XML-based way to represent disconnected data, can be returned from a **Web Service** method. This is an extremely powerful use of XML Web services, as DataSets can store complex information and relationships in an intelligent structure. By exposing DataSets through a service, you can limit the database connections your data server is experiencing.

The method **GetTitleAuthors** connects to a database and issues two SQL statements: one that returns a list of authors, and another that returns a list of book titles. It places both result sets into a single DataSet called ds, and returns this DataSet.

The method **PutTitleAuthors** illustrates a **Web Service** method that takes a DataSet as a parameter, returning an integer that represents the number of rows received in the "Authors" table of the DataSet. Although the implementation of this method is somewhat simplistic, this method could also intelligently merge the passed data with the database server.



VB DataService.asmx

[Run Sample] | [View Source]

VB DataService.asmx?wsdl

[View Sample]

The client application for this XML Web service calls GetTitleAuthors and binds the Authors table to a DataGrid control, as you've seen in previous examples. To illustrate the **PutTitleAuthors** method, the client removes three rows of data from the DataSet before calling this method, printing out the number of rows received by the service.

VB DataServiceClient.aspx

[Run Sample] | [View Source]

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Use Objects and Intrinsics

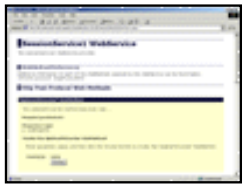
This sample illustrates how to access ASP.NET intrinsics such as the **Session** and **Application** objects. It also shows how to turn off **Session** on a per-[WebMethod] basis.

The first method in the sample .asmx file, **UpdateHitCounter**, accesses the **Session** and adds 1 to the "HitCounter" value. It then returns this value as a String. The second method, **UpdateAppCounter** does the same thing, but with the **Application**. Notice the following:

```
<WebMethod(EnableSession:=true)>
```

VB

Session state for XML Web services is disabled by default, and you have to use a special attribute property to enable **Sessions**. However, **Sessions** aren't needed for this object, since it only uses the **Application** object.



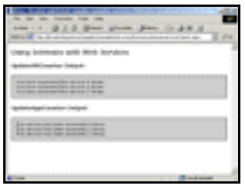
VB SessionService.asmx

[Run Sample] | [View Source]

VB SessionService.asmx?wsdl

[View Sample]

When the client proxy is accessed, it contains a cookie collection. This collection is used to accept and return the APSESSIONID cookie that ASP.NET uses to track Sessions. This is what allows this client to receive varying answers to the **Session** hit method.



VB SessionServiceClient.aspx

[Run Sample] | [View Source]

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization

The WebService Behavior

Microsoft recently released a new SOAP-enabled DHTML behavior for Microsoft Internet Explorer 5.0 and later versions. The new **WebService** behavior enables client-side script to invoke remote methods exposed by Microsoft .NET XML Web services, or other Web servers that support the Simple Object Access Protocol (SOAP). The **WebService** behavior is implemented with an HTML Components (HTC) file as an attached behavior, so it can be used in Internet Explorer.

The purpose of the **WebService** behavior is to provide a simple way of using and leveraging SOAP, without requiring expert knowledge of its implementation. The **WebService** behavior supports the use of a wide variety of data types, including intrinsic SOAP data types, arrays, and Extensible Markup Language (XML) data. This flexible component enables Internet Explorer to retrieve information from XML Web services and to update a page dynamically using DHTML and script, without requiring navigation or a full page refresh.

The next generation of .NET development tools and infrastructure, including Visual Studio .NET, the .NET Framework, and the .NET Enterprise Servers, are designed for the development of applications based on the XML Web services model. The **WebService** behavior is particularly significant because it enables Internet Explorer to use these next-generation XML Web services.

The Microsoft Developer Network (MSDN) site provides the following documentation.

WebService Behavior Overview	http://msdn.microsoft.com/workshop/author/webservice/overview.asp
Using the WebService Behavior	http://msdn.microsoft.com/workshop/author/webservice/using.asp
The WebService Behavior	http://msdn.microsoft.com/workshop/author/webservice/webservice.asp

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

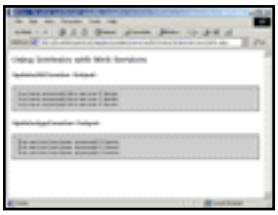
Configuration
Configuration Overview

HTML Text Pattern Matching

This example demonstrates how to create a client proxy for any URI that serves up text. Instead of authoring the .asmx file, you can create a WSDL file that describes an HTML (or XML or any other nonbinary format) page you currently offer. The WSDL can be used to generate a client proxy, using the WSDL.exe command line tool that will use RegEx to parse the named HTML page and extract values.

You can do this by adding <Match> tags in the Response section of the WSDL. These tags take an attribute called **pattern**, which is the Regular Expression that corresponds to the piece of text on the page that is the property's value. (Note: the property from the proxy class is read-only.)

The consuming code can then create the object, access the **Matches** object that is returned by the functioned name in the WSDL, and gain access to any piece of the HTML as a property. No understanding of WSDL, regular expressions, or even HTML is needed to use the proxy class. It behaves like any other .NET Framework class would.



VB MatchClient.aspx

[Run Sample] | [View Source]

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication

Application Overview

- What is an ASP.NET Application?
- Creating an Application
- Lifetime of an Application
- A Note on Multiple Threads
- Section Summary

What is an ASP.NET Application?

ASP.NET defines an application as the sum of all files, pages, handlers, modules, and executable code that can be invoked or run in the scope of a given virtual directory (and its subdirectories) on a Web application server. For example, an "order" application might be published in the "/order" virtual directory on a Web server computer. For IIS the virtual directory can be set up in the Internet Services Manager; it contains all subdirectories, unless the subdirectories are virtual directories themselves.

Each ASP.NET Framework application on a Web server is executed within a unique .NET Framework application domain, which guarantees class isolation (no versioning or naming conflicts), security sandboxing (preventing access to certain machine or network resources), and static variable isolation.

ASP.NET maintains a pool of **HttpApplication** instances over the course of a Web application's lifetime. ASP.NET automatically assigns one of these instances to process each incoming HTTP request that is received by the application. The particular **HttpApplication** instance assigned is responsible for managing the entire lifetime of the request and is reused only after the request has been completed. This means that user code within the **HttpApplication** does not need to be reentrant.

Creating an Application

To create an ASP.NET Framework application you can use an existing virtual directory or create a new one. For example, if you installed Windows 2000 Server including IIS, you probably have a directory C:\InetPub\WWWRoot. You can configure IIS using the Internet Services Manager, available under Start -> Programs -> Administrative Tools. Right-click on an existing directory and choose either New (to create a new virtual directory) or Properties (to promote an existing regular directory).

By placing a simple .aspx page like the following in the virtual directory and accessing it with the browser, you trigger the creation of the ASP.NET application.

```
<%@Page Language="VB"%>
<html>
<body>
<h1>hello world, <% Response.Write(DateTime.Now.ToString()) %></h1>
</body>
</html>
```

VB

Now you can add appropriate code to use the Application object--to store objects with application scope, for example. By creating a global.asax file you also can define various event handlers-- for the **Application_Start** event, for example.

Lifetime of an Application

Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

An ASP.NET Framework application is created the first time a request is made to the server; before that, no ASP.NET code executes. When the first request is made, a pool of **HttpApplication** instances is created and the **Application_Start** event is raised. The **HttpApplication** instances process this and subsequent requests, until the last instance exits and the **Application_End** event is raised.

Note that the **Init** and **Dispose** methods of **HttpApplication** are called per instance and thus can be called several times between **Application_Start** and **Application_End**. Only these events are shared among all instances of **HttpApplication** in one ASP.NET application.

A Note on Multiple Threads

If you use objects with application scope, you should be aware that ASP.NET processes requests concurrently and that the **Application** object can be accessed by multiple threads. Therefore the following code is dangerous and might not produce the expected result, if the page is repeatedly requested by different clients at the same time.

```
<%
Application("counter") = CType(Application("counter") + 1, Int32)
%>
```

VB

To make this code thread safe, serialize the access to the **Application** object using the **Lock** and **UnLock** methods. However, doing so also means accepting a considerable performance hit:

```
<%
Application.Lock()
Application("counter") = CType(Application("counter") + 1, Int32)
Application.UnLock()
%>
```

VB

Another solution is to make the object stored with an application scope thread safe. For example, note that the collection classes in the **System.Collections** namespace are not thread safe for performance reasons.

Section Summary

1. ASP.NET Framework applications consist of everything under one virtual directory of the Web server.
2. You create an ASP.NET Framework application by adding files to a virtual directory on the Web server.
3. The lifetime of an ASP.NET Framework application is marked by **Application_Start** and **Application_End** events.
4. Access to application-scope objects must be safe for multithreaded access.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Using the Global.asax File

- The Global.asax File
- Application or Session-Scoped Events
- Application or Session-Scoped Objects
- Section Summary

The Global.asax File

In addition to writing UI code, developers can also add application level logic and event handling code into their Web applications. This code does not handle generating UI and is typically not invoked in response to individual page requests. Instead, it is responsible for handling higher-level application events such as **Application_Start**, **Application_End**, **Session_Start**, **Session_End**, and so on. Developers author this logic using a **Global.asax** file located at the root of a particular Web application's virtual directory tree. ASP.NET automatically parses and compiles this file into a dynamic .NET Framework class--which extends the **HttpApplication** base class--the first time any resource or URL within the application namespace is activated or requested.

The Global.asax file is parsed and dynamically compiled by ASP.NET into a .NET Framework class the first time any resource or URL within its application namespace is activated or requested. The Global.asax file is configured to automatically reject any direct URL request so that external users cannot download or view the code within.

Application or Session-Scoped Events

Developers can define handlers for events of the **HttpApplication** base class by authoring methods in the Global.asax file that conform to the naming pattern "Application_EventName(AppropriateEventArgumentSignature)". For example:

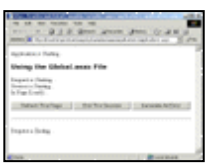
```
<script language="VB" runat="server">

Sub Application_Start(Sender As Object, E As EventArgs)
  ' Application startup code goes here
End Sub
</script>
```

VB

If the event handling code needs to import additional namespaces, the **@ import** directive can be used on an .aspx page, as follows:

```
<%@ Import Namespace="System.Text" %>
```

The following sample illustrates the lifetime of **Application**, **Session**, and **Request**.



VB Application1.aspx

[Run Sample] | [View Source]

The first time the page is opened, the **Start** event is raised for the application and the session:

```
Sub Application_Start(Sender As Object, E As EventArgs)
  ' Application startup code goes here
End Sub
```

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

```
Sub Session_Start(Sender As Object, E As EventArgs)
  Response.Write("Session is Starting...<br>")
  Session.Timeout = 1
End Sub
```

VB

The **BeginRequest** and **EndRequest** events are raised on each request. When the page is refreshed, only messages from **BeginRequest**, **EndRequest**, and the **Page_Load** method will appear. Note that by abandoning the current session (click the "End this session" button) a new session is created and the **Session_Start** event is raised again.

Application or Session-Scoped Objects

Static objects, .NET Framework classes, and COM components all can be defined in the Global.asax file using the object tag. The scope can be **appinstance**, **session**, or **application**. The **appinstance** scope denotes that the object is specific to one instance of **HttpApplication** and is not shared.

```
<object id="id" runat="server" class=".NET Framework class Name"
scope="appinstance"/>
<object id="id" runat="server" progid="COM ProgID" scope="session"/>
<object id="id" runat="server" classid="COM ClassID" scope="application"/>
```

Section Summary

1. ASP.NET Framework applications can define event handlers with application-wide or session-wide scope in the Global.asax file.
2. ASP.NET Framework applications can define objects with application-wide or session-wide scope in the Global.asax file.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

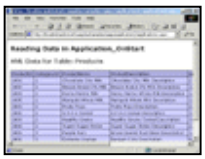
Localization

Managing Application State

- Using Application State
- Using Session State
- Using Client-Side Cookies
- Using ViewState
- Section Summary

Using Application State

This sample illustrates the use of application state to read a dataset in **Application_Start**.



VB Application2.aspx

[Run Sample] | [View Source]

Because an application and all the objects it stores can be concurrently accessed by different threads, it is better to store only infrequently modified data with application scope. Ideally an object is initialized in the **Application_Start** event and further access is read-only.

In the following sample a file is read in **Application_Start** (defined in the Global.asax file) and the content is stored in a **DataView** object in the application state.

```
Sub Application_Start()
    Dim ds As New DataSet()

    Dim fs As New
FileStream(Server.MapPath("schemadata.xml"),FileMode.Open,FileAccess.Read)
    Dim reader As New StreamReader(fs)
    ds.ReadXml(reader)
    fs.Close()

    Dim view As New DataView (ds.Tables(0))
    Application("Source") = view
End Sub
```

VB

In the **Page_Load** method, the **DataView** is then retrieved and used to populate a **DataGrid** object:

```
Sub Page_Load(sender As Object, e As EventArgs)
    Dim Source As New DataView  = CType(Application("Source"), DataView)
    ...
    MyDataGrid.DataSource = Source
    ...
End Sub
```

VB

The advantage of this solution is that only the first request pays the price of retrieving the data. All subsequent requests use the already existing **DataView** object. As the data is never modified after initialization, you do not have to make any provisions for serializing access.

Using Session State

Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

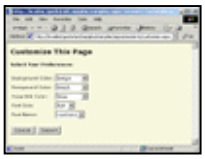
Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

The following sample illustrates the use of session state to store volatile user preferences.



VB Session1.aspx

[Run Sample] | [View Source]

To provide individual data for a user during a session, data can be stored with session scope. In the following sample, values for user preferences are initialized in the **Session_Start** event in the Global.asax file.

```
Sub Session_Start()
    Session("BackColor") = "beige"
    ...
End Sub
```

VB

In the following customization page, values for user preferences are modified in the **Submit_Click** event handler according to user input.

```
Protected Sub Submit_Click(sender As Object, e As EventArgs)
    Session("BackColor") = BackColor.Value
    ...

    Response.Redirect(State("Referer").ToString())
End Sub
```

VB

The individual values are retrieved using the **GetStyle** method:

```
Protected GetStyle(key As String) As String
    Return(Session(key).ToString())
End Sub
```

VB

The **GetStyle method** is used to construct session-specific styles:

```
<style>
    body
    {
      font: <%=GetStyle("FontSize")%> <%=GetStyle("FontName")%>;
      background-color: <%=GetStyle("BackColor")%>;
    }
    a
    {
        color: <%=GetStyle("LinkColor")%>
    }
</style>
```

To verify that the values are really stored with session scope, open the sample page twice, then change one value in the first browser window and refresh the second one. The second window picks up the changes because both browser instances share a common **Session** object.

Configuring session state: Session state features can be configured via the **<sessionState>** section in a web.config file. To double the default timeout of 20 minutes, you can add the following to the web.config file of an application:

```
<sessionState timeout="40" />
```

By default, ASP.NET will store the session state in the same process that processes the request, just as ASP does. If cookies are not available, a session can be tracked by adding a session identifier to the URL. This can be enabled by setting the following:

```
<sessionState
  cookieless="true"
/>
```

By default, ASP.NET will store the session state in the same process that processes the request, just as ASP does. Additionally, ASP.NET can store session data in an external process, which can even reside on another machine. To enable this feature:

- Start the ASP.NET state service, either using the Services snap-in or by executing "net start aspnet_state" on the command line. The state service will by default listen on port 42424. To change the port, modify the registry key for the service: HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\aspnet_state\Parameters\Port
- Set the **mode** attribute of the **<sessionState>** section to "StateServer".
- Configure the **stateConnectionString** attribute with the values of the machine on which you started aspnet_state.

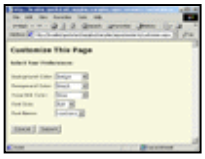
The following sample assumes that the state service is running on the same machine as the Web server ("localhost") and uses the default port (42424):

```
<sessionState
  mode="StateServer"
  stateConnectionString="tcpip=localhost:42424"
/>
```

Note that if you try the sample above with this setting, you can reset the Web server (enter `iisreset` on the command line) and the session state value will persist.

Using Client-Side Cookies

The following sample illustrates the use of client-side cookies to store volatile user preferences.



VB Cookies1.aspx

[Run Sample] | [View Source]

Storing cookies on the client is one of the methods that ASP.NET's session state uses to associate requests with sessions. Cookies can also be used directly to persist data between requests, but the data is then stored on the client and sent to the server with every request. Browsers place limits on the size of a cookie; therefore, only a maximum of 4096 bytes is guaranteed to be acceptable.

When the data is stored on the client, the **Page_Load** method in the file cookies1.aspx checks whether the client has sent a cookie. If not, a new cookie is created and initialized and stored on the client:

```
Protected Sub Page_Load(sender As Object, e As EventArgs)
    If Request.Cookies("preferences1") = Null Then
        Dim cookie As New HttpCookie("preferences1")
        cookie.Values.Add("ForeColor", "black")
        ...
        Response.AppendCookie(cookie)
    End If
```

```
End Sub
```

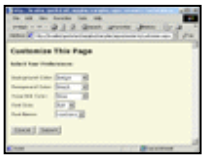
VB

On the same page, a **GetStyle** method is used again to provide the individual values stored in the cookie:

```
Protected Function GetStyle(key As String) As String
  Dim cookie As HttpCookie = Request.Cookies("preferences1")
  If cookie <> Null Then
    Select Case key
      Case "ForeColor"
        Return(cookie.Values("ForeColor"))
      Case ...
    End Select
  End If
  Return("")
End Function
```

VB

Verify that the sample works by opening the cookies1.aspx page and modifying the preferences. Open the page in another window, it should pick up the new preferences. Close all browser windows and open the cookies1.aspx page again. This should delete the temporary cookie and restore the default preference values.



VB Cookies2.aspx

[Run Sample] | [View Source]

To make a cookie persistent between sessions, the **Expires** property on the **HttpCookie** class has to be set to a date in the future. The following code on the customization.aspx page is identical to the previous sample, with the exception of the assignment to **Cookie.Expires**:

```
Protected Sub Submit_Click(sender As Object, e As EventArgs)
    Dim cookie As New HttpCookie("preferences2")
    cookie.Values.Add("ForeColor",ForeColor.Value)
    ...
    cookie.Expires = DateTime.MaxValue ' Never Expires

    Response.AppendCookie(cookie)

    Response.Redirect(State("Referer").ToString())
End Sub
```

VB

Verify that the sample is working by modifying a value, closing all browser windows, and opening cookies2.aspx again. The window should still show the customized value.

Using ViewState

This sample illustrates the use of the **ViewState** property to store request-specific values.

VB PageState1.aspx

[Run Sample] | [View Source]

ASP.NET provides the server-side notion of a view state for each control. A control can save its internal state between requests using the **ViewState** property on an instance of the class **StateBag**. The **StateBag** class provides a dictionary-like interface to store objects associated with a string key.

The file pagestate1.aspx displays one visible panel and stores the index of it in the view state of the page with the key **PanelIndex**:

```
Protected Sub Next_Click(sender As Object, e As EventArgs)
    Dim PrevPanelId As String = "Panel" + ViewState("PanelIndex").ToString()
    ViewState("PanelIndex") = CType(ViewState("PanelIndex") + 1, Integer)
    Dim PanelId As String = "Panel" + ViewState("PanelIndex").ToString()
    ...
End Sub
```

VB

Note that if you open the page in several browser windows, each browser window will initially show the name panel. Each window can independently navigate between the panels.

Section Summary

1. Use application state variables to store data that is modified infrequently but used often.
2. Use session state variables to store data that is specific to one session or user. The data is stored entirely on the server. Use it for short-lived, bulky, or sensitive data.
3. Store small amounts of volatile data in a nonpersistent cookie. The data is stored on the client, sent to the server on each request, and expires when the client ends execution.
4. Store small amounts of non-volatile data in a persistent cookie. The data is stored on the client until it expires and is sent to the server on each request.
5. Store small amounts of request-specific data in the view state. The data is sent from the server to the client and back.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

HTTP Handlers and Factories

- Overview
- Configuring HTTP Handlers and Factories
- Creating a Custom HTTP Handler
- Section Summary

Overview

ASP.NET provides a low-level request/response API that enables developers to use .NET Framework classes to service incoming HTTP requests. Developers accomplish this by authoring classes that support the **System.Web.IHTTPHandler** interface and implement the **ProcessRequest()** method. Handlers are often useful when the services provided by the high-level page framework abstraction are not required for processing the HTTP request. Common uses of handlers include filters and CGI-like applications, especially those that return binary data.

Each incoming HTTP request received by ASP.NET is ultimately processed by a specific instance of a class that implements **IHTTPHandler**. **IHttpHandlerFactory** provides the infrastructure that handles the actual resolution of URL requests to **IHttpHandler** instances. In addition to the default **IHttpHandlerFactory** classes provided by ASP.NET, developers can optionally create and register factories to support rich request resolution and activation scenarios.

Configuring HTTP Handlers and Factories

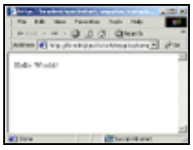
HTTP handlers and factories are declared in the ASP.NET configuration as part of a web.config file. ASP.NET defines an **<httphandlers>** configuration section where handlers and factories can be added and removed. Settings for **HttpHandlerFactory** and **HttpHandler** are inherited by subdirectories.

For example, ASP.NET maps all requests for .aspx files to the **PageHandlerFactory** class in the global machine.config file:

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

Creating a Custom HTTP Handler

The following sample creates a custom **HttpHandler** that handles all requests to "SimpleHandler.aspx".



VB SimpleHandler

[Run Sample] | [View Source]

A custom HTTP handler can be created by implementing the **IHttpHandler** interface, which contains only two methods. By calling **IsReusable**, an HTTP factory can query a handler to determine whether the same instance can be used to service multiple requests. The **ProcessRequest** method takes an **HttpContext** instance as a parameter, which gives it access to the **Request** and **Response** intrinsics. In the following sample, request data is ignored and a constant string is sent as a response to the client.

```
Public Class SimpleHandler : Inherits IHttpHandler
  Public Sub ProcessRequest(context As HttpContext)
    context.Response.Write("Hello World!")
  End Sub

  Public Function IsReusable() As Boolean
    Return(True)
  End Function
End Class
```

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

VB

After placing the compiled handler assembly in the application's \bin directory, the handler class can be specified as a target for requests. In this case, all requests for "SimpleHandler.aspx" will be routed to an instance of the **SimpleHandler** class, which lives in the namespace **Acme.SimpleHandler**.

```
<httphandlers> <add verb="*" path="SimpleHandler.aspx"
type="Acme.SimpleHandler,SimpleHandler" /> </httphandlers>
```

Section Summary

1. HTTP Handlers and factories are the backbone of the ASP.NET page framework.
2. Factories assign each request to one handler, which processes the request.
3. Factories and handlers are defined in the web.config file. Settings for factories are inherited by subdirectories.
4. To create a custom handler, implement **IHttpHandler** and add the class in the **<httphandlers>** section of the web.config in the directory.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview

Caching Overview

Caching is a technique widely used in computing to increase performance by keeping frequently accessed or expensive data in memory. In the context of a Web application, caching is used to retain pages or data across HTTP requests and reuse them without the expense of recreating them.

ASP.NET has three kinds of caching that can be used by Web applications:

- Output caching, which caches the dynamic response generated by a request.
- Fragment caching, which caches portions of a response generated by a request.
- Data caching, which caches arbitrary objects programmatically. To support this, ASP.NET provides a full-featured cache engine that allows programmers to easily retain data across requests.

Output caching is useful when the contents of an entire page can be cached. On a heavily accessed site, caching frequently accessed pages for even a minute at a time can result in substantial throughput gains. While a page is cached by the output cache, subsequent requests for that page are served from the output page without executing the code that created it.

Sometimes it is not practical to cache an entire page - perhaps portions of the page must be created or customized for each request. In this case, it is often worthwhile to identify objects or data that are expensive to construct and are eligible for caching. Once these items are identified, they can be created once and then cached for some period of time. Additionally, fragment caching can be used to cache regions of a page's output.

Choosing the time to cache an item can be an interesting decision. For some items, the data might be refreshed at regular intervals or the data is valid for a certain amount of time. In that case, the cache items can be given an expiration policy that causes them to be removed from the cache when they have expired. Code that accesses the cache item simply checks for the absence of the item and recreates it, if necessary.

Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications

The ASP.NET cache supports file and cache key dependencies, allowing developers to make a cache item dependent on an external file or another cache item. This technique can be used to invalidate items when their underlying data source changes.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model

Page Output Caching

Output caching is a powerful technique that increases request/response throughput by caching the content generated from dynamic pages. Output caching is enabled by default, but output from any given response is not cached unless explicit action is taken to make the response cacheable.

To make a response eligible for output caching, it must have a valid expiration/validation policy and public cache visibility. This can be done using either the low-level **OutputCache** API or the high-level **@ OutputCache** directive. When output caching is enabled, an output cache entry is created on the first **GET** request to the page. Subsequent **GET** or **HEAD** requests are served from the output cache entry until the cached request expires.

The output cache also supports variations of cached **GET** or **POST** name/value pairs.

The output cache respects the expiration and validation policies for pages. If a page is in the output cache and has been marked with an expiration policy that indicates that the page expires 60 minutes from the time it is cached, the page is removed from the output cache after 60 minutes. If another request is received after that time, the page code is executed and the page can be cached again. This type of expiration policy is called absolute expiration - a page is valid until a certain time.

The following example demonstrates a simple way to output cache responses using the **@ OutputCache** directive. The example simply displays the time when the response was generated. To see output caching in action, invoke the page and note the time at which the response was generated. Then refresh the page and note that the time has not changed, indicating that the second response is being served from the output cache.

VB Outputcache1.aspx

[Run Sample] | [View Source]

The following directive activates output caching on the response:

```
<%@ OutputCache Duration="60" VaryByParam="none"%>
```

This directive simply indicates that the page should be cached for 60 seconds and that the page does not vary by any **GET** or **POST** parameters. Requests received while the page is still cached are satisfied from the cache. After 60 seconds, the page is removed from the cache; the next request is handled explicitly and caches the page again.

Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

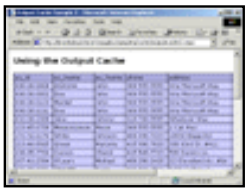
Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

Of course, in the previous example, very little work is saved by output caching. The following example shows the same technique for output caching, but queries a database and displays the results in a grid.



VB Outputcache2.aspx

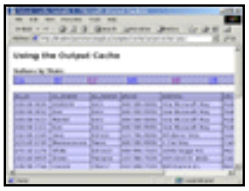
[Run Sample] | [View Source]

In the final example, the application is modified slightly to allow the user to selectively query for authors in various states. This example demonstrates caching requests varying by the name/value pairs in the query string using the **VaryByParam** attribute of the **@ OutputCache** directive.

```
<%@ OutputCache Duration="60" VaryByParam="state" %>
```

For each state in the data set, there is a link that passes the desired state as part of the query string. The application then constructs the appropriate database query and shows authors belonging only to the selected state.

Note that the first time you click the link for a given state, it generates a new timestamp at the bottom of the page. Thereafter, whenever a request for that state is resubmitted within a minute, you get the original timestamp indicating that the request has been cached.



VB Outputcache3.aspx

[Run Sample] | [View Source]

Applications that want more control over the HTTP headers related to caching can use the functionality provided by the **System.Web.HttpCachePolicy** class. The following example shows the code equivalent to the page directives used in the previous samples.

```
Response.Cache.SetExpires(DateTime.Now.AddSeconds(60))
Response.Cache.SetCacheability(HttpCacheability.Public)
```

VB

To make this a sliding expiration policy, where the expiration time out resets each time the page is requested, set the **SlidingExpiration** property as shown in the following code.

```
Response.Cache.SetExpires(DateTime.Now.AddSeconds(60))
Response.Cache.SetCacheability(HttpCacheability.Public)
Response.Cache.SetSlidingExpiration(True)
```

VB

Note: When sliding expiration is enabled (**SetSlidingExpiration(true)**), a request made to the origin server always generates a response. Sliding expiration is useful in scenarios where there are downstream caches that can satisfy client requests, if the content has not expired yet, without requesting the content from the origin server.

Applications being ported from ASP may already be setting cache policy using the ASP properties; for example:

```
Response.CacheControl = "Public"
Response.Expires = 60
```

VB

These properties are supported by ASP.NET and have the same effect as the other examples that have been shown.

Section Summary

1. Output caching caches the content generated by ASP.NET pages.
2. Pages are not placed in the output cache unless they have a valid expiration or validation policy and public cache visibility.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview

Page Fragment Caching

In addition to output caching an entire page, ASP.NET provides a simple way for you to output cache regions of page content, which is appropriately named fragment caching. You delineate regions of your page with a user control, and mark them for caching using the **@ OutputCache** directive introduced in the previous section. This directive specifies the duration (in seconds) that the output content of the user control should be cached on the server, as well as any optional conditions by which it should be varied.

For example, the following directive instructs ASP.NET to output cache the user control for 120 seconds, and to vary the caching using the "CategoryID" and "SelectedID" querystring or form post parameters.

```
<%@ OutputCache Duration="120" VaryByParam="CategoryID;SelectedID"%>
```

The **VaryByParam** attribute is extremely powerful and allows user control authors to instruct ASP.NET to cache/store multiple instances of an output cache region on the server. For example, the following URLs to the host page of the previous user control cache separate instances of the user control content.

http://localhost/mypage.aspx?categoryid=foo&selectedid=0
http://localhost/mypage.aspx?categoryid=foo&selectedid=1

Logic within a user control can then dynamically generate different content (which is cached separately) depending on the arguments provided.

In addition to supporting the **VaryByParam** attribute, fragment caching also supports a **VaryByControl** attribute. Whereas the **VaryByParam** attribute varies cached results based on name/value pairs sent using **POST** or **GET**, the **VaryByControl** attribute varies the cached fragment by controls within the user control. For example:

```
<%@ OutputCache Duration="120" VaryByParam="none" VaryByControl="Category" %>
```

Note that similar to output-cached pages, explict use of **VaryByParam** is required even if it is not used.

If the user control contained a drop-down select box control named Category, the user control's output would vary based on the selected value within that control.

Just as it is possible to nest user controls recursively within a page (that is, a user control declared within another server control), it is also possible to nest output-cached user controls recursively. This provides a powerful composition model that enables cached regions to be composed of further subcached regions.

The following sample code demonstrates how to cache two menu sections of a page using a declarative user control.

```
<%@ Register TagPrefix="Acme" TagName="Menu" Src="Menu.ascx" %>

<html>
  <body>
    <table>
      <tr>
        <td>
          <Acme:Menu Category="LeftMenu" runat=server/>
        </td>
        <td>
          <h1>Hi, the time is now: <%=Now%> </h1>
        </td>
        <td>
          <Acme:Menu Category="RightMenu" runat=server/>
        </td>
      <tr>
    </table>
  </body>
</html>
```

VB

The following sample code shows the implementation of the "Acme:Menu" user control with caching support.

Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

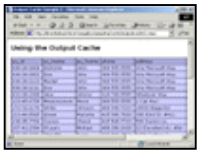
Get URL for this page

```
<%@ OutputCache Duration="120" VaryByParam="none" %>

<script language="VB" runat=server>

    Public Category As String;

    Sub Page_Load(sender As Object, e As EventArgs)

        Dim conn As AdoConnection = New AdoConnection("MyDSN")

        MyMenu.DataSource = conn.Execute("select * from menu where category=" &
Category)
        MyMenu.DataBind()
    End Sub

</script>

<asp:datagrid id="MyMenu" runat=server/>
```

VB

Note that this example output caches the response of each user control for a period of 120 seconds. All logic necessary to recreate each menu user control in the event of a cache miss (either because 120 seconds has expired or because memory conditions on the server have become scarce) is encapsulated cleanly within the user control.

The following example shows simple fragment caching. The sample caches the output from a control that retrieves data from an SQL Server database, while keeping the dynamic properties of the parent page. You can see that the page is dynamic because the time is updated with every refresh, while the control is only updated every 60 seconds.



VB FragmentCache1.aspx

[Run Sample] | [View Source]

Caveats

Note: Attempts to programmatically manipulate an output-cached control from its containing page result in an error. For example, attempts to use a declarative data binding expression on the user control tag generates parser errors, as shown in the following code.

```
<!-- The following tags generate parser errors. -->
<Acme:Menu Category='<%# Container.DataItem("Category")'  runat="server"/>
```

The reason for this is simple. In cases when the content of a user control is output cached, an instance of the control is created only on the first request; thus, once cached, the control is no longer available. Instead, you should encapsulate all the logic necessary to create the content of a user control directly within the control itself; this is typically done within the user control's **Page_Load** event or **Page_PreRender** event.

You can declare and use other declarative property parameters to customize the control. For example, the previous user control can be customized as follows:

```
<Acme:Menu Category="LeftMenu" runat=server/>
<Acme:Menu Category="RightMenu" runat=server/>
```

These declarations cause the appropriate code to be generated and executed by the page compiler in the event that the control is created as a result of a cache miss. User control developers can then access these settings just as they would in a non-cached user control scenario.

Section Summary

1. In addition to output caching an entire page, ASP.NET provides a simple way for you to output cache regions of page content, which is appropriately named fragment caching.
2. You delineate regions of your page with a user control and mark them for caching using the **@ OutputCache** directive introduced in the previous section.
3. Just as it is possible to nest user controls recursively within a page (that is, a user control declared within another server control), it is also possible to nest output-cached user controls recursively.
4. Attempts to programmatically manipulate an output-cached control from its containing page result in an error. Instead, you should encapsulate all the logic necessary to create the content of a user control directly within the control itself, typically within the user control's **Page_Load** event or **Page_PreRender** event.
5. It is possible to declare and use other declarative property parameters to customize the control.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security

Page Data Caching

- Introduction to Data Caching
- Using the Data Cache
- Section Summary

Introduction to Data Caching

ASP.NET provides a full-featured cache engine that can be used by pages to store and retrieve arbitrary objects across HTTP requests. The ASP.NET cache is private to each application and stores objects in memory. The lifetime of the cache is equivalent to the lifetime of the application; that is, when the application is restarted, the cache is recreated.

The cache provides a simple dictionary interface that allows programmers to easily place objects in and retrieve them from the cache. In the simplest case, placing an item in the cache is just like adding an item to a dictionary:

```
Cache("mykey") = myValue
```

VB

Retrieving the data is just as simple:

```
myValue = Cache("mykey")
If myValue <> Null Then
    DisplayData(myValue)
End If
```

VB

For applications that need more sophisticated functionality, the ASP.NET cache supports scavenging, expiration, and file and key dependencies.

- Scavenging means that the cache attempts to remove infrequently used or unimportant items if memory becomes scarce. Programmers who want to control how scavenging occurs can provide hints to the scavenger when items are inserted into the cache that indicate the relative cost of creating the item and the relative rate at which the item must be accessed to remain useful.

- Expiration allows programmers to give cache items lifetimes, which can be explicit (for example, expire at 6:00) or can be relative to an item's last use (for example, expire 20 minutes after the item was last accessed). After an item has expired, it is removed from the cache and future attempts to retrieve it return the null value unless the item is reinserted into the cache.

- File and key dependencies allow the validity of a cache item to be based on an external file or on another cache item. If a dependency changes, the cache item is invalidated and removed from the cache. For an example of how you might use this functionality, consider the following

Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

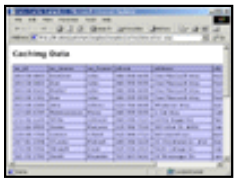
Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

scenario: an application reads financial information from an XML file that is periodically updated. The application processes the data in the file and creates a graph of objects that represent that data in a consumable format. The application caches that data and inserts a dependency on the file from which the data was read. When the file is updated, the data is removed from the cache and the application can reread it and reinsert the updated copy of the data.

Using the Data Cache

The following sample shows a simple use of the cache. It executes a database query and caches the result, which it continues to use for the lifetime of the application. When you run the sample, note the message at the bottom of the page. When first requested, it indicates that the data was explicitly retrieved from the database server. After refreshing the page, the page notes that the cached copy was used.

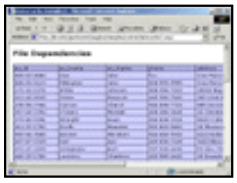


VB Datacache1.aspx

[Run Sample] | [View Source]

The next example shows a cache item that depends on an XML file. It is similar to the first example, although in this case the data is retrieved from an XML data source instead of a database server. When the data is cached, the XML file is added as a dependency.

When a new record is added using the form at the bottom of the page, the XML file is updated and the cached item must be recreated.



VB Datacache2.aspx

[Run Sample] | [View Source]

Note that a file dependency is added by using **Cache.Insert** and supplying a **CacheDependency** object referencing the XML file.

```
Cache.Insert("MyData", Source, _
        New CacheDependency(Server.MapPath("authors.xml")))
```

VB

A cache item can depend on a single or multiple files or keys. As mentioned previously, an application can also set expiration policy on a cache item. The

following code sets an absolute cache expiration time.

```
Cache.Insert("MyData", Source, null, _
             DateTime.Now.AddHours(1), TimeSpan.Zero)
```

VB

The relevant parameter is the call to **DateTime.Now.AddHours(1)**, which indicates that the item expires 1 hour from the time it is inserted. The final argument, **TimeSpan.Zero** indicates that there is no relative expiration policy on this item.

The following code shows how to set a relative expiration policy. It inserts an item that expires 20 minutes after it is last accessed. Note the use of **DateTime.MaxValue**, which indicates that there is no absolute expiration policy on this item.

```
Cache.Insert("MyData", Source, null, DateTime.MaxValue, _
             TimeSpan.FromMinutes(20))
```

VB

Section Summary

1. Data caching allows arbitrary objects to be cached programmatically.
2. The ASP.NET cache supports expiration and dependencies.
3. The cache is scoped to an application and its lifetime is equivalent to the lifetime of the application.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview

Configuration Overview

A central requirement of any Web application server is a rich and flexible configuration system--one that enables developers to easily associate settings with an installable application (without having to "bake" values into code) and enables administrators to easily customize these values post-deployment. The ASP.NET configuration system has been designed to meet the needs of both of these audiences; it provides a hierarchical configuration infrastructure that enables extensible configuration data to be defined and used throughout an application, site, and/or machine. It has the following qualities that make it uniquely suited to building and maintaining Web applications:

- ASP.NET allows configuration settings to be stored together with static content, dynamic pages, and business objects within a single application directory hierarchy. A user or administrator simply needs to copy a single directory tree to set up an ASP.NET Framework application on a machine.

- Configuration data is stored in plain text files that are both human-readable and human-writable. Administrators and developers can use any standard text editor, XML parser, or scripting language to interpret and update configuration settings.

- ASP.NET provides an extensible configuration infrastructure that enables third-party developers to store their own configuration settings, define the persistence format of their own configuration settings, intelligently participate in their processing, and control the resulting object model through which those settings are ultimately exposed.

- Changes to ASP.NET configuration files are automatically detected by the system and are applied without requiring any user intervention (in other words, an administrator does not need to restart the Web server or reboot the machine for them to take effect).

- Configuration sections can be locked down when using the **<location>** tag and the **allowOverride** attribute.

To learn more about the ASP.NET configuration system and

Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications

how it works, see Configuration File Format and Retrieving Configuration.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications

Configuration File Format

ASP.NET configuration files are XML-based text files--each named web.config--that can appear in any directory on an ASP.NET Web application server. Each web.config file applies configuration settings to the directory it is located in and to all virtual child directories beneath it. Settings in child directories can optionally override or modify settings specified in parent directories. The root configuration file--WinNT\Microsoft.NET\Framework\<version>\config\machine.config--provides default configuration settings for the entire machine. ASP.NET configures IIS to prevent direct browser access to web.config files to ensure that their values cannot become public (attempts to access them will cause ASP.NET to return 403: Access Forbidden).

At run time ASP.NET uses these web.config configuration files to hierarchically compute a unique collection of settings for each incoming URL target request (these settings are calculated only once and then cached across subsequent requests; ASP.NET automatically watches for file changes and will invalidate the cache if any of the configuration files change).

For example, the configuration settings for the URL http://myserver/myapplication/mydir/page.aspx would be computed by applying web.config file settings in the following order:

```
Base configuration settings for machine.
C:\WinNT\Microsoft.NET\Framework\v.1.00\config\machine.config

Overridden by the configuration settings for the site (or the root application).
C:\inetpub\wwwroot\web.config

Overridden by application configuration settings.
D:\MyApplication\web.config

Overridden by subdirectory configuration settings.
D:\MyApplication\MyDir\web.config
```

If a web.config file is present at the root directory for a site, for example "Inetpub\wwwroot", its configuration settings will apply to every application in that site. Note that the presence of a web.config file within a given directory or application root is completely optional. If a web.config file is not present, all configuration settings for the directory are automatically inherited from the parent directory.

Configuration Section Handlers and Sections

A web.config file is an XML-based text file that can contain standard XML document elements, including well-formed tags, comments, text, cdata, and so on. The file may be ANSI, UTF-8, or Unicode; the system automatically detects the encoding. The root element of a web.config file is always a **<configuration>** tag. ASP.NET and end-user settings are then encapsulated within the tag, as follows:

```
<configuration>
    <!- Configuration settings would go here. -->
</configuration>
```

The **<configuration>** tag typically contains three different types of elements: 1) configuration section handler declarations, 2) configuration section groups, and 3) configuration section settings.

- **Configuration section handlers** - The ASP.NET configuration infrastructure makes no assumptions regarding the file format or supported settings within a web.config file. Instead, it delegates the processing of web.config data to configuration section handlers, .NET Framework classes that implement the **IConfigurationSectionHandler** interface. An individual **IConfigurationSectionHandler** declaration needs to appear only once, typically in the machine.config file. The web.config files in child directories automatically inherit this declaration. Configuration section handlers are declared within a web.config file using section tag directives nested within a **<configSections>** tag. Section tags may be further qualified by section group tags to organize them into logical groups (see below). Each section tag identifies a tag name denoting a specific section of configuration data and an associated **IConfigurationSectionHandler** class that processes it.

- **Configuration section groups** - ASP.NET configuration allows hierarchical grouping of sections for organizational purposes. A **<sectionGroup>** tag may appear inside a **<configSections>** tag or inside other **<sectionGroup>** tags. For example, ASP.NET section handlers all appear within the **<system.web>** section group.

- **Configuration sections** - ASP.NET configuration settings are represented within configuration tag sections, also nested within a **<configuration>** tag (and optional section group tags). For each configuration section, an appropriate section handler must be defined in the config hierarchy. For

Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

example, in the sample below, the tag **<httpModules>** is the configuration section that defines the HTTP modules configuration data. The **System.Configuration.HttpModulesConfigurationHandler** class is responsible for interpreting the content contained within the **<httpModules>** tag at run time. Note that both the section handler definition and the section must have the same section group qualifier (in this case, **<system.web>**). Also note that tag names are case-sensitive and must be typed exactly as shown. Various attributes and settings for ASP.NET are also case-sensitive and will not be examined by the configuration runtime if the case does not match.

```
<configuration>

  <configSections>
    <sectionGroup name="system.web">
      <section
        name="httpModules"
        type="System.Web.Configuration.HttpModulesConfigurationHandler,System.Web"
      />
    </sectionGroup>
  </configSections>

  <system.web>
    <httpModules>
      <add
        name="CookielessSession"
        type="System.Web.SessionState.CookielessSessionModule,System.Web"
      />
      <add
        name="OutputCache"
        type="System.Web.Caching.OutputCacheModule,System.Web"
      />
      <add
        name="Session"
        type="System.Web.SessionState.SessionStateModule,System.Web"
      />
      <add
        name="WindowsAuthentication"
        type="System.Web.Security.WindowsAuthenticationModule,System.Web"
      />
      <add
        name="FormsAuthentication"
        type="System.Web.Security.FormsAuthenticationModule,System.Web"
      />
      <add
        name="PassportAuthentication"
        type="System.Web.Security.PassportAuthenticationModule,System.Web"
      />
      <add
        name="UrlAuthorization"
        type="System.Web.Security.UrlAuthorizationModule,System.Web"
      />
      <add
        name="FileAuthorization"
        type="System.Web.Security.FileAuthorizationModule,System.Web"
      />
    </httpModules>
  </system.web>

</configuration>
```

Using Location and Path

By default, all configuration settings defined within the top-level **<configuration>** tag are applied to the current directory location of the containing web.config file and to all of the child paths beneath it. You can optionally apply configuration settings to specific child paths under the current config file by using the **<location>** tag with an appropriately constraining **path** attribute. If the config file is the main machine.config file, you can apply settings to specific virtual directories or applications. If the config file is a web.config file, you can apply settings to a specific file, child directory, virtual directory, or application.

```
<configuration>

  <location path="EnglishPages">
    <system.web>
      <globalization
        requestEncoding="iso-8859-1"
        responseEncoding="iso-8859-1"
```

```
      />
    </system.web>
  </location>

  <location path="EnglishPages/OneJapanesePage.aspx">
    <system.web>
      <globalization
        requestEncoding="Shift-JIS"
        responseEncoding="Shift-JIS"
      />
    </system.web>
  </location>

</configuration>
```

Locking down configuration settings

In addition to specifying path information using the **<location>** tag, you can also specify security so that settings cannot be overridden by another configuration file further down the configuration hierarchy. To lock down a group of settings, you can specify an **allowOverride** attribute on the surrounding **<location>** tag and set it to false. The following code locks down impersonation settings for two different applications.

```
<configuration>

  <location path="app1" allowOverride="false">
    <system.web>
        <identity impersonate="false" userName="app1" password="app1pw" />
    </system.web>
  </location>

  <location path="app2" allowOverride="false">
    <system.web>
        <identity impersonate="false" userName="app2" password="app2pw" />
    </system.web>
  </location>

</configuration>
```

Note that if a user tries to override these settings in another configuration file, the configuration system will throw an error:

```
<configuration>

  <system.web>
      <identity userName="developer" password="loginpw" />
  </system.web>

</configuration>
```

Standard ASP.NET Configuration Section

ASP.NET ships with a number of standard configuration section handlers that are used to process configuration settings within web.config files. The following table provides brief descriptions of the sections, along with pointers to more information.

Section Name	Description
<httpModules>	Responsible for configuring HTTP modules within an application. HTTP modules participate in the processing of every request into an application. Common uses include security and logging.
<httpHandlers>	Responsible for mapping incoming URLs to **IHttpHandler** classes. Subdirectories do not inherit these settings. Also responsible for mapping incoming URLs to **IHttpHandlerFactory** classes. Data represented in **<httpHandlers>** sections are hierarchically inherited by subdirectories. For more information, see the Http Handlers and Factories section of this tutorial.

<sessionState>	Responsible for configuring the session state HTTP module. For more information, see the Managing Application State section of this tutorial.
<globalization>	Responsible for configuring the globalization settings of an application. For more information, see the Localization section of this tutorial.
<compilation>	Responsible for all compilation settings used by ASP.NET. For more information, see the Business Objects and Debugging sections of this tutorial.
<trace>	Responsible for configuring the ASP.NET trace service. For more information, see the Tracing section of this tutorial.
<processModel>	Responsible for configuring the ASP.NET process model settings on IIS Web server systems.
<browserCaps>	Responsible for controlling the settings of the browser capabilities component. For more information, see the Retrieving Configuration section of this tutorial.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

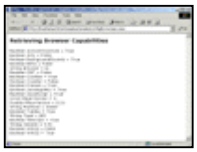
Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding

Retrieving Configuration

ASP.NET allows developers to access configuration settings from within an application either by exposing configuration settings directly (as strongly typed properties) or by using general configuration APIs. The following sample shows a page that accesses the **<browserCaps>** configuration section using the **Browser** property of the **System.Web.HttpRequest** class. This is a hash table of attributes that reflect the capabilities of the browser client that is currently accessing the page. The actual **<browserCaps>** section data is included in the machine.config file.



VB BrowsCaps.aspx

[Run Sample] | [View Source]

In addition to accessing configuration settings, as demonstrated above, developers also can use the **System.Configuration.ConfigurationSettings** class to retrieve the data for any arbitrary configuration section. Note that the particular object returned by **ConfigurationSettings** depends on the section handler mapped to the configuration section (see **IConfigurationSectionHandler.Create**). The following code demonstrates how you can access the configuration data exposed for a **<customconfig>** section. In this example, it is assumed that the configuration section handler returns an object of type **CustomConfigSettings** with the property **Enabled**.

```
Dim config As CustomConfigSettings = CType(ConfigurationSettings("customconfig"),
CustomConfigSettings)

If config.Enabled = True Then
    ' Do something here.
End If
```

VB

Using Application Settings

Configuration files are perfectly suited for storing custom application settings, such as database connection strings, file paths, or remote XML Web service URLs. The default configuration sections (defined in the machine.config file) include an **<appSettings>** section that may be used to store these settings as name/value pairs. The following example shows an **<appSettings>** configuration section that defines database connection strings for an application.

```
<configuration>
  <appSettings>
    <add key="pubs"
value="server=(local)\NetSDK;database=pubs;Trusted_Connection=yes" />
    <add key="northwind"
value="server=(local)\NetSDK;database=northwind;Trusted_Connection=yes" />
  </appSettings>
</configuration>
```

The **ConfigurationSettings** object exposes a special **AppSettings** property that can be used to retrieve these settings:

```
Dim dsn As String = ConfigurationSettings.AppSettings("pubs")
```

VB

The following sample illustrates this technique.

Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

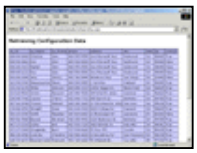
Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page



VB Config1.aspx

[Run Sample] | [View Source]

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview

Deploying ASP.NET Applications

- File System Layout of ASP.NET Applications
- Resolving Class References to Assemblies
- ASP.NET Framework application Startup and Class Resolution
- Code Replacement
- Section Summary

File System Layout of ASP.NET Applications

ASP.NET can be used to host multiple Web applications, each identified using a unique URL prefix within a Web site (where a Web site is represented on a Web server as a unique **HostName**/**Port** combination). For example, a single Microsoft Internet Information Services (IIS) Web server with two mapped IP addresses (one aliased to "www.msn.com" and the other to "intranet") and three logical sites (http://intranet, http://www.msn.com, http://www.msn.com port 81) could expose the following six ASP.NET applications.

Application URL	Description
http://intranet	"Root" application on intranet site.
http://www.msn.com	"Root" application on www.msn.com site.
http://www.msn.com:81	"Root" application on www.msn.com port 81 site.
http://intranet/training	"Training" application on intranet site.
http://intranet/hr	"HR" application on intranet site.
http://intranet/hr/compensation/	"Compensation" application on intranet site.

Note: The URL for the compensation application mentioned in the table is rooted within the HR application URL namespace. However, this URL hierarchy notation does not imply that the compensation application is contained or nested within the HR application. Rather, each application maintains an independent set of configuration and class resolution properties; both are logical peer child sites of the intranet site.

Each ASP.NET Framework application exposed within a URL namespace is backed using a file system directory located on either a local or remote file share. Application directories are not required to be centrally located within a contiguous part of the file system; they can be scattered throughout a disk. For example, the ASP.NET applications mentioned previously could be located in the different directories listed in the following table.

Application URL	Physical path
http://intranet	c:\inetpub\wwwroot
http://www.msn.com	c:\inetpub\msnroot
http://www.msn.com:81	d:\msnroot81
http://intranet/training	d:\serverapps\trainingapp
http://intranet/hr	\\hrweb\sillystuff\reviews
http://intranet/hr/compensation/	c:\inetpub\wwwroot\compensation

Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

Resolving Class References to Assemblies

Assemblies are the unit of class deployment in the common language runtime. Developers writing .NET Framework classes using Visual Studio .NET version 7.0 will produce a new assembly with each Visual Studio project that they compile. Although it is possible to have an assembly span multiple portable executable (PE) files (several module DLLs), Visual Studio .NET will, by default, compile all assembly code into a single DLL (1 Visual Studio .NET project = 1 .NET Framework assembly = 1 physical DLL).

You can use an assembly on a computer by deploying it into an assembly cache. The assembly cache can be either global to a computer or local to a particular application. Only code intended to be shared across multiple applications should be placed in the global system assembly cache. Code specific to a particular application, such as most Web application logic, should be deployed in the application's local assembly cache. One advantage of deploying an assembly within an application's local assembly cache is that only code within that application can access it. (This is a nice feature for scenarios involving ISPs.) It also facilitates side-by-side versioning of the same application because classes are private to each application version instance.

An assembly can be deployed into an application's local assembly cache by simply copying, XCOPYing, or FTPing the appropriate files into a directory that has been marked as an "assembly cache location" for that particular application. No additional registration tool must be run once the appropriate files are copied, and no reboot is necessary. This eliminates some of the difficulties currently associated with deploying COM components within ASP applications (currently, an administrator must log on to the Web server locally and run Regsvr32.exe).

By default, an ASP.NET Framework application is automatically configured to use the \bin subdirectory, located immediately under the application root, as its local assembly cache. The \bin directory is also configured to deny any browser access so that a remote client cannot download and steal the code. The following example shows a possible directory layout for an ASP.NET application, where the \bin directory is immediately under the application root.

```
C:\inetpub\wwwroot Web.cfg Default.aspx \bin <= Application
assembly cache directory MyPages.dll MyBizLogic.dll \order
SubmitOrder.aspx OrderFailed.aspx \img HappyFace.gif
```

ASP.NET Framework application Startup and Class Resolution

ASP.NET Framework applications are lazily constructed the first time a client requests a URL resource from them. Each ASP.NET Framework application is launched within a unique application domain (**AppDomain**)--a new common language runtime construct that enables process hosts to provide extensive code, security, and configuration isolation at run time.

ASP.NET is responsible for manually creating an application domain when a new application is started. As part of this process, ASP.NET provides configuration settings for the common language runtime to use. These settings include:

- The directory paths that make up the local assembly cache. (Note: It is the .NET Framework application domain isolation architecture that allows each application to maintain its own local assembly cache.)
- The application's security restrictions (what the application can access on the system).

Because ASP.NET does not have compile-time knowledge of any applications you write on top of it, it cannot use static references to resolve and reference application code. Instead, ASP.NET must use a dynamic class/assembly resolution approach to make the transition from the ASP.NET runtime into application code.

ASP.NET configuration and page activation files will enable you to dynamically reference a target-compiled .NET Framework class by specifying an assembly and class name combination. The string format for this union follows the pattern

```
classname, assemblyname
```

. The common language runtime can then use this simple string reference to resolve and load the appropriate class.

Code Replacement

.NET Framework assemblies are typically compiled and deployed into a Windows DLL-based PE format. When the common language runtime's loader resolves a class implemented within this type of assembly, it calls the Windows LoadLibrary routine on the file (which locks its access on disk), and then maps the appropriate code data into memory for run-time execution. Once loaded, the DLL file will remain locked on disk until the application domain referencing it is either torn down or manually recycled.

Although ASP.NET cannot prevent the common language runtime from locking a loaded assembly DLL on disk, it can support you by ensuring that the physical DLLs in a Web application's private assembly cache are never actually loaded by the runtime. Instead, shadow copies of the assembly DLLs are made immediately prior to their use. These shadow assemblies--not the original files--are then locked and loaded by the runtime.

Because the original assembly files always remain unlocked, you are free to delete, replace, or rename them without cycling the Web server or having to use a registration utility. FTP and similar methods work just fine. ASP.NET maintains an active list of all assemblies loaded within a particular application's application domain and uses file-change monitoring code to watch for any updates to the original files.

Section Summary

1. ASP.NET Framework applications are identified by a unique URL and live in the file system of the Web server.
2. ASP.NET can use shared assemblies, which reside in the global cache, and application-specific assemblies, which reside in the \bin directory of the application's virtual root.
3. ASP.NET Framework applications run in the context of application domains (AppDomains), which provide isolation and enforce security restrictions.
4. Classes can be dynamically referenced using "classname, assemblyname".
5. ASP.NET uses shadow copies of assembly files to avoid locking, it and monitors the files so that changes are picked up immediately.

Copyright 2001 Microsoft Corporation. All rights reserved.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview

Using the Process Model

- Process Model Configuration
- Reactive Process Recycling
- Proactive Process Recycling
- Logging Process Model Events
- Enabling Web Gardens
- Section Summary

One of the most important requirements for ASP.NET Framework applications is reliability. The architecture of applications running inside the server process (in IIS, Inetinfo.exe) does not produce a solid foundation for building reliable applications that can continue to run over a long period of time. Too many resources are shared on the process level, and it is too easy for an error to bring down the entire server process.

To solve this problem, ASP.NET provides an out-of-process execution model, which protects the server process from user code. It also enables you to apply heuristics to the lifetime of the process to improve the availability of your Web applications. Using asynchronous interprocess communication enables you to provide the best balance of performance, scalability, and reliability.

Process Model Configuration

Process model settings are exposed in the root configuration file for the computer, Machine.config. The configuration section is named **<processModel>** and is shown in the following example. The process model is enabled by default (enable="true").

```
<processModel
    enable="true"
    timeout="infinite"
    idleTimeout="infinite"
    shutdownTimeout="0:00:05"
    requestLimit="infinite"
    requestQueueLimit="5000"
    memoryLimit="80"
    webGarden="false"
    cpuMask="0xffffffff"
    userName=""
    password=""
```

Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal

```
        logLevel="errors"
        clientConnectedCheck="0:00:05"
    />
```

Most of these settings control when a new worker process is started to serve requests (gracefully taking the place of an old worker process). The process model supports two types of recycling: reactive and proactive.

Reactive Process Recycling

Reactive process recycling occurs when a process is misbehaving or unable to serve requests. The process typically displays detectable symptoms, such as deadlocks, access violations, memory leaks, and so on, in order to trigger a process recycle. You can control the conditions that trigger a process restart by using the configuration settings described in the following table.

Setting	Description
requestQueueLimit	Handles deadlock conditions. The DWORD value is set to the maximum allowed number of requests in the queue, after which the worker process is considered to be misbehaving. When the number is exceeded, a new process is launched and the requests are reassigned. The default is 5000 requests.
memoryLimit	Handles memory leak conditions. The DWORD value is set to the percentage of physical memory that the worker process can consume before it is considered to be misbehaving. When that percentage is exceeded, a new process is launched and the requests are reassigned. The default is 80%.

shutdownTimeout	Specifies the amount of time the worker process has to shut itself down gracefully (string value in hr:min:sec format). When the time out expires, the ASP.NET ISAPI shuts down the worker process. The default is "00:00:05".

Proactive Process Recycling

Proactive process recycling restarts the worker process periodically even if the process is healthy. This can be a useful way to prevent denials of service due to conditions the process model is unable to detect. A process can be restarted after a specific number of requests or after a time-out period has elapsed.

Setting	Description
timeout	String value in hr:min:sec format that configures the time limit after which a new worker process will be launched to take the place of the current one. The default is **infinite**, a keyword indicating that the process should not be restarted.
idleTimeout	String value in hr:min:sec format that configures the amount of inactivity, after which the worker process is automatically shut down. The default is **infinite**, a keyword indicating that the process should not be restarted.
requestLimit	DWORD value set to the number of requests after which a new worker process will be launched to take the place of the current one. The default is **infinite**, a keyword indicating that the process should not be restarted.

Logging Process Model Events

The process model can write events to the Windows event log when process cycling occurs. This is controlled by the **logLevel** attribute in the **<processModel>** configuration section.

Setting	Description
logLevel	Controls that process cycling events are logged to the event log. The value can be: • **All**: All process cycling events are logged. • **None**: No events are logged. • **Errors**: Only unexpected events are logged.

When a cycling event occurs, if logging is enabled for that event, the following events are written to the application event log.

Shutdown Reason	Event Log Type	Description
Unexpected	Error	The ASP.NET worker process has shut down unexpectedly.
RequestQueueLimit	Error	The ASP.NET worker process has been restarted because the request queue limit was exceeded.
RequestLimit	Information	The ASP.NET worker process has been restarted because the request limit was exceeded.

Timeout	Information	The ASP.NET worker process has been restarted because the time-out interval was met.
IdleTimeout	Information	The ASP.NET worker process has been shut down because the idle time-out interval was met.
MemoryLimitExceeded	Error	The ASP.NET worker process was restarted because the process memory limit was exceeded.

Enabling Web Gardens

The process model helps enable scalability on multiprocessor computers by distributing the work to several processes, one per CPU, each with processor affinity set to its CPU. This eliminates cross-processor lock contention and is ideal for large SMP systems. This technique is called Web gardening. The configuration settings for enabling Web gardens are listed in the following table. Note that these settings take effect only after a server is restarted. IIS must be cycled in order for this change to take place.

Setting	Description
webGarden	Controls CPU affinity. **True** indicates that processes should be affinitized to the corresponding CPU. The default is **False**.

cpuMask	Controls the number of processes and how the Web garden works. One process is launched for each CPU where the corresponding bit in the mask set to 1. When **UseCPUAffinity** is set to 0, the **cpuMask** setting only controls the number of worker processes (number of bits set to 1). The maximum-allowed number of worker processes is the number of CPUs. By default, all CPUs are enabled; the same number of worker processes is launched as there are CPUs. The default value is 0xffffffff.

Web gardening has some side effects that you should be aware of:

- If your application uses session state, it must choose an out-of-process provider (NT Service or SQL).
- Application state and application statics are per process, not per computer.
- Caching is per process, not per computer.

Section Summary

1. ASP.NET provides an out-of-process execution model, which protects the server process from user code. It also enables you to apply heuristics to the lifetime of the process to improve the overall availability of Web applications.
2. The **<processModel>** settings are exposed in the root configuration file for the computer's Machine.config file. The process model is enabled by default.
3. The process model supports two types of recycling: reactive and proactive. Reactive process recycling occurs when a process is misbehaving or unable to serve requests. Proactive process recycling restarts the worker process periodically, even when the process may be healthy.
4. The process model can write events to the Windows event log when process cycling occurs. This is controlled by the log-level attribute in the **<processModel>** configuration section.
5. The process model helps enable scalability on multiprocessor computers by distributing the work to several processes, one per CPU, each with processor affinity set to its CPU. This technique is called Web

gardening.

Copyright 2001 Microsoft Corporation. All rights reserved.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Handling Errors

- Customizing Error Pages
- Handling Errors Programmatically
- Writing to the Event Log
- Section Summary

When an error occurs on a page, ASP.NET sends information about the error to the client. Errors are divided into four categories:

- Configuration errors: Occur when the syntax or structure of a Web.config file in the configuration hierarchy is incorrect.
- Parser errors: Occur when the ASP.NET syntax on a page is malformed.
- Compilation errors: Occur when statements in a page's target language are incorrrect.
- Run-time errors: Occur during a page's execution, even though the errors could not be detected at compile time.

By default, the information shown for a run-time error is the call stack (the chains of procedure calls leading up to the exception). If debug mode is enabled, ASP.NET displays the line number in source code where the run-time error originated. Debug mode is a valuable tool for debugging your application. You can enable debug mode at the page level, using the following directive:

```
<%@ Page Debug="true" %>
```

You can also enable debug mode at the application level, using the Web.config file in the application's root directory, as shown in the following example.

Note: Running debug mode incurs a heavy performance penalty. Be sure to disable it before deploying your finished application.

The following example demonstrates the use of debug mode to show source line numbers for a run-time exception.

VB Error1.aspx

[Run Sample] | [View Source]

Customizing Error Pages

Depending on the circumstances, you might want to handle application errors in different ways. For example, at development time you probably want to see the detailed error pages that ASP.NET provides to help you identify and fix problems. However, once an application is being served in a production environment, you probably do not want to display detailed errors to your customer clients. You can use ASP.NET to specify whether errors are shown to local clients, to remote clients, or to both. By default, errors are shown only to local clients (those clients on the same computer as the server). You can also specify a custom error page to redirect clients to if an error occurs.

Custom errors are enabled in the Web.config file for an application. For example:

```
<configuration>
     <system.web>
         <customErrors defaultRedirect="genericerror.htm" mode="remoteonly" />
     </system.web>
</configuration>
```

This configuration enables local clients to see the default ASP.NET detailed error pages but redirects remote clients to a custom page, genericerror.htm. This page could be an .aspx page as well. ASP.NET passes the path of the page on which the error occurred to the error page as a **QueryString** argument. Note that if the execution of the error page generates an error, a blank page is sent back to the remote client.

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

```
<%@ Page Language="VB" Description="Error page"%>

<html>
<head>
<title>Error page</title>
</head>

<body>
<h1>Error page</h1>
Error originated on: <%=Request.QueryString("ErrorPage") %>
</body>
</html>
```

VB

Note: Only files mapped to the aspnet_isapi.dll extension in IIS generate these errors. Files not served through the aspnet_isapi.dll are not processed by ASP.NET and generate IIS errors. See the IIS documentation for information about configuring IIS custom errors.

The following table describes the configuration attributes and values for the **<customerrors>** tag.

Attribute	Description
Mode	Indicates whether custom errors are enabled, disabled, or only shown to remote computers. Values: **On**, **Off**, **RemoteOnly** (default).
DefaultRedirect	Indicates the default URL to which a browser should be redirected if an error occurs. This attribute is optional.

The **Mode** attribute determines whether errors are shown to local clients, remote clients, or both. The effects of each setting are described in the following table.

Mode	Local host request	Remote host request
On	Custom error page.	Custom error page.
Off	ASP.NET error page.	ASP.NET error page.
RemoteOnly	ASP.NET error page.	Custom error page.

The following sample demonstrates how the **<customerrors>** configuration section is used.

VB Custom1.aspx

[Run Sample] | [View Source]

In addition to redirecting to a common page for all errors, you can also assign specific error pages to specific error status codes. The **<customerrors>** configuration section supports an inner **<error>** tag that associates HTTP status codes with custom error pages. For example:

```
<configuration>
  <system.web>
    <customErrors mode="RemoteOnly" defaultRedirect="/genericerror.htm">
      <error statusCode="500" redirect="/error/callsupport.htm"/>
      <error statusCode="404" redirect="/error/notfound.aspx"/>
      <error statusCode="403" redirect="/error/noaccess.aspx"/>
    </customErrors>
  </system.web>
</configuration>
```

The following table describes the attributes and values for the **<error>** tag.

Attribute	Description
StatusCode	HTTP status code of errors for which the custom error page should be used. Examples: 403 Forbidden, 404 Not Found, or 500 Internal Server Error.

Redirect	URL to which the client browser should be redirected if an error occurs.

The following example demonstrates how to use the **<error>** tag. Note that the example specifies an .aspx page for "File Not Found" errors so that the missing page URL passed via the **QueryString** can be printed.

VB Custom1.aspx

[Run Sample] | [View Source]

Handling Errors Programmatically

You can also handle errors in code, at either the page level or the application level. The **Page** base class exposes a **Page_Error** method, which you can override in your pages. The method is called whenever an uncaught exception is thrown at run time.

```
<script language="C#" runat="server">

Sub Page_Error(Source As Object, E As EventArgs)
    Dim message As String = "<font face=verdana color=red>" _
                            & "<h4>" & Request.Url.ToString() & "</h4>" _
                            & "<pre><font color='red'>" &
Server.GetLastError().ToString() & "</pre>" _
                            & "</font>"

    Response.Write(message)
End Sub

</script>
```

VB

The following sample demonstrates the **Page_Error** method.

VB Error2.aspx

[Run Sample] | [View Source]

You could do something useful in this method, like sending an e-mail to the site administrator saying that the page failed to execute properly. ASP.NET provides a set of classes in the **System.Web.Mail** namespace for just this purpose. To import this namespace, use an **@Import** directive at the top of your page, as follows:

```
<%@ Import Namespace="System.Web.Mail" %>
```

You can then use the **MailMessage** and **SmtpMail** objects to programmatically send e-mail.

```
Dim mail As New MailMessage
mail.From = "automated@yourservername.com"
mail.To = "administrator@yourservername.com"
mail.Subject = "Site Error"
mail.Body = message
mail.BodyFormat = MailFormat.Html
SmtpMail.Send(mail)
```

VB

The following sample shows how to send a mail message in response to a page error.

Note: This sample does not actually send mail unless you have configured the SMTP mail service on your machine. For more information about the SMTP mail service, consult the IIS documentation.

VB Error3.aspx

[Run Sample] | [View Source]

In addition to handling errors at the page level, you might want to handle errors at the application level. To do this, use the **Application_Error** event in Global.asax. This event occurs for any unhandled exception thrown within the application.

```
Sub Application_Error(sender As Object, e As EventArgs)
    '...Do something here
End Sub
```

VB

Writing to the Event Log

The **System.Diagnostics** namespace provides classes for writing to the Windows event log. To use this namespace in your pages, you must first import the namespace, as follows:

```
<%@ Import Namespace="System.Diagnostics"%>
```

The **EventLog** class encapsulates the log itself. It provides static methods for detecting or creating logs and can be instantiated to write log entries from code. The following example shows this functionality within the **Application_Error** method of Global.asax. Whenever an unhandled exception occurs in the application, an entry containing the error message and stack trace is made to the application log.

```
Sub Application_Error(sender As Object, e As EventArgs)

     Dim Message As String = "\n\nURL:\n http://localhost/" & Request.Path _
                                     & "\n\nMESSAGE:\n " &
Server.GetLastError().Message _
                                     & "\n\nSTACK TRACE:\n" &
Server.GetLastError().StackTrace

    ' Create event log if it does not exist

     Dim LogName As String = "Application"
     If (Not EventLog.SourceExists(LogName))
        EventLog.CreateEventSource(LogName, LogName)
     End If

     ' Insert into event log
     Dim Log As New EventLog
     Log.Source = LogName
     Log.WriteEntry(Message, EventLogEntryType.Error)

End Sub
```

VB

The complete source code for the preceding example follows. Note that this code is disabled so that it cannot run here, to prevent entries to your Windows event log. If you would like to see this code run, create an IIS virtual root pointing to the directory that contains this file.

VB Global.asax

[View Source]

Section Summary

1. Errors are divided into four categories: configuration errors, parser errors, compilation errors, and run-time errors.
2. By default, the information shown for a run-time error is the call stack (the chain of procedure calls leading up to the exception). If debug mode is enabled, ASP.NET displays the line number in source code where the run-time error originated.
3. ASP.NET enables you to specify whether errors are shown to local clients, to remote clients, or to both. By default, errors are only shown to local clients (clients on the same computer as the server). You can also specify a custom error page to redirect clients to if an error occurs.
4. In addition to redirecting to a common page for all errors, you can also assign specific error pages to specific error status codes. The **<customerrors>** configuration section supports an inner **<error>** tag for associating HTTP status codes with custom error pages.
5. You can also handle errors in code, at either the page level or application level. The **Page** base class exposes a **HandleError** method, which you can override in your pages. The method will be called whenever an uncaught exception is thrown at run time.
6. The **System.Web.Mail** namespace exposes classes for programmatically sending e-mail. This is useful for notifying an administrator when an error occurs.
7. In addition to handling errors at the page level, you can use the **Application_Error** event in Global.asax to handle errors at the application level. This event will occur for any unhandled exception thrown within the application.
8. The **System.Diagnostics** namespace provides classes for writing to the Windows event log.

Copyright 2001 Microsoft Corporation. All rights reserved.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview

Security Overview

An important part of many Web applications is the ability to identify users and control access to resources. The act of determining the identity of the requesting entity is known as authentication. Generally, the user must present credentials, such as a name/password pair in order to be authenticated. Once an authenticated identity is available, it must be determined whether that identity can access a given resource. This process is known as authorization. ASP.NET works in conjunction with IIS to provide authentication and authorization services to applications.

An important feature of COM objects is the ability to control the identity under which COM object code is executed. When a COM object executes code with the identity of the requesting entity, this is known as impersonation. ASP.NET Framework applications can optionally choose to impersonate requests.

Some applications also want to be able to dynamically tailor content, based on the requesting identity or based on a set of roles that a requesting identity belongs to. ASP.NET Framework applications can dynamically check whether the current requesting identity participates in a particular role. For example, an application might want to check to see whether the current user belongs to the manager's role, in order to conditionally generate content for managers.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format

Authentication and Authorization

ASP.NET works in conjunction with IIS to support authentication, using Basic, Digest, and Windows authentication. ASP.NET supports the Microsoft Passport authentication service, which provides single sign-on services and support for user profile services. ASP.NET also provides a robust service for applications that want to use forms-based authentication. Forms-based authentication uses cookies to authenticate users and allows the application to do its own credential verification.

It is important to realize that ASP.NET authentication services are subject to the authentication services provided by IIS. For example, in order to use Basic authentication in an IIS application, you must configure the use of Basic authentication for the application using the Internet Service Manager tool.

ASP.NET provides two types of authorization services:

- Checks against ACLs or permissions on a resource to determine whether the authenticated user account can access the resources
- URL authorization, which authorizes an identity for pieces of the Web space

To illustrate the difference, consider a scenario in which an application is configured to allow anonymous access using the IUSR_MYMACHINE account. When a request for an ASP.NET page (such as "/default.aspx") is authorized, a check is done against the ACLs on that file (for example, "c:\inetpub\wwwroot\default.aspx") to see whether the IUSR_MYMACHINE account has permission to read the file. If it does, then access is authorized. File authorization is performed automatically.

For URL authorization, the anonymous user is checked against the configuration data computed for the ASP.NET application. If access is allowed for the requested URL, the request is authorized. In this case, ASP.NET checks to see whether the anonymous user has access to /Default.aspx (that is, the check is done against the URL itself, not against the file that the URL ultimately resolves to).

This might seem a subtle distinction, but it enables applications to use authentication schemes likes forms-based authentication or Passport authentication, in which the users do not correspond to a machine or domain account. It also enables authorization against virtual resources, for which there is no physical file underlying the resource. For example, an

Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application

application could choose to map all requests for files ending in .stk to a handler that serves stock quotes based on variables present in the query string. In such a case, there is no physical .stk against which to do ACL checks, so URL authorization is used to control access to the virtual resource.

File authorization is always performed against the authenticated account provided by IIS. If anonymous access is allowed, this is the configured anonymous account. Otherwise, it uses an NT account. This works in exactly the same way as ASP.

File ACLs are set for a given file or directory using the **Security** tab in the Explorer property page. URL authorization is configured as part of an ASP.NET Framework application and is described fully in Authorizing Users and Roles.

To activate an ASP.NET authentication service, you must configure the **<authentication>** element in the application's configuration file. This element can have any of the values listed in the following table.

Value	Description
None	No ASP.NET authentication services are active. Note that IIS authentication services can still be present.
Windows	ASP.NET authentication services attach a **WindowsPrincipal** (**System.Security.Principal.WindowsPrincipal**) to the current request to enable authorization against NT users or groups.
Forms	ASP.NET authentication services manage cookies and redirect unauthenticated users to a logon page. This is often used in conjunction with the IIS option to allow anonymous access to an application.
Passport	ASP.NET authentication services provide a convenient wrapper around the services provided by the Passport SDK, which must be installed on the machine.

For example, the following configuration file enables forms-based (cookie) authentication for an application:

```
<configuration>
  <system.web>
    <authentication mode="Forms"/>
  </system.web>
</configuration>
```

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Windows-based Authentication

When you use ASP.NET Windows authentication, ASP.NET attaches a **WindowsPrincipal** object to the current request. This object is used by URL authorization. The application can also use it programatically to determine whether a requesting identity is in a given role.

```
If User.IsInRole("Administrators") Then
    DisplayPrivilegedContent()
End If
```

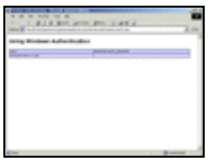
VB

The **WindowsPrincipal** class determines roles by NT group membership. Applications that want to determine their own roles can do so by handling the **WindowsAuthentication_OnAuthenticate** event in their Global.asax file and attaching their own class that implements **System.Security.Principal.IPrincipal** to the request, as shown in the following example:

```
' Create a class that implements IPrincipal
Public Class MyPrincipal : Inherits IPrincipal
  ' Implement application-defined role mappings
End Class

' In a Global.asax file
Public Sub WindowsAuthentication_OnAuthenticate(Source As Object, e As
WindowsAuthenticationEventArgs)
  ' Attach a new application-defined class that implements IPrincipal to
  ' the request.
  ' Note that since IIS has already performed authentication, the provided
  ' identity is used.
  e.User = New MyPrincipal(e.Identity)
End Sub
```

VB

The following sample shows how to access the name of an authenticated user, which is available as **User.Identity.Name**. Programmers familiar with ASP should note that this value is also still available as the AUTH_USER server variable:



VB Windows Authentication

[Run Sample] | [View Source]

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Forms-Based Authentication

Forms-based authentication is an ASP.NET authentication service that enables applications to provide their own logon UI and do their own credential verification. ASP.NET authenticates users, redirecting unauthenticated users to the logon page, and performing all the necessary cookie management. This sort of authentication is a popular technique used by many Web sites.

An application has to be configured to use forms-based authentication by setting **<authentication>** to **Forms**, and denying access to anonymous users. The following example shows how this can be done in the Web.config file for the desired application:

```
<configuration>
  <system.web>
    <authentication mode="Forms"/>
    <authorization>
        <deny users="?" />
    </authorization>
  </system.web>
</configuration>
```

Administrators use forms-based authentication to configure the name of the cookie to use, the protection type, the URL to use for the logon page, length of time the cookie is in effect, and the path to use for the issued cookie. The following table shows the valid attributes for the **<Forms>** element, which is a sub-element of the **<authentication>** element shown in the following example:

```
<authentication mode="Forms">
   <forms name=".ASPXCOOKIEDEMO" loginUrl="login.aspx" protection="all" timeout="30"
path="/">
                    <!-- protection="[All|None|Encryption|Validation]" -->
   </forms>
</authentication>
```

Attribute	Description
loginUrl	Logon URL to which unauthenticated users are redirected. This can be on the same computer or a remote one. If it is on a remote computer, both computers need to be using the same value for the **decryptionkey** attribute.
name	Name of the HTTP cookie to use for authentication purposes. Note that if more than one application wants to use forms-based authentication services on a single computer, they should each configure a unique cookie value. In order to avoid causing dependencies in URLs, ASP.NET uses "/" as the Path value when setting authentication cookies, so that they are sent back to every application on the site.
timeout	Amount of time in integer minutes, after which the cookie expires. The default value is 30. The timeout attribute is a sliding value, expiring n minutes from the time the last request was received. In order to avoid adversely affecting performance and to avoid multiple browser warnings for those who have cookies warnings turned on, the cookie is updated if the time is more than half gone. (This means a loss of possible precision in some cases.)
path	Path to use for the issued cookie. The default value is "/" to avoid difficulties with mismatched case in paths, since browsers are strictly case-sensitive when returning cookies. Applications in a shared-server environment should use this directive to maintain private cookies. (Alternatively, they can specify the path at runtime using the APIs to issue cookies.)
protection	Method used to protect cookie data. Valid values are as follows: • **All**: Use both data validation and encryption to protect the cookie. The configured data validation algorithm is based on the element. Triple DES is used for encryption, if available and if the key is long enough (48 bytes). **All** is the default (and suggested) value. • **None**: Use for sites that are only using cookies for personalization and have weaker security requirements. Both encryption and validation can be disabled. Although you should use caution if you use cookies in this way, this setting provides the best performance of any method of doing personalization using the .NET Framework. • **Encryption**: Encrypts the cookie using TripleDES or DES, but data validation is not done on the cookie. This type of cookie can be subject to chosen plaintext attacks. • **Validation**: Does not encrypt the contents of the cookie, but

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

	validates that the cookie data has not been altered in transit. To create the cookie, the validation key is concatenated in a buffer with the cookie data and a MAC is computed and appended to the outgoing cookie.

After the application has been configured, you need to provide a logon page. The following example shows a simple logon page. When the sample is run, it requests the Default.aspx page. Unauthenticated requests are redirected to the logon page (Login.aspx), which presents a simple form that prompts for an e-mail address and a password. (Use Username="jdoe@somewhere.com" and Password="password" as credentials.)

After validating the credentials, the application calls the following:

```
FormsAuthentication.RedirectFromLoginPage(UserEmail.Value, PersistCookie.Checked)
```

VB

This redirects the user back to the originally requested URL. Applications that do not want to perform the redirection can call either **FormsAuthentication.GetAuthCookie** to retrieve the cookie value or **FormsAuthentication.SetAuthCookie** to attach a properly encrypted cookie to the outgoing response. These techniques can be useful for applications that provide a logon UI embedded in the containing page or that want to have more control over where users are redirected. Authentication cookies can either be temporary or permanent ("persistent"). Temporary cookies last only for the duration of the current browser session. When the browser is closed, the cookie is lost. Permanent cookies are saved by the browser and are sent back across browser sessions unless explicitly deleted by the user.

VB Forms-Based/Cookie Authentication

[Run Sample] | [View Source]

The authentication cookie used by forms authentication consists of a linear version of the **System.Web.Security.FormsAuthenticationTicket** class. The information includes the user name (but not the password), the version of forms authentication used, the date the cookie was issued, and a field for optional application-specific data.

Application code can revoke or remove authentication cookies using the **FormsAuthentication.SignOut** method. This removes the authentication cookie regardless of whether it is temporary or permanent.

It is also possible to supply forms-based authentication services with a list of valid credentials using configuration, as shown in the following example:

```
<authentication>
    <credentials passwordFormat="SHA1" >
        <user name="Mary" password="GASDFSA9823598ASDBAD"/>
        <user name="John" password="ZASDFADSFASD23483142"/>
    </credentials>
</authentication>
```

The application can then call **FormsAuthentication.Authenticate**, supplying the username and password, and ASP.NET will verify the credentials. Credentials can be stored in cleartext, or as SHA1 or MD5 hashes, according to the following values of the **passwordFormat** attribute:

Hash Type	Description
Clear	Passwords are stored in cleartext
SHA1	Passwords are stored as SHA1 digests
MD5	Passwords are stored as MD5 digests

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview

Authorizing Users and Roles

ASP.NET is used to control client access to URL resources. It is configurable for the HTTP method used to make the request (**GET** or **POST**) and can be configured to allow or deny access to groups of users or roles. The following example shows access being granted to a user named John and a role named Admins. All other users are denied access.

```
<authorization>
    <allow users="jdoe@somewhere.com" />
    <allow roles="Admins" />
    <deny users="*" />
</authorization>
```

Permissible elements for authorization directives are either **allow** or **deny**. Each **allow** or **deny** element must contain a **users** or a **roles** attribute. Multiple users or roles can be specified in a single element by providing a comma-separated list.

```
<allow users="John,Mary" />
```

The HTTP method can be indicated using the **Verb** attribute:

```
<allow VERB="POST" users="John,Mary" />
<deny VERB="POST" users="*" />
<allow VERB="GET" users="*" />
```

This example lets Mary and John **POST** to the protected resources, while only allowing everyone else to use **GET**.

There are two special usernames:

- *****: All users
- **?**: Anonymous (unauthenticated) users

These special usernames are commonly used by applications using forms-based authentication to deny access to unauthenticated users, as shown in the following example:

Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications

```
<authorization>
    <deny users="?" />
</authorization>
```

URL authorization is computed hierarchically and the rules used to determine access are as follows:

- Rules relevant to the URL are collected from across the hiearchy and a merged list of rules is constructed.
- The most recent rules are placed at the head of the list. This means that configuration in the current directory is at the head of the list, followed by configuration in the immediate parent, and so on, up to the top-level file for the computer.
- Rules are checked until a match is found. If the match is allowable, access is granted. If not, access is disallowed.

What this means is that applications that are not interested in inheriting their configuration should explicitly configure all of the possiblities relevant to their applications.

The default top-level Web.config file for a given computer allows access to all users. Unless an application is configured to the contrary (and assuming that a user is authenticated and passes the file authorization ACL check), access is granted.

When roles are checked, URL authorization effectively marches down the list of configured roles and does something that looks like the following pseudocode:

```
If User.IsInRole("ConfiguredRole") Then
  ApplyRule()
End If
```

VB

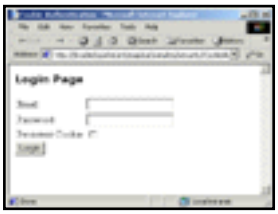
What this means for your application is that you use your own class that implements **System.Security.Principal.IPrincipal** to provide your own role-mapping semantics, as explained in Windows-based Authentication.

The following sample uses forms-based authentication

A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

services. It explicitly denies access to jdoe@somewhere.com and anonymous users. Try logging into the sample with Username="jdoe@somewhere.com" and Password="password". Access will be denied and you will be redirected back to the logon page. Now log on as Username="mary@somewhere.com" and Password="password". You will see that access is granted.



VB Forms-Based/Cookie Authentication with URL Authorization

[Run Sample] | [View Source]

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview

User Account Impersonation

As mentioned in the Security Overview, impersonation refers to a process in which a COM object executes with the identity of the entity on behalf of which it is performing work. What this means for a Web application is that if a server is impersonating, it is doing work using the identity of the client making the request.

By default, ASP.NET does not do per-request impersonation. This is different from ASP, which does impersonate on every request. If desired, you can configure an application to impersonate on every request with the following **Configuration** directive:

```
<identity impersonate="true" />
```

Since ASP.NET does dynamic compilation, enabling impersonation requires that all accounts have read/write access to the application's Codegen directory (where dynamically compiled objects are stored by the ASP.NET runtime) as well as the global assembly cache (%Windir%\assembly). Some applications require impersonation to be enabled for ASP compatibility or to use Windows authentication services.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Security and WebServices

- Windows Authentication and Authorization
- Custom Authentication and Authorization (with Soap Headers)
- Section Summary

This section describes methods for securing your XML Web services. If you haven't already read the Security section of this tutorial, take the time to do so now before continuing in this topic.

Windows Authentication and Authorization

You use the same technique to secure your XML Web services using Windows authentication that you used for .aspx pages (described in the Windows-based Authentication section). To require authentication, you enable **Integrated Windows authentication** for your application and disable **Anonymous access** in the IIS management console. To allow or deny specific users access to your service, use the ASP.NET configuration system or set ACLs on the service file itself, as shown in the following example:

```
<configuration>

  <system.web>
    <authentication mode="Windows"/>
  </system.web>

  <location path="secureservice.asmx">

    <system.web>
      <authorization>
        <allow users="Administrator"/>
        <allow users="DOMAIN\Bradley"/>
        <deny roles="BUILTIN\Power Users"/>
      </authorization>
    </system.web>

  </location>

</configuration>
```

This works well when you know that the client of the XML Web service will be running as a specific Windows user. A more interesting case is is that of a client running as one user, but acting on behalf of another. Consider an ASP.NET page that accesses a secure XML Web service that does not impersonate the client who accesses it. In such a case, you should programmatically set the username and password before connecting to the Web service. The following example uses basic authentication and illustrates a simple **WebService**:

```
<%@ WebService language="VB" Class="SecureService" %>

Imports System.Web.Services
Imports System

Class SecureService : Inherits WebService


    <WebMethod()> Public Function SecureTest As String
        Return "Hello from the secure web service"
    End
End Class
```

VB

You could require basic authentication for this service by making appropriate settings in IIS as follows:

1. Open the IIS MMC console.

   ```
   Start->Run "inetmgr"
   ```

2. In the left pane, expand the tree to find your virtual directory.
3. In the right pane, right-click **Secureservice.asmx**, and choose **Properties**.
4. Select the **File Security** tab. Under **Anonymous Access and Authentication Control**, click **Edit**.

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

- Disable anonymous access.
- Disable integrated Windows authentication.
- Enable basic authentication.

5. Click **OK** to save these settings and exit the MMC console.

The base **WebService** proxy class provides two properties, **Username** and **Password**, that you can use to specify the credentials with which to connect to the remote Web service. These must be set to valid Windows credentials on the Web service's computer or domain.

```
<%@ Import Namespace="SecureService" %>

<html>
<script language="VB" runat="server">

    Public Sub Page_Load(sender As Object, e As EventArgs)

        Dim s As New SecureService

        s.Credentials = New System.Net.NetworkCredential("Administrator", "test123")

        Message.Text = s.SecureTest()
    End Sub

</script>

<body>
  <h4><font face="verdana">
    <asp:Label id="Message" runat="server"/>
  </font></h4>
</body>

</html>
```

VB

The base **WebService** class also provides a **User** property of type **System.Security.Principal.IPrincipal**, which you can use to retrieve information about the client user. Again, you can authorize access to your Web service using the **Authorization** section in the ASP.NET configuration system.

Custom Authentication and Authorization with Soap Headers

Windows authentication works well for intranet scenarios, in which you are authenticating against a user in your own domain. On the Internet, however, you probably want to perform custom authentication and authorization, perhaps against a SQL database. In that case, you should pass custom credentials (such as the username and password) to your service and let it handle the authentication and authorization itself.

A convenient way to pass extra information along with a request to a XML Web service is a SOAP header. To do this, define a class that derives from **SOAPHeader** in your service, and then declare a public field of your service as that type. This is exposed in the public contract for your service, and made available to the client when the proxy is created from WebServiceUtil.exe, as in the following example:

```
Imports System.Web.Services
Imports System.Web.Services.Protocols

' AuthHeader class extends from SoapHeader
Public Class AuthHeader : Inherits SoapHeader
    Public Username As String
    Public Password As String
End Class

Public Class HeaderService : Inherits WebService
    Public sHeader As AuthHeader
    ...
End Class
```

VB

Each **WebMethod** in your service can define a set of associated headers using the **SoapHeader** custom attribute. By default, the header is required, but it is possible to define optional headers as well. The **SoapHeader** attribute specifies the name of a public field or property of the **Client** or **Server** class (referred to as a **Headers** property in this topic). **WebServices** sets the value of a **Headers** property before the method is

called for input headers, and retrieves the value when the method returns for output headers. For more information about output or optional headers see the .NET Framework SDK documentation.

```
<WebMethod(), SoapHeader("sHeader")> Public Function SecureMethod() As String

    If (sHeader Is Nothing)
        Return "ERROR: Please supply credentials"
    Else
        Return "USER: " & sHeader.Username
    End If
End Function
```

VB

A client then sets the header on the proxy class directly before making a method call that requires it, as shown in the following example:

```
Dim h As New HeaderService
Dim myHeader As New AuthHeader
myHeader.Username = "JohnDoe"
myHeader.Password = "password"
h.AuthHeader = myHeader
Dim result As String = h.SecureMethod()
```

VB

To see this code in action, run the following sample:

VB SoapHeaders.aspx

[Run Sample] | [View Source]

Section Summary

1. Securing your XML Web services on the server using Windows authentication follows exactly the same model as described for .aspx page.
2. You can also programmatically set Windows credentials using the **Username** and **Password** properties on the **WebService** proxy class.
3. Lastly, you can do custom authentication by passing credential information as **SOAPHeaders**, along with a SOAP request to the method that requires it.

Copyright 2001 Microsoft Corporation. All rights reserved.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview

Internationalization Overview

- Encoding Support
- Localization Support
- Configuration Settings
- Section Summary

Encoding Support

ASP.NET internally uses Unicode. In addition, ASP.NET utilizes the **String** class of the .NET Framework class library and the related utility functions, which are also internally Unicode. When interfacing with the outside world, ASP.NET can be configured in several ways to use a defined encoding, which includes the encoding of .aspx files, request data, and response data. For example, it is possible to store .aspx files with Unicode encoding and convert the HTML output of a page to an ANSI code page like ISO-8859-1.

Localization Support

Properties of a locale are accessible through the **CultureInfo** class. Additionally, ASP.NET tracks two properties of a default culture per thread and request: **CurrentCulture** for the default of locale-dependent functions and **CurrentUICulture** for locale-specific lookup of resource data.

The following code displays the culture values on the Web server. Note that the **CultureInfo class** is fully qualified.

```
<%@Import Namespace="System.Globalization"%>
...
<%=CultureInfo.CurrentCulture.NativeName%>
<%=CultureInfo.CurrentUICulture.NativeName%>
```

The result is as follows:

```
English (United States)
English (United States)
```

For locale-dependent data like date/time formats or currency, ASP.NET leverages the support of the .NET Framework class library in the common language runtime. Code on ASP.NET pages can use locale-dependent formatting routines like **DateTime.Format**. For example, the following code displays the current date in a long format: the first line according to the system locale, the second one according to the German ("de") locale:

```
<%=DateTime.Now.ToString("f")%>
<%=DateTime.Now.ToString("f", new System.Globalization.CultureInfo("de-DE"))%>
```

The result is as follows:

```
Friday, March 22, 2002 12:24 PM
Freitag, 22. März 2002 12:24
```

Configuration Settings

When creating ASP.NET pages or code-behind modules, developers can use the .NET Framework class library to provide features necessary for a globalized environment or to localize the application. ASP.NET also provides configuration settings to ease development and administration of ASP.NET applications.

ASP.NET utilizes configuration files to provide directory settings that are usually also inherited by subdirectories. Each file can contain a **Globalization** section in which you can specify default encodings and cultures. Values are valid if they are accepted by the related classes **Encoding** and **CultureInfo**. You can find more information about the **Encoding** and **CultureInfo** classes in the .NET Framework SDK.

```
<configuration> <system.web> <globalization fileEncoding="utf-8"
requestEncoding="utf-8" responseEncoding="utf-8" culture="en-US"
uiCulture="de-DE" /> </system.web> </configuration>
```

Within the **Globalization** section, the value of **fileEncoding** determines the way in which ASP.NET

Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

encodes .aspx files; the values of **requestEncoding** and **responseEncoding** determine the way in which request data and response data are encoded, respectively.

The attributes of the **Globalization** section in the Web.config file can also be specified on the **Page** directive (with the exception of **fileEncoding**, because it applies to the file itself). These settings are only valid for a specific page and override the settings of the Web.config file. The following sample directive specifies that the page should use French culture settings and UTF-8 encoding for the response:

```
<%@Page Culture="fr-FR" UICulture="fr-FR" ResponseEncoding="utf-8"%>
```

Note: Within a page, the culture values can be changed programmatically by setting **Thread.CurrentCulture** and **Thread.UICulture**.

Section Summary

1. ASP.NET supports a wide range of encodings for .aspx files, request data, and response data.
2. Support for locale-dependent data is provided by the **CultureInfo** class, where the two values **CurrentCulture** and **CurrentUICulture** are tracked.
3. Internationalization settings can be configured for each computer, for each directory, and for each page.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Setting Culture and Encoding

- Encodings
- Using CultureInfo
- Using RegionInfo
- Section Summary

Encodings

Internally, ASP.NET handles all string data as Unicode. By using the **ResponseEncoding** attribute in the following sample, ASP.NET is asked to also send the page with UTF-8 encoding. Note that any arbitrary encoding could be chosen without affecting the .aspx file. ASP.NET also sets the **CharSet** attribute on the **Content Type** of the HTTP header according to the value of **ResponseEncoding**. This enables browsers to determine the encoding without a metatag or having to guess the correct encoding from the content.

VB i18n_encodings.aspx

[Run Sample] | [View Source]

Note: If some characters appear as empty rectangles, you must install the additional language support for Japanese and Hebrew. To do this on a Windows 2000 platform, open **Regional Options** on the **Control Panel** and add the required language support.

The preceding sample demonstrates how to use different national character sets on the same page. The page contains English text (ASCII), German text with one umlaut character, Japanese text, and Hebrew text (uses dir="rtl"). The source for the page itself is stored with codepage-neutral UTF-8 encoding, as specified in Web.config.

```
<configuration> <system.web> <globalization fileEncoding="utf-8" ... />
</system.web> </configuration>
```

The **Page** directive specifies **ResponseEncoding** on the page itself:

```
<%@Page ... ResponseEncoding="utf-8"%>
```

Note: The **ResponseEncoding** in Web.config is also specified as UTF-8, so repeating it on the page is redundant. However, if the .aspx file is moved to a server that does not use UTF-8, the file would still specify the right encoding.

Using CultureInfo

Code on ASP.NET pages can use the **CultureInfo** class to supply localized settings. In the following sample, the properties of a culture, initially the culture of the server, is set as follows:

```
culture = CultureInfo.CurrentCulture
```

VB

If the name of a new culture is submitted, it will be used instead:

```
culture = New CultureInfo(NewCulture.Value)
```

VB

The submitted culture is set to be the new default value and some properties are displayed:

```
<%
Thread.CurrentThread.CurrentCulture = culture
```

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

```
%>
...
Current Culture is <%= CultureInfo.CurrentCulture.Name %>
(<%=Thread.CurrentThread.CurrentCulture.Name%>),
<%= CultureInfo.CurrentCulture.EnglishName
%>/<%=CultureInfo.CurrentCulture.NativeName%>,
The localized date is: <%= DateTime.Now.ToString("D", CultureInfo.CurrentCulture) %>
```

VB

VB i18n_cultureinfo.aspx

[Run Sample] | [View Source]

Using RegionInfo

Code on ASP.NET pages can also use the **RegionInfo** class to supply regional settings. In the following sample, the properties of a region are displayed. The initial display is the server's default region.

```
region = RegionInfo.CurrentRegion
...
Current region is <%= region.EnglishName %> (<%=region.DisplayName%>),
currency is <%= region.CurrencySymbol %>.
```

VB

On subsequent requests the entered region is displayed:

```
region = New RegionInfo(NewRegion.Value)
```

VB

VB I18N_Regional.aspx

[Run Sample] | [View Source]

Section Summary

1. ASP.NET can use pages that are stored with UTF-8 encoding to support different national characters.
2. The **CultureInfo** class can be set and used programmatically to localize pages.
3. The **RegionInfo** class can be used to provide regional settings on ASP.NET pages.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization

Localizing ASP.NET Applications

- Copy and Translate
- Localization and Controls
- Section Summary

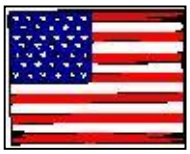
Copy and Translate

The easiest way to localize a Web page is usually to create a copy and translate it to the target language. This works well for static content that does not require a lot of maintenance. To support this model for ASP.NET pages, you can set the **Culture** attribute using the **Page** directive. All locale-dependent methods pick up the value of the **Culture** attribute.

The following sample shows how to do this for three independent, localized versions of a page. The **Culture** property is set on each page to determine the format of the date:

```
<%@Page Culture="de-DE" Language="VB" %>
...
<%=DateTime.Now.ToString("f", Nothing)%>
```

VB

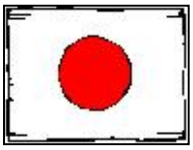


VB news-en-us.aspx

[Run Sample] | [View Source]

VB news-de.aspx

[Run Sample] | [View Source]



VB news-ja.aspx

[Run Sample] | [View Source]

Localization and Controls

An improvement over the simple copy-and-translate approach is to use controls to pick up the culture of the main page. In the following sample, the image of the flag and the search bar are controls. Depending on the culture of the hosting page, they render different content. To support this, the **UICulture** attribute is also added to each page:

```
<%@Page Culture="de-DE" UICulture="de-DE" Language="VB" %>
```

VB

The flag control (Flag.ascx), for example, just uses the culture name to build the **Src** attribute of an **<img>** tag:

```
<%@Import Namespace="System.Globalization"%>

<script runat="Server" Language="VB">
Overrides Protected Sub Render(writer As HtmlTextWriter)
    FlagImage.Src = "../../flags/" & CultureInfo.CurrentCulture.Name & ".jpg"
    FlagImage.Alt = CultureInfo.CurrentCulture.NativeName
    MyBase.Render(writer)
End Sub
</script>

<img runat="server" id="FlagImage" />
```

Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

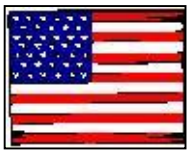
Get URL for this page

VB

The search control (Search.ascx) uses a switch statement to initialize the values of a label and a text box, but the culture name could also be the parameter for a database query:

```
Sub LocalizeSearchText()
  Select Case String.Intern(CultureInfo.CurrentUICulture.Name))
    Case "en-US"
      SearchText.Text = "Clinton"
      SearchButton.Text = "Search"

    Case "de-DE"
      ...
    Case "ja-JP"
      ...
    Case Else
      SearchButton.Text = "Search"
  End Select
End Sub
```

VB

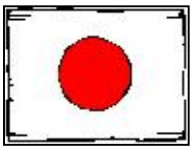


VB news-en-us.aspx

[Run Sample] | [View Source]

VB news-de.aspx

[Run Sample] | [View Source]



VB news-ja.aspx

[Run Sample] | [View Source]

Section Summary

1. ASP.NET pages support **Culture** and **UICulture** attributes to support independent localized pages.
2. Controls on pages can pick the culture of the page and can render culture-dependent content.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization

Working with Resource Files

- Creating Resources
- Using Resources on a Page
- Using Satellite Assemblies
- Using Satellite Assemblies for Controls
- Section Summary

Creating Resources

Resource management, a feature of the .NET Framework class library, can be used to extract localizable elements from source code and to store them with a string key as resources. At runtime an instance of the **ResourceManager** class can be used to resolve the key to the original resource or a localized version. Resources can be stored as independent ("loose") files or as a part of an assembly.

ASP.NET pages can utilize resource files; compiled code-behind controls can, in addition, utilize resources embedded or linked into their assembly.

Resources can be created using the **ResourceWriter** class programmatically or by the tool Resgen.exe. Resgen.exe can use a simple key=value format as input or an XML file in .resx format.

```
; ; Lines beginning with a semicolon can be used for comments. ;
[strings] greeting=Welcome ! more=Read more ... ...
```

ResourceWriter and Resgen.exe create a .resources file, which can be used as is or as part of an assembly. To include a .resources file in an assembly, use the related compiler switch or the Al.exe tool. Assemblies containing only localized resources and no code are called satellite assemblies.

Using Resources on a Page

The following sample implements only one .aspx page, which is localized for each request. The supported languages are English, German, and Japanese. The language is determined by examining the **Content-Language** field of the HTTP header in the Global.asax file. The contents of the field are accessible through the **UserLanguages** collection:

```
Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture(Request.UserLanguages(0))
```

VB

To change the initial language setting, you can use differently localized clients or change the language setting on your browser. For Internet Explorer 5.x, for example, select **Tools** -> **Internet Options** from the menu and click the **Languages** button at the bottom. In the following dialog you can add additional languages and define their priority. For simplicity the sample always chooses the first entry.

After the page is loaded the first time, the user can select another culture in the drop-down list control **MyUICulture**. If a valid culture is selected, this value overrides the setting acquired from **UserLanguages**:

```
Dim SelectedCulture As String = MyUICulture.SelectedItem.Text
If Not(SelectedCulture.StartsWith("Choose")) Then
  '  If another culture was selected, use that instead.
  Thread.CurrentThread.CurrentCulture =
CultureInfo.CreateSpecificCulture(Request.UserLanguages(0))
  Thread.CurrentThread.CurrentUICulture = Thread.CurrentThread.CurrentCulture
End If
```

VB

In the previous code, the use of the CreateSpecificCulture method is required because you cannot set the current CultureInfo of your Thread to a neutral culture. However, the string available from the UserLanguages setting may be a neutral culture. Therefore, the CreateSpecificCulture method takes this string, and makes an appropriate CultureInfo from it.

Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

Also, in the Global.asax file, a **ResourceManager** instance with application scope is initialized. This way, resources are only loaded once per application. Because resources are read-only, no lock contention should occur.

```
Public Sub Application_Start()
    Application("RM") = New ResourceManager("articles", _
                Server.MapPath("resources") + Path.DirectorySeparatorChar, _
                Nothing)
End Sub
```

VB

The resource manager then can easily be used on the page. The greeting string is simply localized by:

```
<%=rm.GetString("greeting")%>
```

global.asax

[View Source]

VB news.aspx

[Run Sample] | [View Source]

Using Satellite Assemblies

If you look at the structure of the directories in the sample in the previous section, you see that the resources for the sample are loaded not from DLLs, but from .resource files. Although this is certainly one solution, you can also compile your code into satellite assemblies. A satellite assembly is defined as an assembly with resources only, no executable code. For more information on satellite assemblies, see the section How Do I... Create Resources? .

The benefit of using satellite assemblies becomes apparent when you realize that .resources files are not shadow-copied because they are not DLLs, and therefore Web sites can encounter locking problems when using them. The alternative is to use a parallel main assembly for application resources. The main assembly contains fallback resources; the satellites (one per culture) contain localized resources. The main assembly is installed into the \bin directory, and the satellites are stored in the usual xx-XX subdirectories (see How Do I... Create Resources?). Being assemblies, they are shadow-copied and are not locked. To create an assembly-aware .asp application:

1. Create the resource DLL and copy it into the \bin directory. For example:

   ```
   resgen qq.txt qq.resources
   al /embed:qq.resources,qq.resources /out:qq.dll
   ```

   The "y" refers to whether the blob should be visible to other assemblies. Since the **ResourceManager** lives in Mscorlib and is a different assembly from "qq", the .resources file must be publically visible. The "y" says whether this should be public.

2. On your page, include the following statement. Note that the name of the assembly here is in the **System.Reflection** namespace defined in Mscorlib (which is always referenced for you when compiling):

   ```
   &LT;%
   Dim a As Assembly = Assembly.Load("qq")
   Dim rm As ResourceManager = New ResourceManager("qq", a)
   Response.Write(rm.GetString("key"))
   %>
   ```

   VB

3. Compile each satellite resource into its own assembly, placing it into the correct required directory structure within the /bin directory:

   al /embed:qq.en-US.resources,qq.en-US.resources /out:qq.resources.dll /c:en-US

   Substitute the code for the culture into which you are localizing for en-US. Remember that the **/c:** tag is the culture specifier.

After the DLLs are in the right locations (/bin and /bin/en-US in the above samples), the resources can be retrieved appropriately. Note that everything gets shadow-copied by assembly cache and thus is replaceable, avoiding potential locking scenarios.

Using Satellite Assemblies for Controls

Compiled code-behind controls can also use satellite assemblies to supply localized content. From a deployment perspective, this is an especially good thing, because satellite assemblies can be version-independent from the code. As a result, support for additional languages can be provided just by copying the module of the satellite to the server, and no code change is required.

The following sample contains the **LocalizedButton** control in the assembly **LocalizedControls** (module LocalizedControls.dll). On the page Showcontrols.aspx, the compiled control is registered and used later on:

```
<%@Register TagPrefix="Loc" namespace="LocalizedControls" %>
...
<Loc:LocalizedButton runat="server" Text="ok" />
```

The **LocalizedButton** control stores a **ResourceManager** instance, which is shared by all instances of **LocalizedButton**. Whenever a control is rendered, the value of the **Text** property is replaced with the localized version:

```
_rm = New ResourceManager("LocalizedStrings", _
                          Assembly.GetExecutingAssembly(), _
                          Nothing, _
                          True )
...
Overrides Protected Sub Render (writer As HtmlTextWriter)
    Text = ResourceFactory.RManager.GetString(Text)
    base.Render(writer)
End Sub
```

VB

The **ResourceManager** instance is responsible for resolving the key to a localized resource. If a satellite assembly with the correct culture is not available and no related culture is found, the neutral resource of the main assembly is used ("en-us" -> "en" -> neutral). Support for another language is simply granted by copying the module file for the new satellite assembly in place.

Localized Controls

[View Source]

VB Using Localized Controls

[Run Sample] | [View Source]

Section Summary

1. ASP.NET pages can utilize the resource classes to isolate localizable content in resources, which are selected at runtime.
2. A good alternative is to use satellite assemblies rather than the intermediate .resources files for

loading your resources, since this can avoid locking issues.

3. Compiled controls can contain resources of their own and will select the correct localized content, depending on the **UICulture** of the hosting page.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview

Tracing Overview

When you are developing an application, it is often helpful to be able to insert debugging print statements into your code to output variables or structures, assert whether a condition is met, or just generally trace through the execution path of the application. ASP.NET provides two levels of tracing services that make it easy to do just that.

- **Page-level Tracing**: At the page level, developers can use the **TraceContext** instrinsic to write custom debugging statements that appear at the end of the client output delivered to the requesting browser. ASP.NET also inserts some helpful statements regarding the start/end of lifecycle methods, like **Init**, **Render**, and **PreRender**, in addition to the inputs and outputs to a page, such as form and **QueryString** variables or headers, and important statistics about the page's execution (control hierarchy, session state, and application state). Because tracing can be explicitly enabled or disabled for a page, these statements can be left in the production code for a page with no impact to the page's performance. Each statement can be associated with a user-defined category for organizational purposes, and timing information is automatically collected by the ASP.NET runtime. The resulting output can be ordered by either time or category.

- **Application-level Tracing**: Application-level tracing provides a view of several requests to an application's pages at once. Like page-level tracing, it also displays inputs and outputs to a page, such as form and **QueryString** variables or headers, as well as some important statistics (control hierarchy, session state, and application state). Application-level tracing is enabled through the ASP.NET configuration system, and accessed as a special mapped URL into that application (Trace.axd). When application tracing is enabled, page-level tracing is automatically enabled for all pages in that application (provided there is no page-level directive to explicitly disable trace).

To learn more about how the Trace feature works, read the following two sections: Trace Logging to Page Output and Application-level Trace Logging.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Trace Logging to Page Output

Page-level tracing enables you to write debugging statements directly to a page's output, and conditionally run debugging code when tracing is enabled. To enable tracing for a page, include the following directive at the top of the page code:

```
<%@ Page Trace="true"%>
```

Trace statements can also be organized by category, using the **TraceMode** attribute of the **Page** directive. If no **TraceMode** attribute is defined, the default value is **SortByTime**.

```
<%@ Page Trace="true" TraceMode="SortByCategory" %>
```

The following example shows the default output when page-level tracing is enabled. Note that ASP.NET inserts timing information for important places in the page's execution lifecycle:

VB Trace1.aspx

[Run Sample] | [View Source]

The page exposes a **Trace** property (of type **TraceContext**), which can be used to output debugging statements to the page output, provided tracing is enabled. Using **TraceContext**, you can write debugging statements using the **Trace.Write** and **Trace.Warn** methods, which each take a message string or a category and message string. **Trace.Warn** statements are identical to **Trace.Write** statements, except they are output in red.

```
' Trace(Message)
Trace.Write("Begging User Code...")
...
Trace.Warn("Array count is Nothing!")
' Trace(Category, Message)
Trace.Write("Custom Trace","Beginning User Code...")
...
Trace.Warn("Custom Trace","Array count is null!")
```

VB

When tracing is disabled (that is, when **Trace="false"** on the **Page** directive, or is not present), these statements do not run and no Trace output appears in the client browser. This makes it possible to keep debugging statements in production code and enable them conditionally at a later time.

Often you might need to run additional code to construct the statements to pass to the **Trace.Write** or **Trace.Warn** methods, where this code should only run if tracing is enabled for the page. To support this, **Page** exposes a Boolean property, **Trace.IsEnabled**, which returns true only if tracing is enabled for the page. You should check this property first to guarantee that your debugging code can only run when tracing is on.

```
If Trace.IsEnabled Then
    For i=0 To ds.Tables("Categories").Rows.Count-1
        Trace.Write("ProductCategory",ds.Tables("Categories").Rows(i)(0).ToString())
    Next
End if
```

VB

The following example shows the use of **Trace.Write** and **Trace.Warn** to output debugging statements. Also note the use of the **Trace.IsEnabled** property to conditionally run extra debugging code. In this example, the trace information has been sorted by category.

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

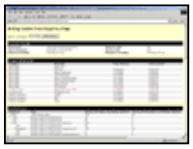
Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page



VB Trace2.aspx

[Run Sample] | [View Source]

ASP.NET also provides a way to enable tracing for the entire application, not just a single page. For more about application-level tracing, click here.

Section Summary

1. Page-level tracing is enabled using a **Trace="true"** attribute on the top-level **Page** directive.
2. Page-level tracing enables you to write debugging statements as part of a page's client output. Trace statements are output using the **Trace.Write** and **Trace.Warn** methods, passing a category and message for each statement.
3. Debugging code can be conditionally run, depending on whether tracing is enabled for the page. Use the **Trace.IsEnabled** property of the page to determine whether tracing is enabled.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

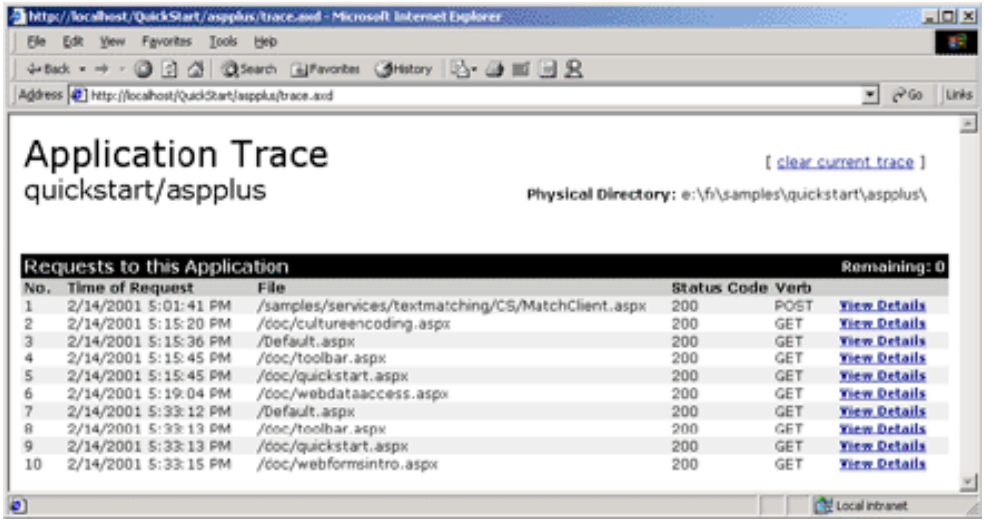
Security

Application-level Trace Logging

In addition to the Page-level Trace funtionality, ASP.NET provides a way to enable trace output for an entire application. Enabling Trace at the application level has the effect of enabling Page-level Trace for every page within that application (provided there is no page-level directive to explicitly disable trace). When application-level tracing is enabled, the ASP.NET runtime also collects several additional statistics, such as the state of the control hierarchy, the contents of session and application state, the form and querystring input values, and other characteristics of request's execution. These statistics are collected for a specified number of requests as determined by the application's configuration file. To enable tracing for an application, place the following in the application's web.config file at the application root directory:

```
<configuration>
  <system.web>
    <trace enabled="true"/>
  </system.web>
</configuration>
```

Using the above configuration, each page in the application will run its page-level trace statements to be output in the client browser. To access the additional page statistics, request a specially-mapped "trace.axd" URL from the application root. For example, if the URL to your application is http://localhost/myapplication, you would request the URL http://localhost/myapplication/trace.axd to access the trace statistics for that application.



By default, trace information will be collected for up to 10 requests (you can use the "clear current trace" link to reset the request counter). The trace section of the configuration file also supports an attribute for controlling whether trace statements are output to the client browser, or whether they are only available from trace.axd. The attributes supported in the trace configuration section are listed in the table below:

Value	Description
enabled	Set to true | false, indicates whether Tracing is enabled for the application (default is false)

Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

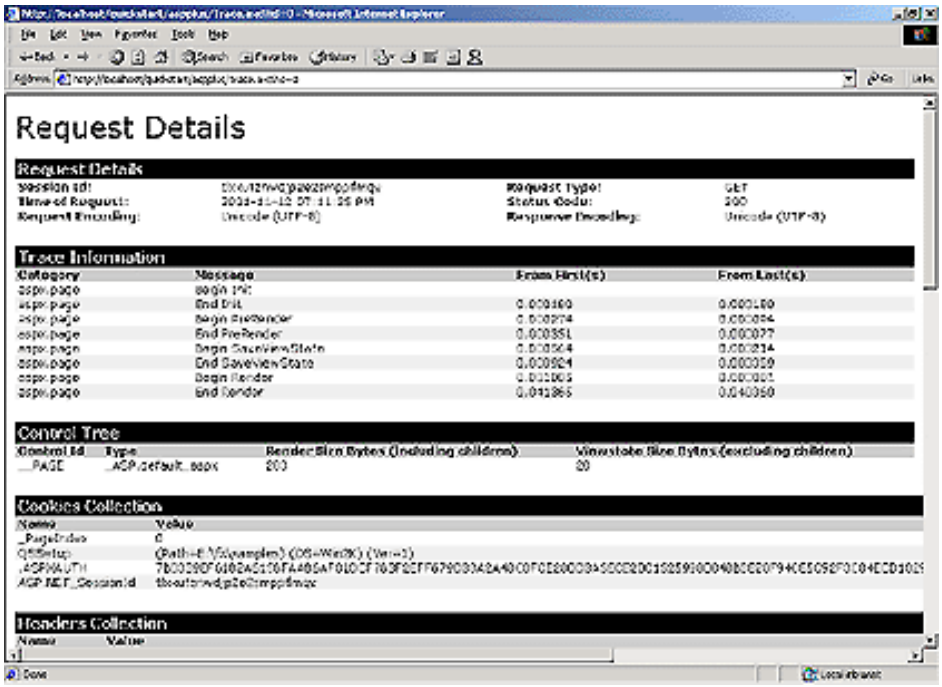
pageOutput	Set to true | false, indicates whether trace information should be rendered at the end of each page - or only accessible via the trace.axd utility (default is false)
requestLimit	Number of trace requests to store on the server (default is 10)
traceMode	Set to SortByTime | SortByCategory, indicates the display order for Trace messages (default is SortByTime)
localOnly	Set to true | false, indicates whether Tracing is enabled for localhost users or for all users (default is true)

For example, the following configuration collects trace information for up to 40 requests, and prevents trace statements from being output to the requesting browser (provided there is no page-level directive to explicitly enable trace). The messages are displayed in order of category:

```
<configuration>
  <system.web>
    <trace
      enabled="true"
      traceMode="SortByCategory"
      requestLimit="40"
      pageOutput="false"
      localOnly="true"
    />
  </system.web>
</configuration>
```

Application Trace Request Details

After making a series of requests to the application, accessing trace.axd will list those requests in time-order. You can drill-down into the details for each request by selecting the "View Details" link.



The trace application presents the following detailed information for each request:

Request Detail	
Value	**Description**
Session Id	The Session Id for this request
Time of Request	The time the request was made
Status Code	The returned status code for this request
Request Type	GET | POST
Request Encoding	Encoding for the request
Response Encoding	Encoding for the response

Trace Information	
Value	**Description**
Category	The category for a Trace statement written to the TraceContext
Message	The message string for this Trace statement
From First (s)	Time in seconds from the first Trace statement
From Last (s)	Time in seconds from the previous Trace statement

Control Hierarchy	
Value	**Description**
Control ID	The ID for the control
Type	The fully qualified type of the control

Render Size	The size of the control's rendering in bytes including children
ViewState Size	The size of the control's viewstate in bytes excluding children

Session State	
Value	**Description**
Key	The key for an object in Session State
Type	The fully qualified type of the object
Value	The value of the object

Application State	
Value	**Description**
Key	The key for an object in Application State
Type	The fully qualified type of the object
Value	The value of the object

Cookies Collection	
Value	**Description**
Name	The name of the cookie
Value	The value of the cookie, or sub-keys/values if multi-valued
Size	The size of the cookie rendering in Bytes

Headers Collection	
Value	**Description**
Name	The name of the header
Value	The value of the header

Form Collection	
Value	**Description**
Name	The name of the form variable
Value	The value of the form variable

QueryString Collection	
Value	**Description**
Name	The name of the querystring variable
Value	The value of the querystring variable

Server Variables	
Value	**Description**
Name	The name for the server variable
Value	TThe value of the server variable

Section Summary

1. Application-level Tracing is enabled using a "trace" section in the configuration file at the application root directory.
2. Application-level Tracing enables trace log output for every page within an application (provided there is no page-level directive to explicitly disable trace).
3. After making a series of requests, details for those requests may be accessed by requesting "trace.axd" from the application root.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview

The Microsoft .NET Framework SDK Debugger

No matter how skilled a programmer you are, you are bound to make mistakes once in a while. Tracking down problems in your code can be baffling without the appropriate tool. Fortunately, the compiled nature of ASP.NET means that debugging Web applications is no different than debugging any other managed applications, and the .NET Framework SDK includes a lightweight debugger that is perfectly suited for this task.

This section describes the steps required to debug ASP.NET Framework applications using the debugger provided in this SDK. The debugger supports manual-attach debugging of processes on a local development computer. The debugger documentation included in this SDK is your best resource for information about specific features.

Enabling Debug Mode for ASP.NET Applications

Because many parts of an ASP.NET Framework application are dynamically compiled at runtime (.aspx and .asmx files, for example), you must configure the ASP.NET runtime to compile the application with symbolic information before the application can be debugged. Symbols (.pdb files) tell the debugger how to find the original source files for a binary, and how to map breakpoints in code to lines in those source files. To configure an application to compile with symbols, include a **debug** attribute on the **compilation** section within the **system.web** group of the Web.config file at the application's root directory, as follows:

```
<configuration>
  <compilation debug="true"/>
</configuration>
```

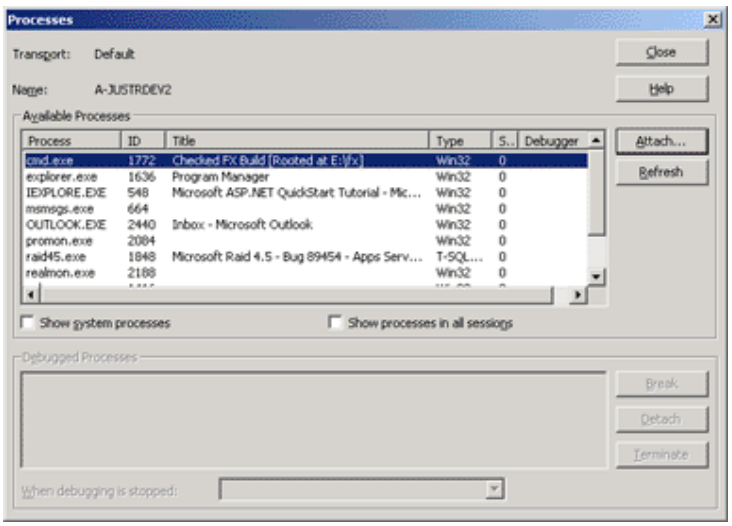
Important: You should only enable this setting when you are debugging an application, because it can significantly affect application performance.

Debugging ASP.NET Applications

When you have enabled debugging for the application, you should issue a request to the page you want to debug. This ensures that the ASP.NET runtime process (Aspnet_wp.exe) is created and the application is loaded into memory.

To begin debugging:

1. Launch the .NET Framework debugger, DbgClr.exe.
2. Use the **File...Miscellaneous Files...Open File** menu to open the source file for the page you want to debug.
3. From the **Tools** menu, choose **Debug Processes**. The screen in the figure following these instructions will appear.
4. Check the **Show system processes** checkbox, if it is not checked.
5. Find the Aspnet_wp.exe process and double-click it to bring up the **Attach to Process** dialog.
6. Make sure your application appears in the list of running applications, and select **OK** to attach.
7. Close the **Programs** dialog.



Important: When you attach to the Aspnet_wp.exe process, all threads in that process are frozen. Under

Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

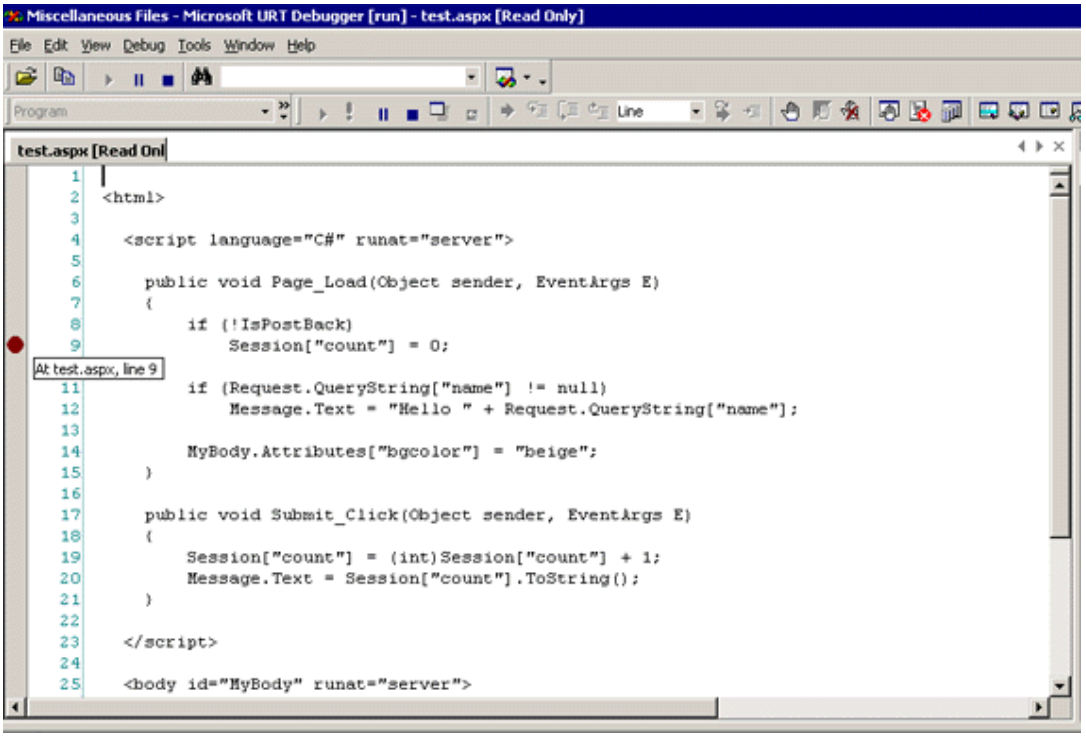
Get URL for this page

no circumstances should you attempt to debug a live production application, because client requests can not execute normally until the debugger is detached.

Setting Breakpoints

To set a breakpoint in your page, click the left-hand margin on a line containing an executable statement or function/method signature. A red dot appears where the breakpoint is set. Move the mouse over the breakpoint to ensure that it is appropriately mapped to the correct application instance in the Aspnet_wp.exe process.

Reissue the request to the page from your browser. The debugger will stop at the breakpoint and gain the current window focus. From this point, you can step, set variable watches, view locals, stack information, disassembly, and so on. You can see the intrinsic objects on the page, like **Request**, **Response**, and **Session** by using this (C#) or Me (VB) in the watch window.



Generating Symbols for Pre-Compiled Components

To debug pre-compiled components, such as business objects or code-behind files, you must compile with symbolic information prior to debugging. Symbols for assemblies are typically found by means of a path-based search algorithm. The algorithm used by the PDB library (Mspdb70.dll) to find symbolic information is as follows:

1. Search the same path as the assembly. This is the normal location for .pdb files.
   For local assemblies, place the symbols (.pdb files) in the application's /bin directory with the DLLs.
2. Search path as specified in the PE file (the NB10 debug header).
3. Search NT symbol file locations (environment variables **_NT_SYMBOL_PATH** and **_NT_ALT_SYMBOL_PATH**).

Note: If symbolic information cannot be found, the debugger prompts for a user-specified location.

Section Summary

1. The debugger described in this section supports manual-attach debugging of processes on a local development computer.
2. Debugging allows the ASP.NET runtime to dynamically compile with symbolic information. Enable this by setting <compilation debug="true"/> in the Web.config file located in the application's root directory. The debugger setting should only be enabled when you are debugging an application, because it degrades application performance.
3. To debug an application, issue a request to a page, attach the debugger to the Aspnet_wp.exe process, set breakpoints, and reissue the page request.

4. When attached to the Aspnet_wp.exe process, all threads in that process are frozen. Under no circumstances should you debug a live production application, since client requests can not execute normally until the debugger is detached.
5. To debug pre-compiled components, such as business objects or code-behind files, you must compile with symbolic information prior to debugging.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview

Performance Overview

Feature-rich web applications are not very useful if they cannot perform well. The demands of the Web are so great that code is expected to do more in less time than ever before. This section describes some key principles of Web application performance, tips for writing code that performs well, and tools for measuring performance.

ASP.NET provides a number of built-in performance enhancements. For example, pages are compiled only once and cached for subsequent requests. Because these compiled pages are saved to disk, even a complete server restart does not invalidate them. ASP.NET also caches internal objects, such as server variables, to speed user code access. Further, ASP.NET benefits from all of the performance enhancements to the common language runtime: just-in-time compiling, a fine-tuned common language runtime for both single- and multiprocessor computers, and so on.

However, all of these enhancements cannot protect you from writing code that does not perform well. Ultimately, you must ensure that your application can meet the demands of its users. The next section describes a few of the common ways to avoid performance bottlenecks. However, first you need to understand the following metrics:

- **Throughput**: The number of requests a Web application can serve per unit of time, often measured in requests/second. Throughput can vary, depending on the load (number of client threads) applied to the server. This is usually considered the most important performance metric to optimize.

- **Response Time**: The length of time between the issuance of a request and the first byte returned to the client from the server. This is often the most perceptable aspect of performance to the client user. If an application takes a long time to respond, the user can become impatient and go to another site. The response time of an application can vary independently of (even inversely to) the rate of throughput.

- **Execution Time**: The time it takes to process a request, usually measured between the first byte

Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications

and the last byte returned to the client from the server. Execution time directly affects the throughput calculation.

- **Scalability**: The measurement of an application's ability to perform better as more resources (memory, processors, or computers) are allocated to it. Often, it is a measurement of the rate of change of throughput with respect to the number of processors.

Writing applications that perform well is all about striking a balance between these metrics. No single measurement can characterize how your application will behave under varying circumstances, but several measurements taken together can paint a reasonable picture of an application's performance.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output

Performance Tuning Tips

Any programming model has its common performance pitfalls, and ASP.NET is no exception. This section describes some of the ways in which you can avoid performance bottlenecks in your code.

1. **Disable Session State when not in use**: Not all applications or pages require per-user session state. If it is not required, disable it completely. This is easily accomplished using a page-level directive, such as the following:

   ```
   <%@ Page EnableSessionState="false" %>
   ```

   Note: If a page requires access to session variables but does not create or modify them, set the value of the directive to ReadOnly. Session State can also be disabled for XML Web service methods. See Using Objects and Intrinsics in the XML Web services section.

2. **Choose your Session State provider carefully**: ASP.NET provides three distinct ways to store session data for your application: in-process session state, out-of-process session state as a Windows Service, and out-of-process session state in a SQL database. Each has its advantages, but in-process session state is by far the fastest solution. If you are only storing small amounts of volatile data in session state you should use the in-process provider. The out-of-process solutions are primarily useful in Web garden and Web farm scenarios or in situations in which data cannot be lost in the event of a server/process restart.

3. **Avoid excessive round trips to the server**: The Web Forms page framework is one of the best features of ASP.NET, because it can dramatically reduce the amount of code you need to write to accomplish a task. Programmatic access to page elements using server controls and the postback event handling model are arguably the most time-saving features. However, there are appropriate and inappropriate ways to use these features, and it is important to know when it is appropriate to use them.

   An application typically needs to make a round trip to the server only when retrieving data or storing data. Most data manipulations can take place on the client between round trips. For example, validating form entries can often take place on the client before the user submits data. In general, if you do not need to relay information back to the server, then you should not make a round trip to the server.

   If you are writing your own server controls, consider having them render client-side code for up-level (ECMAScript-capable) browsers. By employing "smart" controls, you can dramatically reduce the number of unecessary hits to your Web server.

4. **Use Page.IsPostback to avoid extra work on a round trip**: If you are handling server control postbacks, you often need to execute different code the first time the page is requested from the code you do use for the round trip when an event is fired. If you check the **Page.IsPostBack** property, your code can execute conditionally, depending on whether there is an initial request for the page or a responce to a server control event. It might seem obvious to do this, but in practice it is possible to omit this check without changing the behavior of the page. For example:

   ```
   <script language="VB" runat="server">

       Public ds As DataSet
       ...

       Sub Page_Load(sender As Object, e As EventArgs)
           ' ...set up a connection and command here...
           If Not (Page.IsPostBack)
               Dim query As String = "select * from Authors where FirstName like
   '%JUSTIN%'"
               myCommand.Fill(ds, "Authors")
               myDataGrid.DataBind()
           End If
       End Sub

       Sub Button_Click(sender As Object, e As EventArgs)
           Dim query As String = "select * from Authors where FirstName like '%BRAD%'"
           myCommand.Fill(ds, "Authors")
           myDataGrid.DataBind()
       End Sub

   </script>

   <form runat="server">
     <asp:datagrid datasource='<%# ds.Tables["Authors"].DefaultView %>'
   runat="server"/><br>
     <asp:button onclick="Button_Click" runat="server"/>
   </form>
   ```

   VB

   The `Page_Load` event executes on every request, so we checked **Page.IsPostBack** so that the first query does not execute when we process the `Button_Click` event postback. Note that even without this check our page

Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

would behave identically, since the binding from the first query would be overturned by the call to **DataBind** in the event handler. Keep in mind that it can be easy to overlook this simple performance improvement when you write your pages.

5. **Use server controls sparingly and appropriately**: Even though it is extremely easy to use, a server control might not always be the best choice. In many cases, a simple rendering or databinding substitution will accomplish the same thing. For example:

```
<script language="VB" runat="server">

    Public imagePath As String
    Sub Page_Load(sender As Object, e As EventArgs)
        '...retrieve data for imagePath here...
        DataBind()
    End Sub

</script>

<%--the span and img server controls are unecessary...--%>
The path to the image is: <span innerhtml='<%# imagePath %>' runat="server"/><br>
<img src='<%# imagePath %>' runat="server"/>

<br><br>

<%-- use databinding to substitute literals instead...--%>
The path to the image is: <%# imagePath %><br>
<img src='<%# imagePath %>' />

<br><br>

<%-- or a simple rendering expression...--%>
The path to the image is: <%= imagePath %><br>
<img src='<%= imagePath %>' />
```

VB

In this example, a server control is not needed to substitute values into the resulting HTML sent back to the client. There are many other cases where this technique works just fine, even in server control templates. However, if you want to programmatically manipulate the control's properties, handle events from it, or take advantage of its state preservation, then a server control would be appropriate. You should examine your use of server controls and look for code you can optimize.

6. **Avoid excessive server control view state**: Automatic state management is a feature that enables server controls to re-populate their values on a round trip without requiring you to write any code. This feature is not free however, since the state of a control is passed to and from the server in a hidden form field. You should be aware of when **ViewState** is helping you and when it is not. For example, if you are binding a control to data on every round trip (as in the datagrid example in tip #4), then you do not need the control to maintain it's view state, since you will wipe out any re-populated data in any case.

   ViewState is enabled for all server controls by default. To disable it, set the **EnableViewState** property of the control to false, as in the following example:

```
<asp:datagrid EnableViewState="false" datasource="..." runat="server"/>
```

   You can also turn **ViewState** off at the page level. This is useful when you do not post back from a page at all, as in the following example:

```
<%@ Page EnableViewState="false" %>
```

   Note that this attribute is also supported by the **User Control** directive. To analyze the amount of view state used by the server controls on your page, enable tracing and look at the View State column of the Control Hierarchy table. For more information about the Trace feature and how to enable it, see the Application-level Trace Logging feature.

7. **Use Response.Write for String concatenation**: Use the HttpResponse.Write method in your pages or user controls for string concatenation. This method offers buffering and concatenation services that are very efficient. If you are performing extensive concatenation, however, the technique in the following example, using multiple calls to **Response.Write**, is faster than concatenating a string with a single call to the **Response.Write** method.

```
Response.Write("a")
Response.Write(myString)
Response.Write("b")
Response.Write(myObj.ToString())
Response.Write("c")
Response.Write(myString2)
Response.Write("d")
```

VB

8. **Do not rely on exceptions in your code**: Exceptions are very expensive and should rarely occur in your code. You should never use exceptions as a way to control normal program flow. If it is possible to detect in code a condition that would cause an exception, you should do that instead of waiting to catch the exception before handling that condition. Common scenarios include checking for null, assigning to a string that will be parsed into a numeric value, or checking for specific values before applying math operations. For example:

```
' Consider changing this:

Try
   result = 100 / num

Catch (e As Exception)
  result = 0
End Try

// To this:

If Not (num = 0)
   result = 100 / num
Else
  result = 0
End If
```

VB

9. **Use early binding in Visual Basic or JScript code**: One of the advantages of Visual Basic, VBScript, and JScript is their typeless nature. Variables can be created simply by using them and need no explicit type declaration. When assigning from one type to another, conversions are performed automatically, as well. This can be both an advantage and a disadvantage, since late binding is a very expensive convenience in terms of performance.

   The Visual Basic language now supports type-safe programming through the use of a special **Option Strict** compiler directive. For backward compatibility, ASP.NET does not enable **Option Strict** by default. However, for optimal perfomance, you should enable **Option Strict** for your pages by using a **Strict** attribute on the page or Control directive:

```
<%@ Page Language="VB" Strict="true" %>

<%

Dim B
Dim C As String

' This causes a compiler error:
A = "Hello"

' This causes a compiler error:
B = "World"

' This does not:
C = "!!!!!!"

' But this does:
C = 0

%>
```

   JScript also supports typeless programming, though it offers no compiler directive to force early binding. A variable is late-bound if:

   - It is declared explicitly as an object.
   - It is a field of a class with no type declaration.
   - It is a private function/method member with no explicit type declaration and the type cannot be inferred from its use.

   The last distinction is complicated. The JScript compiler optimizes if it can figure out the type, based on how a variable is used. In the following example, the variable A is early-bound but the variable B is late-bound:

```
var A;
var B;

A = "Hello";
B = "World";
B = 0;
```

For the best performance, declare your JScript variables as having a type. For example, "var A : String".

10. **Port call-intensive COM components to managed code**: The .NET Framework provides a remarkably easy way to interoperate with traditional COM components. The benefit is that you can take advantage of the new platform while preserving your existing code. However, there are some circumstances in which the performance cost of keeping your old components is greater than the cost to migrate your components to managed code. Every situation is unique, and the best way to decide what needs to be changed is to measure site performance. In general, however, the performance impact of COM interoperability is proportional to the number of function calls made or the amount of data marshaled from unmanaged to managed code. A component that requires a high volume of calls to interact with it is called "chatty," due to the number of communications between layers. You should consider porting such components to fully managed code to benefit from the performance gains provided by the .NET platform. Alternatively, you might consider redesigning your component to require fewer calls or to marshal more data at once.

11. **Use SQL stored procedures for data access**: Of all the data access methods provided by the .NET Framework, SQL-based data access is the best choice for building scalable web applications with the best performance. When using the managed SQL provider, you can get an additional performance boost by using compiled stored procedures instead of ad hoc queries. For an example of using SQL stored procedures, refer to the Server-Side Data Access section of this tutorial.

12. **Use SqlDataReader for a fast-forward, read-only data cursor**: A **SqlDataReader** object provides a forward, read-only cursor over data retrieved from a SQL database. **SqlDataReader** is a more performant option than using a **DataSet** if it can be used for your scenario. Because **SqlDataReader** supports the **IEnumerable** interface, you can even bind server controls, as well. For an example of using **SqlDataReader**, see the Server-Side Data Access section of this tutorial.

13. **Cache data and output wherever possible**: The ASP.NET programming model provides a simple mechanism for caching page output or data when it does not need to be dynamically computed for every request. You can design your pages with caching in mind to optimize those places in your application that you expect to have the most traffic. More than any feature of the .NET Framework, the appropriate use of caching can enhance the performance of your site, sometimes by an order of magnitude or more. For more information about how to use caching, see the Cache Services section of this tutorial.

14. **Enable Web gardening for multiprocessor computers**: The ASP.NET process model helps enable scalability on multiprocessor machines by distributing the work to several processes, one for each CPU, each with processor affinity set to its CPU. The technique is called Web gardening, and can dramatically improve the performance of some applications. To learn how to enable Web gardening, refer to the Using the Process Model section.

15. **Do not forget to disable Debug mode**: The **<compilation>** section in ASP.NET configuration controls whether an application is compiled in Debug mode, or not. Debug mode degrades performance significantly. Always remember to disable Debug mode before you deploy a production application or measure performance. For more information about Debug mode, refer to the section entitled The SDK Debugger.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview

Measuring Performance

Measuring Web server performance is a skill that can only be refined by repeated experience and experimentation. There are many variables at play, such as the number of clients, speed of client connections, server resources, application code, and so on. It helps to have good tools at your disposal, and fortunately those are available.

Microsoft provides the Web Application Stress (WAS) tool, which simulates multiple HTTP clients hitting your Web site. You can control the client load, number of connections, format of cookies, headers, and several other parameters from the tool's graphical interface. After a test run, WAS provides you with reports containing performance metrics such as response time, throughput, and performance counter data relevant to your application. The goal is simple: to maximize throughput and CPU utilization under high degrees of load. WAS is available from the Microsoft Internet Information Server Resource Kit and is also downloadable separately from http://webtool.rte.microsoft.com.

ASP.NET also exposes a number of performance counters that can be used to track the execution of your applications. Unlike traditional ASP, most of these performance counters are exposed per-application, instead of globally for the entire machine. The per-application counters are available under the ASP.NET Framework applications performance object, and you need to select a particular application instance when selecting a counter to monitor. Of course, you can still see the counter values for all applications using a special "__Total__" application instance in System Monitor. ASP.NET also exposes global-only counters which are not bound to a particular application instance. These counters are located under the ASP.NET System performance object. To view all available counters for ASP.NET (on Windows 2000 systems):

1. Select **Start->Programs->Administrative Tools->Performance**.
2. Click the **View Report** button in System Monitor.
3. Click the **Add** button.
4. Select **ASP.NET Applications**, then choose the **All counters** radio button. Click **OK**.
5. Select **ASP.NET**, then choose the **All counters** radio button. Click **OK**.

Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications

The ASP.NET Trace feature is also useful for identifying performance bottlenecks in your code. It can show you important timing information between successive trace output statements, as well as information about the server control heierarchy, the amount of viewstate used, and the render size of controls on your page. For more information about the Trace feature, refer to the Tracing section of this tutorial.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview

Migration Overview

Installing ASP.NET will not break your existing ASP applications. It uses a separate file name extension (.aspx instead of .asp), separate configuration settings, and an entirely separate common language runtime (Asp.dll has not been modified). ASP pages and applications can continue to use the existing ASP engine, with no interference from ASP.NET. That said, the benefits of migrating your existing applications to ASP.NET are enormous. ASP.NET easily provides many times the features of traditional ASP, and moving your ASP applications to the new platform provides a huge opportunity for improvement. Among the new features you can take advantage of are:

- Improved performance and scalability
- Web farm support and XCopy deployment
- Output caching and custom security
- Web Forms page controls
- XML Web services infrastructure

ASP.NET is designed to help preserve your investment in traditional ASP and COM technologies. It balances support for existing ASP syntax and semantics with the need for a forward-looking platform that can last well into the next age of Internet application development. While ASP.NET preserves the majority of ASP's feature set, 100% compatibility between the two was not possible if the platform was to move forward, so there are a few changes to the old way of doing things.

The good news is that your ASP skills will translate easily to ASP.NET. There are only a few differences, which are usually easy to fix. However, migrating ASP applications to ASP.NET does require some work. Relatively simple pages might migrate without any changes, but more complex applications probably will require some modifications. The following sections describe the changes and the ways in which they might affect your existing application code. They also demonstrate some of the ways in which you can reuse ASP and COM code in ASP.NET.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication

Syntax and Semantics

ASP.NET is fully API-compatible with traditional ASP, with the following three exceptions:

- **Request()**: ASP returns an array of strings; ASP.NET returns a string.
- **Request.QueryString()**: ASP returns an array of strings; ASP.NET returns a string.
- **Request.Form()**: ASP returns an array of strings; ASP.NET returns a string.

In ASP, the **Request**, **Request.QueryString**, and **Request.Form** collections return string arrays from lookups. For example, in traditional ASP the query string values from a request to http://localhost/test/Test.asp?values=45&values=600 would be accessed as follows:

```
<%
    ' Below line outputs: "45, 600"
    Response.Write Request.QueryString("values")

    ' Below line outputs: "45"
    Response.Write Request.QueryString("values")(1)
%>
```

In ASP.NET, these collections require an explicit method to get array access. These arrays are also now 0-index based. For example, in ASP.NET the query string values from a request to http://localhost/test/Test.aspx?values=45&values=600 would be accessed as follows:

```
<%
    ' Below line outputs: "45, 600"
    Response.Write(Request.QueryString("values"))

    ' Below line outputs: "45"
    Response.Write(Request.QueryString.GetValues("values")(0))
%>
```

VB

These arrays are most commonly used when form values are posted from multiselect list boxes (<select multiple>) or when multiple check boxes have the same name.

Semantic Differences Between ASP.NET and ASP

ASP.NET pages also have several semantic changes from existing ASP pages. The following issues are the ones most likely to affect you:

- **ASP.NET pages only support a single language.**

  ASP allowed multiple languages to be used on a single page, which was useful for script library scenarios. Because of ASP.NET's compiled nature, it supports only a single language on a page. However, it is still possible to have multiple pages, each with a separate language, within a single application. User Controls might also have a different language from the page that contains them. This enables you to integrate functionality written in different languages in a single page. This is an adequate substitute for the multiple-language Include files that are prevalent in traditional ASP applications.

- **ASP.NET page functions must be declared in <script runat=server> blocks.**

  In ASP, page functions could be declared within <% %> blocks:

  ```
  <%
      Sub DoSomething()
          Response.Write "Hello World!"
  ```

Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

```
    End Sub

    DoSomething
%>
```

In ASP.NET, page functions must be declared in <script runat=server> blocks:

```
<script language="VB" runat=server>

    Sub DoSomething()
        Response.Write ("Hello World!")
    End Sub

</script>

<%
    DoSomething()
%>
```

VB

- **ASP.NET does not support page-render functions.**

In ASP, page-render functions could be declared with <% %> blocks:

```
<% Sub RenderSomething() %>
       <font color="red"> Here is the time: <%=Now %> </font>
<% End Sub %>

<%
   RenderSomething
   RenderSomething
%>
```

In ASP.NET, this must be rewritten:

```
<script language="VB" runat=server>

     Sub RenderSomething()
        Response.Write("<font color=red> ")
        Response.Write("Here is the time: " & Now)
     End Sub

</script>

<%
   RenderSomething()
   RenderSomething()
%>
```

VB

Section Summary

1. With three exceptions, ASP.NET is 100% API-compatible with traditional ASP. The API changes are that, now, **Request()**, **Request.QueryString()**, and **Request.Form()** all return individual strings, rather than string arrays.
2. ASP.NET pages support only a single language.
3. ASP.NET page functions must be declared in <script runat=server> blocks.
4. Page-render functions are not supported.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview

Language Compatibility

The differences between the VBScript used in ASP and the Visual Basic .NET language used in ASP.NET are by far the most extensive of all the potential migration issues. Not only has ASP.NET departed from the VBScript language to "true" Visual Basic, but the Visual Basic language itself has undergone significant changes in this release. The changes are designed to:

- Make the language more consistent by bringing together features of the language with similar purposes.
- Simplify the language by redesigning the features that made Visual Basic less than "basic."
- Improve readability and maintainability by redesigning features that hid too many important details from the programmer.
- Improve robustness by enforcing better practices, such as type-safe programming.

This section highlights some common issues you are likely to encounter when you begin to use the new Visual Basic language.

- **No more Set and Let.** Instead, use simple variable assignment.

```
<%
    ' Old ASP syntax.
    Dim MyConn
    Set MyConn = Server.CreateObject("ADODB.Connection")

    ' New ASP.NET syntax.
    Dim MyConn
    MyConn = Server.CreateObject("ADODB.Connection")
%>
```

- **No more non-indexed default properties.** Non-indexed default properties enable an expression that normally refers to an object to refer to a default property of the object instead. The unfortunate consequence of support for default properties is that it makes programs more difficult to read, since the meaning of an expression depends on its context. In Visual Basic .NET, non-indexed properties must always be specified explicitly within code.

```
<%
' Old ASP syntax (retrieving recordset column value).
Set MyConn = Server.CreateObject("ADODB.Connection")
MyConn.Open("TestDB")
Set RS = MyConn.Execute("Select * from Products")
Response.Write RS("Name")

' New ASP.NET syntax (retrieving recordset column value).
MyConn = Server.CreateObject("ADODB.Connection")
MyConn.Open("TestDB")
RS = MyConn.Execute("Select * from Products")
Response.Write RS("Name").Value
%>
```

Indexed default properties are still supported:

```
<%
Dim RS As RecordSet
```

Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

```
' This is allowed (indexed).
RS.Fields(1).Value = RS.Fields(2).Value

' But these are not allowed (non-indexed).
RS(1) = RS(2)
RS(1).Value = RS(2).Value
%>
```

- **Parentheses are now required for calling subroutines.** Visual Basic now supports exactly the same syntax for calling subroutines and functions.

```
' Note parentheses with Response.Write.
Sub DoSomething()
     Response.Write("Hello World!")
End Sub

' Note parenthesws with DoSomething.
DoSomething()
```

- **The new default is by-value arguments.** In Visual Basic 6, if a user does not explicitly specify `ByVal` or `ByRef` on a parameter declaration, the calling convention defaults to `ByRef`. In the new Visual Basic .NET, the default is `ByVal`. This applies both to regular parameters for which the default can be overridden by explicitly specifying `ByRef` and to parameters passed to a ParamArray parameter where the default can not be overridden. This has been changed because it is much more common for a parameter to be used solely for passing a value into a procedure than for altering a passed-in variable. Changing the default to `ByVal` increases performance and decreases the likleihood of accidental side-effects.

  You can still use by-reference arguments by explicitly using the `ByRef` modifier:

```
<script language="VB" runat=server>

   Sub DoSomething(ByRef value)
      value = 4343
   End Sub

</script>

<%
   Dim number = 55
   DoSomething (number)
   Response.Write ("Number: " & number)
%>
```

Note: There are many additional differences between Visual Basic 6 and Visual Basic .NET. Consult the language documentation for more information.

Section Summary

1. The differences between the VBScript used in ASP and the Visual Basic .NET language used in ASP.NET are by far the most extensive of all the potential migration issues. The changes have been made to simplify the language and improve consistency, readability, maintainability, and robustness.
2. Set and Let assignments are no longer supported in Visual Basic .NET. Use standard variable assignment instead.
3. Non-indexed default properties are not supported in Visual Basic .NET. Indexed default properties are still supported.

4. Parentheses are required for calling subroutines in Visual Basic .NET.
5. The new default is by-value arguments. You can still use by-reference arguments by explicitly using the `ByRef` modifier.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

COM Interoperability

The common language runtime enables .NET objects to interoperate seamlessly with traditional COM components. ASP.NET exposes the familiar **Server.CreateObject(ProgId)** API to developers for creating late-bound references to COM.

```
Dim myConn
myConn = Server.CreateObject("ADODB.Connection");
```

You can also use early-bound, traditional COM components by creating runtime callable wrappers (RCWs), which optimize the performance of calls between unmanaged and managed code. You can create an RCW using the Tlbimp.exe utility included in the .NET Framework SDK. For more information on Tlbimp.exe, see the Interoperability section of the Common Tasks QuickStart. The ASP.NET Performance section contains more information comparing late binding with early binding.

Like ASP, you can also create traditional COM components using the `<object>` tag with either a `progid` or a `classid` attribute. In addition to using the `<object>` tag in pages, you can also use it in the Global.asax file for the application. In this case, the object is added to the **Page.Application.StaticObjects** collection and can be accessed programmatically by simply using its `id` attribute. Note that you cannot create single-threaded apartment (STA) objects statically in the Global.asax file because doing so generates a runtime error, as it does in ASP.

ASP.NET also continues to support the existing ASP intrinsic interfaces **ObjectContext Intrinsic Flow**, **OnStartPage**, and **OnEndPage**. Supporting these interfaces means that you can use existing components (Commerce Server, Exchange, and so on) in ASP.NET pages. These interfaces are not enabled by default but are explicitly turned on using the following page directive:

```
<%@ Page ASPCompat="true" %>
```

This directive causes ASP.NET to create unmanaged ASP intrinsic objects and pass them to COM components used in the page. It also runs the page in an STA thread pool. See the following section for information.

Performance Considerations

In ASP.NET, the thread pool is a multithreaded apartment (MTA) by default, which can affect the performance of traditional apartment-threaded Visual Basic 5 and Visual Basic 6 components. The `ASPCompat="true"` attribute enables an STA

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

thread pool to address performance with existing Visual Basic components on a per-page basis.

Calling between managed and unmanaged components also incurs a marshaling cost, which can impede the performance of your pages. Every scenario yields different performance characteristics, so it is important to test adequately before deciding whether interoperability is the right choice for your application. However, in nearly all scenarios, rewriting your COM components in managed code provides performance benefits. See the ASP.NET Performance section for more information and important tips.

Section Summary

1. ASP.NET exposes the familiar **Server.CreateObject** API to developers for creating late-bound references to COM.
2. You can also use early-bound, traditional COM components by creating runtime callable wrappers, which optimize the performance of calls between unmanaged and managed code.
3. ASP.NET continues to support the existing ASP intrinsic interfaces **ObjectContext Intrinsic Flow**, **OnStartPage**, and **OnEndPage**. These interfaces are explicitly enabled using the page directive `<%@ Page ASPCompat="true" %>`.
4. The `ASPCompat="true"` attribute enables STA thread pools on a per-page basis to address performance with existing Visual Basic components.
5. In nearly all scenarios, rewriting your COM components in managed code provides performance benefits.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model

MTS Transactions

A transaction is an operation or set of operations that succeeds or fails as a logical unit. A good example of a transaction is the transfer of funds from one bank account to another. In this case, the funds must be debited from the first account and credited to the second account before the operation can be considered a success. If the funds are successfully debited but not credited, the debit from the first account must be undone to leave both accounts in a correct and consistent state.

Transactions are normally managed by declaring boundaries around a set of operations. Operations that execute in the context of the transaction boundary then succeed or fail as a unit. For ASP.NET, the transaction boundary is the execution of a single request to a page, which might contain nested components that participate in the same transaction. While the page is executing, if an operation on the page itself or a nested component in the same transaction fails, it can call **ContextUtil.SetAbort**. This is then picked up by the current transaction context, the entire transaction fails, and any operations that were already completed are undone. If nothing fails, the transaction is committed.

ASP.NET support for transactions consists of the ability to allow pages to participate in ongoing Microsoft .NET Framework transactions. Transaction support is exposed via an **@Transaction** directive that indicates the desired level of support:

```
<%@ Transaction="Required" %>
```

The following table defines the supported transaction attributes. The absence of a transaction directive is the same as an explicit directive to "Disabled". Unlike ASP, ASP.NET has no explicit directive for none (that is, Transaction="None").

Attribute	Description
Required	The page requires a transaction. It runs in the context of an existing transaction, if one exists. If not, it starts one.
RequiresNew	The page requires a transaction and a new transaction is started for each request.
Supported	The page runs in the context of an existing transaction, if one exists. If not, it runs without a transaction.
NotSupported	The page does not run within the scope of transactions. When a request is processed, its object context is created without a transaction, regardless of whether there is an active transaction.

A transaction can be explicitly committed or aborted using static methods of the **System.EnterpriseServices.ContextUtil** class. You can explicitly call the **SetComplete** or **SetAbort** method to commit or abort an ongoing transaction.

Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications
A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page

Note: A transaction will commit or abort at the end of page's lifetime depending on the whether SetComplete or SetAbort was called last, provided there is no other object join the same transaction.

```
' Try to do something crucial to transaction completing.
If (Not DoSomeWork())
    ContextUtil.SetAbort()
End If
```

VB

Section Summary

1. A transaction is an operation or set of operations that succeeds or fails as a logical unit.
2. ASP.NET transaction support consists of the ability to allow pages to participate in ongoing Microsoft .NET Framework transactions. Transaction support is exposed via an **@Transaction** directive that indicates the desired level of support.
3. A transaction can be explicitly committed or aborted using static methods of the **System.EnterpriseServices.ContextUtil** class. Developers can explicitly call the **SetComplete** or **SetAbort** method to commit or abort an ongoing transaction.

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview

A Personalized Portal

This sample illustrates a personalized portal home page application. The application allows users to customize a home page to show various modules of their choosing, such as a site directory or favorite links list. Each module is implemented as a user control, which is dynamically added to the home page if the user has chosen to include it. The custom personalization settings are maintained in a SQL database and are retrieved using a personalization HTTP module component (which works much as the session state and application state HTTP modules do). Every page in the application inherits from a common code-behind base **Page** class, which uses the personalization component to expose a special dictionary called UserState. This UserState dictionary provides the application's pages with access to the per-user customization settings (as key/value string pairs). In addition to storing the user's module selections, the UserState dictionary stores other customization parameters such as color schemes. Individual modules can use the UserState dictionary to store their own customization settings as well.

The portal application employs the **FormsAuthenticationModule** for user authentication. When a user first requests the home page, the settings for an anonymous user are displayed. If the user tries to access a portion of the portal that is restricted to authenticated users (such as the module customization page), the **FormsAuthenticationModule** redirects the user to a login page to enter credentials. A user who has not logged in before can use a registration form to create a new user account and password. On subsequent visits to the portal home page, a user can simply log in using these account credentials (which are then verified against a SQL database).

To get started exploring the portal application, follow the steps described above to create a user account. Once your account is created you can browse and customize the entire portal.

Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing
Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging
The SDK Debugger

Performance
Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration
Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications

VB Portal Application

[Run Sample] | [View Source]

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview

An E-Commerce Storefront

The following sample application is a mock-up of a typical e-commerce storefront. The application shows the most common elements of the following types of applications: a product browser, a session-based shopping cart, product details, and so forth. A SQL Server database is used to store the product data, and the **DataList** and **Repeater** controls render this data. The data access portion of the application is implemented as a managed component.

VB GrocerToGo.aspx

[Run Sample] | [View Source]

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview
Configuration File Format
Retrieving Configuration

Deployment
Deploying Applications
Using the Process Model
Handling Errors

Security
Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization
Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

A Class Browser Application

The following sample application implements a .NET Framework-based class browser, using the **System.Reflection** APIs to gather information about a class. To simplify the .aspx code, the application employs a managed component that encapsulates the reflection details. The .aspx page itself relies heavily on several **DataList** controls for rendering the namespaces, classes, and class details. The sample also shows the use of nested **DataList** controls for rendering the parameter lists. To view the sample, click the icon below.

VB ClassBrowser.aspx

[Run Sample] | [View Source]

The class browser also uses the ASP.NET configuration system to determine which modules to load and reflect upon. A configuration section is mapped to the **HashtableSectionHandler**, which maintains key/value pairs for the assembly name and file. You can add assemblies to this list by appending a line to the class browser application's configuration section, as follows:

```
<configuration>
  <configSections>
    <sectionGroup name="system.web">
      <section name="ClassBrowser"
type="System.Configuration.NameValueSectionHandler,
       System,Version=1.0.3300.0,Culture=neutral,PublicKeyToken=b77a5c561934e089"/>
    </sectionGroup>
  </configSections>

  <system.web>
    <ClassBrowser>
       <add key="ASP.NET Class Library" value="System.Web, Version=1.0.3300.0,
Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />
       <add key=".NET Framework class Library" value="mscorlib, Version=1.0.3300.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089" />
    </ClassBrowser>
  </system.web>

</configuration>
```

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
Introduction
What is ASP.NET?
Language Support

ASP.NET Web Forms
Introducing Web Forms
Working with Server Controls
Applying Styles to Controls
Server Control Form Validation
Web Forms User Controls
Data Binding Server Controls
Server-Side Data Access
Data Access and Customization
Working with Business Objects
Authoring Custom Controls
Web Forms Controls Reference
Web Forms Syntax Reference

ASP.NET Web Services
Introducing Web Services
Writing a Simple Web Service
Web Service Type Marshalling
Using Data in Web Services
Using Objects and Intrinsics
The WebService Behavior
HTML Pattern Matching

ASP.NET Web Applications
Application Overview
Using the Global.asax File
Managing Application State
HttpHandlers and Factories

Cache Services
Caching Overview
Page Output Caching
Page Fragment Caching
Page Data Caching

Configuration
Configuration Overview

IBuySpy.com

The IBuySpy ASP.NET sample application was built to show how you can use the new Microsoft .NET Framework and ASP.NET to build a full-featured e-commerce application. It offers all the functionality of typical shopping applications, including product searches, shopping cart management, user login and registration, and even the ability to view and edit your own product reviews.

Portions of IBuySpy were designed and developed by Vertigo Software, Inc.

IBuySpy.com

[View Sample]

Copyright 2001 Microsoft Corporation. All rights reserved.

Getting Started
- Introduction
- What is ASP.NET?
- Language Support

ASP.NET Web Forms
- Introducing Web Forms
- Working with Server Controls
- Applying Styles to Controls
- Server Control Form Validation
- Web Forms User Controls
- Data Binding Server Controls
- Server-Side Data Access
- Data Access and Customization
- Working with Business Objects
- Authoring Custom Controls
- Web Forms Controls Reference
- Web Forms Syntax Reference

ASP.NET Web Services
- Introducing Web Services
- Writing a Simple Web Service
- Web Service Type Marshalling
- Using Data in Web Services
- Using Objects and Intrinsics
- The WebService Behavior
- HTML Pattern Matching

ASP.NET Web Applications
- Application Overview
- Using the Global.asax File
- Managing Application State
- HttpHandlers and Factories

Cache Services
- Caching Overview
- Page Output Caching
- Page Fragment Caching
- Page Data Caching

Configuration
- Configuration Overview
- Configuration File Format
- Retrieving Configuration

Deployment
- Deploying Applications
- Using the Process Model
- Handling Errors

Security

Welcome to the ASP.NET QuickStart Tutorial

The ASP.NET QuickStart is a series of ASP.NET samples and supporting commentary designed to quickly acquaint developers with the syntax, architecture, and power of the ASP.NET Web programming framework. The QuickStart samples are designed to be short, easy-to-understand illustrations of ASP.NET features. By the time you have completed the QuickStart tutorial, you will be familiar with:

- **ASP.NET Syntax.** While some of the ASP.NET syntax elements will be familiar to veteran ASP developers, several are unique to the new framework. The QuickStart samples cover each element in detail.
- **ASP.NET Architecture and Features.** The QuickStart introduces the features of ASP.NET that enable developers to build interactive, world-class applications with much less time and effort than ever before.
- **Best Practices.** The QuickStart samples demonstrate the best ways to exercise the power of ASP.NET while avoiding potential pitfalls along the way.

What Level of Expertise Is Assumed in the QuickStart?
If you have never developed Web pages before, the QuickStart is not for you. You should be fluent in HTML and general Web development terminology. You do not need previous ASP experience, but you should be familiar with the concepts behind interactive Web pages, including forms, scripts, and data access.

Working with the QuickStart Samples
The QuickStart samples are best experienced in the order in which they are presented. Each sample builds on concepts discussed in the preceding sample. The sequence begins with a simple form submittal and builds up to integrated application scenarios.

Copyright 2001 Microsoft Corporation. All rights reserved.

Security Overview
Authentication & Authorization
Windows-based Authentication
Forms-based Authentication
Authorizing Users and Roles
User Account Impersonation
Security and WebServices

Localization

Internationalization Overview
Setting Culture and Encoding
Localizing ASP.NET Applications
Working with Resource Files

Tracing

Tracing Overview
Trace Logging to Page Output
Application-level Trace Logging

Debugging

The SDK Debugger

Performance

Performance Overview
Performance Tuning Tips
Measuring Performance

ASP to ASP.NET Migration

Migration Overview
Syntax and Semantics
Language Compatibility
COM Interoperability
Transactions

Sample Applications

A Personalized Portal
An E-Commerce Storefront
A Class Browser Application
IBuySpy.com

Get URL for this page