1
Architecture

UNLIKE ITS PREDECESSOR, ASP.NET 2.0 is not a fundamentally new way of building Web applications. Instead, ASP.NET 2.0 primarily adds new features on top of an existing architecture with the goal of simplifying many common tasks. This is not to say that this is a small release—quite the contrary. With more than double the number of classes and over 40 new controls, there is more than enough "newness" to keep even the most avid ASP.NET developer busy for quite some time exploring new features.

The core architecture, however, which consists of pages being parsed into class definitions and compiled into assemblies remains essentially unchanged, as does the HTTP pipeline used to process requests. In fact, it is possible to host most sites built for ASP.NET 1.1 directly in 2.0 without modification, as all existing features of the 1.1 runtime are completely supported in this release.

With that in mind, this introductory chapter focuses on the architectural changes that *are* made in 2.0 and how these changes affect the way you build Web applications in ASP.NET. These changes include a new codebehind mechanism, several new Page events, new specially named compilation directories, the new ASP.NET compiler utility that enables static site compilation, and Web Application Projects.

Fundamentals

We'll begin with a brief review of the fundamentals of ASP.NET, leading directly to a discussion of some of the new features in ASP.NET 2.0. This section presents the evolution of generating dynamic content, beginning with traditional classic ASP techniques and culminating in the new declarative data-binding model introduced in ASP.NET 2.0.

Dynamic Content

Writing a page to process a request in ASP.NET 2.0 is very much the same as it has been for the last few years with ASP.NET 1.1, and in fact for simple pages, it's very much the same as its earlier predecessor ASP. The same in-line evaluation syntax is still supported, as are server-side script blocks so that someone with a background only in building classic ASP pages should find this release of ASP.NET very approachable. Listing 1-1 shows a simple page that uses a server-side script block to define a helper method (GetDisplayItem) and then uses interspersed script to dynamically render elements in an unordered list. It also uses the server-side evaluation syntax (<%= %>) to intersperse dynamic content among static content—all common techniques dating back to building pages with ASP. The result of accessing the page through a browser is shown in Figure 1-1.

Listing 1-1: SimplePage.aspx—A simple .aspx file with dynamic content

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

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
const int _itemCount = 10;

string GetDisplayItem(int n)
{
    return "Item #" + n.ToString();
}
</script>

<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
    <title>Simple page</title>
</head>
```

```
<body>
    <h1>Test ASP.NET 2.0 Page</h1>
    <ul>
    <% for (int i=0; i<_itemCount; i++) { %>
    <li><%=GetDisplayItem(i)%></li>
    <% } %>
    </ul>
    <%
    Response.Write("<h2>Total number of items = " +
                   _itemCount.ToString() + "</h2>");
    %>
</body>
</html>
```

The difference between this page and a classic ASP page is that in ASP.NET the entire file's contents is parsed into a class definition and then compiled into an assembly. Server-side script blocks are added directly to the class definition. Interspersed script is merged into a Render method of the class, which when called writes all of the static and dynamic content to the response. The class itself inherits from System.Web.UI.Page, which in turn implements the IHttpHandler interface to become an endpoint in the request-processing architecture of ASP.NET. This was the primary shift in the transition from classic ASP to ASP.NET: Instead of using script on your pages to interact with classes, your page becomes a class, and the interaction with other classes is identical to what it would be from any other class. Listing 1-2 shows a slightly simplified version of what the page in Listing 1-1 turns into after ASP.NET parses it into a class definition.

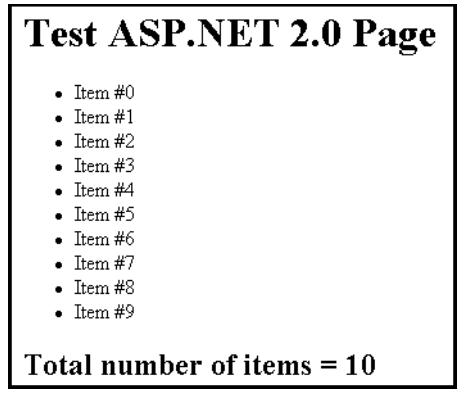

FIGURE 1-1: Rendering of SimplePage.aspx

Listing 1-2: Parsed class generated by ASP.NET (simplified)[1]

```
namespace ASP
{
  public class simplepage_aspx : Page
  {
    const int _itemCount = 10;

    string GetDisplayItem(int n)
    {
        return "Item #" + n.ToString();
    }

    protected override void Render(HtmlTextWriter writer)
    {
      writer.Write("<!DOCTYPE html PUBLIC \"-//W3C//DTD "  +
                   "XHTML 1.0 Transitional//EN\" \"http://www.w3"  +
                   ".org/TR/xhtml1/DTD/xhtml1-transitional.dtd\">");

      writer.Write("\r\n\r\n<html xmlns=\"http://www.w3.org/" +
                "1999/xhtml\" >\r\n<head>\r\n<title>Simple page<" +
                "/title>\r\n</head>\r\n<body>\r\n"                +
                "<h1>Test ASP.NET 2.0 Page</h1>\r\n    "          +
                "<ul>\r\n    ");

      for (int i = 0; i < _itemCount; i++)
      {
        writer.Write("\r\n<li> ");
        writer.Write(GetDisplayItem(i));
        writer.Write("</li>\r\n    ");
      }
      writer.Write("\r\n</ul>\r\n");

      Response.Write("<h2>Total number of items = " +
                     _itemCount.ToString() + "</h2>");

      writer.Write("\r\n</body>\r\n</html>\r\n");

      base.Render(writer);
    }
  }
}
```

1. The page shown in this listing is similar to the code that ASP.NET will generate when it parses your page. Steps have been taken to simplify some of the details for the purpose of presentation, but conceptually it is identical to the code generated by ASP.NET. If you would like to view the actual code that ASP.NET produces for any given page, do the following:
   1. Add Debug="true" to your @Page directive.
   2. Place <%= GetType().Assembly.Location %> somewhere on your page.

   This will print the location of the assembly generated for your page. If you go to that directory, you will also see source code files (*.cs or *.vb) that contain the class definitions.

Server-Side Controls

Of course, most ASP.NET pages do not use interspersed script to inject dynamic content at all. Instead, they rely on the server-side control architecture introduced with ASP.NET. Server-side controls look much like the rest of the HTML elements in an .aspx page, but are marked with the runat="server" attribute and are typically prefixed with the "asp" namespace. When ASP.NET parses the page, these controls are added to the generated class definition not as methods or code, but as member variables representing the specific element (or collection of elements) they are designed to render. As an example, Listing 1-3 shows the same page we've been using rewritten to use server-side controls instead of interspersed script to render the list and H2 elements.

LISTING 1-3: SimplePageWithControls.aspx—A simple .aspx file using server-side controls

```
<%@ Page Language="C#" Debug="true" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
const int _itemCount = 10;

string GetDisplayItem(int n)
{
  return "Item #" + n.ToString();
}

protected override void OnLoad(EventArgs e)
{
  // Clear out items populated by static declaration
  _displayList.Items.Clear();

  for (int i=0; i<_itemCount; i++)
    _displayList.Items.Add(new ListItem(GetDisplayItem(i)));

  _messageH2.InnerText = "Total number of items = " +
                         _itemCount.ToString();

  base.OnLoad(e);
}
</script>

<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
  <title>Simple page with controls</title>
```

continues

```
</head>
<body>
  <form runat="server" id="_form" >
  <h1>Test ASP.NET 2.0 Page with controls</h1>

  <asp:BulletedList runat="server" ID="_displayList">
      <asp:ListItem>Sample Item 1</asp:ListItem>
      <asp:ListItem>Sample Item 2 ...</asp:ListItem>
  </asp:BulletedList>

  <h2 runat="server" id="_messageH2">Total number of items = xx</h2>
  </form>
</body>
</html>
```

One of the primary advantages of using server-side controls is the complete separation of layout from programmatic logic. Note that instead of adding script elements to generate the dynamic portions of the page, we used the object model of the BulletedList control and the HtmlGenericControl (representing the H2 element) to populate their contents. This example also uses the fairly common technique of populating the static declarations of the server-side controls with sample content so that it is somewhat representative of what it will look like at runtime and will display properly in the designer for layout purposes.

The code generated by ASP.NET is actually a bit cleaner as well. No longer does the runtime have to create a special Render method to interweave interspersed script with static HTML strings. Instead, all static content on the page is represented using LiteralControls, which act like placeholders in the rendering process and return their associated strings when requested to render. Listing 1-4 shows the parsed class definition created by ASP.NET for the page shown in Listing 1-3 (the code has been simplified somewhat for clarity).

Listing 1-4: Parsed class generated by ASP.NET with server-side controls (simplified)

```
namespace ASP
{
  class SimplePageWithControls_aspx : Page
  {
    const int _itemCount = 10;

    // Control declarations
    protected BulletedList       _displayList;
    protected HtmlGenericControl _messageH2;
    protected HtmlForm           _form;
```

```
string GetDisplayItem(int n)
{
  return "Item #" + n.ToString();
}

protected override void OnLoad(EventArgs e)
{
  // Clear out items populated by control initialization
  _displayList.Items.Clear();

  for (int i = 0; i < _itemCount; i++)
    _displayList.Items.Add(new ListItem(GetDisplayItem(i)));

  _messageH2.InnerText = "Total number of items = " +
                         _itemCount.ToString();

  base.OnLoad(e);
}

protected override void  FrameworkInitialize()
{
  Controls.Add(new LiteralControl("<!DOCTYPE html " +
            "PUBLIC \"-//W3C//DTD " +
            "XHTML 1.0 Transitional//EN\" \"http://www.w3" +
            ".org/TR/xhtml1/DTD/xhtml1-transitional.dtd\"> "));
  Controls.Add(new LiteralControl("<html xmlns=\"http://" +
            "www.w3.org/1999/xhtml\" >" +
            "\r\n<head>\r\n    <title>Simple page " +
            "with controls</title>\r\n</head>\r\n<body>\r\n"));

  _form = new HtmlForm();
  _form.Controls.Add(new LiteralControl("<h1>Test ASP.NET 2.0 " +
                               "Page with controls</h1>\r\n    "));

  _displayList = new BulletedList();
  _displayList.ID = "_displayList";
  _displayList.Items.Add(new ListItem("Sample Item 1"));
  _displayList.Items.Add(new ListItem("Sample Item 2"));
  _displayList.Items.Add(new ListItem("Sample Item 3"));
  _form.Controls.Add(_displayList);

  _messageH2 = new HtmlGenericControl("h2");
  _messageH2.ID = "_messageH2";
  _messageH2.Controls.Add(new LiteralControl("Total number " +
                                             "of items = xx"));
  _form.Controls.Add(_messageH2);

  Controls.Add(_form);
```

continues

```
        Controls.Add(
              new LiteralControl("\r\n\r\n</body>\r\n</html>\r\n"));

        base.OnPreInit(e);
      }
    }
}
```

Data Binding

Most of the time you won't even have to go through the trouble of programmatically populating the elements of a list control like we did in Listing 1-3, as all list controls (as well as others) in ASP.NET support data binding. We will cover the details of data binding in Chapter 3, but suffice it to say that you can take any enumerable collection of items and bind it to a server-side control (like the BulletedList control) and it will autocreate the items for you at runtime. Listing 1-5 shows an example of binding an array of strings to the BulletedList control. The process of binding a collection of data to a control consists of setting the DataSource property to an enumerable collection (like the array in our case or, in general, any type that implements the IEnumerable interface). You can also set up data binding completely declaratively, as we will discuss in Chapter 3.

Listing 1-5: SimplePageWithDataBinding.aspx

```
<%@ Page Language="C#" Debug="true" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
string[] _displayItemData = {"Item #1", "Item #2", "Item #3", "Item #4",
    "Item #5", "Item #6", "Item #7", "Item #8", "Item #9", "Item #10"};

protected override void OnLoad(EventArgs e)
{
    _messageH2.InnerText = "Total number of items = " +
                           _displayItemData.Length.ToString();

    _displayItems.DataSource = _displayItemData;
    _displayItems.DataBind();

    base.OnLoad(e);
}
</script>
```

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
    <title>Simple page with controls</title>
</head>
<body>
    <form runat="server" id="_form">
        <h1>Test ASP.NET 2.0 Page with data binding</h1>

        <asp:BulletedList runat="server" ID="_displayItems">
            <asp:ListItem>Sample item 1</asp:ListItem>
            <asp:ListItem>Sample item 2 ...</asp:ListItem>
        </asp:BulletedList>

        <h2 runat="server" id="_messageH2">
            Total number of items = xx</h2>
    </form>
</body>
</html>
```

The trio of parsing pages into class declarations, server-side controls, and a generic data-binding architecture are really the three pillars of Web development with ASP.NET. With these three core features and the .NET runtime environment to build on, developers have created many well-designed, scalable Web applications that are in use today. ASP.NET 2.0 builds on these pillars to give developers a more productive set of tools to work from, as we will see over the next several chapters.

Codebehind

One of the big changes in this release of ASP.NET 2.0 is the way you specify a codebehind class for a page. We'll start with a review of how codebehind classes work in ASP.NET 1.x (which is still supported), and then introduce the new codebehind model introduced in ASP.NET 2.0.

Codebehind Basics

ASP.NET 1.0 introduced a new mechanism for separating programmatic logic from static page layout called **codebehind**. This technique involves creating an intermediate base class that sits between the Page base class and the machine-generated class from the .aspx file. The intermediate base class derives directly from Page, and the class generated from the .aspx file derives from the intermediate base class instead of directly from Page.

With this technique, you can add fields, methods, and event handlers in your codebehind class and have these features inherited by the class created from the .aspx file, eliminating the need to sprinkle code throughout the .aspx file.

Listings 1-6 and 1-7 show a sample .aspx file and its corresponding codebehind file using the 1.0 inheritance model. Note the use of the Src attribute in the Page directive that tells ASP.NET which file to compile to create the base class for this page. You can also leave off the Src attribute altogether and compile the codebehind file yourself, placing the resulting assembly in the /bin directory of your application (this is, in fact, the most common deployment model with Visual Studio .NET 2003).

Listing 1-6: SimplePageWithCodeBehindV1.aspx

```
<%@ Page Language="C#" AutoEventWireup="true"
          Src="SimplePageWithCodeBehind.aspx.cs"
          Inherits="EssentialAspDotNet.SimplePageWithCodeBehindV1" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Simple page with codebehind V1</title>
</head>
<body>
    <form id="form1" runat="server">

    <h1>Test ASP.NET 2.0 Page with codebehind V1</h1>
    <asp:BulletedList runat="server" ID="_displayList">
        <asp:ListItem>Sample Item 1</asp:ListItem>
        <asp:ListItem>Sample Item 2 ...</asp:ListItem>
    </asp:BulletedList>

    <h2 runat="server" id="_messageH2">Total number of items = xx</h2>

    </form>
</body>
</html>
```

Listing 1-7: SimplePageWithCodeBehindV1.aspx.cs

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

```
namespace EssentialAspDotNet
{

  public class SimplePageWithCodeBehindV1 : Page
  {
    protected BulletedList       _displayList;
    protected HtmlGenericControl _messageH2;

    string[] _displayItemData =
              {"Item #1", "Item #2", "Item #3", "Item #4",
               "Item #5", "Item #6", "Item #7", "Item #8",
               "Item #9", "Item #10"};

    protected override void OnLoad(EventArgs e)
    {
      _messageH2.InnerText = "Total number of items = " +
                             _displayItemData.Length.ToString();

      _displayList.DataSource = _displayItemData;
      _displayList.DataBind();

      base.OnLoad(e);
    }
  }
}
```

Codebehind 2.0

In this release of ASP.NET, the codebehind mechanism has changed slightly (although the existing 1.0 syntax is still completely supported). The change is so subtle that you may not even notice that it has changed unless you look really closely. Listings 1-8 and 1-9 show the new syntax using the same page as the previous example.

Listing 1-8: SimplePageWithCodeBehind.aspx using new 2.0 model

```
<%@ Page Language="C#" AutoEventWireup="true"
          CodeFile="SimplePageWithCodeBehind.aspx.cs"
          Inherits="EssentialAspDotNet.SimplePageWithCodeBehind" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Simple page with codebehind</title>
</head>
```

continues

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

    <h1>Test ASP.NET 2.0 Page with codebehind</h1>
    <asp:BulletedList runat="server" ID="_displayList">
        <asp:ListItem>Sample Item 1</asp:ListItem>
        <asp:ListItem>Sample Item 2 ...</asp:ListItem>
    </asp:BulletedList>

    <h2 runat="server" id="_messageH2">Total number of items = xx</h2>

    </form>
</body>
</html>
```

Listing 1-9: SimplePageWithCodeBehind.aspx.cs using new 2.0 model

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

namespace EssentialAspDotNet
{
  public partial class SimplePageWithCodeBehind : Page
  {
    string[] _displayItemData =
              {"Item #1", "Item #2", "Item #3", "Item #4",
               "Item #5", "Item #6", "Item #7", "Item #8",
               "Item #9", "Item #10"};

    protected override void OnLoad(EventArgs e)
    {
      _messageH2.InnerText = "Total number of items = " +
                             _displayItemData.Length.ToString();

      _displayList.DataSource = _displayItemData;
      _displayList.DataBind();

      base.OnLoad(e);
    }
  }
}
```

There are two significant differences between this model and the standard 1.x model: the introduction of the CodeFile attribute in the @Page directive and the declaration of the codebehind class as a partial class. As you start building the page, you will notice another difference: server-side

controls no longer need to be explicitly declared in your codebehind class, but you still have complete access to them programmatically, as shown in Listing 1-9.

The reason this works has to do with the partial keyword applied to your codebehind class. In addition to turning your .aspx file into a class definition with methods for rendering the page, as it has always done, ASP.NET now will also generate a sibling partial class for your codebehind class that contains protected control member variable declarations. Your class is then compiled together with this generated class definition, merged together, and then it becomes the base class for the class generated for the .aspx file. The end result is that you essentially write codebehind classes the way you always have, but you no longer have to declare (or let the designer declare for you) member variable declarations of server-side controls. This was always a somewhat fragile relationship in 1.x, since if you ever accidentally modified one of the control declarations so that it no longer matched the ID of the control declared on the form, things suddenly stopped working. Now the member variables are declared implicitly and will always be correct. Listings 1-10, 1-11, and 1-12 show the relationship between your codebehind class and the ASP.NET-generated classes.

Listing 1-10: Class for .aspx file generated by ASP.NET

```
namespace ASP
{
  public class samplepagewithcodebehind_aspx :
                    EssentialAspDotNet.SamplePageWithCodeBehind
  {
    ...
  }
}
```

Listing 1-11: Sibling partial class generated by ASP.NET

```
namespace EssentialAspDotNet
{
  public partial class SamplePageWithCodeBehind
  {
    protected BulletedList       _displayList;
    protected HtmlGenericControl _messageH2;
    protected HtmlForm           _form1;
    ...
  }
}
```

Listing 1-12: Codebehind partial class that you write

```
namespace EssentialAspDotNet
{
  public partial class SamplePageWithCodeBehind : Page
  {
    string[] _displayItemData =
              {"Item #1", "Item #2", "Item #3", "Item #4",
               "Item #5", "Item #6", "Item #7", "Item #8",
               "Item #9", "Item #10"};

    protected override void OnLoad(EventArgs e)
    {
      _messageH2.InnerText = "Total number of items = " +
                             _displayItemData.Length.ToString();

      _displayList.DataSource = _displayItemData;
      _displayList.DataBind();

      base.OnLoad(e);
    }
  }
}
```

Note that this partial class model is only used if you use the CodeFile keyword in your @Page directive. If you use the Inherits keyword without CodeFile (or with the Src attribute instead), ASP.NET reverts to the 1.1 codebehind style and simply places your class as the sole base class for the .aspx file. Also, if you have no codebehind at all, the class generation acts very much the same as it does in 1.1. Since ASP.NET 2.0 is backward compatible with 1.1, we now have a range of codebehind options at our disposal as Web developers. Visual Studio 2005 uses the new partial class codebehind model for any WebForms, and it will also happily convert Visual Studio .NET 2003 projects to using the new model as well if you use the Conversion wizard. It is best, if possible, to convert all files to the new codebehind model, since some of the new features of ASP.NET 2.0 depend on it.[2]

2. As an example, strongly typed access to the Profile property bag is added to the sibling partial class for codebehind classes in 2.0, but if you use the 1.1 codebehind model, that strongly typed accessor is added directly to the .aspx-generated class definition, and it will be unavailable to your codebehind class. This is also true for strongly typed master page and previous page access.

Page Lifecycle

One of the most important things to understand when building Web applications with ASP.NET is the sequence of events during the processing of a page. If you're not careful, you can make changes to a control that are then overwritten, which can result in unexpected behavior. As you build a page, you must take care that the code you write is called at the right time during the request processing to have the impact you expect. Fortunately, there are many events at your disposal in the Page base class. You can usually find the correct point in time to populate controls with default values, dynamically alter the control hierarchy, harvest POST data from the client, or whatever else you are trying to accomplish.

Common Events

The most common events to handle in ASP.NET are the Init and Load events. The **Load event** is issued prior to the rendering of a page, and it is the ideal location to initialize control state. It is also called after the state in a POST request has been processed and used to populate the control hierarchy, and so it can be used to inspect the contents of data sent by the client. The **Init event**, on the other hand, is called before any state restoration occurs and is commonly used to prepare the Page for processing a request. You can even do things like modify the control hierarchy in the Init event if there are dynamic changes you would like to make to a page. Most ASP.NET applications use a fairly standard event scheme in their interactive pages.

1. They initialize the state of controls for the first time in the Load event if it is the initial GET request to a page (the IsPostBack property of the page is set to false).
2. Next, they process user responses inside the server-side event of the control that generates the subsequent POST request.

Listings 1-13 and 1-14 show an example of this common practice with a page and its codebehind class.

Listing 1-13: CommonEvents.aspx

```
<%@ Page Language="C#" AutoEventWireup="true"
                 CodeFile="CommonEvents.aspx.cs"
                 Inherits="EssentialAspDotNet.CommonEvents" %>

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Common Events Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
    <h3>Enter name: </h3>
    <asp:TextBox id="_nameTextBox" runat="server"/>
    <h3>Personality: </h3>
    <asp:DropDownList id="_personalityDropDownList" runat="server" />
    <asp:Button id="_enterButton" Text="Enter" runat="server"
                OnClick="_enterButton_Click" /><br />
    <asp:Label runat="server" id="_messageLabel" />
    </div>
    </form>
</body>
</html>
```

Listing 1-14: CommonEvents.aspx.cs

```
namespace EssentialAspDotNet
{
  public partial class CommonEvents : System.Web.UI.Page
  {
    protected void Page_Load(object sender, EventArgs e)
    {
      if (!IsPostBack)
      {
        _personalityDropDownList.Items.Add(new ListItem("extraverted"));
        _personalityDropDownList.Items.Add(new ListItem("introverted"));
        _personalityDropDownList.Items.Add(new ListItem("in-between"));
      }
    }

    protected void _enterButton_Click(object sender, EventArgs e)
    {
      _messageLabel.Text = "Hi " + _nameTextBox.Text +
                           ", you selected " +
                           _personalityDropDownList.SelectedItem.Text;
    }
  }
}
```

New Events

This release of ASP.NET introduces even more events in the Page class, increasing the number of options you have for how to interact with the request processing of the Page. For the most part, the new events are "pre" and "complete" events that wrap one of the existing events. For example, there is now both a PreLoad event as well as a LoadComplete event in addition to the standard Load event. Figure 1-2 shows the updated sequence of events as well as the activities that occur during the processing of the page between the events.

Most notable among these new events are the PreInit and LoadComplete events. **PreInit** is important because, as you will see in Chapter 2, themes and master pages are applied between PreInit and Init. This means that PreInit is your only opportunity to make programmatic modifications to the selected theme or associated master page of a page. The **LoadComplete**

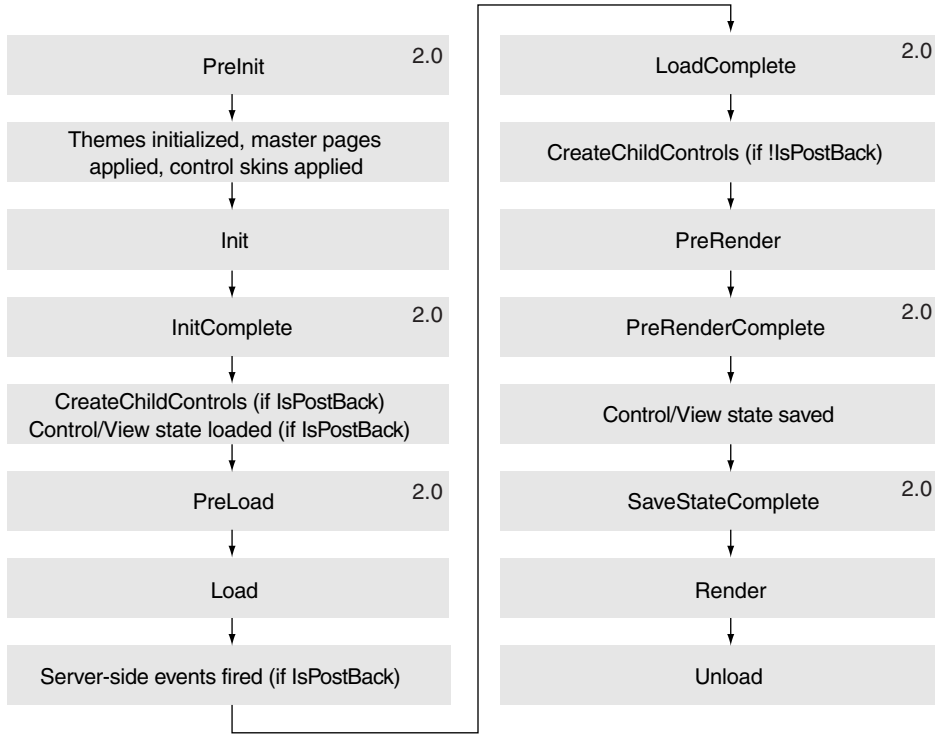

FIGURE 1-2: Events in the page lifecycle

event is also potentially quite useful as it is fired after the server-side events have fired but before the PreRender event takes place. Many applications written in ASP.NET today resort to using the PreRender event to make last-minute changes to control contents after server-side events fire. LoadComplete is now the proper place to make post-event modifications to a control, leaving PreRender as a hook for other activities.

Note that the PreInit, InitComplete, PreLoad, LoadComplete, PreRenderComplete, and SaveStateComplete are brand new events, and that they are only available in the Page class but not in individual controls as the other events are.

Implicit Event Subscription

One of the first things you will notice that is different in ASP.NET 2.0 is that Visual Studio 2005 creates event handlers for page events by enabling AutoEventWireUp and using specially named methods that are implicitly registered as event handlers—instead of explicitly registering delegates as the previous release did.[3] For example, to add a handler for the Load event of the Page class, Visual Studio 2005 adds a method to your codebehind class (or inline in a server-side script block) named Page_Load. Table 1-1 shows the complete list of method names that will be implicitly subscribed to events if they are added to your Page class.

Implicit delegate wireup occurs when a page has the AutoEventWireUp attribute set to true (which is the default), and one or more of the page's methods matches one of the names shown in Table 1-1. The method must also have the correct signature expected by the delegate defining the event (typically just EventHandler). At the beginning of the request cycle, the Page class invokes its SetIntrinsics method, which in addition to setting the intrinsics (meaning the Response, Request, Session, Application, and so on) calls the TemplateControl base class' HookUpAutomaticHandlers method. This method walks through the list of method names shown in Table 1-1 and uses reflection to identify methods with the same name and proper signature defined in your class. If it finds a match, it creates a new delegate of the appropriate type, initializes it with your method, and adds it to the list of delegates to fire when that event occurs.

3. This is only true for Web applications written in C#, as VB.NET still uses its "Handles" syntax to wire up control events.

TABLE 1-1: Method names and the events they bind to in ASP.NET 2.0

Method Name	Event
Page_PreInit	Page.PreInit
Page_Init	Control.Init
Page_InitComplete	Page.InitComplete
Page_PreLoad	Page.PreLoad
Page_Load	Control.Load
Page_LoadComplete	Page.LoadComplete
Page_PreRender	Control.PreRender
Page_DataBind	Control.DataBinding
Page_PreRenderComplete	Page.PreRenderComplete
Page_SaveStateComplete	Page.SaveStateComplete
Page_Unload	Control.Unload
Page_Error	TemplateControl.Error
Page_AbortTransaction	TemplateControl.AbortTransaction
OnTransactionAbort	TemplateControl.AbortTransaction
Page_CommitTransaction	TemplateControl.CommitTransaction
OnTransactionCommit	TemplateControl.CommitTransaction

Each of these events is fired by a virtual method defined in the Page base class (or a virtual method in the Control base class and inherited by Page). This means that it is technically possible to register for any of these events in three different ways. For example, to handle the Load event, you can do any of the following:

- Wire up a delegate explicitly to the event yourself (typically in your Page's Init handler).

- Write a method named Page_Load with the event signature.
- Override the virtual OnLoad method.

Each of these techniques essentially accomplishes the same task, and in the end it doesn't matter which way you do it. The virtual method override is going to be marginally faster than the explicitly or implicitly wired delegate approaches, but in general the difference in overhead will typically be dwarfed by other activities in your page (like data access). If you are using Visual Studio 2005, the technique it uses for you by default is the implicit delegate wireup based on the method's name.

Compilation

The number of ways you can compile your code increases many times over with the release of ASP.NET 2.0. In addition to the precompiled bin directory and the delay-compiled Src attribute deployment options in ASP.NET 1.x, you can now deploy raw source files to specially named directories (like /App_Code). There is a new utility, aspnet_compiler.exe, which will precompile an entire virtual directory to create a zero-source deployment (including .aspx file content). Web Deployment Projects also has a supplemental addition to Visual Studio 2005, which provides even more alternatives for compilation and deployment. We will cover each of these new compilation features in this section.

Compilation Directories

In ASP.NET 1.0, the only way to deploy supplemental classes with your Web application locally is to compile them into an assembly and place them in the /bin directory under the virtual application root. Any assembly placed in the /bin directory of an application is shadow copied to a private directory during site compilation, and every compile that ASP.NET issues for that site includes a reference to the shadow-copied assembly. This ensures that you can replace the assembly in the /bin directory with an updated one without having to shut down the Web server. When the timestamp on a particular assembly in the /bin directory is updated (typically by replacing it with a new version), the contents of the site is recompiled with references to the new assembly (which is again shadow copied prior to reference).

This technique of deploying precompiled assemblies is still supported in ASP.NET 2.0, and depending on how you structure your site and your build process, this may still be your best option going forward. There is another option in this release, however, which is to place any source code files that you would like to have compiled and referenced by your site's other elements in the new top-level App_Code directory. In fact, there are seven new top-level folders that have special meaning in ASP.NET 2.0, as shown in Table 1-2.

TABLE 1-2: Special compilation folders in ASP.NET 2.0

Directory Name	Contents	Compilation
App_Browsers	.browser files (XML format files used to describe browser capabilities)	Each .browser file is compiled into a method in the local ApplicationBrowserCapabilitiesFactory class that is used to populate an instance of HttpBrowserCapabilities when needed.
App_Code	Source code files (.cs, .vb, etc.), .wsdl files, .xsd files (extensible)	Source code files are compiled into an assembly for use in your application. .wsdl files are parsed into Web service proxies and then compiled. .xsd files are parsed into strongly typed DataSet classes and then compiled.
App_Data	Database files, xml data sources, other data source files	No compilation takes place.
App_GlobalResources	Resource files (.resx and .resources)	Compiled into a resource-only assembly with global scope.
App_LocalResources	Resource files (.resx and .resources) that are associated with a particular page or user control	Compiled into a resource-only assembly for access by the associated page or user control.

continues

TABLE 1-2: Special compilation folders in ASP.NET 2.0 (*continued*)

Directory Name	Contents	Compilation
App_Themes	.skin, .css, images, and other resources	Compiled into a separate assembly containing resources for a particular theme.
App_WebReferences	.wsdl, .xsd, .disco, .discomap	Generates a Web service proxy for each endpoint described.
Bin	.dll assembly files	No compilation takes place. Assemblies placed in this directory are shadow copied and referenced during all other compilations associated with the site.

Any source files placed in the App_Code folder will be compiled along with all of your pages and their codebehind files when ASP.NET processes requests for your site (or during site precompilation). This means that you now have a complete range of deployment options, ranging from placing all of your source code on the server (including utility classes, business layers, data access layers, etc.) to precompiling any subset and placing the resulting assemblies in the /bin directory (or even deploying machine-wide in the global assembly cache (GAC)). Note that the decision of where to place your code files is purely a matter of convenience and organization. There is no difference in performance between a precompiled assembly and code that is placed in the App_Code folder and compiled at request time once the compilation has taken place.[4]

As an example of using the App_Code folder, consider the class in Listing 1-15 that we intend to use as a data source for various pages in our site.

4. There can be additional overhead the first time a request is made to a site using the App_Code folder, but even this can be eliminated by precompiling the site (which we will discuss shortly).

Listing 1-15: Custom data source class for deployment in the App_Code directory

```
// File: MyDataSource.cs
namespace EssentialAspDotNet.Architecture
{
  public static class MyDataSource
  {
    static string[] _items =
      {"Item #1", "Item #2", "Item #3", "Item #4",
       "Item #5", "Item #6", "Item #7", "Item #8",
       "Item #9", "Item #10"};

    public static string[] GetItems()
    {
      return _items;
    }
  }
}
```

To deploy this file, you would manually create a directory at the root of your Web application named App_Code and place MyDataSource.cs in it. All pages (and other generated types) in your site would then have access to the compiled class implicitly. The class will be compiled as part of the request sequence in much the same way .aspx files and their codebehind files are compiled. For example, we could now rewrite our earlier data-binding example using the ObjectDataSource control to declaratively associate the GetItems method as the data source for our BulletedList as shown in Listing 1-16. Chapter 3 covers the details of the ObjectDataSource; for now, note that it can be initialized with a type name and a method name, and when associated with the DataSourceID of a data-bound control, it will bind the results of invoking the method on the object to the control prior to rendering.

Listing 1-16: Simple page with declarative data binding using custom data source class

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

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Simple page with declarative data binding</title>
</head>
<body>
    <form id="form1" runat="server">
```

continues

```
    <h1>Test ASP.NET 2.0 Page with declarative data binding</h1>
    <asp:BulletedList runat="server" ID="_displayList"
                      DataSourceID="_itemsDataSource" >
        <asp:ListItem>Sample Item 1</asp:ListItem>
        <asp:ListItem>Sample Item 2 ...</asp:ListItem>
    </asp:BulletedList>

    <h2 runat="server" id="_messageH2">Total number of items = xx</h2>

    <asp:ObjectDataSource runat="server" ID="_itemsDataSource"
             TypeName="EssentialAspDotNet.Architecture.MyDataSource"
             SelectMethod="GetItems" />
    </form>
</body>
</html>
```

Site Compilation

Perhaps the most significant addition to the process of compilation in this release is the introduction of the ASP.NET compiler. The **ASP.NET compiler** (aspnet_compiler.exe) gives you the ability to completely precompile an entire site, making it possible to deploy nothing but binary assemblies (even .aspx and .ascx files are precompiled). This is compelling because it eliminates any on-demand compilation when requests are made, eliminating the first post-deployment hit seen in some sites using ASP.NET 1.0. It also makes it more difficult for modifications to be made to the deployed site (since you can't just open .aspx files and change things), which can be appealing when deploying applications that you want to be changed only through a standard deployment process.

Figure 1-3 shows an invocation of the aspnet_compiler.exe utility using the binary deployment option and the resulting output to a deployment directory. Note that the .aspx files present in the deployment directory are just marker files with no content. They are there to ensure that a file with the endpoint name is present if the "Check that file exists" option for the .aspx extension in an IIS application is set. The PrecompiledApp.config file is used to keep track of how the application was deployed and whether ASP.NET needs to compile any files at request time. Note that this utility is also accessible graphically through the Build | Publish Web Site menu item of Visual Studio shown in Figure 1-4.

In addition to the binary-only deployment model, the aspnet_compiler also supports an "updatable" deployment model, where all source code in

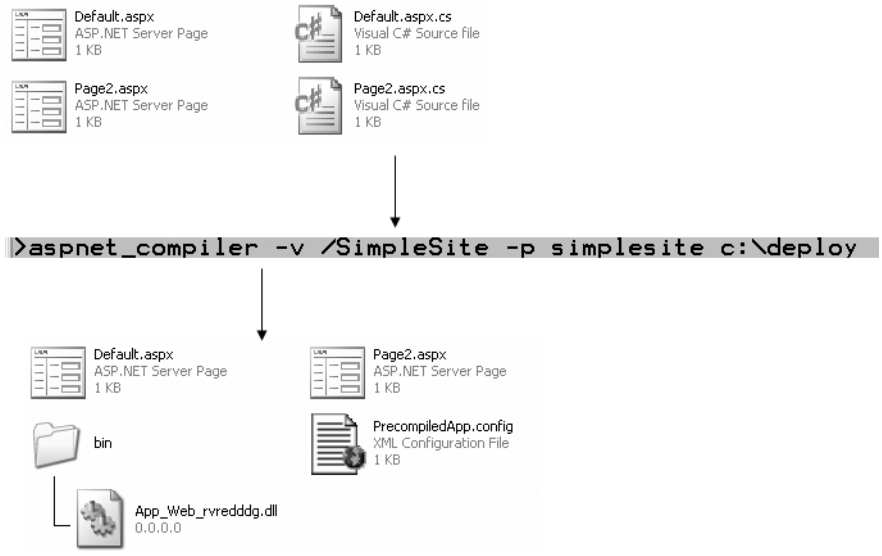

FIGURE 1-3: Binary deployment with aspnet_compiler.exe

a site is precompiled into binary assemblies, but all .aspx, .ascx, .master, .ashx, and .asax files are left intact so that changes can be made on the server. This model is possible because of the inheritance in the codebehind model so that the sibling partial classes containing control declarations can

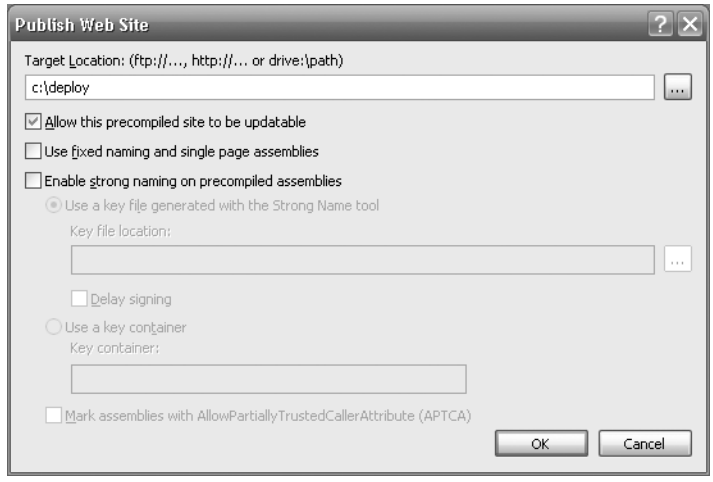

FIGURE 1-4: Build | Publish Web Site tool in Visual Studio 2005

be generated and compiled independently of the actual .aspx file class definitions. To generate the "updatable" site you would use -u with the command line utility, and the resulting .aspx files would contain their original content (and not be empty marker files).

With the aspnet_compiler utility in hand, you can work on your applications without worrying about how your application will be deployed, for the most part, since any site can now be deployed in any of three ways: all source, all binary, or updatable (source code in binary and .aspx files in source), without any modification to page attributes or code files used in development. This was not possible in previous releases of ASP.NET, since you had to decide at development time whether to use the Src attribute to reference codebehind files or to precompile them and deploy the assemblies to the /bin directory. Complete binary deployment was not even an option.

Assembly Generation

Now that compilation into assemblies can happen in one of three places (explicitly by the developer, using aspnet_compiler.exe, or during request processing), understanding the mapping of files into assemblies becomes even more important. In fact, depending on how you write your pages, you can actually end up with an application that works fine when deployed as all source or all binary, but which fails to compile when deployed using the updatable switch.

The general model ASP.NET uses is to create separate assemblies for the contents of the App_Code directory as well as the global.asax file (if present), and then to compile all of the .aspx pages in each directory into a separate assembly. User controls and master pages are also compiled independently from .aspx pages. It is also possible to configure the App_Code directory to create multiple assemblies if, for example, you wanted to include both VB.NET and C# source code in a project, as you will see shortly. Table 1-3 describes which of your Web site components compile into separate assemblies based on the deployment mode you are using (note that we are ignoring the resource, theme, and browser directories since they don't contain code, although they are compiled into separate assemblies as well).

There is one other twist in the assembly generation picture: You can use the -fixednames option in the aspnet_compiler to request that each .aspx

TABLE 1-3: Assembly generation

	Deployment Mode		
	All Source	**All Binary**	**Updatable (Mixed)**
What Compiles into a Unique Assembly	App_Code directory global.asax .ascx and associated codebehind file (separate assembly for each user control) .master and associated codebehind file (separate assembly for each master page) All .aspx files and their codebehind files in a given directory (separate assembly per directory)	App_Code directory global.asax .ascx and .master files and their associated codebehind files All .aspx files and their codebehind files in a given directory (separate assembly per directory)	App_Code directory (D) global.asax (R) .ascx and .master files (R) Codebehind files for .ascx and .master files (D) All .aspx files in a given directory (separate assembly per directory) (R) All codebehind files associated with .aspx files in a given directory (separate assembly per directory) (D)
When It's Compiled	Request time	Deployment time	(R) = Compiled at request time (D) = Compiled at deployment time

file be compiled into a separate assembly whose name remains the same across different invocations of the compiler. This can be useful if you want to be able to update individual pages without modifying other assemblies on the deployment site. It can also generate a large number of assemblies for any site of significant size, so be sure to test this option before depending on it.

If this is sounding complicated, the good news is that most of the time you shouldn't have to think about which files map to separate assemblies. Your .aspx files are always compiled last, and always include references to all other assemblies generated, so typically things will just work no matter which deployment model you choose.

Customizing Assembly Generation

You have additional control in how assemblies are generated in the App_Code directory. You can use the codeSubDirectories element to further specify that subdirectories should be compiled into individual assemblies. This can be useful if you find the need to house C# and VB.NET source code in the same project, as usually they could not be placed in the same App_Code directory. Figure 1-5 shows a sample layout that maps four distinct directories to different assemblies during compilation.

Web Application Projects

In May of 2006, Microsoft released an addition to Visual Studio 2005 called **Web Application Projects,**[5] which gives you a completely different model for building Web applications with ASP.NET 2.0, one much more similar to the model developers are familiar with using Visual Studio .NET 2003. As with Web projects in Visual Studio .NET 2003, all code files in the project are built into a single assembly, which is deployed to the local /bin

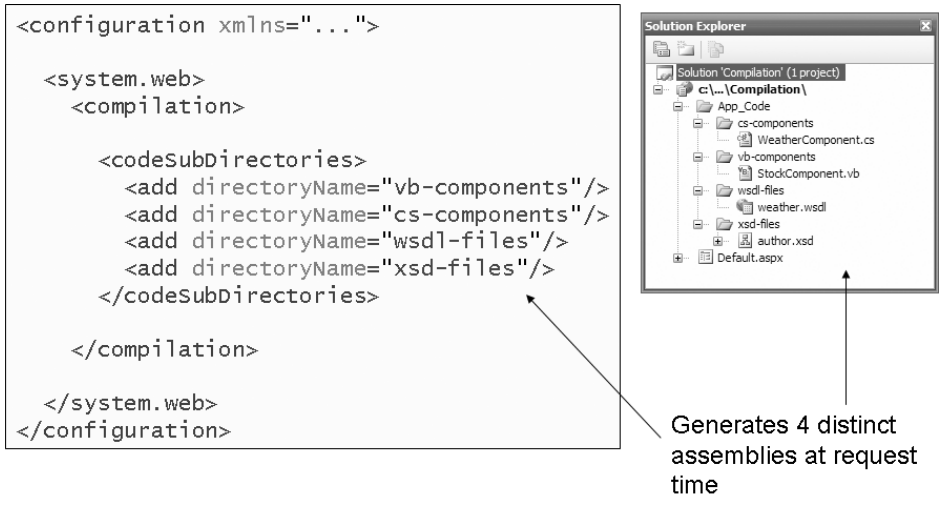

FIGURE 1-5: Creating multiple assemblies from the App_Code directory

5. You can download the Web Application Projects installation from http://msdn.microsoft.com/asp.net/reference/infrastructure/wap/default.aspx. Note that this project model does not install with the Express versions of the product.

directory. Because Web Application Projects have a project file and are compiled like any class library project, they have complete support for all class library project settings. Figure 1-6 shows a sample Solution Explorer window from a Web Application Project.

Unlike Web projects in Visual Studio .NET 2003, Web Application Projects do not require a virtual directory to be set up properly before the project can be opened. By default, they use the same ASP.NET Development Server listening on an open port for hosting pages, just as the Web site model does. They of course support the ability to work directly against a virtual directory hosted in IIS just as you can with Web sites.

The new partial class codebehind model is used by default with this model just like the Web site model does; however, it also supports the 1.1 style of codebehind with no issues, which means that migrating a site from Visual Studio .NET 2003 to Visual Studio 2005 is trivial using the Web Application Projects model. This makes it very appealing for larger sites that need to migrate to 2.0 without reworking all of their project settings and codebehind files. When the partial codebehind class model is used, there is a new source code file that is added called "*xxx*.aspx.designer.cs," where *xxx* is the name of the Web form. This file contains the control declarations that are added implicitly by ASP.NET in the Web site model. Many developers find this approach more compelling because all of the code for your codebehind classes is in one place and easy to view. It is no longer necessary to use the App_Code directory, because all source code files are compiled as part of the project (if they are included).

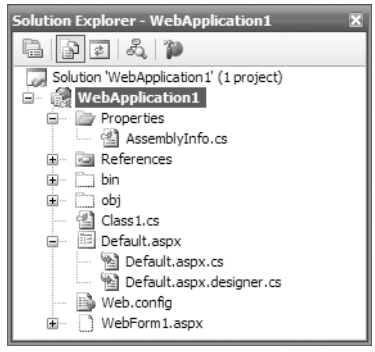

FIGURE 1-6: Web Application Projects' Solution Explorer window

The following are some other advantages to using the Web Application Projects model:

- Faster compile times. Since only the source code for the project is compiled when a build is performed, build times are faster—sometimes much faster. The drawback is that syntax errors on .as*x files will not be detected until runtime.
- All code files are compiled into a single assembly deployed in the local /bin directory, so it is easy to understand the dependencies in your project.
- The project file lets you easily exclude files (and directories) from the build process, whereas in the Web site model you must rename a file with the .exclude extension to exclude it.
- It uses the standard MSBuild compilation process, which can be extended using the MSBuild extensibility rules.
- All debugging features available for projects are available, including features like Edit and Continue and pre- and post-processing steps.

Which model you end up using for building ASP.NET 2.0 applications depends on what environment you are working in and what you are used to. Web Application Projects were introduced to give enterprise developers used to working with project files and performing builds a way to incorporate their Web applications into their standard working environments. Future releases of Visual Studio will include Web Application Projects as one of the built-in options for creating Web applications with ASP.NET.

SUMMARY

This release of ASP.NET builds upon the substrate for building Web applications introduced in version 1.0. All of the architectural features of the ASP.NET 1.x runtime are still present in 2.0, but elements were added to make development of Web applications more intuitive and efficient. One of the most significant additions is the partial class codebehind model, in which instead of manually declaring control variables in your codebehind file, ASP.NET generates a sibling class that is merged with your class defi-

nition to provide control variable declarations. The events in the lifetime of a Page were augmented as well, to include many pre- and post-events to give more granular access to points in time during a Page's lifecycle. The other major change architecturally is the compilation model. It is now possible to deploy Web sites as nothing but binary assemblies, as well as all source, and many gradients in between. Developers now have many more options for both development and deployment of Web applications with ASP.NET.