1

The SharePoint 2007 Architecture

SharePoint 2007 is an extension of ASP.NET and IIS. This chapter walks through the main architectural components of IIS and ASP.NET and shows you how these components are extended to add support for SharePoint functionalities.

Because IIS is one of the many options for hosting ASP.NET, the discussion begins with the coverage of the ASP.NET hosting environment where HTTP worker requests and runtime classes are discussed. Next, the chapter covers IIS concepts such as web sites and application pools followed by discussions of the related SharePoint object model classes. The ASP.NET HTTP Runtime Pipeline and the SharePoint extensions to this pipeline are discussed in depth. You'll also learn about the ASP.NET dynamic compilation model and the role it plays in SharePoint.

ASP.NET Hosting Environment

One of the great architectural characteristics of the ASP.NET Framework is its isolation from its hosting environment, which allows you to run your ASP.NET applications in different hosting scenarios such as IIS 5.0, IIS 5.1, IIS 6.0, IIS 7.0, or even a custom managed application. This section discusses these architectural aspects of the ASP.NET Framework as well as the most common hosting scenario, IIS.

Hosting ASP.NET

As mentioned, ASP.NET can be hosted in different environments, such as IIS 5.0, IIS 6.0, IIS 7.0, or even a custom managed application such as a console application. Hosting ASP.NET in a given environment involves two major components:

- **Worker request class.** This is a class that directly or indirectly inherits from the HttpWorkerRequest abstract base class. As you'll see later, an ASP.NET component named HttpRuntime uses the worker request class to communicate with the underlying environment. All worker request classes implement the HttpWorkerRequest API,

which isolates HttpRuntime from the environment-specific aspects of the communications between HttpRuntime and the underlying environment. This allows ASP.NET to be hosted in any environment as long as the environment comes with a worker request class that implements the HttpWorkerRequest API.

- **Runtime class.** By convention, the name of this class ends with the word *Runtime*. The runtime class must perform two tasks for every client request. These tasks are to:
  - Instantiate and initialize an instance of the appropriate worker request class.
  - Call the appropriate method of HttpRuntime, passing in the worker request instance to process the request.

The worker request and runtime classes are discussed in the following sections.

HttpWorkerRequest

As mentioned previously, this API isolates HttpRuntime from its environment, allowing ASP.NET to be hosted in different types of environments. For example, as you can see from Listing 1-1, the HttpWorkerRequest class exposes a method named CloseConnection that HttpRuntime calls to close the connection with the client without knowing the environment-specific details that close the connection under the hood. Or HttpRuntime calls the FlushResponse method (see Listing 1-1) to flush the response without knowing the environment-specific details that flush the response under the hood.

Listing 1-1: The HttpWorkerRequest API

```
public abstract class HttpWorkerRequest
{
  public virtual void CloseConnection();
  public abstract void EndOfRequest();
  public abstract void FlushResponse(bool finalFlush);
  public virtual byte[] GetPreloadedEntityBody();
  public virtual int GetPreloadedEntityBody(byte[] buffer, int offset);
  public abstract string GetQueryString();
  public virtual int ReadEntityBody(byte[] buffer, int size);
  public abstract void SendResponseFromFile(string filename, long offset,
                                            long length);
  public abstract void SendResponseFromMemory(byte[] data, int length);
}
```

ASP.NET comes with several standard implementations of the HttpWorkerRequest API, each one specifically designed to handle the communications with a specific hosting environment such as IIS 5.1, IIS 6.0, IIS 7.0, and so on. ASP.NET also comes with a standard implementation of the HttpWorkerRequest API called SimpleWorkerRequest, which you can use to host ASP.NET in custom managed environments, such as a console application.

Runtime Class

The ASP.NET 2.0 Framework comes with two important runtime classes named ISAPIRuntime and PipelineRuntime. Each class is described next:

- **ISAPIRuntime.** This is the runtime class for ISAPI-based IIS environments, including IIS 5, IIS 6, and IIS 7 running in ISAPI mode.
- **PipelineRuntime.** This is the runtime class for IIS 7 running in integrated mode.

These runtime classes are very different from one another because they represent different runtime environments. However, they both feature a method (though each method is different) that processes the request:

- ISAPIRuntime uses the ProcessRequest method.
- PipelineRuntime uses the GetExecuteDelegate method.

The request processing method for each runtime class takes two important steps. First, it instantiates and initializes an instance of the appropriate worker request class. Second, it passes the instance to the appropriate method of HttpRuntime to process the request.

Internet Information Services (IIS)

Microsoft Internet Information Services (IIS), the Windows web server, is an integral part of the Windows 2000 Server, Windows XP Professional, Windows Server 2003, Windows Vista, and Windows Server 2008 operating systems. IIS is an instance of a Win32 executable named inetinfo.exe, which is located in the following folder on your machine:

```
%SystemRoot%\System32\inetsrv
```

The version of IIS running on your machine depends on your OS version. Each IIS version presents a somewhat different ASP.NET request processing model. An ASP.NET request processing model is a set of steps taken to process incoming requests. These steps vary from one IIS version to another. We'll only cover IIS 6.0 here.

One of the great architectural advantages of IIS 6.0 is its extensibility model, which allows you to write your own ISAPI extension and filter modules to extend the functionality of the web server. An ISAPI extension module is a Win32 DLL that can be loaded into the IIS process (inetinfo.exe) itself or another process.

IIS communicates with ISAPI extension modules through a standard API that contains an important function named HttpExtensionProc as shown in the following code snippet:

```
DWORD WINAPI HttpExtensionProc (LPEXTENSION_CONTROL_BLOCK lpECB);
```

This function takes a parameter named lpECB that references the Extension_Control_Block data structure associated with the current request. Every ISAPI extension is specifically designed to process requests for resources with specific file extensions. For example, the asp.dll ISAPI extension handles requests for resources with an .asp file extension. Every version of ASP.NET comes with a specific version of the

ISAPI extension module named aspnet_isapi.dll. This module, like any other ISAPI module, is a Win32 dynamic link library (DLL). As such it's an unmanaged component. The ASP.NET ISAPI extension module (aspnet_isapi.dll) is located in the following folder on your machine:

```
%SystemRoot%\Microsoft.NET\Framework\versionNumber\aspnet_isapi.dll
```

Upon installation, ASP.NET automatically registers the ASP.NET ISAPI extension module with the IIS metabase for handling requests for resources with the ASP.NET-specific file extensions such as .aspx, .asmx, .asax, .ashx, and so on. The IIS metabase is where the IIS configuration settings are stored. You can use the Application Configuration dialog to access the IIS metabase.

IIS passes the Extension_Control_Block data structure to the HttpExtensionProc method of the ASP.NET ISAPI extension module, and is then finally passed into the ProcessRequest method of the ISAPIRuntime class. This is discussed in greater detail later in the chapter.

Application Pools

IIS 6.0 allows you to group your web applications into what are known as *application pools* (see Figure 1-1). Web applications residing in the same application pool share the same worker process. The worker process is an instance of the w3wp.exe executable.

Because different application pools run in different worker processes, application pools are separated by process boundaries. This has the following important benefits:

- Because web applications do not run inside the IIS process, application misbehaviors will not affect the IIS process itself. This dramatically improves the reliability and stability of the web server.
- Because application pools are isolated by process boundaries, application failure in one pool has no effect on applications running in other pools.

  Because upgrades and troubleshooting are done on a per application pool basis, upgrading and troubleshooting one pool has no effect on other pools. This provides tremendous benefits to system administrators because they don't have to restart the whole web server or all applications running on the web server to perform a simple upgrade or troubleshooting that affects only a few applications.

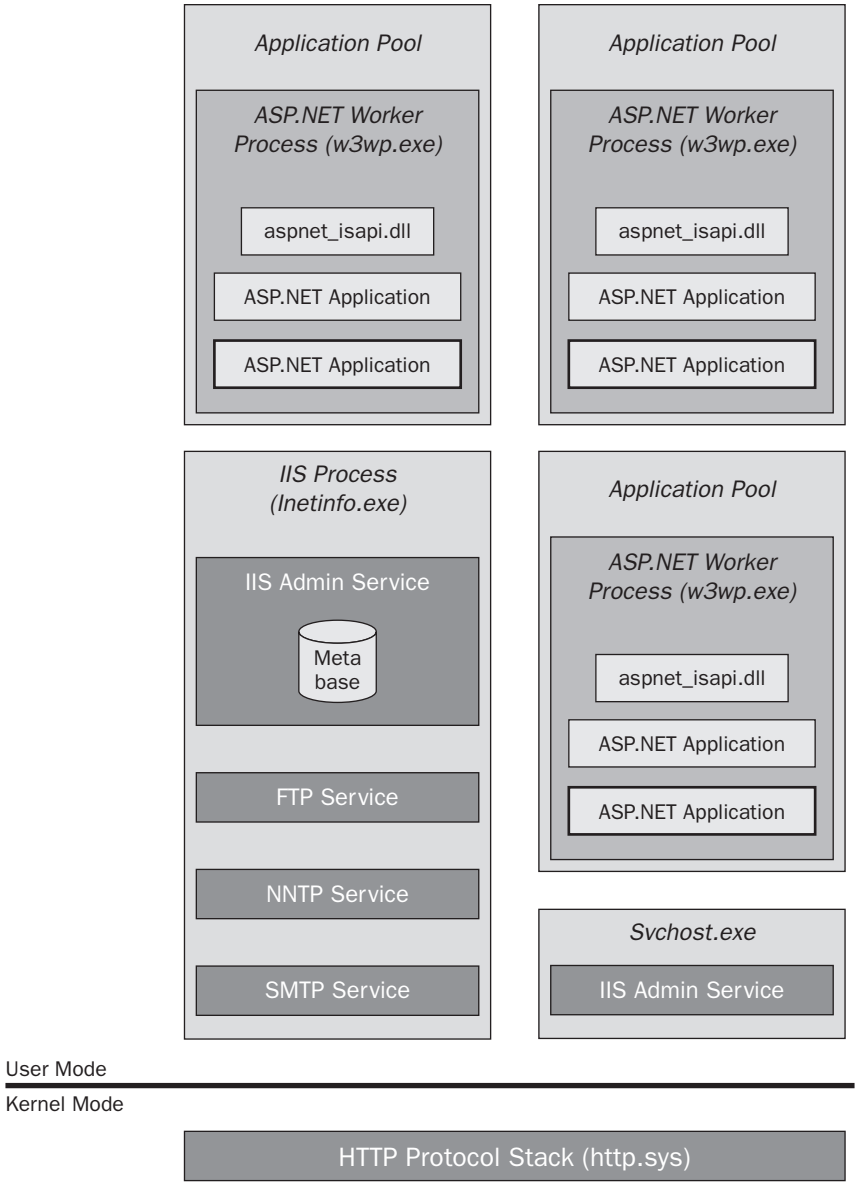

Figure 1-1: The default_aspx class

The w3wp.exe executable is an IIS 6.0-specific executable located in the following IIS-specific directory on your machine:

```
%SystemRoot%\System32\inetsvc\w3wp.exe
```

As such, you have to configure the w3wp worker process from the Internet Information Services (IIS) 6.0 Manager.

IIS 6.0 introduces a new kernel-mode component named the HTTP Protocol Stack (http.sys) that eliminates the need for interprocess communications between the worker process and the IIS process. Earlier IIS versions use the user-mode Windows Socket API (WinSock) to receive HTTP requests from the clients and to send HTTP responses back to the clients. IIS 6.0 replaces this user-mode component with the kernel-mode http.sys driver. Here is how this driver manages to avoid the interprocess communications.

When you add a new virtual directory for a new web application belonging to a particular application pool, IIS 6.0 registers this virtual directory with the kernel-mode http.sys driver. The main responsibility of http.sys is to listen for an incoming HTTP request and pass it onto the worker process responsible for processing requests for the associated application pool. http.sys maintains a kernel-mode queue for each application pool, allowing it to queue the request in the kernel-mode request queue of the associated application pool. The worker process then picks up the request directly from the kernel-mode request queue. As you can see, this avoids the interprocess communications between the web server and worker process. When the worker process is done with processing the request, it returns the response directly to http.sys, avoiding the interprocess communication overhead.

http.sys not only eliminates the interprocess communication overhead but also improves the availability of your applications. Here's why: Imagine that the worker process responsible for processing requests for a particular application pool starts to misbehave. http.sys will keep receiving HTTP requests and queueing them in the associated kernel-level queue while the WWW Service is starting a new worker process to process the requests. The users may feel a little delay, but their requests will not be denied.

Another added performance benefit of http.sys is that it caches the response in a kernel-mode cache. Therefore the next requests are directly serviced from this kernel-mode cache without switching to the user mode.

SharePoint Extensions

SharePoint extends IIS and ASP.NET to add support for SharePoint functionality. This section provides more detailed information about these extensions and the roles they play in the overall SharePoint architecture. The SharePoint object model also contains types whose instances represent typical IIS entities such as application pools, web sites, and so on. These types allow you to program against these entities within your managed code.

SPApplicationPool

SharePoint represents each IIS application pool with an instance of a type named SPApplicationPool, which allows you to program against IIS application pools from your managed code. Here are some of the public properties of the SPApplicationPool class that you can use in your managed code:

- **CurrentIdentityType.** This property gets or sets an IdentityType enumeration value that specifies the type of identity (such as the type of Windows account) under which this application pool is running. The possible values are LocalService, LocalSystem, NetworkService, and SpecificUser.
- **DisplayName.** This read-only property gets the display name of this application pool.

- **Farm.** This read-only property gets a reference to the SPFarm object that represents the SharePoint farm where this application pool resides.
- **Id.** This property gets or sets a GUID that uniquely identifies this application pool.
- **Name.** This property gets or sets a string that contains the name of this application pool.
- **Password.** This property gets or sets a string that specifies the password of the Windows account under which this application pool is running.
- **Username.** This property gets or sets a string that specifies the username of the Windows account under which this application pool is running.
- **Status.** This property gets or sets an SPObjectStatus enumeration value that specifies the current status of this application pool. The possible values are Disabled, Offline, Online, Provisioning, Unprovisioning, and Upgrading.

Here are some of the public methods of the SPApplicationPool class:

- **Delete.** This method deletes the application pool.
- **Provision.** This method creates the application pool.
- **Unprovision.** This method removes the application pool.
- **Update.** This method updates and commits the changes made to the application pool.
- **UpdateCredentials.** This method updates and commits the credentials under which the application pool is running.

Listing 1-2 presents an example that shows you how to use the SPApplicationPool class to program against the IIS application pools.

Listing 1-2: A page that displays the application pools running in the local farm

```
<%@ Page Language="C#" %>
<%@ Assembly Name="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
PublicKeyToken=71E9BCE111E9429C" %>
<%@ Import Namespace="Microsoft.SharePoint" %>
<%@ Import Namespace="Microsoft.SharePoint.Administration" %>
<%@ Import Namespace="System.ComponentModel" %>
<%@ Import Namespace="System.Collections.Generic" %>

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

<script runat="server">
  void Page_Load(object sender, EventArgs e)
  {
    SPWebServiceCollection wsc = new SPWebServiceCollection(SPFarm.Local);

    foreach (SPWebService ws in wsc)
    {
      SPApplicationPoolCollection apc = ws.ApplicationPools;
```

(continued)

Listing 1-2 *(continued)*

```
        foreach (SPApplicationPool ap in apc)
        {
          Response.Write(ap.Name);
          Response.Write("</br>");
        }
      }
    }
  </script>

  <html xmlns="http://www.w3.org/1999/xhtml">
  <head runat="server">
    <title>Untitled Page</title>
  </head>
  <body>
    <form id="form1" runat="server">
    <div>
    </div>
    </form>
  </body>
  </html>
```

SPIisWebSite

SharePoint runs on top of IIS. As such, it uses IIS web sites. An IIS web site is an entry point into IIS and is configured to listen for incoming requests on a specific port over a specific IP address and/or with a specific host header. Upon installation, IIS creates an IIS web site named Default Web Site and configures it to listen for incoming requests on port 80. You have the option of creating one or more IIS web sites and configuring them to listen for incoming requests on other ports and/or IP addresses supported on the IIS web server.

One of the great things about IIS web sites is that their security settings can be configured independently. For example, you can have an IIS web site such as Default Web Site that acts as an Internet-facing web site for your company, allowing anonymous users to access its contents. You can then create another IIS web site to act as an intranet-facing web site and configure it to use integrated Windows authentication, allowing only users with Windows accounts on the web server or a trusted domain to access its content.

SharePoint's object model comes with a class named SPIisWebSite whose instances represent IIS web sites. The following list presents the public properties of this class:

- **Exists.** This gets a Boolean value that specifies whether the IIS web site that the SPIisWebSite object represents exists in the metabase.
- **InstanceId.** This gets an integer value that uniquely identifies the IIS web site that the SPIisWebSite represents.
- **ServerBindings.** This gets or sets a string array where each string in the array specifies a server binding that the IIS web site that the SPIisWebSite object represents serves.

- **ServerComment.** This gets or sets a string that contains the display name of the IIS web site that the SPIisWebSite object represents.
- **ServerState.** This gets an SPIisServerState enumeration value that specifies the server state. The possible values are Continuing, Paused, Pausing, Started, Starting, Stopped, and Stopping.

The following list presents the public methods of the SPIisWebSite class:

- **Provision.** This provisions (creates) the IIS web site that the SPIisWebSite object represents.
- **Unprovision.** This unprovisions (removes) the IIS web site that the SPIisWebSite object represents.
- **Update.** This commits all changes to the registry.

You can create an instance of this class to access a specified IIS web site from within your managed code as shown in Listing 1-3.

Listing 1-3: An application page that shows how to access an IIS web site

```
<%@ Page Language="C#" %>
<%@ Assembly Name="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
PublicKeyToken=71E9BCE111E9429C" %>
<%@ Import Namespace="Microsoft.SharePoint" %>
<%@ Import Namespace="Microsoft.SharePoint.Administration" %>
<%@ Import Namespace="System.Globalization" %>

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

<script runat="server">
  void Page_Load(object sender, EventArgs e)
  {
    HttpContext context = HttpContext.Current;
    int instanceId = int.Parse(context.Request.ServerVariables["INSTANCE_ID"],
                               NumberFormatInfo.InvariantInfo);
    SPIisWebSite site = new SPIisWebSite(instanceId);
    lbl.Text = "Exists: " + site.Exists + "<br/>";
    lbl.Text += "InstanceId: " + site.InstanceId + "<br/>";
    lbl.Text += "Server bindings served by this IIS Web site:<br/>";
    foreach (string serverBinding in site.ServerBindings)
    {
      lbl.Text += "    Server Binding  " +
                  serverBinding + "<br/>";
    }
    lbl.Text += "<br/>";
    lbl.Text += "ServerComment: " + site.ServerComment + "<br/>";
    lbl.Text += "Server State: " + site.ServerState + "<br/>";
  }
</script>

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

(continued)

Listing 1-3 *(continued)*

```
<head runat="server">
  <title>Untitled Page</title>
</head>
<body>
  <form id="form1" runat="server">
  <div>
    <asp:Label ID="lbl" runat="server" />
  </div>
  </form>
</body>
</html>
```

As Listing 1-3 shows, the Page_Load method of this application page first accesses the current ASP.NET HTTP context as shown next:

```
HttpContext context = HttpContext.Current;
```

Next, it uses the current ASP.NET HTTP context to access the instance ID of the current IIS web site, which hosts the current SharePoint web application:

```
    int instanceId = int.Parse(context.Request.ServerVariables["INSTANCE_ID"],
                               NumberFormatInfo.InvariantInfo);
```

Then, it instantiates an SPIisWebSite object to represent the current IIS web site:

```
    SPIisWebSite site = new SPIisWebSite(instanceId);
```

Next, it iterates through the properties of this SPIisWebSite and prints their values:

```
    lbl.Text = "Exists: " + site.Exists + "<br/>";
    lbl.Text += "InstanceId: " + site.InstanceId + "<br/>";
    lbl.Text += "Server bindings served by this IIS Web site:<br/>";
    foreach (string serverBinding in site.ServerBindings)
    {
      lbl.Text += "    Server Binding  " +
                  serverBinding + "<br/>";
    }
    lbl.Text += "<br/>";
    lbl.Text += "ServerComment: " + site.ServerComment + "<br/>";
    lbl.Text += "Server State: " + site.ServerState + "<br/>";
```

As the following application page shows, you can also update the properties of an IIS web site:

```
<%@ Page Language="C#" %>
<%@ Assembly Name="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
PublicKeyToken=71E9BCE111E9429C" %>
<%@ Import Namespace="Microsoft.SharePoint" %>
<%@ Import Namespace="Microsoft.SharePoint.Administration" %>
```

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

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

<script runat="server">
  void Page_Load(object sender, EventArgs e)
  {
    HttpContext context = HttpContext.Current;
    int instanceId = int.Parse(context.Request.ServerVariables["INSTANCE_ID"],
                               NumberFormatInfo.InvariantInfo);
    SPIisWebSite site = new SPIisWebSite(instanceId);
    site.ServerComment = "My " + site.ServerComment;
    site.Update();
  }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Untitled Page</title>
</head>
<body>
  <form id="form1" runat="server">
  <div>
    <asp:Label ID="lbl" runat="server" />
  </div>
  </form>
</body>
</html>
```

As this code listing shows, this application page first creates an SPIisWebSite object to represent the current IIS web site as discussed earlier. Then, it sets the value of the ServerComment property on this SPIisWebSite object. Finally, it invokes the Update method on this SPIisWebSite object to commit the change.

SPWebApplication

SharePoint takes an IIS web site through a one-time configuration process to enable it to host SharePoint sites. Such an IIS web site is known as a web application in SharePoint terminology. This configuration process adds a wildcard application map to the IIS metabase to have IIS route all incoming requests targeted to the specified IIS web site to the aspnet_isapi.dll ISAPI extension module. Because this ISAPI extension module routes the requests to ASP.NET in turn, it ensures that all requests go through the ASP.NET HTTP Runtime Pipeline and are properly initialized with ASP.NET execution context before they are routed to SharePoint. This avoids a lot of the awkward problems associated with request processing in previous versions of SharePoint.

The SharePoint object model comes with a class named SPWebApplication that you can use to program against web applications from your managed code.

The following list describes some of the public properties of the SPWebApplication class:

- **AllowAccessToWebPartCatalog.** This gets or sets a Boolean value that specifies whether sites in the web application can use Web parts from the global Web part catalog.
- **AllowPartToPartCommunication.** This gets or sets a Boolean value that specifies whether the web application allows Web parts to communicate with each other to share data.
- **ApplicationPool.** This gets or sets the SPApplicationPool object that represents the application pool in which the web application is running.
- **BlockedFileExtensions.** This gets a Collection<string> object that contains the list of blocked file extensions. Files with such file extensions cannot be uploaded to the sites in the web application or downloaded from the sites in the web application.
- **ContentDatabases.** This gets the SPContentDatabaseCollection collection that contains the SPContentDatabase objects that represent the content databases that are available to the web application.
- **DaysToShowNewIndicator.** This gets or sets an integer value that specifies how many days the New icon is displayed next to new list items or documents.
- **DefaultTimeZone.** This gets or sets an integer value that specifies the default time zone for the web application.
- **DisplayName.** This gets the string that contains the display name of the web application.
- **Farm.** This gets the SPFarm object that represents the SharePoint farm in which the web application resides.
- **Features.** This gets the SPFeatureCollection collection that contains the SPFeature objects that represent the features activated for the web application.
- **Id.** This gets or sets the GUID that uniquely identifies the web application.
- **IsAdministrationWebApplication.** This gets or sets a Boolean value that indicates whether the web application is the Central Administration application.
- **Name.** This gets or sets a string that contains the name of the web application.
- **Properties.** This gets a property bag that is used to store properties for the web application. SharePoint automatically stores and retrieves the custom objects that you place in this property bag.
- **Sites.** This gets a reference to the SPSiteCollection collection that contains the SPSite objects that represent all site collections in the web application.
- **Status.** This gets or sets the SPObjectStatus enumeration value that specifies the status of the web application. The possible values are Disabled, Offline, Online, Provisioning, Unprovisioning, and Upgrading.
- **WebService.** This gets a reference to the SPWebService object that contains the web application.

The following list presents the public methods of the SPWebApplication class:

- **Delete.** This deletes the web application that the SPWebApplication object represents.
- **GrantAccessToProcessIdentity.** This takes a string that contains a username as its argument and grants the user access to the process identity of the SharePoint web application. This basically gives the user full control.
- **Lookup.** This takes a string that contains the URL of a web application and returns a reference to the SPWebApplication object that represents the web application.
- **Provision.** This provisions (creates) the web application that the SPWebApplication object represents on the local server.
- **Unprovision.** This unprovisions (removes) the web application that the SPWebApplication object represents from all local IIS web sites.
- **Update.** This serializes the state of the web application and propagates changes to all web servers in the server farm.
- **UpdateCredentials.** This commits new credentials (username or password) to the database.
- **UpdateMailSettings.** This commits the new email settings. These email settings are used to send emails.

Use one of the following methods to access the SPWebApplication object that represents a given SharePoint web application:

- The SPSite object that represents a SharePoint site collection exposes a property named WebApplication. WebApplication returns a reference to the SPWebApplication object that represents the SharePoint web application that hosts the site collection.
- The SPWeb object that represents a SharePoint site exposes a property named WebApplication returning a reference to the SPWebApplication object that represents the SharePoint web application that hosts the site.
- The SharePoint object model comes with a class named SPWebService that mainly acts as a container for SPWebApplication objects representing a SharePoint farm's collection of web applications. The SPWebService object for a given SharePoint farm exposes a collection property of type SPWebApplicationCollection named WebApplications that contains the farm's web applications. Use the name or GUID of a web application as an index into this collection to return a reference to the SPWebApplication object that represents the web application.
- The SPWebApplication class exposes a static method named Lookup. This method takes a URI object that specifies the URI of a web application and then returns the SPWebApplication object that represents it.

The following code listing presents a web page that uses the SharePoint object model to display the names of all web applications in the local farm:

```
<%@ Page Language="C#" %>
<%@ Assembly Name="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
PublicKeyToken=71E9BCE111E9429C" %>
<%@ Import Namespace="Microsoft.SharePoint" %>
<%@ Import Namespace="Microsoft.SharePoint.Administration" %>

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

<script runat="server">
  void Page_Load(object sender, EventArgs e)
  {
    SPWebServiceCollection wsc = new SPWebServiceCollection(SPFarm.Local);
    foreach (SPWebService ws in wsc)
    {
      SPWebApplicationCollection wac = ws.WebApplications;
      foreach (SPWebApplication wa in wac)
      {
        Response.Write(wa.Name);
        Response.Write("<br/>");
      }
    }
  }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Untitled Page</title>
</head>
<body>
  <form id="form1" runat="server">
  <div>
  </div>
  </form>
</body>
</html>
```

SPWebApplicationBuilder

Use an instance of the SPWebApplicationBuilder class to create an instance of the SPWebApplication class. The SPWebApplicationBuilder instance automatically provides default values for all the required settings, allowing you to override only the desired settings.

The following list presents the public properties of the SPWebApplicationBuilder class:

- **AllowAnonymousAccess.** This gets or sets a Boolean value that indicates whether anonymous users are allowed to access the new web application.

- **ApplicationPoolId.** This gets or sets a string that contains the GUID that uniquely identifies the application pool in which the new web application is created.
- **ApplicationPoolPassword.** This gets or sets the password of the Windows account under which the new application pool for the web application is to run.
- **ApplicationPoolUsername.** This gets or sets a string that contains the username of the Windows account under which the new application pool for the new web application is to run.
- **CreateNewDatabase.** This gets or sets a Boolean value that indicates whether to create a new content database for the web application.
- **DatabaseName.** This gets or sets a string that specifies the name for the new content database.
- **DatabasePassword.** This gets or sets a string that specifies the password for the new content database.
- **DatabaseServer.** This gets or sets a string that contains the database server name and instance in which to create the new content database.
- **DatabaseUsername.** This gets or sets a string that contains the username for the new content database.
- **Id.** This gets or sets the GUID that uniquely identifies the web application.
- **IdentityType.** This gets or sets an IdentityType enumeration value that specifies the process identity type of the application pool for the web application. The possible values are LocalService, LocalSystem, NetworkService, and SpecificUser.
- **Port.** This gets or sets an integer that specifies the port number of the new web application.
- **RootDirectory.** This gets or sets the DirectoryInfo object that represents the file system directory in which to install static files such as web.config for the new web application.
- **ServerComment.** This gets or sets a string that contains the server comment for the web application.
- **WebService.** This gets or sets the SPWebService object that represents the web service that contains the web application.

The following list presents the methods of the SPWebApplicationBuilder class:

- **Create.** This takes no arguments, uses the specified settings, creates a new web application, and returns a WebApplication object that represents the newly created web application.
- **ResetDefaults.** This takes no arguments and initializes all values with the best defaults that SharePoint can determine.

The following code listing creates and provisions a new web application:

```
<%@ Page Language="C#" %>
<%@ Assembly Name="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
PublicKeyToken=71E9BCE111E9429C" %>
<%@ Import Namespace="Microsoft.SharePoint" %>
<%@ Import Namespace="Microsoft.SharePoint.Administration" %>
<%@ Import Namespace="System.ComponentModel" %>
<%@ Import Namespace="System.Collections.Generic" %>

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

<script runat="server">
  void Page_Load(object sender, EventArgs e)
  {
    SPWebApplicationBuilder wab = new SPWebApplicationBuilder(SPFarm.Local);
    wab.Port = 12000;
    SPWebApplication wa = wab.Create();
    wa.Provision();
  }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
  <title>Untitled Page</title>
</head>
<body>
  <form id="form1" runat="server">
  <div>
  </div>
  </form>
</body>
</html>
```

SPWebService

The SPWebService acts as a container for SPWebApplication objects that represents one or more web applications in a SharePoint farm. The following list presents the public properties of this class:

- **AdministrationService.** This is a static property that gets a reference to the SPWebService object that contains the SPWebApplication object that represents the SharePoint Central Administration web application.
- **ApplicationPools.** This gets a reference to the SPApplicationPoolCollection collection that contains the SPApplicationPool objects that represent the IIS application pools available to the web service that this SPWebService object represents.
- **ContentService.** This gets a reference to the SPWebService object that contains the SPWebApplication objects that represent the content web applications.

- **DefaultDatabaseInstance.** This gets or sets a reference to the SPDatabaseServiceInstance object that represents the default named SQL Server installation for new content databases.
- **DefaultDatabasePassword.** This gets or sets a string that contains the default password for new content databases.
- **DefaultDatabaseUsername.** This gets or sets a string that contains the default username for new content databases.
- **DisplayName.** This gets a string that contains the display name of the web service that this SPWebService object represents.
- **Farm.** This gets a reference to the SPFarm object that represents the SharePoint farm where this web service resides.
- **Features.** This gets a reference to the SPFeatureCollection collection that contains SPFeature objects that represent farm-level scoped features.
- **Id.** This gets or sets the GUID that uniquely identifies this web service.
- **Name.** This gets or sets a string that contains the name of this web service. The name of the web service uniquely identifies the service.
- **Properties.** This gets a reference to a hash table where you can store arbitrary name/value pairs. SharePoint automatically takes care of persistence and retrieval of this pair just like it does for any other persistable SharePoint data.
- **Status.** This gets or sets the SPObjectStatus enumeration value that specifies the status of this web service. The possible values are Disabled, Offline, Online, Provisioning, Unprovisioning, and Upgrading.
- **WebApplications.** This gets a reference to an SPWebApplicationCollection collection that contains the SPWebApplication objects that represent the web applications that this web service contains.

The following list contains the public methods of the SPWebService class:

- **Delete.** This removes the web service from the SharePoint farm.
- **Provision.** This provisions the web service into the local server by making the necessary changes to the local server.
- **Unprovision.** This unprovisions the web service by making the necessary changes to the local server to clean up after deleting the object.
- **Update.** This commits and propagates the change made to this web service to all the machines in the farm.

The following code listing iterates through the application pools for each web service and prints their names:

```
<%@ Page Language="C#" %>
<%@ Assembly Name="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
PublicKeyToken=71E9BCE111E9429C" %>
<%@ Import Namespace="Microsoft.SharePoint" %>
<%@ Import Namespace="Microsoft.SharePoint.Administration" %>

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

<script runat="server">
  void Page_Load(object sender, EventArgs e)
  {
    SPWebServiceCollection wsc = new SPWebServiceCollection(SPFarm.Local);
    foreach (SPWebService ws in wsc)
    {
      SPApplicationPoolCollection apc = ws.ApplicationPools;
      foreach (SPApplicationPool ap in apc)
      {
        Response.Write(ap.Name);
        Response.Write("<br/>");
      }
    }
  }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
  <title>Untitled Page</title>
</head>
<body>
  <form id="form1" runat="server">
  <div>
  </div>
  </form>
</body>
</html>
```

ISAPIRuntime

As discussed earlier, when a request arrives, the HttpExtensionProc function of the ASP.NET ISAPI extension module is invoked and the Extension_Control_Block data structure is passed into it. The HttpExtensionProc function uses this data structure to communicate with IIS. The same Extension_Control_Block data structure is finally passed into the ProcessRequest method of the ISAPIRuntime object (see Listing 1-4).

Listing 1-4: The ProcessRequest method of ISAPIRuntime

```
public int ProcessRequest(IntPtr ecb, int iWRType)
{
  HttpWorkerRequest request =  CreateWorkerRequest(ecb, iWRType);
  HttpRuntime.ProcessRequest(request);
  return 0;
}
```

The ProcessRequest method of ISAPIRuntime, like the request processing method of any runtime class, first instantiates and initializes the appropriate HttpWorkerRequest object and then calls the ProcessRequest static method of an ASP.NET class named HttpRuntime and passes the newly instantiated HttpWorkerRequest object into it. HttpRuntime is the entry point into what is known as the ASP.NET HTTP Runtime Pipeline. This pipeline is discussed in detail in the next section.

ASP.NET HTTP Runtime Pipeline

The main responsibility of this pipeline is to process incoming requests and to generate the response text for the client. Listing 1-5 presents the internal implementation of the ProcessRequest method.

Listing 1-5: The ProcessRequest method of HttpRuntime

```
public static void ProcessRequest(HttpWorkerRequest wr)
{
  HttpContext context1 = new HttpContext(wr, true);
  IHttpHandler handler1 = HttpApplicationFactory.GetApplicationInstance(context1);

  if (handler1 is IHttpAsyncHandler)
  {
    IHttpAsyncHandler handler2 = (IHttpAsyncHandler)handler1;
    context1.AsyncAppHandler = handler2;
    handler2.BeginProcessRequest(context1, _handlerCompletionCallback,
                                 context1);
  }
  else
  {
    handler1.ProcessRequest(context1);
    FinishRequest(context1.WorkerRequest, context1, null);
  }
}
```

ProcessRequest performs these three main tasks:

- It instantiates an instance of an ASP.NET class named HttpContext, passing in the HttpWorkerRequest object. This HttpContext instance represents the ASP.NET execution context for the current request:

```
HttpContext context1 = new HttpContext(wr, true);
```

- It calls the GetApplicationInstance method of an ASP.NET class named HttpApplicationFactory to return an instance of an ASP.NET class that implements the IHttpHandler. This class inherits from another ASP.NET class named HttpApplication. ASP.NET represents each ASP.NET application with one or more HttpApplication objects. Also notice that HttpRuntime passes the HttpContext object into the GetApplicationInstance method:

```
IHttpHandler handler1 = HttpApplicationFactory.GetApplicationInstance(context1);
```

- It checks whether the object that GetApplicationInstance returns implements the IHttpAsyncHandler interface. If so, it calls the BeginProcessRequest method of the object to process the current request asynchronously:

```
httpApplication.BeginProcessRequest(context1, _handlerCompletionCallback,
                                    context1);
```

  If not, it calls the ProcessRequest method of the object to process the request synchronously:

```
handler1.ProcessRequest(context1);
```

  HTTP requests are always processed asynchronously to improve the performance and throughput of ASP.NET applications. As such, GetApplicationInstance always invokes the BeginProcessRequest method.

Notice that HttpRuntime passes the HttpContext object to HttpApplicationFactory, which is then passed to HttpApplication. HttpRuntime, HttpApplicationFactory, and HttpApplication, together with some other components discussed later in this chapter, form a pipeline of managed components known as the ASP.NET HTTP Runtime Pipeline. Each component in the pipeline receives the HttpContext object from the previous component, extracts the needed information from the object, processes the information, stores the processed information back into the object, and passes the object to the next component in the pipeline.

Each incoming HTTP request is processed with a distinct ASP.NET HTTP Runtime Pipeline. No two requests share the same pipeline. The HttpContext object provides the context within which an incoming HTTP request is processed.

Every application domain contains a single instance of the HttpApplicationFactory object. The HttpApplicationFactory class features a field named _theApplicationFactory that returns this single instance. The main responsibility of the GetApplicationInstance method of HttpApplicationFactory is to return an instance of a class to represent the current application. The type of this class depends on whether the application uses the global.asax file. This file is optional but every application can contain only a single instance of the file, which must be placed in the root directory of the application. The instances of the file placed in the subdirectories of the root directory are ignored.

If the root directory of an application doesn't contain the global.asax file, the GetApplicationInstance method returns an instance of an ASP.NET class named HttpApplication to represent the current application. If the root directory does contain the file, the GetApplicationInstance method returns an instance of a class that derives from the HttpApplication class to represent the application.

This HttpApplication-derived class is not one of the standard classes such as HttpApplication that ships with the ASP.NET Framework. Instead it's a class that the GetApplicationInstance method dynamically

generates on the fly from the content of the global.asax file. In other words, the GetApplicationInstance method automatically turns what's inside the global.asax file into a class that derives from HttpApplication, dynamically compiles this class into an assembly, and creates an instance of this dynamically generated compiled class to represent the application.

Listing 1-6 presents the internal implementation of the GetApplicationInstance method.

Listing 1-6: The internal implementation of the GetApplicationInstance method

```
public class HttpApplicationFactory
{
  private Stack _freeList;
  private int _numFreeAppInstances;

  internal static IHttpHandler GetApplicationInstance(HttpContext context)
  {
    GenerateAndCompileAppClassIfNecessary(context);
    HttpApplication appInstance = null;
    lock (_freeList)
    {
      if (_numFreeAppInstances > 0)
      {
        appInstance = (HttpApplication)_freeList.Pop();
        _numFreeAppInstances--;
      }
    }
    if (appInstance == null)
    {
      appInstance = InstantiateAppInstance();
      appInstance.InitializeAppInstance();
    }
    return appInstance;
  }
}
```

The GetApplicationInstance method performs these tasks. First, it calls the GenerateAndCompileAppClassIfNecessary method of the HttpApplicationFactory:

```
GenerateAndCompileAppClassIfNecessary(context);
```

The GenerateAndCompileAppClassIfNecessary method takes these steps. First, it determines whether the root directory of the application contains the global.asax file. If so, it instantiates an instance of an ASP.NET class named ApplicationBuildProvider. As the name suggests, ApplicationBuildProvider is the build provider responsible for generating the source code for the class that represents the global.asax file. As mentioned earlier, this class inherits from the HttpApplication base class.

After generating the source code for this class, the GenerateAndCompileAppClassIfNecessary method uses the AssemblyBuilder to dynamically compile this source code into an assembly and loads the assembly into the application domain. This dynamic code generation and compilation process is performed only when the first request hits the application. The process is repeated only if the timestamp of the global.asax and root web.config file of the application change. The timestamp of a file could change for a number of reasons. One obvious case is when you change the content of a file. Another less

obvious case is when you run a program such as an antivirus program that changes this timestamp even though the content of the file hasn't changed. If the timestamp of the global.asax file changes, the global.asax file is recompiled when the next request arrives.

Now back to Listing 1-6. The HttpApplicationFactory maintains a pool of HttpApplication-derived instances that represent the current application. In other words, more than one instance may be processing requests for the same application to improve the performance. When a request hits the application, GetApplicationInstance checks whether there's a free HttpApplication-derived instance available. If so, it returns that instance:

```
lock (_freeList)
{
  if (_numFreeAppInstances > 0)
  {
    appInstance = _freeList.Pop();
    _numFreeAppInstances--;
  }
}
```

Notice that the HttpApplicationFactory maintains the pool members in an internal instance of the Stack class named _freeList. As the previous listing shows, GetApplicationInstance simply calls the Pop method of this internal Stack object to access the free HttpApplication object and decrements the number of available HttpApplication objects by one. Also notice that GetApplicationInstance locks the Stack object before it accesses it because the same Stack object is accessed by multiple threads handling multiple requests.

If the pool has no free HttpApplication object available, GetApplicationInstance calls the InstantiateAppInstance method to create a new HttpApplication object:

```
if (appInstance == null)
{
  appInstance = InstantiateAppInstance();
  appInstance.InitializeAppInstance();
}
```

Notice that GetApplicationInstance finally calls the InitializeAppInstance method of the newly instantiated HttpApplication-derived object to initialize it. What this initialization process entails is discussed later in this chapter.

As mentioned, the GetApplicationInstance method parses the content of the global.asax file into a class that derives from HttpApplication. This means that you can have this method generate a different type of class by implementing and adding the global.asax file to your application. The global.asax file like many other ASP.NET files can have an associated code-behind class. You must use the Inherits attribute of the @Application directive inside the global.asax file to specify the complete information about this code-behind class, which contains the fully qualified name of the type of the class, including its complete namespace containment hierarchy and the complete information about the assembly that contains the class such as assembly name, version, culture, and public key token.

SharePoint uses the same approach to introduce a code-behind class named SPHttpApplication. This means that the objects that represent a given application are instances of a dynamically generated class that inherits from SPHttpApplication. As such, SharePoint applications are represented by instances of the SPHttpApplication class. This class like any code-behind class used in the global.asax file inherits from the HttpApplication class, which means that it exposes the same familiar API as HttpApplication class, which you can use to program against SharePoint applications. As such, we first need to study the HttpApplication class.

HttpApplication

The HttpApplication class is the base class for the class that represents the current application. As such, it provides its subclass with the base functionality that it needs to process the request. Recall that the GetApplicationInstance method of HttpApplicationFactory invokes the InitializeAppInstance method of HttpApplication on the newly instantiated HttpApplication-derived instance to initialize the instance. This method performs these tasks. First, it reads the contents of the <httpModules> sections of the configuration files. The <httpModules> section of a configuration file contains zero or more <add> child elements where each <add> element is used to register an ASP.NET component known as an HTTP module.

Listing 1-7 presents the portion of the <httpModules> section of the root web.config file, which registers the standard ASP.NET HTTP modules.

Listing 1-7: The <httpModules> section of the root web.config file

```
<configuration>
  <system.web>
    <httpModules>
      <add name="OutputCache" type="System.Web.Caching.OutputCacheModule" />
      <add name="Session" type="System.Web.SessionState.SessionStateModule" />
      <add name="WindowsAuthentication"
       type="System.Web.Security.WindowsAuthenticationModule" />
      <add name="FormsAuthentication"
       type="System.Web.Security.FormsAuthenticationModule" />
      ...
    </httpModules>
  </system.web>
</configuration>
```

An HTTP module is a class that implements the IHttpModule interface. Recall from the previous section that HttpRuntime creates an instance of a class named HttpContext, which provides the execution context for the current request. Each module must extract the required information from the HttpContext object, process the information, then store the processed information back in the object.

For example, the FormsAuthenticationModule authenticates the request, creates an IPrincipal object to represent the security context of the current request, and assigns this object to the User property of the HttpContext object. Listing 1-8 presents the definition of the IHttpModule interface.

Listing 1-8: The IHttpModule interface

```
public interface IHttpModule
{
  void Dispose();
  void Init(HttpApplication context);
}
```

InitializeAppInstance reads the contents of the <httpModules> section of the root web.config and other configuration files. Notice that the <add> element features two important attributes, that is, name and type, which respectively contain the friendly name and type information of the HTTP module. The friendly name provides an easy way to reference an HTTP module. The type attribute contains the complete type information needed to instantiate the module.

InitializeAppInstance first uses the value of the type attribute of each <add> element and .NET reflection to dynamically instantiate an instance of the associated HTTP module. It then calls the Init method of each HTTP module to initialize the module. As Listing 1-8 shows, every HTTP module implements the Init method.

The main function of the Init method of an HTTP module is to register event handlers for one or more of the events of the HttpApplication object. HttpApplication exposes a bunch of application-level events as described in the following table. Notice that this table lists the events in the order in which they are fired.

Event	Description
BeginRequest	Fires when ASP.NET begins processing the request
AuthenticateRequest	Fires when ASP.NET authenticates the request
PostAuthenticateRequest	Fires after ASP.NET authenticates the request
AuthorizeRequest	Fires when ASP.NET authorizes the request
PostAuthorizeRequest	Fires after ASP.NET authorizes the request
ResolveRequestCache	Fires when ASP.NET is determining whether the request can be serviced from the cache
PostResolveRequestCache	Fires after ASP.NET determines that the request can indeed be serviced from the cache bypassing the execution of the request handler
MapRequestHandler	Fires when ASP.NET determines the request handler
PostMapRequestHandler	Fires after ASP.NET determines the HTTP request handler
AcquireRequestState	Fires when ASP.NET acquires the request state
PostAcquireRequestState	Fires after ASP.NET acquires the request state
PreRequestHandlerExecute	Fires before ASP.NET executes the request handler
RequestHandlerExecute	Fires when ASP.NET executes the request handler

Event	Description
PostRequestHandlerExecute	Fires after ASP.NET executes the request handler
ReleaseRequestState	Fires when ASP.NET stores the request state
PostReleaseRequestState	Fires after ASP.NET stores the request state
UpdateRequestCache	Fires when ASP.NET is caching the response
PostUpdateRequestCache	Fires after ASP.NET caches the response so the next request is serviced from the cache bypassing the execution of the request handler
LogRequest	Fires when ASP.NET is logging the request
PostLogRequest	Fires after ASP.NET logs the request
EndRequest	Fires when ASP.NET ends processing the request
Disposed	Fires when ASP.NET releases all resources used by the application

The following three events could be raised at any time during the lifecycle of a request:

Event	Description
Error	Fires when an unhandled exception is thrown
PreSendRequestContent	Fires before ASP.NET sends the request content or body
PreSendRequestHeaders	Fires before ASP.NET sends the request HTTP headers

So far, I've covered the synchronous versions of the events of the HttpApplication object. The asynchronous version of each event follows this same format: AddOnXXXAsync where XXX is the placeholder for the event name. AddOnXXXAsync is a method that registers an event handler for the specified events. For example, in the case of the BeginRequest event, this method is AddOnBeginRequestAsync.

Now back to the InitializeAppInstance method of HttpApplication. So far, you've learned that this method instantiates and initializes the registered HTTP modules. Next it instantiates an instance of a class named ApplicationStepManager.

To understand the role of ApplicationStepManager, you need to understand how HttpApplication processes the request. The request processing of HttpApplication consists of a set of execution steps, which are executed in order. Each execution step is represented by an object of type IExecutionStep. The IExecutionStep interface features a method named Execute. As the name suggests, this method executes the step. Each IExecutionStep object is associated with a particular event of HttpApplication. For example, there's an IExecutionStep object associated with the BeginRequest event of HttpApplication.

As the name implies, ApplicationStepManager manages the building and executing the IExecutionStep objects associated with the HttpApplication events. ApplicationStepManager exposes two methods

named BuildSteps and ExecuteStage where the former creates these IExecutionStep objects and the latter calls the Execute method of these IExecutionStep objects to execute them. Now let's see who calls these two methods of ApplicationStepManager.

After instantiating the ApplicationStepManager, the InitializeAppInstance method of HttpApplication calls the BuildSteps method to create the IExecutionStep objects associated with the HttpApplication events. These objects are instantiated and added to an internal collection in the same order as their associated events. The order of these events was discussed in the previous table.

Next, you see who calls the ExecuteStage method. Recall that the ProcessRequest method of HttpRuntime calls the BeginProcessRequest method of the HttpApplication object as shown in the boldfaced section of the following code listing:

```
public static void ProcessRequest(HttpWorkerRequest wr)
{
  HttpContext context1 = new HttpContext(wr, true);
  IHttpHandler handler1 = HttpApplicationFactory.GetApplicationInstance(context1);

  IHttpAsyncHandler handler2 = (IHttpAsyncHandler)handler1;
  context1.AsyncAppHandler = handler2;
  handler2.BeginProcessRequest(context1, _handlerCompletionCallback, context1);
}
```

BeginProcessRequest internally calls the ExecuteStage method of the ApplicationStepManager, which, in turn, iterates through the internal collection that contains the IExecutionStep objects and calls their Execute methods in the order in which they were added to the collection.

SPHttpApplication

Launch the Windows Explorer and navigate to the physical root directory of your SharePoint web application, which is located in the following folder on the file system of the front-end web server:

```
Local_Drive:\inetpub\wwwroot\wss\VirtualDirectories
```

There you should see the global.asax file. SharePoint automatically adds this file every time you create a new SharePoint web application. If you open this file in your favorite editor, you should see the following:

```
<%@ Assembly Name="Microsoft.SharePoint"%>
<%@ Application Language="C#"
Inherits="Microsoft.SharePoint.ApplicationRuntime.SPHttpApplication" %>
```

As you can see, the global.asax file contains two directives. The @Assembly directive references the Microsoft.SharePoint.dll assembly, which contains the ApplicationRuntime namespace. Note that the Inherits attribute of the @Application directive instructs ASP.NET to use SPHttpApplication as the base class for the class that it dynamically creates and instantiates. Recall that instances of this dynamically generated class represent the SharePoint web application.

Listing 1-9 presents the internal implementation of the SPHttpApplication class.

Listing 1-9: The internal implementation of the SPHttpApplication class

```
public class SPHttpApplication : HttpApplication
{
  private ReaderWriterLock readerWriterLock = new ReaderWriterLock();
  private List<IVaryByCustomHandler> varyByCustomHandlers =
                                                  new List<IVaryByCustomHandler>();

  [SharePointPermission(SecurityAction.Demand, ObjectModel = true)]
  public sealed override string GetVaryByCustomString(HttpContext context,
                                                      string custom)
  {
    StringBuilder stringBuilder = new StringBuilder();
    readerWriterLock.AcquireReaderLock(-1);
    try
    {
      foreach (IVaryByCustomHandler varyByCustomHandler in varyByCustomHandlers)
      {
        stringBuilder.Append(
              varyByCustomHandler.GetVaryByCustomString(this, context, custom));
      }
    }
    finally
    {
      readerWriterLock.ReleaseReaderLock();
    }
    return stringBuilder.ToString();
  }

  [SharePointPermission(SecurityAction.Demand, ObjectModel = true)]
  public override void Init()
  {
    AppDomain currentAppDomain = AppDomain.CurrentDomain;
    if (currentAppDomain != null)
      currentAppDomain.UnhandledException +=
               new UnhandledExceptionEventHandler(UnhandledExceptionHandler);
  }

  public void RegisterGetVaryByCustomStringHandler(IVaryByCustomHandler
                                                   varyByCustomHandler)
  {
    if (varyByCustomHandler!= null)
    {
      readerWriterLock.AcquireWriterLock(-1);
      try
      {
        varyByCustomHandlers.Add(varyByCustomHandler);
      }
      finally
```

(continued)

Listing 1-9 *(continued)*

```
        {
          readerWriterLock.ReleaseWriterLock();
        }
      }
    }

    public void DeregisterGetVaryByCustomStringHandler(IVaryByCustomHandler
                                                       varyByCustomHandler)
    {
      if (varyByCustomHandler != null)
      {
        readerWriterLock.AcquireWriterLock(-1);
        try
        {
          varyByCustomHandlers.Remove(varyByCustomHandler);
        }
        finally
        {
          readerWriterLock.ReleaseWriterLock();
        }
      }
    }
  }
```

As you can see, the SPHttpApplication class overrides the GetVaryByCustomString and Init methods of the HttpApplication base class. Note that SPHttpApplication exposes a public method named RegisterGetVaryByCustomStringHandler that takes an IVaryByCustomHandler handler and adds it to an internal list. When ASP.NET finally invokes the GetVaryByCustomString method, this method iterates through these IVaryByCustomHandler handlers and invokes their GetVaryByCustomString methods, passing in these three parameters: a reference to the SPHttpApplication object, a reference to the current HttpContext object, and the string that contains the custom parameter. The GetVaryByCustomString method then collects the string values returned from the GetVaryByCustomString methods of these handlers in a string and returns this string to its caller.

SPRequestModule

When you create a new SharePoint web application, SharePoint automatically adds a web.config file to the root directory of the application. This file, among many other settings, includes the <httpModules> configuration section shown in Listing 1-10.

As discussed earlier, the InitializeAppInstance method of SPHttpApplication reads the content of the <httpModules> configuration section, instantiates the registered HTTP modules, and invokes their Init methods to allow them to register event handlers for one or more of the SPHttpApplication events. Note that the InitializeAppInstance method invokes the Init methods of the registered HTTP modules in the order in which these modules are added inside the <httpModules> configuration section. Therefore,

the HTTP modules that are added first get to register their event handlers first. This means that when the respective events are raised, their registered event handlers are the first to be invoked.

SharePoint comes with an HTTP module of its own named SPRequestModule that registers event handlers for most of these events. SharePoint uses these event handlers to initialize the SharePoint runtime environment. Because these initializations must be performed before any other ASP.NET HTTP modules get to perform their own tasks, SharePoint first adds the <clear/> element as the first element of the <httpModules> configuration section to clear up all the registered ASP.NET HTTP modules and then registers the SPRequestModule HTTP module (see Listing 1-10). SharePoint adds all the ASP.NET HTTP modules back inside the <httpModules> element after the SPRequestModule module. This ensures that the SPRequestModule HTTP module initializes the SharePoint runtime environment before any ASP.NET HTTP module gets involved.

Listing 1-10: The content of the web.config file

```
<configuration>
 <system.web>
    <httpModules>
      <clear />
      <add name="SPRequest"
      type="Microsoft.SharePoint.ApplicationRuntime.SPRequestModule,
            Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
            PublicKeyToken=71e9bce111e9429c" />
      <add name="OutputCache" type="System.Web.Caching.OutputCacheModule" />
      <add name="FormsAuthentication"
      type="System.Web.Security.FormsAuthenticationModule" />
      <add name="UrlAuthorization"
      type="System.Web.Security.UrlAuthorizationModule" />
      <add name="WindowsAuthentication"
      type="System.Web.Security.WindowsAuthenticationModule" />
      <add name="RoleManager" type="System.Web.Security.RoleManagerModule" />
      <add name="PublishingHttpModule"
      type="Microsoft.SharePoint.Publishing.PublishingHttpModule,
            Microsoft.SharePoint.Publishing, Version=12.0.0.0, Culture=neutral,
            PublicKeyToken=71e9bce111e9429c" />
      <add name="Session" type="System.Web.SessionState.SessionStateModule" />
    </httpModules>
  </system.web>
</configuration>
```

Listing 1-11 presents the portion of the internal implementation of the Init method of the SPRequestModule.

Listing 1-11: The portion of the implementation of the Init method of the SPRequestModule

```
public sealed class SPRequestModule : IHttpModule
{
  void IHttpModule.Init(HttpApplication app)
  {
    if (app is SPHttpApplication)
    {
      if (!_virtualPathProviderInitialized)
      {
        lock (_virtualServerDataInitializedSyncObject)
        {
          if (!_virtualPathProviderInitialized)
          {
            SPVirtualPathProvider virtualPathProvider =
                                                    new SPVirtualPathProvider();
            HostingEnvironment.RegisterVirtualPathProvider(virtualPathProvider);
            _virtualPathProviderInitialized = true;
          }
        }
      }
    }

    else
      return;

    app.BeginRequest += new EventHandler(this.BeginRequestHandler);
    app.PostResolveRequestCache +=
                            new EventHandler(this.PostResolveRequestCacheHandler);
    app.PostMapRequestHandler += new EventHandler(this.PostMapRequestHandler);
    app.ReleaseRequestState += new EventHandler(this.ReleaseRequestStateHandler);
    app.PreRequestHandlerExecute +=
                            new EventHandler(this.PreRequestExecuteAppHandler);
    app.PostRequestHandlerExecute +=
                            new EventHandler(this.PostRequestExecuteHandler);
    app.AuthenticateRequest += new EventHandler(this.AuthenticateRequestHandler);
    app.PostAuthenticateRequest +=
                            new EventHandler(this.PostAuthenticateRequestHandler);
    app.Error += new EventHandler(this.ErrorAppHandler);
    app.EndRequest += new EventHandler(this.EndRequestHandler);
  }
}
```

As you can see, the Init method first instantiates and registers an instance of a class named SPVirtualPathProvider and then registers event handlers for different events of the SPHttpApplication object representing the current SharePoint web application. As discussed earlier, these event handlers are responsible for initializing the SharePoint runtime environment.

SPVirtualPathProvider

SharePoint site pages are normally provisioned from a page template, which resides on the file system of each front-end web server on a SharePoint farm. A site page remains in a state known as *ghosted* until it is customized. When a request for a ghosted site page arrives, SharePoint checks whether this is the first

request for any site page instance of the associated page template. If so, SharePoint loads the page template from the file system into memory and passes it to the ASP.NET page parser, which in turn parses the page template into a dynamically generated class, compiles the class into an assembly, loads the assembly into the current application domain, instantiates an instance of the compiled class, and then passes the request into it for processing.

When another request for a site page that is an instance of the same page template arrives, the same compiled class is used to process the request. Because this compiled class represents the page template, which resides in the file system of the front-end web server, as opposed to the requested site page, it is as if the requester requested the page template as opposed to the site page, hence the name *ghosted*.

As a result, all requests for a SharePoint web application's site pages that are instances of the same page template are processed through the same compiled class, which is already loaded into memory. This improves the performance of the ghosted site pages dramatically. As you can see, SharePoint ghosted site pages are processed just like a normal ASP.NET page.

When you customize a SharePoint site page in the SharePoint Designer and save the changes, the SharePoint Designer stores the content of the site page file into the content database. This alters the state of the site page from ghosted to unghosted. When a request for an unghosted site page arrives, SharePoint must load the page from the content database and pass that to the ASP.NET page parser. In other words, the page template on the file system is no longer used to process a request for an unghosted page, hence the name *unghosted*.

Because the ASP.NET page parser in ASP.NET 1.1 can only parse pages loaded from the file system, the previous version of SharePoint comes with its own page parser, which allows it to parse files loaded from the content database. Unfortunately the SharePoint page parser is not as rich as the ASP.NET page parser. For example, it does not handle user controls.

ASP.NET 2.0 has changed all that. ASP.NET 2.0 has moved the logic that loads the page from the page parser to a dedicated component known as virtual path provider. This component is a class that inherits a base class named VirtualPathProvider. The ASP.NET 2.0 page parser communicates with these components through the VirtualPathProvider API. It is the responsibility of the configured virtual path provider, not the page parser, to load the ASP.NET page from whatever data source it is designed to work with.

SharePoint 2007 comes with an implementation of the VirtualPathProvider API named SPVirtualPathProvider that incorporates the logic that loads an ASP.NET page from

- The file system of the front-end web server if the page being loaded is an application page or a ghosted site page (and this is the first request for any instance of the associated page template)
- The content database if the page being loaded is an unghosted site page

This means that SharePoint 2007 no longer uses its own page parser. When a request for an ASP.NET page arrives, the SPVirtualPathProvider loads the page from the appropriate source and passes it along to the ASP.NET 2.0 page parser for parsing.

As the following excerpt from Listing 1-11 shows, the Init method of the SPRequestModule HTTP module instantiates an instance of the SPVirtualPathProvider and uses the RegisterVirtualPathProvider static method of the HostingEnvironment class to register it with ASP.NET:

```
SPVirtualPathProvider virtualPathProvider = new SPVirtualPathProvider();
HostingEnvironment.RegisterVirtualPathProvider(virtualPathProvider);
```

Take these steps if you need to customize the behavior of the SPVirtualPathProvider:

1. Implement a custom virtual path provider that inherits the VirtualPathProvider base class where your implementation of the methods of this base class should delegate to the associated methods of the previous virtual path providers. Keep in mind that ASP.NET chains the registered virtual path providers together.
2. Implement a custom HTTP module where your implementation of the Init method must use the RegisterVirtualPathProvider static method of the HostingEnvironment class to register your custom virtual path provider with ASP.NET.
3. Add your HTTP module after the SPRequestModule inside the <httpModules> configuration section of the web.config file of the SharePoint web application. This ensures that the SPRequestModule HTTP module gets to register the SPVirtualPathProvider first so this virtual provider path comes before your custom virtual path provider in the chain of virtual path providers. This allows your virtual path provider's implementation of the methods of the VirtualPathProvider base class to use the Previous property to delegate to the associated methods of the SPVirtualPathProvider.

IHttpHandlerFactory and IHttpHandler

HttpApplication features an event named MapRequestHandler, which like any other HttpApplication event is associated with an IExecutionStep object. The main responsibility of the Execute method of this IExecutionStep object is to find a class that either knows how to handle the current request or knows the class that knows how to handle the current request. The class that knows how to handle HTTP requests for a resource with a specified file extension is known as an HTTP handler. The class that knows the HTTP handler that handles HTTP requests for a resource with a specified file extension is known as an HTTP handler factory. All HTTP handlers implement an interface named IHttpHandler. Listing 1-12 presents the definition of this interface.

Listing 1-12: The IHttpHandler interface

```
public interface IHttpHandler
{
  void ProcessRequest(HttpContext context);
  bool IsReusable { get; }
}
```

The following table describes the members of the IHttpHandler interface:

Member	Description
ProcessRequest	The main function of this method is to process the request, that is, to generate the response text sent to the client.
IsReusable	This gets a Boolean value that specifies whether the same IHttpHandler instance can be reused to handle other requests.

All HTTP handler factories implement an interface named IHttpHandlerFactory. Listing 1-13 contains the definition of this interface.

Listing 1-13: The IHttpHandlerFactory interface

```
public interface IHttpHandlerFactory
{
  IHttpHandler GetHandler(HttpContext context, string requestType, string url,
                          string pathTranslated);
  void ReleaseHandler(IHttpHandler handler);
}
```

The GetHandler method takes four parameters: the current HttpContext object, the HTTP verb used to make the request, the virtual path of the requested file, and the physical path of the requested file.

As discussed earlier, the main responsibility of the Execute method of the IExecutionStep object associated with the MapRequestHandler event of HttpApplication is to determine the HTTP handler that knows how to process requests for the resource with the specified file extension or the HTTP handler factory that knows the HTTP handler. The Execute method uses the <httpHandlers> section of the configuration files to make this determination. Listing 1-14 shows the portion of the <httpHandlers> section of the web.config file of an ASP.NET application.

Listing 1-14: The portion of the <httpHandler> section

```
<configuration>
  <system.web>
    <httpHandlers>
      <add path="*.aspx" verb="*" type="System.Web.UI.PageHandlerFactory" />
      <add path="*.ashx" verb="*" type="System.Web.UI.SimpleHandlerFactory" />
      <add path="*.asmx" verb="*"
       type="System.Web.Services.Protocols.WebServiceHandlerFactory,
             System.Web.Services, Version=2.0.0.0, Culture=neutral,
             PublicKeyToken=b03f5f7f11d50a3a" />
      ...
    </httpHandlers>
  </system.web>
</configuration>
```

Note that the <add> element features three important attributes: path, verb, and type. The type attribute contains a comma-separated list of up to five substrings that provide the Execute method of the associated IExecutionStep object with the complete type information needed to instantiate the specified implementation of the IHttpHandlerFactory or IHttpHandler. The only required substring is the first substring, which contains the fully qualified name of the type of the specified implementation of the IHttpHandlerFactory or IHttpHandler.

The path attribute specifies the virtual path of the resource(s) that the specified implementation of the IHttpHandlerFactory or IHttpHandler handles. For example, as Listing 1-14 shows, PageHandlerFactory handles requests for ASP.NET pages, that is, files with the file extension .aspx. Or, WebServiceHandlerFactory handles requests for ASP.NET web services, that is, files with the file extension .asmx.

The verb attribute contains a comma-separated list of HTTP verbs. The * value specifies that all HTTP verbs are supported. For example, PageHandlerFactory handles requests for ASP.NET pages no matter what HTTP verb the client uses to make the request.

The Execute method of the IExecutionStep object associated with the MapRequestHandler event searches the content of the <httpHandlers> section for the IHttpHandlerFactory or IHttpHandler implementation that handles the request for the file with the specified file extension. For example, if the client has made the request for an ASP.NET page, that is, a file with the file extension .aspx, the Execute method will look for an <add> element whose path attribute has been set to *.aspx or something such as * that includes the file extension .aspx. The method then reads the value of the type attribute of this <add> element and uses that information and .NET reflection to dynamically instantiate an instance of the specified IHttpHandlerFactory or IHttpHandler implementation. For example, if the request is made for an ASP.NET page, the Execute method instantiates an instance of the PageHandlerFactory class.

What the Execute method does next depends on whether the type or class that the type attribute specifies is an HTTP handler or HTTP handler factory. If it's an HTTP handler, Execute simply calls the ProcessRequest method of the handler to process the request. Recall that every HTTP handler implements the ProcessRequest method of the IHttpHandler interface. If it's an HTTP handler factory, Execute first calls the GetHandler method of the handler factory to return the HTTP handler responsible for processing the request and then calls the ProcessRequest method of the HTTP handler to process the request.

For example, in the case of a request for a resource with the file extension .aspx, the Execute method calls the GetHandler method of PageHandlerFactory to return an instance of the HTTP handler responsible for processing the request. This HTTP handler is a class that inherits from the Page base class.

SPHttpHandler

ASP.NET comes with a special HTTP handler named DefaultHttpHandler, which allows you to custom handle the incoming requests. Doing so involves four steps:

1. Add a wildcard application map to the IIS metabase to have IIS route all requests to the aspnet_isapi.dll ISAPI extension module.
2. Implement a custom HTTP handler that inherits from DefaultHttpHandler.
3. Add the following <remove> element as the first child element of the <httpHandlers> configuration section of the web.config file at the root directory of your ASP.NET application:

```
<remove verb="GET,HEAD,POST" path="*" />
```

   As you can see, this <remove> element removes all the HTTP handler factories and HTTP handlers registered for handling requests made through any HTTP verb for resources with any file extension (path="*"). This effectively removes all the ASP.NET registered HTTP handler factories and HTTP handlers.

4. Add your custom HTTP handler immediately after the preceding <remove> child element.

SharePoint follows this same four-step process to custom handle all SharePoint requests. Following these steps, SharePoint comes with a custom HTTP handler named SPHttpHandler that inherits from the DefaultHttpHandler HTTP handler.

When you create a SharePoint web application, SharePoint automatically adds the <httpHandlers> configuration section shown in Listing 1-15 to the web.config file at the root directory of the web application.

Listing 1-15: The web.config file

```
<configuration>
 <system.web>
   <httpHandlers>
     <remove verb="GET,HEAD,POST" path="*" />

     <add verb="GET,HEAD,POST" path="*"
     type="Microsoft.SharePoint.ApplicationRuntime.SPHttpHandler,
           Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
           PublicKeyToken=71e9bce111e9429c" />
     ...

    </httpHandlers>
  </system.web>
</configuration>
```

So far you've learned how to configure IIS and ASP.NET to route all requests to a custom DefaultHttpHandler-derived HTTP handler for processing. This HTTP handler, like any other HTTP handler, exposes a method named ProcessRequest. As the name implies, this method processes the request. Listing 1-16 presents the internal implementation of the DefaultHttpHandler HTTP handler.

Listing 1-16: The DefaultHttpHandler HTTP handler

```
public class DefaultHttpHandler : IHttpAsyncHandler
{
  private HttpContext context;
  private NameValueCollection executeUrlHeaders;

  public virtual IAsyncResult BeginProcessRequest(HttpContext context,
                               AsyncCallback asyncCallback, object asyncState)
  {
    this.context = context;
    string virtualPath = OverrideExecuteUrlPath();
    ...
    return context.Response.BeginExecuteUrlForEntireResponse(virtualPath,
                                 executeUrlHeaders, asyncCallback, asyncState);
    }
    ...
  }

  public virtual void EndProcessRequest(IAsyncResult asyncResult);

  public virtual void OnExecuteUrlPreconditionFailure() { }

  public virtual string OverrideExecuteUrlPath()
  {
    return null;
  }

  protected NameValueCollection ExecuteUrlHeaders { get; }
  protected HttpContext Context {get;}
  public virtual bool IsReusable {get;}
}
```

Note that this HTTP handler implements the IHttpAsyncHandler interface, which in turn extends the standard IHttpHandler to add support for asynchronous request processing. Listing 1-17 presents the definition of this interface.

Listing 1-17: The IHttpAsynchronousHandler interface

```
public interface IHttpAsyncHandler : IHttpHandler
{
  IAsyncResult BeginProcessRequest(HttpContext context, AsyncCallback cb,
                                   object extraData);
  void EndProcessRequest(IAsyncResult result);
}
```

As you can see, the IHttpAsyncHandler interface exposes the following two methods:

- **BeginProcessRequest.** This method takes a reference to the current HttpContext object, a reference to an AsyncCallback delegate, and a reference to an optional object and returns an IAsyncResult instance. The caller of this method must wrap a callback method in the AsyncCallback delegate. When the HTTP handler is done with processing the current request, it automatically invokes this callback method, passing in the same IAsyncResult object that the BeginProcessRequest method returns. This object exposes a property named AsyncState that references the optional object that the caller passed into the BeginProcessRequest as the third argument.
- **EndProcessRequest.** This method takes an IAsyncResult object. It is the responsibility of the callback method wrapped in the AsyncCallback delegate to call the EndProcessRequest method.

In effect, the IHttpAsyncHandler interface allows you to process the request asynchronously where your callback method is automatically invoked after the handler is done with processing request. Now back to the DefaultHttpHandler HTTP handler's implementation of the BeginProcessRequest method as shown in Listing 1-16. As you can see, this method first calls the OverrideExecuteUrlPath method:

```
string virtualPath = OverrideExecuteUrlPath();
```

The DefaultHttpHandler HTTP handler's implementation of the OverrideExecuteUrlPath method does not do anything. It simply returns null. It is the responsibility of the HTTP handler that derives from the DefaultHttpHandler to override this method where it must take two important steps. First, it must populate the ExecuteUrlHeaders NameValueCollection property with the appropriate request headers. Second, it must return the appropriate virtual path for the request.

After calling the OverrideExecuteUrlPath method and accessing the virtual path for the request, the BeginProcessRequest method of DefaultHttpHandler invokes the BeginExecuteUrlForEntireResponse method on the HttpResponse object that represents the current HTTP request, passing in the virtual path of the requested resource, the NameValueCollection collection that contains the names and values of the request headers, the AsyncCallback delegate discussed earlier, and the object that was passed in as the third argument of the BeginProcessRequest method:

```
return context.Response.BeginExecuteUrlForEntireResponse(virtualPath,
                                  executeUrlHeaders, asyncCallback, asyncState);
```

Listing 1-18 presents the internal implementation of the BeginExecuteUrlForEntireResponse method.

Listing 1-18: The BeginExecuteUrlForEntireResponse method

```
public sealed class HttpResponse
{
  internal IAsyncResult BeginExecuteUrlForEntireResponse(string pathOverride,
                          NameValueCollection requestHeaders,
                          AsyncCallback asyncCallback, object asyncState)
  {
    string userName = context.User.Identity.Name;
    string authenticationType = context.User.Identity.AuthenticationType;

    string rewrittenUrl = Request.RewrittenUrl;
    if (pathOverride != null)
      rewrittenUrl = pathOverride;

    string requestHeadersStr;
    if (requestHeaders != null)
    {
      if (requestHeaders.Count > 0)
      {
        StringBuilder stringBuilder = new StringBuilder();
        for (int i = 0; i < requestHeaders.Count; i++)
        {
          stringBuilder.Append(requestHeaders.GetKey(i));
          stringBuilder.Append(": ");
          stringBuilder.Append(requestHeaders.Get(i));
          stringBuilder.Append("\r\n");
        }
        requestHeadersStr = stringBuilder.ToString();
      }
    }

    byte[] entityBody = context.Request.EntityBody;

    IAsyncResult asyncResult =
       this.workerRequest.BeginExecuteUrl(
                          rewrittenUrl, null, requestHeadersStr, true, true,
                          this.workerRequest.GetUserToken(), userName,
                     authenticationType, entityBody, asyncCallback, asyncState);
    headersWritten = true;
    ended = true;
    return asyncResult;
  }
}
```

This method first accesses the username of the Windows account under which the current request is executing:

```
string userName = context.User.Identity.Name;
```

Next, it determines the authentication type:

```
string authenticationType = context.User.Identity.AuthenticationType;
```

Then, it specifies the virtual path for the request:

```
string rewrittenUrl = Request.RewrittenUrl;
if (pathOverride != null)
  rewrittenUrl = pathOverride;
```

Next, it iterates through the NameValueCollection collection and creates a string consisting of an "\r\n" separated list of substrings, each substring consisting of two parts separated by a colon character where the first part is a request header name and the second part is the value of the header:

```
string requestHeadersStr = null;
if (requestHeaders != null)
{
  if (requestHeaders.Count > 0)
  {
    StringBuilder stringBuilder = new StringBuilder();
    for (int i = 0; i < requestHeaders.Count; i++)
    {
      stringBuilder.Append(requestHeaders.GetKey(i));
      stringBuilder.Append(": ");
      stringBuilder.Append(requestHeaders.Get(i));
      stringBuilder.Append("\r\n");
    }
    requestHeadersStr = stringBuilder.ToString();
  }
}
```

Then, it determines the body of the request:

```
byte[] entityBody = context.Request.EntityBody;
```

Finally, it invokes the BeginExecuteUrl method on the HttpWorkerRequest to execute the specified URL, passing in the virtual path of the requested resource, request headers, username of the Windows account, authentication type, body of the request, the AsyncCallback delegate discussed earlier, and the optional object discussed earlier:

```
IAsyncResult asyncResult =
       this.workerRequest.BeginExecuteUrl(
                          rewrittenUrl, null, requestHeadersStr, true, true,
                          this.workerRequest.GetUserToken(), userName,
                      authenticationType, entityBody, asyncCallback, asyncState);
```

Recall that the HttpWorkerRequest facilitates the communications between ASP.NET and its host, which is IIS in this case. In other words, the BeginExecuteUrl method of the HttpWorkerRequest method in effect starts a new request with a new URL and new request headers.

As you can see, the DefaultHttpHandler HTTP handler provides a powerful approach to custom request processing where you can restart a whole new request with a new URL and new set of custom request headers. Your custom DefaultHttpHandler-derived HTTP handler must override the OverrideExecuteUrlPath method of the DefaultHttpHandler where it must determine the new URL and the new set of custom request headers needed for the new request.

One of the great things about using a DefaultHttpHandler-derived HTTP handler is that all requests, regardless of the HTTP verbs used to make them and regardless of the file extensions of the requested resources, go through all the registered HTTP modules. Each HTTP module performs a specific preprocessing task on each request and stores the outcome of this task in the HttpContext object. For example, the FormAuthenticationModule HTTP module instantiates and initializes an IPrinciple object and assigns it to the User property of the HttpContext object to represent the security context of the request. This means that the new request for the new URL with new set of custom request headers now carries with it the outcome of all the HTTP modules that the previous request went through. As a matter of fact, as you can see from Listing 1-18, the BeginExecuteUrlForEntireResponse method of the current HttpResponse object passes the username and authentication types as arguments into the BeginExecuteUrl method of the HttpWorkerRequest object. In other words, all requests, regardless of the HTTP verbs used to make them and regardless of the file extensions of the requested resources, get authenticated and authorized through the same ASP.NET authentication and authorization modules.

SPHttpHandler inherits from DefaultHttpHandler and overrides its OverrideExecuteUrlPath method as shown in Listing 1-19 to determine the new URL and new set of custom request headers for the new request.

Listing 1-19: The internal implementation of SPHttpHandler

```
public sealed class SPHttpHandler : DefaultHttpHandler
{
  public override string OverrideExecuteUrlPath()
  {
    base.ExecuteUrlHeaders.Add("VTI_TRANSLATE",
                          base.Context.Request.ServerVariables["HTTP_TRANSLATE"]);
    base.ExecuteUrlHeaders.Remove("TRANSLATE");
    base.ExecuteUrlHeaders.Add("VTI_REQUEST_METHOD",
                               base.Context.Request.HttpMethod);
    bool invalidUnicode = false;
    string url = SPHttpUtility.UrlPathEncode(base.Context.Request.RawUrl,
                                             true, true, ref invalidUnicode);
    base.ExecuteUrlHeaders.Add("VTI_SCRIPT_NAME", url);
    base.ExecuteUrlHeaders.Add(this.AppDomainIdHeader,
                AppDomain.CurrentDomain.Id.ToString(CultureInfo.InvariantCulture));
    base.ExecuteUrlHeaders.Add(this.ContentLengthHeader,
       base.Context.Request.ContentLength.ToString(CultureInfo.InvariantCulture));
    if (SPSecurity.AuthenticationMode == AuthenticationMode.Windows)
      base.ExecuteUrlHeaders.Add(this.AuthModeHeader, "Windows");

    else
```

(continued)

Listing 1-19 *(continued)*

```
      {
        uint num;
        base.ExecuteUrlHeaders.Add(this.AuthModeHeader, "Forms");
        if (SPSecurity.UseMembershipUserKey &&
            base.Context.User.Identity.IsAuthenticated)
        {
          string userKey = UTF7Encode(SPUtility.GetFullUserKeyFromLoginName(
                                                  base.Context.User.Identity.Name));
          base.ExecuteUrlHeaders.Add(this.MembershipUserKeyHeader, userKey);
        }
        string membershipProviderName = UTF7Encode(Membership.Provider.Name);
        base.ExecuteUrlHeaders.Add(this.AuthProviderHeader, membershipProviderName);
        string roles;
        SPSecurity.GetRolesForUser(out num, out roles);
        if (num > 0)
        {
          base.ExecuteUrlHeaders.Add(this.RoleCountHeader,
                                     num.ToString(CultureInfo.InvariantCulture));
          base.ExecuteUrlHeaders.Add(this.RolesHeader, UTF7Encode(roles));
        }
      }
      HttpCookieCollection cookies = base.Context.Response.Cookies;
      if ((cookies != null) && (cookies.Count > 0))
      {
        bool supportsHttpOnly = SupportsHttpOnly();
        for (int i = 0; i < cookies.Count; i++)
        {
          string cookieName = this.ManagedCookiesHeader + "_" +
                        i.ToString(CultureInfo.InvariantCulture);
          string cookieValue = UTF7Encode(this.GetCookieString(cookies[i],
                                    supportsHttpOnly));
          base.ExecuteUrlHeaders.Add(cookieName, cookieValue);
        }
      }
      ...
    }
  }
```

The OverrideExecuteUrlPath method of SPHttpHandler, like the OverrideExecuteUrlPath method of any other DefaultHttpHandler-derived HTTP handler, first populates the ExecuteUrlHeader NameValueCollection collection with the appropriate header names and values. Recall that the BeginProcessRequest method of DefaultHttpHandler passes the content of this collection as a parameter into the BeginExecuteUrlForEntireResponse method of the current HttpResponse object.

Note that OverrideExecuteUrlPath also stores the authentication and authorization information into the ExecuteUrlHeader NameValueCollection collection:

```
if (SPSecurity.AuthenticationMode == AuthenticationMode.Windows)
  base.ExecuteUrlHeaders.Add(this.AuthModeHeader, "Windows");

else
{
  uint num;
  base.ExecuteUrlHeaders.Add(this.AuthModeHeader, "Forms");
  if (SPSecurity.UseMembershipUserKey &&
      base.Context.User.Identity.IsAuthenticated)
  {
    string userKey = UTF7Encode(SPUtility.GetFullUserKeyFromLoginName(
                                          base.Context.User.Identity.Name));
    base.ExecuteUrlHeaders.Add(this.MembershipUserKeyHeader, userKey);
  }
  string membershipProviderName = UTF7Encode(Membership.Provider.Name);
  base.ExecuteUrlHeaders.Add(this.AuthProviderHeader, membershipProviderName);
  string roles;
  SPSecurity.GetRolesForUser(out num, out roles);
  if (num > 0)
  {
    base.ExecuteUrlHeaders.Add(this.RoleCountHeader,
                               num.ToString(CultureInfo.InvariantCulture));
    base.ExecuteUrlHeaders.Add(this.RolesHeader, UTF7Encode(roles));
  }
}
```

OverrideExecuteUrlPath also stores response cookies into this collection:

```
HttpCookieCollection cookies = base.Context.Response.Cookies;
if ((cookies != null) && (cookies.Count > 0))
{
  bool supportsHttpOnly = SupportsHttpOnly();
  for (int i = 0; i < cookies.Count; i++)
  {
    string cookieName = this.ManagedCookiesHeader + "_" +
                  i.ToString(CultureInfo.InvariantCulture);
    string cookieValue = UTF7Encode(this.GetCookieString(cookies[i],
                              supportsHttpOnly));
    base.ExecuteUrlHeaders.Add(cookieName, cookieValue);
  }
}
```

In summary, thanks to the SPHttpHandler HTTP handler, regardless of whether it targets ASP.NET or non-ASP.NET resources, all requests go through all the HTTP modules in the ASP.NET HTTP Runtime Pipeline, including the SPRequestModule HTTP module. This has two important effects. First, because all requests go through the SPRequestModule HTTP module, they're all initialized with the SharePoint execution context. Second, because all requests go through all ASP.NET HTTP modules, they're all initialized with the ASP.NET execution context.

Developing Custom HTTP Handler Factories, HTTP Handlers, and HTTP Modules

Deep integration of SharePoint and ASP.NET 2.0 allows you to implement your own custom HTTP handler factories, HTTP handlers, and HTTP modules, and plug them into the ASP.NET HTTP Runtime Pipeline to customize this pipeline.

ASP.NET Dynamic Compilation

Let's begin our discussion with a simple example. Create a new ASP.NET web site in Visual Studio and add the simple ASP.NET web page named default.aspx to it, as shown in Listing 1-20.

Listing 1-20: The default.aspx page

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
Inherits="Default" %>
<html xmlns="http://www.w3.org/1999/xhtml">
<body>
  <form id="form1" runat="server">
    <div>
      <table>
        <tr>
          <td align="right">
            Display Name:</td>
          <td>
            <asp:TextBox runat="server" ID="DisplayNameTbx" /></td>
        </tr>
        <tr>
          <td align="right">
            Email:</td>
          <td>
            <asp:TextBox runat="server" ID="EmailTbx" /></td>
        </tr>
        <tr>
          <td colspan="2" align="center">
            <asp:Button runat="server" ID="SubmitBtn"
            OnClick="SubmitCallback" Text="Submit" />
          </td>
        </tr>
        <tr>
          <td colspan="2" align="left">
            <asp:Label runat="server" ID="Info" />
          </td>
        </tr>
      </table>
    </div>
  </form>
</body>
</html>
```

Now introduce a compilation error and hit F5 to compile the page. You should get the page shown in Figure 1-2, which has a link titled Show Complete Compilation Source. Click the link to access the code partly shown in Listing 1-21.

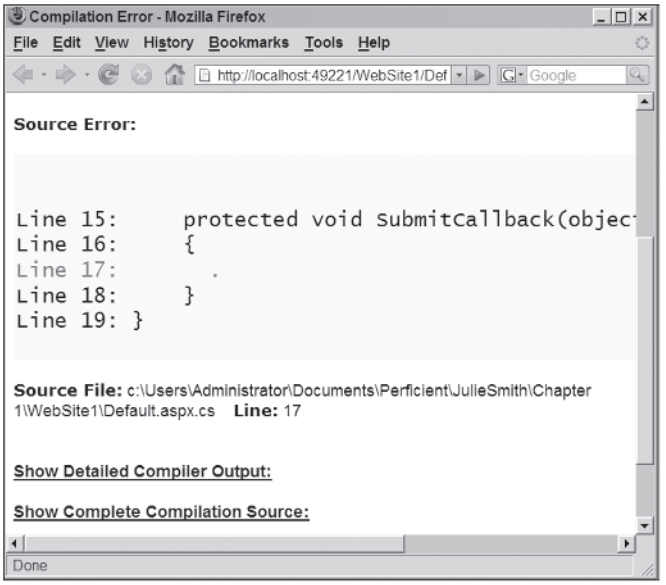

Figure 1-2: The error page

Listing 1-21: The Compilation Source

```
namespace ASP
{
  ...
  public class default_aspx : Default, System.Web.IHttpHandler
  {
    ...
    private TextBox @__BuildControlDisplayNameTbx()
    {
      TextBox @__ctrl;
      @__ctrl = new TextBox();
      this.DisplayNameTbx = @__ctrl;
      @__ctrl.ApplyStyleSheetSkin(this);
      @__ctrl.ID = "DisplayNameTbx";
      return @__ctrl;
    }
    ...
  }
}
```

When you hit F5, the ASP.NET build environment performs these tasks:

1. Parses the MySimplePage.aspx markup file shown in Listing 1-20.
2. Generates the source code partly shown in Listing 1-21. Notice that this source code contains a class named default_aspx belonging to a namespace named ASP. You can think of this class as the programmatic representation of the MySimplePage.aspx markup file shown in Listing 1-20.
3. Stores this source code in a file in the following directory on your machine:

```
%SystemRoot%\Microsoft.NET\Framework\versionNumber\ASP.NET Temporary Files
```

   By default, the ASP.NET build environment deletes this file after it compiles it into an assembly. However, you can change this default behavior by setting the Debug attribute on the @Page directive to true. This will allow you to view this file in a text editor.

4. Compiles the preceding source code file into an assembly and stores the assembly in the ASP.NET Temporary Files directory.
5. Loads the compiled assembly into the application domain that contains the ASP.NET application to make the dynamically generated default_aspx class available to the managed code running in the application domain.

In other words, the ASP.NET build environment converts the markup code shown in Listing 1-20 into the procedural code partly shown in Listing 1-21. You may be wondering why this conversion from markup to procedural code is necessary. As a matter of fact, the older web development technologies such as ASP don't do this conversion. Instead, they interpret the markup. This is very similar to what browsers do when they're displaying an HTML page. The browsers scan through the HTML page and interpret the HTML markup they run into. For example, when they see a <table> HTML element, they take it that they're asked to render a table. The great thing about markup programming is its convenience. Writing code in an HTML-like markup language is more convenient than writing code in a procedural language such as C# or VB.NET and significantly improves the developer productivity. Markup programming is also the basis on which visual designers such as Visual Studio operate. The disadvantage of markup programming is the performance-degradation due to the underlying interpretation mechanism.

This is where the ASP.NET build environment comes into play. The ASP.NET build environment allows you or visual designers to use markup programming to implement your ASP.NET web page and transparently compiles your markup code into procedural code. This allows you to enjoy both the convenience and productivity boost of markup programming and the performance boost of procedural programming. Who says you can't have your cake and eat it too?

The GetHandler method of the PageHandlerFactory internally uses the ASP.NET page parser to parse a requested ASP.NET page such as the one shown in Listing 1-20 into a class that inherits from the Page base class. The name of this class follows the ASP.NET internal naming convention. According to this convention, the name of the class consists of two parts. The first part is the name of the ASP.NET page being processed. The second part, on the other hand, is the file extension of the ASP.NET page, that is, .aspx. For example, the name of the HTTP handler that handles requests for the default.aspx file is default_aspx. All dynamically generated classes such as default_aspx belong to an assembly named ASP. To see this in action, create an ASP.NET Web Site project in Visual Studio and add a Web Form to this page. Then switch to the code-behind file for this page and start typing ASP as shown in Figure 1-3.

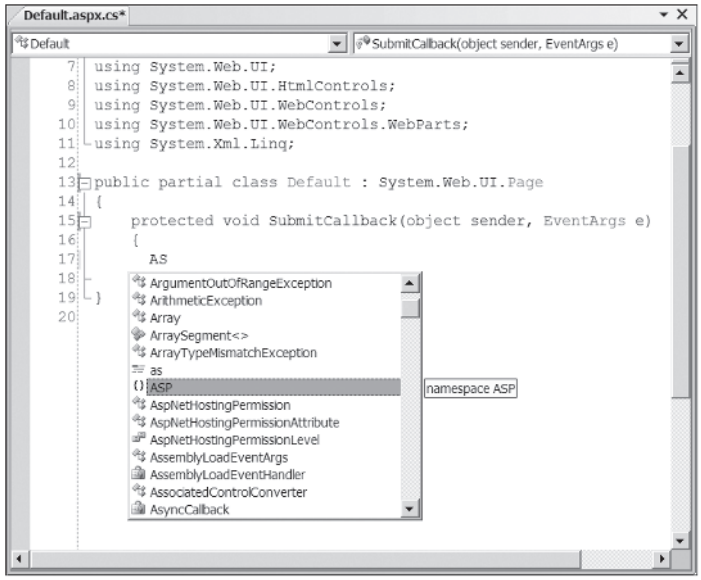

Figure 1-3: The default_aspx class

As you can see, this popup menu contains an entry for the ASP namespace. If you add a dot after ASP, you should see the menu shown in Figure 1-4.

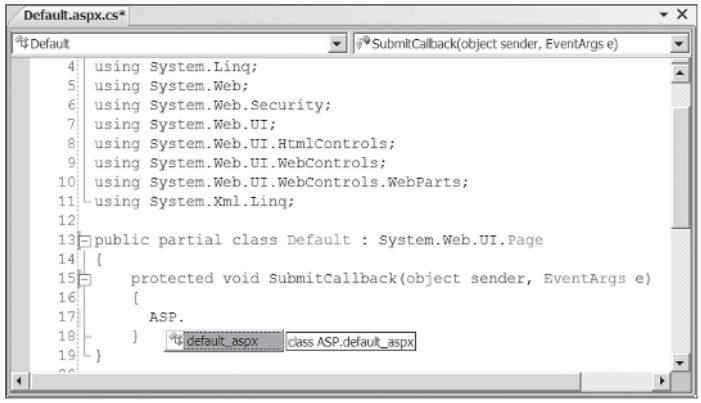

Figure 1-4: Temporary files for the application

As the tooltip shows, the ASP namespace contains a class named default_aspx. Note that the name of the class is the concatenation of the file name (default) and the file extension (aspx) separated by an underscore (_) character as discussed earlier.

As mentioned, the ASP.NET build environment temporarily stores the source code for this dynamically generated class in the Temporary ASP.NET Files directory. This directory contains one directory for each ASP.NET application that has ever run on the machine. The directory is named after the virtual root directory of its associated ASP.NET application. A couple of directories down this root directory, you can find the temporary files that contain the dynamically generated classes such as default_aspx. As mentioned, the ASP.NET build environment deletes these files immediately after the compilation process ends, unless you set the Debug attribute on the @Page directive of the ASP.NET page to true.

Next, take a look at the contents of the directory that contains the temporary files for the application. You'll start with hash.web file, which is located in the hash folder. This file contains the hash value for this directory. ASP.NET uses this hash value to generate hash values that are used as part of the names of the files contained in this directory. As you'll see later, these hash values ensure the uniqueness of the names of these files. Next, we'll discuss these four files: App_Web_jgyyqxek.0.cs, App_Web_jgyyqxek.1.cs, App_Web_jgyyqxek.2.cs, and App_Web_jgyyqxek.dll. Listing 1-22 will help you understand the role and significance of these four files.

Recall that Listing 1-20 contains the content of the Default.aspx file. Notice that this file registers an event handler named SubmitCallback for the Click event of the Submit button. The Default.aspx.cs code-behind file contains the implementation of this event handler as shown in Listing 1-22.

Listing 1-22: The code-behind file

```
using System;

public partial class Default : System.Web.UI.Page
{
  protected void SubmitCallback(object sender, EventArgs e)
  {
    Info.Text = "<b>Display Name: </b>" + DisplayNameTbx.Text + "<br/>";
    Info.Text += "<b>Email: </b>" + EmailTbx.Text;
  }
}
```

The App_Web_jgyyqxek.1.cs file contains the class defined in the code-behind file, that is, the Default class. Notice that this class references the server controls defined in the markup file, that is, Default.aspx. Also note that the class does not expose these server controls as protected fields as it used to do in Visual Studio 2003. Thanks to partial classes the addition of these tool-generated protected fields is no longer necessary when you're developing your pages. When you hit F5 to build and run the application, ASP.NET automatically generates another partial class exposing the required protected fields. The App_Web_jgyyqxek.0.cs file contains this partial class as shown in Listing 1-23.

Listing 1-23: The App_Web_jgyyqxek.0.cs file

```
public partial class Default : System.Web.SessionState.IRequiresSessionState
{
  protected global::System.Web.UI.WebControls.TextBox DisplayNameTbx;
  protected global::System.Web.UI.WebControls.TextBox EmailTbx;
  protected global::System.Web.UI.WebControls.Button SubmitBtn;
  protected global::System.Web.UI.WebControls.Label Info;
  protected global::System.Web.UI.HtmlControls.HtmlForm form1;
  ...
}
```

You may be wondering why the Default class shown in Listing 1-23 implements the IRequiresSessionState interface. What is this interface anyway? This interface is a marker interface; that is, it doesn't have any methods, properties, or events. Implementing this interface marks the implementor (the class that implements the interface) as a class that needs write access to the session data. The Default class shown in Listing 1-23 implements this interface when the value of the EnableSessionState attribute on the @Page directive is set to true (default).

When a compiler sees a partial class like the Default class shown in Listing 1-22 (App_Web_jgyyqxek.1.cs), it knows that the definition of this class is not complete and the class is missing some members. Therefore the compiler doesn't attempt to compile the class. Instead it first merges the partial Default classes shown in Listings 1-22 and 1-23 into a complete Default class as shown in Listing 1-24.

Listing 1-24: The complete Default class

```
public class Default : System.Web.UI.Page,
                       System.Web.SessionState.IRequiresSessionState
{
  protected global::System.Web.UI.WebControls.TextBox DisplayNameTbx;
  protected global::System.Web.UI.WebControls.TextBox EmailTbx;
  protected global::System.Web.UI.WebControls.Button SubmitBtn;
  protected global::System.Web.UI.WebControls.Label Info;
  protected global::System.Web.UI.HtmlControls.HtmlForm form1;
  ...

  protected void SubmitCallback(object sender, EventArgs e)
  {
    Info.Text = "<b>Display Name: </b>" + DisplayNameTbx.Text + "<br/>";
    Info.Text += "<b>Email: </b>" + EmailTbx.Text;
  }
}
```

Next, the ASP.NET build environment uses the PageBuildProvider to generate the source code for the class that represents the markup file, that is, the Default.aspx file shown in Listing 1-20. The App_Web_jgyyqxek.0.cs file contains the source code for this class as shown in Listing 1-21. Notice that this class inherits from the Default class shown in Listing 1-24.

Finally, Listing 1-25 presents the content of the App_Web_jgyyqxek.2.cs file. Notice that this file simply defines a fast object factory for the default_aspx type. The PageBuilderProvider calls the GenerateTypeFactory method of the AssemblyBuilder to generate this fast object factory class.

Listing 1-25: The type factory

```
namespace @__ASP
{
  internal class FastObjectFactory_app_web_jgyyqxek
  {
    static object Create_ASP_default_aspx()
    {
      return new ASP.default_aspx();
    }
  }
}
```

Notice that the directory whose content is shown on the right panel of the Windows Explorer shown in Figure 1-5 contains the following files that capture the information that the compiler generates:

- **jgyyqxek.cmdline.** This file contains the command line used to compile the code:

```
/t:library /utf8output
/R:"C:\Windows\assembly\GAC_32\System.EnterpriseServices\2.0.0.0__b03f5f7f11d50a3a\
System.EnterpriseServices.dll"
/R:"C:\Windows\assembly\GAC_MSIL\System.Core\3.5.0.0__b77a5c561934e089\
System.Core.dll"
...
/out:"C:\Windows\Microsoft.NET\Framework\v2.0.50727\Temporary ASP.NET Files\
website1\a49191ca\d589c0e2\App_Web_jgyyqxek.dll"
/D:DEBUG /debug+ /optimize- /w:4 /nowarn:1659;1699;1701 /warnaserror-
"C:\Windows\Microsoft.NET\Framework\v2.0.50727\Temporary ASP.NET Files\website1\
a49191ca\d589c0e2\App_Web_jgyyqxek.0.cs"
"C:\Windows\Microsoft.NET\Framework\v2.0.50727\Temporary ASP.NET Files\website1\
a49191ca\d589c0e2\App_Web_jgyyqxek.1.cs"
"C:\Windows\Microsoft.NET\Framework\v2.0.50727\Temporary ASP.NET Files\website1\
a49191ca\d589c0e2\App_Web_jgyyqxek.2.cs"
```

  As the boldfaced portions of this listing shows, this command line compiles the App_Web_jgyyqxek.0.cs, App_Web_jgyyqxek.1.cs, and App_Web_jgyyqxek.2.cs files into an assembly named App_Web_jgyyqxek.dll.

- **gyyqxek.out.** This file contains any text that the compiler generates while it's compiling.
- **jgyyqxek.err.** This file contains any error text that the compiler generates.

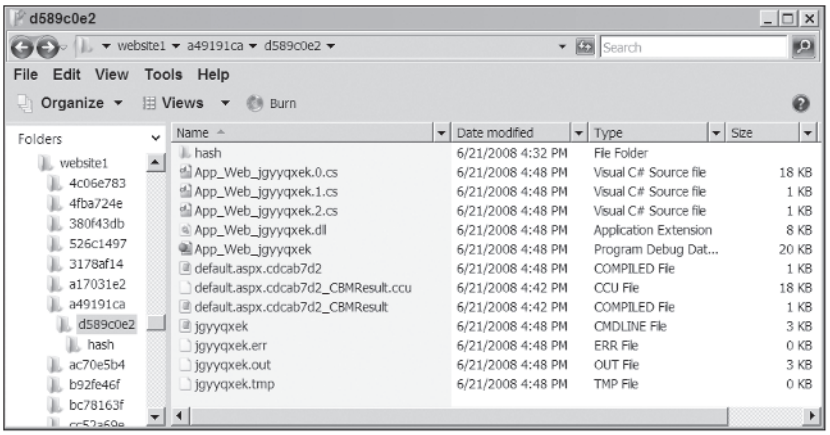

Figure 1-5: The dynamically generated files for an ASP.NET application

There is one more important file in the right panel of the Windows Explorer; Listing 1-26 shows the content of this file.

Listing 1-26: The default.aspx.cdcab7d2 compiled file

```
<?xml version="1.0" encoding="utf-8"?>
<preserve resultType="3" virtualPath="/WebSite1/Default.aspx"hash="fffffff8cb1ce302"
filehash="28511aded9d26a0a" flags="110000" assembly="App_Web_jgyyqxek"
type="ASP.default_aspx">
  <filedeps>
    <filedep name="/WebSite1/Default.aspx" />
    <filedep name="/WebSite1/Default.aspx.cs" />
  </filedeps>
</preserve>
```

As discussed earlier, the ASP.NET build environment compiles the App_Web_jgyyqxek.0.cs, App_Web_jgyyqxek.1.cs, and App_Web_jgyyqxek.2.cs files (the default_aspx class) into App_Web_jgyyqxek.dll (which is stored in the ASP.NET Temporary Files directory), loads this compiled assembly into the application domain where the current application is running, instantiates the default_aspx class, and calls its ProcessRequest method to process the current request.

Now let's see what happens when the next request for the same page (Default.aspx) arrives. ASP.NET first reads the content of the default.aspx.cdcab7d2 file shown in Listing 1-26. Notice that this file is an XML file with a document element named <preserve> that features the following important attributes:

- **virtualPath.** The virtual path of the Default.aspx file
- **type.** The fully qualified name of the type or class that represents the Default.aspx file, that is, the ASP.default_aspx class
- **assembly.** The name of the assembly that contains the ASP.default_aspx class

ASP.NET retrieves two important pieces of information from the default.aspx.cdcab7d2 file: the name of the class that represents the Default.aspx file and the name of the assembly that contains this file. ASP.NET then simply locates this assembly in the application domain and calls the type factory shown in Listing 1-25 to create an instance of the ASP.default_aspx class and finally calls the ProcessRequest method of this instance to process the request. In other words, processing the second request does not go through the code generation and compilation steps that the first request went through. Therefore, the second request will not experience the code generation and compilation delay that the first request experienced. As you'll see later, SharePoint application pages and page templates for ghosted site pages are processed through this same exact ASP.NET compilation model. In other words, these pages are treated just like any other normal ASP.NET pages. That is why SharePoint application pages and ghosted site pages perform much better than unghosted site pages.

Notice that the <preserve> document element features a single child element named <filedeps>, which contains one or more <filedep> elements where each <filedep> element features an attribute that specifies the virtual path of the file that the ASP.default_aspx class depends on. When one of these files changes, ASP.NET checks whether the assembly that contains the ASP.default_aspx class contains other classes. If so, it doesn't delete the assembly. Instead it uses the same code generation and compilation steps discussed in this chapter to compile a new assembly and loads this assembly into the application domain where the current application is running. This means that now both the old assembly and the new assembly are running side-by-side inside the application domain. This is possible because the assembly name contains a randomly generated hash value that ensures the uniqueness of the assembly names. In other words, each time ASP.NET compiles a new assembly, it gives it a new name.

If the old assembly doesn't contain any other classes, ASP.NET attempts to remove it from the ASP.NET Temporary Files directory. If another request is using this assembly, the assembly cannot be deleted because it's locked. If that is the case, ASP.NET simply renames the assembly to App_Web_jgyyqxek.dll .delete. The assemblies that are marked as deleted are deleted when the application restarts. Even though ASP.NET may not be able to remove the old assembly, it can still go ahead with building the new assembly and loading it into the application domain, which means that now we have two different versions of the same assembly running side-by-side inside the application domain.

This raises the following question: What happens to the old assembly when the request using it ends? Unfortunately, you cannot unload an assembly from an application domain, which means that the old assembly will remain in the application domain until the application domain shuts down even though no one is using this assembly. This is because the application domain is the CLR unloading unit. You have to unload the entire application domain in order to unload an assembly. Unloading an application domain unloads all the assemblies loaded into the application domain.

Therefore, every time you make a little change in the Default.aspx or Default.aspx.cs file, ASP.NET loads a new assembly into the application domain without unloading the old ones. In other words, after several recompiles, you end up littering the application domain with a bunch of useless assemblies. That is why ASP.NET puts an upper limit on the number of allowable recompiles. When an application domain reaches this limit, ASP.NET automatically unloads the application domain, which automatically unloads all the assemblies loaded into the application domain.

Keeping unused old assemblies in web server memory causes major problems for SharePoint web servers that host thousands of sites, each with numerous site pages. As an example let's consider the home pages of these sites. The home page of a site is a site page provisioned from a page template named default.aspx. When a site page is provisioned from a page template it remains in ghosted state until it is customized. When the first request for the home page of a site arrives, if this is the first request made to any instance of the default.aspx page template, that is, if the home page of no other sites has been requested yet, the SPVirtualPathProvider simply loads the default.aspx page template from the file system of the front-end web server and passes it along to the ASP.NET page parser for the standard ASP.NET compilation processing as just thoroughly discussed where:

- The default.aspx page template is parsed into a dynamically generated class named ASP.default_aspx
- The ASP.default_aspx class is dynamically compiled into an assembly, which is stored in the ASP.NET Temporary Files folder on the file system of the front-end web server
- An instance of the ASP.default_aspx class is dynamically instantiated and assigned to the task of processing the request

The next request for the home page of the same site or any other sites in the same SharePoint application will be directly served from the same assembly where a new instance of the ASP.default_aspx class is instantiated and assigned to the task of processing the request. In other words, the first two steps of the previous three steps are not repeated for the next request to the home pages of the same or other sites of the same web applications. As you can see, this is a great boost in performance for these requests.

Now imagine the case where the site administrators of these sites, which could be thousands of them, customize the home pages of these sites in the SharePoint Designer. When a site administrator saves his or her changes, the SharePoint Designer saves the content of the home page in the content database. This means that each site now has its own version of the default.aspx page template stored in the content

database. When a request for the home page of one of these sites arrives, the SPVirtualPathProvider loads the associated version of the default.aspx page template from the content database and passes it along to ASP.NET parser for the standard ASP.NET compilation processing discussed earlier.

This means that now each version of the default.aspx page template, that is, each site page, which could run into thousands, is compiled into a separate assembly. As you can see, we end up having literally thousands of assemblies in the web server memory and consequently the application domain reaches it upper limit of assemblies and unloads. As you can imagine, this is simply not scalable in large SharePoint web applications. You may be wondering why not simply unload the assembly right after the request for its associated site page is processed. As mentioned earlier .NET does not allow unloading individual assemblies from an application domain. When an assembly is loaded into memory it must remain in memory until the application domain is unloaded.

That is why unghosted site pages do not go through the standard ASP.NET compilation process, which means that they are not compiled into assemblies. Instead they are simply parsed and interpreted on the fly. This is known as no-compile mode and these pages are known as no-compile pages. When a request for an unghosted site page arrives and the SPVirtualPathProvider determines that the requested resource is an unghosted site page, SPVirtualPathProvider downloads the unghosted site page from the content database into the web server memory and passes it along to the ASP.NET page parser as usual. However, SPVirtualPathProvider also instructs the ASP.NET page parser to process the unghosted site page in no-compile mode. This allows SharePoint to unload the page from memory after the request is processed to release precious web server resources for next requests. This is obviously a much more scalable solution than compiled pages.

This, however, introduces a restriction on site pages. Because site pages are not compiled into assemblies, they cannot contain inline code. There is also a security aspect involved here. Because site pages are stored in the content database, by default, they are processed in safe mode where no inline code is allowed, and where the site pages can only contain server controls that are registered as safe controls. These security measures are in place to ensure that no one can mount an attack on the content database and consequently an attack on the web server by injecting malicious inline code or server controls into a site page.

When you implement a custom server control that you want to be used in site pages, you must add an entry into the web.config file of the SharePoint web application to register your server control as a safe control before anyone can use it in a site page. This entry is an element named <SafeControl>, which exposes the following attributes:

- **Assembly.** This is the complete information about the assembly that contains the server controls being registered as safe, including assembly name, version, culture, and public key token.
- **Namespace.** This is the complete namespace containment hierarchy of the server controls being registered as safe.
- **Type.** This is the type of the server control being registered as safe. Set this attribute to * to register all server controls in the specified namespace in the specified assembly as safe.
- **Safe.** Set this attribute to true to register the specified server controls as safe.
- **AllowRemoteDesigner.** Set this attribute to true to allow the remote designer.

When you provision a SharePoint web application, SharePoint automatically registers all standard ASP.NET and SharePoint server controls as safe. The following excerpt for the web.config file of a typical SharePoint web application shows a few of these SafeControl entries:

```
<configuration>
  <SharePoint>
    <SafeControls>
      <SafeControl Assembly="System.Web, Version=1.0.5000.0, Culture=neutral,
                             PublicKeyToken=b03f5f7f11d50a3a"
      Namespace="System.Web.UI.WebControls" TypeName="*" Safe="True"
      AllowRemoteDesigner="True" />

      <SafeControl Assembly="System.web, Version=1.0.5000.0, Culture=neutral,
                           PublicKeyToken=b03f5f7f11d50a3a"
      Namespace="System.Web.UI.HtmlControls" TypeName="*" Safe="True"
      AllowRemoteDesigner="True" />

      <SafeControl Assembly="System.Web, Version=2.0.0.0, Culture=neutral,
                             PublicKeyToken=b03f5f7f11d50a3a"
      Namespace="System.Web.UI" TypeName="*" Safe="True"
      AllowRemoteDesigner="True" />

     <SafeControl Assembly="Microsoft.SharePoint, Version=12.0.0.0,
                           Culture=neutral, PublicKeyToken=71e9bce111e9429c"
     Namespace="Microsoft.SharePoint.WebControls" TypeName="*" Safe="True"
     AllowRemoteDesigner="True" />
    </SafeControls>
  </SharePoint>
</configuration>
```

Keep in mind that when you provision a site page from a page template, the site page remains in ghosted state until it is customized. As discussed earlier, requests for ghosted site pages are processed through a normal ASP.NET compiled page scenario where the SharePoint safe mode is not involved. This means that the page template could in principle contain inline code and unsafe server controls. However, as soon as someone customizes the site page, SharePoint will process the next requests through the safe mode. Therefore the next requests will get an exception if the page template contains inline code and/or unsafe server controls. To avoid this problem, you should never contain inline code and/or unsafe server controls in your page templates.

Summary

This chapter provided an in-depth coverage of the SharePoint 2007 architecture and its main components. You learned how SharePoint extends ASP.NET and IIS to add support for SharePoint-specific functionality and features. The chapter discussed the main ASP.NET and IIS components and the SharePoint extensions to these components. You also learned a great deal about the ASP.NET HTTP Runtime Pipeline and the ASP.NET dynamic compilation and the role they play in SharePoint.

The next chapter moves on to the Collaborative Application Markup Language (CAML) and shows you how to use these power markup languages to query SharePoint data and to implement various SharePoint components. You'll also learn a great deal about custom actions, features, and application pages.