What's in ASP.NET Config Files?

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifContents

[Machine.config](http://msdn.microsoft.com/en-us/magazine/cc163928.aspx#S1)  
[Web.config](http://msdn.microsoft.com/en-us/magazine/cc163928.aspx#S2)  
[Changing Settings](http://msdn.microsoft.com/en-us/magazine/cc163928.aspx#S3)  
[Using the <appSettings> Section](http://msdn.microsoft.com/en-us/magazine/cc163928.aspx#S4)  
[Using NameValueFileSectionHandler](http://msdn.microsoft.com/en-us/magazine/cc163928.aspx#S5)  
[Custom Configuration Sections](http://msdn.microsoft.com/en-us/magazine/cc163928.aspx#S6)  
[Conclusion](http://msdn.microsoft.com/en-us/magazine/cc163928.aspx#S7)

Even though you've been using ASP.NET for a while, how much do you really know about ASP.NET configuration files? While you've probably touched the Web.config file from time to time, there are some nuances involved in configuring ASP.NET that you may not have noticed. This month I'll discuss the architecture behind ASP.NET configuration files and show how changes to the configuration affect running programs. I'll also cover several ways to add your own custom configuration settings, including how to implement a separate assembly for handling custom configuration settings for your specific application.

**Machine.config**

When you install the Microsoft® .NET Framework, an XML file named Machine.config is installed into the \Windows\Microsoft.Net\Framework\*xxx*\config directory (where *xxx* is 1.0.3705 for version 1.0 of the Framework, or 1.1.4322 for version 1.1). Machine.config holds the default settings for your installation and sets the parameters for how .NET runs on the entire machine.

When browsing through Machine.config, you'll find nodes for all kinds of machine settings. You'll see <system.net>, <system.diagnostic>, and <system.runtime.remoting>, among others. As you might guess, <system.diagnostic> includes settings for managing the diagnostic behavior of your installation—how trace messages and listeners are managed on your machine. The <system.runtime.remoting> section group includes settings for managing how remoting behaves on your machine. These settings include items such as what components are used to manage the channels.

The section group we want to look at is <system.web>, which includes the settings that tell your system just how ASP.NET should run on your machine. The <system.web> node contains further sections and section groups. Near the top of Web.config you'll find the section groupings within <system.web>. For example, **Figure 1** shows the sections that define browser capabilities, client targets, compilation options, and page processing settings. Notice these are not the actual configuration settings (we'll get to those later). These elements simply map a section within the configuration file to a specific assembly and type that parses the configuration settings and interprets them at run time.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFigure 1 system.web Section Group

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl22_ctl00_ctl00_code');" \o "Copy Code)

<sectionGroup name="system.web">

<section name="browserCaps" type=

"System.Web.Configuration.HttpCapabilitiesSectionHandler,

System.Web, Version=1.0.5000.0, Culture=neutral,

PublicKeyToken=b03f5f7f11d50a3a"/>

<section name="clientTarget" type=

"System.Web.Configuration.ClientTargetSectionHandler,

System.Web, Version=1.0.5000.0, Culture=neutral,

PublicKeyToken=b03f5f7f11d50a3a"/>

<section name="compilation"

type="System.Web.UI.CompilationConfigurationHandler,

System.Web, Version=1.0.5000.0, Culture=neutral,

PublicKeyToken=b03f5f7f11d50a3a"/>

<section name="pages"

type="System.Web.UI.PagesConfigurationHandler,

System.Web, Version=1.0.5000.0, Culture=neutral,

PublicKeyToken=b03f5f7f11d50a3a"/>

•••

The <system.web> settings appear a few pages lower in the file. There you'll find all the various ways you may configure items such as session state, authentication, page compilation, and debug settings. For example, the <sessionState> element includes values for defining how session state is managed (disabled, stored in the current process's memory, stored in SQL Server™, or stored in an ASP.NET state server on a specific machine). Here are the default settings for session state defined in a fresh .NET installation:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl05_code');" \o "Copy Code)

<sessionState mode="InProc"

stateConnectionString="tcpip=127.0.0.1:42424"

stateNetworkTimeout="10" sqlConnectionString=

"data source=127.0.0.1;Integrated Security=SSPI"

cookieless="false" timeout="20"/>

Go back up to the top of Machine.config and you'll see the following in the <system.web> section group:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl06_code');" \o "Copy Code)

<section name="sessionState"

type=

"System.Web.SessionState.SessionStateSectionHandler,

System.Web, Version=1.0.5000.0, Culture=neutral,

PublicKeyToken=b03f5f7f11d50a3a"

allowDefinition="MachineToApplication" />

This line in Machine.config tells ASP.NET to use a class named SessionStateSectionHandler in the System.Web.dll assembly to manage state. When ASP.NET loads, a class needs to interpret the session state element, and SessionStateSectionHandler is the class that will do that.

As you can see from the code, Machine.config specifies that session state should be managed in-process. For most single-server applications or for Web farms that maintain machine affinity when using load balancing, "InProc" session state works just fine. However, for a large Web farm you probably want to store session state elsewhere (for example, on a machine running SQL Server). If you change the session state settings to use SQL Server in Machine.config, then all the applications on that one box will use SQL Server. Of course, you probably want to control session state on a per-application basis. That's why individual ASP.NET applications include a Web.config file.

**Web.config**

Web.config files modify the default settings stipulated in Machine.config. If you do a search for Web.config, you'll find quite a few on your machine. There may be a Web.config file in \inetpub\wwwroot that controls <system.web> settings for the entire site. Then there's a Web.config file in your application's virtual directory. In addition, each subdirectory below an application's virtual directory may also have a Web.config file.

Machine.config defines the default settings for the machine. After that, ASP.NET looks to the Web.config in the \inetpub\wwwroot directory for settings. Finally, ASP.NET looks to the Web.config files in the application virtual directory and its subdirectories to change settings. By the time ASP.NET gets to the settings stipulated by a leaf subdirectory of your application, it's conceivable (though unlikely) the Web.config file may have completely changed the default settings.

The following shows how settings within the Web.config files are applied to an application. ASP.NET reads configuration files and makes the changes in this order:

* Machine.config (sets the defaults for the machine)
* \inetpub\wwwroot\Web.config (modifies the settings for the main site)
* *appvirtualdir*\Web.config (modifies the settings for this application)
* *appvirtualdir*\*subdir*\Web.config (modifies the settings for this subdirectory)

The settings are modified in a hierarchical manner, with each subdirectory getting a chance to modify the previously stipulated settings found in each Web.config file.

If you don't feel like littering your application subdirectories with Web.config files, the <location> element lets you specify path-specific configuration settings within the main Web.config file. To use the <location> element, place the element in your Web.config file and specify the path to which the settings apply. The following shows an example:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl07_code');" \o "Copy Code)

<?xml version="1.0" ?>

<configuration>

<location path="SubDir">

<system.web>

<!-- settings for "SubDir" go here... -->

</system.web>

</location>

</configuration>

Of course, there may be some settings that you'd like to maintain without being overridden. The <location> element includes an allowOverride attribute that prevents overriding when set to false.

**Changing Settings**

ASP.NET reads these configuration files when the application starts. The settings are held in memory, ensuring high performance. ASP.NET also watches the configuration files. When you change settings and save a configuration file, ASP.NET automatically restarts the application and reloads the updated configuration into memory. That means the application domain hosting the application within ASPNET\_WP.EXE will be torn down and recreated. You'll lose any in-process state when saving the configuration file (such as session state that has not been flushed to a state server).

While there are quite a number of default settings within ASP.NET, it's often useful to create your own configuration settings. For example, it's a bad practice to hardcode sensitive and malleable data, such as database connection strings, directly in your application. What happens when a database administrator changes the connection string (for example, the login password)? You'll get a phone call because the application breaks—usually at home in the middle of the night or on a weekend! And then you need to change your source code, recompile, and redeploy the application.

So, rather than hardcoding, it makes sense to use a configuration file. That way, the system administrators are called in the middle of the night to change the configuration (and you can stay in bed). Let's take a look at some of the ways to store and retrieve settings using Web.config.

**Using the <appSettings> Section**

If you have only a simple value you want to store in Web.config, the most convenient place to do so is in the <appSettings> node. The <appSettings> node falls within the <configuration> element, but outside of the <system.web> element, as shown in **Figure 2**. Notice that the <appSettings> syntax includes the verb "add" that instructs the NameValueFileSectionHandler component to add key/value pairs to a collection. **Figure 2** also shows how to retrieve the keys during the Page\_Load event as well as how to retrieve the actual values in response to a button click.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFigure 2 Retrieving the Application Settings

Web.config

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl26_ctl00_ctl00_code');" \o "Copy Code)

<configuration>

<appSettings>

<add key="ConnectionString" value="Value for ConnectionString"/>

<add key="Other appSetting" value="Value for Other appSetting"/>

</appSettings>

<!-- other settings-->

<system.web>

<!--system Web settings go here... -->

</system.web>

</configuration>

Form1.aspx.cs

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl26_ctl00_ctl01_code');" \o "Copy Code)

private void Page\_Load(object sender, System.EventArgs e)

{

if(!this.IsPostBack)

{

DropDownListappSettingsKeys.DataSource =

ConfigurationSettings.AppSettings.Keys;

DropDownListappSettingsKeys.DataBind();

}

}

private void ButtonGetValueOfAppSettingKey\_Click(

object sender, System.EventArgs e)

{

string strKey = DropDownListappSettingsKeys.SelectedItem.Text;

string strValueOfKey = ConfigurationSettings.AppSettings[strKey];

LabelValueOfappSettingsKey.Text = strValueOfKey;

}

**Using NameValueFileSectionHandler**

Most configuration settings tend to be name/value pairs (as with the example just shown). The <appSettings> section is already built in, and you can easily access configuration parameters at run time. However, there are instances in which you may want to isolate a set of configuration settings or otherwise define them separately. For example, the applications in your domain may have some special configuration information you want to include in a separately named group (that is, not in appSettings).

The easiest way to do this is to set up your own named section, but use the existing NameValueSectionHandler component to parse it and build a key/value collection that you may access at run time. **Figure 3** shows how to define a separate configuration group and tell ASP.NET to use the NameValueSectionHandler to parse it. Notice how the Web.config for the application adds a <sectionGroup> and a named <section> to the <configSections> element and tells ASP.NET to use the NameValueSectionHandler component to parse it. System.dll, which contains the NameValueSectionHandler type, is a strongly named system assembly, so you need to use the assembly's full name in order to specify the handler.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFigure 3 Defining a Separate Configuration Group

Web.config

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl27_ctl00_ctl00_code');" \o "Copy Code)

<configuration>

<configSections>

<sectionGroup name="system.web">

<section name="customSection"

type="System.Configuration.NameValueSectionHandler,

System, Version=1.0.5000.0, Culture=neutral,

PublicKeyToken=b77a5c561934e089"/>

</configSections>

<customSection>

<add key="customSection Setting 1"

value="Value for customSection Setting 1"/>

<add key=" customSection Setting 2"

value="Value for customSection Setting 2"/>

</customSection>

</configuration>

Form1.aspx.cs

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl27_ctl00_ctl01_code');" \o "Copy Code)

private void Page\_Load(object sender, System.EventArgs e)

{

if(!this.IsPostBack)

{

NameValueCollection nvccustomSection;

nvccustomSection = (NameValueCollection)

ConfigurationSettings.GetConfig("customSection");

DropDownListcustomSectionKeys.DataSource = nvccustomSection.Keys;

DropDownListcustomSectionKeys.DataBind();

}

}

private void ButtonGetValueOfcustomSectionKey\_Click(

object sender, System.EventArgs e)

{

string strKey = DropDownListcustomSectionKeys.SelectedItem.Text;

NameValueCollection nvccustomSection;

nvccustomSection = (NameValueCollection)

ConfigurationSettings.GetConfig("customSection");

string strValueOfKey = nvccustomSection[strKey];

LabelValueOfcustomSectionKey.Text = strValueOfKey;

}

**Custom Configuration Sections**

If you decide that simple name/value pairs aren't quite good enough for the configuration settings of your application, you may write your own configuration section handler component. For example, if you want to use your own XML schema in a section of the configuration file, you'll need access to the XML directly from the configuration file. Interpreting the XML directly gives you a means of checking the syntax of the configuration information and perhaps throwing an exception if there's a problem.

Writing a component to read sections out of the configuration file is fairly straightforward. You simply write a class deriving from IConfigurationSectionHandler and then reference it in your own Web.config file. The IConfigurationSectionHandler interface is shown in the following lines of code:

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl14_code');" \o "Copy Code)

public interface IConfigurationSectionHandler

{

object Create(object parent, object input, XmlNode node);

}

IConfigurationSectionHandler contains only one method: Create. The sole purpose of Create is to parse the configuration section. Create takes three parameters: two objects (parent and input) and an XmlNode. The first parameter, parent, represents the configuration settings in a corresponding parent configuration section—that is, the configuration information that's been read in already. The second object, input, represents an HttpConfigurationContext. There's not much to HttpConfigurationContext—just the virtual path to the Web.config configuration file. Conceivably, you could use the path to open the Web.config file directly. However, the third parameter, called node, represents the XmlNode containing the configuration information from the configuration file. This is how you can have direct access to the XML contents of the configuration section.

**Figure 4** shows an implementation of IConfigurationSectionHandler used in this month's sample. The custom configuration handler configures a logging component for recording requests to the application. The CustomConfigSettings class contains the variables (a string to store a log file name and a Boolean to store whether or not logging should be done). The CustomConfigHandler class is used by the ASP.NET infrastructure to parse the custom configuration section and provide the setting to the rest of the app.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFigure 4 Custom Configuration Parser

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl29_ctl00_ctl00_code');" \o "Copy Code)

public class CustomConfigSettings

{

private string \_strlogfilename;

private bool \_bLogRequests;

public CustomConfigSettings(string strlogfilename, bool bLogRequests)

{

\_strlogfilename = strlogfilename;

\_bLogRequests = bLogRequests;

}

public string LogFileName

{

get { return \_strlogfilename; }

}

public bool LogRequests

{

get { return \_bLogRequests; }

}

}

public class CustomConfigHandler : IConfigurationSectionHandler

{

public virtual object Create(object parent, object input, XmlNode node)

{

XmlNode n = node.Attributes.RemoveNamedItem("LogRequests");

if(n == null)

{

throw new ConfigurationException("Attribute expected: LogRequests");

}

bool bLogRequests;

if(string.Compare(n.Value, "TRUE", true) == 0)

{

bLogRequests = true;

// Set up the request logger here...

}

else bLogRequests = false;

n = node.Attributes.RemoveNamedItem("LogFileName");

if(n == null)

{

throw new ConfigurationException("Attribute expected: LogFileName");

}

return new CustomConfigSettings(n.Value, bLogRequests);

}

}

When parsing the configuration file, the CustomConfigurationHandler class expects to see an element containing two attributes: LogFileName and a LogRequests, the latter representing whether or not to do the logging. **Figure 5** shows how the configuration section handler is referenced in Web.config. Notice that Web.config defines a new section, named <customConfigHandling>, and specifies the CustomConfigHandler as the class to parse the configuration information for that section. Then the <system.web> node includes an element for logging configuration.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifFigure 5 Custom Configuration Handler

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl30_ctl00_ctl00_code');" \o "Copy Code)

<configuration>

<configSections>

<sectionGroup name="system.web">

<section name="customConfigHandling"

type="CustomConfigHandling.CustomConfigHandler,

CustomConfigHandlerLib"/>

</sectionGroup>

</configSections>

<system.web>

<customConfigHandling LogRequests="true"

LogFileName="c:\requestlog.txt"/>

</system.web>

CustomConfigHandler.Create is called by the system to parse the configuration information. ASP.NET hands the configuration information that has already been gathered (the first parameter) and an XmlNode representing the element that is to be parsed to the Create method. Create then pulls out the LogFileName and LogFile flag and returns them in an instance of the CustomConfigSettings class.

As with all other configuration information, you can pick up the custom configuration settings at run time. The following code shows how to access the custom configuration information during a Page\_Load event.

[Copy Code](javascript:CopyCode('ctl00_MTContentSelector1_mainContentContainer_ctl19_code');" \o "Copy Code)

private void Page\_Load(object sender, System.EventArgs e)

{

CustomConfigSettings customConfigSettings;

customConfigSettings = (CustomConfigSettings)

ConfigurationSettings.GetConfig("system.web/customConfigHandling");

if(customConfigSettings != null)

{

LabelLogFileName.Text = customConfigSettings.LogFileName;

LabelLogRequests.Text = customConfigSettings.LogRequests.ToString();

}

}

Note that the object returned by GetConfig is an instance of CustomConfigSettings.

**Conclusion**

This month I looked at how ASP.NET manages configuration settings. Machine.config includes all the default settings for your entire machine. You saw that you can touch Machine.config to modify the default settings for ASP.NET, but doing so affects the entire machine. Therefore ASP.NET applications look to a local Web.config file for defining settings specific to the application. While there are a number of standard system settings already defined, you probably need to define your own configuration settings once in a while.

ASP.NET offers several ways to add your own configuration settings and even lets you write your own configuration section handler. While I have only scratched the surface, you'll see as you delve deeper that the configuration system is extensible and flexible enough to do pretty much anything you'd like it to do.