Why, Where, and How of .NET Configuration Files

Eric Lynch
12 Mar 2021 CPOL

This article provides a quick overview of .NET configuration files and links to more in-depth information.

Update: *The world has moved on. Much of the information in this article is now dated. The preferred mechanism for configuration is now in the* `Microsoft.Extensions.Configuration` *NuGet package and related packages. For more details, see the documentation for the* `ConfigurationBuilder` *class found here: https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.configuration.configurationbuilder.*

Introduction

I've been working with .NET configuration files for a number of years now. I thought it might be helpful to others if I provided a quick introduction to them.

Within this document, many of the C# code examples assume you have both included a reference to System.Configuration.dll in your project and that you have included the following using statement in your code:

```
using System.Configuration;
```

Both of these are necessary to access the `ConfigurationManager` class, which is one method of accessing configuration information.

Why

The .NET framework provides a rich set of classes, and techniques, to simplify application configuration. Essentially, all of these classes make it easy to read and write that configuration information from an XML configuration file.

The configuration file includes a number of standard sections, some custom sections for common .NET features, and also allows the developer to create their own custom configuration sections.

The standard sections have evolved over time. Initially, standard configuration was done mostly through the `appSettings` section, which contains name / value pairs for each setting. Over time, transparent, type-safe support was provided via a generated C# `Settings` class and the corresponding `applicationSettings` and `userSettings` configuration sections.

Where

Where do I find the configuration file? This is a deceptively complicated problem. Since configuration is hierarchical, there are actually multiple configuration files that may affect an application. These include the machine configuration file, the application (or web) configuration file, the user local settings file, and the user roaming settings file.

Machine Configuration

The machine configuration file lives with the, not so easily found, .NET framework files. The location of the configuration file is dependent on the version of .NET and type of platform (e.g. 64 bit) used by your application.

A typical example, might be: C:\Windows\Microsoft.NET\Framework\v4.0.30319\CONFIG\machine.config

In your C# application code, the following will return the location of the file:

```
System.Runtime.InteropServices.RuntimeEnvironment.GetRuntimeDirectory() +
@"CONFIG\machine.config"
```

Application Configuration

The application configuration file usually lives in the same directory as your application. For web applications, it is named Web.config. For non-web applications, it starts life with the name of App.config. Following a build, it is copied to the same name as your .exe file. So, for the program MyProgram.exe, you might expect to find MyProgram.exe.config, in the same directory.

In your C# application code, the following will return the location of the file:

```
AppDomain.CurrentDomain.SetupInformation.ConfigurationFile
```

While it is not generally recommended, you may find that some applications alter the location of the application configuration file as follows:

```
AppDomain.CurrentDomain.SetData("APP_CONFIG_FILE", "NewName.config")
```

User Settings

The user settings are almost impossible to find, perhaps by design. The names of the directories vary with versions of Windows. To complicate matters further, the parent folder is generally hidden. The folder structure also incorporates the company name (of the application vendor), the application name, a unique identity for the application, and the application version.

An example, on Windows 7, for local user settings might look like:

C:\Users\MyUsername\AppData\Local\CompanyName\MyProgram.exe_Url_pnbmzrpiumd43n0cw05z2h4o23fdxzkn\1.0.0.0\user.config

In C#, you can get the base directory for local user settings (first line) or roaming user settings (second line) as follows:

```
Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData)
Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData)
```

In C# (see notes in Overview), you can get the exact file path for local user settings (first line) or roaming user settings (second line) as follows:

```
ConfigurationManager.OpenExeConfiguration(ConfigurationUserLevel.PerUserRoamingAndLocal).FilePa
th
ConfigurationManager.OpenExeConfiguration(ConfigurationUserLevel.PerUserRoaming).FilePath
```

Other Configuration

If this isn't all confusing enough, you should be aware of a few other files. There is a root Web.config (located in the same directory as machine.config). Also, sub-directories of a web application may provide additional overrides of inherited settings, via a Web.config specific to that sub-directory.

Lastly, IIS provides some of its own configuration. A typical location would be: C:\Windows\System32\inetsrv\ApplicationHost.config

How

As mentioned earlier, the application configuration file is broken into a number of fairly standard configuration sections. Here, we briefly discuss a few of the most common sections.

appSettings Section

The simplest of the standard configuration sections is `appSettings`, which contains a collection of name / value pairs for each of the settings:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="MySetting" value="MySettingValue" />
  </appSettings>
</configuration>
```

In C# (see notes in Overview), you can reference the value of a setting as follows:

```
string mySetting = ConfigurationManager.AppSettings["MySetting"];
```

connectionStrings Section

Since database connections are so common in .NET, a special section is provided for database connection strings. The section is called `connectionStrings`:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <add name="MyConnectionStringName"
      connectionString="Data Source=localhost;Initial Catalog=MyDatabase;Integrated
Security=True"
      providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

In C#, you can get the connection string as follows:

```
string connectionString = ConfigurationManager.ConnectionStrings[
  "MyConnectionStringName"].ConnectionString;
```

Initially, one might wonder at the need to reference a `ConnectionString` property of a "connection string". In truth, the connectionStrings section is poorly named. A better name might have been `connectionStringSettings`, since each entry contains both a connection string and a database provider.

The syntax of a connection string is wholly determined by the database provider. In this case `System.Data.SqlClient`, is the most common database provider for the Microsoft SQL Server database.

applicationSettings and userSettings Section

With .NET 2.0, Microsoft tried to make it even easier to use configuration files. They introduced a settings file. A careful observer will note that the "settings" start their life in the application configuration file and, later, get copied to the user settings configuration file.

With Windows Form and WPF applications, you'll find a file Settings.settings in the Properties folder of your project. For Console applications, and others, you can also take advantage of settings. Open the Properties for your project, and click on the Settings button/tab. You'll be offered the option of adding a default settings file.

Typically, you edit this settings file (or the settings for your project) rather than editing the configuration file directly. The example below is provided only to demonstrate that the settings do actually live in the configuration file.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
```

```
  <configSections>
    <sectionGroup name="userSettings"
      type="System.Configuration.UserSettingsGroup, System, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089">
      <section name="WinFormConfigTest.Properties.Settings"
        type="System.Configuration.ClientSettingsSection, System, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089"
        allowExeDefinition="MachineToLocalUser"
        requirePermission="false" />
    </sectionGroup>
    <sectionGroup name="applicationSettings"
       type="System.Configuration.ApplicationSettingsGroup, System, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089">
      <section name="WinFormConfigTest.Properties.Settings"
        type="System.Configuration.ClientSettingsSection, System, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089"
        requirePermission="false" />
    </sectionGroup>
  </configSections>
  <userSettings>
    <WinFormConfigTest.Properties.Settings>
      <setting name="MyUserSetting" serializeAs="String">
        <value>MyUserSettingValue</value>
      </setting>
    </WinFormConfigTest.Properties.Settings>
  </userSettings>
  <applicationSettings>
    <WinFormConfigTest.Properties.Settings>
      <setting name="MyApplicationSetting" serializeAs="String">
        <value>MyApplicationSettingValue</value>
      </setting>
    </WinFormConfigTest.Properties.Settings>
  </applicationSettings>
</configuration>
```

To reference one of the settings, you simply use the `Settings` class, which is automatically created for you. A typical reference, might look as follows:

```
string myUserSetting = Properties.Settings.Default.MyUserSetting;
string myApplicationSetting = Properties.Settings.Default.MyApplicationSetting;
```

Note: `Properties` is a namespace that is automatically created in your application's name space.

To change a user's settings, you simply assign a value to the property and save the changes, as follows:

```
Properties.Settings.Default.MyUserSetting = newValueForMyUserSetting;
Properties.Settings.Default.Save();
```

Upgrading Settings

Eventually, you'll want to release a new version of your application. Here, you may encounter a common problem. Since the user settings are version specific, they will be lost following the upgrade.

Thankfully, the framework anticipates this requirement and provides the `Upgrade` method. A typical way of handling this is to include a boolean `Upgraded` user setting, with an initial value of false (when your application is first deployed).

So, typical code to handle the upgrade (and retain previous user settings) looks as follows:

```
if (!Properties.Settings.Default.Upgraded)
{
  Properties.Settings.Default.Upgrade();
  Properties.Settings.Default.Upgraded = true;
  Properties.Settings.Default.Save();
}
```

What Next

In the previous section, you may have noticed the rather verbose `configSections` section of the configuration file. This is how Microsoft extended the configuration file to add the new userSettings and applicationSettings sections.

It is also how you can add your own custom configuration sections.

In a nutshell, you do this by extending the `ConfigurationSection` and `ConfigurationElement` classes. Within each of your derived classes, you will be decorating the members with attributes like `ConfigurationPropertyAttribute`.

Simple right? Just kidding. Others have provided excellent descriptions of this slightly complicated, but not too difficult mechanism. I include links at the end of this document for further reading.

Here, I only wanted to provide a couple of hints that can get lost in the longer descriptions.

When you are all done, you'll want to create an XML Schema Document (XSD file), that describes your section, and include it in your project. When you have your configuration file open, you can use the XML...Schemas menu item to be sure that Intellisense finds your XSD file.

Depending on your version of Visual Studio, you may hear some whining about the xmlns attribute on your configuration section. If you do, there is an easy solution. I generally derive from my own intermediate class, `XmlnsConfigurationSection`, rather than directly from `ConfigurationSection`. It makes the problem go bye-bye!

```
using System.Configuration;

namespace Extras.Configuration
{
  abstract public class XmlnsConfigurationSection : ConfigurationSection
  {
    [ConfigurationProperty("xmlns")]
    public string Xmlns
    {
      get { return (string)this["xmlns"]; }
      set { this["xmlns"] = value; }
    }
  }
}
```

Further Reading

- ASP.NET Configuration File Hierarchy and Inheritance
- Unraveling the Mysteries of .NET 2.0 Configuration
- Decoding the Mysteries of .NET 2.0 Configuration
- Cracking the Mysteries of .NET 2.0 Configuration
- System.Configuration Namespace
- ConfigurationSection Class
- ConfigurationElement Class
- ConfigurationPropertyAttribute Class
- ConfigurationElementCollection Class

History

- 7/4/2013 - The original posting of this article.
- 3/11/2021 - The article was updated to warn of more recent configuration mechanisms in .NET.

License

About the Author

Eric Lynch
Software Developer (Senior)
United States

Eric is a Senior Software Engineer with 30+ years of experience working with enterprise systems, both in the US and internationally. Over the years, he's worked for a number of Fortune 500 companies (current and past), including Thomson Reuters, Verizon, MCI WorldCom, Unidata Incorporated, Digital Equipment Corporation, and IBM. While working for Northeastern University, he received co-author credit for six papers published in the Journal of Chemical Physics. Currently, he's enjoying a little time off to work on some of his own software projects, explore new technologies, travel, and write the occasional article for CodeProject or ContentLab.

Comments and Discussions

34 messages have been posted for this article Visit **https://www.codeproject.com/Articles/616065/Why-Where-and-How-of-NET-Configuration-Files** to post and view comments on this article, or click **here** to get a print view with messages.