App.Config: Basics and Best Practices

August 21, 2018 Peter Morlion Articles, CodeBasics, PeterMorlion

No Comments

In one of my previous posts, I wrote about the .NET build configuration system. I mentioned the app.config file, but didn't really dive into it. So let's take a closer look at this file now.
When you create a (non-web) .NET Framework application in Visual Studio, an app.config file is added to your project. When you create a class library or a .NET Core project, such a file is not included, although it can be done afterward.

In a web project (i.e. ASP.NET) you will use a similar file, the web.config. A lot of what I will cover in this article is applicable to web projects, but there are subtle differences. That is why I'll only focus on the app.config.

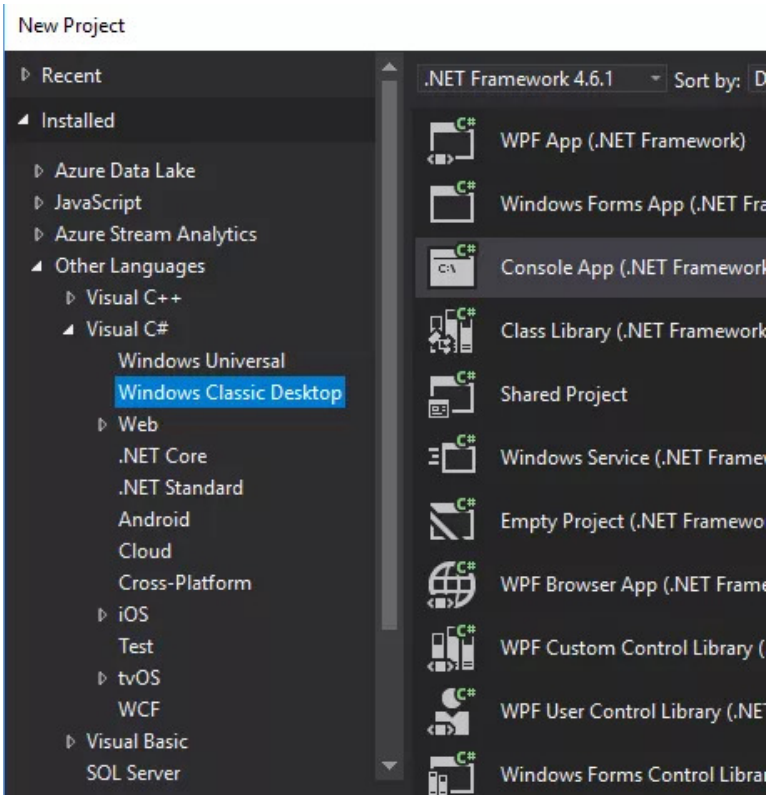
What Is It?

The app.config file is an XML file whose goal it is to contain any variable configuration of your application. It is a central place to put:

- Connection strings to databases
- Connection details to external services
- Application settings
- Other details on how the application should be run and/or hosted

What Does It Look Like?

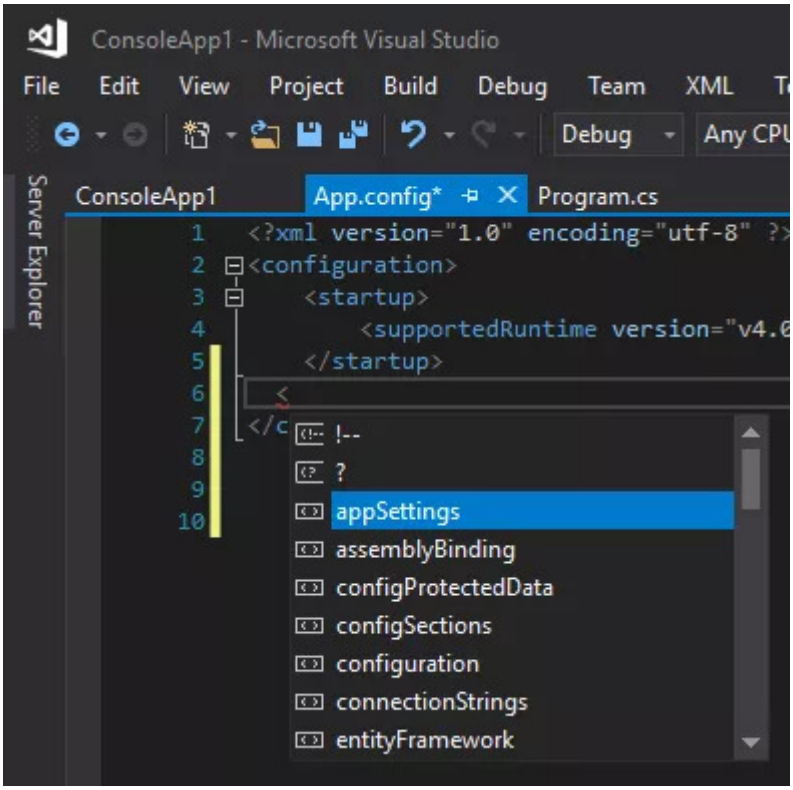
An app.config file is automatically added to your project when you create a new application under the Windows Classic Desktop header in Visual Studio:



When you open the file, there's not much in it:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6.1" />
    </startup>
</configuration>
```

You can start typing under the <configuration> section and Visual Studio's Intellisense will provide you with the possible options:



The most well-known sections are "appSettings" and the "connectionStrings".

AppSettings

AppSettings provide an easy way to access string values, based on a certain key. Take these appSettings for example:

```
<appSettings>
  <add key="pollInterval" value="4"/>
</appSettings>
```

We can access this value by using this piece of code:

```
var pollInterval = ConfigurationManager.AppSettings["pollInterval"];
```

As a side note: you will need to import the System.Configuration namespace too, by adding "using System.Configuration" at the top of your file.

A possible problem with AppSettings is that we will get back a string. Yet in the above example, we're clearly working with an integer. We will have to parse the value for it to be more

meaningful. Another issue is that there is no compile-time checking that we entered the correct key, as it is just a string.

If your needs are simple, you could be fine with this. If things start to get a little more complicated, you might want to look into “Settings”, which I’ll cover later.

ConnectionStrings

The connectionStrings section is the place to put your connection strings to any databases you want to access. This is a simple example:

```
<connectionStrings>
  <add name="DefaultConnectionString"
       connectionString="Data Source=serverName;Initial Catalog=Northwind;Persist Security Info=True;User ID=userName;Password=password"
       providerName="System.Data.SqlClient"/>
</connectionStrings>
```

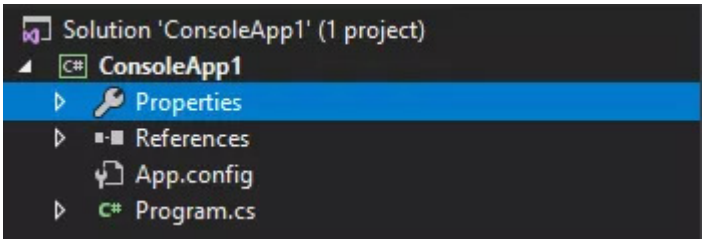
Notice how there is also a “providerName”. This tells ORM libraries like Entity Framework which provider they should be using. The provider will differ depending on the database you’re trying to access (e.g. SQL Server, PostgreSQL, MariaDB, etc.).

If you need to access the connection string directly, you can use code like this:

```
using (var sqlConnection = new SqlConnection(ConfigurationManager.ConnectionStrings["DefaultConnectionString"].ConnectionString))
using (var sqlCommand = new SqlCommand("...", sqlConnection))
{
    // Use sql command here
}
```

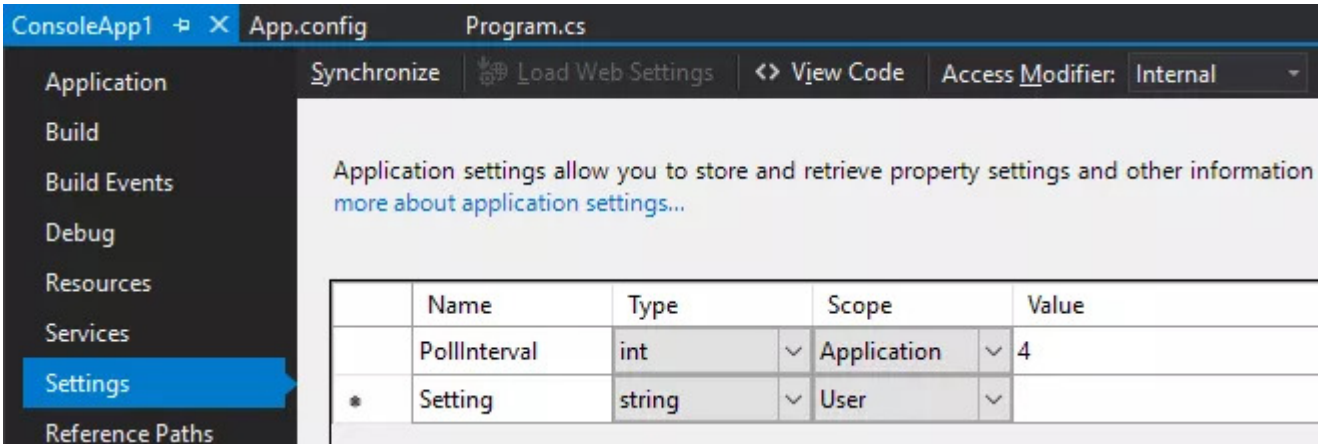
Settings

Remember how the AppSettings only provided us with string values? The Settings system was introduced in .NET 2.0 to improve this. To use this, double click the properties of your project:



Then go to the settings tab. In the beginning, your project will not yet have a default settings file, so click on the link to create one. You’ll see a nice UI where you can define your settings, their

values, scopes, and types:



I've added our "pollInterval" in these settings, and now I can access it like this:

```
var pollInterval = Properties.Settings.Default.PollInterval;
```

The nice thing is that we have compile-time checking now, and we're sure the "pollInterval" variable is an integer.

The Settings system has now created two new files under the Properties folder: Settings.Designer.cs and Settings.settings. You do not need to look into these files, and it's best not to change them manually. They are automatically generated, so any changes will be overwritten anyway.

The result of these two files is a class containing the settings as properties. By the way, you can change the access (public or internal) by using the dropdown in the Settings tab of the project properties.

Working with the Settings system will also alter your app.config. In the above example, our app.config now includes this:

```
<applicationSettings>
  <ConsoleApp1.Properties.Settings>
    <setting name="PollInterval" serializeAs="String">
      <value>4</value>
    </setting>
  </ConsoleApp1.Properties.Settings>
</applicationSettings>
```

For more on the .NET settings system and the difference in scopes, check out the MSDN documentation.

Custom Configuration Sections

There is a next step in app.config possibilities: custom configuration sections. These merit a separate article, so I will only cover them here shortly.

As I mentioned at the beginning of this article, a class library doesn't contain an app.config file. But sometimes, the author of a class library might want to allow other developers to configure certain parts of the library in the host's app.config file.

One option would be for the author to document the necessary AppSettings. But the AppSettings have the disadvantages I've already mentioned, and they don't allow for any deeper hierarchy.

Custom configuration sections give the class library author the solution to this. When the author has set it all up correctly, the developer using the library can add something like this to the app.config file:

```
<configuration>
  <configSections>
    <sectionGroup name="pageAppearanceGroup">
      <section
        name="pageAppearance"
        type="Samples.AspNet.PageAppearanceSection"
        allowLocation="true"
        allowDefinition="Everywhere"
      />
    </sectionGroup>
  </configSections>

  ...

  <pageAppearanceGroup>
    <pageAppearance remoteOnly="true">
      <font name="TimesNewRoman" size="18"/>
      <color background="000000" foreground="FFFFFF"/>
    </pageAppearance>
  </pageAppearanceGroup>

</configuration>
```

This is taken from the .NET docs, and even though it concerns ASP.NET, the same mechanism is used in classic desktop apps.

As you can see in the above example, the developer using the library

- first defines a new sectionGroup under the configSections tag
- then defines a section under that, pointing to the correct type that will handle the config section
- and finally provides the configuration details.

You can see that the pageAppearance is more complex than the simple AppSettings or Settings sections can handle.

As I said, this is beyond the purpose of this article, but I encourage you to check out the documentation I linked to above.

There's More

So far, we've only covered application settings. But we can consider connection strings as a sort of application setting too.

But the app.config file can contain more than that. Remember how a default app.config contained this:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6.1" />
    </startup>
</configuration>
```

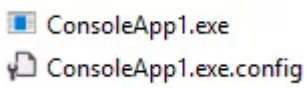
This is where we define which .NET runtime(s) the application supports. You could change this if your application needs to run on other (higher or lower) runtimes.

The assemblyBinding tag is another tag you will encounter easily. In short, it allows you to point to a certain version of an assembly, even when a different version is required. Again, this is out of the scope of this article, but the documentation should help you further.

Another example is the system.serviceModel tag, which is the place to configure any connections to WCF services.

What Happens When We Compile This?

Good, by now you know what the app.config is for and how to use it. Now you might be wondering what happens behind the scenes. Well, when you compile your application, the compiler actually copies the app.config file to the output folder, but gives it another name:



When you start your application (ConsoleApp1.exe in our example), the matching config file will be loaded too.

It is possible to change this file while the application is running, but you will need to restart the application for the changes to take effect. Luckily, there are ways around this:

- A call to ConfigurationManager.RefreshSection("...") will reload the given section
- For applicationSettings, you will need to call Properties.Settings.Default.Reload();

Keep in mind that a new compilation or deployment can overwrite the changed values. So it's better to make the changes in your source code (which is under source control, right?) and redeploy.

Secrets

You should consider not putting confidential information in your config files. Things like passwords and API keys should be kept out of the app.config file. In an age where more and more code is made open source, you could risk exposing your secrets to the outside world. Even if your repository is private, it's a best practice to put the secrets elsewhere. First, if you find it normal to do so, you won't do it when you're working on a public and open source project. But second, you never know when your private repository may become public, on purpose or by accident.

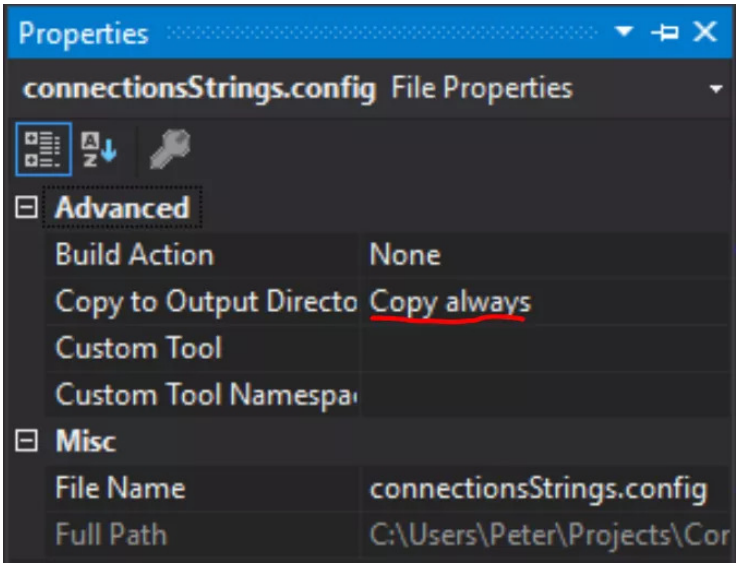
You can easily solve this by putting your secrets in separate files:

```
<appSettings file="app.SECRETS.config">
  <add key="pollInterval" value="4"/>
</appSettings>

<connectionStrings configSource="connectionStrings.config">
</connectionStrings>
```

See how both the appSettings and the connectionStrings point to another file? Also note that we can combine an external file with appSettings, but for connectionStrings we need to choose.

If we make sure we don't put these files under source control, they can remain absolutely private. Another thing to remember is to tell the compiler to copy these files to the output directory:



After that, you can use the appSettings and connectionStrings in your code like you did before.

A Final Tip

I want to revisit the point I made about the appSettings not being type-safe. If you have multiple places where you access your appSettings, you might not like having to repeat the (string) key everywhere, and having to convert the string value to the correct type every time. A simple solution to this is to create your own class that wraps the call to ConfigurationManager:

```
public class MyConfigManager
{
    public int PollInterval => int.Parse(ConfigurationManager.AppSettings["pollInterval"]);
}
```

In all other locations of your code, you can now call this class and be sure you're getting an integer.

Happy Configuring!

This article aimed at showing you the basics of the app.config file, and some best practices. The app.config file is a basic piece of the .NET Framework, yet I've seen several projects putting their configuration in other places (like plain text files or the registry). Unless you have a very good reason to do so, it's more convenient and familiar to use the app.config file. You can change it easily, the .NET Framework supports it out of the box and almost all .NET developers know how it works.

About the author

Peter Morlion

Contributing Author

Peter Morlion is a passionate programmer that helps people and companies improve the quality of their code, especially in legacy codebases. He firmly believes that industry best-practices are invaluable when working towards this goal.

Related

Understanding the .NET Build Configuration System
May 22, 2018
In "Articles"

StyleCop: A Detailed Guide to Starting and Using It
April 10, 2018
In "Articles"

.NET Code Documentation So Easy It's an Afterthought
July 20, 2017
In "Articles"