Wrapping up Configuration Manager

Many developers use the ConfigurationManager class to retrieve settings from the .Config file of your application. This class allows you to retrieve settings from the <appSettings> element. With just a single line of code as shown in the following line:

```
file = ConfigurationManager.AppSettings["CustomerFile"];
```

The above code assumes you have the following setting declared in your App.Config or Web.Config file for your application:

```
<configuration>
  <appSettings>
    <add key="CustomerFile"
         value="D:\Samples\Customers.xml" />
  </appSettings>
</configuration>
```

This works great except in the case where you misspell the key name in your line of code, or you forget to add the element in the <appSettings> element in your .Config file. In either case, the value you get back will be a null value. Sometimes these null values can wreak havoc on your code because you forgot to check the value for null and the attempt to perform some operation on that value such as opening the file name. For example, the following code would cause a ArgumentNullException to be raised because the key value is misspelled.

```
string value = string.Empty;

value = ConfigurationManager.AppSettings["CustomerFil"];

System.IO.File.OpenText(value);
```

To fix the above code you need to test the “value” variable to see it is null prior to attempting to open the file as shown in the following code:

```
string value = string.Empty;

value = ConfigurationManager.AppSettings["TheValue"];

if (value == null)
  MessageBox.Show("Can't open the file");
else
  System.IO.File.OpenText(value);

MessageBox.Show(value.ToString());
```

Another issue with the above code is if you wish to change the location of where you are storing your application settings, you have to modify every line of code throughout your whole application where you used ConfigurationManager.AppSettings. For example, sometime in the future you might want to store your settings in an XML file located on a central server, or maybe store them in a database. This would be a lot of extra work due to you probably have a lot of places where you are retrieving configuration settings. Instead let’s wrap up the calls to the ConfigurationManager.AppSettings method in your own class.

Wrap Up Configuration Manager

A best practice that I have employed for over twenty years has been to wrap up any method calls that could potentially change in the future. Configuration settings, database access calls, registry settings, WCF Services, and similar types of classes are all candidates for wrapping up. Creating your own class allows you to change the methods for getting data from another source without you having to change the code in your application.

For the configuration settings shown previously, creating a class to wrap up the call to ConfigurationManager.AppSettings is a very simple task. Below is the complete class that will allow you to get a string and an integer value from a configuration file.

```
public class AppConfig
{
  public string XmlPath { get; set; }
  public int DefaultType { get; set; }

  // Wrapper around ConfigurationManager.AppSettings call
  protected string GetSetting(string key)
  {
    return ConfigurationManager.AppSettings[key];
  }

  public string GetSetting(string key, string defaultValue)
  {
    string value;

    value = GetSetting(key);
    if (value == null)
      value = defaultValue;

    return value;
  }

  public int GetSetting(string key, int defaultValue)
  {
    int ret;
    string value;

    value = GetSetting(key);
    if (value == null)
      ret = defaultValue;
    else
      ret = Convert.ToInt32(value);

    return ret;
  }
}
```

Notice how each of the **public** GetSetting methods call a **protected** method named GetSetting where the call to ConfigurationManager.AppSettings is made. Now, if you want to store your configuration settings in a database table, you only need to change the protected GetSetting method. All of the rest of the methods in this class and the calls you make in your application to the GetSetting methods in this class do not need to change at all.

In addition to wrapping up the call to the ConfigurationManager.AppSettings method call, you can add on additional functionality as well. For instance you can allow the programmer to pass in a default value to return if the value is not found in your settings storage location.

You call the GetSetting methods in AppConfig class using the following code:

```
AppConfig config = new AppConfig();

config.XmlPath = config.GetSetting("XmlPath", @"C:\");

MessageBox.Show(config.XmlPath);
```

In the above code if the “XmlPath” key was not found, then C:\ would be returned. The same call can be used for the overloaded version of GetSetting that returns an integer value. You pass in an integer value that you wish to have returned if the key “DefaultType” is not found.

```
AppConfig config = new AppConfig();

config.DefaultType = config.GetSetting("DefaultType", 1);

MessageBox.Show(config.DefaultType.ToString());
```

Summary

Wrapping up calls to .NET classes and their methods can give you more flexibility in the future if you wish to change the implementation of a method. In addition it allows you to add on functionality that is not present in the original calls. Wrapping up methods like this also protects you from changes that Microsoft might introduce in future versions of the .NET Framework. If they obsolete a class or method, you now only need to change your code in just place. The rest of your application does not need to change. In other words, you are helping to future-proof your code.

NOTE: You can download this article and the sample code that goes with this blog entry at my website. http://www.pdsa.com/downloads. Select “Tips and Tricks”, then “Wrapping up Configuration Manager” from the drop down list.

Good Luck with your Coding,

Paul Sheriff

** SPECIAL OFFER FOR MY BLOG READERS **

We frequently offer a FREE gift for readers of my blog. Visit http://www.pdsa.com/Event/Blog for your FREE gift!