Building a better .NET Application Configuration Class - revisited

December 28, 2012 - from Maui, Hawaii   Tweet 0   g+1 20   29 comments

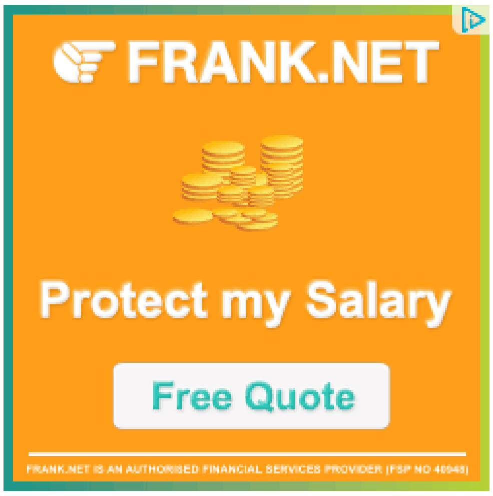
Many years ago, when I was just starting out with .NET (in the 1.0 days - now nearly 10 years ago), I wrote an article about an Application Configuration component I created to improve application configuration using strongly typed classes. To me, application configuration is something that I've been adamant about since - well forever, even pre-dating .NET. An easy to access configuration store along with an easy application access mechanism to access and maintain configuration settings through code are crucial to building applications that are adaptable. I found that the default configuration features built into .NET are pretty good, but they still take a bit of effort to maintain and manage.

So I built my own long ago, with focus on a code-first approach. The configuration library I created is small, low-impact and simple to use - you create a class and add properties, instantiate and fire away at configuration values. The library has been updated and majorly refactored a few times over the years to adapt to changes in .NET and common usage patterns. In this post I'll describe the latest updated version of this library as I think many of you might find it useful.

The very latest version of this library is now available on its own as a smaller self contained component. You can find the actual files and basic usage info here:

- **West Wind Application Configuration Home Page**
- **Source Code on GitHub**
- **NuGet for Westwind.Utilities.Configuration**

But before I jump in and describe the library lets spend a few minutes reviewing what configuration options are available in the box in .NET and why I think it made sense to maintain a custom configuration solution over the years.


FRANK.NET
Protect my Salary
Free Quote
FRANK.NET IS AN AUTHORISED FINANCIAL SERVICES PROVIDER (FSP NO 40048)

Configuration? How boring, but...

I consider configuration information a vital component of any application and I use it extensively for allowing customization of the application both at runtime and through external configuration settings typically using .config files and occasionally in shared environments with settings stored in a database. The trick to effective configuration in any application is to make creating new configuration values and using them in your application drop dead easy. To me the easiest way to do this is by simply creating a class that holds configuration values, along with a mechanism for easily serializing that configuration data. You shouldn't have to think about configuration - it should just work like just about any other class in your projects :-)

In my applications, I try to make as many options as possible user configurable and configure everything from user application settings, to administrative configuration details, to some top level business logic options all the way to developer options that allow me to do things like switch in and out of detailed debug modes, turn on logging or tracing and so on. Configuration information can be varied so it should also be easy to have multiple configuration stores and switch between them easily.

This is not exactly a sexy feature, but one that is quite vital to the usability and especially the configurability of an application. If I have to think about setting or using of configuration data too much, have to remember every setting in my head (ie. "magic strings"), or have to write a bunch of code to retrieve values, I'll end up not using them as much as I should, and consequently end up with an application that isn’t as configurable as it could be.

What does .NET provide for Configuration Management?

So what do you use for configuration? If you're like most like developers you probably rely on the AppSettings class which provides single level configuration values at the appSettings key. You know the kind that's stored in your web.config or application.config file:

```
<configuration>
  <appSettings>
    <add key="ApplicationTitle" value="Configuration Sample (default)" />
    <add key="ApplicationSubTitle" value="Making ASP.NET Easier to use" />
    <add key="DebugMode" value="Default" />
    <add key="MaxPageItems" value="0" />
  appSettings>
configuration>
```

All the configuration setting values are stored in string format in the appSettings section of an application's configuration file.

To access settings, the System.Configuration assembly in .NET provides a fairly easy way to access these configuration values via code from within applications:

```
string applicationTitle = ConfigurationSettings.AppSettings["ApplicationTitle"];
```

Easy enough, right? But it gets a little more complex if you need to grab a value that's not a string. For numeric or enum values you need to first ensure the value exists (is non-null) and then convert the string explicitly to whatever type, since configuration values are always strings. Here's what this looks like:

```
int maxPageItems = 0;
string tInt = ConfigurationManager.AppSettings["MaxPageItems"];
if (tInt != null)
    int.TryParse(tInt, out maxPageItems);

DebugModes mode = DebugModes.Default;
string tenum = ConfigurationManager.AppSettings["DebugMode"];
if (tenum != null)
    Enum.TryParse(tenum,out mode);
```

which is a bit verbose if you have to go through this every time you want to use a configuration value. You also have to remember the values and use the ConfigurationManager.AppSettings class. Minor but 'noisy' in application level code.

AppSettings values are also limited to a single *AppSettings* section inside of an application's or Web config file. Luckily you can also create custom configuration sections use the same configuration format in custom sections in a config file as long as those custom sections get declared:

```
<configuration>
  <configSections>
    <section name="CustomConfiguration" requirePermission="false"
             type="System.Configuration.NameValueSectionHandler,System,Version=1.0.3300.
  configSections>
  <CustomConfiguration>
    <add key="ApplicationName" value="Configuration Tests" />
    <add key="DebugMode" value="Default" />
    <add key="MaxDisplayListItems" value="15" />
    <add key="SendAdminEmailConfirmations" value="False" />
    <add key="MailServer" value="3v7daoNQzllLoX0yJE2weBlljCp0MgyY8/DVkRijRTI=" />
    <add key="MailServerPassword" value="ud+2+RJyqPifhK4FXm3leg==" />
  CustomConfiguration>
configuration>
```

You can then access a custom section with:

```
var settings = ConfigurationManager.GetSection("CustomConfiguration") as NameValueCollec
Console.WriteLine(settings["ApplicationName"]);
```

and essentially get the same behavior as you get with the AppSettings keys. The collection you get back is a read-only NameValueCollection that's easy to run through and read from.

.NET's configuration provider also supports strongly typed configuration sections via code, which involves creating classes based base on the *ConfigurationSection* class. This gives you a slightly different configuration format that's a little less verbose than the add/key/value structure of NameValue type configuration:

```
xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="MyCustomConfiguration" requirePermission="false"
             type="Westwind.Utilities.Configuration.MyCustomConfigurationSection,Westwir
  configSections>
  <MyCustomConfiguration
        ApplicationName="Configuration Tests"
        MaxDisplayItems="25"
        DebugMode ="ApplicationErrorMessage"
        />
configuration>
```

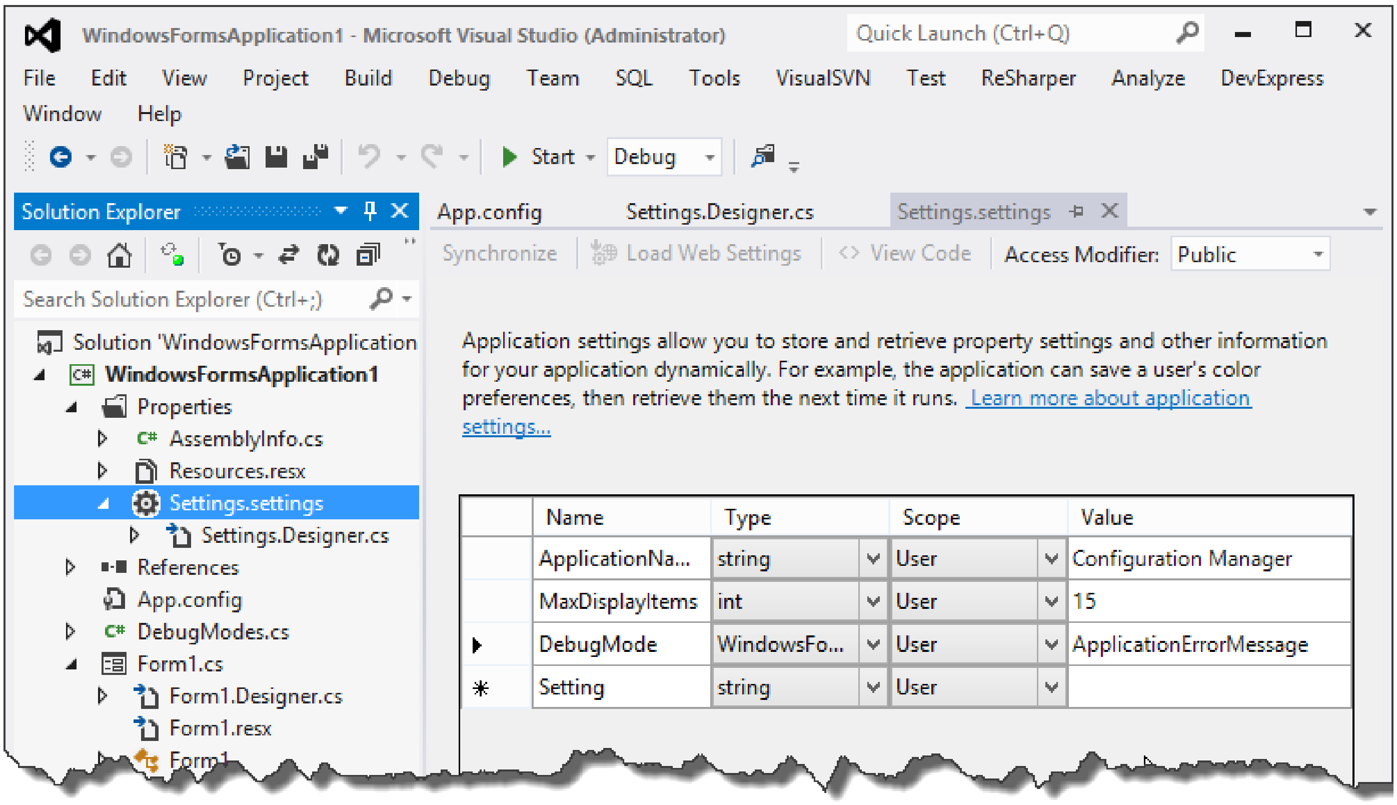
This is a little more involved in that you need to define a class and define each property along with some inherited logic to retrieve the configuration value.

```
class MyCustomConfigurationSection : ConfigurationSection
{
    [ConfigurationProperty("ApplicationName")]
    public string ApplicationName
    {
        get { return (string) this["ApplicationName"]; }
        set { this["ApplicationName"] = value; }
    }

    [ConfigurationProperty("MaxDisplayItems",DefaultValue=15)]
    public int MaxDisplayItems
    {
        get { return (int) this["MaxDisplayItems"]; }
        set { this["MaxDisplayItems"] = value; }
    }

    [ConfigurationProperty("DebugMode")]
    public DebugModes DebugMode
    {
        get { return (DebugModes) this["DebugMode"]; }
```

```
            set { this["DebugMode"] = value; }
        }
    }
```

but the advantage is that you can reference the class as a strongly typed class in your application. With a bit of work you can even get Intellisense to work on your configuration settings inside of the configuration file. You can find out more from this detailed article from Rob Seeder. Strongly typed configuration classes are useful for static components that have lots of configuration settings, but for typical dynamic configuration settings that frequently change in applications the more dynamic section of key value pairs is more flexible and easier to work with dynamically.

For certain kinds of desktop applications, Visual Studio can also create a strongly typed Settings class. If you create a WinForms or WPF project for example it adds a Settings.settings file, which lets you visually assign properties in a designer. When saved the designer creates a class that accesses the AppSettings values indirectly.


WindowsFormsApplication1 - Microsoft Visual Studio (Administrator)
Quick Launch (Ctrl+Q)
File Edit View Project Build Debug Team SQL Tools VisualSVN Test ReSharper Analyze DevExpress Window Help
Start Debug
Solution Explorer
App.config Settings.Designer.cs Settings.settings
Synchronize Load Web Settings View Code
Search Solution Explorer (Ctrl+;)
Solution 'WindowsFormsApplication
WindowsFormsApplication1
Properties
AssemblyInfo.cs
Resources.resx
Settings.settings
Settings.Designer.cs
References
App.config
DebugModes.cs
Form1.cs
Form1.Designer.cs
Form1.resx
Form1

Access Modifier: Public

Application settings allow you to store and retrieve property settings and other information for your application dynamically. For example, the application can save a user's color preferences, then retrieve them the next time it runs. Learn more about application settings...

	Name	Type	Scope	Value
	ApplicationNa...	string	User	Configuration Manager
	MaxDisplayItems	int	User	15
▶	DebugMode	WindowsFo...	User	ApplicationErrorMessage
*	Setting	string	User	

This is pretty nice in that it keeps all configuration information inside of a class that is managed for you as you add values. You also get default values and you can easily use the class in code:

```
var settings = new Settings();
var mode = Settings.DebugMode;
MessageBox.Show(mode.ToString());
```

The class is strongly typed and internally simply references a custom configuration section of values that are read from the config file. This is a nice feature, but it's limited to desktop Windows applications Console, WinForms, WPF and Windows 8 applications. It's also limited to a single configuration class.

Writing Values to the Config File

.NET also has support for writing configuration values back into configuration files via the configuration manager. You can load up a configuration, load a section and make changes to it, then write the entire configuration back out to disk assuming you have permissions to do this.

```
var config = ConfigurationManager.OpenExeConfiguration(ConfigurationUserLevel.None);
var section = config.GetSection("appSettings") as AppSettingsSection;
section.Settings.Add("NewKey", "Value");
config.Save();
```

This also works for Web Applications where you can use:

```
var config = WebConfigurationManager.OpenWebConfiguration("~");
```

to read the top level configuration.

Permissions are crucial here and often you will not be able to write configuration back using this approach. For Web applications Full Trust and read/write access to the web.config file are required. For desktop applications write file rights in the folder are required, which is often not available - with User Account Control on you typically don't have rights to write into the Configuration folder.

Clearly there are a lot of choices available in .NET to handle configuration storage and retrieval. It's great that the ConfigurationManager is available to provide base features to create simple configuration storage quickly.

Creating a better Application Configuration Class

Personally though, I prefer a more structured approach for configuration management in my applications. Like everything else in my applications I expect my configuration settings to be based on one or more classes that I can simply add properties to and persist that data easily in my application.

There are some native choices available for that - after all .NET includes easy to use tools for serializing to XML and JSON. It's pretty trivial to create some code to arbitrarily take a class and serialize it. However, wouldn't it be nice if the format was easily switchable and if you didn't have to worry about writing out the data yourself?

When I created the ApplicationConfiguration component years ago that was my goal. The current incarnation of the Westwind ApplicationConfiguration library provides the following features:

- Strongly typed Configuration Classes
- Simply create a class and add Properties
- Automatic type conversion for configuration values
- Default values so you never have to worry about read failures
- Automatic synching of class and configuration store if values are missing
- Easily usable from any kind of .NET application or component
- Support for multiple configuration objects
- Multiple configuration formats
    - Standard .NET config files
        - Custom Sections
        - External Config files
        - AppSettings
    - Standalone XML files (XML Serialization)
    - Strings
    - Sql Server Tables
    - Customizable with easy to create ConfigurationProviders

How to use the AppConfiguration Class

The core of the Westwind Application Configuration library relies on a configuration class that you implement simply by inheriting from the *Westwind.Utilities.Configuration.AppConfiguration* class. This base class provides the core features for reading and writing configuration values that are properties of the class that you create. You simply create properties and instantiate the class and call Initialize() to initially assign the provider and load the initial configuration data.

To create a configuration class is as easy as creating a class and adding properties:

```
class MyConfiguration : Westwind.Utilities.Configuration.AppConfiguration
{
    public string ApplicationName { get; set; }
    public DebugModes DebugMode { get; set; }
    public int MaxDisplayListItems { get; set; }
    public bool SendAdminEmailConfirmations { get; set; }
    public string MailServer { get; set; }
    public string MailServerPassword { get; set; }

    public MyConfiguration()
    {
        ApplicationName = "Configuration Tests";
        DebugMode = DebugModes.Default;
        MaxDisplayListItems = 15;
        SendAdminEmailConfirmations = false;
        MailServer = "mail.MyWickedServer.com:334";
        MailServerPassword = "seekrity";
    }
}
```

To use the configuration class you can then simply instantiate the class and call Initialize() with no parameters to get the default provider behavior and then fire away at the configuration values with the class properties:

```
var config = new MyConfiguration();
config.Initialize();

// Read values
string appName = config.ApplicationName;
DebugModes mode = config.DebugMode;
int maxItems = config.MaxDisplayListItems;
```

Note that the *Initialize()* method should *always* be called on a new instance to initialize the configuration class. Initialize() internally assigns the provider and reads the initial configuration data from a store like the configuration file/section.

Once the class is instantiated and initialized you can go ahead and read values from the class. The values are loaded only once during Initialize() (or Read() if you decide to re-read settings manually) and are cached in the properties after the initial load. The values of the properties reflect the values of the configuration store - here from the application's config or web.config file, in a MyConfiguration section.

If the configuration file or section or values don't exist and the file is writable the releavant .config file is created. The content of the file looks like this:

```
<configuration>
  <configSections>
    <section name="MyConfiguration" requirePermission="false"
             type="System.Configuration.NameValueSectionHandler,System,Version=1.0.3300.
  configSections>
  <MyConfiguration>
    <add key="ApplicationName" value="Configuration Tests" />
    <add key="DebugMode" value="Default" />
    <add key="MaxDisplayListItems" value="15" />
    <add key="SendAdminEmailConfirmations" value="False" />
    <add key="MailServer" value="mail.MyWickedServer.com:334" />
    <add key="MailServerPassword" value="seekrity" />
  MyConfiguration>
configuration>
```

Note that a custom Section is created in the config with standard key values. The Initialize() method also takes an optional sectionName parameter that lets you explicitly override the section name. You can also use *appSettings* as the section name in which case the standard appSettings section is used without any custom configuration section configuration.

In the code above the configuration section is written automatically as part of the Initialize() code - but you can also explicitly write out configuration information using the Write() method:

```
var config = new MyConfiguration();
config.Initialize();

config.DebugMode = DebugModes.ApplicationErrorMessage;
config.MaxDisplayListItems = 20;

config.Write();
```

The key with calling the Write() method is that you have to have the permissions to write to the configuration store. For example, typical Web applications don't have rights to write to web.config unless you give explicit Write permissions to the file to the Web user account. Likewise, typical Windows applications installed in the Program Files folder can't write to files in the installation folder due to User Account Control permissions, unless you explicitly add rights for the user to write there. Location matters, so it's important to understand your environment before writing configuration values or expecting them to auto-created when initializing.

Using the Configuration Class as a Singleton

Because configuration data tends to be fairly static in most applications, it's a not a good idea to instantiate the Configuration class every time you need to access the configuration data. It's fairly expensive to read a file from disk, or access a database and the deserialize the values from the configuration into an object. It's much better to set up the configuration class once at application startup or using a static property to keep an active instance of the configuration class around.

Personally I prefer the latter by using a 'global' application object that I tend to have in every application. I can attach the configuration object on this application class as a static property. The advantage of a static property on a 'global' object is that it's portable and I can stick it into my business layer and use it in a Web app, a service or desktop application without any changes (at least when using config files). In Web applications static properties are also available to all threads so many simultaneous Web requests can share configuration information from the single instance.

To create a static Singleton is easy with code like this:

```
public class App
{
    public static MyConfiguration Configuration { get; set; }

    static App()
    {
        Configuration = new MyConfiguration();
        Configuration.Initialize();
    }
}
```

Now anytime you need access to the configuration class you can simply use:

```
DebugModes mode = App.Configuration.DebugMode;
int maxItems = App.Configuration.MaxDisplayListItems;
```

You never need to worry about instantiating the configuration class in your application code, it's just always there and cached to boot.

Using and customizing Configuration Providers

So far I've used only the default provider which is the ConfigurationFileConfigurationProvider class using default options, which use the standard .NET application configuration file and a section with the same name as the class within it. This means yourexe.exe.config or web.config for Web applications typically.

The default provider behavior using the ConfigurationFileConfigurationProvider is the most likely use case for the configuration configuration, but you can certainly customize the provider or even the behavior of the a provider by passing a custom provider to the Initialize() method. Initialize() takes a parameter for a Provider instance, a section name and arbitrary configData.

For example to use a custom section in the default configuration file you can specify the sectionName parameter in Initialize():

```
var config = new MyConfiguration();
config.Initialize(sectionName: "MyAdminConfiguration");
```

Of course you can also pass in a completely configured ConfigurationProvider instance which allows you to set all the options available on a provider:

```
var config = new AutoConfigFileConfiguration();

// Create a customized provider to set provider options
var provider = new ConfigurationFileConfigurationProvider<AutoConfigFileConfiguration>()
{
    ConfigurationSection = "MyCustomConfiguration",
    EncryptionKey = "seekrit123",
    PropertiesToEncrypt = "MailServer,MailServerPassword"
};

config.Initialize(provider);

// Config File and custom section should exist
string text = File.ReadAllText(TestHelpers.GetTestConfigFilePath());

Assert.IsFalse(string.IsNullOrEmpty(text));
Assert.IsTrue(text.Contains(""));
```

You can also opt to use a completely different provider than the ConfigurationFileConfigurationProvider used so far in the examples. It's easy to create a provider instance and assign it during initialization, but realistically you'll want to embed that default logic directly into the provider itself so the logic to instantiate is encapsulated within the provider itself.

The following is an example of a configuration class that defaults to a database provider:

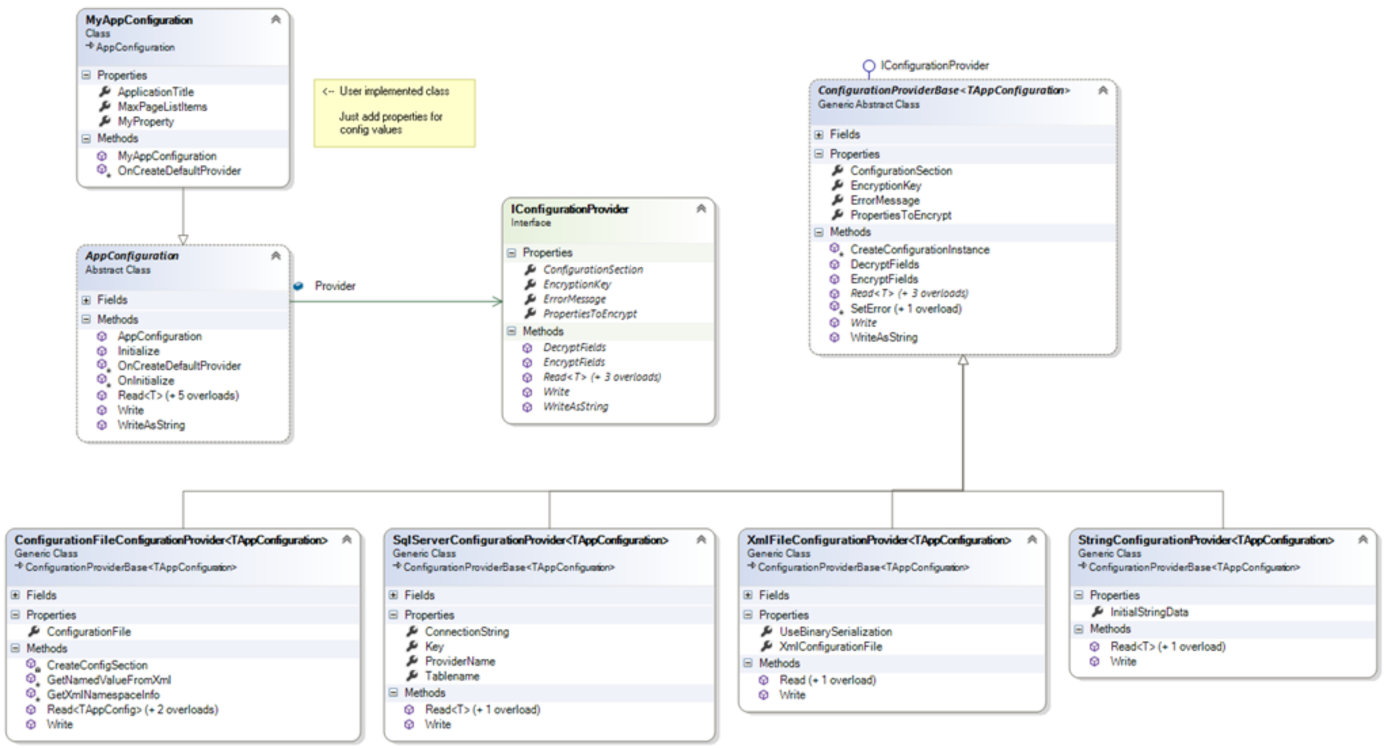
```
public class DatabaseConfiguration : Westwind.Utilities.Configuration.AppConfiguration
{
    public string ApplicationName { get; set; }
    public DebugModes DebugMode { get; set; }
    public int MaxDisplayListItems { get; set; }
    public bool SendAdminEmailConfirmations { get; set; }
    public string Password { get; set; }
    public string AppConnectionString { get; set; }

    // Must implement public default constructor
    public DatabaseConfiguration()
    {
        ApplicationName = "Configuration Tests";
        DebugMode = DebugModes.Default;
        MaxDisplayListItems = 15;
        SendAdminEmailConfirmations = false;
        Password = "seekrit";
        AppConnectionString = "server=.;database=hosers;uid=bozo;pwd=seekrit;";
    }
    /////

    ///// Override this method to create the custom default provider - in this case a da
    ///// provider with a few options.
    /////

    protected override IConfigurationProvider OnCreateDefaultProvider(string sectionName
    {
        string connectionString = "LocalDatabaseConnection";
        string tableName = "ConfigurationData";

        var provider = new SqlServerConfigurationProvider<DatabaseConfiguration>()
            {
                Key = 0,
                ConnectionString = connectionString,
                Tablename = tableName,
                ProviderName = "System.Data.SqlServerCe.4.0",
                EncryptionKey = "ultra-seekrit", // use a generated value here
                PropertiesToEncrypt = "Password,AppConnectionString"
            };

        return provider;
    }
}
```

This class implements the OnCreateDefaultProvider() method which is overridden to provide... a customized provider instance. The method receives a section name (which may or may not be used) and an optional configData parameter. configData can contain any arbitrary data that you can pass to the Initialize() method. For example, you might pass in a connection string value, or an anonymous object that contains both the connection string and table name that are hardcoded into the method above.

By implementing the above method the default behavior now loads the database provider, but you can still override the provider by explicitly passing one into the Initialize() method.

Providers really are the key to the functionality provided by the Application Configuration library - they're the work horses that do the work of retrieving and storing configuration data. The Westwind Application Configuration library consists of the AppConfiguration base class, plus a host of configuration providers that provide the actual logic of reading and writing of configuration values.

Here's the overall class layout:



Your class inherits from AppConfiguration which in turn contains a configuration provider instance. The instance is created during the Initialize() call - either using the default Configuration File provider or the custom provider passed in, or the provider you explicitly implement in OnCreateDefaultProvider(). The providers then implement the Read() and Write() methods responsible for retrieving the configuration data.

Configuration File Provider

The configuration file provider uses the .NET ConfigurationManager API to read values and direct XML DOM manipulation to add values to the config file. I opted for using the DOM rather than that ConfigurationManager to write values out, as there are fewer permissions issues. The Configuration API requires Full Trust to write because it has access to machine level configuration and using XML DOM and file IO allows writing of config files as long as the file permission are valid and it can work even in Medium trust. Configuration values are read one at a time and populated on the object in place, which means that if you use this you can call Initialize() as part of the constructor and automate instantiation without requiring a separate call to Initialize() from application code.

The configuration file provider allows for writing configuration files in separate locations, in customized sections as well as using the standard appSettings section. The default is to use the normal application configuration file in a section with the same name as the class.

Key Properties:

- ConfigurationFile
- ConfigurationSection

XmlFileConfigurationProvider

At first glance the XML file configuration provider sounds a lot like the ConfigurationFile provider. Both write output into files and use XML, but the XML provider is separate from .NET's configuration file system. This means configuration values written out to file don't automatically notify the calling app of changes. The XML File provider also relies on standard .NET serialization to produce the file output, which means that you can save much more complex data in a configuration class than with configuration file sections, which require a flat object structure.

XML files allow for more complex structure as you can go directly from object to serialized XML output. So if your configuration data includes nested data or needs to track collections of values, the XML Configuration provider can be a much better choice than .config files. This provider uses XML serialization to write XML directly to and from files.

Key Properties:

- XmlConfigurationFile

SqlServer Configuration Provider

The SQL Server configuration provider stores configuration information in a SQL Server table with two fields: A numeric Id and a single text field that contains an XML serialized string. Multiple sets of configuration values can be stored in configuration table using data rows based on the Id. The provider works with SQL Server and SQL Server Compact and may also work with other providers (not tested). To use this provider you provide a connection string to the database plus an optional table name and an optional Id value for the configuration. The table name defaults to ConfigurationSettings and that table has an integer Id and ConfigData text field.

This provider also relies on XML Serialization - it attempts to read the data from the database, if it doesn't exist creates the table and inserts the value. You can specify an Id number that identifies the configuration instance - so you can create multiple configuration classes and map them to separate records in the configuration table.

Key Properties:

- ConnectionString (Connection string or Connection Strings Entry)
- Tablename
- ProviderName (ADO.NET Provider name)
- Key (integer Id for configuration record)

String Configuration Provider

String serialization is mostly useful to capture the configuration data and push it into some alternate and unsupported storage mechanism. Some applications might store configuration data entirely in memory, or maybe the configuration data is user specific and can live in the ASP.NET Session store for example.

This provider is largely unnecessary as string serialization is built directly into the core AppConfiguration class itself. You can use assign XML data to the config object with config.Read(string xml) to read configuration values from an XML string in XmlSerialization format and use the WriteAsString() method to produce a serialized XML string.

Key Property:

- InitialStringData (the initial string to deserialize from)

Configuration Property Encryption

The base configuration provider also allows for encryption of individual configuration properties. Rather than encrypting an entire configuration section it's possible to encrypt only certain sensitive values like passwords and connection strings etc. which makes it easy to change most keys in a configuration file as needed, and leave a few sensitive values to be encrypted either on a development machine or via a Write() operation of the configuration.

To create encrypted keys specify the PropertiesToEncrypt property with a comma delimited list of properties that are to be encrypted. You also need to provide a string encryption key, which is used to handle the two-way encryption:

```
var provider = new ConfigurationFileConfigurationProvider<MyConfiguration>()
{
    ConfigurationSection = sectionName,
    EncryptionKey = "ultra-seekrit",  // recommend to use a generated value here
    PropertiesToEncrypt = "Password,AppConnectionString"
};
```

This provider produces a section in the configuration file that looks like this:

```
<MyConfiguration>
  <add key="ApplicationName" value="Changed" />
  <add key="DebugMode" value="DeveloperErrorMessage" />
  <add key="MaxDisplayListItems" value="12" />
  <add key="SendAdminEmailConfirmations" value="True" />
  <add key="Password" value="ADoCNO6L1HIm8V7TyI4deg==" />
  <add key="AppConnectionString" value="z6+T5mzXbtJBEgWqpQNYbBss0csbtw2b/qdge7PUixE=" />
MyConfiguration>
```

When the AppConfiguration class reads the values from the configuration file (or other configuration store), the values are automatically decrypted so the configuration properties are always unencrypted when accessed. The Write() operation writes out the encrypted values into the configuration file. As you can see, using encryption only works if you can somehow write to the file, otherwise the encrypted values never are stored in the configuration. This means you need to have permissions to write to the file, either at development time to create the original values, or on the live site.

Writing Values to the Configuration Store - Permissions

Application level configuration is pretty nice and because the Configuration class is just a plain class it's easy to create an updatable configuration management interface in your applications. You can bascially display and capture configuration values directly from the UI via Databinding or ModelBinding and then simply call config.Write() to write out configuration data to the configuration store. It's easy to do. For example, in an MVC application I can simply have a Configuration Controller Action and View that displays and captures the configuration data directly of the Configuration object. You can update the values in the UI and then simply call the Write() method to write the configuration data out to the store.

The key is that you have to have permissions for this to work. If you store configuration settings in web.config you need to give rights to Web account to be able to write the file. For Web applications it might actually be better to use an external configuration file for the configuration settings to avoid having to explicitly give write access to web.config. Similar issues might be necessary in some desktop scenarios - rather than writing configuration information into a file in the installation/execution folder of an application, read and write the configuration data to a file located in My Documents or AppData where the logged on user has full access.

Summary

Configuration is an important part for just about any application, and this component has been very useful to me over the years, making it an absolute no brainer to just drop in a configuration class into just about any application I build. As I go along during development, just about any parameterizable setting gets added to one or more configuration classes. In most of my applications I have an application level configuration class that holds app specific settings like customizable messages, sizes, measurements, default values etc. as well as an Admin specific configuration that holds things like mail server and sender information, logging options, debugging and profiling options etc.. In Web applications in particular it's super nice to make these kind of changes in web.config files, and have the change immediately take effect. It's a very satisfying experience.

Recently I took the time to clean up this component a bit and extract it from the West Wind Web Toolkit where's it's been living for some time in obscurity. It's still in the toolkit and its forthcoming new version, but I figured pulling it out as a standalone component and sharing it on GitHub might give a little more attention to this useful component. I hope some of you find it useful.

Resources

- **West Wind Application Configuration Home Page**
- **Source Code on GitHub**
- **NuGet for Westwind.Utilities.Configuration**

Posted in **.NET** **ASP.NET**

Make A Donation

Tweet 0 +1 20

Feedback for this Weblog Entry

# re: Building a better .NET Application Configuration Class - revisited

by Dan Nemec December 28, 2012 @ 12:39pm

You mention that there's support for multiple configuration stores on the Github README, but it seems like all that means is "create a configuration store, then create another one". Is there support at all for layered configuration stores?

For example, suppose that there are machine-wide configuration settings but each user has his/her own configuration file that can selectively override some settings. The most common scenario I find myself needing is support for a "default" configuration file (separate from a hardcoded constructor) that can be source controlled and copied to each (development, staging, and production) server without modification. On top of that, each server has a custom configuration file that may (or may not) override certain settings like connection strings.

I'm surprised the .Net community doesn't have a solution for that since it's a common pattern when working with Python/Django projects (although Python is a scripting language so the "configuration files" are just normal code files). I ended up having to write my own since I couldn't find an existing implementation that worked the way I needed it to.

# re: Building a better .NET Application Configuration Class - revisited

by Rick Strahl December 28, 2012 @ 2:10pm

@Dan - You're right this configuration manager basically expects you to create one or more self contained classes that map to a single store - there's no inheritance per se. But you can of course isolate configuration settings and store them separately - even in completely separate types of stores.

While I think that would be a useful feature, I'm not sure if this would be so common in an application/component level situation? I can see this being important for administrative type settings, but not so much for application/user level settings.

# re: Building a better .NET Application Configuration Class - revisited

by Dan Nemec December 29, 2012 @ 12:05pm

The most obvious situation I can think of is loading general settings from a file (or some other provider) and then allowing command line parameters to selectively override certain settings at runtime *without* needing a lot of glue code eg.

```
if(commandline.ConnectionString != null)
{
    settings.ConnectionString = commandline.ConnectionString
}
if(commandline.Timeout != null)
{
    settings.Timeout = commandline.Timeout
}
...etc.
```

Maybe it's not as common as I'm expecting it to be.

# re: Building a better .NET Application Configuration Class - revisited

by Rick Strahl December 29, 2012 @ 2:33pm

@Dan - sure that works just fine for application code, but how would you make that generic to fit as part of a component? Or how would you even want to specify this? Hand coding the above sort of thing is easy but how to make that generic and extensible.

# re: Building a better .NET Application Configuration Class - revisited
by Dan Nemec   December 29, 2012 @ 4:17pm

I'm not entirely sure what you mean "as part of a component" or if I'm misunderstanding what you're asking, but Reflection would enable it to scale to any number of configuration properties. So long as each Provider uses the same Type, we can just iterate over all properties and copy over any non-default values in order to merge the two objects. To detect whether a value is non-default, we can apply a `new()` constraint to our Configuration object and use the property values of a new'ed up object as the reference.

Given this configuration class:

```
public class Config
{
    public int Age { get; set; }

    public string Name { get; set; }
}
```

```
public static void MergeWith<T>(this T primary, T secondary, T defaultObject)
{
    foreach (var pi in typeof(T).GetProperties())
    {
        var secValue = pi.GetValue(secondary, null);
        var defaultValue = pi.GetValue(defaultObject, null);

        if (!Equals(secValue, defaultValue))
        {
            pi.SetValue(primary, secValue, null);
        }
    }
}
```

Imagine configA was populated from an XML file and configB was populated from a SQL database provider:

```
var defaultConfig = new Config();  // Age == 0 and Name == null

var configA = new Config
{
    Name = "Nate",
    Age = 10
};

var configB = new Config
{
    Name = "Fred",
    Age = 0  // This is the default value, it shouldn't be merged
};

configA.MergeWith(configB, defaultConfig);

// // Only the non-default properties were copied over:
// assert(configA.Name == "Fred")
// assert(configA.Age == 10)
```

Now we can pass configA on to whatever application logic needs the configuration object. As I mentioned earlier it allows us to keep a configuration file of sane defaults source controlled and use a separate, non source controlled file of "local overrides" if any values need to be changed per-host.

Most of the time we don't really care *where* the settings come from, just that each provider has an appropriate priority (eg. any values in the SQL provider take precedence over the XML provider) and that we shouldn't need to check multiple configuration objects every time we need a single config value.

# re: Building a better .NET Application Configuration Class - revisited
by Toni December 30, 2012 @ 11:58am

Interesting approach to this problem. I built my AppSettings (https://github.com/tparvi/appsettings) library about one year ago. I wanted something simple so I didn't implement support for anything else than basic settings (no sections).

# re: Building a better .NET Application Configuration Class - revisited
by Matt January 04, 2013 @ 5:14am

.settings files are not limited to Windows/WPF apps, you can use them just fine in web apps too.

The item template is missing from Visual Studio, but if you go to Add New Item > Text Document, and then change the file extension from ".txt" to ".settings", you'll get a fully functional settings designer.

A couple of caveats - you need to make sure all the settings are machine scoped, and you need to change the access modifier to public if you want to access the settings from a Razor view.

# re: Building a better .NET Application Configuration Class - revisited
by GJL January 04, 2013 @ 2:13pm

Nice work Rick

I used your previous version in an app a while back (storing in app.config), and was pleased.

I dont think there's any one solution for all the issues out there - I usually end up using a hybrid of this and say a data-table in SQLite or such... for my most recent app Im writing, I'm testing JSON as a configuration 'language'/representation rather than say XML - I like the ability to have arrays in JSON which map to lists of objects in my Config Object class - one negative about using JSON, I like to comment my config files, here XML wins :-)

One thing Im working on for either case is the ability to have a key and such a value :-

${app-path}/data/test-file.ext

And when the config value is loaded, ${app-path} is translated/interpolated. I know there are some frameworks out there for this sort of thing, String Template (Terence Parr?) comes to mind, but 'too big' for this as far as I see ..

Keep up the work and the thought provoking dialog
cheers
GJL

# re: Building a better .NET Application Configuration Class - revisited
by Rick Strahl January 04, 2013 @ 3:11pm

@GJL - I had thought about adding a JSON Configuration Provider but didn't for a couple of reasons. The main one is that would add a dependency on JSON.NET (or some other 3rd party library) in order to provide nicely formatted JSON support. The other is that XML Serialization (which is supported via the XmlFile Provider) really provides complex structure support so if you want to have nested objects or arrays/lists/collections of objects you can use that with XML.

As popular as JSON is, I think it's even MORE geeky than XML for a user editable format. I think even non-tech people probably can figure out XML formatting. JSON however, has requirements for quotes, escape characters and various other things. If user editing doesn't matter than using XML Serialization in binary mode is more efficient yet :-)

I'm going to look into the JSON Provider nevertheless though. several people have commented/asked for that in the past. Maybe using a dynamically loaded reference to JSON.NET using Reflection instantiation and dynamic would work to keep a hard coded reference out of the project.

# re: Building a better .NET Application Configuration Class - revisited
by Jeffl January 16, 2013 @ 11:35am

Rick,

I'm having a hard time with your class because the Initialize method is protected.

# re: Building a better .NET Application Configuration Class - revisited
by Rick Strahl   January 16, 2013 @ 11:09pm

@Jeff - you need to use the new westwind.utilities.configuration assembly - the old Westwind.Utilities version doesn't have the 2.0 version of this new functionality. Unfortunately there's no good way right now to mix the two short of copying the new code into the old project.

I'm in the process of reworking Westwind.Utilities for a new release that will have the new features - this will be a 2.0 version that includes some refactorings and also some removal of stuff the framework provides. Kind of a housecleaning to bring it up to date. Unfortunately I've not been able to push this out to Github just yet as there are still a bunch of housekeeping issues and documentation to take care of.

# re: Building a better .NET Application Configuration Class - revisited
by Jason Finch   January 30, 2013 @ 5:11pm

I found this Json Config library pretty handing dealing with ever changing configurations.

https://github.com/Dynalon/JsonConfig

Supports merging of configs.
Nested objects etc.

@Rick, you mention JSON is even more Geeky than XML.. Just wait until you have to use CDATA's or escape ampersands and angle brackets, XML can look geeky too!

Also for end users this helps to edit JSON. http://www.jsoneditoronline.org/

# re: Building a better .NET Application Configuration Class - revisited
by Rick Strahl   January 31, 2013 @ 4:26am

@Jason - well XML is not optimal for human editing either, but IMHO still way better than JSON editing with all of its encoding requirements. In the config class the XML generated is plain (either .config files or XML Serialization). As I mentioned it'd be pretty trivial to add JSON configuration format to this class, but it does add a dependency for a JSON serializer library (JSON.NET or System.Web.Extensions).

# re: Building a better .NET Application Configuration Class - revisited
by Fred Peters   April 01, 2013 @ 6:24pm

I had used your utility years ago and tried to use the new one from NuGet. But like Jeffl, no matter how I set things up I get the protected error on the Initialize method.

# re: Building a better .NET Application Configuration Class - revisited
by Rick Strahl   April 02, 2013 @ 2:28pm

@Fred - not sure why that would be. Where are you getting the NuGet package? It should be:

```
install-package Westwind.Utilities.Configuration
```

from the package manager console. I just tried this to confirm that indeed the right code is in there and that it works. Here's a small console example to demonstrate (New Console Project, add the NuGet is all that's needed):

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine(App.Configuration.ApplicationName);
        Console.WriteLine(App.Configuration.MaxDisplayItems);
        Console.Read();
    }
}
```

```
public class ApplicationConfiguration : AppConfiguration
{

    public string ApplicationName { get; set; }
    public int MaxDisplayItems { get; set; }

    public ApplicationConfiguration()
    {
        ApplicationName = "Test Application";
        MaxDisplayItems = 15;
    }
}

public class App
{
    public static ApplicationConfiguration Configuration { get; set; }

    static App()
    {
        Configuration = new ApplicationConfiguration();
        Configuration.Initialize();
    }
}
```

If you are using the existing Westwind.Utilities assembly from the West Wind Toolkit, that version is the old configuration object which requires constructor configuration.

There are new versions of these that are in beta now and can be found on GitHub:

https://github.com/RickStrahl/WestwindToolkit

The new version is not 100% backwards compatible. Almost all functionality is still there, but a number of things have been refactored into more logical classes etc. Currently none of these are on NuGet, but you can definitely pick up the new binaries out of the GitHub project.

Update:
I've also updated the Westwind.Utilities assembly to version 2.0 which is now available on NuGet and which includes the new configuration components supporting above syntax.

http://nuget.org/packages/Westwind.Utilities/

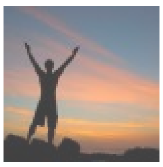
# re: Building a better .NET Application Configuration Class - revisited

by Fred Peters   April 22, 2013 @ 7:58am

Thanks Rick, it does work for the console application. I'll have to see what's different in my Winforms app.

# re: Building a better .NET Application Configuration Class - revisited

by Rick Strahl   April 22, 2013 @ 12:46pm



@Fred - Last week I also updated the NuGet package for Westwind.Utilities 2.0 which includes the new Application Configuration class. Just be aware that the new version is not 100% backwards compatible. The ApplicationConfiguration is the most prominent change - others are refactored classes and namespaces for more logic groupings.

PM > install-package Westwind.Utilities

If you use this package remove the Westwind.Utilities.Configuration assembly.

# re: Building a better .NET Application Configuration Class - revisited

by Andrew   June 06, 2013 @ 2:23am

Just want to clarify one thing regarding licensing. Your page says:

Usage for open source or personal projects is free. Any other usage commercial or otherwise requires registration for use of product on a per developer basis. This includes application in commerical, non-profit and government organizations. Once registered (either paid or personal/open source) you are allowed an Unlimited Runtime Distribution of compiled components for any number of applications.

If I'm going to use your compiled library in a commercial project or in a project for internal use of some company, and it is not a reusing of your code since our project is not a configuration library itself, do I need to obtain license? Or I'm free to use your binaries like that?

Please advise.
Thanks.

# re: Building a better .NET Application Configuration Class - revisited

by Trevor Moyakhe June 11, 2013 @ 3:33am

Used the XML serializer until I had to serialize arbitrary types contained in config classes. These unknown type would be created by plugin writers, and the main application just needed to know that there will be settings for a particular plugin. The object type is never relevant to the main app, since it never directly uses the plugin settings. I did, however, centralize the reading and writing of settings.

I also felt cheated, having Googled my way into this page, getting hold of the binaries and using them, before I read up on the licensing :-(

So, the XML serializer will not serialize unknown types and I need to pay if I want to wire these binaries to a solution I make money from. Don't think so.

Came up with the following two classes. You can use them without paying anyone, including me.

1. Serializer.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Xml;
using System.Runtime.Serialization;

namespace Ios.Serialization
{
    public class Serializer
    {
        public static void WriteObject(string fileName, object obj)
        {
            if (obj != null && !string.IsNullOrWhiteSpace(fileName))
            {
                FileStream stream = new FileStream(fileName, FileMode.Create);
                XmlDictionaryWriter writer = XmlDictionaryWriter.CreateTextWriter(stream);
                NetDataContractSerializer serializer = new NetDataContractSerializer();

                serializer.WriteObject(writer, obj);
                writer.Close();
            }
        }

        public static object ReadObject(string fileName)
        {
            object obj = null;

            if (!string.IsNullOrWhiteSpace(fileName) && File.Exists(fileName))
            {
                FileStream stream = new FileStream(fileName, FileMode.Open);
                XmlDictionaryReader reader = XmlDictionaryReader.CreateTextReader(stream, new Xm
                NetDataContractSerializer serializer = new NetDataContractSerializer();

                obj = serializer.ReadObject(reader, true);

                reader.Close();
            }

            return obj;
        }
    }
}
```

2. ConfigProvider.cs

```
using System.IO;

namespace Ios.Serialization
{
    public class ConfigurationProvider<T> where T : new()
    {
        private string _FileName;
        private T _Configuration;

        public ConfigurationProvider(string fileName)
        {
            _FileName = fileName;

            ReadConfig();
        }

        public T Configuration
        {
            get { return _Configuration; }
            set { _Configuration = value; }
        }

        public void ReadConfig()
        {
            if (!string.IsNullOrWhiteSpace(_FileName) && File.Exists(_FileName))
            {
                object obj = Ios.Serialization.Serializer.ReadObject(_FileName);

                if (obj != null && obj is T)
                    _Configuration = (T)obj;
                else
                    _Configuration = new T();
            }
            else
                _Configuration = new T();
        }

        public void WriteConfig()
        {
            if (!string.IsNullOrWhiteSpace(_FileName) && _Configuration != null)
            {
                Ios.Serialization.Serializer.WriteObject(_FileName, _Configuration);
            }
        }
    }
}
```

Usage:
You will need to ensure that the class you need to serialize provides a public, parameterless constructor. (See the constraint in ConfigurationProvider class)

Class to store settings...

```
public class SomeConfigClass
{
    public SomeConfigClass()
    {
        //do something here
    }

    public string StringProp { get; set; }

    public int IntProp { get; set; }

    public object ObjectProp { get; set; }
}
```

Using the classes

```
ConfigurationProvider<SomeConfigClass> configProvider = new ConfigurationProvider<SomeConfigClas
```

At this point, configProvider.Configuration should provide an instance of SomeConfigClass. This class will either be deserialized from the xml file provided, or a new instance will be created.

You can then call configProvider.ReadConfig() if you need to reload the settings from file.
To persist the settings to file, use configProvider.WriteConfig().

Credits:
The author of this article, for showing that it's possible.
Microsoft documentation, for the serialization code.

Live Long and prosper.
Trevor Moyakhe

# re: Building a better .NET Application Configuration Class - revisited

by Zx August 16, 2013 @ 9:18am

Any idea why this doesn't work in Debug mode?

# re: Building a better .NET Application Configuration Class - revisited

by Rick Strahl August 18, 2013 @ 10:58pm

@Zx - works for me in Debug mode. What 'doesn't work' exactly?

# re: Building a better .NET Application Configuration Class - revisited

by Mat Polutta August 23, 2013 @ 7:57am

I added your project to one of the government projects I work on, before seeing the License in the github help. Unfortunately, paying the licensing fee is a nightmare. So, I have to replace it. We have something similar that I created using Castle IoC to apply database supplied configuration settings to a Windows Service that schedules various tasks at runtime. No problem - back to work to integrate it into this project.

In this project (and others), our government client wishes to change configurations from a SQL Server data table, so that a service may be reconfigured without going through the Kafka-ish Change Configuration Management (yes - changing a config file in production takes 200 man-hours and coordinating who-knows-how-many meetings, phone calls, emails, shared-screen sessions in 3 different environments). Our client prefers to change the settings using an Everything Database Application in which he edits each key-value pair in its own table row.

Let me know if you are interested - didn't want to copy the whole class, but here is most of the code, which is based on your SqlServerConfigurationProvider:

```
    public class EdbSqlServerConfigurationProvider<TAppConfiguration> : ConfigurationProviderBas
        where TAppConfiguration : AppConfiguration, new()
    {

        private static string CreateTemplate =
@"CREATE TABLE [{0}](
    [ID] [int] IDENTITY(1,1) NOT NULL,
    [SectionKey] [int] NOT NULL,
    [Name] [varchar](200) NOT NULL,
    [Value] [varchar](8000) NOT NULL,
    [Description] [varchar](1000) NULL,
    [created_by] [nvarchar](50) NULL,
    [created_datetime] [smalldatetime] NULL,
    [updated_by] [nvarchar](50) NULL,
    [updated_datetime] [smalldatetime] NULL,
 CONSTRAINT [PK_{0}] PRIMARY KEY CLUSTERED
(
    [ID] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  =
) ON [PRIMARY]";
...
```

```
        public override T Read<T>()
        {
            using (SqlDataAccess data = new SqlDataAccess(ConnectionString, ProviderName))
            {
                string sql = "select [ID], [Name], [Value] from [" + Tablename + "] where Sectio
...

                List<string> items = new List<string>();
                string itemTemplate = "<{0}>{1}</{0}>";
                while (reader.Read())
                {
                    items.Add(string.Format(itemTemplate, (string)reader["Name"], (string)reader
                    /// TODO: Validate - remove if element is not allowed, since deserialization
                }
                /// TODO: Add missing Properties and set to default

                reader.Close();
                data.CloseConnection();

                //Convert Dictionary to XML
                string xmlTemplate =
                    "<?xml version=\"1.0\" encoding=\"utf-8\"?>" +
                    "<{0} xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\" xmlns:xsd=\"ht
                string itemContent = string.Join("", items.ToArray());

                T instance = null;
                //Empty Content List indicates nothing to serialize to XML, so treat as empty.
                if (string.IsNullOrEmpty(itemContent))
                {
                    IsEmpty = true;
                    instance = new T();
                    instance.Provider = this;
                }
                else // Has content, so serialize to XML.
                {
                    string xmlConfig = null;
                    xmlConfig = string.Format(xmlTemplate, typeof(T).Name, itemContent);
                    instance = Read<T>(xmlConfig);
                }
                return instance;
            }
        }

        public override bool Read(AppConfiguration config)
        {
            TAppConfiguration newConfig = Read<TAppConfiguration>();
            if (newConfig == null)
                return false;

            ///TODO: Rick had this backwards - config is the source, newConfig is the target
            //DataUtils.CopyObjectData(newConfig, config,"Provider,ErrorMessage");
            DataUtils.CopyObjectData(config, newConfig, "Provider,ErrorMessage");

            if (CreatedTable || IsEmpty)
            {
                CreatedTable = false;
                IsEmpty = false;
                //Seed the Section with values from the config data
                Write(newConfig);
            }
            return true;
        }

...

        public override bool Write(AppConfiguration config)
        {
            Dictionary<string, string> nameValuePairs = GetNameValuePairsDictionary(config);

            // Set a flag for missing fields
            // If we have any we'll need to write them out into .config
            //bool missingFields = false;

            string fieldsToEncrypt = "," + PropertiesToEncrypt.ToLower() + ",";

            SqlDataAccess data = new SqlDataAccess(ConnectionString, ProviderName);
            string sqlTemplate =
@"declare @pairCount int
select @pairCount=count(ID) from [Configuration] where SectionKey=@SectionKey AND [Name]=@Name
if (@pairCount = 0)
begin
    Insert [{0}] (SectionKey,[Name],[Value],[created_by],[created_datetime]) values (@SectionKey
end
else
begin
    Update [{0}] set [Value]=@ConfigData, [updated_by]=@UpdatedBy, [updated_datetime]=@UpdatedDa
end";

            foreach (var item in nameValuePairs)
            {
                string sql = string.Format(sqlTemplate, Tablename);
                //, Key, item.Key, "code", DateTime.Now.ToString("s"));
                var parameters = new[] {
                    new SqlParameter() { ParameterName="@SectionKey", SqlDbType = System.Data.Sq
                    new SqlParameter() { ParameterName="@Name", SqlDbType = System.Data.SqlDbTyp
                    new SqlParameter() { ParameterName="@UpdatedBy", SqlDbType = System.Data.Sql
                    new SqlParameter() { ParameterName="@UpdatedDatetime", SqlDbType = System.Da
                    new SqlParameter() { ParameterName="@ConfigData", SqlDbType = System.Data.Sq
                };

                int result = 0;
                try
                {
                    //Execute sqlTemplate
                    result = data.ExecuteNonQuery(sql, parameters);

...

        private Dictionary<string, string> GetNameValuePairsDictionary(AppConfiguration config)
        {
            Dictionary<string, string> pairs = new Dictionary<string, string>();
            string fieldsToEncrypt = "," + PropertiesToEncrypt.ToLower() + ",";
            Type typeWebConfig = config.GetType();
            MemberInfo[] Fields = typeWebConfig.GetMembers(BindingFlags.Public | BindingFlags.In

            // Loop through all fields and properties
            foreach (MemberInfo Member in Fields)
            {
                string typeName = null;

                FieldInfo field = null;
                PropertyInfo property = null;
                Type fieldType = null;
                string value = null;
                object rawValue = null;

                if (Member.MemberType == MemberTypes.Field)
                {
                    field = (FieldInfo)Member;
                    fieldType = field.FieldType;
                    typeName = fieldType.Name.ToLower();
                    rawValue = field.GetValue(config);
                    if (rawValue != null)
                        value = rawValue.ToString();
                }
                else if (Member.MemberType == MemberTypes.Property)
                {
                    property = (PropertyInfo)Member;
                    fieldType = property.PropertyType;
                    typeName = fieldType.Name.ToLower();
                    rawValue = property.GetValue(config, null);
                    if (rawValue != null)
                        value = rawValue.ToString();
                }
                else
                    continue;

                string fieldName = Member.Name;
                    //.ToLower();

                // Error Message is an internal public property
                string fieldNameLowered = fieldName.ToLower();
                if (fieldNameLowered == "errormessage" || fieldNameLowered == "provider")
                    continue;

                if (value == null)
                {
                    value = "";
                }
                else //Validate and Coerce the Values to conform to XML Specification
                {
                    if (typeName == "boolean")
                        value = value.ToLower();
                }

                // If we're encrypting decrypt any field that are encyrpted
                if (value != string.Empty && fieldsToEncrypt.IndexOf("," + fieldName + ",") > -1
                    value = Encryption.EncryptString(value, EncryptionKey);

                pairs.Add(fieldName, value);
            }

            return pairs;
        }
    }
```

# re: Building a better .NET Application Configuration Class - revisited

by Rick Strahl   August 24, 2013 @ 7:05pm

@Mat - I recently changed the licensing to be plain MIT License with an optional availability of a commercial license. The commercial license is not required though and is only for those that need official support and or are required to have licensed software.

# re: Building a better .NET Application Configuration Class - revisited

by Jes Singh   January 30, 2014 @ 7:37pm

Rick,

I use the assembly in a .net 4 VB.NET project and it always encrypts the keys by default.
Your code is as-is downloaded and compiled into a assembly for which I added the assembly reference.

This is the inherited call in VB.NET

Public Class MyConfiguration : Inherits Westwind.Utilities.Configuration.AppConfiguration

Private _FirstName As String
Private _LastName As String

Public Property FirstName As String
Get
Return _FirstName
End Get
Set(value As String)
_FirstName = value
End Set
End Property

Public Property LastName As String
Get
Return _LastName
End Get
Set(value As String)
_LastName = value
End Set
End Property

Private Shared _this As MyConfiguration

Public Sub New()
FirstName = "John"
LastName = "Doe"

```
End Sub


Public Shared Function _Instance() As MyConfiguration
If _this Is Nothing Then
_this = New MyConfiguration
_this.Initialize(Nothing, "NewSection", Nothing)
End If
Return _this
End Function
End Class
```

And then in my main form of the application I call

MyConfiguration._Instance()

And the result I see in the existing app.config file is

```
.....
.....
....
<x77dcf153982bff74>
<add key="x7c540434d0d79540" value="John" />
<add key="x34580293846923b7" value="Doe" />
</x77dcf153982bff74>
</configuration>
```

Note the encrypted keys and section name

# re: Building a better .NET Application Configuration Class - revisited

by Rick Strahl   January 31, 2014 @ 1:44am

@Jes - that's not coming from the AppConfiguration class - it doesn't encode the keys nor elements only the values and your values aren't encoded.

Maybe you have .NET level encryption enabled on your config file?

More info here:
http://msdn.microsoft.com/library/dtkwfdky.aspx

# re: Building a better .NET Application Configuration Class - revisited

by Jes Singh   January 31, 2014 @ 4:10am

@Rick, no the other keys in the .config are not affected. Only the ones that are written by your code. Any suggestion what am I doing wrong here!

# re: Building a better .NET Application Configuration Class - revisited

by Rick Strahl   January 31, 2014 @ 1:55pm

@Jes - like I said we're not encrypting any of the keys or elements, so this has to be something else. All the config class does is read the XML file (the .config) from disk, add or update the keys, then writes it back.

I've never seen or heard anything like this before. You have the source code - you can step through to see and see what values are written.

# re: Building a better .NET Application Configuration Class - revisited

by Jes Singh   January 31, 2014 @ 2:52pm

@Rick found the issue.
We obfuscate the .NET exe for deployment after we build it. SO that was the one which was renaming the Properties. And I guess because your code picks up the properties during runtime, so it was writing the obfuscated names.

We had to instruct the obfuscat'or to not obfuscate the MyConfigurationManager class.

Thank you!

# re: Building a better .NET Application Configuration Class - revisited

by Rick Strahl January 31, 2014 @ 3:39pm

@Jes - Ah that's a good one! Yikes that would cause problems with just about any code trying to reflect over properties...

Add a Comment