















Strongly typed, code-first configuration classes for .NET applications
<http://west-wind.com/westwind.applicationconfiguration/>

51 commits 3 branches 3 releases 1 contributor

branch: master Westwind.ApplicationConfiguration / +

Update assembly version.		
	RickStrahl authored 4 days ago	latest commit 6cde6e6930 
	.nuget Merge in configuration changes from Westwind.Utilities.	7 days ago
	Westwind.ApplicationConfig... Merge in configuration changes from Westwind.Utilities.	7 days ago
	Westwind.Utilities.Configurat... Fix bug with appSettings read operation when using configuration sect...	4 days ago
	Westwind.Utilities.Configurat... Update assembly version.	4 days ago
	lib Update assembly version.	4 days ago
	packages Merge in configuration changes from Westwind.Utilities.	7 days ago
	.gitignore Merged JSON Serialization features into Master	6 months ago
	AppConfiguration.png Update assembly and project documentation.	9 months ago
	Changelog.md Documentation Updates	7 days ago
	Readme.md Documentation Updates	7 days ago
	Westwind.ApplicationConfig... Merge in configuration changes from Westwind.Utilities.	7 days ago
	west wind application config... Version update and minor documentation tweaks.	8 months ago

Readme.md

West Wind Application Configuration

Strongly typed configuration classes for .NET Applications

.NET library that provides for code-first creation of configuration settings using strongly typed .NET classes. Create a class, add properties for configuration values, instantiate and automatically read and optionally write your strongly typed configuration values from the configuration store.

Configuration data can be stored in various configuration formats and can auto-sync from store to class and vice versa. You can use standard .NET config files (default), including custom sections and external files, or other custom stores including plain XML files, strings or a database. It's also easy to create your own configuration providers to store config data in some other format.

Unlike the built-in .NET Configuration Manager, the classes you create are strongly typed and automatically convert config store values to strong types. You can also write configuration data from the class to the configuration store and if a store or store value doesn't exist it's automatically created (provided permissions allow it).

This library provides:

- Strongly typed classes for configuration values
- Automatic type conversion from configuration store to class properties
- Default values for configuration values (never worry about read failures)
- Optional encryption of individual keys
- Automatic creation of configuration store if it doesn't exist
- Automatic addition of values that don't exist in configuration store (your store stays in sync with your class automatically)
- Support for multiple configuration objects simultaneously

<> Code

Issues 1


Pull Requests 1

Pulse

Graphs

Network

HTTPS clone URL

https://github.com 

You can clone with HTTPS or Subversion. ⓘ

Clone in Desktop

Download ZIP

- Works in any kind of .NET application: Web, Desktop, Console, Service...

Provided Configuration Storage formats:

- Standard .NET .config files
 - AppSettings
 - Custom Configuration Sections
 - External Configuration Files
- Standalone, plain XML files
- Json files (requires JSON.NET)
- Strings
- Sql Server Tables
- Customizable to create your own Configuration Providers

More detailed documentation is available as part of the Westwind.WebToolkit here:

- [Main Product Page](#)
- [Get it from NuGet](#)
- [Building a better .NET Application Configuration Class - Blog Entry](#)
- [Change Log](#)
- [Westwind.ApplicationConfiguration Documentation](#)
- [Westwind.ApplicationConfiguration Class Reference](#)
- [Support Forum](#)
- [License \(MIT and commercial option\)](#)

West Wind Application Configuration is also part of: [West Wind Toolkit's Westwind.Utilities library](#)

Getting Started

To create configurations, simply create a class that holds properties for each configuration value. The class maps configuration values stored in the configuration file. Values are stored as string in the configuration store, but are accessed as strongly typed values in your class.

The library allows for reading and writing of configuration data (assuming you have permissions) and assignment of default values if values don't exist in the configuration store. Your class ALWAYS has default values.

To use you simply create a class derived from `AppConfiguration` and add properties:

```
public class ApplicationConfiguration : Westwind.Utilities.Configuration.AppConfiguration
{
    public string ApplicationTitle { get; set; }
    public string ConnectionString {get; set; }
    public DebugModes DebugMode {get; set; } // enum
    public int MaxPageItems {get; set; } // number

    public ApplicationConfiguration()
    {
        ApplicationTitle = "West Wind Web Toolkit Sample";
        DebugMode = DebugModes.ApplicationErrorMessage;
        MaxPageItems = 20;
    }
}
```

Each property maps to a configuration store setting.

To use the class you simply create an instance and call `Initialize()` then read configuration values that were read from the configuration store, or default values if store values don't exist:

```
// Create an instance - typically you'd use a static singleton instance
var config = new ApplicationConfiguration();
config.Initialize();

// Now read values retrieved from web.config/ApplicationConfiguration Section
// If write access is available, the section is auto-created if it doesn't exist
string title = config.ApplicationTitle;
DebugModes modes = config.DebugMode;

// You can also update values
config.MaxPageItems = 15;
config.DebugMode = DebugModes.ApplicationErrorMessage;

// Save values to configuration store if permissions allow
```

```
config.Write();
```

The above instantiation works, but typically in an application you'll want to reuse the configuration object without having to reinstantiate it each time.

More effectively, create a static instance in application scope and initialize it once, then re-use everywhere in your application or component:

```
public class App
{
    // static property on any class in your app or component
    public static ApplicationConfiguration Configuration { get; set; }

    // static constructor ensures this code runs only once
    // the first time any static property is accessed
    static App()
    {
        /// Load the properties from the Config store
        Configuration = new ApplicationConfiguration();
        Configuration.Initialize();
    }
}
```

You can then use the configuration class anywhere, globally without recreating:

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

Once instantiated you can also use Read() and Write() to re-read or write values to the underlying configuration store.

Note that you can easily create multiple application configuration classes, which is great for complex apps that need to categorize configuration, or for self-contained components that need to handle their own internal configuration settings.

Configuration Providers

By default configuration information is stored in standard config files. When calling the stock Initialize() method with no parameters, you get configuration settings stored in an app/web.config file with a section that matches the class name.

To customize the configuration provider you can create an instance and pass in one of the providers with customizations applied:

```
public static App
{
    App.Config = new AutoConfigFileConfiguration();

    // Create a customized provider to set provider options
    // Note: several different providers are available
    var provider = new ConfigurationFileConfigurationProvider<AutoConfigFileConfiguration>()
    {
        ConfigurationSection = "CustomConfiguration",
        EncryptionKey = "seekrit123",
        PropertiesToEncrypt = "MailServer,MailServerPassword"
    };

    App.Config.Initialize(provider);
}
```

Alternately you can abstract the above logic directly into your configuration class by overriding the OnInitialize() method to provide your default initialization logic which keeps all configuration related logic in one place.

The following creates a new configuration using the Database provider to store the configuration information:

```
public class DatabaseConfiguration : Westwind.Utilities.Configuration.AppConfiguration
{
    public DatabaseConfiguration()
    {
        ApplicationName = "Configuration Tests";
        DebugMode = DebugModes.Default;
        MaxDisplayListItems = 15;
        SendAdminEmailConfirmations = false;
        Password = "seekrit";
    }
}
```

```

AppConnectionString = "server=.;database=hosers;uid=bozo;pwd=seekrit;";
}

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; }

/// <summary>
/// Override this method to create the custom default provider - in this case a database
/// provider with a few options. Config data can be passed in for connectionString and table
/// </summary>
protected override IConfigurationProvider OnCreateDefaultProvider(string sectionName, object configData)
{
    // default connect values
    string connectionString = "LocalDatabaseConnection";
    string tableName = "ConfigurationData";

    // ConfigData: new { ConnectionString = "...", Tablename = "..." }
    if (configData != null)
    {
        dynamic data = configData;
        connectionString = data.ConnectionString;
        tableName = data.Tablename;
    }

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

    return provider;
}

/// <summary>
/// Optional - simplify Initialize() for database provider default
/// </summary>
public void Initialize(string connectionString, string tableName = null)
{
    base.Initialize(configData: new { ConnectionString = connectionString, Tablename = tableName });
}
}

```

You can override the OnCreateDefaultProfile() method and configure a provider, or the slightly higher level OnInitialize() which creates a provider and then reads the content. Either one allows customization of the default Initialization() when Initialize is called with no explicit Provider.

Multiple Configuration Stores

To create multiple configuration stores simply create multiple classes and access each class individually. A single app can easily have multiple configuration classes to separate distinct sections or tasks within an application. Ideally you store the configuration objects on a global static instance like this:

```

App.Configuration = new MyApplicationConfiguration();
App.Configuration.Initialize();

App.AdminConfiguration = new AdminConfiguration();
App.AdminConfiguration.Initialize();

```

This allows for nice compartmentalization of configuration settings and also for multiple components/assemblies to have their own private configuration settings.

Complex Type Serialization

If you're using a configuration store other than .NET .config files you can easily create complex hierarchical types as these types are serialized using either XML or JSON serialization - anything those serialization formats

support you can write out to.

If you're using the .config format you're limited to key value pairs in configuration sections, so your configuration objects have to preferably be single level without child types.

However, the ConfigurationFileConfigurationProvider **does** support serialization of some complex types and IList-based lists.

Complex Types using ToString()/FromString()

One of the easiest way to serialize configuration child objects is to create a custom type that implements a ToString() and static FromString() method that effectively provides two-way serialization.

Here's an example:

```
public class LicenseInformation
{
    public string Name { get; set; }
    public string Company { get; set; }
    public int LicenseKey { get; set; }

    public static LicenseInformation FromString(string data)
    {
        return StringSerializer.Deserialize<LicenseInformation>(data, ",");
    }

    public override string ToString()
    {
        return StringSerializer.SerializeObject(this, ",");
    }
}
```

Here a StringSerializer helper is used that basically does a string.Join()/Split() to create a serialized string with a separator of an object in the ToString() and FromString() methods. You can of course use any mechanism to create a string that represents the serialized object data but StringSerializer is a quick and easy way to do so.

If you now add this to a configuration object like this:

```
public class CustomConfigFileConfiguration : Westwind.Utilities.Configuration.AppConfiguration
{
    public string ApplicationName { get; set; }
    public LicenseInformation ComplexType { get; set; }

    public CustomConfigFileConfiguration()
    {
        ApplicationName = "Configuration Tests";
        ComplexType = new LicenseInformation()
        {
            Name = "Rick",
            Company = "West Wind",
            LicenseKey = 10
        };
    }
}
```

you get a serialized configuration section like this:

```
<CustomConfig>
  <add name="ApplicationName" value="Configuration Tests" />
  <add key="ComplexType" value="Rick,West Wind,10" />
</CustomConfig>
```

TypeConverters

You can also use custom type converters on the object to serialize which is a bit more involved, but

You can find out more here: http://west-wind.com/westwindtoolkit/docs/_1cx0ymket.htm

IList Types

In addition to complex objects you can also serialize IList values or objects in .config files. List values are enumerated and read/written with indexes such as ItemList1, ItemList2, ItemList3 etc. with each item representing either a single value such as string, or a complex object that supports either the ToString/FromString() or TypeConverter serialization discussed in the previous section.

```

public class CustomConfigFileConfiguration : Westwind.Utilities.Configuration.AppConfiguration
{
    public string ApplicationName { get; set; }
    public List<string> ServerList { get; set; }

    public CustomConfigFileConfiguration()
    {
        ApplicationName = "Configuration Tests";
        ServerList = new List<string>()
        {
            "DevServer",
            "Maximus",
            "Tempest"
        };
    }
}

```

produces the following .config:

```

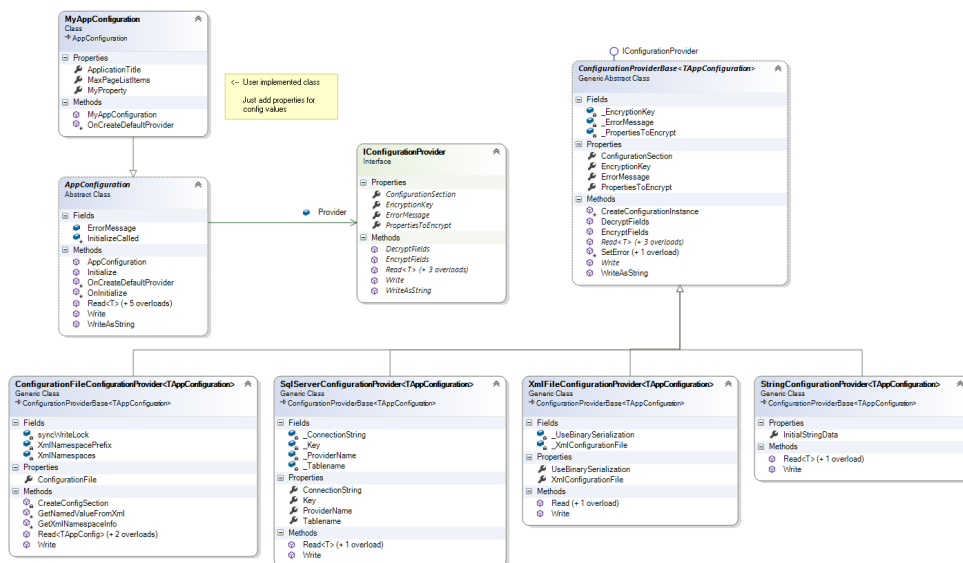
<CustomConfigFileConfiguration>
  <add key="ApplicationName" value="Configuration Tests" />
  <add key="ServerList1" value="DevServer" />
  <add key="ServerList2" value="Maximus" />
  <add key="ServerList3" value="Tempest" />
</CustomConfigFileConfiguration>

```

Class Structure

This library consists of the main AppConfiguration class plus provider logic. Providers are based on a IConfigurationProvider interface with a ConfigurationProviderBase class providing base functionality.

Here's the complete class layout:



Many More Options

Many more configuration options are available. Please check the full documentation for more information.

- [Westwind.ApplicationConfiguration Documentation](#)
- [Westwind.ApplicationConfiguration Class Reference](#)

