**[Writing a Custom ConfigurationSection to handle a Collection](http://www.abhisheksur.com/2011/09/writing-custom-configurationsection-to.html)**

Posted by Abhishek Sur on Monday, September 12, 2011

Labels: [.NET](http://www.abhisheksur.com/search/label/.NET), [.NET 4.0](http://www.abhisheksur.com/search/label/.NET%204.0), [architecture](http://www.abhisheksur.com/search/label/architecture), [beyondrelational](http://www.abhisheksur.com/search/label/beyondrelational), [C#](http://www.abhisheksur.com/search/label/C%23), [CodeProject](http://www.abhisheksur.com/search/label/CodeProject), [Configuration](http://www.abhisheksur.com/search/label/Configuration), [WPF](http://www.abhisheksur.com/search/label/WPF), [XAML](http://www.abhisheksur.com/search/label/XAML) [3 Comments](http://www.abhisheksur.com/2011/09/writing-custom-configurationsection-to.html" \l "disqus_thread)

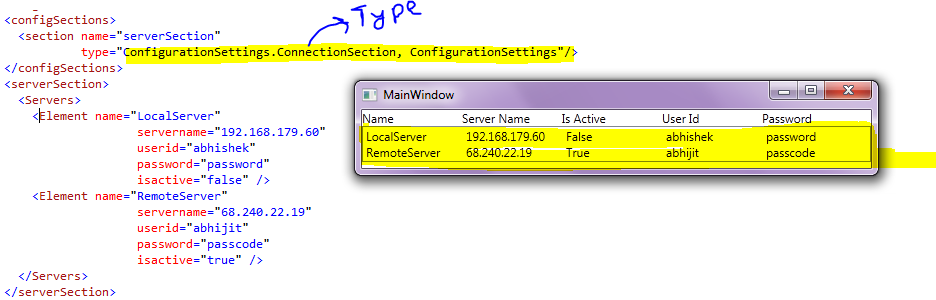
Configuration is one of the major thing that you need to keep in mind while building any application. Either its an Windows Forms application or a Web site, configuration file is always needed. We write all the configuration that are needed to be changed after it is being deployed in confugration files. It is an XML File which lists all the configuration blocks itself and also allows you to define your own custom configuration sections yourself. Today I am building my own custom configuration section and show how easily you can build yourself.

While dealing with Configurations, there are two things that you need to address

1. ConfigurationSection
2. ConfigurationElement
3. ConfigurationElementCollection

For most of the simple configurations, it is pretty much common to use these two classes, but when you need more complex configuration block, like appsettings which actually puts a Collection of ConfigurationElements, you might need to use ConfigurationElementCollection to hold the collection of ConfigurationElement.

Here in the article I will build one Configuration similar to appsettings so that it would be easier to build one for your own application.

[](http://1.bp.blogspot.com/-8O1BOMChjqs/Tm1Ba6QMSQI/AAAAAAAAD3I/5UfYipTZK_4/s1600/configsection.PNG)

**ConfgurationSection** maps to the whole configuration, reading through the whole configuration you define in your config file. It has its own serialization and deserialization technique internally to open and close one specific ConfigurationSection when you want. For each Configuration file, it searches the Type of the ConfigSection, (in out case it is ConnectionSection which is within the assembly Configurationsettings). The name of the section indicates the Tag which you use for your ConfigurationSection.  
  
**ConfigurationElement** maps with the individual element of the Section. It generally points to the entire XML configuration Tag that we use for the configuration. It is the serialized object with all the information about your configuration. When used as a Collection (as in our case) the ConfigurationElement maps to individual configuration blocks (in our case it is Element).  
  
**ConfigurationElementCollection** : Each ConfigurationSection can have a collection of ConfigurationElement. In our case Servers represents the collection of Element.  
  
Now lets see how we code to retrieve data from this Configuration.  
  
Implementation of ConfigurationElement  
  
On the lowest level, each Configuration block in the collection represents one .NET object. Lets think that the enter tag here is converted to a .NET object and vice-versa. So how your .NET object would look like ?

public class Element : ConfigurationElement

{

[ConfigurationProperty("name", DefaultValue = "", IsKey = true, IsRequired = true)]

public string name

{

get { return (string)base["name"]; }

set { base["name"] = value; }

}

[ConfigurationProperty("servername", DefaultValue = "", IsKey = false, IsRequired = true)]

public string servername

{

get { return (string)base["servername"]; }

set { base["servername"] = value; }

}

[ConfigurationProperty("isactive", DefaultValue = "true", IsKey = false, IsRequired = false)]

public bool isactive

{

get { return (bool)base["isactive"]; }

set { base["isactive"] = value; }

}

[ConfigurationProperty("userid", DefaultValue = "abhi", IsKey = false, IsRequired = false)]

public string userid

{

get { return (string)base["userid"]; }

set { base["userid"] = value; }

}

[ConfigurationProperty("password", DefaultValue = "password", IsKey = false, IsRequired = false)]

public string password

{

get { return (string)base["password"]; }

set { base["password"] = value; }

}

}

Here the class Element represents exactly as we define our Configuration element in the collection. Our Element in configuration looks like :

<Element name="RemoteServer"

servername="68.240.22.19"

userid="abhijit"

password="passcode"

isactive="true" />

So our idea to create a type that maps to this.  Remember XML is case - sensitive.  
Hence we create a class. We map the individual Properties with ConfigurationProperty. You can see each individual property has few attributes that you can define. Such as,  DefaultValue would be used whenever the attribute is not present in actual configuration; IsRequired indicate that the property is mandatory etc.  
  
You should always consider one of the property of the **ConfigurationElement** as Key field. Key field represents the unique identifier for the object.  
  
Implementation of ConfigurationElementCollection  
  
Now as we have a collection of Element in our configuration block, we need to wrap the individual Element inside a ConfigurationElementCollection. This class is an abstract implementation of a ConfigCollection. It has few members that we need to address. Lets see how do we implement it.

[ConfigurationCollection(typeof(Element))]

public class ServerAppearanceCollection : ConfigurationElementCollection

{

internal const string PropertyName = "Element";

public override ConfigurationElementCollectionType CollectionType

{

get

{

return ConfigurationElementCollectionType.BasicMapAlternate;

}

}

protected override string ElementName

{

get

{

return PropertyName;

}

}

protected override bool IsElementName(string elementName)

{

return elementName.Equals(PropertyName, StringComparison.InvariantCultureIgnoreCase);

}

public override bool IsReadOnly()

{

return false;

}

protected override ConfigurationElement CreateNewElement()

{

return new Element();

}

protected override object GetElementKey(ConfigurationElement element)

{

return ((Element)(element)).name;

}

public Element this[int idx]

{

get

{

return (Element)BaseGet(idx);

}

}

}

First and foremost, we need to annotate the class using ConfigurationCollection attribute. Generally, each configurationSection is read by the .NET configuration reader using Reflection. It chooses appropriate classes based on the Attribute. Hence it is important to define this for each configuration.  
  
Here the first thing that you notice is we define an Indexer for the class and return BaseGet(index). The BaseGet actually reads the configuration collection and use Reflection to create object of the ConfigurationElement.  
  
Another important thing that you need to address in this implementation is CreateNewElement. You need to return the actual implementation of your ConfigurationElement here. The ElementName represents the name of the Tag that you use in Config file.  
  
Implementation of ConfigurationSection  
  
Finally we need to actually map the collection to a single ConfigurationSection. Lets see the implementation first :

public class ConnectionSection : ConfigurationSection

{

[ConfigurationProperty("Servers")]

public ServerAppearanceCollection ServerElement

{

get { return ((ServerAppearanceCollection)(base["Servers"])); }

set { base["Servers"] = value; }

}

}

Here basically we create an object of our Collection and return the whole XML block using base["name of the block"]. You must notice that we wrapped the whole configuration inside one XML tag called Servers. We use this to indicate the whole collection.  
  
As a Side Note  
  
Yes, of course you can read configuration directly without using existing APIs available. This article demonstrates how to handle complex configuration blocks for your application.  
  
If you are looking for something simplier than this, check the following links :  
[3 Easy steps to create Configuration](http://haacked.com/archive/2007/03/12/custom-configuration-sections-in-3-easy-steps.aspx)  
  
In much simple scenario you dont need to use ConfigurationSectionCollection rather you can simply use ConfigurationSection itself to handle your entire configuration.  
  
  
To Get information about Configuration  
  
To  retrieve the information of the configuration lets define a class and enumerate all the Element objects that can be found from Configuration.

public class ConfigSettings

{

public ConnectionSection ServerAppearanceConfiguration

{

get

{

return (ConnectionSection)ConfigurationManager.GetSection("serverSection");

}

}

public ServerAppearanceCollection ServerApperances

{

get

{

return this.ServerAppearanceConfiguration.ServerElement;

}

}

public IEnumerable<Element> ServerElements

{

get

{

foreach (Element selement in this.ServerApperances)

{

if (selement != null)

yield return selement;

}

}

}

}

Here we use GetSection to get the specific section that we have created, and we can easily cast the object returned by GetSection to ConnectionSection. The GetSection automatically creates the instance of the class (if everything is alright) . Now as our section also contains a Collection of custom settings, we have yield the elements from ServerAppearanceCollection.  
  
[Download Source application](https://skydrive.live.com/embedicon.aspx/.Public?cid=bafa39a62a57009c&sc=documents)  
  
Conclusion  
  
Use of Custom Section for app.config is generally very handy. The System.Configuration api exposes a number of good classes that can handle configuration for you. Here I have shown how to create a section that might act similar to AppSettings or any collection oriented settings.