

4/20/2009



MOBILECONTRIB

MOBILE CLIENT SOFTWARE FACTORY: CONFIGURATION GETTING STARTED GUIDE

Configuration Getting Started | Nick Randolph

Contents

Introduction 3

Configuration 1: Basic Configuration Section 4

Configuration 2: Nested Configuration Information..... 9

Configuration 3: Encrypted Sections..... 13

Summary 15

Introduction

In building an application it may be necessary to specify some initial setup, or configuration, information which will be used to determine application behaviour. This may include connection strings in order to connect to a database or service endpoint information. Prior to .NET there were a number of different ways to do this, such as the old school .ini files or the use of the registry. With the .NET Framework re-emerged the idea of this information being stored in a file that accompanied the application. .NET configuration files are well structured and are usually accessed via the System.Configuration namespace using strongly typed classes to represent the information held in the configuration files.

Unlike the full .NET Framework, the .NET Compact Framework doesn't currently support the use of configuration files. The Configuration block that is part of the Mobile Client Software Factory provides a subset of the functionality of the System.Configuration namespace found in the full framework and has been specifically designed for use with the .NET Compact Framework.

In this Getting Started guide you will walk through three worked examples showing how to use the Configuration application block within your application. Whilst the samples here are simple, in order to illustrate the important concepts, the Configuration application block can easily be retrofitted into any existing .NET Compact Framework application.

Configuration 1: Basic Configuration Section

In this section you will learn how to reference the Configuration application block and use it to access settings contained in a simple configuration file.

Referencing the Configuration Application Block

The first step to using configuration files within your .NET Compact Framework application is to reference the Configuration application block. You can do this by right-clicking the Solution node of the Solution Explorer of a new, or existing .NET Compact Framework application, and selecting Add → Existing Project... Browse to the location of the Configuration application block and select the project file, Mobile.Configuration.csproj.

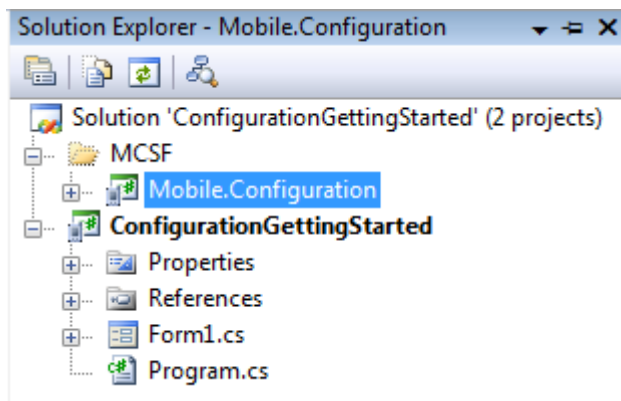


Figure 1 - Configuration Application Block in Solution Explorer

In Figure 1 you can see that the Mobile.Configuration project has been added to a solution folder called MCSF. Solution folders are a useful way to keep your solution tidy and easy to navigate and can easily be created by selecting Add → New Solution Folder from the right-click menu from the Solution Explorer.

In order to use the Configuration block you also need to add a reference to the Mobile.Configuration project to your application. Right-click on your application within the Solution Explorer and select Add Reference... Select the Mobile.Configuration project from the Projects tab.

Creating a Configuration File

As the .NET Compact Framework doesn't support the use of configuration files, adding a configuration file to your application isn't as easy as adding one for an application based on the full .NET Framework. Instead you have to create the file based on the xml file template in the Add New Item dialog. Right-click your application project in Solution Explorer and select Add → New Item...

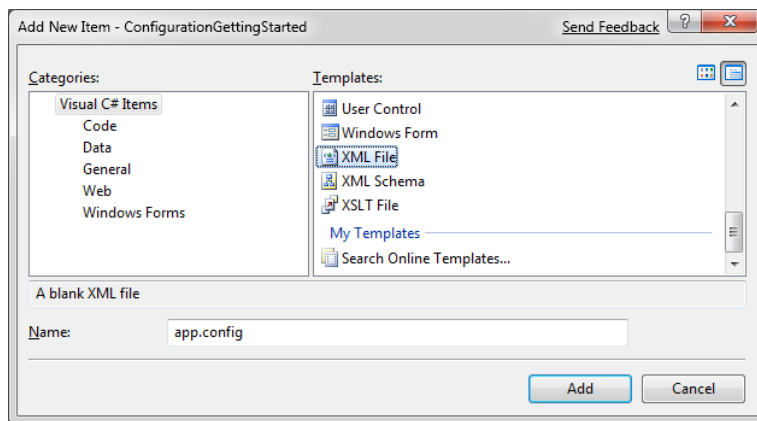
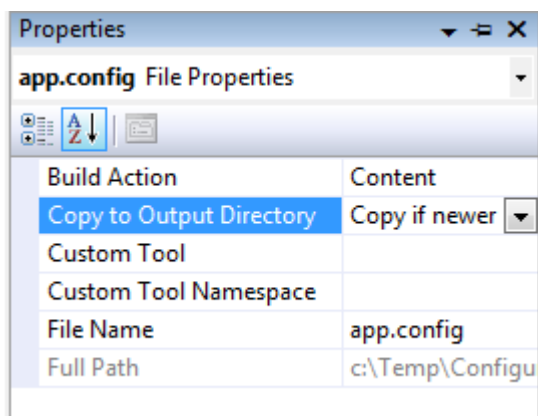


Figure 2 - Adding a Configuration File

In Figure 2 the XML File template is selected but the name of the file has been set to app.config. By default this file will have a build action set to None, which means that it will not be copied to the output directory as part of debugging your application. To make sure your application file is deployed along with your application select the app.config file within Solution Explorer and then open the Properties tool window. Change the Build Action to Content and Copy to Output Directory to Copy if newer.



The configuration file you have just created is essentially just an empty xml file with the xml declaration tag at the top of the file. To make this into a configuration file you simply need to add the configuration tags, as shown in the following code.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

</configuration>
```

You will notice as you are typing the configuration tags that intellisense will appear. This is one of the nice side effects of naming the configuration file the same as it would be for an application for the full .NET Framework. Visual Studio decides that it's an application configuration file and enables the appropriate intellisense when opening the file for editing.

Configuration Sections

Configuration files for the .NET Framework are structured xml files. However unlike most xml files where there is a set schema to which they conform it is acknowledged that the type of information

stored in configuration files will vary from application to application. As such, configuration files have an element at the beginning of the file called `configSections` within which each section of the configuration file is declared, along with a .NET class that is used to read information from that section. The rest of the configuration file is made up of these sections.

In the following example you can see that there is a section declared with the name *summary*. This section appears later in the configuration file and accessing this section will be done via the *ConfigurationGettingStarted.ApplicationSummary* class. This class is known as a configuration section handler. The summary section only contains two attributes, name and url so is a very simple example of what you can do with configuration sections.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="summary"
      type="ConfigurationGettingStarted.ApplicationSummary,
      ConfigurationGettingStarted"/>
  </configSections>

  <summary name="Getting Started with Configuration"
    url="http://mobilecontrib.codeplex.com" />
</configuration>
```

In order to access the information contained in the summary element you need to create a class called *ApplicationSummary* in the *ConfigurationGettingStarted* assembly (it also needs to be in the *ConfigurationGettingStarted* namespace). Right-click on your application and select Add → Class... Call the class *ApplicationSummary* and hit Ok to add the class to your application.

A class that is to be used to read information from a section of a configuration file needs to inherit from the *ConfigurationSection* class that is in the *Microsoft.Practices.Mobile.Configuration* namespace. As you will be referencing a number of classes from this namespace it makes sense to add this namespace with a *using* statement, to the top of your newly created class file.

```
using Microsoft.Practices.Mobile.Configuration;
```

For each of the attributes of the summary element you need to add a new property to the *ApplicationSummary* class. In the following code you will see that there are *Name* and *Url* properties and that they are annotated with the *ConfigurationProperty* attribute. The single parameter for this attribute is the name of the attribute within the summary element in the configuration file.

```
class ApplicationSummary: ConfigurationSection
{
    private const string AttributeName = "name";
    private const string AttributeUrl = "url";

    [ConfigurationProperty(AttributeName)]
    public string Name
    {
        get { return (string)this[AttributeName]; }
        set { this[AttributeName] = value; }
    }

    [ConfigurationProperty(AttributeUrl)]
    public string Url
    {
        get { return (string)this[AttributeUrl]; }
        set { this[AttributeUrl] = value; }
    }
}
```

```

        {
            get { return (string)this[AttributeUrl]; }
            set { this[AttributeUrl] = value; }
        }
    }
}

```

If you haven't worked with configuration section handlers before the properties in this code snippet may look a bit different. Essentially the ConfigurationSection class provides a dictionary into which name-value pairs are stored. This makes reading configuration files quicker and easier for the ConfigurationManager as it doesn't have to reflect over the class to set property values. For simplicity the name of each attribute on the summary element is used as the key within this dictionary. The attribute names have been added as constants to the top of the class to make them easier to change should the need ever arise.

The last thing to do is to access this section via the ConfigurationManager. This is done by calling the static *GetSection()* method on the ConfigurationManager, passing in the name of the section within the configuration file to retrieve. Based on the list of configuration section handlers specified at the top of the configuration file, the ConfigurationManager will generate a new instance of the ApplicationSummary class, and set the appropriate properties based on the values on the summary tag within the configuration file.

In the following code the Name and Url properties of the ApplicationSummary class are displayed via labels added to Form1.

```

using System;
using System.Windows.Forms;
using Microsoft.Practices.Mobile.Configuration;

namespace ConfigurationGettingStarted
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            ConfigurationManager.ConfigFileName = "app.config";
            ApplicationSummary summary =
            ConfigurationManager.GetSection("summary") as ApplicationSummary;
            this.NameLabel.Text = summary.Name;
            this.UrlLabel.Text = summary.Url;
        }
    }
}

```

There are a couple of points to be aware of here. The first is that the name of the configuration file has been explicitly set to app.config. When you use an app.config file in an application for the full .NET Framework when you build the application this file gets renamed to <application>.exe.config. However, as the .NET Compact Framework doesn't support configuration files it doesn't do this rename so the file gets deployed as app.config. The Configuration block by default looks for a file by the name of <application>.exe.config. You can either rename your app.config file, or you can specify

the `ConfigFileName` property on the `ConfigurationManager` prior to accessing any of the sections in your configuration file.

The next point to be aware of is that there is a strong tie between the name used in the call to `GetSection` (ie “summary”) and the name of the section in the configuration file. If you change one, you must remember to change the other, otherwise the `ConfigurationManager` will fail to load your configuration information.

Before we move onto a more complex configuration example it’s also important to note that whilst the `Name` and `Url` properties of the `ApplicationSummary` class have been expressed as attributes in the configuration file, they could also be expressed as sub-elements. For example the following xml would yield the same `ApplicationSummary` instance as that used earlier.

```
<summary>
  <name>Getting Started with Configuration</name>
  <url> http://mobilecontrib.codeplex.com</url>
</summary>
```


Configuration 2: Nested Configuration Information

In the previous section you saw how to create a simple configuration section that held two pieces of application specific information. For a lot of cases this may be sufficient but you may want to hold more complex or structured data in the configuration file. Take the following configuration information in which user information is recorded.

```
<user>
  <profile>
    <name>Nick Randolph</name>
    <company>Anonymous Mobile Developers</company>
    <age>35</age>
  </profile>
  <contact>
    <phone>+1 425 001 0001</phone>
    <email>temp@test.com</email>
  </contact>
</user>
```

To access this information you need to create a configuration section that consists of three different classes. The User class represents the configuration section. Within this class there are instances of both the UserProfile and the UserContact class. These two classes inherit from ConfigurationElement which is used to represent sub-elements within a configuration section.

```
class ApplicationUser : ConfigurationSection
{
    private const string AttributeProfile = "profile";
    private const string AttributeContact = "contact";

    [ConfigurationProperty(AttributeProfile)]
    public UserProfile Profile
    {
        get { return (UserProfile)this[AttributeProfile]; }
        set { this[AttributeProfile] = value; }
    }

    [ConfigurationProperty(AttributeContact)]
    public UserContact Contact
    {
        get { return (UserContact)this[AttributeContact]; }
        set { this[AttributeContact] = value; }
    }
}

class UserProfile : ConfigurationElement
{
    private const string AttributeName = "name";
    private const string AttributeCompany = "company";
    private const string AttributeAge = "age";

    [ConfigurationProperty(AttributeName)]
    public string Name
    {
        get { return (string)this[AttributeName]; }
        set { this[AttributeName] = value; }
    }

    [ConfigurationProperty(AttributeCompany)]
    public string Company
```

```

    {
        get { return (string)this[AttributeCompany]; }
        set { this[AttributeCompany] = value; }
    }

    [ConfigurationProperty(AttributeAge)]
    public int Age
    {
        get { return (int)this[AttributeAge]; }
        set { this[AttributeAge] = value; }
    }
}

class UserContact : ConfigurationElement
{
    private const string AttributePhone = "phone";
    private const string AttributeEmail = "email";

    [ConfigurationProperty(AttributePhone)]
    public string Phone
    {
        get { return (string)this[AttributePhone]; }
        set { this[AttributePhone] = value; }
    }

    [ConfigurationProperty(AttributeEmail)]
    public string Email
    {
        get { return (string)this[AttributeEmail]; }
        set { this[AttributeEmail] = value; }
    }
}

```

Another example of where you would want to use nesting of configuration information is when you have a collection or a list of information. For example you might want a name-value application settings collection or a connection strings collection – both of these are supported out of the box with configuration files for the full .NET Framework. There is a subtle difference here in the way that collections are implemented using the Configuration block that prevent you from having a collection at the section level (which is the way appSettings and connectionStrings appear in the full .NET Framework). Instead you have to create them as a sub-element of a section such as in the following xml.

```

<settings>
  <appSettings>
    <add key="CompanyWebsite" value="http://mobilecontrib.codeplex.com"/>
    <add key="SecurityKey" value="349238493"/>
  </appSettings>

  <connectionStrings>
    <add name="CompanyConnectionString"
      connectionString="Data Source=myserver;Initial Catalog=companyX"
      providerName="System.Data.SqlClient" />
  </connectionStrings>
</settings>

```

The corresponding classes for the settings configuration section and appSettings collection would be.

```
class Settings : ConfigurationSection
{
    private const string ElementAppSettings = "appSettings";
    private const string ElementConnectionStrings =
"connectionStrings";

    [ConfigurationProperty(ElementAppSettings, IsRequired =
true)]
    public AppSettingCollection AppSettings
    {
        get { return
(AppSettingCollection)this[ElementAppSettings]; }
    }

    [ConfigurationProperty(ElementConnectionStrings, IsRequired =
true)]
    public ConnectionStringCollection ConnectionStrings
    {
        get { return
(ConnectionStringCollection)this[ElementConnectionStrings]; }
    }
}

class AppSettingCollection : ConfigurationElementCollection
{
    protected override ConfigurationElement CreateNewElement()
    {
        return new Setting();
    }

    protected override object GetElementKey(ConfigurationElement
element)
    {
        Setting e = (Setting)element;

        return e.Key;
    }

    public Setting GetMenuItem(int id)
    {
        return (Setting)BaseGet(id);
    }
}

class Setting : ConfigurationElement
{
    private const string AttributeKey = "key";
    private const string AttributeValue = "value";

    [ConfigurationProperty(AttributeKey)]
    public string Key
    {
        get { return (string)this[AttributeKey]; }
        set { this[AttributeKey] = value; }
    }

    [ConfigurationProperty(AttributeValue)]
```

```
public string Value
{
    get { return (string)this[AttributeValue]; }
    set { this[AttributeValue] = value; }
}
```

Configuration 3: Custom Configuration Sections

The Configuration application block provides a simple mechanism for working with a configuration file, allowing sections to be read as structured classes. However, there are cases where the use of the ConfigurationElement and ConfigurationElementCollection doesn't give you the flexibility you require. Take the previous example where we attempted to replicate the appSettings and connectionStrings sections from the full .NET Framework configuration files. In this case the ConfigurationElementCollection could only be used if it was placed within a ConfigurationSection.

To fully implement the appSettings class you can build your own ConfigurationSection by overriding the default behaviour. As the appSettings element will still be a section in the configuration file you need to start the same way as with the previous examples by creating a class that inherits from the ConfigurationSection class.

```
class AppSettings : ConfigurationSection
{
    private Dictionary<string, string> settings = new
Dictionary<string, string>();
    protected override void
DeserializeElement(System.Xml.XmlReader xml)
    {
        XElement x = XElement.Parse(xml.ReadOuterXml());
        var pairs = from nameval in x.Elements("add")
                    select new { Key =
nameval.Attribute("key").Value, Value =
nameval.Attribute("value").Value };
        foreach (var item in pairs)
        {
            settings[item.Key] = item.Value;
        }
    }

    public string this[string key]{
        get
        {
            return settings[key];
        }
    }
}
```

In the previous examples you have used a set of properties which have been annotated with the ConfigurationProperty attribute. The ConfigurationManager uses the presence of these attributes to determine how to deserialize the configuration file. By overriding the DeserializeElement method you can provide your own deserialization implementation. In this code snippet an xlinq query is used to extract out the key-value pairs that make up the appSettings section.

The corresponding section in the configuration file would look similar to what you would see in a full .NET Framework configuration file.

```
<appSettings>
  <add key="CompanyWebsite"
value="http://mobilecontrib.codeplex.com"/>
  <add key="SecurityKey" value="349238493"/>
</appSettings>
```

Configuration 4: Encrypted Sections

Summary

Configuration files enable you to specify information that can change the way your application behaves without having to recompile your application. Although the Configuration block enables your .NET Compact Framework application to easily read from a configuration file it does have an overhead which can significantly increase the load time of your application. Use of configuration files should be considered with care and appropriate testing done to ensure your application will still perform with different configuration values.