# SmartConfig v8.0.1

With SmartConfig you can setup a strongly typed configuration within a few minutes.

## Changelog

### v8.0.1

* Updated references to Reusable.
* Where now allows you to specify an expression for the column name.
* Data stores are now available as separate packages.
* Removed the LoadFailedEvent and the TryLoad method from the Configuration. The new Reload method now throws exceptions.

The core package on NuGet: Install-Package SmartConfig contains just the core functionality. You need to install a data store to be able to use it.

* AppSettingsStore
* ConnectionStringsStore
* SqlServerStore
* RegistryStore

## Quick Start

To create a configuration you need to define a static class and decorate it with the [SmartConfig] attribute. It can contain any number of **static** properties and nested classes.

The below example show all kinds of settings that are supported by SmartConfig.

Among the simple types you can also use complex types. In order to use them you need to specify them as JSON and add the Converters attribute to the configuration where you specify the exact converters for them.

Another possibility to specify collections, especially dynamic ones is by using the [Itemized] attribute. This means that each item of the collection is stored in a separate row in the data store. Arrays are required to have an additional [index] but the number actually doesn't matter. Dictionary keys can be of any type supported by the converters (int, string, enum etc.).

[SmartConfig]  
[Converters(  
 typeof(JsonToObjectConverter<int[]>),  
 typeof(JsonToObjectConverter<Dictionary<string, int>>)  
)]  
public static class FullConfig  
{  
 public static string StringSetting { get; set; }  
  
 [Optional]  
 public static string OptionalStringSetting { get; set; } = "Waldo";  
   
 public static int[] ArraySetting1 { get; set; }  
  
 [Itemized]  
 public static int[] ArraySetting2 { get; set; }  
  
 public static Dictionary<string, int> DictionarySetting1 { get; set; }  
  
 [Itemized]  
 public static Dictionary<string, int> DictionarySetting2 { get; set; }  
  
 public static class NestedConfig  
 {  
 public static string StringSetting { get; set; }  
 }  
  
 [Ignore]  
 public static class IgnoredConfig  
 {  
 public static string StringSetting { get; set; } = "Grault";  
 }  
}

You load the configuration by specifying the source with the From method and the configuration.

Loading a configuration from an app.config is as simple as this:

Configuration.Load  
 .From(new AppSettingsStore())  
 .Select(typeof(FullConfig));

The respecitve app.config file:

<appSettings>  
  
 <add key="StringSetting" value="Foo"/>  
 <add key="ArraySetting2[0]" value="5"/>  
 <add key="ArraySetting2[1]" value="8"/>  
 <add key="DictionarySetting2[foo]" value="21"/>  
 <add key="DictionarySetting2[bar]" value="34"/>  
 <add key="NestedConfig.StringSetting" value="Bar"/>  
 <add key="IgnoredConfig.StringSetting" value="Qux"/>  
   
</appSettings>

That's all.

Using a database is not that different. You just need to specify another data store:

Configuration.Load  
 .From(new SqlServerStore("name=SmartConfigTest"))  
 .Select(typeof(FullConfig));

For this example you need to have the following table:

CREATE TABLE [dbo].[Setting](  
 [Name] [nvarchar](50) NOT NULL,  
 [Value] [nvarchar](max) NULL  
) ON [PRIMARY] TEXTIMAGE\_ON [PRIMARY]

The same applies to other data sources:

SQLiteStore

Configuration.Load  
 .From(new SQLiteStore("name=configdb"))  
 .Select(typeof(FullConfig));

RegistryStore

const string TestRegistryKey = @"Software\SmartConfig\Tests";  
Configuration.Load  
 .From(RegistryStore.CreateForCurrentUser(TestRegistryKey))  
 .Select(typeof(FullConfig));

## Attributes

With SmartConfig it is possible to use more then one identifier (Name) for a setting. The additional criteria are called attributes and allow us to further describe a setting. Not all data stores allow this. That's why we use a database in this example.

A common example is the *Environment*.

The config class itself does not change at all. You add the attribute when loading:

Configuration.Load  
 .From(new SqlServerStore("name=SmartConfigTest").Column("Environment", SqlDbType.NVarChar, 200))  
 .Where("Environment", "Qux")  
 .Select(typeof(FullConfig));

Here we've added the Where to set the *Environment* and we configured the *Environment* column by setting its data-type and length.

The respecitve table is:

CREATE TABLE [dbo].[Setting](  
 [Name] [nvarchar](50) NOT NULL,  
 [Value] [nvarchar](max) NULL,  
 [Environment] [nvarchar](50) NOT NULL  
) ON [PRIMARY] TEXTIMAGE\_ON [PRIMARY]

## Converters

SmartConfig already supports a lot of data types:

* Integral types: sbyte, byte, char, short, ushort, int, uint, long, ulong
* Floating-point types: float, double
* Other types: decimal, bool, Enum, DateTime, string
* JSON
* Colors (System.Drawing.Color as Name (Red), Hex (#FF00AA), Decimal (122, 134,90)

In most cases for complex data structures we would want to use json. To be able to parse your type you first need to register a new converter. You add it with an attribute to your config class:

[SmartConfig]  
[Converters(typeof(JsonToObjectConverter<List<Int32>>))]  
public static class FullConfig  
{  
 public static List<int> JsonArray { get; set; } // "[1, 2, 3]"  
}

Now SmartConfig knows how to deserialize the list.