

# Inversion of Control and Dependency Injection with Castle Windsor Container - Part III

In the previous article I showed how to take advantage of some of the features offered by Windsor Container to configure components and supply dependencies. You have seen how to deal with compulsory and optional dependencies, as well as how to inject simple values and component references, either individually or collected into arrays, lists and dictionaries.

In this article I conclude the discussion about the core features of Windsor Container while evolving the simple example that accompanied you through the whole series of these articles. The next article will tackle more advanced topics.

## Inversion of Control/Dependency Injection Series

- [IoC/DI - Part I](#) This article introduces Inversion of Control/Dependency Injection in a simple and affordable fashion, with a real example which evolves step by step to take advantage from IoC and DI, leveraging the features offered by an open source framework called Castle Project.
- [IoC/DI - Part II](#) In the previous article I introduced the concepts of Inversion of Control and Dependency Injection, and showed how to benefit from their use when developing a simple application. In this article I will resume the previous discussion extending the requirements of the former example to show how IoC deals with changes and what is its real potential.
- [IoC/DI - Part III](#) In this article I conclude the discussion about the core features of Windsor Container while evolving the simple example that accompanied you through the whole series of these articles. The next article will tackle more advanced topics.
- [IoC/DI - Part IV](#) In this article I introduce the missing core features I didn't tackle before. They require a little more knowledge of Windsor, which I hope I've been able to hand down so far.

### Introduction

So far I've discussed and presented some simple features available in Windsor. It exposes many more configuration options, not just concerning the way you can inject dependencies, but even how to control their behavior. As usual, I will explain the remaining set of core features and some advanced ones by introducing small incremental changes in the requirements of the sample application.

By now, the sample application is capable of downloading files via a couple of protocols - HTTP and FTP - and can easily be extended to support any other, just by following these simple steps:

- Create a component exposing the `IFileDownloader` service.
- Register it into the Windsor configuration file.
- Add it to the array/list/dictionary of accepted components as a constructor-mandatory dependency of the `HtmlTitleRetriever` class.

Just as a refresher, let's create a new implementation of the service that is able to retrieve files from the file system.

**Note:** Uris referencing files on the local file system have the Scheme property set to "file".

The snippet below shows the code for the `FileSystemFileDownloader` class.

```
01. public class FileSystemFileDownloader : IFileDownloader
02. {
03.     public string Download(Uri file)
04.     {
05.         using (StreamReader reader = new StreamReader(file.LocalPath))
06.             return reader.ReadToEnd();
07.     }
08.
09.     public bool SupportsUriScheme(Uri file)
10.     {
11.         return file.Scheme == "file";
12.     }
13. }
```

Obviously, the component has to be registered into the configuration file and added to the array of downloaders accepted by the constructor of the `HtmlTitleRetriever` class.

```
01. <component id="HtmlTitleRetriever"
02.           type="WindsorSample.HtmlTitleRetriever, WindsorSample">
03.     <parameters>
04.         <AdditionalMessage>This is the title of the document:</AdditionalMessage>
05.     <downloaders>
06.
```

```

    <array>
07.     <item>${HttpFileDownloader}</item>
08.     <item>${FtpFileDownloader}</item>
09.     <item>${FileSystemFileDownloader}</item>
10.   </array>
11. </downloaders>
12. </parameters>
13. </component>
14.
15. <component id="FileSystemFileDownloader"
16.           service="WindsorSample.IFileDownloader, WindsorSample"
17.           type="WindsorSample.FileSystemFileDownloader, WindsorSample">
18. </component>

```

Now let's change the requirements of the application and introduce new concepts.

## Configuration properties and includes

The new requirement specifies that there are three files to be downloaded; one via FTP, one via HTTP and one from the file system. These files are mostly static but shouldn't be hard coded since they may eventually change. The choice is to specify them via configuration file and edit the code of the `HtmlTitleRetriever` class to enable it to accept the Uris of the files either as a compulsory or optional dependency.

Let's think about the requirement for a while. Should you make the files an optional or mandatory dependency? Will you supply them via an array, a list or a dictionary? In this case it's just a matter of taste. If you opt for the compulsory dependency then you will supply them as a constructor parameter and you're done. Otherwise you will use a read/write property with a backing field initialized to an empty array/list/dictionary, to avoid null references in case no parameter is specified. Let's go the optional dependency way, so that the code of the class becomes the one shown below.

```

01. public class HtmlTitleRetriever
02. {
03.     private readonly IFileDownloader[] downloaders;
04.     private readonly ITitleScraper scraper;
05.
06.     private string additionalMessage = "Title: ";
07.
08.     public Uri[] Files
09.     {
10.         get { return files; }
11.         set { files = value; }
12.     }
13.
14.     private Uri[] files;
15.
16.     public string AdditionalMessage
17.     {
18.         get { return additionalMessage; }
19.         set { additionalMessage = value; }
20.     }
21.
22.     public HtmlTitleRetriever(IFileDownloader[] downloaders,
23.                               ITitleScraper scraper)
24.     {
25.         this.downloaders = downloaders;
26.         this.scraper = scraper;
27.     }
28.
29.     public string GetTitle(Uri file)
30.     {
31.         foreach(IFileDownloader downloader in downloaders)
32.             if(downloader.SupportsUriScheme(file))
33.                 return string.Concat(additionalMessage,
34.                                       scraper.Scrape(downloader.Download(file)));
35.
36.         return string.Empty;
37.     }
38.
39.     public IEnumerable<string> GetTitles()
40.     {
41.         foreach(Uri file in files)
42.             yield return GetTitle(file);
43.     }
44. }

```

Other than the new `Files` property, a new method called `GetTitles` has been added. It doesn't accept any input parameters and simply iterates through the array of files, calling the `GetTitle` method on each of them. The

rest of the code is unchanged. These changes have to be reflected in the configuration file as well. The related section becomes as follows:

```

01. <component id="HtmlTitleRetriever"
02.     type="WindsorSample.HtmlTitleRetriever, WindsorSample">
03.     <parameters>
04.         <AdditionalMessage>This is the title of the document:</AdditionalMessage>
05.         <Files>
06.             <array>
07.                 <item>http://mi.mirror.garr.it/mirrors/postfix/index.html</item>
08.                 <item>ftp://mi.mirror.garr.it/mirrors/postfix/index.html</item>
09.                 <item>file://c:\index.html</item>
10.             </array>
11.         </Files>
12.         <downloaders>
13.             <array>
14.                 <item>${HttpFileDownloader}</item>
15.                 <item>${FtpFileDownloader}</item>
16.                 <item>${FileSystemFileDownloader}</item>
17.             </array>
18.         </downloaders>
19.     </parameters>
20. </component>

```

Note that I chose a mirror of the [Garr](#) to download files because it makes them available both via HTTP and FTP. The other file is a file I created locally.

Now, a little problem arises. You can see that along with complexity and requirements, the verbosity of the configuration file increases. In particular, static information like component registration is mixed up with more variable stuff, like `AdditionalMessage` and `Files` dependencies. It may be desirable to have them separated from the other stuff, so that they can be changed easily without fiddling into the whole configuration file. Windsor offers a feature called configuration properties which fits exactly this need.

By using configuration properties you can create an indirection mechanism that lets you specify references values in another section of the configuration file. The syntax for configuration properties is `#{property name}`. This doesn't require any change in the code of the classes. The changes to the configuration file are shown in the following snippet.

```

01. <castle>
02.
03.     <properties>
04.         <files>
05.             <array>
06.                 <item>http://mi.mirror.garr.it/mirrors/postfix/index.html</item>
07.                 <item>ftp://mi.mirror.garr.it/mirrors/postfix/index.html</item>
08.                 <item>file://c:\index.html</item>
09.             </array>
10.         </files>
11.         <message>This is the title of the document:</message>
12.     </properties>
13.
14.     <components>
15.         <component id="HtmlTitleRetriever"
16.             type="WindsorSample.HtmlTitleRetriever, WindsorSample">
17.             <parameters>
18.                 <AdditionalMessage>#{message}</AdditionalMessage>
19.                 <Files>#{files}</Files>
20.                 <downloaders>
21.                     <array>
22.                         <item>${HttpFileDownloader}</item>
23.                         <item>${FtpFileDownloader}</item>
24.                         <item>${FileSystemFileDownloader}</item>
25.                     </array>
26.                 </downloaders>
27.             </parameters>
28.         </component>
29.     </components>
30. </castle>

```

A new child section called `properties` has been added inside the `castle` section. It contains the values that appeared directly in the components declarations. These values are referenced by the `HtmlTitleRetriever` component using the syntax introduced before.

Now, this wouldn't be such an advantage if you couldn't really separate the properties and components sections. If you needed to deploy the application several times, the only changes may involve the properties section, while the components declaration would remain mostly unchanged. For this purpose Windsor lets you

use several configuration files, which are inspected and merged at startup. This is achieved via includes. Includes support loading configuration files from several mediums such as the file system, a network share or an assembly resource.

The snippet below shows what the App.config file looks like when using includes. Note that the Properties.config is loaded from the file system, using a path relative to the executable of the application, while the Components.config file is loaded from the assembly.

```

01. <?xml version="1.0" encoding="utf-8" ?>
02. <configuration>
03.
04.   <configSections>
05.     <section name="castle"
06.       type="Castle.Windsor.Configuration.AppDomain.CastleSectionHandler,
07.       Castle.Windsor" />
08.   </configSections>
09.
10.   <castle>
11.     <include uri="file://Properties.config"></include>
12.     <include uri="assembly://WindsorSample/Components.config"></include>
13.   </castle>
14.
15. </configuration>

```

Below is reported the Properties.config file. Note that in order for the application to be able to retrieve it, it needs to be placed in the same directory as the app executable file. Under Visual Studio, this is accomplished by setting the file's *Copy to Output Directory* to *Always* or *Copy if newer*.

```

01. <?xml version="1.0" encoding="utf-8" ?>
02. <configuration>
03.   <properties>
04.     <files>
05.       <array>
06.         <item>http://mi.mirror.garr.it/mirrors/postfix/index.html</item>
07.         <item>ftp://mi.mirror.garr.it/mirrors/postfix/index.html</item>
08.         <!--<item>file://c:\index.html</item>-->
09.       </array>
10.     </files>
11.     <message>This is the title of the document:</message>
12.   </properties>
13. </configuration>

```

I commented out the file retrieved from the file system to avoid forcing you to create one when running the sample application. The source code is available for download.

Finally, the snippet below shows the Components.config file. Since it is retrieved as an assembly resource, its Build Action property must be set to Embedded Resource.

```

01. <?xml version="1.0" encoding="utf-8" ?>
02. <configuration>
03.   <components>
04.     <component id="HtmlTitleRetriever"
05.       type="WindsorSample.HtmlTitleRetriever, WindsorSample">
06.       <parameters>
07.         <AdditionalMessage>#{message}</AdditionalMessage>
08.         <Files>#{files}</Files>
09.         <downloaders>
10.           <array>
11.             <item>${HttpFileDownloader}</item>
12.             <item>${FtpFileDownloader}</item>
13.             <item>${FileSystemFileDownloader}</item>
14.           </array>
15.         </downloaders>
16.       </parameters>
17.     </component>
18.     <component id="StringParsingTitleScrapper"
19.       service="WindsorSample.ITitleScrapper, WindsorSample"
20.       type="WindsorSample.StringParsingTitleScrapper, WindsorSample">
21.     </component>
22.
23.     <component id="FileSystemFileDownloader"
24.       service="WindsorSample.IFileDownloader, WindsorSample"
25.       type="WindsorSample.FileSystemFileDownloader, WindsorSample">
26.     </component>
27.     <component id="HttpFileDownloader"
28.       service="WindsorSample.IFileDownloader, WindsorSample"
29.       type="WindsorSample.HttpFileDownloader, WindsorSample">

```

```
31.     </component>
32.     <component id="FtpFileDownloader"
33.               service="WindsorSample.IFileDownloader, WindsorSample"
34.               type="WindsorSample.FtpFileDownloader, WindsorSample">
35.     </component>
36. </components>
</configuration>
```

Using includes you can separate static configuration information from volatile one, and ease its lookup and editing.

## Custom type converters

If you run the sample application as is, you will run into an exception of type `ConverterException` saying "No converter registered to handle the type `System.Uri`". Windsor Container, in fact, can convert your configuration parameters from strings to the destination type as long as they are of simple type. The conversions supported out of the box by the container are listed in [this documentation page](#).

Actually, the `Files` dependency of the `HtmlTitleRetriever` component is of type `Uri[]`. Arrays are supported, but `Uri`s aren't. However, Castle provides a not very straightforward API to supply custom converters, and this requires a bit of work.

First, a class inheriting from the abstract `AbstractConverter` class needs to be created, which performs the conversion from string to `Uri`. This is trivial, as shown in the following snippet.

```
01. public class UriTypeConverter : AbstractConverter
02. {
03.     public override bool CanHandleType(Type type)
04.     {
05.         return type.IsAssignableFrom(typeof(Uri));
06.     }
07.
08.     public override object PerformConversion(string value, Type targetType)
09.     {
10.         return new Uri(value);
11.     }
12.
13.     public override object PerformConversion(IConfiguration configuration,
14.         Type targetType)
15.     {
16.         return PerformConversion(configuration.Value, targetType);
17.     }
18. }
```

Then, and here comes the tricky part, you need to inform the container about it. The suggested approach is to create a class inheriting from `WindsorContainer` and perform the type converter registration in the constructor. Here's the code of the `CustomContainer` class.

```
01. public class CustomContainer : WindsorContainer
02. {
03.     public CustomContainer(IConfigurationInterpreter interpreter)
04.     {
05.         // Register the type converter
06.         IConversionManager manager = (IConversionManager)
07.             Kernel.GetSubSystem(Castle.MicroKernel.SubSystemConstants.ConversionManagerKey);
08.
09.         manager.Add(new UriTypeConverter());
10.
11.         // Process the configuration
12.         interpreter.ProcessResource(interpreter.Source, Kernel.ConfigurationStore);
13.
14.         // Install the components
15.         Installer.SetUp(this, Kernel.ConfigurationStore);
16.     }
17. }
```

Now the sample application instantiates a `CustomContainer` instead of a `WindsorContainer` and everything works as expected.

## Decorators

`Decorator` is a design pattern formalized for the first time in the book *Design Patterns: Elements of reusable object oriented software*. If you are not familiar with it, you just need to know that it decorates an object by adding features to it, without changing the original class. IoC lets you benefit of this pattern in a very straightforward way. First, let's introduce a new requirement for the sample application. Let's talk about the parsing mechanism.

So far, you've extracted the title information from the contents of the HTML file using string functions. You may be wondering whether this is more or less performance-wise. I did, and I tried to think about another mechanism for parsing the file. This obviously entails creating a new class implementing the `ITitleScrapper` interface.

There are a lot of HTML parsers for .NET, but for this simple application you aren't going to need them. Instead, I chose to take the regular expressions route. So here's the code for the `RegexTitleScrapper` class, which hopefully will perform better than the `StringParsingTitleScrapper` class.

```
1. public class RegexTitleScrapper : ITitleScrapper
2. {
3.     public string Scrape(string fileContents)
4.     {
5.         return Regex.Match(fileContents, "<title>(?'title'.*)</title>",
6.                             RegexOptions.Singleline).Groups["title"].Value;
7.     }
8. }
```

Now you can switch from the old scraping component to the new one by registering it in the configuration file and specifying it in the configuration section of the `HtmlTitleRetriever` component.

```
01. <component id="HtmlTitleRetriever"
02.           type="WindsorSample.HtmlTitleRetriever, WindsorSample">
03.     <parameters>
04.       <AdditionalMessage>#{message}</AdditionalMessage>
05.       <Files>#{files}</Files>
06.       <downloaders>
07.         <array>
08.           <item>${HttpFileDownloader}</item>
09.           <item>${FtpFileDownloader}</item>
10.           <item>${FileSystemFileDownloader}</item>
11.         </array>
12.       </downloaders>
13.       <scrapper>${RegexTitleScrapper}</scrapper>
14.     </parameters>
15. </component>
16. <component id="StringParsingTitleScrapper"
17.           service="WindsorSample.ITitleScrapper, WindsorSample"
18.           type="WindsorSample.StringParsingTitleScrapper, WindsorSample">
19. </component>
20. <component id="RegexTitleScrapper"
21.           service="WindsorSample.ITitleScrapper, WindsorSample"
22.           type="WindsorSample.RegexTitleScrapper, WindsorSample">
23. </component>
```

How do I know that using regular expressions the parsing is faster than using strings? I'd need a benchmarking mechanism. Using the `StopWatch` class you can measure with a good precision the time taken by a certain operation. Doing this requires changing the code of both the scraping classes. Once it is determined which one is faster, you remove the benchmarking code and tell Windsor to use the faster one. But you don't want to do this, since there might be a better way. In fact there is, and it consists in applying the decorator pattern. I've created a `BenchmarkingTitleScrapperDecorator` class which does exactly this.

```
01. public class BenchmarkingTitleScrapperDecorator : ITitleScrapper
02. {
03.     private readonly ITitleScrapper inner;
04.
05.     private readonly Stopwatch watch = new Stopwatch();
06.
07.     public BenchmarkingTitleScrapperDecorator(ITitleScrapper inner)
08.     {
09.         this.inner = inner;
10.     }
11.
12.     public string Scrape(string fileContents)
13.     {
14.         watch.Start();
15.         string result = inner.Scrape(fileContents);
16.         watch.Stop();
17.
18.         Console.WriteLine("Scraping via {0} took {1} ticks to complete.",
19.                             inner.GetType(), watch.ElapsedTicks);
20.
21.         watch.Reset();
22.         return result;
23.     }
24. }
```

The class implements the `ITitleScraper` interface. Therefore, it's interchangeable with the other scrapers seen so far. The difference is that this one doesn't perform any scraping. Instead, it delegates the operation to its inner scraper, supplied as a constructor parameter. This is just the logic behind the decorator pattern. Now let's see how to take advantage of it using Windsor.

```

01. <component id="HtmlTitleRetriever"
02.         type="WindsorSample.HtmlTitleRetriever, WindsorSample">
03.     <parameters>
04.         <AdditionalMessage>#{message}</AdditionalMessage>
05.         <Files>#{files}</Files>
06.         <downloaders>
07.             <array>
08.                 <item>${HttpFileDownloader}</item>
09.                 <item>${FtpFileDownloader}</item>
10.                 <item>${FileSystemFileDownloader}</item>
11.             </array>
12.         </downloaders>
13.         <scraper>${BenchmarkingTitleScraperDecorator}</scraper>
14.     </parameters>
15. </component>
16. <component id="BenchmarkingTitleScraperDecorator"
17.         service="WindsorSample.ITitleScraper, WindsorSample"
18.         type="WindsorSample.BenchmarkingTitleScraperDecorator, WindsorSample">
19.     <parameters>
20.         <inner>${StringParsingTitleScraper}</inner>
21.     </parameters>
22. </component>
23. <component id="StringParsingTitleScraper"
24.         service="WindsorSample.ITitleScraper, WindsorSample"
25.         type="WindsorSample.StringParsingTitleScraper, WindsorSample">
26. </component>
27. <component id="RegexTitleScraper"
28.         service="WindsorSample.ITitleScraper, WindsorSample"
29.         type="WindsorSample.RegexTitleScraper, WindsorSample">
30. </component>

```

You can see that a chain has been created. `HtmlTitleRetriever` is supplied a `BenchmarkingTitleScraperDecorator`, which in turn takes a `StringParsingTitleScraper` as a constructor dependency. This simple mechanism lets you decorate the parsers with benchmarking capabilities, in order to measure their performance. Just switch to `RegexTitleScraper` to measure its performance and compare it with the other parser. Once done, supply the best scraper directly to the `HtmlTitleRetriever` component and trash away the benchmarker and the other scraper. Easy, isn't it? As a matter of fact, on my machine the string parser outperforms the regular expressions.

## Defines and ifs

Another feature which might come useful in many cases is the ability to define flags similar to the preprocessor directives you use in .NET code. Windsor can understand defines and ifs, which follow the syntax shown below in the configuration file. They can be employed to register components conditionally, or to choose which component to instantiate based on a flag. In the sample application, I will use them to switch between the `BenchmarkingTitleScraperDecorator` and a real implementation of a scraper so that it becomes easier to do benchmarking tests during debug and then go back to production mode with no benchmarking. Here's how the configuration for the main component changes to benefit of this feature.

```

01. <component id="HtmlTitleRetriever"
02.         type="WindsorSample.HtmlTitleRetriever, WindsorSample">
03.     <parameters>
04.         <AdditionalMessage>#{message}</AdditionalMessage>
05.         <Files>#{files}</Files>
06.         <downloaders>
07.             <array>
08.                 <item>${HttpFileDownloader}</item>
09.                 <item>${FtpFileDownloader}</item>
10.                 <item>${FileSystemFileDownloader}</item>
11.             </array>
12.         </downloaders>
13.         <?if DEBUG?>
14.             <scraper>${BenchmarkingTitleScraperDecorator}</scraper>
15.         <?else?>
16.             <scraper>${StringParsingTitleScraper}</scraper>
17.         <?end?>
18.     </parameters>
19. </component>
20. <component id="BenchmarkingTitleScraperDecorator"
21.         service="WindsorSample.ITitleScraper, WindsorSample"
22.         type="WindsorSample.BenchmarkingTitleScraperDecorator, WindsorSample">

```

```
24.     <parameters>
25.       <inner>${StringParsingTitleScrapper}</inner>
26.     </parameters>
27.   </component>
28.   <component id="StringParsingTitleScrapper"
29.     service="WindsorSample.ITitleScrapper, WindsorSample"
30.     type="WindsorSample.StringParsingTitleScrapper, WindsorSample">
31.   </component>
32.   <component id="RegexTitleScrapper"
33.     service="WindsorSample.ITitleScrapper, WindsorSample"
34.     type="WindsorSample.RegexTitleScrapper, WindsorSample">
35.   </component>
```

As you can guess from the code above, the `HtmlTitleRetriever` component will use the benchmarking parser when a `DEBUG` flag is defined; the string parser otherwise. I will define and undefine that flag in the main configuration file for the application, making it easy to switch from debug to production mode and vice versa.

```
1. <castle>
2.   <?define DEBUG?>
3.
4.   <include uri="file://Properties.config"></include>
5.   <include uri="assembly://WindsorSample/Components.config"></include>
6. </castle>
```

To switch mode you just need to comment out the `define`. `Defines` and `ifs` can be used with no restrictions on their location within the configuration files, with the only constrain that `ifs` can't be nested.

## Summary

In this article you've seen how Windsor lets you isolate volatile configuration parameters into a specific section of the configuration file. Then, I talked about includes and how they help separating configuration files to achieve a higher modularity during configuration and deployment. Next, you've seen how to deal with parameter types not directly supported by Windsor, by implementing custom type converters and using them by subclassing the `WindsorContainer` class. Finally, I've presented one of my favorite features whose implementation is extremely simplified by Windsor: The decorator pattern and how easy is to switch between implementations using `ifs` and `defines`. In the next article I'll talk about factories, lifecycles and lifestyles.

## References

- Martin Fowler - [Inversion of Control Containers and the Dependency Injection pattern](#) - 2004
- Hamilton Verissimo - [Introducing Castle, Part I](#) - 2004
- Oren Eini - [Inversion of Control and Dependency Injection: Working with Windsor Container](#) - 2006
- Alex Henderson - [Container Tutorials](#) - 2007
- [Castle Windsor Container documentation](#)

Original Url: <http://dotnetslackers.com/articles/designpatterns/InversionOfControlAndDependencyInjectionWithCastleWindsorContainerPart3.aspx>