

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

Since the previous article didn't show much of the features offered by Castle Windsor Container, 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.

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. Although that naive example didn't take advantage of IoC and DI as much as a real life application would, it was meaningful in order to let developers new to these concepts start thinking in the right direction.

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

In the previous article I didn't show much of the features offered by Windsor Container. This time I'm going to introduce small incremental changes to the requirements of the sample application to show how Windsor can deal with them.

So far, you've seen how you can instruct the container to retrieve and instantiate components, as well as resolve dependencies on the constructor. Here's what the signature of the constructor of the `HtmlTitleRetriever` class looked like:

```
1. | public HtmlTitleRetriever(IFileDownloader downloader,  
2. |                               ITitleScraper scraper)
```

Therefore, the `HtmlTitleRetriever` class had dependencies on two references to objects implementing `IFileDownloader` and `ITitleScraper` interfaces (services) to be satisfied in order to be instantiated. With a canonical approach, you have to create these references and then supply them to the constructor. By using Windsor you could use the configuration file and resolve them in the following way:

```
01. | <component id="HtmlTitleRetriever"  
02. |           type="WindsorSample.HtmlTitleRetriever, WindsorSample">  
03. | </component>  
04. | <component id="StringParsingTitleScraper"  
05. |           service="WindsorSample.ITitleScraper, WindsorSample"  
06. |           type="WindsorSample.StringParsingTitleScraper, WindsorSample">  
07. | </component>  
08. | <component id="HttpFileDownloader"  
09. |           service="WindsorSample.IFileDownloader, WindsorSample"  
10. |           type="WindsorSample.HttpFileDownloader, WindsorSample">  
11. | </component>
```

Constructor injection and compulsory dependencies

```
1. | IWindsorContainer container = new WindsorContainer(new XmlInterpreter());  
2. | HtmlTitleRetriever retriever = container.Resolve<HtmlTitleRetriever>();
```

When an instance of the `HtmlTitleRetriever` class was required using the snippet above, the container was able to infer that, in order to instantiate the class, it needed to supply its dependencies to the

constructor. The instantiation was successful because those dependencies were satisfied by the other two components registered in the configuration file. If just one of them wasn't registered, the container wouldn't have been able to resolve them and would have thrown a `Castle.MicroKernel.Handlers.HandlerException` exception.

As you may guess, the dependencies on the constructor are compulsory, because you can't instantiate an object if you don't supply all of them. The operation of resolving and supplying constructor dependencies is called *constructor injection* in terms of IoC, and it's one of two ways of supplying dependencies to components.

You should rely on constructor injection whenever it makes sense for a component to compulsorily require a dependency in order to work. In other cases, you might want to provide a default implementation for a service or a general dependency, thus making it optional. This concept will make sense shortly.

Setter injection and optional dependencies

In order to illustrate how to deal with optional dependencies, let's make a small change in the requirements of the sample application. The `GetTitle` method of the `HtmlTitleRetriever` class should return a string containing the title of the HTML document supplied as input parameter (as before); but now prefixed by a descriptive message, which can be a custom or a default one. You can think of satisfying this requirement by introducing a property called `AdditionalMessage` with a backing field whose value defaults to `Title:`, which can be changed using the setter of the property. The changes are shown in the following snippet:

```
01. public class HtmlTitleRetriever
02. {
03.     private readonly IFileDownloader downloader;
04.     private readonly ITitleScraper scraper;
05.
06.     private string additionalMessage = "Title: ";
07.
08.     public string AdditionalMessage
09.     {
10.         get { return additionalMessage; }
11.         set { additionalMessage = value; }
12.     }
13.
14.     public HtmlTitleRetriever(IFileDownloader downloader,
15.         ITitleScraper scraper)
16.     {
17.         this.downloader = downloader;
18.         this.scraper = scraper;
19.     }
20.
21.     public string GetTitle(Uri file)
22.     {
23.         string fileContents = downloader.Download(file);
24.         return string.Concat(additionalMessage,
25.             scraper.Scrape(fileContents));
26.     }
27. }
```

Of course this change in the code of the class doesn't require any modifications to the configuration file of the container. The application will work fine since if no value is provided, the getter of the property will return the value set on the backing field during instantiation.

Now, what if you want to change the default message returned by the `GetTitle` method along with the title of the document? Remember, the responsibility to instantiate the `HtmlTitleRetriever` class is no longer yours, but the container's.

For this and other purposes Windsor offers a lot of configuration options, which I will try to illustrate in the remainder of this article and in the next one.

Going back to the new requirement, you want to be able to supply a different value for the `AdditionalMessage` property; other than the default value hard coded in the class. This can be accomplished via configuration parameters for components, which can be specified in the configuration file.

The snippet below shows how the configuration for the `HtmlTitleRetriever` component changes to

satisfy the requirement, by setting the `AdditionalMessage` property to `"This is the title of the document:"`.

```

1. <component id="HtmlTitleRetriever"
2.     type="WindsorSample.HtmlTitleRetriever, WindsorSample">
3.     <parameters>
4.         <AdditionalMessage>This is the title of the document:</AdditionalMessage>
5.     </parameters>
6. </component>

```

Notice the syntax used to specify the value of the property. Inside the component element you have inserted a child element called `parameters`, whose child is another element with the same name as the property you want to set. The mechanism of specifying dependencies by using properties is called *setter injection*.

Differently from constructor injection, setter injection has to do with optional dependencies, since the only constrain which needs to be satisfied when instantiating an object is that all the parameters accepted by its constructor must be supplied. Optional dependencies, instead, don't prevent the container from instantiating the component. They are usually assigned a default value which can be changed by specifying a new value in the configuration.

Another thing to note is that the syntax used above in the configuration file is valid for both setter and constructor injection. In this case, the constructor of the `HtmlTitleRetriever` class can be resolved automatically by the container because those are dependencies on other components registered in the configuration file. In case I make the dependency on the `AdditionalMessage` property compulsory by modifying the constructor and adding a third parameter of type string called `additionalMessage`, the syntax of the configuration file would be the same as above (the `AdditionalMessage` element in the xml file should be `additionalMessage` for coherence, to reflect the name of the parameter, but the container is case insensitive, so you can even write it all lowercase), with the only difference that this time the dependency would be compulsory. If not provided in the configuration file, an exception would be thrown by the container.

Back to setter injection. Upon component request, Windsor realizes that you have specified a configuration parameter which maps to a property of the class endowed with a setter (hence setter injection) and thus assigns to the property the value you specified after an instance is created. Remember: If no value is specified in the configuration file, no operation would be performed by the container on the property, and the backing field would keep its default value. Therefore, to avoid incurring in null references or incoherent values, always give a default value to optional dependencies.

Injecting components

Now on to something more interesting. While reading the last paragraph, you may have noticed the difference between the string parameter/property dependency-that you could specify by giving it a value in the configuration file-and the dependencies on other components, like the ones the `HtmlTitleRetriever` class has on the `IFileDownloader` and `ITitleScraper` services. Suppose you need to retrieve files via the FTP protocol instead of HTTP. You'd need to create another class implementing the `IFileDownloader` interface and register it in the configuration file as a new component. Let's see how it's done:

```

01. public class FtpFileDownloader : IFileDownloader
02. {
03.     public string Download(Uri file)
04.     {
05.         FtpWebRequest request = WebRequest.Create(file) as FtpWebRequest;
06.
07.         if(request != null)
08.             using(StreamReader reader =
09.                 new StreamReader(request.GetResponse().GetResponseStream()))
10.                 return reader.ReadToEnd();
11.
12.         return string.Empty;
13.     }
14. }

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

```

```

1. </component>
07. <component id="StringParsingTitleScraper"
08.     service="WindsorSample.ITitleScraper, WindsorSample"
09.     type="WindsorSample.StringParsingTitleScraper, WindsorSample">
10. </component>
11. <component id="HttpFileDownloader"
12.     service="WindsorSample.IFileDownloader, WindsorSample"
13.     type="WindsorSample.HttpFileDownloader, WindsorSample">
14. </component>
15. <component id="FtpFileDownloader"
16.     service="WindsorSample.IFileDownloader, WindsorSample"
17.     type="WindsorSample.FtpFileDownloader, WindsorSample">
18. </component>

```

Now you might be asking: how do I switch from one implementation to the other? There's obviously a way to explicitly specify a particular component. In case of no explicit configuration and multiple components implementing the same service, Windsor should choose the first component registered in the configuration file.

I said *should* because I noticed that there's a little subtlety (read, *issue*) that needs to be taken into account. At the time of writing, it looks like the expected first registered-first chosen behavior can be achieved only as long as the IDs of the components stay below the 8 characters in length. Otherwise, the choice seems unpredictable.

This is a good reason to show you how to explicitly switch implementations at runtime using Windsor configuration. The syntax to accomplish this task differs from the one used to specify dependencies on explicit values like strings, numbers, dates and so on. Otherwise, the container would be unable to infer what you want to do. The syntax used to specify components as parameters uses the `{component id}` format. Let's see an example. Having registered the `FtpFileDownloader` component, you can choose to use it when resolving the `HtmlTitleRetriever` class by simply adding a new configuration parameter to the `HtmlTitleRetriever` component, as shown in the following snippet.

```

1. <component id="HtmlTitleRetriever"
2.     type="WindsorSample.HtmlTitleRetriever, WindsorSample">
3.     <parameters>
4.         <AdditionalMessage>This is the title of the document:</AdditionalMessage>
5.         <downloader>{FtpFileDownloader}</downloader>
6.     </parameters>
7. </component>

```

Notice how the new syntax is inserted in the parameters section just like you did before with a simple string parameter. Now you can switch between the two implementations of the `IFileDownloader` service just by referencing their IDs. The xml item is called `downloader` because it matches the name of the constructor parameter, like you did before with the `AdditionalMessage` property.

Working with arrays, lists and dictionaries

At the moment the `HtmlTitleRetriever` class is not very useful when it comes to downloading files with different protocols or stored in different locations. Each time you'd have to change the configuration file to reference a specific component for that protocol. Let's try resolving this issue.

Note: actually, the `WebClient` class is able to retrieve files via FTP, so there would be no real need for a new class. The redundant class is implemented for educational purposes only.

The `HtmlTitleRetriever` class should be able to deduce how the file supplied to the `GetTitle` method needs to be retrieved. It should be able to retrieve the file if it can rely on a component created for that purpose. In order to do this, let's first change the contract of the `IFileDownloader` service to provide a method called `SupportsUriScheme`.

```

1. public interface IFileDownloader
2. {
3.     string Download(Uri file);
4.     bool SupportsUriScheme(Uri file);
5. }

```

This method will need to be implemented by inheritors to return a Boolean values indicating if they are able to download a file with that specific schema. So here's how the `HttpFileDownloader` and `FtpFileDownloader` classes change.

```
01.
```

```
02. public class HttpFileDownloader : IFileDownloader
03. {
04.     public string Download(Uri file)
05.     {
06.         return new WebClient().DownloadString(file);
07.     }
08.     public bool SupportsUriScheme(Uri file)
09.     {
10.         return file.Scheme == "http";
11.     }
12. }

01. public class FtpFileDownloader : IFileDownloader
02. {
03.     public string Download(Uri file)
04.     {
05.         FtpWebRequest request = WebRequest.Create(file) as FtpWebRequest;
06.
07.         if(request != null)
08.             using (StreamReader reader =
09.                 new StreamReader(request.GetResponse().GetResponseStream()))
10.                 return reader.ReadToEnd();
11.
12.         return string.Empty;
13.     }
14.
15.     public bool SupportsUriScheme(Uri file)
16.     {
17.         return file.Scheme == "ftp";
18.     }
19. }
```

With this trick, each component providing the `IFileDownloader` service has knowledge about its capability to download a specific file. You will see shortly why this makes sense.

Going back to our not-very-useful `HtmlTitleRetriever` class; in order to let it download files with different protocols it needs references to several implementations of the `IFileDownloader` service, each for a supported protocol/location. This can be accomplished in several ways; let's see how to supply multiple components using arrays. The constructor of the class now accepts an array of `IFileDownloaders`, as shown in the following snippet.

```
01. public class HtmlTitleRetriever
02. {
03.     private readonly IFileDownloader[] downloaders;
04.     private readonly ITitleScraper scraper;
05.
06.     private string additionalMessage = "Title: ";
07.
08.     public string AdditionalMessage
09.     {
10.         get { return additionalMessage; }
11.         set { additionalMessage = value; }
12.     }
13.
14.     public HtmlTitleRetriever(IFileDownloader[] downloaders,
15.                             ITitleScraper scraper)
16.     {
17.         this.downloaders = downloaders;
18.         this.scraper = scraper;
19.     }
20.
21.     public string GetTitle(Uri file)
22.     {
23.         foreach(IFileDownloader downloader in downloaders)
24.             if(downloader.SupportsUriScheme(file))
25.                 return string.Concat(additionalMessage,
26.                                     scraper.Scrape(downloader.Download(file)));
27.
28.         return string.Empty;
29.     }
30. }
```

```
    }
```

The body of the `GetTitle` method has changed. Now it iterates through the array of downloaders and checks if it can find one suitable for downloading the file. Once it finds one, it downloads the file and scrapes the title as done before. If no suitable downloader is found, an empty string is returned. So far nothing new, but you need to change the configuration to let Windsor understand that you want the class populated with an array of downloaders. Here's how the configuration file takes advantage of this feature.

```
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.       </array>
10.     </downloaders>
11.   </parameters>
12. </component>
```

The code is pretty simple to understand, so I will not delve deeper into it. As for the parameters element, the array element can accept any parameter—be it a string, Boolean, DateTime or a reference to a component. The only difference is in the syntax used to specify the component, which requires its ID surrounded by brackets and preceded by the dollar `$` symbol.

Just like arrays, you can pass lists and dictionaries to constructors and properties. The syntax is similar and is shown in the following snippets. The two snippets below show how to achieve the same result using lists instead of arrays.

```
01. public class HtmlTitleRetriever
02. {
03.     private readonly List<IFileDownloader> downloaders;
04.     private readonly ITitleScraper scraper;
05.
06.     private string additionalMessage = "Title: ";
07.
08.     public string AdditionalMessage
09.     {
10.         get { return additionalMessage; }
11.         set { additionalMessage = value; }
12.     }
13.
14.     public HtmlTitleRetriever(List<IFileDownloader> downloaders,
15.         ITitleScraper scraper)
16.     {
17.         this.downloaders = downloaders;
18.         this.scraper = scraper;
19.     }
20.
21.     public string GetTitle(Uri file)
22.     {
23.         foreach(IFileDownloader downloader in downloaders)
24.             if(downloader.SupportsUriScheme(file))
25.                 return string.Concat(additionalMessage,
26.                     scraper.Scraper(downloader.Download(file)));
27.
28.         return string.Empty;
29.     }
30. }
```

```
01. <component id="HtmlTitleRetriever"
02.     type="WindsorSample.HtmlTitleRetriever, WindsorSample">
03.   <parameters>
04.     <AdditionalMessage>This is the title of the document:</AdditionalMessage>
05.     <downloaders>
06.       <list>
07.         <item>${HttpFileDownloader}</item>
08.         <item>${FtpFileDownloader}</item>
09.       </list>
10.     </downloaders>
```

```

11.     </downloaders>
12. </parameters>
13. </component>

```

The following snippets show how to use dictionaries.

```

01. public class HtmlTitleRetriever
02. {
03.     private readonly IDictionary<string, IFileDownloader> downloaders;
04.     private readonly ITitleScraper scraper;
05.
06.     private string additionalMessage = "Title: ";
07.
08.     public string AdditionalMessage
09.     {
10.         get { return additionalMessage; }
11.         set { additionalMessage = value; }
12.     }
13.
14.     public HtmlTitleRetriever(IDictionary<string, IFileDownloader>
15.         downloaders, ITitleScraper scraper)
16.     {
17.         this.downloaders = downloaders;
18.         this.scraper = scraper;
19.     }
20.
21.     public string GetTitle(Uri file)
22.     {
23.         foreach(IFileDownloader downloader in downloaders.Values)
24.             if (downloader.SupportsUriScheme(file))
25.                 return string.Concat(additionalMessage,
26.                     scraper.Scrape(downloader.Download(file)));
27.
28.         return string.Empty;
29.     }
30. }

```

```

01. <component id="HtmlTitleRetriever"
02.     type="WindsorSample.HtmlTitleRetriever, WindsorSample">
03.     <parameters>
04.         <AdditionalMessage>This is the title of the document:</AdditionalMessage>
05.         <downloaders>
06.             <dictionary>
07.                 <item key="http">${HttpFileDownloader}</item>
08.                 <item key="ftp">${FtpFileDownloader}</item>
09.             </dictionary>
10.         </downloaders>
11.     </parameters>
12. </component>

```

Note that by using dictionaries I could have omitted the `SupportsUriScheme` method, since I could have specified the name of the supported protocol as the key of the dictionary entry and then retrieved it by key in the `GetTitle` method.

Summary

In this article you've seen how to deal with more sophisticated requirements using Windsor container configuration. In particular, I've talked about optional and compulsory dependencies, and how they map respectively to properties and constructors, as well as how to switch service implementations using a specific syntax offered by Windsor. Finally, I've given a quick overview of how to work with arrays, lists and dictionaries. In the next article I will talk about more configuration options and introduce advanced concepts like decorators 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 Container Tutorials](#) - 2007
- [Castle Windsor Container documentation](#)

Original Url: <http://dotnetslackers.com/articles/designpatterns/InversionOfControlAndDependencyInjectionWithCastleWindsorContainerPart2.aspx>