

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

In this article I introduce the missing core features I didn't tackle before.

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 case you didn't read the previous parts I highly suggest doing so. The sample application I used to illustrate the framework has been developed step by step since the beginning. 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. As usual, any feedback, questions and votes are very welcome, but now on to the nitty-gritty.

Lifestyles

So far I haven't delved much into discussing the way components are instantiated and released. Actually, retrieving them from the container using `IWindsorContainer.Resolve` and releasing them using `IWindsorContainer.Release` doesn't necessarily mean that they are first instantiated and then disposed of by setting the reference (obtained with the `Resolve` method call) to `null`.

Instead, the default behavior supplied by the container is called singleton, which means that upon the first call to `Resolve` the component is instantiated by calling the class constructor that best matches the supplied dependencies as defined in the configuration file. Then all subsequent calls just return a reference to the same object in memory. In other words, the component behavior is that defined by the [Singleton pattern](#), which means that a single instance of it is alive in the application domain, which is cached and returned at each request forwarded to the container. By default, calls to the `Release` method don't have any effect on the component. This makes sense since a singleton object shouldn't be garbage collected as long as the application is running.

In Windsor Container, the way components are resolved and released is called *lifestyle*. In the sample application, as a consequence, all the components are given the default singleton lifestyle, since no additional configuration is supplied. However, Windsor offers several other lifestyles to choose from, as well as an API to create your own custom lifestyle.

The built-in lifestyles offered by Windsor Container are illustrated in the following table, along with the actions performed by the `Resolve` and `Release` methods.

Table 1: Built-in lifestyles offered by Windsor Container

Lifestyle	Upon Resolve	Upon Release
Singleton (default)	The first call instantiates the class and returns a reference to it, subsequent calls return references to the same object.	No action.
Transient	Each call creates and returns a reference to a new object.	No action.

Lifestyle	Upon Resolve	Upon Release
Thread	A single instance is created per thread and then cached. Each thread owns a single instance of a component.	No Action.
Pooled	Components instances are retrieved from a pool as long as it contains any, otherwise they are instantiated.	Components are pushed back into the pool and made available for subsequent requests until the pool is full, thrashed away otherwise.
Custom	Custom behavior.	Custom behavior.

Switching lifestyles can either be accomplished programmatically via attributes or declaratively using the configuration file. For example, the transient lifestyle can be applied to a component in one of either ways shown in the following snippets.

```

1. [Transient]
2. public class HtmlTitleRetriever
3. {
4.     //...
5. }

1. <component id="HtmlTitleRetriever"
2.     type="WindsorSample.HtmlTitleRetriever, WindsorSample"
3.     lifestyle="transient">
4. </component>

```

Lifestyles are very useful and pretty straightforward to understand. The Pooled style needs some explanation, since it requires a little more work on the configuration. It accepts two more configuration parameters which indicate the initial and maximum size of the pool, as shown below.

```

01. [Pooled(2,5)]
02. public class HtmlTitleRetriever
03. {
04.     //...
05. }
06.
07. <component id="HtmlTitleRetriever"
08.     type="WindsorSample.HtmlTitleRetriever, WindsorSample"
09.     lifestyle="pooled"
10.     initialPoolSize="2"
11.     maxPoolSize="5">
12. </component>

```

Note: As for the other lifestyles you only need to specify the lifestyle using one of the two methods, either attributes or configuration files.

Furthermore, its behavior intrinsically deviates from the standard behavior, since the pool is filled to its initial size as soon as the container is instantiated - before any instance of a component is ever requested. In other words, when the container is instantiated, a number of any pooled components equal to the respective initial pool size configuration parameter are immediately created. Requests of those components made to the container are satisfied by the pool, with no additional overhead introduced by just-in-time instantiation. Once the number of requests exceeds the initial pool size, new instances are created. On the other side of the lifestyle, upon release, component references are returned back to the pool and made available to satisfy subsequent requests made to the container, until the pool size reaches its maximum value. Subsequent releases are simply discarded by the container, as if those components didn't belong to the pool.

Note: The lifestyle of a component can be influenced by other configuration options exposed by the container. They will be discussed in the following sections.

To illustrate how lifestyles influence the behavior of components, let's edit the body of the main method of the sample application to request two references to a component and print out the hash code of their pointed object.

```

01. public class Program
02. {
03.     private static void Main()
04.     {
05.

```

```

06.         IWindsorContainer container = new CustomContainer(new XmlInterpreter());
07.         HtmlTitleRetriever retriever = container.Resolve<HtmlTitleRetriever>();
08.         HtmlTitleRetriever retriever1 = container.Resolve<HtmlTitleRetriever>();
09.
10.         Console.WriteLine("retriever hashCode: {0}", retriever.GetHashCode());
11.         Console.WriteLine("retriever1 hashCode: {0}", retriever1.GetHashCode());
12.
13.         foreach(string title in retriever.GetTitles())
14.             Console.WriteLine(title);
15.
16.         container.Release(retriever);
17.     }
18. }

```

If the default singleton lifestyle is used, the two hash codes are equal, since two references to the same `HtmlTitleRetriever` singleton object are returned. If the class is made transient, then the program will output different hash codes, since a new instance is created at each request.

In addition to the built-in lifestyles Windsor exposes a public API to create custom ones. To do so you need to implement the `ILifestyleManager` interface and inform the container that you want to use it instead of the built-in managers. The most straightforward way of creating a new lifestyle, however, is to inherit from the `AbstractLifestyleManager` abstract class, which already provides some support for common operations performed by all lifestyle managers. A sample custom lifestyle can be found at [this page](#) in the Windsor documentation.

Note: Lifestyles different from the default singleton didn't find any use in the sample application, so this part is not implemented in any of the components of the attached code.

Extending the container: Facilities

So far I've illustrated most of the features offered by Windsor Container, either basic and advanced ones. However, you may have noticed that they are naive features, in that they supply extremely simple services, although they carry them out in a very smart way.

Now, what if you wanted the container to supply a custom feature which is not provided out of the box? There are several options; you would either fill a feature request to the project's developers, create the feature on your own by changing the source code of the container or subclass the container and work with your custom container (as I did to support type conversion for Uri properties). All of this if Windsor didn't provide an extension mechanism called *Facilities*.

Note: Actually, facilities are part of the MicroKernel, although I'll employ them at Windsor Container level mostly using configuration files.

Following is the definition of facilities as given in the official documentation:

Facilities augment the MicroKernel capabilities by integrating it with a different project or technology, or by implementing new semantics.

Facilities can be grouped into three categories, as shown in the following table.

Table 2: Facilities

Category	Description
Integration facilities	Provide support for integration with other frameworks and projects, like NHibernate and ActiveRecord.
Basic services facilities	Provide support for services built into CastleProject. Services are just a means to carry out common tasks using a standard API.
Semantic facilities	Extend the features of the container by supplying new semantics for component configuration and management.

I will narrow the discussion to semantic facilities only, since they are intrinsically related to the container and don't interact with external services or frameworks.

Note: A list of all the facilities supported by the MicroKernel can be found at [this page](#) in the documentation.

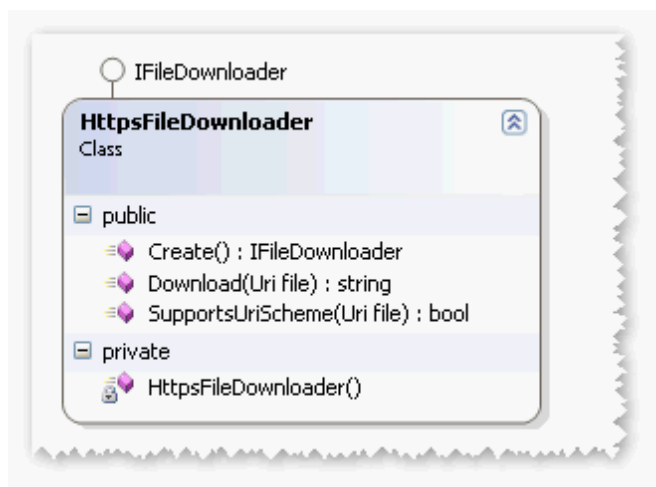
The following sections will take advantage of two semantic facilities in order to respond to additional requisites for the sample application.

FactorySupport facility

Now let's move on to something more interesting. The new requirement is that the sample application needs to be able to retrieve files via the HTTPS protocol as well. Actually, the `WebClient` class provided by the .NET Framework is able to handle it; but in a real project the new requirement would involve some more work.

Suppose that the handling of the HTTPS protocol would require a lot of work, and that somehow you already own a class which provides this service. Luckily - although very unlikely in a real world case - this class, called `HttpsFileDownloader`, already implements the `IFileDownloader` interface. As a consequence, it can be registered together with the other file downloaders in the configuration file; and supply the same service. Unfortunately, it doesn't supply a public constructor, but instead just a static method called `Create` which is the only public access point for creating instances of the class. Furthermore, you don't own the source code, but just a compiled assembly, so there's no straightforward way to circumvent the constraint imposed by the absence of a public constructor, which is needed by Windsor to instantiate components.

Figure 1: HttpsFileDownloader class diagram



One of the facilities supplied by the MicroKernel is called *FactorySupport* facility. Its name comes from the *FactoryMethod* design pattern. The facility provides a means of instantiating components without using their constructor, which is the default route.

In order to use any facility together with Windsor, it first needs to be registered in the configuration file as shown in the following snippet.

```

1. <castle>
2.   <facilities>
3.     <facility id="factory.support"
4.       type="Castle.Facilities.FactorySupport.FactorySupportFacility,
5.         Castle.MicroKernel" />
6.   </facilities>
7. </castle>
  
```

By registering the `FactorySupport` facility you have indirectly extended the container with a full new API which lets you use a new syntax to configure components and benefit from the facility. As shown in Figure 1, everyone who needs an instance of the `HttpsFileDownloader` class needs to call the `Create` method, which is the only method capable of calling the private constructor of the class. So how do you instruct the container that in order to instantiate that component it needs to perform these steps?

If the component is registered as is, upon resolution an exception of type `ComponentActivatorException` is raised, since no accessible constructor was found. The `FactorySupport` facility, instead, provides you with a new syntax which lets you specify the class responsible of instantiating a component as well as the name of the method to call on that class. The following configuration snippet shows how to benefit from the new configuration options provided by the facility.

```

1. <component id="HttpsFileDownloader"
2.   service="WindsorSample.IFileDownloader, WindsorSample"
3.   type="WindsorSample.HttpsFileDownloader, WindsorSample"
4.   factoryId="HttpsFileDownloader"
5. >
  
```

```

6. |         factoryCreate="Create">
    | </component>

```

You can see that the configuration of the component has gained two new parameters, `factoryId` and `factoryCreate`. The first is the ID of the component supplying the factory method, while the second is the name of the method to call. In this case, the class exposing the factory method is the same as the class which needs to be instantiated, but you may even split up the responsibilities in two different classes.

Note that this is working because the factory method is static. If it was an instance method the container wouldn't be able to instantiate the component because a circular dependency would exist. The snippet below shows the source code of the `HttpsFileDownloader` class.

```

01. | public class HttpsFileDownloader : IFileDownloader
02. | {
03. |     private HttpsFileDownloader ()
04. |     {
05. |
06. |     public static IFileDownloader Create()
07. |     {
08. |         return new HttpsFileDownloader();
09. |     }
10. |
11. |     public string Download(Uri file)
12. |     {
13. |         return new WebClient().DownloadString(file);
14. |     }
15. |
16. |     public bool SupportsUriScheme(Uri file)
17. |     {
18. |         return file.Scheme == Uri.UriSchemeHttps;
19. |     }
20. | }

```

With these simple steps you have instructed the container to call the `Create` method in order to instantiate the component. For the rest, the component behaves just like any other.

Startable facility

The other semantic facility I'm going to talk about is the Startable facility. To understand what it can be used for let's introduce a new requirement to the sample application. So far, once the `HtmlTitleRetriever` class was requested to the container, you had explicitly asked it to retrieve the titles of the files using the `GetTitles` method. This entails that each time you have to wait for it to complete the download and the parsing before continuing the execution of the program. What about taking advantage of multithreading to accomplish this task so that you don't have to block the UI? And what about doing it implicitly - that is, as soon as the application starts and then at regular intervals - so that the waiting time can be virtually reduced to zero?

This may sound weird, but I'll show what I mean very soon. The idea is to make the `HtmlTitleRetriever` class *startable*, so that it begins retrieving titles as soon as the application starts, and without blocking the user interface. I'll accomplish this using the `System.Threading.Timer` class.

The Startable facility lets you benefit from a new semantic to start and stop components. This means that components made startable are instantiated and "started" as soon as possible when the application begins; and then "stopped" when they are released or when the container is disposed.

As with any facilities, the first step is to register it in the configuration file, as follows.

```

1. | <facilities>
2. |     <facility id="startable"
3. |         type="Castle.Facilities.Startable.StartableFacility,
4. |         Castle.MicroKernel" />
5. | </facilities>

```

To make a component startable you can either implement the `IStartable` interface or use the configuration file. The `IStartable` interface contract exposes two methods, called `Start` and `Stop`, as shown below.

```

1. | public class HtmlTitleRetriever : IStartable
2. | {
3. |

```

```
4.     public void Start()
5.     {
6.     }
7.     public void Stop()
8.     {
9.     }
10. }
```

Taking the configuration route lets you optionally set the method names to start and stop your component, as shown in the two following snippets.

```
1. <component id="HtmlTitleRetriever"
2.           type="WindsorSample.HtmlTitleRetriever, WindsorSample"
3.           startable="true"
4.           startMethod="Begin"
5.           stopMethod="End">
6. </component>
```

```
1. public class HtmlTitleRetriever
2. {
3.     public void Begin()
4.     {
5.     }
6.     public void End()
7.     {
8.     }
9. }
```

Now, let's implement the new requirement in the `HtmlTitleRetriever` class and delegate the startable facility the task of starting and stopping it. As stated above, I'll use a timer to do the work of downloading and scraping the titles asynchronously on a thread supplied by the CLR thread pool. The code of the edited `HtmlTitleRetriever` class is shown below.

```
01. public class HtmlTitleRetriever : IStartable
02. {
03.     private readonly IFileDownloader[] downloaders;
04.     private readonly ITitleScraper scraper;
05.
06.     private Timer timer;
07.     private readonly List<string> scrapedTitles = new List<string>();
08.     private readonly object scrapedTitlesListLock = new object();
09.
10.     private string additionalMessage = "Title: ";
11.     private Uri[] files;
12.
13.     public HtmlTitleRetriever(IFileDownloader[] downloaders, ITitleScraper scraper)
14.     {
15.         this.downloaders = downloaders;
16.         this.scraper = scraper;
17.     }
18.
19.     public Uri[] Files
20.     {
21.         get { return files; }
22.         set { files = value; }
23.     }
24.
25.     public string AdditionalMessage
26.     {
27.         get { return additionalMessage; }
28.         set { additionalMessage = value; }
29.     }
30.
31.     #region IStartable Members
32.
33.     public void Start()
34.     {
35.         if(timer == null)
36.             timer = new Timer(GetTitles, null, TimeSpan.Zero, new TimeSpan(0, 10, 0));
37.     }
38.
39.     public void Stop()
40.     {
```

```

        if(timer != null)
            timer.Change(Timeout.Infinite, Timeout.Infinite);
    }
}
#endregion

public string GetTitle(Uri file)
{
    foreach(IFileDownloader downloader in downloaders)
        if(downloader.SupportsUriScheme(file))
            return string.Concat(additionalMessage,
                scraper.Scrape(downloader.Download(file)));

    return string.Empty;
}

public IEnumerable<string> GetTitles(bool forceRefresh)
{
    if(forceRefresh)
        foreach(Uri file in files)
            yield return GetTitle(file);
    else
        lock(scrapedTitlesListLock)
            foreach(string title in scrapedTitles)
                yield return title;
}

private void GetTitles(object state)
{
    IEnumerable<string> tempTitles = GetTitles(true);

    lock(scrapedTitlesListLock)
    {
        scrapedTitles.Clear();
        scrapedTitles.AddRange(tempTitles);
    }
}
}

```

The `Start` and `Stop` methods respectively start and stop a timer which executes a new callback method called `GetTitles(object)`. This method simply retrieves the titles of the files stored in the `Files` property and caches them into a list so that they can be returned upon request without re-downloading them. Synchronization is required since access to the list may occur on different threads.

Another change has been introduced in the `GetTitles()` method. Before, it didn't take any arguments. Now a Boolean value is required, which is used to specify if you want to retrieve the values cached in the list or force a new download. The code has changed quite a bit, so take the time to look at it and make sense of what it does.

No additional configuration is needed for Windsor to be able to manage the component correctly. By making it startable, the `HtmlTitleRetriever` component is no longer instantiated when explicitly requested. Instead, it is created upon container instantiation, and the start method is immediately called. Thereby, even before the first request to be resolved, it has already been created and started; and hopefully it has already finished downloading files. Below is the code with the changes made to the sample application.

```

01. private static void Main()
02. {
03.     IWindsorContainer container = new CustomContainer(new XmlInterpreter());
04.
05.     // Here the HtmlTitleRetriever has already been instantiated and its Start
06.     // method called
07.     // Simulate some work on the current thread, files are being downloaded on
08.     // another thread
09.     Console.WriteLine("Doing some work...");
10.     Thread.Sleep(10000);
11.
12.     HtmlTitleRetriever retriever = container.Resolve<HtmlTitleRetriever>();
13.     foreach(string title in retriever.GetTitles(false))
14.         Console.WriteLine(title);

```

```

15.     container.Release(retriever);
16.     container.Dispose();
17.     // Here the Stop method has been called
18. }

```

As highlighted in the code snippet above, the component is instantiated during container creation. What might not be completely intuitive is when it gets released. That is, when its `Stop` method gets called. You may think that this occurs during the call to the container's `Release` method. Actually, this happens during container disposal.

This is more intuitive if you think in terms of component lifestyle. I kept the default singleton lifestyle, therefore a single instance of the component is created in the `AppDomain` of the application. When you release it using the `Release` method, no action is performed because the component might still be requested again or might have other references active; it mustn't be neither stopped nor disposed. Only when the container itself is disposed then the component can be stopped too, because no one else will ever require it again. This behavior might change if the component had a different lifestyle. I will talk about release policy shortly.

Other facilities

In addition to those discussed earlier, Windsor provides several other ready-to-use facilities, either semantic, integration and based on services. They all can be useful during the development of a real project, in particular those which ease integration with external frameworks, like NHibernate or ActiveRecord. I didn't cover them during this series since they would require a more articulated example that would entail cooperation with a storage system. I can say that they make working with those frameworks a lot easier than it would be otherwise.

One facility based on services which is worth mentioning is the *Logging* facility. It makes integrating logging services in classes a breeze, along with the decorator pattern, as discusses in the previous articles.

Lifecycles

Lifecycles provide a means of customizing the tasks performed by the container upon component creation and destruction. They can be grouped in two categories, using Castle official terminology: *commission* and *decommission*. Commission tasks are performed right after the component is created - in most cases when its constructor is called and optional dependencies have been injected. Decommission tasks are executed upon component disposal.

As explained before, the disposal of a component doesn't necessarily occur when the `Release` method is called, but it depends on its lifestyle. To clarify: If a component is transient, its disposal - and thus the decommission tasks - are performed when the `Release` method is called on the container. If the component is a singleton, it's not disposed when the `Release` method is called; it happens when the `Dispose` method is called on the container. As stated before, this makes sense because a singleton component may have other references still active and shouldn't be destroyed until all those references become inactive. This, in turn, can be ensured only when the container is disposed.

The means by which you can execute custom code during commission and decommission is via interfaces. The MicroKernel supports two interfaces; one for commission and one for decommission, as shown in the following table.

Table 3: Interfaces for commission and decommission

Category	Supported Interfaces
Commission	Castle.Model.IInitializable System.ComponentModel.ISupportInitialize
Decommission	System.IDisposable

By implementing these interfaces you can be sure that the methods they expose will be called by the container at the right time, depending on the lifestyle of the component.

In practice, this appears much useful than it seems in words. Suppose you have a class which should dispose managed and/or unmanaged resources when garbage collected. How does Windsor know that it should call its `Dispose` method? You can see that lifecycles is just a bright way to say that the container is smart enough to guess that if your class implements those interfaces, then it should call their methods at the right time, just as you would do if you didn't uptake IoC.

Let's see how to benefit from this feature in the sample application. As from the last edit to the code, the `HtmlTitleRetriever` class instantiates and uses a `Timer`, which implements the `IDisposable` interface since it makes use of the Win32 API. It makes sense to dispose it when the singleton object is no longer referenced. The most straightforward way of doing this is by implementing the `IDisposable` interface and -inside the `Dispose` method - call the namesake method on the timer. This ensures that when the component is destroyed the timer has the chance to release its resources. This wouldn't happen if you didn't implement the `IDisposable` interface and the container wasn't smart enough to understand that it should call the `Dispose` method.

The snippet below shows the changes needed to take advantage of the decommission lifecycle on the `HtmlTitleRetriever` class.

```
01. public class HtmlTitleRetriever : IStartable, IDisposable
02. {
03.     //...
04.
05.     public void Dispose()
06.     {
07.         if(timer != null)
08.         {
09.             timer.Dispose();
10.             timer = null;
11.         }
12.     }
13. }
```

Note that no additional configuration was required to apply the decommission lifecycle to the component. As for the `IStartable` interface, the container was smart enough to infer the task it needed to accomplish.

Note: The above implementation of the disposable pattern is not recommended according to the .NET Framework Design Guidelines. It has been simplified to fit the needs of the sample application.

Release policy

Lifestyles, startable components and lifecycles may be a bit confusing at first, as well as the usefulness of calling the `Release` method on the container to release components. The general rule is that it's good to release as soon as you've finished using a component. You've seen that often the `Release` method - in practice - doesn't perform any action. If a component has no decommission concerns, then the `Release` method is a no-op, unless its lifestyle is pooled. In this case, its release pushes it back into the pool so it needs to be released once used, to avoid emptying the pool. Even if a component is transient the release won't perform any action, unless it has decommission concerns. In that case, the `Dispose` method will be called because the component it's not referenced anymore. In the sample application all components were singletons, so both decommission and startable concerns occurred at container disposal instead of component release. In general, to avoid mistakes and make sense of how this all works, my advice is to use the debugger to see when all those methods are called.

Summary

In this last article I've introduced the remaining and less intuitive features of Windsor Container, such as lifestyles, facilities and lifecycles. At the same time they represent the most flexible extension points of Windsor. I've shown you how to take advantage of them in the sample application I've developed step-by-step throughout the set of articles.

I hope I've been able to hand on my knowledge and pleasure of working with Windsor; and that you've enjoyed reading through the articles.

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](#)
- [Castle Windsor/MicroKernel forum](#)

Original Url: <http://dotnetslackers.com/articles/designpatterns/InversionOfControlAndDependencyInjectionWithCastleWindsorContainerPart4.aspx>

