AOP way of Logging and Tracing using Castle Windsor IoC container

Published By:	***Sanjay Kumar***
Technology Category:	Microsoft – ASP.Net MVC 4.0 (C#)
Request Raised By:	Reusable Code Component Group
Published Date:	2014-01-27
Version:	1.0
Source File Name:	LoggingAndTracingUsingAop.zip
Reviewed/Tested By:	Vijay L Reddy

Table of Contents

Component Revision History

Rev. No.	Author	Date	Reason for Change	Change Description
1	Sanjay Kumar	2014-01-27	Created	

Introduction

There are some aspects to application programming, such as logging, tracing, profiling, exception handling, application-wide caching and pooling, authentication and authorization etc. that cut across the business objects. These are difficult to deal with in an object-oriented paradigm without resorting to code-injection, code-duplication or interdependencies. If we analyze the source code of any software system we will soon realize that the different parts of the code can be grouped according to their responsibilities. For example some parts of the code are responsible for implementing business logic, other parts manage transactions, yet other parts are used to handle exceptions etc.

In AOP terms those responsibilities are called *concerns*. The business logic is considered the *core concern* of the application; whereas the latter examples represent *aspects* that somehow interact with the core concerns. Because they interact with the core concerns, aspects are often called *cross-cutting concerns*.

Cross-cutting concerns are aspects of a software system that are difficult to be cleanly decomposed and separated in terms of design and implementation from the business logic of the system. Examples of cross cutting concerns are: Logging, Exception Handling, Tracing Transaction, Caching, Security, Validation, Instrumentation, and Method Invocation Retries.

With OOP, these cross cutting concerns are implemented as separate classes and then later on invoked by code within Service/Business/Data/UI layers. The result is violation of **DRY** principle and low cohesion.

AOP attempts to solve this problem. The paradigm of AOP is a means by which cross cutting concerns can be isolated, encapsulated, modularized and applied uniformly throughout a system thus avoiding many of the problems.

- Adding cross-cutting concerns with little or no modification to existing business logic code
- Applying these behaviors to multiple methods, classes, and assemblies with minimal additional code.
- Eliminating redundant boilerplate code that must be copied from method to method.

In ASP.NET MVC, you can use attributes in the form of action filters to provide a neater way of implementing these cross-cutting concerns, but this might not always be enough for what you want and are only limited to Controllers and their Action methods. Windsor Interceptors allow for a much more powerful way to intercept method calls and act on them, on any of your classes.

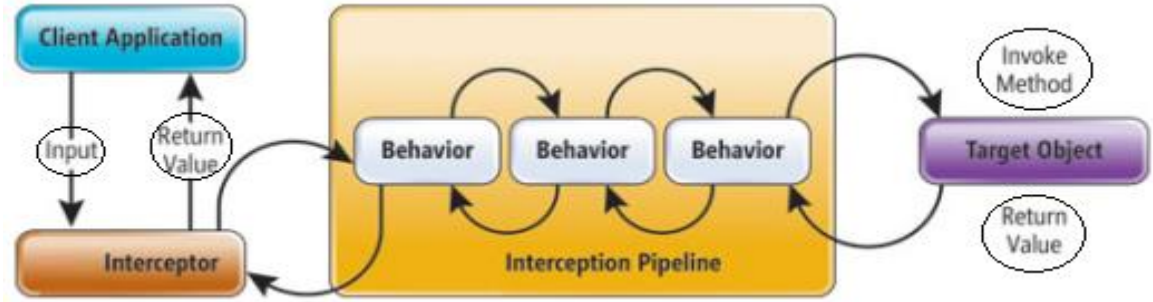
Here, we will be talking about Castle.Windsor Interceptors which we think brings AOP to the masses within the .NET community. Castle.Windsor Interceptors work by proxying classes and allows you to act before and after a method is invoked on a target class. We will show a simple example to integrate Castle Interceptors in both client & Server Side with an ASP.NET MVC 4 application and do logging through Interceptors.

Castle Windsor as AOP Framework:

Dependency Injection (DI) Containers (like Castle Windsor we use) is used to manage dependencies and lifecycles of object is your application. So, it helps you to build loosely coupled components/modules in your application. It's generally initialized on startup of your application. In an ASP.NET application, global.asax is the mostly used place to initialize it, where as in WCF side you can create a static constructor to invoke the DI.

With a canonical approach, you hard code the classes of the objects you want to instantiate in the source of your application, supply parameters to their constructors and manage their interactions. Each object knows at compile time which is the real classes of the objects they need to interact with, and they will call them directly. But Castle Windsor is a framework which makes your application smart enough to guess which objects to instantiate, how to instantiate them and, in general, how to control their behavior. Instead of working with concrete classes you'll work with abstractions like interfaces or abstract classes, letting your application decide which concrete classes to use and how to satisfy their dependencies on other components.

This diagram may help to illustrate how proxy based Castle Windsor AOP works:



Sample Application:

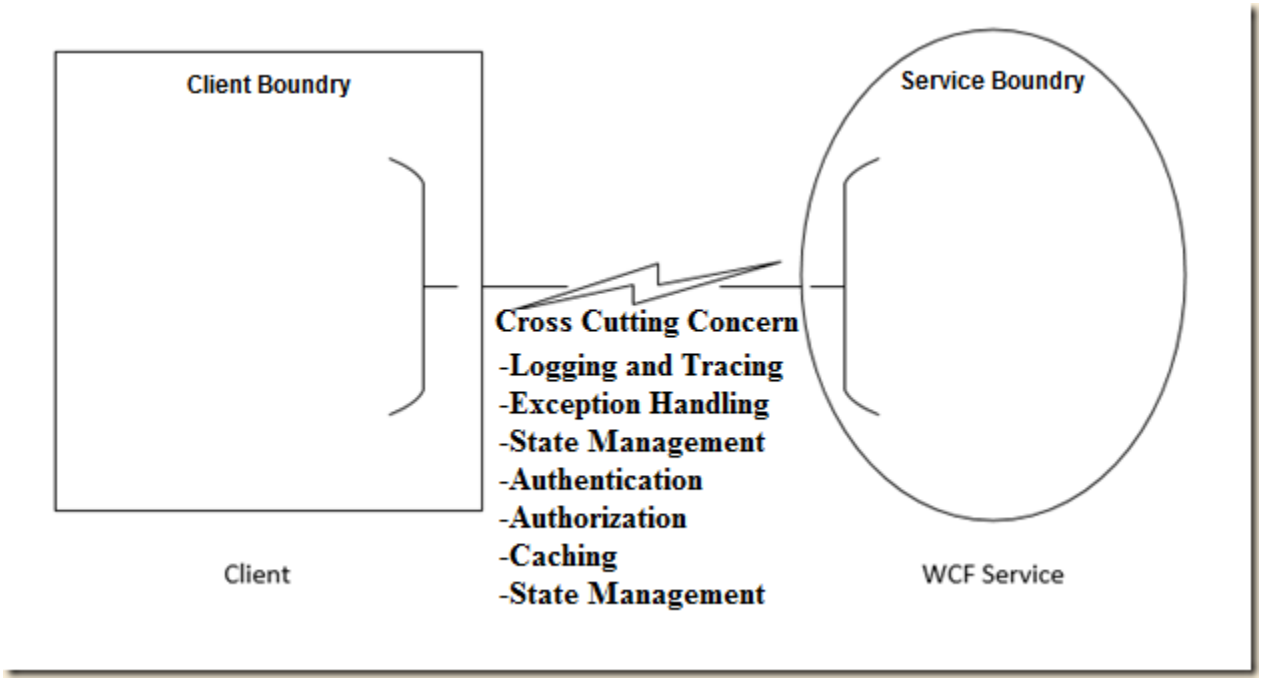
Requirements of the sample application:

Consider this scenario: you expose some services with WCF, sometimes customers tell you that your services have bugs or they encountered an exception, or they get wrong result, etc. In this situation you receive information like: “I got exception from your service”, and no more details, so you need to spent time trying to understand what has happened. Since this is probably the most detailed information you can have from the customer, having a good log system is vital. You need to identify what has happened and you need to build a system to log:

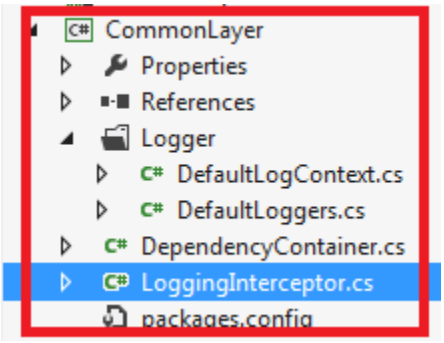
- Every call to service methods
- all parameters value for the call
- return value and exception that are raised (if one).

The goal is having a full and configurable log of what has happened in the service to retrace exception and problems experienced by users. Here, we are filtering logging and tracing details at client & WCF boundary.

The below diagram gives you an idea of how the component has been implemented –



So let's create a Project With below structure: This layer holds the concept of Logging and tracing, which will be used for both Client and WCF side latter to logging and tracing details.



Description with Implementation Details

Create Class: - LoggingInterceptor.cs:

/** Castle Windsor has an interface that you must implement in order to write an interceptor called IInterceptor that looks like the below code. The **IInvocation** variable represents the method that does the real work i.e. implements our business logic. Calling the Proceed method invokes it. We can do logging before or after our method is called depending on our needs. ****/**

```
public class LoggingInterceptor : IInterceptor
{
    #region IInterceptor Members

    public void Intercept(IInvocation invocation)
    {
        try
        {
```

```
            invocation.Proceed();
        }
    }
}
```

As you can see, the IInterceptor interface has only a single method, called Intercept}, that gets called whenever a method or property of the wrapped interface is called. Now we need only to instruct Windsor to intercept an object with this interceptor. The below class is created for the same purpose.

Create Class: -DependencyContainer.cs

```
/*** this class contains code to intercept an object with an interceptor: Windsor registers all the assemblies on PerWebRequest/LifestylePerWcfOperation ***/

Register:

public static void RegisterAllClassesFromAssembly(String a)
    {
Container.Register(AllTypes.FromAssemblyNamed(a).Pick().WithService.AllInterfaces().Lifestyle PerWcfOperation());
    }


Select methods to intercept:
 public class InterceptorsSettings : IModelInterceptorsSelector
  {
/** This is a simple example on how you can filter method to intercept, the HasInterceptors() can be implemented in this way.**/
    public Boolean HasInterceptors(Castle.Core.ComponentModel model)
    {
        String[] interceptClasses =
ConfigurationManager.AppSettings["InterceptClasses"].Split(';');
        foreach (var interceptClass in interceptClasses)
        {
            if (!String.IsNullOrWhiteSpace(interceptClass))
            {
                if (model.Name.EndsWith(interceptClass))
                {
                    return true;
                }
            }
        }
        return false;
    }

    public InterceptorReference[] SelectInterceptors(Castle.Core.ComponentModel model,
Castle.Core.InterceptorReference[] interceptors)
    {
```

```
        var intercept = new List<InterceptorReference>();
        String enableInstrumentation =
ConfigurationManager.AppSettings["EnableInstrumentation"];
        if (String.IsNullOrEmpty(enableInstrumentation) != true &&
            Convert.ToBoolean(enableInstrumentation, CultureInfo.InvariantCulture))
        {
            intercept.Add(InterceptorReference.ForType<LoggingInterceptor>());
        }
        return intercept.Concat(interceptors).ToArray();
    }
  }
```

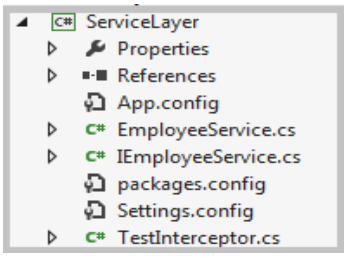
As you can verify you simply need to register the interceptor as LoggingInterceptor, telling castle that it implements the IInterceptor interface.In the above example I simply registered a concrete implementation for the IEmployeeService class, and I specify that I want to use the LoggingInterceptor registered interceptor to it.

Add Folder: Logger, **Create Classes** DefaultLogContext.cs & DefaultLoggers.cs

DefaultLogContext.cs:- This class contains LogPolicy, logsource and initialize() method to read from .config files.
DefaultLoggers.cs: This Class mainly contains File-path detail of .config for logging or tracing at exact path.

WCF Side Description:

Add a Project With below structure:

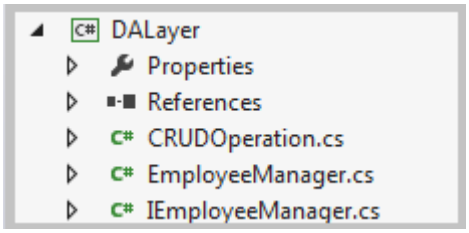


Create a simple Class EmployeeService.cs which holds the employee data and also create its simple interface IEmployeeService.cs.
Castle Windsor Interceptor generally initialized on startup of your application. In an ASP.NET application, global.asax is the mostly used place to initialize it, where as in WCF side you can create a constructor to invoke the DI. Here We have created a constructor called EmployeeService().

```
public EmployeeService()
    {
        var empCRUDOperation = DependencyContainer.GetInstance<IEmployeeManager>();
    }
```

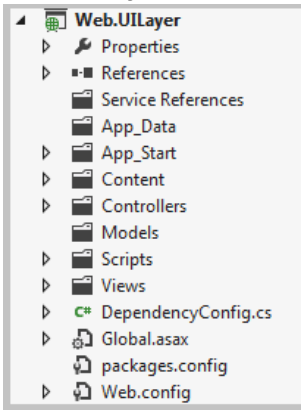
Create app.config File which holds all the AppSettings key-value pairs such as LogSource, LogPolicy, LogDirectory, EnableInstrumentation etc. And Add the Two file name anywhere like this: For Client Side Logging: "C:\LII.BOSS\Logs\", For WCF Side Logging: "C:\LII.BOSS\Logs1\". Note: Here, we are logging and tracing saving in file, but you can save in database, xml format etc.

Add a Project With below structure: This layer only involved to deal with data from service layer.



Client Side description:

Add a Project With below structure:



Create a Controller with name EmployeeController.cs and add Views for a particular action. Create a Class Name as DependencyConfig.cs where RegisterDependency() method resolve the dependency. GetControllerInstance and ReleaseController are the two override method which assure to resolve the dependency issues.

Register this RegisterDependency in global.asax file using below code.

```
protected void Application_Start()
{
    DependencyConfig.RegisterDependency();
}
```

Features, Advantages and Limitations

Features:

- Ability to work in a container agnostic way - this means your container can bootstrap all the code for you and you can take full advantage of its rich capabilities without ever referencing any of Castle.*.dll assemblies in your non-infrastructure assemblies.

- Very rich extensibility/extensions ecosystem which can give you really powerful capabilities and greatly reduce amount of plumbing code you have to write. You can tweak and twist it to do almost anything you may need.
- Windsor has a nice mix of usability, extensibility and integration modules.

- **Usability**: for example, you can use XML and/or code registration (it also has a fluent API like most containers nowadays).
- **Extensibility**: Lots of extension points that you can use to customize or override pretty much any default behavior.
- **Integration**: Windsor has lots of facilities (modules) that allow for easy integration with other frameworks/libraries. Other integrations include ASP.NET MVC, MonoRail, Workflow Foundation,NServiceBus, MassTransit, Rhino Service Bus, Quartz.Net, SolrNet, SolrSharp, Windows Fax Services.

Advantages:

- The usage of IOC container like Castle Windsor aids in decoupling dependencies. This decoupling makes for better unit test.
- Castle Windsor has features like Decorators, Interceptors, Proxies, as well as handling object lifestyle. With a container like Windsor, this enables you to do some hard things really simple, with extremely little coupling.

Limitations:

- The problem with proxy based AOP frameworks such as Castle Windsor is the fact that they are using proxies usually means that any method/property you wish to allow to be intercepted must be marked as virtual.

References and Links if any

Links:

For more information on Castle Windsor and AOP frameworks, please refer the following links –

1. http://docs.castleproject.org/ (Castle Windsor Documentation)
2. http://msdn.microsoft.com/en-us/library/ee658105.aspx (For cross cutting Concerns)
3. http://blog.andreloker.de/category/Castle.aspx (Basics Castle windsor examples)

Glossary

AOP – **Aspect Oriented programming.**