

پایاده سازی authorization به روش AOP به کمک کتابخانه های SNAP و StructureMap

عنوان:

نویسنده: کاوه شهبازی

تاریخ: ۱۷:۵۵ ۱۳۹۲/۰۸/۰۱

آدرس: www.dotnettips.info

برچسب‌ها: Authorization, AOP, SNAP, StructureMap

همانطور که پیشتر در [این مقاله](#) بحث شده است، بوسیله AOP می‌توان قابلیت‌هایی که قسمت عمده‌ای از برنامه را تحت پوشش قرار می‌دهند، کپسوله کرد. یکی از قابلیت‌هایی که در بخشهای مختلف یک سیستم نرم‌افزاری مورد نیاز است، Authorization یا اعتبارسنجی‌ست. در ادامه به بررسی یک پایاده‌سازی به این روش می‌پردازیم.

کتابخانه SNAP

[کتابخانه SNAP](#) به گفته سازنده آن، با یکپارچه‌سازی AOP با IoC Container های محبوب، برنامه‌نویسی به این سبک را ساده می‌کند. این کتابخانه هم اکنون علاوه بر structureMap از IoC Provider های LinFu, Ninject, Autofac و Castle Windsor نیز پشتیبانی میکند.

دریافت SNAP.StructureMap

برای دریافت آن نیاز است دستور پاورشل ذیل را در کنسول [نیوگت](#) ویزوال استودیو اجرا کنید:

```
PM> Install-Package snap.structuremap
```

پس از اجرای دستور فوق، کتابخانه SNAP.StructureMap که در زمان نگارش این مطلب نسخه 1.8.0 آن موجود است به همراه کليه نیازمندی‌های آن که شامل موارد زیر می‌باشد نصب خواهد شد.

```
StructureMap (≥ 2.6.4.1)
CommonServiceLocator.StructureMapAdapter (≥ 1.1.0.3)
SNAP (≥ 1.8)
fasterflect (≥ 2.1.2)
Castle.Core (≥ 3.1.0)
CommonServiceLocator (≥ 1.0)
```

تنظیمات SNAP

از آنجا که تنظیمات SNAP همانند تنظیمات StructureMap تنها باید یک بار اجرا شود، بهترین جا برای آن در یک برنامه وب، Application_Start فایل Global.asax است.

```
namespace Framework.UI.Asp
{
    public class Global : HttpApplication
    {
        void Application_Start(object sender, EventArgs e)
        {
            initSnap();
            initStructureMap();
        }

        private static void initSnap()
        {
            SnapConfiguration.For<StructureMapAspectContainer>(c =>
            {
                // Tell Snap to intercept types under the "Framework.ServiceLayer..." namespace.
                c.IncludeNamespace("Framework.ServiceLayer.*");
                // Register a custom interceptor (a.k.a. an aspect).
                c.Bind<Framework.ServiceLayer.Aspects.AuthorizationInterceptor>()
                  .To<Framework.ServiceLayer.Aspects.AuthorizationAttribute>();
            });
        }

        void Application_EndRequest(object sender, EventArgs e)
        {
            ObjectFactory.ReleaseAndDisposeAllHttpScopedObjects();
        }
    }
}
```

```

    }
    private static void initStructureMap()
    {
        var thread = StructureMap.Pipeline.Lifecycles.GetLifecycle(InstanceScope.HttpSession);
        ObjectFactory.Configure(x =>
        {
            x.For<IUserManager>().Use<EFUserManager>();
            x.For<IAuthorizationManager>().LifecycleIs(thread)
              .Use<EFAuthorizationManager>().Named("_AuthorizationManager");
            x.For<Framework.DataLayer.IUnitOfWork>()
              .Use<Framework.DataLayer.Context>();

            x.SetAllProperties(y =>
            {
                y.OfType<IUserManager>();
                y.OfType<Framework.DataLayer.IUnitOfWork>();
                y.OfType<Framework.Common.Web.IPageHelpers>();
            });
        });
    }
}

```

بخش اعظم کدهای فوق در مقاله‌های « [استفاده از StructureMap به عنوان یک IoC Container](#) » و « [تزریق خودکار وابستگی‌ها در برنامه‌های ASP.NET Web forms](#) » شرح داده شده‌اند، تنها بخش جدید متد `initSnap()` است، که خط اول آن به `snap` می‌گوید همه کلاس‌هایی که در فضای نام `Framework.ServiceLayer` و زیرمجموعه‌های آن هستند را پوشش دهد. خط دوم نیز کلاس `AuthorizationInterceptor` را به عنوان مرجعی برای `handle` کردن `AuthorizationAttribute` معرفی می‌کند.

در ادامه به بررسی کلاس `AuthorizationInterceptor` می‌پردازیم.

```

namespace Framework.ServiceLayer.Aspects
{
    public class AuthorizationInterceptor : MethodInterceptor
    {
        public override void InterceptMethod(IInvocation invocation, MethodBase method, Attribute attribute)
        {
            var AuthManager = StructureMap.ObjectFactory
                .GetInstance<Framework.ServiceLayer.UserManager.IAuthorizationManager>();
            var FullName = GetMethodFullName(method);
            if (!AuthManager.IsActionAuthorized(FullName))
                throw new Common.Exceptions.UnauthorizedAccessException("");

            invocation.Proceed(); // the underlying method call
        }

        private static string GetMethodFullName(MethodBase method)
        {
            var TypeName = (((System.Reflection.MemberInfo)(method)).DeclaringType).FullName;
            return TypeName + "." + method.Name;
        }
    }

    public class AuthorizationAttribute : MethodInterceptAttribute
    {
    }
}

```

کلاس مذکور از کلاس `MethodInterceptor` کتابخانه `snap` ارث بری کرده و متد `InterceptMethod` را تحریف می‌کند. این متد، کار اجرای متد اصلی ای که با این `Aspect` تزئین شده را بر عهده دارد. بنابراین می‌توان پیش از اجرای متد اصلی، اعتبارسنجی را انجام داد. **کلاس `MethodInterceptor`**

کلاس `MethodInterceptor` شامل چندین متد دیگر نیز هست که می‌توان برای سایر مقاصد از جمله مدیریت خطا و `Event` `logging` از آنها استفاده کرد.

```

namespace Snap
{

```

```
public abstract class MethodInterceptor : IAttributeInterceptor, IInterceptor, IHideBaseTypes
{
    protected MethodInterceptor();

    public int Order { get; set; }
    public Type TargetAttribute { get; set; }

    public virtual void AfterInvocation();
    public virtual void BeforeInvocation();
    public void Intercept(IInvocation invocation);
    public abstract void InterceptMethod(IInvocation invocation, MethodBase method, Attribute
attribute);
    public bool ShouldIntercept(IInvocation invocation);
}
}
```

یک نکته

نکته مهمی که در اینجا پیش می آید این است که برای اعتبارسنجی، کد کاربری شخصی که لاگین کرده، باید به طریقی در اختیار متد `IsActionAuthorized()` قرار بگیرد. برای این کار می توان در یک `HttpMudole` به عنوان مثال همان مازولی که برای تسهیل در کار تزریق خودکار وابستگی ها در سطح فرم ها استفاده می شود، با استفاده از امکانات `structureMap` به وهله ی ایجاد شده از `AuthorizationManager` (که با کمک `structureMap` با طول عمر `InstanceScope.HttpSession` ساخته شده است) دسترسی پیدا کرده و خاصیت مربوطه را مقداردهی کرد.

```
private void Application_PreRequestHandlerExecute(object source, EventArgs e)
{
    var page = HttpContext.Current.Handler as BasePage; // The Page handler
    if (page == null)
        return;

    WireUpThePage(page);
    WireUpAllUserControls(page);

    var Usrcod = HttpContext.Current.Session["Usrcod"];
    if (Usrcod != null)
    {
        var _AuthorizationManager = ObjectFactory
.GetNamedInstance<Framework.ServiceLayer.UserManager.IAuthorizationManager>("_AuthorizationManager");

        ((Framework.ServiceLayer.UserManager.EFAuthorizationManager)_AuthorizationManager)
.AuditUserId = Usrcod.ToString();
    }
}
```

روش استفاده

نحوه استفاده از Aspect تعریف شده در کد زیر قابل مشاهده است:

```
namespace Framework.ServiceLayer.UserManager
{
    public class EFUserManager : IUserManager
    {
        IUnitOfWork _uow;
        IDbSet<User> _users;

        public EFUserManager(IUnitOfWork uow)
        {
            _uow = uow;
            _users = _uow.Set<User>();
        }

        [Framework.ServiceLayer.Aspects.Authorization]
        public List<User> GetAll()
        {
            return _users.ToList<User>();
        }
    }
}
```

نظرات خوانندگان

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۸/۰۱ ۱۹:۷

با تشکر از شما. چند سؤال: متد AuthManager.IsActionAuthorized چگونه تعریف شده؟ و همچنین EFAuthorizationManager حدوداً چه تعریفی دارد؟

نویسنده: کاوه شهبازی
تاریخ: ۱۳۹۲/۰۸/۰۱ ۱۹:۵۱

۱- متد IsActionAuthorized نام کامل متدی که قرار است اجرا شود را به عنوان پارامتر گرفته و در دیتابیس (در این پیاده سازی به وسیله EntityFramework) چک میکند که کاربری که Id اش در AuthManager.AuditUserId است (یعنی کاربری که درخواست اجرای متد را داده است) اجازه اجرای این متد را دارد یا نه. بسته به نیازمندی برنامه شما این دسترسی میتواند به طور ساده فقط مستقیماً برای کاربر ثبت شود و یا ترکیبی از دسترسی خود کاربر و دسترسی گروه هایی که این کاربر در آن عضویت دارد باشد.

۲- EFAuthorizationManager کلاس ساده ایست

```
namespace Framework.ServiceLayer.UserManager
{
    public class EFAuthorizationManager : IAuthorizationManager
    {
        public String AuditUserId { get; set; }
        IUnitOfWork _uow;

        public EFAuthorizationManager(IUnitOfWork uow)
        {
            _uow = uow;
        }

        public bool IsActionAuthorized(string actionName)
        {
            var res = _uow.Set<User>()
                .Any(u => u.Id == AuditUserId &&
                    u.AllowedActions.Any(a => a.Name == actionName));
            return res;
        }

        public bool IsPageAuthorized(string pageURL)
        {
            //TODO: بررسی وجود دسترسی باید پیاده سازی شود
            //فقط برای تست
            return true;
        }
    }
}
```

خلاصه ای از کلاسهای مدل مرتبط را هم در زیر مشاهده می کنید

```
namespace Framework.DataModel
{
    public class User : BaseEntity
    {
        public string UserName { get; set; }
        public string Password { get; set; }

        //...

        [Display(Name = "عملیات مجاز")]
        public virtual ICollection<Action> AllowedActions { get; set; }
    }

    public class Action:BaseEntity
    {
        public string Name { get; set; }
        public Entity RelatedEntity { get; set; }
    }
}
```

```
        //...
        public virtual ICollection<User> AllowedUsers { get; set; }
    }
    public abstract class BaseEntity
    {
        [Key]
        public int Id { get; set; }
        //...
    }
}
```

چند روز پیش فرصتی پیش آمد تا بتوانم مروری بر مطلب منتشر شده درباره [AOP](#) داشته باشم. به حق مطلب مورد نظر، بسیار خوب و مناسب شرح داده شده بود و همانند سایر مقالات جناب نصیری چیزی کم نداشت. اما امروز قصد پیاده سازی یک مثال AOP، با استفاده از Microsoft Unity Application Block را به عنوان IOC Container دارم. اگر شما هم، مانند من از UnityContainer به عنوان IOC Container در پروژه‌های خود استفاده می‌کنید نگران نباشید. این کتابخانه به خوبی از مباحث Interception پشتیبانی می‌کند. در ادامه طی یک مقاله این مورد را با هم بررسی می‌کنیم.

برای دوستانی که با AOP آشنایی ندارند پیشنهاد می‌شود ابتدا [مطلب مورد نظر](#) را یک بار مطالعه نمایند. برای شروع یک پروژه در VS.Net بسازید و ارجاع به اسمبلی‌های زیر را در پروژه فراموش نکنید:

Microsoft.Practices.EnterpriseLibrary.Common«

Microsoft.Practices.Unity«

Microsoft.Practices.Unity.Configuration«

Microsoft.Practices.Unity.Interception«

Microsoft.Practices.Unity.Interception.Configuration«

یک اینترفیس به نام IMyOperation بسازید:

```
public interface IMyOperation
{
    void DoIt();
}
```

کلاسی می‌سازیم که اینترفیس بالا را پیاده سازی نماید:

```
public void DoIt()
{
    Console.WriteLine( "this is main block of code" );
}
```

قصد داریم با استفاده از AOP یک سری کد مورد نظر خود(در این مثال کد لاگ کردن عملیات در یک فایل مد نظر است) را به کدهای متدهای مورد نظر تزریق کنیم. یعنی با فراخوانی این متد کدهای لاگ عملیات در یک فایل ذخیره شود بدون تکرار یا فراخوانی دستی متد لاگ.

ابتدا یک کلاس برای لاگ عملیات می‌سازیم:

```
public class Logger
{
    const string path = @"D:\Log.txt";

    public static void WriteToFile( string methodName )
    {
        object lockObject = new object();
        if ( !File.Exists( path ) )
        {
            File.Create( path );
        }
        lock ( lockObject )
        {
            using ( TextWriter writer = new StreamWriter( path , true ) )
            {
                writer.WriteLine( string.Format( "{0} at {1}" , methodName , DateTime.Now ) );
            }
        }
    }
}
```

حال نیاز به یک Handler برای مدیریت فراخوانی کدهای تزریق شده داریم. برای این کار یک کلاس می‌سازیم که اینترفیس ICallHandler را پیاده سازی نماید.

```
public class LogHandler : ICallHandler
{
    public IMethodReturn Invoke( IMethodInvocation input , GetNextHandlerDelegate getNext )
    {
        Logger.WriteToFile( input.MethodBase.Name );
        var methodReturn = getNext()( input , getNext );
        return methodReturn;
    }
    public int Order { get; set; }
}
```

کلاس بالا یک متد به نام Invoke دارد که فراخوانی متدهای مورد نظر برای تزریق کد را در دست خواهد گرفت. در این متد ابتدا عملیات لاگ در فایل مورد نظر ثبت می‌شود (با استفاده از Logger.WriteToFile). سپس با استفاده از getNext که از نوع GetNextHandlerDelegate است، اجرا را به کدهای اصلی برنامه منتقل می‌کنیم.

```
var methodReturn = getNext()( input , getNext );
```

برای مدیریت بهتر عملیات لاگ یک Attribute می‌سازیم که فقط متد هایی که نیاز به لاگ کردن دارند را مشخص کنیم. به صورت زیر:

```
public class LogAttribute : HandlerAttribute
{
    public override ICallHandler CreateHandler( Microsoft.Practices.Unity.IUnityContainer container )
    {
        return new LogHandler();
    }
}
```

فقط دقت داشته باشید که کلاس مورد نظر به جای ارث بری از کلاس Attribute باید از کلاس HandlerAttribute که در فضای نام Microsoft.Practices.Unity.InterceptionExtension تعبیه شده است ارث برد (خود این کلاس از کلاس Attribute ارث برده است). کافایت در متد CreateHandler آن که Override شده است یک نمونه از کلاس LogHandler را برگشت دهیم. برای آماده سازی Ms Unity جهت عملیات Interception باید کدهای زیر در ابتدا برنامه قرار داده شود:

```
var unityContainer = new UnityContainer();
unityContainer.AddNewExtension<Interception>();
unityContainer.Configure<Interception>().SetDefaultInterceptorFor<IMyOperation>( new
InterfaceInterceptor() );
unityContainer.RegisterType<IMyOperation, MyOperation>();
```

توضیح چند مطلب:

بعد از نمونه سازی از کلاس UnityContainer باید Interception به عنوان یک Extension به این Container اضافه شود. سپس با استفاده از متد Configure برای اینترفیس IMyOperation یک Interceptor پیش فرض تعیین می‌کنیم. در پایان هم به وسیله متد RegisterType کلاس MyOperation به اینترفیس IMyOperation رجیستر می‌شود. از این پس هر گاه درخواستی برای اینترفیس IMyOperation از unityContainer شود یک نمونه از کلاس MyOperation در اختیار خواهیم داشت. به عنوان نکته آخر متد DoIt در اینترفیس بالا باید دارای LogAttribute باشد تا عملیات مزین سازی با کدهای لاگ به درستی انجام شود.

یک نکته تکمیلی:

در هنگام مزین سازی متد set خاصیت ها، به دلیل اینکه اینترفیسی برای این کار وجود ندارد باید مستقیما عملیات AOP به خود کلاس اعمال شود. برای این کار باید به صورت زیر عمل نمود:

```
var container = new UnityContainer();
container.RegisterType<Book , Book>();

container.AddNewExtension<Interception>();

var policy = container.Configure<Interception>().SetDefaultInterceptorFor<Book>( new
VirtualMethodInterceptor() ).AddPolicy( "MyPolicy" );

policy.AddMatchingRule( new PropertyMatchingRule( "*" , PropertyMatchingOption.Set ) );
policy.AddCallHandler<Handler.NotifyChangedHandler>();
```

همان طور که مشاهده می کنید عملیات Interception مستقیما برای کلاس پیکر بندی می شود و به جای InterfaceInterceptor از VirtualMethodInterceptor برای تزریق کد به بدنه متدها استفاده شده است. در پایان نیز با تعریف یک Policy می توانیم به راحتی (با استفاده از "*") متد Set تمام خواص کلاس را به NotifyChangedHandler مزین نماییم.

[سورس کامل مثال بالا](#)

هنگامی که از روش [AOP](#) استفاده می‌کنیم گاهی نیاز است متد تزئین‌شده را از متدی درون خود کلاس فراخوانی کنیم و می‌خواهیم aspectهای آن متد نیز فراخوانی شوند.

پیش‌نیاز: دوره‌ی AOP

(برای سادگی کار از تعریف attribute خودداری کردم. شما می‌توانید با توجه به آموزش، attributeهای دلخواه را به متدها بیافزایید).

Interface و کلاس پیاده‌سازی‌شده‌ی آن در لایه سرویس:

```
public interface IMyService
{
    void foo();
    void bar();
}

public class MyService : IMyService
{
    public void foo()
    {
        Console.WriteLine("foo");
        bar();
    }

    public void bar()
    {
        Console.WriteLine("bar");
    }
}
```

نام متد در خروجی نوشته می‌شود. همچنین می‌خواهیم پیش از فراخوانی این متدها، متنی در خروجی نوشته شود.

آماده‌سازی Interceptor

یک interceptor ساده که نام متد را در خروجی می‌نویسد.

```
//using Castle.DynamicProxy;

public class Interceptor : IInterceptor
{
    public void Intercept(IInvocation invocation)
    {
        Console.WriteLine("Intercepted: " + invocation.Method.Name);
        invocation.Proceed();
    }
}
```

معرفی Interceptor به سیستم

همانند قبل:

```
//using System;
//using Castle.DynamicProxy;
//using StructureMap;

class Program
{
    static void Main(string[] args)
    {
        ObjectFactory.Initialize(x =>
        {
            var dynamicProxy = new ProxyGenerator();
            x.For<IMyService>()
                .EnrichAllWith(myTypeInterface =>
                    dynamicProxy.CreateInterfaceProxyWithTarget(myTypeInterface, new
Intercept()))
                .Use<MyService>();
        });
    }
}
```

```

    });
    var myService = ObjectFactory.GetInstance<IMyService>();
    myService.foo();
}
}

```

انتظار ما این است که خروجی زیر تولید شود:

```

Intercepted foo
foo
Intercepted bar
bar

```

اما نتیجه این می‌شود که دلخواه ما نیست:

```

Intercepted foo
foo
bar

```

راه حل

برای حل این مشکل دو کار باید انجام داد:
1- متد تزئین‌شده باید **virtual** باشد.

```

public class MyService : IMyService
{
    public virtual void foo()
    {
        Console.WriteLine("foo");
        bar();
    }

    public virtual void bar()
    {
        Console.WriteLine("foo");
        bar();
    }
}

```

2- شیوه معرفی متد به سیستم باید به روش زیر باشد:

```

// جایگزین روش پیشین در متد
x.For<IMyService>()
    .EnrichAllWith(myTypeInterface => dynamicProxy.CreateClassProxy<MyService>(new
Intercept()))
    .Use<MyService>();

```

دلیل این مسئله به دو روش proxy برمی‌گردد که برای اطلاع بیشتر به [مستندات پروژه Castle](#) مراجعه کنید.
در این‌جا روش Inheritance-based به کار رفته است. در این روش، تنها متدهای **virtual** را می‌توان intercept کرد. در روش پیشین (Composition-based) برای تمامی متدها عملیات intercept انجام می‌شد (کلاس proxy پیاده‌سازی‌شده‌ی interface ما بود) که در این‌جا این‌گونه نیست و می‌تواند به سرعت برنامه کمک کند.