عنوان: Dependency Injection در Asp.Net WebApi (روش اول)
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۳/۱۱/۱۵ ۱:۲۵
آدرس: www.dotnettips.info
گروه‌ها: ASP.NET Web API, Dependency Injection, Castle Windsor

طی این پست با تزریق وابستگی‌ها در Asp.net MVC آشنا شدید. روش ذکر شده در آن برای کنترلرهای Web Api جوابگو نیست و باید از روش‌های دیگری برای این منظور استفاده نماییم.

نکته 1: برای پیاده سازی این مثال‌ها، Castle Windsor به عنوان IOC Container انتخاب شده است. بدیهی است می‌توانید از Ioc Container مورد نظر خود نیز بهره ببرید.
نکته 2 : می‌توانید از مقاله [هاست سرویس‌های Web Api با استفاده از OWIN و TopShelf] جهت هاست سرویس‌های web Api خود استفاده نمایید.

روش اول

اگر قبلا در این زمینه جستجو کرده باشید، به احتمال زیاد با مفهوم IDependencyResolver بیگانه نیستید. درباره استفاده از این روش مقالات متعددی نوشته شده است؛ حتی در مثال‌های موجود در خود سایت MSDN نیز این روش را مرسوم دانسته و آن را به اشتراک می‌گذارند. جهت نمونه می‌توانید این پروژه را دانلود کرده و کدهای آن را بررسی کنید.
در این روش، قدم اول، ساخت یک کلاس و پیاده سازی اینترفیس IDependencyResolver می‌باشد؛ به صورت زیر:

```
public class ApiDependencyResolver : IDependencyResolver
    {
        public ApiDependencyResolver(IWindsorContainer container)
        {
            Container = container;
        }

        public IWindsorContainer Container
        {
            get;
            private set;
        }

        public object GetService(Type serviceType)
        {
            try
            {
                return Container.Kernel.HasComponent(serviceType) ? Container.Resolve(serviceType) :
null;
            }
            catch (Kernel.ComponentNotFoundException)
            {
                return null;
            }
        }

        public IEnumerable<object> GetServices(Type serviceType)
        {
            try
            {
                return Container.ResolveAll(serviceType).Cast<object>();
            }
            catch (Kernel.ComponentNotFoundException)
            {
                return Enumerable.Empty<object>();
            }
        }
        public IDependencyScope BeginScope()
        {
            return new SharedDependencyResolver(Container);
        }

        public void Dispose()
        {
            Container.Dispose();
        }
```

```
    }
```

اینترفیس IDependencyResolver از اینترفیس دیگری به نام IDependencyScope ارث می‌برد که دارای دو متد اصلی به نام‌های GetService و GetServices است که جهت وهله سازی کنترلرها استفاده می‌شوند. با فراخوانی این متدها، نمونه‌ی ساخته شده توسط Container بازگشت داده خواهد شد.

کاربرد متد BeginScope چیست ؟

کنترلرها به صورت (Per Request) بر اساس هر درخواست وهله سازی خواهند شد. جهت مدیریت چرخه‌ی عمر کنترلرها و منابع در اختیار آن‌ها، از متد BeginScope استفاده می‌شود. به این صورت که نمونه‌ی اصلی DependencyResolver در هنگام شروع برنامه به GlobalConfiguration پروژه Attach خواهد شد. سپس به ازای هر درخواست، جهت وهله سازی Controllerها، متد GetService از محدوده داخلی (منظور فراخوانی متد BeginScope است) باعث ایجاد نمونه و بعد از اتمام فرآیند، متد Dispose باعث آزاد سازی منابع موجود خواهد شد.
پیاده سازی متد BeginScope وابسته به IocContainer مورد استفاده شما است. در این جا کلاس SharedDependencyResolver را به صورت زیر پیاده سازی کردم:

```
public class SharedDependencyResolver : IDependencyScope
    {
        public SharedDependencyResolver(IWindsorContainer container)
        {
            Container = container;
            Scope = Container.BeginScope();
        }

        public IWindsorContainer Container
        {
            get;
            private set;
        }

        public IDisposable Scope
        {
            get;
            private set;
        }

        public object GetService(Type serviceType)
        {
            try
            {
                return Container.Kernel.HasComponent(serviceType) ? Container.Resolve(serviceType) :
null;
            }
            catch (ComponentNotFoundException)
            {
                return null;
            }
        }

        public IEnumerable<object> GetServices(Type serviceType)
        {
            try
            {
                return Container.ResolveAll(serviceType).Cast<object>();
            }
            catch (ComponentNotFoundException)
            {
                return null;
            }
        }

        public void Dispose()
        {
            Scope.Dispose();
        }
    }
```

اگر از UnityContainer استفاده می‌کنید کافیست تکه کد زیر را جایگزین کلاس بالا نمایید:

```
public IDependencyScope BeginScope()
{
    var child = container.CreateChildContainer();
    return new ApiDependencyResolver(child);
}
```

برای جستجوی خودکار کنترلرها و رجیستر کردن آن‌ها به برنامه Windsor امکانات جالبی را در اختیار ما قرار می‌دهد. ابتدا یک Installer ایجاد می‌کنیم:

```
public class KernelInstaller : IWindsorInstaller
    {
        public void Install(IWindsorContainer container, IConfigurationStore store)
        {

container.Register(Classes.FromThisAssembly().BasedOn<ApiController>().LifestyleTransient());
            container.Kernel.Resolver.AddSubResolver(new CollectionResolver(container.Kernel, true));
        }
    }
```

در پایان در کلاس Startup نیز کافیست مراحل زیر را انجام دهید:

»ابتدا Installer نوشته شده را به WindsorContainer معرفی نمایید.

»DependencyResolver نوشته شده را به HttpConfiguration معرفی کنید.

»عملیات Routing مورد نظر را ایجاد و سپس config مورد نظر را در اختیار appBuilder قرار دهید.

```
public class Startup
    {
        public void Configuration( IAppBuilder appBuilder )
        {
           var container = new WindsorContainer();
            container.Install(new KernelInstaller());

            var config = new HttpConfiguration
            {
                 DependencyResolver = new ApiDependencyResolver(container)
            };

            config.MapHttpAttributeRoutes();

            config.Routes.MapHttpRoute(
                name: "Default" ,
                routeTemplate: "{controller}/{action}/{name}" ,
                defaults: new { name = RouteParameter.Optional }
            );

            config.EnsureInitialized();

            appBuilder.UseWebApi( config );
        }
    }
```

نکته: این روش به دلیل استفاده از الگوی ServiceLocator و همچنین نداشتن Context درخواست ها روشی منسوخ شده می‌باشد که طی این مقاله جناب نصیری به صورت کامل به این مبحث پرداخته اند.

نظرات خوانندگان

نویسنده: هادی احمدی
تاریخ: ۱۳۹۳/۱۱/۱۵ ۱۰:۴۱

سلام
خسته نباشید، سپاس از مطلب مفیدتون

نقد هایی بر استفاده از DependencyResolver وارد هست که یکی از آنها نداشتن Context هنگام Resolve کردن وابستگی هاست.
(برای اطلاعات بیشتر به این پست از آقای Mark Seeman نویسنده‌ی کتاب Dependency Injection in .NET مراجعه کنید)
به همین دلیل (و دلایل دیگر مثل انعطاف پذیری کم و ...) استفاده از CotrollerActivator نسبت به این روش پیشنهاد می‌شود
که مطمئنا شما در سری‌های بعدی این مقاله به آن خواهید پرداخت. بنده هم در این مقاله در مورد استفاده از
ControllerActivator برای پیاده سازی DI نوشته ام.
موفق باشید

نویسنده: میثم شریفی
تاریخ: ۱۳۹۴/۰۲/۱۱ ۱۵:۵۲

مطلب شما را مطالعه کردم. تنها ایرادی که می‌توان اشاره کرد Implement کردن دستی تک تک سرویسها در Windsor Container می‌باشد. آیا راه حلی برای این موضوع هست ؟ (مطمئنا در پروژه هایی با تعداد سرویس و کنترل زیاد مشکل ایجاد می‌کند)

نویسنده: هادی احمدی
تاریخ: ۱۳۹۴/۰۲/۱۱ ۱۸:۵۷

سلام
بله اتفاقا Windsor امکانات بسیار جالبی برای Register کردن دسته ای وابستگی‌ها دارد. اگر دقت کنید در آخر مقاله هم من از این امکانات برای ثبت دسته ای تمام Controllerهای موجود در Assembly فوق استفاده کرده ام. منتها به علت ساده بودن مثال و وجود تنها یک Service، از ثبت دسته ای سرویس‌ها در مثال خودداری کردم.
مستندات Windsor برای ثبت دسته ای وابستگی‌ها : http://docs.castleproject.org/Windsor.Registering-components-by-conventions.ashx