عنوان: **استفاده از SimpleIoc به عنوان یک IoC Container**
نویسنده: مهدی ع
تاریخ: ۱۳۹۲/۰۳/۱۱ ۱۶:۲۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns, MVVM, Dependency Injection, SimpleIoc, Ioc, mvvm light

SimpleIoc به صورت پیش فرض در پروژه های MVVM Light موجود می‌باشد. قطعه کد پایین به صورت پیش فرض در پروژه‌های MVVM Lightایجاد می‌شود.
در کلاس ViewModelLocator ما تمام میانجی(Interface)ها و اشیا(Objects)ی مورد نیازمان را ثبت(register) می‌کنیم.
در ادامه اجزای مختلف آن را شرح می‌دهیم.

```
class ViewModelLocator
{
    static ViewModelLocator()
    {
        ServiceLocator.SetLocatorProvider(() => SimpleIoc.Default);
        if (ViewModelBase.IsInDesignModeStatic)
        {
            SimpleIoc.Default.Register<IDataService, Design.DesignDataService>();
        }
        else
        {
            SimpleIoc.Default.Register<IDataService, DataService>();
        }
        SimpleIoc.Default.Register<MainViewModel>();
        SimpleIoc.Default.Register<SecondViewModel>();
    }

    public MainViewModel Main
    {
        get
        {
            return ServiceLocator.Current.GetInstance<MainViewModel>();
        }
    }
}
```

1) هر شیء که به صورت پیش فرض ایجاد می‌شود با الگوی Singlton ایجاد می‌شود.

```
SimpleIoc.Default.GetInstance<MainViewModel>(Guid.NewGuid().ToString());
```

2) جهت ثبت یک کلاس مرتبط با میانجی آن از روش زیر استفاده می‌شود.

```
SimpleIoc.Default.Register<IDataService, Design.DesignDataService>();
```

3) جهت ثبت یک شی مرتبط با میانجی از روش زیر استفاده می‌شود.

```
SimpleIoc.Default.Register<IDataService>(myObject);
```

4) جهت ثبت یک نوع (Type) به طریق زیر عمل می‌کنیم.

```
SimpleIoc.Default.Register<MainViewModel>();
```

5) جهت گرفتن وهله (Instance) از یک میانجی خاص، از روش زیر استفاده می‌کنیم.

```
SimpleIoc.Default.GetInstance<IDataService>();
```

6) جهت گرفتن وهله ای به صورت مستقیم، 'ایجاد و وضوح وابستگی(dependency resolution)' از روش زیر استفاد می‌کنیم.

```
SimpleIoc.Default.GetInstance();
```

7) برای ایجاد داده‌های زمان طراحی از روش زیر استفاده می‌کنیم.

```
if (ViewModelBase.IsInDesignModeStatic)
        {
            SimpleIoc.Default.Register<IDataService, Design.DesignDataService>();
        }
        else
        {
            SimpleIoc.Default.Register<IDataService, DataService>();
        }
```

در حالت زمان طراحی، سرویس‌های زمان طراحی به صورت خودکار ثبت می‌شوند. و می‌توان این داده‌ها را در ViewModelها و Viewها حین طراحی مشاهده نمود.

منبع

عنوان: **استفاده از الگوی Adapter در تزریق وابستگی‌ها**
نویسنده: محمد رضا منشادی
تاریخ: ۱۳۹۲/۰۸/۱۹ ۱۸:۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns, Dependency Injection, Ioc, Dependency Inversion

در بعضی از مواقع ممکن است که در هنگام استفاده از اصل تزریق وابستگی‌ها، با یک مشکل روبرو شویم و آن این است که اگر از کلاسی استفاده می‌کنیم که به سورس آن دسترسی نداریم، نمی‌توانیم برای آن یک Interface تهیه کنیم و اصل (Depend on abstractions, not on concretions) از بین می‌رود، حال چه باید کرد.
برای اینکه موضوع تزریق وابستگی‌ها (DI) به صورت کامل در قسمتهای دیگر سایت توضیح داده شده است، دوباره آن را برای شما بازگو نمی‌کنیم .
لطفا به کدهای ذیل توجه کنید:

کد بدون تزریق وابستگی ها

به سازنده کلاس ProductService و تهیه یک نمونه جدید از وابستگی مورد نیاز آن دقت نمائید:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Web;

namespace ASPPatterns.Chap2.Service
{
    public class Product
    {
    }

    public class ProductRepository
    {
        public IList<Product> GetAllProductsIn(int categoryId)
        {
            IList<Product> products = new List<Product>();
            // Database operation to populate products …
            return products;
        }
    }

    public class ProductService
    {
        private ProductRepository _productRepository;
        public ProductService()
        {
            _productRepository = new ProductRepository();
        }

        public IList<Product> GetAllProductsIn(int categoryId)
        {
            IList<Product> products;
            string storageKey = string.Format("products_in_category_id_{0}", categoryId);
            products = (List<Product>)HttpContext.Current.Cache.Get(storageKey);
            if (products == null)
            {
                products = _productRepository.GetAllProductsIn(categoryId);
                HttpContext.Current.Cache.Insert(storageKey, products);
            }
            return products;
        }
    }
}
```

همان کد با تزریق وابستگی

```
using System;
using System.Collections.Generic;
```

```
namespace ASPPatterns.Chap2.Service
{
    public interface IProductRepository
    {
        IList<Product> GetAllProductsIn(int categoryId);
    }

    public class ProductRepository : IProductRepository
    {
        public IList<Product> GetAllProductsIn(int categoryId)
        {
            IList<Product> products = new List<Product>();
            // Database operation to populate products …
            return products;
        }
    }

    public class ProductService
    {
        private IProductRepository _productRepository;
        public ProductService(IProductRepository  productRepository)
        {
            _productRepository = productRepository;
        }

        public IList<Product> GetAllProductsIn(int categoryId)
        {
            //…
        }
    }
}
```

همانطور که ملاحظه می‌کنید به علت دسترسی به سورس، به راحتی برای استفاده از کلاس ProductRepository در کلاس ProductService، از تزریق وابستگی‌ها استفاده کرده‌ایم.

اما از این جهت که شما دسترسی به سورس Http context class را ندارید، نمی‌توانید به سادگی یک Interface را برای آن ایجاد کنید و سپس یک تزریق وابستگی را مانند کلاس ProductRepository برای آن تهیه نمائید.

خوشبختانه این مشکل قبلا حل شده است و الگویی که به ما جهت پیاده سازی آن کمک کند، وجود دارد و آن الگوی آداپتر (Adapter Pattern) می‌باشد.

این الگو عمدتا برای ایجاد یک Interface از یک کلاس به صورت یک Interface سازگار و قابل استفاده می‌باشد. بنابراین می‌توانیم این الگو را برای تبدیل HTTP Context caching API به یک API سازگار و قابل استفاده به کار ببریم.

در ادامه می‌توان Interface سازگار جدید را در داخل productservice که از اصل تزریق وابستگی‌ها (DI) استفاده می‌کند تزریق کنیم.

یک اینترفیس جدید را با نام ICacheStorage به صورت ذیل ایجاد می‌کنیم:

```
public interface ICacheStorage
{
    void Remove(string key);
    void Store(string key, object data);
    T Retrieve<T>(string key);
}
```

حالا که شما یک اینترفیس جدید دارید، می‌توانید کلاس produceservic را به شکل ذیل به روز رسانی کنید تا از این اینترفیس، به جای HTTP Context استفاده کند.

```
public class ProductService
{
    private IProductRepository _productRepository;
    private ICacheStorage _cacheStorage;
    public ProductService(IProductRepository  productRepository,
    ICacheStorage cacheStorage)
    {
        _productRepository = productRepository;
        _cacheStorage = cacheStorage;
```

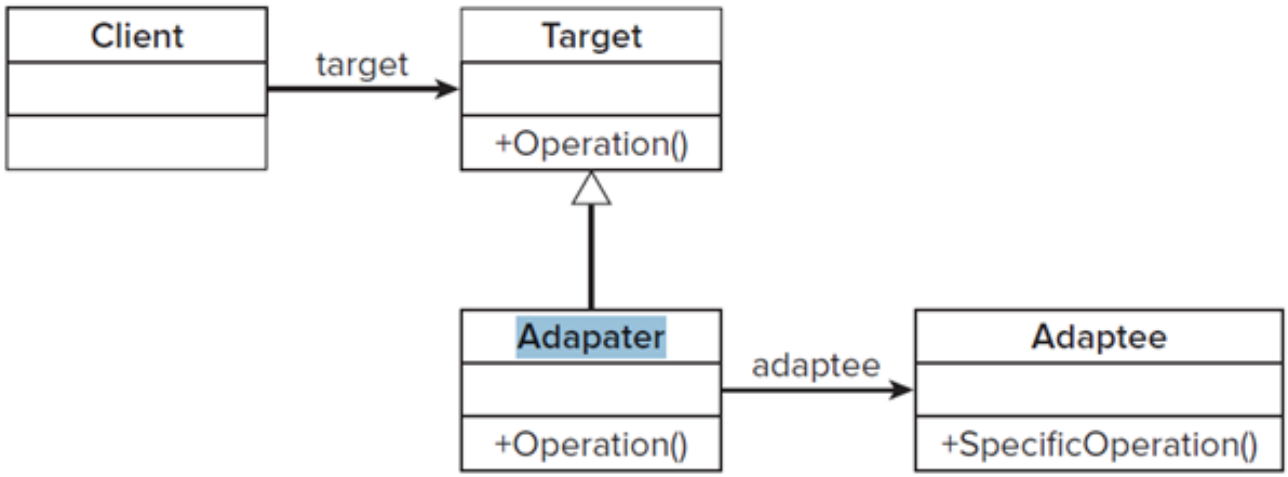
```
    }

    public IList<Product> GetAllProductsIn(int categoryId)
    {
        IList<Product> products;
        string storageKey = string.Format("products_in_category_id_{0}", categoryId);
        products = _cacheStorage.Retrieve<List<Product>>(storageKey);
        if (products == null)
        {
            products = _productRepository.GetAllProductsIn(categoryId);
            _cacheStorage.Store(storageKey, products);
        }
        return products;
    }
}
```

مسئله ای که در اینجا وجود دارد این است که HTTP Context Cache API صریحا نمی‌تواند Interface ایی که ما ایجاد کرده‌ایم را اجرا کند.

پس چگونه الگوی Adapter می‌تواند به ما کمک کند تا از این مشکل خارج شویم؟

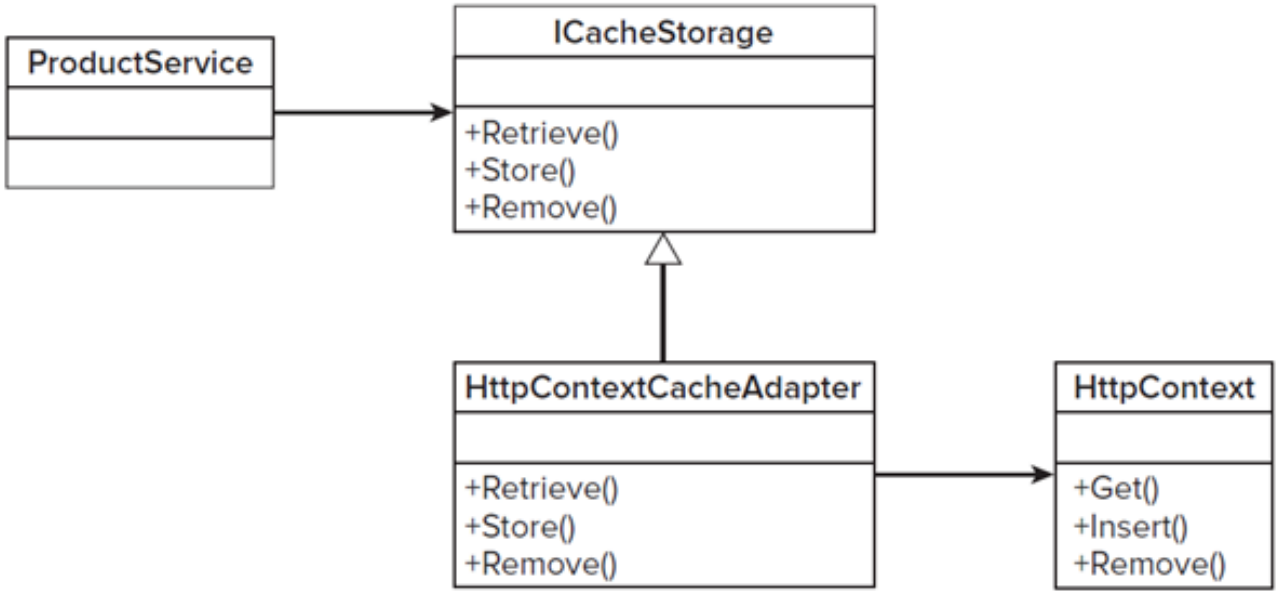
هدف این الگو به صورت ذیل در GOF مشخص شده است .«تبدیل Interface از یک کلاس به یک Interface مورد انتظار Client»

تصویر ذیل، مدل این الگو را به کمک UML نشان می‌دهد:



همانطور که در این تصویر ملاحظه می‌کنید، یک Client ارجاعی به یک Abstraction در تصویر (Target) دارد (ICacheStorage در کد نوشته شده). کلاس Adapter اجرای Target را بر عهده دارد و به سادگی متدهای Interface را نمایندگی می‌کند. در اینجا کلاس Adapter، یک نمونه از کلاس Adaptee را استفاده می‌کند و در هنگام اجرای قراردادهای Target، از این نمونه استفاده خواهد کرد.

اکنون کلاس‌های خود را در نمودار UML قرار می‌دهیم که به شکل ذیل آنها را ملاحظه می‌کنید.



در شکل ملاحظه می‌نمایید که یک کلاس جدید با نام HttpContextCacheAdapter مورد نیاز است. این کلاس یک کلاس روکش (محصور کننده یا Wrapper) برای متدهای HTTP Context cache است. برای اجرای الگوی Adapter کلاس HttpContextCacheAdapter را به شکل ذیل ایجاد می‌کنیم:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Web;
namespace ASPPatterns.Chap2.Service
{
    public class HttpContextCacheAdapter : ICacheStorage
    {
        public void Remove(string key)
        {
            HttpContext.Current.Cache.Remove(key);
        }

        public void Store(string key, object data)
        {
            HttpContext.Current.Cache.Insert(key, data);
        }

        public T Retrieve<T>(string key)
        {
            T itemStored = (T)HttpContext.Current.Cache.Get(key);
            if (itemStored == null)
                itemStored = default(T);
            return itemStored;
        }
    }
}
```

حال به سادگی می‌توان یک caching solution دیگر را پیاده سازی کرد بدون اینکه در کلاس ProductService اثر یا تغییری ایجاد کند .

منبع : Professional Asp.Net Design Pattern

عنوان: **آموزش LightInject IoC Container - قسمت 1**
نویسنده: میثم خوشبخت
تاریخ: ۱۳۹۳/۰۲/۰۸ ۱۶:۳۰
آدرس: www.dotnettips.info
گروه‌ها: Design patterns, Dependency Injection, Ioc, Dependency Inversion, LightInject

LightInject در حال حاضر یکی از قدرتمندترین IoC Containerها است که از لحاظ سرعت و کارآیی در بالاترین جایگاه در میان IoC Containerهای موجود قرار دارد. جهت بررسی کارایی IoC Containerها می‌توانید به این لینک مراجعه کنید . LightInject یک IoC Container فوق العاده سبک وزن می‌باشد که تمامی قابلیت‌های متداولی که از یک Service Container انتظار می‌رود را شامل می‌شود. تنها شامل یک فایل .cs می‌باشد که تمامی کدهای آن در همین یک فایل نوشته شده‌اند. در پروژه‌های کوچک تا بزرگ بدون از دست دادن کارآیی، با بالاترین سرعت ممکن عمل تزریق وابستگی را انجام می‌دهد. در این مجموعه مقالات به بررسی کامل این IoC Container می‌پردازیم و تمامی قابلیت‌های آن را آموزش می‌دهیم.

نحوه نصب و راه اندازی LightInject

در پنجره Package Manager Console می‌توانید با نوشتن دستور ذیل، نسخه باینری آن را نصب کنید که به فایل .dll آن Reference میدهد.

```
PM> Install-Package LightInject
```

همچنین می‌توانید توسط دستور ذیل فایل .cs آن را به پروژه اضافه نمایید.

```
PM> Install-Package LightInject.Source
```

آماده سازی پروژه نمونه

قبل از شروع کار با LightInject، یک پروژه Windows Forms Application را با ساختار کلاس‌های ذیل ایجاد نمایید. (در مقالات بعدی و پس از آموزش کامل LightInject نحوه استفاده از آن را در ASP.NET MVC نیز آموزش می‌دهیم)

```
public class PersonModel
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Family { get; set; }
        public DateTime Birth { get; set; }
    }

    public interface IRepository<T> where T:class
    {
        void Insert(T entity);
        IEnumerable<T> FindAll();
    }

    public interface IPersonRepository:IRepository<PersonModel>
    {
    }

    public class PersonRepository:IPersonRepository
    {
        public void Insert(PersonModel entity)
        {
            throw new NotImplementedException();
        }

        public IEnumerable<PersonModel> FindAll()
        {
            throw new NotImplementedException();
        }
    }

    public interface IPersonService
    {
        void Insert(PersonModel entity);
        IEnumerable<PersonModel> FindAll();
```

```
    }

    public class PersonService:IPersonService
    {
        private readonly IPersonRepository _personRepository;

        public PersonService(IPersonRepository personRepository)
        {
            _personRepository = personRepository;
        }

        public void Insert(PersonModel entity)
        {
            _personRepository.Insert(entity);
        }

        public IEnumerable<PersonModel> FindAll()
        {
            return _personRepository.FindAll();
        }
    }
```

توضیحات PersonModel: ساختار داده ای جدول Person در سمت Application، که در لایه Domain Model ایجاد می‌گردد. **توجه:** جهت سهولت تست و تسریع کدنویسی از لایه بندی و از کلاس‌های ViewModel استفاده نکردیم. **IRepository:** یک Interface عمومی برای تمامی Interfaceهای مربوط به Repository که عملیات مربوط به پایگاه داده مثل بروزرسانی و واکشی اطلاعات را انجام می‌دهند. **IPersonRepository:** واسط بین لایه Service و لایه Repository می‌باشد. **PersonRepository:** پیاده سازی واقعی عملیات مربوط به پایگاه داده برای PersonModel می‌باشد. به کلاسهایی که حاوی پیاده سازی واقعی کد می‌باشند Concrete Class می‌گویند. **IPersonService:** واسط بین رابط کاربری و لایه سرویس می‌باشد. رابط کاربری به جای دسترسی مستقیم به PersonService از IPersonService استفاده می‌کند. **PersonService:** دریافت درخواست‌های رابط کاربری و بررسی قوانین تجاری، سپس ارسال درخواست به لایه Repository در صورت صحت درخواست، و در نهایت ارسال پاسخ دریافتی به رابط کاربری. در واقع واسطی بین Repository و UI می‌باشد.

پس از ایجاد ساختار فوق کد مربوط به Form1 را بصورت زیر تغییر دهید.

```
public partial class Form1 : Form
    {
        private readonly IPersonService _personService;
        public Form1(IPersonService personService)
        {
            _personService = personService;
            InitializeComponent();
        }
    }
```

توضیحات

در کد فوق به منظور ارتباط با سرویس از IPersonService استفاده نمودیم که به عنوان پارامتر ورودی برای سازنده Form1 تعریف شده است. حتما با Dependency Inversion و انواع Dependency Injection آشنا هستید که به سراغ مطالعه این مقاله آمدید و علت این نوع کدنویسی را هم می‌دانید. بنابراین توضیح بیشتری در این مورد نمی‌دهم.

حال اگر برنامه را اجرا کنید در Program.cs با خطای عدم وجود سازنده بدون پارامتر برای Form1 مواجه می‌شوید که کد آن را باید به صورت زیر تغییر می‌دهیم.

```
static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            var container = new ServiceContainer();
            container.Register<IPersonService, PersonService>();
            container.Register<IPersonRepository, PersonRepository>();
            Application.Run(new Form1(container.GetInstance<IPersonService>()));
        }
```

توضیحات

کلاس ServiceContainer وظیفه‌ی Register کردن یک کلاس را برای یک Interface دارد. زمانی که می‌خواهیم Form1 را نمونه سازی نماییم و Application را راه اندازی کنیم، باید نمونه ای را از جنس IPersonService ایجاد نموده و به سازنده‌ی Form1 ارسال نماییم. با رعایت اصل DIP، نمونه سازی واقعی یک کلاس لایه دیگر، نباید در داخل کلاس‌های لایه جاری انجام شود. برای این منظور از شیء container استفاده نمودیم و توسط متد GetInstance، نمونه‌ای از جنس IPersonService را ایجاد نموده و به

Form1 پاس دادیم. حال container از کجا متوجه می‌شود که چه کلاسی را برای IPersonService نمونه سازی نماید؟
در خطوط قبلی توسط متد Register، کلاس PersonService را برای IPersonService ثبت نمودیم. container نیز برای نمونه سازی به کلاس هایی که برایش Register نمودیم مراجعه می‌نماید و نمونه سازی را انجام می‌دهد. جهت استفاده از PersonService به پارامتر ورودی IPersonRepository برای سازنده‌ی آن نیاز داریم که کلاس PersonRepository را برای IPersonRepository ثبت کردیم.
حال اگر برنامه را اجرا کنید، به درستی اجرا خواهد شد. برنامه را متوقف کنید و به کد موجود در Program.cs مراجعه نموده و دو خط مربوط به Register را Comment نمایید. سپس برنامه را اجرا کنید و خطای تولید شده را ببینید. این خطا بیان می‌کند که امکان نمونه سازی برای IPersonService را ندارد. چون قبلا هیچ کلاسی را برای آن Register نکرده ایم. **Named Services**
در برخی مواقع، بیش از یک کلاس وجود دارند که ممکن است از یک Interface ارث بری نمایند. در این حالت و در زمان Register، باید به ServiceContainer بگوییم که کدام کلاس را باید نمونه سازی نماید. برای بررسی این موضوع، کلاسهای زیر را به ساختار پروژه اضافه نمایید.

```
public class WorkerModel:PersonModel
    {
        public ManagerModel Manager { get; set; }
    }

    public class ManagerModel:PersonModel
    {
        public IEnumerable<WorkerModel> Workers { get; set; }
    }

    public class WorkerRepository:IPersonRepository
    {
        public void Insert(PersonModel entity)
        {
            throw new NotImplementedException();
        }

        public IEnumerable<PersonModel> FindAll()
        {
            throw new NotImplementedException();
        }
    }

    public class ManagerRepository:IPersonRepository
    {
        public void Insert(PersonModel entity)
        {
            throw new NotImplementedException();
        }

        public IEnumerable<PersonModel> FindAll()
        {
            throw new NotImplementedException();
        }
    }

    public class WorkerService:IPersonService
    {
        private readonly IPersonRepository _personRepository;

        public WorkerService(IPersonRepository personRepository)
        {
            _personRepository = personRepository;
        }

        public void Insert(PersonModel entity)
        {
            var worker = entity as WorkerModel;
            _personRepository.Insert(worker);
        }

        public IEnumerable<PersonModel> FindAll()
        {
            return _personRepository.FindAll();
        }
    }

    public class ManagerService:IPersonService
    {
        private readonly IPersonRepository _personRepository;
```

```
        public ManagerService(IPersonRepository personRepository)
        {
            _personRepository = personRepository;
        }

        public void Insert(PersonModel entity)
        {
            var manager = entity as ManagerModel;
            _personRepository.Insert(manager);
        }

        public IEnumerable<PersonModel> FindAll()
        {
            return _personRepository.FindAll();
        }
    }
```

توضیحات

دو کلاس Manager و Worker به همراه سرویس‌ها و Repository هایشان اضافه شده اند که از IPersonService و IPersonRepository مشتق شده اند.

حال کد کلاس Program را به صورت زیر تغییر می‌دهیم

```
...
 var container = new ServiceContainer();
            container.Register<IPersonService, PersonService>();
            container.Register<IPersonService, WorkerService>();
            container.Register<IPersonRepository, PersonRepository>();
            container.Register<IPersonRepository, WorkerRepository>();
            Application.Run(new Form1(container.GetInstance<IPersonService>()));
```

توضیحات

در کد فوق، چون WorkerService بعد از PersonService ثبت یا Register شده است، LightInject در زمان ارسال پارامتر به Form1، نمونه ای از کلاس WorkerService را ایجاد میکند. اما اگر بخواهیم از کلاس PersonService نمونه سازی نماید باید کد را به صورت زیر تغییر دهیم.

```
...
            container.Register<IPersonService, PersonService>("PersonService");
            container.Register<IPersonService, WorkerService>();
            container.Register<IPersonRepository, PersonRepository>();
            container.Register<IPersonRepository, WorkerRepository>();
            Application.Run(new Form1(container.GetInstance<IPersonService>("PersonService")));
```

همانطور که مشاهده می‌نمایید، در زمان Register نامی را به آن اختصاص دادیم که در زمان نمونه سازی از این نام استفاده شده است.

اگر در زمان ثبت، نامی را به نمونه‌ی مورد نظر اختصاص داده باشیم، و فقط یک Register برای آن Interface معرفی نموده باشیم، در زمان نمونه سازی، LightInject آن نمونه را به عنوان سرویس پیش فرض در نظر می‌گیرد.

```
container.Register<IPersonService, PersonService>("PersonService");
  Application.Run(new Form1(container.GetInstance<IPersonService>()));
```

در کد فوق، چون برای IPersonService فقط یک کلاس برای نمونه سازی معرفی شده است، با فراخوانی متد GetInstance، حتی بدون ذکر نام، نمونه ای را از کلاس PersonService ایجاد می‌کند. **<IEnumerable<T>**

زمانی که چند کلاس را که از یک Interface مشتق شده اند، با هم Register می‌نمایید، LightInject این قابلیت را دارد که این کلاس‌های Register شده را در قالب یک لیست شمارشی برگردانید.

```
container.Register<IPersonService, PersonService>();
            container.Register<IPersonService, WorkerService>("WorkerService");
            var personList = container.GetInstance<IEnumerable<IPersonService>>();
```

در کد فوق لیستی با دو آیتم ایجاد می‌شود که یک آیتم از نوع PersonService و دیگری از نوع WorkerService می‌باشد. همچنین از کد زیر نیز می‌توانید استفاده کنید:

```
container.Register<IPersonService, PersonService>();
            container.Register<IPersonService, WorkerService>("WorkerService");
            var personList = container.GetAllInstances<IPersonService>();
```

به جای متد GetInstance از متد GetAllInstances استفاده شده است.

LightInject از Collectionهای زیر نیز پشتیبانی می‌نماید:

Array

<ICollection<T

<IList<T

<IReadOnlyCollection<T

<IReadOnlyList<T

Values توسط LightInject می‌توانید مقادیر ثابت را نیز تعریف کنید

```
container.RegisterInstance<string>("SomeValue");
            var value = container.GetInstance<string>();
```

متغیر value با رشته "SomeValue" مقداردهی می‌گردد. اگر چندین ثابت رشته ای داشته باشید می‌توانید نام جداگانه ای را به هر کدام اختصاص دهید و در زمان فراخوانی مقدار به آن نام اشاره کنید.

```
container.RegisterInstance<string>("SomeValue","String1");
            container.RegisterInstance<string>("OtherValue","String2");
            var value = container.GetInstance<string>("String2");
```

متغیر value با رشته "OtherValue" مقداردهی می‌گردد.

نظرات خوانندگان

نویسنده: احمد زاده
تاریخ: ۱۳۹۳/۰۲/۲۱ ۱:۲۷

ممنون از مطلب خوبتون
من یه مقایسه دیگه دیدم که اونجا گفته بود Ligth Inject از Instance Per Request پشتیبانی نمی‌کنه
میخواستم جایگزین Unity کنم برای حالتی که unit of work داریم و DBContext for per request اگر راهنمایی کنید، ممنون میشم

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۲/۲۱ ۱۰:۳۲

از حالت طول عمر PerRequestLifetime پشتیبانی می‌کند.

نویسنده: میثم خوشبخت
تاریخ: ۱۳۹۳/۰۲/۲۱ ۱۱:۱۴

خواهش می‌کنم
همانطور که آقای نصیری نیز عنوان کردند، از PerRequestLifeTime استفاده می‌شود که در مقاله بعدی در مورد آن صحبت خواهم کرد.

عنوان: **استفاده از Unity در پیاده سازی الگوی Service locator**
نویسنده: میثم نوایی
تاریخ: ۱۳۹۳/۰۷/۱۷ ۲۳:۵۰
آدرس: www.dotnettips.info
گروه‌ها: Dependency Injection, UnityContainer, Ioc, Service Factory, Service locator

یکی از راهکارهای پیاده سازی IOC یا همان Inversion Of Control در پروژه‌های MVC استفاده از Unity و معرفی آن به DependencyResolver خود دات نت است.
برای آشنایی با Unity و قابلیت‌های آن میتوانید به اینجا و اینجا سر بزنید.
اما برای استفاده از Unity در پروژه‌های MVC کافی است در Global یا فایل راه انداز (bootstrapper) تک تک انتزاع‌ها (Interface) را به کلاس‌های مرتبط شان معرفی کنید.

```
var container = new UnityContainer();
```

```
container.RegisterType<ISomeService, SomeService>(new PerRequestLifetimeManager());
container.RegisterType<ISomeBusiness, SomeBusiness>(new PerRequestLifetimeManager());
container.RegisterType<ISomeController, SomeController>(new PerRequestLifetimeManager());
```

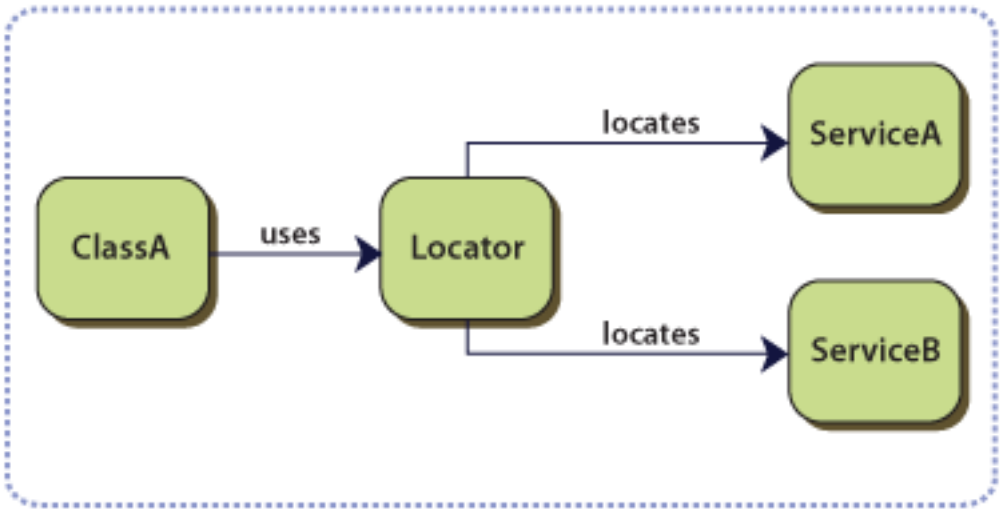
و بعد از ایجاد container از نوع UnityContainer میتوانیم آنرا به MVC معرفی کنیم:

```
DependencyResolver.SetResolver(new UnityDependencyResolver(container));
```

تا به اینجا به‌راحتی میتوانید از سرویس‌های معرفی شده در پروژه MVC استفاده کنید.

```
var someService=(ISomeService)DependencyResolver.Current.GetService(typeof(ISomeService));
var data=someService.GetData();
```

اما اگر بخواهیم از کلاس‌های معرفی شده در Unity در لایه‌های دیگر (مثلا Business) استفاده کنیم چه باید کرد؟
برای هر این مشکل راهکارهای متفاوتی وجود دارد. من در لایه سرویس از Service locator بهره برده ام. برای آشنایی با این الگو اینجا را بخوانید. اکثر برنامه نویسان الگوهای IOC و Service Locator را با هم اشتباه میگیرند یا آنها را اشتباها بجای هم بکار میبرند.
برای درک تفاوت الگوی IOC و Service locator اینجا را بخوانید.



در لایه سرویس یک کلاس Service Factory داریم که قرار است همه سرویس‌ها، برای برقراری ارتباط با یکدیگر از آن استفاده

کنند.این کلاس معمولا در لایه سرویس به اشکال گوناگونی پیاده سازی میشود که کارش وهه سازی از Interfaceهای درخواستی است. اما برای یکپارچه کردن آن با Unity من آنرا به شکل زیر پیاده سازی کرده ام

```
public class ServiceFactory : MarshalByRefObject
    {
        static IUnityContainer uContainer = new UnityContainer();
        public static Type DataContextType { get; set; }

        public static void Initialise(IUnityContainer unityContainer, Type dbContextType)
        {
            uContainer = unityContainer;
            DataContextType = dbContextType;
            uContainer.RegisterType(typeof(BaseDataContext), DataContextType, new
HierarchicalLifetimeManager());
        }
        public static T Create<T>()
        {
            return (T)Activator.CreateInstance<T>();
        }
        public static T Create<T>(string fullTypeName)
        {
            return (T)System.Reflection.Assembly.GetExecutingAssembly().CreateInstance(fullTypeName);
        }
        public static T Create<T>(Type entityType)
        {
            return (T)Activator.CreateInstance(entityType);
        }
        public static dynamic Create(Type entityType)
        {
            return Activator.CreateInstance(entityType);
        }

        public static T Get<T>()
        {
            return uContainer.Resolve<T>();
        }
        public static object Get(Type type)
        {
            return uContainer.Resolve(type);
        }
    }
```

در این کلاس ما بجای ایجاد داینامیک آبجکت‌ها، از Unity استفاده کرده‌ایم. در همان ابتدا که برنامه‌ی وب ما برای اولین بار اجرا میشود و بعد از Register کردن کلاس‌ها، می‌توانیم container را به صورت پارامتر سازنده به کلاس Service Factory ارسال کنیم. به این ترتیب برای استفاده از سرویس‌ها در لایه Business از Unity بهره میبریم.

البته استفاده از Unity برای DataContext خیلی منطقی نیست و بهتر است نوع DataContext را در ابتدا بگیریم و هرجا نیاز داشتیم با استفاده از متد Create از آن وهله سازی بکنیم.