

Managed Extensibility Framework یا **MEF** کامپوننتی از Framework 4 است که برای ایجاد برنامه‌های توسعه پذیر (Extensible) با حجم کم کد استفاده می‌شود. این تکنولوژی به برنامه نویسان این امکان رو می‌ده که توسعه‌های (Extension) برنامه رو بدون پی‌کربندی استفاده کنند. همچنین به توسعه دهندگان این اجازه رو می‌ده که به آسانی کدها رو کپسوله کنند.

MEF به عنوان بخشی از .NET 4 و Silverlight 4 معرفی شد. MEF یک راه حل ساده برای مشکل توسعه در حال اجرای برنامه‌ها ارائه می‌کند. تا قبل از این تکنولوژی، هر برنامه‌ای که می‌خواست یک مدل Plugin را پشتیبانی کنه لازم بود که خودش زیر ساخت‌ها را از ابتدا ایجاد کنه. این Plugin‌ها اغلب برای برنامه‌های خاصی بودند و نمی‌توانستند در پیاده سازی‌های چندگانه دوباره استفاده شوند. ولی MEF در راستای حل این مشکلات، روش استاندارد رو برای میزبانی برنامه‌های کاربردی پیاده کرده است.

برای فهم بهتر مفاهیم یک مثال ساده رو با MEF پیاده سازی می‌کنم.

ابتدا یک پروژه از نوع Console Application ایجاد کنید. بعد با استفاده از Add Reference یک ارجاع به

System.ComponentModel.Composition بدید. سپس یک Interface به نام IViewModel را به صورت زیر ایجاد کنید:

```
public interface IViewModel
{
    string Name { get; set; }
}
```

یک خاصیت به نام Name برای دسترسی به نام ViewModel ایجاد می‌کنیم.

سپس 2 تا ViewModel دیگه ایجاد می‌کنیم که IViewModel را پیاده سازی کنند. به صورت زیر:

:ViewModelFirst

```
[Export( typeof( IViewModel ) )]
public class ViewModelFirst : IViewModel
{
    public ViewModelFirst()
    {
        this.Name = "ViewModelFirst";
    }

    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }
    private string _name;
}
```

:ViewModelSecond

```
[Export( typeof( IViewModel ) )]
public class ViewModelSecond : IViewModel
{
    public ViewModelSecond()
    {
```

```

        this.Name = "ViewModelSecond";
    }

    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }
    private string _name;
}

```

Export Attribute استفاده شده در بالای کلاس‌های ViewModel به این معنی است که این کلاس‌ها اینترفیس IViewModel رو Export کردند تا در جای مناسب بتونیم این ViewModel ها Import کنیم. (Import , Export از مفاهیم اصلی در MEF هستند) حالا نوبت به پیاده سازی کلاس Plugin می‌رسه.

```

public class PluginManager
{
    public PluginManager()
    {
    }

    public IList<IViewModel> ViewModels
    {
        get
        {
            return _viewModels;
        }
        private set
        {
            _viewModels = value;
        }
    }

    [ImportMany( typeof( IViewModel ) )]
    private IList<IViewModel> _viewModels = new List<IViewModel>();

    public void SetupManager()
    {
        AggregateCatalog aggregateCatalog = new AggregateCatalog();

        CompositionContainer container = new CompositionContainer( aggregateCatalog );

        CompositionBatch batch = new CompositionBatch();

        batch.AddPart( this );

        aggregateCatalog.Catalogs.Add( new AssemblyCatalog( Assembly.GetExecutingAssembly() ) );

        container.Compose( batch );
    }
}

```

کلاس PluginManager برای شناسایی و استفاده از کلاس‌هایی که صفتهای Export رو دارند نوشته شده (دقیقا شبیه یک UnityContainer در Microsoft Unity Application Block یا IKernal در Ninject) عمل می‌کنه با این تفاوت که نیازی به Register یا Bind کردن ندارند)

ابتدا یک لیست از کلاس‌هایی که IViewModel رو Export کردند داریم.

بعد در متد SetupManager ابتدا یک AggregateCatalog نیاز داریم تا بتونیم Composition Part ها رو بهش اضافه کنیم. به کد زیر توجه کنید:

```
aggregateCatalog.Catalogs.Add( new AssemblyCatalog( Assembly.GetExecutingAssembly() ) );
```

تو این قطعه کد من یک Assembly Catalog رو که به Assembly جاری برنامه اشاره می‌کنه به AggregateCatalog اضافه کردم. متد batch.AddPart(this) در واقع به این معنی است که به MEF گفته می‌شود این کلاس ممکن است شامل Export هایی باشد که به یک یا چند Import وابستگی دارند.

متد AddExport(this) در CompositionBatch به این معنی است که این کلاس ممکن است شامل Export هایی باشد که به Import وابستگی ندارند.

حالا برای مشاهده نتایج کد زیر را در کلاس Program اضافه می‌کنیم:

```
static void Main( string[] args )
{
    PluginManager plugin = new PluginManager();

    Console.WriteLine( string.Format( "Number Of ViewModels Before Plugin Setup Is [ {0} ]",
plugin.ViewModels.Count ) );

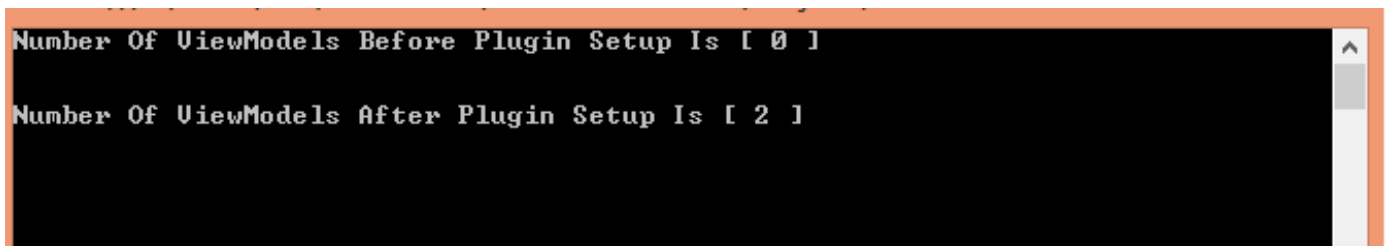
    Console.WriteLine( Environment.NewLine );

    plugin.SetupManager();

    Console.WriteLine( string.Format( "Number Of ViewModels After Plugin Setup Is [ {0} ]",
plugin.ViewModels.Count ) );

    Console.ReadLine();
}
```

در کلاس بالا ابتدا تعداد کلاس‌های موجود در لیست ViewModels رو قبل از Setup کردن Plugin نمایش داده سپس بعد از Setup کردن Plugin دوباره تعداد کلاس‌های موجود در لیست ViewModel رو مشاهده می‌کنیم. که خروجی به شکل زیر تولید خواهد شد.



The screenshot shows a console window with a black background and orange border. It displays two lines of text in a monospaced font: "Number Of ViewModels Before Plugin Setup Is [0]" followed by a blank line, and then "Number Of ViewModels After Plugin Setup Is [2]". A vertical scrollbar is visible on the right side of the console window.

متد SetupManager در کلاس Plugin (با توجه به AggregateCatalog) که در این برنامه فقط Assembly جاری رو بهش اضافه کردیم تمام کلاس‌هایی رو که نوع IViewModel رو Export کردند پیدا کرده و در لیست اضافه می‌کنه (این کار رو با توجه به ImportMany Attribute) انجام میده. در پست‌های بعدی روش استفاده از MEF رو در Prism یا WAF توضیح می‌دم.

نظرات خوانندگان

نویسنده: MehRad

تاریخ: ۱۱:۴۷ ۱۳۹۱/۱۱/۲۵

با تشکر از مطلب خوبتون

اگر امکان داره استفاده از MEF رو در ASP.NET MVC هم توضیح بدید

نویسنده: مسعود م. پاکدل

تاریخ: ۱۳:۲۳ ۱۳۹۱/۱۱/۲۵

ممنون.

بله حتما در پست‌های بعدی در مورد MEF و استفاده اون در WAF (WPF Application Framework) و MVC و Prism توضیحاتی رو خواهد داد.

نویسنده: علیرضا پایدار

تاریخ: ۱۳:۸ ۱۳۹۲/۰۶/۲۶

ممنون مفید بود.

توی Ninject میتونستیم مشخص کنیم یک پلاگین وابسته به پلاگین دیگه باشه. این کار در MEF به چه شکلی انجام میگیرد؟

نویسنده: مسعود پاکدل

تاریخ: ۱۳:۴۷ ۱۳۹۲/۰۶/۲۶

MEF برای پیاده سازی مبحث Chaining Dependencies از مفهوم Contract در Export Attribute استفاده می‌کند. پارامتر اول در Export برای ContractName است. به صورت زیر:

```
[Export( "ModuleA" , typeof( IMyInterface) )]
public class ClassA : IMyInterface
{
}

[Export( "ModuleB" , typeof( IMyInterface))]
public class ClassB : IMyInterface
{
}
```

در نتیجه در هنگام Import کردن کلاس‌های بالا باید حتما ContractName آن‌ها را نیز مشخص کنیم:

```
public class ModuleA
{
    [ImportingConstructor]
    public ModuleA([ImportMany( "ModuleA" , IMyInterface)] IEnumerable<IMyInterface> controllers )
    {
    }
}
```

با استفاده از ImportMany Attribute و ContractName به راحتی می‌توانیم تمام آبجکت‌ها Export شده در هر ماژول را تفکیک کرد.

در این پست قصد دارم روش استفاده از ServiceLocator رو به وسیله یک مثال ساده بهتون نمایش بدم. Microsoft Unity روش توصیه شده Microsoft برای پایه سازی Dependency Injection و ServiceLocator Pattern است. یک ServiceLocator در واقع وظیفه تهیه Instance‌های مختلف از کلاس‌ها رو برای پایه سازی Dependency Injection بر عهده داره. برای شروع یک پروژه از نوع Console Application ایجاد کنید و یک ارجاع به Assembly‌های زیر رو در برنامه قرار بدید.

Microsoft.Practices.ServiceLocation

Microsoft.Practices.Unity

Microsoft.Practices.EnterpriseLibrary.Common

اگر Assembly‌های بالا رو در اختیار ندارید می‌تونید اون‌ها رو از [اینجا](#) دانلود کنید. Microsoft Enterprise Library یک کتابخانه تهیه شده توسط شرکت Microsoft است که شامل موارد زیر است و بعد از نصب می‌تونید در قسمت‌های مختلف برنامه از اون‌ها استفاده کنید.

Enterprise Library Caching Application Block : یک CacheManager قدرتمند در اختیار ما قرار می‌ده که می‌تونید از اون برای کش کردن داده‌ها استفاده کنید.

Enterprise Library Exception Handling Application Block : یک کتابخانه مناسب و راحت برای پایه سازی یک Exception Handler در برنامه‌ها است.

Enterprise Library Loggin Application Block : برای تهیه یک Log Manager در برنامه استفاده می‌شود.

Enterprise Library Validation Application Block : برای اجرای Validation برای Entity‌ها با استفاده از Attribute می‌تونید از این قسمت استفاده کنید.

Enterprise Library DataAccess Application Block : یک کتابخانه قدرتمند برای ایجاد یک DataAccess Layer است با Performance بسیار بالا.

Enterprise Library Shared Library : برای استفاده از تمام موارد بالا در پروژه باید این Dll رو هم به پروژه Reference بدید. چون برای همشون مشترک است.

برای اجرای مثال ابتدا کلاس زیر رو به عنوان مدل وارد کنید.

```
public class Book
{
    public string Title { get; set; }
    public string ISBN { get; set; }
}
```

حالا باید Repository مربوطه رو تهیه کنید. ابتدا یک Interface به صورت زیر ایجاد کنید.

```
public interface IBookRepository
{
    List<Book> GetBooks();
}
```

سپس کلاسی ایجاد کنید که این Interface رو پیاده سازی کنه.

```
public class BookRepository : IBookRepository
{
    public List<Book> GetBooks()
    {
        List<Book> listOfBooks = new List<Book>();

        listOfBooks.AddRange( new Book[]
        {
            new Book(){Title="Book1" , ISBN="123"},
            new Book(){Title="Book2" , ISBN="456"},
            new Book(){Title="Book3" , ISBN="789"},
            new Book(){Title="Book4" , ISBN="321"},
            new Book(){Title="Book5" , ISBN="654"},
        } );

        return listOfBooks;
    }
}
```

کلاس BookRepository یک لیست از Book رو ایجاد میکنه و اونو برگشت میده. در مرحله بعد باید Service مربوطه برای استفاده از این Repository ایجاد کنید. ولی باید Repository رو به Constructor این کلاس Service پاس بدید. اما برای انجام این کار باید از ServiceLocator استفاده کنیم.

```
public class BookService
{
    public BookService()
        : this( ServiceLocator.Current.GetInstance<IBookRepository>() )
    {
    }

    public BookService( IBookRepository bookRepository )
    {
        this.BookRepository = bookRepository;
    }

    public IBookRepository BookRepository
    {
        get;
        private set;
    }

    public void PrintAllBooks()
    {
        Console.WriteLine( "List Of All Books" );

        BookRepository.GetBooks().ForEach( ( Book item ) =>
        {
            Console.WriteLine( item.Title );
        } );
    }
}
```

همان طور که می بینید این کلاس دو تا Constructor داره که در حالت اول باید یک IBookRepository رو به کلاس پاس داد و در حالت دوم ServiceLocator این کلاس رو برای استفاده دز اختیار سرویس قرار میده. متد Print هم تمام کتابهای مربوطه رو برامون چاپ می کنه. در مرحله آخر باید ServiceLocator رو تنظیم کنید. برای این کار کدهای زیر رو در کلاس Program قرار بدید.

```
class Program
{
    static void Main( string[] args )
    {
        IUnityContainer unityContainer = new UnityContainer();

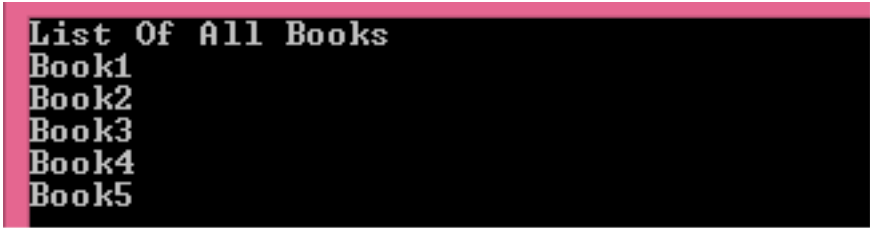
        unityContainer.RegisterType<IBookRepository, BookRepository>();

        ServiceLocator.SetLocatorProvider( () => new UnityServiceLocator( unityContainer ) );

        BookService service = new BookService();
    }
}
```

```
        service.PrintAllBooks();  
        Console.ReadLine();  
    }  
}
```

در این کلاس ابتدا یک `UnityContainer` ایجاد کردم و اینترفیس `IBookRepository` رو به کلاس `BookRepository` Register کردم تا هر جا که به `IBookRepository` نیاز داشتم یک Instance از کلاس `BookRepository` ایجاد بشه. در خط بعدی `ServiceLocator` برنامه رو ست کردم و برای این کار از کلاس `UnityServiceLocator` استفاده کردم. بعد از اجرای برنامه خروجی زیر قابل مشاهده است.



```
List Of All Books  
Book1  
Book2  
Book3  
Book4  
Book5
```

نظرات خوانندگان

نویسنده: sunn

تاریخ: ۱۱:۱۶ ۱۳۹۳/۰۳/۲۰

سلام اول از همه ممنون بابت این همه تلاش، دوم چرا بدون این همه کد نویسی نمیایم از یه دستور linq ساده استفاده کنیم، نامها رو بگیریم و با یک Foreach ساده پاس بدیم و این همه راه رفتیم و الان MVC از این روشها و استفاده از اینترفیسها و تزریقات وابستگی و ... که من نمیدونم این تزریقات وابستگی چیه استفاده میکنیم ، ممنون میشم توضیح بدین یا به جایی ارجاء بدین منو

نویسنده: وحید نصیری

تاریخ: ۱۱:۴۲ ۱۳۹۳/۰۳/۲۰

جهت مطالعه مباحث مقدماتی تزریق وابستگی‌ها، مراجعه کنید به دوره « [بررسی مفاهیم معکوس سازی وابستگی‌ها و ابزارهای مرتبط با آن](#) ».

عنوان: ایجاد ServiceLocator با استفاده از Ninject

نویسنده: مسعود پاکدل

تاریخ: ۲۰:۵ ۱۳۹۱/۱۲/۰۴

آدرس: www.dotnettips.info

برچسب‌ها: Dependency Injection, ServiceLocator, Ninject

در [پست قبلی](#) روش استفاده از ServiceLocator رو با استفاده از Microsoft Unity بررسی کردیم. در این پست قصد داریم همون مثال رو با استفاده از Ninject پیاده سازی کنیم. Ninject ابزاری برای پیاده سازی Dependency Injection در پروژه‌های دات نت است که کار کردن با اون واقعا راحت. برای شروع کلاس‌های Book و BookRepository و BookService و اینترفیس IBookRepository از این [پست](#) دریافت کنید.

حالا با استفاده از NuGet باید ServiceLocator رو برای Ninject دریافت کنید. برای این کار در Package Manager Console دستور زیر رو وارد کنید.

```
PM> Install-Package CommonServiceLocator.NinjectAdapter
```

بعد از دانلود و نصب Reference‌های زیر به پروژه اضافه می‌شوند.

Ninject

NinjectAdapter

Microsoft.Practices.ServiceLocation

اگر دقت کنید برای ایجاد ServiceLocator داریم از Enterprise Library:ServiceLocator استفاده می‌کنیم. ولی برای این کار به جای استفاده از UnityServiceLocator باید از NinjectServiceLocator استفاده کنیم.

ابتدا

برای پیاده سازی مثال قبل در کلاس Program کدهای زیر رو وارد کنید.

```
using System;
using Ninject;
using NinjectAdapter;
using Microsoft.Practices.ServiceLocation;

namespace ServiceLocatorPattern
{
    class Program
    {
        static void Main( string[] args )
        {
            IKernel kernel = new StandardKernel();

            kernel.Bind<IBookRepository>().To<BookRepository>();

            ServiceLocator.SetLocatorProvider( () => new NinjectServiceLocator( kernel ) );

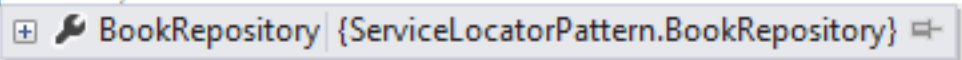
            BookService service = new BookService();

            service.PrintAllBooks();

            Console.ReadLine();
        }
    }
}
```

همان طور که می‌بینید روال انجام کار دقیقا مثل قبل هست فقط Syntax کمی متفاوت شده. برای مثال به جای استفاده از IUnityContainer از IKernel استفاده کردم و به جای دستور RegisterType از دستور Bind استفاده شده. در نهایت هم ServiceLocator به یک NinjectServiceLocator ای که IKernel رو دریافت کرده ست شد. به تصویر زیر دقت کنید.

```
public BookService()
{
    BookRepository = ServiceLocator.Current.GetInstance<IBookRepository>();
}
```

A tooltip is shown for the `BookRepository` property access. It contains a plus icon, a wrench icon, the text `BookRepository`, and a list of available types: `{ServiceLocatorPattern.BookRepository}`.

همان طور که می‌بینید در هر جای پروژه که نیاز به یک Instance از یک کلاس داشته باشید می‌تونید با استفاده از ServiceLocator این کار خیلی راحت انجام بدید.

بعد از اجرای پروژه خروجی دقیقا مانند مثال قبل خواهد بود.

عنوان: آزمون واحد در MVVM به کمک تزریق وابستگی

نویسنده: شاهین کیاست

تاریخ: ۱۷:۰ ۱۳۹۲/۰۱/۰۴

آدرس: www.dotnettips.info

برچسب‌ها: MVVM, Unit testing, Dependency Injection

یکی از خوبی‌های استفاده از Presentation Pattern ها بالا بردن تست پذیری برنامه و در نتیجه نگهداری کد می‌باشد. MVVM الگوی محبوب برنامه نویسان WPF و Silverlight می‌باشد. به صرف استفاده از الگوی MVVM نمی‌توان اطمینان داشت که ViewModel کاملاً تست پذیری داریم. به عنوان مثلاً اگر در ViewModel خود مستقیماً DialogBox کنیم یا ارجاعی از View دیگری داشته باشیم نوشتن آزمون‌های واحد تقریباً غیر ممکن می‌شود. قبلاً درباره‌ی این مشکلات و راه حل آن مطلب در سایت منتشر شده است :

[- MVVM و نمایش دیالوگ‌ها](#)

در این مطلب قصد داریم سناریویی را بررسی کنیم که ViewModel از Background Worker جهت انجام عملیات مانند دریافت داده‌ها استفاده می‌کند.

Background Worker کمک می‌کند تا اعمال طولانی در یک Thread دیگر اجرا شود در نتیجه رابط کاربری Freeze نمی‌شود.

به این مثال ساده توجه کنید :

```
public class BackgroundWorkerViewModel : BaseViewModel
{
    private List<string> _myData;

    public BackgroundWorkerViewModel()
    {
        LoadDataCommand = new RelayCommand(OnLoadData);
    }

    public RelayCommand LoadDataCommand { get; set; }

    public List<string> MyData
    {
        get { return _myData; }
        set
        {
            _myData = value;
            RaisePropertyChanged(() => MyData);
        }
    }

    public bool IsBusy { get; set; }

    private void OnLoadData()
    {
        var backgroundWorker = new BackgroundWorker();
        backgroundWorker.DoWork += (sender, e) =>
        {
            MyData = new List<string> {"Test"};
            Thread.Sleep(1000);
        };
        backgroundWorker.RunWorkerCompleted += (sender, e) => { IsBusy = false; };
        backgroundWorker.RunWorkerAsync();
    }
}
```

در این ViewModel با اجرای دستور LoadDataCommand داده‌ها از یک منبع داده دریافت می‌شود. این عمل می‌تواند چند ثانیه طول بکشد ، در نتیجه برای قفل نشدن رابط کاربر این عمل را به کمک Background Worker به صورت Async در پشت صحنه انجام شده است.

آزمون واحد این ViewModel اینگونه خواهد بود :

[TestFixture]

```
public class BackgroundWorkerViewModelTest
{
    #region Setup/Teardown

    [SetUp]
    public void Setup()
    {
        _backgroundWorkerViewModel = new BackgroundWorkerViewModel();
    }

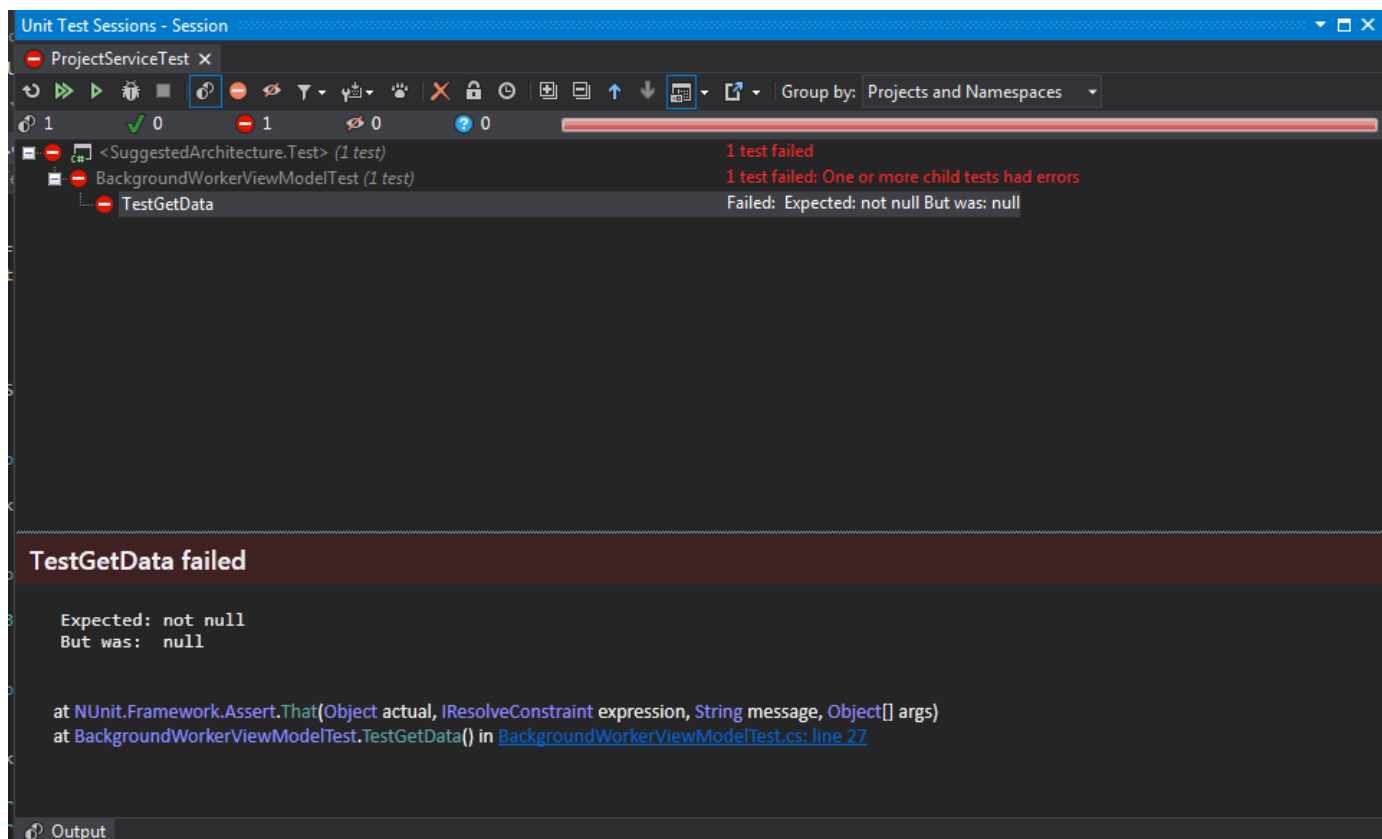
    #endregion

    private BackgroundWorkerViewModel _backgroundWorkerViewModel;

    [Test]
    public void TestGetData()
    {
        _backgroundWorkerViewModel.LoadDataCommand.Execute(_backgroundWorkerViewModel);

        Assert.NotNull(_backgroundWorkerViewModel.MyData);
        Assert.IsNotEmpty(_backgroundWorkerViewModel.MyData);
    }
}
```

با اجرای این آزمون واحد نتیجه با آن چیزی که در زمان اجرا رخ می‌دهد متفاوت است و با وجود صحیح بودن کدها آزمون واحد شکست می‌خورد. چون Unit Test به صورت همزمان اجرا می‌شود و برای عملیات‌های پشت صحنه صبر نمی‌کند در نتیجه این آزمون واحد شکست می‌خورد.



یک راه حل تزریق BackgroundWorker به صورت وابستگی به ViewModel می‌باشد. همانطور که قبلاً اشاره شده یکی از مزایای استفاده از تکنیک‌های [تزریق وابستگی](#) سهولت Unit testing می‌باشد.

در نتیجه یک Interface عمومی و 2 پیاده سازی همزمان و غیر همزمان جهت استفاده در برنامه‌ی واقعی و آزمون واحد تهیه می‌کنیم :

```
public interface IWorker
{
    void Run(DoWorkEventHandler doWork);
    void Run(DoWorkEventHandler doWork, RunWorkerCompletedEventHandler onComplete);
}
```

جهت استفاده در برنامه‌ی واقعی :

```
public class AsyncWorker : IWorker
{
    public void Run(DoWorkEventHandler doWork)
    {
        Run(doWork, null);
    }

    public void Run(DoWorkEventHandler doWork, RunWorkerCompletedEventHandler onComplete)
    {
        var backgroundWorker = new BackgroundWorker();
        backgroundWorker.DoWork += doWork;
        if (onComplete != null)
            backgroundWorker.RunWorkerCompleted += onComplete;
        backgroundWorker.RunWorkerAsync();
    }
}
```

جهت اجرا در آزمون واحد :

```
public class SyncWorker : IWorker
{
    #region IWorker Members

    public void Run(DoWorkEventHandler doWork)
    {
        Run(doWork, null);
    }

    public void Run(DoWorkEventHandler doWork, RunWorkerCompletedEventHandler onComplete)
    {
        Exception error = null;
        var doWorkEventArgs = new DoWorkEventArgs(null);
        try
        {
            doWork(this, doWorkEventArgs);
        }
        catch (Exception ex)
        {
            error = ex;
            throw;
        }
        finally
        {
            onComplete(this, new RunWorkerCompletedEventArgs(doWorkEventArgs.Result, error,
doWorkEventArgs.Cancel));
        }
    }

    #endregion
}
```

در نتیجه ViewModel اینگونه تغییر خواهد کرد :

```
public class BackgroundWorkerViewModel : BaseViewModel
{
    private readonly IWorker _worker;
    private List<string> _myData;
```

```
public BackgroundWorkerViewModel(IWorker worker)
{
    _worker = worker;
    LoadDataCommand = new RelayCommand(OnLoadData);
}

public RelayCommand LoadDataCommand { get; set; }

public List<string> MyData
{
    get { return _myData; }
    set
    {
        _myData = value;
        RaisePropertyChanged(() => MyData);
    }
}

public bool IsBusy { get; set; }

private void OnLoadData()
{
    IsBusy = true; // view is bound to IsBusy to show 'loading' message.

    _worker.Run(
        (sender, e) =>
        {
            MyData = new List<string> {"Test"};
            Thread.Sleep(1000);
        },
        (sender, e) => { IsBusy = false; });
}
```

کلاس مربوطه به آزمون واحد را مطابق با تغییرات ViewModel :

```
[TestFixture]
public class BackgroundWorkerViewModelTest
{
    #region Setup/Teardown

    [SetUp]
    public void Setup()
    {
        _backgroundWorkerViewModel = new BackgroundWorkerViewModel(new SyncWorker());
    }

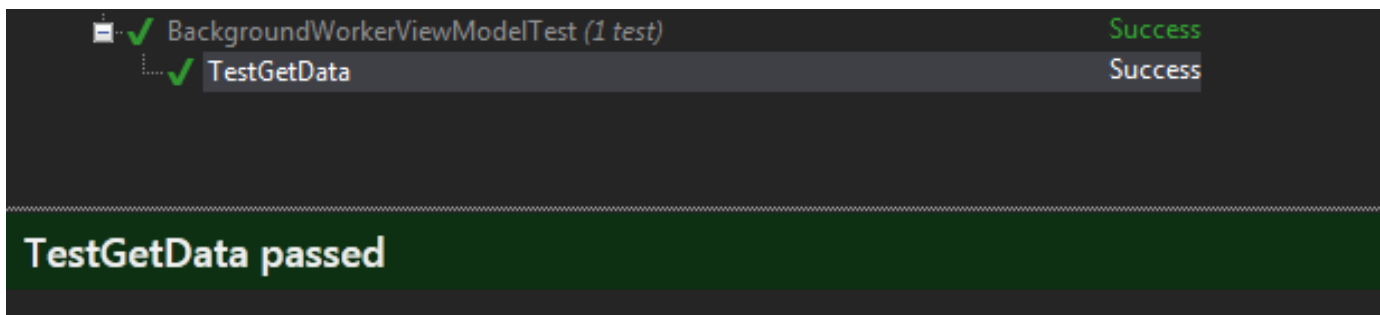
    #endregion

    private BackgroundWorkerViewModel _backgroundWorkerViewModel;

    [Test]
    public void TestGetData()
    {
        _backgroundWorkerViewModel.LoadDataCommand.Execute(_backgroundWorkerViewModel);

        Assert.NotNull(_backgroundWorkerViewModel.MyData);
        Assert.IsNotEmpty(_backgroundWorkerViewModel.MyData);
    }
}
```

اکنون اگر Unit Test را اجرا کنیم نتیجه اینگونه خواهد بود :



SimpleIoc به صورت پیش فرض در پروژه های MVVM Light موجود می باشد. قطعه کد پایین به صورت پیش فرض در پروژه های MVVM Light ایجاد می شود.

در کلاس ViewModelLocator ما تمام میانجی (Interface) ها و اشیا (Objects) ی مورد نیازمان را ثبت (register) می کنیم. در ادامه اجزای مختلف آن را شرح می دهیم.

```
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 می شود با الگوی Singleton ایجاد می شود.

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

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

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

3) جهت ثبت یک شیء مرتبط با میانجی از روش زیر استفاده می شود.

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

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

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

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

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

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


```
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](#)

تشریح مسئله : در MEF به صورت پیش فرض نوع نمونه ساخته شده از اشیا به صورت Singleton است. در صورتی که بخواهیم یک نمونه جدید از اشیا به ازای هر درخواست ساخته شود باید PartCreationPolicyAttribute رو به ازای هر کلاس مجدداً تعریف کنیم و نوع اون رو به NonShared تغییر دهیم. در پروژه‌های بزرگ این مسئله کمی آزار دهنده است. برای تغییر رفتار Container در MEF هنگام نمونه سازی Objectها باید چه کار کرد؟

نکته: آشنایی با مفاهیم MEF برای درک بهتر مطالب الزامی است.

*در صورتی که با مفاهیم MEF آشنایی ندارید می‌توانید از [اینجا](#) شروع کنید.

در MEF سه نوع PartCreationPolicy وجود دارد:

#1 Shared : آبجکت مورد نظر فقط یک بار در کل طول عمر Composition Container ساخته می‌شود. (Singleton)

#2 NonShared : آبجکت مورد نظر به ازای هر درخواست دوباره نمونه سازی می‌شود.

#3 Any : از حالت پیش فرض CompositionContainer برای نمونه سازی استفاده می‌شود که همان مورد اول است (Shared)

در اکثر پروژه‌ها ساخت نمونه اشیا به صورت Singleton میسر نیست و باعث اشکال در پروژه می‌شود. برای حل این مشکل باید PartCreationPolicy رو برای هر شی مجزا تعریف کنیم. برای مثال

```
[Export]
[PartCreationPolicy( CreationPolicy.NonShared )]
internal class ShellViewModel : ViewModel<IShellView>
{
    private readonly DelegateCommand exitCommand;

    [ImportingConstructor]
    public ShellViewModel( IShellView view )
        : base( view )
    {
        exitCommand = new DelegateCommand( Close );
    }
}
```

حال فرض کنید تعداد آبجکت شما در یک پروژه بیش از چند صد تا باشد. در صورتی که یک مورد را فراموش کرده باشید و UnitTest قوی و مناسب در پروژه تعبیه نشده باشد قطعاً در طی پروژه مشکلاتی به وجود خواهد آمد و امکان Debug سخت خواهد شد.

برای حل این مسئله بهتر است که رفتار Composition Container رو در هنگام نمونه سازی تغییر دهیم. یعنی آبجکت‌ها به صورت پیش فرض به صورت NonShared تولید شوند و در صورت نیاز به نمونه Shared این Attribute رو در کلاس مورد نظر استفاده کنیم. کافیه از کلاس Composition Container که قلب MEF محسوب می‌شود ارث برده و رفتار مورد نظر را Override کنیم. برای نمونه :

```
public class CustomCompositionContainer : CompositionContainer
{
    public CustomCompositionContainer(ComposablePartCatalog catalog)
        : base(catalog)
    {
    }

    protected override IEnumerable<Export> GetExportsCore(ImportDefinition definition)
    {
        definition = AdaptDefinition(definition);

        return base.GetExportsCore(definition);
    }
}
```

```
private ImportDefinition AdaptDefinition(ImportDefinition definition)
{
    ContractBasedImportDefinition namedDefinition = definition as ContractBasedImportDefinition;
    if (namedDefinition != null && namedDefinition.RequiredCreationPolicy == CreationPolicy.Any)
    {
        definition = new ContractBasedImportDefinition(namedDefinition.ContractName,
                                                         namedDefinition.RequiredMetadata,
                                                         namedDefinition.Cardinality,
                                                         namedDefinition.IsRecomposable,
                                                         namedDefinition.IsPrerequisite,
                                                         CreationPolicy.NonShared);
    }
    return definition;
}
```

مشاهده می‌کنید که متد GetExportCore در کلاس بالا Override شده است و توسط متد AdaptDefinition اگر PartCreationPolicy به صورت Any بود نمونه ساخته شده به صورت NonShared ایجاد می‌شود. حال فقط کافیست در پروژه به جای استفاده از CompositionContainer از CustomCompositionContainer استفاده کنیم.

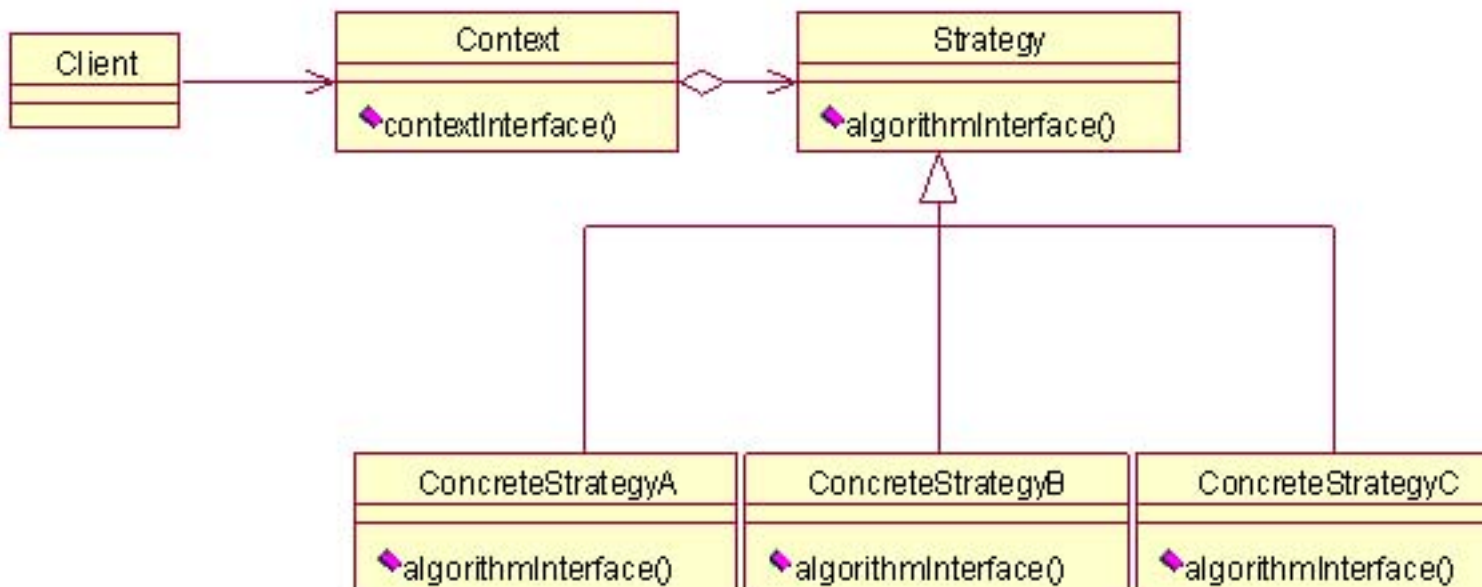
الگوی استراتژی (Strategy) اجازه می‌دهد که یک الگوریتم در یک کلاس بسته بندی شود و در زمان اجرا برای تغییر رفتار یک شیئی تعویض شود.

برای مثال فرض کنید که ما در حال طراحی یک برنامه مسیریابی برای یک شبکه هستیم. همانطوریکه می‌دانیم برای مسیر یابی الگوریتم‌های مختلفی وجود دارد که هر کدام دارای مزایا و معایبی هستند. و با توجه به وضعیت موجود شبکه یا عملی که قرار است انجام پذیرد باید الگوریتمی را که دارای بالاترین کارایی است انتخاب کنیم. همچنین این برنامه باید امکانی را به کاربر بدهد که کارائی الگوریتم‌های مختلف را در یک شبکه فرضی بررسی کنید. حالا طراحی پیشنهادی شما برای این مسئله چیست؟ دوباره فرض کنید که در مثال بالا در بعضی از الگوریتم‌ها نیاز داریم که گره‌های شبکه را بر اساس فاصله‌ی آنها از گره مبدا مرتب کنیم. دوباره برای مرتب سازی الگوریتم‌های مختلف وجود دارد و هر کدام در شرایط خاص، کارائی بهتری نسبت به الگوریتم‌های دیگر دارد. مسئله دقیقاً شبیه مسئله بالا است و این مسئله می‌تواند دارای طراحی شبیه مسئله بالا باشد. پس اگر ما بتوانیم یک طراحی خوب برای این مسئله ارائه دهیم می‌توانیم این طراحی را برای مسائل مشابه به کار ببریم.

هر کدام از ما می‌توانیم نسبت به درک خود از مسئله و سلیقه کاری، طراحی‌های مختلفی برای این مسئله ارائه دهیم. اما یک طراحی که می‌تواند یک جواب خوب و عالی باشد، الگوی استراتژی است که توانسته است بارها و بارها به این مسئله پاسخ بدهد.

الگوی استراتژی گزینه مناسبی برای مسائلی است که می‌توانند از چندین الگوریتم مختلف به مقصود خود برسند.

نمودار UML الگوی استراتژی به صورت زیر است :



اجازه بدهید، شیوه کار این الگو را با مثال مربوط به مرتب سازی بررسی کنیم. فرض کنید که ما تصمیم گرفتیم که از سه الگوریتم زیر برای مرتب سازی استفاده کنیم.

1 - الگوریتم مرتب سازی Shell Sort - الگوریتم مرتب سازی Quick Sort

3 - الگوریتم مرتب سازی Merge Sort

ما برای مرتب سازی در این برنامه دارای سه استراتژی هستیم. که هر کدام را به عنوان یک کلاس جداگانه در نظر می گیریم (همان کلاس های ConcreteStrategy). برای اینکه کلاس Client بتواند به سادگی یک از استراتژی ها را انتخاب کنید بهتر است که تمام کلاس های استراتژی دارای اینترفیس مشترک باشند. برای این کار می توانیم یک کلاس abstract تعریف کنیم و ویژگی های مشترک کلاس های استراتژی را در آن قرار دهیم و کلاس های استراتژی آنها را به ارث ببرند (همان کلاس Strategy) و پیاده سازی کنند.

در زیر کلاس Abstract که کل کلاس های استراتژی از آن ارث می برند را مشاهده می کنید :

```
abstract class SortStrategy
{
    public abstract void Sort(ArrayList list);
}
```

کلاس مربوط به QuickSort

```
class QuickSort : SortStrategy
{
    public override void Sort(ArrayList list)
    {
        // الگوریتم مربوطه
    }
}
```

کلاس مربوط به ShellSort

```
class ShellSort : SortStrategy
{
    public override void Sort(ArrayList list)
    {
        // الگوریتم مربوطه
    }
}
```

کلاس مربوط به MergeSort

```
class MergeSort : SortStrategy
{
    public override void Sort(ArrayList list)
    {
        // الگوریتم مربوطه
    }
}
```

و در آخر کلاس Context که یکی از استراتژی ها را برای مرتب کردن به کار می برد :

```
class SortedList
{
    private ArrayList list = new ArrayList();
    private SortStrategy sortstrategy;

    public void SetSortStrategy(SortStrategy sortstrategy)
    {
        this.sortstrategy = sortstrategy;
    }
    public void Add(string name)
```

```
{
    list.Add(name);
}
public void Sort()
{
    sortstrategy.Sort(list);
}
}
```

نظرات خوانندگان

نویسنده: علی

تاریخ: ۱۳۹۲/۰۶/۲۰ ۱۲:۴۶

با سلام؛ لطفا کلاس آخری را بیشتر توضیح دهید.

نویسنده: محسن خان

تاریخ: ۱۳۹۲/۰۶/۲۰ ۱۲:۵۵

کلاس آخری با یک پیاده سازی عمومی کار می‌کنه. دیگه نمی‌دونه نحوه مرتب سازی چطور پیاده سازی شده. فقط می‌دونه یک متد Sort هست که دراختیارش قرار داده شده. حالا شما راحت می‌تونن الگوریتم مورد استفاده رو عوض کنی، بدون اینکه نیاز داشته باشی کلاس آخری رو تغییر بدی. باز هست برای توسعه. بسته است برای تغییر. به این نوع طراحی رعایت open closed principle هم می‌گن.

نویسنده: SB

تاریخ: ۱۳۹۲/۰۶/۲۰ ۱۴:۲۳

بنظر شما متد Sort کلاس اولیه، نباید از نوع Virtual باشد؟

نویسنده: محسن خان

تاریخ: ۱۳۹۲/۰۶/۲۰ ۱۴:۴۸

نوع کلاسش abstract هست.

نویسنده: مجتبی شاطری

تاریخ: ۱۳۹۲/۰۶/۲۰ ۱۶:۴۷

در صورتی از virtual استفاده می‌کنیم که یک پیاده سازی از متد Sort در SortStrategy داشته باشیم، اما در اینجا طبق فرموده دوستمون کلاس ما فقط انتزاعی (Abstract) هست.

نویسنده: سید ایوب کوبی

تاریخ: ۱۳۹۲/۰۶/۳۱ ۱۱:۲۲

چرا استراتژی توسط Abstract پیاده سازی شده و از اینترفیس استفاده نشده؟

نویسنده: وحید نصیری

تاریخ: ۱۳۹۲/۰۶/۳۱ ۱۲:۵۱

تفاوت مهمی **نداره**؛ فقط اینترفیس ورژن پذیر نیست. یعنی اگر در این بین متدی رو به تعاریف اینترفیس خودتون اضافه کردید، تمام استفاده کننده‌ها مجبور هستند اون رو پیاده سازی کنند. اما کلاس Abstract می‌تونه شامل یک پیاده سازی پیش فرض متد خاصی هم باشه و به همین جهت ورژن پذیری بهتری داره. بنابراین کلاس Abstract یک اینترفیس است که می‌تواند پیاده سازی هم داشته باشه. همین مساله خاص نگارش پذیری، در طراحی ASP.NET MVC به کار گرفته شده: ([^](#)) برای من نوعی شاید این مساله اهمیتی نداشته باشه. اگر من قرارداد اینترفیس کتابخانه خودم را تغییر دادم، بالاخره شما با یک حداقل نق زدن مجبور به روز رسانی کار خودتان خواهید شد. اما اگر مایکروسافت چنین کاری را انجام دهد، هزاران نفر شروع خواهند کرد به بد گفتن از نحوه مدیریت پروژه تیم‌های مایکروسافت و اینکه چرا پروژه جدید آن‌ها با یک نگارش جدید MVC کامپایل نمی‌شود. بنابراین انتخاب بین این دو بستگی دارد به تعداد کاربر پروژه شما و استراتژی ورژن پذیری قرار دادهای کتابخانه‌ای که ارائه می‌دهید.

نویسنده: سید ایوب کوکبی
تاریخ: ۱۳۹۲/۰۶/۳۱ ۱۳:۲۷

اطلاعات خوبی بود، ممنون، ولی با توجه به تجربه تون، در پروژه‌های متن باز فعلی تحت بستر دات نت بیشتر از کدام مورد استفاده میشه؟ اینترفیس روحیه نظامی خاصی به کلاس‌های مصرف کننده اش میده، یه همین دلیل من زیاد رقبت به استفاده از اون ندارم، آیا مواردی هست که چاره ای نباشه حتما از یکی از این دو نوع استفاده بشه؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۳۱ ۱۳:۴۹

- اگر پروژه خودتون هست، از اینترفیس استفاده کنید. تغییرات آن و نگارش‌های بعدی آن تحت کنترل خودتان است و build دیگران را تحت تاثیر قرار نمی‌دهد.
- در پروژه‌های سورس باز دات نت، عموماً از ترکیب این دو استفاده می‌شود. مواردی که قرار است در اختیار عموم باشند حتی دو لایه هم می‌شوند. مثلاً در MVC یک اینترفیس IController هست و بعد یک کلاس Abstract به نام Controller، که این اینترفیس را پیاده سازی کرده برای ورژن پذیری بعدی و کنترلرهای پروژه‌های عمومی MVC از این کلاس Abstract مشتق می‌شوند یا در پروژه RavenDB از کلاس‌های Abstract زیاد استفاده شده، مانند AbstractIndexCreationTask و AbstractMultiMapIndexCreationTask و غیره.

نویسنده: جمشیدی فر
تاریخ: ۱۳۹۲/۰۷/۰۱ ۱۶:۱۱

توابع abstract بطور ضمنی virtual هستند.

نویسنده: جمشیدی فر
تاریخ: ۱۳۹۲/۰۷/۰۱ ۱۸:۳۸

در کلاس abstract نیز می‌توان از پیاده سازی پیشفرض استفاده کرد. یکی از تفاوت‌های کلاس abstract با Interface همین ویژگی است که سبب ورژن پذیری آن شده است.

نویسنده: جمشیدی فر
تاریخ: ۱۳۹۲/۰۸/۲۱ ۹:۱۵

بهتر نیست در کلاس SortedList برای مشخص کردن استراتژی مرتب سازی، از روش تزریق وابستگی - Dependency Injection - استفاده بشه؟

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۸/۲۱ ۹:۲۵

خوب، الان هم وابستگی کلاس یاد شده از طریق سازنده آن در اختیار آن قرار گرفته و داخل خود کلاس وهله سازی نشده. (در این مطلب طراحی بیشتر مدنظر هست تا اینکه حالا این وابستگی به چه صورتی و کجا قرار هست وهله سازی بشه و در اختیار کلاس قرار بگیره؛ این مساله ثانویه است)

نویسنده: جمشیدی فر
تاریخ: ۱۳۹۲/۰۸/۲۱ ۱۱:۴۳

از طریق سازنده کلاس SortedList؟ بنظر نمیداداز طریق سازنده انجام شده باشه. ولی ظاهراً این امکان هست که کلاس بالادستی که می‌خواهد از SortedList استفاده کند، بتواند از طریق تابع SetSortStrategy کلاس مورد نظر رادر اختیار SortedList قرار دهد. به نظر شبیه Setter Injection می‌شود.

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. لطفاً به کدهای ذیل توجه کنید:

کد بدون تزریق وابستگی‌ها

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 نمایید. خوشبختانه این مشکل قبلاً 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 عمدتاً 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 صریحاً 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.

اگر قصد داشته باشیم که تزریق وابستگی (Dependency Injection) را برای سرویس های WCF پایاده سازی کنیم نیاز به یک Instance Provider سفارشی داریم. در ابتدا باید سرویس های مورد نظر را در یک Ioc Container رجیستر نماییم سپس با استفاده از InstanceProvider عملیات و هله سازی از سرویس ها همراه با تزریق وابستگی انجام خواهد گرفت. فرض کنید سرویسی به صورت زیر داریم:

```
[ServiceBehavior( IncludeExceptionDetailInFaults = true)]
public class BookService : IBookService
{
    public BookService(IBookRepository bookRepository)
    {
        Repository = bookRepository;
    }

    public IBookRepository Repository
    {
        get;
        private set;
    }

    public IList<Entity.Book> GetAll()
    {
        return Repository.GetAll();
    }
}
```

همانطور که می بینید برای عملیات و هله سازی از این سرویس نیاز به تزریق کلاس BookRepository است که این کلاس باید ابنتر فیس IBookRepository را پایاده سازی کرده باشد. برای این که Instance Provider ما بتواند عملیات تزریق وابستگی را به درستی انجام دهد، ابتدا باید BookRepository و BookService را به یک IocContainer (در این جا از الگوی [ServiceLocator](#) و [UnityContainer](#) استفاده کردم) رجیستر نماییم. به صورت زیر:

```
var container = new UnityContainer();

container.RegisterType<IBookRepository, BookRepository>();
container.RegisterType<BookService, BookService>();

ServiceLocator.SetLocatorProvider(new ServiceLocatorProvider(() => { return container; }));
```

حال باید InstanceProvider را به صورت زیر ایجاد نماییم:

```
public class UnityInstanceProvider : IInstanceProvider
{
    private Type serviceType;

    public UnityInstanceProvider( Type serviceType )
    {
        this.serviceType = serviceType;
    }

    public object GetInstance( InstanceContext instanceContext, Message message )
    {
        return ServiceLocator.Current.GetInstance( serviceType );
    }

    public object GetInstance( InstanceContext instanceContext )
    {
        return GetInstance( instanceContext, null );
    }
}
```

```

public void ReleaseInstance( InstanceContext instanceContext, object instance )
{
    if ( instance is IDisposable )
    {
        ( ( IDisposable )instance ).Dispose();
    }
}

```

با پیاده سازی متدهای اینترفیس `IInstanceProvider` می توان عملیات وهله سازی سرویس های WCF را تغییر داد. متد `GetInstance` همین کار را برای ما انجام خواهد داد. در این متد ما با توجه به نوع `ServiceType` سرویس مورد نظر را از `ServiceLocator` تامین خواهیم کرد. چون وابستگی های سرویس هم در `IOC Cotnainer` موجود است در نتیجه سرویس به درستی وهله سازی خواهد شد. از آن جا که در WCF عملیات وهله سازی از سرویس ها به طور مستقیم به نوع سرویس بستگی دارد، هیچ نیازی به نوع `Contract` مربوطه نیست. در نتیجه `Service Type` به صورت مستقیم در اختیار این کلاس قرار خواهد گرفت. مرحله آخر معرفی `IInstanceProvider` به عنوان یک `Service Behavior` است. برای این کار کدهای زیر را در کلاسی به نام `UnityInstanceProviderContext` کپی نمایید:

```

public class UnityInstanceProviderContext : IServiceBehavior
{
    public void AddBindingParameters( ServiceDescription serviceDescription , ServiceHostBase
serviceHostBase , Collection<ServiceEndpoint> endpoints , BindingParameterCollection bindingParameters
)
    {
    }

    public void ApplyDispatchBehavior( ServiceDescription serviceDescription , ServiceHostBase
serviceHostBase )
    {
        serviceHostBase.ChannelDispatchers.ToList().ForEach( channelDispatcherBase =>
        {
            var channelDispatcher = ( channelDispatcherBase as ChannelDispatcher );
            if ( channelDispatcher != null )
            {
                channelDispatcher.Endpoints.ToList().ForEach( endpoint =>
                {
                    endpoint.DispatchRuntime.InstanceProvider = new UnityInstanceProvider(
serviceDescription.ServiceType );
                } );
            }
        } );
    }

    public void Validate( ServiceDescription serviceDescription , ServiceHostBase serviceHostBase )
    {
    }
}

```

در متد `ApplyDispatchBehavior` همان طور دیده می شود به ازای تمام `EndPoint` های هر `ChannelDispatcher` یک نمونه از کلاس `UnityInstanceProvider` به همراه پارامتر سازنده آن که نوع سرویس مورد نظر است به خاصیت `InstanceProvider` در `DispatchRuntime` معرفی می گردد. در هنگام هاست سرویس مورد نظر هم باید تمام این عملیات به عنوان یک `Behavior` در اختیار `Service Host` قرار گیرد. همانند نمونه کد ذیل:

```

using (ServiceHost host = new ServiceHost(typeof(BookService)))
{
    host.Description.Behaviors.Add( new UnityInstanceProviderContext() );
}

```

نظرات خوانندگان

نویسنده: وحید م
تاریخ: ۱۳۹۲/۱۱/۱۱ ۸:۴۸

با سلام؛ اگر یک برنامه چند لایه داشته باشیم (UI- DomainLayer-DataAccess- Service) و مثلاً یک پروژه‌ای هم از نوع webapi یا wcf در آن معماری قرار داشت، می‌خواهم در web api یا wcf هم برای اعمال dependency Injection با آن mapping ها که در لایه ui قرار دادم هم استفاده کنم. با تشکر.

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۱۱/۱۱ ۹:۳۶

در مطلب فوق محل قرارگیری container.RegisterType در نقطه آغاز برنامه است؛ جایی که نگاشت‌های مورد نیاز در سایر لایه‌ها هم انجام می‌شود. بنابراین فرقی نمی‌کند.

نویسنده: وحید م
تاریخ: ۱۳۹۲/۱۱/۱۱ ۱۰:۴۳

ممنون ولی سوال بنده کلی بود؟ وقتی یک معماری دارم بگونه ای گفته شد یا مثل cms IRIS آقای سعیدی فر و خواستم به پروژه پروژه دیگری اضافه کنم از نوع webapi یا wcf که به نوعی از لایه service هم برای اتصال به بانک استفاده میکنه DI را باید چگونه برای آن اعمال کرد؟ آیا می‌بایست تنظیمات و mapping های داخل global مربوط به structuremap درون ui را در داخل پروژه webapi یا wcf هم قرار داد یا خیر؟ اگر webapi را جدا هاست کنیم چه تضمینی وجود دارد دیگر به پروژه webapi دسترسی نداشته باشد

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۱۱/۱۱ ۱۰:۵۰

صرف نظر از اینکه برنامه شما از چند DLL نهایتاً تشکیل میشه، تمام این‌ها داخل یک Application Domain اجرا می‌شن. یعنی عملاً یک برنامه‌ی واحد شما دارید که از اتصال قسمت‌های مختلف با هم کار می‌کنه. IoC Container هم تنظیماتش اول کار برنامه کش می‌شه. یعنی یکبار که تنظیم شد، در سراسر آن برنامه قابل دسترسی هست. بنابراین نیازی نیست همه جا تکرار بشه. یکبار آغاز کار برنامه اون رو تنظیم کنید کافی هست.

هر از چندگاهی یک چنین آدرس‌های یافت نشدی را در لاگ‌های سایت مشاهده می‌کنم:

```
http://www.dotnettips.info/jquery
http://www.dotnettips.info/mvc
http://www.dotnettips.info/برنامه
```

[روش متداول](#) مدیریت این نوع آدرس‌ها، هدایت خودکار به صفحه‌ی 404 است. اما شاید بهتر باشد بجای اینکار، کاربران به صورت خودکار به صفحه‌ی جستجوی سایت هدایت شوند. در ادامه مراحل اینکار را بررسی خواهیم کرد.

الف) ساختار کنترلر جستجوی سایت

فرض کنید جستجوی سایت در کنترلری به نام Search و توسط اکشن متد پیش فرضی با فرمت زیر مدیریت می‌شود:

```
[ValidateInput(false)] //برنامه نویسی‌ها نیاز دارند تگ‌ها را جستجو کنند
public virtual ActionResult Index(string term)
{
```

ب) مدیریت کنترلرهای یافت نشد

اگر از یک IoC Container در برنامه‌ی ASP.NET MVC خود مانند [StructureMap](#) استفاده می‌کنید، نوشتن کد متداول زیر کافی نیست:

```
public class StructureMapControllerFactory : DefaultControllerFactory
{
    protected override IController GetControllerInstance(RequestContext requestContext, Type
controllerType)
    {
        return ObjectFactory.GetInstance(controllerType) as Controller;
    }
}
```

از این جهت که اگر کاربر آدرس <http://www.dotnettips.info/test> را وارد کند، controllerType درخواستی نال خواهد بود؛ چون جزو کنترلرهای سایت نیست. به همین جهت نیاز است موارد نال را هم مدیریت کرد:

```
public class StructureMapControllerFactory : DefaultControllerFactory
{
    protected override IController GetControllerInstance(RequestContext requestContext, Type
controllerType)
    {
        if (controllerType == null)
        {
            var url = requestContext.HttpContext.Request.RawUrl;
            //string.Format("Page not found: {0}", url).LogException();

            requestContext.RouteData.Values["controller"] = MVC.Search.Name;
            requestContext.RouteData.Values["action"] = MVC.Search.ActionNames.Index;
            requestContext.RouteData.Values["term"] = url.GetPostSlug().Replace("-", " ");
            return ObjectFactory.GetInstance(typeof(SearchController)) as Controller;
        }
        return ObjectFactory.GetInstance(controllerType) as Controller;
    }
}
```

کاری که در اینجا انجام شده، هدایت خودکار کلیه کنترلرهای یافت نشد برنامه، به کنترلر Search است. اما در این بین نیاز است سه مورد را نیز اصلاح کرد. در RouteData.Values جاری، نام کنترلر باید به نام کنترلر Search تغییر کند. زیرا مقدار پیش فرض آن، همان عبارتی است که کاربر وارد کرده. همچنین باید مقدار action را نیز اصلاح کرد، چون اگر آدرس وارد شده برای مثال <http://www.dotnettips.info/mvc/test> بود، [مقدار پیش فرض](#) action همان test می‌باشد. بنابراین صرف بازگشت وهله‌ای از

SearchController تمام موارد را پوشش نمی‌دهد و نیاز است دقیقاً جزئیات سیستم مسیریابی نیز اصلاح شوند. همچنین پارامتر term اکشن متد index را هم در اینجا می‌شود مقدار دهی کرد. برای مثال در اینجا عبارت وارد شده اندکی تمیز شده (مطابق روش متد تولید Slug) و سپس به عنوان مقدار term تنظیم می‌شود.

ج) مدیریت آدرس‌های یافت نشد پسوند دار

تنظیمات فوق کلیه آدرس‌های بدون پسوند را مدیریت می‌کند. اما اگر درخواست رسیده به شکل <http://www.dotnettips.info/mvc/test/file.aspx> بود، خیر. در اینجا حداقل سه مرحله را باید جهت مدیریت و هدایت خودکار آن به صفحه‌ی جستجو انجام داد

- باید فایل‌های پسوند دار را وارد سیستم مسیریابی کرد :

```
routes.RouteExistingFiles = true; // نیاز هست دانلود عمومی فایل‌ها تحت کنترل قرار گیرد
```

- در ادامه نیاز است مسیریابی Catch all اضافه شود:

پس از [مسیریابی پیش فرض](#) سایت (نه قبل از آن)، مسیریابی ذیل باید اضافه شود:

```
routes.MapRoute(
    "CatchAllRoute", // Route name
    "{*url}", // URL with parameters
    new { controller = "Search", action = "Index", term = UrlParameter.Optional, area = "" }, // Parameter defaults
    new { term = new UrlConstraint() }
);
```

مسیریابی پیش فرض، تمام آدرس‌های سازگار با ساختار MVC را می‌تواند مدیریت کند. فقط حالتی از آن عبور می‌کند که پسوند داشته باشد. با قرار دادن این مسیریابی جدید پس از آن، کلیه آدرس‌های مدیریت نشده به کنترلر Search و اکشن متد Index آن هدایت می‌شوند.

مشکل! نیاز است پارامتر term را به صورت پویا مقدار دهی کنیم. برای اینکار می‌توان یک RouteConstraint سفارشی نوشت:

```
public class UrlConstraint : IRouteConstraint
{
    public bool Match(System.Web.HttpContextBase httpContext,
        Route route, string parameterName,
        RouteValueDictionary values,
        RouteDirection routeDirection)
    {
        var url = httpContext.Request.RawUrl;
        //string.Format("Page not found: {0}", url).LogException();

        values["term"] = url.GetPostSlug().Replace("-", " ");
        return true;
    }
}
```

UrlConstraint مطابق تنظیم CatchAllRoute فقط زمانی فراخوانی خواهد شد که برنامه به این مسیریابی خاص برسد (و نه در سایر حالات متداول کار با کنترلر جستجو). در اینجا فرصت خواهیم داشت تا مقدار term را به RouteValueDictionary آن اضافه کنیم.

[LightInject](#) در حال حاضر یکی از قدرتمندترین IoC Container ها است که از لحاظ سرعت و کارایی در بالاترین جایگاه در میان IoC Container های موجود قرار دارد. جهت بررسی کارایی IoC Container ها 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 کارایی، 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 واکنشی 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. حتماً 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 قبلاً 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 برگرداند.

```

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 `Collection` های زیر نیز پشتیبانی می نماید:

```
Array
<ICollection<T>
<IList<T>
<IReadOnlyCollection<T>
<IReadOnlyList<T>
```

Values توسط LightInject می توانید مقادیر ثابت را نیز تعریف کنید

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

متغیر `value` با رشته `"SomeValue"` مقداردهی می گردد. اگر چندین ثابت رشته ای داشته باشید می توانید 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 می گردد.

نظرات خوانندگان

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

ممنون از مطلب خوبتون
من به مقایسه دیگه دیدم که اونجا گفته بود Ligth Inject از Instance Per Request پشتیبانی نمی‌کنه
میخواستم جایگزین Unity کنم برای حالتی که حالتی 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.