

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 چندانگانه 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 ) );

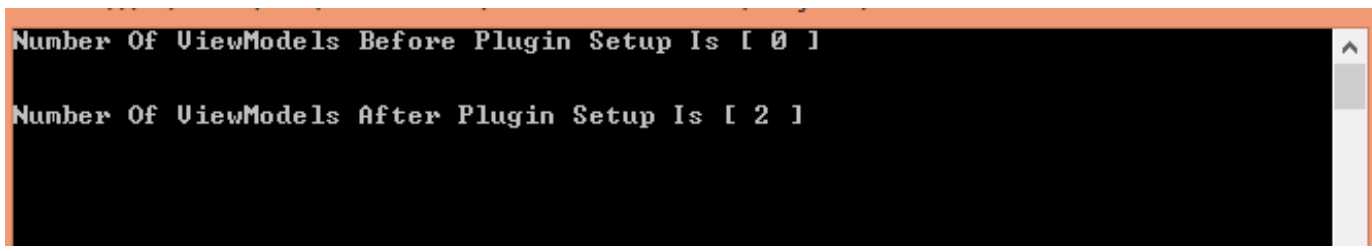
    Console.WriteLine( Environment.NewLine );

    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.



```
Number Of ViewModels Before Plugin Setup Is [ 0 ]

Number Of ViewModels After Plugin Setup Is [ 2 ]
```

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 پیاده سازی Dependency Injection و ServiceLocator Pattern است. یک ServiceLocator در واقع وظیفه تهیه Instance‌های مختلف از کلاس‌ها رو برای پیاده 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 پیاده 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();

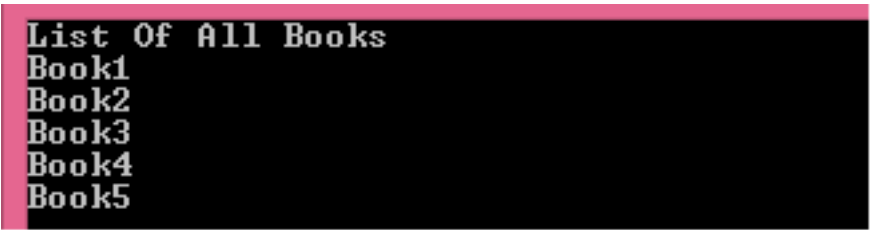
        unityContainer.RegisterType<IBookRepository, BookRepository>();

        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 نماییم 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 ) );

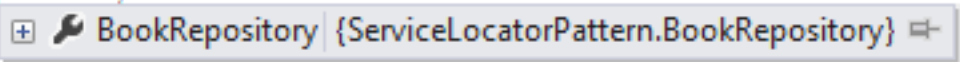
            BookService service = new BookService();

            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

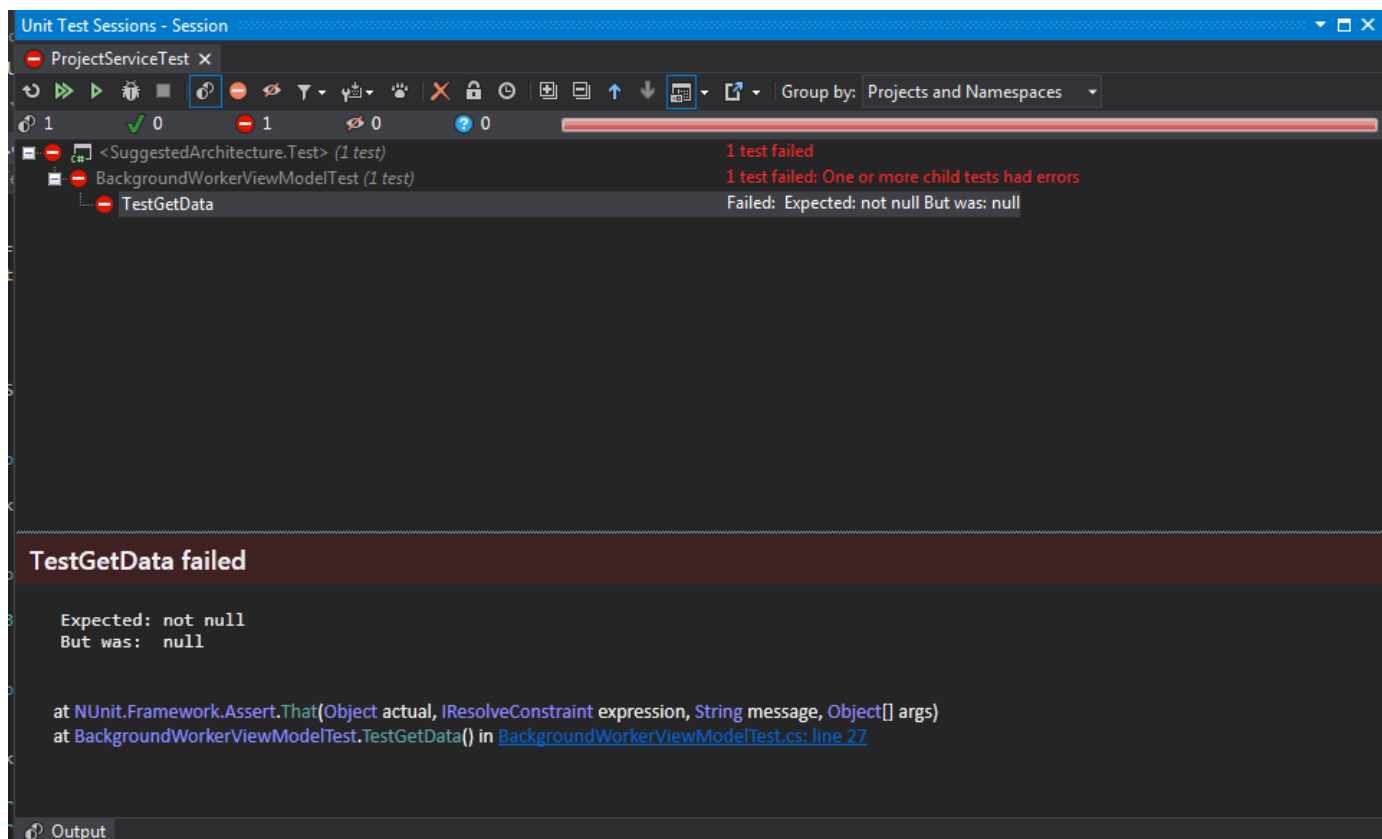
    private BackgroundWorkerViewModel _backgroundWorkerViewModel;

    [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());
    }

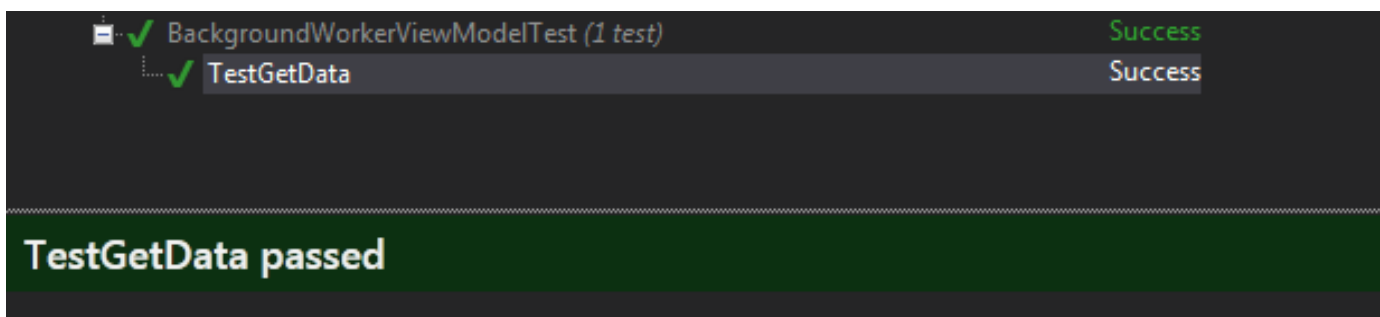
    #endregion

    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.

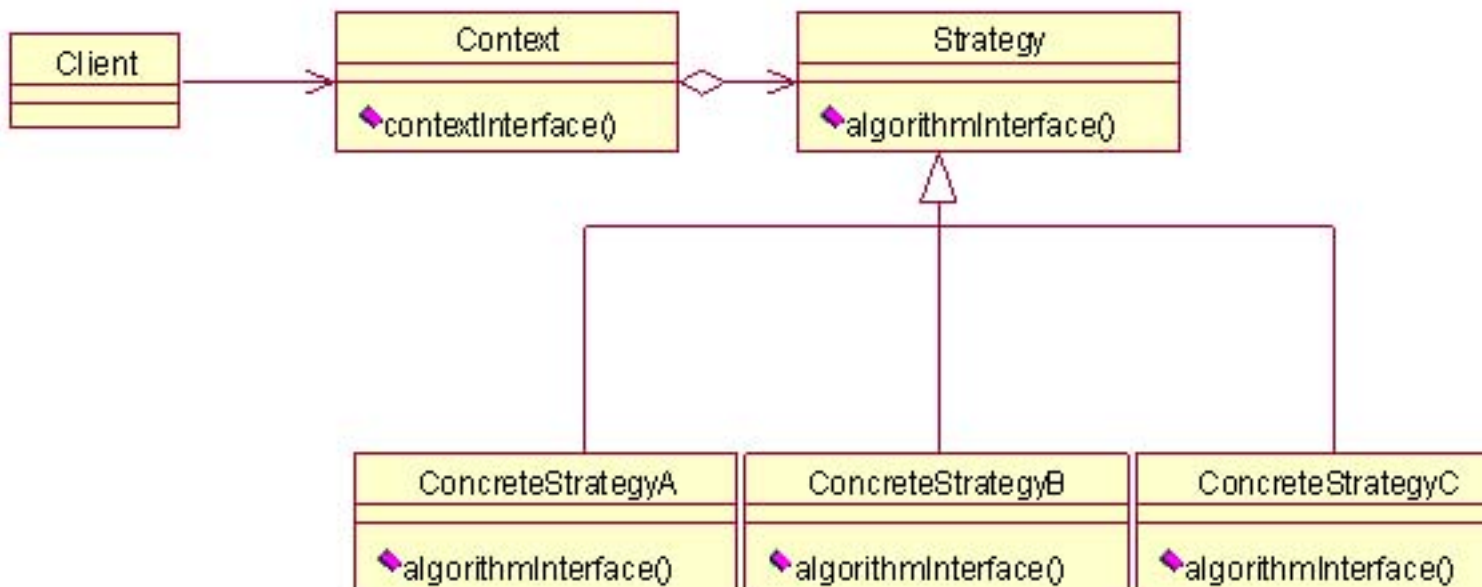
الگوی استراتژی (Strategy) اجازه می‌دهد که یک الگوریتم در یک کلاس بسته بندی شود و در زمان اجرا برای تغییر رفتار یک شیئی تعویض شود.

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 2 - 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 کوکبی

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: ۱۳:۲۷ ۱۳۹۲/۰۶/۳۱

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: ۱۳:۴۹ ۱۳۹۲/۰۶/۳۱

- 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: ۱۶:۱۱ ۱۳۹۲/۰۷/۰۱

توابع abstract بطور ضمنی virtual هستند.

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

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: ۹:۱۵ ۱۳۹۲/۰۸/۲۱

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

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

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: ۱۱:۴۳ ۱۳۹۲/۰۸/۲۱

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 API HTTP Context caching 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 نمایم:

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

    public UnityInstanceProvinder( 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 داندلود 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. 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 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 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.

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:

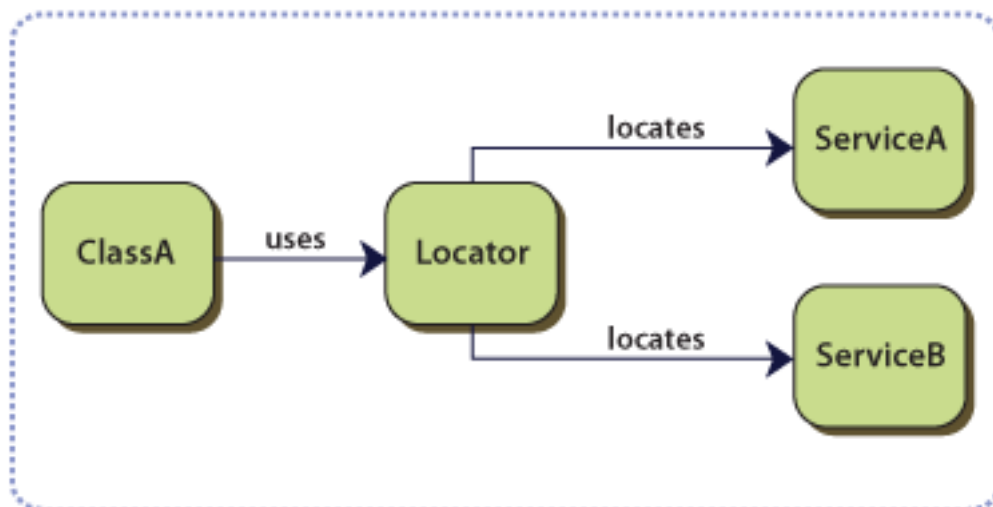
```
DependencyResolver.SetResolver(new UnityDependencyResolver(container));
```

تا به اینجا به راحتی 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 معمولاً در لایه سرویس به اشکال گوناگونی پیاده سازی میشود که کارش و همه سازی از Interface های 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 آبجکت ها، از Unity استفاده کرده ایم. در همان ابتدا که برنامه ی وب ما برای اولین بار اجرا میشود و بعد از Register کردن کلاس ها، می توانیم container را به صورت پارامتر سازنده به کلاس Service Factory ارسال کنیم. به این ترتیب برای استفاده از سرویس ها 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 هر جا نیاز داشتیم با استفاده از متد Create از آن وهله سازی بکنیم.

در باب ضرورت نوشتن کدهای تست پذیر، توسعه کلاس‌های کوچک تک مسئولیتی و اهمیت تزریق وابستگی‌ها بارها و بارها بحث شده و مطلب نوشته شده است. این روزها کم پیش می‌آید که نرم افزاری توسعه داده شود و از پایگاه داده به جهت ذخیره و بازیابی داده‌ها استفاده نکند. با گسترش و رواج ORM ها، نوشتن کدهای دسترسی به داده‌ها سهولت یافته است و استفاده از ORM در لایه‌ی سرویس که نگهدارنده‌ی منطق تجاری برنامه است، امری اجتناب ناپذیر می‌باشد.

در این مطلب نحوه‌ی نوشتن آزمون واحد برای کلاس سرویسی که وابسته به DbContext می‌باشد، به همراه محدودیت‌ها شرح داده می‌شود. ابتدا یک روش که در آن مستقیماً از DbContext در سرویس استفاده شده را بررسی می‌کنیم. در مثال زیر کلاس ProductService وظیفه‌ی برگرداندن لیست کالاها را به ترتیب نام دارد. در آن DbContext مستقیماً و هله سازی شده و از آن جهت انجام تراکنش‌های دیتابیس کمک گرفته شده است:

```
public class ProductService
{
    public IEnumerable<Product> GetOrderedProducts()
    {
        using (var ctx = new Entites())
        {
            return ctx.Products.OrderBy(x => x.Name).ToList();
        }
    }
}
```

برای این کلاس نمی‌توان Unit Test نوشت چرا که یک وابستگی به شی DbContext دارد و این وابستگی مستقیماً درون متد GetOrderedProducts نمونه سازی شده است. در مطالب پیشین شرح داده شد که برای تست پذیر کردن کدها باید این وابستگی‌ها را از بیرون، در اختیار کلاس مورد نظر قرار داد.

برای نوشتن تست برای کلاس ProductService حداقل دو روش در اختیار است:

- نوشتن Integration Test :

یعنی کلاس جاری را به همین شکل نگاه داریم و در تست، مستقیماً به یک پایگاه داده که به منظور تست فراهم شده وصل شویم. برای سهولت مدیریت پایگاه داده می‌توان عمل درج را در یک Transaction قرار داد و پس از پایان یافتن تست Transaction را RollBack کرد. این روش مورد بحث مطلب جاری نمی‌باشد، لطفاً برای آشنایی این دو مطلب را مطالعه بفرمایید:

[Using Entity Framework in integration tests](#)

[How We Do Database Integration Tests With Entity Framework Migrations](#)

- بهره جستن از تزریق وابستگی و نوشتن Unit Test که وابستگی به دیتابیس ندارد

یکی از قانون‌های یک آزمون واحد این است که وابستگی به منابع خارجی مثل پایگاه داده نداشته باشد. [این مطلب](#) نحوه‌ی صحیح پیاده سازی الگوی Unit of Work را شرح داده است. بعد از پیاده سازی Unit Of Work، کلاس DbContext به شرح زیر می‌شود. همانطور که مشاهده می‌کنید، اکنون DbContext یک Interface را پیاده سازی کرده است.

```
public interface IUnitOfWork
{
    IDbSet<TEntity> Set<TEntity>() where TEntity : class;
    int SaveAllChanges();
}

public class Entites : DbContext, IUnitOfWork
```

```

{
    public virtual DbSet<Product> Products { get; set; } // This is virtual because Moq needs to
    override the behaviour

    public new virtual IDbSet<TEntity> Set<TEntity>() where TEntity : class // This is virtual
    because Moq needs to override the behaviour
    {
        return base.Set<TEntity>();
    }

    public int SaveAllChanges()
    {
        return base.SaveChanges();
    }
}

```

در این حالت می‌توان به جای وهله سازی مستقیم DbContext در ProductService آن را خارج از کلاس سرویس در اختیار استفاده کننده قرار داد:

```

public class ProductService
{
    private readonly IDbSet<Product> _products;
    private readonly IUnitOfWork _uow;
    public ProductService(IUnitOfWork uow)
    {
        _uow = uow;
        _products = _uow.Set<Product>();
    }
    public IEnumerable<Product> GetOrderedProducts()
    {
        return _products.OrderBy(x => x.Name).ToList();
    }
}

```

همانطور که مشاهده می‌کنید، الان IUnitOfWork به کلاس سرویس تزریق شده و در متدها، خبری از وهله سازی یک وابستگی (DbContext) نمی‌باشد.

اکنون برای تست این سرویس می‌توان پیاده سازی دیگری را از IUnitOfWork انجام داد و در کدهای تست به سرویس مورد نظر تزریق کرد. برای سهولت این امر قصد داریم از moq به عنوان چارچوب تقلید (Mocking framework) استفاده کنیم. برای [نصب moq](#) می‌توان از [بسته‌ی نیوگت](#) آن بهره جست. پیشتر [مطلبی](#) در رابطه با چارچوب‌های تقلید در سایت نوشته شده است. با توجه به اینکه ProductService به دیتابیس وابستگی دارد، مقصود این است که این وابستگی با ایجاد یک نمونه‌ی mock از IUnitOfWork حذف شود. برای این منظور در سازنده‌ی کلاس، تعدادی کالای درون حافظه ایجاد شده و به صورت IQueryable جایگزین شده DbSet است.

اگر به تعریف کلاس Entities که همان DbContext می‌باشد دقت کنید، مشاهده می‌شود که Products و تابع Set، هر دو به صورت Virtual تعریف شده‌اند. برای تغییر رفتار DbContext نیاز است در آزمون واحد، این دو با داده‌های درون حافظه کار کنند و رفتار آنها قرار است عوض شود. این تغییر رفتار از طریق چند ریختی (Polymorphism) خواهد بود. کلاس تست در نهایت اینگونه تعریف می‌شود:

```

[TestFixture]
public class ProductServiceTest
{
    private readonly ProductService _productService;
    public ProductServiceTest()
    {
        IQueryable<Product> data = GetRoadNetworks().AsQueryable();
        var mockSet = new Mock<DbSet<Product>>();
        mockSet.As<IQueryable<Product>>().Setup(m => m.Provider).Returns(data.Provider);
        mockSet.As<IQueryable<Product>>().Setup(m => m.Expression).Returns(data.Expression);
        mockSet.As<IQueryable<Product>>().Setup(m => m.ElementType).Returns(data.ElementType);
        mockSet.As<IQueryable<Product>>().Setup(m =>
        m.GetEnumerator()).Returns(data.GetEnumerator());
        var context = new Mock<Entities>();
        context.Setup(c => c.Products).Returns(mockSet.Object);
        context.Setup(m => m.Set<Product>()).Returns(mockSet.Object);
        _productService = new ProductService(context.Object);
    }
}

```

```

}
private IEnumerable<Product> GetRoadNetworks()
{
    return new List<Product>
    {
        new Product
        {
            Id = 1,
            Name = "A"
        },
        new Product
        {
            Id = 2,
            Name = "B"
        },
        new Product
        {
            Id = 3,
            Name = "C"
        }
    };
}
[Test]
public void GetOrderedProductTest()
{
    IEnumerable<Product> products = _productService.GetOrderedProducts();
    List<string> names = products.Select(x => x.Name).ToList();
    var expected = new List<string> {"A", "B", "C"};
    CollectionAssert.AreEqual(names, expected);
}
}

```

همانطور که مشاهده می‌شود، در سازنده‌ی کلاس تست، یک منبع داده‌ی درون حافظه‌ای به صورت IQueryable تولید شده و پیاده سازی‌های تقلیدی از DbContext به همراه تابع Set و همچنین DbSet کالاهای به کمک Moq ایجاد گردیده و در اختیار ProductService قرار داده شده است.

در نهایت، در یک تست تلاش شده است تا منطق متد GerOrderedProducts مورد آزمون قرار گیرد. **محدودیت این روش:** با اینکه LINQ یک روش و سینتکس یکتا برای دسترسی به منابع داده‌ای مختلف را محیا می‌کند، اما این الزامی برای یکسان بودن نتایج، هنگام استفاده از Providerهای مختلف LINQ نمی‌باشد. در تست نوشته شده از LINQ To Objects برای کوئری گرفتن از منبع داده استفاده شده است؛ در صورتیکه در برنامه‌ی اصلی از LINQ To Entities استفاده می‌شود و الزامی نیست که یک کوئری LINQ در دو Provider متفاوت یک رفتار را داشته باشد.

این نکته در قسمت Limitations of EF in-memory test doubles [این مطلب](#) هم شرح داده شده است. در نهایت این پرسش به وجود می‌آید که با وجود محدودیت ذکر شده، از این روش استفاده شود یا خیر؟ پاسخ این پرسش، بسته به هر سناریو، متفاوت است.

به عنوان نمونه اگر در یک سناریو داده‌ها با یک کوئری نه چندان پیچیده از منبع داده ای گرفته می‌شود و اعمال دیگری دیگری روی نتیجه‌ی کوئری درون حافظه انجام می‌شود می‌توان این روش را قابل اعتماد قلمداد کرد. [EFTesting.zip](#) برای مطالعه‌ی بیشتر مطالب متعددی در سایت در رابطه با [تزیق وابستگی](#) و آزمون‌های واحد نوشته شده است.

نظرات خوانندگان

نویسنده: شاهین کیاست
تاریخ: ۱۸:۴۶ ۱۳۹۳/۰۹/۰۳

-نکته تکمیلی در صورتی که از AsNoTracking در کدهای لایه‌ی سرویس استفاده شده برای Mock کردن آن می‌توان به این صورت عمل کرد:

```
context.Setup(c => c..AsNoTracking()).Returns(mockSet.Object);
```

در صورت عدم درج کد بالا تست‌ها با خطای Null Exception متوقف می‌شوند. [اطلاعات بیشتر](#)

عنوان: اعمال تزریق وابستگی‌ها به مثال رسمی ASP.NET Identity

نویسنده: وحید نصیری

تاریخ: ۱۳:۳۵ ۱۳۹۳/۰۹/۲۹

آدرس: www.dotnettips.info

گروه‌ها: ASP.Net, Entity framework, MVC, Dependency Injection, ASP.NET Identity

پروژه‌ی [ASP.NET Identity](#) که نسل جدید سیستم Authentication و Authorization مخصوص ASP.NET است، دارای دو سری مثال رسمی است:

الف) [مثال‌های کدپلکس](#)

ب) [مثال نیوگت](#)

در ادامه قصد داریم مثال نیوگت آن‌را که مثال کاملی است از نحوه‌ی استفاده از ASP.NET Identity در ASP.NET MVC، جهت اعمال الگوی واحد کار و تزریق وابستگی‌ها، بازنویسی کنیم.

پیشنیازها

- برای درک مطلب جاری نیاز است ابتدا [دوره‌ی مرتبطی را در سایت مطالعه کنید](#) و همچنین با [نحوه‌ی پیاده سازی الگوی واحد کار در EF Code First](#) آشنا باشید.

- به علاوه فرض بر این است که یک پروژه‌ی خالی ASP.NET MVC 5 را نیز آغاز کرده‌اید و توسط کنسول پاور شل نیوگت، فایل‌های مثال `Microsoft.AspNet.Identity.Samples` را به آن افزوده‌اید:

```
PM> Install-Package Microsoft.AspNet.Identity.Samples -Pre
```

ساختار پروژه‌ی تکمیلی

همانند مطلب [پیاده سازی الگوی واحد کار در EF Code First](#)، این پروژه‌ی جدید را با چهار اسمبلی `class library` دیگر به نام‌های

```
AspNetIdentityDependencyInjectionSample.DataLayer
AspNetIdentityDependencyInjectionSample.DomainClasses
AspNetIdentityDependencyInjectionSample.IocConfig
AspNetIdentityDependencyInjectionSample.ServiceLayer
```

تکمیل می‌کنیم.

ساختار پروژه‌ی `AspNetIdentityDependencyInjectionSample.DomainClasses`

مثال `Microsoft.AspNet.Identity.Samples` بر مبنای `primary key` از نوع `string` است. برای نمونه کلاس کاربران آن‌را به نام `ApplicationUser` در فایل `Models\IdentityModels.cs` می‌توانید مشاهده کنید. در مطلب جاری، این نوع پیش فرض، به نوع متداول `int` تغییر خواهد یافت. به همین جهت نیاز است کلاس‌های ذیل را به پروژه‌ی `DomainClasses` اضافه کرد:

```
using System.ComponentModel.DataAnnotations.Schema;
using Microsoft.AspNet.Identity.EntityFramework;

namespace AspNetIdentityDependencyInjectionSample.DomainClasses
{
    public class ApplicationUser : IdentityUser<int, CustomUserLogin, CustomUserRole, CustomUserClaim>
    {
        // سایر خواص اضافی در اینجا

        [ForeignKey("AddressId")]
        public virtual Address Address { get; set; }
        public int? AddressId { get; set; }
    }
}

using System.Collections.Generic;
```

```

namespace AspNetIdentityDependencyInjectionSample.DomainClasses
{
    public class Address
    {
        public int Id { get; set; }
        public string City { get; set; }
        public string State { get; set; }

        public virtual ICollection<ApplicationUser> ApplicationUsers { set; get; }
    }
}

using Microsoft.AspNet.Identity.EntityFramework;

namespace AspNetIdentityDependencyInjectionSample.DomainClasses
{
    public class CustomRole : IdentityRole<int, CustomUserRole>
    {
        public CustomRole() { }
        public CustomRole(string name) { Name = name; }
    }
}

using Microsoft.AspNet.Identity.EntityFramework;

namespace AspNetIdentityDependencyInjectionSample.DomainClasses
{
    public class CustomUserClaim : IdentityUserClaim<int>
    {
    }
}

using Microsoft.AspNet.Identity.EntityFramework;

namespace AspNetIdentityDependencyInjectionSample.DomainClasses
{
    public class CustomUserLogin : IdentityUserLogin<int>
    {
    }
}

using Microsoft.AspNet.Identity.EntityFramework;

namespace AspNetIdentityDependencyInjectionSample.DomainClasses
{
    public class CustomUserRole : IdentityUserRole<int>
    {
    }
}

```

در اینجا نحوه‌ی تغییر primary key از نوع string را به نوع int، مشاهده می‌کنید. این تغییر نیاز به اعمال به کلاس‌های کاربران و همچنین نقش‌های آن‌ها نیز دارد. به همین جهت صرفاً تغییر کلاس ابتدایی ApplicationUser کافی نیست و باید کلاس‌های فوق را نیز اضافه کرد و تغییر داد.

بدیهی است در اینجا کلاس پایه کاربران را می‌توان سفارشی سازی کرد و خواص دیگری را نیز به آن افزود. برای مثال در اینجا یک کلاس جدید آدرس تعریف شده است که ارجاعی از آن در کلاس کاربران نیز قابل مشاهده است. سایر کلاس‌های مدل‌های اصلی برنامه که جداول بانک اطلاعاتی را تشکیل خواهند داد نیز در آینده به همین اسمبلی DomainClasses اضافه می‌شوند.

ساختار پروژه‌ی AspNetIdentityDependencyInjectionSample.DataLayer جهت اعمال الگوی واحد کار

اگر به همان فایل Models\IdentityModels.cs ابتدایی پروژه که اکنون کلاس ApplicationUser آن را به پروژه‌ی DomainClasses منتقل کرده ایم، مجدداً مراجعه کنید، کلاس DbContext مخصوص ASP.NET Identity نیز در آن تعریف شده است:

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
```


این کلاس را به پروژه‌ی DataLayer منتقل می‌کنیم و از آن به عنوان DbContext اصلی برنامه استفاده خواهیم کرد. بنابراین دیگر نیازی نیست چندین DbContext در برنامه داشته باشیم. IdentityDbContext، در اصل از DbContext مشتق شده‌است. اینترفیس IUnitOfWork برنامه، در پروژه‌ی DataLayer چنین شکلی را دارد که نمونه‌ای از آن را در مطلب [آشنایی با نحوه‌ی پیاده سازی الگوی واحد کار در EF Code First](#)، بیشتر ملاحظه کرده‌اید.

```
using System.Collections.Generic;
using System.Data.Entity;

namespace AspNetIdentityDependencyInjectionSample.DataLayer.Context
{
    public interface IUnitOfWork
    {
        IDbSet<TEntity> Set<TEntity>() where TEntity : class;
        int SaveAllChanges();
        void MarkAsChanged<TEntity>(TEntity entity) where TEntity : class;
        IList<T> GetRows<T>(string sql, params object[] parameters) where T : class;
        IEnumerable<TEntity> AddThisRange<TEntity>(IEnumerable<TEntity> entities) where TEntity : class;
        void ForceDatabaseInitialize();
    }
}
```

اکنون کلاس ApplicationDbContext منتقل شده به DataLayer یک چنین امضایی را خواهد یافت:

```
public class ApplicationDbContext :
    IdentityDbContext<ApplicationUser, CustomRole, int, CustomUserLogin, CustomUserRole,
    CustomUserClaim>,
    IUnitOfWork
{
    public DbSet<Category> Categories { set; get; }
    public DbSet<Product> Products { set; get; }
    public DbSet<Address> Addresses { set; get; }
```

تعریف آن باید جهت اعمال کلاس‌های سفارشی سازی شده‌ی کاربران و نقش‌های آن‌ها برای استفاده از primary key از نوع int به شکل فوق، تغییر یابد. همچنین در انتهای آن مانند قبل، IUnitOfWork نیز ذکر شده‌است. پیاده سازی کامل این کلاس را از پروژه‌ی پیوست انتهای بحث می‌توانید دریافت کنید. کار کردن با این کلاس، هیچ تفاوتی با DbContext‌های متداول EF Code First ندارد و تمام اصول آن‌ها یکی است.

در ادامه اگر به فایل App_Start\IdentityConfig.cs مراجعه کنید، کلاس ذیل در آن قابل مشاهده‌است:

```
public class ApplicationDbInitializer : DropCreateDatabaseIfModelChanges<ApplicationDbContext>
```

نیازی به این کلاس به این شکل نیست. آن را حذف کنید و در پروژه‌ی DataLayer، کلاس جدید ذیل را اضافه نمایید:

```
using System.Data.Entity.Migrations;

namespace AspNetIdentityDependencyInjectionSample.DataLayer.Context
{
    public class Configuration : DbMigrationsConfiguration<ApplicationDbContext>
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = true;
            AutomaticMigrationDataLossAllowed = true;
        }
    }
}
```

در این مثال، [بحث migrations](#) به حالت خودکار تنظیم شده‌است و تمام تغییرات در پروژه‌ی DomainClasses را به صورت خودکار به بانک اطلاعاتی اعمال می‌کند. تا همینجا کار تنظیم DataLayer به پایان می‌رسد.

ساختار پروژه‌ی `AspNetIdentityDependencyInjectionSample.ServiceLayer`

در ادامه مابقی کلاس‌های موجود در فایل `App_Start\IdentityConfig.cs` را به لایه سرویس برنامه منتقل خواهیم کرد. همچنین برای آن‌ها یک سری اینترفیس جدید نیز تعریف می‌کنیم، تا تزریق وابستگی‌ها به نحو صحیحی صورت گیرد. اگر به فایل‌های کنترلر این مثال پیش فرض مراجعه کنید (پیش از تغییرات بحث جاری)، هرچند به نظر در کنترلرها، کلاس‌های موجود در فایل `App_Start\IdentityConfig.cs` تزریق شده‌اند، اما به دلیل عدم استفاده از اینترفیس‌ها، وابستگی کاملی بین جزئیات پیاده سازی این کلاس‌ها و نمونه‌های تزریق شده به کنترلرها وجود دارد و عملاً معکوس سازی واقعی وابستگی‌ها رخ نداده‌است. بنابراین نیاز است این مسایل را اصلاح کنیم.

الف) انتقال کلاس `ApplicationUserManager` به لایه سرویس برنامه

کلاس `ApplicationUserManager` فایل `App_Start\IdentityConfig.c` را به لایه سرویس منتقل می‌کنیم:

```
using System;
using System.Security.Claims;
using System.Threading.Tasks;
using AspNetIdentityDependencyInjectionSample.DomainClasses;
using AspNetIdentityDependencyInjectionSample.ServiceLayer.Contracts;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin.Security.Cookies;
using Microsoft.Owin.Security.DataProtection;

namespace AspNetIdentityDependencyInjectionSample.ServiceLayer
{
    public class ApplicationUserManager
        : UserManager<ApplicationUser, int>, IApplicationUserManager
    {
        private readonly IDataProtectionProvider _dataProtectionProvider;
        private readonly IIdentityMessageService _emailService;
        private readonly IApplicationRoleManager _roleManager;
        private readonly IIdentityMessageService _smsService;
        private readonly IUserStore<ApplicationUser, int> _store;

        public ApplicationUserManager(IUserStore<ApplicationUser, int> store,
            IApplicationRoleManager roleManager,
            IDataProtectionProvider dataProtectionProvider,
            IIdentityMessageService smsService,
            IIdentityMessageService emailService)
            : base(store)
        {
            _store = store;
            _roleManager = roleManager;
            _dataProtectionProvider = dataProtectionProvider;
            _smsService = smsService;
            _emailService = emailService;

            createApplicationUserManager();
        }

        public void SeedDatabase()
        {
        }

        private void createApplicationUserManager()
        {
            // Configure validation logic for usernames
            this.UserValidator = new UserValidator<ApplicationUser, int>(this)
            {
                AllowOnlyAlphanumericUserNames = false,
                RequireUniqueEmail = true
            };

            // Configure validation logic for passwords
            this.PasswordValidator = new PasswordValidator
            {
                RequiredLength = 6,
                RequireNonLetterOrDigit = true,
                RequireDigit = true,
                RequireLowercase = true,
                RequireUppercase = true,
            };
        }
    }
}
```

```
// Configure user lockout defaults
this.UserLockoutEnabledByDefault = true;
this.DefaultAccountLockoutTimeSpan = TimeSpan.FromMinutes(5);
this.MaxFailedAccessAttemptsBeforeLockout = 5;

// Register two factor authentication providers. This application uses Phone and Emails as a step
of receiving a code for verifying the user
// You can write your own provider and plug in here.
this.RegisterTwoFactorProvider("PhoneCode", new PhoneNumberTokenProvider<ApplicationUser, int>
{
    MessageFormat = "Your security code is: {0}"
});
this.RegisterTwoFactorProvider("EmailCode", new EmailTokenProvider<ApplicationUser, int>
{
    Subject = "SecurityCode",
    BodyFormat = "Your security code is {0}"
});
this.EmailService = _emailService;
this.SmsService = _smsService;

if (_dataProtectionProvider != null)
{
    var dataProtector = _dataProtectionProvider.Create("ASP.NET Identity");
    this.UserTokenProvider = new DataProtectorTokenProvider<ApplicationUser, int>(dataProtector);
}
}
```

تغییراتی که در اینجا اعمال شده‌اند، به شرح زیر می‌باشند:

- متد استاتیک Create این کلاس حذف و تعاریف آن به سازنده‌ی کلاس منتقل شده‌اند. به این ترتیب با هربار وهله سازی این کلاس توسط IoC Container به صورت خودکار این تنظیمات نیز به کلاس پایه UserManager اعمال می‌شوند.
- اگر به کلاس پایه UserManager دقت کنید، به آرگومان‌های جنریک آن یک int هم اضافه شده‌است. این مورد جهت استفاده از primary key از نوع int ضروری است.
- در کلاس پایه UserManager تعدادی متد وجود دارند. تعاریف آن‌ها را به اینترفیس IApplicationUserManager منتقل خواهیم کرد. نیازی هم به پیاده سازی این متدها در کلاس جدید ApplicationUser نیست؛ زیرا کلاس پایه UserManager پیشتر آن‌ها را پیاده سازی کرده‌است. به این ترتیب می‌توان به یک تزریق وابستگی واقعی و بدون وابستگی به پیاده سازی خاص UserManager رسید. کنترلری که با IApplicationUserManager بجای ApplicationUser کار می‌کند، قابلیت تعویض پیاده سازی آن‌را جهت آزمون‌های واحد خواهد یافت.
- در کلاس اصلی ApplicationDbContext پیش فرض این مثال، متد Seed هم قابل مشاهده‌است. این متد را از کلاس جدید Configuration اضافه شده به DataLayer حذف کرده‌ایم. از این جهت که در آن از متدهای کلاس ApplicationUser مستقیماً استفاده شده‌است. متد Seed اکنون به کلاس جدید اضافه شده به لایه سرویس منتقل شده و در آغاز برنامه فراخوانی خواهد شد. DataLayer نباید وابستگی به لایه سرویس داشته باشد. لایه سرویس است که از امکانات DataLayer استفاده می‌کند.
- اگر به سازنده‌ی کلاس جدید ApplicationUser دقت کنید، چند اینترفیس دیگر نیز به آن تزریق شده‌اند. اینترفیس IApplicationRoleManager را ادامه تعریف خواهیم کرد. سایر اینترفیس‌های تزریق شده مانند IUserStore، IIdentityMessageService و IDataProtectionProvider جزو تعاریف اصلی ASP.NET Identity بوده و نیازی به تعریف مجدد آن‌ها نیست. فقط کلاس‌های EmailService و SmsService فایل App_Start\IdentityConfig.c را نیز به لایه سرویس منتقل کرده‌ایم. این کلاس‌ها بر اساس تنظیمات IoC Container مورد استفاده، در اینجا به صورت خودکار تزریق خواهند شد. حالت پیش فرض آن، وهله سازی مستقیم است که مطابق کدهای فوق به حالت تزریق وابستگی‌ها بهبود یافته‌است.

ب) انتقال کلاس ApplicationSignInManager به لایه سرویس برنامه

کلاس ApplicationSignInManager فایل App_Start\IdentityConfig.c را نیز به لایه سرویس منتقل می‌کنیم.

```
using AspNetIdentityDependencyInjectionSample.DomainClasses;
using AspNetIdentityDependencyInjectionSample.ServiceLayer.Contracts;
using Microsoft.AspNet.Identity.Owin;
using Microsoft.Owin.Security;

namespace AspNetIdentityDependencyInjectionSample.ServiceLayer
{
    public class ApplicationSignInManager :
```

```

SignInManager<ApplicationUser, int>, IApplicationSignInManager
{
    private readonly ApplicationUserManager _userManager;
    private readonly IAuthenticationManager _authenticationManager;

    public ApplicationSignInManager(ApplicationUserManager userManager,
        IAuthenticationManager authenticationManager) :
        base(userManager, authenticationManager)
    {
        _userManager = userManager;
        _authenticationManager = authenticationManager;
    }
}

```

در اینجا نیز اینترفیس جدید IApplicationSignInManager را برای مخفی سازی پیاده سازی کلاس پایه توکار SignInManager، اضافه کرده‌ایم. این اینترفیس دقیقاً حاوی تعاریف متدهای کلاس پایه SignInManager است و نیازی به پیاده سازی مجدد در کلاس ApplicationSignInManager نخواهد داشت.

ج) انتقال کلاس ApplicationRoleManager به لایه سرویس برنامه

کلاس ApplicationRoleManager فایل App_Start\IdentityConfig.c را نیز به لایه سرویس منتقل خواهیم کرد:

```

using AspNetIdentityDependencyInjectionSample.DomainClasses;
using AspNetIdentityDependencyInjectionSample.ServiceLayer.Contracts;
using Microsoft.AspNet.Identity;

namespace AspNetIdentityDependencyInjectionSample.ServiceLayer
{
    public class ApplicationRoleManager : RoleManager<CustomRole, int>, IApplicationRoleManager
    {
        private readonly IRoleStore<CustomRole, int> _roleStore;
        public ApplicationRoleManager(IRoleStore<CustomRole, int> roleStore)
            : base(roleStore)
        {
            _roleStore = roleStore;
        }

        public CustomRole FindRoleByName(string roleName)
        {
            return this.FindByName(roleName); // RoleManagerExtensions
        }

        public IdentityResult CreateRole(CustomRole role)
        {
            return this.Create(role); // RoleManagerExtensions
        }
    }
}

```

روش کار نیز در اینجا همانند دو کلاس قبل است. اینترفیس جدید IApplicationRoleManager را که حاوی تعاریف متدهای کلاس پایه توکار RoleManager است، به لایه سرویس اضافه می‌کنیم. کنترلرهای برنامه با این اینترفیس بجای استفاده مستقیم از کلاس ApplicationRoleManager کار خواهند کرد.

تا اینجا کار تنظیمات لایه سرویس برنامه به پایان می‌رسد.

ساختار پروژه‌ی AspNetIdentityDependencyInjectionSample.IocConfig

پروژه‌ی IocConfig جایی است که تنظیمات StructureMap را به آن منتقل کرده‌ایم:

```

using System;
using System.Data.Entity;
using System.Threading;
using System.Web;
using AspNetIdentityDependencyInjectionSample.DataLayer.Context;

```

```

using AspNetIdentityDependencyInjectionSample.DomainClasses;
using AspNetIdentityDependencyInjectionSample.ServiceLayer;
using AspNetIdentityDependencyInjectionSample.ServiceLayer.Contracts;
using Microsoft.AspNet.Identity;
using Microsoft.AspNet.Identity.EntityFramework;
using Microsoft.Owin.Security;
using StructureMap;
using StructureMap.Web;

namespace AspNetIdentityDependencyInjectionSample.IocConfig
{
    public static class SmObjectFactory
    {
        private static readonly Lazy<Container> _containerBuilder =
            new Lazy<Container>(defaultContainer, LazyThreadSafetyMode.ExecutionAndPublication);

        public static IContainer Container
        {
            get { return _containerBuilder.Value; }
        }

        private static Container defaultContainer()
        {
            return new Container(ioc =>
            {
                ioc.For<IUnitOfWork>()
                    .HybridHttpOrThreadLocalScoped()
                    .Use<ApplicationDbContext>();

                ioc.For<ApplicationDbContext>().HybridHttpOrThreadLocalScoped().Use<ApplicationDbContext>();
                ioc.For<DbContext>().HybridHttpOrThreadLocalScoped().Use<ApplicationDbContext>();

                ioc.For<IUserStore<ApplicationUser, int>>()
                    .HybridHttpOrThreadLocalScoped()
                    .Use<UserStore<ApplicationUser, CustomRole, int, CustomUserLogin, CustomUserRole, CustomUserClaim>>());

                ioc.For<IRoleStore<CustomRole, int>>()
                    .HybridHttpOrThreadLocalScoped()
                    .Use<RoleStore<CustomRole, int, CustomUserRole>>());

                ioc.For<IAuthenticationManager>()
                    .Use(() => HttpContext.Current.GetOwinContext().Authentication);

                ioc.For<IApplicationSignInManager>()
                    .HybridHttpOrThreadLocalScoped()
                    .Use<ApplicationSignInManager>();

                ioc.For<IApplicationUserManager>()
                    .HybridHttpOrThreadLocalScoped()
                    .Use<ApplicationUserManager>();

                ioc.For<IApplicationRoleManager>()
                    .HybridHttpOrThreadLocalScoped()
                    .Use<ApplicationRoleManager>();

                ioc.For<IIIdentityMessageService>().Use<SmsService>();
                ioc.For<IIIdentityMessageService>().Use<EmailService>();
                ioc.For<ICustomRoleStore>()
                    .HybridHttpOrThreadLocalScoped()
                    .Use<CustomRoleStore>();

                ioc.For<ICustomUserStore>()
                    .HybridHttpOrThreadLocalScoped()
                    .Use<CustomUserStore>();

                //config.For<IDataProtectionProvider>().Use(()=> app.GetDataProtectionProvider()); //
            });
        }
    }
}

```

In Startup class

```

ioc.For<ICategoryService>().Use<EfCategoryService>();
ioc.For<IProductService>().Use<EfProductService>();
}
}

```

در اینجا نحوه‌ی اتصال اینترفیس‌های برنامه را به کلاس‌ها و یا نمونه‌هایی که آن‌ها را می‌توانند پیاده سازی کنند، مشاهده می‌کنید.

برای مثال IUnitOfWork به ApplicationDbContext مرتبط شده‌است و یا دوبار تعاریف متناظر با DbContext را مشاهده می‌کنید. از این تعاریف به صورت توکار توسط ASP.NET Identity زمانیکه قرار است UserStore و RoleStore را و هله سازی کند، استفاده می‌شوند و ذکر آن‌ها الزامی است.

در تعاریف فوق یک مورد را به فایل Startup.cs موکول کرده‌ایم. برای مشخص سازی نمونه‌ی پیاده سازی کننده‌ی IDataProtectionProvider نیاز است به IApplicationBuilder کلاس Startup برنامه دسترسی داشت. این کلاس آغازین Owin اکنون به نحو ذیل بازنویسی شده‌است و در آن، تنظیمات IDataProtectionProvider را به همراه و هله سازی CreatePerOwinContext مشاهده می‌کنید:

```
using System;
using AspNetIdentityDependencyInjectionSample.IocConfig;
using AspNetIdentityDependencyInjectionSample.ServiceLayer.Contracts;
using Microsoft.AspNet.Identity;
using Microsoft.Owin;
using Microsoft.Owin.Security.Cookies;
using Microsoft.Owin.Security.DataProtection;
using Owin;
using StructureMap.Web;

namespace AspNetIdentityDependencyInjectionSample
{
    public class Startup
    {
        public void Configuration(IApplicationBuilder app)
        {
            configureAuth(app);
        }

        private static void configureAuth(IApplicationBuilder app)
        {
            SmObjectFactory.Container.Configure(config =>
            {
                config.For<IDataProtectionProvider>()
                    .HybridHttpOrThreadLocalScoped()
                    .Use(() => app.GetDataProtectionProvider());
            });
            SmObjectFactory.Container.GetInstance<IApplicationUserManager>().SeedDatabase();

            // Configure the db context, user manager and role manager to use a single instance per
            request
            app.CreatePerOwinContext(() =>
            SmObjectFactory.Container.GetInstance<IApplicationUserManager>());

            // Enable the application to use a cookie to store information for the signed in user
            // and to use a cookie to temporarily store information about a user logging in with a
            third party login provider
            // Configure the sign in cookie
            app.UseCookieAuthentication(new CookieAuthenticationOptions
            {
                AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
                LoginPath = new PathString("/Account/Login"),
                Provider = new CookieAuthenticationProvider
                {
                    // Enables the application to validate the security stamp when the user logs in.
                    // This is a security feature which is used when you change a password or add an
                    external login to your account.
                    OnValidateIdentity =
                    SmObjectFactory.Container.GetInstance<IApplicationUserManager>().OnValidateIdentity()
                }
            });
            app.UseExternalSignInCookie(DefaultAuthenticationTypes.ExternalCookie);

            // Enables the application to temporarily store user information when they are verifying
            the second factor in the two-factor authentication process.
            app.UseTwoFactorSignInCookie(DefaultAuthenticationTypes.TwoFactorCookie,
            TimeSpan.FromMinutes(5));

            // Enables the application to remember the second login verification factor such as phone
            or email.
            // Once you check this option, your second step of verification during the login process
            will be remembered on the device where you logged in from.
            // This is similar to the RememberMe option when you log in.
            app.UseTwoFactorRememberBrowserCookie(DefaultAuthenticationTypes.TwoFactorRememberBrowserCookie);
        }
    }
}
```

}

این تعاریف از فایل پیش فرض Startup.Auth.cs پوشه‌ی App_Start دریافت و جهت کار با IoC Container برنامه، بازنویسی شده‌اند.

تنظیمات برنامه‌ی اصلی ASP.NET MVC، جهت اعمال تزریق وابستگی‌ها

الف) ابتدا نیاز است فایل Global.asax.cs را به نحو ذیل بازنویسی کنیم:

```
using System;
using System.Data.Entity;
using System.Web;
using System.Web.Mvc;
using System.Web.Optimization;
using System.Web.Routing;
using AspNetIdentityDependencyInjectionSample.DataLayer.Context;
using AspNetIdentityDependencyInjectionSample.IocConfig;
using StructureMap.Web.Pipeline;

namespace AspNetIdentityDependencyInjectionSample
{
    public class MvcApplication : HttpApplication
    {
        protected void Application_Start()
        {
            AreaRegistration.RegisterAllAreas();
            FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);

            setDbInitializer();
            //Set current Controller factory as StructureMapControllerFactory
            ControllerBuilder.Current.SetControllerFactory(new StructureMapControllerFactory());
        }

        protected void Application_EndRequest(object sender, EventArgs e)
        {
            HttpContextLifecycle.DisposeAndClearAll();
        }

        public class StructureMapControllerFactory : DefaultControllerFactory
        {
            protected override IController GetControllerInstance(RequestContext requestContext, Type controllerType)
            {
                if (controllerType == null)
                    throw new InvalidOperationException(string.Format("Page not found: {0}",
                        requestContext.HttpContext.Request.RawUrl));
                return SmObjectFactory.Container.GetInstance(controllerType) as Controller;
            }

            private static void setDbInitializer()
            {
                Database.SetInitializer(new MigrateDatabaseToLatestVersion<ApplicationDbContext,
                    Configuration>());
                SmObjectFactory.Container.GetInstance<IUnitOfWork>().ForceDatabaseInitialize();
            }
        }
    }
}
```

در اینجا در متد setDbInitializer، نحوه‌ی استفاده و تعریف فایل Configuration لایه Data را ملاحظه می‌کنید؛ به همراه متد آغاز بانک اطلاعاتی و اعمال تغییرات لازم به آن در ابتدای کار برنامه. همچنین ControllerFactory برنامه نیز به StructureMapControllerFactory تنظیم شده‌است تا کار تزریق وابستگی‌ها به کنترلرهای برنامه به صورت خودکار میسر شود. در پایان کار هر درخواست نیز منابع Disposable رها می‌شوند.

ب) به پوشه‌ی Models برنامه مراجعه کنید. در اینجا در هر کلاسی که Id از نوع string وجود داشت، باید تبدیل به نوع int

شوند. چون primary key برنامه را به نوع int تغییر داده‌ایم. برای مثال کلاس‌های EditUserViewModel و RoleViewModel باید تغییر کنند.

(ج) اصلاح کنترلرهای برنامه جهت اعمال تزریق وابستگی‌ها

اکنون اصلاح کنترلرها جهت اعمال تزریق وابستگی‌ها ساده‌است. در ادامه نحوه‌ی تغییر امضای سازنده‌های این کنترلرها را جهت استفاده از اینترفیس‌های جدید مشاهده می‌کنید:

```
[Authorize]
public class AccountController : Controller
{
    private readonly IAuthenticationManager _authenticationManager;
    private readonly IApplicationSignInManager _signInManager;
    private readonly IApplicationUserManager _userManager;
    public AccountController(IApplicationUserManager userManager,
                           IApplicationSignInManager signInManager,
                           IAuthenticationManager authenticationManager)
    {
        _userManager = userManager;
        _signInManager = signInManager;
        _authenticationManager = authenticationManager;
    }

    [Authorize]
    public class ManageController : Controller
    {
        // Used for XSRF protection when adding external logins
        private const string XsrfKey = "XsrfId";

        private readonly IAuthenticationManager _authenticationManager;
        private readonly IApplicationUserManager _userManager;
        public ManageController(IApplicationUserManager userManager, IAuthenticationManager authenticationManager)
        {
            _userManager = userManager;
            _authenticationManager = authenticationManager;
        }

        [Authorize(Roles = "Admin")]
        public class RolesAdminController : Controller
        {
            private readonly IApplicationRoleManager _roleManager;
            private readonly IApplicationUserManager _userManager;
            public RolesAdminController(IApplicationUserManager userManager,
                                       IApplicationRoleManager roleManager)
            {
                _userManager = userManager;
                _roleManager = roleManager;
            }

            [Authorize(Roles = "Admin")]
            public class UsersAdminController : Controller
            {
                private readonly IApplicationRoleManager _roleManager;
                private readonly IApplicationUserManager _userManager;
                public UsersAdminController(IApplicationUserManager userManager,
                                           IApplicationRoleManager roleManager)
                {
                    _userManager = userManager;
                    _roleManager = roleManager;
                }
            }
        }
    }
}
```

پس از این تغییرات، فقط کافی است بجای خواص برای مثال RoleManager سابق از فیلدهای تزریق شده در کلاس، مثلاً _roleManager جدید استفاده کرد. امضای متدهای یکی است و تنها به یک search و replace نیاز دارد. البته تعدادی اکشن متد نیز در اینجا وجود دارند که از string id استفاده می‌کنند. این‌ها را باید به int? Id تغییر داد تا با نوع primary key جدید مورد استفاده تطابق پیدا کنند.

کدهای کامل این مثال را از اینجا می‌توانید دریافت کنید:

نظرات خوانندگان

نویسنده: سیروان عقیفی
تاریخ: ۱۶:۱۱ ۱۳۹۳/۱۰/۰۵

ممنون از مطلب شما؛

برای تغییر نام جداول تشکیل شده، درون متد OnModelCreating کد زیر را نوشتم اما با بروز رسانی دیتابیس تغییری در اسامی جداول حاصل نشد:

```
modelBuilder.Entity<ApplicationUser>().ToTable("User").Property(p => p.Id).HasColumnName("Id");
modelBuilder.Entity<CustomUserRole>().ToTable("UserRole").HasKey(p => new { p.RoleId,
p.UserId });
modelBuilder.Entity<CustomUserLogin>().ToTable("UserLogin").HasKey(p => new {
p.LoginProvider, p.ProviderKey, p.UserId });
modelBuilder.Entity<CustomUserClaim>().ToTable("UserClaim").HasKey(p => p.Id).Property(p =>
p.Id).HasColumnName("UserClaimId");
modelBuilder.Entity<CustomRole>().ToTable("Role").Property(p =>
p.Id).HasColumnName("RoleId");
```

نویسنده: وحید نصیری
تاریخ: ۱۸:۵۳ ۱۳۹۳/۱۰/۰۵

این روش پس از امتحان، [جواب داد](#) :

```
protected override void OnModelCreating(DbModelBuilder builder)
{
    base.OnModelCreating(builder);

    builder.Entity<ApplicationUser>().ToTable("Users");
    builder.Entity<CustomRole>().ToTable("Roles");
    builder.Entity<CustomUserClaim>().ToTable("UserClaims");
    builder.Entity<CustomUserRole>().ToTable("UserRoles");
    builder.Entity<CustomUserLogin>().ToTable("UserLogins");
}
```

اگر متد base.OnModelCreating(builder) را در انتهای کار قرار دهید، تنظیمات پیش فرض کلاس پایه IdentityDbContext (یعنی همان [نام‌های قدیمی](#)) اعمال می‌شوند و تنظیمات شما بازنویسی خواهند شد. این متد باید در ابتدای کار فراخوانی شود.

نویسنده: فخاری
تاریخ: ۱۹:۲۱ ۱۳۹۳/۱۰/۰۸

با سلام

ممنون از مطلب مفید شما. واقعا عالی بود ☺

ولی قسمتی از کد برام نامفهوم بود :

```
public ApplicationDbContext()
    : base("connectionString1")
{
    //this.Database.Log = data => System.Diagnostics.Debug.WriteLine(data);
    //فقط تعریف شده تا یک برک پوینت در اینجا قرار داده شود برای آزمایش تعداد بار فراخوانی آن
}

protected override void Dispose(bool disposing)
{
    base.Dispose(disposing);
    //فقط تعریف شده تا یک برک پوینت در اینجا قرار داده شود برای آزمایش فراخوانی آن
}
```

مخصوصا بخش Dispose .

چون قبلا که از الگوی واحد کار استفاده می‌شد این بخش وجود نداشت !
آیا وجود این کدها که در بالا اومده الزامی است؟

نویسنده: وحید نصیری
تاریخ: ۱۹:۲۴ ۱۳۹۳/۱۰/۰۸

خیر. متد Dispose را حذف کنید (در کلاس پایه هست). وجود سازنده هم صرفا جهت ساده سازی تعریف و انتخاب رشته اتصال تعریف شده در web.config است.

نویسنده: وحید نصیری
تاریخ: ۱۳:۲۷ ۱۳۹۳/۱۰/۱۱

نکته‌ای مهم در مورد مدیریت استراکچرمپ در این مثال

اگر از Sm ObjectFactory مطلب فوق استفاده می‌کنید، Container آن با ObjectFactory یکی نیست یا به عبارتی ObjectFactory اطلاعاتی در مورد تنظیمات کلاس سفارشی Sm ObjectFactory ندارد. بنابراین دیگر نباید از ObjectFactory قدیمی استفاده کنید. در این حالت هرجایی ObjectFactory قدیمی را داشتید، با SmObjectFactory.Container تعویض می‌شود.

نویسنده: صابر فتح الهی
تاریخ: ۱۸:۴۶ ۱۳۹۳/۱۰/۱۴

نحوه دسترسی به کاربری که لاگین کرده چطوره؟
من با دستور HttpContext.Current.User کار میکنم
بعضی اوقات نال بر میگرددونه

نویسنده: وحید نصیری
تاریخ: ۱۹:۱۷ ۱۳۹۳/۱۰/۱۴

User.Identity را در مثال فوق جستجو کنید:

```
User.Identity.GetUserName()  
User.Identity.GetUserId()
```

نویسنده: سیروان عقیفی
تاریخ: ۱۹:۲۱ ۱۳۹۳/۱۰/۱۴

برای مشخص کردن نمونه پیاده‌سازی کننده IDataProtectionProvider در یک برنامه کنسول نیز باید از فایل Startup استفاده کرد؟ بیشتر هدفم Seed کردن دیتابیس است (مثلا ایمپورت تعداد زیادی کاربر از طریق یک فایل و...). اینکار رو در متد SeedDatabase هم انجام دادم ولی هر بار استثنای UserId not found رو در:

```
result = this.SetLockoutEnabled(user.Id, false);
```

صادر میکنه، با گذاشتن Breakpoint متوجه شدم که برای Id صفر رو در نظر میگیره! از این جهت ترجیح دادم برای اینکار از طریق برنامه کنسول ویندوزی هم آن را تست کنم، مثل روشی که در [اینجا](#) برای ایجاد کاربر نوشته شده.

نویسنده: صابر فتح الهی
تاریخ: ۱:۱۵ ۱۳۹۳/۱۰/۱۵

در قسمت تزریق وابستگی کد زیر وجود داره

```
ioc.For<IIIdentityMessageService>().Use<SmsService>();  
ioc.For<IIIdentityMessageService>().Use<EmailService>();
```

در صورت استفاده از اینترفیس مربوطه کدام سرویس نمونه سازی میشه؟ در صورتی که هر دو سرویس وجود داشته باشه؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۰/۱۵ ۱:۵۵

این مشکل [اصلاح شد](#) . باید از named instance در این حالت خاص استفاده شود.

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۳/۱۰/۱۵ ۶:۰۶

با انجام اصلاحات فوق، زمانی که کاربر اقدام به تایید شماره تلفن همراه کنه با اینکه PhoneCode انتخاب میشه اما بازم قسمت ارسال ایمیل اجرا میشه.

نویسنده: سیروان عقیفی
تاریخ: ۱۳۹۳/۱۰/۱۵ ۱۱:۴۸

از همان متد SeedDatabase استفاده کردم، مشکل این بود که در متد Create نوع پسورد از این لحاظ که حداقل باید 6 کاراکتر باشه و درست بودن ایمیل و... نیز بررسی میشه، اگر مقادیر معتبر نباشه مقدار user.Id برابر با صفر میشه.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۰/۱۵ ۱۳:۲۲

نیاز به تنظیمات setter injection هم داشت که [اضافه شد](#) .

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۳/۱۰/۱۵ ۲۰:۲۳

با این تنظیمات

```
// map same interface to different concrete classes
ioc.For<IIIdentityMessageService>().Use<SmsService>().Named("smsService");
ioc.For<IIIdentityMessageService>().Use<EmailService>().Named("emailService");
ioc.For<IApplicationUserManager>().HybridHttpOrThreadLocalScoped()
    .Use<ApplicationUserManager>()
    .Ctor<IIIdentityMessageService>("smsService").Is<SmsService>()
    .Ctor<IIIdentityMessageService>("emailService").Is<EmailService>()
    .Setter<IIIdentityMessageService>(userManager =>
userManager.SmsService).Is<SmsService>()
    .Setter<IIIdentityMessageService>(userManager =>
userManager.EmailService).Is<EmailService>());
```

و این کد سازنده UserManager

```
public ApplicationUserManager(
    IApplicationUserStore store,
    IApplicationRoleManager roleManager,
    IDataProtectionProvider dataProtectionProvider,
    IIIdentityMessageService smsService,
    IIIdentityMessageService emailService)
    : base(store as IUserStore<User,int>)
{
    this.roleManager = roleManager;
    this.dataProtectionProvider = dataProtectionProvider;
    EmailService = emailService;
    SmsService = smsService;
    Debug.WriteLine("Ticks = {0}",DateTime.Now.Ticks);
    Debug.WriteLine(emailService.ToString());
    Debug.WriteLine(smsService.ToString());
    CreateApplicationUserManager();
}
```

در زمان دیباگ اینجور خروجی توی پنجره دیباگ گرفتم

```
Ticks = 635560859158196052
PWS.ServiceLayer.EF.Identity.EmailService
PWS.ServiceLayer.EF.Identity.SmsService
Ticks = 635560859158246098
PWS.ServiceLayer.EF.Identity.EmailService
PWS.ServiceLayer.EF.Identity.EmailService
```

نمی‌دونم چرا دوبار سازنده فراخوانی شده در بار اول تزریق وابستگی درست انجام شده اما در آخرین بار هم سرویس ایمیل به ایمیل و پیامک هر تزریق شده.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۰/۱۶ ۲:۴۵

جهت مدیریت تک و هله‌ای ApplicationUserManger [این تغییرات](#) را اعمال کنید؛ یا از این [فایل نهایی](#) استفاده کنید.

تست واحد چیست؟

تست واحد ابزاری است برای مشاهده چگونگی عملکرد یک متد که توسط خود برنامه نویس نوشته میشود. به این صورت که پارامترهای ورودی، برای یک متد ساخته شده و آن متد فراخوانی و خروجی متد بسته به حالت مطلوب بررسی میشود. چنانچه خروجی مورد نظر مطلوب باشد تست واحد با موفقیت انجام میشود.

اهمیت انجام تست واحد چیست؟

درستی یک متد، مهمترین مسئله برای بررسی است و بارها مشاهده شده، استثناهایی رخ میدهند که توان تولید را به دلیل فرسایش تکراری رخداد می‌کاهند. نوشتن تست واحد منجر به این می‌شود چنانچه بعدها تغییری در بیزنس متد ایجاد شود و ورودی و خروجی‌ها تغییر نکند، صحت این تغییر بیزنس، توسط تست بررسی میشود؛ حتی میتوان این تست‌ها را در build پروژه قرار داد و در ابتدای اجرای یک Solution تمامی تست‌ها اجرا و درستی بخش به بخش اعضا چک شوند.

شروع تست واحد:

یک پروژه‌ی ساده را داریم برای تعریف حساب‌های بانکی شامل نام مشتری، مبلغ سپرده، وضعیت و 3 متد واریز به حساب و برداشت از حساب و تغییر وضعیت حساب که به صورت زیر است:

```
/// <summary>
/// حساب بانکی
/// </summary>
public class Account
{
    /// <summary>
    /// مشتری
    /// </summary>
    public string Customer { get; set; }
    /// <summary>
    /// موجودی حساب
    /// </summary>
    public float Balance { get; set; }
    /// <summary>
    /// وضعیت
    /// </summary>
    public bool Active { get; set; }

    public Account(string customer, float balance)
    {
        Customer = customer;
        Balance = balance;
        Active = true;
    }
    /// <summary>
    /// افزایش موجودی / واریز به حساب
    /// </summary>
    /// <param name="amount">مبلغ واریز</param>
    public void Credit(float amount)
    {
        if (!Active)
            throw new Exception("این حساب مسدود است");
        if (amount < 0)
            throw new ArgumentOutOfRangeException("amount");
        Balance += amount;
    }
    /// <summary>
    /// کاهش موجودی / برداشت از حساب
    /// </summary>
    /// <param name="amount">مبلغ برداشت</param>
    public void Debit(float amount)
    {

```

```

        if (!Active)
            throw new Exception("این حساب مسدود است.");
        if (amount < 0)
            throw new ArgumentOutOfRangeException("amount");
        if (Balance < amount)
            throw new ArgumentOutOfRangeException("amount");
        Balance -= amount;
    }
    /// <summary>
    /// انسداد / رفع انسداد
    /// </summary>
    public void ChangeStateAccount()
    {
        Active = !Active;
    }
}

```

تابع اصلی نیز به صورت زیر است:

```

class Program
{
    static void Main(string[] args)
    {
        var account = new Account("Ali", 1000);

        account.Credit(4000);
        account.Debit(2000);
        Console.WriteLine("Current balance is ${0}", account.Balance);
        Console.ReadKey();
    }
}

```

به Solution، یک پروژه از نوع تست واحد اضافه میکنیم.

در این پروژه ابتدا Reference ایی از پروژه‌ای که مورد تست هست میگیریم. سپس در کلاس تست مربوطه شروع به نوشتن متدی برای انواع تست متدهای پروژه اصلی میکنیم.

توجه داشته باشید که Data Annotation های بالای کلاس تست و متدهای تست، در تعیین نوع نگاه کامپایلر به این بلوک‌ها موثر است و باید این مسئله به درستی رعایت شود. همچنین در صورت نیاز میتوان از کلاس Startup برای شروع تست استفاده کرد که عمدتاً برای تعریف آن از نام ClassInit استفاده میشود و در بالای آن از [ClassInitialize] استفاده میشود. در Library تست واحد میتوان به دو صورت چگونگی صحت عملکرد یک تست را بررسی کرد: با استفاده از Assert و با استفاده از ExpectedException، که در زیر به هر دو صورت آن میپردازیم.

```

[TestClass]
public class UnitTest
{
    /// <summary>
    /// تعریف حساب جدید و بررسی تمامی فرآیندهای معمول روی حساب
    /// </summary>
    [TestMethod]
    public void Create_New_Account_And_Check_The_Process()
    {
        //Arrange
        var account = new Account("Hassan", 4000);
        var account2 = new Account("Ali", 10000);
        //Act
        account.Credit(5000);
        account2.Debit(3000);
        account.ChangeStateAccount();
        account2.Active = false;
        account2.ChangeStateAccount();
        //Assert
        Assert.AreEqual(account.Balance, 9000);
        Assert.AreEqual(account2.Balance, 7000);
        Assert.IsTrue(account2.Active);
        Assert.AreEqual(account.Active, false);
    }
}

```

همانطور که مشاهده میشود ابتدا در قسمت Arrange، خوراک تست آماده میشود. سپس در قسمت Act، فعالیت‌هایی که زیر ذره

بین تست هستند صورت می‌پذیرند و سپس در قسمت Assert درستی مقادیر با مقادیر مورد انتظار ما مطابقت داده میشوند. برای بررسی خطاهای تعیین شده هنگام نوشتن یک متد نیز میتوان به صورت زیر عمل کرد:

```
/// <summary>
/// زمانی که کاربر بخواهد به یک حساب مسدود واریز کند باید جلوی آن گرفته شود.
/// </summary>
[TestMethod]
[ExpectedException(typeof (Exception))]
public void When_Deactive_Account_Wants_To_add_Credit_Should_Throw_Exception()
{
    //Arrange
    var account = new Account("Hassan", 4000) {Active = false};
    //Act
    account.Credit(4000);
    //Assert
    //Assert is handled with ExpectedException
}

[TestMethod]
[ExpectedException(typeof (ArgumentOutOfRangeException))]
public void
When_Customer_Wants_To_Debit_More_Than_Balance_Should_Throw_ArgumentOutOfRangeException()
{
    //Arrange
    var account = new Account("Hassan", 4000);
    //Act
    account.Debit(5000);
    //Assert
    //Assert is handled with ArgumentOutOfRangeException
}
```

همانطور که مشخص است نام متد تست باید کامل و شفاف به صورتی انتخاب شود که بیانگر رخداد درون متد تست باشد. در این متدها Assert مورد انتظار با DataAnnotation که پیش از این توضیح داده شد کنترل گردیده است و بدین صورت کار میکند که وقتی Act انجام میشود، متد بررسی می‌کند تا آن Assert رخ بدهد.

استفاده از Library Moq در تست واحد

ابتدا باید به این توضیح بپردازیم که این کتابخانه چه کاری میکند و چه امکانی را برای انجام تست واحد فراهم میکند. در پروژه‌های بزرگ و زمانی که ارتباطات بین لایه‌های زیادی موجود است و اصول SOLID رعایت میشود، شما در یک لایه برای ارائه فعالیت‌ها و خدمات متدهایتان با Interface های لایه‌های دیگر در ارتباط هستید و برای نوشتن تست واحد متدهایتان، مشکلی بزرگ دارید که نمیتوانید به این لایه‌ها دسترسی داشته باشید و ماهیت تست واحد را زیر سوال میبرید. Library Moq این امکان را به شما میدهد که از این Interface ها یک تصویر مجازی بسازید و همانند Snap Shot با آن کار کنید؛ بدون اینکه در لایه‌های دیگر بروید و ماهیت تست واحد را زیر سوال ببرید.

برای استفاده از متدهایی که در این Interface ها موجود است شما باید یک شیء از نوع Mock<> از آنها بسازید و سپس با استفاده از متد Setup به صورت مجازی متد مورد نظر را فراخوانی کنید و مقدار بازگشتی مورد انتظار را با Return معرفی کنید، سپس از آن استفاده کنید.

همچنین برای دسترسی به خود شیء از Property ایی با نام Objeet از موجودیت mock شده استفاده میکنیم. برای شناسایی بهتر اینکه از چه اینترفیس هایی باید Mock<> بسازید، میتوانید به متد سازنده کلاسی که معرف لایه ایست که برای آن تست واحد مینویسید، مراجعه کنید.

نحوه اجرای یک تست واحد با استفاده از Moq با توجه به توضیحات بالا به صورت زیر است:

پروژه مورد بررسی لایه Service برای تعریف واحدهای سازمانی است که با الگوریتم DDD و CQRS پیاده سازی شده است. ابتدا به Constructor خود لایه سرویس نگاه میکنیم تا بتوانید شناسایی کنید از چه Interface هایی باید Mock<> کنیم.

```
public class OrganizationalService : ICommandHandler<CreateUnitTypeCommand>,
    ICommandHandler<DeleteUnitTypeCommand>,
{
    private readonly IUnitOfWork _unitOfWork;
    private readonly IUnitTypeRepository _unitTypeRepository;
    private readonly IOrganizationUnitRepository _organizationUnitRepository;
    private readonly IOrganizationUnitDomainService _organizationUnitDomainService;
```



```

    public OrganizationalService(IUnitOfWork unitOfWork, IUnitTypeRepository unitTypeRepository,
        IOrganizationUnitRepository organizationUnitRepository, IOrganizationUnitDomainService
        organizationUnitDomainService)
    {
        _unitOfWork = unitOfWork;
        _unitTypeRepository = unitTypeRepository;
        _organizationUnitRepository = organizationUnitRepository;
        _organizationUnitDomainService = organizationUnitDomainService;
    }

```

مشاهده میکنید که 4 Interface استفاده شده و در متد سازنده نیز مقدار دهی شده اند. پس 4 Mock نیاز داریم. در پروژه تست به صورت زیر و در ClassInitialize عمل میکنیم.

```

[TestClass]
public class OrganizationServiceTest
{
    private static OrganizationalService _organizationalService;
    private static Mock<IUnitTypeRepository> _mockUnitTypeRepository;
    private static Mock<IUnitOfWork> _mockUnitOfWork;
    private static Mock<IOrganizationUnitRepository> _mockOrganizationUnitRepository;
    private static Mock<IOrganizationUnitDomainService> _mockOrganizationUnitDomainService;

    [ClassInitialize]
    public static void ClassInit(TestContext context)
    {
        TestBootstrapper.ConfigureDependencies();
        _mockUnitOfWork = new Mock<IUnitOfWork>();
        _mockUnitTypeRepository = new Mock<IUnitTypeRepository>();
        _mockOrganizationUnitRepository = new Mock<IOrganizationUnitRepository>();
        _mockOrganizationUnitDomainService = new Mock<IOrganizationUnitDomainService>();
        _organizationalService = new OrganizationalService(_mockUnitOfWork.Object,
        _mockUnitTypeRepository.Object,
        _mockOrganizationUnitRepository.Object, _mockOrganizationUnitDomainService.Object);
    }
}

```

از خود لایه سرویس با نام OrganizationService یک آبجکت میگیریم و 4 واسط دیگر به صورت Mock شده تعریف میشوند. همچنین در کلاس بارگذار از همان نوع مقدار دهی میگردند تا در اجرای تمامی متدهای تست، در دست کامپایلر باشند. همچنین برای new کردن خود سرویس از mock.obect که حاوی مقدار اصلی است استفاده میکنیم. خود متد اصلی به صورت زیر است:

```

/// <summary>
/// یک نوع واحد سازمانی را حذف مینماید
/// </summary>
/// <param name="command"></param>
public void Handle>DeleteUnitTypeCommand command)
{
    var unitType = _unitTypeRepository.FindBy(command.UnitTypeId);
    if (unitType == null)
        throw new DeleteEntityNotFoundException();

    ICanDeleteUnitTypeSpecification canDeleteUnitType = new
    CanDeleteUnitTypeSpecification(_organizationUnitRepository);
    if (canDeleteUnitType.IsSatisfiedBy(unitType))
        throw new UnitTypeIsUnderUsingException(unitType.Title);
    _unitTypeRepository.Remove(unitType);
}

```

متدهای تست این متد نیز به صورت زیر هستند:

```

/// <summary>
/// کامند حذف نوع واحد سازمانی باید به درستی حذف کند.
/// </summary>
[TestMethod]
public void DeleteUnitTypeCommand_Should_Delete_UnitType()
{
    //Arrange
    var unitTypeId = new Guid();
    var deleteUnitTypeCommand = new DeleteUnitTypeCommand { UnitTypeId = unitTypeId };
    var unitType = new UnitType("خوشه");
    var org = new List<OrganizationUnit>();
}

```

```

        _mockUnitTypeRepository.Setup(d =>
d.FindBy(deleteUnitTypeCommand.UnitTypeId)).Returns(unitType);
        _mockUnitTypeRepository.Setup(x => x.Remove(unitType));
        _mockOrganizationUnitRepository.Setup(z => z.FindBy(unitType)).Returns(org);
        try
        {
            //Act
            _organizationalService.Handle(deleteUnitTypeCommand);
        }
        catch (Exception ex)
        {
            //Assert
            Assert.Fail(ex.Message);
        }
    }
}

```

همانطور که مشاهده میشود ابتدا یک Guid به عنوان آی دی نوع واحد سازمانی گرفته میشود و همان آی دی برای تعریف کامند حذف به آن ارسال میشود. سپس یک نوع واحد سازمانی دلخواه تستی ساخته میشود و همچنین یک لیست خالی از واحدهای سازمانی که برای چک شدن توسط خود متد Handle استفاده شده است ساخته میشود. در اینجا این متد خالی است تا شرط غلط شود و عمل حذف به درستی صورت پذیرد.

برای اعمالی که در Handle انجام میشود و متدهایی که از Interface ها صدا زده میشوند Setup میکنیم و آنهایی را که Return دارند به object هایی که مورد انتظار خودمان هست نسبت میدهیم. در Setup اول میگوییم که آن Guid مربوط به "خوشه" است. در Setup بعدی برای عمل Remove کدی مینویسیم و چون عمل حذف Return ندارد میتواند، این خط به کل حذف شود! به طور کلی Setup هایی که Return ندارند میتوانند حذف شوند. در Setup بعدی از Interface دیگر متد FindBy که قرار است چک کند این نوع واحد سازمانی برای تعریف واحد سازمانی استفاده شده است، در Return به آن یک لیست خالی اختصاص میدهیم تا نشان دهیم لیست خالی برگشته است. عملیات Act را وارد Try میکنیم تا اگر به هر دلیل انجام نشد، Assert ما باشد. دو حالت رخداد استثناء که در متد اصلی تست شده است در دو متد تست به طور جداگانه تست گردیده است:

```

/// <summary>
/// کامند حذف یک نوع واحد سازمانی باید پیش از حذف بررسی کند که این شناسه داده شده برای حذف
/// موجود باشد.
/// </summary>
[TestMethod]
[ExpectedException(typeof(DeleteEntityNotFoundException))]
public void DeleteUnitTypeCommand_ShouldNot_Delete_When_UnitTypeId_NotExist()
{
    //Arrange
    var unitTypeId = new Guid();
    var deleteUnitTypeCommand = new DeleteUnitTypeCommand();
    var unitType = new UnitType("خوشه");
    var org = new List<OrganizationUnit>();
    _mockUnitTypeRepository.Setup(d => d.FindBy(unitTypeId)).Returns(unitType);
    _mockUnitTypeRepository.Setup(x => x.Remove(unitType));
    _mockOrganizationUnitRepository.Setup(z => z.FindBy(unitType)).Returns(org);

    //Act
    _organizationalService.Handle(deleteUnitTypeCommand);
}

/// <summary>
/// کامند حذف یک نوع واحد سازمانی نباید اجرا شود وقتی که نوع واحد برای تعریف واحدهای سازمان
/// استفاده شده است.
/// </summary>
[TestMethod]
[ExpectedException(typeof(UnitTypeIsUnderUsingException))]
public void
DeleteUnitTypeCommand_ShouldNot_Delete_When_UnitType_Exist_but_UsedForDefineOrganizationUnit()
{
    //Arrange
    var unitTypeId = new Guid();
    var deleteUnitTypeCommand = new DeleteUnitTypeCommand { UnitTypeId = unitTypeId };
    var unitType = new UnitType("خوشه");
    var org = new List<OrganizationUnit>()
    {
        new OrganizationUnit("مدیریت یک", unitType, null),
        new OrganizationUnit("مدیریت دو", unitType, null)
    };
    _mockUnitTypeRepository.Setup(d =>
d.FindBy(deleteUnitTypeCommand.UnitTypeId)).Returns(unitType);

```

```
_mockUnitTypeRepository.Setup(x => x.Remove(unitType));
_mockOrganizationUnitRepository.Setup(z => z.FindBy(unitType)).Returns(org);

//Act
_organizationalService.Handle(deleteUnitTypeCommand);
}
```

متد `DeleteUnitTypeCommand_ShouldNot_Delete_When_UnitTypeId_NotExist` همانطور که از نامش معلوم است بررسی میکند که نوع واحد سازمانی که ID آن برای حذف ارسال میشود در Database وجود دارد و اگر نباشد Exception مطلوب ما باید داده شود.

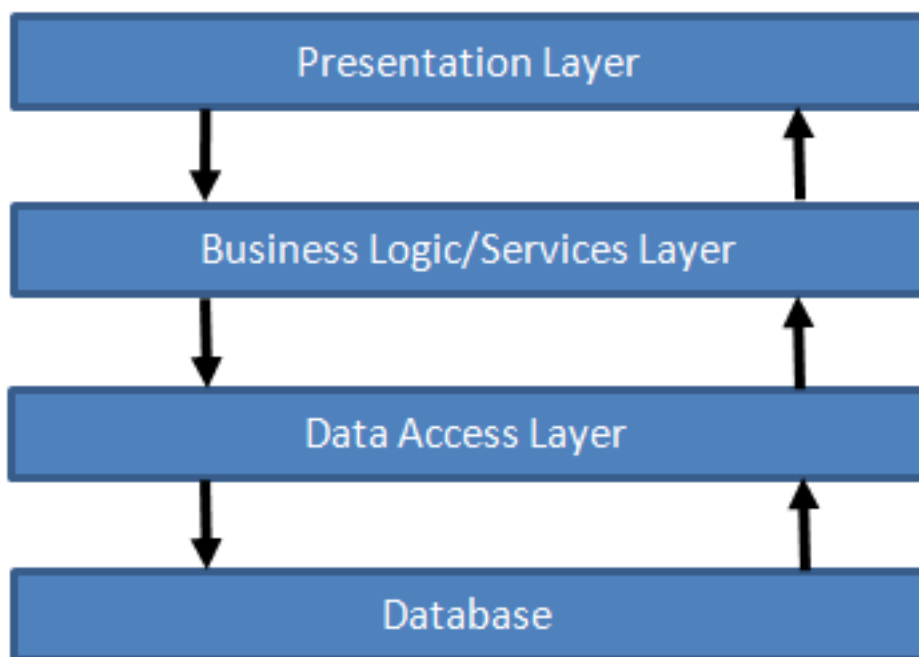
در متد `DeleteUnitTypeCommand_ShouldNot_Delete_When_UnitType_Exist_but_UsedForDefineOrganizationUnit` بررسی میشود که از این نوع واحد سازمانی برای تعریف واحد سازمانی استفاده شده است یا نه و صحت این مورد با الگوی Specification صورت گرفته است. استثنای مطلوب ما Assert و شرط درستی این متد تست، میباشد.

مقدمه

نوشتن تست برای کدها بسیار عالی است، در صورتیکه بدانید چگونه این کار را بدرستی انجام دهید. متأسفانه بسیاری از منابع آموزشی موجود، این مطلب که چگونه کد قابل تست بنویسیم را رها می‌کنند؛ بدلیل اینکه آنها مراقبت در بین لایه‌هایی که در کدهای واقعی وجود دارند گیر نکنند، جایی که شما لایه‌های خدمات (Service Layer)، لایه‌های داده، و غیره را دارید. به ضرورت، وقتی میخواهید کدی را تست کنید که این وابستگی‌ها را دارد، تست‌ها بسیار کند و برای نوشتن دشوار هستند و اغلب بدلیل وابستگی‌ها شکست می‌خورند و نتیجه غیر قابل انتظاری خواهند داشت.

پیش زمینه

کدی که به خوبی نوشته شده باشد از لایه‌های جداگانه‌ای تشکیل شده است که هر لایه مسئول یک قسمت متفاوت از وظایف برنامه خواهد بود. لایه‌های واقعی بر اساس نیاز و نظر توسعه دهندگان متفاوت است، ولی یک ساختار رایج به شکل زیر خواهد بود.



لایه نمایش / رابط کاربری : این قسمت کد منطق نمایش و تعامل رابط کاربری می‌باشد. **منطق تجاری / لایه خدمات :** این قسمت منطق تجاری کد شما میباشد. برای نمونه در کد مربوط به یک کارت خرید، این کارت خرید میداند چگونه جمع کارت را محاسبه نماید و یا چگونه اقلام موجود در سفارش را شمارش کند. **لایه دستیابی به داده / لایه ماندگاری :** این کد میداند چگونه به منبع داده متصل شود و یک کارت خرید را بازگرداند و یا چگونه یک کارت را در منبع داده ذخیره نماید. **منبع داده :** اینجا جایی است که محتویات کارت در آن ذخیره میشود. **مزیت مدیریت وابستگی‌ها**

بدون مدیریت وابستگی‌ها، وقتی شما برای لایه نمایش تست می‌نویسید، کد شما در خدمات واقعی که به کد واقعی دستیابی به منبع داده وابسته است، گرفتار می‌شود و سپس به منبع داده اصلی متصل می‌شود. در واقع، وقتی شما در حال تست نویسی برای

گزینه "اضافه به کارت" و یا "دریافت تعداد اقلام" هستید، می‌خواهید کد را به صورت مجزا تست کنید و قادر باشید نتایج قابل پیش بینی را تضمین نمایید. بدون مدیریت وابستگی‌ها، تست‌های نمایش شما برای گزینه "اضافه به کارت" کند هستند و وابستگی‌ها نتایج غیر قابل پیش بینی بازمیگردانند که میتوانند باعث پاس نشدن تست شما شوند.

راه حل تزریق وابستگی است

راه حل این مسأله تزریق وابستگی است. تزریق وابستگی برای کسانی که تا بحال از آن استفاده نکرده اند، اغلب گیج کننده و پیچیده به نظر میرسد، اما در واقع، مفهومی بسیار ساده و فرآیندی با چند اصل ساده است. آنچه می‌خواهیم انجام دهیم مرکزیت دادن به وابستگی هاست. در این مورد، استفاده از شیء کارت خرید است و سپس رابطه بین کدها را کم می‌کنیم تا جاییکه وقتی شما برنامه را اجرا می‌کنید، از خدمات و منابع واقعی استفاده کند و وقتی آنرا تست می‌کنید، می‌توانید از خدمات جعلی (mocking) استفاده نمایید که سریع و قابل پیش بینی هستند. توجه داشته باشید که رویکردهای متفاوت بسیاری وجود دارند که می‌توانید استفاده کنید، ولی من برای ساده نگهداشتن این مطلب، فقط رویکرد Constructor Injection را شرح می‌دهم.

گام 1 - وابستگی‌ها را شناسایی کنید

وابستگی‌ها وقتی اتفاق می‌افتند که کد شما از لایه‌های دیگر استفاده می‌نماید. برای نمونه، وقتی لایه نمایش از لایه خدمات استفاده می‌نماید. کد نمایش شما به لایه خدمات وابسته است، ولی ما می‌خواهیم کد لایه نمایش را به صورت مجزا تست کنیم.

```
public class ShoppingCartController : Controller
{
    public ActionResult GetCart()
    {
        //shopping cart service as a concrete dependency
        ShoppingCartService shoppingCartService = new ShoppingCartService();
        ShoppingCart cart = shoppingCartService.GetContents();
        return View("Cart", cart);
    }
    public ActionResult AddItemToCart(int itemId, int quantity)
    {
        //shopping cart service as a concrete dependency
        ShoppingCartService shoppingCartService = new ShoppingCartService();
        ShoppingCart cart = shoppingCartService.AddItemToCart(itemId, quantity);
        return View("Cart", cart);
    }
}
```

گام 2 - وابستگی‌ها را مرکزیت دهید

این کار با چندین روش قابل انجام است؛ در این مثال من می‌خواهم یک متغیر عضو از نوع ShoppingCartService ایجاد کنم و سپس آنرا به وهله ای که در Constructor ایجاد خواهم کرد، منتسب کنم. حال هر جا ShoppingCartService نیاز باشد بجای آنکه یک وهله جدید ایجاد کنم، از این وهله استفاده می‌نمایم.

```
public class ShoppingCartController : Controller
{
    private ShoppingCartService _shoppingCartService;
    public ShoppingCartController()
    {
        _shoppingCartService = new ShoppingCartService();
    }
    public ActionResult GetCart()
    {
        //now using the shared instance of the shoppingCartService dependency
        ShoppingCart cart = _shoppingCartService.GetContents();
        return View("Cart", cart);
    }
    public ActionResult AddItemToCart(int itemId, int quantity)
    {
        //now using the shared instance of the shoppingCartService dependency
        ShoppingCart cart = _shoppingCartService.AddItemToCart(itemId, quantity);
        return View("Cart", cart);
    }
}
```

گام 3 - از بین بردن ارتباط لایه‌ها (Loose Coupling) بجای استفاده از اشیاء واقعی ، براساس interface ها برنامه نویسی کنید.

اگر شما کد خود را با استفاده از **IShoppingCartService** به عنوان یک interface بجای استفاده از شیء واقعی **ShoppingCartService** نوشته باشید، زمانیکه تست را مینویسید، میتوانید یک سرویس کارت خرید جعلی (mocking) که **IShoppingCartService** را پیاده سازی کرده جایگزین شیء اصلی نمایید. در کد زیر، توجه کنید تنها تغییر این است که متغیر عضو اکنون از نوع **IShoppingCartService** است بجای **ShoppingCartService**.

```
public interface IShoppingCartService
{
    ShoppingCart GetContents();
    ShoppingCart AddItemToCart(int itemId, int quantity);
}
public class ShoppingCartService : IShoppingCartService
{
    public ShoppingCart GetContents()
    {
        throw new NotImplementedException("Get cart from Persistence Layer");
    }
    public ShoppingCart AddItemToCart(int itemId, int quantity)
    {
        throw new NotImplementedException("Add Item to cart then return updated cart");
    }
}
public class ShoppingCart
{
    public List<product> Items { get; set; }
}
public class Product
{
    public int ItemId { get; set; }
    public string ItemName { get; set; }
}
public class ShoppingCartController : Controller
{
    //Concrete object below points to actual service
    //private ShoppingCartService _shoppingCartService;
    //loosely coupled code below uses the interface rather than the
    //concrete object
    private IShoppingCartService _shoppingCartService;
    public ShoppingCartController()
    {
        _shoppingCartService = new ShoppingCartService();
    }
    public ActionResult GetCart()
    {
        //now using the shared instance of the shoppingCartService dependency
        ShoppingCart cart = _shoppingCartService.GetContents();
        return View("Cart", cart);
    }
    public ActionResult AddItemToCart(int itemId, int quantity)
    {
        //now using the shared instance of the shoppingCartService dependency
        ShoppingCart cart = _shoppingCartService.AddItemToCart(itemId, quantity);
        return View("Cart", cart);
    }
}
```

گام 4 - وابستگی‌ها را تزریق کنید

اکنون ما تمام وابستگی‌ها را در یک جا مرکزیت داده‌ایم و کد ما رابطه کمی با آن وابستگی‌ها دارد. همانند گذشته، چندین راه برای پیاده سازی این گام وجود دارد. بدون استفاده از ابزارهای کمکی برای این مفهوم، ساده‌ترین راه دوباره نویسی (Overload) متد ایجاد کننده است:

```
//loosely coupled code below uses the interface rather
//than the concrete object
private IShoppingCartService _shoppingCartService;
```

```
//MVC uses this constructor
public ShoppingCartController()
{
    _shoppingCartService = new ShoppingCartService();
}
//You can use this constructor when testing to inject the
//ShoppingCartService dependency
public ShoppingCartController(IShoppingCartService shoppingCartService)
{
    _shoppingCartService = shoppingCartService;
}
```

گام 5 - تست را با استفاده از یک شیء جعلی (Mocking) انجام دهید

مثالی از یک سناریوی تست ممکن در زیر آمده است. توجه کنید که یک شیء جعلی از نوع کلاس `ShoppingCartService` ساخته ایم. این شیء جعلی فرستاده می شود به شیء `Controller` و متد `GetContents` پیاده سازی میشود تا بجای آنکه کد اصلی را که به منبع داده مراجعه می کند اجرا نماید، داده های جعلی و شبیه سازی شده را برگرداند. بدلیل آنکه تمام کد را نوشته ایم، بسیار سریعتر از اجرای کوئری بر روی دیتابیس اجرا خواهد شد و دیگر نگرانی بابت تهیه داده تستی و یا تصحیح داده بعد از اتمام تست (بازگرداندن داده به حالت قبل از تست) نخواهیم داشت. توجه داشته باشید که بدلیل مرکزیت دادن به وابستگی ها در گام 2، تنها باید یکبار آنرا تزریق نماییم و بخاطر کاری که در گام 3 انجام شد، وابستگی ما به حدی پایین آمده که میتوانیم هر شیء ایی را (جعلی و یا حقیقی) ارسال کنیم و فقط کافیسست شیء مورد نظر `IShoppingCartService` را پیاده سازی کرده باشد.

```
[TestClass]
public class ShoppingCartControllerTests
{
    [TestMethod]
    public void GetCartSmokeTest()
    {
        //arrange
        ShoppingCartController controller =
            new ShoppingCartController(new ShoppingCartServiceStub());
        // Act
        ActionResult result = controller.GetCart();
        // Assert
        Assert.IsInstanceOfType(result, typeof(ViewResult));
    }
}
/// <summary>
/// This is is a stub of the ShoppingCartService
/// </summary>
public class ShoppingCartServiceStub : IShoppingCartService
{
    public ShoppingCart GetContents()
    {
        return new ShoppingCart
        {
            Items = new List<product> {
                new Product {ItemId = 1, ItemName = "Widget"}
            };
        };
    }
    public ShoppingCart AddItemToCart(int itemId, int quantity)
    {
        throw new NotImplementedException();
    }
}
```

مطالب تکمیلی از یک ابزار کنترل وابستگی (IoC/DI) استفاده کنید:

از رایج ترین و عمومی ترین ابزارهای کنترل وابستگی برای .Net می توان به StructureMap و CastleWindsor اشاره کرد. در کد نویسی واقعی، شما وابستگی های بسیاری خواهید داشت، که این وابستگی ها هم وابستگی هایی دارند که به سرعت از مدیریت شما خارج خواهند شد. راه حل این مشکل استفاده از یک ابزار کنترل وابستگی خواهد بود. از یک چارچوب تجزیه (Isolation Framework) استفاده نمایید:

برای ایجاد اشیاء جعلی ممکن است کار زیادی لازم باشد و استفاده از یک Isolation Framework میتواند زمان و میزان کد نویسی شما را کم کند. از رایج ترین این ابزارها میتوان Rhino Mocks و Moq را نام برد.