

**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 ) );

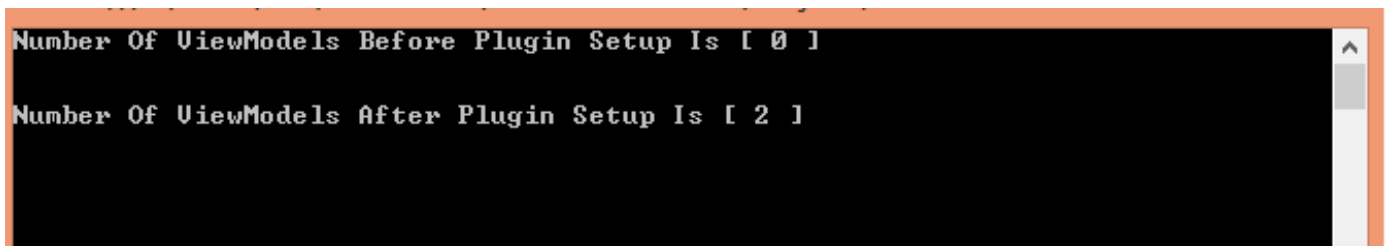
    Console.WriteLine( Environment.NewLine );

    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.



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

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

## نظرات خوانندگان

نویسنده: MehRad

تاریخ: ۱۱:۴۷ ۱۳۹۱/۱۱/۲۵

با تشکر از مطلب خوبتون

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

نویسنده: مسعود م. پاکدل

تاریخ: ۱۳:۲۳ ۱۳۹۱/۱۱/۲۵

ممنون.

بله حتما در پست‌های بعدی در مورد MEF و استفاده اون در WAF (WPF Application Framework) و MVC و

Prism توضیحاتی رو خواهد داد.

نویسنده: علیرضا پایدار

تاریخ: ۱۳:۸ ۱۳۹۲/۰۶/۲۶

ممنون مفید بود.

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

نویسنده: مسعود پاکدل

تاریخ: ۱۳:۴۷ ۱۳۹۲/۰۶/۲۶

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

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

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

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

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

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