

یکی از راهکارهای پیاده سازی IOC یا همان Inversion Of Control در پروژه های MVC استفاده از [Unity](#) و معرفی آن به DependencyResolver خود دات نت است.

برای آشنایی با Unity و قابلیت های آن میتوانید به [اینجا](#) و [اینجا](#) سر بزنید.

اما برای استفاده از Unity در پروژه های MVC کافی است در Global یا فایل راه انداز (bootstrapper) تک تک انتزاعها (Interface) را به کلاس های 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:

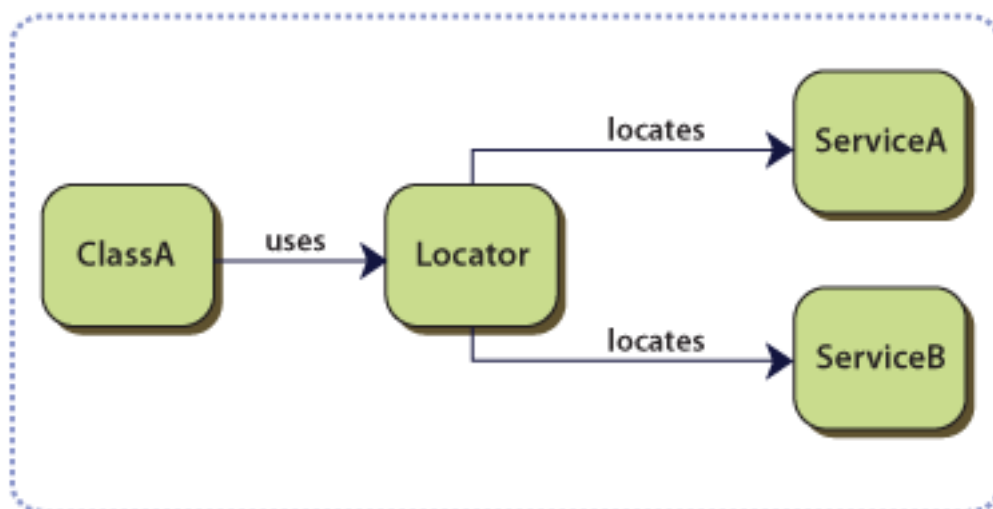
```
DependencyResolver.SetResolver(new UnityDependencyResolver(container));
```

تا به اینجا به راحتی میتوانید از سرویس های معرفی شده در پروژه MVC استفاده کنید.

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

اما اگر بخواهیم از کلاس های معرفی شده در Unity در لایه های 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 سرویس ها، 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.