عنوان: **خودکارسازی فرآیند نگاشت اشیاء در AutoMapper**
نویسنده: سیروان عفیفی
تاریخ: ۱۳۹۴/۰۶/۰۱ ۱۷:۲۵
آدرس: www.dotnettips.info
گروه‌ها: Refactoring, AutoMapper

قرار دادن تمامی تنظیمات نگاشت‌ها درون کلاس‌های پروفایل تا حدودی حجم کدهای ما را در آینده زیاد خواهد کرد.

```
public class TestProfile1 : Profile
{
    protected override void Configure()
    {
        // این تنظیم سراسری هست و به تمام خواص زمانی اعمال می‌شود
        this.CreateMap<DateTime, string>().ConvertUsing(new DateTimeToPersianDateTimeConverter());
        this.CreateMap<User, UserViewModel>();
       // Other mappings
     }

    public override string ProfileName
    {
        get { return this.GetType().Name; }
    }
}
```

در ادامه می‌خواهیم به روشی جهت سازماندهی بهتر این نوع کلاس‌ها بپردازیم. به طوری‌که تعاریف مربوط به نگاشت‌ها در کنار View Modelهای برنامه قرار گیرند. برای اینکار ابتدا اینترفیس‌های زیر را ایجاد خواهیم کرد:

```
public interface IMapFrom<T>
{

}
public interface IHaveCustomMappings
{
     void CreateMappings(IConfiguration configuration);
}
```

خوب، همانطور که مشاهده می‌کنید، در اینترفیس IMapFrom امضای هیچ متدی تعریف نشده است. در واقع View Modelهای ما از این اینترفیس جهت تشخیص اینکه به چه مدلی قرار است نگاشت شوند، استفاده خواهند کرد. اما در حالتی‌که نیاز به نگاشت صریح پراپرتی‌های یک View Model داشتیم می‌توانیم اینترفیس IHaveCustomMappings را پیاده‌سازی کرده و جزئیات نگاشت را درون متد CreateMappings تعیین کنیم.
به عنوان مثال View Model زیر را در نظر بگیرید:

```
public class PersonViewModel : IMapFrom<Person>
{
      public string Name { get; set; }
      public string LastName { get; set; }
}
```

خوب، در اینجا با پیاده‌سازی اینترفیس IMapFrom نوع مبدا را برای ویومدل فوق مشخص کرده‌ایم. در این‌حالت هدف ما نگاشت تمامی خواص کلاس Person به تمامی خواص کلاس PersonViewModel خواهد بود. برای حالت‌های خاص نیز که نیاز به نگاشت دقیق خواص باشد به اینصورت عمل خواهیم کرد:

```
public class PersonViewModel : IHaveCustomMapping
{
     public string Name { get; set; }
     // دیگر پراپرتی‌ها

     public void CreateMappings(IConfiguration configuration)
     {
           configuration.CreateMap<ApplicationUser, PersonViewModel>()
                .ForMember(m => m.Name, opt =>
                     opt.MapFrom(u => u.ApplicationUser.UserName));
           // دیگر نگاشت‌ها
     }
}
```

خوب، در نهایت با استفاده از امکانات LINQ و Reflection کار پردازش تنظیمات نگاشت‌های هر View Model و خودکارسازی فرآیند نگاشت را انجام خواهیم داد. اینکار را می‌توانیم درون یک کلاس با نام AutoMapperConfig و با پیاده‌سازی اینترفیس IRunInit انجام دهیم:

```
public void Execute()
{
      var types = Assembly.GetExecutingAssembly().GetExportedTypes();

      LoadStandardMappings(types);

      LoadCustomMappings(types);
}
```

در داخل متد Execute دو متد به نام‌های LoadStandardMappings و LoadCustomMapping را فراخوانی کرده‌ایم. متد اول برای پردازش حالتی است که اینترفیس IMapFrom را پیاده‌سازی کرده باشیم و متد دوم نیز برای حالتی است که اینترفیس IHaveCustomMappings را پیاده‌سازی کرده باشیم.

متد LoadStandardMappings :

```
private static void LoadStandardMappings(IEnumerable <Type> types)
{
     var maps = (from t in types
                       from i in t.GetInterfaces()
                       where i.IsGenericType && i.GetGenericTypeDefinition() == typeof(IMapFrom< >)  &&
!t.IsAbstract && !t.IsInterface
                       select new {
                                Source = i.GetGenericArguments()[0],
                                Destination = t
                       }).ToArray();

      foreach(var map in maps)
      {
               Mapper.CreateMap(map.Source, map.Destination);
      }
}
```

توضیح کدهای فوق :

ابتدا تمامی typeهای تعریف شده در پروژه به متد فوق پاس داده خواهند شد.

برای هر type تمامی اینترفیس‌هایی که توسط این type پیاده‌سازی شده باشند را دریافت خواهیم کرد.

سپس هر type که اینترفیس IMapFrom را پیاده‌سازی کرده باشد را پردازش می‌کنیم.

سپس از نوع‌های Abstract و Interface صرفنظر خواهیم کرد.

انواع مبدا و مقصد را برای AutoMapper فراهم خواهیم کرد.

در نهایت AutoMapper براساس آنها نگاشت را ایجاد خواهد کرد.

متد LoadCustomMapping:

```
private static void LoadCustomMappings(IEnumerable <Type> types)
{
     var maps = (from t in types
                       from i in t.GetInterfaces()
                       where typeof(IHaveCustomMappings).IsAssignableFrom(t) && !t.IsAbstract &&
!t.IsInterface
                       select(IHaveCustomMappings) Activator.CreateInstance(t)).ToArray();

     foreach(var map in maps)
     {
               map.CreateMappings(Mapper.Configuration);
```

```
    }
}
```

توضیح کدهای فوق:

این متد نیز همانند متد قبلی، تمامی typeها را پردازش خواهد کرد. با این تفاوت که مواردی که اینترفیس IHaveCustomMappings را پیاده‌سازی کرده باشند، دریافت کرده و در نهایت متد CreateMappings آنها را فراخوانی خواهیم کرد. اکنون کدهای نگاشت برنامه از اصول Open and Closed پیروی می‌کنند. در نتیجه می‌توانیم نگاشت‌های جدید را به سادگی و با ایجاد View Model ها تعریف کنیم.