

پیشنیازها

- مطالعه‌ی [مطالب گروه AutoMapper](#) در سایت، دید خوبی را برای شروع به کار با آن فراهم می‌کنند و در اینجا قصد تکرار این مباحث پایه‌ای را نخواهیم داشت. هدف بیشتر بررسی یک سری نکات پیشرفته‌تر و عمیق‌تر است از کار با AutoMapper.

- [آشنایی با Lazy loading و Eager loading در حین کار با EF](#)

ساختار و پیشنیازهای برنامه‌ی مطلب جاری

جهت سهولت پیگیری مطلب و تمرکز بیشتر بر روی مفاهیم اصلی مورد بحث، یک برنامه‌ی کنسول را آغاز کرده و سپس بسته‌های نیوگت ذیل را به آن اضافه کنید:

```
PM> install-package AutoMapper
PM> install-package EntityFramework
```

به این ترتیب بسته‌های AutoMapper و EF به پروژه‌ی جاری اضافه خواهند شد.

آشنایی با ساختار مدل‌های برنامه

در اینجا ساختار جداول مطالب یک بلاگ را به همراه نویسندگان آن‌ها، مشاهده می‌کنید:

```
public class BlogPost
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    [ForeignKey("UserId")]
    public virtual User User { get; set; }
    public int UserId { get; set; }
}

public class User
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }

    public virtual ICollection<BlogPost> BlogPosts { get; set; }
}
```

هر کاربر می‌تواند تعدادی مطلب تهیه کند و هر مطلب توسط یک کاربر نوشته شده‌است.

هدف از این مثال

فرض کنید اطلاعاتی که قرار است به کاربر نمایش داده شوند، توسط ViewModel ذیل تهیه می‌شود:

```
public class UserViewModel
{
    public int Id { set; get; }
    public string Name { set; get; }

    public ICollection<BlogPost> BlogPosts { get; set; }
}
```

در اینجا می‌خواهیم اولین کاربر ثبت شده را یافته و سپس لیست مطالب آن را نمایش دهیم. همچنین می‌خواهیم این کوئری تهیه شده به صورت خودکار اطلاعاتش را بر اساس ساختار ViewModel ایی که مشخص کردیم (و این ViewModel الزاما تمام عناصر آن با عناصر مدل اصلی یکی نیست)، بازگشت دهیم.

تهیه نگاشت‌های AutoMapper

برای مدیریت بهتر نگاشت‌های AutoMapper توصیه شده‌است که کلاس‌های Profile ایی را به شکل ذیل تهیه کنیم:

```
public class TestProfile : Profile
{
    protected override void Configure()
    {
        this.CreateMap<User, UserViewModel>();
    }

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

کار با ارث بری از کلاس پایه Profile کتابخانه‌ی AutoMapper شروع می‌شود. سپس باید متد Configure آن را بازنویسی کنیم. در اینجا می‌توان با استفاده از متدی مانند CreateMap مشخص کنیم که قرار است اطلاعاتی با ساختار شیء User، به اطلاعاتی با ساختار از نوع شیء UserViewModel به صورت خودکار نگاشت شوند.

ثبت و معرفی پروفایل‌های AutoMapper

پس از تهیه‌ی پروفایل مورد نیاز، در ابتدای برنامه با استفاده از متد Mapper.Initialize، کار ثبت این تنظیمات صورت خواهد گرفت:

```
Mapper.Initialize(cfg => // In Application_Start()
{
    cfg.AddProfile<TestProfile>();
});
```

روش متداول کار با AutoMapper جهت نگاشت اطلاعات User به ViewModel آن

در ادامه به نحو متداولی، ابتدا اولین کاربر ثبت شده را یافته و سپس با استفاده از متد Mapper.Map اطلاعات این شیء user به ViewModel آن نگاشت می‌شود:

```
using (var context = new MyContext())
{
    var user1 = context.Users.FirstOrDefault();
    if (user1 != null)
    {
        var uiUser = new UserViewModel();
        Mapper.Map(source: user1, destination: uiUser);

        Console.WriteLine(uiUser.Name);
        foreach (var post in uiUser.BlogPosts)
        {
            Console.WriteLine(post.Title);
        }
    }
}
```

تا اینجا اگر برنامه را اجرا کنید، مشکلی را مشاهده نخواهید کرد، اما این کدها سبب اجرای حداقل دو کوئری خواهند شد: الف) یافتن اولین کاربر

ب) واکنشی لیست مطالب او در یک کوئری دیگر

کاهش تعداد رفت و برگشت‌ها به سرور با استفاده از متدهای ویژه‌ی AutoMapper

در حالت متداول کار با EF، با استفاده از متد Include می‌توان این Lazy loading را لغو کرد و در همان اولین کوئری، مطالب کاربر یافت شده را نیز دریافت نمود:

```
var user1 = context.Users.Include(user => user.BlogPosts).FirstOrDefault();
```

و سپس این اطلاعات را توسط AutoMapper نگاشت کرد.

در این حالت، AutoMapper برای ساده سازی این مراحل، متدهای Project To را معرفی کرده‌است:

```
var uiUser = context.Users.Project().To<UserViewModel>().FirstOrDefault();
```

در اینجا نیز Lazy loading لغو شده و به صورت خودکار جویی به جدول مطالب کاربران ایجاد خواهد شد. بنابراین با استفاده از متدهای Project To می‌توان از ذکر Include های EF صرف‌نظر کرد و همچنین دیگر نیازی به نوشتن متد Select جهت نگاشت دستی خواص مورد نظر به خواص ViewModel نیست.

کدهای کامل این قسمت را از اینجا می‌توانید دریافت کنید:

[AM_Sample01.zip](#)

نظرات خوانندگان

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۲/۰۹ ۱۱:۴۶

یک نکته‌ی تکمیلی

فراخوانی متدهای متداول EF مانند ToList و FirstOrDefault و امثال آن، اگر به همراه Select نباشند، سبب واکنشی تمام فیلدها و خواص جدول مورد نظر می‌شوند. اما اگر از متد Project To مانند مطلب فوق استفاده کنید، واکنشی انجام شده به صورت خودکار تنها بر اساس خواص موجود در ViewModel صورت می‌گیرد و به این ترتیب حجم کمتری از اطلاعات رد و بدل خواهد شد (چون AutoMapper کار نوشتن Select را بر اساس خواص ViewModel، در پشت صحنه انجام داده‌است و این Select حاوی تمام خواص کلاس جدول مورد استفاده نیست).

فرض کنید مدل معادل با جدول بانک اطلاعاتی ما چنین ساختاری را دارد:

```
public class User
{
    public int Id { set; get; }
    public string Name { set; get; }
    public DateTime RegistrationDate { set; get; }
}
```

و ViewModel ایی که قرار است به کاربر نمایش داده شود این ساختار را دارد:

```
public class UserViewModel
{
    public int Id { set; get; }
    public string Name { set; get; }
    public string RegistrationDate { set; get; }
}
```

در اینجا می‌خواهیم حین تبدیل User به UserViewModel، تاریخ میلادی به صورت خودکار، تبدیل به یک رشته‌ی شمسی شود. برای مدیریت یک چنین سناریوهایی توسط AutoMapper، امکان نوشتن تبدیلگرهای سفارشی نیز پیش بینی شده‌است.

تبدیلگر سفارشی تاریخ میلادی به شمسی مخصوص AutoMapper

در ذیل یک تبدیلگر سفارشی مخصوص AutoMapper را با پیاده سازی اینترفیس ITypeConverter آن ملاحظه می‌کنید:

```
public class DateTimeToPersianDateTimeConverter : ITypeConverter<DateTime, string>
{
    private readonly string _separator;
    private readonly bool _includeHourMinute;

    public DateTimeToPersianDateTimeConverter(string separator = "/", bool includeHourMinute = true)
    {
        _separator = separator;
        _includeHourMinute = includeHourMinute;
    }

    public string Convert(ResolutionContext context)
    {
        var objDateTime = context.SourceValue;
        return objDateTime == null ? string.Empty : toShamsiDateTime((DateTime)context.SourceValue);
    }

    private string toShamsiDateTime(DateTime info)
    {
        var year = info.Year;
        var month = info.Month;
        var day = info.Day;
        var persianCalendar = new PersianCalendar();
        var pYear = persianCalendar.GetYear(new DateTime(year, month, day, new GregorianCalendar()));
        var pMonth = persianCalendar.GetMonth(new DateTime(year, month, day, new GregorianCalendar()));
        var pDay = persianCalendar.GetDayOfMonth(new DateTime(year, month, day, new GregorianCalendar()));
        return _includeHourMinute ?
            string.Format("{0}{1}{2}{1}{3} {4}:{5}", pYear, _separator, pMonth.ToString("00",
                CultureInfo.InvariantCulture), pDay.ToString("00", CultureInfo.InvariantCulture),
                info.Hour.ToString("00"), info.Minute.ToString("00"))
            : string.Format("{0}{1}{2}{1}{3}", pYear, _separator, pMonth.ToString("00",
                CultureInfo.InvariantCulture), pDay.ToString("00", CultureInfo.InvariantCulture));
    }
}
```

ITypeConverter دو پارامتر جنریک را قبول می‌کند. پارامتر اول نوع ورودی و پارامتر دوم، نوع خروجی مورد انتظار است. در اینجا باید خروجی متد Convert را بر اساس آرگومان دوم ITypeConverter مشخص کرد. توسط ResolutionContext می‌توان به context.SourceValue که معادل DateTime دریافتی است، دسترسی یافت. سپس این DateTime را بر اساس متد toShamsiDateTime تبدیل کرده و بازگشت می‌دهیم.

ثبت و معرفی تبدیلگرهای سفارشی AutoMapper

پس از تعریف یک تبدیلگر سفارشی AutoMapper، اکنون نیاز است آن را به AutoMapper معرفی کنیم:

```
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>();
    }

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

جهت مدیریت بهتر نگاشت‌های AutoMapper ابتدا یک کلاس Profile را آغاز خواهیم کرد و سپس توسط متدهای CreateMap، کار معرفی نگاشت‌ها را آغاز می‌کنیم.

همانطور که مشاهده می‌کنید در اینجا دو نگاشت تعریف شده‌اند. یکی برای تبدیل User به UserViewModel و دیگری، معرفی نحوه‌ی نگاشت DateTime به string، توسط تبدیلگر سفارشی DateTimeToPersianDateTimeConverter است که به کمک متد الحاقی ConvertUsing صورت گرفته‌است. باید دقت داشت که تنظیمات تبدیلگرهای سفارشی سراسری هستند و در کل برنامه و به تمام پروفایل‌ها اعمال می‌شوند.

بررسی خروجی تبدیلگر سفارشی تاریخ

اکنون کار استفاده از تنظیمات AutoMapper با ثبت پروفایل تعریف شده آغاز می‌شود:

```
Mapper.Initialize(cfg => // In Application_Start()
{
    cfg.AddProfile<TestProfile1>();
});
```

سپس نحوه‌ی استفاده از متد Mapper.Map همانند قبل خواهد بود:

```
var dbUser1 = new User
{
    Id = 1,
    Name = "Test",
    RegistrationDate = DateTime.Now.AddDays(-10)
};

var uiUser = new UserViewModel();

Mapper.Map(source: dbUser1, destination: uiUser);
```

در اینجا در حین کار تبدیل و نگاشت dbUser به uiUser، زمانیکه AutoMapper به هر خاصیت DateTime ایی می‌رسد، مقدار آن را با توجه به تبدیلگر سفارشی تاریخی که به آن معرفی کردیم، تبدیل به معادل رشته‌ای شمسی می‌کند.

نوشتن تبدیلگرهای غیر سراسری

همانطور که عنوان شد، معرفی تبدیلگرها به AutoMapper سراسری است و در کل برنامه اعمال می‌شود. اگر نیاز است فقط برای یک مدل خاص و یک خاصیت خاص آن تبدیلگر نوشته شود، باید نگاشت مورد نظر را به صورت ذیل تعریف کرد:

```
this.CreateMap<User, UserViewModel>()
    .ForMember(userViewModel => userViewModel.RegistrationDate,
        opt => opt.ResolveUsing(src =>
        {
            var dt = src.RegistrationDate;
            return dt.ToShortDateString();
        }));
```

اینبار در همان کلاس پروفایل ابتدای بحث، نگاشت User به ViewModel آن با کمک متد ForMember، سفارشی سازی شده‌است. در اینجا عنوان شده‌است که اگر به خاصیت ویژه‌ی RegistrationDate رسیدی، مقدار آن را با توجه به فرمولی که مشخص شده، محاسبه کرده و بازگشت بده. این تنظیم خصوصی است و به کل برنامه اعمال نمی‌شود.

خصوصی سازی تبدیلگرها با تدارک موتورهای نگاشت اختصاصی

اگر می‌خواهید تنظیمات TestProfile1 به کل برنامه اعمال نشود، نیاز است یک MappingEngine جدید و مجزای از MappingEngine سراسری AutoMapper را ایجاد کرد:

```
var configurationStore = new ConfigurationStore(new TypeMapFactory(), MapperRegistry.Mappers);
configurationStore.AddProfile<TestProfile1>();
var mapper = new MappingEngine(configurationStore);
mapper.Map(source: dbUser1, destination: uiUser);
```

به صورت پیش فرض و در پشت صحنه، متد Mapper.Map از یک MappingEngine سراسری استفاده می‌کند. اما می‌توان در یک برنامه چندین MappingEngine مجزا داشت که نمونه‌ای از آن را در اینجا مشاهده می‌کنید.

کدهای کامل این قسمت را از اینجا می‌توانید دریافت کنید:

[AM_Sample02.zip](#)

اگر مطلب « [Refactoring به تزریق وابستگی‌ها](#) » را به خاطر داشته باشید، جهت تشخیص وابستگی‌های یک کلاس، کار از بررسی کلمات new و همچنین فراخوانی‌های استاتیک، شروع می‌شود و ... متد استاتیک Mapper.Map کتابخانه‌ی AutoMapper نیز از همین دست است. در ادامه قصد داریم بجای فراخوانی مستقیم Mapper.Map از اینترفیس IMapperEngine به عنوان تامین کننده‌ی متد Map استفاده کنیم. همچنین کلاس‌های Profile نوشته شده را نیز به صورت خودکار به برنامه اضافه نمائیم.

تنظیمات IoC Container مختص به AutoMapper

در ذیل یک کلاس Registry مخصوص StructureMap را مشاهده می‌کنید که جهت کپسوله کردن اطلاعات خاص AutoMapper تهیه شده است. می‌توان این اطلاعات را در داخل تنظیمات new Container خود قرار داد و یا می‌توان آن‌ها را جهت شلوغ نشدن سایر تنظیمات IoC Container، به یک کلاس Registry منتقل کرد:

```
public class AutoMapperRegistry : Registry
{
    public AutoMapperRegistry()
    {
        var platformSpecificRegistry = PlatformAdapter.Resolve<IPlatformSpecificMapperRegistry>();
        platformSpecificRegistry.Initialize();

        For<ConfigurationStore>().Singleton().Use<ConfigurationStore>()
            .Ctor<IEnumerable<IObjectMapper>>().Is(MapperRegistry.Mappers);

        For<IConfigurationProvider>().Use(ctx => ctx.GetInstance<ConfigurationStore>());

        For<IConfiguration>().Use(ctx => ctx.GetInstance<ConfigurationStore>());

        For<ITypeMapFactory>().Use<TypeMapFactory>();

        For<IMappingEngine>().Singleton().Use<MappingEngine>()
            .SelectConstructor(() => new MappingEngine(null));

        this.Scan(scanner =>
        {
            scanner.AssembliesFromApplicationBaseDirectory();

            scanner.ConnectImplementationsToTypesClosing(typeof(ITypeConverter<,>))
                .OnAddedPluginTypes(t => t.HybridHttpOrThreadLocalScoped());

            scanner.ConnectImplementationsToTypesClosing(typeof(ValueResolver<,>))
                .OnAddedPluginTypes(t => t.HybridHttpOrThreadLocalScoped());
        });
    }
}
```

هدف اصلی، وهله سازی خودکار IMapperEngine است و برای رسیدن به آن، باید تمام وابستگی‌های کلاس MappingEngine را مانند IConfigurationProvider و سایر مواردی که مشاهده می‌کنید، مشخص کرد. پس از این تنظیمات، کلاس ObjectFactory سفارشی برنامه به شکل ذیل جهت معرفی AutoMapperRegistry تغییر خواهد کرد:

```
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()
    {
        var container = new Container(cfg =>
        {

```



```

        cfg.AddRegistry<AutomapperRegistry>();
        cfg.Scan(scan =>
        {
            scan.TheCallingAssembly();
            scan.WithDefaultConventions();
            scan.AddAllTypesOf<Profile>().NameBy(item => item.FullName);
        });
    });

    configureAutoMapper(container);

    return container;
}

private static void configureAutoMapper(IContainer container)
{
    var configuration = container.TryGetInstance<IConfiguration>();
    if (configuration == null) return;
    //saying AutoMapper how to resolve services
    configuration.ConstructServicesUsing(container.GetInstance);
    foreach (var profile in container.GetAllInstances<Profile>())
    {
        configuration.AddProfile(profile);
    }
}
}

```

در اینجا علاوه بر معرفی AutomapperRegistry، یک مورد دیگر نیز اضافه شده‌است: یافتن خودکار کلاس‌هایی از نوع Profile و همچنین فراخوانی متد AddProfile کتابخانه‌ی AutoMapper به صورت خودکار. به این ترتیب دیگر نیازی نخواهد بود تا در ابتدای کار برنامه، متد Mapper.Initialize را جهت معرفی کلاس‌های Profile فراخوانی کرد و این کار به صورت خودکار توسط متد configureAutoMapper انجام می‌شود.

تغییرات لایه سرویس برنامه جهت استفاده از IoC Container

اکنون که IoC Container ما با نحوه‌ی یافتن وابستگی‌های IMappingEngine آشنا شده‌است، تنها کافی است این اینترفیس را در سازنده‌ی کلاس سرویس خود تزریق کنیم:

```

public class UsersService : IUsersService
{
    private readonly IMappingEngine _mappingEngine;

    public UsersService(IMappingEngine mappingEngine)
    {
        _mappingEngine = mappingEngine;
    }

    public UserViewModel GetName(int id)
    {
        var dbUser1 = new User
        {
            Id = 1,
            Name = "Test",
            RegistrationDate = DateTime.Now.AddDays(-10)
        };

        var uiUser = new UserViewModel();
        _mappingEngine.Map(source: dbUser1, destination: uiUser);
        return uiUser;
    }
}

```

و پس از آن از متد Map این اینترفیس بجای فراخوانی مستقیم Mapper.Map می‌توان استفاده کرد. به این ترتیب وابستگی مورد نیاز این کلاس، از طریق سازنده‌ی آن به آن تزریق شده‌است و دیگر فراخوانی‌های استاتیک را در اینجا مشاهده نمی‌کنیم.

کدهای کامل این قسمت را از اینجا می‌توانید دریافت کنید:

[AM_Sample03.zip](#)

نظرات خوانندگان

نویسنده: سیروان عقیفی
تاریخ: ۱۱:۰۱۳۹۴/۰۲/۰۹

ممنون از شما،

یک سوال: بنده کلاس ObjectFactory را همانطور که فرمودید به [این](#) صورت تغییر دادم. در لایه سرویس نیز این متد را تهیه کرده‌ام:

```
public IList<AdvertisementViewModel> GetAdvertisementsByMe(int userId)
{
    var adsList = _advertisements.Where(x => x.UserId == userId).ToList();
    var adsViewModel = new List<AdvertisementViewModel>();
    _mappingEngine.Map(source: adsList, destination: adsViewModel);
    return adsViewModel;
}
```

در متد فوق کلاس [Advertisement](#) به کلاس زیر نگاشت داده شده است:

```
public class AdvertisementViewModel
{
    public string Image { get; set; }
    public string Title { get; set; }
    public string ExpireDate { get; set; }
}
```

اما با فراخوانی متد GetAdvertisementsByMe استثناء AutoMapperMappingException صادر می‌شود:

Missing type map configuration or unsupported mapping.

Mapping types:

Advertisement -> AdvertisementViewModel

Project.DomainClasses.Advertisement -> Project.Models.AdvertisementViewModel

Destination path:

List`1[0]

Source value:

System.Data.Entity.DynamicProxies.Advertisement_E82DFF273E08C95AA785F8F7A0D2B5ABC8E54C4566DFE1C8A92D8D3C447608AE

نویسنده: وحید نصیری
تاریخ: ۱۱:۳۱۱۳۹۴/۰۲/۰۹

- آیا کلاس پروفایل این نگاشت مورد نظر تعریف شده‌است (کلاس حاوی CreateMap)?

- اگر بله، در متد configureAutoMapper روی سطر configuration.AddProfile یک break point قرار دهید و بررسی کنید که

آیا فراخوانی می‌شود؟ یعنی آیا به صورت خودکار یافت شده و به سیستم اضافه می‌شود یا خیر؟

- اگر این break point فراخوانی نمی‌شود، این کلاس پروفایل در چه اسمبلی قرار دارد؟ بازه‌ی اسکن استراکچرمپ را باید تغییر

دهید یا وسیع‌تر کنید. برای مثال scan.TheCallingAssembly فقط اسمبلی فراخوان را اسکن می‌کند. اگر نیاز است اسمبلی دیگری

هم اسکن شود، از متد AssemblyContainingType استفاده کنید:

```
Scan(scan =>
{
    scan.AssemblyContainingType<خاص اسمبلی خاص>();
    //....
});
```

نویسنده: وحید نصیری

تاریخ: ۱۱:۴۲ ۱۳۹۴/۰۲/۰۹

یک نکته‌ی تکمیلی:

اگر با EF کار می‌کنید و LINQ to Objects نیست، از متد [Project To](#) بهتر است استفاده کنید:

```
ctx.Advertisements.Where(...).Project().To<AdvertisementViewModel>().ToList()
```

مزیت این روش این است که فقط خواص موجود در ViewModel از بانک اطلاعاتی واکنشی می‌شوند؛ برای نمونه در اینجا و این مثال، فقط سه مورد. اگر ابتدا ToList خود EF را فراخوانی کنید، تمام خواص کلاس Advertisement از بانک اطلاعاتی واکنشی خواهند شد و مرحله‌ی بعد LINQ to Objects می‌شود.

نویسنده: سیروان عقیفی
تاریخ: ۱۱:۴۶ ۱۳۹۴/۰۲/۰۹

(: خیلی ممنون، من اصلاً حواسم نبود دقیقاً مشکل همین بود.

نویسنده: سیروان عقیفی
تاریخ: ۱۳:۴۵ ۱۳۹۴/۰۲/۰۹

در این صورت نگاشت کلاس‌ها باید داخل لایه سرویس توسط Mapper.Map صورت گیرد:

```
public IList<AdvertisementViewModel> GetAdvertisementsByMe(int userId)
{
    Mapper.CreateMap<Advertisement, AdvertisementViewModel>();
    var adsList = _advertisements.Where(x => x.UserId ==
userId).Project().To<AdvertisementViewModel>().ToList();
    return adsList;
}
```

چون در غیر اینصورت استثنای InvalidOperationException صادر خواهد شد.

نویسنده: وحید نصیری
تاریخ: ۱۴:۱۹ ۱۳۹۴/۰۲/۰۹

کاری که در اینجا انجام شده، ایجاد یک Mapping Engine سفارشی هست که با Mapping Engine اصلی استاتیک یکی نیست. به همین جهت برای نمونه متد Project آرگومان `Project(_mappingEngine)` هم دارد. اگر قید نشود، یعنی قرار است از موتور نگاشت استاتیک سراسری پیش فرض آن استفاده شود.

نویسنده: مجتبی آزاد
تاریخ: ۱۴:۲۰ ۱۳۹۴/۰۳/۰۶

سلام؛ محل صحیح قرار دادن Mapping‌ها دقیقاً کجای پروژه است؟ آیا مثال همین مطلب صحیح‌ترین محل قرار دادن Mapping‌ها و AutoMapper است؟ در پروژه‌های مختلف و بعضی از مطالب همین وبسایت دیده‌ام که محل دیگری غیر از پروژه Service نیز برای قرار دادن Mapping‌ها انتخاب می‌شود.

نویسنده: وحید نصیری
تاریخ: ۱۴:۳۴ ۱۳۹۴/۰۳/۰۶

- «... صحیح‌ترین ...» «... محل دیگری غیر از پروژه Service ...» این «مثال» اساساً یک پروژه بیشتر نبود؛ صرفاً جهت نمایش مفهوم مورد بحث. در همین «مثال» تعاریف نگاشت‌ها داخل پوشه‌ی سرویس نیست.

در کل می‌توانید یک اسمبلی جداگانه برای آن در نظر بگیرید به نام مثلاً `AutoMapperConfig`. تنها قسمت مهم آن، بارگذاری و خواندن این نگاشت‌ها [در زمان آغاز](#) برنامه است که در مثال جاری، اینکار توسط `SmObjectFactory` به صورت خودکار انجام می‌شود.

در کل هدف از اکثر مثال‌های این سایت یا سایت‌های مشابه دیگر، رساندن یک مفهوم است؛ نه ارائه‌ی یک راه حل جامع و مانع. همینقدر که مثال زده شده، عنوان مورد بحث را پوشش دهد، کافی است.

نویسنده: مجتبی آزاد
تاریخ: ۱۵:۴۲ ۱۳۹۴/۰۳/۰۶

ممنونم از پاسختون.

هدف من بیشتر از طرح این سوال این هست که در طراحی معماری پروژه و به طور خاص جایگاه Mapping در پروژه، بین دو مورد تصمیم گیری کنم:

- ۱- قرار دادن تعاریف Mapping و view model در لایه UI و استفاده از لایه سرویس (با خروجی Entity Model در هر تابع)
- ۲- قراردادن تعاریف Mapping و view model هر کدام در یک پروژه مجزا و استفاده از آن در لایه سرویس، با این توضیح که خروجی متدها در لایه سرویس Viewmodel باشد کدام یک از این موارد صحیح‌تر هست؟

نویسنده: وحید نصیری
تاریخ: ۱۶:۱ ۱۳۹۴/۰۳/۰۶

- محل تعریف نگاشت‌ها و کلاس‌های پروفایل، مهم نیست. چون اساساً هرجایی که قرار گیرند، دو وابستگی بیشتر نخواهند داشت: کلاس‌های مدل و کلاس‌های `ViewModel`.

- [محل فراخوانی اولیه‌ی](#) تعاریف نگاشت‌ها جهت معرفی آن‌ها به سیستم، مهم است.

+ اگر از کاربر اطلاعاتی را دریافت می‌کنید، در لایه UI هست که کار نگاشت اطلاعات دریافتی از کاربر و از `ViewModel`‌ها به `Model`‌های اصلی برنامه انجام می‌شود (توسط متد `Mapper.Map`). اگر قرار است اطلاعاتی را بازگشت دهید، متدهای جدیدی مانند `Project To` بسیار بهینه‌تر هستند از روش قدیمی `Mapper.Map` و این متد را بهتر است در لایه سرویس استفاده کنید. متد `Project To` کارش بهینه سازی کوئری SQL ارسالی به سرور هست. اگر از روش `Mapper.Map` در لایه UI استفاده کنید، این قابلیت را از دست خواهید داد؛ چون `Mapper.Map` به معنای کار با اشیاء درون حافظه و `LINQ to Objects` است. کار متد ویژه‌ی `Project To` افزونه‌ای برای کار با `Entity Framework` و بهینه سازی آن است.

نویسنده: مجتبی آزاد
تاریخ: ۱۳:۲ ۱۳۹۴/۰۳/۰۹

من تنظیمات تزریق وابستگی مربوط به `AutoMapper` را در همان محل قرارگیری تزریق وابستگی `Service`‌ها قرار داده ام و `ObjectFactory` به این شکل شد:

```
public static class WoObjectFactory
{
    private static readonly Lazy<Container> ContainerBuilder =
        new Lazy<Container>(DefaultContainer, LazyThreadSafetyMode.ExecutionAndPublication);

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

    private static Container DefaultContainer()
    {
        var _container = new Container(x =>
        {
            var platformSpecificRegistry =
                PlatformAdapter.Resolve<IPlatformSpecificMapperRegistry>();
            platformSpecificRegistry.Initialize();
        });
    }
}
```

```

x.For<ConfigurationStore>().Singleton().Use<ConfigurationStore>().Ctor<IEnumerable<IObjectMapper>>().Is
(MapperRegistry.Mappers);
x.For<IConfigurationProvider>().Use(ctx => ctx.GetInstance<ConfigurationStore>());
x.For<IConfiguration>().Use(ctx => ctx.GetInstance<ConfigurationStore>());
x.For<ITypeMapFactory>().Use<TypeMapFactory>();
x.For<IMappingEngine>().Singleton().Use<MappingEngine>().SelectConstructor(() => new
MappingEngine(null));

x.For<IUnitOfWork>().HybridHttpOrThreadLocalScoped().Use(() => new
WirelessOrganizationContext());

x.Scan(scan =>
{
    scan.AssemblyContainingType<IDeviceService>();
    scan.TheCallingAssembly();
    scan.WithDefaultConventions();
});
x.Scan(scanner =>
{
    scanner.AssembliesFromApplicationBaseDirectory();

    scanner.ConnectImplementationsToTypesClosing(typeof(ITypeConverter<,>))
        .OnAddedPluginTypes(t => t.HybridHttpOrThreadLocalScoped());

    scanner.ConnectImplementationsToTypesClosing(typeof(ValueResolver<,>))
        .OnAddedPluginTypes(t => t.HybridHttpOrThreadLocalScoped());
});

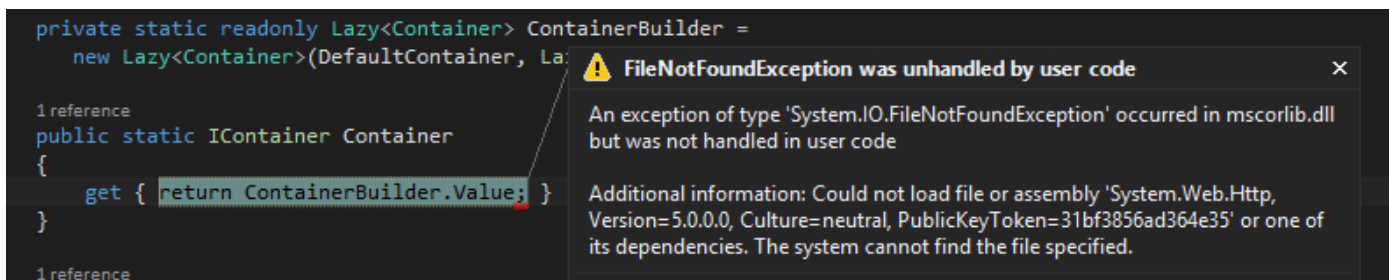
x.Scan(scan =>
{
    scan.TheCallingAssembly();
    scan.WithDefaultConventions();
    scan.AddAllTypesOf<Profile>().NameBy(item => item.FullName);
});

});
ConfigureAutoMapper(_container);
return _container;
}

private static void ConfigureAutoMapper(IContainer container)
{
    var configuration = container.TryGetInstance<IConfiguration>();
    if (configuration == null) return;
    //saying AutoMapper how to resolve services
    configuration.ConstructServicesUsing(container.GetInstance);
    foreach (var profile in container.GetAllInstances<Profile>())
    {
        configuration.AddProfile(profile);
    }
}
}

```

اما وقتی پروژه اجرا می‌شود اکسپشن زیر اتفاق می‌افتد، که البته بعد از اضافه کردن تنظیمات مربوط به تزریق وابستگی AutoMapper اتفاق افتاد، در این Value مربوط به ContainerBuilder مقداری ندارد:



تاریخ: ۱۳:۳۳ ۱۳۹۴/۰۳/۰۹

- استثنای صادر شده مربوط است به یافت نشدن اسمبلی `System.Web.Http`. در لیست ارجاعات برنامه، این ارجاع را یافته و خاصیت `copy to local` آن را `true` کنید؛ چیزی شبیه به [این مشکل](#)

- همچنین اگر Solution شما چند پروژه‌ای است، احتمال دارد که قسمت‌های مختلف آن از اسمبلی‌های مشابهی، اما با نگارش‌های مختلفی استفاده می‌کنند. اگر این اسمبلی‌ها از طریق نیوگت اضافه شده‌اند، دستور ذیل را صادر کنید:

PM> Update-Package

اگر خیر، فایل‌های `csproj` را باید تک تک بررسی کنید و شماره نگارش‌های اسمبلی‌های مشابه را تطابق دهید.

- مطلب « [به روز رسانی قسمت assemblyBinding فایل‌های config توسط NuGet](#) » را هم مدنظر داشته باشید.

نویسنده: مجتبی آزاد
تاریخ: ۱۵:۵۹ ۱۳۹۴/۰۳/۰۹

لزوما باید برای تمام نگاشت‌ها کلاس پروفایل ساخت؟

آیا امکان این وجود دارد با روشی که در بالا گفته شد، این کار به صورت خودکار انجام شود؟

اگر خیر، علت اینکه DynamicMap بدون نیاز به پروفایل به درستی عمل می‌کند چیست؟

نویسنده: وحید نصیری
تاریخ: ۱۶:۱۶ ۱۳۹۴/۰۳/۰۹

وجود تنظیمات صریح در ابتدای برنامه، کار برپایی مقدمات Fast Reflection را ساده‌تر می‌کند و در نتیجه روی سرعت و کارایی برنامه [تاثیر مثبتی](#) خواهد داشت. به این ترتیب این تنظیمات یکبار ایجاد شده و کش می‌شوند.

نویسنده: غلامرضا ربال
تاریخ: ۸:۱ ۱۳۹۴/۰۵/۰۳

با ورژن قبلی آن مشکلی وجود نداشت ولی در ورژن 4 آن اینترفیس `IPlatformSpecificMapperRegistry` موجود نیست. به چه شکل باید عمل کنیم برای فراخوانی متد `Resolve` ؟

نویسنده: وحید نصیری
تاریخ: ۹:۵۸ ۱۳۹۴/۰۵/۰۳

- هر زمان که نگارش 4 آن [نهایی شد](#)، این مسایل باید بررسی شوند.

- همچنین این مسایل را هم باید با [نویسنده‌ی اصلی آن](#) مطرح کنید؛ نه اینجا.

نویسنده: وحید نصیری
تاریخ: ۱۹:۲ ۱۳۹۴/۰۵/۱۹

جهت تکمیل بحث

در نگارش 4 صرفا این دو سطر را حذف کنید:

```
var platformSpecificRegistry = PlatformAdapter.Resolve<IPlatformSpecificMapperRegistry>();
platformSpecificRegistry.Initialize();
```

این موارد به صورت توکار توسط خود AutoMapper لحاظ شده‌است و نیازی به آن‌ها نیست.

پلتفرم‌های مختلف در نگارش 4 به صورت یک اسمبلی مجزا به ازای هر پلتفرم ارائه شده‌اند و اینبار مانند قبل یکی نیستند.

فرض کنید مدل‌های بانک اطلاعاتی ما چنین ساختاری را دارند:

```
public abstract class BaseEntity
{
    public int Id { get; set; }
}

public class User : BaseEntity
{
    public string Name { get; set; }

    public virtual ICollection<Advertisement> Advertisements { get; set; }
}

public class Advertisement : BaseEntity
{
    public string Title { get; set; }
    public string Description { get; set; }

    [ForeignKey("UserId")]
    public virtual User User { get; set; }
    public int UserId { get; set; }
}
```

و همچنین مدل‌های رابط کاربری یا ViewModel‌های برنامه نیز به صورت ذیل تعریف شده‌اند:

```
public class AdvertisementViewModel
{
    public int Id { get; set; }
    public string Title { get; set; }
    public int UserId { get; set; }
}

public class UserViewModel
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<AdvertisementViewModel> Advertisements { get; set; }
}
```

به روز رسانی خواص راهبری Entity framework توسط AutoMapper

در کلاس‌های فوق، یک کاربر، تعدادی تبلیغات را می‌تواند ثبت کند. در این حالت اگر بخواهیم خاصیت User کلاس Advertisement را توسط AutoMapper به روز کنیم، با رعایت دو نکته، اینکار به سادگی انجام خواهد شد:

الف) همانطور که در کلاس Advertisement جهت تعریف کلید خارجی مشخص است، UserId نیز علاوه بر User ذکر شده‌است. این مورد کار نگاشت UserId اطلاعات دریافتی از کاربر را ساده کرده و در این حالت نیازی به یافتن اصل User این UserId از بانک اطلاعاتی نخواهد بود.

ب) چون در اطلاعات دریافتی از کاربر تنها Id او را داریم و نه کل شیء مرتبط را، بنابراین باید به AutoMapper اعلام کنیم تا از این خاصیت صرف‌نظر کند که اینکار توسط متد Ignore به نحو ذیل قابل انجام است:

```
this.CreateMap<AdvertisementViewModel, Advertisement>()
    .ForMember(advertisement => advertisement.Description, opt => opt.Ignore())
    .ForMember(advertisement => advertisement.User, opt => opt.Ignore());
```

فرض کنید چنین اطلاعاتی از کاربر و رابط کاربری برنامه دریافت شده است:

```
var uiUser1 = new UserViewModel
{
    Id = 1,
    Name = "user 1",
    Advertisements = new List<AdvertisementViewModel>
    {
        new AdvertisementViewModel
        {
            Id = 1,
            Title = "Adv 1",
            UserId = 1
        },
        new AdvertisementViewModel
        {
            Id = 2,
            Title = "Adv 2",
            UserId = 1
        }
    }
};
```

اکنون می‌خواهیم معادل این رکورد را از بانک اطلاعاتی یافته و سپس اطلاعات آن‌را بر اساس اطلاعات UI به روز کنیم. شاید در نگاه اول چنین روشی پیشنهاد شود:

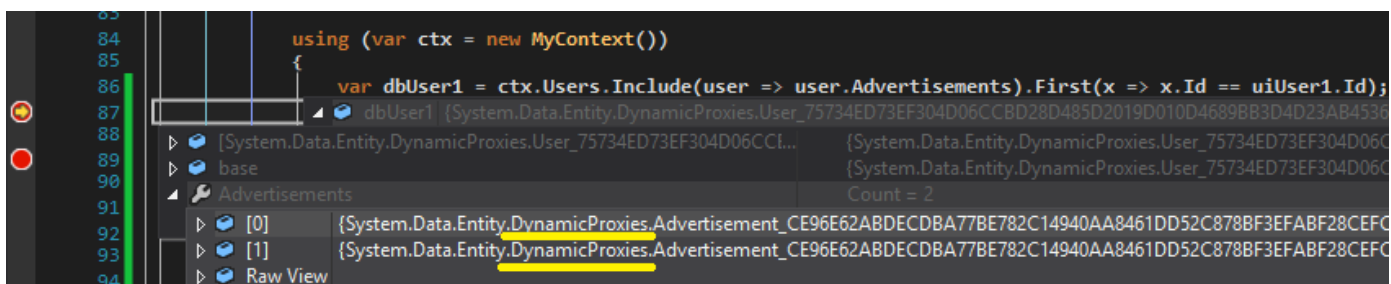
```
var dbUser1 = ctx.Users.Include(user => user.Advertisements).First(x => x.Id == uiUser1.Id);
Mapper.Map(source: uiUser1, destination: dbUser1);
```

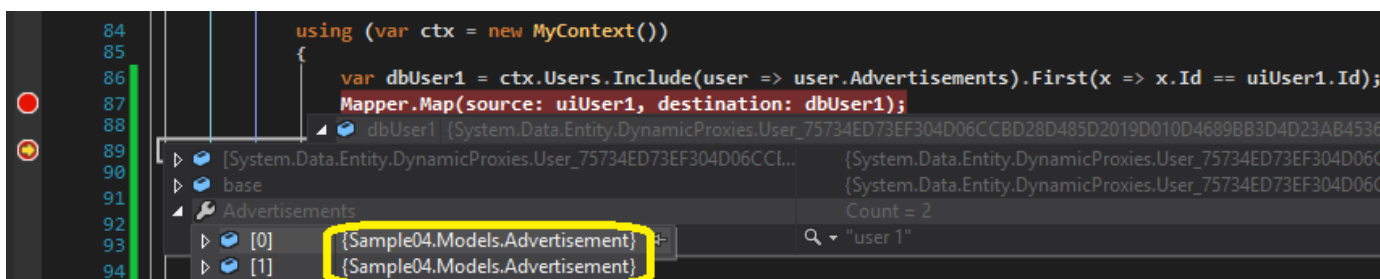
ابتدا کاربری را که Id آن مساوی uiUser1.Id است، یافته و سپس به AutoMapper اعلام می‌کنیم تا تمام اطلاعات آن‌را به صورت یکجا به روز کند. این نگاشت را نیز برای آن تعریف خواهیم کرد:

```
this.CreateMap<UserViewModel, User>()
```

در یک چنین حالتی، ابتدا شیء user 1 از بانک اطلاعاتی دریافت شده (و با توجه به وجود Include، تمام تبلیغات او نیز دریافت می‌شوند)، سپس ... دو رکورد دریافتی از کاربر، کاملاً جایگزین اطلاعات موجود می‌شوند. این جایگزینی سبب تخریب پروکسی‌های EF می‌گردند. برای مثال اگر پیشتر تبلیغی با Id=1 در بانک اطلاعاتی وجود داشته، اکنون با نمونه‌ی جدیدی جایگزین می‌شود که سیستم Tracking و ردیابی EF اطلاعاتی در مورد آن ندارد. به همین جهت اگر در این حالت ctx.SaveChanges فراخوانی شود، عملیات ثبت و یا به روز رسانی با شکست مواجه خواهد شد.

علت را در این دو تصویر بهتر می‌توان مشاهده کرد:





تصویر اول که مستقیماً از بانک اطلاعاتی حاصل شده‌است، دارای پروکسی‌های EF است. اما در تصویر دوم، جایگزین شدن این پروکسی‌ها را مشاهده می‌کنید که سبب خواهد شد این اشیاء دیگر تحت نظارت EF نباشند.

راه حل:

در این مورد خاص باید به AutoMapper اعلام کنیم تا کاری با لیست تبلیغات کاربر دریافت شده‌ی از بانک اطلاعاتی نداشته باشد و آن‌را راساً جایگزین نکند:

```
this.CreateMap<UserViewModel, User>().ForMember(user => user.Advertisements, opt => opt.Ignore());
```

در اینجا متد Ignore را بر روی لیست تبلیغات کاربر بانک اطلاعاتی فراخوانی کرده‌ایم، تا اطلاعات آن پس از اولین نگاهشت انجام شده‌ی توسط AutoMapper دست نخورده باقی بماند. سپس کار ثبت یا به روز رسانی را به صورت نیمه خودکار مدیریت می‌کنیم:

```
using (var ctx = new MyContext())
{
    var dbUser1 = ctx.Users.Include(user => user.Advertisements).First(x => x.Id == uiUser1.Id);
    Mapper.Map(source: uiUser1, destination: dbUser1);

    foreach (var uiUserAdvertisement in uiUser1.Advertisements)
    {
        var dbUserAdvertisement = dbUser1.Advertisements.FirstOrDefault(ad => ad.Id == uiUserAdvertisement.Id);
        if (dbUserAdvertisement == null)
        {
            // Add new record
            var advertisement = Mapper.Map<AdvertisementViewModel, Advertisement>(uiUserAdvertisement);
            dbUser1.Advertisements.Add(advertisement);
        }
        else
        {
            // Update the existing record
            Mapper.Map(uiUserAdvertisement, dbUserAdvertisement);
        }
    }

    ctx.SaveChanges();
}
```

- در اینجا ابتدا db user معادل اطلاعات ui user از بانک اطلاعاتی، به همراه لیست تبلیغات او دریافت می‌شود و اطلاعات ابتدایی او نگاهشت خواهند شد.

- سپس بر روی اطلاعات تبلیغات دریافتی از کاربر، یک حلقه را تشکیل خواهیم داد. در اینجا هربار بررسی می‌کنیم که آیا معادل این تبلیغ هم اکنون به شیء db user متصل است یا خیر؟ اگر متصل نبود، یعنی یک رکورد جدید است و باید Add شود. اگر متصل بود صرفاً باید به روز رسانی صورت گیرد.

- برای حالت ایجاد شیء جدید بانک اطلاعاتی، بر اساس uiUserAdvertisement دریافتی، می‌توان از متد Mapper.Map استفاده کرد؛ خروجی این متد، یک شیء جدید تبلیغ است.

- برای حالت به روز رسانی اطلاعات db user موجود، بر اساس اطلاعات ارسالی کاربر نیز می‌توان از متد Mapper.Map کمک

گرفت.

نکته‌ی مهم

چون در اینجا از متد Include استفاده شده‌است، فراخوانی‌های FirstOrDefault داخل حلقه، سبب رفت و برگشت اضافه‌تری به بانک اطلاعاتی نخواهند شد.

کدهای کامل این قسمت را از اینجا می‌توانید دریافت کنید:

[AM_Sample04.zip](#)

در سناریوهای متداول نگاشت اشیاء، مشخص است که نوع ViewModel برنامه چیست و معادل Model آن کدام است. اما حالت‌هایی مانند کار با anonymous objects و یا data reader و data table و امثال آن نیز وجود دارند که در این حالت‌ها، نوع منبع داده‌ی مورد استفاده، شیء مشخصی نیست که بتوان آن‌را در قسمت CreateMap مشخص کرد. برای مدیریت یک چنین حالت‌هایی، متد DynamicMap طراحی شده‌است.

مثال اول: تبدیل یک DataTable به لیست جنریک معادل

فرض کنید یک DataTable را با ساختار و داده‌های ذیل در اختیار داریم:

```
var dataTable = new DataTable("SalaryList");
dataTable.Columns.Add("User", typeof (string));
dataTable.Columns.Add("Month", typeof (int));
dataTable.Columns.Add("Salary", typeof (decimal));

var rnd = new Random();
for (var i = 0; i < 200; i++)
    dataTable.Rows.Add("User " + i, rnd.Next(1, 12), rnd.Next(400, 2000));
```

نوع این DataTable کاملاً پویا است و می‌تواند هر بار در قسمت‌های مختلف برنامه تعریف متفاوتی داشته باشد. در ادامه معادل کلاس ساختار ستون‌های این DataTable را به صورت ذیل تهیه می‌کنیم.

```
public class SalaryList
{
    public string User { set; get; }
    public int Month { set; get; }
    public decimal Salary { set; get; }
}
```

اکنون می‌خواهیم اطلاعات DataTable را به لیستی جنریک از SalaryList نگاشت کنیم. برای اینکار تنها کافی است از متد DaynamicMap استفاده نمائیم:

```
var salaryList = AutoMapper.Mapper.DynamicMap<IDataReader,
List<SalaryList>>(dataTable.CreateDataReader());
```

منبع داده را از نوع IDataReader بر اساس متد CreateDataReader مشخص کرده‌ایم. به این ترتیب AutoMapper قادر خواهد بود تا اطلاعات این DataTable را به صورت خودکار پیمایش کند. سپس مقصد را نیز لیست جنریکی از کلاس SalaryList تعیین کرده‌ایم. مابقی کار را متد DynamicMap انجام می‌دهد.

کار با AutoMapper نسبت به [راه حل‌های Reflection متداول](#) بسیار سریعتر است. زیرا AutoMapper از مباحث [Fast reflection](#) به صورت توکار استفاده می‌کند.

مثال دوم: تبدیل لیستی از اشیاء anonymous به لیستی جنریک

در اینجا قصد داریم یک شیء anonymous را به شیء معادل SalaryList آن نگاشت کنیم. این کار را نیز می‌توان توسط متد DynamicMap انجام داد:

```
var anonymousObject = new
{
    User = "User 1",
    Month = 1,
    Salary = 100000
}
```

```
};  
var salary = Mapper.DynamicMap<SalaryList>(anonymousObject);
```

و یا نمونه‌ی دیگر آن تبدیل یک لیست anonymous به معادل جنریک آن است که به نحو ذیل قابل انجام است:

```
var anonymousList = new[]  
{  
    new  
    {  
        User = "User 1",  
        Month = 1,  
        Salary = 100000  
    },  
    new  
    {  
        User = "User 2",  
        Month = 1,  
        Salary = 300000  
    }  
};  
var salaryList = anonymousList.Select(item => Mapper.DynamicMap<SalaryList>(item)).ToList();
```

این نکته در مورد حاصل کوئری‌های LINQ یا IQueryable ها نیز صادق است.

مثال سوم: نگاشت پویا به یک اینترفیس

فرض کنید یک چنین اینترفیسی، در برنامه تعریف شده‌است و همچنین دارای هیچ نوع پیاده سازی هم در برنامه نیست:

```
public interface ICustomerService  
{  
    string Code { get; set; }  
    string Name { get; set; }  
}
```

اکنون قصد داریم یک شیء anonymous را به آن نگاشت کنیم:

```
var anonymousObject = new  
{  
    Code = "111",  
    Name = "Test 1"  
};  
var result = Mapper.DynamicMap<ICustomerService>(anonymousObject);
```

در این حالت خاص، AutoMapper با استفاده از یک [Dynamic Proxy](#) به نام LinFu (که با اسمبلی آن Merge شده‌است)، پیاده سازی پویایی را از اینترفیس مشخص شده تهیه کرده و سپس کار نگاشت را انجام می‌دهد.

کدهای کامل این قسمت را از اینجا می‌توانید دریافت کنید:

[AM_Sample05.zip](#)

AutoMapper تنها کتابخانه‌ی نگاشت اشیاء مخصوص دات نت نیست. در این مطلب قصد داریم سرعت AutoMapper را با حالت نگاشت دستی، نگاشت توسط [EmitMapper](#) و نگاشت به کمک [ValueInjector](#)، مقایسه کنیم.

مدل مورد استفاده

در اینجا قصد داریم، شیء User را یک میلیون بار توسط روش‌های مختلف، به خودش نگاشت کنیم و سرعت انجام این کار را در حالت‌های مختلف اندازه گیری نمائیم:

```
public class User
{
    public int Id { get; set; }
    public string UserName { get; set; }
    public string Password { get; set; }
    public DateTime LastLogin { get; set; }
}
```

روش بررسی سرعت انجام هر روش

برای کاهش کدهای تکراری، می‌توان قسمت تکرار شونده را به صورت یک Action، در بین سایر کدهایی که هر بار نیاز است به یک شکل فراخوانی شوند، قرار داد:

```
public static void RunActionMeasurePerformance(Action action)
{
    GC.Collect();
    var initMemUsage = Process.GetCurrentProcess().WorkingSet64;
    var stopwatch = new Stopwatch();
    stopwatch.Start();
    action();
    stopwatch.Stop();
    var currentMemUsage = Process.GetCurrentProcess().WorkingSet64;
    var memUsage = currentMemUsage - initMemUsage;
    if (memUsage < 0) memUsage = 0;
    Console.WriteLine("Elapsed time: {0}, Memory Usage: {1:N2} KB", stopwatch.Elapsed, memUsage /
1024);
}
```

انجام آزمایش

در مثال زیر، ابتدا یک میلیون شیء User ایجاد می‌شوند و سپس هر بار توسط روش‌های مختلفی به شیء User دیگری نگاشت می‌شوند:

```
static void Main(string[] args)
{
    var length = 1000000;
    var users = new List<User>(length);
    for (var i = 0; i < length; i++)
    {
        var user = new User
        {
            Id = i,
            UserName = "User" + i,
            Password = "1" + i + "2" + i,
            LastLogin = DateTime.Now
        }
    }
}
```

```

    };
    users.Add(user);
}

Console.WriteLine("Custom mapping");
RunActionMeasurePerformance(() =>
{
    var userList =
        users.Select(
            o =>
                new User
                {
                    Id = o.Id,
                    UserName = o.UserName,
                    Password = o.Password,
                    LastLogin = o.LastLogin
                }).ToList();
});

Console.WriteLine("EmitMapper mapping");
RunActionMeasurePerformance(() =>
{
    var map = EmitMapper.ObjectMapperManager.DefaultInstance.GetMapper<User, User>();
    var emitUsers = users.Select(o => map.Map(o)).ToList();
});

Console.WriteLine("ValueInjector mapping");
RunActionMeasurePerformance(() =>
{
    var valueUsers = users.Select(o => (User)new User().InjectFrom(o)).ToList();
});

Console.WriteLine("AutoMapper mapping, DynamicMap using List");
RunActionMeasurePerformance(() =>
{
    var userMap = Mapper.DynamicMap<List<User>>(users).ToList();
});

Console.WriteLine("AutoMapper mapping, Map using List");
RunActionMeasurePerformance(() =>
{
    var userMap = Mapper.Map<List<User>>(users).ToList();
});

Console.WriteLine("AutoMapper mapping, Map using IEnumerable");
RunActionMeasurePerformance(() =>
{
    var userMap = Mapper.Map<IEnumerable<User>>(users).ToList();
});

Console.ReadKey();
}

```

خروجی آزمایش

در ادامه یک نمونه‌ی خروجی نهایی را مشاهده می‌کنید:

```

Custom mapping
Elapsed time: 00:00:00.4869463, Memory Usage: 58,848.00 KB

EmitMapper mapping
Elapsed time: 00:00:00.6068193, Memory Usage: 62,784.00 KB

ValueInjector mapping
Elapsed time: 00:00:15.6935578, Memory Usage: 21,140.00 KB

AutoMapper mapping, DynamicMap using List
Elapsed time: 00:00:00.6028971, Memory Usage: 7,164.00 KB

AutoMapper mapping, Map using List
Elapsed time: 00:00:00.0106244, Memory Usage: 680.00 KB

AutoMapper mapping, Map using IEnumerable
Elapsed time: 00:00:01.5954456, Memory Usage: 40,248.00 KB

```

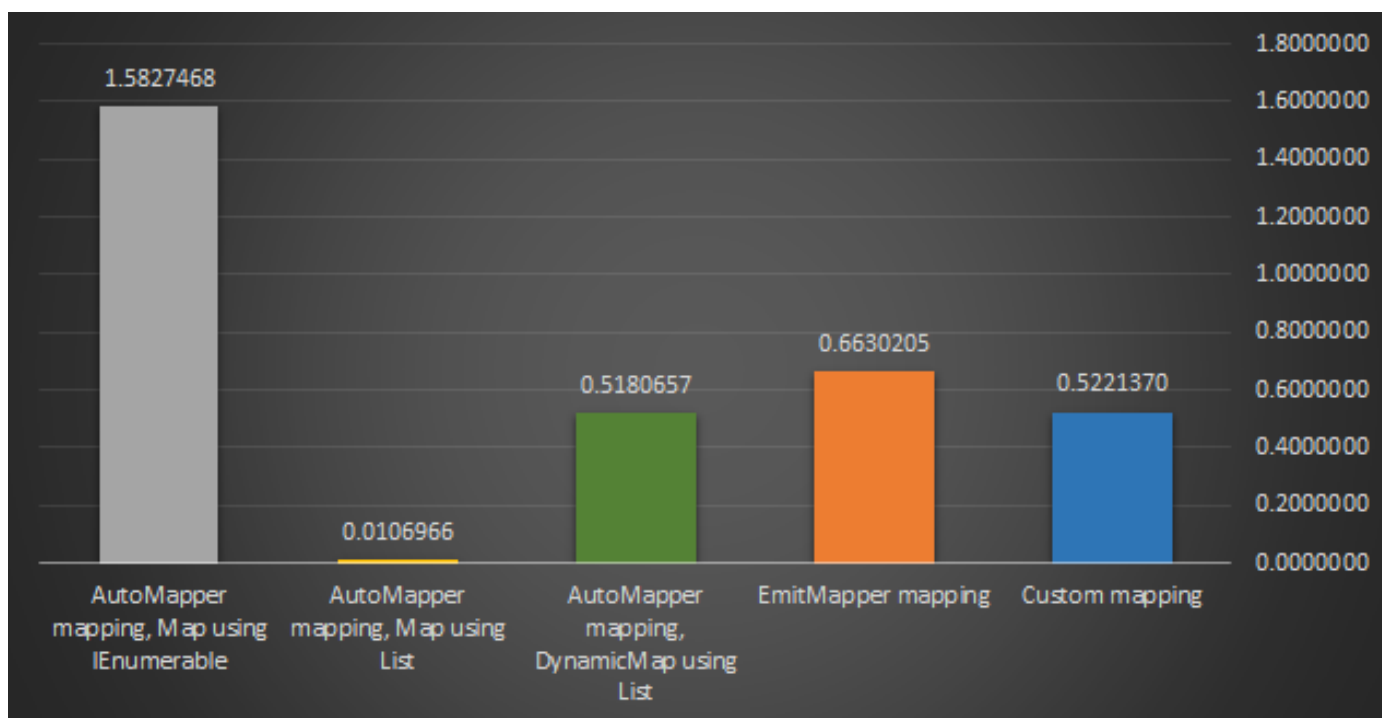
ValueInjector از همه کندتر است.

EmitMapper از AutoMapper سریعتر است (البته فقط در بعضی از حالت‌ها).

سرعت AutoMapper زمانیکه نوع آرگومان ورودی به آن به IEnumerable تنظیم شود، نسبت به حالت استفاده از List معمولی، به مقدار قابل توجهی کندتر است. زمانیکه از List استفاده شده، سرعت آن از سرعت حالت نگاشت دستی (مورد اول) هم بیشتر است.

متد DynamicMap اندکی کندتر است از متد Map.

در این بین اگر ValueInjector را از لیست حذف کنیم، به نمودار ذیل خواهیم رسید (اعداد آن برحسب ثانیه هستند):



البته حین انتخاب یک کتابخانه، باید به آخرین تاریخ به روز شدن آن نیز دقت داشت و همچنین میزان استقبال جامعه‌ی برنامه نویس‌ها و از این لحاظ، AutoMapper نسبت به سایر کتابخانه‌های مشابه در صدر قرار می‌گیرد.

کدهای کامل این قسمت را از اینجا می‌توانید دریافت کنید:

[AM_Sample06.zip](#)

Mini ORM ها برخلاف ORM های کاملی مانند Entity framework یا NHibernate، کوئری های LINQ را تبدیل به SQL نمی کنند. در اینجا کار با SQL نویسی مستقیم شروع می شود و مهم ترین کار این کتابخانه ها، نگاشت نتیجه ی دریافتی از بانک اطلاعاتی به اشیاء دات نت هستند. خوب ... AutoMapper هم دقیقاً همین کار را انجام می دهد! بنابراین در ادامه قصد داریم یک Mini ORM را به کمک AutoMapper طراحی کنیم.

کلاس پایه AdoMapper

```
public abstract class AdoMapper<T> where T : class
{
    private readonly SqlConnection _connection;

    protected AdoMapper(string connectionString)
    {
        _connection = new SqlConnection(connectionString);
    }

    protected virtual IEnumerable<T> ExecuteCommand(SqlCommand command)
    {
        command.Connection = _connection;
        command.CommandType = CommandType.StoredProcedure;
        _connection.Open();

        try
        {
            var reader = command.ExecuteReader();
            try
            {
                return Mapper.Map<IDataReader, IEnumerable<T>>(reader);
            }
            finally
            {
                reader.Close();
            }
        }
        finally
        {
            _connection.Close();
        }
    }

    protected virtual T GetRecord(SqlCommand command)
    {
        command.Connection = _connection;
        _connection.Open();
        try
        {
            var reader = command.ExecuteReader();
            try
            {
                reader.Read();
                return Mapper.Map<IDataReader, T>(reader);
            }
            finally
            {
                reader.Close();
            }
        }
        finally
        {
            _connection.Close();
        }
    }

    protected virtual IEnumerable<T> GetRecords(SqlCommand command)
    {
        command.Connection = _connection;
        _connection.Open();
    }
}
```



```

    try
    {
        var reader = command.ExecuteReader();
        try
        {
            return Mapper.Map<IDataReader, IEnumerable<T>>(reader);
        }
        finally
        {
            reader.Close();
        }
    }
    finally
    {
        _connection.Close();
    }
}
}

```

در اینجا کلاس پایه Mini ORM طراحی شده را ملاحظه می‌کنید. برای نمونه قسمت GetRecords آن مانند مباحث استاندارد ADO.NET است. فقط کار خواندن و همچنین نگاشت رکوردهای دریافت شده از بانک اطلاعاتی به شیءایی از نوع T توسط AutoMapper انجام خواهد شد.

نحوه‌ی استفاده از کلاس پایه AdoMapper

در کدهای ذیل نحوه‌ی ارث بری از کلاس پایه AdoMapper و سپس استفاده از متدهای آن را ملاحظه می‌کنید:

```

public class UsersService : AdoMapper<User>, IUserService
{
    public UsersService(string connectionString)
        : base(connectionString)
    {
    }

    public IEnumerable<User> GetAll()
    {
        using (var command = new SqlCommand("SELECT * FROM Users"))
        {
            return GetRecords(command);
        }
    }

    public User GetById(int id)
    {
        using (var command = new SqlCommand("SELECT * FROM Users WHERE Id = @id"))
        {
            command.Parameters.Add(new SqlParameter("id", id));
            return GetRecord(command);
        }
    }
}

```

در این مثال نحوه‌ی تعریف کوئری‌های پارامتری نیز در متد GetById به نحو متداولی مشخص شده‌است. کار نگاشت حاصل این کوئری‌ها به اشیاء دات نتی را AutoMapper انجام خواهد داد. نحوه‌ی کار نیز، نگاشت فیلد f1 به خاصیت f1 است (هم نام‌ها به هم نگاشت می‌شوند).

تعریف پروفایل مخصوص AutoMapper

ORM‌های تمام عیار، کار نگاشت فیلدهای بانک اطلاعاتی را به خواص اشیاء دات نتی، به صورت خودکار انجام می‌دهند. در اینجا همانند روش‌های متداول کار با AutoMapper نیاز است این نگاشت را به صورت دستی یکبار تعریف کرد:

```

public class UsersProfile : Profile
{
    protected override void Configure()
    {
    }
}

```

```

        this.CreateMap<IDataRecord, User>();
    }

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

```

و سپس در ابتدای برنامه آنرا به AutoMapper معرفی نمود:

```

Mapper.Initialize(cfg => // In Application_Start()
{
    cfg.AddProfile<UsersProfile>();
});

```

سفارشی سازی نگاشت‌های AutoMapper

فرض کنید کلاس Advertisement زیر، معادل است با جدول Advertisements بانک اطلاعاتی؛ با این تفاوت که در کلاس تعریف شده، خاصیت TitleWithOtherName تطابقی با هیچکدام از فیلدهای بانک اطلاعاتی ندارد. بنابراین اطلاعاتی نیز به آن نگاشت نخواهد شد.

```

public class Advertisement
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Description { get; set; }
    public int UserId { get; set; }

    public string TitleWithOtherName { get; set; }
}

```

برای رفع این مشکل می‌توان حین تعریف پروفایل مخصوص Advertisement، آنرا سفارشی سازی نیز نمود:

```

public class AdvertisementsProfile : Profile
{
    protected override void Configure()
    {
        this.CreateMap<IDataRecord, Advertisement>()
            .ForMember(dest => dest.TitleWithOtherName,
                options => options.MapFrom(src =>
                    src.GetString(src.GetOrdinal("Title"))));
    }

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

```

در اینجا پس از تعریف نگاشت مخصوص کار با IDataRecord ها، عنوان شده‌است که هر زمانیکه به خاصیت TitleWithOtherName رسیدی، مقدارش را از فیلد Title دریافت و جایگزین کن.

کدهای کامل این مطلب را [از اینجا](#) می‌توانید دریافت کنید.

نظرات خوانندگان

نویسنده: امین کاشانی
تاریخ: ۱۳۹۴/۰۲/۱۸ ۲۳:۵۰

باسلام
به نظرتون auto mapper در مقایسه با dapper کدام یک بهتر و کامل تر هست؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۲/۱۸ ۲۳:۵۳

این قیاس صحیح نیست چون AutoMapper یک Mini ORM نیست؛ اما می‌توان بر اساس آن یک Mini ORM ساخت که نمونه‌ای از آن در اینجا مطرح شده‌است.

فرض کنید مدل متناظر با جدول بانک اطلاعاتی دانشجویان، به صورت ذیل تعریف شده‌است و دارای یک فیلد محاسباتی است:

```
public class Student
{
    public int Id { get; set; }

    [Required]
    [StringLength(450)]
    public string LastName { get; set; }

    [Required]
    [StringLength(450)]
    public string FirstName { get; set; }

    [NotMapped]
    public string FullName
    {
        get
        {
            return LastName + ", " + FirstName;
        }
    }
}
```

فیلد محاسباتی FullName را نمی‌توان در کوئری‌های EF بکار برد؛ زیرا کوئری‌های LINQ نوشته شده در اینجا باید قابلیت ترجمه‌ی به SQL را داشته باشند و چون در بانک اطلاعاتی برنامه، فیلدی به نام FullName وجود ندارد، نمی‌توان FullName را مورد استفاده قرار داد.

تبدیل خواص محاسباتی به کوئری‌های SQL به کمک DelegateDecompiler

هر «نمی‌توانی» را می‌توان تبدیل به یک پروژه‌ی ابتکاری کرد! و اینکار توسط پروژه‌ای به نام [DelegateDecompiler](#) انجام شده‌است.

کوئری متداول ذیل، قابل اجرا نیست و با یک استثناء متوقف می‌شود؛ زیرا همانطور که عنوان شد، FullName قابل تبدیل به SQL نیست.

```
var fullNames = context.Students.Select(x => x.FullName).ToList();
```

اما اگر همین کوئری را توسط DelegateDecompiler بازنویسی کنیم:

```
var fullNames = context.Students.Select(x => x.FullName).Decompile().ToList();
```

بدون مشکل اجرا می‌شود. اینبار FullName به صورت ذیل تبدیل به عبارت SQL معادلی خواهد شد:

```
SELECT
    [Extent1].[LastName] + N', ' + [Extent1].[FirstName] AS [C1]
FROM [dbo].[Students] AS [Extent1]
```

برای استفاده‌ی از DelegateDecompiler دو کار باید انجام شود:

الف) خاصیت محاسباتی مدنظر را با ویژگی Computed مزین کنید:

```
[NotMapped]
[Computed]
public string FullName
```

ب) از متد الحاقی Decpile در کوئری تهیه شده استفاده نمائید.

استفاده از DelegateDecompiler به همراه AutoMapper

فرض کنید ViewModel ایی که قرار است به کاربر نمایش داده شود، ساختار ذیل را دارد:

```
public class StudentViewModel
{
    public int Id { get; set; }
    public string FullName { get; set; }
}
```

کوئری ذیل که از [Project To](#) مخصوص AutoMapper جهت نگاشت اطلاعات دریافتی از بانک اطلاعاتی به StudentViewModel استفاده می‌کند، با توجه به اینکه کار نوشتن Select را به صورت خودکار بر اساس خاصیت FullName انجام می‌دهد، قابلیت اجرای بر روی بانک اطلاعاتی را نخواهد داشت:

```
var students = context.Students.Project().To<StudentViewModel>().ToList();
```

برای رفع این مشکل تنها کافی است از متد Decpile کتابخانه‌ی DelegateDecompiler به نحو ذیل استفاده کنیم:

```
var students = context.Students
    .Project()
    .To<StudentViewModel>()
    .Decpile()
    .ToList();
```

اینبار کوئری ارسال شده‌ی به بانک اطلاعاتی، یک چنین شکلی را پیدا می‌کند:

```
SELECT
    [Extent1].[Id] AS [Id],
    [Extent1].[LastName] + N', ' + [Extent1].[FirstName] AS [C1]
FROM [dbo].[Students] AS [Extent1]
```

کدهای کامل این مطلب را [از اینجا](#) می‌توانید دریافت کنید.

نظرات خوانندگان

نویسنده: وحید نصیری
تاریخ: ۱۹:۳۸ ۱۳۹۴/۰۴/۱۷

یک نکته‌ی تکمیلی

همین کتابخانه به صورت یک افزونه تحت عنوان [AutoMapper.EF6](#) در دسترس است.

فرض کنید کلاس‌های مدل برنامه از سه کلاس مشتری، سفارشات مشتری‌ها و اقلام هر سفارش تشکیل شده‌است:

```
public class Customer
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Bio { get; set; }

    public virtual ICollection<Order> Orders { get; set; }

    [Computed]
    [NotMapped]
    public string FullName
    {
        get { return FirstName + ' ' + LastName; }
    }
}

public class Order
{
    public int Id { get; set; }
    public string OrderNo { get; set; }
    public DateTime PurchaseDate { get; set; }
    public bool ShipToHomeAddress { get; set; }

    public virtual ICollection<OrderItem> OrderItems { get; set; }

    [ForeignKey("CustomerId")]
    public virtual Customer Customer { get; set; }
    public int CustomerId { get; set; }

    [Computed]
    [NotMapped]
    public decimal Total
    {
        get { return OrderItems.Sum(x => x.TotalPrice); }
    }
}

public class OrderItem
{
    public int Id { get; set; }
    public decimal Price { get; set; }
    public string Name { get; set; }
    public int Quantity { get; set; }

    [ForeignKey("OrderId")]
    public virtual Order Order { get; set; }
    public int OrderId { get; set; }

    [Computed]
    [NotMapped]
    public decimal TotalPrice
    {
        get { return Price * Quantity; }
    }
}
```

در اینجا برای پیاده سازی خواص محاسباتی، از نکته‌ی مطرح شده‌ی در مطلب « [نگاشت خواص محاسبه شده به کمک](#)

[DelegateDecompiler و AutoMapper](#) » استفاده شده‌است.

در ادامه می‌خواهیم اطلاعات حاصل از این کلاس‌ها را با شرایط خاصی به ViewModel‌های مشخصی جهت نمایش در برنامه نگاشت کنیم.

نمایش اطلاعات مشتری‌ها

می‌خواهیم اطلاعات مشتری‌ها را مطابق فرمت کلاس ذیل بازگشت دهیم:

```
public class CustomerViewModel
{
    public string Bio { get; set; }
    public string CustomerName { get; set; }
}
```

با این شرایط که

- اگر Bio نال بود، بجای آن N/A نمایش داده شود.

- CustomerName از خاصیت محاسباتی FullName کلاس مشتری تامین گردد.

برای حل این مساله، نیاز است نگاشت زیر را تهیه کنیم:

```
this.CreateMap<Customer, CustomerViewModel>()
    .ForMember(dest => dest.CustomerName, opt => opt.MapFrom(entity => entity.FullName))
    .ForMember(dest => dest.Bio, opt => opt.MapFrom(entity => entity.Bio ?? "N/A"));
```

AutoMapper برای جایگزین کردن خواص با مقدار نال، با یک مقدار مشخص، از متدی به نام [NullSubstitute](#) استفاده می‌کند. اما در این حالت خاص که قصد داریم از [Project To](#) استفاده کنیم، این روش پاسخ نمی‌دهد و [محدودیت‌هایی دارد](#). به همین جهت از روش map from و بررسی مقدار خاصیت، استفاده شده‌است. همچنین در اینجا مطابق نگاشت فوق، خاصیت CustomerName از خاصیت FullName کلاس مشتری دریافت می‌شود.

کوثری نهایی استفاده کننده‌ی از این اطلاعات به شکل زیر خواهد بود:

```
using (var context = new MyContext())
{
    var viewCustomers = context.Customers
        .Project()
        .To<CustomerViewModel>()
        .Decompile()
        .ToList();
    // don't use
    // var viewCustomers = Mapper.Map<IEnumerable<Customer>,
    IEnumerable<CustomerViewModel>>(dbCustomers);
    foreach (var customer in viewCustomers)
    {
        Console.WriteLine("{0} - {1}", customer.CustomerName, customer.Bio);
    }
}
```

در اینجا از متدهای Project To و همچنین Decompile استفاده شده‌است (جهت پردازش خاصیت محاسباتی).

نمایش اطلاعات سفارش‌های مشتری‌ها

در ادامه قصد داریم اطلاعات سفارش‌ها را با فرمت ViewModel ذیل نمایش دهیم:

```
public class OrderViewModel
{
    public string CustomerName { get; set; }
    public decimal Total { get; set; }
    public string OrderNumber { get; set; }
    public IEnumerable<OrderItemsViewModel> OrderItems { get; set; }
}

public class OrderItemsViewModel
{
    public string Name { get; set; }
    public int Quantity { get; set; }
    public decimal Price { get; set; }
}
```


با این شرایط که

- CustomerName از خاصیت محاسباتی FullName کلاس مشتری تامین گردد.

- خاصیت OrderNumber آن از خاصیت OrderNo تهیه گردد.

به همین جهت کار را با تهیه‌ی نگاشت ذیل ادامه می‌دهیم:

```
this.CreateMap<Order, OrderViewModel>()
    .ForMember(dest => dest.OrderNumber, opt => opt.MapFrom(src => src.OrderNo))
    .ForMember(dest => dest.CustomerName, opt => opt.MapFrom(src => src.Customer.FullName));
```

بر این اساس کوئری مورد استفاده نیز به نحو ذیل تشکیل می‌شود:

```
using (var context = new MyContext())
{
    var viewOrders = context.Orders
        .Project()
        .To<OrderViewModel>()
        .Decompile()
        .ToList();
    // don't use
    // var viewOrders = Mapper.Map<IEnumerable<Order>, IEnumerable<OrderViewModel>>(dbOrders);
    foreach (var order in viewOrders)
    {
        Console.WriteLine("{0} - {1} - {2}", order.OrderNumber, order.CustomerName, order.Total);
    }
}
```

در اینجا چون از خاصیت OrderItems کلاس ViewModel صرف‌نظر نشده‌است، اطلاعات آن نیز به همراه viewOrders موجود است. یعنی می‌توان چنین کوئری را نیز جهت نمایش اطلاعات تو در توی اقلام هر سفارش نیز نوشت:

```
using (var context = new MyContext())
{
    var viewOrders = context.Orders
        .Project()
        .To<OrderViewModel>()
        .Decompile()
        .ToList();
    // don't use
    // var viewOrders = Mapper.Map<IEnumerable<Order>, IEnumerable<OrderViewModel>>(dbOrders);
    foreach (var order in viewOrders)
    {
        Console.WriteLine("{0} - {1} - {2}", order.OrderNumber, order.CustomerName, order.Total);
        foreach (var item in order.OrderItems)
        {
            Console.WriteLine("{0} {1} - {2}", item.Quantity, item.Name, item.Price);
        }
    }
}
```

اگر می‌خواهید OrderItems به صورت خودکار واکشی نشود، نیاز است در نگاشت تهیه شده، توسط متد Ignore از آن صرف‌نظر کنید:

```
this.CreateMap<Order, OrderViewModel>()
    .ForMember(dest => dest.OrderNumber, opt => opt.MapFrom(src => src.OrderNo))
    .ForMember(dest => dest.OrderItems, opt => opt.Ignore())
    .ForMember(dest => dest.CustomerName, opt => opt.MapFrom(src => src.Customer.FullName));
```

نمایش اطلاعات یک سفارش، با فرمتی خاص

تا اینجا نگاشت‌های انجام شده بر روی لیستی از اشیاء صورت گرفتند. در ادامه می‌خواهیم اولین سفارش ثبت شده را با فرمت

ذیل نمایش دهیم:

```
public class OrderDateViewModel
{
    public int PurchaseHour { get; set; }
    public int PurchaseMinute { get; set; }
    public string CustomerName { get; set; }
}
```

به همین منظور ابتدا نگاشت ذیل را تهیه می‌کنیم:

```
this.CreateMap<Order, OrderDateViewModel>()
    .ForMember(dest => dest.PurchaseHour, opt => opt.MapFrom(src => src.PurchaseDate.Hour))
    .ForMember(dest => dest.PurchaseMinute, opt => opt.MapFrom(src => src.PurchaseDate.Minute))
    .ForMember(dest => dest.CustomerName, opt => opt.MapFrom(src => src.Customer.FullName));
```

در اینجا ساعت و دقیقه‌ی خرید، از خاصیت PurchaseDate استخراج شده‌اند. همچنین CustomerName نیز از خاصیت FullName کلاس مشتری دریافت گردیده‌است.

پس از این تنظیمات، کوئری نهایی به شکل ذیل خواهد بود:

```
using (var context = new MyContext())
{
    var viewOrder = context.Orders
        .Project()
        .To<OrderDateViewModel>()
        .Decompile()
        .FirstOrDefault();
    // don't use
    // var viewOrder = Mapper.Map<Order, OrderDateViewModel>(dbOrder);

    if (viewOrder != null)
    {
        Console.WriteLine("{0}, {1}:{2}", viewOrder.CustomerName, viewOrder.PurchaseHour,
            viewOrder.PurchaseMinute);
    }
}
```

فرمت کردن سفارشی اطلاعات در حین نگاشت‌ها

در مورد فرمت کننده‌های سفارشی و [تبدیلگرها](#) پیشتر بحث کرده‌ایم. اما اغلب آن‌ها را در حالت خاص LINQ to Entities نمی‌توان بکار برد، زیرا قابلیت تبدیل به SQL را ندارند. برای مثال فرض کنید می‌خواهیم خاصیت ShipToHomeAddress کلاس Order را به خاصیت ShipHome کلاس ذیل نگاشت کنیم:

```
public class OrderShipViewModel
{
    public string ShipHome { get; set; }
    public string CustomerName { get; set; }
}
```

با این شرط که اگر مقدار آن True بود، Yes را نمایش دهد. با توجه به ساختار مدنظر، نگاشت ذیل را می‌توان تهیه کرد که در آن فرمت کردن سفارشی، به متد MapFrom واگذار شده‌است:

```
this.CreateMap<Order, OrderShipViewModel>()
    .ForMember(dest => dest.ShipHome, opt => opt.MapFrom(src => src.ShipToHomeAddress? "Yes": "No"))
    .ForMember(dest => dest.CustomerName, opt => opt.MapFrom(src => src.Customer.FullName));
```

با این کوئری جهت استفاده‌ی از این تنظیمات:

```
using (var context = new MyContext())
```

```
{
    var viewOrders = context.Orders
        .Project()
        .To<OrderShipViewModel>()
        .Decompile()
        .ToList();
    // don't use
    // var viewOrders = Mapper.Map<IEnumerable<Order>, IEnumerable<OrderShipViewModel>>(dbOrders);
    foreach (var order in viewOrders)
    {
        Console.WriteLine("{0} - {1}", order.CustomerName, order.ShipHome);
    }
}
```

کدهای کامل این مطلب را [از اینجا](#) می‌توانید دریافت کنید.

در طی دو مطلب ([۱](#) و [۲](#)) با نحوه‌ی قرار دادن خواص محاسباتی، درون کلاس‌های مدل‌های بانک اطلاعاتی مورد استفاده‌ی توسط Entity Framework آشنا شدیم. در اینجا قصد داریم این خواص محاسباتی را از کلاس‌های اصلی مدل‌های بانک اطلاعاتی خود خارج و به ViewModel ها منتقل کنیم؛ چون اساسا هدف از این نوع خواص ویژه، ارائه اطلاعات نمایشی است به کاربر و نه ذخیره سازی آن‌ها در بانک اطلاعاتی.

مدل‌ها و تنظیمات برنامه

مدل‌ها و تنظیمات مورد استفاده‌ی در مثال جاری، با مدل‌های مطلب «[لغو Lazy Loading در حین کار با AutoMapper و Entity Framework](#)» یکی است. فقط ViewModel مورد استفاده اینبار یک چنین ساختاری را دارد:

```
public class UserViewModel
{
    public int Id { set; get; }
    public string CustomName { set; get; }
    public int PostsCount { set; get; }
}
```

در اینجا می‌خواهیم در حین نگاشت اطلاعات جدول کاربران بانک اطلاعاتی به UserViewModel :
 - خاصیت CustomName از جمع نام و سن شخص تشکیل شود.
 - خاصیت PostsCount بیانگر جمع مطالب ارسالی آن شخص باشد.

نگاشت‌های AutoMapper می‌توانند حاوی توابع تجمعی نیز باشند

برای حل مساله‌ی فوق تنها کافی است نگاشت ذیل را تهیه کنیم:

```
public class TestProfile : Profile
{
    protected override void Configure()
    {
        this.CreateMap<User, UserViewModel>()
            .ForMember(dest => dest.CustomName,
                opt => opt.MapFrom(src => src.Name + "[" + src.Age + "]"))
            .ForMember(dest => dest.PostsCount,
                opt => opt.MapFrom(src => src.BlogPosts.Count()));
    }

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

در این نگاشت عنوان شده‌است که اطلاعات CustomName را مطابق فرمول خاص جمع نام شخص و سن او تهیه کن. همچنین مقدار PostsCount، باید از جمع تعداد مطالب ارسالی او تشکیل شود.

کوئری نهایی استفاده کننده از تنظیمات نگاشت تهیه شده

در ادامه متدهای Project To جهت استفاده‌ی از تنظیمات نگاشت فوق بکار می‌گیریم:

```
using (var context = new MyContext())
{
    var user1 = context.Users
```

```

        .Project()
        .To<UserViewModel>()
        .FirstOrDefault();

    if (user1 != null)
    {
        Console.WriteLine(user1.CustomName);
        Console.WriteLine(user1.PostsCount);
    }
}

```

این کوئری یک چنین خروجی SQL ایی را به همراه دارد:

```

SELECT
    [Limit1].[Id] AS [Id],
    [Limit1].[C1] AS [C1],
    [Limit1].[C2] AS [C2]
FROM ( SELECT TOP (1)
    [Project1].[Id] AS [Id],
    CASE WHEN ([Project1].[Name] IS NULL) THEN N'' ELSE [Project1].[Name] END
    + N'[' + CAST( [Project1].[Age] AS nvarchar(max)) + N']' AS [C1],
    [Project1].[C1] AS [C2]
    FROM ( SELECT
        [Extent1].[Id] AS [Id],
        [Extent1].[Name] AS [Name],
        [Extent1].[Age] AS [Age],
        (SELECT
            COUNT(1) AS [A1]
            FROM [dbo].[BlogPosts] AS [Extent2]
            WHERE [Extent1].[Id] = [Extent2].[UserId]) AS [C1]
        FROM [dbo].[Users] AS [Extent1]
        ) AS [Project1]
    ) AS [Limit1]

```

همانطور که مشاهده می‌کنید، تنظیمات نگاشت تهیه شده (نحوه‌ی تهیه‌ی نام و جمع تعداد مطالب شخص) به SQL ترجمه شده‌اند.

کدهای کامل این مطلب را [از اینجا](#) می‌توانید دریافت کنید.

فرض کنید مدل کاربران سایت، دارای دو خاصیت راهبری (navigation properties) آدرس‌های مختلف یک کاربر و ایمیل‌های متفاوت او است:

```
public class SiteUser
{
    public int Id { get; set; }
    public string Name { get; set; }

    public virtual ICollection<Address> Addresses { get; set; }
    public virtual ICollection<Email> Emails { get; set; }
}

public class Email
{
    public int Id { get; set; }
    public string Text { get; set; }

    [ForeignKey("SiteUserId")]
    public virtual SiteUser SiteUser { get; set; }
    public int SiteUserId { get; set; }
}

public class Address
{
    public int Id { get; set; }
    public string Text { get; set; }

    [ForeignKey("SiteUserId")]
    public virtual SiteUser SiteUser { get; set; }
    public int SiteUserId { get; set; }
}
```

همچنین ViewModel ایی را هم که تعریف کرده‌ایم، شامل همان خواص راهبری مدل می‌شود:

```
public class UserViewModel
{
    public int Id { get; set; }
    public string Name { get; set; }

    public ICollection<Address> Addresses { get; set; }
    public ICollection<Email> Emails { get; set; }
}
```

در این حالت کوئری ذیل:

```
var user1 = context.Users.Project().To<UserViewModel>().FirstOrDefault();
```

سبب خواهد شد تا تمام خواص راهبری ذکر شده‌ی در ViewModel، در طی یک کوئری از بانک اطلاعاتی دریافت شده و مقدار دهی شوند. اما ... شاید در حین استفاده‌ی از آن، صرفاً به لیست ایمیل‌های شخص نیاز داشته باشیم و نیازی نباشد تا حتماً آدرس‌های او نیز واکشی شوند. برای حل این بارگذاری اضافی، می‌توان از تنظیم ExplicitExpansion استفاده کرد:

```
public class TestProfile : Profile
{
    protected override void Configure()
    {
        this.CreateMap<SiteUser, UserViewModel>()
            .ForMember(dest => dest.Addresses, opt => opt.ExplicitExpansion())
            .ForMember(dest => dest.Emails, opt => opt.ExplicitExpansion());
    }

    public override string ProfileName
    {

```

```
    get { return this.GetType().Name; }  
}
```

ExplicitExpansion به این معنا است که تا در کوئری مدنظر صریحا قید نشود که قرار است کدام خاصیت راهبری بسط یابد، اطلاعات آن از بانک اطلاعاتی دریافت نخواهد شد.

پس از تنظیم فوق، اگر کوئری ذکر شده را اجرا کنید، مشاهده خواهید کرد که دو خاصیت آدرس‌ها و ایمیل‌های شخص، نال هستند.

برای ذکر صریح خواص راهبری مورد نیاز، اینبار می‌توان از پارامترهای متد Project To مانند مثال ذیل استفاده کرد:

```
using (var context = new MyContext())  
{  
    var user1 = context.Users  
        .Project()  
        .To<UserViewModel>(parameters: new { }, membersToExpand: viewModel =>  
viewModel.Emails)  
        .FirstOrDefault();  
  
    if (user1 != null)  
    {  
        foreach (var email in user1.Emails)  
        {  
            Console.WriteLine(email.Text);  
        }  
    }  
}
```

این کوئری سبب خواهد شد تا صرفا خاصیت Emails از بانک اطلاعاتی واکشی شود و آدرس‌ها خیر. به این ترتیب می‌توان بر روی نحوه‌ی بارگذاری خواص راهبری کنترل کاملی داشت.

کدهای کامل این مطلب را [از اینجا](#) می‌توانید دریافت کنید.

نکته‌ی بسیار مهمی را که حین کار با AutoMapper باید بخاطر داشت، عدم thread safety متد **CreateMap** Mapper آن است و استفاده‌ی از آن در برنامه‌های چند ریسمانی و خصوصا برنامه‌های وب، مشکلات متعددی را به همراه خواهد داشت. بنابراین بهترین محل تعریف و معرفی این نگاشت‌ها، در حین آغاز برنامه‌است؛ برای مثال در متد `Application_Start` فایل `global.asax` برنامه‌های وب، یا ابتدای متد `Main` برنامه‌های دسکتاپ. برای نمونه یک چنین کدی را نباید در برنامه‌های خود داشته باشید:

```
public ActionResult Index()
{
    Mapper.CreateMap<UserViewModel, User>();
    //ادامه‌ی کدها
```

در اینجا از متد استاتیک **CreateMap** Mapper، در یک اکشن متد برنامه‌ی ASP.NET MVC استفاده شده‌است. این متد `thread safe` نیست و چون کار تنظیمات اولیه‌ی این نگاشت‌ها (پیش از کش شدن آن‌ها) اندکی زمانبر است، ممکن است در این بین، دو کاربر همزمان به این قطعه کد رسیده و شاهد این باشند که تعدادی از خواص در اینجا نگاشت نشده‌اند.

نمونه‌ی دیگر آن، یک چنین کدهایی هستند:

```
using (var context = new TestDbContext())
{
    Mapper.CreateMap<SourceClass, DestinationClass>()
        .AfterMap((src, dest) =>
        {
            //using context
        });

    var dest = Mapper.Map<DestinationClass>(source);
}
```

در اینجا برحسب نیاز از `context` مربوط به Entity framework داخل تنظیمات `Mapper.CreateMap` استفاده شده‌است. متد **CreateMap** Mapper استاتیک است و `context` استفاده شده‌ی در آن `thread safe` نیست. همینجا است که مشکلات تخریب اطلاعات را شاهد خواهید بود.

اگر در یک چنین حالتی نیاز به استفاده‌ی `context` داشتید، بهتر است متدهای استاتیک AutoMapper را فراموش کرده و به نحو ذیل یک موتور محلی نگاشت را ایجاد کنید. چون سطح دید و دسترسی این موتور، عمومی و سراسری نیست، مشکلات `thread safety` را نخواهد داشت.

```
var configurationStore = new ConfigurationStore(new TypeMapFactory(), MapperRegistry.Mappers);
configurationStore.AddProfile<TestProfile1>();
var mapper = new MappingEngine(configurationStore);
configurationStore.CreateMap<SourceClass, DestinationClass>()
//ادامه‌ی کدها
```


عموما مدل‌های ASP.NET MVC یک چنین شکلی را دارند:

```
public class UserModel
{
    public int Id { get; set; }

    [Required(ErrorMessage = "(*)")]
    [Display(Name = "نام")]
    [StringLength(maximumLength: 10, MinimumLength = 3, ErrorMessage = "نام باید حداقل 3 و حداکثر 10 حرف باشد")]
    public string FirstName { get; set; }

    [Required(ErrorMessage = "(*)")]
    [Display(Name = "نام خانوادگی")]
    [StringLength(maximumLength: 10, MinimumLength = 3, ErrorMessage = "نام خانوادگی باید حداقل 3 و حداکثر 10 حرف باشد")]
    public string LastName { get; set; }
}
```

و ViewModel مورد استفاده برای نمونه چنین ساختاری را دارد:

```
public class UserViewModel
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

مشکلی که در اینجا وجود دارد، نیاز به کپی و تکرار تک تک ویژگی‌های (Data Annotations/Attributes) خاصیت‌های مدل، به خواص مشابه آن‌ها در ViewModel است؛ از این جهت که می‌خواهیم برچسب خواص ViewModel، از ویژگی Display دریافت شوند و همچنین اعتبارسنجی‌های فیلدهای اجباری و بررسی حداقل و حداکثر طول فیلدها نیز حتما اعمال شوند (هم در سمت کاربر و هم در سمت سرور). در ادامه قصد داریم راه حلی را به کمک جایگزین سازی Provider های توکار ASP.NET MVC با نمونه‌ی سازگار با AutoMapper، ارائه دهیم، به نحوی که دیگر نیازی نباشد تا این ویژگی‌ها را در ViewModel ها تکرار کرد.

قسمت‌هایی از ASP.NET MVC که باید جهت انتقال خودکار ویژگی‌ها تعویض شوند

ASP.NET MVC به صورت توکار دارای یک ModelMetadataProviders.Current است که از آن جهت دریافت ویژگی‌های هر خاصیت استفاده می‌کند. می‌توان این تامین کننده‌ی ویژگی‌ها را به نحو ذیل سفارشی سازی نمود. در اینجا IConfigurationProvider همان Mapper.Engine.ConfigurationProvider مربوط به AutoMapper است. از آن جهت استخراج اطلاعات نگاشت‌های AutoMapper استفاده می‌کنیم. برای مثال کدام خاصیت Model به کدام خاصیت ViewModel نگاشت شده‌است. این کارها توسط متد الحاقی GetMappedAttributes انجام می‌شوند که در ادامه‌ی مطلب معرفی خواهد شد.

```
public class MappedMetadataProvider : DataAnnotationsModelMetadataProvider
{
    private readonly IConfigurationProvider _mapper;

    public MappedMetadataProvider(IConfigurationProvider mapper)
    {
        _mapper = mapper;
    }

    protected override ModelMetadata CreateMetadata(
        IEnumerable<Attribute> attributes,
        Type containerType,
        Func<object> modelAccessor,
        Type modelType,
        string propertyName)
    {
        // Implementation logic here
    }
}
```

```

{
    var mappedAttributes =
        containerType == null ?
            attributes :
            _mapper.GetMappedAttributes(containerType, propertyName, attributes.ToList());
    return base.CreateMetadata(mappedAttributes, containerType, modelAccessor, modelType,
propertyName);
}
}

```

شبهه به همین کار را باید برای ModelValidatorProviders.Providers نیز انجام داد. در اینجا یکی از تامین کننده‌های ModelValidator، از نوع DataAnnotationsModelValidatorProvider است که حتما نیاز است این مورد را نیز به نحو ذیل سفارشی سازی نمود. در غیراینصورت error messages موجود در ویژگی‌های تعریف شده، به صورت خودکار منتقل نخواهند شد.

```

public class MappedValidatorProvider : DataAnnotationsModelValidatorProvider
{
    private readonly IConfigurationProvider _mapper;

    public MappedValidatorProvider(IConfigurationProvider mapper)
    {
        _mapper = mapper;
    }

    protected override IEnumerable<ModelValidator> GetValidators(
        ModelMetadata metadata,
        ControllerContext context,
        IEnumerable<Attribute> attributes)
    {
        var mappedAttributes =
            metadata.ContainerType == null ?
                attributes :
                _mapper.GetMappedAttributes(metadata.ContainerType, metadata.PropertyName,
attributes.ToList());
        return base.GetValidators(metadata, context, mappedAttributes);
    }
}

```

و در اینجا پیاده سازی متد GetMappedAttributes را ملاحظه می‌کنید.

ASP.NET MVC هر زمانیکه قرار است توسط متدهای توکار خود مانند Html.TextBoxFor, Html.ValidationMessageFor، اطلاعات خاصیت‌ها را تبدیل به المان‌های HTML کند، از تامین کننده‌های فوق جهت دریافت اطلاعات ویژگی‌های مرتبط با هر خاصیت استفاده می‌کند. در اینجا فرصت داریم تا ویژگی‌های مدل را از تنظیمات AutoMapper دریافت کرده و سپس بجای ویژگی‌های خاصیت معادل ViewModel درخواست شده، بازگشت دهیم. به این ترتیب ASP.NET MVC تصور خواهد کرد که ViewModel ما نیز دقیقاً دارای همان ویژگی‌های Model است.

```

public static class AutoMapperExtensions
{
    public static IEnumerable<Attribute> GetMappedAttributes(
        this IConfigurationProvider mapper,
        Type viewModelType,
        string viewModelPropertyName,
        IList<Attribute> existingAttributes)
    {
        if (viewModelType != null)
        {
            foreach (var typeMap in mapper.GetAllTypeMaps().Where(i => i.DestinationType ==
viewModelType))
            {
                var propertyMaps = typeMap.GetPropertyMaps()
                    .Where(propertyMap => !propertyMap.IsIgnored() && propertyMap.SourceMember != null)
                    .Where(propertyMap => propertyMap.DestinationProperty.Name ==
viewModelPropertyName);

                foreach (var propertyMap in propertyMaps)
                {
                    foreach (Attribute attribute in propertyMap.SourceMember.GetCustomAttributes(true))
                    {
                        if (existingAttributes.All(i => i.GetType() != attribute.GetType()))
                        {
                            yield return attribute;
                        }
                    }
                }
            }
        }
    }
}

```

```

        }
    }
}

if (existingAttributes == null)
{
    yield break;
}

foreach (var attribute in existingAttributes)
{
    yield return attribute;
}
}
}

```

ثبت تامین کننده‌های سفارشی سازی شده توسط AutoMapper

پس از تهیه‌ی تامین کننده‌های انتقال ویژگی‌ها، اکنون نیاز است آن‌ها را به ASP.NET MVC معرفی کنیم:

```

protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();
    WebApiConfig.Register(GlobalConfiguration.Configuration);
    FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
    RouteConfig.RegisterRoutes(RouteTable.Routes);

    Mappings.RegisterMappings();
    ModelMetadataProviders.Current = new MappedMetadataProvider(Mapper.Engine.ConfigurationProvider);

    var modelValidatorProvider = ModelValidatorProviders.Providers
        .Single(provider => provider is DataAnnotationsModelValidatorProvider);
    ModelValidatorProviders.Providers.Remove(modelValidatorProvider);
    ModelValidatorProviders.Providers.Add(new
    MappedValidatorProvider(Mapper.Engine.ConfigurationProvider));
}

```

در اینجا `ModelMetadataProviders.Current` با `MappedMetadataProvider` جایگزین شده‌است. در قسمت کار با `ModelValidatorProviders.Providers`، ابتدا صرفاً همان تامین کننده‌ی از نوع `DataAnnotationsModelValidatorProvider` پیش فرض، یافت شده و حذف می‌شود. سپس تامین کننده‌ی سفارشی سازی شده‌ی خود را معرفی می‌کنیم تا جایگزین آن شود.

مثالی جهت آزمایش انتقال خودکار ویژگی‌های مدل به ViewModel

کنترلر مثال برنامه به شرح زیر است. در اینجا از متد `Mapper.Map` جهت تبدیل خودکار مدل کاربر به `ViewModel` آن استفاده شده‌است:

```

public class HomeController : Controller
{
    public ActionResult Index()
    {
        var model = new UserModel { FirstName = "و", Id = 1, LastName = "ن" };
        var viewModel = Mapper.Map<UserViewModel>(model);
        return View(viewModel);
    }

    [HttpPost]
    public ActionResult Index(UserViewModel data)
    {
        return View(data);
    }
}

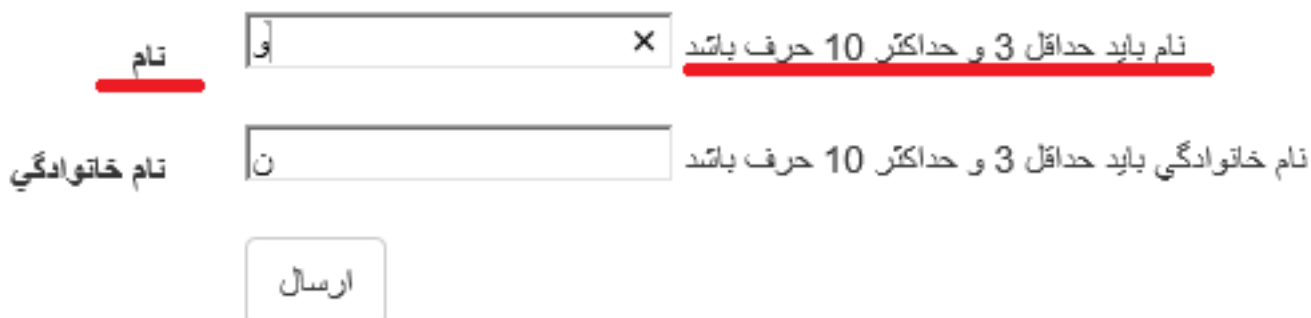
```

با این View که جهت ثبت اطلاعات مورد استفاده قرار می‌گیرد. این View، اطلاعات مدل خود را از ViewModel معرفی شده‌ی در ابتدای بحث دریافت می‌کند:

```
@model Sample12.ViewModels.UserViewModel

@using (Html.BeginForm("Index", "Home", FormMethod.Post, htmlAttributes: new { @class = "form-horizontal", role = "form" }))
{
    <div class="row">
        <div class="form-group">
            @Html.LabelFor(d => d.FirstName, htmlAttributes: new { @class = "col-md-2 control-label" })
            <div class="col-md-10">
                @Html.TextBoxFor(d => d.FirstName)
                @Html.ValidationMessageFor(d => d.FirstName)
            </div>
        </div>
        <div class="form-group">
            @Html.LabelFor(d => d.LastName, htmlAttributes: new { @class = "col-md-2 control-label" })
            <div class="col-md-10">
                @Html.TextBoxFor(d => d.LastName)
                @Html.ValidationMessageFor(d => d.LastName)
            </div>
        </div>
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="ارسال" class="btn btn-default" />
            </div>
        </div>
    </div>
}
```

در این حالت اگر برنامه را اجرا کنیم به شکل زیر خواهیم رسید:



در این شکل هر چند نوع مدل View مورد استفاده از ViewModel ایی تامین شده‌است که دارای هیچ ویژگی و Data Annotations/Attributes نیست، اما برچسب هر فیلد از ویژگی Display دریافت شده‌است. همچنین اعتبارسنجی سمت کاربر فعال بوده و برچسب‌های آن‌ها نیز به درستی دریافت شده‌اند.

کدهای کامل این مثال را [از اینجا](#) می‌توانید دریافت کنید .

نظرات خوانندگان

نویسنده: سیدمجتبی حسینی
تاریخ: ۱۳۹۴/۰۳/۱۷ ۱۲:۳۳

سلام

با توجه به بخش Other Notes در [این مطلب](#) استفاده همزمان از [انتقال خودکار Data Annotations](#) و [تزریق وابستگی‌های AutoMapper](#) در لایه سرویس برنامه چگونه است؟

متشکرم

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۳/۱۷ ۱۲:۳۹

به چه مشکلی برخوردید؟ کار نکرد؟ خطا داد؟ چه خطایی داد؟ چطور استفاده کردید؟

نویسنده: سیدمجتبی حسینی
تاریخ: ۱۳۹۴/۰۳/۱۷ ۱۲:۴۱

بخش تزریق وابستگی به خوبی کار میکند اما بخش انتقال خودکار Data Annotations عمل نمی‌کند و انتقال صورت نمی‌گیرد. علیرغم اینکه تمام بخش‌های آن اجرا می‌شود. توالی کدهای مربوط در global.asax بصورت زیر است:

```
setDbInitializer();
ModelMetadataProviders.Current = new MappedMetadataProvider(Mapper.Engine.ConfigurationProvider);
var modelValidatorProvider = ModelValidatorProviders.Providers
    .Single(provider => provider is DataAnnotationsModelValidatorProvider);
ModelValidatorProviders.Providers.Remove(modelValidatorProvider);
ModelValidatorProviders.Providers.Add(new
    MappedValidatorProvider(Mapper.Engine.ConfigurationProvider));
```

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۳/۱۷ ۱۲:۴۹

همینطور هست. علت آن را [در نظرات مطلب](#) تزریق وابستگی‌های AutoMapper توضیح دادم: «کاری که در اینجا انجام شده، ایجاد یک Mapping Engine سفارشی هست که با Mapping Engine اصلی استاتیک یکی نیست. به همین جهت برای نمونه متد Project آرگومان (`_mappingEngine`) هم دارد. اگر قید نشود، یعنی قرار است از موتور نگاشت استاتیک سراسری پیش فرض آن استفاده شود.» در مثال فوق هم `Mapper.Engine.ConfigurationProvider` از همان موتور نگاشت استاتیک سراسری استفاده شده است (در متد `Application_Start` برنامه). این مورد را باید با یک وهله‌ی `IConfigurationProvider` تامین شده‌ی توسط `IoC Container` جایگزین کنید؛ مثلاً:

```
SmObjectFactory.Container.GetInstance<IConfigurationProvider>()
```

نویسنده: سیدمجتبی حسینی
تاریخ: ۱۳۹۴/۰۳/۱۷ ۱۲:۵۵

متشکرم مشکل حل شد.

استفاده از این وهله در `Application_Start` مشکل ساز نیست؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۳/۱۷ ۱۲:۵۷

IConfigurationProvider وابستگی به ASP.NET ندارد و همچنین طول عمر ConfigurationStore آن هم Singleton است و یکبار که ایجاد شد، کش می‌شود.

نویسنده: کمال حمیدی
تاریخ: ۱۳۹۴/۰۴/۲۶ ۱۳:۴۰

آیا از این روش همیشه تمام Data Annotations های مدل رو برای ViewModel فرستاد؟
مثلا من توی مدل ام از ویژگی AdditionalMetadata استفاده کردم و توی View هم از کد زیر برای نمایش اطلاعات آنها استفاده میکنم.

```
@ModelMetadata.FromLambdaExpression(x => x.Name, ViewData).AdditionalValues["HelpTag"]
```

اما خطای زیر ارسال میشه:
The given key was not present in the dictionary

متد Project To را می‌توان به عنوان متد پیش فرض حین کار با ORM‌ها در نظر گرفت؛ با این مزایا:

- جلوگیری از Lazy loading اشتباه

- کاهش تعداد فیلدهای بازگشت داده شده‌ی از دیتابیس و محدود ساختن آن‌ها به خواصی که قرار است نگاشت شوند. در حالت معمولی استفاده‌ی از متد Mapper.Map، تمام فیلدهای مدل بارگذاری شده و سپس در سمت کلاینت توسط AutoMapper نگاشت خواهند شد. اما در حالت استفاده‌ی از متد ویژه‌ی Project To، کوئری SQL ارسالی به بانک اطلاعاتی نیز مطابق نگاشت تعریف شده، تغییر کرده و خلاصه خواهد شد.

در این حالت یک چنین سناریویی را در نظر بگیرید. مدل متناظر با جدول بانک اطلاعاتی ما چنین ساختاری را دارد:

```
public class UserModel
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

و اطلاعاتی که قرار است در رابط کاربری نمایش داده شوند، به این شکل تعریف شده‌اند:

```
public class UserViewModel
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string UserIdentityName { get; set; }
}
```

در اینجا خاصیت UserIdentityName قرار است در زمان اجرا، برای مثال توسط مقدار User.Identity.Name تأمین شود و در حالت کلی، خاصیت یا خاصیت‌های ثابتی را داریم که نیاز است در حین نگاشت انجام شده، در زمان اجرا مقدار ثابت خود را دریافت کنند.

تعریف نگاشت‌های پارامتری

برای حل این مساله، از روش زیر استفاده می‌شود:

```
string userIdentityName = null;
this.CreateMap<UserModel, UserViewModel>()
    .ForMember(d => d.UserIdentityName, opt => opt.MapFrom(src => userIdentityName));
```

ابتدا یک متغیر خالی را تعریف می‌کنیم. از آن جهت تهیه‌ی یک lambda expression صحیح در قسمت MapFrom استفاده خواهیم کرد. کار این متغیر خالی، تهیه‌ی یک عبارت جایگزین شونده‌ی در زمان اجرا است. اکنون جهت استفاده‌ی از این متغیر با قابلیت جایگزینی، می‌توان به نحو ذیل عمل کرد:

```
var uiUsers = users.AsQueryable()
    .Project()
    .To<UserViewModel>(new { userIdentityName = "User.Identity.Name Value Here" })
    .ToList();
```

در اینجا لیست کاربران بانک اطلاعاتی، به لیست UserViewModel‌ها نگاشت شده و همچنین مقدار خاصیت UserIdentityName آن‌ها نیز از پارامتری که به متد Project To ارسال گردیده‌است، تأمین خواهد شد.

کدهای کامل این مثال را [از اینجا](#) می‌توانید دریافت کنید.