

با توجه به اصل **Dry** تا می توان باید از نوشتن کدهای تکراری خودداری کرد و کدها را تا جایی که ممکن است به قسمت هایی با قابلیت استفاده ی مجدد تبدیل کرد. حین کار کردن با ORM های معروف مثل NHibernate و EntityFramework زمان زیادی نوشتن کوثری ها جهت واکنشی داده ها از دیتابیس صرف می شود. اگر بتوان کوثری هایی با قابلیت استفاده ی مجدد نوشت علاوه بر کاهش زمان توسعه قابلیت هایی قدرتمندی مانند زنجیر کردن کوثری ها به دنبال هم به دست می آید. با یک مثال نحوه ی نوشتن و مزایای کوثری با قابلیت استفاده ی مجدد را بررسی می کنیم :

برای مثال دو جدول شهرها و دانش آموزان را در نظر بگیرید:

```
namespace ReUsableQueries.Model
{
    public class Student
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string LastName { get; set; }
        public int Age { get; set; }
        [ForeignKey("BornInCityId")]
        public virtual City BornInCity { get; set; }
        public int BornInCityId { get; set; }
    }

    public class City
    {
        public int Id { get; set; }
        public string Name { get; set; }

        public virtual ICollection<Student> Students { get; set; }
    }
}
```

در ادامه این کلاس ها را در معرض دید EF Code first قرار داده:

```
using System.Data.Entity;
using ReUsableQueries.Model;

namespace ReUsableQueries.DAL
{
    public class MyContext : DbContext
    {
        public DbSet<City> Cities { get; set; }
        public DbSet<Student> Students { get; set; }
    }
}
```

و همچنین تعدادی رکورد آغازین را نیز به جداول مرتبط اضافه می کنیم:

```
public class Configuration : DbMigrationsConfiguration<MyContext>
{
    public Configuration()
    {
        AutomaticMigrationsEnabled = true;
        AutomaticMigrationDataLossAllowed = true;
    }
    protected override void Seed(MyContext context)
    {
        var city1 = new City { Name = "city-1" };
        var city2 = new City { Name = "city-2" };
        context.Cities.Add(city1);
        context.Cities.Add(city2);
        var student1 = new Student() { Name = "Shaahin", LastName = "Kiassat", Age=22, BornInCity =
city1};
        var student2 = new Student() { Name = "Mehdi", LastName = "Farzad", Age = 31, BornInCity =
```

```
city1 };
= city2 };
    var student3 = new Student() { Name = "James", LastName = "Hetfield", Age = 49, BornInCity
    context.Students.Add(student1);
    context.Students.Add(student2);
    context.Students.Add(student3);
    base.Seed(context);
    }
}
```

فرض کنید قرار است یک کوئری نوشته شود که در جدول دانش آموزان بر اساس نام ، نام خانوادگی و سن جستجو کند :

```
var context = new MyContext();
var query= context.Students.Where(x => x.Name.Contains(name)).Where(x =>
x.LastName.Contains(lastName)).Where(
    x => x.Age == age);
```

احتمالا هنوز کسانی هستند که فکر می کنند کوئری های LINQ همان لحظه که تعریف می شوند اجرا می شوند [اما اینگونه نیست](#) . در واقع این کوئری فقط یک Expression از رکوردهای جستجو شده است و تا زمانی که متد ToArray یا ToList روی آن اجرا نشود هیچ داده ای برگردانده نمی شود.

در یک برنامه ی واقعی داده های باید به صورت صفحه بندی شده و مرتب شده برگردانده شود پس کوئری به این صورت خواهد بود :

```
var query= context.Students.Where(x => x.Name.Contains(name)).Where(x =>
x.LastName.Contains(lastName)).Where(
    x => x.Age == age).OrderBy(x=>x.LastName).Skip(skip).Take(take);
```

ممکن است بخواهیم در متد دیگری در لیست دانش آموزان بر اساس نام ، نام خانوادگی ، سن و شهر جستجو کنیم و سپس خروجی را اینبار بر اساس سن مرتب کرده و صفحه بندی نکنیم:

```
var query = context.Students.Where(x => x.Name.Contains(name)).Where(x =>
x.LastName.Contains(lastName)).Where(
    (
        x => x.Age == age).Where(x => x.BornInCityId == 1).OrderBy(x => x.Age);
```

همانطور که می بینید قسمت هایی از این کوئری با کوئری هایی که قبلا نوشتیم یکی است ، همچنین حتی ممکن است در قسمت دیگری از برنامه نتیجه ی همین کوئری را به صورت صفحه بندی شده لازم داشته باشیم.

اکنون نوشتن این کوئری ها میان کد های Business Logic باعث شده هیچ استفاده ی مجددی نتوانیم از این کوئری ها داشته باشیم. حال بررسی می کنیم که چگونه می توان کوئری هایی با قابلیت استفاده ی مجدد نوشت :

```
namespace ReUsableQueries.Queries
{
    public static class StudentQueryExtension
    {
        public static IQueryable<Student> FindStudentsByName(this IQueryable<Student> students, string
name)
        {
            return students.Where(x => x.Name.Contains(name));
        }
        public static IQueryable<Student> FindStudentsByLastName(this IQueryable<Student> students,
string lastName)
        {
            return students.Where(x => x.LastName.Contains(lastName));
        }
        public static IQueryable<Student> SkipAndTake(this IQueryable<Student> students, int skip , int
take)
        {
            return students.Skip(skip).Take(take);
        }
        public static IQueryable<Student> OrderByAge(this IQueryable<Student> students)
```

```

    {
        return students.OrderBy(x=>x.Age);
    }
}

```

همان طور که مشاهده می کنید به کمک متدهای الحاقی برای شیء `IQueryable<Student>` چند کوئری نوشته ایم . اکنون در محل استفاده از کوئری ها می توان این کوئری ها را به راحتی به هم زنجیر کرد. همچنین اگر روزی قرار شد منطق یکی از کوئری ها عوض شود با عوض کردن آن در یک قسمت برنامه همه جا اعمال می شود. نحوه ی استفاده از این متدهای الحاقی به این صورت خواهد بود :

```

var query =
context.Students.FindStudentsByName(name).FindStudentsByLastName(lastName).SkipAndTake(skip,take);

```

فرض کنید قرار است یک سیستم جستجوی پیشرفته به برنامه اضافه شود که بر اساس شرطهای مختلف باید یک شرط در کوئری اعمال شود یا نشود ، به کمک این طراحی جدید به راحتی می توان بر اساس شرطهای مختلف یک کوئری را اعمال کرد یا نکرد :

```

var query = context.Students.AsQueryable();
if (searchByName)
{
    query= query.FindStudentsByName(name);
}
if (orderByAge)
{
    query = query.OrderByAge();
}
if (paging)
{
    query = query.SkipAndTake(skip, take);
}
return query.ToList();

```

همچنین این کوئری ها وابسته به ORM خاصی نیستند البته این نکته هم مد نظر است که LINQ Provider بعضی ORM ها ممکن است بعضی کوئری ها را پشتیبانی نکند.

نظرات خوانندگان

نویسنده: محمد باقر سیف الهی
تاریخ: ۲۲:۲۳ ۱۳۹۱/۰۸/۱۸

ممنون از مطلب خوبتون... می خواستم بدونم اگه بخوام این متدها رو (در کلاس StudentQueryExtension) جوری بنویسم که با Anonymous Type هم قابل استفاده باشه چه راه حلی وجود داره؟ (یعنی تمام ستون ها رو برنگردونم و فقط اونهایی رو که نیاز دارم نمایش بدم و این اعلام نیاز بتونه داینامیک باشه و از طریق پارامتر به تابع پاس داده بشه یا چیزی شبیه این!) . نوع خروجی متدها بهتره چجوری نوشته بشن؟

نویسنده: شاهین کیاست
تاریخ: ۲۲:۴۱ ۱۳۹۱/۰۸/۱۸

خواهش می کنم.
با توجه به این که متدهای الحاقی برای

IQueryable<Entity>

نوشته شده اند پس نوع خروجی هم باید از همین نوع باشد ، راه حلی که به نظر می آید اینه که برای برگرداندن چند ستون نوع برگشتی را از نوع یک CustomObject بگذارید مثلا StudentDTO در مورد داینامیک بودن نمی دانم چه کار باید کرد اما برای خودم هم جالب هست که آیا میشه این کار رو کرد یا خیر .

نویسنده: وحید نصیری
تاریخ: ۱:۲۵ ۱۳۹۱/۰۸/۱۹

- هیچ تغییری را در متدهای الحاقی همه منظوره ایجاد نکنید. این متدها رکوردی رو بر نمی گردونند (در متن لینک داده شده). فقط یک سری عبارت هستند. Select نهایی ویژه را پیش از ToList آخر کار انجام بدید.
- برای پویا کردن LINQ امکان استفاده از رشته ها وجود داره: (^)
- نوع خروجی متد در این حالت خاص می تونه object یا IEnumerable خالی باشد.

نویسنده: محسن د.
تاریخ: ۱:۴۷ ۱۳۹۱/۰۸/۱۹

اول تشکر می کنم بابت مطلب خوبتون ..
اگر سوال جناب سیف الهی رو درست متوجه شده باشم ، ایشون می خوان که فیلدهایی رو که از یک تابع برگشت داده میشه خودشون انتخاب کنن و محدود به مقدار بازگشتی از نوع Student برای مثال نباشن .
ایده ای که به ذهن من رسید (بر اساس برداشتی که از سوال داشتم) استفاده از قابلیت بسیار کاربردی Func هستش . یک Func با ورودی از نوع Entity و مقدار بازگشتی از نوع anonymous Type . در هنگام فراخوانی هم میشه از نوع dynamic برای دریافت نتیجه استفاده کرد . یک نمونه از پیاده سازی همچین چیزی رو [اینجا](#) قرار دادم .

نویسنده: شاهین کیاست
تاریخ: ۲:۱۸ ۱۳۹۱/۰۸/۱۹

ممنونم.
نمونه کد خیلی خوبی بود تشکر.

نویسنده: ابراهیم
تاریخ: ۱۱:۴۶ ۱۳۹۱/۰۸/۱۹

سلام. ممنون از مطلب خوبتان. می‌خواستم نظرتان را در رابطه با الگوی [Repository](#) بدانم، به نظر من این الگو با اینکه محبوبیت زیادی هم پیدا کرده ولی به پیچیدگی نالازمی نسبت به روش شما دارد. سوالی نیز داشتم، امکان نداشت به شیوه‌ای از IEnumerable به جای IQueryable استفاده شود؟ به نظر من مزیت آن در این است که بتوان خارج از چارچوب ORM از این کوئری‌ها استفاده شود و برای آن‌ها تست ایجاد نمود.
باز هم ممنون

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۸/۱۹ ۱۲:۱

- مطلب جاری نفی کننده وجود لایه سرویس در برنامه نیست و مکمل آن است.
- پیاده سازی الگوی مخزنی را که لینک دادید اشتباه است. دلایل اشتباه بودن آن را در این مطلب مطالعه کنید: ([^](#))

نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۸/۱۹ ۱۲:۲

سلام ؛ [استفاده از الگوی Repository اضافی در EF Code first؛ آری یا خیر؟!](#)

لطفا مطلب [تفاوت‌های IQueryable و IEnumerable](#) را مطالعه بفرمایید.
اگر از IEnumerable استفاده شود دیگر نمی‌توان کوئری‌ها را به هم زنجیر کرد .

نویسنده: محمد باقر سیف الهی
تاریخ: ۱۳۹۱/۰۸/۲۰ ۹:۴

بسیار ممنون از تمام دوستان...

نویسنده: امیر هاشم زاده
تاریخ: ۱۳۹۱/۰۸/۲۰ ۱۷:۹

در قسمت زنجیر کردن کوئری‌ها نباید

```
var query =  
context.Students.FindStudentsByName(name).FindStudentsByLastName(lastName).SkipAndTake(skip,take);
```

به

```
var query =  
context.Students.AsQueryable().FindStudentsByName(name).FindStudentsByLastName(lastName).SkipAndTake(sk  
ip,take);
```

تغییر کند؟! اگر جواب منفی است چرا؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۸/۲۰ ۱۸:۴۷

نیازی نیست چون DbSet از یک سری کلاس منجمله IQueryable مشتق می‌شود.

نویسنده: کیا
تاریخ: ۱۳۹۱/۰۸/۲۱ ۹:۱۲

برای حالتی که بخواین بصورت دینامیک و Anonymous ستونها رو پاس بدین می‌تونین بصورت زیر عمل کنین. در سمت سرویس :

```
public IEnumerable<dynamic> GetCustomColumnsDynamic(Func<Student, dynamic> pColumns)
{
    return _entities.Select(pColumns).ToList();
}
```

و برای استفاده :

```
var resultDynamic = _serviceStudent.GetCustomColumnsDynamic
(
    x=> new { x.Id, x.LastName, x.Age }
);
MessageBox.Show(resultDynamic.LastName);
```

و البته همونطور که می‌دونین چون نتیجه بصورت dynamic در اختیار شما قرار می‌گیره از امکانات کامپایلر بی نصیب هستید

نویسنده: محمد جواد تواضعی
تاریخ: ۱۷:۳۰ ۱۳۹۱/۰۸/۲۹

سلام

شاهین جان بابت مطلب بسیار عالی بود.

می‌خواستم نظرت در مورد اینکه برای گرفتن کوئری با قابلیت مجدد از این روش استفاده بشود چیست ؟ [Expression tree](#)

و برای کوئری با قابلیت مجدد کدام روش بهینه‌تر می‌باشد ؟

نویسنده: کوروش شاهی
تاریخ: ۱۳:۱۷ ۱۳۹۳/۰۲/۲۸

با توجه به مطلبی که در مبحث « [تفاوت بین IQueryable و IEnumerable در حین کار با ORMs](#) » بیان شده، خروجی متد یا باید List و یا باید IEnumerable باشد ؟
اگه مثالی هم بیان بشه این مهم بیشتر قابل درک است و یا لینکی که با مثال این رو توضیح داده باشه.
متشکر.

نویسنده: محسن خان
تاریخ: ۱۳:۳۴ ۱۳۹۳/۰۲/۲۸

بستگی داره در چه لایه‌ای کار می‌کنید و این خروجی قراره در چه لایه‌ای استفاده بشه. خروجی لایه سرویس قراره در لایه UI نمایش داده بشه؟ خروجی لایه سرویس نباید IQueryable باشه. داخل لایه سرویس می‌خواهید کوئری‌ها را با هم ترکیب کنید؟ باید IQueryable باشه.

نویسنده: کوروش شاهی
تاریخ: ۱۵:۱۳ ۱۳۹۳/۰۲/۲۸

با توجه به موارد و بستگی هایی که بیان کردین، فقط در لایه سرویس(بیزینس) باید IQueryable بودن یا نبودن خروجی متد رو مشخص کنیم و یا همچنین در لایه Repository یا همون DAL هم باید این موارد رو در نظر بگیریم ؟
با تشکر.

نویسنده: کوروش شاهی
تاریخ: ۱۶:۵۷ ۱۳۹۳/۰۲/۲۹

اگر منبع معتبری هم باشه که این موارد رو در قبال مثال توضیح داده باشه، میتونه خیلی بیشتر مثر ثمر واقع بشه. من خیلی گوگل کردم ولی روش ها بسیار متنوع بود و آدم سردرگم میشه بیشتر. متشکرم.

نویسنده: شاهین کیاست
تاریخ: ۱۷:۳۰ ۱۳۹۳/۰۲/۲۹

من یک دور بازخوردهای شما را خواندم اما متوجه موردی که برای شما ابهام ایجاد کرده نشدم. آیا شما از Entity Framework استفاده می کنید؟ اگر پاسخ مثبت است، خود EF لایه ی Repository را پیاده سازی کرده است، و این پیاده سازی یک IQueryable جهت انجام Query های متفاوت در اختیار شما قرار می دهد. شما می توانید مستقیما از DbContext سمت لایه ی سرویس استفاده کنید و داده ها را جهت استفاده برای استفاده کننده ی لایه ی سرویس فراهم کنید. لایه ی سرویس باید داده ها را درون حافظه برگرداند، نه اینکه یک IQueryable برگرداند که استفاده کننده آن را اجرا کند. از Repository در لایه ی سرویس استفاده کنید.

در دنیای دات نت گرایشی برای تجزیه (abstract) کردن EF پشت الگوی Repository وجود دارد. این تمایل اساسا بد است و در ادامه سعی می‌کنم چرای آن را توضیح دهم.

پایه و اساس

عموما این باور وجود دارد که با استفاده از الگوی Repository می‌توانید (در مجموع) دسترسی به داده‌ها را از لایه دامنه (Domain) تفکیک کنید و "داده‌ها را بصورت سازگار و استوار عرضه کنید".

اگر به هر کدام از پیاده سازی‌های الگوی Repository در کنار UnitOfWork (EF) دقت کنید خواهید دید که تفکیک (decoupling) قابل ملاحظه ای وجود ندارد.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Data;
using ContosoUniversity.Models;

namespace ContosoUniversity.DAL
{
    public class StudentRepository : IStudentRepository, IDisposable
    {
        private SchoolContext context;

        public StudentRepository(SchoolContext context)
        {
            this.context = context;
        }

        public IEnumerable<Student> GetStudents()
        {
            return context.Students.ToList();
        }

        public Student GetStudentByID(int id)
        {
            return context.Students.Find(id);
        }

        // <snip>
        public void Save()
        {
            context.SaveChanges();
        }
    }
}
```

این کلاس بدون SchoolContext نمی‌تواند وجود داشته باشد، پس دقیقا چه چیزی را در اینجا decouple کردیم؟ **هیچ چیز را!!**

در این قطعه کد - از MSDN - چیزی که داریم یک پیاده سازی مجدد از LINQ است که مشکل کلاسیک API Repository های بی انتها را بدست می‌دهد. منظور از API Repository های بی انتها، متدهای جالبی مانند GetStudentById, GetStudentByBirthday, GetStudentByOrderNumber و غیره است.

اما این مشکل اساسی نیست. مشکل اصلی روتین Save() است. این متد یک دانش آموز (Student) را ذخیره می‌کند .. اینطور بنظر می‌رسد. دیگر چه چیزی را ذخیره می‌کند؟ آیا می‌توانید حدس بزنید؟ من که نمی‌توانم .. بیشتر در ادامه.

UnitOfWork تراکنشی است یک UnitOfWork همانطور که از نامش بر می‌آید برای **انجام کاری** وجود دارد. این کار می‌تواند به

سادگی واکنشی اطلاعات و نمایش آنها، و یا به پیچیدگی پردازش یک سفارش جدید باشد. هنگامی که شما از EntityFramework استفاده می‌کنید و یک DbContext را و هله سازی می‌کنید، در واقع یک UnitOfWork می‌سازید.

در EF می‌توانید با فراخوانی SubmitChanges() تمام تغییرات را فلاش کرده و بازنشانی کنید (flush and reset). این کار بیت‌های مقایسه change tracker را تغییر می‌دهد. افزودن رکوردهای جدید، بروز رسانی و حذف آنها. هر چیزی که تعیین کرده باشید. و تمام این دستورات در یک تراکنش یا Transaction انجام می‌شوند.

یک Repository مطلقاً یک UnitOfWork نیست

هر متد در یک Repository قرار است فرمانی اتمی (Atomic) باشد - چه واکنشی اطلاعات و چه ذخیره آنها. مثلاً می‌توانید یک Repository داشته باشید با نام SalesRepository که اطلاعات کاتالوگ شما را واکنشی می‌کند، و یا یک سفارش جدید را ثبت می‌کند. منظور از فرمان‌های اتمیک این است، که هر متد تنها یک دستور را باید اجرا کند. تراکنشی وجود ندارد و امکاناتی مانند ردیابی تغییرات و غیره هم جایی ندارند.

یکی دیگر از مشکلات استفاده از Repository ها این است که بزودی و به آسانی از کنترل خارج می‌شوند و نیاز به ارجاع دیگر مخازن پیدا می‌کنند. به دلیل اینکه مثلاً نمی‌دانستید که SalesRepository نیاز به ارجاع ReportRepository داشته است (یا چیزی مانند این).

این مشکل به سرعت مشکل ساز می‌شود، و نیز به همین دلیل است که به UnitOfWork تمایل پیدا می‌کنیم.

بدترین کاری که می‌توانید انجام دهید: <T>Repository این الگو دیوانه وار است. این کار عملاً انتزاعی از یک انتزاع دیگر است (abstraction of an abstraction). به قطعه کد زیر دقت کنید، که به دلیلی نامشخص بسیار هم محبوب است.

```
public class CustomerRepository : Repository < Customer > {
    public CustomerRepository(DbContext context){
        //a property on the base class
        this.DB = context;
    }

    //base class has Add/Save/Remove/Get/Fetch
}
```

در نگاه اول شاید بگویید مشکل این کلاس چیست؟ همه چیز را کپسوله می‌کند و کلاس پایه Repository هم به کانتکست دسترسی دارد. پس مشکل کجاست؟

مشکلات عدیده اند .. بگذارید نگاهی بیاندازیم.

آیا می‌دانید این DbContext از کجا آمده است؟

خیر، نمی‌دانید. این آبجکت به کلاس تزریق (Inject) می‌شود، و نمی‌دانید که چه متدی آن را باز کرده و به چه دلیلی. ایده اصلی پشت الگوی Repository استفاده مجدد از کد است. بدین منظور که مثلاً برای عملیات CRUD از کلاسی پایه استفاده کنید تا برای هر موجودیت و فرمی نیاز به کدنویسی مجدد نباشد. برگ برنده این الگو نیز دقیقاً همین است. مثلاً اگر بخواهید از کدی در چند فرم مختلف استفاده کنید از این الگو استفاده میشد.

الگوی UnitOfWork همه چیز در نامش مشخص است. اگر قرار باشد آنرا بدین شکل تزریق کنید، نمی‌توانید بدانید که از کجا آمده است.

شناسه مشتری جدید را نیاز داشتیم

کد بالا در CustomerRepository را در نظر بگیرید - که یک مشتری جدید را به دیتابیس اضافه می‌کند. اما CustomerID جدید چه می‌شود؟ مثلاً به این شناسه نیاز دارید تا یک log بسازید. چه می‌کنید؟ گزینه‌های شما اینها هستند:

متد SubmitChanges() را صدا بزنید تا تغییرات ثبت شوند و بتوانید به CustomerID جدید دسترسی پیدا کنید. CustomerRepository خود را باز کنید و متد پایه Add را بازنویسی (override) کنید. بدین منظور که پیش از بازگشت دادن، متد SubmitChanges() را فراخوانی کند. این راه حلی است که MSDN به آن تشویق می‌کند، و بمبی ساعتی است که در انتظار انفجار است.

تصمیم بگیرید که تمام متدهای Add/Remove/Save در مخازن شما باید SubmitChanges() را فراخوانی کنند.

مشکل را می‌بینید؟ مشکل در خود پیاده سازی است. در نظر بگیرید که چرا New Customer ID را نیاز دارید؟ احتمالا برای استفاده از آن در ثبت یک سفارش جدید، و یا ثبت یک ActivityLog.

اگر بخواهیم از StudentRepository بالا برای ایجاد دانش آموزان جدید پس از خرید آنها از فروشگاه کتاب مان استفاده کنیم چه؟ اگر DbContext خود را به مخزن تزریق کنید و دانش آموز جدید را ذخیره کنید .. اوه .. تمام تراکنش شما فلاش شده و از بین رفته!

حالا گزینه‌های شما اینها هستند: 1) از StudentRepository استفاده نکنید (از OrderRepository یا چیز دیگری استفاده کنید). و یا 2) فراخوانی SubmitChanges() را حذف کنید و به باگ‌های متعددی اجازه ورود به کد تان را بدهید.

اگر تصمیم بگیرید که از StudentRepository استفاده نکنید، حالا کدهای تکراری (duplicate) خواهید داشت.

شاید بگویید که برای دستیابی به شناسه رکورد جدید نیازی به SubmitChanges() نیست، چرا که خود EF این عملیات را در قالب یک تراکنش انجام می‌دهد!

دقیقا درست است، و نکته من نیز همین است. در ادامه به این قسمت باز خواهیم گشت.

متدهای Repositories قرار است اتمیک باشند

به هر حال تئوری اش که چنین است. چیزی که در Repository ها داریم حتی اصلا Repository هم نیست. بلکه یک abstraction برای عملیات CRUD است که هیچ کاری مربوط به منطق تجاری اپلیکیشن را هم انجام نمی‌دهد. مخازن قرار است روی دستورات مشخصی تمرکز کنند (مثلا ثبت یک رکورد یا واکنشی لیستی از اطلاعات)، اما این مثال‌ها چنین نیستند.

همانطور که گفته شده استفاده از چنین رویکردهایی به سرعت مشکل ساز می‌شوند و با رشد اپلیکیشن شما نیز مشکلات عدیده ای برایتان بوجود می‌آروند.

خوب، راه حل چیست؟

برای جلوگیری از این abstraction های غیر منطقی دو راه وجود دارد. اولین راه استفاده از Command/Query Separation است که ممکن است در ابتدا کمی عجیب و بنظر برسند اما لازم نیست کاملا CQRS را دنبال کنید. تنها از سادگی انجام کاری که مورد نیاز است لذت ببرید، و نه بیشتر.

آبجکت‌های Command/Query

Jimmy Bogard مطلب خوبی در اینباره نوشته است و با تغییراتی جزئی برای بکارگیری Properties کدی مانند لیست زیر خواهیم داشت. مثلا برای مطالعه بیشتر درباره آبجکت‌های Command/Query به [این لینک](#) سری بزنید.

```
public class TransactOrderCommand {
    public Customer NewCustomer {get;set;}
    public Customer ExistingCustomer {get;set;}
    public List<Product> Cart {get;set;}
    //all the parameters we need, as properties...
    //...

    //our UnitOfWork
    StoreContext _context;
    public TransactOrderCommand(StoreContext context){
        //allow it to be injected - though that's only for testing
    }
}
```

```

    _context = context;
}

public Order Execute(){
    //allow for mocking and passing in... otherwise new it up
    _context = _context ?? new StoreContext();

    //add products to a new order, assign the customer, etc
    //then...
    _context.SubmitChanges();

    return newOrder;
}
}

```

همین کار را با یک آجکت Query نیز می‌توانید انجام دهید. می‌توانید پست Jimmy را بیشتر مطالعه کنید، اما ایده اصلی این است که آجکت‌های Query و Command برای دلیل مشخصی وجود دارند. می‌توانید آجکت‌ها را در صورت نیاز تغییر دهید و یا mock کنید.

DataContext خود را در آغوش بگیرید ایده ای که در ادامه خواهید دید را شخصا بسیار می‌پسندم (که توسط [Ayende](#) معرفی شد). چیزهایی که به آنها نیاز دارید را در قالب یک فیلتر wrap کنید و یا از یک کلاس کنترلر پایه استفاده کنید (با این فرض که از اپلیکیشن‌های وب استفاده می‌کنید).

```

using System;
using System.Web.Mvc;

namespace Web.Controllers
{
    public class DataController : Controller
    {
        protected StoreContext _context;

        protected override void OnActionExecuting(ActionExecutingContext filterContext)
        {
            //make sure your DB context is globally accessible
            MyApp.StoreDB = new StoreDB();
        }

        protected override void OnActionExecuted(ActionExecutedContext filterContext)
        {
            MyApp.StoreDB.SubmitChanges();
        }
    }
}

```

این کار به شما اجازه می‌دهد که از DataContext خود در خلال یک درخواست واحد (request) استفاده کنید. تنها کاری که باید بکنید این است که از این کلاس پایه ارث بری کنید. این بدین معنا است که هر درخواست به اپلیکیشن شما یک UnitOfWork خواهد بود. که بسیار هم منطقی و قابل قبول است. در برخی موارد هم شاید این فرض درست یا کارآمد نباشد، که در این هنگام می‌توانید از آجکت‌های Command/Query استفاده کنید.

ایده‌های بعدی: چه چیزی بدست آوردیم؟ چیزهای متعددی بدست آوردیم.

تراکنش‌های روشن و صریح : دقیقا می‌دانیم که DbContext ما از کجا آمده و در هر مرحله روی چه UnitOfWork ای کار می‌کنیم. این امر هم الان، و هم در آینده بسیار مفید خواهد بود

انتزاع کمتر == شفافیت بیشتر : ما Repository ها را از دست دادیم، که دلیلی برای وجود داشتن نداشتند. به جز اینکه یک abstraction از abstraction دیگر باشند. رویکرد آجکت‌های Command/Query تمیزتر است و دلیل وجود هرکدام و مسئولیت آنها نیز روشن‌تر است

شانس کمتر برای باگ ها : رویکردهای مبتنی بر Repository باعث می‌شوند که با تراکنش‌های ناموفق یا پاره ای (partially-executed) مواجه شویم که نهایتا به یکپارچگی و صحت داده‌ها صدمه می‌زند. لازم به ذکر نیست که خطایابی و رفع چنین مشکلاتی شدیداً زمان بر و دردسر ساز است

برای مطالعه بیشتر

[ایجاد Repositories بر روی UnitOfWork](#)

[به الگوی Repository در لایه DAL خود نه بگویید!](#)

[پیاده سازی generic repository یک ضد الگو است](#)

[نگاهی به generic repositories](#)

[بدون معکوس سازی وابستگی‌ها، طراحی چند لایه شما ایراد دارد](#)

نظرات خوانندگان

نویسنده: شهرز جعفری
تاریخ: ۱۹:۵۴ ۱۳۹۳/۰۲/۰۸

سلام آرمین جان ممنون از مطلب
به نظرم جای یک بحثی خالی اونم تست پذیری کد.

نویسنده: مسعود پاکدل
تاریخ: ۲۱:۱۳ ۱۳۹۳/۰۲/۰۸

ممنون.
با بیشتر مطالب شما موافقم ولی Repository ها نیز دلیلی برای وجود دارند.
«الگوی Repository بسیار پروژه را تست پذیر می‌کند. به راحتی با استفاده از کتابخانه‌های Mock می‌توان بخش دسترسی به داده را تست کرد.
«اگر منظور شما از StoreContext ، کلاسی است که مستقیم از DbContext ارث برده است، در نتیجه امکان استفاده از دستوراتی نظیر Set of T و Entry of T یا مواردی مربوط به Change Tracking نیز به صورت مستقیم حتی در الگوی CQRS نیز وجود دارد.
چگونه می‌توانید دستوراتی این چنینی را Mock کنید؟ استفاده از کتابخانه‌های Mock نظیر Moq برای تست دستوراتی نظیر Entry Of T و SetCurrentValues و GetCurrentValues کمکی به شما نمی‌کند. (برای DbSet کتابخانه ای نظیر FakeDbSet وجود دارد ولی برای سایر دستورات خیر...)
«اگر از روش توصیه شده در [این جا](#) استفاده کنید باز برای Mock آجکت IUnitOfWork به مشکل بر خواهید خورد. در این حالت برای تست لایه‌های دسترسی بهتر است از کتابخانه‌هایی نظیر Effort استفاده نمایید.
«در بخش **شناسه مشتری جدید را نیاز داشتیم** یک راه حل را فراموش کردید و آن استفاده از GUID برای تعریف Id هر entity است در نتیجه دیگر نیازی به واکنشی مجدد رکورد نخواهید داشت.
«بهتر است متد Save را نیز در Repository قرار ندهید. متد Save باید توسط UnitOfWork به اشتراک گذاشته شده فراخوانی شود.

نویسنده: وحید نصیری
تاریخ: ۲۱:۴۰ ۱۳۹۳/۰۲/۰۸

- [How EF6 Enables Mocking DbSet more easily](#)
- [Testing with a mocking framework - EF6 onwards](#)

+ شخصا اعتقادی به Unit tests درون حافظه‌ای، [در مورد لایه دسترسی به داده‌ها ندارم](#) . به قسمت « [Limitations of EF in-memory test doubles](#) » مراجعه کنید؛ توضیحات خوبی را ارائه داده‌است.
تست درون حافظه‌ای LINQ to Objects با تست واقعی LINQ to Entities که روی یک بانک اطلاعاتی واقعی اجرا می‌شود، الزاما نتایج یکسانی نخواهد داشت (به دلیل انواع قیود بانک اطلاعاتی، پشتیبانی از SQL خاص تولید شده تا بارگذاری اشیاء مرتبط و غیره) و نتایج مثبت آن به معنای درست کار کردن برنامه در دنیای واقعی نخواهد بود. در اینجا Integration tests بهتر جواب می‌دهند و نه Unit tests.

نویسنده: جلال
تاریخ: ۲۱:۵۷ ۱۳۹۳/۰۲/۰۸

خدا از دهنش بشنوفه. مدت هاست منم به همین نتیجه رسیدم تازه وقتی فهمیدم بدون اون بازم میشه قابلیت تست پذیری رو داشت. کافیه [یه واسط از خود DbContext برنامه سازی](#) .
ولی الگوی Repository توی استفاده از کلاس‌های پایه ADO.NET مثل DbCommand و DbConnection کارایی خوبی داره.

نویسنده: مسعود پاکدل
تاریخ: ۲۲:۲ ۱۳۹۳/۰۲/۰۸

ممنونم جناب نصیری. دلیل اشاره من به عدم تست پذیری قابل قبول در حالت استفاده مستقیم از Context به خاطر وجود دستوراتی نظیر Entry of T یا موارد مربوط به ChangeTracking است که با تست درون حافظه ای نتیجه مطلوب حاصل نمی‌شود، در نتیجه بهتر است از Effort برای تست لایه دسترسی استفاده شود که عملیات را در قالب یک دیتابیس SqlCE تست می‌کند و نسخه Effort.Ef6 آن نیز از Entity Framework 6 به خوبی پشتیبانی می‌کند.

نویسنده: Ara
تاریخ: ۲۳:۱۳ ۱۳۹۳/۰۲/۰۸

با توجه به متن قضاوتتون عجولانه است !

تو پروژه‌های Huge که توصیه خود میکروسافت استفاده از Domain Driven و CQRS می‌باشد ، Repository یکی از اصول Domain Driven و Enterprise Application Pattern می‌باشد !

نویسنده: آرمین ضیاء
تاریخ: ۲۳:۵۹ ۱۳۹۳/۰۲/۰۸

با تشکر از همگی دوستان

شخصاً نظرم به نظر جناب نصیری نزدیک‌تر است. جناب پاکدل هم به نکات خوبی اشاره فرمودند. اما صرفاً توصیه‌های میکروسافت و دیگران دال بر درستی یا کارآمدی یک رویکرد نمی‌تواند باشد. مطلب پست شده مبتنی بر چندین پست از توسعه دهندگان مطرح دنیای دات نت ترجمه و تالیف شده. مسلماً هیچ راه حل نهایی (silver-bullet) ای وجود ندارد و توسعه ساختار پروژه بر اساس نیازها و تعاریف اپلیکیشن‌ها به پیش می‌رود. اما در کل می‌توان اینگونه نتیجه گیری کرد که استفاده از الگوی Repository در کنار فریم ورک‌های ORM مانند EF که مبتنی بر UnitOfWork کار می‌کنند ایده خوبی نیست. برای مطالعات بیشتر به چند لینک نمونه زیر مراجعه شود.

^ , ^ , ^ , ^ , ^ , ^

نویسنده: محسن موسوی
تاریخ: ۱:۲ ۱۳۹۳/۰۲/۰۹

در پروژه <http://nopcommerce.codeplex.com> استفاده از Repository جهت اجبار به رویکرد Command/Query بوده است.(البته اینطور برداشت میشود) جهت مطالعه <http://nopcommerce.codeplex.com/SourceControl/latest#src/Libraries/Nop.Data/EfRepository.cs> و همینطور جهت تست پذیری پروژه، راه حل‌های اشاره شده را پیاده سازی کرده.

نویسنده: محسن موسوی
تاریخ: ۱:۱۱ ۱۳۹۳/۰۲/۰۹

آقای پاکدل لطفا راجع به این جمله بیشتر توضیح بدید:

«بهتر است متد Save را نیز در Repository قرار ندهید. متد Save باید توسط UnitOfWork به اشتراک گذاشته شده فراخوانی شود. در پروژه <http://nopcommerce.codeplex.com/SourceControl/latest#src/Libraries/Nop.Data/EfRepository.cs> در نهایت این لایه سرویس است که باید اطمینان از انجام عملیات درخواستی و یا عدم انجام آنرا بدهد و برای عملیات‌های

پیچیده‌تر نیز بایستی سیاست خود را بسط دهد. منظور انجام عملیات Save و ادامه عملیات میباشد. لایه UI وظیفه فراهم آوری اطلاعات را دارد و مابقی مسائل در لایه سرویس پوشش داده میشوند. الزام این کار هم به وظیفه این لایه برمیگردد که یا این کار را میتوانم انجام دهم و یا خیر. پیاده سازی ارائه شده نقضی بر جمله‌ی نقل قول شده میباشد؟

نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۳/۰۲/۰۹ ۱۰:۱

به طور کلی هدف اصلی از الگوی واحد کار یا UnitOfWork به اشتراک گذاشتن یک Context بین همه نمونه‌های ساخته شده از Repository یا سرویس‌های برنامه است. فرض کنید در یک کنترلر شما از دو یا سه نمونه از سرویس‌ها یا Repository ها و هله سازی کرده اید. اگر قرار باشد برای اعمال تغییرات، مجبور به فراخوانی متد Save هر Repository باشیم چرا اصلا الگوی واحد کار را به کار بردیم؟ فراخوانی SaveChanges الگوی واحد کار معادل است با فراخوانی متدهای Save تمام Repository های و هله سازی شده در طی یک درخواست.

نویسنده: آرایه
تاریخ: ۱۳۹۳/۰۲/۰۹ ۱۰:۲۰

دلایل منطقی هستند و کد ارائه شده در مثال‌ها واقعاً مشکل دارد. بعضی آثار را شاید بتوان کاهش داد. مثلاً برای رفع Repository API های بی‌انتهای شاید استفاده از متدی که IQueryable برگرداند و بعد ادامه دادن کوئری در خروجی آن متد کمک کند. یک پروژه برای پیاده سازی Generic از Repository و Unit Of Work [اینجا](#) هست که مشکلات کمتری دارد.

نویسنده: محسن موسوی
تاریخ: ۱۳۹۳/۰۲/۰۹ ۱۰:۲۸

صد در صد درست. ولی فکر میکنم این مسئله باید در لایه سرویس حل بشه. در یک Application انتظار چندین و چند عملیات در طی یک Request میره. برای نمونه میگم:

- در یک کنترلر قراره یک مشتری تعریف بشه. از طرفی هم لاگ گیری‌های عمومی سیستم نیز باید انجام باشه که اصولاً در بعضی از عملیات‌ها مستقل از همدیگه باید باشند. پس باید چند بار SaveChanges فراخوانی بشه.
- عملیات لاگ گیری سیستم حتماً باید انجام بشه ولی عملیات تعریف یک مشتری میتونه دارای خطایی باشه. (استقلال بعضی از عملیات‌های سیستم در UOW)
- عملیات‌هایی که در طی یک Action در کنترلر انجام میشه: بایستی تمام اینها به لایه‌ی سرویس منتقل بشه و اونجا در طی یک SaveChange عملیات مورد نظر نتیجه بده. (رویکرد Command/Query)
- [الگوی واحد کار](#) هدف‌های بیشتری داره.
- مسئولیت هر متد در لایه سرویس مشخصه و نتیجه بازگشتی از لایه سرویس عملاً بایستی دلالت بر نتیجه‌ی عملیات رو داشته باشه. نه اینکه در یک متد در لایه سرویس عملیات درج رو انجام بده و بعد در UI عملیات خطا بده.
- جدا سازی منطق لایه‌ها در این کار مشخص نیست. (تا حدی)
- *مدیریت پیچیده وظایف در لایه سرویس به درستی انجام بشه SaveChanges ها با کمترین سربار و بهترین کارایی انجام میشه. البته فکر میکنم در پروژه اشاره شده نیز به همین مسئله دلالت داره.

<http://nopcommerce.codeplex.com/SourceControl/latest#src/Libraries/Nop.Data/EfRepository.cs>

و اینکه در بعضی از مسائل نیز باید تغییراتی صورت بگیره. مانند عملیات‌های گروهی.

و در نهایت [صحبت آقای ضیا](#) دلالت بر تفکرات متفاوت درستره.

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۰۲/۰۹ ۱۱:۳

[This is a leaky abstraction](#)

تو مبحث DDD دلیل اصلی که Repository وارد داستان شده Persistence Ignorance می باشد ، همونطور که میدونید ، این قضیه می گه که شما تو Domain نباید بگید EF این طوری Select میزنه Nhibernate یک نوع دیگه ، NoSql یک نحو دیگه (NoSql) ها هم بخاطر اینکه میتونند براحتی یک Aggregate رو ذخیره کنند میتونند ابزار خوبی برای DDD باشند!

چون Domain نباید به تکنولوژی وابسته باشد ! نباید رفرنسی به دیتا اکسس یا EF و یا ... داشته باشد فقط یک سری Interface تعریف می کند ، که یکی که بعدا به نام لایه دیتا اکسس می باشد باید این اینترفیس رو Implement کند ! در مورد CQRS هم چون معمولا Application Layer بر روی Rest هاست می شوند پس هر Request فقط شامل یک Command می باشد که Unit Of work رو هم فقط روی همان Command ایجاد می کنند

جالبه براتون بگم که در Domain Driven Design اصل بر این هست که شما در هر ترانزاکشن فقط یک Aggregate رو باید ذخیره کنید و تغییر در Aggregate های دیگه بوسیله Event Source ها Publish می شه

و از توصیه های اولیه DDD اینه که برای پروژه های Complex و Huge استفاده بشه ، پس قطعا برای یک پروژه که از این متد استفاده نمی شه و یا در ابعاد کوچکتتر می باشد کاملا حرف شما درست باشد و از پیچیده شدن برنامه جلوگیری می کند

پیاده سازی خوبیه البته زمانی که از DDD استفاده می شه استفاده از IQueryable در IRepository به عنوان نشت اطلاعات خوانده می شود و تاکید نباید استفاده شود !

این Repository های Generic میتوانند داخل یک کلاس که IRepository را Implement کرده استفاده شوند و یا به عنوان کلاس Base ان باشند

این generic repository الان از امکانات async در EF 6 داره استفاده می کنه. برای مثال NH چنین توانمندی async ای رو در حال حاضر نداره. آیا در این حالت Persistence Ignorance تامین شده؟ یعنی راحت میشه زیر ساخت این مخزن رو عوض کرد و سوئیچ کرد به یک ORM دیگه؟ و اگر نخواهیم از async استفاده کنیم، خوب یک ORM داریم که توانمندی های جدیدش رو باید ازش صرف نظر کرد. خروجی IQueryable آن که جای خودش. ORM های مختلف متدهای الحاقی خاص خودشون رو دارند و پیاده سازی یکسانی از LINQ رو ندارند. یعنی اگر با EF کار کردید و متد Include آن توسط این generic repository بخاطر خروجی IQueryable در دسترس بود، معادلی در سایر ORM ها نداره (متدهای الحاقی اون ها فرق می کنه). یا مثلا NH سطح دوم کش رو با متد الحاقی Cacheable پیاده سازی کرده. فرض کنید این رو در generic repository قرار دادیم (یک روکش روی این متد تا به ظاهر مستقیما در دسترس نباشه). خوب، الان فلان ORM دیگه که متد Cacheable رو نداره چکار باید بامش کرد؟ این برنامه و سیستم به این سادگی ها قابل تبدیل به یک ORM دیگه نیست. رسیدن به Persistence Ignorance در دنیای واقعی کار ساده ای نیست مگر اینکه از توانمندی های خوب ORM انتخاب شده صرف نظر کنیم و به قولی دست و پا شو ببریم تا قد بقیه بشه.

گذشته از این ها بحث مدل سازی هم هست. نگاشت های کلاس ها و خواص اون ها به جداول بانک اطلاعاتی در ORM های مختلف 100 درصد با هم متفاوت هست. حداقل EF و NH روش های خاص خودشون رو دارند که انطباقی با هم ندارند. یعنی این Persistence Ignorance محدود نیست به روکش کشیدن روی insert/update/delete. اینجا صحبت از یک سیستم هست که اجزای هماهنگ زیادی داره که باید در نظر گرفته بشه! از نگاشت ها تا اعتبارسنجی های خاص تا قابلیت های ویژه و صد درصد اختصاصی. به این می گن تا خرخره فرو رفتن!

در مورد async راست می‌گید! باید بینم راهی داره یا نه، در ضمن در لایه دیتا اکسس هر جور که می‌خواهید می‌تونید include و غیره بنویسید مشکلی وجود نداره، چون رو اینترفیس از شما می‌خواهند که یک entity چه چیزهایی همراهش باشه یا نباشه خوب اگه به cqrs نگاه کنید در سمت Command شما قسمت اصلی و insert، update، delete و Get رو دارید و برخی مواقع getAll که خیلی کمه ولی سمت query کاملا دستتون بازه هر جور که کار کنید کلا پشت query سرویس هر جور که راحتی با هر چی که راحتی کار کن!

نویسنده: Ara

تاریخ: ۱۷:۱۷ ۱۳۹۳/۰۲/۱۳

مثل اینکه async کردن متدهای Repository زیاد پیچیده نمی‌باشد!

پس می‌شه query های NH رو هم Async کرد، پس روی IRepository می‌تونیم هم متد Async هم متد Sync رو با هم داشته باشیم

نویسنده: علیرضا

تاریخ: ۱۲:۵۷ ۱۳۹۳/۰۲/۲۶

- 1- من دقیقا متوجه نشدم منظور شما از decoupling اول مقاله چیه؟ منظورتون تفکیک Domain از DAL هست؟ اگر اینطوره چه ربطی به UoW و انواع پیاده سازی اون داره؟
اگر منظورتون انفکاک بین EFContext و Repository هست، توجه شما رو به این نکته جلب میکنم که StudentRepository که در اول مقاله آورده شده در حقیقت یک پیاده سازی برپایه EF هست به عبارتی EFStudentRepository اسم مناسبتری میتونه باشه. بنابراین تزریق Context با هیچ اصلی مغایر نیست. چرا که این Repository یک پیاده سازی خاص از IStudentRepository است.
- 2- وجود متد Save در Repository؟ نه تنها قابل قبول نیست که اصلا اگر قرار باشه هر Repository مستقلا Save رو صدا بزنه که مفهوم Transaction از بین میره یا حداقل سخت میشه بهش رسید.
- 3- با شما موافقم که Generic Repository ایده خوبی نیست. البته فقط تا اینجا موافقم که این الگو برای Expose کردن Interface یک Repository مناسب نیست. چه بسا Repository هایی که فقط SELECT میکنند. ولی اگر پیاده سازی خاصی از یک Repository مد نظر دارید (مثلا پیاده سازی برپایه EF یا NHibernate) اونوقت دقیقا چیزی که به کمک شما میاد همین Generic Repository برای جلوگیری از کدهای تکراریه.
- 4- اصولا Repository برای اینکه منطق برنامه (یا به قول شما منطق تجاری) رو پیاده سازی کنه نیست. در حقیقت لایه ای که استفاده کننده مستقیم از Repository است میداند که چه موقع به چه Repository فرمانهای CRUD بده تا منطق برنامه پیاده سازی بشه.

5- در واقع استفاده از امکانات هر ORM تا حد بینهایتی امکان پذیره به شرطی که ORM و توانمندیهاشو در همون لایه DAL محصور کنید مثلا IQueryable و Cachable و گرنه Leaky Abstraction به طور خزنده و ساکتی کل برنامه رو مثل سرطان در خودش میکشه.

نهایتا اینکه همیشه یک پیاده سازی مشکل دار از مفهوم Repository + UoW رو بدون در نظر گرفتن مفاهیم مهمی مثل Service Layer و Domain Model نقد کرد و بعدا نتیجه گرفت که این الگوها صحیح نیستند. ضمن اینکه این موضوع بسته به تجربه و نظر هر برنامه نویس و معماری میتونه پیاده سازی خاص خودشو داشته باشه که من شخصا هنوز موارد جالب و جدیدی که یک برنامه نویس باهوش برداشت کرده رو میبینم و نتیجه میگیرم که مفهوم Repository + UoW در بین ماها هنوز به یک تعریف جهانشمول نرسیده.

نویسنده: وحید نصیری

تاریخ: ۱۳:۱۸ ۱۳۹۳/۰۲/۲۶

- مباحث الگوی مخزن، در حالت کلی درست هستند؛ یک بحث انتزاعی، بدون در نظر گرفتن فناوری پیاده سازی کننده‌ی آن.
- در مورد EF به خصوص (در این مطلب)، DbSet و DbContext آن پیاده سازی کننده‌ی الگوهای Repository و Uow هستند (و منکر آن نیستند). به همین جهت عنوان می‌کنند که روی Repository آن، دوباره یک Repository درست نکنید. در بحث هم اشاره به «یک abstraction از abstraction دیگر» همین مطلب است.

```
public class MyContext : DbContext
class System.Data.Entity.DbContext
A DbContext instance represents a combination of the Unit Of Work and Repository patterns
```

تصویری است از قرار دادن کرسر ماوس بر روی DbContext در VS.NET که به صراحت در آن از پیاده سازی الگوی مخزن یاد شده

[اینترفیس IDbSet](#) معروف در EF دقیقا یک abstraction است و بیانگر ساختار الگوی مخزن. کاملا هم قابلیت mocking دارد؛ از نگارش 6 به بعد EF البته (^ و ^ و ^).

- راه حل‌های ارائه شده به دلیل اینکه Uow را تزریق نمی‌کنند مشکل دارند. اساسا هرگونه لایه بندی بدون تزریق وابستگی‌ها مشکل دارد؛ نمی‌شود یک وهله از یک شیء را بین چندین کلاس درگیر به اشتراک گذاشت (مباحث مدیریت طول عمر در IoC Containerها). مثلا در راه حل آخر ارائه شده فقط آغاز و پایان اجرای یک متد از یک کنترلر مشخص تحت نظر هستند. واقعیت این است که تا اجرای یک اکشن متد به پایان برسد، در طول یک درخواست، پردازش referrer رسیده هم در کلاسی دیگر به موازت آن باید انجام شود (در یک HTTP Module مجزا) و امثال آن. در این حالت چون یک وهله از Uow به اشتراک گذاشته نشده، مدام باید وهله سازی شود؛ بجای اینکه از آن تا پایان درخواست، استفاده‌ی مجدد شود. برای حل آن، در متن ذکر شده مطمئن شوید که «globally accessible» است. این مورد و راه حل‌های استاتیک (مانند نحوه‌ی فراخوانی MyApp آن) و singleton در برنامه‌های وب تا حد ممکن باید پرهیز شود. چون به معنای به اشتراک گذاری آن در بین تمام کاربران سایت. این مورد تخریب اطلاعات را به همراه خواهد داشت. چون DbContext جاری در حال استفاده توسط کاربر الف است و در همان زمان کاربر ب هم چون دسترسی عمومی به آن تعریف شده، مشغول به استفاده از آن خواهد شد. در این بین عملا تراکنش تعریف شده بی‌معنا است چون اطلاعات آن خارج از حدود متدهای مدنظر توسط سایر کاربران تغییر کرده‌اند.
همچنین به دلیل عدم تزریق وابستگی‌ها، پیاده سازی‌های آن تعویض پذیر نیستند و قابلیت آزمایش واحد پایینی خواهند داشت. برای مثال در بحث mocking که مطرح شد، می‌توانید بگویید بجای این متد خاص از کلاس اصلی، نمونه‌ی آزمایشی من را استفاده کن.