عنوان: Repository ها روی UnitOfWork ایده خوبی نیستند
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۳/۰۲/۰۸ ۱۹:۳۰
آدرس: www.dotnettips.info
گروه‌ها: Design patterns, Entity framework, ORM

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 Repository APIهای بی انتها را بدست می‌دهد. منظور از Repository APIهای 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 واکشی اطلاعات و نمایش آنها، و یا به پیچیدگی پردازش یک سفارش جدید باشد. هنگامی که شما از EntityFramework استفاده می‌کنید و یک DbContext را وهله 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 مطلقا یک UnitOfWork نیست

هر متد در یک Repository قرار است فرمانی اتمی (Atomic) باشد - چه واکشی اطلاعات و چه ذخیره آنها. مثلا می‌توانید یک Repository داشته باشید با نام SalesRepository که اطلاعات کاتالوگ شما را واکشی 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 Repositoryها این است که بزودی و به آسانی از کنترل خارج می‌شوند و نیاز به ارجاع دیگر مخازن پیدا می‌کنند. به دلیل اینکه مثلا 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: Repository<T> این الگو دیوانه وار است. این کار عملا 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 مثلا برای عملیات CRUD از کلاسی پایه استفاده کنید تا برای هر موجودیت و فرمی نیاز به کدنویسی مجدد نباشد. برگ برنده این الگو نیز دقیقا همین است. مثلا 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 داشتم

کد بالا در CustomerRepository را در نظر بگیرید - که یک مشتری جدید را به دیتابیس اضافه می‌کند. اما CustomerID جدید چه می‌شود؟ مثلا 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 Repositoryها 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 واکشی 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 می‌آورند.

خوب، راه حل چیست؟

برای جلوگیری از این abstractionهای 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 آبجکت Query نیز می‌توانید انجام دهید. می‌توانید پست Jimmy را بیشتر مطالعه کنید، اما ایده اصلی این است که آبجکت‌های Query و Command برای دلیل مشخصی وجود دارند. می‌توانید آبجکت‌ها 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 آبجکت‌های 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 Repositoryها را از دست دادیم، که دلیلی برای وجود داشتن نداشتند. به جز اینکه یک abstraction از abstraction دیگر باشند. رویکرد آبجکت‌های 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 شدیدا 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: شهروز جعفری
تاریخ: ۱۳۹۳/۰۲/۰۸ ۱۹:۵۴

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

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

ممنون.
با بیشتر مطالب شما موافقم ولی Repositoryها 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 Entry of T و Set 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 آبجکت IUnitOfWork به مشکل بر خواهید خورد. در این حالت برای تست لایه‌های دسترسی بهتر است از کتابخانه هایی نظیر Effort استفاده نمایید.

» در بخش **شناسه مشتری جدید را نیاز داشتم** یک راه حل را فراموش کردید و آن استفاده از GUID برای تعریف Id هر entity است در نتیجه دیگر نیازی به واکشی 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: ۱۳۹۳/۰۲/۰۸ ۲۱:۴۰

- How EF6 Enables Mocking DbSets 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 حافظه‌ی 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: ۱۳۹۳/۰۲/۰۸ ۲۱:۵۷

خدا از دهنت 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 بسازی .
ولی الگوی Repository توی استفاده از کلاس‌های پایه ADO.NET مثل DbCommand و DbConnection کارایی خوبی داره.

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

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: ۱۳۹۳/۰۲/۰۸ ۲۳:۱۳

با توجه به متن قضاوتتون عجولانه است !

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

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

با تشکر از همگی دوستان

شخصا نظرم به نظر جناب نصیری نزدیک‌تر است. جناب پاکدل هم به نکات خوبی اشاره فرمودند. اما صرفا توصیه‌های مایکروسافت 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. مسلما 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: ۱۳۹۳/۰۲/۰۹ ۱:۲

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: ۱۳۹۳/۰۲/۰۹ ۱:۱۱

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 اینگونه عمل شده
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 خودرا 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 اشتراگ 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 Repositoryها وهله 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 Repositoryهای وهله 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 APIهای بی‌انتها 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 اصولا در بعضی از عملیات‌ها مستقل از همدیگه باید باشند.پس باید چند بار SaveChanges فراخوانی بشه.
عملیات لاگ گیری سیستم حتما باید انجام بشه ولی عملیات تعریف یک مشتری میتونه دارای خطایی باشه.(استقلال بعضی از عملیات‌های سیستم در UOW)
-عملیات هایی 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 عملا 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 SaveChangesها 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

نویسنده: Ara
تاریخ: ۱۳۹۳/۰۲/۱۳ ۰:۵۳

تو مبحث DDD دلیل اصلی که Repository وارد داستان شده Persistence Ignorance می‌باشد ،
همونطور که میدونید ، این قضیه می‌گه که شما تو Domain نباید بگید EF این طوری Select می‌زنه Nhibernate یک نوع دیگر ، NoSql یک نحو دیگر (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 می‌کند ، که یکی که بعدا به نام لایه دیتا اکسس می‌باشد باید این اینترفیس رو Implement کتد !
در مورد CQRS هم چون معمولا Application Layer بر روی Rest هاست می‌شوند پس هر Request فقط شامل یک Command می‌باشد که Unit Of work رو هم فقط روی همان Command ایجاد می‌کنند
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 می‌شه

و از توصیه‌های اولیه DDD اینه که برای پروژه‌های Complex و Huge استفاده بشه ،پس قطعا برای یک پروژه که از این متد استفاده نمی‌شه و یا در ابعاد کوچکتر می‌باشد کاملا حرف شما درست باشد و از پیچیده شدن برنامه جلوگیری می‌کند

نویسنده: Ara
تاریخ: ۱۳۹۳/۰۲/۱۳ ۱:۳

پیاده سازی خوبیه
البته زمانی که از DDD استفاده می‌شه استفاده از IQueryable در IRepository به عنوان نشت اطلاعات خوانده می‌شود و تاکیدا 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

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۰۲/۱۳ ۱:۱۸

این generic repository الان از امکانات async در EF 6 داره استفاده می‌کنه. برای مثال NH چنین توانمندی async ایی 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 توانمندی‌های جدیدش رو باید ازش صرفنظر کرد. خروجی IQueryable آن که جای خودش. ORMهای 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 ORMها نداره (متدهای الحاقی اون‌ها فرق می‌کنه). 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 باهاش کرد؟ این برنامه و سیستم به این سادگی‌ها قابل تبدیل به یک ORM دیگه نیست. رسیدن به Persistence Ignorance در دنیای واقعی کار ساده‌ای نیست مگر اینکه از توانمندی‌های خوب ORM انتخاب شده صرفنظر کنیم و به قولی دست و پاشو ببریم تا قد بقیه بشه.
گذشته از این‌ها بحث مدل سازی هم هست. نگاشت‌های کلاس‌ها و خواص اون‌ها به جداول بانک اطلاعاتی در ORMهای مختلف 100 درصد با هم متفاوت هست. حداقل EF و NH روش‌های خاص خودشون رو دارند که انطباقی با هم ندارن. 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 درنظر گرفته بشه؛ از نگاشت‌ها تا اعتبارسنجی‌های خاص تا قابلیت‌های ویژه و صددرصد اختصاصی. به این می‌گن تا خرخره فرو رفتن!

نویسنده: Ara
تاریخ: ۱۳۹۳/۰۲/۱۳ ۷:۴۱

در مورد async راست می‌گید ! باید ببینم راهی داره یا نه ، در ضمن در لایه دیتا اکسس هر جور که می‌خواهید می‌تونید include و غیره بزنید مشکلی وجود نداره ، چون رو اینتفریس 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: ۱۳۹۳/۰۲/۱۳ ۱۷:۱۷

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: ۱۳۹۳/۰۲/۲۶ ۱۲:۵۷

-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 منظورشما 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.

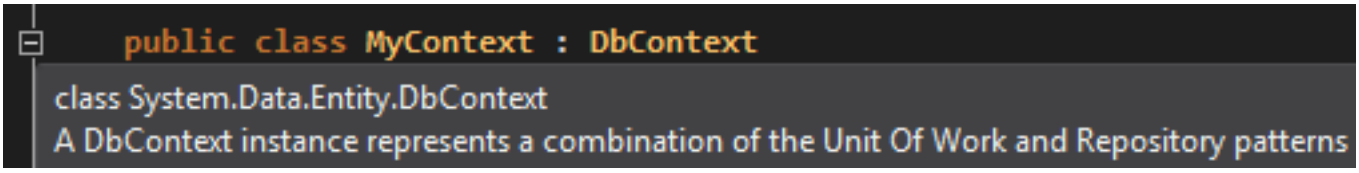
-4 اصولا Repository برای اینکه منطق برنامه (یا به قول شما منطق تجاری) رو پیاده سازی کنه نیست. در حقیقت لایه ای که استفاده کننده مستقیم از Repository است میداند که چه موقع به چه Repository فرمانهای CRUD بده تا منطق برنامه پیاده سازی بشه.

-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 IQueriable و Cachable و گرنه Leaky Abstraction به طور خزنده و ساکتی کل برنامه رو مثل سرطان در خودش میکشه.

نهایتا اینکه نمیشه یک پیاده سازی مشکل دار از مفهوم Repository + UoW رو بدون درنظر 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: ۱۳۹۳/۰۲/۲۶ ۱۳:۱۸

- مباحث الگوی مخزن، در حالت کلی درست هستند؛ یک بحث انتزاعی، بدون درنظر 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.



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.

نویسنده: ح مرادف
تاریخ: ۱۳۹۳/۱۱/۲۵ ۱۴:۱۰

سلام؛ کاملا با گفته شما موافقم. فقط مشکلی که دارم اینه که با کدهای تکراری لایه سرویس چه کنیم (CRUD). آیا راهی برای فرار از این کدها و صرفه جویی در زمان داریم ؟

نویسنده: ح مرادف
تاریخ: ۱۳۹۴/۰۲/۰۴ ۱۸:۴۲

پیشنهاد بنده برای فرار از نوشتن کدهای تکراری CRUD استفاده از یک سرویس جنریک در پروژه می‌باشد که در کمترین حالتش می‌تونه عملیات Insert و Update و Delete رو انجام بده و در متد Select نیز حداقل یک لیست از کل رکوردها خروجی بده.

بدین صورت لایه سرویس یک همچین شکلی میشه :

```
IGenericService<T>
```

```
GenericService<T> : IGenericService

IUserService : IGenericService<User>
UserService : GenericService<User>, IUserService
```

حال آنکه متدهای Add,Delete,GetAll,Update و بصورت Virtual ایجاد می‌نماییم تا در صورت نیاز (معمولا نیاز خواهیم داشت که متد Update رو در برخی موارد Override کنیم) بتونیم درون کلاس‌ها متدهای را Override کنیم.
بنده این تکنیک در پروژه عملی تست کرده ام و مشکلی نداشته ام