

در باب ضرورت نوشتن کدهای تست پذیر، توسعه کلاس‌های کوچک تک مسئولیتی و اهمیت تزریق وابستگی‌ها بارها و بارها بحث شده و مطلب نوشته شده است. این روزها کم پیش می‌آید که نرم افزاری توسعه داده شود و از پایگاه داده به جهت ذخیره و بازیابی داده‌ها استفاده نکند. با گسترش و رواج ORM ها، نوشتن کدهای دسترسی به داده‌ها سهولت یافته است و استفاده از ORM در لایه‌ی سرویس که نگهدارنده‌ی منطق تجاری برنامه است، امری اجتناب ناپذیر می‌باشد.

در این مطلب نحوه‌ی نوشتن آزمون واحد برای کلاس سرویسی که وابسته به DbContext می‌باشد، به همراه محدودیت‌ها شرح داده می‌شود. ابتدا یک روش که در آن مستقیماً از DbContext در سرویس استفاده شده را بررسی می‌کنیم. در مثال زیر کلاس ProductService وظیفه‌ی برگرداندن لیست کالاها را به ترتیب نام دارد. در آن DbContext مستقیماً و هله سازی شده و از آن جهت انجام تراکنش‌های دیتابیس کمک گرفته شده است:

```
public class ProductService
{
    public IEnumerable<Product> GetOrderedProducts()
    {
        using (var ctx = new Entites())
        {
            return ctx.Products.OrderBy(x => x.Name).ToList();
        }
    }
}
```

برای این کلاس نمی‌توان Unit Test نوشت چرا که یک وابستگی به شی DbContext دارد و این وابستگی مستقیماً درون متد GetOrderedProducts نمونه سازی شده است. در مطالب پیشین شرح داده شد که برای تست پذیر کردن کدها باید این وابستگی‌ها را از بیرون، در اختیار کلاس مورد نظر قرار داد.

برای نوشتن تست برای کلاس ProductService حداقل دو روش در اختیار است:

- نوشتن Integration Test :

یعنی کلاس جاری را به همین شکل نگاه داریم و در تست، مستقیماً به یک پایگاه داده که به منظور تست فراهم شده وصل شویم. برای سهولت مدیریت پایگاه داده می‌توان عمل درج را در یک Transaction قرار داد و پس از پایان یافتن تست Transaction را RollBack کرد. این روش مورد بحث مطلب جاری نمی‌باشد، لطفاً برای آشنایی این دو مطلب را مطالعه بفرمایید:

[Using Entity Framework in integration tests](#)

[How We Do Database Integration Tests With Entity Framework Migrations](#)

- بهره جستن از تزریق وابستگی و نوشتن Unit Test که وابستگی به دیتابیس ندارد

یکی از قانون‌های یک آزمون واحد این است که وابستگی به منابع خارجی مثل پایگاه داده نداشته باشد. [این مطلب](#) نحوه‌ی صحیح پیاده سازی الگوی Unit of Work را شرح داده است. بعد از پیاده سازی Unit Of Work، کلاس DbContext به شرح زیر می‌شود. همانطور که مشاهده می‌کنید، اکنون DbContext یک Interface را پیاده سازی کرده است.

```
public interface IUnitOfWork
{
    IDbSet<TEntity> Set<TEntity>() where TEntity : class;
    int SaveAllChanges();
}

public class Entites : DbContext, IUnitOfWork
```

```

{
    public virtual DbSet<Product> Products { get; set; } // This is virtual because Moq needs to
    override the behaviour

    public new virtual IDbSet<TEntity> Set<TEntity>() where TEntity : class // This is virtual
    because Moq needs to override the behaviour
    {
        return base.Set<TEntity>();
    }

    public int SaveAllChanges()
    {
        return base.SaveChanges();
    }
}

```

در این حالت می‌توان به جای وهله سازی مستقیم DbContext در ProductService آن را خارج از کلاس سرویس در اختیار استفاده کننده قرار داد:

```

public class ProductService
{
    private readonly IDbSet<Product> _products;
    private readonly IUnitOfWork _uow;
    public ProductService(IUnitOfWork uow)
    {
        _uow = uow;
        _products = _uow.Set<Product>();
    }
    public IEnumerable<Product> GetOrderedProducts()
    {
        return _products.OrderBy(x => x.Name).ToList();
    }
}

```

همانطور که مشاهده می‌کنید، الان IUnitOfWork به کلاس سرویس تزریق شده و در متدها، خبری از وهله سازی یک وابستگی (DbContext) نمی‌باشد.

اکنون برای تست این سرویس می‌توان پیاده سازی دیگری را از IUnitOfWork انجام داد و در کدهای تست به سرویس مورد نظر تزریق کرد. برای سهولت این امر قصد داریم از moq به عنوان چارچوب تقلید (Mocking framework) استفاده کنیم. برای [نصب moq](#) می‌توان از [بسته‌ی نیوگت](#) آن بهره جست. پیشتر [مطلبی](#) در رابطه با چارچوب‌های تقلید در سایت نوشته شده است. با توجه به اینکه ProductService به دیتابیس وابستگی دارد، مقصود این است که این وابستگی با ایجاد یک نمونه‌ی mock از IUnitOfWork حذف شود. برای این منظور در سازنده‌ی کلاس، تعدادی کالای درون حافظه ایجاد شده و به صورت IQueryable جایگزین شده DbSet است.

اگر به تعریف کلاس Entities که همان DbContext می‌باشد دقت کنید، مشاهده می‌شود که Products و تابع Set، هر دو به صورت Virtual تعریف شده‌اند. برای تغییر رفتار DbContext نیاز است در آزمون واحد، این دو با داده‌های درون حافظه کار کنند و رفتار آنها قرار است عوض شود. این تغییر رفتار از طریق چند ریختی (Polymorphism) خواهد بود. کلاس تست در نهایت اینگونه تعریف می‌شود:

```

[TestFixture]
public class ProductServiceTest
{
    private readonly ProductService _productService;
    public ProductServiceTest()
    {
        IQueryable<Product> data = GetRoadNetworks().AsQueryable();
        var mockSet = new Mock<DbSet<Product>>();
        mockSet.As<IQueryable<Product>>().Setup(m => m.Provider).Returns(data.Provider);
        mockSet.As<IQueryable<Product>>().Setup(m => m.Expression).Returns(data.Expression);
        mockSet.As<IQueryable<Product>>().Setup(m => m.ElementType).Returns(data.ElementType);
        mockSet.As<IQueryable<Product>>().Setup(m =>
        m.GetEnumerator()).Returns(data.GetEnumerator());
        var context = new Mock<Entities>();
        context.Setup(c => c.Products).Returns(mockSet.Object);
        context.Setup(m => m.Set<Product>()).Returns(mockSet.Object);
        _productService = new ProductService(context.Object);
    }
}

```

```

}
private IEnumerable<Product> GetRoadNetworks()
{
    return new List<Product>
    {
        new Product
        {
            Id = 1,
            Name = "A"
        },
        new Product
        {
            Id = 2,
            Name = "B"
        },
        new Product
        {
            Id = 3,
            Name = "C"
        }
    };
}
[Test]
public void GetOrderedProductTest()
{
    IEnumerable<Product> products = _productService.GetOrderedProducts();
    List<string> names = products.Select(x => x.Name).ToList();
    var expected = new List<string> {"A", "B", "C"};
    CollectionAssert.AreEqual(names, expected);
}
}

```

همانطور که مشاهده می‌شود، در سازنده‌ی کلاس تست، یک منبع داده‌ی درون حافظه‌ای به صورت IQueryable تولید شده و پیاده سازی‌های تقلیدی از DbContext به همراه تابع Set و همچنین DbSet کالاهای به کمک Moq ایجاد گردیده و در اختیار ProductService قرار داده شده است.

در نهایت، در یک تست تلاش شده است تا منطق متد GerOrderedProducts مورد آزمون قرار گیرد. **محدودیت این روش:** با اینکه LINQ یک روش و سینتکس یکتا برای دسترسی به منابع داده‌ای مختلف را محیا می‌کند، اما این الزامی برای یکسان بودن نتایج، هنگام استفاده از Providerهای مختلف LINQ نمی‌باشد. در تست نوشته شده از LINQ To Objects برای کوئری گرفتن از منبع داده استفاده شده است؛ در صورتیکه در برنامه‌ی اصلی از LINQ To Entities استفاده می‌شود و الزامی نیست که یک کوئری LINQ در دو Provider متفاوت یک رفتار را داشته باشد.

این نکته در قسمت Limitations of EF in-memory test doubles [این مطلب](#) هم شرح داده شده است. در نهایت این پرسش به وجود می‌آید که با وجود محدودیت ذکر شده، از این روش استفاده شود یا خیر؟ پاسخ این پرسش، بسته به هر سناریو، متفاوت است.

به عنوان نمونه اگر در یک سناریو داده‌ها با یک کوئری نه چندان پیچیده از منبع داده ای گرفته می‌شود و اعمال دیگری دیگری روی نتیجه‌ی کوئری درون حافظه انجام می‌شود می‌توان این روش را قابل اعتماد قلمداد کرد. [EFTesting.zip](#) برای مطالعه‌ی بیشتر مطالب متعددی در سایت در رابطه با [تزیق وابستگی](#) و آزمون‌های واحد نوشته شده است.

نظرات خوانندگان

نویسنده: شاهین کیاست
تاریخ: ۱۸:۴۶ ۱۳۹۳/۰۹/۰۳

-نکته تکمیلی در صورتی که از AsNoTracking در کدهای لایه‌ی سرویس استفاده شده برای Mock کردن آن می‌توان به این صورت عمل کرد:

```
context.Setup(c => c..AsNoTracking()).Returns(mockSet.Object);
```

در صورت عدم درج کد بالا تست‌ها با خطای Null Exception متوقف می‌شوند. [اطلاعات بیشتر](#)

نویسنده: ح مراداف
تاریخ: ۰:۳۶ ۱۳۹۳/۱۱/۱۸

با سلام و تشکر بابت مقاله جذابتون.

درون سایت Rhino Moq معرفی شده و شما Moq رو معرفی کردید ، بنده با Moq و کدنویسی اون احساس راحتی بیشتری می‌کنم ، می‌خواستم بدونم توی عملکرد آیا با هم فرقی دارن ؟
بنده بیشتر درگیر ساخت یک تقلید از کانتکس هستم (دقیقا مشابه کاری که شما در مقاله جاری انجام داده اید)
می‌خواستم ببینم اگر Rhino امکانات خاصی در این زمینه ارائه نمیده با Moq کار کنم.
(دنبال یک فریم ورک تقید خوب هستم که همیشه با اون کار کنم و باهانش راحت باشم)