

طراحی یک معماری خوب و مناسب یکی از عوامل مهم تولید یک برنامه کاربردی موفق می باشد. بنابراین انتخاب یک ساختار مناسب به منظور تولید برنامه کاربردی بسیار مهم و تا حدودی نیز سخت است. در اینجا یاد خواهیم گرفت که چگونه یک طراحی مناسب را انتخاب نماییم. همچنین روش های مختلف تولید برنامه های کاربردی را که مطمئنا شما هم از برخی از این روشها استفاده نمودید را بررسی می نماییم و مزایا و معایب آن را نیز به چالش می کشیم.

ضد الگو (Antipattern) - رابط کاربری هوشمند (Smart UI)

با استفاده از Visual Studio یا به اختصار VS ، می توانید برنامه های کاربردی را به راحتی تولید نمایید. طراحی رابط کاربری به آسانی عمل کشیدن و رها کردن (Drag & Drop) کنترل ها بر روی رابط کاربری قابل انجام است. همچنین در پشت رابط کاربری (Code Behind) تمامی عملیات مربوط به مدیریت رویدادها، دسترسی به داده ها، منطق تجاری و سایر نیازهای برنامه کاربردی، کد نویسی خواهند شد. مشکل این نوع کدنویسی بدین شرح است که تمامی نیازهای برنامه در پشت رابط کاربری قرار می گیرند و موجب تولید کدهای تکراری، غیر قابل تست، پیچیدگی کدنویسی و کاهش قابلیت استفاده مجدد از کد می گردد.

به این روش کد نویسی Smart UI می گویند که موجب تسهیل تولید برنامه های کاربردی می گردد. اما یکی از مشکلات عمده ای این روش، کاهش قابلیت نگهداری و پشتیبانی و عمر کوتاه برنامه های کاربردی می باشد که در برنامه های بزرگ به خوبی این مشکلات را حس خواهید کرد.

از آنجایی که تمامی برنامه نویسان مبتدی و تازه کار، از جمله من و شما در روزهای اول برنامه نویسی، به همین روش کدنویسی می کردیم، لزومی به ارائه مثال در رابطه با این نوع کدنویسی نمی بینم.

تفکیک و جدا سازی اجزای برنامه کاربردی (Separating Your Concern)

راه حل رفع مشکل Smart UI ، لایه بندی یا تفکیک اجزای برنامه از یکدیگر می باشد. لایه بندی برنامه می تواند به شکل های مختلفی صورت بگیرد. این کار می تواند توسط تفکیک کدها از طریق فضای نام (Namespace) ، پوشه بندی فایل های حاوی کد و یا جداسازی کدها در پروژه های متفاوت انجام شود. در شکل زیر نمونه ای از معماری لایه بندی را برای یک برنامه کاربردی بزرگ می بینید.



به منظور پیاده سازی یک برنامه کاربردی لایه بندی شده و تفکیک اجزای برنامه از یکدیگر، مثالی را پیاده سازی خواهیم کرد. ممکن است در این مثال با مسائل جدید و شیوه‌های پیاده سازی جدیدی مواجه شوید که این نوع پیاده سازی برای شما قابل درک نباشد. اگر کمی صبر پیشه نمایید و این مجموعه‌ی آموزشی را پیگیری کنید، تمامی مسائل نامانوس با جزئیات بیان خواهند شد و درک آن برای شما ساده خواهد گشت. قبل از شروع این موضوع را هم به عرض برسانم که علت اصلی این نوع پیاده سازی، انعطاف پذیری بالای برنامه کاربردی، پشتیبانی و نگهداری آسان، قابلیت تست پذیری با استفاده از ابزارهای تست، پیاده سازی پروژه بصورت تیمی و تقسیم بخشهای مختلف برنامه بین اعضای تیم و سایر مزایای فوق العاده آن می‌باشد.

1- Visual Studio را باز کنید و یک Solution خالی با نام SoCPatterns.Layered ایجاد نمایید.

• جهت ایجاد Solution خالی، پس از انتخاب New Project، از سمت چپ گزینه Other Project Types و سپس Visual Studio Solutions را انتخاب نمایید. از سمت راست گزینه Blank Solution را انتخاب کنید.

2- بر روی Solution کلیک راست نموده و از گزینه Add > New Project یک پروژه Class Library با نام SoCPatterns.Layered.Repository ایجاد کنید.

3- با استفاده از روش فوق سه پروژه Class Library دیگر با نامهای زیر را به Solution اضافه کنید:

SoCPatterns.Layered.Model

SoCPatterns.Layered.Service

SoCPatterns.Layered.Presentation

4- با توجه به نیاز خود یک پروژه دیگر را باید به Solution اضافه نمایید. نوع و نام پروژه در زیر لیست شده است که شما باید با توجه به نیاز خود یکی از پروژههای موجود در لیست را به Solution اضافه کنید.

(Windows Forms Application (SoCPatterns.Layered.WinUI

(WPF Application (SoCPatterns.Layered.WpfUI

(ASP.NET Empty Web Application (SoCPatterns.Layered.WebUI

(ASP.NET MVC 4 Web Application (SoCPatterns.Layered.MvcUI

5- بر روی پروژه SoCPatterns.Layered.Repository کلیک راست نمایید و با انتخاب گزینه Add Reference به پروژهی SoCPatterns.Layered.Model ارجاع دهید.

6- بر روی پروژه SoCPatterns.Layered.Service کلیک راست نمایید و با انتخاب گزینه Add Reference به پروژههای SoCPatterns.Layered.Repository و SoCPatterns.Layered.Model ارجاع دهید.

7- بر روی پروژه SoCPatterns.Layered.Presentation کلیک راست نمایید و با انتخاب گزینه Add Reference به پروژههای SoCPatterns.Layered.Service و SoCPatterns.Layered.Model ارجاع دهید.

8- بر روی پروژهی UI خود به عنوان مثال SoCPatterns.Layered.WebUI کلیک راست نمایید و با انتخاب گزینه Add Reference به پروژههای SoCPatterns.Layered.Model ، SoCPatterns.Layered.Repository ، SoCPatterns.Layered.Service و SoCPatterns.Layered.Presentation ارجاع دهید.

9- بر روی پروژهی UI خود به عنوان مثال SoCPatterns.Layered.WebUI کلیک راست نمایید و با انتخاب گزینه Set as StartUp Project پروژهی اجرایی را مشخص کنید.

10- بر روی Solution کلیک راست نمایید و با انتخاب گزینه Add > New Solution Folder پوشه‌های زیر را اضافه نموده و پروژه‌های مرتبط را با عمل Drag & Drop در داخل پوشه‌ی مورد نظر قرار دهید.

UI .1

SoCPatterns.Layered.WebUI §

Presentation Layer .2

SoCPatterns.Layered.Presentation §

Service Layer .3

SoCPatterns.Layered.Service §

Domain Layer .4

SoCPatterns.Layered.Model §

Data Layer .5

SoCPatterns.Layered.Repository §

توجه داشته باشید که پوشه بندی برای مرتب سازی لایه ها و دسترسی راحت تر به آنها می باشد.

پیاده سازی ساختار لایه بندی برنامه به صورت کامل انجام شد. حال به پیاده سازی کدهای مربوط به هر یک از لایه ها و بخش ها می پردازیم و از لایه Domain شروع خواهیم کرد.

نظرات خوانندگان

نویسنده: آرمان فرقانی
تاریخ: ۲۰:۲ ۱۳۹۱/۱۲/۲۸

مباحثی از این دست بسیار مفید و ضروری است و به شدت استقبال می‌کنم از شروع این سری مقالات. البته پیش‌تر هم مطالبی از این دست در سایت ارائه شده است که امیدوارم این سری مقالات بتونه تا حدی پراکندگی مطالب مربوطه را از بین ببرد. فقط لطف بفرمایید در این سری مقالات مرز بندی مشخصی برای برخی مفاهیم در نظر داشته باشید. به عنوان مثال گاهی در یک مقاله مفهوم Repository معادل مفهوم لایه سرویس در مقاله دیگر است. یا Domain Model مرز مشخصی با View Model داشته باشد. همچنین بحث‌های خوبی مهندس نصیری عزیز در مورد عدم نیاز به ایجاد Repository در مفهوم متداول در هنگام استفاده از EF داشتند که در رفرنس‌های معتبر دیگری هم مشاهده می‌شود. لطفاً در این مورد نیز بحث بیشتری با مرز بندی مشخص داشته باشید.

نویسنده: حسن
تاریخ: ۲۲:۵ ۱۳۹۱/۱۲/۲۸

آیا صرفاً تعریف چند ماژول مختلف برنامه را لایه بندی می‌کند و ضمانتی است بر لایه بندی صحیح، یا اینکه استفاده از الگوهای MVC و MVVM می‌توانند ضمانتی باشند بر جدا سازی حداقل لایه نمایشی برنامه از لایه‌های دیگر، حتی اگر تمام اجزای یک برنامه داخل یک اسمبلی اصلی قرار گرفته باشند؟

نویسنده: میثم خوشبخت
تاریخ: ۰:۵۳ ۱۳۹۱/۱۲/۲۹

این سری مقالات جمع بندی کامل معماری لایه بندی نرم افزار است. پس از پایان مقالات یک پروژه کامل رو با معماری منتخب پیاده سازی میکنم تا تمامی شک و شبهات برطرف بشه. در مورد مرز بندی لایه‌ها هم صحبت می‌کنم و مفهوم هر کدام را دقیقاً توضیح میدم.

نویسنده: میثم خوشبخت
تاریخ: ۰:۵۹ ۱۳۹۱/۱۲/۲۹

اگر مقاله فوق رو با دقت بخونید متوجه میشوید که MVC و MVVM در لایه UI پیاده سازی میشن. البته در MVC لایه Model رو به Domain Model و Repository در برخی مواقع لایه Controller رو در لایه Presentation قرار میدن. در MVVM نیز لایه Model در Domain Model و Repository و لایه View Model نیز در لایه Presentation قرار میگیره. همچنین View Model‌ها نیز در لایه Service قرار میگیرن.

در مورد ماژول بندی هم اگر در مقاله خونده باشید میتونید لایه‌ها رو از طریق پوشه‌ها، فضای نام و یا پروژه‌ها از هم جدا کنید

نویسنده: حسن
تاریخ: ۱۰:۱۴ ۱۳۹۱/۱۲/۲۹

شما در مطلبتون با ضدالگو شروع کردید و عنوان کردید که روش code behind یک سری مشکلاتی رو داره. سؤال من هم این بود که آیا صرفاً تعریف چند ماژول جدید می‌تواند ضمانتی باشد بر رفع مشکل code behind یا اینکه با این ماژول‌ها هم نهایتاً همان مشکل قبل پابرجا است یا می‌تواند پابرجا باشد.

ضمن اینکه تعریف شما از لایه دقیقاً چی هست؟ به نظر فقط تعریف یک اسمبلی در اینجا لایه نام گرفته.

نویسنده: آرمان فرقانی
تاریخ: ۱۱:۵۸ ۱۳۹۱/۱۲/۲۹

صحبت شما کاملاً صحیح است و صرفاً با ماژولار کردن به معماری چند لایه نمی‌رسیم. اما نویسنده مقاله نیز چنین نگفته و در پایان مقاله بحث پایان ساختار چند لایه است و نه پایان پروژه. این قسمت اول این سری مقالات است و قطعاً در هنگام پیاده سازی کدهای هر لایه مباحثی مطرح خواهد شد تا تضمین مفهوم مورد نظر شما باشد.

نویسنده: میثم خوشبخت

تاریخ: ۱۳۹۱/۱۲/۲۹ ۱۲:۳۸

با تشکر از دوست عزیزم جناب آقای آرمان فرقانی با توضیحی که دادند. یکی از دلایل این شیوه کد نویسی امکان تست نویسی برای هر یک از لایه‌ها و همچنین استقلال لایه‌ها از هم دیگه هست که هر لایه بتونه بدون وجود لایه‌ی دیگه تست بشه. ماژولار کردنه ممکنه مشکل Smart UI رو حل کنه و ممکنه حل نکنه. بستگی به شیوه کد نویسی داره.

نویسنده: بهروز

تاریخ: ۱۳۹۱/۱۲/۲۹ ۱۳:۱۱

وقتی نظرات زیر مطلب شما رو میخونم میفهمم که نیاز به این سری آموزشی که دارید ارائه میدید چقدر زیاد احساس میشه فقط می‌خواستم بگم بر سر این مبحثی که دارید ارائه می‌دید اختلاف بین علما زیاد است! (حتی در عمل و در شرکت‌های نرم افزاری که تا به حال دیدم چه برسد در سطح آموزش...) امیدوارم این حساسیت رو در نظر بگیرید و همه ما پس از مطالعه این سری آموزشی به فهم مشترک و یکسانی در مورد مفاهیم موجود برسیم فکر می‌کنم وجود یک پروژه برای دست یافتن به این هدف هم ضروری باشد باز هم تشکر

نویسنده: میثم خوشبخت

تاریخ: ۱۳۹۱/۱۲/۲۹ ۱۳:۵۹

من هم وقتی کار بر روی این معماری رو شروع کردم با مشکلات زیادی روبرو بودم و خیلی از مسائل برای من هم نامانوس و غیر قابل هضم بود. ولی بعد از اینکه چند پروژه نرم افزاری رو با این معماری پیاده سازی کردم فهم بیشتری نسبت به اون پیدا کردم و خیلی از مشکلات موجود رو با دقت بالا و با در نظر گرفتن تمامی الگوها رفع کردم. امیدوارم این حس مشترک بوجود بیاد. ولی دلیل اصلی ایجاد تکنولوژی‌ها و معماری‌های جدید اختلاف نظر بین علماست. این اختلاف نظر در اکثر مواقع میتونه مفید باشه. ممنون دوست عزیز

نویسنده: مسعود مشهدی

تاریخ: ۱۳۹۲/۰۱/۰۴ ۱۸:۳۳

با سلام

بابت مطالبتون سپاسگذارم

همون طور که خودتون گفتید نظرات و شیوه‌های متفاوتی در نوع لایه بندی‌ها وجود داره.

در مقام مقایسه لایه بندی زیر چه وجه اشتراک و تفاوتی با لایه بندی شما داره.

Application.Web

Application.Manager

Application.DAL

Application.DTO

Application.Core

با تشکر

نویسنده: محمود راستین
تاریخ: ۱۳۹۴/۰۵/۲۵ ۳:۳۰

با سلام

ممنون بابت به اشتراک گذاری این مطلب.

میخواستم بدونم چرا ما باید Presentation Layer رو ایجاد کنیم ؟ شما در Presentation Layer اومدید MVP Pattern رو پیاده سازی کردید ، مثلا اگر من از MVC استفاده کنم مگه هر دوی اونها (یعنی MVP و MVC) یک presentation pattern نیستند. پس چرا باید از هر دو استفاده کنیم ؟

فرمودید برای آزمون واحد راحتتر هستیم وقتی Presentation Layer رو ایجاد کنیم ، مگر MVC این قابلیت رو نداره ؟

نویسنده: محسن خان
تاریخ: ۱۳۹۴/۰۵/۲۵ ۸:۳۶

این قسمت اول بود و یک طرح کلی غیر وابسته به فناوری خاصی. در قسمت سوم آن به مثال وب فرمها می‌رسید.

نویسنده: محمود راستین
تاریخ: ۱۳۹۴/۰۵/۲۵ ۱۴:۴۷

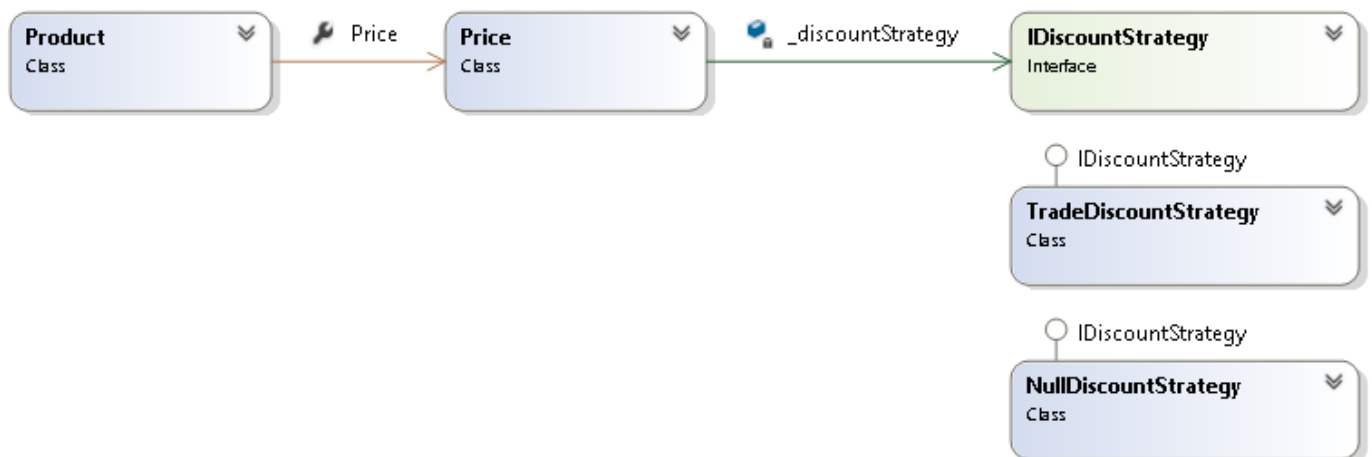
من قسمت سوم رو هم خوندم. اونجا هم به MVC اشاره ای نشده. اگر در Web Form از الگوی MVP استفاده بشه منطقی به نظر میرسه ولی سوال من این بود وقتی از MVC استفاده کنیم باز هم باید از MVP استفاده کنیم ؟ آیا این منطقیه ؟

Business Layer یا Domain Model

پیاده سازی را از منطق تجاری یا Business Logic آغاز می‌کنیم. در روش کد نویسی Smart UI ، منطق تجاری در Code Behind قرار می‌گرفت اما در روش لایه بندی، منطق تجاری و روابط بین داده‌ها در Domain Model طراحی و پیاده سازی می‌شوند. در مطالب بعدی راجع به Domain Model و الگوهای پیاده سازی آن بیشتر صحبت خواهیم کرد اما بصورت خلاصه این لایه یک مدل مفهومی از سیستم می‌باشد که شامل تمامی موجودیت‌ها و روابط بین آنهاست.

الگوی Domain Model جهت سازماندهی پیچیدگی‌های موجود در منطق تجاری و روابط بین موجودیت‌ها طراحی شده است.

شکل زیر مدلی را نشان می‌دهد که می‌خواهیم آن را پیاده سازی نماییم. کلاس Product موجودیتی برای ارائه محصولات یک فروشگاه می‌باشد. کلاس Price جهت تشخیص قیمت محصول، میزان سود و تخفیف محصول و همچنین استراتژی‌های تخفیف با توجه به منطق تجاری سیستم می‌باشد. در این استراتژی همکاران تجاری از مشتریان عادی تفکیک شده اند.



Domain Model را در پروژه SoCPatterns.Layered.Model پیاده سازی می‌کنیم. بنابراین به این پروژه یک Interface به نام IDiscountStrategy را با کد زیر اضافه نمایید:

```

public interface IDiscountStrategy
{
    decimal ApplyExtraDiscountsTo(decimal originalSalePrice);
}
  
```

علت این نوع نامگذاری Interface فوق، انطباق آن با الگوی Strategy Design Pattern می‌باشد که در مطالب بعدی در مورد این الگو بیشتر صحبت خواهیم کرد. استفاده از این الگو نیز به این دلیل بود که این الگو مختص الگوریتم‌هایی است که در زمان اجرا قابل انتخاب و تغییر خواهند بود.

توجه داشته باشید که معمولا نام Design Pattern انتخاب شده برای پیاده سازی کلاس را بصورت پسوند در انتهای نام کلاس ذکر می کنند تا با یک نگاه، برنامه نویس بتواند الگوی مورد نظر را تشخیص دهد و مجبور به بررسی کد نباشد. البته به دلیل تشابه برخی از الگوها، امکان تشخیص الگو، در پاره ای از موارد وجود ندارد و یا به سختی امکان پذیر است.

الگوی Strategy یک الگوریتم را قادر می سازد تا در داخل یک کلاس کپسوله شود و در زمان اجرا به منظور تغییر رفتار شی، بین رفتارهای مختلف سوئیچ شود.

حال باید دو کلاس به منظور پیاده سازی روال تخفیف ایجاد کنیم. ابتدا کلاسی با نام TradeDiscountStrategy را با کد زیر به پروژه SoCPatterns.Layered.Model اضافه کنید:

```
public class TradeDiscountStrategy : IDiscountStrategy
{
    public decimal ApplyExtraDiscountsTo(decimal originalSalePrice)
    {
        return originalSalePrice * 0.95M;
    }
}
```

سپس با توجه به الگوی Null Object کلاسی با نام NullDiscountStrategy را با کد زیر به پروژه SoCPatterns.Layered.Model اضافه کنید:

```
public class NullDiscountStrategy : IDiscountStrategy
{
    public decimal ApplyExtraDiscountsTo(decimal originalSalePrice)
    {
        return originalSalePrice;
    }
}
```

از الگوی Null Object زمانی استفاده می شود که نمی خواهید و یا در برخی مواقع نمی توانید یک نمونه (Instance) معتبر را برای یک کلاس ایجاد نمایید و همچنین مایل نیستید که مقدار Null را برای یک نمونه از کلاس برگردانید. در مباحث بعدی با جزئیات بیشتری در مورد الگوها صحبت خواهیم کرد.

با توجه به استراتژی های تخفیف کلاس Price را ایجاد کنید. کلاسی با نام Price را با کد زیر به پروژه SoCPatterns.Layered.Model اضافه کنید:

```
public class Price
{
    private IDiscountStrategy _discountStrategy = new NullDiscountStrategy();
    private decimal _rrp;
    private decimal _sellingPrice;
    public Price(decimal rrp, decimal sellingPrice)
    {
        _rrp = rrp;
        _sellingPrice = sellingPrice;
    }
    public void SetDiscountStrategyTo(IDiscountStrategy discountStrategy)
    {
        _discountStrategy = discountStrategy;
    }
    public decimal SellingPrice
    {
        get { return _discountStrategy.ApplyExtraDiscountsTo(_sellingPrice); }
    }
}
```

```

    }
    public decimal Rrp
    {
        get { return _rrp; }
    }
    public decimal Discount
    {
        get {
            if (Rrp > SellingPrice)
                return (Rrp - SellingPrice);
            else
                return 0;
        }
    }
    public decimal Savings
    {
        get{
            if (Rrp > SellingPrice)
                return 1 - (SellingPrice / Rrp);
            else
                return 0;
        }
    }
}

```

کلاس Price از نوعی Dependency Injection به نام Setter Injection در متد SetDiscountStrategyTo استفاده نموده است که استراتژی تخفیف را برای کالا مشخص می‌نماید. نوع دیگری از Dependency Injection با نام Constructor Injection وجود دارد که در مباحث بعدی در مورد آن بیشتر صحبت خواهیم کرد.

جهت تکمیل لایه Model ، کلاس Product را با کد زیر به پروژه SoCPatterns.Layered.Model اضافه کنید:

```

public class Product
{
    public int Id {get; set;}
    public string Name { get; set; }
    public Price Price { get; set; }
}

```

موجودیت‌های تجاری ایجاد شدند اما باید روشی اتخاذ نمایید تا لایه Model نسبت به منبع داده ای بصورت مستقل عمل نماید. به سرویسی نیاز دارید که به کلاینت‌ها اجازه بدهد تا با لایه مدل در ارتباط باشند و محصولات مورد نظر خود را با توجه به تخفیف اعمال شده برای رابط کاربری برگردانند. برای اینکه کلاینت‌ها قادر باشند تا نوع تخفیف را مشخص نمایند، باید یک نوع شمارشی ایجاد کنید که به عنوان پارامتر ورودی متد سرویس استفاده شود. بنابراین نوع شمارشی CustomerType را با کد زیر به پروژه SoCPatterns.Layered.Model اضافه کنید:

```

public enum CustomerType
{
    Standard = 0,
    Trade = 1
}

```

برای اینکه تشخیص دهیم کدام یک از استراتژی‌های تخفیف باید بر روی قیمت محصول اعمال گردد، نیاز داریم کلاسی را ایجاد کنیم تا با توجه به CustomerType تخفیف مورد نظر را اعمال نماید. کلاسی با نام DiscountFactory را با کد زیر ایجاد نمایید:

```

public static class DiscountFactory
{

```

```

public static IDiscountStrategy GetDiscountStrategyFor
(CustomerType customerType)
{
    switch (customerType)
    {
        case CustomerType.Trade:
            return new TradeDiscountStrategy();
        default:
            return new NullDiscountStrategy();
    }
}

```

در طراحی کلاس فوق از الگوی Factory استفاده شده است. این الگو یک کلاس را قادر می‌سازد تا با توجه به شرایط، یک شی معتبر را از یک کلاس ایجاد نماید. همانند الگوهای قبلی، در مورد این الگو نیز در مباحث بعدی بیشتر صحبت خواهیم کرد.

لایه‌ی سرویس با برقراری ارتباط با منبع داده‌ای، داده‌های مورد نیاز خود را بر می‌گرداند. برای این منظور از الگوی Repository استفاده می‌کنیم. از آنجایی که لایه Model باید مستقل از منبع داده‌ای عمل کند و نیازی به شناسایی نوع منبع داده‌ای ندارد، جهت پیاده‌سازی الگوی Repository از Interface استفاده می‌شود. یک Interface به نام IProductRepository را با کد زیر به پروژه SoCPatterns.Layered.Model اضافه کنید:

```

public interface IProductRepository
{
    IList<Product> FindAll();
}

```

الگوی Repository به عنوان یک مجموعه‌ی در حافظه (In-Memory Collection) یا انباره‌ای از موجودیت‌های تجاری عمل می‌کند که نسبت به زیر بنای ساختاری منبع داده‌ای کاملاً مستقل می‌باشد.

کلاس سرویس باید بتواند استراتژی تخفیف را بر روی مجموعه‌ای از محصولات اعمال نماید. برای این منظور باید یک Collection سفارشی ایجاد نماییم. اما من ترجیح می‌دهم از Extension Methods برای اعمال تخفیف بر روی محصولات استفاده کنم. بنابراین کلاسی به نام ProductListExtensionMethods را با کد زیر به پروژه SoCPatterns.Layered.Model اضافه کنید:

```

public static class ProductListExtensionMethods
{
    public static void Apply(this IList<Product> products,
                            IDiscountStrategy discountStrategy)
    {
        foreach (Product p in products)
        {
            p.Price.SetDiscountStrategyTo(discountStrategy);
        }
    }
}

```

الگوی Separated Interface تضمین می‌کند که کلاینت از پیاده‌سازی واقعی کاملاً نامطلع می‌باشد و می‌تواند برنامه نویسی را به سمت Abstraction و Dependency Inversion به جای پیاده‌سازی واقعی سوق دهد.

حال باید کلاس Service را ایجاد کنیم تا از طریق این کلاس، کلاینت با لایه Model در ارتباط باشد. کلاسی به نام ProductService را با کد زیر به پروژه SoCPatterns.Layered.Model اضافه کنید:

```
public class ProductService
{
    private IProductRepository _productRepository;
    public ProductService(IProductRepository productRepository)
    {
        _productRepository = productRepository;
    }
    public IList<Product> GetAllProductsFor(CustomerType customerType)
    {
        IDiscountStrategy discountStrategy =
            DiscountFactory.GetDiscountStrategyFor(customerType);
        IList<Product> products = _productRepository.FindAll();
        products.Apply(discountStrategy);
        return products;
    }
}
```

در اینجا کدنویسی منطق تجاری در Domain Model به پایان رسیده است. همانطور که گفته شد، لایه‌ی Business یا همان Domain Model به هیچ منبع داده‌ای خاصی وابسته نیست و به جای پیاده‌سازی کدهای منبع داده‌ای، از Interface ها به منظور برقراری ارتباط با پایگاه داده استفاده شده است. پیاده‌سازی کدهای منبع داده‌ای را به لایه‌ی Repository واگذار نمودیم که در بخش‌های بعدی نحوه پیاده‌سازی آن را مشاهده خواهید کرد. این امر موجب می‌شود تا لایه Model درگیر پیچیدگی‌ها و کد نویسی‌های منبع داده‌ای نشود و بتواند به صورت مستقل و فارغ از بخش‌های مختلف برنامه تست شود. لایه بعدی که می‌خواهیم کد نویسی آن را آغاز کنیم، لایه‌ی Service می‌باشد.

در کد نویسی‌های فوق از الگوهای طراحی (Design Patterns) متعددی استفاده شده است که به صورت مختصر در مورد آنها صحبت کردم. اصلاً جای نگرانی نیست، چون در مباحث بعدی به صورت مفصل در مورد آنها صحبت خواهیم کرد. در ضمن، ممکن است روال یادگیری و آموزش بسیار نامفهوم باشد که برای فهم بیشتر موضوع، باید کدها را بصورت کامل تست نموده و مثالهایی را پیاده‌سازی نمایید.

نظرات خوانندگان

نویسنده: سینا کردی
تاریخ: ۱۳۹۱/۱۲/۳۰ ۴:۱۰

سلام
ممنون از شما این بخش هم کامل و زیبا بود
ولی کمی فشرده بود
لطفا اگر ممکن هست در مورد معماری ها و الگوها و بهترین های آنها کمی توضیح دهید یا منبعی معرفی کنید تا این الگوها و معماری برای ما بیشتر مفهوم بشه
من در این زمینه تازه کارم و از شما میخوام که من رو راهنمایی کنید که چه مقدماتی در این زمینه ها نیاز دارم
باز هم ممنون.

نویسنده: علی
تاریخ: ۱۳۹۱/۱۲/۳۰ ۹:۱۲

در همین سایت مباحث [الگوهای طراحی](#) و [Refactoring](#) مفید هستند.

و یا الگوهای طراحی Agile رو هم [در اینجا](#) می تونید پیگیری کنید.

نویسنده: میثم خوشبخت
تاریخ: ۱۳۹۱/۱۲/۳۰ ۱۱:۳۸

فشرده گی این مباحث بخاطر این بود که میخواستم فعلا یک نمونه پروژه رو آموزش بدم تا یک شمای کلی از کاری که می خواهیم انجام بدیم رو ببینید. در مباحث بعدی این مباحث رو بازتر می کنم. خود من برای مطالعه و جمع بندی این مباحث منابع زیادی رو مطالعه کردم. واقعا برای بعضی مباحث همیشه به یک منبع اکتفا کرد.

نویسنده: محسن د.
تاریخ: ۱۳۹۱/۱۲/۳۰ ۱۷:۱

بسیار عالی

آیا فراخوانی مستقیم تابع SetDiscountStrategyTo کلاس Price در تابع الحاقی Apply از نظر کپسوله سازی مورد اشکال نیست ؟ بهتر نیست که برای خود کلاس Product یک تابع پیاده سازی کنیم که در درون خودش تابع Price.SetDiscountStrategyTo را فراخوانی کند و به این شکل کلاس های بیرونی رو از تغییرات درونی کلاس Product مستقل کنیم ؟

نویسنده: میثم خوشبخت
تاریخ: ۱۳۹۱/۱۲/۳۰ ۱۸:۱

دوست عزیزم. متد Apply یک Extension Method برای `ICollection<Product>` است. اگر این متد تعریف نمی شد شما باید در کلاس سرویس حلقه foreach رو قرار می دادید. البته با این حال در قسمت هایی از طراحی کلاسها که الگوهای طراحی را زیر سوال نمی برد و تست پذیری را دچار مشکل نمی کند، طراحی سلیقه ای است. مقاله من هم آیهی نازل شده نیست که دستخوش تغییرات نشود. شما می توانید با سلیقه و دید فنی خود تغییرات مورد نظر رو اعمال کنید. ولی اگر نظر من را بخواهید این طراحی مناسب تر است.

نویسنده: رضا عرب
تاریخ: ۱۳۹۲/۰۱/۰۹ ۱۴:۴۵

خسته نباشید، واقعا ممنونم آقای خوشبخت، لطفا به نگارش این دست مطالب مرتبط با طراحی ادامه دهید، زمینه بکریه که کمتر عملی به آن پرداخته شده و این نوع نگارش شما فراتر از یک معرفیه که واقعا جای تشکر داره.

نویسنده: f.tahan36
تاریخ: ۱۷:۱۰ ۱۳۹۲/۰۲/۲۹

با سلام

تفاوت factory با design factory در چیست؟ (با مثال کد)

و virtual کردن یک تابع معمولی با virtual کردن تابع سازنده چه تفاوتی دارد؟

با تشکر

نویسنده: محسن خان
تاریخ: ۰:۴۰ ۱۳۹۲/۰۲/۳۰

از همون رندهایی هستی که تمرین کلاسیت رو آوردی اینجا؟! :

Service Layer

نقش لایه‌ی سرویس این است که به عنوان یک مدخل ورودی به برنامه کاربردی عمل کند. در برخی مواقع این لایه را به عنوان لایه‌ی Facade نیز می‌شناسند. این لایه، داده‌ها را در قالب یک نوع داده‌ای قوی (Strongly Typed) به نام View Model، برای لایه‌ی Presentation فراهم می‌کند. کلاس View Model یک Strongly Typed محسوب می‌شود که نماهای خاصی از داده‌ها را که متفاوت از دید یا نمای تجاری آن است، بصورت بهینه ارائه می‌نماید. در مورد الگوی View Model در مباحث بعدی بیشتر صحبت خواهیم کرد.

الگوی Facade یک Interface ساده را به منظور کنترل دسترسی به مجموعه‌ای از Interface‌ها و زیر سیستم‌های پیچیده ارائه می‌کند. در مباحث بعدی در مورد آن بیشتر صحبت خواهیم کرد.

کلاسی با نام ProductViewModel را با کد زیر به پروژه SoCPatterns.Layered.Service اضافه کنید:

```
public class ProductViewModel
{
    Public int ProductId {get; set;}
    public string Name { get; set; }
    public string Rrp { get; set; }
    public string SellingPrice { get; set; }
    public string Discount { get; set; }
    public string Savings { get; set; }
}
```

برای اینکه کلاینت با لایه‌ی سرویس در تعامل باشد باید از الگوی Request/Response Message استفاده کنیم. بخش Request توسط کلاینت تغذیه می‌شود و پارامترهای مورد نیاز را فراهم می‌کند. کلاسی با نام ProductListRequest را با کد زیر به پروژه SoCPatterns.Layered.Service اضافه کنید:

```
using SoCPatterns.Layered.Model;

namespace SoCPatterns.Layered.Service
{
    public class ProductListRequest
    {
        public CustomerType CustomerType { get; set; }
    }
}
```

در شی Response نیز بررسی می‌کنیم که درخواست به درستی انجام شده باشد، داده‌های مورد نیاز را برای کلاینت فراهم می‌کنیم و همچنین در صورت عدم اجرای صحیح درخواست، پیام مناسب را به کلاینت ارسال می‌نماییم. کلاسی با نام ProductListResponse را با کد زیر به پروژه SoCPatterns.Layered.Service اضافه کنید:

```
public class ProductListResponse
{
    public bool Success { get; set; }
}
```

```

    public string Message { get; set; }
    public IList<ProductViewModel> Products { get; set; }
}

```

به منظور تبدیل موجودیت Product به ProductViewModel، به دو متد نیاز داریم، یکی برای تبدیل یک Product و دیگری برای تبدیل لیستی از Product. شما می‌توانید این دو متد را به کلاس Product موجود در Domain Model اضافه نمایید، اما این متدها نیاز واقعی منطق تجاری نمی‌باشند. بنابراین بهترین انتخاب، استفاده از Extension Method ها می‌باشد که باید برای کلاس Product و در لایه‌ی سرویس ایجاد نمایید. کلاسی با نام ProductMapperExtensionMethods را با کد زیر به پروژه SoCPatterns.Layered.Service اضافه کنید:

```

public static class ProductMapperExtensionMethods
{
    public static ProductViewModel ConvertToProductViewModel(this Model.Product product)
    {
        ProductViewModel productViewModel = new ProductViewModel();
        productViewModel.ProductId = product.Id;
        productViewModel.Name = product.Name;
        productViewModel.RRP = String.Format("{0:C}", product.Price.RRP);
        productViewModel.SellingPrice = String.Format("{0:C}", product.Price.SellingPrice);
        if (product.Price.Discount > 0)
            productViewModel.Discount = String.Format("{0:C}", product.Price.Discount);
        if (product.Price.Savings < 1 && product.Price.Savings > 0)
            productViewModel.Savings = product.Price.Savings.ToString("#%");
        return productViewModel;
    }
    public static IList<ProductViewModel> ConvertToProductListViewModel(
        this IList<Model.Product> products)
    {
        IList<ProductViewModel> productViewModels = new List<ProductViewModel>();
        foreach (Model.Product p in products)
        {
            productViewModels.Add(p.ConvertToProductViewModel());
        }
        return productViewModels;
    }
}

```

حال کلاس ProductService را جهت تعامل با کلاس سرویس موجود در Domain Model و به منظور برگرداندن لیستی از محصولات و تبدیل آن به لیستی از ProductViewModel، ایجاد می‌نماییم. کلاسی با نام ProductService را با کد زیر به پروژه SoCPatterns.Layered.Service اضافه کنید:

```

public class ProductService
{
    private Model.ProductService _productService;
    public ProductService(Model.ProductService ProductService)
    {
        _productService = ProductService;
    }
    public ProductListResponse GetAllProductsFor(
        ProductListRequest productListRequest)
    {
        ProductListResponse productListResponse = new ProductListResponse();
        try
        {
            IList<Model.Product> productEntities =
                _productService.GetAllProductsFor(productListRequest.CustomerType);
            productListResponse.Products = productEntities.ConvertToProductListViewModel();
            productListResponse.Success = true;
        }
        catch (Exception ex)
        {
            // Log the exception...
            productListResponse.Success = false;
            // Return a friendly error message
        }
    }
}

```



```

        productListResponse.Message = ex.Message;
    }
    return productListResponse;
}

```

کلاس Service تمامی خطاها را دریافت نموده و پس از مدیریت خطا، پیغامی مناسب را به کلاینت ارسال می‌کند. همچنین این لایه محل مناسبی برای Log کردن خطاها می‌باشد. در اینجا کد نویسی لایه سرویس به پایان رسید و در ادامه به کدنویسی Data Layer می‌پردازیم.

Data Layer

برای ذخیره سازی محصولات، یک بانک اطلاعاتی با نام Shop01 ایجاد کنید که شامل جدولی به نام Product با ساختار زیر باشد:

| Product | | | |
|---------|--------------|--------------|--------------------------|
| | Column Name | Data Type | Allow Nulls |
| 🔑 | ProductId | int | <input type="checkbox"/> |
| | ProductName | nvarchar(50) | <input type="checkbox"/> |
| | Rrp | smallmoney | <input type="checkbox"/> |
| | SellingPrice | smallmoney | <input type="checkbox"/> |
| | | | <input type="checkbox"/> |

برای اینکه کدهای بانک اطلاعاتی را سریعتر تولید کنیم از روش Linq to SQL در Data Layer استفاده می‌کنیم. برای این منظور یک Data Context برای Linq to SQL به این لایه اضافه می‌کنیم. بر روی پروژه SoCPatterns.Layered.Repository کلیک راست نمایید و گزینه Add > New Item را انتخاب کنید. در پنجره ظاهر شده و از سمت چپ گزینه Data و سپس از سمت راست گزینه Linq to SQL Classes را انتخاب نموده و نام آن را Shop.dbml تعیین نمایید.

از طریق پنجره Server Explorer به پایگاه داده مورد نظر متصل شوید و با عمل Drag & Drop جدول Product را به بخش Design کشیده و رها نمایید.



اگر به یاد داشته باشید، در لایه Model برای برقراری ارتباط با پایگاه داده از یک Interface به نام `IProductRepository` استفاده نمودیم. حال باید این Interface را پیاده سازی نماییم. کلاسی با نام `ProductRepository` را با کد زیر به پروژه `SoCPatterns.Layered.Repository` اضافه کنید:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using SoCPatterns.Layered.Model;

namespace SoCPatterns.Layered.Repository
{
    public class ProductRepository : IProductRepository
    {
        public IList<Model.Product> FindAll()
        {
            var products = from p in new ShopDataContext().Products
                           select new Model.Product
                           {
                               Id = p.ProductId,
                               Name = p.ProductName,
                               Price = new Model.Price(p.Rrp, p.SellingPrice)
                           };
            return products.ToList();
        }
    }
}
```

در متد `FindAll`، با استفاده از دستورات `LinQ to SQL`، لیست تمامی محصولات را برگرداندیم. کدنویسی لایه `Data` هم به پایان رسید و در ادامه به کدنویسی لایه `Presentation` و `UI` می‌پردازیم.

به منظور جداسازی منطق نمایش (Presentation) از رابط کاربری (User Interface)، از الگوی Model View Presenter یا همان MVP استفاده می‌کنیم که در مباحث بعدی با جزئیات بیشتری در مورد آن صحبت خواهیم کرد. یک Interface با نام IProductListView را با کد زیر به پروژه SoCPatterns.Layered.Presentation اضافه کنید:

```
using SoCPatterns.Layered.Service;

public interface IProductListView
{
    void Display(IList<ProductViewModel> Products);
    Model.CustomerType CustomerType { get; }
    string ErrorMessage { set; }
}
```

این Interface توسط Web Form های ASP.NET و یا Win Form ها باید پیاده سازی شوند. کار با Interface ها موجب می‌شود تا تست View ها به راحتی انجام شوند. کلاسی با نام ProductListPresenter را با کد زیر به پروژه SoCPatterns.Layered.Presentation اضافه کنید:

```
using SoCPatterns.Layered.Service;

namespace SoCPatterns.Layered.Presentation
{
    public class ProductListPresenter
    {
        private IProductListView _productListView;
        private Service.ProductService _productService;
        public ProductListPresenter(IProductListView ProductListView,
            Service.ProductService ProductService)
        {
            _productService = ProductService;
            _productListView = ProductListView;
        }
        public void Display()
        {
            ProductListRequest productListRequest = new ProductListRequest();
            productListRequest.CustomerType = _productListView.CustomerType;
            ProductListResponse productResponse =
                _productService.GetAllProductsFor(productListRequest);
            if (productResponse.Success)
            {
                _productListView.Display(productResponse.Products);
            }
            else
            {
                _productListView.ErrorMessage = productResponse.Message;
            }
        }
    }
}
```

کلاس Presenter وظیفه‌ی واکنشی داده‌ها، مدیریت رویدادها و بروزرسانی UI را دارد. در اینجا کدنویسی لایه‌ی Presentation به پایان رسیده است. از مزایای وجود لایه‌ی Presentation این است که تست نویسی مربوط به نمایش داده‌ها و تعامل بین کاربر و سیستم به سهولت انجام می‌شود بدون آنکه نگران دشواری Unit Test نویسی Web Form ها باشید. حال می‌توانید کد نویسی مربوط به UI را انجام دهید که در ادامه به کد نویسی در Win Forms و Web Forms خواهیم پرداخت.

نظرات خوانندگان

نویسنده: محسن
تاریخ: ۱۸:۲۹ ۱۳۹۲/۰۱/۰۲

ممنون از زحمات شما.

چند سؤال و نظر:

- با تعریف الگوی مخزن به چه مزیتی دست پیدا کردید؟ برای مثال آیا هدف این است که کدهای پیاده سازی آن، با توجه به وجود اینترفیس تعریف شده، شاید روزی با مثلاً NHibernate تعویض شود؟ در عمل متاسفانه حتی پیاده سازی LINQ اینها هم متفاوت است و من تابحال در عمل ندیدم که ORM یک پروژه بزرگ رو عوض کنند. یعنی تا آخر و تا روزی که پروژه زنده است با همان انتخاب اول سر می‌کنند. یعنی شاید بهتر باشه قسمت مخزن و همچنین سرویس یکی بشن.
- چرا لایه سرویس تعریف شده از یک یا چند اینترفیس مشتق نمی‌شود؟ اینطوری تهیه تست برای اون ساده‌تر میشه. همچنین پیاده سازی‌ها هم وابسته به یک کلاس خاص نمی‌شن چون از اینترفیس دارن استفاده می‌کنند.
- این اشیاء Request و Response هم در عمل به نظر نوعی ViewModel هستند. درسته؟ اگر اینطوره بهتر یک مفهوم کلی دنبال بشه تا سردرگمی‌ها رو کمتر کنه.

یک سری نکته جانبی هم هست که می‌تونه برای تکمیل بحث جالب باشه:

- مثلاً الگوی Context per request بجای نوشتن new ShopDataContext بهتر استفاده بشه تا برنامه در طی یک درخواست در یک تراکنش و اتصال کار کنه.
- در مورد try/catch و استفاده از اون بحث زیاد هست. خیلی‌ها توصیه می‌کنن که یا اصلاً استفاده نکنید یا استفاده از اون‌ها رو به بالاترین لایه برنامه موکول کنید تا این وسط کرش یک قسمت و بروز استثناء در اون، از ادامه انتشار صدمه به قسمت‌های بعدی جلوگیری کنه.

نویسنده: میثم خوشبخت
تاریخ: ۲۳:۳۵ ۱۳۹۲/۰۱/۰۲

محسن عزیز. از شما ممنونم که به نکته‌های ظریفی اشاره کردید.

در سری مقالات اولیه فقط دارم یک دید کلی به کسایی میدم که تازه دارن با این مفاهیم آشنا میشن. این پروژه اولیه دستخوش تغییرات زیادی میشه. در واقع محصول نهایی این مجموعه مقالات بر پایه همین نوع لایه بندی ولی بادید و طراحی مناسب‌تر خواهد بود.

در مورد ORM هم من با چند Application سروکار داشتم که در روال توسعه بخش‌های جدید رو بنا به دلایلی با ORM یا DB متفاوتی توسعه داده اند. غیر از این موضوع، حتی بخشهایی از مدل، سرویس و یا مخزن رو در پروژه‌های دیگری استفاده کرده اند. همچنین برخی از نکات مربوط به تفکیک لایه‌ها به منظور تست پذیری راحت‌تر رو هم در نظر بگیرید.

در مورد اشیاء Request و Response هم باید خدمتان عرض کنم که برای درخواست و پاسخ به درخواست استفاده می‌شوند که چون پروژه ای که مثال زدم کوچک بوده ممکنه کاملاً درکش نکرده باشید. ما کلاسهای Request و Response متعددی در پروژه داریم که ممکنه خیلی از اونها فقط از یک View Model استفاده کنن ولی پارامترهای ارسالی یا دریافتی آنها متفاوت باشد.

در مورد try...catch هم من با شما کاملاً موافقم. به دلیل هزینه ای که دارد باید در آخرین سطح قرار بگیرد. در این مورد ما میتونیم اونو به Presentation و یا در MVC به Controller منتقل کنیم.

در مورد DbContext هم هنوز الگویی رو معرفی نکردم. در واقع هنوز وارد جزئیات لایه‌ی Data نشدم. در مورد اون اگه اجازه بدی بعداً صحبت میکنم.

نویسنده: ایلیا
تاریخ: ۰۰:۴۳ ۱۳۹۲/۰۱/۰۳

آقای خوشبخت خداقوت.

مرسی از مطالب خوبتون.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۱/۰۳ ۰:۴۸

لطفا برای اینکه نظرات حالت فنی تر و غنای بیشتری پیدا کنند، از ارسال پیام های تشکر خودداری کنید. برای ابراز احساسات و همچنین تشکر، لطفا از گزینه رای دادن به هر مطلب که ذیل آن قرار دارد استفاده کنید. این مطلب تا این لحظه 76 بار دیده شده، اما فقط 4 رای دارد. لطفا برای ابراز تشکر، امتیاز بدهید. ممنون.

نویسنده: محسن
تاریخ: ۱۳۹۲/۰۱/۰۳ ۱:۰۰

- من در عمل تفاوتی بین لایه مخزن و سرویس شما مشاهده نمی کنم. یعنی لایه مخزن داره GetAll می کنه، بعد لایه سرویس هم داره همون رو به یک شکل دیگری بر می گردونه. این تکرار کد نیست؟ این دو یکی نیستند؟

عموما در منابع لایه مخزن رو به صورت روکشی برای دستورات مثلا EF یا LINQ to SQL معرفی می کنند. فرضشون هم این است که این روش ما رو از تماس مستقیم با ORM برحذر می داره (شاید فکر می کنند ایدز می گیرند اگر مستقیم کار کنند!). ولی عرض کردم این روکش در واقعیت فقط شاید با EF یا L2S قابل تعویض باشه نه با ORM های دیگر با روش های مختلف و بیشتر یک تصور واهی هست که جنبه عملی نداره. بیشتر تئوری هست بدون پایه تجربه دنیای واقعی. ضمن اینکه این روکش باعث میشه نتونید از خیلی از امکانات ORM مورد استفاده درست استفاده کنید. مثلا ترکیب کوئری ها یا روش های به تاخیر افتاده و امثال این.

- پس در عمل شما Request ViewModel و Response ViewModel تعریف کردید.

نویسنده: شاهین کیاست
تاریخ: ۱۳۹۲/۰۱/۰۳ ۱۲:۲۷

سپاس از سری مطالبی که منتشر می کنید.

- پیشنهادی که من دارم اینه که لایه Repository حذف شود، همانطور که در مطالب قبلی ذکر شده DbSet در Entity Framework همان پیاده سازی الگوی مخزن هست و ایجاد Repository جدید روی آن یک Abstraction اضافه هست. در نتیجه اگر Repository حذف شود همه ی منطق ها مانند GetBlaBla به Service منتقل می شود.

- یک پیشنهاد دیگر اینکه استفاده از کلمه New در Presentation Layer را به حداقل رساند و همه جا نیاز مندی ها را به صورت وابستگی به کلاس های استفاده کننده تزریق شود تا در زمان نوشتن تست ها همه ی اجزاء قابل تعویض با Mock objects باشند.

نویسنده: افشین
تاریخ: ۱۳۹۲/۰۱/۰۶ ۱۱:۱۵

لطفا دمو یا سورس برنامه رو هم قرار بدید که یادگیری و آموزش سریعتر انجام بشه.

ممنون

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۲/۰۱/۱۰ ۰:۱۱

با سلام از کار بزرگی که دارین می کنین سپاس
یک سوال؟
جای الگوی Unit Of Work در این پروژه کجا میشه؟

در این [پست](#) جناب آقای مهندس نصیری در لایه سرویس الگوی واحد کار را پیاده کرده اند، با توجه به وجود الگوی Repository

در پروژه شما ممنون میشم شرح بیشتری بدین که جایگاه پیاده سازی الگو واحد کار با توجه به مزایایی که دارد در کدام لایه است؟

نویسنده: رام
تاریخ: ۵:۲۹ ۱۳۹۲/۰۱/۱۶

محسن جان، چیزی که من از این الگو در مورد واکنشی و نمایش داده‌ها برداشت میکنم اینه:

کلاس‌های لایه مخزن با دریافت دستور از لایه سرویس آبجکت مدل مربوطه را پر میکنند و به بالا (لایه سرویس) پاس میدهند.

بعد

در لایه سرویس نمونه‌ی مدل مربوطه به ویومدل متناظر باهاش تبدیل میشه و به لایه بالاتر فرستاده میشه

بنابراین

کار در "لایه مخزن" روی "مدل‌ها" انجام میگیره

و

کار در "لایه سرویس" روی "ویومدل‌ها" انجام میشه

نتیجه: لایه سرویس هدف دیگری را نسبت به لایه مخزن دنبال میکند و این هدف آنقدر بزرگ و مهم هست که برایش یه لایه مجزا در نظر گرفته بشه

نویسنده: رام
تاریخ: ۵:۴۹ ۱۳۹۲/۰۱/۱۶

شاهین جان، من با حذف لایه مخزن مخالف هستم. زیرا:

ما لایه ای به نام "لایه مخزن" را میسازیم تا در نهایت کلیه متدهایی که برای حرف زدن با داده هامون را نیاز داریم داشته باشیم. حالا این اطلاعات ممکنه از پایگاه داده یا جاهای دیگه جمع آوری بشوند (و الزاما توسط EF قابل دسترسی و ارائه نباشند)

همچنین گاهی نیاز هست که بر مبنای چند متد که EF به ما میرسونه (مثلا چند SP) یک متد کلی‌تر را تعریف کنیم (چند فراخوانی را در یک متد مثلا متد X در لایه مخزن انجام دهیم) و در لایه بالاتر آن متد را صدا بزنیم (بجای نوشتن و تکرار پاپی همه کدهای نوشت شده در متد X)

علاوه بر این در لایه مخزن میشه چند ORM را هم کنار هم دید (نه فقط EF) که همونطور که آقای خوشبخت در کامنت‌ها نوشتند گاهی نیاز میشه.

بنابراین:

من وجود لایه مخزن را ضروری میدونم.

(فراموش نکنیم که هدف از این آموزش تعریف یک الگوی معماری مناسب برای پروژه‌های بزرگ هست و الا بدون خیلی از اینها هم میشه برنامه ساخت. همونطور که اکثرا بدون این ساختارها و خیلی ساده‌تر میسازند)

نویسنده: محسن
تاریخ: ۹:۳ ۱۳۹۲/۰۱/۱۶

- بحث آقای شاهین و من در مورد مثال عینی بود که زده شد. در مورد کار با ORM که کدهاش دقیقا ارائه شده. این روش قابل نقد و رد است.

شما الان اومدی یک بحث انتزاعی کلی رو شروع کردید. بله. اگر ORM رو کنار بگذارید مثلاً می‌رسید به ADO.NET (یک نمونه که خیلی‌ها در این سایت حداقل یکبار باهاش کار کردن). این افراد پیش از اینکه این مباحث مطرح باشن برای خودشون لایه DAL داشتند و تمام جزئیات ADO.NET رو کپسوله کرده بودن در اون. حالا با اومدن ORM‌ها این لایه DAL کنار رفته چون خود ORM هست که کپسوله کننده ADO.NET است. همین‌ها هم یک لایه دیگر داشتند به نام BLL که از لایه DAL استفاده می‌کرد برای پیاده سازی منطق تجاری برنامه. این لایه الان اسمش شده لایه سرویس.

یعنی تمام مواردی رو که عنوان کردید در مورد ADO.NET صدق می‌کنه. یکی اسمش رو می‌ذاره شما اسمش رو گذاشتید Repository. ولی این مباحث ربطی به یک ORM تمام عیار که کپسوله کننده ADO.NET است ندارد.

- ترکیب چند SP در لایه مخزن انجام نمیشه. چیزی رو که عنوان کردید یعنی پیاده سازی منطق تجاری و این مورد باید در لایه سرویس باشه. اگر از ADO.NET استفاده میشه، می‌تونیم با استفاده از DAL جزئیات دسترسی به SP رو مخفی و ساده‌تر کنیم با کدی یک دست‌تر در تمام برنامه. اگر از EF استفاده می‌کنیم، باز همین ساده سازی در طی فراخوانی فقط یک متد انجام شده. بنابراین بهتر است وضعیت و سطح لایه‌ای رو که داریم باهاش کار می‌کنیم خوب بررسی و درک کنیم.

- می‌تونید در عمل در بین پروژه‌های سورس باز و معتبر موجود فقط یک نمونه رو به من ارائه بدید که در اون از 2 مورد ORM مختلف همزمان استفاده شده باشه؟ این مورد یعنی سؤ مدیریت. یعنی پراکندگی و انجام کاری بسیار مشکل مثلاً یک نمونه: ORM لایه‌ای دارند به نام سطح اول کش که مثلاً در EF اسمش هست Trackig API. این لایه فقط در حین کار با Context همون ORM کار می‌کنه. اگر دو مورد رو با هم مخلوط کنید، قابل استفاده نیست، ترکیب پذیر نیستند. از این دست باز هم هست مثلاً در مورد نحوه تولید پروکسی‌هایی که برای lazy loading تولید می‌کنند و خیلی از مسایل دیگری از این دست. ضمن اینکه مدیریت چند Context فقط در یک لایه خودش یعنی نقض اصل تک مسئولیتی کلاس‌ها.

نویسنده: محسن
تاریخ: ۱۳۹۲/۰۱/۱۶ ۹:۱۵

سعی نکنید انتزاعی بحث کنید. چون در این حالت این حرف می‌تونه درست باشه یا حتی نباشه. اگر از ADO.NET استفاده می‌کنید، درسته. اگر از EF استفاده می‌کنید غلط هست. لازم هست منطق کار با ADO.NET رو یک سطح کپسوله کنیم. چون از تکرار کد جلوگیری می‌کنه و نهایتاً به یک کد یک دست خواهیم رسید. لازم نیست اعمال یک ORM رو در لایه‌ای به نام مخزن کپسوله کنیم، چون خودش کپسوله سازی ADO.NET رو به بهترین نحوی انجام داده. برای نمونه در همین مثال عینی بالا به هیچ مزیتی نرسیدیم. فقط یک تکرار کد است. فقط بازی با کدها است.

نویسنده: رام
تاریخ: ۱۳۹۲/۰۱/۱۶ ۱۶:۴۶

من منظور شما را خوب متوجه میشم ولی حرفام به بحث انتزاعی نیست چون پروژه عملی زیر دستم دارم که توی اون هم با پر کردن View Model کار میکنم.

مشکل از اینجا شروع میشه که شما فکر میکنید همیشه مدل ای که در EF ساختید را باید بدون تغییر در ساختارش به پوسته برنامه برسونید و از پوسته هم دقیقاً نمونه ای از همون را بگیرید و به لایه‌های پایین بفرستید ولی یکی از مهمترین کارهای View Model اینه که این قانون را از این سفتی و سختی در بیاره چون خیلی مواقع هست که شما در پوسته برنامه به شکل دیگه ای از داده‌ها (متفاوت با اونچه در Model تعریف کردید و EF باهاش کار میکنه) نیاز دارید. مثلاً فیلد تاریخ از نوع DateTime در Model و نوع String در پوسته و یا حتی اضافه و کم کردن فیلدهای یک Model و ایجاد ساختارهای متفاوتی از اون برای عملیات‌های Select, Update و Delete. لذا لایه سرویس قرار نیست فقط همون کار لایه مخزن را تکرار کنه (به قول شما GetAll). بلکه در زمان لزوم تغییرات لازم که نام بردم را هم رووش اعمال میکنه (که به نظر من آقای خوشبخت هم به خوبی از کلمه Convert در لایه سرویس استفاده کردند).

اما بحث اینکه ما در لایه مخزن روی EF یک سطح کپسوله میسازیم جای گفتگو داره هرچند من در اون مورد هم با وجد لایه مخزن بیشتر موافقم تا گفتگوی مستقیم لایه سرویس با چیزی مثل EF

نتیجه: فرقی نمیکنه شما از Asp.Net استفاده میکنید یا هر ORM مورد نظرتون. کلاس‌های مدل باید در ارتباط با لایه بالاتر خودشون به ویو مدل تبدیل بشند و در این الگو این کار در لایه سرویس انجام میشه.

نویسنده:

محسن

تاریخ:

۱۷:۱۰ ۱۳۹۲/۰۱/۱۶

- پیاده سازی الگوی مخزن در عمل (بر اساس بحث فعلی که در مورد کار با ORMها است) به صورت کپسوله سازی ORM در همه جا مطرح میشه و اینکار اساسا اشتباه هست. چون هم شما رو محروم می‌کنه از قابلیت‌های پیشرفته ORM و هم ارزش افزوده‌ای رو به همراه نداره. دست آخر می‌بینید در لایه مخزن GetAll دارید در لایه سرویس هم GetAll دارید. این مساله هیچ مزیتی نداره. یک زمانی در ADO.NET برای GetAll کردن باید کلی کد شبیه به کدهای یک ORM نوشته می‌شد. خود ORM الان اومده این‌ها رو کپسوله کرده و لایه‌ای هست روی اون. اینکه ما مجدداً یک پوستره روی این بکشیم حاصلی نداره بجز تکرار کد. عده‌ای عنوان می‌کنند که حاصل اینکار امکان تعویض ORM رو ممکن می‌کنه ولی این‌ها هم بعد از یک مدت تجربه با ORMهای مختلف به این نتیجه می‌رسند که ای بابا! حتی پیاده سازی LINQ این ORMها یکی نیست چه برسه به قابلیت‌های پیشرفته‌ای که در یکی هست در دوتای دیگر نیست (واقع بینی، بجای بحث تئوری محض).

- اینکه این تبدیلات (پر کردن ViewModel از روی مدل) هم می‌تونه و بهتره که (نه الزاما) در لایه سرویس انجام بشه، نتیجه مناسبی هست.

نویسنده:

مجتبی آزاد

تاریخ:

۱۴:۳ ۱۳۹۴/۰۴/۱۰

بعد از پیاده سازی UOW و لایه‌بندی نرم‌افزار به این شکل که در مطلب فعلی توضیح داده شد، فرض کنید دو ویومدل زیر را داریم:

```
public class PersonFormViewModel
{
    public long Id { get; set; }
    public long RequestId { get; set; }
    [DisplayName("نام کاربری"), Required(ErrorMessage = "نام کاربری الزامی می‌باشد")]
    public string Username { get; set; }
    public bool Accepted { get; set; }
    [DisplayName("مدل")]
    public string DeviceModel { get; set; }
    public DateTime? ExpireDate { get; set; }
    public RequestViewModel RequestViewModel { get; set; }
}
```

```
public class RequestViewModel
{
    public long Id { get; set; }
    public string Username { get; set; }
    [DisplayName("توضیحات")]
    [DataType(DataType.MultilineText)]
    public string Description { get; set; }
    public DateTime CreateDate { get; set; }
    public Nullable<long> DeviceId { get; set; }
    public Nullable<long> ParentId { get; set; }
    public long RequestTypeId { get; set; }
    public bool IsFinalized { get; set; }
    public virtual PersonFormViewModel PersonFormViewModel { get; set; }
}
```

سناریو به این شکل است که ما فرمی داریم برای ایجاد یک درخواست (RequestViewModel) که با ایجاد آن در واقع اطلاعات شخص (PersonFormViewModel) را نیز دریافت می‌کنیم.

برای افزودن RequestViewModel به دیتابیس این دو روش قابل پیاده‌سازی است:
روش اول:

تنها RequestViewModel را از طریق RequestService اضافه می‌کنیم و به دلیل وجود PersonFormViewModel داخل RequestViewModel اطلاعات شخص به خودی خود داخل entity مربوطه اضافه می‌شود:

```
_requestService.Add(requestViewModel);
```



```
_uow.SaveChanges();
```

روش دوم:

ابتدا RequestViewModel را از طریق سرویس مربوطه اضافه می‌کنیم و بعد به طور جداگانه PersonViewModel را از طریق سرویس PersonFormService اضافه می‌کنیم:

```
var addedRequest = _requestService.Add(requestViewModel );
var personViewModel = requestViewModel .PersonFormViewModel;
_personFormService.Add(personViewModel);
_uow.SaveChanges();
```

در حال حاضر روش درست کدام است؟

نویسنده: میثم خوشبخت

تاریخ: ۱۳/۰۴/۱۳۹۴ ۱۹:۲۴

اگر مشخصات شخص به همراه درخواست ثبت می‌شود و مدخل ورودی به سیستم درخواست می‌باشد، همان روش اول اتفاق می‌افتد.

یعنی با ثبت درخواست، شخص نیز به صورت خودکار ثبت خواهد شد.

به این نکته توجه داشته باشید که روش دوم به این دلایل غلط می‌باشند:

- 1- طبق تعریفی که در معماری سرویس گرا شده است، هیچ متدی از سرویس، نباید به متد سرویس دیگری وابسته باشد
- 2- چون ثبت این دو آیتم باید با هم انجام شود، پس بهتر است در یک متد دو موجودیت ثبت شوند
- 3- و همینطور چون باید در یک تراکنش قرار بگیرند و تجربه نشان داده در هر عملیات بهتر است DataContext نمونه سازی گردد پس چندین کار که قرار است در یک مرحله انجام شوند بهتر است در یک متد سرویس انجام گیرند

UI

در نهایت نوبت به طراحی و کدنویسی UI می‌رسد تا بتوانیم محصولات را به کاربر نمایش دهیم. اما قبل از شروع باید موضوعی را یادآوری کنم. اگر به یاد داشته باشید، در کلاس ProductService موجود در لایه Domain، از طریق یکی از روشهای الگوی Dependency Injection به نام Constructor Injection، فیلدی از نوع IProductRepository را مقداردهی نمودیم. حال زمانی که بخواهیم نمونه‌ای را از ProductService ایجاد نماییم، باید به عنوان پارامتر ورودی سازنده، شی ایجاد شده از جنس کلاس ProductRepository موجود در لایه Repository را به آن ارسال نماییم. اما از آنجایی که می‌خواهیم تفکیک پذیری لایه‌ها از بین نرود و UI بسته به نیاز خود، نمونه مورد نیاز را ایجاد نموده و به این کلاس ارسال کند، از ابزارهایی برای این منظور استفاده می‌کنیم. یکی از این ابزارها StructureMap می‌باشد که یک Inversion of Control Container یا به اختصار IoC Container نامیده می‌شود. با Inversion of Control در مباحث بعدی بیشتر آشنا خواهید شد. StructureMap ابزاری است که در زمان اجرا، پارامترهای ورودی سازنده کلاسهایی را که از الگوی Dependency Injection استفاده نموده‌اند و قطعاً پارامتر ورودی آنها از جنس یک Interface می‌باشد را، با ایجاد شی مناسب مقداردهی می‌نماید.

به منظور استفاده از StructureMap در Visual Studio 2012 باید بر روی پروژه UI خود کلیک راست نموده و گزینه‌ی Manage NuGet Packages را انتخاب نمایید. در پنجره ظاهر شده و از سمت چپ گزینه‌ی Online و سپس در کادر جستجوی سمت راست و بالای پنجره واژه‌ی structuremap را جستجو کنید. توجه داشته باشید که باید به اینترنت متصل باشید تا بتوانید Package مورد نظر را نصب نمایید. پس از پایان عمل جستجو، در کادر میانی structuremap ظاهر می‌شود که می‌توانید با انتخاب آن و فشردن کلید Install آن را بر روی پروژه نصب نمایید.

جهت آشنایی بیشتر با NuGet و نصب آن در سایر نسخه‌های Visual Studio می‌توانید به لینک‌های زیر رجوع کنید:

1. [آشنایی](#)

با [NuGet قسمت اول](#)

2. [آشنایی](#)

با [NuGet قسمت دوم](#)

3. [Installing](#)

[NuGet](#)

کلاسی با نام BootStrapper را با کد زیر به پروژه UI خود اضافه کنید:

```
using StructureMap;
using StructureMap.Configuration.DSL;
using SoCPatterns.Layered.Repository;
using SoCPatterns.Layered.Model;

namespace SoCPatterns.Layered.WebUI
{
    public class BootStrapper
    {
        public static void ConfigureStructureMap()
        {
            ObjectFactory.Initialize(x => x.AddRegistry<ProductRegistry>());
        }
    }
}
```

```
public class ProductRegistry : Registry
{
    public ProductRegistry()
    {
        For<IProductRepository>().Use<ProductRepository>();
    }
}
```

ممکن است یک WinUI ایجاد کرده باشید که در این صورت به جای فضای نام SoCPatterns.Layered.WebUI از SoCPatterns.Layered.WinUI استفاده نمایید.

هدف کلاس BootStrapper این است که تمامی وابستگی‌ها را توسط StructureMap در سیستم Register نماید. زمانی که کدهای کلاینت می‌خواهند به یک کلاس از طریق StructureMap دسترسی داشته باشند، StructureMap وابستگی‌های آن کلاس را تشخیص داده و بصورت خودکار پیاده سازی واقعی (Concrete Implementation) آن کلاس را، براساس همان چیزی که ما برایش تعیین کردیم، به کلاس تزریق می‌نماید. متد ConfigureStructureMap باید در همان لحظه ای که Application آغاز به کار می‌کند فراخوانی و اجرا شود. با توجه به نوع UI خود یکی از روالهای زیر را انجام دهید:

در WebUI :

فایل Global.asax را به پروژه اضافه کنید و کد آن را بصورت زیر تغییر دهید:

```
namespace SoCPatterns.Layered.WebUI
{
    public class Global : System.Web.HttpApplication
    {
        protected void Application_Start(object sender, EventArgs e)
        {
            BootStrapper.ConfigureStructureMap();
        }
    }
}
```

در WinUI :

در فایل Program.cs کد زیر را اضافه کنید:

```
namespace SoCPatterns.Layered.WinUI
{
    static class Program
    {
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
        }
    }
}
```

;)BootStrapper.ConfigureStructureMap

```
Application.Run(new Form1());
}
```

سپس برای طراحی رابط کاربری، با توجه به نوع UI خود یکی از روالهای زیر را انجام دهید:

صفحه Default.aspx را باز نموده و کد زیر را به آن اضافه کنید:

```
<asp:DropDownList AutoPostBack="true" ID="ddlCustomerType" runat="server">
  <asp:ListItem Value="0">Standard</asp:ListItem>
  <asp:ListItem Value="1">Trade</asp:ListItem>
</asp:DropDownList>
<asp:Label ID="lblErrorMessage" runat="server" ></asp:Label>
<asp:Repeater ID="rptProducts" runat="server" >
  <HeaderTemplate>
    <table>
      <tr>
        <td>Name</td>
        <td>RRP</td>
        <td>Selling Price</td>
        <td>Discount</td>
        <td>Savings</td>
      </tr>
      <tr>
        <td colspan="5"><hr /></td>
      </tr>
    </HeaderTemplate>
    <ItemTemplate>
      <tr>
        <td><%= Eval("Name") %></td>
        <td><%= Eval("RRP")%></td>
        <td><%= Eval("SellingPrice") %></td>
        <td><%= Eval("Discount") %></td>
        <td><%= Eval("Savings") %></td>
      </tr>
    </ItemTemplate>
    <FooterTemplate>
      </table>
    </FooterTemplate>
  </asp:Repeater>
```

فایل Form1.Designer.cs را باز نموده و کد آن را بصورت زیر تغییر دهید:

```
#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
    this.cmbCustomerType = new System.Windows.Forms.ComboBox();
    this.dgvProducts = new System.Windows.Forms.DataGridView();
    this.colName = new System.Windows.Forms.DataGridViewTextBoxColumn();
    this.colRrp = new System.Windows.Forms.DataGridViewTextBoxColumn();
    this.colSellingPrice = new System.Windows.Forms.DataGridViewTextBoxColumn();
    this.colDiscount = new System.Windows.Forms.DataGridViewTextBoxColumn();
    this.colSavings = new System.Windows.Forms.DataGridViewTextBoxColumn();
    ((System.ComponentModel.ISupportInitialize)(this.dgvProducts)).BeginInit();
    this.SuspendLayout();
    //
    // cmbCustomerType
    //
    this.cmbCustomerType.DropDownStyle =
System.Windows.Forms.ComboBoxStyle.DropDownList;
    this.cmbCustomerType.FormattingEnabled = true;
    this.cmbCustomerType.Items.AddRange(new object[] {
        "Standard",
        "Trade"});
    this.cmbCustomerType.Location = new System.Drawing.Point(12, 90);
```

```

        this.cmbCustomerType.Name = "cmbCustomerType";
        this.cmbCustomerType.Size = new System.Drawing.Size(121, 21);
        this.cmbCustomerType.TabIndex = 3;
        //
        // dgvProducts
        //
        this.dgvProducts.ColumnHeadersHeightSizeMode =
System.Windows.Forms.DataGridViewColumnHeadersHeightSizeMode.AutoSize;
        this.dgvProducts.Columns.AddRange(new System.Windows.Forms.DataGridViewColumn[] {
            this.colName,
            this.colRrp,
            this.colSellingPrice,
            this.colDiscount,
            this.colSavings});
        this.dgvProducts.Location = new System.Drawing.Point(12, 117);
        this.dgvProducts.Name = "dgvProducts";
        this.dgvProducts.Size = new System.Drawing.Size(561, 206);
        this.dgvProducts.TabIndex = 2;
        //
        // colName
        //
        this.colName.DataPropertyName = "Name";
        this.colName.HeaderText = "Product Name";
        this.colName.Name = "colName";
        this.colName.ReadOnly = true;
        //
        // colRrp
        //
        this.colRrp.DataPropertyName = "Rrp";
        this.colRrp.HeaderText = "RRP";
        this.colRrp.Name = "colRrp";
        this.colRrp.ReadOnly = true;
        //
        // colSellingPrice
        //
        this.colSellingPrice.DataPropertyName = "SellingPrice";
        this.colSellingPrice.HeaderText = "Selling Price";
        this.colSellingPrice.Name = "colSellingPrice";
        this.colSellingPrice.ReadOnly = true;
        //
        // colDiscount
        //
        this.colDiscount.DataPropertyName = "Discount";
        this.colDiscount.HeaderText = "Discount";
        this.colDiscount.Name = "colDiscount";
        //
        // colSavings
        //
        this.colSavings.DataPropertyName = "Savings";
        this.colSavings.HeaderText = "Savings";
        this.colSavings.Name = "colSavings";
        this.colSavings.ReadOnly = true;
        //
        // Form1
        //
        this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
        this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
        this.ClientSize = new System.Drawing.Size(589, 338);
        this.Controls.Add(this.cmbCustomerType);
        this.Controls.Add(this.dgvProducts);
        this.Name = "Form1";
        this.Text = "Form1";
        ((System.ComponentModel.ISupportInitialize)(this.dgvProducts)).EndInit();
        this.ResumeLayout(false);
    }
}
#endregion
private System.Windows.Forms.ComboBox cmbCustomerType;
private System.Windows.Forms.DataGridView dgvProducts;
private System.Windows.Forms.DataGridViewTextBoxColumn colName;
private System.Windows.Forms.DataGridViewTextBoxColumn colRrp;
private System.Windows.Forms.DataGridViewTextBoxColumn colSellingPrice;
private System.Windows.Forms.DataGridViewTextBoxColumn colDiscount;
private System.Windows.Forms.DataGridViewTextBoxColumn colSavings;

```

سپس در Code Behind ، با توجه به نوع UI خود یکی از روالهای زیر را انجام دهید:

وارد کد نویسی صفحه Default.aspx شده و کد آن را بصورت زیر تغییر دهید:

```
using System;
using System.Collections.Generic;
using SoCPatterns.Layered.Model;
using SoCPatterns.Layered.Presentation;
using SoCPatterns.Layered.Service;
using StructureMap;

namespace SoCPatterns.Layered.WebUI
{
    public partial class Default : System.Web.UI.Page, IProductListView
    {
        private ProductListPresenter _productListPresenter;
        protected void Page_Init(object sender, EventArgs e)
        {
            _productListPresenter = new
ProductListPresenter(this, ObjectFactory.GetInstance<Service.ProductService>());
            this.ddlCustomerType.SelectedIndexChanged +=
                delegate { _productListPresenter.Display(); };
        }
        protected void Page_Load(object sender, EventArgs e)
        {
            if(!Page.IsPostBack)
                _productListPresenter.Display();
        }
        public void Display(IList<ProductViewModel> products)
        {
            rptProducts.DataSource = products;
            rptProducts.DataBind();
        }
        public CustomerType CustomerType
        {
            get { return (CustomerType) int.Parse(ddlCustomerType.SelectedValue); }
        }
        public string ErrorMessage
        {
            set
            {
                lblErrorMessage.Text =
                    String.Format("<p><strong>Error:</strong><br/>{0}</p>", value);
            }
        }
    }
}
```

وارد کدنویسی Form1 شوید و کد آن را بصورت زیر تغییر دهید:

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
using SoCPatterns.Layered.Model;
using SoCPatterns.Layered.Presentation;
using SoCPatterns.Layered.Service;
using StructureMap;

namespace SoCPatterns.Layered.WinUI
{
    public partial class Form1 : Form, IProductListView
    {
        private ProductListPresenter _productListPresenter;
        public Form1()
        {
            InitializeComponent();
            _productListPresenter =
                new ProductListPresenter(this, ObjectFactory.GetInstance<Service.ProductService>());
            this.cmbCustomerType.SelectedIndexChanged +=
                delegate { _productListPresenter.Display(); };
            dgvProducts.AutoGenerateColumns = false;
        }
    }
}
```

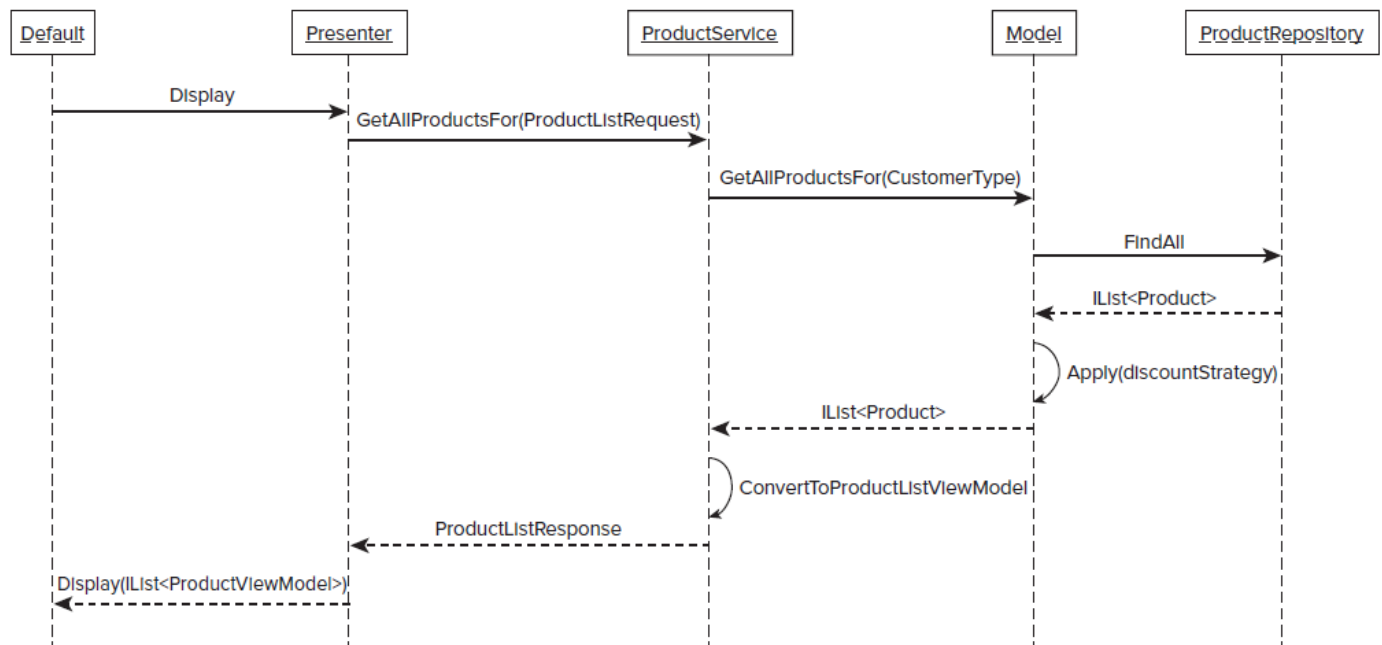
```

        cmbCustomerType.SelectedIndex = 0;
    }
    public void Display(IList<ProductViewModel> products)
    {
        dgvProducts.DataSource = products;
    }
    public CustomerType CustomerType
    {
        get { return (CustomerType)cmbCustomerType.SelectedIndex; }
    }
    public string ErrorMessage
    {
        set
        {
            MessageBox.Show(
                String.Format("Error:{0}{1}", Environment.NewLine, value));
        }
    }
}
}
}

```

با توجه به کد فوق، نمونه ای را از کلاس ProductListPresenter، در لحظه‌ی نمونه سازی اولیه‌ی کلاس UI، ایجاد نمودیم. با استفاده از متد ObjectFactory.GetInstance مربوط به StructureMap، نمونه ای از کلاس ProductService ایجاد شده است و به سازنده‌ی کلاس ProductListPresenter ارسال گردیده است. در مورد Structuremap در مباحث بعدی با جزئیات بیشتری صحبت خواهیم کرد. پیاده سازی معماری لایه بندی در اینجا به پایان رسید.

اما اصلاً نگران نباشید، شما فقط پرواز کوتاه و مختصری را بر فراز کدهای معماری لایه بندی داشته اید که این فقط یک دید کلی را به شما در مورد این معماری داده است. این معماری هنوز جای زیادی برای کار دارد، اما در حال حاضر شما یک Application با پیوند ضعیف (Loosely Coupled) بین لایه‌ها دارید که دارای قابلیت تست پذیری قوی، نگهداری و پشتیبانی آسان و تفکیک پذیری قدرتمند بین اجزای آن می‌باشد. شکل زیر تعامل بین لایه‌ها و وظایف هر یک از آنها را نمایش می‌دهد.



نظرات خوانندگان

نویسنده: حامد

تاریخ: ۱۴:۲۹ ۱۳۹۲/۰۱/۰۳

ممنون از مقاله خوبتون

به نظر شما امکانش هست برای این معماری یک generator بسازیم به طوری که فقط ما تمام جداول دیتابیس و رابطه‌ی آنها را بسازیم و بعد این generator از روی اون تمام لایه‌ها را بر اساس آن بسازه و بعد ما صرفا جاهایی که نیاز به جزییات داره را کامل کنیم

آیا نمونه ای از این برنامه‌ها هست که این معماری یا معماری‌های مشابه را بسازه؟

نویسنده: شاهین کیاست

تاریخ: ۱۵:۳۱ ۱۳۹۲/۰۱/۰۳

اگر با T4 آشنایی داشته باشید بر اساس هر قالبی می‌توانید کد تولید کنید.

نویسنده: حامد

تاریخ: ۱۶:۳۶ ۱۳۹۲/۰۱/۰۳

متأسفانه آشنایی ندارم میشه یه توضیح مختصر بدین و یا منبع معرفی کنید

نویسنده: محسن

تاریخ: ۲۳:۵۹ ۱۳۹۲/۰۱/۰۳

چون اینجا بحث طراحی مطرح شده یک اصل رو در برنامه‌های وب باید رعایت کرد:

هیچ وقت متن خطای حاصل رو به کاربر نمایش ندید (از لحاظ امنیتی). فقط به ذکر عبارت خطایی رخ داده بسنده کنید. خطا رو مثلا توسط ELMAH لاگ کنید برای بررسی بعدی برنامه نویس.

نویسنده: شاهین کیاست

تاریخ: ۱۰:۲۰ ۱۳۹۲/۰۱/۰۴

<http://codepanic.blogspot.com/2012/03/t4-enum.html>

نویسنده: M.Q

تاریخ: ۲۲:۱۵ ۱۳۹۲/۰۱/۰۴

دوست عزیز غیر از ELMAH ابزار دیگری برای لاگ گیری از خطاها وجود دارد که قابل اعتماد باشد؟

همچنین اگر ابزاری جهت لاگ گیری از عملیات کاربران (CRUD => حالا R خیلی مهم نیست) می‌شناسید معرفی نمائید.

با سپاس

نویسنده: محسن

تاریخ: ۰:۳۳ ۱۳۹۲/۰۱/۰۵

متد auditFields مطرح شده در [مطلب ردیابی اطلاعات](#) این سایت برای مقصود شما مناسب است.

نویسنده: صابر فتح الهی
تاریخ: ۱۴:۲۳ ۱۳۹۲/۰۱/۱۲

سلام با تشکر از شما
من نفهمیدم که توی ASP.NET MVC شما چگونه از الگوی MVP استفاده کردین؟
ظاهرا مثال این قسمت هم توی پست وجود نداره، اگر اشتباه می‌کنم لطفا تصحیح بفرمایید.

نویسنده: علی
تاریخ: ۱۶:۳ ۱۳۹۲/۰۱/۱۲

مثال وب فرم هست. page load و post back داره.

نویسنده: شاهین کیاست
تاریخ: ۱۶:۴ ۱۳۹۲/۰۱/۱۲

اگر توجه کنید از الگوی MVP در Web Forms استفاده شده و نه در MVC.

نویسنده: صابر فتح الهی
تاریخ: ۱۸:۳۰ ۱۳۹۲/۰۱/۱۲

آقای کیاست و علی آقا
می‌دونم که پروژه چی هست، یکی از UIهای ما قرار بود MVC باشه خواستم بدونم چطور می‌خوان استفاده کنن، اینجا (در این پست) که می‌دونم ASP.NET Web form هست و در MVC می‌دونم که Page_Load .. وجود نداره سوال من چیز دیگه بود دوستان

نویسنده: شاهین کیاست
تاریخ: ۱۸:۴۴ ۱۳۹۲/۰۱/۱۲

شما گفتید:

سلام با تشکر از شما
من نفهمیدم که توی ASP.NET MVC شما چگونه از الگوی MVP استفاده کردین؟
ظاهرا مثال این قسمت هم توی پست وجود نداره، اگر اشتباه می‌کنم لطفا تصحیح بفرمایید.
با خواندن کامنت شما برداشت کردم شما تصور کردید کدهای پست جاری مربوط به تکنولوژی ASP.NET MVC هست.

به نظر نویسنده هنوز برای MVC و WPF مثال‌ها را ایجاد نکرده و توضیح نداده اند.
اما برای استفاده از این نوع معماری در MVC کار خاصی لازم نیست انجام شود. همانطور که قبلا در مثال‌های آقای نصیری دیده ایم کافی است Service Layer در Controller مدل مناسب را تغذیه کند و برای View فراهم کند.

نویسنده: صابر فتح الهی
تاریخ: ۱۹:۲۶ ۱۳۹۲/۰۱/۱۲

من هم با توجه به مثال آقای نصیری و استفاده از الگوی کار گیج شدم، این معماری یک لایه Repository دارد، من الگوی کار توی این لایه پیاده کردم، با پیاده سازی در این لایه به نظر میاد لایه سرویس کاربردی از دست می‌ده توی پست‌های قبل هم از آقای خوشبخت سوال کردم اما ظاهرا هنوز وقت نکردن پاسخ بدن.

مورد دوم اینکه در این پست الگوی کار شرح داده شده و پیاده سازی شده، و در این پست گفته شده "حین استفاده از EF code first، الگوی واحد کار، همان DbContext است و الگوی مخزن، همان DbSet ها. ضرورتی به ایجاد یک لایه محافظ اضافی بر روی

این‌ها وجود ندارد. " با توجه به این مسائل کلا مسائل قاطی کردم متاسفانه آقای نصیری هم سرشون شلوغ و درگیر [دوره ها](#) است، که بحثی بر سر این معماری بشه.

نویسنده: شاهین کیاست
تاریخ: ۲۰:۴۶ ۱۳۹۲/۰۱/۱۲

روشی که در مثال آقای نصیری گفته شده با روش این سری مقالات کمی متفاوت هست. در آنجا از روکش اضافه برای Repository استفاده نشده همچنین از الگوی واحد کار استفاده شده. به علاوه این سری مقالات ممکن است هنوز تکمیل نشده باشند. به نظر من هر کس با توجه به میزان اطلاعاتی که دارد و درکی که از الگوها دارد با مقایسه‌ی روش‌ها و مقالات می‌تواند تصمیم بگیرد چه معماری به کار بگیرد.

نویسنده: صابر فتح الهی
تاریخ: ۲۱:۳ ۱۳۹۲/۰۱/۱۲

حرف شما کاملا متین هست

من قبلا معماری سه لایه کار می‌کردم، که نمونه اون توی همین سایت [بخش پروژه ها](#) گذاشتم، اما الان با EF , MVC کمی به مشکل بر خوردم و درست نتونستم تا حالا لایه‌های مورد نظر برای خودم در پروژه‌ها تفکیک کنم، این معماری به نظرم جالب اومد، خواستم که الگوی کار هم توی اون به کار ببرم که به مشکل بر خوردم (چون درک درستی از الگوی کار پیدا نکردم یا شایدم کلا دارم اشتباه می‌کنم). البته به قول شما شاید این معماری هنوز تکمیل نشده پروژه اش، در هر صورت از پاسخ‌های شما متشکرم.

نویسنده: شاهین کیاست
تاریخ: ۲۱:۷ ۱۳۹۲/۰۱/۱۲

خواهش می‌کنم.
فقط جهت یادآوری [مثال](#) روش آقای نصیری با پوشش MVC و EF قابل دریافت است.

نویسنده: ابوالفضل روشن ضمیر
تاریخ: ۱:۴۵ ۱۳۹۲/۰۱/۱۷

سلام
با تشکر فروان از شما ...
اگر امکان داره این مثال که در قالی یک پروژه نوشته شده برای دانلود قرار دهید ... تا بهتر بتوانیم برنامه را تجزیه و تحلیل کنیم
.... ممنون

نویسنده: فرشید علی اکبری
تاریخ: ۱۵:۵۳ ۱۳۹۲/۰۱/۱۹

با سلام و تشکر از زحمات کلیه دوستان
با زحمتی که آقای خوشبخت تا اینجا کشیدن فکر کنم در صورتیکه خودمون مقاله مربوطه به این پروژه رو قدم به قدم بخونیم و طراحی کنیم خیلی بهتر متوجه میشیم تا اینکه اونو آماده دانلود کنیم. من با این روش پیش رفتم و برای ایجاد اون با step by step کردن مراحلش حدود 45 دقیقه وقت گذاشتم ولی درصد یادگیری خیلی بالاتر بود تا گرفتن فایل آماده...
درضمن لازمه بگم که بخاطر رفع شک و شبهه در سرعت پردازش و بالا اومدن اطلاعات، من تست این روش رو با تعداد 155 هزار رکورد انجام دادم که کمتر از سه ثانیه برام لود شد... باوجودیکه کامپوننت‌های دات نت بار مختلفی رو هم روی فرم قرار دادم که بیشتر به اهمیت لود اطلاعاتم در پروژه و فرمهای واقعی پی ببرم.
سؤال اینکه :

به نظر شما ما می‌تونیم روی این لایه‌ها الگوی واحد کار رو هم ایجاد کنیم یا نه؟ اصلا ضرورتی داره ؟

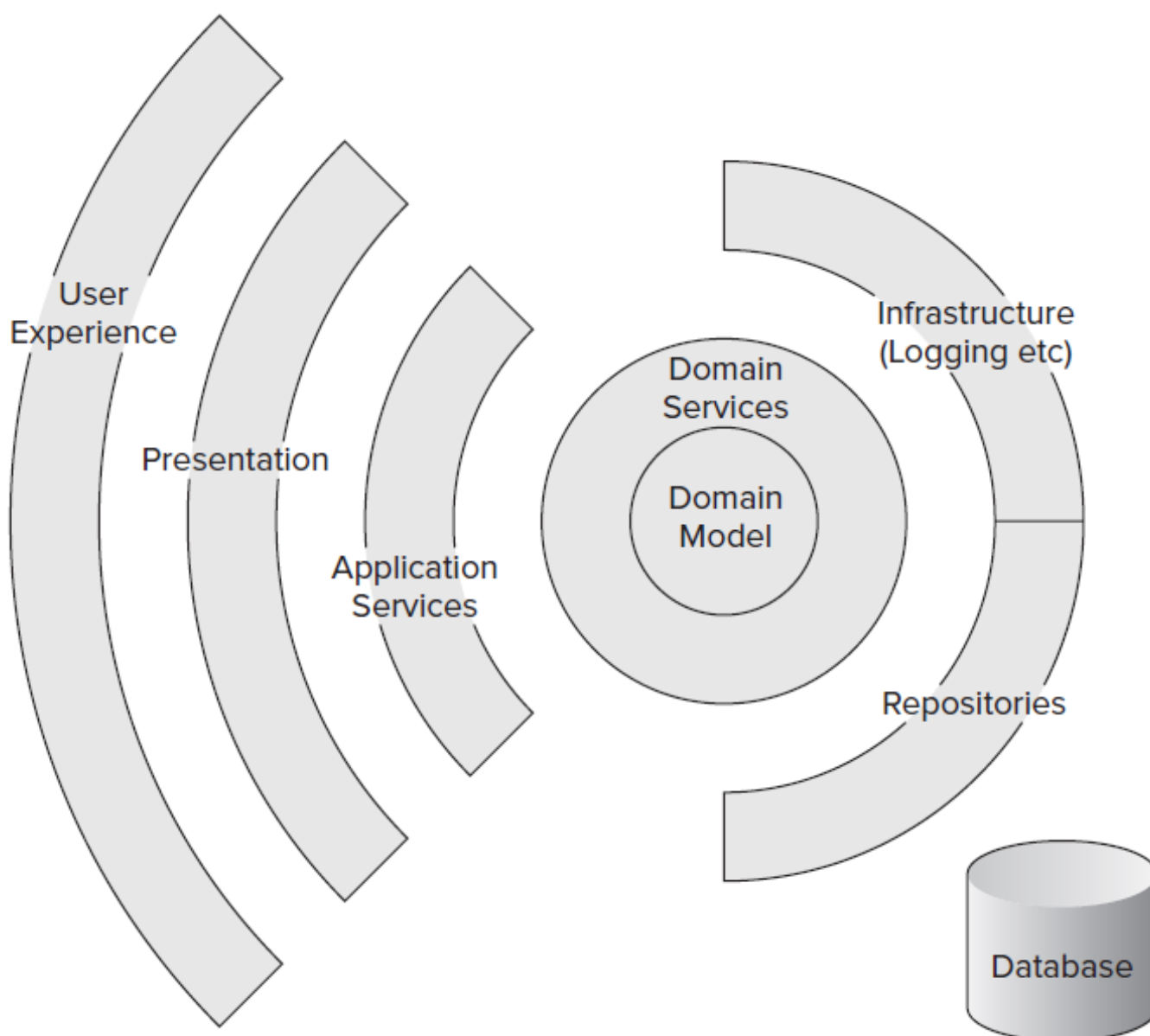
نویسنده: علیرضا کیانی مقدم
تاریخ: ۱۳۹۲/۰۱/۲۸ ۱۲:۸

با تشکر از نویسنده مقاله و اهتمام ایشان به بررسی دقیق مفاهیم ،
از آنجا که flexible و reusable بودن برنامه‌ها را نمی‌توان نادیده گرفت تا آنجا که این تفکیک پذیری خود به مسئله ای بگرنج تبدیل نشده و تکرر داده‌ها و پاس دادن غیر ضرور آنها را موجب نشود تلاش در این باره مفید خواهد بود .
امروزه توسعه دهنده گان به سمت کم کردن لایه‌های فرسایشی و حذف پیچیدگی‌های غیر ضرور قدم بر می‌دارند. خلق عبارات لامبادا در دات نت و delegate ها نمونه هایی از تلاش بشر برنامه نویس در این باره است .

نویسنده: مسعود 2
تاریخ: ۱۳۹۲/۰۲/۰۹ ۹:۱۲

سلام

business Rule 1ها و validation-2ها در کجای این معماری اعمال میشوند؟



منظور از DomainService چیست؟

ممکنه منابع بیشتری معرفی نمایید؟
ممنون.

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۲/۰۹ ۱۶:۲۶

[منبع برای مطالعه بیشتر](#)

نویسنده: محمدرضاتقی پور
تاریخ: ۱۳۹۴/۰۶/۰۷ ۱۵:۵

سلام
مرحله آخر که قراره اطلاعات را به کاربر نشون بدیم
اگر پروژه UI از نوع MCV باشه
چه تفاوتی خواهد کرد؟
مرسی

تمام اپلیکیشن ها را نمی توان در یک پروسس بسته بندی کرد، بدین معنا که تمام اپلیکیشن روی یک سرور فیزیکی قرار گیرد. در عصر حاضر معماری بسیاری از اپلیکیشن ها چند لایه است و هر لایه روی سرور مجزایی توزیع می شود. بعنوان مثال یک معماری کلاسیک شامل سه لایه نمایش (presentation)، اپلیکیشن (application) و داده (data) است. لایه بندی منطقی (logical layering) یک اپلیکیشن می تواند در یک App Domain واحد پیاده سازی شده و روی یک کامپیوتر میزبانی شود. در این صورت لازم نیست نگران مباحثی مانند پراکسی ها، مرتب سازی (serialization)، پروتوکل های شبکه و غیره باشیم. اما اپلیکیشن های بزرگی که چندین کلاینت دارند و در مراکز داده میزبانی می شوند باید تمام این مسائل را در نظر بگیرند. خوشبختانه پیاده سازی چنین اپلیکیشن هایی با استفاده از Entity Framework و دیگر تکنولوژی های مایکروسافت مانند WCF, Web API ساده تر شده است. منظور از n-Tier معماری اپلیکیشن هایی است که لایه های نمایش، منطق تجاری و دسترسی داده هر کدام روی سرور مجزایی میزبانی می شوند. این تفکیک فیزیکی لایه ها به بسط پذیری، مدیریت و نگهداری اپلیکیشن ها در دراز مدت کمک می کند، اما معمولاً تأثیری منفی روی کارایی کلی سیستم دارد. چرا که برای انجام عملیات مختلف باید از محدوده ماشین های فیزیکی عبور کنیم.

معماری N-Tier چالش های بخصوصی را برای قابلیت های change-tracking در EF اضافه می کند. در ابتدا داده ها توسط یک آبجکت EF Context بارگذاری می شوند اما این آبجکت پس از ارسال داده ها به کلاینت از بین می رود. تغییراتی که در سمت کلاینت روی داده ها اعمال می شوند ردیابی (track) نخواهند شد. هنگام بروز رسانی، آبجکت Context جدیدی برای پردازش اطلاعات ارسالی باید ایجاد شود. مسلماً آبجکت جدید هیچ چیز درباره Context پیشین یا مقادیر اصلی موجودیت ها نمی داند.

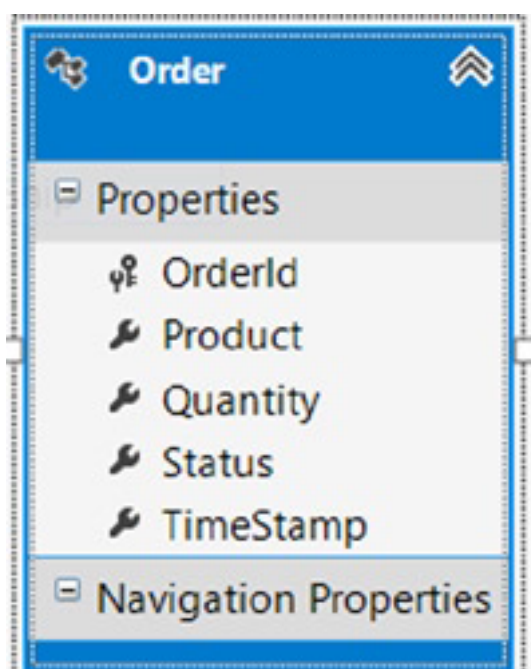
در نسخه های قبلی Entity Framework توسعه دهندگان با استفاده از قالب ویژه ای بنام Self-Tracking Entities می توانستند تغییرات موجودیت ها را ردیابی کنند. این قابلیت در نسخه EF 6 از رده خارج شده است و گرچه هنوز توسطObjectContext پشتیبانی می شود، آبجکت DbContext از آن پشتیبانی نمی کند.

در این سری از مقالات روی عملیات پایه CRUD تمرکز می کنیم که در اکثر اپلیکیشن های n-Tier استفاده می شوند. همچنین خواهیم دید چگونه می توان تغییرات موجودیت ها را ردیابی کرد. مباحثی مانند همزمانی (concurrency) و مرتب سازی (serialization) نیز بررسی خواهند شد. در قسمت یک این سری مقالات، به بروز رسانی موجودیت های منفصل (disconnected) توسط سرویس های Web API نگاهی خواهیم داشت.

بروز رسانی موجودیت های منفصل با Web API

سناریویی را فرض کنید که در آن برای انجام عملیات CRUD از یک سرویس Web API استفاده می شود. همچنین مدیریت داده ها با مدل Code-First پیاده سازی شده است. در مثال جاری یک کلاینت Console Application خواهیم داشت که یک سرویس Web API را فراخوانی می کند. توجه داشته باشید که هر اپلیکیشن در Solution مجزایی قرار دارد. تفکیک پروژه ها برای شبیه سازی یک محیط n-Tier انجام شده است.

فرض کنید مدلی مانند تصویر زیر داریم.



همانطور که می بینید مدل جاری، سفارشات یک اپلیکیشن فرضی را معرفی می کند. می خواهیم مدل و کد دسترسی به داده ها را در یک سرویس Web API پیاده سازی کنیم، تا هر کلاینتی که از HTTP استفاده می کند بتواند عملیات CRUD را انجام دهد. برای ساختن سرویس مورد نظر مراحل زیر را دنبال کنید.

در ویژوال استودیو پروژه جدیدی از نوع ASP.NET Web Application بسازید و قالب پروژه را Web API انتخاب کنید. نام پروژه را به Recipe1.Service تغییر دهید.

کنترلر جدیدی از نوع WebApi Controller با نام OrderController به پروژه اضافه کنید.

کلاس جدیدی با نام Order در پوشه مدل ها ایجاد کنید و کد زیر را به آن اضافه نمایید.

```
public class Order
{
    public int OrderId { get; set; }
    public string Product { get; set; }
    public int Quantity { get; set; }
    public string Status { get; set; }
    public byte[] TimeStamp { get; set; }
}
```

با استفاده از NuGet Package Manager کتابخانه Entity Framework 6 را به پروژه اضافه کنید.

حال کلاسی با نام Recipe1Context ایجاد کنید و کد زیر را به آن اضافه نمایید.

```
public class Recipe1Context : DbContext
{
    public Recipe1Context() : base("Recipe1ConnectionString") { }

    public DbSet<Order> Orders { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Order>().ToTable("Orders");
        // Following configuration enables timestamp to be concurrency token
        modelBuilder.Entity<Order>().Property(x => x.TimeStamp)
            .IsConcurrencyToken()
            .HasDatabaseGeneratedOption(DatabaseGeneratedOption.Computed);
    }
}
```

فایل Web.config پروژه را باز کنید و رشته اتصال زیر را به قسمت ConnectionStrings اضافه نمایید.

```
<connectionStrings>
  <add name="Recipe1ConnectionString"
    connectionString="Data Source=.;
    Initial Catalog=EFRecipes;
    Integrated Security=True;
    MultipleActiveResultSets=True"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

فایل Global.asax را باز کنید و کد زیر را به آن اضافه نمایید. این کد بررسی Entity Framework Compatibility را غیرفعال می‌کند.

```
protected void Application_Start()
{
    // Disable Entity Framework Model Compatibility
    Database.SetInitializer<Recipe1Context>(null);
    ...
}
```

در آخر کد کنترلر Order را با لیست زیر جایگزین کنید.

```
public class OrderController : ApiController
{
    // GET api/order
    public IEnumerable<Order> Get()
    {
        using (var context = new Recipe1Context())
        {
            return context.Orders.ToList();
        }
    }

    // GET api/order/5
    public Order Get(int id)
    {
        using (var context = new Recipe1Context())
        {
            return context.Orders.FirstOrDefault(x => x.OrderId == id);
        }
    }

    // POST api/order
    public HttpResponseMessage Post(Order order)
    {
        // Cleanup data from previous requests
        Cleanup();

        using (var context = new Recipe1Context())
        {
            context.Orders.Add(order);
            context.SaveChanges();
            // create HttpResponseMessage to wrap result, assigning Http Status code of 201,
            // which informs client that resource created successfully
            var response = Request.CreateResponse(HttpStatusCode.Created, order);
            // add location of newly-created resource to response header
            response.Headers.Location = new Uri(Url.Link("DefaultApi",
                new { id = order.OrderId }));
            return response;
        }
    }

    // PUT api/order/5
    public HttpResponseMessage Put(Order order)
    {
        using (var context = new Recipe1Context())
        {
            context.Entry(order).State = EntityState.Modified;
            context.SaveChanges();
            // return Http Status code of 200, informing client that resource updated successfully
            return Request.CreateResponse(HttpStatusCode.OK, order);
        }
    }

    // DELETE api/order/5
```

```

public HttpResponseMessage Delete(int id)
{
    using (var context = new Recipe1Context())
    {
        var order = context.Orders.FirstOrDefault(x => x.OrderId == id);
        context.Orders.Remove(order);
        context.SaveChanges();
        // Return Http Status code of 200, informing client that resource removed successfully
        return Request.CreateResponse(HttpStatusCode.OK);
    }
}

private void Cleanup()
{
    using (var context = new Recipe1Context())
    {
        context.Database.ExecuteSqlCommand("delete from [orders]");
    }
}
}

```

قابل ذکر است که هنگام استفاده از Entity Framework در MVC یا Web API، بکارگیری قابلیت Scaffolding بسیار مفید است. این فریم ورک های ASP.NET می توانند کنترلر هایی کاملاً اجرایی برایتان تولید کنند که صرفه جویی چشمگیری در زمان و کار شما خواهد بود.

در قدم بعدی اپلیکیشن کلاینت را می سازیم که از سرویس Web API استفاده می کند.

در ویژوال استودیو پروژه جدیدی از نوع Console Application بسازید و نام آن را به Recipe1.Client تغییر دهید. کلاس موجودیت Order را به پروژه اضافه کنید. همان کلاسی که در سرویس Web API ساختیم.

نکته: قسمت هایی از اپلیکیشن که باید در لایه های مختلف مورد استفاده قرار گیرند - مانند کلاس های موجودیت ها - بهتر است در لایه مجزایی قرار داده شده و به اشتراک گذاشته شوند. مثلاً می توانید پروژه ای از نوع Class Library بسازید و تمام موجودیت ها را در آن تعریف کنید. سپس لایه های مختلف این پروژه را ارجاع خواهند کرد.

فایل program.cs را باز کنید و کد زیر را به آن اضافه نمایید.

```

private HttpClient _client;
private Order _order;

private static void Main()
{
    Task t = Run();
    t.Wait();

    Console.WriteLine("\nPress <enter> to continue...");
    Console.ReadLine();
}

private static async Task Run()
{
    // create instance of the program class
    var program = new Program();
    program.ServiceSetup();
    program.CreateOrder();
    // do not proceed until order is added
    await program.PostOrderAsync();
    program.ChangeOrder();
    // do not proceed until order is changed
    await program.PutOrderAsync();
    // do not proceed until order is removed
    await program.RemoveOrderAsync();
}

private void ServiceSetup()
{
    // map URL for Web API call
    _client = new HttpClient { BaseAddress = new Uri("http://localhost:3237/") };
    // add Accept Header to request Web API content
}

```



```
// negotiation to return resource in JSON format
_client.DefaultRequestHeaders.Accept.
    Add(new MediaTypeWithQualityHeaderValue("application/json"));
}

private void CreateOrder()
{
    // Create new order
    _order = new Order { Product = "Camping Tent", Quantity = 3, Status = "Received" };
}

private async Task PostOrderAsync()
{
    // leverage Web API client side API to call service
    var response = await _client.PostAsJsonAsync("api/order", _order);
    Uri newOrderUri;

    if (response.IsSuccessStatusCode)
    {
        // Capture Uri of new resource
        newOrderUri = response.Headers.Location;
        // capture newly-created order returned from service,
        // which will now include the database-generated Id value
        _order = await response.Content.ReadAsAsync<Order>();
        Console.WriteLine("Successfully created order. Here is URL to new resource: {0}",
newOrderUri);
    }
    else
        Console.WriteLine("{0} ({1})", (int)response.StatusCode, response.ReasonPhrase);
}

private void ChangeOrder()
{
    // update order
    _order.Quantity = 10;
}

private async Task PutOrderAsync()
{
    // construct call to generate HttpPut verb and dispatch
    // to corresponding Put method in the Web API Service
    var response = await _client.PutAsJsonAsync("api/order", _order);

    if (response.IsSuccessStatusCode)
    {
        // capture updated order returned from service, which will include new quantity
        _order = await response.Content.ReadAsAsync<Order>();
        Console.WriteLine("Successfully updated order: {0}", response.StatusCode);
    }
    else
        Console.WriteLine("{0} ({1})", (int)response.StatusCode, response.ReasonPhrase);
}

private async Task RemoveOrderAsync()
{
    // remove order
    var uri = "api/order/" + _order.OrderId;
    var response = await _client.DeleteAsync(uri);

    if (response.IsSuccessStatusCode)
        Console.WriteLine("Sucessfully deleted order: {0}", response.StatusCode);
    else
        Console.WriteLine("{0} ({1})", (int)response.StatusCode, response.ReasonPhrase);
}
```

اگر اپلیکیشن کلاینت را اجرا کنید باید با خروجی زیر مواجه شوید:

Successfully created order: http://localhost:3237/api/order/1054

Successfully updated order: OK

Sucessfully deleted order: OK

شرح مثال جاری

با اجرای اپلیکیشن Web API شروع کنید. این اپلیکیشن یک کنترلر Web API دارد که پس از اجرا شما را به صفحه خانه هدایت می‌کند. در این مرحله اپلیکیشن در حال اجرا است و سرویس‌های ما قابل دسترسی هستند.

حال اپلیکیشن کنسول را باز کنید. روی خط اول کد program.cs یک breakpoint تعریف کرده و اپلیکیشن را اجرا کنید. ابتدا آدرس سرویس Web API را پیکربندی کرده و خاصیت Accept Header را مقدار دهی می‌کنیم. با این کار از سرویس مورد نظر درخواست می‌کنیم که داده‌ها را با فرمت JSON بازگرداند. سپس یک آبجکت Order می‌سازیم و با فراخوانی متد PostAsJsonAsync آن را به سرویس ارسال می‌کنیم. این متد روی آبجکت HttpClient تعریف شده است. اگر به اکشن متد Post در کنترلر Order یک breakpoint اضافه کنید، خواهید دید که این متد سفارش جدید را بعنوان یک پارامتر دریافت می‌کند و آن را به لیست موجودیت‌ها در Context جاری اضافه می‌نماید. این عمل باعث می‌شود که آبجکت جدید بعنوان Added علامت گذاری شود، در این مرحله Context جاری شروع به ردیابی تغییرات می‌کند. در آخر با فراخوانی متد SaveChanges داده‌ها را ذخیره می‌کنیم. در قدم بعدی کد وضعیت 201 (Created) و آدرس منبع جدید را در یک آبجکت HttpResponseMessage قرار می‌دهیم و به کلاینت ارسال می‌کنیم. هنگام استفاده از Web API باید اطمینان حاصل کنیم که کلاینت‌ها درخواست‌های ایجاد رکورد جدید را بصورت POST ارسال می‌کنند. درخواست‌های HTTP Post بصورت خودکار به اکشن متد متناظر نگاشت می‌شوند.

در مرحله بعد عملیات بعدی را اجرا می‌کنیم، تعداد سفارش را تغییر می‌دهیم و موجودیت جاری را با فراخوانی متد PutAsJsonAsync به سرویس Web API ارسال می‌کنیم. اگر به اکشن متد Put در کنترلر سرویس یک breakpoint اضافه کنید، خواهید دید که آبجکت سفارش بصورت یک پارامتر دریافت می‌شود. سپس با فراخوانی متد Entry و پاس دادن موجودیت جاری بعنوان رفرنس، خاصیت State را به Modified تغییر می‌دهیم، که این کار موجودیت را به Context جاری می‌چسباند. حال فراخوانی متد SaveChanges یک اسکریپت بروز رسانی تولید خواهد کرد. در مثال جاری تمام فیلدهای آبجکت Order را بروز رسانی می‌کنیم. در شماره‌های بعدی این سری از مقالات، خواهیم دید چگونه می‌توان تنها فیلدهایی را بروز رسانی کرد که تغییر کرده اند. در آخر عملیات را با بازگرداندن کد وضعیت 200 (OK) به اتمام می‌رسانیم.

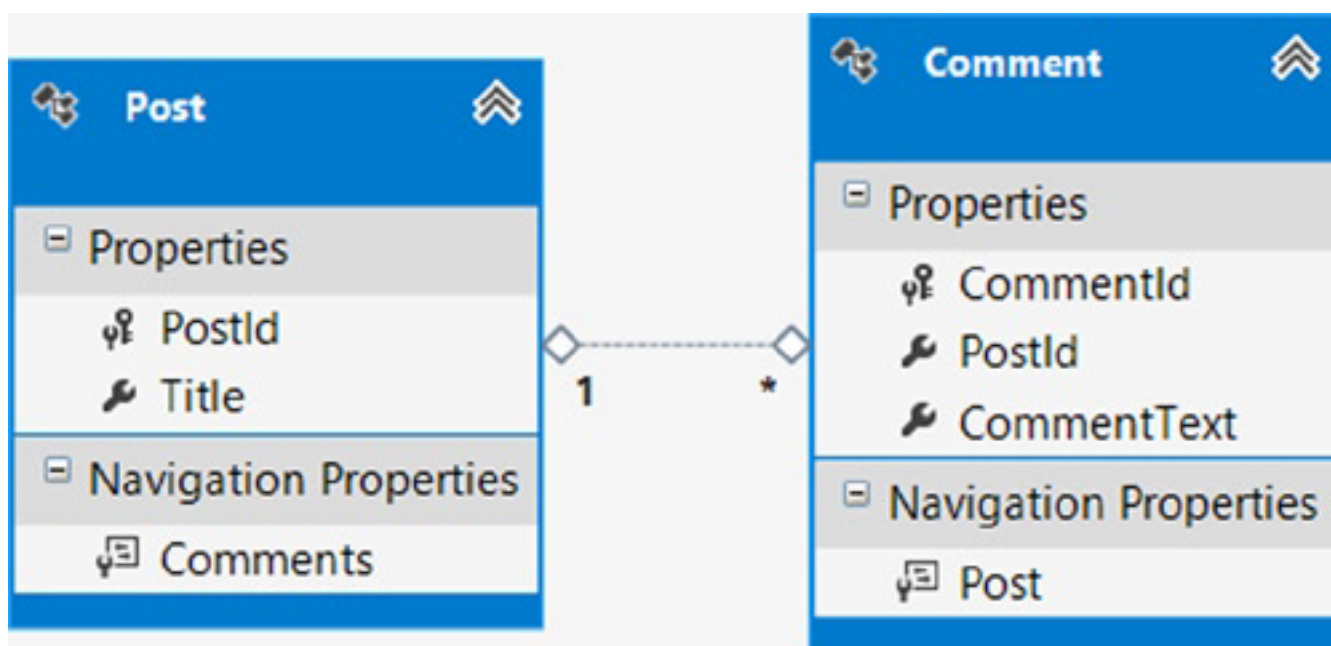
در مرحله بعد، عملیات نهایی را اجرا می‌کنیم که موجودیت Order را از منبع داده حذف می‌کند. برای اینکار شناسه (Id) رکورد مورد نظر را به آدرس سرویس اضافه می‌کنیم و متد DeleteAsync را فراخوانی می‌کنیم. در سرویس Web API رکورد مورد نظر را از دیتابیس دریافت کرده و متد Remove را روی Context جاری فراخوانی می‌کنیم. این کار موجودیت مورد نظر را بعنوان Deleted علامت گذاری می‌کند. فراخوانی متد SaveChanges یک اسکریپت Delete تولید خواهد کرد که نهایتاً منجر به حذف شدن رکورد می‌شود.

در یک اپلیکیشن واقعی بهتر است کد دسترسی داده‌ها از سرویس Web API تفکیک شود و در لایه مجزایی قرار گیرد.

در [قسمت قبل](#) معماری اپلیکیشن های N-Tier و بروز رسانی موجودیت های منفصل توسط Web API را بررسی کردیم. در این قسمت بروز رسانی موجودیت های منفصل توسط WCF را بررسی می کنیم.

بروز رسانی موجودیت های منفصل توسط WCF

سناریویی را در نظر بگیرید که در آن عملیات CRUD توسط WCF پیاده سازی شده اند و دسترسی داده ها با مدل Code-First انجام می شود. فرض کنید مدل اپلیکیشن مانند تصویر زیر است.



همانطور که می بینید مدل ما متشکل از پست ها و نظرات کاربران است. برای ساده نگاه داشتن مثال جاری، اکثر فیلدها حذف شده اند. مثلاً متن پست ها، نویسنده، تاریخ و زمان انتشار و غیره. می خواهیم تمام کد دسترسی داده ها را در یک سرویس WCF پیاده سازی کنیم تا کلاینت ها بتوانند عملیات CRUD را توسط آن انجام دهند. برای ساختن این سرویس مراحل زیر را دنبال کنید. در ویژوال استودیو پروژه جدیدی از نوع Class Library بسازید و نام آن را به Recipe2 تغییر دهید.

با استفاده از NuGet Package Manager کتابخانه Entity Framework 6 را به پروژه اضافه کنید. سه کلاس با نام های Post، Comment و Recipe2Context به پروژه اضافه کنید. کلاس های Post و Comment موجودیت های مدل ما هستند که به جداول متناظرشان نگاشت می شوند. کلاس Recipe2Context آبجکت DbContext ما خواهد بود که بعنوان درگاه عملیاتی EF عمل می کند. دقت کنید که خاصیت های لازم WCF یعنی DataContract و DataMember در کلاس های موجودیت ها بدرستی استفاده می شوند. لیست زیر کد این کلاس ها را نشان می دهد.

```

[DataContract(IsReference = true)]
public class Post
{
    public Post()
    {
        comments = new HashSet<Comments>();
    }

    [DataMember]
    public int PostId { get; set; }
}
    
```

```
[DataMember]
public string Title { get; set; }
[DataMember]
public virtual ICollection<Comment> Comments { get; set; }
}

[DataContract(IsReference=true)]
public class Comment
{
    [DataMember]
    public int CommentId { get; set; }
    [DataMember]
    public int PostId { get; set; }
    [DataMember]
    public string CommentText { get; set; }
    [DataMember]
    public virtual Post Post { get; set; }
}

public class EFRecipesEntities : DbContext
{
    public EFRecipesEntities() : base("name=EFRecipesEntities") {}

    public DbSet<Post> posts;
    public DbSet<Comment> comments;
}
```

یک فایل App.config به پروژه اضافه کنید و رشته اتصال زیر را به آن اضافه نمایید.

```
<connectionStrings>
  <add name="Recipe2ConnectionString"
    connectionString="Data Source=.;
    Initial Catalog=EFRecipes;
    Integrated Security=True;
    MultipleActiveResultSets=True"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

حال یک پروژه WCF به Solution جاری اضافه کنید. برای ساده نگاه داشتن مثال جاری، نام پیش فرض Service1 را بپذیرید. فایل IService1.cs را باز کنید و کد زیر را با محتوای آن جایگزین نمایید.

```
[ServiceContract]
public interface IService1
{
    [OperationContract]
    void Cleanup();
    [OperationContract]
    Post GetPostByTitle(string title);
    [OperationContract]
    Post SubmitPost(Post post);
    [OperationContract]
    Comment SubmitComment(Comment comment);
    [OperationContract]
    void DeleteComment(Comment comment);
}
```

فایل Service1.svc.cs را باز کنید و کد زیر را با محتوای آن جایگزین نمایید. بیاد داشته باشید که پروژه Recipe2 را ارجاع کنید و فضای نام آن را وارد نمایید. همچنین کتابخانه EF 6 را باید به پروژه اضافه کنید.

```
public class Service1 : IService1
{
    public void Cleanup()
    {
        using (var context = new EFRecipesEntities())
        {
            context.Database.ExecuteSqlCommand("delete from [comments]");
            context.Database.ExecuteSqlCommand("delete from [posts]");
        }
    }

    public Post GetPostByTitle(string title)
```

```

{
    using (var context = new EFRecipesEntities())
    {
        context.Configuration.ProxyCreationEnabled = false;
        var post = context.Posts.Include(p => p.Comments).Single(p => p.Title == title);
        return post;
    }
}

public Post SubmitPost(Post post)
{
    context.Entry(post).State =
        // if Id equal to 0, must be insert; otherwise, it's an update
        post.PostId == 0 ? EntityState.Added : EntityState.Modified;
    context.SaveChanges();
    return post;
}

public Comment SubmitComment(Comment comment)
{
    using (var context = new EFRecipesEntities())
    {
        context.Comments.Attach(comment);
        if (comment.CommentId == 0)
        {
            // this is an insert
            context.Entry(comment).State = EntityState.Added;
        }
        else
        {
            // set single property to modified, which sets state of entity to modified, but
            // only updates the single property - not the entire entity
            context.entry(comment).Property(x => x.CommentText).IsModified = true;
        }
        context.SaveChanges();
        return comment;
    }
}

public void DeleteComment(Comment comment)
{
    using (var context = new EFRecipesEntities())
    {
        context.Entry(comment).State = EntityState.Deleted;
        context.SaveChanges();
    }
}
}

```

در آخر پروژه جدیدی از نوع Windows Console Application به Solution جاری اضافه کنید. از این اپلیکیشن بعنوان کلاینتی برای تست سرویس WCF استفاده خواهیم کرد. فایل program.cs را باز کنید و کد زیر را با محتوای آن جایگزین نمایید. روی نام پروژه کلیک راست کرده و گزینه Add Service Reference را انتخاب کنید، سپس ارجاعی به سرویس Service1 اضافه کنید. رفرنسی هم به کتابخانه کلاس‌ها که در ابتدای مراحل ساختید باید اضافه کنید.

```

class Program
{
    static void Main(string[] args)
    {
        using (var client = new ServiceReference2.Service1Client())
        {
            // cleanup previous data
            client.Cleanup();
            // insert a post
            var post = new Post { Title = "POCO Proxies" };
            post = client.SubmitPost(post);
            // update the post
            post.Title = "Change Tracking Proxies";
            client.SubmitPost(post);
            // add a comment
            var comment1 = new Comment { CommentText = "Virtual Properties are cool!", PostId =
post.PostId };
            var comment2 = new Comment { CommentText = "I use ICollection<T> all the time", PostId =
post.PostId };
            comment1 = client.SubmitComment(comment1);
            comment2 = client.SubmitComment(comment2);
            // update a comment

```

```

        comment1.CommentText = "How do I use ICollection<T>?";
        client.SubmitComment(comment1);
        // delete comment 1
        client.DeleteComment(comment1);
        // get posts with comments
        var p = client.GetPostByTitle("Change Tracking Proxies");
        Console.WriteLine("Comments for post: {0}", p.Title);
        foreach (var comment in p.Comments)
        {
            Console.WriteLine("\tComment: {0}", comment.CommentText);
        }
    }
}

```

اگر اپلیکیشن کلاینت (برنامه کنسول) را اجرا کنید با خروجی زیر مواجه می‌شوید.

Comments for post: Change Tracking Proxies

Comment: I use ICollection<T> all the time

شرح مثال جاری

ابتدا با اپلیکیشن کنسول شروع می‌کنیم، که کلاینت سرویس ما است. نخست در یک بلاک `using {}` وهله ای از کلاینت سرویس مان ایجاد می‌کنیم. درست همانطور که وهله ای از یک EF Context می‌سازیم. استفاده از بلوک‌های `using` توصیه می‌شود چرا که متد `Dispose` بصورت خودکار فراخوانی خواهد شد، چه بصورت عادی چه هنگام بروز خطا. پس از آنکه وهله ای از کلاینت سرویس را در اختیار داشتیم، متد `Cleanup` را صدا می‌زنیم. با فراخوانی این متد تمام داده‌های تست پیشین را حذف می‌کنیم. در چند خط بعدی، متد `SubmitPost` را روی سرویس فراخوانی می‌کنیم. در پیاده سازی فعلی شناسه پست را بررسی می‌کنیم. اگر مقدار شناسه صفر باشد، خاصیت `State` موجودیت را به `Added` تغییر می‌دهید تا رکورد جدیدی ثبت کنیم. در غیر اینصورت فرض بر این است که چنین موجودیتی وجود دارد و قصد ویرایش آن را داریم، بنابراین خاصیت `State` را به `Modified` تغییر می‌دهیم. از آنجا که مقدار متغیرهای `int` بصورت پیش فرض صفر است، با این روش می‌توانیم وضعیت پست‌ها را مشخص کنیم. یعنی تعیین کنیم رکورد جدیدی باید ثبت شود یا رکوردی موجود بروز رسانی گردد. رویکردی بهتر آن است که پارامتری اضافی به متد پاس دهیم، یا متدی مجزا برای ثبت رکوردهای جدید تعریف کنیم. مثلاً رویکردی با نام `InsertPost`. در هر حال، بهترین روش بستگی به ساختار اپلیکیشن شما دارد.

اگر پست جدیدی ثبت شود، خاصیت `PostId` با مقدار مناسب جدید بروز رسانی می‌شود و وهله پست را باز می‌گردانیم. ایجاد و بروز رسانی نظرات کاربران مشابه ایجاد و بروز رسانی پست‌ها است، اما با یک تفاوت اساسی: بعنوان یک قانون، هنگام بروز رسانی نظرات کاربران تنها فیلد متن نظر باید بروز رسانی شود. بنابراین با فیلدهای دیگری مانند تاریخ انتشار و غیره اصلاً کاری نخواهیم داشت. بدین منظور تنها خاصیت `CommentText` را بعنوان علامت گذاری می‌کنیم. این امر منجر می‌شود که Entity Framework عبارتی برای بروز رسانی تولید کند که تنها این فیلد را در بر می‌گیرد. توجه داشته باشید که این روش تنها در صورتی کار می‌کند که بخواهید یک فیلد واحد را بروز رسانی کنید. اگر می‌خواستیم فیلدهای بیشتری را در موجودیت `Comment` بروز رسانی کنیم، باید مکانیزمی برای ردیابی تغییرات در سمت کلاینت در نظر می‌گرفتیم. در مواقعی که خاصیت‌های متعددی می‌توانند تغییر کنند، معمولاً بهتر است کل موجودیت بروز رسانی شود تا اینکه مکانیزمی پیچیده برای ردیابی تغییرات در سمت کلاینت پیاده گردد. بروز رسانی کل موجودیت بهینه‌تر خواهد بود.

برای حذف یک دیدگاه، متد `Entry` را روی آبجکت `DbContext` فراخوانی می‌کنیم و موجودیت مورد نظر را بعنوان آرگومان پاس می‌دهیم. این امر سبب می‌شود که موجودیت مورد نظر بعنوان `Deleted` علامت گذاری شود، که هنگام فراخوانی متد `SaveChanges` اسکریپت لازم برای حذف رکورد را تولید خواهد کرد.

در آخر متد `GetPostByTitle` یک پست را بر اساس عنوان پیدا کرده و تمام نظرات کاربران مربوط به آن را هم بارگذاری می‌کند. از آنجا که ما کلاس‌های POCO را پیاده سازی کرده ایم، Entity Framework آبجکتی را بر می‌گرداند که Dynamic Proxy نامیده می‌شود. این آبجکت پست و نظرات مربوط به آن را در بر خواهد گرفت. متاسفانه WCF نمی‌تواند آبجکت‌های پروکسی را مرتب سازی (serialize) کند. اما با غیرفعال کردن قابلیت ایجاد پروکسی‌ها (`ProxyCreationEnabled=false`) ما به Entity Framework

می‌گوییم که خود آجکت‌های اصلی را بازگرداند. اگر سعی کنید آجکت پروکسی را سریال کنید با پیغام خطای زیر مواجه خواهید شد:

The underlying connection was closed: The connection was closed unexpectedly

می‌توانیم غیرفعال کردن تولید پروکسی را به متد سازنده کلاس سرویس منتقل کنیم تا روی تمام متدهای سرویس اعمال شود.

در این قسمت دیدیم چگونه می‌توانیم از آجکت‌های POCO برای مدیریت عملیات CRUD توسط WCF استفاده کنیم. از آنجا که هیچ اطلاعاتی درباره وضعیت موجودیت‌ها روی کلاینت ذخیره نمی‌شود، متدهایی مجزا برای عملیات CRUD ساختیم. در قسمت‌های بعدی خواهیم دید چگونه می‌توان تعداد متدهایی که سرویس مان باید پیاده سازی کند را کاهش داد و چگونه ارتباطات بین کلاینت و سرور را ساده‌تر کنیم.

نظرات خوانندگان

نویسنده: جلال

تاریخ: ۱۳۹۲/۱۲/۱۰ ۲۰:۲۰

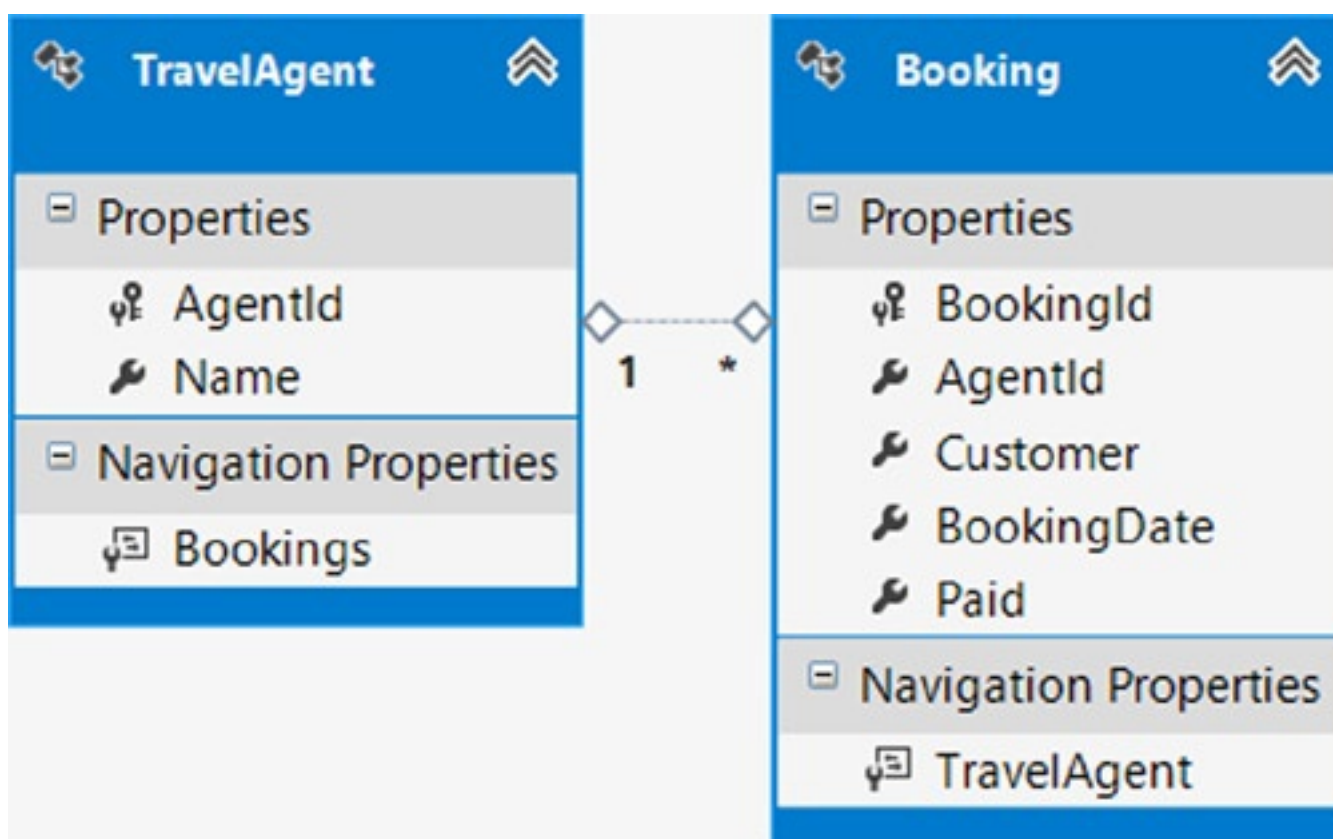
در این سناریو، فرض را بر این گذاشته اید که موجودیتهای جدید هستند و یا ویرایش شده اند و بنابراین حتی اگر یک پست کامنتهایی داشته باشد که ویرایش نشده اند، برای آنها دستور update صادر میشود و این، در مواردیکه تعداد کامنتها(که البته همیشه این موجودیتهای اینگونه به سادگی کامنت نیستند) زیاد باشد، روی کارایی تأثیر منفی خواهد داشت، چه راهی برای تشخیص موجودیتهایی که سمت کلاینت تغییری نکرده اند، پیشنهاد میدهید؟

در [قسمت قبلی](#) بروز رسانی موجودیت های منفصل با WCF را بررسی کردیم. در این قسمت خواهیم دید چگونه می توان تغییرات موجودیت ها را تشخیص داد و عملیات CRUD را روی یک Object Graph اجرا کرد.

تشخیص تغییرات با Web API

فرض کنید می خواهیم از سرویس های Web API برای انجام عملیات CRUD استفاده کنیم، اما بدون آنکه برای هر موجودیت متدهایی مجزا تعریف کنیم. به بیان دیگر می خواهیم عملیات مذکور را روی یک Object Graph انجام دهیم. مدیریت داده ها هم با مدل Code-First پیاده سازی می شود. در مثال جاری یک اپلیکیشن کنسول خواهیم داشت که بعنوان یک کلاینت سرویس را فراخوانی می کند. هر پروژه نیز در Solution مجزایی قرار دارد، تا یک محیط n-Tier را شبیه سازی کنیم.

مدل زیر را در نظر بگیرید.



همانطور که می بینید مدل ما آژانس های مسافرتی و رزرواسیون آنها را ارائه می کند. می خواهیم مدل و کد دسترسی داده ها را در یک سرویس Web API پیاده سازی کنیم تا هر کلاینتی که به HTTP دسترسی دارد بتواند عملیات CRUD را انجام دهد. برای ساختن سرویس مورد نظر مراحل زیر را دنبال کنید:

در ویژوال استودیو پروژه جدیدی از نوع ASP.NET Web Application بسازید و قالب پروژه را Web API انتخاب کنید. نام پروژه را به Recipe3.Service تغییر دهید.

کنترلر جدیدی بنام TravelAgentController به پروژه اضافه کنید.

دو کلاس جدید با نام های TravelAgent و Booking بسازید و کد آنها را مطابق لیست زیر تغییر دهید.

```
public class TravelAgent
{
    public TravelAgent()
    {
        this.Bookings = new HashSet<Booking>();
    }

    public int AgentId { get; set; }
    public string Name { get; set; }
    public virtual ICollection<Booking> Bookings { get; set; }
}

public class Booking
{
    public int BookingId { get; set; }
    public int AgentId { get; set; }
    public string Customer { get; set; }
    public DateTime BookingDate { get; set; }
    public bool Paid { get; set; }
    public virtual TravelAgent TravelAgent { get; set; }
}
```

با استفاده از NuGet Package Manager کتابخانه Entity Framework 6 را به پروژه اضافه کنید.

کلاس جدیدی بنام Recipe3Context بسازید و کد آن را مطابق لیست زیر تغییر دهید.

```
public class Recipe3Context : DbContext
{
    public Recipe3Context() : base("Recipe3ConnectionString") { }
    public DbSet<TravelAgent> TravelAgents { get; set; }
    public DbSet<Booking> Bookings { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<TravelAgent>().HasKey(x => x.AgentId);
        modelBuilder.Entity<TravelAgent>().ToTable("TravelAgents");
        modelBuilder.Entity<Booking>().ToTable("Bookings");
    }
}
```

فایل Web.config پروژه را باز کنید و رشته اتصال زیر را به قسمت ConnectionStrings اضافه کنید.

```
<connectionStrings>
  <add name="Recipe3ConnectionString"
    connectionString="Data Source=.;
    Initial Catalog=EFRecipes;
    Integrated Security=True;
    MultipleActiveResultSets=True"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

فایل Global.asax را باز کنید و کد زیر را به متد Application_Start اضافه نمایید. این کد بررسی Model Compatibility در EF را غیرفعال می کند. همچنین به JSON serializer می گوئیم که self-referencing loop خاصیت های پیمایشی را نادیده بگیرد. این حلقه بدلیل ارتباط bidirectional بین موجودیت ها بوجود می آید.

```
protected void Application_Start()
{
    // Disable Entity Framework Model Compatibility
    Database.SetInitializer<Recipe1Context>(null);

    // The bidirectional navigation properties between related entities
    // create a self-referencing loop that breaks Web API's effort to
    // serialize the objects as JSON. By default, Json.NET is configured
    // to error when a reference loop is detected. To resolve problem,
    // simply configure JSON serializer to ignore self-referencing loops.
    GlobalConfiguration.Configuration.Formatters.JsonFormatter
        .SerializerSettings.ReferenceLoopHandling =
        Newtonsoft.Json.ReferenceLoopHandling.Ignore;
    ...
}
```

```
}
```

فایل RouteConfig.cs را باز کنید و قوانین مسیریابی را مانند لیست زیر تغییر دهید.

```
public static void Register(HttpConfiguration config)
{
    config.Routes.MapHttpRoute(
        name: "ActionMethodSave",
        routeTemplate: "api/{controller}/{action}/{id}",
        defaults: new { id = RouteParameter.Optional });
}
```

در آخر کنترلر TravelAgent را باز کنید و کد آن را مطابق لیست زیر بروز رسانی کنید.

```
public class TravelAgentController : ApiController
{
    // GET api/travelagent
    [HttpGet]
    public IEnumerable<TravelAgent> Retrieve()
    {
        using (var context = new Recipe3Context())
        {
            return context.TravelAgents.Include(x => x.Bookings).ToList();
        }
    }

    /// <summary>
    /// Update changes to TravelAgent, implementing Action-Based Routing in Web API
    /// </summary>
    public HttpResponseMessage Update(TravelAgent travelAgent)
    {
        using (var context = new Recipe3Context())
        {
            var newParentEntity = true;
            // adding the object graph makes the context aware of entire
            // object graph (parent and child entities) and assigns a state
            // of added to each entity.
            context.TravelAgents.Add(travelAgent);
            if (travelAgent.AgentId > 0)
            {
                // as the Id property has a value greater than 0, we assume
                // that travel agent already exists and set entity state to
                // be updated.
                context.Entry(travelAgent).State = EntityState.Modified;
                newParentEntity = false;
            }

            // iterate through child entities, assigning correct state.
            foreach (var booking in travelAgent.Bookings)
            {
                if (booking.BookingId > 0)
                {
                    // assume booking already exists if ID is greater than zero.
                    // set entity to be updated.
                    context.Entry(booking).State = EntityState.Modified;
                }
            }

            context.SaveChanges();
            HttpResponseMessage response;
            // set Http Status code based on operation type
            response = Request.CreateResponse(newParentEntity ? HttpStatusCode.Created :
            HttpStatusCode.OK, travelAgent);
            return response;
        }
    }

    [HttpDelete]
    public HttpResponseMessage Cleanup()
    {
        using (var context = new Recipe3Context())
        {
            context.Database.ExecuteSqlCommand("delete from [bookings]");
            context.Database.ExecuteSqlCommand("delete from [travelagents]");
        }
        return Request.CreateResponse(HttpStatusCode.OK);
    }
}
```

}

در قدم بعدی کلاینت پروژه را می‌سازیم که از سرویس Web API مان استفاده می‌کند.

در ویژوال استودیو پروژه جدیدی از نوع Console application بسازید و نام آن را به Recipe3.Client تغییر دهید.
فایل program.cs را باز کنید و کد آن را مطابق لیست زیر بروز رسانی کنید.

```
internal class Program
{
    private HttpClient _client;
    private TravelAgent _agent1, _agent2;
    private Booking _booking1, _booking2, _booking3;
    private HttpResponseMessage _response;

    private static void Main()
    {
        Task t = Run();
        t.Wait();
        Console.WriteLine("\nPress <enter> to continue...");
        Console.ReadLine();
    }

    private static async Task Run()
    {
        var program = new Program();
        program.ServiceSetup();
        // do not proceed until clean-up is completed
        await program.CleanupAsync();
        program.CreateFirstAgent();
        // do not proceed until agent is created
        await program.AddAgentAsync();
        program.CreateSecondAgent();
        // do not proceed until agent is created
        await program.AddSecondAgentAsync();
        program.ModifyAgent();
        // do not proceed until agent is updated
        await program.UpdateAgentAsync();
        // do not proceed until agents are fetched
        await program.FetchAgentsAsync();
    }

    private void ServiceSetup()
    {
        // set up infrastructure for Web API call
        _client = new HttpClient {BaseAddress = new Uri("http://localhost:6687/")};
        // add Accept Header to request Web API content negotiation to return resource in JSON format
        _client.DefaultRequestHeaders.Accept.Add(new
        MediaTypeWithQualityHeaderValue("application/json"));
    }

    private async Task CleanupAsync()
    {
        // call cleanup method in service
        _response = await _client.DeleteAsync("api/travelagent/cleanup/");
    }

    private void CreateFirstAgent()
    {
        // create new Travel Agent and booking
        _agent1 = new TravelAgent {Name = "John Tate"};
        _booking1 = new Booking
        {
            Customer = "Karen Stevens",
            Paid = false,
            BookingDate = DateTime.Parse("2/2/2010")
        };

        _booking2 = new Booking
        {
            Customer = "Dolly Parton",
            Paid = true,
            BookingDate = DateTime.Parse("3/10/2010")
        };

        _agent1.Bookings.Add(_booking1);
        _agent1.Bookings.Add(_booking2);
    }
}
```

```

}

private async Task AddAgentAsync()
{
    // call generic update method in Web API service to add agent and bookings
    _response = await _client.PostAsync("api/travelagent/update/",
        _agent1, new JsonMediaTypeFormatter());

    if (_response.IsSuccessStatusCode)
    {
        // capture newly created travel agent from service, which will include
        // database-generated Ids for each entity
        _agent1 = await _response.Content.ReadAsAsync<TravelAgent>();
        _booking1 = _agent1.Bookings.FirstOrDefault(x => x.Customer == "Karen Stevens");
        _booking2 = _agent1.Bookings.FirstOrDefault(x => x.Customer == "Dolly Parton");

        Console.WriteLine("Successfully created Travel Agent {0} and {1} Booking(s)",
            _agent1.Name, _agent1.Bookings.Count);
    }
    else
        Console.WriteLine("{0} ({1})", (int) _response.StatusCode, _response.ReasonPhrase);
}

private void CreateSecondAgent()
{
    // add new agent and booking
    _agent2 = new TravelAgent {Name = "Perry Como"};
    _booking3 = new Booking {
        Customer = "Loretta Lynn",
        Paid = true,
        BookingDate = DateTime.Parse("3/15/2010")};
    _agent2.Bookings.Add(_booking3);
}

private async Task AddSecondAgentAsync()
{
    // call generic update method in Web API service to add agent and booking
    _response = await _client.PostAsync("api/travelagent/update/", _agent2, new
JsonMediaTypeFormatter());

    if (_response.IsSuccessStatusCode)
    {
        // capture newly created travel agent from service
        _agent2 = await _response.Content.ReadAsAsync<TravelAgent>();
        _booking3 = _agent2.Bookings.FirstOrDefault(x => x.Customer == "Loretta Lynn");
        Console.WriteLine("Successfully created Travel Agent {0} and {1} Booking(s)",
            _agent2.Name, _agent2.Bookings.Count);
    }
    else
        Console.WriteLine("{0} ({1})", (int) _response.StatusCode, _response.ReasonPhrase);
}

private void ModifyAgent()
{
    // modify agent 2 by changing agent name and assigning booking 1 to him from agent 1
    _agent2.Name = "Perry Como, Jr.";
    _agent2.Bookings.Add(_booking1);
}

private async Task UpdateAgentAsync()
{
    // call generic update method in Web API service to update agent 2
    _response = await _client.PostAsync("api/travelagent/update/", _agent2, new
JsonMediaTypeFormatter());
    if (_response.IsSuccessStatusCode)
    {
        // capture newly created travel agent from service, which will include Ids
        _agent1 = _response.Content.ReadAsAsync<TravelAgent>().Result;
        Console.WriteLine("Successfully updated Travel Agent {0} and {1} Booking(s)", _agent1.Name,
            _agent1.Bookings.Count);
    }
    else
        Console.WriteLine("{0} ({1})", (int) _response.StatusCode, _response.ReasonPhrase);
}

private async Task FetchAgentsAsync()
{
    // call Get method on service to fetch all Travel Agents and Bookings
    _response = _client.GetAsync("api/travelagent/retrieve").Result;
    if (_response.IsSuccessStatusCode)
    {

```

```
// capture newly created travel agent from service, which will include Ids
var agents = await _response.Content.ReadAsAsync<IEnumerable<TravelAgent>>();

foreach (var agent in agents)
{
    Console.WriteLine("Travel Agent {0} has {1} Booking(s)", agent.Name,
agent.Bookings.Count());
}
}
else
    Console.WriteLine("{0} ({1})", (int) _response.StatusCode, _response.ReasonPhrase);
}
}
```

در آخر کلاس های TravelAgent و Booking را به پروژه کلاینت اضافه کنید. اینگونه کدها بهتر است در لایه مجزایی قرار گیرند و بین پروژه ها به اشتراک گذاشته شوند.

اگر اپلیکیشن کنسول (کلاینت) را اجرا کنید با خروجی زیر مواجه خواهید شد.

```
Successfully created Travel Agent John Tate and 2 Booking(s)
Successfully created Travel Agent Perry Como and 1 Booking(s)
Successfully updated Travel Agent Perry Como, Jr. and 2 Booking(s)
Travel Agent John Tate has 1 Booking(s)
Travel Agent Perry Como, Jr. has 2 Booking(s)
```

شرح مثال جاری

با اجرای اپلیکیشن Web API شروع کنید. این اپلیکیشن یک کنترلر MVC Web Controller دارد که پس از اجرا شما را به صفحه خانه هدایت می کند. در این مرحله سایت در حال اجرا است و سرویس ها قابل دسترسی هستند.

سپس اپلیکیشن کنسول را باز کنید، روی خط اول کد فایل program.cs یک breakpoint قرار دهید و آن را اجرا کنید. ابتدا آدرس سرویس Web API را نگاشت می کنیم و با تنظیم مقدار خاصیت Accept Header از سرویس درخواست می کنیم که اطلاعات را با فرمت JSON بازگرداند.

بعد از آن با استفاده از آبجکت HttpClient متد DeleteAsync را فراخوانی می کنیم که روی کنترلر TravelAgent تعریف شده است. این متد تمام داده های پیشین را حذف میکند.

در قدم بعدی سه آبجکت جدید می سازیم: یک آژانس مسافرتی و دو رزرواسیون. سپس این آبجکت ها را با فراخوانی متد PostAsync روی آبجکت HttpClient به سرویس ارسال می کنیم. اگر به متد Update در کنترلر TravelAgent یک breakpoint اضافه کنید، خواهید دید که این متد آبجکت آژانس مسافرتی را بعنوان یک پارامتر دریافت می کند و آن را به موجودیت TravelAgents در Context جاری اضافه می نماید. این کار آبجکت آژانس مسافرتی و تمام آبجکت های فرزند آن را در حالت Added اضافه می کند و باعث می شود که context جاری شروع به ردیابی (tracking) آنها کند.

نکته: قابل ذکر است که اگر موجودیت های متعددی با مقداری یکسان در خاصیت کلید اصلی (Primary-key value) دارید باید مجموعه آبجکت های خود را Add کنید و نه Attach. در مثال جاری چند آبجکت Booking داریم که مقدار کلید اصلی آنها صفر است (Bookings with Id = 0). اگر از Attach استفاده کنید EF پیغام خطایی صادر می کند چرا که چند موجودیت با مقادیر کلید اصلی یکسان به context جاری اضافه کرده اید.

بعد از آن بر اساس مقدار خاصیت Id مشخص می کنیم که موجودیت ها باید بروز رسانی شوند یا خیر. اگر مقدار این فیلد بزرگتر از صفر باشد، فرض بر این است که این موجودیت در دیتابیس وجود دارد بنابراین خاصیت EntityState را به Modified تغییر می دهیم. علاوه بر این فیلدی هم با نام newParentEntity تعریف کرده ایم که توسط آن بتوانیم کد وضعیت مناسبی به کلاینت بازگردانیم. در صورتی که مقدار فیلد Id در موجودیت TravelAgent برابر با یک باشد، مقدار خاصیت EntityState را به همان

Added رها می کنیم.

سپس تمام آبجکت های فرزند آژانس مسافرتی (رزرواسیون ها) را بررسی میکنیم و همین منطق را روی آنها اعمال می کنیم. یعنی در صورتی که مقدار فیلد Id آنها بزرگتر از 0 باشد وضعیت EntityState را به Modified تغییر می دهیم. در نهایت متد SaveChanges را فراخوانی می کنیم. در این مرحله برای موجودیت های جدید اسکریپت های Insert و برای موجودیت های تغییر کرده اسکریپت های Update تولید می شود. سپس کد وضعیت مناسب را به کلاینت بر می گردانیم. برای موجودیت های اضافه شده کد وضعیت 201 (Created) و برای موجودیت های بروز رسانی شده کد وضعیت 200 (OK) باز می گردد. کد 201 به کلاینت اطلاع می دهد که رکورد جدید با موفقیت ثبت شده است، و کد 200 از بروز رسانی موفقیت آمیز خبر می دهد. هنگام تولید سرویس های REST-based بهتر است همیشه کد وضعیت مناسبی تولید کنید.

پس از این مراحل، آژانس مسافرتی و رزرواسیون جدیدی می سازیم و آنها را به سرویس ارسال می کنیم. سپس نام آژانس مسافرتی دوم را تغییر می دهیم، و یکی از رزرواسیون ها را از آژانس اولی به آژانس دومی منتقل می کنیم. اینبار هنگام فراخوانی متد Update تمام موجودیت ها شناسه ای بزرگتر از 1 دارند، بنابراین وضعیت EntityState آنها را به Modified تغییر می دهیم تا هنگام ثبت تغییرات دستورات بروز رسانی مناسب تولید و اجرا شوند.

در آخر کلاینت ما متد Retrieve را روی سرویس فراخوانی می کند. این فراخوانی با کمک متد GetAsync انجام می شود که روی آبجکت HttpClient تعریف شده است. فراخوانی این متد تمام آژانس های مسافرتی به همراه رزرواسیون های متناظرشان را دریافت می کند. در اینجا با استفاده از متد Include تمام رکوردهای فرزند را به همراه تمام خاصیت هایشان (properties) بارگذاری می کنیم.

دقت کنید که مرتب کننده JSON تمام خواص عمومی (public properties) را باز می گرداند، حتی اگر در کد خود تعداد مشخصی از آنها را انتخاب کرده باشید.

نکته دیگر آنکه در مثال جاری از قراردادهای توکار Web API برای نگاشت درخواست های HTTP به اکشن متدها استفاده نکرده ایم. مثلاً بصورت پیش فرض درخواست های POST به متدهایی نگاشت می شوند که نام آنها با "Post" شروع می شود. در مثال جاری قواعد مسیریابی را تغییر داده ایم و رویکرد مسیریابی RPC-based را در پیش گرفته ایم. در اپلیکیشن های واقعی بهتر است از قواعد پیش فرض استفاده کنید چرا که هدف Web API ارائه سرویس های REST-based است. بنابراین بعنوان یک قاعده کلی بهتر است متدهای سرویس شما به درخواست های متناظر HTTP نگاشت شوند. و در آخر آنکه بهتر است لایه مجزایی برای میزبانی کدهای دسترسی داده ایجاد کنید و آنها را از سرویس Web API تفکیک نمایید.

نظرات خوانندگان

نویسنده: وحید

تاریخ: ۱۱:۶ ۱۳۹۲/۱۱/۱۱

با سلام شما فرمودید: " و در آخر آنکه بهتر است لایه مجزایی برای میزبانی کدهای دسترسی داده ایجاد کنید و آنها را از سرویس Web API تفکیک نمایید. " برای برقراری امنیت در این سرویس چه باید کرد؟ اگر شخصی آدرس سرویس ما رو داشت و در خواست های را به آن ارسال کرد چگونه آن را نسبت به بقیه کاربران تمیز کند؟ چون در حقیقت webapi را در پروژه جدیدی در solution قرار دادیم و جدا هاست می شود. ممنون

نویسنده: محسن خان

تاریخ: ۱۱:۴۲ ۱۳۹۲/۱۱/۱۱

برای برقراری امنیت، تعیین هویت و اعتبارسنجی در وب API عموماً یا از [Forms authentication](#) استفاده می شود و یا از [ASP.NET Identity](#) . زیر ساخت آن یکی است و مشترک.

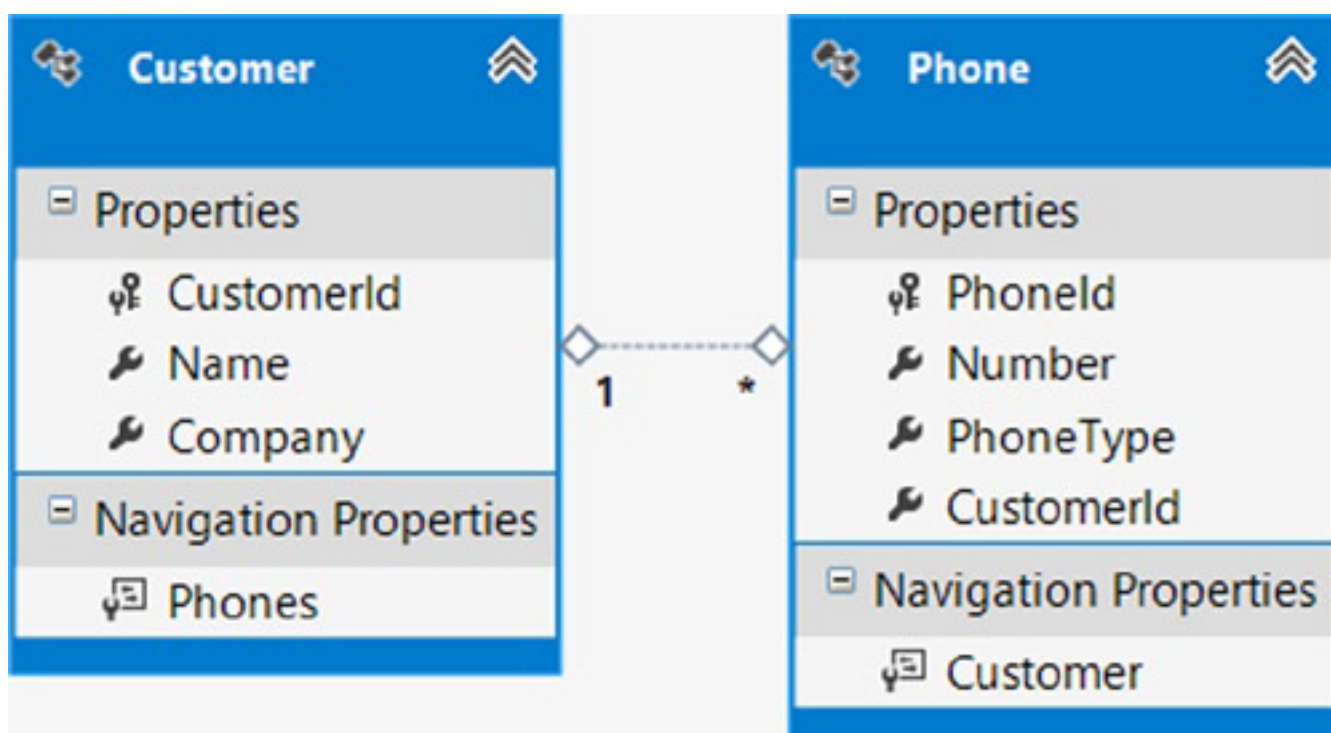
در [قسمت قبل](#) تشخیص تغییرات توسط Web API را بررسی کردیم. در این قسمت نگاهی به پیاده سازی Change-tracking در سمت کلاینت خواهیم داشت.

ردیابی تغییرات در سمت کلاینت توسط Web API

فرض کنید می‌خواهیم از سرویس‌های REST-based برای انجام عملیات CRUD روی یک Object graph استفاده کنیم. همچنین می‌خواهیم رویکردی در سمت کلاینت برای بروز رسانی کلاس موجودیت‌ها پیاده سازی کنیم که قابل استفاده مجدد (reusable) باشد. علاوه بر این دسترسی داده‌ها توسط مدل Code-First انجام می‌شود.

در مثال جاری یک اپلیکیشن کلاینت (برنامه کنسول) خواهیم داشت که سرویس‌های ارائه شده توسط پروژه Web API را فراخوانی می‌کند. هر پروژه در یک Solution مجزا قرار دارد، با این کار یک محیط n-Tier را شبیه سازی می‌کنیم.

مدل زیر را در نظر بگیرید.



همانطور که می‌بینید مدل مثال جاری مشتریان و شماره تماس آنها را ارائه می‌کند. می‌خواهیم مدل‌ها و کد دسترسی به داده‌ها را در یک سرویس Web API پیاده سازی کنیم تا هر کلاینتی که به HTTP دسترسی دارد بتواند از آن استفاده کند. برای ساخت سرویس مذکور مراحل زیر را دنبال کنید.

در ویتوال استودیو پروژه جدیدی از نوع ASP.NET Web Application بسازید و قالب پروژه را Web API انتخاب کنید. نام پروژه را به Recipe4.Service تغییر دهید.

کنترلر جدیدی با نام CustomerController به پروژه اضافه کنید.

کلاسی با نام BaseEntity ایجاد کنید و کد آن را مطابق لیست زیر تغییر دهید. تمام موجودیت‌ها از این کلاس پایه مشتق خواهند

شد که خاصیتی بنام TrackingState را به آنها اضافه می‌کند. کلاینت‌ها هنگام ویرایش آبجکت موجودیت‌ها باید این فیلد را مقدار دهی کنند. همانطور که می‌بینید این خاصیت از نوع TrackingState enum مشتق می‌شود. توجه داشته باشید که این خاصیت در دیتابیس ذخیره نخواهد شد. با پیاده سازی enum وضعیت ردیابی موجودیت‌ها بدین روش، وابستگی‌های EF را برای کلاینت از بین می‌بریم. اگر قرار بود وضعیت ردیابی را مستقیماً از EF به کلاینت پاس دهیم وابستگی‌های بخصوصی معرفی می‌شدند. کلاس DbContext اپلیکیشن در متد OnModelCreating به EF دستور می‌دهد که خاصیت TrackingState را به جدول موجودیت نگاشت نکند.

```
public abstract class BaseEntity
{
    protected BaseEntity()
    {
        TrackingState = TrackingState.Nochange;
    }

    public TrackingState TrackingState { get; set; }
}

public enum TrackingState
{
    Nochange,
    Add,
    Update,
    Remove,
}
```

کلاس‌های موجودیت Customer و PhoneNumber را ایجاد کنید و کد آنها را مطابق لیست زیر تغییر دهید.

```
public class Customer : BaseEntity
{
    public int CustomerId { get; set; }
    public string Name { get; set; }
    public string Company { get; set; }
    public virtual ICollection<Phone> Phones { get; set; }
}

public class Phone : BaseEntity
{
    public int PhoneId { get; set; }
    public string Number { get; set; }
    public string PhoneType { get; set; }
    public int CustomerId { get; set; }
    public virtual Customer Customer { get; set; }
}
```

با استفاده از NuGet Package Manager کتابخانه Entity Framework 6 را به پروژه اضافه کنید. کلاسی با نام Recipe4Context ایجاد کنید و کد آن را مطابق لیست زیر تغییر دهید. در این کلاس از یکی از قابلیت‌های جدید EF 6 بنام "Configuring Unmapped Base Types" استفاده کرده ایم. با استفاده از این قابلیت جدید هر موجودیت را طوری پیکربندی می‌کنیم که خاصیت TrackingState را نادیده بگیرند. برای اطلاعات بیشتر درباره این قابلیت EF 6 به [این لینک](#) مراجعه کنید.

```
public class Recipe4Context : DbContext
{
    public Recipe4Context() : base("Recipe4ConnectionString") { }
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Phone> Phones { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // Do not persist TrackingState property to data store
        // This property is used internally to track state of
        // disconnected entities across service boundaries.
        // Leverage the Custom Code First Conventions features from Entity Framework 6.
        // Define a convention that performs a configuration for every entity
        // that derives from a base entity class.
        modelBuilder.Types<BaseEntity>().Configure(x => x.Ignore(y => y.TrackingState));
        modelBuilder.Entity<Customer>().ToTable("Customers");
        modelBuilder.Entity<Phone>().ToTable("Phones");
    }
}
```

فایل Web.config پروژه را باز کنید و رشته اتصال زیر را به قسمت ConnectionStrings اضافه نمایید.

```
<connectionStrings>
  <add name="Recipe4ConnectionString"
    connectionString="Data Source=.;
    Initial Catalog=EFRecipes;
    Integrated Security=True;
    MultipleActiveResultSets=True"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

فایل Global.asax را باز کنید و کد زیر را به متد Application_Start اضافه نمایید. این کد بررسی Entity Framework Model Compatibility را غیرفعال می‌کند و به JSON serializer دستور می‌دهد که self-referencing loop خواص پیمایشی را نادیده بگیرد. این حلقه بدلیل رابطه bidirectional بین موجودیت‌های Customer و PhoneNumber بوجود می‌آید.

```
protected void Application_Start()
{
    // Disable Entity Framework Model Compatibility
    Database.SetInitializer<Recipe1Context>(null);
    // The bidirectional navigation properties between related entities
    // create a self-referencing loop that breaks Web API's effort to
    // serialize the objects as JSON. By default, Json.NET is configured
    // to error when a reference loop is detected. To resolve problem,
    // simply configure JSON serializer to ignore self-referencing loops.
    GlobalConfiguration.Configuration.Formatters.JsonFormatter
        .SerializerSettings.ReferenceLoopHandling =
            Newtonsoft.Json.ReferenceLoopHandling.Ignore;
    ...
}
```

کلاسی با نام EntityStateFactory بسازید و کد آن را مطابق لیست زیر تغییر دهید. این کلاس مقدار خاصیت TrackingState که به کلاینت‌ها ارائه می‌شود را به مقادیر متناظر کامپوننت‌های ردیابی EF تبدیل می‌کند.

```
public static EntityState Set(TrackingState trackingState)
{
    switch (trackingState)
    {
        case TrackingState.Add:
            return EntityState.Added;
        case TrackingState.Update:
            return EntityState.Modified;
        case TrackingState.Remove:
            return EntityState.Deleted;
        default:
            return EntityState.Unchanged;
    }
}
```

در آخر کد کنترلر CustomerController را مطابق لیست زیر بروز رسانی کنید.

```
public class CustomerController : ApiController
{
    // GET api/customer
    public IEnumerable<Customer> Get()
    {
        using (var context = new Recipe4Context())
        {
            return context.Customers.Include(x => x.Phones).ToList();
        }
    }

    // GET api/customer/5
    public Customer Get(int id)
    {
        using (var context = new Recipe4Context())
        {
            return context.Customers.Include(x => x.Phones).FirstOrDefault(x => x.CustomerId == id);
        }
    }
}
```

```

}

[ActionName("Update")]
public HttpResponseMessage UpdateCustomer(Customer customer)
{
    using (var context = new Recipe4Context())
    {
        // Add object graph to context setting default state of 'Added'.
        // Adding parent to context automatically attaches entire graph
        // (parent and child entities) to context and sets state to 'Added'
        // for all entities.
        context.Customers.Add(customer);
        foreach (var entry in context.ChangeTracker.Entries<BaseEntity>())
        {
            entry.State = EntityStateFactory.Set(entry.Entity.TrackingState);
            if (entry.State == EntityState.Modified)
            {
                // For entity updates, we fetch a current copy of the entity
                // from the database and assign the values to the original values
                // property from the Entry object. OriginalValues wrap a dictionary
                // that represents the values of the entity before applying changes.
                // The Entity Framework change tracker will detect
                // differences between the current and original values and mark
                // each property and the entity as modified. Start by setting
                // the state for the entity as 'Unchanged'.
                entry.State = EntityState.Unchanged;
                var databaseValues = entry.GetDatabaseValues();
                entry.OriginalValues.SetValues(databaseValues);
            }
        }

        context.SaveChanges();
    }

    return Request.CreateResponse(HttpStatusCode.OK, customer);
}

[HttpDelete]
[ActionName("Cleanup")]
public HttpResponseMessage Cleanup()
{
    using (var context = new Recipe4Context())
    {
        context.Database.ExecuteSqlCommand("delete from phones");
        context.Database.ExecuteSqlCommand("delete from customers");
        return Request.CreateResponse(HttpStatusCode.OK);
    }
}
}

```

حال اپلیکیشن کلاینت (برنامه کنسول) را می‌سازیم که از این سرویس استفاده می‌کند.

در ویژوال استودیو پروژه جدیدی از نوع Console Application بسازید و نام آن را به Recipe4.Client تغییر دهید. فایل program.cs را باز کنید و کد آن را مطابق لیست زیر تغییر دهید.

```

internal class Program
{
    private HttpClient _client;
    private Customer _bush, _obama;
    private Phone _whiteHousePhone, _bushMobilePhone, _obamaMobilePhone;
    private HttpResponseMessage _response;

    private static void Main()
    {
        Task t = Run();
        t.Wait();
        Console.WriteLine("\nPress <enter> to continue...");
        Console.ReadLine();
    }

    private static async Task Run()
    {
        var program = new Program();
        program.ServiceSetup();
        // do not proceed until clean-up completes
        await program.CleanupAsync();
    }
}

```

```

        program.CreateFirstCustomer();
        // do not proceed until customer is added
        await program.AddCustomerAsync();
        program.CreateSecondCustomer();
        // do not proceed until customer is added
        await program.AddSecondCustomerAsync();
        // do not proceed until customer is removed
        await program.RemoveFirstCustomerAsync();
        // do not proceed until customers are fetched
        await program.FetchCustomersAsync();
    }

    private void ServiceSetup()
    {
        // set up infrastructure for Web API call
        _client = new HttpClient { BaseAddress = new Uri("http://localhost:62799/") };
        // add Accept Header to request Web API content negotiation to return resource in JSON format
        _client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue
            ("application/json"));
    }

    private async Task CleanupAsync()
    {
        // call the cleanup method from the service
        _response = await _client.DeleteAsync("api/customer/cleanup/");
    }

    private void CreateFirstCustomer()
    {
        // create customer #1 and two phone numbers
        _bush = new Customer
        {
            Name = "George Bush",
            Company = "Ex President",
            // set tracking state to 'Add' to generate a SQL Insert statement
            TrackingState = TrackingState.Add,
        };
        _whiteHousePhone = new Phone
        {
            Number = "212 222-2222",
            PhoneType = "White House Red Phone",
            // set tracking state to 'Add' to generate a SQL Insert statement
            TrackingState = TrackingState.Add,
        };
        _bushMobilePhone = new Phone
        {
            Number = "212 333-3333",
            PhoneType = "Bush Mobile Phone",
            // set tracking state to 'Add' to generate a SQL Insert statement
            TrackingState = TrackingState.Add,
        };
        _bush.Phones.Add(_whiteHousePhone);
        _bush.Phones.Add(_bushMobilePhone);
    }

    private async Task AddCustomerAsync()
    {
        // construct call to invoke UpdateCustomer action method in Web API service
        _response = await _client.PostAsync("api/customer/updatecustomer/", _bush, new
        JsonMediaTypeFormatter());
        if (_response.IsSuccessStatusCode)
        {
            // capture newly created customer entity from service, which will include
            // database-generated Ids for all entities
            _bush = await _response.Content.ReadAsAsync<Customer>();
            _whiteHousePhone = _bush.Phones.FirstOrDefault(x => x.CustomerId == _bush.CustomerId);
            _bushMobilePhone = _bush.Phones.FirstOrDefault(x => x.CustomerId == _bush.CustomerId);
            Console.WriteLine("Successfully created Customer {0} and {1} Phone Numbers(s)",
                _bush.Name, _bush.Phones.Count);
            foreach (var phoneType in _bush.Phones)
            {
                Console.WriteLine("Added Phone Type: {0}", phoneType.PhoneType);
            }
        }
        else
            Console.WriteLine("{0} ({1})", (int)_response.StatusCode, _response.ReasonPhrase);
    }

    private void CreateSecondCustomer()
    {
        // create customer #2 and phone numbers
        _obama = new Customer
    
```

```

    {
        Name = "Barack Obama",
        Company = "President",
        // set tracking state to 'Add' to generate a SQL Insert statement
        TrackingState = TrackingState.Add,
    };
    _obamaMobilePhone = new Phone
    {
        Number = "212 444-4444",
        PhoneType = "Obama Mobile Phone",
        // set tracking state to 'Add' to generate a SQL Insert statement
        TrackingState = TrackingState.Add,
    };
    // set tracking state to 'Modified' to generate a SQL Update statement
    _whiteHousePhone.TrackingState = TrackingState.Update;
    _obama.Phones.Add(_obamaMobilePhone);
    _obama.Phones.Add(_whiteHousePhone);
}

private async Task AddSecondCustomerAsync()
{
    // construct call to invoke UpdateCustomer action method in Web API service
    _response = await _client.PostAsync("api/customer/updatecustomer/", _obama, new
JsonMediaTypeFormatter());
    if (_response.IsSuccessStatusCode)
    {
        // capture newly created customer entity from service, which will include
        // database-generated Ids for all entities
        _obama = await _response.Content.ReadAsAsync<Customer>();
        _whiteHousePhone = _bush.Phones.FirstOrDefault(x => x.CustomerId == _obama.CustomerId);
        _bushMobilePhone = _bush.Phones.FirstOrDefault(x => x.CustomerId == _obama.CustomerId);
        Console.WriteLine("Successfully created Customer {0} and {1} Phone Numbers(s)",
            _obama.Name, _obama.Phones.Count);
        foreach (var phoneType in _obama.Phones)
        {
            Console.WriteLine("Added Phone Type: {0}", phoneType.PhoneType);
        }
    }
    else
        Console.WriteLine("{0} ({1})", (int)_response.StatusCode, _response.ReasonPhrase);
}

private async Task RemoveFirstCustomerAsync()
{
    // remove George Bush from underlying data store.
    // first, fetch George Bush entity, demonstrating a call to the
    // get action method on the service while passing a parameter
    var query = "api/customer/" + _bush.CustomerId;
    _response = _client.GetAsync(query).Result;

    if (_response.IsSuccessStatusCode)
    {
        _bush = await _response.Content.ReadAsAsync<Customer>();
        // set tracking state to 'Remove' to generate a SQL Delete statement
        _bush.TrackingState = TrackingState.Remove;
        // must also remove bush's mobile number -- must delete child before removing parent
        foreach (var phoneType in _bush.Phones)
        {
            // set tracking state to 'Remove' to generate a SQL Delete statement
            phoneType.TrackingState = TrackingState.Remove;
        }
        // construct call to remove Bush from underlying database table
        _response = await _client.PostAsync("api/customer/updatecustomer/", _bush, new
JsonMediaTypeFormatter());
        if (_response.IsSuccessStatusCode)
        {
            Console.WriteLine("Removed {0} from database", _bush.Name);
            foreach (var phoneType in _bush.Phones)
            {
                Console.WriteLine("Remove {0} from data store", phoneType.PhoneType);
            }
        }
        else
            Console.WriteLine("{0} ({1})", (int)_response.StatusCode, _response.ReasonPhrase);
    }
    else
    {
        Console.WriteLine("{0} ({1})", (int)_response.StatusCode, _response.ReasonPhrase);
    }
}

```

```
private async Task FetchCustomersAsync()
{
    // finally, return remaining customers from underlying data store
    _response = await _client.GetAsync("api/customer/");
    if (_response.IsSuccessStatusCode)
    {
        var customers = await _response.Content.ReadAsAsync<IEnumerable<Customer>>();
        foreach (var customer in customers)
        {
            Console.WriteLine("Customer {0} has {1} Phone Numbers(s)",
                customer.Name, customer.Phones.Count());
            foreach (var phoneType in customer.Phones)
            {
                Console.WriteLine("Phone Type: {0}", phoneType.PhoneType);
            }
        }
    }
    else
    {
        Console.WriteLine("{0} ({1})", (int)_response.StatusCode, _response.ReasonPhrase);
    }
}
}
```

در آخر کلاس های Customer, Phone و BaseEntity را به پروژه کلاینت اضافه کنید. چنین کدهایی بهتر است در لایه مجزایی قرار گیرند و بین لایه های مختلف اپلیکیشن به اشتراک گذاشته شوند.

اگر اپلیکیشن کلاینت را اجرا کنید با خروجی زیر مواجه خواهید شد.

```
Successfully created Customer Geroge Bush and 2 Phone Numbers(s)
Added Phone Type: White House Red Phone
Added Phone Type: Bush Mobile Phone
Successfully created Customer Barrack Obama and 2 Phone Numbers(s)
Added Phone Type: Obama Mobile Phone
Added Phone Type: White House Red Phone
Removed Geroge Bush from database
Remove Bush Mobile Phone from data store
Customer Barrack Obama has 2 Phone Numbers(s)
Phone Type: White House Red Phone
Phone Type: Obama Mobile Phone
```

شرح مثال جاری

با اجرای اپلیکیشن Web API شروع کنید. این اپلیکیشن یک MVC Web Controller دارد که پس از اجرا شما را به صفحه خانه هدایت می کند. در این مرحله سایت در حال اجرا است و سرویس ها قابل دسترسی هستند.

سپس اپلیکیشن کنسول را باز کنید و روی خط اول کد فایل program.cs یک breakpoint قرار داده و آن را اجرا کنید. ابتدا آدرس سرویس را نگاشت می کنیم و از سرویس درخواست می کنیم که اطلاعات را با فرمت JSON بازگرداند.

سپس توسط متد DeleteAsync که روی آبجکت HttpClient تعریف شده است اکشن متد Cleanup را روی سرویس فراخوانی می کنیم. این فراخوانی تمام داده های پیشین را حذف می کند.

در قدم بعدی یک مشتری به همراه دو شماره تماس می سازیم. توجه کنید که برای هر موجودیت مشخصا خاصیت TrackingState

را مقدار دهی می‌کنیم تا کامپوننت‌های Change-tracking در EF عملیات لازم SQL برای هر موجودیت را تولید کنند.

سپس توسط متد PostAsync که روی آبجکت HttpClient تعریف شده اکشن متد UpdateCustomer را روی سرویس فراخوانی می‌کنیم. اگر به این اکشن متد یک breakpoint اضافه کنید خواهید دید که موجودیت مشتری را بعنوان یک پارامتر دریافت می‌کند و آن را به context جاری اضافه می‌نماید. با اضافه کردن موجودیت به کانتکست جاری کل object graph اضافه می‌شود و EF شروع به ردیابی تغییرات آن می‌کند. دقت کنید که آبجکت موجودیت باید Add شود و نه Attach.

قدم بعدی جالب است، هنگامی که از خاصیت DbChangeTracker استفاده می‌کنیم. این خاصیت روی آبجکت context تعریف شده و یک `IEnumerable<DbEntityEntry>` را با نام Entries ارائه می‌کند. در اینجا بسادگی نوع پایه `EntityType` را تنظیم می‌کنیم. این کار به ما اجازه می‌دهد که در تمام موجودیت‌هایی که از نوع `BaseEntity` هستند پیمایش کنیم. اگر بیاد داشته باشید این کلاس، کلاس پایه تمام موجودیت‌ها است. در هر مرحله از پیمایش (iteration) با استفاده از کلاس `EntityStateFactory` مقدار خاصیت `TrackingState` را به مقدار متناظر در سیستم ردیابی EF تبدیل می‌کنیم. اگر کلاینت مقدار این فیلد را به `Modified` تنظیم کرده باشد پردازش بیشتری انجام می‌شود. ابتدا وضعیت موجودیت را از `Modified` به `Unchanged` تغییر می‌دهیم. سپس مقادیر اصلی را با فراخوانی متد `GetDatabaseValues` روی آبجکت `Entry` از دیتابیس دریافت می‌کنیم. فراخوانی این متد مقادیر موجود در دیتابیس را برای موجودیت جاری دریافت می‌کند. سپس مقادیر بدست آمده را به کلکسیون `OriginalValues` اختصاص می‌دهیم. پشت پرده، کامپوننت‌های EF Change-tracking بصورت خودکار تفاوت‌های مقادیر اصلی و مقادیر ارسالی را تشخیص می‌دهند و فیلدهای مربوطه را با وضعیت `Modified` علامت گذاری می‌کنند. فراخوانی‌های بعدی متد `SaveChanges` تنها فیلدهایی که در سمت کلاینت تغییر کرده اند را بروز رسانی خواهد کرد و نه تمام خواص موجودیت را.

در اپلیکیشن کلاینت عملیات افزودن، بروز رسانی و حذف موجودیت‌ها توسط مقداردهی خاصیت `TrackingState` را نمایش داده ایم.

متد `UpdateCustomer` در سرویس ما مقادیر `TrackingState` را به مقادیر متناظر EF تبدیل می‌کند و آبجکت‌ها را به موتور change-tracking ارسال می‌کند که نهایتاً منجر به تولید دستورات لازم SQL می‌شود.

نکته: در اپلیکیشن‌های واقعی بهتر است کد دسترسی داده‌ها و مدل‌های دامنه را به لایه مجزایی منتقل کنید. همچنین پیاده سازی فعلی change-tracking در سمت کلاینت می‌تواند توسعه داده شود تا با انواع جنریک کار کند. در این صورت از نوشتن مقادیر زیادی کد تکراری جلوگیری خواهید کرد و از یک پیاده سازی می‌توانید برای تمام موجودیت‌ها استفاده کنید.

نظرات خوانندگان

نویسنده: امیرحسین

تاریخ: ۱۳۹۲/۱۱/۱۰ ۰:۴

میشه در مورد async کمی توضیح بدین که چرا و به چه دلیلی استفاده شده ؟

نویسنده: آرمین ضیاء

تاریخ: ۱۳۹۲/۱۱/۱۰ ۱:۲۵

الزامی به استفاده از قابلیت های async نیست، اما توصیه میشه در مواقعی که امکانش هست و مناسب است از این قابلیت استفاده کنید. لزوما کارایی (performance) بهتری بدست نمیاری ولی مسلما تجربه کاربری بهتری خواهید داشت. عملیاتی که بصورت async اجرا میشن ریسمان جاری (current thread) رو قفل نمی کنند، بنابراین اجرای اپلیکیشن ادامه پیدا می کنه و پاسخگویی بهتری بدست میارید. برای مطالعه بیشتر به [این لینک](#) مراجعه کنید.

مطالعه بیشتر

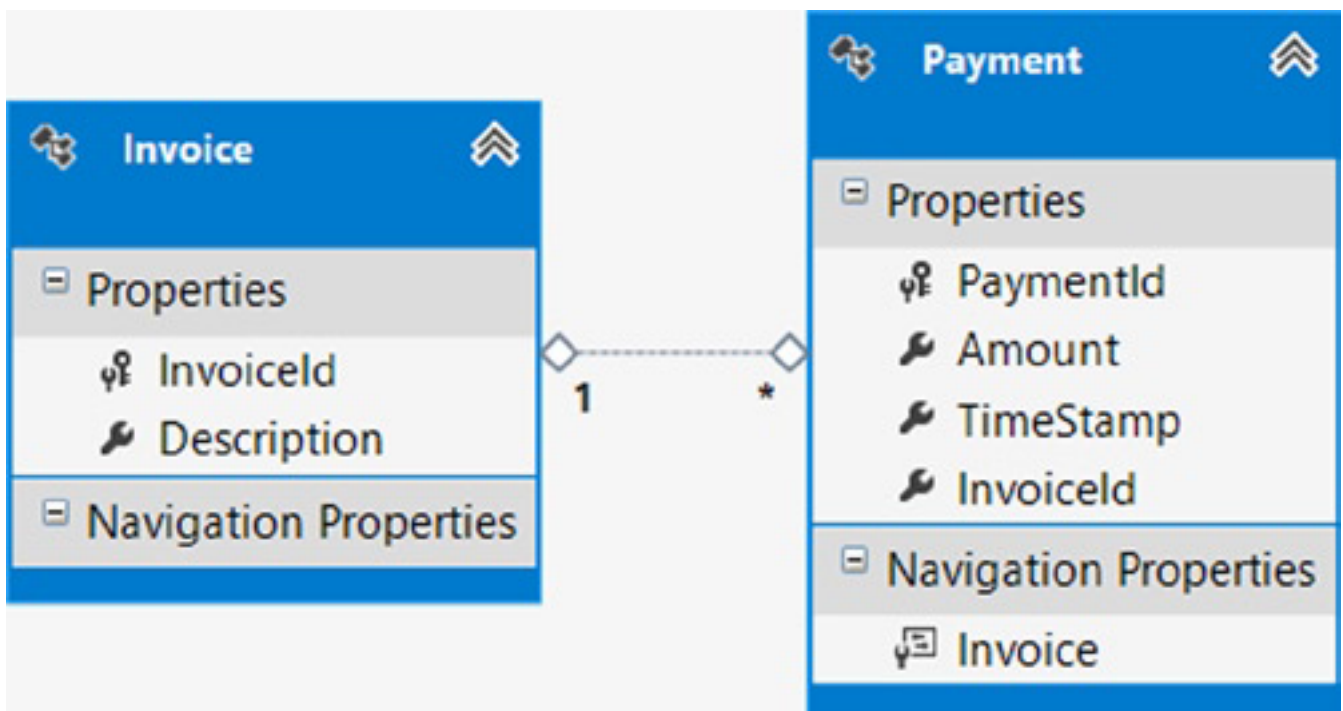
[Using Asynchronous Methods in ASP.NET 4.5](#)

[Async and Await](#)

در [قسمت قبل](#) پیاده سازی change-tracking در سمت کلاینت توسط Web API را بررسی کردیم. در این قسمت نگاهی به حذف موجودیت های منفصل یا disconnected خواهیم داشت.

حذف موجودیت های منفصل

فرض کنید موجودیتی را از یک سرویس WCF دریافت کرده اید و می خواهید آن را برای حذف علامت گذاری کنید. مدل زیر را در نظر بگیرید.



همانطور که می بینید مدل ما صورت حساب ها و پرداخت های متناظر را ارائه می کند. در اپلیکیشن جاری یک سرویس WCF پیاده سازی کرده ایم که عملیات دیتابسی کلاینت ها را مدیریت می کند. می خواهیم توسط این سرویس آبجکتی را (در اینجا یک موجودیت پرداخت) حذف کنیم. برای ساده نگاه داشتن مثال جاری، مدل ها را در خود سرویس تعریف می کنیم. برای ایجاد سرویس مذکور مراحل زیر را دنبال کنید.

در ویژوال استودیو پروژه جدیدی از نوع WCF Service Library بسازید و نام آن را به Recipe5 تغییر دهید.

روی پروژه کلیک راست کنید و گزینه Add New Item را انتخاب کنید. سپس گزینه های ADO.NET Entity Data Model -> Data را برگزینید.

از ویزارد ویژوال استودیو برای اضافه کردن یک مدل با جداول Invoice و Payment استفاده کنید. برای ساده نگه داشتن مثال جاری، فیلد پیمایشی Payments را از موجودیت Invoice حذف کرده ایم (برای این کار روی خاصیت پیمایشی Payments کلیک راست کنید و گزینه Delete From Model را انتخاب کنید). روی خاصیت TimeStamp موجودیت Payment کلیک راست کنید و گزینه Properties را انتخاب کنید. سپس مقدار Concurrency Mode آن را به Fixed تغییر دهید. این کار باعث می شود که مقدار این فیلد برای کنترل همزمانی بررسی شود. بنابراین مقدار TimeStamp در عبارت WHERE تمام دستورات بروز رسانی و حذف درج خواهد شد.

فایل IService1.cs را باز کنید و تعریف سرویس را مانند لیست زیر تغییر دهید.

```
[ServiceContract]
public interface IService1
{
    [OperationContract]
    Payment InsertPayment();
    [OperationContract]
    void DeletePayment(Payment payment);
}
```

فایل Service1.cs را باز کنید و پیاده سازی سرویس را مانند لیست زیر تغییر دهید.

```
public class Service1 : IService1
{
    public Payment InsertPayment()
    {
        using (var context = new EFRecipesEntities())
        {
            // delete the previous test data
            context.Database.ExecuteSqlCommand("delete from [payments]");
            context.Database.ExecuteSqlCommand("delete from [invoices]");
            var payment = new Payment { Amount = 99.95M, Invoice =
                new Invoice { Description = "Auto Repair" } };
            context.Payments.Add(payment);
            context.SaveChanges();
            return payment;
        }
    }

    public void DeletePayment(Payment payment)
    {
        using (var context = new EFRecipesEntities())
        {
            context.Entry(payment).State = EntityState.Deleted;
            context.SaveChanges();
        }
    }
}
```

برای تست این سرویس به یک کلاینت نیاز داریم. یک پروژه جدید از نوع Console Application به راه حل جاری اضافه کنید و کد آن را مطابق لیست زیر تغییر دهید. فراموش نکنید که ارجاعی به سرویس هم اضافه کنید. روی پروژه کلاینت کلیک راست کرده و Add Service Reference را انتخاب نمایید. ممکن است پیش از آنکه بتوانید سرویس را ارجاع کنید، نیاز باشد پروژه سرویس را ابتدا اجرا کنید (کلیک راست روی پروژه سرویس و انتخاب گزینه Debug -> Start Instance).

```
class Program
{
    static void Main()
    {
        var client = new Service1Client();
        var payment = client.InsertPayment();
        client.DeletePayment(payment);
    }
}
```

اگر روی خط اول متد Main() یک breakpoint قرار دهید می‌توانید مراحل ایجاد و حذف یک موجودیت Payment را دنبال کنید.

شرح مثال جاری

در مثال جاری برای بروز رسانی و حذف موجودیت‌های منفصل از الگویی رایج استفاده کرده ایم که در سرویس‌های WCF و Web API استفاده می‌شود.

در کلاینت با فراخوانی متد InsertPayment یک پرداخت جدید در دیتابیس ذخیره می‌کنیم. این متد، موجودیت Payment ایجاد شده را باز می‌گرداند. موجودیتی که به کلاینت باز می‌گردد از DbContext منفصل (disconnected) است، در واقع در چنین وضعیتی آبجکت context ممکن است در فضای پروسس دیگری قرار داشته باشد، یا حتی روی کامپیوتر دیگری باشد.

برای حذف موجودیت Payment از متد DeletePayment استفاده می‌کنیم. این متد به نوبه خود با فراخوانی متد Entry روی آبجکت context و پاس دادن موجودیت پرداخت بعنوان آرگومان، موجودیت را پیدا می‌کند. سپس وضعیت موجودیت را به EntityState.Deleted تغییر می‌دهیم که این کار آبجکت را برای حذف علامت گذاری می‌کند. فراخوانی‌های بعدی متد SaveChanges() موجودیت را از دیتابیس حذف خواهد کرد.

آبجکت پرداختی که برای حذف به context الحاق کرده ایم تمام خاصیت هایش مقدار دهی شده اند، درست مانند هنگامی که این موجودیت به دیتابیس اضافه شده بود. اما از آنجا که از foreign key association استفاده می‌کنیم، تنها فیلدهای کلید موجودیت، خاصیت همزمانی (concurrency) و TimeStamp برای تولید عبارت where مناسب لازم هستند که نهایتاً منجر به حذف موجودیت خواهد شد. تنها استثنا درباره این قاعده هنگامی است که موجودیت شما یک یا چند خاصیت از نوع پیچیده یا Complex Type داشته باشد. از آنجا که خاصیت‌های پیچیده، اجزای ساختاری یک موجودیت محسوب می‌شوند نمی‌توانند مقادیر null بپذیرند. یک راه حل ساده این است که هنگامی که EF مشغول ساختن عبارت SQL Delete لازم برای حذف موجودیت بر اساس کلید و خاصیت همزمانی آن است، وهله جدیدی از نوع داده پیچیده خود بسازید. اگر فیلدهای complex type را با مقادیر null رها کنید، فراخوانی متد SaveChanges() با خطا مواجه خواهد شد.

اگر از یک independent association استفاده می‌کنید که در آن کثرت (multiplicity) موجودیت مربوطه یک، یا صفر به یک است، EF انتظار دارد که کلیدهای موجودیت‌ها بدرستی مقدار دهی شوند تا بتواند عبارت where مناسب را برای دستورات بروز رسانی و حذف تولید کند. اگر در مثال جاری از یک رابطه independent association بین موجودیت‌های Invoice و Payment استفاده می‌کردیم، لازم بود تا خاصیت پیمایشی Invoice را با وهله ای از صورت حساب مقدار دهی کنیم که خاصیت InvoiceId آن نیز بدرستی مقدار دهی شده باشد. در این صورت عبارت where نهایی شامل فیلدهای InvoiceId، PaymentId، TimeStamp و InvoiceId خواهد بود.

نکته: هنگام پیاده سازی معماری‌های n-Tier با Entity Framework، استفاده از رویکرد Foreign Key Association برای موجودیت‌های مرتبط باید با ملاحظات جدی انجام شود. پیاده سازی رویکرد Independent Association مشکل است و می‌تواند کد شما را بسیار پیچیده کند. برای مطالعه بیشتر درباره این رویکردها و مزایا و معایب آنها به [این لینک](#) مراجعه کنید که توسط یکی از برنامه نویسان تیم EF نوشته شده است.

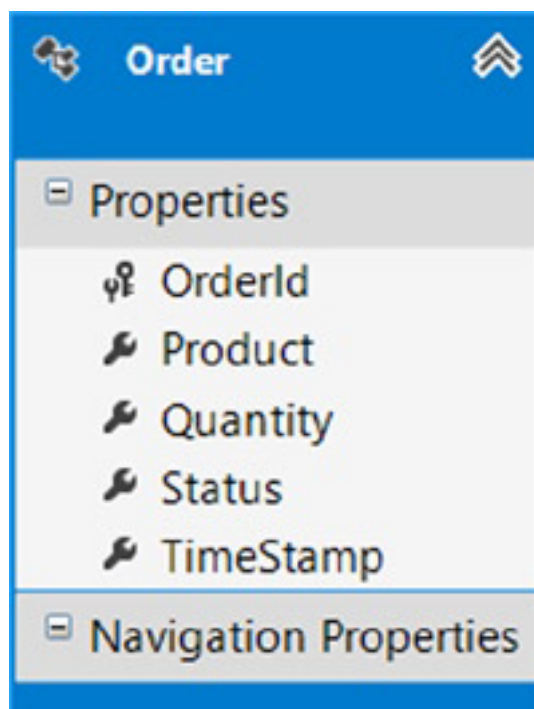
اگر موجودیت شما تعداد متعددی Independent Association دارد، مقدار دهی تمام آنها می‌تواند خسته کننده شود. رویکردی ساده‌تر این است که وهله مورد نظر را از دیتابیس دریافت کنید و آن را برای حذف علامت گذاری نمایید. این روش کد شما را ساده‌تر می‌کند، اما هنگامی که آبجکت را از دیتابیس دریافت می‌کنید EF کوئری جاری را بازنویسی می‌کند تا تمام روابط یک، یا صفر به یک بارگذاری شوند. مگر آنکه از گزینه NoTracking روی context خود استفاده کنید. اگر در مثال جاری رویکرد Independent Association را پیاده سازی کرده بودیم، هنگامی که موجودیت Payment را از دیتابیس دریافت می‌کنیم (قبل از علامت گذاری برای حذف) EF یک Object state entry برای موجودیت پرداخت و یک Relationship entry برای رابطه بین Payment و Invoice می‌ساخت. سپس وقتی که موجودیت پرداخت را برای حذف علامت گذاری می‌کنیم، EF رابطه بین پرداخت و صورت حساب را هم برای حذف علامت گذاری می‌کند. در اینجا عبارت where تولید شده مانند قبل، شامل فیلدهای PaymentId، InvoiceId و TimeStamp خواهد بود.

یک گزینه دیگر برای حذف موجودیت‌ها در Independent Associations این است که تمام موجودیت‌های مرتبط را مشخصاً بارگذاری کنیم (eager loading) و کل Object graph را برای حذف به سرویس WCF یا Web API بفرستیم. در مثال جاری می‌توانستیم موجودیت صورتحساب مرتبط با موجودیت پرداخت را مشخصاً بارگذاری کنیم. اگر می‌خواستیم موجودیت Payment را حذف کنیم، می‌توانستیم کل گراف را که شامل هر دو موجودیت می‌شود به سرویس ارسال کنیم. اما هنگام استفاده از چنین روشی باید بسیار دقت کنید، چرا که این رویکرد پهنای باند بیشتری مصرف می‌کند و زمان پردازش بیشتری هم برای مرتب سازی (serialization) صرف می‌کند. بنابراین هزینه این رویکرد نسبت به سادگی کدی که بدست می‌آید به مراتب بیشتر است.

در [قسمت قبل](#) رویکردهای مختلف برای حذف موجودیت های منفصل را بررسی کردیم. در این قسمت مدیریت همزمانی یا Concurrency را بررسی خواهیم کرد.

فرض کنید می خواهیم مطمئن شویم که موجودیتی که توسط یک کلاینت WCF تغییر کرده است، تنها در صورتی بروز رسانی شود که شناسه (token) همزمانی آن تغییر نکرده باشد. به بیان دیگر شناسه ای که هنگام دریافت موجودیت بدست می آید، هنگام بروز رسانی باید مقداری یکسان داشته باشد.

مدل زیر را در نظر بگیرید.



می خواهیم یک سفارش (order) را توسط یک سرویس WCF بروز رسانی کنیم در حالی که اطمینان حاصل می کنیم موجودیت سفارش از زمانی که دریافت شده تغییری نکرده است. برای مدیریت این وضعیت دو رویکرد تقریباً متفاوت را بررسی می کنیم. در هر دو رویکرد از یک ستون همزمانی استفاده می کنیم، در این مثال فیلد TimeStamp.

در ویژوال استودیو پروژه جدیدی از نوع WCF Service Library بسازید و نام آن را به Recipe6 تغییر دهید.

روی نام پروژه کلیک راست کنید و گزینه Add New Item را انتخاب کنید. سپس گزینه های Data -> Entity Data Model را برگزینید. از ویزارد ویژوال استودیو برای اضافه کردن مدل جاری و جدول Orders استفاده کنید. در EF Designer روی فیلد TimeStamp کلیک راست کنید و گزینه Properties را انتخاب کنید. سپس مقدار CuncurrencyMode آنرا به Fixed تغییر دهید. فایل IService1.cs را باز کنید و تعریف سرویس را مطابق لیست زیر بروز رسانی کنید.

```
[ServiceContract]
public interface IService1
{
    [OperationContract]
```

```

Order InsertOrder();
[OperationContract]
void UpdateOrderWithoutRetrieving(Order order);
[OperationContract]
void UpdateOrderByRetrieving(Order order);
}

```

فایل Service1.cs را باز کنید و پیاده سازی سرویس را مطابق لیست زیر تکمیل کنید.

```

public class Service1 : IService1
{
    public Order InsertOrder()
    {
        using (var context = new EFRecipesEntities())
        {
            // remove previous test data
            context.Database.ExecuteSqlCommand("delete from [orders]");
            var order = new Order
            {
                Product = "Camping Tent",
                Quantity = 3,
                Status = "Received"
            };
            context.Orders.Add(order);
            context.SaveChanges();
            return order;
        }
    }

    public void UpdateOrderWithoutRetrieving(Order order)
    {
        using (var context = new EFRecipesEntities())
        {
            try
            {
                context.Orders.Attach(order);
                if (order.Status == "Received")
                {
                    context.Entry(order).Property(x => x.Quantity).IsModified = true;
                    context.SaveChanges();
                }
            }
            catch (OptimisticConcurrencyException ex)
            {
                // Handle OptimisticConcurrencyException
            }
        }
    }

    public void UpdateOrderByRetrieving(Order order)
    {
        using (var context = new EFRecipesEntities())
        {
            // fetch current entity from database
            var dbOrder = context.Orders
                .Single(o => o.OrderId == order.OrderId);
            if (dbOrder != null &&
                // execute concurrency check
                StructuralComparisons.StructuralEqualityComparer.Equals(order.TimeStamp,
                dbOrder.TimeStamp))
            {
                dbOrder.Quantity = order.Quantity;
                context.SaveChanges();
            }
            else
            {
                // Add code to handle concurrency issue
            }
        }
    }
}

```

برای تست این سرویس به یک کلاینت نیاز داریم. پروژه جدیدی از نوع Console Application به راه حل جاری اضافه کنید و کد آن را مطابق لیست زیر تکمیل کنید. با کلیک راست روی نام پروژه و انتخاب گزینه Add Service Reference سرویس پروژه را هم ارجاع کنید. دقت کنید که ممکن است پیش از آنکه بتوانید سرویس را ارجاع کنید نیاز باشد روی آن کلیک راست کرده و از منوی

Debug گزینه Start Instance را انتخاب کنید تا و هله از سرویس به اجرا در بیاید.

```
class Program
{
    static void Main(string[] args)
    {
        var service = new Service1Client();
        var order = service.InsertOrder();
        order.Quantity = 5;
        service.UpdateOrderWithoutRetrieving(order);
        order = service.InsertOrder();
        order.Quantity = 3;
        service.UpdateOrderByRetrieving(order);
    }
}
```

اگر به خط اول متد Main() یک breakpoint اضافه کنید و اپلیکیشن را اجرا کنید می‌توانید افزودن و بروز رسانی یک Order با هر دو رویکرد را بررسی کنید.

شرح مثال جاری

متد InsertOrder() داده‌های پیشین را حذف می‌کند، سفارش جدیدی می‌سازد و آن را در دیتابیس ثبت می‌کند. در آخر موجودیت جدید به کلاینت باز می‌گردد. موجودیت بازگشتی هر دو مقدار OrderId و TimeStamp را دارا است که توسط دیتابیس تولید شده اند. سپس در کلاینت از دو رویکرد نسبتاً متفاوت برای بروز رسانی موجودیت استفاده می‌کنیم.

در رویکرد نخست، متد UpdateOrderWithoutRetrieving() موجودیت دریافت شده از کلاینت را Attach می‌کند و چک می‌کند که مقدار فیلد Status چیست. اگر مقدار این فیلد "Received" باشد، فیلد Quantity را با EntityState.Modified علامت گذاری می‌کنیم و متد SaveChanges() را فراخوانی می‌کنیم. EF دستورات لازم برای بروز رسانی را تولید می‌کند، که فیلد quantity را مقدار دهی کرده و یک عبارت where هم دارد که فیلدهای OrderId و TimeStamp را چک می‌کند. اگر مقدار TimeStamp توسط یک دستور بروز رسانی تغییر کرده باشد، بروز رسانی جاری با خطا مواجه خواهد شد. برای مدیریت این خطا ما بدنه کد را در یک بلاک try/catch قرار می‌دهیم، و استثنای OptimisticConcurrencyException را مهار می‌کنیم. این کار باعث می‌شود اطمینان داشته باشیم که موجودیت Order دریافت شده از متد InsertOrder() تاکنون تغییری نکرده است. دقت کنید که در مثال جاری تمام خواص موجودیت بروز رسانی می‌شوند، صرفنظر از اینکه تغییر کرده باشند یا خیر.

رویکرد دوم نشان می‌دهد که چگونه می‌توان وضعیت همزمانی موجودیت را پیش از بروز رسانی مشخصا دریافت و بررسی کرد. در اینجا می‌توانید مقدار TimeStamp موجودیت را از دیتابیس بگیرید و آن را با مقدار موجودیت کلاینت مقایسه کنید تا وجود تغییرات مشخص شود. این رویکرد در متد UpdateOrderByRetrieving() نمایش داده شده است. گرچه این رویکرد برای تشخیص تغییرات خواص موجودیت‌ها و یا روابط شان مفید و کارآمد است، اما بهترین روش هم نیست. مثلاً ممکن است از زمانی که موجودیت را از دیتابیس دریافت می‌کنید، تا زمانی که مقدار TimeStamp آن را مقایسه می‌کنید و نهایتاً متد SaveChanges() را صدا می‌زنید، موجودیت شما توسط کلاینت دیگری بروز رسانی شده باشد.

مسئله رویکرد دوم هزینه برتر از رویکرد اولی است، چرا که برای مقایسه مقادیر همزمانی موجودیت‌ها، یکبار موجودیت را از دیتابیس دریافت می‌کنید. اما این رویکرد در مواقعی که Object graph های بزرگ یا پیچیده (complex) دارید بهتر است، چون پیش از ارسال موجودیت‌ها به context در صورت برابر نبودن مقادیر همزمانی پروسس را لغو می‌کنید.

نظرات خوانندگان

نویسنده: Senator
تاریخ: ۱۸:۵۴ ۱۳۹۲/۱۱/۱۶

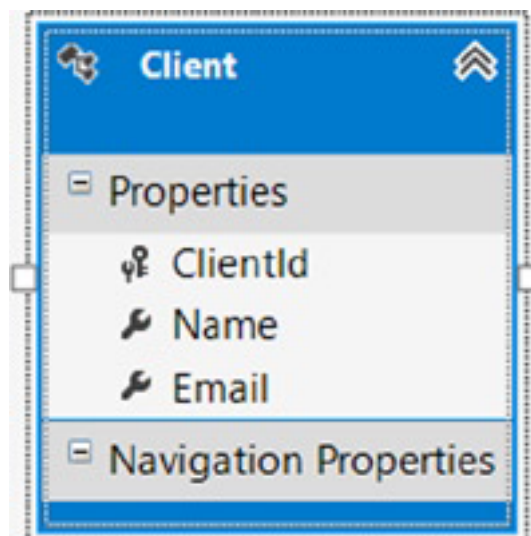
خیلی ممنون.
عالی بود ...

در [قسمت قبلی](#) مدیریت همزمانی در بروز رسانی‌ها را بررسی کردیم. در این قسمت مرتب سازی (serialization) پراکسی‌ها در سرویس‌های WCF را بررسی خواهیم کرد.

مرتب سازی پراکسی‌ها در سرویس‌های WCF

فرض کنید یک پراکسی دینامیک (dynamic proxy) از یک کوثری دریافت کرده اید. حال می‌خواهید این پراکسی را در قالب یک آبجکت CLR سریال کنید. هنگامی که آبجکت‌های موجودیت را بصورت POCO-based پیاده سازی می‌کنید، EF بصورت خودکار یک آبجکت دینامیک مشتق شده را در زمان اجرا تولید می‌کند که *dynamix proxy* نام دارد. این آبجکت برای هر موجودیت POCO تولید می‌شود. این آبجکت پراکسی بسیاری از خواص مجازی (virtual) را بازنویسی می‌کند تا عملیاتی مانند ردیابی تغییرات و بارگذاری تنبل را انجام دهد.

فرض کنید مدلی مانند تصویر زیر دارید.



ما از کلاس *ProxyDataContractResolver* برای *deserialize* کردن یک آبجکت پراکسی در سمت سرور و تبدیل آن به یک آبجکت POCO روی کلاینت WCF استفاده می‌کنیم. مراحل زیر را دنبال کنید.

در ویژوال استودیو پروژه جدیدی از نوع WCF Service Application بسازید. یک Entity Data Model به پروژه اضافه کنید و جدول Clients را مطابق مدل مذکور ایجاد کنید.

کلاس POCO تولید شده توسط EF را باز کنید و کلمه کلیدی *virtual* را به تمام خواص اضافه کنید. این کار باعث می‌شود EF کلاس‌های پراکسی دینامیک تولید کند. کد کامل این کلاس در لیست زیر قابل مشاهده است.

```
public class Client
{
    public virtual int ClientId { get; set; }
    public virtual string Name { get; set; }
    public virtual string Email { get; set; }
}
```

نکته: بیاد داشته باشید که هرگاه مدل EDMX را تغییر می‌دهید، EF بصورت خودکار کلاس‌های موجودیت‌ها را مجدداً ساخته و تغییرات مرحله قبلی را بازنویسی می‌کند. بنابراین یا باید این مراحل را هر بار تکرار کنید، یا قالب T4 مربوطه را ویرایش کنید. اگر هم از مدل Code-First استفاده می‌کنید که نیازی به این کارها نخواهد بود.

ما نیاز داریم که DataContractSerializer از یک کلاس ProxyDataContractResolver استفاده کند تا پراکسی کلاینت را به موجودیت کلاینت برای کلاینت سرویس WCF تبدیل کند. یعنی client proxy به client entity تبدیل شود، که نهایتاً در اپلیکیشن کلاینت سرویس استفاده خواهد شد. بدین منظور یک ویژگی operation behavior می‌سازیم و آن را به متد GetClient() در سرویس الحاق می‌کنیم. برای ساختن ویژگی جدید از کدی که در لیست زیر آمده استفاده کنید. بیاد داشته باشید که کلاس ProxyDataContractResolver در فضای نام Entity Framework قرار دارد.

```
using System.ServiceModel.Description;
using System.ServiceModel.Channels;
using System.ServiceModel.Dispatcher;
using System.Data.Objects;

namespace Recipe8
{
    public class ApplyProxyDataContractResolverAttribute :
        Attribute, IOperationBehavior
    {
        public void AddBindingParameters(OperationDescription description,
            BindingParameterCollection parameters)
        {
        }
        public void ApplyClientBehavior(OperationDescription description,
            ClientOperation proxy)
        {
            DataContractSerializerOperationBehavior
                dataContractSerializerOperationBehavior =
                description.Behaviors
                    .Find<DataContractSerializerOperationBehavior>();
            dataContractSerializerOperationBehavior.DataContractResolver =
                new ProxyDataContractResolver();
        }
        public void ApplyDispatchBehavior(OperationDescription description,
            DispatchOperation dispatch)
        {
            DataContractSerializerOperationBehavior
                dataContractSerializerOperationBehavior =
                description.Behaviors
                    .Find<DataContractSerializerOperationBehavior>();
            dataContractSerializerOperationBehavior.DataContractResolver =
                new ProxyDataContractResolver();
        }
        public void Validate(OperationDescription description)
        {
        }
    }
}
```

فایل IService1.cs را باز کنید و کد آن را مطابق لیست زیر تکمیل نمایید.

```
[ServiceContract]
public interface IService1
{
    [OperationContract]
    void InsertTestRecord();
    [OperationContract]
    Client GetClient();
    [OperationContract]
    void Update(Client client);
}
```

فایل IService1.svc.cs را باز کنید و پیاده سازی سرویس را مطابق لیست زیر تکمیل کنید.

```
public class Client
{
    [ApplyProxyDataContractResolver]
    public Client GetClient()
```

```

{
    using (var context = new EFRecipesEntities())
    {
        context.Configuration.LazyLoadingEnabled = false;
        return context.Clients.Single();
    }
}
public void Update(Client client)
{
    using (var context = new EFRecipesEntities())
    {
        context.Entry(client).State = EntityState.Modified;
        context.SaveChanges();
    }
}
public void InsertTestRecord()
{
    using (var context = new EFRecipesEntities())
    {
        // delete previous test data
        context.ExecuteNonQuery("delete from [clients]");
        // insert new test data
        context.ExecuteStoreCommand(@"insert into
            [clients](Name, Email) values ('Armin Zia','armin.zia@gmail.com')");
    }
}
}

```

حال پروژه جدیدی از نوع Console Application به راه حل جاری اضافه کنید. این اپلیکیشن، کلاینت تست ما خواهد بود. پروژه سرویس را ارجاع کنید و کد کلاینت را مطابق لیست زیر تکمیل نمایید.

```

using Recipe8Client.ServiceReference1;

namespace Recipe8Client
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var serviceClient = new Service1Client())
            {
                serviceClient.InsertTestRecord();

                var client = serviceClient.GetClient();
                Console.WriteLine("Client is: {0} at {1}", client.Name, client.Email);

                client.Name = "Armin Zia";
                client.Email = "arminzia@live.com";
                serviceClient.Update(client);

                client = serviceClient.GetClient();
                Console.WriteLine("Client changed to: {0} at {1}", client.Name, client.Email);
            }
        }
    }
}

```

اگر اپلیکیشن کلاینت را اجرا کنید با خروجی زیر مواجه خواهید شد.

```

Client is: Armin Zia at armin.zia@gmail.com
Client changed to: Armin Zia at arminzia@live.com

```

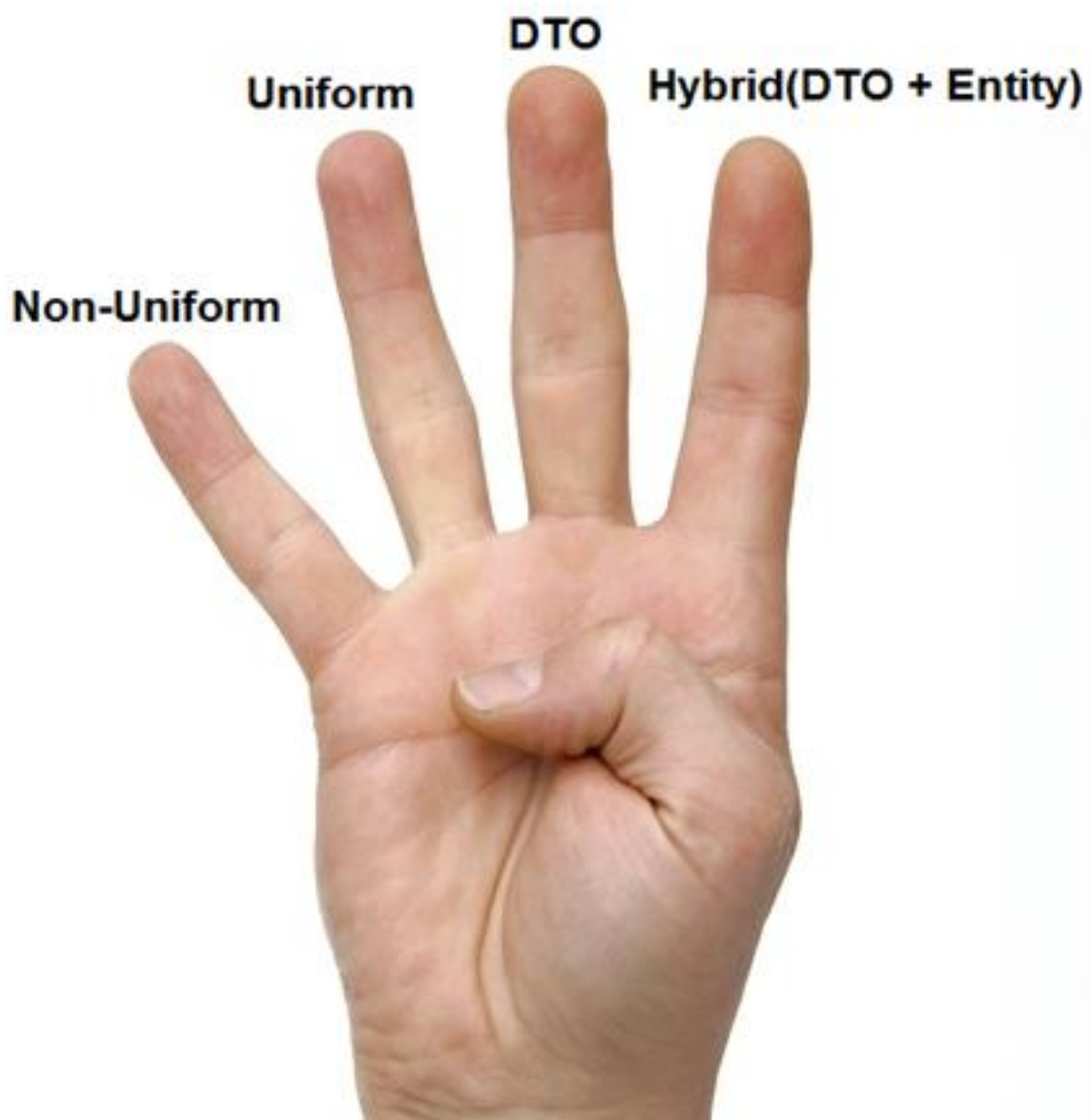
شرح مثال جاری

مایکروسافت استفاده از آبجکت های POCO با WCF را توصیه می کند چرا که مرتب سازی (serialization) آبجکت موجودیت را ساده تر می کند. اما در صورتی که از آبجکت های POCO ای استفاده می کنید که *changed-based notification* دارند (یعنی خواص موجودیت را با virtual علامت گذاری کرده اید و کلکسیون های خواص پیمایشی از نوع *ICollection* هستند)، آنگاه EF برای موجودیت های بازگشتی کوثری ها پراکسی های دینامیک تولید خواهد کرد.

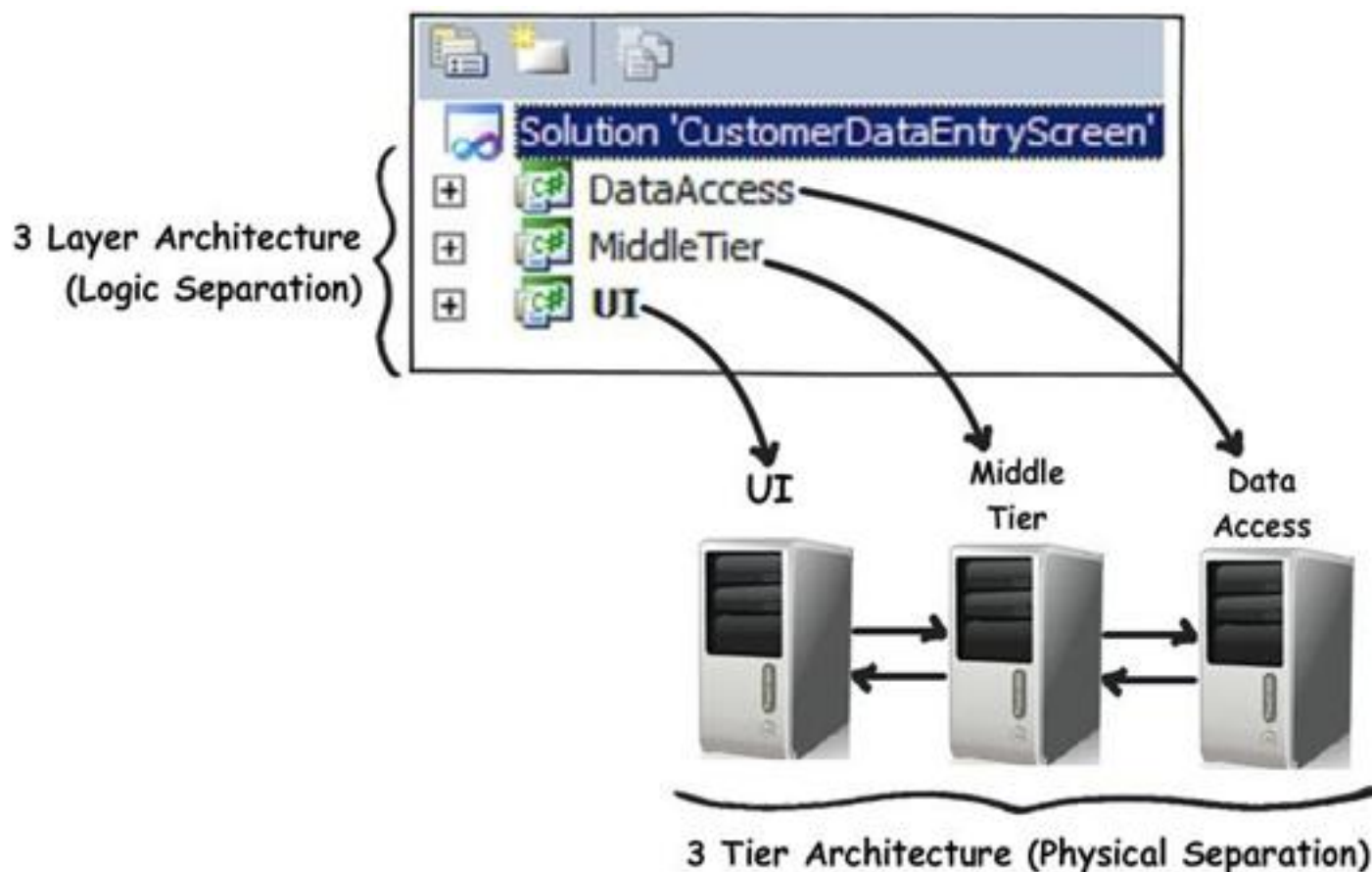
استفاده از پراکسی های دینامیک با WCF دو مشکل دارد. مشکل اول مربوط به سریال کردن پراکسی است. کلاس `DataContractSerializer` تنها قادر به `serialize/deserialize` کردن انواع شناخته شده (`known types`) است. در مثال جاری این یعنی موجودیت `Client`. اما از آنجا که EF برای موجودیت ها پراکسی می سازد، حالا باید آبجکت پراکسی را سریال کنیم، نه خود کلاس `Client` را. اینجا است که از `DataContractResolver` استفاده می کنیم. این کلاس می تواند حین سریال کردن آبجکت ها، نوعی را به نوع دیگر تبدیل کند. برای استفاده از این کلاس ما یک ویژگی سفارشی ساختیم، که پراکسی ها را به کلاس های `POCO` تبدیل می کند. سپس این ویژگی را به متد `GetClient()` اضافه کردیم. این کار باعث می شود که پراکسی دینامیکی که توسط متد `GetClient()` برای موجودیت `Client` تولید می شود، به درستی سریال شود.

مشکل دوم استفاده از پراکسی ها با WCF مربوط به بارگذاری تبل یا `Lazy Loading` می شود. هنگامی که `DataContractSerializer` موجودیت ها را سریال می کند، تمام خواص موجودیت را دستیابی خواهد کرد که منجر به اجرای `lazy-loading` روی خواص پیمایشی می شود. مسلما این رفتار را نمی خواهیم. برای رفع این مشکل، مشخصا قابلیت بارگذاری تبل را خاموش یا غیرفعال کرده ایم.

معماری لایه بندی شده، یک معماری بسیار همه گیر می باشد. به این خاطر که به راحتی decoupling ، SOC و قدرت درک کد را بسیار بالا می برد. امروزه کمتر برنامه نویس و فعال حوضه ی نرم افزاری است که با لایه های کلی و وظایف آنها آشنا نباشد (UI layer آنچه که ما می بینیم، middle layer برای مقاصد منطق کاری، data access layer برای هندل کردن دسترسی به داده ها). اما مسئله ای که بیشتر برنامه نویسان و توسعه دهندگان نرم افزار با استانداردهای آن آشنا نیستند، راه های تبادل داده ها مابین layer ها می باشد. در این مقاله سعی داریم راه های تبادل داده ها را مابین لایه ها، تشریح کنیم.

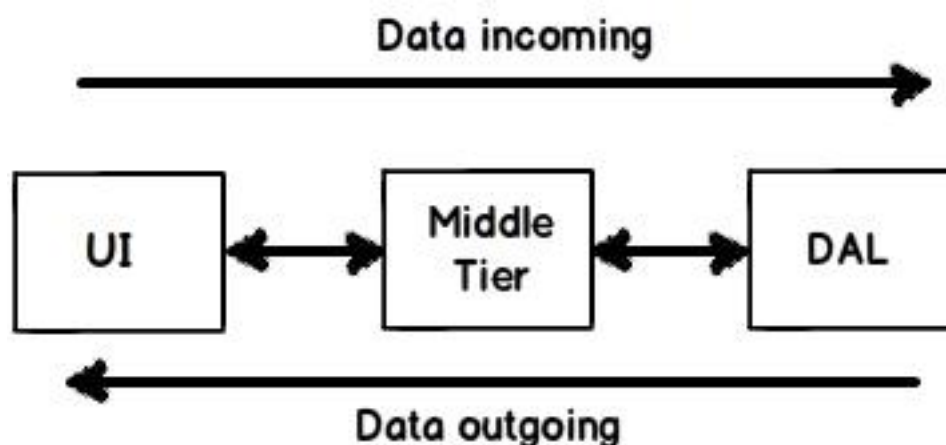


Layer با Tier متفاوت است. هنگامیکه در مورد مفهوم layer و Tier دچار شک شدید، دیاگرام ذیل می تواند به شما بسیار کمک کند. layer به مجزاسازی منطقی کد و Tier هم به مجزا سازی فیزیکی در ماشین های مختلف اطلاق می شود. توجه داشته باشید که این نکته یک شفاف سازی کلی در مورد یک مسئله مهم بود.



داده های وارد شونده (incoming) و خارج شونده (outgoing)

ما باید تبادل داده ها را از دو جنبه مورد بررسی قرار دهیم؛ اول اینکه داده ها چگونه به سمت لایه Data Access می روند، دوم اینکه داده ها چگونه به لایه UI پاس می شوند، در ادامه شما دلیل این مجزا سازی را درک خواهید کرد.



روش اول: Non-uniform

این روش اولین روش و احتمالاً عمومی‌ترین روش می‌باشد. خوب، اجازه دهید از لایه‌ی UI به لایه DAL شروع کنیم. داده‌ها از لایه UI به Middle با استفاده از getter ها و setter ها ارسال خواهد شد. کد ذیل این مسئله را به روشنی نمایش می‌دهد.

```
Customer objCust = new Customer();
objCust.CustomerCode = "c001";
objCust.CustomerName = "Shivprasad";
```

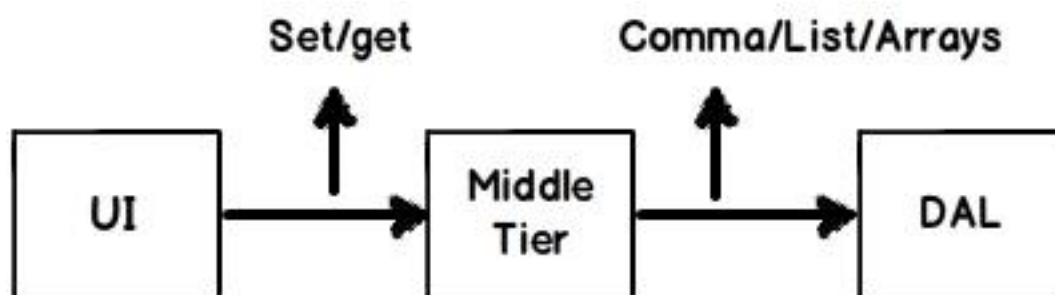
بعد از آن، از لایه Middle به لایه Data Access داده‌ها با استفاده از مجزاسازی به وسیله comma و سایر روش‌های non-uniform پاس داده می‌شوند. در کد ذیل به متد Add دقت کنید که چگونه فراخوانی به لایه Data Access را با استفاده از پارامترهای ورودی انجام می‌دهد.

```
public class Customer
{
    private string _CustomerName = "";
    private string _CustomerCode = "";
    public string CustomerCode
    {
        get { return _CustomerCode; }
        set { _CustomerCode = value; }
    }
    public string CustomerName
    {
        get { return _CustomerName; }
        set { _CustomerName = value; }
    }
    public void Add()
    {
        CustomerDal obj = new CustomerDal();
        obj.Add(_CustomerName,_CustomerCode);
    }
}
```

کد ذیل، متد add در لایه Data Access را با استفاده از دو متد نمایش می‌دهد.

```
public class CustomerDal
{
    public bool Add(string CustomerName,string CustomerCode)
    {
        // Insert data in to DB
    }
}
```

بنابراین اگر بخواهیم به صورت خلاصه نحوه پاس دادن داده ها را در روش non-uniform بیان کنیم، شکل ذیل به زیبایی این مسئله را نشان می دهد.



• از لایه UI به لایه Middle با استفاده از getter و setter

• از لایه Middle به لایه data access با استفاده از comma , input , array

حال نوبت این است بررسی کنیم که چگونه داده ها از DAL به UI در روش non-uniform پاس خواهند شد. بنابراین اجازه دهید که اول از UI شروع کنیم. از لایه UI داده ها با استفاده از object های لایه Middle واکشی می شوند.

```
Customer obj = new Customer();  
List<Customer> oCustomers = obj.getCustomers();
```

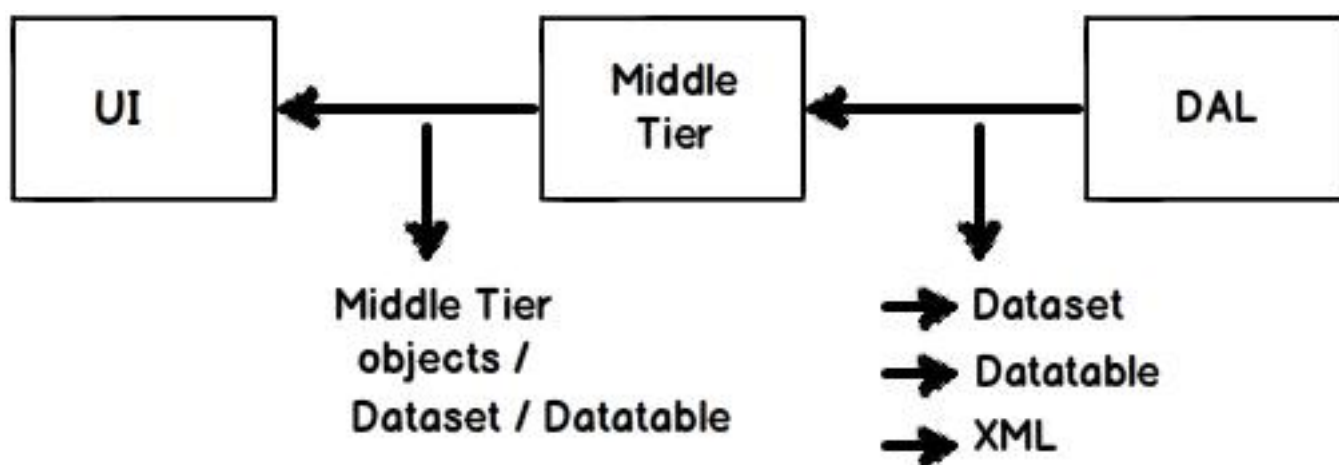
از لایه Middle هم داده ها با استفاده از datatable , dataset و xml پاس خواهند شد. مهمترین مسئله برای لایه loop , middle بر روی dataset و تبدیل آن به strong type object ها می باشد. برای مثال می توانید کد تابع getCustomers که بر روی dataset loop می زند و یک لیست از Customer ها را آماده می کند در ذیل مشاهده کنید. این تبدیل باید انجام شود، به این دلیل که UI به کلاس های strongly typed دسترسی دارد.

```
public class Customer  
{  
    private string _CustomerName = "";  
    private string _CustomerCode = "";  
    public string CustomerCode  
    {  
        get { return _CustomerCode; }  
        set { _CustomerCode = value; }  
    }  
    public string CustomerName  
    {  
        get { return _CustomerName; }  
        set { _CustomerName = value; }  
    }  
    public List<Customer> getCustomers()  
    {  
        CustomerDal obj = new CustomerDal();  
        DataSet ds = obj.getCustomers();  
        List<Customer> oCustomers = new List<Customer>();  
        foreach (DataRow orow in ds.Tables[0].Rows)  
        {  
            // Fill the list  
        }  
        return oCustomers;  
    }  
}
```


با انجام این تبدیل به یکی از بزرگترین اهداف معماری لایه بندی شده می‌رسیم؛ یعنی اینکه « UI نمی‌تواند به طور مستقیم به کامپوننت‌های لایه Data Access مانند OLEDB ، ADO.NET و غیره دستیابی داشته باشد. با این روش اگر ما در ادامه متدولوژی Data Access را تغییر دهیم تاثیری بر روی لایه UI نمی‌گذارد.» آخرین مسئله اینکه کلاس CustomerDal یک Dataset را با استفاده از ADO.NET بر می‌گرداند و Middle از آن استفاده می‌کند.

```
public class CustomerDal
{
    public DataSet getCustomers()
    {
        // fetch customer records
        return new DataSet();
    }
}
```

حال اگر بخواهیم حرکت داده‌ها را به لایه UI ، به صورت خلاصه بیان کنیم، شکل ذیل کامل این مسئله را نشان می‌دهد.



• داده‌ها از لایه DAL به لایه Middle با استفاده از XML ، Datareader ، Dataset ارسال خواهند شد.

• از لایه Middle به UI از strongly typed classes استفاده می‌شود.

مزایا و معایب روش non-uniform

یکی از مزایای non-uniform

• به راحتی قابل پیاده سازی می‌باشد، در مواردی که روش data access تغییر نمی‌کند این روش کارآیی لازم را دارد.

تعدادی از معایب این روش

• به خاطر اینکه یک ساختار uniform نداریم، بنابراین نیاز داریم که همیشه در هنگام عبور از یک لایه به یک لایه دیگر از یک ساختار به یک ساختار دیگر تبدیل را انجام دهیم.

• برنامه نویسان از روش‌های خودشان برای پاس دیتا استفاده می‌کنند؛ بنابراین این مسئله خود باعث پیچیدگی می‌شود.

• اگر برای مثال شما بخواهید متدولوژی Data Access خود را تغییر دهید، تغییرات بر تمام لایه‌ها تاثیر می‌گذارد.

نظرات خوانندگان

نویسنده: بابک جهانگیری
تاریخ: ۱۳:۲۰ ۱۳۹۴/۰۴/۰۴

آیا در این روش می توان به صورت DataView لیست مشتریها رو برگردوند به جای اینکه از `<List<Customer>` استفاده کنیم ؟ باز هم به آن non-uniform می گویند ؟

نویسنده: ریوف مدرسی
تاریخ: ۱۷:۵۳ ۱۳۹۴/۰۴/۰۵

در این روش مسئله اصلی این نیست که داده ها رو به صورت list یا DataView برگردونید، بلکه مسئله اصلی این است که شما مجبورید در گذر از هر لایه تبدیل ساختار داده ها را انجام دهید، پس نکته این روش این است که تعداد تبدیل ساختار داده ها زیاد است.

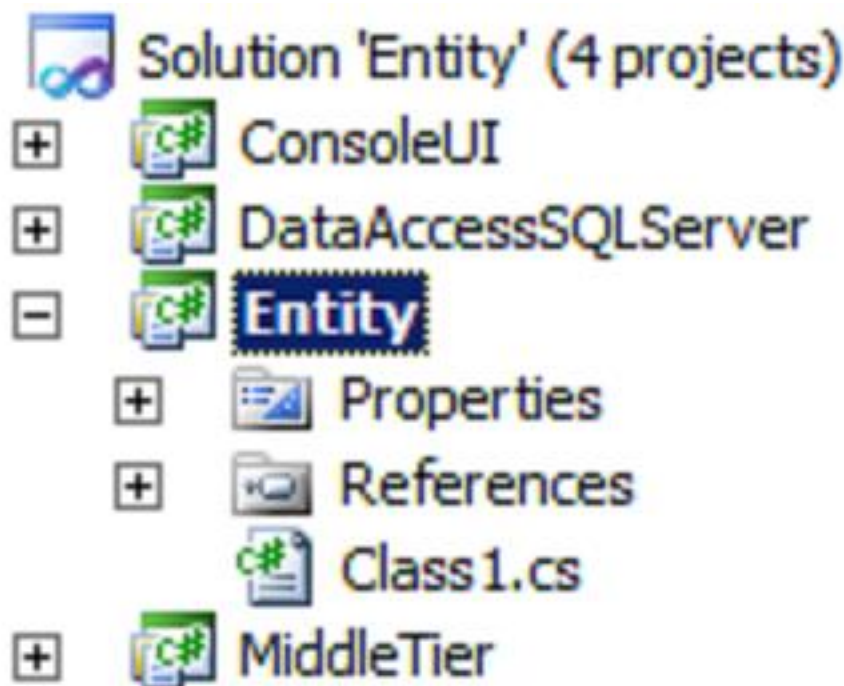
نویسنده: محسن اسماعیل پور
تاریخ: ۸:۲۵ ۱۳۹۴/۰۴/۰۸

مدل Customer که شما برای مثالهایتان از آن استفاده کرده اید از [Active record pattern](#) تبعیت میکند. از آنجا که Entity یا Model با عملیات CRUD لایه دیتا Couple شده و بعضا ممکن است Business Logic داخل این متدها قرار گیرد، این مسئله با Separation Of Concern منافات دارد.

قسمت اول : تبادل داده ها بین لایه ها- قسمت اول

روش دوم: (Uniform Entity classes)

روش دیگر پاس دادن داده ها، روش uniform است. در این روش کلاس های Entity ، یک سری کلاس ساده به همراه یکسری Property های Get و Set می باشند. این کلاس ها شامل هیچ منطق کاری نمی باشند. برای مثال کلاس CustomerEntity که دارای دو Property ، Customer Name و Customer Code می باشد. شما می توانید تمام Entity ها را به صورت یک پروژه ی مجزا ایجاد کرده و به تمام لایه ها رفرنس دهید.



```
public class CustomerEntity
{
    protected string _CustomerName = "";
    protected string _CustomerCode = "";
    public string CustomerCode
    {
        get { return _CustomerCode; }
        set { _CustomerCode = value; }
    }
    public string CustomerName
    {
        get { return _CustomerName; }
        set { _CustomerName = value; }
    }
}
```

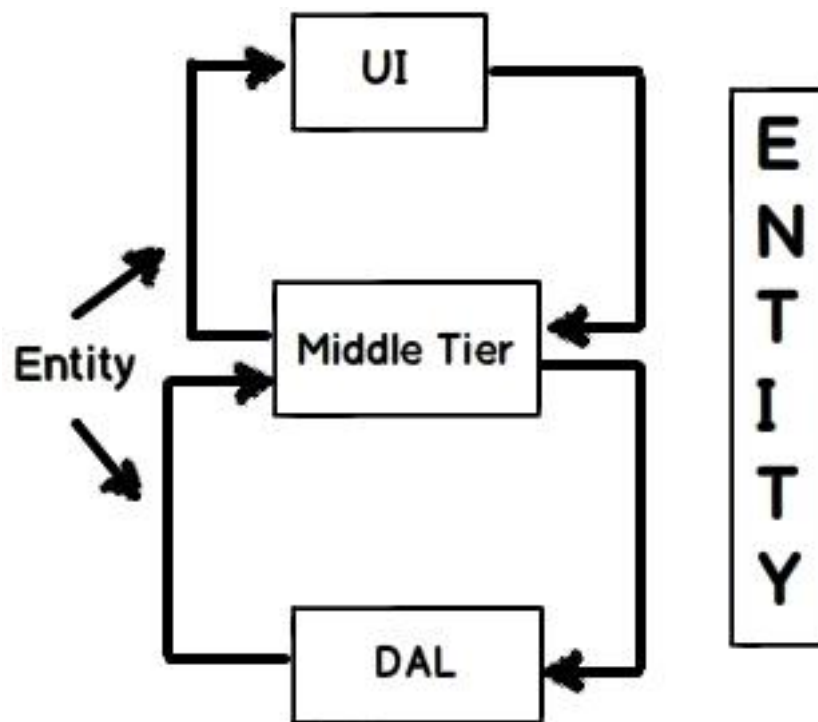
خوب، اجازه دهید تا از CustomerDal شروع کنیم. این کلاس یک Collection از CustomerEntity را بر می گرداند و همچنین یک CustomerEntity را برای اضافه کردن به دیتابیس. توجه داشته باشید که لایه Data Access وظیفه دارد تا دیتای دریافتی از دیتابیس را به CustomerEntity تبدیل کند.

```
public class CustomerDal
{
    public List<CustomerEntity> getCustomers()
    {
        // fetch customer records
        return new List<CustomerEntity>();
    }
    public bool Add(CustomerEntity obj)
    {
        // Insert in to DB
        return true;
    }
}
```

لایه Middle از CustomerEntity ارث بری می کند و یکسری operation را به entity class اضافه خواهد کرد. داده ها در قالب Entity Class به لایه Data Access ارسال می شوند و در همین قالب نیز بازگشت داده می شوند. این مسئله در کد ذیل به روشنی مشاهده می شود.

```
public class Customer : CustomerEntity
{
    public List<CustomerEntity> getCustomers()
    {
        CustomerDal obj = new CustomerDal();
        return obj.getCustomers();
    }
    public void Add()
    {
        CustomerDal obj = new CustomerDal();
        obj.Add(this);
    }
}
```

لایه UI هم با تعریف یک Customer و فراخوانی operation های مربوط به آن، داده ی مد نظر خود را در قالب CustomerEntity بازیابی خواهد کرد. اگر بخواهیم عمکرد روش uniform را خلاصه کنیم باید بگوییم، در این روش دیتای رد و بدل شده ی مابین کلیه لایه ها با یک ساختار استاندارد، یعنی Entity پاس داده می شوند.



مزایا و معایب روش uniform

مزایا

Strongly typed به صورت در تمامی لایه ها قابل دسترسی و استفاده می باشد.

```

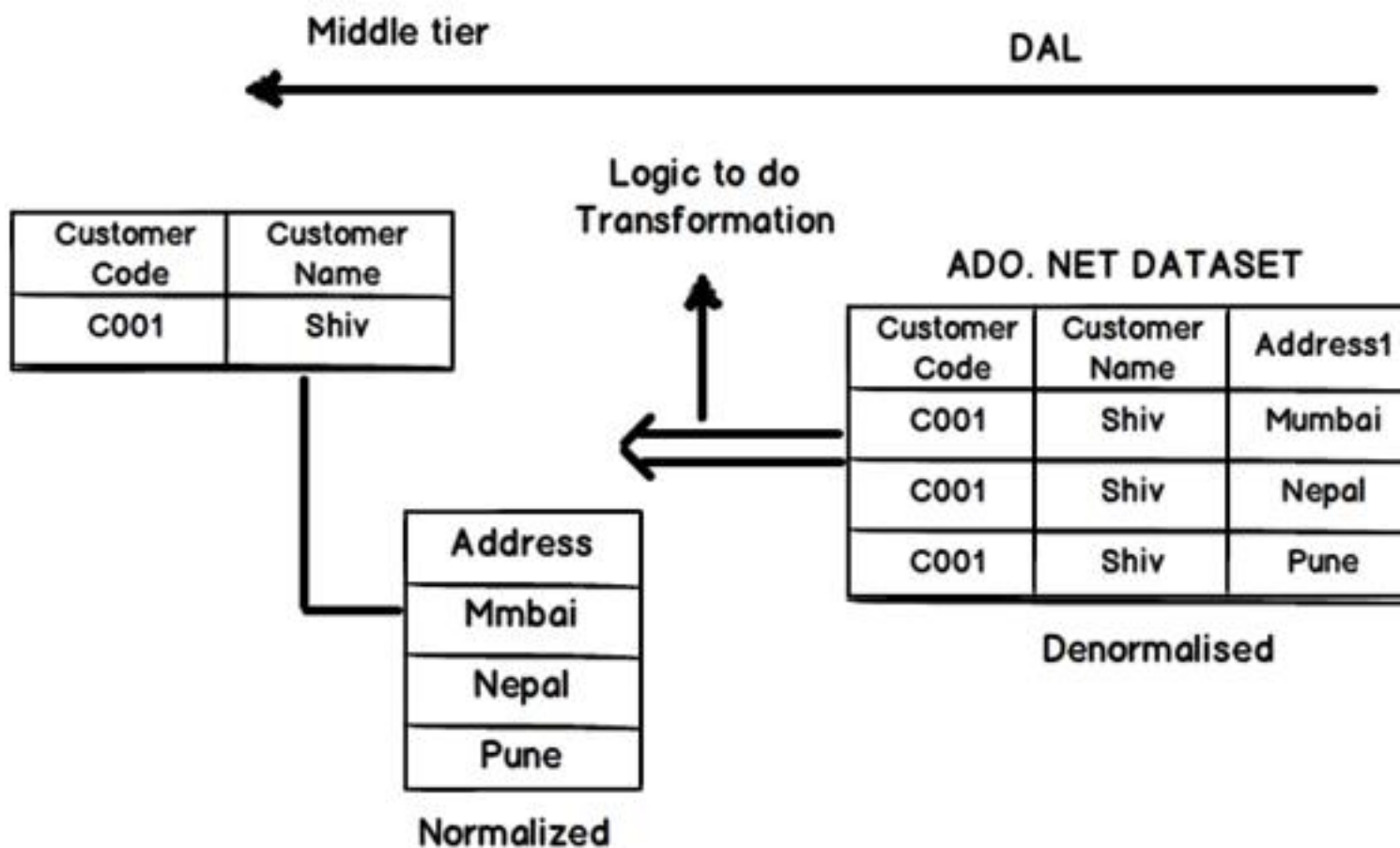
public class CustomerDal
{
    public List<CustomerEntity>
    {
        // fetch customer record
        return new List<CustomerEntity>()
    }
    public bool Add(CustomerEntity obj)
    {
        obj.CustomerCode = 1;
        obj.CustomerName = "CustomerName";
        return true;
    }
}

```

• به دلیل اینکه از ساختار عمومی Entity استفاده می‌کند، بنابراین فقط یکبار نیاز به تبدیل داده‌ها وجود دارد. به این معنی که کافی است یک بار دیتای واکنشی شده از دیتابیس را به یک ساختار Entity تبدیل کنید و در ادامه بدون هیچ تبدیل دیگری از این Entity استفاده کنید.

معایب

• تنها مشکلی که این روش دارد، مشکلی است به نام Double Loop. هنگامیکه شما در مورد کلاس‌های entity بحث می‌کنید، ساختارهای دنیای واقعی را مدل می‌کنید. حال فرض کنید شما به دلیل یکسری مسایل فنی دیتابیس خود را Optimize کرده اید. بنابراین ساختار دنیای واقعی با ساختاری که شما در نرم افزار مدل کرده‌اید متفاوت می‌باشد. بگذارید یک مثال بزنیم؛ فرض کنید که یک customer دارید، به همراه یکسری Address. همان طور که ذکر کردیم، به دلیل برخی مسایل فنی (denormalized) به صورت یک جدول در دیتابیس ذخیره شده است. بنابراین سرعت واکنشی اطلاعات بیشتر است. اما خوب اگر ما بخواهیم این ساختار را در دنیای واقعی بررسی کنیم، ممکن است با یک ساختار یک به چند مانند شکل ذیل برخورد کنیم.



بنابراین مجبوریم یکسری کد جهت این تبدیل همانند کد ذیل بنویسیم.

```
foreach (DataRow o1 in oCustomers.Tables[0].Rows)
{
    obj.Add(new CustomerEntyityAddress()); // Fills customer
    foreach (DataRow o in oAddress.Tables[0].Rows)
    {
        obj[0].Add(new AddressEntity()); // Fills address
    }
}
```

روش سوم: DTO (Data transfer objects)

در [قسمت‌های قبلی](#) دو روش از روش‌های موجود جهت تبادل داده‌ها بین لایه‌ها، ذکر گردید و علاوه بر این، مزایا و معایب هر کدام از آنها نیز ذکر شد. در این قسمت دو روش دیگر، به همراه مزایا و معایب آنها برشمرده می‌شود. لازم به ذکر است هر کدام از این روش‌ها می‌تواند با توجه به شرایط موجود و نظر طراح نرم افزار، دارای تغییراتی جهت رسیدن به یکسری اهداف و فاکتورها در نرم افزار باشد.

در این روش ما سعی می‌کنیم طراحی کلاس‌ها را به اصطلاح مسطح (flatten) کنیم تا بر مشکل double loop که در قسمت قبل بحث کردیم غلبه کنیم. در کد ذیل مشاهده می‌کنید که چگونه کلاس CustomerDTO از CustomerEntity مشتق می‌شود و کلاس Address را با CustomerEntity ادغام می‌کند؛ تا برای افزایش سرعت لود و نمایش داده‌ها، یک کلاس de-normalized شده ایجاد نماید.

```
public class CustomerDTO : CustomerEntity
{
    public AddressEntity _Address = new AddressEntity();
}
```

در کد ذیل می‌توانید مشاهده کنید که چگونه با استفاده از فقط یک loop یک کلاس de-normalized شده را پر می‌کنیم.

```
foreach (DataRow o1 in oCustomers.Tables[0].Rows)
{
    CustomerDTO o = new CustomerDTO();
    o.CustomerCode = o1[0].ToString();
    o.CustomerName = o1[1].ToString();
    o._Address.Address1 = o1[2].ToString();
    o._Address.Address2 = o1[3].ToString();
    obj.Add(o);
}
```

UI هم به راحتی می‌تواند DTO را فراخوانی کرده و دیتا را دریافت کند.

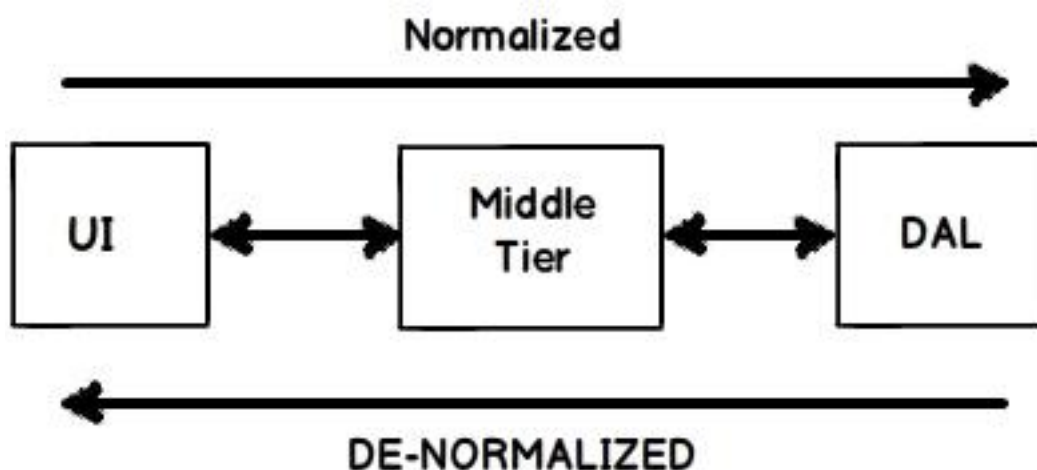
مزایا و معایب روش DTO

یکی از بزرگترین مزایای این روش سرعت زیاد در بارگذاری اطلاعات، به دلیل استفاده کردن از ساختار de-normalized می‌باشد. اما همین مسئله خود یک عیب محسوب می‌شود؛ به این دلیل که اصول شی گرای را نقض می‌کند.

روش چهارم: Hybrid approach (Entity + DTO)

از یک طرف کلاس‌های Entity که دنیای واقعی را مدل خواهند کرد و همچنین اصول شی گرای را رعایت می‌کنند و از یک طرف دیگر DTO نیز یک ساختار flatten را برای رسیدن به اهداف کارآیی دنبال خواهند کرد. خوب، به نظر می‌رسد که بهترین کار استفاده از هر دو روش و مزایای آن روش‌ها باشد.

زمانیکه سیستم، اهدافی مانند انجام اعمال CRUD را دنبال می‌کند و شما می‌خواهید مطمئن شوید که اطلاعات، دارای integrity می‌باشند و یا اینکه می‌خواهید این ساختار را مستقیماً به کاربر نهایی ارائه دهید، استفاده کردن از روش (Entity) به عنوان یک روش normalized می‌تواند بهترین روش باشد. اما اگر می‌خواهید حجم بزرگی از دیتا را نمایش دهید، مانند گزارشات طولانی، بنابراین استفاده از روش DTO با توجه به اینکه یک روش de-normalized به شمار می‌رود بهترین روش می‌باشد.



کدام روش بهتر است؟

Non-uniform : این روش برای حالتی است که متدهای مربوط به data access تغییرات زیادی را تجربه نخواهند کرد. به عبارت دیگر، اگر پروژه‌ی شما در آینده دیتابیس‌های مختلفی را مبتنی بر تکنولوژی‌های متفاوت، لازم نیست پشتیبانی کند، این روش می‌تواند بهترین روش باشد.

Uniform : Entity, DTO, or hybrid : اگر امکان دارد که پروژه‌ی شما با انواع مختلف دیتابیس‌ها مانند Oracle و Postgres ارتباط برقرار کند، استفاده کردن از این روش پیشنهاد می‌شود.