عنوان: **بروز رسانی استفاده از SqlServer Compact در Entityframework 6.0**
نویسنده: مهدی پایروند
تاریخ: ۱۳۹۲/۰۸/۱۵ ۱۰:۵
آدرس: www.dotnettips.info
برچسب‌ها: Entity framework, NuGet, SQL Server CE, Entity Framework 6, Update-Package

تغییراتی در Entity framework 6 صورت گرفته و در ذیل لیستی از موارد آن آمده است. همچنین پیشتر در همین سایت نیز به آن‌ها اشاره‌ای شده که باز تولید پروایدرها برای نسخه جدید Entity framework یکی از آن‌ها می‌باشد:

```
Rebuilding EF Providers for EF6
Updating Applications to use EF6
EF Tools: adding EF6 support & enabling out-of-band releases
Async Query and Save
Connection Resiliency
Code-Based Configuration
Dependency Resolution
Interception/SQL logging
Custom implementations of Equals or GetHashCode on entity classes
Custom Code First Conventions
Code First Mapping to Insert/Update/Delete Stored Procedures
Configurable Migrations History Table
Multiple Contexts per Database
```

اکنون برای به‌روز رسانی به نسخه جدید، جهت ادامه استفاده از SqlServer Compact مواردی باید لحاظ شود که به آنها اشاره خواهیم کرد و قبل از آنها رعایت یک سری از پیشنیازها لازم است. برای مثال در وب کانفیگ برای استفاده از پروایدر SqlServer Compact بعنوان پروایدر پیش فرض باید قسمت مربوطه را به نحو ذیل تغییر داده باشیم:

```
<entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.SqlCeConnectionFactory,
EntityFramework">
      <parameters>
        <parameter value="System.Data.SqlServerCe.4.0" />
      </parameters>
    </defaultConnectionFactory>
    <providers>
      <provider invariantName="System.Data.SqlServerCe.4.0"
type="System.Data.Entity.SqlServerCompact.SqlCeProviderServices, EntityFramework.SqlServerCompact" />
    </providers>
  </entityFramework>
```

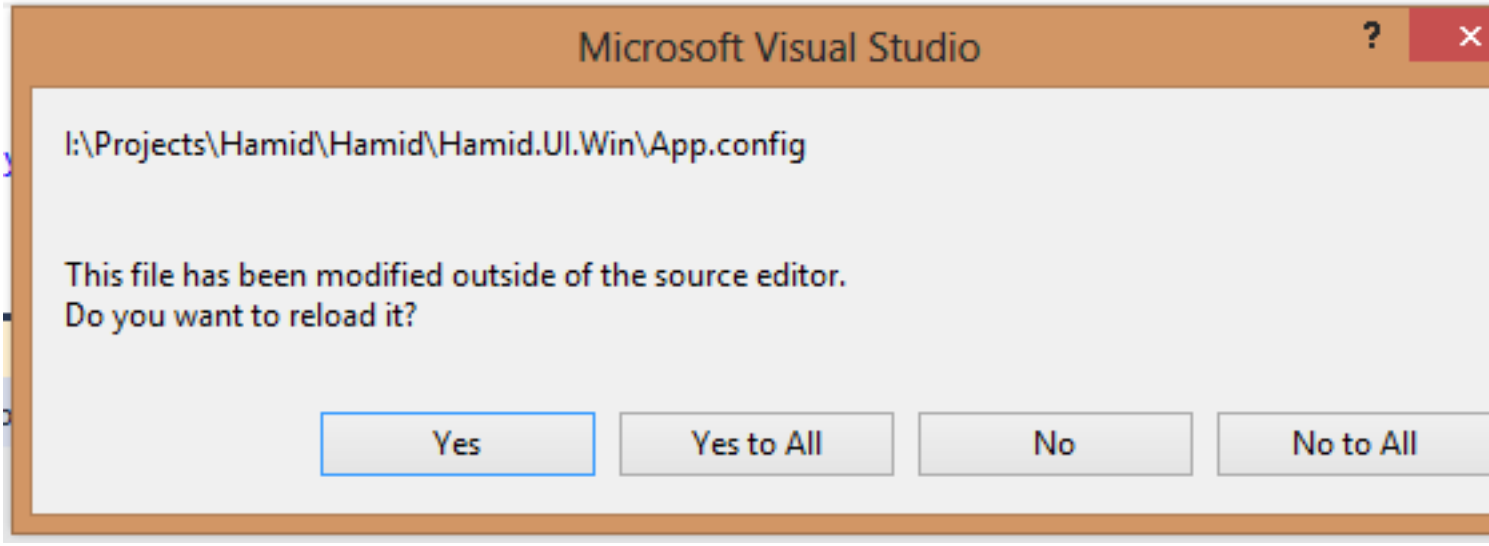
حالا در کنسول نیوگت دستور زیر را برای به‌روزرسانی فقط Entity Framework وارد و اجرا میکنیم:

```
Update-Package EntityFramework
```

پیغام موفقیت آمیز بودن به‌روز رسانی در خروجی نیوگت ظاهر می‌شود

```
PM> Update-Package EntityFramework
Updating 'EntityFramework' from version '5.0.0' to '6.0.1' in project 'Hamid.Core.Model'.
Removing 'EntityFramework 5.0.0' from Hamid.Core.Model.
Successfully removed 'EntityFramework 5.0.0' from Hamid.Core.Model.
```

و نیز تاییدی برای اعمال تغییرات به‌روز رسانی Entity framework انجام میشود تا فایل کانفیگ پروژه را تغییر دهد:



این تغییرات شامل موارد ذیل می‌باشند (در صورت به‌روز رسانی دستی، منظور کپی پکیج بصورت دستی، اعمال تغییرات در کانفیگ‌ها مورد نیاز است):

```
<!-- 1. Change in <configuration><configSections> -->
    <section name="entityFramework"
type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework, Version=4.4.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089" requirePermission="false" />
    <section name="entityFramework"
type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework, Version=6.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089" requirePermission="false" />
<!-- 2. Add in <entityFramework><providers> -->
      <provider invariantName="System.Data.SqlClient"
type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlServer" />
```

بعد از به‌روز رسانی Entityframework باید پکیج EntityFramework.SqlServerCompact برای ادامه استفاده از پروایدر نصب شود که با دستور نیوگت زیر این امر نیز میسر است:

```
PM> Install-Package EntityFramework.SqlServerCompact
```

حالا بدون مشکل می‌توان از پروژه بیلد گرفت و کار توسعه را ادامه داد.

عنوان: **ایجاد سرویس چندلایه‌ی WCF با Entity Framework در قالب پروژه - 2**
نویسنده: حامد قنادی
تاریخ: ۱۳۹۲/۱۰/۲۵ ۲۰:۳۰
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, LINQ, WCF, Entity Framework 6

برای استفاده از کلاس‌های Entity که در نوشتار پیشین ایجاد کردیم در WCF باید آن کلاس‌ها را دست‌کاری کنیم. متن کلاس tblNews را در نظر بگیرید:

```
namespace MyNewsWCFLibrary
{
    using System;
    using System.Collections.Generic;

    public partial class tblNews
    {
        public int tblNewsId { get; set; }
        public int tblCategoryId { get; set; }
        public string Title { get; set; }
        public string Description { get; set; }
        public System.DateTime RegDate { get; set; }
        public Nullable<bool> IsDeleted { get; set; }

        public virtual tblCategory tblCategory { get; set; }
    }
}
```

مشاهده می‌کنید که برای تعریف کلاس‌ها از کلمه کلیدی partial استفاده شده است. استفاده از کلمه کلیدی partial به شما اجازه می‌دهد که یک کلاس را در چندین فایل جداگانه تعریف کنید. به عنوان مثال می‌توانید فیلدها، ویژگی ها و سازنده‌ها را در یک فایل و متدها را در فایل دیگر قرار دهید.
به صورت خودکار کلیه‌ی ویژگی‌ها به توجه به پایگاه داده ساخته شده اند. برای نمونه ما برای فیلد IsDeleted در SQL Server ستون Allow Nulls را کلیک کرده بودیم که در نتیجه در اینجا عبارت Nullable پیش از نوع فیلد نشان داده شده است. برای استفاده از این کلاس در WCF باید صفت DataContract را به کلاس داد. این قرارداد به ما اجازه استفاده از ویژگی‌هایی که صفت DataMember را می‌گیرند را می‌دهد.
کلاس بالا را به شکل زیر بازنویسی کنید:

```
using System.Runtime.Serialization;

namespace MyNewsWCFLibrary
{
    using System;
    using System.Collections.Generic;

    [DataContract]
    public partial class tblNews
    {
        [DataMember]
        public int tblNewsId { get; set; }
        [DataMember]
        public int tblCategoryId { get; set; }
        [DataMember]
        public string Title { get; set; }
        [DataMember]
        public string Description { get; set; }
        [DataMember]
        public System.DateTime RegDate { get; set; }
        [DataMember]
        public Nullable<bool> IsDeleted { get; set; }

        public virtual tblCategory tblCategory { get; set; }
    }
}
```

هم‌چنین کلاس tblCategory را به صورت زیر تغییر دهید:

```
namespace MyNewsWCFLibrary
{
    using System;
```

```
    using System.Collections.Generic;
    using System.Runtime.Serialization;

    [DataContract]
    public partial class tblCategory
    {
        public tblCategory()
        {
            this.tblNews = new HashSet<tblNews>();
        }

        [DataMember]
        public int tblCategoryId { get; set; }
        [DataMember]
        public string CatName { get; set; }
        [DataMember]
        public bool IsDeleted { get; set; }

        public virtual ICollection<tblNews> tblNews { get; set; }
    }
}
```

با انجام کد بالا از بابت مدل کارمان تمام شده است. ولی فرض کنید در اینجا تصمیم به تغییری در پایگاه داده می‌گیرید. برای نمونه می‌خواهید ویژگی Allow Nulls فیلد IsDeleted را نیز False کنیم و مقدار پیش‌گزیده به این فیلد بدهید. برای این کار باید دستور زیر را در SQL Server اجرا کنیم:

```
BEGIN TRANSACTION
GO
ALTER TABLE dbo.tblNews
DROP CONSTRAINT FK_tblNews_tblCategory
GO
ALTER TABLE dbo.tblCategory SET (LOCK_ESCALATION = TABLE)
GO
COMMIT
BEGIN TRANSACTION
GO
CREATE TABLE dbo.Tmp_tblNews
(
tblNewsId int NOT NULL IDENTITY (1, 1),
tblCategoryId int NOT NULL,
Title nvarchar(50) NOT NULL,
Description nvarchar(MAX) NOT NULL,
RegDate datetime NOT NULL,
IsDeleted bit NOT NULL
)  ON [PRIMARY]
 TEXTIMAGE_ON [PRIMARY]
GO
ALTER TABLE dbo.Tmp_tblNews SET (LOCK_ESCALATION = TABLE)
GO
ALTER TABLE dbo.Tmp_tblNews ADD CONSTRAINT
DF_tblNews_IsDeleted DEFAULT 0 FOR IsDeleted
GO
SET IDENTITY_INSERT dbo.Tmp_tblNews ON
GO
IF EXISTS(SELECT * FROM dbo.tblNews)
 EXEC('INSERT INTO dbo.Tmp_tblNews (tblNewsId, tblCategoryId, Title, Description, RegDate, IsDeleted)
SELECT tblNewsId, tblCategoryId, Title, Description, RegDate, IsDeleted FROM dbo.tblNews WITH (HOLDLOCK
TABLOCKX)')
GO
SET IDENTITY_INSERT dbo.Tmp_tblNews OFF
GO
DROP TABLE dbo.tblNews
GO
EXECUTE sp_rename N'dbo.Tmp_tblNews', N'tblNews', 'OBJECT'
GO
ALTER TABLE dbo.tblNews ADD CONSTRAINT
PK_tblNews PRIMARY KEY CLUSTERED
(
tblNewsId
) WITH( STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS =
ON) ON [PRIMARY]

GO
ALTER TABLE dbo.tblNews ADD CONSTRAINT
FK_tblNews_tblCategory FOREIGN KEY
(
tblCategoryId
) REFERENCES dbo.tblCategory
```

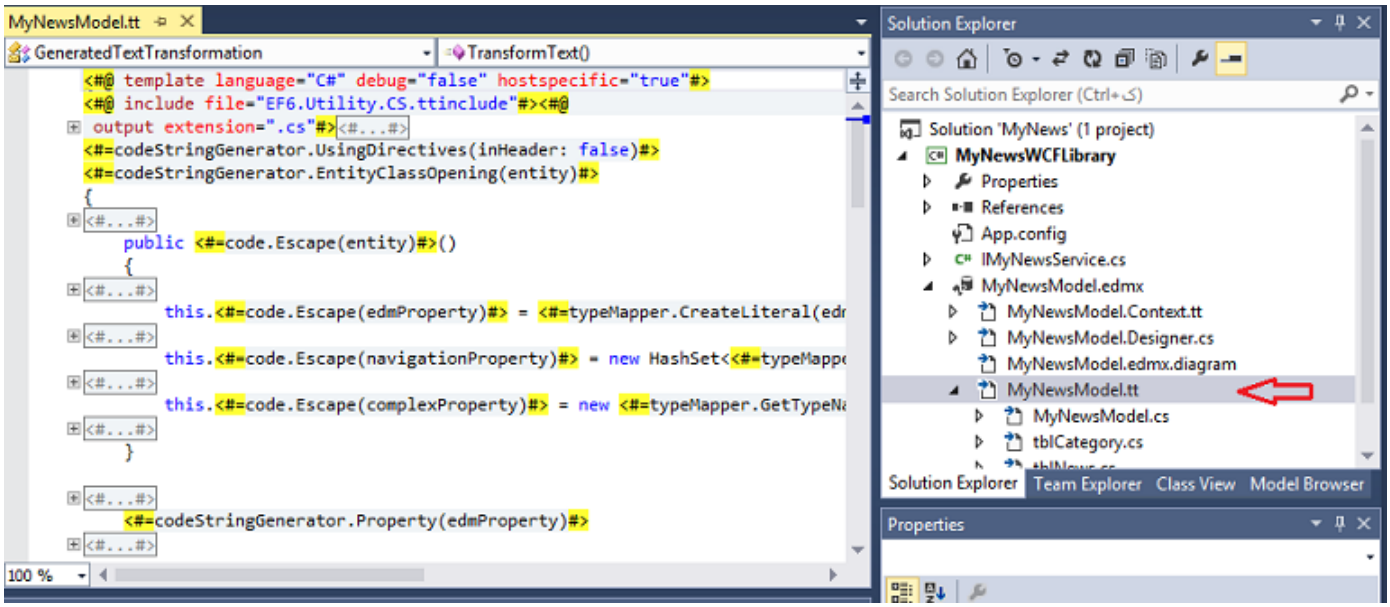
```
(
tblCategoryId
) ON UPDATE  NO ACTION
 ON DELETE  NO ACTION

GO
COMMIT
```

پس از آن مدل Entity Framework را باز کنید و در جایی از صفحه راست‌کلیک کرده و از منوی بازشده گزینه Update Model from Database را انتخاب کنید. سپس در پنجره بازشده، چون هیچ جدول، نما یا روالی به پایگاه داده‌ها نیفزوده ایم؛ دگمه‌ی Finish را کلیک کنید. دوباره کلاس tblNews را بازکنید. متوجه خواهید شد که همه‌ی DataContractها و DataMemberها را حذف شده است. ممکن است بگویید می‌توانستیم کلاس یا مدل را تغییر دهیم و به وسیله‌ی Generate Database from Model به‌هنگام کنیم. ولی در نظر بگیرید که نیاز به ایجاد چندین جدول دیگر داریم و مدلی با ده‌ها Entity دارید. در این صورت همه‌ی تغییراتی که در کلاس داده ایم زدوده خواهد شد.

در بخش پسین، درباره‌ی این‌که چه کنیم که عبارت‌هایی که به کلاس‌ها می‌افزاییم حذف نشود؛ خواهم نوشت.

عنوان: **ایجاد سرویس چندلایه‌ی WCF با Entity Framework در قالب پروژه - 3**
نویسنده: حامد قنادی
تاریخ: ۱۳۹۲/۱۰/۲۶ ۱۴:۵
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, LINQ, WCF, Entity Framework 6

پیش از ادامه‌ی نوشتار بهتر است توضیحاتی درباره‌ی قالب‌های T4 داده شود. این قالب‌های مصنوعی حاوی کدهایی که است که هدف آن صرفه‌جویی در نوشتن کد توسط برنامه نویس است. مثلاً در MVC شما یکبار قالبی برای صفحه Index خود تهیه می‌کنید که برای نمونه بجای ساخت جدول ساده، از گرید Kendo استفاده کند و همچنین دارای دکمه ویرایش و جزئیات باشد. از این پس هر بار که نیاز به ساخت یک نمای نوع لیست برای یک ActionResult داشته باشید فرم ساز MVC از قالب شما استفاده خواهد کرد.
روشن است که خود Visual Studio نیز از T4 در ساخت بسیاری از فرم‌ها و کلاس‌ها بهره می‌برد.
خبر خوب این‌که برای ساخت کلاس‌های هر موجودیت در Entity Framework نیز از قالب‌های T4 استفاده می‌شود و این‌که این قالب‌ها در دسترس توسعه‌دهندگان برای ویرایش یا افزودن است.
افزونه‌ی Tangible را دریافت کنید و سپس نصب کنید. این افزونه ظاهر نامفهوم قالب‌های T4 را ساده و روشن می‌کند.
ما نیاز داریم که خود Visual Studio زحمت این سه کار را بکشد:
1- بالای هر کلاس موجودیت عبارت using System.Runtime.Serialization; را بنویسید.
2- صفت [DataContract] را پیش از تعریف کلاس بیفزاید.
3- صفت [DataMember] را پیش از تعریف هر ویژگی بیفزاید.
همانند شکل زیر روی فایل MyNewsModel.tt دوکلیک کنید تا محتوای آن در سمت چپ نشان داده شود. این محتوا باید ظاهری همانند شکل پیدا کرده باشد:



کد زیر را در محتوای فایل جست‌وجو کنید:

```
public string Property(EdmProperty edmProperty)
    {
        return string.Format(
            CultureInfo.InvariantCulture,
            "{0} {1} {2} {{ {3}get; {4}set; }}",
            Accessibility.ForProperty(edmProperty),
            _typeMapper.GetTypeName(edmProperty.TypeUsage),
            _code.Escape(edmProperty),
            _code.SpaceAfter(Accessibility.ForGetter(edmProperty)),
            _code.SpaceAfter(Accessibility.ForSetter(edmProperty)));
```

```
    }
```

متن آن‌را به این صورت تغییر دهید:

```
public string Property(EdmProperty edmProperty)
    {
        return string.Format(
            CultureInfo.InvariantCulture,
"[DataMember]" + Environment.NewLine +
            "{0} {1} {2} {{ {3}get; {4}set; }}",
            Accessibility.ForProperty(edmProperty),
            _typeMapper.GetTypeName(edmProperty.TypeUsage),
            _code.Escape(edmProperty),
            _code.SpaceAfter(Accessibility.ForGetter(edmProperty)),
            _code.SpaceAfter(Accessibility.ForSetter(edmProperty)));
    }
```

بار دیگر به دنبال این کد بگردید:

```
public string EntityClassOpening(EntityType entity)
    {
        return string.Format(
            CultureInfo.InvariantCulture,
            "{0} {1}partial class {2}{3}",
            Accessibility.ForType(entity),
            _code.SpaceAfter(_code.AbstractOption(entity)),
            _code.Escape(entity),
            _code.StringBefore(" : ", _typeMapper.GetTypeName(entity.BaseType)));
    }
```

این کد را نیز به این صورت تغییر دهید:

```
public string EntityClassOpening(EntityType entity)
    {
        return string.Format(
            CultureInfo.InvariantCulture,
"[DataContract]" + Environment.NewLine +
            "{0} {1}partial class {2}{3}",
            Accessibility.ForType(entity),
            _code.SpaceAfter(_code.AbstractOption(entity)),
            _code.Escape(entity),
            _code.StringBefore(" : ", _typeMapper.GetTypeName(entity.BaseType)));
    }
```

برای واپسین تغییر به دنبال کد زیر بگردید:

```
public string UsingDirectives(bool inHeader, bool includeCollections = true)
    {
        return inHeader == string.IsNullOrEmpty(_code.VsNamespaceSuggestion())
            ? string.Format(
                CultureInfo.InvariantCulture,
                "{0}using System;{1}" +
                "{2}",
                inHeader ? Environment.NewLine : "",
                includeCollections ? (Environment.NewLine + "using System.Collections.Generic;") : "",
                inHeader ? "" : Environment.NewLine)
            : "";
    }
```

سپس کد زیر را جاگزین آن کنید:

```
public string UsingDirectives(bool inHeader, bool includeCollections = true)
    {
        return inHeader == string.IsNullOrEmpty(_code.VsNamespaceSuggestion())
            ? string.Format(
                CultureInfo.InvariantCulture,
"using System.Runtime.Serialization;" + Environment.NewLine +
```

```
                "{0}using System;{1}" +
                "{2}",
                inHeader ? Environment.NewLine : "",
                includeCollections ? (Environment.NewLine + "using System.Collections.Generic;") : "",
                inHeader ? "" : Environment.NewLine)
            : "";
    }
```

فایل MyNewsModel.tt را ذخیره کنید و از آن خارج شوید. بار دیگر هر کدام از کلاس‌های tblNews و tblCategory را باز کنید. خواهید دید که به صورت خودکار تغییرات مد نظر ما به آن افزوده شده است. از این پس بدون هیچ دلواپسی بابت حذف صفت‌ها، می‌توانید هرچند بار که خواستید مدل خود را به‌هنگام کنید.

در بخش پسین دوباره به WCF بازخواهیم گشت و به تعریف روال‌های مورد نیاز خواهیم پرداخت.

نظرات خوانندگان

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۱۰/۲۶ ۱۴:۸

با تشکر از شما. روش دیگری برای حل مساله استفاده از AOP است:

استفاده از IL Code Weaving برای تولید ویژگی‌های تکراری مورد نیاز در WCF

نویسنده: حمید
تاریخ: ۱۳۹۲/۱۰/۲۷ ۴:۱

هرچند که به نکته خوبی، اشاره کردین اما این کار از اساس غلط است چون شما دارید کلاسهای لایه داده خود را expose می‌کنید. سرویس‌ها بادید DTOها را به بیرون EXPOSE کنند و تبدیل کلاسهای لایه BUSINESS به dtoها از طریق ابزاری مثل AUTOMAPPER انجام می‌شود. متشکرم

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۱۰/۲۷ ۹:۲۸

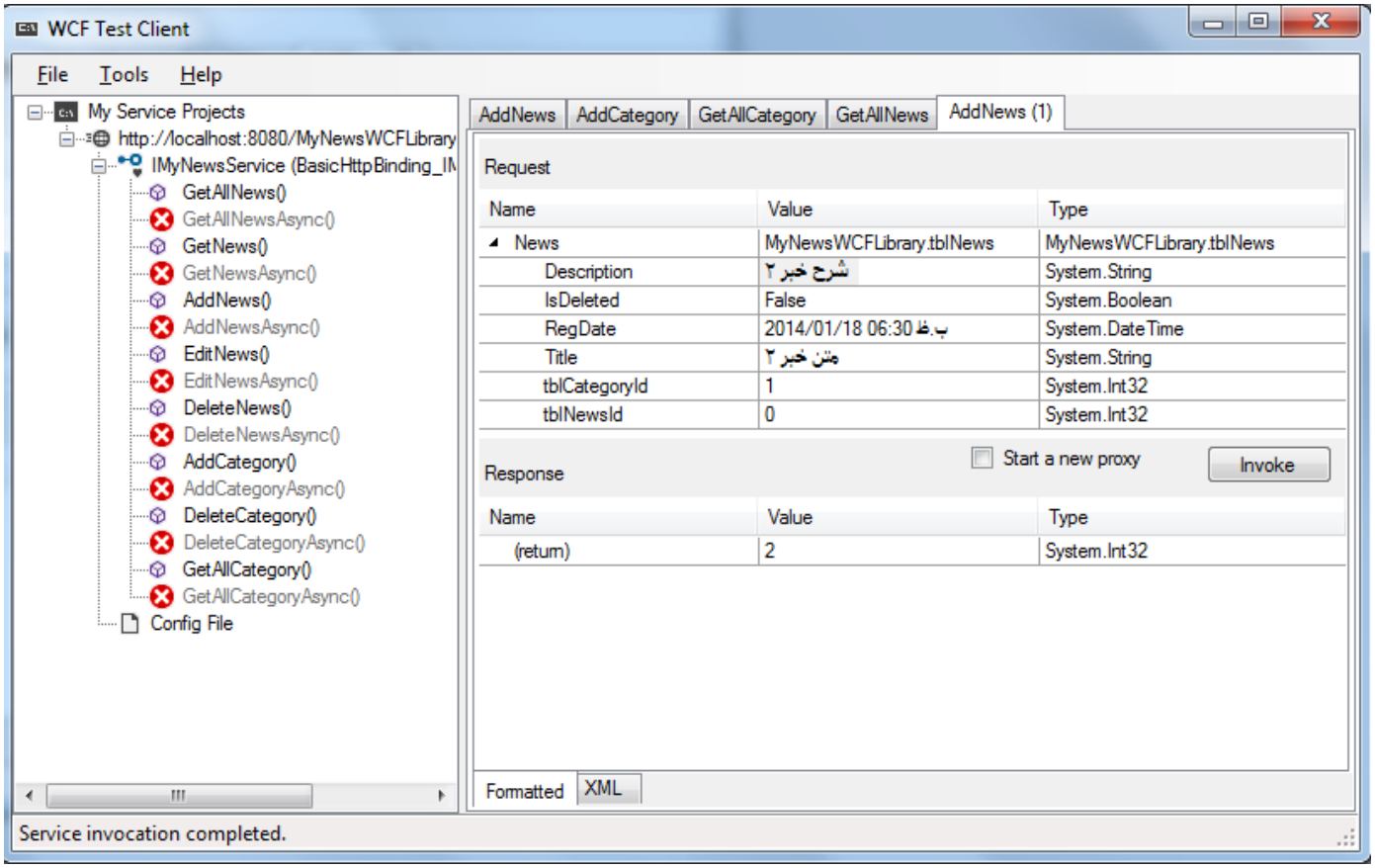
بایدی وجود ندارد در این حالت و بهتر است که اینگونه باشد یا حتی مخلوطی از این دو در عمل:

Pros and Cons of Data Transfer Objects

In large projects with so many entities, DTOs add a remarkable level of (extra) complexity and work to do. In short, a pure, 100% DTO solution is often just a 100 percent painful solution

عنوان: **ایجاد سرویس چندلایه‌ی WCF با Entity Framework در قالب پروژه - 6**
نویسنده: حامد قنادی
تاریخ: ۱۳۹۲/۱۰/۲۹ ۵:۱۰
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, WCF, Entity Framework 6

پروژه را اجرا کنید و در WCF Test Client به وسیله‌ی متد AddNews دو خبر جدید درج کنید.



روی متدهای GetAllCategory و GetAllNews به صورت جداگانه کلیک کنید. متوجه خواهید شد که هرچند در کلاس tblNews شی‌ای از نوع tblCategory و در کلاس tblCategory شی‌ای از نوع مجموعه‌ی tblNews به صورت Virtual تعریف شده است ولی در بر خلاف انتظارمان اثری از آن در این‌جا دیده نمی‌شود. نتیجه‌ی مشاهده‌شده به خاطر است که در هر دو تعریف صفت DataMember را به ویژگی‌های ناوبری اختصاص نداده ایم و این می‌تواند راهبرد ما در طراحی WCF باشد. ولی اگر می‌خواهید ویژگی ناوبری میان موجودیت‌ها در متدهای ما هم دیده شود ادامه‌ی این درس را بخوانید وگرنه ممکن است تصمیم داشته باشید در صورت نیاز به پیوند میان موجودیت‌ها، متد جدیدی بنویسید و از دستورهای Linq استفاده کنید و یا برای این‌کار از Stored Procedured بهره ببرید.

در اینجا من این سناریو را دنبال می‌کنم که در صورتی که متد GetAllNews اجرا شود؛ بدون این‌که نیاز باشد برای دانستن نام دسته‌ی خبر از متد دیگری مانند GetAllCategory استفاده کنیم؛ رکورد وابسته موجودیت دسته در هر خبر نشان داده شود.

از Solution Explorer فایل MyNewsModel.tt را باز کنید و دنبال کد زیر بگردید:

```
public string NavigationProperty(NavigationProperty navigationProperty)
    {
        var endType = _typeMapper.GetTypeName(navigationProperty.ToEndMember.GetEntityType());
        return string.Format(
```

```
            CultureInfo.InvariantCulture,
            "{0} {1} {2} {{ {3}get; {4}set; }}",
            AccessibilityAndVirtual(Accessibility.ForProperty(navigationProperty)),
            navigationProperty.ToEndMember.RelationshipMultiplicity == RelationshipMultiplicity.Many ?
("ICollection<" + endType + ">") : endType,
            _code.Escape(navigationProperty),
            _code.SpaceAfter(Accessibility.ForGetter(navigationProperty)),
            _code.SpaceAfter(Accessibility.ForSetter(navigationProperty)));
    }
```

سپس آن‌را به صورت زیر ویرایش کنید:

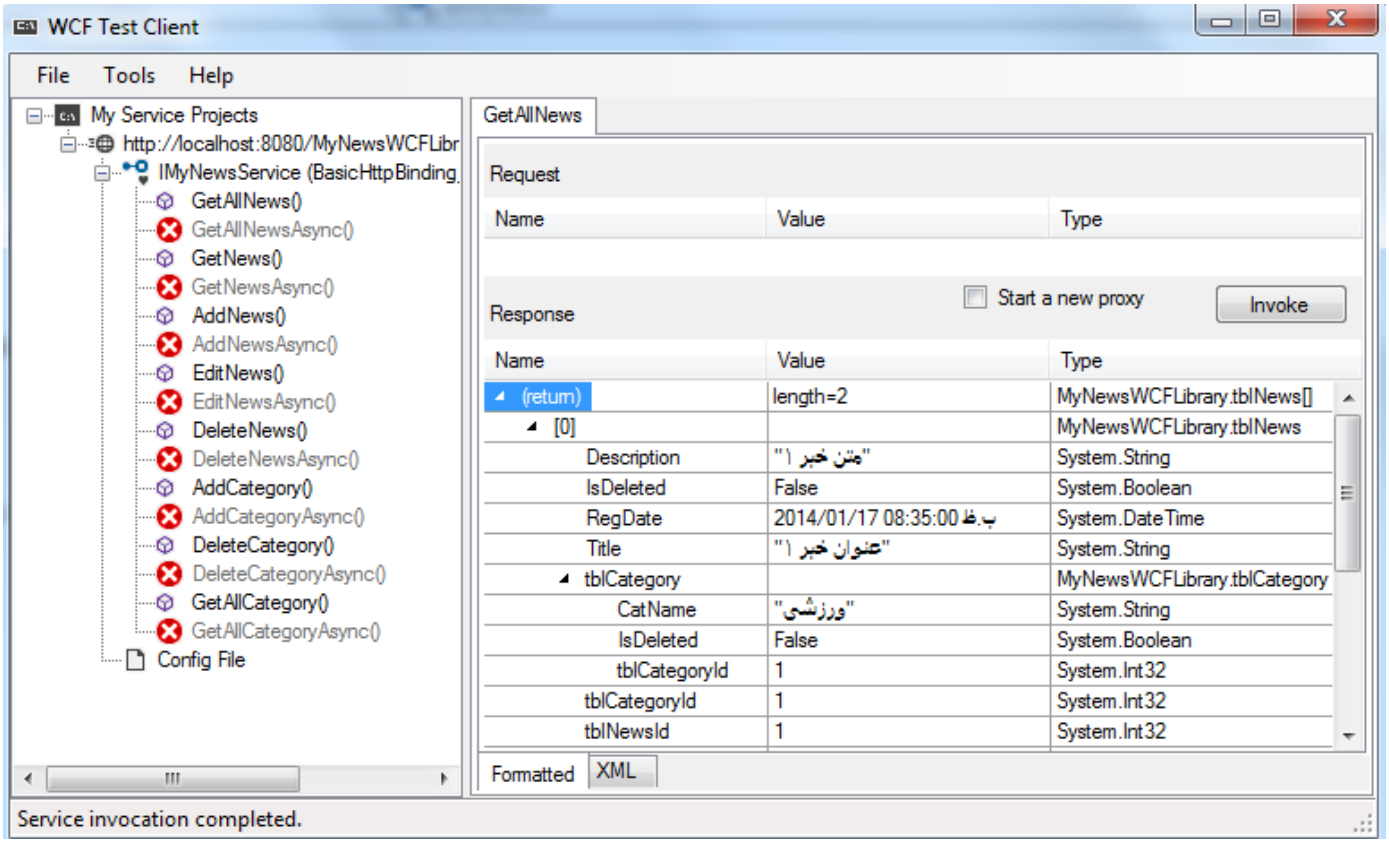
```
public string NavigationProperty(NavigationProperty navigationProperty)
    {
        var endType = _typeMapper.GetTypeName(navigationProperty.ToEndMember.GetEntityType());
        return string.Format(
            CultureInfo.InvariantCulture,
            "{0}{1} {2} {3} {{ {4}get; {5}set; }}",
navigationProperty.ToEndMember.RelationshipMultiplicity != RelationshipMultiplicity.Many ?
"[DataMember]" + Environment.NewLine : "",
            AccessibilityAndVirtual(Accessibility.ForProperty(navigationProperty)),
            navigationProperty.ToEndMember.RelationshipMultiplicity == RelationshipMultiplicity.Many ?
("ICollection<" + endType + ">") : endType,
            _code.Escape(navigationProperty),
            _code.SpaceAfter(Accessibility.ForGetter(navigationProperty)),
            _code.SpaceAfter(Accessibility.ForSetter(navigationProperty)));
    }
```

پس از ذخیره‌ی فایل، خواهید دید که صفت DataMember در کلاس tblNews پیش از ویژگی tblCategory افزوده شده است. بار دیگر پروژه را اجرا کنید. روی متد GetAllNews کلیک کنید و روی دکمه Invoke بفشارید. خواهید دید که هرچند tblCategory در ویژگی‌های آن قرار گرفته است ولی مقدار آن Null است. برای حل این مشکل باید از Solution Explorer فایل MyNewsService.cs را باز کنید و به به جای کد مربوط به متدهای GetAllNews و GetNews کدهای زیر را قرار دهید:

```
public List<tblNews> GetAllNews()
        {
            return dbMyNews.tblNews.Include(p=>p.tblCategory).Where(c=>c.IsDeleted == false).ToList();
```

```
}

        public tblNews GetNews(int tblNewsId)
        {
            return dbMyNews.tblNews.Include(p => p.tblCategory).FirstOrDefault(p => p.tblNewsId ==
tblNewsId);
        }
```

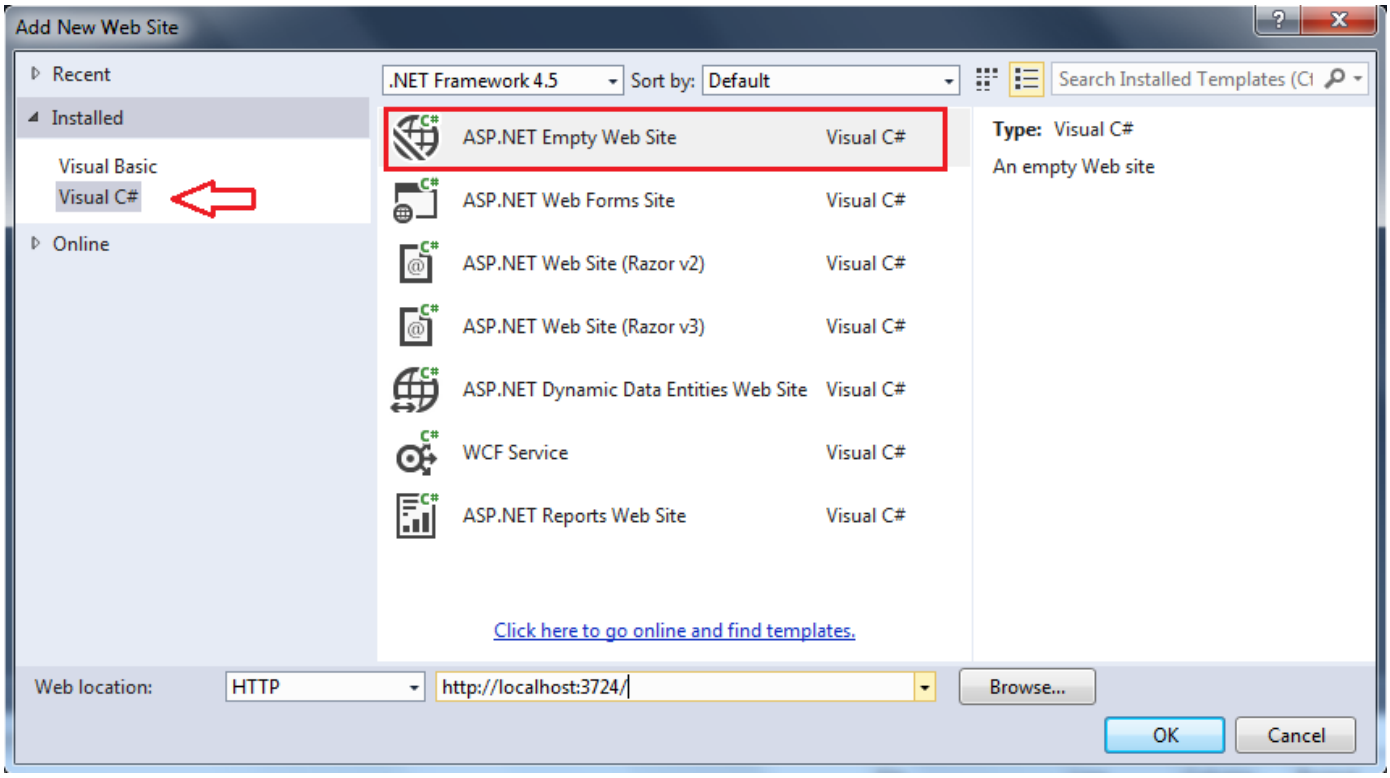
این بار اگر پروژه را اجرا کنید با نتیجه‌ای مانند شکل زیر روبه‌رو خواهید شد:



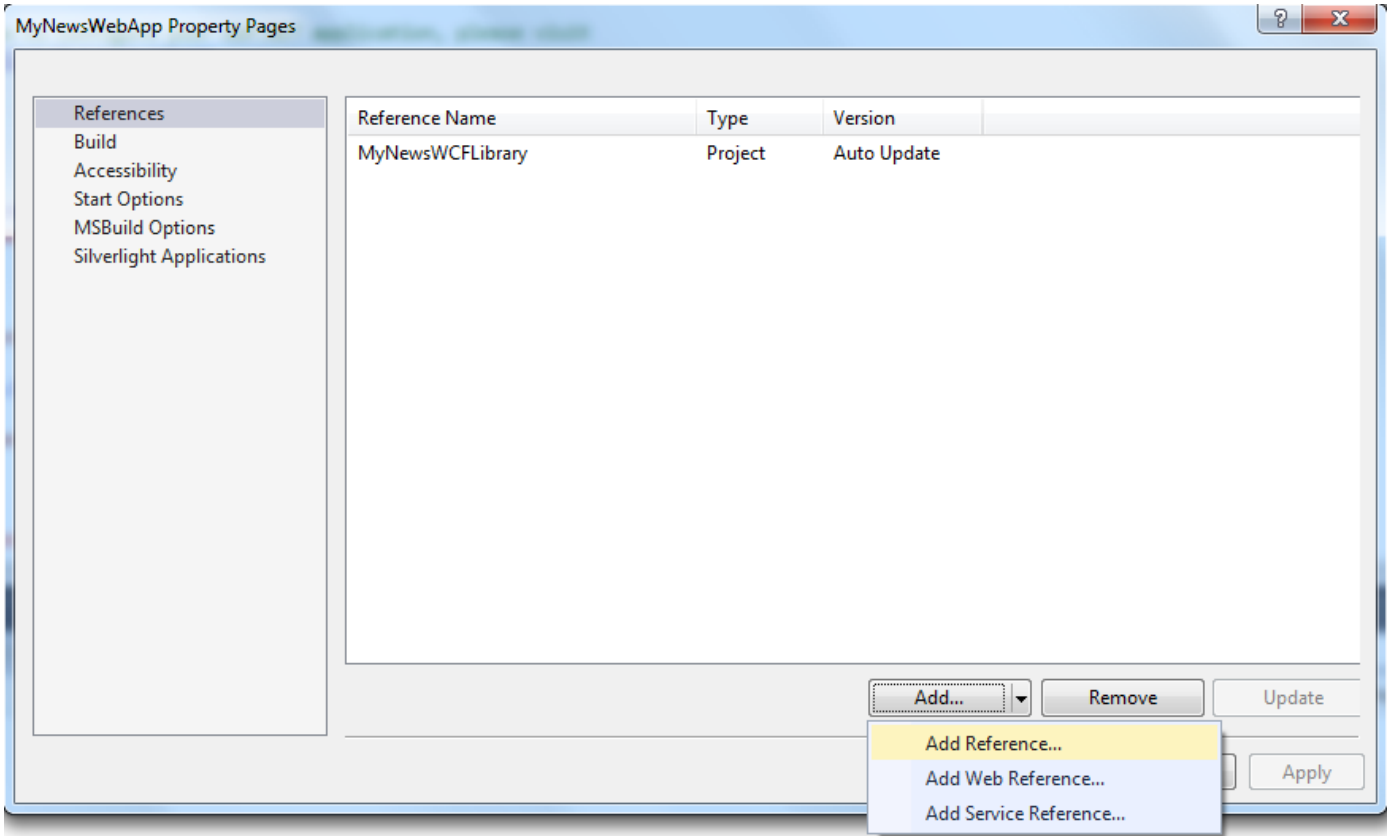
در بخش هفتم پیرامون میزبانی WCF Library خواهم نوشت.

عنوان: **ایجاد سرویس چندلایه‌ی WCF با Entity Framework در قالب پروژه - 7**
نویسنده: حامد قنادی
تاریخ: ۱۳۹۲/۱۰/۳۰ ۵:۱۰
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, WCF, Entity Framework 6

خروجی پروژه‌ی WCF Service Library یک فایل DLL است که هنگامی که با کنسول WCF Test Client اجرا می‌شود در آدرسی که در Web.Config تنظیم کرده بودیم اجرا می‌شود. اگر یک پروژه‌ی ویندوزی در همین راه حل بسازیم؛ خواهیم توانست از این آدرس برای دسترسی به WCF بهره ببریم. ولی اگر بخواهیم در IIS سرور قرار دهیم؛ باید در وب‌سایت آن‌را میزبانی کنیم. برای این‌کار از Solution Explorer روی راه حل MyNews راست‌کلیک کنید و از منوی باز شده روی New Web Site -> Add کلیک کنید. سپس مراحل زیر را برابر با شکل‌های زیر انجام دهید:



سپس روی Web Site ایجادشده راست کلیک کنید و از منوی بازشده Property Pages را انتخاب کنید. روی گزینه‌ی Add Reference کلیک کنید، سپس پروژه‌ی MyNewsWCFLibrary را از قسمت Solution انتخاب کرده و دکمه‌ی OK را بفشارید.



دکمه‌ی OK را بفشارید و از Solution Explorer فایل Web.Config را باز کنید. پیش از تغییرات مد نظر باید چنین محتوایی داشته باشد:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
  For more information on how to configure your ASP.NET application, please visit
  http://go.microsoft.com/fwlink/?LinkId=169433
  -->
<configuration>
  <system.web>
    <compilation debug="true" targetFramework="4.5" />
    <httpRuntime targetFramework="4.5" />
  </system.web>
</configuration>
```

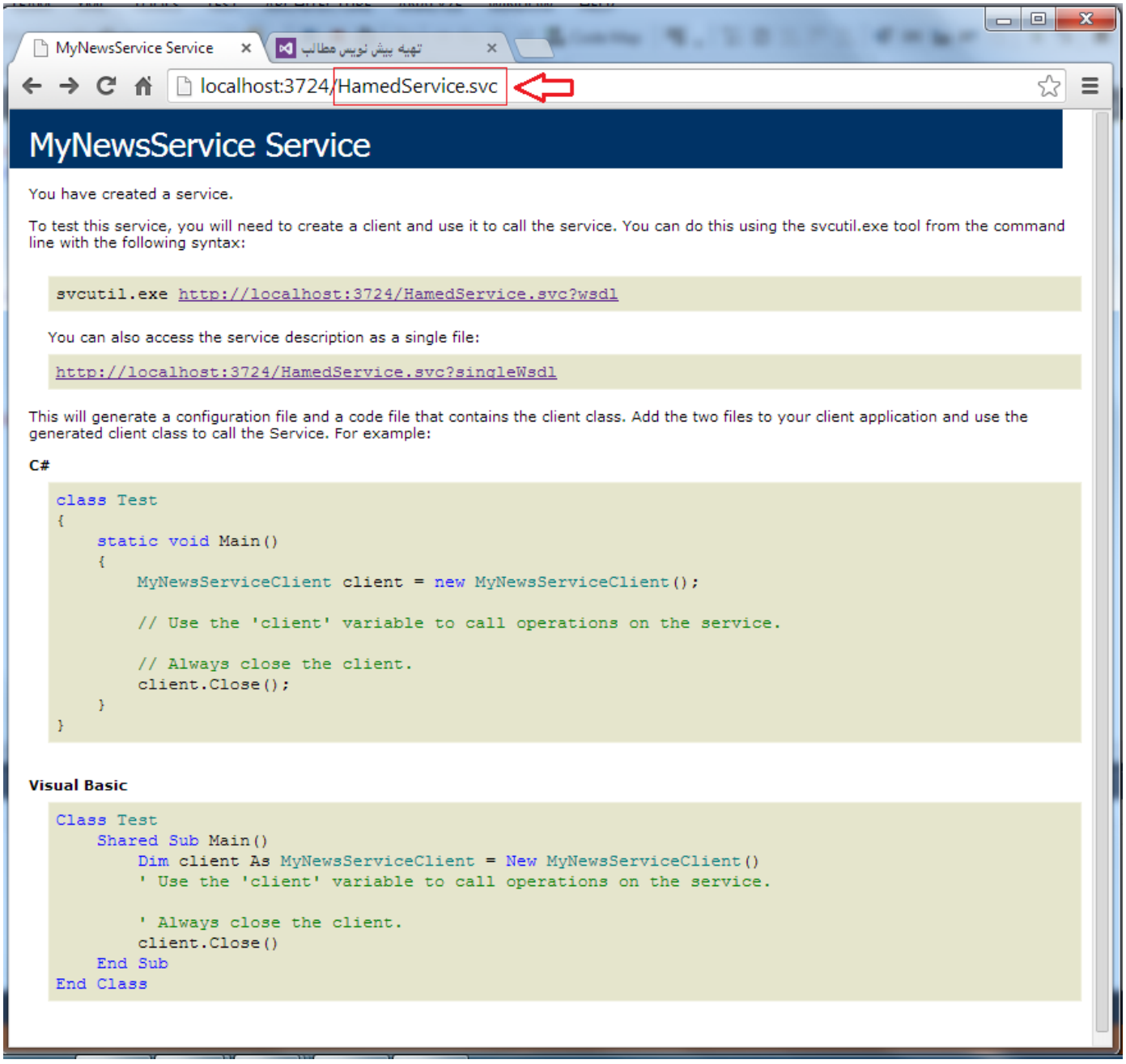
متن آن‌را به این صورت تغییر دهید:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
  For more information on how to configure your ASP.NET application, please visit
  http://go.microsoft.com/fwlink/?LinkId=169433
  -->
<configuration>
  <system.web>
    <compilation debug="true" targetFramework="4.5" />
    <httpRuntime targetFramework="4.5" />
  </system.web>
  <system.serviceModel>
    <serviceHostingEnvironment>
      <serviceActivations>
        <add factory="System.ServiceModel.Activation.ServiceHostFactory"
relativeAddress="./HamedService.svc" service="MyNewsWCFLibrary.MyNewsService"/>
      </serviceActivations>
    </serviceHostingEnvironment>
    <behaviors>
      <serviceBehaviors>
        <behavior>
```

```
        <serviceMetadata httpGetEnabled="true"/>
      </behavior>
    </serviceBehaviors>
  </behaviors>
 </system.serviceModel>
</configuration>
```

همان‌گونه که مشاهده می‌کنید به وسیله‌ی تگ add factory سرویس‌ها را به وب‌سایت معرفی می‌کنیم. با relativeAddress می‌توانیم هر نامی را به عنوان نام سرویس که در URL قرار می‌گیرد معرفی کنیم. چنان‌که من به جای MyNewsService از نام HamedService استفاده کردم. و در صفت service فضای نام و نام کلاس سرویس را معرفی می‌کنیم.

اکنون پروژه را اجرا کنید. در مرورگر باید صفحه را به این‌صورت مشاهده کنید:



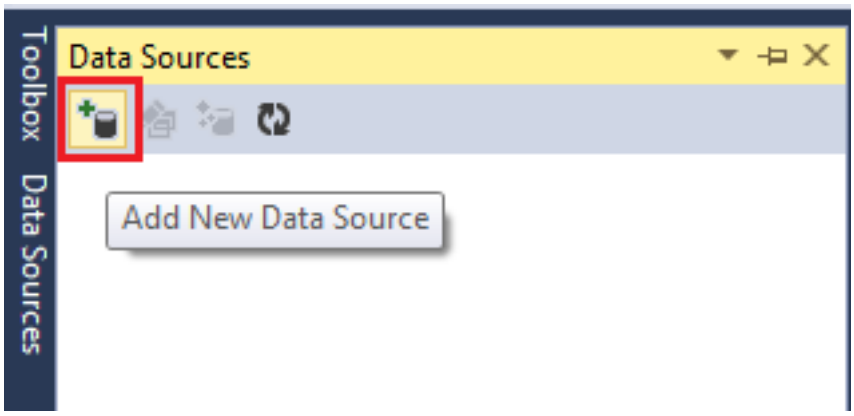
نیازی به یادآوری نیست که شما می‌توانید این پروژه را در IIS سرور راه‌اندازی کنید تا کلیه‌ی مشتری‌ها به آن دسترسی داشته باشند. هرچند پیش از آن باید امنیت را نیز در WCF برقرار کنید.

توجه داشته باشید که روشی که در این بخش به عنوان میزبانی WCF مطرح کردم یکی از روش‌های میزبانی WCF است. مثلاً شما می‌توانستید به جای ایجاد یک WCFLibrary و یک Web Site به صورت جداگانه یک پروژه از نوع WCF Service و یا Web Site ایجاد می‌کردید و سرویس‌ها و مدل Entity Framework را به طور مستقیم در آن می‌افزودید. روشی که در این درس از آن بهره برده ایم البته مزایایی دارد از جمله این‌که خروجی پروژه فقط یک فایل DLL است و با هر بار تغییر فقط کافی است همان فایل را در پوشه Bin از وب‌سایتی که روی سرور می‌گذارید کپی کنید.

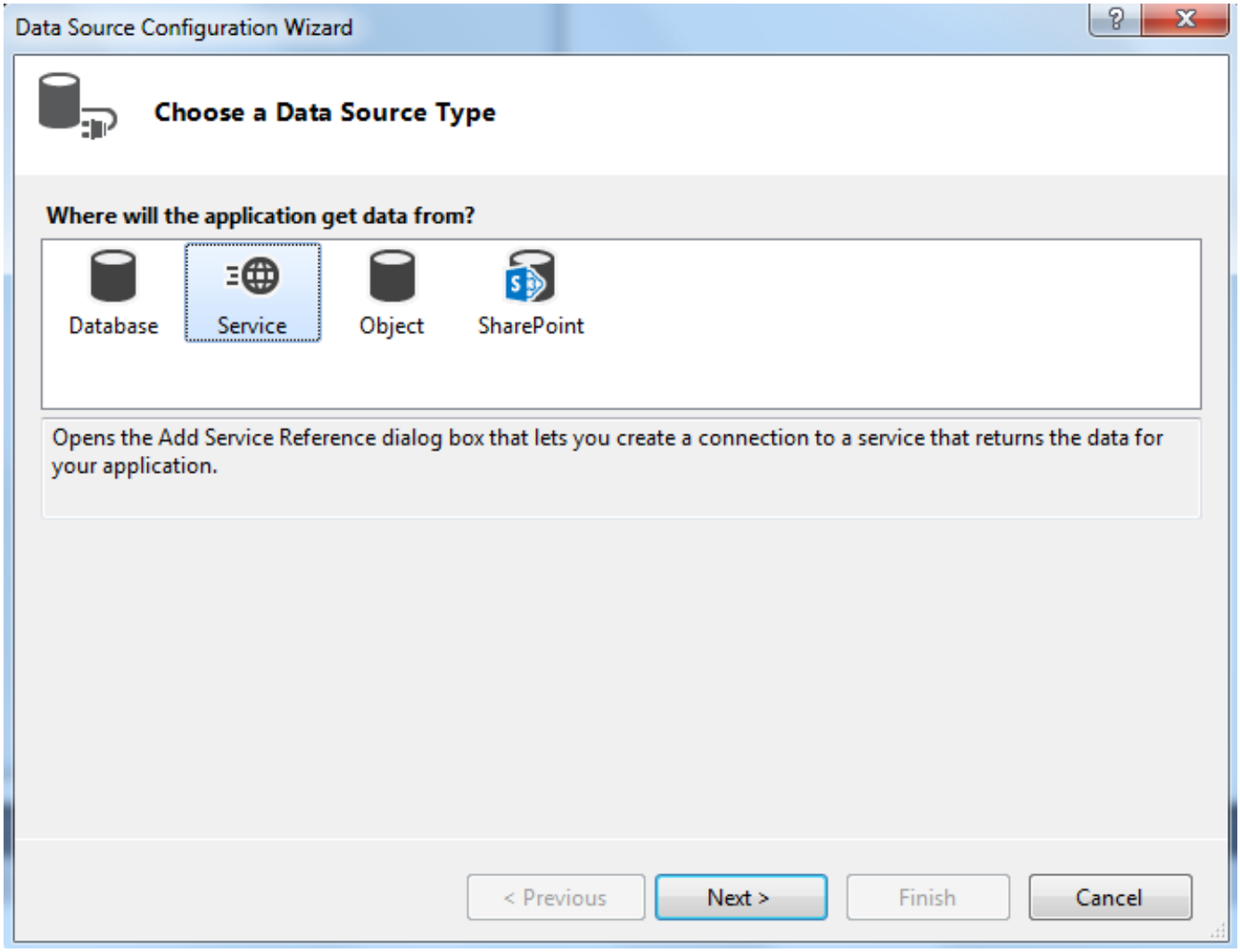
در بخش هشتم با هم یک پروژه‌ی تحت ویندوز خواهیم ساخت و از سرویس WCF ای که ساخته ایم در آن استفاده خواهیم کرد.

عنوان: **ایجاد سرویس چندلایه‌ی WCF با Entity Framework در قالب پروژه - 8**
نویسنده: حامد قنادی
تاریخ: ۱۳۹۲/۱۰/۳۰ ۲۲:۳۵
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, WCF, C#.NET, Entity Framework 6

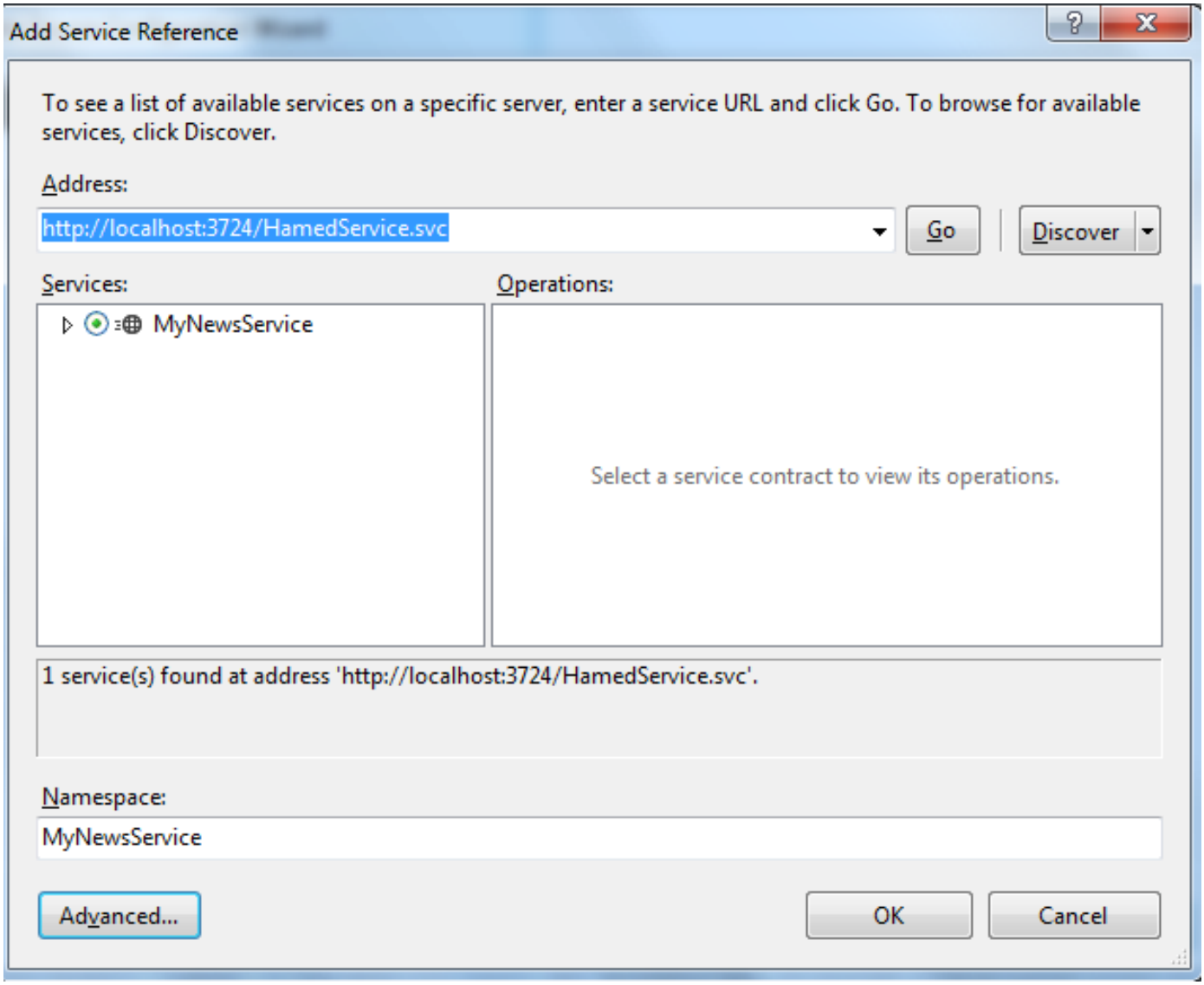
در Solution Explorer روی نام راه حل - MyNews - راست کلیک کنید و Add-> New Project را انتخاب کنید. سپس یک پروژه از نوع Windows Forms Application انتخاب کنید و نام آن‌را MyNewsWinApp بگذارید. یا کلیدهای ترکیبی Shift + Alt + D پنجره‌ی Data Sources را نمایان کنید. برابر با شکل روی ابزار Add New Data Source کلیک کنید:



از پنجره‌ی باز شده روی گزینه‌ی Service کلیک کنید:



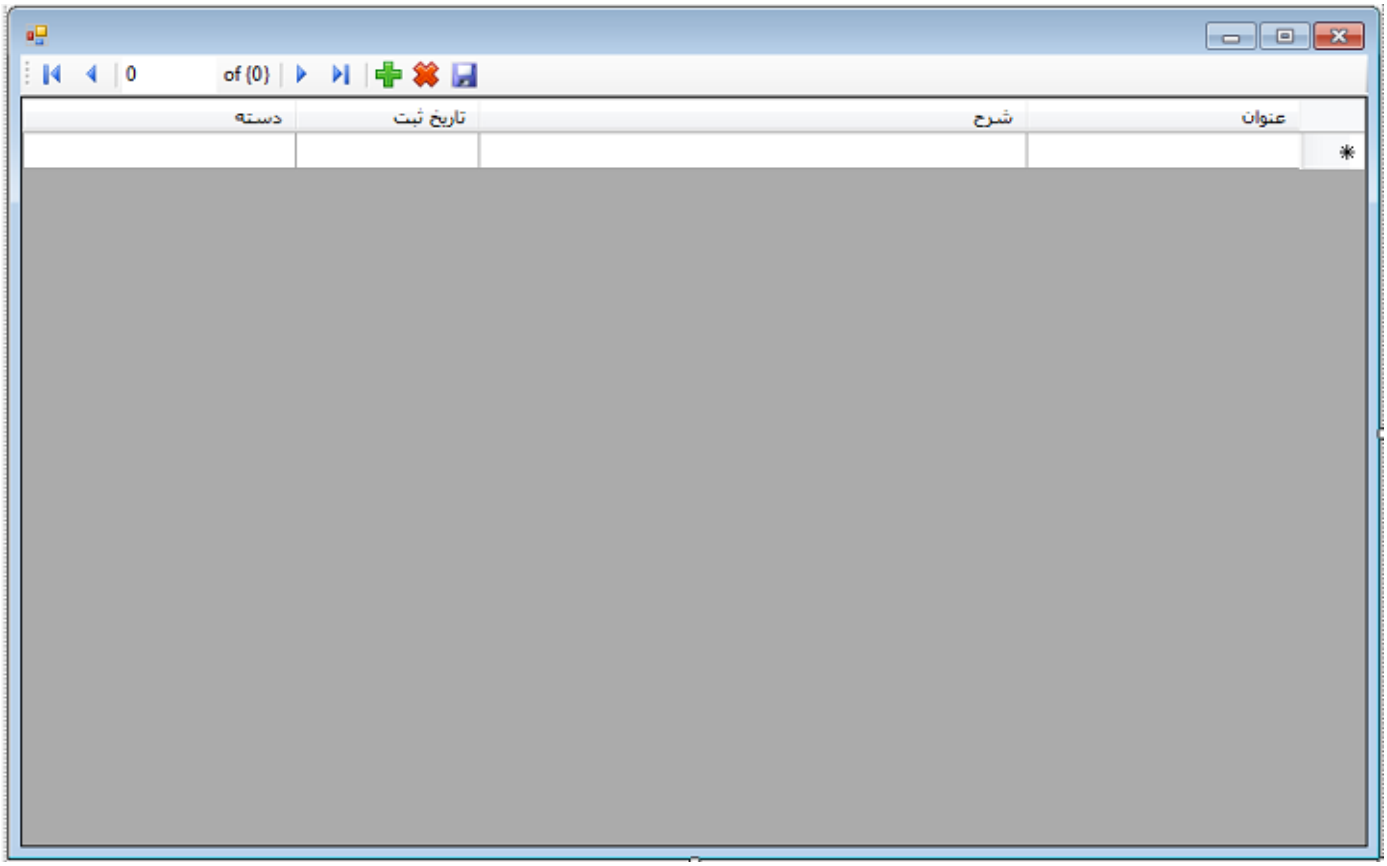
روی گزینه‌ی Next کلیک کنید و در پنجره‌ای که باز می‌شود در قسمت Address نشانی وب‌سایتی که در بخش پیشین تولید کردیم و ممکن است شما در IIS افزوده باشید؛ قرار دهید و روی دکمه‌ی GO بفشارید تا سرویس در کادر پایین افزوده شود. سپس در قسمت Namespace نامی مناسب برای فراخوانی سرویس وارد کنید آن‌گاه دکمه‌ی OK را بفشارید.



از پنجره‌ی بازشده روی دکمه‌ی Finish کلیک کنید. پس از مکثی کوتاه سرویس به همراه دو موجودیت آن درون Data Sources دیده خواهد شد. از آن‌طرف در Solution Explorer نیز در پوشه‌ی Service References سرویس تعریف‌شده ارجاع داده خواهد گرفت.

از Data Sources روی tblNews کلیک کنید سپس آن‌را کشیده و به روی فرم رها کنید. خواهید دید که یک DataGridView شامل همه‌ی ویژگی‌های موجودیت tblNews و یک Binding Navigator که با موجودیت tblNews در پیوند است و یک منبع داده به نام tblNewsBindingSource به صورت خودکار در فرم افزوده خواهد شد.

چیدمان فرم، رنگ‌ها، اندازه‌ها و فونت را آن‌گونه که می‌پسندید تنظیم کنید. سپس ستون‌هایی که به آن‌ها نیازی ندارید حذف یا پنهان کرده و عنوان ستون‌های مانده را ویرایش کنید. کلیدهای افزودن، حذف و ذخیره را روی Navigator ایجاد کنید و بقیه‌ی کلیدها را اگر به آن نیازی ندارید حذف کنید. البته می‌توانید بنا به سلیقه‌ی کاری‌تان یک Panel برای این‌کار اختصاص دهید. در این‌جا یک فرم ساده در نظر گرفته شده است:



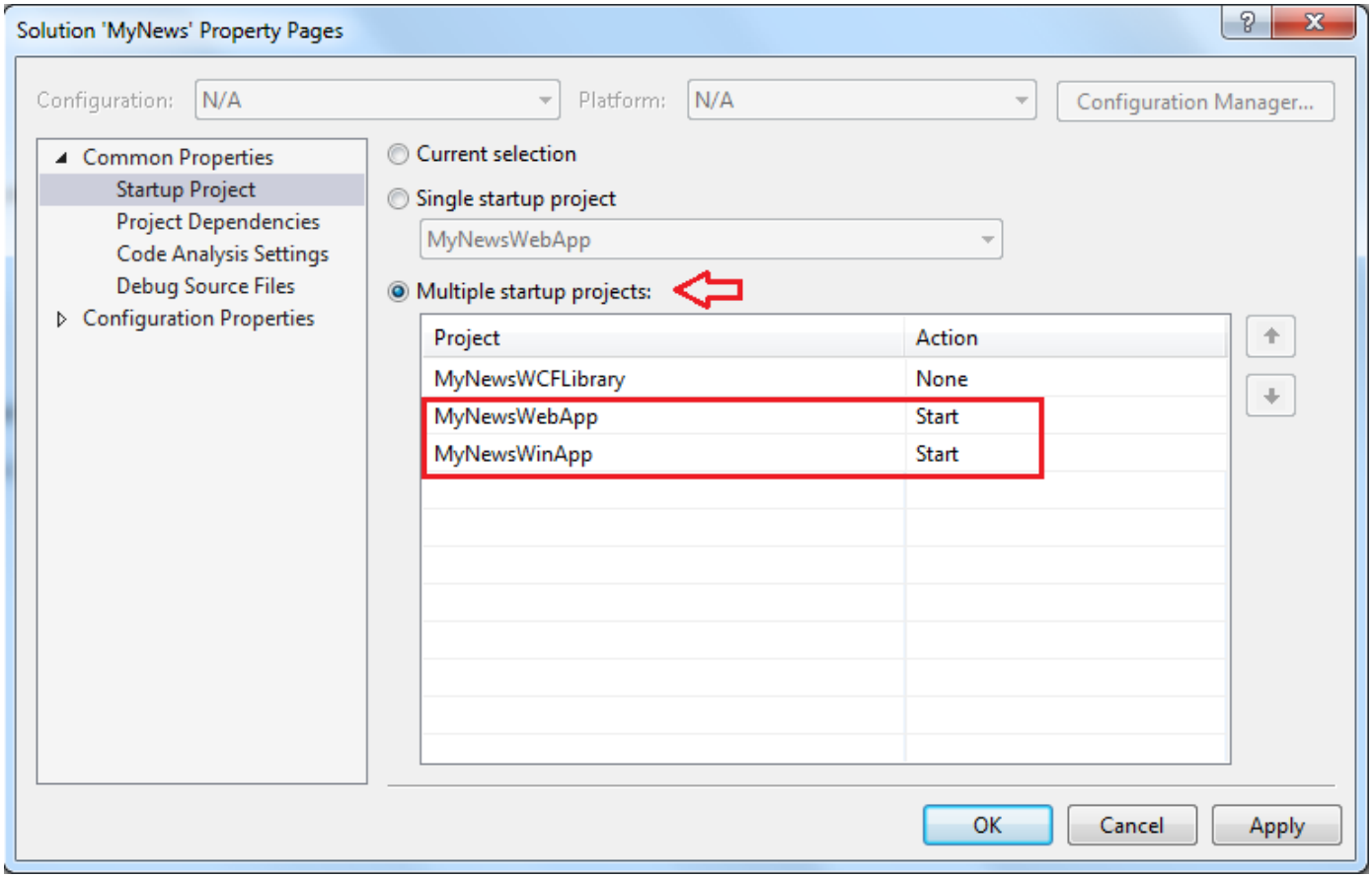
اکنون نوبت به کدنویسی است. سورس فرم را بازکنید و نخست سرویس را به این صورت در جای مناسب تعریف کنید:

```
MyNewsService.MyNewsServiceClient MyNews = new MyNewsService.MyNewsServiceClient();
```

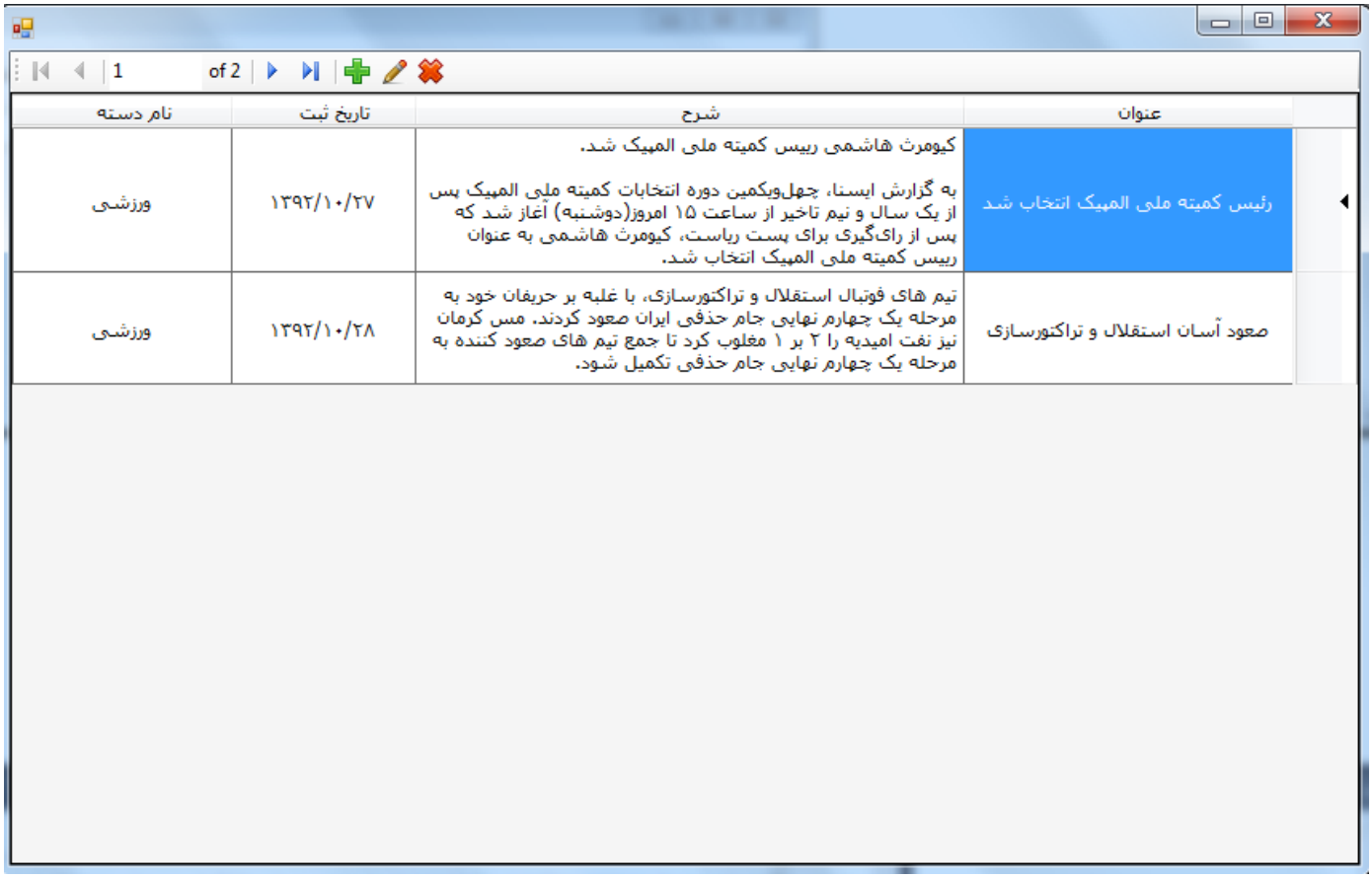
یک تابع کوچک برای تبدیل تاریخ میلادی به شمسی بنویسید سپس رویداد Load فرم را به این صورت بنویسید:

```
string MiladiToShamsi(DateTime MyDate)
        {
            System.Globalization.PersianCalendar pers = new System.Globalization.PersianCalendar();
            return string.Format("{0}/{1}/{2}", pers.GetYear(MyDate),
pers.GetMonth(MyDate).ToString("D2"), pers.GetDayOfMonth(MyDate).ToString("D2"));
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            tblNewsBindingSource.DataSource = MyNews.GetAllNews().Select(p => new {p.tblNewsId,
p.tblCategory.CatName, p.Title, p.Description, RegDate= MiladiToShamsi( p.RegDate) });
        }
```

پیش از اجرای پروژه از Solution Explorer روی نام راه حل راست‌کلیک کنید و گزینه‌ی Properties را انتخاب کنید. در پنجره‌ی بازشده تنظیمات زیر را انجام دهید:



این کار باعث می‌شود که به طور هم‌زمان پروژه‌ی وب‌سایت و ویندوز اجرا شود. اکنون پروژه را اجرا کنید. اگر با پیغام خطا روبه‌رو شدید؛ تگ Connection String را از App.Config پروژه WCF Library به Web.Config پروژه وب‌سایت کپی کنید. در این صورت پروژه به راحتی اجرا خواهد شد.



در بخش پسین پیرامون افزودن، ویرایش و حذف و برخی توضیحات برای توسعه‌ی کار خواهم نوشت.

عنوان: **استفاده از EF در اپلیکیشن های N-Tier : قسمت اول**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۱/۰۷ ۲:۳۵
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, Tips, ASP.NET Web API, N-Layer Architecture, Entity Framework 6

تمام اپلیکیشن‌ها را نمی‌توان در یک پروسس بسته بندی کرد، بدین معنا که تمام اپلیکیشن روی یک سرور فیزیکی قرار گیرد. در عصر حاظر معماری بسیاری از اپلیکیشن‌ها چند لایه است و هر لایه روی سرور مجزایی توزیع می‌شود. بعنوان مثال یک معماری کلاسیک شامل سه لایه نمایش (presentation)، اپلیکیشن (application) و داده (data) است. لایه بندی منطقی (logical layering) یک اپلیکیشن می‌تواند 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)، پروتوکل‌های شبکه و غیره باشیم. اما اپلیکیشن‌های بزرگی که چندین کلاینت دارند و در مراکز داده میزبانی می‌شوند باید تمام این مسائل را در نظر بگیرند. خوشبختانه پیاده سازی چنین اپلیکیشن هایی با استفاده از Entity Framework و دیگر تکنولوژی‌های مایکروسافت مانند WCF, Web API ساده‌تر شده است.
منظور از n-Tier معماری اپلیکیشن هایی است که لایه‌های نمایش، منطق تجاری و دسترسی داده هر کدام روی سرور مجزایی میزبانی می‌شوند. این تفکیک فیزیکی لایه‌ها به بسط پذیری، مدیریت و نگهداری اپلیکیشن‌ها در دراز مدت کمک می‌کند، اما معمولا تاثیری منفی روی کارایی کلی سیستم دارد. چرا که برای انجام عملیات مختلف باید از محدوده ماشین‌های فیریکی عبور کنیم.

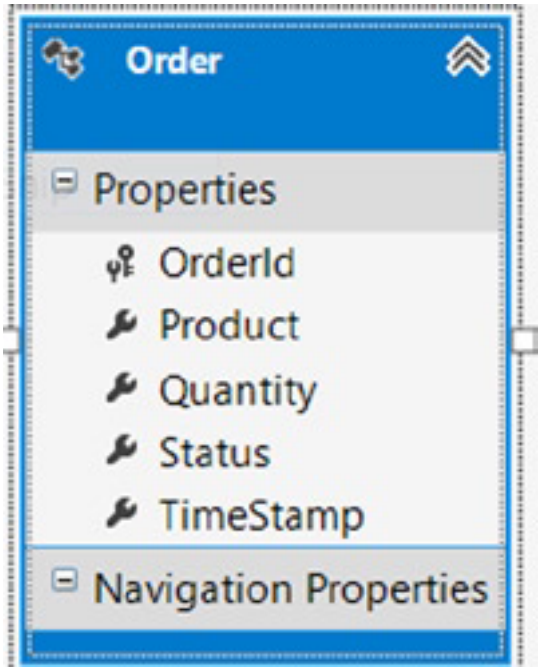
معماری N-Tier چالش‌های بخصوصی را برای قابلیت‌های change-tracking در EF اضافه می‌کند. در ابتدا داده‌ها توسط یک آبجکت EF Context بارگذاری می‌شوند اما این آبجکت پس از ارسال داده‌ها به کلاینت از بین می‌رود. تغییراتی که در سمت کلاینت روی داده‌ها اعمال می‌شوند ردیابی (track) نخواهند شد. هنگام بروز رسانی، آبجکت Context جدیدی برای پردازش اطلاعات ارسالی باید ایجاد شود. مسلما آبجکت جدید هیچ چیز درباره Context پیشین یا مقادیر اصلی موجودیت‌ها نمی‌داند.

در نسخه‌های قبلی Entity Framework توسعه دهندگان با استفاده از قالب ویژه ای بنام Self-Tracking Entities می‌توانستند تغییرات موجودیت‌ها را ردیابی کنند. این قابلیت در نسخه EF 6 از رده خارج شده است و گرچه هنوز توسط ObjectContext پشتیبانی می‌شود، آبجکت DbContext از آن پشتیبانی نمی‌کند.

در این سری از مقالات روی عملیات پایه CRUD تمرکز می‌کنیم که در اکثر اپلیکیشن‌های n-Tier استفاده می‌شوند. همچنین خواهیم دید چگونه می‌توان تغییرات موجودیت‌ها را ردیابی کرد. مباحثی مانند همزمانی (concurrency) و مرتب سازی (serialization) نیز بررسی خواهند شد. در قسمت یک این سری مقالات، به بروز رسانی موجودیت‌های منفصل (disconnected) توسط سرویس‌های Web API نگاهی خواهیم داشت.

بروز رسانی موجودیت‌های منفصل با Web API
سناریویی را فرض کنید که در آن برای انجام عملیات CRUD از یک سرویس Web API استفاده می‌شود. همچنین مدیریت داده‌ها با مدل Code-First پیاده سازی شده است. در مثال جاری یک کلاینت Console Application خواهیم داشت که یک سرویس Web API را فراخوانی می‌کند. توجه داشته باشید که هر اپلیکیشن در Solution مجزایی قرار دارد. تفکیک پروژه‌ها برای شبیه سازی یک محیط n-Tier انجام شده است.

فرض کنید مدلی مانند تصویر زیر داریم.



همانطور که می‌بینید مدل جاری، سفارشات یک اپلیکیشن فرضی را معرفی می‌کند. می‌خواهیم مدل و کد دسترسی به داده‌ها را در یک سرویس Web API پیاده سازی کنیم، تا هر کلاینتی که از HTTP استفاده می‌کند 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 مدل‌ها 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 Compatibilty
    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 resouce 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 resouce 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 ورک‌های ASP.NET می‌توانند کنترلرهایی کاملا اجرایی برایتان تولید کنند که صرفه جویی چشمگیری در زمان و کار شما خواهد بود.

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

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 لایه‌های مختلف مورد استفاده قرار گیرند - مانند کلاس‌های موجودیت‌ها - بهتر است در لایه مجزایی قرار داده شده و به اشتراک گذاشته شوند. مثلا می‌توانید پروژه ای از نوع Class Library بسازید و تمام موجودیت‌ها را در آن تعریف کنید. سپس لایه‌های 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 cal
    _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 quanity
        _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 نهایتا منجر به حذف شدن رکورد می‌شود.

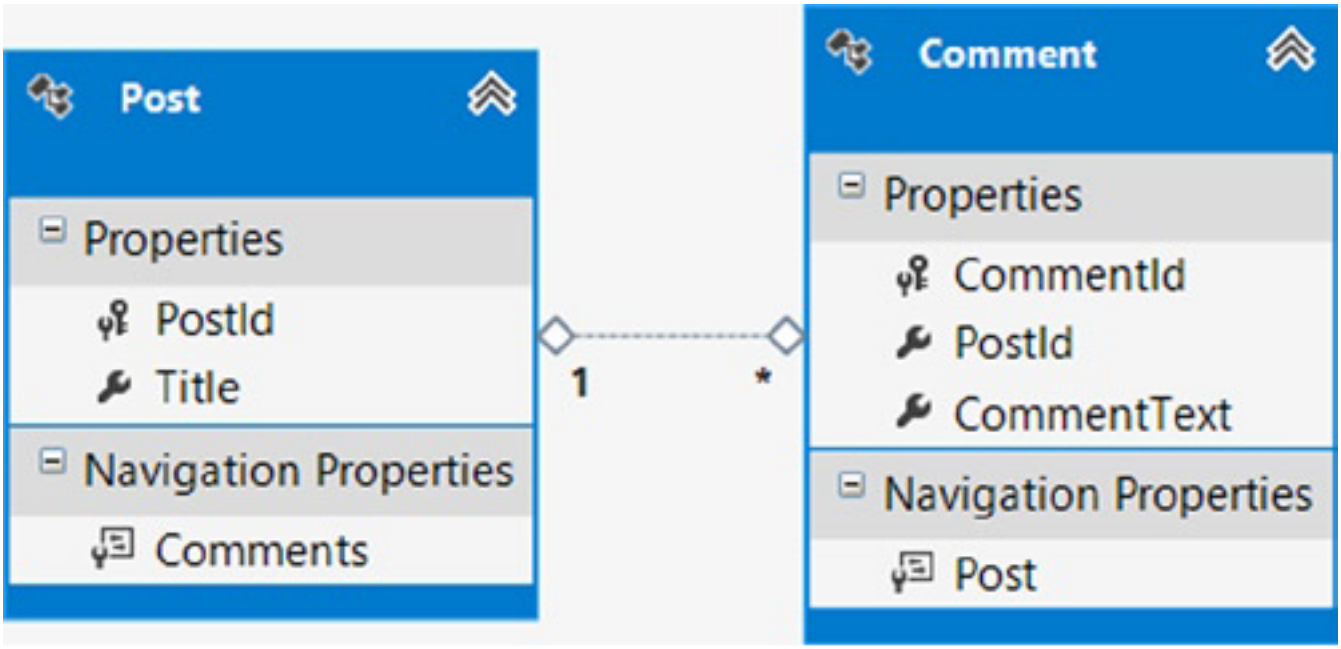
در یک اپلیکیشن واقعی بهتر است کد دسترسی داده‌ها از سرویس Web API تفکیک شود و در لایه مجزایی قرار گیرد.

عنوان: **استفاده از EF در اپلیکیشن های N-Tier : قسمت دوم**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۱/۰۷ ۷:۴۵
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, Tips, WCF, N-Layer Architecture, Entity Framework 6

در قسمت قبل معماری اپلیکیشن‌های N-Tier و بروز رسانی موجودیت‌های منفصل توسط Web API را بررسی کردیم. در این قسمت بروز رسانی موجودیت‌های منفصل توسط WCF را بررسی می‌کنیم.

بروز رسانی موجودیت‌های منفصل توسط WCF

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



همانطور که می‌بینید مدل ما متشکل از پست‌ها و نظرات کاربران است. برای ساده نگاه داشتن مثال جاری، اکثر فیلدها حذف شده اند. مثلا متن پست ها، نویسنده، تاریخ و زمان انتشار و غیره. می‌خواهیم تمام کد دسترسی داده‌ها را در یک سرویس WCF پیاده سازی کنیم تا کلاینت‌ها 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 نام‌های Post, Comment و Recipe2Context به پروژه اضافه کنید. کلاس‌های Post و Comment موجودیت‌های مدل ما هستند که به جداول متناظرشان نگاشت می‌شوند. کلاس Recipe2Context آبجکت DbContext ما خواهد بود که بعنوان درگاه عملیاتی EF عمل می‌کند. دقت کنید که خاصیت‌های لازم WCF یعنی DataContract و DataMember در کلاس‌های موجودیت‌ها بدرستی استفاده می‌شوند. لیست زیر کد این کلاس‌ها را نشان می‌دهد.

```
[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 : IService
{
    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. مثلا رکوردی 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 اصلا کاری نخواهیم داشت. بدین منظور تنها خاصیت CommentText را بعنوان Modified 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، معمولا 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 آبجکت‌های اصلی را بازگرداند. اگر سعی کنید آبجکت پروکسی را سریال کنید با پیغام خطای زیر مواجه خواهید شد:

```
The underlying connection was closed: The connection was closed unexpectedly
```

می توانیم 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 آبجکت‌های 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 موجودیتها یا جدید هستند و یا ویرایش شده اند و بنابراین حتی اگر یک پست کامنتهایی داشته باشد که ویرایش نشده اند، برای انها 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؟

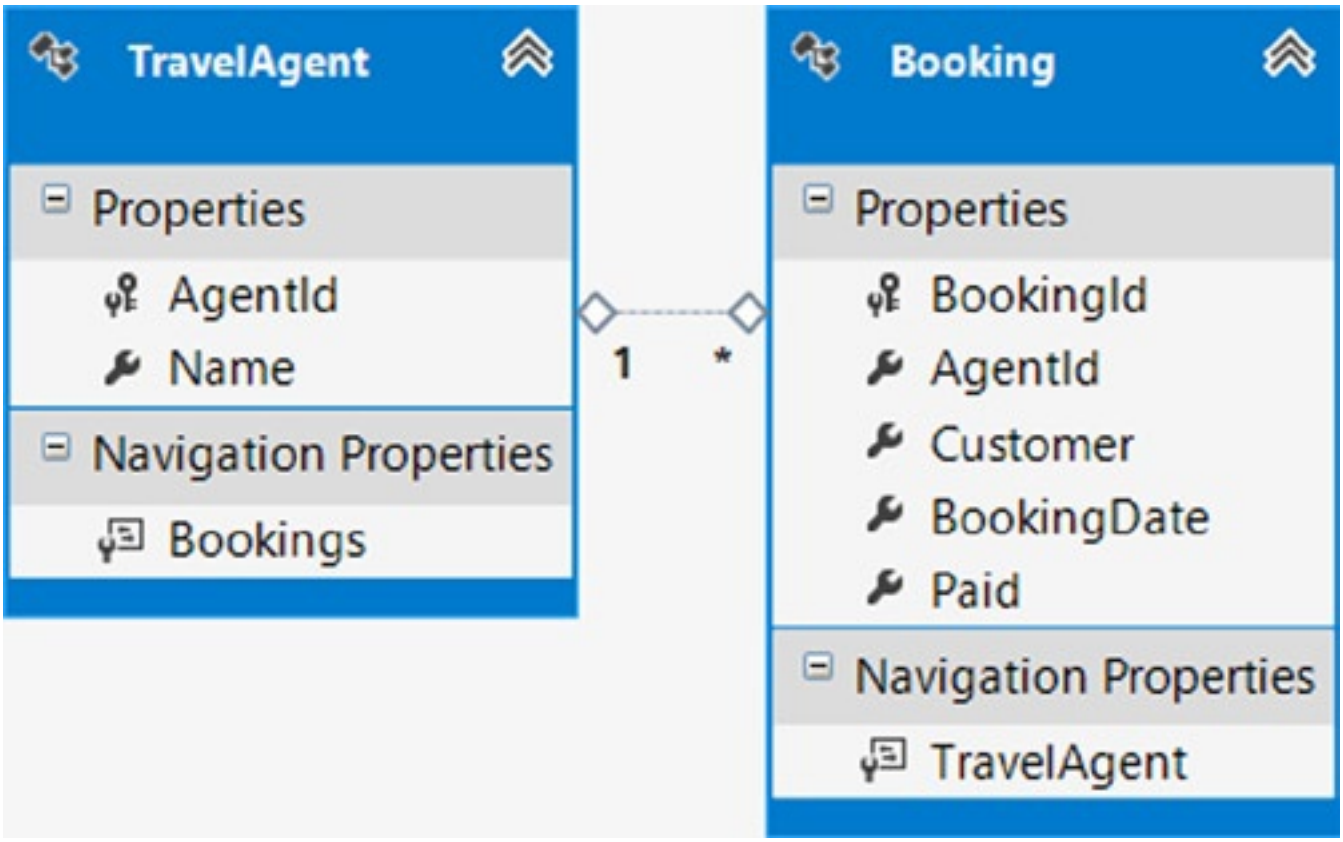
عنوان: **استفاده از EF در اپلیکیشن های N-Tier : قسمت سوم**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۱/۰۸ ۱۰:۵
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, Tips, ASP.NET Web API, N-Layer Architecture, Web API, Entity Framework 6

در قسمت قبلی بروز رسانی موجودیت‌های منفصل با WCF را بررسی کردیم. در این قسمت خواهیم دید چگونه می‌توان تغییرات موجودیت‌ها را تشخیص داد و عملیات CRUD را روی یک Object Graph اجرا کرد.

تشخیص تغییرات با Web API

فرض کنید می‌خواهیم از سرویس‌های Web API برای انجام عملیات CRUD استفاده کنیم، اما بدون آنکه برای هر موجودیت متدهایی مجزا تعریف کنیم. به بیان دیگر می‌خواهیم عملیات مذکور را روی یک Object Graph انجام دهیم. مدیریت داده‌ها هم با مدل Code-First پیاده سازی می‌شود. در مثال جاری یک اپلیکیشن کنسول خواهیم داشت که بعنوان یک کلاینت سرویس را فراخوانی می‌کند. 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 می‌بینید مدل ما آژانس‌های مسافرتی و رزرواسیون آنها را ارائه می‌کند. می‌خواهیم مدل و کد دسترسی داده‌ها 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 نام‌های 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 می‌کند. همچنین به JSON serializer می‌گوییم که self-referencing loop خاصیت‌های پیمایشی را نادیده بگیرد. این حلقه بدلیل ارتباط bidirectional بین موجودیت‌ها بوجود می‌آید.

```
protected void Application_Start()
{
    // Disable Entity Framework Model Compatibilty
    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 کلاس‌های TravelAgent و Booking را به پروژه کلاینت اضافه کنید. اینگونه کدها بهتر است در لایه مجزایی قرار گیرند و بین پروژه‌ها 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 می‌کند. در این مرحله سایت در حال اجرا است و سرویس‌ها 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 می‌کنیم و با تنظیم مقدار خاصیت Accept Header از سرویس درخواست می‌کنیم که اطلاعات را با فرمت JSON بازگرداند.

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

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

نکته: قابل ذکر است که اگر موجودیت‌های متعددی با مقداری یکسان در خاصیت کلید اصلی (Primary-key value) دارید باید مجموعه آبجکت‌های 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 می‌کند چرا که چند موجودیت با مقادیر کلید اصلی یکسان به context جاری اضافه کرده اید.

بعد از آن بر اساس مقدار خاصیت Id مشخص می‌کنیم که موجودیت‌ها 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 می‌دهیم. 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 می‌کنیم.

سپس تمام آبجکت‌های فرزند آژانس مسافرتی (رزرواسیون ها) را بررسی میکنیم و همین منطق را روی آنها اعمال می‌کنیم. یعنی در صورتی که مقدار فیلد Id آنها بزرگتر از 0 باشد وضعیت EntityState را به Modified تغییر می‌دهیم. در نهایت متد SaveChanges را فراخوانی می‌کنیم. در این مرحله برای موجودیت‌های جدید اسکریپت‌های Insert و برای موجودیت‌های تغییر کرده اسکریپت‌های Update تولید می‌شود. سپس کد وضعیت مناسب را به کلاینت بر می‌گردانیم. برای موجودیت‌های اضافه شده کد وضعیت 201 (Created) و برای موجودیت‌های بروز رسانی شده کد وضعیت 200 (OK) باز می‌گردد. کد 201 به کلاینت اطلاع می‌دهد که رکورد جدید با موفقیت ثبت شده است، و کد 200 از بروز رسانی موفقیت آمیز خبر می‌دهد. هنگام تولید سرویس‌های REST-based بهتر است همیشه کد وضعیت مناسبی تولید کنید.

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

در آخر کلاینت ما متد Retreive را روی سرویس فراخوانی می‌کند. این فراخوانی با کمک متد GetAsync انجام می‌شود که روی آبجکت HttpClient تعریف شده است. فراخوانی این متد تمام آژانس‌های مسافرتی بهمراه رزرواسیون‌های متناظرشان را دریافت می‌کند. در اینجا با استفاده از متد Include تمام رکوردهای فرزند را بهمراه تمام خاصیت هایشان (properties) بارگذاری می‌کنیم.

دقت کنید که مرتب کننده JSON تمام خواص عمومی (public properties) را باز می‌گرداند، 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 درخواست‌های HTTP به اکشن متدها استفاده نکرده ایم. مثلا بصورت پیش فرض درخواست‌های POST به متدهایی نگاشت می‌شوند که نام آنها با "Post" شروع می‌شود. در مثال جاری قواعد مسیریابی را تغییر داده ایم و رویکرد مسیریابی RPC-based را در پیش گرفته ایم. در اپلیکیشن‌های واقعی بهتر است از قواعد پیش فرض استفاده کنید چرا که هدف Web API ارائه سرویس‌های REST-based است. بنابراین بعنوان یک قاعده کلی بهتر است متدهای سرویس شما به درخواست‌های 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: ۱۳۹۲/۱۱/۱۱ ۱۱:۶

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 خواست‌های 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 می‌شود. ممنون

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۱۱/۱۱ ۱۱:۴۲

برای برقراری امنیت، تعیین هویت و اعتبارسنجی در وب API عموما یا از Forms authentication استفاده می‌شود و یا از ASP.NET Identity . زیر ساخت آن یکی است و مشترک.

عنوان: **استفاده از EF در اپلیکیشن های N-Tier : قسمت چهارم**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۱/۰۹ ۸:۴۰
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, Tips, ASP.NET Web API, N-Layer Architecture, Entity Framework 6, ASP.NET Web API 2

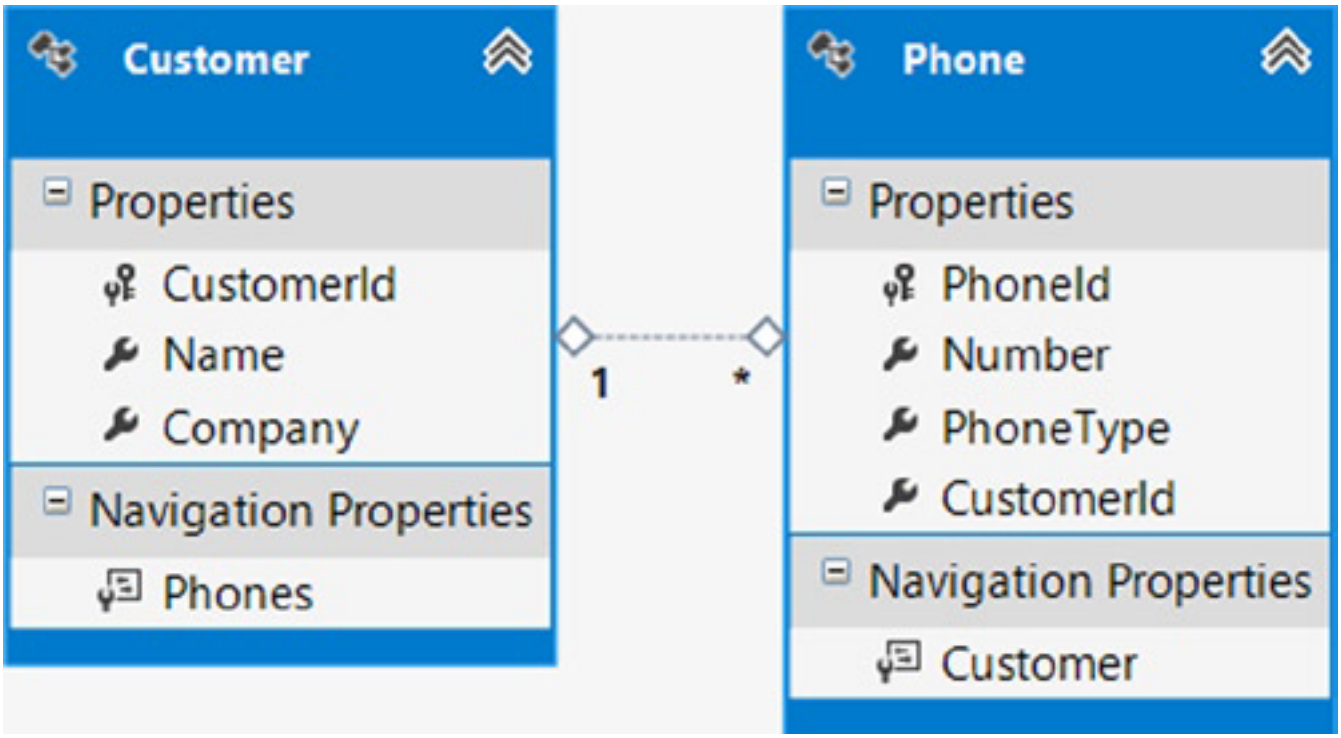
در قسمت قبل تشخیص تغییرات توسط Web API را بررسی کردیم. در این قسمت نگاهی به پیاده سازی Change-tracking در سمت کلاینت خواهیم داشت.

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 ویژوال 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 مستقیما 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 Compatibilty
    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 orginal 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 'Modifed' 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 کلاس‌های Customer, Phone و BaseEntity را به پروژه کلاینت اضافه کنید. چنین کدهایی بهتر است در لایه مجزایی قرار گیرند و بین لایه‌های 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 می‌کند. در این مرحله سایت در حال اجرا است و سرویس‌ها 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 می‌کنیم و از سرویس درخواست می‌کنیم 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 می‌کنیم. این فراخوانی تمام داده‌های پیشین را حذف می‌کند.

در قدم بعدی یک مشتری بهمراه دو شماره تماس می‌سازیم. 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 موجودیت هایی 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 نهایتا 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 قابلیت‌های async نیست، اما توصیه میشه در مواقعی که امکانش هست و مناسب است از این قابلیت استفاده کنید. لزوما کارایی (performance) بهتری بدست نمیارید ولی مسلما تجربه کاربری بهتری خواهید داشت. عملیاتی که بصورت async اجرا میشن ریسمان جاری (current thread) رو قفل نمی‌کنند، بنابراین اجرای اپلیکیشن ادامه پیدا می‌کنه و پاسخگویی بهتری بدست میارید. برای مطالعه بیشتر به این لینک مراجعه کنید.

مطالعه بیشتر

Using Asynchronous Methods in ASP.NET 4.5

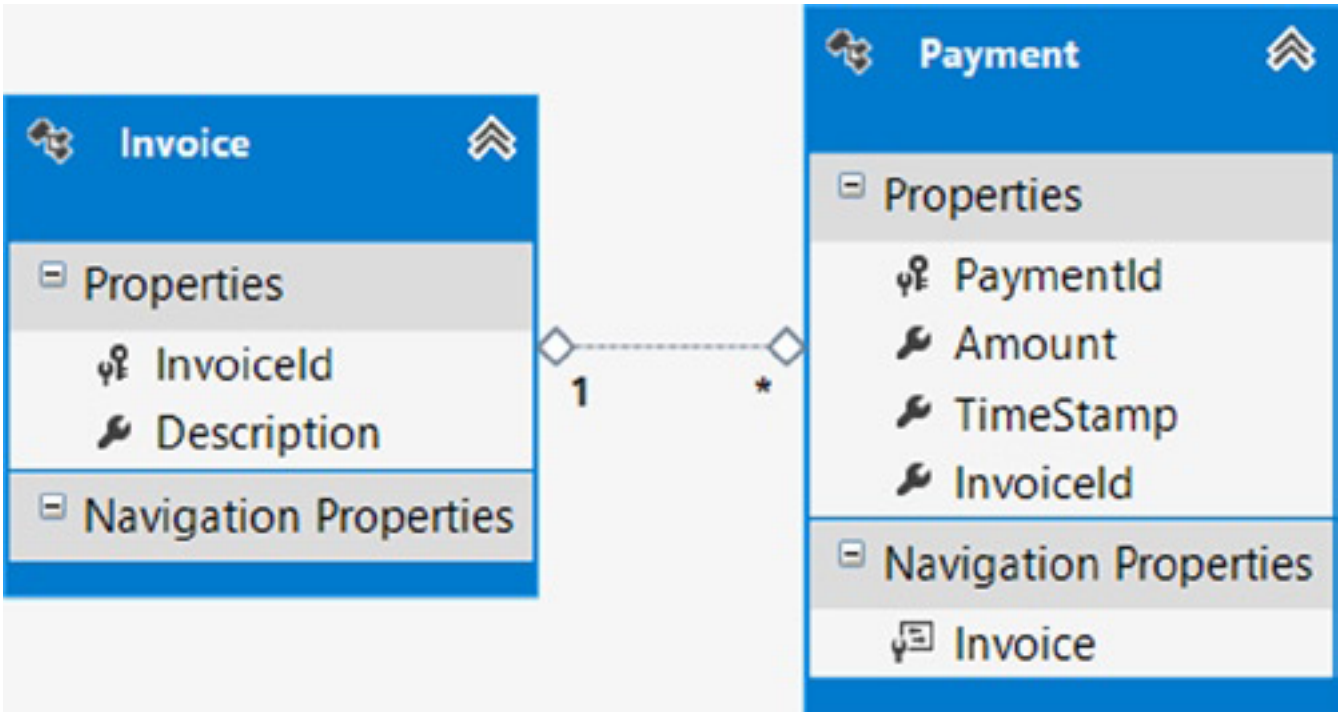
Async and Await

عنوان: **استفاده از EF در اپلیکیشن های N-Tier : قسمت پنجم**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۱/۱۰ ۲:۵۵
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, Tips, N-Layer Architecture, Entity Framework 6

در قسمت قبل پیاده سازی change-tracking در سمت کلاینت توسط Web API را بررسی کردیم. در این قسمت نگاهی به حذف موجودیت‌های منفصل یا disconnected خواهیم داشت.

حذف موجودیت‌های منفصل

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



همانطور که می‌بینید مدل ما صورت حساب‌ها و پرداخت‌های متناظر را ارائه می‌کند. در اپلیکیشن جاری یک سرویس WCF پیاده سازی کرده ایم که عملیات دیتابیسی کلاینت‌ها را مدیریت می‌کند. می‌خواهیم توسط این سرویس آبجکتی را (در اینجا یک موجودیت پرداخت) حذف کنیم. برای ساده نگاه داشتن مثال جاری، مدل‌ها را در خود سرویس تعریف می‌کنیم. 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 گزینه‌های Data -> ADO.NET Entity Data Model 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 می‌شود 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 نهایتا 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 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, TimeStamp و InvoiceId خواهد بود.

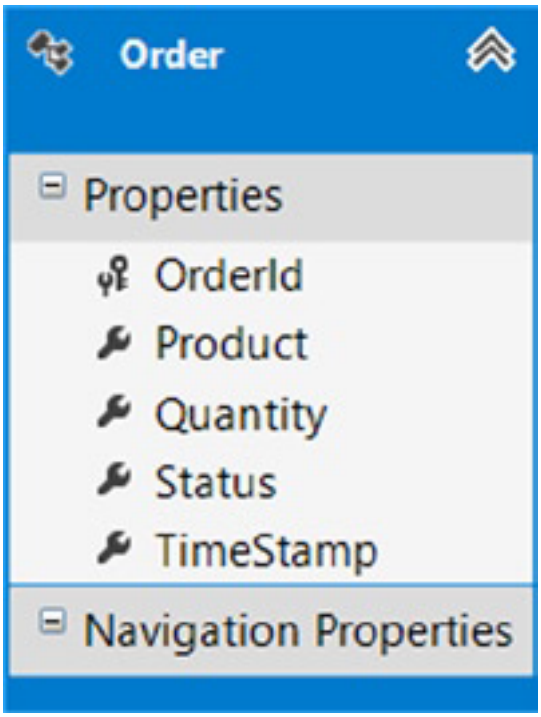
یک گزینه دیگر برای حذف موجودیت‌ها در Independent Associations این است که تمام موجودیت‌های مرتبط را مشخصا بارگذاری کنیم (eager loading) و کل Object graph را برای حذف به سرویس WCF یا Web API بفرستیم. در مثال جاری می‌توانستیم موجودیت صورتحساب مرتبط با موجودیت پرداخت را مشخصا بارگذاری کنیم. اگر می‌خواستیم 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.

عنوان: **استفاده از EF در اپلیکیشن های N-Tier : قسمت ششم**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۱/۱۱ ۰:۲۵
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, Tips, WCF, N-Layer Architecture, Web API, Entity Framework 6

در قسمت قبل رویکردهای مختلف برای حذف موجودیت‌های منفصل را بررسی کردیم. در این قسمت مدیریت همزمانی یا Concurrency را بررسی خواهیم کرد.

فرض کنید می‌خواهیم 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 می‌آید، هنگام بروز رسانی باید مقداری یکسان داشته باشد.

مدل زیر را در نظر بگیرید.



می‌خواهیم یک سفارش (order) را توسط یک سرویس WCF بروز رسانی کنیم در حالی که اطمینان حاصل می‌کنیم موجودیت سفارش از زمانی که دریافت شده تغییری نکرده است. برای مدیریت این وضعیت دو رویکرد تقریبا متفاوت را بررسی می‌کنیم. در هر دو رویکرد از یک ستون همزمانی استفاده می‌کنیم، 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 گزینه‌های 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 وهله 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 نسبتا 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. مثلا ممکن است از زمانی که موجودیت را از دیتابیس دریافت می‌کنید، تا زمانی که مقدار TimeStamp آن را مقایسه می‌کنید و نهایتا متد ()SaveChanges را صدا میزنید، موجودیت شما توسط کلاینت دیگری بروز رسانی شده باشد.

مسلما رویکرد دوم هزینه برتر از رویکرد اولی است، چرا که برای مقایسه مقادیر همزمانی موجودیت ها، یکبار موجودیت را از دیتابیس دریافت می‌کنید. اما این رویکرد در مواقعی که Object graphهای 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: ۱۳۹۲/۱۱/۱۶ ۱۸:۵۴

خیلی ممنون.
عالی بود ...

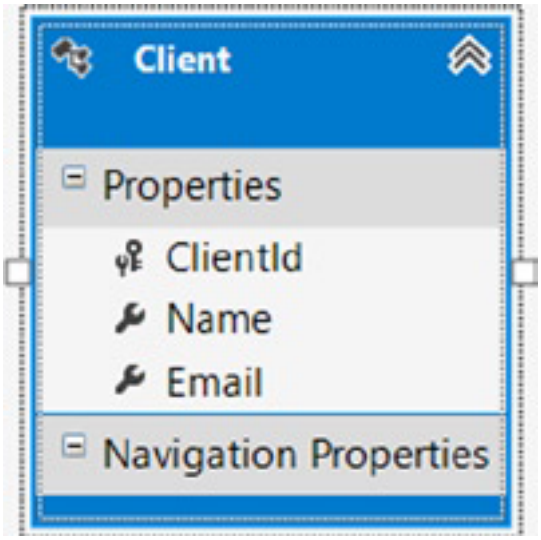
عنوان: **استفاده از EF در اپلیکیشن های N-Tier : قسمت هفتم**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۱/۱۱ ۱:۱۰
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, Tips, WCF, N-Layer Architecture, Entity Framework 6

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 کوئری 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 مجددا 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 نهایتا 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.Cofiguration.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.ExecuteSqlCommand("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 آبجکت‌های POCO با WCF را توصیه می‌کند چرا که مرتب سازی (serialization) آبجکت موجودیت را ساده‌تر می‌کند. اما در صورتی که از آبجکت‌های POCO ای استفاده می‌کنید که *changed-based notification* دارند (یعنی خواص موجودیت را با virtual علامت گذاری کرده اید و کلکسیون‌های خواص پیمایشی از نوع ICollection هستند)، آنگاه EF برای موجودیت‌های بازگشتی کوئری‌ها *پراکسی‌های دینامیک* تولید خواهد کرد.

استفاده از پراکسی‌های 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 موجودیت‌ها پراکسی می‌سازد، حالا باید آبجکت پراکسی را سریال کنیم، نه خود کلاس Client را. اینجا است که از DataContractResolver استفاده می‌کنیم. این کلاس می‌تواند حین سریال کردن آبجکت ها، نوعی را به نوع دیگر تبدیل کند. برای استفاده از این کلاس ما یک ویژگی سفارشی ساختیم، که پراکسی‌ها را به کلاس‌های POCO تبدیل می‌کند. سپس این ویژگی را به متد ()GetClient اضافه کردیم. این کار باعث می‌شود که پراکسی دینامیکی که توسط متد ()GetClient برای موجودیت Client تولید می‌شود، به درستی سریال شود.

مشکل دوم استفاده از پراکسی‌ها با WCF مربوط به بارگذاری تبل یا Lazy Loading می‌شود. هنگامی که DataContractSerializer موجودیت‌ها را سریال می‌کند، تمام خواص موجودیت را دستیابی خواهد کرد که منجر به اجرای lazy-loading روی خواص پیمایشی می‌شود. مسلما این رفتار را نمی‌خواهیم. برای رفع این مشکل، مشخصا قابلیت بارگذاری تنبل را خاموش یا غیرفعال کرده ایم.

عنوان: **بهبود عملکرد SQL Server Locks در سیستم‌های با تعداد تراکنش بالا در Entity Framework**
نویسنده: یاسر مرادی
تاریخ: ۱۳۹۳/۰۴/۲۷ ۲۳:۵۰
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, SQL Server, Performance, transaction, Entity Framework 6

بر اساس رفتار پیش فرض در دیتابیس SQL Server، در زمان انجام دادن یک دستور که منجر به ایجاد تغییرات در اطلاعات موجود در جدول می‌شود (برای مثال دستور Update)، جدول مربوطه به صورت کامل Lock می‌شود، ولو آن دستور Update، فقط با یکی از رکوردهای آن جدول کار داشته باشد.

در سیستم‌های با تعداد تراکنش بالا و دارای تعداد زیاد کلاینت، این رفتار پیش فرض موجب ایجاد صفی از تراکنش‌های در حال انتظار بر روی جداولی می‌شود که ویرایش‌های زیادی بر روی آنها رخ می‌دهد.
اگر چه که بنظر این مشکل راه حل‌های زیادی دارد ، لکن آن راه حلی که همیشه موثر عمل می‌کند استفاده از SQL Server Table Hints است.
SQL Server Table Hints به تمامی آن دستوراتی گفته می‌شود که هنگام اجرای دستور اصلی (برای مثال Select و یا Update) رفتار پیش فرض SQL Server را بر اساس Hint ارائه شده تغییر می‌دهند.
لیست کامل این Hintها را می‌توانید در اینجا مشاهده کنید .
Hint ای که در اینجا برای ما مفید است، آن است که به SQL Server بگوییم هنگام اجرای دستور Update، به جای Lock کردن کل جدول، فقط رکورد در حال ویرایش را Lock کند، و این باعث می‌شود تا باقی تراکنش ها، که ای بسا با سایر رکوردهای آن جدول کار داشته باشند متوقف نشوند، که البته این مسئله کمی به افزایش مصرف حافظه می‌انجامد، لکن مقدار افزایش بسیار ناچیز است.
این Hint که rowlock نام دارد در تراکنش‌های با Isolation Level تنظیم شده بر روی Snapshot باید با یک Table Hint دیگر با نام updlock ترکیب شود.

توضیحات مفصل‌تر این دو Hint در لینک مربوطه آمده است.
بنابر این، بجای دستور

```
update products
set Name = "Test"
Where Id = 1
```

داریم

```
update products with (nolock,updlock)
set Name = "Test"
where Id = 1
```

تا اینجا مشکل خاصی وجود ندارد، آنچه که از اینجا به بعد اهمیت دارد این است که در هنگام کار با Entity Framework، اساسا ما نویسنده دستورات Update نیستیم که به آنها Hint اضافه کنیم یا نه، بلکه دستورات SQL بوسیله Entity Framework ایجاد می‌شوند.
در Entity Framework، مکانیزمی تعبیه شده است با نام Db Command Interceptor که به شما اجازه می‌دهد دستورات SQL ساخته شده را Log کنید و یا قبل از اجرا تغییر دهید ، که برای اضافه نمودن Table Hintها ما از این روش استفاده می‌کنیم، برای انجام این کار داریم: (توضیحات در ادامه)

```
public class UpdateRowLockHintDbCommandInterceptor : IDbCommandInterceptor
    {
        public void NonQueryExecuting(DbCommand command, DbCommandInterceptionContext<Int32>
interceptionContext)
        {
            if (command.CommandType != CommandType.Text) return; // (1)
            if (!(command is SqlCommand)) return; // (2)
            SqlCommand sqlCommand = (SqlCommand)command;
            String commandText = sqlCommand.CommandText;
            String updateCommandRegularExpression = "(update) ";
```

```
            Boolean isUpdateCommand = Regex.IsMatch(commandText, updateCommandRegularExpression,
RegexOptions.IgnoreCase | RegexOptions.Multiline); // You may use better regular expression pattern
here.
            if (isUpdateCommand)
            {
                Boolean isSnapshotIsolationTransaction = sqlCommand.Transaction != null &&
sqlCommand.Transaction.IsolationLevel == IsolationLevel.Snapshot;
                String tableHintToAdd = isSnapshotIsolationTransaction ? " with (rowlock , updlock) set
" : " with (rowlock) set ";
                commandText = Regex.Replace(commandText, "^(set) ", (match) =>
                {
                    return tableHintToAdd;
                }, RegexOptions.IgnoreCase | RegexOptions.Multiline);
                command.CommandText = commandText;
            }
        }
```

این کد در قسمت (1) ابتدا تشخیص می‌دهد که آیا این یک Command دارای Command Text است یا خیر، برای مثال اگر فراخوانی یک Stored Procedure است، ما با آن کاری نداریم.

در قسمت دوم تشخیص می‌دهیم که آیا با SQL Server در حال تعامل هستیم، یا برای مثال با Oracle و ...، که ما برای Tableها فقط با SQL Server کار داریم.

سپس باید تشخیص دهیم که آیا این یک دستور update است یا خیر ؟ برای این منظور از Regular Expressionها استفاده کرده ایم، که خیلی به بحث آموزش این پست مربوط نیست، به صورت کلی از Regular Expressionها برای یافتن و بررسی و جایگزینی عبارات با قاعده در هنگام کار با رشته‌ها استفاده می‌شود.

ممکن است Regular Expression ای که شما می‌نویسید بسیار بهتر از این نمونه باشد، که در این صورت خوشحال می‌شوم در قسمت نظرات آنرا قرار دهید.

در نهایت با بررسی Transaction Isolation Level مربوطه که Snapshot است یا خیر، به درج یک یا هر دو Table Hint مربوطه اقدام می‌نماییم.