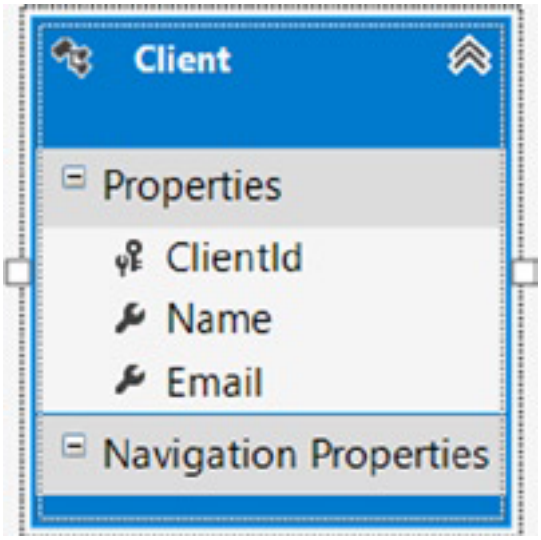
عنوان: **استفاده از EF در اپلیکیشن های N-Tier : قسمت هفتم**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۱/۱۱ ۱:۱۰
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, Tips, WCF, N-Layer Architecture, Entity Framework 6

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.