عنوان: **نوشتن پرس و جو در Entity Framework با استفاده از LINQ To Entity قسمت اول**
نویسنده: محسن جمشیدی
تاریخ: ۱۳۹۲/۰۴/۰۷ ۱۳:۲۰
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, LINQ to Entity

موجودیت‌های زیر را در نظر بگیرید:

```
public class Customer
{
    public Customer()
    {
        Orders = new ObservableCollection<Order>();
    }
    public Guid Id { get; set; }
    public string Name { get; set; }
    public string Family { get; set; }

    public string FullName
    {
        get
        {
            return Name + " " + Family;
        }
    }

    public virtual IList<Order> Orders { get; set; }
}
```

```
public class Product
{
    public Product()
    {
    }

    public Guid Id { get; set; }
    public string Name { get; set; }
    public int Price { get; set; }
}
public class OrderDetail
{
    public Guid Id { get; set; }
    public Guid ProductId { get; set; }
    public int Count { get; set; }
    public Guid OrderId { get; set; }
    public int Price { get; set; }

    public virtual Order Order { get; set; }
    public virtual Product Product { get; set; }

    public string ProductName
    {
        get
        {
            return Product != null ? Product.Name : string.Empty;
        }
    }
}
```

```
public class Order
{
    public Order()
    {
        OrderDetail = new ObservableCollection<OrderDetail>();
    }
    public Guid Id { get; set; }
    public DateTime Date { get; set; }

    public Guid CustomerId { get; set; }
    public virtual Customer Customer { get; set; }
    public virtual IList<OrderDetail> OrderDetail { get; set; }

    public string CustomerFullName
    {
        get
```

```
        {
            return Customer == null ? string.Empty : Customer.FullName;
        }
    }

    public int TotalPrice
    {
        get
        {
            if (OrderDetail == null)
                return 0;

            return
                OrderDetail.Where(orderdetail => orderdetail.Product != null)
                .Sum(orderdetail => orderdetail.Price*orderdetail.Count);
        }
    }
}
```

و نگاشت موجودیت ها:

```
public class CustomerConfiguration : EntityTypeConfiguration<Customer>
{
    public CustomerConfiguration()
    {
        HasKey(c => c.Id);
        Property(c => c.Id).HasDatabaseGeneratedOption(DatabaseGeneratedOption.Identity);
    }
}
public class ProductConfiguration : EntityTypeConfiguration<Product>
{
    public ProductConfiguration()
    {
        HasKey(p => p.Id);
        Property(p => p.Id).HasDatabaseGeneratedOption(DatabaseGeneratedOption.Identity);
    }
}
public class OrderDetailConfiguration : EntityTypeConfiguration<OrderDetail>
{
    public OrderDetailConfiguration()
    {
        HasKey(od => od.Id);
        Property(od => od.Id).HasDatabaseGeneratedOption(DatabaseGeneratedOption.Identity);
    }
}
public class OrderConfiguration: EntityTypeConfiguration<Order>
{
    public OrderConfiguration()
    {
        HasKey(o => o.Id);
        Property(o => o.Id).HasDatabaseGeneratedOption(DatabaseGeneratedOption.Identity);
    }
}
```

و برای معرفی موجودیت‌ها به Entity Framwork کلاس StoreDbContext را به صورت زیر تعریف می‌کنیم:

```
public class StoreDbContext : DbContext
{
    public StoreDbContext()
        : base("name=StoreDb")
    {
    }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Configurations.Add(new CustomerConfiguration());
        modelBuilder.Configurations.Add(new OrderConfiguration());
        modelBuilder.Configurations.Add(new OrderDetailConfiguration());
        modelBuilder.Configurations.Add(new ProductConfiguration());
    }
```

```
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Product> Products { get; set; }
    public DbSet<Order> Orders { get; set; }
    public DbSet<OrderDetail> OrderDetails { get; set; }
}
```

جهت مقدار دهی اولیه به database تستی یک DataBaseInitializer به صورت زیر تعریف می‌کنیم:

```
public class MyTestDb : DropCreateDatabaseAlways<StoreDbContext>
{
    protected override void Seed(StoreDbContext context)
    {
        var customer1 = new Customer { Name = "Vahid", Family = "Nasiri" };
        var customer2 = new Customer { Name = "Mohsen", Family = "Jamshidi" };
        var customer3 = new Customer { Name = "Mohsen", Family = "Akbari" };

        var product1 = new Product {Name = "CPU", Price = 350000};
        var product2 = new Product {Name = "Monitor", Price = 500000};
        var product3 = new Product {Name = "Keyboard", Price = 30000};
        var product4 = new Product {Name = "Mouse", Price = 20000};
        var product5 = new Product {Name = "Power", Price = 70000};
        var product6 = new Product {Name = "Hard", Price = 250000};

        var order1 = new Order
        {
            Customer = customer1, Date = new DateTime(2013, 1, 1),
            OrderDetail = new List<OrderDetail>
            {
                new OrderDetail {Product = product1, Count = 1, Price = product1.Price},
                new OrderDetail {Product = product2, Count = 1, Price = product2.Price},
                new OrderDetail {Product = product3, Count = 1, Price = product3.Price},
            }
        };

        var order2 = new Order
        {
            Customer = customer1,
            Date = new DateTime(2013, 1, 5),
            OrderDetail = new List<OrderDetail>
            {
                new OrderDetail {Product = product1, Count = 2, Price = product1.Price},
                new OrderDetail {Product = product3, Count = 4, Price = product3.Price},
            }
        };

        var order3 = new Order
        {
            Customer = customer1,
            Date = new DateTime(2013, 1, 9),
            OrderDetail = new List<OrderDetail>
            {
                new OrderDetail {Product = product1, Count = 4, Price = product1.Price},
                new OrderDetail {Product = product3, Count = 5, Price = product3.Price},
                new OrderDetail {Product = product5, Count = 6, Price = product5.Price},
            }
        };

        var order4 = new Order
        {
            Customer = customer2,
            Date = new DateTime(2013, 1, 9),
            OrderDetail = new List<OrderDetail>
            {
                new OrderDetail {Product = product4, Count = 1, Price = product4.Price},
                new OrderDetail {Product = product3, Count = 1, Price = product3.Price},
                new OrderDetail {Product = product6, Count = 1, Price = product6.Price},
            }
        };

        var order5 = new Order
        {
            Customer = customer2,
            Date = new DateTime(2013, 1, 12),
            OrderDetail = new List<OrderDetail>
            {
```

```
                new OrderDetail {Product = product4, Count = 1, Price = product4.Price},
                new OrderDetail {Product = product5, Count = 2, Price = product5.Price},
                new OrderDetail {Product = product6, Count = 5, Price = product6.Price},
            }
        };

        context.Customers.Add(customer3);

        context.Orders.Add(order1);
        context.Orders.Add(order2);
        context.Orders.Add(order3);
        context.Orders.Add(order4);
        context.Orders.Add(order5);

        context.SaveChanges();
    }
```

و در ابتدای برنامه کد زیر را جهت مقداردهی اولیه به Database مان قرار می‌دهیم:

```
Database.SetInitializer(new MyTestDb());
```

در انتها ConnectionString را در App.Config به صورت زیر تعریف می‌کنیم:

```
<connectionStrings>
    <add name="StoreDb" connectionString="Data Source=.\SQLEXPRESS;
Initial Catalog=StoreDBTest;Integrated Security = true" providerName="System.Data.SqlClient"/>
</connectionStrings>
```

بسیار خوب، حالا همه چیز محیاست برای اجرای اولین پرس و جو:

```
using (var context = new StoreDbContext())
{
    var query = context.Customers;

    foreach (var customer in query)
    {
        Console.WriteLine("Customer Name: {0}, Customer Family: {1}",
                                customer.Name, customer.Family);
    }
}
```

پرس و جوی تعریف شده لیست تمام Customerها را باز می‌گرداند. query فقط یک "عبارت" پرس و جو هست و زمانی اجرا می‌شود که از آن درخواست نتیجه شود. در مثال بالا این درخواست در اجرای حلقه foreach اتفاق می‌افتد و درست در این لحظه است که دستور SQL ساخته شده و به Database فرستاده می‌شود. EF در این حالت تمام داده‌ها را در یک لحظه باز نمی‌گرداند بلکه این ارتباط فعال است تا حلقه به پایان برسد و تمام داده‌ها از database واکشی شود. خروجی به صورت زیر خواهد بود:

```
Customer Name: Vahid, Customer Family: Nasiri
Customer Name: Mohsen, Customer Family: Jamshidi
Customer Name: Mohsen, Customer Family: Akbari
```

نکته : با هر بار درخواست نتیجه از query ، پرس و جوی مربوطه دوباره به database فرستاده می‌شود که ممکن است مطلوب ما نباشد و باعث افت سرعت شود. برای جلوگیری از تکرار این عمل کافیست با استفاده از متد ToList پرس و جو را در لحظه تعریف به اجرا در آوریم

```
var customers = context.Customers.ToList();
```

خط بالا دیگر یک عبارت پرس و جو نخواهد بود بلکه لیست تمام Customer هاست که به یکباره از database بازگشت داده شده است. در ادامه هرجا که از customers استفاده کنیم دیگر پرس و جویی به database فرستاده نخواهد شد.

پرس و جوی زیر مشتریهایی که نام آنها Mohsen هست را باز می‌گرداند:

```
private static void Query3()
{
    using (var context = new StoreDbContext())
    {
        var methodSyntaxquery = context.Customers
                    .Where(c => c.Name == "Mohsen");
        var sqlSyntaxquery = from c in context.Customers
                             where c.Name == "Mohsen"
                             select c;

        foreach (var customer in methodSyntaxquery)
        {
            Console.WriteLine("Customer Name: {0}, Customer Family: {1}",
                                    customer.Name, customer.Family);
        }
    }

    // Output:
    // Customer Name: Mohsen, Customer Family: Jamshidi
    // Customer Name: Mohsen, Customer Family: Akbari
}
```

همانطور که مشاهده می‌کنید پرس و جو به دو روش Method Syntax و Sql Syntax نوشته شده است.

روش Method Syntax روشی است که از متدهای الحاقی (Extention Method) و عبارت‌های لامبدا (Lambda Expersion) برای نوشتن پرس و جو استفاده می‌شود. اما #C روش Sql Syntax را که همانند دستورات SQL هست، نیز فراهم کرده است تا کسانیکه آشنایی با این روش دارند، از این روش استفاده کنند. در نهایت این روش به Method Syntax تبدیل خواهد شد بنابراین پیشنهاد می‌شود که از همین روش استفاده شود تا با دست و پنجه نرم کردن با این روش، از مزایای آن در بخشهای دیگر کدنویسی استفاده شود.

اگر به نوع Customers که در DbContext تعریف شده است، دقت کرده باشید، خواهید دید که DbSet می‌باشد. DbSet کلاس و اینترفیس‌های متفاوتی را پیاده سازی کرده است که در ادامه با آنها آشنا خواهیم شد:

IQueryable<TEntity>, IEnumerable<TEntity>, IQueryable, IEnumerable: که امکان استفاده از متدهای نام آشنای LINQ را برای ما فراهم می‌کند. البته فراموش نشود که EF از Provider ای با نام LINQ To Entity برای تفسیر پرس و جوی ما و ساخت دستور SQL متناظر آن استفاده می‌کند. بنابراین تمامی متدهایی که در LINQ To Object استفاده می‌شوند در اینجا قابل استفاده نیستند. بطور مثال اگر در پرس و جو از LastOrDefault روی Customer استفاده شود در زمان اجرا با خطای زیر مواجه خواهیم شد و در نتیجه در استفاده از این متدها به این مسئله باید دقت شود.

LINQ to Entities does not recognize the method 'Store.Model.Customer

LastOrDefault[Customer](System.Linq.IQueryable`1[Store.Model.Customer], System.Linq.Expressions.Expression`1[System.Func`2[Store.Model.Customer,System.Boolean]])' method, and this method cannot be translated into a store expression.

<DbSet<TEntity>: که دارای متدهای Add, Attach, Create, Find, Remove, Local می‌باشد و برای بحث ما Find و Local جهت ساخت پرس و جو استفاده می‌شوند که در ادامه توضیح داده خواهند شد.

<DbQuery<TEntity>: که دارای متدهای AsNoTracking و Include می‌باشد و در ادامه توضیح داده خواهند شد.

متد Find: این متد کلید اصلی را به عنوان ورودی گرفته و برای بازگرداندن نتیجه مراحل زیر را طی می‌کند:

داده‌های موجود در حافظه را بررسی می‌کند یعنی آنهایی که Load و یا Attach شده اند.

داده هایی که به DbContext اضافه (Add) ولی هنوز در database درج نشده اند.

داده هایی که در database هستند ولی هنوز Load نشده اند.

Find در صورت پیدا نکردن Exception ای صادر نمی‌کند بلکه مقدار null را بر می‌گرداند.

```
private static void Query4()
{
    using (var context = new StoreDbContext())
    {
        var customer = context.Customers.Find(new Guid("2ee2fd32-e0e9-4955-bace-1995839d4367"));

        if (customer == null)
            Console.WriteLine("Customer not found");
        else
            Console.WriteLine("Customer Name: {0}, Customer Family: {1}", customer.Name,
customer.Family);
    }
}
```

با توجه به اینکه Idها توسط Database ساخته می‌شوند. شما باید از Id دیگری که موجود می‌باشد، استفاده کنید تا نتیجه ای برگشت داده شود.
نکته: در صورتیکه کلید اصلی شما از دو یا چند فیلد تشکیل شده بود. می‌بایست این دو یا چند مقدار را به عنوان پارامتر به Find بفرستید.

متد Single: گاهی نیاز هست که داده‌ای پرس و جو شود اما نه با کلید اصلی بلکه با شرط دیگری، در این حالت از Single استفاده می‌شود. این متد یک مقدار را باز می‌گرداند و در صورتی که صفر یا بیش از یک مقدار در شرط صدق کند exception صادر می‌کند. متد SingleOrDefault رفتاری مشابه دارد اما اگر مقداری در شرط صدق نکند مقدار پیش فرض را باز می‌گرداند.
نکته: مقدار پیش فرض بستگی به نوع خروجی دارد که اگر object باشد مقدار null و اگر بطور مثال نوع عددی باشد، صفر می‌باشد.

```
private static void Query5()
{
    using (var context = new StoreDbContext())
    {
        try
        {
            var customer1 = context.Customers.Single(c => c.Name == "Unkown");  // Exception: Sequence
contains no elements
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
```

```
        }

        try
        {
            var customer2 = context.Customers.Single(c => c.Name == "Mohsen");  // Exception: Sequence
contains more than one element
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }

        var customer3 = context.Customers.SingleOrDefault(c => c.Name == "Unkown");  // customer3 ==
null

        var customer4 = context.Customers.Single(c => c.Name == "Vahid");  // customer4 != null
    }
}
```

متد First: در صورتیکه به اولین نتیجه پرس و جو نیاز هست می‌توان از First استفاده کرد. اگر پرس و جو نتیجه در بر نداشته باشد یعنی null باشد exception صادر خواهد شد اما اگر FirstOrDefault استفاده شود مقدار پیش فرض برگردانده خواهد شد.

```
private static void Query6()
{
    using (var context = new StoreDbContext())
    {
        try
        {
            var customer1 = context.Customers.First(c => c.Name == "Unkown");  // Exception: Sequence
contains no elements
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }

        var customer2 = context.Customers.FirstOrDefault(c => c.Name == "Unknown");  // customer2 ==
null
        var customer3 = context.Customers.First(c => c.Name == "Mohsen");
    }
}
```

نظرات خوانندگان

نویسنده: پژمان
تاریخ: ۱۳۹۲/۱۱/۲۰ ۱۱:۵

خیلی ممنون مهندس، فقط اینکه در داخل سازنده StoreDbContext چرا به این شکل عمل کرده اید:

```
public StoreDbContext()
: base("name=StoreDb")
{
}
```

نویسنده: محسن جمشیدی
تاریخ: ۱۳۹۲/۱۱/۲۰ ۱۱:۳۳

StoreDb نام مدخل Connection String ای هست که در AppConfig ایجاد شده

نویسنده: پژمان
تاریخ: ۱۳۹۲/۱۱/۲۰ ۱۱:۴۶

ممنون از پاسخگوییتون ، در واقع اگر اینکارو نمیکردید باید در AppConfig نام Connection را StoredDbContext میگذاشتیم؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۱۱/۲۰ ۱۲:۲۶

چندین روش برای تعریف رشته اتصالی در EF وجود دارد. بیشتر در اینجا

نویسنده: محسن جمشیدی
تاریخ: ۱۳۹۲/۱۱/۲۰ ۱۴:۳۰

بله البته به همراه namespace اگه اشتباه نکنم

نویسنده: جمشیدی فر
تاریخ: ۱۳۹۳/۰۸/۰۵ ۱۱:۳۷

متد FirstOrDefault باعث اجرای کوئری روی database می‌شه یا از context درون حافظه رکورد مورد نظر را بر میگرداند؟
اگر نیاز باشه که یک رکورد با اجرای کوئری از دیتابیس بازیابی بشه، و نه از context جاری، راه حل چیه؟

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۰۸/۰۵ ۱۲:۲۹

بجای حدس و گمان، خروجی رو لاگ کنید: نمایش خروجی SQL کدهای Entity framework 6 در کنسول دیباگ ویژوال استودیو

نویسنده: حمیدرضا کبیری
تاریخ: ۱۳۹۳/۰۸/۱۰ ۱۴:۴۷

در پرس و جوهای معمولی ، بدین شکل عمل می‌شود که در نهایت نتیجه با شرط یک Id یا چیزی شبیه این مقایسه می‌شود .

```
var Id=1;
var books = (from b in db.Books
                              where b.bookId == Id
                              select new
                              {
                                 //...
```

```
                                }).ToList();
```

حالا اگر شرط من بجای داشتن فقط یک Id لیستی از Id باشد چطور عمل کنم ؟

```
var booksId= new list<int>(){ 1 , 2 , 6 , 7};
var books = (from b in db.Books
                                where b.bookId == ???
                                select new
                                {
                                   //...

                                }).ToList();
```

چطور میتونم لیستی رو که دارم بجای مقایسه با یک Id ، با یک لیستی از Idها مقایسه کنم و نتیجه را بگیرم ؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۸/۱۰ ۱۴:۵۸

از متد Contains استفاده کنید که به where in ترجمه می‌شود:

```
from book in db.Books
where booksId.Contains(book.bookId)
```

عنوان: **نوشتن پرس و جو در Entity Framework با استفاده از LINQ To Entity قسمت دوم**
نویسنده: محسن جمشیدی
تاریخ: ۱۳۹۲/۰۴/۱۳ ۱۰:۵
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, LINQ to Entity

در قسمت قبل با نحوه اجرای پرس و جو آشنا شدید و همچنین به بررسی متدهای Find و Single و First و تفاوت‌های آنها پرداختیم. در این قسمت با خصوصیت Local و متد Load آشنا خواهیم شد. همانطور که در قسمت قبل دیدید، مقادیر اولیه‌ای برای Database و جداولمان مشخص کردیم. برای جدول Customer این داده‌ها را داشتیم:

ID	Name	Family
یک مقدار Guid	Vahid	Nasiri
یک مقدار Guid	Mohsen	Akbari
یک مقدار Guid	Mohsen	Jamshidi

ID توسط Database تولید می‌شوند به همین دلیل از ذکر مقداری مشخص خودداری شده است.
به کد زیر دقت کنید:

```
private static void Query7()
{
    using (var context = new StoreDbContext())
    {
        // Add
        context.Customers.Add(new Customer { Name = "Ali", Family = "Jamshidi" });

        // change
        var customer1 = context.Customers.Single(c => c.Family == "Jamshidi");
        customer1.Name = "Mohammad";

        // Remove
        var customer2 = context.Customers.Single(c => c.Family == "Akbari");
        context.Customers.Remove(customer2);

        var customers = context.Customers.Where(c => c.Name != "Vahid");

        foreach (var cust in customers)
        {
            Console.WriteLine("Customer Name: {0}, Customer Family: {1}", cust.Name, cust.Family)
        }
    }
}
```

همانطور که مشاهده می‌کنید عمل اضافه، تغییر و حذف روی Customer انجام شده ولی هنوز هیچ تغییری در Database ذخیره نشده است. آخرین پرس و جو چه نتیجه ای را دربر خواهد داشت؟

بله، فقط تغییر یک موجودیت در نظر گرفته شده است ولی اضافه و حذف نه!
نتیجه مهمی که حاصل می‌شود این است که در پرس و جوهایی که روی Database اجرا می‌شوند سه مورد را باید در نظر داشت:

داده هایی که اخیرا به DbContext اضافه شده‌اند ولی هنوز در Database ذخیره نشده‌اند، درنظر گرفته نخواهند شد.

داده هایی که در DbContext حذف شده‌اند ولی در Database هستند، در نتیجه پرس و جو خواهند بود.

داده هایی که قبلا از database توسط پرس و جوی دیگری گرفته شده و تغییر کرده‌اند، آن تغییرات در نتیجه پرس و جو موثر

خواهند بود.

پس پرس و جوهای LINQ ابتدا روی database انجام می‌شوند و idهای بازگشت داده شده با idهای موجود در DbContext مطابقت داده می‌شوند یا در DbContext وجود دارند که در این صورت آن موجودیت بازگشت داده می‌شود یا وجود ندارند که در این صورت موجودیتی که از Database خوانده شده، بازگشت داده می‌شوند.
برای درک بیشتر کد زیر را در نظر بگیرید:

```
private static void Query7_1()
{
    using (var context = new StoreDbContext())
    {
        // Add
        context.Customers.Add(new Customer { Name = "Ali", Family = "Jamshidi" });

        // change
        var customer1 = context.Customers.Single(c => c.Family == "Jamshidi");
        customer1.Name = "Vahid";

        // Remove
        var customer2 = context.Customers.Single(c => c.Family == "Akbari");
        context.Customers.Remove(customer2);

        var customers = context.Customers.Where(c => c.Name != "Vahid");

        foreach (var cust in customers)
        {
            Console.WriteLine("Customer Name: {0}, Customer Family: {1}", cust.Name, cust.Family)
        }
    }
}
```

این کد همان کد قبلی است اما نام customer1 در DbContext (که Mohsen بوده در Database) به Vahid تغییر کرده و پرس و جو روی نام هایی زده شده است که Vahid نباشند خروجی به صورت زیر خواهد بود:

Customer Name: Vahid, Customer Family: Jamshidi

Customer Name: Mohsen, Customer Family: Akbari Vahid در خروجی آمده در صورتیکه در شرط صدق نمی‌کند چراکه پرس و جو روی Database زده شده، جاییکه نام این مشتری Mohsen بوده اما موجودیتی بازگشت داده شده که دارای همان Id هست اما در DbContext دستخوش تغییر شده است.

Local: همانطور که قبلا اشاره شد خصوصیتی از DbSet می‌باشد که شامل تمام داده هایی هست که:

اخیرا از database پرس و جو شده است (می‌تواند تغییر کرده یا نکرده باشد)

اخیرا به Context اضافه شده است (توسط متد Add)

دقت شود که Local شامل داده‌هایی که از database خوانده شده و از Context، حذف (Remove) شده‌اند، نمی‌باشد.
نوع این خصوصیت ObservableCollection می‌باشد که می‌توان از آن برای Binding در پروژه‌های ویندوزی استفاده کرد.
به کد زیر دقت کنید:

```
private static void Query8()
{
    using (var context = new StoreDbContext())
    {
        // Add
        context.Customers.Add(new Customer { Name = "Ali", Family = "Jamshidi" });

        // change
        var customer1 = context.Customers.Single(c => c.Family == "Jamshidi");
```

```
        customer1.Name = "Mohammad";

        // Remove
        var customer2 = context.Customers.Single(c => c.Family == "Akbari");
        context.Customers.Remove(customer2);

        var customers = context.Customers.Local;

        foreach (var cust in customers)
        {
            Console.WriteLine("Customer Name: {0}, Customer Family: {1}", cust.Name, cust.Family);
        }
    }
}
```

کد بالا شبیه به کد قبلی می‌باشد با این تفاوت که در انتها foreach روی Local زده شده است. خروجی به صورت زیر خواهد بود:

همانطور که ملاحظه می‌کنید Local شامل Ali Jamshidi که اخیرا اضافه شده (ولی در Database ذخیره نشده) و Mohammad Jamshidi که از Database خوانده شده و تغییر کرده، می‌باشد اما شامل Mohsen Akbari که از Database خوانده شده اما در Context حذف شده است، نمی‌باشد.
می‌توان روی Local نیز پرس و جوی اجرا کرد. در این صورت از پروایدر LINQ To Object استفاده خواهد شد و درنتیجه دست بازتر هست و تمام امکانات این پروایدر می‌توان استفاده کرد.

Load: یکی دیگر از مواردی که باعث اجرای پرس و جو می‌شود متد Load می‌باشد که یک Extension Method می‌باشد. این متد در حقیقت یک پیمایش روی پرس و جو انجام می‌دهد و باعث بارگذاری داده‌ها در Context می‌شود. مانند استفاده از ToList البته بدون ساختن List که سربار ایجاد می‌کند.

```
private static void Query9()
{
    using (var context = new StoreDbContext())
    {
        var customers = context.Customers.Where(c => c.Name == "Mohsen");
        customers.Load();

        foreach (var cust in context.Customers.Local)
        {
            Console.WriteLine("Customer Name: {0}, Customer Family: {1}", cust.Name, cust.Family);
        }
    }
    // Output:
    // Customer Name: Mohsen, Customer Family: Akbari
    // Customer Name: Mohsen, Customer Family: Jamshidi
}
```

نظرات خوانندگان

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۴/۱۳ ۱۰:۴۸

این دوجمله رو

«- داده هایی که اخیرا به DbContext اضافه شده‌اند ولی هنوز در Database ذخیره نشده‌اند، درنظر گرفته نخواهند شد.
- داده هایی که در DbContext حذف شده‌اند ولی در Database هستند، در نتیجه پرس و جو خواهند بود.»

میشه خلاصه‌اش کرد به «تا زمانیکه SaveChanges فراخوانی نشه، از اطلاعات تغییر کرده نمیشه کوئری گرفت (کوئری‌ها همیشه روی دیتابیس انجام می‌شن)؛ اما خاصیت Local این تغییرات محلی رو داره یا اینکه در change tracker میشه موارد EntityState.Added | EntityState.Modified | EntityState.Unchanged رو هم کوئری گرفت».

نویسنده: محسن جمشیدی
تاریخ: ۱۳۹۲/۰۴/۱۳ ۱۴:۷

دقیقا!
جهت تاکید بیشتر روی "اجرا شدن پرس و جو در Database نه DbContext" متد Query7_1 به متن اضافه شد

نویسنده: مجید_فاضلی نسب
تاریخ: ۱۳۹۲/۰۸/۲۶ ۲۳:۱۴

DbContext دقیقا چیه ؟

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۸/۲۷ ۰:۵۱

قسمت‌های 11 و 12 سری EF رو مطالعه کنید.

نویسنده: پژمان
تاریخ: ۱۳۹۲/۱۱/۲۰ ۱۶:۵۳

با تشکر از مقاله آموزندتون، خواستم بدونم که مثلا در چه سناریویی بهتر است از متد .Localاستفاده کرد(یا به عبارتی این متد کی به درد کار ما میخورد)، با توجه به اینکه این متد اطلاعاتی که به اصطلاح In-Memory هستند را برای ما میآورد؟

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۱۱/۲۰ ۱۷:۷

بیشتر برای استفاده در WPF و برنامه‌های دسکتاپ است .

نویسنده: پژمان
تاریخ: ۱۳۹۲/۱۱/۲۰ ۱۷:۱۰

خیلی ممنون، در مورد Load هم تو نت گشتم چیزی دستگیرم نشد، اگه در این مورد هم مقاله ای باز سراغ دارید ممنون میشم لینکشو بدید ممنوون

نویسنده: محسن خان

تاریخ: ۱۳۹۲/۱۱/۲۰ ۱۷:۲۵

در همان مقاله‌ای که لینک دادم، اواسط آن به Load هم پرداخته. کاربرد عملی آن در پروژه طراحی فریم ورک WPF با EF در سایت هست.

نویسنده: مهدی سعیدی فر
تاریخ: ۱۳۹۲/۱۱/۲۰ ۱۸:۱۲

یکی از کاربرداش در حذف اشیاء مرتبط

عنوان: نوشتن پرس و جو در Entity Framework با استفاده از LINQ To Entity قسمت سوم
نویسنده: محسن جمشیدی
تاریخ: ۱۳۹۲/۰۴/۲۰ ۱۸:۵۵
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, LINQ to Entity

اجرای پرس و جو روی داده‌های به هم مرتبط (Related Data)

اگر به موجودیت Customer دقت کنید دارای خصوصیتی با نام Orders می‌باشد که از نوع IList<Order> هست یعنی دارای لیستی از Order هاست بنابراین یک رابطه یک به چند بین Customer و Order وجود دارد. در ادامه به بررسی نحوه پرس و جو کردن روی داده‌های به هم مرتبط خواهیم پرداخت.

ابتدا به کد زیر دقت کنید:

```
private static void Query10()
{
    using (var context = new StoreDbContext())
    {
        var customers = context.Customers;
        foreach (var customer in customers)
        {
            Console.WriteLine("Customer Name: {0}, Customer Family: {1}", customer.Name,
customer.Family);
            foreach (var order in customer.Orders)
            {
                Console.WriteLine("\t Order Date: {0}", order.Date);
            }
        }
    }
}
```

اگر کد بالا را اجرا کنید هنگام اجرای حلقه داخلی با خطای زیر مواجه خواهید شد:

```
System.InvalidOperationException: There is already an open DataReader associated with this Command
which must be closed first
```

همانطور که قبلا اشاره شد EF با اجرای یک پرس و جو به یکباره داده‌ها را باز نمی‌گرداند بنابراین در حلقه اصلی که روی Customers زده شده است با هر پیمایش یک customer از Database فراخوانی می‌شود درنتیجه DataReader تا پایان یافتن حلقه باز می‌ماند. حال آنکه حلقه داخلی نیز برای خواندن Orderها نیاز به اجرای یک پرس و جو دارد بنابراین DataReader ای جدید باز می‌شود و در نتیجه با خطایی مبنی بر اینکه DataReader دیگری باز است، مواجه می‌شویم. برای حل این مشکل می‌بایست جهت باز بودن چند DataReader همزمان، کد زیر را به ConnectionString اضافه کنیم

```
MultipleActiveResultSets = true
```

که با این تغییر کد بالا به درستی اجرا می‌شود.

در بارگذاری داده‌های به هم مرتبط EF سه روش را در اختیار ما قرار می‌دهد:

Lazy Loading

Eager Loading

Explicit Loading

که در ادامه به بررسی آنها خواهیم پرداخت.

Lazy Loading: در این روش داده‌های مرتبط در صورت نیاز با یک پرس وجوی جدید که به صورت اتوماتیک توسط EF ساخته می‌شود، گرفته خواهند شد. کد زیر را در نظر بگیرید:

```
private static void Query11()
{
    using (var context = new StoreDbContext())
    {
        var customer = context.Customers.First();

        Console.WriteLine("Customer Name: {0}, Customer Family: {1}", customer.Name, customer.Family);
        foreach (var order in customer.Orders)
        {
            Console.WriteLine("\t Order Date: {0}", order.Date);
        }
    }
}
```

اگر این کد را اجرا کنید خواهید دید که یک بار پرس و جویی مبنی بر دریافت اولین Customer روی database زده خواهد شد و پس از چاپ آن در ادامه برای نمایش Orderهای این Customer پرس و جوی دیگری زده خواهد شد. در حقیقت پرس و جوی اول فقط Customer را بازگشت می‌دهد و در ادامه، اول حلقه، جایی که نیاز به Orderهای این Customer می‌شود EF پرس و جو دوم را بصورت هوشمندانه و اتوماتیک اجرا می‌کند. به این روش بارگذاری داده‌های مرتبط Lazy Loading گفته می‌شود که به صورت پیش فرض در EF فعال است.

برای غیرفعال کردن این روش، کد زیر را اجرا کنید:

```
context.Configuration.LazyLoadingEnabled = false;
```

EF از dynamic proxy برای Lazy Loading استفاده می‌کند. به این صورت که در زمان اجرا کلاسی جدید که از کلاس POCO مان ارث برده است، ساخته می‌شود. این کلاس proxy می‌باشد و در آن navigation propertyها بازنویسی شده‌اند و کمی منطق برای خواندن داده‌های وابسته اضافه شده است.

برای ایجاد dynamic proxy شروط زیر لازم است:

- کلاس POCO می‌بایست public بوده و sealed نباشد.
- Navigation propertyها می‌بایست virtual باشد.

در صورتیکه هرکدام از این دو شرط برقرار نباشند کلاس proxy ساخته نمی‌شود و Lazy Loading حتی در صورت فعال بودن انجام نخواهد شد. مثلا اگر پراپرتی Orders در کلاس Customer مان virtual نباشد. در شروع حلقه کد بالا پرس و جوی جدید اجرا نشده و در نتیجه مقدار این پراپرتی null خواهد ماند.

Lazy Loading به ما در عدم بارگذاری داده‌های مرتبط که به آنها نیازی نداریم، کمک می‌کند. اما در صورتیکه به داده‌های مرتبط نیاز داشته باشیم "مسئله Select n+1" پیش خواهد آمد که باید این مسئله را مد نظر داشته باشیم.

مسئله Select n+1: کد زیر را در نظر بگیرد

```
private static void Query12()
{
    using (var context = new StoreDbContext())
    {
        var customers = context.Customers;
        foreach (var customer in customers)
        {
            Console.WriteLine("Customer Name: {0}, Customer Family: {1}", customer.Name,
customer.Family);
            foreach (var order in customer.Orders)
            {
                Console.WriteLine("\t Order Date: {0}", order.Date);
            }
        }
    }
}
```

هنگام اجرای کد بالا یک پرس و جو برای خواندن Customerها زده خواهد شد و به ازای هر Customer یک پرس و جوی دیگر برای گرفتن Orderها زده خواهد شد. در این صورت پرس و جوی اول ما اگر n مشتری را برگرداند، n پرس و جو نیز برای گرفتن Orderها زده خواهد شد که روهم n+1 دستور Select می‌شود. این تعداد پرس و جو موجب عدم کارایی می‌شود و برای رفع این مسئله نیاز به امکانی جهت بارگذاری هم زمان داده‌های مرتبط مورد نیاز خواهد بود. این امکان با استفاده از Eager Loading

برآورده می‌شود.

روش Eager Loading: هنگامی که در یک پرس و جو نیاز به بارگذاری همزمان داده‌های مرتبط نیز باشد، از این روش استفاده می‌شود. برای این منظور از متد Include استفاده می‌شود که ورودی آن navigation property مربوطه می‌باشد. این پارامتر ورودی را همانطور که در کد زیر مشاهده می‌کنید، می‌توان به صورت string و یا Lambda Expression مشخص کرد.
دقت شود که برای حالت Lambda Expression بایدSystem.Data.Entity به usingها اضافه شود.

```
private static void Query13()
{
    using (var context = new StoreDbContext())
    {
        var customers = context.Customers.Include(c => c.Orders);
        //var customers = context.Customers.Include("Orders");
        foreach (var customer in customers)
        {
            Console.WriteLine("Customer Name: {0}, Customer Family: {1}", customer.Name,
customer.Family);
            foreach (var order in customer.Orders)
            {
                Console.WriteLine("\t Order Date: {0}", order.Date);
            }
        }
}
```

در این صورت یک پرس و جو به صورت join اجرا خواهد شد.
اگر داده‌های مرتبط در چند سطح باشند، می‌توان با دادن مسیر داده‌های مرتبط اقدام به بارگذاری آنها کرد. به مثالهای زیر توجه کنید:

```
context.OrderDetails.Include(o => o.Order.Customer)
```

در پرس و جوی بالا به ازای هر OrderDetail داده‌های مرتبط Order و Customer آن بارگذاری می‌شود.

```
context.Orders.Include(o => o.OrderDetail.Select(od => od.Product))
```

در پرس و جوی بالا به ازای هر Order لیست OrderDetail ها و برای هر OrderDetail داده مرتبط Product آن بارگذاری می‌شود.

```
context.Orders.Include(o => o.Customer).Include(o => o.OrderDetail)
```

در پرس و جوی بالا به ازای هر Order داده‌های مرتبط OrderDetail و Customer آن بارگذاری می‌شود.

روش Explicit Loading: این روش مانند Lazy Loading می‌باشد که می‌توان داده‌های مرتبط را جداگانه فراخوانی کرد اما نه به صورت اتوماتیک توسط EF بلکه به صورت صریح توسط خودمان انجام می‌شود. این روش حتی اگر navigation propertyهای ما virtual نباشند نیز قابل انجام است. برای انجام این روش از متد DbContext.Entry استفاده می‌شود.

```
private static void Query14()
{
    using (var context = new StoreDbContext())
    {
        var customer = context.Customers.First(c => c.Family == "Jamshidi");

        context.Entry(customer).Collection(c => c.Orders).Load();

        foreach (var order in customer.Orders)
        {
            Console.WriteLine(order.Date);
        }
    }
}
```

در پرس و جوی بالا تمام Orderهای یک Customer به صورت جدا گرفته شده است برای این منظور از چون Orders یک لیست می‌باشد، از متد Collection استفاده شده است.

```
private static void Query15()
{
    using (var context = new StoreDbContext())
    {
        var order = context.Orders.First();

        context.Entry(order).Reference(o => o.Customer).Load();

        Console.WriteLine(order.Customer.FullName);
    }
}
```

در پرس و جوی بالا Customer یک Order صراحتا و به صورت جداگانه از database گرفته شده است.
با توجه به دو مثال بالا مشخص است که اگر داده مرتبط ما به صورت لیست است از Collection و درغیر این صورت از Reference استفاده می‌شود.
در صورتیکه بخواهیم ببینیم آیا داده‌ی مرتبط مان بازگذاری شده است یا خیر، از خصوصیت IsLoaded به صورت زیر استفاده می‌کنیم:

```
if (context.Entry(order).Reference(o => o.Customer).IsLoaded)
    context.Entry(order).Reference(o => o.Customer).Load();
```

و در آخر اگر بخواهیم روی داده‌های مرتبط پرس و جو اجرا کنیم نیز این قابلیت وجود دارد. برای این منظور از Query استفاده می‌کنیم.

```
private static void Query16()
{
    using (var context = new StoreDbContext())
    {
        var customer = context.Customers.First(c => c.Family == "Jamshidi");

        IQueryable<Order> query = context.Entry(customer).Collection(c => c.Orders).Query();

        var order = query.First();
    }
}
```

نظرات خوانندگان

نویسنده: مهدی زارعی
تاریخ: ۱۳۹۲/۰۵/۲۹ ۱۱:۱۴

این سری مطالب بسیار خوب و مفید است .از نویسنده محترم خواهش می‌کنم نگارش این مجموعه را متوقف نکند .
در صورتی که برای ایشون امکان پذیر نیست خواهش می‌کنم منبع یا منابعی که از آن‌ها در مورد این سری مقالات به آن رجوع کرده اند را معرفی کنند.
با تشکر از زحماتشان

نویسنده: محسن جمشیدی
تاریخ: ۱۳۹۲/۰۵/۲۹ ۱۲:۸

منبع کتابهایی هست که در اینجا معرفی شده

نویسنده: علیرضا
تاریخ: ۱۳۹۲/۰۵/۳۱ ۰:۴۰

در خصوص این قسمت:
".... در پرس و جوی بالا به ازای هر OrderDetail داده‌های مرتبط Order و Customer آن بارگذاری می‌شود.

context.Orders.Include(o => o.OrderDetail.Select(od => od.Product))	1

" بهتره برای ابهام زدایی ذکر کنید که OrderDetail یک Collection است و نمیتوان مانند پرس و جوی مثال قبلی از o=> o.OrderDerail.Product استفاده کرد.

نویسنده: علیرضا
تاریخ: ۱۳۹۲/۰۵/۳۱ ۰:۴۳

در صورتیکه بخواهیم ببینیم آیا داده‌ی مرتبط مان بازگذاری شده است یا خیر، از خصوصیت IsLoaded به صورت زیر استفاده می‌کنیم:

```
if (context.Entry(order).Reference(o => o.Customer).IsLoaded)
```

منظور Not Isloaded بوده که ظاهرا ! جا افتاده

عنوان: متد LastOrDefault در EF
نویسنده: حسین مرادی نیا
تاریخ: ۱۳۹۲/۰۵/۰۱ ۸:۳۰
آدرس: www.dotnettips.info
برچسب‌ها: Entity framework, LINQ to Entity

اگر بخواهیم اولین رکورد از یک جدول را توسط EF درخواست نماییم از متد FirstیاFirstOrDefault استفاده می‌شود. برای مثال واکشی اولین رکورد از جدول Student به صورت زیر است:

```
var student=context.Students.FirstOrDefault();
```

در این حالت اولین رکورد از جدول student واکشی می‌شود و اگر رکوردی موجود نباشد یک مقدار null بازگشت داده می‌شود. حال اگر بخواهید به جای اولین رکورد آخرین رکورد را واکشی نمایید چطور؟ برای یافتن آخرین رکورد در لیست‌ها ی Generic و کلا لیست‌های Enumerable از متد LastOrDefault استفاده می‌شود. با این حال این متد توسط Entity Framework پشتیبانی نمی‌شود و در صورتی که از کد زیر استفاده کنید برنامه با خطا متوقف خواهد شد:

```
var student=context.Students.LastOrDefault();
```

دو راه حل برای رفع این مشکل به ذهن می‌رسد:
روش اول: می‌توان خروجی را ابتدا به یک نوع Enumerable مانند List تبدیل کرد و سپس از متد LastOrDefault استفاده کرد. کد زیر را در نظر بگیرید:

```
var student=context.Students.ToList().LastOrDefault();
```

در کد بالا ابتدا رکوردهای جدول Student از درون بانک اطلاعات به صورت کامل واکشی شده و سپس رکورد آخر از میان آنها جدا می‌شود. این حالت در حالی که رکوردهای کمی در جدول وجود داشته باشد روش بدی به حساب نمی‌آید ولی اگر تعداد رکوردها زیاد باشد (اکثر مواقع نیز به همین شکل است) روش مناسبی نمی‌باشد و باعث کندی برنامه می‌شود.

روش دوم: با توجه به اینکه تنها به یک رکورد (آخرین رکورد) نیاز داریم بهتر است یک رکورد هم واکشی شود. در این روش برای اینکه بتوان به آخرین رکورد رسید ابتدا رکوردهای جدول را به صورت نزولی مرتب می‌کنیم و سپس از متد FirstOrDefault برای واکشی آخرین رکورد استفاده می‌نمایید. برای مثال:

```
var student=context.Students.OrderByDescending(s=>s.Id).FirstOrDefault();
```

در کد بالا ابتدا رکوردها را بر اساس فیلد مورد نظر به صورت نزولی مرتب کرده ایم(در نظر داشته باشید عملیات مرتب سازی را می‌توان بر اساس فیلدی که مورد نظر است انجام داد) و پس از آن با توجه به اینکه رکوردها به صورت نزولی مرتب شده اند و رکورد آخر به اول منتقل شده است از متد FirstOrDefault جهت دسترسی به آخرین رکورد که در حال حاضر اول لیست رکوردها است استفاده شده است. سرعت این روش به مراتب از روش اول بیشتر می‌باشد. برای بالا رفتن سرعت مرتب سازی در جداول بزرگ نیز می‌توان از تدابیری همچون Index گذاری بر روی فیلدها در DataBase استفاده کرد(با توجه به فیلدی که قرار است مرتب سازی بر اساس آن انجام شود).

نظرات خوانندگان

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۵/۰۱ ۸:۳۸

ممنون. روش دوم به select top 1 در حین استفاده از SQL Server ترجمه میشه.

نویسنده: سامان هاشمی
تاریخ: ۱۳۹۲/۰۵/۰۱ ۱۱:۵۴

مورد اول اصلا توصیه نکنید بعدها به دلیل مشکل کارآیی که داره خیلی اذیت میکنه همون مورد دوم تنها گزینه و بهترین گزینه است!

نویسنده: ابوالفضل
تاریخ: ۱۳۹۲/۰۵/۰۲ ۱۲:۳۴

یک نکته اینکه : زمانی که قصد داریم آخرین رکورد افزوده شده رو به این طریق و بر اساس فیلدی غیر از کلید واکشی کنیم (به فرض تاریخ فاکتور و ...) حتما باید برای آن فیلد در صورت کلید نبودنش ، ایندکس ایجاد کنیم تا واکشی در کوتاهترین زمان ممکن در حجم بالای اطلاعات صورت گیرد .

نویسنده: باغبان
تاریخ: ۱۳۹۲/۰۵/۰۳ ۱۵:۳۹

ممنون از آموزش خوبتون می‌خواستم بپرسم در حالت دوم فقط یک رکورد از دیتابیس واکشی میشه؟ یا اینکه از میون رکوردهای واکشی شده یک رکورد را انتخاب می‌کنه؟

نویسنده: حسین مرادی نیا
تاریخ: ۱۳۹۲/۰۵/۰۳ ۱۷:۸

در روش دوم فقط یک رکورد واکشی می‌شود.

عنوان: **پراپرتی سفارشی در EF Database First**
نویسنده: جمشیدی فر
تاریخ: ۱۳۹۲/۰۸/۰۵ ۱۸:۲۰
آدرس: www.dotnettips.info
برچسب‌ها: Entity framework, LINQ to Entity, EF Database First

فرض کنید در روش EF Database First می‌خواهید فیلدی به مدل اضافه شود که در دیتابیس وجود ندارد، درواقع فیلدی محاسباتی به مدل اضافه کنید.راه حل چیست؟ اولین روشی که ممکن است به ذهن برسد این است که به کلاس مدل که از جدول دیتابیس ساخته شده، فیلدی محاسباتی اضافه می‌کنیم.

```
public class Person
{
    public string FullName {
        get
        {
            return FirstName + " " + LastName;
        }
    }
}
```

به نظر راه حل درستی می‌رسد، اما مشکل این روش چیست؟ اگر مدل برنامه از روی دیتابیس بروزشود، چه اتفاقی می‌افتد؟ خب قاعدتا این فیلد محاسباتی از دست می‌رود و باید دوباره آن را به کلاس مدل جدید و بروز شده اضافه کنیم. که به نظر راه حل منطقی و خوبی نمیرسد. در چنین مواقعی می‌توان به جای اینکه این پراپرتی را به کلاس تولید شده از روی دیتابیس اضافه کرد، این فیلد را به یک Partial Class از کلاس تولید شده که درهمان پروژه و فضای نام کلاس تولید شده از دیتابیس قراردارد، اضافه کرد. به این ترتیب با هر بار بروز شدن مدل از روی دیتابیس، این فیلد از بین نمی‌رود.

```
public partial class Person
{
    public string FirstName {get; set;}
    public string LastName {get; set; }
}
public partial class Person
{
    public string FullName {
        get
        {
            return FirstName + " " + LastName;
        }
    }
}
```

اما این روش محدودیت هایی نیز دارد :

1. همه قسمت‌های Partial Class باید در یک اسمبلی باشند.
2. پراپرتی‌های درون Partial Class در دیتابیس ذخیره نمی‌شوند.
3. ونیز این پراپرتی‌ها در عبارات Linq قابل استفاده نیستند.چون عبارت Linq در نهایت به یک رشته SQL تبدیل شده و در دیتابیس اجرا می‌شود.البته با فرض این که دیتاپرووایدر، SQL باشد.

نظرات خوانندگان

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۸/۰۵ ۲۱:۴۱

در مورد روش دوم؛ (با توجه به جمله بندی) آیا در روش اول این خاصیت‌های محاسباتی در بانک اطلاعاتی ذخیره می‌شوند؟ ضمنا جهت تکمیل بحث، این خاصیت‌ها (هر دو حالت) در عبارات LINQ to Objects قابل استفاده هستند.

نویسنده: جمشیدی فر
تاریخ: ۱۳۹۲/۰۸/۰۶ ۸:۳۸

در مورد سوال اول، خیر؛ در روش اول هم این خاصیت‌ها در دیتابیس ذخیره نمی‌شوند.اینجا مساله این است که ما می‌خواهیم با هر بار اپدیت مدل از دیتابیس، فیلد محاسباتی دوباره محاسبه شود و ازبین نرود.

درمورد سوال دوم، شما درست می‌فرمایید. من باید Linq رو به Linq to Entity تغییر بدم.