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