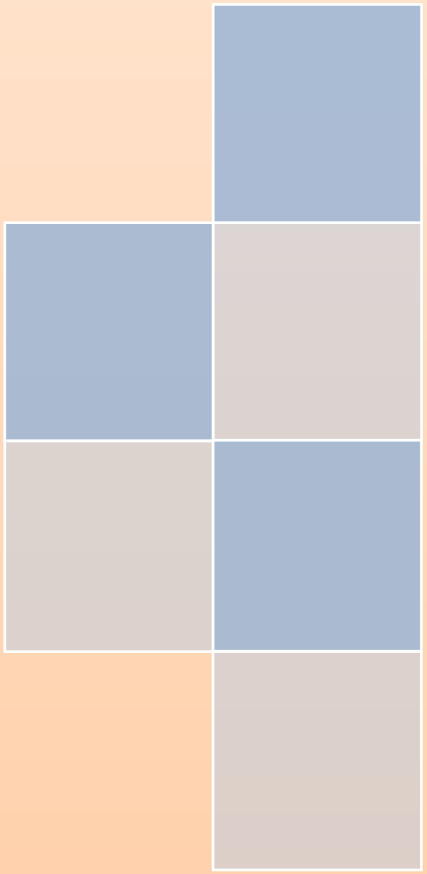
آموزش NHibernate

مقالات آموزشی وبلاگ مهندس وحید نصیری

کتابخانه‌ی تبدیل شده پروژه بسیار محبوب Hibernate جاوا به سی شارپ است و یکی از ORM های بسیار موفق، به شمار می‌رود. در طی تعدادی مقاله قصد آشنایی با این فریم ورک را داریم.



تهیه کننده: مهدی پایروند

بهار 1389

فهرست مطالب

قسمت اول 4

چرا نیاز است تا از یک ORM استفاده شود؟ 4

بررسی مدل سیستم ثبت سفارشات 4

نگاشت مدل 5

آماده سازی سیستم برای استفاده از NHibernate 5

برپایی یک پروژه جدید 7

قسمت دوم 9

آزمون واحد کلاس نگاشت تهیه شده 9

توضیحات: 10

نکته: 12

قسمت سوم 13

نکته: 20

قسمت چهارم 25

توضیحات: 28

نکته: 30

قسمت پنجم 31

استفاده از LINQ جهت انجام کوئری‌ها توسط NHibernate 31

نکته: 34

قسمت ششم 36

آشنایی با Automapping در فریم ورک Fluent Nhibernate 36

مباحث تکمیلی AutoMapping 43

نکته: 44

قسمت هفتم 45

مدیریت بهینه‌ی سشن فکتوری 45

مدیریت سشن فکتوری در برنامه‌های وب 47

مدیریت سشن فکتوری در برنامه‌های غیر وب 52

خلاصه‌ای از آغاز به کار باNHibernate 52

قسمت هشتم 53

معرفی الگوی Repository 53
بررسی مدل برنامه 54
پیاده سازی الگوی مخزن 56
قسمت نهم 61
استفاده از Log4Net جهت ثبت خروجی‌های SQL حاصل از NHibernate 61
توضیحاتی در مورد تنظیمات فوق: 62
قسمت دهم 63
آشنایی با کتابخانه Validator NHibernate 63
کامپایل پروژه اعتبار سنجی NHibernate 63
بررسی مدل برنامه 63
تعریف اعتبار سنجی دومین با کمک ویژگی‌ها (attributes) 64
تعریف اعتبار سنجی با کمک کلاس ValidationDef 65
استفاده از قیودات تعریف شده به صورت دستی 66
نکته مهم: 67
استفاده از قیودات تعریف شده و سیستم اعتبار سنجی به صورت یکپارچه با NHibernate 67
نکته: 69
قسمت یازدهم 70
یکسان سازی ی و ک دریافتی حین استفاده ازNHibernate 70
قسمت دوازدهم 72
NHibernate 3.0و ارائه‌ی جایگزینی جهتICriteria API 72
قسمت سیزدهم 73
NHibernate 3.0و عدم وابستگی مستقیم بهLog4Net 73
قسمت چهاردهم 74
NHibernate 3.0و خواص تنبل!(lazy properties) 74
ب) ساختار جداول متناظر (تولید شده به صورت خودکار توسط Fluent NHibernate در اینجا) 76
ج) صفحه‌ی ثبت صورتحساب‌ها 77
ه) پیاده سازی با framework Entity 79
قسمت پانزدهم 81
ذخیره سازی SQL تولیدی درNH3 81

SQLتولیدی در NHibernate از کدام متد صادر شده است؟ 81
قسمت شانزدهم 84
مدیریت Join در NHibernate 3.0 84
مدل برنامه: 84
دریافت اطلاعات : 87
قسمت هفدهم 91
سرویس جمع و مفرد سازی اسامی 91
قسمت هجدهم 93
NHibernateو سطح اول cache آن 93
نحوه‌ی نگاشت فیلدهای فرمول درFluent NHibernate 98
مکان اصلی یافتن آخرین نگارش‌هایFluent NHibernate 101
NH 3.2و تاثیر آن بر آینده‌یFHN 103
فعال سازی سطح دوم کش درFluent Nhibernate 104

قسمت اول

کتابخانه‌ی تبدیل شده پروژه بسیار محبوب Hibernate جاوا به سی شارپ است و یکی از ORM های بسیار موفق، به شمار می‌رود. در طی تعدادی مقاله قصد آشنایی با این فریم ورک را داریم.

چرا نیاز است تا از یك ORM استفاده شود؟

تهیه قسمت و یا لایه دسترسی به داده‌ها در یک برنامه عموما تا 30 درصد زمان کل تهیه یک محصول را تشکیل می‌دهد. اما باید در نظر داشت که این پروسه‌ی تکراری هیچ کار خارق العاده‌ای نبوده و ارزش افزوده‌ی خاصی را به یک برنامه اضافه نمی‌کند. تقریبا تمام برنامه‌های تجاری نیاز به لایه دسترسی به داده‌ها را دارند. پس چرا ما باید به ازای هر پروژه، این کار تکراری و کسل کننده را بارها و بارها تکرار کنیم؟

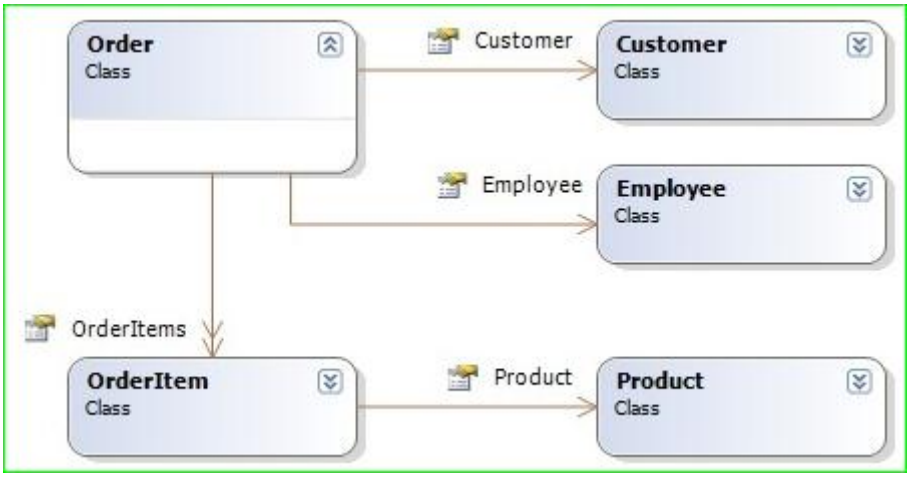
هدف NHibernate ، کاستن این بار از روی شانه‌های یک برنامه نویس است. با کمک این کتابخانه، دیگر رویه ذخیره شده‌ای را نخواهید نوشت. دیگر هیچگاه با ADO.Net سر و کار نخواهید داشت. به این صورت می‌توان عمده وقت خود را صرف قسمت‌های اصلی و طراحی برنامه کرد تا کد نویسی یک لایه تکراری. همچنین عده‌ای از بزرگان اینگونه ابزارها اعتقاد دارند که برنامه نویس‌هایی که لایه دسترسی به داده‌ها را خود طراحی می‌کنند، مشغول کلاهبرداری از مشتری‌های خود هستند! (صرف زمان بیشتر برای تهیه یک محصول و همچنین وجود باگ‌های احتمالی در لایه دسترسی به داده‌های طراحی شده توسط یک برنامه نویس نه چندان حرفه‌ای)

برای مشاهده سایر مزایای استفاده از یک ORM لطفا به مقاله "5 دلیل برای استفاده از یک ابزار ORM" مراجعه نمائید.

در ادامه برای معرفی این کتابخانه یک سیستم ثبت سفارشات را با هم مرور خواهیم کرد.

بررسي مدل سيستم ثبت سفارشات

در این مدل ساده‌ی ما، مشتری‌ها (customers) امکان ثبت سفارشات (orders) را دارند. سفارشات توسط یک کارمند (employee) که مسؤول ثبت آن‌ها است به سیستم وارد می‌شود. هر سفارش می‌تواند شامل یک یا چند (one-to-many) آیتم (order items) باشد و هر آیتم معرف یک محصول (product) است که قرار است توسط یک مشتری (customer) خریداری شود. کلاس دیاگرام این مدل به صورت زیر می‌تواند باشد.



نگاشت مدل

زمانیکه مدل سیستم مشخص شد، اکنون نیاز است تا حالات (داده‌ها) آن‌را در مکانی ذخیره کنیم. عموما اینکار با کمک سیستم‌های مدیریت پایگاه‌های داده مانند SQL Server، Oracle، IBM DB2 و MySql و امثال آن‌ها صورت می‌گیرد. زمانیکه از NHibernate استفاده کنید اهمیتی ندارد که برنامه شما قرار است با چه نوع دیتابیسی کار کند؛ زیرا این کتابخانه اکثر دیتابیس‌های شناخته شده موجود را پشتیبانی می‌کند و برنامه از این لحاظ مستقل از نوع دیتابیس عمل خواهد کرد و اگر نیاز بود روزی بجای اس کیوال سرور از مای اس کیوال استفاده شود، تنها کافی است تنظیمات ابتدایی NHibernate را تغییر دهید (بجای بازنویسی کل برنامه).

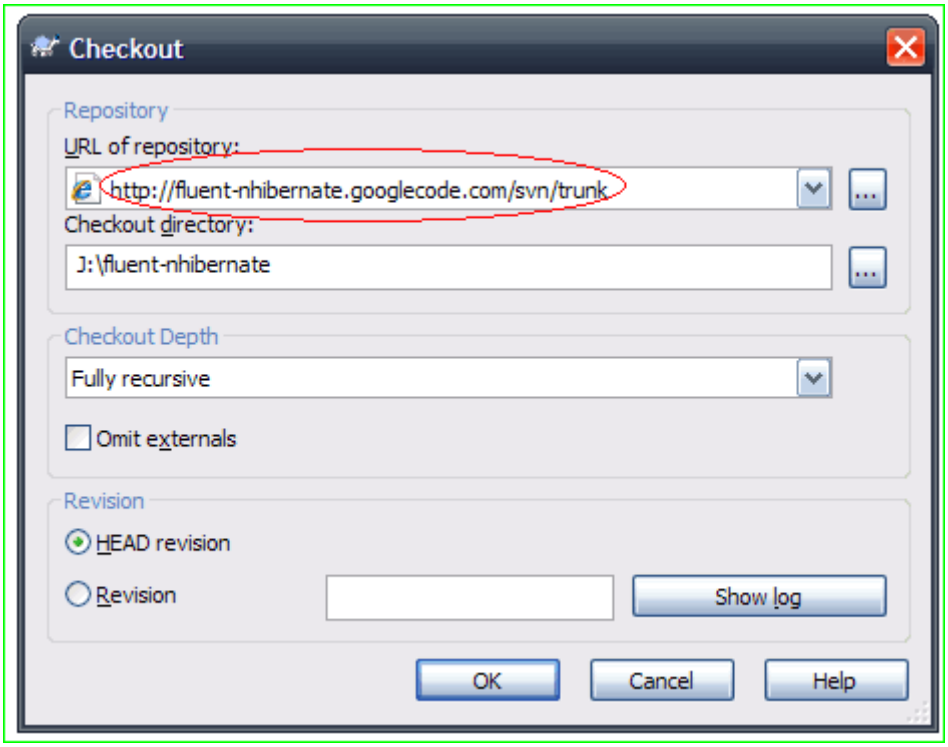
اگر برای ذخیره سازی داده‌ها و حالات سیستم از دیتابیس استفاده کنیم، نیاز است تا اشیاء مدل خود را به جداول دیتابیس نگاشت نمائیم. این نگاشت عموما یک به یک نیست (لزومی ندارد که حتما یک شیء به یک جدول نگاشت شود). در گذشته‌ی نچندان دور کتابخانه‌ی NHibernate، این نگاشت عموما توسط فایل‌های XML ایی به نام hbm صورت می‌گرفت. این روش هنوز هم پشتیبانی شده و توسط بسیاری از برنامه نویس‌ها بکار گرفته می‌شود. روش دیگری که برای تعریف این نگاشت مرسوم است، مزین سازی اشیاء و خواص آن‌ها با یک سری از ویژگی‌ها می‌باشد که فریم ورک برتر این عملیات Castle Active Record نام دارد.

اخیرا کتابخانه‌ی دیگری برای انجام این نگاشت تهیه شده به نام Fluent NHibernate که بسیار مورد توجه علاقمندان به این فریم ورک واقع گردیده است. با کمک کتابخانه‌ی Fluent NHibernate عملیات نگاشت اشیاء به جداول، بجای استفاده از فایل‌های XML، توسط کدهای برنامه صورت خواهند گرفت. این مورد مزایای بسیاری را همانند استفاده از یک زبان برنامه نویسی کامل برای تعریف نگاشت‌ها، بررسی خودکار نوع‌های داده‌ای و حتی امکان تعریف منطقی خاص برای قسمت نگاشت برنامه، به همراه خواهد داشت.

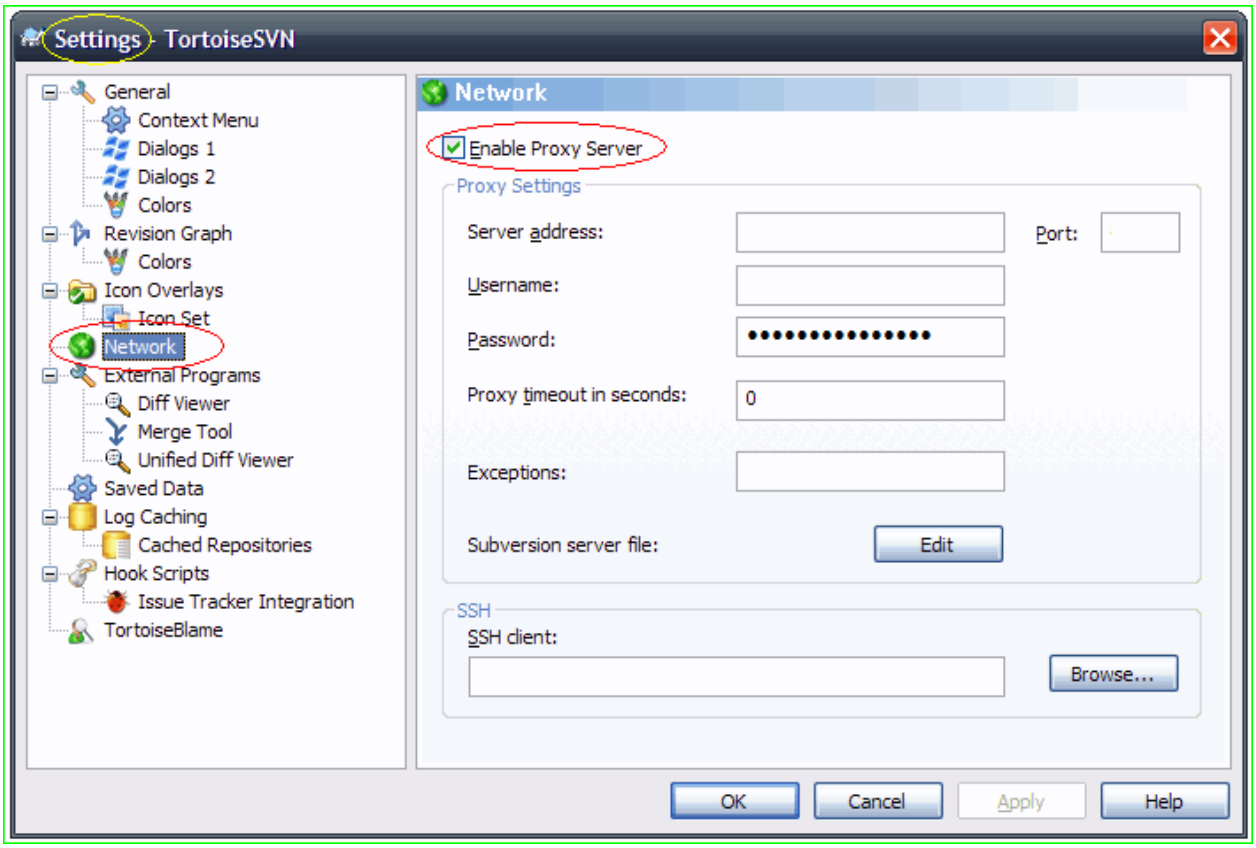
آماده سازي سيستم براي استفاده از NHibernate

در ادامه بجای دریافت پروژه سورس باز NHibernate از سایت سورس فورج، پروژه سورس باز Fluent NHibernate را از سایت گوگل کد دریافت خواهیم کرد که بر فراز کتابخانه‌ی NHibernate بنا شده است و آن‌را کاملا پوشش می‌دهد. سورس این کتابخانه را با checkout مسیر زیر توسط TortoiseSVN می‌توان دریافت کرد.

http://fluent-nhibernate.googlecode.com/svn/trunk



البته احتمالا برای دریافت آن از گوگل کد با توجه به تحریم موجود نیاز به پروکسی خواهد بود. برای تنظیم پروکسی در TortoiseSVN به قسمت تنظیمات آن مطابق تصویر ذیل مراجعه کنید:



همچنین جهت سهولت کار، آخرین نگارش موجود در زمان نگارش این مقاله را از این آدرس نیز می‌توانید دریافت نمائید.

پس از دریافت پروژه، باز کردن فایل solution آن در VS و سپس build کل مجموعه، اگر به پوشه‌های آن مراجعه نمائید، فایل‌های زیر قابل مشاهده هستند:

Nhibernate.dll: اسمبلی فریم ورک NHibernate است.

NHibernate.Linq.dll: اسمبلی پرویدر LINQ to NHibernate می‌باشد.

FluentNHibernate.dll: اسمبلی فریم ورک Fluent NHibernate است.

Iesi.Collections.dll: یک سری مجموعه‌های ویژه مورد استفاده NHibernate را ارائه می‌دهد.

Log4net.dll: فریم ورک لاگ کردن اطلاعات NHibernate می‌باشد. (این فریم ورک نیز جهت عملیات logging بسیار معروف و محبوب است)

Castle.Core.dll: کتابخانه پایه Castle.DynamicProxy2.dll است.

Castle.DynamicProxy2.dll: جهت اعمال lazy loading در فریم ورک NHibernate بکار می‌رود.

System.Data.SQLite.dll: پرویدر دیتابیس SQLite است.

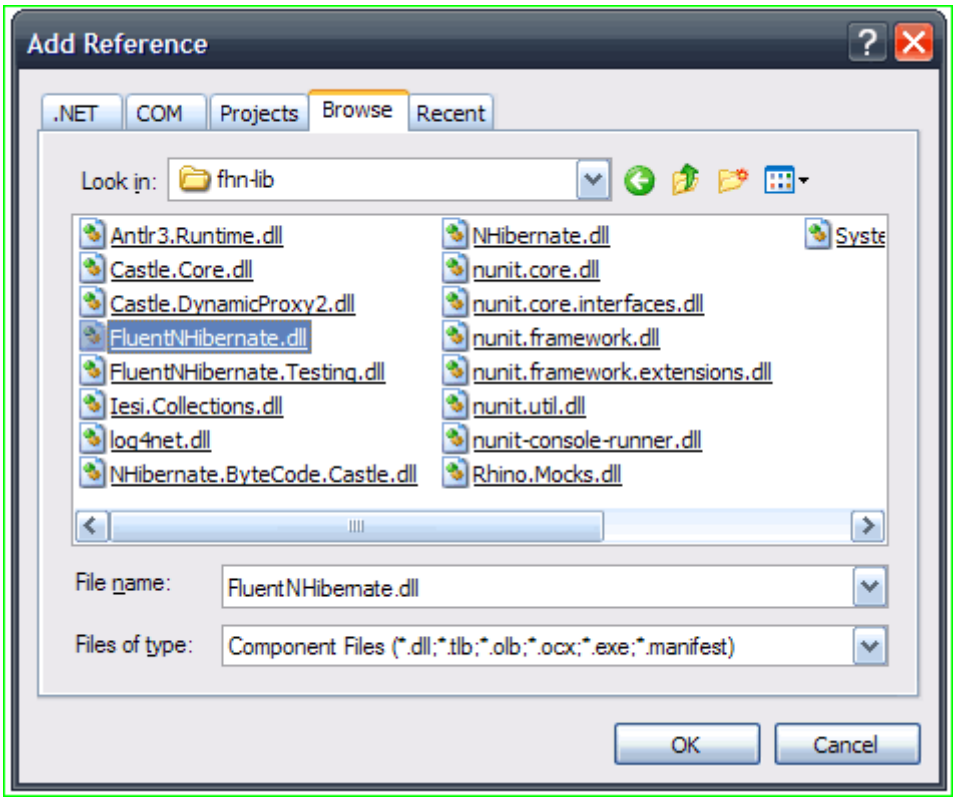
Nunit.framework.dll: نیز یکی از فریم ورک‌های بسیار محبوب آزمون واحد در دات نت فریم ورک است.

برای سادگی مراجعات بعدی، این فایل‌ها را یافته و در پوشه‌ای به نام lib کپی نمائید.

برپايي يك پروژه جديد

پس از دریافت NHibernate Fluent**، یک پروژه** Class Library **جدید را در** VS.Net **آغاز کنید (برای مثال به نام** NHSample1**). سپس یک پروژه دیگر را نیز از نوع** Class Library **به نام** UnitTests **به این** solution **ایجاد شده جهت انجام آزمون‌های واحد برنامه اضافه نمائید.**

اکنون به پروژه NHSample1**، ارجاع هایی را به فایل‌های** FluentNHibernate.dll **و سپس** NHibernate.dll **در که پوشه** lib **ایی که در قسمت قبل ساختیم، قرار دارند، اضافه نمائید.**



در ادامه یک پوشه جدید به پروژه NHSample1 **به نام** Domain **اضافه کنید. سپس به این پوشه، کلاس** Customer **را اضافه نمائید:**

```
namespace NHSample1.Domain
{
    public class Customer
    {
        public int Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string AddressLine1 { get; set; }
        public string AddressLine2 { get; set; }
        public string PostalCode { get; set; }
        public string City { get; set; }
        public string CountryCode { get; set; }
    }
}
```

اکنون نوبت تعریف نگاشت این شیء است. این کلاس باید از کلاس پایه ClassMap **مشتق شود. سپس نگاشت‌ها در سازنده‌ی این کلاس باید تعریف گردند.**

```
using FluentNHibernate.Mapping;

namespace NHSample1.Domain
{
    class CustomerMapping : ClassMap<Customer>
    {
    }
}
```

همانطور که ملاحظه می‌کنید، نوع این کلاس Generic ، همان کلاسی است که قصد داریم نگاشت مرتبط با آن را تهیه نمائیم. در ادامه تعریف کامل این کلاس نگاشت را در نظر بگیرید:

```
using FluentNHibernate.Mapping;

namespace NHSample1.Domain
{
    class CustomerMapping : ClassMap<Customer>
    {
        public CustomerMapping()
        {
            Not.LazyLoad();
            Id(c => c.Id).GeneratedBy.HiLo("1000");
            Map(c => c.FirstName).Not.Nullable().Length(50);
            Map(c => c.LastName).Not.Nullable().Length(50);
            Map(c => c.AddressLine1).Not.Nullable().Length(50);
            Map(c => c.AddressLine2).Length(50);
            Map(c => c.PostalCode).Not.Nullable().Length(10);
            Map(c => c.City).Not.Nullable().Length(50);
            Map(c => c.CountryCode).Not.Nullable().Length(2);
        }
    }
}
```

به صورت پیش فرض نگاشت‌های Fluent NHibernate از نوع lazy load هستند که در اینجا عکس آن در نظر گرفته شده است. سپس وضعیت نگاشت تک تک خواص کلاس Customer را مشخص می‌کنیم. توسط Id(c => c.Id).GeneratedBy.HiLo به سیستم اعلام خواهیم کرد که فیلد Id از نوع identity است که از 1000 شروع خواهد شد. مابقی موارد هم بسیار واضح هستند. تمامی خواص کلاس Customer ذکر شده، نال را نمی‌پذیرند (منهای AddressLine2) و طول آن‌ها نیز مشخص گردیده است.
با کمک Fluent NHibernate ، بحث بررسی نوع‌های داده‌ای و همچنین یکی بودن موارد مطرح شده در نگاشت با کلاس اصلی Customer به سادگی توسط کامپایلر بررسی شده و خطاهای آتی کاهش خواهند یافت.

برای آشنایی بیشتر با lambda expressions می‌توان به مقاله زیر مراجعه کرد:

Step-by-step Introduction to Delegates and Lambda Expressions

قسمت دوم

آزمون واحد کلاس نگاشت تهیه شده

در مورد آشنایی با آزمون‌های واحد لطفا به برچسب مربوطه در سمت راست سایت مراجعه بفرمائید. همچنین در مورد اینکه چرا به این نوع API کلمه Fluent اطلاق می‌شود، می‌توان به تعریف آن جهت مطالعه بیشتر مراجعه نمود.

در این قسمت قصد داریم برای بررسی وضعیت کلاس نگاشت تهیه شده یک آزمون واحد تهیه کنیم. برای این منظور ارجاعی را به اسمبلی nunit.framework.dll به پروژه UnitTests که در ابتدای کار به solution جاری در VS.Net افزوده بودیم، اضافه نمائید (همچنین ارجاع‌هایی به اسمبلی‌های پروژه NHSample1، FluentNHibernate، System.Data.SQLite، NHibernate.ByteCode.Castle و Nhibernate نیز نیاز هستند). تمام اسمبلی‌های این فریم ورک‌ها از پروژه FluentNHibernate قابل استخراج هستند.

سپس سه کلاس زیر را به پروژه آزمون واحد اضافه خواهیم کرد.

کلاس TestModel: (جهت مشخص سازی محل دریافت اطلاعات نگاشت)

```
using FluentNHibernate;
using NHSample1.Domain;

namespace UnitTests
{
    public class TestModel : PersistenceModel
    {
        public TestModel()
        {
            AddMappingsFromAssembly(typeof(CustomerMapping).Assembly);
        }
    }
}
```

کلاس FixtureBase: (جهت ایجاد سشن NHibernate در ابتدای آزمون واحد و سپس پاکسازی اشیاء در پایان کار)

```
using NUnit.Framework;
using NHibernate;
using FluentNHibernate;
using FluentNHibernate.Cfg;
using FluentNHibernate.Cfg.Db;

namespace UnitTests
{
    public class FixtureBase
    {
        protected SessionSource SessionSource { get; set; }
        protected ISession Session { get; private set; }

        [SetUp]
        public void SetupContext()
        {
            var cfg =
Fluently.Configure().Database(SQLiteConfiguration.Standard.InMemory);

            SessionSource = new SessionSource(
                                cfg.BuildConfiguration().Properties,
```

```
                new TestModel());

            Session = SessionSource.CreateSession();
            SessionSource.BuildSchema(Session);
        }

        [TearDown]
        public void TearDownContext()
        {
            Session.Close();
            Session.Dispose();
        }
    }
}
```

و کلاس CustomerMapping_Fixture.cs **:(جهت بررسی صحت نگاشت تهیه شده با کمک دو کلاس قبل)**

```
using NUnit.Framework;
using FluentNHibernate.Testing;
using NHSample1.Domain;

namespace UnitTests
{
    [TestFixture]
    public class CustomerMapping_Fixture : FixtureBase
    {
        [Test]
        public void can_correctly_map_customer()
        {
            new PersistenceSpecification<Customer>(Session)
                    .CheckProperty(c => c.Id, 1001)
                    .CheckProperty(c => c.FirstName, "Vahid")
                    .CheckProperty(c => c.LastName, "Nasiri")
                    .CheckProperty(c => c.AddressLine1, "Addr1")
                    .CheckProperty(c => c.AddressLine2, "Addr2")
                    .CheckProperty(c => c.PostalCode, "1234")
                    .CheckProperty(c => c.City, "Tehran")
                    .CheckProperty(c => c.CountryCode, "IR")
                    .VerifyTheMappings();
        }
    }
}
```

توضیحات:

اکنون به عنوان یک برنامه نویس متعهد نیاز است تا کار صورت گرفته در قسمت قبل را آزمایش کنیم.

کار بررسی صحت نگاشت تعریف شده در قسمت قبل توسط کلاس استاندارد PersistenceSpecification **فریم ورک** FluentNHibernate **انجام خواهد شد (در کلاس** CustomerMapping_Fixture**). این کلاس برای انجام عملیات آزمون واحد نیاز به کلاس پایه دیگری به نام** FixtureBase **دارد که در آن کار ایجاد سشن** NHibernate **(در قسمت استاندارد** SetUp **آزمون واحد) و سپس آزاد سازی آن را در هنگام خاتمه کار ، انجام می‌دهد (در قسمت** TearDown **آزمون واحد).**

این ویژگی‌ها که در مباحث آزمون واحد نیز به آن‌ها اشاره شده است، سبب اجرای متدهایی پیش از اجرا و بررسی هر آزمون واحد و سپس آزاد سازی خودکار منابع خواهند شد.

برای ایجاد یک سشن NHibernate **نیاز است تا نوع دیتابیس و همچنین رشته اتصالی به آن (کانکشن استرینگ) مشخص شوند. فریم ورک** Fluent NHibernate **با ایجاد کلاس‌های کمکی برای این امر، به شدت سبب ساده سازی انجام آن شده است. در این مثال، نوع دیتابیس به** SQLite **و در حالت دیتابیس در حافظه (**in memory**)، تنظیم شده است (برای انجام امور آزمون واحد با سرعت بالا).**

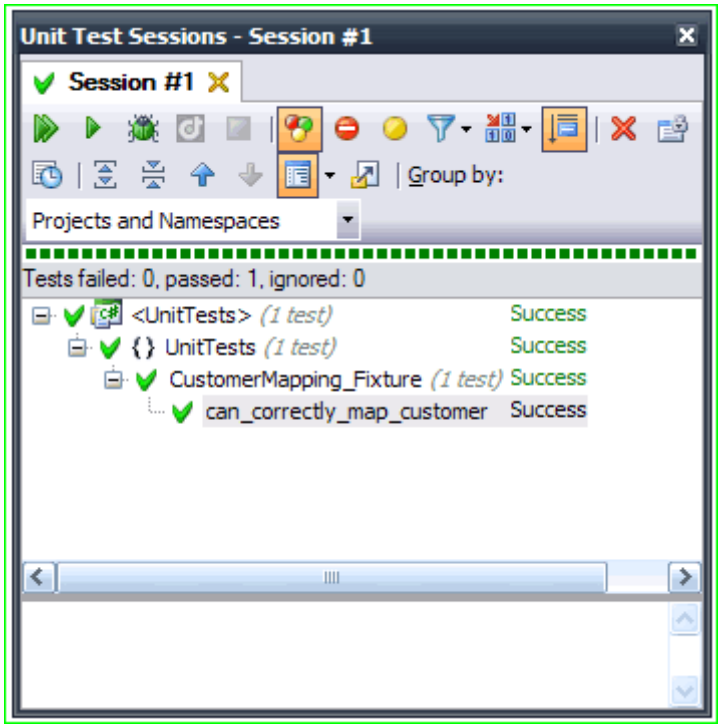
جهت اجرای هر دستوری در NHibernate نیاز به یک سشن می‌باشد. برای تعریف شیء سشن، نه تنها نیاز به مشخص سازی نوع و حالت دیتابیس مورد استفاده داریم، بلکه نیاز است تا وهله‌ای از کلاس استاندارد PersistanceModel را نیز جهت مشخص سازی کلاس نگاشت مورد استفاده مشخص نمائیم. برای این منظور کلاس TestModel فوق تعریف شده است تا این نگاشت را از اسمبلی مربوطه بخواند و مورد استفاده قرار دهد (بر پایی اولیه این مراحل شاید در ابتدای امر کمی زمانبر باشد اما در نهایت یک پروسه استاندارد است). توسط این کلاس به سیستم اعلام خواهیم کرد که اطلاعات نگاشت را باید از کدام کلاس دریافت کند.

تا اینجای کار شیء SessionSource را با معرفی نوع دیتابیس و همچنین محل دریافت اطلاعات نگاشت اشیاء معرفی کردیم. در دو سطر بعدی متد SetupContext کلاس FixtureBase ، ابتدا یک سشن را از این منبع سشن تهیه می‌کنیم. شیء منبع سشن در این فریم ورک در حقیقت یک factory object است (الگوهای طراحی برنامه نویسی شیءگرا) که امکان دسترسی به انواع و اقسام دیتابیس‌ها را فراهم می‌سازد. برای مثال اگر روزی نیاز بود از دیتابیس اس کیوال سرور استفاده شود، می‌توان از کلاس MsSqlConfiguration بجای SQLiteConfiguration استفاده کرد و همینطور الی آخر.

در ادامه توسط شیء SessionSource کار ساخت database schema را نیز به صورت پویا انجام خواهیم داد. بله، همانطور که متوجه شده‌اید، کار ساخت database schema نیز به صورت پویا توسط فریم ورک NHibernate با توجه به اطلاعات کلاس‌های نگاشت، صورت خواهد گرفت.

این مراحل، نحوه ایجاد و بر پایی یک آزمایشگاه آزمون واحد فریم ورک Fluent NHibernate را مشخص ساخته و در پروژه‌های شما می‌توانند به کرات مورد استفاده قرار گیرند.

در ادامه اگر آزمون واحد را اجرا نمائیم (متد can_correctly_map_customer در کلاس CustomerMapping_Fixture)، نتیجه باید شبیه به شکل زیر باشد:

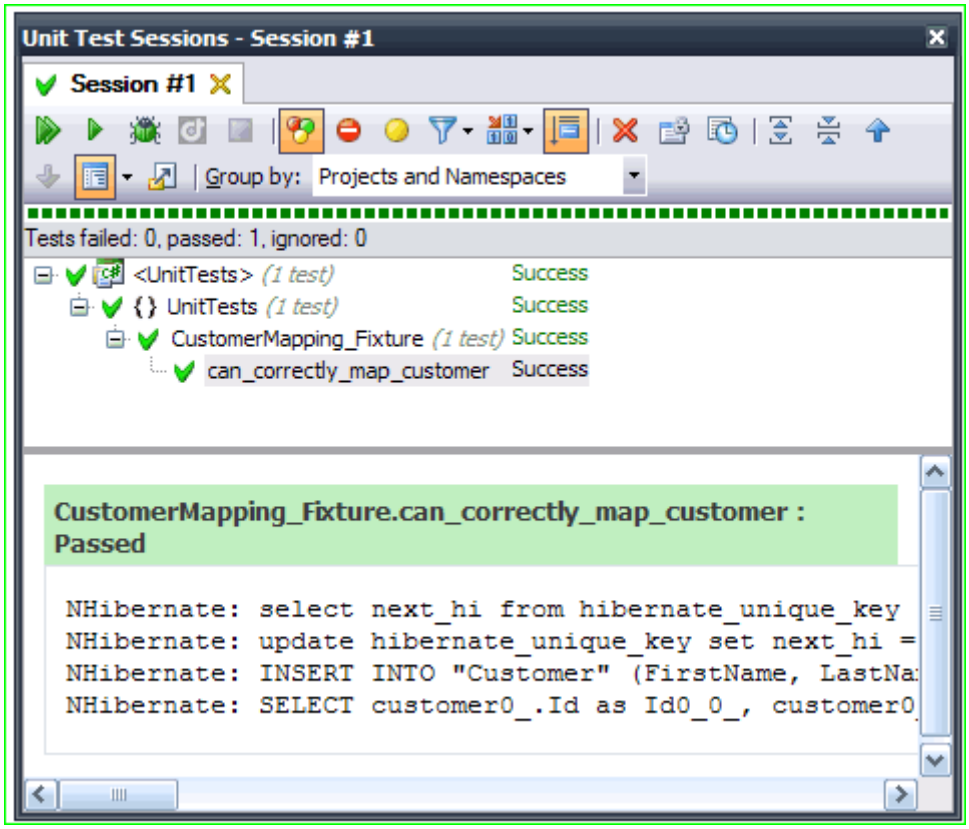


توسط متد CheckProperty کلاس PersistenceSpecification ، امکان بررسی نگاشت تهیه شده میسر است. اولین پارامتر آن، یک lambda expression خاصیت مورد نظر جهت بررسی است و دومین آرگومان آن، مقداری است که در حین آزمون به خاصیت تعریف شده انتساب داده می‌شود.

نکته:

شاید سؤال بپرسید که در تابع `can_correctly_map_customer` **عملا چه اتفاقاتی رخ داده است؟ برای بررسی آن در متد** `SetupContext` **کلاس** `FixtureBase` **، اولین سطر آن‌را به صورت زیر تغییر دهید تا عبارات** `SQL` **نهایی تولید شده را نیز بتوانیم در حین عملیات تست مشاهده نمائیم:**

```
var cfg =
Fluently.Configure().Database(SQLiteConfiguration.Standard.ShowSql().InMemory);
```



مطابق متد تست فوق، عبارات تولید شده به شرح زیر هستند:

```
NHibernate: select next_hi from hibernate_unique_key
NHibernate: update hibernate_unique_key set next_hi = @p0 where next_hi =
@p1;@p0 = 2, @p1 = 1
NHibernate: INSERT INTO "Customer" (FirstName, LastName, AddressLine1,
AddressLine2, PostalCode, City, CountryCode, Id) VALUES (@p0, @p1, @p2, @p3,
@p4, @p5, @p6, @p7);@p0 = 'Vahid', @p1 = 'Nasiri', @p2 = 'Addr1', @p3 =
'Addr2', @p4 = '1234', @p5 = 'Tehran', @p6 = 'IR', @p7 = 1001
NHibernate: SELECT customer0_.Id as Id0_0_, customer0_.FirstName as
FirstName0_0_, customer0_.LastName as LastName0_0_, customer0_.AddressLine1 as
AddressL4_0_0_, customer0_.AddressLine2 as AddressL5_0_0_,
customer0_.PostalCode as PostalCode0_0_, customer0_.City as City0_0_,
customer0_.CountryCode as CountryC8_0_0_ FROM "Customer" customer0_ WHERE
customer0_.Id=@p0;@p0 = 1001
```

نکته جالب این عبارات، استفاده از کوئری‌های پارامتری است به صورت پیش فرض که در نهایت سبب بالا رفتن امنیت بیشتر برنامه (یکی از راه‌های جلوگیری از تزریق اس کیوال در `ADO.Net` **که در نهایت توسط تمامی این فریم ورک‌ها در پشت صحنه مورد استفاده قرار خواهند گرفت) و همچنین سبب بکار افتادن سیستم‌های کش دیتابیس‌های پیشرفته مانند اس کیوال سرور می‌شوند (**`execution plan` **کوئری‌های پارامتری در اس کیوال سرور جهت بالا رفتن کارآیی سیستم کش می‌شوند و اهمیتی هم ندارد که حتما رویه ذخیره شده باشند یا خیر).**

قسمت سوم

در ادامه، تعاریف سایر موجودیت‌های سیستم ثبت سفارشات و نگاشت آن‌ها را بررسی خواهیم کرد.

کلاس `Product` **تعریف شده در فایل جدید** `Product.cs` **در پوشه** `domain` **برنامه:**

```
namespace NHSample1.Domain
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public decimal UnitPrice { get; set; }
        public bool Discontinued { get; set; }
    }
}
```

کلاس `ProductMapping` **تعریف شده در فایل جدید** `ProductMapping.cs` **(توصیه شده است که به ازای هر کلاس یک فایل جداگانه در نظر گرفته شود)، در پوشه** `Mappings` **برنامه:**

```
using FluentNHibernate.Mapping;
using NHSample1.Domain;

namespace NHSample1.Mappings
{
    public class ProductMapping : ClassMap<Product>
    {
        public ProductMapping()
        {
            Not.LazyLoad();
            Id(p => p.Id).GeneratedBy.HiLo("1000");
            Map(p => p.Name).Length(50).Not.Nullable();
            Map(p => p.UnitPrice).Not.Nullable();
            Map(p => p.Discontinued).Not.Nullable();
        }
    }
}
```

همانطور که ملاحظه می‌کنید، روش تعریف آن‌ها همانند شیء `Customer` **است که در قسمت‌های قبل بررسی شد و نکته جدیدی ندارد.**

آزمون واحد بررسی این نگاشت نیز همانند مثال قبلی است.

کلاس `ProductMapping_Fixture` **را در فایل جدید** `ProductMapping_Fixture.cs` **به پروژه** `UnitTests` **خود (که ارجاعات آن‌را در قسمت قبل مشخص کردیم) خواهیم افزود:**

```
using NUnit.Framework;
using FluentNHibernate.Testing;
using NHSample1.Domain;

namespace UnitTests
{
    [TestFixture]
    public class ProductMapping_Fixture : FixtureBase
    {
        [Test]
        public void can_correctly_map_product()
        {
            new PersistenceSpecification<Product>(Session)
                    .CheckProperty(p => p.Id, 1001)
```

```
                    .CheckProperty(p => p.Name, "Apples")
                    .CheckProperty(p => p.UnitPrice, 10.45m)
                    .CheckProperty(p => p.Discontinued, true)
                    .VerifyTheMappings();
        }
    }
}
```

و پس از اجرای این آزمون واحد، عبارات SQL ایی که به صورت خودکار توسط این ORM جهت بررسی عملیات نگاشت صورت خواهند گرفت به صورت زیر می‌باشند:

```
ProductMapping_Fixture.can_correctly_map_product : Passed
NHibernate: select next_hi from hibernate_unique_key
NHibernate: update hibernate_unique_key set next_hi = @p0 where next_hi =
@p1;@p0 = 2, @p1 = 1
NHibernate: INSERT INTO "Product" (Name, UnitPrice, Discontinued, Id) VALUES
(@p0, @p1, @p2, @p3);@p0 = 'Apples', @p1 = 10.45, @p2 = True, @p3 = 1001
NHibernate: SELECT product0_.Id as Id1_0_, product0_.Name as Name1_0_,
product0_.UnitPrice as UnitPrice1_0_, product0_.Discontinued as Disconti4_1_0_
FROM "Product" product0_ WHERE product0_.Id=@p0;@p0 = 1001
```

در ادامه تعریف کلاس کارمند، نگاشت و آزمون واحد آن به صورت زیر خواهند بود:

```
using System;
namespace NHSample1.Domain
{
    public class Employee
    {
        public int Id { set; get; }
        public string LastName { get; set; }
        public string FirstName { get; set; }
    }
}
```

```
using NHSample1.Domain;
using FluentNHibernate.Mapping;

namespace NHSample1.Mappings
{
   public class EmployeeMapping : ClassMap<Employee>
   {
       public EmployeeMapping()
       {
           Not.LazyLoad();
           Id(e => e.Id).GeneratedBy.Assigned();
           Map(e => e.LastName).Length(50);
           Map(e => e.FirstName).Length(50);
       }
   }
}
```

```
using NUnit.Framework;
using NHSample1.Domain;
using FluentNHibernate.Testing;

namespace UnitTests
{
   [TestFixture]
   public class EmployeeMapping_Fixture : FixtureBase
```

```
    {
        [Test]
        public void can_correctly_map_employee()
        {
            new PersistenceSpecification<Employee>(Session)
                    .CheckProperty(p => p.Id, 1001)
                    .CheckProperty(p => p.FirstName, "name1")
                    .CheckProperty(p => p.LastName, "lname1")
                    .VerifyTheMappings();
        }
    }
}
```

خروجی SQL حاصل از موفقیت آزمون واحد آن:

```
NHibernate: select next_hi from hibernate_unique_key
NHibernate: update hibernate_unique_key set next_hi = @p0 where next_hi =
@p1;@p0 = 2, @p1 = 1
NHibernate: INSERT INTO "Employee" (LastName, FirstName, Id) VALUES (@p0, @p1,
@p2);@p0 = 'lname1', @p1 = 'name1', @p2 = 1001
NHibernate: SELECT employee0_.Id as Id4_0_, employee0_.LastName as
LastName4_0_, employee0_.FirstName as FirstName4_0_ FROM "Employee" employee0_
WHERE employee0_.Id=@p0;@p0 = 1001
```

همانطور که ملاحظه می‌کنید، این آزمون‌های واحد 4 مرحله را در یک سطر انجام می‌دهند:

الف) ایجاد یک وهله از کلاس Employee

ب) ثبت اطلاعات کارمند در دیتابیس

ج) دریافت اطلاعات کارمند در وهله‌ای جدید از شیء Employee

د) و در پایان بررسی می‌کند که آیا شیء جدید ایجاد شده با شیء اولیه مطابقت دارد یا خیر

اکنون در ادامه پیاده سازی سیستم ثبت سفارشات، به قسمت جالب این مدل می‌رسیم. قسمتی که در آن ارتباطات اشیاء و روابط one-to-many تعریف خواهند شد. تعاریف کلاس‌های OrderItem و OrderItemMapping را به صورت زیر در نظر بگیرید:

کلاس OrderItem تعریف شده در فایل جدید OrderItem.cs واقع شده در پوشه domain پروژه:

که در آن هر سفارش (order) دقیقا از یک محصول (product) تشکیل می‌شود و هر محصول می‌تواند در سفارشات متعدد و مختلفی درخواست شود.

```
namespace NHSample1.Domain
{
    public class OrderItem
    {
        public int Id { get; set; }
        public int Quantity { get; set; }
        public Product Product { get; set; }
    }
}
```

کلاس OrderItemMapping تعریف شده در فایل جدید OrderItemMapping.cs:

```
using FluentNHibernate.Mapping;
using NHSample1.Domain;

namespace NHSample1.Mappings
{
    public class OrderItemMapping : ClassMap<OrderItem>
```

```
    {
        public OrderItemMapping()
        {
            Not.LazyLoad();
            Id(oi => oi.Id).GeneratedBy.Assigned();
            Map(oi => oi.Quantity).Not.Nullable();
            References(oi => oi.Product).Not.Nullable();
        }
    }
}
```

نکته جدیدی که در این کلاس نگاشت مطرح شده است، واژه کلیدی References می‌باشد که جهت بیان این ارجاعات و وابستگی‌ها بکار می‌رود. این ارجاع بیانگر یک رابطه many-to-one بین سفارشات و محصولات است. همچنین در ادامه آن Not.Nullable ذکر شده است تا این ارجاع را اجباری نمائید (در غیر اینصورت سفارش غیر معتبر خواهد بود).

نکته‌ی دیگر مهم آن این مورد است که Id در اینجا به صورت یک کلید تعریف نشده است. یک آیتم سفارش داده شده، موجودیت به حساب نیامده و فقط یک شیء مقداری (value object) است و به خودی خود امکان وجود ندارد. هر وهله از آن تنها توسط یک سفارش قابل تعریف است. بنابراین id در اینجا فقط به عنوان یک index می‌تواند مورد استفاده قرار گیرد و فقط توسط شیء Order زمانیکه یک OrderItem به آن اضافه می‌شود، مقدار دهی خواهد شد.

اگر برای این نگاشت نیز آزمون واحد تهیه کنیم، به صورت زیر خواهد بود:

```
using NUnit.Framework;
using NHSample1.Domain;
using FluentNHibernate.Testing;

namespace UnitTests
{
    [TestFixture]
    public class OrderItemMapping_Fixture : FixtureBase
    {
        [Test]
        public void can_correctly_map_order_item()
        {
            var product = new Product
            {
                Name = "Apples",
                UnitPrice = 4.5m,
                Discontinued = true
            };

            new PersistenceSpecification<OrderItem>(Session)
                        .CheckProperty(p => p.Id, 1)
                        .CheckProperty(p => p.Quantity, 5)
                        .CheckReference(p => p.Product, product)
                        .VerifyTheMappings();
        }
    }
}
```

مشکل! این آزمون واحد با شکست مواجه خواهد شد، زیرا هنوز مشخص نکرده‌ایم که دو شیء Product را که در قسمت CheckReference فوق برای این منظور معرفی کرده‌ایم، چگونه باید با هم مقایسه کرد. در مورد مقایسه نوع‌های اولیه و اصلی مانند int و string و امثال آن مشکلی نیست، اما باید منطق مقایسه سایر اشیاء سفارشی خود را با پیاده سازی اینترفیس IEqualityComparer دقیقا مشخص سازیم:

```
using System.Collections;
using NHSample1.Domain;
```

```
namespace UnitTests
{
    public class CustomEqualityComparer : IEqualityComparer
    {
        public bool Equals(object x, object y)
        {
            if (ReferenceEquals(x, y)) return true;
            if (x == null || y == null) return false;

            if (x is Product && y is Product)
                return (x as Product).Id == (y as Product).Id;

            if (x is Customer && y is Customer)
                return (x as Customer).Id == (y as Customer).Id;

            if (x is Employee && y is Employee)
                return (x as Employee).Id == (y as Employee).Id;

            if (x is OrderItem && y is OrderItem)
                return (x as OrderItem).Id == (y as OrderItem).Id;


            return x.Equals(y);
        }

        public int GetHashCode(object obj)
        {
            //دیگر وقتی شاید
            return obj.GetHashCode();
        }
    }
}
```

در اینجا فقط Id این اشیاء با هم مقایسه شده است. در صورت نیاز تمامی خاصیت‌های این اشیاء را نیز می‌توان با هم مقایسه کرد (یک سری از اشیاء بکار گرفته شده در این کلاس در ادامه بحث معرفی خواهند شد).

سپس برای بکار گیری این کلاس جدید، سطر مربوط به استفاده از PersistenceSpecification به صورت زیر تغییر خواهد کرد:

```
new PersistenceSpecification<OrderItem>(Session, new CustomEqualityComparer())
```

پس از این تغییرات و مشخص سازی نحوه‌ی مقایسه دو شیء سفارشی، آزمون واحد ما پاس شده و خروجی SQL تولید شده آن به صورت زیر می‌باشد:

```
NHibernate: select next_hi from hibernate_unique_key
NHibernate: update hibernate_unique_key set next_hi = @p0 where next_hi =
@p1;@p0 = 2, @p1 = 1
NHibernate: INSERT INTO "Product" (Name, UnitPrice, Discontinued, Id) VALUES
(@p0, @p1, @p2, @p3);@p0 = 'Apples', @p1 = 4.5, @p2 = True, @p3 = 1001
NHibernate: INSERT INTO "OrderItem" (Quantity, Product_id, Id) VALUES (@p0,
@p1, @p2);@p0 = 5, @p1 = 1001, @p2 = 1
NHibernate: SELECT orderitem0_.Id as Id0_1_, orderitem0_.Quantity as
Quantity0_1_, orderitem0_.Product_id as Product3_0_1_, product1_.Id as Id3_0_,
product1_.Name as Name3_0_, product1_.UnitPrice as UnitPrice3_0_,
product1_.Discontinued as Disconti4_3_0_ FROM "OrderItem" orderitem0_ inner
join "Product" product1_ on orderitem0_.Product_id=product1_.Id WHERE
orderitem0_.Id=@p0;@p0 = 1
```

قسمت پایانی کار تعاریف کلاس‌های نگاشت، مربوط به کلاس Order است که در ادامه بررسی خواهد شد.

```
using System;
using System.Collections.Generic;

namespace NHSample1.Domain
{
    public class Order
    {
        public int Id { set; get; }
        public DateTime OrderDate { get; set; }
        public Employee Employee { get; set; }
        public Customer Customer { get; set; }
        public IList<OrderItem> OrderItems { get; set; }
    }
}
```

نکته‌ی مهمی که در این کلاس وجود دارد استفاده از IList جهت معرفی مجموعه‌ای از آیتم‌های سفارشی است (بجای List و یا IEnumerable که در صورت استفاده خطای cast exception type در حین نگاشت حاصل می‌شد).

```
using NHSample1.Domain;
using FluentNHibernate.Mapping;

namespace NHSample1.Mappings
{
    public class OrderMapping : ClassMap<Order>
    {
        public OrderMapping()
        {
            Not.LazyLoad();
            Id(o => o.Id).GeneratedBy.GuidComb();
            Map(o => o.OrderDate).Not.Nullable();
            References(o => o.Employee).Not.Nullable();
            References(o => o.Customer).Not.Nullable();
            HasMany(o => o.OrderItems)
                .AsList(index => index.Column("ListIndex").Type<int>());
        }
    }
}
```

در تعاریف نگاشت این کلاس نیز دو ارجاع به اشیاء کارمند و مشتری وجود دارد که با References مشخص شده‌اند.
قسمت جدید آن HasMany است که جهت تعریف رابطه one-to-many بکار گرفته شده است. یک سفارش رابطه many-to-one با یک مشتری و همچنین کارمندی که این رکورد را ثبت می‌کند، دارد. در اینجا مجموعه آیتم‌های یک سفارش به صورت یک لیست بازگشت داده می‌شود و ایندکس آن به ستونی به نام ListIndex در یک جدول دیتابیس نگاشت خواهد شد. نوع این ستون، int می‌باشد.

```
using System;
using System.Collections.Generic;
using NUnit.Framework;
using NHSample1.Domain;
using FluentNHibernate.Testing;

namespace UnitTests
{
    [TestFixture]
    public class OrderMapping_Fixture : FixtureBase
    {
        [Test]
        public void can_correctly_map_an_order()
```

```
{
    {
        var product1 =
            new Product
            {
                Name = "Apples",
                UnitPrice = 4.5m,
                Discontinued = true
            };
        var product2 =
            new Product
            {
                Name = "Pears",
                UnitPrice = 3.5m,
                Discontinued = false
            };

        Session.Save(product1);
        Session.Save(product2);

        var items = new List<OrderItem>
                        {
                            new OrderItem
                            {
                                Id = 1,
                                Quantity = 100,
                                Product = product1
                            },
                            new OrderItem
                            {
                                Id = 2,
                                Quantity = 200,
                                Product = product2
                            }
                        };

        var customer = new Customer
        {
            FirstName = "Vahid",
            LastName = "Nasiri",
            AddressLine1 = "Addr1",
            AddressLine2 = "Addr2",
            PostalCode = "1234",
            City = "Tehran",
            CountryCode = "IR"
        };

        var employee =
            new Employee
            {
                FirstName = "name1",
                LastName = "lname1"
            };



        var order = new Order
        {
            Customer = customer,
```

```
                    Employee = employee,
                    OrderDate = DateTime.Today,
                    OrderItems = items
                };

                new PersistenceSpecification<Order>(Session, new
CustomEqualityComparer())
                    .CheckProperty(o => o.OrderDate, order.OrderDate)
                    .CheckReference(o => o.Customer, order.Customer)
                    .CheckReference(o => o.Employee, order.Employee)
                    .CheckList(o => o.OrderItems, order.OrderItems)
                    .VerifyTheMappings();
            }
        }
    }
}
```

همانطور که ملاحظه می‌کنید در این متد آزمون واحد، نیاز به مشخص سازی منطق مقایسه اشیاء سفارش، مشتری و آیتم‌های سفارش داده شده نیز وجود دارد که پیشتر در کلاس `CustomEqualityComparer` معرفی شدند؛ درغیر اینصورت این آزمون واحد با شکست مواجه می‌شد.

متد آزمون واحد فوق کمی طولانی است؛ زیرا در آن باید تعاریف انواع و اقسام اشیاء مورد استفاده را مشخص نمود (و ارزش کار نیز دقیقا در همینجا مشخص می‌شود که بجای `SQL` نوشتن، با اشیایی که توسط کامپایلر تحت نظر هستند سر و کار داریم).

تنها نکته جدید آن استفاده از `CheckList` برای بررسی `IList` تعریف شده در قسمت قبل است.

خروجی `SQL` این آزمون واحد پس از اجرا و موفقیت آن به صورت زیر است:

```
NHibernate: select next_hi from hibernate_unique_key
NHibernate: update hibernate_unique_key set next_hi = @p0 where next_hi =
@p1;@p0 = 2, @p1 = 1
NHibernate: INSERT INTO "Product" (Name, UnitPrice, Discontinued, Id) VALUES
(@p0, @p1, @p2, @p3);@p0 = 'Apples', @p1 = 4.5, @p2 = True, @p3 = 1001
NHibernate: INSERT INTO "OrderItem" (Quantity, Product_id, Id) VALUES (@p0,
@p1, @p2);@p0 = 5, @p1 = 1001, @p2 = 1
NHibernate: SELECT orderitem0_.Id as Id0_1_, orderitem0_.Quantity as
Quantity0_1_, orderitem0_.Product_id as Product3_0_1_, product1_.Id as Id3_0_,
product1_.Name as Name3_0_, product1_.UnitPrice as UnitPrice3_0_,
product1_.Discontinued as Disconti4_3_0_ FROM "OrderItem" orderitem0_ inner
join "Product" product1_ on orderitem0_.Product_id=product1_.Id WHERE
orderitem0_.Id=@p0;@p0 = 1
```

تا اینجای کار تعاریف اشیاء ، نگاشت آن‌ها و همچنین بررسی صحت این نگاشت‌ها به پایان می‌رسد.

نکته:

دیتابیس برنامه را جهت آزمون‌های واحد برنامه، از نوع `SQLite` ساخته شده در حافظه مشخص کردیم. اگر علاقمند باشید که `database schema` تولید شده توسط `NHibernate` را مشاهده نمائید، در متد `SetupContext` کلاس `FixtureBase` که در قسمت قبل معرفی شد، سطر آخر را به صورت زیر تغییر دهید، تا اسکریپت دیتابیس نیز به صورت خودکار در خروجی اس کیوال آزمون واحد لحاظ شود (پارامتر دوم آن مشخص می‌کند که `schema` ساخته شده، نمایش داده شود یا خیر):

```
SessionSource.BuildSchema(Session, true);
```

پس از این تغییر و انجام مجدد آزمون واحد، اسکریپت دیتابیس ما به صورت زیر خواهد بود (که جهت ایجاد یک دیتابیس `SQLite` می‌تواند مورد استفاده قرار گیرد):

```
    drop table if exists "OrderItem"
```

```
drop table if exists "Order"

drop table if exists "Customer"

drop table if exists "Product"

drop table if exists "Employee"

drop table if exists hibernate_unique_key

create table "OrderItem" (
    Id INTEGER not null,
   Quantity INTEGER not null,
   Product_id INTEGER not null,
   Order_id INTEGER,
   ListIndex INTEGER,
   primary key (Id)
)

create table "Order" (
    Id INTEGER not null,
   OrderDate DATETIME not null,
   Employee_id INTEGER not null,
   Customer_id INTEGER not null,
   primary key (Id)
)

create table "Customer" (
    Id INTEGER not null,
   FirstName TEXT not null,
   LastName TEXT not null,
   AddressLine1 TEXT not null,
   AddressLine2 TEXT,
   PostalCode TEXT not null,
   City TEXT not null,
   CountryCode TEXT not null,
   primary key (Id)
)

create table "Product" (
    Id INTEGER not null,
   Name TEXT not null,
   UnitPrice NUMERIC not null,
   Discontinued INTEGER not null,
   primary key (Id)
)

create table "Employee" (
    Id INTEGER not null,
   LastName TEXT,
   FirstName TEXT,
   primary key (Id)
)

create table hibernate_unique_key (
     next_hi INTEGER
)
```

البته اگر مستندات SQLite را مطالعه کرده باشید می‌دانید که مفهوم کلید خارجی در این دیتابیس وجود دارد اما اعمال نمی‌شود! (برای اعمال آن باید تریگر نوشت) به همین جهت در این اسکریپت تولیدی خبری از کلید خارجی نیست.

برای اینکه از دیتابیس اس کیوال سرور استفاده کنیم، در همان متد SetupContext کلاس مذکور، سطر اول را به صورت زیر تغییر دهید (نوع دیتابیس اس کیوال سرور 2008 مشخص شده و سپس رشته اتصالی به دیتابیس ذکر گردیده است):

```
            var cfg = Fluently.Configure().Database(
                // SQLiteConfiguration.Standard.ShowSql().InMemory
                MsSqlConfiguration
                .MsSql2008
                .ShowSql()
                .ConnectionString("Data Source=(local);Initial
Catalog=testdb2009;Integrated Security = true")
                 );
```

اکنون اگر مجددا آزمون واحد را اجرا نمائیم، اسکریپت تولیدی به صورت زیر خواهد بود (در اینجا مفهوم استقلال برنامه از نوع دیتابیس را به خوبی می‌توان درک کرد):

```
    if exists (select 1 from sys.objects where object_id =
OBJECT_ID(N'[FK3EF88858466CFBF7]') AND parent_object_id =
OBJECT_ID('[OrderItem]'))
alter table [OrderItem]  drop constraint FK3EF88858466CFBF7


    if exists (select 1 from sys.objects where object_id =
OBJECT_ID(N'[FK3EF888589F32DE52]') AND parent_object_id =
OBJECT_ID('[OrderItem]'))
alter table [OrderItem]  drop constraint FK3EF888589F32DE52


    if exists (select 1 from sys.objects where object_id =
OBJECT_ID(N'[FK3117099B1EBA72BC]') AND parent_object_id = OBJECT_ID('[Order]'))
alter table [Order]  drop constraint FK3117099B1EBA72BC


    if exists (select 1 from sys.objects where object_id =
OBJECT_ID(N'[FK3117099BB2F9593A]') AND parent_object_id = OBJECT_ID('[Order]'))
alter table [Order]  drop constraint FK3117099BB2F9593A


    if exists (select * from dbo.sysobjects where id =
object_id(N'[OrderItem]') and OBJECTPROPERTY(id, N'IsUserTable') = 1) drop
table [OrderItem]

    if exists (select * from dbo.sysobjects where id = object_id(N'[Order]')
and OBJECTPROPERTY(id, N'IsUserTable') = 1) drop table [Order]

    if exists (select * from dbo.sysobjects where id = object_id(N'[Customer]')
and OBJECTPROPERTY(id, N'IsUserTable') = 1) drop table [Customer]

    if exists (select * from dbo.sysobjects where id = object_id(N'[Product]')
and OBJECTPROPERTY(id, N'IsUserTable') = 1) drop table [Product]

    if exists (select * from dbo.sysobjects where id = object_id(N'[Employee]')
and OBJECTPROPERTY(id, N'IsUserTable') = 1) drop table [Employee]
```

```
    if exists (select * from dbo.sysobjects where id =
object_id(N'hibernate_unique_key') and OBJECTPROPERTY(id, N'IsUserTable') = 1)
drop table hibernate_unique_key

    create table [OrderItem] (
        Id INT not null,
       Quantity INT not null,
       Product_id INT not null,
       Order_id INT null,
       ListIndex INT null,
       primary key (Id)
    )

    create table [Order] (
        Id INT not null,
       OrderDate DATETIME not null,
       Employee_id INT not null,
       Customer_id INT not null,
       primary key (Id)
    )

    create table [Customer] (
        Id INT not null,
       FirstName NVARCHAR(50) not null,
       LastName NVARCHAR(50) not null,
       AddressLine1 NVARCHAR(50) not null,
       AddressLine2 NVARCHAR(50) null,
       PostalCode NVARCHAR(10) not null,
       City NVARCHAR(50) not null,
       CountryCode NVARCHAR(2) not null,
       primary key (Id)
    )

    create table [Product] (
        Id INT not null,
       Name NVARCHAR(50) not null,
       UnitPrice DECIMAL(19,5) not null,
       Discontinued BIT not null,
       primary key (Id)
    )

    create table [Employee] (
        Id INT not null,
       LastName NVARCHAR(50) null,
       FirstName NVARCHAR(50) null,
       primary key (Id)
    )

    alter table [OrderItem]
        add constraint FK3EF88858466CFBF7
        foreign key (Product_id)
        references [Product]

    alter table [OrderItem]
        add constraint FK3EF888589F32DE52
        foreign key (Order_id)
        references [Order]

    alter table [Order]
```

```
    add constraint FK3117099B1EBA72BC
    foreign key (Employee_id)
    references [Employee]

alter table [Order]
    add constraint FK3117099BB2F9593A
    foreign key (Customer_id)
    references [Customer]

create table hibernate_unique_key (
     next_hi INT
)
```

که نکات ذیل در مورد آن جالب توجه است:

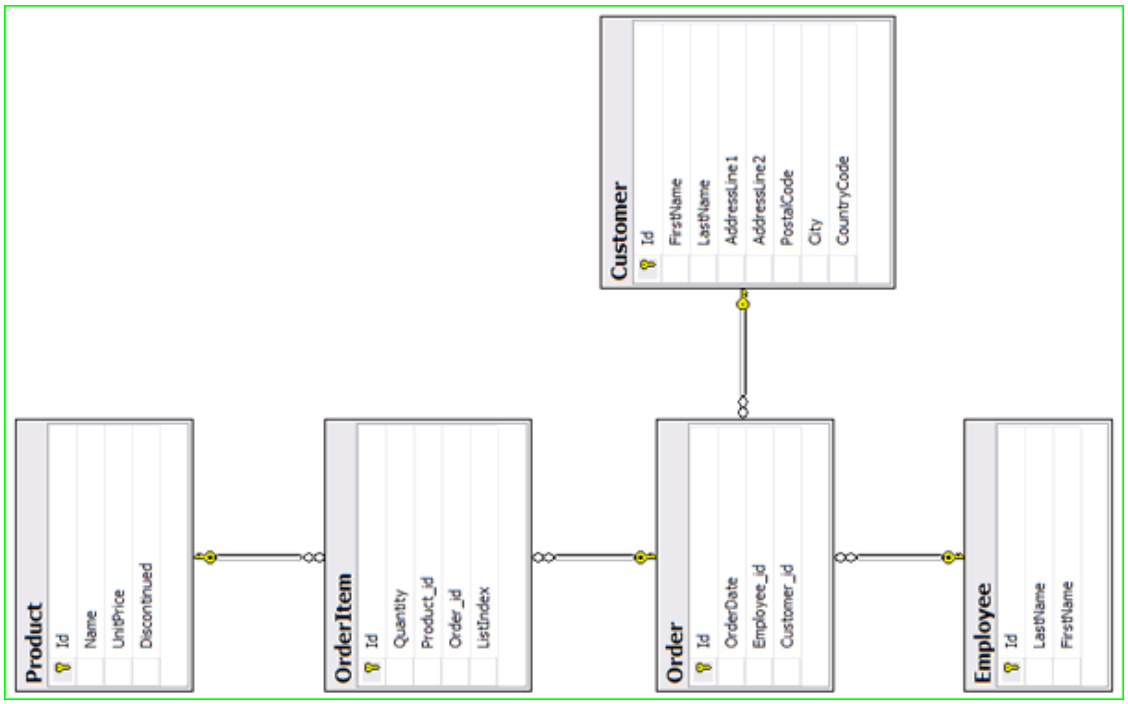
الف) جداول مطابق نام کلاس‌های ما تولید شده‌اند.

ب) نام فیلدها دقیقا مطابق نام خواص کلاس‌های ما تشکیل شده‌اند.

ج) `Id` ها به صورت `primary key` تعریف شده‌اند (از آنجائیکه ما در هنگام تعریف نگاشت‌ها، آن‌ها را از نوع `identity` مشخص کرده بودیم).

د) رشته‌ها به نوع `nvarchar` با اندازه `50` نگاشت شده‌اند.

ه) کلیدهای خارجی بر اساس نام جدول با پسوند `_id` تشکیل شده‌اند.



قسمت چهارم

در این قسمت یک مثال ساده از insert، load و delete را بر اساس اطلاعات قسمت‌های قبل با هم مرور خواهیم کرد. برای سادگی کار از یک برنامه Console استفاده خواهد شد (هر چند مرسوم شده است که برای نوشتن آزمایشات از آزمون‌های واحد بجای این نوع پروژه‌ها استفاده شود). همچنین فرض هم بر این است که database schema برنامه را مطابق قسمت قبل در اس کیوال سرور ایجاد کرده اید (نکته آخر بحث قسمت سوم).

یک پروژه جدید از نوع کنسول را به solution برنامه (همان NHSample1 که در قسمت‌های قبل ایجاد شد)، اضافه نمائید.
سپس ارجاعاتی را به اسمبلی‌های زیر به آن اضافه کنید:

FluentNHibernate.dll
NHibernate.dll
NHibernate.ByteCode.Castle.dll
NHSample1.dll: **در قسمت‌های قبل تعاریف موجودیت‌ها و نگاشت آن‌ها را در این پروژه class library ایجاد کرده بودیم و اکنون قصد استفاده از آن را داریم.**

اگر دیتابیس قسمت قبل را هنوز ایجاد نکرده‌اید، کلاس CDb را به برنامه افزوده و سپس متد CreateDb آن‌را به برنامه اضافه نمائید.

```
using FluentNHibernate;
using FluentNHibernate.Cfg;
using FluentNHibernate.Cfg.Db;
using NHSample1.Mappings;

namespace ConsoleTestApplication
{
    class CDb
    {
        public static void CreateDb(IPersistenceConfigurer dbType)
        {
            var cfg = Fluently.Configure().Database(dbType);

            PersistenceModel pm = new PersistenceModel();
            pm.AddMappingsFromAssembly(typeof(CustomerMapping).Assembly);
            var sessionSource = new SessionSource(
                                cfg.BuildConfiguration().Properties,
                                pm);

            var session = sessionSource.CreateSession();
            sessionSource.BuildSchema(session, true);
        }
    }
}
```

اکنون برای ایجاد دیتابیس اس کیوال سرور بر اساس نگاشت‌های قسمت قبل، تنها کافی است دستور ذیل را صادر کنیم:

```
CDb.CreateDb(
              MsSqlConfiguration
                 .MsSql2008
                 .ConnectionString("Data Source=(local);Initial
Catalog=HelloNHibernate;Integrated Security = true")
                 .ShowSql());
```

تمامی جداول و ارتباطات مرتبط در دیتابیسی که در کانکشن استرینگ فوق ذکر شده است، ایجاد خواهد شد.

در ادامه یک کلاس جدید به نام Config را به برنامه کنسول ایجاد شده اضافه کنید:

```
using FluentNHibernate.Cfg;
using FluentNHibernate.Cfg.Db;
using NHibernate;
using NHSample1.Mappings;

namespace ConsoleTestApplication
{
    class Config
    {
        public static ISessionFactory
CreateSessionFactory(IPersistenceConfigurer dbType)
        {
            return
               Fluently.Configure().Database(dbType
                ).Mappings(m =>
m.FluentMappings.AddFromAssembly(typeof(CustomerMapping).Assembly))
                 .BuildSessionFactory();
        }
    }
}
```

اگر بحث را دنبال کرده باشید، این کلاس را پیشتر در کلاس `FixtureBase` **آزمون واحد خود، به نحوی دیگر دیده بودیم. برای کار با** `NHibernate` **نیاز به یک سشن مپ شده به موجودیت‌های برنامه می‌باشد که توسط متد** `CreateSessionFactory` **کلاس فوق ایجاد خواهد شد. این متد را به این جهت استاتیک تعریف کرده‌ایم که هیچ نوع وابستگی به کلاس جاری خود ندارد. در آن نوع دیتابیس مورد استفاده (برای مثال اس کیوال سرور** `2008` **یا هر مورد دیگری که مایل بودید)، به همراه اسمبلی حاوی اطلاعات نگاشت‌های برنامه معرفی شده‌اند.**

اکنون سورس کامل مثال برنامه را در نظر بگیرید:

کلاس `CDbOperations` **جهت اعمال ثبت و حذف اطلاعات:**

```
using System;
using NHibernate;
using NHSample1.Domain;

namespace ConsoleTestApplication
{
    class CDbOperations
    {
        ISessionFactory _factory;

        public CDbOperations(ISessionFactory factory)
        {
            _factory = factory;
        }

        public int AddNewCustomer()
        {
            using (ISession session = _factory.OpenSession())
            {
                using (ITransaction transaction = session.BeginTransaction())
                {
                    Customer vahid = new Customer()
                    {
                        FirstName = "Vahid",
                        LastName = "Nasiri",
```

```
                        AddressLine1 = "Addr1",
                        AddressLine2 = "Addr2",
                        PostalCode = "1234",
                        City = "Tehran",
                        CountryCode = "IR"
                    };

                    Console.WriteLine("Saving a customer...");

                    session.Save(vahid);
                    session.Flush();//بعد و هم با عملیات چندین

                    transaction.Commit();

                    return vahid.Id;
                }
            }
        }

        public void DeleteCustomer(int id)
        {
            using (ISession session = _factory.OpenSession())
            {
                using (ITransaction transaction = session.BeginTransaction())
                {
                    Customer customer = session.Load<Customer>(id);
                    Console.WriteLine("Id:{0}, Name: {1}", customer.Id,
customer.FirstName);

                    Console.WriteLine("Deleting a customer...");
                    session.Delete(customer);

                    session.Flush();//بعد و هم با عملیات چندین

                    transaction.Commit();
                }
            }
        }
    }
}
```

و سپس استفاده از آن در برنامه

```
using System;
using FluentNHibernate.Cfg.Db;
using NHibernate;
using NHSample1.Domain;

namespace ConsoleTestApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            //CDb.CreateDb(SQLiteConfiguration.Standard.ConnectionString("data
source=sample.sqlite").ShowSql());
            //return;

            //todo: Read ConnectionString from app.config or web.config
            using (ISessionFactory session = Config.CreateSessionFactory(
```

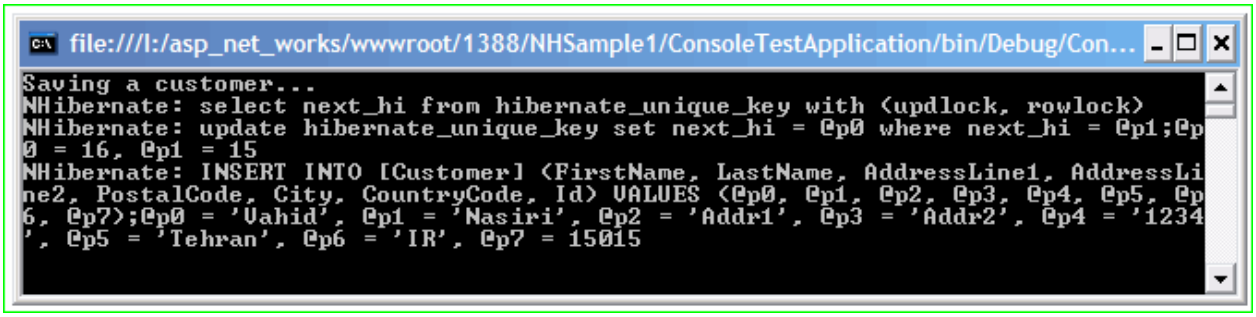
```
                    MsSqlConfiguration
                        .MsSql2008
                        .ConnectionString("Data Source=(local);Initial
Catalog=HelloNHibernate;Integrated Security = true")
                        .ShowSql()
                    ))
                {
                    CDbOperations db = new CDbOperations(session);
                    int id = db.AddNewCustomer();
                    Console.WriteLine("Loading a customer and delete it...");
                    db.DeleteCustomer(id);
                }

                Console.WriteLine("Press a key...");
                Console.ReadKey();
            }
        }
}
```

توضیحات:

نیاز است تا ISessionFactory را برای ساخت سشن‌های دسترسی به دیتابیس ذکر شده در تنظیمات آن جهت استفاده در تمام تردهای برنامه، ایجاد نمائیم. لازم به ذکر است که تا قبل از فراخوانی BuildSessionFactory این تنظیمات باید معرفی شده باشند و پس از آن دیگر اثری نخواهند داشت.
ایجاد شیء ISessionFactory هزینه بر است و گاهی بر اساس تعداد کلاس‌هایی که باید مپ شوند، ممکن است تا چند ثانیه به طول انجامد. به همین جهت نیاز است تا یکبار ایجاد شده و بارها مورد استفاده قرار گیرد. در برنامه به کرات از using استفاده شده تا اشیاء IDisposable را به صورت خودکار و حتمی، معدوم نماید.

بررسی متد AddNewCustomer:
در ابتدا یک سشن را از ISessionFactory موجود درخواست می‌کنیم. سپس یکی از بهترین تمرین‌های کاری جهت کار با دیتابیس‌ها ایجاد یک تراکنش جدید است تا اگر در حین اجرای کوئری‌ها مشکلی در سیستم، سخت افزار و غیره پدید آمد، دیتابیسی ناهماهنگ حاصل نشود. زمانیکه از تراکنش استفاده شود، تا هنگامیکه دستور transaction.Commit آن با موفقیت به پایان نرسیده باشد، اطلاعاتی در دیتابیس تغییر نخواهد کرد و از این لحاظ استفاده از تراکنش‌ها جزو الزامات یک برنامه اصولی است.
در ادامه یک وهله از شیء Customer را ایجاد کرده و آن‌را مقدار دهی می‌کنیم (این شیء در قسمت‌های قبل ایجاد گردید). سپس با استفاده از session.Save دستور ثبت را صادر کرده، اما تا زمانیکه transaction.Commit فراخوانی و به پایان نرسیده باشد، اطلاعاتی در دیتابیس ثبت نخواهد شد.
نیازی به ذکر سطر فلاش در این مثال نبود و NHibernate اینکار را به صورت خودکار انجام می‌دهد و فقط از این جهت عنوان گردید که اگر چندین عملیات را با هم معرفی کردید، استفاده از session.Flush سبب خواهد شد که رفت و برگشت‌ها به دیتابیس حداقل شود و فقط یکبار صورت گیرد.
در پایان این متد، Id ثبت شده در دیتابیس بازگشت داده می‌شود.

چون در متد CreateSessionFactory، متد ShowSql را نیز ذکر کرده بودیم، هنگام اجرای برنامه، عبارات SQL ایی که در پشت صحنه توسط NHibernate تولید می‌شوند را نیز می‌توان مشاهده نمود:



بررسی متد `DeleteCustomer`:

ایجاد سشن و آغاز تراکنش آن همانند متد `AddNewCustomer` **است. سپس در این سشن، یک شیء از نوع** `Customer` **با** `Id` **ایی مشخص** `load` **خواهد گردید. برای نمونه، نام این مشتری نیز در کنسول نمایش داده می‌شود. سپس این شیء مشخص و بارگذاری شده را به متد** `session.Delete` **ارسال کرده و پس از فراخوانی** `transaction.Commit`**، این مشتری از دیتابیس حذف می‌شود.**

برای نمونه خروجی `SQL` **پشت صحنه این عملیات که توسط** `NHibernate` **مدیریت می‌شود، به صورت زیر است:**

```
Saving a customer...
NHibernate: select next_hi from hibernate_unique_key with (updlock, rowlock)
NHibernate: update hibernate_unique_key set next_hi = @p0 where next_hi =
@p1;@p0 = 17, @p1 = 16
NHibernate: INSERT INTO [Customer] (FirstName, LastName, AddressLine1,
AddressLine2, PostalCode, City, CountryCode, Id) VALUES (@p0, @p1, @p2, @p3,
@p4, @p5, @p6, @p7);@p0 = 'Vahid', @p1 = 'Nasiri', @p2 = 'Addr1', @p3 =
'Addr2', @p4 = '1234', @p5 = 'Tehran', @p6 = 'IR', @p7 = 16016
Loading a customer and delete it...
NHibernate: SELECT customer0_.Id as Id2_0_, customer0_.FirstName as
FirstName2_0_, customer0_.LastName as LastName2_0_, customer0_.AddressLine1 as
AddressL4_2_0_, customer0_.AddressLine2 as AddressL5_2_0_,
customer0_.PostalCode as PostalCode2_0_, customer0_.City as City2_0_,
customer0_.CountryCode as CountryC8_2_0_ FROM [Customer] customer0_ WHERE
customer0_.Id=@p0;@p0 = 16016
Id:16016, Name: Vahid
Deleting a customer...
NHibernate: DELETE FROM [Customer] WHERE Id = @p0;@p0 = 16016
Press a key...
```

استفاده از دیتابیس SQLite بجای SQL Server در مثال فوق:

فرض کنید از هفته آینده قرار شده است که نسخه سبک و تک کاربره‌ای از برنامه ما تهیه شود. بدیهی است `SQL server` **برای این منظور انتخاب مناسبی نیست (هزینه بالا برای یک مشتری، مشکلات نصب، مشکلات نگهداری و امثال آن برای یک کاربر نهایی و نه یک سازمان بزرگ که حتما ادمینی برای این مسایل در نظر گرفته می‌شود).**

اکنون چه باید کرد؟ باید برنامه را از صفر بازنویسی کرد یا قسمت دسترسی به داده‌های آن‌را کلا مورد باز بینی قرار داد؟ اگر برنامه اسپاگتی ما اصلا لایه دسترسی به داده‌ها را نداشت چه؟! همه جای برنامه پر است از `SqlCommand` **و** `Open` **و** `Close`**! و عملا استفاده از یک دیتابیس دیگر یعنی باز نویسی کل برنامه.**

همانطور که ملاحظه می‌کنید، زمانیکه با `NHibernate` **کار شود، مدیریت لایه دسترسی به داده‌ها به این فریم ورک محول می‌شود و اکنون برای استفاده از دیتابیس** `SQLite` **تنها باید تغییرات زیر صورت گیرد:**

ابتدا ارجاعی را به اسمبلی `System.Data.SQLite.dll` **اضافه نمائید (تمام این اسمبلی‌های ذکر شده به همراه مجموعه** `FluentNHibernate` **ارائه می‌شوند). سپس:**

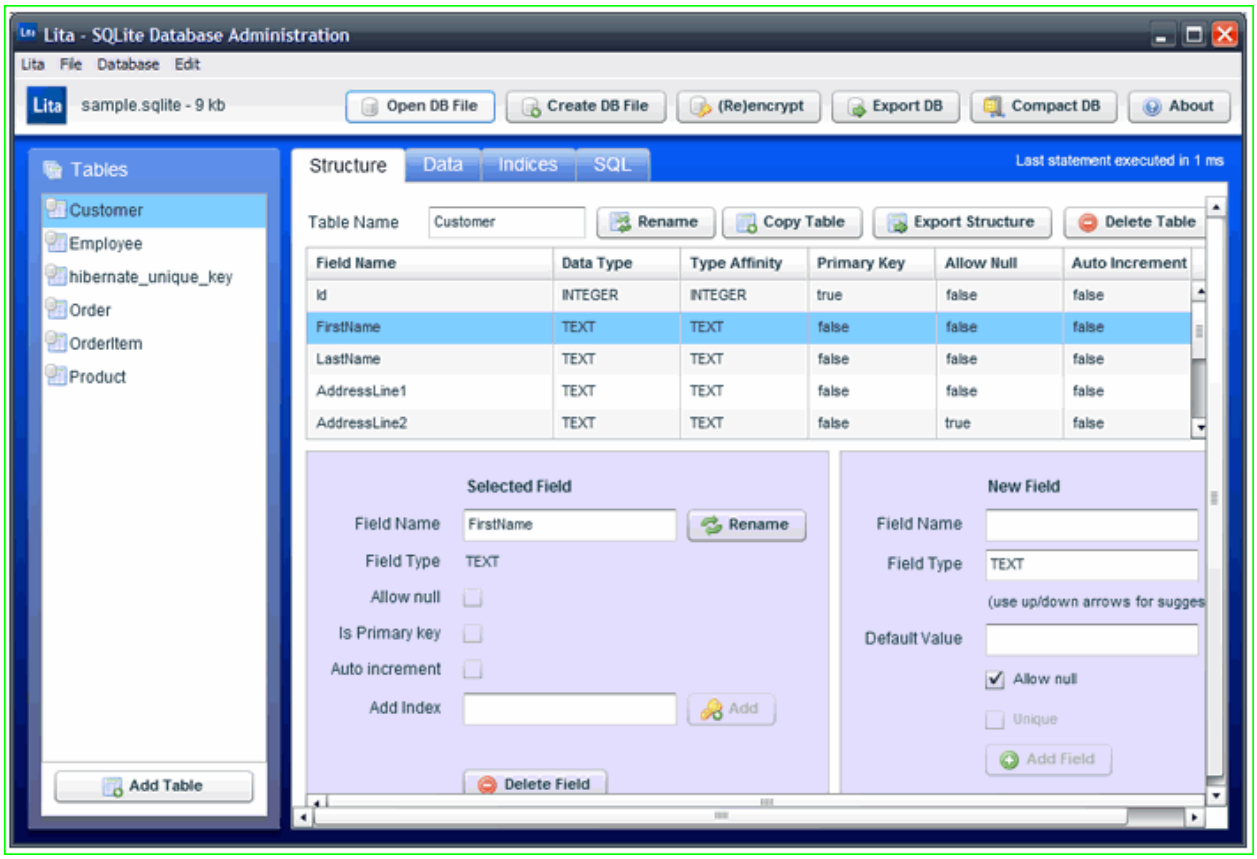
الف) ایجاد یک دیتابیس خام بر اساس کلاس‌های `domain` **و** `mapping` **تعریف شده در قسمت‌های قبل به صورت خودکار**

```
CDb.CreateDb(SQLiteConfiguration.Standard.ConnectionString("data
source=sample.sqlite").ShowSql());
```

ب) تغییر آرگومان متد `CreateSessionFactory`

```
          //todo: Read ConnectionString from app.config or web.config
          using (ISessionFactory session = Config.CreateSessionFactory(
                        SQLiteConfiguration.Standard.ConnectionString("data
source=sample.sqlite").ShowSql()
                    ))
          {
 ...
```

نمایی از دیتابیس `SQLite` **تشکیل شده پس از اجرای متد قسمت الف ، در برنامه** `Lita` **:**



[دریافت سورس برنامه تا این قسمت]

نکته:

در سه قسمت قبل، تمام خواص پابلیک کلاس‌های پوشه `domain` **را به صورت معمولی و متداول معرفی کردیم. اگر نیاز به** `lazy loading` **در برنامه وجود داشت، باید تمامی کلاس‌ها را ویرایش کرده و واژه کلیدی** `virtual` **را به کلیه خواص پابلیک آن‌ها اضافه کرد. علت هم این است که برای عملیات** `lazy loading` **، فریم ورک** `NHibernate` **باید یک سری پروکسی را به صورت خودکار جهت کلاس‌های برنامه ایجاد نماید و برای این امر نیاز است تا بتواند این خواص را تحریف (**`override`**) کند. به همین جهت باید آن‌ها را به صورت** `virtual` **تعریف کرد. همچنین تمام سطرهای** `Not.LazyLoad` **نیز باید حذف شوند.**

قسمت پنجم

استفاده از LINQ جهت انجام کوئری‌ها توسط NHibernate

نگارش نهایی 1.0 کتابخانه‌ی LINQ to NHibernate اخیرا (حدود سه ماه قبل) منتشر شده است. در این قسمت قصد داریم با کمک این کتابخانه، اعمال متداول انجام کوئری‌ها را بر روی دیتابیس قسمت قبل انجام دهیم.

توسط این نگارش ارائه شده، کلیه اعمال قابل انجام با criteria API این فریم ورک را می‌توان از طریق LINQ نیز انجام داد (NHibernate برای کار با داده‌ها و جستجوهای پیشرفته بر روی آن‌ها، HQL : Hibernate Query Language و Criteria API را سال‌ها قبل توسعه داده است).

جهت دریافت پروایدر LINQ مخصوص NHibernate به آدرس زیر مراجعه نمائید:

http://sourceforge.net/projects/nhibernate/files

پس از دریافت آن، به همان برنامه کنسول قسمت قبل، دو ارجاع را باید افزود:

الف) ارجاعی به اسمبلی NHibernate.Linq.dll

ب) ارجاعی به اسمبلی استاندارد System.Data.Services.dll دات نت فریم ورک سه و نیم

در ابتدای متد Main برنامه قصد داریم تعدادی مشتری را به دیتابیس اضافه نمائیم. به همین منظور متد AddNewCustomers را به کلاس CDbOperations برنامه کنسول قسمت قبل اضافه نمائید. این متد لیستی از مشتری‌ها را دریافت کرده و آن‌ها را در طی یک تراکنش به دیتابیس اضافه می‌کند:

```
public void AddNewCustomers(params Customer[] customers)
{
    using (ISession session = _factory.OpenSession())
    {
        using (ITransaction transaction = session.BeginTransaction())
        {
            foreach (var data in customers)
                session.Save(data);

            session.Flush();

            transaction.Commit();
        }
    }
}
```

در اینجا استفاده از واژه کلیدی params سبب می‌شود که بجای تعریف الزامی یک آرایه از نوع مشتری‌ها، بتوانیم تعداد دلخواهی پارامتر از نوع مشتری را به این متد ارسال کنیم.

پس از افزودن این ارجاعات، کلاس جدیدی را به نام CLinqTest به برنامه کنسول اضافه نمائید. ساختار کلی این کلاس که قصد استفاده از پروایدر LINQ مخصوص NHibernate را دارد باید به شکل زیر باشد (به کلاس پایه NHibernateContext دقت نمائید):

```
using System.Collections.Generic;
using System.Linq;
using NHibernate;
using NHibernate.Linq;
using NHSample1.Domain;

namespace ConsoleTestApplication
```

```
{
    class CLinqTest : NHibernateContext
    { }
}
```

اکنون پس از مشخص شدن context یا زمینه، نحوه ایجاد یک کوئری ساده NHibernate LINQ to به صورت زیر می‌تواند باشد:

```
using System.Collections.Generic;
using System.Linq;
using NHibernate;
using NHibernate.Linq;
using NHSample1.Domain;

namespace ConsoleTestApplication
{
    class CLinqTest : NHibernateContext
    {
        ISessionFactory _factory;

        public CLinqTest(ISessionFactory factory)
        {
            _factory = factory;
        }

        public List<Customer> GetAllCustomers()
        {
            using (ISession session = _factory.OpenSession())
            {
                var query = from x in session.Linq<Customer>() select x;
                return query.ToList();
            }
        }
    }
}
```

ابتدا علاوه بر سایر فضاهای نام مورد نیاز، فضای نام NHibernate.Linq به پروژه افزوده می‌شود. سپس از extension متدی به نام Linq بر روی اشیاء ISession از نوع یکی از موجودیت‌های تعریف شده در برنامه در قسمت‌های قبل، می‌توان جهت تهیه کوئری‌های Linq مورد نظر بهره برد.

در این کوئری، لیست تمامی مشتری‌ها بازگشت داده می‌شود.

سپس جهت استفاده و بررسی آن در متد Main برنامه خواهیم داشت:

```
        static void Main(string[] args)
        {
            using (ISessionFactory session = Config.CreateSessionFactory(
                  MsSqlConfiguration
                      .MsSql2008
                      .ConnectionString("Data Source=(local);Initial
Catalog=HelloNHibernate;Integrated Security = true")
                      .ShowSql()
                ))
            {

                var customer1 = new Customer()
                {
                    FirstName = "Vahid",
                    LastName = "Nasiri",
                    AddressLine1 = "Addr1",
```

```
                    AddressLine2 = "Addr2",
                    PostalCode = "1234",
                    City = "Tehran",
                    CountryCode = "IR"
                };

                var customer2 = new Customer()
                {
                    FirstName = "Ali",
                    LastName = "Hasani",
                    AddressLine1 = "Addr..1",
                    AddressLine2 = "Addr..2",
                    PostalCode = "4321",
                    City = "Shiraz",
                    CountryCode = "IR"
                };

                var customer3 = new Customer()
                {
                    FirstName = "Mohsen",
                    LastName = "Shams",
                    AddressLine1 = "Addr...1",
                    AddressLine2 = "Addr...2",
                    PostalCode = "5678",
                    City = "Ahwaz",
                    CountryCode = "IR"
                };

                CDbOperations db = new CDbOperations(session);
                db.AddNewCustomers(customer1, customer2, customer3);

                CLinqTest lt = new CLinqTest(session);
                foreach (Customer customer in lt.GetAllCustomers())
                {
                    Console.WriteLine("Customer: LastName = {0}",
customer.LastName);
                }
            }

            Console.WriteLine("Press a key...");
            Console.ReadKey();

        }
```

در این متد ابتدا تعدادی رکورد تعریف و سپس به دیتابیس اضافه شدند. در ادامه لیست تمامی آن‌ها از دیتابیس دریافت و نمایش داده می‌شود.

مهمترین مزیت استفاده از LINQ **در این نوع کوئری‌ها نسبت به روش‌های دیگر، استفاده از کدهای** strongly typed **دات نتی تحت نظر کامپایلر است، نسبت به رشته‌های معمولی** SQL **که کامپایلر کنترلی را بر روی آن‌ها نمی‌تواند داشته باشد (برای مثال اگر نوع یک ستون تغییر کند یا نام آن، در حالت استفاده از** LINQ **بلافاصله یک خطا را از کامپایلر جهت تصحیح مشکلات دریافت خواهیم کرد که این مورد در زمان استفاده از یک رشته معمولی صادق نیست). همچنین مزیت فراهم بودن** Intellisense **را حین نوشتن کوئری‌هایی از این دست نیز نمی‌توان ندید گرفت.**

مثالی دیگر:

لیست تمام مشتری‌های شیرازی را نمایش دهید:

ابتدا متد GetCustomersByCity **را به کلاس** CLinqTest **فوق اضافه می‌کنیم:**

```
        public List<Customer> GetCustomersByCity(string city)
        {
            using (ISession session = _factory.OpenSession())
            {
                var query = from x in session.Linq<Customer>()
                            where x.City == city
                            select x;
                return query.ToList();
            }
        }
```

سپس برای استفاده از آن، چند سطر ساده زیر به ادامه متد Main **اضافه می‌شوند:**

```
                foreach (Customer customer in lt.GetCustomersByCity("Shiraz"))
                {
                    Console.WriteLine("Customer: LastName = {0}",
customer.LastName);
                }
```

یکی دیگر از مزایای استفاده از LINQ to NHibernate **، امکان بکارگیری** LINQ **بر روی تمامی دیتابیس‌های پشتیبانی شده توسط** NHibernate **است؛ برای مثال مای اس کیوال، اوراکل و ....**

لیست کامل دیتابیس‌های پشتیبانی شده توسط NHibernate **را در این آدرس می‌توانید مشاهده نمائید. (البته به نظر لیست آن، آنچنان هم به روز نیست؛ چون در نگارش آخر** NHibernate **، پشتیبانی از اس کیوال سرور** 2008 **هم اضافه شده است)**

نکته:

در کوئری‌های مثال‌های فوق همواره باید session.Linq<T> **را ذکر کرد. اگر علاقمند بودید شبیه به روشی که در** LINQ to SQL **موجود است مثلا** db.TableName **بجای** session.Linq<T> **در کوئری‌ها ذکر گردد، می‌توان اصلاحاتی را به صورت زیر اعمال کرد:**

یک کلاس جدید را به نام SampleContext **به برنامه کنسول جاری با محتویات زیر اضافه نمائید:**

```
using System.Linq;
using NHibernate;
using NHibernate.Linq;
using NHSample1.Domain;

namespace ConsoleTestApplication
{
    class SampleContext : NHibernateContext
    {
        public SampleContext(ISession session)
            : base(session)
        { }

        public IOrderedQueryable<Customer> Customers
        {
            get { return Session.Linq<Customer>(); }
        }

        public IOrderedQueryable<Employee> Employees
        {
            get { return Session.Linq<Employee>(); }
        }

        public IOrderedQueryable<Order> Orders
        {
            get { return Session.Linq<Order>(); }
        }
```

```
        public IOrderedQueryable<OrderItem> OrderItems
        {
            get { return Session.Linq<OrderItem>(); }
        }

        public IOrderedQueryable<Product> Products
        {
            get { return Session.Linq<Product>(); }
        }
    }
}
```

در این کلاس به ازای تمام موجودیت‌های تعریف شده در پوشه domain **برنامه اصلی خود (همان** NHSample1 **قسمت‌های اول و دوم)، یک متد از نوع** IOrderedQueryable **را باید تشکیل دهیم که پیاده سازی آن‌را ملاحظه می‌نمائید.**

سپس بازنویسی متد GetCustomersByCity **بر اساس** SampleContext **فوق به صورت زیر خواهد بود که به کوئری‌های** LINQ **بسیار شبیه است:** to SQL

```
using System.Collections.Generic;
using System.Linq;
using NHibernate;
using NHSample1.Domain;

namespace ConsoleTestApplication
{
    class CSampleContextTest
    {
        ISessionFactory _factory;

        public CSampleContextTest(ISessionFactory factory)
        {
            _factory = factory;
        }

        public List<Customer> GetCustomersByCity(string city)
        {
            using (ISession session = _factory.OpenSession())
            {
                using (SampleContext db = new SampleContext(session))
                {
                    var query = from x in db.Customers
                                where x.City == city
                                select x;
                    return query.ToList();
                }
            }
        }
    }
}
```

دریافت سورس برنامه تا این قسمت

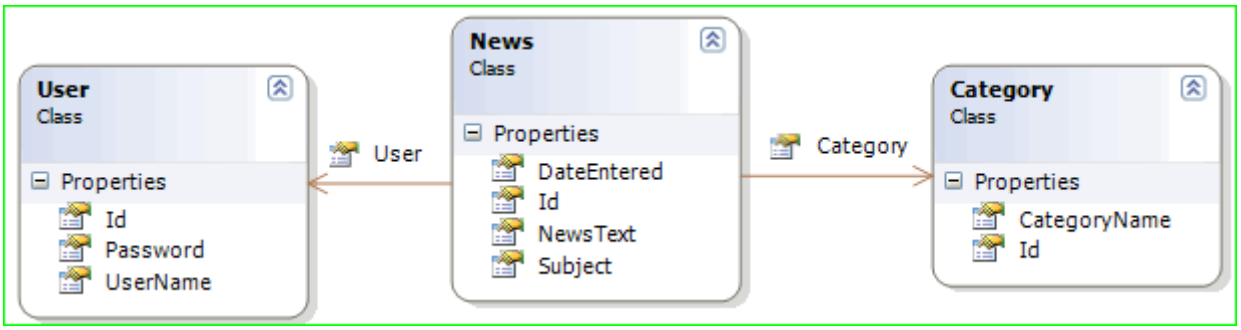
و در تکمیل این بحث، می‌توان به لیستی از 101 **مثال** LINQ **ارائه شده در** MSDN **اشاره کرد که یکی از بهترین و سریع ترین مراجع یادگیری مبحث** LINQ **است.**

قسمت ششم

آشنايي با Automapping در فريم ورك Fluent Nhibernate

اگر قسمت‌های قبل را دنبال کرده باشید، احتمالا به پروسه طولانی ساخت نگاشت‌ها توجه کرده‌اید. با کمک فریم ورک Fluent NHibernate می‌توان پروسه نگاشت domain model خود را به model data متناظر آن به صورت خودکار نیز انجام داد و قسمت عمده‌ای از کار به این صورت حذف خواهد شد. (این مورد یکی از تفاوت‌های مهم NHibernate با نمونه‌های مشابهی است که مایکروسافت تا تاریخ نگارش این مقاله ارائه داده است. برای مثال در نگارش‌های فعلی LINQ to SQL یا Entity framework ، اول دیتابیس مطرح است و بعد ساخت کد از روی آن، در حالیکه در اینجا ابتدا کد و طراحی سیستم مطرح است و بعد نگاشت آن به سیستم داده‌ای و دیتابیس)

امروز قصد داریم یک سیستم ساده ثبت خبر را از صفر با NHibernate پیاده سازی کنیم و همچنین مروری داشته باشیم بر قسمت‌های قبلی.



مطابق کلاس دیاگرام فوق، این سیستم از سه کلاس خبر، کاربر ثبت کننده‌ی خبر و گروه خبری مربوطه تشکیل شده است.

ابتدا یک پروژه کنسول جدید را به نام NHSample2 آغاز کنید. سپس ارجاعاتی را به اسمبلی‌های زیر به آن اضافه نمائید:

FluentNHibernate.dll
NHibernate.dll
NHibernate.ByteCode.Castle.dll
NHibernate.Linq.dll

و ارجاعی به اسمبلی استاندارد System.Data.Services.dll دات نت فریم ورک سه و نیم

سپس پوشه‌ای را به نام Domain به این پروژه اضافه نمائید (کلیک راست روی نام پروژه در VS.Net و سپس مراجعه به منوی Add->New folder). در این پوشه تعاریف موجودیت‌های برنامه را قرار خواهیم داد. سه کلاس جدید Category، User و News را در این پوشه ایجاد نمائید. محتویات این سه کلاس به شرح زیر هستند:

```
namespace NHSample2.Domain
{
    public class User
    {
        public virtual int Id { get; set; }
        public virtual string UserName { get; set; }
        public virtual string Password { get; set; }
    }
}

namespace NHSample2.Domain
{
    public class Category
    {
```

```
        public virtual int Id { get; set; }
        public virtual string CategoryName { get; set; }
    }
}

using System;

namespace NHSample2.Domain
{
  public class News
  {
      public virtual Guid Id { get; set; }
      public virtual string Subject { get; set; }
      public virtual string NewsText { get; set; }
      public virtual DateTime DateEntered { get; set; }
      public virtual Category Category { get; set; }
      public virtual User User { get; set; }
  }
}
```

همانطور که در قسمت‌های قبل نیز ذکر شد، تمام خواص پابلیک کلاس‌های Domain ما به صورت virtual تعریف شده‌اند تا lazy loading را در NHibernate فعال سازیم. در حالت lazy loading، اطلاعات تنها زمانیکه به آن‌ها نیاز باشد بارگذاری خواهند شد. این مورد در حالتیکه نیاز به نمایش اطلاعات تنها یک شیء وجود داشته باشد بسیار مطلوب می‌باشد، یا هنگام ثبت و به روز رسانی اطلاعات نیز یکی از بهترین روش‌ها است. اما زمانیکه با لیستی از اطلاعات سروکار داشته باشیم باعث کاهش افت کارآیی خواهد شد زیرا برای مثال نمایش آن‌ها سبب خواهد شد که 100 ها کوئری دیگر جهت دریافت اطلاعات هر رکورد در حال نمایش اجرا شود (مفهوم دسترسی به اطلاعات تنها در صورت نیاز به آن‌ها). Lazy loading و eager loading (همانند مثال‌های قبلی) هر دو در NHibernate به سادگی قابل تنظیم هستند (برای مثال LINQ to SQL به صورت پیش فرض همواره lazy load است و تا این تاریخ راه استانداردی برای امکان تغییر و تنظیم این مورد پیش بینی نشده است).

اکنون کلاس جدید Config را به برنامه اضافه نمائید:

```
using FluentNHibernate.Automapping;
using FluentNHibernate.Cfg;
using FluentNHibernate.Cfg.Db;
using NHibernate;
using NHibernate.Cfg;
using NHibernate.Tool.hbm2ddl;

namespace NHSample2
{
    class Config
    {
        public static Configuration GenerateMapping(IPersistenceConfigurer
dbType)
        {
            var cfg = dbType.ConfigureProperties(new Configuration());

            new AutoPersistenceModel()
                .Where(x => x.Namespace.EndsWith("Domain"))

.AddEntityAssembly(typeof(NHSample2.Domain.News).Assembly).Configure(cfg);

            return cfg;
        }
```

```
        public static void GenerateDbScript(Configuration config, string
filePath)
        {
            bool script = true;//گردد تولید دیتابیس اسکریپت فقط
            bool export = false;//شود اجرا هم دیتابیس روي بر نیست نیازي
            new SchemaExport(config).SetOutputFile(filePath).Create(script,
export);
        }

        public static void BuildDbSchema(Configuration config)
        {
            bool script = false;//شود داده نمایش هم کنسول در خروجي آیا
            bool export = true;//شود اجرا هم دیتابیس روي بر آیا
            bool drop = false;//شوند دراپ موجود اطلاعات آیا
            new SchemaExport(config).Execute(script, export, drop);
        }

        public static void CreateSQL2008DbPlusScript(string connectionString,
string filePath)
        {
            Configuration cfg =
                GenerateMapping(
                MsSqlConfiguration
                .MsSql2008
                .ConnectionString(connectionString)
                .ShowSql()
                );
            GenerateDbScript(cfg, filePath);
            BuildDbSchema(cfg);
        }

        public static ISessionFactory
CreateSessionFactory(IPersistenceConfigurer dbType)
        {
            return
               Fluently.Configure().Database(dbType)
                .Mappings(m => m.AutoMappings
                            .Add(
                                new AutoPersistenceModel()
                                    .Where(x => x.Namespace.EndsWith("Domain"))

.AddEntityAssembly(typeof(NHSample2.Domain.News).Assembly))
                                    )
                 .BuildSessionFactory();
        }
    }
}
```

در متد `GenerateMapping` **از قابلیت** `Automapping` **موجود در فریم ورک** `Fluent Nhibernate` **استفاده شده است (بدون نوشتن حتی یک سطر جهت تعریف این نگاشت‌ها). این متد نوع دیتابیس مورد نظر را جهت ساخت تنظیمات خود دریافت می‌کند. سپس با کمک کلاس** `AutoPersistenceModel` **این فریم ورک، به صورت خودکار از اسمبلی برنامه نگاشت‌های لازم را به کلاس‌های موجود در پوشه** `Domain` **ما اضافه می‌کند (مرسوم است که این پوشه در یک پروژه** `Class library` **مجزا تعریف شود که در این برنامه جهت سهولت کار در خود برنامه قرار گرفته است). قسمت** `Where` **ذکر شده به این جهت معرفی گردیده است تا** `Fluent Nhibernate` **برای تمامی کلاس‌های موجود در اسمبلی جاری، سعی در تعریف نگاشت‌های لازم نکند. این نگاشت‌ها تنها به کلاس‌های موجود در پوشه دومین ما محدود شده‌اند.**

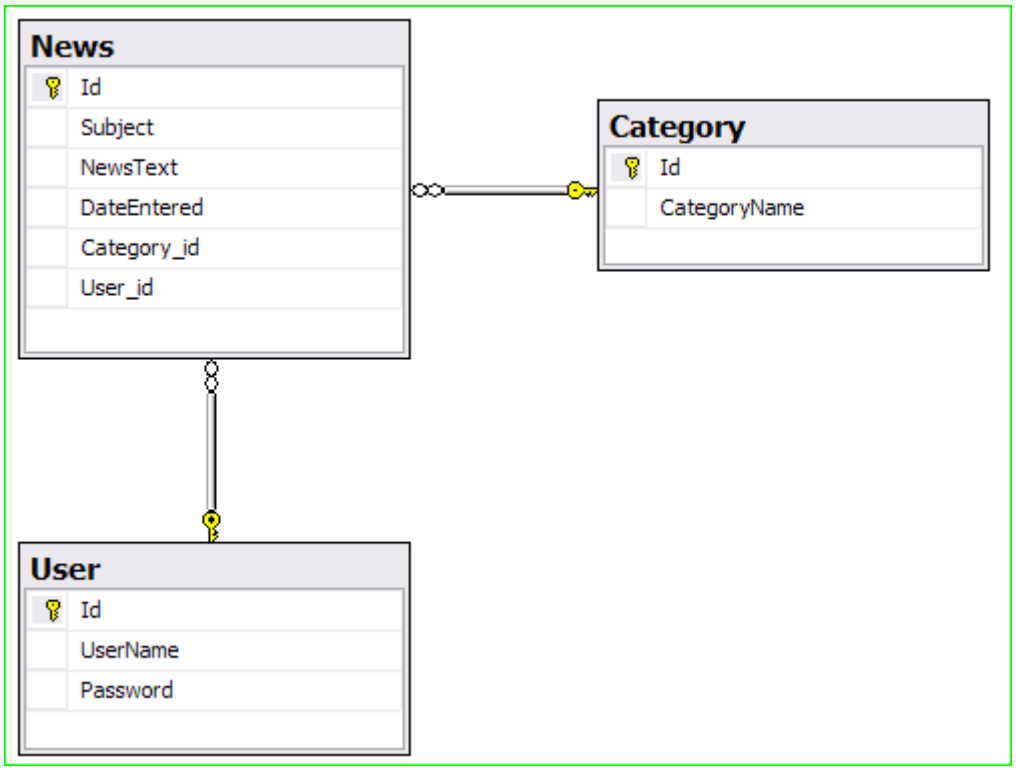
سه متد بعدی آن، جهت ایجاد اسکریپت دیتابیس از روی این نگاشت‌های تعریف شده و سپس اجرای این اسکریپت بر روی دیتابیس جاری معرفی شده، تهیه شده‌اند. برای مثال `CreateSQL2008DbPlusScript` **یک مثال ساده از استفاده دو متد قبلی جهت ایجاد**

اسکریپت و دیتابیس متناظر اس کیوال سرور 2008 بر اساس نگاشت‌های برنامه است.
با متد CreateSessionFactory در قسمت‌های قبل آشنا شده‌اید. تنها تفاوت آن در این قسمت، استفاده از کلاس AutoPersistenceModel جهت تولید خودکار نگاشت‌ها است.

در ادامه دیتابیس متناظر با موجودیت‌های برنامه را ایجاد خواهیم کرد:

```
using System;

namespace NHSample2
{
    class Program
    {
        static void Main(string[] args)
        {
            Config.CreateSQL2008DbPlusScript(
                "Data Source=(local);Initial
Catalog=HelloNHibernate;Integrated Security = true",
                "db.sql");

            Console.WriteLine("Press a key...");
            Console.ReadKey();
        }
    }
}
```

پس از اجرای برنامه، ابتدا فایل اسکریپت دیتابیس به نام db.sql در پوشه اجرایی برنامه تشکیل خواهد شد و سپس این اسکریپت به صورت خودکار بر روی دیتابیس معرفی شده اجرا می‌گردد. دیتابیس دیاگرام حاصل را در شکل زیر می‌توانید ملاحظه نمائید:



همچنین اسکریپت تولید شده آن، صرفنظر از عبارات drop اولیه، به صورت زیر است:

```
create table [Category] (
    Id INT IDENTITY NOT NULL,
   CategoryName NVARCHAR(255) null,
   primary key (Id)
```

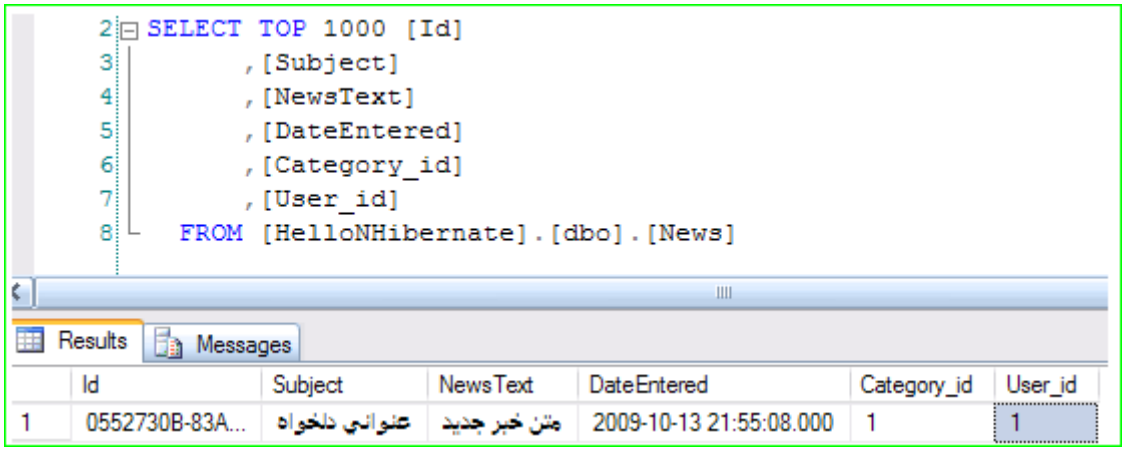
```
)

create table [User] (
    Id INT IDENTITY NOT NULL,
   UserName NVARCHAR(255) null,
   Password NVARCHAR(255) null,
   primary key (Id)
)

create table [News] (
    Id UNIQUEIDENTIFIER not null,
   Subject NVARCHAR(255) null,
   NewsText NVARCHAR(255) null,
   DateEntered DATETIME null,
   Category_id INT null,
   User_id INT null,
   primary key (Id)
)

alter table [News]
    add constraint FKE660F9E1C9CF79
    foreign key (Category_id)
    references [Category]

alter table [News]
    add constraint FKE660F95C1A3C92
    foreign key (User_id)

    references [User]
```

اکنون یک سری گروه خبری، کاربر و خبر را به دیتابیس خواهیم افزود:

```
using System;
using FluentNHibernate.Cfg.Db;
using NHibernate;
using NHSample2.Domain;

namespace NHSample2
{
    class Program
    {
        static void Main(string[] args)
        {
            using (ISessionFactory sessionFactory =
Config.CreateSessionFactory(
                    MsSqlConfiguration
                        .MsSql2008
                        .ConnectionString("Data Source=(local);Initial
Catalog=HelloNHibernate;Integrated Security = true")
                        .ShowSql()
                    ))
            {
                using (ISession session = sessionFactory.OpenSession())
                {
                    using (ITransaction transaction =
session.BeginTransaction())
                    {
```

```
                        //هاگروه باید ابتدا شده تعریف خارجي کلیدهاي به توجه با
کرد اضافه را
                        Category ca = new Category() { CategoryName = "Sport"
};
                        session.Save(ca);
                        Category ca2 = new Category() { CategoryName = "IT" };
                        session.Save(ca2);
                        Category ca3 = new Category() { CategoryName =
"Business" };
                        session.Save(ca3);

                        //کنیممي اضافه دیتابیس به را کاربر یك سپس
                        User u = new User() { Password = "123$5@1", UserName =
"VahidNasiri" };
                        session.Save(u);

                        //کرد ثبت را جدید خبر یك توانمي اکنون

                        News news = new News()
                        {
                            Category = ca,
                            User = u,
                            DateEntered = DateTime.Now,
                            Id = Guid.NewGuid(),
                            NewsText = "جدید خبر متن",
                            Subject = "دلخواه عنواني"
                        };
                        session.Save(news);

                        transaction.Commit(); //تراکنش پایان
                    }
                }
            }

            Console.WriteLine("Press a key...");
            Console.ReadKey();
        }
    }
}
```

جهت بررسی انجام عملیات ثبت هم می‌توان به دیتابیس مراجعه کرد، برای مثال:



```
SELECT TOP 1000 [Id]
      ,[Subject]
      ,[NewsText]
      ,[DateEntered]
      ,[Category_id]
      ,[User_id]
  FROM [HelloNHibernate].[dbo].[News]
```

Results | Messages

	Id	Subject	NewsText	DateEntered	Category_id	User_id
1	0552730B-83A...	عنواني دلخواه	متن خبر جدید	2009-10-13 21:55:08.000	1	1

و یا می‌توان از LINQ استفاده کرد:

برای مثال کاربر VahidNasiri تعریف شده را یافته، اطلاعات آن‌را نمایش دهید؛ سپس نام او را به Vahid ویرایش کرده و دیتابیس را به

روز کنید.

برای اینکه کوئری‌های LINQ ما شبیه به LINQ to SQL شوند، کلاس NewsContext را به صورت ذیل تشکیل می‌دهیم. این کلاس از کلاس پایه NHibernateContext مشتق شده و سپس به ازای تمام موجودیت‌های برنامه، یک متد از نوع IOrderedQueryable را تشکیل خواهیم داد.

```
using System.Linq;
using NHibernate;
using NHibernate.Linq;
using NHSample2.Domain;

namespace NHSample2
{
    class NewsContext : NHibernateContext
    {
        public NewsContext(ISession session)
            : base(session)
        { }

        public IOrderedQueryable<News> News
        {
            get { return Session.Linq<News>(); }
        }

        public IOrderedQueryable<Category> Categories
        {
            get { return Session.Linq<Category>(); }
        }

        public IOrderedQueryable<User> Users
        {
            get { return Session.Linq<User>(); }
        }
    }
}
```

اکنون جهت یافتن کاربر و به روز رسانی اطلاعات او در دیتابیس خواهیم داشت:

```
using System;
using FluentNHibernate.Cfg.Db;
using NHibernate;
using System.Linq;
using NHSample2.Domain;

namespace NHSample2
{
    class Program
    {
        static void Main(string[] args)
        {
            using (ISessionFactory sessionFactory =
Config.CreateSessionFactory(
                    MsSqlConfiguration
                        .MsSql2008
                        .ConnectionString("Data Source=(local);Initial
Catalog=HelloNHibernate;Integrated Security = true")
                        .ShowSql()
                    ))
```

```
            {
                using (ISession session = sessionFactory.OpenSession())
                {
                    using (ITransaction transaction =
session.BeginTransaction())
                    {
                        using (NewsContext db = new NewsContext(session))
                        {
                            var query = from x in db.Users
                                        where x.UserName == "VahidNasiri"
                                        select x;

                            //شد یافت چیزی اگر
                            if (query.Any())
                            {
                                User vahid = query.First();
                                //کاربر اطلاعات نمایش
                                Console.WriteLine("Id: {0}, UserName: {0}",
vahid.Id, vahid.UserName);
                                //کاربر نام رسانی روز به
                                vahid.UserName = "Vahid";
                                session.Update(vahid);

                                transaction.Commit(); //تراکنش پایان
                            }
                        }
                    }
                }
            }

            Console.WriteLine("Press a key...");
            Console.ReadKey();
        }
    }
}
```

مباحث تکمیلی AutoMapping

اگر به اسکریپت دیتابیس تولید شده دقت کرده باشید، عملیات AutoMapping **یک سری** پیش فرض‌هایی **را اعمال کرده است. برای مثال فیلد** Id **را از نوع** identity **و به صورت کلید تعریف کرده، یا رشته‌ها را به صورت** nvarchar **با طول** 255 **ایجاد نموده است. امکان سفارشی سازی این موارد نیز وجود دارد.**

مثال:

```
      using FluentNHibernate.Conventions.Helpers;

      public static Configuration GenerateMapping(IPersistenceConfigurer
dbType)
      {
          var cfg = dbType.ConfigureProperties(new Configuration());

          new AutoPersistenceModel()
          .Conventions.Add()
              .Where(x => x.Namespace.EndsWith("Domain"))
               .Conventions.Add(
```

```
                PrimaryKey.Name.Is(x => "ID"),
                DefaultLazy.Always(),
                ForeignKey.EndsWith("ID"),
                Table.Is(t => "tbl" + t.EntityType.Name)
                )
            .AddEntityAssembly(typeof(NHSample2.Domain.News).Assembly)
            .Configure(cfg);

        return cfg;
    }
```

تابع GenerateMapping معرفی شده را اینجا با قسمت Conventions.Add تکمیل کرده‌ایم. به این صورت دقیقا مشخص شده است که فیلدهایی با نام ID باید primary key در نظر گرفته شوند، همواره lazy loading صورت گیرد و نام کلید خارجی به ID ختم شود. همچنین نام جداول با tbl شروع گردد.

روش دیگری نیز برای معرفی این قرار دادها و پیش فرض‌ها وجود دارد. فرض کنید می‌خواهیم طول رشته پیش فرض را از 255 به 500 تغییر دهیم. برای اینکار باید اینترفیس IPropertyConvention را پیاده سازی کرد:

```
using FluentNHibernate.Conventions;
using FluentNHibernate.Conventions.Instances;

namespace NHSample2.Conventions
{
    class MyStringLengthConvention : IPropertyConvention
    {
        public void Apply(IPropertyInstance instance)
        {
            instance.Length(500);
        }
    }
}
```

سپس نحوه‌ی معرفی آن به صورت زیر خواهد بود:

```
    public static Configuration GenerateMapping(IPersistenceConfigurer
dbType)
    {
        var cfg = dbType.ConfigureProperties(new Configuration());

        new AutoPersistenceModel()
        .Conventions.Add()
            .Where(x => x.Namespace.EndsWith("Domain"))
             .Conventions.Add<MyStringLengthConvention>()
            .AddEntityAssembly(typeof(NHSample2.Domain.News).Assembly)
            .Configure(cfg);

        return cfg;
    }
```

نکته:

اگر برای یافتن اطلاعات بیشتر در این مورد در وب جستجو کنید، اکثر مثال‌هایی را که مشاهده خواهید کرد بر اساس نگارش بتای fluent NHibernate هستند و هیچکدام با نگارش نهایی این فریم ورک کار نمی‌کنند. در نگارش رسمی نهایی ارائه شده، تغییرات بسیاری صورت گرفته که آن‌ها را در این آدرس می‌توان مشاهده کرد.

دریافت سورس برنامه قسمت ششم

قسمت هفتم

مدیریت بهینه‌ي سشن فکتوري

ساخت یک شیء SessionFactory بسیار پر هزینه و زمانبر است. به همین جهت لازم است که این شیء یکبار حین آغاز برنامه ایجاد شده و سپس در پایان کار برنامه تخریب شود. انجام اینکار در برنامه‌های معمولی ویندوزی (WPF،WinForms و ...)، ساده است اما در محیط Stateless وب و برنامه‌های ASP.Net ، نیاز به راه حلی ویژه وجود خواهد داشت و تمرکز اصلی این مقاله حول مدیریت صحیح سشن فکتوری در برنامه‌های ASP.Net است.

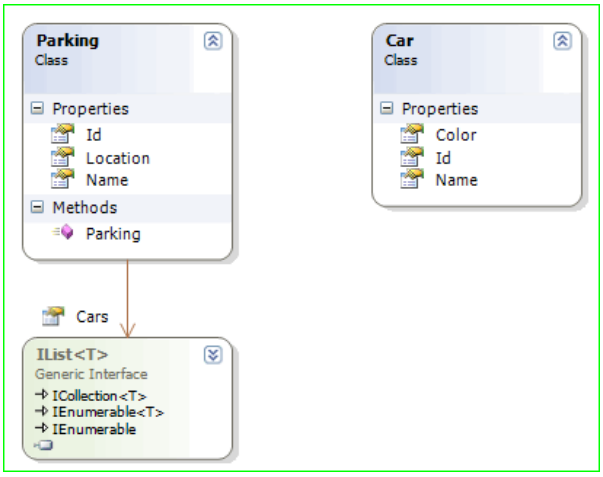
برای پیاده سازی شیء سشن فکتوری به صورتی که یکبار در طول برنامه ایجاد شود و بارها مورد استفاده قرار گیرد باید از یکی از الگوهای معروف طراحی برنامه نویسی شیء گرا به نام Singleton Pattern استفاده کرد. پیاده سازی نمونه‌ی thread safe آن که در برنامه‌های ذاتا چند ریسمانی وب و همچنین برنامه‌های معمولی ویندوزی می‌تواند مورد استفاده قرار گیرد، در آدرس ذیل قابل مشاهده است:

[Implementing the Singleton Pattern in C#]

از پنجمین روش ذکر شده در این مقاله جهت ایجاد یک singleton lazy, lock-free, thread-safe استفاده خواهیم کرد.

بررسی مدل برنامه

در این مدل ساده ما یک یا چند پارکینگ داریم که در هر پارکینگ یک یا چند خودرو می‌توانند پارک شوند.

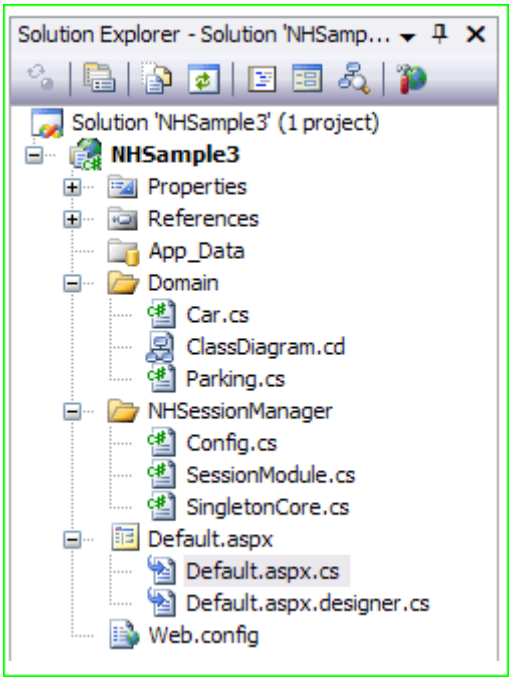


یک برنامه ASP.Net را آغاز کرده و ارجاعاتی را به اسمبلی‌های زیر به آن اضافه نمائید:

FluentNHibernate.dll
NHibernate.dll
NHibernate.ByteCode.Castle.dll
NHibernate.Linq.dll

و همچنین ارجاعی به اسمبلی استاندارد System.Data.Services.dll دات نت فریم ورک سه و نیم

تصویر نهایی پروژه ما به شکل زیر خواهد بود:



پروژه ما دارای یک پوشه `domain` **، تعریف کننده موجودیت‌های برنامه جهت تهیه نگاشت‌های لازم از روی آن‌ها است. سپس یک پوشه جدید را به نام** `NHSessionManager` **به آن جهت ایجاد یک** `Http module` **مدیریت کننده سشن‌های** `NHibernate` **در برنامه اضافه خواهیم کرد.**

ساختار دومین برنامه (مطابق کلاس دیاگرام فوق):

```
namespace NHSample3.Domain
{
    public class Car
    {
        public virtual int Id { get; set; }
        public virtual string Name { get; set; }
        public virtual string Color { get; set; }
    }
}
using System.Collections.Generic;

namespace NHSample3.Domain
{
  public class Parking
  {
      public virtual int Id { get; set; }
      public virtual string Name { get; set; }
      public virtual string Location { get; set; }
      public virtual IList<Car> Cars { get; set; }

      public Parking()
      {
          Cars = new List<Car>();
      }
  }
}
```

در این قسمت قصد داریم Http Module **ایی را جهت مدیریت سشن‌های** NHibernate **ایجاد نمائیم.**

در ابتدا کلاس Config **را در پوشه مدیریت سشن** NHibernate **با محتویات زیر ایجاد کنید:**

```
using FluentNHibernate.Automapping;
using FluentNHibernate.Cfg;
using FluentNHibernate.Cfg.Db;
using NHibernate.Tool.hbm2ddl;

namespace NHSessionManager
{
    public class Config
    {
        public static FluentConfiguration GetConfig()
        {
            return
              Fluently.Configure()
                      .Database(
                         MsSqlConfiguration
                           .MsSql2008
                           .ConnectionString(x =>
x.FromConnectionStringWithKey("DbConnectionString"))
                          )
                        .ExposeConfiguration(
                             x =>
x.SetProperty("current_session_context_class", "managed_web")
                         )
                        .Mappings(
                          m => m.AutoMappings.Add(
                             new AutoPersistenceModel()
                                 .Where(x => x.Namespace.EndsWith("Domain"))

.AddEntityAssembly(typeof(NHSample3.Domain.Car).Assembly))
                                 );
        }

        public static void CreateDb()
        {
            bool script = false;//آیا خروجي در کنسول هم نمایش داده شود
            bool export = true;//آیا روي بر دیتابیس هم اجرا شود
            bool dropTables = false;//آیا جداول موجود دراپ شوند
            new SchemaExport(GetConfig().BuildConfiguration()).Execute(script,
export, dropTables);
        }
    }
}
```

با این کلاس در قسمت‌های قبل آشنا شده‌اید. در این کلاس با کمک امکانات Auto mapping **موجود در** Fluent Nhibernate **(مطلب قسمت قبلی این سری آموزشی) اقدام به تهیه نگاشت‌های خودکار از کلاس‌های قرار گرفته در پوشه دومین خود خواهیم کرد (فضای نام این پوشه به دومین ختم می‌شود که در متد** GetConfig **مشخص است).**

دو نکته جدید در متد GetConfig **وجود دارد:**

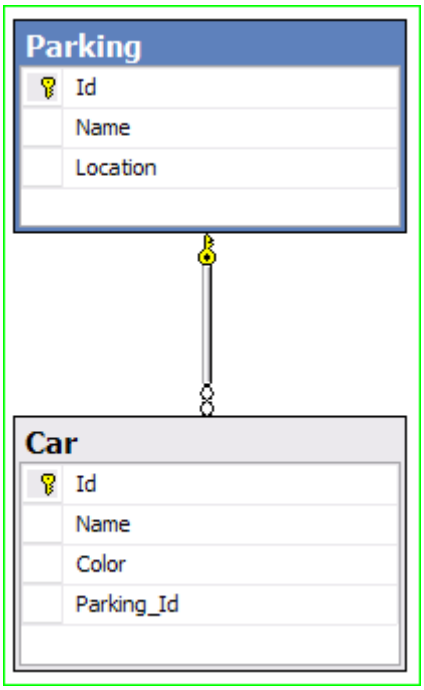
الف) استفاده از متد FromConnectionStringWithKey **، بجای تعریف مستقیم کانکشن استرینگ در متد مذکور که روشی است توصیه شده. به این صورت فایل وب کانفیگ ما باید دارای تعریف کلید مشخص شده در متد** GetConfig **به نام** DbConnectionString **باشد:**

```
<connectionStrings>
  <!--NHSessionManager-->
  <add name="DbConnectionString"
         connectionString="Data Source=(local);Initial
Catalog=HelloNHibernate;Integrated Security = true" />
</connectionStrings>
```

ب) قسمت ExposeConfiguration آن نیز جدید است.

در اینجا به AutoMapper خواهیم گفت که قصد داریم از امکانات مدیریت سشن مخصوص وب فریم ورک NHibernate استفاده کنیم. فریم ورک NHibernate دارای کلاسی است به نام NHibernate.Context.ManagedWebSessionContext که جهت مدیریت سشن‌های خود در پروژه‌های وب ASP.Net پیش بینی کرده است و از این متد در Http module ایی که ایجاد خواهیم کرد جهت ردگیری سشن جاری آن کمک خواهیم گرفت.

اگر متد CreateDb را فراخوانی کنیم، جداول نگاشت شده به کلاس‌های پوشه دومین برنامه، به صورت خودکار ایجاد خواهند شد که دیتابیس دیاگرام آن به صورت زیر می‌باشد:



سپس کلاس SingletonCore را جهت تهیه تنها و تنها یک وهله از شیء سشن فکتوری در کل برنامه ایجاد خواهیم کرد (همانطور که عنوان شده، ایده پیاده سازی این کلاس safe thread ، از مقاله معرفی شده در ابتدای بحث گرفته شده است):

```
using NHibernate;

namespace NHSessionManager
{
    /// <summary>
    /// lazy, lock-free, thread-safe singleton
    /// </summary>
    public class SingletonCore
    {
        private readonly ISessionFactory _sessionFactory;

        SingletonCore()
        {
            _sessionFactory = Config.GetConfig().BuildSessionFactory();
        }

        public static SingletonCore Instance
```

```
        {
            get
            {
                return Nested.instance;
            }
        }

        public static ISession GetCurrentSession()
        {
            return Instance._sessionFactory.GetCurrentSession();
        }

        public static ISessionFactory SessionFactory
        {
            get { return Instance._sessionFactory; }
        }

        class Nested
        {
            // Explicit static constructor to tell C# compiler
            // not to mark type as beforefieldinit
            static Nested()
            {
            }

            internal static readonly SingletonCore instance = new
SingletonCore();
        }
    }
}
```

اکنون می‌توان از این Singleton object **جهت تهیه یک** Http Module **کمک گرفت. برای این منظور کلاس** SessionModule **را به برنامه اضافه کنید:**

```
using System;
using System.Web;
using NHibernate;
using NHibernate.Context;

namespace NHSessionManager
{
    public class SessionModule : IHttpModule
    {
        public void Dispose()
        { }

        public void Init(HttpApplication context)
        {
            if (context == null)
                throw new ArgumentNullException("context");

            context.BeginRequest += Application_BeginRequest;
            context.EndRequest += Application_EndRequest;
        }

        private void Application_BeginRequest(object sender, EventArgs e)
        {
            ISession session = SingletonCore.SessionFactory.OpenSession();
            ManagedWebSessionContext.Bind(HttpContext.Current, session);
```

```
            session.BeginTransaction();
        }

        private void Application_EndRequest(object sender, EventArgs e)
        {
            ISession session = ManagedWebSessionContext.Unbind(
                HttpContext.Current, SingletonCore.SessionFactory);
            if (session == null) return;

            try
            {
                if (session.Transaction != null &&
                    !session.Transaction.WasCommitted &&
                    !session.Transaction.WasRolledBack)
                {
                    session.Transaction.Commit();
                }
                else
                {
                    session.Flush();
                }
            }
            catch (Exception)
            {
                session.Transaction.Rollback();
            }
            finally
            {
                if (session != null && session.IsOpen)
                {
                    session.Close();
                    session.Dispose();
                }
            }
        }
    }
}
```

کلاس فوق کار پیاده سازی اینترفیس IHttpModule **را جهت دخالت صریح در** handling pipeline request **برنامه** ASP.Net **جاری انجام می‌دهد. در این کلاس مدیریت متدهای استاندارد** Application_BeginRequest **و** Application_EndRequest **به صورت خودکار صورت می‌گیرد.**

در متد Application_BeginRequest **، در ابتدای هر درخواست یک سشن جدید ایجاد و به مدیریت سشن وب** NHibernate **بایند می‌شود، همچنین یک تراکنش نیز آغاز می‌گردد. سپس در پایان درخواست، این انقیاد فسخ شده و تراکنش کامل می‌شود، همچنین کار پاکسازی اشیاء نیز صورت خواهد گرفت.**

با توجه به این موارد، دیگر نیازی به ذکر using **جهت** dispose **کردن سشن جاری در کدهای ما نخواهد بود، زیرا در پایان هر درخواست اینکار به صورت خودکار صورت می‌گیرد. همچنین نیازی به ذکر تراکنش نیز نمی‌باشد، چون مدیریت آن‌را خودکار کرده‌ایم.**

جهت استفاده از این module Http **تهیه شده باید چند سطر زیر را به وب کانفیگ برنامه اضافه کرد:**

```
<httpModules>
  <!--NHSessionManager-->
  <add name="SessionModule" type="NHSessionManager.SessionModule"/>
</httpModules>
```

بدیهی است اگر نخواهید از Http module **استفاده کنید باید این کدها را در فایل** Global.asax **برنامه قرار دهید.**

اکنون مثالی از نحوه‌ی استفاده از امکانات فراهم شده فوق به صورت زیر می‌تواند باشد:

ابتدا کلاس `ParkingContext` **را جهت مدیریت مطلوب‌تر** `LINQ to NHibernate` **تشکیل می‌دهیم.**

```
using System.Linq;
using NHibernate;
using NHibernate.Linq;
using NHSample3.Domain;

namespace NHSample3
{
    public class ParkingContext : NHibernateContext
    {
        public ParkingContext(ISession session)
            : base(session)
        { }

        public IOrderedQueryable<Car> Cars
        {
            get { return Session.Linq<Car>(); }
        }

        public IOrderedQueryable<Parking> Parkings
        {
            get { return Session.Linq<Parking>(); }
        }
    }
}
```

سپس در فایل `Default.aspx.cs` **برنامه ، برای نمونه تعدادی رکورد را افزوده و نتیجه را در یک گرید ویوو نمایش خواهیم داد:**

```
using System;
using System.Collections.Generic;
using System.Linq;
using NHibernate;
using NHSample3.Domain;
using NHSessionManager;

namespace NHSample3
{
    public partial class _Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            //نیاز صورت در دیتابیس ایجاد
            //Config.CreateDb();

            //دیتابیس در رکورد سري يك ثبت
            ISession session = SingletonCore.GetCurrentSession();

            Car car1 = new Car() { Name = "رنو", Color = "مشکلي" };
            session.Save(car1);
            Car car2 = new Car() { Name = "پژو", Color = "سفید" };
            session.Save(car2);

            Parking parking1 = new Parking()
            {
                Location = "نظر مورد پارکینگ آدرس",
```

```
                Name = "یک پارکینگ",
                Cars = new List<Car> { car1, car2 }
            };

            session.Save(parking1);

            //ویوو گرید یک در حاصل نمایش
            ParkingContext db = new ParkingContext(session);
            var query = from x in db.Cars select new { CarName = x.Name,
CarColor = x.Color };
            GridView1.DataSource = query.ToList();
            GridView1.DataBind();
        }
    }
}
```

مدیریت سشن فکتوري در برنامه‌هاي غیر وب

در برنامه‌های ویندوزی مانند WPF ،WinForms و غیره، تا زمانیکه یک فرم باز باشد، کل فرم و اشیاء مرتبط با آن به یکباره تخریب نخواهند شد، اما در یک برنامه ASP.Net جهت حفظ منابع سرور در یک محیط چند کاربره، پس از پایان نمایش یک صفحه وب، اثری از آثار اشیاء تعریف شده در کدهای آن صفحه در سرور وجود نداشته و همگی بلافاصله تخریب می‌شوند. به همین جهت بحث‌های ویژه state management در ASP.Net در اینباره مطرح است و مدیریت ویژه‌ای باید روی آن صورت گیرد که در قسمت قبل مطرح شد.
از بحث فوق، تنها استفاده از کلاس‌های Config و SingletonCore ، جهت استفاده و مدیریت بهینه‌ی سشن فکتوری در برنامه‌های ویندوزی کفایت می‌کنند.

دریافت سورس برنامه قسمت هفتم

خلاصه‌اي از آغاز به کار باNHibernate

اگر شش یا هفت قسمت قبل را بخواهیم به صورت سریع مرور کنیم می‌توان به ویدیوی زیر مراجعه کرد:

Getting Started with NHibernate

در طی یک ربع، خیلی سریع به دریافت فایل‌های لازم، ایجاد یک پروژه جدید، افزودن ارجاعات لازم، استفاده از fluent NHibernate برای ساخت نگاشت‌ها و سپس استفاده از LINQ to NHibernate برای کوئری گرفتن از اطلاعات دیتابیس اشاره کرده است (که از این لحاظ کاملا به روز است).

قسمت هشتم

معرفي الگوي Repository

روش متداول کار با فناوری‌های مختلف دسترسی به داده‌ها عموما بدین شکل است:
الف) یافتن رشته اتصالی رمزنگاری شده به دیتابیس از یک فایل کانفیگ (در یک برنامه اصولی البته!)
ب) باز کردن یک اتصال به دیتابیس
ج) ایجاد اشیاء Command برای انجام عملیات مورد نظر
د) اجرا و فراخوانی اشیاء مراحل قبل
ه) بستن اتصال به دیتابیس و آزاد سازی اشیاء

اگر در برنامه‌های یک تازه کار به هر محلی از برنامه او دقت کنید این 5 مرحله را می‌توانید مشاهده کنید. همه جا! قسمت ثبت، قسمت جستجو، قسمت نمایش و ...
مشکلات این روش:
1- حجم کارهای تکراری انجام شده بالا است. اگر قسمتی از فناوری دسترسی به داده‌ها را به اشتباه درک کرده باشد، پس از مطالعه بیشتر و مشخص شدن نحوه‌ی رفع مشکل، قسمت عمده‌ای از برنامه را باید اصلاح کند (زیرا کدهای تکراری همه جای آن پراکنده‌اند).
2- برنامه نویس هر بار باید این مراحل را به درستی انجام دهد. اگر در یک برنامه بزرگ تنها قسمت آخر در یکی از مراحل کاری فراموش شود دیر یا زود برنامه تحت فشار کاری بالا از کار خواهد افتاد (و متاسفانه این مساله بسیار شایع است).
3- برنامه منحصرا برای یک نوع دیتابیس خاص تهیه خواهد شد و تغییر این رویه جهت استفاده از دیتابیسی دیگر (مثلا کوچ برنامه از اکسس به اس کیوال سرور)، نیازمند بازنویسی کل برنامه می‌باشد.
و ...

همین برنامه نویس پس از مدتی کار به این نتیجه می‌رسد که باید برای این‌کارهای متداول، یک لایه و کلاس دسترسی به داده‌ها را تشکیل دهد. اکنون هر قسمتی از برنامه برای کار با دیتابیس باید با این کلاس مرکزی که انجام کارهای متداول با دیتابیس را خلاصه می‌کند، کار کند. به این صورت کد نویسی یک نواختی با حذف کدهای تکراری از سطح برنامه و همچنین بدون فراموش شدن قسمت مهمی از مراحل کاری، حاصل می‌گردد. در اینجا اگر روزی قرار شد از یک دیتابیس دیگر استفاده شود فقط کافی است یک کلاس برنامه تغییر کند و نیازی به بازنویسی کل برنامه نخواهد بود.

این روزها تشکیل این لایه دسترسی به داده‌ها (data access layer یا DAL) نیز مزموم است! و دلایل آن در مباحث چرا به یک ORM نیازمندیم برشمرده شده است. جهت کار با ORM ها نیز نیازمند یک لایه دیگر می‌باشیم تا یک سری اعمال متداول با آن‌هارا کپسوله کرده و از حجم کارهای تکراری خود بکاهیم. برای این منظور قبل از اینکه دست به اختراع بزنیم، بهتر است به الگوهای طراحی برنامه نویسی شیء گرا رجوع کرد و از رهنمودهای آن استفاده نمود.

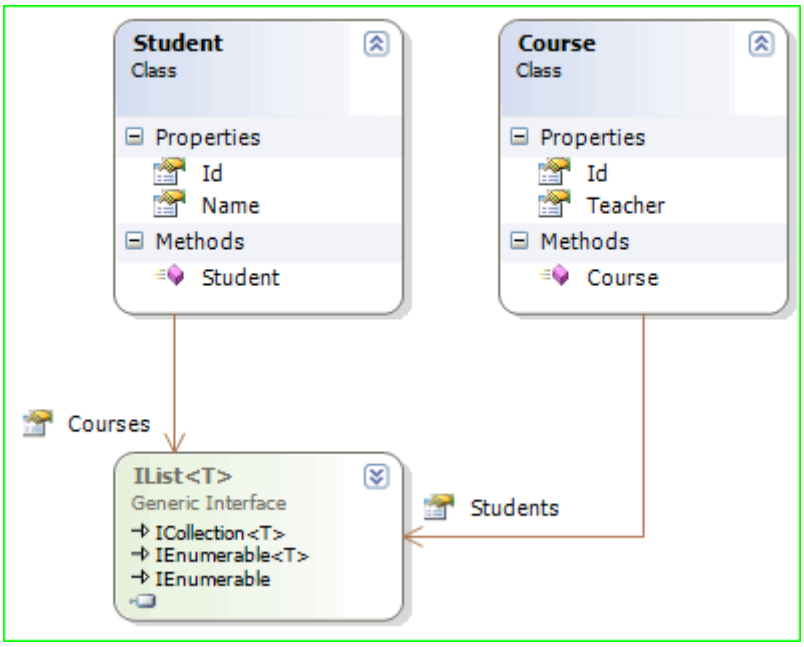
الگوی Repository یکی از الگوهای برنامه نویسی با مقیاس سازمانی است. با کمک این الگو لایه‌ای بر روی لایه نگاشت اشیاء برنامه به دیتابیس تشکیل شده و عملا برنامه را مستقل از نوع ORM مورد استفاده می‌کند. به این صورت هم از تشکیل یک سری کدهای تکراری در سطح برنامه جلوگیری شده و هم از وابستگی بین مدل برنامه و لایه دسترسی به داده‌ها (که در اینجا همان NHibernate می‌باشد) جلوگیری می‌شود. الگوی Repository (مخزن)، کار ثبت، حذف، جستجو و به روز رسانی داده‌ها را با ترجمه آن‌ها به روش‌های بومی مورد استفاده توسط ORM مورد نظر، کپسوله می‌کند. به این شکل شما می‌توانید یک الگوی مخزن عمومی را برای کارهای خود تهیه کرده و به سادگی از یک ORM به ORM دیگر کوچ کنید؛ زیرا کدهای برنامه شما به هیچ ORM خاصی گره نخورده و این عملیات بومی کار با ORM توسط لایه‌ای که توسط الگوی مخزن تشکیل شده، صورت گرفته است.

طراحی کلاس مخزن باید شرایط زیر را برآورده سازد:
الف) باید یک طراحی عمومی داشته باشد و بتواند در پروژه‌های متعددی مورد استفاده مجدد قرار گیرد.

ب) باید با سیستمی از نوع اول طراحی و کد نویسی و بعد کار با دیتابیس، سازگاری داشته باشد.
ج) باید امکان انجام آزمایشات واحد را سهولت بخشد.
د) باید وابستگی کلاس‌های دومین برنامه را به زیر ساخت ORM مورد استفاده قطع کند (اگر سال بعد به این نتیجه رسیدید که ORM ایی به نام XYZ برای کار شما بهتر است، فقط پیاده سازی این کلاس باید تغییر کند و نه کل برنامه).
ه) باید استفاده از کوئری‌هایی از نوع strongly typed را ترویج کند (مثل کوئری‌هایی از نوع LINQ).

بررسي مدل برنامه

مدل این قسمت (برنامه NHSample4 از نوع کنسول با همان ارجاعات متداول ذکر شده در قسمت‌های قبل)، از نوع many-to-many می‌باشد. در اینجا یک واحد درسی توسط چندین دانشجو می‌تواند اخذ شود یا یک دانشجو می‌تواند چندین واحد درسی را اخذ نماید که برای نمونه کلاس دیاگرام و کلاس‌های متشکل آن به شکل زیر خواهند بود:




```
using System.Collections.Generic;

namespace NHSample4.Domain
{
    public class Course
    {
        public virtual int Id { get; set; }
        public virtual string Teacher { get; set; }
        public virtual IList<Student> Students { get; set; }

        public Course()
        {
            Students = new List<Student>();
        }
    }
}
```

```
using System.Collections.Generic;

namespace NHSample4.Domain
{
```

```
 public class Student
 {
   public virtual int Id { get; set; }
   public virtual string Name { get; set; }
   public virtual IList<Course> Courses { get; set; }

   public Student()
   {
       Courses = new List<Course>();
   }
 }
}
```

کلاس کانفیگ برنامه جهت ایجاد نگاشت‌ها و سپس ساخت دیتابیس متناظر

```
using FluentNHibernate.Automapping;
using FluentNHibernate.Cfg;
using FluentNHibernate.Cfg.Db;
using NHibernate.Tool.hbm2ddl;

namespace NHSessionManager
{
    public class Config
    {
        public static FluentConfiguration GetConfig()
        {
            return
              Fluently.Configure()
                      .Database(
                         MsSqlConfiguration
                           .MsSql2008
                           .ConnectionString(x =>
x.FromConnectionStringWithKey("DbConnectionString"))
                          )
                        .Mappings(
                          m => m.AutoMappings.Add(
                             new AutoPersistenceModel()
                                 .Where(x => x.Namespace.EndsWith("Domain"))

.AddEntityAssembly(typeof(NHSample4.Domain.Course).Assembly))
                                 .ExportTo(System.Environment.CurrentDirectory)
                                 );
        }

        public static void CreateDb()
        {
            bool script = false;//شود داده نمایش هم کنسول در خروجي آیا
            bool export = true;//شود اجرا هم دیتابیس روي بر آیا
            bool dropTables = false;//شوند دراپ موجود جداول آیا
            new SchemaExport(GetConfig().BuildConfiguration()).Execute(script,
export, dropTables);
        }
    }
}
```

چند نکته در مورد این کلاس:

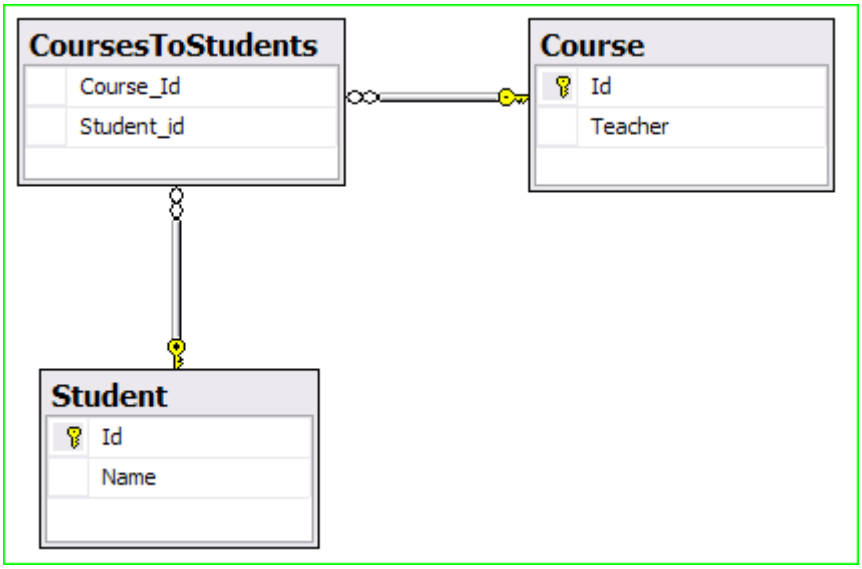
الف) با توجه به اینکه برنامه از نوع ویندوزی است، برای مدیریت صحیح کانکشن استرینگ، فایل App.Config **را به برنامه افروده و محتویات آن‌را به شکل زیر تنظیم می‌کنیم (تا کلید** DbConnectionString **توسط متد** GetConfig **مورد استفاده قرارگیرد):**

```
<?xml version="1.0" encoding="utf-8" ?>
```

```
<configuration>
  <connectionStrings>
    <!--NHSessionManager-->
    <add name="DbConnectionString"
        connectionString="Data Source=(local);Initial
Catalog=HelloNHibernate;Integrated Security = true"/>
  </connectionStrings>
</configuration>
```

ب) در NHibernate سنتی (!) کار ساخت نگاشت‌ها توسط یک سری فایل xml صورت می‌گیرد که با معرفی فریم ورک Fluent NHibernate و استفاده از قابلیت‌های Auto Mapping آن، این‌کار با سهولت و دقت هر چه تمام‌تر قابل انجام است که توضیحات نحوه‌ی انجام آن‌را در قسمت‌های قبل مطالعه فرمودید. اگر نیاز بود تا این فایل‌های XML نیز جهت بررسی شخصی ایجاد شوند، تنها کافی است از متد ExportTo آن همانگونه که در متد GetConfig استفاده شده، کمک گرفته شود. به این صورت پس از ایجاد خودکار نگاشت‌ها، فایل‌های XML متناظر نیز در مسیری که به عنوان آرگومان متد ExportTo مشخص گردیده است، تولید خواهند شد (دو فایل NHSample4.Domain.Course.hbm.xml و NHSample4.Domain.Student.hbm.xml را در پوشه‌ای که محل اجرای برنامه است خواهید یافت).

با فراخوانی متد CreateDb این کلاس، پس از ساخت خودکار نگاشت‌ها، database schema متناظر، در دیتابیسی که توسط کانکشن استرینگ برنامه مشخص شده، ایجاد خواهد شد که دیتابیس دیاگرام آن‌را در شکل ذیل مشاهده می‌نمائید (جداول دانشجویان و واحدها هر کدام به صورت موجودیتی مستقل ایجاد شده که ارجاعات آن‌ها در جدولی سوم نگهداری می‌شود).



پیاده سازي الگوي مخزن

اینترفیس عمومی الگوی مخزن به شکل زیر می‌تواند باشد:

```
using System;
using System.Linq;
using System.Linq.Expressions;

namespace NHSample4.NHRepository
{
    //Repository Interface
    public interface IRepository<T>
    {
        T Get(object key);

        T Save(T entity);
```

```
        T Update(T entity);
        void Delete(T entity);

        IQueryable<T> Find();
        IQueryable<T> Find(Expression<Func<T, bool>> predicate);
    }
}
```

سپس پیاده سازی آن با توجه به کلاس `SingletonCore` **ایی که در قسمت قبل تهیه کردیم (جهت مدیریت صحیح سشن فکتوری)، به صورت زیر خواهد بود.**

این کلاس کار آغاز و پایان تراکنش‌ها را نیز مدیریت کرده و جهت سهولت کار اینترفیس `IDisposable` **را نیز پیاده سازی می‌کند :**

```
using System;
using System.Linq;
using NHSessionManager;
using NHibernate;
using NHibernate.Linq;

namespace NHSample4.NHRepository
{
    public class Repository<T> : IRepository<T>, IDisposable
    {
        private ISession _session;
        private bool _disposed = false;

        public Repository()
        {
            _session = SingletonCore.SessionFactory.OpenSession();
            BeginTransaction();
        }

        ~Repository()
        {
            Dispose(false);
        }

        public T Get(object key)
        {
            if (!isSessionSafe) return default(T);

            return _session.Get<T>(key);
        }

        public T Save(T entity)
        {
            if (!isSessionSafe) return default(T);

            _session.Save(entity);
            return entity;
        }

        public T Update(T entity)
        {
            if (!isSessionSafe) return default(T);

            _session.Update(entity);
            return entity;
        }
```

```
        public void Delete(T entity)
        {
            if (!isSessionSafe) return;

            _session.Delete(entity);
        }

        public IQueryable<T> Find()
        {
            if (!isSessionSafe) return null;

            return _session.Linq<T>();
        }

        public IQueryable<T> Find(System.Linq.Expressions.Expression<Func<T,
bool>> predicate)
        {
            if (!isSessionSafe) return null;

            return Find().Where(predicate);
        }

        void Commit()
        {
            if (!isSessionSafe) return;

            if (_session.Transaction != null &&
                _session.Transaction.IsActive &&
                !_session.Transaction.WasCommitted &&
                !_session.Transaction.WasRolledBack)
            {
                _session.Transaction.Commit();
            }
            else
            {
                _session.Flush();
            }
        }

        void Rollback()
        {
            if (!isSessionSafe) return;

            if (_session.Transaction != null && _session.Transaction.IsActive)
            {
                _session.Transaction.Rollback();
            }
        }

        private bool isSessionSafe
        {
            get
            {
                return _session != null && _session.IsOpen;
            }
        }

        void BeginTransaction()
        {
```

```
            if (!isSessionSafe) return;

            _session.BeginTransaction();
        }


        public void Dispose()
        {
            Dispose(true);
            // tell the GC that the Finalize process no longer needs to be run
for this object.
            GC.SuppressFinalize(this);
        }

        protected virtual void Dispose(bool disposeManagedResources)
        {
            if (_disposed) return;
            if (!disposeManagedResources) return;
            if (!isSessionSafe) return;

            try
            {
                Commit();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.ToString());
                Rollback();
            }
            finally
            {
                if (isSessionSafe)
                {
                    _session.Close();
                    _session.Dispose();
                }
            }

            _disposed = true;
        }
    }
}
```

اکنون جهت استفاده از این کلاس مخزن به شکل زیر می‌توان عمل کرد:

```
using System;
using System.Collections.Generic;
using NHSample4.Domain;
using NHSample4.NHRepository;

namespace NHSample4
{
    class Program
    {
        static void Main(string[] args)
        {
            //نیاز صورت در دیتابیس ایجاد
            //NHSessionManager.Config.CreateDb();

            //کنیممی اضافه را دانشجو یک ابتدا
```

```
            Student student = null;
            using (var studentRepo = new Repository<Student>())
            {
                student = studentRepo.Save(new Student() { Name = "Vahid" });
            }

            //کنيم مي برقرار دانشجو با راآن ارجاع و افزوده را واحد يك اكنون
            using (var courseRepo = new Repository<Course>())
            {
                courseRepo.Save(new Course()
                {
                    Teacher = "Shams",
                    Students = new List<Student>() { student }
                });
            }

            //دهيممي نمايش را خاص استادي دروس شماره سپس
            using (var courseRepo = new Repository<Course>())
            {
                var query = courseRepo.Find(t => t.Teacher == "Shams");

                foreach (var courseItem in query)
                    Console.WriteLine(courseItem.Id);
            }

            Console.WriteLine("Press a key...");
            Console.ReadKey();
        }
    }
}
```

همانطور که ملاحظه می‌کنید در این سطح دیگر برنامه هیچ درکی از ORM مورد استفاده ندارد و پیاده سازی نحوه‌ی تعامل با NHibernate در پس کلاس مخزن مخفی شده است. کار آغاز و پایان تراکنش‌ها به صورت خودکار مدیریت گردیده و همچنین آزاد سازی منابع را نیز توسط اینترفیس IDisposable مدیریت می‌کند. به این صورت امکان فراموش شدن یک سری از اعمال متداول به حداقل رسیده، میزان کدهای تکراری برنامه کم شده و همچنین هر زمانیکه نیاز بود، صرفا با تغییر پیاده سازی کلاس مخزن می‌توان به ORM دیگری کوچ کرد؛ بدون اینکه نیازی به بازنویسی کل برنامه وجود داشته باشد.

دریافت سورس برنامه قسمت هشتم

قسمت نهم

استفاده از Log4Net جهت ثبت خروجی‌های SQL حاصل از NHibernate

هنگام استفاده از NHibernate**، پس از افزودن ارجاعات لازم به اسمبلی‌های مورد نیاز آن به برنامه، یکی از اسمبلی‌هایی که به پوشه** build **برنامه به صورت خودکار کپی می‌شود، فایل** log4net.dll **است (حتی اگر ارجاعی را به آن اضافه نکرده باشیم) که جهت ثبت وقایع مرتبط با** NHibernate **مورد استفاده قرار می‌گیرد. خوب اگر مجبوریم که این وابستگی کتابخانه** NHibernate **را نیز در پروژه‌های خود داشته باشیم، چرا از آن استفاده نکنیم؟!**

شرح مفصل استفاده از این کتابخانه سورس باز را در سایت اصلی آن می‌توان مشاهده کرد:

Log4Net

برای اینکه از این کتابخانه در برنامه خود جهت ثبت عبارات SQL **تولیدی توسط** NHibernate **استفاده کنیم، باید مراحل زیر طی شوند:**

الف) ارجاعی را به اسمبلی log4net.dll **اضافه نمائید (کنار اسمبلی** NHibernate **در پوشه** build **برنامه باید موجود باشد)**

ب) فایل app.config **برنامه را (برنامه ویندوزی) به صورت زیر ویرایش کرده و چند سطر مربوطه را اضافه نمائید (در مورد برنامه‌های وب هم به همین شکل است. فایل** web.config **تنظیم شده و سپس تنظیمات** log4net **را قبل از بسته شدن تگ** configuration **اضافه نمائید) :**

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="log4net"
             type="log4net.Config.Log4NetConfigurationSectionHandler,log4net"
/>
  </configSections>

  <connectionStrings>
    <!--NHSessionManager-->
    <add name="DbConnectionString"
         connectionString="Data Source=(local);Initial
Catalog=HelloNHibernate;Integrated Security = true"/>
  </connectionStrings>

  <log4net>
    <appender name="rollingFile"
              type="log4net.Appender.RollingFileAppender,log4net" >
      <param name="File" value="NHibernate_Log.txt" />
      <param name="AppendToFile" value="true" />
      <param name="DatePattern" value="yyyy.MM.dd" />
      <rollingStyle value="Size" />
      <maxSizeRollBackups value="10" />
      <maximumFileSize value="500KB" />
      <staticLogFileName value="true" />
      <layout type="log4net.Layout.PatternLayout,log4net">
        <conversionPattern value="%d %p %m%n" />
      </layout>
    </appender>
    <logger name="NHibernate.SQL">
      <level value="ALL" />
      <appender-ref ref="rollingFile" />
    </logger>
  </log4net>

</configuration>
```

ج) سپس باید فراخوانی زیر نیز در ابتدای کار برنامه صورت گیرد:

```
log4net.Config.XmlConfigurator.Configure();
```

در یک برنامه ASP.Net این فراخوانی باید در Application_Start فایل Global.asax.cs صورت گیرد.
یا در یک برنامه از نوع WinForms تنها کافی است سطر زیر را به فایل AssemblyInfo.cs برنامه اضافه کرد:

```
// Configure log4net using the .config file
[assembly: log4net.Config.XmlConfigurator(Watch = true)]
```

یا این سطر را به فایل Global.asax.cs یک برنامه ASP.Net نیز می‌توان اضافه کرد. آن، با کمک Watch=true تغییرات فایل کانفیگ را تحت نظر داشته و هر بار که تغییر کند بلافاصله، تغییرات جدید را اعمال خواهد کرد.

د) هنگام استفاده از کتابخانه Fluent NHibernate حتما باید متد ShowSql در جایی که دیتابیس برنامه را تنظیم می‌کنیم (Fluently.Configure().Database) ذکر گردد (که نمونه آن‌را در مثال‌های قسمت‌های قبل ملاحظه کرده‌اید).

توضیحاتي در مورد تنظیمات فوق:

configSections حتما باید در ابتدای فایل app.config ذکر شود و گرنه برنامه کار نخواهد کرد.
سپس کانکشن استرینگ مورد استفاده در قسمت کانفیگ برنامه ذکر شده است.
در ادامه تنظیمات استاندارد مربوط به log4net را مشاهده می‌کنید.
در تنظیمات این کتابخانه، appender مشخص کننده محل ثبت وقایع است. زمانیکه که از RollingFileAppender استفاده کنیم، اطلاعات را در یک سری فایل ذخیره خواهد کرد (امکان ثبت وقایع در EventLog ویندوز، ارسال از طریق ایمیل و غیره نیز میسر است که جهت توضیحات بیشتر می‌توان به مستندات آن رجوع نمود).
سپس نام فایلی که اطلاعات وقایع در آن ثبت خواهند شد ذکر شده است (برای مثال NHibernate_Log.txt)، در ادامه مشخص گردیده که اطلاعات باید هر بار به این فایل Append و اضافه شوند. سطرهای بعدی مشخص می‌کنند که هر زمانیکه این لاگ فایل به 10 مگابایت رسید، یک فایل جدید تولید کن و هر بار 10 فایل آخر را نگه دار و مابقی فایل‌های قدیمی را حذف کن.
در قسمت PatternLayout مشخصات می‌کنیم که خروجی ثبت شده با چه فرمتی باشد. برای مثال یک سطر خروجی مطابق با تنظیمات فوق به شکل زیر خواهد بود:

```
2009-10-18 20:03:54,187 DEBUG INSERT INTO [Student] (Name) VALUES (@p0); select
SCOPE_IDENTITY();@p0 = 'Vahid'
```

در قسمت Logger یک نام دلخواه ذکر شده و میزان اطلاعاتی که باید درج شود، از طریق مقدار level مورد نظر، قابل تنظیم است که می‌تواند یکی از مقادیر ALL، DEBUG، INFO، WARN، ERROR، FATAL و یا OFF باشد. اینجا level در نظر گرفته شده ALL است که تمامی اطلاعات مرتبط با اعمال پشت صحنه NHibernate را لاگ خواهد کرد.
توسط appender-ref آن appender ایی را که در ابتدای کار تعریف و تنظیم کردیم، مشخص خواهیم کرد.

اگر هم با برنامه نویسی بخواهیم اطلاعاتی را به این لاگ فایل اضافه کنیم تنها کافی است بنویسیم:

```
log4net.LogManager.GetLogger("NHibernate.SQL").Info("test1");
```

اطلاعات بیشتر

قسمت دهم

آشنايي با كتابخانه Validator NHibernate

پروژه جدیدی به پروژه NHibernate Contrib **در سایت سورس فورج اضافه شده است به نام** NHibernate Validator **که از آدرس زیر قابل دریافت است:**

http://sourceforge.net/projects/nhcontrib/files/NHibernate.Validator

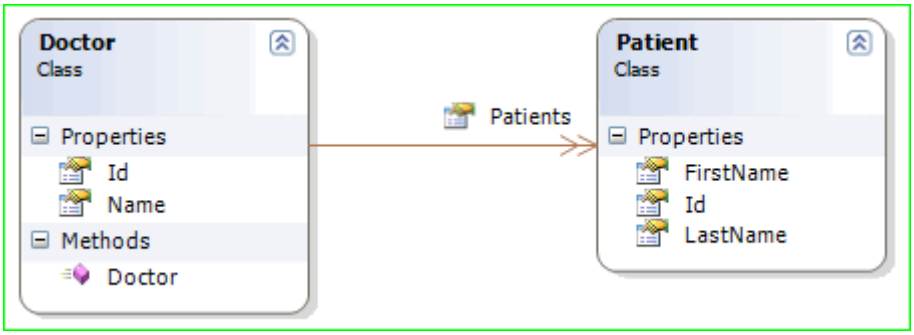
این پروژه که توسط Dario Quintana **توسعه یافته است، امکان اعتبار سنجی اطلاعات را پیش از افزوده شدن آن‌ها به دیتابیس به دو صورت دستی و یا خودکار و یکپارچه با** NHibernate **فراهم می‌سازد؛ که امروز قصد بررسی آن‌را داریم.**

كامپايل پروژه اعتبار سنجي NHibernate

پس از دریافت آخرین نگارش موجود کتابخانه NHibernate Validator **از سایت سورس فورج، فایل پروژه آن‌را در** VS.Net **گشوده و یکبار آن‌را کامپایل نمائید تا فایل اسمبلی** NHibernate.Validator.dll **حاصل گردد.**

بررسي مدل برنامه

در این مدل ساده، تعدادی پزشک داریم و تعدادی بیمار. در سیستم ما هر بیمار تنها توسط یک پزشک مورد معاینه قرار خواهد گرفت. رابطه آن‌ها را در کلاس دیاگرام زیر می‌توان مشاهده نمود:



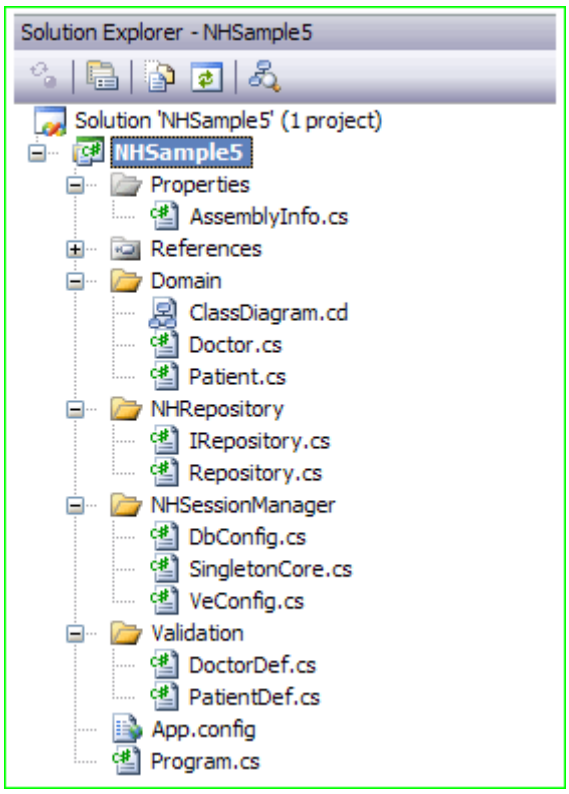
به این صورت پوشه دومین برنامه از کلاس‌های زیر تشکیل خواهد شد:

```
namespace NHSample5.Domain
{
    public class Patient
    {
        public virtual int Id { get; set; }
        public virtual string FirstName { get; set; }
        public virtual string LastName { get; set; }
    }
}

using System.Collections.Generic;

namespace NHSample5.Domain
{
   public class Doctor
   {
```

```
        public virtual int Id { get; set; }
        public virtual string Name { get; set; }
        public virtual IList<Patient> Patients { get; set; }

        public Doctor()
        {
            Patients = new List<Patient>();
        }
    }
}
```

برنامه این قسمت از نوع کنسول با ارجاعاتی به اسمبلی‌های FluentNHibernate.dll، log4net.dll، NHibernate.dll، NHibernate.ByteCode.Castle.dll، NHibernate.Linq.dll، NHibernate.Validator.dll و System.Data.Services.dll است.

ساختار کلی این پروژه را در شکل زیر مشاهده می‌کنید:



اطلاعات این برنامه بر مبنای NHRepository و NHSessionManager ایی است که در قسمت‌های قبل توسعه دادیم و پیشنیاز ضروری مطالعه آن می‌باشند (سورس پیوست شده شامل نمونه تکمیل شده این موارد نیز هست). همچنین از قسمت ایجاد دیتابیس از روی مدل نیز صرفنظر می‌شود و همانند قسمت‌های قبل است.

تعریف اعتبار سنجي دومين با كمك ويژگي‌ها (attributes)

فرض کنید می‌خواهیم بر روی طول نام و نام خانوادگی بیمار محدودیت قرار داده و آن‌ها را با کمک کتابخانه NHibernate Validator، اعتبار سنجی کنیم. برای این منظور ابتدا فضای نام NHibernate.Validator.Constraints به کلاس بیمار اضافه شده و سپس با کمک ویژگی‌هایی که در این کتابخانه تعریف شده‌اند می‌توان قیود خود را به خواص کلاس تعریف شده اعمال نمود که نمونه‌ای از آن را مشاهده می‌نمائید:

```
using NHibernate.Validator.Constraints;
```

```
namespace NHSample5.Domain
{
    public class Patient
    {
        public virtual int Id { get; set; }

        [Length(Min = 3, Max = 20, Message = "طول نام باید بین 3 و 20 کاراکتر باشد")]
        public virtual string FirstName { get; set; }

        [Length(Min = 3, Max = 60, Message = "طول نام خانوادگی باید بین 3 و 60 کاراکتر باشد")]
        public virtual string LastName { get; set; }
    }
}
```

اعمال این قیود از این جهت مهم هستند که نباید وقت برنامه و سیستم را با دریافت خطای نهایی از دیتابیس تلف کرد. آیا بهتر نیست قبل از اینکه اطلاعات به دیتابیس وارد شوند و رفت و برگشتی در شبکه صورت گیرد، مشخص گردد که این فیلد حتما نباید خالی باشد یا طول آن باید دارای شرایط خاصی باشد و امثال آن؟

مثالی دیگر:

جهت اجباری کردن و همچنین اعمال `Regular expressions` **برای اعتبار سنجی یک فیلد می‌توان دو ویژگی زیر را به بالای آن فیلد مورد نظر افزود:**

```
[NotNull]
[Pattern(Regex = "[A-Za-z0-9]+")]
```

تعریف اعتبار سنجي با كمك كلاس ValidationDef

راه دوم تعریف اعتبار سنجی، کمک گرفتن از کلاس `ValidationDef` **این کتابخانه و استفاده از روش** `fluent configuration` **است. برای این منظور، پوشه جدیدی را به برنامه به نام** `Validation` **اضافه خواهیم کرد و سپس دو کلاس** `DoctorDef` **و** `PatientDef` **را به آن به صورت زیر خواهیم افزود:**

```
using NHibernate.Validator.Cfg.Loquacious;
using NHSample5.Domain;

namespace NHSample5.Validation
{
    public class DoctorDef : ValidationDef<Doctor>
    {
        public DoctorDef()
        {
            Define(x => x.Name).LengthBetween(3, 50);
            Define(x => x.Patients).NotNullableAndNotEmpty();
        }
    }
}
```

```
using NHSample5.Domain;
using NHibernate.Validator.Cfg.Loquacious;

namespace NHSample5.Validation
{
   public class PatientDef : ValidationDef<Patient>
```

```
    {
        public PatientDef()
        {
            Define(x => x.FirstName)
                .LengthBetween(3, 20)
                .WithMessage("باشد كاراكتر 20 و 3 بين بايد نام طول");

            Define(x => x.LastName)
                .LengthBetween(3, 60)
                .WithMessage("باشد كاراكتر 60 و 3 بين بايد خانوادگي نام طول");
        }
    }
}
```

استفاده از قيودات تعريف شده به صورت دستي

می‌توان از این کتابخانه اعتبار سنجی به صورت مستقیم نیز اضافه کرد. روش انجام آن‌را در متد زیر مشاهده می‌نمائید.

```
    /// <summary>
    /// مستقيم صورت به ويژه سنجي اعتبار از استفاده
    /// هاويژگي از استفاده صورت در
    /// </summary>
    static void WithoutConfiguringTheEngine()
    {
        //معتبر غير بيمار يك تعريف
        var patient1 = new Patient() { FirstName = "V", LastName = "N" };
        var ve = new ValidatorEngine();
        var invalidValues = ve.Validate(patient1);
        if (invalidValues.Length == 0)
        {
            Console.WriteLine("patient1 is valid.");
        }
        else
        {
            Console.WriteLine("patient1 is NOT valid!");
            //فيلد هر به مربوط شده تعريف هاپيغام نمايش
            foreach (var invalidValue in invalidValues)
            {
                Console.WriteLine(
                    "{0}: {1}",
                    invalidValue.PropertyName,
                    invalidValue.Message);
            }
        }

        //اعمالي قيودات اساس بر معتبر بيمار يك تعريف
        var patient2 = new Patient() { FirstName = "وحيد", LastName = "نصيري" };
        if (ve.IsValid(patient2))
        {
            Console.WriteLine("patient2 is valid.");
        }
        else
        {
            Console.WriteLine("patient2 is NOT valid!");
        }
    }
```

ابتدا شیء `ValidatorEngine` **تعریف شده و سپس متد** `Validate` **آن بر روی شیء بیماری غیر معتبر فراخوانی می‌گردد. در صورتیکه این عتبار سنجی با موفقیت روبر نشود، خروجی این متد آرایه‌ای خواهد بود از فیلدهای غیرمعتبر به همراه پیغام‌هایی که برای آن‌ها**

تعریف کرده‌ایم. یا می‌توان به سادگی همانند بیمار شماره دو، تنها از متد IsValid آن نیز استفاده کرد.

در اینجا اگر سعی در اعتبار سنجی یک پزشک نمائیم، نتیجه‌ای حاصل نخواهد شد زیرا هنگام استفاده از کلاس ValidationDef، باید نگاشت لازم به این قیودات را نیز دقیقا مشخص نمود تا مورد استفاده قرار گیرد که نحوه‌ی انجام این عملیات را در متد زیر می‌توان مشاهده نمود.

```
        public static ValidatorEngine GetFluentlyConfiguredEngine()
        {
            var vtor = new ValidatorEngine();
            var configuration = new FluentConfiguration();
            configuration
                    .Register(
                        Assembly
                          .GetExecutingAssembly()
                          .GetTypes()
                          .Where(t =>
t.Namespace.Equals("NHSample5.Validation"))
                          .ValidationDefinitions()
                     )
                    .SetDefaultValidatorMode(ValidatorMode.UseExternal);
            vtor.Configure(configuration);
            return vtor;
        }
```

نکته مهم:

فراخوانی GetFluentlyConfiguredEngine نیز باید یکبار در طول برنامه صورت گرفته و سپس حاصل آن بارها مورد استفاده قرار گیرد. بنابراین نحوه‌ی صحیح دسترسی به آن باید حتما از طریق الگوی Singleton که در قسمت‌های قبل در مورد آن بحث شد، انجام شود.

استفاده از قیودات تعریف شده و سیستم اعتبار سنجي به صورت یکپارچه با NHibernate

کتابخانه NHibernate Validator زمانیکه با NHibernate یکپارچه گردد دو رخداد PreInsert و PreUpdate آن‌را به صورت خودکار تحت نظر قرار داده و پیش از اینکه اطلاعات ثبت و یا به روز شوند، ابتدا کار اعتبار سنجی خود را انجام داده و اگر اعتبار سنجی مورد نظر با شکست مواجه شود، با ایجاد یک exception از ادامه برنامه جلوگیری می‌کند. در این حالت استثنای حاصل شده از نوع InvalidStateException خواهد بود.

برای انجام این مرحله یکپارچه سازی ابتدا متد BuildIntegratedFluentlyConfiguredEngine را به شکل زیر باید فراخوانی نمائیم:

```
        /// <summary>
        /// شود گرفته کمك باید فکتوري سشن آغاز براي کانفیگ این از
        /// </summary>
        /// <param name="nhConfiguration"></param>
        public static void BuildIntegratedFluentlyConfiguredEngine(ref
Configuration nhConfiguration)
        {
            var vtor = new ValidatorEngine();
            var configuration = new FluentConfiguration();
            configuration
                    .Register(
```

```
                        Assembly
                            .GetExecutingAssembly()
                            .GetTypes()
                            .Where(t =>
t.Namespace.Equals("NHSample5.Validation"))
                            .ValidationDefinitions()
                        )
                    .SetDefaultValidatorMode(ValidatorMode.UseExternal)
                    .IntegrateWithNHibernate
                    .ApplyingDDLConstraints()
                    .And
                    .RegisteringListeners();
            vtor.Configure(configuration);

            //Registering of Listeners and DDL-applying here
            ValidatorInitializer.Initialize(nhConfiguration, vtor);
        }
```

این متد کار دریافت `Configuration` **مرتبط با** `NHibernate` **را جهت اعمال تنظیمات اعتبار سنجی به آن انجام می‌دهد. سپس از** `nhConfiguration` **تغییر یافته در این متد جهت ایجاد سشن فکتوری استفاده خواهیم کرد (در غیر اینصورت سشن فکتوری درکی از اعتبار سنجی‌های تعریف شده نخواهد داشت). اگر قسمت‌های قبل را مطالعه کرده باشید، کلاس** `SingletonCore` **را جهت مدیریت بهینه‌ی سشن فکتوری به خاطر دارید. این کلاس اکنون باید به شکل زیر وصله شود:**

```
        SingletonCore()
        {
            Configuration cfg = DbConfig.GetConfig().BuildConfiguration();
            VeConfig.BuildIntegratedFluentlyConfiguredEngine(ref cfg);
            //شود شروع کار باید سنجي اعتبار براي شده تنظيم كانفيگ همان با
            _sessionFactory = cfg.BuildSessionFactory();
        }
```

از این لحظه به بعد، نیاز به فراخوانی متدهای `Validate` **و یا** `IsValid` **نبوده و کار اعتبار سنجی به صورت خودکار و یکپارچه با** `NHibernate` **انجام می‌شود. لطفا به مثال زیر دقت بفرمائید:**

```
        /// <summary>
        /// خودكار و يكپارچه سنجي اعتبار از استفاده
        /// </summary>
        static void tryToSaveInvalidPatient()
        {
            using (Repository<Patient> repo = new Repository<Patient>())
            {
                try
                {
                    var patient1 = new Patient() { FirstName = "V", LastName =
"N" };

                    repo.Save(patient1);
                }
                catch (InvalidStateException ex)
                {
                    Console.WriteLine("Validation failed!");
                    foreach (var invalidValue in ex.GetInvalidValues())
                        Console.WriteLine(
                            "{0}: {1}",
                            invalidValue.PropertyName,
                            invalidValue.Message);
                    log4net.LogManager.GetLogger("NHibernate.SQL").Error(ex);
```

```
            }
        }
    }

    /// <summary>
    /// خودکار و یکپارچه سنجي اعتبار از استفاده
    /// </summary>
    static void tryToSaveValidPatient()
    {
        using (Repository<Patient> repo = new Repository<Patient>())
        {
            var patient1 = new Patient() { FirstName = "Vahid", LastName =
"Nasiri" };
            repo.Save(patient1);
        }
    }
```

در اینجا از کلاس `Repository` **که در قسمت‌های قبل توسعه دادیم، استفاده شده است. در متد** `tryToSaveInvalidPatient` **، بدلیل استفاده از تعریف بیماری غیرمعتبر، پیش از انجام عملیات ثبت، استثنایی حاصل شده و پیش از هرگونه رفت و برگشتی به دیتابیس، سیستم از بروز این مشکل مطلع خواهد شد. همچنین پیغام‌هایی را که هنگام تعریف قیودات مشخص کرده بودیم را نیز توسط آرایه** `ex.GetInvalidValues` **می‌توان دریافت کرد.**

نکته:

اگر کار ساخت `database schema` **را با کمک کانفیگ تنظیم شده توسط کتابخانه اعتبار سنجی آغاز کنیم، طول فیلدها دقیقا مطابق با حداکثر طول مشخص شده در قسمت تعاریف قیود هر یک از فیلدها تشکیل می‌گردد (حاصل از اعمال متد** `ApplyingDDLConstraints` **در متد** `BuildIntegratedFluentlyConfiguredEngine` **ذکر شده می‌باشد).**

```
    public static void CreateValidDb()
    {
        bool script = false;//آیا در خروجي کنسول هم نمایش داده شود
        bool export = true;//آیا روي بر دیتابیس هم اجرا شود
        bool dropTables = false;//آیا موجود جداول پاک شوند

        Configuration cfg = DbConfig.GetConfig().BuildConfiguration();
        VeConfig.BuildIntegratedFluentlyConfiguredEngine(ref cfg);
        //شود شروع کار باید سنجي اعتبار براي شده تنظیم کانفیگ همان با

        new SchemaExport(cfg).Execute(script, export, dropTables);
    }
```

[دریافت سورس کامل قسمت دهم]

قسمت یازدهم

یکسان سازي ي و ك دریافتي حین استفاده ازNHibernate

تصویر فوق، یکی از تصویرهایی است که شاید از طریق ایمیل‌هایی تحت عنوان "فقط در ایران!" به دست شما هم رسیده باشد. تصور کاربر نهایی (که این ایمیل را با تعجب ارسال کرده) این است که در اینجا به او گفته شده مثلا "مرتضی" را جستجو نکنید و امثال آن. چون برای او تفاوتی بین ی و ی وجود ندارد. همچنین بکار بردن "اقلامی" هم کمی غلط انداز است و بیشتر ذهن را به سمت کلمه سوق می‌دهد تا حرف.

در ادامه‌ی بحث آلرژی مزمن به وجود انواع "ی" و "ک" در بانک اطلاعاتی (+ و + و +)، اینبار قصد داریم این اطلاعات را به NHibernate بسط دهیم. شاید یک روش اعمال یک دست سازی "ی" و "ک" این باشد که در کل برنامه هر جایی که قرار است update یا insert ایی صورت گیرد، خواص رشته‌ای را یافته و تغییر دهیم. این روش "کار می‌کنه" ولی ایده آل نیست؛ چون حجم کار تکراری در برنامه زیاد خواهد شد و نگهداری آن هم مشکل می‌شود. همچنین امکان فراموش کردن اعمال آن هم وجود دارد.

در NHibernate یک سری EventListener وجود دارند که کارشان گوش فرا دادن به یک سری رخدادها مانند مثلا update یا insert است. این رخدادها می‌توانند پیش یا پس از هرگونه ثبت یا ویرایشی در برنامه صادر شوند. بنابراین بهترین جایی که جهت اعمال این نوع ممیزی (Auditing) بدون بالا بردن حجم برنامه یا اضافه کردن بیش از حد یک سری کد تکراری در حین کار با NHibernate می‌توان یافت، روال‌های مدیریت کننده‌ی همین EventListener ها هستند.

کلاس YeKeAuditorEventListener نهایی با پیاده سازی IPreInsertEventListener و IPreUpdateEventListenerبه شکل زیر خواهد بود:

```
using NHibernate.Event;

namespace NHYeKeAuditor
{
    public class YeKeAuditorEventListener : IPreInsertEventListener,
IPreUpdateEventListener
    {
        // Represents a pre-insert event, which occurs just prior to performing
the
        // insert of an entity into the database.
        public bool OnPreInsert(PreInsertEvent preInsertEvent)
        {
            var entity = preInsertEvent.Entity;
            CorrectYeKe.ApplyCorrectYeKe(entity);
            return false;
        }
```

```
        // Represents a pre-update event, which occurs just prior to performing
the
        // update of an entity in the database.
        public bool OnPreUpdate(PreUpdateEvent preUpdateEvent)
        {
            var entity = preUpdateEvent.Entity;
            CorrectYeKe.ApplyCorrectYeKe(entity);
            return false;
        }
    }
}
```

در کدهای فوق روال‌های `OnPreInsert` **و** `OnPreUpdate` **پیش از ثبت و ویرایش اطلاعات فراخوانی می‌شوند (همواره و بدون نیاز به نگرانی از فراموش شدن فراخوانی کدهای مربوطه). اینجا است که فرصت داریم تا تغییرات مورد نظر خود را جهت یکسان سازی "ی" و "ک" دریافتی اعمال کنیم (کد کلاس** `CorrectYeKe` **را در پیوست خواهید یافت).**

تا اینجا فقط تعریف `YeKeAuditorEventListener` **انجام شده است. اما** `NHibernate` **چگونه از وجود آن مطلع خواهد شد؟ برای تزریق کلاس** `YeKeAuditorEventListener` **به تنظیمات برنامه باید به شکل زیر عمل کرد:**

```
using System;
using System.Linq;
using FluentNHibernate.Cfg;
using NHibernate.Cfg;

namespace NHYeKeAuditor
{
    public static class MappingsConfiguration
    {
        public static FluentConfiguration InjectYeKeAuditorEventListener(this
FluentConfiguration fc)
        {
            return fc.ExposeConfiguration(configListeners());
        }

        private static Action<Configuration> configListeners()
        {
            return
                c =>
                {
                    var listener = new YeKeAuditorEventListener();
                    c.EventListeners.PreInsertEventListeners =
                        c.EventListeners.PreInsertEventListeners
                        .Concat(new[] { listener })
                        .ToArray();
                    c.EventListeners.PreUpdateEventListeners =
                        c.EventListeners.PreUpdateEventListeners
                        .Concat(new[] { listener })
                        .ToArray();
                };
        }
    }
}
```

به این معنا که `FluentConfiguration` **خود را همانند قبل ایجاد کنید. درست در زمان پایان کار تنها کافی است متد** `InjectYeKeAuditorEventListener` **فوق بر روی آن اعمال گردد و بس (یعنی پیش از فراخوانی** `BuildSessionFactory`**).**

کدهای `NHYeKeAuditor` **را از اینجا می‌توانید دریافت کنید.**

قسمت دوازدهم

NHibernate 3.0و ارائه‌ي جايگزيني جهتICriteria API

ICriteria API در NHibernate پیاده سازی الگوی Object Query است. مشکلی هم که این روش دارد استفاده از رشته‌ها جهت ایجاد کوئری‌های متفاوت است؛ به عبارتی Type safe نیست. ایرادی هم به آن وارد نیست چون پیاده سازی اولیه آن از جاوا صورت گرفته و مباحث Lambda Expressions و Extension Methods هنوز در آن زبان به صورت رسمی ارائه نشده است (در JDK 7 تحت عنوان Closures قرار است اضافه شود). NHibernate 3.0 از ویژگی‌های جدید زبان‌های دات نتی جهت ارائه‌ی محصور کننده‌ای Type safe حول ICriteria API استاندارد به نام QueryOver API سود جسته است. این پیاده سازی بسیار شبیه به عبارات LINQ است اما نباید با آن اشتباه گرفته شود زیرا NHibernate LINQ to یک ویژگی دیگر جدید، یکپارچه و استاندارد NHibernate 3.0 به شمار می‌رود.

برای نمونه در یک ICriteria query متداول، فراخوانی‌های ذیل متداول است:

```
.Add(Expression.Eq("Name", "Smith"))
```

اکنون شما در NHibernate 3.0 می‌توانید دستورات فوق را به صورت ذیل وارد نمائید:

```
.Where<Person>(p => p.Name == "Smith")
```

مزیت‌های این روش (strongly-typed fluent API) به شرح زیر است:

- خبری از رشته‌ها جهت استفاده از یک خاصیت وجود ندارد. برای مثال در اینجا خاصیت Name کلاس Person تحت کنترل کامپایلر قرار می‌گیرد و اگر در کلاس Person تغییراتی حاصل شود، برای مثال Name به LName تغییر کند، برنامه دیگر کامپایل نخواهد شد. اما در حالت ICriteria API یا باید به نتایج حاصل از Unit testing مراجعه کرد یا باید به نتایج بازخورد کاربران برنامه مانند: "باز برنامه رو تغییر دادی، یکجای دیگر از کار افتاد!" دقت نمود!
- اگر در حین ویرایش کلاس Person از ابزارهای Refactoring استفاده شود، تغییرات حاصل به صورت خودکار به تمام برنامه نیز اعمال خواهد شد. بدیهی است این اعمال تغییرات تنها در صورتی میسر است که خاصیت مورد نظر به صورت رشته معرفی نگردیده و ارجاعات به اشیاء تعریف شده به سادگی قابل parse باشند.
- در این حالت امکان بررسی نوع خواص تغییر کرده نیز توسط کامپایلر به سادگی میسر است و اگر ارجاعات تعریف شده به نحو صحیحی از این نوع جدید استفاده نکنند باز هم برنامه تا رفع این مشکلات کامپایل نخواهد شد که این هم مزیت مهمی در نگهداری ساده‌تر یک برنامه است.
- با بکارگیری Extension methods و پیاده سازی Fluent API جدید، مدت زمان یادگیری این روش نیز به شدت کاهش یافته، زیرا Intellisense موجود در VS.NET بهترین راهنمای استفاده از امکانات فراهم شده است. برای مثال جهت استفاده از ویژگی جدید QueryOver به سادگی می‌توان پس از ساختن یک session جدید به صورت زیر عمل نمود:

```
IList<Cat> cats = session.QueryOver<Cat>().Where(c => c.Name == "Max").List();
```

در اینجا اگر متدهای نمایش داده شده توسط Intellisense را دنبال کنیم دیگر حتی نیازی به مراجعه به مستندات QueryOver در مورد اینکه چه متدها و امکاناتی را فراهم کرده است نیز نخواهد بود.

جهت مشاهده‌ی معرفی کامل آن می‌توان به مستندات NHibernate 3.0 مراجعه کرد.

قسمت سیزدهم

NHibernate 3.0و عدم وابستگي مستقيم بهLog4Net

اولین موردی که پس از دریافت NHibernate 3.0 ممکن است به چشم بخورد، نبود اسمبلی Log4Net است. مطابق درخواست‌های کاربران، ارجاع مستقیم به این کتابخانه حذف شده و با یک اینترفیس عمومی به نام IInternalLogger جایگزین گشته است (قرار گرفته در فضای نام NHibernate.Logging). به این صورت می‌توان از انواع و اقسام کتابخانه‌های ثبت وقایع نوشته شده برای دات نت استفاده کرد؛ مانند: log4net، Nlog، EntLib Logging و غیره.
البته لازم به ذکر است که همان روش قبلی استفاده از Log4Net هنوز هم پشتیبانی می‌شود (بدون نیاز به تغییر خاصی در کدهای خود)، زیرا پیاده سازی اینترفیس جدید IInternalLogger برای استفاده از آن به صورت پیش فرض توسط NHibernate انجام شده است.

یک مثال:

کتابخانه‌ی سورس باز Common.Logging واقع شده در سورس فورج، در واقع یک logging abstraction framework است. به این معنا که تا به حال کتابخانه‌های ثبت وقایع مختلف و متعددی برای دات نت فریم ورک نوشته شده است و هر کدام راه و روش و پیاده سازی خاص خود را دارند. کتابخانه‌ی Common.Logging لایه‌ای است عمومی بر روی تمام این کتابخانه‌ها مانند Log4Net، Logging Enterprise Library، Nlog و غیره که برنامه‌ی شما را از وابستگی مستقیم به هر کدام از موارد ذکر شده رها می‌سازد.
اکنون با توجه به وجود اینترفیس IInternalLogger در NHibernate 3.0 ، تنها کافی است این اینترفیس جهت استفاده از کتابخانه‌ی Common.Logging پیاده سازی شود (abstraction اندر abstraction!). البته نیازی نیست اینکار انجام شود، زیرا پیشتر توسط پروژه‌ی NHibernate.Logging در سایت کدپلکس اینکار صورت گرفته است.
بنابراین تنها کاری که باید انجام داد این است که :
الف) ارجاعاتی را به اسمبلی‌های Common.Logging.dll (واقع در سورس فورج) و NHibernate.Logging.CommonLogging.dll (واقع در کدپلکس) به پروژه‌ی خود اضافه کنید.
ب) ارجاعی را به اسمبلی کتابخانه‌ی ثبت وقایع مورد نظر خود نیز باید اضافه نمائید (مثلا NLog).
ج) سپس باید چند سطر زیر را به فایل app.config خود اضافه کنید:

```
<appSettings>
  <add key="nhibernate-logger"
           value="NHibernate.Logging.CommonLogging.CommonLoggingLoggerFactory,
Hibernate.Logging.CommonLogging"/>
</appSettings>
```

NHibernate.Logging.CommonLogging.dll وقایع داخلی NHibernate را با پیاده سازی IInternalLogger به Common.Logging.dll منتقل می‌کند. سپس Common.Logging.dll این وقایع را به زبان کتابخانه‌ی ثبت وقایع مورد نظر ترجمه خواهد کرد.

اطلاعات بیشتر: (+)

قسمت چهاردهم

NHibernate 3.0و خواص تنبل!(lazy properties)

احتمالا مطلب "دات نت 4 و کلاس Lazy" را پیشتر مطالعه کرده‌اید. هر چند NHibernate 3.0 بر اساس دات نت فریم ورک 3 و نیم تهیه شده، اما شبیه به این مفهوم را در مورد بارگذاری به تاخیر افتاده‌ی مقادیر خواص یک کلاس که به ندرت مورد استفاده قرار می‌گیرند، پیاده سازی کرده است. Lazy را در اینجا تنبل، به تعویق افتاده، با تاخیر و شبیه به آن می‌توان ترجمه کرد؛ خواص معوقه!
برای مثال فرض کنید یکی از خواص کلاس مورد استفاده، متن، تصویر یا فایلی حجیم است. در مکانی هم که قرار است وهله‌ای از این کلاس مورد استفاده قرار گیرد نیازی به این اطلاعات حجیم نیست؛ با سایر خواص آن سر و کار داریم و نیازی به اشغال حافظه و منابع سیستم در این مورد خاص نیست.

سؤال: چگونه آن‌را تعریف کنیم؟
اگر از NHibernate سنتی استفاده می‌کنید (یا به عبارتی فایل‌های hbm.xml را دستی تهیه می‌کنید)، ویژگی Lazy را به صورت زیر می‌توان مشخص کرد:

```
<property name="Text" lazy="true"/>
```

اگر از Fluent NHibernate استفاده می‌کنید (و فایل‌های hbm.xml به صورت خودکار از کلاس‌های شما تهیه خواهند شد)، روش کار به صورت زیر است (فراخوانی متد LazyLoad روی خاصیت مورد نظر):

```
public class Post
{
    public virtual int Id { set; get; }
    public virtual string PostText { set; get; }
}
public class PostMappings : ClassMap<Post>
{
    public PostMappings()
    {
        Id(p => p.Id, "PostId").GeneratedBy.Identity();
        Map(p => p.PostText).LazyLoad();
        //…
        Table("Posts");
    }
}
```

در این حالت در پشت صحنه در مورد خاصیت PostText چنین نگاشتی تعریف خواهد شد:

```
  <property name="PostText" type="System.String, mscorlib, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089" lazy="true" />
```

سؤال: چه زمانی نباید از این روش استفاده کرد؟
فرض کنید در شرایطی دیگر نیاز است تا اطلاعات تمام رکوردهای جدول مذکور به همراه مقدار PostText نمایش داده شود. در این حالت بسته به تعداد رکوردها، ممکن است هزاران هزار کوئری به دیتابیس ایجاد شود که مطلوب نیست (به ازای هربار دسترسی به خاصیت PostText یک کوئری تولید می‌شود).

البته امکان لغو موقت این روش تنها در حین استفاده از HQL (یکی دیگر از روش‌های دسترسی به داده‌ها در NHibernate) میسر است.

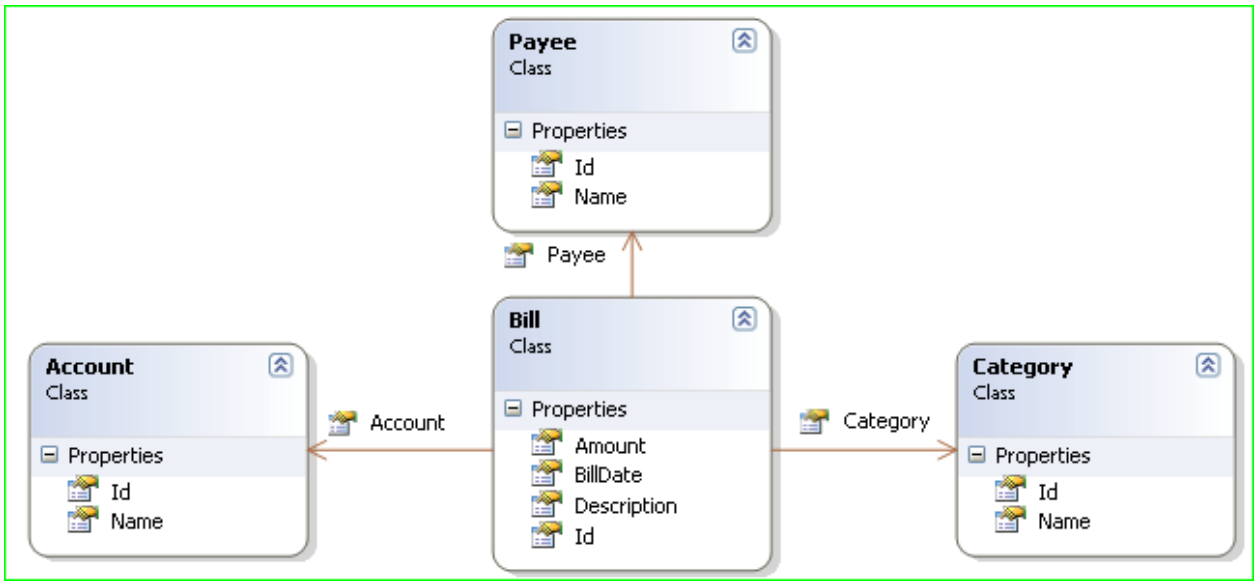
اطلاعات بیشتر: (+)

مقدار دهی کلیدهای خارجی در NHibernate وEntity framework

ORM های NHibernate و Entity framework روش‌های متفاوتی را برای به روز رسانی کلید خارجی با حداقل رفت و برگشت به دیتابیس ارائه می‌دهند که در ادامه معرفی خواهند شد.

صورت مساله:
فرض کنید می‌خواهیم برنامه‌ای را بنویسیم که ریز پرداخت‌های روزانه‌ی ما را ثبت کند. برای اینکار حداقل به یک جدول گروه‌های اقلام خریداری شده، یک جدول حساب‌های تامین کننده‌ی مخارج، یک جدول فروشنده‌ها و نهایتا یک جدول صورتحساب‌های پرداختی بر اساس جداول ذکر شده نیاز خواهد بود.

الف) بررسي مدل برنامه

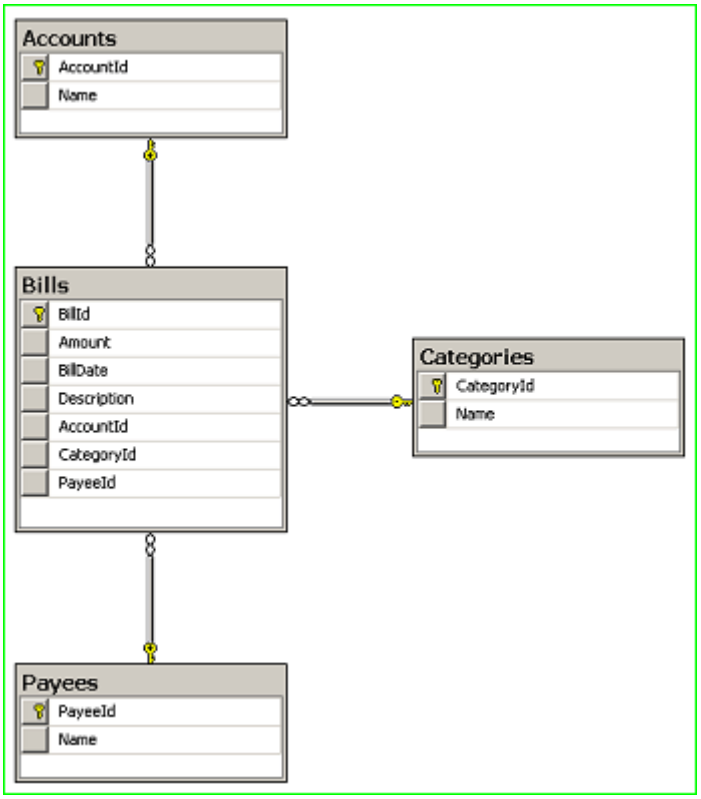


در اینجا جهت تعریف ویژگی‌ها یا Attributes تعریف شده در این کلاس‌ها از validator NHibernate استفاده شده (+). مزیت اینکار هم علاوه بر اعتبارسنجی سمت کلاینت (پیش از تبادل اطلاعات با بانک اطلاعاتی)، تولید جداولی با همین مشخصات است. برای مثال Fluent NHibernate بر اساس ویژگی Length تعریف شده با طول حداکثر 120 ، یک فیلد nvarchar با همین طول را ایجاد می‌کند.

```
public class Account
{
    public virtual int Id { get; set; }

    [NotNullNotEmpty]
    [Length(Min = 3, Max = 120, Message = "طول نام باید بین 3 و 120 کاراکتر باشد")]
    public virtual string Name { get; set; }
}

public class Category
{
    public virtual int Id { get; set; }

    [NotNullNotEmpty]
    [Length(Min = 3, Max = 130, Message = "طول نام باید بین 3 و 130 کاراکتر باشد")]
    public virtual string Name { get; set; }
}

public class Payee
```

```
{
    public virtual int Id { get; set; }

    [NotNullNotEmpty]
    [Length(Min = 3, Max = 120, Message = "طول نام باید بین 3 و 120 کاراکتر
باشد")]
    public virtual string Name { get; set; }
}

public class Bill
{
    public virtual int Id { get; set; }

    [NotNull]
    public virtual Account Account { get; set; }

    [NotNull]
    public virtual Category Category { get; set; }

    [NotNull]
    public virtual Payee Payee { get; set; }

    [NotNull]
    public virtual decimal Amount { set; get; }

    [NotNull]
    public virtual DateTime BillDate { set; get; }

    [NotNullNotEmpty]
    [Length(Min = 1, Max = 500, Message = "طول توضیحات باید بین 1 و 500 کاراکتر
باشد")]
    public virtual string Description { get; set; }
}
```

ب) ساختار جداول متناظر (تولید شده به صورت خودکار توسط Fluent NHibernate در اینجا)



در مورد نحوه‌ی استفاده از ویژگی AutoMapping و همچنین تولید خودکار ساختار بانک اطلاعاتی از روی جداول در NHibernate قبلا توضیح داده شده است. البته بدیهی است که ترکیب مقاله‌ی Validation و آشنایی با AutoMapping در اینجا جهت اعمال ویژگی‌ها باید بکار گرفته شود که در همان مقاله‌ی Validation مفصل توضیح داده شده است.

نکته‌ی مهم database schema تولیدی، کلیدهای خارجی (foreign key) تعریف شده بر روی جدول Bills است (همان AccountId، CategoryId و PayeeId تعریف شده) که به primary key جداول متناظر اشاره می‌کند.

```
 create table Accounts (
    AccountId INT IDENTITY NOT NULL,
   Name NVARCHAR(120) not null,
   primary key (AccountId)
)

create table Bills (
    BillId INT IDENTITY NOT NULL,
   Amount DECIMAL(19,5) not null,
   BillDate DATETIME not null,
   Description NVARCHAR(500) not null,
   AccountId INT not null,
   CategoryId INT not null,
   PayeeId INT not null,
   primary key (BillId)
)

create table Categories (
    CategoryId INT IDENTITY NOT NULL,
   Name NVARCHAR(130) not null,
   primary key (CategoryId)
)

create table Payees (
    PayeeId INT IDENTITY NOT NULL,
   Name NVARCHAR(120) not null,
   primary key (PayeeId)
)

alter table Bills
    add constraint fk_Account_Bill
    foreign key (AccountId)
    references Accounts

alter table Bills
    add constraint fk_Category_Bill
    foreign key (CategoryId)
    references Categories

alter table Bills
    add constraint fk_Payee_Bill
    foreign key (PayeeId)
    references Payees
```

ج) صفحه‌ي ثبت صورتحساب‌ها

صفحات ثبت گروه‌های اقلام، حساب‌ها و فروشنده‌ها، نکته‌ی خاصی ندارند. چون این جداول وابستگی خاصی به جایی نداشته و به سادگی اطلاعات آن‌ها را می‌توان ثبت یا به روز کرد.

صفحه‌ی مشکل در این مثال، همان صفحه‌ی ثبت صورتحساب‌ها است که از سه کلید خارجی به سه جدول دیگر تشکیل شده است.

عموما برای طراحی این نوع صفحات، کلیدهای خارجی را با drop down list نمایش می‌دهند و اگر در جهت سهولت کار کاربر قدم

برداشته شود، باید از یک Auto complete drop down list استفاده کرد تا کاربر برنامه جهت یافتن آیتم‌های از پیش تعریف شده کمتر سختی بکشد.

اگر از Silverlight یا WPF استفاده شود، امکان بایند یک لیست کامل از اشیاء با تمام خواص مرتبط به آن‌ها وجود دارد (هر رکورد نمایش داده شده در دراپ داون لیست، دقیقا معادل است با یک شیء متناظر با کلاس‌های تعریف شده است). اگر از ASP.NET استفاده شود (یعنی یک محیط بدون حالت که پس از نمایش یک صفحه دیگر خبری از لیست اشیاء بایند شده وجود نخواهد داشت و همگی توسط وب سرور جهت صرفه جویی در منابع تخریب شده‌اند)، بهتر است datatextfield را با فیلد نام و datavaluefield را با فیلد Id مقدار دهی کرد تا کاربر نهایی، نام را جهت ثبت اطلاعات مشاهده کند و برنامه از Id موجود در لیست جهت ثبت کلیدهای خارجی استفاده نماید.
و نکته‌ی اصلی هم همینجا است که چگونه؟! چون ما زمانیکه با یک ORM سر و کار داریم، برای ثبت یک رکورد در جدول Bills باید یک وهله از کلاس Bill را ایجاد کرده و خواص آن‌را مقدار دهی کنیم. اگر به تعریف کلاس Bill مراجعه کنید، سه خاصیت آن از نوع سه کلاس مجزا تعریف شده است. به به عبارتی با داشتن فقط یک id از رکوردهای این کلاس‌ها باید بتوان سه وهله‌ی متناظر آن‌ها را از بانک اطلاعاتی خواند و سپس به این خواص انتساب داد:

```
var newBill = new Bill
                  {
                      Account = accountRepository.GetByKey(1),
                      Amount = 1,
                      BillDate = DateTime.Now,
                      Category = categoryRepository.GetByKey(1),
                      Description = "testestest...",
                      Payee = payeeRepository.GetByKey(1)
                  };
```

یعنی برای ثبت یک رکورد در جدول Bills فوق، چهار بار رفت و برگشت به دیتابیس خواهیم داشت:
- یکبار برای دریافت رکورد متناظر با گروه‌ها بر اساس کلید اصلی آن (که از دراپ داون لیست مربوطه دریافت می‌شود)
- یکبار برای دریافت رکورد متناظر با فروشنده‌ها بر اساس کلید اصلی آن (که از دراپ داون لیست مربوطه دریافت می‌شود)
- یکبار برای دریافت رکورد متناظر با حساب‌ها بر اساس کلید اصلی آن (که از دراپ داون لیست مربوطه دریافت می‌شود)
- یکبار هم ثبت نهایی اطلاعات در بانک اطلاعاتی

متد GetByKey فوق همان متد session.Get استاندارد NHibernate است (چون به primary key ها از طریق drop down list دسترسی داریم، به سادگی می‌توان بر اساس متد Get استاندارد ذکر شده عمل کرد).

SQL نهایی تولیدی هم به صورت واضحی این مشکل را نمایش می‌دهد (4 بار رفت و برگشت؛ سه بار select یکبار هم insert نهایی):

```
SELECT account0_.AccountId as AccountId0_0_, account0_.Name as Name0_0_
FROM Accounts account0_ WHERE account0_.AccountId=@p0;@p0 = 1 [Type: Int32 (0)]

SELECT category0_.CategoryId as CategoryId2_0_, category0_.Name as Name2_0_
FROM Categories category0_ WHERE category0_.CategoryId=@p0;@p0 = 1 [Type: Int32
(0)]

SELECT payee0_.PayeeId as PayeeId3_0_, payee0_.Name as Name3_0_
FROM Payees payee0_ WHERE payee0_.PayeeId=@p0;@p0 = 1 [Type: Int32 (0)]

INSERT INTO Bills (Amount, BillDate, Description, AccountId, CategoryId,
PayeeId)
VALUES (@p0, @p1, @p2, @p3, @p4, @p5);
select SCOPE_IDENTITY();
@p0 = 1 [Type: Decimal (0)],
@p1 = 2010/12/27 11:48:33 Þ.Ù [Type: DateTime (0)],
@p2 = 'testestest...' [Type: String (500)],
@p3 = 1 [Type: Int32 (0)],
@p4 = 1 [Type: Int32 (0)],
@p5 = 1 [Type: Int32 (0)]
```

کسانی که قبلا با رویه‌های ذخیره شده کار کرده باشند (stored procedures) احتمالا الان خواهند گفت؛ ما که گفتیم این روش کند است! سربار زیادی دارد! فقط کافی است یک SP بنویسید و کل عملیات را با یک رفت و برگشت انجام دهید.
اما در ORMs نیز برای انجام این مورد در طی یک حرکت یک ضرب راه حل‌هایی وجود دارد که در ادامه بحث خواهد شد:

د) پیاده سازي با NHibernate

برای حل این مشکل در NHibernate با داشتن primary key (برای مثال از طریق datavaluefield ذکر شده)، بجای session.Get از session.**Load** استفاده کنید.
session.Get یعنی همین الان برو به بانک اطلاعاتی مراجعه کن و رکورد متناظر با کلید اصلی ذکر شده را بازگشت بده و یک شیء از آن را ایجاد کن (حالت‌های دیگر دسترسی به اطلاعات مانند استفاده از LINQ یا Criteria API یا هر روش مشابه دیگری نیز در اینجا به همین معنا خواهد بود).
session.Load یعنی فعلا دست نگه دار! مگر در جدول نهایی نگاشت شده، اصلا چیزی به نام شیء مثلا گروه وجود دارد؟ مگر این مورد واقعا یک فیلد عددی در جدول Bills بیشتر نیست؟ ما هم که الان این عدد را داریم (به کمک عناصر دراپ داون لیست)، پس لطفا در پشت صحنه یک پروکسی برای ایجاد شیء مورد نظر ایجاد کن (uninitialized proxy to the entity) و سپس عملیات مرتبط را در حین تشکیل SQL نهایی بر اساس این عدد موجود انجام بده. یعنی نیازی به رفت و برگشت به بانک اطلاعاتی نیست. در این حالت اگر SQL نهایی را بررسی کنیم فقط یک سطر زیر خواهد بود (سه select ذکر شده حذف خواهند شد):

```
INSERT INTO Bills (Amount, BillDate, Description, AccountId, CategoryId, PayeeId)
VALUES (@p0, @p1, @p2, @p3, @p4, @p5);
select SCOPE_IDENTITY();
@p0 = 1 [Type: Decimal (0)],
@p1 = 2010/12/27 11:58:22 Þ.Ù [Type: DateTime (0)],
@p2 = 'testestest...' [Type: String (500)],
@p3 = 1 [Type: Int32 (0)],
@p4 = 1 [Type: Int32 (0)],
@p5 = 1 [Type: Int32 (0)]
```

ه) پیاده سازي با framework Entity

Entity framework زمانیکه بانک اطلاعاتی فوق را (به روش database first) به کلاس‌های متناظر تبدیل/نگاشت می‌کند، حاصل نهایی مثلا در مورد کلاس Bill به صورت خلاصه به شکل زیر خواهد بود:

```
public partial class Bill : EntityObject
{
    public global::System.Int32 BillId { set; get; }
    public global::System.Decimal Amount { set; get; }
    public global::System.DateTime BillDate { set; get; }
    public global::System.String Description { set; get; }
    public global::System.Int32 AccountId { set; get; }
    public global::System.Int32 CategoryId { set; get; }
    public global::System.Int32 PayeeId { set; get; }
    public Account Account { set; get; }
    public Category Category { set; get; }
}
```

به عبارتی فیلدهای کلیدهای خارجی، در تعریف نهایی این کلاس هم مشاهده می‌شوند. در اینجا فقط کافی است سه کلید خارجی، از نوع `int` **مقدار دهی شوند (و نیازی به مقدار دهی سه شیء متناظر نیست). در این حالت نیز برای ثبت اطلاعات، فقط یکبار رفت و برگشت به بانک اطلاعاتی خواهیم داشت.**

قسمت پانزدهم

ذخیره سازی SQL تولیدی در NH3

همانطور که در مطلب "NHibernate 3.0 و عدم وابستگی مستقیم به Log4Net" عنوان شد، از اینترفیس جدید IInternalLogger آن می‌توان جهت ثبت وقایع داخلی NHibernate استفاده کرد. اگر در این بین صرفا بخواهیم SQL های تولیدی را لاگ کنیم، خلاصه‌ی آن به صورت زیر خواهد بود:

```
public class LoggerFactory : ILoggerFactory
{
    public IInternalLogger LoggerFor(System.Type type)
    {
       if (type == typeof(NHibernate.Tool.hbm2ddl.SchemaExport))
       //log it
    }

    public IInternalLogger LoggerFor(string keyName)
    {
        if (keyName == "NHibernate.SQL")
        //log it
    }
}
```

یا کلید NHibernate.SQL باید پردازش شود (جهت ثبت SQL های کوئری‌ها) یا نوع NHibernate.Tool.hbm2ddl.SchemaExport جهت ثبت SQL ساخت ساختار جداول بانک اطلاعاتی باید بررسی گردد.

سورس کامل این کتابخانه‌ی کوچک را از اینجا می‌توانید دریافت کنید. جهت استفاده از آن تنها کافی است چند سطر زیر به فایل app.config یا web.config برنامه‌ی شما اضافه شوند:

```
<appSettings>
  <add key="nhibernate-logger" value="NH3SQLLogger.LoggerFactory, NH3SQLLogger"
/>
</appSettings>
```

کلید nhibernate-logger، به صورت مستقیم توسط NHibernate بررسی می‌شود و صرف نظر از اینکه از کدامیک از مشتقات NHibernate استفاده می‌کنید، با تمام آن‌ها کار خواهد کرد.

لازم به ذکر است که اگر برنامه‌ی شما از نوع ASP.NET است، این کتابخانه اطلاعات را در پوشه‌ی استاندارد App_Data ثبت خواهد کرد؛ در غیراینصورت فایل‌ها در کنار فایل اجرایی برنامه تشکیل خواهند شد.

SQLتولیدی در NHibernate از کدام متد صادر شده است؟

اگر مطلب "ذخیره سازی SQL تولیدی در NH3" را دنبال کرده باشید که یک مثال عملی از "NHibernate 3.0 و عدم وابستگی مستقیم به Log4Net" بود، خروجی حاصل از آن به صورت زیر است:

```
---+ 12/29/2010 05:35:59.75 +---
SQL ...

---+ 12/29/2010 05:35:59.75 +---
SQL ...
```

و پس از مدتی این فایل هیچ حسی را منتقل نمی‌کند! یک سری SQL که لاگ شده‌اند. مشخص نیست کدام متد در کدام کلاس و کدام فضای نام، سبب صدور این عبارت SQL ثبت شده، گردیده است.

خوشبختانه در دات نت فریم ورک می‌توان با بررسی Stack trace، رد کاملی را از فراخوان‌های متدها یافت:

```
StackTrace stackTrace = new StackTrace();
```

```
StackFrame stackFrame = stackTrace.GetFrame(1);
MethodBase methodBase = stackFrame.GetMethod();
Type callingType=methodBase.DeclaringType;
```

با بررسی StackFrame ها امکان یافتن نام متدها، فضاهای نام و غیره میسر است. مثلا یکی از کاربردهای مهم این روش، ثبت فراخوان‌های متدی است که استثنایی را ثبت کرده است.
بر این اساس سورس مثال قبل را جهت درج اطلاعات فراخوان‌های متدها تکمیل کرده‌ام، که از این آدرس قابل دریافت است.

اکنون اگر از این ماژول جدید استفاده کنیم، خروجی نمونه‌ی آن به صورت زیر خواهد بود:

```
---+ 01/02/2011 02:19:24.98 +---
-- Void ASP.feedback_aspx.ProcessRequest(System.Web.HttpContext)
[File=App_Web_4nvdip40.5.cs, Line=0]
--- Void Prog.Web.UserCtrls.FeedbacksList.Page_Load(System.Object,
System.EventArgs) [File=FeedbacksList.ascx.cs, Line=23]
---- Void Prog.Web.UserCtrls.FeedbacksList.BindTo()
[File=FeedbacksList.ascx.cs, Line=43]
----- System.Collections.Generic.IList`1[Feedback]
Prog.GetAllUserFeedbacks(Int32) [File=FeedbackWebRepository.cs, Line=66]

SELECT ...
@p0 = 3 [Type: Int32 (0)]
```

به این معنا که عبارت SQL ثبت شده، حاصل از پردازش صفحه‌ی feedback.aspx، سپس متد Page_Load آن که از یوزر کنترل FeedbacksList.ascx استفاده می‌کند، می‌باشد. در اینجا فراخوانی متد BindTo سبب فراخوانی متد GetAllUserFeedbacks در فایل FeedbackWebRepository.cs واقع در سطر 66 آن گردیده است.
اینطوری حداقل می‌توان دریافت که SQL تولیدی دقیقا به کجا بر می‌گردد و چه متدی سبب صدور آن شده است.

ملاحظات:
این ماژول تنها در صورت وجود فایل pdb معتبر کنار اسمبلی‌های شما این خروجی مفصل را تولید خواهد کرد. در غیر اینصورت از آن صرفنظر می‌کند. (برای مثال نام فایل سورس فراخوان، شماره‌ی سطر فراخوان، حتی محل قرارگیری آن فایل بر روی کامپیوتر شما در فایل‌های pdb ثبت می‌گردند؛ به همین جهت توصیه اکید حین ارائه‌ی نهایی برنامه، حذف این نوع فایل‌ها است)

آیا دیتابیس مورد استفاده در NHibernate با نگاشت‌های تعریف شده همخوانی دارد؟
زمانیکه خاصیتی به یکی از کلاس‌های نگاشت‌های تعریف شده اضافه می‌شود یا حذف می‌گردد، دقیقا باید این به روز رسانی در سمت بانک اطلاعاتی هم انجام شود. امکان تهیه و همچنین اعمال اسکریپت نهایی تولید database schema مهیا است، اما ممکن است به هر علتی این کار فراموش شود. اکنون سؤال این است که آیا می‌توان سریع بررسی کرد که دیتابیس مورد استفاده با نگاشت‌های برنامه همخوانی و تطابق دارد؟
جهت پاسخ به این سؤال بهترین راه ایجاد یک کوئری Select بر اساس تمام خواص تعریف شده در یک کلاس است. اگر یکی از خواص یا حتی خود جدول وجود نداشته باشد، انجام این کوئری خودبخود با شکست مواجه شده و یک استثناء صادر خواهد شد. همین ایده را به سادگی می‌توان با NHibernate هم پیاده سازی کرد:

```
public class ConfirmDatabaseMatchesMappings
{
    public static void ValidateDatabaseSchemaAgainstMappings()
    {
        //در اینجا باید سشن فکتوری سراسری تعریف شده را دریافت و استفاده کرد
        using (var session = sessionManager.OpenSession())
        {
            var allClassMetadata =
session.SessionFactory.GetAllClassMetadata();
```

```
            foreach (var entry in allClassMetadata)
            {

session.CreateCriteria(entry.Value.GetMappedClass(EntityMode.Poco))
                        .SetMaxResults(0).List();
            }
        }
    }
}
```

برای مثال اگر فیلدی در کلاس‌های برنامه موجود باشد اما در بانک اطلاعاتی خیر، استثنای حاصل شبیه به عبارات ذیل خواهد بود:

```
NHibernate.Exceptions.GenericADOException was unhandled
Message=could not execute query
...
```

و اگر کمی سایر اطلاعات این استثناء را بررسی کنیم، به همان عبارات آشنای فلان فیلد یافت نشد یا فلان جدول وجود ندارد، می‌رسیم.

قسمت شانزدهم

مدیریت Join در NHibernate 3.0

مباحث eager fetching/loading (واکشی حریصانه) و lazy loading/fetching (واکشی در صورت نیاز، با تاخیر، تنبل) جزو نکات کلیدی کار با ORM های پیشرفته بوده و در صورت عدم اطلاع از آن‌ها و یا استفاده‌ی ناصحیح از هر کدام، باید منتظر از کار افتادن زود هنگام سیستم در زیر بار چند کاربر همزمان بود. به همین جهت تصور اینکه "با استفاده از ORMs دیگر از فراگیری SQL راحت شدیم!" یا اینکه "به من چه که پشت صحنه چه اتفاقی می‌افته!" بسی مهلک و نادرست است!
در ادامه به تفصیل به این موضوع پرداخته خواهد شد.

ابزار مورد نیاز

در این مطلب از برنامه‌ی NHProf استفاده خواهد شد.
اگر مطالب NHibernate این سایت را دنبال کرده باشید، در مورد لاگ کردن SQL تولیدی به اندازه‌ی کافی توضیح داده شده یا حتی یک ماژول جمع و جور هم برای مصارف دم دستی نوشته شده است. این موارد شاید این ایده را به همراه داشته باشند که چقدر خوب می‌شد یک برنامه‌ی جامع‌تر برای این نوع بررسی‌ها تهیه می‌شد. حداقل SQL نهایی فرمت می‌شد (یعنی برنامه باید مجهز به یک SQL Parser تمام عیار باشد که کار چند ماهی هست ...؛ با توجه به اینکه مثلا NHibernate از افزونه‌های SQL ویژه بانک‌های اطلاعاتی مختلف هم پشتیبانی می‌کند، مثلا T-SQL مایکروسافت با یک سری ریزه کاری‌های منحصر به MySQL متفاوت است)، یا پس از فرمت شدن، syntax highlighting به آن اضافه می‌شد، در ادامه مشخص می‌کرد کدام کوئری‌ها سنگین‌تر هستند، کدامیک نشانه‌ی عدم استفاده‌ی صحیح از ORM مورد استفاده است، چه مشکلی دارد و از این موارد.
خوشبختانه این ایده‌ها یا آرزوها با برنامه‌ی NHProf محقق شده است. این برنامه برای استفاده‌ی یک ماه اول آن رایگان است (آدرس ایمیل خود را وارد کنید تا یک فایل مجوز رایگان یک ماهه برای شما ارسال گردد) و پس از یک ماه، باید حداقل 300 دلار هزینه کنید.

واکشي حریصانه و غیرحریصانه چیست؟

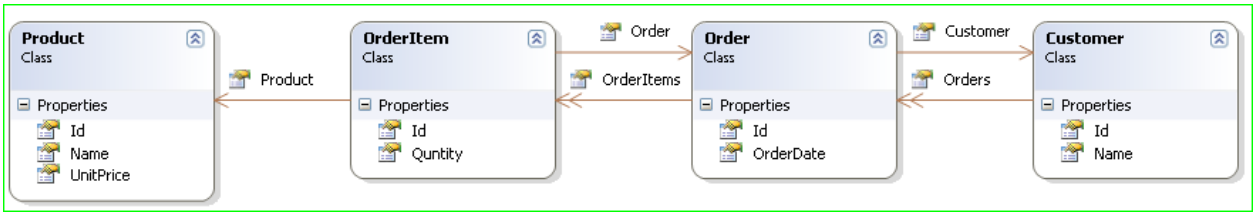
رفتار یک ORM جهت تعیین اینکه آیا نیاز است برای دریافت اطلاعات بین جداول Join صورت گیرد یا خیر، واکشی حریصانه و غیرحریصانه را مشخص می‌سازد.
در حالت واکشی حریصانه به ORM خواهیم گفت که لطفا جهت دریافت اطلاعات فیلدهای جداول مختلف، از همان ابتدای کار در پشت صحنه، Join های لازم را تدارک ببین. در حالت واکشی غیرحریصانه به ORM خواهیم گفت به هیچ عنوان حق نداری Join ایی را تشکیل دهی. هر زمانی که نیاز به اطلاعات فیلدی از جدولی دیگر بود باید به صورت مستقیم به آن مراجعه کرده و آن مقدار را دریافت کنی.
به صورت خلاصه برنامه نویس در حین کار با ORM های پیشرفته نیازی نیست Join بنویسد. تنها باید ORM را طوری تنظیم کند که آیا اینکار را حتما خودش در پشت صحنه انجام دهد (واکشی حریصانه)، یا اینکه خیر، به هیچ عنوان SQL های تولیدی در پشت صحنه نباید حاوی Join باشند (lazy loading).

چگونه واکشي حریصانه و غیرحریصانه را در NHibernate 3.0 تنظیم کنیم؟

در NHibernate اگر تنظیم خاصی را تدارک ندیده و خواص جداول خود را به صورت virtual معرفی کرده باشید، تنظیم پیش فرض دریافت اطلاعات همان lazy loading است. به مثالی در این زمینه توجه بفرمائید:

مدل برنامه:

مدل برنامه همان مثال کلاسیک مشتری و سفارشات او می‌باشد. هر مشتری چندین سفارش می‌تواند داشته باشد. هر سفارش به یک مشتری وابسته است. هر سفارش نیز از چندین قلم جنس تشکیل شده است. در این خرید، هر جنس نیز به یک سفارش وابسته است.




```
using System.Collections.Generic;
namespace CustomerOrdersSample.Domain
{
    public class Customer
    {
        public virtual int Id { get; set; }
        public virtual string Name { get; set; }
        public virtual IList<Order> Orders { get; set; }
    }
}
```

```
using System;
using System.Collections.Generic;
namespace CustomerOrdersSample.Domain
{
    public class Order
    {
        public virtual int Id { get; set; }
        public virtual DateTime OrderDate { set; get; }
        public virtual Customer Customer { get; set; }
        public virtual IList<OrderItem> OrderItems { set; get; }
    }
}
namespace CustomerOrdersSample.Domain
{
    public class OrderItem
    {
        public virtual int Id { get; set; }
        public virtual Product Product { get; set; }
        public virtual int Quntity { get; set; }
        public virtual Order Order { set; get; }
    }
}
```

```
namespace CustomerOrdersSample.Domain
{
    public class Product
    {
        public virtual int Id { set; get; }
        public virtual string Name { get; set; }
        public virtual decimal UnitPrice { get; set; }
    }
}
```

که جداول متناظر با آن به صورت زیر خواهند بود:

```
    create table Customers (
    CustomerId INT IDENTITY NOT NULL,
    Name NVARCHAR(255) null,
    primary key (CustomerId)
 )
```

```
create table Orders (
    OrderId INT IDENTITY NOT NULL,
    OrderDate DATETIME null,
    CustomerId INT null,
    primary key (OrderId)
)

create table OrderItems (
    OrderItemId INT IDENTITY NOT NULL,
    Quntity INT null,
    ProductId INT null,
    OrderId INT null,
    primary key (OrderItemId)
)

create table Products (
    ProductId INT IDENTITY NOT NULL,
    Name NVARCHAR(255) null,
    UnitPrice NUMERIC(19,5) null,
    primary key (ProductId)
)

alter table Orders
     add constraint fk_Customer_Order
     foreign key (CustomerId)
     references Customers

alter table OrderItems
     add constraint fk_Product_OrderItem
     foreign key (ProductId)
     references Products

alter table OrderItems
     add constraint fk_Order_OrderItem
     foreign key (OrderId)
     references Orders
```

همچنین یک سری اطلاعات آزمایشی زیر را هم در نظر بگیرید: (بانک اطلاعاتی انتخاب شده SQL CE است)

```
SET IDENTITY_INSERT [Customers] ON;
GO
INSERT INTO [Customers] ([CustomerId],[Name]) VALUES (1,N'Customer1');
GO
SET IDENTITY_INSERT [Customers] OFF;
GO
SET IDENTITY_INSERT [Products] ON;
GO
INSERT INTO [Products] ([ProductId],[Name],[UnitPrice]) VALUES
(1,N'Product1',1000.00000);
GO
INSERT INTO [Products] ([ProductId],[Name],[UnitPrice]) VALUES
(2,N'Product2',2000.00000);
GO
INSERT INTO [Products] ([ProductId],[Name],[UnitPrice]) VALUES
(3,N'Product3',3000.00000);
GO
SET IDENTITY_INSERT [Products] OFF;
GO
SET IDENTITY_INSERT [Orders] ON;
```

```
GO
INSERT INTO [Orders] ([OrderId],[OrderDate],[CustomerId]) VALUES (1,{ts '2011-01-07 11:25:20.000'},1);
GO
SET IDENTITY_INSERT [Orders] OFF;
GO
SET IDENTITY_INSERT [OrderItems] ON;
GO
INSERT INTO [OrderItems] ([OrderItemId],[Quntity],[ProductId],[OrderId]) VALUES (1,10,1,1);
GO
INSERT INTO [OrderItems] ([OrderItemId],[Quntity],[ProductId],[OrderId]) VALUES (2,5,2,1);
GO
INSERT INTO [OrderItems] ([OrderItemId],[Quntity],[ProductId],[OrderId]) VALUES (3,20,3,1);
GO
SET IDENTITY_INSERT [OrderItems] OFF;
GO
```

دریافت اطلاعات :

می‌خواهیم نام کلیه محصولات خریداری شده توسط مشتری‌ها را به همراه نام مشتری و زمان خرید مربوطه، نمایش دهیم (دریافت اطلاعات از 4 جدول بدون `join` نویسی):

```
var list = session.QueryOver<Customer>().List();

foreach (var customer in list)
{
    foreach (var order in customer.Orders)
    {
        foreach (var orderItem in order.OrderItems)
        {
              Console.WriteLine("{0}:{1}:{2}", customer.Name, order.OrderDate, orderItem.Product.Name);
         }
     }
}
```

خروجی به صورت زیر خواهد بود:

```
Customer1:2011/01/07 11:25:20 :Product1
Customer1:2011/01/07 11:25:20 :Product2
Customer1:2011/01/07 11:25:20 :Product3
```

همانطور که مشاهده می‌کنید در اینجا اطلاعات از 4 جدول مختلف دریافت می‌شوند اما ما Join ایی را ننوشته‌ایم. ORM هرجایی که به اطلاعات فیلدهای جداول دیگر نیاز داشته، به صورت مستقیم به آن جدول مراجعه کرده و یک کوئری، حاصل این عملیات خواهد بود (مطابق تصویر جمعا 6 کوئری در پشت صحنه برای نمایش سه سطر خروجی فوق اجرا شده است).

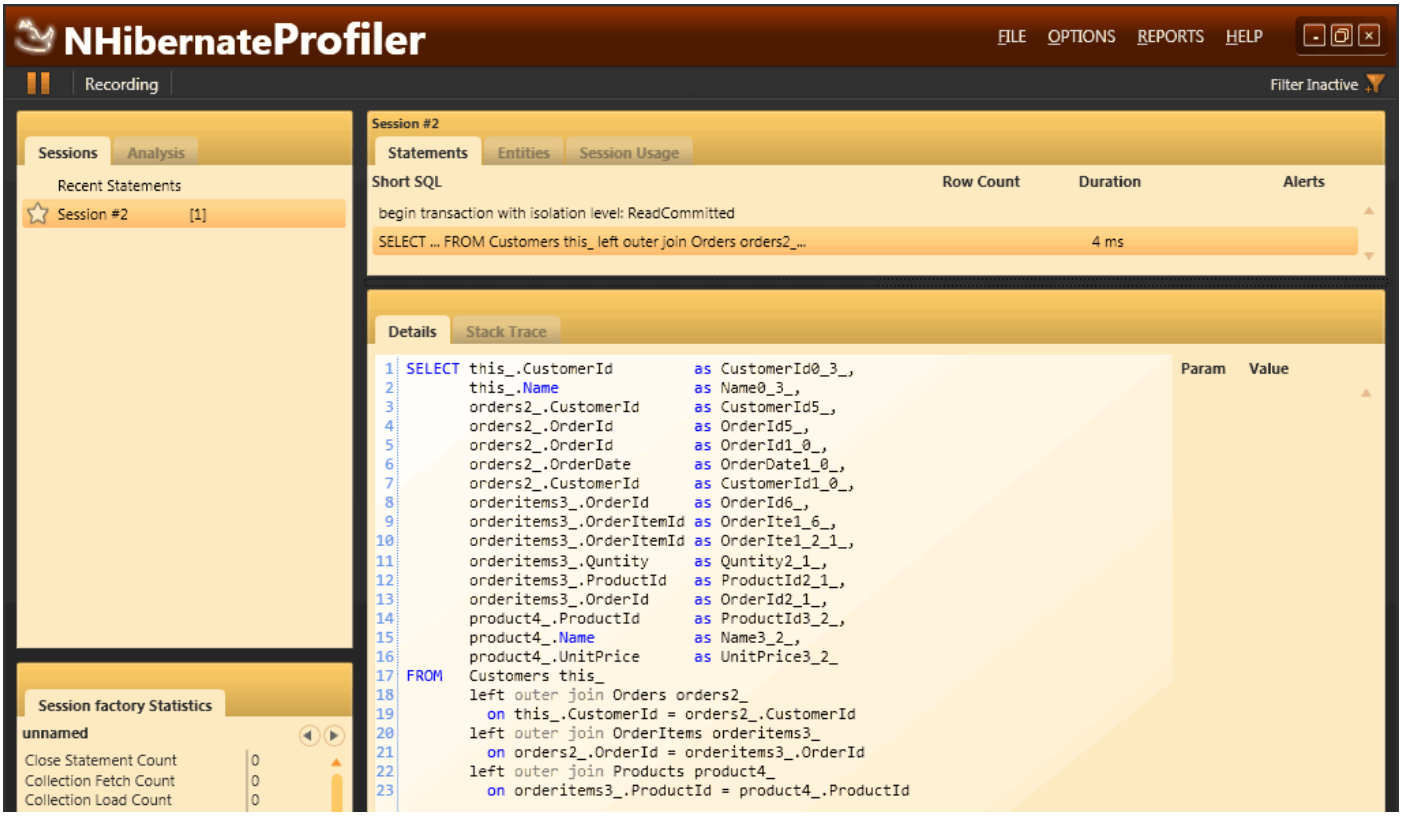
این حالت فقط و فقط با تعداد رکورد کم بهینه است (و به همین دلیل هم تدارک دیده شده است). بنابراین اگر برای مثال قصد نمایش اطلاعات حاصل از 4 جدول فوق را در یک گرید داشته باشیم، بسته به تعداد رکوردها و تعداد کاربران همزمان برنامه (خصوصا در برنامه‌های تحت وب)، بانک اطلاعاتی باید بتواند هزاران هزار کوئری رسیده حاصل از lazy loading را پردازش کند و این یعنی مصرف بیش از حد منابع IO) بالا، مصرف حافظه بالا) به همراه بالا رفتن CPU usage و از کار افتادن زود هنگام سیستم.

کسانی که پیش از این با SQL نویسی خو گرفته‌اند احتمالا الان منابع موجود را در مورد نحوه‌ی نوشتن Join در NHibernate زیر و رو خواهند کرد؛ زیرا پیش از این آموخته‌اند که برای دریافت اطلاعات از دو یا چند جدول مرتبط باید Join نوشت. اما همانطور که پیشتر نیز عنوان شد، اگر با جزئیات کار با NHibernate آشنا شویم، نیازی به Join نویسی نخواهیم داشت. اینکار را خود ORM در پشت صحنه باید و می‌تواند مدیریت کند. اما چگونه؟

در NHibernate 3.0 با معرفی QueryOver که جایگزینی از نوع strongly typed همان ICriteria API قدیمی است، یا با معرفی Query که همان LINQ to NHibernate می‌باشد، متدی به نام Fetch نیز تدارک دیده شده است که استراتژی‌های lazy loading eager و loading را به سادگی توسط آن می‌توان مشخص نمود.

مثال: دریافت اطلاعات با استفاده از QueryOver

```
var list = session
                .QueryOver<Customer>()
                .Fetch(c => c.Orders).Eager
                .Fetch(c => c.Orders.First().OrderItems).Eager
                .Fetch(c => c.Orders.First().OrderItems.First().Product).Eager
                .List();

foreach (var customer in list)
{
   foreach (var order in customer.Orders)
   {
         foreach (var orderItem in order.OrderItems)
         {
             Console.WriteLine("{0}:{1}:{2}", customer.Name, order.OrderDate,
orderItem.Product.Name);
          }
    }
}
```



اینبار فقط یک کوئری حاصل عملیات بوده و `join` ها به صورت خودکار با توجه به متدهای `Fetch` ذکر شده که حالت `eager loading` آن‌ها صریحا مشخص شده است، تشکیل شده‌اند (6 بار رفت و برگشت به بانک اطلاعاتی به یکبار تقلیل یافت).

نکته 1: نتایج تکراری

اگر حاصل `join` آخر را نمایش دهیم، نتایجی تکراری خواهیم داشت که مربوط است به مقدار دهی `customer` با سه وهله از شیء مربوطه تا بتواند واکشی حریصانه‌ی مجموعه اشیاء فرزند آن‌را نیز پوشش دهد. برای رفع این مشکل یک سطر `TransformUsing` باید اضافه شود:

```
...
.TransformUsing(NHibernate.Transform.Transformers.DistinctRootEntity)
.List();
```

دریافت اطلاعات با استفاده از NHibernate3.0 LINQ to

برای اینکه بتوان متدهای `Fetch` ذکر شده را به `NHibernate 3.0 LINQ to` اعمال نمود، ذکر فضای نام `NHibernate.Linq` ضروری است. پس از آن خواهیم داشت:

```
var list = session
                .Query<CUSTOMER>()
                .FetchMany(c => c.Orders)
                .ThenFetchMany(o => o.OrderItems)
                .ThenFetch(p => p.Product)
                .ToList();
```

اینبار از `FetchMany`، سپس `ThenFetchMany` (برای واکشی حریصانه مجموعه‌های فرزند) و در آخر از `ThenFetch` استفاده خواهد شد.

همانطور که ملاحظه می‌کنید حاصل این کوئری، با کوئری قبلی ذکر شده یکسان است. هر دو، اطلاعات مورد نیاز از دو جدول مختلف را نمایش می‌دهند. اما یکی در پشت صحنه شامل چندین و چند کوئری برای دریافت اطلاعات است، اما دیگری تنها از یک کوئری `Join` دار تشکیل شده است.

نکته 2: خطاهای ممکن

ممکن است حین تعریف متدهای Fetch **در زمان اجرا به خطاهای** Antlr.Runtime.MismatchedTreeNodeException **و یا** Specified method is not supported **و یا موارد مشابهی برخورد نمائید. تنها کاری که باید انجام داد جابجا کردن مکان بکارگیری** extension methods **است. برای مثال متد** Fetch **باید پس از** Where **در حالت استفاده از** LINQ **ذکر شود و نه قبل از آن.**

بالا بردن سرعت بارگذاری اولیهNHibernate

در زمان اولین بارگذاری NHibernate **، ساخت تمام نگاشت‌ها صورت گرفته و همچنین** session factory **ایجاد می‌گردد. به همین جهت به کمک الگوی** thread safe singleton **نسبت به کش کردن آن در طول عمر یک برنامه استفاده می‌گردد.**
در برنامه‌ای که در یک محیط کاری مورد استفاده قرار می‌گیرد این زمان اصلا مهم نیست، زیرا تنها یکبار باید انجام شود. اما به عنوان یک برنامه نویس شاید در طول روز صدها بار نیاز به باز و بسته کردن برنامه جهت آزمودن آن داشته باشیم و این مورد پس از مدتی تبدیل به عذاب می‌شود! خوشبختانه امکان serialize **نمودن تنظیمات تولیدی** session factory **به فایل و سپس خواندن از آن نیز وجود دارد که این امر در حین توسعه‌ی برنامه بسیار ارزشمند است.**
جهت مطالعه بیشتر می‌توان به مطالب زیر مراجعه کرد:

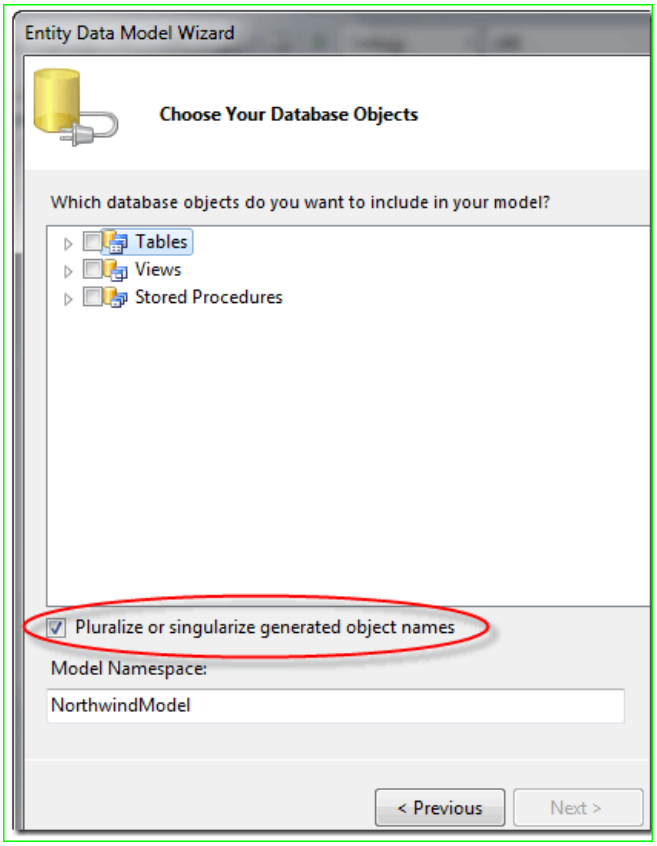
- Speed up nHibernate startup with object serialization
- An improvement on SessionFactory Initialization
- Optimizing application startup time with Fluent NHibernate and uNhAddIns
- NHibernate Hidden Gems - Speed up NHibernate startup time

و حاصل تمام این مقالات در پروژه‌ی Effectus**، فایل** Effectus\Infrastructure\BootStrapper.cs **آن گردآوری شده است.**

قسمت هفدهم

سرویس جمع و مفرد سازی اسامی

اگر به Entity data model wizard **در** VS.Net 2010 **دقت کرده باشید، گزینه‌ی** "Pluralize or singularize generated object names" **نیز به آن اضافه شده است:**



این مورد از این جهت حائز اهمیت است که عموما نام جداول در بانک اطلاعاتی، جمع است و نام کلاس متناظر ایجاد شده برای آن در کدهای برنامه بهتر است مفرد باشد. برای مثال نام جدول، Customers **است و نام کلاس آن بهتر است** Customer **تعریف گردد. به این صورت کار کردن با آن توسط یک** ORM **با معناتر خواهد بود؛ زیرا زمانیکه یک وهله از شیء** Customer **ایجاد می‌شود، فقط یک رکورد از بانک اطلاعاتی مد نظر است؛ در حالیکه یک جدول مجموعه‌ای است از رکوردها.**
زبان انگلیسی هم پر است از اسامی جمع و مفرد باقاعده و بی‌قاعده و کل عملیات با اضافه و حذف کردن یک s **و یا** es **پایان نمی‌یابد؛ برای مثال** phenomenon **و** phenomena **را در نظر بگیرید تا** Money **و** Moneys.

این امکان مهیا شده توسط Entity Framework 4.0 **یا همان** EF v2 **با برنامه نویسی هم قابل دسترسی است و در اسمبلی** System.Data.Entity.Design.dll **و فضای نام** System.Data.Entity.Design.PluralizationServices **قرار گرفته است.**
این اسمبلی جزیی از دات نت 4 **است و اگر آن‌را توسط گزینه‌ی** Add references **در** VS.NET **مشاهده نمی‌کنید، علت آن است که در تنظیمات پروژه جاری، گزینه‌ی** Target framework **بر روی** Client profile **قرار گرفته است که باید به دات نت** 4 **کامل تغییر یابد.**
استفاده از آن هم به صورت زیر است:

```
using System;
using System.Data.Entity.Design.PluralizationServices;
using System.Globalization;

namespace PluralizationServicesTest
{
    class Program
```

```
    {
        static void Main(string[] args)
        {
            var service =
PluralizationService.CreateService(CultureInfo.GetCultureInfo("en"));
            Console.WriteLine(service.Pluralize("mouse"));
            Console.WriteLine(service.IsPlural("phenomena"));
        }
    }
}
```

ملاحظات:

این روش فعلا به زبان انگلیسی محدود است و اگر `Culture` **را به مورد دیگری تنظیم کنید با خطای** `"We don't support locales other than English yet"` **متوقف خواهید شد.**

روش دیگر:

کتابخانه‌ی سورس باز `Castle ActiveRecord` **نیز دارای کلاسی است به نام** `Inflector` **که برای همین منظور طراحی شده است:**

Inflector.cs

کاربرد آن در Fluent NHibernate

در `Fluent NHibernate` **کار نگاشت کلاس‌ها به جداول به صورت خودکار صورت می‌گیرد و همچنین تولید ساختار بانک اطلاعاتی نیز به همین نحو می‌باشد. اما می‌توان تولید نام جداول را سفارشی نیز نمود. برای مثال از کلاس** `Book` **به صورت خودکار ساختار جدولی به نام** `Books` **را تولید کند:**

```
using FluentNHibernate.Conventions;
using FluentNHibernate.Conventions.Instances;
using NHibernate.Helper.Toolkit;

namespace NHibernate.Helper.MappingConventions
{
    public class TableNameConvention : IClassConvention
    {
        public void Apply(IClassInstance instance)
        {
            instance.Table(Inflector.Pluralize(instance.EntityType.Name));
        }
    }
}
```

و برای تزریق آن خواهیم داشت:

```
... = new AutoPersistenceModel()
                                .Where(...)
                                .Conventions.Setup(c
=>c.Add<TableNameConvention>())
                                .AddEntityAssembly(...)
...
```

قسمت هجدهم

NHibernateو سطح اول cache آن

این روزها هیچکدام از فناوری‌های دسترسی به داده بدون امکان یکپارچگی آن‌ها با سیستم‌ها و روش‌های متفاوت caching ، مطلوب شمرده نمی‌شوند. ایده اصلی caching هم به زبان ساده به این صورت است : فراهم آوردن روش‌هایی جهت میسر ساختن دسترسی سریعتر به داده‌هایی که به صورت متناوب در برنامه مورد استفاده قرار می‌گیرند، بجای مراجعه مستقیم به بانک اطلاعاتی و خواندن اطلاعات از دیسک سخت.

یکی از تفاوت‌های مهم NHibernate با اکثر ORM های موجود داشتن دو سطح متفاوت cache است : first level cache & second level cache.

برای نمونه Entity framework (در زمان نگارش این مطلب) تنها first level caching را پشتیبانی می‌کند و پروایدر توکار و یکپارچه‌ای را جهت second level caching ارائه نمی‌دهد.

در این قسمت قصد داریم First Level Cache را بررسی کنیم.

سطح اول caching در NHibernate چیست؟

سطح اول caching در تمام ORM هایی که آن‌را پشتیبانی می‌کنند مانند NHibernate ، در طول عمر یک تراکنش تعریف می‌گردد. در این حالت در طی یک تراکنش و طول عمر یک سشن، دریافت اطلاعات هر رکورد از بانک اطلاعاتی، تنها یکبار انجام خواهد شد؛ صرفنظر از اینکه کوئری دریافت اطلاعات آن چندبار فراخوانی می‌گردد. یکی از دلایل این روش هم آن است که هیچ دو شیء متفاوتی که هم اکنون در حافظه قرار دارند نباید بیانگر یک رکورد واحد از بانک اطلاعاتی باشند.

در NHibernate به صورت پیش فرض هر زمانیکه از شیء استاندارد session استفاده می‌کنید، سطح اول caching نیز فعال است. درست در زمانیکه سشن خاتمه می‌یابد، این سطح از caching نیز به صورت خودکار تخلیه خواهد گردید.

به first level caching اصطلاحا thought-out cache system یا Cache Through pattern و یا identity map هم گفته می‌شود.

مثال:

روش متداول و استاندارد کار با NHibernate عموما به صورت زیر است:

الف) دریافت شیء Session از Session Factory

ب) شروع یک تراکنش با فراخوانی متد BeginTransaction شیء Session

ج) برای مثال دریافت اطلاعات رکوردی با ID مساوی یک به کمک متد Get مرتبط با شیء Session : این اطلاعات مستقیما از بانک اطلاعاتی دریافت خواهد شد.

د) سپس مجددا سعی در دریافت رکوردی با ID مساوی یک. اینبار اطلاعات این شیء مستقیما از cache خوانده می‌شود و رفت و برگشتی به بانک اطلاعاتی نخواهیم داشت. به همین جهت به این روش identity map هم گفته می‌شود، زیرا NHibernate بر اساس ID منحصربفرد این اشیاء ، identity map خود را تشکیل می‌دهد.

ه) خاتمه‌ی سشن با فراخوانی متد Close آن

بلافاصله

الف) دریافت شیء Session از Session Factory

ب) شروع یک تراکنش با فراخوانی متد BeginTransaction شیء Session

ج) برای مثال دریافت اطلاعات رکوردی با ID مساوی یک به کمک متد Get مرتبط با شیء Session : این اطلاعات مستقیما از بانک اطلاعاتی دریافت خواهد شد (زیرا در یک سشن جدید قرار داریم و همچنین سشن قبلی بسته شده و کش آن تخلیه گشته است).

د) خاتمه‌ی سشن با فراخوانی متد Close آن

سؤال: آیا استفاده از یک سشن سراسری در برنامه صحیح است؟

پاسخ: خیر!

توضیحات: زمانیکه از یک سشن سراسری استفاده می‌کنید، کش NHibernate را در اختیار تمام کاربران همزمان سیستم قرار داده‌اید. در طی یک سشن، همانطور که عنوان شد، بر اساس IDهای اشیاء، یک identity map تشکیل می‌شود و در این حالت به ازای هر رکورد بانک اطلاعاتی فقط و فقط یک شیء در حافظه وجود خواهد داشت که این روش در محیط‌های چندکاربره مانند برنامه‌های وب به زودی تبدیل به نشت اطلاعات و یا تخریب اطلاعات می‌گردد. به همین جهت در این نوع برنامه‌ها روش session-per-request بهترین حالت کاری است.

سؤال: حین به روز رسانی اشیاء جدید، به خطا بر می‌خورم. مشکل در کجاست؟

فرض کنید شیء مفروض Customer را توسط متد session.Get از بانک اطلاعاتی دریافت و تعدادی از خواص آن‌را جهت ساخت شیء جدیدی از کلاس Customer استفاده کرده‌ایم. اکنون اگر بخواهیم این شیء جدید را در بانک اطلاعاتی ذخیره یا به روز رسانی کنیم، NHibernate این اجازه را نمی‌دهد! چرا؟

پاسخ:

خطای متداول این حالت عموما به صورت زیر است:

```
a different object with the same identifier value was already associated with t
he session
```

اگر شخصی با مکانیزم سطح اول caching در NHibernate آشنایی نداشته باشد، شاید ساعاتی را در انجمن‌های مرتبط، جهت یافتن روش حل خطای فوق سپری کند.

همانطور که عنوان شد، در طول یک سشن، نمی‌توان دو شیء با یک ID را به عنوان یک رکورد بانک اطلاعاتی مورد استفاده قرار داد. اولین فراخوانی Get ، سبب کش شدن آن شیء در identity map سطح اول caching می‌گردد.

راه حل:

الف) از چندین و چند شیء استفاده نکنید. هر رکورد باید تنها با یک وهله از شیءایی متناظر باشد.

ب) می‌توان پیش از update، کش سطح اول را به صورت دستی خالی کرد. برای این منظور از متد Clear شیء سشن استفاده کنید.

ج) بجای استفاده از متد saveOrUpdate شیء سشن، از متد Merge آن استفاده کنید. به این صورت شیء جدید ایجاد شده با شیء موجود در کش یکی خواهد شد.

د) می‌توان بجای تخلیه کل کش (حالت ب)، کش مرتبط با شیء Customer را به صورت دستی خالی کرد. برای این منظور از متد Evict شیء سشن استفاده نمائید.

و لازم به ذکر است که متد Flush سبب تخلیه کش نمی‌گردد. کار این متد اعمال کلیه تغییرات اعمالی موجود در کش به بانک اطلاعاتی است و بیشتر جهت هماهنگ سازی این دو مورد استفاده قرار می‌گیرد.

سؤال: آیا می‌توان سطح اول caching را غیرفعال کرد؟

پاسخ:بله.

توضیحات:

عموما کلیه ORMs جهت Batching یا Bulk data operations (برای مثال ثبت تعداد زیادی رکورد یا به روز رسانی تعداد بالایی از آن‌ها، یا نمایش فقط خواندنی تعداد زیادی رکورد و گزارشگیری از آن‌ها) کارآیی مطلوبی ندارند. نمونه‌ای از آن‌را در مبحث جاری ملاحظه کرده‌اید. هر شیءایی که به نحوی به سشن جاری وارد می‌شود تحت نظر قرار می‌گیرد و این مورد در تعداد بالای ثبت یا به روز رسانی رکوردها، یعنی کاهش سرعت و کارآیی، به علاوه مصرف بالای حافظه. به همین جهت باید به خاطر داشت که ORMs جهت سناریوهای OLTP مناسب هستند و کسانی که سرعت و کارآیی ORMs را با Batch processing اندازه گیری می‌کنند، کلا درکی از فلسفه‌ی وجودی ORMs و ساختار درونی آن‌ها ندارند!

خوشبختانه NHibernate با معرفی Stateless Sessions بر این مشکل فائق آمده است. در اینجا بجای ISession تنها کافی است از IStatelessSession استفاده گردد:

```
using (IStatelessSession statelessSession =
sessionFactory.OpenStatelessSession())
using (ITransaction transaction = statelessSession.BeginTransaction())
{
 //now insert 1,000,000 records!
}
```

در این حالت سیستم دو مزیت عمده را تجربه خواهد کرد: سرعت بالای ثبت اطلاعات با تعداد زیاد رکورد و همچنین مصرف پایین حافظه از آنجائیکه یک IStatelessSession ارجاعی را به اشیایی که بارگذاری می‌کند، در خود نگهداری نخواهد کرد.
تنها باید به خاطر داشت که در این حالت lazy loading پشتیبانی نمی‌شود و همچنین رخدادهای درونی NHibernate نیز لغو خواهند شد.

سطح دوم cache درNHibernate
عموما دو الگوی اصلی caching در برنامه‌ها وجود دارند: cache aside **و** cache through.
در الگوی cache through**، سیستم** caching **داخل** DAL **(که در اینجا همان** NHibernate **است)، تعبیه می‌شود؛ مانند سطح اول** caching **که پیشتر در مورد آن صحبت شد. در این حالت** cache **از دید سایر قسمت‌های برنامه مخفی است و** DAL **به صورت خودکار آن‌را مدیریت می‌کند.**
در الگوی cache aside **، کار مدیریت سیستم** caching **دستی است و خارج از** NHibernate **قرار می‌گیرد و** DAL **هیچگونه اطلاعی از وجود آن ندارد. در این حالت لایه** caching **موظف است تا هنگام به روز شدن بانک اطلاعاتی، اطلاعات خود را نیز به روز نماید. این لایه عموما توسط سایر شرکت‌ها یا گروه‌ها برنامه نویسی تهیه می‌شود.**
NHibernate **جهت سهولت کار با این نوع** cache providers **خارجی، نقاط تزریق ویژه‌ای را تدارک دیده است که به** second level cache **معروف است. هدف از** second level cache **فراهم آوردن دیدی کش شده از بانک اطلاعاتی است تا فراخوانی‌های کوئری‌ها به سرعت و بدون تماس با بانک اطلاعاتی صورت گیرد.**
در حال حاضر (زمان نگارش این مطلب)، entity framework **این لایه‌ی دوم** caching **یا به عبارتی دیگر، امکان تزریق ساده‌تر** cache providers **خارجی را به صورت توکار ارائه نمی‌دهد.**
در NHibernate **طول عمر** second level cache **در سطح** session factory **(یا به عبارتی طول عمر تمام برنامه) تعریف می‌شود و برخلاف سطح اولیه** caching **محدود به یک سشن نیست. در این حالت هر زمانیکه یک موجودیت به همراه** ID **منحصربفرد آن تحت نظر** NHibernate **قرارگیرد و همچنین سطح دوم** caching **نیز فعال باشد، این موجودیت در تمام سشن‌های برنامه بدون نیاز به مراجعه به بانک اطلاعاتی در دسترس خواهد بود (بنابراین باید دقت داشت که هدف از این سیستم، کار سریعتر با اطلاعاتی است که سطح دسترسی عمومی دارند).**

در ادامه لیستی از cache providers **خارجی مهیا جهت استفاده در سطح دوم** caching **را ملاحظه می‌نمائید:**

- AppFabric Caching Services**: بر اساس** Microsoft's AppFabric Caching Services **که یک پلتفرم** caching **محسوب می‌شود (+). (این پروژه پیشتر به نام** Velocity **معروف شده بود و قرار بود تنها برای** ASP.NET **ارائه شود که سیاست آن به گونه‌ای جامع‌تر تغییر کرده است)**
- MemCache**: بر اساس سیستم معروف** MemCached **تهیه شده است (+).**
- NCache**: (+)**
- ScaleOut**: (+)**
- Prevalence**: (+)**

- SysCache : بر اساس همان روش آشنای متداول در برنامه‌های ASP.NET به کمک System.Web.Caching.Cache کار می‌کند؛ یا به قولی همان IIS caching

- SysCache2 : همانند SysCache است با این تفاوت که SQL dependencies ویژه SQL Server را نیز پشتیبانی می‌کند.

- SharedCache : یک سیستم distributed caching نوشته شده برای دات نت است (+).

این موارد و پرویدرها جزو پروژه‌ی nhcontrib در سایت سورس فورج هستند (+).

مطالب تکمیلی:

- مستندات NHibernate

- توضیحات مفصلی در مورد سطح اول و دوم caching در NHibernate

- مقایسه‌ای در مورد مبحث caching در EF و NHibernate

- چگونه fluent NHibernate را جهت استفاده از سطح دوم caching تنظیم کنیم؟

- توضیحات جامعی در مورد استفاده از SysCache

اعمال تغییرات سفارشی به ویژگی AutoMapping در Fluent NHibernate

با کمک Fluent NHibernate می‌توان نگاشت‌ها را به دو صورت خودکار و یا دستی تعریف کرد. در حالت خودکار، روابط بین کلاس‌ها بررسی شده و بدون نیاز به تعریف هیچگونه ویژگی (attribute) خاصی بر روی فیلدها، امکان تشخیص خودکار حالت‌های کلید خارجی، روابط یک به چند، چند به چند و امثال آن وجود دارد. یا اگر نیاز باشد تا اسکریپت تولیدی جهت به روز رسانی بانک اطلاعاتی، طول خاصی را به فیلدی اعمال کند می‌توان از ویژگی‌های NHibernate validator استفاده کرد؛ مانند تعریف طول و نال نبودن یک فیلد که علاوه بر بکارگیری اطلاعات آن در حین تعیین اعتبار ورودی دریافتی، بر روی نحوه‌ی به روز رسانی بانک اطلاعاتی هم تاثیر گذار است:

```
public class Product
{
    public virtual int Id { set; get; }

    [Length(120)]
    [NotNullNotEmpty]
    public virtual string Name { get; set; }

    public virtual decimal UnitPrice { get; set; }
}
```

این نگاشت خودکار یا AutoMapping، تقریبا در 90 درصد موارد کافی است. فیلد Id را بر اساس یک سری پیش فرض‌هایی که این مورد هم قابل تنظیم است به صورت primary key تعریف می‌کند، طول فیلدها و نحوه‌ی پذیرفتن نال آن‌ها، از ویژگی‌های NHibernate validator گرفته می‌شود و روابط بین کلاس‌ها به صورت خودکار به روابط یک به چند و مانند آن ترجمه می‌شود و نیازی نیست تا کلاسی جداگانه را جهت مشخص سازی صریح این موارد تهیه کرد، یا ویژگی مشخص کننده‌ی دیگری را به فیلدها افزود. اما اگر برای مثال بخواهیم در

این کلاس فیلد Name **را به صورت** Unique **معرفی کنیم چه باید کرد؟ به عبارتی تمام آنچه که ویژگی** AutoMapping **در** Fluent NHibernate **انجام می‌دهد، بسیار هم عالی؛ اما فقط می‌خواهیم مقادیر یک فیلد منحصربفرد باشد. برای این منظور اینترفیس** IAutoMappingOverride **تدارک دیده شده است:**

```
public class ProductCustomMappings : IAutoMappingOverride<Product>
{
    public void Override(AutoMapping<Product> mapping)
    {
        mapping.Id(u => u.Id).GeneratedBy.Identity(); //است ضروري
        mapping.Map(p => p.Name).Unique();
    }
}
```

در حالت استفاده از اینترفیس IAutoMappingOverride **مشخص سازی نحوه‌ی تولید** primary key **ضروری است و سپس برای نمونه، فیلد** Name **به صورت منحصربفرد تعریف می‌گردد. در اینجا کل عملیات هنوز از روش** AutoMapping **پیروی می‌کند اما فیلد** Name **علاوه بر اعمال ویژگی‌های** NHibernate validator**، به صورت منحصربفرد نیز معرفی خواهد شد.**

قسمت هجدهم

نحوه‌ي نگاشت فيلدهاي فرمول درFluent NHibernate

اگر با SQL Server کار کرده باشید حتما با مفهوم و امکان Computed columns (فیلدهای محاسبه شده) آن آشنایی دارید. چقدر خوب می‌شد اگر این امکان برای سایر بانک‌های اطلاعاتی که از تعریف فیلدهای محاسبه شده پشتیبانی نمی‌کنند، نیز مهیا می‌شد. زیرا یکی از اهداف مهم استفاده‌ی صحیح از ORMs ، مستقل شدن برنامه از نوع بانک اطلاعاتی است. برای مثال امروز می‌خواهیم با MySQL کار کنیم، ماه بعد شاید بخواهیم یک نسخه‌ی سبک‌تر مخصوص کار با SQLite را ارائه دهیم. آیا باید قسمت دسترسی به داده برنامه را از نو بازنویسی کرد؟ اینکار در NHibernate فقط با تغییر نحوه‌ی اتصال به بانک اطلاعاتی میسر است و نه بازنویسی کل برنامه (و صد البته شرط مهم و اصلی آن هم این است که از امکانات ذاتی خود NHibernate استفاده کرده باشید. برای مثال وسوسه‌ی استفاده از رویه‌های ذخیره شده را فراموش کرده و به عبارتی ORM مورد استفاده را به امکانات ویژه‌ی یک بانک اطلاعاتی گره نزده باشید).
خوشبختانه در NHibernate امکان تعریف فیلدهای محاسباتی با کمک تعریف نگاشت خواص به صورت فرمول مهیا است. برای توضیحات بیشتر لطفا به مثال ذیل دقت بفرمائید:
در ابتدا کلاس کاربر تعریف می‌شود:

```
using System;
using NHibernate.Validator.Constraints;

namespace FormulaTests.Domain
{
    public class User
    {
        public virtual int Id { get; set; }

        [NotNull]
        public virtual DateTime JoinDate { set; get; }

        [NotNullNotEmpty]
        [Length(450)]
        public virtual string FirstName { get; set; }

        [NotNullNotEmpty]
        [Length(450)]
        public virtual string LastName { get; set; }

        [Length(900)]
        public virtual string FullName { get; private set; } //از طریق تعریف فرمول گردد‌می دهی مقدار

        public virtual int DayOfWeek { get; private set; }//از طریق تعریف فرمول گردد‌می دهی مقدار
    }
}
```

در این کلاس دو خاصیت FullName و DayOfWeek به صورت فقط خواندنی به کمک private set ذکر شده، تعریف گردیده‌اند.
قصد داریم روی این دو خاصیت فرمول تعریف کنیم:

```
using FluentNHibernate.Automapping;
using FluentNHibernate.Automapping.Alterations;

namespace FormulaTests.Domain
{
    public class UserCustomMappings : IAutoMappingOverride<User>
    {
```

```
        public void Override(AutoMapping<User> mapping)
        {
            mapping.Id(u => u.Id).GeneratedBy.Identity(); //است ضروري
            mapping.Map(x => x.DayOfWeek).Formula("DATEPART(dw, JoinDate) -
1");
            mapping.Map(x => x.FullName).Formula("FirstName + ' ' + LastName");
        }
    }
}
```

نحوه‌ی انتساب فرمول‌های مبتنی بر SQL را در نگاشت فوق ملاحظه می‌نمائید. برای مثال FullName از جمع دو فیلد نام و نام خانوادگی حاصل خواهد شد و DayOfWeek از طریق فرمول SQL دیگری که ملاحظه می‌نمائید (یا هر فرمول SQL دلخواه دیگری که صلاح می‌دانید).

اکنون اگر Fluent NHibernate را وادار به تولید اسکریپت متناظر با این دو کلاس کنیم حاصل به صورت زیر خواهد بود:

```
    create table Users (
       UserId INT IDENTITY NOT NULL,
      JoinDate DATETIME not null,
      FirstName NVARCHAR(450) not null,
      LastName NVARCHAR(450) not null,
      primary key (UserId)
   )
```

همانطور که ملاحظه می‌کنید در اینجا خبری از دو فیلد محاسباتی تعریف شده نیست. این فیلدها در تعاریف نگاشت‌ها به صورت خودکار ظاهر می‌شوند:

```
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
 default-access="property" auto-import="true" default-cascade="none" default-
lazy="true">
  <class xmlns="urn:nhibernate-mapping-2.2" mutable="true"
  name="FormulaTests.Domain.User, FormulaTests, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null" table="Users">
    <id name="Id" type="System.Int32" unsaved-value="0">
      <column name="UserId" />
      <generator class="identity" />
    </id>
    <property name="DayOfWeek" formula="DATEPART(dw, JoinDate) - 1"
type="System.Int32" />
    <property name="FullName" formula="FirstName + ' ' + LastName"
type="System.String" />
    <property name="JoinDate" type="System.DateTime">
      <column name="JoinDate" />
    </property>
    <property name="FirstName" type="System.String">
      <column name="FirstName" />
    </property>
    <property name="LastName" type="System.String">
      <column name="LastName" />
    </property>
  </class>
</hibernate-mapping>
```

اکنون اگر کوئری زیر را در برنامه اجرا نمائیم:

```
var list = session.Query<User>.ToList();
foreach (var item in list)
{
   Console.WriteLine("{0}:{1}", item.FullName, item.DayOfWeek);
}
```

به صورت خودکار به SQL ذیل ترجمه خواهد شد و اکنون نحوه‌ی بکارگیری فیلدهای فرمول، بهتر مشخص می‌گردد:

```
select
       user0_.UserId as UserId0_,
       user0_.JoinDate as JoinDate0_,
       user0_.FirstName as FirstName0_,
       user0_.LastName as LastName0_,
       DATEPART(user0_.dw, user0_.JoinDate) - 1 as formula0_, ---  همان فرمول
تعریف شده است          user0_.FirstName + ' ' + user0_.LastName as formula1_ ---
از طریق فرمول تعریف شده حاصل گردیده است
    from
        Users user0_
```

به روز رسانی ارجاعات یک اسمبلی دارای امضای دیجیتال بدون کامپایل مجدد

سؤال: امروز NHibernate به روز شده اما Fluent NHibernate خیر! چکار باید کرد؟!

Fluent NHibernate کتابخانه‌ای است جهت رهایی برنامه نویس‌ها از نوشتن فایل‌های XML نگاشت کلاس‌ها به جداول به همراه قابلیت‌های دیگری مانند نگاشت خودکار و غیره. بنابراین این کتابخانه بدون NHibernate اصلی بدون کاربرد است. تیم توسعه آن هم با تیم اصلی NHibernate یکی نیست. عموما NHibernate به روز می‌شود اما Fluent NHibernate ممکن است تا دو ماه بعد از آن هم به روز نشود. در این مواقع چه باید کرد؟

دو کار را می‌توان انجام داد:

الف) سورس Fluent NHibernate را دریافت کنیم و سپس ارجاعات قبلی به NHibernate قدیمی را حذف و ارجاعاتی را به اسمبلی‌های جدید آن اضافه و پروژه را کامپایل کنیم.

Fluent NHibernate در طی این مدت به اندازه کافی رشد کرده و به قولی پخته است. کاری را هم که ادعا می‌کند به خوبی انجام می‌دهد. اما چون اسمبلی‌های اصلی NHibernate و همچنین Fluent NHibernate دارای امضای دیجیتال هستند، نمی‌توان از اسمبلی‌های جدید NHibernate به همراه Fluent NHibernate قدیمی استفاده کرد. خطای حاصل شبیه به عبارات ذیل خواهد بود:

```
System.IO.FileLoadException: Could not load file or assembly 'nameOfAssembly',
Version=specificVersion, Culture=neutral, PublicKeyToken=publicKey' or one of
it's dependencies.
The located assembly's manifest definition does not match the assembly
reference.
(Exception from HRESULT: 0x80131040)
```

حذف ارجاعات به NHibernate قدیمی و افزودن مجدد ارجاعات به فایل‌های جدید و کامپایل نهایی پروژه یک راه حل است.

ب) راه حل دیگر استفاده از ویژگی bindingRedirect است بدون دریافت سورس، حذف و افزودن ارجاعات و کامپایل مجدد:

```
<runtime>
  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
    <dependentAssembly>
      <assemblyIdentity name="NHibernate"
                        publicKeyToken="aa95f207798dfdb4"
                        culture="neutral" />
      <bindingRedirect oldVersion="3.0.0.4000"
                       newVersion="3.1.0.4000"/>
    </dependentAssembly>
  </assemblyBinding>
</runtime>
```

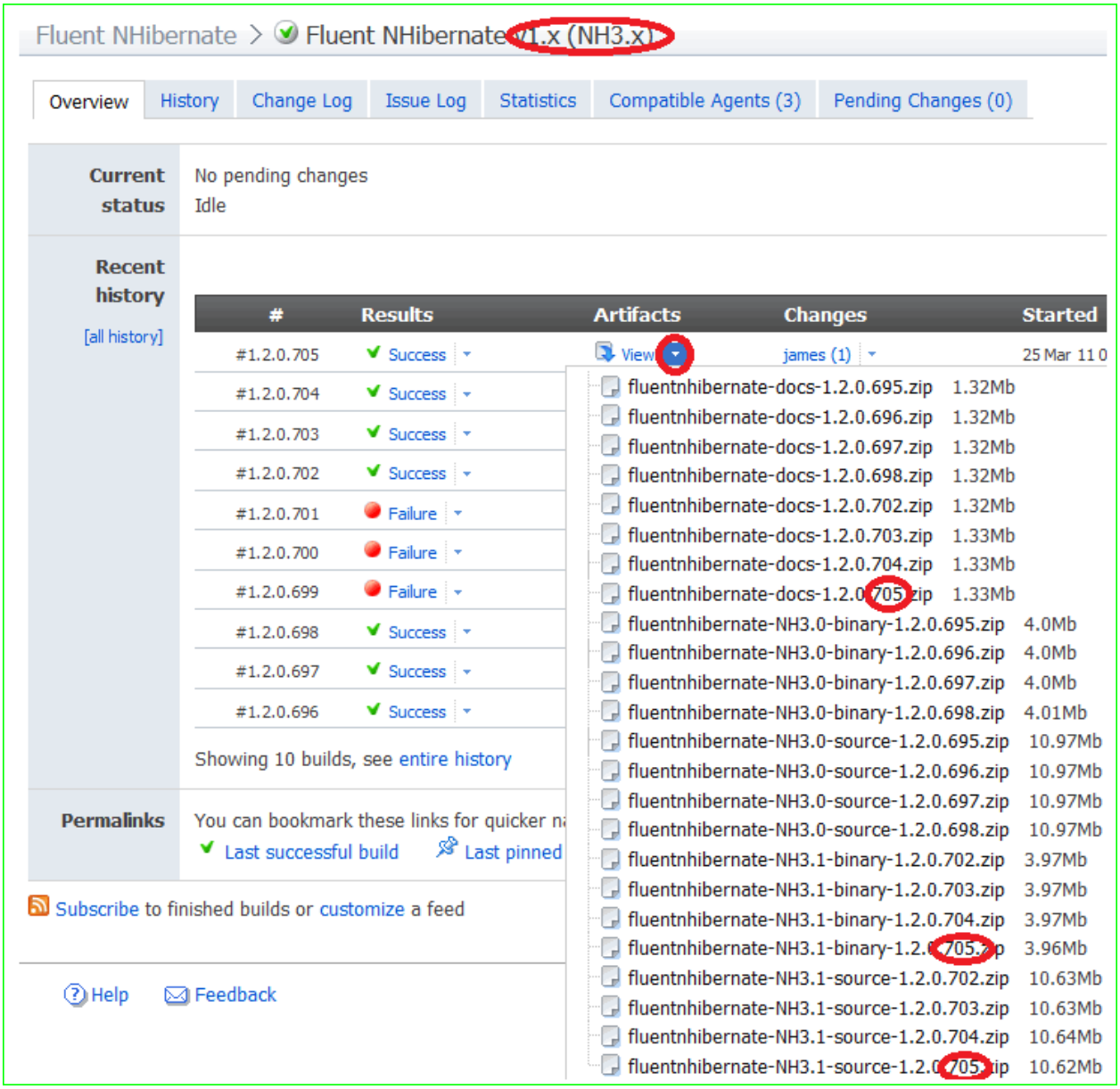
در این مثال، پس از افزودن تعاریف فوق به فایل config برنامه، به سادگی می‌توان از اسمبلی اصلی NHibernate دارای نگارش 3.1.0.4000 به جای اسمبلی قدیمی‌تر 3.0.0.4000 آن استفاده کرد (همان نگارشی که Fluent NHibernate ما بر اساس

آن کامپایل شده) و دیگر نیازی هم به کامپایل مجدد پروژه‌ای که از یک اسمبلی قدیمی Fluent NHibernate استفاده می‌کند، نخواهد بود.

قسمت نوزدهم

مکان اصلي يافتن آخرين نگارش‌هاي Fluent NHibernate

اگر علاقمند باشید که به آخرین نگارش‌های Fluent NHibernate دسترسی داشته باشید، مکان اصلی نگهداری و Build آن‌ها در سایت teamcity.codebetter.com می‌باشد. ثبت نام در آن رایگان است و سپس در آدرس ذیل می‌توانید آخرین Build ها را مشاهده و دریافت کنید:



Fluent NHibernate v1.x (NH3.x)

برای نمونه:

- fluentnhibernate-NH3.1-source-1.2.0.705.zip

- fluentnhibernate-NH3.1-binary-1.2.0.705.zip
- fluentnhibernate-docs-1.2.0.705.zip

قسمت بیستم

NH 3.2 و تاثیر آن بر آینده‌ی FHN

در این عنوان، NH همان NHibernate است و FHN همان Fluent NHibernate

نگارش آزمایشی NH 3.2 هم اکنون در دسترس است و یکی از مهمترین مباحثی را که پوشش داده، جایگزین کردن فایل‌های XML تهیه نگاشت‌ها با کدنویسی است. دقیقا چیزی شبیه به Fluent NHibernate البته اینبار از یک کتابخانه دیگر به نام ConfOrm کدها یکی شده‌اند.
باید توجه داشت که نگارش 3.2 خاصیت AutoMapping مربوط به FHN را پشتیبانی نمی‌کند (یا هنوز در این نگارش به این حد نرسیده است)، بنابراین نمی‌تواند جایگزین صد در صدی برای FHN باشد اما باز هم تا حدود 75 درصد کار FHN را پوشش می‌دهد و می‌تواند علاقمندان را از این وابستگی خارجی (!) نجات دهد.
و ... این مساله نویسنده‌ی اصلی FHN را کمی دلگیر کرده است که آیا FHN را ادامه دهد یا خیر. اصل مطلب رو می‌تونید اینجا بخونید.
نظر بعضی‌ها هم در این بین این بوده!

```
ConfOrm looks like lipstick on a pig as far as fluent interfaces go
```

قسمت بيست و يكم

فعال سازي سطح دوم كش درFluent Nhibernate

سطح اول كش در NHibernate در يك تراكنش معنا پيدا مي‌كند (+)؛ اما نتايج حاصل از اعمال سطح دوم (+) آن، در اختيار تمام تراكنش‌هاي جاري برنامه خواهند بود. در ادامه قصد داريم نحوه فعال سازي سطح دوم كش NHibernate را توسط Fluent NHibernate بررسي كنيم.

الف) دريافت كش پروايدر

براي اين منظور به صفحه اصلي آن در سايت سورس فورج مراجعه نمائيد(+). اگر به علت تحريم‌ها امكان دريافت فايل‌هاي مرتبط را نداشتيد از اين برنامه استفاده كنيد(+). پس از دريافت، مي‌خواهيم نحوه فعال سازي NHibernate.Caches.SysCache.dll را بررسي كنيم (اين اسمبلي، در برنامه‌هاي وب و دسكتاپ بدون مشكل كار مي‌كند).

ب) اعمال به قسمت تعاريف اوليه

پس از دريافت اسمبلي NHibernate.Caches.SysCache.dll و افزودن ارجاعي به آن، اكنون نوبت به معرفي آن به تنظيمات Fluent NHibernate مي‌باشد. اين‌كار هم بسيار ساده است: