عنوان: OutputCache در ASP.NET MVC
نویسنده: مرتضی دلیل
تاریخ: ۱۳۹۳/۰۴/۲۸ ۱۴:۱۵
آدرس: www.dotnettips.info
گروه‌ها: MVC, Performance, Caching

### مقدمه

OutputCaching باعث می‌شود خروجیِ یک اکشن متد در حافظه نگهداری شود. با اعمال این نوع کشینگ، ASP.NET در خواست‌های بعدی به این اکشن را تنها با بازگرداندن همان مقدار قبلیِ نگهداری شده در کش، پاسخ می‌دهد. در حقیقت با OutputCaching از تکرار چند باره کد درون یک اکشن در فراخوانی‌های مختلف جلوگیری کرده‌ایم. کش کردن باعث می‌شود که کارایی و سرعت سایت افزایش یابد؛ اما باید دقت کنیم که چه موقع و چرا از کش کردن استفاده میکنیم و چه موقع باید از این کار امتناع کرد.

### فواید کش کردن

- انجام عملیات هزینه دار فقط یکبار صورت میگیرد. (هزینه از لحاظ فشار روی حافظه سرور و کاهش سرعت بالا آمدن سایت)

- بار روی سرور در زمان‌های پیک کاهش می‌یابد.

- سرعت بالا آمدن سایت بیشتر میشود.

### چه زمانی باید کش کرد؟

- وقتی محتوای نمایشی برای همه کاربران یکسان است.

- وقتی محتوای نمایشی برای نمایش داده شدن، فشار زیادی روی سرور تحمیل میکند.

- وقتی محتوای نمایشی به شکل مکرر در طول روز باید نمایش داده شود.

- وقتی محتوای نمایشی به طور مکرر آپدیت نمی‌شود. (در مورد تعریف کیفیت "مکرر"، برنامه نویس بهترین تصمیم گیرنده است)

### طرح مساله

فرض کنید صفحه اول سایت شما دارای بخش‌های زیر است :

خلاصه اخبار بخش علمی، خلاصه اخبار بخش فرهنگی ، ده کامنت آخر، لیستی از کتگوری‌های موجود در سایت.

روش‌های مختلفی برای کوئری گرفتن وجود دارد، به عنوان مثال ما به کمک یک یا چند کوئری و توسط یک ViewModel جامع، میخواهیم اطلاعات را به سمت ویو ارسال کنیم. پس در اکشن متد Index ، حجم تقریبا کمی از اطلاعات را باید به کمک کوئری(کوئری های) تقریبا پیچیده ای دریافت کنیم و اینکار به ازای هر ریکوئست هزینه دارد و فشار به سرور وارد خواهد شد. از طرفی میدانیم صفحه اول ممکن است در طول یک یا چند روز تغییر نکند و همچنین شاید در طول یکساعت چند بار تغییر کند! به هر حال در جایی از سایت قرار داریم که کوئری (کوئری های) مورد نظر زیاد صدا زده میشوند ، در حقیقت صفحه اول احتمالا بیشترین فشار ترافیکی را در بین صفحات ما دارد، البته این فقط یک احتمال است و ما دقیقا از این موضوع اطلاع نداریم.

یکی از راه‌های انجام یک کش موفق و دانستن لزوم کش کردن، این است که دقیقا بدانیم ترافیک سایت روی چه صفحه ای بیشتر است. در واقع باید بدانیم در کدام صفحه "هزینه‌ی اجرای عملیات موجود در کد" بیشترین است.

فشار ترافیکی(ریکوئست‌های زیاد) و آپدیت‌های روزانه‌ی دیتابیس را، در دو کفه ترازو قرار دهید؛ چه کار باید کرد؟ این تصمیمی است که شما باید بگیرید. نگرانی خود را در زمینه آپدیت‌های روزانه و ساعتی کمتر کنید؛ در ادامه راهی را معرفی میکنیم که آپدیت‌های هر از گاهِ شما، در پاسخِ ریکوئست‌ها دیده شوند. کمی کفه‌ی کش کردن را سنگین کنید.

به هر حال، فعال کردن قابلیت کش کردن برای یک اکشن، بسیار ساده است، کافیست ویژگی ( attribute ) آن را بالای اکشن بنویسید :

```
[OutputCache(Duration = "60", Location = OutputCacheLocation.Server)]
        public ActionResult Index()
        {
            کوئری یا کوئری‌های لازم برای استفاده در صفحه اصلی و تبدیل آن به یک ویو مدل جامع//
        }
```

```
[OutputCache(CacheProfile = "FirstPageIndex",Location=OutputCacheLocation.Server)]
        public ActionResult Index()
        {
            کوئری یا کوئری‌های لازم برای استفاده در صفحه اصلی و تبدیل آن به یک ویو مدل جامع//
        }
```

دو روش فوق برای کش کردن خروجی Index از لحاظ عملکرد یکسان است، به شرطی که در حالت دوم در وب کانفیگ و در بخش system.web آن ، یک پروفایل ایجاد کنیم کنیم :

```
<caching>
      <outputCacheSettings>
        <outputCacheProfiles>
          <add name="FirstPageIndex" duration="60"/>
        </outputCacheProfiles>
      </outputCacheSettings>
    </caching>
```

در حالت دوم ما یک پروفایل برای کشینگ ساخته ایم و در ویژگی بالای اکشن متد، آن پروفایل را صدا زده ایم. از لحاظ منطقی در حالت دوم، چون امکان استفاده مکرر از یک پروفایل در جاهای مختلف فراهم شده، روش بهتری است. محل ذخیره کش نیز در هر دو حالت سرور تعریف شده است.

برای تست عملیات کشینگ، کافیست یک BreakPoint درون Index قرار دهید و برنامه را اجرا کنید. پس از اجرا، برنامه روی Break Point می‌ایستد و اگر F5 را بزنیم، سایت بالا می‌آید. بار دیگر صفحه را رفرش کنیم، **اگر این "بار دیگر" در کمتر از 60 ثانیه پس از رفرش قبلی اتفاق افتاده باشد برنامه روی Break Point متوقف نخواهد شد،** چون خروجی اکشن، در کش بر روی سرور ذخیره شده است و این یعنی ما فشار کمتری به سرور تحمیل کرده ایم، صفحه با سرعت بالاتری در دسترس خواهد بود.

ما از تکرار اجرای کد جلوگیری کرده ایم و عدم اجرای کد بهترین نوع بهینه سازی برای یک سایت است. [اسکات الن، پلورال سایت]

**چطور زمان مناسب برای کش کردن یک اکشن را انتخاب کنیم؟**

- **کشینگ با زمان کوتاه** ؛ فرض کنید زمان کش را روی 1 ثانیه تنظیم کرده اید. این یعنی اگر ریکوئست هایی به یک اکشن ارسال شود و همه در طول یک ثانیه اتفاق بیفتد، آن اکشن فقط برای بار اول اجرا میشود، و در بارهای بعد(در طول یک ثانیه) فقط محتوای ذخیره شده در آن یک اجرا، بدون اجرای جدید، نمایش داده میشود. پس سرور شما فقط به یک ریکوئست در ثانیه در طول روز جواب خواهد داد و ریکوئست‌های تقریبا همزمان دیگر، در طول همان ثانیه، از نتایج آن ریکوئست (اگر موجود باشد) استفاده خواهند کرد

- **کشینگ با زمان طولانی** ؛ ما در حقیقت با اینکار منابع سرور را حفاظت میکنیم، چون عملیاتِ هزینه دار(مثل کوئری‌های حجیم) تنها یکبار در طول زمان کشینگ اجرا خواهند شد. مثلا اگر تنظیم زمان روی عدد 86400 تنظیم شود(یک روز کامل)، پس از اولین

ریکوئست به اکشن مورد نظر، تا 24 ساعت بعد، این اکشن اجرا نخواهد شد و فقط خروجی آن نمایش داده خواهد شد. آیا دلیلی دارد که یک کوئری هزینه دار را که قرار نیست خروجی اش در طول روز تغییر کند به ازای هر ریکوئست یک بار اجرا کنیم؟

**اگر اطلاعات موجود در دیتابیس را تغییر دهیم چه کار کنیم که کشینگ رفرش شود؟**

فرض کنید در همان مثال ابتدای این مقاله، شما یک پست به دیتابیس اضافه کرده اید، اما چون مثلا duration مربوط به کشینگ را روی 86400 تعریف کرده اید تا 24 ساعت از زمان ریکوئست اولیه نگذرد، سایت آپدیت نخواهد شد و محتوا همان چیزهای قبلی باقی خواهند ماند. اما چاره چیست؟

کافیست در بخش ادمین، وقتی که یک پست ایجاد میکنید یا پستی را ویرایش میکند در اکشن‌های مرتبط با Create یا Edit یا Delete چنین کدی را پس از فرمان ذخیره تغییرات در دیتابیس، بنویسید:

```
Response.RemoveOutputCacheItem(Url.Action("index", "home"));
```

واضح است که ما داریم کشینگ مرتبط با یک اکشن متد مشخص را پاک میکنیم. با اینکار در اولین ریکوئست پس از تغییرات اعمال شده در دیتابیس، ASP.NET MVC چون میبیند گشی برای این اکشن وجود ندارد، متد را اجرا میکند و کوئری‌های درونش را خواهد دید و اولین ریکوئست پیش از گش شدن را انجام خواهد داد. با اینکار کشینگ ریست شده است و پس از این ریکوئست و استخراج اطلاعات جدید، زمان کشینگ صفر شده و آغاز میشود.

میتوانید یک دکمه در بخش ادمین سایت طراحی کنید که هر موقع دلتان خواست کلیه کش‌ها را به روش فوق پاک کنید! تا اپلیکیشن منتظر ریکوئست‌های جدید بماند و کش‌ها دوباره ایجاد شوند.

**جمع بندی**

ویژگی OutputCatch دارای پارامترهای زیادیست و در این مقاله فقط به توضیح عملکرد این اتریبیوت اکتفا شده است. بطور کلی این مبحث ظاهر ساده ای دارد، ولی نحوه استفاده از کشینگ کاملا وابسته به هوش برنامه نویس است و پیچیدگی‌های مرتبط با خود را دارد. در واقع خیلی مشکل است که بتوانید یک زمان مناسب برای کش کردن تعیین کنید. باید برنامه خود را در یک محیط شبیه سازی تحت بار قرار دهید و به کمک اندازه گیری و محاسبه به یک قضاوت درست از میزان زمان کش دست پیدا کنید. گاهی متوجه خواهید شد، از مقدار زیادی از حافظه سیستم برای کش کردن استفاده کرده اید و در حقیقت آنقدر ریکوئست ندارید که احتیاج به این هزینه کردن باشد.

یکی از روش‌های موثر برای دستیابی به زمان بهینه برای کش کردن استفاده از CacheProfile درون وب کانفیگ است. وقتی از کشینگ استفاده میکنید، در همان ابتدا مقدار زمانی مشخص برای آن در نظر نگرفته اید(در حقیقت مقدار زمان مشخصی نمیدانید) پس مجبور به آزمون و خطا و تست و اندازه گیری هستید تا بدانید چه مقدار زمانی را برای چه پروفایلی قرار دهید. مثلا پروفایل هایی به شکل زیر تعریف کرده اید و نام آنها را به اکشن‌های مختلف نسبت داده اید. به راحتی میتوانید از طریق دستکاری وب کانفیگ مقادیر آن را تغییر دهید تا به حالت بهینه برسید، بدون آنکه کد خود را دستکاری کنید.

```
<caching>
      <outputCacheSettings>
        <outputCacheProfiles>
          <add name="Long" duration="86400"/>
          <add name="Average" duration="43600"/>
          <add name="Short" duration="600"/>
        </outputCacheProfiles>
      </outputCacheSettings>
    </caching>
```

برای مطالعه جزئیات بیشتر در مورد OutputCaching مقالات زیر منابع مناسبی هستند.

[ اینجا ] و [ اینجا ]

## نظرات خوانندگان

نویسنده: ابوالفضل رجب پور
تاریخ: ۱۳۹۳/۰۴/۲۹ ۹:۵۳

سلام
یک ابهام در یک مثال واقعی مثلا سایت خبری.
اگر بخواهیم خروجی اکشن اخبار رو کش کنیم، و در عین حال تعداد بازدید از هر خبر رو هم ثبت کنیم، چطور باید این کار رو انجام داد؟

نویسنده: مرتضی دلیل
تاریخ: ۱۳۹۳/۰۴/۲۹ ۱۰:۲۴

قاعدتا اگر اکشن مربوط به نمایش هر خبر مستقل از اکشن نمایش "آخرین اخبار" باشد، با کش کردن اکشن "آخرین اخبار" مشکلی برای اکشن نمایش دهنده هر خبر بوجود نخواهد آمد و میتوان در این اکشن، متدها یا عملیات مورد نظر را بدون نگرانی اعمال کرد. (اگر منظور از "ثبت" ، ذخیره‌ی اطلاعات باشد)

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۴/۲۹ ۱۰:۲۶

- با استفاده از jQuery که یک بحث سمت کاربر است، زمانیکه صفحه نمایش داده شد، یک درخواست Ajax ایی به اکشن متدی خاص، جهت به روز رسانی تعداد بار مشاهده ارسال کنید. به این روش client side tracking هم می‌گویند (کل اساس کار Google analytics به همین نحو است).
- روش دوم استفاده از Donut Caching است. در یک چنین حالتی، کد زیر مجاز است:

```
[LogThis]
[DonutOutputCache(Duration=5, Order=100)]
public ActionResult Index()
```

اطلاعات بیشتر

نویسنده: ایلیا اکبری فرد
تاریخ: ۱۳۹۳/۰۸/۱۲ ۱۶:۵۲

با سلام.
متدی به روش زیر در کنترلر خود ایجاد کرده ام:

```
[OutputCache(Duration = (7 * 24 * 60 * 60), VaryByParam = "none")]
[AllowAnonymous]
public virtual ActionResult Notification()
{
      ....
}
```

و در قسمت ادمین سیستم که در یک area جداگانه قرار دارد در اکشن متد خود اینگونه نوشتم:

```
Response.RemoveOutputCacheItem(Url.Action("Notification", "Article"));
Response.RemoveOutputCacheItem(Url.Action("Notification", "Article", new { area = "" }));
```

هیچکدام از دو روش بالا برایم جواب نمی‌دهد و کش خالی نمی‌شود. علت چیست؟

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۰۸/۱۲ ۱۸:۴۱

آیا از Html.RenderAction برای نمایش آن استفاده کردید؟ اگر بله، متد یاد شده تاثیری روی کش آن نداره، چون نحوه‌ی کش شدن child actionها متفاوته.

نویسنده: ایلیا اکبری فرد
تاریخ: ۱۳۹۳/۰۸/۱۲ ۱۹:۱۵

بله. راه حل مشکل چیست؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۸/۱۳ ۱۵:۱۴

به این صورت؛ البته این روش کش تمام child actionها را با هم پاک می‌کند:

```
OutputCacheAttribute.ChildActionCache = new MemoryCache("NewRandomStringNameToClearTheCache");
```

عنوان: **بهینه سازی سرعت یافت ویوها با سفارشی سازی Lookup Caching در Razor View Engine**
نویسنده: سید مهران موسوی
تاریخ: ۱۳۹۳/۰۵/۰۲ ۹:۴۰
آدرس: www.dotnettips.info
گروه‌ها: MVC, Razor, Performance, Caching

در این مقاله سعی داریم تا سرعت یافت و جستجوی Viewهای متناظر با هر اکشن را در View Engine، با پیاده سازی قابلیت Caching نتیجه یافت آدرس فیزیکی viewها در درخواست‌های متوالی، افزایش دهیم تا عملا بازده سیستم را تا حدودی بهبود ببخشیم.

طی مطالعاتی که بنده بر روی سورس MVC داشتم، به صورت پیش فرض، در زمانیکه پروژه در حالت Release اجرا می‌شود، نتیجه حاصل از یافت آدرس فیزیکی ویوهای متناظر با اکشن متدها در Application cache ذخیره می‌شود (HttpContext.Cache). این امر سبب اجتناب از عمل یافت چند باره بر روی آدرس فیزیکی ویوها در درخواست‌های متوالی ارسال شده برای رندر یک ویو خواهد شد.

نکته ای که وجود دارد این هست که علاوه بر مفید بودن این امر و بهبود سرعت در درخواست‌های متوالی برای اکشن متدها، این عمل با توجه به مشاهدات بنده از سورس MVC علاوه بر مفید بودن، تا حدودی هزینه بر هم هست و هزینه‌ای که متوجه سیستم می‌شود شامل مسائل مدیریت توکار حافظه کش توسط MVC است که مسائلی مانند سیاستهای مدیریت زمان انقضاء مداخل موجود در حافظه‌ی کش اختصاص داده شده به Lookup Cahching و مدیریت مسائل thread-safe و ... را شامل می‌شود.

همانطور که می‌دانید، معمولا تعداد ویوها اینقدر زیاد نیست که Caching نتایج یافت مسیر فیزیکی view ها، حجم زیادی از حافظه Ram را اشغال کند پس با این وجود به نظر می‌رسد که اشغال کردن این میزان اندک از حافظه در مقابل بهبود سرعت، قابل چشم پوشی است و سیاست‌های توکار نامبرده فقط عملا تاثیر منفی در روند Lookup Caching پیشفرض MVC خواهند گذاشت. برای جلوگیری از تاثیرات منفی سیاست‌های نامبرده و عملا بهبود سرعت Caching نتایج Lookup آدرس فیزیکی ویوها میتوانیم یک لایه Caching سطح بالاتر به View Engine اضافه کنیم .

خوشبختانه تمامی View Engineهای MVC شامل Web Forms و Razor از کلاس VirtualPathProviderViewEngine مشتق شده‌اند که نکته مثبت که توسعه Caching اختصاصی نامبرده را برای ما مقدور می‌کند. در اینجا خاصیت ( Property ) قابل تنظیم ViewLocationCache از نوع IViewLocationCache هست .

بنابراین ما یک کلاس جدید ایجاد کرده و از اینترفیس IViewLocationCache مشتق میکنیم تا به صورت دلخواه بتوانیم اعضای این اینترفیس را پیاده سازی کنیم .

خوب؛ بنابر این اوصاف، من کلاس یاد شده را به شکل زیر پیاده سازی کردم :

```
public class CustomViewCache : IViewLocationCache
    {

        private readonly static string s_key = "_customLookupCach" + Guid.NewGuid().ToString();
        private readonly IViewLocationCache _cache;

        public CustomViewCache(IViewLocationCache cache)
        {
            _cache = cache;
        }

        private static IDictionary<string, string> GetRequestCache(HttpContextBase httpContext)
        {
            var d = httpContext.Cache[s_key] as IDictionary<string, string>;
            if (d == null)
            {
                d = new Dictionary<string, string>();
                httpContext.Cache.Insert(s_key, d, null, Cache.NoAbsoluteExpiration, new TimeSpan(0,
15, 0));
            }
            return d;
        }
```

```
        public string GetViewLocation(HttpContextBase httpContext, string key)
        {
            var d = GetRequestCache(httpContext);
            string location;
            if (!d.TryGetValue(key, out location))
            {
                location = _cache.GetViewLocation(httpContext, key);
                d[key] = location;
            }
            return location;
        }

        public void InsertViewLocation(HttpContextBase httpContext, string key, string virtualPath)
        {
            _cache.InsertViewLocation(httpContext, key, virtualPath);
        }
    }
```

و به صورت زیر می‌توانید از آن استفاده کنید:

```
protected void Application_Start() {
    ViewEngines.Engines.Clear();
    var ve = new RazorViewEngine();
    ve.ViewLocationCache = new CustomViewCache(ve.ViewLocationCache);
    ViewEngines.Engines.Add(ve);
    ...
}
```

نکته: فقط به یاد داشته باشید که اگر View جدیدی اضافه کردید یا یک View را حذف کردید، برای جلوگیری از بروز مشکل، حتما و حتما اگر پروژه در مراحل توسعه بر روی IIS قرار دارد app domain را ری‌استارت کنید تا حافظه کش مربوط به یافت‌ها پاک شود (و به روز رسانی) تا عدم وجود آدرس فیزیکی View جدید در کش، شما را دچار مشکل نکند.

## نظرات خوانندگان

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۰۵/۰۲ ۹:۵۶

ضمن تشکر از ایده‌ای که مطرح کردید. طول عمر httpContext.Items فقط محدوده به یک درخواست و پس از پایان درخواست از بین می‌ره. مثلا یکی از کاربردهاش ذخیره اطلاعات Unit of work در طول یک درخواست هست و بعد از بین رفتن خودکار آن. بنابراین در این مثال cache.GetViewLocation اصلی بعد از یک درخواست مجددا فراخوانی میشه، چون GetRequestCache نه فقط طول عمر کوتاهی داره، بلکه اساسا کاری به key متد GetViewLocation نداره. کار s_key تعریف شده عموما تعریف lock هست نه استفاده ازش به عنوان کلید دیکشنری. بنابراین اگر خود MVC از HttpContext.Cache استفاده کرده، کار درستی بوده، چون به ازای هر درخواست نیازی نیست مجددا محاسبه بشه.

نویسنده: سید مهران موسوی
تاریخ: ۱۳۹۳/۰۵/۰۲ ۱۲:۲۱

ممنون از توجهتون ، بله من اشتباها httpContext.Items رو به کار برده بودم . کد موجود در مقاله اصلاح شد

نویسنده: حامد سبزیان
تاریخ: ۱۳۹۳/۰۵/۰۲ ۱۸:۴

بهبودی حاصل نشده. در DefaultViewLocationCache خود MVC مسیرها از httpContext.Cache خوانده می‌شود، در کد شما هم از همان. استفاده از httpContext.Items در کد شما ممکن است اندکی بهینه بودن را افزایش دهد، به شرط استفاده بیش از یک بار از یک (چند) View در طول یک درخواست.

Optimizing ASP.NET MVC view lookup performance

همان طور که در انتهای مقاله اشاره شده است، استفاده از یک ConcurrentDictionary می‌تواند کارایی خوبی داشته باشد اما خوب استاتیک است و به حذف و اضافه شدن فیزیکی Viewها حساس نیست.

عنوان: **سطح دوم کش در Nhibernate 4**
نویسنده: احسان زینلی
تاریخ: ۱۳۹۳/۰۷/۰۸ ۹:۴۵
آدرس: www.dotnettips.info
گروه‌ها: NHibernate, Caching, NHibernate 4

**Second Level Cache In NHibernate 4**

همان طور که می‌دانیم کش در NHibernate در دو سطح قابل انجام می‌باشد:
- کش سطح اول که همان اطلاعات سشن، در تراکنش جاری هست و با اتمام تراکنش، محتویات آن خالی می‌گردد. این سطح همیشه فعال می‌باشد و در این بخش قصد پرداختن به آن را نداریم.
- کش سطح دوم که بین همه‌ی تراکنش‌ها مشترک و پایدار می‌باشد. این مورد به طور پیش فرض فعال نمی‌باشد و می‌بایستی از طریق کانفیگ برنامه فعال گردد.

جهت پیاده سازی باید قسمت‌های ذیل را در کانفیگ مربوط به NHibernate اضافه نمود:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
<configSections>
        <section name="hibernate-configuration" type="NHibernate.Cfg.ConfigurationSectionHandler,
NHibernate"/>
         <section name="syscache" type="NHibernate.Caches.SysCache.SysCacheSectionHandler,
NHibernate.Caches.SysCache" requirePermission="false"/>
</configSections>

<hibernate-configuration  xmlns="urn:nhibernate-configuration-2.2" >
<session-factory>
       <property name="connection.provider">NHibernate.Connection.DriverConnectionProvider</property>
     <property name="dialect">NHibernate.Dialect.MsSql2005Dialect</property>
     <property name="connection.driver_class">NHibernate.Driver.SqlClientDriver</property>
     <property name="connection.connection_string_name">LocalSqlServer</property>
     <property name="show_sql">false</property>
     <property name="hbm2ddl.keywords">none</property>
     <property name="cache.use_second_level_cache">true</property>
     <property name="cache.use_query_cache" >true</property>
     <property name="cache.provider_class">NHibernate.Caches.SysCache.SysCacheProvider,
NHibernate.Caches.SysCache</property>
</session-factory>
</hibernate-configuration>

<syscache>
      <cache region="LongExpire" expiration="3600" priority="5"/>
      <cache region="ShortExpire" expiration="600" priority="3"/>
</syscache>
</configuration>
```

پیاده سازی Caching در NHibernate در سه مرحله قابل اعمال می‌باشد :
- کش در سطح Load موجودیت‌های مستقل
- کش در سطح Load موجودیت‌های وابسته Set , List , Bag , ...
- کش در سطح Query ها

Providerهای مختلفی برای اعمال و پیاده سازی آن وجود دارند که معروف‌ترین آن‌ها SysCache بوده و ما هم از همان استفاده می‌نماییم.
- مدت زمان پیش فرض کش سطح دوم، ۵ دقیقه می‌باشد و در صورت نیاز به تغییر آن، باید تگ مربوط به SysCache را تنظیم نمود. محدودیتی در تعریف تعداد متفاوتی از زمان‌های خالی شدن کش وجود ندارد و مدت زمان آن بر حسب ثانیه مشخص می‌گردد. نحوه‌ی تخصیص زمان انقضای کش به هر مورد بدین شکل صورت می‌گیرد که region مربوطه در آن معرفی می‌گردد.

جهت اعمال کش در سطح Load موجودیت‌های مستقل، علاوه بر کانفیگ اصلی، می‌بایستی کدهای زیر را به Mapping موجودیت اضافه نمود مانند :

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2" assembly="Core.Domain"
namespace="Core.Domain.Model">
  <class name="Organization" table="Core_Enterprise_Organization">
    <cache usage="nonstrict-read-write" region="ShortExpire"/>
    <id name="Id" >
      <generator/>
    </id>
    <version name="Version"/>
    <property name="Title" not-null="true" unique="true"/>
    <property name="Code" not-null="true" unique="true"/>
  </class>
</hibernate-mapping>
```

این مورد برای موجودیت‌های وابسته هم نیز صادق است؛ به شکل کد زیر:

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2" assembly="Core.Domain"
namespace="Core.Domain.Model">
  <class name="Party" table="Core_Enterprise_Party">
    <id name="Id" >
      <generator />
    </id>
    <version name="Version"/>
    <property name="Username" unique="true"/>
    <property name="DisplayName" not-null="true"/>
    <bag name="PartyGroups" inverse="true" table="Core_Enterprise_PartyGroup" cascade="all-delete-
orphan">
      <cache usage="nonstrict-read-write" region="ShortExpire"/>
      <key column="Party_id_fk"/>
      <one-to-many/>
    </bag>
  </class>
</hibernate-mapping>
```

ویژگی usage نیز با مقادیر زیر قابل تنظیم است:

- read-only : این مورد جهت موجودیت‌هایی مناسب است که امکان بروزرسانی آن‌ها توسط کاربر وجود ندارد. این مورد بهترین کارآیی را دارد.
- read-write : این مورد جهت موجودیت‌هایی بکار می‌رود که امکان بروزرسانی آن‌ها توسط کاربر وجود دارد. این مورد کارآیی پایین‌تری دارد.
- nonstrict-read-write : این مورد جهت موجودیت‌هایی مناسب می‌باشد که امکان بروزرسانی آن‌ها توسط کاربر وجود دارد؛ اما امکان همزمان بروز کردن آن‌ها توسط چندین کاربر وجود نداشته باشد. این مورد در قیاس، کارآیی بهتر و بهینه‌تری نسبت به مورد قبل دارد.

جهت اعمال کش در کوئری‌ها نیز باید مراحل خاص خودش را انجام داد. به عنوان مثال برای یک کوئری Linq به شکل زیر خواهیم داشت:

```
public IList<Organization> Search(QueryOrganizationDto dto)
{
     var q = SessionInstance.Query<Organization>();

     if (!String.IsNullOrEmpty(dto.Title))
          q = q.Where(x => x.Title.Contains(dto.Title));

     if (!String.IsNullOrEmpty(dto.Code))
          q = q.Where(x => x.Code.Contains(dto.Code));

     q = q.OrderBy(x => x.Title);

      q = q.CacheRegion("ShortExpire").Cacheable();
     return q.ToList();
}
```

در واقع کد اضافه شده به کوئری بالا، قابل کش بودن کوئری را مشخص می‌نماید و مدت زمان کش شدن آن نیز از طریق کانفیگ مربوطه مشخص می‌گردد. این نکته را هم درنظر داشته باشید که کش در سطح کوئری برای کوئری‌هایی که دقیقا مثل هم هستند اعمال می‌گردد و با افزوده یا کاسته شدن یک شرط جدید به کوئری، مجددا کوئری سمت پایگاه داده ارسال می‌گردد.

در انتها لینک‌های زیر هم جهت مطالعه بیشتر پیشنهاد می‌گردند:

http://www.nhforge.org/doc/nh/en/index.html#performance-cache-readonly

http://nhforge.org/blogs/nhibernate/archive/2009/02/09/quickly-setting-up-and-using-nhibernate-s-second-level-cache.aspx

http://www.klopfenstein.net/lorenz.aspx/using-syscache-as-secondary-cache-in-nhibernate

http://stackoverflow.com/questions/1837651/hibernate-cache-strategy

عنوان: **بازنویسی سطح دوم کش برای Entity framework 6**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۱/۰۶ ۱۶:۵۰
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, Performance, Caching

چندی قبل مطلبی را در مورد پیاده سازی سطح دوم کش در EF در این سایت مطالعه کردید . اساس آن مقاله‌ای بود که نحوه‌ی کش کردن اطلاعات حاصل از LINQ to Objects را بیان کرده بود ( ^ ). این مقاله پایه‌ی بسیاری از سیستم‌های کش مشابه نیز شده‌است ( ^ و ^ و ...).
مشکل مهم این روش عدم سازگاری کامل آن با EF است. برای مثال در آن تفاوتی بین Include(x=>x.Tags) و Include(x=>x.Users) وجود ندارد. به همین جهت در این نوع موارد، قادر به تولید کلید منحصربفردی جهت کش کردن اطلاعات یک کوئری مشخص نیست. در اینجا یک کوئری LINQ، به معادل رشته‌ای آن تبدیل می‌شود و سپس Hash آن محاسبه می‌گردد. این هش، کلید ذخیره سازی اطلاعات حاصل از کوئری، در سیستم کش خواهد بود. زمانیکه دو کوئری Include دار متفاوت EF، هش‌های یکسانی را تولید کنند، عملا این سیستم کش، کارآیی خودش را از دست می‌دهد. برای رفع این مشکل پروژه‌ی دیگری به نام EF cache ارائه شده‌است. این پروژه بسیار عالی طراحی شده و می‌تواند جهت ایده دادن به تیم EF نیز بکار رود. اما در آن فرض بر این است که شما می‌خواهید کل سیستم را در یک کش قرار دهید. وارد مکانیزم DBCommand و DataReader می‌شود و در آن‌جا کار کش کردن تمام کوئری‌ها را انجام می‌دهد؛ مگر آنکه به آن اعلام کنید از کوئری‌های خاصی صرفنظر کند.
با توجه به این مشکلات، روش بهتری برای تولید هش یک کوئری LINQ to Entities بر اساس کوئری واقعی SQL تولید شده توسط EF، پیش از ارسال آن به بانک اطلاعاتی به صورت زیر وجود دارد:

```
private static ObjectQuery TryGetObjectQuery<T>(IQueryable<T> source)
        {
            var dbQuery = source as DbQuery<T>;

            if (dbQuery != null)
            {
                const BindingFlags privateFieldFlags =
                    BindingFlags.NonPublic | BindingFlags.Instance | BindingFlags.Public;

                var internalQuery =
                    source.GetType().GetProperty("InternalQuery", privateFieldFlags)
                        .GetValue(source);

                return
                    (ObjectQuery)internalQuery.GetType().GetProperty("ObjectQuery", privateFieldFlags)
                        .GetValue(internalQuery);
            }

            return null;
        }
```

این متد یک کوئری LINQ مخصوص EF را دریافت می‌کند و با کمک Reflection، اطلاعات درونی آن که شامل ObjectQuery اصلی است را استخراج می‌کند. سپس فراخوانی متد objectQuery.ToTraceString بر روی حاصل آن، سبب تولید SQL معادل کوئری LINQ اصلی می‌گردد. همچنین objectQuery امکان دسترسی به پارامترهای تنظیم شده‌ی کوئری را نیز میسر می‌کند. به این ترتیب می‌توان به معادل رشته‌ای منطقی‌تری از یک کوئری LINQ رسید که قابلیت تشخیص JOINها و متد Include نیز به صورت خودکار در آن لحاظ شده‌است.

این اطلاعات، پایه‌ی تهیه‌ی کتابخانه‌ی جدیدی به نام EFSecondLevelCache گردید. برای نصب آن کافی است دستور ذیل را در کنسول پاورشل نیوگت صادر کنید:

```
PM> Install-Package EFSecondLevelCache
```

سپس برای کش کردن کوئری معمولی مانند:

```
var products = context.Products.Include(x => x.Tags).FirstOrDefault();
```

می‌توان از متد جدید Cacheable آن به نحو ذیل استفاده کرد (این روش بسیار تمیزتر است از روش مقاله‌ی قبلی و امکان استفاده‌ی از انواع و اقسام متدهای EF را به صورت متداولی میسر می‌کند):

```
var products = context.Products.Include(x => x.Tags).Cacheable().FirstOrDefault(); // Async methods are supported too.
```

پس از آن نیاز است کدهای کلاس Context خود را نیز به نحو ذیل ویرایش کنید:

```
namespace EFSecondLevelCache.TestDataLayer.DataLayer
{
    public class SampleContext : DbContext
    {
        // public DbSet<Product> Products { get; set; }

        public SampleContext()
            : base("connectionString1")
        {
        }

        public override int SaveChanges()
        {
            return SaveAllChanges(invalidateCacheDependencies: true);
        }

        public int SaveAllChanges(bool invalidateCacheDependencies = true)
        {
            var changedEntityNames = getChangedEntityNames();
            var result = base.SaveChanges();
            if (invalidateCacheDependencies)
            {
               new EFCacheServiceProvider().InvalidateCacheDependencies(changedEntityNames);
            }
            return result;
        }

        private string[] getChangedEntityNames()
        {
            return this.ChangeTracker.Entries()
                .Where(x => x.State == EntityState.Added ||
                            x.State == EntityState.Modified ||
                            x.State == EntityState.Deleted)
                .Select(x => ObjectContext.GetObjectType(x.Entity.GetType()).FullName)
                .Distinct()
                .ToArray();
        }
    }
}
```

متد InvalidateCacheDependencies سبب می‌شود تا اگر تغییری در بانک اطلاعاتی رخداد، به صورت خودکار کش‌های کوئری‌های مرتبط غیر معتبر شوند و برنامه اطلاعات قدیمی را از کش نخواند.

**کدهای کامل این پروژه را از مخزن کد ذیل می‌توانید دریافت کنید:**

EFSecondLevelCache

پ.ن.
این کتابخانه هم اکنون در سایت جاری در حال استفاده است.

## نظرات خوانندگان

نویسنده: اس حیدری
تاریخ: ۱۳۹۳/۱۱/۰۷ ۹:۹

برای داشتن دو یا چند Context و یا تغییر کانکشن Context می‌توان از این Cash استفاده کرد؟

چرا که کلید بر اساس معادل اسکیول عبارت Linq ایجاد می‌شود

نویسنده: ایمان دارابی
تاریخ: ۱۳۹۳/۱۱/۰۷ ۹:۴۹

این هم کتابخانه خوبی هست. البته expire شدن کش را با استفاده از تگ هندل می‌کنه. خوبیش اینه بچ دیلیت و آپدید و امکانات دیگه هم داره.
می شه از تگ به صورت اتوماتیک با روش شما ایجاد کرد و از کش همین کتابخانه استفاده کرد.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۱/۰۷ ۹:۵۵

رشته اتصالی هم در حین تولید کلید درنظر گرفته شده‌است . همچنین در صورت نیاز یک عبارت دلخواه را که به آن در اینجا saltKey گفته می‌شود، می‌توانید به رشته‌ی نهایی که از آن کلید تولید می‌شود، اضافه کنید. برای اینکار پارامتر EFCachePolicy را مقدار دهی کنید.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۱/۰۷ ۱۰:۵

در انتهای سطر دوم مطلب، به این کتابخانه اشاره شده‌است. این مشکلات را دارد:
- چون از روش LINQ to Objects برای تهیه معادل رشته‌ای کوئری درخواستی استفاده می‌کند (دقیقا این روش: ^ ) قادر نیست Includeها و جوین‌های EF را پردازش کند و در این حالت برای تمام جوین‌ها یک هش مساوی را در سیستم خواهید داشت.
- چون قادر نیست cache dependencies را از کوئری به صورت خودکار استخراج کند، شما نیاز خواهید داشت تا پارامتر تگ‌های آن‌را به صورت دستی به ازای هر کوئری تنظیم کنید. این‌کار به صورت خودکار در پروژه‌ی جاری انجام می‌شود. cache dependencies به این معنا است که کوئری جاری به چه موجودیت‌هایی در سیستم وابستگی دارد. از آن برای به روز رسانی کش استفاده می‌شود. برای مثال اگر یک کوئری به سه موجودیت وابستگی دارد، با تغییر یکی از آن‌ها، باید کش غیرمعتبر شده و در درخواست بعدی مجددا ساخته شود.

نویسنده: محمد عیدی مراد
تاریخ: ۱۳۹۳/۱۱/۰۷ ۱۰:۲۲

ظاهرا در حالت Lazy Loading زمانی که آبجکتی از کش لود میشه، پراپرتی‌های Navigation استثنای زیر را صادر میکنن:
The ObjectContext instance has been disposed and can no longer be used for operations that require a connection

تیکه کدی که این ارور رو بر میگردونه:

```
var userInRoles = user.UserInRoles.Union(user.UsersSurrogate.Where(a => a.SurrogateFromDate != null &&
a.SurrogateToDate != null && a.SurrogateFromDate <= DateTime.Now && a.SurrogateToDate >=
DateTime.Now).SelectMany(a => a.UserInRoles));
  result = userInRoles.Any(a => a.Role.FormRoles.Any(b => b.IsActive && (b.Select && b.Form.SelectPath
!= null && b.Form.SelectPath.ToLower().Split(',').Contains(roleName))));
```

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۱/۰۷ ۱۰:۴۴

Lazy loading با کش سازگاری ندارد؛ چون اتصال اشیاء موجود در کش از context قطع شده‌اند. در بار اول فراخوانی یک کوئری که قرار است کش شود، از context و دیتابیس استفاده می‌شود. اما در بارهای بعد دیگر به context و دیتابیس مراجعه نخواهد شد. تمام اطلاعات از کش سیستم بارگذاری می‌شوند و حتی یک کوئری اضافی نیز به بانک اطلاعاتی ارسال نخواهد شد. به همین جهت در این موارد باید از متد Include برای eager loading اشیایی که نیاز دارید استفاده کنید.

نویسنده: اس حیدری
تاریخ: ۱۳۹۳/۱۱/۰۷ ۱۱:۲۹

همچنین اتوماتیک بودن Cash به ازای کلیه Queryها هم می‌تواند یک آپشن در نظر گرفته شود و در مواری که دسترسی به کوئری‌های داخلی نیست مفید واقع شود.

مثلا اگر برای اعتبار سنجی کاربر از Identity استفاده شود عملا نمی‌توان به کوئری‌های داخلی Identity دسترسی پیدا کرد و نیاز است که آن کوئری‌ها Cash شود، چرا که بسیار پرکاربرد می‌باشند.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۱/۰۷ ۱۱:۳۷

- کش سطح دوم نباید برای کش کردن اطلاعات خصوصی استفاده شود؛ یا کلا اطلاعاتی که نیاز به سطح دسترسی دارند. هدف آن کش کردن اطلاعات عمومی و پر مصرف است. اطلاعات خاص یک کاربر نباید کش شوند.
- در تمام سیستم‌ها، برای مواردی که به کوئری‌های آن دسترسی ندارید تا متد Cacheable را به آن‌ها اضافه کنید، نتیجه‌ی کوئری‌ها را باید خودتان از طریق روش‌های متداول کش کنید (مانند کلاس CacheManager مطلب یاد شده).

نویسنده: امین کاشانی
تاریخ: ۱۳۹۳/۱۲/۰۸ ۱:۲۵

من در کد زیر expiretime را 60 ثانیه گذاشتم .ولی در هربار فراخوانی در بازه زمانی 60 ثانیه از کش نمی‌خواند و دیتا از دیتابیس برمی گرداند. ایراد کار کجاست؟ ولی بدون نوشتن پارامتر زمان در متد cacheable کش درست عمل می‌کند.

```
public async Task<IList<Bestankaran>> GetBestankaran()
        {
         EFCachePolicy expirationTime = new EFCachePolicy { AbsoluteExpiration = new
DateTime().AddSeconds(60) };

            var result =
                Task.Run(() =>
                    _bestankaran.Cacheable(expirationTime).ToListAsync());

            return await result;

        }
```

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۲/۰۸ ۱:۳۲

- زمانیکه از متد ToListAsync استفاده می‌کنید، نیازی به استفاده از Task.Run نیست. اطلاعات بیشتر
- بجای new DateTime باید از DateTime.Now استفاده کنید.

نویسنده: غلامرضا ربال
تاریخ: ۱۳۹۴/۰۵/۰۹ ۱۶:۴۷

با تشکر.

آیا این کتابخانه با کتابخانه EntityFramework.Extended سازگاری دارد؟

چون قصد دارم از این دو کنار هم استفاده کنم. یه کاری شبیه به کار زیر

```
public IList<string> GetUserPermissions(int[] roleIds, int userId)
        {
            var permissionsOfRoles = (from p in _permissions
                                      from r in p.ApplicationRoles
                                      where roleIds.Contains(r.Id)
                                      select p.Name).Cacheable().Future();

            var permissionsOfUser = (from p in _permissions
                                     from r in p.AssignedUsers
                                     where userId == r.Id
                                     select p.Name).Cacheable().Future().ToList();
            return permissionsOfUser.Union(permissionsOfRoles).ToList();
        }
```

ولی با خطای

```
The source query must be of type ObjectQuery or DbQuery.
Parameter name: source
```

```
[ArgumentException: The source query must be of type ObjectQuery or DbQuery.
Parameter name: source]
   EntityFramework.Extensions.FutureExtensions.Future(IQueryable`1 source) +249
```

مواجه شدم. که مشخص است برای اعمال متد Future باید مبدا از نوع IQueryable باشد.آیا اعمال متد AsQueryable در روند کار کتابخانه EFSecondLevelCache مشکلی ایجاد نخواهد شد؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۵/۰۹ ۱۷:۲

ترکیب Cacheable().Future() غیرضروری است. چون این کوئری‌های Cacheable برای بار دوم به بعد، از کش و از حافظه خوانده می‌شوند و کاری به اطلاعات Context ندارند. بار اول اتصال به دیتابیس آن هم فقط یکبار انجام می‌شود و سربار آنچنانی ندارد.

عنوان: **ذخیره تنظیمات متغیر مربوط به یک وب اپلیکیشن ASP.NET MVC با استفاده از EF**
نویسنده: غلامرضا ربال
تاریخ: ۱۳۹۴/۰۴/۲۷ ۹:۱۵
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, MVC, Reflection, ASP.NET Identity, Caching

طی این مقاله، **نحوه‌ی ذخیره سازی تنظیمات متغیر و پویای یک برنامه را به صورت Strongly Typed** ارائه خواهم داد. برای این منظور، یک API را که از Reflection ، Cache ، Lazy Loading و Entity Framework بهره میگیرد، خواهیم ساخت.
برنامه‌ی هدف ما که از این API استفاده می‌کند، یک اپلیکیشن Asp.net MVC است. قبل از شروع به ساخت API مورد نظر، یک دید کلی در مورد آنچه که قرار است در نهایت توسعه یابد، در زیر مشاهده میکنید:

```
public SettingsController(ISettings settings)
{
  // example of saving
  _settings.General.SiteName = "دات نت تیپس";
  _settings.Seo.HomeMetaTitle = ".Net Tips";
  _settings.Seo.HomeMetaKeywords = "Asp.net MVC,Entity Framework,Reflection";
  _settings.Seo.HomeMetaDescription = "ذخیره تنظیمات برنامه";
  _settings.Save();
}
```

همانطور که در کدهای بالا مشاهده میکنید، شی setting_ ما دارای دو پراپرتی فقط خواندنی بنام‌های General و Seo است که شامل تنظیمات مورد نظر ما هستند و این دو کلاس از کلاس پایه‌ی SettingBase ارث بری کرده‌اند. دو دلیل برای انجام این کار وجود دارد:
تنظیمات به صورت گروه بندی شده در کنار هم قرار گرفته‌اند و یافتن تنظیمات برای زمانی که نیاز به دسترسی به آنها داریم، راحت‌تر و ساده‌تر خواهد بود.

به این شکل تنظیمات قابل دسترس در یک گروه، از دیتابیس بازیابی خواهند شد.

### اصلا چرا باید این تنظیمات را در دیتابیس ذخیره کنیم؟

شاید فکر کنید چرا باید تنظیمات را در دیتابیس ذخیره کنیم در حالی که فایل web.config در درسترس است و می‌توان توسط کلاس ConfigurationManager به اطلاعات آن دسترسی داشت. **جواب:** دلیل این است که با تغییر فایل web.config، برنامه‌ی وب شما ری استارت خواهد شد ( چه زمان‌هایی یک برنامه Asp.net ری استارت میشود ).
برای جلوگیری از این مساله، راه حل مناسب برای ذخیره سازی اطلاعاتی که نیاز به تغییر در زمان اجرا دارند، استفاده از از دیتابیس می‌باشد. در این مقاله از Entity Framework و پایگاه داده Sql Sever استفاده می‌کنم.

مراحل ساخت Setting API مورد نظر به شرح زیر است:
ساخت یک Asp.net Web Application

ساخت مدل Setting و افزودن آن به کانتکست Entity Framework

ساخت کلاس SettingBase برای بازیابی و ذخیره سازی تنظیمات با رفلکشن

ساخت کلاس GenralSettins و SeoSettings که از کلاس SettingBase ارث بری کرده‌اند.

ساخت کلاس Settings به منظور مدیریت تمام انواع تنظیمات

یک برنامه‌ی Asp.Net Web Application را از نوع MVC ایجاد کنید. تا اینجا مرحله‌ی اول ما به پایان رسید؛ چرا که ویژوال استودیو کارهای مورد نیاز ما را انجام خواهد داد.
لازم است مدل خود را به ApplicationDbContext موجود در فایل IdentityModels.cs معرفی کنیم. به شکل زیر:

```
namespace DynamicSettingAPI.Models
{
    public interface IUnitOfWork
    {
        DbSet<Setting> Settings { get; set; }
        int SaveChanges();
    }
}

public class ApplicationDbContext : IdentityDbContext<ApplicationUser>,IUnitOfWork
    {
        public DbSet<Setting> Settings { get; set; }
        public ApplicationDbContext()
            : base("DefaultConnection", throwIfV1Schema: false)
        {
        }

        public static ApplicationDbContext Create()
        {
            return new ApplicationDbContext();
        }
    }


namespace DynamicSettingAPI.Models
{
    public class Setting
    {
        public string Name { get; set; }
        public string Type { get; set; }
        public string Value { get; set; }
    }
}
```

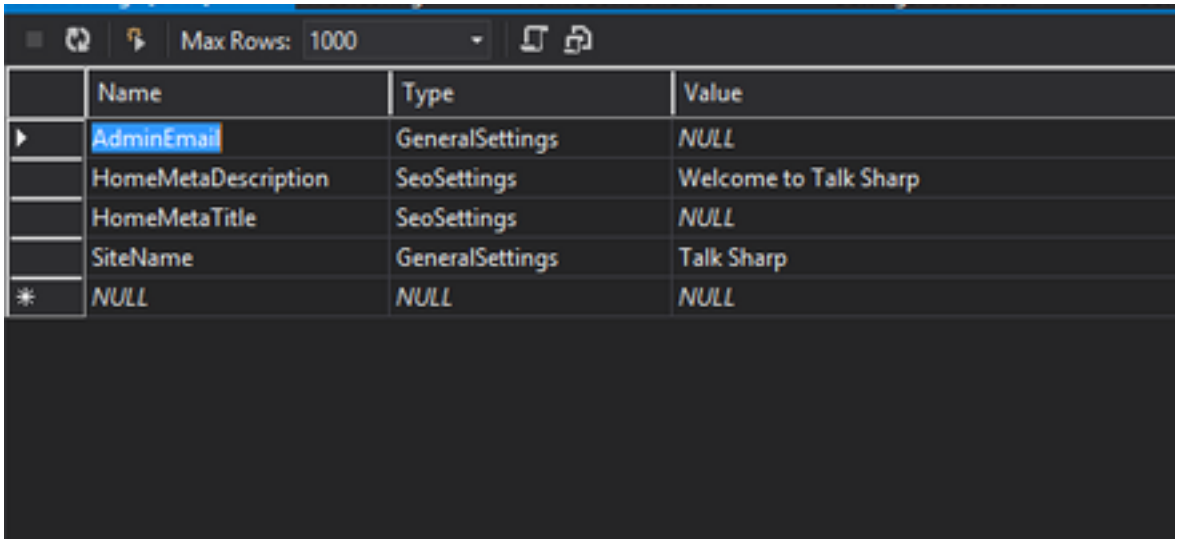
مدل تنظیمات ما خیلی ساده است و دارای سه پراپرتی به نام‌های Value ، Type ، Name هست که به ترتیب برای دریافت مقدار تنظیمات، نام کلاسی که از کلاس SettingBase ارث برده و نام تنظیمی که لازم داریم ذخیره کنیم، در نظر گرفته شده‌اند.
لازم است تا متد OnModelCreating مربوط به ApplicationDbContext را نیز تحریف کنیم تا کانفیگ مربوط به مدل خود را نیز اعمال نمائیم.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Setting>()
                    .HasKey(x => new { x.Name, x.Type });

            modelBuilder.Entity<Setting>()
                        .Property(x => x.Value)
                        .IsOptional();

            base.OnModelCreating(modelBuilder);
        }
```

ساختاری به شکل زیر مد نظر ماست:




کلاس SettingBase ما همچین ساختاری را خواهد داشت:

```
namespace DynamicSettingAPI.Service
{
    public abstract class SettingsBase
    {
        //1
        private readonly string _name;
        private readonly PropertyInfo[] _properties;

        protected SettingsBase()
        {
            //2
            var type = GetType();
            _name = type.Name;
            _properties = type.GetProperties();
        }

        public virtual void Load(IUnitOfWork unitOfWork)
        {
            //3 get setting for this type name
            var settings = unitOfWork.Settings.Where(w => w.Type == _name).ToList();

            foreach (var propertyInfo in _properties)
            {
                //get the setting from setting list
                var setting = settings.SingleOrDefault(s => s.Name == propertyInfo.Name);
                if (setting != null)
                {
                    //4 set
                    propertyInfo.SetValue(this, Convert.ChangeType(setting.Value,
propertyInfo.PropertyType));
                }
            }
        }
        public virtual void Save(IUnitOfWork unitOfWork)
        {
            //5 get all setting for this type name
            var settings = unitOfWork.Settings.Where(w => w.Type == _name).ToList();

            foreach (var propertyInfo in _properties)
            {
                var propertyValue = propertyInfo.GetValue(this, null);
                var value = (propertyValue == null) ? null : propertyValue.ToString();

                var setting = settings.SingleOrDefault(s => s.Name == propertyInfo.Name);
                if (setting != null)
                {
                    // 6 update existing value
                    setting.Value = value;
                }
                else
                {
```

```
                    // 7 create new setting
                    var newSetting = new Setting()
                    {
                        Name = propertyInfo.Name,
                        Type = _name,
                        Value = value,
                    };
                    unitOfWork.Settings.Add(newSetting);
                }
            }
        }
    }
}
```

این کلاس قرار است توسط کلاس‌های تنظیمات ما به ارث برده شود و در واقع کارهای مربوط به رفلکشن را در این کلاس کپسوله کرده‌ایم. همانطور که مشخص است ما دو فیلد را به نام‌های _name و _properties به صورت فقط خواندنی در نظر گرفته ایم که نام کلاس مورد نظر ما که از این کلاس به ارث خواهد برد، به همراه پراپرتی‌های آن، در این ظرف‌ها قرار خواهند گرفت.
متد Load وظیفه‌ی واکشی تمام تنظیمات مربوط به Type و ست کردن مقادیر به دست آمده را به خصوصیات کلاس ما، برعهده دارد. کد زیر مقدار دریافتی از دیتابیس را به نوع داده پراپرتی مورد نظر تبدیل کرده و نتیجه را به عنوان Value پراپرتی ست میکند.

```
propertyInfo.SetValue(this, Convert.ChangeType(setting.Value, propertyInfo.PropertyType));
```

متد Save نیز وظیفه‌ی ذخیره سازی مقادیر موجود در خصوصیات کلاس تنظیماتی را که از کلاس SettingBase ما به ارث برده است، به عهده دارد.
این متد دیتاهای موجود دردیتابیس را که متعلق به کلاس ارث برده مورد نظر ما هستند، واکشی میکند و در یک حلقه، اگر خصوصیتی در دیتابیس موجود بود، آن را ویرایش کرده وگرنه یک رکورد جدید را ثبت میکند.

کلاس‌های تنظیمات شخصی سازی شده خود را به شکل زیر تعریف میکنیم :

```
public class GeneralSettings : SettingsBase
    {
        public string SiteName { get; set; }
        public string AdminEmail { get; set; }
        public bool RegisterUsersEnabled { get; set; }
    }

 public class GeneralSettings : SettingsBase
    {
        public string SiteName { get; set; }
        public string AdminEmail { get; set; }
    }
```

نیازی به توضیح ندارد.
برای اینکه تنظیمات را به صورت یکجا داشته باشیم و Abstraction ای را برای استفاده از این API ارائه دهیم، یک اینترفیس و یک کلاس که اینترفیس مذکور را پیاده کرده است در نظر میگیریم:

```
public interface ISettings
{
    GeneralSettings General { get; }
    SeoSettings Seo { get; }
    void Save();
}

public class Settings : ISettings
{
    // 1
    private readonly Lazy<GeneralSettings> _generalSettings;
    // 2
    public GeneralSettings General { get { return _generalSettings.Value; } }

    private readonly Lazy<SeoSettings> _seoSettings;
    public SeoSettings Seo { get { return _seoSettings.Value; } }
```

```
    private readonly IUnitOfWork _unitOfWork;
    public Settings(IUnitOfWork unitOfWork)
    {
        _unitOfWork = unitOfWork;
        // 3
        _generalSettings = new Lazy<GeneralSettings>(CreateSettings<GeneralSettings>);
        _seoSettings = new Lazy<SeoSettings>(CreateSettings<SeoSettings>);
    }

    public void Save()
    {
        // only save changes to settings that have been loaded
        if (_generalSettings.IsValueCreated)
            _generalSettings.Value.Save(_unitOfWork);

        if (_seoSettings.IsValueCreated)
            _seoSettings.Value.Save(_unitOfWork);

        _unitOfWork.SaveChanges();
    }
    // 4
    private T CreateSettings<T>() where T : SettingsBase, new()
    {
        var settings = new T();
        settings.Load(_unitOfWork);
        return settings;
    }
}
```

این اینترفیس مشخص می‌کند که ما به چه نوع تنظیماتی، دسترسی داریم و متد Save آن برای آپدیت کردن تنظیمات، در نظر گرفته شده است. هر کلاسی که از کلاس SettingBase ارث بری کرده را به صورت فیلد فقط خواندنی و با استفاده از کلاس Lazy درون آن ذکر میکنیم و به این صورت کلاس تنظیمات ما زمانی ساخته خواهد شد که برای اولین بار به آن دسترسی داشته باشیم.
متد CreateSetting وظیفه‌ی لود دیتا را از دیتابیس، بر عهده دارد که برای این منظور، متد لود Type مورد نظر را فراخوانی میکند. این متد وقتی به کلاس تنظیمات مورد نظر برای اولین بار دسترسی پیدا کنیم، فراخوانی خواهد شد.

حتما امکان این وجود دارد که شما از امکان Caching هم بهره ببرید برای مثال همچین متد و سازنده‌ای را در کلاس Settings در نظر بگیرید:

```
private readonly ICache _cache;
public Settings(IUnitOfWork unitOfWork, ICache cache)
{
    // ARGUMENT CHECKING SKIPPED FOR BREVITY
    _unitOfWork = unitOfWork;
    _cache = cache;
    _generalSettings = new Lazy<GeneralSettings>(CreateSettingsWithCache<GeneralSettings>);
    _seoSettings = new Lazy<SeoSettings>(CreateSettingsWithCache<SeoSettings>);
}

private T CreateSettingsWithCache<T>() where T : SettingsBase, new()
{
    // this is where you would implement loading from ICache
    throw new NotImplementedException();
}
```

در آخر هم به شکل زیر میتوان (به عنوان دمو فقط ) از این API استفاده کرد.

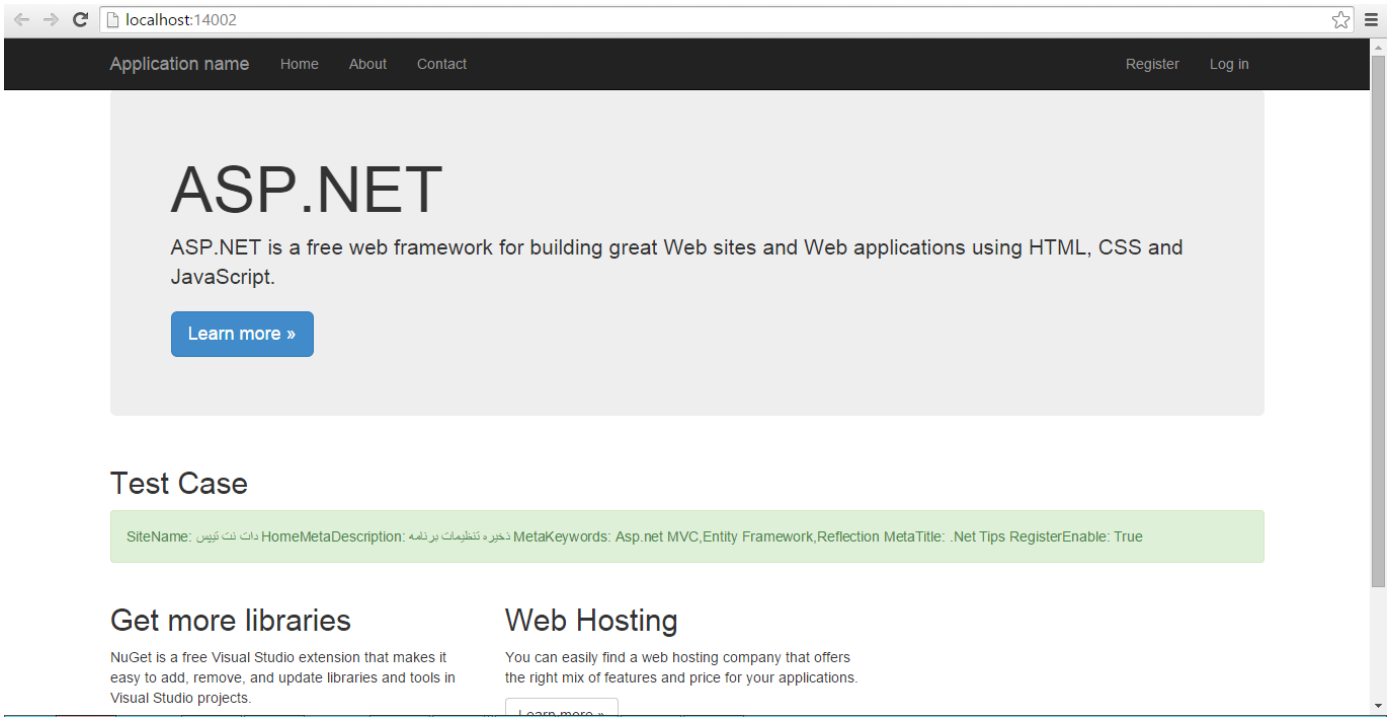
```
public ActionResult Index()
        {
            using (var uow = new ApplicationDbContext())
            {
                var _settings = new Settings(uow);
                _settings.General.SiteName = "دات نت تیپس";
                _settings.General.AdminEmail = "admin@gmail.com";
                _settings.General.RegisterUsersEnabled = true;
                _settings.Seo.HomeMetaTitle = ".Net Tips";
                _settings.Seo.MetaKeywords = "Asp.net MVC,Entity Framework,Reflection";
                _settings.Seo.HomeMetaDescription = "ذخیره تنظیمات برنامه";

                var settings2 = new Settings(uow);
```

```
                var output = string.Format("SiteName: {0} HomeMetaDescription: {1}  MetaKeywords:  {2}
MetaTitle:  {3}  RegisterEnable:  {4}",
                    settings2.General.SiteName,
                    settings2.Seo.HomeMetaDescription,
                    settings2.Seo.MetaKeywords,
                    settings2.Seo.HomeMetaTitle,
                    settings2.General.RegisterUsersEnabled.ToString()
                    );
                return Content(output);
            }

        }
```

خروجی :




**نکته:** در پروژه ای که جدیدا در سایت ارائه داده‌ام و در حال تکمیل آن هستم، از بهبود یافته‌ی این مقاله استفاده می‌شود. حتی برای اسلاید شوهای سایت هم میشود از این روش استفاده کرد و از فرمت json بهره برد برای این منظور. حتما در پروژه‌ی مذکور همچین امکانی را هم در نظر خواهم گرفتم.
پیشنها میکنم سورس SmartStore را بررسی کنید. آن هم به شکل مشابهی ولی پیشرفته‌تر از این مقاله، همچین امکانی را دارد.
DynamicSettingAPI.zip