

برای اجرای متد درون یک کلاس از طریق Reflection ابتدا نوع آن کلاس را به دست می‌آوریم و سپس از طریق کلاس Activator.CreateInstance یک نمونه از آن کلاس را ساخته و در متغیری از نوع object ذخیره کرده و با استفاده از GetMethod اطلاعات متد مورد نظر خود را در متغیری ذخیره کرده و سپس از طریق دستور Invoke آن متد را اجرا می‌کنیم. دستور Invoke سربارگذاری دارد که در یک نوع از آن، متغیر حاوی نمونه کلاس و پارامترهای متد مورد نظر، در قالب یک آرایه از نوع object، به عنوان آرگومان پذیرفته می‌شود. با امضای زیر

```
public Object Invoke(Object obj, Object[] parameters)
```

به مثال زیر که چگونگی این عملیات را شرح می‌دهد، توجه کنید:

```
public class TestMath
{
    public int Squar(int i)
    {
        return i*i;
    }
}

static void Main(string[] args)
{
    Type type = typeof (TestMath); // به دست آوردن نوع کلاس
    object obj = Activator.CreateInstance(type); // ساختن نمونه‌ای از نوع مورد نظر
    MethodInfo methodInfo = type.GetMethod("Squar"); // یافتن اطلاعات متد مورد نظر
    Console.WriteLine(methodInfo.Invoke(obj, new object[] { 100 })); // نمایش نتیجه و object[]
    // ارسال عدد 100 به صورت
    Console.Read();
}
```

توجه کنید که دو متد GetMethod و Invoke در فضای نام System.Reflection قرار دارند.

روش دیگر

در شیوه دیگر برای انجام این کار، نیازی به استفاده از GetMethod و Invoke نیست و فراخوانی متد مورد نظر بسیار شبیه فراخوانی عادی متدهاست و نیازی به ساخت متغیر ویژه‌ای از نوع object[] برای ارسال پارامترها نیست. برای انجام این کار فقط کافیست نوع متغیری که نوع نمونه‌سازی شده را نگه می‌دارد (در اینجا نمونه‌ای از کلاس را نگه می‌دارد) به صورت dynamic باشد:

```
static void Main(string[] args)
{
    Type type = typeof (TestMath);
    dynamic obj = Activator.CreateInstance(type);
    Console.WriteLine(obj.Square(100));
    Console.Read();
}
```

توجه کنید که بعد از تعریف obj، با درج نقطه در کنار آن، منوی Code Insight متد Square را شامل نمی‌شود اما کامپایلر آن را می‌پذیرد.

نظرات خوانندگان

نویسنده: KishIsland

تاریخ: ۱۹:۴۷ ۱۳۹۱/۱۲/۱۵

برای تست کد بالا من یک کلاس بشکل زیر تعریف کردم:

```
class Test
{
    public void func1()
    {
        Console.WriteLine("Hello World!");
    }
}
```

و در قسمت main برنامه فراخوانی رو بشکل زیر نوشتم مطابق روش دوم ولی برنامه Exception داد:

```
Type type = typeof(Test);
dynamic obj = Activator.CreateInstance(type);
Console.WriteLine(obj.func1());
```

علت بروز استثناء void بودن متد هست. می خواستم بدونم برای مواقعی که متد بصورت void تعریف شده چکار باید کرد؟
سپاس

نویسنده: وحید نصیری

تاریخ: ۲۱:۲۰ ۱۳۹۱/۱۲/۱۵

در این حالت بجای

```
Console.WriteLine(obj.func1());
```

فقط کافی هست بنویسید

```
obj.func1();
```

یک از ابتدایی‌ترین مواردی که در یادگیری دات نت آموزش داده می‌شود مباحث مربوط به کپسوله سازی است. برای مثال فیلدها و خواص Private که به صورت خصوصی هستند یا Protected هستند از خارج کلاس قابل دسترسی نیستند. برای دسترسی به این کلاس‌ها باید از خواص یا متدهای عمومی استفاده کرد.

```
public class Book
{
    private int code = 10;

    public int GetCode()
    {
        return code;
    }
}
```

یا فیلدها و خواصی که به صورت فقط خواندنی هستند، (ReadOnly) امکان تغییر مقدار برای اون‌ها وجود ندارد. برای مثال کد پایین کامپایل نخواهد شد.

```
public class Book
{
    private readonly int code = 10;

    public int GetCode()
    {
        return code = 20;
    }
}
```

اما در دات نت با استفاده از Reflection می‌تونیم تمام قوانین بالا رو نادیده بگیریم. یعنی می‌تونیم هم به خواص و فیلدهای غیر عمومی کلاس دسترسی پیدا کنیم و هم می‌تونیم مقدار فیلدهای فقط خواندنی رو تغییر بدیم. به مثال‌های زیر دقت کنید.

#مثال اول

```
using System.Reflection;

public class Book
{
    private int code = 10;
}

public class Program
{
    static void Main( string[] args )
    {
        Book book = new Book();
        var codeField = book.GetType().GetField( "code", BindingFlags.NonPublic |
BindingFlags.Instance );
        codeField.SetValue( book, 20 );
        var value = codeField.GetValue( book );
    }
}
```

ابتدا یک کلاس که دارای یک متغیر به نام کد است ساخته ایم که مقدار 10 را دارد. فیلد به صورت private است. بعد از اجرا به راحتی مقدار Code را به دست می‌آوریم.

```
class Program
{
    static void Main( string[] args )
    {
        Book book = new Book();
        var codeField = book.GetType().GetField( "code", BindingFlags.NonPublic | BindingFlags.Instance );
        var value = codeField.GetValue( book );
    }
}
```



حتی امکان تغییر مقدار فیلد private هم امکان پذیر است.

```
class Program
{
    static void Main( string[] args )
    {
        Book book = new Book();
        var codeField = book.GetType().GetField( "code", BindingFlags.NonPublic | BindingFlags.Instance );
        codeField.SetValue( book, 100 );
        var value = codeField.GetValue( book );
    }
}
```



#مثال دوم.

در این مثال قصد داریم مقدار یک فیلد، از نوع فقط خواندنی رو تغییر دهیم.

```
using System.Reflection;

public class Book
{
    private readonly int code = 10;
}

public class Program
{
    static void Main( string[] args )
    {
        Book book = new Book();
        var codeField = book.GetType().GetField( "code", BindingFlags.NonPublic |
BindingFlags.Instance );
        codeField.SetValue( book, 50);
        var value = codeField.GetValue( book );
    }
}
```

بعد از اجرا مقدار متغیر code به 50 تغییر می‌یابد.

```
Book book = new Book();  
var codeField = book.GetType().GetField( "code" );  
codeField.SetValue( book, 50 );  
var value = codeField.GetValue( book );
```



[مطالب تکمیلی](#)

نظرات خوانندگان

نویسنده: وحید نصیری
تاریخ: ۱۰:۱۰ ۱۳۹۲/۰۳/۱۷

البته مباحث Reflection، تابع سطح دسترسی کد فراخوان است (همان لینک آخر بحث جهت تاکید بیشتر و همچنین تنویر مقدمه):

« [Security Considerations for Reflection](#) »

برای نمونه در حالت medium trust، گزینه ReflectionPermission غیرفعال است. برای آزمایش این مسایل می‌شود از دو برنامه [Permview](#) و [Permcalc](#) استفاده کرد.

نویسنده: سالار خلیل زاده
تاریخ: ۹:۱۸ ۱۳۹۲/۰۳/۱۸

ReflectionMagic جهت همین کار طراحی شده
<http://nuget.org/packages/ReflectionMagic>

نویسنده: reza
تاریخ: ۱۷:۹ ۱۳۹۳/۰۵/۲۵

آیا می‌توان به کمک رفلکشن به خصوصیتی که مثلاً Set ندارد مقدار دهی کرد. بعنوان مثال

```
class Program
{
    static void Main(string[] args)
    {
        Book book = new Book();
        var codeprop = book.GetType().GetProperty("Code", BindingFlags.Public | BindingFlags.Instance);
        codeprop.SetValue(book, 20, null);
        var value = codeprop.GetValue(book, null);
    }
}

public class Book
{
    public int code;
    public int Code
    {
        get { return code; }
    }
}
```

نویسنده: مسعود پاکدل
تاریخ: ۱۴:۰ ۱۳۹۳/۰۵/۲۶

خیر! با یک ArgumentException و پیغام Property set method not found مواجه خواهید شد. اما در مثال بالا می‌توان مقدار فیلد code را تغییر داد که در نتیجه خاصیت Code نیز مقدار جدید را برگشت می‌دهد.

در بسیاری از پروژه‌های دات نت، نیاز به استفاده از فایل‌های نرم افزار آفیس، از قبیل ورد و اکسل و ... وجود دارد. برای مثال گاهی لازم است اطلاعات یک گرید، یا هر منبع داده‌ای، در قالب اکسل به کاربر نمایش داده شود. بدین شکل که این فایلها در زمان اجرا ساخته شده و به کاربر نمایش داده شود. حال فرض کنید شما روی سیستم خودتان Office2007 را نصب کرده اید و به اسمبلی‌های این ورژن دسترسی دارید. البته بدون نیاز به نصب آفیس نیز میتوان به این توابع دسترسی داشت و از آنها در برنامه استفاده کرد که همان استفاده از [Primary Interop Assemblies](#) میباشد. مشکلی که ممکن است پیش آید این است که در کامپیوترهای کاربران ممکن است ورژن‌های مختلفی از آفیس نصب باشد مانند 2003-2007-2010-2013 و اگر با ورژن اسمبلی‌هایی که فراخوانی‌های فایل‌های اکسل از طریق آن انجام شده باشد متفاوت باشد، برنامه اجرا نمی‌شود.

در حالت معمول برای نمایش یک فایل آفیس مثل اکسل در برنامه، ابتدا اسمبلی مربوطه را (اکسل در این مثال) که به نام Microsoft.Office.Interop.Excel میباشد به اسمبلی‌های برنامه اضافه کرده (از طریق add reference) و برای نمایش یک فایل اکسل در زمان اجرا از کدهای زیر استفاده مینماییم:

```
try
{
    var application =
    (Microsoft.Office.Interop.Excel.Application)Activator.CreateInstance(Type.GetTypeFromProgID("Excel.Application"));
    Workbook wrkBook;
    var wbk = application.Workbooks;
    wrkBook = wbk.Add();
    wrkBook.Activate();
    application.Visible = true;
}
catch (Exception ex)
{
    Error(ex.Message);
}
```

حال اگر آفیس 2010 به عنوان مثال در سیستم ما نصب باشد، ورژن این اسمبلی 14 می‌باشد و اگر این برنامه را در کامپیوتر کلاینتی که آفیس 2007 بر روی آن نصب باشد انتشار دهیم اجرا نمیشود. برای حل این مشکل بنده با استفاده از روش [dynamic](#) این موضوع را حل کردم و بنظر می‌رسد راه‌های دیگری نیز برای حل آن وجود داشته باشد.

در این روش با توجه به ورژن آفیزی که بر روی سیستم کاربر نصب شده اسمبلی مربوطه را از سیستم کاربر لود کرده و فایل‌های آفیس را اجرا مینماییم. در ابتدا تشخیص میدهیم چه ورژنی از آفیس بر روی سیستم کاربر نصب است:

```
string strVersion = null;
dynamic objEApp = Activator.CreateInstance(Type.GetTypeFromProgID("Excel.Application"));
if (objEApp.Version == "12.0")
{
    strVersion = "2007";
}
else if (objEApp.Version == "14.0")
{
    strVersion = "2010";
}
```

روش دیگر برای انجام اینکار استفاده از اطلاعات رجیستری ویندوز است:

```
string strEVersionSubKey = "\\Excel.Application\\CurVer";
string strValue = null; //Value Present In Above Key
string strVersion = null; //Determines Excel Version
RegistryKey rkVersion = null; //Registry Key To Determine Excel Version
rkVersion = Registry.ClassesRoot.OpenSubKey(strEVersionSubKey, false); //Open Registry Key
```

```

if ((rkVersion != null)) //If Key Exists
{
    strValue = (string)rkVersion.GetValue(string.Empty); //Get Value
    strValue = strValue.Substring(strValue.LastIndexOf(".") + 1); //Store Value
    switch (strValue) //Determine Version
    {
        case "11":
            strVersion = "2003";
            break;

        case "12":
            strVersion = "2007";
            break;

        case "14":
            strVersion = "2010";
            break;
    }
}

```

حال با استفاده از تابع `assembly.load()` اسمبلی مورد نیاز را لود کرده و در برنامه استفاده مینماییم :

```

if (strVersion == "2007")
{
    string strAssemblyOff2007 =
        "Microsoft.Office.Interop.Excel, Version=12.0.0.0, Culture=neutral,
        PublicKeyToken=71e9bce111e9429c";
    try
    {
        Assembly xslExcelAssembly = Assembly.Load(strAssemblyOff2007); //Load Assembly
        Type type = xslExcelAssembly.GetType().Single(t => t.Name ==
        "ApplicationClass");
        dynamic application = Activator.CreateInstance(type);
        var workbooks = application.Workbooks;
        var workbook = workbooks.Add();
        var worksheet = workbook.Worksheets[1];
        workbook.Activate();
        application.Visible = true;
    }
    catch (Exception ex)
    {
    }
}

```

در این حالت بدون اینکه بدانیم بر روی سیستم کاربر چه ورژنی از آفیس نصب است میتوان فایل‌های آفیس را در زمان اجرا لود کرده و استفاده کرد .

نظرات خوانندگان

نویسنده: حسین

تاریخ: ۲۰:۳۳ ۱۳۹۲/۱۱/۱۹

با تشکر از مقاله مفید شما
من یه بار توی یه پروژه یک تمپلت ورد ایجاد کردم و توش انواع اقسام چارت‌ها و جدول‌ها رو توش رسم کردم و کلی هم روش
کار کردم تا گزارش خوبی از کار در بیارد
واقعا اطلاع نداشتم با ورژن‌ها مختلف اجرا نمیشه!
الان عذاب وجدان گرفتم :)

مدل زیر را در نظر بگیرید:

```
/// <summary>
///
/// </summary>
public class CompanyModel
{
    /// <summary>
    /// Table Identity
    /// </summary>
    public int Id { get; set; }

    /// <summary>
    /// Company Name
    /// </summary>
    [DisplayName("نام شرکت")]
    public string CompanyName { get; set; }

    /// <summary>
    /// Company Abbreviation
    /// </summary>
    [DisplayName("نام اختصاری شرکت")]
    public string CompanyAbbr { get; set; }
}
```

از View زیر جهت نمایش لیستی از شرکت‌ها متناظر با مدل جاری استفاده میشود:

```
@{
    const string viewTitle = "شرکت ها";
    ViewBag.Title = viewTitle;
    const string gridName = "companies-grid";
}
<div class="col-md-12">
    <div class="form-panel">
        <header>
            <div class="title">
                <i class="fa fa-book"></i>
                @viewTitle
            </div>
        </header>
        <div class="panel-body">
            <div id="@gridName">
                </div>
            </div>
        </div>
    </div>
</div>
</div>
@section scripts
{
    <script type="text/javascript">
        $(document).ready(function () {
            $("#@gridName").kendoGrid({
                dataSource: {
                    type: "json",
                    transport: {
                        read: {
                            url: "@Html.Raw(Url.Action(MVC.Company.CompanyList()))",
                            type: "POST",
                            dataType: "json",
                            contentType: "application/json"
                        }
                    },
                    schema: {
                        data: "Data",
                        total: "Total",
                        errors: "Errors"
                    }
                },
                pageSize: 10,
```

```

        serverPaging: true,
        serverFiltering: true,
        serverSorting: true
    },
    pageable: {
        refresh: true
    },
    sortable: {
        mode: "multiple",
        allowUnsort: true
    },
    editable: false,
    filterable: false,
    scrollable: false,
    columns: [ {
        field: "CompanyName",
        title: "نام شرکت",
        sortable: true,
    }, {
        field: "CompanyAbbr",
        title: "مخفف نام شرکت",
        sortable: true
    } ]
    });
});
</script>
}

```

مشکلی که در کد بالا وجود دارد این است که با تغییر نام هر یک از متغیر هایمان ، اطلاعات گرید در ستون مربوطه نمایش داده نمیشود. همچنین عناوین ستونها نیز از DisplayName مدل پیروی نمیکنند. توسط متدهای الحاقی زیر این مشکل برطرف شده است.

```

/// <summary>
///
/// </summary>
public static class PropertyExtensions
{
    /// <summary>
    ///
    /// </summary>
    /// <typeparam name="T"></typeparam>
    /// <param name="expression"></param>
    /// <returns></returns>
    public static MemberInfo GetMember<T>(this Expression<Func<T, object>> expression)
    {
        var mbody = expression.Body as MemberExpression;

        if (mbody != null) return mbody.Member;
        //This will handle Nullable<T> properties.
        var ubody = expression.Body as UnaryExpression;
        if (ubody != null)
        {
            mbody = ubody.Operand as MemberExpression;
        }
        if (mbody == null)
        {
            throw new ArgumentException("Expression is not a MemberExpression", "expression");
        }
        return mbody.Member;
    }

    /// <summary>
    ///
    /// </summary>
    /// <typeparam name="T"></typeparam>
    /// <param name="expression"></param>
    /// <returns></returns>
    public static string PropertyName<T>(this Expression<Func<T, object>> expression)
    {
        return GetMember(expression).Name;
    }

    /// <summary>
    ///
    /// </summary>
}

```

```

    /// <typeparam name="T"></typeparam>
    /// <param name="expression"></param>
    /// <returns></returns>
    public static string PropertyDisplay<T>(this Expression<Func<T, object>> expression)
    {
        var propertyMember = GetMember(expression);
        var displayAttributes = propertyMember.GetCustomAttributes(typeof(DisplayNameAttribute),
true);
        return displayAttributes.Length == 1 ?
((DisplayNameAttribute)displayAttributes[0]).DisplayName : propertyMember.Name;
    }
}

```

```
public static string PropertyName<T>(this Expression<Func<T, object>> expression)
```

جهت بدست آوردن نام متغیر هایمان استفاده مینماییم.

```
public static string PropertyDisplay<T>(this Expression<Func<T, object>> expression)
```

جهت بدست آوردن DisplayNameAttribute استفاده میشود. در صورتیکه این DisplayNameAttribute یافت نشود نام متغیر بازگشت داده میشود.

بنابراین View مربوطه را اینگونه بازنویسی میکنیم:

```

@using Models
@{
    const string viewTitle = "شرکت ها";
    ViewBag.Title = viewTitle;
    const string gridName = "companies-grid";
}
<div class="col-md-12">
    <div class="form-panel">
        <header>
            <div class="title">
                <i class="fa fa-book"></i>
                @viewTitle
            </div>
        </header>
        <div class="panel-body">
            <div id="@gridName">
            </div>
        </div>
    </div>
</div>
</div>
@section scripts
{
    <script type="text/javascript">
        $(document).ready(function () {
            $("#@gridName").kendoGrid({
                dataSource: {
                    type: "json",
                    transport: {
                        read: {
                            url: "@Html.Raw(Url.Action(MVC.Company.CompanyList()))",
                            type: "POST",
                            dataType: "json",
                            contentType: "application/json"
                        }
                    },
                    schema: {
                        data: "Data",
                        total: "Total",
                        errors: "Errors"
                    }
                }
            });
        });
    </script>
}

```

```
        },
        pageSize: 10,
        serverPaging: true,
        serverFiltering: true,
        serverSorting: true
    },
    pageable: {
        refresh: true
    },
    sortable: {
        mode: "multiple",
        allowUnsort: true
    },
    editable: false,
    filterable: false,
    scrollable: false,
    columns: [ {
        field: "@(PropertyExtensions.PropertyName<CompanyModel>(a => a.CompanyName))",
        title: "@(PropertyExtensions.PropertyDisplay<CompanyModel>(a => a.CompanyName))",
        sortable: true,
    }, {
        field: "@(PropertyExtensions.PropertyName<CompanyModel>(a => a.CompanyAbbr))",
        title: "@(PropertyExtensions.PropertyDisplay<CompanyModel>(a => a.CompanyAbbr))",
        sortable: true
    } ]
    });
</script>
}
```

نظرات خوانندگان

نویسنده:

وحید نصیری

تاریخ:

۱۳۹۳/۰۴/۱۶ ۱۲:۳۸

با تشکر از شما. حالت پیشرفته‌تر این مساله، کار با مدل‌های تو در تو هست. برای مثال:

```
public class CompanyModel
{
    public int Id { get; set; }
    public string CompanyName { get; set; }
    public string CompanyAbbr { get; set; }

    public Product Product { set; get; }
}

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

در اینجا اگر بخواهیم Product.Id را بررسی کنیم:

```
var data = PropertyExtensions.PropertyName<CompanyModel>(x => x.Product.Id);
```

فقط Id آن دریافت می‌شود.

راه حلی که از کدهای EF برای این مساله استخراج شده به صورت زیر است (نمونه‌اش متد Include تو در تو بر روی چند خاصیت):

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Linq.Expressions;

namespace PropertyExtensionsApp
{
    public class PropertyHelper : ExpressionVisitor
    {
        private Stack<string> _stack;
        public string GetNestedPropertyPath(Expression expression)
        {
            _stack = new Stack<string>();
            Visit(expression);
            return _stack.Aggregate((s1, s2) => s1 + "." + s2);
        }

        protected override Expression VisitMember(MemberExpression expression)
        {
            if (_stack != null)
                _stack.Push(expression.Member.Name);
            return base.VisitMember(expression);
        }

        public string GetNestedPropertyName<TEntity>(Expression<Func<TEntity, object>> expression)
        {
            return GetNestedPropertyPath(expression);
        }
    }
}
```

در این حالت خواهیم داشت:

```
var name = new PropertyHelper().GetNestedPropertyName<CompanyModel>(x => x.Product.Id);
```

که خروجی Product.Id را بر می‌گرداند.

نویسنده: محسن موسوی
تاریخ: ۱۸:۸ ۱۳۹۳/۰۷/۲۶

در نهایت این متد به این شکل اصلاح شود:

```
/// <summary>
///
/// </summary>
/// <typeparam name="T"></typeparam>
/// <param name="expression"></param>
/// <returns></returns>
public static string PropertyName<T>(this Expression<Func<T, object>> expression)
{
    return new PropertyHelper().GetNestedPropertyName(expression);
}
```

نویسنده: وحید نصیری
تاریخ: ۲۳:۵۷ ۱۳۹۳/۰۹/۱۰

روش دیگری در اینجا: « [Strongly-Typed ID References to Razor-Generated Fields](#) »

نویسنده: محسن موسوی
تاریخ: ۱۴:۱ ۱۳۹۳/۰۹/۱۱

با تشکر

- نظر نویسنده مقاله تغییر کرده، بدلیل دوباره کاری انجام شده.(توضیحات بیشتر در کامنتهای مقاله ارجاعی)
- روش جاری وابسته به مدل ویو نیست و به همین دلیل محدودیت نداره، بطور مثال یک ویو شامل عملیاتهای اضافه و ویرایش و حذف و گرید لیست آنهاست.

عنوان: کدام سلسله متدها، متد جاری را فراخوانی کرده‌اند؟

نویسنده: وحید نصیری

تاریخ: ۹:۴۰ ۱۳۹۳/۰۷/۰۶

آدرس: www.dotnettips.info

گروه‌ها: Entity framework, Reflection, Profiler, StackTrace

یکی از نیازهای نوشتن یک برنامه‌ی پروفایلر، نمایش اطلاعات متدهایی است که سبب لاگ شدن اطلاعاتی شده‌اند. برای مثال [در](#) طراحی [interceptor](#) های EF 6 به یک چنین متدهایی می‌رسیم:

```
public void ScalarExecuted(DbCommand command,
                           DbCommandInterceptionContext<object> interceptionContext)
{
}
```

سؤال: در زمان اجرای `ScalarExecuted` دقیقا در کجا قرار داریم؟ چه متدی در برنامه، در کدام کلاس، سبب رسیدن به این نقطه شده‌است؟
تمام این اطلاعات را در زمان اجرا توسط کلاس `StackTrace` می‌توان بدست آورد:

```
public static string GetCallingMethodInfo()
{
    var stackTrace = new StackTrace(true);
    var frameCount = stackTrace.FrameCount;

    var info = new StringBuilder();
    var prefix = "-- ";
    for (var i = frameCount - 1; i >= 0; i--)
    {
        var frame = stackTrace.GetFrame(i);
        var methodInfo = getStackFrameInfo(frame);
        if (string.IsNullOrEmpty(methodInfo))
            continue;

        info.AppendLine(prefix + methodInfo);
        prefix = "- " + prefix;
    }

    return info.ToString();
}
```

ایجاد یک نمونه جدید از کلاس `StackTrace` با پارامتر `true` به این معنا است که می‌خواهیم اطلاعات فایل‌های متناظر را نیز در صورت وجود دریافت کنیم.

خاصیت `stackTrace.FrameCount` مشخص می‌کند که در زمان فراخوانی متد `GetCallingMethodInfo` که اکنون برای مثال درون متد `ScalarExecuted` قرار گرفته‌است، از چند سطح بالاتر این فراخوانی صورت گرفته‌است. سپس با استفاده از متد `stackTrace.GetFrame` می‌توان به اطلاعات هر سطح دسترسی یافت.
در هر `StackFrame` دریافتی، با فراخوانی `stackFrame.GetMethod` می‌توان نام متد فراخوان را بدست آورد. متد `stackFrame.GetFileName` دقیقا شماره سطر را که فراخوانی از آن صورت گرفته، بازگشت می‌دهد و `stackFrame.GetFileName` نیز نام فایل مرتبط را مشخص می‌کند.

یک نکته:

شرط عمل کردن متدهای `stackFrame.GetFileName` و `stackFrame.GetFileLineNumber` در زمان اجرا، وجود فایل PDB اسمبلی در حال بررسی است. بدون آن اطلاعات محل قرارگیری فایل سورس مرتبط و شماره سطر فراخوان، قابل دریافت نخواهند بود.

اکنون بر اساس این اطلاعات، متد `getStackFrameInfo` چنین پیاده سازی را خواهد داشت:

```
private static string getStackFrameInfo(StackFrame stackFrame)
{
    if (stackFrame == null)
        return string.Empty;

    var method = stackFrame.GetMethod();
```



```

    if (method == null)
        return string.Empty;

    if (isFromCurrentAsm(method) || isMicrosoftType(method))
    {
        return string.Empty;
    }

    var methodSignature = method.ToString();
    var lineNumber = stackFrame.GetFileLineNumber();
    var filePath = stackFrame.GetFileName();

    var fileLine = string.Empty;
    if (!string.IsNullOrEmpty(filePath))
    {
        var fileName = Path.GetFileName(filePath);
        fileLine = string.Format("[File={0}, Line={1}]", fileName, lineNumber);
    }

    var methodSignatureFull = string.Format("{0} {1}", methodSignature, fileLine);
    return methodSignatureFull;
}

```

و خروجی آن برای مثال چنین شکلی را خواهد داشت:

```
Void Main(System.String[]) [File=Program.cs, Line=28]
```

که وجود file و line آن تنها به دلیل وجود فایل PDB اسمبلی مورد بررسی است.

در اینجا خروجی نهایی متد GetCallingMethodInfo به شکل زیر است که در آن چند سطح فراخوانی را می‌توان مشاهده کرد:

```

-- Void Main(System.String[]) [File=Program.cs, Line=28]
--- Void disposedContext() [File=Program.cs, Line=76]
---- Void Opened(System.Data.Common.DbConnection,
System.Data.Entity.Infrastructure.Interception.DbConnectionInterceptionContext)
[File=DatabaseInterceptor.cs,Line=157]

```

جهت تعدیل خروجی متد GetCallingMethodInfo، عموماً نیاز است مثلاً از کلاس یا اسمبلی جاری صرف‌نظر کرد یا اسمبلی‌های میکروسافت نیز در این بین شاید اهمیتی نداشته باشند و بیشتر هدف بررسی سورس‌های موجود است تا فراخوانی‌های داخلی یک اسمبلی ثالث:

```

private static bool isFromCurrentAsm(MethodBase method)
{
    return method.ReflectedType == typeof(CallingMethod);
}

private static bool isMicrosoftType(MethodBase method)
{
    if (method.ReflectedType == null)
        return false;

    return method.ReflectedType.FullName.StartsWith("System.") ||
           method.ReflectedType.FullName.StartsWith("Microsoft.");
}

```

کد کامل CallingMethod.cs را از اینجا می‌توانید دریافت کنید:

[CallingMethod.cs](#)

نظرات خوانندگان

نویسنده: علیرضا
تاریخ: ۱۳۹۳/۰۷/۱۰ ۲۳:۳۸

چه موقعی GetMethod میتواند Null برگرداند؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۷/۱۱ ۰:۳۷

زمانیکه کامپایلر مباحث inlining متدها را جهت بهینه سازی اعمال کند.

نویسنده: علی یگانه مقدم
تاریخ: ۱۳۹۴/۰۴/۱۰ ۸:۵۳

بیان این نکته خالی از لطف نیست که در دات نت ۴.۵ به بعد یک سری attribute هم برای راحتی کار ارائه شده است. یک نمونه آن callermemberinfo است که در این [مقاله](#) یکی از استفاده‌های کاربردی آن را می‌بینید

نویسنده: محسن خان
تاریخ: ۱۳۹۴/۰۴/۱۰ ۹:۴۲

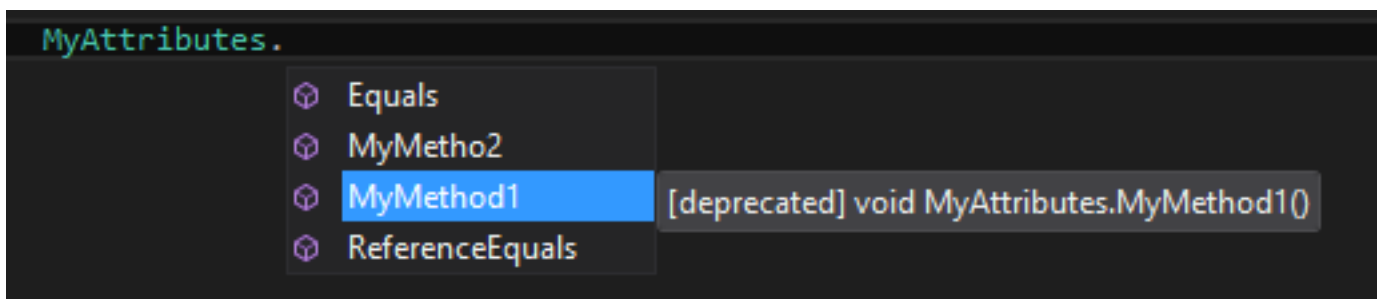
callermemberinfo فقط یک سطح را بازگشت می‌دهد و همچنین امضای متدها را هم باید تغییر داد. زمانیکه قرار هست پروفایلی را تهیه کنید، این پروفایلر نباید سبب تغییر کدهای اصلی برنامه شود.

در قسمت های مختلفی از منابع آموزشی این سایت از متادیتاها attributes استفاده شده و در برخی آموزش هایی چون [EF](#) و [MVC](#) حداقل یک قسمت کامل را به خود اختصاص داده اند. متادیتاها کلاس هایی هستند که به روشی سریع و کوتاه در بالای یک Type معرفی شده و ویژگی هایی را به آن اضافه می کنند. به عنوان مثال متادیتای زیر را ببینید. این متادیتا در بالای یک متد در یک کلاس تعریف شده است و این متد را منسوخ شده اعلام می کند و به برنامه نویس می گوید که در نسخه ی جاری کتابخانه، این متد که احتمال میرود در نسخه های پیشین کاربرد داشته است، الان کارآیی خوبی برای استفاده نداشته و بهتر است طبق مستندات آن کلاس، از یک متد جایگزین که برای آن فراهم شده است استفاده کند.

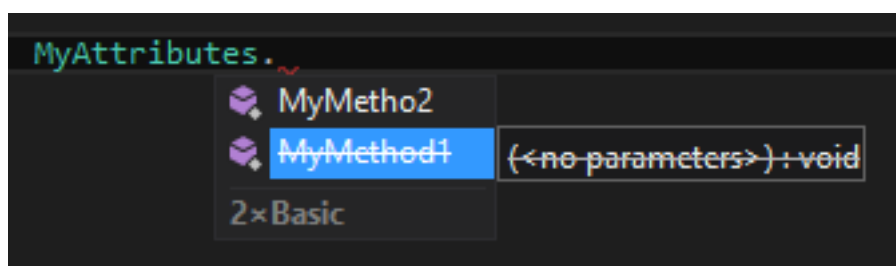
```
public static class MyAttributes
{
    [Obsolete]
    public static void MyMethod1()
    {
    }

    public static void MyMetho2()
    {
    }
}
```

همانطور که ملاحظه می کنید می توانید اخطار آن را مشاهده کنید:



البته توصیه می کنم از ابزارهایی چون [Resharper](#) در کارهایتان استفاده کنید، تا طعم کدنویسی را بهتر بچشید. نحوه ی نمایش آن در Resharper به مراتب واضح تر و گویاتر است:



حال در این بین این سؤال پیش می‌آید که چگونه ما هم می‌توانیم متادیتاهایی را با سلیقه‌ی خود ایجاد کنیم. برای تهیه‌ی یک متادیتا از کلاس `system.attribute` استفاده می‌کنیم:

```
public class MyMaxLength:Attribute
{
}
```

در چنین حالتی شما یک متادیتا ساخته‌اید که می‌توان از آن به شکل زیر استفاده کرد:

```
[MyMaxLength]
public class GetCustomProperties
{
//...
```

ولی اگر بخواهید توسط این متادیتا اطلاعاتی را دریافت کنید، می‌توانید به روش زیر عمل کنید. در اینجا من دوست دارم یک متادیتا به اسم `MyMaxLength` را ایجاد کرده تا جایگزین `MaxLength` دات نت کنم، تا طبق میل من رفتار کند.

```
public class MyMaxLength:Attribute
{
    private int max;
    public string ErrorText = "";

    public MyMaxLength(int max)
    {
        this.max = max;
        ErrorText = string.Format("max Length is {0} chars", max);
    }
}
```

در کد بالا، یک متادیتا با یک پارامتر اجباری در سازنده تعریف شده است. این کلاس هم می‌تواند مثل سایر کلاس‌ها سازنده‌های مختلفی داشته باشد تا چندین شکل تعریف متادیتا داشته باشیم. متغیر `ErrorText` به عنوان یک پارامتر معرفی نشده، ولی از آن جا که `public` تعریف شده است می‌تواند مورد استفاده‌ی مستقیم قرار بگیرد و استفاده‌ی از آن نیز اختیاری است. نحوه‌ی معرفی این متادیتا نیز به صورت زیر است:

```
[MyMaxLength(30)]
public class GetCustomProperties
{
//...
}

//or
[MyMaxLength(30,ErrorText = "شما اجازه ندارید بیش از 30 کاراکتر وارد نمایید")]
public class GetCustomProperties
{
//...
}
```

در حالت اول از آنجا که متغیر `ErrorText` اختیاری است، تعریف نشده است. پس در نتیجه با مقدار `Max length is (x=max) chars` پر خواهد شد ولی در حالت دوم برنامه نویسی متن خطا را به خود کلاس واگذار نکرده است و آن را طبق میل خود تغییر داده است.

اجباری کردن Type

هر متادیتا می‌تواند مختص یک نوع `Type` باشد که این نوع می‌تواند یک کلاس، متد، پراپرتی یا ساختار و ... باشد. نحوه‌ی محدود سازی آن توسط یک متادیتا مشخص می‌شود:

```
[System.AttributeUsage(System.AttributeTargets.Class | System.AttributeTargets.Struct)]
public class MyMaxLength:Attribute
{
    private int max;
    public string ErrorText = "";

    public MyMaxLength(int max)
    {
        this.max = max;
        ErrorText = string.Format("max Length is {0} chars", max);
    }
}
```

الان این کلاس توسط متادیتای AttributeUsage که پارامتر ورودی آن Enum است محدود به دو ساختار کلاس و Struct شده است. البته در ویژوال بیسیک با نام Structure معرفی شده است. اگر ساختار شمارشی AttributeTarget را مشاهده کنید، لیستی از نوعها را چون All (همه موارد) ، دلگیت، سازنده، متد و ... را مشاهده خواهید کرد و از آن جا که این متادیتای ما کاربردش در پراپرتیها خلاصه می شود، از متادیتای زیر بر روی آن استفاده می کنیم:

```
[AttributeUsage(AttributeTargets.Property)]
```

```
public class User
{
    [MyMaxLength(30, ErrorText = "شما اجازه ندارید بیش از 30 کاراکتر وارد نمایید")]
    public string Name { get; set; }
}
```

یکی دیگر از ویژگی های AttributeUsage خصوصیتی به اسم AllowMultiple است که اجازه می دهد بیش از یک بار این متادیتا، بر روی یک نوع استفاده شود:

```
[AttributeUsage(AttributeTargets.Property,AllowMultiple = true)]
public class MyMaxLength:Attribute
{
    //....
}
```

که تعریف چندگانه آن به شکل زیر می شود:

```
[MyMaxLength(40, ErrorText = "شما اجازه ندارید بیش از 40 کاراکتر وارد نمایید")
[MyMaxLength(50, ErrorText = "شما اجازه ندارید بیش از 50 کاراکتر وارد نمایید")
[MyMaxLength(30, ErrorText = "شما اجازه ندارید بیش از 30 کاراکتر وارد نمایید")
public string Name { get; set; }
```

در این مثال ما فقط اجازه ی یکبار استفاده را خواهیم داد؛ پس مقدار این ویژگی را false قرار می دهیم.

آخرین ویژگی که این متادیتا در دسترس ما قرار می دهد، استفاده از خصوصیت ارث بری است که به طور پیش فرض با True مقداردهی شده است. موقعی که شما یک متادیتا را به ویژگی ارث بری مین کنید، در صورتی که آن کلاس که برایش متادیتا تعریف می کنید به عنوان والد مورد استفاده قرار بگیرد، فرزند آن هم به طور خودکار این متادیتا برایش منظور می گردد. به مثال های زیر دقت کنید:

دو عدد متادیتا تعریف شده که یکی از آنها ارث بری در آن فعال شده و دیگری خیر.

```
public class MyAttribute : Attribute
{
    //...
}
```

```
[AttributeUsage(AttributeTargets.Method, Inherited = false)]
public class YourAttribute : Attribute
{
    //...
}
```

هر دو متادیتا بر سر یک متد در یک کلاسی که بعد از آن ارث بری می شود تعریف شده اند.

```
public class MyClass
{
    [MyAttribute]
    [YourAttribute]
    public virtual void MyMethod()
    {
        //...
    }
}
```

در کد زیر کلاس بالا به عنوان والد معرفی شده و متد کلاس فرزند الان شامل متادیتایی به اسم MyAttribute است، ولی متادیتای YourAttribute بر روی آن تعریف نشده است.

```
public class YourClass : MyClass
{
    public override void MyMethod()
    {
        //...
    }
}
```

الان که با نحوه ی تعریف یکی از متادیتاها آشنا شدیم، این بحث پیش می آید که چگونه Type مورد نظر را تحت تاثیر این متادیتا قرار دهیم. الان چگونه میتوانم حداکثر متنی که یک پراپرتی می گیرد را کنترل کنم. در اینجا ما از مفهومی به نام [Reflection](#) استفاده می کنیم. با استفاده از این مفهوم ما میتوانیم به تمامی قسمت های یک Type دسترسی داشته باشیم. متأسفانه دسترسی مستقیمی از داخل کلاس متادیتا به نوع مورد نظر نداریم. کد زیر تمامی پراپرتی های یک کلاس را چک میکند و سپس ویژگی های هر پراپرتی را دنبال کرده و در صورتیکه متادیتای مورد نظر به آن پراپرتی ضمیمه شده باشد، حالا می توانید عملیات را انجام دهید. کد زیر میتواند در هر جایی نوشته شود. داخل کلاسی که که به آن متادیتا ضمیمه می کنید یا داخل تابع Main در اپلیکشین ها و هر جای دیگر. مقدار True که به متد GetCustomAttributes پاس می شود باعث می شود تا متادیتاهای ارث بری شده هم لحاظ گردند.

```
Type type = typeof (User);

foreach (PropertyInfo property in type.GetProperties())
{
    foreach (Attribute attribute in property.GetCustomAttributes(true))
    {
        MyMaxLength max = attribute as MyMaxLength;
        if (max != null)
        {
            string Max = max.ErrorText;
            //انجام عملیات
        }
    }
}
```

البته یک ترفند جهت دسترسی به کلاس ها از داخل کلاس متادیتا وجود دارد و آن هم این هست که نوع را از طریق پارامتر به سمت متادیتا ارسال کنید. هر چند این کار زیبایی ندارد ولی به هر حال روش خوبی برای کنترل از داخل کلاس متادیتا و همچنین منظم سازی و دسته بندی و کم کردن کد دارد.

```
[MyMaxLength(30, typeof(User))]
```

در این مقاله قصد داریم عملیات Reflection را بیشتر در انجام ساده‌تر عملیات ببینیم. عملیاتی که به همراه کار اضافه، تکراری و خسته کننده است و با استفاده از Reflection این کارها حذف شده و تعداد خطوط هم پایین می‌آید. حتی گاهی ممکن است موجب استفاده‌ی مجدد از کدها شود که همگی این عوامل موجب بالا رفتن امتیاز Refactoring می‌شوند.

در مثال‌های زیر مجموعه‌ای از Reflection‌های ساده و کاملاً کاربردی است که من با آن‌ها روبرو شده‌ام.

کوتاه سازی کدهای نمایش یک View در ASP.NET MVC با Reflection

یکی از قسمت‌هایی که مرتباً با آن سر و کار دارید، نمایش اطلاعات است. حتی یک جدول را هم که می‌خواهید بسازید، باید ستون‌های آن جدول را یک به یک معرفی کنید. ولی در عمل، یک Reflection ساده این کار به یک تابع چند خطی و سپس برای ترسیم هر ستون جدول از دو خط استفاده خواهید کرد ولی مزیتی که دارد این است که این تابع برای تمامی جدول‌ها کاربردی عمومی پیدا می‌کند. برای نمونه دوست داشتم برای بخش مدیر، قسمت پروفایلی را ایجاد کنم و در آن اطلاعاتی چون نام، نام خانوادگی، تاریخ تولد، تاریخ ایجاد و خیلی از اطلاعات دیگر را نمایش دهم. به جای اینکه بیایم برای هر قسمت یک خط partial ایجاد کنم، با استفاده از reflection و یک حلقه، تمامی اطلاعات را به آن پارشال پاس می‌کنم. مزیت این روش این است که اگر بخواهم در یک جای دیگر، اطلاعات یک محصول یا یک فاکتور را هم نمایش دهم، باز هم همین تابع برایم کاربرد خواهد داشت:

تصویر زیر را که برگرفته از یک قالب Bootstrap است، ملاحظه کنید. اصلاً علاقه ندارم که برای یک به یک آن‌ها، یک سطر جدید را تعریف کنم و به View بگویم این پراپرتی را نشان بده؛ دوباره مورد بعدی هم به همین صورت و دوباره و دوباره و دوست دارم یک تابع عمومی، همه‌ی این کارها را خودکار انجام دهد.

First Name	: Jonathan	Last Name	: Smith
Country	: Australia	Birthday	: 13 July 1983
Occupation	: UI Designer	Email	: jsmith@flatlab.com
Mobile	: (12) 03 4567890	Phone	: 88 (02) 123456

ساختار اطلاعاتی تصویر فوق به شرح زیر است:

```
<div>
    <div>
        <div>
            <p><span>First Name </span>: Jonathan</p>
        </div>
    </div>
</div>
```

که به دو فایل پارشال تقسیم شده است Bio و BioRow_ که محتویات هر پارشال هم به شرح زیر است:

_BioRow

@model System.Web.UI.WebControls.ListItem

```
<div>
  <p><span>@Model.Text </span>: @Model.Value</p>
</div>
```

در پارشال بالا ورودی از نوع listItem است که یک متن دارد و یک مقدار.(شاید به نظر شبیه حالت جفت کلید و مقدار باشد ولی در این کلاس خبری از کلید نیست).

پارشال پایینی هم دربرگیرنده‌ی پارشال بالاست که قرار است چندین و چند بار پارشال بالا در خودش نمایش دهد.

_Bio

```
@using System.Web.UI.WebControls
@using ZekrWebApp.Filters
@model ZekrModel.Admin

<div>
  <h1>Bio Graph</h1>
  <div>

    @{
      ListItemCollection collection = GetCustomProperties.Get(Model,exclude:new
string[]{"Poems","Id"});
      foreach (var item in collection)
      {
        Html.RenderPartial(MVC.Shared.Views._BioRow, item);
      }
    }

  </div>
</div>
```

پارشال بالا یک مدل از کلاس Admin را می‌پذیرد که قرار است اطلاعات شخصی مدیر را نمایش دهد. در ابتدا متدی از یک کلاس ایستا وجود دارد که کدهای Reflection درون آن قرار دارند که یک مجموعه از ListItem‌ها را بر می‌گرداند و سپس با یک حلقه، پارشال _BioRow را صدا می‌زند.

کد درون این کلاس ایستا را بررسی می‌کنیم؛ این کلاس دو متد دارد یکی عمومی و دیگری خصوصی است:

```
public class GetCustomProperties
{
  private static PropertyInfo[] getObjectsInfos(object obj,string[] include,string[] exclude )
  {
    var list = obj.GetType().GetProperties();
    PropertyInfo[] outputPropertyInfos = null;
    if (include != null)
    {
      return list.Where(propertyInfo => include.Contains(propertyInfo.Name)).ToArray();
    }
    if (exclude != null)
    {
      return list.Where(propertyInfo => !exclude.Contains(propertyInfo.Name)).ToArray();
    }
    return list;
  }
}
```

کد بالا که یک کد خصوصی است، سه پارامتر را می‌پذیرد. اولی مدل یا کلاسی است که به آن پاس کرده‌ایم. دو پارامتر بعدی اختیاری است و در کد پارشال بالا Exclude را تعریف کرده ایم و تنه‌ای یکی از دو پارامتر بالا هم باید مورد استفاده قرار بگیرند و Include ارجحیت دارد. وظیفه‌ی این دو پارامتر این است که آرایه ای از رشته‌ها را دریافت می‌کنند که نام پراپرتی‌ها در آن‌ها ذکر شده است. آرایه Include می‌گوید که فقط این پراپرتی‌ها را برگردان ولی اگر دوست دارید همه‌ی پارامترها را نمایش دهید و تنها یکی یا چندتا از آن‌ها را حذف کنید، از آرایه Exclude استفاده کنید. در صورتی که این دو آرایه خالی باشند، همه‌ی پراپرتی‌ها

بازگشت داده می‌شوند و در صورتی که یکی از آن‌ها وارد شده باشد، طبق دستورات Linq بالا بررسی می‌کند که (Include) آیا اسامی مشترکی بین آن‌ها وجود دارد یا خیر؟ اگر وجود دارد آن را در لیست قرار داده و بر می‌گرداند و در حالت Exclude این مقایسه به صورت برعکس انجام می‌گیرد و باید لیستی برگردانده شود که اسامی، نکته مشترکی نداشته باشند.

متد عمومی که در این کلاس قرار دارد به شرح زیر است:

```
public static List<Item> Get(object obj, string[] include=null, string[] exclude=null)
{
    var propertyInfos = getObjectsInfos(obj, include, exclude);
    if (propertyInfos == null) throw new ArgumentNullException("propertyInfos is null");

    var collection = new List<Item>();
    foreach (PropertyInfo propertyInfo in propertyInfos)
    {
        string name = propertyInfo.Name;

        foreach (Attribute attribute in propertyInfo.GetCustomAttributes(true))
        {
            DisplayAttribute displayAttribute = attribute as DisplayAttribute;

            if (displayAttribute != null)
            {
                name = displayAttribute.Name;
                break;
            }
        }

        string value = "";
        object objvalue = propertyInfo.GetValue(obj);
        if (objvalue != null) value = objvalue.ToString();

        collection.Add(new Item(name, value));
    }
    return collection;
}
```

این متد سه پارامتر را از کاربر دریافت و به سمت متد خصوصی ارسال می‌کند. موقعی که پراپرتی‌ها بازگشت داده می‌شوند، دو قسمت آن مهم است؛ یکی عنوان پراپرتی و دیگری مقدار پراپرتی. از آن جا که نام پراپرتی‌ها طبق سلیقه‌ی برنامه نویس و با حروف انگلیسی نوشته می‌شوند، در صورتی که برنامه نویس از متادیتای Display در مدل بهره برده باشد، به جای نام پراپرتی مقداری را که به متادیتای Display داده‌ایم، بر می‌گردانیم.

کد بالا پراپرتی‌ها را دریافت و یک به یک متادیتاهای آن را بررسی کرده و در صورتی که از متادیتای Display استفاده کرده باشند، مقدار آن را جایگزین نام پراپرتی خواهد کرد. در مورد مقدار هم از آنجا که اگر پراپرتی با Null پر شده باشد، تبدیل به رشته‌ای با پیام خطای روبرو خواهد شد. در نتیجه بهتر است یک شرط احتیاط هم روی آن پیاده شود. در آخر هم از متن و مقدار، یک آیتم ساخته و درون Collection اضافه می‌کنیم و بعد از اینکه همه پراپرتی‌ها بررسی شدند، Collection را بر می‌گردانیم.

```
[Display(Name = "نام کاربری")]
public string UserName { get; set; }
```

کد کامل کلاس:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Linq.Expressions;
using System.Reflection;
using System.Web;
using System.Web.Mvc.Html;
using System.Web.UI.WebControls;
using Links;

namespace ZekrWepApp.Filters
```

```

{
    public class GetCustomProperties
    {
        public static ListItemCollection Get(object obj,string[] include=null,string[] exclude=null)
        {
            var propertyInfos = getObjectsInfos(obj, include, exclude);
            if (propertyInfos == null) throw new ArgumentNullException("propertyInfos is null");

            var collection = new ListItemCollection();

            foreach (PropertyInfo propertyInfo in propertyInfos)
            {
                string name = propertyInfo.Name;
                foreach (Attribute attribute in propertyInfo.GetCustomAttributes(true))
                {
                    DisplayAttribute displayAttribute = attribute as DisplayAttribute;

                    if (displayAttribute != null)
                    {
                        name = displayAttribute.Name;
                        break;
                    }
                }

                string value = "";
                object objvalue = propertyInfo.GetValue(obj);
                if (objvalue != null) value = objvalue.ToString();

                collection.Add(new ListItem(name,value));
            }
            return collection;
        }
        private static PropertyInfo[] getObjectsInfos(object obj,string[] include,string[] exclude )
        {
            var list = obj.GetType().GetProperties();

            PropertyInfo[] outputPropertyInfos = null;

            if (include != null)
            {
                return list.Where(propertyInfo => include.Contains(propertyInfo.Name)).ToArray();
            }
            if (exclude != null)
            {
                return list.Where(propertyInfo => !exclude.Contains(propertyInfo.Name)).ToArray();
            }
            return list;
        }
    }
}

```

لیستی از پارامترها با Reflection

مورد بعدی که ساده‌تر بوده و از کد بالا مختصرتر هم هست، این است که قرار بود برای یک درگاه، یک سری اطلاعات را با متد Post ارسال کنم که نحوه‌ی ارسال اطلاعات به شکل زیر بود:

```
amount=1000&orderId=452&Pid=xxx&....
```

کد زیر را من جهت ساخت قالب‌های این چینی استفاده می‌کنم:

```

using System;
using System.Collections.Generic;
using System.Linq;

namespace Utils
{
    public class QueryStringParametersList
    {
        private string Symbol = "&";
        private List<KeyValuePair<string, string>> list { get; set; }
    }
}

```

```

public QueryStringParametersList()
{
    list = new List<KeyValuePair<string, string>>();
}
public QueryStringParametersList(string symbol)
{
    Symbol = symbol;
    list = new List<KeyValuePair<string, string>>();
}

public int Size
{
    get { return list.Count; }
}
public void Add(string key, string value)
{
    list.Add(new KeyValuePair<string, string>(key, value));
}

public string GetQueryStringPostfix()
{
    return string.Join(Symbol, list.Select(p => Uri.EscapeDataString(p.Key) + "=" +
Uri.EscapeDataString(p.Value)));
}
}

```

یک متغیر به نام symbol دارد و در صورتی در شرایط متفاوت، قصد چسپاندن چیزی را به یکدیگر با علامتی خاص داشته باشید، این تابع می‌تواند کاربرد داشته باشد. این متد از یک لیست کلید و مقدار استفاده کرده و پارامترهایی را که به آن پاس می‌شود، نگهداری و سپس توسط متد GetQueryStringPostfix آن‌ها را با یکدیگر الحاق کرده و در قالب یک رشته بر می‌گرداند. کاربرد Reflection در اینجا این است که من باید دوبار به شکل زیر، دو نوع اطلاعات متفاوت را پست کنم. یکی موقع ارسال به درگاه و دیگری موقع بازگشت از درگاه.

```

QueryStringParametersList queryparamsList = new QueryStringParametersList();

queryparamsList.Add("consumer_key", requestPayment.Consumer_Key);
queryparamsList.Add("amount", requestPayment.Amount.ToString());
queryparamsList.Add("callback", requestPayment.Callback);
queryparamsList.Add("description", requestPayment.Description);
queryparamsList.Add("email", requestPayment.Email);
queryparamsList.Add("mobile", requestPayment.Mobile);
queryparamsList.Add("name", requestPayment.Name);
queryparamsList.Add("irderid", requestPayment.OrderId.ToString());

```

ولی با استفاده از کد Reflection که در بالاتر عنوان شد، باید نام و مقدار پراپرتی را گرفته و در یک حلقه آن‌ها را اضافه کنیم، بدین شکل:

```

private QueryStringParametersList ReadParams(object obj)
{
    PropertyInfo[] propertyInfos = obj.GetType().GetProperties();

    QueryStringParametersList queryparamsList = new QueryStringParametersList();
    for (int i = 0; i < propertyInfos.Count(); i++)
    {
        queryparamsList.Add(propertyInfos[i].Name.ToLower(), propertyInfos[i].GetValue(obj).ToString());
    }
    return queryparamsList;
}

```

در کد بالا هر بار پراپرتی‌های کلاس را خوانده و نام و مقدار آن‌ها را گرفته و به کلاس QueryString اضافه می‌کنیم. پارامتر ورودی این متد به این خاطر object در نظر گرفته شده است که تا هر کلاسی را بتوانیم به آن پاس کنیم که خودم در همین کلاس درگاه، دو کلاس را به آن پاس کردم.

فرض کنید یک چنین کلاسی طراحی شده‌است:

```
public class NestedClass
{
    private int _field2;
    public NestedClass()
    {
        _field2 = 12;
    }
}

public class MyClass
{
    private int _field1;
    private NestedClass _nestedClass;

    public MyClass()
    {
        _field1 = 1;
        _nestedClass = new NestedClass();
    }

    private string GetData()
    {
        return "Test";
    }
}
```

می‌خواهیم از طریق Reflection مقادیر فیلدها و متدهای مخفی آن‌را بخوانیم.
حالت متداول دسترسی به فیلد خصوصی آن از طریق Reflection، یک چنین شکلی را دارد:

```
var myClass = new MyClass();

var field1Obj = myClass.GetType().GetField("_field1", BindingFlags.NonPublic | BindingFlags.Instance);
if (field1Obj != null)
{
    Console.WriteLine(Convert.ToInt32(field1Obj.GetValue(myClass)));
}
```

و یا دسترسی به مقدار خروجی متد خصوصی آن، به نحو زیر است:

```
var getDataMethod = myClass.GetType().GetMethod("GetData", BindingFlags.NonPublic | BindingFlags.Instance);
if (getDataMethod != null)
{
    Console.WriteLine(getDataMethod.Invoke(myClass, null));
}
```

در اینجا دسترسی به مقدار فیلد مخفی NestedClass، شامل مراحل زیر است:

```
var nestedClassObj = myClass.GetType().GetField("_nestedClass", BindingFlags.NonPublic | BindingFlags.Instance);
if (nestedClassObj != null)
{
    var nestedClassFieldValue = nestedClassObj.GetValue(myClass);
    var field2Obj = nestedClassFieldValue.GetType().GetField("_field2", BindingFlags.NonPublic | BindingFlags.Instance);
    if (field2Obj != null)
    {
        Console.WriteLine(Convert.ToInt32(field2Obj.GetValue(nestedClassFieldValue)));
    }
}
```

البته این مقدار کد فقط برای دسترسی به دو سطح تو در تو بود.

چقدر خوب بود اگر می‌شد بجای این همه کد، نوشت:

```
myClass._field1
myClass._nestedClass._field2
myClass.GetData()
```

نه؟!

برای این مشکل راه حلی معرفی شده‌است به نام Dynamic Proxy که در ادامه به معرفی آن خواهیم پرداخت.

معرفی Dynamic Proxy

Dynamic Proxy یکی از [مفاهیم AOP](#) است. به این معنا که توسط آن یک محصور کننده نامرئی، اطراف یک شیء تشکیل خواهد شد. از این غشای نامرئی عموماً جهت مباحث ردیابی اطلاعات، مانند پروکسی‌های Entity framework، همانجایی که تشخیص می‌دهد کدام خاصیت به روز شده‌است یا خیر، استفاده می‌شود و یا این غشای نامرئی کمک می‌کند که در حین دسترسی به خاصیت یا متدی، بتوان منطق خاصی را در این بین تزریق کرد. برای مثال فرآیند تکراری logging سیستم را به این غشای نامرئی منتقل کرد و به این ترتیب می‌توان به کدهای تمیزتری رسید. یکی دیگر از کاربردهای این محصور کننده یا غشای نامرئی، ساده سازی مباحث Reflection است که نمونه‌ای از آن در پروژه‌ی [EntityFramework.Extended](#) بکار رفته‌است. در اینجا، کار با محصور سازی نمونه‌ای از کلاس مورد نظر با Dynamic Proxy شروع می‌شود. سپس کل عملیات Reflection فوق در همین چند سطر ذیل به نحوی کاملاً عادی و طبیعی قابل انجام است:

```
// Accessing a private field
dynamic myClassProxy = new DynamicProxy(myClass);
dynamic field1 = myClassProxy._field1;
Console.WriteLine((int)field1);

// Accessing a nested private field
dynamic field2 = myClassProxy._nestedClass._field2;
Console.WriteLine((int)field2);

// Accessing a private method
dynamic data = myClassProxy.GetData();
Console.WriteLine((string)data);
```

خروجی Dynamic Proxy از نوع dynamic دات نت 4 است. پس از آن می‌توان در اینجا هر نوع خاصیت یا متد دلخواهی را به شکل dynamic تعریف کرد و سپس به مقادیر آن‌ها دسترسی داشت.

بنابراین با استفاده از Dynamic Proxy فوق می‌توان به دو مهم دست یافت:

1) ساده سازی و زیبا سازی کدهای کار با Reflection

2) استفاده‌ی ضمنی از مباحث [Fast Reflection](#). در کتابخانه‌ی Dynamic Proxy معرفی شده، دسترسی به خواص و متدها، توسط [کدهای IL](#) بهینه سازی شده‌است و در دفعات آتی کار با آن‌ها، دیگر شاهد سربار بالای Reflection نخواهیم بود.

کدهای کامل این مثال را از اینجا می‌توانید دریافت کنید:

[DynamicProxyTests.zip](#)

طی این مقاله، نحوه‌ی ذخیره سازی تنظیمات متغیر و پویای یک برنامه را به صورت Strongly Typed ارائه خواهیم داد. برای این منظور، یک API را که از Lazy Loading ، Cache ، Reflection 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 دسترس 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 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 Name ، Type ، Value 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 تعریف 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:

	Name	Type	Value
▶	AdminEmail	GeneralSettings	NULL
	HomeMetaDescription	SeoSettings	Welcome to Talk Sharp
	HomeMetaTitle	SeoSettings	NULL
	SiteName	GeneralSettings	Talk Sharp
*	NULL	NULL	NULL

کلاس SettingBase ما همچنین 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 در دیتابیس 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.

```

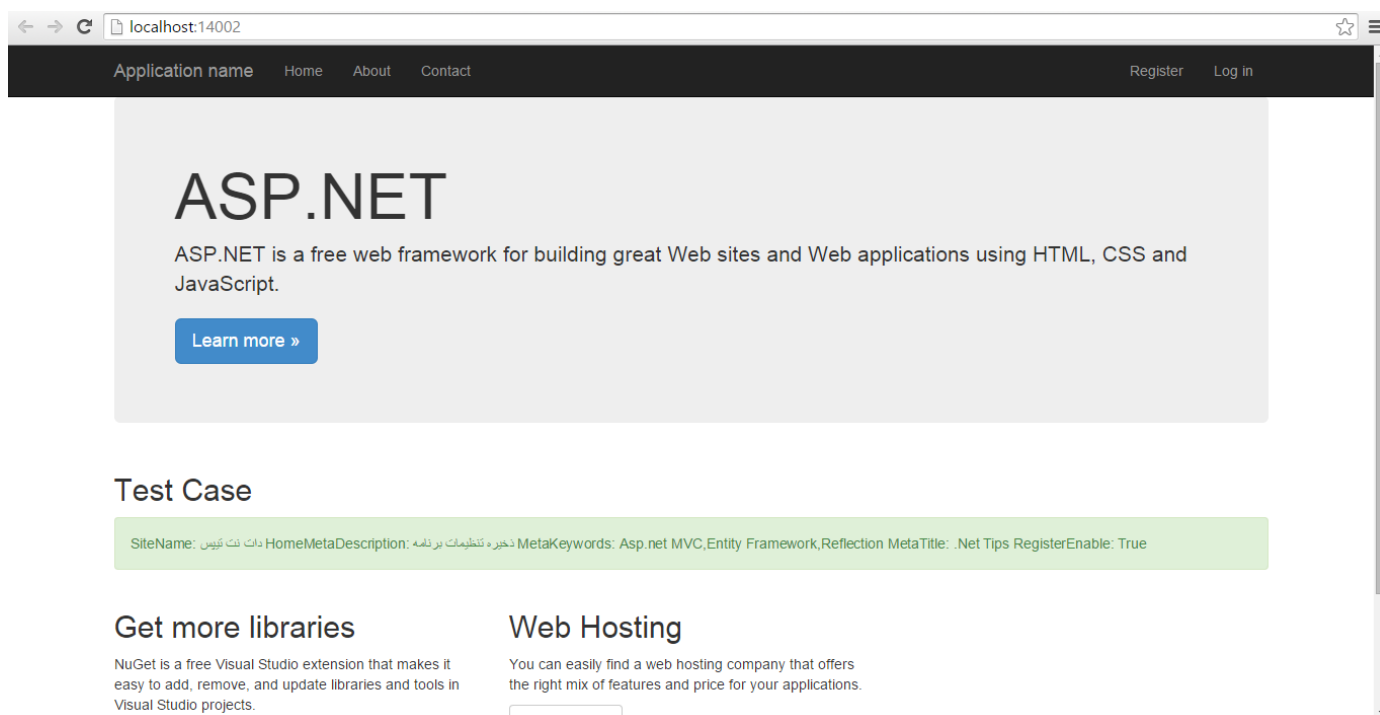
public ActionResult Index()
{
    using (var uow = new ApplicationDbContext())
    {
        var _settings = new Settings(uow);
        _settings.General.SiteName = "دات نت تیپس";
        _settings.General.AdminEmail = "admin@gmail.com";
        _settings.General.RegisterUsersEnabled = true;
        _settings.Seo.HomeMetaTitle = ".Net Tips";
        _settings.Seo.MetaKeywords = "AspNet 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 جدیداً](#) 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. حتماً در پروژه‌ی مذکور همچنین امکانی را هم در نظر خواهم گرفت.

پیشنهاد میکنم سورس [SmartStore](#) را بررسی کنید. آن هم به شکل مشابهی ولی پیشرفته‌تر از این مقاله، همچنین امکانی را دارد. [DynamicSettingAPI.zip](#)

چند روز پیش در حال استفاده از افزونه‌ی [jQuery Bootgrid](#) بودم که داده‌های خود را در قالب زیر به صورت کوئری استرینگ ارسال می‌کند.

```
current=1&rowCount=10&sort[sender]=asc&searchPhrase=&id=b0df282a-0d67-40e5-8558-c9e93b7befed
```

قبلا هم با کوئری استرینگ‌ها کار کرده‌ایم و [نحوه دریافت](#) آن را یاد گرفته‌ایم و میدانیم که اگر کلاس شما شامل پراپرتی‌های همنام با کلیدهای کوئری استرینگ باشد مستقیماً در کلاس شما جا می‌گیرند؛ ولی من دوست داشتم که پراپرتی‌های کلاس‌م نام دلخواه من را داشته باشد و اجباری به استفاده از نام‌های کوئری استرینگ نداشته باشد. اصلاً ممکن است افراد Back End یک سری کد نوشته‌اند و کلاسشان را هم ساخته‌اند و اصلاً کاری با من ندارند که چطوری داده‌ها را و با چه اسامی از آن‌ها دریافت می‌کنم و فقط انتظار دارند که کلاس آن‌ها را با اطلاعات دریافتی پر کنم و ارسال کنم. اگر بخواهیم به طور دستی هر یک از کلیدها را چک کنیم، هم کدنویسی طولانی می‌شود و هم کد قابلیت استفاده مجدد ندارد. پس بهترین کار این است که یک کد با قابلیت استفاده مجدد بنویسیم.

دوست دارم چیزی شبیه به Deserialize کردن فرمت json توسط کتابخانه Json.net داشته باشم؛ پس در اولین قدم یک [attribute](#) با مشخصات زیر می‌سازیم:

```
[AttributeUsage(AttributeTargets.Property, Inherited = true)]
public class RequestBodyField:Attribute
{
    public string Field;
    public RequestBodyField(string field)
    {
        this.Field = field;
    }
}
```

سپس در کلاس اصلی، ما این خصوصیت‌ها را در بالای propertyها تعریف کرده و با کلیدهای موجود در کوئری استرینگ برابر می‌کنیم:

```
public class EmployeesRequestBody
{
    [RequestBodyField("current")]
    public int CurrentPage { get; set; }

    [RequestBodyField("rowCount")]
    public int RowCount { get; set; }

    [RequestBodyField("searchPhrase")]
    public string SearchPhrase { get; set; }

    [RequestBodyField("sort")]
    public NameValueCollection SortDictionary { get; set; }
}
```

سپس کلاس زیر را می‌نویسیم که وظیفه دارد کلاس‌های از جنس بالا را با query stringها رسیده در درخواست مطابقت دهد:

```
public T GetFromQueryString<T>() where T : new()
{
    var obj = new T();

    var queryString = HttpContext.Current.Request.QueryString;
    var queries = HttpUtility.ParseQueryString(queryString.ToString());
```

```
var properties = typeof(T).GetProperties();
foreach (var property in properties)
{
    foreach (Attribute attribute in property.GetCustomAttributes(true))
    {
        var requestBodyField = attribute as RequestBodyField;
        if (requestBodyField == null) continue;

        //get value of query string
        var valueAsString = queries[requestBodyField.Field];

        var converter = TypeDescriptor.GetConverter(property.PropertyType);
        var value = converter.ConvertFrom(valueAsString);

        if (value == null)
            continue;

        property.SetValue(obj, value, null);
    }
}
return obj;
}
```

این متد یک تعریف کلاس را دریافت می‌کند. سپس رشته‌ی کوئری استرینگ موجود در بدنه درخواست را دریافت کرده و با استفاده از کد زیر اشیا را به صورت nameValueCollection دریافت می‌کنیم.

```
HttpContext.Current.Request.QueryString
```

نکته: همچنین متد httputility.parseQueryString یک رشته کوئری استرینگ دریافت می‌کند و کوئری استرینگ را به زوج نام و مقدار nameValueCollection تبدیل می‌کند.

سپس در مرحله‌ی بعدی با استفاده از [Reflection](#) پراپرتی‌هایی را که دارای attribute تعریف شده هستند، پیدا می‌کنیم.

مقدار داده شده به attribute را در nameValueCollection بررسی می‌کنیم و در صورت موجود بودن، مقدار آن را می‌گیریم. از آنجا که این مقدار از نوع رشته است و ممکن است مقدار داخل آن عددی یا هر نوع دیگری باشد، باید آن را به نوع صحیح تبدیل کنیم که خطوط زیر کار تبدیل را انجام می‌دهند:

```
var converter = TypeDescriptor.GetConverter(property.PropertyType);
var value = converter.ConvertFrom(valueAsString);
```

در خط اول بر اساس نوع property کلاس، یک converter دریافت می‌کنیم و سپس مقدار ارسال شده را به آن می‌دهیم تا مقدار جدید را با نوع صحیح خود، دریافت کنیم. سپس در صورتی که مقدار صحیح دریافت شود و برابر null نباشد، مقدار را در پراپرتی مربوطه جا می‌دهیم.

نکته‌ای که در اینجا نیاز به تلاش بیشتر دارد، کلید sort در کوئری استرینگ است. با نگاهی دقیق‌تر متوجه می‌شوید که خود کلید دو مقدار دارد که یکی از مقادیرش با کلید ترکیب شده است. این حالت روش ارسال آرایه‌ها با نام کلیدی متفاوت در کوئری استرینگ است. این حالت ارسال باعث می‌شود که گرید بتواند حالت multi sort را نیز پیاده سازی کند. پس برای دریافت این نوع مقادیر کمی کد به آن اضافه می‌کنیم. برای دریافت مقادیر آرایه‌ای کد زیر را به سیستم اضافه می‌کنیم:

```
if (valueAsString == null)
{
    var keys = from key in queries.AllKeys where
    key.StartsWith(requestBodyField.Field) select key;

    var collection = new NameValueCollection();
    foreach (var key in keys)
    {
```

```

        var openBracketIndex = key.IndexOf("[", StringComparison.Ordinal);
        var closeBracketIndex = key.IndexOf("]", StringComparison.Ordinal);

        if (openBracketIndex < 0 || closeBracketIndex < 0)
            throw new Exception("query string is corrupted.");

        openBracketIndex++;
        //get key in [...]
        var fieldName = key.Substring(openBracketIndex, closeBracketIndex -
openBracketIndex);

        collection.Add(fieldName, queries[key] );
    }
    property.SetValue(obj, collection, null);
    continue;
}

```

در صورتیکه شما کلید sort را درخواست کنید و از آنجا که کلید اصلی با نام sort[sender] است، مقدار null بازگشت می‌دهد. پس ما می‌توانیم به این مقدار شک کنیم که شاید این کلید حاوی مقدار مورد نظر ماست؛ پس این حالت را بررسی می‌کنیم. برای بررسی، با استفاده از linq بررسی می‌کنیم که اگر کلیدهای namValueCollection با این کلید (در اینجا sort) آغاز می‌شوند، پس به احتمال زیاد همان حالت مورد نظر ما رخ داده است. پس اندیس‌های [و] را می‌گیریم و اگر اندیس هر دو بزرگتر از صفر بود مقدار ما بین آن را به عنوان کلید بیرون می‌کشیم و در یک namValueCollection جدید قرار می‌دهیم و در نهایت به پراپرتی پاس می‌دهیم. کد نهایی این متد به شکل زیر است:

```

public T GetFromQueryString<T>() where T : new()
{
    var obj = new T();
    var properties = typeof(T).GetProperties();

    var queryString = HttpContext.Current.Request.QueryString;
    var queries = HttpUtility.ParseQueryString(queryString.ToString());

    foreach (var property in properties)
    {
        foreach (Attribute attribute in property.GetCustomAttributes(true))
        {
            var requestBodyField = attribute as RequestBodyField;
            if (requestBodyField == null) continue;

            //get value of query string
            var valueAsString = queries[requestBodyField.Field];

            if (valueAsString == null)
            {
                var keys = from key in queries.AllKeys where
key.StartsWith(requestBodyField.Field) select key;

                var collection = new NameValueCollection();

                foreach (var key in keys)
                {
                    var openBracketIndex = key.IndexOf("[", StringComparison.Ordinal);
                    var closeBracketIndex = key.IndexOf("]", StringComparison.Ordinal);

                    if (openBracketIndex < 0 || closeBracketIndex < 0)
                        throw new Exception("query string is corrupted.");

                    openBracketIndex++;
                    //get key in [...]
                    var fieldName = key.Substring(openBracketIndex, closeBracketIndex -
openBracketIndex);

                    collection.Add(fieldName, queries[key]);
                }
                property.SetValue(obj, collection, null);
                continue;
            }

            var converter = TypeDescriptor.GetConverter(property.PropertyType);
            var value = converter.ConvertFrom(valueAsString);

            if (value == null)
                continue;

            property.SetValue(obj, value, null);
        }
    }
}

```

```
    }  
    return obj;  
}
```

حال به صورت زیر این متد را صدا می‌زنیم:

```
public virtual ActionResult GetEmployees()  
{  
    var request = new Requests().GetFromQueryString<EmployeesRequestBody>();  
}
```

نظرات خوانندگان

نویسنده: علی یگانه مقدم
تاریخ: ۱۴:۰۱۳۹۴/۰۷/۱۷

جهت رفع باگ کد زیر را

```
var keys = from key in queries.AllKeys where key.StartsWith(requestBodyField.Field+"") select key;
if(!keys.Any())
    continue;
```

بعد از خط

```
if (valueAsString == null)
{
```

اضافه کنید.

نویسنده: علی یگانه مقدم
تاریخ: ۱۴:۲۶۱۳۹۴/۰۷/۱۷

جهت افزودن پشتیبانی از حالت POST کد متد را به شکل زیر تغییر دهید:

```
public T GetFromQueryString<T>(RequestType type=RequestType.GET) where T : new()
{
    var obj = new T();
    var properties = typeof(T).GetProperties();

    var queries =new NameValueCollection();
    if(type==RequestType.GET)
        queries= HttpContext.Current.Request.QueryString;
    else
    {
        queries = HttpContext.Current.Request.Form;
    }

    foreach (var property in properties)
    {
        foreach (Attribute attribute in property.GetCustomAttributes(true))
        {
            var requestBodyField = attribute as RequestBodyField;
            if (requestBodyField == null) continue;

            //get value of query string
            var valueAsString = queries[requestBodyField.Field];

            if (valueAsString == null)
            {
                var keys = from key in queries.AllKeys where
key.StartsWith(requestBodyField.Field) select key;

                if(!keys.Any())
                    continue;

                var collection = new NameValueCollection();

                foreach (var key in keys)
                {
                    var openBracketIndex = key.IndexOf("[", StringComparison.Ordinal);
                    var closeBracketIndex = key.IndexOf("]", StringComparison.Ordinal);

                    if (openBracketIndex < 0 || closeBracketIndex < 0)
                        throw new Exception("query string is crupted.");

                    openBracketIndex++;
                    //get key in [...]
```



```
        var fieldName = key.Substring(openBracketIndex, closeBracketIndex -
openBracketIndex);
        collection.Add(fieldName, queries[key]);
    }
    property.SetValue(obj, collection, null);
    continue;
}

var converter = TypeDescriptor.GetConverter(property.PropertyType);
var value = converter.ConvertFrom(valueAsString);

if (value == null)
    continue;

property.SetValue(obj, value, null);
}
}
return obj;
}
```