

اگر بازار هدف یک محصول شامل چندین کشور، منطقه یا زبان مختلف باشد، طراحی و پیاده سازی آن برای پشتیبانی از ویژگی‌های چندزبانه یک فاکتور مهم به حساب می‌آید. یکی از بهترین روش‌های پیاده سازی این ویژگی در دات نت استفاده از فایل‌های Resource است. درواقع هدف اصلی استفاده از فایل‌های Resource نیز Globalization است. Globalization برابر است با Internationalization + Localization که به اختصار به آن glnn می‌گویند. در تعریف، Internationalization (یا به اختصار i18n) به فرایند طراحی یک محصول برای پشتیبانی از فرهنگ(culture)ها و زبانهای مختلف و Localization (یا L10n) یا بومی‌سازی به شخصی‌سازی یک برنامه برای یک فرهنگ یا زبان خاص گفته میشود. (اطلاعات بیشتر در [اینجا](#)).

استفاده از این فایل‌ها محدود به پیاده سازی ویژگی چندزبانه نیست. شما میتوانید از این فایل‌ها برای نگهداری تمام رشته‌های موردنیاز خود استفاده کنید. نکته دیگری که باید بدان اشاره کرد این است که تقریباً تمامی منابع مورد استفاده در یک محصول را میتوان درون این فایل‌ها ذخیره کرد. این منابع در حالت کلی شامل موارد زیر است:

- انواع رشته‌های مورد استفاده در برنامه چون لیبل‌ها و پیغام‌ها و یا مسیرها (مثلاً نشانی تصاویر یا نام کنترلرها و اکشن‌ها) و یا حتی برخی تنظیمات ویژه برنامه (که نمیخواهیم براحتی قابل نمایش یا تغییر باشد و یا اینکه بخواهیم با تغییر زبان تغییر کنند مثل direction و امثال آن)
- تصاویر و آیکونها و یا فایل‌های صوتی و انواع دیگر فایل‌ها
- و ...

نحوه بهره برداری از فایل‌های Resource در دات نت، پیاده سازی نسبتاً آسانی را در اختیار برنامه نویس قرار میدهد. برای استفاده از این فایل‌ها نیز روش‌های متنوعی وجود دارد که در مطلب جاری به چگونگی استفاده از آنها در پروژه‌های ASP.NET MVC پرداخته میشود.

Globalization در دات نت

فرمت نام یک culture دات نت (که در کلاس [CultureInfo](#) پیاده شده است) بر اساس استاندارد RFC 4646 ([^](#) و [^](#)) است. (در [اینجا](#) اطلاعاتی راجع به RFC یا Request for Comments آورده شده است). در این استاندارد نام یک فرهنگ (کالچر) ترکیبی از نام زبان به همراه نام کشور یا منطقه مربوطه است. نام زبان برپایه استاندارد ISO 639 که یک عبارت دوحرفی با حروف کوچک برای معرفی زبان است مثل fa برای فارسی و en برای انگلیسی و نام کشور یا منطقه نیز برپایه استاندارد ISO 3166 که به عبارت دوحرفی با حروف بزرگ برای معرفی یک کشور یا یک منطقه است مثل IR برای ایران یا US برای آمریکا است. برای نمونه میتوان به fa-IR برای زبان فارسی کشور ایران و یا en-US برای زبان انگلیسی آمریکایی اشاره کرد. البته در این روش نامگذاری یکی دو مورد استثنا هم وجود دارد (اطلاعات کامل کلیه زبانها: [National Language Support \(NLS\) API Reference](#)). یک فرهنگ خنثی (Neutral Culture) نیز تنها با استفاده از دو حرف نام زبان و بدون نام کشور یا منطقه معرفی میشود. مثل fa برای فارسی یا de برای آلمانی. در این بخش نیز دو استثنا وجود دارد ([^](#)).

در دات نت دو نوع culture وجود دارد: **Culture** و **UICulture**. هر دوی این مقادیر در هر Thread مقداری منحصر به فرد دارند. مقدار Culture بر روی توابع وابسته به فرهنگ (مثل فرمت رشته‌های تاریخ و اعداد و پول) تاثیر میگذارد. اما مقدار UICulture تعیین میکند که سیستم مدیریت منابع دات نت (Resource Manager) از کدام فایل Resource برای بارگذاری داده‌ها استفاده کند. درواقع در دات نت با استفاده از پراپرتی‌های موجود در کلاس استاتیک Thread برای ثرد جاری (که عبارتند از CurrentCulture و CurrentUICulture) برای فرمت کردن و یا انتخاب Resource مناسب تصمیم گیری میشود. برای تعیین کالچر جاری به صورت دستی میتوان بصورت زیر عمل کرد:

```
Thread.CurrentThread.CurrentUICulture = new CultureInfo("fa-IR");
Thread.CurrentThread.CurrentUICulture = CultureInfo.CreateSpecificCulture("fa-IR");
```

در اینجا باید اشاره کنم که کار انتخاب Resource مناسب با توجه به کالچر ثرد جاری توسط ResourceProviderFactory پیشفرض دات نت انجام میشود. در مطالب بعدی به نحوه تعریف یک پرووایدر شخصی سازی شده هم خواهیم پرداخت.

پشتیبانی از زبانهای مختلف در MVC

برای استفاده از ویژگی چندزبانه در MVC دو روش کلی وجود دارد.

1. استفاده از فایل‌های Resource برای تمامی رشته‌های موجود

2. استفاده از View‌های مختلف برای هر زبان

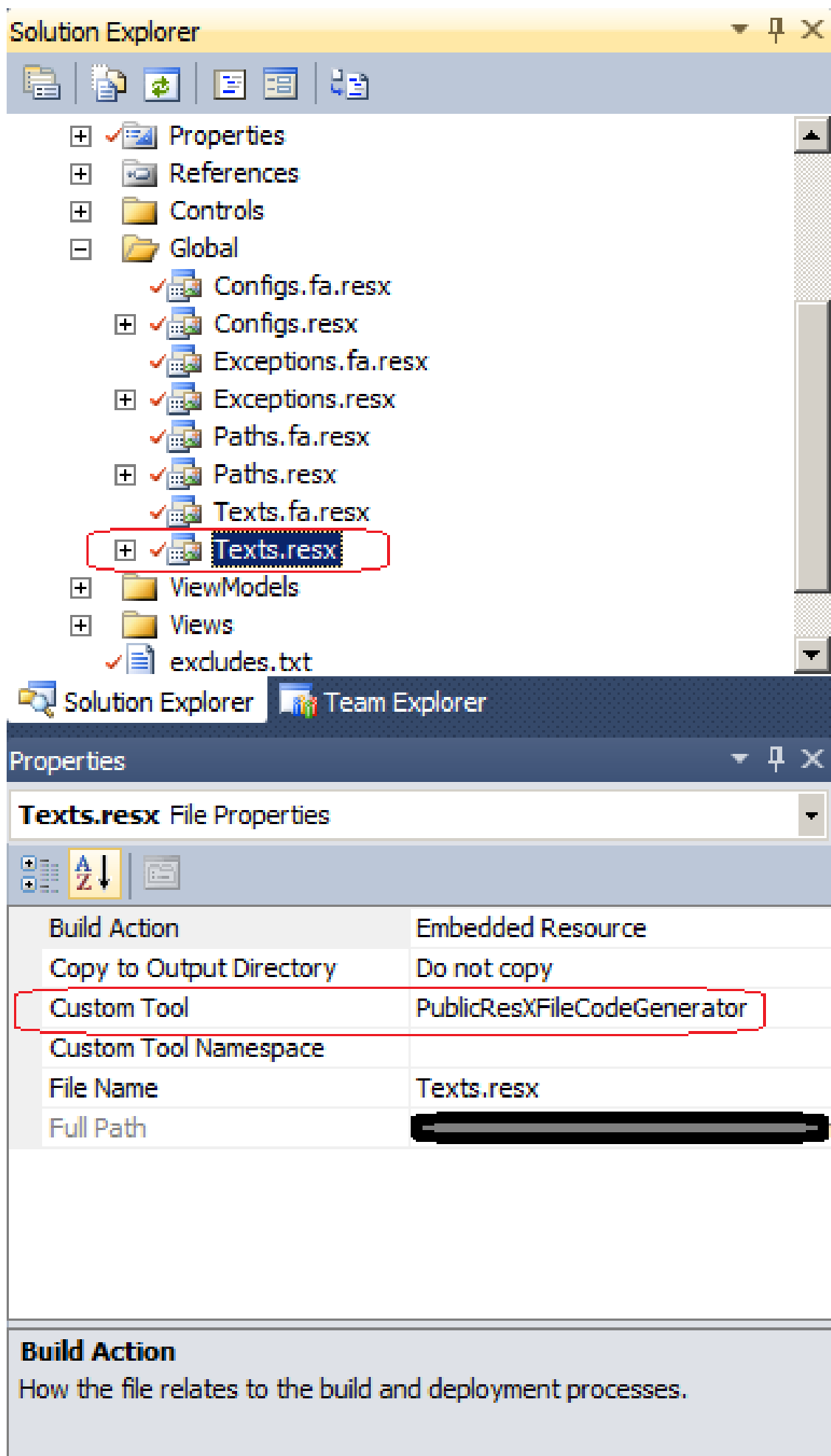
البته روش سوم هم که از ترکیب این دو روش استفاده میکند نیز وجود دارد. انتخاب روش مناسب کمی به سلیقه‌ها و عادات برنامه نویسی بستگی دارد. اگر فکر میکنید که استفاده از ویوهای مختلف به دلیل جداسازی مفاهیم درگیر در کالچرها (مثل جانمایی اجزای مختلف ویوها یا بحث Direction) باعث مدیریت بهتر و کاهش هزینه‌های پشتیبانی میشود بهتر است از روش دوم یا ترکیبی از این دو روش استفاده کنید. خودم به شخصه سعی میکنم از روش اول استفاده کنم. چون معتقدم استفاده از ویوهای مختلف باعث افزایش بیش از اندازه حجم کار میشود. اما در برخی موارد استفاده از روش دوم یا ترکیبی از دو روش میتواند بهتر باشد.

تولید فایل‌های Resource

بهترین مکان برای نگهداری فایل‌های Resource در یک پروژه جداگانه است. در پروژه‌های از نوع وبسایت پوشه‌هایی با نام App_GlobalResources یا App_LocalResources وجود دارد که میتوان از آنها برای نگهداری و مدیریت این نوع فایل‌ها استفاده کرد. اما همانطور که در [اینجا](#) توضیح داده شده است این روش مناسب نیست. بنابراین ابتدا یک پروژه مخصوص نگهداری فایل‌های Resource ایجاد کنید و سپس اقدام به تهیه این فایل‌ها نمایید. سعی کنید که عنوان این پروژه به صورت زیر باشد. برای کسب اطلاعات بیشتر درباره نحوه نامگذاری اشیای مختلف در دات نت به [این مطلب](#) رجوع کنید.

SolutionName>.Resources>

برای افزودن فایل‌های Resource به این پروژه ابتدا برای انتخاب زبان پیش فرض محصول خود تصمیم بگیرید. پیشنهاد میکنم که از زبان انگلیسی (en-US) برای اینکار استفاده کنید. ابتدا یک فایل Resource (با پسوند .resx) مثلا با نام Texts.resx به این پروژه اضافه کنید. با افزودن این فایل به پروژه، ویژوال استودیو به صورت خودکار یک فایل cs حاوی کلاس متناظر با این فایل را به پروژه اضافه میکند. این کار توسط ابزار توکاری به نام ResXFileCodeGenerator انجام میشود. اگر به پراپرتی‌های این فایل .resx رجوع کنید میتوانید این عنوان را در پراپرتی Custom Tool ببینید. البته ابزار دیگری برای تولید این کلاسها نیز وجود دارد. این ابزارهای توکار برای سطوح دسترسی مختلف استفاده میشوند. ابزار پیش فرض در ویژوال استودیو یعنی همان ResXFileCodeGenerator، این کلاسها را با دسترسی internal تولید میکند که مناسب کار ما نیست. ابزار دیگری که برای اینکار درون ویژوال استودیو وجود دارد PublicResXFileCodeGenerator است و همانطور که از نامش پیداست از سطح دسترسی public استفاده میکند. برای تغییر این ابزار کافی است تا عنوان آن را دقیقاً در پراپرتی Custom Tool تایپ کنید.



نکته: درباره پراپرتی مهم Build Action این فایلها در مطالب بعدی بیشتر بحث میشود. برای تعیین سطح دسترسی Resource موردنظر به روشی دیگر، میتوانید فایل Resource را باز کرده و Access Modifier آن را به Public تغییر دهید.



سپس برای پشتیبانی از زبانی دیگر، یک فایل دیگر Resource به پروژه اضافه کنید. نام این فایل باید همانم فایل اصلی به همراه نام کالچر موردنظر باشد. مثلاً برای زبان فارسی عنوان فایل باید Texts.fa-IR.resx یا به صورت ساده‌تر برای کالچر خنثی (بدون نام کشور) Texts.fa.resx باشد. دقت کنید اگر نام فایل را در همان پنجره افزودن فایل وارد کنید ویژوال استودیو این همانمی را به صورت هوشمند تشخیص داده و تغییراتی را در پراپرتی‌های پیش فرض فایل Resource ایجاد میکند.

نکته: این هوشمندی مرتبه نسبتاً بالایی دارد. بدین صورت که تنها در صورتیکه عبارت بعد از نام فایل اصلی Resource (رشته بعد از نقطه مثلاً fa در اینجا) متعلق به یک کالچر معتبر باشد این تغییرات اعمال خواهد شد.

مهمترین این تغییرات این است که ابزاری را برای پراپرتی Custom Tool این فایلها انتخاب نمیکند! اگر به پراپرتی فایل Texts.fa.resx مراجعه کنید این مورد کاملاً مشخص است. در نتیجه دیگر فایل cs حاوی کلاسی جداگانه برای این فایل ساخته نمیشود. همچنین اگر فایل Resource جدید را باز کنید میبینید که برای Access Modifier آن گزینه No Code Generation انتخاب شده است.

در ادامه شروع به افزودن عناوین موردنظر در این دو فایل کنید. در اولی (بدون نام زبان) رشته‌های مربوط به زبان انگلیسی و در دومی رشته‌های مربوط به زبان فارسی را وارد کنید. سپس در هرجایی که یک لیبل یا یک رشته برای نمایش وجود دارد از این کلیدهای Resource استفاده کنید مثل:

```
SolutionName>.Resources.Texts.Save>
SolutionName>.Resources.Texts.Cancel>
```

استفاده از Resource در ویومدل ها

دو خاصیت معروفی که در ویومدلها استفاده میشوند عبارتند از: DisplayName و Required. پشتیبانی از کلیدهای Resource به صورت توکار در خاصیت Required وجود دارد. برای استفاده از آنها باید به صورت زیر عمل کرد:

```
[Required(ErrorMessageResourceName = "ResourceKeyName", ErrorMessageResourceType =
typeof(<SolutionName>.Resources.<ResourceClassName>))]
```

در کد بالا باید از نام فایل Resource اصلی (فایل اول که بدون نام کالچر بوده و به عنوان منبع پیشفرض به همراه یک فایل cs حاوی کلاس مربوطه نیز هست) برای معرفی ErrorMessageResourceType استفاده کرد. چون ابزار توکار ویژوال استودیو از نام این فایل برای تولید کلاس مربوطه استفاده میکند.

متأسفانه خاصیت DisplayName که در فضای نام System.ComponentModel (در فایل System.dll) قرار دارد قابلیت استفاده از کلیدهای Resource را به صورت توکار ندارد. در دات نت 4 خاصیت دیگری در فضای نام System.ComponentModel.DataAnnotations به نام Display (در فایل System.ComponentModel.DataAnnotations.dll) وجود دارد که این امکان را به صورت توکار دارد. اما قابلیت استفاده از این خاصیت تنها در MVC 3 وجود دارد. برای نسخه‌های قدیمتر

MVC امکان استفاده از این خاصیت حتی اگر نسخه فریمورک هدف 4 باشد وجود ندارد، چون هسته این نسخه‌های قدیمی امکان استفاده از ویژگی‌های جدید فریمورک با نسخه بالاتر را ندارد. برای رفع این مشکل میتوان کلاس خاصیت DisplayName را برای استفاده از خاصیت Display به صورت زیر توسعه داد:

```
public class LocalizationDisplayNameAttribute : DisplayNameAttribute
{
    private readonly DisplayAttribute _display;
    public LocalizationDisplayNameAttribute(string resourceName, Type resourceType)
    {
        _display = new DisplayAttribute { ResourceType = resourceType, Name = resourceName };
    }
    public override string DisplayName
    {
        get
        {
            try
            {
                return _display.GetName();
            }
            catch (Exception)
            {
                return _display.Name;
            }
        }
    }
}
```

در این کلاس با ترکیب دو خاصیت نامبرده امکان استفاده از کلیدهای Resource فراهم شده است. در پیاده سازی این کلاس فرض شده است که نسخه فریمورک هدف حداقل برابر 4 است. اگر از نسخه‌های پایین‌تر استفاده میکنید در پیاده سازی این کلاس باید کاملاً به صورت دستی کلید موردنظر را از Resource معرفی شده بدست آورید. مثلاً به صورت زیر:

```
public class LocalizationDisplayNameAttribute : DisplayNameAttribute
{
    private readonly PropertyInfo nameProperty;
    public LocalizationDisplayNameAttribute(string displayNameKey, Type resourceType = null)
        : base(displayNameKey)
    {
        if (resourceType != null)
            nameProperty = resourceType.GetProperty(base.DisplayName, BindingFlags.Static |
BindingFlags.Public);
    }
    public override string DisplayName
    {
        get
        {
            if (nameProperty == null) base.DisplayName;
            return (string)nameProperty.GetValue(nameProperty.DeclaringType, null);
        }
    }
}
```

برای استفاده از این خاصیت جدید میتوان به صورت زیر عمل کرد:

```
[LocalizationDisplayName("ResourceKeyName", typeof(<SolutionName>.Resources.<ResourceClassName>))]
```

البته بیشتر خواص متداول در ویومدلها از ویژگی موردبحث پشتیبانی میکنند.

نکته: به کارگیری این روش ممکن است در پروژه‌های بزرگ کمی گیج کننده و دردسرساز بوده و باعث پیچیدگی بی‌مورد کد و نیز افزایش بیش از حد حجم کدنویسی شود. در مقاله آقای فیل هک ([Model Metadata and Validation Localization using Conventions](#)) روش بهتر و تمیزتری برای مدیریت پیامهای این خاصیت‌ها آورده شده است.

پشتیبانی از ویژگی چند زبانه

مرحله بعدی برای چندزبانه کردن پروژه‌های MVC تغییراتی است که برای مدیریت Culture جاری برنامه باید پیاده شوند. برای

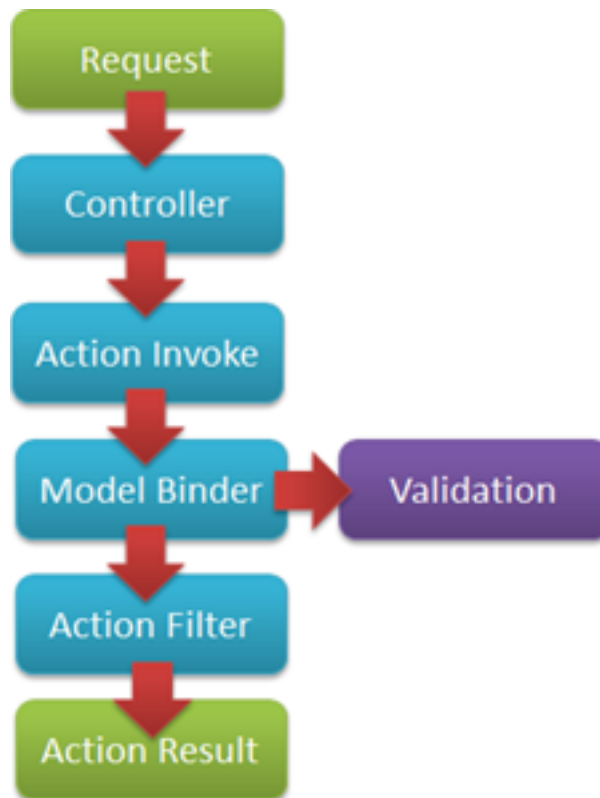
اینکار باید خاصیت `CurrentUICulture` در ثرد جاری کنترل و مدیریت شود. یکی از مکانهایی که برای نگهداری زبان جاری استفاده میشود کوکی است. معمولا برای اینکار از کوکی‌های دارای تاریخ انقضای طولانی استفاده میشود. میتوان از تنظیمات موجود در فایل کانفیگ برای ذخیره زبان پیش فرض سیستم نیز استفاده کرد. روشی که معمولا برای مدیریت زبان جاری میتوان از آن استفاده کرد پیاده سازی یک کلاس پایه برای تمام کنترلرها است. کد زیر راه حل نهایی را نشان میدهد:

```
public class BaseController : Controller
{
    private const string LanguageCookieName = "MyLanguageCookieName";
    protected override void ExecuteCore()
    {
        var cookie = HttpContext.Request.Cookies[LanguageCookieName];
        string lang;
        if (cookie != null)
        {
            lang = cookie.Value;
        }
        else
        {
            lang = ConfigurationManager.AppSettings["DefaultCulture"] ?? "fa-IR";
            var httpCookie = new HttpCookie(LanguageCookieName, lang) { Expires = DateTime.Now.AddYears(1) };
            HttpContext.Response.SetCookie(httpCookie);
        }
        Thread.CurrentThread.CurrentUICulture = CultureInfo.CreateSpecificCulture(lang);
        base.ExecuteCore();
    }
}
```

راه حل دیگر استفاده از یک `ActionFilter` است که نحوه پیاده سازی یک نمونه از آن در زیر آورده شده است:

```
public class LocalizationActionFilterAttribute : ActionFilterAttribute
{
    private const string LanguageCookieName = "MyLanguageCookieName";
    public override void OnActionExecuting(ActionExecutingContext filterContext)
    {
        var cookie = filterContext.HttpContext.Request.Cookies[LanguageCookieName];
        string lang;
        if (cookie != null)
        {
            lang = cookie.Value;
        }
        else
        {
            lang = ConfigurationManager.AppSettings["DefaultCulture"] ?? "fa-IR";
            var httpCookie = new HttpCookie(LanguageCookieName, lang) { Expires = DateTime.Now.AddYears(1) };
            filterContext.HttpContext.Response.SetCookie(httpCookie);
        }
        Thread.CurrentThread.CurrentUICulture = CultureInfo.CreateSpecificCulture(lang);
        base.OnActionExecuting(filterContext);
    }
}
```

نکته مهم: تعیین زبان جاری (یعنی همان مقداردهی پراپرتی `CurrentCulture` ثرد جاری) در یک اکشن فیلتر بدرستی عمل نمیکند. برای بررسی بیشتر این مسئله ابتدا به تصویر زیر که ترتیب رخ دادن رویدادهای مهم در ASP.NET MVC را نشان میدهد دقت کنید:



همانطور که در تصویر فوق مشاهده میکنید رویداد `OnActionExecuting` که در یک اکشن فیلتر به کار میرود بعد از عملیات مدل بایندینگ رخ میدهد. بنابراین قبل از تعیین کالچر جاری، عملیات `validation` و یافتن متن خطاها از فایل‌های `Resource` انجام میشود که منجر به انتخاب کلیدهای مربوط به کالچر پیشفرض سرور (و نه آنچه که کاربر تنظیم کرده) خواهد شد. بنابراین استفاده از یک اکشن فیلتر برای تعیین کالچر جاری مناسب نیست. راه حل مناسب استفاده از همان کنترلر پایه است، زیرا متد `ExecuteCore` قبل از تمامی این عملیات صدا زده میشود. بنابراین همیشه کالچر تنظیم شده توسط کاربر به عنوان مقدار جاری آن در ثرد ثبت میشود.

امکان تعیین/تغییر زبان توسط کاربر

برای تعیین یا تغییر زبان جاری سیستم نیز روشهای گوناگونی وجود دارد. استفاده از زبان تنظیم شده در مرورگر کاربر، استفاده از عنوان زبان در آدرس صفحات درخواستی و یا تعیین زبان توسط کاربر در تنظیمات برنامه/سایت و ذخیره آن در کوکی یا دیتابیس و مواردی از این دست روشهایی است که معمولاً برای تعیین زبان جاری از آن استفاده میشود. در کدهای نمونه ای که در بخشهای قبل آورده شده است فرض شده است که زبان جاری سیستم درون یک کوکی ذخیره میشود بنابراین برای استفاده از این روش میتوان از قطعه کدی مشابه زیر (مثلاً در فایل `_Layout.cshtml`) برای تعیین و تغییر زبان استفاده کرد:

```

<select id="langs" onchange="languageChanged()">
  <option value="fa-IR">فارسی</option>
  <option value="en-US">انگلیسی</option>
</select>
<script type="text/javascript">
  function languageChanged() {
    setCookie("MyLanguageCookieName", $('#langs').val(), 365);
    window.location.reload();
  }
  document.ready = function () {
    $('#langs').val(getCookie("MyLanguageCookieName"));
  };
  function setCookie(name, value, exdays, path) {
    var exdate = new Date();
    exdate.setDate(exdate.getDate() + exdays);
    var newValue = escape(value) + ((exdays == null) ? "" : "; expires=" + exdate.toUTCString()) +
    ((path == null) ? "" : "; path=" + path);
    document.cookie = name + "=" + newValue;
  }
</script>

```

```
function getCookie(name) {
    var i, x, y, cookies = document.cookie.split(";");
    for (i = 0; i < cookies.length; i++) {
        x = cookies[i].substr(0, cookies[i].indexOf("="));
        y = cookies[i].substr(cookies[i].indexOf("=") + 1);
        x = x.replace(/^\s+|\s+$/g, "");
        if (x == name) {
            return unescape(y);
        }
    }
}
</script>
```

متدهای `getCookie` و `setCookie` جاوا اسکریپتی در کد بالا از [اینجا](#) گرفته شده اند البته پس از کمی تغییر.

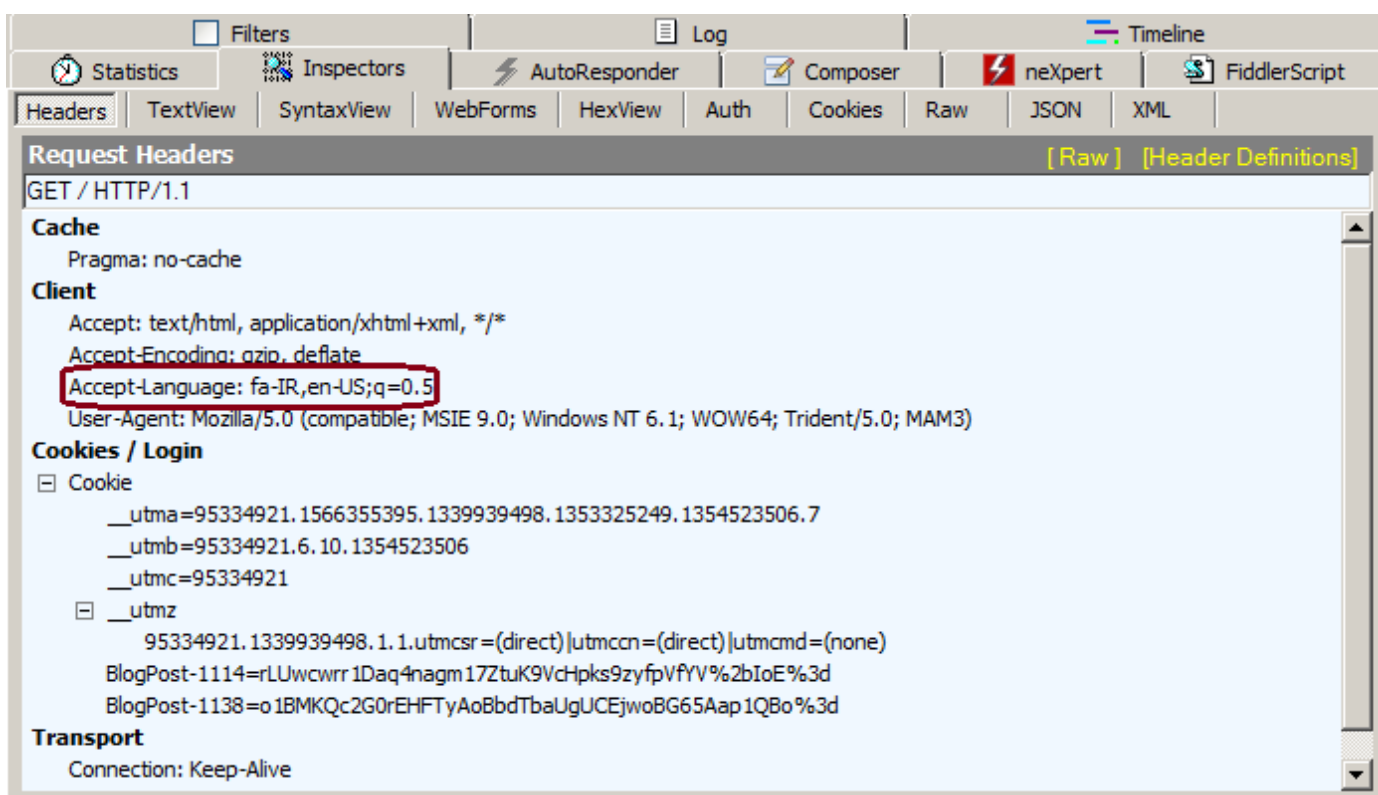
نکته : مطلب `Cookie` بحثی نسبتاً مفصل است که در جای خودش باید به صورت کامل آورده شود. اما در اینجا تنها به همین نکته اشاره کنم که عدم توجه به پراپرتی `path` کوکی‌ها در این مورد خاص برای خود من بسیار گیج‌کننده و دردسرساز بود.

به عنوان راهی دیگر میتوان به جای روش ساده استفاده از کوکی، تنظیماتی در اختیار کاربر قرار داد تا بتواند زبان تنظیم شده را درون یک فایل یا دیتابیس ذخیره کرد البته با در نظر گرفتن مسائل مربوط به کش کردن این تنظیمات.

راه حل بعدی میتواند استفاده از تنظیمات مرورگر کاربر برای دریافت زبان جاری تنظیم شده است. مرورگرها تنظیمات مربوط به زبان را در قسمت `Accept-Languages` در `HTTP Header` درخواست ارسالی به سمت سرور قرار میدهند. بصورت زیر:

```
GET http://www.dotnettips.info HTTP/1.1
...
Accept-Language: fa-IR,en-US;q=0.5
...
```

این هم تصویر مربوط به [Fiddler](#) آن:



نکته: پارامتر `q` در عبارت مشخص شده در تصویر فوق `relative quality factor` نام دارد و به نوعی مشخص کننده اولویت زبان مربوطه است. مقدار آن بین 0 و 1 است و مقدار پیش فرض آن 1 است. هرچه مقدار این پارامتر بیشتر باشد زبان مربوطه اولویت

بالاتری دارد. مثلاً عبارت زیر را در نظر بگیرید:

```
Accept-Language: fa-IR, fa;q=0.8,en-US;q=0.5,ar-BH;q=0.3
```

در این حالت اولویت زبان fa-IR برابر 1 و fa برابر 0.8 (fa;q=0.8) است. اولویت دیگر زبانهای تنظیم شده نیز همانطور که نشان داده شده است در مراتب بعدی قرار دارند. در تنظیم نمایش داده شده برای تغییر این تنظیمات در IE میتوان همانند تصویر زیر اقدام کرد:



در تصویر بالا زبان فارسی اولویت بالاتری نسبت به انگلیسی دارد. برای اینکه سیستم g11n دات نت به صورت خودکار از این مقادیر جهت زبان ثرد جاری استفاده کند میتوان از تنظیم زیر در فایل کانفیگ استفاده کرد:

```
<system.web>
  <globalization enableClientBasedCulture="true" uiCulture="auto" culture="auto"></globalization>
</system.web>
```

در سمت سرور نیز برای دریافت این مقادیر تنظیم شده در مرورگر کاربر میتوان از کدهای زیر استفاده کرد. مثلاً در یک اکشن فیلتر:

```
var langs = filterContext.HttpContext.Request.UserLanguages;
```

پراپرتی UserLanguages از کلاس Request حاوی آرایه‌ای از استرینگ است. این آرایه درواقع از Split کردن مقدار Accept-Languages با کاراکتر ',' بدست می‌آید. بنابراین اعضای این آرایه رشته‌ای از نام زبان به همراه پارامتر q مربوطه خواهند بود (مثل "fa;q=0.8").

راه دیگر مدیریت زبانها استفاده از عنوان زبان در مسیر درخواستی صفحات است. مثلاً آدرسی شبیه به www.MySite.com/fa/employees نشان میدهد کاربر درخواست نسخه فارسی از صفحه Employees را دارد. نحوه استفاده از این عناوین و نیز موقعیت فیزیکی این عناوین در مسیر صفحات درخواستی کاملاً به سلیقه برنامه نویس و یا کارفرما بستگی دارد. روش کلی بهره برداری از این روش در تمام موارد تقریباً یکسان است.

برای پیاده سازی این روش ابتدا باید یک route جدید در فایل Global.asax.cs اضافه کرد:

```
routes.MapRoute(
    "Localization", // Route name
    "{lang}/{controller}/{action}/{id}", // URL with parameters
    new { controller = "Home", action = "Index", id = UrlParameter.Optional } // Parameter defaults
);
```

دقت کنید که این route باید قبل از تمام route‌های دیگر ثبت شود. سپس باید کلاس پایه کنترلر را به صورت زیر پیاده سازی کرد:

```
public class BaseController : Controller
{
    protected override void ExecuteCore()
    {
        var lang = RouteData.Values["lang"];
        if (lang != null && !string.IsNullOrEmpty(lang.ToString()))
        {
            Thread.CurrentThread.CurrentUICulture = CultureInfo.CreateSpecificCulture(lang.ToString());
        }
        base.ExecuteCore();
    }
}
```

این کار را در یک اکشن فیلتر هم میتوان انجام داد اما با توجه به توضیحاتی که در قسمت قبل داده شد استفاده از اکشن فیلتر برای تعیین زبان جاری کار مناسبی نیست.

نکته: به دلیل آوردن عنوان زبان در مسیر درخواستها باید کنترلر دقیقتری بر کلیه مسیرهای موجود داشت!

استفاده از ویوهای جداگانه برای زبانهای مختلف

برای اینکار ابتدا ساختار مناسبی را برای نگهداری از ویوهای مختلف خود در نظر بگیرید. مثلاً میتوانید همانند نامگذاری فایل‌های Resource از نام زبان یا کالچر به عنوان بخشی از نام فایل‌های ویو استفاده کنید و تمام ویوها را در یک مسیر ذخیره کنید. همانند تصویر زیر:



البته اینکار ممکن است به مدیریت این فایلها را کمی مشکل کند چون به مرور زمان تعداد فایلهای ویو در یک فولدر زیاد خواهد شد. روش دیگری که برای نگهداری این ویوها میتوان به کار برد استفاده از فولدرهای جداگانه با عناوین زبانهای موردنظر است. مانند تصویر زیر:



روش دیگری که برای نگهداری و مدیریت بهتر ویوهای زبانهای مختلف از آن استفاده میشود به شکل زیر است:



استفاده از هرکدام از این روشها کاملاً به سلیقه و راحتی مدیریت فایلها برای برنامه نویس بستگی دارد. در هر صورت پس از

انتخاب یکی از این روشها باید اپلیکشن خود را طوری تنظیم کنیم که با توجه به زبان جاری سیستم، ویوی مربوطه را جهت نمایش انتخاب کند.

مثلا برای روش اول نامگذاری ویوها میتوان از روش دستکاری متد `OnActionExecuted` در کلاس پایه کنترلر استفاده کرد:

```
public class BaseController : Controller
{
    protected override void OnActionExecuted(ActionExecutedContext context)
    {
        var view = context.Result as ViewResultBase;
        if (view == null) return; // not a view
        var viewName = view.ViewName;
        view.ViewName = GetGlobalizationViewName(viewName, context);
        base.OnActionExecuted(context);
    }
    private static string GetGlobalizationViewName(string viewName, ControllerContext context)
    {
        var cultureName = Thread.CurrentThread.CurrentUICulture.Name;
        if (cultureName == "en-US") return viewName; // default culture
        if (string.IsNullOrEmpty(viewName))
            return context.RouteData.Values["action"] + "." + cultureName; // "Index.fa"
        int i;
        if ((i = viewName.IndexOf('.')) > 0) // ex: Index.cshtml
            return viewName.Substring(0, i + 1) + cultureName + viewName.Substring(i); // "Index.fa.cshtml"
        return viewName + "." + cultureName; // "Index" ==> "Index.fa"
    }
}
```

همانطور که قبلا نیز شرح داده شد، چون متد `ExecuteCore` قبل از `OnActionExecuted` صدا زده میشود بنابراین از تنظیم درست مقدار کالچر در ثرد جاری اطمینان داریم.

روش دیگری که برای مدیریت انتخاب ویوهای مناسب استفاده از یک ویوانجین شخصی سازی شده است. مثلا برای روش سوم نامگذاری ویوها میتوان از کد زیر استفاده کرد:

```
public sealed class RazorGlobalizationViewEngine : RazorViewEngine
{
    protected override IView CreatePartialView(ControllerContext controllerContext, string partialPath)
    {
        return base.CreatePartialView(controllerContext, GetGlobalizationViewPath(controllerContext, partialPath));
    }
    protected override IView CreateView(ControllerContext controllerContext, string viewPath, string masterPath)
    {
        return base.CreateView(controllerContext, GetGlobalizationViewPath(controllerContext, viewPath), masterPath);
    }
    private static string GetGlobalizationViewPath(ControllerContext controllerContext, string viewPath)
    {
        //var controllerName = controllerContext.RouteData.GetRequiredString("controller");
        var request = controllerContext.HttpContext.Request;
        var lang = request.Cookies["MyLanguageCookie"];
        if (lang != null && !string.IsNullOrEmpty(lang.Value) && lang.Value != "en-US")
        {
            var localizedViewPath = Regex.Replace(viewPath, "^~/Views/",
            string.Format("~/Views/Globalization/{0}/", lang.Value));
            if (File.Exists(request.MapPath(localizedViewPath))) viewPath = localizedViewPath;
        }
        return viewPath;
    }
}
```

و برای ثبت این ViewEngine در فایل `Global.asax.cs` خواهیم داشت:

```
protected void Application_Start()
{
    ViewEngines.Engines.Clear();
    ViewEngines.Engines.Add(new RazorGlobalizationViewEngine());
}
```

محتوای یک فایل Resource

ساختار یک فایل .resx به صورت XML استاندارد است. در زیر محتوای یک نمونه فایل Resource با پسوند .resx را مشاهده میکنید:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <!--
    Microsoft ResX Schema ...
  -->
  <xsd:schema id="root" xmlns="" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
    ...
  </xsd:schema>
  <resheader name="resmimetype">
    <value>text/microsoft-resx</value>
  </resheader>
  <resheader name="version">
    <value>2.0</value>
  </resheader>
  <resheader name="reader">
    <value>System.Resources.ResXResourceReader, System.Windows.Forms, Version=4.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089</value>
  </resheader>
  <resheader name="writer">
    <value>System.Resources.ResXResourceWriter, System.Windows.Forms, Version=4.0.0.0, Culture=neutral,
    PublicKeyToken=b77a5c561934e089</value>
  </resheader>
  <data name="RightToLeft" xml:space="preserve">
    <value>>false</value>
    <comment>RightToLeft is false in English!</comment>
  </data>
</root>
```

در قسمت ابتدایی تمام فایل‌های .resx که توسط ویژوال استودیو تولید میشود کامنتی طولانی وجود دارد که به صورت خلاصه به شرح محتوا و ساختار یک فایل Resource میپردازد. در ادامه تگ نسبتاً طولانی xsd:schema قرار دارد. از این قسمت برای معرفی ساختار داده‌ای فایل‌های XML استفاده میشود. برای آشنایی بیشتر با XSD (یا XML Schema) به [اینجا](#) مراجعه کنید. به صورت خلاصه میتوان گفت که XSD برای تعیین ساختار داده‌ها یا تعیین نوع داده‌ای اطلاعات موجود در یک فایل XML به کار میرود. درواقع تگهای XSD به نوعی فایل XML ما را Strongly Typed میکند. با توجه به اطلاعات این قسمت، فایل‌های .resx شامل 4 نوع گره اصلی هستند که عبارتند از: metadata و assembly و data و resheader. در تعریف هر یک از گره‌ها در این قسمت مشخصاتی چون نام زیر

گره‌های قابل تعریف در هر گره و نام و نوع خاصیت‌های هر یک معرفی شده است. بخش موردنظر ما در این مطلب قسمت انتهایی این فایل‌هاست (تگهای resheader و data). همانطور در بالا مشاهده میکنید تگهای reheader شامل تنظیمات مربوط به فایل .resx با ساختاری ساده به صورت name/value است. یکی از این تنظیمات resmimetype فایل resource را معرفی میکند که درواقع مشخص کننده نوع محتوای (Content Type) فایل XML است ([^](#)). برای فایل‌های .resx این مقدار برابر text/microsoft-resx است. تنظیم بعدی نسخه مربوط به فایل .resx (یا Microsoft ResX Schema) را نشان میدهد. در حال حاضر نسخه جاری (در VS 2010) برابر 2.0 است. تنظیم بعدی مربوط به کلاسهای reader و writer تعریف شده برای استفاده از این فایل‌هاست. به نوع این کلاسهای خواننده و نویسنده فایل‌های .resx و مکان فیزیکی و فضای نام آنها دقت کنید که در مطالب بعدی از آنها برای ویرایش و بروزرسانی فایل‌های resource در زمان اجرا استفاده خواهیم کرد.

در پایان نیز تگهای data که برای نگهداری داده‌ها از آنها استفاده میشود. هر گره data شامل یک خاصیت نام (name) و یک زیرگره مقدار (value) است. البته امکان تعیین یک کامنت در زیرگره comment نیز وجود دارد که اختیاری است. هر گره data میتواند شامل خاصیت type و یا mimetype نیز باشد. خاصیت type مشخص کننده نوعی است که تبدیل text/value را با استفاده از ساختار [TypeConverter](#) پشتیبانی میکند. البته اگر در نوع مشخص شده این پشتیبانی وجود نداشته باشد، داده موردنظر پس از سریالایز شدن با فرمت مشخص شده در خاصیت mimetype ذخیره میشود. این mimetype اطلاعات موردنیاز را برای کلاس خواننده این فایل‌ها (ResXResourceReader به صورت پیشفرض) جهت چگونگی بازیابی آبجکت موردنظر فراهم میکند. مشخص کردن این دو خاصیت برای انواع رشته‌ای نیاز نیست. انواع mimetype قابل استفاده عبارتند از:

- application/x-microsoft.net.object.binary.base64: آبجکت موردنظر باید با استفاده از کلاس

System.Runtime.Serialization.Formatters.Binary.BinaryFormatter سریالایز شده و سپس با فرمت base64 به یک رشته

انکد شود (راجع به انکدینگ base64 و [^](#)).

- application/x-microsoft.net.object.soap.base64: آبجکت موردنظر باید با استفاده از کلاس

شود. `System.Runtime.Serialization.Formatters.Soap.SoapFormatter` سریالایز شده و سپس با فرمت base64 به یک رشته انکد

- application/x-microsoft.net.object.bytearray.base64: آبجکت ابتدا باید با استفاده از یک `System.ComponentModel.TypeConverter` به آرایه ای از بایت سریالایز شده و سپس با فرمت base64 به یک رشته انکد شود. **نکته**: امکان جاسازی کردن (embed) فایل‌های resx. در یک اسمبلی یا کامپایل مستقیم آن به یک سَتلایت اسمبلی (ترجمه مناسبی برای [satellite assembly](#) پیدا نکردم، چیزی شبیه به اسمبلی قمری یا وابسته و از این قبیل ...) وجود ندارد. ابتدا باید این فایل‌های resx. به فایل‌های resources. تبدیل شوند. اینکار با استفاده از ابزار Resource File Generator (نام فایل اجرایی آن resgen.exe است) انجام میشود ([^](#) و [^](#)). سپس میتوان با استفاده از Assembly Linker ستلایت اسمبلی مربوطه را تولید کرد ([^](#)). کل این عملیات در ویژوال استودیو با استفاده از ابزار msbuild به صورت خودکار انجام میشود!

نحوه یافتن کلیدهای Resource در بین فایل‌های مختلف Resx توسط پرووایدر پیش فرض در دات نت

عملیات ابتدا با بررسی خاصیت `CurrentUICulture` از ثرد جاری آغاز میشود. سپس با استفاده از عنوان استاندارد کالچر جاری، فایل مناسب Resource یافته میشود. در نهایت بهترین گزینه موجود برای کلید درخواستی از منابع موجود انتخاب میشود. مثلاً اگر کالچر جاری fa-IR و کلید درخواستی از کلاس Texts باشد ابتدا جستجو برای یافتن فایل Texts.fa-IR.resx آغاز میشود و اگر فایل موردنظر یا کلید درخواستی در این فایل یافته نشد جستجو در فایل Texts.fa.resx ادامه می‌یابد. اگر باز هم یافته نشد در نهایت این عملیات جستجو در فایل resource اصلی خاتمه می‌یابد و مقدار کلید منبع پیش فرض به عنوان نتیجه برگشت داده میشود. یعنی در تمامی حالات سعی میشود تا دقیقترین و بهترین و نزدیکترین نتیجه انتخاب شود. البته در صورتیکه از یک پرووایدر شخصی سازی شده برای کار خود استفاده میکنید باید چنین الگوریتمی را جهت یافتن کلیدهای منابع خود از فایل‌های Resource (یا هر منبع دیگر مثل دیتابیس یا حتی یک وب سرویس) در نظر بگیرید.

Globalization در کلاینت (javascript g11n)

یکی دیگر از موارد استفاده g11n در برنامه نویسی سمت کلاینت است. با وجود استفاده گسترده از جاوا اسکریپت در برنامه نویسی سمت کلاینت در وب اپلیکیشن‌ها، متأسفانه تا همین اواخر عملاً ابزار یا کتابخانه مناسبی برای مدیریت g11n در این زمینه وجود نداشته است. یکی از اولین کتابخانه‌های تولید شده در این زمینه کتابخانه jQuery Globalization است که توسط مایکروسافت توسعه داده شده است (برای آشنایی بیشتر با این کتابخانه به [^](#) و [^](#) مراجعه کنید). این کتابخانه بعداً تغییر نام داده و اکنون با عنوان Globalize شناخته میشود. Globalize یک کتابخانه کاملاً مستقل است که وابستگی به هیچ کتابخانه دیگر ندارد (یعنی برای استفاده از آن نیازی به jQuery نیست). این کتابخانه حاوی کالچرهای بسیاری است که عملیات مختلفی چون فرمت و parse انواع داده‌ها را نیز در سمت کلاینت مدیریت میکند. همچنین با فراهم کردن منابعی حاوی جفت‌های key/culture میتوان از مزایایی مشابه مواردی که در این مطلب بحث شد در سمت کلاینت نیز بهره برد. نشانی این کتابخانه در github [اینجا](#) است. با اینکه خود این کتابخانه ابزار کاملی است اما در بین کالچرهای موجود در فایل‌های آن متأسفانه پشتیبانی کاملی از زبان فارسی نشده است. ابزار دیگری که برای اینکار وجود دارد پلاگین [jquery localize](#) است که برای بحث g11n رشته‌ها پیاده‌سازی بهتر و کاملتری دارد.

در مطالب بعدی به مباحث تغییر مقادیر کلیدهای فایل‌های resource در هنگام اجرا با استفاده از روش مستقیم تغییر محتوای فایل‌ها و کامپایل دوباره توسط ابزار msbuild و نیز استفاده از یک ResourceProvider شخصی سازی شده به عنوان یک راه حل بهتر برای اینکار میپردازم.

در تهیه این مطلب از منابع زیر استفاده شده است: [Localization in ASP.NET MVC – 3 Days Investigation, 1 Day Job](#)

[ASP.NET MVC 3 Internationalization](#)

[Localization and skinning in ASP.NET MVC 3 web applications](#) [Simple ASP.Net MVC Globalization with Graceful](#)

[Fallback](#)

[Globalization, Internationalization and Localization in ASP.NET MVC 3, JavaScript and jQuery - Part 1](#)

نظرات خوانندگان

نویسنده: امیرحسین مرجانی
تاریخ: ۲۳:۵ ۱۳۹۱/۱۰/۲۱

سلام آقای یوسف نژاد
من بعد از تلاش‌های زیاد توی پروژه‌های مختلف این مطالبی که شما نوشته اید رو پیاده سازی کردم ، ولی خیلی پراکنده.
ولی حالا می‌بینم شما به زیبایی این مطالب رو کنار هم قرار دادید.
می‌خواستم بابت مطلب خوب و مفیدتون و همچنین وقتی که گذاشتید تشکر کنم.
ممنونم بابت زحمات شما

اگر ممکنه برچسب MVC رو هم به مطلبتون اضافه کنید.

نویسنده: یوسف نژاد
تاریخ: ۲۳:۲۴ ۱۳۹۱/۱۰/۲۱

با سلام و تشکر بابت نظر لطف شما.
البته باید بگم که همه دوستانی که اینجا به عنوان نویسنده کمک میکنند هدفشون اشتراک مطالبی هست که یاد گرفته اند تا سایر دوستان هم استفاده کنند.

برچسب MVC هم اضافه شد. با تشکر از دقت نظر شما.

نویسنده: امیرحسین جلوداری
تاریخ: ۱:۳ ۱۳۹۱/۱۰/۲۲

کاملا مشخصه که مطلب از روی تجربه‌ی کاریه و بسیار عالی جمع آوری شده ... ممنون ... به طرز عجیبی منتظر قسمت بعدم :دی

نویسنده: پندار
تاریخ: ۲۱:۳۸ ۱۳۹۱/۱۲/۰۸

گویا در MVC 4 این روش پاسخ نمیدهد. لطفا در این مورد برای MVC 4 راه حلی بدهید

نویسنده: محسن
تاریخ: ۲۳:۶ ۱۳۹۱/۱۲/۰۸

MVC 4 فقط یک سری افزونه بیشتر از MVC3 داره. مثلا razor آن بهبود پیدا کرده، فشرده سازی فایل‌های CSS به اون اضافه شده یا Web API رو به صورت یکپارچه داره. از لحاظ کار با فایل‌های منبع فرقی نکرده.

نویسنده: پندار
تاریخ: ۹:۲۱ ۱۳۹۱/۱۲/۰۹

متن نشانی زیر را مطالعه کنید

<http://geekswithblogs.net/shaunxu/archive/2012/09/04/localization-in-asp.net-mvc-ndash-upgraded.aspx>

نویسنده: محسن
تاریخ: ۹:۴۳ ۱۳۹۱/۱۲/۰۹

مطلبی که لینک دادی در مورد آپدیت یک helper شخصی توسعه داده شده توسط شخص ثالث است از MVC2 به MVC4. اگر کسی

از این راه حل شخصی و خاص استفاده نکرده باشه، اصول فوق فرقی نکرده.

نویسنده: صابر فتح الهی
تاریخ: ۱۶:۵ ۱۳۹۱/۱۲/۱۴

مطلب خیلی خوبی بود کلی استفاده کردیم.
مهندس کالچر زبان کردی چی میشه؟ توی لیست منابعی که دادین گیر نیاوردم

نویسنده: وحید نصیری
تاریخ: ۱۷:۲۲ ۱۳۹۱/۱۲/۱۴

kur هست [مطابق استاندارد](#) .

نویسنده: صابر فتح الهی
تاریخ: ۲:۱۱ ۱۳۹۱/۱۲/۱۷

سلام
اما مهندس کلاس Culture Info این مقدار قبول نمی‌کنه

نویسنده: وحید نصیری
تاریخ: ۹:۳ ۱۳۹۱/۱۲/۱۷

می‌تونید کلاس [فرهنگ سفارشی](#) را ایجاد و [استفاده](#) کنید.

نویسنده: صابر فتح الهی
تاریخ: ۱۰:۱۲ ۱۳۹۱/۱۲/۱۷

اما روش گفته شده نیاز به دسترسی مدیریت دارد که روی سرورهای اشتراکی ممکن نیست

نویسنده: وحید نصیری
تاریخ: ۱۱:۱۹ ۱۳۹۱/۱۲/۱۷

نحوه توسعه اکثر برنامه‌ها و کتابخانه‌ها در طول زمان، بر اساس تقاضا و پیگیری مصرف کننده است. اگر بعد از بیش از 10 سال، چنین فرهنگی اضافه نشده یعنی درخواستی نداشته. مراجعه کنید به [محل پیگیری این نوع مسایل](#) .

نویسنده: صابر فتح الهی
تاریخ: ۱۰:۱۴ ۱۳۹۱/۱۲/۱۹

سلام مهندس یوسف نژاد (ابتدا ممنونم از پست خوب شما)
با پیروی از پست شما
ابتدا فایل‌های ریسورس در پروژه جاری فولدر App_GlobalResources گذاشتم و پروژه در صفحات aspx با قالب زیر به راحتی
تغییر زبان داده میشد:

```
<asp:Literal ID="Literal1" Text='<%%$ Resources:resource, Title %>' runat="server" />
```

اما بعدش فایل هارو توی یک پروژه کتابخانه ای جدید گذاشتم و Build Action فایل‌های ریسورس روی Embedded Resource
تنظیم کردم، پروژه با موفقیت اجرا شد و در سمت سرور با کد زیر راحت به مقادیر دسترسی دارم:

```
Literal1.Text=ResourceManager.Resource.Title;
```

اما در سمت صفحات aspx با کد قبلی به شکل زیر نمایش نمیده و خطا صادر میشه:

```
<asp:Literal ID="Literal1" runat="server" Text='<%= $ResourcesManager.Resource:resource, Title %>' />
```

و خطای زیر صادر میشه:

Parser Error

Description: An error occurred during the parsing of a resource required to service this request. Please review the following specific parse error details and modify your source file appropriately.

Parser Error Message: The expression prefix 'ResourcesManager.Resource' was not recognized. Please correct the prefix or register the prefix in the <expressionBuilders> section of configuration.

Source Error:

مراحل این [یست](#) روی هم دنبال کردم اما باز نمشد.
چه تنظیماتی ست نکردم ؟

نویسنده: یوسف نژاد
تاریخ: ۱۳۹۲/۰۱/۳۱ ۱۲:۴۴

ببخشید یه چند وقتی فعال نبودم و پاسخ این سوال رو دیر دارم میدم.
امکان استفاده از کلیدهای Resource برای مقداردهی خواص سمت سرور کنترلها در صفحات aspx به صورت مستقیم وجود ندارد. بنابراین برای استفاده از این کلیدها همانند روش پیش فرض موجود در ASP.NET باید از یکسری ExpressionBuilder استفاده شود که کار Parse عبارت وارده برای این خواص را در سمت سرور انجام میدهد. کلاس پیش فرض برای اینکار در ASP.NET Web Form که از پیشوند Resources استفاده میکند تنها برای Resourceهای محلی (Local) موجود در فولدرهای پیش فرض (App_GlobalResources و App_LocalResources) کاربرد دارد و برای استفاده از Resourceهای موجود در منابع ریفرنس داده شده به پروژه باید از روشی مثل اونچه که خود شما لینکش رو دادین استفاده کرد.
من این روش رو استفاده کردم و پیاده سازی موفق داشتم. نمیدونم مشکل شما چیه...

نویسنده: یوسف نژاد
تاریخ: ۱۳۹۲/۰۱/۳۱ ۱۲:۵۲

اگر مشکلی در پیاده سازی روش بالا دارین، تمام مراحل که من طی کردم دقیقا اینجا میارم:
ابتدا کلاس ExpressionBuilder رو به صورت زیر مثلا در خود پروژه Resources اضافه میکنیم:

```
using System.Web.Compilation;
using System.CodeDom;
namespace Resources
{
    [ExpressionPrefix("MyResource")]
    public class ResourceExpressionBuilder : ExpressionBuilder
    {
        public override System.CodeDom.CodeExpression GetCodeExpression(System.Web.UI.BindPropertyEntry entry, object parsedData, System.Web.Compilation.ExpressionBuilderContext context)
        {
            return new CodeSnippetExpression(entry.Expression);
        }
    }
}
```

سپس تنظیمات زیر رو به Web.config اضافه میکنیم:

```
<compilation debug="true" targetFramework="4.0">
  <expressionBuilders>
    <add expressionPrefix="MyResource" type="Resources.ResourceExpressionBuilder, Resources" />
  </expressionBuilders>
</compilation>
```

```
</expressionBuilders>
</compilation>
```

در نهایت به صورت زیر میتوان از این کلاس استفاده کرد:

```
<asp:Literal ID="Literal1" runat="server" Text="<%= MyResource: Resources.Resource1.String2 %>" />
```

هرچند ظاهراً مقدار پیشوند معرفی شده در Attribute کلاس ResourceExpressionBuilder اهمیت چندانی ندارد! امیدوارم مشکلتون حل بشه.

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۲/۰۲/۰۱ ۲:۲۹

ممنونم از پاسخ شما
همون روش شمارو دنبال کردم پاسخ گرفتم، اشکال از خودم بود
با تشکر از شما

نویسنده: صادق نجاتی
تاریخ: ۱۳۹۲/۱۲/۲۷ ۱۲:۰۰

با سلام
ضمن تشکر از مطلب بسیار خوبتون
خاصیت DisplayFormat قابلیت استفاده از کلیدهای Resource را ندارد !
لطفا راهنمایی فرمایید که چطور میشه از این خاصیت برای DisplayFormat استفاده کرد؟
من می‌خواهم برای تاریخ در زبانه فارسی از فرمت {yyyy-MM-dd} و در زبانه انگلیسی از {yyyy-dd-MM} استفاده کنم.
با سپاس فراوان

به‌روزرسانی فایل‌های Resource در زمان اجرا یکی از ویژگی‌های مهمی که در پیاده‌سازی محصول با استفاده از فایل‌های Resource باید به آن توجه داشت، امکان بروز رسانی محتوای این فایل‌ها در زمان اجراست. از آنجا که احتمال اینکه کاربران سیستم خواهان تغییر این مقادیر باشند بسیار زیاد است، بنابراین در نظر گرفتن چنین ویژگی‌ای برای محصول نهایی می‌تواند بسیار تعیین‌کننده باشد. متأسفانه پیاده‌سازی چنین امکانی درباره فایل‌های Resource چندان آسان نیست. زیرا این فایل‌ها همانطور که در قسمت [قبل](#) توضیح داده شد پس از کامپایل به صورت اسمبلی‌های ستلایت (Satellite Assembly) درآمده و دیگر امکان تغییر محتوای آنها بصورت مستقیم و به آسانی وجود ندارد.

نکته: البته نحوه پیاده‌سازی این فایل‌ها در اسمبلی نهایی (و در حالت کلی نحوه استفاده از هر فایلی در اسمبلی نهایی) در ویژوال استودیو توسط خاصیت Build Action تعیین می‌شود. برای کسب اطلاعات بیشتر راجع به این خاصیت به [اینجا](#) رجوع کنید.

یکی از روش‌های نسبتاً مندرآوردی که برای ویرایش و به‌روزرسانی کلیدهای Resource وجود دارد بدین صورت است:
- ابتدا باید اصل فایل‌های Resource به همراه پروژه پابلیش شود. بهترین مکان برای نگهداری این فایل‌ها فولدر App_Data است. زیرا محتویات این فولدر توسط سیستم FCN (همان *File Change Notification*) در ASP.NET رصد می‌شود.

نکته: علت این حساسیت این است که FCN در ASP.NET تقریباً تمام محتویات فولدر سایت در سرور (فولدر App_Data یکی از معدود استثناهاست) را تحت نظر دارد و رفتار پیشفرض این است که با هر تغییری در این محتویات، AppDomain سایت Unload می‌شود که پس از اولین درخواست دوباره Load می‌شود. این اتفاق موجب از دست دادن تمام سشن‌ها و محتوای کش‌ها و ... می‌شود (اطلاعات بیشتر و کاملتر درباره نحوه رفتار FCN در [اینجا](#)).

- سپس با استفاده یک مقدار کدنویسی امکاناتی برای ویرایش محتوای این فایل‌ها فراهم شود. از آنجا که محتوای این فایل‌ها به صورت XML ذخیره می‌شود بنابراین براحتی می‌توان با امکانات موجود این ویژگی را پیاده‌سازی کرد. اما در فضای نام System.Windows.Forms کلاس‌هایی وجود دارد که مخصوص کار با این فایل‌ها طراحی شده‌اند که کار نمایش و ویرایش محتوای فایل‌های Resource را ساده‌تر می‌کند. به این کلاس‌ها در قسمت [قبلی](#) اشاره کوتاهی شده بود.

- پس از ویرایش و به‌روزرسانی محتوای این فایل‌ها باید کاری کنیم تا برنامه از این محتوای تغییر یافته به عنوان منبع جدید بهره بگیرد. اگر از این فایل‌های Resource به صورت embed استفاده شده باشد در هنگام build پروژه محتوای این فایل‌ها به صورت Satellite Assembly در کنار کتابخانه‌های دیگر تولید می‌شود. اسمبلی مربوط به هر زبان هم در فولدری با عنوان زبان مربوطه ذخیره می‌شود. مسیر و نام فایل این اسمبلی‌ها مثلاً به صورت زیر است:

bin\fa\Resources.resources.dll

بنابراین در این روش برای استفاده از محتوای به‌روز رسانی شده باید عملیات Build این کتابخانه دوباره انجام شود و کتابخانه‌های جدیدی تولید شود. راه حل اولی که به ذهن می‌رسد این است که از ابزارهای پایه و اصلی برای تولید این کتابخانه‌ها استفاده شود. این ابزارها (همانطور که در قسمت [قبل](#) نیز توضیح داده شد) عبارتند از Resource Generator و Assembly Linker. اما استفاده از این ابزارها و پیاده‌سازی روش مربوطه سخت‌تر از آن است که به نظر می‌آید. خوشبختانه درون مجموعه عظیم دات نت ابزار مناسب‌تری برای این کار نیز وجود دارد که کار تولید کتابخانه‌های موردنظر را به سادگی انجام می‌دهد. این ابزار با عنوان Microsoft Build شناخته می‌شود که در [اینجا](#) توضیح داده شده است.

خواندن محتویات یک فایل resx.

همانطور که در بالا توضیح داده شد برای راحتی کار می‌توان از کلاس زیر که در فایل System.Windows.Forms.dll قرار دارد استفاده کرد:

System.Resources.ResXResourceReader

این کلاس چندین کانستراکتور دارد که مسیر فایل resx. یا استریم مربوطه به همراه چند گزینه دیگر را به عنوان ورودی میگیرد. این کلاس یک Enumerator دارد که یک شی از نوع IDictionaryEnumerator برمیگرداند. هر عضو این enumerator از نوع object است. برای استفاده از این اعضا ابتدا باید آنرا به نوع DictionaryEntry تبدیل کرد. مثلاً بصورت زیر:

```
private void TestResXResourceReader()
{
    using (var reader = new ResXResourceReader("Resource1.fa.resx"))
    {
        foreach (var item in reader)
        {
            var resource = (DictionaryEntry)item;
            Console.WriteLine("{0}: {1}", resource.Key, resource.Value);
        }
    }
}
```

همانطور که ملاحظه میکنید استفاده از این کلاس بسیار ساده است. از آنجاکه DictionaryEntry یک struct است، به عنوان یک راه حل مناسبتر بهتر است ابتدا کلاسی به صورت زیر تعریف شود:

```
public class ResXResourceEntry
{
    public string Key { get; set; }
    public string Value { get; set; }
    public ResXResourceEntry() { }
    public ResXResourceEntry(object key, object value)
    {
        Key = key.ToString();
        Value = value.ToString();
    }
    public ResXResourceEntry(DictionaryEntry dictionaryEntry)
    {
        Key = dictionaryEntry.Key.ToString();
        Value = dictionaryEntry.Value != null ? dictionaryEntry.Value.ToString() : string.Empty;
    }
    public DictionaryEntry ToDictionaryEntry()
    {
        return new DictionaryEntry(Key, Value);
    }
}
```

سپس با استفاده از این کلاس خواهیم داشت:

```
private static List<ResXResourceEntry> Read(string filePath)
{
    using (var reader = new ResXResourceReader(filePath))
    {
        return reader.Cast<object>().Cast<DictionaryEntry>().Select(de => new
ResXResourceEntry(de)).ToList();
    }
}
```

حال این متد برای استفاده‌های آتی آماده است.

نوشتن در فایل resx.

برای نوشتن در یک فایل resx. میتوان از کلاس ResXResourceWriter استفاده کرد. این کلاس نیز در کتابخانه System.Windows.Forms در فایل System.Windows.Forms.dll قرار دارد:

System.Resources.ResXResourceWriter

متأسفانه در این کلاس امکان افزودن یا ویرایش یک کلید به تنهایی وجود ندارد. بنابراین برای ویرایش یا اضافه کردن حتی یک کلید کل فایل باید دوباره تولید شود. برای استفاده از این کلاس نیز میتوان به شکل زیر عمل کرد:

```
private static void Write(IEnumerable<ResXResourceEntry> resources, string filePath)
{
    using (var writer = new ResXResourceWriter(filePath))
    {
```

```

        foreach (var resource in resources)
        {
            writer.AddResource(resource.Key, resource.Value);
        }
    }
}

```

در متد فوق از همان کلاس ResXResourceEntry که در قسمت قبل معرفی شد، استفاده شده است. از متد زیر نیز میتوان برای حالت کلی حذف یا ویرایش استفاده کرد:

```

private static void AddOrUpdate(ResXResourceEntry resource, string filePath)
{
    var list = Read(filePath);
    var entry = list.SingleOrDefault(l => l.Key == resource.Key);
    if (entry == null)
    {
        list.Add(resource);
    }
    else
    {
        entry.Value = resource.Value;
    }
    Write(list, filePath);
}

```

در این متد از متدهای Read و Write که در بالا نشان داده شده‌اند استفاده شده است.

حذف یک کلید در فایل .resx

برای اینکار میتوان از متد زیر استفاده کرد:

```

private static void Remove(string key, string filePath)
{
    var list = Read(filePath);
    list.RemoveAll(l => l.Key == key);
    Write(list, filePath);
}

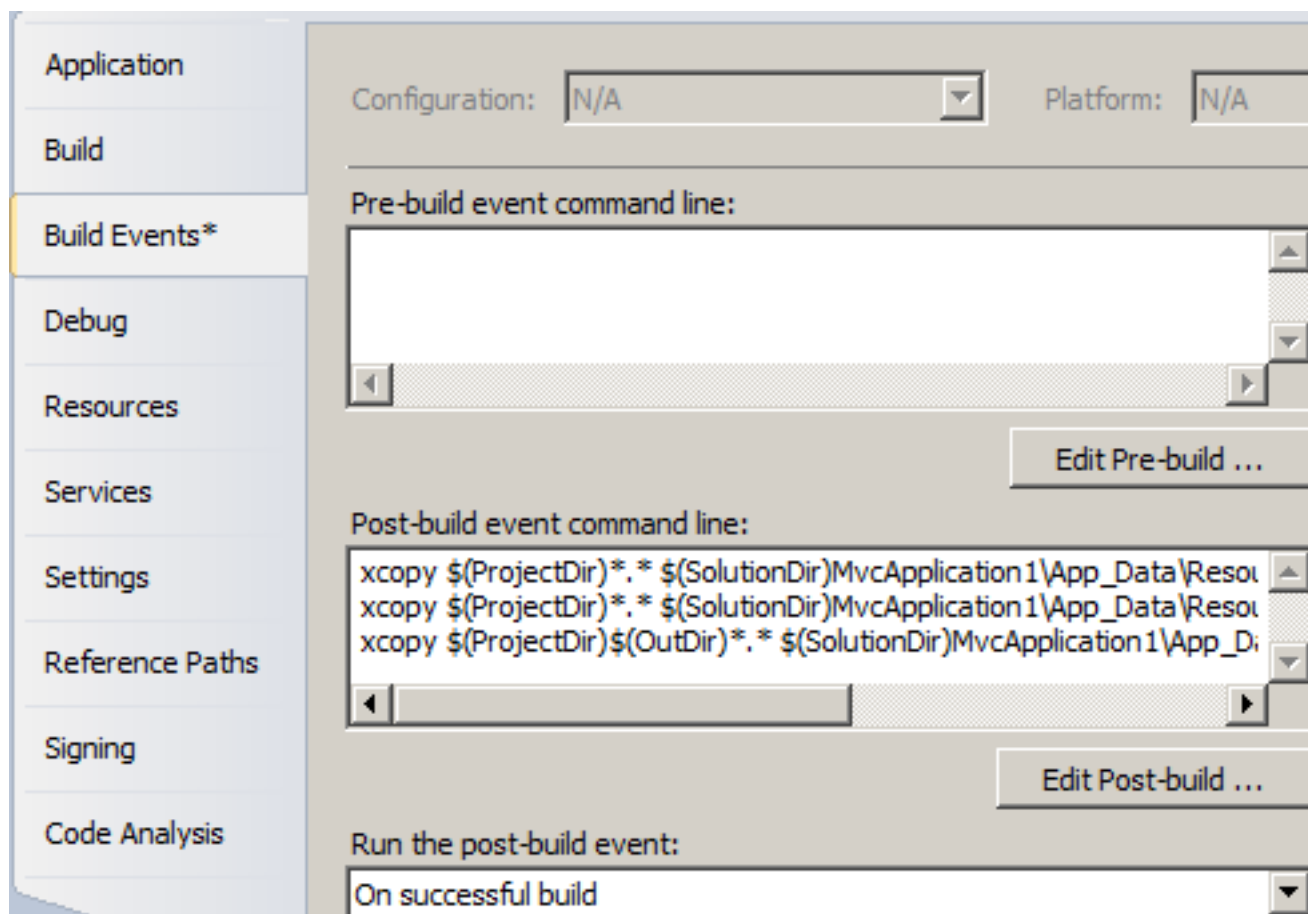
```

در این متد، از متد Write که در قسمت معرفی شد، استفاده شده است.

راه حل نهایی

قبل از بکارگیری روشهای معرفی شده در این مطلب بهتر است ابتدا یکسری قرارداد بصورت زیر تعریف شوند:

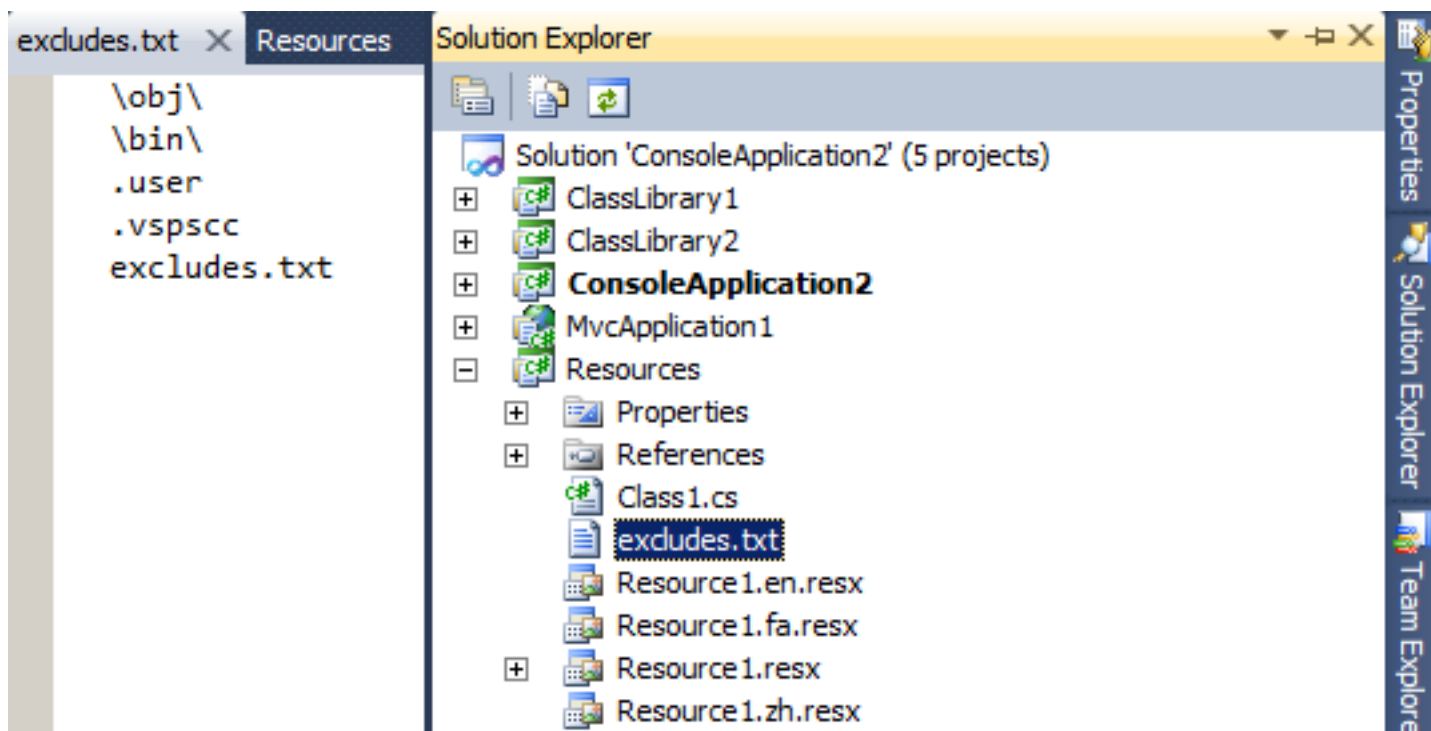
- طبق راهنماییهای موجود در قسمت [قبل](#) یک پروژه جداگانه با عنوان Resources برای نگهداری فایل‌های .resx ایجاد شود.
- همواره آخرین نسخه از محتویات موردنیاز از پروژه Resources باید درون فولدری با عنوان Resources در پوشه App_Data قرار داشته باشد.
- آخرین نسخه تولیدی از محتویات موردنیاز پروژه Resource در فولدری با عنوان Defaults در مسیر App_Data\Resources برای فراهم کردن امکان "بازگرداندن به تنظیمات اولیه" وجود داشته باشد.
- برای فراهم کردن این موارد بهترین راه حل استفاده از تنظیمات Post-build event command line است. اطلاعات بیشتر درباره Build Event ها در [اینجا](#) .



برای اینکار من از دستور xcopy استفاده کردم که نسخه توسعه یافته دستور copy است. دستورات استفاده شده در این قسمت عبارتند از:

```
xcopy $(ProjectDir)*.* $(SolutionDir)MvcApplication1\App_Data\Resources /e /y /i
/exclude:$(ProjectDir)excludes.txt
xcopy $(ProjectDir)*.* $(SolutionDir)MvcApplication1\App_Data\Resources\Defaults /e /y /i
/exclude:$(ProjectDir)excludes.txt
xcopy $(ProjectDir)$(OutDir)*.* $(SolutionDir)MvcApplication1\App_Data\Resources\Defaults\bin /e /y /i
```

در دستورات فوق آرگومان /e برای کپی تمام فولدرها و زیرفولدرها، /y برای تایید تمام کانفیرم ها، و /i برای ایجاد خودکار فولدرهای موردنیاز استفاده میشود. آرگومان /exclude نیز همانطور که از نامش پیداست برای خارج کردن فایلها و فولدرهای موردنظر از لیست کپی استفاده میشود. این آرگومان مسیر یک فایل متنی حاوی لیست این فایلها را دریافت میکند. در تصویر زیر یک نمونه از این فایل و مسیر و محتوای مناسب آن را مشاهده میکنید:



با استفاده از این فایل excludes.txt فولدرهای bin و obj و نیز فایل‌های با پسوند .user و .vspsc (مربوط به TFS) و نیز خود فایل excludes.txt از لیست کپی دستور xcopy حذف میشوند و بنابراین کپی نمیشوند. در صورت نیاز میتوانید گزینه‌های دیگری نیز به این فایل اضافه کنید.

همانطور که در [اینجا](#) اشاره شده است، در تنظیمات Post-build event command line یکسری متغیرهای از پیش تعریف شده (Macro) وجود دارند که از برخی از آنها در دستورات فوق استفاده شده است:

- \$(ProjectDir) : مسیر کامل و مطلق پروژه جاری به همراه یک کاراکتر \ در انتها
- \$(SolutionDir) : مسیر کامل و مطلق سولوشن به همراه یک کاراکتر \ در انتها
- \$(OutDir) : مسیر نسبی فولدر Output پروژه جاری به همراه یک کاراکتر \ در انتها

نکته: این دستورات باید در Post-Build Event پروژه Resources افزوده شوند.

با استفاده از این تنظیمات مطمئن میشویم که پس از هر Build آخرین نسخه از فایل‌های مورد نیاز در مسیرهای تعیین شده کپی میشوند. در نهایت با استفاده از کلاس ResXResourceManager که در زیر آورده شده است، کل عملیات را ساماندهی میکنیم:

```
public class ResXResourceManager
{
    private static readonly object Lock = new object();
    public string ResourcesPath { get; private set; }
    public ResXResourceManager(string resourcesPath)
    {
        ResourcesPath = resourcesPath;
    }
    public IEnumerable<ResXResourceEntry> GetAllResources(string resourceCategory)
    {
        var resourceFilePath = GetResourceFilePath(resourceCategory);
        return Read(resourceFilePath);
    }
    public void AddOrUpdateResource(ResXResourceEntry resource, string resourceCategory)
    {
        var resourceFilePath = GetResourceFilePath(resourceCategory);
        AddOrUpdate(resource, resourceFilePath);
    }
    public void DeleteResource(string key, string resourceCategory)
    {
        var resourceFilePath = GetResourceFilePath(resourceCategory);
        Remove(key, resourceFilePath);
    }
}
```



```

    }
    private string GetResourceFilePath(string resourceCategory)
    {
        var extension = Thread.CurrentThread.CurrentUICulture.TwoLetterISOLanguageName == "en" ? ".resx" :
        ".fa.resx";
        var resourceFilePath = Path.Combine(ResourcesPath, resourceCategory.Replace(".", "\\") +
        extension);
        return resourceFilePath;
    }
    private static void AddOrUpdate(ResXResourceEntry resource, string filePath)
    {
        var list = Read(filePath);
        var entry = list.SingleOrDefault(l => l.Key == resource.Key);
        if (entry == null)
        {
            list.Add(resource);
        }
        else
        {
            entry.Value = resource.Value;
        }
        Write(list, filePath);
    }
    private static void Remove(string key, string filePath)
    {
        var list = Read(filePath);
        list.RemoveAll(l => l.Key == key);
        Write(list, filePath);
    }
    private static List<ResXResourceEntry> Read(string filePath)
    {
        lock (Lock)
        {
            using (var reader = new ResXResourceReader(filePath))
            {
                var list = reader.Cast<object>().Cast<DictionaryEntry>().ToList();
                return list.Select(l => new ResXResourceEntry(l)).ToList();
            }
        }
    }
    private static void Write(IEnumerable<ResXResourceEntry> resources, string filePath)
    {
        lock (Lock)
        {
            using (var writer = new ResXResourceWriter(filePath))
            {
                foreach (var resource in resources)
                {
                    writer.AddResource(resource.Key, resource.Value);
                }
            }
        }
    }
}

```

در این کلاس تغییراتی در متدهای معرفی شده در قسمت‌های بالا برای مدیریت دسترسی همزمان با استفاده از بلاک lock ایجاد شده است.

با استفاده از کلاس BuildManager عملیات تولید کتابخانه‌ها مدیریت میشود. (در مورد نحوه استفاده از MSBuild در [اینجا](#) توضیحات کافی آورده شده است):

```

public class BuildManager
{
    public string ProjectPath { get; private set; }
    public BuildManager(string projectPath)
    {
        ProjectPath = projectPath;
    }
    public void Build()
    {
        var regKey = Registry.LocalMachine.OpenSubKey(@"SOFTWARE\Microsoft\MSBuild\ToolsVersions\4.0");
        if (regKey == null) return;
        var msBuildExeFilePath = Path.Combine(regKey.GetValue("MSBuildToolsPath").ToString(),
        "MSBuild.exe");
        var startInfo = new ProcessStartInfo
        {
            FileName = msBuildExeFilePath,

```

```

        Arguments = ProjectPath,
        WindowStyle = ProcessWindowStyle.Hidden
    };
    var process = Process.Start(startInfo);
    process.WaitForExit();
}
}

```

در نهایت مثلاً با استفاده از کلاس ResXResourceFileManager مدیریت فایل‌های این کتابخانه‌ها صورت می‌پذیرد:

```

public class ResXResourceFileManager
{
    public static readonly string BinPath =
        Path.GetDirectoryName(Assembly.GetExecutingAssembly().GetName().CodeBase.Replace("file:///", ""));
    public static readonly string ResourcesPath = Path.Combine(BinPath, @"..\App_Data\Resources");
    public static readonly string ResourceProjectPath = Path.Combine(ResourcesPath, "Resources.csproj");
    public static readonly string DefaultsPath = Path.Combine(ResourcesPath, "Defaults");
    public static void CopyDlls()
    {
        File.Copy(Path.Combine(ResourcesPath, @"bin\debug\Resources.dll"), Path.Combine(BinPath,
"Resources.dll"), true);
        File.Copy(Path.Combine(ResourcesPath, @"bin\debug\fa\Resources.resources.dll"),
Path.Combine(BinPath, @"fa\Resources.resources.dll"), true);
        Directory.Delete(Path.Combine(ResourcesPath, "bin"), true);
        Directory.Delete(Path.Combine(ResourcesPath, "obj"), true);
    }
    public static void RestoreAll()
    {
        RestoreDlls();
        RestoreResourceFiles();
    }
    public static void RestoreDlls()
    {
        File.Copy(Path.Combine(DefaultsPath, @"bin\Resources.dll"), Path.Combine(BinPath, "Resources.dll"),
true);
        File.Copy(Path.Combine(DefaultsPath, @"bin\fa\Resources.resources.dll"), Path.Combine(BinPath,
@"fa\Resources.resources.dll"), true);
    }
    public static void RestoreResourceFiles(string resourceCategory)
    {
        RestoreFile(resourceCategory.Replace(".", "\\"));
    }
    public static void RestoreResourceFiles()
    {
        RestoreFile(@"Global\Configs");
        RestoreFile(@"Global\Exceptions");
        RestoreFile(@"Global\Paths");
        RestoreFile(@"Global\Texts");

        RestoreFile(@"ViewModels\Employees");
        RestoreFile(@"ViewModels\LogOn");
        RestoreFile(@"ViewModels\Settings");

        RestoreFile(@"Views\Employees");
        RestoreFile(@"Views\LogOn");
        RestoreFile(@"Views\Settings");
    }
    private static void RestoreFile(string subPath)
    {
        File.Copy(Path.Combine(DefaultsPath, subPath + ".resx"), Path.Combine(ResourcesPath, subPath +
".resx"), true);
        File.Copy(Path.Combine(DefaultsPath, subPath + ".fa.resx"), Path.Combine(ResourcesPath, subPath +
".fa.resx"), true);
    }
}

```

در این کلاس از مفهومی با عنوان resourceCategory برای استفاده راحت‌تر در ویوها استفاده شده است که بیانگر فضای نام نسبی فایل‌های Resource و کلاسهای متناظر با آنهاست که براساس استانداردها باید برطبق مسیر فیزیکی آنها در پروژه باشد مثل Global.Texts یا Views.LogOn. همچنین در متد RestoreResourceFiles نمونه‌هایی از مسیرهای این فایل‌ها آورده شده است.

پس از اجرای متد Build از کلاس BuildManager، یعنی پس از build پروژه Resource در زمان اجرا، باید ابتدا فایل‌های تولیدی به

مسیرهای مربوطه در فولدر bin برنامه کپی شده سپس فولدرهای تولیدشده توسط msbuild حذف شوند. این کار در متد CopyDlls از کلاس ResXResourceManager انجام میشود. هرچند در این قسمت فرض شده است که فایل csprj موجود برای حالت debug تنظیم شده است.

نکته: دقت کنید که در این قسمت بلافاصله پس از کپی فایلها در مقصد با توجه به توضیحات ابتدای این مطلب سایت Restart خواهد شد که یکی از ضعفهای عمده این روش به شمار میرود. سایر متدهای موجود نیز برای برگرداندن تنظیمات اولیه بکار میروند. در این متدها از محتویات فولدر Defaults استفاده میشود. **نکته :** در صورت ساخت دوباره اسمبلی و یا بازگرداندن اسمبلیهای اولیه، از آنجاکه وبسایت Restart خواهد شد، بنابراین بهتر است تا صفحه جاری بلافاصله پس از اتمام عملیات، دوباره بارگذاری شود. مثلا اگر از ajax برای اعمال این دستورات استفاده شده باشد میتوان با استفاده از کدی مشابه زیر در پایان فرایند صفحه را دوباره بارگذاری کرد:

```
window.location.reload();
```

در قسمت بعدی راه حل بهتری با استفاده از فراهم کردن پرووایدر سفارشی برای مدیریت فایلهای Resource ارائه میشود.

نظرات خوانندگان

نویسنده: بهمن خلفی
تاریخ: ۹:۲۲ ۱۳۹۲/۰۲/۰۱

با سلام خدمت شما
مطلب بسیار مفیدی است و امیدوارم ادامه دهید...

نویسنده: امیرمحسن یک
تاریخ: ۲۳:۵۰ ۱۳۹۴/۰۲/۰۴

با سلام و با تشکر؛ پس از پیاده سازی مطالب در محیط لوکال، جواب صحیح را گرفتم ولی پس از آپلود نمودن سایت در محیط وب در هنگام Build نمودن پروژه پوشه fa_ir در پوشه bin ایجاد نمی گردد بنابراین قابلیت تغییر عناوین فارسی را ندارم. لازم به عرض که دسترسی write برای پوشه bin و تمامی زیر پوشه ها فعال می باشد. نکته بعدی اینکه فایل d11 اصلی ساخته می شود و فقط در ساخت پوشه fa-ir که فایل d11 زبان فارسی در آن می باشد اشکال وجود دارد.

نویسنده: محسن خان
تاریخ: ۹:۳۷ ۱۳۹۴/۰۲/۰۵

کلاس BuildManager ارائه شده در این مطلب بعیده که روی یک هاست با دسترسی کم اجرا شه. عملا دسترسی اجرای یک فایل exe رو روی هاست ندارید.

نویسنده: امیرمحسن یک
تاریخ: ۱۴:۳۹ ۱۳۹۴/۰۲/۰۵

کلاس BuildManager مگه اجرایی هست؟
موضوع اینه که مشکل من با Build پروژه نیست (Build با موفقیت انجام میشود)
موضوع عدم ایجاد پوشه fa-ir و بالطبع فایل d11 داخل آن می باشد.
خواهشمند است اگر سوال بنده جابیش نامفهوم است بفرمائید که اصلاح نمایم تا به پاسخ برسیم.

نویسنده: محسن خان
تاریخ: ۱۵:۰۶ ۱۳۹۴/۰۲/۰۵

کلاس BuildManager فایل MSBuild.exe رو اجرا می کنه (مطابق سورس مطلب فوق). بنابراین دو سؤال هست: آیا این فایل exe روی سرور هست؟ اگر هست آیا دسترسی Process.Start دارید؟ یعنی دسترسی دارید فایل exe اجرا کنید با برنامه ی ASP.NET؟

نویسنده: امیرمحسن یک
تاریخ: ۱۷:۲۴ ۱۳۹۴/۰۲/۰۵

بلی MSBuild قابلیت اجرا بر روی IIS سرورهای arvixe را دارند (علت را نمی دانم)

مشکل بنده با Build پروژه نیست (Build با موفقیت انجام میشود و فایل d11 زبان اصلی با موفقیت ساخته می شود)
موضوع عدم ایجاد پوشه fa-ir و بالطبع فایل d11 داخل آن می باشد.

نویسنده: امیرمحسن یک
تاریخ: ۱:۷ ۱۳۹۴/۰۵/۰۴

با تشکر از شما جواب در به هم خوردن سطح دسترسی بود که با توجه به [لینک زیر](#) تصحیح شد.

قبل از ادامه، بهتر است یک مقدمه کوتاه درباره انواع منابع موجود در ASP.NET ارائه شود تا درک مطالب بعدی آسانتر شود.

نکات اولیه

- یک فایل Resource درواقع یک فایل XML شامل رشته هایی برای ذخیره سازی مقادیر (منابع) مورد نیاز است. مثلاً رشته هایی برای ترجمه به زبانهای دیگر، یا مسیرهایی برای یافتن تصاویر یا فایلها و ... پسوند این فایلها .resx است (مثل MyResource.resx).

- این فایلها برای ذخیره منابع از جفت داده‌های کلید-مقدار (key-value pair) استفاده می‌کنند. هر کلید معرف یک ورودی مجزاست. نام این کلیدها حساس به حروف بزرگ و کوچک نیست (Not Case-Sensitive).

- برای هر زبان (مثل fa برای فارسی) یا کالچر مورد نظر (مثل fa-IR برای فارسی ایرانی) می‌توان یک فایل Resource جداگانه تولید کرد. عنوان زبان یا کالچر باید جزئی از نام فایل Resource مربوطه باشد (مثل MyResource.fa.resx یا MyResource.fa-IR.resx). هر منبع باید دارای یک فایل اصلی (پیش فرض) Resource باشد. این فایل، فایلی است که برای حالت پیش فرض برنامه (بدون کالچر) تهیه شده است و در عنوان آن از نام زبان یا کالچری استفاده نشده است (مثل MyResource.resx). برای اطلاعات بیشتر به [قسمت اول](#) این سری مراجعه کنید.

- تمامی فایل‌های Resource باید دارای کلیدهای یکسان با فایل اصلی Resource باشند. البته لزومی ندارد که این فایل‌ها حاوی تمامی کلیدهای منبع پیش فرض باشند. در صورت عدم وجود کلیدی در یک فایل Resource عملیات پیش فرض موجود در دات نت با استفاده از فرایند مشهور به fallback مقدار کلید مورد نظر را از نزدیکترین و مناسبترین فایل موجود انتخاب می‌کند (درباره این رفتار در [قسمت اول](#) توضیحاتی ارائه شده است).

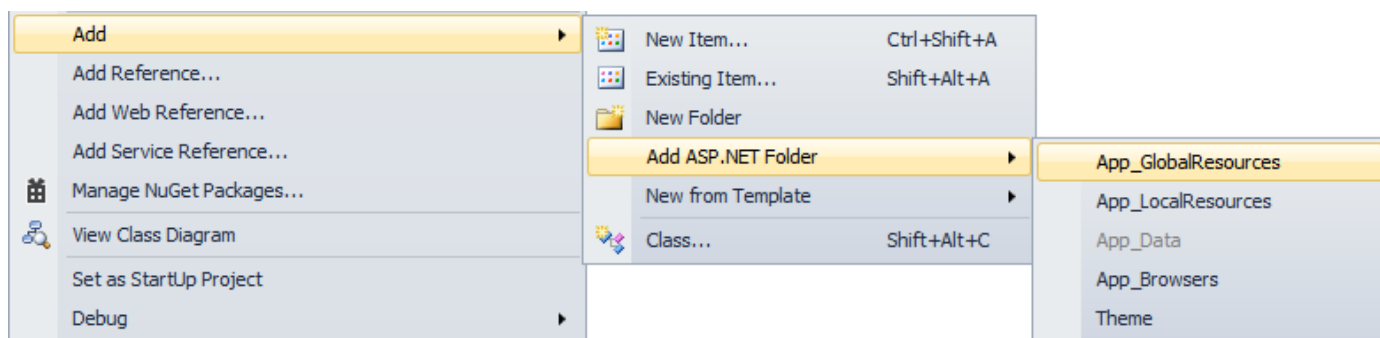
- در زمان اجرا موتور پیش فرض مدیریت منابع دات نت با توجه به کالچر UI در ثرد جاری اقدام به انتخاب مقدار مناسب برای کلیدهای درخواستی (به همراه فرایند fallback) می‌کند. فرایند نسبتاً پیچیده fallback در [اینجا](#) شرح داده شده است.

منابع Global و Local

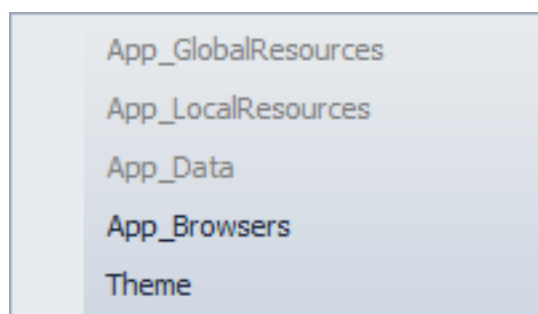
در ASP.NET دو نوع کلی Resource وجود دارد که هر کدام برای موقعیت‌های خاصی مورد استفاده قرار می‌گیرند:

- Resource های Global : منابعی کلی هستند که در تمام برنامه در دسترسند. این فایل‌ها در مسیر رزرو شده APP_GlobalResources در ریشه سایت قرار می‌گیرند. محتوای هر فایل .resx موجود در این فولدر دارای دسترسی کلی خواهد بود.

- Resource های Local : این منابع همان‌طور که از نامشان پیداست محلی هستند و درواقع مخصوص همان مسیری هستند که در آن تعبیه شده اند! در استفاده از منابع محلی به ازای هر صفحه وب (aspx یا master) یا هر یوزرکنترل (ascx) یک فایل .resx تولید می‌شود که تنها در همان صفحه یا یوزرکنترل در دسترس است. این فایل‌ها درون فولدر رزرو شده APP_LocalResources در مسیرهای مورد نظر قرار می‌گیرند. درواقع در هر مسیری که نیاز به این نوع از منابع باشد، باید فولدري با عنوان App_LocalResources ایجاد شود و فایل‌های .resx مرتبط با صفحه‌ها یا یوزرکنترل‌های آن مسیر در این فولدر مخصوص قرار گیرد. در تصویر زیر چگونگی افزودن این فولدرهای مخصوص به پروژه وب اپلیکیشن نشان داده شده است:



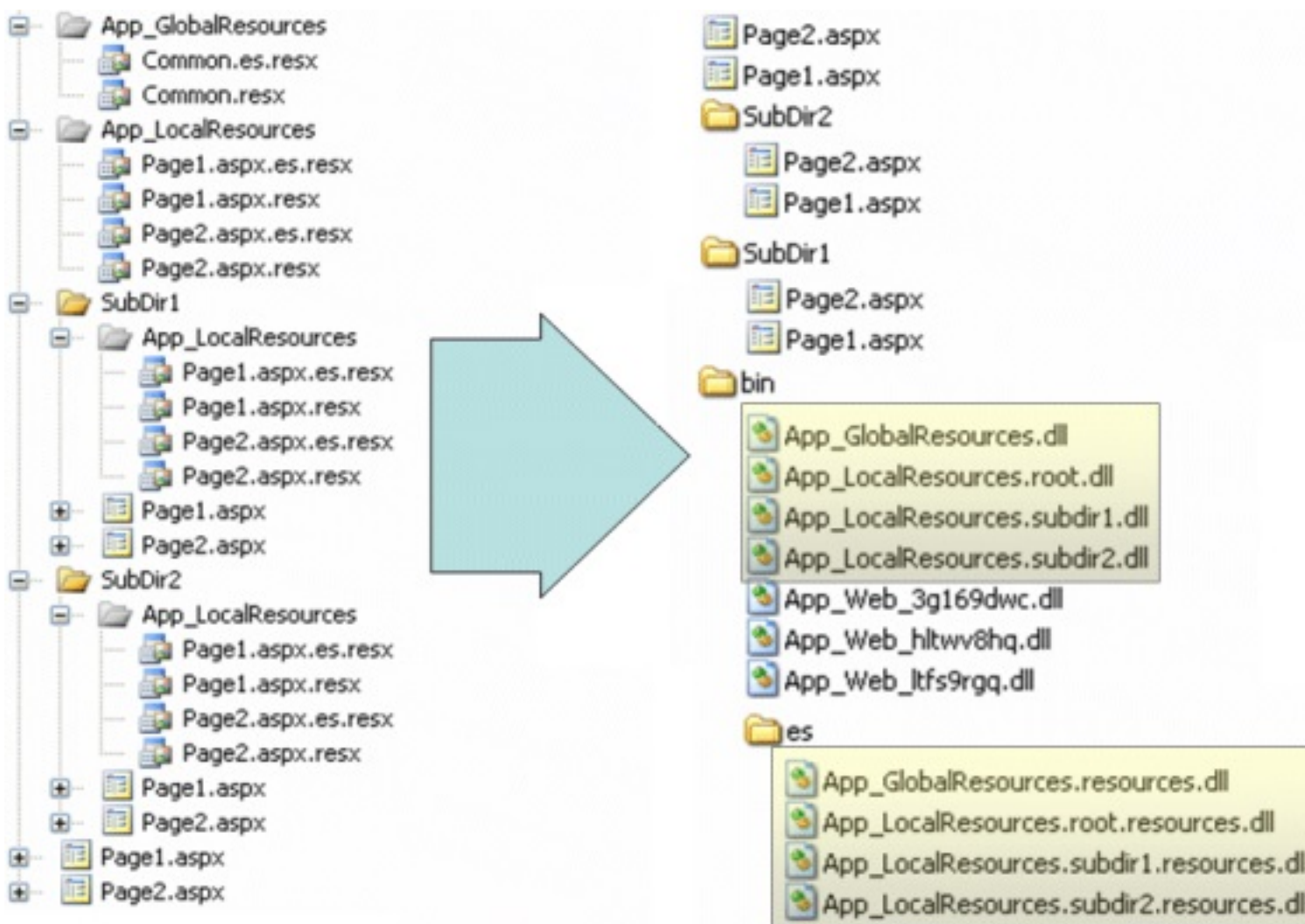
نکته: دقت کنید که تنها یک فولدر App_GlobalResources به هر پروژه می‌توان افزود. همچنین در ریشه هر مسیر موجود در پروژه تنها می‌توان یک فولدر App_LocalResources داشت. پس از افزودن هر یک از این فولدرهای مخصوص، منوی فوق به صورت زیر در خواهد آمد:



نکته: البته با تغییر نام یک فولدر معمولی به این نام‌های رزرو شده نتیجه یکسانی بدست خواهد آمد.

نکته: در زمان اجرا، عملیات استخراج داده‌های موجود در این نوع منابع، به صورت **خودکار** توسط ASP.NET انجام می‌شود. این داده‌ها پس از استخراج در حافظه سرور کش خواهند شد.

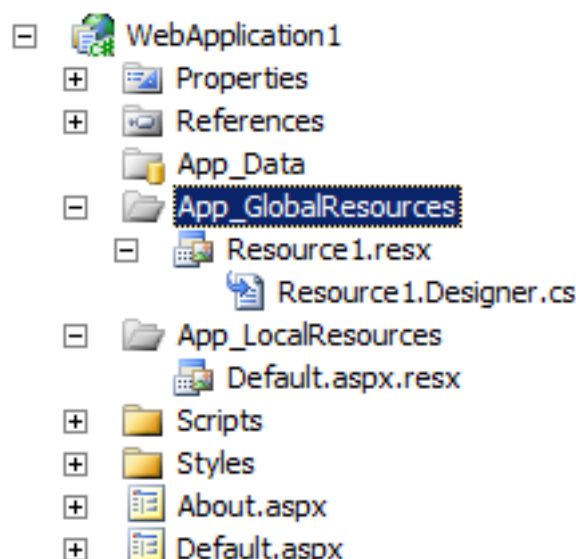
برای روشن‌تر شدن مطالب اشاره شده در بالا به تصویر فرضی! زیر توجه کنید (اسمبلی‌های تولید شده برای منابع کلی و محلی فرضی است):



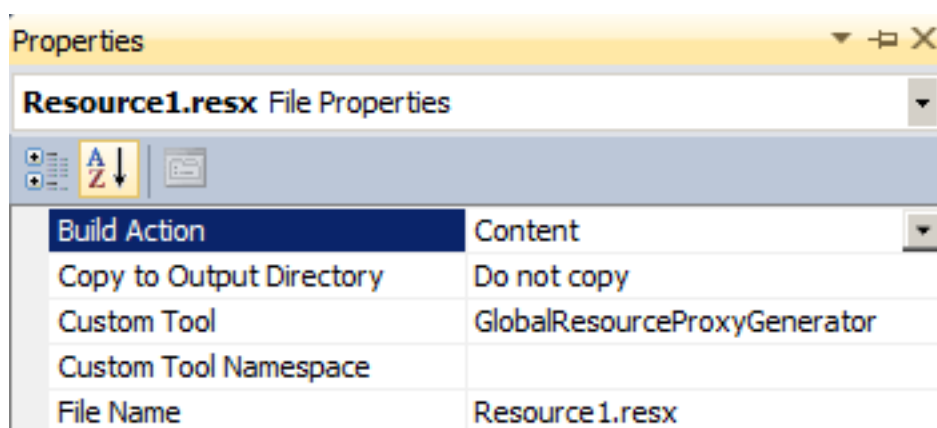
در تصویر بالا محل قرارگیری انواع مختلف فایل‌های Resource و نیز محل نهایی فرضی اسمبلی‌های ستلایت تولید شده، برای حداقل یک زبان غیر از زبان پیش فرض برنامه، نشان داده شده است.

نکته: نحوه برخورد با این نوع از فایل‌های Resource در پروژه‌های Web Site و Web Application کمی باهم فرق می‌کند. موارد اشاره شده در این مطلب بیشتر درباره Web Application ها صدق می‌کند.

برای آشنایی بیشتر بهتر است یک برنامه **وب اپلیکیشن** جدید ایجاد کرده و همانند تصویر زیر یکسری فایل Resource به فولدرهای اشاره شده در بالا اضافه کنید:

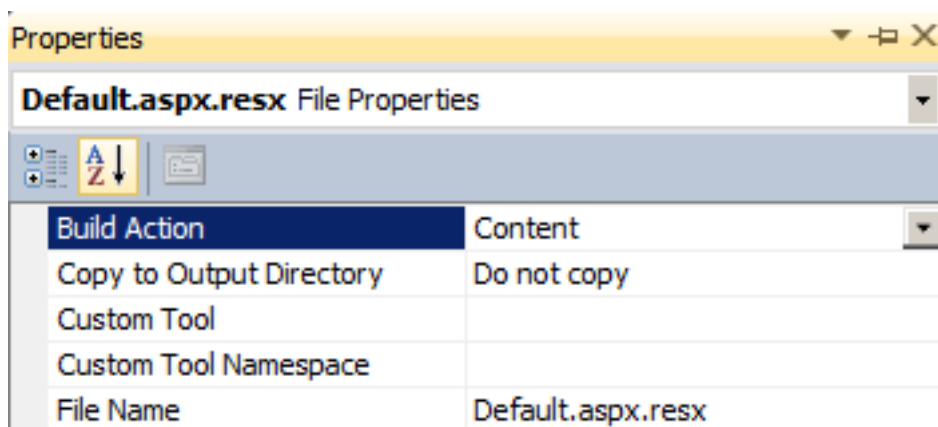


همانطور که مشاهده می‌کنید به صورت پیش‌فرض برای منابع کلی یک فایل cs. تولید می‌شود. اما اثری از این فایل برای منابع محلی نیست. حال اگر پنجره پراپرتی فایل منبع کلی را باز نمایید با چیزی شبیه به تصویر زیر مواجه خواهید شد:



می‌بینید که خاصیت Build Action آن به Content مقداردهی شده است. این مقدار موجب می‌شود تا این فایل به همین صورت و در همین مسیر مستقیماً در پابلیش نهایی برنامه ظاهر شود. در [قسمت قبل](#) به خاصیت Build Action و مقادیر مختلف آن اشاره شده است.

همچنین می‌بینید که مقدار پراپرتی Custom Tool به **GlobalResourceProxyGenerator** تنظیم شده است. این ابزار مخصوص تولید کلاس مربوط به منابع کلی در ویژوال استودیو است. با استفاده از این ابزار فایل `Resource1.Designer.cs` که در تصویر قبلی نیز نشان داده شده، تولید می‌شود. حالا پنجره پراپرتی‌های منبع محلی را باز کنید:



می‌بینید که همانند منبع کلی خاصیت Build Action آن به Content تنظیم شده است. همچنین مقداری برای پراپرتی Custom Tool تنظیم نشده است. این مقدار پیش فرض را تغییر ندهید، چون با تنظیم مقداری برای آن چیز مفیدی عایدتان نمی‌شود!

نکته: برای به روز رسانی مقادیر کلیدهای منابعی که با توجه به توضیحات بالا به همراه برنامه به صورت فایل‌های resx. پابلیش می‌شوند، کافی است تا محتوای فایل‌های resx. مربوطه با استفاده از یک ابزار (همانند نمونه ای که در قسمت [قبل](#) شرح داده شد) تغییر داده شوند. بقیه عملیات توسط ASP.NET انجام خواهد شد. اما با تغییر محتوای این فایل‌های resx. با توجه به رفتار FCN در ASP.NET (که در قسمت [قبل](#) نیز توضیح داده شد) سایت Restart خواهد شد. البته این روش تنها برای منابع کلی و محلی درون مسیرهای مخصوص اشاره شده کار خواهد کرد.

استفاده از منابع Local و Global

پس از تولید فایل‌های Resource، می‌توان از آن‌ها در صفحات وب استفاده کرد. معمولاً از این نوع منابع برای مقداردی پراپرتی کنترل‌ها در صفحات وب استفاده می‌شود. برای استفاده از کلیدهای منابع محلی می‌توان از روشی همانند زیر بهره برد:

```
<asp:Label ID="lblLocal" runat="server" meta:resourcekey="lblLocalResources" ></asp:Label>
```

اما برای منابع کلی تنها می‌توان از روش زیر استفاده کرد (یعنی برای منابع محلی نیز می‌توان از این روش استفاده کرد):

```
<asp:Label ID="lblGlobal" runat="server" Text="<%"$ Resources:CommonTerms, HelloText %"></asp:Label>
```

به این عبارات که با فوت پررنگ مشخص شده اند اصطلاحاً «عبارات بومی‌سازی» (Localization Expression) می‌گویند. در ادامه این سری مطالب با نحوه تعریف نمونه‌های سفارشی آن آشنا خواهیم شد.

به نمونه اول که برای منابع محلی استفاده می‌شود نوع ضمنی (Implicit Localization Expression) می‌گویند. زیرا نیازی نیست تا محل کلید موردنظر صراحتاً ذکر شود!

به نمونه دوم که برای منابع کلی استفاده می‌شود نوع صریح (Explicit Localization Expression) می‌گویند. زیرا برای یافتن کلید موردنظر باید آدرس دقیق آن ذکر شود!

بومی سازی ضمنی (Implicit Localization) با منابع محلی عنوان کلید مربوطه در این نوع عبارات همانطور که در بالا نشان داده شده است، با استفاده از پراپرتی مخصوص meta:resourcekey مشخص می‌شود. در استفاده از منابع محلی تنها یک نام برای کل خواص کنترل مربوطه در صفحات وب کفایت می‌کند. زیرا عنوان کلیدهای این منبع باید از طرح زیر پیروی کند:

ResourceKey.Property

ResourceKey.Property-SubProperty یا ResourceKey.Property.SubProperty

برای مثال در لیبل بالا که نام کلید Resource آن به lblLocalResources تنظیم شده است، اگر نام صفحه وب مربوطه page1.aspx باشد، برای تنظیم خواص آن در فایل page1.aspx.resx مربوطه باید از کلیدهایی با عناوینی مثل عنوان‌های زیر استفاده کرد:

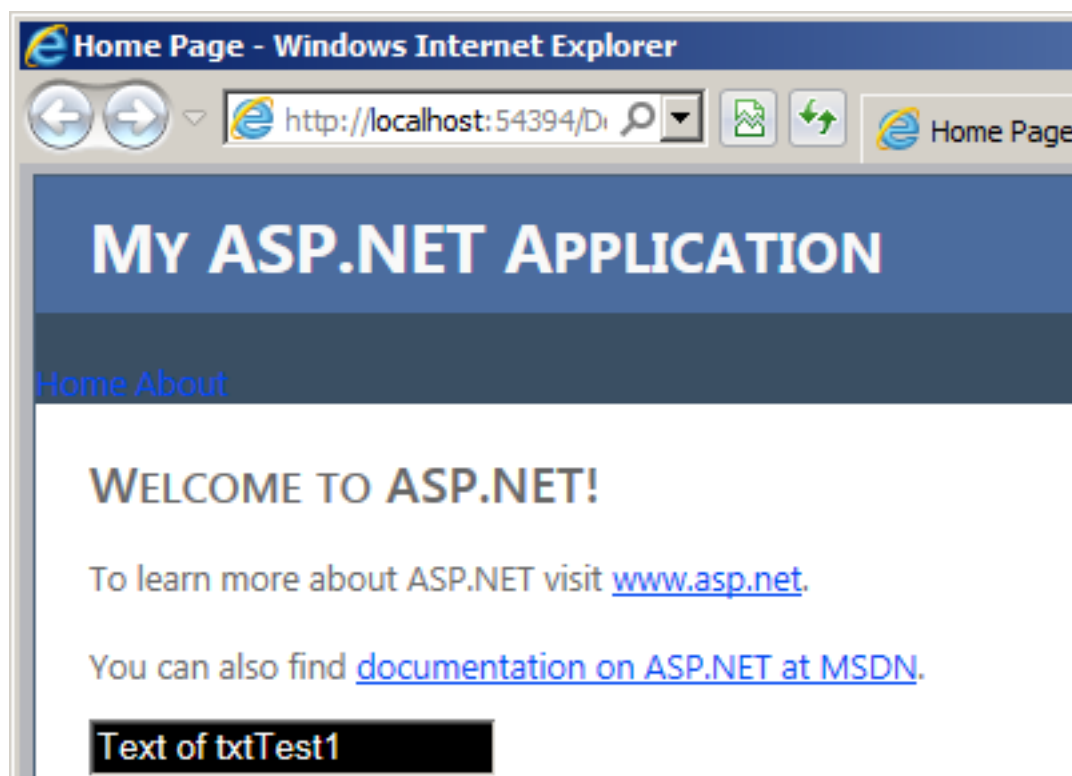
lblLocalResources.Text

lblLocalResources.BackColor

برای نمونه به تصاویر زیر دقت کنید:

```
<asp:TextBox ID="txtTest" runat="server" meta:resourcekey="txtTest" />
```

Default.aspx.resx X Default.aspx Default.aspx.cs		
Strings	Add Resource	Remove Resource
		Access Modifier: No code gener
	Name	Value
	txtTest.Text	Text of txtTest1
	txtTest.BackColor	Black
	txtTest.ForeColor	White



بومی سازی صریح (Explicit Localization)

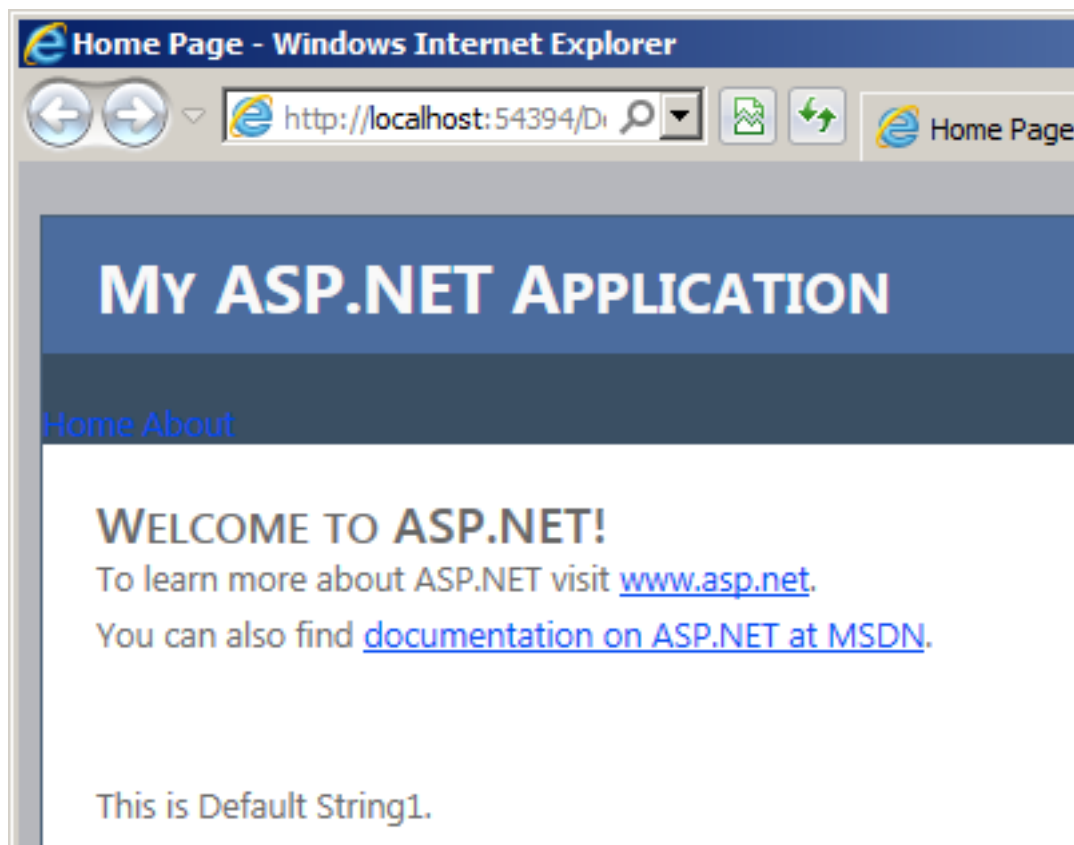
در استفاده از این نوع عبارات، پراپرتی مربوطه و نام فایل منبع صراحتاً در تگ کنترل مربوطه آورده می‌شود. بنابراین برای هر خاصیتی که می‌خواهیم مقدار آن از منبعی خاص گرفته شود باید از عبارتی با طرح زیر استفاده کنیم:

```
<% Resources: Class, ResourceKey %>
```

در این عبارت، رشته Resources **پیشوند (Prefix)** نام دارد و مشخص کننده استفاده از نوع صریح عبارات بومی سازی است. Class نام کلاس مربوط به فایل منبع بوده و اختیاری است که تنها برای منابع کلی باید آورده شود. ResourceKey نیز کلید مربوطه را در فایل منبع مشخص می‌کند.
برای نمونه به تصاویر زیر دقت کنید:

```
<asp:Label ID="Label1" runat="server" Text="<%"$ Resources: Resource1, String1 %"
```

Resource1.resx		Default.aspx.resx	Default.aspx	Default.aspx.cs
Strings		Add Resource	Remove Resource	Access Modifier:
		Name	Value	
		String1	This is Default String1.	



نکته: استفاده همزمان از این دو نوع عبارت بومی سازی در یک کنترل مجاز نیست!

نکته: به دلیل تولید کلاسی مخصوص منابع کلی (با توجه به توضیحات ابتدای این مطلب راجع به پراپرتی Custom Tool)، امکان استفاده مستقیم از آن درون کد نیز وجود دارد. این کلاسها که به صورت خودکار تولید می‌شوند، به صورت مستقیم از کلاس ResourceManager برای یافتن کلیدهای منابع استفاده می‌کنند. اما روش مستقیمی برای استفاده از کلیدهای منابع محلی درون کد وجود ندارد.

نکته: درون کلاس System.Web.UI.TemplateControl و نیز کلاس HttpContext دو متد با نامهای GetGlobalResourceObject و GetLocalResourceObject وجود دارد که برای یافتن کلیدهای منابع به صورت غیرمستقیم استفاده می‌شوند. مقدار برگشتی این دو متد از نوع object است. این دو متد به صورت مستقیم از کلاس ResourceManager استفاده نمی‌کنند! هم‌چنین از آنجاکه کلاس Page از کلاس TemplateControl مشتق شده است، بنابراین این دو متد در صفحات وب در دسترس هستند.

دسترسی با برنامه نویسی

همانطور که در بالا اشاره شد امکان دستیابی به کلیدهای منابع محلی و کلی از طریق دو متد GetGlobalResourceObject و GetLocalResourceObject نیز امکان پذیر است. این دو متد با فراخوانی ResourceProviderFactory جاری سعی در یافتن مقادیر کلیدهای درخواستی در منابع موجود می‌کنند. درباره این فرایند در مطالب بعدی به صورت مفصل بحث خواهد شد.

کلاس TemplateControl

این دو متد در کلاس TemplateControl از نوع Instance (غیر استاتیک) هستند. امضای (Signature) این دو متد در این کلاس به صورت زیر است:

متد GetLocalResourceObject:

```
protected object GetLocalResourceObject(string resourceKey)
protected object GetLocalResourceObject(string resourceKey, Type objType, string propName)
```

در متد اول، پارامتر resourceKey در متد GetLocalResourceObject معرف کلید منبع مربوطه در فایل منبع محلی متناظر با صفحه جاری است. مثلا lblLocalResources.Text. از آنجاکه به صورت پیش فرض موقعیت فایل منبع محلی مرتبط با صفحات وب مشخص است بنابراین تنها ارائه کلید مربوطه برای یافتن مقدار آن کافی است. مثال:

```
txtTest.Text = GetLocalResourceObject("txtTest.Text") as string;
```

متد دوم برای استخراج کلیدهای منبع محلی با مشخص کردن نوع داده محتوا (معمولا برای داده های غیر رشته ای) و پراپرتی موردنظر به کار می رود. در این متد پارامتر objType برای معرفی نوع داده متناظر با داده موجود در کلید resourceKey استفاده می شود. از پارامتر propName نیز همانطور که از نامش پیداست برای مشخص کردن پراپرتی موردنظر از این نوع داده معرفی شده استفاده می شود.

متد GetGlobalResourceObject:

```
protected object GetGlobalResourceObject(string className, string resourceKey)
protected object GetGlobalResourceObject(string className, string resourceKey, Type objType, string propName)
```

در این دو متد، پارامتر className مشخص کننده نام کلاس متناظر با فایل منبع اصلی (فایل منبع اصلی که کلاس مربوطه با نام آن ساخته می شود) است. سایر پارامترها همانند دو متد قبلی است. مثال:

```
TextBox1.Text = GetGlobalResourceObject("Resource1", "String1") as string;
```

کلاس HttpContext

در این کلاس دو متد موردبحث از نوع استاتیک و به صورت زیر تعریف شده اند:

متد GetLocalResourceObject:

```
public static object GetLocalResourceObject(string virtualPath, string resourceKey)
public static object GetLocalResourceObject(string virtualPath, string resourceKey, CultureInfo culture)
```

در این دو متد، پارامتر virtualPath مشخص کننده مسیر نسبی صفحه وب متناظر با فایل منبع محلی موردنظر است، مثل "~/Default.aspx". پارامتر resourceKey نیز کلید منبع را تعیین می کند و پارامتر culture نیز به کالچر موردنظر اشاره دارد. مثال:

```
txtTest.Text = HttpContext.GetLocalResourceObject("~/Default.aspx", "txtTest.Text") as string;
```

متد GetGlobalResourceObject:

```
public static object GetGlobalResourceObject(string classKey, string resourceKey)
public static object GetGlobalResourceObject(string classKey, string resourceKey, CultureInfo culture)
```

در این دو متد، پارامتر className مشخص کننده نام کلاس متناظر با فایل منبع اصلی (فایل منبع بدون نام زبان که کلاس مربوطه با نام آن ساخته می شود) است. سایر پارامترها همانند دو متد قبلی است. مثال:

```
TextBox1.Text = HttpContext.GetGlobalResourceObject("Resource1", "String1") as string;
```

نکته: بدیهی است که در MVC تنها می‌توان از متدهای کلاس HttpContext استفاده کرد.

روش دیگری که تنها برای منابع کلی در دسترس است، استفاده مستقیم از کلاسی است که به صورت خودکار توسط ابزارهای Visual Studio برای فایل منبع اصلی تولید می‌شود. نمونه‌ای از این کلاس را که برای یک فایل Resource1.resx (که تنها یک ورودی با نام String1 دارد) در پوشه App_GlobalResources تولید شده است، در زیر مشاهده می‌کنید:

```
//-----
// <auto-generated>
// This code was generated by a tool.
// Runtime Version:4.0.30319.17626
//
// Changes to this file may cause incorrect behavior and will be lost if
// the code is regenerated.
// </auto-generated>
//-----

namespace Resources {
    using System;

    /// <summary>
    /// A strongly-typed resource class, for looking up localized strings, etc.
    /// </summary>
    // This class was auto-generated by the StronglyTypedResourceBuilder
    // class via a tool like ResGen or Visual Studio.
    // To add or remove a member, edit your .ResX file then rerun ResGen
    // with the /str option or rebuild the Visual Studio project.

    [global::System.CodeDom.Compiler.GeneratedCodeAttribute("Microsoft.VisualStudio.Web.Application.StronglyTypedResourceProxyBuilder", "10.0.0.0")]
    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [global::System.Runtime.CompilerServices.CompilerGeneratedAttribute()]
    internal class Resource1 {

        private static global::System.Resources.ResourceManager resourceMan;

        private static global::System.Globalization.CultureInfo resourceCulture;

        [global::System.Diagnostics.CodeAnalysis.SuppressMessageAttribute("Microsoft.Performance",
        "CA1811:AvoidUncalledPrivateCode")]
        internal Resource1() {
        }

        /// <summary>
        /// Returns the cached ResourceManager instance used by this class.
        /// </summary>

        [global::System.ComponentModel.EditorBrowsableAttribute(global::System.ComponentModel.EditorBrowsableState.Advanced)]
        internal static global::System.Resources.ResourceManager ResourceManager {
            get {
                if (object.ReferenceEquals(resourceMan, null)) {
                    global::System.Resources.ResourceManager temp = new
global::System.Resources.ResourceManager("Resources.Resource1",
global::System.Reflection.Assembly.Load("App_GlobalResources"));
                    resourceMan = temp;
                }
                return resourceMan;
            }
        }

        /// <summary>
        /// Overrides the current thread's CurrentUICulture property for all
        /// resource lookups using this strongly typed resource class.
        /// </summary>

        [global::System.ComponentModel.EditorBrowsableAttribute(global::System.ComponentModel.EditorBrowsableState.Advanced)]
        internal static global::System.Globalization.CultureInfo Culture {
            get {
                return resourceCulture;
            }
            set {
                resourceCulture = value;
            }
        }

        /// <summary>
```

```

    /// Looks up a localized string similar to String1.
    /// </summary>
    internal static string String1 {
        get {
            return ResourceManager.GetString("String1", resourceCulture);
        }
    }
}

```

نکته: فضای نام پیش‌فرض برای منابع کلی در این کلاس‌ها همیشه Resources است که برابر پیشوند (Prefix) عبارت بومی سازی صریح است.

نکته: در کلاس بالا نحوه نمونه سازی کلاس ResourceManager نشان داده شده است. همانطور که مشاهده می‌کنید تعیین کردن مشخصات فایل اصلی Resource مربوطه که در اسمبلی نهایی تولید و کش می‌شود، اجباری است! در مطلب بعدی با این کلاس بیشتر آشنا خواهیم شد.

نکته: همانطور که قبلاً نیز اشاره شد، کار تولید اسمبلی مربوط به فایل‌های منابع کلی و محلی و کش کردن آن‌ها در اسمبلی در زمان اجرا کاملاً بر عهده ASP.NET است. مثلاً در نمونه کد بالا می‌بینید که کلاس ResourceManager برای استخراج نوع Resources.Resource1 از اسمبلی App_GlobalResources نمونه‌سازی شده است، با اینکه این اسمبلی و نوع مذکور در زمان کامپایل و پابلیش وجود ندارد!

برای استفاده از این کلاس می‌توان به صورت زیر عمل کرد:

```

TextBox1.Text = Resources.Resource1.String1;

```

نکته: همانطور که قبلاً هم اشاره شد، متأسفانه روش بالا (برخلاف دو متدی که در قسمت قبل توضیح داده شد) به صورت مستقیم از کلاس ResourceManager استفاده می‌کند، که برای بحث سفارشی سازی پرووایدرهای منابع مشکل‌زاست. در مطالب بعدی با معایب آن و نیز راه حل‌های موجود آشنا خواهیم شد.

نکات نهایی

حال که با مفاهیم کلی بیشتری آشنا شدیم بهتر است کمی هم به نکات ریزتر بپردازیم:

نکته: فایل تولیدی توسط ویژوال استودیو در فرایند مدیریت منابع ASP.NET تاثیرگذار نیست! باز هم تأکید می‌کنم که کار استخراج کلیدهای Resource از درون فایل‌های resx. کاملاً به صورت جداگانه و خودکار و در زمان اجرا انجام می‌شود (درباره این فرایند در مطالب بعدی شرح مفصلی خواهد آمد). درواقع شما می‌توانید خاصیت Custom Tool مربوط به منابع کلی را نیز همانند منابع محلی به رشته‌ای خالی مقداردهی کنید و ببینید که خللی در فرایند مربوطه رخ نخواهد داد!

نکته: تنها برای حالتی که بخواهید از روش آخری که در بالا اشاره شد برای دسترسی با برنامه‌نویسی به منابع کلی بهره ببرید (روش مستقیم)، به این کلاس تولیدی توسط ویژوال استودیو نیاز خواهید داشت. دقت کنید که در این کلاس نیز کار اصلی برعهده کلاس ResourceManager است. درواقع می‌توان کلاً از این فایل خودکار تولیدشده صرف‌نظر کرد و کار استخراج کلیدهای منابع را به صورت مستقیم به نمونه‌ای از کلاس ResourceManager سپرد. این روش نیز در قسمت‌های بعدی شرح داده خواهد شد.

نکته: اگر فایل‌های Resource درون اسمبلی‌های جداگانه‌ای باشند (مثلاً در یک پروژه جداگانه، همانطور که در قسمت اول این سری مطالب پیشنهاد شده است)، موتور پیش‌فرض منابع در ASP.NET ببرد نخواهد خورد! بنابراین یا باید از نمونه‌های اختصاصی کلاس ResourceManager استفاده کرد (کاری که کلاس‌های خودکار تولیدشده توسط ابزارهای ویژوال استودیو انجام می‌دهند)، یا باید از پرووایدرهای سفارشی استفاده کرد که در مطالب بعدی نحوه تولید آن‌ها شرح داده خواهد شد.

همانطور که در ابتدای این مطلب اشاره شد، این مقدمه در اینجا صرفاً برای آشنایی بیشتر با این دونوع Resource آورده شده تا ادامه مطلب روشن‌تر باشد، زیرا با توجه به مطالب ارائه شده در [قسمت اول](#) این سری، در پروژه‌های MVC استفاده از یک پروژه جداگانه برای نگهداری این منابع راه حل مناسبتری است.

در مطلب بعدی به شرح نحوه تولید پرووایدرهای سفارشی می‌پردازم.

منابع:

<http://msdn.microsoft.com/en-us/library/aa905797.aspx>
<http://msdn.microsoft.com/en-us/library/ms227427.aspx> <http://www.west-wind.com/presentations/wfdbresourceprovider>
[http://msdn.microsoft.com/en-us/library/1ztca10y\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/1ztca10y(v=vs.100).aspx)
[http://msdn.microsoft.com/en-us/library/ms227982\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/ms227982(v=vs.100).aspx)
[http://msdn.microsoft.com/en-us/library/sb6a8618\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/sb6a8618(v=vs.100).aspx)

نظرات خوانندگان

نویسنده: میهمان
تاریخ: ۱۳۹۲/۰۲/۱۸ ۱:۳

ممنون از مطلب بسیار مفیدتان

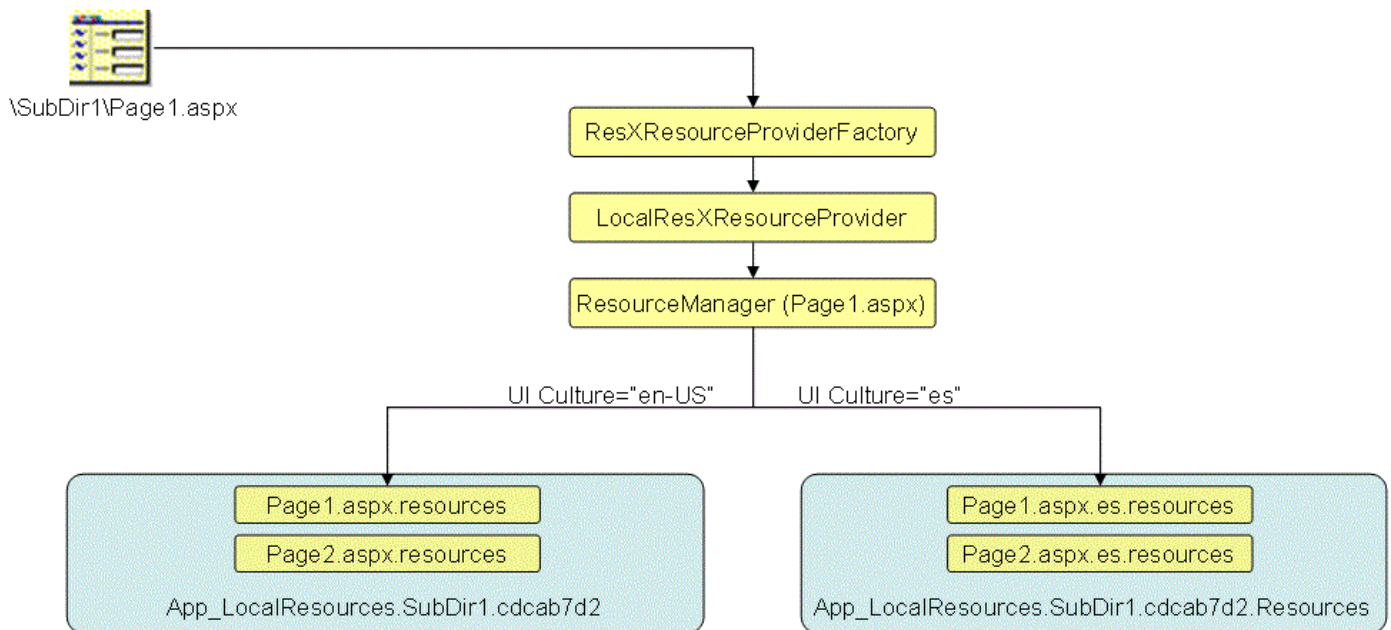
در [قسمت قبل](#) مقدمه ای راجع به انواع منابع موجود در ASP.NET و برخی مسائل پیرامون آن ارائه شد. در این قسمت راجع به نحوه رفتار ASP.NET در برخورد با انواع منابع بحث می‌شود.

مدیریت منابع در ASP.NET

در مدل پرووایدر منابع در ASP.NET کار مدیریت منابع از کلاس **ResourceProviderFactory** شروع می‌شود. این کلاس که از نوع **abstract** تعریف شده است، دو متد برای فراهم کردن پرووایدرهای کلی و محلی دارد. کلاس پیش‌فرض در ASP.NET برای پیاده‌سازی **ResourceProviderFactory** در اسمبلی **System.Web** قرار دارد. این کلاس که **ResXResourceProviderFactory** نام دارد نمونه‌هایی از کلاس‌های **LocalResXResourceProvider** و **GlobalResXResourceProvider** را برمی‌گرداند. درباره این کلاس‌ها در ادامه بیشتر بحث خواهد شد.

نکته: هر سه کلاس پیش‌فرض اشاره شده در بالا و نیز سایر کلاس‌های مربوط به عملیات مدیریت منابع در آن‌ها، همگی در فضای نام **System.Web.Compilation** قرار دارند و متاسفانه دارای سطح دسترسی **internal** هستند. بنابراین به صورت مستقیم در دسترس نیستند.

برای نمونه با توجه به تصویر فرضی نشان داده شده در [قسمت قبل](#)، در اولین بارگذاری صفحه **SubDir1\Page1.aspx** عبارات ضمنی بکاربرده شده در این صفحه برای منابع محلی (در [قسمت قبل](#) شرح داده شده است) باعث فراخوانی متد مربوط به **LocalResources** در کلاس **ResXResourceProviderFactory** می‌شود. این متد نمونه‌ای از کلاس **LocalResXResourceProvider** برمی‌گرداند. (در ادامه با نحوه سفارشی‌سازی این کلاس‌ها نیز آشنا خواهیم شد). رفتار پیش‌فرض این پرووایدر این است که نمونه‌ای از کلاس **ResourceManager** با توجه به کلید درخواستی برای صفحه موردنظر (مثلاً نوع **Page1.aspx** در اسمبلی **App_LocalResources.subdir1.XXXXXX** که در تصویر موجود در [قسمت قبل](#) نشان داده شده است) تولید می‌کند. حال این کلاس با استفاده از کالچر مربوط به درخواست موردنظر، ورودی موردنظر را از منبع مربوطه استخراج می‌کند. مثلاً اگر کالچر موردبحث **es** (اسپانیایی) باشد، اسمبلی ستلایت موجود در مسیر نسبی **\es** انتخاب می‌شود. برای روشن‌تر شدن بحث به تصویر زیر که عملیات مدیریت منابع پیش‌فرض در ASP.NET در درخواست صفحه **Page1.aspx** از پوشه **SubDir1** را نشان می‌دهد، دقت کنید:



همانطور که در [قسمت اول](#) این سری مطالب عنوان شد، رفتار کلاس ResourceManager برای یافتن کلیدهای Resource، استخراج آن از نزدیکترین گزینه موجود است. یعنی مثلاً برای یافتن کلیدی در کالچر es در مثال بالا، ابتدا اسمبلی‌های مربوط به این کالچر جستجو می‌شود و اگر ورودی موردنظر یافته نشد، جستجو در اسمبلی‌های ستلایت پیش‌فرض سیستم موجود در ریشه فولدر bin برنامه ادامه می‌یابد، تا در نهایت نزدیک‌ترین گزینه پیدا شود (فرایند fallback).

نکته: همانطور که در تصویر بالا نیز مشخص است، نحوه نامگذاری اسمبلی منابع محلی به صورت `App_LocalResources.<SubDirectory>.<A random code>` است.

نکته: پس از اولین بارگذاری هر اسمبلی، آن اسمبلی به همراه خود نمونه کلاس ResourceManager که مثلاً توسط کلاس LocalResXResourceProvider تولید شده است در حافظه سرور کش می‌شوند تا در استفاده‌های بعدی به کار روند.

نکته: فرایند مشابهی برای یافتن کلیدها در منابع کلی (Global Resources) به انجام می‌رسد. تنها تفاوت آن این است که کلاس ResXResourceProviderFactory نمونه‌ای از کلاس GlobalResXResourceProvider تولید می‌کند.

چرا پرووایدر سفارشی؟

تا اینجا بالا با کلیات عملیاتی که ASP.NET برای بارگذاری منابع محلی و کلی به انجام می‌رساند، آشنا شدیم. حالا باید به این پرسش پاسخ داد که چرا پرووایدری سفارشی نیاز است؟ علاوه بر دلایلی که در قسمت‌های قبلی به آنها اشاره شد، می‌توان دلایل زیر را نیز برشمرد:

- **استفاده از منابع و یا اسمبلی‌های ستلایت موجود** - اگر بخواهید در برنامه خود از اسمبلی‌هایی مشترک، بین برنامه‌های ویندوزی و وبی استفاده کنید، و یا بخواهید به هر دلیلی از اسمبلی‌های جداگانه‌ای برای این منابع استفاده کنید، مدل پیش‌فرض موجود در ASP.NET جوابگو نخواهد بود.

- **استفاده از منابع دیگری به غیر از فایل‌های resx**. مثل دیتابیس - برای برنامه‌های تحت وب که صفحات بسیار زیاد به همراه ورودی‌های بیشماری از Resource دارند، استفاده از مدل پرووایدر منابع پیش‌فرض در ASP.NET و ذخیره تمامی این ورودی‌ها درون فایل‌های resx، بار نسبتاً زیادی روی حافظه سرور خواهد گذاشت. در صورت مدیریت بهینه فراخوانی‌های سمت دیتابیس می‌توان با بهره‌برداری از جداول یک دیتابیس به عنوان منبع، کمک زیادی به وب سرور کرد! همچنین با استفاده از دیتابیس می‌توان

مدیریت بهتری بر ورودی‌ها داشت و نیز امکان ذخیره‌سازی حجم بیشتری از داده‌ها در اختیار توسعه دهنده قرار خواهد گرفت. البته به غیر از دیتابیس و فایل‌های resx. نیز گزینه‌های دیگری برای ذخیره‌سازی ورودی‌های این منابع وجود دارند. به عنوان مثال می‌توان مدیریت این منابع را کلاً به سیستم دیگری سپرد و درخواست ورودی‌های موردنیاز را به یکسری وب‌سرویس سپرد. برای پیاده‌سازی چنین سیستمی نیاز است تا مدلی سفارشی تهیه و استفاده شود.

- **پیاده‌سازی امکان به روزرسانی منابع در زمان اجرا** - در صورتی که بخواهیم امکان بروزسانی ورودی‌ها را در زمان اجرا در استفاده از فایل‌های resx. داشته باشیم، یکی از راه‌حل‌ها، سفارشی‌سازی این پرووایدرهاست.

مدل پرووایدر منابع

همانطور که قبلاً هم اشاره شد، وظیفه استخراج داده‌ها از Resourceها به صورت پیش‌فرض، در نهایت بر عهده نمونه‌ای از کلاس ResourceManager است. در واقع این کلاس کل فرایند انتخاب مناسب‌ترین کلید از منابع موجود را با توجه به کالچر رابط کاربری (UI Culture) در ثرد جاری کپسوله می‌کند. درباره این کلاس در ادامه بیشتر بحث خواهد شد.

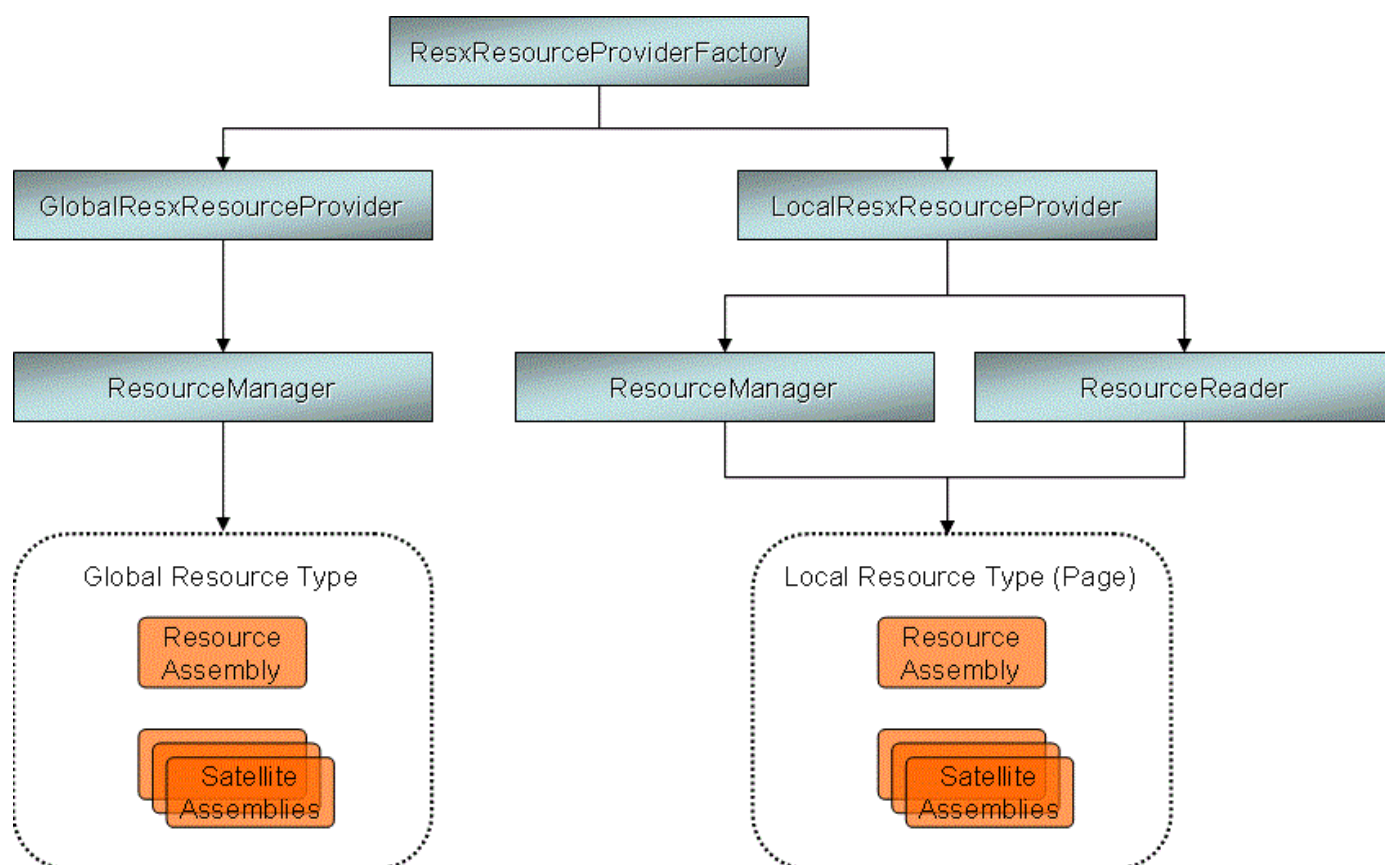
هم‌چنین بازهم همانطور که قبلاً توضیح داده شد، استفاده از ورودی‌های منابع موجود به دو روش انجام می‌شود. استفاده از عبارات بومی‌سازی و نیز با استفاده از برنامه‌نویسی که از طریق دومتد GetLocalResourceObject و GetGlobalResourceObject انجام می‌شود. در ضمن کلیه عبارات بومی‌سازی در زمان رندر صفحات وب در نهایت تبدیل به فراخوانی‌هایی از این دو متد در کلاس TemplateControl خواهند شد.

عملیات پس از فراخوانی این دو متد جایی است که مدل Resource Provider پیش‌فرض ASP.NET وارد کار می‌شود. این فرایند ابتدا با فراخوانی نمونه‌ای از کلاس ResourceProviderFactory آغاز می‌شود که پیاده‌سازی پیش‌فرض آن در کلاس ResXResourceProviderFactory قرار دارد.

این کلاس سپس با توجه به نوع منبع درخواستی (Global یا Local) نمونه‌ای از پرووایدر مربوطه (که باید اینترفیس IResourceProvider را پیاده‌سازی کرده باشند) را تولید می‌کند. پیاده‌سازی پیش‌فرض این پرووایدرها در ASP.NET در کلاس‌های GlobalResXResourceProvider و LocalResXResourceProvider قرار دارد.

این پروایدرها در نهایت باتوجه به محل ورودی درخواستی، نمونه مناسب از کلاس ResourceManager را تولید و استفاده می‌کنند. هم‌چنین در پروایدرهای محلی، برای استفاده از عبارات بومی‌سازی ضمنی، نمونه‌ای از کلاس ResourceReader مورد استفاده قرار می‌گیرد. در زمان تجزیه و تحلیل صفحه وب درخواستی در سرور، با استفاده از این کلاس کلیدهای موردنظر یافته می‌شوند. این کلاس درواقع پیاده‌سازی اینترفیس IResourceReader بوده که حاوی یک Enumerator که جفت داده‌های Key-Value از کلیدهای Resource را برمی‌گرداند، است.

تصویر زیر نمایی کلی از فرایند پیش‌فرض موردبحث را نشان می‌دهد:



این فرایند باتوجه به پیاده سازی نسبتاً جامع آن، قابلیت بسیاری برای توسعه و سفارشی سازی دارد. بنابراین قبل از ادامه مبحث بهتر است، کلاس‌های اصلی این مدل بیشتر شرح داده شوند.

پیاده‌سازی‌ها

کلاس `ResourceProviderFactory` به صورت زیر تعریف شده است:

```
public abstract class ResourceProviderFactory
{
    public abstract IResourceProvider CreateGlobalResourceProvider(string classKey);
    public abstract IResourceProvider CreateLocalResourceProvider(string virtualPath);
}
```

همانطور که مشاهده می‌کنید دو متد برای تولید پرووایدرهای مخصوص منابع کلی و محلی در این کلاس وجود دارد. پرووایدر کلی تنها نیاز به نام کلید `Resource` برای یافتن داده موردنظر دارد. اما پرووایدر محلی به مسیر صفحه درخواستی برای اینکار نیاز دارد که با توجه به توضیحات ابتدای این مطلب کاملاً بدیهی است.

پس از تولید پرووایدر موردنظر با استفاده از متد مناسب با توجه به شرایط شرح داده شده در بالا، نمونه تولیدشده از کلاس پرووایدر موردنظر وظیفه فراهم کردن کلیدهای `Resource` را برعهده دارد. پرووایدرهای موردبحث باید اینترفیس `IResourceProvider` را که به صورت زیر تعریف شده است، پیاده سازی کنند:

```
public interface IResourceProvider
{
    IResourceReader ResourceReader { get; }
    object GetObject(string resourceKey, CultureInfo culture);
}
```

همانطور که می‌بینید این پرووایدرها باید یک `ResourceReader` برای خواندن کلیدهای `Resource` فراهم کنند. همچنین یک متد با عنوان `GetObject` که کار اصلی برگرداندن داده ذخیره‌شده در ورودی موردنظر را برعهده دارد باید در این پرووایدرها پیاده‌سازی

شود. همانطور که قبلا اشاره شد، پیاده‌سازی پیش‌فرض این کلاس‌ها در نهایت نمونه‌ای از کلاس ResourceManager را برای یافتن مناسب‌ترین گزینه از بین کلیدهای موجود تولید می‌کند. این نمونه مورد بحث در متد GetObject مورد استفاده قرار می‌گیرد.

نکته: کدهای نشان‌داده‌شده در ادامه مطلب با استفاده از ابزار محبوب ReSharper استخراج شده‌اند. این ابزار برای دریافت این کدها معمولا از API‌های سایت SymbolSource.org استفاده می‌کند. البته منبع اصلی تمام کدهای دات نت فریمورک همان referencesource.microsoft.com است.

کلاس ResXResourceProviderFactory

پیاده‌سازی پیش‌فرض کلاس ResourceProviderFactory در ASP.NET که در کلاس ResXResourceProviderFactory قرار دارد، به صورت زیر است:

```
// Type: System.Web.Compilation.ResXResourceProviderFactory
// Assembly: System.Web, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
// Assembly location:
C:\Windows\Microsoft.NET\assembly\GAC_32\System.Web\v4.0.0.0__b03f5f7f11d50a3a\System.Web.dll

using System.Runtime;
using System.Web;
namespace System.Web.Compilation
{
    internal class ResXResourceProviderFactory : ResourceProviderFactory
    {
        [TargetedPatchingOptOut("Performance critical to inline this type of method across NGen image boundaries")]
        public ResXResourceProviderFactory() { }
        public override IResourceProvider CreateGlobalResourceProvider(string classKey)
        {
            return (IResourceProvider) new GlobalResXResourceProvider(classKey);
        }
        public override IResourceProvider CreateLocalResourceProvider(string virtualPath)
        {
            return (IResourceProvider) new LocalResXResourceProvider(VirtualPath.Create(virtualPath));
        }
    }
}
```

در این کلاس برای تولید پرووایدر منابع محلی از کلاس VirtualPath استفاده شده است که امکاناتی جهت استخراج مسیرهای موردنظر با توجه به مسیر نسبی و مجازی ارائه‌شده فراهم می‌کند. متاسفانه این کلاس نیز با سطح دسترسی internal تعریف شده است و امکان استفاده مستقیم از آن وجود ندارد.

کلاس GlobalResXResourceProvider

پیاده‌سازی پیش‌فرض اینترفیس IResourceProvider در ASP.NET برای منابع کلی که در کلاس GlobalResXResourceProvider قرار دارد، به صورت زیر است:

```
internal class GlobalResXResourceProvider : BaseResXResourceProvider
{
    private string _classKey;
    internal GlobalResXResourceProvider(string classKey)
    {
        _classKey = classKey;
    }
    protected override ResourceManager CreateResourceManager()
    {
        string fullClassName = BaseResourcesBuildProvider.DefaultResourcesNamespace + "." + _classKey;
        // If there is no app resource assembly, return null
        if (BuildManager.AppResourcesAssembly == null)
            return null;
        ResourceManager resourceManager = new ResourceManager(fullClassName,
            BuildManager.AppResourcesAssembly);
        resourceManager.IgnoreCase = true;
        return resourceManager;
    }
    public override IResourceReader ResourceReader
    {
        get
        {

```

```
// App resources don't support implicit resources, so the IResourceReader should never be needed
throw new NotSupportedException();
}
}
```

در این کلاس عملیات تولید نمونه مناسب از کلاس ResourceManager انجام می‌شود. مقدار BaseResourcesBuildProvider.DefaultResourcesNamespace به صورت زیر تعریف شده است:

```
internal const string DefaultResourcesNamespace = "Resources";
```

که قبلاً هم درباره این مقدار پیش فرض اشاره‌ای شده بود. پارامتر classKey درواقع اشاره به نام فایل اصلی منبع کلی دارد. مثلاً اگر این مقدار برابر Resource1 باشد، کلاس ResourceManager برای نوع داده Resources.Resource1 تولید خواهد شد. هم‌چنین اسمبلی موردنظر برای یافتن ورودی‌های منابع کلی که از BuildManager.AppResourcesAssembly دریافت شده است، به صورت پیش فرض هم‌نام با مسیر منابع کلی و با عنوان App_GlobalResources تولید می‌شود. کلاس BuildManager فرایندهای کامپایل کدها و صفحات برای تولید اسمبلی‌ها و نگهداری از آن‌ها در حافظه را مدیریت می‌کند. این کلاس که محتوای نسبتاً مفصلاً دارد (نزدیک به 2000 خط کد) به صورت public و sealed تعریف شده است. بنابراین با ریفرنس دادن اسمبلی System.Web در فضای نام System.Web.Compilation در دسترس است، اما نمی‌توان کلاسی از آن مشتق کرد. BuildManager حاوی تعداد زیادی اعضای استاتیک برای دسترسی به اطلاعات اسمبلی‌هاست. اما متأسفانه بیشتر آن‌ها سطح دسترسی عمومی ندارند.

نکته: همانطور که در بالا نیز اشاره شد، ازآنجاکه کلاس ResourceReader در اینجا تنها برای عبارات بومی سازی ضمنی کاربرد دارد، و نیز عبارات بومی‌سازی ضمنی تنها برای منابع محلی کاربرد دارند، در این کلاس برای خاصیت مربوطه در پیاده سازی اینترفیس IResourceProvider یک خطای عدم پشتیبانی (NotSupportedException) صادر شده است.

کلاس LocalResXResourceProvider

پیاده‌سازی پیش‌فرض اینترفیس IResourceProvider در ASP.NET برای منابع محلی که در کلاس LocalResXResourceProvider قرار دارد، به صورت زیر است:

```
internal class LocalResXResourceProvider : BaseResXResourceProvider
{
    private VirtualPath _virtualPath;
    internal LocalResXResourceProvider(VirtualPath virtualPath)
    {
        _virtualPath = virtualPath;
    }
    protected override ResourceManager CreateResourceManager()
    {
        ResourceManager resourceManager = null;
        Assembly pageResAssembly = GetLocalResourceAssembly();
        if (pageResAssembly != null)
        {
            string fileName = _virtualPath.FileName;
            resourceManager = new ResourceManager(fileName, pageResAssembly);
            resourceManager.IgnoreCase = true;
        }
        else
        {
            throw new
InvalidOperationException(SR.GetString(SR.ResourceExpresionBuilder_PageResourceNotFound));
        }
        return resourceManager;
    }
    public override IResourceReader ResourceReader
    {
        get
        {
            // Get the local resource assembly for this page
            Assembly pageResAssembly = GetLocalResourceAssembly();
```

```

        if (pageResAssembly == null) return null;
        // Get the name of the embedded .resource file for this page
        string resourceFileName = _virtualPath.FileName + ".resources";
        // Make it lower case, since GetManifestResourceStream is case sensitive
        resourceFileName = resourceFileName.ToLower(CultureInfo.InvariantCulture);
        // Get the resource stream from the resource assembly
        Stream resourceStream = pageResAssembly.GetManifestResourceStream(resourceFileName);
        // If this page has no resources, return null
        if (resourceStream == null) return null;
        return new ResourceReader(resourceStream);
    }
}
[PermissionSet(SecurityAction.Assert, Unrestricted = true)]
private Assembly GetLocalResourceAssembly()
{
    // Remove the page file name to get its directory
    VirtualPath virtualDir = _virtualPath.Parent;
    // Get the name of the local resource assembly
    string cacheKey = BuildManager.GetLocalResourcesAssemblyName(virtualDir);
    BuildResult result = BuildManager.GetBuildResultFromCache(cacheKey);
    if (result != null)
    {
        return ((BuildResultCompiledAssembly)result).ResultAssembly;
    }
    return null;
}
}

```

عملیات موجود در این کلاس باتوجه به فرایندهای مربوط به یافتن اسمبلی مربوطه با استفاده از مسیر ارائه شده، کمی پیچیده تر از کلاس قبلی است.

در متد `GetLocalResourceAssembly` عملیات یافتن اسمبلی متناظر با درخواست جاری انجام می شود. اینکار باتوجه به نحوه نامگذاری اسمبلی منابع محلی که در ابتدای این مطلب اشاره شد انجام می شود. مثلاً اگر صفحه درخواستی در مسیر `~/SubDir1/Page1.aspx` باشد، در این متد با استفاده از ابزارهای موجود عنوان اسمبلی نهایی برای این مسیر که به صورت `App_LocalResources.SubDir1.XXXXX` است تولید و در نهایت اسمبلی مربوطه استخراج می شود. در ضمن در اینجا هم کلاس `ResourceManager` برای نوع داده متناظر با نام فایل اصلی منبع محلی تولید می شود. مثلاً برای مسیر مجازی `~/SubDir1/Page1.aspx` نوع داده ای با نام `Page1.aspx` در نظر گرفته خواهد شد (با توجه به نام فایل منبع محلی که باید به صورت `Page1.aspx.resx` باشد. در [قسمت قبل](#) در این باره شرح داده شده است).

نکته: کلاس `SR` (مخفف `String Resources`) که در فضای نام `System.Web` قرار دارد، حاوی عناوین کلیدهای `Resource` های مورد استفاده در اسمبلی `System.Web` است. این کلاس با سطح دسترسی `internal` و به صورت `sealed` تعریف شده است. عنوان تمامی کلیدها به صورت ثوابتی از نوع رشته تعریف شده اند.

`SR` درواقع یک `Wrapper` بر روی کلاس `ResourceManager` است تا از تکرار عناوین کلیدهای منابع که از نوع رشته هستند، در جاهای مختلف برنامه جلوگیری شود. کار این کلاس مشابه کاری است که کتابخانه [T4MVC](#) برای نگهداری عناوین کنترلرها و اکشنها به صورت رشته های ثابت انجام می دهد. از این روش در جای جای دات نت فریمورک برای نگهداری رشته های ثابت استفاده شده است!

نکته: باتوجه به استفاده از عبارات بومی سازی ضمنی در استفاده از ورودی های منابع محلی، خاصیت `ResourceReader` در این کلاس نمونه ای متناظر برای درخواست جاری از کلاس `ResourceReader` با استفاده از `Stream` استخراج شده از اسمبلی یافته شده، تولید می کند.

کلاس پایه `BaseResXResourceProvider`

کلاس پایه `BaseResXResourceProvider` که در دو پیاده سازی نشان داده شده در بالا استفاده شده است (هر دو کلاس از این کلاس مشتق شده اند)، به صورت زیر است:

```

internal abstract class BaseResXResourceProvider : IResourceProvider
{
    private ResourceManager _resourceManager;
    ///// IResourceProvider implementation
    public virtual object GetObject(string resourceKey, CultureInfo culture)
    {

```



```
// Attempt to get the resource manager
EnsureResourceManager();
// If we couldn't get a resource manager, return null
if (_resourceManager == null) return null;
if (culture == null) culture = CultureInfo.CurrentCulture;
return _resourceManager.GetObject(resourceKey, culture);
}
public virtual IResourceReader ResourceReader { get { return null; } }
///// End of IResourceProvider implementation
protected abstract ResourceManager CreateResourceManager();
private void EnsureResourceManager()
{
    if (_resourceManager != null) return;
    _resourceManager = CreateResourceManager();
}
}
```

در این کلاس پیاده‌سازی اصلی اینترفیس IResourceProvider انجام شده است. همانطور که می‌بینید کار نهایی استخراج ورودی‌های منابع در متد GetObject با استفاده از نمونه فراهم شده از کلاس ResourceManager انجام می‌شود.

نکته: دقت کنید که در کد بالا در صورت فراهم نکردن مقداری برای کالچر، از کالچر UI در ثرد جاری (CultureInfo.CurrentCulture) به عنوان مقدار پیش‌فرض استفاده می‌شود.

کلاس ResourceManager

در زمان اجرا ASP.NET کلید مربوط به منبع موردنظر را با استفاده از کالچر جاری UI انتخاب می‌کند. در [قسمت اول](#) این سری مطالب شرح کوتاهی بابت انواع کالچرها داده شد، اما برای توضیحات کاملتر به [اینجا](#) مراجعه کنید. در ASP.NET به صورت پیش‌فرض تمام منابع در زمان اجرا از طریق نمونه‌ای از کلاس ResourceManager در دسترس خواهند بود. به ازای هر نوع Resource که درخواستی برای یک کلید آن ارسال می‌شود یک نمونه از کلاس ResourceManager ساخته می‌شود. در این هنگام (یعنی پس از اولین درخواست به کلیدهای یک منبع) اسمبلی ستلایت مناسب آن پس از یافته شدن (یا تولید شدن در زمان اجرا) به دامین ASP.NET جاری بارگذاری می‌شود و تا زمانی که این دامین Unload نشود در حافظه سرور باقی خواهد ماند.

نکته: کلاس ResourceManager **تنها** توانایی استخراج کلیدهای Resource از اسمبلی‌های ستلایتی (فایل‌های resources) که در [قسمت اول](#) به آن‌ها اشاره شد) که در AppDomain جاری بارگذاری شده‌اند را دارد.

کلاس ResourceManager به صورت زیر نمونه سازی می‌شود:

```
System.Resources.ResourceManager(string baseName, Assembly assemblyName)
```

پارامتر baseName به نام کامل ریشه اسمبلی اصلی موردنظر (با فضای نام و ...) اما بدون پسوند اسمبلی مربوطه (resources) اشاره دارد. این نام که برابر نام کلاس نهایی تولید شده برای منبع موردنظر است همانم با فایل اصلی و پیش‌فرض منبع (فایلی که حاوی عنوان هیچ زبان و کالچری نیست) تولید می‌شود. مثلاً برای اسمبلی ستلایت با عنوان MyApplication.MyResource.fa-IR.resources باید از عبارت MyApplication.MyResource استفاده شود. پارامتر assemblyName نیز به اسمبلی حاوی اسمبلی ستلایت اصلی اشاره دارد. درواقع همان اسمبلی اصلی که نوع داده مربوط به فایل منبع اصلی درون آن embed شده است. مثلاً:

```
var manager = new System.Resources.ResourceManager("Resources.Resource1", typeof(Resource1).Assembly)
```

یا

```
var manager = new System.Resources.ResourceManager("Resources.Resource1",
Assembly.LoadFile(@"c:\MyResources\MyGlobalResources.dll"))
```

روش دیگری نیز برای تولید نمونه‌ای از این کلاس وجود دارد که با استفاده از متد استاتیک زیر که در خود کلاس ResourceManager تعریف شده است انجام می‌شود:

```
public static ResourceManager CreateFileBasedResourceManager(string baseName, string resourceDir, Type usingResourceSet)
```

در این متد کار استخراج ورودی‌های منابع مستقیماً از فایل‌های resources انجام می‌شود. در اینجا baseName نام فایل اصلی منبع بدون پیشوند resources است. resourceDir نیز مسیری است که فایل‌های resources در آن قرار دارند. usingResourceSet نیز نوع کلاس سفارشی سازی شده از ResourceSet برای استفاده به جای کلاس پیش‌فرض است که معمولاً مقدار null برای آن وارد می‌شود تا از همان کلاس پیش‌فرض استفاده شود (چون برای بیشتر نیازها همین کلاس پیش‌فرض کفایت می‌کند).
نکته: برای تولید فایل resources از یک فایل resx میتوان از ابزار resgen همانند زیر استفاده کرد:

```
resgen d:\MyResources\MyResource.fa.resx
```

نکته: عملیاتی که درون کلاس ResourceManager انجام می‌شود پیچیده‌تر از آن است که به نظر می‌آید. این عملیات شامل فرایندهای بسیاری شامل بارگذاری کلیدهای مختلف یافته شده و مدیریت ذخیره موقت آن‌ها در حافظه (کش)، کنترل و مدیریت انواع Resource Set ها، و مهمتر از همه مدیریت عملیات Fallback و ... که در نهایت شامل هزاران خط کد است که با یک جستجوی ساده قابل مشاهده و بررسی است ([^](#)).

نمونه‌سازی مناسب از ResourceManager

در کدهای نشان داده شده در بالا برای پیاده‌سازی پیش‌فرض در ASP.NET، مهمترین نکته همان تولید نمونه مناسب از کلاس ResourceManager است. پس از آماده شدن این کلاس عملیات استخراج ورودی‌های منابع بر راحتی و با مدیریت کامل انجام می‌شود. اما از آنجاکه تقریباً تمامی API های مورد نیاز با سطح دسترسی internal تعریف شده‌اند، متأسفانه تهیه و تولید این نمونه مناسب خارج از اسمبلی System.Web به صورت مستقیم وجود ندارد.
در هر صورت، برای آشنایی بیشتر با فرایند نشان داده شده، تولید این نمونه مناسب و استفاده مستقیم از آن می‌تواند مفید و نیز جالب باشد. پس از کمی تحقیق و با استفاده از Reflection به کدهای زیر رسیدم:

```
private ResourceManager CreateGlobalResourceManager(string classKey)
{
    var baseName = "Resources." + classKey;
    var buildManagerType = typeof(BuildManager);
    var property = buildManagerType.GetProperty("AppResourcesAssembly", BindingFlags.Static |
BindingFlags.NonPublic | BindingFlags.GetField);
    var appResourcesAssembly = (Assembly)property.GetValue(null, null);
    return new ResourceManager(baseName, appResourcesAssembly) { IgnoreCase = true };
}
```

تنها نکته کد فوق دسترسی به اسمبلی منابع کلی در خاصیت AppResourcesAssembly از کلاس BuildManager با استفاده از BindingFlags های نشان داده شده است. نحوه استفاده از این متد هم به صورت زیر است:

```
var manager = CreateGlobalResourceManager("Resource1");
Label1.Text = manager.GetString("String1");
```

اما برای منابع محلی کار کمی پیچیده‌تر است. کد مربوط به تولید نمونه مناسب از ResourceManager برای منابع محلی به صورت زیر خواهد بود:

```
private ResourceManager CreateLocalResourceManager(string virtualPath)
{
    var virtualPathType = typeof(BuildManager).Assembly.GetType("System.Web.VirtualPath", true);
    var virtualPathInstance = Activator.CreateInstance(virtualPathType, BindingFlags.NonPublic |
BindingFlags.Instance, null, new object[] { virtualPath }, CultureInfo.InvariantCulture);
    var buildResultCompiledAssemblyType =
```

```
typeof(BuildManager).Assembly.GetType("System.Web.Compilation.BuildResultCompiledAssembly", true);
var propertyResultAssembly = buildResultCompiledAssemblyType.GetProperty("ResultAssembly",
BindingFlags.NonPublic | BindingFlags.Instance);
var methodGetLocalResourcesAssemblyName =
typeof(BuildManager).GetMethod("GetLocalResourcesAssemblyName", BindingFlags.NonPublic |
BindingFlags.Static);
var methodGetBuildResultFromCache = typeof(BuildManager).GetMethod("GetBuildResultFromCache",
BindingFlags.NonPublic | BindingFlags.Static, null, new Type[] { typeof(string) }, null);

var fileNameProperty = virtualPathType.GetProperty("FileName");
var virtualPathFileName = (string)fileNameProperty.GetValue(virtualPathInstance, null);

var parentProperty = virtualPathType.GetProperty("Parent");
var virtualPathParent = parentProperty.GetValue(virtualPathInstance, null);

var localResourceAssemblyName = (string)methodGetLocalResourcesAssemblyName.Invoke(null, new object[]
{ virtualPathParent });
var buildResultFromCache = methodGetBuildResultFromCache.Invoke(null, new object[] {
localResourceAssemblyName });
Assembly localResourceAssembly = null;
if (buildResultFromCache != null)
    localResourceAssembly = (Assembly)propertyResultAssembly.GetValue(buildResultFromCache, null);

if (localResourceAssembly == null)
    throw new InvalidOperationException("Unable to find the matching resource file.");

return new ResourceManager(virtualPathFileName, localResourceAssembly) { IgnoreCase = true };
}
```

ازجمله نکات مهم این متد تولید یک نمونه از کلاس VirtualPath برای Parse کردن مسیر مجازی وارد شده برای صفحه درخواستی است. از این کلاس برای بدست آوردن نام فایل منبع محلی به همراه مسیر فولدر مربوطه جهت استخراج اسمبلی متناظر استفاده میشود.

نکته مهم دیگر این کد دسترسی به متد GetLocalResourcesAssemblyName از کلاس BuildManager است که با استفاده از مسیر فولدر مربوط به صفحه درخواستی نام اسمبلی منبع محلی مربوطه را برمی گرداند.

درنهایت با استفاده از متد GetBuildResultFromCache از کلاس BuildManager اسمبلی موردنظر بدست می آید. همانطور که از نام این متد برمی آید این اسمبلی از کش خوانده می شود. البته مدیریت این اسمبلی ها کاملاً توسط BuildManager و سایر ابزارهای موجود در ASP.NET انجام خواهد شد.

نحوه استفاده از متد فوق نیز به صورت زیر است:

```
var manager = CreateLocalResourceManager("~/Default.aspx");
Label1.Text = manager.GetString("Label1.Text");
```

نکته: ارائه و شرح کدهای پیاده سازی های پیش فرض برای آشنایی با نحوه صحیح سفارشی سازی این کلاس ها آورده شده است. پس با دقت بیشتر بر روی این کدها سعی کنید نحوه پیاده سازی مناسب را برای سفارشی سازی موردنظر خود پیدا کنید.

تا اینجا با مقدمات فرایند تولید پرووایدرهای سفارشی برای استفاده در فرایند بارگذاری ورودی های Resource ها آشنا شدیم. در ادامه به بحث تولید پرووایدرهای سفارشی برای استفاده از دیگر انواع منابع (به غیر از فایل های .resx) خواهیم پرداخت.

منابع: <http://msdn.microsoft.com/en-us/library/aa905797.aspx>

<http://msdn.microsoft.com/en-us/library/ms227427.aspx> <http://www.westwind.com/presentations/wfdbresourceprovider>

<http://www.codeproject.com/Articles/104667/Under-the-Hood-of-BuildManager-and-Resource-Handli>

<http://www.onpreinit.com/2009/06/updatable-aspnet-resx-resource-provider.html> [http://msdn.microsoft.com/en-us/library/h6270d0z\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/h6270d0z(v=vs.100).aspx)

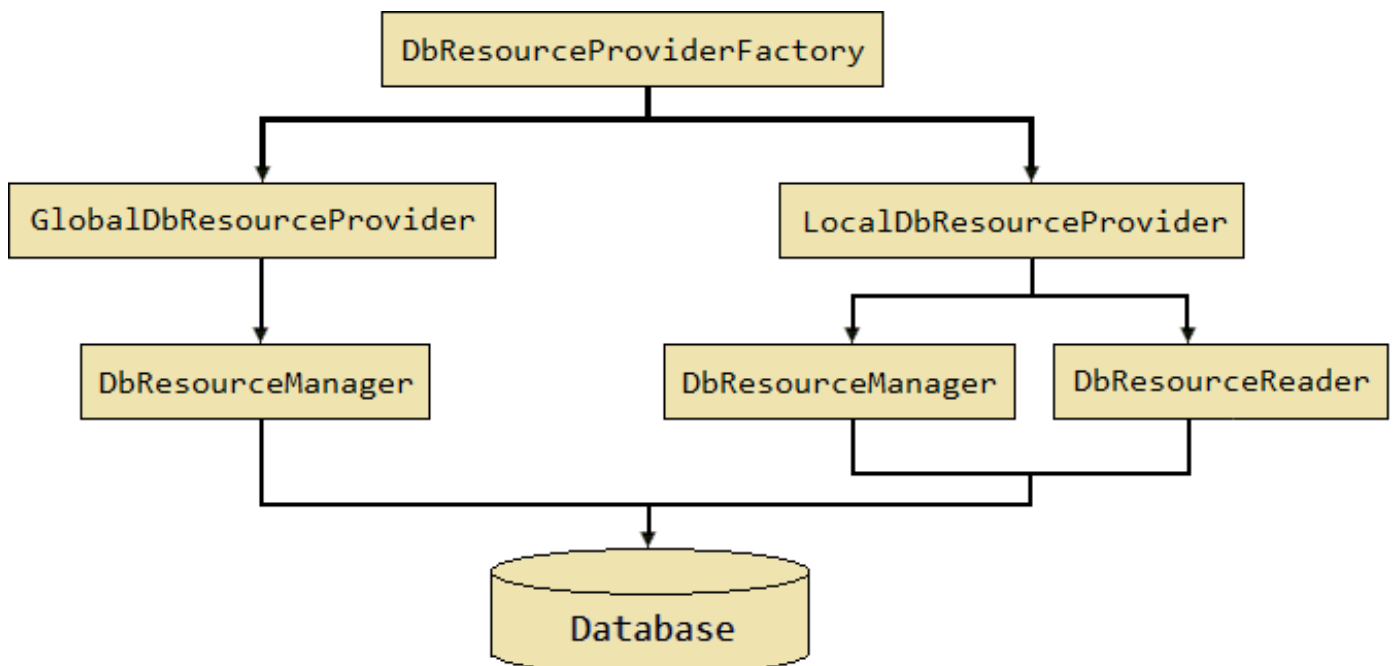
<http://msdn.microsoft.com/en-us/library/system.web.compilation.resourceproviderfactory.aspx>

در [قسمت قبل](#) راجع به مدل پیش‌فرض پرووایدر منابع در ASP.NET بحث نسبتاً مفصلاً شد. در این قسمت تولید یک پرووایدر سفارشی برای استفاده از دیتابیس به جای فایل‌های resx. به عنوان منبع نگهداری داده‌ها بحث می‌شود. قبلاً هم اشاره شده بود که در پروژه‌های بزرگ ذخیره تمام ورودی‌های منابع درون فایل‌های resx. بازدهی مناسبی نخواهد داشت. همچنین به مرور زمان و با افزایش تعداد این فایل‌ها، کار مدیریت آن‌ها بسیار دشوار و طاقت‌فرسا خواهد شد. درضمن به دلیل رفتار سیستم کشینگ این منابع در ASP.NET، که محتویات کل یک فایل را بلافاصله پس از اولین درخواست یکی از ورودی‌های آن در حافظه سرور کش می‌کند، در صورت وجود تعداد زیادی فایل منبع و با ورودی‌های بسیار، با گذشت زمان بازدهی کلی سایت به شدت تحت تأثیر قرار خواهد گرفت.

بنابراین استفاده از یک منبع مثل دیتابیس برای چنین شرایطی و نیز کنترل مدیریت دسترسی به ورودی‌های آن به صورت سفارشی، می‌تواند به بازدهی بهتر برنامه کمک زیادی کند. درضمن فرایند به‌روزرسانی مقادیر این ورودی‌ها در صورت استفاده از یک دیتابیس می‌تواند ساده‌تر از حالت استفاده از فایل‌های resx. انجام شود.

تولید یک پرووایدر منابع دیتابیسی - بخش اول

در بخش اول این مطلب با نحوه پیاده‌سازی کلاس‌های اصلی و اولیه موردنیاز آشنا خواهیم شد. مفاهیم پیشرفته‌تر (مثل کش کردن ورودی‌ها و عملیات fallback) و نیز ساختار مناسب جدول یا جداول موردنیاز در دیتابیس و نحوه ذخیره ورودی‌ها برای انواع منابع در دیتابیس در مطلب بعدی آورده می‌شود. با توجه به توضیحاتی که در [قسمت قبل](#) داده شد، می‌توان از طرح اولیه‌ای به صورت زیر برای سفارشی‌سازی یک پرووایدر منابع دیتابیسی استفاده کرد:



اگر مطالب قسمت قبل را خوب مطالعه کرده باشید، پیاده‌سازی اولیه طرح بالا نباید کار سختی باشد. در ادامه یک نمونه از

پیاده‌سازی‌های ممکن نشان داده شده است.

برای آغاز کار ابتدا یک پروژه ClassLibrary جدید مثلاً با نام DbResourceProvider ایجاد کنید و رفرنسی از اسمبلی System.Web به این پروژه اضافه کنید. سپس کلاس‌هایی که در ادامه شرح داده شده‌اند را به آن اضافه کنید.

کلاس DbResourceProviderFactory

همه چیز از یک ResourceProviderFactory شروع می‌شود. نسخه سفارشی نشان داده شده در زیر برای منابع محلی و کلی از کلاس‌های پرووایدر سفارشی استفاده می‌کند که در ادامه آورده شده‌اند.

```
using System.Web.Compilation;
namespace DbResourceProvider
{
    public class DbResourceProviderFactory : ResourceProviderFactory
    {
        #region Overrides of ResourceProviderFactory
        public override IResourceProvider CreateGlobalResourceProvider(string classKey)
        {
            return new GlobalDbResourceProvider(classKey);
        }
        public override IResourceProvider CreateLocalResourceProvider(string virtualPath)
        {
            return new LocalDbResourceProvider(virtualPath);
        }
        #endregion
    }
}
```

درباره اعضای کلاس ResourceProviderFactory در [قسمت قبل](#) توضیحاتی داده شد. در نمونه سفارشی بالا دو متد این کلاس برای برگرداندن پرووایدرهای سفارشی منابع محلی و کلی بازنویسی شده‌اند. سعی شده است تا نمونه‌های سفارشی در اینجا رفتاری همانند نمونه‌های پیش‌فرض در ASP.NET داشته باشند، بنابراین برای پرووایدر منابع کلی (GlobalDbResourceProvider) نام منبع درخواستی (className) و برای پرووایدر منابع محلی (LocalDbResourceProvider) مسیر مجازی درخواستی (virtualPath) به عنوان پارامتر کانستراکتور ارسال می‌شود.

نکته: برای استفاده از این کلاس به جای کلاس پیش‌فرض ASP.NET باید یکسری تنظیمات در فایل کانفیگ برنامه مقصد اعمال کرد که در ادامه آورده شده است.

کلاس BaseDbResourceProvider

برای پیاده‌سازی راحت‌تر کلاس‌های موردنظر، بخش‌های مشترک بین دو پرووایدر محلی و کلی در یک کلاس پایه به صورت زیر قرار داده شده است. این طرح دقیقاً مشابه نمونه پیش‌فرض ASP.NET است.

```
using System.Globalization;
using System.Resources;
using System.Web.Compilation;
namespace DbResourceProvider
{
    public abstract class BaseDbResourceProvider : IResourceProvider
    {
        private DbResourceManager _resourceManager;
        protected abstract DbResourceManager CreateResourceManager();
        private void EnsureResourceManager()
        {
            if (_resourceManager != null) return;
            _resourceManager = CreateResourceManager();
        }
        #region Implementation of IResourceProvider
        public object GetObject(string resourceKey, CultureInfo culture)
        {
            EnsureResourceManager();
            if (_resourceManager == null) return null;
            if (culture == null) culture = CultureInfo.CurrentUICulture;
            return _resourceManager.GetObject(resourceKey, culture);
        }
        public virtual IResourceReader ResourceReader { get { return null; } }
        #endregion
    }
}
```

کلاس بالا چون یک کلاس صرفاً پایه است بنابراین به صورت abstract تعریف شده است. در این کلاس، از نمونه سفارشی DbResourceManager برای بازیابی داده‌ها از دیتابیس استفاده شده است که در ادامه شرح داده شده است. در اینجا، از متد CreateResourceManager برای تولید نمونه مناسب از کلاس DbResourceManager استفاده می‌شود. این متد به صورت abstract و protected تعریف شده است بنابراین پیاده‌سازی آن باید در کلاس‌های مشتق شده که در ادامه آورده شده‌اند انجام شود.

در متد EnsureResourceManager کار بررسی نال نبودن _resourceManager انجام می‌شود تا در صورت نال بودن آن، بلافاصله نمونه‌ای تولید شود.

نکته: از آنجاکه نقطه آغازین فرایند یعنی تولید نمونه‌ای از کلاس DbResourceProviderFactory توسط خود ASP.NET انجام خواهد شد، بنابراین مدیریت تمام نمونه‌های ساخته شده از کلاس‌هایی که در این مطلب شرح داده می‌شوند در نهایت عملاً برعهده ASP.NET است. در ASP.NET در طول عمر یک برنامه تنها یک نمونه از کلاس Factory تولید خواهد شد، و متدهای موجود در آن در حالت عادی تنها یکبار به ازای هر منبع درخواستی (کلی یا محلی) فراخوانی می‌شوند. در نتیجه به ازای هر منبع درخواستی (کلی یا محلی) هر یک از کلاس‌های پرووایدر منابع تنها یکبار نمونه‌سازی خواهد شد. بنابراین بررسی نال نبودن این متغیر و تولید نمونه‌ای جدید تنها در صورت نال بودن آن، کاری منطقی است. این نمونه بعداً توسط ASP.NET به ازای هر منبع یا صفحه درخواستی کش می‌شود تا در درخواست‌های بعدی تنها از این نسخه کش‌شده استفاده شود.

در متد GetObject نیز کار استخراج ورودی منابع انجام می‌شود. ابتدا با استفاده از متد EnsureResourceManager از وجود نمونه‌ای از کلاس DbResourceManager اطمینان حاصل می‌شود. سپس در صورتی که مقدار این کلاس همچنان نال باشد مقدار نال برگشت داده می‌شود. این حالت وقتی پیش می‌آید که نتوان با استفاده از داده‌های موجود نمونه‌ای مناسب از کلاس DbResourceManager تولید کرد.

سپس مقدار کالچر ورودی بررسی می‌شود و در صورتی که نال باشد مقدار کالچر UI ثرد جاری که در CultureInfo.CurrentCulture قرار دارد برای آن در نظر گرفته می‌شود. در نهایت با فراخوانی متد GetObject از DbResourceManager تولیدی برای کلید و کالچر مربوطه کار استخراج ورودی درخواستی پایان می‌پذیرد. پراپرتی ResourceReader در این کلاس به صورت virtual تعریف شده است تا بتوان پیاده‌سازی مناسب آن را در هر یک از کلاس‌های مشتق‌شده اعمال کرد. فعلاً برای این کلاس پایه مقدار نال برگشت داده می‌شود.

کلاس GlobalDbResourceProvider

برای پرووایدر منابع کلی از این کلاس استفاده می‌شود. نحوه پیاده‌سازی آن نیز دقیقاً همانند طرح نمونه پیش‌فرض ASP.NET است.

```
using System;
using System.Resources;
namespace DbResourceProvider
{
    public class GlobalDbResourceProvider : BaseDbResourceProvider
    {
        private readonly string _classKey;
        public GlobalDbResourceProvider(string classKey)
        {
            _classKey = classKey;
        }
        #region Implementation of BaseDbResourceProvider
        protected override DbResourceManager CreateResourceManager()
        {
            return new DbResourceManager(_classKey);
        }
        public override IResourceReader ResourceReader
        {
            get { throw new NotSupportedException(); }
        }
        #endregion
    }
}
```

GlobalDbResourceProvider از کلاس پایه‌ای که در بالا شرح داده شد مشتق شده است. بنابراین تنها بخش‌های موردنیاز یعنی متد CreateResourceManager و پراپرتی ResourceReader در این کلاس پیاده‌سازی شده است.

در اینجا نمونه مخصوص کلاس ResourceManager (همان DbResourceManager) با توجه به نام فایل مربوط به منبع کلی تولید می‌شود. نام فایل در اینجا همان چیزی است که در دیتابیس برای نام منبع مربوطه ذخیره می‌شود. ساختار آن بعداً بحث می‌شود.

همان‌طور که می‌بینید برای پراپرتی ResourceReader خطای عدم پشتیبانی صادر می‌شود. دلیل آن در [قسمت قبل](#) و نیز به صورت کمی دقیق‌تر در ادامه آورده شده است.

کلاس LocalDbResourceProvider

برای منابع محلی نیز از طرحی مشابه نمونه پیش‌فرض ASP.NET که در [قسمت قبل](#) نشان داده شد، استفاده شده است.

```
using System.Resources;
namespace DbResourceProvider
{
    public class LocalDbResourceProvider : BaseDbResourceProvider
    {
        private readonly string _virtualPath;
        public LocalDbResourceProvider(string virtualPath)
        {
            _virtualPath = virtualPath;
        }
        #region Implementation of BaseDbResourceProvider
        protected override DbResourceManager CreateResourceManager()
        {
            return new DbResourceManager(_virtualPath);
        }
        public override IResourceReader ResourceReader
        {
            get { return new DbResourceReader(_virtualPath); }
        }
        #endregion
    }
}
```

این کلاس نیز از کلاس پایه‌ای BaseDbResourceProvider مشتق شده و پیاده‌سازی‌های مخصوص منابع محلی برای متد CreateResourceManager و پراپرتی ResourceReader در آن انجام شده است. در متد CreateResourceManager کار تولید نمونه‌ای از DbResourceManager با استفاده از مسیر مجازی صفحه درخواستی انجام می‌شود. این فرایند شبیه به پیاده‌سازی پیش‌فرض ASP.NET است. در واقع در پیاده‌سازی جاری، نام منابع محلی همان‌ام با مسیر مجازی متناظر آن‌ها در دیتابیس ذخیره می‌شود. درباره ساختار جدول دیتابیس بعداً بحث می‌شود. در این کلاس کار بازخوانی کلیدهای موجود برای پراپرتی‌های موجود در یک صفحه از طریق نمونه‌ای از کلاس DbResourceReader انجام شده است. شرح این کلاس در ادامه آمده است.

نکته: همان‌طور که در [قسمت قبل](#) هم اشاره کوتاهی شده بود، از خاصیت ResourceReader در پرووایدر منابع برای تعیین تمام پراپرتی‌های موجود در منبع استفاده می‌شود تا کار جستجوی کلیدهای موردنیاز در عبارات بومی‌سازی **ضمنی** برای رندر صفحه وب راحت‌تر انجام شود. بنابراین از این پراپرتی تنها در پرووایدر منابع **محلی** استفاده می‌شود. از آنجاکه در عبارات بومی‌سازی **ضمنی** تنها قسمت اول نام کلید ورودی منبع آورده می‌شود، بنابراین قسمت دوم (و یا قسمت‌های بعدی) کلید موردنظر که همان نام پراپرتی کنترل متناظر است از جستجو میان ورودی‌های یافته شده توسط این پراپرتی بدست می‌آید تا ASP.NET بداند که برای رندر صفحه چه پراپرتی‌هایی نیاز به رجوع به پرووایدر منبع محلی مربوطه دارد (برای آشنایی بیشتر با عبارت بومی‌سازی **ضمنی** رجوع شود به [قسمت قبل](#)).

نکته: دقت کنید که پس از اولین درخواست، خروجی حاصل از enumerator این ResourceReader کش می‌شود تا در درخواست‌های بعدی از آن استفاده شود. بنابراین در حالت عادی، به ازای هر صفحه تنها یکبار این پراپرتی فراخوانده می‌شود. درباره این enumerator در ادامه بحث شده است.

کلاس DbResourceManager

کار اصلی مدیریت و بازیابی ورودی‌های منابع از دیتابیس از طریق کلاس DbResourceManager انجام می‌شود. نمونه‌ای بسیار ساده

و اولیه از این کلاس را در زیر مشاهده می‌کنید:

```
using System.Globalization;
using DbResourceProvider.Data;
namespace DbResourceProvider
{
    public class DbResourceManager
    {
        private readonly string _resourceName;
        public DbResourceManager(string resourceName)
        {
            _resourceName = resourceName;
        }
        public object GetObject(string resourceKey, CultureInfo culture)
        {
            var data = new ResourceData();
            return data.GetResource(_resourceName, resourceKey, culture.Name).Value;
        }
    }
}
```

کار استخراج ورودی‌های منابع با استفاده از نام منبع درخواستی در این کلاس مدیریت خواهد شد. این کلاس با استفاده نام منبع درخواستی به عنوان پارامتر کانستراکتور ساخته می‌شود. با استفاده از متد `GetObject` که نام کلید ورودی موردنظر و کالچر مربوطه را به عنوان پارامتر ورودی دریافت می‌کند فرایند استخراج انجام می‌شود. برای کپسوله‌سازی عملیات از کلاس جداگانه‌ای (`ResourceData`) برای تبادل با دیتابیس استفاده شده است. شرح بیشتر درباره این کلاس و نیز پیاده سازی کامل‌تر کلاس `DbResourceManager` به همراه مدیریت کش ورودی‌های منابع و نیز عملیات `fallback` در مطلب بعدی آورده می‌شود.

کلاس `DbResourceReader`

این کلاس که درواقع پیاده‌سازی اینترفیس `IResourceReader` است برای یافتن تمام کلیدهای تعریف شده برای یک منبع به‌کار می‌رود، پیاده‌سازی آن نیز به صورت زیر است:

```
using System.Collections;
using System.Resources;
using System.Security;
using DbResourceProvider.Data;
namespace DbResourceProvider
{
    public class DbResourceReader : IResourceReader
    {
        private readonly string _resourceName;
        private readonly string _culture;
        public DbResourceReader(string resourceName, string culture = "")
        {
            _resourceName = resourceName;
            _culture = culture;
        }
        #region Implementation of IResourceReader
        public void Close() { }
        public IDictionaryEnumerator GetEnumerator()
        {
            return new DbResourceEnumerator(new ResourceData().GetResources(_resourceName, _culture));
        }
        #endregion
        #region Implementation of IEnumerable
        IEnumerator IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }
        #endregion
        #region Implementation of IDisposable
        public void Dispose()
        {
            Close();
        }
        #endregion
    }
}
```

این کلاس تنها با استفاده از نام منبع و عنوان کالچر موردنظر کار بازخوانی ورودی‌های موجود را انجام می‌دهد.

تنها نکته مهم در کد بالا متد GetEnumerator است که نمونه‌ای از اینترفیس IDictionaryEnumerator را برمی‌گرداند. در اینجا از کلاس DbResourceEnumerator که برای کار با دیتابیس طراحی شده، استفاده شده است. همانطور که قبلاً هم اشاره شده بود، هر یک از اعضای این enumerator از نوع DictionaryEntry هستند که یک struct است. این کلاس در ادامه شرح داده شده است. متد Close برای بستن و از بین بردن منابعی است که در تهیه enumerator مورد بحث نقش داشته‌اند. مثل منابع شبکه‌ای یا فایلی که باید قبل از اتمام کار با این کلاس به صورت کامل بسته شوند. هرچند در نمونه جاری چنین موردی وجود ندارد و بنابراین این متد بلااستفاده است. در کلاس فوق نیز برای دریافت اطلاعات از ResourceData استفاده شده است که بعداً به همراه ساختار مناسب جدول دیتابیس شرح داده می‌شود.

نکته: دقت کنید که در پیاده‌سازی نشان داده شده برای کلاس LocalDbResourceProvider برای یافتن ورودی‌های موجود از مقدار پیش‌فرض (یعنی رشته خالی) برای کالچر استفاده شده است تا از ورودی‌های پیش‌فرض که در حالت عادی باید شامل تمام موارد تعریف شده موجود هستند استفاده شود (قبلاً هم شرح داده شد که منبع اصلی و پیش‌فرض یعنی همانی که برای زبان پیش‌فرض برنامه در نظر گرفته می‌شود و بدون نام کالچر مربوطه است، باید شامل حداکثر ورودی‌های تعریف شده باشد. منابع مربوطه به سایر کالچرها می‌توانند همه این ورودی‌های تعریف شده در منبع اصلی و یا قسمتی از آن را شامل شوند. عملیات fallback تضمین می‌دهد که در نهایت نزدیک‌ترین گزینه متناظر با درخواست جاری را برگشت دهد).

کلاس DbResourceEnumerator

کلاس دیگری که در اینجا استفاده شده است، DbResourceEnumerator است. این کلاس در واقع پیاده‌سازی اینترفیس IDictionaryEnumerator است. محتوای این کلاس در زیر آورده شده است:

```
using System.Collections;
using System.Collections.Generic;
using DbResourceProvider.Models;
namespace DbResourceProvider
{
    public sealed class DbResourceEnumerator : IDictionaryEnumerator
    {
        private readonly List<Resource> _resources;
        private int _dataPosition;
        public DbResourceEnumerator(List<Resource> resources)
        {
            _resources = resources;
            Reset();
        }
        public DictionaryEntry Entry
        {
            get
            {
                var resource = _resources[_dataPosition];
                return new DictionaryEntry(resource.Key, resource.Value);
            }
        }
        public object Key { get { return Entry.Key; } }
        public object Value { get { return Entry.Value; } }
        public object Current { get { return Entry; } }
        public bool MoveNext()
        {
            if (_dataPosition >= _resources.Count - 1) return false;
            ++_dataPosition;
            return true;
        }
        public void Reset()
        {
            _dataPosition = -1;
        }
    }
}
```

تفاوت این اینترفیس با IEnumerable در سه عضو اضافی است که برای استفاده در سیستم مدیریت منابع ASP.NET نیاز است. همان‌طور که در کد بالا مشاهده می‌کنید این سه عضو عبارتند از پراپرتی‌های Entry و Key و Value. پراپرتی Entry که ورودی جاری در enumerator را مشخص می‌کند از نوع DictionaryEntry است. پراپرتی‌های Key و Value هم که از نوع object تعریف شده‌اند برای کلید و مقدار ورودی جاری استفاده می‌شوند. این کلاس لیستی از Resource به عنوان پارامتر کانستراکتور برای تولید enumerator دریافت می‌کند. کلاس Resource مدل تولیدی از ساختار جدول دیتابیس برای ذخیره ورودی‌های منابع است که در مطلب بعدی شرح داده می‌شود. بقیه قسمت‌های کد فوق هم پیاده‌سازی معمولی یک enumerator است.

نکته: به جای تعریف کلاس جداگانه‌ای برای enumerator اینترفیس IResourceProvider می‌توان از enumerator کلاس‌هایی که IDictionary را پیاده‌سازی کرده‌اند نیز استفاده کرد، مانند کلاس Dictionary<object,object> یا ListDictionary.

تنظیمات فایل کانفیگ

برای اجبار کردن ASP.NET به استفاده از Factory موردنظر باید تنظیمات زیر را در فایل web.config اعمال کرد:

```
<system.web>
  <...
  <globalization resourceProviderFactoryType="نام پرووایدر فکتوری به همراه نام کامل اسمبلی مربوطه" />
  <...
</system.web>
```

روش نشان داده شده در بالا حالت کلی تعریف و تنظیم یک نوع داده در فایل کانفیگ را نشان می‌دهد. درباره نام کامل اسمبلی در [اینجا](#) شرح داده شده است. مثلاً برای پیاده‌سازی نشان داده شده در این مطلب خواهیم داشت:

```
<globalization resourceProviderFactoryType="DbResourceProvider.DbResourceProviderFactory,
DbResourceProvider" />
```

در مطلب بعدی درباره ساختار مناسب جدول یا جداول دیتابیس برای ذخیره ورودهای منابع و نیز پیاده‌سازی کامل‌تر کلاس‌های مورد استفاده بحث خواهد شد.

منابع: <http://msdn.microsoft.com/en-us/library/aa905797.aspx>

<http://msdn.microsoft.com/en-us/library/ms227427.aspx> <http://www.westwind.com/presentations/wfdbresourceprovider>

<http://www.onpreinit.com/2009/06/updatable-aspnet-resx-resource-provider.html>

<http://msdn.microsoft.com/en-us/library/system.web.compilation.resourceproviderfactory.aspx>

<http://www.codeproject.com/Articles/14190/ASP-NET-2-0-Custom-SQL-Server-ResourceProvider>

<http://www.codeproject.com/Articles/104667/Under-the-Hood-of-BuildManager-and-Resource-Handli>

نظرات خوانندگان

نویسنده: ابوالفضل رجب پور

تاریخ: ۱۱:۲۲ ۱۳۹۲/۰۳/۰۶

سلام جناب یوسف نژاد
برای پروژه م می‌خوام از روند شما استفاده کنم. بی صبرانه منتظر قسمت بعدی هستم
تشکر


در [قسمت قبل](#) ساختار اصلی و پیاده‌سازی ابتدایی یک پرووایدر سفارشی دیتابیزی شرح داده شد. در این قسمت ادامه بحث و مطالب پیشرفته‌تر آورده شده است.

تولید یک پرووایدر منابع دیتابیزی - بخش دوم

در بخش دوم این سری مطلب، ساختار دیتابیس و مباحث پیشرفته پیاده‌سازی کلاس‌های نشان داده‌شده در بخش اول در [قسمت قبل](#) شرح داده می‌شود. این مباحث شامل نحوه کش صحیح و بهینه داده‌های دریافتی از دیتابیس، پیاده‌سازی فرایند fallback، و پیاده‌سازی مناسب کلاس DbResourceManager برای مدیریت کل عملیات است.

ساختار دیتابیس

برای پیاده‌سازی منابع دیتابیزی روش‌های مختلفی برای آرایش جداول جهت ذخیره انواع ورودی‌ها می‌توان در نظر گرفت. مثلاً در صورتی که حجم و تعداد منابع بسیار باشد و نیز منابع دیتابیزی به اندازه کافی در دسترس باشد، می‌توان به ازای هر منبع یک جدول در نظر گرفت. یا در صورتیکه منابع داده‌ای محدودتر باشند می‌توان به ازای هر کالچر یک جدول در نظر گرفت و تمام منابع مربوط به یک کالچر را درون یک جدول ذخیره کرد. در هر صورت نحوه انتخاب آرایش جداول منابع کاملاً بستگی به شرایط کاری و سلايق برنامه‌نویسی دارد. برای مطلب جاری به عنوان یک راه‌حل ساده و کارآمد برای پروژه‌های کوچک و متوسط، تمام ورودی‌های منابع درون یک جدول با ساختاری مانند زیر ذخیره می‌شود:

	Column Name	Data Type	Allow Nulls
	Id	bigint	<input type="checkbox"/>
	Name	nvarchar(200)	<input type="checkbox"/>
	[Key]	nvarchar(200)	<input type="checkbox"/>
	Culture	nvarchar(6)	<input type="checkbox"/>
	Value	nvarchar(MAX)	<input type="checkbox"/>

نام این جدول را با در نظر گرفتن شرایط موجود می‌توان Resources گذاشت.

ستون Name برای ذخیره نام منبع در نظر گرفته شده است. این نام برابر نام منابع درخواستی در سیستم مدیریت منابع ASP.NET است که در واقع برابر همان نام فایل منبع اما بدون پسوند .resx است.

ستون Key برای نگهداری کلید ورودی منبع استفاده می‌شود که دقیقاً برابر همان مقداری است که درون فایل‌های .resx ذخیره می‌شود.

ستون Culture برای ذخیره کالچر ورودی منبع به کار می‌رود. این مقدار می‌تواند برای کالچر پیش‌فرض برنامه برابر رشته خالی باشد.

ستون Value نیز برای نگهداری مقدار ورودی منبع استفاده می‌شود.

برای ستون Id می‌توان از GUID نیز استفاده کرد. در اینجا برای راحتی کار از نوع داده bigint و خاصیت Identity برای تولید خودکار آن در Sql Server استفاده شده است.

نکته: برای امنیت بیشتر می‌توان یک Unique Constraint بر روی سه فیلد Name و Key و Culture اعمال کرد.

برای نمونه به تصویر زیر که ذخیره تعدادی ورودی منبع را درون جدول Resources نمایش می‌دهد دقت کنید:

Id	Name	Key	Culture	Value
1	GlobalTexts	Yes		yesssss
2	GlobalTexts	Yes	fa	بله
3	GlobalTexts	Yes	fr	oui
4	GlobalTexts	No		no
5	GlobalTexts	No	fa	خیر
6	GlobalTexts	No	fr	pas
7	Default.aspx	Label1.Text		Hello
10	Default.aspx	Label1.ForeColor		red
11	Default.aspx	Label1.Text	en-US	hello
13	Default.aspx	Label1.ForeColor	en-US	blue
14	Default.aspx	Label1.Text	fa	درود
16	Default.aspx	Label1.ForeColor	fa	red
17	Default.aspx	Label2.Text		GoodBye
18	Default.aspx	Label2.ForeColor		orange
19	Default.aspx	Label2.Text	en-US	goodbye
20	Default.aspx	Label2.ForeColor	en-US	green
21	dir 1/page 1.aspx	Label1.Text		sssss
22	dir 1/page 1.aspx	Label2.Text		aaaaa
23	dir 1/page 1.aspx	Label1.Text	en-US	String 1
24	dir 1/page 1.aspx	Label2.Text	en-US	String 2
25	dir 1/page 1.aspx	Label1.Text	fa	رشته 1
26	dir 1/page 1.aspx	Label2.Text	fa	رشته 2

اصلاح کلاس DbResourceProviderFactory

برای ذخیره منابع محلی، جهت اطمینان از یکسان بودن نام منبع، متد مربوطه در کلاس DbResourceProviderFactory باید به صورت زیر تغییر کند:

```
public override IResourceProvider CreateLocalResourceProvider(string virtualPath)
{
    if (!string.IsNullOrEmpty(virtualPath))
    {
        virtualPath = virtualPath.Remove(0, virtualPath.IndexOf('/') + 1); // removes everything from start to the first '/'
    }
    return new LocalDbResourceProvider(virtualPath);
}
```

با این تغییر مسیرهای درخواستی چون "~/Default.aspx" و یا "/Default.aspx" هر دو به صورت "Default.aspx" در می آیند تا با نام ذخیره شده در دیتابیس یکسان شوند.

ارتباط با دیتابیس

خوشبختانه برای تبادل اطلاعات با جدول بالا امروزه راههای زیادی وجود دارد. برای پیاده سازی آن مثلاً می توان از یک اینترفیس استفاده کرد. سپس با استفاده از سازوکارهای موجود مثلاً به کارگیری [IoC](#)، نمونه مناسبی از پیاده سازی اینترفیس مذکور را در اختیار برنامه قرار داد. اما برای جلوگیری از پیچیدگی بیش از حد و دور شدن از مبحث اصلی، برای پیاده سازی فعلی از EF Code First به صورت مستقیم در پروژه استفاده شده است که [سری آموزشی کاملی از آن](#) در همین سایت وجود دارد.

پس از پیاده سازی کلاس های مرتبط برای استفاده از EF Code First، از کلاس ResourceData که در بخش اول نیز نشان داده شده بود، برای کپسوله کردن ارتباط با داده ها استفاده می شود که نمونه ای ابتدایی از آن در زیر آورده شده است:

```
using System.Collections.Generic;
using System.Linq;
using DbResourceProvider.Models;

namespace DbResourceProvider.Data
{
    public class ResourceData
    {
        private readonly string _resourceName;
        public ResourceData(string resourceName)
        {
            _resourceName = resourceName;
        }
        public Resource GetResource(string resourceKey, string culture)
        {
            using (var data = new TestContext())
            {
                return data.Resources.SingleOrDefault(r => r.Name == _resourceName && r.Key == resourceKey && r.Culture == culture);
            }
        }
        public List<Resource> GetResources(string culture)
        {
            using (var data = new TestContext())
            {
                return data.Resources.Where(r => r.Name == _resourceName && r.Culture == culture).ToList();
            }
        }
    }
}
```

کلاس فوق نسبت به نمونه ای که در قسمت قبل نشان داده شد کمی فرق دارد. بدین صورت که برای راحتی بیشتر نام منبع

درخواستی به جای پارامتر متدها، در اینجا به عنوان پارامتر کانستراکتور وارد می‌شود.

نکته: در صورتی که این کلاس‌ها در پروژه‌ای جداگانه قرار دارند، باید ConnectionString مربوطه در فایل کانفیگ برنامه مقصد نیز تنظیم شود.

کش کردن ورودی‌ها

برای کش کردن ورودی‌ها این نکته را که قبلاً هم به آن اشاره شده بود باید در نظر داشت:

پس از اولین درخواست برای هر منبع، نمونه تولیدشده از پرووایدر مربوطه در حافظه سرور کش خواهد شد.

یعنی متدهای کلاس DbResourceProviderFactory به ازای هر منبع تنها یکبار فراخوانی می‌شود. نمونه‌های کش‌شده از پروایدرهای کلی و محلی به همراه تمام محتویاتشان (مثلاً نمونه تولیدی از کلاس DbResourceManager) تا زمان Unload شدن سایت در حافظه سرور باقی می‌مانند. بنابراین عملیات کشینگ ورودی‌ها را می‌توان درون خود کلاس DbResourceManager به ازای هر منبع انجام داد.

برای کش کردن ورودی‌های هر منبع می‌توان چند روش را درپیش گرفت. روش اول این است که به ازای هر کلید درخواستی تنها ورودی مربوطه از دیتابیس فراخوانی شده و در برنامه کش شود. این روش برای حالاتی که تعداد ورودی‌ها یا تعداد درخواست‌های کلیدهای هر منبع کم باشد مناسب خواهد بود.

یکی از پیاده‌سازی این روش این است که ورودی‌ها به ازای هر کالچر ذخیره شوند. پیاده‌سازی اولیه این نوع فرایند کشینگ در کلاس DbResourceManager به صورت زیر است:

```
using System.Collections.Generic;
using System.Globalization;
using DbResourceProvider.Data;
namespace DbResourceProvider
{
    public class DbResourceManager
    {
        private readonly string _resourceName;
        private readonly Dictionary<string, Dictionary<string, object>> _resourceCacheByCulture;
        public DbResourceManager(string resourceName)
        {
            _resourceName = resourceName;
            _resourceCacheByCulture = new Dictionary<string, Dictionary<string, object>>();
        }
        public object GetObject(string resourceKey, CultureInfo culture)
        {
            return GetCachedObject(resourceKey, culture.Name);
        }
        private object GetCachedObject(string resourceKey, string cultureName)
        {
            if (!_resourceCacheByCulture.ContainsKey(cultureName))
                _resourceCacheByCulture.Add(cultureName, new Dictionary<string, object>());
            var cachedResource = _resourceCacheByCulture[cultureName];
            lock (this)
            {
                if (!cachedResource.ContainsKey(resourceKey))
                {
                    var data = new ResourceData(_resourceName);
                    var dbResource = data.GetResource(resourceKey, cultureName);
                    if (dbResource == null) return null;
                    var cachedResources = _resourceCacheByCulture[cultureName];
                    cachedResources.Add(dbResource.Key, dbResource.Value);
                }
            }
            return cachedResource[resourceKey];
        }
    }
}
```

همانطور که قبلاً توضیح داده شد کش پرووایدرهای منابع به ازای هر منبع درخواستی (و به تبع آن نمونه‌های موجود در آن مثل DbResourceManager) برعهده خود ASP.NET است. بنابراین برای کش کردن ورودی‌های درخواستی هر منبع در کلاس DbResourceManager تنها کافی است آن‌ها را درون یک متغیر محلی در سطح کلاس (فیلد) ذخیره کرد. کاری که در کد بالا در متغیر _resourceCacheByCulture انجام شده است. در این متغیر که از نوع دیکشنری تعریف شده است کلیدهای هر عضو آن برابر نام کالچر مربوطه است. مقادیر هر عضو این دیکشنری نیز خود یک دیکشنری است که ورودی‌های منابع مربوط به کالچر مربوطه در

آن ذخیره می‌شوند.

عملیات در متد `GetCachedObject` انجام می‌شود. همان‌طور که می‌بینید ابتدا وجود ورودی موردنظر در متغیر کشینگ بررسی می‌شود و در صورت عدم وجود، مقدار آن مستقیماً از دیتابیس درخواست می‌شود. سپس این مقدار درخواستی ابتدا درون متغیر کشینگ ذخیره شده (به همراه بلاک `lock`) و در نهایت برگشت داده می‌شود.

نکته: کل فرایند بررسی وجود کلید در متغیر کشینگ (شرط دوم در متد `GetCachedObject`) درون بلاک `lock` قرار داده شده است تا در درخواست‌های همزمان احتمال افزودن چندباره یک کلید از بین برود.

پایاده‌سازی دیگر این فرایند کشینگ، ذخیره ورودی‌ها براساس نام کلید به جای نام کالچر است. یعنی کلید دیکشنری اصلی نام کلید و کلید دیکشنری داخلی نام کالچر است که این روش زیاد جالب نیست.

روش دوم که بیشتر برای برنامه‌های بزرگ با ورودی‌ها و درخواست‌های زیاد به کار می‌رود این است که در هر بار درخواست به دیتابیس به جای دریافت تنها همان ورودی درخواستی، تمام ورودی‌های منبع و کالچر درخواستی استخراج شده و کش می‌شود تا تعداد درخواست‌های به سمت دیتابیس کاهش یابد. برای پایاده‌سازی این روش کافی است تغییرات زیر در متد `GetCachedObject` اعمال شود:

```
private object GetCachedObject(string resourceKey, string cultureName)
{
    lock (this)
    {
        if (!_resourceCacheByCulture.ContainsKey(cultureName))
        {
            _resourceCacheByCulture.Add(cultureName, new Dictionary<string, object>());
            var cachedResources = _resourceCacheByCulture[cultureName];
            var data = new ResourceData(_resourceName);
            var dbResources = data.GetResources(cultureName);
            foreach (var dbResource in dbResources)
            {
                cachedResources.Add(dbResource.Key, dbResource.Value);
            }
        }
    }
    var cachedResource = _resourceCacheByCulture[cultureName];
    return !cachedResource.ContainsKey(resourceKey) ? null : cachedResource[resourceKey];
}
```

در اینجا هم می‌توان به جای استفاده از نام کالچر برای کلید دیکشنری اصلی از نام کلید ورودی منبع استفاده کرد که چندان توصیه نمی‌شود.

نکته: انتخاب یکی از دو روش فوق برای فرایند کشینگ کاملاً به شرایط موجود و سلیقه برنامه نویس بستگی دارد.

فرایند Fallback

درباره فرایند `fallback` به اندازه کافی در قسمت‌های قبلی توضیح داده شده است. برای پایاده‌سازی این فرایند ابتدا باید به نوعی به سلسله مراتب کالچرهای موجود از کالچر جاری تا کالچر اصلی و پیش فرض سیستم دسترسی پیدا کرد. برای اینکار ابتدا باید با استفاده از روشی کالچر والد یک کالچر را بدست آورد. کالچر والد کالچری است که عمومیت بیشتری نسبت به کالچر موردنظر دارد. مثلاً کالچر `fa`، کالچر والد `fa-IR` است. همچنین کالچر `Invariant` به عنوان والد تمام کالچرها شناخته می‌شود. خوشبختانه در کلاس `CultureInfo` (که در قسمت‌های قبلی شرح داده شده است) یک پراپرتی با عنوان `Parent` وجود دارد که کالچر والد را برمی‌گرداند.

برای رسیدن به سلسله مراتب مذکور در کلاس `ResourceManager` دات نت، از کلاسی با عنوان `ResourceFallbackManager` استفاده می‌شود. هرچند این کلاس با سطح دسترسی `internal` تعریف شده است اما نام‌گذاری نامناسبی دارد زیرا کاری که می‌کند به عنوان `Manager` هیچ ربطی ندارد. این کلاس با استفاده از یک کالچر ورودی، یک `enumerator` از سلسله مراتب کالچرها که در بالا صحبت شد تهیه می‌کند.

با استفاده پایاده‌سازی موجود در کلاس `ResourceFallbackManager` کلاسی با عنوان `CultureFallbackProvider` تهیه کردم که به صورت زیر است:


```

using System.Collections;
using System.Collections.Generic;
using System.Globalization;
namespace DbResourceProvider
{
    public class CultureFallbackProvider : IEnumerable<CultureInfo>
    {
        private readonly CultureInfo _startingCulture;
        private readonly CultureInfo _neutralCulture;
        private readonly bool _tryParentCulture;
        public CultureFallbackProvider(CultureInfo startingCulture = null,
                                      CultureInfo neutralCulture = null,
                                      bool tryParentCulture = true)
        {
            _startingCulture = startingCulture ?? CultureInfo.CurrentCulture;
            _neutralCulture = neutralCulture;
            _tryParentCulture = tryParentCulture;
        }
        #region Implementation of IEnumerable<CultureInfo>
        public IEnumerator<CultureInfo> GetEnumerator()
        {
            var reachedNeutralCulture = false;
            var currentCulture = _startingCulture;
            do
            {
                if (_neutralCulture != null && currentCulture.Name == _neutralCulture.Name)
                {
                    yield return CultureInfo.InvariantCulture;
                    reachedNeutralCulture = true;
                    break;
                }
                yield return currentCulture;
                currentCulture = currentCulture.Parent;
            } while (_tryParentCulture && !HasInvariantCultureName(currentCulture));
            if (!_tryParentCulture || HasInvariantCultureName(_startingCulture) || reachedNeutralCulture)
                yield break;
            yield return CultureInfo.InvariantCulture;
        }
        #endregion
        #region Implementation of IEnumerable
        IEnumerator IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }
        #endregion
        private bool HasInvariantCultureName(CultureInfo culture)
        {
            return culture.Name == CultureInfo.InvariantCulture.Name;
        }
    }
}

```

این کلاس که اینترفیس `IEnumerable<CultureInfo>` را پیاده‌سازی کرده است، سه پارامتر کانستراکتور دارد. اولین پارامتر، کالچر جاری یا آغازین را مشخص می‌کند. این کالچری است که تولید `enumerator` مربوطه از آن آغاز می‌شود. در صورتی که این پارامتر نال باشد مقدار کالچر UI در ثرد جاری برای آن در نظر گرفته می‌شود. مقدار پیش‌فرضی که برای این پارامتر در نظر گرفته شده است، `null` است. پارامتر بعدی کالچر خنثی موردنظر کاربر است. این کالچری است که در صورت رسیدن `enumerator` به آن کار پایان خواهد یافت. در واقع کالچر پایانی `enumerator` است. این پارامتر می‌تواند نال باشد. مقدار پیش‌فرضی که برای این پارامتر در نظر گرفته شده است، `null` است. پارامتر آخر هم تعیین می‌کند که آیا `enumerator` از کالچرهای والد استفاده بکند یا خیر. مقدار پیش‌فرضی که برای این پارامتر در نظر گرفته شده است، `true` است. کار اصلی کلاس فوق در متد `GetEnumerator` انجام می‌شود. در این کلاس یک حلقه `do-while` وجود دارد که `enumerator` را با استفاده از کلمه کلیدی `yield` تولید می‌کند. در این متد ابتدا در صورت نال نبودن کالچر خنثی ورودی، بررسی می‌شود که آیا نام کالچر جاری حلقه (که در متغیر محلی `currentCulture` ذخیره شده است) برابر نام کالچر خنثی است یا خیر. در صورت برقراری شرط، کار این حلقه با برگشت `CultureInfo.InvariantCulture` پایان می‌یابد. کالچر بدون زبان و فرهنگ و موقعیت مکانی است که در واقع به عنوان کالچر والد تمام کالچرها در نظر گرفته می‌شود. پراپرتی `Name` این کالچر برابر `string.Empty` است.

کار حلقه با برگشت مقدار کالچر جاری enumerator ادامه می‌یابد. سپس کالچر جاری با کالچر والدش مقداردهی می‌شود. شرط قسمت while حلقه تعیین می‌کند که در صورتی که کلاس برای استفاده از کالچرهای والد تنظیم شده باشد، تا زمانی که نام کالچر جاری برابر نام کالچر Invariant نباشد، تولید اعضای enumerator ادامه یابد.

در انتها نیز در صورتی که با شرایط موجود، قبلا کالچر Invariant برگشت داده نشده باشد این کالچر نیز yield می‌شود. در واقع در صورتی که استفاده از کالچرهای والد اجازه داده نشده باشد یا کالچر آغازین برابر کالچر Invariant باشد و یا قبلا به دلیل رسیدن به کالچر خنثی ورودی، مقدار کالچر Invariant برگشت داده شده باشد، enumerator قطع شده و عملیات پایان می‌یابد. در غیر این صورت کالچر Invariant به عنوان کالچر پایانی برگشت داده می‌شود.

استفاده از CultureFallbackProvider

با استفاده از کلاس CultureFallbackProvider می‌توان عملیات جستجوی ورودی‌های درخواستی را با ترتیبی مناسب بین تمام کالچرهای موجود به انجام رسانید.

برای استفاده از این کلاس باید تغییراتی در متد GetObject کلاس DbResourceManager به صورت زیر اعمال کرد:

```
public object GetObject(string resourceKey, CultureInfo culture)
{
    foreach (var currentCulture in new CultureFallbackProvider(culture))
    {
        var value = GetCachedObject(resourceKey, currentCulture.Name);
        if (value != null) return value;
    }
    throw new KeyNotFoundException("The specified 'resourceKey' not found.");
}
```

با استفاده از یک حلقه foreach درون enumerator کلاس CultureFallbackProvider، کالچرهای مورد نیاز برای fallback یافته می‌شوند. در اینجا از مقادیر پیش فرض دو پارامتر دیگر کانستراکتور کلاس CultureFallbackProvider استفاده شده است. سپس به ازای هر کالچر یافته شده مقدار ورودی درخواستی بدست آمده و در صورتی که نال نباشد (یعنی ورودی مورد نظر برای کالچر جاری یافته شود) آن مقدار برگشت داده می‌شود و در صورتی که نال باشد عملیات برای کالچر بعدی ادامه می‌یابد. در صورتی که ورودی درخواستی یافته نشود (خروج از حلقه بدون برگشت مقداری برای ورودی منبع درخواستی) استثنای KeyNotFoundException صادر می‌شود تا کاربر را از اشتباه رخ داده مطلع سازد.

آزمایش پرووایدر سفارشی

ابتدا تنظیمات مورد نیاز فایل کانفیگ را که در [قسمت قبل](#) نشان داده شد، در برنامه خود اعمال کنید.

داده‌های نمونه نشان داده شده در ابتدای این مطلب را در نظر بگیرید. حال اگر در یک برنامه وب اپلیکیشن، صفحه Default.aspx در ریشه سایت حاوی دو کنترل زیر باشد:

```
<asp:Label ID="Label1" runat="server" meta:resourcekey="Label1" />
<asp:Label ID="Label2" runat="server" meta:resourcekey="Label2" />
```

خروجی برای کالچر "en-US" (معمولا پیش فرض، اگر تنظیمات سیستم عامل تغییر نکرده باشد) چیزی شبیه تصویر زیر خواهد بود:

hello goodbye

سپس تغییر زیر را در فایل web.config اعمال کنید تا کالچر UI سایت به fa تغییر یابد (به بخش uiCulture="fa" دقت کنید):

```
<globalization uiCulture="fa" resourceProviderFactoryType =
```

```
"DbResourceProvider.DbResourceProviderFactory, DbResourceProvider" />
```

بنابراین صفحه Default.aspx با همان داده‌های نشان داده شده در بالا به صورت زیر تغییر خواهد کرد:

GoodBye درود

می‌بینید که با توجه به عدم وجود مقداری برای Label12.Text برای کالچر fa، عملیات fallback اتفاق افتاده است.

بحث و نتیجه‌گیری

کار تولید یک پرووایدر منابع سفارشی دیتابیزی به اتمام رسید. تا اینجا اصول کلی تولید یک پرووایدر سفارشی شرح داده شد. بدین ترتیب می‌توان برای هر حالت خاص دیگری نیز پرووایدرهای سفارشی مخصوص ساخت تا مدیریت منابع به آسانی تحت کنترل برنامه نویسی قرار گیرد.

اما نکته‌ای را که باید به آن توجه کنید این است که در پیاده‌سازی‌های نشان داده شده با توجه به نحوه کش‌شدن مقادیر ورودی‌ها، اگر این مقادیر در دیتابیس تغییر کنند، تا زمانیکه سایت ریست نشود این تغییرات در برنامه اعمال نخواهد شد. زیرا همانطور که اشاره شد، مدیریت نمونه‌های تولیدشده از پرووایدرهای منابع برای هر منبع درخواستی در نهایت برعهده ASP.NET است. بنابراین باید مکانیزمی پیاده شود تا کلاس DbResourceManager از به‌روزرسانی ورودی‌های کش‌شده اطلاع یابد تا آنها را ریفرش کند.

در ادامه درباره روش‌های مختلف نحوه پیاده‌سازی قابلیت به‌روزرسانی ورودی‌های منابع در زمان اجرا با استفاده از پرووایدرهای منابع سفارشی بحث خواهد شد. همچنین راه‌حل‌های مختلف استفاده از این پرووایدرهای سفارشی در جاهای مختلف پروژه‌های MVC شرح داده می‌شود.

البته مباحث پیشرفته‌تری چون تزریق وابستگی برای پیاده‌سازی لایه ارتباط با دیتابیس در بیرون و یا تولید یک Factory برای تزریق کامل پرووایدر منابع از بیرون نیز جای بحث و بررسی دارد.

منابع

<http://weblogs.asp.net/thangchung/archive/2010/06/25/extending-resource-provider-for-soring-resources-in-the-database.aspx>

<http://msdn.microsoft.com/en-us/library/aa905797.aspx>

<http://msdn.microsoft.com/en-us/library/system.web.compilation.resourceproviderfactory.aspx>

<http://www.dotnetframework.org/default.aspx/.../ResourceFallbackManager@cs>

<http://www.codeproject.com/Articles/14190/ASP-NET-2-0-Custom-SQL-Server-ResourceProvider>

<http://www.west-wind.com/presentations/wwdbresourceprovider>

نظرات خوانندگان

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۲/۰۳/۰۸ ۰:۴۲

با تشکر از کار زیبای شما
لطفاً برچسب [resource](#) را اضافه کنید تا پیوستگی مطالب حفظ شود.

نویسنده: یوسف نژاد
تاریخ: ۱۳۹۲/۰۳/۰۸ ۱:۴۰

با تشکر از دقت نظر شما.
برچسب Resource هم اضافه شد.

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۲/۰۳/۰۸ ۳:۱۵

مهندس بک سوال؟
مشکلی نداره ما سه جدول:
1- جدولی برای ذخیره نام کالچرها
2- جدولی برای ذخیره عنوان کلیدهای اصلی
3- جدولی برای ذخیره مقادیر یک کالچر برای یک کلید خاص

تعریف کنیم؟
اگر درست فهمیده باشم فقط باید بخش بازیابی کلیدها تغییر کنه درسته؟

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۳/۰۸ ۸:۴۱

اون وقت حداقل 2 تا join باید بنویسید و وجود هر join یعنی کم‌تر شدن سرعت دسترسی به اطلاعات. چرا؟ چه تکرار اطلاعاتی رو مشاهده می‌کنید که قصد دارید تا این حد نرمالش کنید؟ نام و کلید و فرهنگ یک موجودیت هستند.

نویسنده: یوسف نژاد
تاریخ: ۱۳۹۲/۰۳/۰۸ ۹:۱۱

دلیل خاصی برای تفکیک این چینی وجود نداره و همونطور که دوستمون گفتن این روشی که شما اشاره کردین مشکلات و معایبی هم به همراه داره.
روش اشاره شده تو این مطلب تو بیش از 99 درصد پروژه‌ها کفایت میکنه. فقط تو پروژه‌های بسیار بسیار بزرگ با ورودی‌های منابع بسیار زیاد (چند صد هزار و یا بیشتر) تغییر این ساختار برای رسیدن به کارایی مناسب میتونه مفید باشه.
در هر صورت اگر نیاز به تغییر ساختار جدول دارین فقط لایه دسترسی به بانک باید تغییر بکنه و فرایند کلی دسترسی به ورودی‌های منابع ذخیره شده در دیتابیس باید به همون صورتی باشه که در اینجا آورده شده. یعنی در نهایت با استفاده از سه پارامتر نام منبع، نام کالچر و عنوان کلید درخواستی کار استخراج مقدار ورودی باید انجام بشه.

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۲/۰۳/۰۸ ۱۰:۱۴

برای طراحی یک سامانه مدیریت محتوا با کلی مازول فکر می‌کنم حرفم منطقی باشه مهندس، در ضمن همونجوری که مهندس [یوسف نژاد](#) فرمودن اطلاعات در بازیابی اولیه کش میشه و تا ری ستارت شدن سایت در حافظه می‌مونه، فکر می‌کنم چندان تاثیری

بروی کارایی داشته باشه با توجه به فرضیات، فرض کن من 10000 عنوان دارم، 30 تا زبان دارم در این صورت توی یک جدول زبان انگلیسی (en-کالچر انگلیسی) 10000 بار تکرار میشه علاوه بر اون عنوان مثلا "نام کاربری" به ازای 30 زبان 30 بار تکرار میشه زیادم حرف من غیر منطقی نیست و الا حرف شما درسته بله join سرعت پایین میاره اما ما که قرار نیست زیادی دسترسی به این جداول داشته باشیم.

"پس از اولین درخواست برای هر منبع، نمونه تولیدشده از پرووایدر مربوطه در حافظه سرور کش خواهد شد." سخن مهندس

[یوسف نژاد](#)

نویسنده: محسن خان

تاریخ: ۱۳۹۲/۰۳/۰۸ ۱۲:۱۰

یک سری از برآوردها خیلی هستند. حتی میکروسافت هم با لشگر مترجم‌هایی که داره مثلا برای شیرپوینت تجاری خودش زیر 10 تا زبان رو تونسته ارائه بده.

نویسنده: بهنام حقی

تاریخ: ۱۳۹۳/۰۱/۳۱ ۱۷:۰۹

با سلام

من این حالت رو میخوام با uow میخوام پیاده سازی کنم. میخوام یک سری تغییرات تو ساختار جدول بدم. یک جدول برای مدیریت اضافه و حذف زبان (نام، RTL، ISO، Culture و ...) و جدول دیگم برای ریسورس‌ها (کلید، اسم، مقدار) در واقع میخوام مقادیر ریسورس‌ها با اضافه و حذف شدن یک زبان به سیستم مدیریت بشه. میخوام ببینم که چه پیشنهادی برای این حالت دارید؟

در [قسمت قبل](#) مطالب تکمیلی تولید پرووایدر سفارشی منابع دیتابیزی ارائه شد. در این قسمت نحوه برورسانی ورودی‌های منابع در زمان اجرا بحث می‌شود.

تولید یک پرووایدر منابع دیتابیزی - بخش سوم

برای پیاده‌سازی ویژگی به‌روزرسانی ورودی‌های منابع در زمان اجرا راه‌حل‌های مختلفی ممکن است به ذهن برنامه‌نویس خطور کند که هر کدام معایب و مزایای خودش را دارد. اما در نهایت بسته به شرایط موجود انتخاب روش مناسب برعهده خود برنامه‌نویس است.

مثلاً برای پرووایدر سفارشی دیتابیزی تهیه‌شده در مطالب قبلی، تنها کافی است ابزاری تهیه شود تا به کاربران اجازه به‌روزرسانی مقادیر موردنظرشان در دیتابیس را بدهد که کاری بسیار ساده است. بدین ترتیب به‌روزرسانی این مقادیر در زمان اجرا کاری بسیار ابتدایی به نظر می‌رسد. اما در [قسمت قبل](#) نشان داده شد که برای بالا بردن بازدهی بهتر است که مقادیر موجود در دیتابیس در حافظه سرور کش شوند. استراتژی اولیه و ساده‌ای نیز برای نحوه پیاده‌سازی این فرایند کشینگ ارائه شد. بنابراین باید امکاناتی فراهم شود تا در صورت تغییر مقادیر کش‌شده در سمت دیتابیس، برنامه از این تغییرات آگاه شده و نسبت به به‌روزرسانی این مقادیر در متغیر کشینگ اقدامات لازم را انجام دهد.

اما همان‌طور که در [قسمت قبل](#) نیز اشاره شد، نکته‌ای که باید در نظر داشت این است که مدیریت تمامی نمونه‌های تولیدشده از کلاس‌های موردبحث کاملاً برعهده ASP.NET است، بنابراین دسترسی مستقیمی به این نمونه‌ها در بیرون و در زمان اجرا وجود ندارد تا این ویژگی را بتوان در مورد آن‌ها پیاده کرد.

یکی از روش‌های موجود برای حل این مشکل این است که مکانیزمی پیاده شود تا بتوان به تمامی نمونه‌های تولیدی از کلاس DbResourceManager در بیرون از محیط سیستم مدیریت منابع ASP.NET دسترسی داشت. مثلاً یک کلاس حاوی متغیری استاتیک جهت ذخیره نمونه‌های تولیدی از کلاس DbResourceManager، به کتابخانه خود اضافه کرد تا با استفاده از یکسری امکانات بتوان این نمونه‌های تولیدی را از تغییرات رخداده در سمت دیتابیس آگاه کرد. در این قسمت پیاده‌سازی این راه‌حل شرح داده می‌شود.

نکته: قبل از هرچیز برای مناسب شدن طراحی کتابخانه تولیدی و افزایش امنیت آن بهتر است تا سطح دسترسی تمامی کلاس‌های پیاده‌سازی شده تا این مرحله به internal تغییر کند. از آنجاکه سیستم مدیریت منابع ASP.NET از ریفלקشن برای تولید نمونه‌های موردنیاز خود استفاده می‌کند، بنابراین این تغییر تأثیری بر روند کاری آن نخواهد گذاشت.

نکته: با توجه به شرایط خاص موجود، ممکن است نام‌های استفاده شده برای کلاس‌های این کتابخانه کمی گیج‌کننده باشد. پس با دقت بیشتری به مطلب توجه کنید.

پیاده‌سازی امکان پاک‌سازی مقادیر کش‌شده

برای این کار باید تغییراتی در کلاس `DbResourceManager` داده شود تا بتوان این کلاس را از تغییرات بوجود آمده آگاه ساخت. روشی که من برای این کار در نظر گرفتم استفاده از یک اینترفیس حاوی اعضای موردنیاز برای پیاده‌سازی این امکان است تا مدیریت این ویژگی در ادامه راحت‌تر شود.

اینترفیس `IDbCachedResourceManager`

این اینترفیس به صورت زیر تعریف شده است:

```
namespace DbResourceProvider
{
    internal interface IDbCachedResourceManager
    {
        string ResourceName { get; }

        void ClearAll();
        void Clear(string culture);
        void Clear(string culture, string resourceKey);
    }
}
```

در پراپرتی فقط خواندنی `ResourceName` نام منبع کش شده ذخیره خواهد شد.

متد `ClearAll` برای پاک‌سازی تمامی ورودی‌های کش‌شده استفاده می‌شود.

متدهای `Clear` برای پاک‌سازی ورودی‌های کش‌شده یک کالچر به خصوص و یا یک ورودی خاص استفاده می‌شود.

با استفاده از این اینترفیس، پیاده‌سازی کلاس `DbResourceManager` به صورت زیر تغییر می‌کند:

```
using System.Collections.Generic;
using System.Globalization;
using DbResourceProvider.Data;
namespace DbResourceProvider
{
    internal class DbResourceManager : IDbCachedResourceManager
    {
        private readonly string _resourceName;
        private readonly Dictionary<string, Dictionary<string, object>> _resourceCacheByCulture;
        public DbResourceManager(string resourceName)
        {
            _resourceName = resourceName;
            _resourceCacheByCulture = new Dictionary<string, Dictionary<string, object>>();
        }
        public object GetObject(string resourceKey, CultureInfo culture) { ... }
        private object GetCachedObject(string resourceKey, string cultureName) { ... }

        #region Implementation of IDbCachedResourceManager
        public string ResourceName
        {
            get { return _resourceName; }
        }
        public void ClearAll()
        {
            lock (this)
            {
                _resourceCacheByCulture.Clear();
            }
        }
        public void Clear(string culture)
```

```

    {
        lock (this)
        {
            if (!_resourceCacheByCulture.ContainsKey(culture)) return;
            _resourceCacheByCulture[culture].Clear();
        }
    }
    public void Clear(string culture, string resourceKey)
    {
        lock (this)
        {
            if (!_resourceCacheByCulture.ContainsKey(culture)) return;
            _resourceCacheByCulture[culture].Remove(resourceKey);
        }
    }
}
#endregion
}
}

```

اعضای اینترفیس IDbCachedResourceManager به صورت مناسبی در کد بالا پیاده‌سازی شدند. در تمام این پیاده‌سازی‌ها مقادیر مربوطه از درون متغیر کشینگ پاک می‌شوند تا پس از اولین درخواست، بلافاصله از دیتابیس خوانده شوند. برای جلوگیری از دسترسی هم‌زمان نیز از بلاک lock استفاده شده است.

برای استفاده از این امکانات جدید همان‌طور که در بالا نیز اشاره شد باید بتوان نمونه‌های تولیدی از کلاس DbResourceManager توسط ASP.NET درون متغیری استاتیک ذخیره شوند. برای اینکار از کلاس جدیدی با عنوان DbResourceCacheManager استفاده می‌شود که برخلاف تمام کلاس‌های تعریف‌شده تا اینجا با سطح دسترسی public تعریف می‌شود.

کلاس DbResourceCacheManager

مدیریت نمونه‌های تولیدی از کلاس DbResourceManager در این کلاس انجام می‌شود. این کلاس پیاده‌سازی ساده‌ای به‌صورت زیر دارد:

```

using System.Collections.Generic;
using System.Linq;
namespace DbResourceProvider
{
    public static class DbResourceCacheManager
    {
        internal static List<IDbCachedResourceManager> ResourceManagers { get; private set; }
        static DbResourceCacheManager()
        {
            ResourceManagers = new List<IDbCachedResourceManager>();
        }
        public static void ClearAll()
        {
            ResourceManagers.ForEach(r => r.ClearAll());
        }
        public static void Clear(string resourceName)
        {
            GetResouceManagers(resourceName).ForEach(r => r.ClearAll());
        }
        public static void Clear(string resourceName, string culture)
        {
            GetResouceManagers(resourceName).ForEach(r => r.Clear(culture));
        }
        public static void Clear(string resourceName, string culture, string resourceKey)
        {
            GetResouceManagers(resourceName).ForEach(r => r.Clear(culture, resourceKey));
        }

        private static List<IDbCachedResourceManager> GetResouceManagers(string resourceName)
        {
            return ResourceManagers.Where(r => r.ResourceName.ToLower() == resourceName.ToLower()).ToList();
        }
    }
}

```



```
}
}
```

از آنجاکه نیازی به تولید نمونه ای از این کلاس وجود ندارد، این کلاس به صورت استاتیک تعریف شده است. بنابراین تمام اعضای درون آن نیز استاتیک هستند.

از پراپرتی ResourceManagers برای نگهداری لیستی از نمونه‌های تولیدی از کلاس DbResourceManager استفاده می‌شود. این پراپرتی از نوع <IDbCachedResourceManager>List تعریف شده است و برای جلوگیری از دسترسی بیرونی، سطح دسترسی آن internal در نظر گرفته شده است.

در کانستراکتور استاتیک این کلاس (اطلاعات بیشتر درباره static constructor در [اینجا](#)) این پراپرتی با مقداری به یک نمونه تازه از لیست، اصطلاحاً initialize می‌شود.

سایر متدها نیز برای فراخوانی متدهای موجود در اینترفیس IDbCachedResourceManager پیاده‌سازی شده‌اند. تمامی این متدها دارای سطح دسترسی public هستند. همان‌طور که می‌بینید از خاصیت ResourceName برای مشخص کردن نمونه موردنظر استفاده شده است که دلیل آن در [قسمت قبل](#) شرح داده شده است.

دقت کنید که برای اطمینان از انتخاب درست همه موارد موجود در شرط انتخاب نمونه موردنظر در متد GetResouceManagers از متد ToLower برای هر دو سمت شرط استفاده شده است.

نکته مهم: درباره علت برگشت یک لیست از متد انتخاب نمونه موردنظر از کلاس DbResourceManager در کد بالا (یعنی متد GetResouceManagers) باید نکته‌ای اشاره شود. در قسمت قبل عنوان شد که سیستم مدیریت منابع ASP.NET نمونه‌های تولیدی از پرووایدرهای منابع را به ازای هر منبع کش می‌کند. اما یک نکته بسیار مهم که باید به آن توجه کرد این است که این کش برای «عبارات بومی‌سازی ضمنی» و نیز «متد مربوط به منابع محلی» موجود در کلاس HttpContext و یا نمونه مشابه آن در کلاس TemplateControl (همان متد GetLocalResourceObject که درباره این متدها در [قسمت سوم](#) این سری شرح داده شده است) از یکدیگر جدا هستند و استفاده از هریک از این دو روش موجب تولید یک نمونه مجزا از پرووایدر مربوطه می‌شود که متأسفانه کنترل آن از دست برنامه نویس خارج است. دقت کنید که این اتفاق برای منابع کلی رخ نمی‌دهد.

بنابراین برای پاک کردن مناسب ورودی‌های کش‌شده در کلاس فوق به جای استفاده از متد Single در انتخاب نمونه موردنظر از کلاس DbResourceManager (در متد GetResouceManagers) از متد Where استفاده شده و یک لیست برگشت داده می‌شود. چون با توجه به توضیح بالا امکان وجود دو نمونه DbResourceManager از یک منبع درخواستی محلی در لیست نمونه‌های نگهداری شده در این کلاس وجود دارد.

افزودن نمونه‌ها به کلاس DbResourceCacheManager

برای نگهداری نمونه‌های تولید شده از DbResourceManager، باید در یک قسمت مناسب این نمونه‌ها را به لیست مربوطه در کلاس DbResourceCacheManager اضافه کرد. بهترین مکان برای انجام این عمل در کلاس پایه BaseDbResourceProvider است که درخواست تولید نمونه را در متد EnsureResourceManager در صورت نال بودن آن می‌دهد. بنابراین این متد را به صورت زیر تغییر می‌دهیم:

```
private void EnsureResourceManager()
{
    if (_resourceManager != null) return;
    {
        _resourceManager = CreateResourceManager();
        DbResourceCacheManager.ResourceManagers.Add(_resourceManager);
    }
}
```

تا اینجا کار پیاده‌سازی امکان مدیریت مقادیر کش‌شده در کتابخانه تولیدی به پایان رسیده است.

استفاده از کلاس DbResourceCacheManager

پس از پیاده‌سازی تمامی موارد لازم، حالتی را در نظر بگیرید که مقادیر ورودی‌های تعریف شده در منبع "dir1/page1.aspx" تغییر کرده است. بنابراین برای بروزرسانی مقادیر کش‌شده کافی است تا از کدی مثل کد زیر استفاده شود:

```
DbResourceCacheManager.Clear("dir1/page1.aspx");
```

کد بالا کل ورودی‌های کش‌شده برای منبع "dir1/page1.aspx" را پاک می‌کند. برای پاک کردن کالچر یا یک ورودی خاص نیز می‌توان از کدهایی مشابه زیر استفاده کرد:

```
DbResourceCacheManager.Clear("Default.aspx", "en-US");
DbResourceCacheManager.Clear("GlobalTexts", "en-US", "Yes");
```

دریافت کد پروژه

کد کامل پروژه DbResourceProvider به همراه مثال و اسکریپت‌های دیتابسی مربوطه از لینک زیر قابل دریافت است:

[DbResourceProvider.rar](#)

برای استفاده از این مثال ابتدا باید کتابخانه Entity Framework (با نام EntityFramework.dll) را مثلاً از طریق نوگت دریافت کنید. نسخه‌ای که من در این مثال استفاده کردم نسخه 4.4 با حجم حدود 1 مگابایت است.

نکته: در این کد یک بهبود جزئی اما مهم در کلاس ResourceData اعمال شده است. در [قسمت سوم](#) این سری، اشاره شد که نام ورودی‌های منابع Case Sensitive نیست. بنابراین برای پیاده‌سازی این ویژگی، متدهای این کلاس باید به صورت زیر تغییر کنند:

```
public Resource GetResource(string resourceKey, string culture)
{
    using (var data = new TestContext())
    {
        return data.Resources.SingleOrDefault(r => r.Name.ToLower() == _resourceName.ToLower() &&
            r.Key.ToLower() == resourceKey.ToLower() && r.Culture == culture);
    }
}
```

```
public List<Resource> GetResources(string culture)
{
    using (var data = new TestContext())
    {
        return data.Resources.Where(r => r.Name.ToLower() == _resourceName.ToLower() && r.Culture ==
culture).ToList();
    }
}
```

تغییرات اعمال شده همان استفاده از متد ToLower در دو طرف شرط مربوط به نام منابع و کلید ورودی‌هاست.

در آینده...

در ادامه مطالب، بحث تهیه پرووایدر سفارشی فایل‌های resx. برای پیاده‌سازی امکان به‌روزرسانی در زمان اجرا ارائه خواهد شد. بعد از پایان تهیه این پرووایدر سفارشی، این سری مطالب با ارائه نکات استفاده از این پرووایدرها در ASP.NET MVC پایان خواهد یافت.

منابع

<http://msdn.microsoft.com/en-us/library/aa905797.aspx>

<http://www.west-wind.com/presentations/wfdbresourceprovider>

نظرات خوانندگان

نویسنده: محسن خان
تاریخ: ۱۴:۲۳ ۱۳۹۲/۰۳/۱۲

با تشکر از زحمات شما.

یک بهبود جزئی: مطابق [Managed Threading Best Practices](#) بهتره از lock this استفاده نشه و از یک شیء object خصوصی استفاده شود.

.Use caution when locking on instances, for example lock(this) in C# or SyncLock(Me) in Visual Basic
.If other code in your application, external to the type, takes a lock on the object, deadlocks could occur

نویسنده: یوسف نژاد
تاریخ: ۱۴:۳۶ ۱۳۹۲/۰۳/۱۲

مطلب شما کاملا صحیح است.
ممنون بابت یادآوری.

نویسنده: علیرضا همتی
تاریخ: ۲۳:۵۷ ۱۳۹۲/۰۳/۲۹

سلام و تشکر از زحمات شما.
من نتوانستم از این پروایدر در displayAttribute و بقیه اتریبیوتها استفاده کنم. لطفا من و راهنمایی کنید.

نویسنده: یوسف نژاد
تاریخ: ۱۰:۳۰ ۱۳۹۲/۰۳/۳۰

متأسفانه امکان استفاده مستقیم از این پرووایدرهای سفارشی در این attribute ها در MVC میسر نیست. این attribute ها به جای استفاده از پرووایدر منابع برای استخراج مقادیر ورودی ها طوری طراحی شده اند که با استفاده از Reflection از داده های ارائه شده مقادیر را از کلاس و پراپرتی مربوطه استخراج کنند. بنابراین در این attribute ها نمیتوان جایی برای استفاده از پرووایدرهای منابع یافت.

برای حل این مشکل چندین راه حل وجود دارد:

مثلا attribute های موردنیاز توسط خود برنامه نویسی پیاده سازی شوند.

یا اینکه یک کلاس مخصوص ایجاد کرد و استخراج مقادیر ورودی های منابع را در آن پیاده سازی کرد و در attribute های موردنیاز از نام این کلاس و پراپرتی های درون آن استفاده کرد.

یا اگر از فایل های resx استفاده می شود یک ابزار سفارشی برای تولید کلاس مرتبط با منبع اصلی مثل ابزار توکار ویژوال استودیو (PublicResXFileCodeGenerator) تولید کرد تا کلاس های تولیدی به جای استفاده از ResourceManager از پرووایدر منابع استفاده کند (با استفاده از متدهای موجود در HttpContext).
البته این روش ها برای حل مشکلات مربوطه در MVC در ادامه این سری شرح داده می شوند.

نویسنده: علیرضا همتی
تاریخ: ۱۳:۱۱ ۱۳۹۲/۰۳/۳۰

ممنون از شما.

نویسنده:

محسن موسوی

تاریخ:

۱۷:۳۹ ۱۳۹۲/۰۶/۰۳

با تشکر از زحمات شما

[اینجا](#) بیان شده زمانیکه از اسمبلی دیگری برای resource ها استفاده میکنید فقط میتوان **global resources** را پوشش داد. بنابراین برای استفاده از کلاس LocalDbResourceProvider بایستی تغییراتی صورت بگیرد.

چونکه همیشه این متد

```
using System.Web.Compilation;

namespace DbResourceProvider
{
    internal class DbResourceProviderFactory : ResourceProviderFactory
    {
        #region Overrides of ResourceProviderFactory

        public override IResourceProvider CreateGlobalResourceProvider(string classKey)
        {
            return new GlobalDbResourceProvider(classKey);
        }

        ...
    }
}
```

اجرا میشود.

نویسنده:

صابر فتح الهی

تاریخ:

۹:۵۸ ۱۳۹۲/۰۷/۰۸

مهندس عزیز با تشکر از کار گرانقدر شما

یک سوال؟

چگونه می‌توان الگوی کار را در این پروایدر گنجانده؟

آیا اصلاً چنین امکانی دارد یا خیر؟