

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 پیش فرض 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 پیاده سازی چنین سیستمی نیاز است تا مدلی سفارشی تهیه و استفاده شود.

- **پیاده سازی 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، سفارشی سازی 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 GetGlobalResourceObject و GetLocalResourceObject 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 در واقع 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، از آنجاکه 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 واردشده 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)

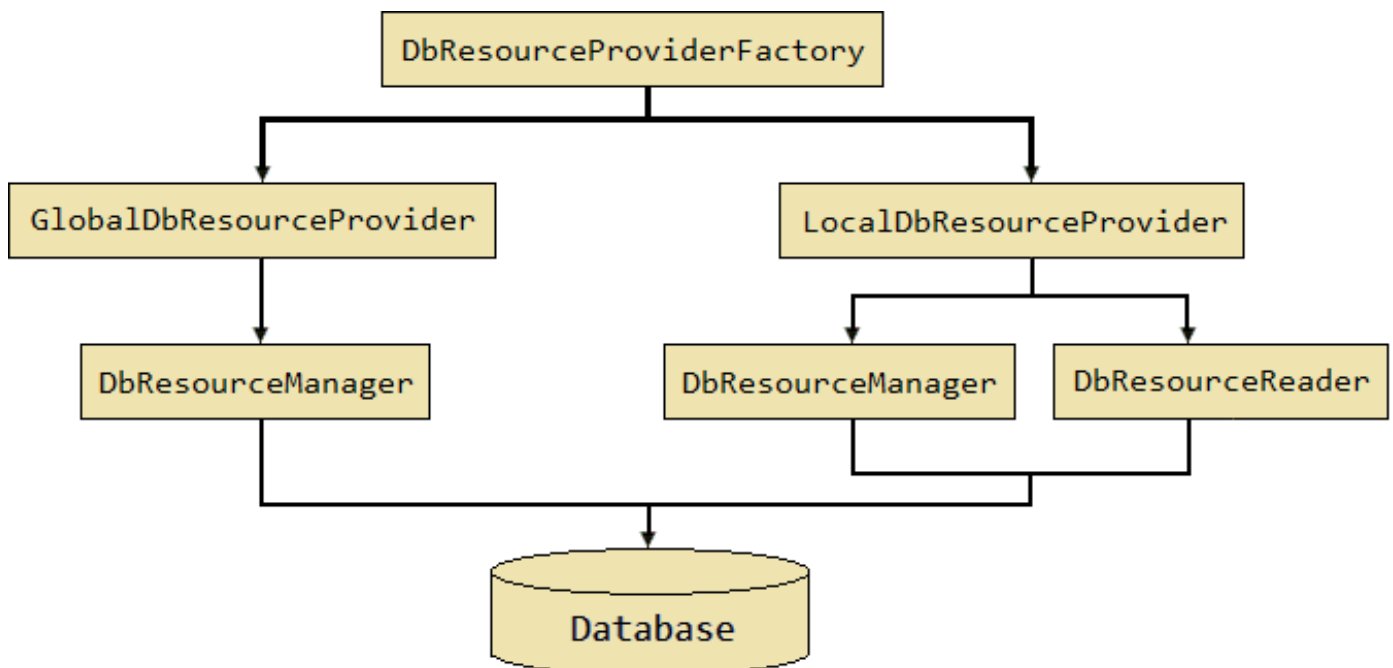
<http://msdn.microsoft.com/en-us/library/system.web.compilation.resourceproviderfactory.aspx>

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 ریفرنسی 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 همان‌نام 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 به کار 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 می‌خواهم از روند شما استفاده کنم. بی صبرانه منتظر قسمت بعدی هستم
تشکر


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

تولید یک پرووایدر منابع دیتابیس - 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 دیتابیس 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 دیتابیس 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 راه های زیادی وجود دارد. برای پیاده سازی آن مثلا می توان از یک اینترفیس استفاده کرد. سپس با استفاده از سازوکارهای موجود مثلا 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` به‌ازای 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 دیتابسی 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؟