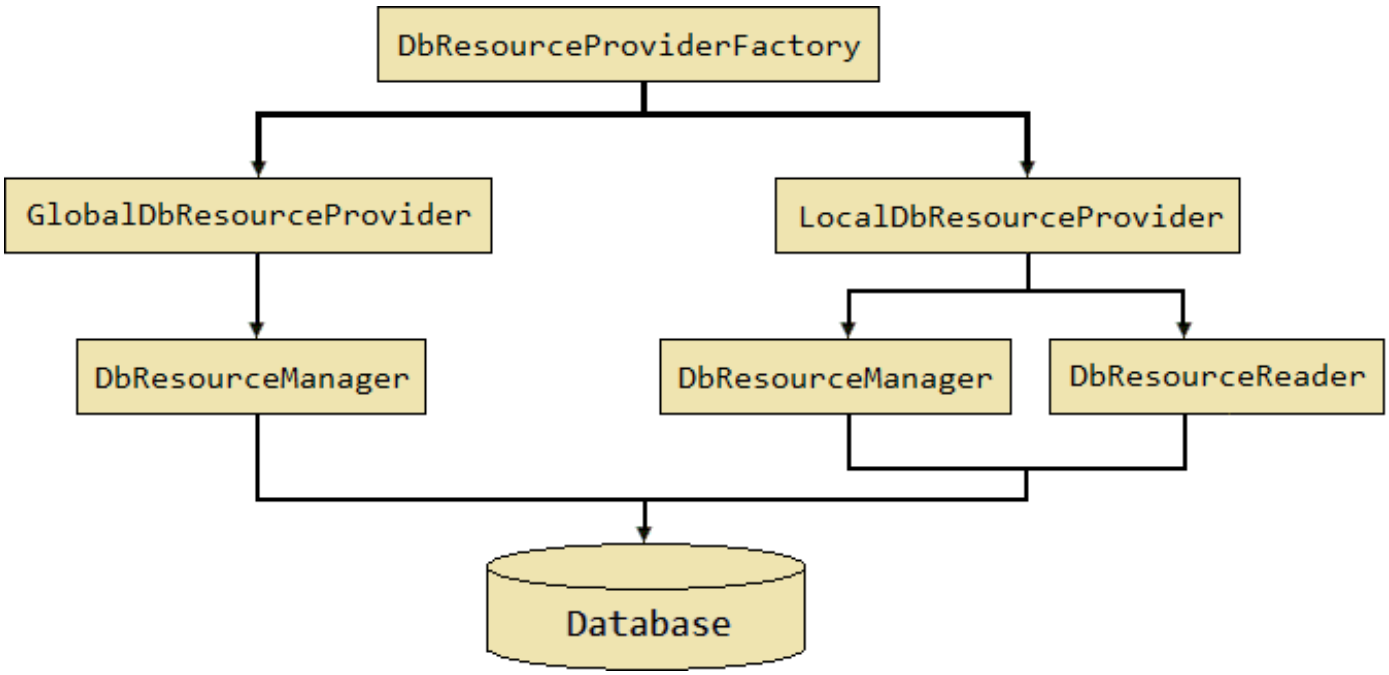
عنوان: **Globalization در ASP.NET MVC - قسمت پنجم**
نویسنده: یوسف نژاد
تاریخ: ۱۳۹۲/۰۳/۰۴ ۱۳:۵۰
آدرس: www.dotnettips.info
برچسب‌ها: Localization, Globalization, Internationalization, Resource, ResourceProviderFactory, Custom Resource Provider, Database Resource Provider

در قسمت قبل راجع به مدل پیش‌فرض پرووایدر منابع در ASP.NET بحث نسبتا مفصلی شد. در این قسمت تولید یک پرووایدر سفارشی برای استفاده از دیتابیس به جای فایل‌های resx. به عنوان منبع نگهداری داده‌ها بحث می‌شود.
قبلا 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 به‌دلیل 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 تاثیر 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 کش‌کردن 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، پیاده سازی 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 مثلا 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 دقیقا 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 صرفا 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 resouceManager_ انجام می‌شود تا درصورت نال بودن آن، بلافاصله نمونه‌ای تولید شود.

نکته: ازآنجاکه 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 درنهایت عملا برعهده ASP.NET است. در ASP.NET درطول 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. درنتیجه 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 بعدا 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 درصورتی‌که 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 درصورتی‌که نال باشد مقدار کالچر UI ثرد جاری که در CultureInfo.CurrentUICulture قرار دارد برای آن درنظر گرفته می‌شود. درنهایت 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. فعلا برای این کلاس پایه مقدار نال برگشت داده می‌شود.

کلاس GlobalDbResourceProvider

برای پرووایدر منابع کلی از این کلاس استفاده می‌شود. نحوه پیاده‌سازی آن نیز دقیقا 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 بعدا بحث می‌شود.

همان‌طور که می‌بینید برای پراپرتی ResourceReader خطای عدم پشتیبانی صادر می‌شود. دلیل آن در قسمت قبل و نیز به‌صورت 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 همنام با مسیر مجازی متناظر آن‌ها در دیتابیس ذخیره می‌شود. درباره ساختار جدول دیتابیس بعدا 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. ازآنجاکه 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 قبلا 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 موردبحث 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 بعدا 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 (قبلا هم شرح داده شد که منبع اصلی و پیش‌فرض یعنی همانی که برای زبان پیش‌فرض برنامه درنظر گرفته می‌شود و بدون نام کالچر مربوطه است، باید شامل حداکثر ورودی‌های تعریف شده باشد. منابع مربوطه به سایر کالچرها می‌توانند همه این ورودی‌های تعریف‌شده در منبع اصلی و یا قسمتی از آن را شامل شوند. عملیات fallback تضمین می‌دهد که درنهایت 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 پیاده سازی 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 اینترفیس 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 کامل اسمبلی مربوطه ,نام پرووایدر فکتوری به همراه
فضای نام آن " />
    ...
</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. مثلا 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 مورداستفاده بحث خواهد شد.

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/wwdbresourceprovider
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: ۱۳۹۲/۰۳/۰۶ ۱۱:۲۲

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

عنوان: Globalization در ASP.NET MVC - قسمت ششم
نویسنده: یوسف نژاد
تاریخ: ۱۳۹۲/۰۳/۰۷ ۲۳:۴۰
آدرس: www.dotnettips.info
گروه‌ها: Localization, Globalization, Internationalization, ResourceProviderFactory, Custom Resource Provider, Database Resource Provider, Resource

در قسمت قبل ساختار اصلی و پیاده‌سازی ابتدایی یک پرووایدر سفارشی دیتابیسی شرح داده شد. در این قسمت ادامه بحث و مطالب پیشرفته‌تر آورده شده است.

تولید یک پرووایدر منابع دیتابیسی - 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 دیتابیسی روش‌های مختلفی برای آرایش جداول جهت ذخیره انواع ورودی‌ها می‌توان درنظر گرفت.
مثلا درصورتی‌که حجم و تعداد منابع بسیار باشد و نیز منابع دیتابیسی به اندازه کافی در دسترس باشد، می‌توان به ازای هر منبع یک جدول درنظر گرفت.
یا درصورتیکه منابع داده‌ای محدودتر باشند می‌توان به ازای هر کالچر یک جدول درنظر گرفت و تمام منابع مربوط به یک کالچر را درون یک جدول ذخیره کرد. درهرصورت نحوه انتخاب آرایش جداول منابع کاملا بستگی به شرایط کاری و سلایق 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	☐
	Name	nvarchar(200)	☐
	[Key]	nvarchar(200)	☐
	Culture	nvarchar(6)	☐
	Value	nvarchar(MAX)	☐

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

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

ستون Key برای نگهداری کلید ورودی منبع استفاده می‌شود که دقیقا 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 تعدای 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	dir1/page1.aspx	Label1.Text		sssss
22	dir1/page1.aspx	Label2.Text		aaaaa
23	dir1/page1.aspx	Label1.Text	en-US	String 1
24	dir1/page1.aspx	Label2.Text	en-US	String 2
25	dir1/page1.aspx	Label1.Text	fa	رشته 1
26	dir1/page1.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 به‌صورت 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 می‌آیند 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 پیاده‌سازی آن مثلا می‌توان از یک اینترفیس استفاده کرد. سپس با استفاده از سازوکارهای موجود مثلا به‌کارگیری IoC ، نمونه مناسبی از پیاده‌سازی اینترفیس مذبور را در اختیار برنامه قرار داد.
اما برای جلوگیری از پیچیدگی بیش از حد و دور شدن از مبحث اصلی، برای پیاده‌سازی 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 پیاده‌سازی کلاس‌های مرتبط برای استفاده از EF Code First، از کلاس ResourceData که در بخش اول نیز نشان داده شده بود، برای کپسوله کردن ارتباط با داده‌ها استفاده می‌شود که نمونه‌ای 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 نمونه‌ای 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: درصورتی‌که 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 قبلا هم به آن اشاره شده بود باید درنظر 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 پرووایدرهای کلی و محلی به همراه تمام محتویاتشان (مثلا 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 قبلا توضیح داده شد کشِ 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 درصورت عدم وجود، مقدار آن مستقیما از دیتابیس درخواست می‌شود. سپس این مقدار درخواستی ابتدا درون متغیر کشینگ ذخیره شده (به همراه بلاک lock) و درنهایت 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 ازبین برود.

پیاده‌سازی 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 به‌کار می‌رود این است که درهر 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 پیاده‌سازی 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];
}
```

دراینجا 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 کاملا 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 پیاده‌سازی 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. مثلا 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 مذبور 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 پیاده‌سازی 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.CurrentUICulture;
      _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. درصورتی‌که این پارامتر نال باشد مقدار کالچر UI در ثرد جاری برای آن درنظر گرفته می‌شود. مقدار پیش‌فرضی که برای این پارامتر درنظر گرفته شده است، null است.

پارامتر بعدی کالچر خنثی موردنظر کاربر است. این کالچری است که درصورت رسیدن enumerator به آن کار پایان خواهد یافت. درواقع کالچر پایانی enumerator است. این پارامتر می‌تواند نال باشد. مقدار پیش‌فرضی که برای این پارامتر درنظر گرفته شده است، null است.

پارمتر آخر هم تعیین می‌کند که آیا enumerator از کالچرهای والد استفاده بکند یا خیر. مقدار پیش‌فرضی که برای این پارامتر درنظر 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 درصورت 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. درصورت برقراری شرط، کار این حلقه با برگشت CultureInfo.InvariantCulture پایان می‌یابد. InvariantCulture کالچر بدون زبان و فرهنگ و موقعیت مکانی است که درواقع به عنوان کالچر والد تمام کالچرها درنظر 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 درصورتی‌که 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.
درانتها نیز درصورتی‌که با شرایط موجود، قبلا کالچر Invariant برگشت داده نشده باشد این کالچر نیز yield می‌شود. درواقع درصورتی‌که 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 اینصورت 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 موردنیاز برای fallback یافته می‌شوند. در اینجا از مقادیر پیش‌فرض دو پارامتر دیگر کانستراکتور کلاس CultureFallbackProvider استفاده شده است.
سپس به ازای هر کالچر یافته شده مقدار ورودی درخواستی بدست آمده و درصورتی‌که نال نباشد (یعنی ورودی موردنظر برای کالچر جاری یافته شود) آن مقدار برگشت داده می‌شود و درصورتی‌که نال باشد عملیات برای کالچر بعدی ادامه می‌یابد.
درصورتی‌که ورودی درخواستی یافته نشود (خروج از حلقه بدون برگشت مقداری برای ورودی منبع درخواستی) استثنای KeyNotFoundException صادر می‌شود تا کاربر را از اشتباه رخداده مطلع سازد.

آزمایش پرووایدر سفارشی

ابتدا تنظیمات موردنیاز 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 درنظر 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 پیش‌فرض، 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:

می‌بینید که با توجه به عدم وجود مقداری برای Label2.Text برای کالچر fa، عملیات fallback اتفاق افتاده است.

بحث و نتیجه‌گیری

کار تولید یک پرووایدر منابع سفارشی دیتابیسی 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 نویس 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 درنهایت 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
لطفا 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
-2جدولی 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 چنینی 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 بسیار زیاد (چند صد هزار و یا بیشتر) تغییر این ساختار برای رسیدن به کارایی مناسب میتونه مفید باشه.
درهرصورت 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 درنهایت 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 ماژول 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 تخیلی هستند. حتی مایکروسافت 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 ۱۷:۹

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، Culture، ISO، RTL و ...) و جدول دیگم برای ریسورس ها(کلید، اسم، مقدار)
در واقع میخوام مقادیر ریسورس‌ها با اضافه و حذف شدن یک زبان به سیستم مدیریت بشه.
میخواستم ببینم که چه پیشنهادی برای این حالت دارید؟

عنوان: Globalization در ASP.NET MVC - قسمت هفتم
نویسنده: یوسف نژاد
تاریخ: ۱۳۹۲/۰۳/۱۲ ۱۳:۳۰
آدرس: www.dotnettips.info
گروه‌ها: Localization, Globalization, Internationalization, Resource, Custom Resource Provider, Database Resource Provider

در قسمت قبل مطالب تکمیلی تولید پرووایدر سفارشی منابع دیتابیسی ارائه شد. در این قسمت نحوه بروزرسانی ورودی‌های منابع در زمان اجرا بحث می‌شود.

تولید یک پرووایدر منابع دیتابیسی - 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 درنهایت بسته به شرایط موجود انتخاب روش مناسب برعهده خود برنامه‌نویس است.

مثلا برای پرووایدر سفارشی دیتابیسی 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 درصورت 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 درنظر داشت این است که مدیریت تمامی نمونه‌های تولیدشده از کلاس‌های موردبحث کاملا 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. مثلا یک کلاس حاول 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. ازآنجاکه سیستم مدیریت منابع ASP.NET از ریفلکشن برای تولید نمونه‌های موردنیاز خود استفاده می‌کند، بنابراین این تغییر تاثیری 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 این‌کار باید تغییراتی در کلاس DbResourceManager داده شود تا بتوان این کلاس را از تغییرات بوجود آمده آگاه ساخت. روشی که من برای این کار درنظر 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();
    }
```

```
  }
}
```

ازآنجاکه 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 List<IDbCachedResourceManager> تعریف شده است و برای جلوگیری از دسترسی بیرونی، سطح دسترسی آن internal درنظر گرفته شده است.

در کانستراکتور استاتیک این کلاس (اطلاعات بیشتر درباره static constructor در اینجا) این پراپرتی با مقداردهی به یک نمونه تازه از لیست، اصطلاحا 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 مشخص‌کردن 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 متاسفانه 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 درصورت 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 درنظر 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 دیتابیسی مربوطه از لینک زیر قابل دریافت است:

DbResourceProvider.rar

برای استفاده از این مثال ابتدا باید کتابخانه Entity Framework (با نام EntityFramework.dll) را مثلا 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/wwdbresourceprovider

نظرات خوانندگان

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

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: ۱۳۹۲/۰۳/۱۲ ۱۴:۳۶

مطلب شما کاملا صحیح است.
ممنون بابت یادآوری.

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

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: ۱۳۹۲/۰۳/۳۰ ۱۰:۳۰

متاسفانه امکان استفاده مستقیم از این پرووایدرهای سفارشی در این attributeها در MVC میسر نیست. این attributeها به جای استفاده از پرووایدر منابع برای استخراج مقادیر ورودی‌ها طوری طراحی شده اند که با استفاده از Reflection از داده‌های ارائه شده مقادیر را از کلاس و پراپرتی مربوطه استخراج کنند. بنابراین در این attributeها نمیتوان جایی برای استفاده از پرووایدرهای منابع یافت.

برای حل این مشکل چندین راه حل وجود دارد:
مثلا attributeهای موردنیاز توسط خود برنامه نویس پیاده‌سازی شوند.
یا اینکه یک کلاس مخصوص ایجاد کرد و استخراج مقادیر ورودی‌های منابع را در آن پیاده‌سازی کرد و در attributeهای موردنیاز از نام این کلاس و پراپرتی‌های درون آن استفاده کرد.
یا اگر از فایلهای resx. استفاده می‌شود یک ابزار سفارشی برای تولید کلاس مرتبط با منبع اصلی مثل ابزار توکار ویژوال استودیو (PublicResXFileCodeGenerator) تولید کرد تا کلاس‌های تولیدی به جای استفاده از ResourceManager از پرووایدر منابع استفاده کند (با استفاده از متدهای موجود در HttpContext).
البته این روش‌ها برای حل مشکلات مربوطه در MVC در ادامه این سری شرح داده می‌شوند.

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

ممنون از شما.

نویسنده: محسن موسوی
تاریخ: ۱۳۹۲/۰۶/۰۳ ۱۷:۳۹

با تشکر از زحمات شما
اینجا بیان شده زمانیکه از اسمبلی دیگری برای resourceها استفاده میکنید فقط میتوان **global resources** را پوشش داد.
بنابراین برای استفاده از کلاس LocalDbResourceProvider بایستی تغییراتی صورت بگیره.
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: ۱۳۹۲/۰۷/۰۸ ۹:۵۸

مهندس عزیز با تشکر از کار گرانقدر شما
یک سوال؟
چگونه می‌توان الگوی کار را در این پروایدر گنجاند؟
آیا اصلا چنین امکانی دارد یا خیر؟