

همانطوری که میدونید نسخه [MVC 4 RC](#) در دسترس قرار گرفته و خالی از لطف نیست که یک بررسی درباره امکانات جدیدش انجام بشه. ابتدا سعی میکنم یک لیست کلی از امکانات این تکنولوژی داشته باشیم و بعد نگاهی هم به Razor و تغییراتش خواهیم داشت.

ASP.NET Web API

Refreshed and modernized default project templates

New mobile project template

Many new features to support mobile apps

Recipes to customize code generation

Enhanced support for asynchronous methods

لیست فوق شامل برترین ویژگی‌های این نسخه است که در پستهای آینده هر کدام از این‌ها بررسی خواهند شد.

تغییرات Razor

Razor از نسخه MVC4 Beta شاهد تغییرات و بهبودهایی بود که این تغییرات بنیادی و رادیکالی نبودند و فقط در جهت بهبود حس کاربری آن صورت گرفت. این حس از آن جهت است که شما نیاز به نوشتن کد کمتری دارید.

استفاده نکردن از @Url.Content

در نسخه قبلی از امکان ذکر شده برای مشخص کردن مسیرهای CSS و فایل‌های JS هم استفاده میشد حالا بجای استفاده از آن میشود:

در نسخه‌های قبلی:

```
<script src="@Url.Content("~/Scripts/Site.js")"></script>
```

در نسخه 4:

```
<script src("~/Scripts/Site.js")></script>
```

زمانی که Razor حرف را ~/ تشخیص می‌دهد خروجی یکسانی با حالت قبلی برای ما درست می‌کند.

شرط‌ها (Conditions)

در نسخه قبلی برای استفاده از attribute ها که ممکن بود Null باشند مجبور به چک کردن آنها بودیم:

```
<div @{if (myClass != null) { <text>class="@myClass"</text> } }>Content</div>
```

اما حالا با خیال راحت می‌توان نوشت:

```
<div class="@myClass">Content</div>
```

که اگر attribute ما null باشد به صورت اتوماتیک تشخیص داده میشود و کد زیر رندر میشود

```
<div>Content</div>
```

در ضمناً کد بالا فقط مربوط به چک کردن Nullable نیست و از آن می‌توان در Boolean ها هم استفاده کرد.

```
<input checked="@ViewBag.Checked" type="checkbox"/>
```

که اگر مقدار True نباشد:

```
<input type="checkbox"/>
```

بطور خلاصه همیشه گفت MVC4 تغییراتش نسبت به نسخه قبلی تو خیلی از زمینه‌ها مربوط میشه بهبود ابزارهای موجود در کل کار با این ابزار بسیار برای من لذت بخشه.
ادامه دارد...

Postal کتابخانه ای برای تولید و ارسال ایمیل توسط نماهای ASP.NET MVC است. برای شروع این کتابخانه را به پروژه خود اضافه کنید. پنجره **Package Manager Console** را باز کرده و فرمان زیر را اجرا کنید.

```
PM> Install-Package Postal
```

شروع به کار با Postal

نحوه استفاده از Postal در کنترلرهای خود را در کد زیر مشاهده می‌کنید.

```
using Postal;

public class HomeController : Controller
{
    public ActionResult Index()
    {
        dynamic email = new Email("Example");
        email.To = "webninja@example.com";
        email.FunnyLink = DB.GetRandomLolcatLink();
        email.Send();
        return View();
    }
}
```

Postal نمای ایمیل را در مسیر Views\Emails\Example.cshtml جستجو می‌کند.

```
To: @ViewBag.To
From: lolcats@website.com
Subject: Important Message

Hello,
You wanted important web links right?
Check out this: @ViewBag.FunnyLink

<3
```

پیکربندی SMTP

Postal ایمیل‌ها را توسط **SmtpClient** ارسال می‌کند که در فریم ورک دات نت موجود است. تنظیمات SMTP را می‌توانید در فایل web.config خود پیکربندی کنید. برای اطلاعات بیشتر به [MSDN Documentation](http://msdn.microsoft.com) مراجعه کنید.

```
<configuration>
...
<system.net>
  <mailSettings>
    <smtp deliveryMethod="network">
      <network host="example.org" port="25" defaultCredentials="true"/>
    </smtp>
  </mailSettings>
</system.net>
...
</configuration>
```

ایمیل‌های Strongly-typed

همه خوششان نمی‌آید از آبجکت‌های دینامیک استفاده کنند. علاوه بر آن آبجکت‌های دینامیک مشکلاتی هم دارند. مثلاً قابلیت

IntelliSense و یا Compile-time error را نخواهید داشت.
 قدم اول - کلاسی تعریف کنید که از Email ارث بری می‌کند.

```
namespace App.Models
{
    public class ExampleEmail : Email
    {
        public string To { get; set; }
        public string Message { get; set; }
    }
}
```

قدم دوم - از این کلاس استفاده کنید!

```
public void Send()
{
    var email = new ExampleEmail
    {
        To = "hello@world.com",
        Message = "Strong typed message"
    };
    email.Send();
}
```

قدم سوم - نمایی ایجاد کنید که از مدل شما استفاده می‌کند. نام نما، بر اساس نام کلاس مدل انتخاب شده است. بنابراین مثلاً *ExampleEmail* نمایی با نام *Example.cshtml* لازم دارد.

```
@model App.Models.ExampleEmail
To: @Model.To
From: postal@example.com
Subject: Example

Hello,
@Model.Message
Thanks!
```

آزمون‌های واحد (Unit Testing)

هنگام تست کردن کدهایی که با Postal کار می‌کنند، یکی از کارهایی که می‌خواهید انجام دهید حصول اطمینان از ارسال شدن ایمیل‌ها است. البته در بدنه تست‌ها نمی‌خواهیم هیچ ایمیلی ارسال شود. Postal یک قرارداد بنام **IService** و یک پیاده‌سازی پیش فرض از آن بنام **EmailService** ارائه می‌کند، که در واقع ایمیل‌ها را ارسال هم می‌کند. با در نظر گرفتن این پیش فرض که شما از یک IoC Container استفاده می‌کنید (مانند StructureMap, Ninject)، آن را طوری پیکربندی کنید تا یک نمونه از IService به کنترلرها تزریق کند. سپس از این سرویس برای ارسال آبجکت‌های ایمیل‌ها استفاده کنید (بجای فراخوانی متد Email.Send()).

```
public class ExampleController : Controller
{
    public ExampleController(IService emailService)
    {
        this.emailService = emailService;
    }

    readonly IService emailService;

    public ActionResult Index()
    {
        dynamic email = new Email("Example");
        // ...
        emailService.Send(email);
        return View();
    }
}
```

این کنترلر را با ساختن یک Mock از اینترفیس IEmailService تست کنید. یک مثال با استفاده از [FakeItEasy](#) را در زیر مشاهده می‌کنید.

```
[Test]
public void ItSendsEmail()
{
    var emailService = A.Fake<IEmailService>();
    var controller = new ExampleController(emailService);
    controller.Index();
    A.CallTo(() => emailService.Send(A<Email>._))
        .MustHaveHappened();
}
```

ایمیل‌های ساده و HTML

Postal ارسال ایمیل‌های ساده (plain text) و HTML را بسیار ساده می‌کند.

قدم اول - نمای اصلی را بسازید. این نما headerها را خواهد داشت و نماهای مورد نیاز را هم رفرنس می‌کند. مسیر نما `~\Views\Emails\Example.cshtml` است.

```
To: test@test.com
From: example@test.com
Subject: Fancy email
Views: Text, Html
```

قدم دوم - نمای تکست را ایجاد کنید. به قوانین نامگذاری دقت کنید، `Example.cshtml` به `Example.Text.cshtml` تغییر یافته. مسیر فایل `Views\Emails\Example.Text.cshtml` است.

```
Content-Type: text/plain; charset=utf-8
```

```
Hello @ViewBag.PersonName,
This is a message
```

دقت داشته باشید که تنها یک Content-Type باید تعریف کنید.

قدم سوم - نمای HTML را ایجاد کنید (باز هم فقط با یک Content-Type). مسیر فایل `~\Views\Emails\Example.Html.cshtml` است.

```
Content-Type: text/html; charset=utf-8
```

```
<html>
<body>
    <p>Hello @ViewBag.PersonName,</p>
    <p>This is a message</p>
</body>
</html>
```

ضمیمه‌ها

برای افزودن ضمائم خود به ایمیل‌ها، متد Attach را فراخوانی کنید.

```
dynamic email = new Email("Example");
email.Attach(new Attachment("c:\\attachment.txt"));
email.Send();
```

جاسازی تصاویر در ایمیل‌ها

Postal یک HTML Helper دارد که امکان جاسازی (embedding) تصاویر در ایمیل‌ها را فراهم می‌کند. دیگر نیازی نیست به یک URL خارجی اشاره کنید.

ابتدا مطمئن شوید که فایل web.config شما فضای نام Postal را اضافه کرده است. این کار دسترسی به HTML Helper مذکور در نمای‌های ایمیل را ممکن می‌سازد.

```
<configuration>
  <system.web.webPages.razor>
    <pages pageBaseType="System.Web.Mvc.WebViewPage">
      <namespaces>
        <add namespace="Postal" />
      </namespaces>
    </pages>
  </system.web.webPages.razor>
</configuration>
```

متد **EmbedImage** تصویر مورد نظر را در ایمیل شما جاسازی می‌کند و توسط یک تگ `` آن را رفرنس می‌کند.

```
To: john@example.org
From: app@example.org
Subject: Image
```

```
@Html.EmbedImage("~/content/postal.jpg")
```

Postal سعی می‌کند تا نام فایل تصویر را، بر اساس مسیر تقریبی ریشه اپلیکیشن شما تعیین کند.

Postal بیرون از ASP.NET

Postal می‌تواند نماهای ایمیل‌ها را بیرون از فضای ASP.NET رندر کند. مثلاً در یک اپلیکیشن کنسول یا یک سرویس ویندوز. این امر توسط یک View Engine سفارشی میسر می‌شود. تنها نماهای Razor پشتیبانی می‌شوند. نمونه کدی را در زیر مشاهده می‌کنید.

```
using Postal;

class Program
{
    static void Main(string[] args)
    {
        // Get the path to the directory containing views
        var viewsPath = Path.GetFullPath(@"..\..\Views");

        var engines = new ViewEngineCollection();
        engines.Add(new FileSystemRazorViewEngine(viewsPath));

        var service = new EmailService(engines);

        dynamic email = new Email("Test");
        // Will look for Test.cshtml or Test.vbhtml in Views directory.
        email.Message = "Hello, non-asp.net world!";
        service.Send(email);
    }
}
```

محدودیت‌ها: نمی‌توانید برای نمای ایمیل‌ها Layoutها استفاده کنید. همچنین در نماهای خود تنها از مدل‌ها (Models) می‌توانید استفاده کنید، و نه ViewBag.

Email Headers: برای در بر داشتن نام، در آدرس ایمیل از فرمت زیر استفاده کنید.

```
To: John Smith <john@example.org>
```

Multiple Values: برخی از headerها می‌توانند چند مقدار داشته باشند. مثلاً Bcc و CC. اینگونه مقادیر را می‌توانید به دو روش در نمای خود تعریف کنید:
جدا کردن مقادیر با کاما:

```
Bcc: john@smith.com, harry@green.com
Subject: Example
```

etc

و یا تکرار header:

Bcc: john@smith.com
 Bcc: harry@green.com
 Subject: Example

etc

ساختن ایمیل بدون ارسال آن

لازم نیست برای ارسال ایمیل هایتان به Postal تکیه کنید. در عوض می‌توانید یک آبجکت از نوع `System.Net.Mail.MailMessage` تولید کنید و به هر نحوی که می‌خواهید آن را پردازش کنید. مثلاً شاید بخواهید بجای ارسال ایمیل‌ها، آنها را به یک صف پیام مثل MSMQ انتقال دهید یا بعداً توسط سرویس دیگری ارسال شوند. این آبجکت `MailMessage` تمامی Header ها، محتوای اصلی ایمیل و ضمائم را در بر خواهد گرفت.

کلاس `EmailService` در Postal متدی با نام `CreateMailMessage` فراهم می‌کند.

```
public class ExampleController : Controller
{
    public ExampleController(IEmailService emailService)
    {
        this.emailService = emailService;
    }

    readonly IEmailService emailService;

    public ActionResult Index()
    {
        dynamic email = new Email("Example");
        // ...

        var message = emailService.CreateMailMessage(email);
        CustomProcessMailMessage(message);

        return View();
    }
}
```

در این پست با امکانات اصلی کتابخانه Postal آشنا شدید و دیدید که به سادگی می‌توانید ایمیل‌های Razor بسازید. برای اطلاعات بیشتر لطفاً به [سایت پروژه Postal](#) مراجعه کنید.

در این مقاله سعی داریم تا سرعت یافت و جستجوی View های متناظر با هر اکشن را در View Engine، با پیاده سازی قابلیت Caching نتیجه یافت آدرس فیزیکی view ها در درخواست های متوالی، افزایش دهیم تا عملاً بازده سیستم را تا حدودی بهبود ببخشیم.

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

نکته ای که وجود دارد این هست که علاوه بر مفید بودن این امر و بهبود سرعت در درخواست های متوالی برای اکشن متدها، این عمل با توجه به مشاهدات بنده از سورس MVC علاوه بر مفید بودن، تا حدودی هزینه بر هم هست و هزینه ای که متوجه سیستم می شود شامل مسائل مدیریت توکار حافظه کش توسط MVC است که مسائلی مانند سیاست های مدیریت زمان انقضای مداخل موجود در حافظه ی کش اختصاص داده شده به Lookup Caching و مدیریت مسائل thread-safe و ... را شامل می شود.

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

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

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

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

```
public class CustomViewCache : IViewLocationCache
{
    private readonly static string s_key = "_customLookupCach" + Guid.NewGuid().ToString();
    private readonly IViewLocationCache _cache;

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

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



```

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

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

```

و به صورت زیر می‌توانید از آن استفاده کنید:

```

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

```

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

نظرات خوانندگان

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

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

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

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

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

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

[Optimizing ASP.NET MVC view lookup performance](#)

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