ASP.NET MVC 4

&

WEB API

Jamie Kurtz

ASP.NET MVC 4
&
Web API

Jamie Kurtz

مهندس سید منصور عمرانی

انتشارات پندار پارس

سرشناسه	:	کورتس، جیمی Kurtz, Jamie
عنوان و نام پدیدآور	:	ASP.NET MVC 4 & Web API /؛ مترجم سیدمنصور عمرانی.
مشخصات نشر	:	تهران : پندار پارس، 1392.
مشخصات ظاهری	:	144 ص.: مصور، جدول.
شابک	:	978-600-6529-41-7:ریال 79000
وضعیت فهرست نویسی	:	فیپا
ترجمه عنوان	:	ای اس پی . نت ام وی سی فور ...
موضوع	:	صفحه‌های سرور فعال
موضوع	:	وب- - سایت‌ها—طراحی
موضوع	:	ای.اس.پی. (پروتکل شبکه کامپیوتری)
شناسه افزوده	:	عمرانی، سیدمنصور، 1356 -، مترجم
رده بندی کنگره	:	TK5105/8885 1392 9الف9ک/
رده بندی دیویی	:	276/005
شماره کتابشناسی ملی	:	3246594

انتشارات پندارپارس

دفتر فروش: انقلاب، ابتدای کارگرجنوبی، کوی رشتچی، شماره 14، واحد 16 **www.pendarepars.com**

تلفن: 66572335 - تلفکس: 66926578 همراه: 09122452348 **info@pendarepars.com**

نام کتاب	:	**ASP.NET MVC 4 & Web API**
ناشر	:	**انتشارات پندار پارس**
تالیف	:	**Jamie Kurtz**
ترجمه	:	**مهندس سید منصور عمرانی**
چاپ نخست	:	**مهر 92**
شمارگان	:	**1000 نسخه**
لیتوگرافی، چاپ، صحافی	:	**ترام‌سنج، فرشیوه، خیام**
قیمت	:	**7900 تومان** **شابک** : 978-600-6529-41-7

پیش‌گفتار مترجم

یکی از ویژگی‌های برجسته‌ی وب این است که کاربران با استفاده از یک مرورگر، قادر به پیمایش همه‌ی سایت‌ها و برنامه‌های تحت وب هستند و می‌توانند با آنها تعامل داشته باشند، بدون آنکه نیازی باشد مرورگر آنها چیزی از API ویژه‌ی برنامه‌های تحت وب بداند. از آن سو با گذشت نزدیک به یک دهه از معرفی معماری سرویس-محور یا SOA که ایده‌ی نوشتن برنامه بر اساس سرویس‌های SOAP را معرفی می‌کند، این معماری هرچه بیشتر اهمیت و جایگاه خود را در توسعه‌ی سیستم‌های توزیع شده‌ی سست اتصال و برآورده کردن اصل جداسازی دغدغه‌ها یا SoC و قابلیت استفاده‌ی مجدد نشان می‌دهد.

با وجودی که پروتکل SOAP به شکل زیبایی، امکان استفاده‌ی آرام و بی‌سر و صدا از سرویس‌ها را فراهم می‌کرد، اما کلاینت‌ها وابسته به قرارداد تعریف شده توسط سرویس‌ها بوده و می‌بایست پیش از بهره‌مندی از سرویس، از API آن آگاه بوده و خود را با ساختار سرویس مطابقت می‌دادند. این مسئله منجر به شکل‌گیری اتصال محکمی میان کلاینت‌ها و سرویس می‌شد. به همین دلیل امروزه مدل REST، که مدل دیگری برای نوشتن سرویس است، از محبوبیت و اقبال زیادی برخوردار شده است، حتی با وجودی که این مدل چندین سال پیش از معماری SOA معرفی شد[1].

بدون شک یکی از عوامل تأثیرگذار بر این مسئله، ادوات و رسانه‌های دیجیتال مختلفی چون موبایل‌ها، تبلت‌ها، گوشی‌های هوشمند و PDAها هستند. زیرا این رسانه‌ها از نرم‌افزارهای سبکی تشکیل می‌شوند که نیازمندی‌های ساده‌تری دارند. در چنین شرایطی، استفاده از مدلی که بتواند بر اساس فناوری‌های ساده و از پیش موجود وب مانند HTML، جاوااسکریپت و Ajax، بدون آگاهی از API یک سرویس با آن تعامل داشته باشد، گزینه‌ی بهتری برای نوشتن برنامه‌های کلاینت محسوب می‌شود. زیرا کلاینت را از پیچیدگی‌های پروتکل SOAP بی‌نیاز می‌کند. البته نباید پنداشت REST به‌عنوان جایگزین SOAP معرفی شده است. اتفاقا REST از برآورده کردن برخی سناریوها عاجز است. با این وجود این مدل یکی از بهترین گزینه‌ها در پیاده‌سازی API برنامه‌های وب برای گوشی‌های تلفن همراه و تبلت‌ها محسوب می‌شود.

در کتاب «ASP.NET MVC 4 و Web API» جمی کورتز مؤلف کتاب، نخست خلاصه‌ای از پروتکل SOAP و تفاوت آن با مدل REST بیان می‌کند. سپس توضیح می‌دهد چگونه بر اساس مدل کمال RMM که روید فیدلینگ مبدع REST تعریف نموده، می‌توان API یک سرویس را به‌شکل RESTful تعریف کرد. پس از آن با مرور قابلیت Web API در ASP.NET MVC 4 مزایای این سکو را برای پیاده‌سازی یک سرویس RESTful بیان می‌کند و در ادامه‌ی کتاب، با پیاده‌سازی سرویسی برای مدیریت وظیفه‌ها، نحوه‌ی پیاده‌سازی عملی یک سرویس REST را با استفاده از ASP.NET MVC 4 و Web API نشان می‌دهد.

یکی از نکات ارزشمند کتاب، سادگی و مختصر بودن بیان آن در معرفی مدل REST و نحوه‌ی تعریف یک API با ویژگی RESTful است. افزون بر این مؤلف کتاب، خواننده را با نحوه‌ی پیاده‌سازی حرفه‌ای یک سرویس REST بر

اساس به‌روزترین و رایج‌ترین ابزارها در سکوی .NET. مانند NHibernate، Log4Net و Ninject آشنا کرده و نحوه‌ی ایمن‌سازی سرویس REST را توضیح می‌دهد. به طور کلی این کتاب برای شروع یادگیری مدل REST و ASP.NET MVC 4 Web API کتاب مناسبی است، اما در عین حال مطالب زیادی در این زمینه، به‌ویژه درباره‌ی امنیت و ایمن‌سازی و تعامل با سرویس‌های REST وجود دارد. مترجم توصیه می‌کند پس از خواندن کتاب، حتما آموخته‌های خود را به طور عملی با پیاده‌سازی یک سرویس REST شخصی با استفاده از ASP.NET MVC 4 Web API اجرا کنید. زیرا در عمل، با نکته‌ها و چالش‌های بیشتری روبرو خواهید شد که باعث می‌شود مهارت بیشتری در این زمینه به‌دست آورید.

در پایان، مترجم امیدوار است ترجمه‌ی انجام شده کیفیت لازم را داشته باشد و خواننده، مباحث آموزش داده شده را به راحتی فرا بگیرد. در صورت داشتن هرگونه نظر، پیشنهاد، انتقاد یا پرسشی می‌توانید به‌وسیله‌ی آدرس ایمیل mansoor.omrani@gmail.com با مترجم تماس بگیرید.

سید منصور عمرانی

تابستان 92

فهرست

فصل 1. ASP.NET MVC به عنوان یک چهارچوب کاری سرویس‌گرا

مایکروسافت در سال‌های پس از معرفی چهارچوب کاری .NET. رویکردهای مختلفی برای ساخت برنامه‌های سرویس‌گرا[1] داشته است. در نخستین نسخه‌ی .NET. در سال 2002، به سادگی می‌توانستید یک وب‌سرویس XML مبتنی بر ASMX بنویسید. هر برنامه‌ای، چه .NET. یا غیر .NET. نیز می‌توانست متدهای وب‌سرویس را فراخوانی کند. آن وب‌سرویس‌ها از نسخه‌های مختلف پروتکل SOAP استفاده می‌کردند اما وب‌سرویس‌های آن زمان، تنها تحت پروتکل HTTP قابل استفاده بودند.

افزون بر وب‌سرویس‌ها، در .NET 1.0. قابلیتی به نام Remoting فراهم شده بود که برنامه‌نویسان با استفاده از آن می‌توانستند سرویس‌هایی بنویسند که به پروتکل HTTP محدود نبود. همانند وب‌سرویس‌های ASMX، قابلیت Remoting نیز بر اساس متدهایی کار می‌کند که از راه دور توسط کلاینت‌ها فراخوانی می‌شود و برای آنها قابلیت فعال‌سازی اشیاء[2] و زمینه‌ی جلسه[3] را فراهم می‌کند. کسی که می‌خواهد از شیء راه دور استفاده کند با یک شیء واسطه یا پراکسی کار می‌کند که فراخوانی متدهای آن در پشت صحنه، به یک درخواست شبکه‌ای تبدیل می‌شود. ماشین اجرای .NET. نیز سریال‌سازی و هدایت داده‌ها را میان شیء پراکسی در سمت کلاینت و شیء سرویس فعال شده در سمت سرور، مدیریت می‌کند.

در پایان سال 2006، مایکروسافت .NET 3.0. را منتشر کرد که حاوی فناوری دیگری به نام WCF یا Windows Communication Foundation بود. WCF نه تنها جایگزین وب‌سرویس‌های ASMX و قابلیت Remoting شد، بلکه جهش بسیار بزرگی در مسیر انعطاف‌پذیری، پیکربندی، توسعه‌پذیری و پشتیبانی از دیگر استانداردهای امنیتی و SOAP ایجاد کرد.

برای نمونه، با استفاده از WCF می‌توانید یک سرویس غیر HTTP بنویسید که با استفاده از توکن‌های SAML از تصدیق هویت پشتیبانی می‌کند و سپس آن را در یک سرویس ویندوز میزبانی کنید. این قابلیت به همراه دیگر

---

[1] service-oriented application
[2] object activation
[3] Session context

امکانات WCF، به‌میزان چشمگیری دامنه‌ی سناریوهایی را که .NET. در آنها می‌توانست برای ساخت یک برنامه‌ی سرویس‌گرا یا سرویس محور به‌کار برود، افزایش داد.

اطلاعات بیشتر درباره‌ی WCF

اگر مایلید بیشتر با WCF آشنا شوید، توصیه می‌کنم کتاب Programming WCF Services نوشته‌ی جوال لاوی از انتشارات O'Reilly در سال 2007 یا کتاب Essential Windows Communication Foundation نوشته‌ی استیو رزنیک، ریچارد کرین و کریس بوئن از انتشارات Addison-Wesley در سال 2008 را بخوانید. هر دو کتاب یاد شده برای کسانی که تازه می‌خواهند WCF را یاد بگیرند و هم برای برنامه‌نویسان کهنه‌کار مناسب هستند، زیرا مباحث آموزشی آنها از مبتدی تا پیشرفته است.

اگر می‌خواهید بین دو برنامه ارتباط ایجاد کنید، صرفنظر از این که هر دو در یک کامپیوتر در حال کار هستند یا هزاران کیلومتر دور از هم قرار دارند، مطمئن باشید WCF از پس آن بر می‌آید. اگر هم قابلیت‌های از پیش آماده‌ی WCF برای شما کفایت نمی‌کند، مدل فوق‌العاده انعطاف‌پذیر WCF امکانات بسیار گسترده‌ای فراهم می‌کند که توسط آنها می‌توانید در سرویس WCF خود هر چیزی که فکرش را بکنید ایجاد کنید.

اما اینجا همان جایی است که در مسیر تحولات جدید برای کسب توانایی‌ها و انعطاف‌پذیری‌های بیشتر، کمی به راست گردش می‌کنیم و مسیرمان را به سمت چیز ساده‌تری تغییر می‌دهیم که برای برخی سناریوهای مشخص، هدفمندتر شده است.

سرزمین جاوااسکریپت و رسانه‌های موبایلی

با وجود بیشتر جنبه‌هایی که اینترنت در طول دو دهه‌ی گذشته در آنها رشد داشته است، تمام سایت‌ها و صفحات وب برای هر چیزی به جز دستکاری ساده‌ی تگ‌های HTML، به کُدهای سمت سرور متکی بودند. اما به‌تازگی ابزارها و فریم‌وُرک‌های جاوااسکریپت و AJAX مانند jQuery، پروتکل جدید HTML5 و برخی ترفندهای CSS، تقاضا برای سرویس‌هایی را افزایش داده‌اند که ساده‌تر از سرویس‌های پیچیده‌ی برنامه‌های رومیزی باشند و کلاینت آنها تنها از صفحاتی تشکیل می‌شود که به‌طور ساده فقط می‌خواهد بسته‌های کوچک اطلاعات را از سرویس دریافت کند.

در این سناریو ارتباط کلاینت با سرویس از قبل تحت پروتکل HTTP انجام می‌شود. زیرا سایت‌های وب خودشان با پروتکل HTTP کار می‌کنند. افزون بر این، گزینه‌های امنیتی مرورگرها نسبت به یک برنامه‌ی غیر مرورگر (مانند یک برنامه‌ی رومیزی)، بسیار ساده‌تر است. به همین دلیل پشتیبانی از انواع و اقسام استانداردهای امنیتی SOAP برای چنین سرویس‌هایی لازم نیست.

در سرویس‌های این سناریوها، افزون بر پروتکل ساده‌تر و نیازمندی‌های امنیتی کمتر، معمولا صفحات وب برای ارتباط با سرویس راه دور به‌جای پیام‌های دودویی از پیام‌های ساده‌ی متنی استفاده می‌کنند. به همین دلیل برای مبادله‌ی اطلاعات تنها کافی است سرویس راه دور از XML یا JSON پشتیبانی کند.

اما افزون بر برنامه‌های وب، امروزه تلفن‌های هوشمند و تبلت‌ها با پشتیبانی از برنامه‌های موبایلی کوچک و هوشمند، تقاضای بسیار گسترده‌ای برای چنین سرویس‌هایی ایجاد کرده‌اند. طبیعت این سرویس‌ها به میزان زیادی همانند

سرویس‌های سایت‌های AJAX است. برای نمونه، آنها نیز تحت HTTP استفاده می‌شوند، مبادله‌ی داده‌ای آنها متنی است، حجم کمی از اطلاعات را مبادله می‌کنند و برای فراهم کردن تجربه‌ی بهتری برای کاربر، مدل امنیتی آنها ساده‌تر است (مستلزم پیکربندی کمتری هستند که دردسر کمتری برای صاحب موبایل یا تبلت برای استفاده از برنامه ایجاد می‌کند). همچنین این سرویس‌ها باعث می‌شوند بتوان از آنها به یک شکل در موبایل‌ها و ادوات مختلف استفاده کرد. در واقع مستقل از سکو هستند.

کوته سخن، اکنون دنیای برنامه‌نویسی به فریم‌وُرک‌های سرویس‌محوری علاقه‌مند است که به‌طور پیش فرض و از ابتدا نیازمندی‌های سرویس‌های ساده و متنی HTTP را پوشش بدهد. اگرچه می‌توان چنین سرویس‌هایی را با WCF نوشت، اما WCF از ابتدا چنین امکاناتی را برای سرویس‌های خود فراهم نمی‌کند و برنامه‌نویس مجبور است هر بار آنها را از نو پیاده‌سازی کند. بدبختانه انعطاف‌پذیری و پیکربندی بسیار زیاد WCF باعث می‌شود به راحتی اشتباه کرده و چیزی را خراب کنید تا سرویس شما به درستی کار نکند.

در اینجا است که ASP.NET MVC وارد صحنه می‌شود.

مزایای استفاده از ASP.NET MVC

پس از آنکه مشاهده کردید در برخی سناریوها به تمام قابلیت‌ها و امکانات WCF نیاز ندارید، می‌توانید سراغ فریم‌وُرک دیگری مانند ASP.NET MVC بروید که سر و صدای کمتری در زمینه‌ی قابلیت‌ها و امکانات سرویس‌گرا دارد و در کنار آن، بسیار مفید نیز هست. در این قسمت برخی از مزایای ASP.NET MVC را توضیح می‌دهیم.

پیکربندی

درست به همان سادگی که می‌توانید سایت وبی با چند صفحه‌ی وب ایجاد کنید، ایجاد و راه‌اندازی یک سرویس MVC نیز به پیکربندی زیادی نیاز ندارد. در سرویس‌های MVC چیزی به‌نام endpoint و contract که WCF تعریف کرده است، وجود ندارد. همان‌گونه که خواهید دید، سرویس‌های MVC در مقایسه با سرویس‌های WCF بسیار سبک‌تر هستند. به طور ساده تنها به یک آدرس URL سبک REST، تعدادی آرگومان و یک پاسخ XML یا JSON نیاز دارند.

از ابتدا دارای خصلت REST

هنگامی که می‌خواهید سرویسی را با استفاده از ASP.NET MVC و Web API ایجاد کنید، بیشتر چیزهایی که برای پایبندی به اصول معماری REST نیاز دارید از پیش مهیا است. این مسئله به طور عمده از قابلیت مسیریابی در ASP.NET MVC سرچشمه می‌گیرد. بر خلاف WCF که سرویس‌های آن بر اساس آدرس یک فایل فیزیکی تعریف می‌شوند (آدرس فیزیکی یک کلاس سرویس یا فایل svc.)، سرویس‌های MVC به طور ساده تنها مسیرهایی هستند که به کنترلرها نگاشت می‌شوند و ویژگی REST دارند. در حقیقت، قابلیت مسیریابی در ASP.NET MVC از قبل با API یا واسط سبک REST هماهنگی دارد.

قابلیت مسیریابی برای درک نحوه‌ی استفاده از MVC برای ساخت سرویس‌ها حیاتی است. برای یادگیری بهتر، در این کتاب نحوه‌ی ساخت یک سرویس REST را در یک مثال عملی برای مدیریت وظیفه یا Task Management نشان

می‌دهیم که در آن گویی به ازای هر وظیفه یک متد مجزا داریم. این متد، شناسه‌ی یک وظیفه یا TaskId را دریافت کرده و مشخصات آن را بر می‌گرداند. پیاده‌سازی این سرویس در WCF می‌تواند بدین شکل باشد:

```
[ServiceContract]
public interface ITaskService
{
  [OperationContract]
  Task GetTask(long taskId);
}
public class TaskService : ITaskService
{
  private readonly IRepository _repository;
  public TaskService(IRepository repository)
  {
    _repository = repository;
  }
  public Task GetTask(long taskId)
  {
    return _repository.Get<Task>(taskId);
  }
}
```

اگر فایل svc. این سرویس و endpointهای آن به درستی پیکربندی شود، آدرس URL این سرویس می‌تواند چنین باشد:

```
http://MyServer/TaskService.svc
```

کسی که می‌خواهد از این سرویس استفاده کند باید یک درخواست SOAP برای استفاده از اکشنی به نام GetTask ارسال کند و شناسه‌ی وظیفه‌ی مورد نظر خود را به عنوان آرگومان متد GetTask مشخص کند. البته هنگامی که برنامه‌ی کلاینت NET. را با استفاده از WCF ایجاد می‌کنید، بیشتر پیچیدگی‌های پیاده‌سازی این کار، از پیش توسط WCF برای شما انجام می‌شود. با این وجود ارسال درخواست SOAP از طریق صفحات وب با استفاده از جاوااسکریپت، دردسر زیادی دارد و دیگر کسی وجود ندارد که پیچیدگی‌های مبادله‌ی اطلاعات تحت پروتکل SOAP را برای شما انجام بدهد. این در حالی است که نیازی هم به این پیچیدگی‌ها نیست.

از آن سو در ASP.NET MVC می‌توانیم این سرویس را به‌جای یک کلاس WCF، با یک کنترلر ساده پیاده‌سازی کنیم. متد بازیابی و برگرداندن وظیفه نیز داخل کلاس کنترلر تعریف می‌شود، اما بر خلاف سرویس WCF نیازی به تعریف contract ندارد. این کنترلر می‌تواند چنین چیزی باشد:

```
public class TasksController : Controller
{
  private readonly IRepository _repository;

  public TasksController(IRepository repository)
  {
    _repository = repository;
  }

  public ActionResult Get(long taskId)
  {
    return Json(_repository.Get<Task>(taskId));
```

```
  }
}
```

با مهیا بودن این کنترلر و مسیری که بتواند درخواست URL را به فراخوانی کنترلر و متد آن تبدیل کند (که از پیش توسط ASP.NET MVC مهیا است)، می‌توانید برای بازیابی یک وظیفه به طور ساده چنین آدرسی به کار ببرید:

http://MyServer/Task/Get/123

توجه کنید در این حالت نام متد Get باید در آدرس URL ذکر شود.

اکنون بگذارید همین مثال را با Web API ببینیم.

```
public class TasksController : ApiController
{
  private readonly IRepository _repository;

  public TasksController(IRepository repository)
  {
    _repository = repository;
  }

  public Task Get(long taskId)
  {
    return repository.Get<Task>(taskId);
  }
}
```

بزرگ‌ترین تغییری که به کلاس کنترلر پیشین دادیم این بود که به‌جای کلاس پایه‌ی Controller، آن را از کلاس ApiController مشتق کردیم. این کلاس به طور ویژه برای پشتیبانی از سرویس‌های REST طراحی شده است. تفاوت دیگر این سرویس Web API با سرویس MVC پیشین این است که نیازی نیست مقدار برگشتی متدهای کنترلر آن، از جنس ActionResult یا مشتقات آن باشد. به جای آن می‌توانید داخل این متدها به طور ساده خود شیء یا اشیاء درخواستی کلاینت را برگردانید. در آخر نیز به‌جای آدرس URL پیشین می‌توانید از آدرس ساده‌تر زیر استفاده کنید.

http://MyServer/Tasks/123

همان‌گونه که می‌بینید دیگر نیازی نیست نام متد Get را در آدرس URL بیاورید. زیرا در Web API، نحوه‌ی درخواست و فعل HTTP استفاده شده (GET، POST، PUT و مانند آن) به طور خودکار به متدهای کنترلر مورد فراخوانی نگاشت می‌شود. بدین ترتیب همان طور که در فصل بعد خواهید دید، می‌توانید API یا واسطی ایجاد کنید که هماهنگی بیشتری با قواعد و قوانین معماری REST دارد.

نکته‌ی مهمی که باید درک کنید این است که استفاده از سرویس Web API تنها به همان آدرس URL خلاصه می‌شود و به هیچ چیز دیگری نیاز نیست. نیازی نیست هیچ پیغام SOAP شلوغ و سنگینی برای استفاده از آن ایجاد کرده و همراه درخواست خود ارسال کنید. در حقیقت این مسئله یکی از اصول REST است که می‌گوید:

منابع باید از طریق آدرس‌های URI قابل دسترس باشند.

مرور سریعی بر REST

REST توسط فردی به‌نام روی فیدلینگ یکی از مولفان مشخصه‌های پروتکل HTTP ابداع شد. هدف از REST این بود که بتوان به جای پروتکل سنگین و شلوغ SOAP، از استانداردها و فناوری‌های از پیش موجود پروتکل HTTP استفاده کرد. به عنوان مثال توصیه می‌شود کسانی که می‌خواهند متدهای REST یا REST API بنویسند به جای نوشتن متدهای SOAP، ، تنها از فعل‌های HTTP استفاده کنند:

- GET
- POST
- PUT
- DELETE

همچنین REST، *منبع محور*[4] است. یعنی نقطه‌ی تمرکز واسط REST و فعل‌های HTTP استفاده از منابع (بازیابی یا دستکاری آنها) است. این منابع همان نام‌هایی هستند که در گفتگوی REST ذکر می‌شود (مانند وظیفه یا Task، کاربر یا User، مشتری یا Customer، سفارش یا Order و مانند آن). فعل‌های HTTP نیز با همین نام‌ها کار می‌کنند. به عبارت دیگر هنگام فراخوانی یک سرویس REST، دارید کاری را روی یک منبع انجام می‌دهید.

افزون بر این، REST از دیگر جنبه‌های سیستم‌های HTTP نیز بهره می‌برد؛ مانند موارد زیر:

- قابلیت کَش
- امنیت
- بی‌وضعیتی
- لایه‌بندی شبکه (دیواره‌های آتش و دروازه‌های مختلفی که میان کلاینت و سرور قرار دارد)

در این کتاب تمام اصول و قوانین REST را برای ساخت سرویس‌های MVC با استفاده از ASP.NET MVC توضیح می‌دهیم. اما در صورت تمایل برای آشنایی کامل با معماری REST، می‌توانید به یک کتاب اختصاصی در این‌باره نیز مراجعه کنید. همچنین می‌توانید فصل پنجم پایان‌نامه‌ی دکترای روی فیلدینگ را که ایده‌ی REST نخستین بار توسط او مطرح شد، از آدرس زیر مطالعه کنید:

http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

پیش از آنکه ادامه بدهیم بگذارید پرسشی را که ممکن است اکنون در ذهن برخی افراد شکل گرفته باشد پاسخ بدهیم: آیا سرویس‌های REST را با WCF نیز می‌توان ایجاد کرد؟ پاسخ مثبت است. اگر در اینترنت جستجو کنید می‌توانید مقاله‌هایی پیدا کنید که نقاط قوت و ضعف روش MVC و WCF را برای ساخت سرویس‌های RESTful با یکدیگر مقایسه کرده‌اند. با این وجود از آنجایی که در این کتاب می‌خواهیم نحوه‌ی ساخت سرویس‌های MVC و Web API را توضیح بدهیم، از این بحث صرفنظر می‌کنیم.

[4] resource-centric

انتزاع با استفاده از مسیرها

تا حدی شبیه واسط سرویس‌های WCF، مسیرهای MVC نیز برای برنامه‌نویس یک لایه‌ی انتزاعی فراهم می‌کنند که میان کلاینت و سرویس قرار می‌گیرد. با استفاده از مسیر می‌توانید الگویی از آدرس‌های URL را به کنترلرها و متدها نگاشت کنید. در صورتی که امضای یک API (یک آدرس URL دارای خصلت REST)، به یک واسط، کلاس یا فایل svc. نگاشت نشده باشد به شرط ثابت ماندن آدرس URL و مشخصات درخواست، هر موقع که بخواهید می‌توانید پیاده‌سازی متد API را تغییر بدهید.

یک مثال کلاسیک از به‌کارگیری آدرس URL برای مدیریت تغییرات پیاده‌سازی، نسخه‌بندی سرویس است. برای نمونه می‌توانید مسیری تعریف کنید که در آدرس URL آن کلمه‌ای مانند v2 گنجانده شده باشد. بدین ترتیب می‌توانید بین پیاده‌سازی سرویس و نسخه‌بندی آن به سادگی نگاشت ایجاد کنید. یعنی امکان تعریف نسخه‌هایی را برای سرویس فراهم کنید که هنوز موجود نیست و در آینده اضافه می‌شود. بدین ترتیب می‌توانید برای شروع تعدادی کنترلر (و متد) بنویسید و یک سال بعد تصمیم بگیرید آنها را به‌عنوان نسخه‌ی دوم API سرویس خود قلمداد کنید.

فعال‌سازی کنترلرها به شکل بسیار خوبی انجام می‌شود

صرفنظر از نوع وب‌سرویس -سرویس‌های قدیمی XML (سرویس‌های ASMX)، سرویس‌های WCF یا MVC -، در هر سه حالت مفهومی به نام *فعال‌سازی سرویس*[5] وجود دارد. از آنجایی که به طور کلی در یک سرویس هر فراخوانی، یک درخواست محسوب می‌شود، موتور اجرای سرویس (WCF یا ASP.NET بسته به نوع سرویس)، به ازای هر درخواست از کلاس سرویس یک نمونه‌ی جدید ایجاد می‌کند. به چنین چیزی *فعال‌سازی سرویس* گفته می‌شود. البته فعال‌سازی سرویس چیزی فراتر از نمونه‌سازی ساده از یک کلاس است. در فصل‌های بعد جزئیات این مسئله را به تفصیل توضیح می‌دهیم.

ASP.NET MVC مکانیزمی دارد که توسط آن می‌توانید کلاس‌هایی به نام *اکشن فیلتر*[6] یا به طور خلاصه فیلتر بنویسید که به طور خودکار پیش و پس از اکشن‌ها (متدهای کنترلرها) اجرا شوند. این فیلترها به شکل ویژگی (زیر کلاسی از ActionFilterAttribute) تعریف شده و به متدهای کنترلرها اعمال می‌شوند. همچنین می‌توانید آنها را به شکل عمومی در فایل پیکربندی برنامه تعریف کنید تا به طور خودکار به همه‌ی اکشن‌های تمام کنترلرها اعمال شوند.

شرح این مسئله مفصل است اما به طور کلی مایکروسافت در ASP.NET MVC بستر بسیار ساده و تر و تمیزی فراهم کرده که یکی از نقاط قوتش این است که هیچ یک از اجزای آن مخفی نیست. به طوری که می‌توانید با یک اشکال‌زدا یا debugger، وارد تمام جزئیات یک سرویس (حتی فیلترهای کنترلرها) بشوید و آنها را خط به خط اشکال‌زدایی کنید.

---

[5] service activation
[6] action filter

تعامل JSON، XML و REST

همان‌گونه که گفتیم، REST بر اساس استانداردهای از پیش موجود HTTP طراحی شده است. بنابراین تمام سکوهایی که توانایی ارسال درخواست HTTP داشته باشند می‌توانند از REST استفاده کنند. این مسئله نه تنها کامپیوترها، تلفن‌های هوشمند و تبلت‌ها، بلکه ادواتی مانند تلفن‌های همراه قدیمی، دستگاه‌های DVR، سیستم‌های تلفن، دستگاه‌های ATM، یخچال‌ها، سیستم‌های هشدار، مرورگرها، ساعت‌های دیجیتال و بسیاری چیزهای دیگر را نیز در بر می‌گیرد. در حقیقت هر دستگاهی که بتواند بر اساس یک آدرس URL درخواست HTTP ایجاد کند، می‌تواند از REST نیزاستفاده کند.

استانداردهایی که REST از آنها استفاده می‌کند JSON و XML را نیزدر بر می‌گیرد. این فناوری‌ها از نظر قالب‌بندی یا ساختار پیام‌های مورد مبادله میان کلاینت و سرویس، نیازمندی‌های بسیار کمتری نسبت به SOAP دارند.

البته از نظر فنی پروتکل SOAP خودش بر اساس XML ابداع شده است. با این حال ساخت یک پیام معتبر SOAP (شامل پاکت، هدر، بدنه)، پیچیده‌تر از نمایش ساده‌ی اطلاعات به شکل تگ‌های XML است. این مسئله در مورد پردازش پیام‌های SOAP که بسیار پیچیده و شلوغ هستند در مقایسه با پیام‌های بسیار ساده‌ی XML و JSON نیز صادق است. اما این پیچیدگی به این معنا است که برنامه‌نویسان برای ارسال درخواست SOAP و واکشی داده‌ها از پاسخ SOAP دریافت شده مجبورند از یک کتابخانه‌ی SOAP استفاده کنند. نیاز به چنین کتابخانه‌ای دامنه‌ی ادوات و رسانه‌هایی را که می‌خواهند از وب‌سرویس استفاده کنند محدود می‌کند.

از آن سو، یکی از مزایای JSON صرفنظر از قالب‌بندی بسیار ساده‌ی آن این است که حجم بسته‌های اطلاعاتی JSON در مقایسه با XML و SOAP بسیار کمتر است. لذا JSON افزون بر ادوات بیشتر شبکه‌های تلفن همراه برای استفاده در رسانه‌هایی که ظرفیت باتری پایینی داشته یا همیشه به شبکه وصل نیستند بسیار مناسب است.

البته این حرف‌ها به این معنی نیست که SOAP بی‌ارزش است یا جایگاهی ندارد. در حقیقت قضیه کاملا عکس این است. توانایی‌های SOAP بسیار فراتر از REST و JSON است. بیشتر قابلیت‌های SOAP به شکل مشخصه‌های WS تعریف شده‌اند (WS سرنام web services است)که به نیازهای پیچیده‌تری مانند امنیت، تراکنش، کشف خدمت، انتشار متادیتا، مسیریابی، روابط مطمئن و فدراسیون هویت[7] مربوط می‌شوند. هیچ یک از این نیازها توسط REST قابل پاسخ‌گویی نیست، زیرا برای برآورده کردن آنها به قابلیت‌هایی فراتر از ظرفیت پروتکل HTTP نیاز است.

معرفی مختصر Web API

مزایایی که تا اینجا برای ASP.NET MVC بیان شد، ارتباطی به قابلیت جدید Web API که به MVC 4 اضافه شده ندارد. زیرا فریم‌وُرک MVC به خودی خود - بدون Web API- بستر ساده و قدرتمندی برای ساخت سرویس‌های REST فراهم می‌کند. اما Web API در MVC 4 قابلیت‌های جدیدی به همراه آورده که ایجاد سرویس‌های REST را سریع‌تر و ساده‌تر می‌کند. بگذارید برخی از این قابلیت‌ها را مرور کنیم:

[7] Identity Federation

- **اکشن‌های قراردادی CRUD** [8]: در Web API فعل‌های HTTP (مانند GET و POST) به طور خودکار به متدهای همنام خود در کنترلرها نگاشت می‌شود. برای نمونه، اگر کنترلری به‌نام Products داشته باشید ارسال درخواست GET به api/products/ باعث فراخوانی متدی به‌نام ()Get در کنترلر Products می‌شود. همچنین، Web API به طور خودکار، آرگومان‌هایی را که همراه آدرس URL ارسال شده است، با نسخه‌ی مناسب متد مورد نظر در کلاس کنترلر مطابقت می‌دهد و نسخه‌ی درست متد را فراخوانی می‌کند. برای نمونه، درخواست آدرس api/products/32/ باعث فراخوانی متد Get(long id) می‌شود. این قاعده به طور مشابه در مورد درخواست‌های POST، PUT و DELETE نیز صادق است.
- **مذاکره‌ی محتوا به شکل درون-ساخت**[9]: در MVC اکشن‌هایی که می‌خواهند اطلاعات JSON یا XML را برگردانند باید این کار را به طور صریح انجام بدهند. اما در Web API تنها کافی است خود اطلاعات خام و غیر قالب‌بندی شده را برگردانند. Web API بسته به درخواست دریافت شده، به طور خودکار اطلاعات برگشتی متدها را به JSON یا XML تبدیل می‌کند. از آن سو کسی که می‌خواهد درخواستی ارسال کند نیز با تنظیم یک هدر HTTP به‌نام Content-Type یا Accept مشخص می‌کند مایل است اطلاعات را در چه قالبی دریافت کند. در واقع Web API تضمین می‌کند اطلاعات برگشتی اکشن‌ها به طور خودکار به قالب درست (چیزی که کلاینت مایل است دریافت کند) تبدیل شود. لذا برای نوع برگشتی متدهای کنترلرها به جای JsonResult کافی است نوع خود اطلاعات مورد تحویل را مشخص کنید (مانند Product، IEnumerable<Product> و نظایر آن).
- **پشتیبانی خودکار از OData**: با اضافه کردن ویژگی [Queryable] به متد یک کنترلر که مقدار برگشتی‌اش IQueryable باشد، کلاینت‌ها می‌توانند از پرس و جوهای OData استفاده کنند.
- **خود میزبانی**[10]: با Web API دیگر نیازی نیست برای میزبانی سرویس‌های HTTP از IIS استفاده کنید. بلکه می‌توانید سرویس‌های REST را تحت یک سرویس ویندوز، برنامه‌ی کنسول یا هر نوع میزبان دیگری که بخواهید میزبانی کنید.

خلاصه

در این فصل مشاهده کردید که چگونه فریم‌وُرک ASP.NET MVC سکوی بسیار مفیدی برای ساخت سرویس‌های REST فراهم می‌کند. در سناریوهایی که به تمام قدرت و انعطاف‌پذیری WCF و SOAP نیاز نیست، MVC می‌تواند گزینه‌ی بسیار مناسبی باشد. این سناریوها شامل برنامه‌هایی است که تنها به HTTP نیاز دارند، به انضمام برنامه‌هایی که تمرکز اصلی‌شان بر پیام‌های ساده‌ی متنی است. همچنین در این فصل با مزایای مختلف ASP.NET MVC شامل پشتیبانی از REST، مسیرهای URL و تعامل میان سرویس‌های REST و JSON آشنا شدید. در پایان نیز به طور مختصر با قابلیت جدید Web API که در MVC 4 معرفی شده است آشنا شدید و برخی از قابلیت‌هایی را که Web API به دنیای سرویس‌های REST مبتنی بر ASP.NET وارد می‌کند، مشاهده کردید.

---

[8] Convention-Based CRUD actions
[9] Built-In Content Negotiation
[10] Self-Hosting

فصل 2. چه چیزی RESTful است؟

در این فصل توضیح می‌دهیم سرویسی که قرار است از معماری REST پیروی کند چه شکلی باید داشته باشد. افزون بر اینکه سرویس‌های REST به طور نظری از فعل‌های HTTP استفاده می‌کنند و تمرکزشان بر منابع قرار دارد، API آنها به طور محسوسی با API سرویس‌های RPC متفاوت است. به همین دلیل ابتدا روش REST را با روش سنتی‌تر RPC یا SOAP در خصوص طراحی API یک سرویس مقایسه می‌کنیم.

همان‌گونه که گفتیم در این کتاب نحوه‌ی ساخت یک سرویس REST را برای مدیریت اشیاء وظیفه نشان می‌دهیم. می‌دانم، موضوع این سرویس آنقدرها جذاب نیست. با این وجود جذاب نبودن دامنه‌ی این سرویس باعث می‌شود بهتر بتوانید تمرکز خود را به جنبه‌های فنی سرویس معطوف کنید. البته طراحی یک واسط RESTful بیشتر از آنچه که فکرش را می‌توانید بکنید به دقت نیاز دارد. به طوری که معمولا برای مدل‌سازی API یک سرویس REST، باید از دیدگاه دیگری متفاوت با دیدگاه سنتی به سرویس نگاه کنید. در ادامه با این مسئله آشنا خواهید شد.

البته این واقعیت که طراحی یک سرویس REST به کار بیشتری نیاز دارد نباید باعث شود جا بزنید یا به همان سرویس‌های پیشین خود برگردید. همان طور که در فصل قبل توضیح دادیم، معماری REST مزایای زیادی دارد. اما درک این مزیت‌ها کمی کار می‌برد. بر خلاف تصور اولیه نمی‌توانید با تبدیل متدهای RPC به آدرس‌های URL، واسط یا API سبک REST ایجاد کنید. به جای آن باید طراحی API سبک REST را بر اساس اصول معماری REST انجام بدهید. همچنین در این حالت باید محدودیت‌های پروتکل HTTP را هم در نظر بگیرید؛ زیرا سکوی نهایی شما پروتکل HTTP خواهد بود.

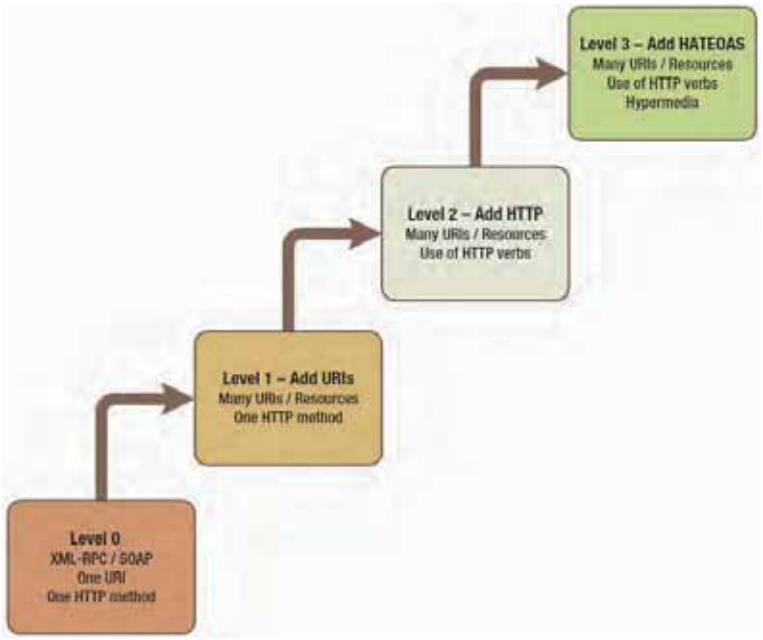
مطالبی که در این فصل یاد می‌گیرید چنین است:

- مدل کمال لئونارد ریچاردسون برای REST
- آدرس‌های URI و منابع
- فعل‌های HTTP
- کُدهای وضعیت HTTP

از RPC تا REST

در ماه نوامبر سال 2008 فردی به نام لئونارد ریچاردسون یک *مدل کمال*[1] برای REST ایجاد کرد. مدل کمال طبق تعریف، نقشه‌ای است که کاربر را با سطوح سلسله مراتبی اعمال یک معماری یا متدولوژی، راهنمایی می‌کند. جمله‌ی سختی بود؟ می‌دانم. ولی تعریف مدل کمال همین است. برای مثال، مدلی مانند CMMI یا Capability Maturity Model Integration به عنوان یک *رهیافت بهبوددهنده‌ی پردازش*[2] برای کمک به سازمان‌ها (معمولا سازمان‌های نرم‌افزاری) ایجاد شد تا کارایی و بازدهی را بهبود بخشد. این مدل از پنج سطح تشکیل می‌شود که در آن هر سطح به‌گونه‌ای طراحی شده که نسبت به سطح پیشین، بازدهی بیشتری در پردازش برای کاربر یا سازمان فراهم کند.

مدل کمالی که ریچاردسون برای REST فراهم کرد و به آن RMM (REST Maturity Model) گفته می‌شود برای کسانی که می‌خواهند API یا سرویس REST بنویسند نقشه‌ی بهبودی فراهم می‌کند که باعث می‌شود API یا سرویس مورد نظر، RESTful شود. این مدل از سطح صفر شروع می‌شود. در این سطح API به سبک RPC تعریف می‌شود. سپس سه سطح دیگر روی آن قرار می‌گیرد تا طراحی API در نهایت به نقطه‌ای برسد که طبق گفته‌ی *روی فیدلینگ*[3] کمترین پیش‌نیاز یک سرویس RESTful محسوب می‌شود. بنابراین اگر طراحی خود را در سطح صفر، 1 یا 2 در مدل RMM متوقف کنید، سرویس شما RESTful نخواهد بود. با این وجود اگر به درستی عمل نکنید حتی در سطح 3 نیز سرویس شما می‌تواند RESTful نباشد. در شکل 1-2 سطوح مختلف مدل RMM نشان داده شده است.



شکل 1-2 نمودار مدل کمال ریچاردسون برای REST

[1] Maturity Model
[2] process-improvement approach
[3] http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

XML-RPC و SOAP

در سطح صفر، API مورد طراحی به API سرویس‌های SOAP شباهت دارد. یعنی تنها بر اساس یک آدرس ساده‌ی URI تعریف می‌شود که تنها از یک فعل یا متد HTTP پشتیبانی می‌کند. در ادامه با فعل‌های مختلف HTTP آشنا خواهید شد. فعلا کافی است بدانید HTTP تنها مجموعه‌ی کوچکی از انواع فعل‌های قابل استفاده را پوشش می‌دهد.

همان طور که در فصل 1 بیان کردیم فرض کنید بخواهیم یک سرویس مدیریت وظیفه ایجاد کنیم. می‌خواهیم امکانی فراهم کنیم تا کلاینت از طریق API سرویس بتواند وظیفه‌ی جدیدی ثبت کند. اگر سرویس ما یک سرویس SOAP سطح صفر می‌بود، می‌توانستیم برای این کار یک کلاس سرویس WCF مثلا به نام TaskService حاوی متدی مانند ()CreateTask تعریف کنیم. پارامترهای این متد می‌تواند چیزی مانند عنوان، دسته‌بندی، وضعیت وظیفه و نظایر آن باشد. این متد یک شیء وظیفه برمی‌گرداند که یکی از مشخصه‌های آن شناسه‌ی وظیفه‌ی ایجاد شده است که مقدار آن بر اساس یک شمارنده‌ی سیستمی تولید می‌شود.

می‌توانیم متدی هم برای بازیابی وظیفه داشته باشیم. بنابراین در کلاس TaskService متدی به نام ()GetTask تعریف می‌کنیم که شناسه‌ی وظیفه‌ی مورد بازیابی را دریافت کرده و آن را بر می‌گرداند که به صورت XML سریال شده و به کلاینت تحویل داده می‌شود. می‌توانیم برای تکمیل کلاس TaskService متدهای دیگری مانند ()UpdateTask، ()SearchTasks و ()CompleteTask نیز تعریف کنیم. هر یک از این متدها یک پیام XML مناسب دریافت می‌کند و پاسخ خود را به شکل یک پیام XML بر می‌گرداند.

در مدل کمال REST - و همچنین اصول روی فیدلینگ- سه ویژگی کلی وجود دارد تا بتوانید API خود را برای استفاده در پروتکل HTTP به شکل یک API سبک REST یا RESTful تعریف کنید. به عبارت دیگر API شما باید سه ویژگی داشته باشد تا RESTful باشد:

- وجود URI یکتا به ازای هر منبع
- استفاده از فعل‌های HTTP
- استفاده از فرا رسانه به عنوان موتور وضعیت برنامه (HATEOAS)[4]

ابتدا بگذارید تعریف واسط کلاس TaskService را در سطح صفر ببینیم (جدول 1-2).

[4] Hypermedia as the engine of application state

جدول 1-2 سرویس مدیریت وظیفه در سطح صفر در مدل RMM

متد	URI	فعل HTTP	تغییر وضعیت/قرارداد
CreateTask	/api/taskservice.svc	POST	کلاینت باید از قبل از آن آگاه باشد یا نحوه‌ی انجام آن را بداند (بر اساس یک سند WSDL)
GetTask	/api/taskservice.svc	POST	کلاینت باید از قبل از آن آگاه باشد یا نحوه‌ی انجام آن را بداند (بر اساس یک سند WSDL)
GetTaskAssignees	/api/taskservice.svc	POST	کلاینت باید از قبل از آن آگاه باشد یا نحوه‌ی انجام آن را بداند (بر اساس یک سند WSDL)
SearchTasks	/api/taskservice.svc	POST	کلاینت باید از قبل از آن آگاه باشد یا نحوه‌ی انجام آن را بداند (بر اساس یک سند WSDL)
UpdateTask	/api/taskservice.svc	POST	کلاینت باید از قبل از آن آگاه باشد یا نحوه‌ی انجام آن را بداند (بر اساس یک سند WSDL)
CompleteTask	/api/taskservice.svc	POST	کلاینت باید از قبل از آن آگاه باشد یا نحوه‌ی انجام آن را بداند (بر اساس یک سند WSDL)

همان‌گونه که می‌بینید همه‌ی متدهای کلاس سرویس فعلی -از دیدگاه وب- مانند یکدیگر هستند (همگی یک آدرس URI دارند و با یک فعل HTTP کار می‌کنند). در واقع این سرویس، خیلی وبی به نظر نمی‌رسد. به عنوان مثال آدرس لازم برای بازیابی همه‌ی وظیفه‌ها (وظیفه‌ای با شناسه‌ی 123 یا 456 یا هر وظیفه‌ی دیگر) یکسان است. حتی این آدرس URI برای ایجاد یک وظیفه‌ی جدید، به‌روزرسانی یک وظیفه، تکمیل یک وظیفه و نظایر آن نیز یکسان است. در حقیقت هیچ احساس آدرس‌دهی هر منبع به صورت یکتا بر اساس URI وجود ندارد. هیچ آدرسی وجود ندارد که به طور مستقیم به یک وظیفه‌ی مشخص یا فهرستی از وظیفه‌ها ارجاع کند.

همچنین این سرویس از فعل‌های HTTP آن‌گونه که انتظار می‌رود بهره نمی‌برد. در فصل 1 کمی در این خصوص توضیح دادیم. در ادامه نیز آن را به طور مفصل‌تر بیان می‌کنیم. در این سرویس برای اکشن‌های API از اسامی شخصی استفاده کرده‌ایم. این در حالی است که برای اینکه سرویس شما یک سرویس RESTful باشد باید از تعریف اکشن‌های شخصی خودداری کنید. به جای آن باید اکشن‌هایی بنویسید که با HTTP سازگاری داشته باشد. به عبارت دقیق‌تر باید از GET، POST، PUT و DELETE استفاده کنید.

نقطه ضعف دیگر روش SOAP این است که کلاینت‌ها باید از قبل، از تمام اکشن‌هایی که برای استفاده مهیا شده آگاه باشند تا بتوانند از آنها استفاده کنند. یعنی یک جور وابستگی و اتصال ضمنی بین کلاینت و سرور وجود دارد که در آن کلاینت به یک قرارداد از پیش نوشته شده و مجموعه‌ای از اکشن‌های تعریف شده وابسته است. بار دیگر این مسئله چندان ماهیت وبی ندارد.

از آن سو در دنیای وب هنگامی که وارد یک سایت وب می‌شوید، تنها چیزی که لازم دارید بدانید آدرس ریشه‌ی سایت است. زیرا وقتی پاسخی را دریافت می‌کنید تمام چیزهای بعدی (آدرس‌های قابل استفاده‌ی بعدی) به طور

خودکار از پاسخ دریافت شده قابل کشف است و از طریق *فرارسانه*[5] (لینک‌ها یا فرم‌های وب) به دیگر المان‌ها یا صفحات دیگر زنجیر شده است. بنابراین نیازی نیست آدرس‌های صفحات بعد را حفظ کنید تا بتوانید کاری انجام بدهید. در واقع فرارسانه، موتور تغییر وضعیت برنامه است. به طوری که می‌توانید تنها با در دست داشتن یک لینک یا ارسال یک فرم وب، وضعیت برنامه‌ی وب را تغییر بدهید (در اینجا ماشین وضعیت، خود سایت وب یا در مقیاس بزرگ‌تر، خود اینترنت است).

همچنین در سایت‌های وب مجبور نیستید فیلدهای فرم‌های وب را (مثلا برای ثبت سفارش یا تکمیل فرم اشتراک یک مجله یا خبرنامه) به خاطر بسپارید. زیرا سرور خودش تمام روابط، فرم‌ها و آدرس‌های لینک‌های صفحات را مشخص می‌کند، بدون آنکه مجبور باشید چیزی را از قبل بدانید. تنها چیزی که نیاز دارید کلاینتی است که بتواند بر اساس یک آدرس URI، درخواست وب ارسال کند. حتی اگر لینک‌ها یا فرم‌های صفحه هم تغییر کند، به احتمال زیاد به هیچ وجه متوجه نخواهید شد یا برایتان اهمیتی نخواهد داشت.

از این رو از قبل نوعی درک ضمنی نسبت به سایت‌های وب وجود دارد که سایت‌ها خودشان ما را به سمت منابع مختلف هدایت کرده و همه‌ی اطلاعاتی را که برای اعمال هر تغییر یا ارسال هر درخواست نیاز داریم، فراهم می‌کنند. بنابراین نیازی نیست ما چیزی را حفظ کنیم.

به این ویژگی، HATEOS گفته می‌شود. همان‌گونه که در ادامه خواهید دید، ویژگی HATEOAS یکی از ویژگی‌های کلیدی سرویس‌های RESTful است. با این حال بیشتر افراد از آن چشم‌پوشی می‌کنند، زیرا مستلزم تغییر نگرش چشم‌گیری در طراحی API سرویس، در روش سنتی به سبک RPC است.

URI و منابع

همان‌گونه که در فصل 1 بیان کردیم، هنگام ایجاد یک API یا واسط RESTful، در نهایت باید به واسطی برسید که منبع-محور[6] باشد. زیرا محور و نقطه‌ی مرکزی طراحی یک واسط RESTful را *منابع* تشکیل می‌دهند. بر خلاف روش طراحی به سبک RPC که در آن متدهای سرویس (فعل‌هایی مانند CreateTask، UpdateTask و نظایر آن)، از قبل مشخص می‌کنند درخواست و پاسخ و دیگر چیزها چه ساختاری باید داشته باشند، طراحی یک واسط REST حول منابع (اسامی یا نام‌هایی مانند task، product و نظایر آن) می‌گردد. اکشن‌های قابل اعمال روی این منابع نیز به حوزه‌ی استفاده محدود می‌شود که حوزه‌ی استفاده در اینجا پروتکل HTTP است. به همین دلیل است که باید API خود را به فعل‌های HTTP نگاشت کنید و اجازه ندارید اکشن‌ها یا فعل‌های جدیدی تعریف کنید.

این، مفهوم مرکزی معماری REST است. اما بگذارید ببینیم اگر کلاس TaskService بخواهد با سطح 1 مدل RMM مطابقت داشته باشد چه شکلی باید داشته باشد. همان‌گونه که در جدول 2-2 نشان داده‌ایم در این حالت هر منبع بر اساس یک آدرس URI به طور یکتا قابل آدرس‌دهی است.

---

[5] hypermedia
[6] resource-centric

جدول 2-2 سرویس مدیریت وظیفه در سطح 1 در مدل RMM

متد	URI	فعل HTTP	تغییر وضعیت/قرارداد
CreateTask	/api/tasks	POST	کلاینت باید از قبل از آن آگاه باشد یا نحوه‌ی انجام آن را بداند (بر اساس یک سند WSDL)
GetTask	/api/tasks/1234	POST	کلاینت باید از قبل از آن آگاه باشد یا نحوه‌ی انجام آن را بداند (بر اساس یک سند WSDL)
GetTaskAssignees	/api/tasks/1234	POST	کلاینت باید از قبل از آن آگاه باشد یا نحوه‌ی انجام آن را بداند (بر اساس یک سند WSDL)
SearchTasks	/api/tasks	POST	کلاینت باید از قبل از آن آگاه باشد یا نحوه‌ی انجام آن را بداند (بر اساس یک سند WSDL)
UpdateTask	/api/tasks/1234	POST	کلاینت باید از قبل از آن آگاه باشد یا نحوه‌ی انجام آن را بداند (بر اساس یک سند WSDL)
CompleteTask	/api/tasks/1234	POST	کلاینت باید از قبل از آن آگاه باشد یا نحوه‌ی انجام آن را بداند (بر اساس یک سند WSDL)

با این وجود کلاینت برای اجرای هر عملیات، هنوز به پیام‌های مشخصی وابسته است. به عبارت دیگر با در دست داشتن یک آدرس URI ثابت مانند api/tasks/ کلاینت نمی‌تواند میان اکشن‌های مختلف مربوط به آن تفاوت قائل شود - مگر آنکه پیش‌تر، از مستندات و قرارداد API سرویس در این زمینه آگاه باشد. همچنین API فعلی تنها از فعل POST استفاده کرده و خود پیام ارسال شده باید اکشن مورد تقاضا را مشخص کند.

فعل‌های HTTP

برای طراحی اکشن‌های یک سرویس باید به چیزی ورای آدرس‌های URI و منابع توجه داشته باشید. در معماری REST متدی که باید توسط سرویس اجرا شود بر اساس فعل HTTP (مانند GET، POST و نظایر آن) که درخواست طی آن ارسال شده مشخص می‌شود. این در حالی است که طبق سرویسی که تا اینجا نوشته‌ایم در پروتکل HTTP فعلی به نام CreateTask وجود ندارد. حتی فعلی به نام Create هم وجود ندارد. بنابراین سرویس ما هنوز RESTful نیست. اگر قرار است از معماری REST و پروتکل HTTP پیروی کنیم، باید متدهای سرویس خود را از میان فعل‌های پروتکل HTTP انتخاب کنیم. این فعل‌ها عبارتند از:

- GET
- PUT
- POST
- DELETE

همان‌گونه که از عنوان فعل‌ها بر می‌آید برای اکشنی مانند CreateTask فعل‌های GET و DELETE انتخاب مناسبی نیستند (زیرا نه در حال بازیابی و نه در حال حذف چیزی هستیم). بنابراین باید از فعل PUT یا POST استفاده کنیم. اما PUT و POST چه تفاوتی با هم دارند؟ همان‌گونه که در جدول 3-2 نشان داده شده فعل PUT برای ایجاد یک منبع جدید یا به‌روزرسانی یک منبع از قبل موجود (بر اساس یک شناسه‌ی مشخص و در نتیجه یک آدرس URI

مشخص) به‌کار می‌رود. فعل POST نیز زمانی به‌کار می‌رود که شناسه‌ی منبع را سیستم قرار است تولید کند (بنابراین آدرس URI منبع از قبل مشخص نیست).

جدول 3-2 استفاده از فعل‌های HTTP برای اکشن‌های وظیفه‌ها

فعل وب	آدرس کلکسیونی http://myserver.com/tasks	آدرس المانی http://myserver.com/tasks/1234
GET	بازیابی لیستی از وظیفه‌ها	بازیابی یک وظیفه‌ی تکی بر اساس آدرس URI فعلی
PUT	جایگزین کردن لیست تمام وظیفه‌ها	ایجاد یا به‌روزرسانی وظیفه‌ای که بر اساس آدرس URI فعلی مشخص شده است
POST	ایجاد یک وظیفه‌ی تکی جدید که شناسه‌ی آن توسط سیستم تولید خواهد شد	ایجاد یک زیر وظیفه تحت وظیفه‌ای که توسط آدرس URI فعلی مشخص شده است
DELETE	حذف تمام وظیفه‌ها	حذف وظیفه‌ای که بر اساس آدرس URI فعلی مشخص شده است

در حالت کلی معماری REST به پروتکل ویژه‌ای وابسته نیست. این مسئله پروتکل HTTP را نیز در بر می‌گیرد. به عبارت دیگر تنها چیزی که در REST لازم دارید پروتکلی است که برای توصیف وضعیت‌ها (یا نمایش‌ها[7]) و تغییر وضعیت‌ها، زبان و مکانیزم مشخصی داشته باشد. از آنجایی که در این کتاب نحوه‌ی ایجاد سرویس REST را در ASP.NET توضیح می‌دهیم، تمرکزمان به REST در HTTP است.

خوشبختانه پروتکل HTTP خودش بیشتر چیزهایی را که نیاز داریم پوشش می‌دهد. بار دیگر یادآوری می‌کنیم که جدول 3-2 هدفی را که در REST از هر یک از فعل‌های HTTP انتظار می‌رود، نشان می‌دهد.

بگذارید برخی از نکات مهم نگاشت فعل‌های HTTP به اکشن‌های مورد انتظار در REST را هم بررسی کنیم. نکته‌ی نخست این است که معنی دقیق هر یک از فعل‌های چهارگانه‌ی HTTP به آدرس URI وابسته است. یعنی یک فعل HTTP به تنهایی معنی و مفهوم خود را مشخص نمی‌کند و باید مشخص شود این فعل به همراه چه نوع آدرسی استفاده می‌شود.

همان‌گونه که در جدول 3-2 نشان داده‌ایم به طور کلی دو نوع آدرس URI وجود دارد: آدرس لیستی یا کلکسیونی (که فاقد شناسه‌ای برای یک منبع تکی است) و آدرس المانی (که در آن شناسه‌ی یک منبع تکی به طور ویژه مشخص می‌شود). در آدرس‌های کلکسیونی با مجموعه‌ای از منابع سروکار داریم، اما در آدرس‌های المانی تنها با یک منبع کار می‌کنیم. با توجه به این مسئله، با وجودی که تنها چهار فعل HTTP داریم، در عمل به ازای هر منبع (مانند وظیفه یا Task)، هشت اکشن مختلف می‌توانیم داشته باشیم که می‌توانیم در طراحی API سرویس REST از آنها استفاده کنیم. تفاوت اکشن‌ها نیز بر اساس آدرس URI مشخص می‌شود که آیا با مجموعه‌ای از منابع یا یک منبع تکی قرار است کار کنند.

[7] representations

نکته‌ی دوم این است که هنگام ایجاد نمونه‌ی جدیدی از یک منبع (مانند یک وظیفه)، اگر می‌خواهیم کلاینت خودش شناسه‌ی منبع جدید را مشخص کند، باید از PUT استفاده کنیم. اما اگر می‌خواهیم شناسه‌ی منبع جدید را سیستم تولید کند، باید از POST و یک آدرس URI کلکسیونی استفاده کنیم. این مسئله به خاصیت «خودنمایی»[8] در ریاضیات جبری مربوط می‌شود.

در REST انتظار می‌رود متدهای PUT و DELETE خاصیت خودنمایی داشته باشند. این یعنی فراخوانی دوباره‌ی آنها هر بار باید یک نتیجه را به دست بدهد؛ بدون آنکه اثر جانبی دیگری تولید شود. برای نمونه کلاینت باید بتواند بدون وقوع خطا یا اثر جانبی هر چند بار که بخواهد اکشن DELETE را برای یک منبع تکی فراخوانی کند. در صورتی که منبع درخواست شده از قبل حذف شده است، کلاینت نباید پیغام خطا دریافت کند. این مسئله در مورد فعل PUT نیز صادق است. به ازای یک آدرس المانی، اگر منبع مورد نظر از قبل وجود ندارد، سیستم باید آن را مطابق درخواست ارسال شده ایجاد کند. اما اگر منبع از قبل وجود دارد ارسال دوباره‌ی درخواست به شکل PUT باید باعث به‌روزرسانی آن شود. بدین ترتیب فراخوانی پی‌درپی PUT همیشه نتیجه‌ی یکسانی تولید خواهد کرد و این همان چیزی است که در REST از PUT انتظار می‌رود.

در معماری REST اکشن GET یک اکشن ایمن (Safe) قلمداد می‌شود، اما الزاما خاصیت خودنمایی ندارد (مگر آنکه خودمان چنین چیزی را قرارداد کنیم). مقصود از ایمنی نیز یعنی اکشن GET نباید چیزی را دستکاری کرده یا چیزی را در سیستم تغییر بدهد. بلکه تنها باید برای خواندن به‌کار برود. این اکشن برای بازیابی مجموعه‌ای از داده‌ها یا یک فقره داده‌ی تکی مناسب است.

نکته‌ی مهمی که وجود دارد این است که خاصیت خودنمایی عملیات GET، PUT و DELETE در یک سرویس REST باید با استانداردهای پروتکل HTTP نیز هماهنگی داشته باشد. بنابراین باید حداکثر تلاش‌تان را بکنید که فراخوانی پی‌درپی این اکشن‌ها برای API سرویس شما اثر جانبی یا خطا نداشته باشد و مجاز قلمداد شود.

اما بر خلاف سه اکشن پیشین، اکشن POST خاصیت خودنمایی ندارد. زیرا اکشن POST برای ایجاد منبع جدیدی به کار می‌رود که شناسه‌اش را سیستم باید تولید کند. بنابراین فراخوانی پی‌درپی آن هر بار باعث ایجاد یک منبع جدید خواهد شد. این در حالی است که فراخوانی تکراری اکشن PUT حداکثر یک منبع جدید تولید می‌کند و در فراخوانی‌های بعدی، همان منبع با همان مشخصات به‌روز می‌شود (که منبع مورد نظر در عمل تغییر نخواهد کرد). اکشن POST زمانی مناسب است که کلاینت خودش نمی‌تواند شناسه‌ی منبع را مشخص کند. اما اکشن PUT زمانی مناسب است که شناسه‌ی منبع را کلاینت مشخص می‌کند.

هنگام مدل‌سازی یک سرویس باید هر منبع را به تعدادی اکشن یا فعل HTTP نگاشت کنید و مشخص کنید کدام اکشن‌ها از نظر شما بر روی منابع مختلف مجاز بوده و API شما از کدام اکشن‌ها پشتیبانی نمی‌کند. بگذارید این را در سرویس مدیریت وظیفه ببینیم. در این مرحله در طراحی API سرویس مدیریت وظیفه از فعل‌های HTTP نیز استفاده می‌کنیم. بدین ترتیب سرویس ما به سطح 2 در مدل RMM می‌رسد (جدول 4-2).

---

[8] idempotency

جدول 4-2 سرویس مدیریت وظیفه در سطح 2 در مدل RMM

متد	URI	فعل HTTP	تغییر وضعیت/قرارداد
CreateTask	/api/tasks	POST	کلاینت باید از قبل از آن آگاه باشد یا نحوه‌ی انجام آن را بداند (بر اساس یک سند WSDL)
GetTask	/api/tasks/1234	GET	کلاینت باید از قبل از آن آگاه باشد یا نحوه‌ی انجام آن را بداند (بر اساس یک سند WSDL)
GetTaskAssignees	/api/tasks/1234	GET	کلاینت باید از قبل از آن آگاه باشد یا نحوه‌ی انجام آن را بداند (بر اساس یک سند WSDL)
SearchTasks	/api/tasks	GET	کلاینت باید از قبل از آن آگاه باشد یا نحوه‌ی انجام آن را بداند (بر اساس یک سند WSDL)
UpdateTask	/api/tasks/1234	PUT	کلاینت باید از قبل از آن آگاه باشد یا نحوه‌ی انجام آن را بداند (بر اساس یک سند WSDL)
CompleteTask	/api/tasks/1234	DELETE	کلاینت باید از قبل از آن آگاه باشد یا نحوه‌ی انجام آن را بداند (بر اساس یک سند WSDL)

همان‌گونه که در سطح 1 دیدید، سرویس مدیریت وظیفه برای ارجاع به منابع، از آدرس‌های URI مجزا و یکتا استفاده کرد. در اینجا در سطح 2 نیز کاری کرده‌ایم به‌جای ارسال درخواست در قالب پیام‌هایی با یک ساختار شخصی (مانند پیام‌های XML در SOAP) به طور ساده از فعل‌های HTTP استفاده شود. اکشن‌های PUT و POST در جدول 3-2 که برای ایجاد یا به‌روزرسانی منابع به‌کار می‌روند به همراه خود، شکلی از منبع را (مثلا به شکل JSON یا XML) حمل خواهد کرد (نیازی هم به تعریف یک ساختار شخصی برای درخواست و پاسخ نیست).

با این وجود کلاینت برای پیمایش دامنه و انجام عملیات پیچیده‌تر نسبت به ایجاد، به‌روزرسانی یا تکمیل یک وظیفه هنوز باید از پیش، از API آگاه باشد. در حالی که در دنیای وب، این ما هستیم که به عنوان طراح یک سایت کلاینت را هدایت و راهنمایی کرده و منابع قابل استفاده و اکشن‌های قابل اجرا روی آنها را از طریق لینک‌ها و فرم‌های وب، برای او فراهم می‌کنیم. این چیزی است که از طریق HATEOAS (فرا رسانه به‌عنوان موتور وضعیت برنامه) در سطح 3 در مدل RMM انجام می‌شود.

HATEOAS

اگر به جدول 3-2 و 4-2 نگاه کنید، خواهید دید برخی عملیات GET باعث برگرداندن کلکسیونی از منابع می‌شود. یکی از راهبردهای REST در HTTP این است که کلاینت‌ها تنها از طریق فرارسانه‌هایی که خود سرور فراهم می‌کند باید بتوانند وضعیت برنامه را تغییر بدهند. به عبارت دیگر با در دست داشتن تنها یک URI ریشه‌ای و بدون هیچ دانشی درباره‌ی اسکیمای URI، کلاینت باید بتواند کلکسیون منابع را پیمایش کرده و از طریق لینک‌هایی که API در پاسخ بر می‌گرداند باید بتواند آنها را تغییر بدهد. بنابراین وقتی سرویس REST منبعی را بر می‌گرداند (به صورت تکی یا کلکسیونی)، اطلاعات برگشتی باید حاوی آدرسی باشد که مشخص کند چگونه می‌توان این منبع را دوباره با یک اکشن GET دیگر به‌طور یکتا بازیابی کرد.

در زیر مثالی از پاسخ XML یک درخواست REST را نشان داده‌ایم که مشخص می‌کند چگونه هر یک از منابع کلکسیون برگردانده شده باید یک آدرس URI داشته باشند.

```
<?xml version="1.0" encoding="utf-8"?>
<Tasks>
  <Task Id="1234" Status="Active" >
    <link rel="self" href="/api/tasks/1234" method="GET" />
  </Task>
  <Task Id="0987" Status="Completed" >
    <link rel="self" href="/api/tasks/0987" method="GET" />
  </Task>
</Tasks>
```

همان‌گونه که می‌بینید هنگام برگرداندن کلکسیونی از منابع می‌توانید تنها برخی از خصوصیت‌های منابع را برگردانید. کلاینت پس از دریافت پاسخ می‌تواند با استفاده از آدرس URI هر منبع، درخواست دیگری برای دریافت مشخصات کامل آن ارسال کند. به‌عنوان مثال در کلکسیون Tasks در مثال بالا به ازای هر وظیفه تنها شناسه‌ی وظیفه و یک آدرس URI وجود دارد که از طریق آدرس‌های URI می‌توان هر وظیفه را بازیابی کرد. پس از ارسال درخواست GET برای بازیابی یک وظیفه‌ی تکی، پاسخ برگردانده شده می‌تواند مشخصات کامل وظیفه مانند عنوان، دسته‌بندی، تاریخ ایجاد، وضعیت، مالک و نظایر آن را تحویل بدهد.

اما اجرای این رهیافت کمی به ترفند نیاز دارد. زیرا مجبورید به ازای هر منبع، دست‌کم دو گونه‌ی مختلف از کلاس یا نوع داده‌ی آن را داشته باشید. روش مرسوم این است که به ازای هر منبع مانند Task، باید یک کلاس TaskInfo و یک کلاس Task تعریف کنید. کلاس TaskInfo تنها برای فراهم کردن مشخصات پایه‌ای وظیفه و کلاس Task برای برگرداندن مشخصات کامل وظیفه به‌کار می‌رود. بر این اساس، کلکسیون وظیفه‌های برگردانده شده می‌تواند چنین چیزی باشد:

```
<?xml version="1.0" encoding="utf-8"?>
<Tasks>
  <TaskInfo Id="1234" Status="Active" >
    <link rel="self" href="/api/tasks/1234" method="GET" />
  </TaskInfo>
  <TaskInfo Id="0987" Status="Completed" >
    <link rel="self" href="/api/tasks/0987" method="GET" />
  </TaskInfo>
</Tasks>
```

مشخصات کامل یک منبع منفرد نیز می‌تواند چنین باشد:

```
<?xml version="1.0" encoding="utf-8"?>
<Task Id="1234" Status="Active" DateCreated="2011-08-15" Owner="Sally" Category="Projects" >
  <link rel="self" href="/api/tasks/1234" method="GET" />
</Task>
```

البته در REST تعریف دو نوع داده‌ی مختلف به ازای هر منبع، الزامی نیست. ممکن است به این نتیجه برسید که نیازی نیست تعریف کلکسیون‌ها را جدا کنید. همچنین ممکن است به این نتیجه برسید که به ازای برخی از منابع به بیش از دو نوع داده نیاز دارید. تمام اینها به سناریوی شما و تعداد خصوصیت‌های منبع بستگی دارد. به‌عنوان نمونه اگر یک وظیفه تنها پنج یا شش خصوصیت داشته باشد، معمولا دو نوع داده کافی است. اما اگر وظیفه دارای 100