

بخش هایی از کتاب "مرجع کامل ASP.NET MVC (با پوشش کامل ASP.NET MVC 4)"

ترجمه و تالیف: بهروز راد

وضعیت: در دست چاپ

Web API چیست؟

Web API، نوع قالب جدیدی برای پروژه‌های مبتنی بر وب در .NET است که بر مبنای اصول و الگوهای موجود در ASP.NET MVC ساخته شده است و همراه با ASP.NET MVC 4 وجود دارد. Web API توسعه گران را قادر می‌سازد تا با استفاده از یک الگوی ساده که در Controllerها پیاده سازی می‌شود، وب سرویس‌های مبتنی بر پروتوکل HTTP را با کدها و تنظیمات کم ایجاد کنند. این سبک جدید برای ایجاد وب سرویس‌ها، می‌تواند در انواع پروژه‌های .NET مانند ASP.NET MVC، ASP.NET Web Forms، Windows Application و ... استفاده شود.

یک سوال کاملاً منطقی در اینجا به وجود می‌آید. چرا نیاز به بستری جدید برای ایجاد وب سرویس داریم؟ آیا در حال حاضر مایکروسافت بستری محبوب و فراگیر برای توسعه‌ی وب سرویس‌هایی که بتوانند با پروتوکل SOAP تعامل داشته باشند در اختیار ندارد؟ مگر وب سرویس‌های ASMX از زمان معرفی ASP.NET وجود نداشته‌اند؟ آیا تکنولوژی WCF مایکروسافت، بیشترین انعطاف پذیری و قدرت را برای تولید وب سرویس‌ها در اختیار قرار نمی‌دهد؟ وب سرویس‌ها جایگاه خود را یافته‌اند و توسعه گران با تکنولوژی‌های موجود به خوبی آنها را پیاده سازی و درک می‌کنند. چرا Web API؟

چرا Web Api؟

برای پاسخ به این سوال، باید برخی مشکلات را بررسی کنیم و ببینیم ابزارهای موجود چه راه حلی برای آنها در نظر گرفته‌اند. اگر با گزینه‌هایی که در ادامه می‌آیند موافق هستید، خواندن این مطلب را ادامه دهید، و اگر اعتقادی به آنها ندارید، پس نیازهای شما به خوبی با بسترهای موجود پاسخ داده می‌شوند. من معتقد هستم که راه بهتری برای ایجاد وب سرویس‌ها وجود دارد. من معتقد هستم که روش‌های ساده‌تری برای ایجاد وب سرویس‌ها وجود دارد و WCF بیش از حد پیچیده است. من معتقد هستم که تکنولوژی‌های پایه‌ی وب مانند GET، POST، PUT و DELETE برای انجام اعمال مختلف توسط وب سرویس‌ها کافی هستند. اگر همچنان در حال خواندن این مطلب هستید، توضیحات خود را با شرح تفاوت میان Web API و تکنولوژی‌های دیگر هم حوزه‌ی آن ادامه می‌دهیم و خواهید دید که استفاده از Web API چقدر آسان است.

تفاوت Web API و WCF

وب سرویس‌های ASMX تا چندین سال، انتخاب اول برای ایجاد وب سرویس‌های مبتنی بر پروتوکل SOAP با استفاده از پروتوکل HTTP بودند. وب سرویس‌های ASMX، از وب سرویس‌های ساده که نیاز به قابلیت تعامل پایین داشتند و در نتیجه به پروتوکل SOAP نیز وابسته نبودند پشتیبانی نمی‌کردند. WCF جای وب سرویس‌های ASMX را گرفت و خود را به عنوان آخرین و بهترین روش برای ایجاد وب سرویس‌ها در بستر .NET معرفی کرد. نمونه‌ای از یک سرویس WCF بر مبنای پروتوکل HTTP در .NET. به صورت ذیل است.

```
[ServiceContract]
public interface IService1
{
    [OperationContract]
    string GetData(int value);
    [OperationContract]
    CompositeType GetDataUsingDataContract(CompositeType composite);
}
...
public class Service1 : IService1
{
    public string GetData(int value)
    {
        return string.Format("You entered: {0}", value);
    }
}
```

```

    }
    public CompositeType GetDataUsingDataContract(CompositeType composite)
    {
        if (composite == null)
        {
            throw new ArgumentNullException("composite");
        }
        if (composite.BoolValue)
        {
            composite.StringValue += "Suffix";
        }
        return composite;
    }
}

```

در WCF، پایه و اساس وب سرویس را یک interface تشکیل می‌دهد. در حقیقت اجزای وب سرویس را باید در یک interface تعریف کرد. هر یک از متدهای وب سرویس در interface تعریف شده که صفت OperationContract برای آنها در نظر گرفته شده باشد، به عنوان یکی از اعمال و متدهای قابل فراخوانی توسط استفاده کننده از وب سرویس در دسترس هستند. سپس کلاسی باید ایجاد کرد که interface ایجاد شده را پیاده سازی می‌کند. در قسمت بعد، با مفاهیم پایه‌ی Web API و برخی کاربردهای آن در محیط ASP.NET MVC آشنا می‌شوید.

نتیجه گیری

Web API، یک روش جدید و آسان برای ایجاد وب سرویس‌ها، بر مبنای مفاهیم آشنای ASP.NET MVC و پایه‌ی وب است. از این روش می‌توان در انواع پروژه‌های NET استفاده کرد.

نظرات خوانندگان

نویسنده: یوسف نژاد
تاریخ: ۱۱:۵۹ ۱۳۹۱/۰۴/۱۱

این مقوله خیلی مفیده و کاربردی هست. خیلی وقت بود میخواستم در موردش بیشتر تحقیق کنم. با تشکر بابت زحماتتون و آغاز این سری مطلب جدید.

نویسنده: شهرز جعفری
تاریخ: ۱۶:۷ ۱۳۹۱/۰۴/۱۱

زمان انتشار کی هس؟

نویسنده: بهروز راد
تاریخ: ۱۹:۵ ۱۳۹۱/۰۴/۱۱

بستگی به ناشر داره. اما نباید بیشتر از دو هفته طول بکشه.

نویسنده: شهرز جعفری
تاریخ: ۱۹:۸ ۱۳۹۱/۰۴/۱۱

نظر کلی من اینه که همیشه به wcf گفت پیچیده آخه هدفش فرق میکنه. و در ضمن مگه API حدوداً همون Rest با معماری خیلی ساده تر نیست؟

نویسنده: شهرز جعفری
تاریخ: ۱۹:۸ ۱۳۹۱/۰۴/۱۱

بیصبرانه منتظرش هستم

نویسنده: بهروز راد
تاریخ: ۲۲:۴۷ ۱۳۹۱/۰۴/۱۱

REST بیشتر برای مواقعی هست که شما عملیات CRUD انجام میدید. در حالی که با Web API میتونید علاوه بر CRUD، کارهای بسیار بیشتری انجام بدید. مفاهیم و قابلیت های موجود در ASP.NET MVC مانند فیلترها به خوبی در Web API پشتیبانی و به راحتی قابل استفاده هستند. ضمن اینکه با Web API میتونید معماری REST رو با تغییر کوچکی در route پیش فرض به دست بیارید و بدین شکل، مهاجرت از REST به Web API بسیار راحت هست. در اوایل معرفی Web API، از پروتوکول OData نیز پشتیبانی اولیه میشد که متأسفانه مایکروسافت در نسخه ی RC این پشتیبانی رو حذف کرد. شاید در نسخه های بعدی این قابلیت نیز اضافه بشه که به قدرتمندتر شدن Web API کمک می کنه.

ضمناً، پشتیبانی مایکروسافت از WCF REST API نیز به اتمام رسیده و پیشنهاد شده که از Web API استفاده کنید.

<http://aspnet.codeplex.com/wikipage?title=WCF%20REST>

نویسنده: رضا.ب
تاریخ: ۳:۳ ۱۳۹۱/۰۴/۱۲

در بند سوم اشاره کردین : من معتقد هستم که تکنولوژی های پایه ی وب مانند آفعال GET، POST، PUT و DELETE برای انجام اعمال مختلف توسط وب سرویس ها کافی هستند. اگر ضروری نیستند بیشتر از CRUD باشند پس خاصیت ویژه ای که شما میگین "کارهای بسیار بیشتری" میتونه انجام بده چی هست که WCF پاسخگو نیست؟

در ضمن فکر میکنم REST فقط با منابع و ور ب های HTTP کار داره. و برای همین سهولت و سادگیش پروتکول SOAP نسخه منسوخ

شده‌ی وب‌سرورها به حساب میاد. اینطور نیست؟

سوال دیگه‌ام در مورد میزان نقش Web API هست. آیا رسالت واقعی یک وب‌سرویس رو هدف گرفته؟ یعنی پیاده سازی یک Endpoint که شامل یه سری interface هستند که امکاناتی رو در اختیار کلاینت قرار میده؟ ممنون از توجه‌تون.

نویسنده: بهروز راد
تاریخ: ۱۳۹۱/۰۴/۱۲ ۸:۳۷

دقت داشته باشید که Web API عرضه نشده تا WCF رو منسوخ کنه. برنامه‌هایی که صرفاً از بستر پروتوکل HTTP به عنوان یک سرویس برای رد و بدل کردن داده‌ها استفاده می‌کنند، بهتره که از این به بعد از Web API استفاده کنند. ضمن سادگی و مفاهیم آشنای ASP.NET MVC، روش یکپارچه‌ای برای ایجاد وب سرویس‌های HTTP نیز به وجود اومده که مشکلات استفاده از WCF رو از بین می‌بره. WCF ذاتاً برای پیغام‌های SOAP محور طراحی شده و به کار گرفتن اون برای وب سرویس‌های HTTP یا به زور خوراندن HTTP به اون بی معنیه. در WCF راه‌های مختلفی برای ایجاد وب سرویس‌های HTTP وجود داره که باعث گمراهی و سردرگمی توسعه گر میشه و حتی فریمورک‌های مختلفی مانند OpenRasta و ServiceStack نیز بدین منظور وجود دارند. بنابراین پشتیبانی WCF از HTTP به یک پروژه‌ی دیگه تحت نام ASP.NET Web API منتقل شده و WCF Web API دیگه پشتیبانی نمیشه. کمی تغییر نام و کمی جابجایی مفاهیم در اینجا صورت گرفته. WCF همچنان قدرتمنده و نباید Web API به هیچ وجه به عنوان جایگزینی برای اون تصور بشه. ایجاد بسترهایی برای ارتباطات دو طرفه یا صفی از پیغام‌ها یا سوئیچ بین کانال‌ها در هنگام فعال نبودن یک کانال، اینها همه از قابلیت‌هایی هست که Web API هرگز جایگزینی برای اونها نخواهد بود و مختص WCF هستند.

نویسنده: ramtin
تاریخ: ۱۳۹۱/۰۴/۱۲ ۹:۲۶

سلام آقای راد
بخشید که سوال بی ربط رو اینجا میپرسم
آیا برنامه‌ای برای انتشار ویرایش جدید کتاب Entity Framework دارین؟

نویسنده: بهروز راد
تاریخ: ۱۳۹۱/۰۴/۱۲ ۱۰:۱۴

لطفاً سوالات اینچنینی رو از طریق ایمیل behrouz.rad[at]signlmail بپرسید.
بله، بعد از کتاب ASP.NET MVC، کتاب Entity Framework رو آپدیت می‌کنم.

نویسنده: torisoft
تاریخ: ۱۳۹۱/۰۴/۱۶ ۲۳:۵

سلام جناب راد
از Web API تو سیلورلایت هم میشه استفاده کرد ؟
اگه استفاده میشه آیا مثبت میدونید استفاده از اونو تو سیلور ؟
با تشکر

نویسنده: بهروز راد
تاریخ: ۱۳۹۱/۰۴/۱۷ ۸:۱۱

بله مشکلی نداره. پروژه‌ی Silverlight رو در یک پروژه‌ی وب Host کنید.
Silverlight هم یک نوع پروژه است، مثل Web و Desktop. اگر پروژه‌ی شما بر مبنای Silverlight هست و نیاز دارید تا امکانات اون رو به صورت سرویس ارائه بدید، می‌تونید از Web API برای عرضه‌ی این امکانات استفاده کنید.

نویسنده: حمید
تاریخ: ۱۳۹۱/۰۴/۱۷ ۱۱:۲۲

سلام. وقت بخیر.

مطالب خیلی خوب و به روزی دارین و خدا قوت..

با عرض معذرت می‌خواستم بگم من MVC4 رو نصب کردم اما بازم بعد انتخاب MVC4 از لیست Template های ویژوال استودیو گزینه Web API رو مشاهده نمی‌کنم. آیا افزونه یا برنامه خاصی باید نصب کنم. از قبل از زحمتتون تشکر می‌کنم.

نویسنده:

وحید نصیری

تاریخ:

۱۱:۵۰ ۱۳۹۱/۰۴/۱۷

مراجعه کنید به [قسمت دوم](#) ، تصویر سوم

نویسنده:

حمید

تاریخ:

۲۲:۵۷ ۱۳۹۱/۰۴/۲۰

سلام. مشکل من همینکه که همین تصویر سوم رو که می‌گین تو این بخش من گزینه Web API رو ندارم.

نویسنده:

وحید نصیری

تاریخ:

۲۳:۱۱ ۱۳۹۱/۰۴/۲۰

از طریق NuGet هم می‌تونید برای نصب آن اقدام کنید. این رو هم تست کنید:

<http://nuget.org/packages/aspnetwebapi>

نویسنده:

Saeed M. Farid

تاریخ:

۱۱:۳ ۱۳۹۱/۰۴/۲۱

سلام و ممنون از مطلب مفید:

امکانش هست در مورد "سویچ بین کانال‌ها در هنگام فعال نبودن یک کانال" کمی بیشتر راهنمایی کنید یا مرجع (لینک) معرفی کنید؟ من از صحبت شما اینطور برداشت کردم که میشه در [channel shape](#) (های)ی که مثلاً برای duplex communications (یعنی [IDuplexChannel](#)) پیاده سازی کردم، اگه چنین کانالی در دسترس نبود، سوئیچ کنه روی طراحی مبتنی بر one-way messaging من؟ اصلاً چنین امکانی در سطح [IChannelListener](#) هست یا [ChannelFactory](#) ؟ کلاً اگه ممکنه یه توضیح کلی در مورد چنین امکانی که در موردش صحبت کردین بدین یا اگه جایی در موردش قبلاً بحث شده (که حتماً شده!) من رو هدایت کنید به اون، چون گلوگاه سیستم‌هام همین مورد هست. پیشاپیش ازتون ممنونم...

نویسنده:

بهروز راد

تاریخ:

۱۹:۲۹ ۱۳۹۱/۰۴/۲۱

اصولاً در Web API چیزی با عنوان Channel با اون مفهوم که در WCF هست نداریم. در Web API فقط یک Transport Channel برای HTTP وجود داره، چون هدف ایجاد Web API، فقط برقراری ارتباط در سطح HTTP هست، نه مثلاً MSMQ. Protocol Channel هم همان مفاهیمی هستند که در ASP.NET MVC وجود دارند و مثلاً قسمتی از اون، تصدیق هویت و تعیین مجوز کاربر برای دسترسی به منابع با استفاده از فیلتر Authorize هست. لطفاً دنبال تطبیق و تناظر بین مفاهیم پیچیده‌ی WCF و یافتن معادل در Web API نباشید. Web API به وجود آمده تا ایجاد وب سرویس‌ها در بستر HTTP رو ساده کنه، همین!

نویسنده:

سیروس

تاریخ:

۱۸:۳۳ ۱۳۹۱/۱۲/۰۲

سلام

یک سوال مهم داشتم، آیا استفاده از web api در Windows Form مانند WCF ممکن است، یعنی پروژه ما هم هاست و هم کلاینت رو MVC یا ASP.Net نیست، اگه میشه یه منبع معرفی کنید.

نویسنده: محسن
تاریخ: ۱۳۹۱/۱۲/۰۲ ۲۲:۳۶

اولین نتیجه جستجوی گوگل در مورد winforms web api :

[Using Microsoft Web API from a Windows and WinRT Client Application](#)

نویسنده: محمد آزاد
تاریخ: ۱۳۹۱/۱۲/۰۲ ۲۲:۳۸

تو مقدمه به این مطلب اشاره شده دوست عزیز
این سبک جدید برای ایجاد وب سرویس ها، می تواند در انواع پروژه های .NET. مانند ASP.NET MVC, ASP.NET Web Forms, Windows Application و ... استفاده شود.

نویسنده: سیروس
تاریخ: ۱۳۹۱/۱۲/۰۵ ۱۰:۱۰

محسن <= دوست عزیز من اون مطلب رو قبلا هم مطالعه کردم، قسمت هاست رو MVC ست. اینقدر بی سواد نیستم که نتونم سرچ کنم.
آزاد <= میدونم که تو Win APP قابل استفاده هست، اما می خوام بدونم پروژه هاست مثل WCF می تونه رو مستقل از Asp.Net باشه یا نه چون ظاهرا پیاده سازی WebAPI فقط روی ASP.Net امکان پذیر است.

نویسنده: محسن
تاریخ: ۱۳۹۱/۱۲/۰۵ ۱۰:۱۶

نتیجه جستجوی گوگل در مورد [wep api self host](#) :

[Self-Host a Web API](#)

نویسنده: یاسر مرادی
تاریخ: ۱۳۹۱/۱۲/۰۵ ۱۲:۳۶

من فکر کنم مطلب این دوستان رو این جوری مطرح کنم بهتره
وقتی شما از WCF Data Services استفاده می کنید، WCF Data Services Client دارید، که به شما امکان نوشتن کوئری های Linq در سمت کلاینت، Merge و Change Tracking و ... رو می ده
اما من همچین آیتی رو برای Web API پیدا نکردم، بهترین چیزی که دیدم HttpClient بوده که در حد مثال زدن خوبه، ولی به درد پروژه نویسی نمی خوره، این که شما یک کلاینت قوی داشته باشید، خیلی مهمه، HttpClient تفاوت مفهومی زیادی با \$.ajax نداره
حتی در JayData هم همین طور هستش، و شما پشتیبانی خیلی بهتری از WCF Data Services می بینید تا از Web API، همین طور در Breeze.js در اندروید و iOS هم شما پشتیبانی WCF Data Services Client رو دارید، ولی Web API خیر موفق باشید

نویسنده: محسن
تاریخ: ۱۳۹۱/۱۲/۰۵ ۱۲:۵۸

سؤال مطرح شده در مورد هاست کردن یک سرویس در برنامه ویندوزی بود که اصطلاحاً Self hosting نام دارد.

Web API امکان استفاده از OData را هم دارد:

[Getting started with ASP.NET Web API OData in 3 simple steps](#)

نویسنده: یاسر مرادی
تاریخ: ۱۳۹۱/۱۲/۰۵ ۱۳:۳۵

قبول، ولی در هر حال آیا راهی جز Http Client برای دسترسی به Web API وجود دارد ؟

مثلاً مدل Linq به OData ؟
به همراه Change Tracking و ...
در ضمن موارد مهمی از OData مانند \$batch در Web API پشتیبانی نشده اند، و باید برایشان Message Formatter نوشت، این نیز کار را سخت می‌کند
بر خلاف نظر دوستان به نظر من به هیچ وجه هیچ فریم ورکی راحت‌تر از WCF Data Services وجود ندارد، که جمعاً با 3 خط کد راه اندازی می‌شود.

نویسنده: محسن
تاریخ: ۱۳۹۱/۱۲/۰۵ ۱۷:۲۴

NuGet مربوط به [Web API OData](#) مرتباً به روز میشه. آخرین به روز رسانی آن مربوط به 5 روز قبل بوده.

ضمن اینکه خروجی OData استاندارد است. بنابراین با کلاینت‌های موجود کار می‌کنه. فرقی نمی‌کنه تولید کننده چی هست تا زمانیکه استاندارد رعایت بشه.

نویسنده: یاسر مرادی
تاریخ: ۱۳۹۱/۱۲/۰۵ ۱۸:۰۵

دوست عزیز، فکر کنم سوال من خیلی واضح باشه
مسئله اول این هستش که مواردی از OData هست که در WCF Data Services وجود داره، ولی در Web API خیر، OData یک سری استاندارد هستش، بالاخره باید یک جایی پیاده سازی بشه، مثل HTML 5، که قسمت‌های مختلفش در درصدهای متفاوت در مرورگرهای متفاوت پیاده سازی شده، در این میان Chrome بهتر از IE هستش، چرا ؟ چون استانداردهای بیشتری رو پیاده سازی کرده
دوم این که آیا شما به صورت عملی از js Breeze و Jay Data و WCF Data Services Client استفاده کرده اید ؟ درسته که اینها به OData وصل می‌شوند، ولی میزان امکانات اینها برای WCF Data Services قابل قیاس با Web API نیست.
سوال اصلی من با این تفاسیر این است :
اگر قبول کنیم که راهی برای دسترسی به Web API وجود ندارد، الا استفاده از jQuery Ajax و Http Client، شما به چه صورت یک پروژه بزرگ رو با Web API می‌نویسید ؟
Change Tracking رو چه جوری پیاده سازی می‌کنید ؟
به چه صورت در کلاینت‌هایی مانند اندروید، و یا Win RT و ... از Linq برای دسترسی به سرویس هاتون استفاده می‌کنید ؟
اگر فرض کنیم که می‌خواهیم یک سرویس عمومی بنویسیم که همه جا به سادگی قابل استفاده باشه، آیا از Web API استفاده می‌کنید ؟
خلاصه : مزیت واقعی Web API چیست و چه زمانی پروژه ای رو با Web API شروع می‌کنید ؟
موفق و پایدار باشید

نویسنده: محسن
تاریخ: ۱۳۹۱/۱۲/۰۵ ۱۸:۳۱

«مزیت واقعی Web API چیست و چه زمانی پروژه ای رو با Web API شروع می‌کنید؟»

[WCF or ASP.NET Web APIs](#)

به علاوه هدف اصلی Web API و یکپارچگی آن با خصوصاً MVC (و بعد وب فرم‌ها) در درجه اول توسعه ActionResult‌های پیش فرض MVC است (به همین جهت اول اسم آن ASP.NET است و نه مثلاً اندروید):

[ASP.NET Web API vs. ASP.NET MVC APIs](#)

نویسنده: یاسر مرادی
تاریخ: ۱۳۹۱/۱۲/۰۵ ۱۹:۳۱

مقاله اول Web API رو با WCF خام مقایسه کرده، نه با WCF Data Services

مقاله دوم هم Action‌های Web API رو با MVC قیاس کرده
اگر شما یک مقاله بنویسید که مثلاً Web API رو با ASP.NET Web Service قیاس بکنه، و نشون بده مزیت‌های Web API بیشتره، این می‌شه مزیت Web API بر ASP.NET Web Service، نه بر WCF Data Services
ممکنه این موارد هم مهم باشند، ولی اون چیزی که برای من سوال شده این هستش که چه زمانی در یک پروژه WCF Data Services رو می‌گذاریم کنار و از Web API استفاده می‌کنیم؟
در واقع با توجه به امکانات واقعاً زیاد WCF Data Services چرا باید اساساً از Web API استفاده بشه، اگر شما می‌فرمایید که 5 روز پیش برای Web API نسخه آمده، این عدد برای Data Services چهار روز پیش بوده
اگر بحث امکانات هست، لیست زیادی از امکانات رو من شمردم و می‌شه شمرد، از امکاناتی که تو Data Services هست، ولی تو Web API نیست.
اگر من اندروید رو مثال زدم، برای سمت کلاینت بود، شما در اندروید با چی به Web API وصل می‌شید؟
با jQuery Ajax؟
یا می‌خواهید به App Server‌های .NET. ای برنامه‌های دیگر، بگویید با Http Client از سرویس‌های شما استفاده کنند؟
با سپاس

نویسنده: محسن
تاریخ: ۱۳۹۱/۱۲/۰۵ ۱۹:۱۶

هدف میکروسافت از یکپارچه کردن WEB API با ASP.NET و خصوصاً MVC ارائه یک سری Super ActionResult است بجای ActionResult‌های معمولی MVC3. برای نمونه:

[Using Kendo UI grid with Web API and OData](#)

نویسنده: میثم 99
تاریخ: ۱۳۹۳/۰۱/۲۱ ۲۳:۰۰

سلام
می‌خواهم بدانم برای امنیت web api در پروژه های web form چه کارهایی باید انجام دهیم بیشتر مطالب در مورد mvc هست مثلاً Anti-Forgery Tokens برای mvc به راحتی می‌توان استفاده کرد ولی برای web form چکار بهتر است انجام دهیم؟
در اینجا ما مستقیماً با دستورات post put و delete کار داریم که اطلاعات بانک اطلاعاتی رو تغییر می‌دهند. حالا چطور می‌توان امنیت رو کاملاً تامین کرد؟

مثلا کاربران شناسایی شده اطلاعات را وارد کنند و اینکه شخصی نتواند با یک دستور ای جکس توسط مرورگر اطلاعات اشتباه در سایت ثبت کند؟ و یا هر مشکل امنیتی دیگری که پیش بیاید؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۲۱ ۲۳:۲۱

روش‌های زیادی برای تامین امنیت در وب API و کار با «کاربران شناسایی شده» وجود دارند. [لیست رسمی](#) از این لیست رسمی، دو مورد معروف آن در سایت جاری بررسی شده:

[ASP.NET Identity](#)

[Forms authentication](#)

مباحث پایه‌ای این‌ها مشترک است بین MVC و وب فرم‌ها و سایر فناوری‌های مشابه.

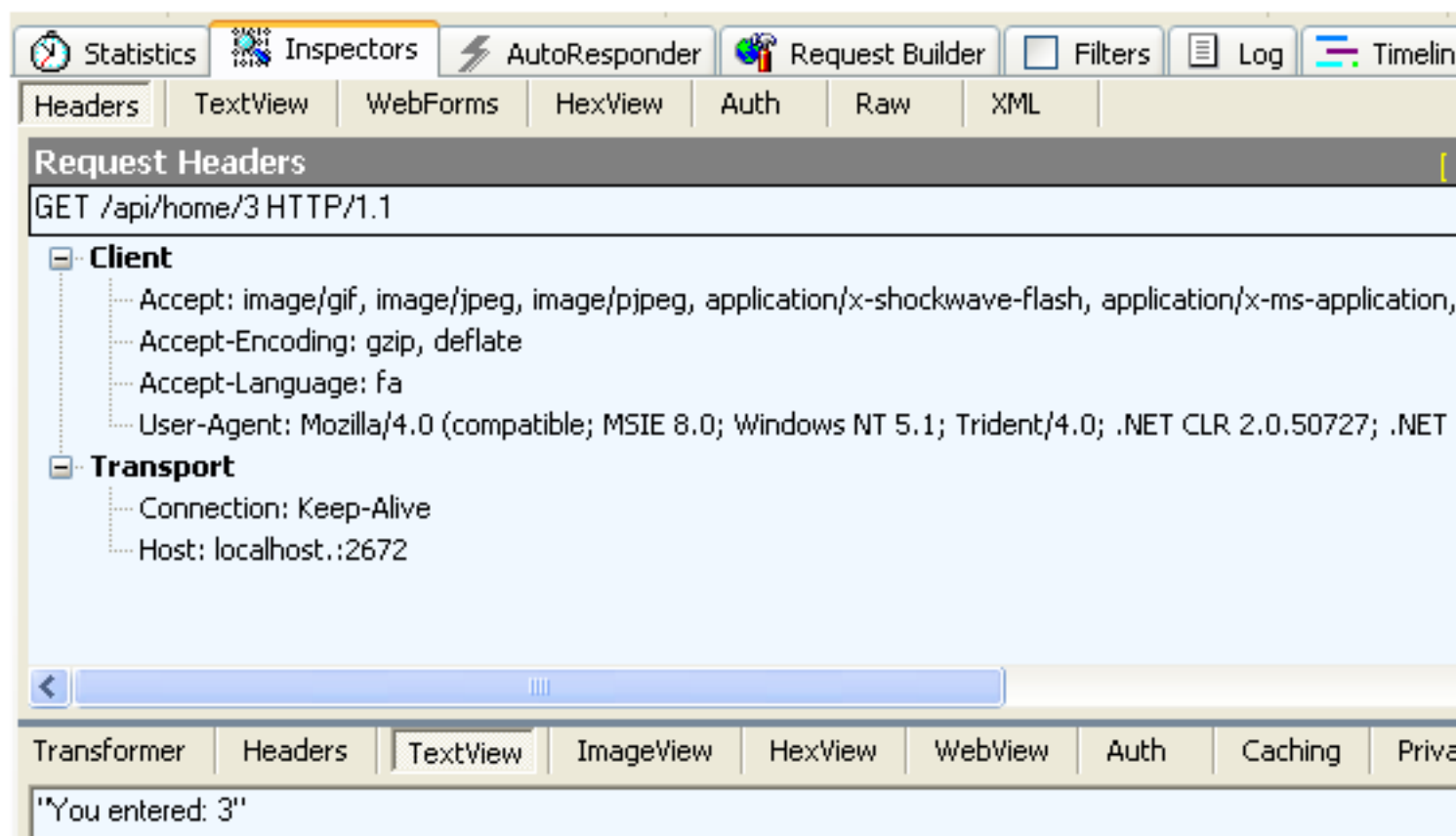
در [قسمت اول](#) به دلایل ایجاد ASP.NET Web API پرداخته شد. در این قسمت، یک مثال ساده از Web API را بررسی می‌کنیم. تلاش‌های بسیاری توسط توسعه گران صورت پذیرفته است تا فرایند ایجاد وب سرویس WCF در بستر HTTP آسان شود. امروزه وب سرویس‌هایی که از قالب REST استفاده می‌کنند مطرح هستند. ASP.NET Web API از مفاهیم موجود در ASP.NET MVC مانند Controllerها استفاده می‌کند و بر مبنای آنها ساخته شده است. بدین شکل، توسعه گر می‌تواند با دانش موجود خود به سادگی وب سرویس‌های مورد نظر را ایجاد کند. Web API، پروتوکل SOAP را به کتاب‌های تاریخی! سپرده است تا از آن به عنوان روشی برای تعامل بین سیستم‌ها یاد شود. امروزه به دلیل فراگیری پروتوکل HTTP، بیشتر محیط‌های برنامه نویسی و سیستم‌ها، از مبنای اولیه‌ی پروتوکل HTTP مانند افعال آن پشتیبانی می‌کنند. حال قصد داریم تا وب سرویسی را که در قسمت اول با WCF ایجاد کردیم، این بار با استفاده از Web API ایجاد کنیم. به تفاوت این دو دقت کنید.

```
using System.Web.Http;

namespace MvcApplication1.Controllers
{
    public class ValuesController : ApiController
    {
        // GET api/values/5
        public string Get(int id)
        {
            return string.Format("You entered: {0}", id);
        }
    }
}
```

اولین تفاوتی که مشهود است، تعداد خطوط کمتر مورد نیاز برای ایجاد وب سرویس با استفاده از Web API است، چون نیاز به interface و کلاس پیاده ساز آن وجود ندارد. در Web API، Controllerهایی که در نقش وب سرویس هستند از کلاس ApiController ارث می‌برند. اعمال مورد نظر در قالب متدها در Controller تعریف می‌شوند. در مثال قبل، متد Get، یکی از اعمال است.

نحوه‌ی برگشت یک مقدار از متدها در Web API، مانند WCF است. می‌توانید خروجی متد Get را با اجرای پروژه‌ی قبل در Visual Studio و تست آن با یک مرورگر ملاحظه کنید. دقت داشته باشید که یکی از اصولی که Web API به آن معتقد است این است که وب سرویس‌ها می‌توانند ساده باشند. در Web API، تست و دیباگ وب سرویس‌ها بسیار راحت است. با مرورگر Internet Explorer به آدرس <http://localhost:{port}/api/values/3> بروید. پیش از آن، برنامه‌ی [Fiddler](#) را اجرا کنید. شکل ذیل، نتیجه را نشان می‌دهد.

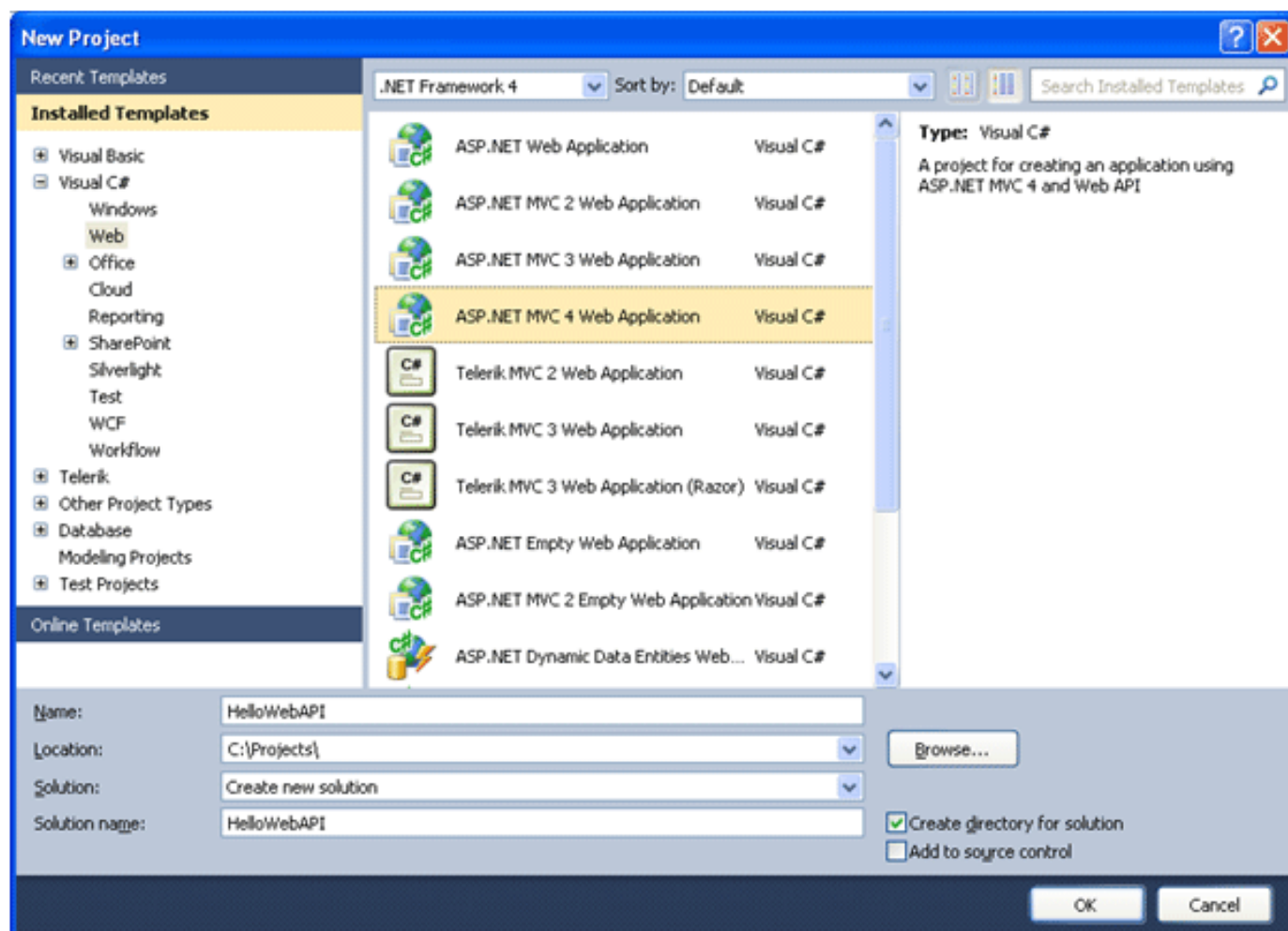


در اینجا نتیجه، عبارت "You entered: 3" است که به صورت یک متن ساده برگشت داده شده است.

ایجاد یک پروژه Web API

در Visual Studio، مسیر ذیل را طی کنید.

File> New> Project> Installed Templates> Visual C#> Web> ASP.NET MVC 4 Web Application
نام پروژه را HelloWorldAPI بگذارید و بر روی دکمه OK کلیک کنید (شکل ذیل)



در فرمی که باز می‌شود، گزینه‌ی Web API را انتخاب و بر روی دکمه‌ی OK کلیک کنید (شکل ذیل). البته دقت داشته باشید که ما همیشه مجبور به استفاده از قالب Web API برای ایجاد پروژه‌های خود نیستیم. می‌توان در هر نوع پروژه ای از Web API استفاده کرد.



اضافه کردن مدل

مدل، شی ای است که نمایانگر داده‌ها در برنامه است. Web API می‌تواند به طور خودکار، مدل را به فرمت XML، JSON یا فرمت دلخواهی که خود می‌توانید برای آن ایجاد کنید تبدیل و سپس داده‌های تبدیل شده را در بدنه‌ی پاسخ HTTP به Client ارسال کند. تا زمانی که Client بتواند فرمت دریافتی را بخواند، می‌تواند از آن استفاده کند. بیشتر Clientها می‌توانند فرمت JSON یا XML را پردازش کنند. به علاوه، Client می‌تواند نوع فرمت درخواستی از Server را با تنظیم مقدار هدر Accept در درخواست ارسالی تعیین کند. اجازه بدهید کار خود را با ایجاد یک مدل ساده که نمایانگر یک محصول است آغاز کنیم. بر روی پوشه‌ی Models کلیک راست کرده و از منوی Add، گزینه‌ی Class را انتخاب کنید.

نام کلاس را Product گذاشته و کدهای ذیل را در آن بنویسید.

```
namespace HelloWebAPI.Models
{
    public class Product
    {
```

```

    public int Id { get; set; }
    public string Name { get; set; }
    public string Category { get; set; }
    public decimal Price { get; set; }
}

```

مدل ما، چهار Property دارد که در کدهای قبل ملاحظه می‌کنید.

اضافه کردن Controller

در پروژه ای که با استفاده از قالب پیش فرض Web API ایجاد می‌شود، دو Controller نیز به طور خودکار در پروژه‌ی Controller قرار می‌گیرند:

HomeController: یک Controller معمولی ASP.NET MVC است که ارتباطی با Web API ندارد.
 ValuesController: یک Controller مختص Web API است که به عنوان یک مثال در پروژه قرار داده می‌شود.

توجه: Controllerها در Web API بسیار شبیه به Controllerها در ASP.NET MVC هستند، با این تفاوت که به جای کلاس Controller، از کلاس ApiController ارث می‌برند و بزرگترین تفاوتی که در نگاه اول در متدهای این نوع کلاس‌ها به چشم می‌خورد این است که به جای برگشت Viewها، داده برگشت می‌دهند.

کلاس ValuesController را حذف و یک Controller به پروژه اضافه کنید. بدین منظور، بر روی پوشه‌ی Controllers، کلیک راست کرده و از منوی Add، گزینه‌ی Controller را انتخاب کنید.

توجه: در ASP.NET MVC 4 می‌توانید بر روی هر پوشه‌ی دلخواه در پروژه کلیک راست کرده و از منوی Add، گزینه‌ی Controller را انتخاب کنید. پیشتر فقط با کلیک راست بر روی پوشه‌ی Controller، این گزینه در دسترس بود. حال می‌توان کلاس‌های مرتبط با Controllerهای معمول را در یک پوشه و Controllerهای مربوط به قابلیت Web API را در پوشه‌ی دیگری قرار داد.

نام Controller را ProductsController بگذارید، از قسمت Template، گزینه‌ی Empty API Controller را انتخاب و بر روی دکمه‌ی OK کلیک کنید (شکل ذیل).

Add Controller

Controller name:
ProductsController

Scaffolding options

Template:
Empty API controller

Model class:

Data context class:

Views:
None

Advanced Options...

Add Cancel

فایلی با نام ProductsController.cs در پوشه‌ی Controllers قرار می‌گیرد. آن را باز کنید و کدهای ذیل را در آن قرار دهید.

```
namespace HelloWebAPI.Controllers
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Net;
    using System.Net.Http;
    using System.Web.Http;
    using HelloWebAPI.Models;

    public class ProductsController : ApiController
    {
        Product[] products = new Product[]
        {
            new Product { Id = 1, Name = "Tomato Soup", Category = "Groceries", Price = 1.39M },
            new Product { Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.75M },
            new Product { Id = 3, Name = "Hammer", Category = "Hardware", Price = 16.99M }
        };

        public IEnumerable<Product> GetAllProducts()
        {
            return products;
        }

        public Product GetProductById(int id)
        {
            var product = products.FirstOrDefault((p) => p.Id == id);
            if (product == null)
            {
                var resp = new HttpResponseMessage(HttpStatusCode.NotFound);
                throw new HttpResponseException(resp);
            }
            return product;
        }
    }
}
```

```

    }
    public IEnumerable<Product> GetProductsByCategory(string category)
    {
        return products.Where(
            (p) => string.Equals(p.Category, category,
                StringComparison.OrdinalIgnoreCase));
    }
}

```

برای ساده نگهداشتن مثال، لیستی از محصولات را در یک آرایه قرار داده ایم اما واضح است که در یک پروژه‌ی واقعی، این لیست از پایگاه داده بازیابی می‌شود. در مورد کلاس‌های `HttpResponseMessage` و `HttpResponseException` بعداً توضیح می‌دهیم. در کدهای `Controller` قبل، سه متد تعریف شده اند:

متد `GetAllProducts` که کل محصولات را در قالب نوع `IEnumerable<Product>` برگشت می‌دهد.
 متد `GetProductById` که یک محصول را با استفاده از مشخصه‌ی آن (خصیصه‌ی `Id`) برگشت می‌دهد.
 متد `GetProductsByCategory` که تمامی محصولات موجود در یک دسته‌ی خاص را برگشت می‌دهد.

تمام شد! حال شما یک وب سرویس با استفاده از `Web API` ایجاد کرده اید. هر یک از متدهای قبل در `Controller`، به یک آدرس به شرح ذیل تناظر دارند.

`/api/products` به `GetAllProducts`

`/api/products/ id` به `GetProductById`

`/api/products/?category= category` به `GetProductsByCategory`

در آدرس‌های قبل، `id` و `category`، مقادیری هستند که همراه با آدرس وارد می‌شوند و در پارامترهای متناظر خود در متدهای مربوطه قرار می‌گیرند. یک `Client` می‌تواند هر یک از متدها را با ارسال یک درخواست از نوع `GET` اجرا کند.

در قسمت بعد، کار خود را با تست پروژه و نحوه‌ی تعامل `jQuery` با آن ادامه می‌دهیم.

نظرات خوانندگان

نویسنده: mze666
تاریخ: ۸:۷ ۱۳۹۱/۰۴/۱۳

سلام آقای راد من MVC رو بلدم ولی کاربرد این WebApi , Web Service رو نمیدونم. یعنی اگر براتون ممکنه چند تا مثال واقعی از این که کجاها استفاده میشه بزنید. ممنون.

نویسنده: بهروز راد
تاریخ: ۸:۱۳ ۱۳۹۱/۰۴/۱۳

وب سرویس‌ها کاربردهای متفاوتی دارند. برای ارتباط بین سیستم‌ها، استفاده از داده‌هایی که توسط یک شرکت عرضه میشه مثل اطلاعات آب و هوا یا بورس، عملیات‌های مختلفی که بر روی پایگاه داده انجام میشه، ارسال SMS، تراکنش‌های بانکی و ...

نویسنده: ایمان اسلامی
تاریخ: ۸:۱۵ ۱۳۹۱/۰۴/۱۳

ممنون از مطالب خوبتون
امیدوارم به همین شکل مطلوب ادامه داشته باشه و بهتر از اون ، به زودی شاهد چاپ کتابتون باشیم.

نویسنده: زهرا
تاریخ: ۸:۲۳ ۱۳۹۱/۰۴/۱۳

سلام آقای راد

میخواستم بپرسم که 4 mvc رو چطور به لیست پروژه هام اضافه کنم؟ و اینکه آیا این کاری که شما انجام دادید در asp.net webform هم جواب میده؟ یا اینکه باید در solution یک پروژه 4 mvc ایجاد کرد و از اون استفاده کرد؟

با تشکر

نویسنده: بهروز راد
تاریخ: ۹:۲۷ ۱۳۹۱/۰۴/۱۳

سلام.
اگر نسخه‌ی آفلاین RC اون رو میخوايد، از [این لینک](#) دریافت کنید.
بله، Web API در ASP.NET Web Forms هم قابل استفاده است.
در پروژه‌های Web Forms، از دیالوگ Add New Item، گزینه‌ی Web API Controller Class رو باید انتخاب کنید. route رو هم باید در متد Application_Start فایل Global.asax به صورت ذیل تعریف کنید.

```
void Application_Start(object sender, EventArgs e)
{
    RouteTable.Routes.MapHttpRoute(
        name: "DefaultApi",
        routeTemplate: "api/{controller}/{id}",
        defaults: new { id = System.Web.Http.RouteParameter.Optional }
    );
}
```

نویسنده: علی

تاریخ: ۷:۱۲ ۱۳۹۱/۰۸/۲۱

سلام آقای راد
 نمی دونم چطور می شه از آدمایی مثل شما تشکر کرد، مطالب واقعا مفید و آموزندس
 خیلی خیلی متشکرم
 آقای راد یک سوال از خدمتتون داشتم، مدتی که من و خانمم در حال ترجمه یک کتاب wcf هستیم ، این اولین کار ترجمه می
 خواستم ازتون بپرسم که میزان محبوبیت wcf الان تو ایران چقدره ، به نظر شما آینده ای داره ؟ کلا چقدر ارزش وقت گذاشتن
 داره ؟

نویسنده: محمد صاحب
 تاریخ: ۸:۴۶ ۱۳۹۱/۰۸/۲۱

دوست عزیز امیدوارم موفق باشید.
 تا آقای راد جواب شما رو بدن این [کامنت](#) و قسمت [حاشیه](#) این پست رو ببیند بی ارتباط نیست...

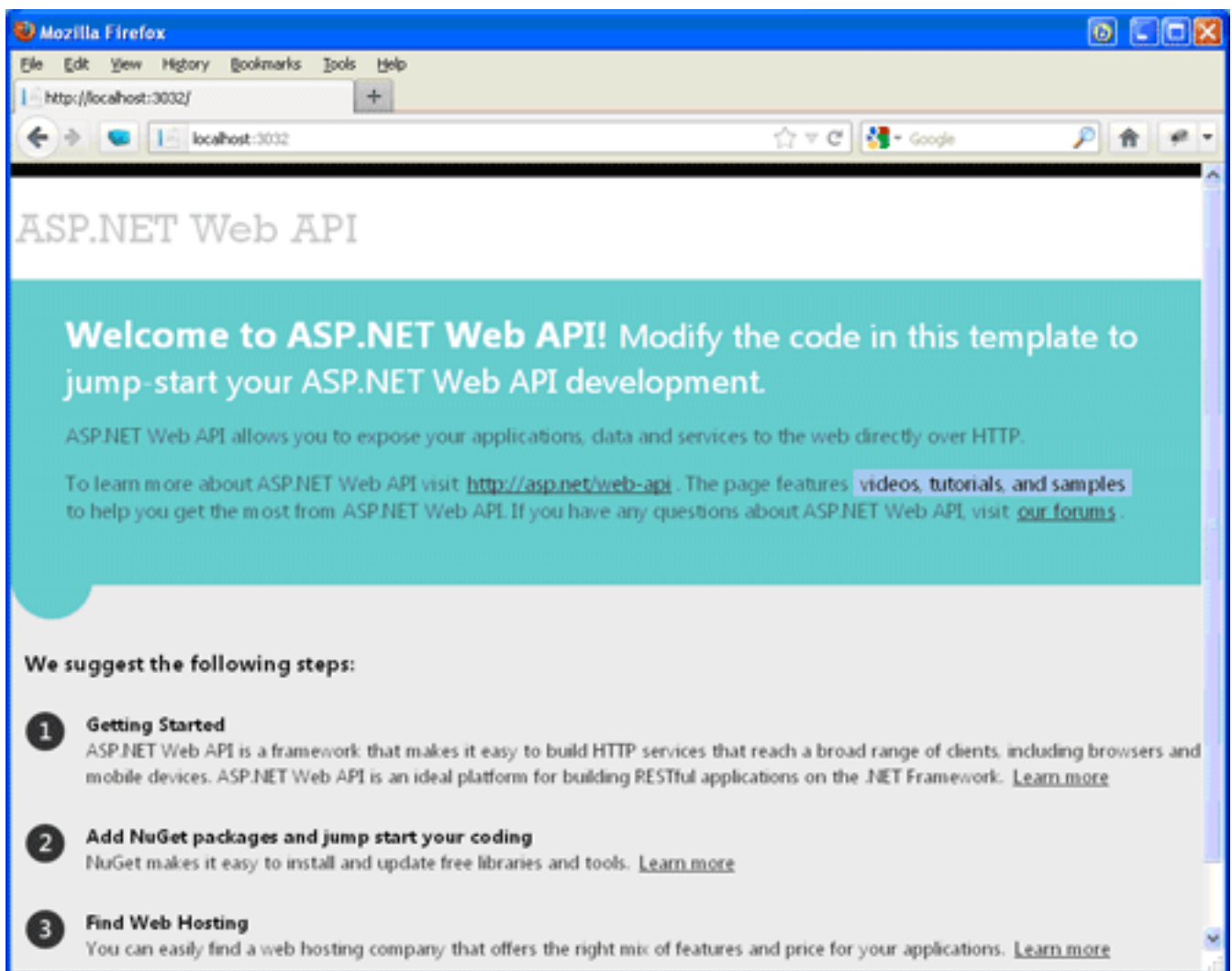
نویسنده: بهروز راد
 تاریخ: ۱۸:۳۴ ۱۳۹۱/۰۸/۲۱

در مورد میزان محبوبیت WCF در ایران اطلاعی ندارم و مطلب خاصی هم در مورد اون در وبلاگ های فارسی زبان منتشر نمیشه. اما
 در حوزه ای که مربوط به خودم هست، حداقل قسمتی از پروژه های شرکت فولاد خوزستان که با تیم سازنده ای اونها ارتباط دارم
 از WCF در پروژه های اتوماسیون استفاده می کنند.
 در مورد قسمت دوم سوالتون هم که دوستمون لینک های خوبی قرار دادند.

در [قسمت اول](#) به دلایل ایجاد Web API پرداخته شد و در [قسمت دوم](#) مثالی ساده از Web API را بررسی کردیم. در این قسمت، مثال قبل را تست کرده و نحوه‌ی تعامل jQuery با آن را بررسی می‌کنیم.

فراخوانی Web API از طریق مرورگر

با فشردن کلید F5، پروژه را اجرا کنید. شکل ذیل ظاهر می‌شود.



صفحه‌ای که ظاهر می‌شود، یک View است که توسط HomeController و متد Index آن برگشت داده شده است. برای فراخوانی متدهای موجود در کلاس Controller مثال قسمت قبل که مربوط به Web API است، باید به یکی از آدرس‌های اشاره شده در قسمت قبل برویم. به عنوان مثال، برای به دست آوردن لیست تمامی محصولات، به آدرس `http://localhost: xxxx /api/products` بروید. xxxx، شماره‌ی پورتی است که Web Server داخلی Visual Studio در هنگام اجرای پروژه به آن اختصاص می‌دهد. آن را نسبت به پروژه‌ی خود تغییر دهید. نتیجه‌ی دریافتی بستگی به نوع مرورگری دارد که استفاده می‌کنید. Internet Explorer از شما در مورد باز کردن یا ذخیره‌ی

فایلی با نام products پرسش می‌کند (شکل ذیل).



محتوای فایل، بدنه‌ی پاسخ دریافتی است. اگر این فایل را باز کنید، خواهید دید که محتوای آن، لیستی از محصولات با فرمت JSON مانند ذیل است.

```
[{"Id":1,"Name":"Tomato soup","Category":"Groceries","Price":1.39}, {"Id":2,"Name":"Yo-yo","Category":"Toys","Price":3.75}, {"Id":3,"Name":"Hammer","Category":"Hardware","Price":16.99}]
```

اما مرورگر Firefox، محصولات را در قالب XML نشان می‌دهد (شکل ذیل).

```
- <ArrayOfProduct>
  - <Product>
    <Category>Groceries</Category>
    <Id>1</Id>
    <Name>Tomato Soup</Name>
    <Price>1.39</Price>
  </Product>
  - <Product>
    <Category>Toys</Category>
    <Id>2</Id>
    <Name>Yo-yo</Name>
    <Price>3.75</Price>
  </Product>
  - <Product>
    <Category>Hardware</Category>
    <Id>3</Id>
    <Name>Hammer</Name>
    <Price>16.99</Price>
  </Product>
</ArrayOfProduct>
```

دلیل تفاوت در نتیجه‌ی دریافتی این است که مرورگر Internet Explorer و Firefox، هر یک مقدار متفاوتی را در هدر Accept درخواست، ارسال می‌کنند. بنابراین، Web API نیز مقدار متفاوتی را در پاسخ برگشت می‌دهد.

حال به آدرس‌های ذیل بروید:

`http://localhost: xxxx /api/products/1`

`http://localhost: xxxx /api/products?category=hardware`

اولین آدرس، باید محصولی با مشخصه‌ی 1 را برگشت دهد و دومین آدرس، لیستی از تمامی محصولات که در دسته‌ی hardware قرار دارند را برگشت می‌دهد (در مثال ما فقط یک آیتم این شرط را دارد).

نکته: در صورتی که در هنگام فراخوانی هر یک از متدهای Web API با خطای ذیل مواجه شدید، دستور `AcceptVerbs("GET",")` را به ابتدای متدها اضافه کنید.

The requested resource does not support http method 'GET'

فراخوانی Web API با استفاده از کتابخانه‌ی jQuery

در قسمت قبل، متدهای Web API را مستقیماً از طریق وارد کردن آدرس آنها در نوار آدرس مرورگر فراخوانی کردیم. اما در اکثر اوقات، این متدها با روش‌های برنامه نویسی توسط یک Client فراخوانی می‌شوند. اجازه بدهید Clientی ایجاد کنیم که با استفاده از jQuery، متدهای ما را فراخوانی می‌کند. در Solution Explorer، از پوشه‌ی Views و سپس Home، فایل Index.cshtml را باز کنید.

تمامی محتویات این View را حذف و کدهای ذیل را در آن قرار دهید.

```
<!DOCTYPE html>
<html>
<head>
  <title>ASP.NET Web API</title>
  <script src="../../Scripts/jquery-1.7.2.min.js"
    type="text/javascript"></script>
</head>
<body>
  <div>
    <h1>All Products</h1>
    <ul id='products' />
  </div>
  <div>
    <label for="prodId">ID:</label>
    <input type="text" id="prodId" size="5"/>
    <input type="button" value="Search" onclick="find();" />
    <p id="product" />
  </div>
</body>
</html>
```

بازیابی لیستی از محصولات

برای بازیابی لیستی از محصولات، فقط کافی است تا یک درخواست از نوع GET به آدرس `"api/products/"` بفرستید. این کار با jQuery به صورت ذیل انجام می‌شود.

```
<script type="text/javascript">
$(document).ready(function () {
    // Send an AJAX request
    $.getJSON("api/products/",
        function (data) {
            // On success, 'data' contains a list of products.
            $.each(data, function (key, val) {

                // Format the text to display.
                var str = val.Name + ': $' + val.Price;

                // Add a list item for the product.
                $('<li/>', { html: str })
                    .appendTo($('#products'));
            });
        });
});
</script>
```

متد `getJSON`، یک درخواست AJAX از نوع GET را ارسال می‌کند و پاسخ دریافتی آن نیز با فرمت JSON خواهد بود. دومین پارامتر متد `getJSON`، یک callback است که پس از دریافت موفقیت آمیز پاسخ اجرا می‌شود.

بازیابی یک محصول با استفاده از مشخصه‌ی آن

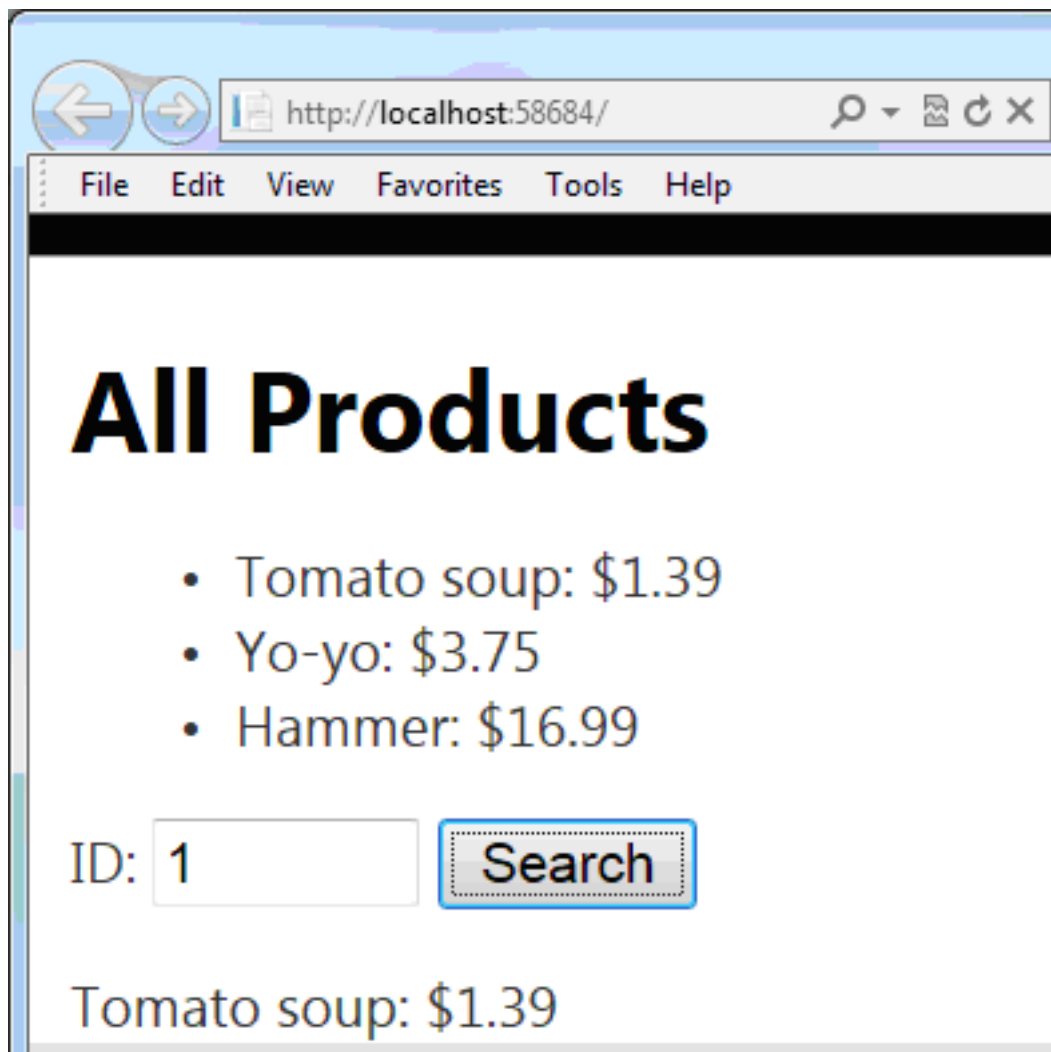
برای بازیابی یک محصول با استفاده از مشخصه‌ی آن، یک درخواست از نوع GET به آدرس `" / api/products/ id"` ارسال کنید. `id`، مشخصه‌ی محصول است. کد ذیل را در ادامه‌ی کد قبل و پیش از تگ `</script>` قرار دهید.

```
function find() {
    var id = $('#prodId').val();
    $.getJSON("api/products/" + id,
        function (data) {
            var str = data.Name + ': $' + data.Price;
            $('#product').html(str);
        })
    .fail(
        function (jqXHR, textStatus, err) {
            $('#product').html('Error: ' + err);
        });
}
```

باز هم از متد `getJSON` استفاده کردیم، اما این بار مقدار `id` برای آدرس از یک Text Box خوانده و آدرس ایجاد می‌شود. پاسخ دریافتی، یک محصول در قالب JSON است.

اجرای پروژه

پروژه را با فشردن کلید F5 اجرا کنید. پس از نمایش فرم، تمامی محصولات بر روی صفحه نمایش داده می‌شوند. عدد 1 را وارد و بر روی دکمه‌ی Search کلیک کنید، محصولی که مشخصه‌ی آن 1 است نمایش داده می‌شود (شکل ذیل).



اگر مشخصه ای را وارد کنید که وجود ندارد، خطای 404 با مضمون "Error: Not Found" بر روی صفحه نمایش داده می‌شود و در صورتی که به جای عدد، عبارتی غیر عددی وارد کنید، خطای 400 با مضمون: "Error: Bad Request" نمایش داده می‌شود. در Web API، تمامی پاسخ‌ها باید در قالب کدهای وضعیت HTTP باشند (شکل ذیل). این یکی از اصول اساسی کار با وب سرویس‌ها است. وفادار ماندن به مفاهیم پایه‌ی وب، دید بهتری در مورد اتفاقاتی که می‌افتد به شما می‌دهد.

Console HTML CSS Script DOM Net				
Clear Persist All HTML CSS JS XHR Images Flash Media				
URL	Status	Domain	Size	Remote IP
+ GET 2344	404 Not Found	localhost:3032	0	127.0.0.1:3032
+ GET test	400 Bad Request	localhost:3032	321 B	127.0.0.1:3032
2 requests			321 B	

در قسمت بعد با مفهوم مسیریابی در ASP.NET Web API آشنا می‌شوید.

نظرات خوانندگان

نویسنده: Nima
تاریخ: ۲۰:۰۶ ۱۳۹۱/۰۴/۱۶

سلام آقای راد

ممنون از مطلب مفیدتون..سوالی از حضورتون داشتم ما در وب سرویسهای asmx میتونستیم از sessionها استفاده کنیم تا مثلاً اگر میخواستیم از طریق jquery بخواهیم اون وب سرویس رو صدا کنیم این کار فقط برای کاربرانی که در سیستم وارد شده اند امکان پذیر باشد. از لحاظ ملاحظات امنیتی و استفاده از session آیا در قسمتهای بعدی بحث میکنید؟

با تشکر

نویسنده: بهروز راد
تاریخ: ۷:۴۶ ۱۳۹۱/۰۴/۱۷

در Web API در حالت پیش فرض نمی‌تونید از Session استفاده کنید. اصولاً REST اصطلاحاً Stateless هست، اما اگر اصرار به استفاده از Session دارید، باید یک Route Handler سفارشی ایجاد و اینترفیس IRequiresSessionState رو پیاده سازی کنید. سپس پیاده سازی جدید رو به عنوان Route Handler برای route مختص Web API تعریف کنید. در مورد تصدیق هویت، معمولاً به این شکل عمل میشه که یک فیلتر Authorize سفارشی ایجاد و نام کاربری و کلمه‌ی عبور از طریق یک Header سفارشی به Server ارسال میشه. Web API به خوبی با مفهوم فیلترها در ASP.NET MVC هماهنگ هست. سعی می‌کنم در مطلب جدایی به این موارد بپردازم.

نویسنده: Nima
تاریخ: ۹:۵۷ ۱۳۹۱/۰۴/۱۷

با تشکر از شما آقای راد اگر این زحمت رو بکشین ممنون میشم. دوستن مسائل امنیتی باعث استفاده بهتر از مواردی که شما فرمودین میشه. موفق باشید

نویسنده: مهران کلانتری
تاریخ: ۱۸:۵۴ ۱۳۹۱/۰۴/۱۷

سلام آقای راد خیلی خوب و سلیس توضیح میدید

در رابطه با مسائل امنیتی در این روش خیلی خوب می‌شه اگر توضیحی ارائه بدید.

متشکرم

نویسنده: شهرز جعفری
تاریخ: ۲۱:۲۵ ۱۳۹۱/۰۴/۱۸

در Rest قابلیت بنام Syndication Feed Formatter وجود دارد در Web API چطور؟

نویسنده: بهروز راد
تاریخ: ۱۰:۳۲ ۱۳۹۱/۰۴/۱۹

در Web API هم این قابلیت وجود داره.

آشنایی با مفهوم مسیریابی در Web API

در این قسمت با نحوه‌ی تناظر آدرس‌ها توسط Web API به متدهای موجود در Controller آشنا می‌شوید. در هر درخواستی که ارسال می‌شود، Web API، انتخاب مناسب را با رجوع به جدولی با نام جدول مسیرها انجام می‌دهد. زمانی که یک پروژه‌ی جدید با استفاده از ASP.NET MVC 4 ایجاد می‌کنید، یک route پیش فرض به صورت ذیل در متد RegisterRoutes قرار می‌گیرد.

```
routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

عبارت api، ثابت است و قسمت‌های {controller} و {id} توسط آدرس مقداردهی می‌شوند. زمانی که آدرسی با این الگو تطبیق داشته باشد، کارهای ذیل انجام می‌گیرد:

- {controller} به نام Controller تناظر پیدا می‌کند.
- نوع درخواست ارسالی (GET, POST, PUT, DELETE) به نام متد تناظر پیدا می‌کند.
- اگر قسمت {id} در آدرس وجود داشته باشد، به پارامتر id متد انتخاب شده پاس داده می‌شود.
- اگر آدرس دارای Query String باشد، به پارامترهای همنام خود در متد، تناظر پیدا می‌کنند.

در ذیل، مثال هایی را از چند آدرس درخواستی و نتیجه‌ی حاصل از فراخوانی آنها مشاهده می‌کنید.

آدرس /api/products با نوع درخواست GET به متد GetAllProducts()

آدرس /api/products/1 با نوع درخواست GET به متد GetProductById(1)

آدرس /api/products?category=hardware با نوع درخواست GET به متد GetProductByCategory("hardware")

در آدرس اول، عبارت "products" به ProductsController تطبیق پیدا می‌کند. درخواست نیز از نوع GET است، بنابراین Web API به دنبال متدی در Controller می‌گردد که نام آن با عبارت GET "آغاز" شده باشد. همچنین، آدرس شامل قسمت {id} نیز نیست. بنابراین، Web API متدی را انتخاب می‌کند که پارامتر ورودی ندارد. متد GetAllProducts در ProductsController، تمامی این شروط را دارد، پس انتخاب می‌شود.

در دومین آدرس، همان حالت قبل وجود دارد، با این تفاوت که در آدرس درخواستی، قسمت {id} وجود دارد. از آنجا که نوع قسمت {id} در متد GetProductById، int تعریف شده است، باید یک عدد صحیح بعد از آدرس /api/products/ وجود داشته باشد تا متد GetProductById فراخوانی شود. این عدد به طور خودکار به نوع int تبدیل شده و در پارامتر اول متد GetProductById قرار می‌گیرد. در ذیل، برخی آدرس‌ها را ملاحظه می‌کنید که معتبر نیستند و باعث بروز خطا می‌شوند.

آدرس /api/products با نوع درخواست POST، باعث خطای 405Method Not Allowed می‌شود.

آدرس /api/users با نوع درخواست GET، باعث خطای 404Not Found می‌شود.

آدرس /api/products/abc با نوع درخواست GET، باعث خطای 400Bad Request می‌شود.

در آدرس اول، Client یک درخواست از نوع POST ارسال کرده است. Web API به دنبال متدی می‌گردد که نام آن با عبارت Post آغاز می‌شود. اما متدی با این شرط در ProductsController وجود ندارد. بنابراین، پاسخی که دریافت می‌شود، عبارت "405 Method Not Allowed" است. درخواست برای آدرس /api/users/ نیز معتبر نیست، چون Controllerی با نام UsersController وجود ندارد. و سومین آدرس نیز بدین دلیل نامعتبر است که قسمت abc نمی‌تواند به یک عدد صحیح تبدیل شود.

مشاهده‌ی درخواست ارسالی و پاسخ دریافتی

زمانی که با یک وب سرویس کار می‌کنید، مشاهده‌ی محتویات درخواست ارسالی و پاسخ دریافتی می‌تواند کاربرد زیادی در درک نحوه‌ی تعامل بین Client و وب سرویس و کشف خطاهای احتمالی داشته باشد. در Firefox با استفاده از افزونه‌ی Firebug و در Internet Explorer 9 به بالا با ابزار Developer Tools آن می‌توان درخواست‌ها و پاسخ‌ها را مشاهده کرد. در Internet Explorer، کلید F12 را برای اجرای ابزار Developer Tools فشار دهید. از قسمت Network بر روی دکمه‌ی Start Capturing کلیک کنید. حال کلید F5 را برای بارگذاری مجدد صفحه فشار دهید. Internet Explorer، درخواست و پاسخ رد و بدل شده بین مرورگر و Web Server را مانیتور کرده و گزارشی را نشان می‌دهد (شکل ذیل).

F12

File Find Disable View Images Cache Tools Validate | Browser Mode: IE9

HTML CSS Console Script Profiler **Network**

Stop capturing Go to detailed view

URL	Met...	Result	Type	Received	Taken
http://localhost:26481/	GET	200	text/html	1.66 KB	78 ms
/Content/css?v=ji3nO1pdg6VLv3CV...	GET	200	text/css	0.89 KB	47 ms
/Content/themes/base/css?v=UM62...	GET	200	text/css	24.73 KB	63 ms
/Scripts/js?v=4h5lPNUsLiFoa0vqrItj...	GET	200	text/javascript	325.88...	62 ms
/Scripts/jquery-1.6.2.min.js	GET	304	application/x-javascript	176 B	47 ms
/api/products/	GET	200	application/json	398 B	281 ms

از ستون URL، آدرس /api/products/ را انتخاب و بر روی دکمه‌ی Go to detailed view کلیک کنید. در قسمتی که باز می‌شود، گزینه‌هایی برای مشاهده‌ی هدرهای درخواست، پاسخ و همچنین بدنه‌ی هر یک وجود دارد. به عنوان مثال، اگر قسمت Request headers را انتخاب کنید، خواهید دید که Internet Explorer از طریق هدر Accept، تقاضای پاسخ در قالب JSON را کرده است (شکل ذیل).

HTML CSS Console Script Profiler Network	
   Stop capturing Back to summary view < Prev	
URL: http://localhost:26481/api/products/	
Request headers Request body Response headers Response body Cookies Initiator Time	
Key	Value
Request	GET /api/products/ HTTP/1.1
X-Requested-With	XMLHttpRequest
Accept	application/json, text/javascript, */*; q=0.01
Referer	http://localhost:26481/
Accept-Language	en-us
Accept-Encoding	gzip, deflate
User-Agent	Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0)
Host	localhost:26481
Connection	Keep-Alive

اگر قسمت Response body را انتخاب کنید، پاسخ دریافت شده در قالب JSON را خواهید دید.

در قسمت بعد، با مدیریت کدهای وضعیت HTTP برای اعمال چهارگانه‌ی CRUD آشنا می‌شوید.

نظرات خوانندگان

نویسنده: نیما

تاریخ: ۱۳۹۱/۰۴/۱۹ ۱۲:۱۲

سلام آقای راد

با تشکر از زحمتی که میکشید. فرمودید که :

"بنابراین web api به دنبال متدی در controller می‌گردد که نام آن با عبارت "get" آغاز شده باشد. "

آیا این کار باعث عدم دقت و ایجاد خطاهای ناخواسته نمیشه؟ این فقط متدی با get شروع بشه شاید برای من که خیلی کم mvc کار کردم یکم مشکل دار به نظر برسه. اگر ما دو متد داشته باشیم که در ابتدای آنها get باشد آیا برنامه خطا میگیرد؟ ممنون میشم یکم در این باره توضیح بدین

نویسنده: بهروز راد

تاریخ: ۱۳۹۱/۰۴/۱۹ ۱۲:۲۶

شما محدود به رفتار پیش فرض Web API نیستید. می‌تونید route رو تغییر بدید و نام Action رو هم در اون ذکر کنید.

```
routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{action}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

و در ProductsController داشته باشید:

```
[HttpGet]
public string Details(int id)
{
    // do something
}
```

حال درخواستی برای /api/products/details/1 باعث اجرای متد Details میشه. یا حتی می‌تونید route رو تغییر ندید و فقط از [HttpGet] و [HttpPost] و امثال اونها برای تعیین فعل استفاده کنید. به عنوان مثال، اگر route پیش فرض رو تغییر ندید و متد Details رو به شکل قبل داشته باشید، آدرسی مانند /api/products/1 با نوع GET باعث میشه تا متد Details اجرا بشه.

نویسنده: نیما

تاریخ: ۱۳۹۱/۰۴/۱۹ ۱۲:۲۷

بسیار ممنونم خیلی مفید بود

نویسنده: آریا

تاریخ: ۱۳۹۱/۰۴/۱۹ ۱۹:۰۱

خیلی عالی بود. متشکر

نویسنده: رضا ب.
تاریخ: ۱۳۹۱/۰۴/۲۱ ۱۰:۳۰

یه سوال که ربط چندانی به این پست نداره؛
asp.net web api رو میتونیم یه لایه abstraction حساب کنیم که در اون منطق سیستم (BL) وجود داره و بنابراین از آن در سطح انتزاعی بالاتری در سیستم یا سیستم‌های مشابه استفاده میشن؟ (تاکید سوالم آنجاست که میزان عملکرد موثر asp.net web api تا کجاست؟) ممنون.

نویسنده: بهروز راد
تاریخ: ۱۳۹۱/۰۴/۲۱ ۱۹:۳۸

شما در سواتون می‌تونید عبارت "ASP.NET Web API" رو با "Web Service تحت HTTP" جایگزین کنید. در Web Service هم منطق سیستم وجود داره، مثلاً محاسبه‌ی نرخ تورم در یک بازه‌ی زمانی با توجه به 30 قلم کالای اساسی. عملکرد Web API، همان عملکردی است که از یک Web Service تحت HTTP مانند ASMX انتظار دارید.

نویسنده: حمزه ء
تاریخ: ۱۳۹۳/۰۲/۲۰ ۱۲:۲۶

در قسمت سوم آموزش این مثال رو داشتیم :

```
$.getJSON("api/products/"+id,
    function (data) {
        var str = data.Name + ' ' + data.Price;
        $('#products').empty();
        $('#products').html(str);
    }
);
```

خب تا اینجا api/products/id اجرا میشه .
فرض کنید چند جستجو داریم و نیاز داریم برای هر کدوم اکشن متناظر با اون اجرا بشه برای مثال:
api/products/id
api/products/details/id
حالا چطور میتونم برای دو دکمه تعیین کنم ، با زدن هر کدوم چه تابعی اجرا بشه ؟
بهتر بگم چطور details رو برای یک دکمه به آدرس اضافه کنم ؟

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۰۲/۲۰ ۱۳:۲

از [متد click](#) استفاده کنید. داخل callback آن درخواست Ajax ایی را ارسال کنید به سرور.

نویسنده: حمزه ء
تاریخ: ۱۳۹۳/۰۲/۲۰ ۱۸:۲

ممنونم .
1- کدها رو رویداد کلیک نوشتن و اجرا شد . ولی توی آدرس بار مرورگر هیچ تغییری بوجود نیومد ؟ چطور میتونم زمانی که یک متد رو از web api فراخوانی کردم ، همزمان آدرس بار مرورگر هم تغییر کنه ؟
2- برای اینکه فقط یوزرهای سایت و آنلاین شده یا role های خاص بتونن از اون متد استفاده کنن ، attribute رو بالای اون اضافه کردم ، آیا درسته ؟

```
[Authorize(Roles="Admin")]
//[Authorize(Users="")]
```

```
public Product GetProductById(int Id)
{
    var product = Products.FirstOrDefault(p => p.Id == Id);
    if(product==null)
    {
        throw new HttpResponseException(HttpStatusCode.NotFound);
    }
    return product;
}
```

نویسنده:

محسن خان

تاریخ:

۱۸:۵۸ ۱۳۹۳/۰۲/۲۰

این ASP.NET MVC نیست. ASP.NET Web API است. می‌تونی دستی آدرس خاصی رو در مرورگر وارد کنی و نهایتاً مثلاً خروجی JSON یا XML بگیری (شاید بهتر باشه یکبار اینکار رو انجام بدی تا حس بهتری نسبت به این فناوری پیدا کنی که کارش چی هست. خروجی‌اش چی هست). در کل هدفش این نیست که خروجی HTML به شما بده. هدفش تامین داده برای کلاینت‌ها هست. سمت کلاینت رو آزاد هستی هر طور که دوست داشتی کار کنی. مثلاً یک صفحه‌ی HTML درست کنی و اطلاعات Web API رو بگیری و نمایش بدی.

مدیریت کدهای وضعیت در Web API

تمامی پاسخ‌های دریافتی از Web API توسط Client، باید در قالب کدهای وضعیت HTTP باشند. دو کلاس جدید با نام‌های `HttpResponseMessage` و `HttpResponseException` همراه با ASP.NET MVC 4 معرفی شده‌اند که ارسال کدهای وضعیت پردازش درخواست به Client را آسان می‌سازند. به عنوان مثال، ارسال وضعیت برای چهار عمل اصلی بازایی، ایجاد، آپدیت و حذف رکورد را بررسی می‌کنیم.

بازایی رکورد

بر اساس مستندات پروتکل HTTP، در صورتی که منبع درخواستی Client پیدا نشد، باید کد وضعیت 404 برگشت داده شود. این حالت را در متد ذیل پیاده سازی کرده ایم.

```
public Product GetProduct(int id)
{
    Product item = repository.Get(id);
    if (item == null)
    {
        throw new HttpResponseException(new HttpResponseMessage(HttpStatusCode.NotFound));
    }
    return item;
}
```

در صورتی که رکوردی با مشخصه‌ی درخواستی پیدا نشد، با استفاده از کلاس `HttpResponseException`، خطایی به Client ارسال خواهد شد. پارامتر سازنده‌ی این کلاس، شی‌ای از نوع کلاس `HttpResponseMessage` است. سازنده‌ی کلاس `HttpResponseMessage`، مقداری از یک enum با نام `HttpStatusCode` را می‌پذیرد. مقدار `NotFound`، نشان از خطای 404 است و زمانی به کار می‌رود که منبع درخواستی وجود نداشته باشد. اگر محصول درخواست شده یافت شد، در قالب JSON برگشت داده می‌شود. در شکل ذیل، پاسخ دریافتی در زمان درخواست محصولی که وجود ندارد را ملاحظه می‌کنید.

Console HTML CSS Script DOM Net Cookies				
<div> <div> <div>Clear</div> <div>Persist</div> <div>All</div> </div> <div>HTML CSS JS XHR Images Flash Media</div> </div>				
URL	Status	Domain	Size	Remote IP
+ GET 8	404 Not Found	localhost:2239	0	127.0.0.1:2239
1 request			0	

ایجاد رکورد

برای ایجاد رکورد، Client درخواستی از نوع POST را همراه با داده‌های رکورد در بدنه‌ی درخواست به Server ارسال می‌کند. در ذیل، پیاده سازی ساده‌ای از این حالت را مشاهده می‌کنید.

```
public Product PostProduct(Product item)
{
    item = repository.Add(item);
    return item;
}
```

این پیاده سازی کار می‌کند اما کمبودهایی دارد:

کد وضعیت پردازش درخواست : به طور پیش فرض، Web API، کد 200 را در پاسخ ارسال می‌کند، اما بر اساس مستندات پروتکل HTTP، زمانی که یک درخواست از نوع POST منجر به تولید منبعی می‌شود، Server باید کد وضعیت 201 را به Client برگشت بدهد.

آدرس منبع جدید ایجاد شده : بر اساس مستندات پروتکل HTTP، زمانی که منبعی بر روی Server ایجاد می‌شود، باید آدرس منبع جدید ایجاد شده از طریق هدر Location به Client ارسال شود. با توجه به این توضیحات، متد قبل به صورت ذیل در خواهد آمد.

```
public HttpResponseMessage PostProduct(Product item)
{
    item = repository.Add(item);
    var response = Request.CreateResponse(HttpStatusCode.Created, item);

    string uri = Url.Link("DefaultApi", new { id = item.Id });
    response.Headers.Location = new Uri(uri);
    return response;
}
```

همان طور که ملاحظه می‌کنید، خروجی متد از نوع کلاس HttpResponseMessage است، چون با استفاده از این نوع می‌توانیم جزئیات مورد نیاز را در مورد نتیجه‌ی پردازش درخواست به مرورگر ارسال کنیم. همچنین، داده‌های رکورد جدید نیز در بدنه‌ی پاسخ، با یک فرمت مناسب مانند XML یا JSON برگشت داده می‌شوند. با استفاده از متد CreateResponse کلاس Request و پاس دادن کد وضعیت و شی‌ای که قصد داریم به Client ارسال شود به این متد، شی‌ای از نوع کلاس HttpResponseMessage ایجاد می‌کنیم. آدرس منبع جدید نیز با استفاده از response.Headers.Location مشخص شده است. نمونه‌ای از پاسخ دریافت شده در سمت Client به صورت ذیل است.





آپدیت رکورد

آپدیت با استفاده از درخواست‌های از نوع PUT انجام می‌گیرد. یک مثال ساده در این مورد.

```
public void PutProduct(int id, Product product)
{
    product.Id = id;
    if (!repository.Update(product))
    {
        throw new HttpResponseException(new HttpResponseMessage(HttpStatusCode.NotFound));
    }
}
```

نام متد با عبارت Put آغاز شده است. بنابراین توسط Web API برای پردازش درخواست‌های از نوع PUT در نظر گرفته می‌شود. متد قبل، دو پارامتر ورودی دارد. id برای مشخصه‌ی محصول، و محصول آپدیت شده که در پارامتر دوم قرار می‌گیرد. مقدار پارامتر id از آدرس دریافت می‌شود و مقدار پارامتر product از بدنه‌ی درخواست. به طور پیش فرض، Web API، مقدار داده‌هایی با نوع ساده مانند string، int و bool را از طریق route، و مقدار نوع‌های پیچیده‌تر مانند داده‌های یک کلاس را از بدنه‌ی درخواست می‌خواند.

حذف یک رکورد

حذف یک رکورد، با استفاده از درخواست‌های از نوع DELETE انجام می‌گیرد. یک مثال ساده در این مورد.

```
public HttpResponseMessage DeleteProduct(int id)
{
    repository.Remove(id);
    return new HttpResponseMessage(HttpStatusCode.NoContent);
}
```

بر اساس مستندات پروتکل HTTP، اگر منبعی که Client قصد حذف آن را دارد از پیش حذف شده است، نباید خطایی به وی گزارش شود. معمولاً در متدهایی که وظیفه‌ی حذف منبع را بر عهده دارند، کد 204 مبنی بر پردازش کامل درخواست و پاسخ خالی برگشت داده می‌شود. این کد با استفاده از مقدار NoContent برای HttpStatusCode مشخص می‌شود.

فراخوانی متدها و مدیریت کدهای وضعیت HTTP در سمت Client

حال ببینیم چگونه می‌توان از متدهای قبل در سمت Client استفاده و خطاهای احتمالی آنها را مدیریت کرد. بهتر است مثال را برای حالتی که در آن رکوردی آپدیت می‌شود بررسی کنیم. کدهای مورد نیاز برای فراخوانی متد PutProduct در سمت Client به صورت ذیل است.

```
var id = $("#myTextBox").val();

$.ajax({
    url: "/api/Test/" + id,
    type: 'PUT',
    data: { Id: "1", Name: "Tomato Soup", Category: "Groceries", Price: "1.39M" },
    cache: false,
    statusCode: {
        200: function (data) {
            alert("آپدیت انجام شد");
        },
        404: function () {
            alert("خطا در آپدیت");
        }
    }
});
```

از متدهای get, getJson یا post در jQuery نمی‌توان برای عمل آپدیت استفاده نمود، چون Web API انتظار دارد تا نام فعل درخواستی، PUT باشد. اما با استفاده از متد ajax و ذکر نام فعل در پارامتر type آن می‌توان نوع درخواست را PUT تعریف کرد. خط 5 بدین منظور است. از طریق خصیصه‌ی statusCode نیز می‌توان کدهای وضعیت مختلف HTTP را بررسی کرد. دو کد 200 و 404 که به ترتیب نشان از موفقیت و عدم موفقیت در آپدیت رکورد هستند تعریف شده و پیغام مناسب به کاربر نمایش داده می‌شود. در حالتی که آپدیت با موفقیت همراه باشد، بدنه‌ی پاسخ به شکل ذیل است.



و در صورتی که خطایی رخ دهد، بدنه‌ی پاسخ دریافتی به صورت ذیل خواهد بود.

The screenshot shows a web browser's developer console with the 'Net' tab selected. A PUT request is visible, labeled 'PUT 2'. The request headers show 'Content-Type: application/x-www-form-urlencoded'. The request body contains the following parameters:

Parameter	Value
Category	Groceries
Id	1
Name	Tomato Soup
Price	1.39M

The response status is '404 Not Found'. The source of the request is 'localhost:2239'. The console also shows the request URL: 'Id=1&Name=Tomato+Soup&Category=Groceries&Price=1.39M'.

نظرات خوانندگان

نویسنده: prncedotnet
تاریخ: ۱۳۹۱/۰۴/۳۱ ۰:۳۰

سلام جناب راد

2 تا سوال داشتم :

- 1.چطور می‌تونم اطلاعات گرفته شده از WebAPI رو توسط JSON.NET در یک پروژه سیلورلایت Deserialize کنم؟
 - 2.چطور مدل هایی که در اون از روابط many to one - many to many یا... در Entity استفاده شده رو از یک WebAPI بگیرم؟
- ممنون

نویسنده: آزاده
تاریخ: ۱۳۹۲/۰۶/۱۹ ۱۱:۴۶

سلام، با تشکر؛ من در صورتی که بخواهم کاری کنم که کاربر فقط از توی فرم و از طریق jqueryی نوشته شده بتونه به اطلاعات دسترسی داشته باشد، یعنی در صورتی که از آدرس بار بروزر استفاده کرد، خروجی رو نگیرد چیکار باید بکنم؟

ممنون

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۶/۱۹ ۱۲:۸

از محدودیت POST استفاده کنید بجای GET.

نویسنده: آزاده
تاریخ: ۱۳۹۲/۰۶/۱۹ ۱۳:۱۲

سلام . ممنون از راهنماییتون.
یعنی همون متدی که دارم رو فقط به نوع Post تغییر بدم کافیه. و از اون به بعد از آدرس بار نمی‌شه بهش دسترسی داشت.
احتیاجی به تنظیمات خاصی نداره دیگه؟

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۶/۱۹ ۱۳:۳۴

کار معمولی با یک آدرس در مرورگر یعنی حالت Get. میشه این رو تغییر داد به Post که با بازکردن ساده آدرس در مرورگر کار نکنه.

با آمدن [Asp.Net Web API](#) کار ساختن [Web API](#) ها برای برنامه نویسی‌ها به خصوص دسته ای که با ساخت [API](#) و وب سرویس آشنا نبودند خیلی ساده‌تر شد. اگر با [Asp.Net MVC](#) آشنا باشید خیلی سریع می‌توانید اولین [Web Service](#) خودتان را بسازید.

در صفحه مربوط به [Asp.Net Web API](#) آمده است که این فریمورک بستر مناسبی برای ساخت و توسعه برنامه های [RESTful](#) است. اما تنها ساختن کنترلر و اکشن و برگشت دادن داده‌ها به سمت کلاینت، به خودی خود برنامه شما رو تبدیل به یک [RESTful API](#) نمی‌کند.

مثل تمام مفاهیم و ابزارها، طراحی و ساختن [RESTful API](#) هم دارای اصول و [Best Practice](#) هایی است که رعایت آنها به خصوص در این زمینه از اهمیت زیادی برخوردار است. همانطور که از تعریف [API](#) برمی‌آید شما در حال طراحی رابطی هستید تا به توسعه دهندگان دیگر امکان دهید از داده‌ها و یا خدمات شما در برنامه‌ها و سرویس‌هایشان استفاده کنند. مانند [API](#) های توئیت و [نقشه گوگل](#) که برنامه‌های زیادی بر مبنای آنها ساخته شده‌اند. در واقع توسعه دهندگان مشتریان [API](#) شما هستند.

بهره وری توسعه دهنده مهمترین اصل

اینطور می‌توان نتیجه گرفت که اولین و مهمترین اصل در طراحی [API](#) باید رضایت و موفقیت توسعه دهنده در درک و یادگیری سریع [API](#) شما، نه تنها با کمترین زحمت بلکه همراه با حس نشاط، باشد. ([تجربه کاربری](#) در اینجا هم می‌تواند صدق کند). سعی کنید در زمان انتخاب از بین روش‌های طراحی موجود، از دیدگاه توسعه دهنده به مسئله نگاه کنید. خود را به جای او قرار دهید و تصور کنید که می‌خواهید با استفاده از [API](#) موجود یک رابط کاربری طراحی کنید یا یک اپلیکیشن برای موبایل بنویسید و اصل را این نکته قرار دهید که بهره وری برنامه نویسی را حداکثر کنید. ممکن است گاهی بین طراحی که بر اساس این اصل برای [API](#) خود در نظر داریم و یکی از اصول یا استانداردها تعارض بوجود بیاید. در این موارد بعد از اینکه مطمئن شدیم این اختلاف ناشی از طراحی و درک اشتباه خودمان نیست (که اکثرا هست) ارجحیت را باید به طراحی بدهیم.

تهیه مستندات API

اگر برای پروژه وب سایتتان هیچ نوشته ای یا توضیحی ندارید، جالب نیست اما خودتان ساختار برنامه خود را می‌شناسید و کار را پیش می‌برید. اما توسعه دهنده ای که از [API](#) شما می‌خواهد استفاده کند و به احتمال زیاد شما را نمی‌شناسد، عضو تیم شما هم نیست، هیچ ایده ای درباره ساختار آن، روش نامگذاری توابع و منابع، ساختار [URL](#) ها، چگونگی و گام‌های پروسه درخواست دریافت پاسخ ندارد، و به مستندات شما وابسته است و تمام اینها باید در مستندات شما باشد. بیشتر توسعه دهندگان قبل از تست کردن [API](#) شما سری به مستندات می‌زنند، دنبال نمونه کد آموزشی می‌گردند و در اینترنت درباره آن جستجو می‌کنند. ازینرو مستندات (کارآمد) یک ضرورت است:

- 1- در مستندات باید هم درباره کلیت و هم در مورد تک تک توابع (پارامترهای معتبر، ساختار پاسخ‌ها و ...) توضیحات وجود داشته باشد.
- 2- باید شامل مثالهایی از سیکل کامل درخواست‌ها / پاسخ‌ها باشند.
- 3- تغییرات اعمال شده نسبت به نسخه‌های قبلی باید در مستندات بیان شوند.
- 4- (در وب) یافتن و جستجو کردن در مستندات که به صورت فایل [Pdf](#) هستند یا برای دسترسی نیاز به [Login](#) داشته باشند سخت و آزاردهنده هستند.
- 5- کسی را داشته باشید تا با و بدون مستندات با [API](#) شما کار کند و از این روش برای تکمیل و اصلاح مستندات استفاده کنید.

رعایت نسخه بندی و حفظ نسخه‌های قبلی به صورت فعال برای مدت معین

یک [API](#) تقریباً هیچوقت کاملاً پایدار نمی‌شود و اعمال تغییرات برای بهبود آن اجتناب ناپذیر هستند. مسئله مهم این است که چطور این تغییرات مدیریت شوند. مستند کردن تغییرات، اعلام به موقع آنها و دادن یک بازه زمانی کافی برای ارتقا یافتن برنامه

هایی که از نسخه‌های قدیمی‌تر استفاده می‌کنند نکات مهمی هستند. همیشه در کنار نسخه بروز و اصلی یک یا دو نسخه (بسته به API و کلاینت‌های آن) قدیمی‌تر را برای زمان مشخصی در حالت سرویس دهی داشته باشید .

داشتن یک روش مناسب برای اعلام تغییرات و ارائه مستندات و البته دریافت بازخورد از استفاده کنندگان

تعامل با کاربران برنامه باید از کانال‌های مختلف وجود داشته باشد. از وبلاگ ، [Google Groups](#) ، Mailing List و دیگر ابزارهایی که در اینترنت وجود دارند برای انتشار مستندات ، اعلام بروزرسانی‌ها ، قرار دادن مقالات و نمونه کدهای آموزشی ، پرسش و پاسخ با کاربران استفاده کنید .

مدیریت خطاها به شکل صحیح که به توسعه دهنده در آزمون برنامه اش کمک کند.

از منظر برنامه نویسی که از API شما استفاده می‌کند هرآنچه در آنسوی API اتفاق می‌افتد یک جعبه سیاه است . به همین جهت خطاهای API شما ابزار کلیدی برای او هستند که خطایابی و اصلاح برنامه در حال توسعه اش را ممکن می‌کنند . علاوه بر این ، زمانی که برنامه نوشته شده با API شما مورد استفاده کاربر نهایی قرار گرفت ، خطاهای به دقت طراحی شده API شما کمک بزرگی برای توسعه دهنده در عیب یابی هستند .

1- از [Status Code](#) های HTTP استفاده کنید و سعی کنید تا حد ممکن آنها را نزدیک به مفهوم استانداردشان بکار ببرید .

2- خطا و علت آن را به زبان روشن توضیح دهید و در توضیح خساست به خرج ندهید .

3- در صورت امکان لینکی به یک صفحه وب که حاوی توضیحات بیشتری است را در خطا بگنجانید .

رعایت ثبات و یکدستی در تمام بخش‌های طراحی که توانایی پیش بینی توسعه دهنده را در استفاده از API افزایش می‌دهد .

داشتن مستندات لازم است اما این بدین معنی نیست که خود API نباید خوانا و قابل پیش بینی باشد . از هر روش و تکنیکی که استفاده می‌کنید آن را در تمام پروژه حفظ کنید . نامگذاری توابع/منابع ، ساختار پاسخ‌ها ، URLها ، نقش و عملیاتی که HTTP methodها در API شما انجام می‌دهند باید ثبات داشته باشند . از این طریق توسعه دهنده لازم نیست برای هر بخشی از API شما به سراغ فایل‌ها راهنما برود . و به سرعت کار خود را به پیش می‌برد .

انعطاف پذیر بودن API

API توسط کلاینت‌های مختلفی و برای افراد مختلفی مورد استفاده قرار می‌گیرد که لزوماً همه‌ی آنها ساختار یکسانی ندارند و API شما باید تا جای ممکن بتواند همه آنها را پوشش دهد . محدود بودن فرمت پاسخ ، ثابت بودن فیلدهای ارسالی به کلاینت ، ندادن امکان صفحه بندی ، مرتب سازی و جستجو در داده‌ها به کلاینت ، داشتن تنها یک نوع احراز هویت ، وابسته بودن به کوکی و ... از مشخصات یک API منجمد و انعطاف ناپذیر هستند .

اینها اصولی کلی بودند که بسیاری از آنها مختص طراحی API نیستند و در تمام حوزه‌ها قابل استفاده بوده ، جز الزامات هستند . در قسمت‌های بعدی نکات اختصاصی‌تری را بررسی خواهیم کرد .

نظرات خوانندگان

نویسنده: وحید

تاریخ: ۱۳۹۲/۱۰/۱۷ ۱:۱۴

با سلام یعنی شما میگویید برای web api میبایست یک لایه جدا تعریف نمود و cors را در آن لحاظ نمود؟

نویسنده: محسن درپرستی

تاریخ: ۱۳۹۲/۱۰/۱۷ ۹:۱۲

لایه جدا از چه چیزی؟ اگر منظورتان در Asp.Net باشد، پروژه هایی که با استفاده از Asp.Net Web API ساخته می شوند خود یک سیستم مستقل هستند.

نویسنده: محسن خان

تاریخ: ۱۳۹۲/۱۰/۲۹ ۲۲:۳۹

سایت silverreader که از asp.net web api استفاده می کنه، نگارش اول کارش با این آدرس شروع میشه

<https://silverreader.com/api/v1>

نویسنده: شهروز جعفری

تاریخ: ۱۳۹۲/۱۲/۰۶ ۷:۳۶

بحث cors یک چیز است و بحث Best Practice هایی برای طراحی RESTful API یک چیز. در کل زمانی که ما می خواهیم یک سرویس ارائه دهیم و باقی از آن استفاده کنند داستانش با یک متد که فقط خودمان از آن استفاده می کنیم فرق می کند. باید به خیلی چیزها حواسمان باشد.

نویسنده: شهروز جعفری

تاریخ: ۱۳۹۲/۱۲/۰۶ ۷:۳۹

من یک سری مطلب درباره نسخه بندی در API پیدا کردم باهاتون به اشتراک میذارم شاید مفید باشه:

<http://www.lexicalscope.com/blog/2012/03/12/how-are-rest-apis-versioned/>

<http://stackoverflow.com/questions/10742594/versioning-rest-api>

ASP.NET Web API 2 به همراه یک سری قابلیت جدید جالب منتشر شده است. در این پست 5 قابلیت برتر از این قابلیت‌های جدید را بررسی می‌کنیم.

1. Attribute Routing

در کنار سیستم routing فعلی، ASP.NET Web API 2 حالا از Attribute Routing هم پشتیبانی می‌کند. در مورد سیستم routing فعلی، می‌توانیم قالب‌های متعددی برای routing بنویسیم. هنگامی که یک درخواست به سرور میرسد، کنترلر مناسب انتخاب شده و اکشن متد مناسب فراخوانی می‌شود. در لیست زیر قالب پیش فرض routing در Web API را مشاهده می‌کنید.

```
Config.Routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{Controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

این رویکرد routing مزایای خود را دارد. از جمله اینکه تمام مسیرها در یک مکان واحد تعریف می‌شوند، اما تنها برای الگوهای مشخص. مثلاً پشتیبانی از nested routing روی یک کنترلر مشکل می‌شود. در ASP.NET Web API 2 به سادگی می‌توانیم الگوی URI ذکر شده را پشتیبانی کنیم. لیست زیر نمونه ای از یک الگوی URI با AttributeRouting را نشان می‌دهد.

URI Pattern --> books/1/authors

```
[Route("books/{bookId}/authors")]
public IEnumerable<Author> GetAuthorByBook(int bookId) { ..... }
```

2. CORS - Cross Origin Resource Sharing

بصورت نرمال، مرورگرها اجازه درخواست‌های cross-domain را نمی‌دهند، که بخاطر same-origin policy است. خوب، CORS (Cross Origin Resource Sharing) چیست؟ CORS یک مکانیزم است که به صفحات وب این را اجازه می‌دهد تا یک درخواست آژاکسی (Ajax Request) به دامنه ای دیگر ارسال کنند. دامنه ای به غیر از دامنه ای که صفحه وب را رندر کرده است. CORS با استانداردهای W3C سازگار است و حالا ASP.NET Web API در نسخه 2 خود از آن پشتیبانی می‌کند.

3. OWIN (Open Web Interface for .NET) self-hosting

ASP.NET Web API 2 به همراه یک پکیج عرضه می‌شود، که *Microsoft.AspNet.WebApi.OwinSelfHost* نام دارد.

طبق گفته وب سایت <http://owin.org>:

OWIN یک اینترفیس استاندارد بین سرورهای دات نت و اپلیکیشن‌های وب تعریف می‌کند. هدف این اینترفیس جداسازی (decoupling) سرور و اپلیکیشن است. تشویق به توسعه ماژول‌های ساده برای توسعه اپلیکیشن‌های وب دات نت. و بعنوان یک استاندارد باز (open standard) اکوسیستم نرم افزارهای متن باز را تحریک کند تا ابزار توسعه اپلیکیشن‌های وب دات نت توسعه یابند.

بنابراین طبق گفته‌های بالا، OWIN گزینه ای ایده آل برای میزبانی اپلیکیشن‌های وب روی پروسس‌هایی به غیر از پروسس IIS است. پیاده سازی‌های دیگری از OWIN نیز وجود دارند، مانند Giacomo, Kayak, Firefly و غیره. اما *Katana* گزینه توصیه شده برای سرورهای مایکروسافت و فریم ورک‌های Web API است.

4. IHttpActionResult

در کنار دو روش موجود فعلی برای ساختن response اکشن متدها در کنترلر ها، ASP.NET Web API 2 حالا از مدل جدیدی هم

پشتیبانی می‌کند. *IHttpResponseMessage* یک اینترفیس است که بعنوان یک فاکتوری (factory) برای *HttpResponseMessage* کار می‌کند. این روش بسیار قدرتمند است بدلیل اینکه web api را گسترش می‌دهد. با استفاده از این رویکرد، می‌توانیم response هایی با هر نوع دلخواه بسازیم. برای اطلاعات بیشتر به [how to serve HTML with IHttpActionResult](#) مراجعه کنید.

5. Web API OData

پروتکل OData (Open Data Protocol) در واقع یک پروتکل وب برای کوئری گرفتن و بروز رسانی داده‌ها است. ASP.NET Web API 2 پشتیبانی از *\$expand*, *\$select* و *\$value* را اضافه کرده است. با استفاده از این امکانات، می‌توانیم نحوه معرفی پاسخ سرور را کنترل کنیم، یعنی representation دریافتی از سرور را می‌توانید سفارشی کنید. **\$expand**: بصورت نرمال، هنگام کوئری گرفتن از یک کالکشن OData، پاسخ سرور موجودیت‌های مرتبط (related entities) را شامل نمی‌شود. با استفاده از *\$expand* می‌توانیم موجودیت‌های مرتبط را بصورت inline در پاسخ سرور دریافت کنیم. **\$select**: از این متد برای انتخاب چند خاصیت بخصوص از پاسخ سرور استفاده می‌شود، بجای آنکه تمام خاصیت‌ها بارگذاری شوند. **\$value**: با این متد مقدار خام (raw) فیلدها را بدست می‌آورید، بجای دریافت آنها در فرمت OData.

چند مقاله خوب دیگر

[Top 10 ASP.NET MVC Interview Questions](#)

[jQuery code snippets every web developer must have 7](#)

[WCF Vs ASMX Web Services](#)

[Top 10 HTML5 Interview Questions](#)

[JavaScript : Validating Letters and Numbers](#)

گرچه ASP.NET Web API به همراه ASP.NET MVC بسته بندی شده و استفاده می‌شود، اما اضافه کردن آن به اپلیکیشن‌های ASP.NET Web Forms کار ساده ای است. در این مقاله مراحل لازم را بررسی می‌کنیم.

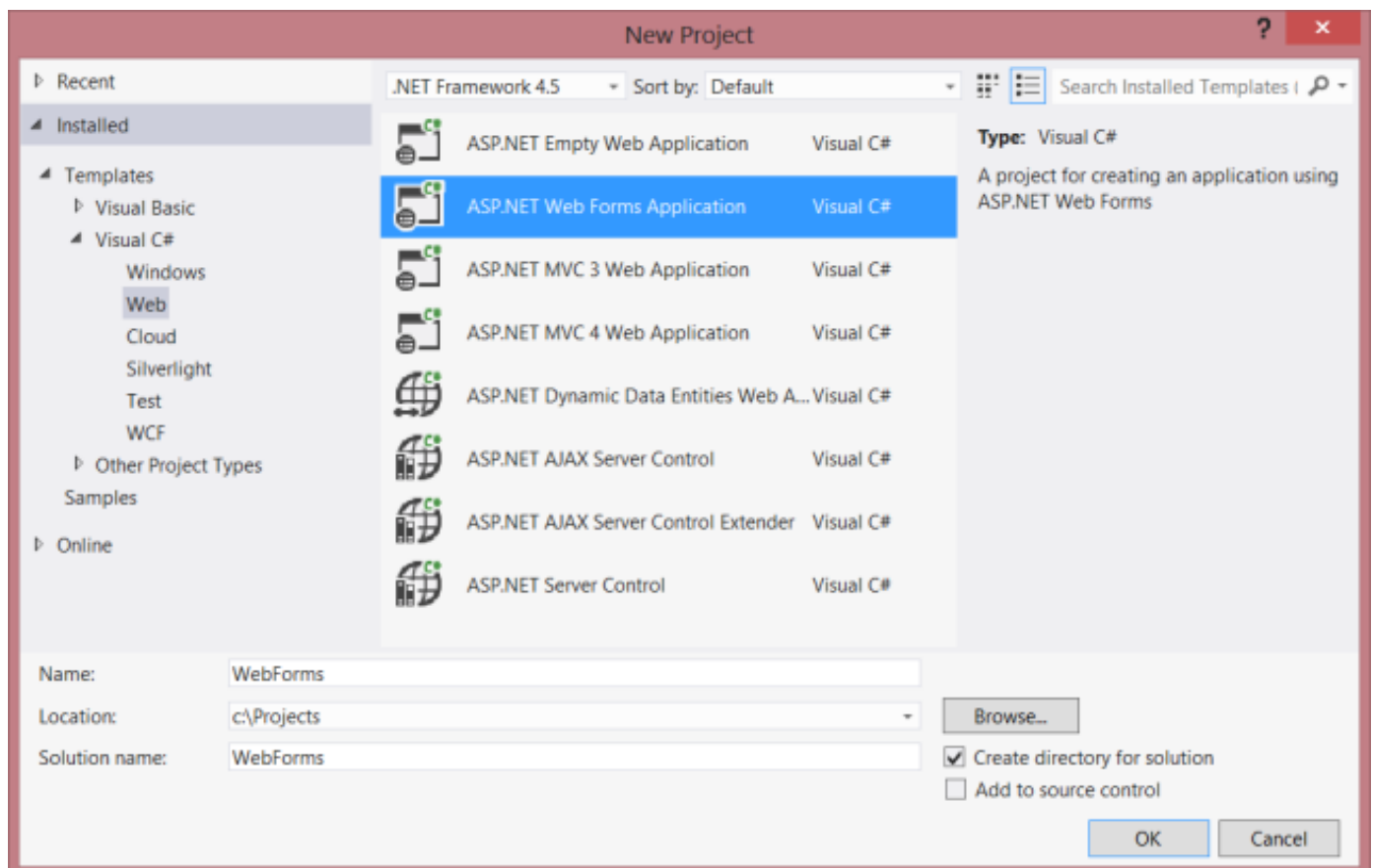
برای استفاده از Web API در یک اپلیکیشن ASP.NET Web Forms دو قدم اصلی باید برداشته شود:

اضافه کردن یک کنترلر Web API که از کلاس **ApiController** مشتق می‌شود.

اضافه کردن مسیرهای جدید به متد **Application_Start**.

یک پروژه Web Forms بسازید

ویژوال استودیو را اجرا کنید و پروژه جدیدی از نوع ASP.NET Web Forms Application ایجاد کنید.



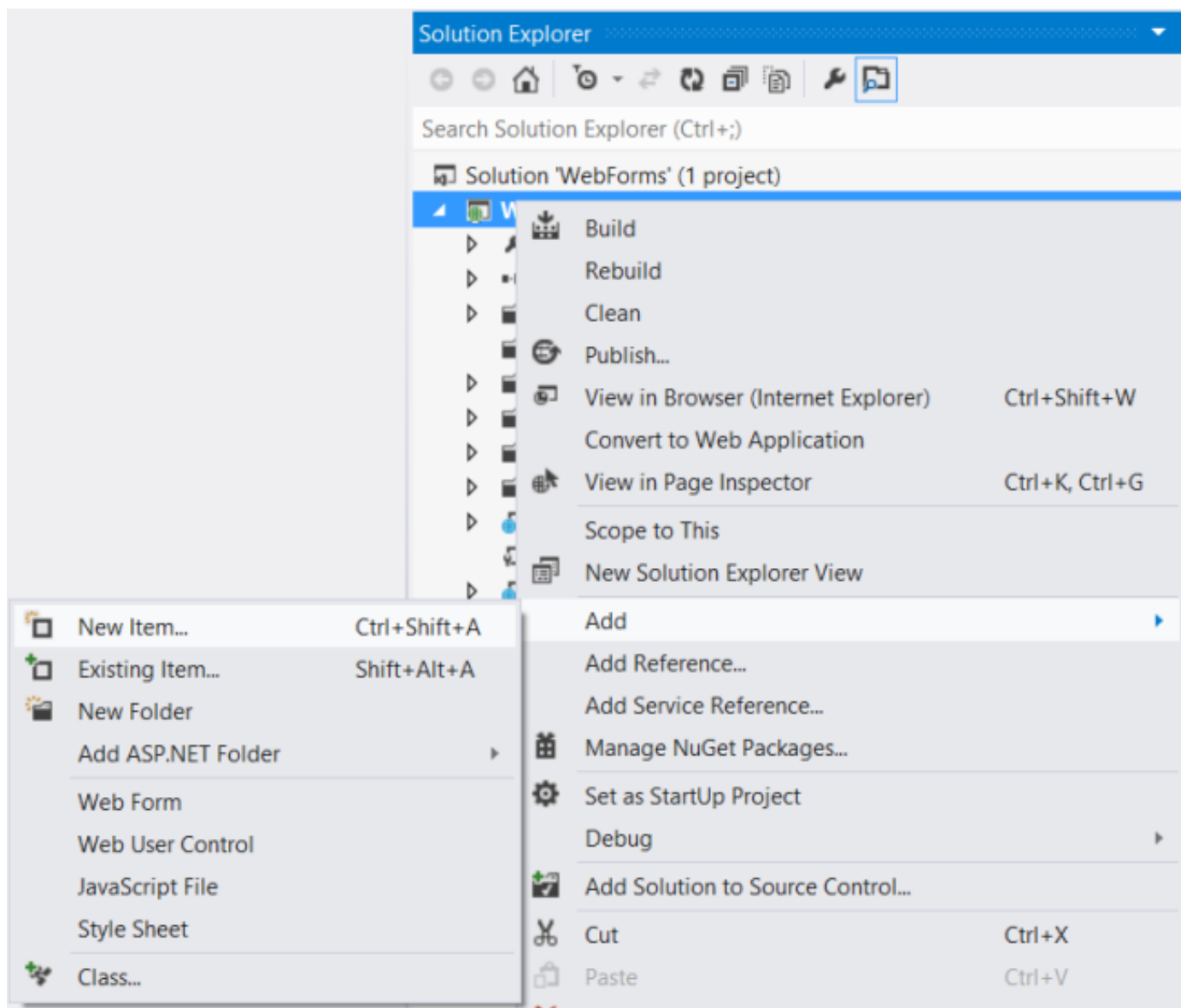
کنترلر و مدل اپلیکیشن را ایجاد کنید

کلاس جدیدی با نام Product بسازید و خواص زیر را به آن اضافه کنید.

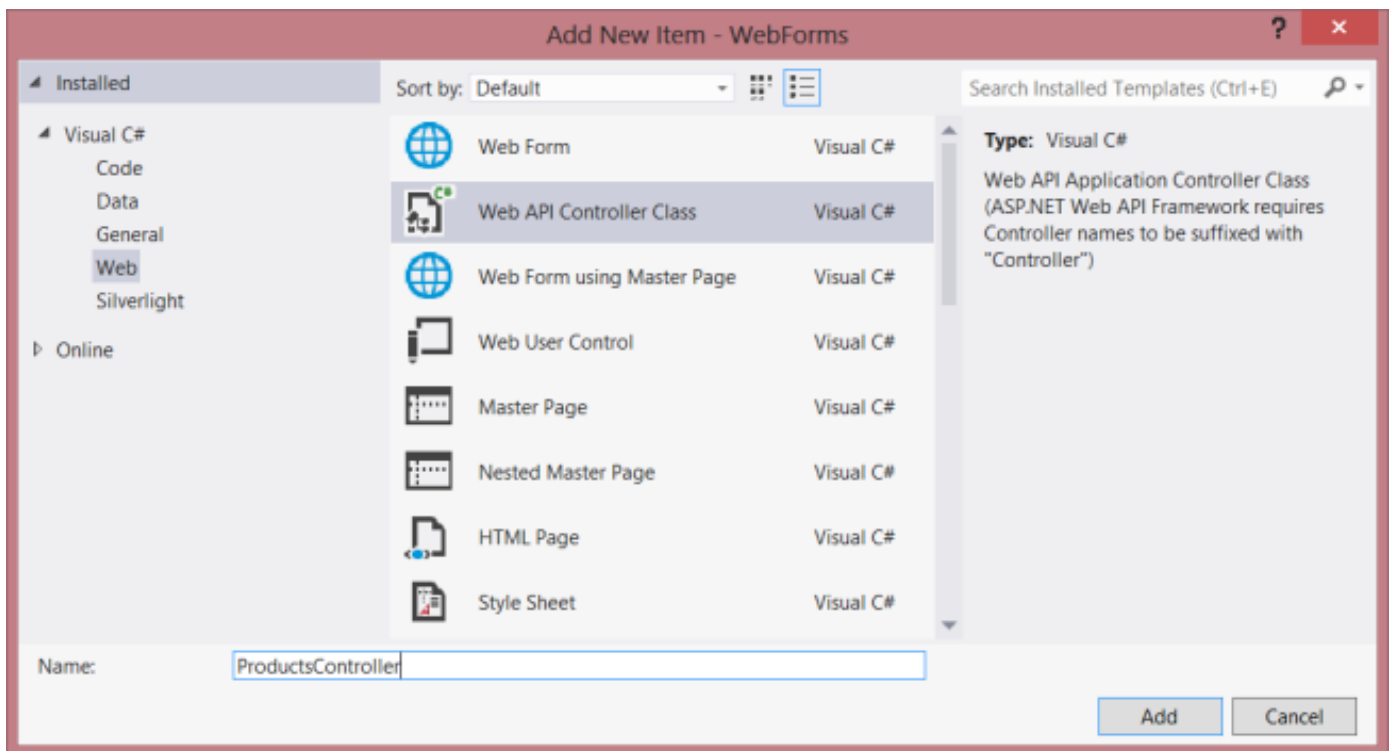
```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string Category { get; set; }
}
```

}

همانطور که مشاهده می‌کنید مدل مثال جاری نمایانگر یک محصول است. حال یک کنترلر Web API به پروژه اضافه کنید. کنترلرهای Web API درخواست‌های HTTP را به اکشن متدها نگاشت می‌کنند. در پنجره Solution Explorer روی نام پروژه کلیک راست کنید و گزینه Add, New Item را انتخاب کنید.



در دیالوگ باز شده گزینه Web را از پانل سمت چپ کلیک کنید و نوع آیتم جدید را **Web API Controller Class** انتخاب نمایید. نام این کنترلر را به "ProductsController" تغییر دهید و OK کنید.



کنترلر ایجاد شده شامل یک سری متد است که بصورت خودکار برای شما اضافه شده اند، آنها را حذف کنید و کد زیر را به کنترلر خود اضافه کنید.

```
namespace WebForms
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Net;
    using System.Net.Http;
    using System.Web.Http;

    public class ProductsController : ApiController
    {
        Product[] products = new Product[]
        {
            new Product { Id = 1, Name = "Tomato Soup", Category = "Groceries", Price = 1 },
            new Product { Id = 2, Name = "Yo-yo", Category = "Toys", Price = 3.75M },
            new Product { Id = 3, Name = "Hammer", Category = "Hardware", Price = 16.99M }
        };

        public IEnumerable<Product> GetAllProducts()
        {
            return products;
        }

        public Product GetProductById(int id)
        {
            var product = products.FirstOrDefault((p) => p.Id == id);
            if (product == null)
            {
                throw new HttpResponseException(HttpStatusCode.NotFound);
            }
            return product;
        }

        public IEnumerable<Product> GetProductsByCategory(string category)
        {
            return products.Where(
                (p) => string.Equals(p.Category, category,
                    StringComparison.OrdinalIgnoreCase));
        }
    }
}
```

```
}
}
```

کنترلر جاری لیستی از محصولات را بصورت استاتیک در حافظه محلی نگهداری می‌کند. متدهایی هم برای دریافت لیست محصولات تعریف شده اند.

اطلاعات مسیریابی را اضافه کنید

مرحله بعدی اضافه کردن اطلاعات مسیریابی (routing) است. در مثال جاری می‌خواهیم آدرس هایی مانند "/api/products" به کنترلر Web API نگاشت شوند. فایل **Global.asax** را باز کنید و عبارت زیر را به بالای آن اضافه نمایید.

```
using System.Web.Http;
```

حال کد زیر را به متد Application_Start اضافه کنید.

```
RouteTable.Routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = System.Web.Http.RouteParameter.Optional }
);
```

برای اطلاعات بیشتر درباره مسیریابی در Web API به [این لینک](#) مراجعه کنید.

دریافت اطلاعات بصورت آژاکسی در کلاینت

تا اینجا شما یک API دارید که کلاینت‌ها می‌توانند به آن دسترسی داشته باشند. حال یک صفحه HTML خواهیم ساخت که با استفاده از jQuery سرویس را فراخوانی می‌کند. صفحه Default.aspx را باز کنید و کدی که بصورت خودکار در قسمت Content تولید شده است را حذف کرده و کد زیر را به این قسمت اضافه کنید:

```
<%@ Page Title="Home Page" Language="C#" MasterPageFile="~/Site.Master"
    AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inherits="WebForms._Default" %>

<asp:Content ID="HeaderContent" runat="server" ContentPlaceHolderID="HeadContent">
</asp:Content>

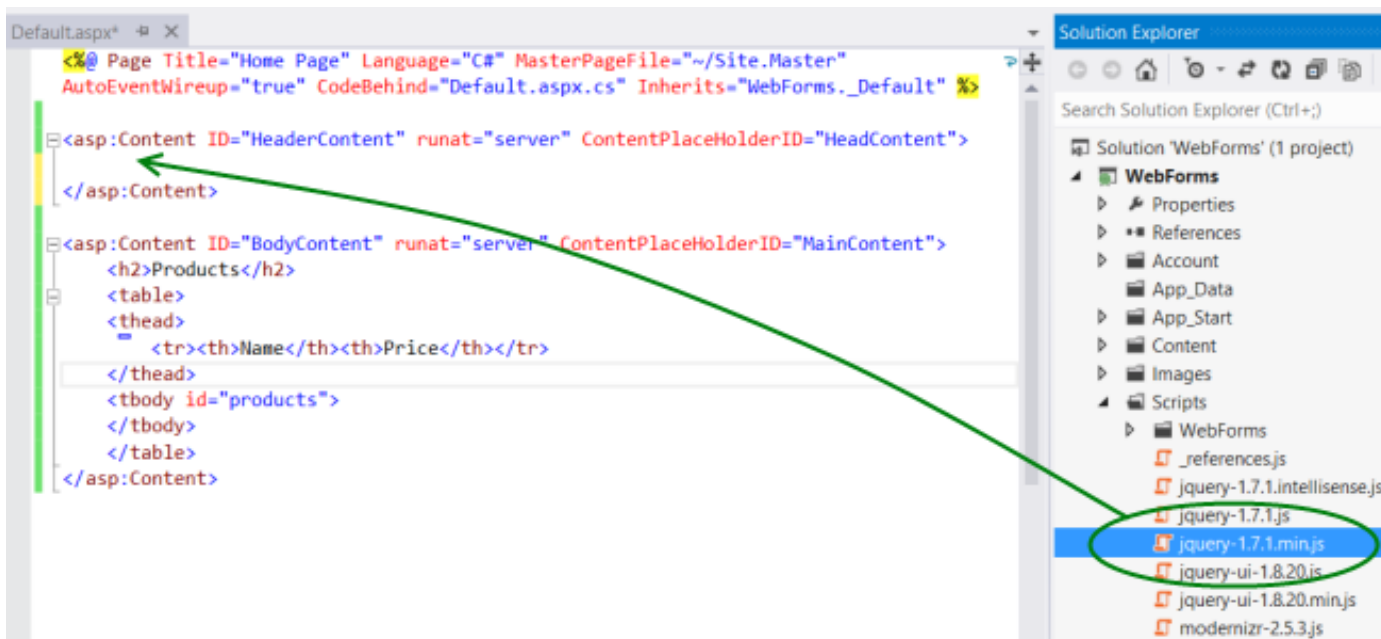
<asp:Content ID="BodyContent" runat="server" ContentPlaceHolderID="MainContent">
    <h2>Products</h2>
    <table>
    <thead>
        <tr><th>Name</th><th>Price</th></tr>
    </thead>
    <tbody id="products">
    </tbody>
    </table>
</asp:Content>
```

حال در قسمت HeaderContent کتابخانه jQuery را ارجاع دهید.

```
<asp:Content ID="HeaderContent" runat="server" ContentPlaceHolderID="HeadContent">
    <script src="Scripts/jquery-1.7.1.min.js" type="text/javascript"></script>
</asp:Content>
```

همانطور که می‌بینید در مثال جاری از فایل محلی استفاده شده است اما در اپلیکیشن‌های واقعی بهتر است از CDN‌ها استفاده کنید.

نکته: برای ارجاع دادن اسکریپت‌ها می‌توانید بسادگی فایل مورد نظر را با drag & drop به کد خود اضافه کنید.



زیر تگ jQuery اسکریپت زیر را اضافه کنید.

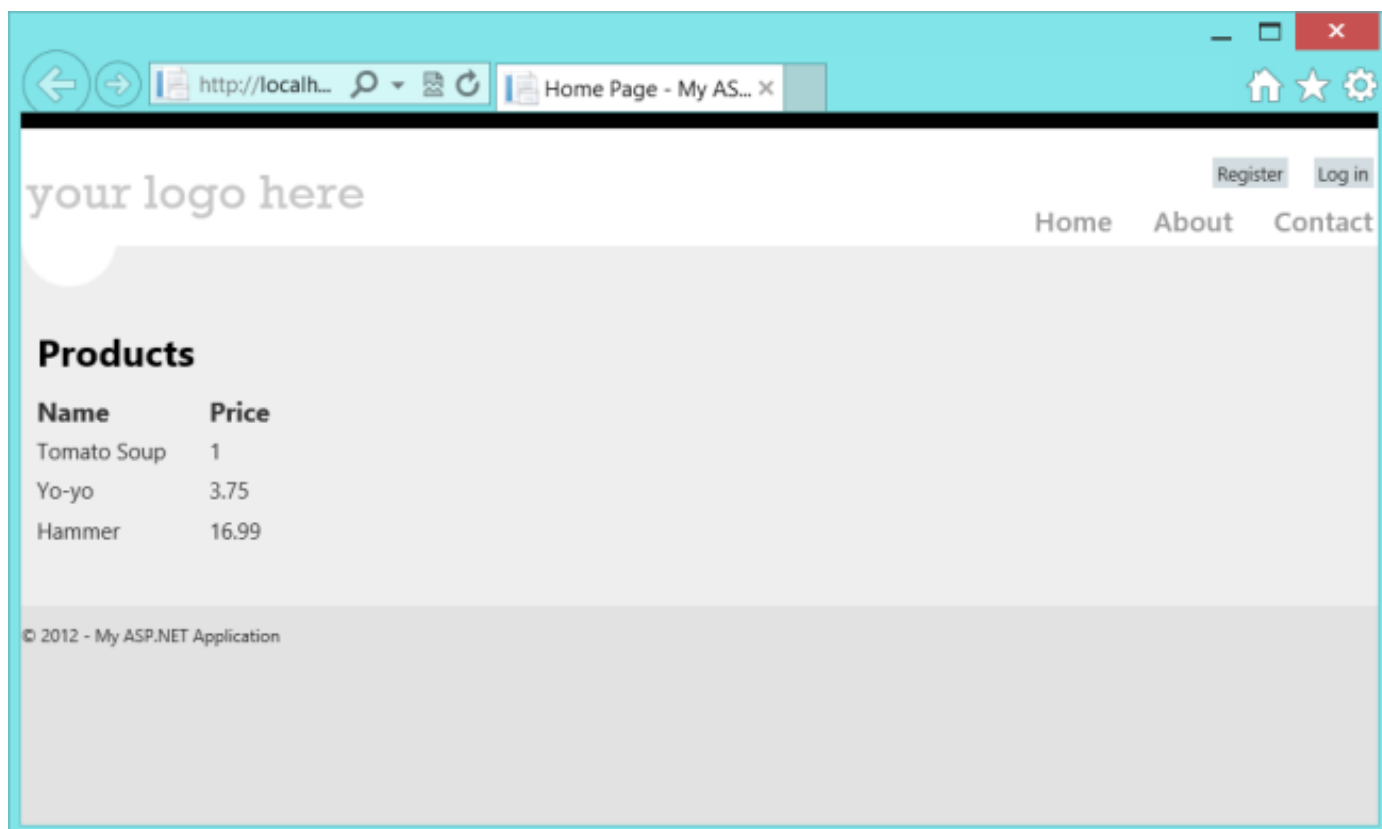
```
<script type="text/javascript">
    function getProducts() {
        $.getJSON("api/products",
            function (data) {
                $('#products').empty(); // Clear the table body.

                // Loop through the list of products.
                $.each(data, function (key, val) {
                    // Add a table row for the product.
                    var row = '<td>' + val.Name + '</td><td>' + val.Price + '</td>';
                    $('<tr>', { text: row }) // Append the name.
                        .appendTo($('#products'));
                });
            });
    }

    $(document).ready(getProducts);
</script>
```

هنگامی که سند جاری (document) بارگذاری شد این اسکریپت یک درخواست آژاکسی به آدرس "/api/products" ارسال می‌کند. سرور ما لیستی از محصولات را با فرمت JSON بر می‌گرداند، سپس این اسکریپت لیست دریافت شده را به جدول HTML اضافه می‌کند.

اگر اپلیکیشن را اجرا کنید باید با نمایی مانند تصویر زیر مواجه شوید:



نظرات خوانندگان

نویسنده:

ارشیا

تاریخ:

۱۳:۲۷ ۱۳۹۲/۱۱/۰۵

اگر بخواهیم زمانی که برای فیلد price مقداری که وارد میکند حتما نوع عددی یا اعشاری که شما در نظر گرفتید باشد ، باید چه کدی را اضافه کنیم . تا زمانی که مقدار عددی و یا اعشاری وارد نکند اجازه اضافه کردن سطر دیگر را ندهد . امکان چنین کاری وجود دارد ؟

نویسنده:

آرمین ضیاء

تاریخ:

۱۹:۵۰ ۱۳۹۲/۱۱/۰۵

می تونید از جاوا اسکریپت و Remote Validation استفاده کنید.

نویسنده:

ارشیا

تاریخ:

۱۱:۲۲ ۱۳۹۲/۱۱/۰۶

امکانش هست لینک مثالی در این باره بفرمایید یا نمونه برنامه ای ؟

نویسنده:

محسن خان

تاریخ:

۱۲:۵ ۱۳۹۲/۱۱/۰۶

مفاهیم [اعتبارسنجی در MVC](#) با [Web Api](#) تقریبا یکی است.

نویسنده:

مهرداد

تاریخ:

۱۶:۴۵ ۱۳۹۳/۰۲/۱۵

- آیا برای عملیات CRUD می توان از آن استفاده کرد؟ اضافه ، حذف ، آپدیت؟ (مثال؟)
 - آیا استفاده از web api جهت عملیات CRUD بجای استفاده از MS AJAX بهتر است ؟
 - برای اینکه فقط یوزرهای سایت به این web api دسترسی داشته باشند ، کد خاصی باید اضافه شود ؟
 - در نهایت سوال آخر : اگر بخواهیم تمام عملیات CRUD سایت(ASP.NET Web forms) را با web api انجام دهیم کار درستی است ؟
 بسیار متشکرم

نویسنده:

وحید نصیری

تاریخ:

۱۷:۲۵ ۱۳۹۳/۰۲/۱۵

- بله. [گروه Web API](#) و EF را در سایت پیگیری کنید.
 - Web API یک بحث سمت سرور است. به آن به زبان ساده به چشم یک وب سرویس مدرن نگاه کنید. برای نمونه بجای وبمتهای استاتیک صفحات aspx یا فایل های ashx یا asmx و حتی سرویس های WCF از نوع REST و امثال آن، بهتر است از Web API استفاده کنید.
 - برای نمونه پایه مباحثی مانند Forms Authentication در اینجا هم کاربرد دارد (البته این یک نمونه است).
 - برای کار با Web API الزاما نیازی به ASP.NET ندارید (نه وب فرم ها و نه MVC)؛ به هیچکدام از نگارش های آن. سمت کاربر آن AngularJS و سمت سرور آن Web API باشد. کار می کند. (اهمیت این مساله در اینجا است که الان می شود یک فریم ورک جدید توسعه ی برنامه های وب را کاملا مستقل از وب فرم ها و MVC طراحی کرد)

وقتی یک Web API می‌سازید بهتر است صفحات راهنمایی هم برای آن در نظر بگیرید، تا توسعه دهندگان بدانند چگونه باید سرویس شما را فراخوانی و استفاده کنند. گرچه می‌توانید مستندات را بصورت دستی ایجاد کنید، اما بهتر است تا جایی که ممکن است آنها را بصورت خودکار تولید نمایید.

بدین منظور فریم ورک ASP.NET Web API کتابخانه ای برای تولید خودکار صفحات راهنما در زمان اجرا (run-time) فراهم کرده است.

ASP.NET Web API

[Home](#)

ASP.NET Web API Help Page

Introduction

This API enables CRUD operations on a set of products.

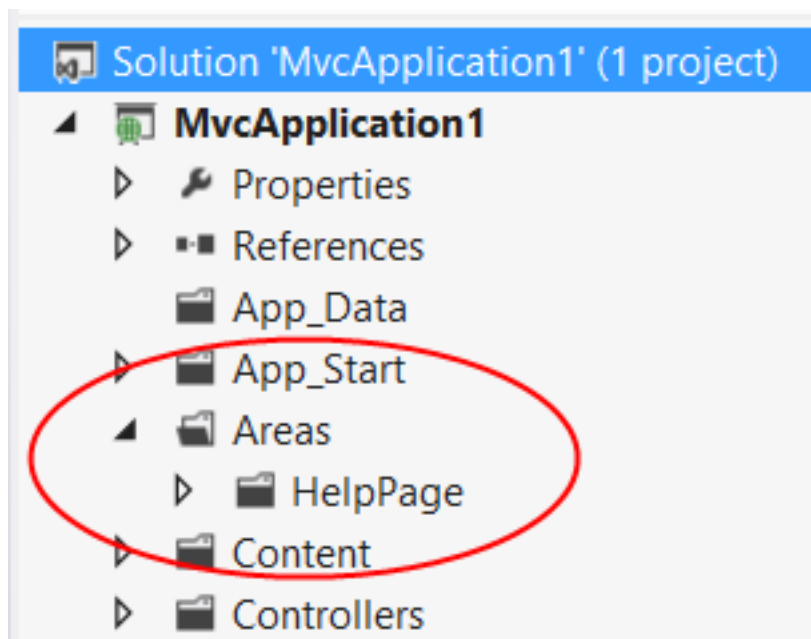
Products

API	Description
GET api/Products	Returns a list of products.
GET api/Products/{id}	Finds a product by ID.
POST api/Products	Creates a new product entity.

ایجاد صفحات راهنمای API

برای شروع ابتدا ابزار [ASP.NET and Web Tools 2012.2 Update](#) را نصب کنید. اگر از ویژوال استودیو 2013 استفاده می‌کنید این ابزار بصورت خودکار نصب شده است. این ابزار صفحات راهنما را به قالب پروژه‌های ASP.NET Web API اضافه می‌کند.

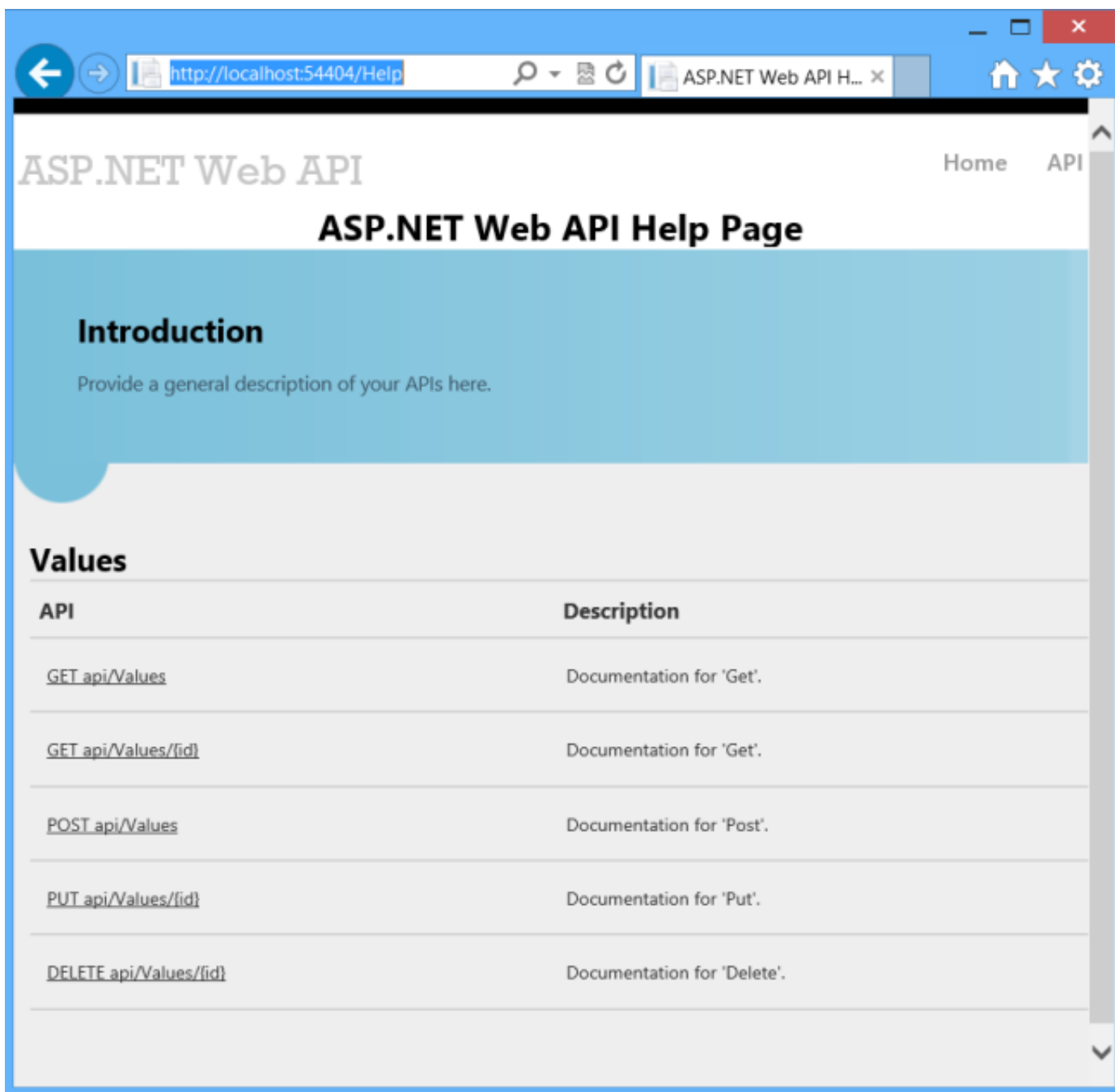
یک پروژه جدید از نوع ASP.NET MVC Application بسازید و قالب Web API را برای آن انتخاب کنید. این قالب پروژه کنترلری بنام ValuesController را بصورت خودکار برای شما ایجاد می‌کند. همچنین صفحات راهنمای API هم برای شما ساخته می‌شوند. تمام کد مربوط به صفحات راهنما در قسمت Areas قرار دارند.



اگر اپلیکیشن را اجرا کنید خواهید دید که صفحه اصلی لینکی به صفحه راهنمای API دارد. از صفحه اصلی، مسیر تقریبی /Help خواهد بود.



این لینک شما را به یک صفحه خلاصه (summary) هدایت می‌کند.



نمای این صفحه در مسیر `Areas/HelpPage/Views/Help/Index.cshtml` قرار دارد. می‌توانید این نما را ویرایش کنید و مثلاً قالب، عنوان، استایل‌ها و دیگر موارد را تغییر دهید.

بخش اصلی این صفحه متشکل از جدولی است که API‌ها را بر اساس کنترلر طبقه‌بندی می‌کند. مقادیر این جدول بصورت خودکار و توسط اینترفیس **IApiExplorer** تولید می‌شوند. در ادامه مقاله بیشتر درباره این اینترفیس صحبت خواهیم کرد. اگر کنترلر جدیدی به API خود اضافه کنید، این جدول بصورت خودکار در زمان اجرا بروز رسانی خواهد شد.

ستون "API" متد HTTP و آدرس نسبی را لیست می‌کند. ستون "Documentation" مستندات هر API را نمایش می‌دهد. مقادیر این ستون در ابتدا تنها `placeholder-text` است. در ادامه مقاله خواهید دید چگونه می‌توان از توضیحات XML برای تولید مستندات استفاده کرد.

هر API لینکی به یک صفحه جزئیات دارد، که در آن اطلاعات بیشتری درباره آن قابل مشاهده است. معمولا مثالی از بدنه‌های درخواست و پاسخ هم ارائه می‌شود.

GET api/Values

Documentation for 'Get'.

Response Information

Response body formats

application/json, text/json

Sample:

```
[
  "sample string 1",
  "sample string 2",
  "sample string 3"
]
```

application/xml, text/xml

Sample:

```
<ArrayOfstring xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.microsoft.com/2003/10/Serialization/Arrays">
  <string>sample string 1</string>
  <string>sample string 2</string>
  <string>sample string 3</string>
</ArrayOfstring>
```

افزودن صفحات راهنما به پروژه ای قدیمی

می‌توانید با استفاده از NuGet Package Manager صفحات راهنمای خود را به پروژه‌های قدیمی هم اضافه کنید. این گزینه مخصوصا هنگامی مفید است که با پروژه ای کار می‌کنید که قالب آن Web API نیست.

از منوی Tools گزینه‌های Library Package Manager, Package Manager Console را انتخاب کنید. در پنجره [Package Manager Console](#) فرمان زیر را وارد کنید.

```
Install-Package Microsoft.AspNet.WebApi.HelpPage
```

این پکیج اسمبلی‌های لازم برای صفحات راهنما را به پروژه اضافه می‌کند و نماهای MVC را در مسیر Areas/HelpPage می‌سازد.

اضافه کردن لینکی به صفحات راهنما باید بصورت دستی انجام شود. برای اضافه کردن این لینک به یک نمای Razor از کدی مانند لیست زیر استفاده کنید.

```
@Html.ActionLink("API", "Index", "Help", new { area = "" }, null)
```

همانطور که مشاهده می‌کنید مسیر نسبی صفحات راهنما "/Help" می‌باشد. همچنین اطمینان حاصل کنید که ناحیه‌ها (Areas) بدرستی رجیستر می‌شوند. فایل Global.asax را باز کنید و کد زیر را در صورتی که وجود ندارد اضافه کنید.

```
protected void Application_Start()
{
    // Add this code, if not present.
    AreaRegistration.RegisterAllAreas();

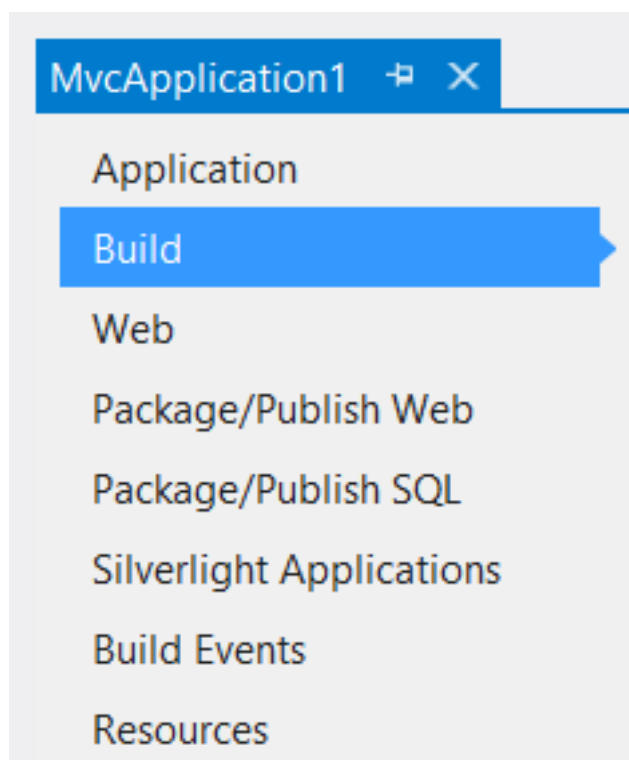
    // ...
}
```

افزودن مستندات API

بصورت پیش فرض صفحات راهنما از placeholder-text برای مستندات استفاده می‌کنند. می‌توانید برای ساختن مستندات از [توضیحات XML](#) استفاده کنید. برای فعال سازی این قابلیت فایل Areas/HelpPage/App_Start/HelpPageConfig.cs را باز کنید و خط زیر را از حالت کامنت درآورید:

```
config.SetDocumentationProvider(new XmlDocumentationProvider(
    HttpContext.Current.Server.MapPath("~/App_Data/XmlDocument.xml")));
```

حال روی نام پروژه کلیک راست کنید و **Properties** را انتخاب کنید. در پنجره باز شده قسمت **Build** را کلیک کنید.



زیر قسمت **Output** گزینه **XML documentation file** را تیک بزنید و در فیلد روبروی آن مقدار "App_Data/XmlDocument.xml" را وارد کنید.

Output

Output path:

bin\

☒ XML documentation file:

App_Data/XmlDocument.xml

حال کنترلر `ValuesController` را از مسیر `Controllers/ValuesController.cs` باز کنید و یک سری توضیحات XML به متدهای آن اضافه کنید. بعنوان مثال:

```

/// <summary>
/// Gets some very important data from the server.
/// </summary>
public IEnumerable<string> Get()
{
    return new string[] { "value1", "value2" };
}

/// <summary>
/// Looks up some data by ID.
/// </summary>
/// <param name="id">The ID of the data.</param>
public string Get(int id)
{
    return "value";
}

```

اپلیکیشن را مجدداً اجرا کنید و به صفحات راهنما بروید. حالا مستندات API شما باید تولید شده و نمایش داده شوند.

API	Description
GET api/Values	Gets some very important data from the server.
GET api/Values/{id}	Looks up some data by ID.

صفحات راهنما مستندات شما را در زمان اجرا از توضیحات XML استخراج می‌کنند. دقت کنید که هنگام توزیع اپلیکیشن، فایل XML را هم منتشر کنید.

توضیحات تکمیلی

صفحات راهنما توسط کلاس **ApiExplorer** تولید می‌شوند، که جزئی از فریم ورک ASP.NET Web API است. به ازای هر API این کلاس یک **ApiDescription** دارد که توضیحات لازم را در بر می‌گیرد. در اینجا منظور از "API" ترکیبی از متدهای HTTP و مسیرهای نسبی است. بعنوان مثال لیست زیر تعدادی API را نمایش می‌دهد:

GET /api/products
 GET /api/products/{id}
 POST /api/products

اگر اکشن‌های کنترلر از متدهای متعددی پشتیبانی کنند، ApiExplorer هر متد را بعنوان یک API مجزا در نظر خواهد گرفت. برای مخفی کردن یک API از ApiExplorer کافی است خاصیت **ApiExplorerSettings** را به اکشن مورد نظر اضافه کنید و مقدار خاصیت **IgnoreApi** آن را به **true** تنظیم نمایید.

```
[ApiExplorerSettings(IgnoreApi=true)]
public HttpResponseMessage Get(int id) { }
```

همچنین می‌توانید این خاصیت را به کنترلرها اضافه کنید تا تمام کنترلر از ApiExplorer مخفی شود.

کلاس **ApiExplorer** متن مستندات را توسط اینترفیس **IDocumentationProvider** دریافت می‌کند. کد مربوطه در مسیر `Areas/HelpPage/XmlDocumentation.cs` قرار دارد. همانطور که گفته شد مقادیر مورد نظر از توضیحات XML استخراج می‌شوند. نکته جالب آنکه می‌توانید با پیاده سازی این اینترفیس مستندات خود را از منبع دیگری استخراج کنید. برای اینکار باید متد الحاقی **SetDocumentationProvider** را هم فراخوانی کنید، که در **HelpPageConfigurationExtensions** تعریف شده است.

کلاس **ApiExplorer** بصورت خودکار اینترفیس **IDocumentationProvider** را فراخوانی می‌کند تا مستندات APIها را دریافت کند. سپس مقادیر دریافت شده را در خاصیت **Documentation** ذخیره می‌کند. این خاصیت روی آبجکت‌های **ApiDescription** و **ApiParameterDescription** تعریف شده است.

مطالعه بیشتر

[Adding a simple Test Client to ASP.NET Web API Help Page](#)
[Making ASP.NET Web API Help Page work on self-hosted services](#)
[Design-time generation of help page \(or client\) for ASP.NET Web API](#)
[Advanced Help Page customizations](#)

نظرات خوانندگان

نویسنده: سعید شیرزادیان
تاریخ: ۱۹:۵۵ ۱۳۹۲/۱۱/۱۸

سلام؛ می‌خواستم بدونم قابلیت فوق نیز بر روی پروژه‌های asp.net وب فرمز نیز فعال می‌گردد؟ با تشکر

نویسنده: محسن خان
تاریخ: ۲۰:۳۵ ۱۳۹۲/۱۱/۱۸

[Enabling ASP.NET Web API Help Pages for ASP.NET Web Forms Applications](#)

تمام اپلیکیشن‌ها را نمی‌توان در یک پروسس بسته بندی کرد، بدین معنا که تمام اپلیکیشن روی یک سرور فیزیکی قرار گیرد. در عصر حاضر معماری بسیاری از اپلیکیشن‌ها چند لایه است و هر لایه روی سرور مجزایی توزیع می‌شود. بعنوان مثال یک معماری کلاسیک شامل سه لایه نمایش (presentation)، اپلیکیشن (application) و داده (data) است. لایه بندی منطقی (logical layering) یک اپلیکیشن می‌تواند در یک App Domain واحد پیاده سازی شده و روی یک کامپیوتر میزبانی شود. در این صورت لازم نیست نگران مباحثی مانند پراکسی ها، مرتب سازی (serialization)، پروتوکل‌های شبکه و غیره باشیم. اما اپلیکیشن‌های بزرگی که چندین کلاینت دارند و در مراکز داده میزبانی می‌شوند باید تمام این مسائل را در نظر بگیرند. خوشبختانه پیاده سازی چنین اپلیکیشن‌هایی با استفاده از Entity Framework و دیگر تکنولوژی‌های مایکروسافت مانند WCF, Web API ساده‌تر شده است. منظور از n-Tier معماری اپلیکیشن‌هایی است که لایه‌های نمایش، منطق تجاری و دسترسی داده هر کدام روی سرور مجزایی میزبانی می‌شوند. این تفکیک فیزیکی لایه‌ها به بسط پذیری، مدیریت و نگهداری اپلیکیشن‌ها در دراز مدت کمک می‌کند، اما معمولاً تأثیری منفی روی کارایی کلی سیستم دارد. چرا که برای انجام عملیات مختلف باید از محدوده ماشین‌های فیزیکی عبور کنیم.

معماری N-Tier چالش‌های بخصوصی را برای قابلیت‌های change-tracking در EF اضافه می‌کند. در ابتدا داده‌ها توسط یک آبجکت EF Context بارگذاری می‌شوند اما این آبجکت پس از ارسال داده‌ها به کلاینت از بین می‌رود. تغییراتی که در سمت کلاینت روی داده‌ها اعمال می‌شوند ردیابی (track) نخواهند شد. هنگام بروز رسانی، آبجکت Context جدیدی برای پردازش اطلاعات ارسالی باید ایجاد شود. مسلماً آبجکت جدید هیچ چیز درباره Context پیشین یا مقادیر اصلی موجودیت‌ها نمی‌داند.

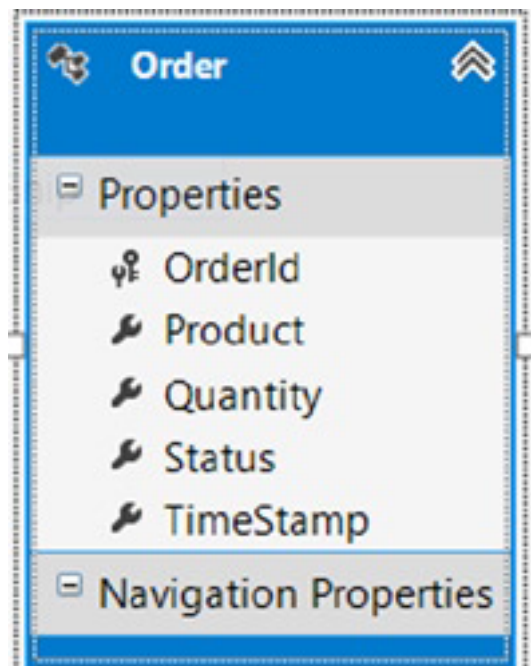
در نسخه‌های قبلی Entity Framework توسعه دهندگان با استفاده از قالب ویژه ای بنام Self-Tracking Entities می‌توانستند تغییرات موجودیت‌ها را ردیابی کنند. این قابلیت در نسخه EF 6 از رده خارج شده است و گرچه هنوز توسطObjectContext پشتیبانی می‌شود، آبجکت DbContext از آن پشتیبانی نمی‌کند.

در این سری از مقالات روی عملیات پایه CRUD تمرکز می‌کنیم که در اکثر اپلیکیشن‌های n-Tier استفاده می‌شوند. همچنین خواهیم دید چگونه می‌توان تغییرات موجودیت‌ها را ردیابی کرد. مباحثی مانند همزمانی (concurrency) و مرتب سازی (serialization) نیز بررسی خواهند شد. در قسمت یک این سری مقالات، به بروز رسانی موجودیت‌های منفصل (disconnected) توسط سرویس‌های Web API نگاهی خواهیم داشت.

بروز رسانی موجودیت‌های منفصل با Web API

سناریویی را فرض کنید که در آن برای انجام عملیات CRUD از یک سرویس Web API استفاده می‌شود. همچنین مدیریت داده‌ها با مدل Code-First پیاده سازی شده است. در مثال جاری یک کلاینت Console Application خواهیم داشت که یک سرویس Web API را فراخوانی می‌کند. توجه داشته باشید که هر اپلیکیشن در Solution مجزایی قرار دارد. تفکیک پروژه‌ها برای شبیه سازی یک محیط n-Tier انجام شده است.

فرض کنید مدلی مانند تصویر زیر داریم.



همانطور که می بینید مدل جاری، سفارشات یک اپلیکیشن فرضی را معرفی می کند. می خواهیم مدل و کد دسترسی به داده ها را در یک سرویس Web API پیاده سازی کنیم، تا هر کلاینتی که از HTTP استفاده می کند بتواند عملیات CRUD را انجام دهد. برای ساختن سرویس مورد نظر مراحل زیر را دنبال کنید.

در ویژوال استودیو پروژه جدیدی از نوع ASP.NET Web Application بسازید و قالب پروژه را Web API انتخاب کنید. نام پروژه را به Recipe1.Service تغییر دهید.

کنترلر جدیدی از نوع WebApi Controller با نام OrderController به پروژه اضافه کنید.

کلاس جدیدی با نام Order در پوشه مدل ها ایجاد کنید و کد زیر را به آن اضافه نمایید.

```
public class Order
{
    public int OrderId { get; set; }
    public string Product { get; set; }
    public int Quantity { get; set; }
    public string Status { get; set; }
    public byte[] TimeStamp { get; set; }
}
```

با استفاده از NuGet Package Manager کتابخانه Entity Framework 6 را به پروژه اضافه کنید.

حال کلاسی با نام Recipe1Context ایجاد کنید و کد زیر را به آن اضافه نمایید.

```
public class Recipe1Context : DbContext
{
    public Recipe1Context() : base("Recipe1ConnectionString") { }

    public DbSet<Order> Orders { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Order>().ToTable("Orders");
        // Following configuration enables timestamp to be concurrency token
        modelBuilder.Entity<Order>().Property(x => x.TimeStamp)
            .IsConcurrencyToken()
            .HasDatabaseGeneratedOption(DatabaseGeneratedOption.Computed);
    }
}
```

فایل Web.config پروژه را باز کنید و رشته اتصال زیر را به قسمت ConnectionStrings اضافه نمایید.

```
<connectionStrings>
  <add name="Recipe1ConnectionString"
    connectionString="Data Source=.;
    Initial Catalog=EFRecipes;
    Integrated Security=True;
    MultipleActiveResultSets=True"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

فایل Global.asax را باز کنید و کد زیر را به آن اضافه نمایید. این کد بررسی Entity Framework Compatibility را غیرفعال می‌کند.

```
protected void Application_Start()
{
    // Disable Entity Framework Model Compatibility
    Database.SetInitializer<Recipe1Context>(null);
    ...
}
```

در آخر کد کنترلر Order را با لیست زیر جایگزین کنید.

```
public class OrderController : ApiController
{
    // GET api/order
    public IEnumerable<Order> Get()
    {
        using (var context = new Recipe1Context())
        {
            return context.Orders.ToList();
        }
    }

    // GET api/order/5
    public Order Get(int id)
    {
        using (var context = new Recipe1Context())
        {
            return context.Orders.FirstOrDefault(x => x.OrderId == id);
        }
    }

    // POST api/order
    public HttpResponseMessage Post(Order order)
    {
        // Cleanup data from previous requests
        Cleanup();

        using (var context = new Recipe1Context())
        {
            context.Orders.Add(order);
            context.SaveChanges();
            // create HttpResponseMessage to wrap result, assigning Http Status code of 201,
            // which informs client that resource created successfully
            var response = Request.CreateResponse(HttpStatusCode.Created, order);
            // add location of newly-created resource to response header
            response.Headers.Location = new Uri(Url.Link("DefaultApi",
                new { id = order.OrderId }));
            return response;
        }
    }

    // PUT api/order/5
    public HttpResponseMessage Put(Order order)
    {
        using (var context = new Recipe1Context())
        {
            context.Entry(order).State = EntityState.Modified;
            context.SaveChanges();
            // return Http Status code of 200, informing client that resource updated successfully
            return Request.CreateResponse(HttpStatusCode.OK, order);
        }
    }

    // DELETE api/order/5
```

```

public HttpResponseMessage Delete(int id)
{
    using (var context = new Recipe1Context())
    {
        var order = context.Orders.FirstOrDefault(x => x.OrderId == id);
        context.Orders.Remove(order);
        context.SaveChanges();
        // Return Http Status code of 200, informing client that resource removed successfully
        return Request.CreateResponse(HttpStatusCode.OK);
    }
}

private void Cleanup()
{
    using (var context = new Recipe1Context())
    {
        context.Database.ExecuteSqlCommand("delete from [orders]");
    }
}
}

```

قابل ذکر است که هنگام استفاده از Entity Framework در MVC یا Web API، بکارگیری قابلیت Scaffolding بسیار مفید است. این فریم ورک های ASP.NET می توانند کنترلر هایی کاملاً اجرایی برایتان تولید کنند که صرفه جویی چشمگیری در زمان و کار شما خواهد بود.

در قدم بعدی اپلیکیشن کلاینت را می سازیم که از سرویس Web API استفاده می کند.

در ویژوال استودیو پروژه جدیدی از نوع Console Application بسازید و نام آن را به Recipe1.Client تغییر دهید. کلاس موجودیت Order را به پروژه اضافه کنید. همان کلاسی که در سرویس Web API ساختیم.

نکته: قسمت هایی از اپلیکیشن که باید در لایه های مختلف مورد استفاده قرار گیرند - مانند کلاس های موجودیت ها - بهتر است در لایه مجزایی قرار داده شده و به اشتراک گذاشته شوند. مثلاً می توانید پروژه ای از نوع Class Library بسازید و تمام موجودیت ها را در آن تعریف کنید. سپس لایه های مختلف این پروژه را ارجاع خواهند کرد.

فایل program.cs را باز کنید و کد زیر را به آن اضافه نمایید.

```

private HttpClient _client;
private Order _order;

private static void Main()
{
    Task t = Run();
    t.Wait();

    Console.WriteLine("\nPress <enter> to continue...");
    Console.ReadLine();
}

private static async Task Run()
{
    // create instance of the program class
    var program = new Program();
    program.ServiceSetup();
    program.CreateOrder();
    // do not proceed until order is added
    await program.PostOrderAsync();
    program.ChangeOrder();
    // do not proceed until order is changed
    await program.PutOrderAsync();
    // do not proceed until order is removed
    await program.RemoveOrderAsync();
}

private void ServiceSetup()
{
    // map URL for Web API call
    _client = new HttpClient { BaseAddress = new Uri("http://localhost:3237/") };
    // add Accept Header to request Web API content
}

```

```

    // negotiation to return resource in JSON format
    _client.DefaultRequestHeaders.Accept.
        Add(new MediaTypeWithQualityHeaderValue("application/json"));
}

private void CreateOrder()
{
    // Create new order
    _order = new Order { Product = "Camping Tent", Quantity = 3, Status = "Received" };
}

private async Task PostOrderAsync()
{
    // leverage Web API client side API to call service
    var response = await _client.PostAsJsonAsync("api/order", _order);
    Uri newOrderUri;

    if (response.IsSuccessStatusCode)
    {
        // Capture Uri of new resource
        newOrderUri = response.Headers.Location;
        // capture newly-created order returned from service,
        // which will now include the database-generated Id value
        _order = await response.Content.ReadAsAsync<Order>();
        Console.WriteLine("Successfully created order. Here is URL to new resource: {0}",
newOrderUri);
    }
    else
        Console.WriteLine("{0} ({1})", (int)response.StatusCode, response.ReasonPhrase);
}

private void ChangeOrder()
{
    // update order
    _order.Quantity = 10;
}

private async Task PutOrderAsync()
{
    // construct call to generate HttpPut verb and dispatch
    // to corresponding Put method in the Web API Service
    var response = await _client.PutAsJsonAsync("api/order", _order);

    if (response.IsSuccessStatusCode)
    {
        // capture updated order returned from service, which will include new quantity
        _order = await response.Content.ReadAsAsync<Order>();
        Console.WriteLine("Successfully updated order: {0}", response.StatusCode);
    }
    else
        Console.WriteLine("{0} ({1})", (int)response.StatusCode, response.ReasonPhrase);
}

private async Task RemoveOrderAsync()
{
    // remove order
    var uri = "api/order/" + _order.OrderId;
    var response = await _client.DeleteAsync(uri);

    if (response.IsSuccessStatusCode)
        Console.WriteLine("Sucessfully deleted order: {0}", response.StatusCode);
    else
        Console.WriteLine("{0} ({1})", (int)response.StatusCode, response.ReasonPhrase);
}

```

اگر اپلیکیشن کلاینت را اجرا کنید باید با خروجی زیر مواجه شوید:

Successfully created order: http://localhost:3237/api/order/1054

Successfully updated order: OK

Sucessfully deleted order: OK

شرح مثال جاری

با اجرای اپلیکیشن Web API شروع کنید. این اپلیکیشن یک کنترلر Web API دارد که پس از اجرا شما را به صفحه خانه هدایت می‌کند. در این مرحله اپلیکیشن در حال اجرا است و سرویس‌های ما قابل دسترسی هستند.

حال اپلیکیشن کنسول را باز کنید. روی خط اول کد `program.cs` یک breakpoint تعریف کرده و اپلیکیشن را اجرا کنید. ابتدا آدرس سرویس Web API را پیکربندی کرده و خاصیت `Accept Header` را مقدار دهی می‌کنیم. با این کار از سرویس مورد نظر درخواست می‌کنیم که داده‌ها را با فرمت JSON بازگرداند. سپس یک آبجکت `Order` می‌سازیم و با فراخوانی متد `PostAsJsonAsync` آن را به سرویس ارسال می‌کنیم. این متد روی آبجکت `HttpClient` تعریف شده است. اگر به اکشن متد `Post` در کنترلر `Order` یک breakpoint اضافه کنید، خواهید دید که این متد سفارش جدید را بعنوان یک پارامتر دریافت می‌کند و آن را به لیست موجودیت‌ها در Context جاری اضافه می‌نماید. این عمل باعث می‌شود که آبجکت جدید بعنوان `Added` علامت گذاری شود، در این مرحله Context جاری شروع به ردیابی تغییرات می‌کند. در آخر با فراخوانی متد `SaveChanges` داده‌ها را ذخیره می‌کنیم. در قدم بعدی کد وضعیت 201 (Created) و آدرس منبع جدید را در یک آبجکت `HttpResponseMessage` قرار می‌دهیم و به کلاینت ارسال می‌کنیم. هنگام استفاده از Web API باید اطمینان حاصل کنیم که کلاینت‌ها درخواست‌های ایجاد رکورد جدید را بصورت POST ارسال می‌کنند. درخواست‌های HTTP Post بصورت خودکار به اکشن متد متناظر نگاشت می‌شوند.

در مرحله بعد عملیات بعدی را اجرا می‌کنیم، تعداد سفارش را تغییر می‌دهیم و موجودیت جاری را با فراخوانی متد `PutAsJsonAsync` به سرویس Web API ارسال می‌کنیم. اگر به اکشن متد `Put` در کنترلر سرویس یک breakpoint اضافه کنید، خواهید دید که آبجکت سفارش بصورت یک پارامتر دریافت می‌شود. سپس با فراخوانی متد `Entry` و پاس دادن موجودیت جاری بعنوان رفرنس، خاصیت `State` را به `Modified` تغییر می‌دهیم، که این کار موجودیت را به Context جاری می‌چسباند. حال فراخوانی متد `SaveChanges` یک اسکریپت بروز رسانی تولید خواهد کرد. در مثال جاری تمام فیلدهای آبجکت `Order` را بروز رسانی می‌کنیم. در شماره‌های بعدی این سری از مقالات، خواهیم دید چگونه می‌توان تنها فیلدهایی را بروز رسانی کرد که تغییر کرده اند. در آخر عملیات را با بازگرداندن کد وضعیت 200 (OK) به اتمام می‌رسانیم.

در مرحله بعد، عملیات نهایی را اجرا می‌کنیم که موجودیت `Order` را از منبع داده حذف می‌کند. برای اینکار شناسه (Id) رکورد مورد نظر را به آدرس سرویس اضافه می‌کنیم و متد `DeleteAsync` را فراخوانی می‌کنیم. در سرویس Web API رکورد مورد نظر را از دیتابیس دریافت کرده و متد `Remove` را روی Context جاری فراخوانی می‌کنیم. این کار موجودیت مورد نظر را بعنوان `Deleted` علامت گذاری می‌کند. فراخوانی متد `SaveChanges` یک اسکریپت `Delete` تولید خواهد کرد که نهایتاً منجر به حذف شدن رکورد می‌شود.

در یک اپلیکیشن واقعی بهتر است کد دسترسی داده‌ها از سرویس Web API تفکیک شود و در لایه مجزایی قرار گیرد.

در [قسمت قبلی](#) بروز رسانی موجودیت های منفصل با WCF را بررسی کردیم. در این قسمت خواهیم دید چگونه می توان تغییرات موجودیت ها را تشخیص داد و عملیات CRUD را روی یک Object Graph اجرا کرد.

تشخیص تغییرات با Web API

فرض کنید می خواهیم از سرویس های Web API برای انجام عملیات CRUD استفاده کنیم، اما بدون آنکه برای هر موجودیت متدهایی مجزا تعریف کنیم. به بیان دیگر می خواهیم عملیات مذکور را روی یک Object Graph انجام دهیم. مدیریت داده ها هم با مدل Code-First پیاده سازی می شود. در مثال جاری یک اپلیکیشن کنسول خواهیم داشت که بعنوان یک کلاینت سرویس را فراخوانی می کند. هر پروژه نیز در Solution مجزایی قرار دارد، تا یک محیط n-Tier را شبیه سازی کنیم.

مدل زیر را در نظر بگیرید.



همانطور که می بینید مدل ما آژانس های مسافرتی و رزرواسیون آنها را ارائه می کند. می خواهیم مدل و کد دسترسی داده ها را در یک سرویس Web API پیاده سازی کنیم تا هر کلاینتی که به HTTP دسترسی دارد بتواند عملیات CRUD را انجام دهد. برای ساختن سرویس مورد نظر مراحل زیر را دنبال کنید:

در ویژوال استودیو پروژه جدیدی از نوع ASP.NET Web Application بسازید و قالب پروژه را Web API انتخاب کنید. نام پروژه را به Recipe3.Service تغییر دهید.

کنترلر جدیدی بنام TravelAgentController به پروژه اضافه کنید.

دو کلاس جدید با نام های TravelAgent و Booking بسازید و کد آنها را مطابق لیست زیر تغییر دهید.

```
public class TravelAgent
{
    public TravelAgent()
    {
        this.Bookings = new HashSet<Booking>();
    }

    public int AgentId { get; set; }
    public string Name { get; set; }
    public virtual ICollection<Booking> Bookings { get; set; }
}

public class Booking
{
    public int BookingId { get; set; }
    public int AgentId { get; set; }
    public string Customer { get; set; }
    public DateTime BookingDate { get; set; }
    public bool Paid { get; set; }
    public virtual TravelAgent TravelAgent { get; set; }
}
```

با استفاده از NuGet Package Manager کتابخانه Entity Framework 6 را به پروژه اضافه کنید.

کلاس جدیدی بنام Recipe3Context بسازید و کد آن را مطابق لیست زیر تغییر دهید.

```
public class Recipe3Context : DbContext
{
    public Recipe3Context() : base("Recipe3ConnectionString") { }
    public DbSet<TravelAgent> TravelAgents { get; set; }
    public DbSet<Booking> Bookings { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<TravelAgent>().HasKey(x => x.AgentId);
        modelBuilder.Entity<TravelAgent>().ToTable("TravelAgents");
        modelBuilder.Entity<Booking>().ToTable("Bookings");
    }
}
```

فایل Web.config پروژه را باز کنید و رشته اتصال زیر را به قسمت ConnectionStrings اضافه کنید.

```
<connectionStrings>
  <add name="Recipe3ConnectionString"
    connectionString="Data Source=.;
    Initial Catalog=EFRecipes;
    Integrated Security=True;
    MultipleActiveResultSets=True"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

فایل Global.asax را باز کنید و کد زیر را به متد Application_Start اضافه نمایید. این کد بررسی Model Compatibility در EF را غیرفعال می کند. همچنین به JSON serializer می گوئیم که self-referencing loop خاصیت های پیمایشی را نادیده بگیرد. این حلقه بدلیل ارتباط bidirectional بین موجودیت ها بوجود می آید.

```
protected void Application_Start()
{
    // Disable Entity Framework Model Compatibility
    Database.SetInitializer<Recipe1Context>(null);

    // The bidirectional navigation properties between related entities
    // create a self-referencing loop that breaks Web API's effort to
    // serialize the objects as JSON. By default, Json.NET is configured
    // to error when a reference loop is detected. To resolve problem,
    // simply configure JSON serializer to ignore self-referencing loops.
    GlobalConfiguration.Configuration.Formatters.JsonFormatter
        .SerializerSettings.ReferenceLoopHandling =
        Newtonsoft.Json.ReferenceLoopHandling.Ignore;
    ...
}
```

```
}
```

فایل RouteConfig.cs را باز کنید و قوانین مسیریابی را مانند لیست زیر تغییر دهید.

```
public static void Register(HttpConfiguration config)
{
    config.Routes.MapHttpRoute(
        name: "ActionMethodSave",
        routeTemplate: "api/{controller}/{action}/{id}",
        defaults: new { id = RouteParameter.Optional });
}
```

در آخر کنترلر TravelAgent را باز کنید و کد آن را مطابق لیست زیر بروز رسانی کنید.

```
public class TravelAgentController : ApiController
{
    // GET api/travelagent
    [HttpGet]
    public IEnumerable<TravelAgent> Retrieve()
    {
        using (var context = new Recipe3Context())
        {
            return context.TravelAgents.Include(x => x.Bookings).ToList();
        }
    }

    /// <summary>
    /// Update changes to TravelAgent, implementing Action-Based Routing in Web API
    /// </summary>
    public HttpResponseMessage Update(TravelAgent travelAgent)
    {
        using (var context = new Recipe3Context())
        {
            var newParentEntity = true;
            // adding the object graph makes the context aware of entire
            // object graph (parent and child entities) and assigns a state
            // of added to each entity.
            context.TravelAgents.Add(travelAgent);
            if (travelAgent.AgentId > 0)
            {
                // as the Id property has a value greater than 0, we assume
                // that travel agent already exists and set entity state to
                // be updated.
                context.Entry(travelAgent).State = EntityState.Modified;
                newParentEntity = false;
            }

            // iterate through child entities, assigning correct state.
            foreach (var booking in travelAgent.Bookings)
            {
                if (booking.BookingId > 0)
                {
                    // assume booking already exists if ID is greater than zero.
                    // set entity to be updated.
                    context.Entry(booking).State = EntityState.Modified;
                }
            }

            context.SaveChanges();
            HttpResponseMessage response;
            // set Http Status code based on operation type
            response = Request.CreateResponse(newParentEntity ? HttpStatusCode.Created :
            HttpStatusCode.OK, travelAgent);
            return response;
        }
    }

    [HttpDelete]
    public HttpResponseMessage Cleanup()
    {
        using (var context = new Recipe3Context())
        {
            context.Database.ExecuteSqlCommand("delete from [bookings]");
            context.Database.ExecuteSqlCommand("delete from [travelagents]");
        }
        return Request.CreateResponse(HttpStatusCode.OK);
    }
}
```

}

در قدم بعدی کلاینت پروژه را می‌سازیم که از سرویس Web API مان استفاده می‌کند.

در ویژوال استودیو پروژه جدیدی از نوع Console application بسازید و نام آن را به Recipe3.Client تغییر دهید.
فایل program.cs را باز کنید و کد آن را مطابق لیست زیر بروز رسانی کنید.

```
internal class Program
{
    private HttpClient _client;
    private TravelAgent _agent1, _agent2;
    private Booking _booking1, _booking2, _booking3;
    private HttpResponseMessage _response;

    private static void Main()
    {
        Task t = Run();
        t.Wait();
        Console.WriteLine("\nPress <enter> to continue...");
        Console.ReadLine();
    }

    private static async Task Run()
    {
        var program = new Program();
        program.ServiceSetup();
        // do not proceed until clean-up is completed
        await program.CleanupAsync();
        program.CreateFirstAgent();
        // do not proceed until agent is created
        await program.AddAgentAsync();
        program.CreateSecondAgent();
        // do not proceed until agent is created
        await program.AddSecondAgentAsync();
        program.ModifyAgent();
        // do not proceed until agent is updated
        await program.UpdateAgentAsync();
        // do not proceed until agents are fetched
        await program.FetchAgentsAsync();
    }

    private void ServiceSetup()
    {
        // set up infrastructure for Web API call
        _client = new HttpClient {BaseAddress = new Uri("http://localhost:6687/")};
        // add Accept Header to request Web API content negotiation to return resource in JSON format
        _client.DefaultRequestHeaders.Accept.Add(new
MediaTypesWithQualityHeaderValue("application/json"));
    }

    private async Task CleanupAsync()
    {
        // call cleanup method in service
        _response = await _client.DeleteAsync("api/travelagent/cleanup/");
    }

    private void CreateFirstAgent()
    {
        // create new Travel Agent and booking
        _agent1 = new TravelAgent {Name = "John Tate"};
        _booking1 = new Booking
        {
            Customer = "Karen Stevens",
            Paid = false,
            BookingDate = DateTime.Parse("2/2/2010")
        };

        _booking2 = new Booking
        {
            Customer = "Dolly Parton",
            Paid = true,
            BookingDate = DateTime.Parse("3/10/2010")
        };

        _agent1.Bookings.Add(_booking1);
        _agent1.Bookings.Add(_booking2);
    }
}
```

```

}

private async Task AddAgentAsync()
{
    // call generic update method in Web API service to add agent and bookings
    _response = await _client.PostAsync("api/travelagent/update/",
        _agent1, new JsonMediaTypeFormatter());

    if (_response.IsSuccessStatusCode)
    {
        // capture newly created travel agent from service, which will include
        // database-generated Ids for each entity
        _agent1 = await _response.Content.ReadAsAsync<TravelAgent>();
        _booking1 = _agent1.Bookings.FirstOrDefault(x => x.Customer == "Karen Stevens");
        _booking2 = _agent1.Bookings.FirstOrDefault(x => x.Customer == "Dolly Parton");

        Console.WriteLine("Successfully created Travel Agent {0} and {1} Booking(s)",
            _agent1.Name, _agent1.Bookings.Count);
    }
    else
        Console.WriteLine("{0} ({1})", (int) _response.StatusCode, _response.ReasonPhrase);
}

private void CreateSecondAgent()
{
    // add new agent and booking
    _agent2 = new TravelAgent {Name = "Perry Como"};
    _booking3 = new Booking {
        Customer = "Loretta Lynn",
        Paid = true,
        BookingDate = DateTime.Parse("3/15/2010")};
    _agent2.Bookings.Add(_booking3);
}

private async Task AddSecondAgentAsync()
{
    // call generic update method in Web API service to add agent and booking
    _response = await _client.PostAsync("api/travelagent/update/", _agent2, new
JsonMediaTypeFormatter());

    if (_response.IsSuccessStatusCode)
    {
        // capture newly created travel agent from service
        _agent2 = await _response.Content.ReadAsAsync<TravelAgent>();
        _booking3 = _agent2.Bookings.FirstOrDefault(x => x.Customer == "Loretta Lynn");
        Console.WriteLine("Successfully created Travel Agent {0} and {1} Booking(s)",
            _agent2.Name, _agent2.Bookings.Count);
    }
    else
        Console.WriteLine("{0} ({1})", (int) _response.StatusCode, _response.ReasonPhrase);
}

private void ModifyAgent()
{
    // modify agent 2 by changing agent name and assigning booking 1 to him from agent 1
    _agent2.Name = "Perry Como, Jr.";
    _agent2.Bookings.Add(_booking1);
}

private async Task UpdateAgentAsync()
{
    // call generic update method in Web API service to update agent 2
    _response = await _client.PostAsync("api/travelagent/update/", _agent2, new
JsonMediaTypeFormatter());
    if (_response.IsSuccessStatusCode)
    {
        // capture newly created travel agent from service, which will include Ids
        _agent1 = _response.Content.ReadAsAsync<TravelAgent>().Result;
        Console.WriteLine("Successfully updated Travel Agent {0} and {1} Booking(s)", _agent1.Name,
            _agent1.Bookings.Count);
    }
    else
        Console.WriteLine("{0} ({1})", (int) _response.StatusCode, _response.ReasonPhrase);
}

private async Task FetchAgentsAsync()
{
    // call Get method on service to fetch all Travel Agents and Bookings
    _response = _client.GetAsync("api/travelagent/retrieve").Result;
    if (_response.IsSuccessStatusCode)
    {

```

```
// capture newly created travel agent from service, which will include Ids
var agents = await _response.Content.ReadAsAsync<IEnumerable<TravelAgent>>();

foreach (var agent in agents)
{
    Console.WriteLine("Travel Agent {0} has {1} Booking(s)", agent.Name,
agent.Bookings.Count());
}
}
else
    Console.WriteLine("{0} ({1})", (int) _response.StatusCode, _response.ReasonPhrase);
}
}
```

در آخر کلاس های TravelAgent و Booking را به پروژه کلاینت اضافه کنید. اینگونه کدها بهتر است در لایه مجزایی قرار گیرند و بین پروژه ها به اشتراک گذاشته شوند.

اگر اپلیکیشن کنسول (کلاینت) را اجرا کنید با خروجی زیر مواجه خواهید شد.

```
Successfully created Travel Agent John Tate and 2 Booking(s)
Successfully created Travel Agent Perry Como and 1 Booking(s)
Successfully updated Travel Agent Perry Como, Jr. and 2 Booking(s)
Travel Agent John Tate has 1 Booking(s)
Travel Agent Perry Como, Jr. has 2 Booking(s)
```

شرح مثال جاری

با اجرای اپلیکیشن Web API شروع کنید. این اپلیکیشن یک کنترلر MVC Web Controller دارد که پس از اجرا شما را به صفحه خانه هدایت می کند. در این مرحله سایت در حال اجرا است و سرویس ها قابل دسترسی هستند.

سپس اپلیکیشن کنسول را باز کنید، روی خط اول کد فایل program.cs یک breakpoint قرار دهید و آن را اجرا کنید. ابتدا آدرس سرویس Web API را نگاشت می کنیم و با تنظیم مقدار خاصیت Accept Header از سرویس درخواست می کنیم که اطلاعات را با فرمت JSON بازگرداند.

بعد از آن با استفاده از آبجکت HttpClient متد DeleteAsync را فراخوانی می کنیم که روی کنترلر TravelAgent تعریف شده است. این متد تمام داده های پیشین را حذف میکند.

در قدم بعدی سه آبجکت جدید می سازیم: یک آژانس مسافرتی و دو رزرواسیون. سپس این آبجکت ها را با فراخوانی متد PostAsync روی آبجکت HttpClient به سرویس ارسال می کنیم. اگر به متد Update در کنترلر TravelAgent یک breakpoint اضافه کنید، خواهید دید که این متد آبجکت آژانس مسافرتی را بعنوان یک پارامتر دریافت می کند و آن را به موجودیت TravelAgents در Context جاری اضافه می نماید. این کار آبجکت آژانس مسافرتی و تمام آبجکت های فرزند آن را در حالت Added اضافه می کند و باعث می شود که context جاری شروع به ردیابی (tracking) آنها کند.

نکته: قابل ذکر است که اگر موجودیت های متعددی با مقداری یکسان در خاصیت کلید اصلی (Primary-key value) دارید باید مجموعه آبجکت های خود را Add کنید و نه Attach. در مثال جاری چند آبجکت Booking داریم که مقدار کلید اصلی آنها صفر است (Bookings with Id = 0). اگر از Attach استفاده کنید EF پیغام خطایی صادر می کند چرا که چند موجودیت با مقادیر کلید اصلی یکسان به context جاری اضافه کرده اید.

بعد از آن بر اساس مقدار خاصیت Id مشخص می کنیم که موجودیت ها باید بروز رسانی شوند یا خیر. اگر مقدار این فیلد بزرگتر از صفر باشد، فرض بر این است که این موجودیت در دیتابیس وجود دارد بنابراین خاصیت EntityState را به Modified تغییر می دهیم. علاوه بر این فیلدی هم با نام newParentEntity تعریف کرده ایم که توسط آن بتوانیم کد وضعیت مناسبی به کلاینت بازگردانیم. در صورتی که مقدار فیلد Id در موجودیت TravelAgent برابر با یک باشد، مقدار خاصیت EntityState را به همان

Added رها می کنیم.

سپس تمام آبجکت های فرزند آژانس مسافرتی (رزرواسیون ها) را بررسی میکنیم و همین منطق را روی آنها اعمال می کنیم. یعنی در صورتی که مقدار فیلد Id آنها بزرگتر از 0 باشد وضعیت EntityState را به Modified تغییر می دهیم. در نهایت متد SaveChanges را فراخوانی می کنیم. در این مرحله برای موجودیت های جدید اسکریپت های Insert و برای موجودیت های تغییر کرده اسکریپت های Update تولید می شود. سپس کد وضعیت مناسب را به کلاینت بر می گردانیم. برای موجودیت های اضافه شده کد وضعیت 201 (Created) و برای موجودیت های بروز رسانی شده کد وضعیت 200 (OK) باز می گردد. کد 201 به کلاینت اطلاع می دهد که رکورد جدید با موفقیت ثبت شده است، و کد 200 از بروز رسانی موفقیت آمیز خبر می دهد. هنگام تولید سرویس های REST-based بهتر است همیشه کد وضعیت مناسبی تولید کنید.

پس از این مراحل، آژانس مسافرتی و رزرواسیون جدیدی می سازیم و آنها را به سرویس ارسال می کنیم. سپس نام آژانس مسافرتی دوم را تغییر می دهیم، و یکی از رزرواسیون ها را از آژانس اولی به آژانس دومی منتقل می کنیم. اینبار هنگام فراخوانی متد Update تمام موجودیت ها شناسه ای بزرگتر از 1 دارند، بنابراین وضعیت EntityState آنها را به Modified تغییر می دهیم تا هنگام ثبت تغییرات دستورات بروز رسانی مناسب تولید و اجرا شوند.

در آخر کلاینت ما متد Retrieve را روی سرویس فراخوانی می کند. این فراخوانی با کمک متد GetAsync انجام می شود که روی آبجکت HttpClient تعریف شده است. فراخوانی این متد تمام آژانس های مسافرتی به همراه رزرواسیون های متناظرشان را دریافت می کند. در اینجا با استفاده از متد Include تمام رکوردهای فرزند را به همراه تمام خاصیت هایشان (properties) بارگذاری می کنیم.

دقت کنید که مرتب کننده JSON تمام خواص عمومی (public properties) را باز می گرداند، حتی اگر در کد خود تعداد مشخصی از آنها را انتخاب کرده باشید.

نکته دیگر آنکه در مثال جاری از قراردادهای توکار Web API برای نگاشت درخواست های HTTP به اکشن متدها استفاده نکرده ایم. مثلاً بصورت پیش فرض درخواست های POST به متدهایی نگاشت می شوند که نام آنها با "Post" شروع می شود. در مثال جاری قواعد مسیریابی را تغییر داده ایم و رویکرد مسیریابی RPC-based را در پیش گرفته ایم. در اپلیکیشن های واقعی بهتر است از قواعد پیش فرض استفاده کنید چرا که هدف Web API ارائه سرویس های REST-based است. بنابراین بعنوان یک قاعده کلی بهتر است متدهای سرویس شما به درخواست های متناظر HTTP نگاشت شوند. و در آخر آنکه بهتر است لایه مجزایی برای میزبانی کدهای دسترسی داده ایجاد کنید و آنها را از سرویس Web API تفکیک نمایید.

نظرات خوانندگان

نویسنده: وحید

تاریخ: ۱۱:۶ ۱۳۹۲/۱۱/۱۱

با سلام شما فرمودید: " و در آخر آنکه بهتر است لایه مجزایی برای میزبانی کدهای دسترسی داده ایجاد کنید و آنها را از سرویس Web API تفکیک نمایید. " برای برقراری امنیت در این سرویس چه باید کرد؟ اگر شخصی آدرس سرویس ما رو داشت و در خواست های را به آن ارسال کرد چگونه آن را نسبت به بقیه کاربران تمیز کند؟ چون در حقیقت webapi را در پروژه جدیدی در solution قرار دادیم و جدا هاست می شود. ممنون

نویسنده: محسن خان

تاریخ: ۱۱:۴۲ ۱۳۹۲/۱۱/۱۱

برای برقراری امنیت، تعیین هویت و اعتبارسنجی در وب API عموماً یا از [Forms authentication](#) استفاده می شود و یا از [ASP.NET Identity](#) . زیر ساخت آن یکی است و مشترک.

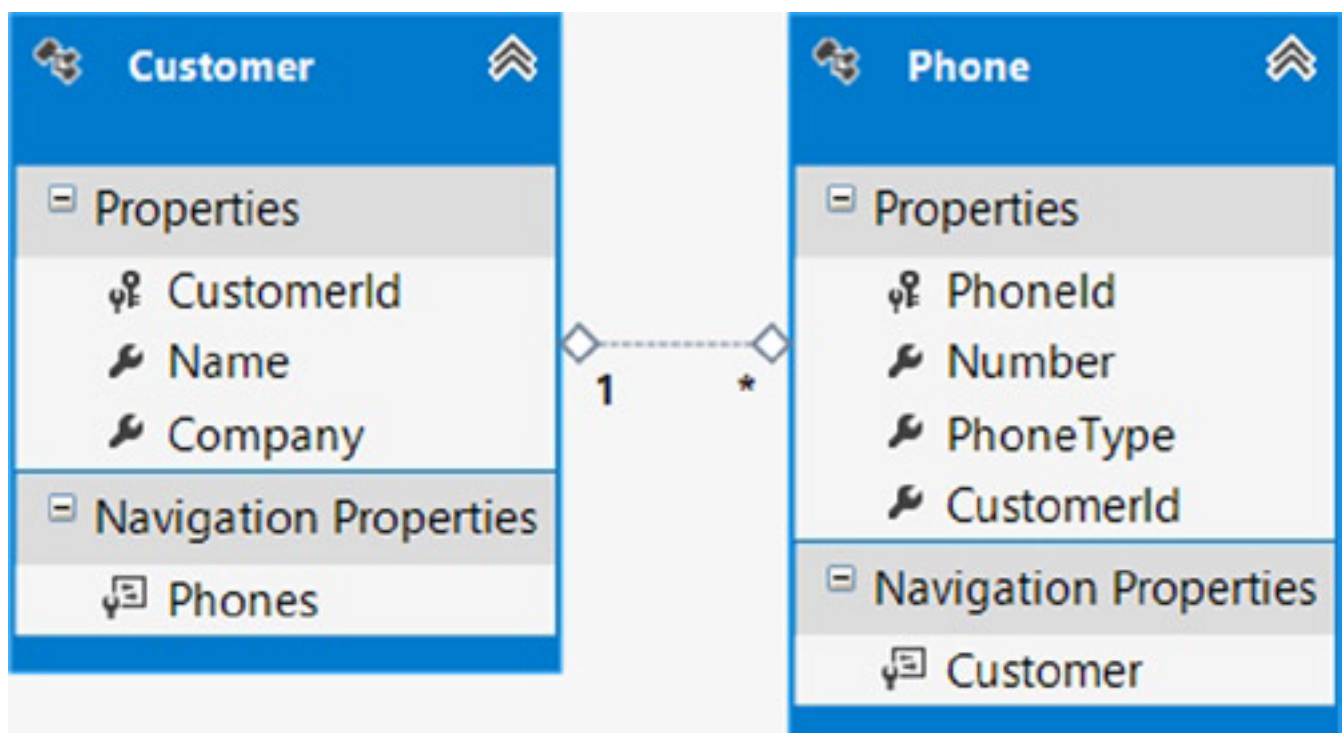
در [قسمت قبل](#) تشخیص تغییرات توسط Web API را بررسی کردیم. در این قسمت نگاهی به پیاده سازی Change-tracking در سمت کلاینت خواهیم داشت.

ردیابی تغییرات در سمت کلاینت توسط Web API

فرض کنید می‌خواهیم از سرویس‌های REST-based برای انجام عملیات CRUD روی یک Object graph استفاده کنیم. همچنین می‌خواهیم رویکردی در سمت کلاینت برای بروز رسانی کلاس موجودیت‌ها پیاده سازی کنیم که قابل استفاده مجدد (reusable) باشد. علاوه بر این دسترسی داده‌ها توسط مدل Code-First انجام می‌شود.

در مثال جاری یک اپلیکیشن کلاینت (برنامه کنسول) خواهیم داشت که سرویس‌های ارائه شده توسط پروژه Web API را فراخوانی می‌کند. هر پروژه در یک Solution مجزا قرار دارد، با این کار یک محیط n-Tier را شبیه سازی می‌کنیم.

مدل زیر را در نظر بگیرید.



همانطور که می‌بینید مدل مثال جاری مشتریان و شماره تماس آنها را ارائه می‌کند. می‌خواهیم مدل‌ها و کد دسترسی به داده‌ها را در یک سرویس Web API پیاده سازی کنیم تا هر کلاینتی که به HTTP دسترسی دارد بتواند از آن استفاده کند. برای ساخت سرویس مذکور مراحل زیر را دنبال کنید.

در ویتوال استودیو پروژه جدیدی از نوع ASP.NET Web Application بسازید و قالب پروژه را Web API انتخاب کنید. نام پروژه را به Recipe4.Service تغییر دهید.

کنترلر جدیدی با نام CustomerController به پروژه اضافه کنید.

کلاسی با نام BaseEntity ایجاد کنید و کد آن را مطابق لیست زیر تغییر دهید. تمام موجودیت‌ها از این کلاس پایه مشتق خواهند

شد که خاصیتی بنام TrackingState را به آنها اضافه می‌کند. کلاینت‌ها هنگام ویرایش آبجکت موجودیت‌ها باید این فیلد را مقدار دهی کنند. همانطور که می‌بینید این خاصیت از نوع TrackingState enum مشتق می‌شود. توجه داشته باشید که این خاصیت در دیتابیس ذخیره نخواهد شد. با پیاده سازی enum وضعیت ردیابی موجودیت‌ها بدین روش، وابستگی‌های EF را برای کلاینت از بین می‌بریم. اگر قرار بود وضعیت ردیابی را مستقیماً از EF به کلاینت پاس دهیم وابستگی‌های بخصوصی معرفی می‌شدند. کلاس DbContext اپلیکیشن در متد OnModelCreating به EF دستور می‌دهد که خاصیت TrackingState را به جدول موجودیت نگاشت نکند.

```
public abstract class BaseEntity
{
    protected BaseEntity()
    {
        TrackingState = TrackingState.Nochange;
    }

    public TrackingState TrackingState { get; set; }
}

public enum TrackingState
{
    Nochange,
    Add,
    Update,
    Remove,
}
```

کلاس‌های موجودیت Customer و PhoneNumber را ایجاد کنید و کد آنها را مطابق لیست زیر تغییر دهید.

```
public class Customer : BaseEntity
{
    public int CustomerId { get; set; }
    public string Name { get; set; }
    public string Company { get; set; }
    public virtual ICollection<Phone> Phones { get; set; }
}

public class Phone : BaseEntity
{
    public int PhoneId { get; set; }
    public string Number { get; set; }
    public string PhoneType { get; set; }
    public int CustomerId { get; set; }
    public virtual Customer Customer { get; set; }
}
```

با استفاده از NuGet Package Manager کتابخانه Entity Framework 6 را به پروژه اضافه کنید. کلاسی با نام Recipe4Context ایجاد کنید و کد آن را مطابق لیست زیر تغییر دهید. در این کلاس از یکی از قابلیت‌های جدید EF 6 بنام "Configuring Unmapped Base Types" استفاده کرده ایم. با استفاده از این قابلیت جدید هر موجودیت را طوری پیکربندی می‌کنیم که خاصیت TrackingState را نادیده بگیرند. برای اطلاعات بیشتر درباره این قابلیت EF 6 به [این لینک](#) مراجعه کنید.

```
public class Recipe4Context : DbContext
{
    public Recipe4Context() : base("Recipe4ConnectionString") { }
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Phone> Phones { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // Do not persist TrackingState property to data store
        // This property is used internally to track state of
        // disconnected entities across service boundaries.
        // Leverage the Custom Code First Conventions features from Entity Framework 6.
        // Define a convention that performs a configuration for every entity
        // that derives from a base entity class.
        modelBuilder.Types<BaseEntity>().Configure(x => x.Ignore(y => y.TrackingState));
        modelBuilder.Entity<Customer>().ToTable("Customers");
        modelBuilder.Entity<Phone>().ToTable("Phones");
    }
}
```

فایل Web.config پروژه را باز کنید و رشته اتصال زیر را به قسمت ConnectionStrings اضافه نمایید.

```
<connectionStrings>
  <add name="Recipe4ConnectionString"
    connectionString="Data Source=.;
    Initial Catalog=EFRecipes;
    Integrated Security=True;
    MultipleActiveResultSets=True"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

فایل Global.asax را باز کنید و کد زیر را به متد Application_Start اضافه نمایید. این کد بررسی Entity Framework Model Compatibility را غیرفعال می‌کند و به JSON serializer دستور می‌دهد که self-referencing loop خواص پیمایشی را نادیده بگیرد. این حلقه بدلیل رابطه bidirectional بین موجودیت‌های Customer و PhoneNumber بوجود می‌آید.

```
protected void Application_Start()
{
    // Disable Entity Framework Model Compatibility
    Database.SetInitializer<Recipe1Context>(null);
    // The bidirectional navigation properties between related entities
    // create a self-referencing loop that breaks Web API's effort to
    // serialize the objects as JSON. By default, Json.NET is configured
    // to error when a reference loop is detected. To resolve problem,
    // simply configure JSON serializer to ignore self-referencing loops.
    GlobalConfiguration.Configuration.Formatters.JsonFormatter
        .SerializerSettings.ReferenceLoopHandling =
            Newtonsoft.Json.ReferenceLoopHandling.Ignore;
    ...
}
```

کلاسی با نام EntityStateFactory بسازید و کد آن را مطابق لیست زیر تغییر دهید. این کلاس مقدار خاصیت TrackingState که به کلاینت‌ها ارائه می‌شود را به مقادیر متناظر کامپوننت‌های ردیابی EF تبدیل می‌کند.

```
public static EntityState Set(TrackingState trackingState)
{
    switch (trackingState)
    {
        case TrackingState.Add:
            return EntityState.Added;
        case TrackingState.Update:
            return EntityState.Modified;
        case TrackingState.Remove:
            return EntityState.Deleted;
        default:
            return EntityState.Unchanged;
    }
}
```

در آخر کد کنترلر CustomerController را مطابق لیست زیر بروز رسانی کنید.

```
public class CustomerController : ApiController
{
    // GET api/customer
    public IEnumerable<Customer> Get()
    {
        using (var context = new Recipe4Context())
        {
            return context.Customers.Include(x => x.Phones).ToList();
        }
    }

    // GET api/customer/5
    public Customer Get(int id)
    {
        using (var context = new Recipe4Context())
        {
            return context.Customers.Include(x => x.Phones).FirstOrDefault(x => x.CustomerId == id);
        }
    }
}
```

```

}

[ActionName("Update")]
public HttpResponseMessage UpdateCustomer(Customer customer)
{
    using (var context = new Recipe4Context())
    {
        // Add object graph to context setting default state of 'Added'.
        // Adding parent to context automatically attaches entire graph
        // (parent and child entities) to context and sets state to 'Added'
        // for all entities.
        context.Customers.Add(customer);
        foreach (var entry in context.ChangeTracker.Entries<BaseEntity>())
        {
            entry.State = EntityStateFactory.Set(entry.Entity.TrackingState);
            if (entry.State == EntityState.Modified)
            {
                // For entity updates, we fetch a current copy of the entity
                // from the database and assign the values to the original values
                // property from the Entry object. OriginalValues wrap a dictionary
                // that represents the values of the entity before applying changes.
                // The Entity Framework change tracker will detect
                // differences between the current and original values and mark
                // each property and the entity as modified. Start by setting
                // the state for the entity as 'Unchanged'.
                entry.State = EntityState.Unchanged;
                var databaseValues = entry.GetDatabaseValues();
                entry.OriginalValues.SetValues(databaseValues);
            }
        }

        context.SaveChanges();
    }

    return Request.CreateResponse(HttpStatusCode.OK, customer);
}

[HttpDelete]
[ActionName("Cleanup")]
public HttpResponseMessage Cleanup()
{
    using (var context = new Recipe4Context())
    {
        context.Database.ExecuteSqlCommand("delete from phones");
        context.Database.ExecuteSqlCommand("delete from customers");
        return Request.CreateResponse(HttpStatusCode.OK);
    }
}
}

```

حال اپلیکیشن کلاینت (برنامه کنسول) را می‌سازیم که از این سرویس استفاده می‌کند.

در ویژوال استودیو پروژه جدیدی از نوع Console Application بسازید و نام آن را به Recipe4.Client تغییر دهید. فایل program.cs را باز کنید و کد آن را مطابق لیست زیر تغییر دهید.

```

internal class Program
{
    private HttpClient _client;
    private Customer _bush, _obama;
    private Phone _whiteHousePhone, _bushMobilePhone, _obamaMobilePhone;
    private HttpResponseMessage _response;

    private static void Main()
    {
        Task t = Run();
        t.Wait();
        Console.WriteLine("\nPress <enter> to continue...");
        Console.ReadLine();
    }

    private static async Task Run()
    {
        var program = new Program();
        program.ServiceSetup();
        // do not proceed until clean-up completes
        await program.CleanupAsync();
    }
}

```

```

        program.CreateFirstCustomer();
        // do not proceed until customer is added
        await program.AddCustomerAsync();
        program.CreateSecondCustomer();
        // do not proceed until customer is added
        await program.AddSecondCustomerAsync();
        // do not proceed until customer is removed
        await program.RemoveFirstCustomerAsync();
        // do not proceed until customers are fetched
        await program.FetchCustomersAsync();
    }

    private void ServiceSetup()
    {
        // set up infrastructure for Web API call
        _client = new HttpClient { BaseAddress = new Uri("http://localhost:62799/") };
        // add Accept Header to request Web API content negotiation to return resource in JSON format
        _client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue
            ("application/json"));
    }

    private async Task CleanupAsync()
    {
        // call the cleanup method from the service
        _response = await _client.DeleteAsync("api/customer/cleanup/");
    }

    private void CreateFirstCustomer()
    {
        // create customer #1 and two phone numbers
        _bush = new Customer
        {
            Name = "George Bush",
            Company = "Ex President",
            // set tracking state to 'Add' to generate a SQL Insert statement
            TrackingState = TrackingState.Add,
        };
        _whiteHousePhone = new Phone
        {
            Number = "212 222-2222",
            PhoneType = "White House Red Phone",
            // set tracking state to 'Add' to generate a SQL Insert statement
            TrackingState = TrackingState.Add,
        };
        _bushMobilePhone = new Phone
        {
            Number = "212 333-3333",
            PhoneType = "Bush Mobile Phone",
            // set tracking state to 'Add' to generate a SQL Insert statement
            TrackingState = TrackingState.Add,
        };
        _bush.Phones.Add(_whiteHousePhone);
        _bush.Phones.Add(_bushMobilePhone);
    }

    private async Task AddCustomerAsync()
    {
        // construct call to invoke UpdateCustomer action method in Web API service
        _response = await _client.PostAsync("api/customer/updatecustomer/", _bush, new
        JsonMediaTypeFormatter());
        if (_response.IsSuccessStatusCode)
        {
            // capture newly created customer entity from service, which will include
            // database-generated Ids for all entities
            _bush = await _response.Content.ReadAsAsync<Customer>();
            _whiteHousePhone = _bush.Phones.FirstOrDefault(x => x.CustomerId == _bush.CustomerId);
            _bushMobilePhone = _bush.Phones.FirstOrDefault(x => x.CustomerId == _bush.CustomerId);
            Console.WriteLine("Successfully created Customer {0} and {1} Phone Numbers(s)",
                _bush.Name, _bush.Phones.Count);
            foreach (var phoneType in _bush.Phones)
            {
                Console.WriteLine("Added Phone Type: {0}", phoneType.PhoneType);
            }
        }
        else
            Console.WriteLine("{0} ({1})", (int)_response.StatusCode, _response.ReasonPhrase);
    }

    private void CreateSecondCustomer()
    {
        // create customer #2 and phone numbers
        _obama = new Customer
    }

```

```

    {
        Name = "Barack Obama",
        Company = "President",
        // set tracking state to 'Add' to generate a SQL Insert statement
        TrackingState = TrackingState.Add,
    };
    _obamaMobilePhone = new Phone
    {
        Number = "212 444-4444",
        PhoneType = "Obama Mobile Phone",
        // set tracking state to 'Add' to generate a SQL Insert statement
        TrackingState = TrackingState.Add,
    };
    // set tracking state to 'Modified' to generate a SQL Update statement
    _whiteHousePhone.TrackingState = TrackingState.Update;
    _obama.Phones.Add(_obamaMobilePhone);
    _obama.Phones.Add(_whiteHousePhone);
}

private async Task AddSecondCustomerAsync()
{
    // construct call to invoke UpdateCustomer action method in Web API service
    _response = await _client.PostAsync("api/customer/updatecustomer/", _obama, new
JsonMediaTypeFormatter());
    if (_response.IsSuccessStatusCode)
    {
        // capture newly created customer entity from service, which will include
        // database-generated Ids for all entities
        _obama = await _response.Content.ReadAsAsync<Customer>();
        _whiteHousePhone = _bush.Phones.FirstOrDefault(x => x.CustomerId == _obama.CustomerId);
        _bushMobilePhone = _bush.Phones.FirstOrDefault(x => x.CustomerId == _obama.CustomerId);
        Console.WriteLine("Successfully created Customer {0} and {1} Phone Numbers(s)",
            _obama.Name, _obama.Phones.Count);
        foreach (var phoneType in _obama.Phones)
        {
            Console.WriteLine("Added Phone Type: {0}", phoneType.PhoneType);
        }
    }
    else
        Console.WriteLine("{0} ({1})", (int)_response.StatusCode, _response.ReasonPhrase);
}

private async Task RemoveFirstCustomerAsync()
{
    // remove George Bush from underlying data store.
    // first, fetch George Bush entity, demonstrating a call to the
    // get action method on the service while passing a parameter
    var query = "api/customer/" + _bush.CustomerId;
    _response = _client.GetAsync(query).Result;

    if (_response.IsSuccessStatusCode)
    {
        _bush = await _response.Content.ReadAsAsync<Customer>();
        // set tracking state to 'Remove' to generate a SQL Delete statement
        _bush.TrackingState = TrackingState.Remove;
        // must also remove bush's mobile number -- must delete child before removing parent
        foreach (var phoneType in _bush.Phones)
        {
            // set tracking state to 'Remove' to generate a SQL Delete statement
            phoneType.TrackingState = TrackingState.Remove;
        }
        // construct call to remove Bush from underlying database table
        _response = await _client.PostAsync("api/customer/updatecustomer/", _bush, new
JsonMediaTypeFormatter());
        if (_response.IsSuccessStatusCode)
        {
            Console.WriteLine("Removed {0} from database", _bush.Name);
            foreach (var phoneType in _bush.Phones)
            {
                Console.WriteLine("Remove {0} from data store", phoneType.PhoneType);
            }
        }
        else
            Console.WriteLine("{0} ({1})", (int)_response.StatusCode, _response.ReasonPhrase);
    }
    else
    {
        Console.WriteLine("{0} ({1})", (int)_response.StatusCode, _response.ReasonPhrase);
    }
}

```

```
private async Task FetchCustomersAsync()
{
    // finally, return remaining customers from underlying data store
    _response = await _client.GetAsync("api/customer/");
    if (_response.IsSuccessStatusCode)
    {
        var customers = await _response.Content.ReadAsAsync<IEnumerable<Customer>>();
        foreach (var customer in customers)
        {
            Console.WriteLine("Customer {0} has {1} Phone Numbers(s)",
                customer.Name, customer.Phones.Count());
            foreach (var phoneType in customer.Phones)
            {
                Console.WriteLine("Phone Type: {0}", phoneType.PhoneType);
            }
        }
    }
    else
    {
        Console.WriteLine("{0} ({1})", (int)_response.StatusCode, _response.ReasonPhrase);
    }
}
}
```

در آخر کلاس های Customer, Phone و BaseEntity را به پروژه کلاینت اضافه کنید. چنین کدهایی بهتر است در لایه مجزایی قرار گیرند و بین لایه های مختلف اپلیکیشن به اشتراک گذاشته شوند.

اگر اپلیکیشن کلاینت را اجرا کنید با خروجی زیر مواجه خواهید شد.

```
Successfully created Customer Geroge Bush and 2 Phone Numbers(s)
Added Phone Type: White House Red Phone
Added Phone Type: Bush Mobile Phone
Successfully created Customer Barrack Obama and 2 Phone Numbers(s)
Added Phone Type: Obama Mobile Phone
Added Phone Type: White House Red Phone
Removed Geroge Bush from database
Remove Bush Mobile Phone from data store
Customer Barrack Obama has 2 Phone Numbers(s)
Phone Type: White House Red Phone
Phone Type: Obama Mobile Phone
```

شرح مثال جاری

با اجرای اپلیکیشن Web API شروع کنید. این اپلیکیشن یک MVC Web Controller دارد که پس از اجرا شما را به صفحه خانه هدایت می کند. در این مرحله سایت در حال اجرا است و سرویس ها قابل دسترسی هستند.

سپس اپلیکیشن کنسول را باز کنید و روی خط اول کد فایل program.cs یک breakpoint قرار داده و آن را اجرا کنید. ابتدا آدرس سرویس را نگاشت می کنیم و از سرویس درخواست می کنیم که اطلاعات را با فرمت JSON بازگرداند.

سپس توسط متد DeleteAsync که روی آبجکت HttpClient تعریف شده است اکشن متد Cleanup را روی سرویس فراخوانی می کنیم. این فراخوانی تمام داده های پیشین را حذف می کند.

در قدم بعدی یک مشتری به همراه دو شماره تماس می سازیم. توجه کنید که برای هر موجودیت مشخصا خاصیت TrackingState

را مقدار دهی می‌کنیم تا کامپوننت‌های Change-tracking در EF عملیات لازم SQL برای هر موجودیت را تولید کنند.

سپس توسط متد PostAsync که روی آبجکت HttpClient تعریف شده اکشن متد UpdateCustomer را روی سرویس فراخوانی می‌کنیم. اگر به این اکشن متد یک breakpoint اضافه کنید خواهید دید که موجودیت مشتری را بعنوان یک پارامتر دریافت می‌کند و آن را به context جاری اضافه می‌نماید. با اضافه کردن موجودیت به کانتکست جاری کل object graph اضافه می‌شود و EF شروع به ردیابی تغییرات آن می‌کند. دقت کنید که آبجکت موجودیت باید Add شود و نه Attach.

قدم بعدی جالب است، هنگامی که از خاصیت DbChangeTracker استفاده می‌کنیم. این خاصیت روی آبجکت context تعریف شده و یک `IEnumerable<DbEntityEntry>` را با نام Entries ارائه می‌کند. در اینجا بسادگی نوع پایه `EntityType` را تنظیم می‌کنیم. این کار به ما اجازه می‌دهد که در تمام موجودیت‌هایی که از نوع `BaseEntity` هستند پیمایش کنیم. اگر بیاد داشته باشید این کلاس، کلاس پایه تمام موجودیت‌ها است. در هر مرحله از پیمایش (iteration) با استفاده از کلاس `EntityStateFactory` مقدار خاصیت `TrackingState` را به مقدار متناظر در سیستم ردیابی EF تبدیل می‌کنیم. اگر کلاینت مقدار این فیلد را به `Modified` تنظیم کرده باشد پردازش بیشتری انجام می‌شود. ابتدا وضعیت موجودیت را از `Modified` به `Unchanged` تغییر می‌دهیم. سپس مقادیر اصلی را با فراخوانی متد `GetDatabaseValues` روی آبجکت `Entry` از دیتابیس دریافت می‌کنیم. فراخوانی این متد مقادیر موجود در دیتابیس را برای موجودیت جاری دریافت می‌کند. سپس مقادیر بدست آمده را به کلکسیون `OriginalValues` اختصاص می‌دهیم. پشت پرده، کامپوننت‌های EF Change-tracking بصورت خودکار تفاوت‌های مقادیر اصلی و مقادیر ارسالی را تشخیص می‌دهند و فیلدهای مربوطه را با وضعیت `Modified` علامت گذاری می‌کنند. فراخوانی‌های بعدی متد `SaveChanges` تنها فیلدهایی که در سمت کلاینت تغییر کرده اند را بروز رسانی خواهد کرد و نه تمام خواص موجودیت را.

در اپلیکیشن کلاینت عملیات افزودن، بروز رسانی و حذف موجودیت‌ها توسط مقداردهی خاصیت `TrackingState` را نمایش داده ایم.

متد `UpdateCustomer` در سرویس ما مقادیر `TrackingState` را به مقادیر متناظر EF تبدیل می‌کند و آبجکت‌ها را به موتور change-tracking ارسال می‌کند که نهایتاً منجر به تولید دستورات لازم SQL می‌شود.

نکته: در اپلیکیشن‌های واقعی بهتر است کد دسترسی داده‌ها و مدل‌های دامنه را به لایه مجزایی منتقل کنید. همچنین پیاده سازی فعلی change-tracking در سمت کلاینت می‌تواند توسعه داده شود تا با انواع جنریک کار کند. در این صورت از نوشتن مقادیر زیادی کد تکراری جلوگیری خواهید کرد و از یک پیاده سازی می‌توانید برای تمام موجودیت‌ها استفاده کنید.

نظرات خوانندگان

نویسنده: امیرحسین

تاریخ: ۱۳۹۲/۱۱/۱۰ ۰:۴

میشه در مورد async کمی توضیح بدین که چرا و به چه دلیلی استفاده شده ؟

نویسنده: آرمین ضیاء

تاریخ: ۱۳۹۲/۱۱/۱۰ ۱:۲۵

الزامی به استفاده از قابلیت های async نیست، اما توصیه میشه در مواقعی که امکانش هست و مناسب است از این قابلیت استفاده کنید. لزوما کارایی (performance) بهتری بدست نمیاری ولی مسلما تجربه کاربری بهتری خواهید داشت. عملیاتی که بصورت async اجرا میشن ریسمان جاری (current thread) رو قفل نمی کنند، بنابراین اجرای اپلیکیشن ادامه پیدا می کنه و پاسخگویی بهتری بدست میارید. برای مطالعه بیشتر به [این لینک](#) مراجعه کنید.

مطالعه بیشتر

[Using Asynchronous Methods in ASP.NET 4.5](#)

[Async and Await](#)

طراحی Uri در Restful API

Uri بخش اصلی و راه ارتباطی API شما با توسعه دهنده است. بنابراین طراحی یک ساختار مناسب و یکپارچه برای Uri ها دارای اهمیت زیادی است.

Uri پایه API خود را ساده و خوانا ، حفظ کنید . داشتن یک Uri پایه ساده استفاده از API را آسان کرده و خوانایی آن را بالا میبرد و باعث می‌شود که توسعه دهنده برای استفاده از آن نیاز کمتری به مراجعه به مستندات داشته باشد. پیشنهاد می‌شود که برای هر منبع تنها دو Uri پایه وجود داشته باشد . یکی برای مجموعه ای از منبع موردنظر و دیگری برای یک واحد مشخص از آن منبع . برای مثال اگر منبع موردنظر ما کتاب باشد ، خواهیم داشت :

.../books

برای مجموعه‌ی کتابها و

.../books/1001

برای کتابی با شناسه 1001

استفاده از این روش یک مزیت دیگر هم به همراه دارد و آن دور کردن افعال از Uri ها است.

بسیاری در زمان طراحی Uri ها و در نامگذاری از فعل‌ها استفاده می‌کنند. برای هر منبعی که مدلسازی می‌کنید هیچ وقت نمی‌توانید آن را به تنهایی و جداافتاده در نظر بگیرید. بلکه همیشه منابع مرتبطی وجود دارند که باید در نظر گرفته شوند. در مثال کتاب می‌توان منابعی مثل نویسنده ، ناشر ، موضوع و ... را بیان کرد. حالا سعی کنید به تمام Uri هایی که برای پوشش دادن تمام درخواست‌های مربوط به منبع کتاب نیاز داریم فکر کنید . احتمالا به چیزی شبیه این می‌رسیم :

.../getAllBooks
.../getBook
.../newBook
.../getNewBooksSince
.../getComputerBooks
.../BooksNotPublished
.../UpdateBookPriceTo
.../bookForPublisher
.../GetLastBooks
.../DeleteBook
...

خیلی زود یک لیست طولانی از Uri ها خواهید داشت که به علت نداشتن یک الگوی ثابت و مشخص استفاده از API شما را واقعا سخت می‌کند.

پس حالا این درخواست‌های متنوع را چطور با دو Ur1 اصلی انجام دهیم ؟
 1- از افعال Http برای کار کردن بر روی منابع استفاده کنید . با استفاده از افعال Http شامل POST ، GET ، PUT و DELETE و دو Ur1 اصلی ، یک مجموعه‌ی مناسب از عملیات‌ها در دسترس توسعه دهنده خواهد بود . به جدول زیر نگاه کنید .

منبع	POST Create	GET Read	PUT Update	DELETE Delete
/books	ثبت کتاب جدید	لیست کتابها	بروزرسانی کلی کتابها	حذف تمام کتابها
/books/1001	خطا	نمایش کتاب ۱۰۰۱	اگر وجود داشته باشد بروزرسانی وگرنه خطا	حذف کتاب ۱۰۰۱

توسعه دهندگان احتمالا نیازی به این جدول برای درک اینکه API چطور کار می‌کند نخواهند داشت.

2- با استفاده از نکته قبلی بخشی از Ur1 های بالا حذف خواهند شد. اما هنوز با روابط بین منابع چکار کنیم؟ منابع تقریبا همیشه دارای روابطی با دیگر منابع هستند . یک روش ساده برای بیان این روابط در API چیست ؟ به مثال کتاب برمیگردیم. کتاب‌ها دارای نویسنده هستند. اگر بخواهیم کتاب‌های یک نویسنده را برگردانیم چه باید بکنیم؟ با استفاده از Ur1 های پایه و افعال Http می‌توان اینکار را انجام داد. یکی از ساختارهای ممکن این است :

GET .../authors/1001/books

اگر بخواهیم یک کتاب جدید به کتابهای این نویسنده اضافه کنیم :

POST .../authors/1001/books

و حدس زدن اینکه برای حذف کتابهای این نویسنده چه باید کرد ، سخت نیست .

3- بیشتر API ها دارای پیچیدگی‌های بیشتری نسبت به Ur1 اصلی یک منبع هستند . هر منبع مشخصات و روابط متنوعی دارد که قابل جستجو کردن، مرتب سازی، بروزرسانی و تغییر هستند. Ur1 اصلی را ساده نگه دارید و این پیچیدگی‌ها را به کوئری استرینگ منتقل کنید.

برای برگرداندن تمام کتابهای با قیمت پنج هزار تومان با قطع جیبی که دارای امتیاز 8 به بالا هستند از کوئری زیر می‌شود استفاده کرد :

GET .../books?price=5000&size=pocket&score=8

و البته فراموش نکنید که لیستی از فیلدهای مجاز را در مستندات خود ارائه کنید.

4 - گفتیم که بهتر است افعال را از URl ها خارج کنیم . ولی در مواردی که درخواست ارسال شده در مورد یک منبع نیست چطور؟ مواردی مثل محاسبه مالیات پرداختی یا هزینه بیمه ، جستجو در کل منابع ، ترجمه یک عبارت یا تبدیل واحدها . هیچکدام از اینها ارتباطی با یک منبع خاص ندارند. در این موارد بهتر است از افعال استفاده شود. و حتما در مستندات خود ذکر کنید که در این موارد از افعال استفاده می‌شود.

```
.../convert?value=25&from=px&to=em  
.../translate?term=web&from=en&to=fa
```

5 - استفاده از اسامی جمع یا مفرد

با توجه به ساختاری که تا اینجا طراحی کرده ایم بکاربردن اسامی جمع بامعنا تر و خوانا تر است. اما مهمتر از روشی که بکار می‌برید ، اجتناب از بکاربردن هر دو روش با هم است ، اینکه در مورد یک منبع از اسم مفرد و در مورد دیگری از اسم جمع استفاده کنید . یکدستی API را حفظ کنید و به توسعه دهنده کمک کنید راحت تر API شما را یاد بگیرد.

6- استفاده از نام‌های عینی به جای نام‌های کلی و انتزاعی

API ی را در نظر بگیرید که محتوایی را در فرمت‌های مختلف ارائه می‌دهد. بلاگ ، ویدئو ، اخبار و حالا فرض کنید این API منابع را در بالاتری سطح مدسازی کرده باشد مثل items/ یا assets/ . درک کردن محتوای این API و کاری که می‌توان با این API انجام داد برای توسعه دهنده سخت است . خیلی راحت تر و مفیدتر است که منابع را در قالب بلاگ ، اخبار ، ویدئو مدسازی کنیم .

Media Type یا **MIME Type** نشان دهنده فرمت یک مجموعه داده است. در HTTP، مدیا تایپ بیان کننده فرمت message body یک درخواست / پاسخ است و به دریافت کننده اعلام می‌کند که چگونه باید پیام را بخواند. محل استاندارد تعیین Mime Type در هدر Content-Type است. درخواست کننده می‌تواند با استفاده از هدر Accept لیستی از MimeTypes های قابل قبول را به عنوان پاسخ، به سرور اعلام کند.

Response Headers

[view source](#)

```
Cache-Control public, max-age=2542200
Connection keep-alive
Content-Disposition attachment; filename=d79319c858e147f281eb2d0eebba7fc6.jpg
Content-Length 7387
Content-Type image/jpeg
Date Sat, 03 May 2014 16:52:31 GMT
Expires Mon, 02 Jun 2014 03:02:31 GMT
Last-Modified Sat, 03 May 2014 03:02:31 GMT
Server Microsoft-IIS/6.0
Vary *
X-Powered-By ASP.NET
```

Request Headers

[view source](#)

```
Accept image/png,image/*;q=0.8,*/*;q=0.5
Accept-Encoding gzip, deflate
Accept-Language en-US,en;q=0.5
Connection keep-alive
Cookie __utma=95334921.1626186387.1386794008.1399109809.1399127295.237; __utms=95334921
```

از [Asp.net Web API](#) MimeType برای تعیین نحوه serialize یا deserialize کردن محتوای دریافتی / ارسالی استفاده می‌کند.

Web API **MediaTypeFormatter** برای خواندن/درج پیام در بدنه درخواست/پاسخ از [MediaTypeFormmater](#) ها استفاده می‌کند. اینها کلاس‌هایی هستند که نحوه‌ی Serialize کردن و deserialize کردن اطلاعات به فرمت‌های خاص را تعیین می‌کنند. Web API به صورت توکار دارای formatter هایی برای نوع‌های XML ، JSON ، BSON و Form-UrlEncoded می‌باشد. همه این‌ها کلاس پایه MediaTypeFormatter را پیاده سازی می‌کنند.

مسئله

یک پروژه Web API بسازید و view model زیر را در آن تعریف کنید:

```
public class NewProduct
{
    [Required]
    public string Name { get; set; }

    public double Price { get; set; }

    public byte[] Pic { get; set; }
}
```

همانطور که می بینید یک فیلد از نوع byte[] برای تصویر محصول در نظر گرفته شده است.
حالا یک کنترلر API ساخته و اکشنی برای دریافت اطلاعات محصول جدید از کاربر می نویسیم :

```
public class ProductsController : ApiController
{
    [HttpPost]
    public HttpResponseMessage PostProduct(NewProduct model)
    {
        if (ModelState.IsValid)
        {
            // ثبت محصول

            return new HttpResponseMessage(HttpStatusCode.Created);
        }
        return Request.CreateErrorResponse(HttpStatusCode.BadRequest, ModelState);
    }
}
```

و یک صفحه html به نام index.html که حاوی یک فرم برای ارسال اطلاعات باشد :

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h1>ساخت MediaTypeFormatter برای Multipart/form-data</h1>
    <h2>محصول جدید</h2>

    <form id="newProduct" method="post" action="/api/products" enctype="multipart/form-data">
        <div>
            <label for="name">نام محصول</label>
            <input type="text" id="name" name="name" />
        </div>
        <div>
            <label for="price">قیمت</label>
            <input type="number" id="price" name="price" />
        </div>
        <div>
            <label for="pic">تصویر</label>
            <input type="file" id="pic" name="pic" />
        </div>
        <div>
            <button type="submit">ثبت</button>
        </div>
    </form>
</body>
</html>
```

زمانی که فرم حاوی فایلی برای آپلود باشد مشخصه enctype باید برابر با [Multipart/form-data](#) مقداردهی شود تا اطلاعات فایل به درستی کد شوند. در زمان ارسال فرم Content-type برابر با Multipart/form-data و فرمت اطلاعات درخواست ارسالی به شکل زیر خواهد بود :

برای ساختن یک MediaTypeFormatter یکی از 2 کلاس [MediaTypeFormatter](#) یا [BufferedMediaTypeFormatter](#) را باید پیاده سازی کنیم. تفاوت این دو در این است که BufferedMediaTypeFormatter برخلاف MediaTypeFormatter از متدهای synchronous استفاده می‌کند.

پیاده سازی :

یک کلاس به نام MultiPartMediaTypeFormatter می‌سازیم و کلاس MediaTypeFormatter را به عنوان کلاس پایه آن قرار می‌دهیم.

```
public class MultiPartMediaTypeFormatter : MediaTypeFormatter
{
    ...
}
```

ابتدا در تابع سازنده کلاس فرمت‌هایی که می‌خواهیم توسط این کلاس خوانده شوند را تعریف می‌کنیم :

```
public MultiPartMediaTypeFormatter()
{
    SupportedMediaTypes.Add(new MediaTypeHeaderValue("multipart/form-data"));
}
```

در اینجا multipart/form-data را به عنوان تنها نوع مجاز تعریف کرده ایم.

سپس با پیاده سازی توابع CanWriteType و CanReadType مربوط به کلاس MediaTypeFormatter مشخص می‌کنیم که چه مدل‌هایی را می‌توان توسط این کلاس serialize / deserialize کرد. در اینجا چون می‌خواهیم این کلاس محدود به یک مدل خاص نباشد، از یک اینترفیس برای شناسایی کلاس‌های مجاز استفاده می‌کنیم.

```
public interface INeedMultiPartMediaTypeFormatter
{
}
```

و آنرا به کلاس newProduct اضافه می‌کنیم :

```
public class NewProduct : INeedMultiPartMediaTypeFormatter
{
    ...
}
```

از آنجا که تنها نیاز به خواندن اطلاعات داریم و قصد نوشتن نداریم، در متد CanWriteType مقدار false را برمی‌گردانیم.

```
public override bool CanReadType(Type type)
{
    return typeof(INeedMultiPartMediaTypeFormatter).IsAssignableFrom(type);
}

public override bool CanWriteType(Type type)
{
    return false;
}
```

و اما تابع ReadFromStreamAsync که کار خواندن محتوای ارسال شده و باید کردن آنها به پارامترها را برعهده دارد

```
public async override Task<object> ReadFromStreamAsync(Type type, Stream stream, HttpContent content,
IFormatterLogger formatterLogger)
```

که در آن پارامتر type مربوط به مدل مشخص شده به عنوان پارامتر اکشن (NewProduct) است و پارامتر content محتوای

درخواست را در خود دارد.

ابتدا محتوای ارسال شده را خوانده و اطلاعات فرم را استخراج می‌کنیم و از طرف دیگر با استفاده از کلاس Activator یک نمونه از مدل جاری را ساخته و لیست property های آنرا استخراج می‌کنیم.

```
MultipartMemoryStreamProvider provider = await content.ReadAsMultipartAsync();
IEnumerable<HttpContent> formData = provider.Contents.AsEnumerable();

var modelInstance = Activator.CreateInstance(type);
IEnumerable<PropertyInfo> properties = type.GetProperties();
```

سپس در یک حلقه به ترتیب برای هر property متعلق به مدل، در میان اطلاعات فرم جستجو می‌کنیم. برای پیدا کردن اطلاعات متناظر با هر property در هدر Content-Disposition که در بالا توضیح داده شد، به دنبال فیلد همنام با property می‌گردیم.

```
foreach (PropertyInfo prop in properties)
{
    var propName = prop.Name.ToLower();
    var propType = prop.PropertyType;

    var data = formData.FirstOrDefault(d =>
        d.Headers.ContentDisposition.Name.ToLower().Contains(propName));
```

در صورتی که فیلدی وجود داشته باشد کار را ادامه می‌دهیم.

گفتیم که هر فیلد یک هدر، Content-Type هم می‌تواند داشته باشد. این هدر به صورت پیش فرض معادل text/plain است و برای فیلدهای عادی قرار داده نمی‌شود. در این مثال چون فقط یک فیلد غیر رشته ای داریم فرض را بر این گرفته ایم که در صورت وجود Content-Type، فیلد مربوط به تصویر است. در صورتیکه ContentType وجود داشته باشد، محتوای فیلد را به شکل Stream خوانده به [byte\[\]](#) تبدیل و با استفاده از متد SetValue در property مربوطه قرار می‌دهیم.

```
if (data != null)
{
    if (data.Headers.ContentType != null)
    {
        using (var fileStream = await data.ReadAsStreamAsync())
        {
            using (MemoryStream ms = new MemoryStream())
            {
                fileStream.CopyTo(ms);
                prop.SetValue(modelInstance, ms.ToArray());
            }
        }
    }
}
```

در صورتی که Content-Type غایب باشد بدین معنی است که محتوای فیلد از نوع رشته است (عدد ، تاریخ ، guid ، رشته) و باید به نوع مناسب تبدیل شود. ابتدا آن را به صورت یک رشته می‌خوانیم و با استفاده از Convert.ChangeType آنرا به نوع مناسب تبدیل می‌کنیم و در property متناظر قرار می‌دهیم.

```
if (data != null)
{
    if (data.Headers.ContentType != null)
    {
        //...
    }
    else
    {
        string rawVal = await data.ReadAsStringAsync();
        object val = Convert.ChangeType(rawVal, propType);

        prop.SetValue(modelInstance, val);
    }
}
```

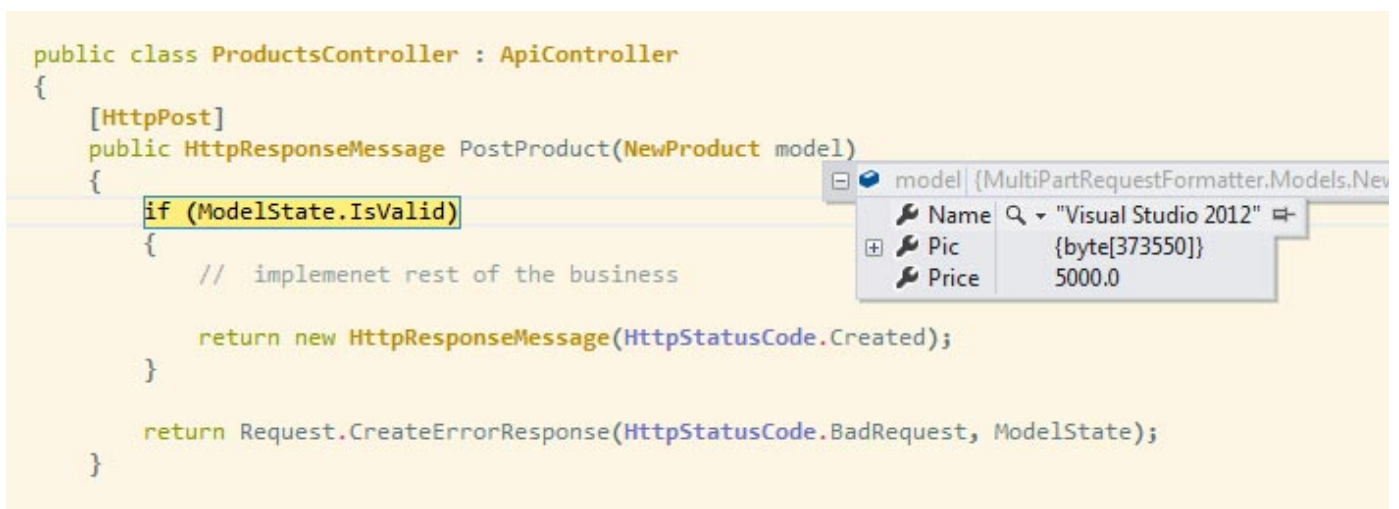

و در نهایت نمونه ساخته شده از مدل را برگشت می‌دهیم.

```
return modelInstance;
```

برای فعال کردن این Formatter باید آنرا به لیست formatters های web api اضافه بکنیم. فایل WebApiConfig در App_Start را باز کرده و خط زیر را به آن اضافه می‌کنیم:

```
config.Formatters.Add(new MultiPartMediaTypeFormatter());
```

حال اگر مجدداً فرم را به سرور ارسال کنیم، با پیام خطایی، مواجه نشده و عمل binding با موفقیت انجام می‌گیرد.



The screenshot shows a C# code snippet for a `ProductsController` inheriting from `ApiController`. The `PostProduct` method is decorated with `[HttpPost]` and takes a `NewProduct` model. The code checks `ModelState.IsValid` and either returns a `HttpStatusCode.Created` response or a `BadRequest` response. A tooltip for the `model` parameter shows the following data contract:

model {MultiPartRequestFormatter.Models.NewProduct}	
Name	"Visual Studio 2012"
Pic	{byte[373550]}
Price	5000.0

زمانیکه از Template های پیش فرض تدارک دیده شده در VS.Net برای اپلیکیشن های وب خود استفاده می کنید، وب اپلیکیشن و سرور با هم یکپارچه هستند و تحت IIS اجرا می شوند. به وسیله [Owin](#) می توان این دو مورد را بدون وابستگی به IIS به صورت مجزا اجرا کرد. در این پست قصد داریم سرویس های Web Api را در قالب یک Windows Service با استفاده از کتابخانه ی [TopShelf](#) هاست نماییم.

پیش نیاز ها:

« [Owin چیست](#) »

« [تبدیل برنامه های کنسول ویندوز به سرویس ویندوز ان تی](#) »

برای شروع یک برنامه Console Application ایجاد کرده و اقدام به نصب پکیج های زیر نمایید:

```
Install-Package Microsoft.AspNet.WebApi.OwinSelfHost
Install-Package TopShelf
```

حال یک کلاس Startup برای پیاده سازی Configuration های مورد نیاز ایجاد می کنیم
در این قسمت می توانید تنظیمات زیر را پیاده سازی نمایید:

«سیستم Routing»

«تنظیم Dependency Resolver برای تزریق وابستگی کنترلرهای Web Api»

«تنظیمات hub های SignalR (در حال حاضر SignalR به صورت پیش فرض نیاز به Owin برای اجرا دارد):»

«رجیستر کردن Owin Middleware های نوشته شده»

«تغییر در Asp.Net Pipeline»

«و...»

```
public class Startup
{
    public void Configuration(IAppBuilder appBuilder)
    {
        HttpConfiguration config = new HttpConfiguration();
        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
        appBuilder.UseWebApi(config);
    }
}
```

* به صورت پیش فرض نام این کلاس باید Startup و نام متد آن نیز باید Configuration باشد.

در این مرحله یک کنترلر Api به صورت زیر به پروژه اضافه نمایید:

```
public class ValuesController : ApiController
{
    public IEnumerable<string> Get()
    {
        return new string[] { "value1", "value2" };
    }

    public string Get(int id)
    {
    }
```

```

        return "value";
    }

    public void Post([FromBody]string value)
    {
    }

    public void Put(int id, [FromBody]string value)
    {
    }
}

```

کلاسی به نام ServiceHost ایجاد نمایید و کدهای زیر را در آن کپی کنید:

```

public class ServiceHost
{
    private IDisposable webApp;

    public static string BaseAddress
    {
        get
        {
            return "http://localhost:8000/";
        }
    }

    public void Start()
    {
        webApp = WebApp.Start<Startup>(BaseAddress);
    }

    public void Stop()
    {
        webApp.Dispose();
    }
}

```

واضح است که متد Start در کلاس بالا با استفاده از متد Start کلاس WebApp، سرویس های Web Api را در آدرس مورد نظر هاست خواهد کرد. با فراخوانی متد Stop این سرویس ها نیز dispose خواهند شد. در مرحله آخر باید شروع و توقف سرویس ها را تحت کنترل کلاس HostFactory کتابخانه TopShelf در آوریم. برای این کار کفایست کلاسی به نام ServiceHostFactory ایجاد کرده و کدهای زیر را در آن کپی نمایید:

```

public class ServiceHostFactory
{
    public static void Run()
    {
        HostFactory.Run( config =>
        {
            config.SetServiceName( "ApiServices" );
            config.SetDisplayName( "Api Services" );
            config.SetDescription( "No Description" );

            config.RunAsLocalService();

            config.Service<ServiceHost>( cfg =>
            {
                cfg.ConstructUsing( builder => new ServiceHost() );

                cfg.WhenStarted( service => service.Start() );
                cfg.WhenStopped( service => service.Stop() );
            } );
        } );
    }
}

```

توضیح کدهای بالا:

ابتدا با فراخوانی متد Run سرویس مورد نظر اجرا خواهد شد. تنظیمات نام سرویس و نام مورد نظر جهت نمایش و همچنین توضیحات در این قسمت انجام می گیرد.

با استفاده از متد ConstructUsing عملیات و هله سازی از سرویس انجام خواهد گرفت. در پایان نیز متد Start و Stop کلاس ServiceHost، به عنوان عملیات شروع و پایان سرویس ویندوز مورد نظر تعیین شد.

حال اگر در فایل Program پروژه، دستور زیر را فراخوانی کرده و برنامه را ایجاد کنید خروجی زیر قابل مشاهده است.

```
ServiceHostFactory.Run();
```

```
Configuration Result:
[Success] Name ApiService
[Success] DisplayName Api Services ]
[Success] Description No Description
[Success] ServiceName ApiService
Topshelf v3.1.122.0, .NET Framework v4.0.30319.18408
```

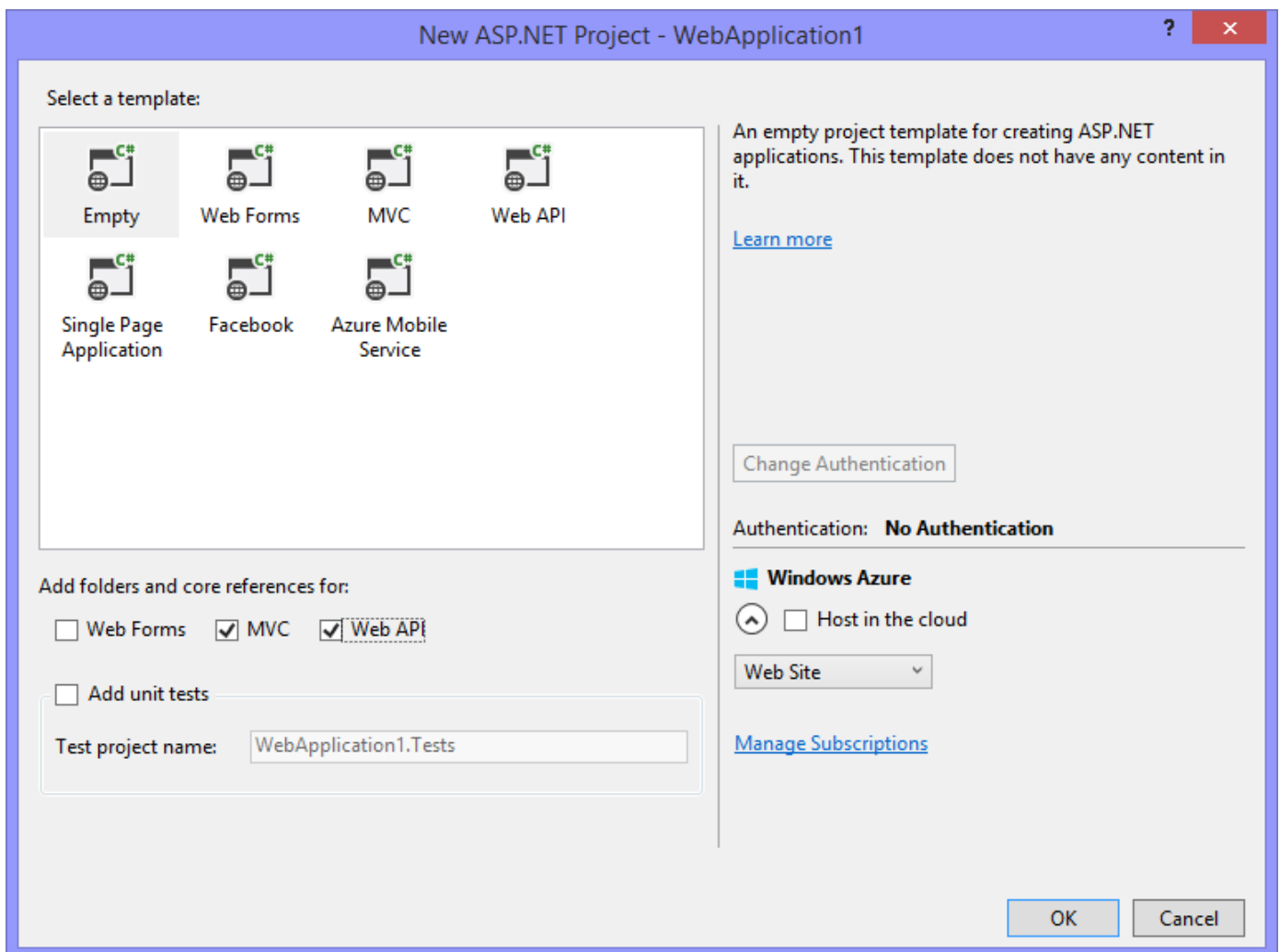
در حالیکه سرویس مورد نظر در حال اجراست، Browser را گشوده و آدرس `http://localhost:8000/api/values/get` را در AddressBar وارد کنید. خروجی زیر را مشاهده خواهید کرد:

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<ArrayOfstring xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.microsoft.com/2003/10/Serialization/Arrays">
  <string>value1</string>
  <string>value2</string>
</ArrayOfstring>
```

فریم ورک ASP.NET Web API صرفاً برای ساخت سرویس‌های ساده‌ای که می‌شناسیم، نیست و در واقع مدل جدیدی برای برنامه نویسی HTTP است. کارهای بسیار زیادی را می‌توان توسط این فریم ورک انجام داد که در این مقاله به یکی از آنها می‌پردازم. فرض کنید می‌خواهیم یک فایل ویدیو را بصورت Asynchronous به کلاینت ارسال کنیم.

ابتدا پروژه جدیدی از نوع ASP.NET Web Application بسازید و قالب آن را MVC + Web API انتخاب کنید.



ابتدا به فایل `WebApiConfig.cs` در پوشه `App_Start` مراجعه کنید و مسیر پیش فرض را حذف کنید. برای مسیریابی سرویس‌ها از قابلیت جدید `Attribute Routing` استفاده خواهیم کرد. فایل مذکور باید مانند لیست زیر باشد.

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        // Web API configuration and services

        // Web API routes
        config.MapHttpAttributeRoutes();
    }
}
```

```

    }
}

```

حال در مسیر ریشه پروژه، پوشه جدیدی با نام *Videos* ایجاد کنید و یک فایل ویدیو نمونه بنام *sample.mp4* در آن کپی کنید. دقت کنید که فرمت فایل ویدیو در مثال جاری *mp4* در نظر گرفته شده اما به سادگی می‌توانید آن را تغییر دهید. سپس در پوشه *Models* کلاس جدیدی بنام *VideoStream* ایجاد کنید. این کلاس مسئول نوشتن داده فایل‌های ویدیویی در *OutputStream* خواهد بود. کد کامل این کلاس را در لیست زیر مشاهده می‌کنید.

```

public class VideoStream
{
    private readonly string _filename;
    private long _contentLength;

    public long FileLength
    {
        get { return _contentLength; }
    }

    public VideoStream(string videoPath)
    {
        _filename = videoPath;
        using (var video = File.Open(_filename, FileMode.Open, FileAccess.Read, FileShare.Read))
        {
            _contentLength = video.Length;
        }
    }

    public async void WriteToStream(Stream outputStream,
        HttpContext content, TransportContext context)
    {
        try
        {
            var buffer = new byte[65536];

            using (var video = File.Open(_filename, FileMode.Open, FileAccess.Read, FileShare.Read))
            {
                var length = (int)video.Length;
                var bytesRead = 1;

                while (length > 0 && bytesRead > 0)
                {
                    bytesRead = video.Read(buffer, 0, Math.Min(length, buffer.Length));
                    await outputStream.WriteAsync(buffer, 0, bytesRead);
                    length -= bytesRead;
                }
            }
        }
        catch (HttpException)
        {
            return;
        }
        finally
        {
            outputStream.Close();
        }
    }
}

```

شرح کلاس *VideoStream*

این کلاس ابتدا دو فیلد خصوصی تعریف می‌کند. یکی *_filename* که فقط-خواندنی است و نام فایل ویدیو درخواستی را نگهداری می‌کند. و دیگری *_contentLength* که سایز فایل ویدیو درخواستی را نگهداری می‌کند.

یک خاصیت عمومی بنام *FileLength* نیز تعریف شده که مقدار خاصیت *_contentLength* را بر می‌گرداند.

متد سازنده این کلاس پارامتری از نوع رشته بنام *videoPath* را می‌پذیرد که مسیر کامل فایل ویدیوی مورد نظر است. در این متد، متغیرهای *_filename* و *_contentLength* مقدار دهی می‌شوند. نکته‌ی قابل توجه در این متد استفاده از پارامتر *FileShare.Read* است که باعث می‌شود فایل مورد نظر هنگام باز شدن قفل نشود و برای پروسه‌های دیگر قابل دسترسی باشد.

در آخر متد *WriteToStream* را داریم که مسئول نوشتن داده فایل‌ها به *OutputStream* است. اول از همه دقت کنید که این متد از کلمه کلیدی *async* استفاده می‌کند بنابراین بصورت *asynchronous* اجرا خواهد شد. در بدنه این متد متغیری بنام *buffer* داریم که یک آرایه بایت با سایز 64KB را تعریف می‌کند. به بیان دیگر اطلاعات فایل‌ها را در پکیج‌های 64 کیلوبایتی برای کلاینت ارسال خواهیم کرد. در ادامه فایل مورد نظر را باز می‌کنیم (مجدداً با استفاده از *FileShare.Read*) و شروع به خواندن اطلاعات آن می‌کنیم. هر 64 کیلوبایت خوانده شده بصورت *async* در جریان خروجی نوشته می‌شود و تا هنگامی که به آخر فایل نرسیده ایم این روند ادامه پیدا می‌کند.

```
while (length > 0 && bytesRead > 0)
{
    bytesRead = video.Read(buffer, 0, Math.Min(length, buffer.Length));
    await outputStream.WriteAsync(buffer, 0, bytesRead);
    length -= bytesRead;
}
```

اگر دقت کنید تمام کد بدنه این متد در یک بلاک *try/catch* قرار گرفته است. در صورتی که با خطایی از نوع *HttpException* مواجه شویم (مثلاً هنگام قطع شدن کاربر) عملیات متوقف می‌شود و در آخر نیز جریان خروجی (*outputStream*) بسته خواهد شد. نکته دیگری که باید بدان اشاره کرد این است که کاربر حتی پس از قطع شدن از سرور می‌تواند ویدیو را تا جایی که دریافت کرده مشاهده کند. مثلاً ممکن است 10 پکیج از اطلاعات را دریافت کرده باشد و هنگام مشاهده پکیج دوم از سرور قطع شود. در این صورت امکان مشاهده ویدیو تا انتهای پکیج دهم وجود خواهد داشت.

حال که کلاس *VideoStream* را در اختیار داریم می‌توانیم پروژه را تکمیل کنیم. در پوشه کنترلرها کلاسی بنام *VideoController* بسازید. کد کامل این کلاس را در لیست زیر مشاهده می‌کنید.

```
public class VideoController : ApiController
{
    [Route("api/video/{ext}/{fileName}")]
    public HttpResponseMessage Get(string ext, string fileName)
    {
        string videoPath = HostingEnvironment.MapPath(string.Format("~/Videos/{0}.{1}", fileName,
ext));
        if (File.Exists(videoPath))
        {
            FileInfo fi = new FileInfo(videoPath);
            var video = new VideoStream(videoPath);

            var response = Request.CreateResponse();

            response.Content = new PushStreamContent((Action<Stream, HttpContext,
TransportContext>)video.WriteToStream,
            new MediaTypeHeaderValue("video/" + ext));

            response.Content.Headers.Add("Content-Disposition", "attachment;filename=" +
fi.Name.Replace(" ", ""));
            response.Content.Headers.Add("Content-Length", video.FileLength.ToString());

            return response;
        }
        else
        {
            return Request.CreateResponse(HttpStatusCode.NotFound);
        }
    }
}
```

شرح کلاس VideoController

همانطور که می‌بینید مسیر دستیابی به این کنترلر با استفاده از قابلیت *Attribute Routing* تعریف شده است.

```
[Route("api/video/{ext}/{fileName}")]
```

نمونه ای از یک درخواست که به این مسیر نگاشت می‌شود:

```
api/video/mp4/sample
```

بنابراین این مسیر فرمت و نام فایل مورد نظر را بدین شکل می‌پذیرد. در نمونه جاری ما فایل *sample.mp4* را درخواست کرده ایم.

متد *Get* این کنترلر دو پارامتر با نام‌های *ext* و *fileName* را می‌پذیرد که همان فرمت و نام فایل هستند. سپس با استفاده از کلاس *HostingEnvironment* سعی می‌کنیم مسیر کامل فایل درخواست شده را بدست آوریم.

```
string videoPath = HostingEnvironment.MapPath(string.Format("~/Videos/{0}.{1}", fileName, ext));
```

استفاده از این کلاس با *Server.MapPath* تفاوتی نمی‌کند. در واقع خود *Server.MapPath* نهایتاً همین کلاس *HostingEnvironment* را فراخوانی می‌کند. اما در کنترلرهای *Web Api* به کلاس *Server* دسترسی نداریم. همانطور که مشاهده می‌کنید فایل مورد نظر در پوشه *Videos* جستجو می‌شود، که در ریشه سایت هم قرار دارد. در ادامه اگر فایل درخواست شده وجود داشت و هله جدیدی از کلاس *VideoStream* می‌سازیم و مسیر کامل فایل را به آن پاس می‌دهیم.

```
var video = new VideoStream(videoPath);
```

سپس آبجکت پاسخ را و هله سازی می‌کنیم و با استفاده از کلاس *PushStreamContent* اطلاعات را به کلاینت می‌فرستیم.

```
var response = Request.CreateResponse();
response.Content = new PushStreamContent((Action<Stream, HttpContext,
TransportContext>)video.WriteToStream, new MediaTypeHeaderValue("video/" + ext));
```

کلاس *PushStreamContent* در فضای نام *System.Net.Http* وجود دارد. همانطور که می‌بینید امضای *Action* پاس داده شده، با امضای متد *WriteToStream* در کلاس *VideoStream* مطابقت دارد.

در آخر دو *Header* به پاسخ ارسالی اضافه می‌کنیم تا نوع داده ارسالی و سایز آن را مشخص کنیم.

```
response.Content.Headers.Add("Content-Disposition", "attachment;filename=" + fileName);
response.Content.Headers.Add("Content-Length", video.FileLength.ToString());
```

افزودن این دو مقدار مهم است. در صورتی که این *Header*ها را تعریف نکنید سایز فایل دریافتی و مدت زمان آن نامعلوم خواهد بود که تجربه کاربری خوبی بدست نمی‌دهد. نهایتاً هم آبجکت پاسخ را به کلاینت ارسال می‌کنیم. در صورتی هم که فایل مورد نظر در پوشه *Videos* پیدا نشود پاسخ *NotFound* را بر می‌گردانیم.

```
if(File.Exists(videoPath))
{
    // removed for brevity
}
else
{
    return Request.CreateResponse(HttpStatusCode.NotFound);
}
```

خوب، برای تست این مکانیزم نیاز به یک کنترلر *MVC* و یک *View* داریم. در پوشه کنترلرها کلاسی بنام *HomeController* ایجاد کنید که با لیست زیر مطابقت داشته باشد.

```
public class HomeController : Controller
{
    // GET: Home
    public ActionResult Index()
    {
```



```

    return View();
}
}

```

نمای این متد را بسازید (با کلیک راست روی متد Index و انتخاب گزینه Add View) و کد آن را مطابق لیست زیر تکمیل کنید.

```

<div>
  <div>
    <video width="480" height="270" controls="controls" preload="auto">
      <source src="/api/video/mp4/sample" type="video/mp4" />
      Your browser does not support the video tag.
    </video>
  </div>
</div>

```

همانطور که مشاهده می‌کنید یک المنت ویدیو تعریف کرده ایم که خواص طول، عرض و غیره آن نیز مقدار دهی شده اند. زیر تگ source متنی درج شده که در صورت لزوم به کاربر نشان داده می‌شود. گرچه اکثر مرورگرهای مدرن از المنت ویدیو پشتیبانی می‌کنند. تگ سورس فایل با مشخصات sample.mp4 را درخواست می‌کند و نوع آن را نیز video/mp4 مشخص کرده ایم.

اگر پروژه را اجرا کنید می‌بینید که ویدیو مورد نظر آماده پخش است. برای اینکه ببینید چطور داده‌های ویدیو در قالب پکیج‌های 64 کیلو بایتی دریافت می‌شوند از ابزار مرورگر تان استفاده کنید. مثلاً در گوگل کروم F12 را بزنید و به قسمت Network بروید. صفحه را یکبار مجدداً بارگذاری کنید تا ارتباطات شبکه مانیتور شود. اگر به المنت sample دقت کنید می‌بینید که با شروع پخش ویدیو پکیج‌های اطلاعات یکی پس از دیگری دریافت می‌شوند و اطلاعات ریز آن را می‌توانید مشاهده کنید.

پروژه نمونه به این مقاله ضمیمه شده است. قابلیت Package Restore فعال شده و برای صرفه جویی در حجم فایل، تمام پکیج‌ها و محتویات پوشه bin حذف شده اند. برای تست بیشتر می‌توانید فایل sample.mp4 را با فایل حجیم‌تر جایگزین کنید تا نحوه دریافت اطلاعات را با روشی که در بالا بدان اشاره شد مشاهده کنید.

[AsyncVideoStreaming.rar](#)

نظرات خوانندگان

نویسنده: علی

تاریخ: ۱۷:۵۵ ۱۳۹۳/۰۶/۱۰

سلام

امروز این مطلب رو دیدم و چند روز پیش خودم انجامش داده بودم. نکته‌ی عجیب اینه که وقتی از این حالت برای پخش ویدئو استفاده می‌کنیم، پلیر میزان فریم‌های بافر شده از ویدیو را نمایش نمیده، در واقع کاربر متوجه نمیشه که تا کجای فیلم از سرور دانلود شده (در صورتی که در حالت پخش مستقیم ویدیو از لینک مستقیم اینگونه نیست).

ممنون میشم اگر به این سه سوال پاسخ بدین :

- 1- مزیت این روش نسبت به روشی که از لینک مستقیم فایل ویدیو استفاد می‌کنیم چیه ؟
- 2- آیا استفاده از این روش باری بر روی پردازنده، رم و... سرور اضافه می‌کنه ؟
- 3- برای پخش ویدیو از این روش استفاده کنیم بهتره یا از لینک مستقیم ؟

با تشکر

چرا JSON.NET؟

[JSON.NET](#) یک کتابخانه‌ی سورس باز کار با اشیاء JSON در دات نت است. تاریخچه‌ی آن به 8 سال قبل بر می‌گردد و توسط یک برنامه نویسی نیوزیلندی به نام James Newton King تهیه شده‌است. اولین نگارش آن در سال 2006 ارائه شد؛ مقارن با زمانی که اولین استاندارد JSON نیز ارائه گردید.

این کتابخانه از آن زمان تا کنون، 6 میلیون بار دانلود شده‌است و به علت کیفیت بالای آن، این روزها پایه اصلی بسیاری از کتابخانه‌ها و فریم ورک‌های دات نت می‌باشد؛ مانند RavenDB تا ASP.NET Web API و SignalR مایکروسافت و همچنین گوگل نیز از آن جهت تدارک کلاینت‌های کار با API خود استفاده می‌کنند.

هرچند دات نت برای نمونه در نگارش سوم آن جهت مصارف WCF کلاسی را به نام [DataContractJsonSerializer](#) ارائه کرد، اما کار کردن با آن محدود است به فرمت خاص WCF به همراه عدم انعطاف پذیری و سادگی کار با آن. به علاوه باید در نظر داشت که JSON.NET از دات نت 2 به بعد تا مونو، Win8 و ویندوز فون را نیز پشتیبانی می‌کند.

برای نصب آن نیز کافی است دستور ذیل را در کنسول پاورشل نیوگت اجرا کنید:

```
PM> install-package Newtonsoft.Json
```

معماری JSON.NET

کتابخانه‌ی JSON.NET از سه قسمت عمده تشکیل شده‌است:

الف) JsonSerializer

ب) LINQ to JSON

ج) JSON Schema

الف) JsonSerializer

کار JsonSerializer تبدیل اشیاء دات نت به JSON و برعکس است. مزیت مهم آن امکانات قابل توجه تنظیم عملکرد و خروجی آن می‌باشد که این تنظیمات را به شکل ویژگی‌های خواص نیز می‌توان اعمال نمود. به علاوه امکان سفارشی سازی هر کدام نیز توسط کلاسی به نام JsonConvert، پیش بینی شده‌است.

یک مثال:

```
var roles = new List<string>
{
    "Admin",
    "User"
};
string json = JsonConvert.SerializeObject(roles, Formatting.Indented);
```

در اینجا نحوه‌ی استفاده از JSON.NET را جهت تبدیل یک شیء دات نت، به معادل JSON آن مشاهده می‌کنید. اعمال تنظیم Formatting.Indented سبب خواهد شد تا خروجی آن دارای Indentation باشد. برای نمونه اگر در برنامه‌ی خود قصد دارید فرمت JSON تو در تویی را به نحو زیبا و خوانایی نمایش دهید یا چاپ کنید، همین تنظیم ساده کافی خواهد بود.

و یا در مثال ذیل استفاده از یک anonymous object را مشاهده می‌کنید:

```
var jsonString = JsonConvert.SerializeObject(new
{
    Id = 1,
    Name = "Test"
}, Formatting.Indented);
```

به صورت پیش فرض تنها خواص عمومی کلاس‌ها توسط JSON.NET تبدیل خواهند شد.

تنظیمات پیشرفته‌تر JSON.NET

مزیت مهم JSON.NET بر سایر کتابخانه‌های موجود مشابه، قابلیت‌های سفارشی سازی قابل توجه آن است. در مثال ذیل نحوه‌ی معرفی JsonSerializerSettings را مشاهده می‌نمائید:

```
var jsonData = JsonConvert.SerializeObject(new
{
    Id = 1,
    Name = "Test",
    DateTime = DateTime.Now
}, new JsonSerializerSettings
{
    Formatting = Formatting.Indented,
    Converters =
    {
        new JavaScriptDateTimeConverter()
    }
});
```

در اینجا با استفاده از تنظیم JavaScriptDateTimeConverter، می‌توان خروجی DateTime استاندارد را به مصرف کنندگان جاوا اسکریپتی سمت کاربر ارائه داد؛ با خروجی ذیل:

```
{
  "Id": 1,
  "Name": "Test",
  "DateTime": new Date(1409821985245)
}
```

نوشتن خروجی JSON در یک استریم

خروجی متد JsonConvert.SerializeObject یک رشته‌است که در صورت نیاز به سادگی توسط متد File.WriteAllText در یک فایل قابل ذخیره می‌باشد. اما برای رسیدن به حداکثر کارایی و سرعت می‌توان از استریم‌ها نیز استفاده کرد:

```
using (var stream = File.CreateText(@"c:\output.json"))
{
    var jsonSerializer = new JsonSerializer
    {
        Formatting = Formatting.Indented
    };
    jsonSerializer.Serialize(stream, new
    {
        Id = 1,
        Name = "Test",
        DateTime = DateTime.Now
    });
}
```

کلاس JsonSerializer و متد Serialize آن یک استریم را نیز جهت نوشتن خروجی می‌پذیرند. برای مثال response.Output برنامه‌های وب نیز یک استریم است و در اینجا نوشتن مستقیم در استریم بسیار سریعتر است از تبدیل شیء به رشته و سپس ارائه خروجی آن؛ زیرا سربار تهیه رشته JSON از آن حذف می‌گردد و نهایتاً GC کار کمتری را باید انجام دهد.

تبدیل رشته‌ای به اشیاء دات نت

اگر رشته‌ی jsonData ای را که پیشتر تولید کردیم، بخواهیم تبدیل به نمونه‌ای از شیء User ذیل کنیم:

```
public class User
{
```

```
public int Id { set; get; }
public string Name { set; get; }
public DateTime DateTime { set; get; }
}
```

خواهیم داشت:

```
var user = JsonConvert.DeserializeObject<User>(jsonData);
```

در اینجا از متد `DeserializeObject` به همراه مشخص سازی صریح نوع شیء نهایی استفاده شده است. البته در اینجا با توجه به استفاده از `JavaScriptDateTimeConverter` برای تولید `jsonData`، نیاز است چنین تنظیمی را نیز در حالت `DeserializeObject` مشخص کنیم:

```
var user = JsonConvert.DeserializeObject<User>(jsonData, new JsonSerializerSettings
{
    Converters = { new JavaScriptDateTimeConverter() }
});
```

مقدار دهی یک نمونه یا وهله‌ی از پیش موجود

متد `JsonConvert.DeserializeObject` یک شیء جدید را ایجاد می‌کند. اگر قصد دارید صرفاً تعدادی از خواص یک وهله‌ی موجود، توسط JSON.NET مقدار دهی شوند از متد `PopulateObject` استفاده کنید:

```
JsonConvert.PopulateObject(jsonData, user);
```

کاهش حجم JSON تولیدی

زمانیکه از متد `JsonConvert.SerializeObject` استفاده می‌کنیم، تمام خواص عمومی تبدیل به معادل JSON آن‌ها خواهند شد؛ حتی خواصی که مقدار ندارند. این خواص در خروجی JSON، با مقدار `null` مشخص می‌شوند. برای حذف این خواص از خروجی JSON نهایی تنها کافی است در تنظیمات `JsonSerializerSettings`، مقدار `NullValueHandling = NullValueHandling.Ignore` مشخص گردد.

```
var jsonData = JsonConvert.SerializeObject(object, new JsonSerializerSettings
{
    NullValueHandling = NullValueHandling.Ignore,
    Formatting = Formatting.Indented
});
```

مورد دیگری که سبب کاهش حجم خروجی نهایی خواهد شد، تنظیم `DefaultValueHandling = DefaultValueHandling.Ignore` است. در این حالت کلیه خواصی که دارای مقدار پیش فرض خودشان هستند، در خروجی JSON ظاهر نخواهند شد. مثلاً مقدار پیش فرض خاصیت `int` مساوی صفر است. در این حالت کلیه خواص از نوع `int` که دارای مقدار صفر می‌باشند، در خروجی قرار نمی‌گیرند.

به علاوه حذف `Formatting = Formatting.Indented` نیز توصیه می‌گردد. در این حالت فشرده‌ترین خروجی ممکن حاصل خواهد شد.

مدیریت ارث بری توسط JSON.NET

در مثال ذیل کلاس کارمند و کلاس مدیر را که خود نیز در اصل یک کارمند می‌باشد، ملاحظه می‌کنید:

```
public class Employee
{
    public string Name { set; get; }
}

public class Manager : Employee
{
    public IList<Employee> Reports { set; get; }
}
```

در اینجا هر مدیر لیست کارمندانی را که به او گزارش می‌دهند نیز به همراه دارد. در ادامه نمونه‌ای از مقدار دهی این اشیاء ذکر شده‌اند:

```
var employee = new Employee { Name = "User1" };
var manager1 = new Manager { Name = "User2" };
var manager2 = new Manager { Name = "User3" };
manager1.Reports = new[] { employee, manager2 };
manager2.Reports = new[] { employee };
```

با فراخوانی

```
var list = JsonConvert.SerializeObject(manager1, Formatting.Indented);
```

یک چنین خروجی JSON ایی حاصل می‌شود:

```
{
  "Reports": [
    {
      "Name": "User1"
    },
    {
      "Reports": [
        {
          "Name": "User1"
        }
      ],
      "Name": "User3"
    }
  ],
  "Name": "User2"
}
```

این خروجی JSON جهت تبدیل به نمونه‌ی معادل دات نتی خود، برای مثال جهت رسیدن به manager1 در کدهای فوق، چندین مشکل را به همراه دارد:

- در اینجا مشخص نیست که این اشیاء، کارمند هستند یا مدیر. برای مثال مشخص نیست User2 چه نوعی دارد و باید به کدام شیء نگاشت شود.

- مشکل دوم در مورد کاربر User1 است که در دو قسمت تکرار شده‌است. این شیء JSON اگر به نمونه‌ی معادل دات نتی خود نگاشت شود، به دو وهله از User1 خواهیم رسید و نه یک وهله‌ی اصلی که سبب تولید این خروجی JSON شده‌است.

برای حل این دو مشکل، تغییرات ذیل را می‌توان به JSON.NET اعمال کرد:

```
var list = JsonConvert.SerializeObject(manager1, new JsonSerializerSettings
{
    Formatting = Formatting.Indented,
    TypeNameHandling = TypeNameHandling.Objects,
    PreserveReferencesHandling = PreserveReferencesHandling.Objects
});
```

با این خروجی:

```
{
  "$id": "1",
  "$type": "JsonNetTests.Manager, JsonNetTests",
  "Reports": [
    {
      "$id": "2",
      "$type": "JsonNetTests.Employee, JsonNetTests",
      "Name": "User1"
    },
    {
      "$id": "3",
      "$type": "JsonNetTests.Manager, JsonNetTests",
      "Reports": [
        {
          "$ref": "2"
        }
      ],
      "Name": "User3"
    }
  ],
  "Name": "User2"
}
```

- با تنظیم `TypeNameHandling = TypeNameHandling.Objects` سبب خواهیم شد تا خاصیت اضافه‌ای به نام `type$` به خروجی JSON اضافه شود. این نوع، در حین فراخوانی متد `JsonConvert.DeserializeObject` جهت تشخیص صحیح نگاشت اشیاء بکار گرفته خواهد شد و اینبار مشخص است که کدام شیء، کارمند است و کدامیک مدیر.

- با تنظیم `PreserveReferencesHandling = PreserveReferencesHandling.Objects` شماره Id خودکاری نیز به خروجی JSON اضافه می‌گردد. اینبار اگر به گزارش دهنده‌ها با دقت نگاه کنیم، مقدار `ref=2$` را خواهیم دید. این مورد سبب می‌شود تا در حین نگاشت نهایی، دو وهله متفاوت از شیء با `Id=2` تولید نشود.

باید دقت داشت که در حین استفاده از `JsonConvert.DeserializeObject` نیز باید `JsonSerializerSettings` یاد شده، تنظیم شوند.

ویژگی‌های قابل تنظیم در JSON.NET

علاوه بر `JsonSerializerSettings` که از آن صحبت شد، در JSON.NET امکان تنظیم یک سری از ویژگی‌ها به ازای خواص مختلف نیز وجود دارند.

- برای نمونه ویژگی `JsonIgnore` معروفترین آن‌ها است:

```
public class User
{
    public int Id { set; get; }

    [JsonIgnore]
    public string Name { set; get; }

    public DateTime DateTime { set; get; }
}
```

`JsonIgnore` سبب می‌شود تا خاصیتی در خروجی نهایی JSON تولیدی حضور نداشته باشد و از آن صرف‌نظر شود.

- با استفاده از ویژگی `JsonProperty` اغلب مواردی را که پیشتر بحث کردیم مانند `TypeNameHandling`، `NullValueHandling` و غیره، می‌توان تنظیم نمود. همچنین گاهی از اوقات کتابخانه‌های جاوا اسکریپتی سمت کاربر، از اسامی خاصی که از روش‌های نامگذاری دات نت پیروی نمی‌کنند، در طراحی خود استفاده می‌کنند. در اینجا می‌توان نام خاصیت نهایی را که قرار است رندر شود نیز صریحاً مشخص کرد. برای مثال:

```
[JsonProperty(PropertyName = "m_name", NullValueHandling = NullValueHandling.Ignore)]
public string Name { set; get; }
```

همچنین در اینجا امکان تنظیم Order نیز وجود دارد. برای مثال مشخص کنیم که خاصیت X در ابتدا قرار گیرد و پس از آن خاصیت Y رندر شود.

- استفاده از ویژگی JsonObject به همراه مقدار OptIn آن به این معنا است که از کلیه خواصی که دارای ویژگی JsonProperty نیستند، صرفنظر شود. حالت پیش فرض آن OptOut است؛ یعنی تمام خواص عمومی در خروجی JSON حضور خواهند داشت منهای مواردی که با JsonIgnore مزین شوند.

```
[JsonObject(MemberSerialization.OptIn)]
public class User
{
    public int Id { set; get; }

    [JsonProperty]
    public string Name { set; get; }

    public DateTime DateTime { set; get; }
}
```

- با استفاده از ویژگی JsonConverter می‌توان نحوه‌ی رندر شدن مقدار خاصیت را سفارشی سازی کرد. برای مثال:

```
[JsonConverter(typeof(JavaScriptDateTimeConverter))]
public DateTime DateTime { set; get; }
```

تهیه یک JsonConverter سفارشی

با استفاده از JsonConverter می‌توان کنترل کاملی را بر روی اعمال serialization و deserialization مقادیر خواص اعمال کرد. مثال زیر را در نظر بگیرید:

```
public class HtmlColor
{
    public int Red { set; get; }
    public int Green { set; get; }
    public int Blue { set; get; }
}

var colorJson = JsonConvert.SerializeObject(new HtmlColor
{
    Red = 255,
    Green = 0,
    Blue = 0
}, Formatting.Indented);
```

در اینجا علاقمندیم، در حین عملیات serialization، بجای اینکه مقادیر اجزای رنگ تهیه شده به صورت int نمایش داده شوند، کل رنگ با فرمت hex رندر شوند. برای اینکار نیاز است یک JsonConverter سفارشی را تدارک دید:

```
public class HtmlColorConverter : JsonConverter
{
    public override bool CanConvert(Type objectType)
    {
        return objectType == typeof(HtmlColor);
    }

    public override object ReadJson(JsonReader reader, Type objectType,
        object existingValue, JsonSerializer serializer)
    {
        throw new NotSupportedException();
    }

    public override void WriteJson(JsonWriter writer, object value, JsonSerializer serializer)
    {
        var color = value as HtmlColor;
        if (color == null)
        {
            // ...
        }
    }
}
```



```

        return;

        writer.WriteValue("#" + color.Red.ToString("X2")
            + color.Green.ToString("X2") + color.Blue.ToString("X2"));
    }
}

```

کار با ارث بری از کلاس پایه `JsonConverter` شروع می‌شود. سپس باید تعدادی از متدهای این کلاس پایه را بازنویسی کرد. در متد `CanConvert` اعلام می‌کنیم که تنها اشیایی از نوع کلاس `HtmlColor` را قرار است پردازش کنیم. سپس در متد `WriteJson` منطق سفارشی خود را می‌توان پیاده سازی کرد. از آنجائیکه این تبدیلگر صرفاً قرار است برای حالت `serialization` استفاده شود، قسمت `ReadJson` آن پیاده سازی نشده‌است.

در آخر برای استفاده از آن خواهیم داشت:

```

var colorJson = JsonConvert.SerializeObject(new HtmlColor
{
    Red = 255,
    Green = 0,
    Blue = 0
}, new JsonSerializerSettings
{
    Formatting = Formatting.Indented,
    Converters = { new HtmlColorConverter() }
});

```

نظرات خوانندگان

نویسنده:

افتابی

تاریخ:

۲۱:۵۴ ۱۳۹۳/۰۶/۱۳

سلام؛ من مطالب مربوطه رو خوندم فقط اینکه توی یه صفحه rozar در mvc من به چه نحو میتونم از آن استفاده کنم ، حتی توی سایت خودش هم رفتم و sample ها رو دیدم فقط میخوام در یک پروژه به چه نحو ازش استفاده کنم و کجا کارش ببرم؟

نویسنده:

وحید نصیری

تاریخ:

۲۱:۵۹ ۱۳۹۳/۰۶/۱۳

در یک اکشن متد، بجای return Json پیش فرض و توکار، می‌شود نوشت:

```
return Content(JsonConvert.SerializeObject(obj));
```

البته این ساده‌ترین روش استفاده از آن است؛ برای مقاصد Ajax ایی. و یا برای ذکر Content type می‌توان به صورت زیر عمل کرد:

```
return new ContentResult
{
    Content = JsonConvert.SerializeObject(obj),
    ContentType = "application/json"
};
```

نویسنده:

رحمت اله رضایی

تاریخ:

۱۴:۳ ۱۳۹۳/۰۶/۱۴

"ASP.NET Web API و SignalR از این کتابخانه استفاده می‌کنند". دلیلی دارد هنوز ASP.NET MVC از این کتابخانه استفاده نکرده است؟

نویسنده:

وحید نصیری

تاریخ:

۱۴:۱۹ ۱۳۹۳/۰۶/۱۴

- تا ASP.NET MVC 5 از JavaScriptSerializer در [JsonResult](#) استفاده می‌شود.
- در نگارش بعدی ASP.NET MVC که با Web API یکی شده (یعنی در یک کنترلر هم می‌توانید ActionResult داشته باشید و هم خروجی‌های متداول Web API را با هم) اینبار تامین کننده‌ی [JsonResult](#) از طریق تزریق وابستگی‌ها تامین می‌شود و می‌تواند هر کتابخانه‌ای که صلاح می‌دانید باشد. البته [یک مقدار پیش فرض](#) هم دارد که دقیقاً از JSON.NET استفاده می‌کند.

نویسنده:

وحید نصیری

تاریخ:

۱۲:۳۵ ۱۳۹۳/۰۷/۱۸

یک نکته‌ی تکمیلی

استفاده از استریم‌ها برای کار با فایل‌ها در JSON.NET

```
public static T DeserializeFromFile<T>(string filePath, JsonSerializerSettings settings = null)
{
    if (!File.Exists(filePath))
        return default(T);

    using (var fileStream = File.OpenRead(filePath))
    {
        using (var streamReader = new StreamReader(fileStream))
        {
            using (var reader = new JsonTextReader(streamReader))
            {
                var serializer = settings == null ? JsonSerializer.Create() :
                JsonSerializer.Create(settings);
            }
        }
    }
}
```

```

        return serializer.Deserialize<T>(reader);
    }
}

public static void SerializeToFile(string filePath, object data, JsonSerializerSettings
settings = null)
{
    using (var fileStream = new FileStream(filePath, FileMode.Create))
    {
        using (var streamReader = new StreamWriter(fileStream))
        {
            using (var reader = new JsonTextWriter(streamReader))
            {
                var serializer = settings == null ? JsonSerializer.Create() :
JsonSerializer.Create(settings);
                serializer.Serialize(reader, data);
            }
        }
    }
}

```

[پس از بررسی مقدماتی](#) امکانات کتابخانه‌ی JSON.NET، در ادامه به تعدادی از تنظیمات کاربردی آن با ذکر مثال‌هایی خواهیم پرداخت.

گرفتن خروجی از CamelCase در JSON.NET

یک سری از کتابخانه‌های جاوا اسکریپتی سمت کلاینت، به نام‌های خواص [CamelCase](#) نیاز دارند و حالت پیش فرض اصول نامگذاری خواص در دات نت عکس آن است. برای مثال بجای UserName به userName نیاز دارند تا بتوانند صحیح کار کنند. روش اول حل این مشکل، استفاده از ویژگی JsonProperty بر روی تک تک خواص و مشخص کردن نام‌های مورد نیاز کتابخانه‌ی جاوا اسکریپتی به صورت صریح است. روش دوم، استفاده از تنظیمات ContractResolver می‌باشد که با تنظیم آن به CamelCasePropertyNameContractResolver به صورت خودکار به تمامی خواص به صورت یکسانی اعمال می‌گردد:

```
var json = JsonConvert.SerializeObject(obj, new JsonSerializerSettings
{
    ContractResolver = new CamelCasePropertyNamesContractResolver()
});
```

درج نام‌های المان‌های یک Enum در خروجی JSON

اگر یکی از عناصر در حال تبدیل به JSON، از نوع enum باشد، به صورت پیش فرض مقدار عددی آن در JSON نهایی درج می‌گردد:

```
using Newtonsoft.Json;

namespace JsonNetTests
{
    public enum Color
    {
        Red,
        Green,
        Blue,
        White
    }

    public class Item
    {
        public string Name { set; get; }
        public Color Color { set; get; }
    }

    public class EnumTests
    {
        public string GetJson()
        {
            var item = new Item
            {
                Name = "Item 1",
                Color = Color.Blue
            };

            return JsonConvert.SerializeObject(item, Formatting.Indented);
        }
    }
}
```

با این خروجی:

```
{
  "Name": "Item 1",
  "Color": 2
}
```

اگر علاقمند هستید که بجای عدد 2، دقیقاً مقدار Blue در خروجی JSON درج گردد، می‌توان به یکی از دو روش ذیل عمل کرد:
الف) مزین کردن خاصیت از نوع enum به ویژگی JsonSerializer.Converters از نوع StringEnumConverter:

```
[JsonConverter(typeof(StringEnumConverter))]
public Color Color { get; set; }
```

ب) و یا اگر می‌خواهید این تنظیم به تمام خواص از نوع enum به صورت یکسانی اعمال شود، می‌توان نوشت:

```
return JsonConvert.SerializeObject(item, new JsonSerializerSettings
{
    Formatting = Formatting.Indented,
    Converters = { new StringEnumConverter() }
});
```

تهیه خروجی JSON از مدل‌های مرتبط، بدون Stack overflow

دو کلاس گروه‌های محصولات و محصولات ذیل را در نظر بگیرید:

```
public class Category
{
    public int Id { get; set; }
    public string Name { get; set; }

    public virtual ICollection<Product> Products { get; set; }

    public Category()
    {
        Products = new List<Product>();
    }
}

public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }

    public virtual Category Category { get; set; }
}
```

این نوع طراحی در Entity framework بسیار مرسوم است. در اینجا طرف‌های دیگر یک رابطه، توسط خاصیتی virtual معرفی می‌شوند که به آن‌ها خواص راهبری یا navigation properties هم می‌گویند.
با توجه به این دو کلاس، سعی کنید مثال ذیل را اجرا کرده و از آن، خروجی JSON تهیه کنید:

```
using System.Collections.Generic;
using Newtonsoft.Json;
using Newtonsoft.Json.Converters;

namespace JsonNetTests
{
    public class SelfReferencingLoops
    {
        public string GetJson()
        {
            var category = new Category
            {
                Id = 1,
                Name = "Category 1"
            };
            var product = new Product
```

```

        {
            Id = 1,
            Name = "Product 1"
        };

        category.Products.Add(product);
        product.Category = category;

        return JsonConvert.SerializeObject(category, new JsonSerializerSettings
        {
            Formatting = Formatting.Indented,
            Converters = { new StringEnumConverter() }
        });
    }
}

```

برنامه با این استثناء متوقف می‌شود:

```

An unhandled exception of type 'Newtonsoft.Json.JsonSerializationException' occurred in
Newtonsoft.Json.dll
Additional information: Self referencing loop detected for property 'Category' with type
'JsonNetTests.Category'. Path 'Products[0]'.

```

اصل خطای معروف فوق «Self referencing loop detected» است. در اینجا کلاس‌هایی که به یکدیگر ارجاع می‌دهند، در حین عملیات Serialization سبب بروز یک حلقه‌ی بازگشتی بی‌نهایت شده و در آخر، برنامه با خطای stack overflow خاتمه می‌یابد.

راه حل اول:

به تنظیمات JSON.NET، مقدار `ReferenceLoopHandling = ReferenceLoopHandling.Ignore` را اضافه کنید تا از حلقه‌ی بازگشتی بی‌پایان جلوگیری شود:

```

return JsonConvert.SerializeObject(category, new JsonSerializerSettings
{
    Formatting = Formatting.Indented,
    ReferenceLoopHandling = ReferenceLoopHandling.Ignore,
    Converters = { new StringEnumConverter() }
});

```

راه حل دوم:

به تنظیمات JSON.NET، مقدار `PreserveReferencesHandling = PreserveReferencesHandling.Objects` را اضافه کنید تا مدیریت ارجاعات اشیاء توسط خود JSON.NET انجام شود:

```

return JsonConvert.SerializeObject(category, new JsonSerializerSettings
{
    Formatting = Formatting.Indented,
    PreserveReferencesHandling = PreserveReferencesHandling.Objects,
    Converters = { new StringEnumConverter() }
});

```

خروجی حالت دوم به این شکل است:

```

{
  "$id": "1",
  "Id": 1,
  "Name": "Category 1",
  "Products": [
    {
      "$id": "2",
      "Id": 1,
      "Name": "Product 1",
      "Category": {
        "$ref": "1"
      }
    }
  ]
}

```

```
]
}
```

همانطور که ملاحظه می‌کنید، دو خاصیت `id$` و `ref$` توسط JSON.NET به خروجی JSON اضافه شده‌است تا توسط آن بتواند ارجاعات و نمونه‌های اشیاء را تشخیص دهد.

نظرات خوانندگان

نویسنده: وحید نصیری
تاریخ: ۱۵:۱۷ ۱۳۹۳/۰۶/۲۳

گرفتن خروجی مرتب شده بر اساس نام خواص (جهت مقاصد نمایشی):

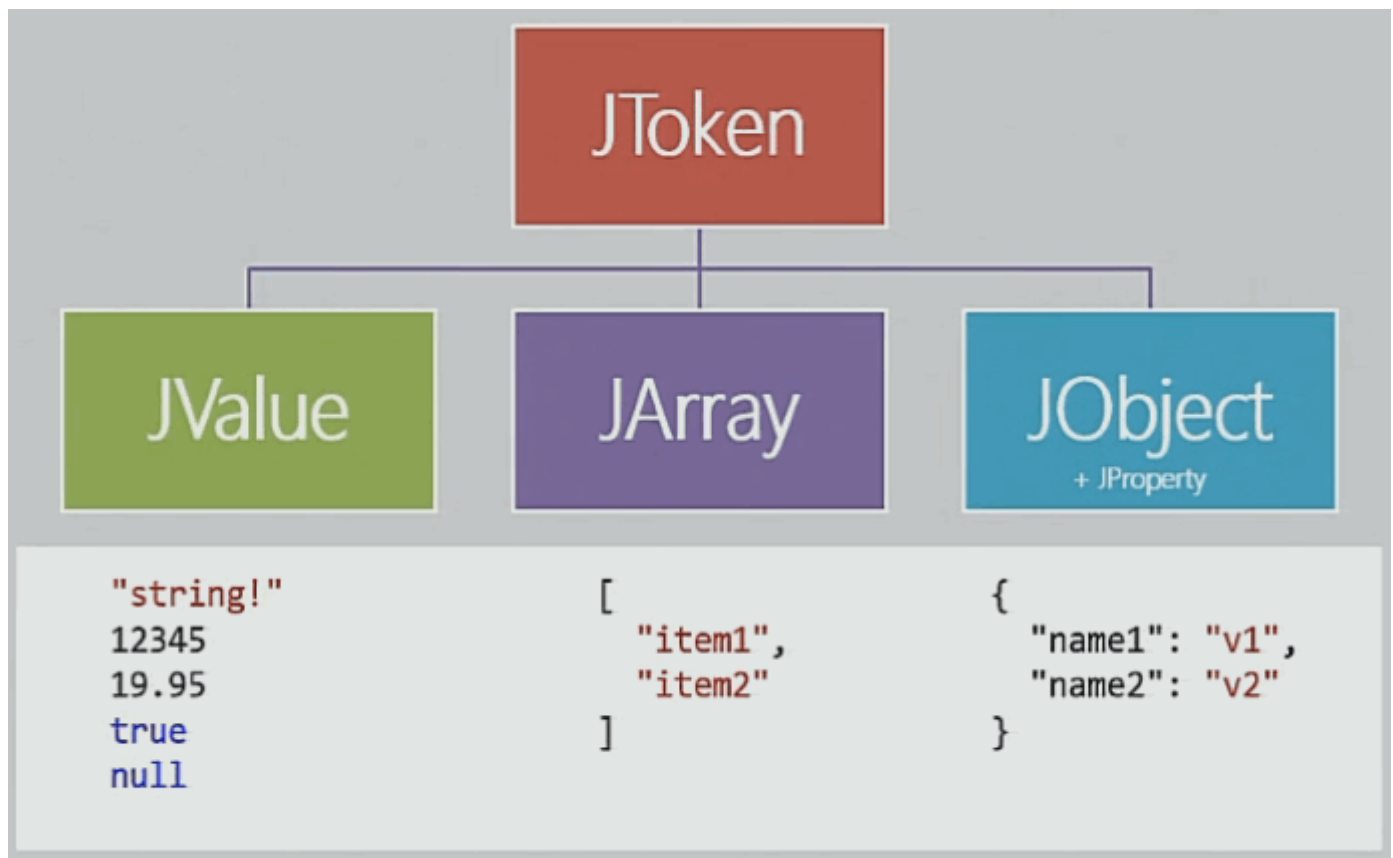
تعریف DefaultContractResolver

```
public class OrderedContractResolver : DefaultContractResolver
{
    protected override IList<JsonProperty> CreateProperties(
        System.Type type, MemberSerialization memberSerialization)
    {
        return base.CreateProperties(type, memberSerialization).OrderBy(p =>
            p.PropertyName).ToList();
    }
}
```

و بعد معرفی آن به نحو ذیل:

```
return JsonConvert.SerializeObject(data, new JsonSerializerSettings
{
    ContractResolver = new OrderedContractResolver()
});
```


عموما از امکانات LINQ to JSON کتابخانه‌ی JSON.NET زمانی استفاده می‌شود که ورودی JSON تو در توی حجیمی را دریافت کرده‌اید اما قصد ندارید به ازای تمام موجودیت‌های آن یک کلاس معادل را جهت نگاشت به آن‌ها تهیه کنید و صرفا یک یا چند مقدار تو در توی آن جهت عملیات استخراج نهایی مدنظر است. به علاوه در اینجا LINQ to JSON واژه‌ی کلیدی dynamic را نیز پشتیبانی می‌کند.



همانطور که در تصویر مشخص است، خروجی‌های JSON عموماً ترکیبی هستند از مقادیر، آرایه‌ها و اشیاء. هر کدام از این‌ها در LINQ to JSON به اشیاء JValue، JArray و JObject نگاشت می‌شوند. البته در حالت JObject هر عضو به یک JProperty و JValue تجزیه خواهد شد.

برای مثال آرایه [1,2] تشکیل شده‌است از یک JArray به همراه دو JValue که مقادیر آن‌را تشکیل می‌دهند. اگر مستقیماً بخواهیم یک JArray را تشکیل دهیم می‌توان از شیء JArray استفاده کرد:

```
var array = new JArray(1, 2, 3);
var arrayToJson = array.ToString();
```

و اگر یک JSON رشته‌ای دریافتی را داریم می‌توان از متد Parse مربوط به JArray کمک گرفت:

```
var json = "[1,2,3]";
var jArray = JArray.Parse(json);
var val = (int)jArray[0];
```

خروجی JArray یک لیست از JToken ها است و با آن می‌توان مانند لیست‌های معمولی کار کرد.

در حالت کار با اشیاء، شیء JObject امکان تهیه اشیاء JSON ایی را دارا است که می‌تواند مجموعه‌ای از JProperty ها باشد:

```
var jobject = new JObject(
    new JProperty("prop1", "value1"),
    new JProperty("prop2", "value2")
);
var jobjectToJson = jobject.ToString();
```

با JObject به صورت dynamic نیز می‌توان کار کرد:

```
dynamic jobj = new JObject();
jobj.Prop1 = "value1";
jobj.Prop2 = "value2";
jobj.Roles = new[] { "Admin", "User"};
```

این روش بسیار شبیه است به حالتی که با اشیاء جاوا اسکریپتی در سمت کلاینت می‌توان کار کرد. و حالت عکس آن توسط متد JObject.Parse قابل انجام است:

```
var json = "{ 'prop1': 'value1', 'prop2': 'value2' }";
var jobj = JObject.Parse(json);
var val1 = (string)jobj["prop1"];
```

اکنون که با اجزای تشکیل دهنده‌ی LINQ to JSON آشنا شدیم، مثال ذیل را در نظر بگیرید:

```
var array = @"[
{
  'prop1': 'value1',
  'prop2': 'value2'
},
{
  'prop1': 'test1',
  'prop2': 'test2'
}
]";
var objects = JArray.Parse(array);
var obj1 = objects.FirstOrDefault(token => (string) token["prop1"] == "value1");
```

خروجی JArray یا JObject از نوع IEnumerable است و بر روی آن‌ها می‌توان کلیه متدهای LINQ را فراخوانی کرد. برای مثال در اینجا اولین شیء‌ایی که مقدار خاصیت prop1 آن مساوی value1 است، یافت می‌شود و یا می‌توان اشیاء را بر اساس مقدار خاصیتی مرتب کرده و سپس آن‌ها را بازگشت داد:

```
var values = objects.OrderBy(token => (string) token["prop1"])
    .Select(token => new { Value = (string) token["prop2"] })
    .ToList();
```

امکان انجام sub queries نیز در اینجا پیش بینی شده‌است:

```
var array = @"[
{
  'prop1': 'value1',
  'prop2': [1,2]
},
{
  'prop1': 'test1',
  'prop2': [1,2,3]
}
]";
var objects = JArray.Parse(array);
var objectContaining3 = objects.Where(token => token["prop2"].Any(v => (int)v == 3)).ToList();
```

در این مثال، خواص prop2 از نوع آرایه‌ای از اعداد صحیح هستند. با کوئری نوشته شده، اشیایی که خاصیت prop2 آن‌ها دارای عضو 3 است، یافت می‌شوند.

ASP.NET Web API در سمت سرور، برای مدیریت ApiController ها و در سمت کلاینت‌های دات نتی آن، برای مدیریت HttpClient، به صورت پیش فرض از JSON.NET استفاده می‌کند. در ادامه نگاهی خواهیم داشت به تنظیمات JSON در سرور و کلاینت‌های ASP.NET Web API.

آماده سازی یک مثال Self host

برای اینکه خروجی‌های JSON را بهتر و بدون نیاز به ابزار خاصی مشاهده کنیم، می‌توان یک پروژه‌ی کنسول جدید را آغاز کرده و سپس آن را تبدیل به Host مخصوص Web API کرد. برای اینکار تنها کافی است در کنسول پاور شل نیوگت دستور ذیل را صادر کنید:

```
PM> Install-Package Microsoft.AspNet.WebApi.OwinSelfHost
```

سپس کنترلر Web API ما از کدهای ذیل تشکیل خواهد شد که در آن در متد Post، قصد داریم اصل محتوای دریافتی از کاربر را نمایش دهیم. توسط متد GetAll آن، خروجی نهایی JSON آن در سمت کاربر بررسی خواهد شد.

```
using System;
using System.Collections.Generic;
using System.Net;
using System.Net.Http;
using System.Threading.Tasks;
using System.Web.Http;

namespace WebApiSelfHostTests
{
    public class UsersController : ApiController
    {
        public IEnumerable<User> GetAllUsers()
        {
            return new[]
            {
                new User{ Id = 1, Name = "User 1", Type = UserType.Admin },
                new User{ Id = 2, Name = "User 2", Type = UserType.User }
            };
        }

        public async Task<HttpResponseMessage> Post(HttpRequestMessage request)
        {
            var jsonContent = await request.Content.ReadAsStringAsync();
            Console.WriteLine("JsonContent (Server Side): {0}", jsonContent);
            return new HttpResponseMessage(HttpStatusCode.Created);
        }
    }
}
```

که در آن شیء کاربر چنین ساختاری را دارد:

```
namespace WebApiSelfHostTests
{
    public enum UserType
    {
        User,
        Admin,
        Writer
    }

    public class User
    {
        public int Id { set; get; }
        public string Name { set; get; }
        public UserType Type { set; get; }
    }
}
```

```
}
}
```

برای اعمال تنظیمات self host ابتدا نیاز است یک کلاس Startup مخصوص Owin را تهیه کرد:

```
using System.Web.Http;
using Newtonsoft.Json;
using Newtonsoft.Json.Converters;
using Owin;

namespace WebApiSelfHostTests
{
    /// <summary>
    /// PM> Install-Package Microsoft.AspNet.WebApi.OwinSelfHost
    /// </summary>
    public class Startup
    {
        public void Configuration(IAppBuilder appBuilder)
        {
            var config = new HttpConfiguration();
            config.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );

            appBuilder.UseWebApi(config);
        }
    }
}
```

که سپس با فراخوانی چند سطر ذیل، سبب راه اندازی سرور Web API، بدون نیاز به IIS خواهد شد:

```
var server = WebApp.Start<Startup>(url: BaseAddress);

Console.WriteLine("Press Enter to quit.");
Console.ReadLine();
server.Dispose();
```

در ادامه اگر در سمت کلاینت، دستورات ذیل را برای دریافت لیست کاربران صادر کنیم:

```
using (var client = new HttpClient())
{
    var response = client.GetAsync(BaseAddress + "api/users").Result;
    Console.WriteLine("Response: {0}", response);
    Console.WriteLine("JsonContent (Client Side): {0}", response.Content.ReadAsStringAsync().Result);
}
```

به این خروجی خواهیم رسید:

```
JsonContent (Client Side): [{"Id":1,"Name":"User 1","Type":1},{ "Id":2,"Name":"User 2","Type":0}]
```

همانطور که ملاحظه می‌کنید، مقدار Type مساوی صفر است. در اینجا چون Type را به صورت enum تعریف کرده‌ایم، به صورت پیش فرض مقدار عددی عضو انتخابی در JSON نهایی درج می‌گردد.

تنظیمات JSON سمت سرور Web API

برای تغییر این خروجی، در سمت سرور تنها کافی است به کلاس Startup مراجعه و HttpConfiguration را به صورت ذیل تنظیم کنیم:

```
public class Startup
```

```
{
    public void Configuration(IApplicationBuilder appBuilder)
    {
        var config = new HttpConfiguration();
        config.Formatters.JsonFormatter.SerializerSettings = new JsonSerializerSettings
        {
            Converters = { new StringEnumConverter() }
        };
    }
}
```

در اینجا با انتخاب `StringEnumConverter`، سبب خواهیم شد تا کلیه مقادیر `enum`، دقیقاً مساوی همان مقدار اصلی رشته‌ای آن‌ها در JSON نهایی درج شوند. اینبار اگر برنامه را اجرا کنیم، چنین خروجی حاصل می‌گردد و در آن دیگر `Type` مساوی صفر نیست:

```
JsonContent (Client Side): [{"Id":1,"Name":"User 1","Type":"Admin"}, {"Id":2,"Name":"User 2","Type":"User"}]
```

تنظیمات JSON سمت کلاینت Web API

اکنون در سمت کلاینت قصد داریم اطلاعات یک کاربر را با فرمت JSON به سمت سرور ارسال کنیم. روش متداول آن توسط کتابخانه‌ی `HttpClient`، استفاده از متد `PostAsJsonAsync` است:

```
var user = new User
{
    Id = 1,
    Name = "User 1",
    Type = UserType.Writer
};

var client = new HttpClient();
client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));

var response = client.PostAsJsonAsync(BaseAddress + "api/users", user).Result;
Console.WriteLine("Response: {0}", response);
```

با این خروجی سمت سرور

```
JsonContent (Server Side): {"Id":1,"Name":"User 1","Type":2}
```

در اینجا نیز `Type` به صورت عددی ارسال شده‌است. برای تغییر آن نیاز است به متدی با سطح پایین‌تر از `PostAsJsonAsync` مراجعه کنیم تا در آن بتوان `JsonMediaTypeFormatter` را مقدار دهی کرد:

```
var jsonMediaTypeFormatter = new JsonMediaTypeFormatter
{
    SerializerSettings = new JsonSerializerSettings
    {
        Converters = { new StringEnumConverter() }
    }
};
var response = client.PostAsync(BaseAddress + "api/users", user,
jsonMediaTypeFormatter).Result;
Console.WriteLine("Response: {0}", response);
```

خاصیت `SerializerSettings` کلاس `JsonMediaTypeFormatter` برای اعمال تنظیمات JSON.NET پیش‌بینی شده‌است. اینبار مقدار دریافتی در سمت سرور به صورت ذیل است و در آن، `Type` دیگر عددی نیست:

```
JsonContent (Server Side): {"Id":1,"Name":"User 1","Type":"Writer"}
```

مثال کامل این بحث را از اینجا می‌توانید دریافت کنید:

نظرات خوانندگان

نویسنده:

وحید نصیری

تاریخ:

۱۰:۴۵ ۱۳۹۳/۰۶/۲۴

یک نکته‌ی تکمیلی

اگر نمی‌خواهید یک وابستگی جدید را (Microsoft.AspNet.WebApi.Client) به پروژه اضافه کنید، کدهای ذیل همان کار HttpClient را برای ارسال اطلاعات، انجام می‌دهند. کلاس WebRequest آن در فضای نام System.Net موجود است:

```
using System;
using System.IO;
using System.Net;
using Newtonsoft.Json;

namespace WebToolkit
{
    public class SimpleHttp
    {
        public HttpStatusCode PostAsJson(string url, object data, JsonSerializerSettings settings)
        {
            if (string.IsNullOrEmpty(url))
                throw new ArgumentNullException("url");

            return PostAsJson(new Uri(url), data, settings);
        }

        public HttpStatusCode PostAsJson(Uri url, object data, JsonSerializerSettings settings)
        {
            if (url == null)
                throw new ArgumentNullException("url");

            var postRequest = (HttpWebRequest)WebRequest.Create(url);
            postRequest.Method = "POST";
            postRequest.UserAgent = "SimpleHttp/1.0";
            postRequest.ContentType = "application/json; charset=utf-8";

            using (var stream = new StreamWriter(postRequest.GetRequestStream()))
            {
                var serializer = JsonSerializer.Create(settings);
                using (var writer = new JsonTextWriter(stream))
                {
                    serializer.Serialize(writer, data);
                    writer.Flush();
                }
            }

            using (var response = (HttpWebResponse)postRequest.GetResponse())
            {
                return response.StatusCode;
            }
        }
    }
}
```

نویسنده:

رشیدیان

تاریخ:

۱۸:۱ ۱۳۹۳/۰۶/۲۶

سلام و وقت بخیر

من وقتی می‌خوام اطلاعات یک فایل جیسون رو به آبجکت تبدیل کنم، با این خطا مواجه میشم:
Additional text encountered after finished reading JSON content: „ Path ", line 1, position 6982

بعد از جستجو متوجه شدم که خطا به دلیل وجود کرکترهای کنترلی هست، پس فایل مذکور رو با روشهای زیر (هر کدام رو جداگانه تست کردم) تمیز کردم:

```
public string RemoveControlCharacters1(string input)
{
    string output = Regex.Replace(input, @"[\u0000-\u001F]", string.Empty);
    return output;
}
```



```

    }

    public string RemoveControlCharacters2(string input)
    {
        string output = new string(input.Where(c => !char.IsControl(c)).ToArray());
        return output;
    }

    public string RemoveControlCharacters3(string inString)
    {
        if (inString == null) return null;
        StringBuilder newString = new StringBuilder();
        char ch;
        for (int i = 0; i < inString.Length; i++)
        {
            ch = inString[i];
            if (!char.IsControl(ch))
            {
                newString.Append(ch);
            }
        }
        return newString.ToString();
    }
}

```

اما کماکان همان خطا را در زمان اجر میبینم.

آیا مشکل چیز دیگری است؟

پرسش: چطور میشود به جیسون دات نت گفت که اصلا کرکتهای کنترلی و یا چیزهایی را که ممکن است خطا ایجاد کنند، ندید بگیرد؟

نویسنده: وحید نصیری
تاریخ: ۱۸:۱۲ ۱۳۹۳/۰۶/۲۶

با تنظیم eventArgs.ErrorContext.Handled = true از خطاهای موجود صرفنظر می‌شود:

```

new JsonSerializerSettings
{
    Error = (sender, eventArgs) =>
    {
        Debug.WriteLine(eventArgs.ErrorContext.Error.Message);
        //if an error happens we can mark it as handled, and it will continue
        eventArgs.ErrorContext.Handled = true;
    }
}

```

نویسنده: رشیدیان
تاریخ: ۱۸:۲۳ ۱۳۹۳/۰۶/۲۶

سپاسگزارم از پاسخ سریع شما.

بخشید کد من به این شکل هست و نمیدونم کجا باید تغییرات رو اعمال کنم:

```
var items = JsonConvert.DeserializeObject<List<Classified>>(json);
```

نویسنده: وحید نصیری
تاریخ: ۱۸:۲۶ ۱۳۹۳/۰۶/۲۶

در پارامتر دوم متد « [تبدیل JSON رشته‌ای به اشیاء دات نت](#) ».

نویسنده: رضایی
تاریخ: ۱۸:۴۶ ۱۳۹۳/۰۶/۲۶

سلام؛ من از کد زیر استفاده کردم

```
myUserApi.Id = UserId;
```

```
return new JsonResult
{
    Data = myUserApi,
    ContentType = "application/json"
};
```

اما این خروجی تولید میشه

```
{
  "$id": "1",
  "Settings": {
    "$id": "2",
    "ReferenceLoopHandling": 0,
    "MissingMemberHandling": 0,
    "ObjectCreationHandling": 0,
    "NullValueHandling": 0,
    "DefaultValueHandling": 0,
    "Converters": [],
    "PreserveReferencesHandling": 0,
    "TypeNameHandling": 0,
    "MetadataPropertyHandling": 0,
    "TypeNameAssemblyFormat": 0,
    "ConstructorHandling": 0,
    "ContractResolver": null,
    "ReferenceResolver": null,
    "TraceWriter": null,
    "Binder": null,
    "Error": null,
    "Context": {
      "$id": "3",
      "m_additionalContext": null,
      "m_state": 0
    },
    "DateFormatString": "yyyy'-'MM'-'dd'T'HH':'mm':'ss.FFFFFFFF",
    "MaxDepth": null,
    "Formatting": 0,
    "DateFormatHandling": 0,
    "DateTimeZoneHandling": 3,
    "DateParseHandling": 1,
    "FloatFormatHandling": 0,
    "FloatParseHandling": 0,
    "StringEscapeHandling": 0,
    "Culture": "(Default)",
    "CheckAdditionalContent": false
  },
  "ContentEncoding": null,
  "ContentType": "application/json",
  "Data": "{\r\n  \"Id\":2\r\n}",
  "JsonRequestBehavior": 1,
  "MaxJsonLength": null,
  "RecursionLimit": null
}
```

نویسنده: وحید نصیری
تاریخ: ۱۸:۵۵ ۱۳۹۳/۰۶/۲۶

مرتبط است به نکته‌ی « [تهیه خروجی JSON از مدل‌های مرتبط، بدون Stack overflow](#) »

در بسیاری از سناریوها این موضوع مطرح می شود که سرویس های طراحی شده بر اساس Asp.Net Web Api، فقط به یک سری آی پی های مشخص سرویس دهند. برای مثال اگر Ip کلاینت در لیست کلاینت های دارای لایسنس خریداری شده بود، امکان استفاده از سرویس میسر باشد؛ در غیر این صورت خیر. بسته به نوع پیاده سازی سرویس های Web api، پیاده سازی این بخش کمی متفاوت خواهد شد. در طی این پست این موضوع را برای سه حالت IIS Host و SelfHost و Owin Host بررسی می کنیم. در اینجا قصد داریم حالتی را پیاده سازی نماییم که اگر درخواست جاری از سوی کلاینتی بود که Ip آن در لیست Ip های غیر مجاز قرار داشت، ادامه ی عملیات متوقف شود.

IIS Hosting

حالت پیش فرض استفاده از سرویس های Web Api همین گزینه است؛ وابستگی مستقیم به System.Web. در مورد مزایا و معایب آن بحث نمی کنیم اما اگر این روش را انتخاب کردید تکه کد زیر این کار را برای ما انجام می دهد:

```
if (request.Properties.ContainsKey["MS_HttpContext"])
{
    var ctx = request.Properties["MS_HttpContext"] as HttpContextWrapper;
    if (ctx != null)
    {
        var ip = ctx.Request.UserHostAddress;
    }
}
```

برای بدست آوردن شی HttpContext می توان آن را از لیست Properties های درخواست جاری به دست آورد. حال کد بالا را در قالب یک Extension Method در خواهیم آورد؛ به صورت زیر:

```
public static class HttpRequestMessageExtensions
{
    private const string HttpContext = "MS_HttpContext";

    public static string GetClientIpAddress(this HttpRequestMessage request)
    {
        if (request.Properties.ContainsKey(HttpContext))
        {
            dynamic ctx = request.Properties[HttpContext];
            if (ctx != null)
            {
                return ctx.Request.UserHostAddress;
            }
        }
        return null;
    }
}
```

Self Hosting

در حالت Self Host می توان عملیات بالا را با استفاده از خاصیت [RemoteEndpointMessageProperty](#) انجام داد که تقریباً شبیه به حالت Web Host است. مقدار این خاصیت نیز در شی جاری *HttpRequestMessage* وجود دارد. فقط باید به صورت زیر آن را واکنشی نماییم:

```
if (request.Properties.ContainsKey[RemoteEndpointMessageProperty.Name])
{
    var remote = request.Properties[RemoteEndpointMessageProperty.Name] as RemoteEndpointMessageProperty;
```

```

    if (remote != null)
    {
        var ip = remote.Address;
    }
}

```

خاصیت [RemoteEndpointMessageProperty](#) به تمامی درخواست ها وارده در سرویس های WCF چه در حالت استفاده از Http و چه در حالت Tcp اضافه می شود و در اسمبلی System.ServiceModel نیز می باشد. از آنجا که Web Api از هسته ای WCF استفاده می کند (WCF Core) در نتیجه می توان از این روش استفاده نمود. فقط باید اسمبلی System.ServiceModel را به پروژه ای خود اضافه نمایید.

ترکیب حالت های قبلی:

اگر می خواهید کدهای نوشته شده شما وابستگی به نوع هاست پروژه نداشته باشد، یا به معنای دیگر، در هر دو حالت به درستی کار کند می توانید به روش زیر حالت های قبلی را با هم ترکیب کنید.
«در این صورت دیگر نیازی به اضافه کردن اسمبلی System.ServiceModel نیست.»

```

public static class HttpRequestMessageExtensions
{
    private const string HttpContext = "MS_HttpContext";
    private const string RemoteEndpointMessage =
        "System.ServiceModel.Channels.RemoteEndpointMessageProperty";

    public static string GetClientIpAddress(this HttpRequestMessage request)
    {
        if (request.Properties.ContainsKey(HttpContext))
        {
            dynamic ctx = request.Properties[HttpContext];
            if (ctx != null)
            {
                return ctx.Request.UserHostAddress;
            }
        }

        if (request.Properties.ContainsKey(RemoteEndpointMessage))
        {
            dynamic remoteEndpoint = request.Properties[RemoteEndpointMessage];
            if (remoteEndpoint != null)
            {
                return remoteEndpoint.Address;
            }
        }

        return null;
    }
}

```

مرحله بعدی طراحی یک DelegatingHandler جهت استفاده از IP به دست آمده است .

```

public class MyHandler : DelegatingHandler
{
    private readonly HashSet<string> deniedIps;

    protected override Task<HttpResponseMessage> SendAsync(HttpRequestMessage request,
        CancellationToken cancellationToken)
    {
        if (deniedIps.Contains(request.GetClientIpAddress()))
        {
            return Task.FromResult( new HttpResponseMessage( HttpStatusCode.Unauthorized ) );
        }

        return base.SendAsync(request, cancellationToken);
    }
}

```

: Owin

زمانی که از [Owin برای هاست سرویس های Web Api](#) خود استفاده می کنید کمی روال انجام کار متفاوت خواهد شد. در این مورد نیز می توانید از DelegatingHandler ها استفاده کنید. معرفی DelegatingHandler طراحی شده به Asp.Net PipeLine به صورت زیر خواهد بود:

```
public class Startup
{
    public void Configuration( IApplicationBuilder appBuilder )
    {
        var config = new HttpConfiguration();
        var routeHandler = HttpClientFactory.CreatePipeline( new HttpControllerDispatcher( config
), new DelegatingHandler[]
        {
            new MyHandler(),
        } );
        config.Routes.MapHttpRoute(
            name: "Default",
            routeTemplate: "{controller}/{action}",
            defaults: null,
            constraints: null,
            handler: routeHandler
        );
        config.EnsureInitialized();
        appBuilder.UseWebApi( config );
    }
}
```

اما نکته ای را که باید به آن دقت داشت، این است که یکی از مزایای استفاده از Owin، یکپارچه سازی عملیات هاستینگ قسمت های مختلف برنامه است. برای مثال ممکن است قصد داشته باشید که بخش هایی که با Asp.Net SignalR نیز پیاده سازی شده اند، قابلیت استفاده از کدهای بالا را داشته باشند. در این صورت بهتر است کل عملیات بالا در قالب یک Owin Middleware عمل نماید تا تمام قسمت های هاست شده ی برنامه از کدهای بالا استفاده نمایند؛ به صورت زیر:

```
public class IpMiddleware : OwinMiddleware
{
    private readonly HashSet<string> _deniedIps;

    public IpMiddleware(OwinMiddleware next, HashSet<string> deniedIps) :
        base(next)
    {
        _deniedIps = deniedIps;
    }

    public override async Task Invoke(OwinRequest request, OwinResponse response)
    {
        var ipAddress = (string)request.Environment["server.RemoteIpAddress"];
        if (_deniedIps.Contains(ipAddress))
        {
            response.StatusCode = 403;
            return;
        }

        await Next.Invoke(request, response);
    }
}
```

برای نوشتن یک Owin Middleware کافیست کلاس مورد نظر از کلاس OwinMiddleware ارث ببرد و متد Invoke را Override کنید. لیست Ip های غیر مجاز، از طریق سازنده در اختیار Middleware قرار می گیرد. اگر درخواست مجاز بود از طریق دستور Next.Invoke(request,response) کنترل برنامه به مرحله بعدی منتقل می شود در غیر صورت عملیات با کد 403 متوقف می شود. در نهایت برای معرفی این Middleware طراحی شده به Application، مراحل زیر را انجام دهید.

```
public class Startup
{
    public void Configuration( IApplicationBuilder appBuilder )
```

```
{
    var config = new HttpConfiguration();
    var deniedIps = new HashSet<string> {"192.168.0.100", "192.168.0.101"};

    app.Use(typeof(IpMiddleware), deniedIps);
    appBuilder.UseWebApi( config );
}
```

نظرات خوانندگان

نویسنده: امیر بختیاری
تاریخ: ۱۳۹۳/۰۶/۲۹ ۲۳:۲۹

با سلام؛ مطلب جالب و مفیدی بود فقط برای استفاده از UserHostAddress در یک پروژه در حال استفاده بودم بعد متوجه شدم تمامی لاگ‌ها با یک آی پی ثبت می‌شود بعد از جستجو فهمیدم که تمام درخواست‌ها از یک فایروال عبور می‌کند و تمام آی پی‌ها یکی می‌شود. به جاش از

```
Request.ServerVariables["HTTP_X_FORWARDED_FOR"]
```

استفاده کردم. البته خالی بودنش رو هم چک کردم و مشکلم حل شد. می‌خواستم بدونم راه حل دیگه ای هم داره یا نه. با تشکر

نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۳/۰۶/۳۰ ۱۳:۴۴

راه حل شما منطقی و درست است. در حالاتی که برای درخواست‌ها عمل forwarding صورت بگیرد تنها آدرسی که مشاهده خواهید کرد آدرس Proxy Server است. در نتیجه در این حالات مقدار آدرس اصلی در خاصیت **HTTP_X_FORWARDED_FOR** ذخیره خواهد شد. و مقدار خاصیت **REMOTE_ADDR** برابر با آدرس Proxy Server است. از آن جا که دستور `Request.UserHostAddress` برابر با کد زیر می‌باشد:

```
Request.ServerVariables["REMOTE_ADDR"]
```

دلیل یکی بودن تمام IP ها نیز همین است که شما همیشه آدرس Proxy Server را مشاهده می‌کنید.

یکی از گزینه های میزبانی WebAPI و SignalR حالت SelfHost می باشد که روش آن قبلا در مطلب « [نگاهی به گزینه های مختلف](#) » [مهیای جهت میزبانی SignalR](#) توضیح داده شده است.

ابتدا نگاه کوچکی به یک مثال داشته باشیم:
هاب زیر را در نظر بگیرید.

```
public class MessageHub : Hub
{
    public void NotifyAllClients()
    {
        Clients.All.Notify();
    }
}
```

برای selfHost کردن از یک برنامه ی کنسول استفاده می کنیم:

```
static void Main(string[] args)
{
    const string baseAddress = "http://localhost:9000/"; // "http://*:9000/";
    using (var webapp = WebApp.Start<Startup>(baseAddress))
    {
        Console.WriteLine("Start app...");

        var hubConnection = new HubConnection(baseAddress);
        IHubProxy messageHubProxy = hubConnection.CreateHubProxy("messageHub");

        messageHubProxy.On("notify", () =>
        {
            Console.WriteLine();
            Console.WriteLine("Notified!");
        });

        hubConnection.Start().Wait();

        Console.WriteLine("Start signalr...");

        bool dontExit = true;
        while (dontExit)
        {
            var key = Console.ReadKey();
            if (key.Key == ConsoleKey.Escape) dontExit = false;

            messageHubProxy.Invoke("NotifyAllClients");
        }
    }
}
```

با کلاس start-up ذیل:

```
public partial class Startup
{
    public void Configuration(IAppBuilder appBuilder)
    {
        var hubConfiguration = new HubConfiguration()
        {
            EnableDetailedErrors = true
        };

        appBuilder.MapSignalR(hubConfiguration);
        appBuilder.UseCors(CorsOptions.AllowAll);
    }
}
```



```
}
}
```

اکنون اگر برنامه را اجرا کنیم، با زدن هر کلید در کنسول، یک پیغام چاپ می‌شود که نشان دهنده صحت کارکرد هاب پیام می‌باشد.

خوب؛ تا الان همه چیز درست کار میکند.

صورت مساله:

معمولا برای منظم کردن و مدیریت بهتر کدهای نرم افزار، آن‌ها را در پروژه‌های مجزا یا در واقع همان class library های مجزا نگاه داری میکنیم.

اکنون در برنامه‌ی فوق ، اگر کلاس messageHub را به یک class library دیگر منتقل کنیم و آن را به برنامه‌ی کنسول ارجاع دهیم و برنامه را مجدد اجرا کنیم، با خطای زیر مواجه می‌شویم:

```
{"StatusCode": 500, "ReasonPhrase": "Internal Server Error", "Version": 1.1, "Content":
System.Net.Http.StreamContent, Headers:{"Date": "Mon, 27 Oct 2014 09:36:48 GMT", "Server":
Microsoft-HTTPAPI/2.0", "Content-Length": 0}}
```

مشکل چیست؟

همانطور که در مطلب « [نگاهی به گزینه‌های مختلف مهبای جهت میزبانی SignalR](#) » عنوان شده‌است، «در حالت SelfHost بر خلاف روش asp.net hosting ، اسمبلی‌های ارجاعی برنامه اسکن نمی‌شوند» و طبیعتا مشکل رخ داده شده در بالا از اینجا ناشی می‌شود.

راه حل:

- این کار باید به صورت دستی انجام پذیرد. با افزودن کد زیر به ابتدای برنامه (قبل از شروع هر کدی) اسمبلی‌های مورد نظر افزوده می‌شوند:

```
AppDomain.CurrentDomain.Load(typeof(MessageHub).Assembly.FullName);
```

طبیعتا افزودن دستی هر اسمبلی مشکل و در خیلی مواقع ممکن است با خطای انسانی فراموش کردن مواجه شود!
کد خودکار زیر، میتواند تکمیل کننده‌ی راه حل بالا باشد:

```
class LoadAssemblyHelper
{
    public static void Load(string searchPattern)
    {
        var path = Assembly.GetExecutingAssembly().Location;
        var entityAssemblies = Directory.GetFiles(Path.GetDirectoryName(path), searchPattern:
searchPattern);
        var assemblyNames = entityAssemblies.Select(e => AssemblyName.GetAssemblyName(e)).ToList();
        assemblyNames.ToList().ForEach(e => AppDomain.CurrentDomain.Load(e));
    }
}
```

و برای فراخوانی آن در ابتدای برنامه می‌نویسیم:

```
static void Main(string[] args)
{
    //AppDomain.CurrentDomain.Load(typeof(MessageHub).Assembly.FullName);
    //AppDomain.CurrentDomain.Load(typeof(MessageController).Assembly.FullName);

    LoadAssemblyHelper.Load("myFramework.*.dll");

    const string baseAddress = "http://*:9000/";
    using (var webapp = WebApp.Start<Startup>(baseAddress))
    {
        ...
    }
}
```

نکته‌ی مهم

این خطا و راه حل آن، در مورد hubهای signalr و هم controllerهای webapi صادق می‌باشد.

Kendo UI DataSource جهت تامین داده‌های سمت کلاینت [ویجت‌های مختلف KendoUI](#) طراحی شده‌است و به عنوان یک اینترفیس استاندارد قابل استفاده توسط تمام کنترل‌های داده‌ای Kendo UI کاربرد دارد. Kendo UI DataSource امکان کار با منابع داده محلی، مانند اشیاء و آرایه‌های جاوا اسکریپتی و همچنین منابع تامین شده از راه دور، مانند JSONP، JSON و XML را دارد. به علاوه توسط آن می‌توان اعمال ثبت، ویرایش و حذف اطلاعات، به همراه صفحه بندی، گروه بندی و مرتب سازی داده‌ها را کنترل کرد.

استفاده از منابع داده محلی

در ادامه مثالی را از نحوه‌ی استفاده از یک منبع داده محلی جاوا اسکریپتی، مشاهده می‌کنید:

```
<script type="text/javascript">
$(function () {
    var cars = [
        { "Year": 2000, "Make": "Hyundai", "Model": "Elantra" },
        { "Year": 2001, "Make": "Hyundai", "Model": "Sonata" },
        { "Year": 2002, "Make": "Toyota", "Model": "Corolla" },
        { "Year": 2003, "Make": "Toyota", "Model": "Yaris" },
        { "Year": 2004, "Make": "Honda", "Model": "CRV" },
        { "Year": 2005, "Make": "Honda", "Model": "Accord" },
        { "Year": 2000, "Make": "Honda", "Model": "Accord" },
        { "Year": 2002, "Make": "Kia", "Model": "Sedona" },
        { "Year": 2004, "Make": "Fiat", "Model": "One" },
        { "Year": 2005, "Make": "BMW", "Model": "M3" },
        { "Year": 2008, "Make": "BMW", "Model": "X5" }
    ];

    var carsDataSource = new kendo.data.DataSource({
        data: cars
    });

    carsDataSource.read();
    alert(carsDataSource.total());
});
</script>
```

در اینجا cars آرایه‌ای از اشیاء جاوا اسکریپتی بیانگر ساختار یک خودرو است. سپس برای معرفی آن به Kendo UI، کار با مقدار دهی خاصیت data مربوط به new kendo.data.DataSource شروع می‌شود. ذکر new kendo.data.DataSource به تنهایی به معنای مقدار دهی اولیه است و در این حالت منبع داده مورد نظر، استفاده نخواهد شد. برای مثال اگر متد total آن را جهت یافتن تعداد عناصر موجود در آن فراخوانی کنید، صفر را بازگشت می‌دهد. برای شروع به کار با آن، نیاز است ابتدا متد read را بر روی این منبع داده مقدار دهی شده، فراخوانی کرد.

استفاده از منابع داده راه دور

در برنامه‌های کاربردی، عموماً نیاز است تا منبع داده را از یک وب سرور تامین کرد. در اینجا نحوه‌ی خواندن اطلاعات JSON بازگشت داده شده از جستجوی توئیتر را مشاهده می‌کنید:

```
<script type="text/javascript">
$(function () {
    var twitterDataSource = new kendo.data.DataSource({
        transport: {
            read: {
                url: "http://search.twitter.com/search.json",
                dataType: "jsonp",
                contentType: 'application/json; charset=utf-8',
                type: 'GET',
                data: { q: "#kendoui" }
            },
            schema: { data: "results" }
        }
    });
});
```

```

    },
    error: function (e) {
        alert(e.errorThrown.stack);
    }
  });
});
</script>

```

در قسمت transport، جزئیات تبادل اطلاعات با سرور راه دور مشخص می‌شود؛ برای مثال url ارائه دهنده‌ی سرویس، dataType بیانگر نوع داده مورد انتظار و data کار مقدار دهی پارامتر مورد انتظار توسط سرویس توئیتر را انجام می‌دهد. در اینجا چون صرفاً عملیات خواندن اطلاعات صورت می‌گیرد، خاصیت read مقدار دهی شده‌است. در قسمت schema مشخص می‌کنیم که اطلاعات JSON بازگشت داده شده توسط توئیتر، در فیلد results آن قرار دارد.

کار با منابع داده OData

علاوه بر فرمت‌های یاد شده، Kendo UI DataSource امکان کار با اطلاعاتی [از نوع OData](#) را نیز دارا است که تنظیمات ابتدایی آن به صورت ذیل است:

```

<script type="text/javascript">
    var moviesDataSource = new kendo.data.DataSource({
        type: "odata",
        transport: {
            read: "http://demos.kendoui.com/service/Northwind.svc/Orders"
        },
        error: function (e) {
            alert(e.errorThrown.stack);
        }
    });
});
</script>

```

همانطور که ملاحظه می‌کنید، تنظیمات ابتدایی آن اندکی با حالت remote data پیشین متفاوت است. در اینجا ابتدا نوع داده‌ی بازگشتی مشخص می‌شود و در قسمت transport، خاصیت read آن، آدرس سرویس را دریافت می‌کند.

یک مثال: دریافت اطلاعات از ASP.NET Web API

یک پروژه‌ی جدید ASP.NET را آغاز کنید. تفاوتی نمی‌کند که Web forms باشد یا MVC؛ از این جهت که مباحث [Web API](#) در هر دو یکسان است.

سپس یک کنترلر جدید Web API را به نام ProductsController با محتوای زیر ایجاد کنید:

```

using System.Collections.Generic;
using System.Web.Http;

namespace KendoUI02
{
    public class Product
    {
        public int Id { set; get; }
        public string Name { set; get; }
    }

    public class ProductsController : ApiController
    {
        public IEnumerable<Product> Get()
        {
            var products = new List<Product>();
            for (var i = 1; i <= 100; i++)
            {
                products.Add(new Product { Id = i, Name = "Product " + i });
            }
            return products;
        }
    }
}

```

در این مثال، هدف صرفاً ارائه یک خروجی ساده JSON از طرف سرور است.
در ادامه نیاز است تعریف مسیریابی ذیل نیز به فایل Global.asax.cs برنامه اضافه شود تا بتوان به آدرس api/products سایت، دسترسی یافت:

```
using System;
using System.Web.Http;
using System.Web.Routing;

namespace KendoUI02
{
    public class Global : System.Web.HttpApplication
    {
        protected void Application_Start(object sender, EventArgs e)
        {
            RouteTable.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );
        }
    }
}
```

در ادامه فایلی را به نام Index.html (یا در یک View و یا یک فایل aspx دلخواه)، محتوای ذیل را اضافه کنید:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>Kendo UI: Implementing the Grid</title>

    <link href="styles/kendo.common.min.css" rel="stylesheet" type="text/css" />
    <link href="styles/kendo.default.min.css" rel="stylesheet" type="text/css" />
    <script src="js/jquery.min.js" type="text/javascript"></script>
    <script src="js/kendo.all.min.js" type="text/javascript"></script>
</head>
<body>

    <div id="report-grid"></div>
    <script type="text/javascript">
        $(function () {
            var productsDataSource = new kendo.data.DataSource({
                transport: {
                    read: {
                        url: "api/products",
                        dataType: "json",
                        contentType: 'application/json; charset=utf-8',
                        type: 'GET'
                    }
                },
                error: function (e) {
                    alert(e.errorThrown.stack);
                },
                pageSize: 5,
                sort: { field: "Id", dir: "desc" }
            });

            $("#report-grid").kendoGrid({
                dataSource: productsDataSource,
                autoBind: true,
                scrollable: false,
                pageable: true,
                sortable: true,
                columns: [
                    { field: "Id", title: "#" },
                    { field: "Name", title: "Product" }
                ]
            });
        });
    </script>
</body>
</html>
```

- ابتدا فایل‌های اسکریپت و CSS مورد نیاز Kendo UI اضافه شده‌اند.
- گرید صفحه، در محل div ایی با id مساوی report-grid تشکیل خواهد شد.
- سپس DataSource ایی که به آدرس api/products اشاره می‌کند، تعریف شده و در آخر productsDataSource را توسط یک kendoGrid نمایش داده‌ایم.
- نحوه‌ی تعریف productsDataSource، در قسمت استفاده از منابع داده راه دور ابتدای بحث توضیح داده شد. در اینجا فقط دو خاصیت pageSize و sort نیز به آن اضافه شده‌اند. این دو خاصیت بر روی نحوه‌ی نمایش گرید نهایی تاثیر گذار هستند. تعداد رکورد هر صفحه را مشخص می‌کند و sort نحوه‌ی مرتب سازی را بر اساس فیلد Id و در حالت نزولی قرار می‌دهد.
- در ادامه، ابتدایی‌ترین حالت کار با kendoGrid را ملاحظه می‌کنید.
- تنظیم dataSource و autoBind: true (حالت پیش فرض)، سبب خواهند شد تا به صورت خودکار، اطلاعات JSON از مسیر api/products خوانده شوند.
- سه خاصیت بعدی صفحه بندی و مرتب سازی خودکار ستون‌ها را فعال می‌کنند.
- در آخر هم دو ستون گرید، بر اساس نام‌های خواص کلاس Product تعریف شده‌اند.

#	Product
5	Product 5
4	Product 4
3	Product 3
2	Product 2
1	Product 1

سورس کامل این قسمت را از اینجا می‌توانید دریافت کنید:

[KendoUI02.zip](#)

نظرات خوانندگان

نویسنده:

ژوپيتر

10:27 1393/11/09

سلام

چرا زمان اجرا به جای نمایش اطلاعات گریذ، پیام undefined داده می‌شود؟ بنده از MVC استفاده کردم و کاملاً مطابق مقاله مسیریابی و ... را اعمال کردم.

نویسندہ:

وحید نصیری

11:10 1393/11/09

روی چه سطری پیام خطا دریافت کردید؟

حین کار با کتابخانه‌های جاوا اسکریپتی باید مدام کنسول developer مرورگر را باز نگه دارید تا بتوانید خطاها را بهتر بررسی و دیباگ کنید.

نویسنده:

ژوپيتر

9:3 1393/11/11

خطا توسط error handler, گرفته می‌شود وقتی این بخش نیز حذف می‌شود مرورگر فقط منتظر دریافت اطلاعات از api/products می‌ماند و خطایی از کد گرفته نمی‌شود.

The screenshot shows the Chrome DevTools Network tab with 13 requests. The 'GET products' request is highlighted in red, indicating a 404 Not Found status. The total size of requests is 2.1 MB (2.0 MB from cache) and the total time is 446ms (onload: 1.53s).

Request	Status	Size	Time
GET kendo.common.min.css	304 Not Modified	40.8 KB	1ms
GET kendo.default.min.css	304 Not Modified	9.3 KB	1ms
GET jquery-1.10.2.min.js	304 Not Modified	91.9 KB	2ms
GET kendo.all.min.js	304 Not Modified	1.9 MB	1ms
GET browserLink	200 OK	58.3 KB	8ms
GET sprite.png	304 Not Modified	27.8 KB	2ms
GET products	404 Not Found	4.3 KB	47ms
GET loading-image.gif	304 Not Modified	6.0 KB	2ms
GET negotiate?requestUrl=ht..c	200 OK	606 B	1ms
GET connect?transport=webSo..	101 Switching Protocols	0 B	4ms
GET 01a1cc15e9d44731ab743a5	200 OK	366 B	6ms

13 requests 2.1 MB (2.0 MB from cache) 446ms (onload: 1.53s)

The screenshot shows the Chrome DevTools Network tab. A GET request to `http://localhost:30523/Home/api/products` is selected. The request status is 200 OK. The response headers are expanded, showing the following information:

- Cache-Control:** private
- Content-Length:** 4412
- Content-Type:** text/html; charset=utf-8
- Date:** Sat, 31 Jan 2015 05:20:19 GMT
- Server:** Microsoft-IIS/8.0
- X-AspNet-Version:** 4.0.30319
- X-Powered-By:** ASP.NET
- X-SourceFiles:** =?UTF-8?B?QxcvXN1cnNwOec2VuXER1c2t0b3BcTVZDV2ViQXBwbG1jYXRpb2ScTVZDV2ViQXBwbG1jYXRpb2Sc2Sc9gc2VZVhcG1ccHJvZHVhZHM=

نویسنده:

وحید نصیری

10:3 1393/11/11

خطای 404 به معنای یافتن نشدن مسیر تنظیمی `url: "api/products"` در برنامه شما است.

مطابق تصویر، مسیر `home /api/product` در حال جستجو است که به ریشه‌ی سایت اشاره نمی‌کند.

- آیا در ASP.NET MVC از یک اکشن متد برای بازگرداندن لیست جی‌سون محصولات استفاده کرده‌اید؟ اگر بله، از مطلب «[نحوه صحیح تولید Url در ASP.NET MVC](#)» کمک بگیرید؛ مثلاً آدرس آن چنین شکلی را پیدا خواهد کرد:

```
@Url.Action("method_name", "Home")
```

- اگر وب API است در یک برنامه‌ی MVC، از روش زیر استفاده کنید:

```
'@Url.RouteUrl("DefaultApi", new { httproute = "", controller = "products" })'
```

و البته فرض بر این است که مسیریابی DefaultApi پیشتر در برنامه‌ی شما ثبت شده‌است:

```
routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

نویسنده: ژوپیتز

تاریخ: ۱۱:۸ ۱۳۹۳/۱۱/۱۱

- ممنون؛ از وب API استفاده شده بود که با راهنمایی شما حل شد. ولی استفاده از خروجی Json کنترلر به‌نظرم بهتر و ساده‌تر اومد. آیا تفاوتی محسوسی بین این دو روش وجود داره؟

- آیا امکان استفاده مستقیم اشیا Strongly Typed هم در توابع این کتابخانه وجود داره؟ (منظورم همون @model به صورت مستقیم یا با واسطه است).

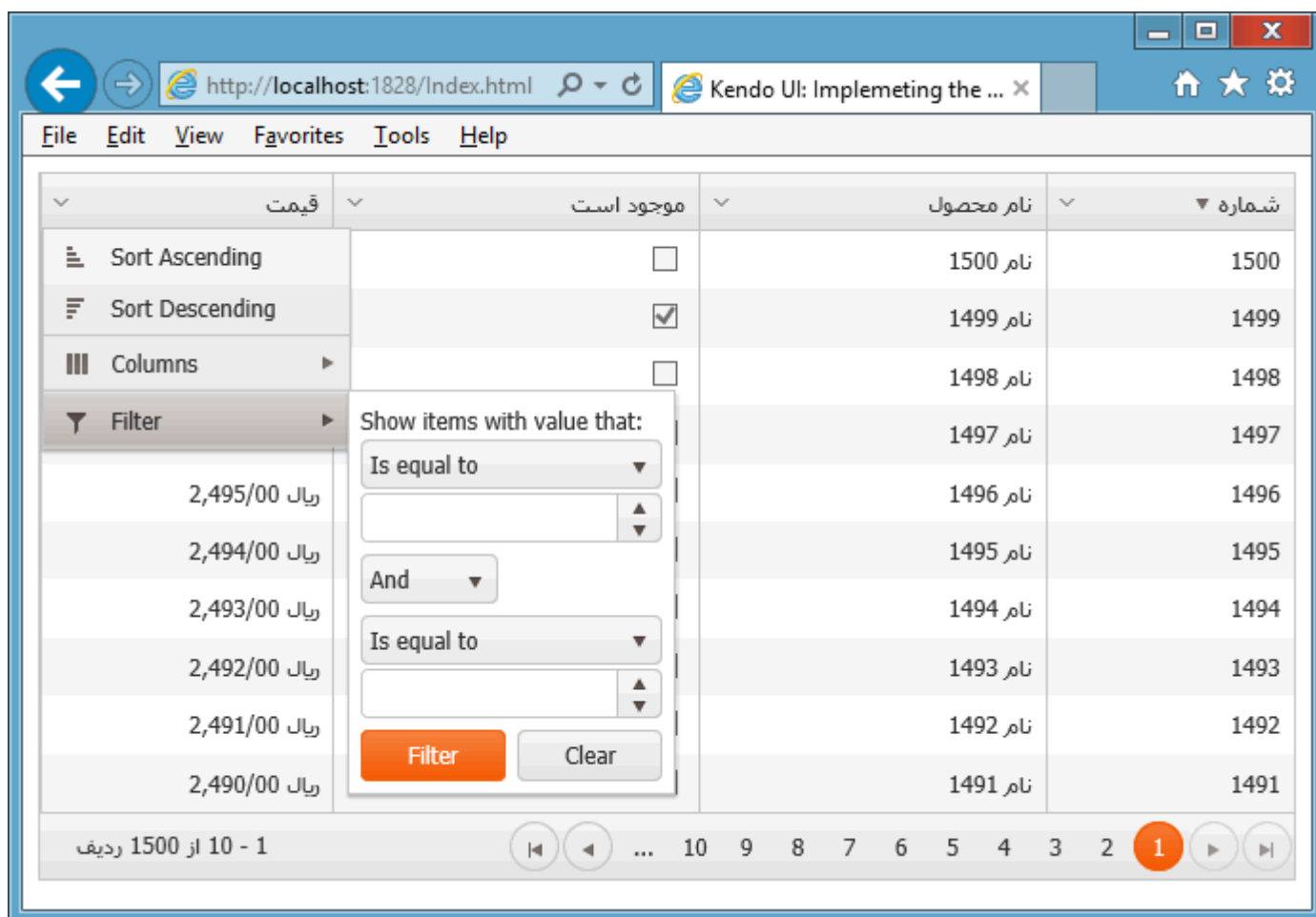
نویسنده: وحید نصیری

تاریخ: ۱۱:۲۰ ۱۳۹۳/۱۱/۱۱

- خیر. اگر هدف صرفاً بازگشت جی‌سون است، تفاوت خاصی ندارند. فقط Web API به صورت پیش فرض از JSON.NET استفاده می‌کند؛ [اطلاعات بیشتر](#) و همچنین با وب فرم‌ها هم سازگار است.

- بله: [استفاده از Expressionها جهت ایجاد Strongly typed view در ASP.NET MVC](#)

پس از آشنایی مقدماتی با [Kendo UI DataSource](#)، اکنون می‌خواهیم از آن جهت صفحه بندی، مرتب سازی و جستجوی پویای سمت سرور استفاده کنیم. در مثال قبلی، هر چند صفحه بندی فعال بود، اما پس از دریافت تمام اطلاعات، این اعمال در سمت کاربر انجام و مدیریت می‌شد.



مدل برنامه

در اینجا قصد داریم لیستی را با ساختار کلاس Product در اختیار Kendo UI گرید قرار دهیم:

```
namespace KendoUI03.Models
{
    public class Product
    {
        public int Id { set; get; }
        public string Name { set; get; }
        public decimal Price { set; get; }
        public bool IsAvailable { set; get; }
    }
}
```

پیشنیاز تامین داده مخصوص Kendo UI Grid

برای ارائه اطلاعات مخصوص Kendo UI Grid، ابتدا باید در نظر داشت که این گرید، درخواست های صفحه بندی خود را با فرمت ذیل ارسال می کند. همانطور که مشاهده می کنید، صرفاً یک کوئری استرینگ با فرمت JSON را دریافت خواهیم کرد:

```
/api/products?{"take":10,"skip":0,"page":1,"pageSize":10,"sort":[{"field":"Id","dir":"desc"}]}
```

سپس این گرید نیاز به سه فیلد، در خروجی JSON نهایی خواهد داشت:

```
{
  "Data":
  [
    {"Id":1500,"Name":"1500 نام","Price":2499.0,"IsAvailable":false},
    {"Id":1499,"Name":"1499 نام","Price":2498.0,"IsAvailable":true}
  ],
  "Total":1500,
  "Aggregates":null
}
```

فیلد Data که رکوردهای گرید را تامین می کند. فیلد Total که بیانگر تعداد کل رکوردها است و Aggregates که برای گروه بندی بکار می رود.

می توان برای تمام این ها، کلاس و Parser تهیه کرد و یا ... پروژه های سورس بازی به نام [Kendo.DynamicLinq](#) نیز چنین کاری را میسر می سازد که در ادامه از آن استفاده خواهیم کرد. برای نصب آن تنها کافی است دستور ذیل را صادر کنید:

```
PM> Install-Package Kendo.DynamicLinq
```

Kendo.DynamicLinq به صورت خودکار [System.Linq.Dynamic](#) را نیز نصب می کند که از آن جهت صفحه بندی پویا استفاده خواهد شد.

تامین کننده ی داده سمت سرور

همانند مطلب کار با [Kendo UI DataSource](#)، یک ASP.NET Web API Controller جدید را به پروژه اضافه کنید و همچنین مسیریابی های مخصوص آن را به فایل global.asax.cs نیز اضافه نمایید.

```
using System.Linq;
using System.Net.Http;
using System.Web.Http;
using Kendo.DynamicLinq;
using KendoUI03.Models;
using Newtonsoft.Json;

namespace KendoUI03.Controllers
{
    public class ProductsController : ApiController
    {
        public DataSourceResult Get(HttpRequestMessage requestMessage)
        {
            var request = JsonConvert.DeserializeObject<DataSourceRequest>(
                requestMessage.RequestUri.ParseQueryString().GetKey(0)
            );

            var list = ProductDataSource.LatestProducts;
            return list.AsQueryable()
                .ToDataSourceResult(request.Take, request.Skip, request.Sort, request.Filter);
        }
    }
}
```

تمام کدهای این کنترلر همین چند سطر فوق هستند. با توجه به ساختار کوئری استرینگی که در ابتدای بحث عنوان شد، نیاز است آنرا توسط کتابخانه‌ی [JSON.NET](#) تبدیل به یک نمونه از [DataSourceRequest](#) نمائیم. این کلاس در Kendo.DynamicLinq تعریف شده است و حاوی اطلاعاتی مانند take و skip کوئری LINQ نهایی است.

ProductDataSource.LatestProducts صرفاً یک لیست جنریک تهیه شده از کلاس Product است. در نهایت با استفاده از متد الحاقی جدید [ToDataSourceResult](#)، به صورت خودکار مباحث صفحه بندی سمت سرور به همراه مرتب سازی اطلاعات، صورت گرفته و اطلاعات نهایی با فرمت [DataSourceResult](#) بازگشت داده می‌شود. DataSourceResult نیز در Kendo.DynamicLinq تعریف شده و سه فیلد یاد شده‌ی Data، Total و Aggregates را تولید می‌کند.

تا اینجا کارهای سمت سرور این مثال به پایان می‌رسد.

تهیه View نمایش اطلاعات ارسالی از سمت سرور

اعمال مباحث بومی سازی

```
<head>
  <meta charset="utf-8" />
  <meta http-equiv="Content-Language" content="fa" />
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

  <title>Kendo UI: Implementing the Grid</title>

  <link href="styles/kendo.common.min.css" rel="stylesheet" type="text/css" />
  <!-- شیوه نامی مخصوص راست به چپ سازی -->
  <link href="styles/kendo.rtl.min.css" rel="stylesheet" />
  <link href="styles/kendo.default.min.css" rel="stylesheet" type="text/css" />
  <script src="js/jquery.min.js" type="text/javascript"></script>
  <script src="js/kendo.all.min.js" type="text/javascript"></script>

  <!-- محل سفارشی سازی پیام‌ها و مسایل بومی -->
  <script src="js/cultures/kendo.culture.fa-IR.js" type="text/javascript"></script>
  <script src="js/cultures/kendo.culture.fa.js" type="text/javascript"></script>
  <script src="js/messages/kendo.messages.en-US.js" type="text/javascript"></script>

  <style type="text/css">
    body {
      font-family: tahoma;
      font-size: 9pt;
    }
  </style>

  <script type="text/javascript">
    // جهت استفاده از فایل kendo.culture.fa-IR.js
    kendo.culture("fa-IR");
  </script>
</head>
```

- در اینجا چند فایل js و css جدید اضافه شده‌اند. فایل kendo.rtl.min.css جهت تامین مباحث RTL توکار Kendo UI کاربرد دارد.

- سپس سه فایل kendo.culture.fa.js، kendo.culture.fa-IR.js و kendo.messages.en-US.js نیز اضافه شده‌اند. فایل‌های fa و fa-IR آن‌ها هر چند به ظاهر برای ایران طراحی شده‌اند، اما نام ماه‌های موجود در آن عربی است که نیاز به ویرایش دارد. به همین جهت به سورس این فایل‌ها، جهت ویرایش نهایی نیاز خواهد بود که در پوشه‌ی src\js\cultures مجموعه‌ی اصلی Kendo UI موجود هستند (ر.ک. فایل پیوست).

- فایل kendo.messages.en-US.js حاوی تمام پیام‌های مرتبط با Kendo UI است. برای مثال «رکوردهای 10 تا 15 از 1000 ردیف» را در اینجا می‌توانید به فارسی ترجمه کنید.

- متد kendo.culture کار مشخص سازی فرهنگ بومی برنامه را به عهده دارد. برای مثال در اینجا به fa-IR تنظیم شده‌است. این مورد سبب خواهد شد تا از فایل kendo.culture.fa-IR.js استفاده گردد. اگر مقدار آن‌را به fa تنظیم کنید، از فایل kendo.culture.fa.js کمک گرفته خواهد شد.

راست به چپ سازی گرید

تنها کاری که برای راست به چپ سازی Kendo UI Grid باید صورت گیرد، محصور سازی div آن در یک div با کلاس مساوی k-

rtl است:

```
<div class="k-rtl">
  <div id="report-grid"></div>
</div>
```

k-rtl و تنظیمات آن در فایل kendo.rtl.min.css قرار دارند که در ابتدای head صفحه تعریف شده است.

تامین داده و نمایش گرید

در ادامه کدهای کامل DataSource و Kendo UI Grid را ملاحظه می کنید:

```
<script type="text/javascript">
$(function () {
    var productsDataSource = new kendo.data.DataSource({
        transport: {
            read: {
                url: "api/products",
                dataType: "json",
                contentType: 'application/json; charset=utf-8',
                type: 'GET'
            },
            parameterMap: function (options) {
                return kendo.stringify(options);
            }
        },
        schema: {
            data: "Data",
            total: "Total",
            model: {
                fields: {
                    "Id": { type: "number" }, // تعیین نوع فیلد برای جستجوی پویا مهم است
                    "Name": { type: "string" },
                    "IsAvailable": { type: "boolean" },
                    "Price": { type: "number" }
                }
            }
        },
        error: function (e) {
            alert(e.errorThrown);
        },
        pageSize: 10,
        sort: { field: "Id", dir: "desc" },
        serverPaging: true,
        serverFiltering: true,
        serverSorting: true
    });

    $("#report-grid").kendoGrid({
        dataSource: productsDataSource,
        autoBind: true,
        scrollable: false,
        pageable: true,
        sortable: true,
        filterable: true,
        reorderable: true,
        columnMenu: true,
        columns: [
            { field: "Id", title: "شماره", width: "130px" },
            { field: "Name", title: "نام محصول",
                template: '<input type="checkbox" #= IsAvailable ? checked="checked" : "" #>'
            },
            { field: "Price", title: "قیمت", format: "{0:c}" }
        ]
    });
});
</script>
```

- با تعاریف مقدماتی Kendo UI DataSource [پیشتر آشنا شده ایم](#) و قسمت read آن جهت دریافت اطلاعات از سمت سرور کاربرد

دارد.

- در اینجا ذکر contentType الزامی است. زیرا ASP.NET Web API بر این اساس است که تصمیم می‌گیرد، خروجی را به صورت JSON ارائه دهد یا XML.
- با استفاده از parameterMap، سبب خواهیم شد تا پارامترهای ارسالی به سرور، با فرمت صحیحی تبدیل به JSON شده و بدون مشکل به سرور ارسال گردند.
- در قسمت schema باید نام فیلدهای موجود در DataSourceResult دقیقاً مشخص شوند تا گرید بداند که data را باید از چه فیلدی استخراج کند و تعداد کل ردیف‌ها در کدام فیلد قرار گرفته‌است.
- نحوه‌ی تعریف model را نیز در اینجا ملاحظه می‌کنید. ذکر نوع فیلدها در اینجا بسیار مهم است و اگر قید نشوند، در حین جستجوی پویا به مشکل برخوردیم خورد. زیرا پیش فرض نوع تمام فیلدها string است و در این حالت نمی‌توان عدد 1 رشته‌ای را با یک فیلد از نوع int در سمت سرور مقایسه کرد.
- در اینجا serverFiltering، serverPaging و serverSorting نیز به true تنظیم شده‌اند. اگر این مقدار دهی‌ها صورت نگیرد، این اعمال در سمت کلاینت انجام خواهند شد.

پس از تعریف DataSource، تنها کافی است آن را به خاصیت dataSource یک kendoGrid نسبت دهیم.

- autoBind: true سبب می‌شود تا اطلاعات DataSource بدون نیاز به فراخوانی متد read آن به صورت خودکار دریافت شوند.
- با تنظیم scrollable: false، اعلام می‌کنیم که قرار است تمام رکوردها در معرض دید قرار گیرند و اسکرول پیدا نکنند.
- pageable: true صفحه بندی را فعال می‌کند. این مورد نیاز به تنظیم pageSize: 10 در قسمت DataSource نیز دارد.
- با sortable: true مرتب سازی ستون‌ها با کلیک بر روی سرستون‌ها فعال می‌گردد.
- filterable: true به معنای فعال شدن جستجوی خودکار بر روی فیلدها است. کتابخانه‌ی Kendo.DynamicLinq حاصل آن را در سمت سرور مدیریت می‌کند.
- reorderable: true سبب می‌شود تا کاربر بتواند محل قرارگیری ستون‌ها را تغییر دهد.
- columnMenu: true اختیاری است. اگر ذکر شود، امکان مخفی سازی انتخابی ستون‌ها نیز مسیر خواهد شد.
- در آخر ستون‌های گرید مشخص شده‌اند. با تعیین format: "{0:c}" سبب نمایش فیلدهای قیمت با سه رقم جدا کننده خواهیم شد. مقدار ریال آن از فایل فرهنگ جاری تنظیم شده دریافت می‌گردد. با استفاده از template تعریف شده نیز سبب نمایش فیلد bool به صورت یک checkbox خواهیم شد.

کدهای کامل این مثال را از اینجا می‌توانید دریافت کنید:

[KendoUI03.zip](#)

نظرات خوانندگان

نویسنده: حمیدرضا کبیری
تاریخ: ۱۹:۳۱ ۱۳۹۳/۰۸/۱۶

آیا kendo UI کاملاً از زبان فارسی پشتیبانی میکند ؟
برای calender آن ، به تقویم شمسی گزینه ای موجود هست ؟
این [گزینه](#) با ورژن ۲۰۱۴/۱/۳۱۸ مطابقت دارد ، آیا با ورژنهای جدید مشکلی نخواهد داشت ؟

نویسنده: احمد رجبی
تاریخ: ۲۰:۱۵ ۱۳۹۳/۰۸/۱۶

میتوانید با اضافه کردن [این اسکریپت](#) تمامی قسمتهای kendo را به زبان فارسی ترجمه کنید.

نویسنده: سعیدجلالی
تاریخ: ۸:۴۴ ۱۳۹۳/۰۸/۱۷

با تشکر از مطلب مفید شما من از wrapper mvc مجموعه kendo استفاده میکنم
توی مطالب شما در مورد استفاده از [Kendo.DynamicLinq](#) صحبت شد خواستم بدونم آیا وقتی از wrapper هم استفاده میکنیم
استفاده از این پکیج لازم هست؟

```
@(Html.Kendo().Grid(Model)
    .Name("gridKendo")
    .Columns(columns =>
    {
        columns.Bound(c => c.Name).Width(50);
        columns.Bound(c => c.Family).Width(100);
        columns.Bound(c => c.Tel);
    })
    .Pageable(pager => pager
        .Input(true)
        .Numeric(true)
        .Info(true)
        .PreviousNext(true)
        .Refresh(true)
        .PageSizes(true)
    )
)
```

چون من با استفاده از telerik profiler وقتی درخواست رو بررسی میکنم توی دستور sql چنین دستوری رو در انتها مشاهده میکنم:
صفحه اول:

```
SELECT *
FROM (
    SELECT
    FROM table a
)
WHERE ROWNUM <= :TAKE
```

صفحات بعد:

```
SELECT *
FROM (
    SELECT
    a.*,
    ROWNUM OA_ROWNUM
    FROM (
    FROM table a
```

```
) a
WHERE ROWNUM <= :TAKE
)
WHERE OA_ROWNUM > :SKIP
```

پایگاه داده اوراکل است.

نویسنده: سعیدجلالی
تاریخ: ۹:۱۲ ۱۳۹۳/۰۸/۱۷

امکان فارسی شدن تمام بخش‌ها وجود دارد. تقویم هم فارسی شده است در [این سایت](#) برای نسخه‌های جدیدتر هم باید دوتا فایل جاوا اسکریپت all و mvc رو خودتون تغییر بدهید (با توجه به الگوی انجام شده در فایل فارسی شده فوق) ولی برای تقویم زمانبندی scheduler من فارسی ندیده ام

نویسنده: وحید نصیری
تاریخ: ۹:۳۱ ۱۳۹۳/۰۸/۱۷

مطلب فوق نه وابستگی خاصی به وب فرم‌ها دارد و نه ASP.NET MVC. ویو آن یک فایل HTML ساده است و سمت سرور آن فقط یک کنترلر ASP.NET Web API که با تمام مشتقات ASP.NET سازگار است. در این حالت یک نفر می‌تواند ASP.NET نگارش خودش را خلق کند؛ بدون اینکه نگران جزئیات وب فرم‌ها باشد یا ASP.NET MVC. ضمناً دانش جاوا اسکریپتی آن هم قابل انتقال است؛ چون اساساً Kendo UI برای فناوری سمت سرور خاصی طراحی نشده است و حالت اصل آن با PHP، Java و امثال آن هم کار می‌کند.

نویسنده: میثم آقا احمدی
تاریخ: ۱۳:۱۷ ۱۳۹۳/۰۸/۱۷

در کنترلر این خط باعث بارگذاری تمامی داده‌ها می‌شود

```
var list = ProductDataSource.LatestProducts;
```

آیا راه حلی وجود دارد که دیتای به تعداد همان pagesize از پایگاه خوانده شود؟

نویسنده: وحید نصیری
تاریخ: ۱۳:۲۸ ۱۳۹۳/۰۸/۱۷

- این فقط یک مثال هست و منبع داده‌ای صرفاً جهت دموی ساده‌ی برنامه. فقط برای اینکه با یک کلیک بتوانید برنامه را اجرا کنید و نیازی به برپایی و تنظیم بانک اطلاعاتی و امثال آن نداشته باشد.
- شما در کدها و کوئری‌های مثلاً EF در اصل با یک سری [IQueryable](#) کار می‌کنید. همینجا باید متد الحاقی ToDataSourceResult را اعمال کنید تا نتیجه‌ی نهایی در حداقل بار تعداد رفت و برگشت و با کوئری مناسبی بر اساس پارامترهای دریافتی به صورت خودکار تولید شود. در انتهای کار بجای مثلاً ToList بنویسید ToDataSourceResult.

نویسنده: امین
تاریخ: ۱۴:۴۱ ۱۳۹۳/۰۸/۱۷

سلام من در ویو خودم نمیتونم اطلاعاتم رو تو kendo.grid بینم و برای من یک لیست استرینگ در ویو نمایش داده میشه و به این شکل در کنترلر و ویو کد نویسی کردم .

```
public class EFController : Controller {
    //
    // GET: /EF/

    public ActionResult AjaxConnected([DataSourceRequest] DataSourceRequest request )
```

```

{
    using (var dbef=new dbTestEntities())
    {
        IQueryable<Person> persons = dbef.People;
        DataSourceResult result = persons.ToDataSourceResult(request);
        return Json(result.Data,JsonRequestBehavior.AllowGet);
    }
}

```

و ویدو

```

@{
    ViewBag.Title = "AjaxConnected";
}

<h2>AjaxConnected</h2>
@(Html.Kendo().Grid<TelerikMvcApp2.Models.Person>(
    .Name("Grid")
    .DataSource(builder => builder
        .Ajax()
        .Read(operationBuilder => operationBuilder.Action("AjaxConnected", "EF"))
    )
    .Columns(factory =>
    {
        factory.Bound(person => person.personId);
        factory.Bound(person => person.Name);
        factory.Bound(person => person.LastName);
    })
    .Pageable()
    .Sortable())

```

و یک لیست استرینگ بهم در عمل خروجی می‌دهد و از خود قالب kendo grid خبری نیست. من اطلاعات رو به طور json پاس میدم و ajaxی میگیرم.

حالا قبلش همچین خطایی داشتم که به allowget ایراد میگرفت ولی در کل با JsonRequestBehavior.AllowGet حل شد و حالا فقط به لیست بهم خروجی می‌دهد! و از ظاهر گرید خبری نیست. و اگر به جای json نوشته بشه view و با ویدو return کنم ظاهر kendo grid رو دارم اما خروجی دارای مقداری نیست! اینم خروجی استرینگ من:

```

[{"personId":1,"Name":"Amin","LastName":"Saadati"}, {"personId":2,"Name":"Fariba","LastName":"Ghochani"}, {"personId":3,"Name":"Rima","LastName":"rad"}, {"personId":4,"Name":"Milad","LastName":"Rahman"}, {"personId":5,"Name":"sahel","LastName":"abasi"}, {"personId":6,"Name":"ali","LastName":"kiva"}, {"personId":7,"Name":"mino","LastName":"kafash"}, {"personId":8,"Name":"medi","LastName":"ghaem"}, {"personId":9,"Name":"toti","LastName":"saadati"}, {"personId":10,"Name":"behzad","LastName":"tizro"}, {"personId":11,"Name":"sadeh","LastName":"hojati"}, {"personId":12,"Name":"parinaz","LastName":"karami"}, {"personId":13,"Name":"farid","LastName":"riazi"}, {"personId":14,"Name":"milad","LastName":"ebadipor"}, {"personId":15,"Name":"said","LastName":"abdoli"}, {"personId":16,"Name":"jamshid","LastName":"k"}, {"personId":17,"Name":"behzad","LastName":"ariaf"}, {"personId":18,"Name":"jamshid","LastName":"k"}]

```

این سوال رو در چند سایت پرسیدم و به جوابی برایش نرسیدم. و نمیدونم ایراد کدهای نوشته شده ام کجاست! متشکرم

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۸/۱۷ ۱۵:۲

- قصد پشتیبانی از wrapperهای آنرا ندارم. لطفا خارج از موضوع سؤال نپرسید. اگر کسی دوست داشت در این زمینه مطلب منتشر کند، خوب. ولی من چنین قصدی ندارم.

- عرض کردم اگر از wrapperها استفاده کنید، به علت عدم درک زیر ساخت اصلی Kendo UI، قادر به دیباگ کار نخواهید بود.

- اگر متن را مطالعه کنید در قسمت «پیشنیاز تامین داده مخصوص Kendo UI Grid» دقیقا شکل نهایی خروجی JSON مورد نیاز ارائه شده‌است. این خروجی در سه فیلد data, total و aggregate قرار می‌گیرد. شما الان فقط قسمت data آنرا بازگشت

داده‌اید؛ بجای اصل و کل آن. نام این سه فیلد هم مهم نیست؛ اما هر چیزی که تعیین می‌شوند، باید در قسمت data source در خاصیت schema آن مانند مثالی که در مطلب جاری آمده (در قسمت «تامین داده و نمایش گرید»)، دقیقاً مشخص شوند، تا Kendo UI بداند که اطلاعات مختلف را باید از چه فیلدهایی از JSON خروجی دریافت کند.

نویسنده: وحید محمدطاهری

تاریخ: ۱۴:۲۴ ۱۳۹۳/۱۰/۰۷

با سلام و خدا قوت

آقای نصیری، model ای که باید در قسمت schema تعریف بشه چطوری میشه اونو دینامیک تولید کرد. من یک چنین حالتی رو ایجاد کردم ولی نمی‌دونم چطوری باید اسم ستونو براش مشخص کنم.

```
public class Field
{
    public string Type { get; set; }
}
```

```
public class Fields : System.Collections.ObjectModel.Collection<Field>
{
    public System.Collections.IEnumerable ToList()
    {
        return System.Linq.Enumerable.ToList( System.Linq.Enumerable.Select( this ,
                                                                                   field => new {
field.Type } ) );
    }
}
```

این قسمت اطلاعاتی است که برای ایجاد گرید باز گردانده می‌شود.

```
return new InitializeInfo
{
    ...
    Model = GetModel()
};
```

```
private Model GetModel ()
{
    return new Model{ Fields = GetFields().ToList() };
}
```

متد GetColumns شامل 3 ستون می‌باشد که نوع، عنوان و سایر مشخصات رو توش تعریف کردم

```
private Fields GetFields()
{
    var fields = new Fields();
    foreach ( var column in GetColumns() )
    {
        fields.Add( new Field { Type = column.DataType } );
    }
    return fields;
}
```

الان خروجی که تولید میشه اینجوریه

```
"model": {
  "fields": [
    {
      "type": "string"
```

```

    },
    {
      "type": "string"
    },
    {
      "type": "datetime"
    }
  ]
}

```

ممنون میشم به راهنمایی کنید.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۰/۰۷ ۱۴:۴۱

- پویا هست و خروجی دسترسی هم گرفتید. زمانیکه تعریف می کنید:

```
new Field { Type = column.DataType }
```

یعنی در لیست نهایی، خاصیتی با نام ثابت Type و با مقدار متغیر column.DataType را تولید کن (نام خاصیت، مقدار ثابت نام خاصیت را در JSON نهایی تشکیل می دهد).
+ نیازی هم به این همه پیچیدگی نداشت. تمام کارهایی را که انجام دادید با تهیه خروجی ساده List<Field> از یک متد دلخواه، یکی هست و نیازی به anonymous type کار کردن نبود.
- به همان کلاس فیلد، خواص دیگر مورد نیاز را اضافه کنید (عنوان و سایر مشخصات یک فیلد) و در نهایت لیست ساده List<Field> را بازگشت دهید. هر خاصیت کلاس Field، یک ستون گرید را تشکیل می دهد.
- همچنین دقت داشته باشید اگر از روش مطلب جاری استفاده می کنید، اطلاعات ستون های نهایی باید در فیلد Data نهایی قرار گیرند (قسمت «پیش نیاز تامین داده مخصوص Kendo UI Grid» در بحث).

نویسنده: وحید محمدطاهری
تاریخ: ۱۳۹۳/۱۰/۰۷ ۱۵:۴۸

با تشکر از پاسختون

درسته این به صورت پویا تولید میشه ولی شکل model ای که شما در این مطلب توضیح دادید با این چیزی که کد من تولید می کنه فرق میکنه

برای شما اول نام فیلد هست بعد نوع اون فیلد، در حالی که نحوه تولید داینامیک اینو نمی دونم چطوری باید باشه.

```

model: {
  fields: {
    "Id": { type: "number" }, // تعیین نوع فیلد برای جستجوی پویا مهم است
    "Name": { type: "string" },
    "IsAvailable": { type: "boolean" },
    "Price": { type: "number" }
  }
}

```

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۰/۰۷ ۱۶:۱۹

باید از Dictionary استفاده کنید برای تعریف خواص پویا:

```

public class Field
{
    [JsonExtensionData]
    public Dictionary<string, object> Property { get; set; }
}

```

```
}  
  
public class FieldType  
{  
    public string Type { get; set; }  
}
```

و بعد نحوه استفاده از آن به صورت زیر خواهد بود:

```
var data = new  
{  
    model = new  
    {  
        fields = new List<Field>  
        {  
            new Field  
            {  
                Property = new Dictionary<string, object>  
                {  
                    {"Id", new FieldType { Type = "number" }},  
                    {"Name", new FieldType { Type = "string" }}  
                }  
            }  
        }  
    }  
};  
var dataJson = JsonConvert.SerializeObject(data, Formatting.Indented);
```

با این خروجی:

```
{  
  "model": {  
    "fields": [  
      {  
        "Id": {  
          "Type": "number"  
        },  
        "Name": {  
          "Type": "string"  
        }  
      }  
    ]  
  }  
}
```

- اگر از Web API استفاده می‌کنید، ذکر سطر `JsonConvert.SerializeObject` ضروری نیست و به صورت توکار از [JSON.NET](#) استفاده می‌کند.
- اگر از ASP.NET MVC استفاده می‌کنید، نیاز است [از آن کمک بگیرید](#). از این جهت که خاصیت `JsonExtensionData` سبب می‌شود تا نام ثابت خاصیت `Property`، از خروجی نهایی حذف شود و اعضای دیکشنری، جزئی از خاصیت‌های موجود شوند.
- نکته‌ی «[گرفتن خروجی CamelCase از JSON.NET](#)» را هم باید مد نظر داشته باشید.

نویسنده: ژوپیتز
تاریخ: ۱۳۹۳/۱۱/۱۲ ۱۲:۴۷

در صورتی بخواهیم `dataSource` مربوطه را از همان کنترلر MVC دریافت کنیم، با توجه به اینکه درخواست ارسال شده توسط گرید پارامتریک است، راهکار چیست؟

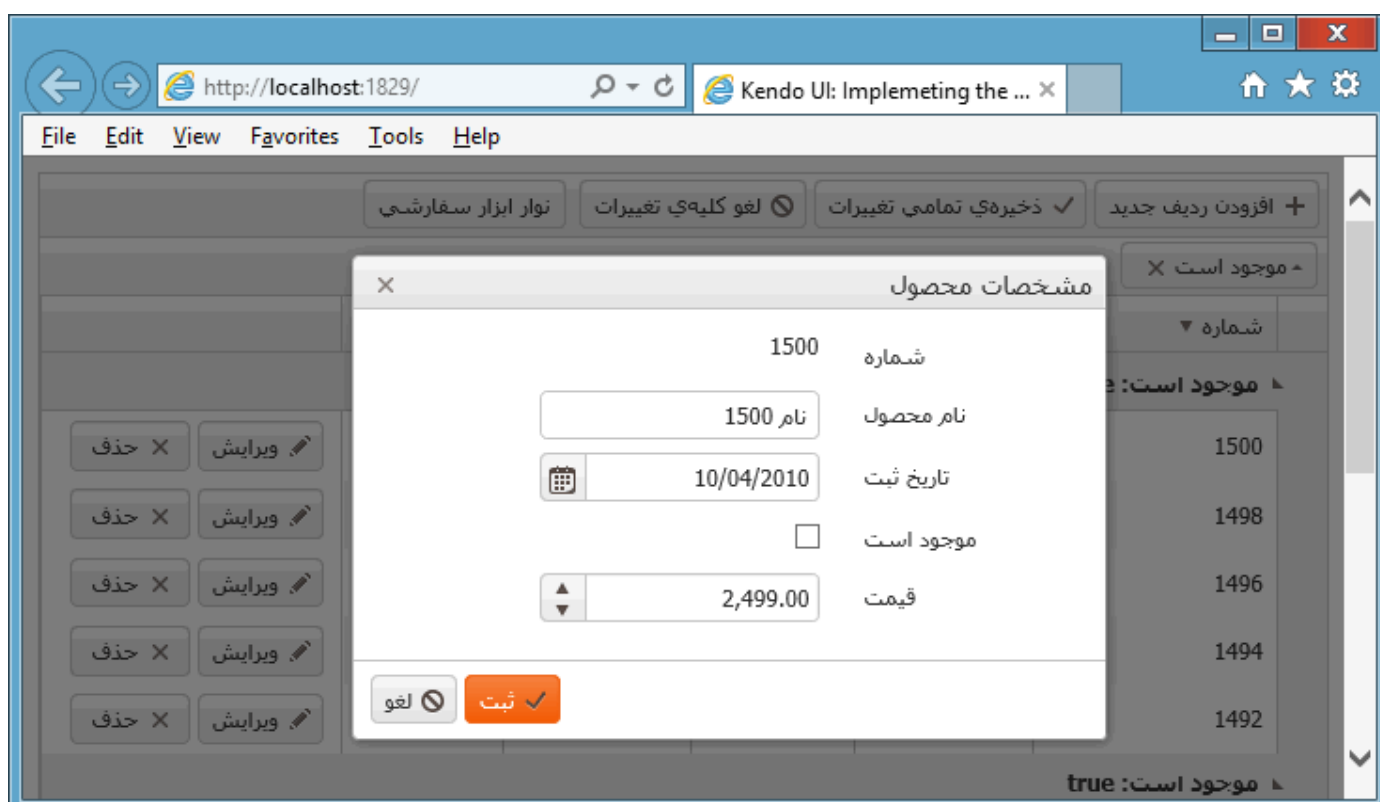
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۱/۱۲ ۱۳:۳۰

دو سری مثال رسمی [kendo-examples-asp-net](#) و [kendo-examples-asp-net-mvc](#) در مورد کار با گرید و یک سری از اجزای مهم آن وجود دارند. سری MVC آن دقیقاً از `Kendo.DynamicLinq` مطرح شده در مطلب جاری، استفاده کرده‌است. برای مثال با [این](#)

پیشنیاز بحث

- « [فرمت کردن اطلاعات نمایش داده شده به کمک Kendo UI Grid](#) »

Kendo UI Grid دارای امکانات ثبت، ویرایش و حذف توکاری است که در ادامه نحوه‌ی فعال سازی آن‌ها را بررسی خواهیم کرد. مثالی که در ادامه بررسی خواهد شد، در تکمیل مطلب « [فرمت کردن اطلاعات نمایش داده شده به کمک Kendo UI Grid](#) » است.



تنظیمات Data Source سمت کاربر

برای فعال سازی [صفحه بندی سمت سرور](#)، با قسمت read منبع داده Kendo UI پیشتر آشنا شده بودیم. جهت فعال سازی قسمت‌های ثبت اطلاعات جدید (create)، به روز رسانی رکوردهای موجود (update) و حذف ردیفی مشخص (destroy) نیاز است تعاریف قسمت‌های متناظر را که هر کدام به آدرس مشخصی در سمت سرور اشاره می‌کنند، اضافه کنیم:

```
var productsDataSource = new kendo.data.DataSource({
    transport: {
        read: {
            url: "api/products",
            dataType: "json",
            contentType: 'application/json; charset=utf-8',
            type: 'GET'
        },
        create: {
            url: "api/products",
            contentType: 'application/json; charset=utf-8',
            type: "POST"
        },
    },
});
```

```

        update: {
            url: function (product) {
                return "api/products/" + product.Id;
            },
            contentType: 'application/json; charset=utf-8',
            type: "PUT"
        },
        destroy: {
            url: function (product) {
                return "api/products/" + product.Id;
            },
            contentType: 'application/json; charset=utf-8',
            type: "DELETE"
        },
        //...
    },
    schema: {
        //...
        model: {
            id: "Id", // define the model of the data source. Required for validation and
            fields: {
                "Id": { type: "number", editable: false }, // تعیین نوع فیلد برای جستجوی پویا/
                "Name": { type: "string", validation: { required: true } },
                "IsAvailable": { type: "boolean" },
                "Price": { type: "number", validation: { required: true, min: 1 } },
                "AddDate": { type: "date", validation: { required: true } }
            }
        }
    },
    batch: false, // enable batch editing - changes will be saved when the user clicks the
    "Save changes" button
    //...
});

```

property types.

مهم است

- همانطور که ملاحظه می‌کنید، حالت‌های update و destroy بر اساس Id ردیف انتخابی کار می‌کنند. این Id را باید در قسمت model مربوط به اسکیمای تعریف شده، دقیقاً مشخص کرد. عدم تعریف فیلد id، سبب خواهد شد تا عملیات update نیز در حالت create تفسیر شود.
- به علاوه در اینجا به ازای هر فیلد، مباحث اعتبارسنجی نیز اضافه شده‌اند؛ برای مثال فیلدهای اجباری با required: true مشخص گردیده‌اند.
- اگر فیلدی نباید ویرایش شود (مانند فیلد Id)، خاصیت editable آن را false کنید.
- در data source امکان تعریف خاصیتی به نام batch نیز وجود دارد. حالت پیش فرض آن false است. به این معنا که در حالت ویرایش، تغییرات هر ردیفی، یک درخواست مجزا را به سمت سرور سبب خواهد شد. اگر آن را true کنید، تغییرات تمام ردیف‌ها در طی یک درخواست به سمت سرور ارسال می‌شوند. در این حالت باید به خاطر داشت که پارامترهای سمت سرور، از حالت یک شیء مشخص باید به لیستی از آن‌ها تغییر یابند.

مدیریت سمت سرور ثبت، ویرایش و حذف اطلاعات

در حالت ثبت، متد Post، توسط آدرس مشخص شده در قسمت create منبع داده‌ها گزیده می‌گردد:

```

namespace KendoUI06.Controllers
{
    public class ProductsController : ApiController
    {
        public HttpResponseMessage Post(Product product)
        {
            if (!ModelState.IsValid)
                return Request.CreateResponse(HttpStatusCode.BadRequest);

            var id = 1;
            var lastItem = ProductDataSource.LatestProducts.LastOrDefault();
            if (lastItem != null)
            {
                id = lastItem.Id + 1;
            }
            product.Id = id;
            ProductDataSource.LatestProducts.Add(product);
        }
    }
}

```

```

        var response = Request.CreateResponse(HttpStatusCode.Created, product);
        response.Headers.Location = new Uri(Url.Link("DefaultApi", new { id = product.Id }));
        // گرید آی دی جدید را به این صورت دریافت می‌کند
        response.Content = new ObjectContent<DataSourceResult>(
            new DataSourceResult { Data = new[] { product } }, new JsonMediaTypeFormatter());
        return response;
    }
}

```

نکته‌ی مهمی که در اینجا باید به آن دقت داشت، نحوه‌ی بازگشت Id رکورد جدید ثبت شده‌است. در این مثال، قسمت schema منبع داده سمت کاربر به نحو ذیل تعریف شده‌است:

```

var productsDataSource = new kendo.data.DataSource({
    //...
    schema: {
        data: "Data",
        total: "Total",
    },
    //...
});

```

از این جهت که خروجی متد Get بازگرداننده‌ی [اطلاعات صفحه بندی شده](#)، از نوع DataSourceResult است و این نوع، دارای خواصی مانند Data، Total و Aggregate است:

```

namespace KendoUI06.Controllers
{
    public class ProductsController : ApiController
    {
        public DataSourceResult Get(HttpRequestMessage requestMessage)
        {
            var request = JsonConvert.DeserializeObject<DataSourceRequest>(
                requestMessage.RequestUri.ParseQueryString().GetKey(0)
            );

            var list = ProductDataSource.LatestProducts;
            return list.AsQueryable()
                .ToDataSourceResult(request.Take, request.Skip, request.Sort, request.Filter);
        }
    }
}

```

بنابراین در متد Post نیز باید بر این اساس، response.Content را از نوع لیستی از DataSourceResult تعریف کرد تا Kendo UI Grid بداند که Id رکورد جدید را باید از فیلد Data، همانند تنظیمات schema منبع داده خود، دریافت کند.

```

response.Content = new ObjectContent<DataSourceResult>(
    new DataSourceResult { Data = new[] { product } }, new
    JsonMediaTypeFormatter());

```

اگر این تنظیم صورت نگیرد، Id رکورد جدید را در گرید، مساوی صفر مشاهده خواهید کرد و عملاً بدون استفاده خواهد شد؛ زیرا قابلیت ویرایش و حذف خود را از دست می‌دهد.

متدهای حذف و به روز رسانی سمت سرور نیز چنین امضایی را خواهند داشت:

```

namespace KendoUI06.Controllers
{
    public class ProductsController : ApiController
    {
        public HttpResponseMessage Delete(int id)
        {
            var item = ProductDataSource.LatestProducts.FirstOrDefault(x => x.Id == id);
            if (item == null)
                return Request.CreateResponse(HttpStatusCode.NotFound);
        }
    }
}

```

```

        ProductDataSource.LatestProducts.Remove(item);
    }
    return Request.CreateResponse(HttpStatusCode.OK, item);
}

[HttpPut] // Add it to fix this error: The requested resource does not support http method
'PUT'
public HttpResponseMessage Update(int id, Product product)
{
    var item = ProductDataSource.LatestProducts
        .Select(
            (prod, index) =>
                new
                {
                    Item = prod,
                    Index = index
                })
        .FirstOrDefault(x => x.Item.Id == id);

    if (item == null)
        return Request.CreateResponse(HttpStatusCode.NotFound);

    if (!ModelState.IsValid || id != product.Id)
        return Request.CreateResponse(HttpStatusCode.BadRequest);

    ProductDataSource.LatestProducts[item.Index] = product;
    return Request.CreateResponse(HttpStatusCode.OK);
}
}
}

```

حالت Update از HTTP Verb خاصی به نام Put استفاده می‌کند و ممکن است در این بین خطای The requested resource does not support http method 'PUT' را دریافت کنید. برای رفع آن ابتدا بررسی کنید که آیا Web.config برنامه دارای تعاریف [ExtensionlessUrlHandler](#) هست یا خیر. همچنین مزین کردن این متد با ویژگی HttpPut، مشکل را برطرف می‌کند.

تنظیمات Kendo UI Grid جهت فعال سازی CRUD

در ادامه کلیه تغییرات مورد نیاز جهت فعال سازی CRUD را در Kendo UI، به همراه مباحث بومی سازی عبارات متناظر با دکمه‌ها و صفحات خودکار مرتبط، مشاهده می‌کنید:

```

$("#report-grid").kendoGrid({
    //....
    editable: {
        confirmation: "آیا مایل به حذف ردیف انتخابی هستید؟",
        destroy: true, // whether or not to delete item when button is clicked
        mode: "popup", // options are "incell", "inline", and "popup"
        //template: kendo.template($("#popupEditorTemplate").html()), // template to use
        for pop-up editing
        update: true, // switch item to edit mode when clicked?
        window: {
            title: "مشخصات محصول" // Localization for Edit in the popup window
        }
    },
    columns: [
        //....
        {
            command: [
                { name: "edit", text: "ویرایش" },
                { name: "destroy", text: "حذف" }
            ],
            title: "&nbsp;", width: "160px"
        }
    ],
    toolbar: [
        { name: "create", text: "افزودن ردیف جدید" },
        { name: "save", text: "ذخیره‌ی تمامی تغییرات" },
        { name: "cancel", text: "لغو کلیه تغییرات" },
        { template: kendo.template($("#toolbarTemplate").html()) }
    ],
    messages: {
        editable: {
            cancelDelete: "لغو",
            confirmation: "آیا مایل به حذف این رکورد هستید؟",

```



```

        confirmDelete: "حذف"
    },
    commands: {
        create: "افزودن ردیف جدید",
        cancel: "لغو کلیه تغییرات",
        save: "ذخیره تمامی تغییرات",
        destroy: "حذف",
        edit: "ویرایش",
        update: "ثبت",
        canceledit: "لغو"
    }
}
});

```

- ساده‌ترین حالت CRUD در Kendo UI با مقدار دهی خاصیت `editable` آن به `true` آغاز می‌شود. در این حالت، ویرایش درون سلولی یا `incell` فعال خواهد شد که مباحث `batching` ابتدای بحث، فقط در این حالت کار می‌کند. زمانی که `incell editing` فعال است، کاربر می‌تواند تمام ردیف‌ها را ویرایش کرده و در آخر کار بر روی دکمه‌ی «ذخیره‌ی تمامی تغییرات» موجود در نوار ابزار، کلیک کند. در سایر حالات، هر بار تنها یک ردیف را می‌توان ویرایش کرد.

- برای فعال سازی تولید صفحات خودکار ویرایش و افزودن ردیف‌ها، نیاز است خاصیت `editable` را به نحوی که ملاحظه می‌کنید، مقدار دهی کرد. خاصیت `mode` آن سه حالت `incell` (پیش فرض)، `inline` و `popup` را پشتیبانی می‌کند.

- اگر حالت‌های `inline` و یا `popup` را فعال کردید، در انتهای ستون‌های تعریف شده، نیاز است ستون ویژه‌ای به نام `command` را مطابق تعاریف فوق، تعریف کنید. در این حالت دو دکمه‌ی ویرایش و ثبت، فعال می‌شوند و اطلاعات خود را از تنظیمات `data source` گرید دریافت می‌کنند. دکمه‌ی ویرایش در حالت `incell` کاربردی ندارد (چون در این حالت کاربر با کلیک درون یک سلول می‌تواند آن را مانند برنامه‌ی اکسل ویرایش کند). اما دکمه‌ی حذف در هر سه حالت قابل استفاده است.

- به نوار ابزار گرید، سه دکمه‌ی افزودن ردیف‌های جدید، ذخیره‌ی تمامی تغییرات و لغو تغییرات صورت گرفته، اضافه شده‌اند. این دکمه‌ها استاندارد بوده و در اینجا نحوه‌ی بومی سازی پیام‌های مرتبط را نیز مشاهده می‌کنید. همانطور که عنوان شد، دکمه‌های «تمامی تغییرات» در حالت فعال سازی `batching` در منبع داده و استفاده از `incell editing` معنا پیدا می‌کند. در سایر حالات این دو دکمه کاربردی ندارند. اما دکمه‌ی افزودن ردیف‌های جدید در هر سه حالت کاربرد دارد و یکسان است.

کدهای کامل این مثال را از اینجا می‌توانید دریافت کنید

[KendoUI06.zip](#)

نظرات خوانندگان

نویسنده: وحید نصیری
تاریخ: ۹:۴۵ ۱۳۹۳/۰۸/۲۱

یک نکته‌ی تکمیلی

در مثال فوق از ASP.NET Web API استفاده شده‌است. اگر علاقمند به استفاده از WCF و یا حتی فایل‌های asmx قدیمی هم باشید، اینکار میسر است. مثال‌هایی را در این زمینه، [در اینجا](#) می‌توانید مشاهده کنید.

نویسنده: شروین ایرانی
تاریخ: ۸:۷ ۱۳۹۳/۱۱/۱۹

در بحث Grid در پلتفرم KendoUI آیا راهی هست که بتونیم بوسیله کوکی براحتی آخرین وضعیت گرید را ذخیره کنیم تا در مراجعه بعدی بصورت اتوماتیک وضعیت قبلی را لود کنه؟ متشکرم

نویسنده: وحید نصیری
تاریخ: ۱۰:۳ ۱۳۹۳/۱۱/۱۹

[Preserve Grid state in a cookie](#)
فایل آن: [PreserveState.zip](#) + [یک مثال](#)

نویسنده: ژوپیتتر
تاریخ: ۱۰:۳۳ ۱۳۹۳/۱۱/۱۹

سلام،

هنگام تغییر خطای زیر را دریافت می‌کنم، هر چند تغییرات ذخیره می‌شوند و فقط این خطا بی‌جهت داده می‌شود. از این روش‌ها ([+](#) و [+](#)) برای دریافت اطلاعات استفاده کردم. به نظر شما مشکل کجاست و یا چطور می‌شه دیباگ کرد؟

نویسنده: وحید نصیری
تاریخ: ۱۱/۱۲/۱۳۹۳

مثال فوق را (KendoUI06) اگر برای ASP.NET MVC بازنویسی کنیم به کدهای ذیل خواهیم رسید:

```
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Mvc;
using Kendo.DynamicLinq;
using KendoUI06Mvc.Models;
using Newtonsoft.Json;

namespace KendoUI06Mvc.Controllers
{
    public class HomeController : Controller
    {
        public ActionResult Index()
        {
            return View(); // shows the page.
        }

        [HttpDelete]
        public ActionResult DeleteProduct(int id)
        {
            var item = ProductDataSource.LatestProducts.FirstOrDefault(x => x.Id == id);
            if (item == null)
                return new HttpNotFoundResult();

            ProductDataSource.LatestProducts.Remove(item);
        }
    }
}
```

```

        return Json(item);
    }

    [HttpGet]
    public ActionResult GetProducts()
    {
        var request = JsonConvert.DeserializeObject<DataSourceRequest>(
            this.Request.Url.ParseQueryString().GetKey(0)
        );

        var list = ProductDataSource.LatestProducts;
        return Json(list.AsQueryable()
            .ToDataSourceResult(request.Take, request.Skip, request.Sort, request.Filter),
            JsonRequestBehavior.AllowGet);
    }

    [HttpPost]
    public ActionResult PostProduct(Product product)
    {
        if (!ModelState.IsValid)
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

        var id = 1;
        var lastItem = ProductDataSource.LatestProducts.LastOrDefault();
        if (lastItem != null)
        {
            id = lastItem.Id + 1;
        }
        product.Id = id;
        ProductDataSource.LatestProducts.Add(product);

        // گرید آی دی جدید را به این صورت دریافت می‌کند
        return Json(new DataSourceResult { Data = new[] { product } });
    }

    [HttpPut] // Add it to fix this error: The requested resource does not support http method
    'PUT'
    public ActionResult UpdateProduct(int id, Product product)
    {
        var item = ProductDataSource.LatestProducts
            .Select(
                (prod, index) =>
                    new
                    {
                        Item = prod,
                        Index = index
                    }
            )
            .FirstOrDefault(x => x.Item.Id == id);

        if (item == null)
            return new HttpNotFoundResult();

        if (!ModelState.IsValid || id != product.Id)
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);

        ProductDataSource.LatestProducts[item.Index] = product;

        //Return HttpStatusCode.OK
        return new HttpStatusCodeResult(HttpStatusCode.OK);
    }
}

```

در این حالت View برنامه فقط جهت ذکر آدرس‌های جدید باید اصلاح شود و نیاز به تغییر دیگری ندارد:

```

var productsDataSource = new kendo.data.DataSource({
    transport: {
        read: {
            url: "@Url.Action("GetProducts","Home")",
            dataType: "json",
            contentType: 'application/json; charset=utf-8',
            type: 'GET'
        },
        create: {
            url: "@Url.Action("PostProduct","Home")",
            contentType: 'application/json; charset=utf-8',
            type: "POST"
        }
    }
});

```

```

    },
    update: {
      url: function (product) {
        return "@Url.Action('UpdateProduct','Home')/" + product.Id;
      },
      contentType: 'application/json; charset=utf-8',
      type: "PUT"
    },
    destroy: {
      url: function (product) {
        return "@Url.Action('DeleteProduct','Home')/" + product.Id;
      },
      contentType: 'application/json; charset=utf-8',
      type: "DELETE"
    },
    parameterMap: function (options) {
      return kendo.stringify(options);
    }
  },
},

```

نویسنده: ژوپیتتر

تاریخ: ۱۴:۴۱ ۱۳۹۳/۱۱/۲۰

برای کسانی که از روش [GitHub](#) لینک داده شده استفاده کردند و مشکل بنده رو هنگام Update اطلاعات دارند: در `ActionResult` مربوط به Update گریدویو `Kendu UI` هنگام بازگشت مقدار `Json` به صورت `null` باید از عبارت رشته‌ای خالی شبیه زیر استفاده کنیم:

```

[HttpPost]
public ActionResult Update(IEnumerable<Product> products)
{
    // ....
    //Return empty result
    return Json("");
}

```

موفق باشید.

پیشنیازها

- « [استفاده از Kendo UI templates](#) »

- « [اعتبار سنجی ورودی‌های کاربر در Kendo UI](#) »

- « [فعال سازی عملیات CRUD در Kendo UI Grid](#) » جهت آشنایی با نحوه‌ی تعریف DataSource ایی که می‌تواند اطلاعات را ثبت، حذف و یا ویرایش کند.

در این مطلب قصد داریم به یک چنین صفحه‌ای برسیم که در آن در ابتدای نمایش، لیست ثبت نام‌های موجود، از سرور دریافت و توسط یک Kendo UI template نمایش داده می‌شود. سپس امکان ویرایش و حذف هر ردیف، وجود خواهد داشت، به همراه امکان افزودن ردیف‌های جدید. در این بین مدیریت نمایش لیست ثبت نام‌ها توسط امکانات binding توکار فریم ورک MVVM مخصوص Kendo UI صورت خواهد گرفت. همچنین کلیه اعمال مرتبط با هر ردیف نیز توسط data binding دو طرفه مدیریت خواهد شد.

The screenshot shows a web application running on localhost:1829. The main view displays a table of user registrations with columns for id, name, course, cost, email, and phone. Each row has 'Add' and 'Delete' buttons. To the right of the table is a form for adding a new registration, with fields for name, course, cost, email, and phone, and a checkbox for 'Accept terms'. Below the table, there is a summary of the total cost: 'Total: 4,000 Rials'. At the bottom, a network panel shows the API calls: POST /api/registrations (201), PUT /api/registrations/2 (200), PUT /api/registrations/4 (200), and DELETE /api/registrations/4 (200). The network panel also shows the data received and sent in bytes and milliseconds.

id	نام	دوره	هزینه	ایمیل	تلفن
1	userx	c++	ریال 1,000	tst1@site.com	12345678
2	usery	css3	ریال 2,000	tst2@site.com	12345678
3	userz	java	ریال 1,000	tst3@site.com	12345678

جمع کل: 4,000 ریال

ثبت نام

نام

دوره

مبلغ پرداختی

پست الکترونیک

تلفن

☒ شرایط دوره را قبول دارم.

ارسال از نو

Network

URL	Protocol	Method	Result	Type	Received	Taken	Initiator	Timings
/api/registrations	HTTP	POST	201	application/json	496 B	218 ms	XMLHttpRequest	
/api/registrations/2	HTTP	PUT	200		331 B	93 ms	XMLHttpRequest	
/api/registrations/4	HTTP	PUT	200		331 B	47 ms	XMLHttpRequest	
/api/registrations/4	HTTP	DELETE	200	application/json	485 B	31 ms	XMLHttpRequest	

Items: 4 Sent: 1.96 KB (2,004 bytes) Received: 1.60 KB (1,643 bytes)

الگوی MVVM یا Model-View-ViewModel که برای اولین بار جهت کاربردهای WPF و Silverlight معرفی شد، برای ساده سازی اتصال تغییرات کنترل‌های برنامه به خواص ViewModel یک View کاربرد دارد. برای مثال با تغییر عنصر انتخابی یک DropDownList در یک View، بلافاصله خاصیت متصل به آن که در ViewModel برنامه تعریف شده است، مقدار دهی و به روز خواهد شد. هدف نهایی آن نیز جدا سازی منطق کدهای UI، از کدهای جاوا اسکریپتی سمت کاربر است. برای این منظور کتابخانه‌هایی مانند [Knockout.js](#) به صورت اختصاصی برای این کار تهیه شده‌اند؛ اما Kendo UI نیز جهت یکپارچگی هرچه تمامتر اجزای آن، دارای یک فریم ورک MVVM توکار نیز می‌باشد. طراحی آن نیز بسیار شبیه به Knockout.js است؛ اما با سازگاری 100 درصد با کل مجموعه. پیاده سازی الگوی MVVM از 4 قسمت تشکیل می‌شود:

- Model که بیانگر خواص متناظر با اشیاء رابط کاربری است.
- View همان رابط کاربری است که به کاربر نمایش داده می‌شود.
- ViewModel واسطی است بین Model و View. کار آن انتقال داده‌ها و رویدادها از View به مدل است و در حالت binding دوطرفه، عکس آن نیز صحیح می‌باشد.
- Declarative data binding جهت رهایی برنامه نویسی‌ها از نوشتن کدهای هماهنگ سازی اطلاعات المان‌های View و خواص ViewModel کاربرد دارد.

در ادامه این اجزا را با پیاده سازی مثالی که در ابتدای بحث مطرح شد، دنبال می‌کنیم.

تعریف Model و ViewModel

در سمت سرور، مدل ثبت نام برنامه چنین شکلی را دارد:

```
namespace KendoUI07.Models
{
    public class Registration
    {
        public int Id { set; get; }
        public string UserName { set; get; }
        public string CourseName { set; get; }
        public int Credit { set; get; }
        public string Email { set; get; }
        public string Tel { set; get; }
    }
}
```

در سمت کاربر، این مدل را به نحو ذیل می‌توان تعریف کرد:

```
<script type="text/javascript">
    $(function () {
        var model = kendo.data.Model.define({
            id: "Id",
            fields: {
                Id: { type: 'number' }, // leave this set to 0 or undefined, so Kendo knows it is
                UserName: { type: 'string' },
                CourseName: { type: 'string' },
                Credit: { type: 'number' },
                Email: { type: 'string' },
                Tel: { type: 'string' }
            }
        });
    });
</script>
```

و ViewModel برنامه در ساده‌ترین شکل آن اکنون چنین تعریفی را خواهد یافت:

```
<script type="text/javascript">
    $(function () {
        var viewModel = kendo.observable({
```

```

        accepted: false,
        course: new model()
    });
});
</script>

```

یک `viewModel` در Kendo UI به صورت یک `observable object` تعریف می‌شود که می‌تواند دارای تعدادی خاصیت و متد دلخواه باشد. هر خاصیت آن به یک عنصر HTML متصل خواهد شد. در اینجا این اتصال دو طرفه است؛ به این معنا که تغییرات UI به خواص `viewModel` و برعکس منتقل و منعکس می‌شوند.

اتصال ViewModel به View برنامه

تعریف فرم ثبت نام را در اینجا ملاحظه می‌کنید. فیلدهای مختلف آن بر اساس نکات اعتبارسنجی HTML 5 با ویژگی‌های خاص آن، مزین شده‌اند. جزئیات آن را در مطلب « [اعتبارسنجی ورودی‌های کاربر در Kendo UI](#) » بیشتر بررسی کرده‌ایم. اگر به تعریف هر فیلد دقت کنید، ویژگی `data-bind` جدیدی را هم ملاحظه خواهید کرد:

```

<div id="coursesSection" class="k-rtl k-header">
    <div class="box-col">
        <form id="myForm" data-role="validator" novalidate="novalidate">
            <h3>ثبت نام</h3>
            <ul>
                <li>
                    <label for="Id">Id</label>
                    <span id="Id" data-bind="text:course.Id"></span>
                </li>
                <li>
                    <label for="UserName">نام</label>
                    <input type="text" id="UserName" name="UserName" class="k-textbox"
                        data-bind="value:course.UserName"
                        required />
                </li>
                <li>
                    <label for="CourseName">دوره</label>
                    <input type="text" dir="ltr" id="CourseName" name="CourseName" required
                        data-bind="value:course.CourseName" />
                    <span class="k-invalid-msg" data-for="CourseName"></span>
                </li>
                <li>
                    <label for="Credit">مبلغ پرداختی</label>
                    <input id="Credit" name="Credit" type="number" min="1000" max="6000"
                        required data-max-msg="6000 و 1000 عددی بین" dir="ltr"
                        data-bind="value:course.Credit"
                        class="k-textbox k-input" />
                    <span class="k-invalid-msg" data-for="Credit"></span>
                </li>
                <li>
                    <label for="Email">پست الکترونیک</label>
                    <input type="email" id="Email" dir="ltr" name="Email"
                        data-bind="value:course.Email"
                        required class="k-textbox" />
                </li>
                <li>
                    <label for="Tel">تلفن</label>
                    <input type="tel" id="Tel" name="Tel" dir="ltr" pattern="\d{8}"
                        required class="k-textbox"
                        data-bind="value:course.Tel"
                        data-pattern-msg="8 رقم" />
                </li>
                <li>
                    <input type="checkbox" name="Accept"
                        data-bind="checked:accepted"
                        required />
                    شرایط دوره را قبول دارم.
                    <span class="k-invalid-msg" data-for="Accept"></span>
                </li>
                <li>
                    <button class="k-button"
                        data-bind="enabled: accepted, click: doSave"
                        type="submit">
                        ارسال
                    </button>
                    <button class="k-button" data-bind="click: resetModel">از نو</button>
                </li>
            </ul>
        </form>
    </div>
</div>

```



```

        </li>
      </ul>
      <span id="doneMsg"></span>
    </form>
  </div>

```

برای اتصال ViewModel تعریف شده به ناحیه‌ی مشخص شده با DIV ایی با Id مساوی coursesSection، می‌توان از متد kendo.bind استفاده کرد.

```

<script type="text/javascript">
$(function () {
    var model = kendo.data.Model.define({
        // ...
    });

    var viewModel = kendo.observable({
        // ...
    });

    kendo.bind($("#coursesSection"), viewModel);
});
</script>

```

به این ترتیب Kendo UI به بر اساس تعریف data-bind یک فیلد، برای مثال تغییرات خواص course.UserName را به text box نام کاربر منتقل می‌کند و همچنین اگر کاربر اطلاعاتی را در این text box وارد کند، بلافاصله این تغییرات در خاصیت course.UserName منعکس خواهند شد.

```

<input type="text" id="UserName" name="UserName" class="k-textbox"
    data-bind="value:course.UserName"
    required />

```

بنابراین تا اینجا به صورت خلاصه، مدلی را توسط متد kendo.data.Model.define، معادل مدل سمت سرور خود ایجاد کردیم. سپس وهله‌ای از این مدل را به صورت یک خاصیت جدید دلخواهی در ViewModel تعریف شده توسط متد kendo.observable در معرض دید View برنامه قرار دادیم. در ادامه اتصال View و ViewModel، با فراخوانی متد kendo.bind انجام شد. اکنون برای دریافت تغییرات کنترل‌های برنامه، تنها کافی است ویژگی‌های data-bind ایی را به آن‌ها اضافه کنیم. در ناحیه‌ی تعریف شده توسط متد kendo.bind، کلیه خواص ViewModel در دسترس هستند. برای مثال اگر به تعریف ViewModel دقت کنید، یک خاصیت دیگر به نام accepted با مقدار false نیز در آن تعریف شده‌است (این خاصیت چون صرفاً کاربرد UI داشت، در model برنامه قرار نگرفت). از آن برای اتصال checkbox تعریف شده، به button ارسال اطلاعات، استفاده کرده‌ایم:

```

<input type="checkbox" name="Accept"
    data-bind="checked:accepted"
    required />

<button class="k-button"
    data-bind="enabled: accepted, click: doSave"
    type="submit">
    ارسال
</button>

```

برای مثال اگر کاربر این checkbox را انتخاب کند، مقدار خاصیت accepted، مساوی true خواهد شد. تغییر مقدار این خاصیت، توسط ViewModel بلافاصله در کل ناحیه coursesSection منتشر می‌شود. به همین جهت ویژگی enabled: accepted که به معنای مقید بودن فعال یا غیرفعال بودن دکمه بر اساس مقدار خاصیت accepted است، دکمه را فعال می‌کند، یا برعکس و برای انجام این عملیات نیازی نیست کدنویسی خاصی را انجام داد. در اینجا بین checkbox و button یک سیم کشی برقرار است.

ارسال داده‌های تغییر کرده‌ی ViewModel به سرور

تا اینجا 4 جزء اصلی الگوی MVVM که در ابتدای بحث عنوان شد، تکمیل شده‌اند. مدل اطلاعات فرم تعریف گردید. ViewModel ایی

که این خواص را به المان‌های فرم متصل می‌کند نیز در ادامه اضافه شده‌است. توسط ویژگی‌های data-bind Declarative کار data binding انجام می‌شود. در ادامه نیاز است تغییرات ViewModel را به سرور، جهت ثبت، به روز رسانی و حذف نهایی منتقل کرد.

```
<script type="text/javascript">
    $(function () {
        var model = kendo.data.Model.define({
            //...
        });

        var dataSource = new kendo.data.DataSource({
            type: 'json',
            transport: {
                read: {
                    url: "api/registrations",
                    dataType: "json",
                    contentType: 'application/json; charset=utf-8',
                    type: 'GET'
                },
                create: {
                    url: "api/registrations",
                    contentType: 'application/json; charset=utf-8',
                    type: "POST"
                },
                update: {
                    url: function (course) {
                        return "api/registrations/" + course.Id;
                    },
                    contentType: 'application/json; charset=utf-8',
                    type: "PUT"
                },
                destroy: {
                    url: function (course) {
                        return "api/registrations/" + course.Id;
                    },
                    contentType: 'application/json; charset=utf-8',
                    type: "DELETE"
                },
                parameterMap: function (data, type) {
                    // Convert to a JSON string. Without this step your content will be form
                    return JSON.stringify(data);
                }
            },
            schema: {
                model: model
            },
            error: function (e) {
                alert(e.errorThrown);
            },
            change: function (e) {
                // فراخوانی در زمان دریافت اطلاعات از سرور و یا تغییرات محلی
                viewModel.set("coursesDataSourceRows", new
                kendo.data.ObservableArray(this.view()));
            }
        });

        var viewModel = kendo.observable({
            //...
        });

        kendo.bind($("#coursesSection"), viewModel);
        dataSource.read(); // دریافت لیست موجود از سرور در آغاز کار
    });
</script>
```

در اینجا تعریف DataSource کار با منبع داده راه دور ASP.NET Web API را مشاهده می‌کنید. تعاریف اصلی آن با تعاریف مطرح شده در مطلب « [فعال سازی عملیات CRUD در Kendo UI Grid](#) » یکی هستند. هر قسمت آن مانند read, create, update و destroy به یکی از متدهای کنترلر ASP.NET Web API اشاره می‌کنند. حالت‌های update و destroy بر اساس Id ردیف انتخابی کار می‌کنند. این Id را باید در قسمت model مربوط به اسکیمای تعریف شده، دقیقاً مشخص کرد. عدم تعریف فیلد id، سبب خواهد شد تا عملیات update نیز در حالت create تفسیر شود.

متصل کردن DataSource به ViewModel

تا اینجا DataSource ایی جهت کار با سرور تعریف شده است؛ اما مشخص نیست که اگر رکوردی اضافه شد، چگونه باید اطلاعات خودش را به روز کند. برای این منظور خواهیم داشت:

```
<script type="text/javascript">
    $(function () {
        $("#coursesSection").kendoValidator({
            // ...
        });

        var model = kendo.data.Model.define({
            // ...
        });

        var dataSource = new kendo.data.DataSource({
            // ...
        });

        var viewModel = kendo.observable({
            accepted: false,
            course: new model(),
            doSave: function (e) {
                e.preventDefault();
                console.log("this", this.course);
                var validator = $("#coursesSection").data("kendoValidator");
                if (validator.validate()) {
                    if (this.course.Id == 0) {
                        dataSource.add(this.course);
                    }
                    dataSource.sync(); // push to the server
                    this.set("course", new model()); // reset controls
                }
            },
            resetModel: function (e) {
                e.preventDefault();
                this.set("course", new model());
            }
        });

        kendo.bind($("#coursesSection"), viewModel);
        dataSource.read(); // دریافت لیست موجود از سرور در آغاز کار
    });
</script>
```

همانطور که در تعاریف تکمیلی viewModel مشاهده می‌کنید، اینبار دو متد جدید دلخواه doSave و resetModel را اضافه کرده‌ایم. در متد doSave، ابتدا بررسی می‌کنیم آیا اعتبارسنجی فرم با موفقیت انجام شده است یا خیر. اگر بله، توسط متد add منبع داده، اطلاعات فرم جاری را توسط شیء course که هم اکنون به تمامی فیلدهای آن متصل است، اضافه می‌کنیم. در اینجا بررسی شده است که آیا Id این اطلاعات صفر است یا خیر. از آنجائیکه از همین متد برای به روز رسانی نیز در ادامه استفاده خواهد شد، در حالت به روز رسانی، Id شیء ثبت شده، از طرف سرور دریافت می‌گردد. بنابراین غیر صفر بودن این Id به معنای عملیات به روز رسانی است و در این حالت نیازی نیست کار بیشتری را انجام داد؛ زیرا شیء متناظر با آن پیشتر به منبع داده اضافه شده است.

استفاده از متد add صرفاً به معنای مطلع کردن منبع داده محلی از وجود رکوردی جدید است. برای ارسال این تغییرات به سرور، از متد sync آن می‌توان استفاده کرد. متد sync بر اساس متد add یک درخواست POST، بر اساس شیء ایی که Id غیر صفر دارد، یک درخواست PUT و با فراخوانی متد remove بر روی منبع داده، یک درخواست DELETE را به سمت سرور ارسال می‌کند. متد دلخواه resetModel سبب مقدار دهی مجدد شیء course با یک وهله‌ی جدید از شیء model می‌شود. همینقدر برای پاک کردن تمامی کنترل‌های صفحه کافی است.

تا اینجا دو متد جدید را در ViewModel برنامه تعریف کرده‌ایم. در مورد نحوه‌ی اتصال آن‌ها به View، به کدهای دو دکمه‌ی موجود در فرم دقت کنید:

```
<button class="k-button"
    data-bind="enabled: accepted, click: doSave"
    type="submit">
```

```
ارسال
</button>
<button class="k-button" data-bind="click: resetModel">از نو</button>
```

این متدها نیز توسط ویژگی‌های data-bind به هر دکمه نسبت داده شده‌اند. به این ترتیب برای مثال با کلیک کاربر بر روی دکمه‌ی submit، متد doSave موجود در ViewModel فراخوانی می‌شود.

مدیریت سمت سرور ثبت، ویرایش و حذف اطلاعات

در حالت ثبت، متد Post توسط آدرس مشخص شده در قسمت create منبع داده، فراخوانی می‌گردد. نکته‌ی مهمی که در اینجا باید به آن دقت داشت، نحوه‌ی بازگشت Id رکورد جدید ثبت شده‌است. اگر این تنظیم صورت نگیرد، Id رکورد جدید را در لیست، مساوی صفر مشاهده خواهید کرد و منبع داده این رکورد را همواره به عنوان یک رکورد جدید، مجدداً به سرور ارسال می‌کند.

```
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;
using KendoUI07.Models;

namespace KendoUI07.Controllers
{
    public class RegistrationsController : ApiController
    {
        public HttpResponseMessage Delete(int id)
        {
            var item = RegistrationsDataSource.LatestRegistrations.FirstOrDefault(x => x.Id == id);
            if (item == null)
                return Request.CreateResponse(HttpStatusCode.NotFound);

            RegistrationsDataSource.LatestRegistrations.Remove(item);
            return Request.CreateResponse(HttpStatusCode.OK, item);
        }

        public IEnumerable<Registration> Get()
        {
            return RegistrationsDataSource.LatestRegistrations;
        }

        public HttpResponseMessage Post(Registration registration)
        {
            if (!ModelState.IsValid)
                return Request.CreateResponse(HttpStatusCode.BadRequest);

            var id = 1;
            var lastItem = RegistrationsDataSource.LatestRegistrations.LastOrDefault();
            if (lastItem != null)
            {
                id = lastItem.Id + 1;
            }
            registration.Id = id;
            RegistrationsDataSource.LatestRegistrations.Add(registration);

            // ارسال آی دی مهم است تا از ارسال رکوردهای تکراری جلوگیری شود
            return Request.CreateResponse(HttpStatusCode.Created, registration);
        }

        [HttpPut] // Add it to fix this error: The requested resource does not support http method
        'PUT'
        public HttpResponseMessage Update(int id, Registration registration)
        {
            var item = RegistrationsDataSource.LatestRegistrations
                .Select(
                    (prod, index) =>
                        new
                        {
                            Item = prod,
                            Index = index
                        })
                .FirstOrDefault(x => x.Item.Id == id);

            if (item == null)
```

```

        return Request.CreateResponse(HttpStatusCode.NotFound);

        if (!ModelState.IsValid || id != registration.Id)
            return Request.CreateResponse(HttpStatusCode.BadRequest);

        RegistrationsDataSource.LatestRegistrations[item.Index] = registration;
        return Request.CreateResponse(HttpStatusCode.OK);
    }
}

```

در اینجا بیشتر امضای این متدها مهم هستند، تا منطق پیاده سازی شده در آنها. همچنین بازگشت Id رکورد جدید، توسط متد Post نیز بسیار مهم است و سبب می‌شود تا DataSource بداند با فراخوانی متد sync آن، باید عملیات Post یا create انجام شود یا Put و update.

نمایش آنی اطلاعات ثبت شده در یک لیست

ردیف‌های اضافه شده به منبع داده را می‌توان بلافاصله در همان سمت کلاینت توسط Kendo UI Template که قابلیت کار با ViewModelها را دارد، نمایش داد:

```

<div id="coursesSection" class="k-rtl k-header">
    <div class="box-col">
        <form id="myForm" data-role="validator" novalidate="novalidate">
            <!-- فرم بحث شده در ابتدای مطلب -->
        </form>
    </div>
    <div id="results">
        <table class="metrotable">
            <thead>
                <tr>
                    <th>Id</th>
                    <th>نام</th>
                    <th>دوره</th>
                    <th>هزینه</th>
                    <th>ایمیل</th>
                    <th>تلفن</th>
                </tr>
            </thead>
            <tbody data-template="row-template" data-bind="source: coursesDataSourceRows"></tbody>
            <tfoot data-template="footer-template" data-bind="source: this"></tfoot>
        </table>
        <script id="row-template" type="text/x-kendo-template">
            <tr>
                <td data-bind="text: Id"></td>
                <td data-bind="text: UserName"></td>
                <td dir="ltr" data-bind="text: CourseName"></td>
                <td>
                    #: kendo.toString(get("Credit"), "c0") #
                </td>
                <td data-bind="text: Email"></td>
                <td data-bind="text: Tel"></td>
                <td><button class="k-button" data-bind="click: deleteCourse">حذف</button></td>
                <td><button class="k-button" data-bind="click: editCourse">ویرایش</button></td>
            </tr>
        </script>
        <script id="footer-template" type="text/x-kendo-template">
            <tr>
                <td colspan="3"></td>
                <td>
                    جمع کل #: kendo.toString(totalPrice(), "c0") #
                </td>
                <td colspan="2"></td>
                <td></td>
                <td></td>
            </tr>
        </script>
    </div>
</div>

```

در ناحیه‌ی coursesSection که توسط متد kendo.bind به viewModel برنامه متصل شده‌است، یک جدول را برای نمایش ردیف‌های ثبت شده توسط کاربر اضافه کرده‌ایم. thead آن بیانگر سر ستون جدول است. قسمت tbody وtfoot این جدول توسط دو Kendo UI Template مقدار دهی شده‌اند. هر کدام نیز منبع داده‌اشان را از view model دریافت می‌کنند. در row-template معادل خواص شیء course را مشاهده می‌کنید. در footer-template متد totalPrice برای نمایش جمع ستون هزینه اضافه شده‌است. بنابراین مطابق این قسمت از View، به یک خاصیت جدید coursesDataSourceRows و سه متد deleteCourse، editCourse و totalPrice نیاز است:

```
<script type="text/javascript">
    $(function () {
        // ...
        var viewModel = kendo.observable({
            accepted: false,
            course: new model(),
            coursesDataSourceRows: new kendo.data.ObservableArray([]),
            doSave: function (e) {
                // ...
            },
            resetModel: function (e) {
                // ...
            },
            totalPrice: function () {
                var sum = 0;
                $.each(this.get("coursesDataSourceRows"), function (index, item) {
                    sum += item.Credit;
                });
                return sum;
            },
            deleteCourse: function (e) {
                // the current data item is passed as the "data" field of the event argument
                var course = e.data;
                dataSource.remove(course);
                dataSource.sync(); // push to the server
            },
            editCourse: function (e) {
                // the current data item is passed as the "data" field of the event argument
                var course = e.data;
                this.set("course", course);
            }
        });

        kendo.bind($("#coursesSection"), viewModel);
        dataSource.read(); // دریافت لیست موجود از سرور در آغاز کار
    });
</script>
```

نحوه‌ی اتصال خاصیت جدید coursesDataSourceRows که به عنوان منبع داده ردیف‌های row-template عمل می‌کند، به این صورت است:

- ابتدا خاصیت دلخواه coursesDataSourceRows به viewModel اضافه می‌شود تا در ناحیه‌ی coursesSection در دسترس قرار گیرد.

- سپس اگر به انتهای تعریف DataSource دقت کنید، داریم:

```
<script type="text/javascript">
    $(function () {
        var dataSource = new kendo.data.DataSource({
            //...
            change: function (e) {
                // فراخوانی در زمان دریافت اطلاعات از سرور و یا تغییرات محلی
                viewModel.set("coursesDataSourceRows", new
                kendo.data.ObservableArray(this.view()));
            }
        });
    });
</script>
```

متد change آن، هر زمانیکه اطلاعاتی در منبع داده تغییر کنند یا اطلاعاتی به سمت سرور ارسال یا دریافت گردد، فراخوانی می‌شود. در همینجا فرصت خواهیم داشت تا خاصیت coursesDataSourceRows را جهت نمایش اطلاعات موجود در منبع داده،

مقدار دهی کنیم. همین مقدار دهی ساده سبب اجرای row-template برای تولید ردیف‌های جدول می‌شود. استفاده از new kendo.data.ObservableArray سبب خواهد شد تا اگر اطلاعاتی در فرم برنامه تغییر کند، این اطلاعات بلافاصله در لیست گزارش برنامه نیز منعکس گردد.

کدهای کامل این مثال را از اینجا می‌توانید دریافت کنید:

[KendoUI07.zip](#)

[در قسمت قبل](#) با مقدمات برپایی یک برنامه‌ی تک صفحه‌ای وب مبتنی بر Ember.js آشنا شدیم. مثال انتهای بحث آن نیز یک لیست ساده را نمایش می‌دهد. در ادامه همین برنامه را جهت نمایش لیستی از اشیاء JSON دریافتی از سرور تغییر خواهیم داد. همچنین یک صفحه‌ی about را نیز به آن اضافه خواهیم کرد.

پیشنیازهای سمت سرور

- ابتدا یک پروژه‌ی خالی ASP.NET را ایجاد کنید. نوع آن مهم نیست که Web Forms باشد یا MVC.
- سپس قصد داریم مدل کاربران سیستم را توسط یک [ASP.NET Web API Controller](#) در اختیار Ember.js قرار دهیم. مباحث پایه‌ای Web API نیز در وب فرم‌ها و MVC یکی است.
- مدل سمت سرور برنامه چنین شکلی را دارد:

```
namespace EmberJS02.Models
{
    public class User
    {
        public int Id { set; get; }
        public string UserName { set; get; }
        public string Email { set; get; }
    }
}
```

کنترلر Web API ای که این اطلاعات را در اختیار کلاینت‌ها قرار می‌دهد، به نحو ذیل تعریف می‌شود:

```
using System.Collections.Generic;
using System.Web.Http;
using EmberJS02.Models;

namespace EmberJS02.Controllers
{
    public class UsersController : ApiController
    {
        // GET api/<controller>
        public IEnumerable<User> Get()
        {
            return UsersDataSource.UsersList;
        }
    }
}
```

در اینجا UsersDataSource.UsersList صرفاً یک لیست جنریک ساده از کلاس User است و کدهای کامل آن را می‌توانید از فایل پیوست انتهای بحث دریافت کنید.

همچنین فرض بر این است که مسیریابی سمت سرور ذیل را نیز به فایل global.asax.cs جهت فعال سازی دسترسی به متدهای کنترلر UsersController تعریف کرده‌اید:

```
using System;
using System.Web.Http;
using System.Web.Routing;

namespace EmberJS02
{
    public class Global : System.Web.HttpApplication
    {
        protected void Application_Start(object sender, EventArgs e)
        {
            RouteTable.Routes.MapHttpRoute(
                name: "DefaultApi",
```



```
routeTemplate: "api/{controller}/{id}",
defaults: new { id = RouteParameter.Optional }
);
}
}
```

پیشنیازهای سمت کاربر

پیشنیازهای سمت کاربر این قسمت با قسمت «[تهیه‌ی اولین برنامه‌ی Ember.js](#)» دقیقاً یکی است. ابتدا فایل‌های مورد نیاز Ember.js به برنامه اضافه شده‌اند:

```
PM> Install-Package EmberJS
```

سپس یک فایل app.js با محتوای ذیل به پوشه‌ی Scripts اضافه شده‌است:

```
App = Ember.Application.create();
App.IndexRoute = Ember.Route.extend({
  setupController: function(controller) {
    controller.set('content', ['red', 'yellow', 'blue']);
  }
});
```

و در آخر یک فایل index.html با محتوای ذیل کار برپایی اولیه‌ی یک برنامه‌ی مبتنی بر Ember.js را انجام می‌دهد:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title></title>
  <script src="Scripts/jquery-2.1.1.js" type="text/javascript"></script>
  <script src="Scripts/handlebars.js" type="text/javascript"></script>
  <script src="Scripts/ember.js" type="text/javascript"></script>
  <script src="Scripts/app.js" type="text/javascript"></script>
</head>
<body>
  <script type="text/x-handlebars" data-template-name="application">
    <h1>Header</h1>
    {{outlet}}
  </script>
  <script type="text/x-handlebars" data-template-name="index">
    Hello,
    <strong>Welcome to Ember.js</strong>!
    <ul>
      {{#each item in content}}
      <li>
        {{item}}
      </li>
      {{/each}}
    </ul>
  </script>
</body>
</html>
```

تا اینجا را [در قسمت قبل](#) مطالعه کرده بودید.

در ادامه قصد داریم به هدر صفحه، دو لینک Home و About را اضافه کنیم؛ به نحوی که لینک Home به مسیریابی index و لینک About به مسیریابی about که صفحه‌ی جدید «درباره‌ی برنامه» را نمایش می‌دهد، اشاره کنند.

تعریف صفحه‌ی جدید About

برنامه‌های Ember.js، برنامه‌های تک صفحه‌ای وب هستند و صفحات جدید در آن‌ها به صورت یک template جدید تعریف می‌شوند که نهایتاً متناظر با یک مسیریابی مشخص خواهند بود.

به همین جهت ابتدا در فایل app.js مسیریابی about را اضافه خواهیم کرد:

```
App.Router.map(function() {  
  this.resource('about');  
});
```

به این ترتیب با فراخوانی آدرس /about در مرورگر توسط کاربر، منابع مرتبط با این آدرس و قالب مخصوص آن، توسط Ember.js پردازش خواهند شد.

بنابراین به صفحه‌ی index.html برنامه مراجعه کرده و صفحه‌ی about را توسط یک قالب جدید تعریف می‌کنیم:

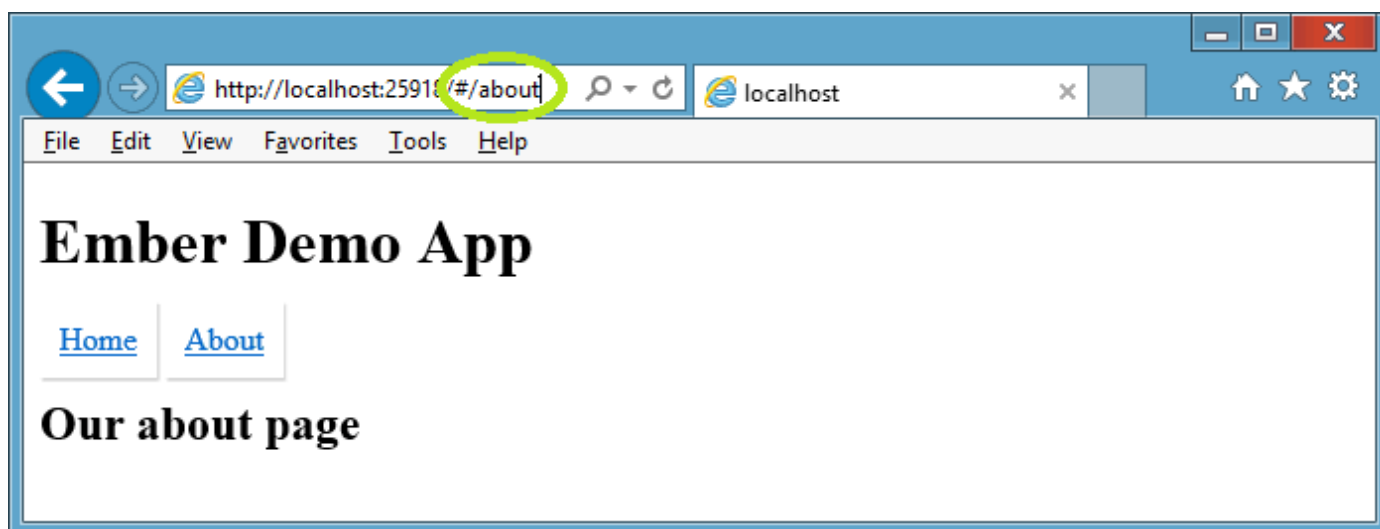
```
<script type="text/x-handlebars" data-template-name="about">  
  <h2>Our about page</h2>  
</script>
```

تنها نکته‌ی مهم در اینجا مقدار data-template-name است که سبب خواهد شد تا به مسیریابی about، به صورت خودکار متصل و مرتبط شود.

در این حالت اگر برنامه را در حالت معمولی اجرا کنید، خروجی خاصی را مشاهده نخواهید کرد. بنابراین نیاز است تا لینکی را جهت اشاره به این مسیر جدید به صفحه اضافه کنیم:

```
<script type="text/x-handlebars" data-template-name="application">  
  <h1>Ember Demo App</h1>  
  <ul class="nav">  
    <li>{{#linkTo 'index'}}Home{{/linkTo}}</li>  
    <li>{{#linkTo 'about'}}About{{/linkTo}}</li>  
  </ul>  
  {{outlet}}  
</script>
```

اگر [از قسمت قبل](#) به خاطر داشته باشید، عنوان شد که قالب ویژه‌ی application به صورت خودکار با وهله سازی Ember.Application.create به صفحه اضافه می‌شود. اگر نیاز به سفارشی سازی آن وجود داشت، خصوصا جهت تعریف عناصری که باید در تمام صفحات حضور داشته باشند (مانند منوها)، می‌توان آن‌را به نحو فوق سفارشی سازی کرد. در اینجا با استفاده از امکان یا directive ویژه‌ای به نام linkTo، لینک‌هایی به مسیریابی‌های index و about اضافه شده‌اند. به این ترتیب اگر کاربری برای مثال بر روی لینک About کلیک کند، کتابخانه‌ی Ember.js او را به صورت خودکار به مسیریابی about سپس نمایش قالب مرتبط با آن (قالب about ایمی که پیشتر تعریف کردیم) هدایت خواهد کرد؛ مانند تصویر ذیل:



همانطور که در آدرس صفحه نیز مشخص است، هرچند صفحه‌ی about نمایش داده شده‌است، اما هنوز نیز در همان صفحه‌ی اصلی برنامه قرار داریم. به علاوه در این قسمت جدید، همچنان منوی بالای صفحه نمایان است؛ از این جهت که تعاریف آن به قالب application اضافه شده‌اند.

دریافت و نمایش اطلاعات از سرور

اکنون که با نحوه‌ی تعریف یک صفحه‌ی جدید و برپایی سیم‌کشی‌های مرتبط با آن آشنا شدیم، می‌خواهیم صفحه‌ی دیگری را به نام Users به برنامه اضافه کنیم و در آن لیست کاربران ارائه شده توسط کنترلر Web API سمت سرور ابتدای بحث را نمایش دهیم. بنابراین ابتدا مسیریابی جدید users را به صفحه اضافه می‌کنیم تا لیست کاربران، در آدرس /users قابل دسترسی شود:

```
App.Router.map(function() {
  this.resource('about');
  this.resource('users');
});
```

سپس نیاز است مدلی را توسط فراخوانی Ember.Object.extend ایجاد کرده و به کمک متد reopenClass آن را توسعه دهیم:

```
App.UsersLink = Ember.Object.extend({});
App.UsersLink.reopenClass({
  findAll: function () {
    var users = [];
    $.getJSON('/api/users').then(function(response) {
      response.forEach(function(item) {
        users.pushObject(App.UsersLink.create(item));
      });
    });
    return users;
  }
});
```

در اینجا متد دلخواهی را به نام findAll اضافه کرده‌ایم که توسط متد getJSON جی‌کوئری، به مسیر /api/users سمت سرور متصل شده و لیست کاربران را از سرور به صورت JSON دریافت می‌کند. در اینجا خروجی دریافتی از سرور به کمک متد pushObject به آرایه کاربران اضافه خواهد شد. همچنین نحوه‌ی فراخوانی متد create مدل UsersLink را نیز در اینجا مشاهده می‌کنید (App.UsersLink.create).

پس از اینکه نحوه‌ی دریافت اطلاعات از سرور مشخص شد، باید اطلاعات این مدل را در اختیار مسیریابی Users قرار داد:

```
App.UsersRoute = Ember.Route.extend({
  model: function() {
    return App.UsersLink.findAll();
  }
});

App.UsersController = Ember.ObjectController.extend({
  customHeader : 'Our Users List'
});
```

به این ترتیب زمانیکه کاربر به مسیر /users مراجعه می‌کند، سیستم مسیریابی می‌داند که اطلاعات مدل خود را باید از کجا تهیه نماید.

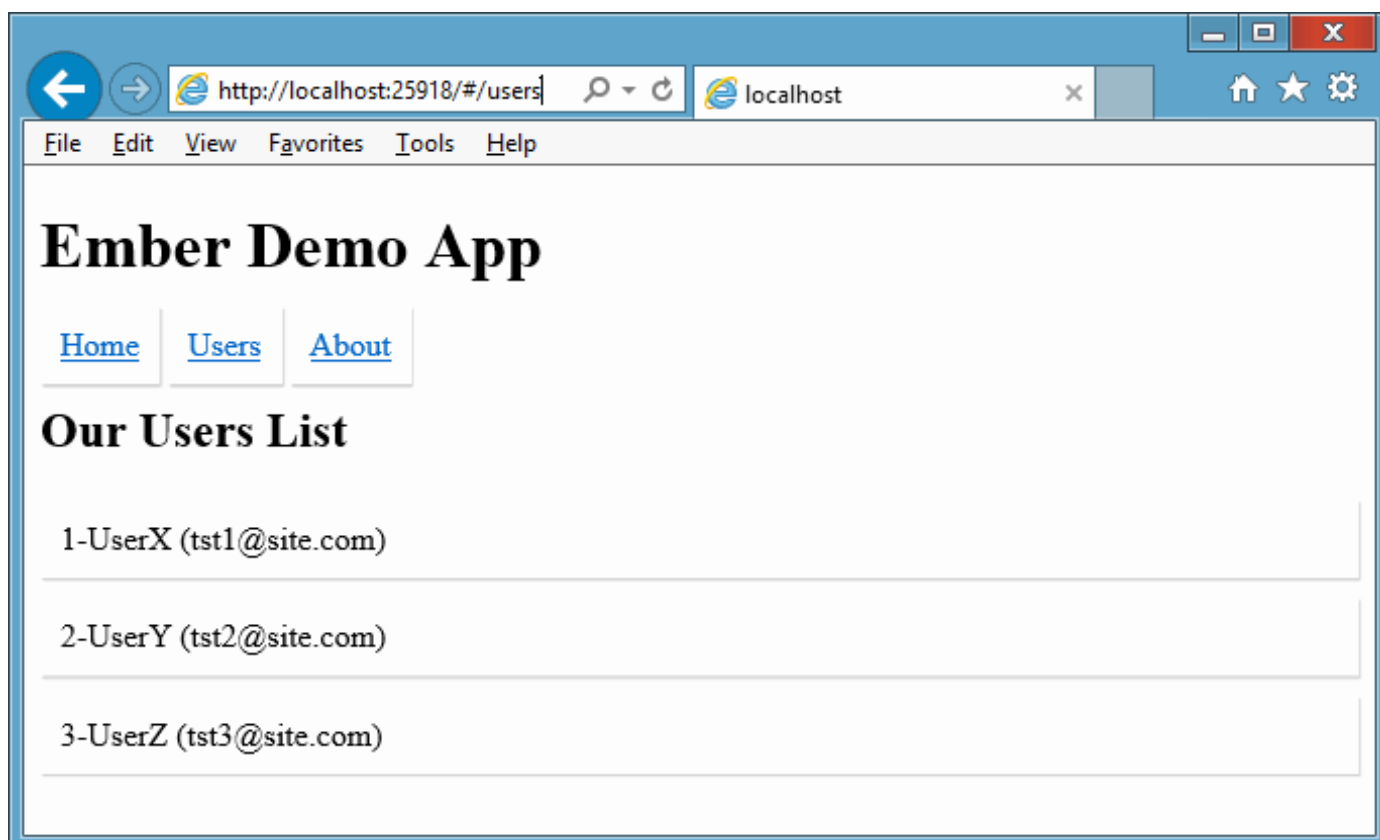
همچنین در کنترلری که تعریف شده، صرفاً یک خاصیت سفارشی و دلخواه جدید، به نام customHeader برای نمایش در ابتدای صفحه تعریف و مقدار دهی گردیده‌است.

اکنون قالبی که قرار است اطلاعات مدل را نمایش دهد، چنین شکلی را خواهد داشت:

```
<script type="text/x-handlebars" data-template-name="users">
  <h2>{{customHeader}}</h2>
  <ul>
    {{#each item in model}}
      <li>
        {{item.Id}}-{{item.UserName}} ({{item.Email}})
      </li>
    {{/each}}
  </ul>
</script>
```

```
</li>
  {{/each}}
</ul>
</script>
```

با تنظیم `data-template-name` به `users` سبب خواهیم شد تا این قالب اطلاعات خودش را از مسیریابی `users` دریافت کند. سپس یک حلقه نوشته‌ایم تا کلیه عناصر موجود در مدل را خوانده و در صفحه نمایش دهد. همچنین در عنوان قالب نیز از خاصیت سفارشی `customHeader` استفاده شده‌است:



کدهای کامل این قسمت را از اینجا می‌توانید دریافت کنید:

[EmberJS02.zip](#)

مقدمات ساخت بلاگ مبتنی بر ember.js در [قسمت قبل](#) به پایان رسید. در این قسمت صرفاً قصد داریم بجای استفاده از HTML 5 local storage از یک REST web service مانند یک ASP.NET Web API Controller یا یک ASP.NET MVC Controller استفاده کنیم و اطلاعات نهایی را به سرور ارسال و یا از آن دریافت کنیم.

تنظیم Ember data برای کار با سرور

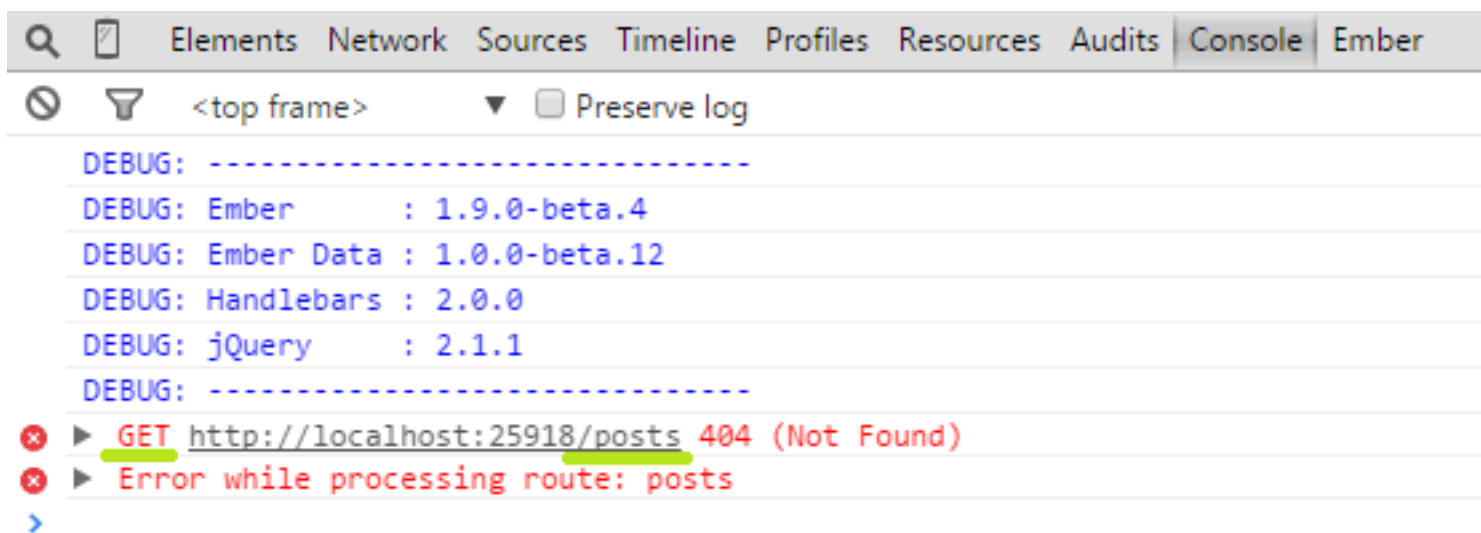
Ember data به صورت پیش فرض و در پشت صحنه با استفاده از Ajax برای کار با یک REST Web Service طراحی شده‌است و کلیه تبادلات آن نیز با فرمت JSON انجام می‌شود. بنابراین تمام کدهای سمت کاربر [قسمت قبل](#) نیز در این حالت کار خواهند کرد. تنها کاری که باید انجام شود، حذف تنظیمات ابتدایی آن برای کار با HTML 5 local storage است. برای این منظور ابتدا فایل index.html را گشوده و سپس مدخل localStorage_adapter.js را از آن حذف کنید:

```
<!--<script src="Scripts/Libs/localstorage_adapter.js" type="text/javascript"></script>-->
```

همچنین دیگر نیازی به store.js نیز نمی‌باشد:

```
<!--<script src="Scripts/App/store.js" type="text/javascript"></script>-->
```

اکنون برنامه را اجرا کنید، چنین پیام خطایی را مشاهده خواهید کرد:



همانطور که عنوان شد، ember data به صورت پیش فرض با سرور کار می‌کند و در اینجا به صورت خودکار، یک درخواست Get را به آدرس <http://localhost:25918/posts> جهت دریافت آخرین مطالب ثبت شده، ارسال کرده‌است و چون هنوز وب سرویسی در برنامه تعریف نشده، با خطای 404 و یا یافت نشد، مواجه شده‌است. این درخواست نیز بر اساس تعاریف موجود در فایل Scripts\Routes\posts.js، به سرور ارسال شده‌است:

```
Blogger.PostsRoute = Ember.Route.extend({
  model: function () {
    return this.store.find('post');
  }
});
```

```
});
```

Ember data شبیه به یک ORM عمل می‌کند. تنظیمات ابتدایی آن را تغییر دهید، بدون نیازی به تغییر در کدهای اصلی برنامه، می‌تواند با یک منبع داده جدید کار کند.

تغییر تنظیمات پیش فرض آغازین Ember data

آدرس درخواستی `http://localhost:25918/posts` به این معنا است که کلیه درخواست‌ها، به همان آدرس و پورت ریشه‌ی اصلی سایت ارسال می‌شوند. اما اگر یک ASP.NET Web API Controller را تعریف کنیم، نیاز است این درخواست‌ها، برای مثال به آدرس `api/posts` ارسال شوند؛ بجای `posts/`. برای این منظور پوشه‌ی جدید `Scripts\Adapters` را ایجاد کرده و فایل `web_api_adapter.js` را با این محتوا به آن اضافه کنید:

```
DS.RESTAdapter.reopen({
  namespace: 'api'
});
```

سپس تعریف مدخل آن را نیز به فایل `index.html` اضافه نمائید:

```
<script src="Scripts/Adapters/web_api_adapter.js" type="text/javascript"></script>
```

[تعریف فضای نام](#) در اینجا سبب خواهد شد تا درخواست‌های جدید به آدرس `api/posts` ارسال شوند.

تغییر تنظیمات پیش فرض ASP.NET Web API

در سمت سرور، بنابر اصول نامگذاری خواص، نام‌ها با حروف بزرگ شروع می‌شوند:

```
namespace EmberJS03.Models
{
    public class Post
    {
        public int Id { set; get; }
        public string Title { set; get; }
        public string Body { set; get; }
    }
}
```

اما در سمت کاربر و کدهای اسکریپتی، عکس آن صادق است. به همین جهت نیاز است که [CamelCasePropertyNameContractResolver](#) را در JSON.NET تنظیم کرد تا به صورت خودکار اطلاعات ارسالی به کلاینت‌ها را به صورت camel case تولید کند:

```
using System;
using System.Web.Http;
using System.Web.Routing;
using Newtonsoft.Json.Serialization;

namespace EmberJS03
{
    public class Global : System.Web.HttpApplication
    {
        protected void Application_Start(object sender, EventArgs e)
        {
            RouteTable.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}",
                defaults: new { id = RouteParameter.Optional }
            );

            var settings =
                GlobalConfiguration.Configuration.Formatters.JsonFormatter.SerializerSettings;
            settings.ContractResolver = new CamelCasePropertyNameContractResolver();
        }
    }
}
```

```
}  
}  
}
```

نحوه‌ی صحیح بازگشت اطلاعات از یک ASP.NET Web API جهت استفاده در Ember data

با تنظیمات فوق، اگر کنترلر جدیدی را به صورت ذیل جهت بازگشت لیست مطالب تهیه کنیم:

```
namespace EmberJS03.Controllers  
{  
    public class PostsController : ApiController  
    {  
        public IEnumerable<Post> Get()  
        {  
            return DataSource.PostsList;  
        }  
    }  
}
```

با یک چنین خطایی در سمت کاربر مواجه خواهیم شد:

```
WARNING: Encountered "0" in payload, but no model was found for model name "0" (resolved model name  
using DS.RESTSerializer.typeForRoot("0"))
```

این خطا از آنجا ناشی می‌شود که Ember data، اطلاعات دریافتی از سرور را بر اساس قرارداد [JSON API](#) دریافت می‌کند. برای حل این مشکل راه‌حل‌های زیادی مطرح شده‌اند که تعدادی از آن‌ها را در لینک‌های زیر می‌توانید مطالعه کنید:

<http://jsonapi.codeplex.com>

<https://github.com/xqiu/MVCSPAWithEmberjs>

<https://github.com/rmichela/EmberDataAdapter>

<https://github.com/MilkyWayJoe/Ember-WebAPI-Adapter>

<http://blog.yodersolutions.com/using-ember-data-with-asp-net-web-api>

<http://emadibrahim.com/2014/04/09/emberjs-and-asp-net-web-api-and-json-serialization>

و خلاصه‌ی آن‌ها به این صورت است:

خروجی JSON تولیدی توسط ASP.NET Web API چنین شکلی را دارد:

```
[  
  {  
    Id: 1,  
    Title: 'First Post'  
  }, {  
    Id: 2,  
    Title: 'Second Post'  
  }  
]
```

اما Ember data نیاز به یک چنین خروجی دارد:

```
{  
  posts: [{  
    id: 1,  
    title: 'First Post'  
  }, {  
    id: 2,  
    title: 'Second Post'  
  }]  
}
```

به عبارتی آرایه‌ی مطالب را از ریشه‌ی posts باید دریافت کند (مطابق فرمت [JSON API](#)). برای انجام اینکار یا از لینک‌های معرفی شده استفاده کنید و یا راه حل ساده‌ی ذیل هم پاسخگو است:

```
using System.Web.Http;
using EmberJS03.Models;

namespace EmberJS03.Controllers
{
    public class PostsController : ApiController
    {
        public object Get()
        {
            return new { posts = DataSource.PostsList };
        }
    }
}
```

در اینجا ریشه‌ی posts را توسط یک anonymous object ایجاد کرده‌ایم. اکنون اگر برنامه را اجرا کنید، در صفحه‌ی اول آن، لیست عناوین مطالب را مشاهده خواهید کرد.

تاثیر قرارداد JSON API در حین ارسال اطلاعات به سرور توسط Ember data

در تکمیل کنترلرهای Web API مورد نیاز (کنترلرهای مطالب و نظرات)، نیاز به متدهای Post، Update و Delete هم خواهد بود. دقیقاً فرامین ارسالی توسط Ember data توسط همین HTTP Verbs به سمت سرور ارسال می‌شوند. در این حالت اگر متد Post کنترلر نظرات را به این شکل طراحی کنیم:

```
public HttpResponseMessage Post(Comment comment)
```

کار نخواهد کرد؛ چون مطابق فرمت [JSON API](#) ارسالی توسط Ember data، یک چنین شیء JSON ایی را دریافت خواهیم کرد:

```
{"comment":{"text":"data...", "post":"3"}}
```

بنابراین Ember data چه در حین دریافت اطلاعات از سرور و چه در زمان ارسال اطلاعات به آن، اشیاء جاوا اسکریپتی را در یک ریشه‌ی هم نام آن شیء قرار می‌دهد.

برای پردازش آن، یا باید از راه حل‌های ثالث مطرح شده در ابتدای بحث استفاده کنید و یا می‌توان مطابق کدهای ذیل، کل اطلاعات JSON ارسالی را توسط کتابخانه‌ی JSON.NET نیز پردازش کرد:

```
namespace EmberJS03.Controllers
{
    public class CommentsController : ApiController
    {
        public HttpResponseMessage Post(HttpRequestMessage requestMessage)
        {
            var jsonContent = requestMessage.Content.ReadAsStringAsync().Result;
            // {"comment":{"text":"data...", "post":"3"}}
            var jsonObj = JObject.Parse(jsonContent);
            var comment = jsonObj.SelectToken("comment", false).ToObject<Comment>();

            var id = 1;
            var lastItem = DataSource.CommentsList.LastOrDefault();
            if (lastItem != null)
            {
                id = lastItem.Id + 1;
            }
            comment.Id = id;
            DataSource.CommentsList.Add(comment);

            // ارسال آی دی با فرمت خاص مهم است
            return Request.CreateResponse(HttpStatusCode.Created, new { comment = comment });
        }
    }
}
```



```
}
```

در اینجا توسط requestMessage به محتوای ارسال شده‌ی به سرور که همان شیء JSON ارسالی است، دسترسی خواهیم داشت. سپس متد JObject.Parse، آنرا به صورت عمومی تبدیل به یک شیء JSON می‌کند و نهایتاً با استفاده از متد SelectToken آن می‌توان ریشه‌ی comment و یا در کنترلر مطالب، ریشه‌ی post را انتخاب و سپس تبدیل به شیء Comment و یا Post کرد. همچنین فرمت return نهایی هم مهم است. در این حالت خروجی ارسالی به سمت کاربر، باید مجدداً با فرمت JSON API باشد؛ یعنی باید comment اصلاح شده را به همراه ریشه‌ی comment ارسال کرد. در اینجا نیز anonymous object تهیه شده، چنین کاری را انجام می‌دهد.

Ember data در Lazy loading

تا اینجا اگر برنامه را اجرا کنید، لیست مطالب صفحه‌ی اول را مشاهده خواهید کرد، اما لیست نظرات آن‌ها را خیر؛ از این جهت که ضرورتی نداشت تا در بار اول ارسال لیست مطالب به سمت کاربر، تمام نظرات متناظر با آن‌ها را هم ارسال کرد. بهتر است زمانیکه کاربر یک مطلب خاص را مشاهده می‌کند، نظرات خاص آنرا به سمت کاربر ارسال کنیم. در تعاریف سمت کاربر Ember data، پارامتر دوم رابطه‌ی hasMany که با async:true مشخص شده‌است، دقیقاً معنای lazy loading را دارد.

```
Blogger.Post = DS.Model.extend({
  title: DS.attr(),
  body: DS.attr(),
  comments: DS.hasMany('comment', { async: true } /* lazy loading */)
});
```

در سمت سرور، دو راه برای فعال سازی این lazy loading تعریف شده در سمت کاربر وجود دارد:
الف) Idهای نظرات هر مطلب را به صورت یک آرایه، در بار اول ارسال لیست نظرات به سمت کاربر، تهیه و ارسال کنیم:

```
namespace EmberJS03.Models
{
  public class Post
  {
    public int Id { set; get; }
    public string Title { set; get; }
    public string Body { set; get; }

    // lazy loading via an array of IDs
    public int[] Comments { set; get; }
  }
}
```

در اینجا خاصیت Comments، تنها کافی است لیستی از Idهای نظرات مرتبط با مطلب جاری باشد. در این حالت در سمت کاربر اگر مطلب خاصی جهت مشاهده‌ی جزئیات آن انتخاب شود، به ازای هر Id ذکر شده، یکبار دستور Get صادر خواهد شد. ب) این روش به علت تعداد رفت و برگشت بیش از حد به سرور، کارایی آنچنانی ندارد. بهتر است جهت مشاهده‌ی جزئیات یک مطلب، تنها یکبار درخواست Get کلیه نظرات آن صادر شود. برای اینکار باید مدل برنامه را به شکل زیر تغییر دهیم:

```
namespace EmberJS03.Models
{
  public class Post
  {
    public int Id { set; get; }
    public string Title { set; get; }
    public string Body { set; get; }

    // load related models via URLs instead of an array of IDs
    // ref. https://github.com/emberjs/data/pull/1371
    public object Links { set; get; }

    public Post()
    {

```

```

        Links = new { comments = "comments" }; // api/posts/id/comments
    }
}

```

در اینجا یک خاصیت جدید به نام Links ارائه شده است. نام Links در Ember data [استاندارد است](#) و از آن برای دریافت کلیه اطلاعات لینک شده‌ی به یک مطلب استفاده می‌شود. با تعریف این خاصیت به نحوی که ملاحظه می‌کنید، اینبار Ember data تنها یکبار درخواست ویژه‌ای را با فرمت api/posts/id/comments، به سمت سرور ارسال می‌کند. برای مدیریت آن، قالب مسیریابی پیش فرض api/{controller}/{id}/{name} را می‌توان به صورت api/{controller}/{id}/{name} اصلاح کرد:

```

namespace EmberJS03
{
    public class Global : System.Web.HttpApplication
    {
        protected void Application_Start(object sender, EventArgs e)
        {
            RouteTable.Routes.MapHttpRoute(
                name: "DefaultApi",
                routeTemplate: "api/{controller}/{id}/{name}",
                defaults: new { id = RouteParameter.Optional, name = RouteParameter.Optional }
            );

            var settings =
                GlobalConfiguration.Configuration.Formatters.JsonFormatter.SerializerSettings;
            settings.ContractResolver = new CamelCasePropertyNamesContractResolver();
        }
    }
}

```

اکنون دیگر درخواست جدید api/posts/3/comments با پیام 404 یا یافت نشد مواجه نمی‌شود. در این حالت در طی یک درخواست می‌توان کلیه نظرات را به سمت کاربر ارسال کرد. در اینجا نیز ذکر ریشه‌ی comments همانند ریشه posts، الزامی است:

```

namespace EmberJS03.Controllers
{
    public class PostsController : ApiController
    {
        //GET api/posts/id
        public object Get(int id)
        {
            return
                new
                {
                    posts = DataSource.PostsList.FirstOrDefault(post => post.Id == id),
                    comments = DataSource.CommentsList.Where(comment => comment.Post == id).ToList()
                };
        }
    }
}

```

پردازش‌های async و متد transitionToRoute در Ember.js

اگر متد حذف مطالب را نیز به کنترلر Posts اضافه کنیم:

```

namespace EmberJS03.Controllers
{
    public class PostsController : ApiController
    {
        public HttpResponseMessage Delete(int id)
        {
            var item = DataSource.PostsList.FirstOrDefault(x => x.Id == id);
            if (item == null)
                return Request.CreateResponse(HttpStatusCode.NotFound);
        }
    }
}

```

```

        DataSource.PostsList.Remove(item);

        // حذف کامنت‌های مرتبط
        var relatedComments = DataSource.CommentsList.Where(comment => comment.Post ==
id).ToList();
        relatedComments.ForEach(comment => DataSource.CommentsList.Remove(comment));

        return Request.CreateResponse(HttpStatusCode.OK, new { post = item });
    }
}

```

قسمت سمت سرور کار تکمیل شده‌است. اما در سمت کاربر، چنین خطایی را دریافت خواهیم کرد:

Attempted to handle event `pushedData` on while in state root.deleted.inFlight.

منظور از حالت `inFlight` در اینجا این است که هنوز کار حذف سمت سرور تمام نشده‌است که متد `transitionToRoute` را صادر کرده‌اید. برای اصلاح آن، فایل `Scripts\Controllers\post.js` را باز کرده و پس از متد `destroyRecord`، متد `then` را قرار دهید:

```

Blogger.PostController = Ember.ObjectController.extend({
  isEditing: false,
  actions: {
    edit: function () {
      this.set('isEditing', true);
    },
    save: function () {
      var post = this.get('model');
      post.save();

      this.set('isEditing', false);
    },
    delete: function () {
      if (confirm('Do you want to delete this post?')) {
        var thisController = this;
        var post = this.get('model');
        post.destroyRecord().then(function () {
          thisController.transitionToRoute('posts');
        });
      }
    }
  }
});

```

به این ترتیب پس از پایان عملیات حذف سمت سرور، [قسمت then اجرا خواهد شد](#). همچنین باید دقت داشت که `this` اشاره کننده به کنترلر جاری را باید پیش از فراخوانی `then` ذخیره و استفاده کرد.

کدهای کامل این قسمت را از اینجا می‌توانید دریافت کنید:

[EmberJS03_05.zip](#)

طی [این پست](#) با تزریق وابستگی‌ها در Asp.net MVC آشنا شدید. روش ذکر شده در آن برای کنترلرهای Web Api جوابگو نیست و باید از روش‌های دیگری برای این منظور استفاده نماییم.

نکته 1: برای پیاده سازی این مثال‌ها، Castle Windsor به عنوان IOC Container انتخاب شده است. بدیهی است می‌توانید از Ioc Container مورد نظر خود نیز بهره ببرید.

نکته 2 : می‌توانید از مقاله [\[هاست سرویس‌های Web Api با استفاده از OWIN و TopShelf\]](#) جهت هاست سرویس‌های web Api خود استفاده نمایید.

روش اول

اگر قبلاً در این زمینه جستجو کرده باشید، به احتمال زیاد با مفهوم IDependencyResolver بیگانه نیستید. درباره استفاده از این روش مقالات متعددی نوشته شده است؛ حتی در مثال‌های موجود در خود سایت MSDN نیز این روش را مرسوم دانسته و آن را به اشتراک می‌گذارند. جهت نمونه می‌توانید این [پروژه](#) را دانلود کرده و کدهای آن را بررسی کنید. در این روش، قدم اول، ساخت یک کلاس و پیاده سازی اینترفیس IDependencyResolver می‌باشد؛ به صورت زیر:

```
public class ApiDependencyResolver : IDependencyResolver
{
    public ApiDependencyResolver(IWindsorContainer container)
    {
        Container = container;
    }

    public IWindsorContainer Container
    {
        get;
        private set;
    }

    public object GetService(Type serviceType)
    {
        try
        {
            return Container.Kernel.HasComponent(serviceType) ? Container.Resolve(serviceType) :
null;
        }
        catch (Kernel.ComponentNotFoundException)
        {
            return null;
        }
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        try
        {
            return Container.ResolveAll(serviceType).Cast<object>();
        }
        catch (Kernel.ComponentNotFoundException)
        {
            return Enumerable.Empty<object>();
        }
    }

    public IDependencyScope BeginScope()
    {
        return new SharedDependencyResolver(Container);
    }

    public void Dispose()
    {
        Container.Dispose();
    }
}
```

```
}
```

اینترفیس `IDependencyResolver` از اینترفیس دیگری به نام `IDependencyScope` ارث می‌برد که دارای دو متد اصلی به نام‌های `GetService` و `GetServices` است که جهت وهله سازی کنترلرها استفاده می‌شوند. با فراخوانی این متدها، نمونه‌ی ساخته شده توسط `Container` بازگشت داده خواهد شد.

کاربرد متد `BeginScope` چیست ؟

کنترلرها به صورت (Per Request) بر اساس هر درخواست وهله سازی خواهند شد. جهت مدیریت چرخه‌ی عمر کنترلرها و منابع در اختیار آن‌ها، از متد `BeginScope` استفاده می‌شود. به این صورت که نمونه‌ی اصلی `DependencyResolver` در هنگام شروع برنامه به `GlobalConfiguration` پروژه `Attach` خواهد شد. سپس به ازای هر درخواست، جهت وهله سازی `Controller`ها، متد `GetService` از محدوده داخلی (منظور فراخوانی متد `BeginScope` است) باعث ایجاد نمونه و بعد از اتمام فرآیند، متد `Dispose` باعث آزاد سازی منابع موجود خواهد شد. پیاده سازی متد `BeginScope` وابسته به `IocContainer` مورد استفاده شما است. در این جا کلاس `SharedDependencyResolver` را به صورت زیر پیاده سازی کردم:

```
public class SharedDependencyResolver : IDependencyScope
{
    public SharedDependencyResolver(IWindsorContainer container)
    {
        Container = container;
        Scope = Container.BeginScope();
    }

    public IWindsorContainer Container
    {
        get;
        private set;
    }

    public IDisposable Scope
    {
        get;
        private set;
    }

    public object GetService(Type serviceType)
    {
        try
        {
            return Container.Kernel.HasComponent(serviceType) ? Container.Resolve(serviceType) :
null;
        }
        catch (ComponentNotFoundException)
        {
            return null;
        }
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        try
        {
            return Container.ResolveAll(serviceType).Cast<object>();
        }
        catch (ComponentNotFoundException)
        {
            return null;
        }
    }

    public void Dispose()
    {
        Scope.Dispose();
    }
}
```

اگر از UnityContainer استفاده می‌کنید کافیسست تکه کد زیر را جایگزین کلاس بالا نمایید:

```
public IDependencyScope BeginScope()
{
    var child = container.CreateChildContainer();
    return new ApiDependencyResolver(child);
}
```

برای جستجوی خودکار کنترلرها و رجیستر کردن آنها به برنامه Windsor امکانات جالبی را در اختیار ما قرار می‌دهد. ابتدا یک Installer ایجاد می‌کنیم:

```
public class KernelInstaller : IWindsorInstaller
{
    public void Install(IWindsorContainer container, IConfigurationStore store)
    {
        container.Register(Classes.FromThisAssembly().BasedOn<ApiController>().LifestyleTransient());
        container.Kernel.Resolver.AddSubResolver(new CollectionResolver(container.Kernel, true));
    }
}
```

در پایان در کلاس Startup نیز کافیسست مراحل زیر را انجام دهید:
 «ابتدا Installer نوشته شده را به WindsorContainer معرفی نمایید.
 «DependencyResolver نوشته شده را به HttpConfiguration معرفی کنید.
 «عملیات Routing مورد نظر را ایجاد و سپس config مورد نظر را در اختیار appBuilder قرار دهید.

```
public class Startup
{
    public void Configuration( IAppBuilder appBuilder )
    {
        var container = new WindsorContainer();
        container.Install(new KernelInstaller());

        var config = new HttpConfiguration
        {
            DependencyResolver = new ApiDependencyResolver(container)
        };

        config.MapHttpAttributeRoutes();

        config.Routes.MapHttpRoute(
            name: "Default",
            routeTemplate: "{controller}/{action}/{name}",
            defaults: new { name = RouteParameter.Optional }
        );

        config.EnsureInitialized();

        appBuilder.UseWebApi( config );
    }
}
```

نکته: این روش به دلیل استفاده از الگوی ServiceLocator و همچنین نداشتن Context درخواست ها روشی منسوخ شده می‌باشد که طی [این مقاله](#) جناب نصیری به صورت کامل به این مبحث پرداخته اند.

نظرات خوانندگان

نویسنده: هادی احمدی
تاریخ: ۱۳۹۳/۱۱/۱۵ ۱۰:۴۱

سلام
خسته نباشید، سپاس از مطلب مفیدتون

نقد هایی بر استفاده از DependencyResolver وارد هست که یکی از آنها نداشتن Context هنگام Resolve کردن وابستگی هاست. (برای اطلاعات بیشتر به [این پست](#) از آقای Mark Seeman نویسنده کتاب Dependency Injection in .NET مراجعه کنید) به همین دلیل (و دلایل دیگر مثل انعطاف پذیری کم و ...) استفاده از CotrollerActivator نسبت به این روش پیشنهاد می شود که مطمئنا شما در سری های بعدی این مقاله به آن خواهید پرداخت. بنده هم در [این مقاله](#) در مورد استفاده از CotrollerActivator برای پیاده سازی DI نوشته ام. موفق باشید

بعد از معرفی نسخه‌ی 2 از Asp.Net Web Api و پشتیبانی رسمی آن از OData بسیاری از توسعه دهندگان سیستم نفس راحتی کشیدند؛ زیرا از آن پس می‌توانستند علاوه بر امکانات جالب و مهمی که تحت پروتکل OData میسر بود، از سایر امکانات تعبیه شده در نسخه‌ی دوم web Api نیز استفاده نمایند. یکی از این قابلیت‌ها، مبحث مهم [Batching Processing](#) است که در طی این پست با آن آشنا خواهیم شد.

منظور از Batch Request این است که درخواست دهنده بتواند چندین درخواست (Multiple Http Request) را به صورت یک Pack جامع، در قالب فقط یک درخواست (Single Http Request) ارسال نماید و به همین روال تمام پاسخ‌های معادل درخواست ارسال شده را به صورت یک Pack دیگر دریافت کرده و آن را پردازش نماید. نوع درخواست نیز مهم نیست یعنی می‌توان در قالب یک Pack چندین درخواست از نوع Post و Get یا حتی Put و ... نیز داشته باشید. بدیهی است که پیاده سازی این قابلیت در جای مناسب و در پروژه‌هایی با تعداد کاربران زیاد می‌تواند باعث بهبود چشمگیر کارایی پروژه شود.

برای شروع همانند سایر مطالب می‌توانید از این [پست](#) جهت راه اندازی هاست سرویس‌های Web Api استفاده نمایید. برای فعال سازی قابلیت batching Request نیاز به یک MessageHandler داریم تا بتوانند درخواست‌هایی از این نوع را پردازش نمایند. خوشبختانه به صورت پیش فرض این Handler پیاده سازی شده‌است و ما فقط باید آن را با استفاده از متد MapHttpBatchRoute به بخش مسیر یابی (Route Handler) پروژه معرفی نماییم.

```
public class Startup
{
    public void Configuration(IAppBuilder appBuilder)
    {
        var config = new HttpConfiguration();

        config.Routes.MapHttpBatchRoute(
            routeName: "Batch",
            routeTemplate: "api/$batch",
            batchHandler: new DefaultHttpBatchHandler(GlobalConfiguration.DefaultServer));

        config.MapHttpAttributeRoutes();

        config.Routes.MapHttpRoute(
            name: "Default",
            routeTemplate: "{controller}/{action}/{name}",
            defaults: new { name = RouteParameter.Optional }
        );

        config.Formatters.Clear();
        config.Formatters.Add(new JsonMediaTypeFormatter());
        config.Formatters.JsonFormatter.SerializerSettings.Formatting =
Newtonsoft.Json.Formatting.Indented;
        config.Formatters.JsonFormatter.SerializerSettings.ContractResolver = new
CamelCasePropertyNamesContractResolver();

        config.EnsureInitialized();
        appBuilder.UseWebApi(config);
    }
}
```

مهم‌ترین نکته‌ی آن استفاده از DefaultHttpBatchHandler و معرفی آن به بخش batchHandler مسیریابی است. کلاس DefaultHttpBatchHandler برای وهله سازی نیاز به آبجکت سروری که سرویس‌های WebApi در آن هاست شده‌اند دارد که با دستور GlobalConfiguration.DefaultServer به آن دسترسی خواهید داشت. در صورتی که HttpServer خاص خود را دارید به صورت زیر عمل نمایید:

```
var config = new HttpConfiguration();
HttpServer server = new HttpServer(config);
```


تنظیمات بخش سرور به اتمام رسید. حال نیاز داریم بخش کلاینت را طوری طراحی نماییم که بتواند درخواست را به صورت دسته‌ای ارسال نماید. در زیر یک مثال قرار داده شده است:

```
using System.Net.Http;
using System.Net.Http.Formatting;

public class Program
{
    private static void Main(string[] args)
    {
        string baseAddress = "http://localhost:8080";
        var client = new HttpClient();
        var batchRequest = new HttpRequestMessage(HttpMethod.Post, baseAddress + "/api/$batch")
        {
            Content = new MultipartContent("mixed")
            {
                new HttpResponseMessage(new HttpRequestMessage(HttpMethod.Post, baseAddress +
"/api/Book/Add")
                {
                    Content = new ObjectContent<string>("myBook", new JsonMediaTypeFormatter())
                }),
                new HttpResponseMessage(new HttpRequestMessage(HttpMethod.Get, baseAddress +
"/api/Book/GetAll"))
            };
        };

        var batchResponse = client.SendAsync(batchRequest).Result;

        MultipartStreamProvider streamProvider =
batchResponse.Content.ReadAsMultipartAsync().Result;
        foreach (var content in streamProvider.Contents)
        {
            var response = content.ReadAsHttpResponseMessageAsync().Result;
        }
    }
}
```

همان طور که می‌دانیم برای ارسال درخواست به سرویس Web Api باید یک نمونه از کلاس `HttpRequestMessage` و هله سازی شود سازنده‌ی آن به نوع `HttpMethod` اکشن نظیر (POST یا GET) و آدرس سرویس مورد نظر نیاز دارد. نکته‌ی مهم آن این است که خاصیت `Content` این درخواست باید از نوع `MultipartContent` و `subType` آن نیز باید `mixed` باشد. در بدنه‌ی آن نیز می‌توان تمام درخواست‌ها را به ترتیب و با استفاده از و هله سازی از کلاس `HttpMessageContent` تعریف کرد. برای دریافت پاسخ این گونه درخواست‌ها نیز از متد الحاقی `ReadAsMultipartAsync` استفاده می‌شود که امکان پیمایش بر بدنه‌ی پیام دریافتی را می‌دهد.

مدیریت ترتیب درخواست‌ها

شاید این سوال به ذهن شما نیز خطور کرده باشد که ترتیب پردازش این گونه پیام‌ها چگونه خواهد بود؟ به صورت پیش فرض ترتیب اجرای درخواست‌ها حائز اهمیت است. یعنی تا زمانیکه پردازش درخواست اول به اتمام نرسد، کنترل اجرای برنامه، به درخواست بعدی نخواهد رسید که این مورد بیشتر زمانی رخ می‌دهد که قصد دریافت اطلاعاتی را داشته باشید که قبل از آن باید عمل `Persist` در پایگاه داده اتفاق بیافتد. اما در حالاتی غیر از این می‌توانید این گزینه را غیر فعال کرده تا تمام درخواست‌ها به صورت موازی پردازش شوند که به طور قطع کارایی آن نسبت به حالت قبلی بهینه‌تر است. برای غیر فعال کردن گزینه‌ی ترتیب اجرای درخواست‌ها، به صورت زیر عمل نمایید:

```
config.Routes.MapHttpBatchRoute(
    routeName: "WebApiBatch",
    routeTemplate: "api/$batch",
    batchHandler: new DefaultHttpBatchHandler(GlobalConfiguration.DefaultServer)
    {
        ExecutionOrder = BatchExecutionOrder.NonSequential
    });
```

تفاوت آن فقط در مقدار دهی خاصیت `ExecutionOrder` به صورت `NonSequential` است.