عنوان: **5 قابلیت برتر جدید در ASP.NET Web API 2**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۲۰ ۴:۳۰
آدرس: www.dotnettips.info
گروه‌ها: ASP.Net, ASP.NET Web API, ASP.NET Web API 2

ASP.NET Web API 2 بهمراه 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 جلمه اینکه تمام مسیرها در یک مکان واحد تعریف می‌شوند، اما تنها برای الگوهایی مشخص. مثلا پشتیبانی از nested routing روی یک کنترلر مشکل می‌شود.
در ASP.NET Web API 2 به سادگی می‌توانیم الگوی URI ذکرد 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 بهمراه 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 پروسس هایی 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 *$select*, *$expand* 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

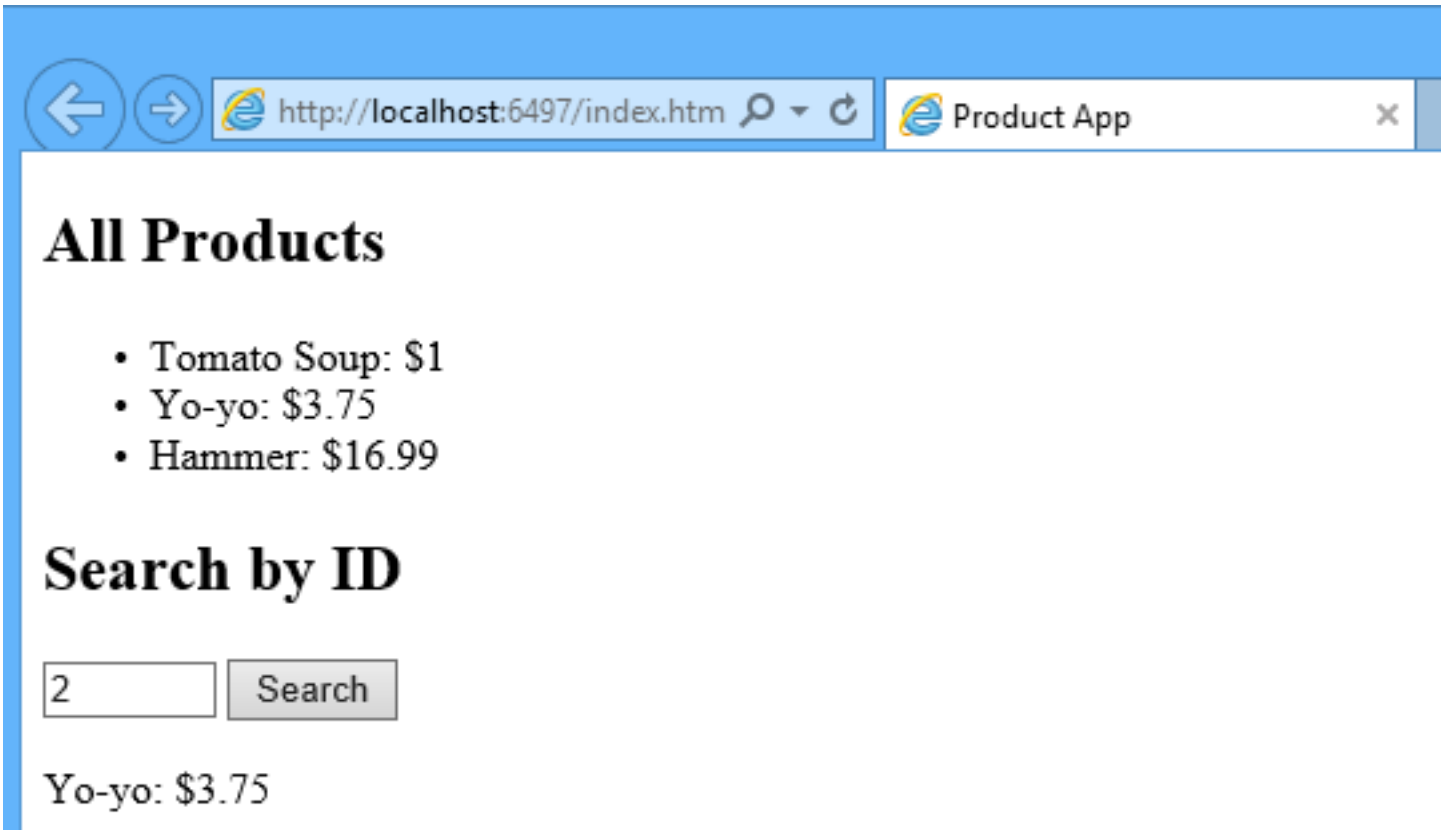
Top 10 HTML5 Interview Questions

JavaScript : Validating Letters and Numbers

عنوان: **شروع کار با ASP.NET Web API 2**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۰/۲۸ ۲۰:۲۵
آدرس: www.dotnettips.info
گروه‌ها: ASP.Net, Web API, ASP.NET Web API 2

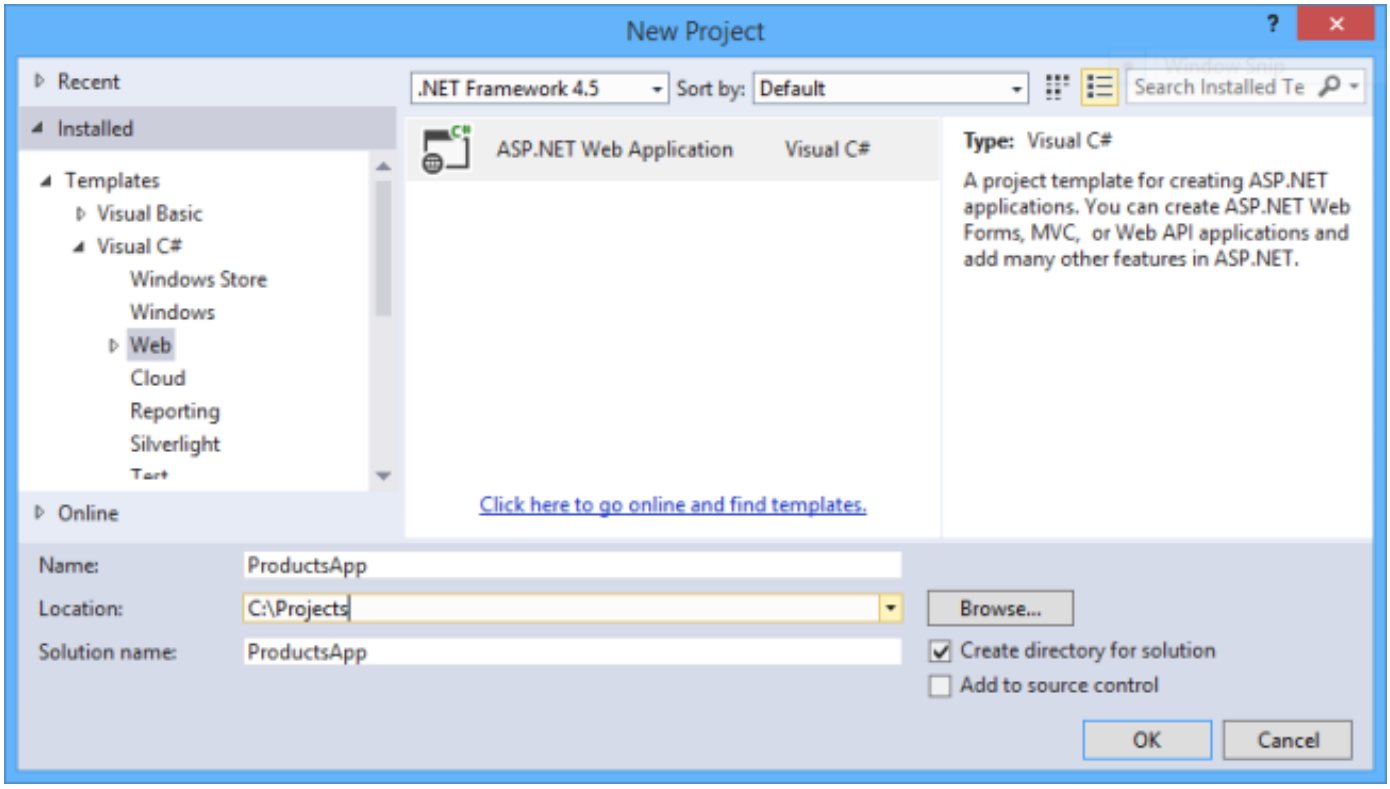
HTTP تنها برای به خدمت گرفتن صفحات وب نیست. این پروتکل همچنین پلتفرمی قدرتمند برای ساختن API هایی است که سرویس‌ها و داده را در دسترس قرار می‌دهند. این پروتکل ساده، انعطاف پذیر و در همه جا حاضر است. هر پلتفرمی که فکرش را بتوانید بکنید کتابخانه ای برای HTTP دارد، بنابراین سرویس‌های HTTP می‌توانند بازه بسیار گسترده ای از کلاینت‌ها را پوشش دهند، مانند مرورگرها، دستگاه‌های موبایل و اپلیکیشن‌های مرسوم دسکتاپ.

ASP.NET Web API فریم ورکی برای ساختن APIهای وب بر روی فریم ورک دات نت است. در این مقاله با استفاده از این فریم ورک، API وبی خواهیم ساخت که لیستی از محصولات را بر می‌گرداند. صفحه وب کلاینت، با استفاده از jQuery نتایج را نمایش خواهد داد.

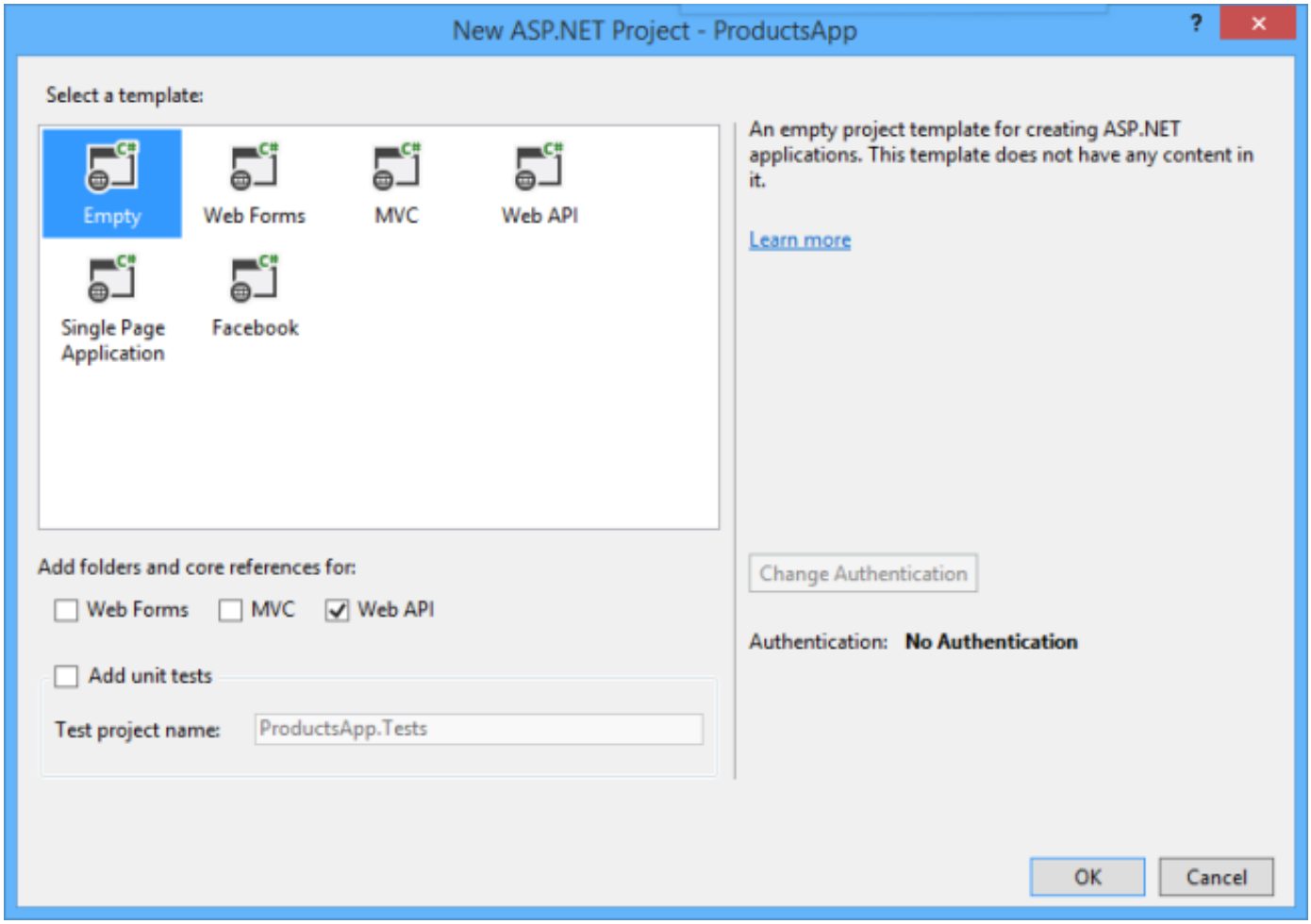


یک پروژه Web API بسازید

در ویژوال استودیو 2013 پروژه جدیدی از نوع ASP.NET Web Application بسازید و نام آن را "ProductsApp" انتخاب کنید.



در دیالوگ New ASP.NET Project قالب Empty را انتخاب کنید و در قسمت "Add folders and core references for" گزینه Web API را انتخاب نمایید.



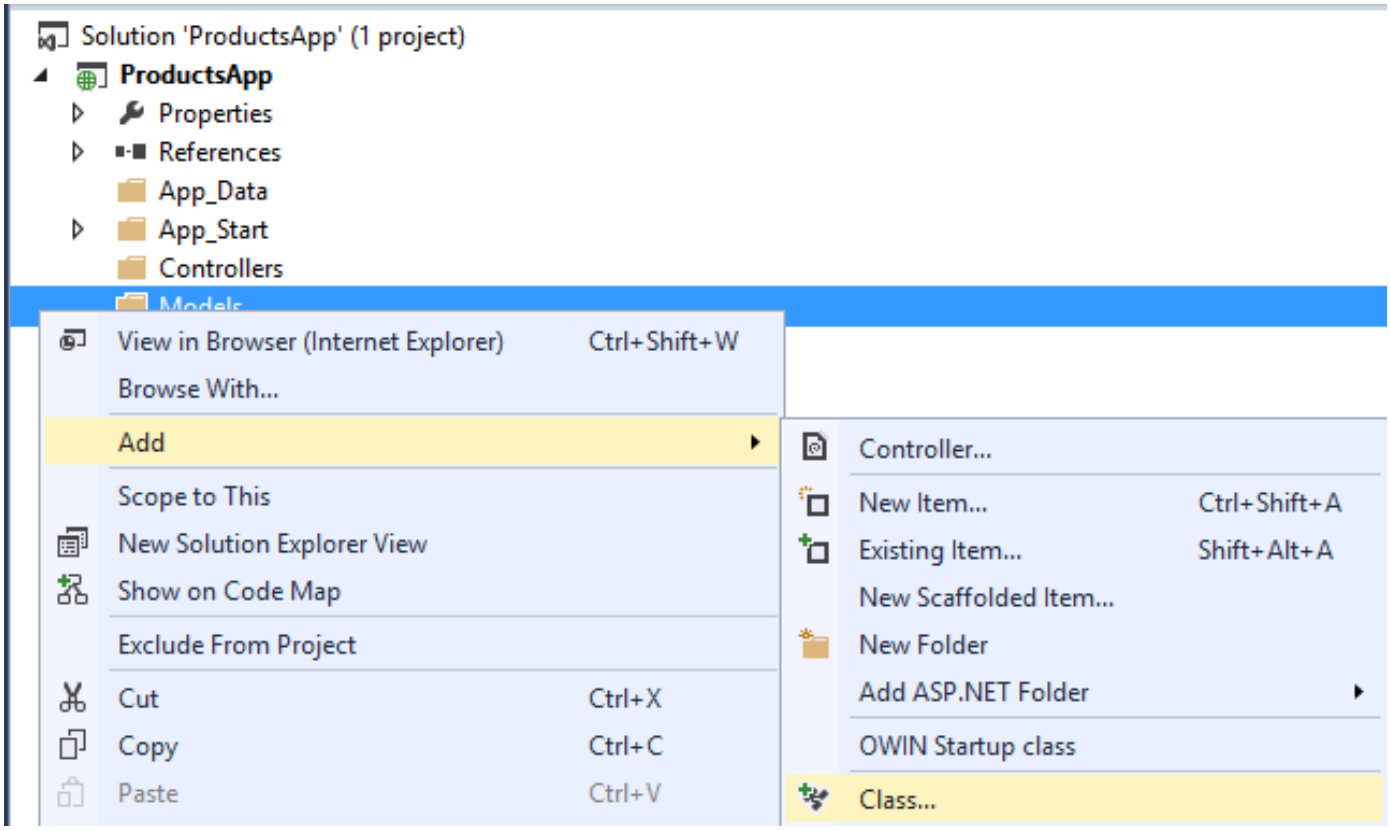
می توانید از قالب Web API هم استفاده کنید. این قالب با استفاده از ASP.NET MVC صفحات راهنمای API را خواهد ساخت. در این مقاله از قالب Empty استفاده میکنیم تا تمرکز اصلی، روی خود فریم ورک Web API باشد. بطور کلی برای استفاده از این فریم ورک لازم نیست با ASP.NET MVC آشنایی داشته باشید.

افزودن یک مدل

یک مدل (model) آبجکتی است که داده اپلیکیشن شما را معرفی می‌کند. ASP.NET Web API می‌تواند بصورت خودکار مدل شما را به JSON, XML و برخی فرمت‌های دیگر مرتب (serialize) کند، و سپس داده مرتب شده را در بدنه پیام HTTP Response بنویسد. تا وقتی که یک کلاینت بتواند فرمت مرتب سازی داده‌ها را بخواند، می‌تواند آبجکت شما را deserialize کند. اکثر کلاینت‌ها می‌توانند XML یا JSON را تفسیر کنند. بعلاوه کلاینت‌ها می‌توانند فرمت مورد نظرشان را با تنظیم Accept header در پیام HTTP Request مشخص کنند.

بگذارید تا با ساختن مدلی ساده که یک محصول (product) را معرفی میکند شروع کنیم.

کلاس جدیدی در پوشه Models ایجاد کنید.



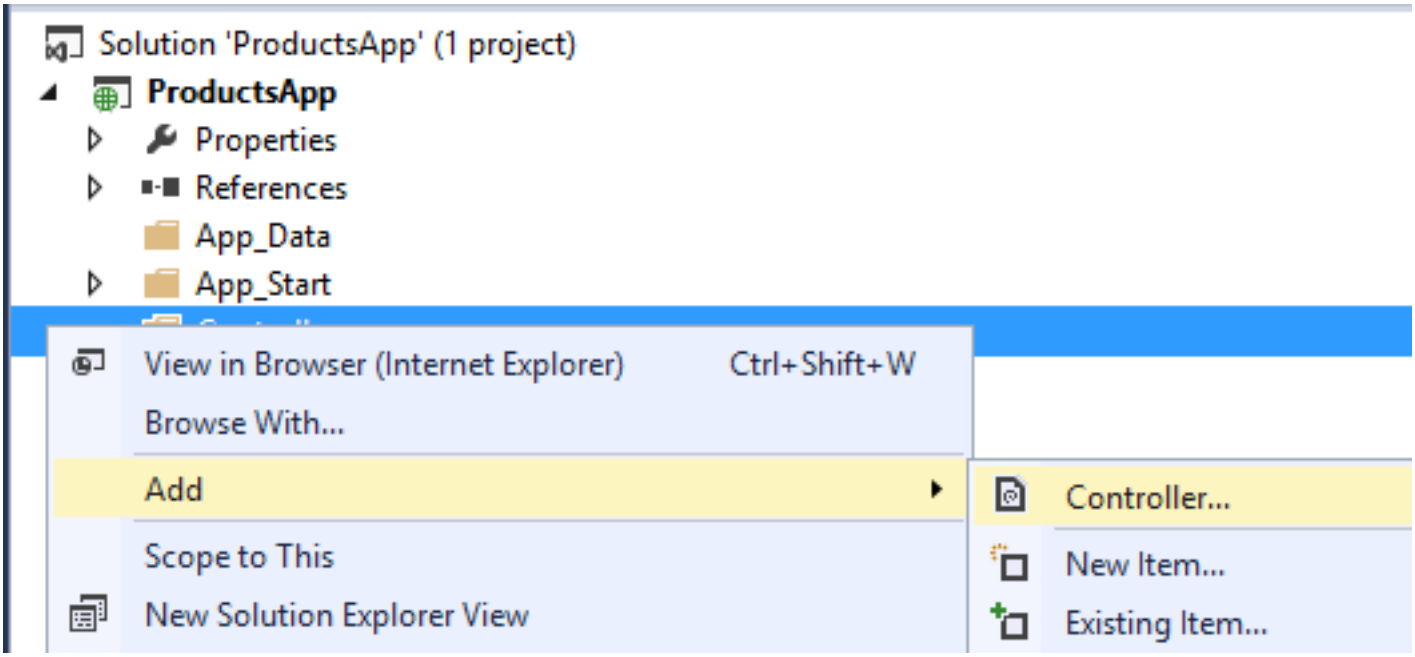
نام کلاس را به "Product" تغییر دهید، و خواص زیر را به آن اضافه کنید.

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

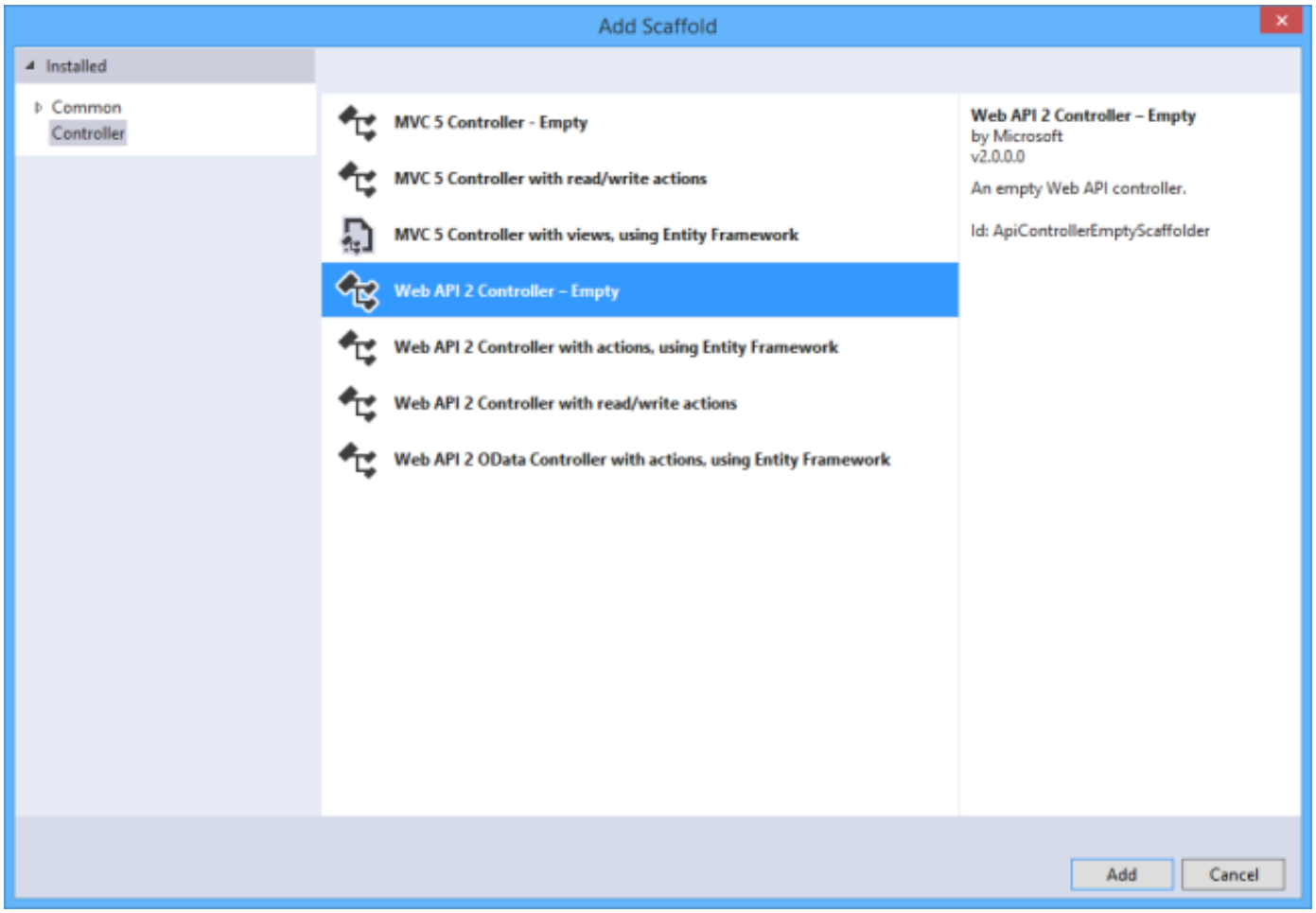
افزودن یک کنترلر

در Web API کنترلرها آبجکت هایی هستند که درخواست‌های HTTP را مدیریت کرده و آنها را به اکشن متدها نگاشت می‌کنند. ما کنترلری خواهیم ساخت که می‌تواند لیستی از محصولات، یا محصولی بخصوص را بر اساس شناسه برگرداند. اگر از ASP.NET MVC استفاده کرده اید، با کنترلرها آشنا هستید. کنترلرهای Web API مشابه کنترلرهای MVC هستند، با این تفاوت که بجای ارث بری از کلاس Controller از کلاس ApiController مشتق می‌شوند.

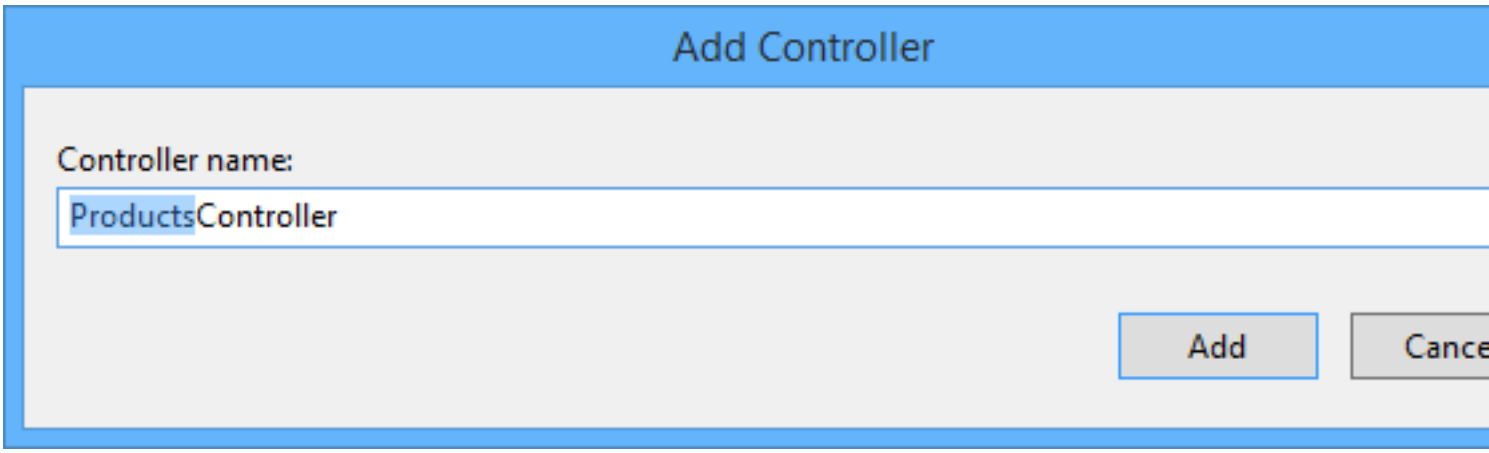
کنترلر جدیدی در پوشه Controllers ایجاد کنید.

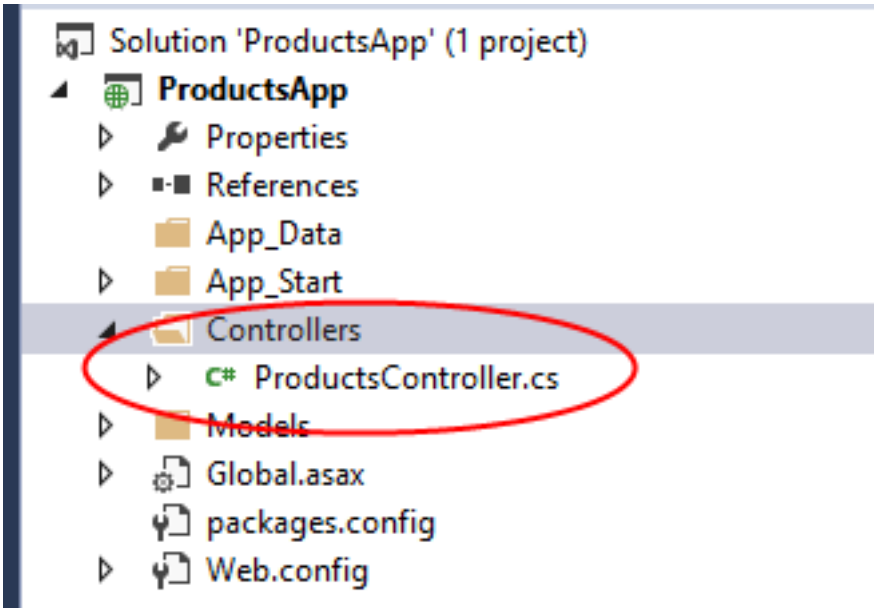


در دیالوگ Add Scaffold گزینه Web API Controller - Empty را انتخاب کرده و روی Add کلیک کنید.



در دیالوگ Add Controller نام کنترلر را به "ProductsController" تغییر دهید و روی Add کلیک کنید.





توجه کنید که ملزم به ساختن کنترلرهای خود در پوشه Controllers نیستید، و این روش صرفا قراردادی برای مرتب نگاه داشتن ساختار پروژه‌ها است. کنترلر ساخته شده را باز کنید و کد زیر را به آن اضافه نمایید.

```
using ProductsApp.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Web.Http;

namespace ProductsApp.Controllers
{
    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 IHttpActionResult GetProduct(int id)
        {
            var product = products.FirstOrDefault((p) => p.Id == id);
            if (product == null)
            {
                return NotFound();
            }
            return Ok(product);
        }
}
```

برای اینکه مثال جاری را ساده نگاه داریم، محصولات مورد نظر در یک آرایه استاتیک ذخیره شده اند. مسلما در یک اپلیکیشن واقعی برای گرفتن لیست محصولات از دیتابیس یا منبع داده ای دیگر کوئری می‌گیرید.

کنترلر ما دو متد برای دریافت محصولات تعریف می‌کند:

متد GetAllProducts لیست تمام محصولات را در قالب یک IEnumerable<Product> بر می‌گرداند.
متد GetProductById سعی می‌کند محصولی را بر اساس شناسه تعیین شده پیدا کند.

همین! حالا یک Web API ساده دارید. هر یک از متدهای این کنترلر، به یک یا چند URI پاسخ می‌دهند:

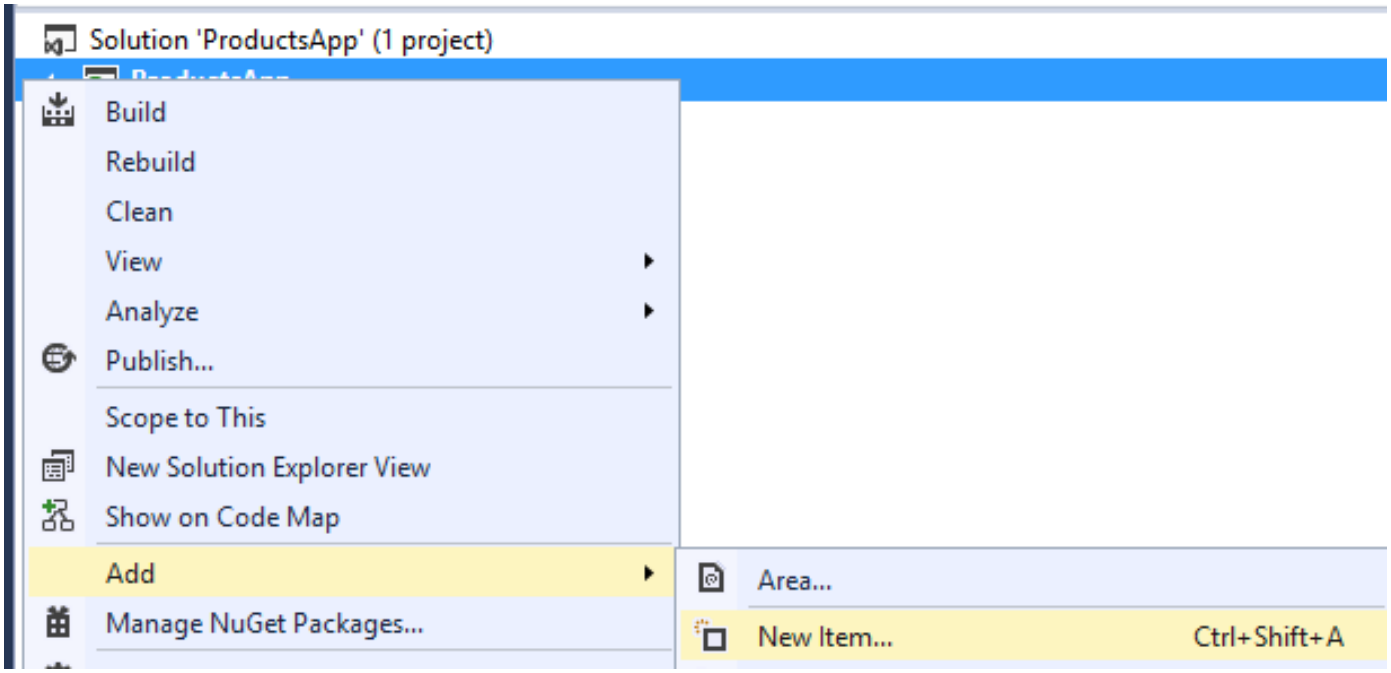
Controller Method	URI
GetAllProducts	api/products/
GetProductById	api/products/ *id* /

برای اطلاعات بیشتر درباره نحوه نگاشت درخواست‌های HTTP به اکشن متدها توسط Web API به این لینک مراجعه کنید.

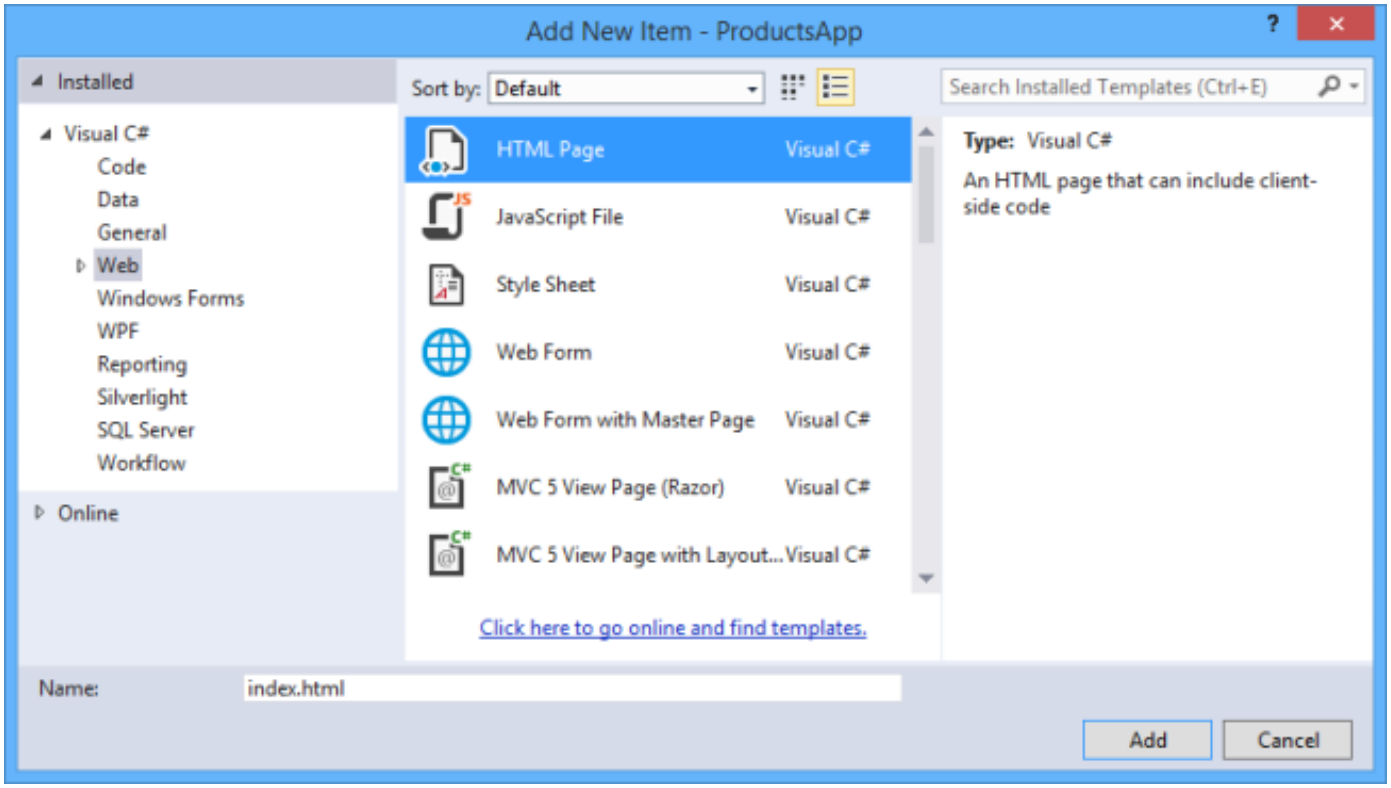
فراخوانی Web API با جاوا اسکریپت و jQuery

در این قسمت یک صفحه HTML خواهیم ساخت که با استفاده از AJAX متدهای Web API را فراخوانی می‌کند. برای ارسال درخواست‌های آژاکسی و بروز رسانی صفحه بمنظور نمایش نتایج دریافتی از jQuery استفاده میکنیم.

در پنجره Solution Explorer روی نام پروژه کلیک راست کرده و گزینه Add, New Item را انتخاب کنید.



در دیالوگ Add New Item قالب HTML Page را انتخاب کنید و نام فایل را به "index.html" تغییر دهید.



حال محتوای این فایل را با لیست زیر جایگزین کنید.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
```

```
  <title>Product App</title>
</head>
<body>

  <div>
    <h2>All Products</h2>
    <ul id="products" />
  </div>
  <div>
    <h2>Search by ID</h2>
    <input type="text" id="prodId" size="5" />
    <input type="button" value="Search" onclick="find();" />
    <p id="product" />
  </div>

  <script src="http://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.0.3.min.js"></script>
  <script>
    var uri = 'api/products';

    $(document).ready(function () {
      // Send an AJAX request
      $.getJSON(uri)
          .done(function (data) {
            // On success, 'data' contains a list of products.
            $.each(data, function (key, item) {
              // Add a list item for the product.
              $('<li>', { text: formatItem(item) }).appendTo($('#products'));
            });
          });
    });

    function formatItem(item) {
      return item.Name + ': $' + item.Price;
    }

    function find() {
      var id = $('#prodId').val();
      $.getJSON(uri + '/' + id)
          .done(function (data) {
            $('#product').text(formatItem(data));
          })
          .fail(function (jqXHR, textStatus, err) {
            $('#product').text('Error: ' + err);
          });
    }
  </script>
</body>
</html>
```

راه‌های مختلفی برای گرفتن jQuery وجود دارد، در این مثال از Microsoft Ajax CDN استفاده شده. می‌توانید این کتابخانه را از http://jquery.com دانلود کنید و بصورت محلی استفاده کنید. همچنین قالب پروژه‌های Web API این کتابخانه را به پروژه نیز اضافه می‌کنند.

گرفتن لیستی از محصولات

برای گرفتن لیستی از محصولات، یک درخواست HTTP GET به آدرس "/api/products" ارسال کنید.

تابع getJSON یک درخواست آژاکسی ارسال می‌کند. پاسخ دریافتی هم آرایه ای از آبجکت‌های JSON خواهد بود. تابع done در صورت موفقیت آمیز بودن درخواست، اجرا می‌شود. که در این صورت ما DOM را با اطلاعات محصولات بروز رسانی می‌کنیم.

```
$(document).ready(function () {
    // Send an AJAX request
    $.getJSON(apiUrl)
        .done(function (data) {
            // On success, 'data' contains a list of products.
            $.each(data, function (key, item) {
                // Add a list item for the product.
                $('<li>', { text: formatItem(item) }).appendTo($('#products'));
            });
        });
});
```

گرفتن محصولی مشخص

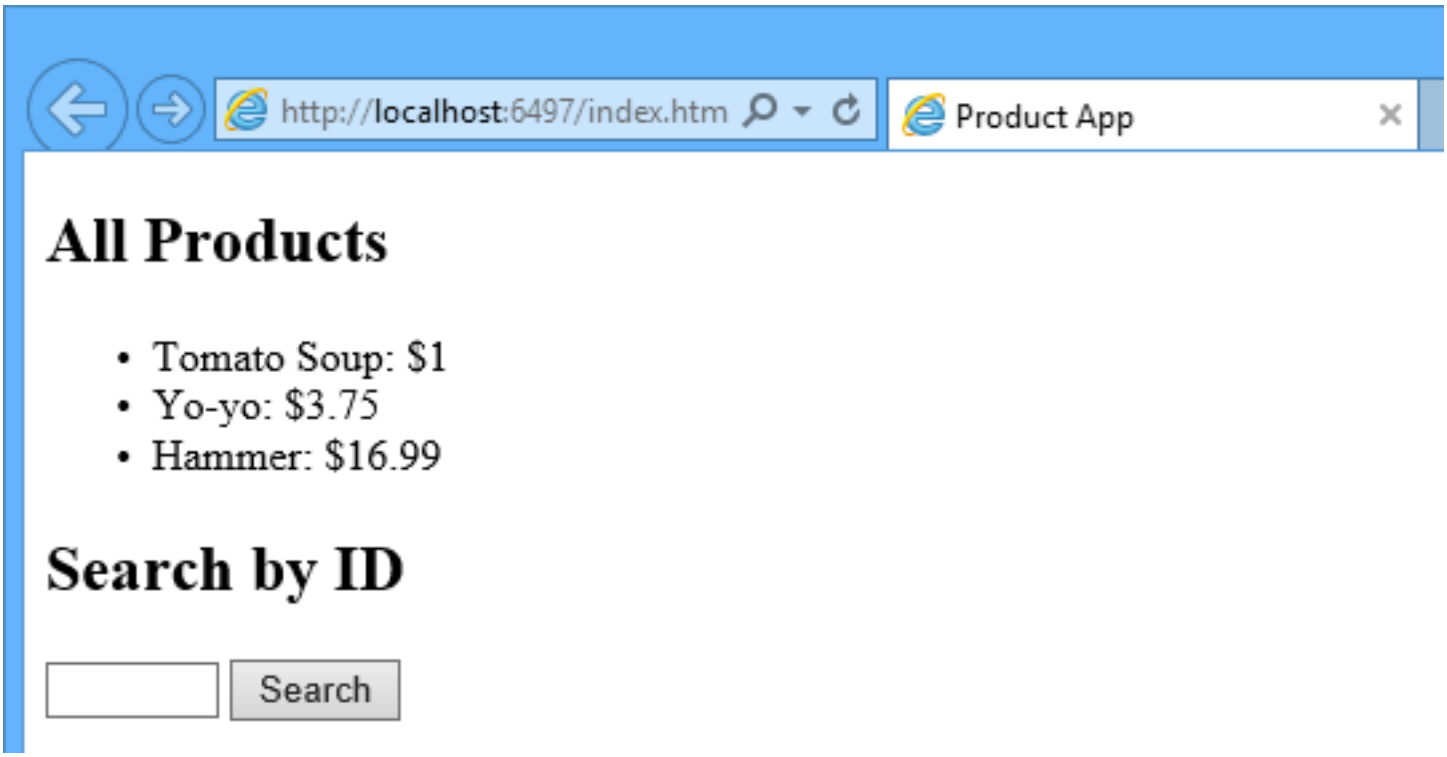
برای گرفتن یک محصول توسط شناسه (ID) آن کافی است یک درخواست HTTP GET به آدرس "/api/products/id" ارسال کنید.

```
function find() {
    var id = $('#prodId').val();
    $.getJSON(apiUrl + '/' + id)
        .done(function (data) {
            $('#product').text(formatItem(data));
        })
        .fail(function (jqXHR, textStatus, err) {
            $('#product').text('Error: ' + err);
        });
}
```

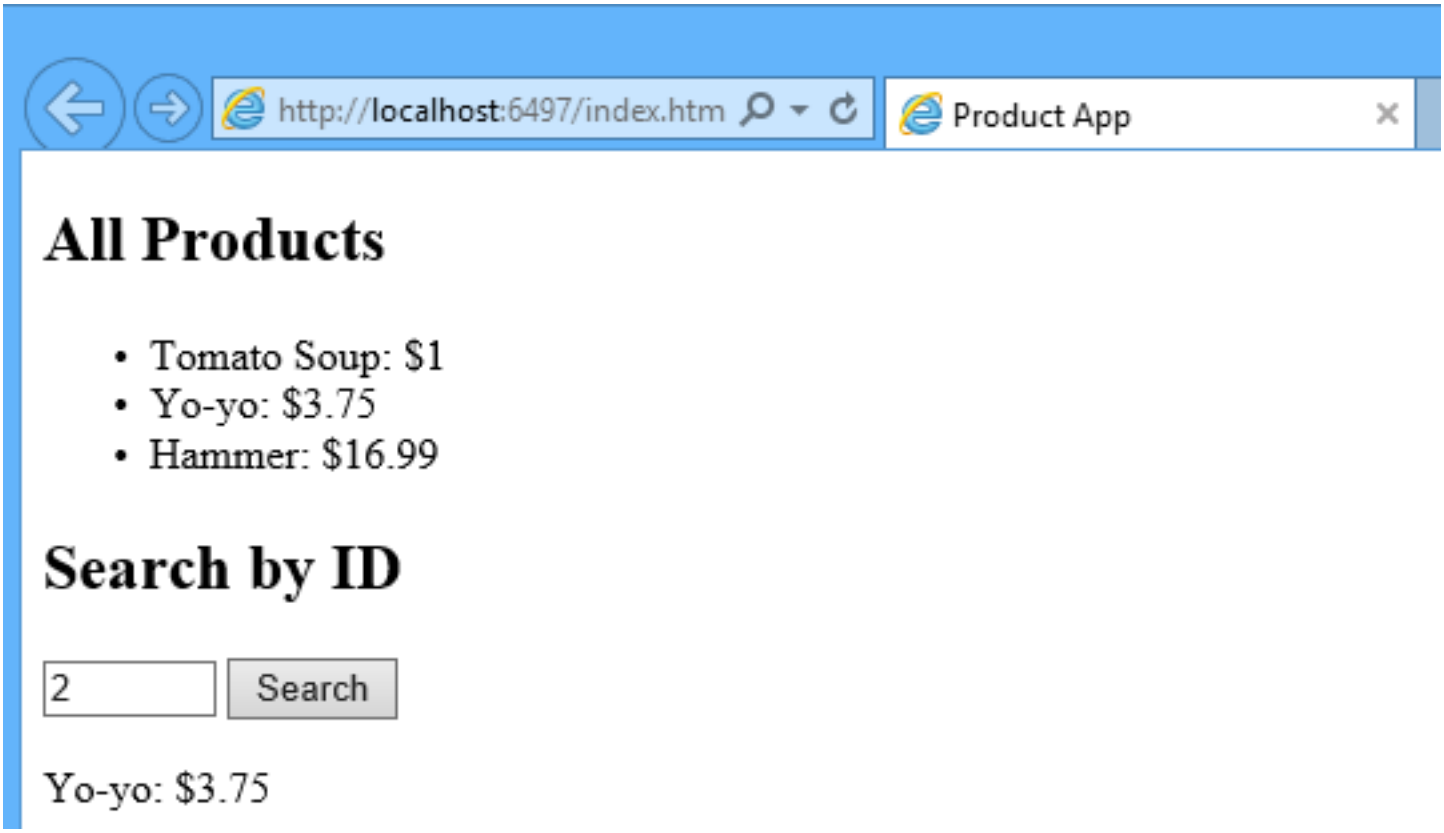
برای این کار هنوز از getJSON برای ارسال درخواست آژاکسی استفاده می‌کنیم، اما اینبار شناسه محصول را هم به آدرس درخواستی اضافه کرده ایم. پاسخ دریافتی از این درخواست، اطلاعات یک محصول با فرمت JSON است.

اجرای اپلیکیشن

اپلیکیشن را با F5 اجرا کنید. صفحه وب باز شده باید چیزی مشابه تصویر زیر باشد.



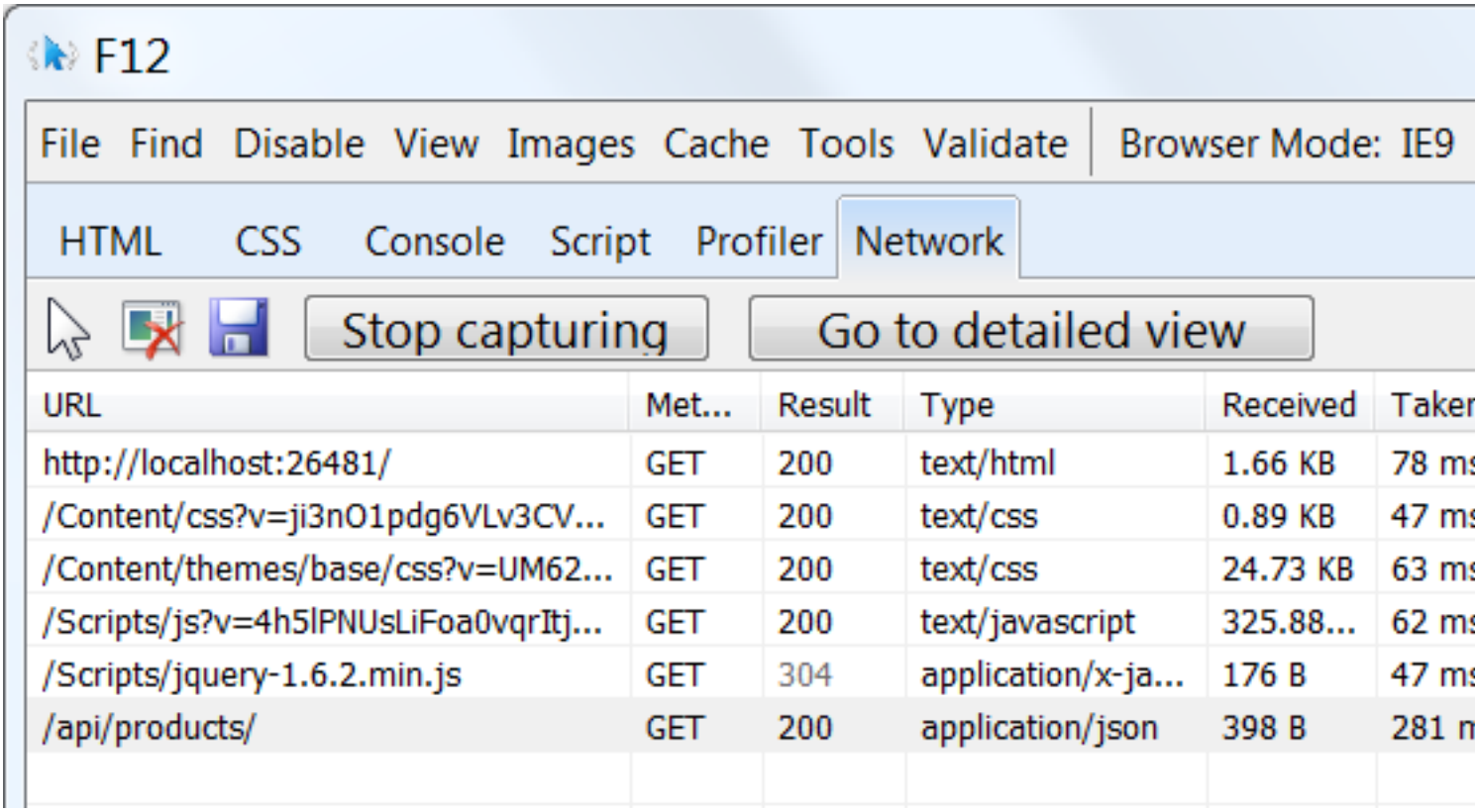
برای گرفتن محصولی مشخص، شناسه آن را وارد کنید و روی Search کلیک کنید.



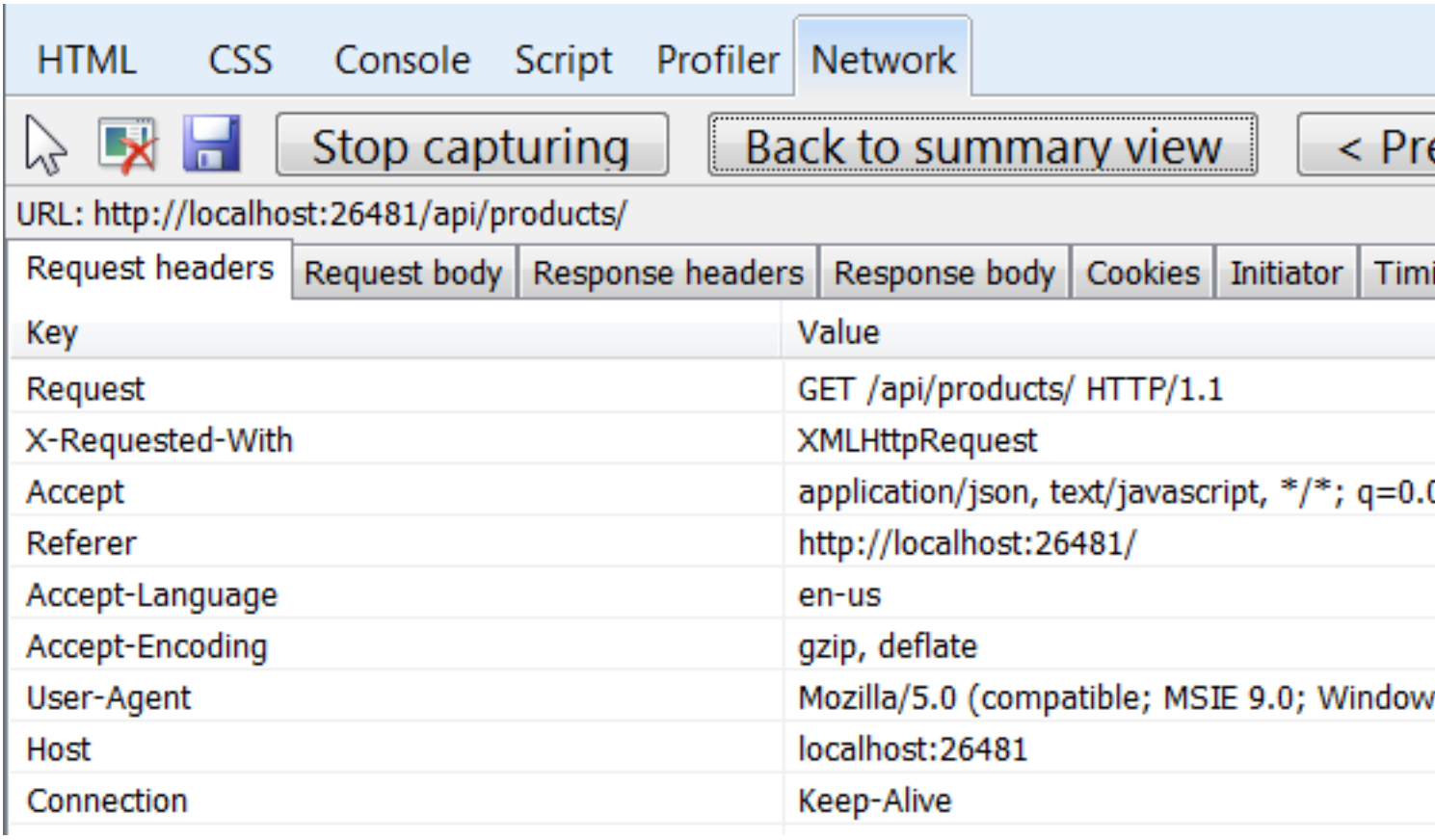
اگر شناسه نامعتبری وارد کنید، سرور یک خطای HTTP بر می‌گرداند.

استفاده از F12 برای مشاهده درخواست‌ها و پاسخ ها

هنگام کار با سرویس‌های HTTP، مشاهده‌ی درخواست‌های ارسال شده و پاسخ‌های دریافتی بسیار مفید است. برای اینکار می‌توانید از ابزار توسعه دهندگان وب استفاده کنید، که اکثر مرورگرهای مدرن، پیاده سازی خودشان را دارند. در اینترنت اکسپلورر می‌توانید با F12 به این ابزار دسترسی پیدا کنید. به برگه Network بروید و روی Start Capturing کلیک کنید. حالا صفحه وب را مجددا بارگذاری (reload) کنید. در این مرحله اینترنت اکسپلورر ترافیک HTTP بین مرورگر و سرور را تسخیر می‌کند. می‌توانید تمام ترافیک HTTP روی صفحه جاری را مشاهده کنید.



به دنبال آدرس نسبی "/api/products" بگردید و آن را انتخاب کنید. سپس روی Go to detailed view کلیک کنید تا جزئیات ترافیک را مشاهده کنید. در نمای جزئیات، می‌توانید headerها و بدنه درخواست‌ها و پاسخ‌ها را ببینید. مثلا اگر روی برگه Request headers کلیک کنید، خواهید دید که اپلیکیشن ما در Accept header داده‌ها را با فرمت "application/json" درخواست کرده است.



اگر روی برگه Response body کلیک کنید، می‌توانید ببینید چگونه لیست محصولات با فرمت JSON سریال شده است. همانطور که گفته شده مرورگرهای دیگر هم قابلیت‌های مشابهی دارند. یک ابزار مفید دیگر Fiddler است. با استفاده از این ابزار می‌توانید تمام ترافیک HTTP خود را مانیتور کرده، و همچنین درخواست‌های جدیدی بسازید که این امر کنترل کاملی روی HTTP headers به شما می‌دهد.

قدم‌های بعدی

برای یک مثال کامل از سرویس‌های HTTP که از عملیات POST,PUT و DELETE پشتیبانی می‌کند به این لینک مراجعه کنید.
برای اطلاعات بیشتر درباره طراحی واکنش گرا در کنار سرویس‌های HTTP به این لینک مراجعه کنید، که اپلیکیشن‌های تک صفحه ای (SPA) را بررسی می‌کند.

نظرات خوانندگان

نویسنده: پوریا منفرد
تاریخ: ۱۳۹۳/۰۳/۰۵ ۱:۲۶

من یه سوالی برام پیش اومده اینه که
میشه از Web API برای پروژه‌های بزرگ مبتنی بر روی HTTP استفاده کرد؟
منظورم از پروژه‌های بزرگ یعنی Request هایی که شاید اطلاعات برگشتی مثلا بیش از 1000 رکورد باشه
آیا شدنیه؟
یعنی منبع داده بتونه بوسیله Web API عملیات‌های Crud رو بر روی بستر اینترنت برای پروژه‌های این چنینی که امکان واکشی اطلاعات بیشمار و ورود اطلاعات همزمان بوسیله کاربرهای مختلف با دیوایس‌های مختلف وجود داره رو ارائه بده؟

نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۳/۰۳/۰۷ ۰:۷

بله. به طور کلی، هر پلتفرمی که دارای کتابخانه ای جهت کار با سرویس‌های Http است می‌تواند از سرویس‌های Asp.Net WebApi استفاده نماید.
اما در هنگام پیاده سازی پروژه‌های مقیاس بزرگ حتما به طراحی زیر ساخت توجه ویژه ای داشته باشید. اگر کتاب‌های Designing Evolvable Web Api With Asp.Net یا Pro Asp.Net Web Api : Http Web Service In Asp.Net را مطالعه نکردید بهتون پیشنهاد می‌کنم قبل از شروع به کار حتما نگاهی به آن‌ها بیندازید.
در همین رابطه:
»مقایسه بین امکانات Web Api و WCF

عنوان: **ساخت یک Web API که از عملیات CRUD پشتیبانی می کند**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۱/۰۲ ۴:۵۵
آدرس: www.dotnettips.info
گروه‌ها: ASP.NET Web API 2

در این مقاله با استفاده از ASP.NET Web API یک سرویس HTTP خواهیم ساخت که از عملیات CRUD پشتیبانی می‌کند. CRUD مخفف Create, Read, Update, Delete است که عملیات پایه دیتابیسی هستند. بسیاری از سرویس‌های HTTP این عملیات را بصورت REST API هم مدل سازی می‌کنند. در مثال جاری سرویس ساده ای خواهیم ساخت که مدیریت لیستی از محصولات (Products) را ممکن می‌سازد. هر محصول شامل فیلدهای شناسه (ID)، نام، قیمت و طبقه بندی خواهد بود.

سرویس ما متدهای زیر را در دسترس قرار می‌دهد.

Action	HTTP method	Relative URl
گرفتن لیست تمام محصولات	GET	api/products/
گرفتن یک محصول بر اساس شناسه	GET	api/products/ *id* /
گرفتن یک محصول بر اساس طبقه بندی	GET	api/products?category= *category* /
ایجاد یک محصول جدید	POST	api/products/
بروز رسانی یک محصول	PUT	api/products/ *id* /
حذف یک محصول	DELETE	api/products/ *id* /

همانطور که مشاهده می‌کنید برخی از آدرس ها، شامل شناسه محصول هم می‌شوند. بعنوان مثال برای گرفتن محصولی با شناسه 28، کلاینت یک درخواست GET را به آدرس زیر ارسال می‌کند:

```
http://hostname/api/products/28
```

منابع

سرویس ما آدرس هایی برای دستیابی به دو نوع منبع (resource) را تعریف می‌کند:

Resource	URI
لیست تمام محصولات	api/products/
یک محصول مشخص	api/products/ *id* /

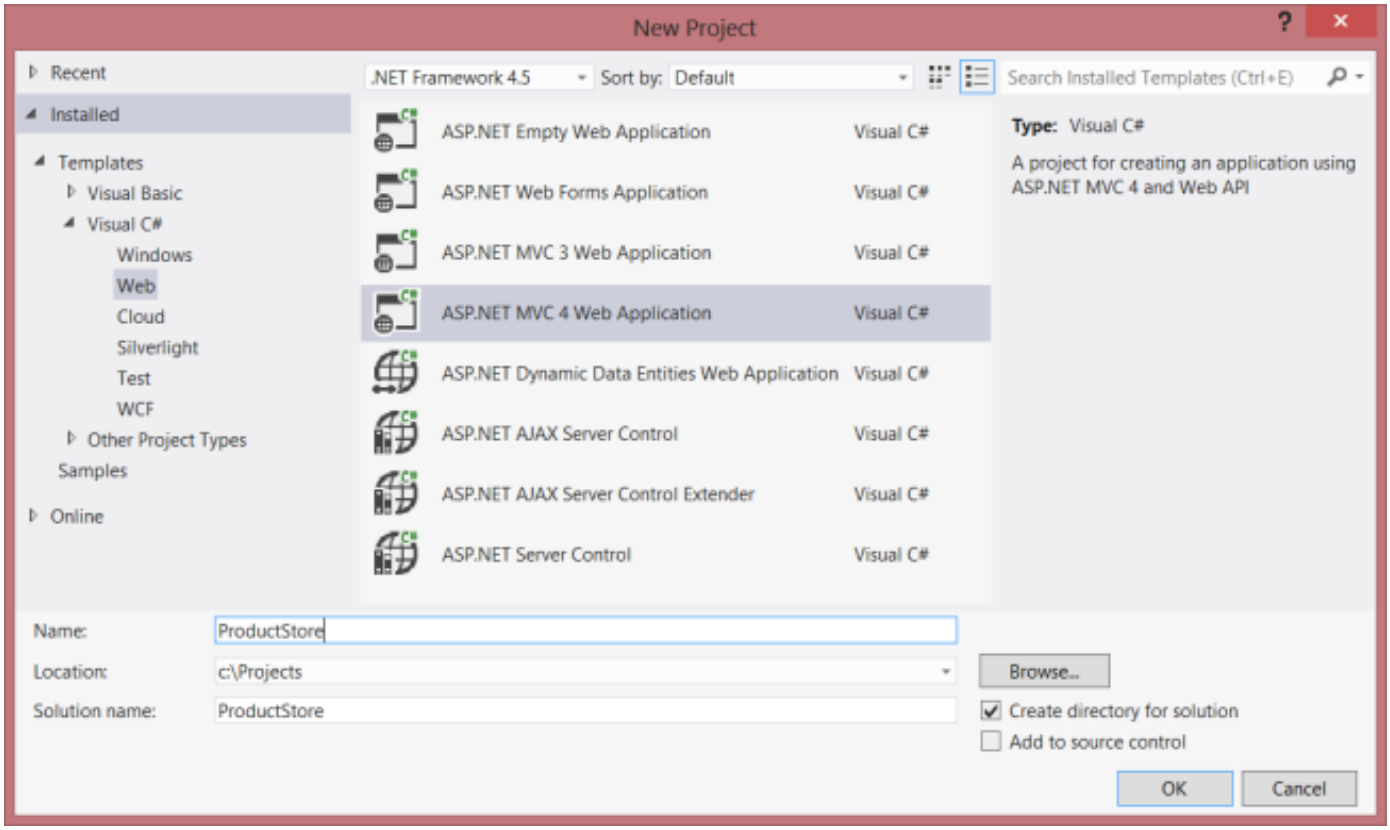
متد ها

چهار متد اصلی HTTP یعنی همان GET, PUT, POST, DELETE می‌توانند بصورت زیر به عملیات CRUD نگاشت شوند:
متد GET یک منبع (resource) را از آدرس تعریف شده دریافت می‌کند. متدهای GET هیچگونه تاثیری روی سرور نباید داشته باشند. مثلا حذف رکوردها با متد اکیدا اشتباه است.
متد PUT یک منبع را در آدرس تعریف شده بروز رسانی می‌کند. این متد برای ساختن منابع جدید هم می‌تواند استفاده شود، البته در صورتی که سرور به کلاینت‌ها اجازه مشخص کردن آدرس‌های جدید را بدهد. در مثال جاری پشتیبانی از ایجاد منابع توسط متد PUT را بررسی نخواهیم کرد.
متد POST منبع جدیدی می‌سازد. سرور آدرس آبجکت جدید را تعیین می‌کند و آن را بعنوان بخشی از پیام Response بر می‌گرداند.
متد DELETE منبعی را در آدرس تعریف شده حذف می‌کند.

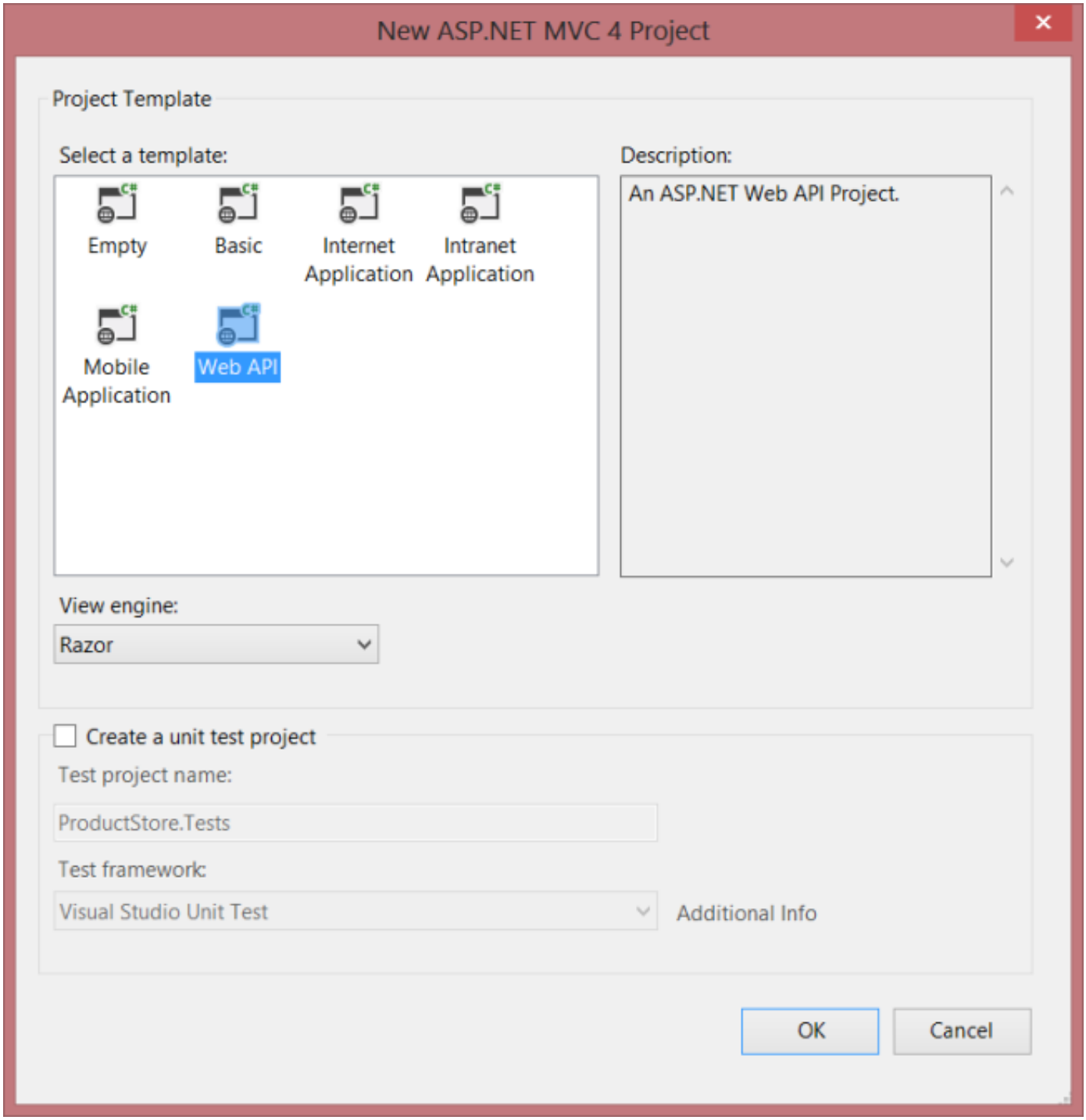
نکته: متد PUT موجودیت محصول (product entity) را کاملا جایگزین میکند. به بیان دیگر، از کلاینت انتظار می‌رود که آبجکت کامل محصول را برای بروز رسانی ارسال کند. اگر می‌خواهید از بروز رسانی‌های جزئی/پاره ای (partial) پشتیبانی کنید متد PATCH توصیه می‌شود. مثال جاری متد PATCH را پیاده سازی نمی‌کند.

یک پروژه Web API جدید بسازید

ویژوال استودیو را باز کنید و پروژه جدیدی از نوع ASP.NET MVC Web Application بسازید. نام پروژه را به "ProductStore" تغییر دهید و OK کنید.



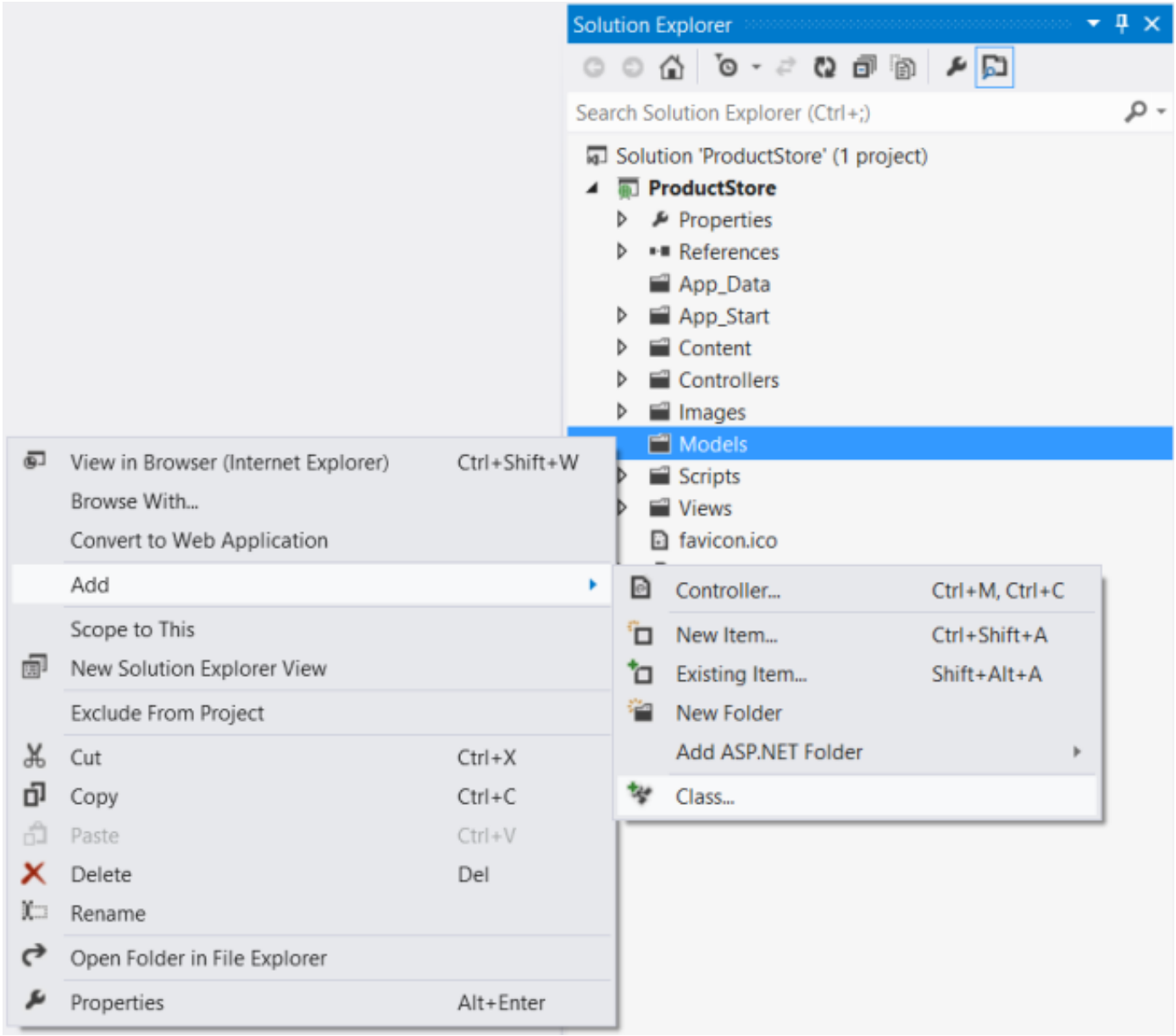
در دیالوگ New ASP.NET Project قالب Web API را انتخاب کرده و تایید کنید.



افزودن یک مدل

یک مدل، آبجکتی است که داده اپلیکیشن شما را نمایندگی می‌کند. در ASP.NET Web API می‌توانید از آبجکت‌های Strongly-typed بعنوان مدل هایتان استفاده کنید که بصورت خودکار برای کلاینت به فرمت‌های JSON, XML مرتب (Serialize) می‌شوند. در مثال جاری، داده‌های ما محصولات هستند. پس کلاس جدیدی بنام Product می‌سازیم.

در پوشه Models کلاس جدیدی با نام Product بسازید.



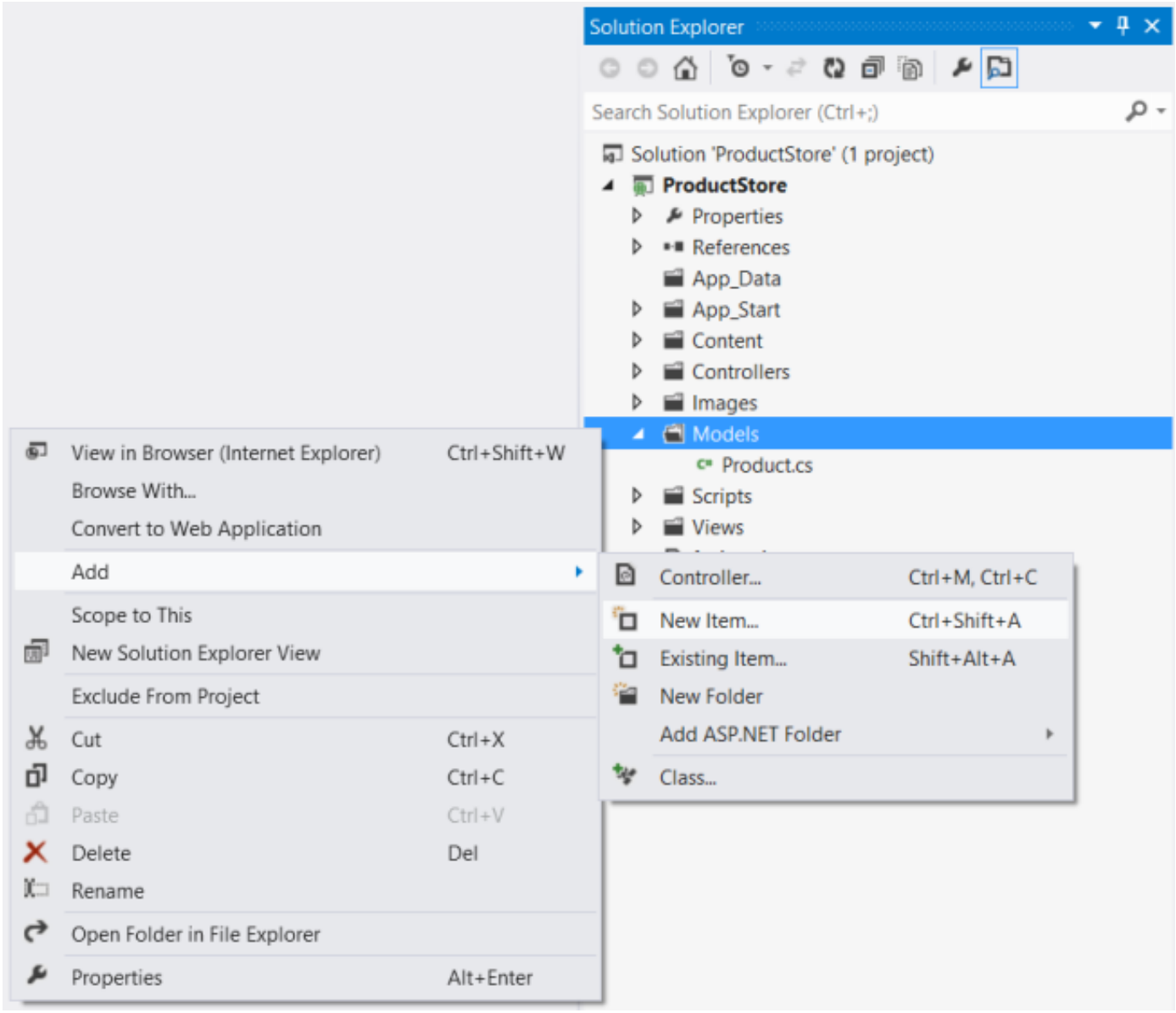
حال خواص زیر را به این کلاس اضافه کنید.

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

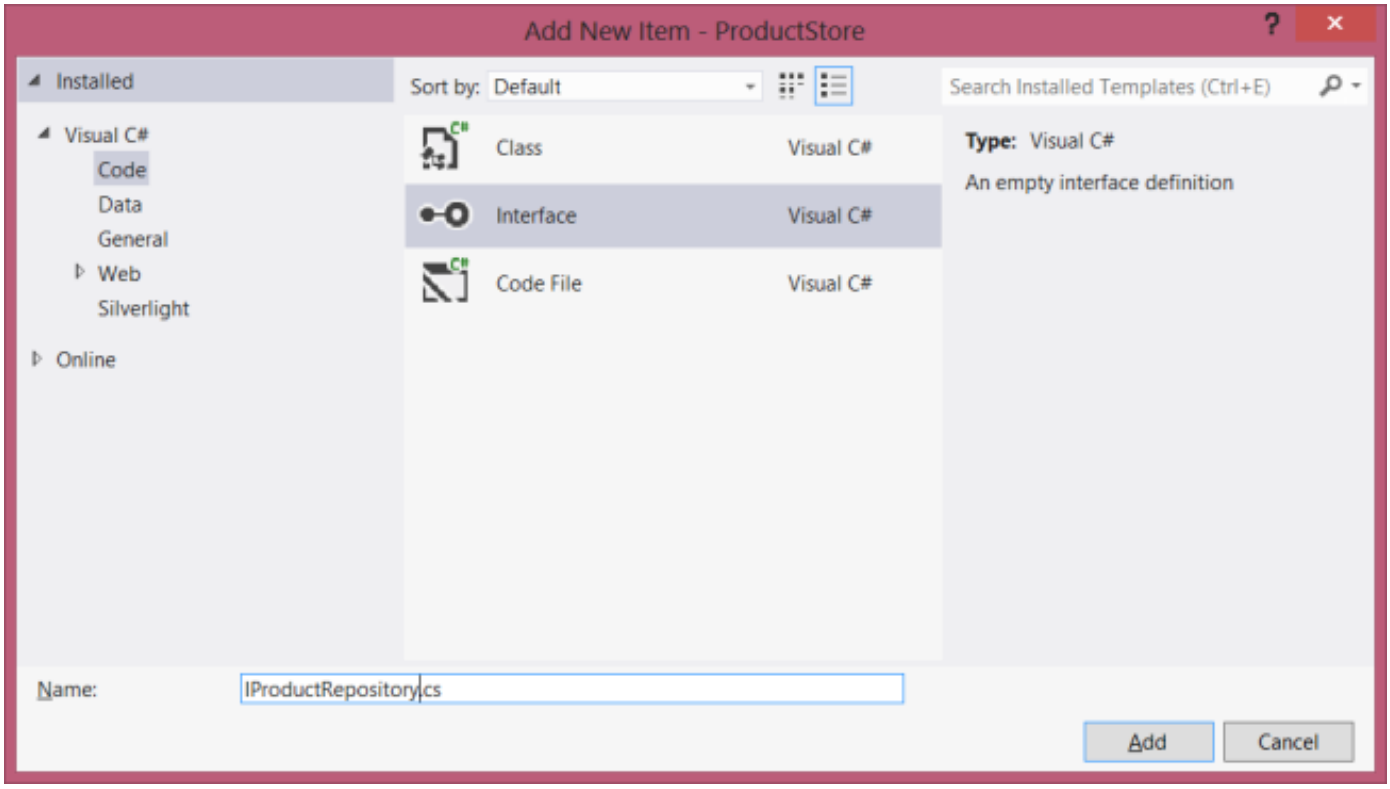
افزودن یک مخزن

ما نیاز به ذخیره کردن کلکسیونی از محصولات داریم، و بهتر است این کلکسیون از پیاده سازی سرویس تفکیک شود. در این صورت بدون نیاز به بازنویسی کلاس سرویس می‌توانیم منبع داده‌ها را تغییر دهیم. این نوع طراحی با نام الگوی مخزن یا Repository Pattern شناخته می‌شود. برای شروع نیاز به یک قرارداد جنریک برای مخزن‌ها داریم.

روی پوشه Models کلیک راست کنید و گزینه Add, New Item را انتخاب نمایید.



نوع آیتم جدید را Interface انتخاب کنید و نام آن را به IProductRepository تغییر دهید.



حال کد زیر را به این اینترفیس اضافه کنید.

```
namespace ProductStore.Models
{
    public interface IProductRepository
    {
        IEnumerable<Product> GetAll();
        Product Get(int id);
        Product Add(Product item);
        void Remove(int id);
        bool Update(Product item);
    }
}
```

حال کلاس دیگری با نام ProductRepository در پوشه Models ایجاد کنید. این کلاس قرارداد IProductRepository را پیاده سازی خواهد کرد. کد زیر را به این کلاس اضافه کنید.

```
namespace ProductStore.Models
{
    public class ProductRepository : IProductRepository
    {
        private List<Product> products = new List<Product>();
        private int _nextId = 1;

        public ProductRepository()
        {
            Add(new Product { Name = "Tomato soup", Category = "Groceries", Price = 1.39M });
            Add(new Product { Name = "Yo-yo", Category = "Toys", Price = 3.75M });
            Add(new Product { Name = "Hammer", Category = "Hardware", Price = 16.99M });
        }

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

        public Product Get(int id)
        {
```

```
            return products.Find(p => p.Id == id);
        }

        public Product Add(Product item)
        {
            if (item == null)
            {
                throw new ArgumentNullException("item");
            }
            item.Id = _nextId++;
            products.Add(item);
            return item;
        }

        public void Remove(int id)
        {
            products.RemoveAll(p => p.Id == id);
        }

        public bool Update(Product item)
        {
            if (item == null)
            {
                throw new ArgumentNullException("item");
            }
            int index = products.FindIndex(p => p.Id == item.Id);
            if (index == -1)
            {
                return false;
            }
            products.RemoveAt(index);
            products.Add(item);
            return true;
        }
    }
}
```

مخزن ما لیست محصولات را در حافظه محلی نگهداری می‌کند. برای مثال جاری این طراحی کافی است، اما در یک اپلیکیشن واقعی داده‌های شما در یک دیتابیس یا منبع داده ابری ذخیره خواهند شد. همچنین استفاده از الگوی مخزن، تغییر منبع داده‌ها در آینده را راحت‌تر می‌کند.

افزودن یک کنترلر Web API

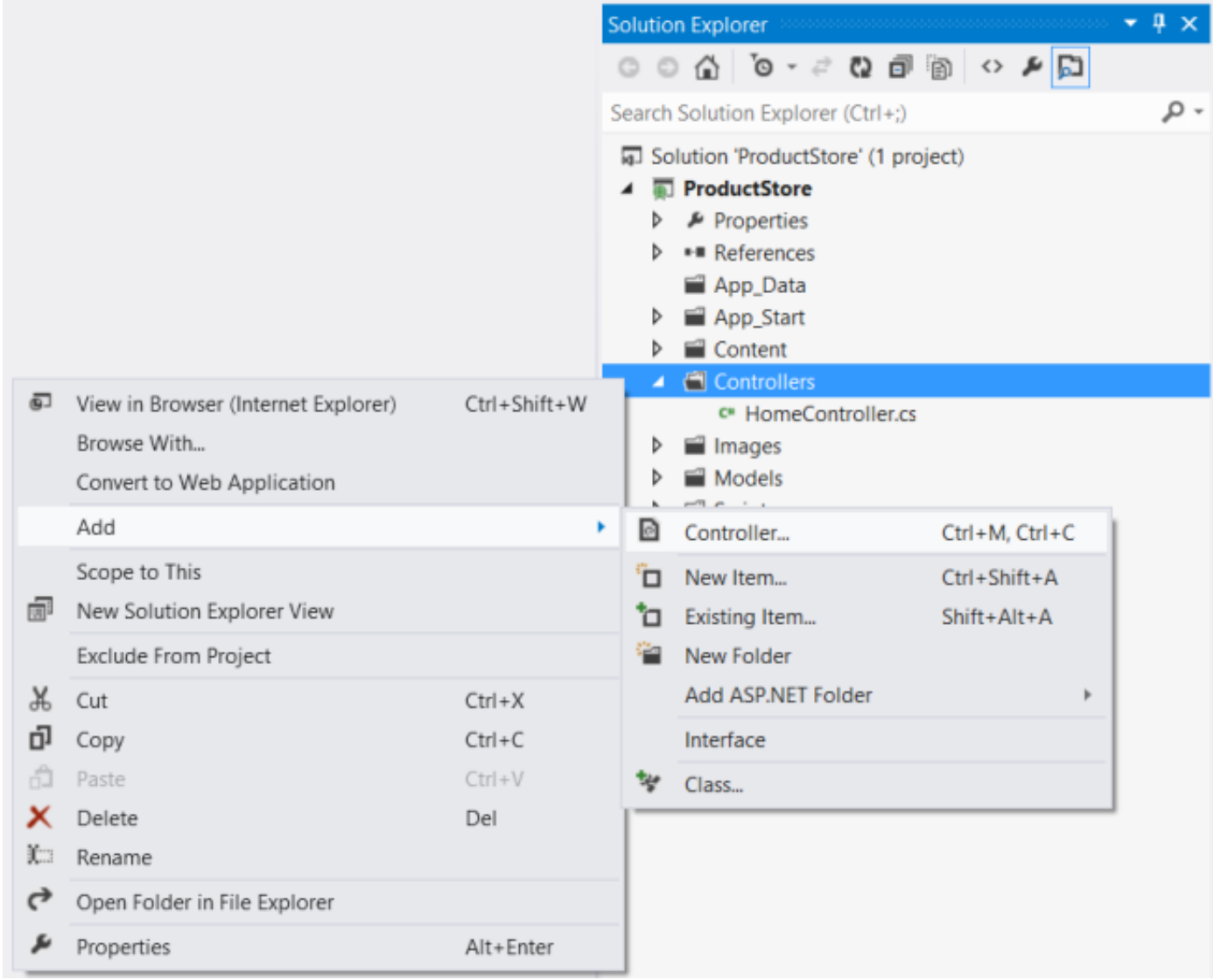
اگر قبلا با ASP.NET MVC کار کرده باشید، با مفهوم کنترلرها آشنایی دارید. در ASP.NET Web API کنترلرها کلاس هایی هستند که درخواست‌های HTTP دریافتی از کلاینت را به اکشن متدها نگاشت می‌کنند. ویژوال استودیو هنگام ساختن پروژه شما دو کنترلر به آن اضافه کرده است. برای مشاهد آنها پوشه Controllers را باز کنید.

HomeController یک کنترلر مرسوم در ASP.NET MVC است. این کنترلر مسئول بکار گرفتن صفحات وب است و مستقیما ربطی به Web API ما ندارد.

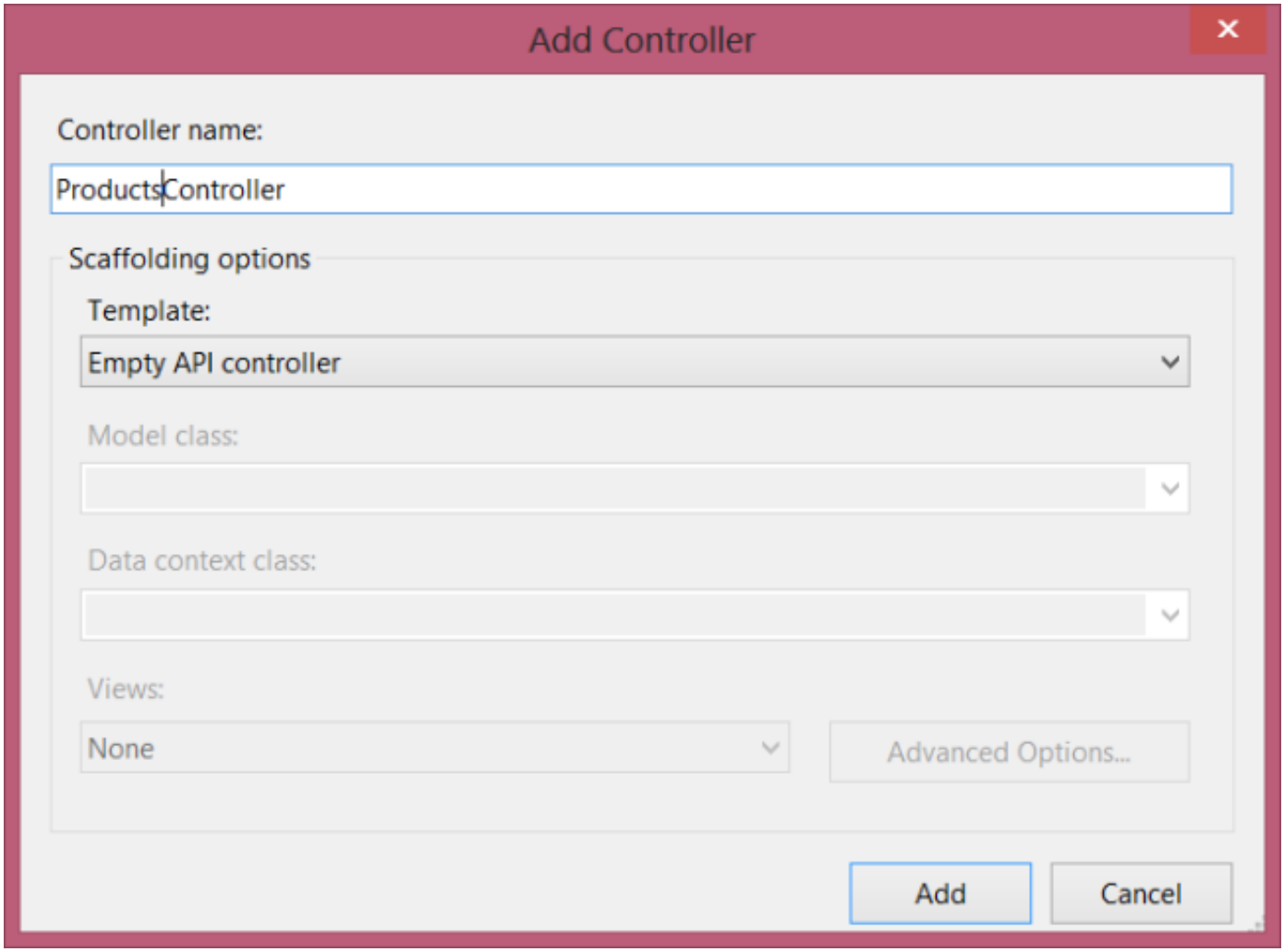
ValuesController یک کنترلر نمونه WebAPI است.

کنترلر ValuesController را حذف کنید، نیازی به این آیتم نخواهیم داشت. حال برای اضافه کردن کنترلری جدید مراحل زیر را دنبال کنید.

در پنجره Solution Explorer روی پوشه Controllers کلیک راست کرده و گزینه Add, Controller را انتخاب کنید.



در دیالوگ Add Controller نام کنترلر را به ProductsController تغییر داده و در قسمت Scaffolding Options گزینه Empty API Controller را انتخاب کنید.



حال فایل کنترلر جدید را باز کنید و عبارت زیر را به بالای آن اضافه نمایید.

```
using ProductStore.Models;
```

یک فیلد هم برای نگهداری وهله ای از IProductRepository اضافه کنید.

```
public class ProductsController : ApiController
{
    static readonly IProductRepository repository = new ProductRepository();
}
```

فراخوانی new ProductRepository() طراحی جالبی نیست، چرا که کنترلر را به پیاده سازی بخصوصی از این اینترفیس گره می‌زند. بهتر است از تزریق وابستگی (Dependency Injection) استفاده کنید. برای اطلاعات بیشتر درباره تکنیک DI در Web API به این لینک مراجعه کنید.

گرفتن منابع

API ProductStore اکشن‌های متعددی در قالب متدهای HTTP GET در دسترس قرار می‌دهد. هر اکشن به متدی در کلاس ProductsController مرتبط است.

Action	HTTP Method	Relative URI
دریافت لیست تمام محصولات	GET	api/products/
دریافت محصولی مشخص بر اساس شناسه	GET	api/products/ *id* /
دریافت محصولات بر اساس طبقه بندی	GET	api/products?category= *category* /

برای دریافت لیست تمام محصولات متد زیر را به کلاس ProductsController اضافه کنید.

```
public class ProductsController : ApiController
{
    public IEnumerable<Product> GetAllProducts()
    {
        return repository.GetAll();
    }
    // ....
}
```

نام این متد با "Get" شروع می‌شود، پس بر اساس قراردادهای توکار پیش فرض به درخواست‌های HTTP GET نگاشت خواهد شد. همچنین از آنجا که این متد پارامتری ندارد، به URI ای نگاشت می‌شود که هیچ قسمتی با نام مثلا *id* نداشته باشد.

برای دریافت محصولی مشخص بر اساس شناسه آن متد زیر را اضافه کنید.

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

نام این متد هم با "Get" شروع می‌شود اما پارامتری با نام *id* دارد. این پارامتر به قسمت *id* مسیر درخواست شده (request URI) نگاشت می‌شود. تبدیل پارامتر به نوع داده مناسب (در اینجا int) هم بصورت خودکار توسط فریم ورک ASP.NET Web API انجام می‌شود.

متد GetProduct در صورت نامعتبر بودن پارامتر id استثنایی از نوع HttpResponseException تولید می‌کند. این استثنا بصورت خودکار توسط فریم ورک Web API به خطای 404 (Not Found) ترجمه می‌شود.

در آخر متدی برای دریافت محصولات بر اساس طبقه بندی اضافه کنید.

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

اگر آدرس درخواستی پارامترهای query string داشته باشد، Web API سعی می‌کند پارامترها را با پارامترهای متد کنترلر تطبیق دهد. بنابراین درخواستی به آدرس "api/products?category= *category* " به این متد نگاشت می‌شود.

ایجاد منبع جدید

قدم بعدی افزودن متدی به ProductsController برای ایجاد یک محصول جدید است. لیست زیر پیاده سازی ساده ای از این متد را نشان می‌دهد.

```
// Not the final implementation!
public Product PostProduct(Product item)
{
    item = repository.Add(item);
    return item;
}
```

به دو چیز درباره این متد توجه کنید:

نام این متد با "Post" شروع می‌شود. برای ساختن محصولی جدید کلاینت یک درخواست HTTP POST ارسال می‌کند.
این متد پارامتری از نوع Product می‌پذیرد. در Web API پارامترهای پیچیده (complex types) بصورت خودکار با deserialize کردن بدنه درخواست بدست می‌آیند. بنابراین در اینجا از کلاینت انتظار داریم که آبجکتی از نوع Product را با فرمت XML یا JSON ارسال کند.

پیاده سازی فعلی این متد کار می‌کند، اما هنوز کامل نیست. در حالت ایده آل ما می‌خواهیم پیام HTTP Response موارد زیر را هم در بر گیرد:
Response code: بصورت پیش فرض فریم ورک Web API کد وضعیت را به 200 (OK) تنظیم می‌کند. اما طبق پروتکل HTTP/1.1 هنگامی که یک درخواست POST منجر به ساخته شدن منبعی جدید می‌شود، سرور باید با کد وضعیت 201 (Created) پاسخ دهد.
Location: هنگامی که سرور منبع جدیدی می‌سازد، باید آدرس منبع جدید را در قسمت Location header پاسخ درج کند.

ASP.NET Web API دستکاری پیام HTTP response را آسان می‌کند. لیست زیر پیاده سازی بهتری از این متد را نشان می‌دهد.

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

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

توجه کنید که حالا نوع بازگشتی این متد HttpResponseMessage است. با بازگشت دادن این نوع داده بجای Product، می‌توانیم جزئیات پیام HTTP response را کنترل کنیم. مانند تغییر کد وضعیت و مقدار دهی Location header.

متد CreateResponse آبجکتی از نوع HttpResponseMessage می‌سازد و بصورت خودکار آبجکت Product را مرتب (serialize) کرده و در بدنه پاسخ می‌نویسد. نکته دیگر آنکه مثال جاری، مدل را اعتبارسنجی نمی‌کند. برای اطلاعات بیشتر درباره اعتبارسنجی مدل‌ها در Web API به این لینک مراجعه کنید.

بروز رسانی یک منبع
بروز رسانی یک محصول با PUT ساده است.

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

نام این متد با "Put" شروع می‌شود، پس Web API آن را به درخواست‌های HTTP PUT نگاشت خواهد کرد. این متد دو پارامتر می‌پذیرد، یکی شناسه محصول مورد نظر و دیگری آبجکت محصول آپدیت شده. مقدار پارامتر *id* از مسیر (route) دریافت می‌شود و پارامتر محصول با deserialize کردن بدنه درخواست.

حذف یک منبع

برای حذف یک محصول متد زیر را به کلاس ProductsController اضافه کنید.

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

    repository.Remove(id);
}
```

اگر یک درخواست DELETE با موفقیت انجام شود، می‌تواند کد وضعیت 200 (OK) را بهمراه بدنه موجودیتی که وضعیت فعلی را نمایش می‌دهد برگرداند. اگر عملیات حذف هنوز در حال اجرا است (Pending) می‌توانید کد 202 (Accepted) یا 204 (No Content) را برگردانید.

در مثال جاری متد DeleteProduct نوع void را بر می‌گرداند، که فریم ورک Web API آن را بصورت خودکار به کد وضعیت 204 (No Content) ترجمه می‌کند.

نظرات خوانندگان

نویسنده: علي
تاریخ: ۱۳۹۲/۱۱/۰۳ ۱۳:۴۸

با سلام و تشکر از شما. در حالت post اگر اطلاعات را به شکل زیر ارسال کنیم، item یا مدل دریافت شده در متد PostProduct نال هست. چرا؟

```
$.post('api/products', JSON.stringify({Id: 1, Name: "name", Category: "test", Price: 1 }));
```

نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۱/۰۴ ۱۷:۵۳

باید نوع داده ارسالی رو مشخص کنید، بعنوان مثال:

```
function postProduct() {
    var product = { Name: "SampleProduct", Category: "TestCategory", Price: 10.99 };

    $.ajax({
        type: 'POST',
        data: JSON.stringify(product),
        url: "/api/products",
        contentType: "application/json"
    }).done(function (data) {
        var message = data.Name + ' $:' + data.Price;
        alert(message);
    });
}
```

مطالعه بیشتر

Parameter for POST Web API 4 method null when called from Fiddler with JSON body

How to pass json POST data to Web API method as object

how to post arbitrary json object to webapi

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۱۱/۰۵ ۱۷:۳۶

ویژگی FromBody رو هم باید به تک پارامتری که تعریف می‌کنید، اضافه کنید.

عنوان: **استفاده از EF در اپلیکیشن های N-Tier : قسمت چهارم**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۱/۰۹ ۸:۴۰
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, Tips, ASP.NET Web API, N-Layer Architecture, Entity Framework 6, ASP.NET Web API 2

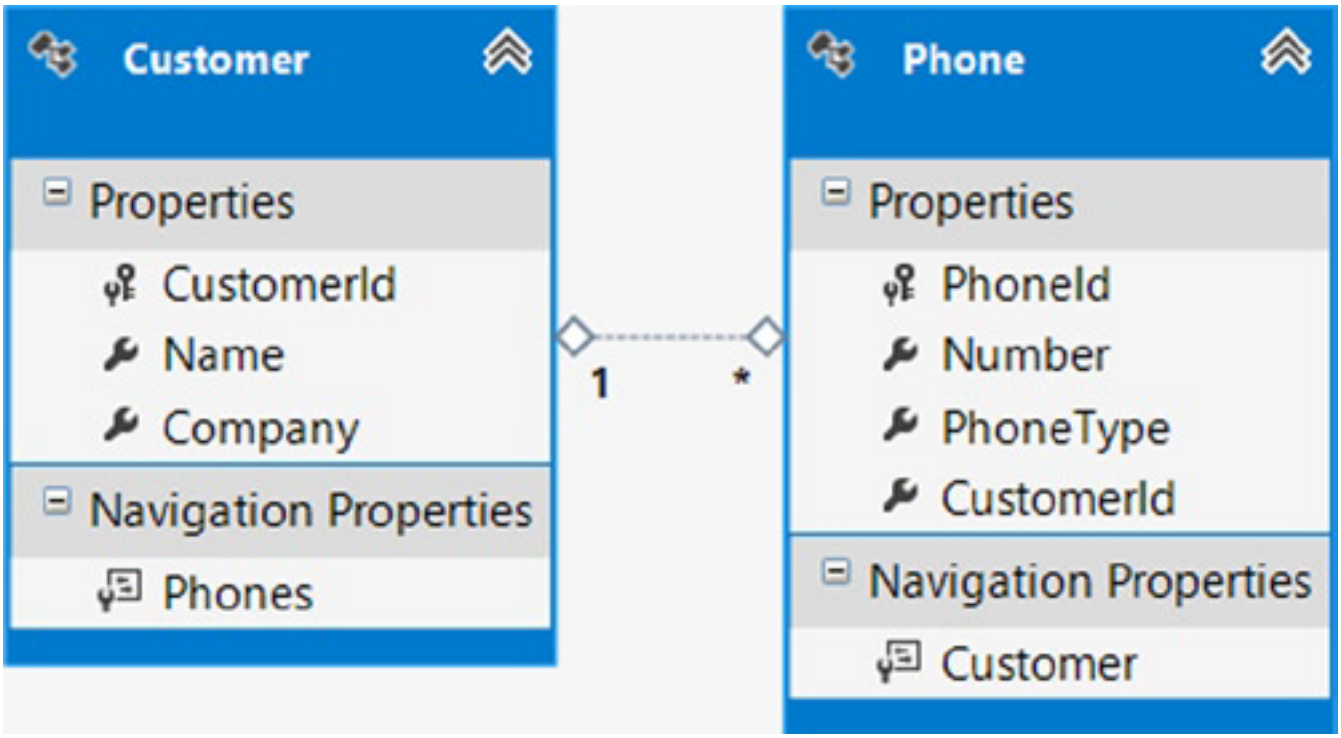
در قسمت قبل تشخیص تغییرات توسط Web API را بررسی کردیم. در این قسمت نگاهی به پیاده سازی Change-tracking در سمت کلاینت خواهیم داشت.

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 ویژوال 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 مستقیما 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 Compatibilty
    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 orginal 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 'Modifed' 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 کلاس‌های Customer, Phone و BaseEntity را به پروژه کلاینت اضافه کنید. چنین کدهایی بهتر است در لایه مجزایی قرار گیرند و بین لایه‌های 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 می‌کند. در این مرحله سایت در حال اجرا است و سرویس‌ها 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 می‌کنیم و از سرویس درخواست می‌کنیم 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 می‌کنیم. این فراخوانی تمام داده‌های پیشین را حذف می‌کند.

در قدم بعدی یک مشتری بهمراه دو شماره تماس می‌سازیم. 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 موجودیت هایی 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 نهایتا 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 قابلیت‌های async نیست، اما توصیه میشه در مواقعی که امکانش هست و مناسب است از این قابلیت استفاده کنید. لزوما کارایی (performance) بهتری بدست نمیارید ولی مسلما تجربه کاربری بهتری خواهید داشت. عملیاتی که بصورت async اجرا میشن ریسمان جاری (current thread) رو قفل نمی‌کنند، بنابراین اجرای اپلیکیشن ادامه پیدا می‌کنه و پاسخگویی بهتری بدست میارید. برای مطالعه بیشتر به این لینک مراجعه کنید.

مطالعه بیشتر

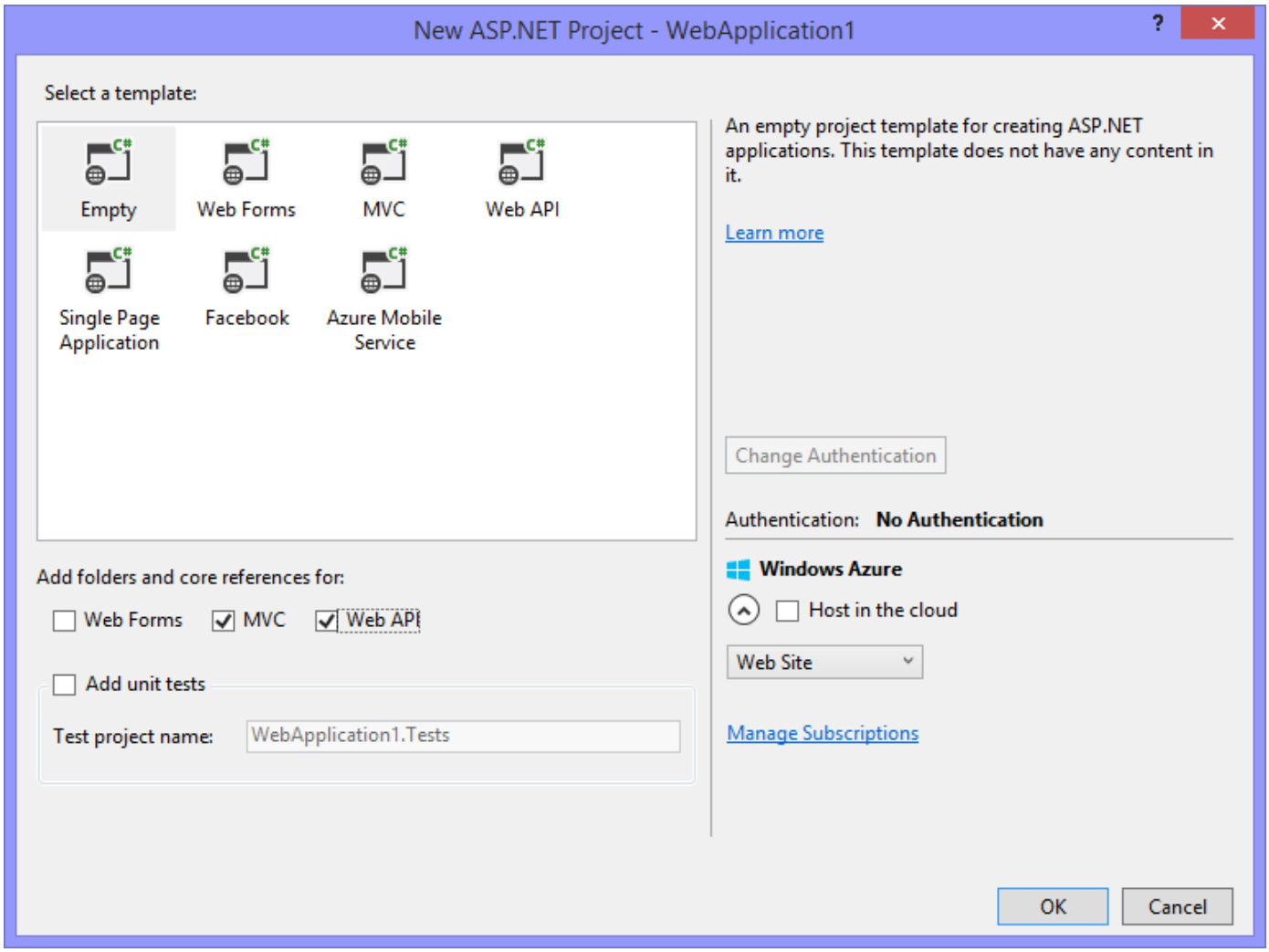
Using Asynchronous Methods in ASP.NET 4.5

Async and Await

عنوان: **ارسال ویدیو بصورت Async توسط Web Api**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۳/۰۵/۲۶ ۱۵:۱۰
آدرس: www.dotnettips.info
گروه‌ها: MVC, ASP.NET Web API, Asynchronous Programming, ASP.NET Web API 2, Network Programming

فریم ورک ASP.NET Web API صرفا 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,
        HttpContent 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 (مجددا 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 (مثلا 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. مثلا 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, HttpContent,
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 نهایتا 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 وهله جدیدی از کلاس VideoStream می‌سازیم و مسیر کامل فایل را به آن پاس می‌دهیم.

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

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

```
var response = Request.CreateResponse();

response.Content = new PushStreamContent((Action<Stream, HttpContent,
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 Headerها را تعریف نکنید سایز فایل دریافتی و مدت زمان آن نامعلوم خواهد بود که تجربه کاربری خوبی بدست نمی‌دهد. نهایتا 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 bravity
}
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 فایلی 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 مرورگرتان استفاده کنید. مثلا در گوگل کروم F12 را بزنید و به قسمت Network بروید. صفحه را یکبار مجددا 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 فایلی 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: ۱۳۹۳/۰۶/۱۰ ۱۷:۵۵

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

عنوان: **افزودن خودکار کلاس های WebAPI و SignalR Hub به برنامه در حالت SelfHost**
نویسنده: ابوالفضل رجب پور
تاریخ: ۱۳۹۳/۰۸/۰۵ ۱۳:۵۵
آدرس: www.dotnettips.info
گروه‌ها: SignalR, ASP.NET Web API, Self-Hosting, ASP.NET Web API 2

یکی از گزینه‌های میزبانی WebAPI و SignalR حالت SelfHost می‌باشد که روش آن قبلا در مطلب « نگاهی به گزینه‌های 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 selfHsot کردن از یک برنامه‌ی کنسول استفاده می‌کنیم:

```
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 libraryهای 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:\r\n{\r\n  Date: Mon, 27 Oct 2014 09:36:48 GMT\r\n  Server:
Microsoft-HTTPAPI/2.0\r\n  Content-Length: 0\r\n}"}
```

مشکل چیست؟
همانطور که در مطلب « نگاهی به گزینه‌های مختلف مهیای 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.

عنوان: **Http Batch Processing در Asp.Net Web Api**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۳/۱۱/۱۶ ۲۳:۴۵
آدرس: www.dotnettips.info
گروه‌ها: ASP.NET Web API, OData, Performance, ASP.NET Web API 2, Batch Request

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 کارآیی 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 HttpMessageContent(new HttpRequestMessage(HttpMethod.Post, baseAddress +
"/api/Book/Add")
                    {
                        Content = new ObjectContent<string>("myBook", new JsonMediaTypeFormatter())
                    }),
                    new HttpMessageContent(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 وهله 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 وهله 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 درخواست ها

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. بعنی 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.