عنوان: **فعال‌سازی استفاده از Session در ASP.NET MVC 4 API Controller ها**
نویسنده: بهمن خلفی
تاریخ: ۱۳۹۲/۰۷/۱۷ ۲۱:۳۰
آدرس: www.dotnettips.info
گروه‌ها: MVC, Session, ASP.Net MVC 4, routing, Web API

در پروژه فروشگاهی تحت Asp.Net MVC 4 بهترین روش برای ایجاد یک سبد خرید استفاده از یک Api Controller می‌باشد.

من در پروژه‌ای که در MVC 3 داشتم این مورد را بدین شکل انجام داده بودم که با ایجاد یک کلاس و درج چند سطر کد در Global.asax این مورد حل میشد و در Api Controller ای که میخواستم اطلاعات را در آن به کمک Session دریافت یا ویرایش کنم، امکان دسترسی به Session را داشتم:
یک کلاس ایجاد کرده و کدهای زیر را در داخل آن درج میکنیم:

```
//using System.Web.Http.WebHost;
//using System.Web.Routing;
//using System.Web.SessionState;


public class MyHttpControllerHandler : HttpControllerHandler, IRequiresSessionState
    {
        public MyHttpControllerHandler(RouteData routeData): base(routeData)
        {
        }
    }
    public class MyHttpControllerRouteHandler : HttpControllerRouteHandler
    {
        protected override IHttpHandler GetHttpHandler(RequestContext requestContext)
        {
            return new MyHttpControllerHandler(requestContext.RouteData);
        }
    }
```

و برای اینکه این مورد، در برنامه‌ای که ساختیم کار کند باید در Global.asax این کدها را درج کنیم (دوباره تاکید می‌کنم این نکته فقط در MVC 3 کار می‌کند)

```
public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

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

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
            );

        }
```

با تنظیمات فوق در برنامه‌های Asp.NET MVC 3 امکان استفاده از Session در API Controllerها در دسترس قرار میگیرد.
اما این کار در ASP.NET MVC 4 قابل استفاده نیست و خیلی بهتر و راحتتر انجام می‌شود (میتوانید به WebApiConfig.cs در فولدر App_Start پروژه مراجعه نمائید)
حال چه تنظیماتی نیاز است تا دوباره این امکان برقرار شود؟
تنها کدهای ذیل را در Global.asax درج نمائید و امکان استفاده از Session را در API Controllerهای MVC4، ایجاد نمائید.

```
public class MvcApplication : System.Web.HttpApplication
{
    public override void Init()
    {
        this.PostAuthenticateRequest += MvcApplication_PostAuthenticateRequest;
```

```
        base.Init();
    }

    void MvcApplication_PostAuthenticateRequest(object sender, EventArgs e)
    {
        System.Web.HttpContext.Current.SetSessionStateBehavior(
        SessionStateBehavior.Required);
    }

    protected void Application_Start()
    {
        AreaRegistration.RegisterAllAreas();

        WebApiConfig.Register(GlobalConfiguration.Configuration);
        FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);
        RouteConfig.RegisterRoutes(RouteTable.Routes);
        BundleConfig.RegisterBundles(BundleTable.Bundles);
        AuthConfig.RegisterAuth();
    }
}
```

نظرات خوانندگان

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۷/۱۷ ۲۱:۳۷

لطفا این قسمت « بهترین روش برای ایجاد یک سبد خرید استفاده از یک Api Controller می‌باشد.» را بیشتر توضیح بدید. با تشکر.

نویسنده: بهمن خلفی
تاریخ: ۱۳۹۲/۰۷/۱۷ ۲۱:۴۶

در اکثر فروشگاههایی که با Asp.net MVC توسعه پیدا کردند اضافه کردن یک کالا به سبد خرید پروسه ای زمان بر بوده و کاربر پسند نیست در حالی که در فروشگاههای متن باز مشابه این عمل بصورت زیبا و کاملا پرسرعت انجام میشود . API یکی از مباحث خوبی است که در MVC براحتی قابل استفاده بوده و این قدرت را به برنامه نویس میده که بتواند از مباحثی مانند Ajax یا JSON مثلا در سبد کاربر استفاده کند.
بهترین روش به نظر من است ! بدلیل اینکه بسیار راحت و امن است . راه اندازی آن در حد 10 الی 15 دقیقه بیشتر طول نمیکشد و شما میتوانید تمامی مباحثی مانند احراز هویت و ... را طبق اصول MVC روی همه Actionهای مورد نیاز اعمال کنید.
اگر اغراق شده فقط یک نظر شخصی است. ;-)

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۷/۱۷ ۲۱:۴۹

ممنون. JSON و Ajax با کنترلرهای معمولی MVC هم قابل دسترسی است. چه مزیت بیشتری رو با Web API به دست آوردید؟

نویسنده: بهمن خلفی
تاریخ: ۱۳۹۲/۰۷/۱۷ ۲۱:۵۷

بله این امکان وجود دارد اما شما این امکان را میتوانید در یک برنامه بصورت کلی استفاده نمائید یعنی در هر جای برنامه باشید میتوانید به این بخش دسترسی داشته باشید .
از طرفی شما با استفاده از API دیگر برخی از اطلاعات یا آیتم‌های صفحه را بدلیل Post شدن صفحه (Dispose) از دست نخواهید داد.
استفاده از Web APIها یک عمل عمومی بحساب می‌آید اما در این مطلب مقصود دسترسی به Session در API Controllerها میباشد.

نویسنده: مهدیار
تاریخ: ۱۳۹۲/۰۷/۱۹ ۱۹:۲۲

استفاده از session ؟
به نظرم یک جدول موقت برای کارت می‌گرفتید بهتر بود . این هم یک مطلب مفید برای استفاده نکردن از session.

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۷/۱۹ ۲۱:۲۶

کوکی هم برای یک طراحی Http friendly خوب هست. خصوصا اینکه یک شخص آنچنان اقلام زیادی را که هربار خرید نمی‌کند.
همچنین HTML5 local storage نیز مفید است.

نویسنده: بهمن خلفی
تاریخ: ۱۳۹۲/۰۷/۲۰ ۸:۵۷

بله دوست عزیز ، استفاده از Session مزایا و معایب خودش را به همراه دارد.

اما در یک سیستم فروشگاهی حذف و اضافه شدن کالا در سبد خرید بصورت مکرر انجام میشود و برای اینکه شما چه کاربران مهمان و چه کاربران عضو را مدیریت نموده و بتوانید ترافیک روی بانک اطلاعاتی خود را کاهش دهید میتوانید از Session استفاده نمایید . (بطور مثال اگر 1000 نفر دریک سایت فروشگاهی که حداقل 100 کالا ارائه شده است را به سبد خرید خود اضافه یا تعداد و ... آنها را ویرایش نمایند در بهترین حال 100000 درخواست به بانک اطلاعاتی ارسال و دریافت خواهد شد)

نویسنده: بهمن خلفی
تاریخ: ۱۳۹۲/۰۷/۲۰ ۸:۵۹

بله دوست عزیز ، این هم یک امکان بسیار خوبی است که در سمت کاربر میتوان ازش استفاده نمود.
در مطالب آینده سعی خواهد شد کاربرد این امکان در یک برنامه کاربردی ارائه شود. باتشکر

عنوان: **واکشی اطلاعات سرویس Web Api با استفاده از TypeScript و AngularJs**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۱۰/۰۹ ۸:۴۵
آدرس: www.dotnettips.info
گروه‌ها: TypeScript, AngularJS, Web API

در پست‌های قبلی با AngularJs ، TypeScript و Web Api آشنا شدید. در این پست قصد دارم از ترکیب این موارد برای پیاده سازی عملیات واکشی اطلاعات سرویس Web Api در قالب یک پروژه استفاده نمایم. برای شروع ابتدا یک پروژه Asp.Net MVC ایجاد کنید.
در قسمت مدل ابتدا یک کلاس پایه برای مدل ایجاد خواهیم کرد:

```
public abstract class Entity
    {
        public Guid Id { get; set; }
    }
```

حال کلاسی به نام Book ایجاد می‌کنیم:

```
public class Book : EntityBase
    {
        public string Name { get; set; }
        public decimal Author { get; set; }
    }
```

در پوشه مدل یک کلاسی به نام BookRepository ایجاد کنید و کدهای زیر را در آن کپی نمایید(به جای پیاده سازی بر روی بانک اطلاعاتی، عملیات بر روی لیست درون حافظه انجام می‌گیرد):

```
public class BookRepository
    {
        private readonly ConcurrentDictionary<Guid, Book> result = new ConcurrentDictionary<Guid,
Book>();

        public IQueryable<Book> GetAll()
        {
            return result.Values.AsQueryable();
        }

        public Book Add(Book entity)
        {
            if (entity.Id == Guid.Empty) entity.Id = Guid.NewGuid();

            if (result.ContainsKey(entity.Id)) return null;

            if (!result.TryAdd(entity.Id, entity)) return null;

            return entity;
        }
    }
```

نوبت به کلاس کنترلر می‌رسد. یک کنترلر Api به نام BooksController ایجاد کنید و سپس کدهای زیر را در آن کپی نمایید:

```
public class BooksController : ApiController
    {
        public static BookRepository repository = new BookRepository();

public BooksController()
        {
            repository.Add(new Book
            {
                Id=Guid.NewGuid(),
                Name="C#",
                Author="Masoud Pakdel"
            });

            repository.Add(new Book
```

```
            {
                Id = Guid.NewGuid(),
                Name = "F#",
                Author = "Masoud Pakdel"
            });

            repository.Add(new Book
            {
                Id = Guid.NewGuid(),
                Name = "TypeScript",
                Author = "Masoud Pakdel"
            });
        }

        public IEnumerable<Book> Get()
        {
            return repository.GetAll().ToArray();
        }
    }
```

در این کنترلر، اکشنی به نام Get داریم که در آن اطلاعات کتاب‌ها از Repository مربوطه برگشت داده خواهد شد. در سازنده این کنترلر ابتدا سه کتاب به صورت پیش فرض اضافه می‌شود و انتظار داریم که بعد از اجرای برنامه، لیست مورد نظر را مشاهده نماییم.

حال نوبت به عملیات سمت کلاینت میرسد. برای استفاده از قابلیت‌های TypeScript و AngularJs در Vs.Net از این مقاله کمک بگیرید. بعد از آماده سازی در فولدر script، پوشه ای به نام app می‌سازیم و یک فایل TypeScript به نام BookModel در آن ایجاد می‌کنیم:

```
module Model {
    export class Book{
        Id: string;
        Name: string;
        Author: string;
    }
}
```

واضح است که ماژولی به نام Model داریم که در آن کلاسی به نام Book ایجاد شده است. برای انتقال اطلاعات از طریق سرویس $http در Angular نیاز به سریالایز کردن این کلاس به فرمت Json خواهیم داشت. قصد داریم View مورد نظر را به صورت زیر ایجاد نماییم:

```
<div ng-controller="Books.Controller">
        <table class="table table-striped table-hover" style="width: 500px;">
            <thead>
                <tr>
                    <th>Name</th>
                    <th>Author</th>
                </tr>
            </thead>
            <tbody>
                <tr ng-repeat="book in books">
                    <td>{{book.Name}}</td>
                    <td>{{book.Author}}</td>
                </tr>
            </tbody>
        </table>
    </div>
```

توضیح کدهای بالا:
ابتدا یک کنترلری که به نام Controller که در ماژولی به نام Book تعریف شده است باید ایجاد شود. اطلاعات تمام کتب ثبت شده باید از سرویس مورد نظر دریافت و با یک ng-repeat در جدول نمایش داده خواهند شود.
در پوشه app یک فایل TypeScript دیگر برای تعریف برخی نیازمندی‌ها به نام AngularModule ایجاد می‌کنیم که کد آن به صورت زیر خواهد بود:

```
declare module AngularModule {
    export interface HttpPromise {
        success(callback: Function) : HttpPromise;
    }
    export interface Http {
        get(url: string): HttpPromise;
    }
}
```

در این ماژول دو اینترفیس تعریف شده است. اولی به نام HttpPromise است که تابعی به نام success دارد. این تابع باید بعد از موفقیت آمیز بودن عملیات فراخوانی شود. ورودی آن از نوع Function است. بعنی اجازه تعریف یک تابع را به عنوان ورودی برای این توابع دارید.
در اینترفیس Http نیز تابعی به نام get تعریف شده است که برای دریافت اطلاعات از سرویس api، مورد استفاده قرار خواهد گرفت. از آن جا که تعریف توابع در اینترفیس فاقد بدنه است در نتیجه این جا فقط امضای توابع مشخص خواهد شد. پیاده سازی توابع به عهده کنترلرها خواهد بود:
مرحله بعد مربوط است به تعریف کنترلری به نام BookController تا اینترفیس بالا را پیاده سازی نماید. کدهای آن به صورت زیر خواهد بود:

```
/// <reference path='AngularModule.ts' />
/// <reference path='BookModel.ts' />

module Books {
    export interface Scope {
        books: Model.Book[];
    }

    export class Controller {
        private httpService: any;

        constructor($scope: Scope, $http: any) {
            this.httpService = $http;

            this.getAllBooks(function (data) {
                $scope.books = data;
            });
            var controller = this;
    }

        getAllBooks(successCallback: Function): void {
            this.httpService.get('/api/books').success(function (data, status) {
                successCallback(data);
            });
        }
    }
}
```

توضیح کدهای بالا:
برای دسترسی به تعاریف انجام شده در سایر ماژول‌ها باید ارجاعی به فایل تعاریف ماژول‌های مورد نظر داشته باشیم. در غیر این صورت هنگام استفاده از این ماژول‌ها با خطای کامپایلری روبرو خواهیم شد. عملیات ارجاع به صورت زیر است:

```
/// <reference path='AngularModule.ts' />
/// <reference path='BookModel.ts' />
```

در پست قبلی توضیح داده شد که برای مقید سازی عناصر بهتر است یک اینترفیس به نام Scope تعریف کنیم تا بتوانیم متغیرهای مورد نظر برای مقید سازی را در آن تعریف نماییم در این جا تعریف آن به صورت زیر است:

```
export interface Scope {
        books: Model.Book[];
    }
```

در این جا فقط نیاز به لیستی از کتاب‌ها داریم تا بتوان در جدول مورد نظر در View آنرا پیمایش کرد. تابعی به نام getAllBooks در کنترلر مورد نظر نوشته شده است که ورودی آن یک تابع خواهد بود که باید بعد از واکشی اطلاعات از سرویس، فراخوانی شود.
اگر به کدهای بالا دقت کنید می‌بینید که در ابتدا سازنده کنترلر،سرویس $http موجود در Angular به متغیری به نام httpService نسبت داده می‌شود. با فراخوانی تابع get و ارسال آدرس سرویس که با توجه به مقدار مسیر یابی پیش فرض کلاس WebApiConfig باید با api شروع شود به راحتی اطلاعات مورد نظر به دست خواهد آمد. بعد از واکشی در صورت موفقیت آمیز بودن عملیات تابع success اجرا می‌شود که نتیجه آن انتساب مقدار به دست آمده به متغیر books تعریف شده در $scope می‌باشد.

در نهایت خروجی به صورت زیر خواهد بود:

Name	Author
C#	Masoud Pakdel
TypeScript	Masoud Pakdel
F#	Masoud Pakdel

سورس پیاده سازی مثال بالا در Visual Studio 2013

نظرات خوانندگان

نویسنده: sadegh hp
تاریخ: ۱۳۹۲/۱۲/۲۳ ۱۱:۳۳

چجوری میشه با jasmine یک تست برای متدی که $http.post رو در یک سرویس انگولار پیاده کرده نوشت؟ تست متدهای async در انگولار چجوریه ؟

نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۱۲/۲۳ ۱۳:۱

angularJs کتابخانه ای برای mock آبجکت‌ها خود تهیه کرده است.(angular-mock) . از آن جا که در angular مبحث تزریق وابستگی بسیار زیبا پیاده سازی شده است با استفاده از این کتابخانه می‌توانید آبجکت‌های متناظر را mock کنید. برای مثال:

```
describe('myApp', function() {
var scope;

 beforeEach(angular.mock.module('myApp'));
 beforeEach(angular.mock.inject(function($rootScope) {
    scope = $rootScope.$new();
});
it('...')
});
```

هم چنین برای تست سرویس $http و شبیه سازی عملیات request و response در انگولار سرویس $httpBackend تعبیه شده است که یک پیاده سازی Fake از $http است که در تست‌ها می‌توان از آن استفاده کرد. برای مثال:

```
describe('Remote tests', function() {
    var $httpBackend, $rootScope, myService;
       beforeEach(inject(
function(_$httpBackend_, _$rootScope_, _myService_) {
      $httpBackend = _$httpBackend_;
      $rootScope = _$rootScope_;
      myService = _myService_;
}));
it('should make a request to the backend', function() {
   $httpBackend.expect('GET', '/v1/api/current_user')
     .respond(200, {userId: 123});
     myService.getCurrentUser();

     $httpBackend.flush();
});
});
```

دستور $httpBackend.expect برای ایجاد درخواست مورد نظر استفاده می‌شود که نوع verb را به عنوان آرگومان اول دریافت می‌کند. respond نیز مقدار بازگشتی مورد انتظار از سرویس مورد نظر را بر میگرداند. می‌توانید از دستورات زیر برای سایر حالات استفاده کنید:

- $httpBackend.expectGet
- $httpBackend.expectPut
- $httpBackend.expectPost
- $httpBackend.expectDelete
- $httpBackend.expectJson
- $httpBackend.expectHead
- $httpBackend.expectPatch

Flush کردن سرویس $httpBackend در پایان تست نیز برای همین مبحث async اجرا شدن سرویس‌های $backend است.

نویسنده: صادق اچ پی
تاریخ: ۱۳۹۲/۱۲/۲۵ ۹:۴۸

ممنون از پاسخ شما.
اما سوال بعد اینکه چرا اصلا باید بیرون از سرویس http رو ساخت؟ فرض کنید که ما دسترسی به محتوی متود درون سرویس نداریم و فقط میخواهیم اون رو صدا کنیم و ببینیم که متود درون سرویس درست کار میکنه یا نه! بدون اینکه بدونیم چجوری داخل متود پیاده سازی شده که در این مورد یک http.post یا get هست.

نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۱۲/۲۵ ۱۰:۴۳

$httpBackend یک پیاده سازی fake از $http است، در نتیجه می‌توانید در هنگام تست، این سرویس را به کنترلرهای خود تزریق کنید. اما قبل از DI باید برای این سرویس مشخص شود که برای مثال در هنگام مواجه شدن با یک درخواست از نوع Get و آدرس X چه خروجی برگشت داده شود. درست شبیه به رفتار mocking framework ها. فرض کنید شما کنترلری به شکل زیر دارید:

```
(function (module) {

    var myController = function ($scope, $http) {

        $http.get("/api/myData")
            .then(function (result) {
                $scope.data= result.data;
            });
    };

    module.controller("MyController",
        ["$scope", "$http", myController]);

}(angular.module("myApp")));
```

همان طور که می‌بینید در این کنترلر از $http استفاده شده است. حال برای تست آن می‌توان نوشت:

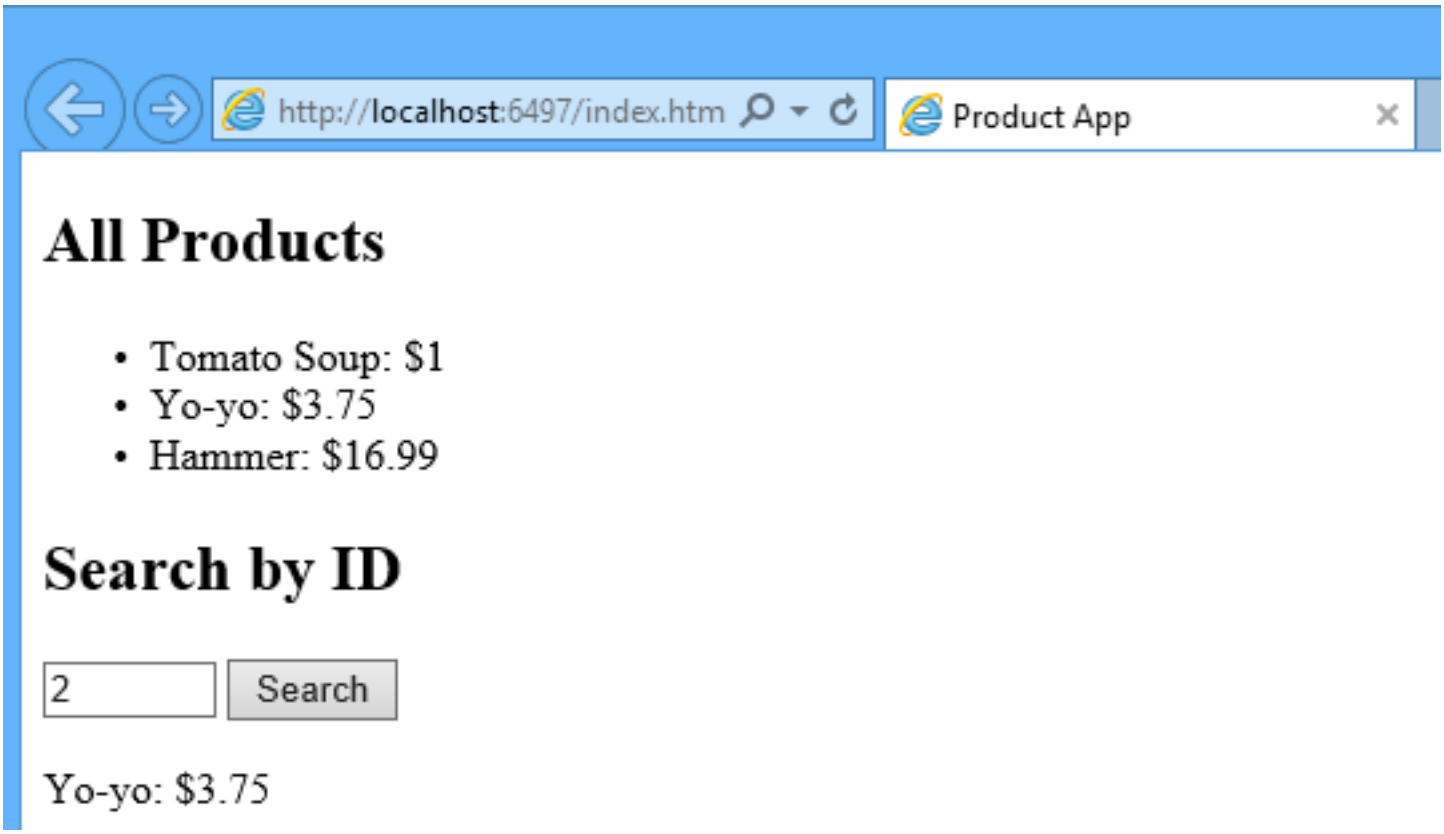
```
describe("myApp", function () {

    beforeEach(module('myApp'));

    describe("MyController", function () {

        var scope, httpBackend;
        beforeEach(inject(function ($rootScope, $controller, $httpBackend, $http) {
            scope = $rootScope.$new();
            httpBackend = $httpBackend;
            httpBackend.when("GET", "/api/myData").respond([{}, {}, {}]);
            $controller('MyController', {
                $scope: scope,
                $http: $http
            });
        }));

        it("should have 3 row", function () {
            httpBackend.flush();
            expect(scope.data.length).toBe(3);
        });
    });
});
```

httpBackend ساخته شده با استفاده از سرویس $controller به کنترلر مورد نظر تزریق می‌شود. حال اگر در یک کنترلر 5 بار از سرویس $http برای فراخوانی resource 5 متفاوت استفاده شده باشد باید برای هر حالت $httpBackend را طوری تنظیم کرد که بداند برای هر منبع چه خروجی در اختیار کنترلر قرار دهد.

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

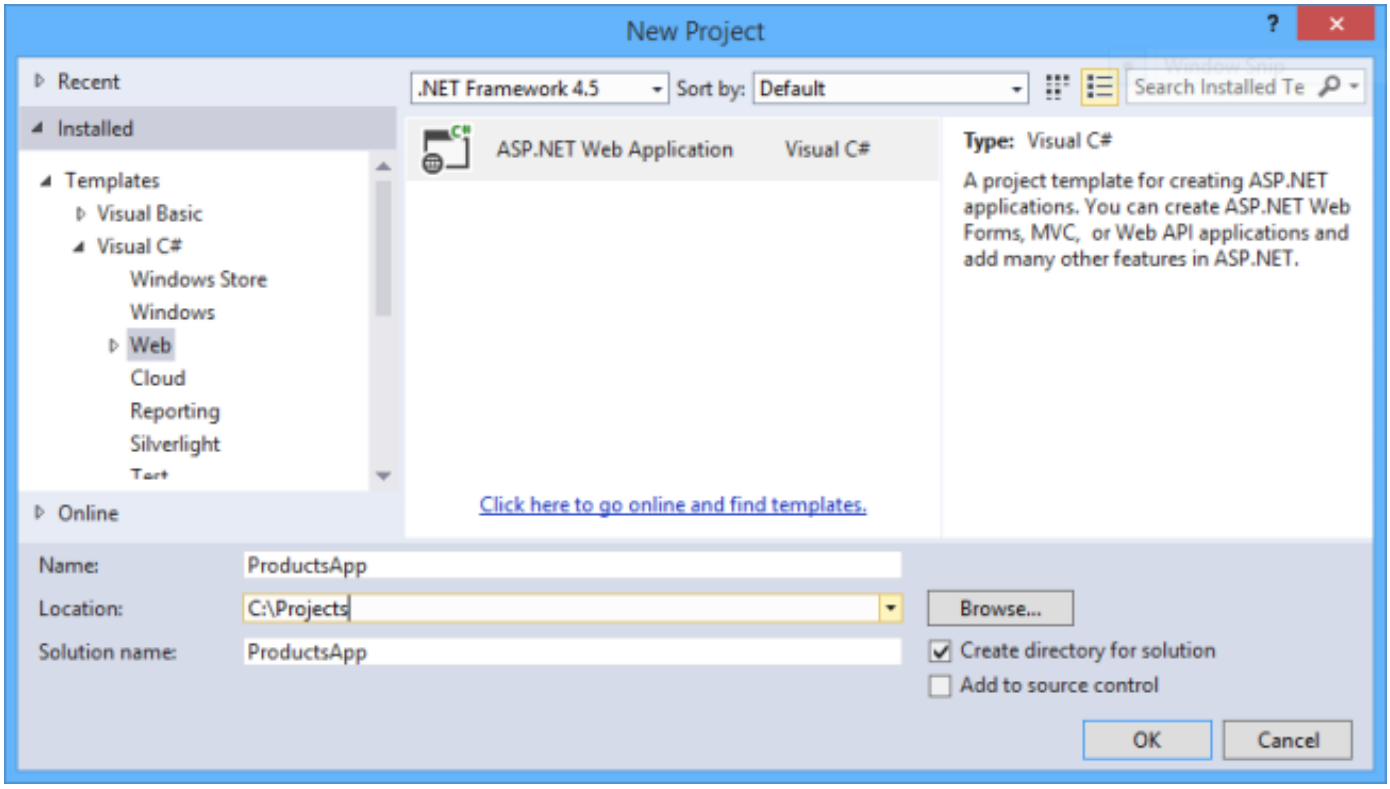
HTTP تنها برای به خدمت گرفتن صفحات وب نیست. این پروتکل همچنین پلتفرمی قدرتمند برای ساختن API هایی است که سرویس‌ها و داده را در دسترس قرار می‌دهند. این پروتکل ساده، انعطاف پذیر و در همه جا حاضر است. هر پلتفرمی که فکرش را بتوانید بکنید کتابخانه ای برای HTTP دارد، بنابراین سرویس‌های HTTP می‌توانند بازه 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.

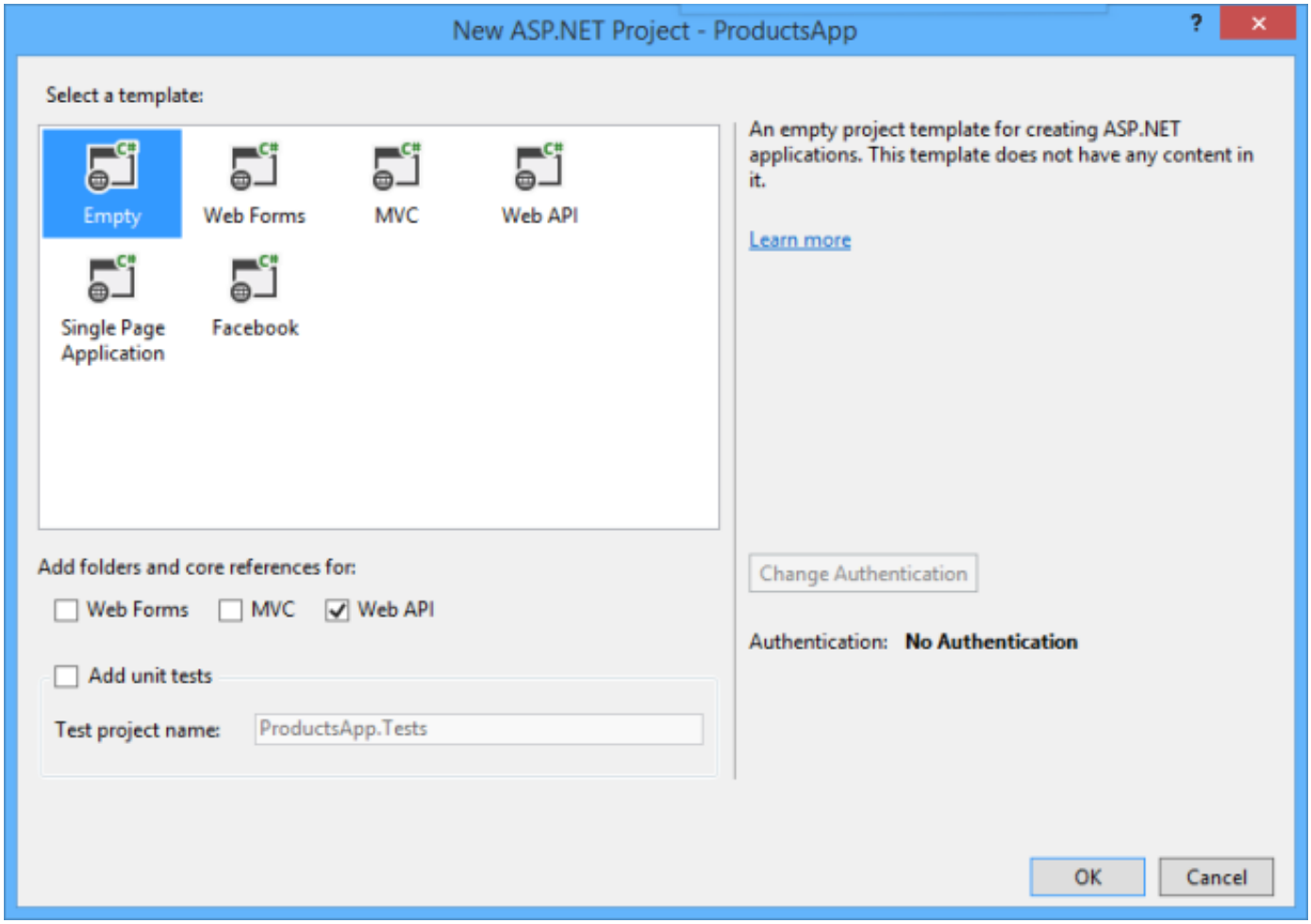


یک پروژه Web API بسازید

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.



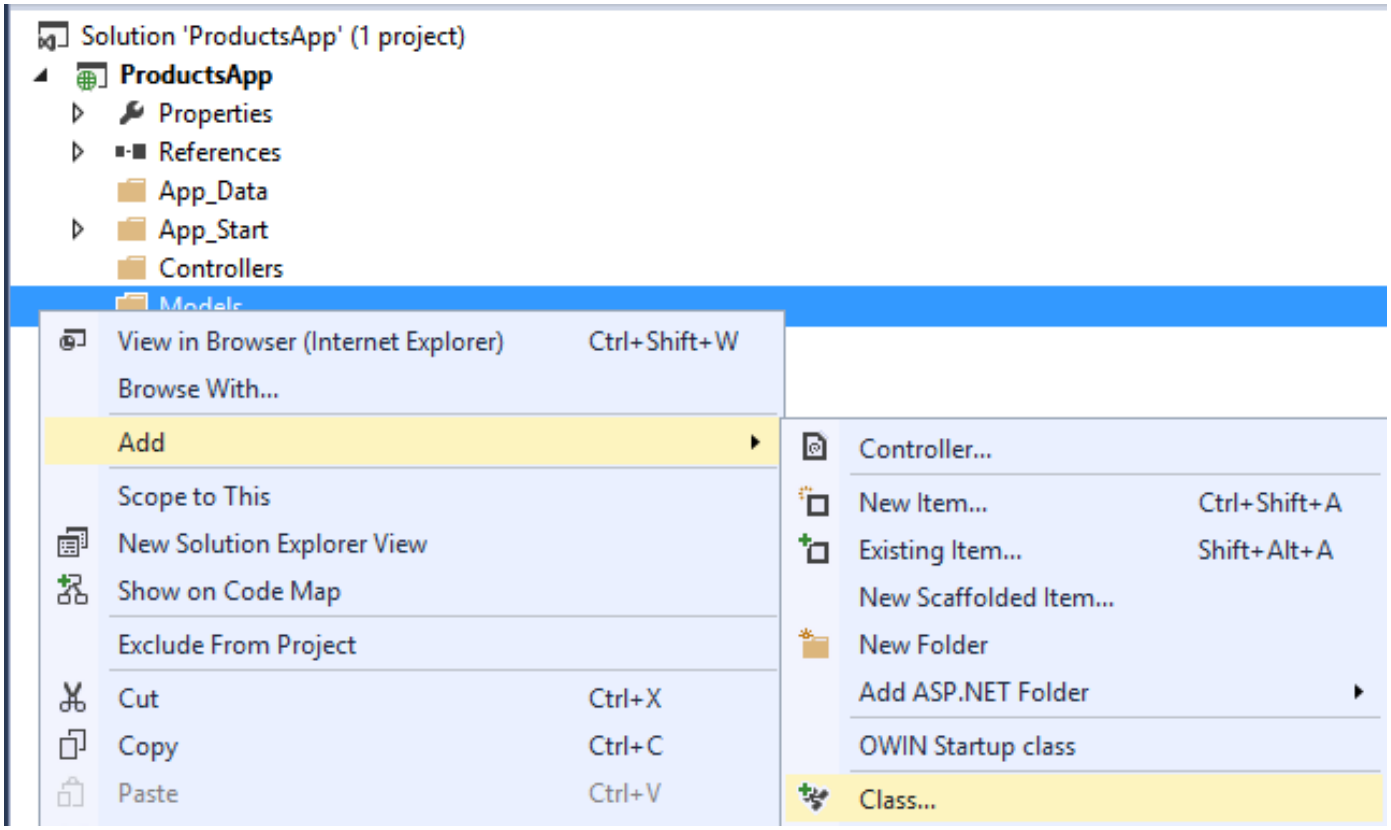
می توانید از قالب Web API هم استفاده کنید. این قالب با استفاده از ASP.NET MVC صفحات راهنمای API را خواهد ساخت. در این مقاله از قالب Empty استفاده میکنیم تا تمرکز اصلی، روی خود فریم ورک Web API باشد. بطور کلی برای استفاده از این فریم ورک لازم نیست با ASP.NET MVC آشنایی داشته باشید.

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.



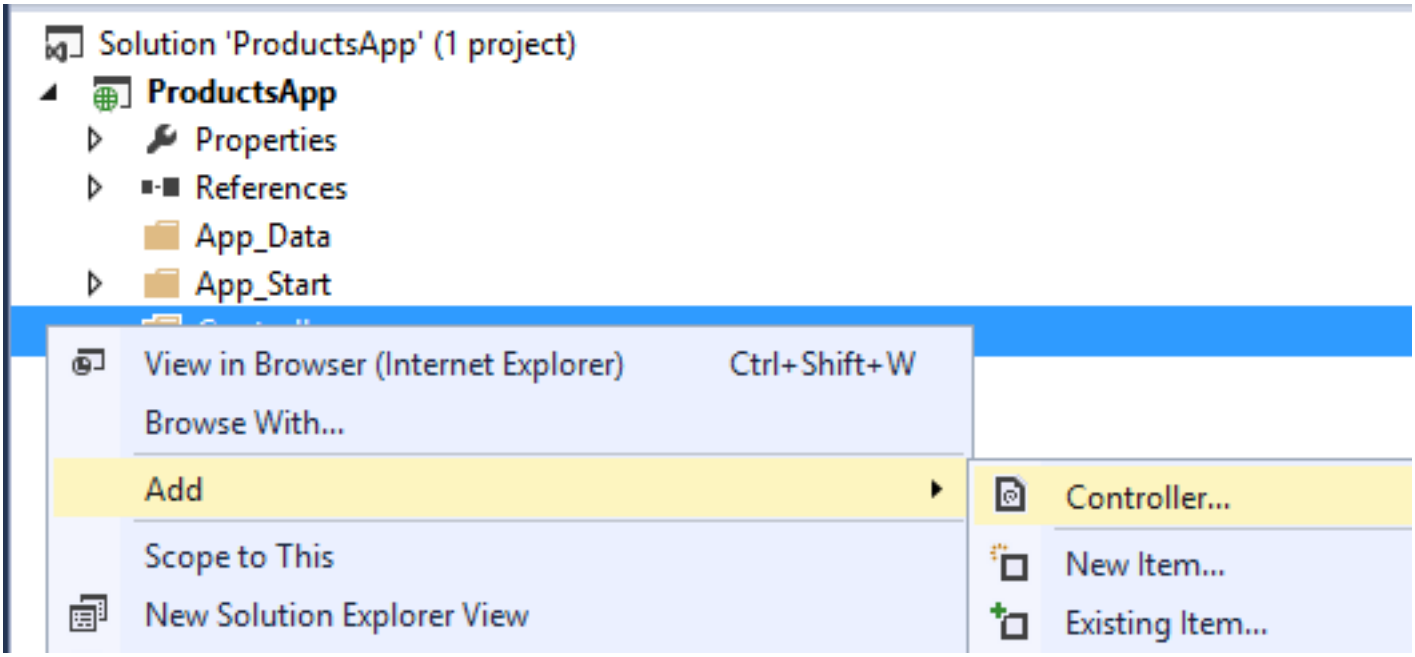
نام کلاس را به "Product" تغییر دهید، و خواص زیر را به آن اضافه کنید.

```
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; }
    }
}
```

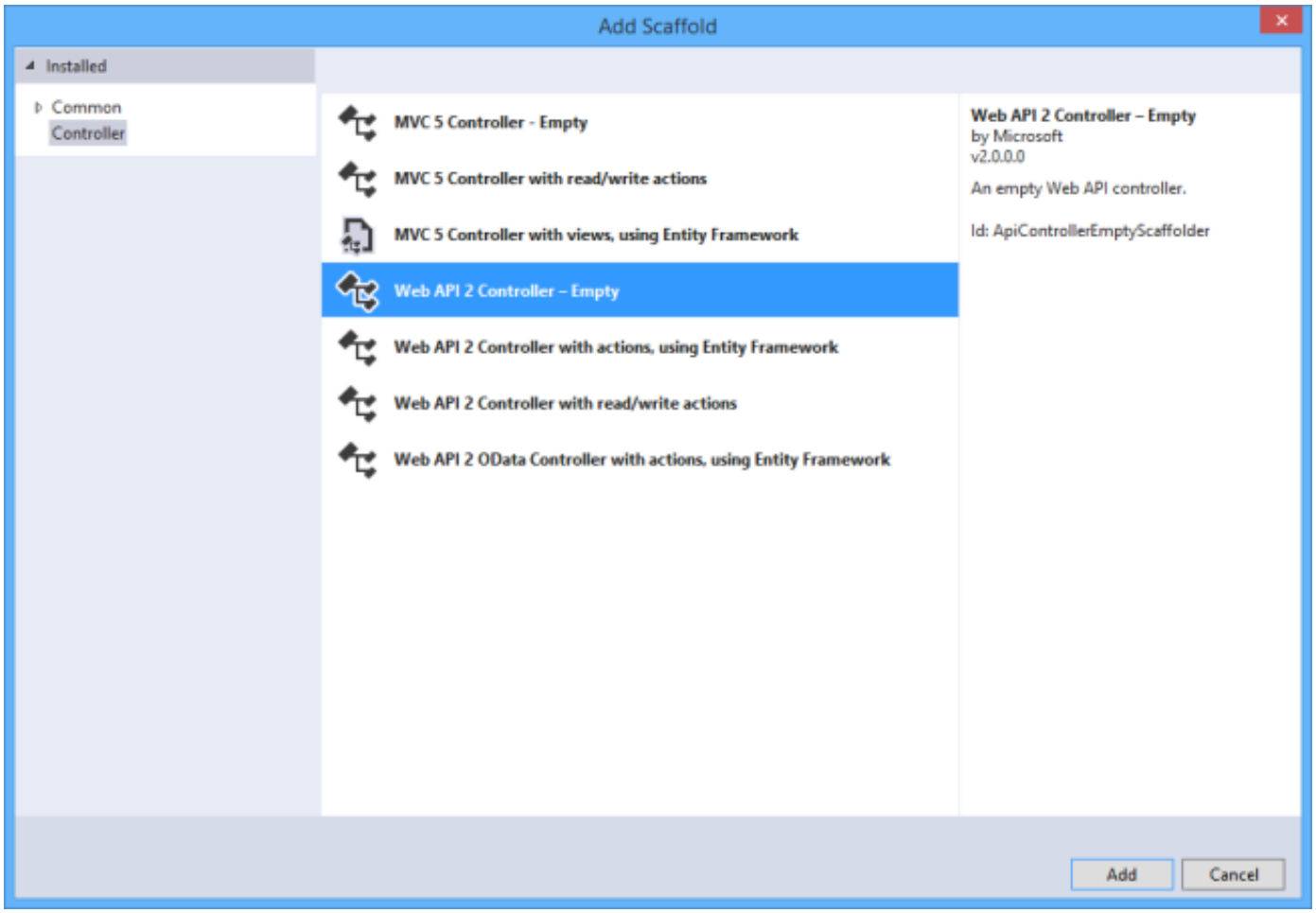
افزودن یک کنترلر

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.

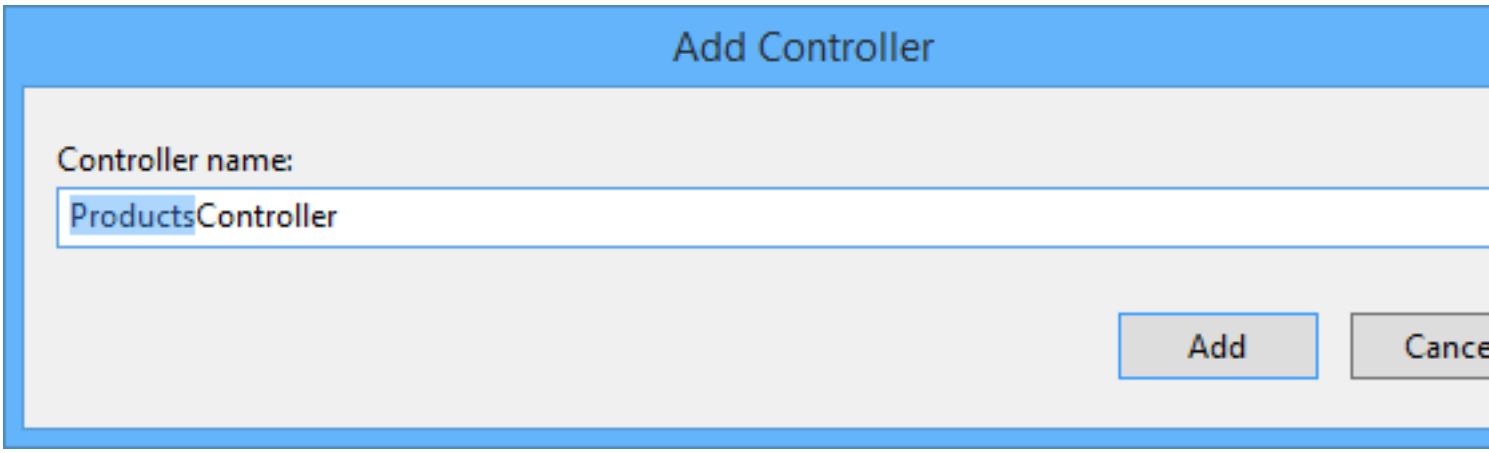
کنترلر جدیدی در پوشه Controllers ایجاد کنید.

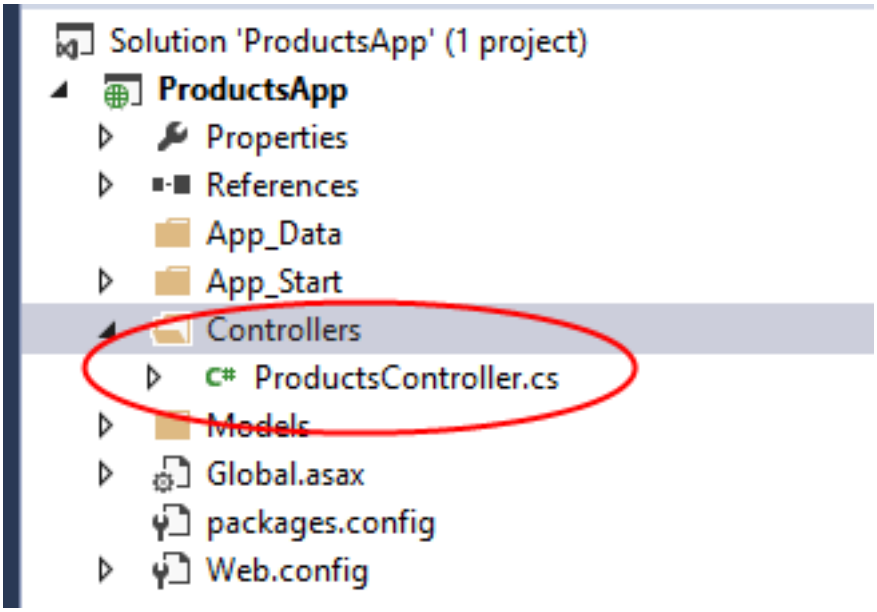


در دیالوگ Add Scaffold گزینه Web API Controller - Empty را انتخاب کرده و روی Add کلیک کنید.



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:

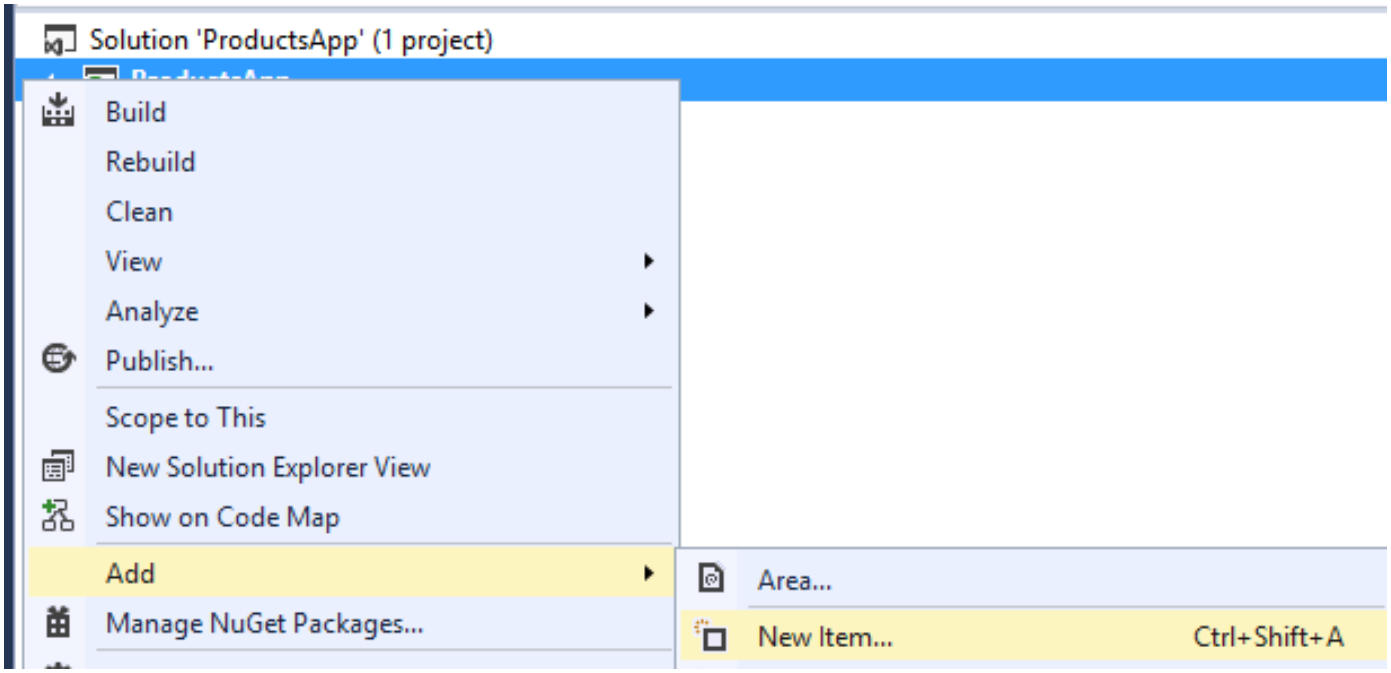
Controller Method	URI
GetAllProducts	api/products/
GetProductById	api/products/ *id* /

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

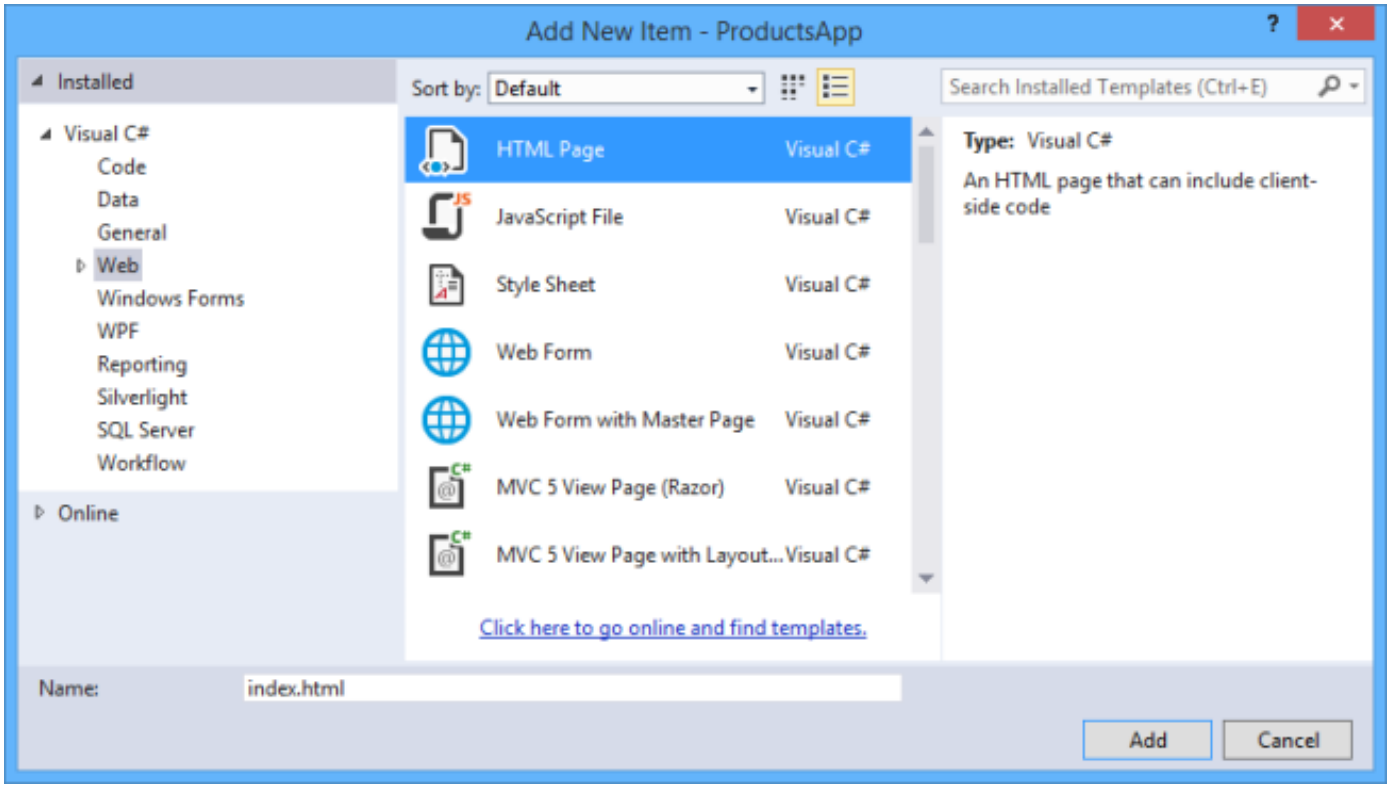
فراخوانی Web API با جاوا اسکریپت و jQuery

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

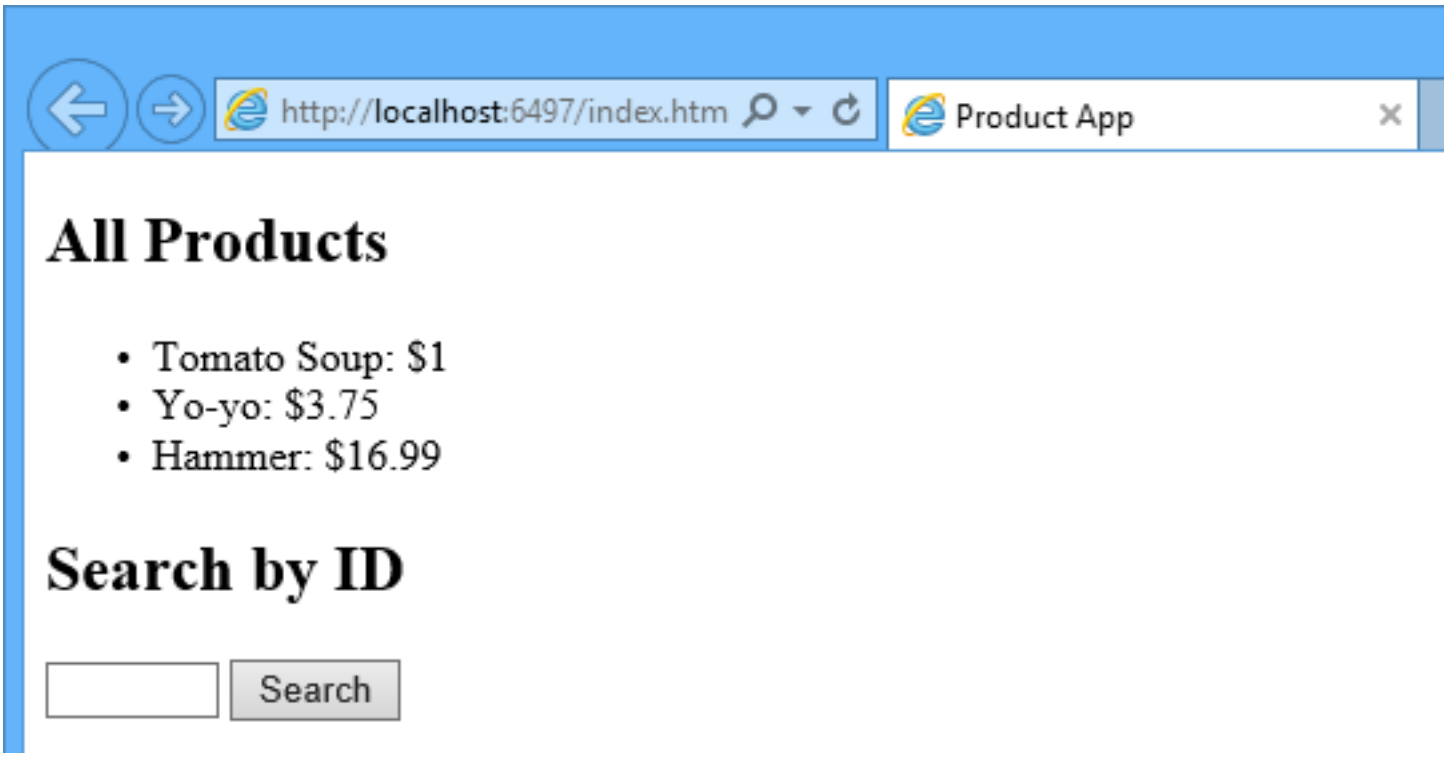
برای گرفتن یک محصول توسط شناسه (ID) آن کافی است یک درخواست HTTP GET به آدرس "/api/products/id" ارسال کنید.

```
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);
        });
}
```

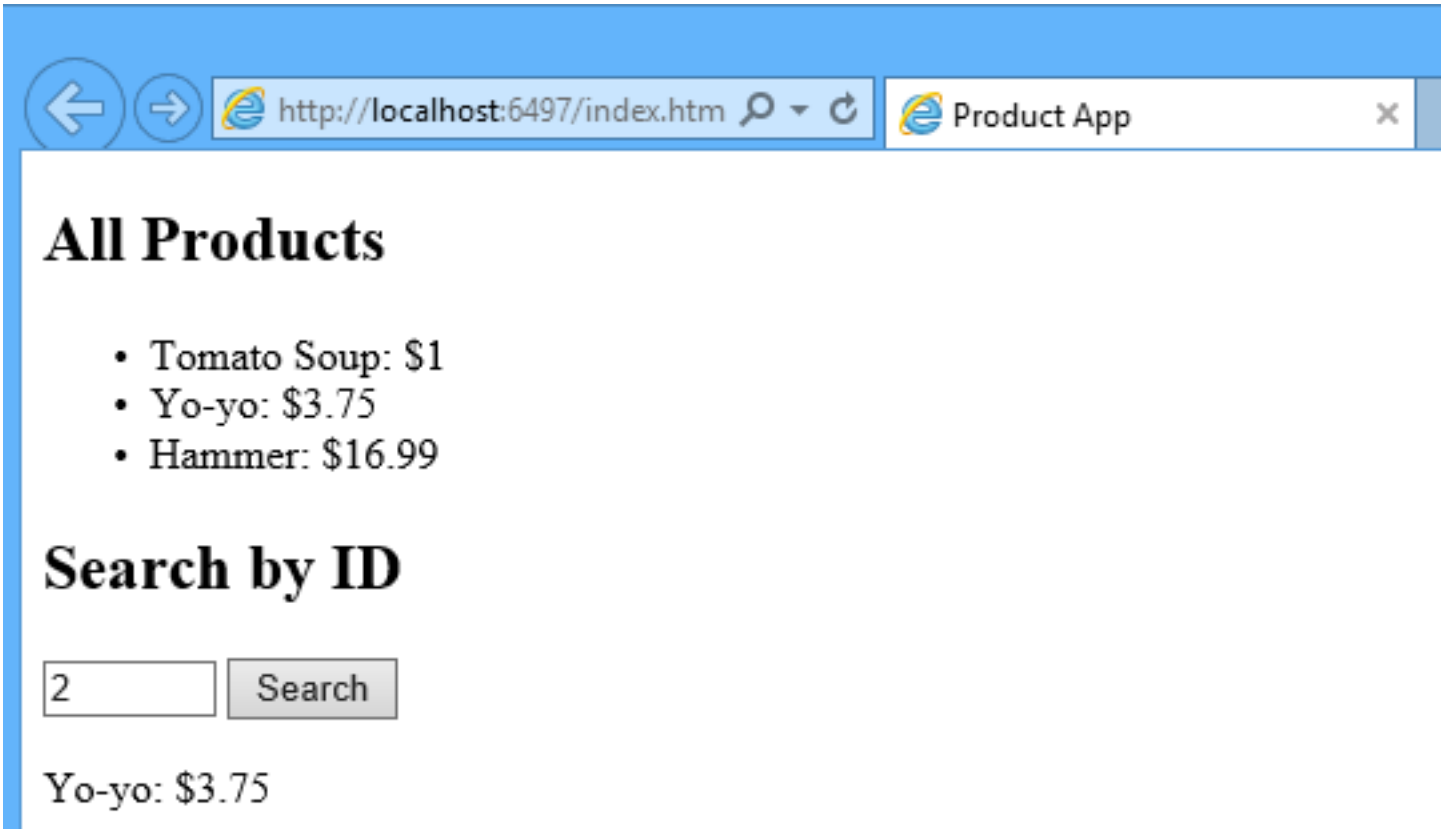
برای این کار هنوز از getJSON برای ارسال درخواست آژاکسی استفاده می‌کنیم، اما اینبار شناسه محصول را هم به آدرس درخواستی اضافه کرده ایم. پاسخ دریافتی از این درخواست، اطلاعات یک محصول با فرمت JSON است.

اجرای اپلیکیشن

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



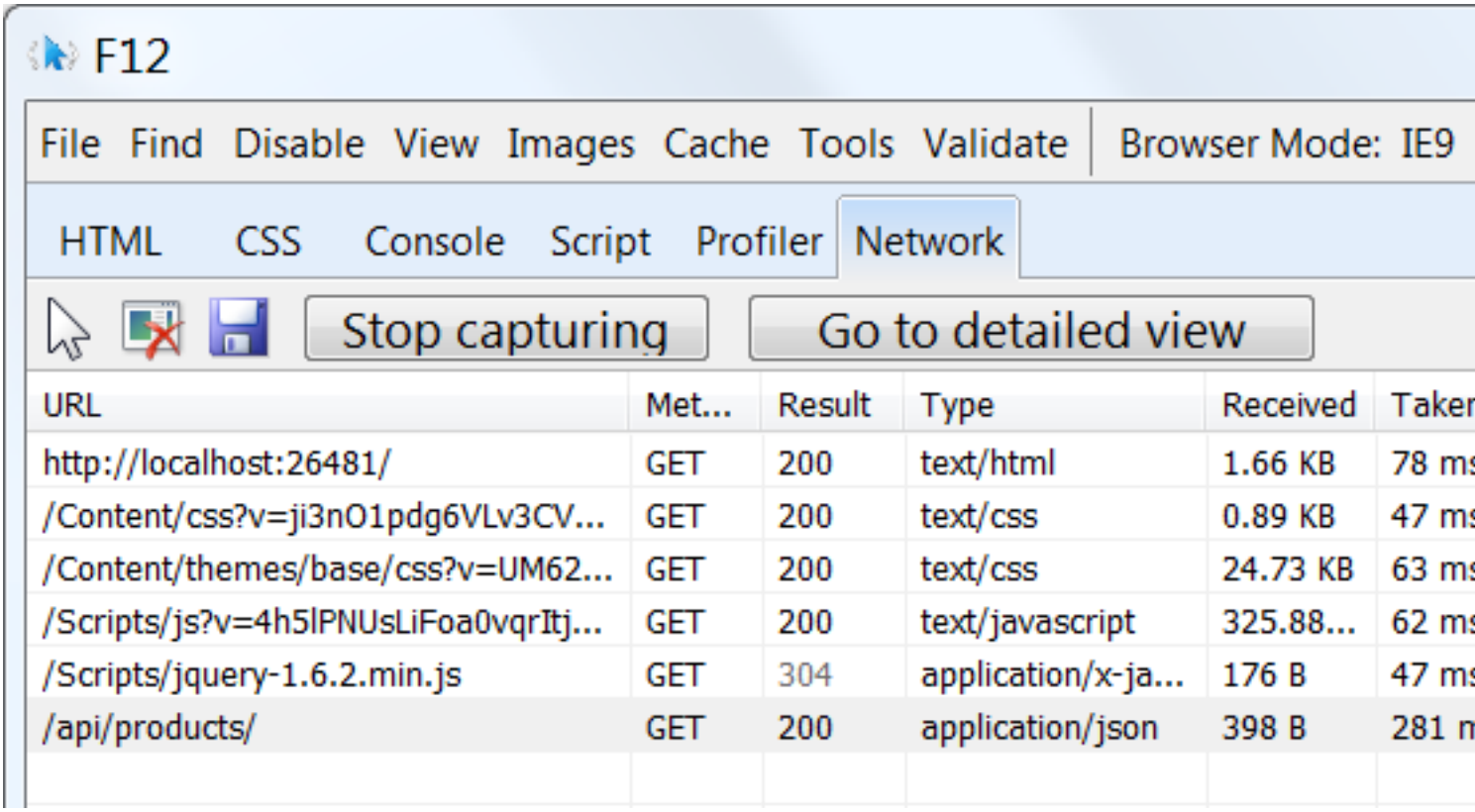
برای گرفتن محصولی مشخص، شناسه آن را وارد کنید و روی Search کلیک کنید.



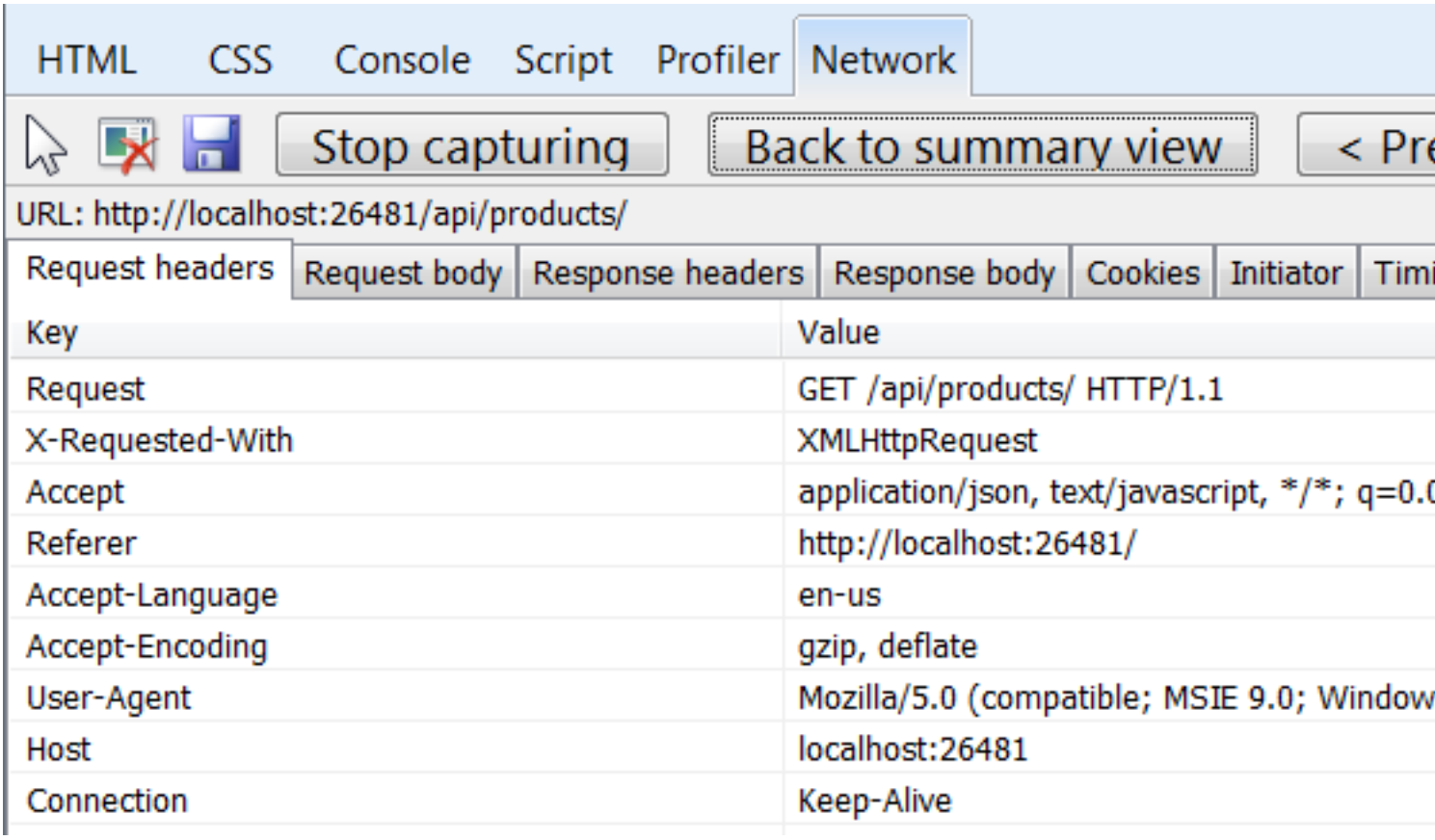
اگر شناسه نامعتبری وارد کنید، سرور یک خطای HTTP بر می‌گرداند.

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: **استفاده از Web API در ASP.NET Web Forms**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۱/۰۲ ۲۳:۵۰
آدرس: www.dotnettips.info
گروه‌ها: ASP.Net, ASP.NET Web API, Web API, ASP.NET Web Forms

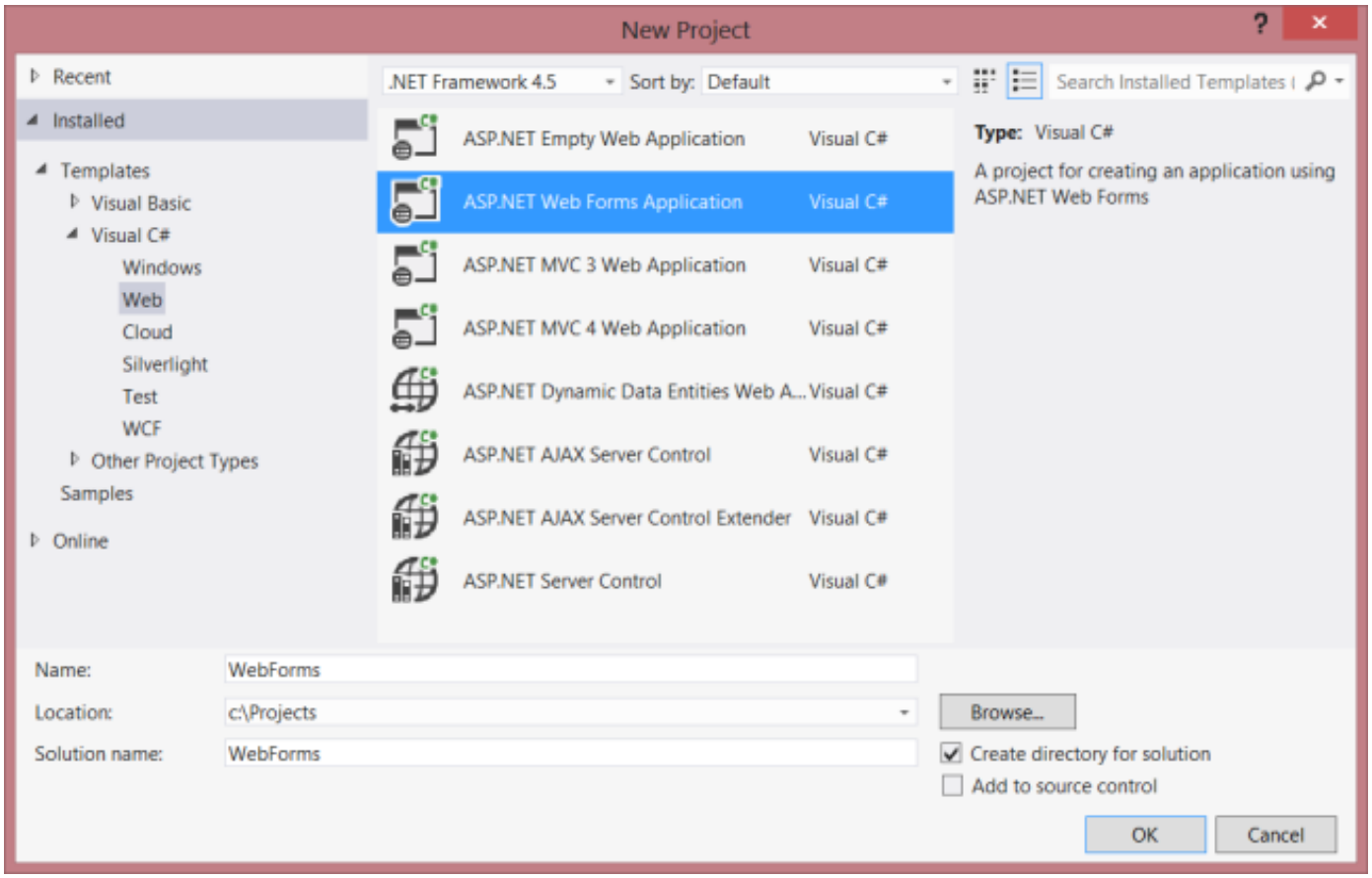
گرچه ASP.NET Web API بهمراه ASP.NET MVC بسته بندی شده و استفاده می‌شود، اما اضافه کردن آن به اپلیکیشن‌های ASP.NET Web Forms کار ساده ای است. در این مقاله مراحل لازم را بررسی می‌کنیم.

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.



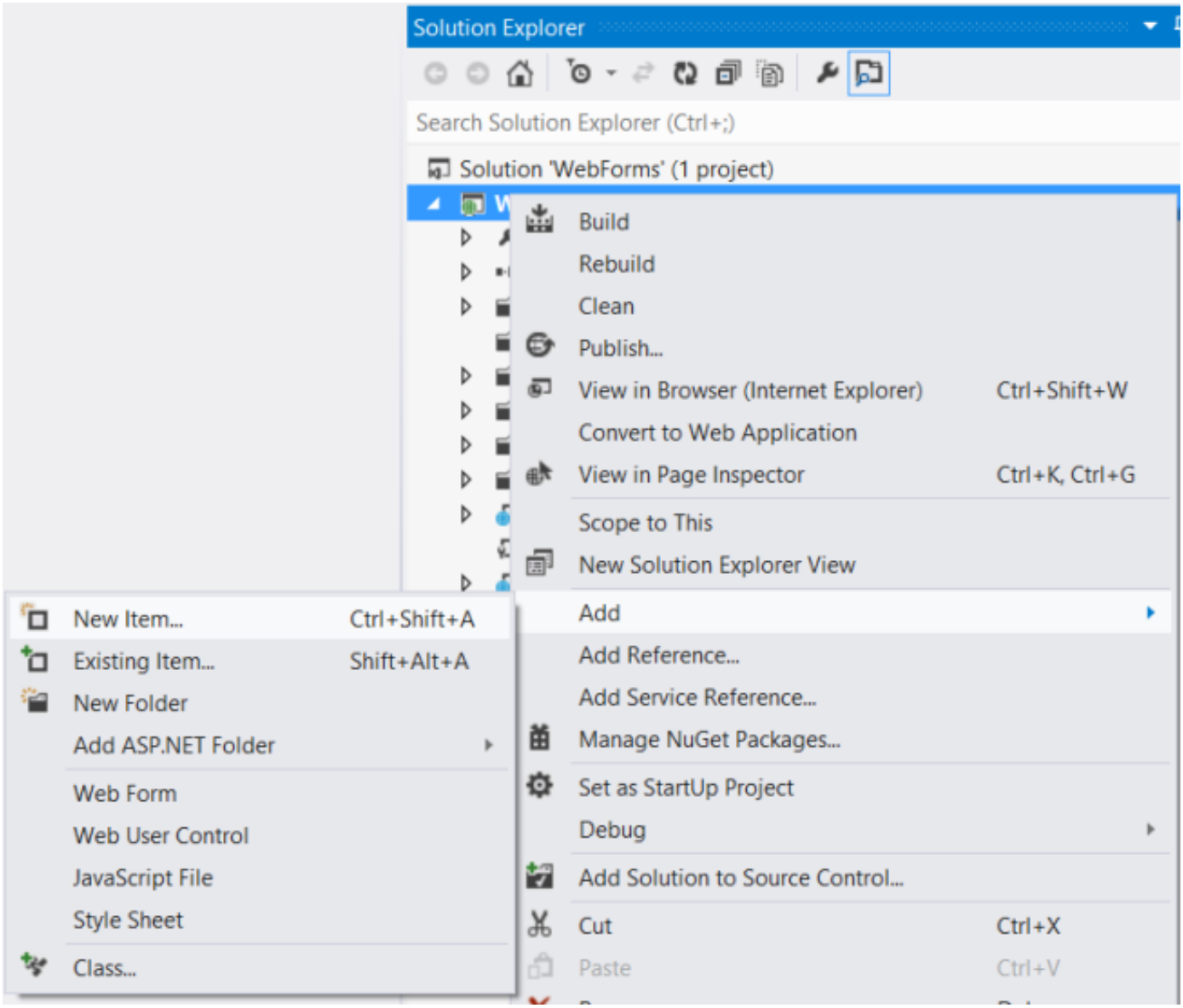
کنترلر و مدل اپلیکیشن را ایجاد کنید

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

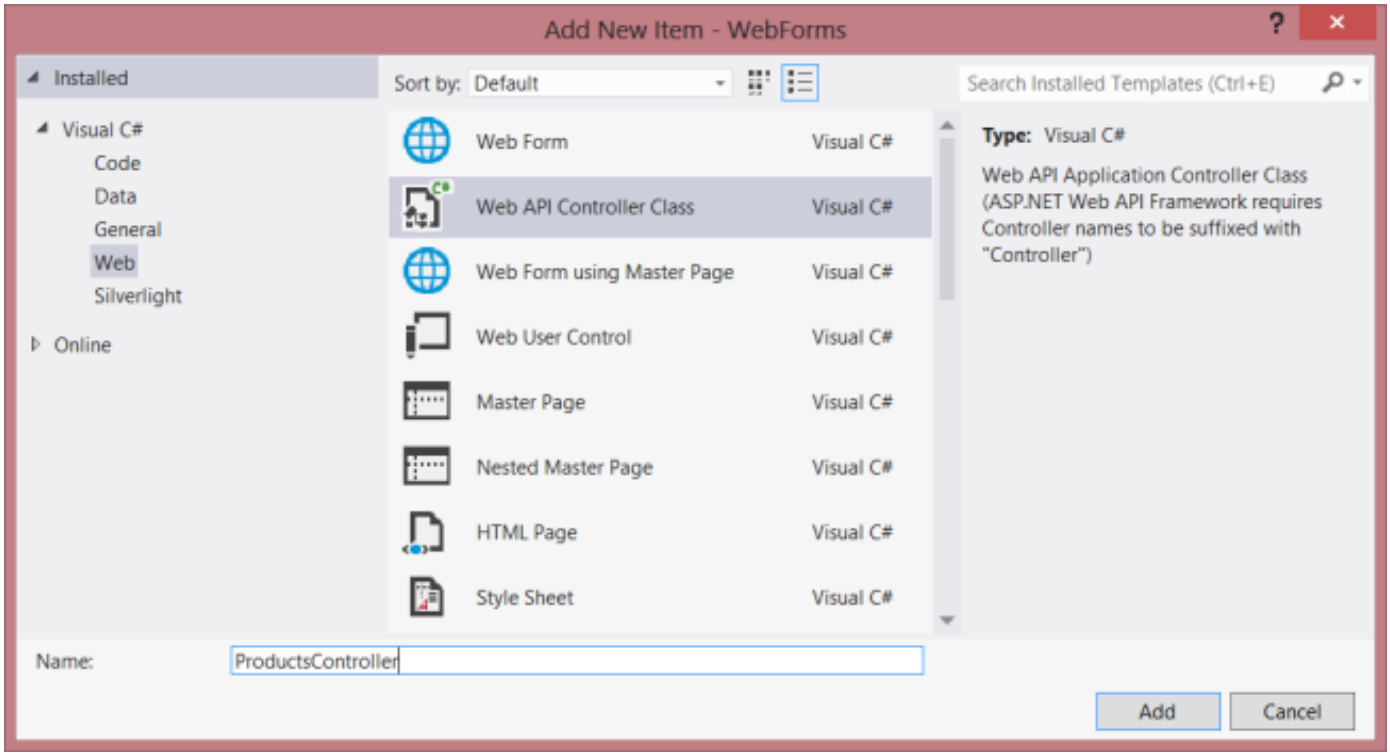
```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public string Category { get; set; }
```

```
}
```

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 صفحهHTML 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.

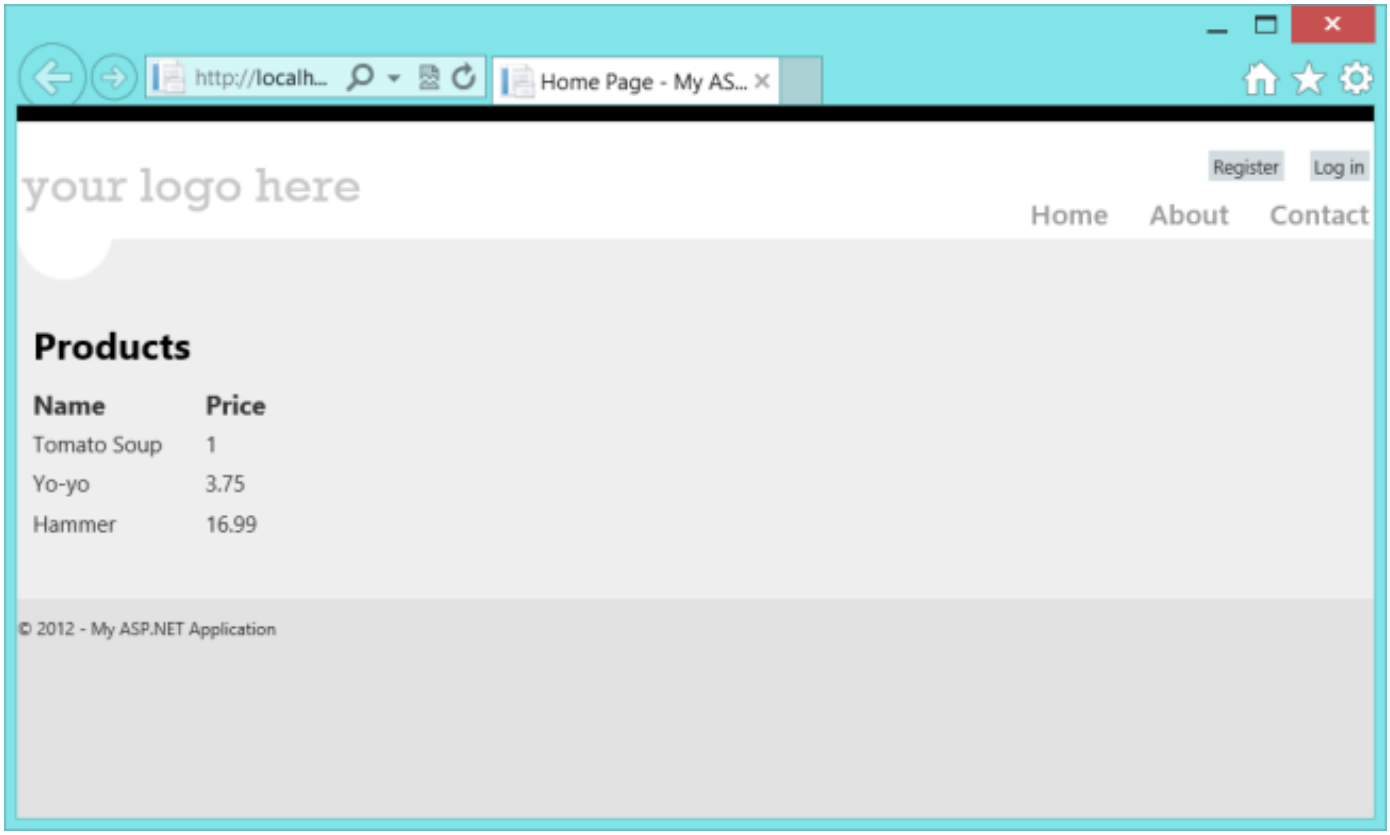
نکته: برای ارجاع دادن اسکریپت‌ها می‌توانید بسادگی فایل مورد نظر را با drag & drop به کد خود اضافه کنید.

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. سرویس 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:


http://localh...
Home Page - My AS...
your logo here
Register
Log in
Home
About
Contact
Products
Name
Price
Tomato Soup
1
Yo-yo
3.75
Hammer
16.99
© 2012 - My ASP.NET Application

نظرات خوانندگان

نویسنده: ارشیا
تاریخ: ۱۳۹۲/۱۱/۰۵ ۱۳:۲۷

اگر بخواهیم زمانی که برای فیلد price مقداری که وارد میکند حتما نوع عددی یا اعشاری که شما در نظر گرفتید باشه ، 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: ۱۳۹۲/۱۱/۰۵ ۱۹:۵۰

می تونید از جاوا اسکریپت و Remote Validation استفاده کنید.

نویسنده: ارشیا
تاریخ: ۱۳۹۲/۱۱/۰۶ ۱۱:۲۲

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

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۱۱/۰۶ ۱۲:۵

مفاهیم اعتبارسنجی در MVC با Web Api تقریبا یکی است.

نویسنده: مهرداد
تاریخ: ۱۳۹۳/۰۲/۱۵ ۱۶:۴۵

- آیا برای عملیات CRUD می‌توان 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

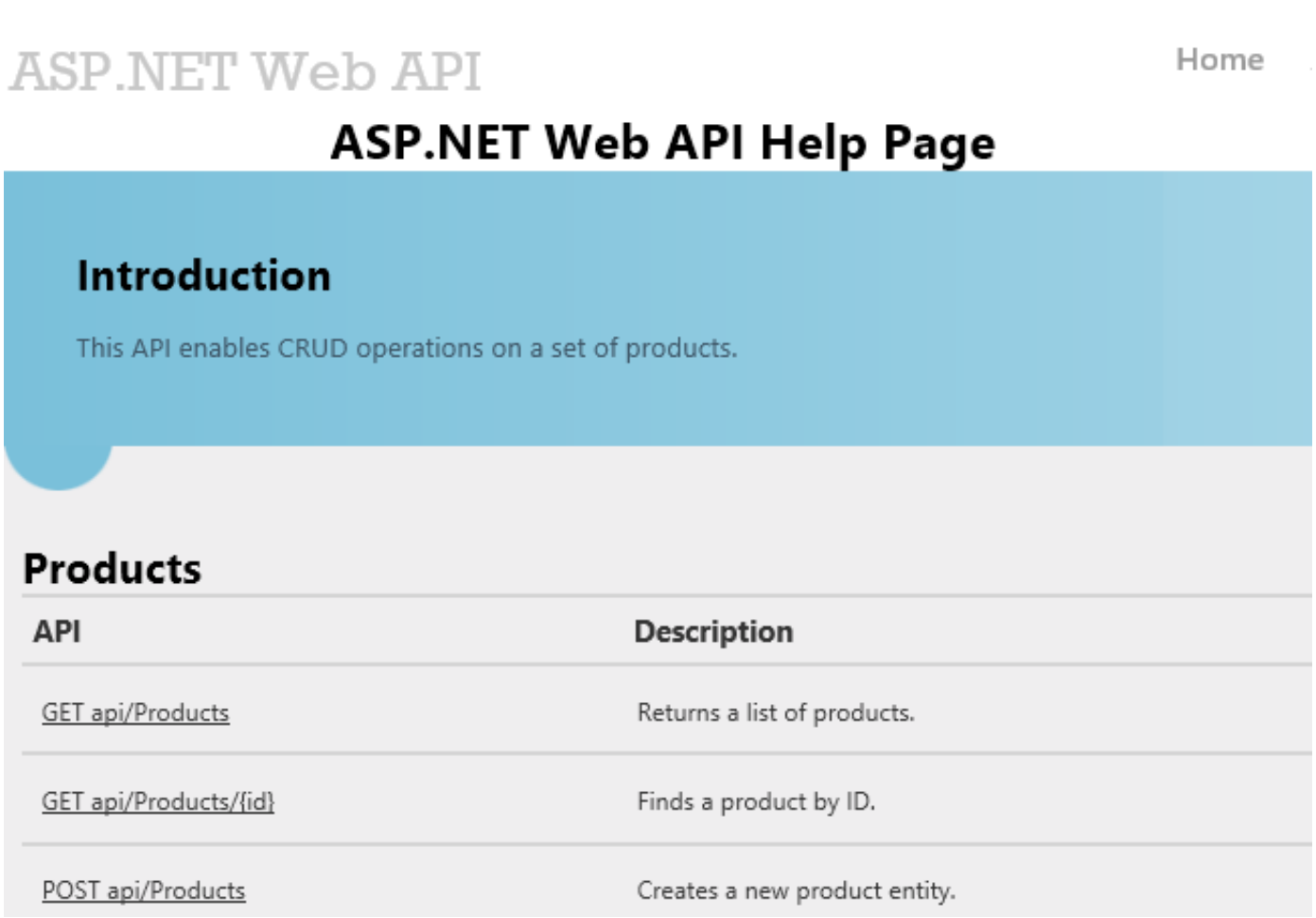
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۲/۱۵ ۱۷:۲۵

- 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 وب‌متدهای استاتیک صفحات aspx یا فایل‌های ashx یا asmx و حتی سرویس‌های 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 فرم‌ها و نه MVC)؛ به هیچکدام از نگارش‌های آن. سمت کاربر آن AngularJS و سمت سرور آن Web API باشد. کار می‌کند. (اهمیت این مساله در اینجا است که الان می‌شود یک فریم ورک جدید توسعه‌ی برنامه‌های وب را کاملا مستقل از وب فرم‌ها و MVC طراحی کرد)

عنوان: **ایجاد صفحات راهنما برای ASP.NET Web API**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۱/۰۳ ۰:۵۰
آدرس: www.dotnettips.info
گروه‌ها: ASP.Net, ASP.NET Web API, Web API

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.

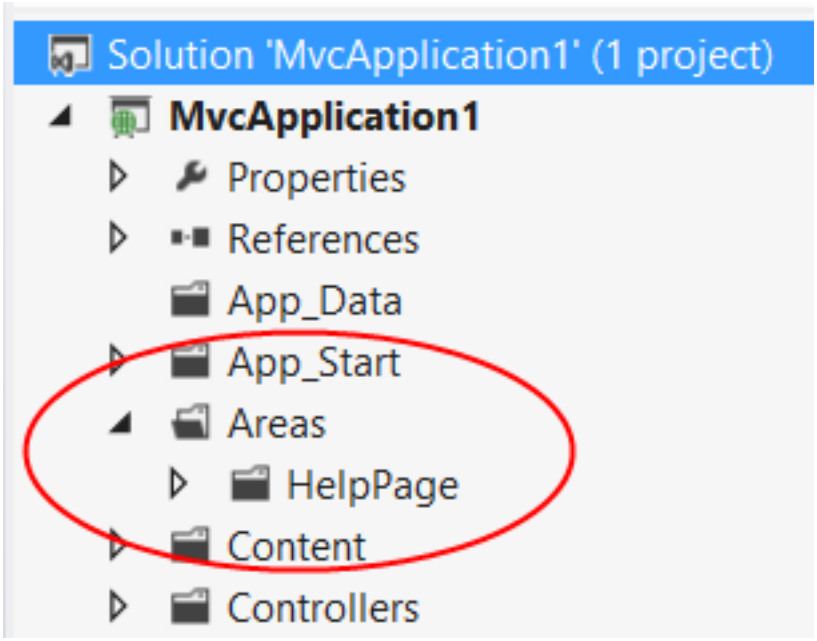
بدین منظور فریم ورک ASP.NET Web API کتابخانه ای برای تولید خودکار صفحات راهنما در زمان اجرا (run-time) فراهم کرده است.



ایجاد صفحات راهنمای API

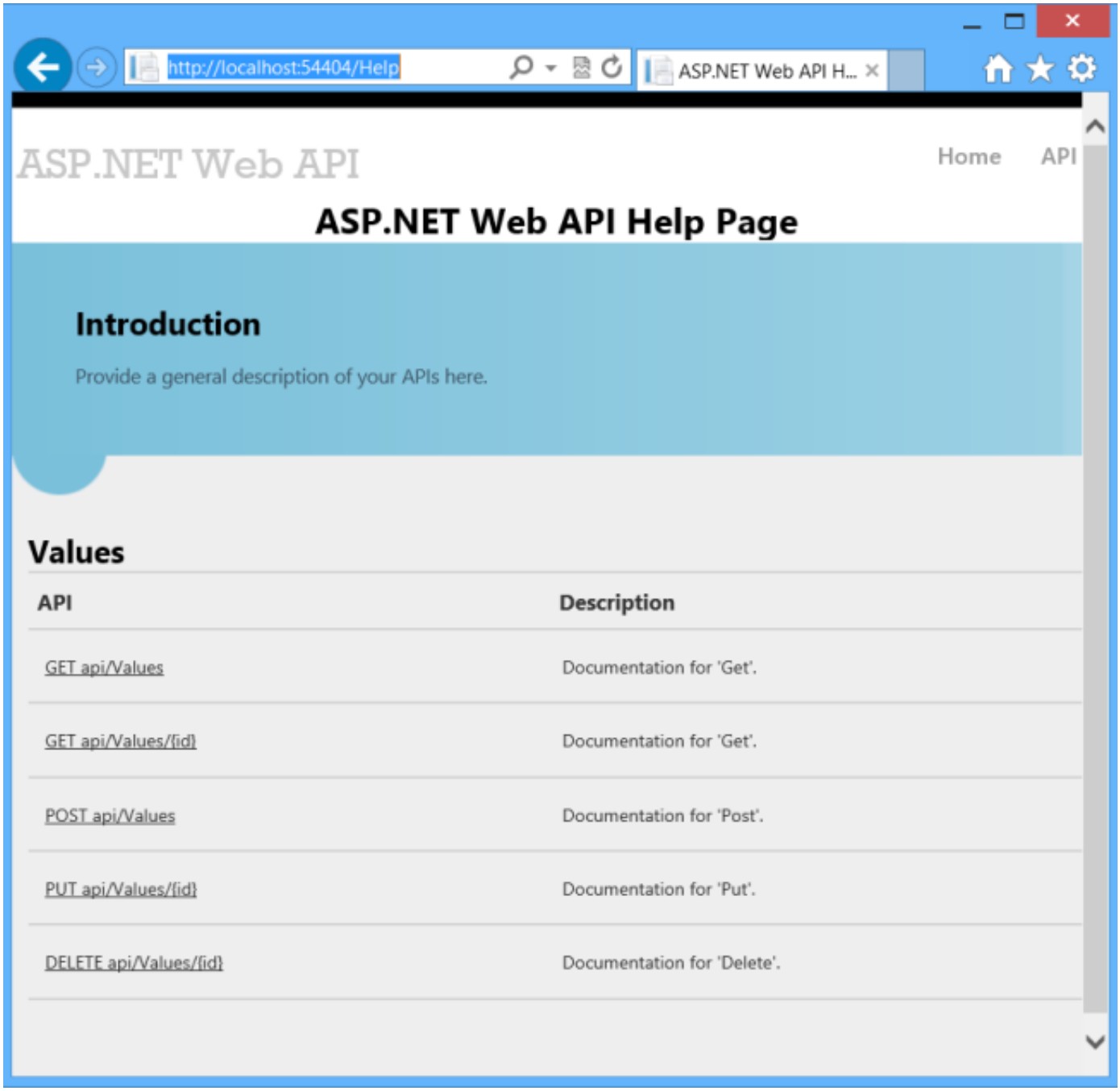
برای شروع ابتدا ابزار ASP.NET and Web Tools 2012.2 Update را نصب کنید. اگر از ویژوال استودیو 2013 استفاده می‌کنید این ابزار بصورت خودکار نصب شده است. این ابزار صفحات راهنما را به قالب پروژه‌های ASP.NET Web API اضافه می‌کند.

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 مثلا قالب، عنوان، استایل‌ها و دیگر موارد را تغییر دهید.

بخش اصلی این صفحه متشکل از جدولی است که APIها را بر اساس کنترلر طبقه بندی 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.

افزودن صفحات راهنما به پروژه ای قدیمی

می توانید 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.

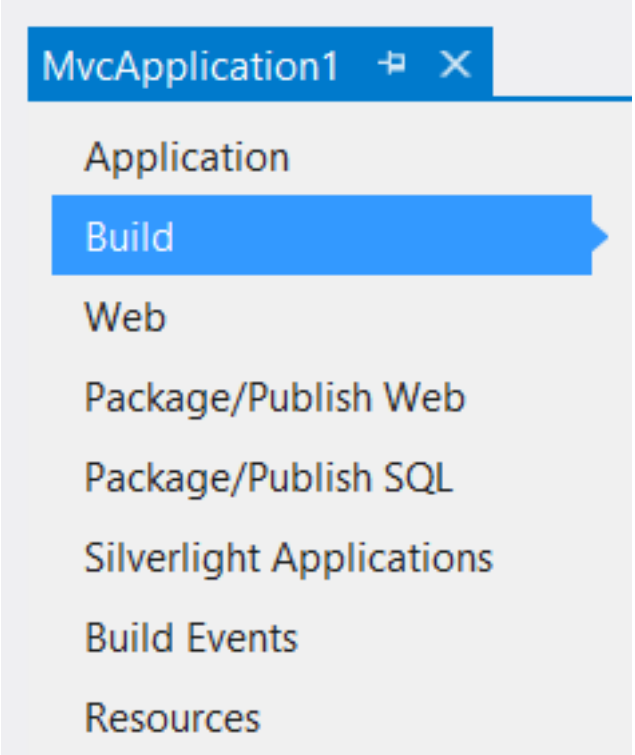
```
protected void Application_Start()
{
    // Add this code, if not present.
    AreaRegistration.RegisterAllAreas();

    // ...
}
```

افزودن مستندات API

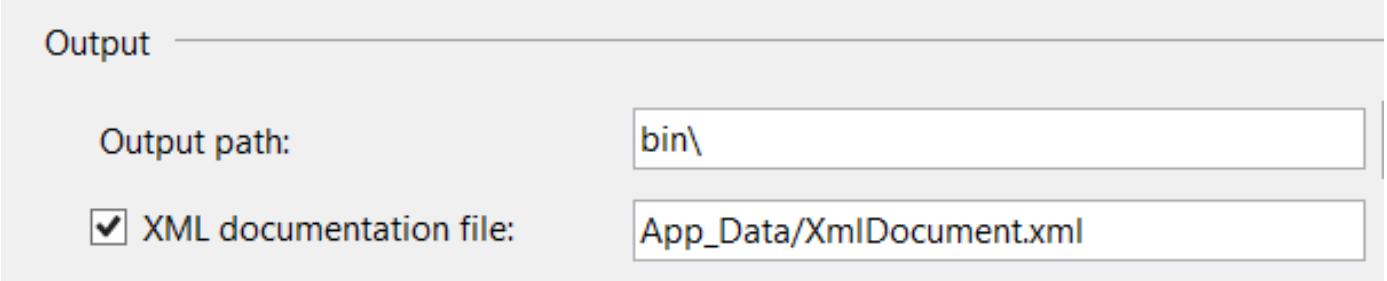
بصورت پیش فرض صفحات راهنما از placeholder-text برای مستندات استفاده می‌کنند. می‌توانید برای ساختن مستندات از توضیحات XML استفاده کنید. برای فعال سازی این قابلیت فایل Areas/HelpPage/App_Start/HelpPageConfig.cs را باز کنید و خط زیر را از حالت کامنت درآورید:

```
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.



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

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: ۱۳۹۲/۱۱/۱۸ ۱۹:۵۵

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

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

Enabling ASP.NET Web API Help Pages for ASP.NET Web Forms Applications

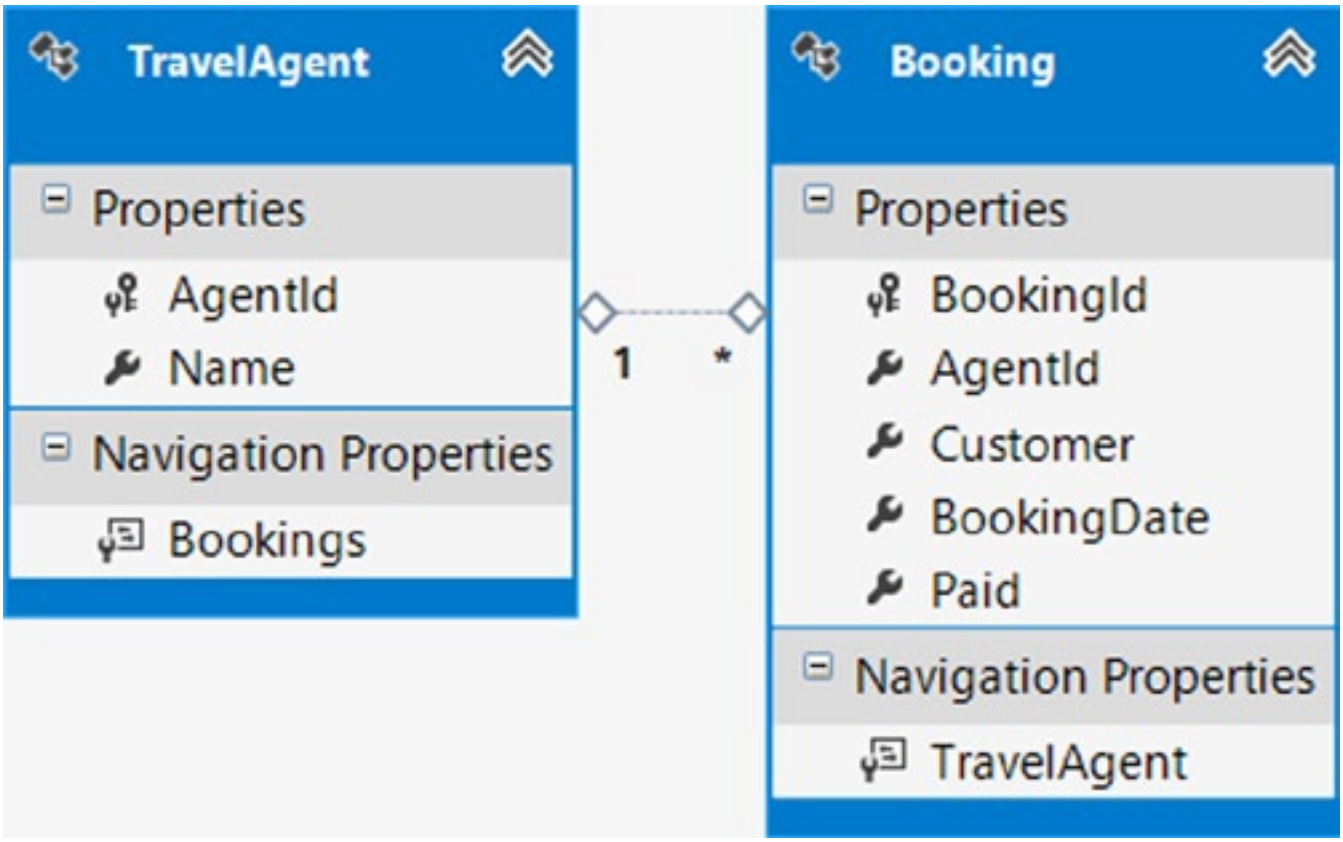
عنوان: **استفاده از EF در اپلیکیشن های N-Tier : قسمت سوم**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۱/۰۸ ۱۰:۵
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, Tips, ASP.NET Web API, N-Layer Architecture, Web API, Entity Framework 6

در قسمت قبلی بروز رسانی موجودیت‌های منفصل با WCF را بررسی کردیم. در این قسمت خواهیم دید چگونه می‌توان تغییرات موجودیت‌ها را تشخیص داد و عملیات CRUD را روی یک Object Graph اجرا کرد.

تشخیص تغییرات با Web API

فرض کنید می‌خواهیم از سرویس‌های Web API برای انجام عملیات CRUD استفاده کنیم، اما بدون آنکه برای هر موجودیت متدهایی مجزا تعریف کنیم. به بیان دیگر می‌خواهیم عملیات مذکور را روی یک Object Graph انجام دهیم. مدیریت داده‌ها هم با مدل Code-First پیاده سازی می‌شود. در مثال جاری یک اپلیکیشن کنسول خواهیم داشت که بعنوان یک کلاینت سرویس را فراخوانی می‌کند. 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 می‌بینید مدل ما آژانس‌های مسافرتی و رزرواسیون آنها را ارائه می‌کند. می‌خواهیم مدل و کد دسترسی داده‌ها 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 نام‌های 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 می‌کند. همچنین به JSON serializer می‌گوییم که self-referencing loop خاصیت‌های پیمایشی را نادیده بگیرد. این حلقه بدلیل ارتباط bidirectional بین موجودیت‌ها 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;
    ...
```

```
}
```

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
MediaTypeWithQualityHeaderValue("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 کلاس‌های TravelAgent و Booking را به پروژه کلاینت اضافه کنید. اینگونه کدها بهتر است در لایه مجزایی قرار گیرند و بین پروژه‌ها 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 می‌کند. در این مرحله سایت در حال اجرا است و سرویس‌ها 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 می‌کنیم و با تنظیم مقدار خاصیت Accept Header از سرویس درخواست می‌کنیم که اطلاعات را با فرمت JSON بازگرداند.

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

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

نکته: قابل ذکر است که اگر موجودیت‌های متعددی با مقداری یکسان در خاصیت کلید اصلی (Primary-key value) دارید باید مجموعه آبجکت‌های 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 می‌کند چرا که چند موجودیت با مقادیر کلید اصلی یکسان به context جاری اضافه کرده اید.

بعد از آن بر اساس مقدار خاصیت Id مشخص می‌کنیم که موجودیت‌ها 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 می‌دهیم. 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 می‌کنیم.

سپس تمام آبجکت‌های فرزند آژانس مسافرتی (رزرواسیون ها) را بررسی میکنیم و همین منطق را روی آنها اعمال می‌کنیم. یعنی در صورتی که مقدار فیلد Id آنها بزرگتر از 0 باشد وضعیت EntityState را به Modified تغییر می‌دهیم. در نهایت متد SaveChanges را فراخوانی می‌کنیم. در این مرحله برای موجودیت‌های جدید اسکریپت‌های Insert و برای موجودیت‌های تغییر کرده اسکریپت‌های Update تولید می‌شود. سپس کد وضعیت مناسب را به کلاینت بر می‌گردانیم. برای موجودیت‌های اضافه شده کد وضعیت 201 (Created) و برای موجودیت‌های بروز رسانی شده کد وضعیت 200 (OK) باز می‌گردد. کد 201 به کلاینت اطلاع می‌دهد که رکورد جدید با موفقیت ثبت شده است، و کد 200 از بروز رسانی موفقیت آمیز خبر می‌دهد. هنگام تولید سرویس‌های REST-based بهتر است همیشه کد وضعیت مناسبی تولید کنید.

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

در آخر کلاینت ما متد Retreive را روی سرویس فراخوانی می‌کند. این فراخوانی با کمک متد GetAsync انجام می‌شود که روی آبجکت HttpClient تعریف شده است. فراخوانی این متد تمام آژانس‌های مسافرتی بهمراه رزرواسیون‌های متناظرشان را دریافت می‌کند. در اینجا با استفاده از متد Include تمام رکوردهای فرزند را بهمراه تمام خاصیت هایشان (properties) بارگذاری می‌کنیم.

دقت کنید که مرتب کننده JSON تمام خواص عمومی (public properties) را باز می‌گرداند، 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 درخواست‌های HTTP به اکشن متدها استفاده نکرده ایم. مثلا بصورت پیش فرض درخواست‌های POST به متدهایی نگاشت می‌شوند که نام آنها با "Post" شروع می‌شود. در مثال جاری قواعد مسیریابی را تغییر داده ایم و رویکرد مسیریابی RPC-based را در پیش گرفته ایم. در اپلیکیشن‌های واقعی بهتر است از قواعد پیش فرض استفاده کنید چرا که هدف Web API ارائه سرویس‌های REST-based است. بنابراین بعنوان یک قاعده کلی بهتر است متدهای سرویس شما به درخواست‌های 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: ۱۳۹۲/۱۱/۱۱ ۱۱:۶

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 خواست‌های 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 می‌شود. ممنون

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۱۱/۱۱ ۱۱:۴۲

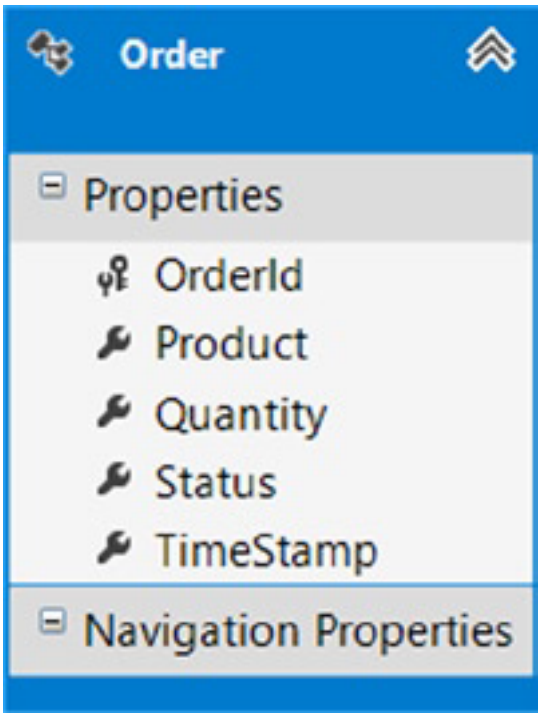
برای برقراری امنیت، تعیین هویت و اعتبارسنجی در وب API عموما یا از Forms authentication استفاده می‌شود و یا از ASP.NET Identity . زیر ساخت آن یکی است و مشترک.

عنوان: **استفاده از EF در اپلیکیشن های N-Tier : قسمت ششم**
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۲/۱۱/۱۱ ۰:۲۵
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, Tips, WCF, N-Layer Architecture, Web API, Entity Framework 6

در قسمت قبل رویکردهای مختلف برای حذف موجودیت‌های منفصل را بررسی کردیم. در این قسمت مدیریت همزمانی یا Concurrency را بررسی خواهیم کرد.

فرض کنید می‌خواهیم مطمئن شویم که موجودیتی که توسط یک کلاینت WCF تغییر کرده است، تنها در صورتی بروز رسانی شود که شناسه (token) همزمانی آن تغییر نکرده باشد. به بیان دیگر شناسه ای که هنگام دریافت موجودیت بدست می‌آید، هنگام بروز رسانی باید مقداری یکسان داشته باشد.

مدل زیر را در نظر بگیرید.



می‌خواهیم یک سفارش (order) را توسط یک سرویس WCF بروز رسانی کنیم در حالی که اطمینان حاصل می‌کنیم موجودیت سفارش از زمانی که دریافت شده تغییری نکرده است. برای مدیریت این وضعیت دو رویکرد تقریبا متفاوت را بررسی می‌کنیم. در هر دو رویکرد از یک ستون همزمانی استفاده می‌کنیم، در این مثال فیلد TimeStamp.

در ویژوال استودیو پروژه جدیدی از نوع WCF Service Library بسازید و نام آن را به Recipe6 تغییر دهید.
روی نام پروژه کلیک راست کنید و گزینه Add New Item را انتخاب کنید. سپس گزینه‌های Data -> Entity Data Model را برگزینید. از ویزارد ویژوال استودیو برای اضافه کردن مدل جاری و جدول Orders استفاده کنید. در EF Designer روی فیلد TimeStamp کلیک راست کنید و گزینه Properties را انتخاب کنید. سپس مقدار CuncurrencyMode آنرا به Fixed تغییر دهید.
فایل IService1.cs را باز کنید و تعریف سرویس را مطابق لیست زیر بروز رسانی کنید.

```
[ServiceContract]
public interface IService1
{
    [OperationContract]
```

```
    Order InsertOrder();
    [OperationContract]
    void UpdateOrderWithoutRetrieving(Order order);
    [OperationContract]
    void UpdateOrderByRetrieving(Order order);
}
```

فایل Service1.cs را باز کنید و پیاده سازی سرویس را مطابق لیست زیر تکمیل کنید.

```
public class Service1 : IService1
{
    public Order InsertOrder()
    {
        using (var context = new EFRecipesEntities())
        {
            // remove previous test data
            context.Database.ExecuteSqlCommand("delete from [orders]");
            var order = new Order
            {
                Product = "Camping Tent",
                Quantity = 3,
                Status = "Received"
            };
            context.Orders.Add(order);
            context.SaveChanges();
            return order;
        }
    }

    public void UpdateOrderWithoutRetrieving(Order order)
    {
        using (var context = new EFRecipesEntities())
        {
            try
            {
                context.Orders.Attach(order);
                if (order.Status == "Received")
                {
                    context.Entry(order).Property(x => x.Quantity).IsModified = true;
                    context.SaveChanges();
                }
            }
            catch (OptimisticConcurrencyException ex)
            {
                // Handle OptimisticConcurrencyException
            }
        }
    }

    public void UpdateOrderByRetrieving(Order order)
    {
        using (var context = new EFRecipesEntities())
        {
            // fetch current entity from database
            var dbOrder = context.Orders
            .Single(o => o.OrderId == order.OrderId);
            if (dbOrder != null &&
                // execute concurrency check
                StructuralComparisons.StructuralEqualityComparer.Equals(order.TimeStamp,
dbOrder.TimeStamp))
            {
                dbOrder.Quantity = order.Quantity;
                context.SaveChanges();
            }
            else
            {
                // Add code to handle concurrency issue
            }
        }
    }
}
```

برای تست این سرویس به یک کلاینت نیاز داریم. پروژه جدیدی از نوع Console Application به راه حل جاری اضافه کنید و کد آن را مطابق لیست زیر تکمیل کنید. با کلیک راست روی نام پروژه و انتخاب گزینه Add Service Reference سرویس پروژه را هم ارجاع کنید. دقت کنید که ممکن است پیش از آنکه بتوانید سرویس را ارجاع کنید نیاز باشد روی آن کلیک راست کرده و از منوی

Debug گزینه Start Instance را انتخاب کنید تا وهله از سرویس به اجرا در بیاید.

```
class Program
{
    static void Main(string[] args)
    {
        var service = new Service1Client();
        var order = service.InsertOrder();
        order.Quantity = 5;
        service.UpdateOrderWithoutRetrieving(order);
        order = service.InsertOrder();
        order.Quantity = 3;
        service.UpdateOrderByRetrieving(order);
    }
}
```

اگر به خط اول متد ()Main یک breakpoint اضافه کنید و اپلیکیشن را اجرا کنید می‌توانید افزودن و بروز رسانی یک Order با هر دو رویکرد را بررسی کنید.

شرح مثال جاری

متد ()InsertOrder داده‌های پیشین را حذف می‌کند، سفارش جدیدی می‌سازد و آن را در دیتابیس ثبت می‌کند. در آخر موجودیت جدید به کلاینت باز می‌گردد. موجودیت بازگشتی هر دو مقدار OrderId و TimeStamp را دارا است که توسط دیتابیس تولید شده اند. سپس در کلاینت از دو رویکرد نسبتا متفاوت برای بروز رسانی موجودیت استفاده می‌کنیم.

در رویکرد نخست، متد ()UpdateOrderWithoutRetrieving موجودیت دریافت شده از کلاینت را Attach می‌کند و چک می‌کند که مقدار فیلد Status چیست. اگر مقدار این فیلد "Received" باشد، فیلد Quantity را با EntityState.Modified علامت گذاری می‌کنیم و متد ()SaveChanges را فراخوانی می‌کنیم. EF دستورات لازم برای بروز رسانی را تولید می‌کند، که فیلد quantity را مقدار دهی کرده و یک عبارت where هم دارد که فیلدهای OrderId و TimeStamp را چک می‌کند. اگر مقدار TimeStamp توسط یک دستور بروز رسانی تغییر کرده باشد، بروز رسانی جاری با خطا مواجه خواهد شد. برای مدیریت این خطا ما بدنه کد را در یک بلاک try/catch قرار می‌دهیم، و استثنای OptimisticConcurrencyException را مهار می‌کنیم. این کار باعث می‌شود اطمینان داشته باشیم که موجودیت Order دریافت شده از متد ()InsertOrder تاکنون تغییری نکرده است. دقت کنید که در مثال جاری تمام خواص موجودیت بروز رسانی می‌شوند، صرفنظر از اینکه تغییر کرده باشند یا خیر.

رویکرد دوم نشان می‌دهد که چگونه می‌توان وضعیت همزمانی موجودیت را پیش از بروز رسانی مشخصا دریافت و بررسی کرد. در اینجا می‌توانید مقدار TimeStamp موجودیت را از دیتابیس بگیرید و آن را با مقدار موجودیت کلاینت مقایسه کنید تا وجود تغییرات مشخص شود. این رویکرد در متد ()UpdateOrderByRetrieving نمایش داده شده است. گرچه این رویکرد برای تشخیص تغییرات خواص موجودیت‌ها و یا روابط شان مفید و کارآمد است، اما بهترین روش هم نیست. مثلا ممکن است از زمانی که موجودیت را از دیتابیس دریافت می‌کنید، تا زمانی که مقدار TimeStamp آن را مقایسه می‌کنید و نهایتا متد ()SaveChanges را صدا میزنید، موجودیت شما توسط کلاینت دیگری بروز رسانی شده باشد.

مسلما رویکرد دوم هزینه برتر از رویکرد اولی است، چرا که برای مقایسه مقادیر همزمانی موجودیت ها، یکبار موجودیت را از دیتابیس دریافت می‌کنید. اما این رویکرد در مواقعی که Object graphهای بزرگ یا پیچیده (complex) دارید بهتر است، چون پیش از ارسال موجودیت‌ها به context در صورت برابر نبودن مقادیر همزمانی پروسس را لغو می‌کنید.

نظرات خوانندگان

نویسنده: Senator
تاریخ: ۱۳۹۲/۱۱/۱۶ ۱۸:۵۴

خیلی ممنون.
عالی بود ...

عنوان: Best Practice هایی برای طراحی RESTful API - قسمت دوم
نویسنده: محسن درپرستی
تاریخ: ۱۳۹۲/۱۲/۰۳ ۱۴:۰
آدرس: www.dotnettips.info
گروه‌ها: ASP.NET Web API, Naming, Web API, RESTful API, Best Practice

طراحی Url در Restful API

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

Url پایه API خود را ساده و خوانا ، حفظ کنید . داشتن یک Url 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 Url 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 Url ها است.

بسیاری در زمان طراحی Url 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 Url 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 Url 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 Url اصلی انجام دهیم ؟
-1 از افعال Http برای کار کردن بر روی منابع استفاده کنید . با استفاده از افعال Http شامل PUT ، GET ، POST و DELETE و دو Url 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 Url 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 Url 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 Url اصلی یک منبع هستند . هر منبع مشخصات و روابط متنوعی دارد که قابل جستجو کردن، مرتب سازی، بروزرسانی و تغییر هستند. Url 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 بامعناتر و خواناتر است. اما مهمتر از روشی که بکار می‌برید ، اجتناب از بکاربردن هر دو روش با هم است ، اینکه در مورد یک منبع از اسم منفرد و در مورد دیگری از اسم جمع استفاده کنید . یکدستی API را حفظ کنید و به توسعه دهنده کمک کنید راحت‌تر API شما را یاد بگیرد.

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

API ی را در نظر بگیرید که محتواهایی را در فرمت‌های مختلف ارائه می‌دهد. بلاگ ، ویدئو ، اخبار و .... حالا فرض کنیداین 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 راحت‌تر و مفیدتر است که منابع را در قالب بلاگ ، اخبار ، ویدئو مدلسازی کنیم .