

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

سرویس ما متدهای زیر را در دسترس قرار می دهد.

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 می کنید برخی از آدرس ها، شامل شناسه محصول هم می شوند. بعنوان مثال برای گرفتن محصولی با شناسه 28، کلاینت یک درخواست GET را به آدرس زیر ارسال می کند:

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

## منابع

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

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 می توانند بصورت زیر به عملیات CRUD نگاشت شوند:

متد GET یک منبع (resource) را از آدرس تعریف شده دریافت می کند. متدهای GET هیچگونه تاثیری روی سرور نباید داشته باشند. مثلاً حذف رکوردها با متد اکیدا اشتباه است.

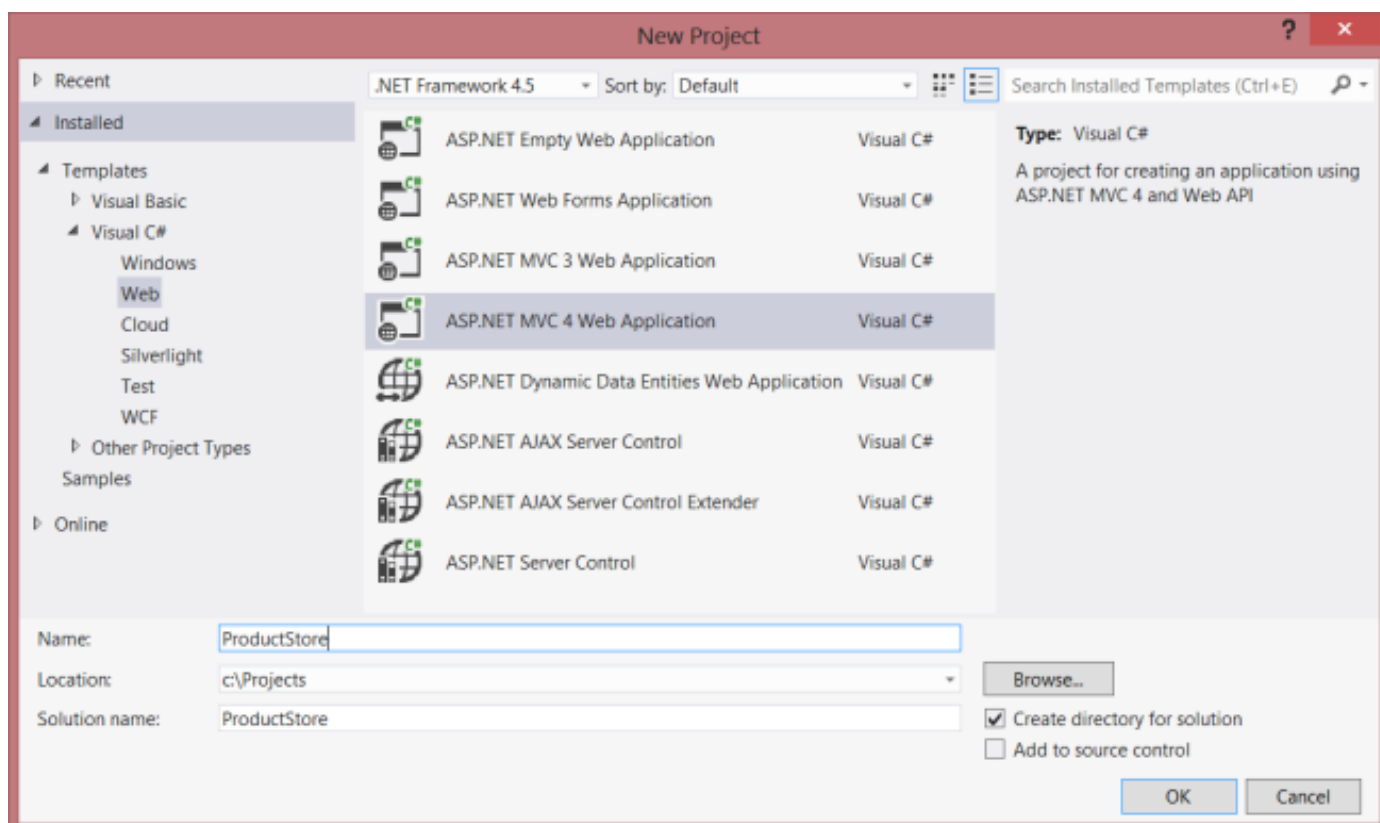
متد PUT یک منبع را در آدرس تعریف شده بروز رسانی می کند. این متد برای ساختن منابع جدید هم می تواند استفاده شود، البته در صورتی که سرور به کلاینت ها اجازه مشخص کردن آدرس های جدید را بدهد. در مثال جاری پشتیبانی از ایجاد منابع توسط متد PUT را بررسی نخواهیم کرد.

متد POST منبع جدیدی می سازد. سرور آدرس آبجکت جدید را تعیین می کند و آن را بعنوان بخشی از پیام Response بر می گرداند. متد DELETE منبعی را در آدرس تعریف شده حذف می کند.

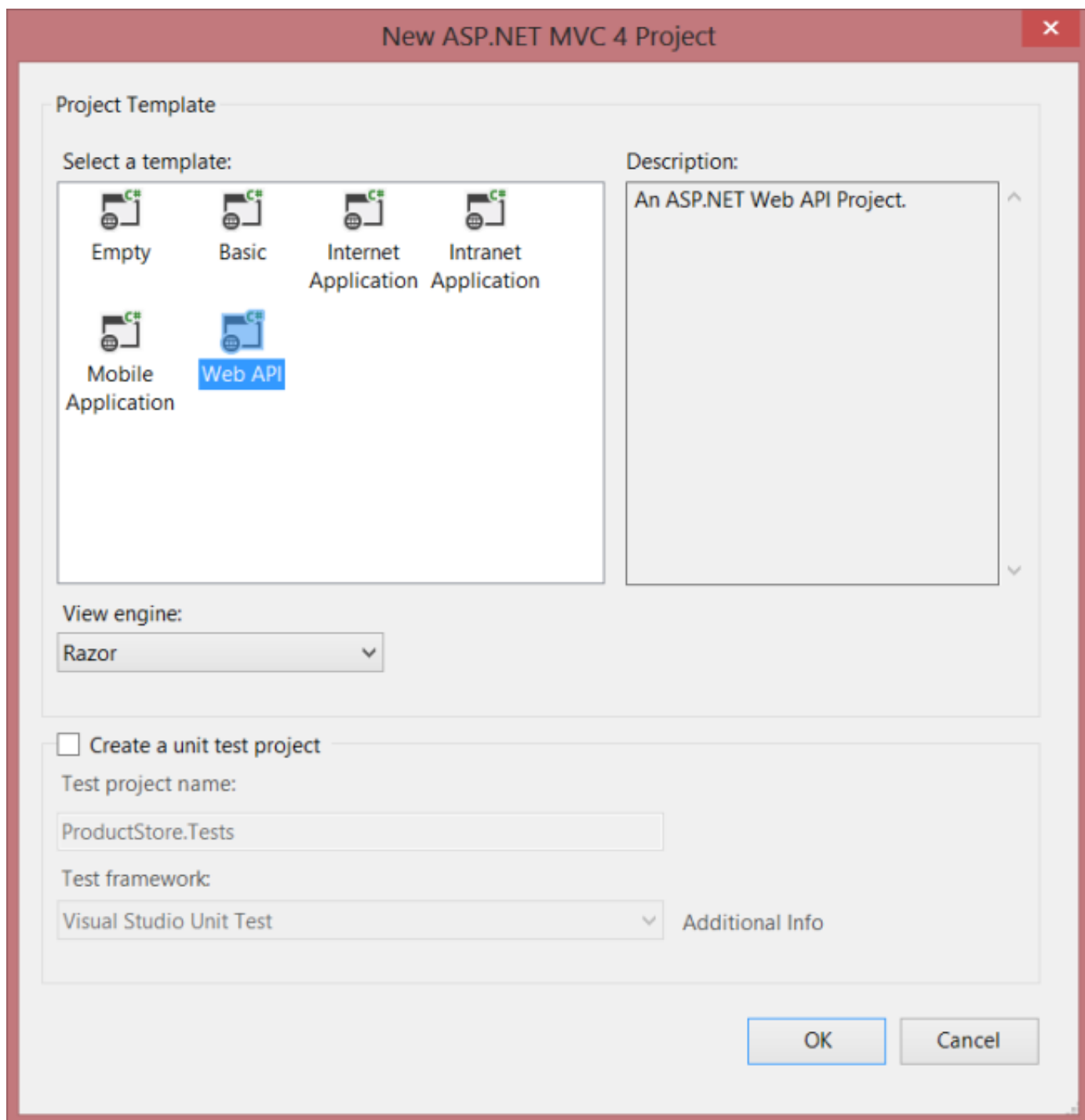
نکته: متد PUT موجودیت محصول (product entity) را کاملاً جایگزین میکند. به بیان دیگر، از کلاینت انتظار می رود که آبجکت کامل محصول را برای بروز رسانی ارسال کند. اگر می خواهید از بروز رسانی های جزئی/پاره ای (partial) پشتیبانی کنید متد PATCH توصیه می شود. مثال جاری متد PATCH را پیاده سازی نمی کند.

### 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.



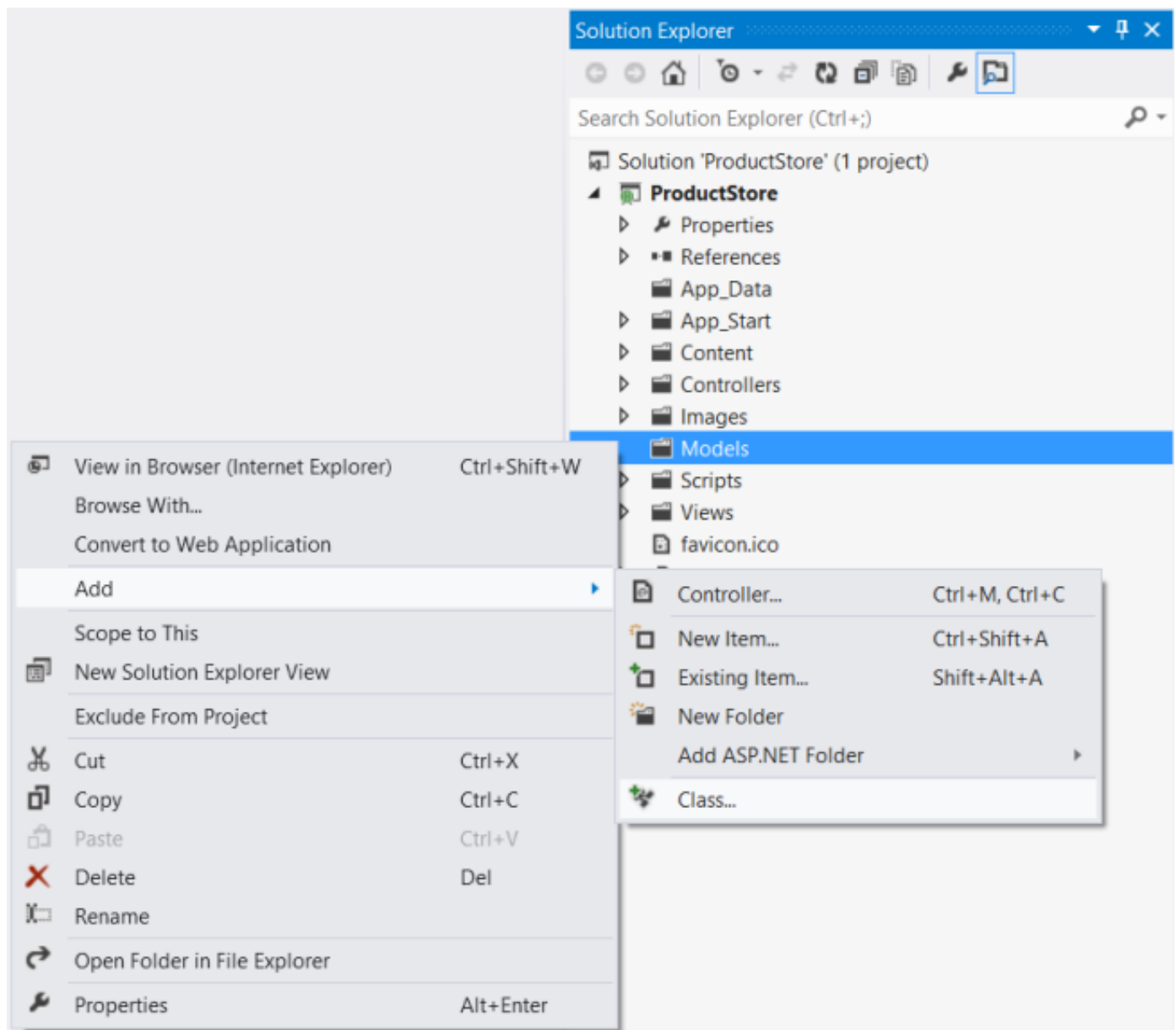
در دیالوگ New ASP.NET Project قالب Web API را انتخاب کرده و تایید کنید.



### افزودن یک مدل

یک مدل، آبجکتی است که داده اپلیکیشن شما را نمایندگی می کند. در ASP.NET Web API می توانید از آبجکت های Strongly-typed بعنوان مدل هایتان استفاده کنید که بصورت خودکار برای کلاینت به فرمت های JSON، XML مرتب (Serialize) می شوند. در مثال جاری، داده های ما محصولات هستند. پس کلاس جدیدی بنام Product می سازیم.

در پوشه Models کلاس جدیدی با نام Product بسازید.



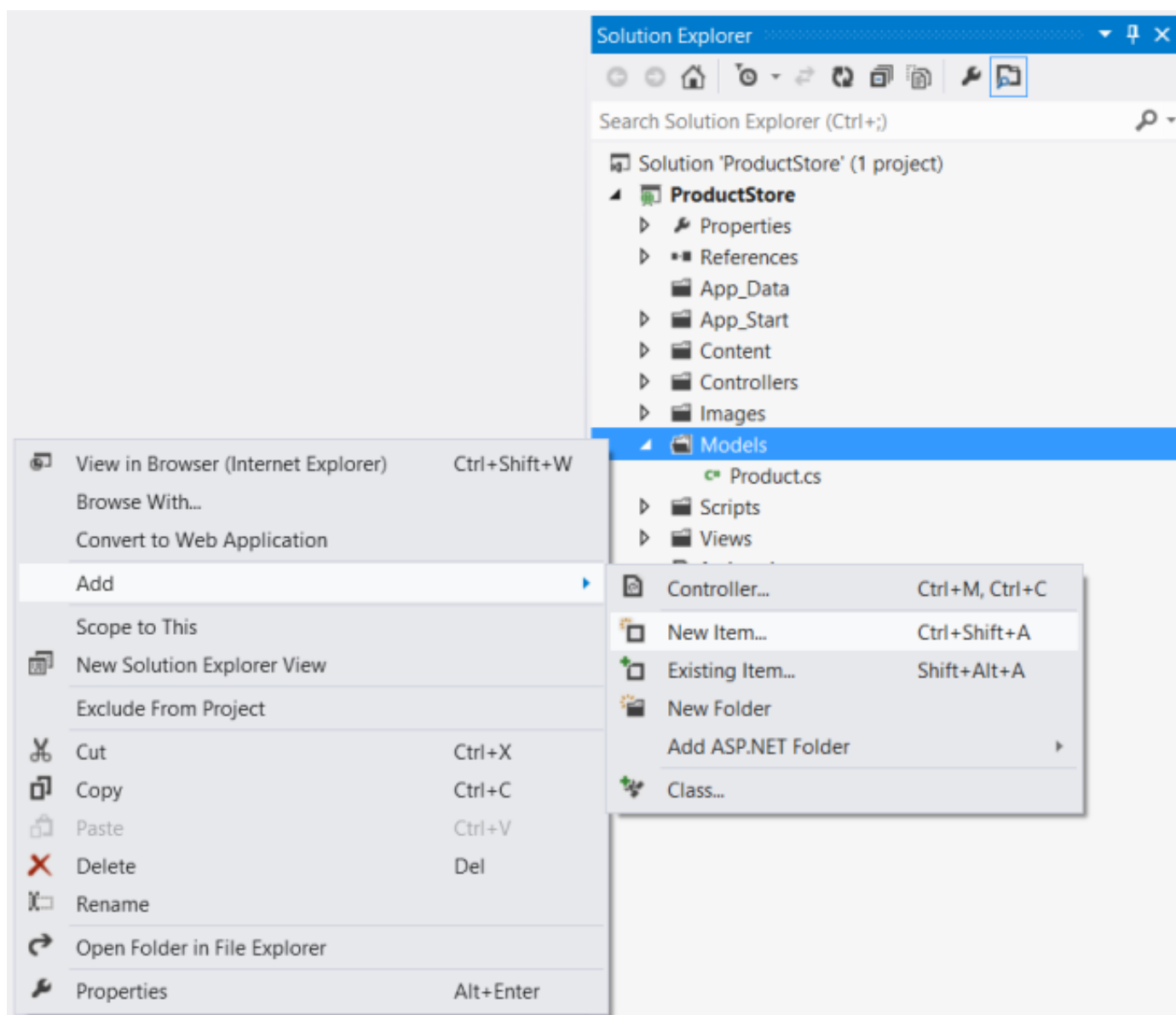
حال خواص زیر را به این کلاس اضافه کنید.

```
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; }
    }
}
```

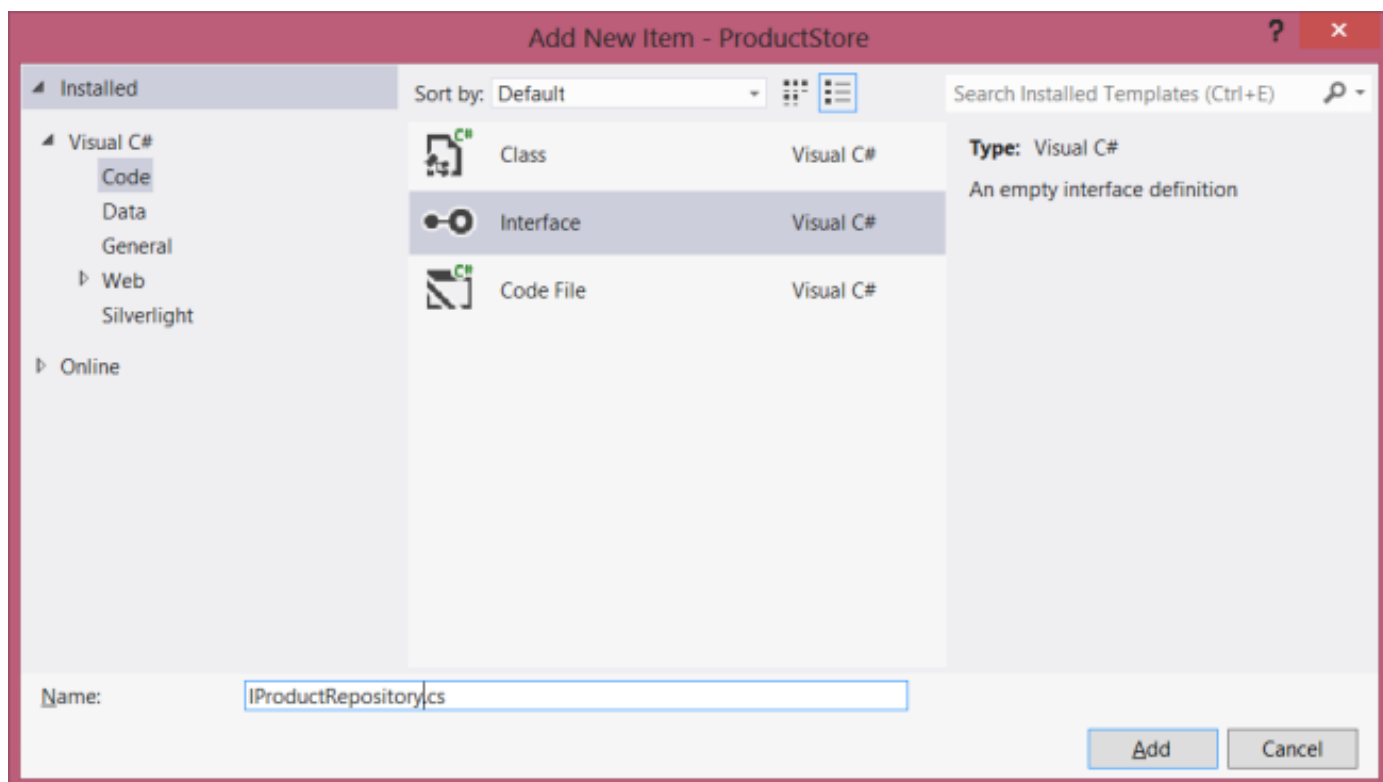
### افزودن یک مخزن

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

```

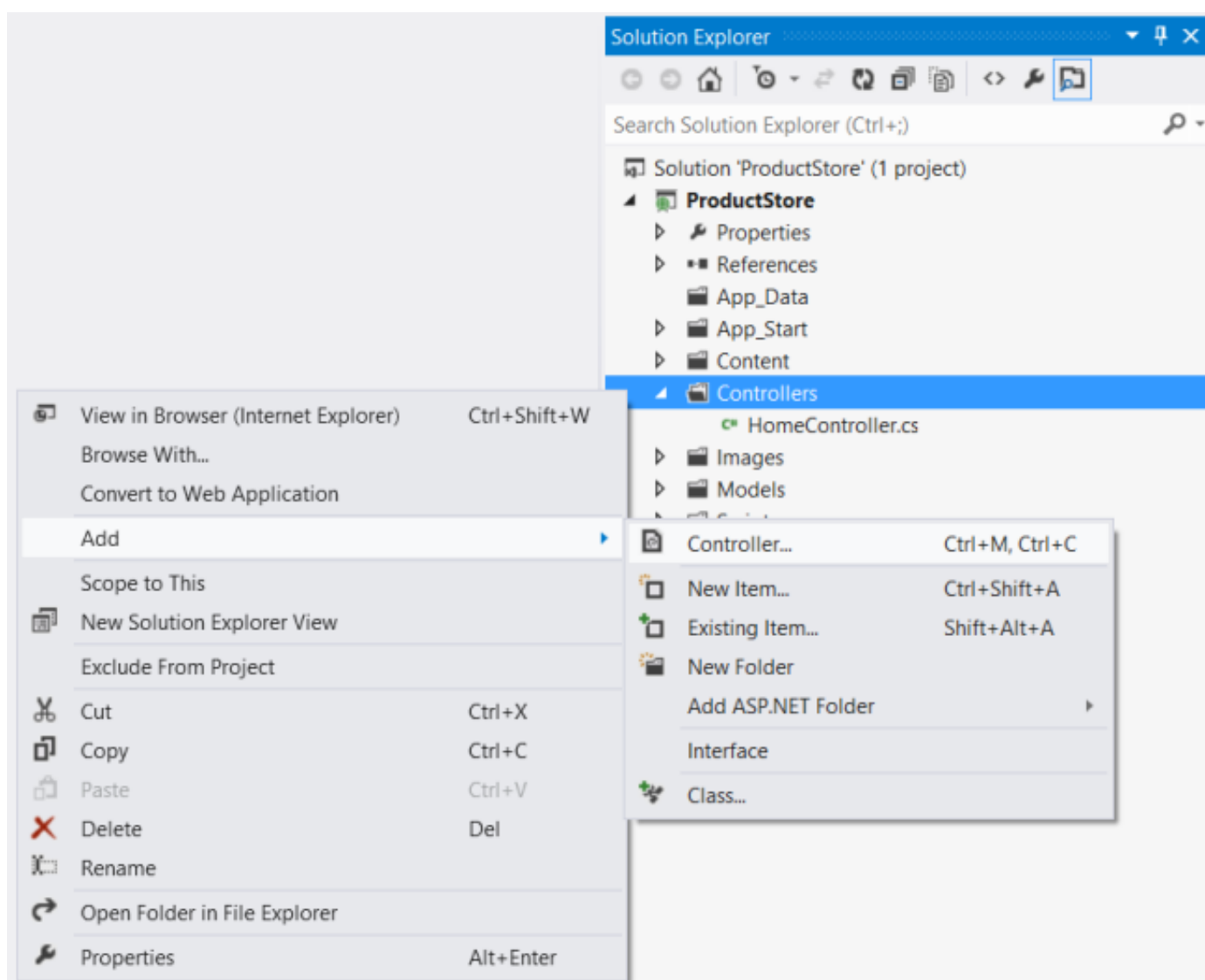
مخزن ما لیست محصولات را در حافظه محلی نگهداری می کند. برای مثال جاری این طراحی کافی است، اما در یک اپلیکیشن واقعی داده های شما در یک دیتابیس یا منبع داده ابری ذخیره خواهند شد. همچنین استفاده از الگوی مخزن، تغییر منبع داده ها در آینده را راحت تر می کند.

### افزودن یک کنترلر Web API

اگر قبلاً با ASP.NET MVC کار کرده باشید، با مفهوم کنترلرها آشنایی دارید. در ASP.NET Web API کنترلرها کلاس هایی هستند که درخواست های HTTP دریافتی از کلاینت را به اکشن متدها نگاشت می کنند. ویژوال استودیو هنگام ساختن پروژه شما دو کنترلر به آن اضافه کرده است. برای مشاهده آنها پوشه Controllers را باز کنید. HomeController یک کنترلر مرسوم در ASP.NET MVC است. این کنترلر مسئول بکار گرفتن صفحات وب است و مستقیماً ربطی به Web API ندارد. ValuesController یک کنترلر نمونه WebAPI است.

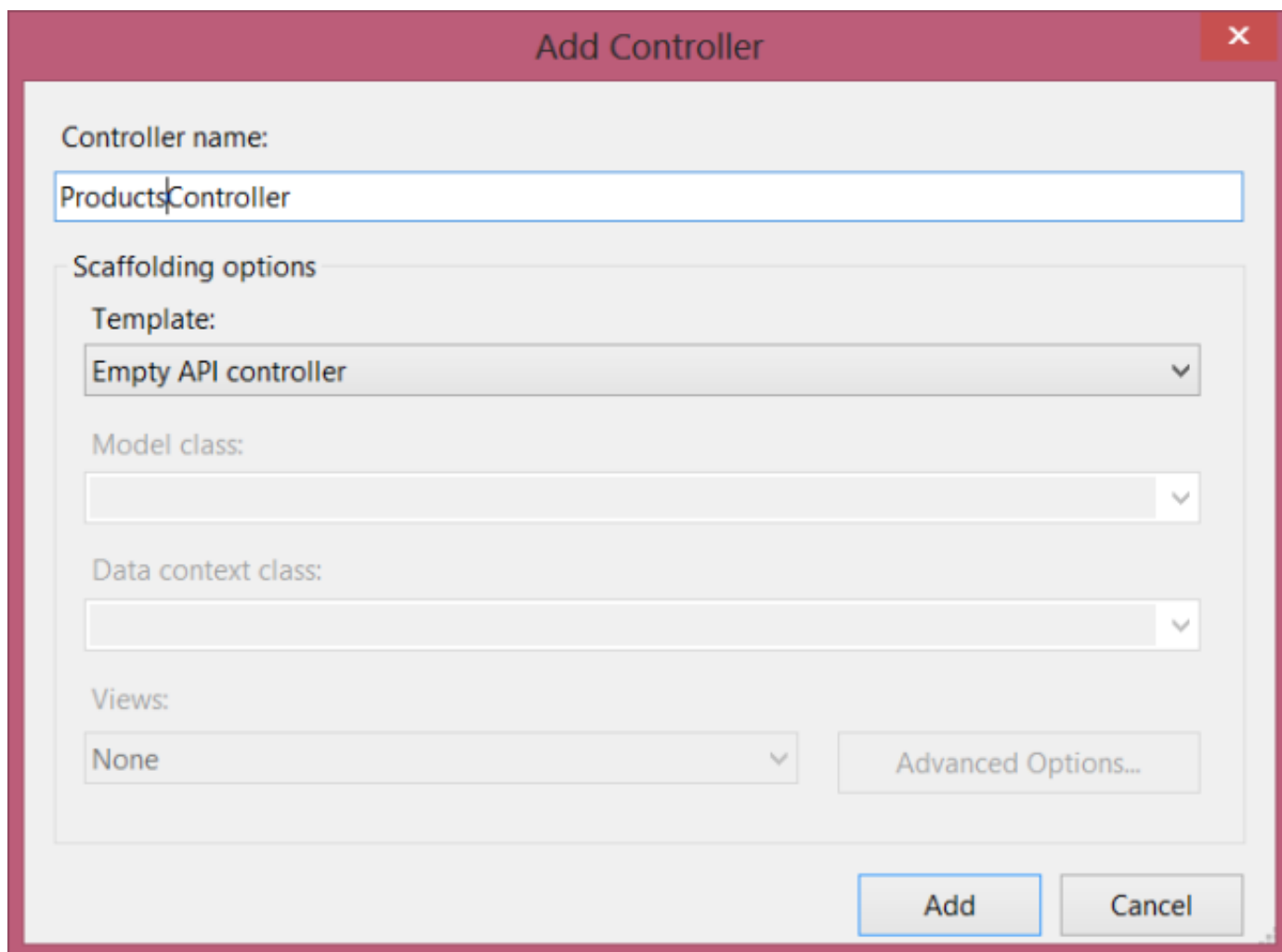
کنترلر ValuesController را حذف کنید، نیازی به این آیتم نخواهیم داشت. حال برای اضافه کردن کنترلری جدید مراحل زیر را دنبال کنید.

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 می زند. 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

ProductStore API اکشن های متعددی در قالب متدهای HTTP GET در دسترس قرار می دهد. هر اکشن به متدی در کلاس `ProductsController` مرتبط است.

Action	HTTP Method	Relative URL
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 می شود، پس بر اساس قراردادهای توکار پیش فرض به درخواست های HTTP GET نگاشت خواهد شد. همچنین از آنجا که این متد پارامتری ندارد، به URL ای نگاشت می شود 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 می شود اما پارامتری با نام *id* دارد. این پارامتر به قسمت *id* مسیر درخواست شده (request URL) نگاشت می شود. تبدیل پارامتر به نوع داده مناسب (در اینجا *int*) هم بصورت خودکار توسط فریم ورک ASP.NET Web API انجام می شود.

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

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 می کند پارامترها را با پارامترهای متد کنترلر تطبیق دهد. بنابراین درخواستی به آدرس "api/products?category= category" به این متد نگاشت می شود.

## 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 می دهد.

```
// 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 می شود. برای ساختن محصولی جدید کلاینت یک درخواست HTTP POST ارسال می کند. این متد پارامتری از نوع Product می پذیرد. در Web API پارامترهای پیچیده (complex types) بصورت خودکار با deserialize کردن بدنه درخواست بدست می آیند. 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 می کند، اما هنوز کامل نیست. در حالت ایده آل ما می خواهیم پیام HTTP Response موارد زیر را هم در بر گیرد:

**Response code:** بصورت پیش فرض فرض فریم ورک Web API کد وضعیت را به 200 (OK) تنظیم می کند. اما طبق پروتکل HTTP/1.1 هنگامی که یک درخواست POST منجر به ساخته شدن منبعی جدید می شود، سرور باید با کد وضعیت 201 (Created) پاسخ دهد. **Location:** هنگامی که سرور منبع جدیدی می سازد، باید آدرس منبع جدید را در قسمت Location header پاسخ درج کند.

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

```
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، می توانیم جزئیات پیام HTTP response را کنترل کنیم. مانند تغییر کد وضعیت و مقدار دهی Location header.

متد CreateResponse آبجکتی از نوع HttpResponseMessage می سازد و بصورت خودکار آبجکت Product را مرتب (serialize) کرده و در بدنه پاسخ می نویسد. نکته دیگر آنکه مثال جاری، مدل را اعتبارسنجی نمی کند. برای اطلاعات بیشتر درباره اعتبارسنجی مدل ها 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 می شود، پس Web API آن را به درخواست های HTTP PUT نگاشت خواهد کرد. این متد دو پارامتر می پذیرد، یکی شناسه محصول مورد نظر و دیگری آبجکت محصول آپدیت شده. مقدار پارامتر *id* از مسیر (route) دریافت می شود 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، می تواند کد وضعیت 200 (OK) را به همراه بدنه موجودیتی که وضعیت فعلی را نمایش می دهد برگرداند. اگر عملیات حذف هنوز در حال اجرا است (Pending) می توانید کد 202 (Accepted) یا 204 (No Content) را برگردانید.

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

## نظرات خوانندگان

نویسنده: علی

تاریخ: ۱۳:۴۸ ۱۳۹۲/۱۱/۰۳

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: ۱۷:۵۳ ۱۳۹۲/۱۱/۰۴

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: ۱۷:۳۶ ۱۳۹۲/۱۱/۰۵

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