Robbin Cremers vs Microsoft Technology

For the brave souls who get this far: Be nice to nerds. Chances are you'll end up working for one …

FEB **16** 2012

Building and consuming REST services with ASP.NET Web API using MediaTypeFormatter and OData support

The **ASP.NET Web API** has been released with the **ASP.NET MVC4 beta release**, which was released 2 days ago (14/02/2012).

You can download the ASP.NET MVC4 beta release, that includes the Web API, here: http://www.microsoft.com/download/en/details.aspx?id=28942 (http://www.microsoft.com/download/en/details.aspx?id=28942)

Quoted directly from microsoft website:

> *ASP.NET MVC 4 also includes ASP.NET Web API,* ***a framework for building and consuming HTTP services that can reach a broad range of clients including browsers, phones, and tablets. ASP.NET Web API is great for building services that follow the REST architectural style****, plus it supports RPC patterns.*

If you do not know what REST stands for and why it could be of any use, you can watch this 1h18m08s video on channel9 by Aaron Skonnard: http://channel9.msdn.com/Blogs/matthijs/Why-REST-by-Aaron-Skonnard (http://channel9.msdn.com/Blogs/matthijs/Why-REST-by-Aaron-Skonnard)

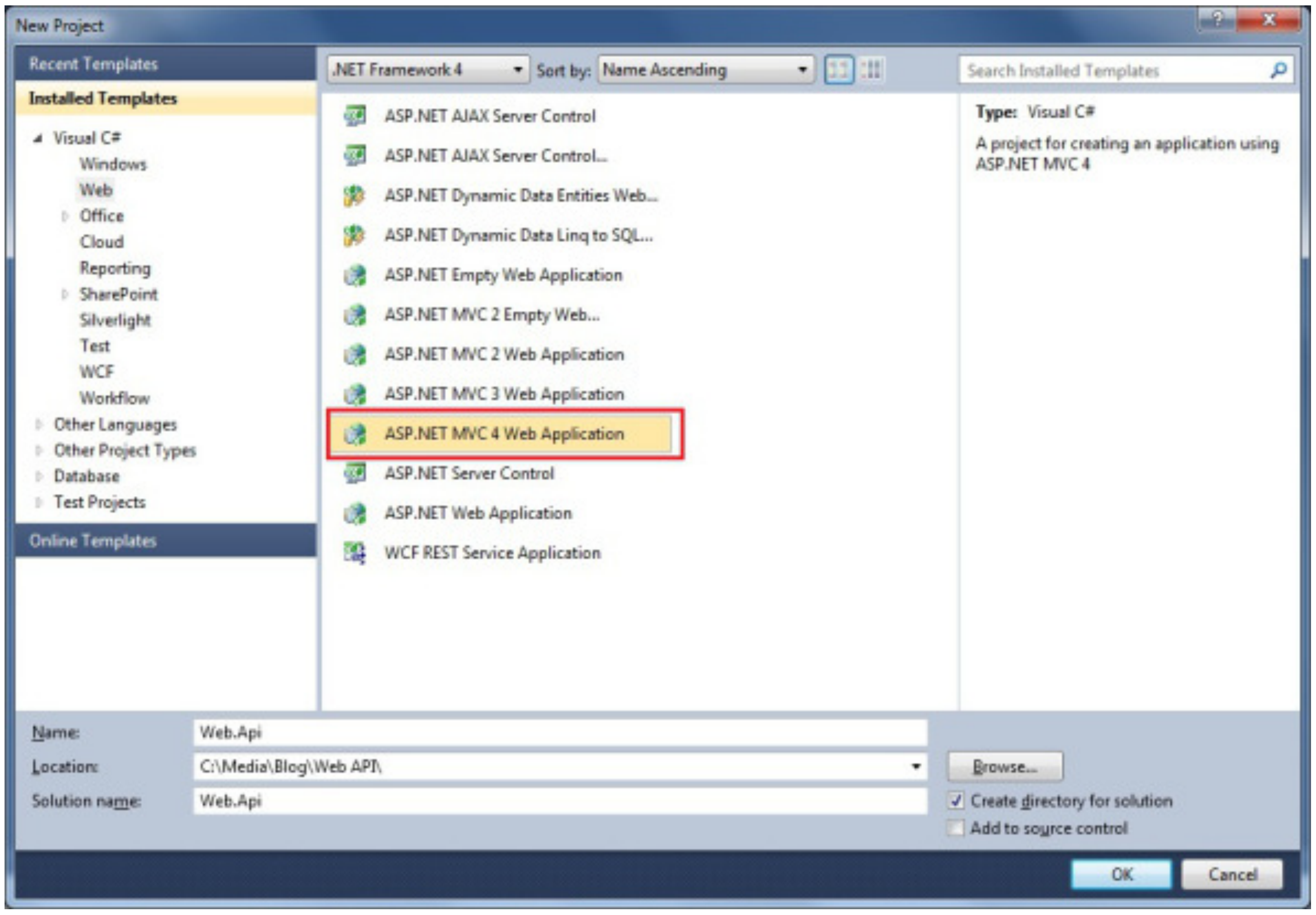
Considering how popular REST is these days, it might be interesting to cover the new Web API in this post. Originally we saw REST services coming up through the WCF pipeline, like the WCF Data Services or using a common WCF service with the WebHttpBinding, which works on HTTP verbs like **GET, POST, PUT and DELETE.** However WCF was created as a messaging platform, on which we are working with SOAP messages. The entire WCF pipeline is also optimized for messaging. REST services do work a bit differently, nor do they use any SOAP. Apparently Microsoft came to the conclusion that the integration of REST was not ideal with the WCF messaging pipeline so they moved the possibility to create REST services within the ASP.NET Platform.

I already wrote some posts on this blog before about REST services:
WCF REST service with XML / JSON response format according to Content-Type header (http://robbincremers.me/2012/01/05/wcf-rest-service-with-xml-json-response-format-according-to-content-type-header/)
WCF REST service operation with JSON and XML also supporting ATOM syndication feed format (http://robbincremers.me/2012/01/05/wcf-rest-service-with-atom-syndication-feed/)
WCF REST service with ODATA and Entity Framework with client context, custom operations and operation interceptors (http://robbincremers.me/2012/01/24/wcf-rest-service-with-odata-and-entity-framework-with-client-context-custom-operations-and-operation-interceptors/)

It might be useful to browse once quickly through these posts, so you understand what REST is, how it works and how **content negotation** works,and how REST currently is implemented by WCF. I will not cover the HTTP verbs nor will I go into detail into content negotation. You can find these basics in the previous posts I've written. I will simply cover the ASP.NET Web API basics.

1. Creating a ASP.NET Web API service

After you installed the ASP.NET MVC4 Beta on your computer, create a new ASP.NET MVC4 Web Application:


New Project
Recent Templates
Installed Templates
Visual C#
Windows
Web
Office
Cloud
Reporting
SharePoint
Silverlight
Test
WCF
Workflow
Other Languages
Other Project Types
Database
Test Projects
Online Templates
.NET Framework 4
Sort by: Name Ascending
Search Installed Templates
ASP.NET AJAX Server Control
ASP.NET AJAX Server Control...
ASP.NET Dynamic Data Entities Web...
ASP.NET Dynamic Data Linq to SQL...
ASP.NET Empty Web Application
ASP.NET MVC 2 Empty Web...
ASP.NET MVC 2 Web Application
ASP.NET MVC 3 Web Application
ASP.NET MVC 4 Web Application
ASP.NET Server Control
ASP.NET Web Application
WCF REST Service Application
Browse...
OK
Cancel

Type: Visual C#

A project for creating an application using ASP.NET MVC 4

Name: Web.Api

Location: C:\Media\Blog\Web API\

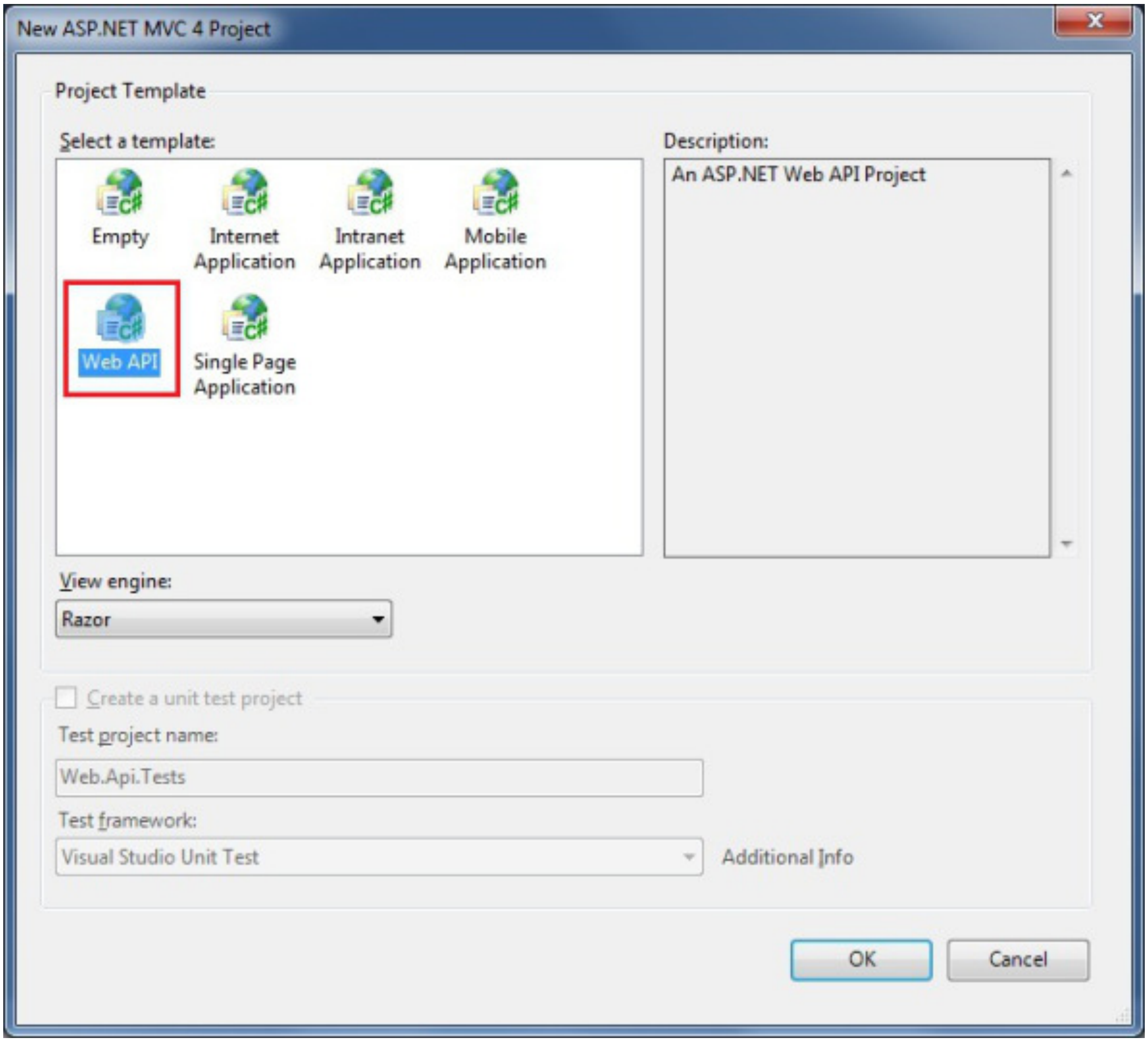
Solution name: Web.Api

[✓] Create directory for solution

[ ] Add to source control

(http://robbincremers.files.wordpress.com/2012/02/12.jpg)

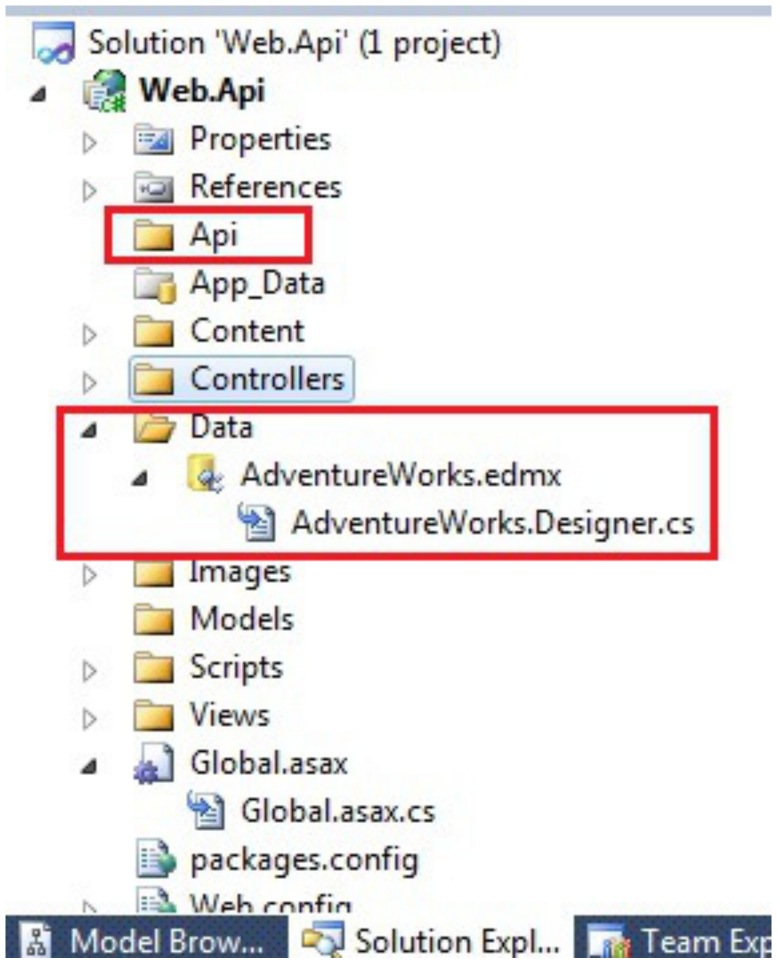
On the next screen, you will see some extra templates are available in comparison to ASP.NET MVC3:



(http://robbincremers.files.wordpress.com/2012/02/21.jpg)

In our case, we will create a Web API project. After creation your project will look like a common ASP.NET MVC project.
We will expose a REST service that will expose some data to our application, clients or mobile users for example.

Our solution will look like this:

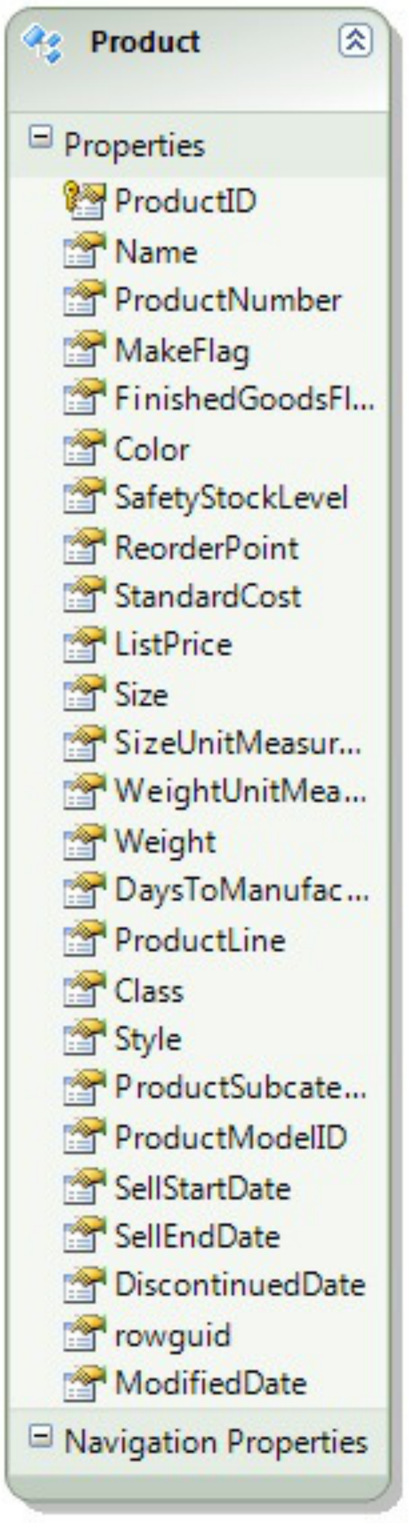


(http://robbincremers.files.wordpress.com/2012/02/31.jpg)

To have some data to work with, I've added an ADO.NET Entity Data Model on the AdventureWorks database. If you do not know what Entity Framework is or how to work with it, you might want to browse to some topics I've writting on Entity Framework: http://robbincremers.me/category/entity-framework (http://robbincremers.me/category/entity-framework/)/

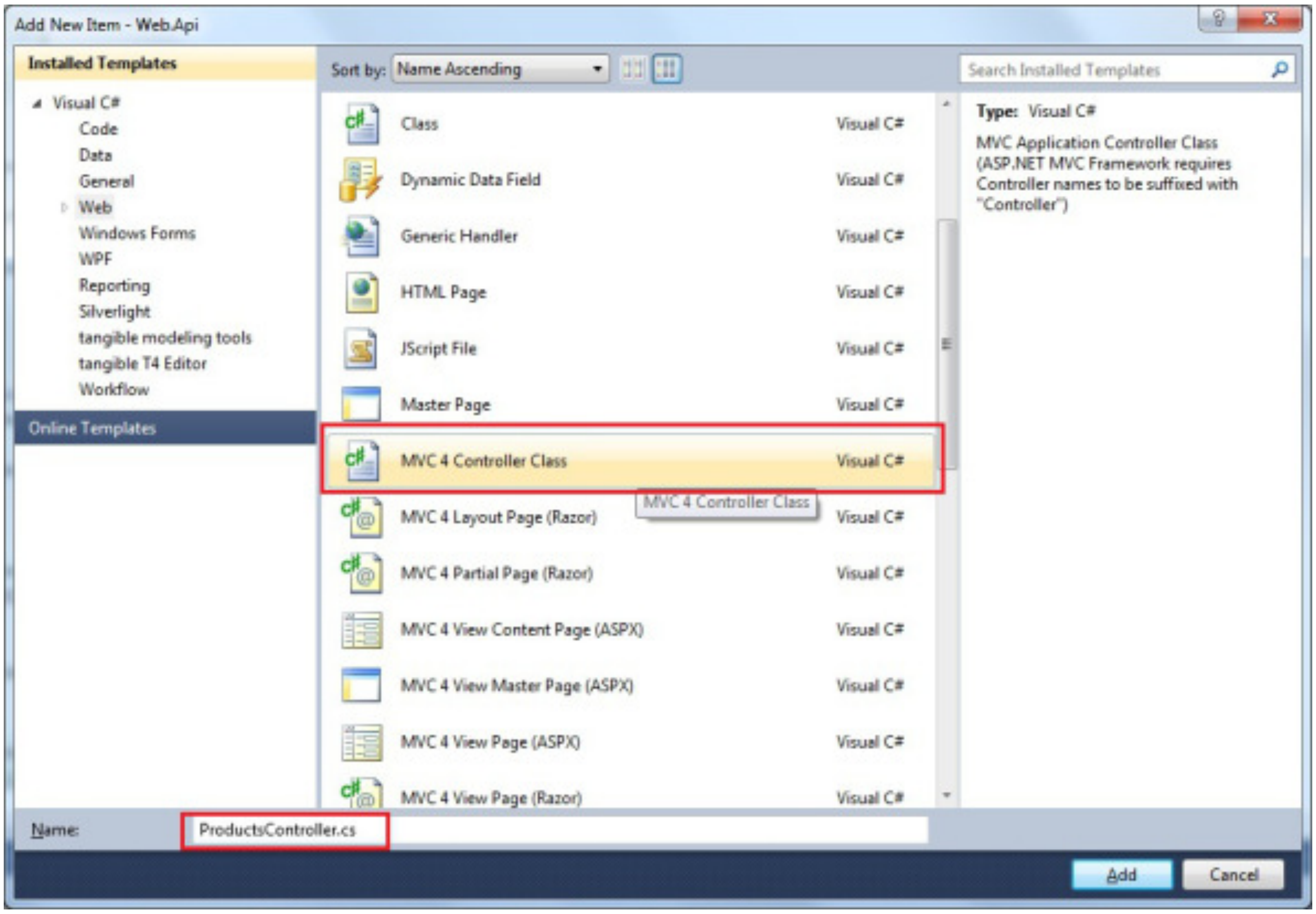
We also added a folder to our solution called ***"Api"* in which we will place the controllers which expose data**. By default under your Controllers folder you will have a file called ValuesController, which is the default template for a Web Api Controller. In our case, we simply remove the file and we will create our own Web API Controller under the API folder.

In our AdventureWorks Entity Data Model we exposed 1 class Product from our AdventureWorks database:



(http://robbincremers.files.wordpress.com/2012/02/41.jpg)

Right-click the Api folder, under which all our Web API controllers will be placed, and add new item and we will add a MVC4 Controller Class, called *"ProductsController"*:


Add New Item - Web.Api

Installed Templates

Sort by: Name Ascending

Search Installed Templates

Visual C#
Code
Data
General
Web
Windows Forms
WPF
Reporting
Silverlight
tangible modeling tools
tangible T4 Editor
Workflow

Online Templates

Class	Visual C#
Dynamic Data Field	Visual C#
Generic Handler	Visual C#
HTML Page	Visual C#
JScript File	Visual C#
Master Page	Visual C#
MVC 4 Controller Class	Visual C#
MVC 4 Layout Page (Razor)	Visual C#
MVC 4 Partial Page (Razor)	Visual C#
MVC 4 View Content Page (ASPX)	Visual C#
MVC 4 View Master Page (ASPX)	Visual C#
MVC 4 View Page (ASPX)	Visual C#
MVC 4 View Page (Razor)	Visual C#

MVC 4 Controller Class

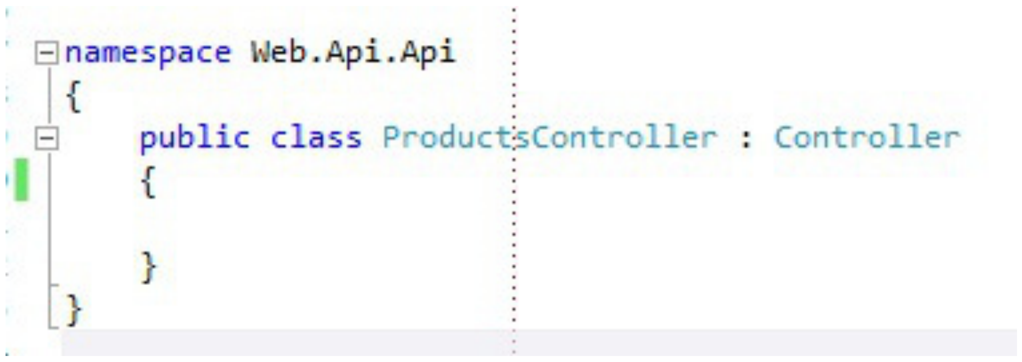
Type: Visual C#

MVC Application Controller Class (ASP.NET MVC Framework requires Controller names to be suffixed with "Controller")

Name: ProductsController.cs

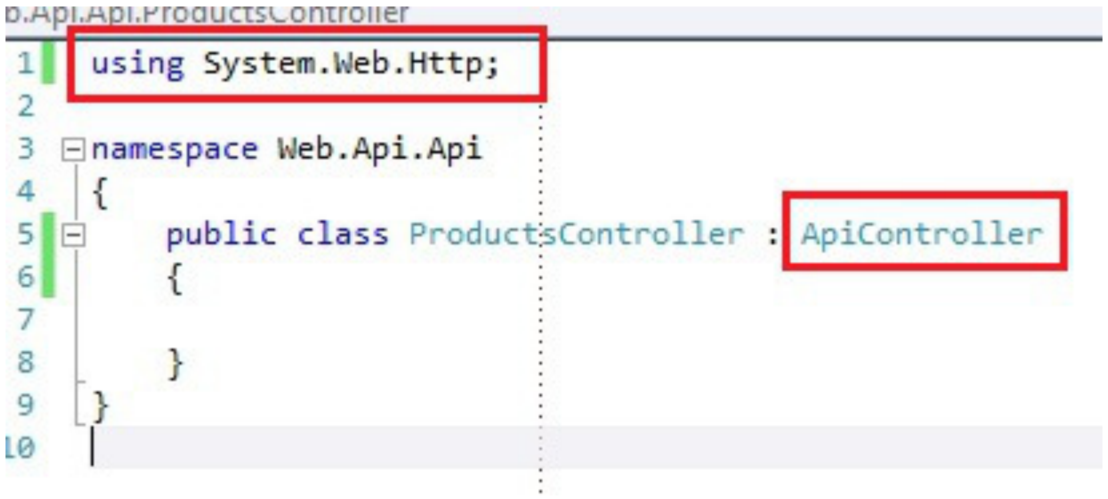
(http://robbincremers.files.wordpress.com/2012/02/51.jpg)

By default, our new controller will inherit from Controller:


```
namespace Web.Api.Api
{
    public class ProductsController : Controller
    {

    }
}
```

(http://robbincremers.files.wordpress.com/2012/02/61.jpg)

To work with a Web API controller, your controller will need to derive from the **ApiController**:

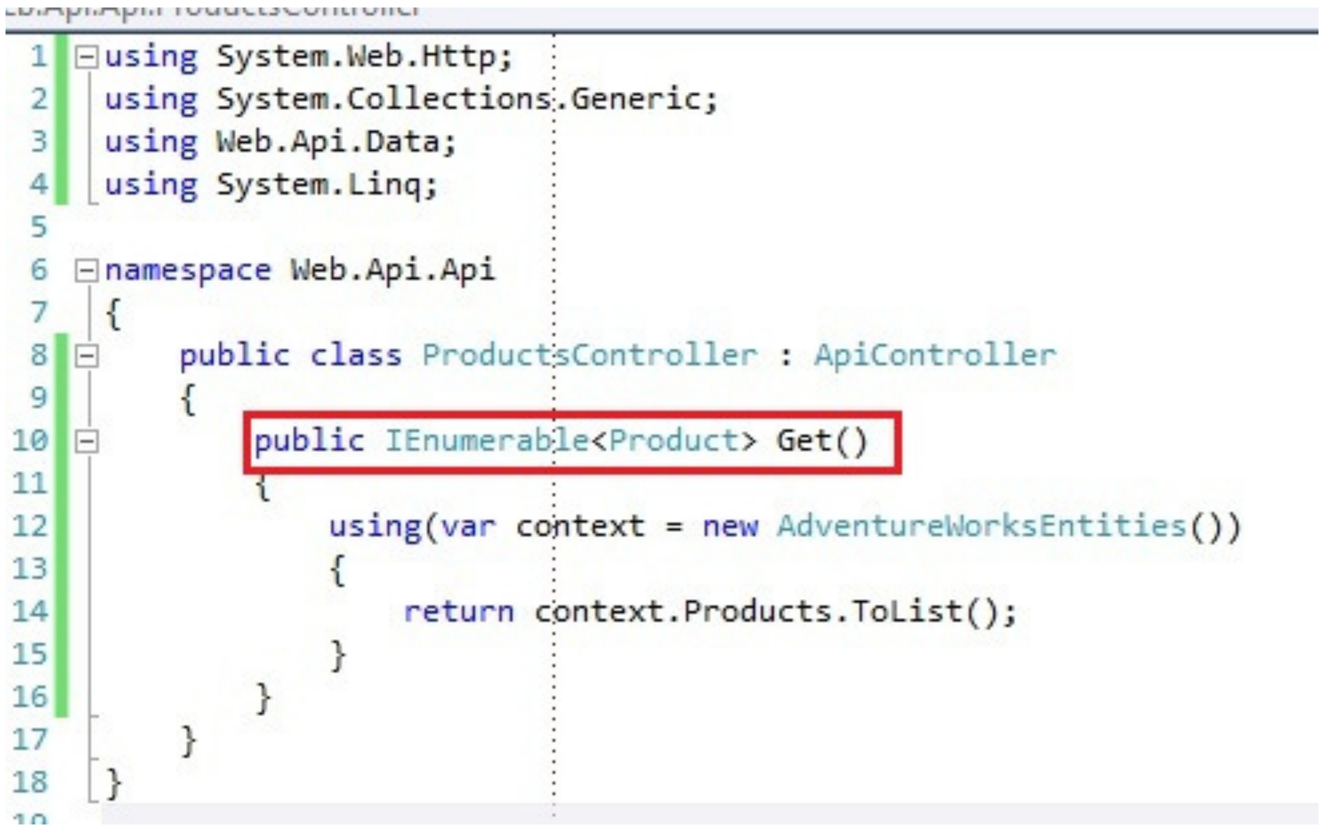


```
using System.Web.Http;

namespace Web.Api.Api
{
    public class ProductsController : ApiController
    {

    }
}
```

(http://robbincremers.files.wordpress.com/2012/02/71.jpg)

The new ApiController and related items for the ASP.NET Web API can be found under the **System.Web.Http** namespace.

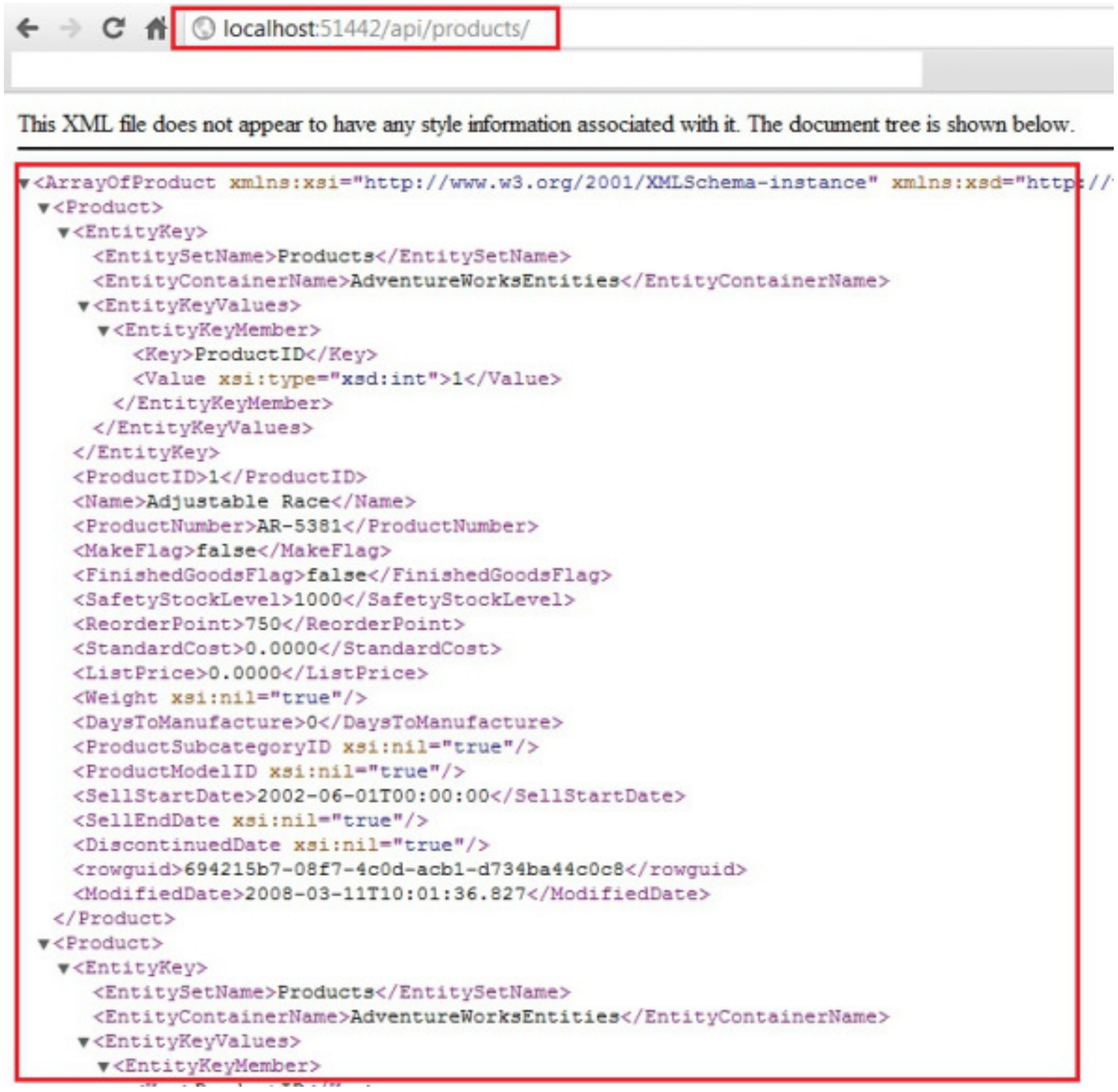
REST services work based on the HTTP verbs like GET, POST etc. **The new ASP.NET Web API is convention based**:



```
using System.Web.Http;
using System.Collections.Generic;
using Web.Api.Data;
using System.Linq;

namespace Web.Api.Api
{
    public class ProductsController : ApiController
    {
        public IEnumerable<Product> Get()
        {
            using(var context = new AdventureWorksEntities())
            {
                return context.Products.ToList();
            }
        }
    }
}
```

(http://robbincremers.files.wordpress.com/2012/02/81.jpg)

We want to expose an operation of the Products that is a GET Operation, which will expose the entire list of products to our clients. Since ASP.NET Web API is convention based, the name of the method has to match the HTTP verb you want to invoke or have a method name that starts with the HTTP verb, like for example GetProduct would work aswell. In our case we want to expose a GET operation on the /products/ uri. Invoking this GET operation from the browser:

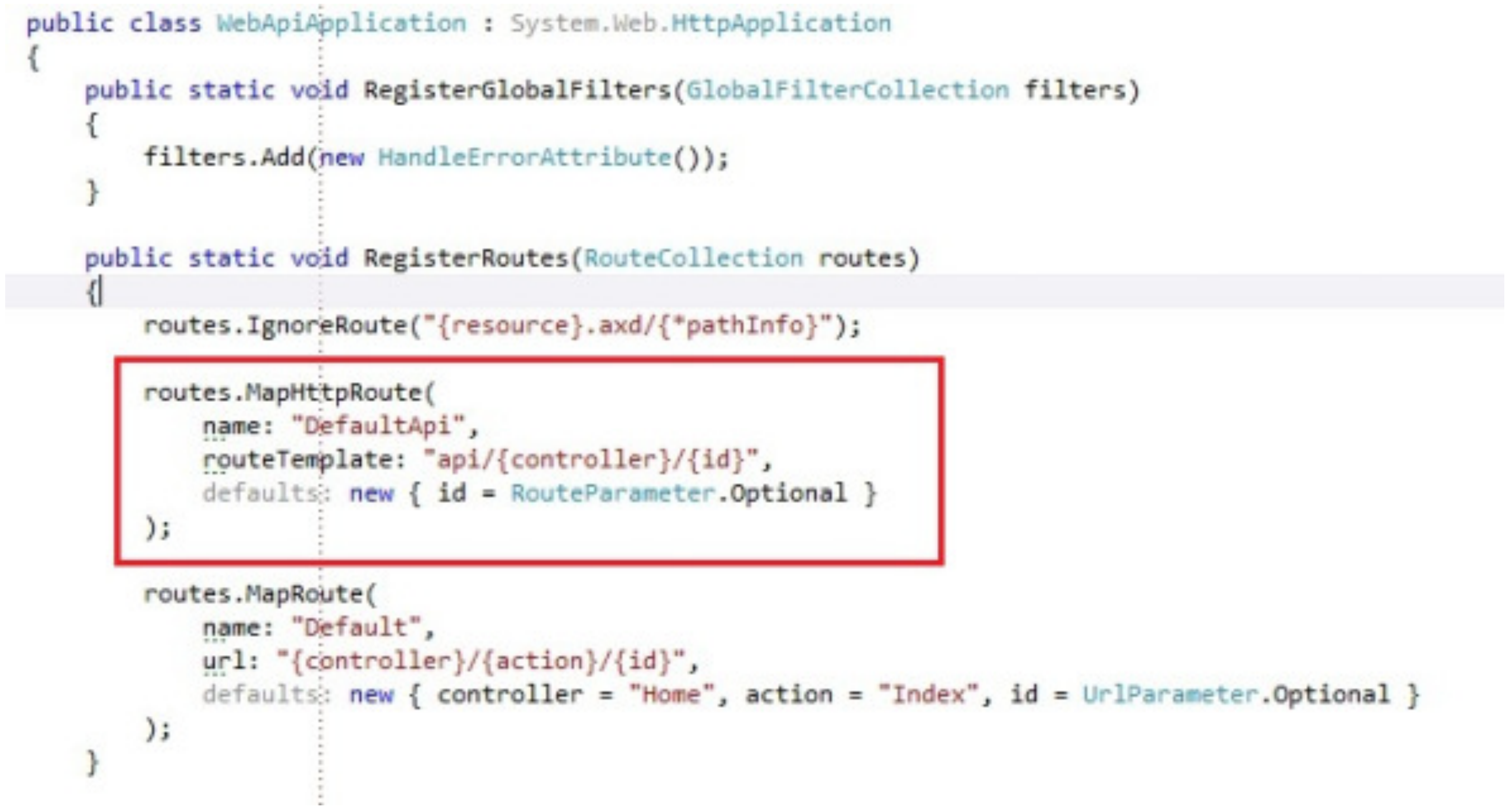


This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<ArrayOfProduct xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
  ▼<Product>
    ▼<EntityKey>
       <EntitySetName>Products</EntitySetName>
       <EntityContainerName>AdventureWorksEntities</EntityContainerName>
      ▼<EntityKeyValues>
        ▼<EntityKeyMember>
           <Key>ProductID</Key>
           <Value xsi:type="xsd:int">1</Value>
         </EntityKeyMember>
       </EntityKeyValues>
     </EntityKey>
     <ProductID>1</ProductID>
     <Name>Adjustable Race</Name>
     <ProductNumber>AR-5381</ProductNumber>
     <MakeFlag>false</MakeFlag>
     <FinishedGoodsFlag>false</FinishedGoodsFlag>
     <SafetyStockLevel>1000</SafetyStockLevel>
     <ReorderPoint>750</ReorderPoint>
     <StandardCost>0.0000</StandardCost>
     <ListPrice>0.0000</ListPrice>
     <Weight xsi:nil="true"/>
     <DaysToManufacture>0</DaysToManufacture>
     <ProductSubcategoryID xsi:nil="true"/>
     <ProductModelID xsi:nil="true"/>
     <SellStartDate>2002-06-01T00:00:00</SellStartDate>
     <SellEndDate xsi:nil="true"/>
     <DiscontinuedDate xsi:nil="true"/>
     <rowguid>694215b7-08f7-4c0d-acb1-d734ba44c0c8</rowguid>
     <ModifiedDate>2008-03-11T10:01:36.827</ModifiedDate>
   </Product>
  ▼<Product>
    ▼<EntityKey>
       <EntitySetName>Products</EntitySetName>
       <EntityContainerName>AdventureWorksEntities</EntityContainerName>
      ▼<EntityKeyValues>
        ▼<EntityKeyMember>
```

(http://robbincremers.files.wordpress.com/2012/02/91.jpg)

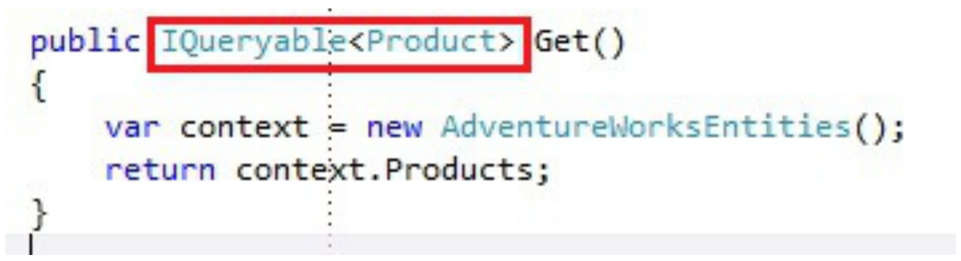
Notice if we browse to the /api/products/ location we directly invoke the GET operation which returns a list of all products present in our database. The /api/ prefix in our uri is defined in the global.asax. By default the template will add the /api/ for the exposed web api services:



```
public class WebApiApplication : System.Web.HttpApplication
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        filters.Add(new HandleErrorAttribute());
    }

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

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

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

(http://robbincremers.files.wordpress.com/2012/02/10.jpg)

You can change this to anything you like. Changing mappings for parameters or exposed services can be done through the routing that is possible in ASP.NET. We now exposed an IEnumerable<Product> on our GET operation, however the ASP.NET Web API also supports exposing **IQueryable<T>** on your operations:

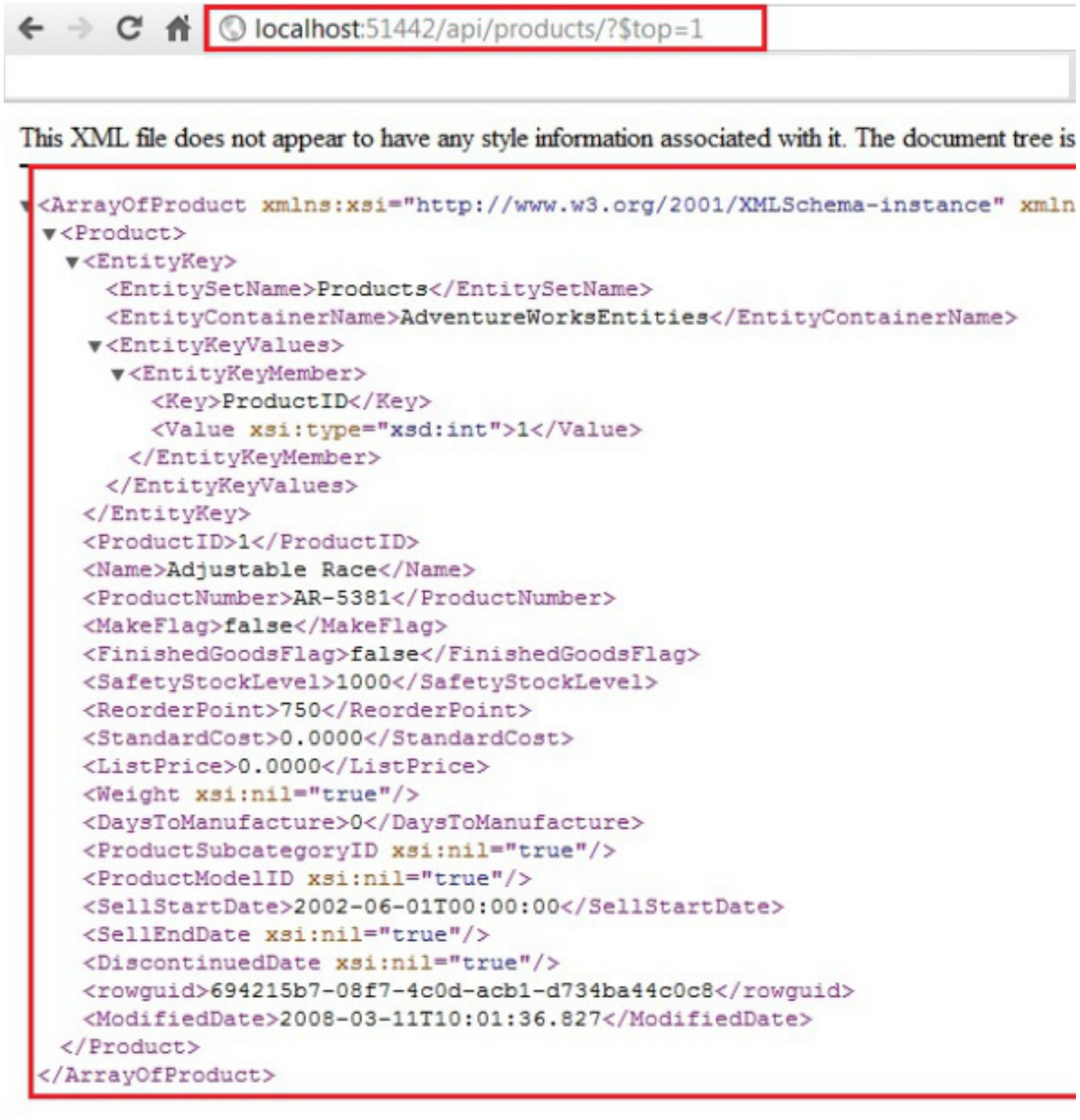


```
public IQueryable<Product> Get()
{
    var context = new AdventureWorksEntities();
    return context.Products;
}
```

(http://robbincremers.files.wordpress.com/2012/02/10_1.jpg)

One of the nice things through exposing an IQueryable is that the client can construct queries with Odata operations and that only the custom query will be invoked on the database, instead of retrieving all products from the database and only applying the filter on the yet retrieved products. However this are basics from the Entity Framework, which I will not cover now. If you do not know what IQueryable is or what lazy loading and deferred loading is, I suggest reading through some of my Entity Framework posts.

The ASP.NET Web API also supports most of the **OData** protocol:



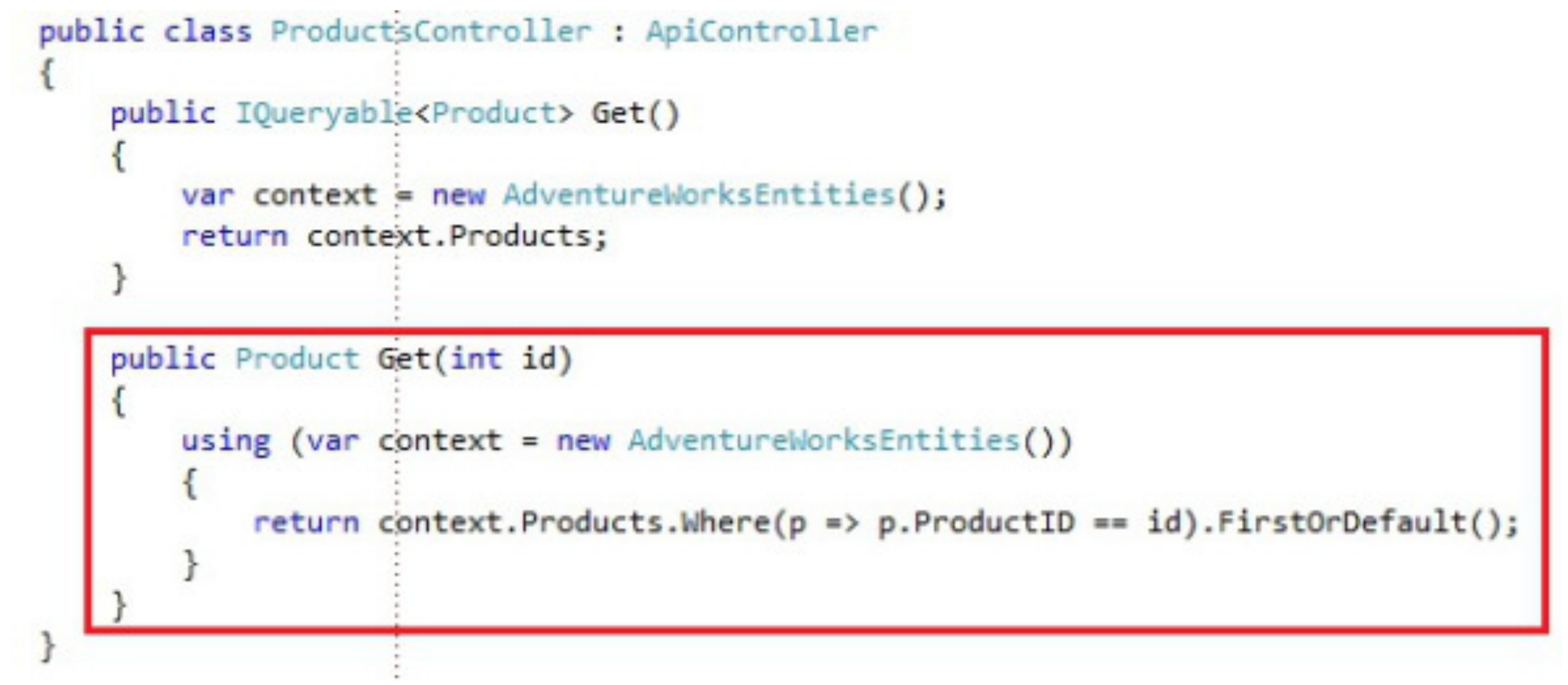
This XML file does not appear to have any style information associated with it. The document tree is

```
<ArrayOfProduct xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmln
 <Product>
   <EntityKey>
     <EntitySetName>Products</EntitySetName>
     <EntityContainerName>AdventureWorksEntities</EntityContainerName>
     <EntityKeyValues>
       <EntityKeyMember>
         <Key>ProductID</Key>
         <Value xsi:type="xsd:int">1</Value>
       </EntityKeyMember>
     </EntityKeyValues>
   </EntityKey>
   <ProductID>1</ProductID>
   <Name>Adjustable Race</Name>
   <ProductNumber>AR-5381</ProductNumber>
   <MakeFlag>false</MakeFlag>
   <FinishedGoodsFlag>false</FinishedGoodsFlag>
   <SafetyStockLevel>1000</SafetyStockLevel>
   <ReorderPoint>750</ReorderPoint>
   <StandardCost>0.0000</StandardCost>
   <ListPrice>0.0000</ListPrice>
   <Weight xsi:nil="true"/>
   <DaysToManufacture>0</DaysToManufacture>
   <ProductSubcategoryID xsi:nil="true"/>
   <ProductModelID xsi:nil="true"/>
   <SellStartDate>2002-06-01T00:00:00</SellStartDate>
   <SellEndDate xsi:nil="true"/>
   <DiscontinuedDate xsi:nil="true"/>
   <rowguid>694215b7-08f7-4c0d-acb1-d734ba44c0c8</rowguid>
   <ModifiedDate>2008-03-11T10:01:36.827</ModifiedDate>
 </Product>
</ArrayOfProduct>
```

(http://robbincremers.files.wordpress.com/2012/02/121.jpg)

You can use OData operations as $skip, $top, $filter etc. If you do not know about OData, you can find the basics in this post:
WCF REST service with ODATA and Entity Framework with client context, custom operations and operation interceptors (http://robbincremers.me/2012/01/24/wcf-rest-service-with-odata-and-entity-framework-with-client-context-custom-operations-and-operation-interceptors/)

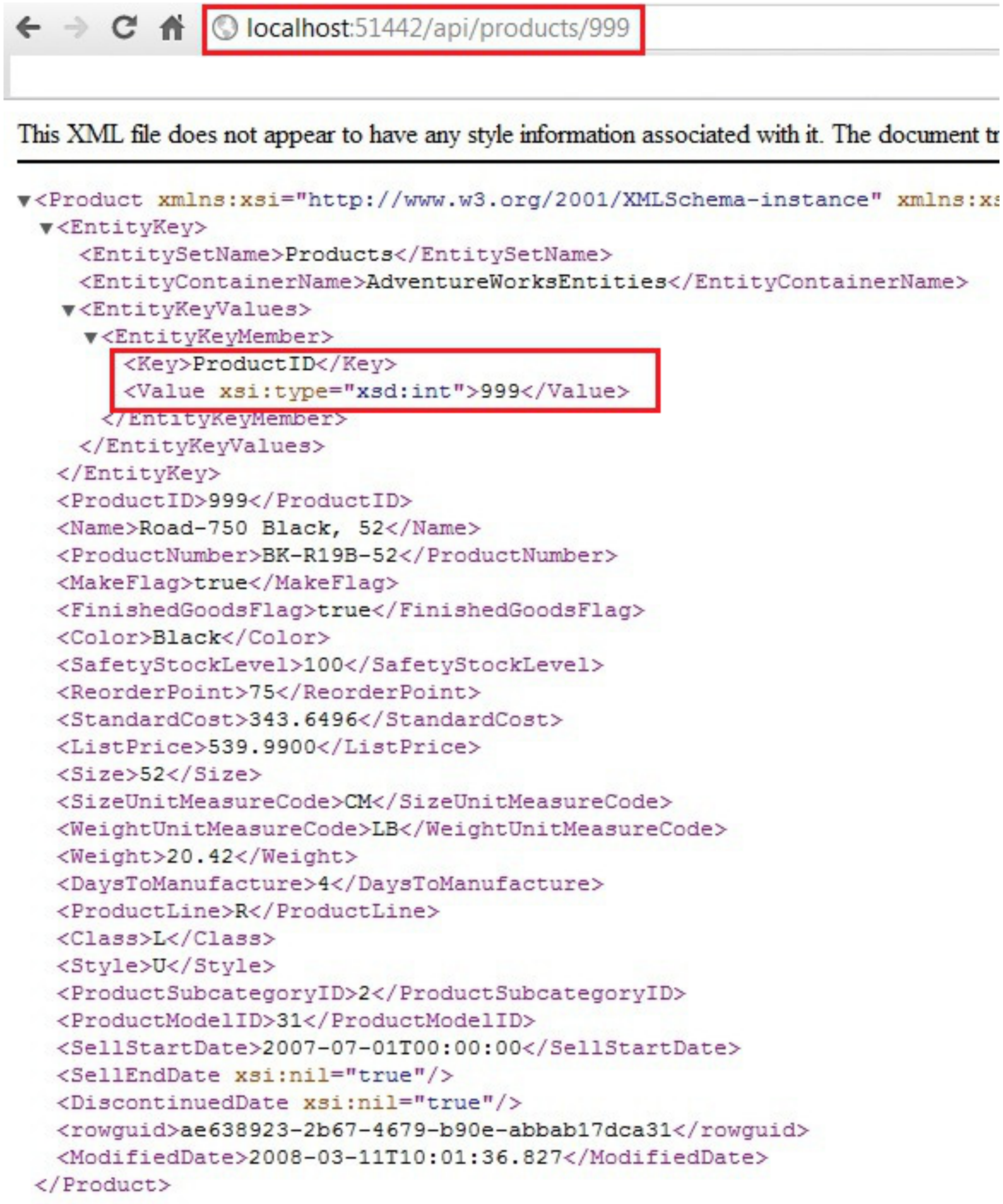
To add a GET operation for a single product, that accepts an id parameter:



```
public class ProductsController : ApiController
{
    public IQueryable<Product> Get()
    {
        var context = new AdventureWorksEntities();
        return context.Products;
    }

    public Product Get(int id)
    {
        using (var context = new AdventureWorksEntities())
        {
            return context.Products.Where(p => p.ProductID == id).FirstOrDefault();
        }
    }
}
```

(http://robbincremers.files.wordpress.com/2012/02/13.jpg)

Note you can also pass along other parameters, you just need to make sure your routing at the global.asax is configured to pass along those parameters. Retrieving a single products through the id:



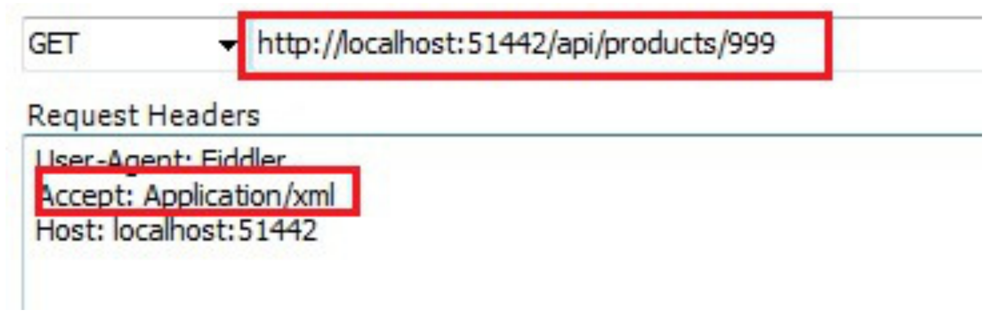
This XML file does not appear to have any style information associated with it. The document tr

```
▼<Product xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs
  ▼<EntityKey>
     <EntitySetName>Products</EntitySetName>
     <EntityContainerName>AdventureWorksEntities</EntityContainerName>
   ▼<EntityKeyValues>
     ▼<EntityKeyMember>
        <Key>ProductID</Key>
        <Value xsi:type="xsd:int">999</Value>
      </EntityKeyMember>
     </EntityKeyValues>
   </EntityKey>
   <ProductID>999</ProductID>
   <Name>Road-750 Black, 52</Name>
   <ProductNumber>BK-R19B-52</ProductNumber>
   <MakeFlag>true</MakeFlag>
   <FinishedGoodsFlag>true</FinishedGoodsFlag>
   <Color>Black</Color>
   <SafetyStockLevel>100</SafetyStockLevel>
   <ReorderPoint>75</ReorderPoint>
   <StandardCost>343.6496</StandardCost>
   <ListPrice>539.9900</ListPrice>
   <Size>52</Size>
   <SizeUnitMeasureCode>CM</SizeUnitMeasureCode>
   <WeightUnitMeasureCode>LB</WeightUnitMeasureCode>
   <Weight>20.42</Weight>
   <DaysToManufacture>4</DaysToManufacture>
   <ProductLine>R</ProductLine>
   <Class>L</Class>
   <Style>U</Style>
   <ProductSubcategoryID>2</ProductSubcategoryID>
   <ProductModelID>31</ProductModelID>
   <SellStartDate>2007-07-01T00:00:00</SellStartDate>
   <SellEndDate xsi:nil="true"/>
   <DiscontinuedDate xsi:nil="true"/>
   <rowguid>ae638923-2b67-4679-b90e-abbab17dca31</rowguid>
   <ModifiedDate>2008-03-11T10:01:36.827</ModifiedDate>
 </Product>
```

(http://robbincremers.files.wordpress.com/2012/02/14.jpg)

ASP.NET Web API also allows you to use **content negotiation to retrieve the format of the data**. Through the **Accept header** you can request for a certain type of data format. If the REST service supports this format, you will get the data in this format back. If it does not, it will return the data in the default format.

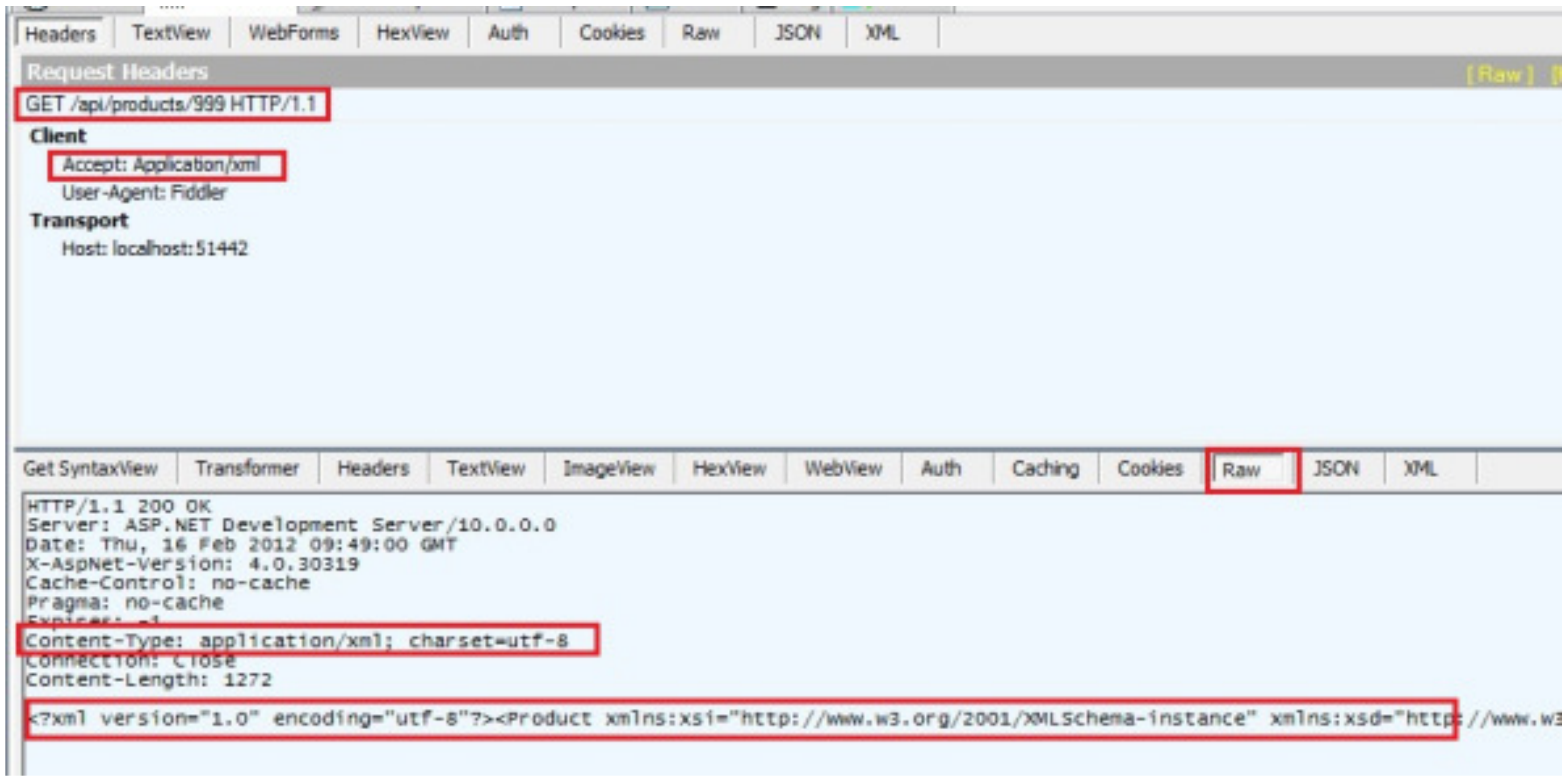
With Fiddler we will request the products with id 999 with a GET operation and pass along the Accept header as “application/xml”. So we are requesting some data and tell the browser that we accept application/xml as content-type. In case the service supports this format, we will get the data back as XML:



```
GET    http://localhost:51442/api/products/999

Request Headers
User-Agent: Fiddler
Accept: Application/xml
Host: localhost:51442
```

(http://robbincremers.files.wordpress.com/2012/02/15.jpg)

The result will be that the data will be returned as XML:

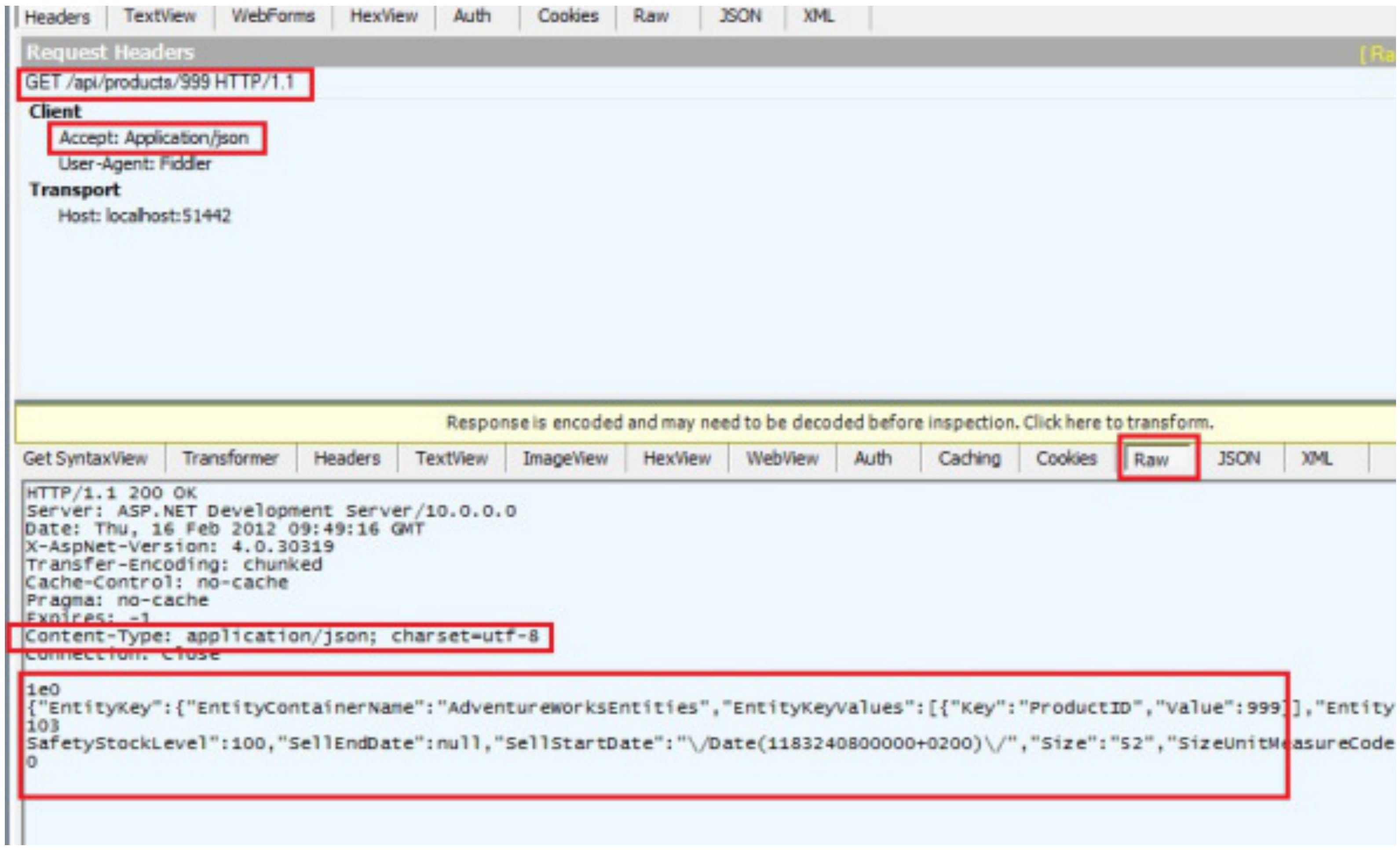


```
GET /api/products/999 HTTP/1.1
Client
    Accept: Application/xml
    User-Agent: Fiddler
Transport
    Host: localhost:51442
```

```
HTTP/1.1 200 OK
Server: ASP.NET Development Server/10.0.0.0
Date: Thu, 16 Feb 2012 09:49:00 GMT
X-AspNet-Version: 4.0.30319
Cache-Control: no-cache
Pragma: no-cache
Expires: -1
Content-Type: application/xml; charset=utf-8
Connection: Close
Content-Length: 1272

<?xml version="1.0" encoding="utf-8"?><Product xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3
```

(http://robbincremers.files.wordpress.com/2012/02/16.jpg)

If we do another GET operation for a specific product and pass the Accept header along as “application/json”, we will get JSON returned:

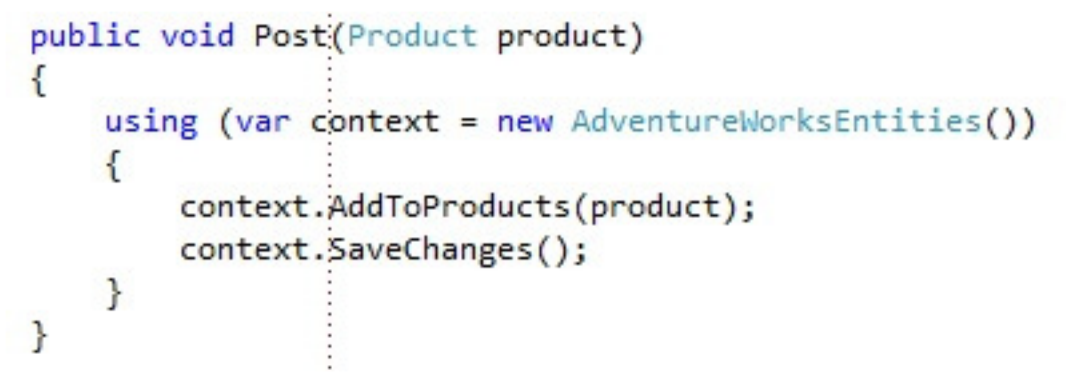


```
GET /api/products/999 HTTP/1.1
Client
    Accept: Application/json
    User-Agent: Fiddler
Transport
    Host: localhost:51442
```

```
HTTP/1.1 200 OK
Server: ASP.NET Development Server/10.0.0.0
Date: Thu, 16 Feb 2012 09:49:16 GMT
X-AspNet-Version: 4.0.30319
Transfer-Encoding: chunked
Cache-Control: no-cache
Pragma: no-cache
Expires: -1
Content-Type: application/json; charset=utf-8
Connection: Close

1e0
{"EntityKey":{"EntityContainerName":"AdventureWorksEntities","EntityKeyValues":[{"Key":"ProductID","Value":999}],"Entity
103
SafetyStockLevel":100,"SellEndDate":null,"SellStartDate":"\/Date(1183240800000+0200)\/","Size":"52","SizeUnitMeasureCode
0
```

(http://robbincremers.files.wordpress.com/2012/02/17.jpg)

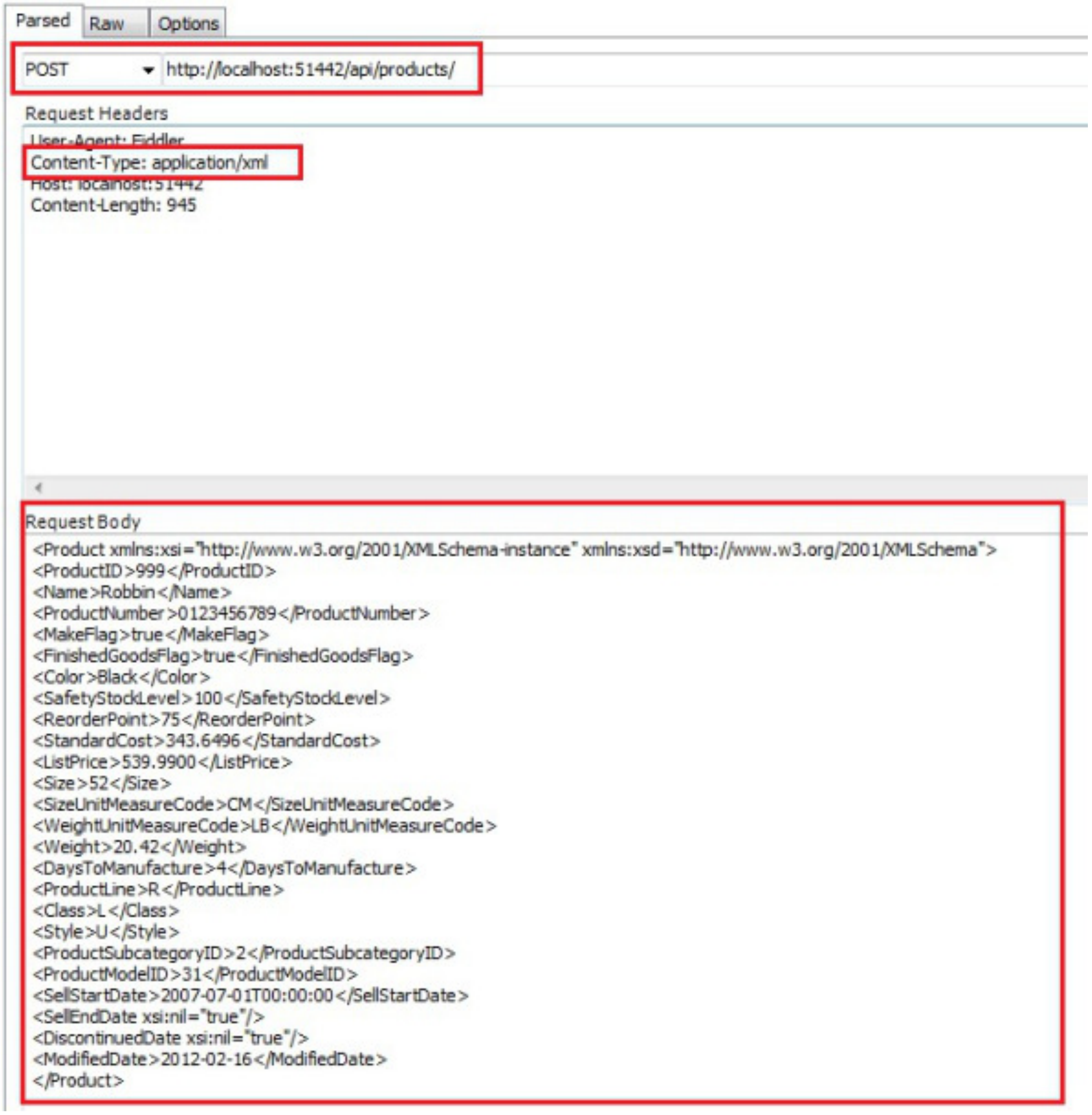
Creating the other operations can be done by using the Post, Put and Delete operations to the controller:



```
public void Post(Product product)
{
    using (var context = new AdventureWorksEntities())
    {
        context.AddToProducts(product);
        context.SaveChanges();
    }
}
```

(http://robbincremers.files.wordpress.com/2012/02/18.jpg)

Invoking the POST operation from the Fiddler client and passing along a new product as XML (Notice we use the POST HTTP verb in Fiddler):



```
POST    http://localhost:51442/api/products/
```

Request Headers

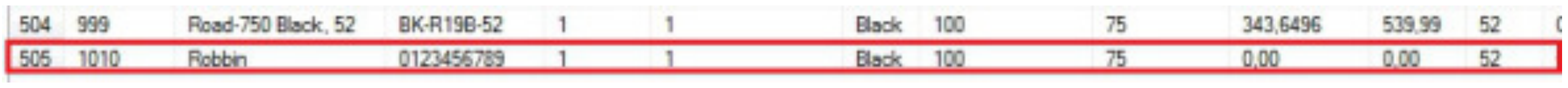
```
User-Agent: Fiddler
Content-Type: application/xml
Host: localhost:51442
Content-Length: 945
```

Request Body

```
<Product xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<ProductID>999</ProductID>
<Name>Robbin</Name>
<ProductNumber>0123456789</ProductNumber>
<MakeFlag>true</MakeFlag>
<FinishedGoodsFlag>true</FinishedGoodsFlag>
<Color>Black</Color>
<SafetyStockLevel>100</SafetyStockLevel>
<ReorderPoint>75</ReorderPoint>
<StandardCost>343.6496</StandardCost>
<ListPrice>539.9900</ListPrice>
<Size>52</Size>
<SizeUnitMeasureCode>CM</SizeUnitMeasureCode>
<WeightUnitMeasureCode>LB</WeightUnitMeasureCode>
<Weight>20.42</Weight>
<DaysToManufacture>4</DaysToManufacture>
<ProductLine>R</ProductLine>
<Class>L</Class>
<Style>U</Style>
<ProductSubcategoryID>2</ProductSubcategoryID>
<ProductModelID>31</ProductModelID>
<SellStartDate>2007-07-01T00:00:00</SellStartDate>
<SellEndDate xsi:nil="true"/>
<DiscontinuedDate xsi:nil="true"/>
<ModifiedDate>2012-02-16</ModifiedDate>
</Product>
```

(http://robbincremers.files.wordpress.com/2012/02/19.jpg)

After invoking the operation, the new product will be added in our database:

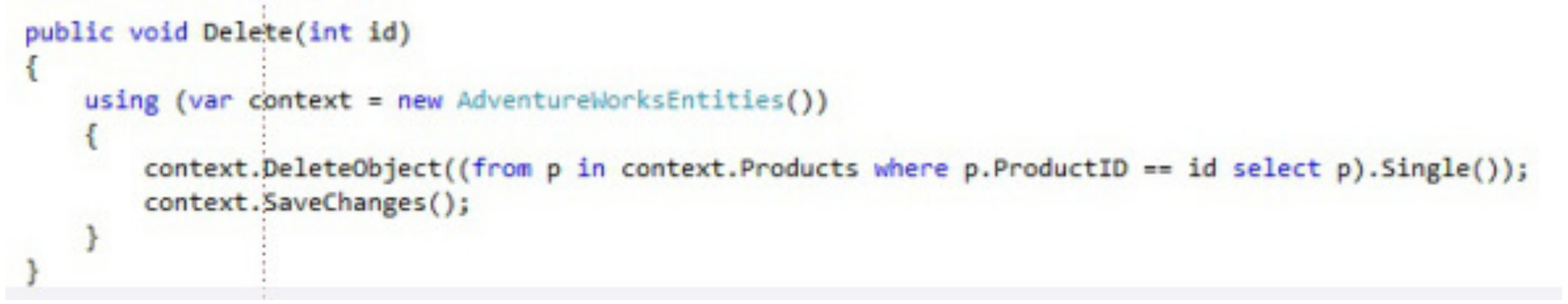


504	999	Road-750 Black, 52	BK-R19B-52	1	1		Black	100	75	343,6496	539,99	52
505	1010	Robbin	0123456789	1	1		Black	100	75	0,00	0,00	52

(http://robbincremers.files.wordpress.com/2012/02/20.jpg)

You could also send products to the service as JSON with the **Content-Type header set as "application/json"**.

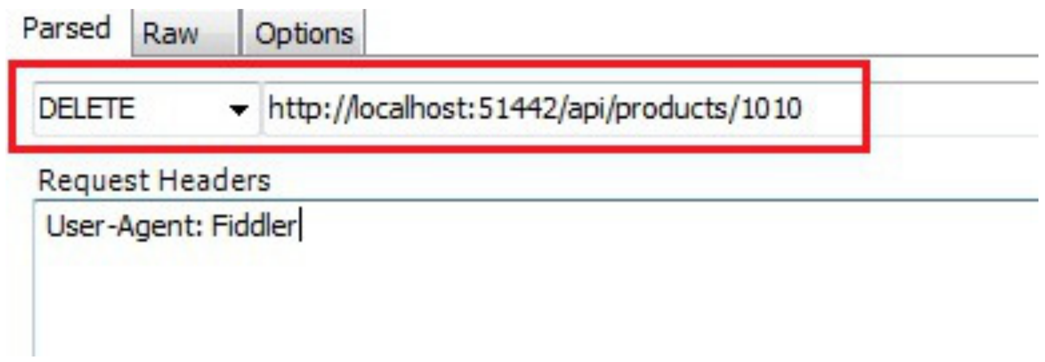
Deleting a products though our Web API service:



```
public void Delete(int id)
{
    using (var context = new AdventureWorksEntities())
    {
        context.DeleteObject((from p in context.Products where p.ProductID == id select p).Single());
        context.SaveChanges();
    }
}
```

(http://robbincremers.files.wordpress.com/2012/02/211.jpg)

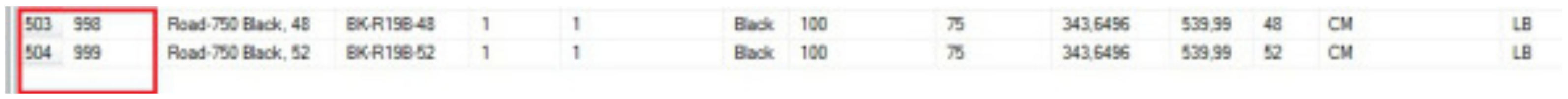
Invoking this operation through Fiddler:



```
DELETE    http://localhost:51442/api/products/1010
Request Headers
User-Agent: Fiddler
```

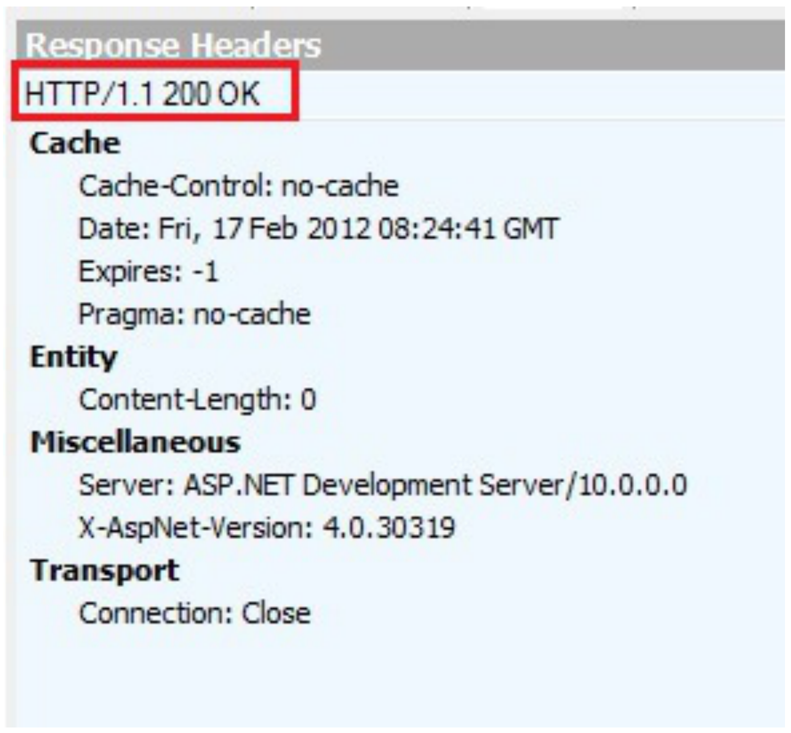
(http://robbincremers.files.wordpress.com/2012/02/22.jpg)

After invocation (Notice the DELETE verb we use in Fiddler), the record is removed again from the database:



503	998	Road-750 Black, 48	BK-R19B-48	1	1	Black	100	75	343,6496	539,99	48	CM	LB
504	999	Road-750 Black, 52	BK-R19B-52	1	1	Black	100	75	343,6496	539,99	52	CM	LB

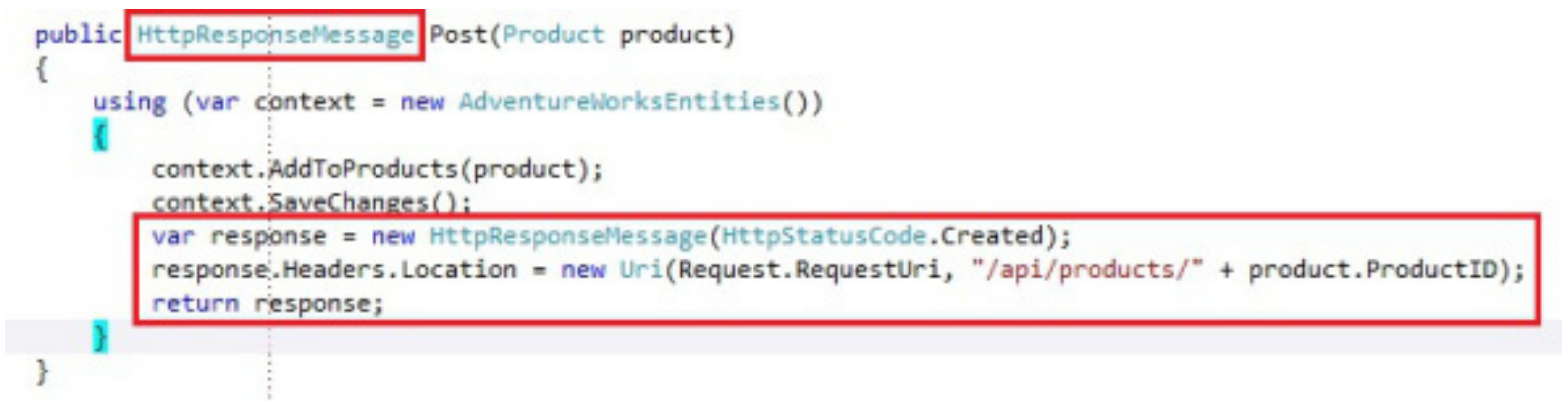
(http://robbincremers.files.wordpress.com/2012/02/23.jpg)
However when building REST services, we should be using the correct HTTP response codes on our operations. If someone invokes the POST operation, an HTTP statuscode of 200 will be returned to the client, which means the operation was successful:



```
Response Headers
HTTP/1.1 200 OK
Cache
    Cache-Control: no-cache
    Date: Fri, 17 Feb 2012 08:24:41 GMT
    Expires: -1
    Pragma: no-cache
Entity
    Content-Length: 0
Miscellaneous
    Server: ASP.NET Development Server/10.0.0.0
    X-AspNet-Version: 4.0.30319
Transport
    Connection: Close
```

(http://robbincremers.files.wordpress.com/2012/02/24.jpg)

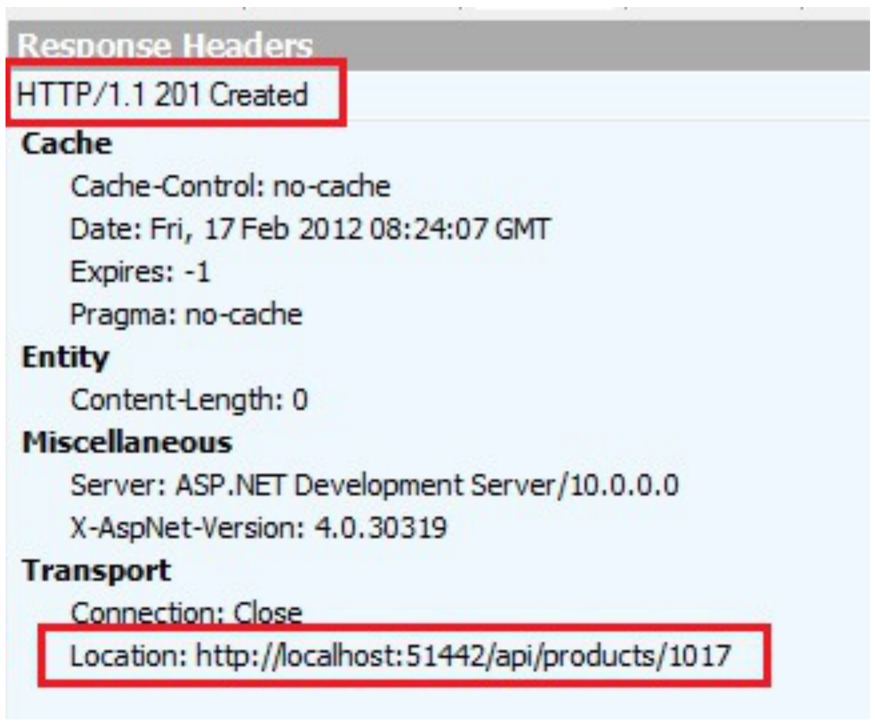
However the correct HTTP statuscode for creation is the HTTP statuscode 201, which stands for created. We can adjust our Post operation so that our operation will return the correct HTTP Statuscode to the client, which indicates the object was created:



```
public HttpResponseMessage Post(Product product)
{
    using (var context = new AdventureWorksEntities())
    {
        context.AddToProducts(product);
        context.SaveChanges();
        var response = new HttpResponseMessage(HttpStatusCode.Created);
        response.Headers.Location = new Uri(Request.RequestUri, "/api/products/" + product.ProductID);
        return response;
    }
}
```

(http://robbincremers.files.wordpress.com/2012/02/25.jpg)

We use an **HttpResponseMessage** as return parameter, in which we set the **Statuscode** to **HttpStatusCode.Created**. We also attach a **Location header** to our response message, indicating what the location is of the newly created product. You will need to reference the **system.net** and **system.net.http** namespaces. If you create a new product by the POST operation:



```
Response Headers
HTTP/1.1 201 Created
Cache
    Cache-Control: no-cache
    Date: Fri, 17 Feb 2012 08:24:07 GMT
    Expires: -1
    Pragma: no-cache
Entity
    Content-Length: 0
Miscellaneous
    Server: ASP.NET Development Server/10.0.0.0
    X-AspNet-Version: 4.0.30319
Transport
    Connection: Close
    Location: http://localhost:51442/api/products/1017
```

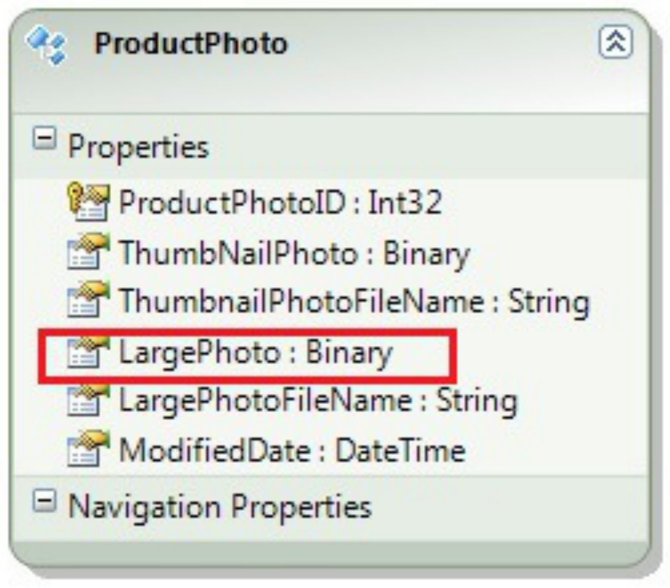
(http://robbincremers.files.wordpress.com/2012/02/26.jpg)

We get the correct HTTP statuscode 201 returned, instead of the default statuscode 200. We also specified a Location header where the new product can be found, which gets returned to the client.

2. Using MediaTypeFormatter to provide content types other then XML or JSON

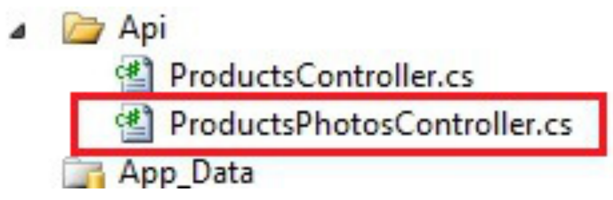
In the AdventureWorks database, each product has a related class ProductPhoto in which a thumbnail and large photo are binary saved in the database. If we want to expose a ProductsPhotos controller which exposes the information of the product photos, we create an ApiController called ProductPhotos:

We import the ProductPhoto from our AdventureWorks database into our ADO.NET Entity Data Model:



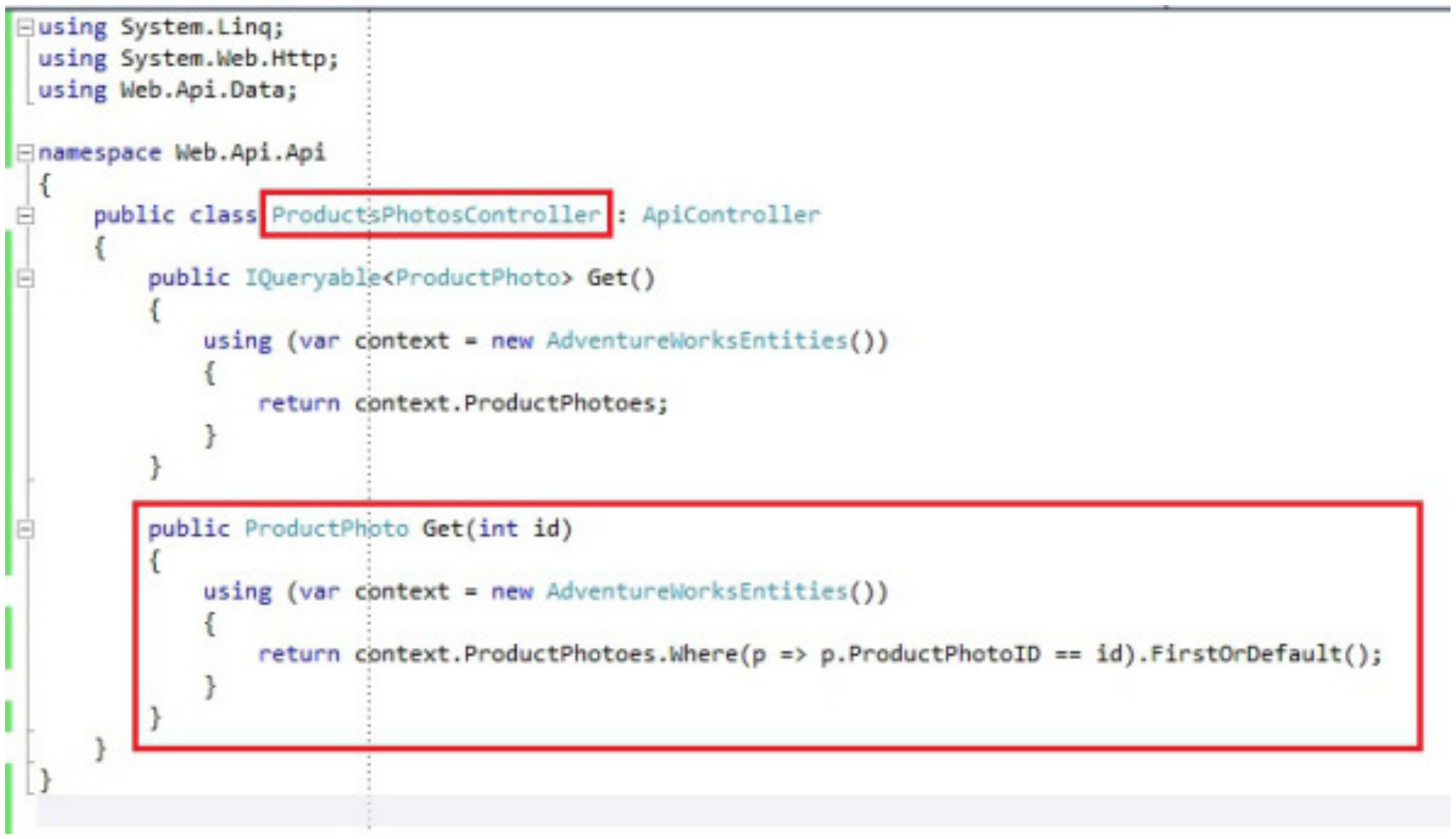
(http://robbincremers.files.wordpress.com/2012/02/28.jpg)

Notice we have a property called LargePhoto which contains the image in binary format in the database. We create a new ApiController for exposing the ProductPhotos:



(http://robbincremers.files.wordpress.com/2012/02/29.jpg)

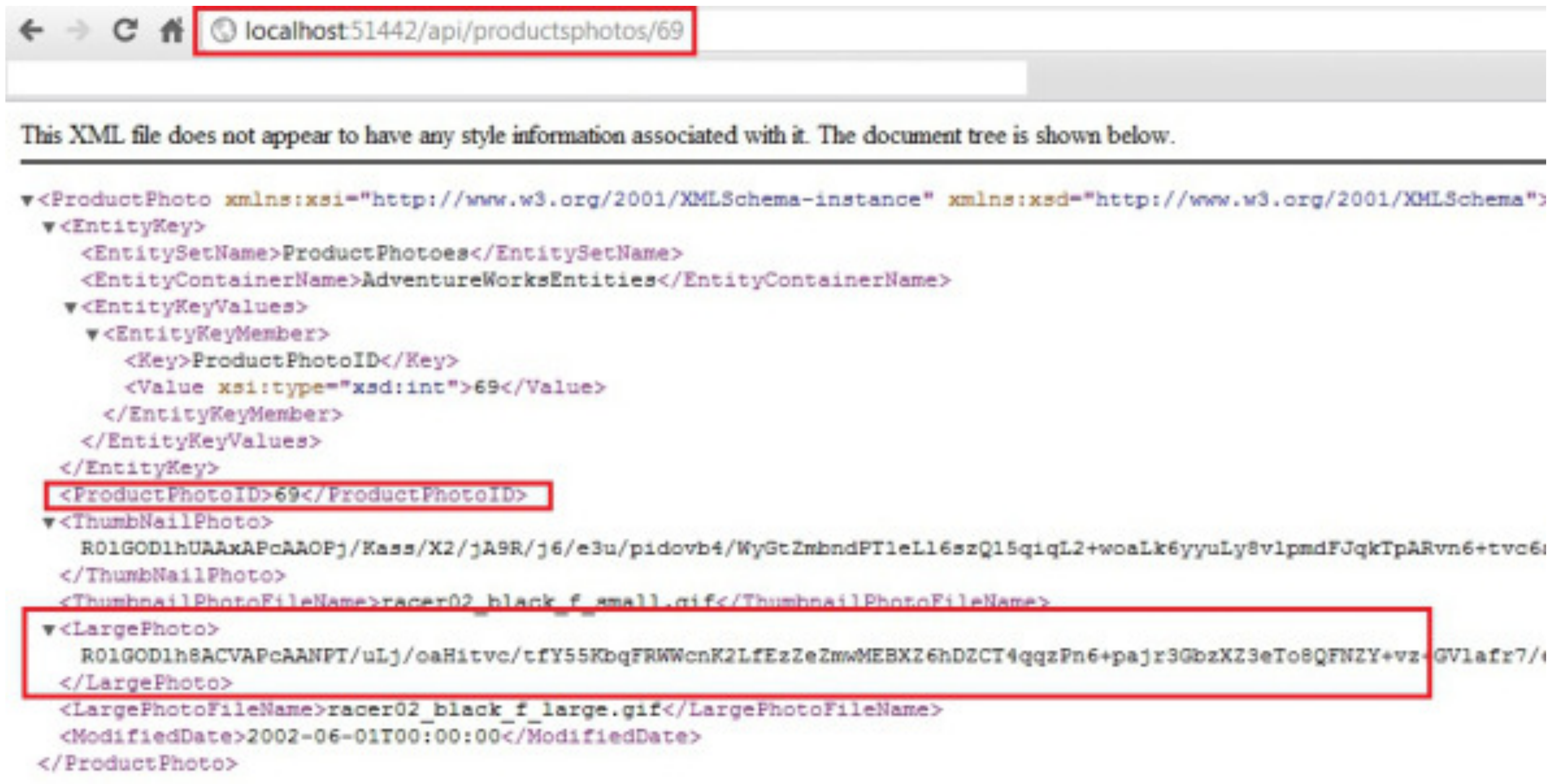
We add some basic code to our new ApiController:



```
using System.Linq;
using System.Web.Http;
using Web.Api.Data;

namespace Web.Api.Api
{
    public class ProductsPhotosController : ApiController
    {
        public IQueryable<ProductPhoto> Get()
        {
            using (var context = new AdventureWorksEntities())
            {
                return context.ProductPhotoes;
            }
        }

        public ProductPhoto Get(int id)
        {
            using (var context = new AdventureWorksEntities())
            {
                return context.ProductPhotoes.Where(p => p.ProductPhotoID == id).FirstOrDefault();
            }
        }
    }
}
```

(http://robbincremers.files.wordpress.com/2012/02/27.jpg)

If you now get the ProductPhoto information for the ProductPhoto with id 69, we will get the information provided in XML:



localhost:51442/api/productsphotos/69

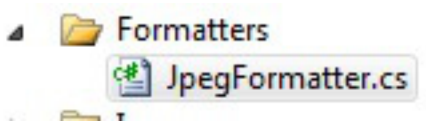
This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<ProductPhoto xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  ▼<EntityKey>
     <EntitySetName>ProductPhotoes</EntitySetName>
     <EntityContainerName>AdventureWorksEntities</EntityContainerName>
   ▼<EntityKeyValues>
     ▼<EntityKeyMember>
        <Key>ProductPhotoID</Key>
        <Value xsi:type="xsd:int">69</Value>
      </EntityKeyMember>
    </EntityKeyValues>
  </EntityKey>
  <ProductPhotoID>69</ProductPhotoID>
  ▼<ThumbNailPhoto>
     R0lGODlhUAAxAPcAAOPj/Kass/X2/jA9R/j6/e3u/pidovb4/WyGtZmbndPTleLl6szQ15qiqL2+woaLk6yyuLy8vlpmdFJqkTpARvn6+tvc6
  </ThumbNailPhoto>
  <ThumbnailPhotoFileName>racer02_black_f_small.gif</ThumbnailPhotoFileName>
  ▼<LargePhoto>
     R0lGODlh8ACVAPcAANPT/uLj/oaHitvc/tfY55KbqFRWWcnK2LfEzZeZmwMEBXZ6hDZCT4qqzPn6+pajr3GbzXZ3eTo8QFNZY+vz+GVlafr7/
  </LargePhoto>
  <LargePhotoFileName>racer02_black_f_large.gif</LargePhotoFileName>
  <ModifiedDate>2002-06-01T00:00:00</ModifiedDate>
</ProductPhoto>
```

(http://robbincremers.files.wordpress.com/2012/02/30.jpg)

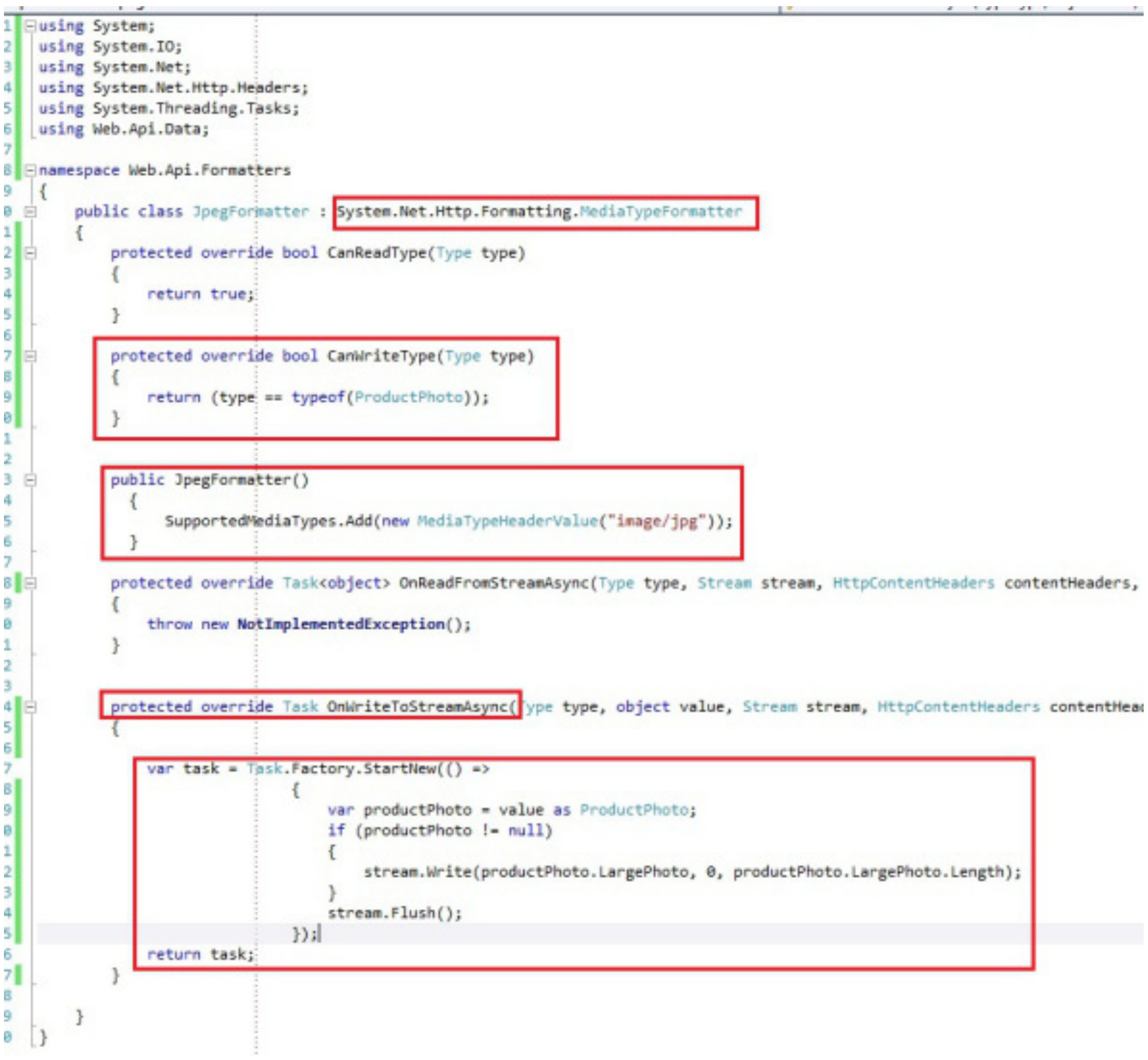
This is nice in case we want to retrieve all the ProductPhoto information. However in some cases we will want to retrieve to image itself and not the binary format of the image that gets written out by default. By default Accept types as "application/xml" or "application/json" are supported. However we want to set the Accept header as "image/jpg" for example and when we use that Accept header, we want the image to be returned.

Doing this can be achieved by creating a class that derives from the **MediaTypeFormatter** class, which belongs to **system.net.http.formatting** namespace.
We create a new MediaTypeFormatter that will write the image of the ProductPhoto to the client if they invoke the **GET operation of the ProductPhoto with an Accept header of "image/jpg"**. Our MediaTypeFormatter will look like this:



(http://robbincremers.files.wordpress.com/2012/02/311.jpg)

The code of our custom MediaTypeFormatter:

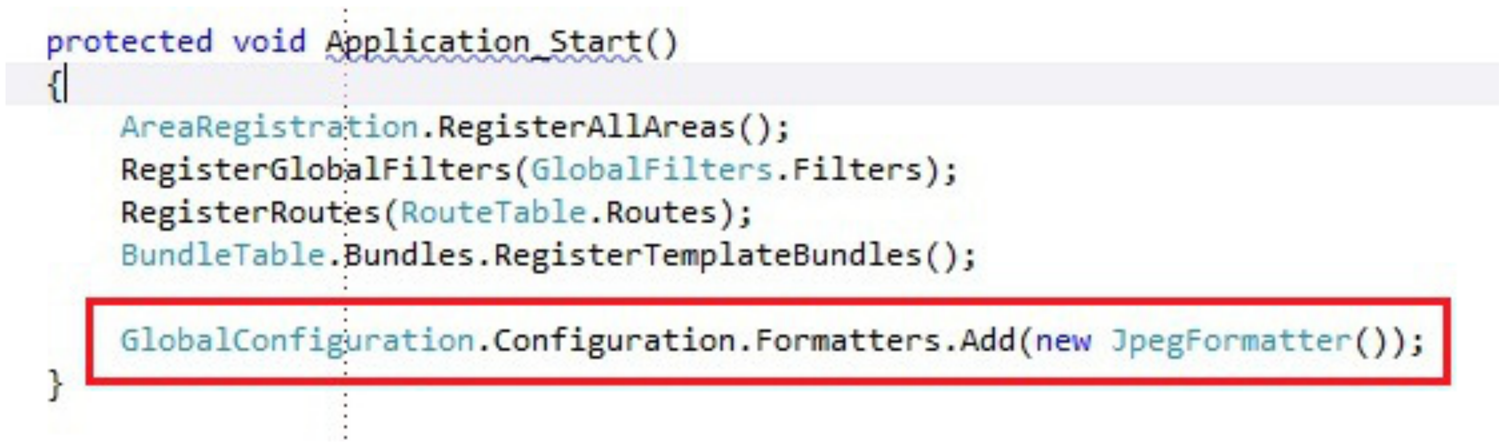


```
using System;
using System.IO;
using System.Net;
using System.Net.Http.Headers;
using System.Threading.Tasks;
using Web.Api.Data;

namespace Web.Api.Formatters
{
    public class JpegFormatter : System.Net.Http.Formatting.MediaTypeFormatter
    {
        protected override bool CanReadType(Type type)
        {
            return true;
        }

        protected override bool CanWriteType(Type type)
        {
            return (type == typeof(ProductPhoto));
        }

        public JpegFormatter()
          {
            SupportedMediaTypes.Add(new MediaTypeHeaderValue("image/jpg"));
          }

        protected override Task<object> OnReadFromStreamAsync(Type type, Stream stream, HttpContentHeaders contentHeaders,
        {
            throw new NotImplementedException();
        }

        protected override Task OnWriteToStreamAsync(Type type, object value, Stream stream, HttpContentHeaders contentHea
        {
            var task = Task.Factory.StartNew(() =>
                       {
                           var productPhoto = value as ProductPhoto;
                           if (productPhoto != null)
                           {
                               stream.Write(productPhoto.LargePhoto, 0, productPhoto.LargePhoto.Length);
                           }
                           stream.Flush();
                       });
            return task;
        }

    }
}
```

(http://robbincremers.files.wordpress.com/2012/02/32.jpg)

In the constructor of our custom MediaTypeFormatter we add a **MediaTypeHeaderValue** of "image/jpeg" to the **SupportedMediaTypes**. We also define that this custom media type can only be written if the object type is ProductPhoto. Finally we create a task in the **OnWriteToStreamAsync** method that will write the binary content of the ProductPhoto.LargePhoto to the stream that is being passed along.

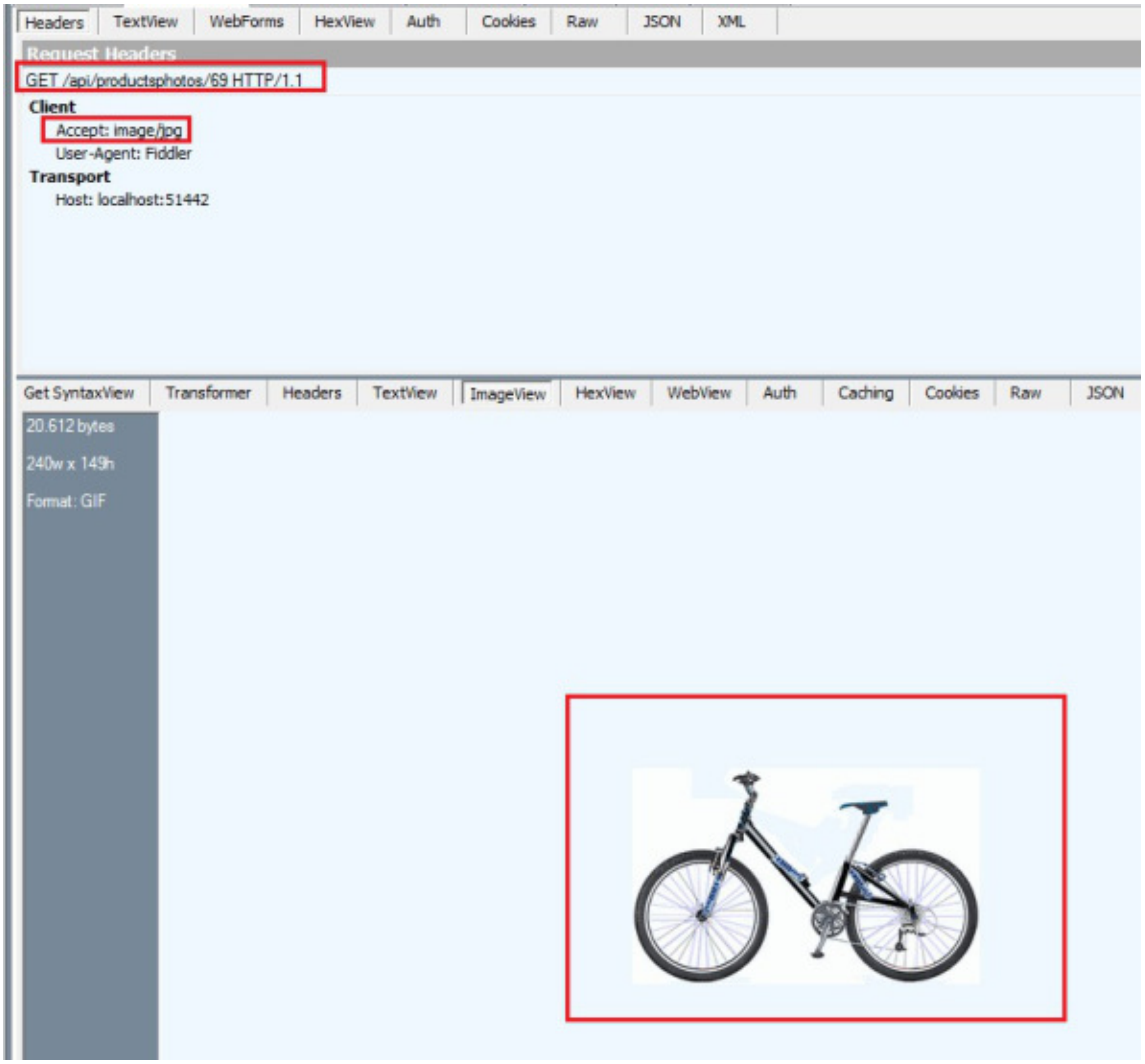
To have this custom MediaTypeFormatter being added in our request/response pipeline, we have to register it. We do this in our global.asax by the **GlobalConfiguration.Configuration.Formatters** collection:



```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();
    RegisterGlobalFilters(GlobalFilters.Filters);
    RegisterRoutes(RouteTable.Routes);
    BundleTable.Bundles.RegisterTemplateBundles();

    GlobalConfiguration.Configuration.Formatters.Add(new JpegFormatter());
}
```

(http://robbincremers.files.wordpress.com/2012/02/34.jpg)

If we with Fiddler invoke a GET request on the /api/productphotos/69/ we will get the following result:



```
GET /api/productsphotos/69 HTTP/1.1
Client
    Accept: image/jpg
    User-Agent: Fiddler
Transport
    Host: localhost:51442
```

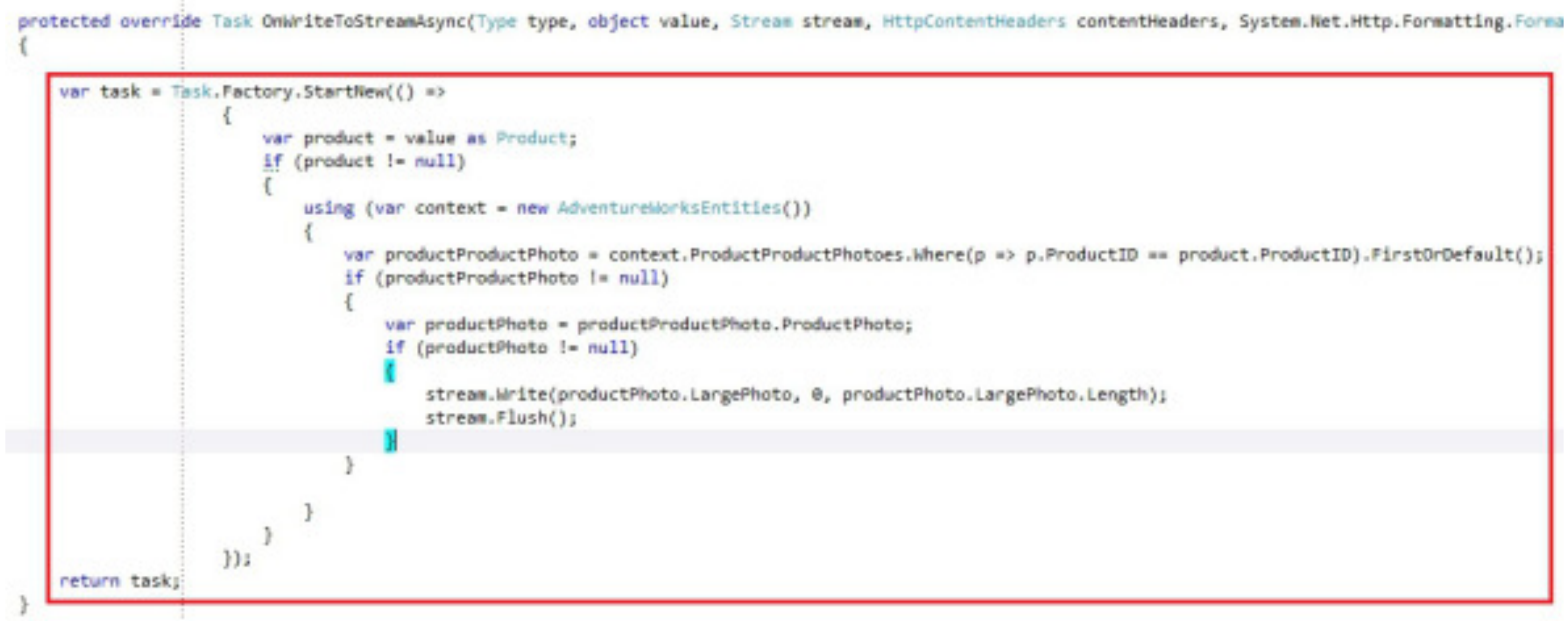
```
20.612 bytes
240w x 149h
Format: GIF
```

(http://robbincremers.files.wordpress.com/2012/02/33.jpg)

So depending on our **Accept header**, we can get the ProductPhoto information back in XML or JSON, but we can also get the ProductPhoto image returned if we add a custom header through the MediaTypeFormatter.

Now if for example you don't want to expose the ProductPhoto information and you only want to expose the image directly on the Product if they ask for a Product with an Accept header of "image/jpeg". In that case you would write your MediaTypeFormatter for the "image/jpeg" Accept header to take a Product object. On the writing of the stream you will do a database query to get the related ProductPhoto and write the binary content to the stream. That way you would not have to expose another ApiController for the only reason to expose an image for a Product.

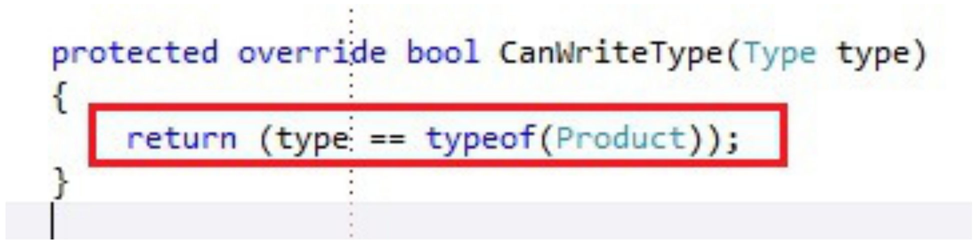
Do get the ProductPhoto directly from the Product GET operation instead of using a ProductsPhotosController, you can rewrite the MediaTypeFormatter like this:



```
protected override Task OnWriteToStreamAsync(Type type, object value, Stream stream, HttpContentHeaders contentHeaders, System.Net.Http.Formatting.Forma
{
    var task = Task.Factory.StartNew(() =>
                {
                    var product = value as Product;
                    if (product != null)
                    {
                        using (var context = new AdventureWorksEntities())
                        {
                            var productProductPhoto = context.ProductProductPhotoes.Where(p => p.ProductID == product.ProductID).FirstOrDefault();
                            if (productProductPhoto != null)
                            {
                                var productPhoto = productProductPhoto.ProductPhoto;
                                if (productPhoto != null)
                                {
                                    stream.Write(productPhoto.LargePhoto, 0, productPhoto.LargePhoto.Length);
                                    stream.Flush();
                                }
                            }

                        }
                    }
                });
    return task;
}
```

(http://robbincremers.files.wordpress.com/2012/02/35.jpg)

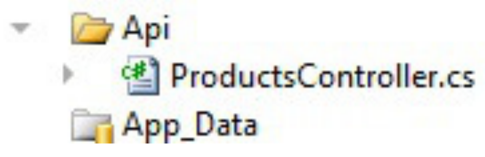
We change the **CanWriteType** operation to only return true when the type is of Product. That means if you request information that is not of type Product, the “image/jpg” formatter will not work and the default format will be returned, which is JSON or XML, depending on the browser you use.



```
protected override bool CanWriteType(Type type)
{
    return (type == typeof(Product));
}
```

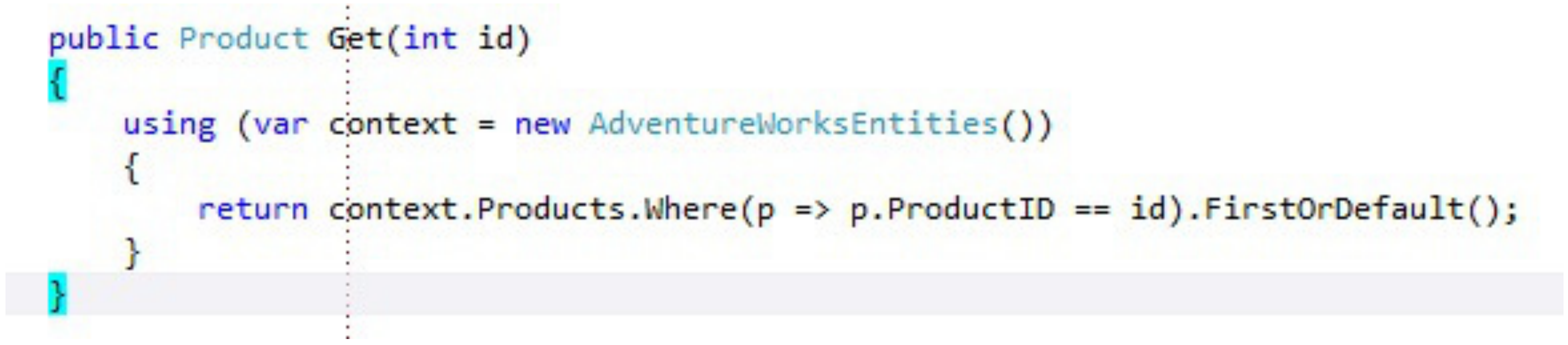
(http://robbincremers.files.wordpress.com/2012/02/37.jpg)

We remove the ProductsPhotosController, since we don’t need it anymore. We are not interested in the ProductPhoto information, we only want to return an image that is related to a product, so we can do that directly on the Product GET Operation by an image/jpg Accept header.



(http://robbincremers.files.wordpress.com/2012/02/38.jpg)

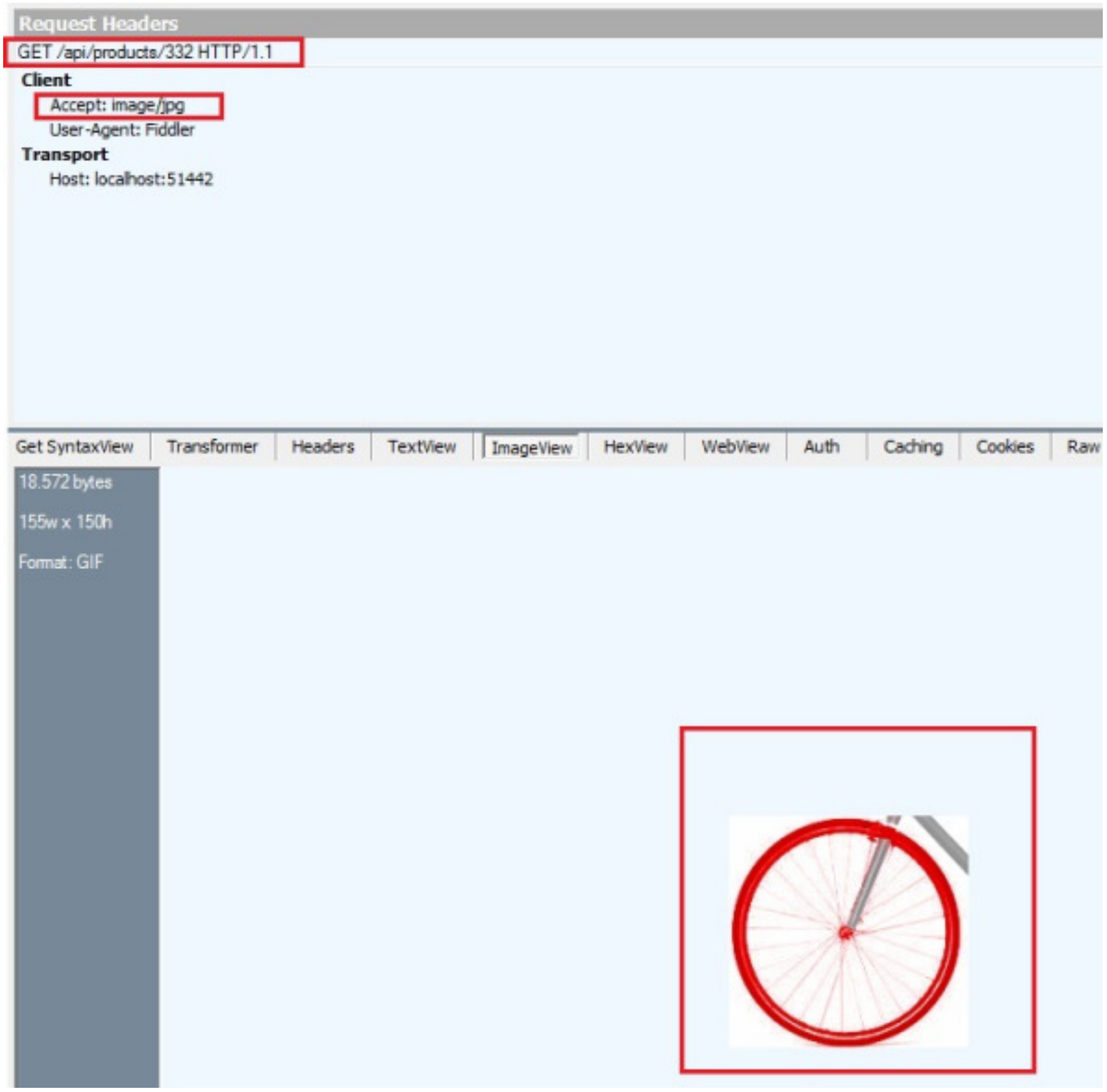
Our GET operation for a specific product still contains the same code:



```
public Product Get(int id)
{
    using (var context = new AdventureWorksEntities())
    {
        return context.Products.Where(p => p.ProductID == id).FirstOrDefault();
    }
}
```

(http://robbincremers.files.wordpress.com/2012/02/40.jpg)

If we request for the Product with ID 332 with an Accept header of image/jpg we get an image back:



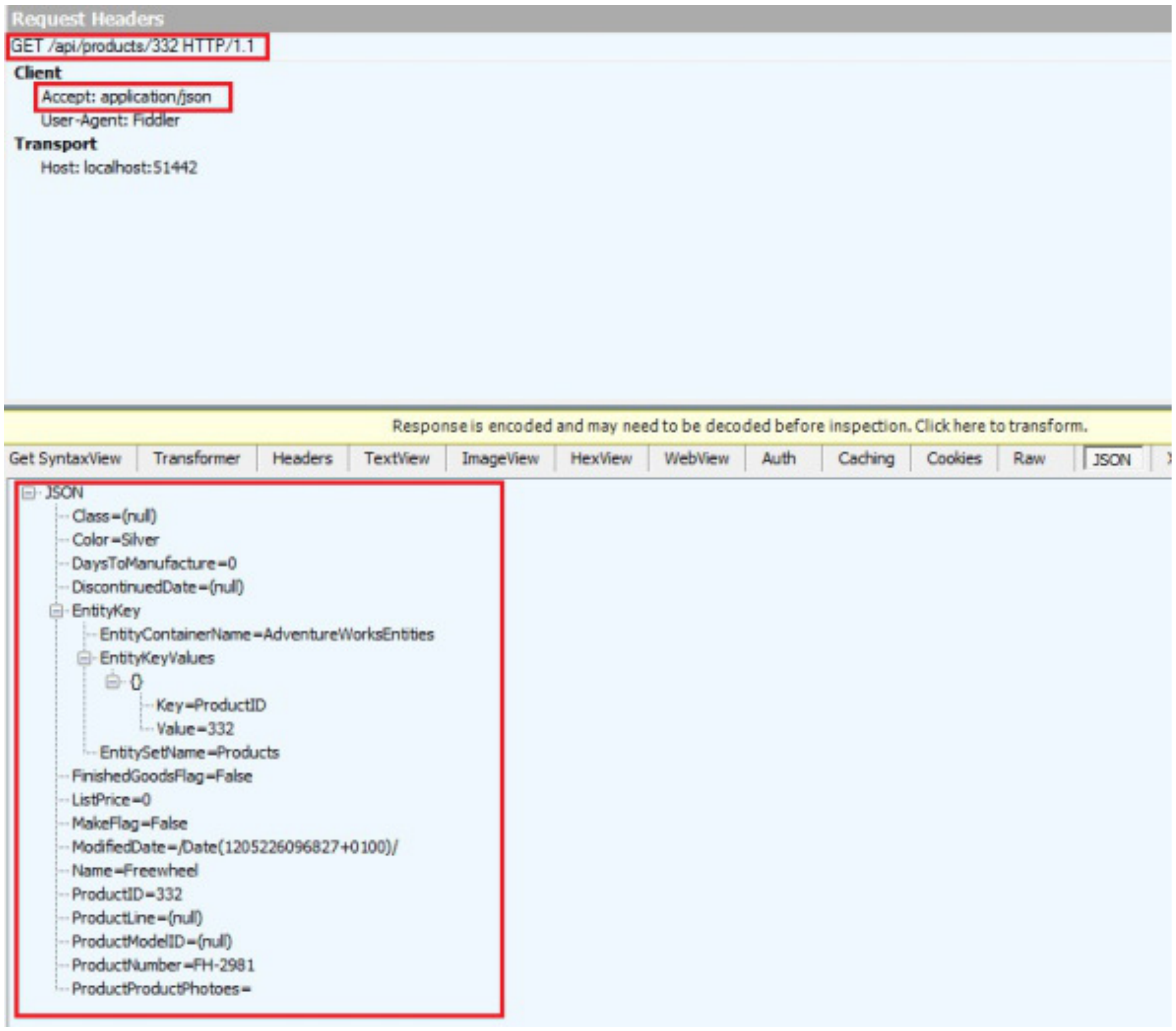
Request Headers

```
GET /api/products/332 HTTP/1.1
Client
    Accept: image/jpg
    User-Agent: Fiddler
Transport
    Host: localhost:51442
```

```
18.572 bytes
155w x 150h
Format: GIF
```

(http://robbincremers.files.wordpress.com/2012/02/36.jpg)

If we now invoke the same uri with an JSON accept header:


Request Headers

GET /api/products/332 HTTP/1.1

Client

Accept: application/json

User-Agent: Fiddler

Transport

Host: localhost:51442

Response is encoded and may need to be decoded before inspection. Click here to transform.

```
JSON
  Class=(null)
  Color=Silver
  DaysToManufacture=0
  DiscontinuedDate=(null)
  EntityKey
    EntityContainerName=AdventureWorksEntities
    EntityKeyValues
      {}
        Key=ProductID
        Value=332
    EntitySetName=Products
  FinishedGoodsFlag=False
  ListPrice=0
  MakeFlag=False
  ModifiedDate=/Date(1205226096827+0100)/
  Name=Freewheel
  ProductID=332
  ProductLine=(null)
  ProductModelID=(null)
  ProductNumber=FH-2981
  ProductProductPhotoes=
```

(http://robbincremers.files.wordpress.com/2012/02/39.jpg)

With content negotation and custom MediaTypeFormatters we can write support for quite some content types, with one and the same operation.

3. Using the JSON.NET serializer instead of the default DataContractJsonSerializer

As someone commented, if you use Entity Framework 4.0 your entity classes will be generated with the [DataContract(IsReference=true)], which is being used for circular references. This is because the EntityObject, which our entity classes derive from, have this IsReference=true attribute set.

However the default DataContractJsonSerializer does not support references, so it is not able to serialize your entity class to a JSON result. You might get an error like this:

The type 'type' cannot be serialized to JSON because its IsReference setting is 'True'. The JSON format does not support references because there is no standardized format for representing references. To enable serialization, disable the IsReference setting on the type or an appropriate parent class of the type.

To solve this issue, you can use the Json.NET serializer instead of the default DataContractJsonSerializer. You can find the third party Json.NET serializer here: http://json.codeplex.com (http://json.codeplex.com)
If you look at the features of Json.NET, you'll see it supports circular references, whereas the default DataContractJsonSerializer does not.

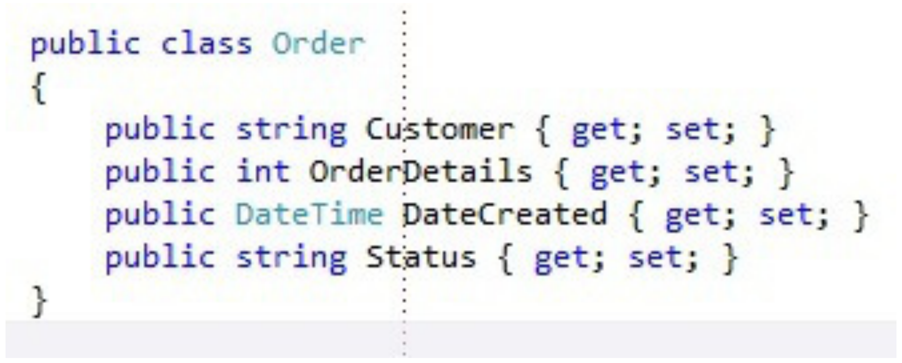
How to implement the Json.NET serializer in ASP.NET Web Api:
http://blogs.msdn.com/b/henrikn/archive/2012/02/18/using-json-net-with-asp-net-web-api.aspx (http://blogs.msdn.com/b/henrikn/archive/2012/02/18/using-json-net-with-asp-net-web-api.aspx)

This issue will be avoided if you use Entity Framework 4.2 Code First with DbContext since you can manage your entity classes yourself.

4. Using the HttpClient to request and serialize data from Web Api

Providing the data is one thing, consuming the data is another. One of the tools you can use to retrieve and post data to your Web Api service is the **HttpClient**, which belongs to the **System.Net.Http** namespace.

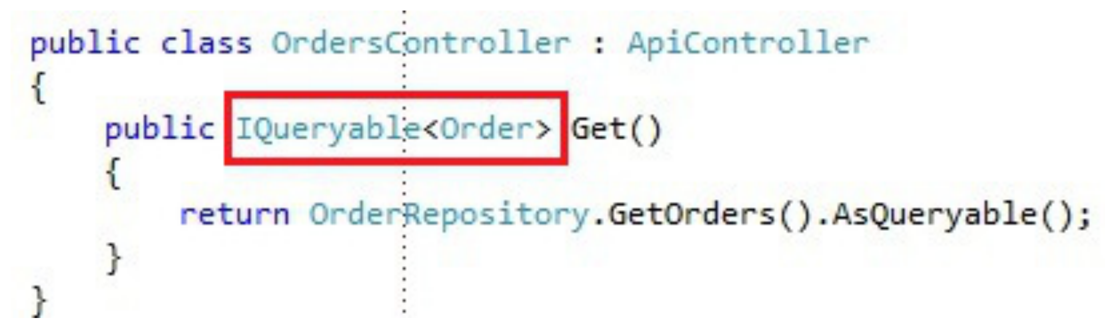
Suppose I have an Order class, which looks like this:



```
public class Order
{
    public string Customer { get; set; }
    public int OrderDetails { get; set; }
    public DateTime DateCreated { get; set; }
    public string Status { get; set; }
}
```

(http://robbincremers.files.wordpress.com/2012/02/425.jpg)

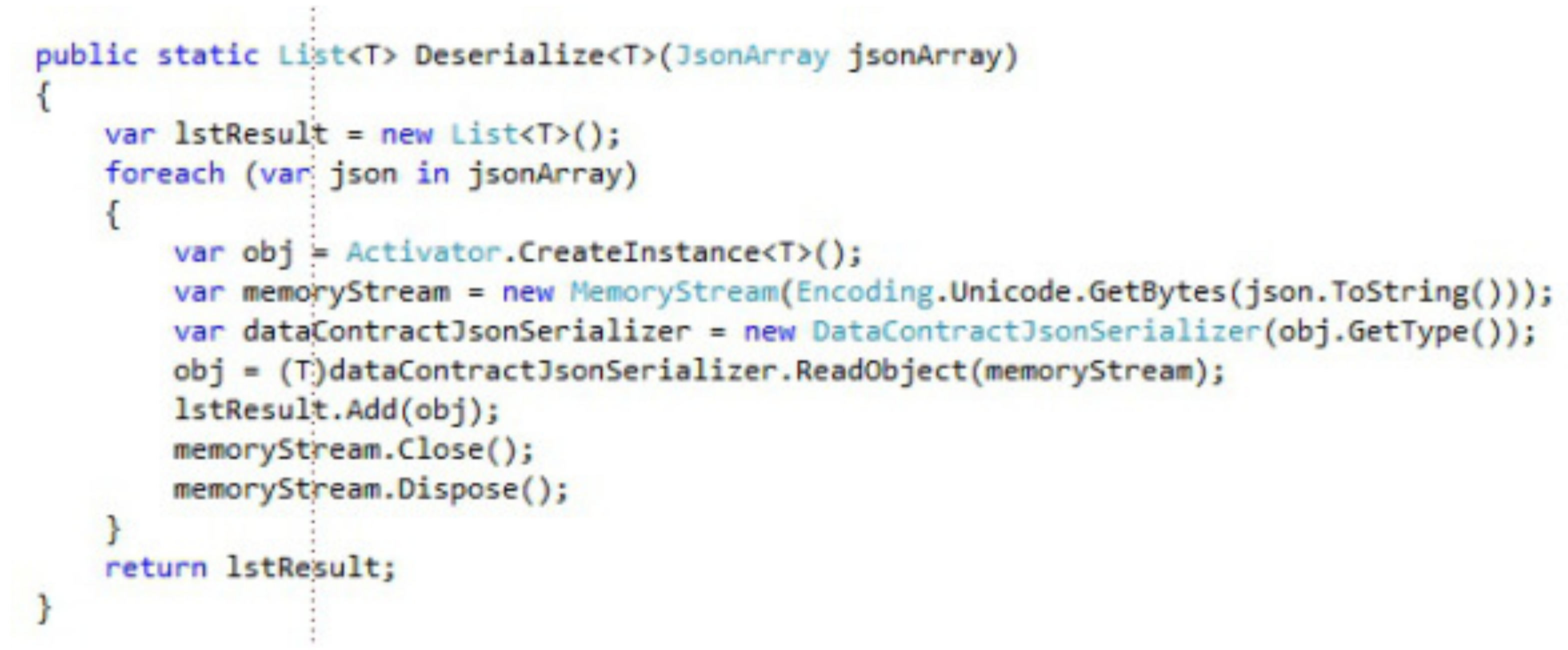
I have an Api OrdersController which exposes a GET operation for an IQuerable<Order>:



```
public class OrdersController : ApiController
{
    public IQueryable<Order> Get()
    {
        return OrderRepository.GetOrders().AsQueryable();
    }
}
```

(http://robbincremers.files.wordpress.com/2012/02/454.jpg)

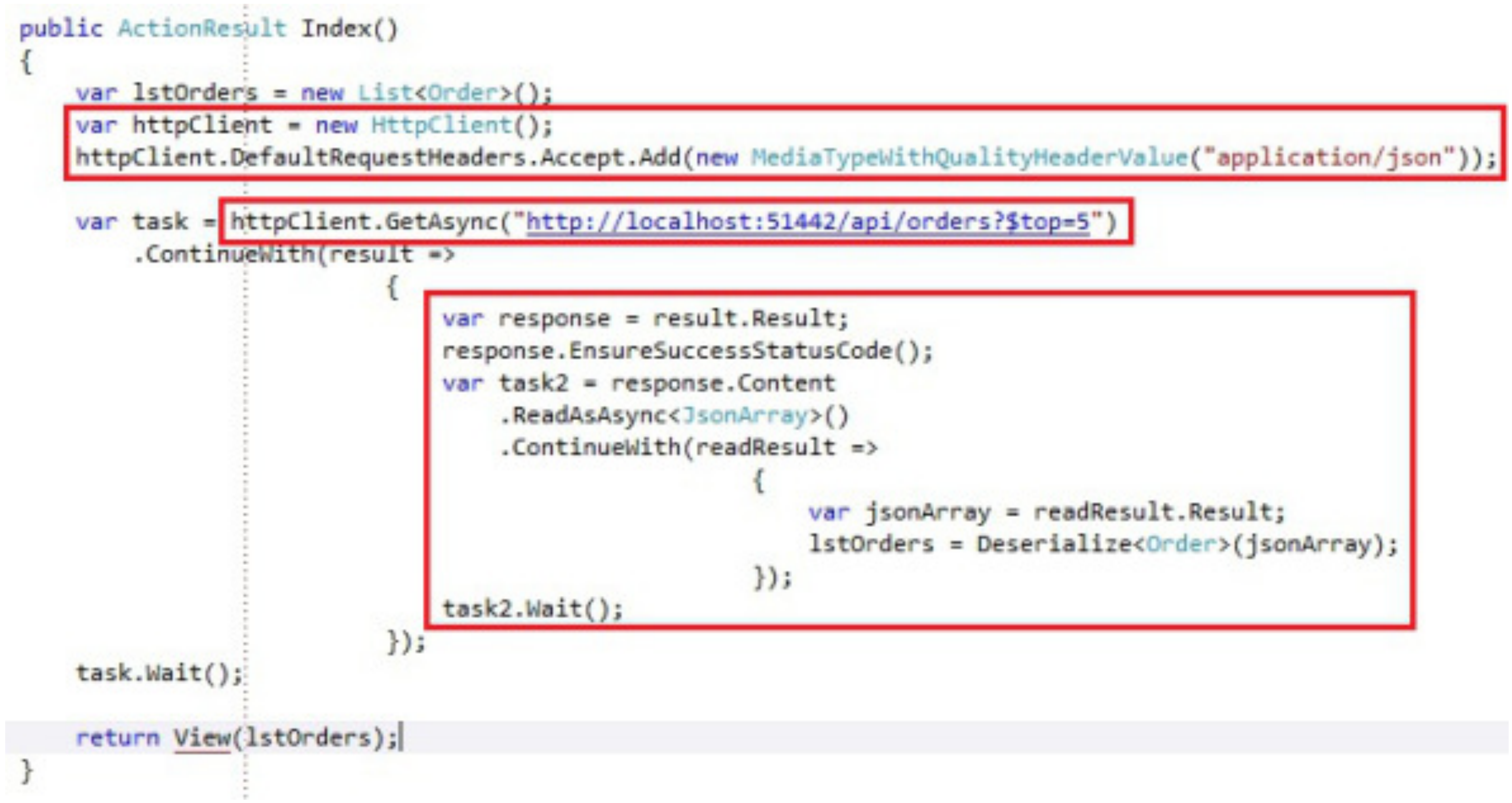
The reason why we expose the data as an IQueryable is so that we can execute OData filters on it. We have an normal OrdersController which will list the orders through a scaffolding list view. We have an operation to Deserialize an JsonArray to a list of object:



```
public static List<T> Deserialize<T>(JsonArray jsonArray)
{
    var lstResult = new List<T>();
    foreach (var json in jsonArray)
    {
        var obj = Activator.CreateInstance<T>();
        var memoryStream = new MemoryStream(Encoding.Unicode.GetBytes(json.ToString()));
        var dataContractJsonSerializer = new DataContractJsonSerializer(obj.GetType());
        obj = (T)dataContractJsonSerializer.ReadObject(memoryStream);
        lstResult.Add(obj);
        memoryStream.Close();
        memoryStream.Dispose();
    }
    return lstResult;
}
```

(http://robbincremers.files.wordpress.com/2012/02/434.jpg)

We create a new HttpClient with an **Accept header of "application/json"**, which means we want to retrieve the orders data in JSON format. You can set the headers on the **DefaultRequestHeaders** property. Finally we have some async task clutter in there to get the result of the response as a JsonArray. Finally we invoke our operation to return any JsonArray to an List<T>, which will return a list of orders. The code I wrote for running the tasks might not be idea as I seriously need to look a bit more into the task threading.

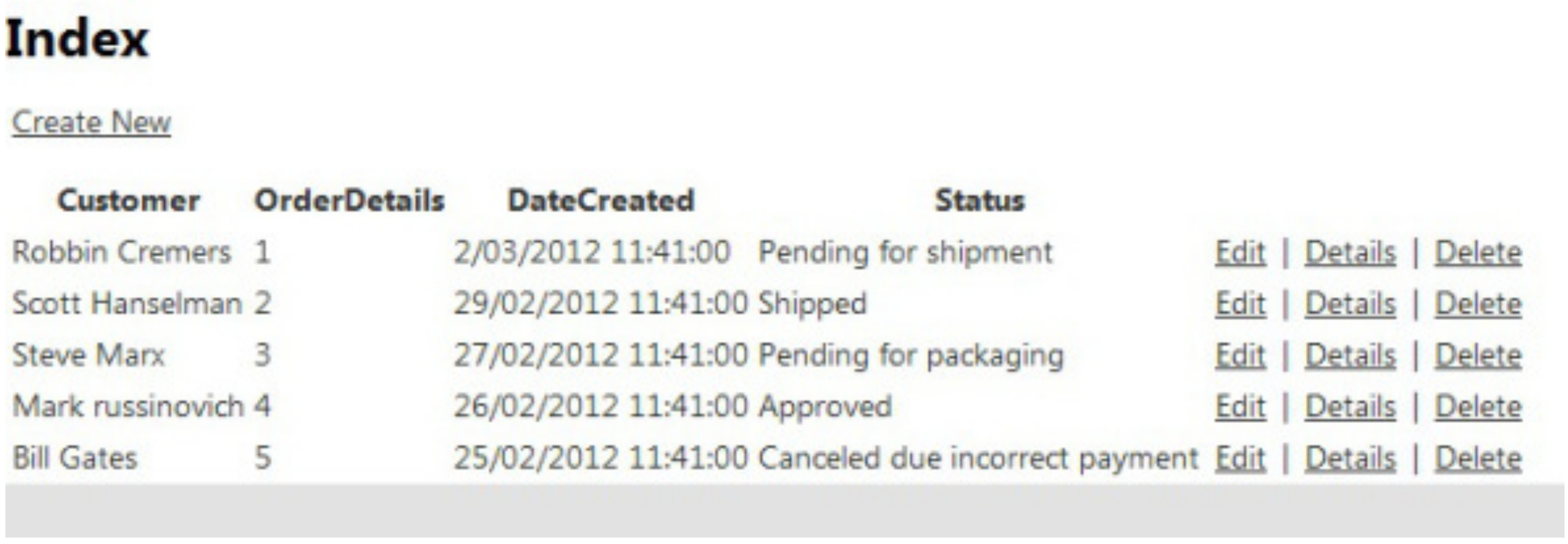


```
public ActionResult Index()
{
    var lstOrders = new List<Order>();
    var httpClient = new HttpClient();
    httpClient.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));

    var task = httpClient.GetAsync("http://localhost:51442/api/orders?$top=5")
        .ContinueWith(result =>
                      {
                          var response = result.Result;
                          response.EnsureSuccessStatusCode();
                          var task2 = response.Content
                              .ReadAsAsync<JsonArray>()
                              .ContinueWith(readResult =>
                                            {
                                                var jsonArray = readResult.Result;
                                                lstOrders = Deserialize<Order>(jsonArray);
                                            });
                          task2.Wait();
                      });
    task.Wait();

    return View(lstOrders);
}
```

(http://robbincremers.files.wordpress.com/2012/02/445.jpg)

We added some code to get the data on the /api/orders?$top=5, to get the top 5 orders from our orders api service. The **$top is an OData operator, which will only work when you exposed the data as an IQueryable**. We get the content from our response and read it as an JsonArray, which we will serialize back to our object.

Visiting the index view for the orders controller, we only get 5 results back:



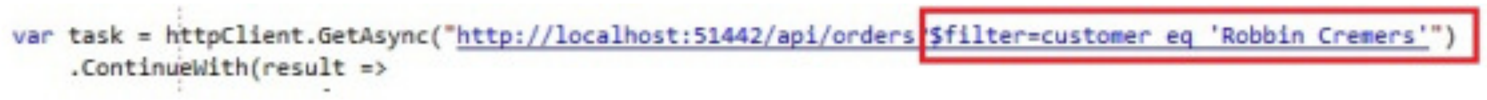
Index

Create New

Customer	OrderDetails	DateCreated	Status	
Robbin Cremers	1	2/03/2012 11:41:00	Pending for shipment	Edit | Details | Delete
Scott Hanselman	2	29/02/2012 11:41:00	Shipped	Edit | Details | Delete
Steve Marx	3	27/02/2012 11:41:00	Pending for packaging	Edit | Details | Delete
Mark russinovich	4	26/02/2012 11:41:00	Approved	Edit | Details | Delete
Bill Gates	5	25/02/2012 11:41:00	Canceled due incorrect payment	Edit | Details | Delete

(http://robbincremers.files.wordpress.com/2012/02/463.jpg)

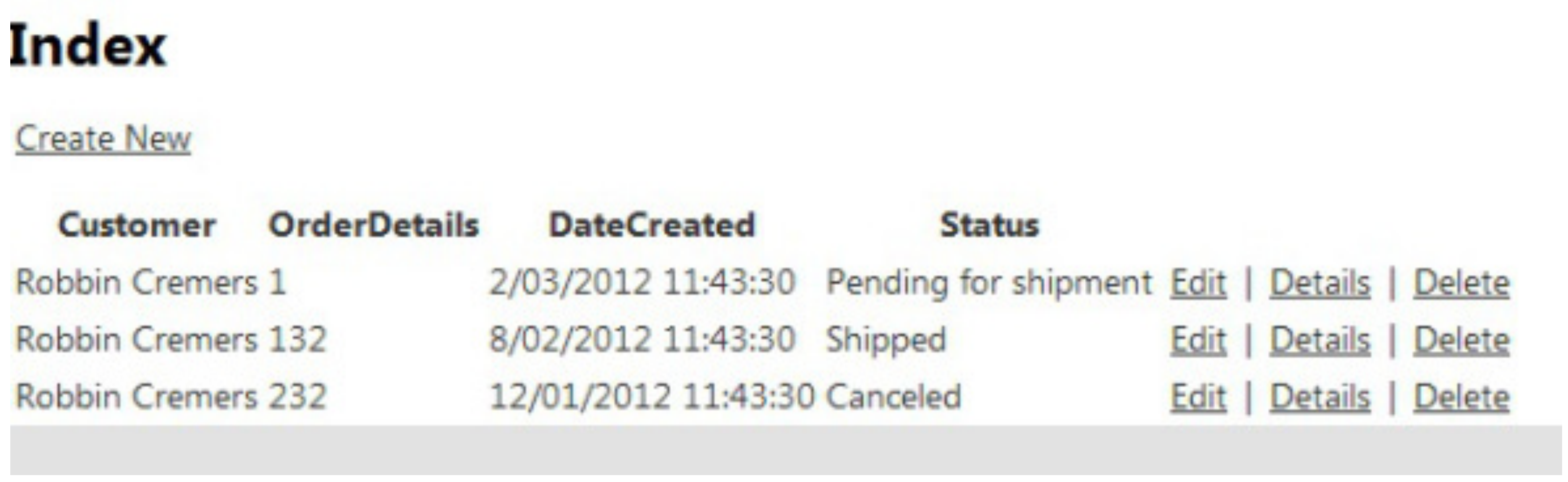
Setting some other OData filter on our request, we will use the **OData $filter operator to filter** all Orders that have a customer with the value of my name



```
var task = httpClient.GetAsync("http://localhost:51442/api/orders$filter=customer eq 'Robbin Cremers'")
    .ContinueWith(result =>
```

(http://robbincremers.files.wordpress.com/2012/02/473.jpg)

And the results:



Index

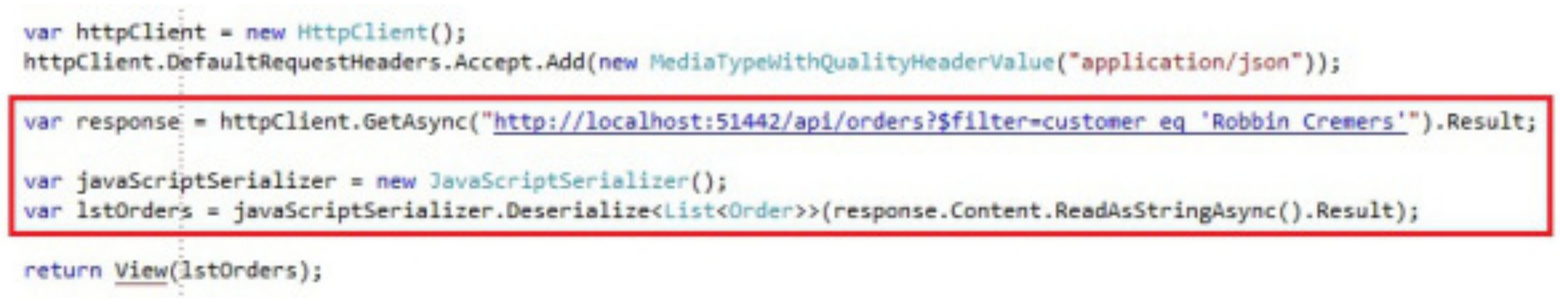
Create New

Customer	OrderDetails	DateCreated	Status	
Robbin Cremers 1		2/03/2012 11:43:30	Pending for shipment	Edit | Details | Delete
Robbin Cremers 132		8/02/2012 11:43:30	Shipped	Edit | Details | Delete
Robbin Cremers 232		12/01/2012 11:43:30	Canceled	Edit | Details | Delete

(http://robbincremers.files.wordpress.com/2012/02/483.jpg)

Through the OData protocol we can apply selecting certain rows with the $skip and $top operations for paging, but we can also do some very strong filtering with the $filter. On our Web Api service only the query is being executed on the database with the filter, so it does not retrieve all information first and then apply the filter.

You could also write the previous code like this:



```
var httpClient = new HttpClient();
httpClient.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));

var response = httpClient.GetAsync("http://localhost:51442/api/orders?$filter=customer eq 'Robbin Cremers'").Result;

var javaScriptSerializer = new JavaScriptSerializer();
var lstOrders = javaScriptSerializer.Deserialize<List<Order>>(response.Content.ReadAsStringAsync().Result);

return View(lstOrders);
```

(http://robbincremers.files.wordpress.com/2012/02/493.jpg)

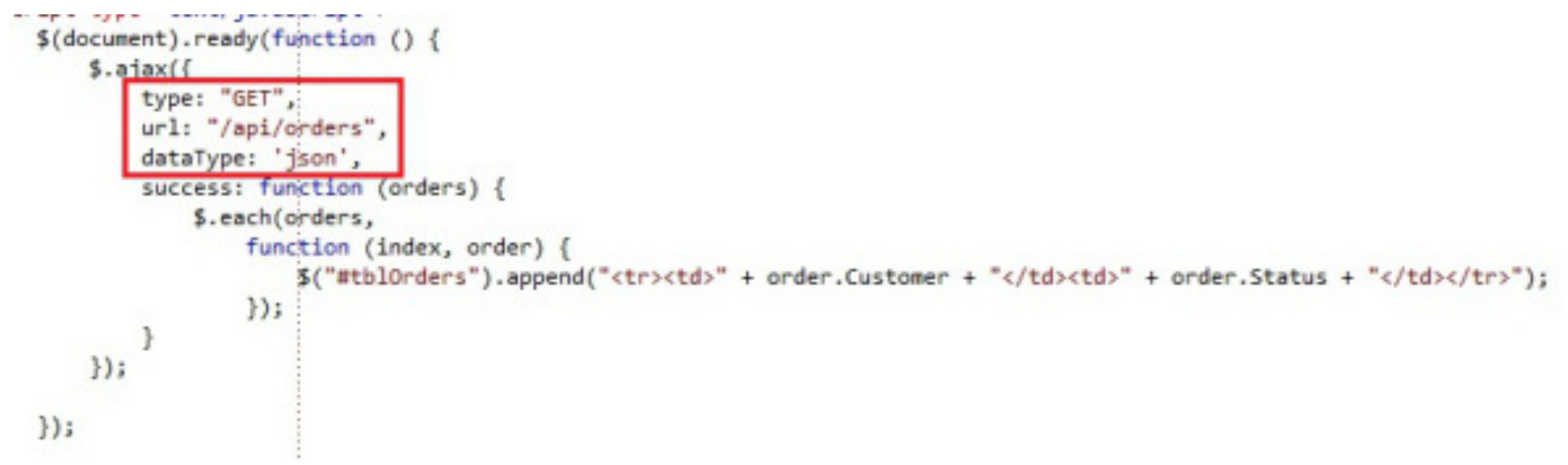
It's more compact code and works in a synchronous manner. We use the **JavaScriptSerializer** in this case, which belongs to the **System.Web.Script.Serialization** namespace. You'll need to import the **System.Web.Extensions.dll** to have access to this namespace. In many cases using the full asynchronous code is better, but at the end of our code, we need to return the list of objects to our view, so it has to be loaded before we pass it along, so you might just as well write it as synchronous code in this case.

The HttpClient also exposes some other functions which are need to manage your requests:

- **GetStreamAsync**: Send a GET request to the specified Uri and return the response body as a stream in an asynchronous operation.
- **GetStringAsync**: Send a GET request to the specified Uri and return the response body as a string in an asynchronous operation.
- **GetByteArrayAsync**: Send a GET request to the specified Uri and return the response body as a byte array in an asynchronous operation.
- **PostAsync**: Send a POST request to the specified Uri as an asynchronous operation.
- **PutAsync**: Send a PUT request to the specified Uri as an asynchronous operation.
- **DeleteAsync**: Send a DELETE request to the specified Uri as an asynchronous operation.

This is still the beta and if I'm correct, the ASP.NET team should have added some features for managing different content types, but guess it'll be waiting till the release. I seriously hope they bring an improved HttpClient that abstracts all the clutter for us and allows use to easily call request and serialize them to objects depending on a content-type of the reponse that comes back.

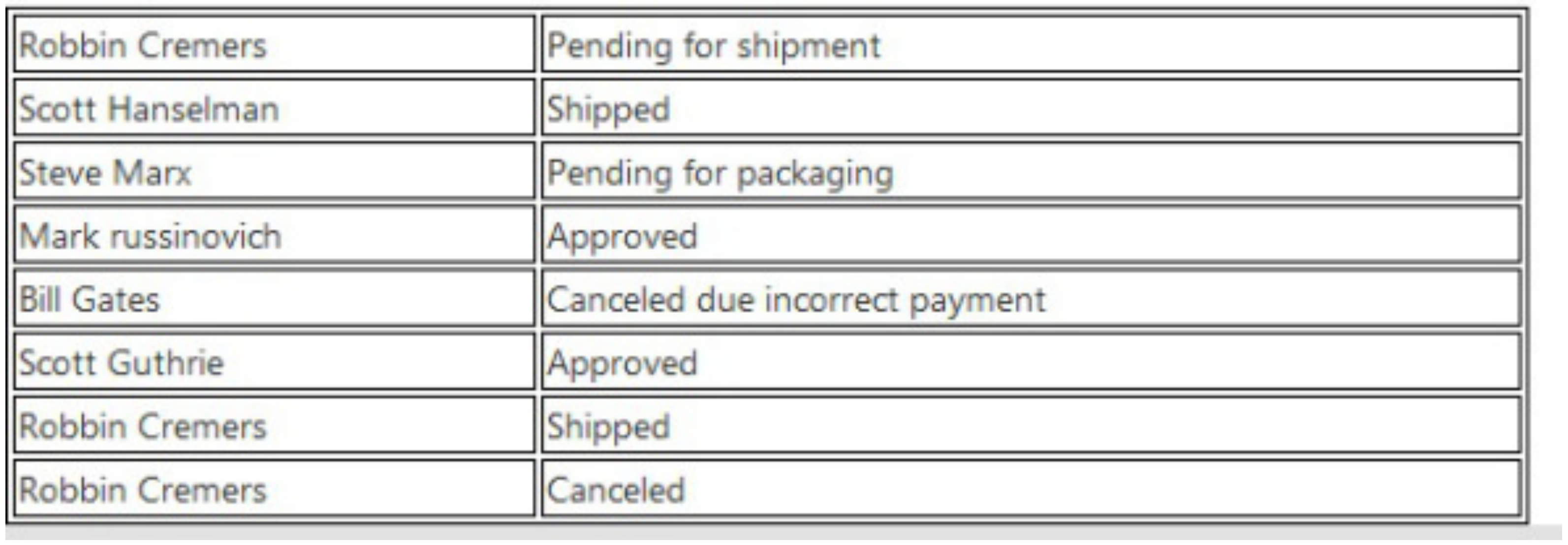
You could also retrieve the **json information from your web api service through a javascript ajax call**, instead of using the HttpClient:



```
$(document).ready(function () {
    $.ajax({
        type: "GET",
        url: "/api/orders",
        dataType: 'json',
        success: function (orders) {
            $.each(orders,
                function (index, order) {
                    $("#tblOrders").append("<tr><td>" + order.Customer + "</td><td>" + order.Status + "</td></tr>");
                });
        }
    });

});
```

(http://robbincremers.files.wordpress.com/2012/02/503.jpg)

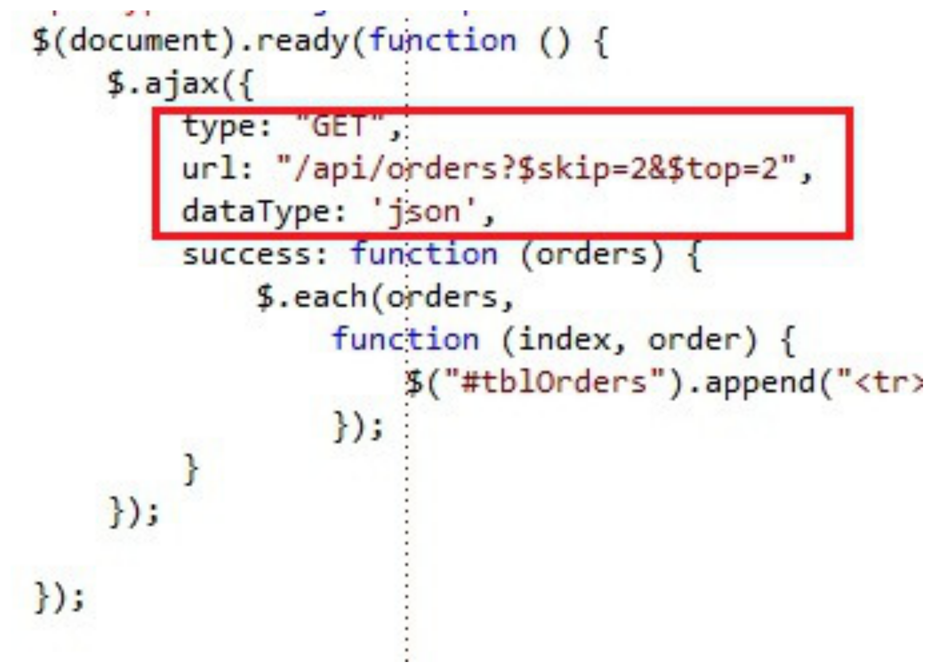
With the result:



Robbin Cremers	Pending for shipment
Scott Hanselman	Shipped
Steve Marx	Pending for packaging
Mark russinovich	Approved
Bill Gates	Canceled due incorrect payment
Scott Guthrie	Approved
Robbin Cremers	Shipped
Robbin Cremers	Canceled

(http://robbincremers.files.wordpress.com/2012/02/5110.jpg)

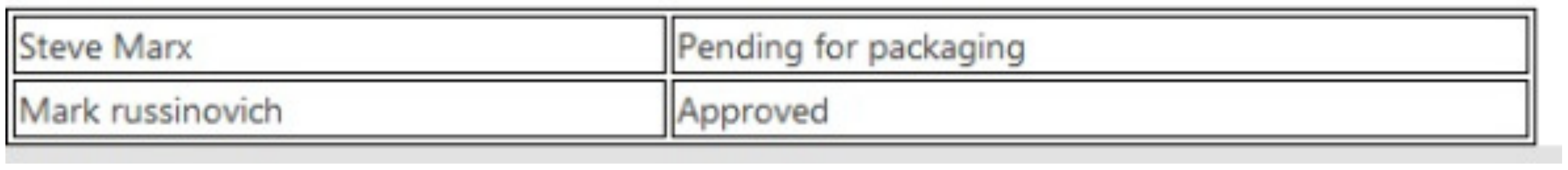
Using **OData** operations is as easy:



```
$(document).ready(function () {
    $.ajax({
        type: "GET",
        url: "/api/orders?$skip=2&$top=2",
        dataType: 'json',
        success: function (orders) {
            $.each(orders,
                function (index, order) {
                    $("#tblOrders").append("<tr>
                });
        }
    });

});
```

(http://robbincremers.files.wordpress.com/2012/02/524.jpg)

The result:



Steve Marx	Pending for packaging
Mark russinovich	Approved

(http://robbincremers.files.wordpress.com/2012/02/534.jpg)

Looking for more information of ASP.NET Web API. You can find an overview of interesting ASP.NET Web API posts here:
http://www.tugberkugurlu.com/archive/getting-started-with-asp-net-web-api-tutorials-videos-samples (http://www.tugberkugurlu.com/archive/getting-started-with-asp-net-web-api-tutorials-videos-samples)

The msdn blog of Henrik F. Nielsen, responsible for Web API, which is covering more advanced topics with ASP.NET Web API:
http://blogs.msdn.com/b/henrikn/ (http://blogs.msdn.com/b/henrikn/)

Any suggestions, remarks or improvements are always welcome.
If you found this information useful, make sure to support me by leaving a comment.

Cheers and have fun,

Robbin

By Robbin Cremers • Posted in ASP.NET • Tagged ASP.NET, OData, REST

20 comments on “Building and consuming REST services with ASP.NET Web API using MediaTypeFormatter and OData support”

Mayrun Digmi
FEBRUARY 19, 2012 @ 12:57 PM
Thanks! Great article, well written and very informative!

0

0

i
Rate This

REPLY
Bonzo
FEBRUARY 28, 2012 @ 9:46 PM
Great post!
I have some problems with getting data in json format however. I’m trying it with my own database and while sending POST request with Accept: Application/json, I get the following error:
“The type user cannot be serialized to JSON because its IsReference setting is ‘True’. The JSON format does not support references because there is no standardized format for representing references. To enable serialization, disable the IsReference setting on the type or an appropriate parent class of the type.
”

Do you know how could I solve it?

0

0

i
Rate This

REPLY
Robbin Cremers
FEBRUARY 29, 2012 @ 7:41 AM

This is because of the [DataContractAttribute(IsReference=true)], which is used for circular references, which is being added to your default generated entity objects with using Entity Framework 4.0. JSON does not support references, so it can not serialize the content to JSON because it does not know how to. The issue is that the EntityObject, which our entity classes derive from, have the IsReference=true set on their DataContractAttribute.

Solution:
Change the JSON serialization from the default DataContractJsonSerializer with the third-party JSON.NET serializer (http://json.codeplex.com/). If you look at the homepage if the JSON.NET serializer, you'll see it supports circular references

How to integrate the JSON.NET serializer in ASP.NET Web Api:
http://blogs.msdn.com/b/henrikn/archive/2012/02/18/using-json-net-with-asp-net-web-api.aspx

PS: When doing a POST request you should use **Content-Type** header to define what the type of the content is that you are submitting. The **accept** header is to define what sorts of content types you can accept, but it does not say what type the content is that you are submitting.

2

0

i
Rate This

REPLY

Bonzo
FEBRUARY 28, 2012 @ 9:51 PM
Moreover, I have also problem with sending POST requests.
I get the following error:
"{"ExceptionType":"System.InvalidOperationException","Message":"No 'MediaTypeFormatter' is available to read an object of type 'User' with the media type "undefined".","StackTrace":" at System.Net.Http.ObjectContent.SelectAndValidateReadFormatter(Boolean acceptNullFormatter)\u000d\u000a at System.Net.Http.ObjectContent.ReadAsyncInternal[T](Boolean allowDefaultIfNoFormatter)\u000d\u000a at System.Web.Http.ModelBinding.RequestContentReader.ReadWholeBodyAsync(HttpActionContex actionContext, Type modelType)\u000d\u000a at System.Web.Http.ModelBinding.RequestContentReader.ReadContentAsync(HttpActionContext actionContext)\u000d\u000a at System.Web.Http.ModelBinding.DefaultActionValueBinder.BindValuesAsync(HttpActionContext actionContext, CancellationToken cancellationToken)\u000d\u000a at System.Web.Http.ApiController.c__DisplayClass3.b__0()\u000d\u000a at System.Web.Http.ApiController.ExecuteAsync(HttpControllerContext controllerContext, CancellationToken cancellationToken)\u000d\u000a at System.Web.Http.Dispatcher.HttpControllerDispatcher.SendAsyncInternal(HttpRequestMessage

request, CancellationToken cancellationToken)\u000d\u000a at System.Web.Http.Dispatcher.HttpControllerDispatcher.SendAsync(HttpRequestMessage request, CancellationToken cancellationToken)"}"

Hope you are able to give me some hints:)
Thanks!

0

1

i
Rate This

REPLY

Robbin Cremers
FEBRUARY 29, 2012 @ 8:10 AM
The same issue as the reply I gave on your previous comment. You need to pass the **Content-Type header as "application/xml" or "application/json" when you are executing a POST request**. If you do not pass the Content-Type HTTP header it does not know what the content type is of the data you submitted, so it takes a content-type of "undefined".

Then it checks if there is a MediaTypeFormatter for a content-type of "undefined", which it does not find. That's the exception you are getting back.

0

1

i
Rate This

REPLY

Bonzo
FEBRUARY 29, 2012 @ 5:12 PM
Great, thanks a lot!:)

0

1

i
Rate This

REPLY

Robbin Cremers

MARCH 1, 2012 @ 7:57 AM

You're welcome. I hope it solves your issues.
I appreciate feedback

0

0

i
Rate This

REPLY

Franklin Ngoun

MARCH 2, 2012 @ 10:15 AM

Utterly pent subject material , regards for information .

0

1

i
Rate This

REPLY

Manuel Ortiz

MARCH 7, 2012 @ 4:51 AM

Hi, I have been looking everywhere but cannot find the answer to this question. I am developing a RESTful API using WebAPI and want to do the following. Suppose you have a list of movies and each movie has a list of actors each with their own properties (eg.: age, sex, birthday, etc.). I would like to present the response of that subset of actors with URI's inside every movie as follows:

The Artist
2011

api/actors/1
api/actors/2

Is there anyway to automatically do the uri thing inside the "actor" tag using WebAPI?

0

1

i
Rate This

REPLY
Manuel Ortiz
MARCH 7, 2012 @ 4:52 AM
Correction: the xml tags were removed from my previous post but the "actor" tag is the one that contains the "api/actors/1″ content. That URI is the one I want to be automatically generated.

0

1

i
Rate This

REPLY
Robbin Cremers
MARCH 7, 2012 @ 7:04 AM
I don't think this currently is supported by ASP.NET Web API. If it would have been supporting OData fully, you could have used the OData $select operator to select a specific property from the returned object data or by using an uri format like you specified. However this currently is not supported.

WCF Data Services support OData operators fully and the feature you want to use:
http://robbincremers.me/2012/01/24/wcf-rest-service-with-odata-and-entity-framework-with-client-context-custom-operations-and-operation-interceptors/
Just before the start of chapter 2, you can see the feature you want to use.

As for ASP.NET Web API, I think you'll have to wait until these features are being added and shipped, which might take a while. For now there is no intent to support the OData $select operator, according to this:
http://forums.asp.net/t/1771116.aspx/1?OData+Support

If you are looking for full OData support, WCF Data Services at the time being is still the thing to go with. I expect the ASP.NET / WCF team to add further OData support to the ASP.NET Web API in the future, but for now you'll have to do without it if you want to use ASP.NET Web API

0

1

i
Rate This

REPLY
Francesco Kisselburg
MARCH 9, 2012 @ 9:56 PM
Wow. Very cool!

0

1

i
Rate This

REPLY

9. **PINGBACK:** What is RESful service and designning RESTful service using ASP.NET | Broken Code

Russell Wyatt
JULY 14, 2012 @ 2:51 PM
Hi Robbin. Don't suppose you have the bit of js / jquery you would use to display the image have you? It's eluding me at the moment. Great article.

0

0

i
Rate This

REPLY
Tim Bez
JULY 16, 2012 @ 11:01 PM
Hi, is there anyway to reduce the amount of content the Json string returns, as Im getting extra entity info in addition to my normal data here's an example of the extra data :
"EntityKey":
{"$id":"4","EntitySetName":"Users","EntityContainerName":"GustareEntities","EntityKeyValues"
[{"Key":"UserID","Type":"System.Int32","Value":"900990"}]}}]
Thanks!

0

0

i
Rate This

REPLY
Brian
JULY 31, 2012 @ 1:03 PM
Hi there,

Great post, have my image serving up nicely in fiddler.
However I now have a silly question, how can i display this image on a webpage in html?

0

0

i
Rate This

REPLY
Brian
JULY 31, 2012 @ 1:31 PM
U know what I figured it out,,

<img src="data:image/jpg;base64,/9j/4AAQSkZJ…….

so looks like in my case i didn't need the content negotiation after all, nice post still the same.

0

0

i
Rate This

REPLY
Lel
AUGUST 23, 2012 @ 1:51 AM
Robbin, great post – I'm unable to find the source code. Can you point me to it.
Thanks!

0

0

i
Rate This

REPLY

14. **PINGBACK:** Interesting post on ASP.NET WebAPI « Synthesis – The combination of ideas into a complex whole

_GET["a"] Array ([0] => sex kamerki (
NOVEMBER 6, 2013 @ 7:28 AM
Hi, I do think this is a great webb site. I stumbledupon it I may revisit yet again since I sabed as a favorite it.Money and freedom is the best way to change,
may you be rich and continue to help other people.

0

0

i
Rate This

REPLY