

Formatters and Content Negotiation in ASP.NET Web API 2

- Akhil Mittal

Contents

Introduction.....	1
Content Negotiation	1
Application Setup.....	2
Formatters in Web API.....	9
Content Negotiation in Web API	14
Accept Headers	14
Content Type	14
Formatters	14
Accept Language Headers	14
Quality Factor	15
Accept Encoding Headers.....	15
Accept Charset Headers	15
Content Negotiation Implementation in Web API	15
Custom Content Negotiation Implementation in Web API.....	18
Conclusion	20
References.....	20

Introduction

As the title suggests, this article will focus on practical aspects of formatters and content negotiation in ASP.NET Web API. This article will explain, what content negotiation is?, why it is necessary ? And how to achieve and get that working in ASP.NET Web API? The article will focus more on implementation part of content negotiation in Web API. The first part of the article will focus on formatters where it describes how to support XML or JSON formats in Web API and how to format the result of the API. We'll take a sample Web API project that caters simple CRUD operations on database using entity framework. We'll not go into details of underlying project architecture and the standard way of architecting the same but will focus on content negotiation part in Web API project. For creating a standard enterprise level application with Web API, you can follow [this](#) series.

Content Negotiation

REST stands for Representational State Transfer, and one of the important aspects of representation is its form. The form of the representation means that when the REST service is invoked then what should be there in the response and in what form the response should be. The best part of REST is that it could be used for multiple platforms , may it be .net client, and HTML client playing on JSON objects or any mobile device

platform .The REST service should be developed in such a way so that it suffice the need of client in terms of what format it is requesting, the format of the response may still include the business entities and informative data intact but may vary in context of returned media type form like XML or JSON or any custom defined media type, the character set, encoding that the client is using to render the data and the language as well. For example, suppose a service is intended to return a list of any entity, then the object needs to be serialized in the form on which the client is expecting it. If the REST service only returns XML, then the client which has bound its controls with JSON object will not be able to make use of the returned data or may have to write some other tricky implementation to first convert the data into desired JSON format then bind the controls which results in an extra overhead or may eventually turn out to be a bottleneck. The server should be able to send the best possible representation available with it as per the client request. ASP.NET Web API provides that capability of making a robust REST service that handles the client's request, understand it and serve the data accordingly. Web API introduces a layer called content negotiation in its underlying architecture having standard HTTP rules to request a data in a desired format.

The following statements from [this link](#) describes the to point explanation of Content Negotiation and its primary mechanism.

" The HTTP specification (RFC 2616) defines content negotiation as "the process of selecting the best representation for a given response when there are multiple representations available." The primary mechanism for content negotiation in HTTP are these request headers:

- **Accept:** Which media types are acceptable for the response, such as "application/json," "application/xml," or a custom media type such as "application/vnd.example+xml"
- **Accept-Charset:** Which character sets are acceptable, such as UTF-8 or ISO 8859-1.
- **Accept-Encoding:** Which content encodings are acceptable, such as gzip.
- **Accept-Language:** The preferred natural language, such as "en-us".

The server can also look at other portions of the HTTP request. For example, if the request contains an X-Requested-With header, indicating an AJAX request, the server might default to JSON if there is no Accept header."



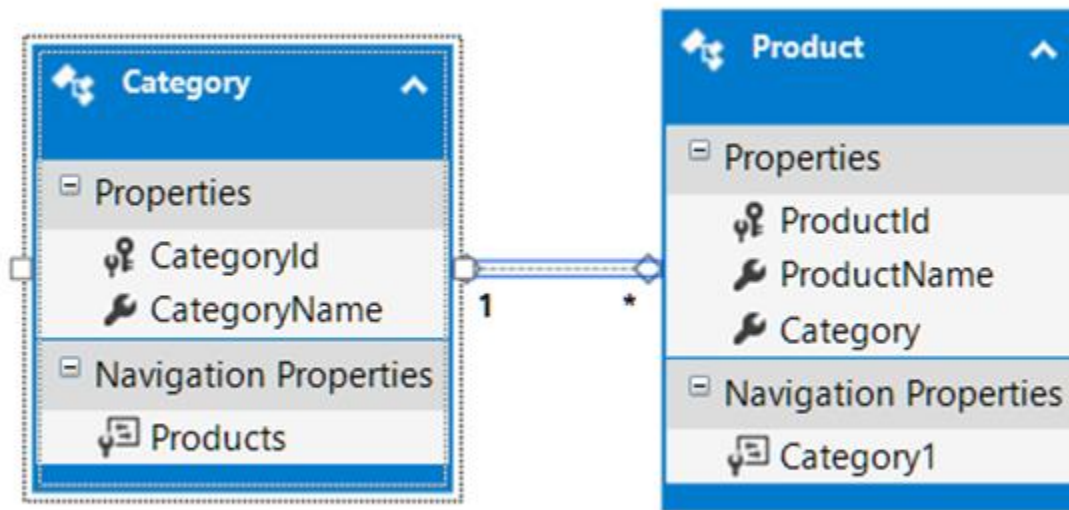
We'll deep dive into content negotiation, and how does it works with Web API. First of all we'll set up a small solution having CRUD operations on database and services exposed as ASP.NET Web API REST services. We'll do a basic implementation of exposing REST services and then on content negotiation part and will not focus much on the architecture of the application.

Application Setup

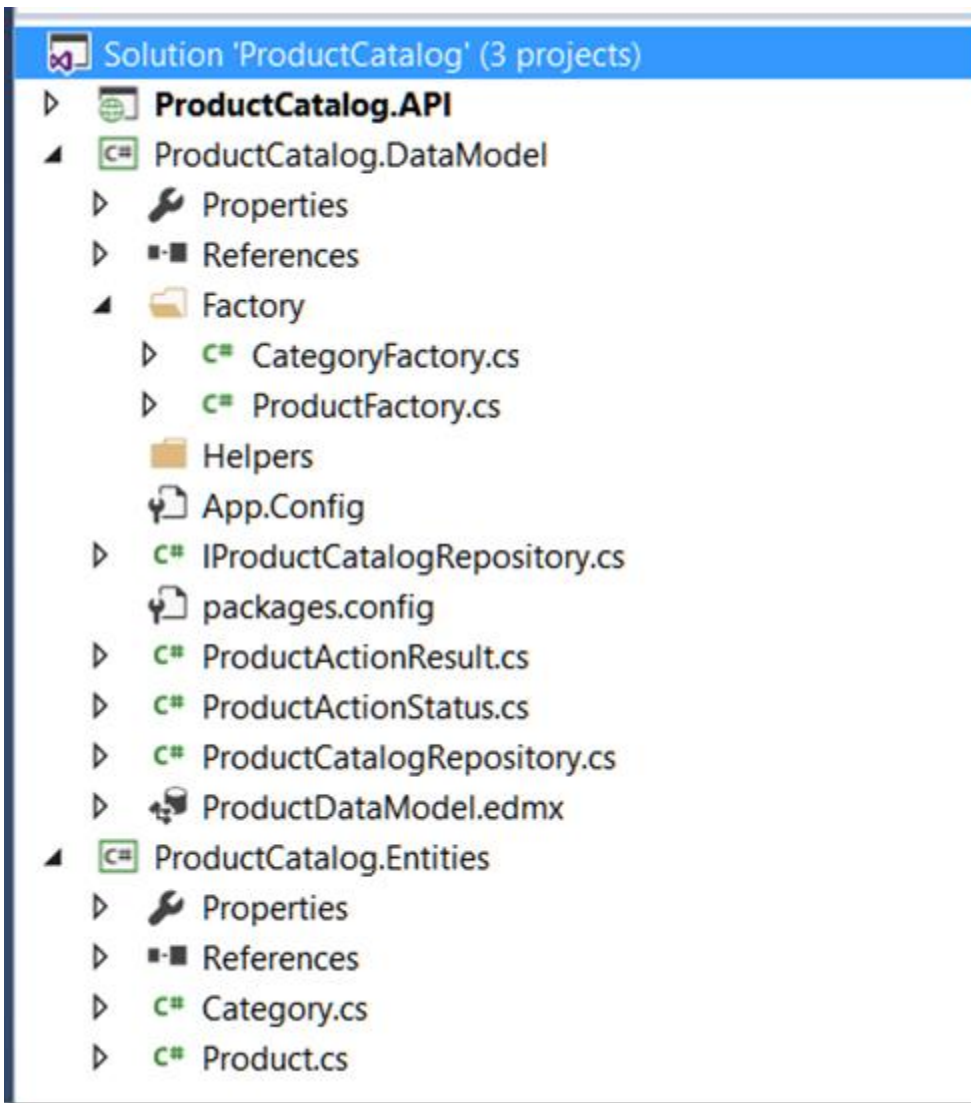
We'll not go into very much detail in setting up the solution step by step, but have an existing solution that I'll explain before we proceed further. I have tried to create a sample application serving product catalog. The complete source code (basic setup + end solution) and database scripts are available for download with this article. Following is the technology stack used in this solution,

- Database : SQL Server Local DB (any database could be used)
- IDE : Visual Studio 2015 Enterprise (any version of Visual Studio can be used that support .Net framework 4.5 or above)
- ORM : Entity Framework 6
- Web API : Web API 2
- .Net Framework : 6.0 (4.5 or above can be used)

The database used is very simple that contains only two tables named Product and Category to which a product belongs. Following is the entity relationship diagram of the database. The application Local DB for database. I purposely used it for this tutorial as it is a small application. One can use any database as per need or requirement in real time scenario.



The solution is divided into three layers as shown below.



ProductCatalog.DataModel project takes the responsibility of database interactions. This is a simple C# class library acting as a data model of the application and directly communicates with database. It contains the Entity data model generated with the help of Entity Framework. Apart from that for transactions the project defines a repository contract named `IProductCatalogRepository` implemented by a concrete class named `ProductCatalogRepository`. This repository contains all the CRUD operations required for DB interactions for Product and Category. `ProductActionResult` class is responsible for returning the result of transaction be it a status or an exception in case of error. `ProductActionStatus` class defines the list of enums showing the status of the response to be returned. The Factory folder contains two factories named `CategoryFactory` and `ProductFactory` responsible for transforming database entities to custom entities to be used for transferring the object and vice-versa.

`IProductCatalogRepository`

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ProductCatalog.DataModel
{
```

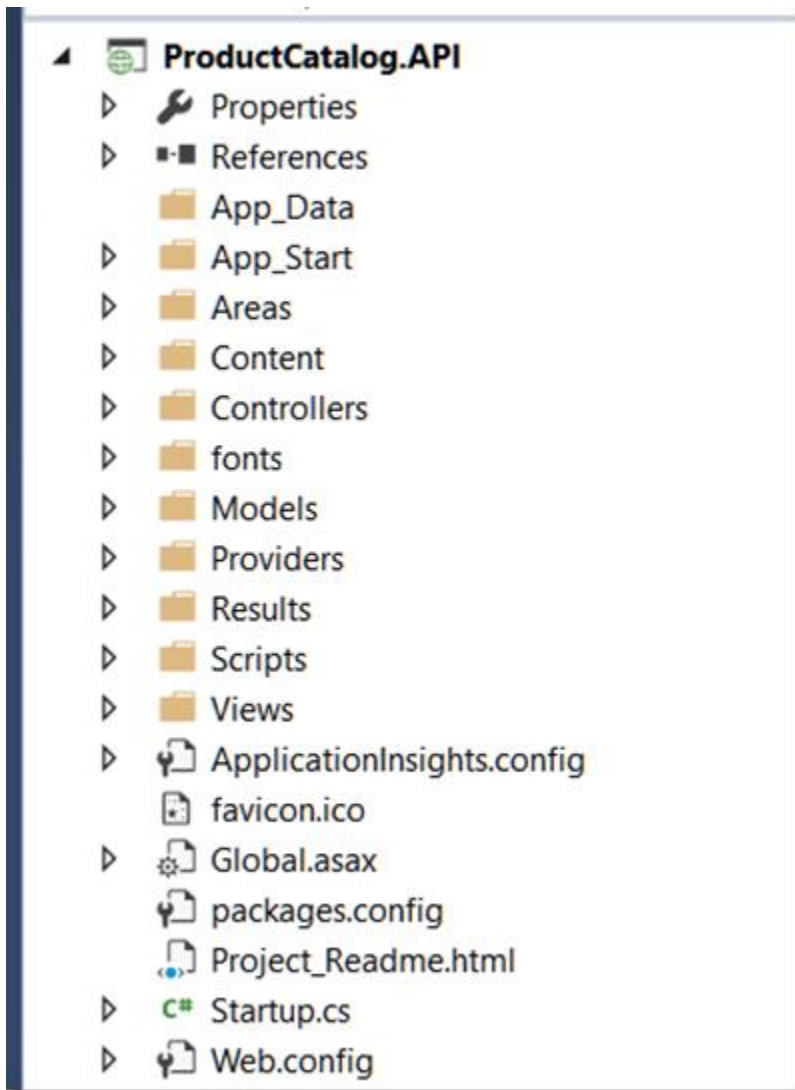
```

public interface IProductCatalogRepository
{
    ProductActionResult<Product> DeleteProduct(int id);
    ProductActionResult<ProductCatalog.DataModel.Category> DeleteCategory(int id);
    ProductCatalog.DataModel.Product GetProduct(int id);
    ProductCatalog.DataModel.Category GetCategory(int id);
    System.Linq.IQueryable<ProductCatalog.DataModel.Category> GetCategories();
    System.Linq.IQueryable<ProductCatalog.DataModel.Product> GetProducts();
    System.Linq.IQueryable<ProductCatalog.DataModel.Product> GetProducts(int categoryId);
    ProductActionResult<ProductCatalog.DataModel.Product>
    InsertProduct(ProductCatalog.DataModel.Product product);
    ProductActionResult<ProductCatalog.DataModel.Category>
    InsertCategory(ProductCatalog.DataModel.Category category);
    ProductActionResult<ProductCatalog.DataModel.Product>
    UpdateProduct(ProductCatalog.DataModel.Product product);
    ProductActionResult<ProductCatalog.DataModel.Category>
    UpdateCategory(ProductCatalog.DataModel.Category category);
}
}

```

ProductCatalog.Entities project is the transfer object or custom entities project. this project contains POCO for Product and Category entities used for to and fro of objects between API and data model project. This project is again a simple C# class library holding two POCO classes. The data model project adds reference to this entities project.

The main part of the application lies in Web API project named ProductCatalog.API. I created this project by adding a Web API project type available in Asp.Net web applications category in Visual Studio 2015.



The default structure of the project is kept as it is apart from adding a ProductsController to the project. The ProductsController simply contains a Get method to fetch all the Products from the database.

ProductsController

```
using ProductCatalog.DataModel;
using ProductCatalog.DataModel.Factory;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;

namespace ProductCatalog.API.Controllers
{
    public class ProductsController : ApiController
    {

```

```

        IProductCatalogRepository _productCatalogRepository;
        ProductFactory _productFactory = new ProductFactory();
        public ProductsController()
        {
            _productCatalogRepository = new ProductCatalogRepository(new
ProductCatalogEntities());
        }
        public ProductsController(IProductCatalogRepository productCatalogRepository)
        {
            _productCatalogRepository = productCatalogRepository;
        }
        public IHttpActionResult Get()
        {
            try
            {
                var products = _productCatalogRepository.GetProducts();
                return Ok(products.ToList().Select(p =>
                _productFactory.CreateProduct(p)));
            }
            catch (Exception)
            {
                return InternalServerError();
            }
        }
    }
}

```

The Web API project adds reference to data model and custom entities project. The ProductsController creates an instance of Repository by instantiating it further with ProductCatalogEntities i.e. the data context used

in data model project. The repository instance is used to fetch records in Get method to fetch all product records and further send a response of 200 i.e. OK by converting the database entities to custom entities. You can develop this structure in a more sophisticated and scalable manner by using IOC and dependency injection and may further add more layers to make it more loosely coupled and secure. Follow [this series](#) to develop that kind of architecture in Web API. This is our basic solution that we'll take forward to learn about Content Negotiation in Web API. Download the source code attached.

Now make the ProductCatalog.API project as startup project and run the application. Since we have not defined any route, the application will take the default route defined in WebAPI.Config file lying in App_Start folder of API project. When the application runs you'll get a home page because our default route redirected the application to Home Controller. Just add "api/products" to the URL and you get following screen showing all the product records.



```
<?xml version='1.0'>
<ArrayOfProduct xmlns:i="http://www.w3.org/2001/XMLSchema-instance" >
  <Product>
    <Category>3</Category>
    <ProductId>1</ProductId>
    <ProductName>IPhone</ProductName>
  </Product>
  <Product>
    <Category>3</Category>
    <ProductId>2</ProductId>
    <ProductName>IPad</ProductName>
  </Product>
  <Product>
    <Category>3</Category>
    <ProductId>3</ProductId>
    <ProductName>Tablet</ProductName>
  </Product>
  <Product>
    <Category>3</Category>
    <ProductId>4</ProductId>
    <ProductName>Windows Surface</ProductName>
  </Product>
  <Product>
    <Category>2</Category>
    <ProductId>5</ProductId>
    <ProductName>Leather Jacket</ProductName>
  </Product>
  <Product>
    <Category>2</Category>
    <ProductId>6</ProductId>
    <ProductName>Jeans</ProductName>
  </Product>
  <Product>
    <Category>2</Category>
    <ProductId>7</ProductId>
    <ProductName>T-Shirt</ProductName>
  </Product>
  <Product>
    <Category>2</Category>
    <ProductId>8</ProductId>
    <ProductName>Trousers</ProductName>
  </Product>
</ArrayOfProduct>
```

The result is shown in the plain XML format, but what if the client needs the result in JSON format? How will the client communicate with server to send the result in desired format and how will server understand and process different requests from different clients? Content negotiation is the answer to all these questions. The temporary part of the application is done, let's explore formatters in Web API now.

Formatters in Web API

Now why does this browser return XML? That has to do with how the browser creates a request message. When we surf to a URI that support fetch API, the browser here is the client of the API. A client creates a request message and the REST service response with a response message. Here, the browser creates a request. That must mean that browser somehow requests XML. To have a more closer look to in, let's use Fiddler. Fiddler is a free web debugging tool to inspect services and their requests and responses. Let's invoke the service again, but now using Fiddler and see what the browser has requested. Launch Fiddler and refresh the URL in browser. You can see that every message , may it be request or response has a header and a body.

The screenshot shows the Fiddler Web Debugger interface. The top menu bar includes File, Edit, Rules, Tools, View, Help, and GeoEdge. Below the menu is a toolbar with icons for WinConfig, Replay, Go, Stream, Decode, and session management. The main table displays a list of sessions. The selected session (ID 1) is a GET request to localhost:12333 /api/products with a status of 200 and a body size of 1,974 bytes. The bottom pane shows the request headers and the XML response.

Fiddler Web Debugger

File Edit Rules Tools View Help GeoEdge

WinConfig Replay Go Stream Decode Keep: All sessions Any Process

#	Result	Protocol	Host	URL	Body	Caching	Content-Type
1	200	HTTP	localhost:12333	/api/products	1,974	no-cac...	application/xml

Request Headers

GET /api/products HTTP/1.1

Cache

Cache-Control: max-age=0

Client

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8

Accept-Encoding: gzip, deflate, sdch

Accept-Language: en-US,en;q=0.8

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.106 Safari/537.36

Security

Upgrade-Insecure-Requests: 1

Transport

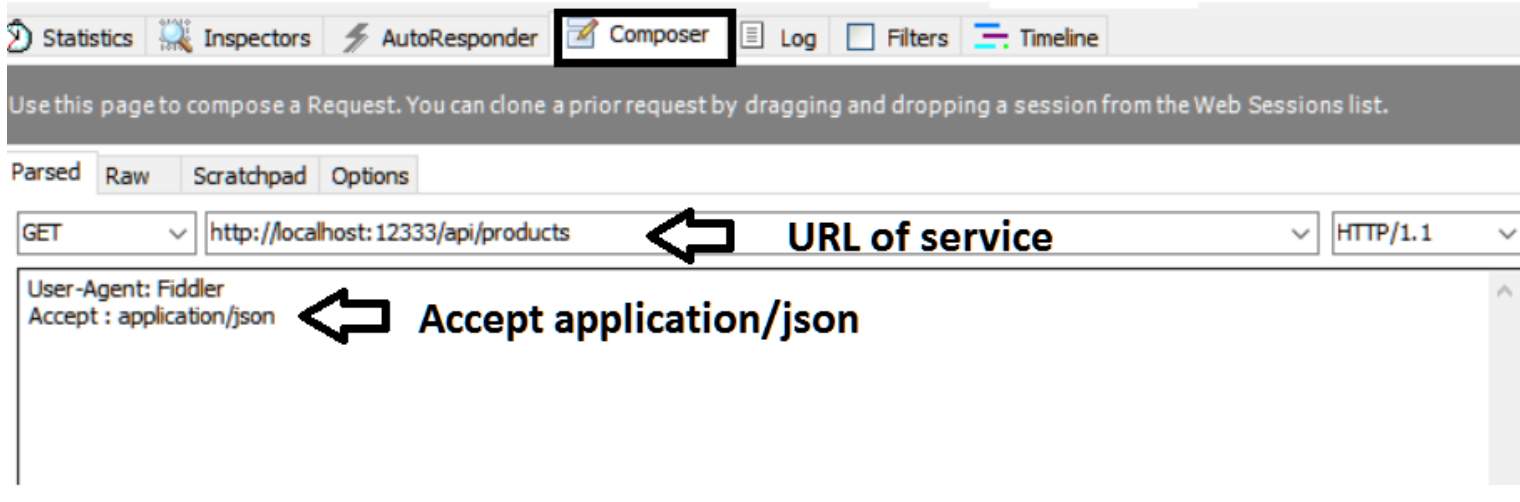
Connection: keep-alive

Get SyntaxView Transformer Headers TextView ImageView HexView Webview Auth Caching Cookies Raw JSON XML

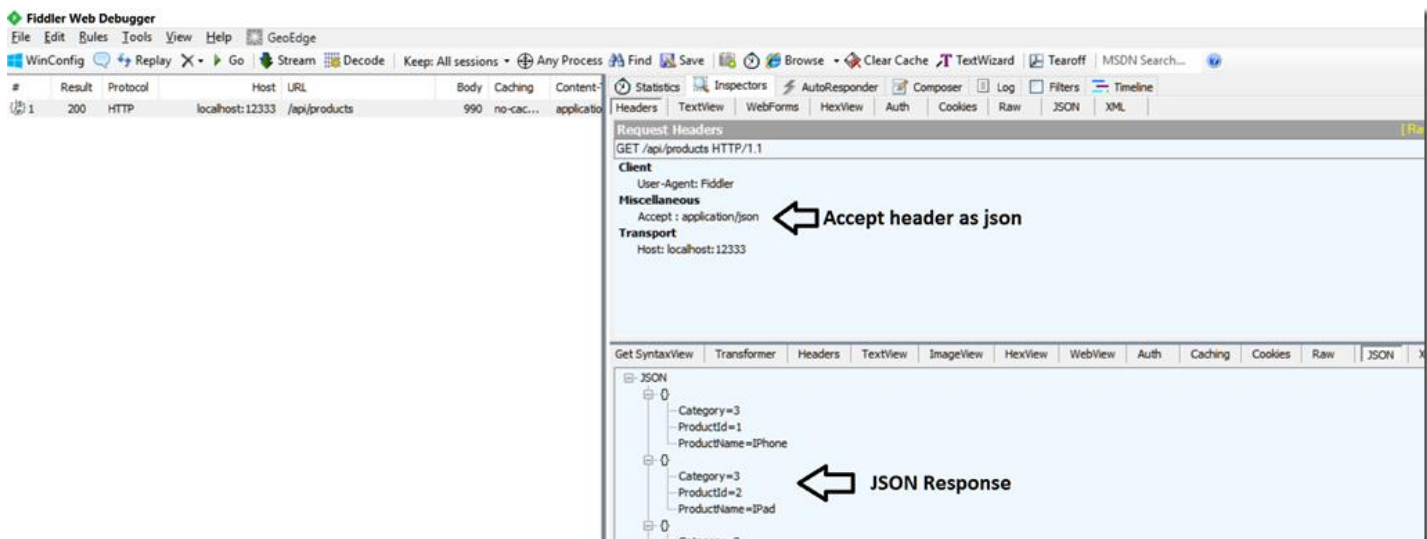
ArrayOfProduct [xmlns:i=http://www.w3.org/2001/XMLSchema-instance xmlns=http://schemas.datacontract.org/2004/07/ProductCatalog.Entities]

- Product
 - Category
 - 3
 - ProductId
 - 1
 - ProductName
 - iPhone
- Product
 - Category

When you see the header collection of request header, it is found that the browser has sent a request with an accept header. Accept header in the request specifies that in what format the client needs the response to be. There is also an Internet Media Type (IME) i.e. text/html , stating the response to be in text html format. It specifies that if prior IME is not available, application/xhtml+xml to be sent. Let's change that using Fiddler. Let's configure our request using Fiddler to get the data in JSON format. Go to Composer tab of Fiddler and invoke the same URL. This time, we'll say that we will accept JSON.



Now look at the response, we see that JSON is returned.



We can also look at a raw response, which makes this even more obvious.



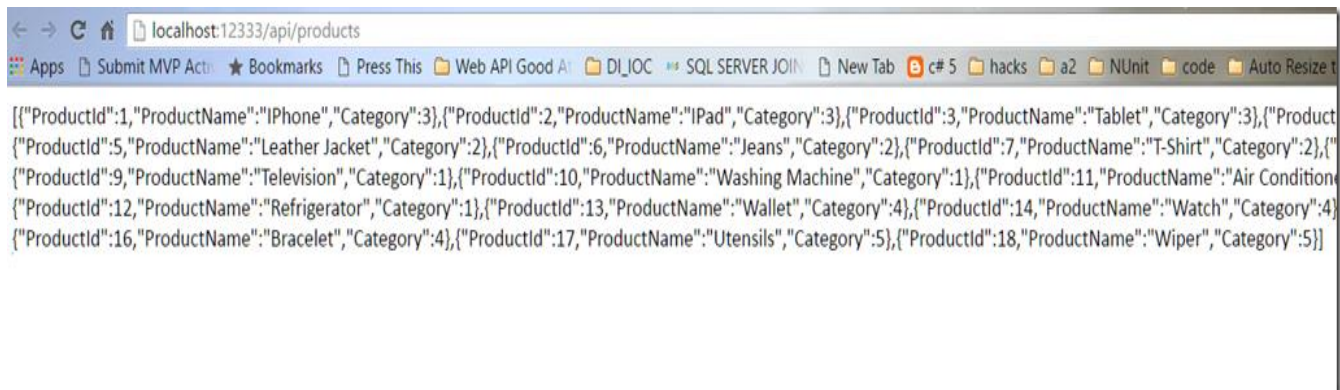
Now it would be nice to have our API automatically return JSON when a browser is the consumer of the API instead of XML.

We can do that. There's two ways to do this, and they both require us to manipulate the supported media types for the formatters supported by our API. We're back in our web API configuration. By default, web API support both XML formatting and JSON formatting. What we now want to do is ensure that JSON format is invoked when a consumer requests text HTML, which is the highest priority IME the browser requests, as we've seen. To do this, we add this media type to the supported media types collection of the JSON formatter.

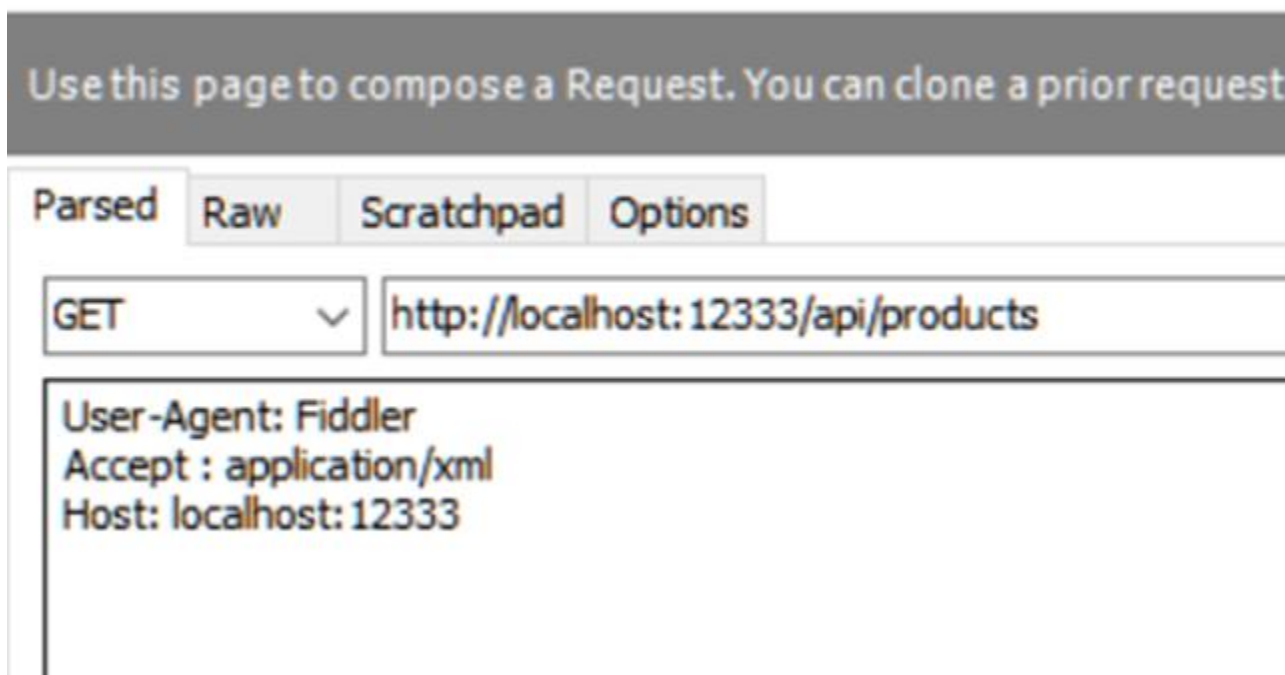
We can access these formatters through `config.Formatters`. Go to the `WebAPI.config` and add JSON formatter's new supported type below the default route defined.

```
config.Formatters.JsonFormatter.SupportedMediaTypes.Add(new  
System.Net.Http.Headers.MediaTypeHeaderValue("text/html"));
```

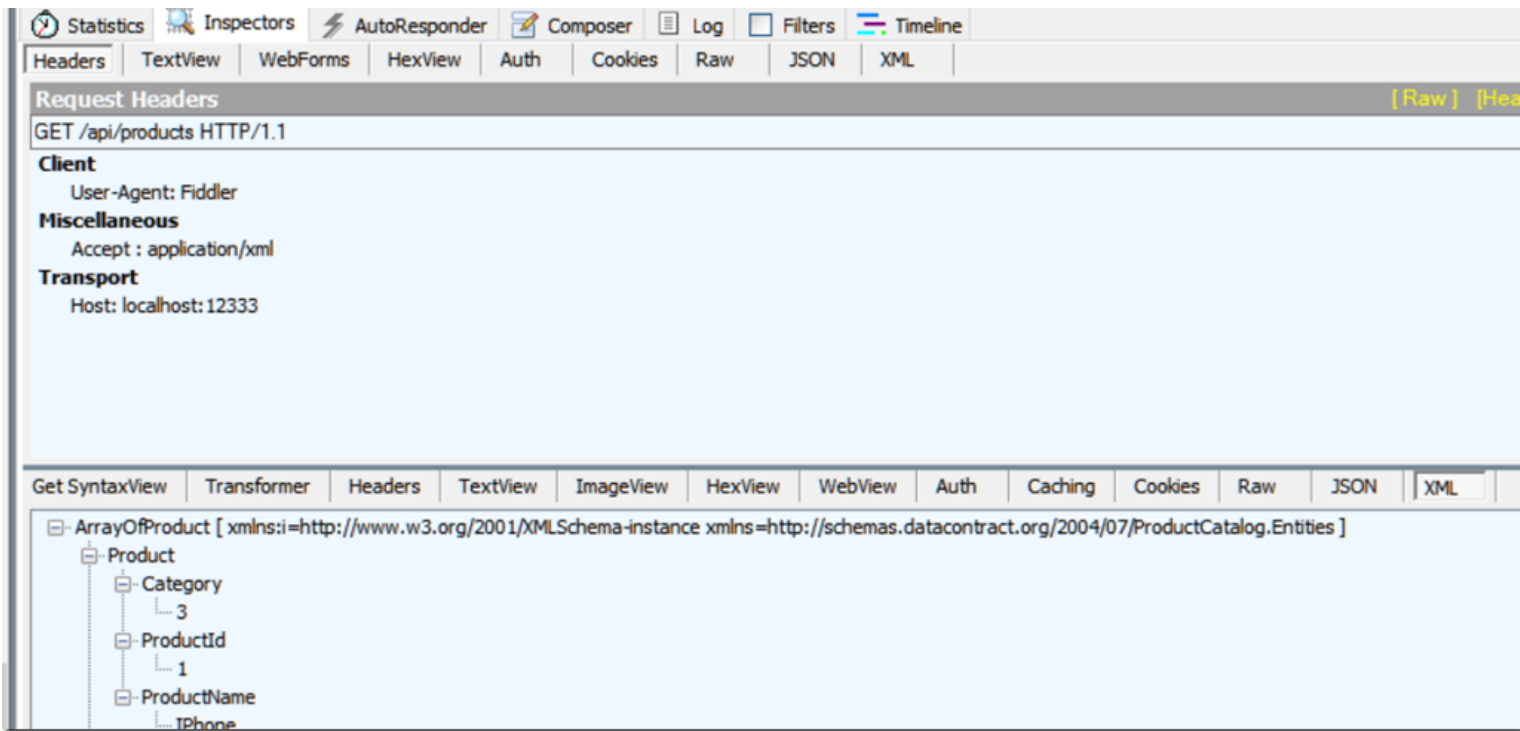
Run the application again in browser.



And this time, the results are returned as JSON. Why? Well, what we've just done means that when a request stated `text/html` as an accept header, the JSON formatter will be invoked for the request because it now contains that media type. Yet, when we ask for `application/xml`, that's a media type that's still supported by the XML formatter, we will still get XML. Let's give that a try with Fiddler. And we're going to say we accept `application/xml`, and there we go.



We've got XML.



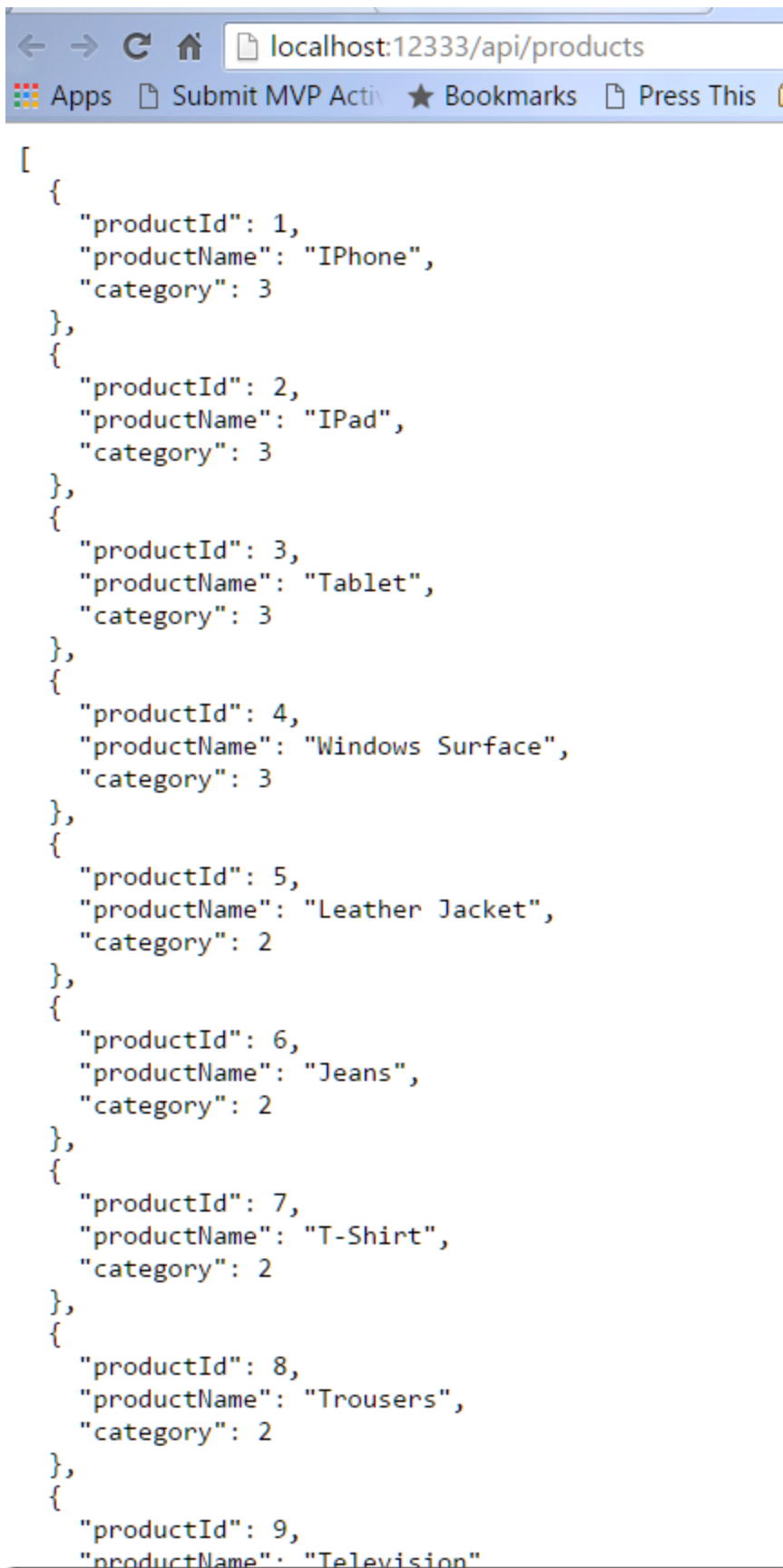
There's another way to do this. Instead of adding text HTML as a support media type to the JSON formatter, we can also simply remove all support media types from the XML formatter.

```
//config.Formatters.JsonFormatter.SupportedMediaTypes.Add(new  
System.Net.Http.Headers.MediaTypeHeaderValue("text/html"));  
  
config.Formatters.XmlFormatter.SupportedMediaTypes.Clear();
```

The difference is that our API now doesn't support returning XML anymore. There's no more support media types. But the best way to handle this is depends. If you want to keep on supporting XML, this isn't a way to do it. On the other hand, if you don't, this is. Not supporting XML is a choice made by a lot of API developers. It removes a possible level of errors. For example, if you want to support XML, you better make sure your XML formatter is bug-free. Anyway, both approaches are viable, and it depends on your requirements. But there's more we can do. The results we just saw in our browser, well, they aren't really nicely formatted, right? I'd like to apply some formatting so the API is more easily discoverable through the browser. We can do that by altering the serializer settings of the JSON formatter. So let's look at that. We can access the serializer settings through `config.Formatters.JsonFormatter.SerializerSettings`. Serializer settings has two interesting properties we want to use for this. First is the formatting property. We can state we want to use indentation when formatting. The second thing I'd like to do is ensure we get results back CamelCased.

```
config.Formatters.JsonFormatter.SerializerSettings.Formatting =  
Newtonsoft.Json.Formatting.Indented;  
  
config.Formatters.JsonFormatter.SerializerSettings.ContractResolver = new  
CamelCasePropertyNamesContractResolver();
```

That's true to contract resolver. And let's set that to a new CamelCased property names contract resolver. And there we go, let's give it a try.



The image shows a web browser window with the address bar displaying `localhost:12333/api/products`. The browser's toolbar includes navigation buttons (back, forward, refresh, home) and a search bar. Below the toolbar, there are tabs for 'Apps', 'Submit MVP Activ', '★ Bookmarks', and 'Press This'. The main content area of the browser displays a JSON array of product objects. The array contains nine objects, each with 'productId', 'productName', and 'category' fields. The products are grouped by category: category 3 (electronics) includes iPhone, iPad, Tablet, and Windows Surface; category 2 (clothing) includes Leather Jacket, Jeans, T-Shirt, and Trousers; and category 1 (home appliances) includes Television.

```
[
  {
    "productId": 1,
    "productName": "iPhone",
    "category": 3
  },
  {
    "productId": 2,
    "productName": "iPad",
    "category": 3
  },
  {
    "productId": 3,
    "productName": "Tablet",
    "category": 3
  },
  {
    "productId": 4,
    "productName": "Windows Surface",
    "category": 3
  },
  {
    "productId": 5,
    "productName": "Leather Jacket",
    "category": 2
  },
  {
    "productId": 6,
    "productName": "Jeans",
    "category": 2
  },
  {
    "productId": 7,
    "productName": "T-Shirt",
    "category": 2
  },
  {
    "productId": 8,
    "productName": "Trousers",
    "category": 2
  },
  {
    "productId": 9,
    "productName": "Television"
  }
]
```

That already looks a lot more discoverable.

Content Negotiation in Web API

We have our solution setup and have an idea of how accept headers and formatters work. Now, let's have a closer look into Content Negotiation. Content Negotiation works more in a similar manner as we saw while learning formatters. Client makes a request with the desired response formats in the header collection and server responds the same. The content negotiator pipeline is aligned with the list of registered media type formatters. When a particular request comes in the pipeline the framework tries to match the first available type with the request and sends the response. `HttpConfiguration` provides `IContentNegotiator` service and pipeline also fetches the list of available formatters in `HttpConfiguration.Formatters`. media type mapping is the first task that pipeline performs for matching. Apart from request header mapping, Web API supports other mappings like `QueryString`, `URIPathExtensions` and `MediaRange` mapping too. This is not limited to these specific mappings, you can also make your custom mapping to achieve the implementation.

Accept Headers

Accept Headers is one of the major criteria for matching the media types. Client can send the request with Accept attribute in the header and define the desired media type as shown below.

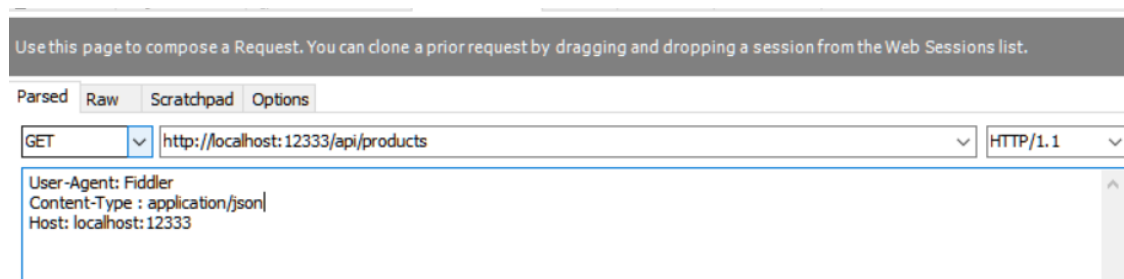
```
GET http://localhost:12333/api/products HTTP/1.1
User-Agent: Fiddler
Accept: application/xml
Host: localhost:12333
```

Accept: application/xml or Accept: application/json

The Content Negotiator takes care of the request and responds accordingly.

Content Type

The request can alternatively contain content type of the body. In case Accept header is not present, the content negotiator looks for the content type of the body and serves the response accordingly. In case both accept headers and content type are present, the precedence is given to accept headers.



Use this page to compose a Request. You can done a prior request by dragging and dropping a session from the Web Sessions list.

Parsed Raw Scratchpad Options

GET http://localhost:12333/api/products HTTP/1.1

User-Agent: Fiddler
Content-Type : application/json
Host: localhost:12333

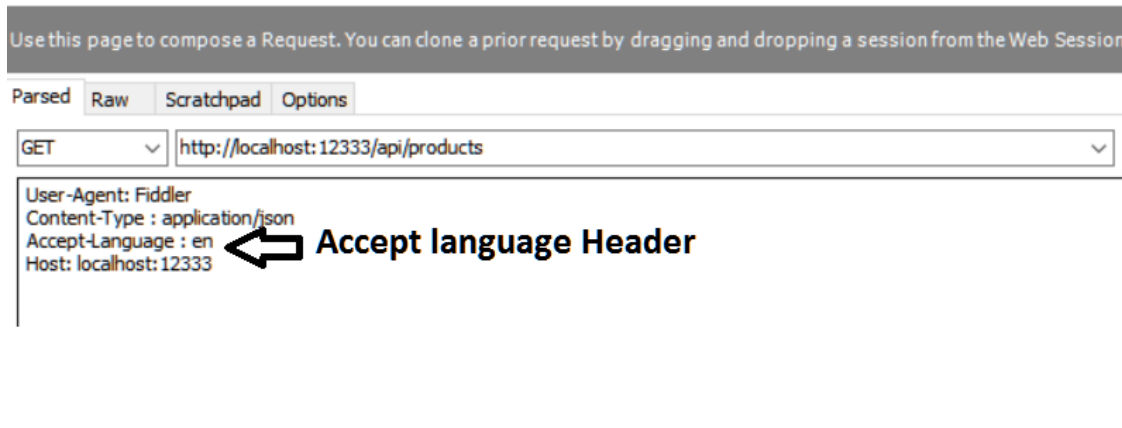
Content-Type : application/json.

Formatters

Formatters comes into picture if neither of the mentioned criteria's like `MediaTypeMappings`, Content Type and Accept Headers are present. In this case the content negotiator pipeline checks the formatters in the `HttpConfiguration.Formatters` collection and in the precedence of order in which they are stored it picks the one and returns response accordingly.

Accept Language Headers

It is very similar to Accept header , but it's purpose is to request the language type from the service. Accept header was used to request the media formatting. Accept language can specify, what language type is requested by the client. The API may have resources like html available to be shared via a representation and in case of multilingual support, the API may contain various versions of html's w.r.t. various languages or messages could also be multi lingual. In these type of cases , a client can request the message or html file as per its culture or language.



Accept-Language : en

The client can also request the language in order of precedence like shown below,

Accept-Language: sv, en-us; q0.8,da;q=0.7

Quality Factor

In the prior example of Accept Language, we see string q=0.7 . q is nothing but termed as Quality Factor in content negotiation. Quality Factor varies from 0.1 to 1.0. The greater the value of quality factor is , more the preference is given to the type. Example of Accept header with quality factor is shown below.

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.7

Accept Encoding Headers

This header specifies the encoding that client accepts i.e. the content encoding , primarily encryption. Some examples of the Accept-Encoding header are gzip, compress, deflate, identity etc. Client can request the type of encoding it needs by just supplying Accept-Encoding : <encoding type> in header request along with the request.

Example : Accept-Encoding : gzip

Accept Charset Headers

This header defines that what type of character encoding is accepted by the client. It specifies basically what type of character sets are acceptable like UTF-8 or ISO 8859-1.

Example : Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.8

Content Negotiation Implementation in Web API

Let's look into how we can implement content negotiation in Web API. `DefaultContentNegotiator` is the name of the class that takes care of content negotiation in Web API. We talked about `IContentNegotiator` service in

the last section. The content negotiator pipeline calls this `IContentNegotiator.Negotiate` service method with the parameters like, the type of object that has to be serialized, the media formatters collection and the associated `Http` request. The return type of this method is `ContentNegotiationResult`, specifying which formatter has to be used and the return media type of the response. For example in our case we can implement the content negotiation in following way in our Web API action.

```
public HttpResponseMessage Get()
{
    try
    {
        var products = _productCatalogRepository.GetProducts();
        var customProducts = products.ToList().Select(p =>
            _productFactory.CreateProduct(p));

        IContentNegotiator negotiator =
            this.Configuration.Services.GetContentNegotiator();

        ContentNegotiationResult result = negotiator.Negotiate(
            typeof(List<Product>), this.Request, this.Configuration.Formatters);
        if (result == null)
        {
            var response = new HttpResponseMessage(HttpStatusCode.NotAcceptable);
            throw new HttpResponseException(response);
        }

        return new HttpResponseMessage()
        {
            Content = new ObjectContent<List<Entities.Product>>(
                customProducts.ToList(), // type of object to be serialized
                result.Formatter,        // The media formatter
                result.MediaType.MediaType // The MIME type
            )
        };
    }
    catch (Exception)
    {
        return new HttpResponseMessage(HttpStatusCode.InternalServerError);
    }
}
```

In code of Action that fetches list of products, I have made some changes to support content negotiation as mentioned in above code. Like explained, we get `IContentNegotiator` service and store in `negotiator` variable. Then `Negotiate` method is called, passing the type of object to be negotiated, the `Http` request and the list of

Formatters. Then an HttpResponseMessage is sent to the client. I tried to capture the best matched media type and best matched formatter in the code, and sent the request from the browser and got following.

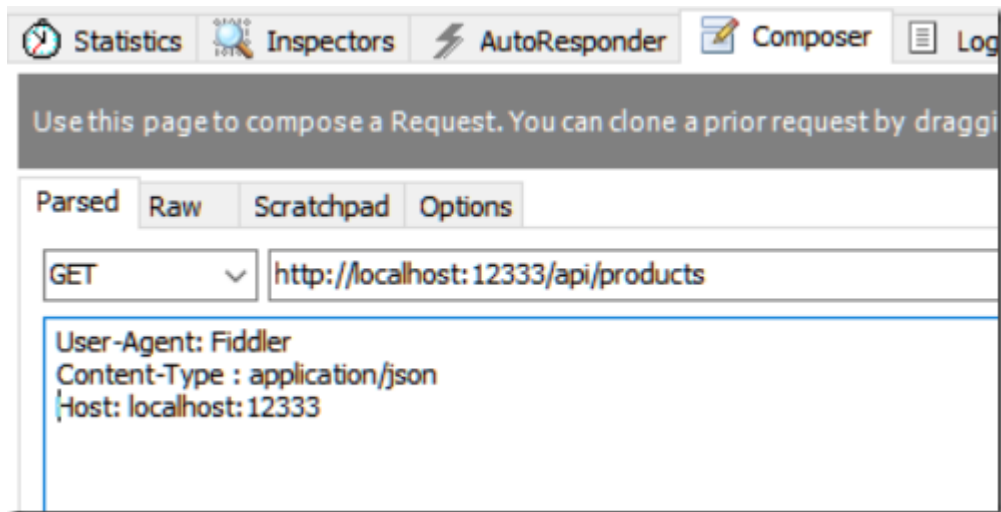
```
private HttpResponseMessage Get()
{
    try
    {
        var products = _productCatalogRepository.GetProducts();
        var customProducts = products.ToList().Select(p => _productFactory.CreateProduct(p));
        IContentNegotiator negotiator = this.Configuration.Services.GetContentNegotiator();

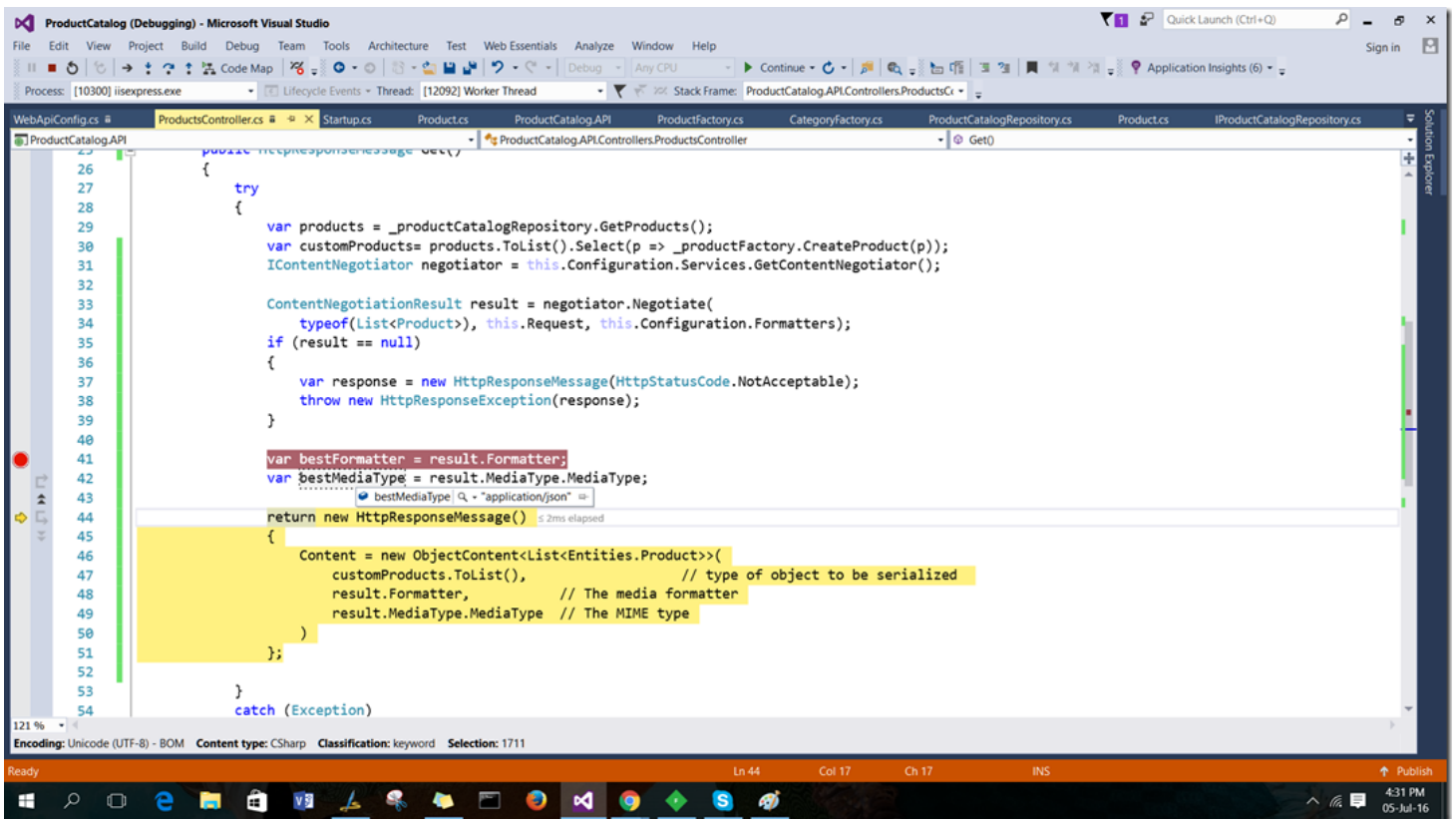
        ContentNegotiationResult result = negotiator.Negotiate(
            typeof(List<Product>), this.Request, this.Configuration.Formatters);
        if (result == null)
        {
            var response = new HttpResponseMessage(HttpStatusCode.NotAcceptable);
            throw new HttpResponseException(response);
        }

        var bestFormatter = result.Formatter;
        var bestMediaType = result.MediaType.MediaType;
        bestMediaType = "application/xml";

        return new HttpResponseMessage()
        {
            Content = new ObjectContent<List<Entities.Product>>(
                customProducts.ToList(), // type of object to be serialized
                result.Formatter, // The media formatter
                result.MediaType.MediaType // The MIME type
            )
        };
    }
}
```

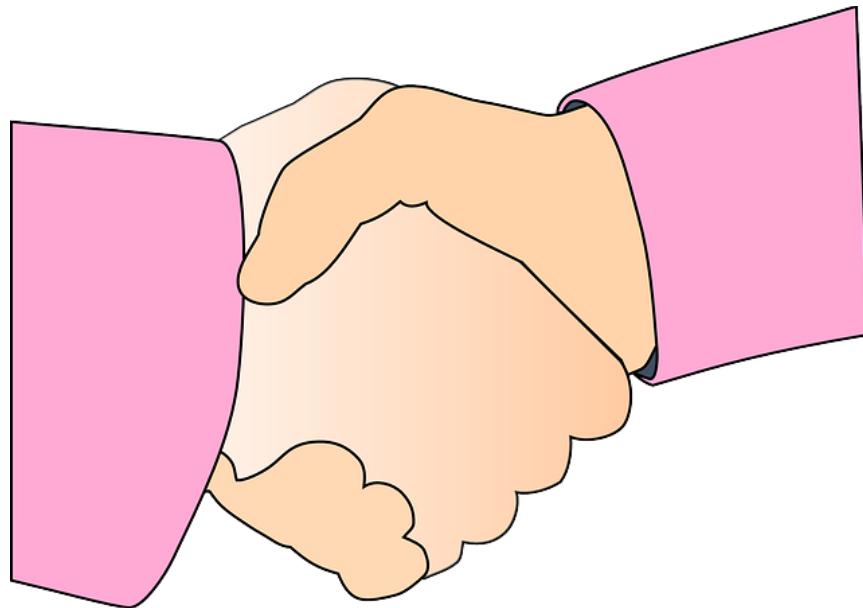
I.e application/xml as the request was from browser with these accept headers. I made the same request with Fiddler by changing the media type to application/json along with the request and got following best media type.





```
26 {
27     try
28     {
29         var products = _productCatalogRepository.GetProducts();
30         var customProducts = products.ToList().Select(p => _productFactory.CreateProduct(p));
31         IContentNegotiator negotiator = this.Configuration.Services.GetContentNegotiator();
32
33         ContentNegotiationResult result = negotiator.Negotiate(
34             typeof(List<Product>), this.Request, this.Configuration.Formatters);
35         if (result == null)
36         {
37             var response = new HttpResponseMessage(HttpStatusCode.NotAcceptable);
38             throw new HttpResponseException(response);
39         }
40
41         var bestFormatter = result.Formatter;
42         var bestMediaType = result.MediaType.MediaType;
43         // bestMediaType | Q - "application/json"
44         return new HttpResponseMessage()
45         {
46             Content = new ObjectContent<List<Entities.Product>>(
47                 customProducts.ToList(), // type of object to be serialized
48                 result.Formatter, // The media formatter
49                 result.MediaType.MediaType // The MIME type
50             )
51         };
52     }
53     catch (Exception)
54     {
55     }
56 }
```

I.e. application/json 😊.



Custom Content Negotiation Implementation in Web API

You can also write your custom content negotiator in Web API by overriding the default one. just create any class that derives from `DefaultContentNegotiator` and override the methods by your own. After which you just need to set up your global configuration in `WebAPI.config` file like shown below.

```
GlobalConfiguration.Configuration.Services.Replace(typeof(IContentNegotiator), new CustomContentNegotiator());
```

Let's see the implementation. Add a class named CustomContentNegotiator and derive it from DefaultContentNegotiator .

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Net.Http.Formatting;
using System.Net.Http.Headers;
using System.Web;

namespace ProductCatalog.API
{
    public class CustomContentNegotiator : DefaultContentNegotiator
    {
    }
}
```

Now add the override Negotiate method in the class to support custom content negotiator.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Net.Http.Formatting;
using System.Net.Http.Headers;
using System.Web;

namespace ProductCatalog.API
{
    public class CustomContentNegotiator : DefaultContentNegotiator
    {
        public override ContentNegotiationResult Negotiate(Type type, HttpRequestMessage request, IEnumerable<MediaTypeFormatter> formatters)
        {
            var result = new ContentNegotiationResult(new JsonMediaTypeFormatter(), new MediaTypeHeaderValue("application/json"));
            return result;
        }
    }
}
```

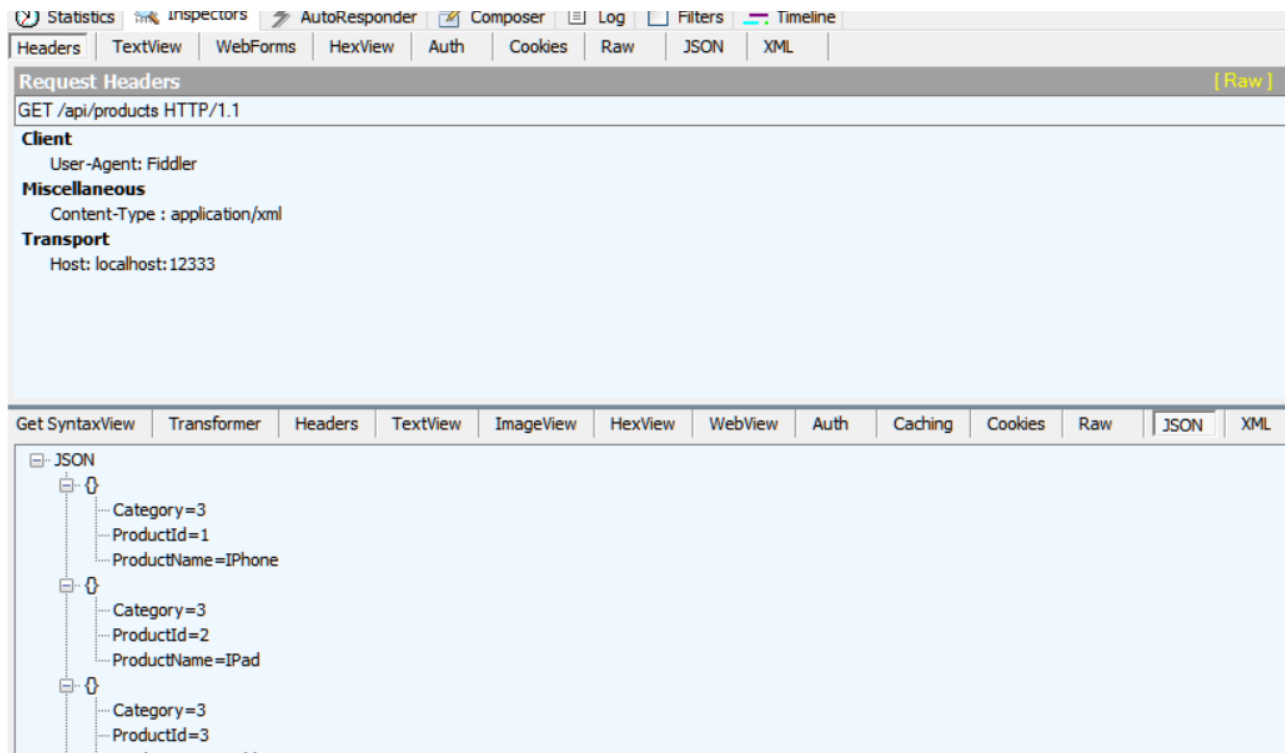
In the above code we are using our custom logic to add application/json format media formatter. Now in WebAPI.config add the following configuration.

```
GlobalConfiguration.Configuration.Services.Replace(typeof(IContentNegotiator), new CustomContentNegotiator());
```

Now run the application and you get json result at the browser.



Even if you try to run the service from Fiddler with accept header as application/xml, you'll get the JSON result. Which shows our custom formatter is working.



Conclusion

This article explained about content negotiation in Asp.NET Web API and its practical implementation. This is a very smart topic and very important too for REST development. Content negotiation helps services to become more robust and extensible. It makes the services far better and standardizes the implementation.

References

Here are some further good reads about Content negotiation in Web API.

<http://www.asp.net/web-api/overview/formats-and-model-binding/content-negotiation>

<http://www.strathweb.com/2012/07/everything-you-want-to-know-about-asp-net-web-api-content-negotiation/>

<https://msdn.microsoft.com/en-us/magazine/dn574797.aspx?f=255&MSPPError=-2147217396>

For more informative articles, reach out to my blog at [CodeTeddy](#).