**In this video we will discuss**

* What is MediaTypeFormatter
* How to return only JSON from ASP.NET Web API Service irrespective of the Accept header value
* How to return only XML from ASP.NET Web API Service irrespective of the Accept header value
* How to return JSON instead of XML from ASP.NET Web API Service when a request is made from the browser
* Point you to an article that describes how to return CSV formatted data from ASP.NET Web API

**What is MediaTypeFormatter**  
MediaTypeFormatter is an abstract class from which JsonMediaTypeFormatter and XmlMediaTypeFormatter classes inherit from. JsonMediaTypeFormatter handles JSON and XmlMediaTypeFormatter handles XML.  
  
**How to return only JSON from ASP.NET Web API Service irrespective of the Accept header value**  
Include the following line in **Register**() method of WebApiConfig.cs file in **App\_Start**folder. This line of code completely removes XmlFormatter which forces ASP.NET Web API to always return JSON irrespective of the Accept header value in the client request. Use this technique when you want your service to support only JSON and not XML.   
  
With this change, irrespective of the Accept header value (application/xml or application/json), the Web API service is always going to return JSON.

config.Formatters.Remove(config.Formatters.XmlFormatter);

**How to return only XML from ASP.NET Web API Service irrespective of the Accept header value**  
Include the following line in Register() method of WebApiConfig.cs file in App\_Start folder. This line of code completely removes JsonFormatter which forces ASP.NET Web API to always return XML irrespective of the Accept header value in the client request. Use this technique when you want your service to support only XML and not JSON.  
  
config.Formatters.Remove(config.Formatters.JsonFormatter);  
  
With this change, irrespective of the Accept header value (application/xml or application/json), the Web API service is always going to return XML.  
  
**How to return JSON instead of XML from ASP.NET Web API Service when a request is made from the browser.**  
So here is what we want the service to do  
1. When a request is issued from the browser, the web API service should return JSON instead of XML.   
2. When a request is issued from a tool like fiddler the Accept header value should be respected. This means if the Accept header is set to application/xml the service should return XML and if it is set to application/json the service should return JSON.  
  
There are 2 ways to achieve this  
  
**Approach 1 :**Include the following line in Register() method of WebApiConfig.cs file in App\_Start folder. This tells ASP.NET Web API to use JsonFormatter when a request is made for text/html which is the default for most browsers. The problem with this approach is that Content-Type header of the response is set to text/html which is misleading.

config.Formatters.JsonFormatter.SupportedMediaTypes

    .Add(new MediaTypeHeaderValue("text/html"));

**Approach 2 :** Include the following class in WebApiConfig.cs file in App\_Start folder. 

public class CustomJsonFormatter : JsonMediaTypeFormatter

{

    public CustomJsonFormatter()

    {

        this.SupportedMediaTypes.Add(new MediaTypeHeaderValue("text/html"));

    }

    public override void SetDefaultContentHeaders(Type type, HttpContentHeadersheaders, MediaTypeHeaderValue mediaType)

    {

        base.SetDefaultContentHeaders(type, headers, mediaType);

        headers.ContentType = new MediaTypeHeaderValue("application/json");

    }

}

**Register the formatter:** Place the following line in Register() method of WebApiConfig.cs file in App\_Start folder  
config.Formatters.Add(new CustomJsonFormatter());  
  
With these 2 changes, when a request is issued from the browser you will get JSON formatted data and the Content-Type header of the response is also set to application/json. If you are using tools like fiddler and if you set Accept header to application/xml you will still get XML formatted data.  
  
ASP.NET Web API is an extinsible framework. This means you can also plugin your own custom formatter. For example, if you want the response to be in CSV format, you can create custom CSVMediaTypeFormatter that inherits from the base abstract class MediaTypeFormatter . The following article describes how to do this.  
<http://www.tugberkugurlu.com/archive/creating-custom-csvmediatypeformatter-in-asp-net-web-api-for-comma-separated-values-csv-format>

s I tried to explain on my previous [MediaTypeFormatters With MediaTypeMappings](http://www.tugberkugurlu.com/archive/asp-net-web-api-mediatypeformatters-with-mediatypemappings" \o "http://www.tugberkugurlu.com/archive/asp-net-web-api-mediatypeformatters-with-mediatypemappings" \t "_blank) post, formatters play a huge role inside the ASP.NET Web API processing pipeline. As Web API framework programming model is so similar to MVC framework, I kind of want to see formatters as views. Formatters handles serializing and deserializing strongly-typed objects into specific format.

I wanted to create CSVMediaTypeFormatter to hook it up for list of objects and I managed to get it working. After I created it, I saw the great [Media Formatters](http://www.asp.net/web-api/overview/formats-and-model-binding/media-formatters) post on ASP.NET web site which does the same thing. I was like "Man, come on!" and I noticed that formatter meant to be used with a specific object, so I figured there is still a validity in my implementation.

Here is the drill:

First of all we need to create a class which will be derived from [MediaTypeFormatter](http://msdn.microsoft.com/en-us/library/system.net.http.formatting.mediatypeformatter(v=vs.108).aspx" \o "http://msdn.microsoft.com/en-us/library/system.net.http.formatting.mediatypeformatter(v=vs.108).aspx" \t "_blank) abstract class. Here is the class with its constructors:

public class CSVMediaTypeFormatter : MediaTypeFormatter {

public CSVMediaTypeFormatter() {

SupportedMediaTypes.Add(new MediaTypeHeaderValue("text/csv"));

}

public CSVMediaTypeFormatter(

MediaTypeMapping mediaTypeMapping) : this() {

MediaTypeMappings.Add(mediaTypeMapping);

}

public CSVMediaTypeFormatter(

IEnumerable<MediaTypeMapping> mediaTypeMappings) : this() {

foreach (var mediaTypeMapping in mediaTypeMappings) {

MediaTypeMappings.Add(mediaTypeMapping);

}

}

}

Above, no matter which constructor you use, we always add **text/csv** media type to be supported for this formatter. We also allow custom **MediaTypeMappings**to be injected.

Now, we need to override two methods: [MediaTypeFormatter.CanWriteType](http://msdn.microsoft.com/en-us/library/system.net.http.formatting.mediatypeformatter.canwritetype(v=vs.108).aspx) and [MediaTypeFormatter.OnWriteToStreamAsync](http://msdn.microsoft.com/en-us/library/system.net.http.formatting.mediatypeformatter.onwritetostreamasync(v=vs.108).aspx).

First of all, here is the CanWriteType method implementation. What this method needs to do is to determine if the type of the object is supported with this formatter or not in order to write it.

protected override bool CanWriteType(Type type) {

if (type == null)

throw new ArgumentNullException("type");

return isTypeOfIEnumerable(type);

}

private bool isTypeOfIEnumerable(Type type) {

foreach (Type interfaceType in type.GetInterfaces()) {

if (interfaceType == typeof(IEnumerable))

return true;

}

return false;

}

What this does here is to check if the object has implemented the IEnumerable interface. If so, then it is cool with that and can format the object. If not, it will return false and framework will ignore this formatter for that particular request.

And finally, here is the actual implementation. We need to do some work with reflection here in order to get the property names and values out of the value parameter which is a type of object:

protected override Task OnWriteToStreamAsync(

Type type,

object value,

Stream stream,

HttpContentHeaders contentHeaders,

FormatterContext formatterContext,

TransportContext transportContext) {

writeStream(type, value, stream, contentHeaders);

var tcs = new TaskCompletionSource<int>();

tcs.SetResult(0);

return tcs.Task;

}

private void writeStream(Type type, object value, Stream stream, HttpContentHeaders contentHeaders) {

//NOTE: We have check the type inside CanWriteType method

//If request comes this far, the type is IEnumerable. We are safe.

Type itemType = type.GetGenericArguments()[0];

StringWriter \_stringWriter = new StringWriter();

\_stringWriter.WriteLine(

string.Join<string>(

",", itemType.GetProperties().Select(x => x.Name )

)

);

foreach (var obj in (IEnumerable<object>)value) {

var vals = obj.GetType().GetProperties().Select(

pi => new {

Value = pi.GetValue(obj, null)

}

);

string \_valueLine = string.Empty;

foreach (var val in vals) {

if (val.Value != null) {

var \_val = val.Value.ToString();

//Check if the value contans a comma and place it in quotes if so

if (\_val.Contains(","))

\_val = string.Concat("\"", \_val, "\"");

//Replace any \r or \n special characters from a new line with a space

if (\_val.Contains("\r"))

\_val = \_val.Replace("\r", " ");

if (\_val.Contains("\n"))

\_val = \_val.Replace("\n", " ");

\_valueLine = string.Concat(\_valueLine, \_val, ",");

} else {

\_valueLine = string.Concat(string.Empty, ",");

}

}

\_stringWriter.WriteLine(\_valueLine.TrimEnd(','));

}

var streamWriter = new StreamWriter(stream);

streamWriter.Write(\_stringWriter.ToString());

}

We are partially done. Now, we need to make use out of this. I registered this formatter into the pipeline with the following code inside Global.asax **Application\_Start**method:

GlobalConfiguration.Configuration.Formatters.Add(

new CSVMediaTypeFormatter(

new QueryStringMapping("format", "csv", "text/csv")

)

);

On my sample application, when you navigate to **/api/cars?format=csv**, it will get you a CSV file but without an extension. Go ahead and add the **csv** extension.