RESTful Day #9: Extending OData support in ASP.NET Web APIs.

Table of Contents

Table of Contents ***1***

Introduction ***1***

Roadmap ***2***

OData ***2***

Query Options ***3***

Setup Solution ***4***

OData Endpoints ***6***

$top **9**

$filter **11**

$orderby **15**

$orderby with $top **19**

$skip **20**

Standard filter operators ***21***

Standard query functions ***22***

Paging ***22***

QueryOptions Constraints ***23***

AllowedQueryOptions **24**

AllowedOrderByProperties **24**

AllowedLogicalOperators **26**

AllowedArithmeticOperators **27**

Conclusion ***27***

References ***27***

Introduction

This is the last article of the RESTful series in which I'll explain how you can leverage OData capabilities in Asp.net WebAPI. I'll explain what OData is and we'll create OData enabled RESTful services. I'll try to keep the article very concise with less theory and more practical implementations.

Roadmap

Following is the roadmap I have setup to learn WebAPI step by step,

- RESTful Day #1: Enterprise-level application architecture with Web APIs using Entity Framework, Generic Repository pattern and Unit of Work.
- RESTful Day #2: Inversion of control using dependency injection in Web APIs using Unity Container and Bootstrapper
- RESTful Day #3: Resolve dependency of dependencies using Inversion of Control and dependency injection in ASP.Net Web APIs with Unity Container and Managed Extensibility Framework (MEF)
- RESTful Day #4: Custom URL Re-Writing/Routing using Attribute Routes in MVC 4 Web APIs
- RESTful Day #5: Basic Authentication and Token-based custom Authorization in Web APIs using Action Filters
- RESTful Day #6: Request logging and Exception handing/logging in Web APIs using Action Filters, Exception Filters and NLog
- RESTful Day #7: Unit Testing and Integration Testing in WebAPI using NUnit and Moq framework (Part1)
- RESTful Day #8: Unit Testing and Integration Testing in WebAPI using NUnit and Moq framework (Part 2)
- **RESTful Day #9: Extending OData support in ASP.NET Web APIs**

I'll purposely use Visual Studio 2010 and .net Framework 4.0 because there are few implementations that are very hard to find in .Net Framework 4.0, but I'll make it easy by showing how we can do it.

OData

OData is a protocol that provides a flexibility of creating query able REST services. It provides certain query options through which the on demand data can be fetched from the server by the client over HTTP. Following is the definition from asp.net,

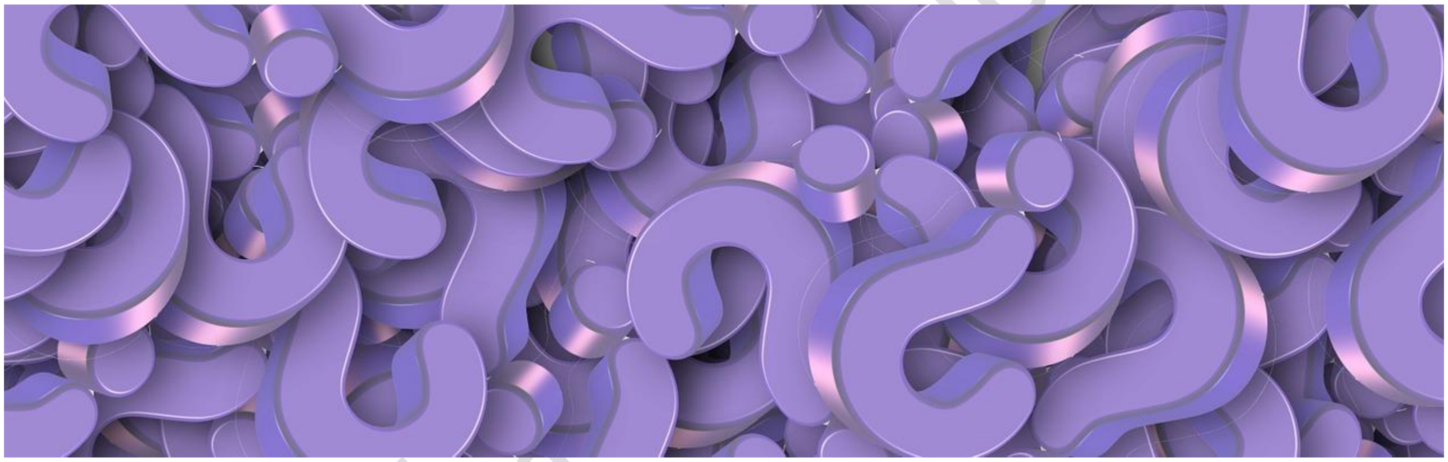
"The Open Data Protocol (OData) is a data access protocol for the web. OData provides a uniform way to query and manipulate data sets through CRUD operations (create, read, update, and delete)."

More elaborated from,

"OData defines parameters that can be used to modify an OData query. The client sends these parameters in the query string of the request URI. For example, to sort the results, a client uses the $orderby parameter:

http://localhost/Products?$orderby=Name

The OData specification calls these parameters query options. You can enable OData query options for any Web API controller in your project — the controller does not need to be an OData endpoint. This gives you a convenient way to add features such as filtering and sorting to any Web API application.
”
Suppose our product table in the database contains more than 50000 products and we want to fetch only top 50 products based on certain conditions like product id or price or name, then as per our current implementation of the service, I’ll have to fetch all the products from the server database and filter them on client or another option could be that I fetch the data at server only and filter the same and send the filtered data to client. In both the cases I am bearing a cost of writing an extra code of filtering the data. Here comes OData in picture. OData allows you to create services that are query able. If the endpoints of the exposed services are OData enabled or supports OData query options then the service implementation would be in such a way that it considers the OData request and process it accordingly. So had that request for 50 records been an OData request, the service would have fetched only 50 records from the server. Not only filtering, but OData provides features like searching, sorting, skipping the data, selecting the data too. I’ll explain the concept with practical implementation. We’ll use our already created service and modify them to be enabled for OData query options.



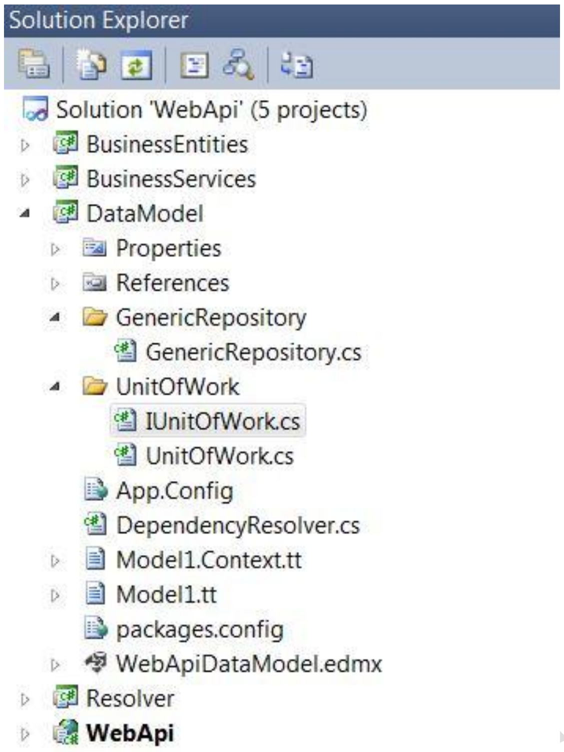
Query Options

Following are the OData query options that asp.net WebAPI supports,

1. **$orderby**: Sorts the fetched record in particular order like ascending or descending.
2. **$select:** Selects the columns or properties in the result set. Specifies which all attributes or properties to include in the fetched result.
3. **$skip:** Used to skip the number of records or results. For e.g. I want to skip first 100 records from the database while fetching complete table data, then I can make use of $skip.
4. **$top:** Fetches only top n records. For e.g. I want to fetch top 10 records from the database, then my particular service should be OData enabled to support $top query option.
5. **$expand**: Expands the related domain entities of the fetched entities.
6. **$filter**: Filters the result set based on certain conditions, it is like where clause of LINQ. For e.g. I want to fetch the records of 50 students who have scored more than 90% marks, and then I can make use of this query option.
7. **$inlinecount**: This query option is mostly used for pagination at client side. It tells the count of total entities fetched from the server to the client.

Setup Solution

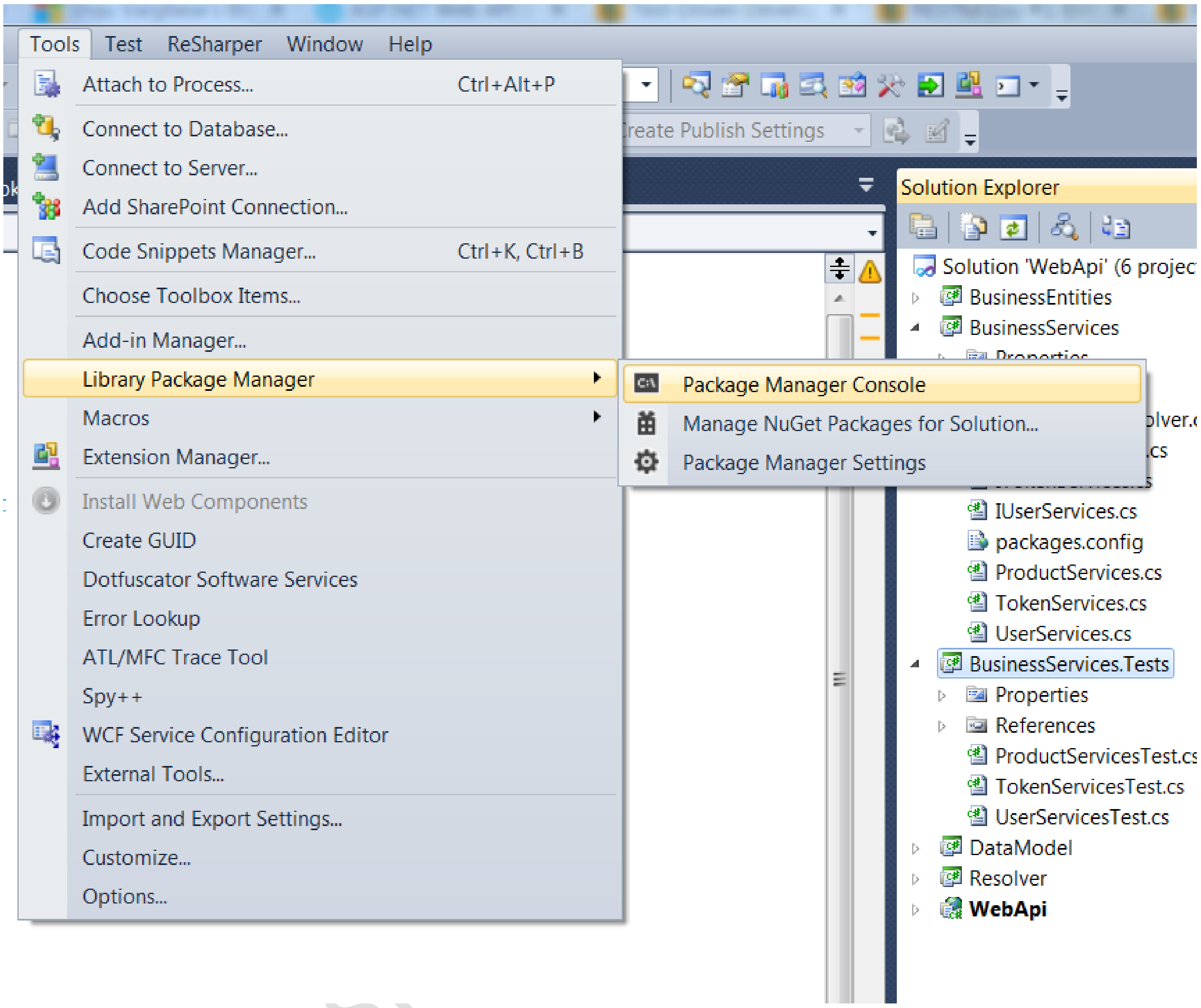
When you take the code base from my last article and open it in visual studio, you'll see the project structure something like as shown in below image,



```
Solution 'WebApi' (5 projects)
  BusinessEntities
  BusinessServices
  DataModel
    Properties
    References
    GenericRepository
      GenericRepository.cs
    UnitOfWork
      IUnitOfWork.cs
      UnitOfWork.cs
    App.Config
    DependencyResolver.cs
    Model1.Context.tt
    Model1.tt
    packages.config
    WebApiDataModel.edmx
  Resolver
  WebApi
```

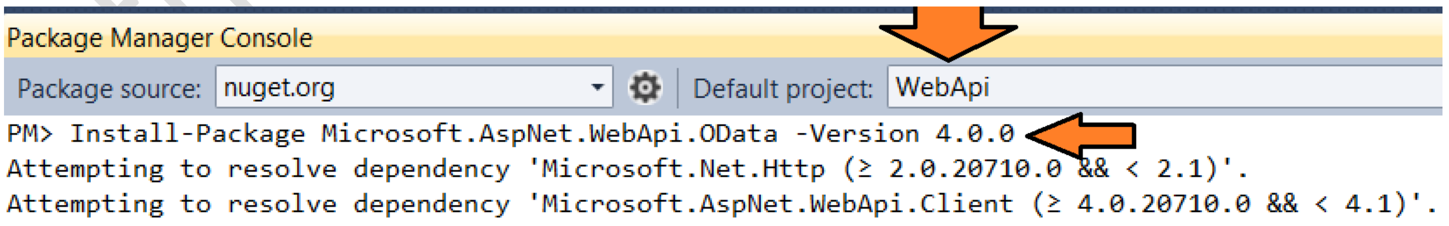
The solution contains the WebAPI application and related projects.

Step1: Click on Tools-> Library Package manager-> Package manager console



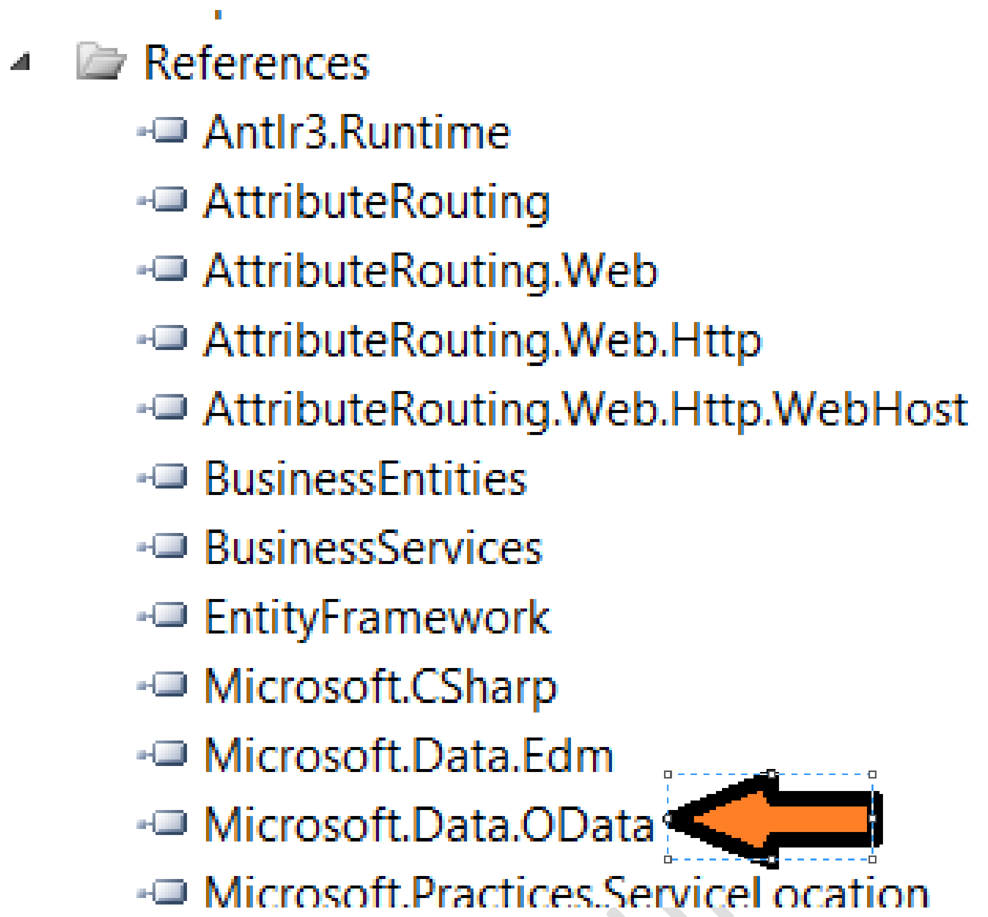
Step2: In Package manager console, select default project as WebApi and run the command: `Install-Package Microsoft.AspNet.WebApi.OData -Version 4.0.0`

Note that, since we are using VS 2010 and .Net framework 4.0, we need to install OData libraries compatible to it.



```
PM> Install-Package Microsoft.AspNet.WebApi.OData -Version 4.0.0
Attempting to resolve dependency 'Microsoft.Net.Http (≥ 2.0.20710.0 && < 2.1)'.
Attempting to resolve dependency 'Microsoft.AspNet.WebApi.Client (≥ 4.0.20710.0 && < 4.1)'.
```

The command will download few dependent packages and reference the dll in your project references. You'll get the OData reference dll in your project references,



Our project is set to make OData endpoints. You can create new services. I'll modify my existing services to demonstrate the OData working.

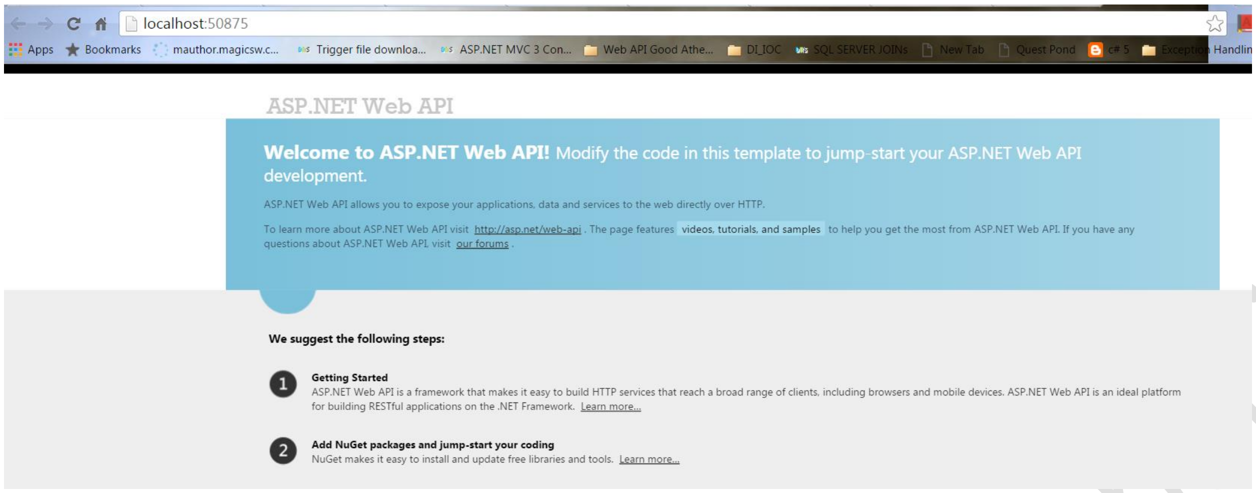
OData Endpoints

Open the ProductController class in WebAPI project and got to Get() method. This method fetches all the product records from the database. Following is the code,

```
[GET("allproducts")]
[GET("all")]
public HttpResponseMessage Get()
{
    var products = _productServices.GetAllProducts();
    var productEntities = products as List<ProductEntity> ?? products.ToList();
    if (productEntities.Any())
        return Request.CreateResponse(HttpStatusCode.OK, productEntities);
    throw new ApiDataException(1000, "Products not found", HttpStatusCode.NotFound);
}
```

Let's run the code through test client,

Just run the application, we get,



localhost:50875

Apps Bookmarks mauthor.magicsw.c... Trigger file downloa... ASP.NET MVC 3 Con... Web API Good Athe... DI_IOC SQL SERVER JOINs New Tab Quest Pond c# 5 Exception Handlin

ASP.NET Web API

Welcome to ASP.NET Web API! Modify the code in this template to jump-start your ASP.NET Web API development.

ASP.NET Web API allows you to expose your applications, data and services to the web directly over HTTP.

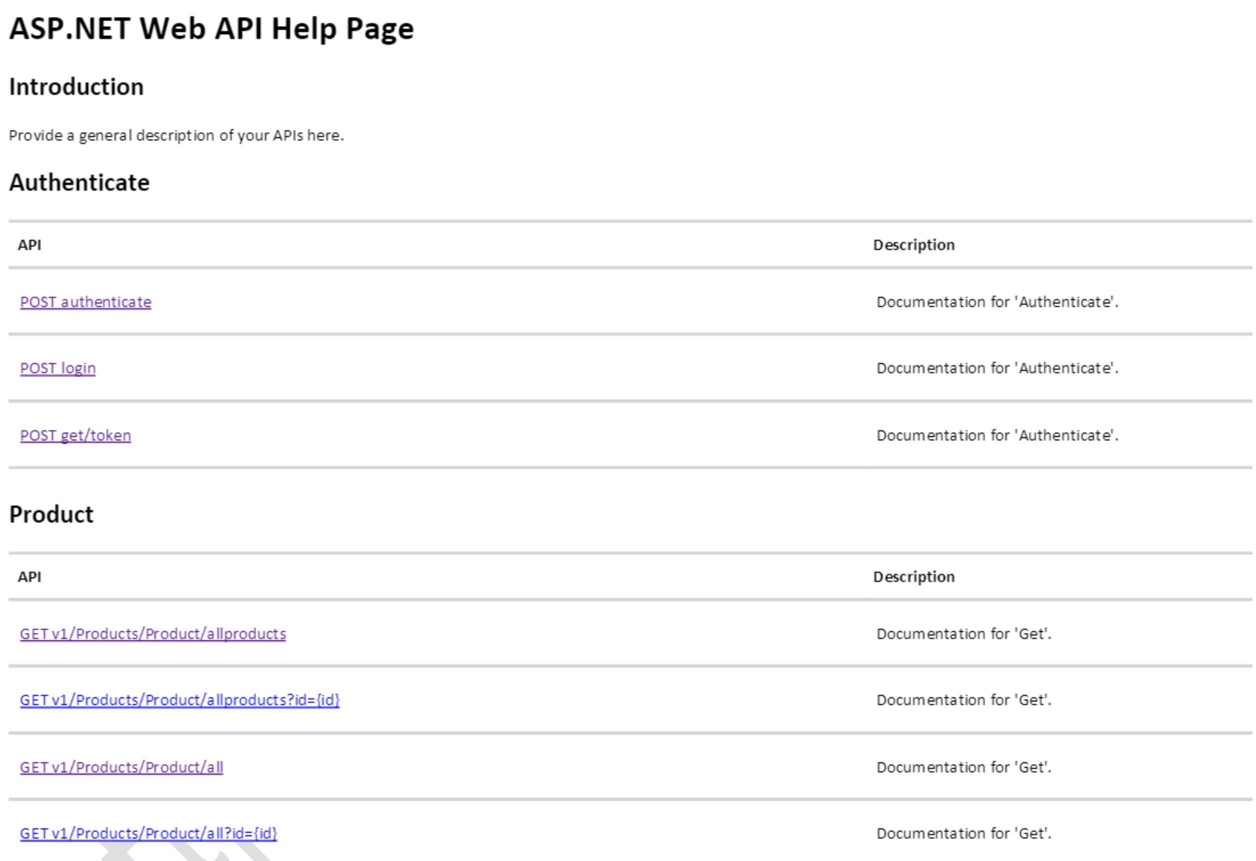
To learn more about ASP.NET Web API visit http://asp.net/web-api . The page features videos, tutorials, and samples to help you get the most from ASP.NET Web API. If you have any questions about ASP.NET Web API, visit our forums .

We suggest the following steps:

1 **Getting Started**
ASP.NET Web API is a framework that makes it easy to build HTTP services that reach a broad range of clients, including browsers and mobile devices. ASP.NET Web API is an ideal platform for building RESTful applications on the .NET Framework. Learn more...

2 **Add NuGet packages and jump-start your coding**
NuGet makes it easy to install and update free libraries and tools. Learn more...

Append/help in the URL and press enter, you'll see the test client.



ASP.NET Web API Help Page

Introduction

Provide a general description of your APIs here.

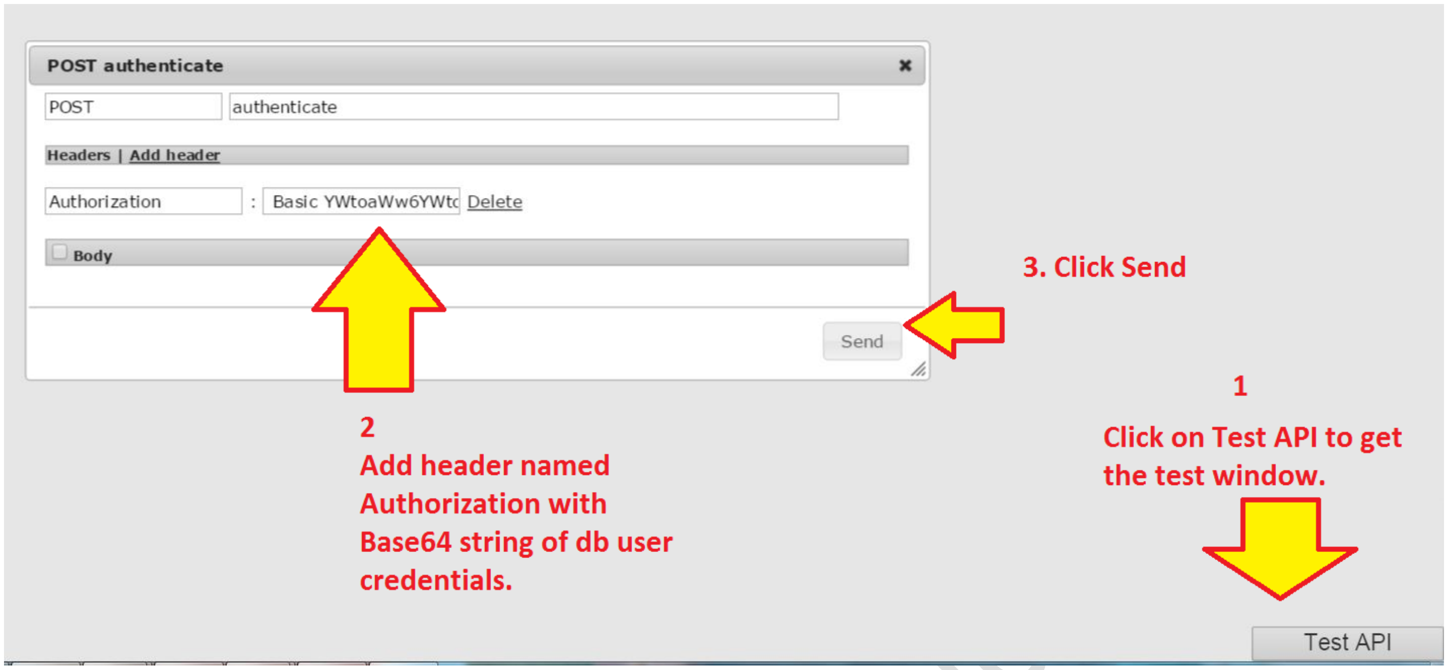
Authenticate

API	Description
POST authenticate	Documentation for 'Authenticate'.
POST login	Documentation for 'Authenticate'.
POST get/token	Documentation for 'Authenticate'.

Product

API	Description
GET v1/Products/Product/allproducts	Documentation for 'Get'.
GET v1/Products/Product/allproducts?id={id}	Documentation for 'Get'.
GET v1/Products/Product/all	Documentation for 'Get'.
GET v1/Products/Product/all?id={id}	Documentation for 'Get'.

Since our product Controller is secured, we need to get an authenticated token from the service and use the same to access product Controller methods. To read about WebAPI security, refer this article. Click on POST authenticate API method and get the TestAPI page of test client,


POST authenticate

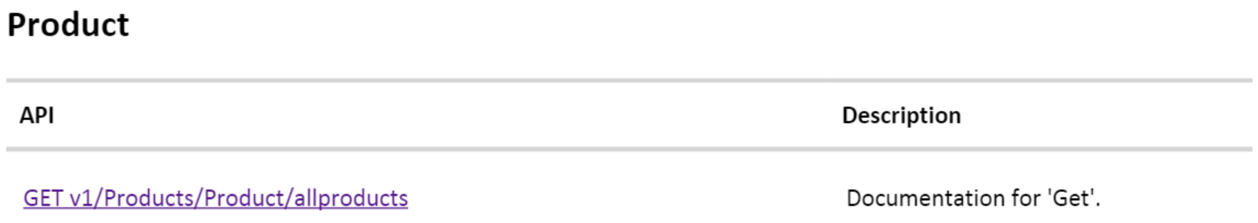
POST	authenticate

Headers | Add header

Authorization	Basic YWtoaWw6YWtc

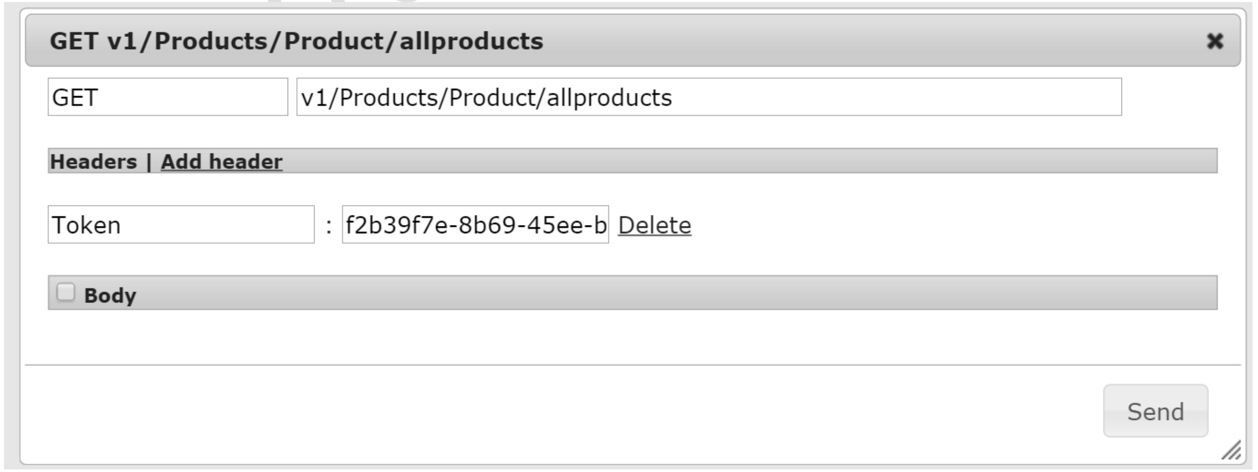
×
Delete
Body
Send
3. Click Send
2
Add header named Authorization with Base64 string of db user credentials.
1
Click on Test API to get the test window.
Test API

Let's send the request with credentials now. Just add a header too with the request. Header should be like ,
Authorization : Basic YWtoaWw6YWtoaWw=
Here **"YWtoaWw6YWtoaWw="** is my Base64 encoded user name and password in database i.e. akhil:akhil
If authorized, you'll get a Token.Just save that token for making further calls to product Controller.
Now open your product controller's "allproducts" endpoint in test client,


Product

API	Description
GET v1/Products/Product/allproducts	Documentation for 'Get'.

Test the endpoint,

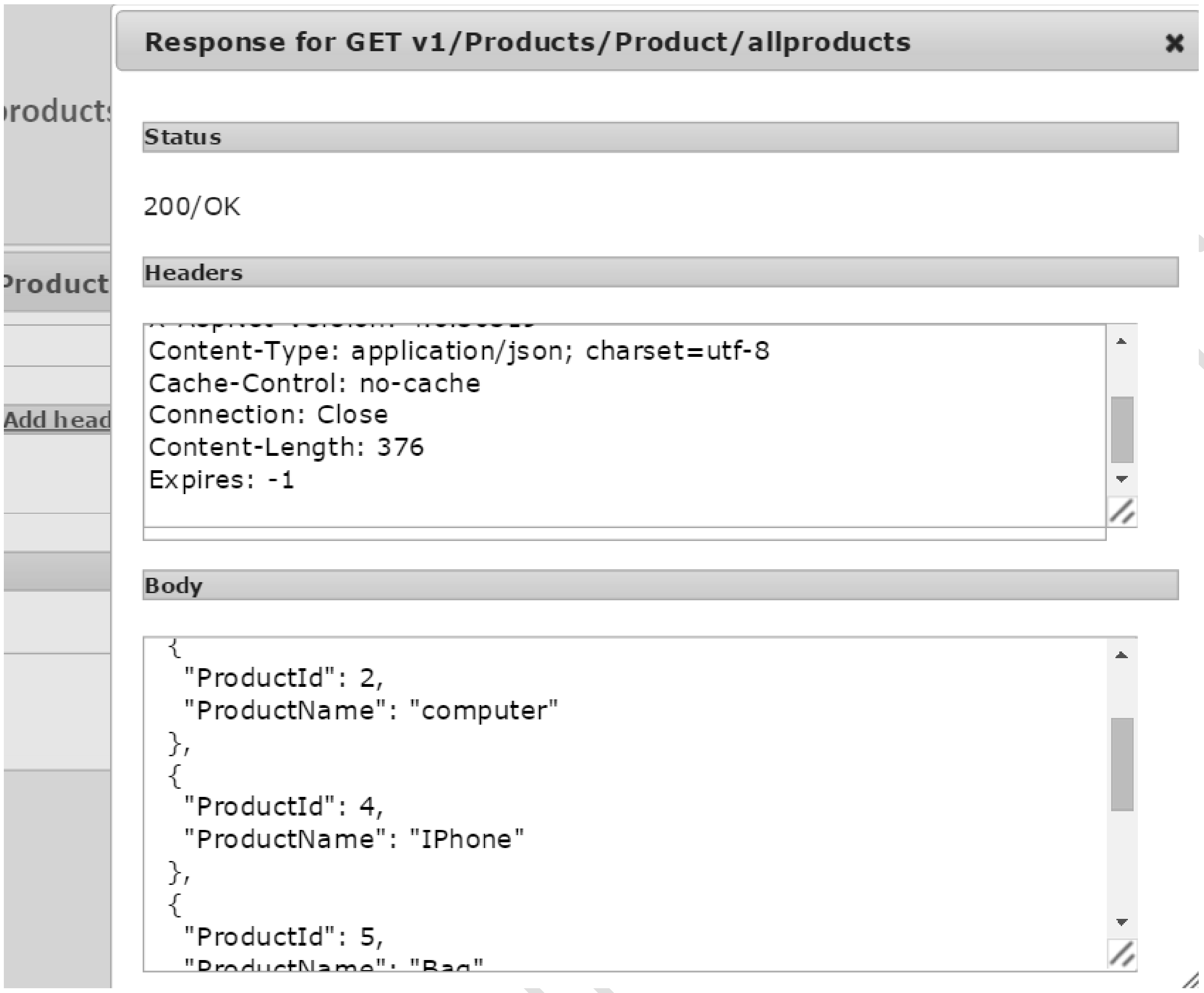

GET v1/Products/Product/allproducts

GET	v1/Products/Product/allproducts

Headers | Add header

Token	f2b39f7e-8b69-45ee-b

We get response with all the 6 products,


Response for GET v1/Products/Product/allproducts

Status

200/OK

Headers

```
Content-Type: application/json; charset=utf-8
Cache-Control: no-cache
Connection: Close
Content-Length: 376
Expires: -1
```

Body

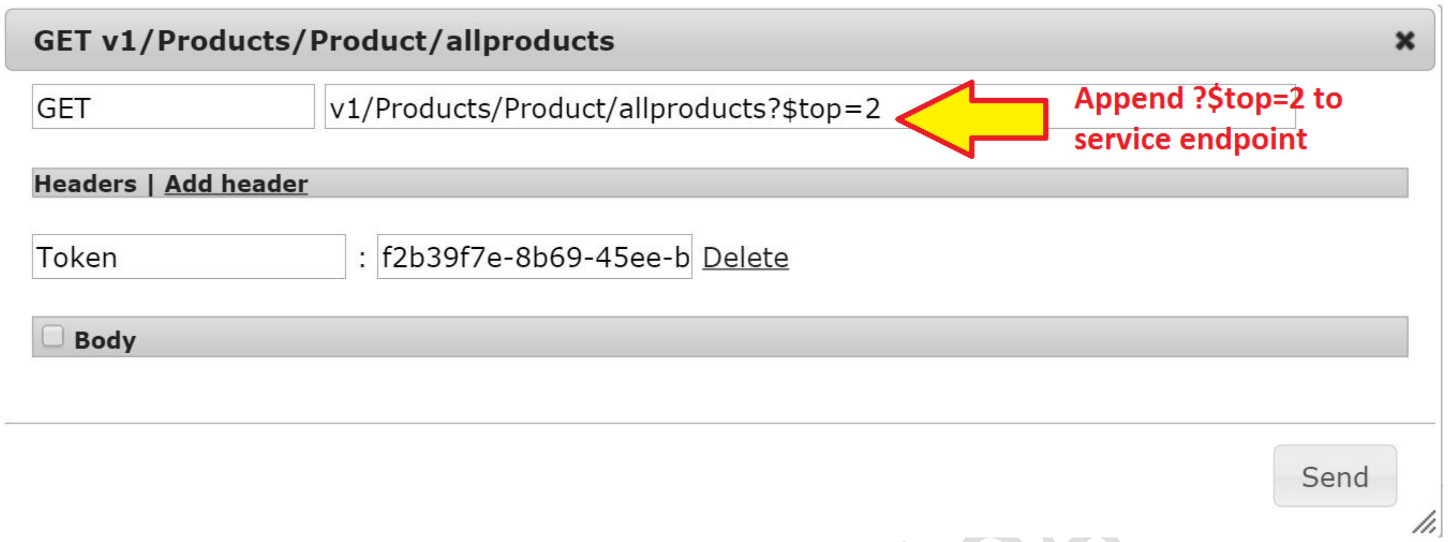
```
{
  "ProductId": 2,
  "ProductName": "computer"
},
{
  "ProductId": 4,
  "ProductName": "IPhone"
},
{
  "ProductId": 5,
  "ProductName": "Bag"
```

I'll use this controller method and make it OData endpoint and perform several query options over it. Add an attribute named [Queryable] above the method and in Request.CreateResponse mark the productEntities to productEntities.AsQueryable().

```
        [Queryable]
        [GET("allproducts")]
        [GET("all")]
        public HttpResponseMessage Get()
        {
            var products = _productServices.GetAllProducts().AsQueryable();
            var productEntities = products as List<ProductEntity> ?? products.ToList();
            if (productEntities.Any())
                return Request.CreateResponse(HttpStatusCode.OK,
productEntities.AsQueryable());
            throw new ApiDataException(1000, "Products not found", HttpStatusCode.NotFound);
        }
```

$top

Now test the API with **$top** query option,



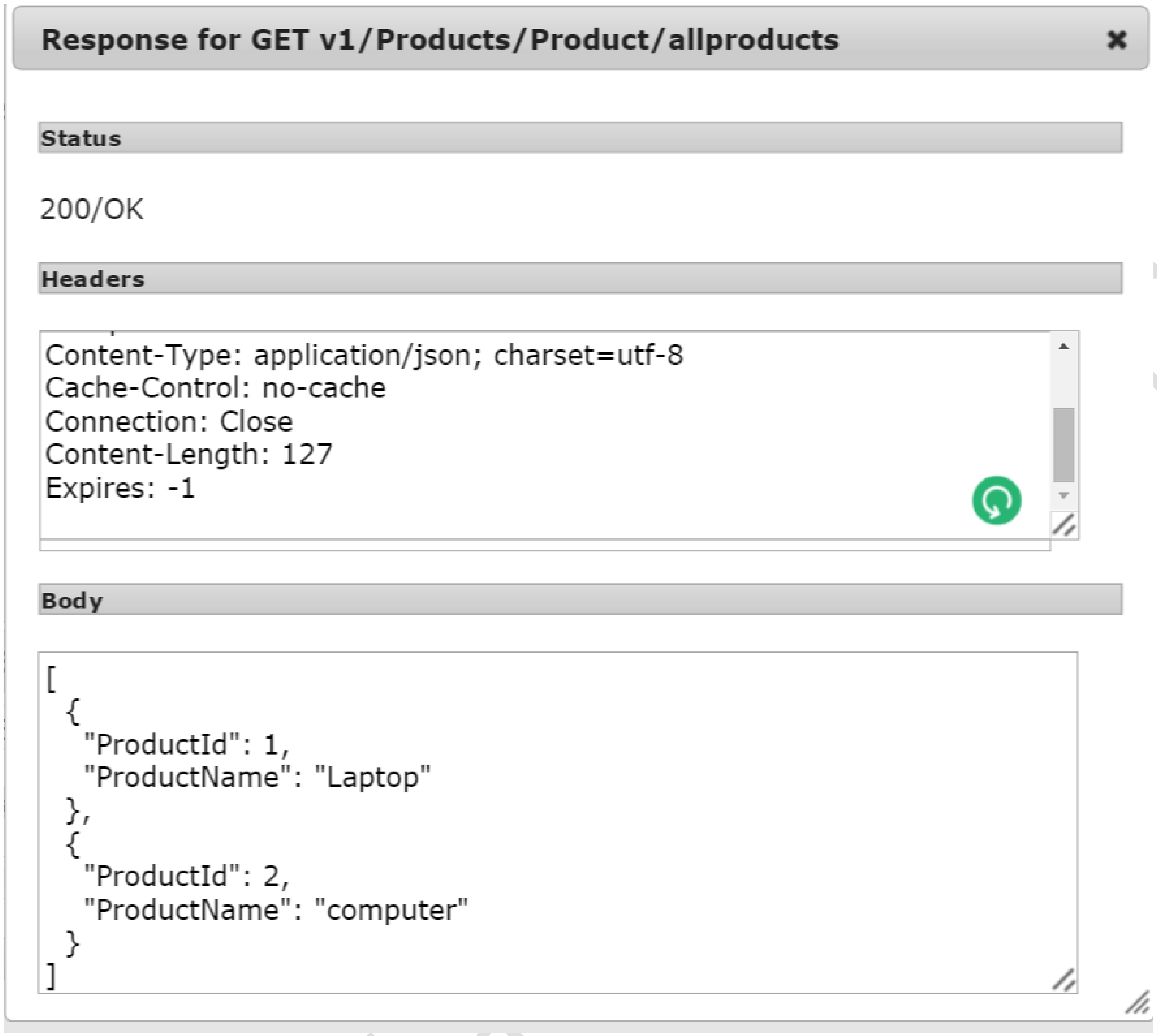
GET v1/Products/Product/allproducts

GET v1/Products/Product/allproducts?$top=2

Headers | Add header

Token : f2b39f7e-8b69-45ee-b Delete

Here in the above endpoint, I have just appended “?**$top=2”** in the endpoint of the service i.e. like we append query strings. This statement means that I want to fetch only top two products from the service and the result is,



Response for GET v1/Products/Product/allproducts

Status

200/OK

Headers

```
Content-Type: application/json; charset=utf-8
Cache-Control: no-cache
Connection: Close
Content-Length: 127
Expires: -1
```

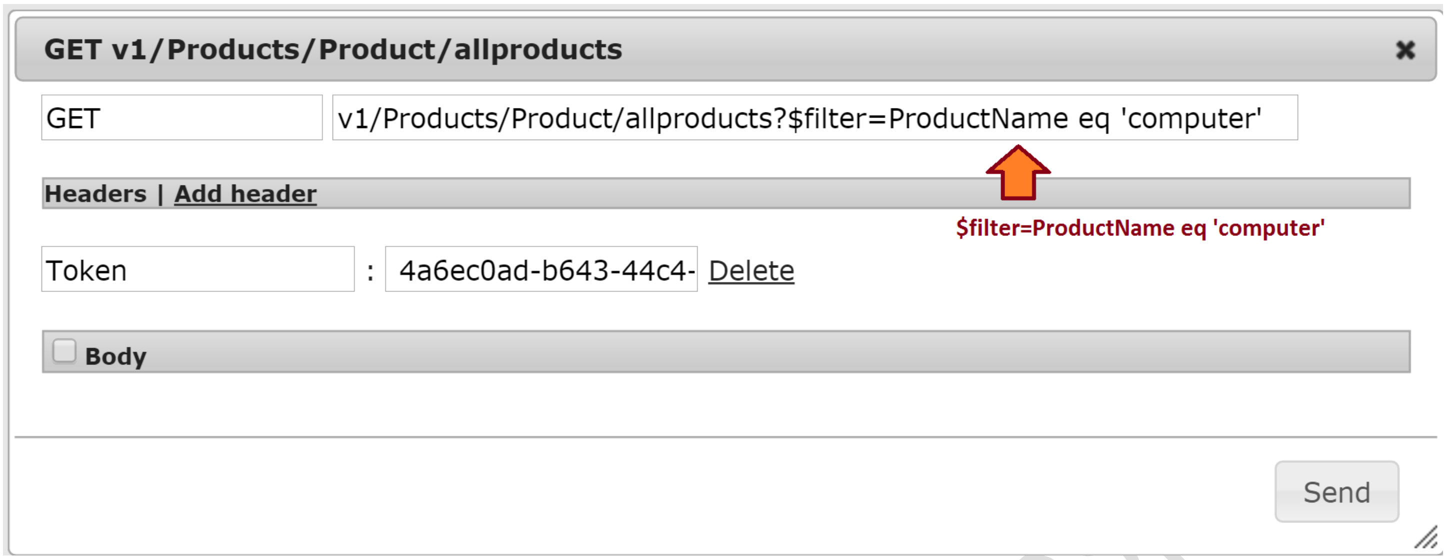
Body

```
[
 {
  "ProductId": 1,
  "ProductName": "Laptop"
 },
 {
  "ProductId": 2,
  "ProductName": "computer"
 }
]
```

We get only two products. So you can see here that it was very simple to make a service endpoint query able, and we did not have to write a new service to achieve this kind of result. Let us try few more options.

$filter

You can perform all the filtering over the records with this option. Let us try $filter query option. Suppose we need to fetch all the products whose name is "computer". You can use the same endpoint with filtering as shown below.


GET v1/Products/Product/allproducts

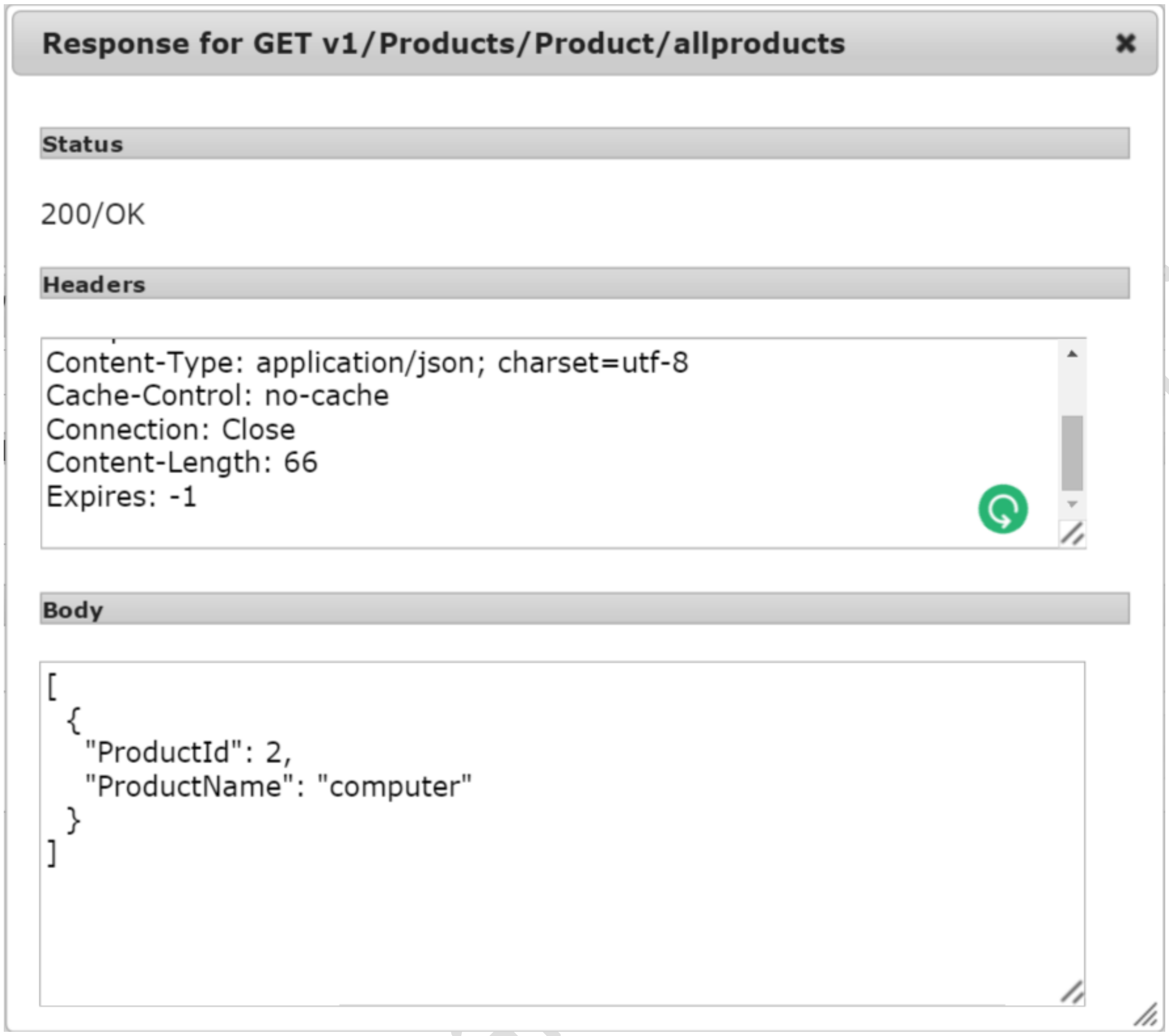
GET | v1/Products/Product/allproducts?$filter=ProductName eq 'computer'

$filter=ProductName eq 'computer'

Headers | Add header

Token : 4a6ec0ad-b643-44c4- Delete

I used **$filter=ProductName eq 'computer'** as a query string, which means fetching product having product name “computer”, as a result we get only one record from products list because there was only one record having product name as “computer”.


Response for GET v1/Products/Product/allproducts

Status

200/OK

Headers

```
Content-Type: application/json; charset=utf-8
Cache-Control: no-cache
Connection: Close
Content-Length: 66
Expires: -1
```

Body

```
[
  {
    "ProductId": 2,
    "ProductName": "computer"
  }
]
```

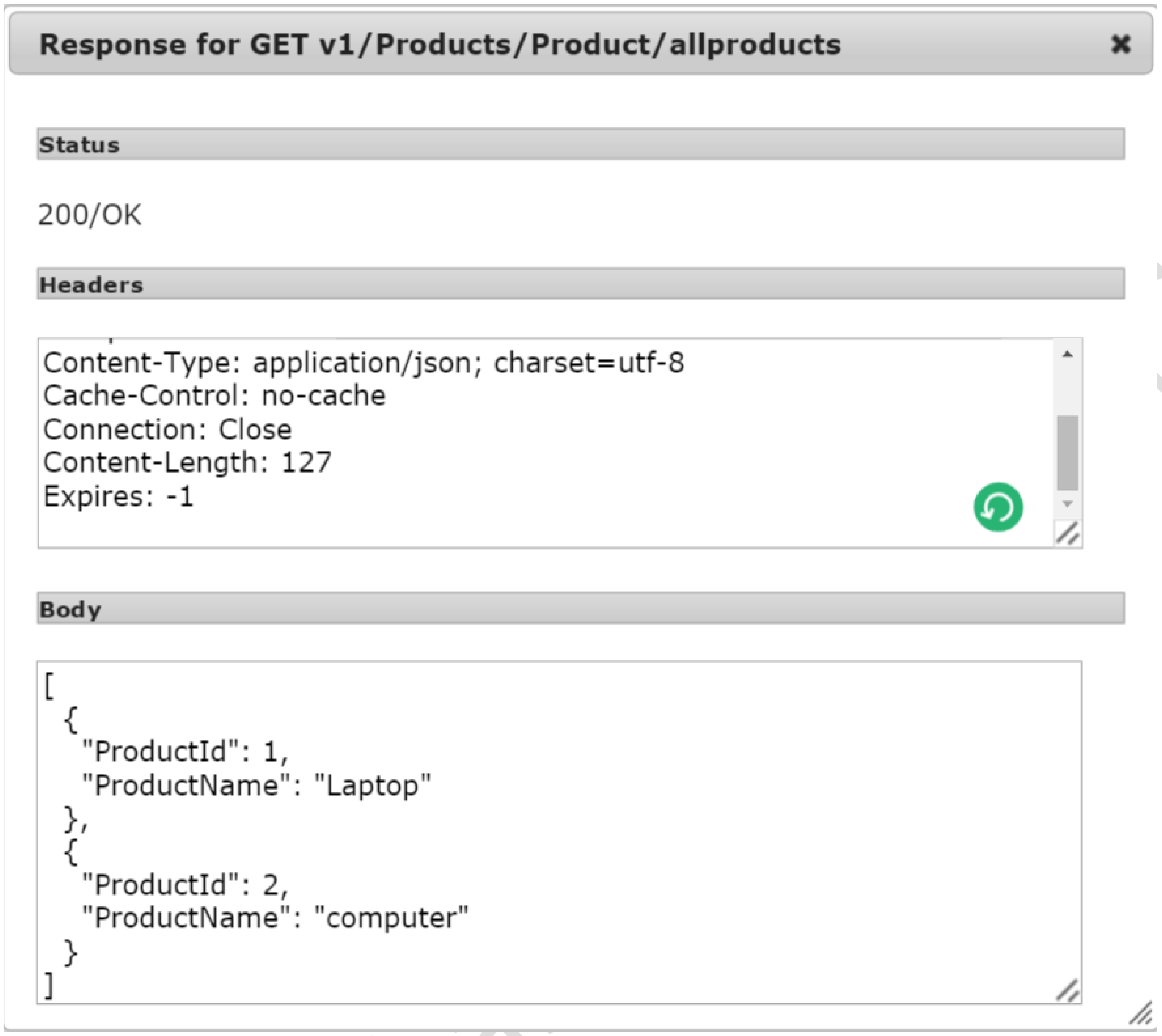
You can use filter in many different ways as shown below,

Return all products with name equal to "computer".

http://localhost:50875/v1/Products/Product/allproducts?$filter=ProductName eq "computer"

Return all products with id less than 3.

http://localhost:50875/v1/Products/Product/allproducts?$filter=ProductId lt 3


Response for GET v1/Products/Product/allproducts

Status

200/OK

Headers

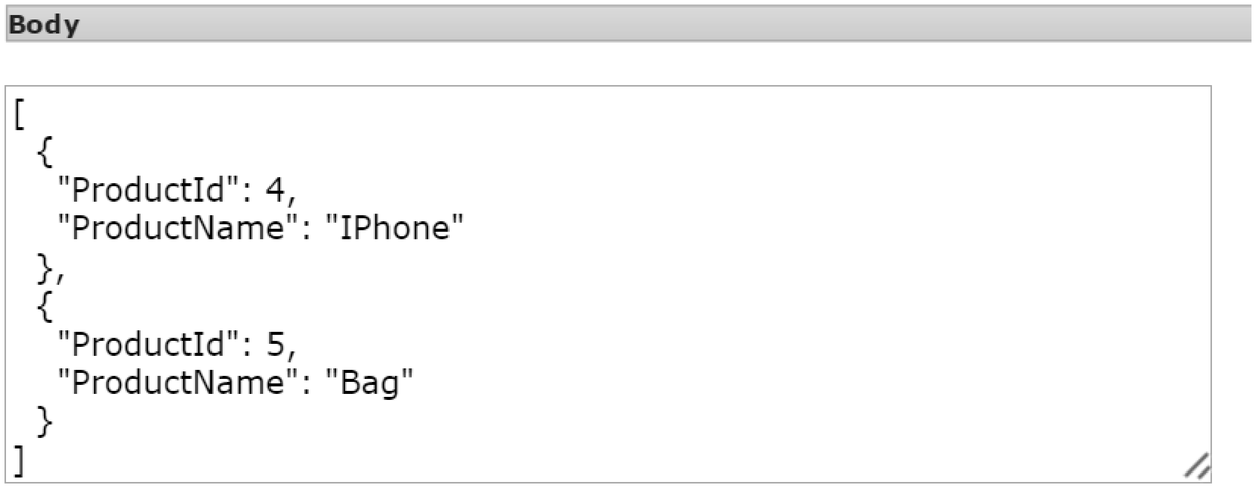
```
Content-Type: application/json; charset=utf-8
Cache-Control: no-cache
Connection: Close
Content-Length: 127
Expires: -1
```

Body

```
[
  {
    "ProductId": 1,
    "ProductName": "Laptop"
  },
  {
    "ProductId": 2,
    "ProductName": "computer"
  }
]
```

Logical operators: Return all products where id >= 3 and id <= 5.

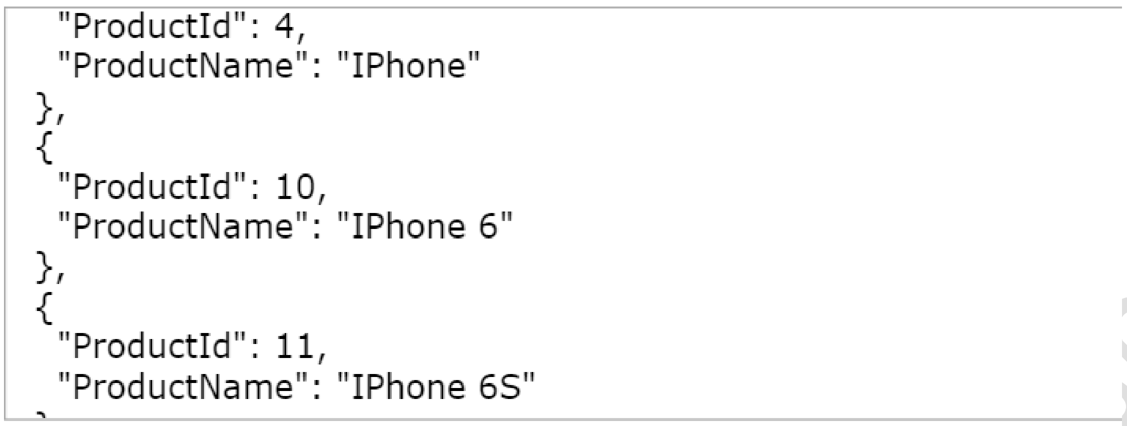
http://localhost:50875/v1/Products/Product/allproducts?$filter=ProductId ge 3 and ProductId le 5


Body

```
[
  {
    "ProductId": 4,
    "ProductName": "IPhone"
  },
  {
    "ProductId": 5,
    "ProductName": "Bag"
  }
]
```

String functions: Return all products with “IPhone” in the name.

http://localhost:50875/v1/Products/Product/allproducts?$filter=substringof('IPhone',ProductName)



```
    "ProductId": 4,
    "ProductName": "IPhone"
  },
  {
    "ProductId": 10,
    "ProductName": "IPhone 6"
  },
  {
    "ProductId": 11,
    "ProductName": "IPhone 6S"
```

Filter option could also be applied on date fields as well.

$orderby

Let us try orderby query with the same endpoint.

Return all products with sorting on product name descending

http://localhost:50875/v1/Products/Product/allproducts?$orderby=ProductName desc

Output:

```
[
  {
    "ProductId":6,
    "ProductName":"Watch"
  },
  {
    "ProductId":8,
    "ProductName":"Titan Watch"
  },
  {
    "ProductId":9,
    "ProductName":"Laptop Bag"
  },
  {
    "ProductId":1,
    "ProductName":"Laptop"
  },
  {
    "ProductId":11,
    "ProductName":"IPhone 6S"
  },
  {
    "ProductId":10,
    "ProductName":"IPhone 6"
  },
  {
    "ProductId":4,
    "ProductName":"IPhone"
  },
  {
    "ProductId":12,
    "ProductName":"HP Laptop"
  },
  {
    "ProductId":2,
    "ProductName":"computer"
  },
  {
    "ProductId":5,
    "ProductName":"Bag"
  }
]
```

Return all products with sorting on product name ascending

http://localhost:50875/v1/Products/Product/allproducts?$orderby=ProductName asc

Output:

```
[
 {
  "ProductId": 5,
  "ProductName": "Bag"
 },
 {
  "ProductId": 2,
  "ProductName": "computer"
 },
 {
  "ProductId": 12,
  "ProductName": "HP Laptop"
 },
 {
  "ProductId": 4,
  "ProductName": "IPhone"
 },
 {
    "ProductId": 10,
    "ProductName": "IPhone 6"
  },
  {
    "ProductId": 11,
    "ProductName": "IPhone 6S"
  },
  {
    "ProductId": 1,
    "ProductName": "Laptop"
  },
  {
    "ProductId": 9,
    "ProductName": "Laptop Bag"
  },
  {
    "ProductId": 8,
    "ProductName": "Titan Watch"
  },
  {
    "ProductId": 6,
    "ProductName": "Watch"
  }
]
```

Return all products with sorting on product id descending

http://localhost:50875/v1/Products/Product/allproducts?$orderby=ProductId desc

Output:

```
[
  {
    "ProductId": 12,
    "ProductName": "HP Laptop"
  },
  {
    "ProductId": 11,
    "ProductName": "IPhone 6S"
  },
  {
    "ProductId": 10,
    "ProductName": "IPhone 6"
  },
  {
    "ProductId": 9,
    "ProductName": "Laptop Bag"
```

```
 },
 {
  "ProductId": 8,
  "ProductName": "Titan Watch"
 },
 {
  "ProductId": 6,
  "ProductName": "Watch"
 },
 {
  "ProductId": 5,
  "ProductName": "Bag"
 },
 {
  "ProductId": 4,
  "ProductName": "IPhone"
 },
 {
  "ProductId": 2,
  "ProductName": "computer"
 },
 {
  "ProductId": 1,
  "ProductName": "Laptop"
 }
]
```

Return all products with sorting on product id ascending

http://localhost:50875/v1/Products/Product/allproducts?$orderby=ProductId asc

Output:

```
[
 {
  "ProductId": 1,
  "ProductName": "Laptop"
 },
 {
  "ProductId": 2,
  "ProductName": "computer"
 },
 {
  "ProductId": 4,
  "ProductName": "IPhone"
 },
 {
    "ProductId": 5,
    "ProductName": "Bag"
  },
  {
    "ProductId": 6,
    "ProductName": "Watch"
  },
  {
    "ProductId": 8,
    "ProductName": "Titan Watch"
  },
  {
    "ProductId": 9,
    "ProductName": "Laptop Bag"
  },
  {
    "ProductId": 10,
    "ProductName": "IPhone 6"
  },
  {
    "ProductId": 11,
    "ProductName": "IPhone 6S"
  },
  {
    "ProductId": 12,
    "ProductName": "HP Laptop"
  }
]
```

$orderby with $top

You can make use of multiple query options to fetch the desired records. Suppose I need to fetch only 5 records from top order by ProductId ascending. To achieve this I can write the following query.

http://localhost:50875/v1/Products/Product/allproducts?$orderby=ProductId asc&$top=5

Output:

```
[
  {
    "ProductId": 1,
    "ProductName": "Laptop"
  },
  {
    "ProductId": 2,
    "ProductName": "computer"
```

```
 },
 {
  "ProductId": 4,
  "ProductName": "IPhone"
 },
 {
  "ProductId": 5,
  "ProductName": "Bag"
 },
 {
  "ProductId": 6,
  "ProductName": "Watch"
 }
]
```

The above output fetches 5 records with sorted ProductId.

$skip

As the name suggests, the skip query option is used to skip the record. Let’s consider following scenarios.

Select top 5 and skip 3

http://localhost:50875/v1/Products/Product/allproducts?$top=5&$skip=3

Output

```
[
 {
  "ProductId": 5,
  "ProductName": "Bag"
 },
 {
  "ProductId": 6,
  "ProductName": "Watch"
 },
 {
  "ProductId": 8,
  "ProductName": "Titan Watch"
 },
 {
  "ProductId": 9,
  "ProductName": "Laptop Bag"
 },
 {
  "ProductId": 10,
    "ProductName": "IPhone 6"
  }
]
```

$skip with $orderby

__Order by ProductName ascending and skip 6__

http://localhost:50875/v1/Products/Product/allproducts?$orderby=ProductName asc &$skip=6

__Output__

```
[
  {
    "ProductId": 1,
    "ProductName": "Laptop"
  },
  {
    "ProductId": 9,
    "ProductName": "Laptop Bag"
  },
  {
    "ProductId": 8,
    "ProductName": "Titan Watch"
  },
  {
    "ProductId": 6,
    "ProductName": "Watch"
  }
]
```

Following are some standard filter operators and query functions you can use to create your query taken from https://msdn.microsoft.com/en-us/library/gg334767.aspx

Standard filter operators

The Web API supports the standard OData filter operators listed in the following table.

Operator	Description	Example
Comparison Operators		
eq	Equal	$filter=revenue eq 100000

ne	Not Equal	$filter=revenue ne 100000
gt	Greater than	$filter=revenue gt 100000
ge	Greater than or equal	$filter=revenue ge 100000
lt	Less than	$filter=revenue lt 100000
le	Less than or equal	$filter=revenue le 100000
Logical Operators		
and	Logical and	$filter=revenue lt 100000 and revenue gt 2000
or	Logical or	$filter=contains(name,'(sample)') or contains(name,'test')
not	Logical negation	$filter=not contains(name,'sample')
Grouping Operators		
( )	Precedence grouping	(contains(name,'sample') or contains(name,'test')) and revenue gt 5000

Standard query functions

The web API supports these standard OData string query functions.

Function	Example
contains	$filter=contains(name,'(sample)')
endswith	$filter=endswith(name,'Inc.')
startswith	$filter=startswith(name,'a')

Paging

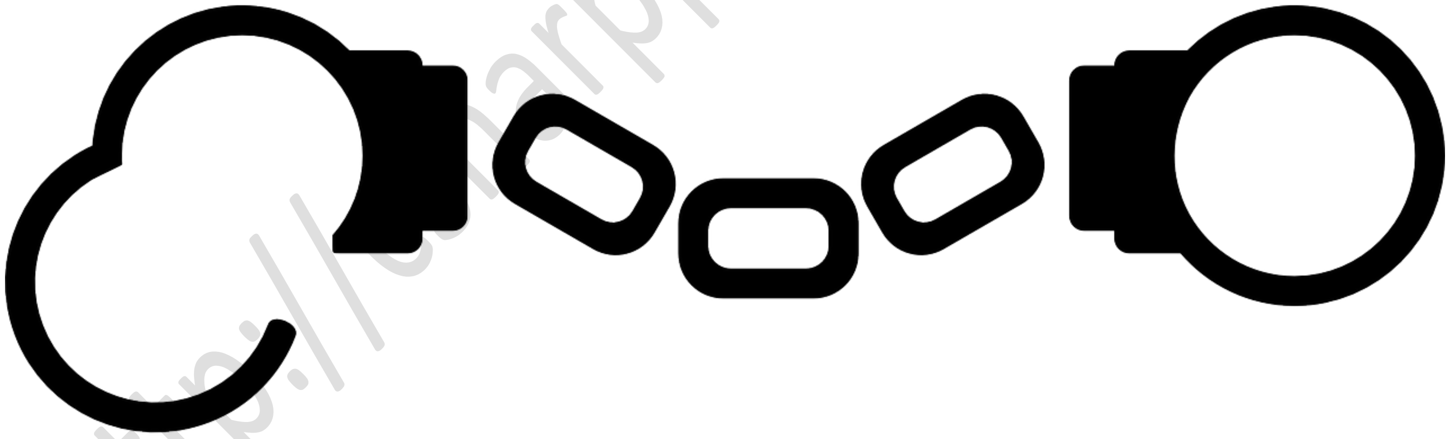
You can create paging enabled endpoint which means, if you have a lot of data on database, and the requirement is that client needs to show the data like 10 records per page. So it is advisable that server itself should send those 10 records per request, so that the entire data payload does not travel on network. This may also improve the performance of your services.

Let’s suppose you have 10000 records on database, so you can enable your endpoint to return 10 records and entertain the request for initial record and number of records to be sent. In this case client will make request every time for next set of records fetch pagination option is used or user navigates to next page. To enable paging, just mention the page count at the [Queryable] attribute. For e.g. [Queryable(PageSize = 10)]

So our method code becomes,

```
        [Queryable(PageSize = 10)]
        [GET("allproducts")]
        [GET("all")]
        public HttpResponseMessage Get()
        {
            var products = _productServices.GetAllProducts().AsQueryable();
            var productEntities = products as List<ProductEntity> ?? products.ToList();
            if (productEntities.Any())
                return Request.CreateResponse(HttpStatusCode.OK, 
productEntities.AsQueryable());
            throw new ApiDataException(1000, "Products not found", HttpStatusCode.NotFound);
        }
```

Query Options Constraints



You can put constraints over your query options too. Suppose you do not want client to access filtering options or skip options, then at the action level you can put constraints to ignore that kind of API request. Query Option constraints are of four types,

AllowedQueryOptions

Example : [Queryable(AllowedQueryOptions =AllowedQueryOptions.Filter | AllowedQueryOptions.OrderBy)]

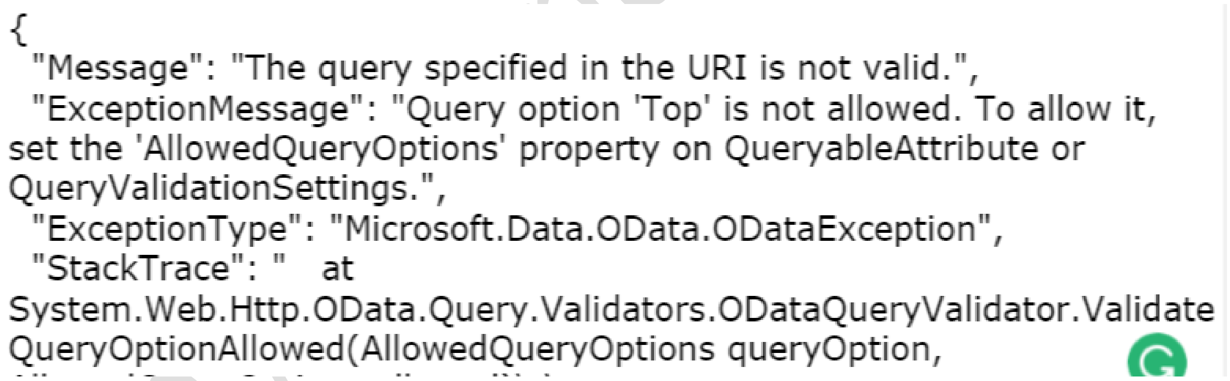
Above example of query option states that only $filter and $orderby queries are allowed on the API.

```
       [Queryable(AllowedQueryOptions =AllowedQueryOptions.Filter |
AllowedQueryOptions.OrderBy)]
        [GET("allproducts")]
        [GET("all")]
        public HttpResponseMessage Get()
        {
            var products = _productServices.GetAllProducts().AsQueryable();
            var productEntities = products as List<ProductEntity> ?? products.ToList();
            if (productEntities.Any())
                return Request.CreateResponse(HttpStatusCode.OK,
productEntities.AsQueryable());
            throw new ApiDataException(1000, "Products not found", HttpStatusCode.NotFound);
        }
```

So when I invoked the endpoint with $top query,

http://localhost:50875/v1/Products/Product/allproducts?$top=10

I got the following response,



```
{
  "Message": "The query specified in the URI is not valid.",
  "ExceptionMessage": "Query option 'Top' is not allowed. To allow it,
set the 'AllowedQueryOptions' property on QueryableAttribute or
QueryValidationSettings.",
  "ExceptionType": "Microsoft.Data.OData.ODataException",
  "StackTrace": "   at
System.Web.Http.OData.Query.Validators.ODataQueryValidator.Validate
QueryOptionAllowed(AllowedQueryOptions queryOption,
```

It says,

"Message": "The query specified in the URI is not valid.",
"ExceptionMessage": "Query option 'Top' is not allowed. To allow it, set the 'AllowedQueryOptions' property on QueryableAttribute or QueryValidationSettings."

That means it is not allowing other kind of queryoptions to work on this API endpoint.

AllowedOrderByProperties

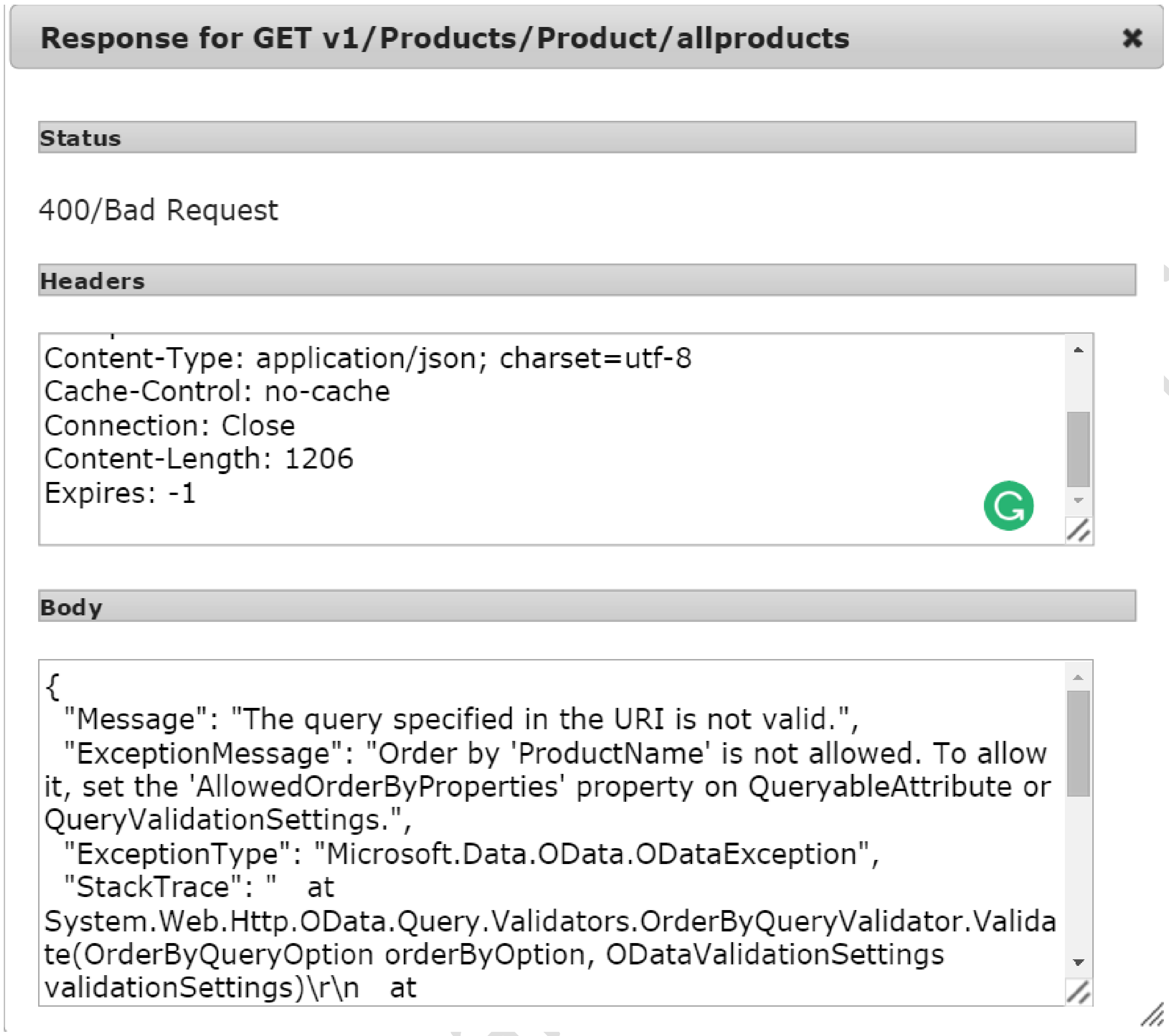
Example : `[Queryable(AllowedOrderByProperties = "ProductId")] // supply list of columns/properties`

This means that the endpoint only supports sorting on the basis of ProductId. You can specify more properties for which you want to enable sorting. So as per following code,

```
        [Queryable(AllowedOrderByProperties = "ProductId")]
        [GET("allproducts")]
        [GET("all")]
        public HttpResponseMessage Get()
        {
            var products = _productServices.GetAllProducts().AsQueryable();
            var productEntities = products as List<ProductEntity> ?? products.ToList();
            if (productEntities.Any())
                return Request.CreateResponse(HttpStatusCode.OK,
productEntities.AsQueryable());
            throw new ApiDataException(1000, "Products not found", HttpStatusCode.NotFound);
        }
```

If I try to invoke the URL : *http://localhost:50875/v1/Products/Product/allproducts?$orderby=ProductName desc*

It gives error in response,


Response for GET v1/Products/Product/allproducts

Status

400/Bad Request

Headers

```
Content-Type: application/json; charset=utf-8
Cache-Control: no-cache
Connection: Close
Content-Length: 1206
Expires: -1
```

Body

```
{
  "Message": "The query specified in the URI is not valid.",
  "ExceptionMessage": "Order by 'ProductName' is not allowed. To allow
it, set the 'AllowedOrderByProperties' property on QueryableAttribute or
QueryValidationSettings.",
  "ExceptionType": "Microsoft.Data.OData.ODataException",
  "StackTrace": "   at
System.Web.Http.OData.Query.Validators.OrderByQueryValidator.Valida
te(OrderByQueryOption orderByOption, ODataValidationSettings
validationSettings)\r\n   at
```

Says,

"Message": "The query specified in the URI is not valid.",
"ExceptionMessage": "Order by 'ProductName' is not allowed. To allow it, set the 'AllowedOrderByProperties' property on QueryableAttribute or QueryValidationSettings."

The URL: *http://localhost:50875/v1/Products/Product/allproducts?$orderby=ProductId desc* will work fine.

AllowedLogicalOperators

Example : `[Queryable(AllowedLogicalOperators = AllowedLogicalOperators.GreaterThan)]`

In the above mentioned example, the statement states that only greaterThan i.e. “gt” logical operator is allowed in the query and query options with any other logical operator other that “gt” will return error. You can try it in your application.

AllowedArithmeticOperators

Example : `[Queryable(AllowedArithmeticOperators = AllowedArithmeticOperators.Add)]`

In the above mentioned example, the statement states that only Add arithmetic operator is allowed while API call. You can try it in your application.

Conclusion

There are lot more things in OData that I cannot cover in one go. The purpose was to give an idea of what we can achieve using OData. You can explore more options and attributes and play around with REST API's. I hope by you'll be able to create a basic WebAPI application with all the required functionalities. The code base attached with all the articles in the series serves as a boilerplate for creating any Enterprise level WebAPI application. Keep exploring REST. Happy coding☺. Download complete source code from **GitHub.**

References

http://www.asp.net/web-api/overview/odata-support-in-aspnet-web-api/supporting-odata-query-options
https://msdn.microsoft.com/en-us/library/azure/gg312156.aspx

Other Series

My other series of articles:

- Introduction to MVC Architecture and Separation of Concerns: Part 1
- Diving Into OOP (Day 1): Polymorphism and Inheritance (Early Binding/Compile Time Polymorphism)

For more informative articles visit my Blog.