**Understanding 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.**

*“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).”*

*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.

*“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?$orderby=Name*](http://localhost/Products?$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.*

**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.

**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’) |

-------------------------------------------------------------------------------------------------------------------------------------

## 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.

### 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.

-------------------------------------------------------------------------------------------------------------------------------------

Let's think that we have written one function to return the employee name when someone will supplies the employee code. Working cool, but in the next data some other guy is asking to get an employee's address and he wants to supply the employee's name.

That is a new requirement. Hmm, I need to write another function. History repeats itself, in another fine morning another guy demands that he wants to get an employee phone number only when he supplies the employee code. Oh, again add another function to the existing service. Now, we know people are greedy and demanding, the new requirements will never stop. Then what is the solution? Shall we add a new function each time a new requirement arrives?

Don't worry, Odata is the solution. Let's understand what Odata is theoretically .

The Open Data (OData) Protocol is a data access protocol for the web. OData provides a uniform way to structure data, query the data, and manipulate the data using CRUD operations (Create, Read, Update and Delete). OData supports both AtomPub (XML) and JSON formats. OData also defines a way to expose metadata about the data. Clients can use the metadata to discover the type information and relationships for the data set.

So, as per our discussion, we know that SOA is Service Oriented Architecture that exposes a few functions to outsource data from the endpoint, this technique is pretty cool when the requirement is fridge or when we know the exact query pattern.

But in this scenario, when we don't know the user's requirements, we can expose our service by maintaining the Odata protocol. Then the client can fire their own query to get data.

**Gives the idea of Resource Oriented Architecture (ROA)**

**The Odata protocol gives the idea of a Resource Oriented Architecture (ROA) where we can expose a resource and enable the user to run various ad-hoc quiries against the resources. This is similar to the approach in the database system where you have tables that represent data and the ability to create various queries. The only difference is that, in ROA we form and pass queries using an URL.**

**Difference between Odata and REST**

This is another misconception in developer's minds. Let's understand the difference clearly. REST is an architectural style that defines certain principals to be followed for a service based application. For example the REST principal says that the service URL should be like a local directory URL. There will be no query string or data passing mechanism at all. (Oh, we wil pass data in the form of an URL part but not by query string) in the URL. The client and server will be totally de-coupled and much more.

So, if our service or application follows those principals then we will say that, ok the service is a RESTFul service. But it's not necessary to be RESTful of all services, the service may be or may not be.

Understood? Cool, now as per our discussion, Odata is a protocol that will sit on top of a service. If the service maintains those protocols then the user will get certain benefits, like the user (read client application) is able to read data as per its need, not depending on the service's return value.

* **http://host/service/Products?$filter=Name eq ‘Milk’**: Returns all products where the name is equal to ‘Milk’
* **http://host/service/Products?$filter=Name eq ‘Milk’ and Price lt 2.55**: Returns all products where the name is equal to ‘Milk’ and the price is less than 2.55
* **http://host/service/Products?$top=10&$skip10**: Returns items 11>21. The $top system query option requests the number of items in the queried collection to be included in the result. The $skip query option requests the number of items in the queried collection that are to be skipped and not included in the result
* **http://host/service/Products?$orderby=Name desc,Id**: Returns all products ordered by the Name in descending order, then ordered by Id
* **http://host/service/Products?$filter=Enabled eq true**: Returns all products where the Enabled field (which is a boolean) is true
* **http://host/service/Products?$filter=substringof(‘Martin’, Name)**: Returns all products the Name field contains the text Martin

### How to use OData query syntax in ASP.net Web API?

This is where things get really awesome – it is super, super easy to change a regular Web API controller to support OData query syntax – once you know how to do this, you’ll never not do it!

If we assume we are starting from a regular Web API project (File > New Project > ASP.net Web Application > Web API), you first need to add a scaffolded controller which you can do by following these steps:

1. Add a model class (I’m using the typical ‘Person’ class with Id, First Name, Last Name, Age etc)
2. Right-click the ‘controllers’ folder > Add > New Scaffolded item > Web API 2 Controller with actions, using Entity Framework > Model = Person

NOTE: You do not need to choose the ‘Web API 2 OData Controller…’ option, you can add the query syntax to any controller

Once you’ve setup your controller, you should end up with a simple controller which has a default GET action that looks a little like this:

// GET: api/People  
public IQueryable<Person> GetPeople()  
{  
return db.People;  
}

This action will simply return all the rows in the ‘People’ table of the database with no ability to filter, sort etc. To add full OData query support, you need to make a few changes:

1. Add the Microsoft.Aspnet.OData package via nugget … simply pop this into you ‘Package Manager Console': Install-Package Microsoft.AspNet.Odata
2. Add this using statement: using System.Web.OData;
3. Add the[EnableQueryAttribute]attribute to your GetPeople action.
4. Add AsQueryable(); to your return line so it looks like this: return db.People.AsQueryable();

Your finished code will look something like this:

// GET: api/People  
[EnableQueryAttribute]  
public IQueryable<Person> GetPeople()  
{  
return db.People.AsQueryable();  
}

References:

https://codeteddy.com/2016/04/05/restful-day-9-odata-in-asp-net-web-apis/

https://blogs.msdn.microsoft.com/martinkearn/2015/03/10/using-odata-query-syntax-with-web-api/

**----------------------------------------------------------------------------**

**OData Sample 1**

Install-Package Microsoft.AspNet.WebApi

**OData Demo**

Step 1.

PM> Install-Package Microsoft.AspNet.Odata

**Configure the OData Endpoint**

Open the file App\_Start/WebApiConfig.cs. Add the following **using** statements:

using ProductService.Models;

using System.Web.OData.Builder;

using System.Web.OData.Extensions;

Then add the following code to the **Register** method:

public static class WebApiConfig

{

public static void Register(HttpConfiguration config)

{

// New code:

ODataModelBuilder builder = new ODataConventionModelBuilder();

builder.EntitySet<Product>("Products");

config.MapODataServiceRoute(

routeName: "ODataRoute",

routePrefix: null,

model: builder.GetEdmModel());

}

}

This code does two things:

* Creates an Entity Data Model (EDM).
* Adds a route.

An EDM is an abstract model of the data. The EDM is used to create the service metadata document. The **ODataConventionModelBuilder**class creates an EDM by using default naming conventions. This approach requires the least code. If you want more control over the EDM, you can use the **ODataModelBuilder** class to create the EDM by adding properties, keys, and navigation properties explicitly.

A *route* tells Web API how to route HTTP requests to the endpoint. To create an OData v4 route, call the **MapODataServiceRoute** extension method.

If your application has multiple OData endpoints, create a separate route for each. Give each route a unique route name and prefix.

## Querying the Entity Set

Add the following methods to ProductsController.

[EnableQuery]

public IQueryable<Product> Get()

{

return db.Products;

}

[EnableQuery]

public SingleResult<Product> Get([FromODataUri] int key)

{

IQueryable<Product> result = db.Products.Where(p => p.Id == key);

return SingleResult.Create(result);

}

The parameterless version of the Get method returns the entire Products collection. The Get method with a key parameter looks up a product by its key (in this case, the Id property).

The **[EnableQuery]** attribute enables clients to modify the query, by using query options such as $filter, $sort, and $page. For more information, see [Supporting OData Query Options](http://www.asp.net/web-api/overview/odata-support-in-aspnet-web-api/supporting-odata-query-options).

**Updating an Entity**

OData supports two different semantics for updating an entity, PATCH and PUT.

* **PATCH** performs a partial update. The client specifies just the properties to update.
* **PUT** replaces the entire entity.

The disadvantage of **PUT** is that the client must send values for all of the properties in the entity, including values that are not changing. The[OData spec](http://docs.oasis-open.org/odata/odata/v4.0/os/part1-protocol/odata-v4.0-os-part1-protocol.html#_Toc372793719) states that PATCH is preferred.

In any case, here is the code for both PATCH and PUT methods:

References:

http://www.asp.net/web-api/overview/odata-support-in-aspnet-web-api/odata-v4/create-an-odata-v4-endpoint