OData Query Options in ASP.NET Web API

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.

> Before enabling query options, please read the topic OData Security Guidance.

Enabling OData Query Options

Web API supports the following OData query options:

Option	Description
$expand	Expands related entities inline.
$filter	Filters the results, based on a Boolean condition.
$inlinecount	Tells the server to include the total count of matching entities in the response. (Useful for server-side paging.)
$orderby	Sorts the results.
$select	Selects which properties to include in the response.
$skip	Skips the first n results.
$top	Returns only the first n the results.

To use OData query options, you must enable them explicitly. You can enable them globally for the entire application, or enable them for specific controllers or specific actions.

To enable OData query options globally, call **EnableQuerySupport** on the **HttpConfiguration** class at startup:

```
public static void Register(HttpConfiguration config)
{
    // ...

    config.EnableQuerySupport();

    // ...
}
```

The **EnableQuerySupport** method enables query options globally for any controller action that returns an **IQueryable** type. If you don't want query options enabled for the entire application, you can enable them for specific controller actions by adding the **[Queryable]** attribute to the action method.

```
public class ProductsController : ApiController
{
    [Queryable]
    IQueryable<Product> Get() {}
}
```

Example Queries

This section shows the types of queries that are possible using the OData query options. For specific details about the query options, refer to the OData documentation at www.odata.org.

For information about $expand and $select, see Using $select, $expand, and $value in ASP.NET Web API OData.

Client-Driven Paging

For large entity sets, the client might want to limit the number of results. For example, a client might show 10 entries at a time, with "next" links to get the next page of results. To do this, the client uses the $top and $skip options.

```
http://localhost/Products?$top=10&$skip=20
```

The $top option gives the maximum number of entries to return, and the $skip option gives the number of entries to skip. The previous example fetches entries 21 through 30.

Filtering

The $filter option lets a client filter the results by applying a Boolean expression. The filter expressions are quite powerful; they include logical and arithmetic operators, string functions, and date functions.

Return all products with category equal to "Toys".	http://localhost/Products?$filter=Category eq 'Toys'
Return all products with price less than 10.	http://localhost/Products?$filter=Price lt 10
Logical operators: Return all products where price >= 5 and price <= 15.	http://localhost/Products?$filter=Price ge 5 and Price le 15
String functions: Return all products with "zz" in the name.	http://localhost/Products?$filter=substringof('zz',Name)
Date functions: Return all products with ReleaseDate after 2005.	http://localhost/Products?$filter=year(ReleaseDate) gt 2005

Sorting

To sort the results, use the $orderby filter.

Sort by price.	http://localhost/Products?$orderby=Price
Sort by price in descending order (highest to lowest).	http://localhost/Products?$orderby=Price desc
Sort by category, then sort by price in descending order within categories.	http://localhost/odata/Products?$orderby=Category,Price desc

Server-Driven Paging

If your database contains millions of records, you don't want to send them all in one payload. To prevent this, the server can limit the number of entries that it sends in a single response. To enable server paging, set the **PageSize** property in the **Queryable** attribute. The value is the maximum number of entries to return.

```
[Queryable(PageSize=10)]
public IQueryable<Product> Get()
{
    return products.AsQueryable();
}
```

If your controller returns OData format, the response body will contain a link to the next page of data:

```
{
  "odata.metadata":"http://localhost/$metadata#Products",
  "value":[
    { "ID":1,"Name":"Hat","Price":"15","Category":"Apparel" },
    { "ID":2,"Name":"Socks","Price":"5","Category":"Apparel" },
    // Others not shown
  ],
  "odata.nextLink":"http://localhost/Products?$skip=10"
}
```

The client can use this link to fetch the next page. To learn the total number of entries in the result set, the client can set the $inlinecount query option with the value "allpages".

`http://localhost/Products?$inlinecount=allpages`

The value "allpages" tells the server to include the total count in the response:

```
{
  "odata.metadata":"http://localhost/$metadata#Products",
  "odata.count":"50",
  "value":[
    { "ID":1,"Name":"Hat","Price":"15","Category":"Apparel" },
    { "ID":2,"Name":"Socks","Price":"5","Category":"Apparel" },
    // Others not shown
  ]
}
```

Next-page links and inline count both require OData format. The reason is that OData defines special fields in the response body to hold the link and count.

For non-OData formats, it is still possible to support next-page links and inline count, by wrapping the query results in a **PageResult<T>**object. However, it requires a bit more code. Here is an example:

```
public PageResult<Product> Get(ODataQueryOptions<Product> options)
{
    ODataQuerySettings settings = new ODataQuerySettings()
    {
        PageSize = 5
    };

    IQueryable results = options.ApplyTo(_products.AsQueryable(), settings);
    return new PageResult<Product>(
        results as IEnumerable<Product>,
        Request.GetNextPageLink(),
        Request.GetInlineCount());
}
```

Here is an example JSON response:

```
{
  "Items": [
    { "ID":1,"Name":"Hat","Price":"15","Category":"Apparel" },
    { "ID":2,"Name":"Socks","Price":"5","Category":"Apparel" },

    // Others not shown

  ],
  "NextPageLink": "http://localhost/api/values?$inlinecount=allpages&$skip=10",
  "Count": 50
}
```

Limiting the Query Options

The query options give the client a lot of control over the query that is run on the server. In some cases, you might want to limit the available options for security or performance reasons. The **[Queryable]** attribute has some built in properties for this. Here are some examples.

Allow only $skip and $top, to support paging and nothing else:

```
[Queryable(AllowedQueryOptions=
    AllowedQueryOptions.Skip | AllowedQueryOptions.Top)]
```

Allow ordering only by certain properties, to prevent sorting on properties that are not indexed in the database:

```
[Queryable(AllowedOrderByProperties="Id")] // comma-separated list of properties
```

Allow the “eq” logical function but no other logical functions:

```
[Queryable(AllowedLogicalOperators=AllowedLogicalOperators.Equal)]
```

Do not allow any arithmetic operators:

```
[Queryable(AllowedArithmeticOperators=AllowedArithmeticOperators.None)]
```

You can restrict options globally by constructing a **QueryableAttribute** instance and passing it to the **EnableQuerySupport** function:

```
var queryAttribute = new QueryableAttribute()
{
    AllowedQueryOptions = AllowedQueryOptions.Top | AllowedQueryOptions.Skip,
    MaxTop = 100
};

config.EnableQuerySupport(queryAttribute);
```

Invoking Query Options Directly

Instead of using the **[Queryable]** attribute, you can invoke the query options directly in your controller. To do so, add an **ODataQueryOptions**parameter to the controller method. In this case, you don't need the **[Queryable]** attribute.

```
public IQueryable<Product> Get(ODataQueryOptions opts)
{
    var settings = new ODataValidationSettings()
    {
        // Initialize settings as needed.
        AllowedFunctions = AllowedFunctions.AllMathFunctions
    };

    opts.Validate(settings);

    IQueryable results = opts.ApplyTo(products.AsQueryable());
    return results as IQueryable<Product>;
}
```

Web API populates the **ODataQueryOptions** from the URI query string. To apply the query, pass an **IQueryable** to the **ApplyTo** method. The method returns another **IQueryable**.

For advanced scenarios, if you do not have an **IQueryable** query provider, you can examine the **ODataQueryOptions** and translate the query options into another form. (For example, see RaghuRam Nadiminti's blog post Translating OData queries to HQL, which also includes asample.)

Query Validation

The **[Queryable]** attribute validates the query before executing it. The validation step is performed in the **QueryableAttribute.ValidateQuery**method. You can also customize the validation process.

Also see OData Security Guidance.

First, override one of the validator classes that is defined in the **Web.Http.OData.Query.Validators** namespace. For example, the following validator class disables the 'desc' option for the $orderby option.

```
public class MyOrderByValidator : OrderByQueryValidator
{
    // Disallow the 'desc' parameter for $orderby option.
    public override void Validate(OrderByQueryOption orderByOption,
                                    ODataValidationSettings validationSettings)
    {
        if (orderByOption.OrderByNodes.Any(
                node => node.Direction == OrderByDirection.Descending))
        {
            throw new ODataException("The 'desc' option is not supported.");
        }
        base.Validate(orderByOption, validationSettings);
    }
}
```

Subclass the **[Queryable]** attribute to override the **ValidateQuery** method.

```
public class MyQueryableAttribute : QueryableAttribute
{
    public override void ValidateQuery(HttpRequestMessage request,
        ODataQueryOptions queryOptions)
    {
        if (queryOptions.OrderBy != null)
        {
            queryOptions.OrderBy.Validator = new MyOrderByValidator();
        }
        base.ValidateQuery(request, queryOptions);
    }
}
```

Then set your custom attribute either globally or per-controller:

```
// Globally:
config.EnableQuerySupport(new MyQueryableAttribute());

// Per controller:
public class ValuesController : ApiController
{
    [MyQueryable]
    public IQueryable<Product> Get()
    {
        return products.AsQueryable();
    }
}
```

If you are using **ODataQueryOptions** directly, set the validator on the options:

```
public IQueryable<Product> Get(ODataQueryOptions opts)
{
    if (opts.OrderBy != null)
    {
        opts.OrderBy.Validator = new MyOrderByValidator();
    }

    var settings = new ODataValidationSettings()
    {
        // Initialize settings as needed.
        AllowedFunctions = AllowedFunctions.AllMathFunctions
    };

    // Validate
    opts.Validate(settings);

    IQueryable results = opts.ApplyTo(products.AsQueryable());
    return results as IQueryable<Product>;
}
```