Nouvelle page

OData as an API technology comes in with so many options that gives API consumers the power to shape, filter, order and navigate through the data with very few lines of code.

In my previous articles I talked in details about how to enable OData on your existing ASP.NET Core API using the EDM model, in addition to that I have provided a code example for you to be able to test the capabilities of OData on your own machine.

In this article, I suggest you keep the very same source code handy to learn more about one of the most powerful query options of OData, which is $select

Introduction to OData $Select

OData $select enables API consumers to shape the data they are consuming before the data is returned from the API endpoint they are calling.

For instance, let’s assume you have an API that returns information about all students in Washington Schools.

The Student model is defined as follows:

```
public class Student
{
  public Guid Id { get; set; }
  public string Name { get; set; }
  public int Score { get; set; }
  public byte[] Diploma { get; set; }
}
```

In the student model above, a service that requires only students’ Names and Ids doesn’t need to pay the network latency tax for transferring the rest of the data an student object may contain.

For instance, the *Diploma* is of type byte[], which can contain up to 2 GB of data or 1 billion characters, in a network transportation process this could add up and cause a less than optimal service to service communication.

With OData select, you can only define the object members that you need for your process, this can be simply achieved by making the following API call:

```
https://localhost:44374/api/students?$select=Id, Name
```

The expected return value of a call like this would be as follows:

```
{
  "@odata.context": "https://localhost:44374/api/$metadata#Students(Id,Name)",
  "value": [
    {
      "Id": "1fa0248e-befa-4999-9c74-9c23dd747c63",
      "Name": "Ken Swan"
    },
    {
      "Id": "1833bb68-00f4-4133-913d-2394e90798ea",
      "Name": "Kailu Hu"
    },
```

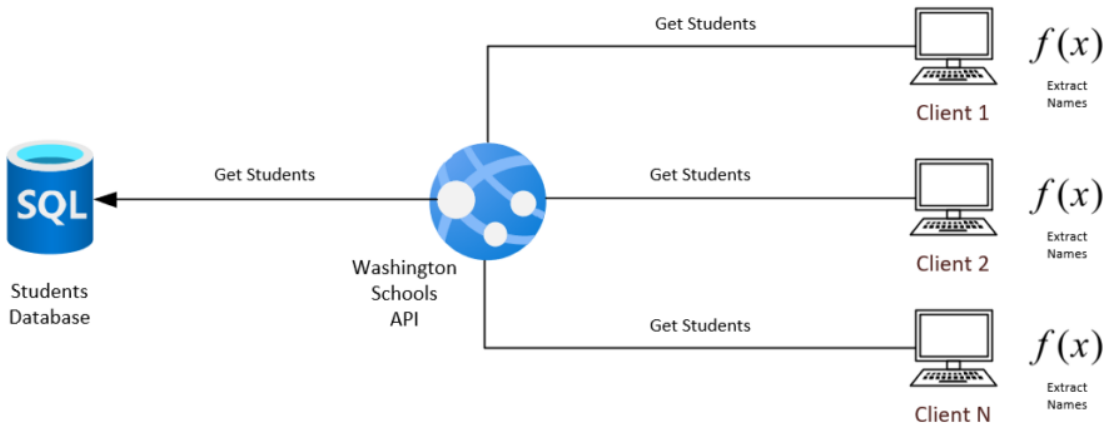
```
    {
      "Id": "02acb647-b8cd-477a-a54f-ebad5a9ccdf6",
      "Name": "Jackie Lee"
    },
    {
      "Id": "5cda4467-72ae-4c86-919e-56fa902d9095",
      "Name": "Vishu Goli"
    }
  ]
}
```

Performance Advantages of Using OData

The advantage of leveraging OData Select on the server-side rather than doing the processing on the client-side is that you don't have to worry about whether your client is going to be able to or have the required memory to process the data, instead it puts more control on the server-side where optimizations can be controlled and the hardware can be scaled for heavy lifting, which inevitably provide a seamless consistent user experience on the client-side from an API consumption perspective.

Non-Optimal Architecture

To give more visualization of this advantage consider the following architecture:

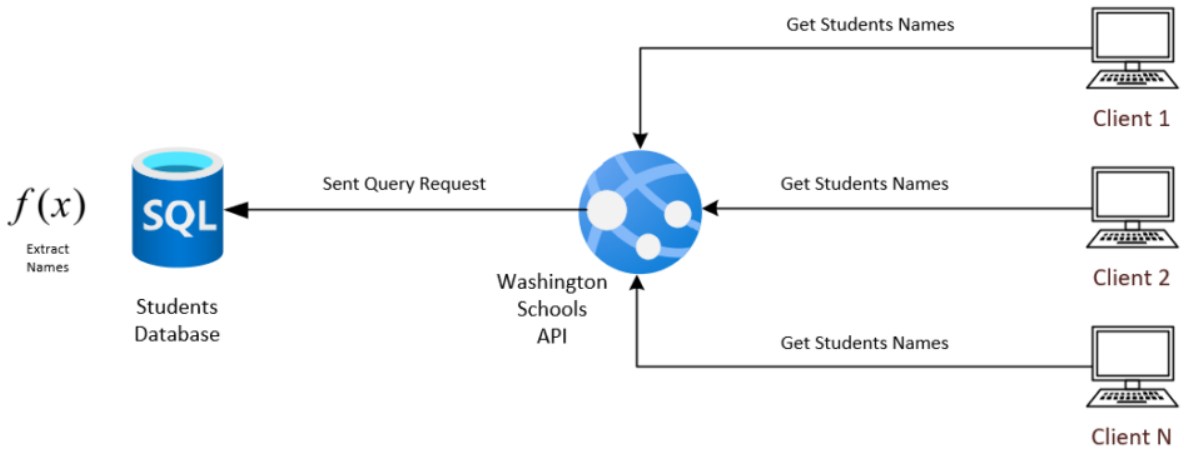


The problem with the architecture above, is that *f(x)* is being repeatedly applied for N number of clients, while repeatedly transferring unnecessarily large amounts of data from the server that eventually gets discarded on the client side.

In other words, the above architecture includes the cost of network traffic of large data sets in addition to the processing time.

Optimal Architecture

But with OData $select, the architecture would look like this:



In that architecture we have exponentially decreased the cost of mapping the data from N to 1, we have also decreased the network traffic costs by minimizing the data transferred to only the required information which should eventually produce a better execution time holistically across the entire system, which will also guarantee consistent UI performance.

Disabling OData $Select

There comes a time when you need to disable certain functions like select for some given business need, mainly to enforce the entire model to be returned to your API consumers.

Disabling the Select functionality could be implemented in three different ways:

Endpoint Level

On a particular endpoint level, you can leverage the options the *EnableQuery()* annotation offers you to select which functionality you would like to allow or disallow, in this case Select will be disabled by applying the following parameter:

1

```
        [HttpGet]
```

```
2
        [EnableQuery(AllowedQueryOptions = Microsoft.AspNet.OData.Query.AllowedQueryOptions.None
3
        public ActionResult<IQueryable<Student>> Get()
4
        {
5
            ...
6
        }
```

You can use the same parameter options to allow or disallow any number of query options you would like, which we will extensively talk about in the next articles.

Controller Level

The other option is to completely disallow Select on the controller level, but moving the EnableQuery() annotation with the same restrictions as we mentioned above to the controller level as follows:

```
1
    [EnableQuery(AllowedQueryOptions = Microsoft.AspNet.OData.Query.AllowedQueryOptions.None)]
2
    public class StudentsController : ControllerBase
3
    {
4
        ...
5
    }
```

Application Level

You can also disallow using Select query option across your entire application by removing the Select query parameter from your route builder setup as follows:

```
            app.UseMvc(routeBuilder =>
            {
                routeBuilder.Select();
                routeBuilder.MapODataServiceRoute("api", "api", GetEdmModel());
            });
```

Final Notes

1. There are so many great advantages in allowing your client to pick and choose which pieces of information they need from your API to fit their business.
2. OData $select minimizes the effort on the client side when it comes to data mapping and processing, it handles all of that to ensure consistent user experience.
3. The current implementation of OData can only support up to ASP.NET Core 2.2 and the team is working diligently to release OData for ASP.NET Core 3.0 by the 2nd Quarter of 2020.

Revision #3

Created Wed, Apr 21, 2021 5:39 AM by Nelson

Updated Wed, Apr 21, 2021 5:43 AM by Nelson