# Web API OData on ASP.NET Core Design Spec (v0.1)

# Overview

ASP.NET Core v1.0 has many breaking changes and redesign compared to the stable ASP.NET v4.x versions. Web API OData built on top of ASP.NET Core must be aligned with these changes. We originally have two options: straightforward port and redesign. But after investigation, we found the boundary between the two options is not that clear. Probably we will end up with a mixed solution, where a relatively smaller portion is redesign (mostly the parts that interact with ASP.NET Core) and the remaining is straightforward port from our existing code.

# SWAG

The scale of SWAG is **full-time** **week(s)/person**. Below listed the **key tasks** to implement Web API OData on ASP.NET Core. Please note that they are NOT listed in a specific order.

*P0 – Must-have (can’t work otherwise)*

*P1 – Nice-to-have (follow ASP.NET Core’s best practices)*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Category | Task | SWAG | Details | Priority |
| Prerequisite | Support .NET Core in ODL | 2 | 1. Add new TFM: netstandard1.1 2. Fully test against netstandard1.1 runtime  3. Modify nuspec and test package output on TFS | P0 |
| Routing | Use Controller instead of ODataController | 2=1+1 | 1. Convention-based routing.  2. ODataFormattingAttribute. | P0 |
| Support POCO class to be a Controller | 1 | Need get HttpContext to use DI | P2 |
| Implement OData route and routing constraints | 4=1+1+1+1 | 1. Implement ODataPathRouteConstraint from IRouteConstraint  2. Implement ODataRoute from Route (from ASP.NET Core)  3. Implement MapODataRoute in UseMvc().  4. Implement ODataActionSelector from IActionSelector | P0 |
| Implement attribute routing | 5=1+1+3 | 1. Investigate on whether we can use attributes from MVC  2. Remove ODataRouteAttribute  3. Change attribute routing to use MVC’s attributes | P0 |
| Infrastructure | Use Dependency Injection from ASP.NET Core | 2 | 1. Remove our own DI implementation and use the one from ASP.NET Core | P0 |
| Implement IODataFeature | 3=1+2 | 1. Remove ODataProperties()  2. Design and implement IODataFeature as part of the feature collection of HttpContext. | P1 |
| Adapt to trivial breaking changes from ASP.NET Core | 4-5 | 1. Namespace changes  2. Class name changes  3. Find alternatives for missing classes  4. EF Core related changes (like ColumnTypeAttribute->  DataTypeAttribute)  … | P0 |
| Query | Redesign EnableQueryAttribute | 2 | 1. Implement new EnableQueryAttribute from IFilterFactory  2. Implement the original logic in EnableQueryFilter from IResultFilter (or continue to use IActionFilter). | P1 |
| Formatter | Implement ODataInputFormatter | 2 |  | P0 |
| Implement ODataOutputFormatter | 2 |  | P0 |
| Implement API to register the formatters | 1 | AddMvc().AddODataFormatters()?  Or  services.AddMvc(options =>  {  options.OutputFormatters.Add(new ODataOutputFormatter());  }); | P0 |
| Batch | Work around | 2 | Need have work around to achieve currently batch support. | P2 |
| Testing | Port existing tests/  Add new tests | 4-8 |  | P0 |
| Documentation/Samples | 4-6 |  | P1 |

In total: **about 39 weeks/person, or about 10 months/person.** Please note that **this number doesn’t include effort on docs/samples and meeting/engineering/QE overhead.**

# Technical Details

## Support .NET Core in ODL

Reference: <https://docs.microsoft.com/en-us/dotnet/articles/standard/library>

We have Profile111 now which can directly run on netstandard1.1 runtimes so the fastest way to support .NET Core is to just copy that assembly into a new NuGet folder “netstandard1.1”.

Why we choose netstandard1.1 NOT HIGHER?

* Profile111 supports at least netstandard1.1 (see reference). And we don’t use any new APIs from netstandard1.2 and above.
* However, the lower version, the better compatibility. A higher version means dropping more supported platforms.
* Most basic building block packages (like DI) target netstandard1.1. We should learn from that.

Though the docs ensure that Profile111 can run perfectly on runtimes that support netstandard1.1, we still need a thorough test on the new packages!

## Use Controller instead of ODataController

### Changes in ASP.NET Core

In ASP.NET v4.x, MVC (which uses Controller) and Web API (which uses ApiController) are two components. But in ASP.NET Core, they were merged into MVC so we only have Controller now and the users don’t necessarily need to inherit their controllers from Controller (though they are encouraged to do so) if they don’t want to use HttpContext.

### Changes in Web API OData

Previously we have ODataController inherited from ApiController and all OData controllers must inherit ODataController. But now to keep our behavior consistent, **ODataController should be removed** and Web API OData should be able to handle the following situations:

* POCO controller type

|  |
| --- |
| public class PeopleController  {  public IQueryable<Person> GetAll()  {  return DataSource.People;  }  } |

* Controller’s subtype

|  |
| --- |
| public class PeopleController : Controller  {  public IActionResult GetAll()  {  return ObjectResult(DataSource.People);  }  } |

## Implement OData route and routing constraints

0. **DON’T consider to implement a IRouter for OData from scratch**. That would let us lose all the useful features from MVC (attribute routing, action filters, routing constraints, etc.) **OData should be based on MVC.**

1. **Implement ODataPathRouteConstraint** from IRouteConstraint: Minor API change from ASP.NET Core. Should be nearly straightforward port.

2. **Implement** **ODataRoute** that inherits from Route: Simple wrap. It takes the ODataPathRouteConstraint as its parameter.

3. **Implement MapODataRoute()** in UseMvc().

|  |
| --- |
| public class Startup  {  public void Configure(IApplicationBuilder app)  {  …  app.UseMvc(routes =>  {  // Traditional HTTP route.  routes.MapRoute(  name: "default",  template: "{controller=Home}/{action=Index}/{id?}");  // OData route.  routes.MapODataRoute(  name: “odata”,  prefix: “odata”);  });  }  }  public static class RouteBuilderExtensions  {  public static void MapODataRoute(this IRouteBuilder routes, string name, string prefix)  {  var constraint = new ODataPathRouteConstraint(prefix);  var route = new ODataRoute(name, constraint);  routes.Add(route);  }  } |

4. **Implement ODataActionSelector** from IActionSelector: some API change from ASP.NET Core. Need redesign to align with the tree-decision selector.

Previously we used to inject the action selector in ODataRoutingAttribute: <https://github.com/OData/WebApi/blob/master/OData/src/System.Web.OData/OData/ODataRoutingAttribute.cs>. Thus we can **remove ODataRoutingAttribute**.

Now we **register ODataActionSelector as a DI service**.

|  |
| --- |
| public class Startup  {  public void ConfigureServices(IServiceCollection services)  {  …  services.AddSingleton<IActionSelector, ODataActionSelector>();  …  }  } |

And in the end, we can **remove the previous MapODataServiceRoute()** overloads,

## Implement attribute routing

Previously we have ODataRouteAttribute with routing templates to put on either Controller or Action that tell the routing system which requests should be routed to a controller or an action.

|  |
| --- |
| public class TodoController  {  [ODataRoute("Todo({id})")]  public IActionResult GetById(string id)  {  …  }  } |

But in ASP.NET Core MVC, we have many new attributes to use: HttpGet, HttpPost, etc. Preserving ODataRouteAttribute looks weird and inconsistent. We’d better conduct an investigation to see if these attributes from MVC support OData routing templates (like the example above, not key-as-segment).

|  |
| --- |
| public class TodoController  {  [HttpGet("{id}", Name = "GetTodo")]  public IActionResult GetById(string id)  {  …  }  } |

If yes, we can go **removing ODataRouteAttribute** and **change the attribute routing to** **use those attributes provided by MVC**. This require significant changes in OData attribute routing because we are now using different attributes for different routes and thus we need to validate the attributes applied on controllers and actions as well.

## Use Dependency Injection from ASP.NET Core

In Web API OData v6.x, we added dependency injection support and we can access the request container by

|  |
| --- |
| request.GetRequestContainer().GetRequiredService<…>(); |

But in ASP.NET Core, the dependency injection is also integrated. We can access the request container by

|  |
| --- |
| (HttpContext)context.RequestServices.GetRequiredService<…>(); |

Though the underlying DI framework implementation is the same (MS DI Framework), they are basically two different request container instances which would cause a lot of troubles. Thus, we need to unify them into one. Of course, the one from ASP.NET Core runtime wins. And we need to **replace all the occurrences of “request.GetRequestContainer” with “context.RequestServices”**. Of course, this requires the HttpContext to be present in any place where HttpRequestMessage used to appear.

We don’t register services through MapODataServiceRoute (it’s removed anyway) anymore. Instead we register the services in Startup.ConfigureServices. Probably we can add an extension method called AddOData() to register the default services from both ODL and WAO.

|  |
| --- |
| public class Startup  {  public void ConfigureServices(IServiceCollection services)  {  …  services.AddOData();  // Users’ custom OData services go here.  …  }  }  public static class ServiceCollectionExtensions  {  public static void AddOData(IServiceCollection services)  {  var builder = new DefaultContainerBuilder(services);  builder.AddDefaultODataServices(); // Services from ODL  builder.AddDefaultWebApiServices(); // Services from WAO  }  } |

We will **modify the constructor of DefaultContainerBuilder** to take an existing ServiceCollection and to continue adding default OData services into that collection.

## Implement IODataFeature

In ASP.NET Core, Properties or handlers associated with a request are now grouped into what we known as Request Features (<https://docs.asp.net/en/latest/fundamentals/request-features.html>), we can **add an interface IODataFeature** to store all the request-related OData properties, ODataPath, ODataQueryOptions, ODataMessageReader/WriterSettings, etc. By doing so, we can entirely **retire ODataProperties()**. This is not a required feature but a nice-to-have one that conforms to the ASP.NET Core’s best practices (though you must be angry about a longer expression…).

|  |  |
| --- | --- |
| Previous | Now |
| request.ODataProperties().Path | (HttpContext)context.Features.Get<IODataFeature>().Path |

## Redesign EnableQueryAttribute

Proposed changes are:

1) **Use IResultFilter.OnResultExecuting instead of IActionFilter.OnActionExecuted**. The reason is that we should only apply query options (logic from EnableQueryAttribute) if the action successfully returns (no exception thrown).

|  |
| --- |
| public interface IActionFilter : IFilterMetadata  {  // Before action is executed, good place to do validation.  void OnActionExecuting(ActionExecutingContext context);  // After action is executed but before the result it returns is executed (sent to client).  // Will be called regardless if the action returns successful result or not.  void OnActionExecuted(ActionExecutedContext context);  } |
| public interface IResultFilter : IFilterMetadata  {  // Before the result is executed (sent to client), good place to apply query options.  // The action must have successfully returned the result.  void OnResultExecuting(ResultExecutingContext context);    // After the result is executed, the response has much likely been sent by now.  void OnResultExecuted(ResultExecutedContext context);  } |

2) **Implement EnableQueryAttribute from IFilterFactory instead of from IActionFilter directly**. The advantage of doing so is that we can then access DI container and inject services into EnableQueryAttribute.

|  |
| --- |
| public class EnableQueryAttribute : Attribute, IFilterFactory  {  // Implement IFilterFactory  public IFilterMetadata CreateInstance(IServiceProvider serviceProvider)  {  return new EnableQueryFilter();  }  }  Internal class EnableQueryFilter : IResultFilter  {  public void OnResultExecuting(ResultExecutingContext context)  {  // The original EnableQuery logic goes here.  }  public void OnResultExecuted(ResultExecutedContext context)  {  }  } |

We can also choose to use TypeFilterAttribute or ServiceFilterAttribute to achieve a similar result but the user code would look ugly.

We can also choose to **add the EnableQueryFilter to global filters** by

|  |
| --- |
| public class Startup  {  public void ConfigureServices(IServiceCollection services)  {  …  services.AddMvc(options => options.Filters.Add(new EnableQueryFilter());  …  }  } |

This will apply the query options globally on any controller and action. Just a possible option for some specific scenarios.

## Implement OData formatters

Previously we implement the formatter logic in ODataMediaTypeFormatter but now in ASP.NET Core, we need to split it into **ODataInputFormatter** and **ODataOutputFormatter**.

|  |
| --- |
| public class ODataInputFormatter : TextInputFormatter  {  …  // Move the logic here from previously ODataMediaTypeFormatter.ReadFromStringAsync()  public async void ReadResponseBodyAsync(…) { … }  …  } |
| public class ODataOutputFormatter : TextOutputFormatter  {  …  // Move the logic here from previously ODataMediaTypeFormatter.WriteToStreamAsync()  public async void WriteResponseBodyAsync(…) { … }  …  } |

All the underlying logic behind should mostly remain the same. That said, it’s a straightforward port for serializers and deserializers. For the implementation of ODataInputFormatter and ODataOutputFormatter, **we can reference the open-source implementation of JsonInputFormatter/JsonOutputFormatter from ASP.NET Core.**

We also need to implement a new API to **register the two formatters into MVC**. Here is a possible implementation. We CANNOT just call AddMvc().AddODataFormatters() in AddOData() because users may have much to configure in AddMvc()!

|  |
| --- |
| // This is a configure pattern from ASP.NET Core.  public class ODataFormattersMvcOptionsSetup : ConfigureOptions<MvcOptions>  {  public ODataFormattersMvcOptionsSetup : base(ConfigureMvc) {}  public static void ConfigureMvc(MvcOptions options)  {  // or get the formatter types from DI container?  options.OutputFormatters.Add(new ODataOutputFormatter());  options.InputFormatters.Add(new ODataInputFormatter());  }  }  public static class MvcBuilderExtensions  {  public static void AddODataFormatters(this IMvcBuilder builder)  {  builder.services.Add<IConfigureOptions<MvcOptions>, ODataFormattersMvcOptionsSetup>();  }  }  public class Startup  {  public void ConfigureServices(IServiceCollection services)  {  …  services.AddMvc(…).AddODataFormatters();  …  }  } |

After this, **ODataFormattingAttribute can be safely removed from ODataController** which is an important step to help remove ODataController itself.