I build my list like this:

public static List<SearchFormula> SearchData(string searchString)

{

var searchResults = new List<SearchFormula>();

SqlDataReader drResults = FormulaUtility.SearchFormulas(searchString);

if ((drResults != null) && (drResults.HasRows))

{

while (drResults.Read())

{

searchResults.Add(new SearchFormula()

{

// id use the GetValue function

Title = drResults.GetString(1),

Description = drResults.GetString(2),

Url = drResults.GetString(3)

// total use the GetValue Function

});

}

}

return searchResults;

}

Using this Object:

public class SearchFormula

{

public string Title { get; set; }

public string Description { get; set; }

public string Url { get; set; }

}

I began using the IHttpActionResult, returning the OK(results); function. I believe this is what started me down the confusing road. I had successfully sent an ArrayList but this did not serialize the way I thought it would.

I tried changing it to ActionResult and attempted to return Json(result) Result being the actual list.

I would like to continue to use the IhttpActionResult and send the serialized data with the OK() method. I also seem to be having a conflict between the built-in json serializer and NewtonSoft json serializer.

What should I use. What is the simplest way of just serializing a generic list and passing the result into the IHttpActionResult OK() method?

I tried the JavaScriptSerializer but it returns XML not Json...

public class SearchController : ApiController

{

public IHttpActionResult Get(string searchTerm)

{

var jsonSerialiser = new JavaScriptSerializer();

var jsonResult = jsonSerialiser.Serialize(SearchUtility.SearchData(searchTerm));

if (jsonResult != null)

{

return Ok(jsonResult);

}

return NotFound();

}

}

Here is the Json.Net Example:

public class SearchController : ApiController

{

public IHttpActionResult Get(string searchTerm)

{

var jsonResult = JsonConvert.SerializeObject(SearchUtility.SearchData(searchTerm));

if (jsonResult != null)

{

return Ok(jsonResult);

}

return NotFound();

}

}

I have tried the MemoryStream... blah blah blah... nothing seems like a clean, straightforward approach and there is no subject matter for this specific solution.

Let me start with this...

How can I serialize a Generic list to Json?

How can I send that result through the IHttpActionResult?

\****Update*\***

This is what I am getting for the serialization from Json.Net. BUT something is wrong with the format... Even Fiddler can not determine that it is Json. My Header looks like this (in Fiddler):

I take this approach which seems a lot simpler and does not involve changing the json serializer for the data that you have.

If you return the objects as a list the default media type formatter will handle the serialization based on the content type specified from the client (providing it is json or xml).

For demo purposes add the below methods which return hard coded objects.

// GET api/search

public List<SearchFormula> Get(string searchTerm)

{

var searchItems = SearchData(searchTerm);

return searchItems;

}

public static List<SearchFormula> SearchData(string searchString)

{

var searchResults = new List<SearchFormula>();

searchResults.Add(new SearchFormula { Description = "desc1", Title = "title1", Url = "http://url.com" });

searchResults.Add(new SearchFormula { Description = "desc2", Title = "title2", Url = "http://url.com" });

return searchResults;

}

https://stackoverflow.com/questions/20620917/return-json-from-generic-list-in-web-api

How do I correctly return a list of "CarTypes" objects (from the second method), where the TyreID that is passed in, is not the primary key of the CarType class - so for example, I want to return a list of all CarTypes, where the TyreID is 5:

// GET api/CarTypes

public IEnumerable<CarTypes> GetCarTypes()

{

return db.CarTypes.AsEnumerable(); //This works fineCar

}

// GET api/CarTypes/5

public IEnumerable<CarTypes> GetCarTypes(long id)

{

CarTypes cartypes = db.CarTypes.Select(t => t.TyreID == id).AsEnumerable();

if (roomtypes == null)

{

throw new HttpResponseException(Request

.CreateResponse(HttpStatusCode.NotFound));

}

return cartypes;

}

It currently shows the error:

Cannot implicitly convert type 'System.Collections.Generic.IEnumerable' to 'MvcApplication4.Models.CarTypes'. An explicit conversion exists (are you missing a cast?)

And does it matter if I use Select/SelectMany/Where in the query?

Firstly you need to use Where instead of Select; secondly you don't need to use AsEnumerable() after you've changed it to Where but you might have to call ToList() so that the Linq2Sql/EntityFramework executes the query before returning the values to the view.

// GET api/CarTypes/5

public IEnumerable<CarTypes> GetCarTypes(long id)

{

var cartypes = db.CarTypes.Where(t => t.TyreID == id).ToList();

if (cartypes == null || !cartypes.Any())

{

throw new HttpResponseException(Request.CreateResponse(HttpStatusCode.NotFound));

}

return cartypes;

}

I've also added in an additional check after the query has executed but you might not need this depending on how you want to handle an empty collection.

Shouldn't you have:

IEnumerable<CarTypes> cartypes = db.CarTypes.Where(t => t.TyreID == id).AsEnumerable();

Instead of:

CarTypes cartypes = db.CarTypes.Select(t => t.TyreID == id).AsEnumerable();

Note: I would have made this a comment under PanJanek's answer but I'm not currenlty allowed beacuse of my low reputation...

https://stackoverflow.com/questions/10946794/asp-net-mvc-web-api-return-list-of-objects

Here is my situation. I built a web service that is designed to populate a report. It receives a variable number of "findings" and then generates a report containing all of the findings.

Is there a way to post, using JSON, multiple findings at once, and have it bind to a List object?

EDIT:

So to be more specific, a finding would look like:

{ title: "title", description: "desc", rating: "High"}

I would like to be able to have my function look like this:

[HttpPost]

public string Post(IList<Finding> findings){

//code...

}

public class Finding{

public string title {get; set;}

//...

}

So essentially I want to bind an array of those JSON findings into an IList

UPDATE: I am looking to be able to have it bind automatically. I am currently able to workaround by posting a JSON string (using JSON.stringify)

Here is my code that works:

[HttpPost]

public string Post([FromBody]object jsonString){

IList<Finding> findingList = JsonConvert.DeserializeObject<IList<Finding>>(jsonString.toString());

//...

}

if you stay with your solution :

[HttpPost]

public string Post([FromBody]object jsonString){

IList<Finding> findingList = JsonConvert.DeserializeObject<IList<Finding>> (jsonString.toString());

//...

}

i think that you can :

[HttpPost]

public string Post(string jsonString){

IList<Finding> findingList = JsonConvert.DeserializeObject<IList<Finding>> (jsonString);

//...

}

...And in your ajax call :

$.ajax({

...

data: JSON.Stringify(data),

...

});

OR,

you can try something like (it comes from my code) :

var LoginFormSubmitHandler = function (e) {

var $form = $(this);

// We check if jQuery.validator exists on the form

if (!$form.valid || $form.valid()) {

$.post($form.attr('action'), $form.serializeArray())

...

...and can you show the value of $form.serializeArray() ??

https://stackoverflow.com/questions/14654146/use-json-to-post-a-collection-of-models-to-webapi

I am trying to use Web API to grab certain fields from my MVC controller. I can't seem to match the right type with the right list. I am fine with converting everything to string.

I either get an error in code (can not convert types), or if I get it to compile, I get this error:

"The 'ObjectContent`1' type failed to serialize the response body for content type 'application/xml; charset=utf-8'."

From other similar posts, people responded with how to create a list, but not with the declaration of the return value of the Get. Please include both.

Also I would prefer not to add additional controllers as I need to do this on a number of my models.

Here is my code--note you can see I tried a number of different ways:

public class APICLIENTsController : ApiController

{

private ApplicationDbContext db = new ApplicationDbContext();

// GET api/<controller>

public IEnumerable<string> Get()

//public IEnumerable<CLIENT> Get()

{

//return db.CLIENTs.OrderBy(x => x.CLIENTNAME).ToList();

string[] listOfUsers = db.CLIENTs.OrderBy(x => x.CLIENTNAME).Select(r => new

{

ID = r.CLIENTID.ToString(),

NAME = r.CLIENTNAME

});

return listOfUsers.ToList();

//return db.CLIENTs.Select(x => new { x.CLIENTNAME }).ToArray();

}

Your query is returning a collection of anonymous objects, not string[] so it will throw an exception. Even if you were to generate string[] by concatenating the CLIENTID and CLIENTNAME properties, it would be a little use to the client.

Create a model to represent what you need to return to the view

public class ClientVM

{

public int ID { get; set; }

public string Name { get; set; }

}

and modify your method to

public IEnumerable<ClientVM> Get()

{

IEnumerable<ClientVM> model = db.CLIENTs.OrderBy(x => x.CLIENTNAME).Select(r => new ClientVM

{

ID = r.CLIENTID,

Name = r.CLIENTNAME

});

return model;

}

Side note: depending on how your calling and consuming this in the client, you may need to change the Content-Type to specifically return json (refer [these answers](https://stackoverflow.com/questions/9847564/how-do-i-get-asp-net-web-api-to-return-json-instead-of-xml-using-chrome) for more detail)

If you want to return JSON use the

JsonResult

type.

public JsonResult Get()

{

//return db.CLIENTs.OrderBy(x => x.CLIENTNAME).ToList();

string[] listOfUsers = db.CLIENTs.OrderBy(x => x.CLIENTNAME).Select(r => new

{

ID = r.CLIENTID.ToString(),

NAME = r.CLIENTNAME

});

return Json(listOfUsers.ToList(), JsonRequestBehavior.AllowGet);

}

https://stackoverflow.com/questions/38020928/proper-way-to-return-json-list-in-web-api-with-mvc-and-partial-model

**What I Learned at Build**

Based on Cesar’s presentation and various conversations with different Microsoft engineers here is what I learned.

* **.NET Core is the framework to use for Microservices**
  + It’s fast
  + It’s small
  + It’s the future
* **Containers/Docker for packaging applications**
  + It’s the next step in infrastructure abstraction at the application level
  + Great development and release stories
* **Use a container orchestrator**
  + Azure Container Service is a great cloud option offering many management APIs (Kubernetes, Mesos, Swarm)
  + ServiceFabric is the microservices framework that Azure uses internally
  + Microsoft doesn’t care if you use one of their orchestrators/services, they want .NET Core to work great with all of them
* **.NET Core microservices will work with outside services**
  + Microsoft knows customers already have investments in other vendors/products
  + This isn’t a lock-in play, they just want customers to be successful with .NET Core
  + They will remove any barriers to that end, including integrating/supporting 3rd party vendors/providers/services

**Summary**

Microsoft is thinking hard about how to make customers with .NET Core to build microservices.  We can all participate with them on Github as their guidance materializes.  We can expect to hear a more polished story in the coming months.  I’m guessing around the .NET Core 2.0 GA timeframe this fall.

What do you think about Microsoft approach? Are you excited about where they are heading?  I’m considering writing a series of posts about building microservices on .NET core, are you interested?  Please leave comments below to continue the conversation and add you two cents.

http://www.dotnetcatch.com/2017/05/23/microservices-update-from-build-2017/

|  |
| --- |
| / For more information on enabling MVC for empty projects, visit http://go.microsoft.com/fwlink/?LinkID=397860 |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| using Microsoft.AspNetCore.Hosting; |
|  |

|  |
| --- |
| using Microsoft.AspNetCore.Mvc; |
|  |

|  |
| --- |
| using Microsoft.Extensions.Options; |
|  |

|  |
| --- |
| using eShopOnContainers.WebSPA; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| namespace eShopConContainers.WebSPA.Server.Controllers |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| public class HomeController : Controller |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| private readonly IHostingEnvironment \_env; |
|  |

|  |
| --- |
| private readonly IOptionsSnapshot<AppSettings> \_settings; |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| public HomeController(IHostingEnvironment env, IOptionsSnapshot<AppSettings> settings) |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| \_env = env; |
|  |

|  |
| --- |
| \_settings = settings; |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| public IActionResult Configuration() |
|  |

|  |
| --- |
| { |
|  |

|  |
| --- |
| return Json(\_settings.Value); |
|  |

|  |
| --- |
| } |
|  |

|  |
| --- |
| } |
|  |

}

https://github.com/dotnet-architecture/eShopOnContainers/blob/master/src/Web/WebSPA/Server/Controllers/HomeController.cs

There are several reasons why it makes sense to build super-lightweight HTTP services (or, despite all the baggage the word brings, “microservices”). I do not need to go into all the operational or architectural benefits of such approach to system development, as [it has been discussed](https://martinfowler.com/articles/microservices.html) a lot elsewhere.

It feels natural that when building such HTTP services, it definitely makes sense to keep the footprint of the technology you chose as small as possible, not to mention the size of the codebase you should maintain long term.

In this point I wanted to show a couple of techniques for building very lightweight HTTP services on top ASP.NET Core, without the use of any framework, and with minimal code bloat.

**Prerequisites**

What I’ll be discussing in this article is based on ASP.NET Core 1.2 packages which at the time of writing have not shipped yet.

I am using the CI feed of ASP.NET Core, so my *Nuget.config* looks like this:

XHTML



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | <?xml version="1.0" encoding="utf-8"?>  <configuration>    <packageSources>      <add key="aspnetcidev" value="https://dotnet.myget.org/F/aspnetcore-ci-dev/api/v3/index.json" />    </packageSources>  </configuration> |

When version 1.2 ships to Nuget, this will no longer be required.

**ASP.NET HTTP endpoints without MVC**

ASP.NET Core allows you to define HTTP endpoints directly on top of the OWIN-like pipeline that it’s built around, rather than using the full-blown MVC framework and its controllers to handle incoming requests. This has been there since the very beginning – you could use middleware components to handle incoming HTTP requests and short circuit a response immediately to the client. A bunch of high profile ASP.NET Core based projects use technique like this already – for example Identity Server 4.

This is not a new concept – something similar existed (albeit in a limited fashion) in classic ASP.NET with HTTP modules and HTTP handlers. Later on, in Web API you could define message handlers for handling HTTP requests without needing to define controllers. Finally, in OWIN and in Project Katana, you could that by plugging in custom middleware components too.

Another alternative is to specify a custom [IRouter](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/routing) and hang the various endpoints off it. The main difference between this approach, and plugging in custom middleware components is that routing *itself* is a single middleware. It also gives you a possibility for much more sophisticated URL pattern matching and route constraint definition – something you’d need handle manually in case of middlewares.

ASP.NET Core 1.2 will introduce a set of [new extension methods](https://github.com/aspnet/Routing/blob/dev/src/Microsoft.AspNetCore.Routing/RequestDelegateRouteBuilderExtensions.cs) on *IRouter*, which will make creation of lightweight HTTP endpoints even easier. It would be possible to also polyfill earlier versions of ASP.NET Core with this functionality by simply copying these new extensions into your project.

**Setting up the base for a lightweight HTTP API**

Here is the *project.json* for our microservice. It contains only the most basic packages.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | {    "dependencies": {      "Microsoft.NETCore.App": {        "version": "1.1.0",        "type": "platform"      },      "Microsoft.AspNetCore.Server.IISIntegration": "1.2.0-preview1-23182",      "Microsoft.AspNetCore.Routing": "1.2.0-preview1-23182",      "Microsoft.AspNetCore.Server.Kestrel": "1.2.0-preview1-23182",      "Microsoft.Extensions.Configuration.Json": "1.2.0-preview1-23182",      "Microsoft.Extensions.Logging": "1.2.0-preview1-23182",      "Microsoft.Extensions.Logging.Console": "1.2.0-preview1-23182",      "Microsoft.Extensions.Options.ConfigurationExtensions": "1.2.0-preview1-23182"    },      "frameworks": {      "netcoreapp1.0": {        "imports": [          "dotnet5.6",          "portable-net45+win8"        ]      }    },      "buildOptions": {      "emitEntryPoint": true,      "preserveCompilationContext": true    },      "runtimeOptions": {      "configProperties": {        "System.GC.Server": true      }    },      "publishOptions": {      "include": [        "wwwroot",        "appsettings.json",        "web.config"      ]    },      "scripts": {      "postpublish": [ "dotnet publish-iis --publish-folder %publish:OutputPath% --framework %publish:FullTargetFramework%" ]    }  } |

We are using the bare minimum here:

* Kestrel and IIS integration to act as server host
* routing package
* logging and configuration packages

In order to keep our code to absolute minimum too, we can even ditch the *Startup* class concept for our API setup, and just do all of the backend API code in a **single file**. To do that, instead of hooking into the typical *Startup* extensibility points like *Configure()* and *ConfigureServices()* methods, we’ll hang everything off *WebHostBuilder*.

*WebHostBuilder* is quite often ignored/overlooked by ASP.NET Core developers, because it’s generated by the template as the entry point inside the *Program* class, and usually you don’t even need to modify it – as it by default points at *Startup* class where almost all of the set up and configuration work happens. However, it also exposes similar hooks that *Startup* has, so it is possible to just define everything on *WebHostBuilder* directly.

Our basic API configuration is shown below. It doesn’t do anything yet in terms of exposing HTTP endpoints, but it’s fully functional from the perspective of the pipeline set up (router is wired in), logging to console and consuming configuration from JSON and environment variables.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | public class Program      {          public static void Main(string[] args)          {              var config = new ConfigurationBuilder()                  .SetBasePath(Directory.GetCurrentDirectory())                  .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)                  .AddEnvironmentVariables().Build();                var host = new WebHostBuilder()                  .UseKestrel()                  .UseConfiguration(config)                  .UseContentRoot(Directory.GetCurrentDirectory())                  .UseIISIntegration()                  .ConfigureLogging(l => l.AddConsole(config.GetSection("Logging")))                  .ConfigureServices(s => s.AddRouting())                  .Configure(app =>                  {                      // to do - wire in our HTTP endpoints                  })                  .Build();                host.Run();          }      } |

I love this approach because it’s so wonderfully concise. In roughly 20 lines of code we have an excellent base for a lightweight HTTP API. Naturally we could enrich this with more features as need – for example add in your own custom services or add token validation using the relevant integration packages from Microsoft.Security or IdetntityServer4.

**Adding HTTP endpoints to our solution**

The final step is to add our HTTP endpoints. We’ll do that using the aforementioned extension methods which will be introduced in ASP.NET Core 1.2. For demonstration purposes we’ll also need some sample model and emulated data, so I’ll be using my standard [Contact](https://github.com/filipw/aspnetcore-api-samples/blob/master/_01%20lightweight/LightweightApi/Program.cs#L125) and [ContactRepository](https://github.com/filipw/aspnetcore-api-samples/blob/master/_01%20lightweight/LightweightApi/Program.cs#L137) examples.

The code below goes into the *Configure()* extension method on the *WebHostBuilder* as it was noted before already. It shows the HTTP handlers for getting all contacts and getting a contact by ID.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | app.UseRouter(r =>  {      var contactRepo = new InMemoryContactRepository();        r.MapGet("contacts", async (request, response, routeData) =>      {          var contacts = await contactRepo.GetAll();          await response.WriteJson(contacts);      });        r.MapGet("contacts/{id:int}", async (request, response, routeData) =>      {          var contact = await contactRepo.Get(Convert.ToInt32(routeData.Values["id"]));          if (contact == null)          {              response.StatusCode = 404;              return;          }            await response.WriteJson(contact);      });  }); |

This code should be rather self descriptive – we delegate the look up operation to the backing repoistory, and then simply write out the result to the HTTP response. The router extension methods also gives us access to route data values, making it easy to handle complex URIs. On top of that, we can use regular ASP.NET Core route templates with all the power of the constraints which is very handy – for example, just like you’d expect, *contacts/{id:int}* will not be matched for non-integer IDs.

Additionally, I helped myself a bit by adding a convenience extension method to write to the response stream.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | public static class HttpExtensions  {      public static Task WriteJson<T>(this HttpResponse response, T obj)      {          response.ContentType = "application/json";          return response.WriteAsync(JsonConvert.SerializeObject(obj));      }  } |

The final step is to add in the HTTP endpoints that would modify the state on the server side:

* POST a new contact
* PUT a contact (modify existing)
* DELETE a contact

We’ll need an extra extension method to simplify that – that is because have to deserialize the request body stream into JSON, and we’d also like to validate the data annotations on our model to ensure the request payload from the client is valid. Obviously it would be silly to repeat that code over and over.

This extension method is shown below, and it makes use of JSON.NET and the *System.ComponentModel.DataAnnotations.Validator*.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | public static class HttpExtensions  {      private static readonly JsonSerializer Serializer = new JsonSerializer();        public static async Task<T> ReadFromJson<T>(this HttpContext httpContext)      {          using (var streamReader = new StreamReader(httpContext.Request.Body))          using (var jsonTextReader = new JsonTextReader(streamReader))          {              var obj = Serializer.Deserialize<T>(jsonTextReader);                var results = new List<ValidationResult>();              if (Validator.TryValidateObject(obj, new ValidationContext(obj), results))              {                  return obj;              }                httpContext.Response.StatusCode = 400;              await httpContext.Response.WriteJson(results);                return default(T);          }      }  } |

Notice that the method will short-circuit a 400 Bad Request response back to the client if the model is not valid (for example a required field was missing) – and it will pass the validation errors too.

The HTTP endpoint definitions are shown next.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | r.MapPost("contacts", async (request, response, routeData) =>  {      var newContact = await request.HttpContext.ReadFromJson<Contact>();      if (newContact == null) return;        await contactRepo.Add(newContact);        response.StatusCode = 201;      await response.WriteJson(newContact);  });    r.MapPut("contacts/{id:int}", async (request, response, routeData) =>  {      var updatedContact = await request.HttpContext.ReadFromJson<Contact>();      if (updatedContact == null) return;        updatedContact.ContactId = Convert.ToInt32(routeData.Values["id"]);      await contactRepo.Update(updatedContact);        response.StatusCode = 204;  });    r.MapDelete("contacts/{id:int}", async (request, response, routeData) =>  {      await contactRepo.Delete(Convert.ToInt32(routeData.Values["id"]));      response.StatusCode = 204;  }); |

And that’s it – you could improve this further by adding for example convenience methods of reading and casting values from *RouteDataDictionary*. It is also not difficult to enhance this code with authentication and even integrate the new ASP.NET Core authorization policies into it.

Our full “microservice” code (without the helper extension methods, I assume you’d want to centralize and reuse them anyway) is shown below – and I’m quite pleased with the result. I find it a very appealing, concise way of building lightweight APIs in ASP.NET Core.

The full source code is [here on Github](https://github.com/filipw/aspnetcore-api-samples/tree/master/01%20Lightweight%20API%20%28no%20MVC%29).



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76 | public class Program  {      public static void Main(string[] args)      {          var config = new ConfigurationBuilder()              .SetBasePath(Directory.GetCurrentDirectory())              .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)              .AddEnvironmentVariables().Build();            var host = new WebHostBuilder()              .UseKestrel()              .UseConfiguration(config)              .UseContentRoot(Directory.GetCurrentDirectory())              .UseIISIntegration()              .ConfigureLogging(l => l.AddConsole(config.GetSection("Logging")))              .ConfigureServices(s => s.AddRouting())              .Configure(app =>              {                  // define all API endpoints                  app.UseRouter(r =>                  {                      var contactRepo = new InMemoryContactRepository();                        r.MapGet("contacts", async (request, response, routeData) =>                      {                          var contacts = await contactRepo.GetAll();                          await response.WriteJson(contacts);                      });                        r.MapGet("contacts/{id:int}", async (request, response, routeData) =>                      {                          var contact = await contactRepo.Get(Convert.ToInt32(routeData.Values["id"]));                          if (contact == null)                          {                              response.StatusCode = 404;                              return;                          }                            await response.WriteJson(contact);                      });                        r.MapPost("contacts", async (request, response, routeData) =>                      {                          var newContact = await request.HttpContext.ReadFromJson<Contact>();                          if (newContact == null) return;                            await contactRepo.Add(newContact);                            response.StatusCode = 201;                          await response.WriteJson(newContact);                      });                        r.MapPut("contacts/{id:int}", async (request, response, routeData) =>                      {                          var updatedContact = await request.HttpContext.ReadFromJson<Contact>();                          if (updatedContact == null) return;                            updatedContact.ContactId = Convert.ToInt32(routeData.Values["id"]);                          await contactRepo.Update(updatedContact);                            response.StatusCode = 204;                      });                        r.MapDelete("contacts/{id:int}", async (request, response, routeData) =>                      {                          await contactRepo.Delete(Convert.ToInt32(routeData.Values["id"]));                          response.StatusCode = 204;                      });                  });              })              .Build();            host.Run();      }  } |

https://www.strathweb.com/2017/01/building-microservices-with-asp-net-core-without-mvc/

https://medium.com/@FurryMogwai/building-a-basket-micro-service-using-asp-net-core-and-akka-net-ea2a32ca59d5

https://github.com/dotnet/docs/blob/master/docs/csharp/tutorials/microservices.md