MICROSOFT ARCHITECT
By Joydip Kanjilal, Columnist, InfoWorld
OCT 19, 2020 3:00 AM PDT

How to use IHttpClientFactory in ASP.NET Core

Take advantage of IHttpClientFactory to create HttpClient instances seamlessly and avoid having to manage their lifetimes manually

When working in applications in ASP.NET Core you might often need to invoke the Web API action methods using HttpClient to check if the endpoints are working fine. To achieve this, you would typically instantiate HttpClient and use the instance to invoke your action methods. However, there are certain downsides to using HttpClient directly, mainly having to do with managing the lifetimes of the instances manually.

You can avoid these downsides by using IHttpClientFactory to create your HttpClient instances. Introduced in .NET Core 2.1, IHttpClientFactory provides a central place to name, configure, and create HttpClient instances and manages the pooling and lifetimes of the instances automatically.

[Also on InfoWorld: The most valuable software developer skills in 2020]

I've discussed HttpClient and HttpClientFactory in an earlier article here. This article further discusses HttpClient and IHttpClientFactory with code examples to illustrate the concepts covered. To work with the code examples provided, you should have Visual Studio 2019 installed in your system. If you don't already have a copy, you can download Visual Studio 2019 here.

Create an ASP.NET Core MVC project in Visual Studio 2019

First off, let's create an ASP.NET Core project in Visual Studio 2019. Assuming Visual Studio 2019 is installed in your system, follow the steps outlined below to create a new ASP.NET Core project in Visual Studio.

1. Launch the Visual Studio IDE.
2. Click on “Create new project.”
3. In the “Create new project” window, select “ASP.NET Core Web Application” from the list of templates displayed.
4. Click Next.
5. In the “Configure your new project” window, specify the name and location for the new project.
6. Optionally check the “Place solution and project in the same directory” check box, depending on your preferences.
7. Click Create.
8. In the “Create a New ASP.NET Core Web Application” window shown next, select .NET Core as the runtime and ASP.NET Core 3.1 (or later) from the drop-down list at the top.
9. Select “Web Application (Model-View-Controller)” as the project template to create a new ASP.NET Core MVC application.
10. Ensure that the check boxes “Enable Docker Support” and “Configure for HTTPS” are unchecked as we won’t be using those features here.
11. Ensure that Authentication is set to “No Authentication” as we won’t be using authentication either.
12. Click Create.

Following these steps should create a new ASP.NET Core MVC project in Visual Studio 2019. In the new project, create a new API controller and save it using the default name, i.e., ValuesController. We’ll use this project in the sections that follow.

HttpClient challenges

Although the HttpClient class doesn’t implement the IDisposable interface directly, it extends the System.Net.Http.HttpMessageInvoker class, which does implement IDisposable. Nevertheless, when working with HttpClient instances, you shouldn’t be disposing of them. Although you can call the Dispose method on an HttpClient instance, it is not a recommended practice.

What should you do instead? One option is to make the HttpClient instance static or wrap a non-static instance of the HttpClient inside a custom class and make it a singleton class. But a better alternative is to use IHttpClientFactory to retrieve an instance of HttpClient and then use the instance for calling action methods.

IHttpClientFactory and HttpClientFactory

IHttpClientFactory is an interface that is implemented by the DefaultHttpClientFactory class, which is an opinionated factory. The DefaultHttpClientFactory implements the IHttpClientFactory and IHttpMessageHandlerFactory interfaces. IHttpClientFactory was introduced to provide ASP.NET Core with excellent built-in support for creating, caching, and disposing of HttpClient instances.

Note that HttpClientFactory (discussed in my earlier article) is just a helper to create HttpClient instances configured with the handlers provided. This class has the following methods:

Create(DelegatingHandler[])
Create(HttpMessageHandler, DelegatingHandler[])
CreatePipeline(HttpMessageHandler, IEnumerable<DelegatingHandler>)

The overloaded Create methods of the HttpClientFactory class look like this:

```
public static HttpClient Create(params DelegatingHandler[] handlers)
{
return Create(new HttpClientHandler(), handlers);
}
```

Source:
https://github.com/aspnet/HttpClientFactory/blob/master/src/Microsoft.Extensions.Http/IHttpClientFactory.cs

```
public static HttpClient Create(HttpMessageHandler innerHandler, params DelegatingHandler[] handlers)
{
HttpMessageHandler pipeline = CreatePipeline(innerHandler, handlers);
return new HttpClient(pipeline);
}
```

Source:
https://github.com/aspnet/HttpClientFactory/blob/master/src/Microsoft.Extensions.Http/IHttpClientFactory.cs

Both HttpClientFactory and IHttpClientFactory were introduced to better manage the lifetime of HttpMessageHandler instances.

Why use IHttpClientFactory?

When you dispose of a HttpClient instance, the connection remains open for up to four minutes. Further, the number of sockets that you can open at any point in time has a limit — you can't have too many sockets open at once. So when you use too many HttpClient instances, you might end up exhausting your supply of sockets.

Here's where IHttpClientFactory comes to the rescue. You can take advantage of IHttpClientFactory to create HttpClient instances for invoking HTTP API methods by adhering to the best practices to avoid issues faced with HttpClient. The primary goal of IHttpClientFactory in ASP.NET Core is to ensure that HttpClient instances are created using the factory while at the same time eliminating socket exhaustion.

Register an IHttpClientFactory instance in ASP.NET Core

You can register an instance of type IHttpClientFactory in the ConfigureServices method of the Startup class by calling the AddHttpClient extension method on the IServiceCollection instance as shown in the code snippet given below.

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddControllersWithViews();
   services.AddHttpClient();
}
```

Inject an IHttpClientFactory instance to your controllers in ASP.NET Core

You can then inject an IHttpClientFactory instance to your controllers as shown in the code snippet given below.

```
public class HomeController : Controller
{
      private IHttpClientFactory _httpClientFactory;
      private readonly ILogger<HomeController> _logger;
      public HomeController(ILogger<HomeController> logger,
      IHttpClientFactory httpClientFactory)
      {
            _logger = logger;
            _httpClientFactory = httpClientFactory;
      }
   //Your action methods go here
}
```

Call your action methods using HttpClient in ASP.NET Core

To create an HttpClient using IHttpClientFactory, you should call the CreateClient method. Once the HttpClient instance is available, you can use the following code in the Index action method of the HomeController class to invoke the Get method of the ValuesController class.

```
public async Task<IActionResult> Index()
{
   HttpClient httpClient = _httpClientFactory.CreateClient();
   httpClient.BaseAddress = new Uri("http://localhost:1810/");
   var response = await httpClient.GetAsync("/api/values");
   string str = await response.Content.ReadAsStringAsync();
   List<string> data = JsonSerializer.Deserialize<List<string>>(str);
   return View(data);
}
```

Use IHttpClientFactory to create and manage HttpClient instances in ASP.NET Core

There are several ways you can use IHttpClientFactory in your application. These include using IHttpClientFactory directly, using named clients, and using typed clients.

The basic or general usage pattern—i.e., using IHttpClientFactory directly—has already been discussed in the preceding sections. Refer to the section “Register an IHttpClientFactory instance” that discusses how you can register an HttpClient instance.

Using named clients

If you would like to use HttpClient instances with different configurations, named clients is a good choice. The following code snippet illustrates how you can create a named client.

```
services.AddHttpClient("github", c =>
{
    c.BaseAddress = new Uri("https://api.github.com/");
    c.DefaultRequestHeaders.Add("Accept",
    "application/vnd.github.v3+json");
    c.DefaultRequestHeaders.Add("User-Agent", "This is a test user agent");
});
```

Using typed clients

A typed client is defined using a custom class that wraps an HttpClient instance, encapsulating the logic for calls to all of the endpoints over the HTTP protocol. The following code snippet illustrates how a custom HttpClient class can be defined.

```
public class ProductService : IProductService
{
    private IHttpClientFactory _httpClientFactory;
    private readonly HttpClient _httpClient;
    private readonly string _baseUrl = "http://localhost:1810/";
    public ProductService(HttpClient httpClient)
    {
        _httpClient = httpClient;
    }
    public async Task<Catalog> GetAllProducts()
    {
        _httpClient = _httpClientFactory.CreateClient();
        _httpClient.BaseAddress = new Uri(_baseUrl);
        var uri = "/api/products";
        var result = await _httpClient.GetStringAsync(uri);
        return JsonConvert.DeserializeObject<Product>(result);
    }
}
```

The following code snippet shows how you can register your custom typed HttpClient.

```
services.AddHttpClient<IProductService, ProductService>();
```

Add MessageHandlers to the pipeline in ASP.NET Core

A Message handler is a class that extends the HttpMessageHandler class, accepts an HTTP request, and returns an HTTP response. If you would like to build your own message handler, you should create a class that extends the DelegatingHandler class.

You can add HttpMessageHandlers to the request processing pipeline. If you're using a named client, you can use the following code in the ConfigureServices method of the Startup class to add message handlers to the pipeline.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpClient("github", c =>
    {
        c.BaseAddress = new Uri("https://api.github.com/");
    })
    .AddHttpMessageHandler<DemoHandler>();
    services.AddTransient<DemoHandler>();
}
```

IHttpClientFactory is an opinionated factory that has been available since .NET Core 2.1. If you use IHttpClientFactory to create your HttpClient instances, then the pooling and lifetime of the underlying HttpClientMessagehandler instances are managed for you automatically. IHttpClientFactory also takes care of common concerns such as logging.

How to do more in ASP.NET Core:

- How to use the ProblemDetails middleware in ASP.NET Core
- How to create route constraints in ASP.NET Core
- How to manage user secrets in ASP.NET Core
- How to build gRPC applications in ASP.NET Core
- How to redirect a request in ASP.NET Core
- How to use attribute routing in ASP.NET Core
- How to pass parameters to action methods in ASP.NET Core MVC
- How to use API Analyzers in ASP.NET Core
- How to use route data tokens in ASP.NET Core
- How to use API versioning in ASP.NET Core
- How to use Data Transfer Objects in ASP.NET Core 3.1
- How to handle 404 errors in ASP.NET Core MVC
- How to use dependency injection in action filters in ASP.NET Core 3.1
- How to use the options pattern in ASP.NET Core
- How to use endpoint routing in ASP.NET Core 3.0 MVC
- How to export data to Excel in ASP.NET Core 3.0
- How to use LoggerMessage in ASP.NET Core 3.0
- How to send emails in ASP.NET Core
- How to log data to SQL Server in ASP.NET Core
- How to schedule jobs using Quartz.NET in ASP.NET Core
- How to return data from ASP.NET Core Web API
- How to format response data in ASP.NET Core
- How to consume an ASP.NET Core Web API using RestSharp
- How to perform async operations using Dapper
- How to use feature flags in ASP.NET Core
- How to use the FromServices attribute in ASP.NET Core
- How to work with cookies in ASP.NET Core
- How to work with static files in ASP.NET Core
- How to use URL Rewriting Middleware in ASP.NET Core
- How to implement rate limiting in ASP.NET Core
- How to use Azure Application Insights in ASP.NET Core

- Using advanced NLog features in ASP.NET Core
- How to handle errors in ASP.NET Web API
- How to implement global exception handling in ASP.NET Core MVC
- How to handle null values in ASP.NET Core MVC
- Advanced versioning in ASP.NET Core Web API
- How to work with worker services in ASP.NET Core
- How to use the Data Protection API in ASP.NET Core
- How to use conditional middleware in ASP.NET Core
- How to work with session state in ASP.NET Core
- How to write efficient controllers in ASP.NET Core

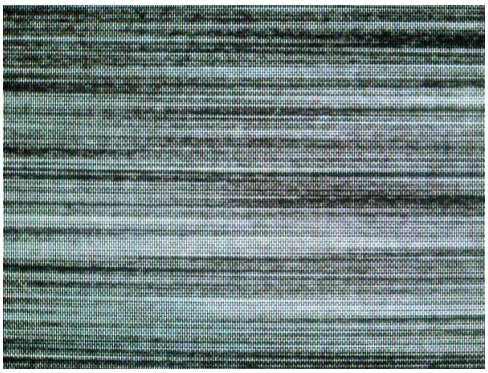
Joydip Kanjilal is a Microsoft MVP in ASP.Net, as well as a speaker and author of several books and articles. He has more than 20 years of experience in IT including more than 16 years in Microsoft .Net and related technologies.

Follow

YOU MAY ALSO LIKE

When to use Task.WaitAll vs. Task.WhenAll in .NET

Deno Company forms to back Node.js rival



Python developers want static typing

Are industry clouds an opportunity or a distraction?

3 enterprise AI success stories

Red Hat OpenShift ramps up security and manageability with...

How IT priorities are shifting during the COVID-19 crisis

2 common cloud computing predictions for 2021 are wrong

2020 was a year of rapid progression of digital transformation for businesses. The following is a snapshot of the digital transformation advancements made across all facets of business.