**Fetching Data and Content Negotiation with HttpClient in ASP.NET Core**

In this article, we are going to learn how to integrate and use HttpClient in ASP.NET Core Applications. We are going to consume the Web API’s resources while learning about different HttpClient functionalities. We are going to learn how to fetch data from Web API and how to use the HttpRequestMessage class directly to accomplish that. In our future articles, we are going to learn how to send POST, PUT and DELETE requests as well as how to send PATCH requests with HttpClient.

To download a source code, you can visit the **[main branch of our repository](https://github.com/CodeMazeBlog/httpclient-aspnetcore" \t "_blank)** to get the starter projects. For the finished project of this article, you can visit our **[Using HttpClient in ASP.NET Core](https://github.com/CodeMazeBlog/httpclient-aspnetcore/tree/fetching-data-with-httpclient" \t "_blank)** repository.

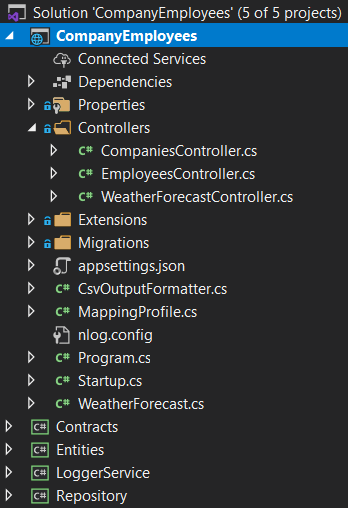
You can also visit our **[HttpClient Tutorial page](https://code-maze.com/httpclient-with-asp-net-core-tutorial" \t "_blank)**, to see all the articles from this tutorial.

We are going to divide this article into the following sections:

* [**Starting Projects Overview**](https://code-maze.com/fetching-data-with-httpclient-in-aspnetcore/#project-overview)
* [**About HttpClient**](https://code-maze.com/fetching-data-with-httpclient-in-aspnetcore/#about-httpclient)
* [**Using HttpClient in ASP.NET Core Applications to Send a GET Request**](https://code-maze.com/fetching-data-with-httpclient-in-aspnetcore/#using-httpclient-for-get-requests)
* [**Supporting Different Response Formats**](https://code-maze.com/fetching-data-with-httpclient-in-aspnetcore/#support-different-response-format)
* [**Preference One over Another Accept Header**](https://code-maze.com/fetching-data-with-httpclient-in-aspnetcore/#adding-preference)
* [**Using HttpRequestMessage Class to Send HTTP Requests**](https://code-maze.com/fetching-data-with-httpclient-in-aspnetcore/#httprequestmessage)
* [**Conclusion**](https://code-maze.com/fetching-data-with-httpclient-in-aspnetcore/#conclusion)

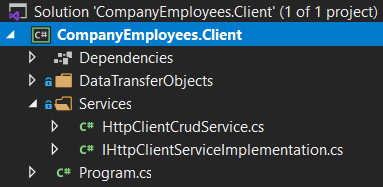
**Starting Projects Overview**

If you open our main branch of the HttpClient repository, you will find two projects: CompanyEmployees and CompanyEmployees.Client. The first project is the ASP.NET Core Web API project, and it will be our server-side project for this tutorial. It consists of several projects:

**[](https://code-maze.com/wp-content/uploads/2021/01/01-Web-Api-Project-Structure.png)**

The API project isn’t that important for our article and the complete series as well. We use this project in our **[Ultimate ASP.NET Core Web API book](https://code-maze.com/ultimate-aspnet-core-web-api/" \t "_blank)**, and if you are interested more in that topic, feel free to visit the linked page. The important part is that it uses the SQL database, so all you have to do is to modify the connection string in the appsettings.json file and run the Update-Migration command. All the required data will be seeded in the database.

Then, there is the client application – ASP.NET Core Console Application. All it has is a single service HttpClientCrudService, which we are going to modify in this article, a single interface IHttpClientServiceImplementation all the HttpClient services will inherit from, data transfer classes and a modified Program class:

**[](https://code-maze.com/wp-content/uploads/2021/01/02-Client-application-for-using-HttpClient-in-ASP.NET-Core-applications.png)**

Of course, let’s show the current code from the Program class:

class Program

**{**

static async Task Main**(**string**[]** args**)**

**{**

var services = new ServiceCollection**()**;

ConfigureServices**(**services**)**;

var provider = services.BuildServiceProvider**()**;

try

**{**

await provider.GetRequiredService**<**IHttpClientServiceImplementation**>()**

.Execute**()**;

**}**

catch **(**Exception ex**)**

**{**

Console.WriteLine**(**$"Something went wrong: {ex}"**)**;

**}**

**}**

private static void ConfigureServices**(**IServiceCollection services**)**

**{**

services.AddScoped**<**IHttpClientServiceImplementation, HttpClientCrudService**>()**;

**}**

**}**

Nothing special here. We prepare our service collection, add service to the IOC, and execute the default method from the service class. As we add different services in this entire tutorial, we are going to expand the ConfigureServices method.

**About HttpClient**

We won’t dive too deep into the theory about HttpClient since we are going to learn a lot from our examples, but let’s look at some basics.

HttpClient is a class that enables us to send HTTP requests and receive HTTP responses from resources identified by URI. We can use this class to send all kinds of HTTP requests like GET, POST, PUT, DELETE, PATCH… and accept responses from the server.

HttpClient uses HTTP message handlers to send requests and get responses. This is the main job of the default message handler. If we read Microsoft’s documentation, we are going to read that the default one for the [**.Net Framework and .NET Core 2.0 and earlier is HttpClientHander**](https://docs.microsoft.com/en-us/dotnet/api/system.net.http.httpclienthandler?view=net-5.0). But from the [**.NET Core 2.1 and later the default one is SocketsHttpHandler**](https://docs.microsoft.com/en-us/dotnet/api/system.net.http.socketshttphandler?view=net-5.0).

But, HttpClient doesn’t have to use only one message handler. We can attach multiple message handlers and create a pipeline. Some of those handlers can manipulate only the headers of the request, some may work with timeouts, etc.

**Using HttpClient in ASP.NET Core Applications to Send a GET Request**

Now, let’s start with the modification of the HttpClientCrudService class:

public class HttpClientCrudService : IHttpClientServiceImplementation

**{**

private static readonly HttpClient \_httpClient = new HttpClient**()**;

public HttpClientCrudService**()**

**{**

\_httpClient.BaseAddress = new Uri**(**"https://localhost:5001/api/"**)**;

\_httpClient.Timeout = new TimeSpan**(**0, 0, 30**)**;

**}**

public async Task Execute**()**

**{**

**}**

**}**

Here, we create a new HttpClient field, initialize it, and add the configuration in a constructor. As a configuration, we pass the URI of our API and set the time out of the request. Of course, we can find a lot more properties to use in this configuration, but for now, this will be enough. Later on, when we start learning about HttpClientFactory, we are going to move this configuration to a different place.

Now, we can add a new method in this class:

public async Task GetCompanies**()**

**{**

var response = await \_httpClient.GetAsync**(**"companies"**)**;

response.EnsureSuccessStatusCode**()**;

var content = await response.Content.ReadAsStringAsync**()**;

var companies = JsonSerializer.Deserialize**<**List**<**CompanyDto**>>(**content, \_options**)**;

**}**

In this method, we use the GetAsync shortcut method from HttpClient and pass the address of our controller’s action. We have to ensure that the response is successful, so we call the EnsureSuccessStatusCode method. Once we are sure that we have a response with a successful status code, we read the content of the response as a string. Finally, we deserialize our response to a list of companies. As you can see, we are using one additional parameter of type JsonSerializerOptions, so let’s add it to our class, and let’s call this method in the Execute method:

private readonly JsonSerializerOptions \_options;

public HttpClientCrudService**()**

**{**

\_httpClient.BaseAddress = new Uri**(**"https://localhost:5001/api/"**)**;

\_httpClient.Timeout = new TimeSpan**(**0, 0, 30**)**;

\_options = new JsonSerializerOptions **{** PropertyNameCaseInsensitive = true **}**;

**}**

public async Task Execute**()**

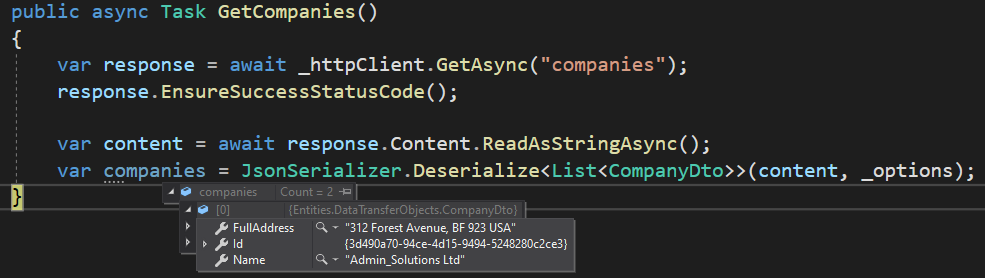
**{**

await GetCompanies**()**;

**}**

We are setting up the case insensitive deserialization option for our JsonSerializer. Without it, our response won’t be deserialized properly.

Now, we can add a breakpoint in the GetCompanies method, start the Web API project, and then start the client app:

**[](https://code-maze.com/wp-content/uploads/2021/01/03-HttpClient-Get-Request-working-solution.png)**

As we can see, we have our result in the companies variable.

We can continue on.

**Supporting Different Response Formats**

In this example, we received a JSON as a default response format. Our API supports that type by default. But some APIs don’t default to JSON, they maybe support XML as a default response format or any other. In such cases, our logic would not work.

In addition to JSON, our API supports an XML response format as well due to **[implemented Content Negotiation.](https://code-maze.com/content-negotiation-dotnet-core/" \t "_blank)** That said, let’s see how we can explicitly ask for a format in our client app.

First of all, the Http request, as well as the response, contains a set of headers, which we can use to pass additional information between the client and the server apps. The common header for HTTP requests is the Acceptheader. We use this header to tell the server which media type the client is going to accept: Accept: application/json, text/xml.

So, let’s see how we can set up the header in our requests:

public HttpClientCrudService**()**

**{**

\_httpClient.BaseAddress = new Uri**(**"https://localhost:5001/api/"**)**;

\_httpClient.Timeout = new TimeSpan**(**0, 0, 30**)**;

\_httpClient.DefaultRequestHeaders.Clear**()**;

\_httpClient.DefaultRequestHeaders.Accept.Add**(**

new MediaTypeWithQualityHeaderValue**(**"application/json"**))**;

\_httpClient.DefaultRequestHeaders.Accept.Add**(**

new MediaTypeWithQualityHeaderValue**(**"text/xml"**))**;

\_options = new JsonSerializerOptions **{** PropertyNameCaseInsensitive = true **}**;

**}**

Here, we use the DefaultRequestHeaders property and clear it out. Then, we use the Accept property, and since it is a collection, we add two MediaTypeWithQualityHeaderValue objects. With a first object, we support a JSON format, and with a second, we support the XML format. For this, we need to add a new using statement: using System.Net.Http.Headers;

Now, if we have a configuration like this one, we have to add some additional code in our method, to decide how to deserialize our response:

public async Task GetCompanies**()**

**{**

var response = await \_httpClient.GetAsync**(**"companies"**)**;

response.EnsureSuccessStatusCode**()**;

var content = await response.Content.ReadAsStringAsync**()**;

var companies = new List**<**CompanyDto**>()**;

if**(**response.Content.Headers.ContentType.MediaType == "application/json"**)**

**{**

companies = JsonSerializer.Deserialize**<**List**<**CompanyDto**>>(**content, \_options**)**;

**}**

else if**(**response.Content.Headers.ContentType.MediaType == "text/xml"**)**

**{**

var doc = XDocument.Parse**(**content**)**;

foreach **(**var element in doc.Descendants**())**

**{**

element.Attributes**()**.Where**(**a =**>** a.IsNamespaceDeclaration**)**.Remove**()**;

element.Name = element.Name.LocalName;

**}**

var serializer = new XmlSerializer**(**typeof**(**List**<**CompanyDto**>))**;

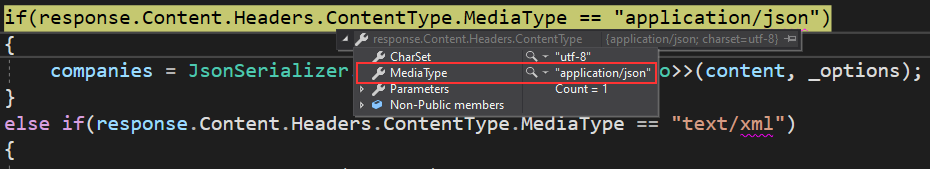
companies = **(**List**<**CompanyDto**>)**serializer.Deserialize**(**new StringReader**(**doc.ToString**()))**;

**}**

**}**

So, since we support both JSON and XML formats, we have to check which ContentType is applied to the response. If it is JSON, we just do the standard deserialization. But if it is XML, we parse the Content into the XDocument type and then just remove the declarations and use a LocalName for the Name property. Finally, we create a new XmlSerializer and deserialize our XDocument.

At this point, if we start both applications, and place a breakpoint inside the method, we will see that our default format is JSON:

**[](https://code-maze.com/wp-content/uploads/2021/01/04-Default-applicaiton-json-media-type.png)**

Of course, once we continue execution, we will get our companies as a result (as we did previously).

**Preference One over Another Accept Header in HttpClient**

With our Accept header setup, we support two formats with equal preference. The value of the preference is 1, which is a maximum value. But, we can set a lower preference for one of these two headers – the value must be between 0 and 1. The one with the higher preference will have an advantage.

So, let’s lower the preference of the JSON Accept header in our constructor:

\_httpClient.DefaultRequestHeaders.Clear**()**;

\_httpClient.DefaultRequestHeaders.Accept.Add**(**

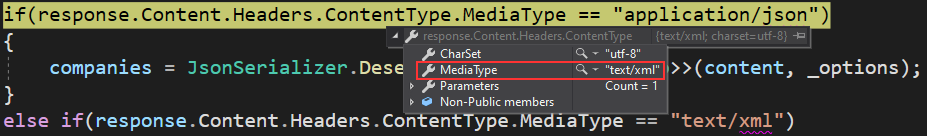
new MediaTypeWithQualityHeaderValue**(**"application/json", 0.9**))**;

\_httpClient.DefaultRequestHeaders.Accept.Add**(**

new MediaTypeWithQualityHeaderValue**(**"text/xml"**))**;

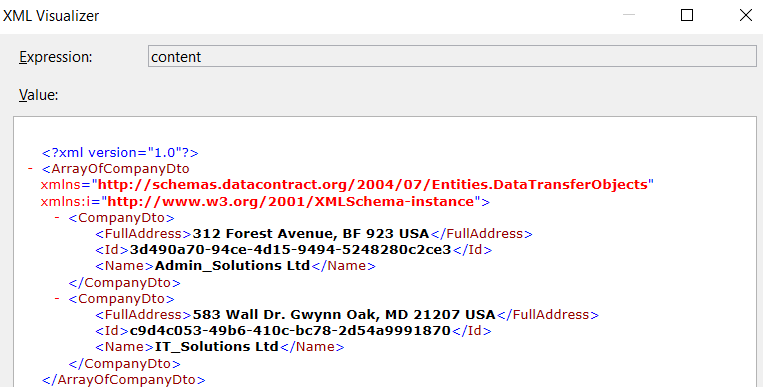
As we can see, the MediaTypeWithQualityHeaderValue constructor accepts another parameter. We set the value for it to 0.9. Since we didn’t add any value for the XML Accept header, the value is 1 by default.

Now, if we start our applications, we are going to find out that XML is the format we prefer:

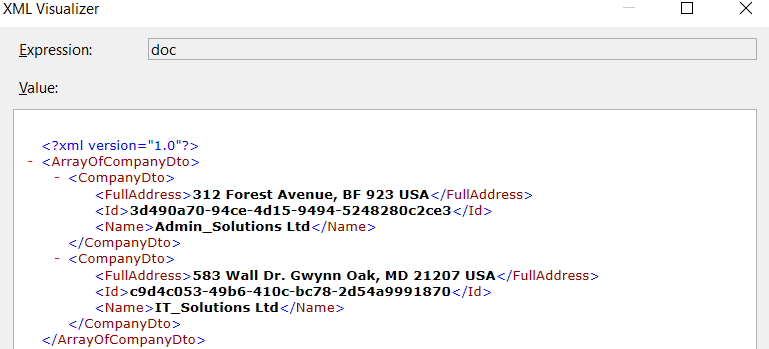
**[](https://code-maze.com/wp-content/uploads/2021/01/05-Media-type-XML-for-Http-Response.png)**

So, execution will skip this part and execute our XML deserialization.

Let’s inspect the response body before XDocument parsing:

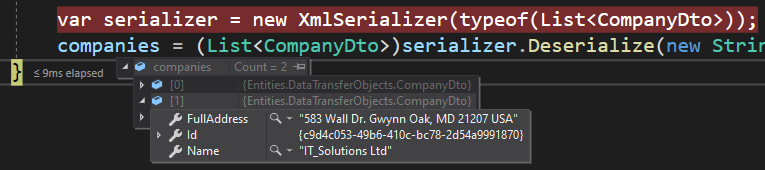
**[](https://code-maze.com/wp-content/uploads/2021/01/06-XML-response-body-in-HTTP-Response.png)**

And then, let’s inspect our doc variable after the parsing actions:

**[](https://code-maze.com/wp-content/uploads/2021/01/07-XML-response-body-after-parsing.png)**

We can see the difference.

After the parsing actions, our deserialization completes successfully:

**[](https://code-maze.com/wp-content/uploads/2021/01/08-Deserialized-XML-response-into-a-list.png)**

Excellent. We’ve seen how we can add preference to our HTTP Accept header in our request.

But now the question emerges.

What if we want to use some headers for some requests and other headers for other requests?

**Using HttpRequestMessage Class to Send HTTP Requests**

With this implementation, we are using the same header configuration for each request. So, if want to send an HTTP request that defaults to JSON format, we can’t do that with the HTTP configuration in this class. That’s because we set the XML format to be the default one.

This means we have to provide a different solution.

If we think this through, we can conclude that the BaseAddress and Timeout properties are related to HttpClient, but the properties of the Accept header are connected to the request itself. Also, when we use the GetAsync method, it internally creates a new HttpRequestMessage with the GET HTTP method. That’s why we call it the shortcut method. That said, we can create our own HttpRequestMessage and provide headers for that request.

The best practice is to set up the default configuration on the HttpClient instance and the request configuration on the HTTP request itself. Of course, if we always want to use the JSON format for the Accept header, we can set it up on the HttpClient instance level.

**Implementation**

Now, let’s see how we can implement the HTTP request using the HttpRequestMessage class.

First, let’s remove the accept header configuration from our constructor:

public HttpClientCrudService**()**

**{**

\_httpClient.BaseAddress = new Uri**(**"https://localhost:5001/api/"**)**;

\_httpClient.Timeout = new TimeSpan**(**0, 0, 30**)**;

\_httpClient.DefaultRequestHeaders.Clear**()**;

\_options = new JsonSerializerOptions **{** PropertyNameCaseInsensitive = true **}**;

**}**

Then, we can revert the GetCompanies method to its previous implementation:

public async Task GetCompanies**()**

**{**

var response = await \_httpClient.GetAsync**(**"companies"**)**;

response.EnsureSuccessStatusCode**()**;

var content = await response.Content.ReadAsStringAsync**()**;

var companies = JsonSerializer.Deserialize**<**List**<**CompanyDto**>>(**content, \_options**)**;

**}**

Finally, we can add our new method:

public async Task GetCompaniesWithXMLHeader**()**

**{**

var request = new HttpRequestMessage**(**HttpMethod.Get, "companies"**)**;

request.Headers.Accept.Add**(**new MediaTypeWithQualityHeaderValue**(**"text/xml"**))**;

var response = await \_httpClient.SendAsync**(**request**)**;

response.EnsureSuccessStatusCode**()**;

var content = await response.Content.ReadAsStringAsync**()**;

var doc = XDocument.Parse**(**content**)**;

foreach **(**var element in doc.Descendants**())**

**{**

element.Attributes**()**.Where**(**a =**>** a.IsNamespaceDeclaration**)**.Remove**()**;

element.Name = element.Name.LocalName;

**}**

var serializer = new XmlSerializer**(**typeof**(**List**<**CompanyDto**>))**;

var companies = **(**List**<**CompanyDto**>)**serializer.Deserialize**(**new StringReader**(**doc.ToString**()))**;

**}**

So, we create a new request with the HttpRequestMessage class providing the HTTP method as an argument and the address of our API action. Then, we add headers to our request and call the SendAsync method to send the request. After we extract the Content, we repeat the same steps we did in our previous method.

We have to do one final thing.

Let’s make sure that this method is called as soon as our client application starts:

public async Task Execute**()**

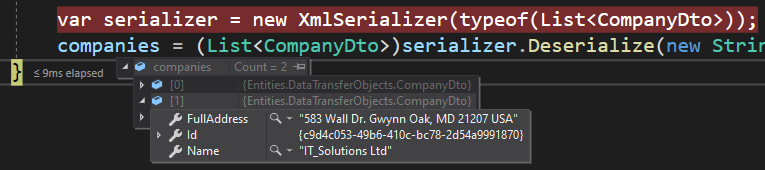
**{**

//await GetCompanies();

await GetCompaniesWithXMLHeader**()**;

**}**

As we did before, we are going to place a breakpoint in this method and start both apps:

**[](https://code-maze.com/wp-content/uploads/2021/01/08-Deserialized-XML-response-into-a-list.png)**

As you can see, we have the same result as before, but this time we are using a separate method with the HttpRequestMessage class to send an HTTP request with the XML Accept header.

**Conclusion**

In this article, we’ve talked about HttpClient, and how we can use it in our ASP.NET Core application to consume data from Web API.

To sum up, we have learned:

* More about the HttpClient class
* How to configure the required properties to use for each request
* The way to use the GetAsync shortcut method to send the HTTP Get request
* How to add headers to our requests and to add a preference for each one
* To use HttpRequestMessage to send the HTTP request

**POST, PUT, and DELETE Requests Using HttpClient in ASP.NET Core**

In **[the previous article](https://code-maze.com/fetching-data-with-httpclient-in-aspnetcore" \t "_blank)**, we have learned how to integrate HttpClient in ASP.NET Core, and how to use it to fetch the data from Web API. Also, we learned how to send the GET request using both the GetAsync method and the HttpRequestMessage class. As a continuation, in this article, we are going to learn how to send POST, PUT, and DELETE requests using HttpClient in ASP.NET Core. We are going to show you both examples for each request with shortcut methods (PostAsync, PutAsync, DeleteAsync) and with the HttpRequestMessage class.

To download a source code, you can visit our [**POST, PUT, DELETE Requests with HttpClient in ASP.NET Core**](https://github.com/CodeMazeBlog/httpclient-aspnetcore/tree/post-put-delete-with-httpclient) repository.

You can also visit our **[HttpClient Tutorial page](https://code-maze.com/httpclient-with-asp-net-core-tutorial/" \t "_blank)**, to see all the articles from this tutorial.

We are going to divide this article into the following sections:

* [**Sending a POST Request Using HttpClient in ASP.NET Core**](https://code-maze.com/httpclient-example-aspnet-core-post-put-delete/#sending-post-request)
* [**Using HttpClient in ASP.NET Core to Send a PUT Request**](https://code-maze.com/httpclient-example-aspnet-core-post-put-delete/#sending-put-request)
* [**Sending a DELETE Request with HttpClient**](https://code-maze.com/httpclient-example-aspnet-core-post-put-delete/#sending-delete-request)
* [**Conclusion**](https://code-maze.com/httpclient-example-aspnet-core-post-put-delete/#conclusion)

Let’s start.

**Sending a POST Request with HttpClient in ASP.NET Core**

If you have read our **[previous article](https://code-maze.com/fetching-data-with-httpclient-in-aspnetcore" \t "_blank)**, you know that we have the HttpClientCrudService class in the CompanyEmployees.Client application. This class already contains two methods, and we are going to expand it with all the methods from this article.

So, since the configuration is already prepared, we can add a new method to send the POST request to the Web API:

private async Task CreateCompany**()**

**{**

var companyForCreation = new CompanyForCreationDto

**{**

Name = "Eagle IT Ltd.",

Country = "USA",

Address = "Eagle IT Street 289"

**}**;

var company = JsonSerializer.Serialize**(**companyForCreation**)**;

var requestContent = new StringContent**(**company, Encoding.UTF8, "application/json"**)**;

var response = await \_httpClient.PostAsync**(**"companies", requestContent**)**;

response.EnsureSuccessStatusCode**()**;

var content = await response.Content.ReadAsStringAsync**()**;

var createdCompany = JsonSerializer.Deserialize**<**CompanyDto**>(**content, \_options**)**;

**}**

We first prepare a company object that we want to create and serialize it with the Serialize method. Then, we create a new StringContent object providing our serialized company, encoding type, and the media type arguments. After that, we use the PostAsync method to send the POST request to the API. After we receive a response, we check if it is a successful one. Then, we apply a well-known logic (from a previous article) by reading the content with the ReadAsStringAsync method and deserializing the content using the JsonSerializerOptions argument. Remember that we created the \_options parameter in a previous article.

At this point, we have to ensure that our client app calls this method. So, let’s modify the Execute method inside the class:

public async Task Execute**()**

**{**

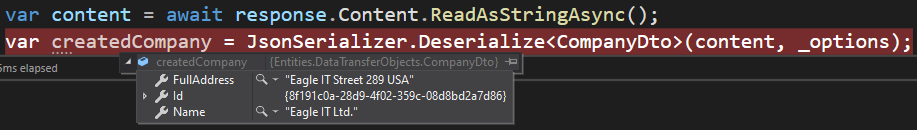
//await GetCompanies();

//await GetCompaniesWithXMLHeader();

await CreateCompany**()**;

**}**

Now, if we place a breakpoint in the CreateCompany method and start both applications:

**[](https://code-maze.com/wp-content/uploads/2021/01/09-Using-PostAsync-method-with-HttpClient-in-ASP.NET-Core-Created-Resource.png)**

We can see our created company as a result. You can also check the database if you want.

**Using HttpRequestMessage Class to Send the POST Request**

The PostAsync method is a shortcut method because it encapsulates the HttpRequestMessage class. And as we could see, it works great. But, if we want to have greater control over our request and also to explicitly set up different request options, like headers, we have to use the HttpRequestMessage class. So, let’s see how we can do that:

private async Task CreateCompanyWithHttpRequestMessage**()**

**{**

var companyForCreation = new CompanyForCreationDto

**{**

Name = "Hawk IT Ltd.",

Country = "USA",

Address = "Hawk IT Street 365"

**}**;

var company = JsonSerializer.Serialize**(**companyForCreation**)**;

var request = new HttpRequestMessage**(**HttpMethod.Post, "companies"**)**;

request.Headers.Accept.Add**(**new MediaTypeWithQualityHeaderValue**(**"application/json"**))**;

request.Content = new StringContent**(**company, Encoding.UTF8**)**;

request.Content.Headers.ContentType = new MediaTypeHeaderValue**(**"application/json"**)**;

var response = await \_httpClient.SendAsync**(**request**)**;

response.EnsureSuccessStatusCode**()**;

var content = await response.Content.ReadAsStringAsync**()**;

var createdCompany = JsonSerializer.Deserialize**<**CompanyDto**>(**content, \_options**)**;

**}**

Again, we start with a new companyForCreation object and its serialization. Then, we create a new HttpRequestMessage object and provide the type of the request and the endpoint’s address. After that, we add an accept header to determine which format we support as a response. Right after that, we populate the Content of our request by using the StringContent class and providing the serialized company object and encoding type. Also, we specify the media type of our request with the ContentType property.

**Note:** We can configure the ContentType in the constructor of the StringContent class as well, but with this implementation, it is easier to understand the process, and where the ContentType property is coming from.

To send the request, we use the SendAsync method. After we are sure a successful status code is returned, we read our content and deserialize it.

Now, we can modify the Execute method:

public async Task Execute**()**

**{**

//await GetCompanies();

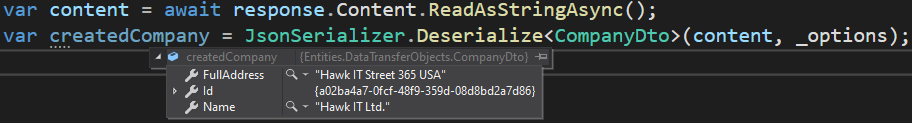
//await GetCompaniesWithXMLHeader();

//await CreateCompany();

await CreateCompanyWithHttpRequestMessage**()**;

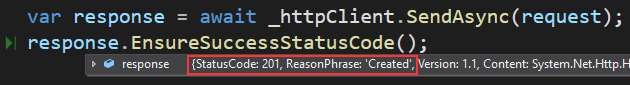
**}**

Place a breakpoint in our new method, and start our client app:

**[](https://code-maze.com/wp-content/uploads/2021/01/10-Using-HttpRequestMessage-to-Send-the-POST-request-with-HttpClient-in-ASP.NET-Core.png)**

There we go. Our company was successfully created.

Also, if we inspect our response, we will find the 201 – Created – status code:

**[](https://code-maze.com/wp-content/uploads/2021/01/11-Status-code-of-the-created-resource.png)**

Excellent.

We can continue.

**Using HttpClient in ASP.NET Core to Send a PUT Request**

Let’s see, how we can send a PUT request using HttpClinet in ASP.NET Core.

As we did with the POST request, we are going to use the shortcut method first, and then show how to do the same thing with the HttpRequestMessage class.

Before we modify the client project, we just want to mention that our Web API’s UpdateCompany action returns a NoContent response. So, there is no content to deserialize on the client-side, just a 204 status code. Of course, you can check the code on your own by visiting our [**GitHub repo**](https://github.com/CodeMazeBlog/httpclient-aspnetcore/tree/post-put-delete-with-httpclient).

That said, we are ready to add a new method in the HttpClientCrudService class:

private async Task UpdateCompany**()**

**{**

var updatedCompany = new CompanyForUpdateDto

**{**

Name = "Eagle IT Ltd.",

Country = "USA",

Address = "Eagle IT Street 289 Updated"

**}**;

var company = JsonSerializer.Serialize**(**updatedCompany**)**;

var requestContent = new StringContent**(**company, Encoding.UTF8, "application/json"**)**;

var uri = Path.Combine**(**"companies", "fc12c11e-33a3-45e2-f11e-08d8bdb38ded"**)**;

var response = await \_httpClient.PutAsync**(**uri, requestContent**)**;

response.EnsureSuccessStatusCode**()**;

**}**

In this method, we create a new updatedCompany object with a modified Address property. Then, as we did in the CreateCompany method, we serialize the object and create a new StringContent providing the serialized object, encoding type, and media type. After that, we create the URI to the controller’s action, which expects the Id of the company we are updating. As soon as we do that, we send our PUT request using the PutAsync shortcut method and just ensure that we receive a successful status code. In this case the 204 - NoContent status code.

With this in place, we can modify the Execute method:

public async Task Execute**()**

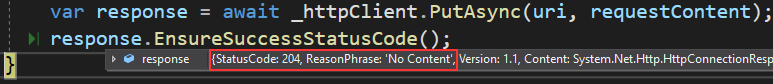
**{**

...

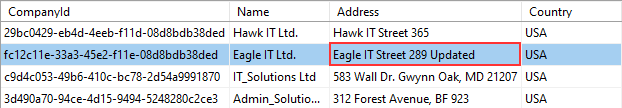
await UpdateCompany**()**;

**}**

Now, let’s put a breakpoint in the UpdateCompany method and start the app:

**[](https://code-maze.com/wp-content/uploads/2021/01/12-Using-PutAsync-method-to-update-a-resource-with-HttpClient-in-ASP.NET-Core.png)**

Also, if we inspect our database:

**[](https://code-maze.com/wp-content/uploads/2021/01/13-Updated-record-in-the-database.png)**

We can confirm our company is updated successfully.

**Using the HttpRequestMessage Class to Send the PUT Request**

As we already said, using the HttpRequestMessage class allows us more control over our requests. So, let’s see how we can utilize it to send the PUT request:

private async Task UpdateCompanyWithHttpRequestMessage**()**

**{**

var updatedCompany = new CompanyForCreationDto

**{**

Name = "Hawk IT Ltd.",

Country = "USA",

Address = "Hawk IT Street 365 Updated"

**}**;

var company = JsonSerializer.Serialize**(**updatedCompany**)**;

var uri = Path.Combine**(**"companies", "29bc0429-eb4d-4eeb-f11d-08d8bdb38ded"**)**;

var request = new HttpRequestMessage**(**HttpMethod.Put, uri**)**;

request.Headers.Accept.Add**(**new MediaTypeWithQualityHeaderValue**(**"application/json"**))**;

request.Content = new StringContent**(**company, Encoding.UTF8**)**;

request.Content.Headers.ContentType = new MediaTypeHeaderValue**(**"application/json"**)**;

var response = await \_httpClient.SendAsync**(**request**)**;

response.EnsureSuccessStatusCode**()**;

**}**

When we look at this method, we can see that it has almost the same implementation as the CreateCompanyWithHttpRequestMessage method.  Of course, here we create an additional uri parameter and we don’t deserialize our response body content since it is empty.

After the method implementation, let’s call it from the Execute method:

public async Task Execute**()**

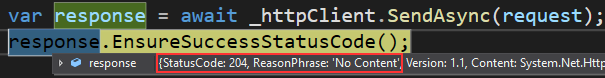
**{**

...

await UpdateCompanyWithHttpRequestMessage**()**;

**}**

And let’s start the app:

**[](https://code-maze.com/wp-content/uploads/2021/01/14-Using-HttpRequestMessage-to-send-a-PUT-request-with-HttpClient.png)**

There we go. We have a successful result.

Feel free to check the database as well.

**Sending a DELETE Request with HttpClient**

Since the DELETE request is the simplest of all the previous ones, we are just going to show the code.

So, let’s first see how to send a DELETE request with a shortcut DeleteAsync method:

private async Task DeleteCompany**()**

**{**

var uri = Path.Combine**(**"companies", "fc12c11e-33a3-45e2-f11e-08d8bdb38ded"**)**;

var response = await \_httpClient.DeleteAsync**(**uri**)**;

response.EnsureSuccessStatusCode**()**;

**}**

Also, let’s call this method from the Execute method:

public async Task Execute**()**

**{**

...

await DeleteCompany**()**;

**}**

As soon as we run the client app, we are going to get the 204 response. This is a valid response for the Delete action in Web API.

Now, let’s see how we can do the same thing using the HttpRequestMessage class:

private async Task DeleteCompanyWithHttpResponseMessage**()**

**{**

var uri = Path.Combine**(**"companies", "29bc0429-eb4d-4eeb-f11d-08d8bdb38ded"**)**;

var request = new HttpRequestMessage**(**HttpMethod.Delete, uri**)**;

request.Headers.Accept.Add**(**new MediaTypeWithQualityHeaderValue**(**"application/json"**))**;

var response = await \_httpClient.SendAsync**(**request**)**;

response.EnsureSuccessStatusCode**()**;

**}**

Again, nothing new here. Just one note. We are still adding the accept header to our request because some APIs return the content if something goes wrong. We then have to deserialize that content on the client-side.

And that’s it.

We can call this method from the Execute method and run the app.

We should get the 204 response.

**Conclusion**

So, in this article, we have learned how to send POST, PUT and DELETE requests from our client application using both the shortcut methods and the HttpRequestMessage class. Combining this with the knowledge we have from the **[previous article](https://code-maze.com/fetching-data-with-httpclient-in-aspnetcore/" \t "_blank)**, we have covered all the CRUD methods with the HttpClient class.

**Using HttpClient to Send HTTP PATCH Requests in ASP.NET Core**

In our two previous articles, we have covered a lot of ground regarding the CRUD operations with HttpClient. If you have read them both, you know **[how to use HttpClient to fetch data from API](https://code-maze.com/fetching-data-with-httpclient-in-aspnetcore" \t "_blank)**, and also to **[send POST, PUT and DELETE requests using HttpClient](https://code-maze.com/httpclient-example-aspnet-core-post-put-delete" \t "_blank)**. When we talk about the PUT request, we know we use it for the full update of our resources. But we can improve that with partial updates using the HTTP PATCH requests. So, in this article, we are going to show you how to use HttpClient to send HTTP PATCH requests to achieve partial updates of our resources, thus improving the application’s performance.

To download a source code, you can visit our **[Using HttpClient to Send HTTP PATCH Requests in ASP.NET Core](https://github.com/CodeMazeBlog/httpclient-aspnetcore/tree/patch-with-httpclient" \t "_blank)** repository.

You can also visit our **[HttpClient Tutorial page](https://code-maze.com/httpclient-with-asp-net-core-tutorial/" \t "_blank)**, to see all the articles from this tutorial.

We are going to divide this article into the following sections:

* [**More About an HTTP PATCH Request**](https://code-maze.com/using-httpclient-to-send-http-patch-requests-in-asp-net-core/#about-patch)
* [**Using HttpClient’s PatchAsync Method to Send the HTTP Patch Request**](https://code-maze.com/using-httpclient-to-send-http-patch-requests-in-asp-net-core/#patch-async)
* [**Using HttRequestMessage to Send the PATCH Request**](https://code-maze.com/using-httpclient-to-send-http-patch-requests-in-asp-net-core/#using-httprequestmessage-for-patch)
* [**Conclusion**](https://code-maze.com/using-httpclient-to-send-http-patch-requests-in-asp-net-core/#conclusion)

So, let’s get going.

**More About an HTTP PATCH Request**

As we already mentioned, we use the PUT request for full updates and the PATCH request for partial updates. But this is not the only difference between these two HTTP requests. First of all, the request body is different. If we inspect the PUT action on the Web API’s side, we can see that the body of the request is a simple object:

[FromBody] CompanyForUpdateDto company

But if we do the same inspection for the PATCH request:

[FromBody] JsonPatchDocument<EmployeeForUpdateDto> patchDoc

We can see that if we want to support a request body for the PATCH requests, we have to use the JsonPatchDocument class. This class helps us describing different sets of operations that we can execute with the PATCH request.

Also, for the PUT request, we use the application/json as a media type. But for the PATCH request, the preferred media type is application/json-patch+json. We can use the application/json media type for the HTTP PATCH request, but as we mentioned the preferred media type is application/json-patch+json, and we are going to use that one in our example.

**HTTP PATCH Operations**

The PATCH request can execute one or multiple operations as a part of the JSON array. That said, let’s see the request body of the PATCH request:

**[**

**{**

"op": "replace",

"path": "/name",

"value": "new name"

**}**,

**{**

"op": "remove",

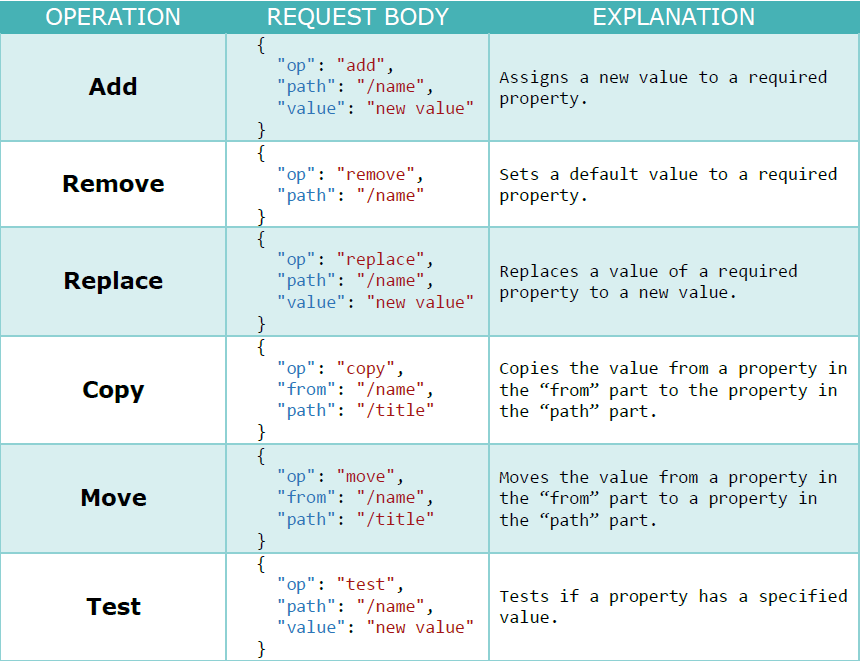
"path": "/name"

**}**

**]**

So, as we can see, the request body is basically an array of JSON objects that specify different operations. That said, we can confirm two operations: Replace and Remove, specified by the op property. The path part signifies the path to the object’s property that we want to modify – the Name property in this case. Finally, the value part represents a new value that we use to replace an old one for the Name property.

We have seen two operations but there are six operations in total. So, let’s explore all of them:

**[](https://code-maze.com/wp-content/uploads/2021/01/15-HTTP-PATCH-Operations.png)**

We have borrowed this picture from our **[Ultimate ASP.NET Core Web API book](https://code-maze.com/ultimate-aspnet-core-web-api/" \t "_blank)** because it explains all the operations in a clear manner.

Now, after the theory part, we can start with some coding.

**Using HttpClient’s PatchAsync Method to Send the HTTP Patch Request**

Before we start with the client project modification, we can quickly take a look at the route for the PATCH action on the API’s side:

**[**Route**(**"api/companies/{companyId}/employees"**)]**

**[**ApiController**]**

public class EmployeesController **:** ControllerBase

So, we can see that we have implemented the PATCH action inside the EmployeesController. Since a single employee can’t exist without a single company, the route to this controller is: api/companies/{companyId}/employees.  But, since we update only a single employee, we need an id for that employee:

**[**HttpPatch**(**"{id}"**)]**

public IActionResult PartiallyUpdateEmployeeForCompany**(**Guid companyId, Guid id, **[**FromBody**]** JsonPatchDocument**<**EmployeeForUpdateDto**>** patchDoc**)**

This means that the route for this action is: api/companies/{companyId}/employees/{id}.

We have removed the rest of the actions from this controller for the sake of simplicity, but to read in great detail about the restful API implementations of parent and child resources, refer to our **[mentioned book](https://code-maze.com/ultimate-aspnet-core-web-api/" \t "_blank)**.

Also, just for reference, let’s show the API’s implementation of the action:

**[**HttpPatch**(**"{id}"**)]**

public IActionResult PartiallyUpdateEmployeeForCompany**(**Guid companyId, Guid id, **[**FromBody**]** JsonPatchDocument**<**EmployeeForUpdateDto**>** patchDoc**)**

**{**

if**(**patchDoc == null**)**

**{**

\_logger.LogError**(**"patchDoc object sent from client is null."**)**;

return BadRequest**(**"patchDoc object is null"**)**;

**}**

var company = \_repository.Company.GetCompany**(**companyId, trackChanges: false**)**;

if **(**company == null**)**

**{**

\_logger.LogInfo**(**$"Company with id: {companyId} doesn't exist in the database."**)**;

return NotFound**()**;

**}**

var employeeEntity = \_repository.Employee.GetEmployee**(**companyId, id, trackChanges: true**)**;

if **(**employeeEntity == null**)**

**{**

\_logger.LogInfo**(**$"Employee with id: {id} doesn't exist in the database."**)**;

return NotFound**()**;

**}**

var employeeToPatch = \_mapper.Map**<**EmployeeForUpdateDto**>(**employeeEntity**)**;

patchDoc.ApplyTo**(**employeeToPatch**)**;

\_mapper.Map**(**employeeToPatch, employeeEntity**)**;

\_repository.Save**()**;

return NoContent**()**;

**}**

We are accepting the JsonPatchDocument from the request body. Next, we check the patchDoc object for a null value, and if the company and employee exist in the database. Then, we map from the Employee type to the EmployeeForUpdateDto type. It is important for us to do that because the patchDoc object can apply only to the EmployeeForUpdateDto type. After calling the ApplyTo method, we map again to the Employee type (from employeeToPatch to employeeEntity) and save changes in the database.

**Client-Side Implementation**

Now, let’s open the client project and add a new service in the Services folder:

public class HttpClientPatchService : IHttpClientServiceImplementation

**{**

private static readonly HttpClient \_httpClient = new HttpClient**()**;

public HttpClientPatchService**()**

**{**

\_httpClient.BaseAddress = new Uri**(**"https://localhost:5001/api/"**)**;

\_httpClient.Timeout = new TimeSpan**(**0, 0, 30**)**;

\_httpClient.DefaultRequestHeaders.Clear**()**;

**}**

public async Task Execute**()**

**{**

throw new NotImplementedException**()**;

**}**

**}**

So, this is the initial configuration of the HttpClient class with the base address and the timeout set. This is the same as we have in our previous example. Once we start learning about the HttpClientFactory, we are going to show you how to store the configuration in a single place without repeating it for each service.

After this, we can implement the logic for sending the HTTP PATCH request by using the shortcut PatchAsync method:

private async Task PatchEmployee**()**

**{**

var patchDoc = new JsonPatchDocument**<**EmployeeForUpdateDto**>()**;

patchDoc.Replace**(**e =**>** e.Name, "Sam Raiden Updated"**)**;

patchDoc.Remove**(**e =**>** e.Age**)**;

var uri = Path.Combine**(**"companies", "C9D4C053-49B6-410C-BC78-2D54A9991870", "employees", "80ABBCA8-664D-4B20-B5DE-024705497D4A"**)**;

var serializedDoc = JsonConvert.SerializeObject**(**patchDoc**)**;

var requestContent = new StringContent**(**serializedDoc, Encoding.UTF8, "application/json-patch+json"**)**;

var response = await \_httpClient.PatchAsync**(**uri, requestContent**)**;

response.EnsureSuccessStatusCode**()**;

**}**

Here, we create a new PATCH document with a help of the JsonPatchDocument class. To be able to use this class, we have to install the Microsoft.AspNetCore.JsonPatch library. Next, we create two operations (Replace and Remove) with the two helper methods from the JsonPatchDocument class. Then, we create a URI to the action, serialize the object, and create a new string content by providing our serialized object, encoding type, and media type. The important thing to notice here is that we don’t use JsonSerializer.Serialize() method from the System.Text.Json library but we use JsonConvert.SerializeObject() method from the Newtonsoft.Json library. We have to do this, otherwise, we get 400 bad request from our API since the patch document isn’t serialized well with System.Text.Json.

Finally, we send the request using the PatchAsync method and ensure that the response has a successful status code.

Now, let’s modify the Execute method:

public async Task Execute**()**

**{**

await PatchEmployee**()**;

**}**

And, let’s register this service in the Program class:

private static void ConfigureServices**(**IServiceCollection services**)**

**{**

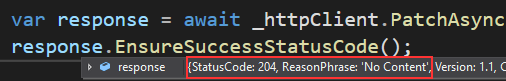
//services.AddScoped<IHttpClientServiceImplementation, HttpClientCrudService>();

services.AddScoped**<**IHttpClientServiceImplementation, HttpClientPatchService**>()**;

**}**

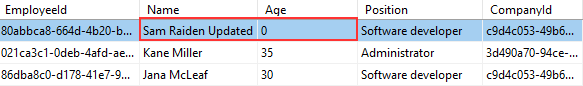
Excellent.

Let’s place the breakpoint in the PatchEmployee method and start both applications:

**[](https://code-maze.com/wp-content/uploads/2021/01/16-HTTP-Patch-Response-204.png)**

And we can see the 204 status code, with the No Content message.

Now, if we check the database:

**[](https://code-maze.com/wp-content/uploads/2021/01/17-Updated-columns-with-Patch-request.png)**

We can see the Name column is modified and the Age column is set to its default value of 0.

Excellent.

Let’s see how we can use the HttpRequestMessage to achieve the same thing.

**Using HttRequestMessage to Send the PATCH Request**

As we did with all the previous HTTP requests, we are going to use the HttpRequestMessage class to send the PATCH request to the server. We already talked about the benefits of this approach in our previous articles from **[this tutorial](https://code-maze.com/httpclient-with-asp-net-core-tutorial/" \t "_blank)**.

So, let’s add another method in the HttpClientPatchService class:

private async Task PatchEmployeeWithHttpRequestMessage**()**

**{**

var patchDoc = new JsonPatchDocument**<**EmployeeForUpdateDto**>()**;

patchDoc.Replace**(**e =**>** e.Name, "Sam Raiden"**)**;

patchDoc.Add**(**e =**>** e.Age, 28**)**;

var uri = Path.Combine**(**"companies", "C9D4C053-49B6-410C-BC78-2D54A9991870", "employees", "80ABBCA8-664D-4B20-B5DE-024705497D4A"**)**;

var serializedDoc = JsonConvert.SerializeObject**(**patchDoc**)**;

var request = new HttpRequestMessage**(**HttpMethod.Patch, uri**)**;

request.Headers.Accept.Add**(**new MediaTypeWithQualityHeaderValue**(**"application/json"**))**;

request.Content = new StringContent**(**serializedDoc, Encoding.UTF8**)**;

request.Content.Headers.ContentType = new MediaTypeHeaderValue**(**"application/json-patch+json"**)**;

var response = await \_httpClient.SendAsync**(**request**)**;

response.EnsureSuccessStatusCode**()**;

**}**

We create our JsonPatchDocument object again but this time, we revert the name of the employee and also add the age of 28. Then, we prepare URI and serialize the object. Once we do that, we create a new HttpRequestMessage providing the HTTP method we want to use and the URI. As we did with all our HttpRequestMessage examples, we add an accept header, content, and the content type for our request. Finally, we send the request using the SendAsync method and ensure a successful status code in the response.

To be able to execute this method, we have to call it in the Execute method:

public async Task Execute**()**

**{**

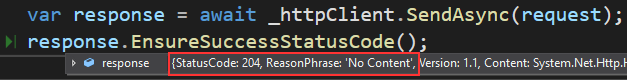
//await PatchEmployee();

await PatchEmployeeWithHttpRequestMessage**()**;

**}**

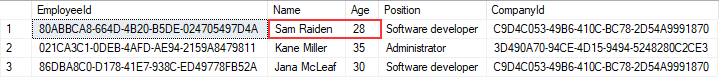
That’s all.

We can place a breakpoint in our method and run both applications:

**[](https://code-maze.com/wp-content/uploads/2021/01/18-Http-Patch-response-for-HttpRequestMessage-example.png)**

And we can see the 204 response.

We can confirm that by looking at the record in the Employees table:

**[](https://code-maze.com/wp-content/uploads/2021/01/19-Database-result-from-the-Patch-request.png)**

Excellent. Both columns have been updated.

**Conclusion**

Great. With this article – including the previous ones, we have covered all the CRUD requests including the HTTP PATCH request. Now we know how to use HttpClient to send all these types of requests using both shortcut methods and the HttpRequestMessage class.

**Using Streams with HttpClient to Improve Performance and Memory Usage**

Up until now, we were using strings to create a request body and also to read the content of the response. But we can optimize our application by improving performance and memory usage with streams. So, in this article, we are going to learn how to use streams with HttpClient while sending requests and reading the content from responses. We are going to use streams with only GET and POST requests because the logic from the POST request can be applied to PUT, and PATCH.

To download a source code, you can visit our **[Using Streams with HttpClient](https://github.com/CodeMazeBlog/httpclient-aspnetcore/tree/using-streams-with-httpclient" \t "_blank)** repository.

You can also visit our **[HttpClient Tutorial page](https://code-maze.com/httpclient-with-asp-net-core-tutorial/" \t "_blank)**, to see all the articles from this tutorial.

We are going to divide this article into the following sections:

* [**More About Streams**](https://code-maze.com/using-streams-with-httpclient-to-improve-performance-and-memory-usage/#more-about-streams)
* [**Using Streams with HttpClient to Fetch the Data**](https://code-maze.com/using-streams-with-httpclient-to-improve-performance-and-memory-usage/#streams-fetch-data)
* [**Additional Improvements with HttpCompletionMode**](https://code-maze.com/using-streams-with-httpclient-to-improve-performance-and-memory-usage/#httpcompletionmode)
* [**Sending a POST Request Using Streams with HttpClient**](https://code-maze.com/using-streams-with-httpclient-to-improve-performance-and-memory-usage/#post-request-streams)
* [**Conclusion**](https://code-maze.com/using-streams-with-httpclient-to-improve-performance-and-memory-usage/#conclusion)

So, let’s start.

**More About Streams**

The stream represents an abstraction of a sequence of bytes in the form of files, input/output devices, or network traffic. The Stream class in C# is an abstract class that provides methods to transfer bytes – read from or write to the source. Since we can read from or write to a stream, this enables us to skip creating variables in the middle (for the request body or response content) that can increase memory usage or decrease performance.

The vital thing to know here is that working with streams on the client side doesn’t have to do anything with the API level. This is totally a separate process. Our API may or may not work with streams but this doesn’t affect the client side. In the client application, we can use streams to prepare a request body or to read from a response regardless of the API implementation. This is an advantage for sure since we can use streams in the client apps to increase performance and decrease memory usage and still consume any API.

**Using Streams with HttpClient to Fetch the Data**

In the**[first article of this series](https://code-maze.com/fetching-data-with-httpclient-in-aspnetcore" \t "_blank)**, we have learned that while fetching the data from the API, we have to:

* Send a request to the API’s URI
* Wait for the response to arrive
* Read the content from the response body with the ReadAsStringAsync method
* And deserialize the content using System.Text.Json

As we said, with streams, we can remove that action in the middle where we use the ReadAsStringAsync method to read the string content from the response body.

So, let’s see how to do that.

First of all, we are going to create a new HttpClientStreamService in the client application:

public class HttpClientStreamService : IHttpClientServiceImplementation

**{**

private static readonly HttpClient \_httpClient = new HttpClient**()**;

private readonly JsonSerializerOptions \_options;

public HttpClientStreamService**()**

**{**

\_httpClient.BaseAddress = new Uri**(**"https://localhost:5001/api/"**)**;

\_httpClient.Timeout = new TimeSpan**(**0, 0, 30**)**;

\_httpClient.DefaultRequestHeaders.Clear**()**;

\_options = new JsonSerializerOptions **{** PropertyNameCaseInsensitive = true **}**;

**}**

public async Task Execute**()**

**{**

throw new NotImplementedException**()**;

**}**

**}**

This is a standard configuration that we’ve seen a couple of times in this series.

Next, we can create a method to send a GET request using streams:

private async Task GetCompaniesWithStream**()**

**{**

using **(**var response = await \_httpClient.GetAsync**(**"companies"**))**

**{**

response.EnsureSuccessStatusCode**()**;

var stream = await response.Content.ReadAsStreamAsync**()**;

var companies = await JsonSerializer.DeserializeAsync**<**List**<**CompanyDto**>>(**stream, \_options**)**;

**}**

**}**

In this method, we use the GetAsync shortcut method to fetch the data from the API. But as we have explained in the **[first article of this series](https://code-maze.com/fetching-data-with-httpclient-in-aspnetcore" \t "_blank)**, you can use the HttpRequestMessage class to do the same thing with higher control over the request. Also, pay attention that this time we are wrapping our response inside the using directive since we are working with streams now.

After we ensure the successful status code, we use the ReadAsStreamAsync method to serialize the HTTP content and return it as a stream. With this in place, we remove the need for string serialization and crating a string variable.

As soon as we have our stream, we call the JsonSerializer.DeserializeAsync method to read from a stream and deserialize the result into the list of company objects.

Before we start our apps, we have to call this method in the Execute method:

public async Task Execute**()**

**{**

await GetCompaniesWithStream**()**;

**}**

And also, register this new service in the Program class:

private static void ConfigureServices**(**IServiceCollection services**)**

**{**

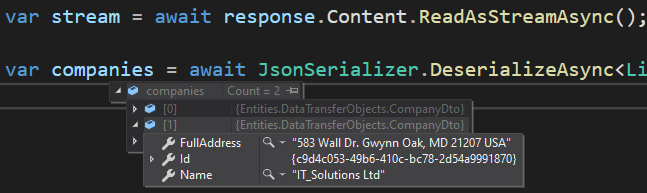
//services.AddScoped<IHttpClientServiceImplementation, HttpClientCrudService>();

//services.AddScoped<IHttpClientServiceImplementation, HttpClientPatchService>();

services.AddScoped**<**IHttpClientServiceImplementation, HttpClientStreamService**>()**;

**}**

That’s it. We can start both apps and inspect the result:

**[](https://code-maze.com/wp-content/uploads/2021/01/20-Result-of-using-streams-with-HttpClient-for-fetching-the-data-from-API.png)**

We can see, we have our result read from a stream.

**Additional Improvements with HttpCompletionMode**

In the previous example, we removed a string creation action when we read the content from the response. And as such, we made an improvement. But, we can improve the solution even more by using HttpCompletionMode. It is an enumeration having two values that control at what point the HttpClient’s actions are considered completed.

The default value is HttpCompletionMode.ResponseContentRead. It means that the HTTP operation is complete only when the entire response is read together with content. This is the case with our previous example.

The second value is HttpCompletionMode.ResponseHeadersRead. When we choose this option in our HTTP request, we state that the operation is complete when the response headers are fully read. At this point, the response body doesn’t have to be fully processed at all. This obviously means that we are going to use less memory because we don’t have to keep an entire content inside the memory. Also, this affects performance since we can work with the data faster.

To implement this improvement, all we have to do is to modify the GetAsync method in the GetCompaniesWithStream method:

private async Task GetCompaniesWithStream**()**

**{**

using **(**var response = await \_httpClient.GetAsync**(**"companies", HttpCompletionOption.ResponseHeadersRead**))**

**{**

response.EnsureSuccessStatusCode**()**;

var stream = await response.Content.ReadAsStreamAsync**()**;

var companies = await JsonSerializer.DeserializeAsync**<**List**<**CompanyDto**>>(**stream, \_options**)**;

**}**

**}**

And that’s all it takes.

If we run our application, we will see the same result as we had in a previous example. But this time, we improve our request even more.

Steve Gordon has **[an excellent article on this topic](https://www.stevejgordon.co.uk/using-httpcompletionoption-responseheadersread-to-improve-httpclient-performance-dotnet" \t "_blank)**, so feel free to check it out to see the benchmark results as well.

Now, let’s see how to use streams with a POST request.

**Sending a POST Request Using Streams with HttpClient**

In our **[second article of the series](https://code-maze.com/httpclient-example-aspnet-core-post-put-delete" \t "_blank)**, we have learned how to send a POST request using HttpClient. In that example, we were serializing our payload into a JSON string before we send the request. Of course, with streams, we can skip that part. Let’s see how.

First, let’s create a new method:

private async Task CreateCompanyWithStream**()**

**{**

var companyForCreation = new CompanyForCreationDto

**{**

Name = "Eagle IT Ltd.",

Country = "USA",

Address = "Eagle IT Street 289"

**}**;

var ms = new MemoryStream**()**;

await JsonSerializer.SerializeAsync**(**ms, companyForCreation**)**;

ms.Seek**(**0, SeekOrigin.Begin**)**;

var request = new HttpRequestMessage**(**HttpMethod.Post, "companies"**)**;

request.Headers.Accept.Add**(**new MediaTypeWithQualityHeaderValue**(**"application/json"**))**;

using **(**var requestContent = new StreamContent**(**ms**))**

**{**

request.Content = requestContent;

requestContent.Headers.ContentType = new MediaTypeHeaderValue**(**"application/json"**)**;

using **(**var response = await \_httpClient.SendAsync**(**request, HttpCompletionOption.ResponseHeadersRead**))**

**{**

response.EnsureSuccessStatusCode**()**;

var content = await response.Content.ReadAsStreamAsync**()**;

var createdCompany = await JsonSerializer.DeserializeAsync**<**CompanyDto**>(**content, \_options**)**;

**}**

**}**

**}**

In this method, we start by creating a new companyForCreation object with all the required properties. Then, we need a memory stream object. With the JsonSerializer.SerializeAsync method, we serialize our companyForCreation object into the created memory stream. Also, we use the Seek method to set a position at the beginning of the stream. Then, we initialize a new instance of the HttpReqestMessage object with the required arguments and set the accept header to application/json.

After that, we create a new stream content object named requestContent using the previously created memory stream. The StreamContent object is going to be the content of our request so, we state that in the code, and we set up the ContentType of our request.

Finally, we send our request using the SendAsync method and providing the HttpCompletionOption argument, ensure that the response is successful, and read our content as a stream. After reading the content, we just deserialize it into the createdCompany object.

So, as you can see, through the entire method, we work with streams avoiding unnecessary memory usage with large strings. Also, we are using the ResponseHeadersRead completion option because it makes sense while working with streams.

All we have to do now is to call this method in the Execute method:

public async Task Execute**()**

**{**

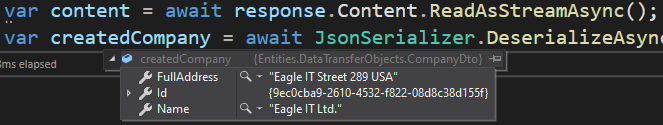
//await GetCompaniesWithStream();

await CreateCompanyWithStream**()**;

**}**

Great.

Let’s run the app and inspect the result:

**[](https://code-maze.com/wp-content/uploads/2021/01/21-Using-Post-request-with-streams.png)**

There we go. We have created a new company.

**Conclusion**

Using streams with HTTP requests can help us reduce memory consumption and optimize the performance of our app. In this article, we have seen how to use streams to fetch data from the server and also to create a StreamContent for our request body while sending a POST request. Additionally, we’ve learned more about completion options and how this can help us in achieving better optimization for our application.

**Canceling HTTP Requests in ASP.NET Core with CancellationToken**

It is quite a possible situation to have a user navigating to the client application’s page that sends an HTTP request to the server. While our app processing the request, a user can navigate away from that page. In such a case, we want to cancel the HTTP request since the response is no longer important to that user. Of course, this is just one of many situations that could happen in a real-world application where we would want to cancel our request. So, in this article, we are going to learn how to use CancellationToken to cancel HTTP requests in our client application.

To download a source code, you can visit our **[CancellationToken with HttpClient](https://github.com/CodeMazeBlog/httpclient-aspnetcore/tree/cancellation-tokens-with-httpclient" \t "_blank)** repository.

You can also visit our **[HttpClient Tutorial page](https://code-maze.com/httpclient-with-asp-net-core-tutorial/" \t "_blank)**, to see all the articles from this tutorial.

We are going to divide this article into the following sections:

* [**Using CancellationToken to Cancel Requests Sent with HttpClient**](https://code-maze.com/canceling-http-requests-in-asp-net-core-with-cancellationtoken/#cancellation-token)
* [**Improving the Solution by Sharing the CancellationToken**](https://code-maze.com/canceling-http-requests-in-asp-net-core-with-cancellationtoken/#sharing-cancellation-token)
* [**Handling TaskCanceledException**](https://code-maze.com/canceling-http-requests-in-asp-net-core-with-cancellationtoken/#handling-exceptions)
* [**Conclusion**](https://code-maze.com/canceling-http-requests-in-asp-net-core-with-cancellationtoken/#conclusion)

So, let’s dive right into it.

**Using CancellationToken to Cancel Requests Sent with HttpClient**

In the introduction, we stated that if a user navigates away from a page, they need the response no more, and thus it is a good practice to cancel that request. But there is more than that. HttpClient is working with async Tasks, therefore canceling a task that is no longer needed will set free the thread that we use to run the task. This means that the thread is going to be returned to a thread pool where this thread can be used for some other work. This will improve the scalability of our application for sure.

Of course, we can’t cancel the request just like that. To execute such an action we have to use CancellationTokenSource and CancellationToken.

We use CancellationTokenSource to create CancellationToken and to notify all the consumers of the CancellationToken that the request has been canceled. In our case, the HttpClient will consume the CancellationToken and listen for the notifications. As soon as the request cancelation notification is received, we are going to cancel that request using the HttpClient.

So, let’s see how to do that.

**Implementing CancellationToken Logic with HttpClient**

The first thing we are going to do is to create a new service for this example:

public class HttpClientCancellationService : IHttpClientServiceImplementation

**{**

private static readonly HttpClient \_httpClient = new HttpClient**()**;

private readonly JsonSerializerOptions \_options;

public HttpClientCancellationService**()**

**{**

\_httpClient.BaseAddress = new Uri**(**"https://localhost:5001/api/"**)**;

\_httpClient.Timeout = new TimeSpan**(**0, 0, 30**)**;

\_httpClient.DefaultRequestHeaders.Clear**()**;

\_options = new JsonSerializerOptions **{** PropertyNameCaseInsensitive = true **}**;

**}**

public async Task Execute**()**

**{**

throw new NotImplementedException**()**;

**}**

**}**

So, this is a familiar code that we have in our previous services as well. We create an HttpClient instance and provide a configuration for it. Also, we do the same with the JSON serialization options. In the next article, we are going to learn about the HttpClientFactory and see how we can move this configuration to a single place without repeating it in all the files, and also learn how to solve problems that HttpClient can cause. For now, we are going to leave it as-is.

Now, let’s add a new method to fetch all the companies:

private async Task GetCompaniesAndCancel**()**

**{**

using **(**var response = await \_httpClient.GetAsync**(**"companies", HttpCompletionOption.ResponseHeadersRead**))**

**{**

response.EnsureSuccessStatusCode**()**;

var stream = await response.Content.ReadAsStreamAsync**()**;

var companies = await JsonSerializer.DeserializeAsync**<**List**<**CompanyDto**>>(**stream, \_options**)**;

**}**

**}**

This is also a familiar code from **[the previous article](https://code-maze.com/using-streams-with-httpclient-to-improve-performance-and-memory-usage" \t "_blank)** and we won’t be explaining it here. If you are not familiar with streams, you can always read the linked article.

Now, let’s assume we want to cancel this request. As we already said, to cancel a request we need the CancellationTokenSource. So, let’s implement it:

private async Task GetCompaniesAndCancel**()**

**{**

var cancellationTokenSource = new CancellationTokenSource**()**;

cancellationTokenSource.CancelAfter**(**2000**)**;

using **(**var response = await \_httpClient.GetAsync**(**"companies",

HttpCompletionOption.ResponseHeadersRead, cancellationTokenSource.Token**))**

**{**

response.EnsureSuccessStatusCode**()**;

var stream = await response.Content.ReadAsStreamAsync**()**;

var companies = await JsonSerializer.DeserializeAsync**<**List**<**CompanyDto**>>(**stream, \_options**)**;

**}**

**}**

Here, we create a new cancellationTokenSource object. For this to work, we have to include the using System.Threading.Tasks namespace. After we create the object, we want to cancel the request. This is usually executed by the user – by pressing the cancel button or navigating away from a page, but for the example purpose, we are going to do it here. To cancel a request, we can use two methods: Cancel(), which cancels the request immediately, and CancelAfter(). For this example, we use the CancelAfter method and provide two seconds as an argument. Finally, we have to notify the HttpClient about the cancellation action. To do that, we provide a cancellation token as an additional argument for the GetAsync shortcut method.

That’s it. We can test it now.

**Testing Cancelling the Request**

Before we start our applications, we need to make sure that our method gets called when the app starts. To do that, we have to modify the Execute method:

public async Task Execute**()**

**{**

await GetCompaniesAndCancel**()**;

**}**

Also, we have to register this service in the Program class:

private static void ConfigureServices**(**IServiceCollection services**)**

**{**

//services.AddScoped<IHttpClientServiceImplementation, HttpClientCrudService>();

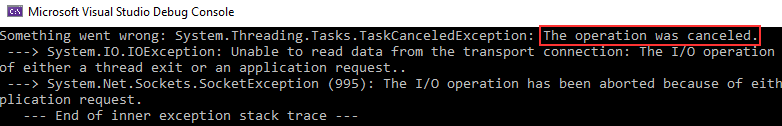
//services.AddScoped<IHttpClientServiceImplementation, HttpClientPatchService>();

//services.AddScoped<IHttpClientServiceImplementation, HttpClientStreamService>();

services.AddScoped**<**IHttpClientServiceImplementation, HttpClientCancellationService**>()**;

**}**

Now, let’s start both applications:

**[](https://code-maze.com/wp-content/uploads/2021/02/22-CancellationToken-implementation-to-cancel-the-GET-request.png)**

And, we can see that our request was canceled. If you want to test it again, make sure to restart the API as well, so it can simulate the longer first request.

**Improving the Solution by Sharing the CancellationToken**

The implementation as-is works great for our learning example. But in a real-world application, we would like to be able to cancel different requests by passing the token to all of them. This would enable canceling all of these requests if we need to. Also, we would like to be able to access this CancellationTokenSource from different parts of the application, for example when the user clicks the cancel button or navigates away from the page. In that case, we don’t want to hide the CancellationTokenSource inside a single method.

That said, let’s add some modifications to our service:

private static readonly HttpClient \_httpClient = new HttpClient**()**;

private readonly JsonSerializerOptions \_options;

private readonly CancellationTokenSource \_cancellationTokenSource;

public HttpClientCancellationService**()**

**{**

\_httpClient.BaseAddress = new Uri**(**"https://localhost:5001/api/"**)**;

\_httpClient.Timeout = new TimeSpan**(**0, 0, 30**)**;

\_httpClient.DefaultRequestHeaders.Clear**()**;

\_options = new JsonSerializerOptions **{** PropertyNameCaseInsensitive = true **}**;

\_cancellationTokenSource = new CancellationTokenSource**()**;

**}**

Here, we create a CancellationTokenSource readonly variable and instantiate it in the constructor.

Then, we want to modify the Execute method:

public async Task Execute**()**

**{**

\_cancellationTokenSource.CancelAfter**(**2000**)**;

await GetCompaniesAndCancel**(**\_cancellationTokenSource.Token**)**;

**}**

In this method, we call the CancelAfter method to specify the period after which we want to cancel our request and also pass the token to the GetCompaniesAndCancel method.

Of course, we have to modify the GetCompaniesAndCancel method as well:

private async Task GetCompaniesAndCancel**(**CancellationToken token**)**

**{**

using **(**var response = await \_httpClient.GetAsync**(**"companies",

HttpCompletionOption.ResponseHeadersRead, token**))**

**{**

response.EnsureSuccessStatusCode**()**;

var stream = await response.Content.ReadAsStreamAsync**()**;

var companies = await JsonSerializer.DeserializeAsync**<**List**<**CompanyDto**>>(**stream, \_options**)**;

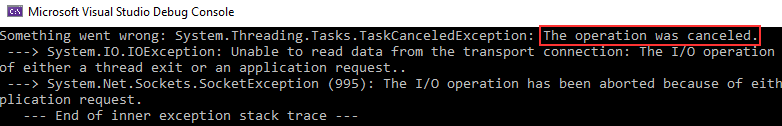
**}**

**}**

As you can see, at this point, our method accepts the token and use it to listen for the cancel notifications.

Now, we can start again our API and also the client app.

As soon as we do that, we are going to see the same exception:

**[](https://code-maze.com/wp-content/uploads/2021/02/22-CancellationToken-implementation-to-cancel-the-GET-request.png)**

That’s good.

We can continue to see how to handle this exception in our application.

**Handling TaskCanceledException**

If we want to handle the exception that our application throws after canceling a request, all we have to do is to wrap our request inside the try-catch block:

private async Task GetCompaniesAndCancel**(**CancellationToken token**)**

**{**

try

**{**

using **(**var response = await \_httpClient.GetAsync**(**"companies",

HttpCompletionOption.ResponseHeadersRead, token**))**

**{**

response.EnsureSuccessStatusCode**()**;

var stream = await response.Content.ReadAsStreamAsync**()**;

var companies = await JsonSerializer.DeserializeAsync**<**List**<**CompanyDto**>>(**stream, \_options**)**;

**}**

**}**

catch **(**OperationCanceledException ocex**)**

**{**

Console.WriteLine**(**ocex.Message**)**;

**}**

**}**

We saw that the app throws the TaskCanceledException but since it inherits from the OperationCanceledException class, we can use that class to catch our exception. Of course in the catch block, we can do a lot of actions but for this example, it is enough to just log the message.

Now, let’s start both applications and inspect the result:

**[Canceled operation message from the catch block](https://code-maze.com/wp-content/uploads/2021/02/23-Canceled-operation-message-from-the-catch-block.png)**

Great.

**Inspecting Status Codes from Responses**

With the implementation as we have it right now if the response is not successful, we are going to throw an exception. Well, to be 100% accurate the EnsureSuccessStatusCode() method will do it. But in many cases, we want to ensure a more user-friendly message depending on the real reason behind the response failure. Well, for that, we can check the status codes of our response.

We are not going to cover all the status codes here, for that you can visit our [**HTTP Reference table.**](https://code-maze.com/the-http-reference/) That said, here we are going to use one of the status codes and show how to provide a better user experience with more meaningful messages.

For this example, we are going to use the HttpClientStreamService class. So, let’s create a new method in that class:

private async Task GetNonExistentCompany**()**

**{**

var uri = Path.Combine**(**"companies", "F8088E81-7EFA-4E49-F824-08D8C38D155C"**)**;

using **(**var response = await \_httpClient.GetAsync**(**uri, HttpCompletionOption.ResponseHeadersRead**))**

**{**

response.EnsureSuccessStatusCode**()**;

var stream = await response.Content.ReadAsStreamAsync**()**;

var companies = await JsonSerializer.DeserializeAsync**<**List**<**CompanyDto**>>(**stream, \_options**)**;

**}**

**}**

This entire code is now familiar to us, but we just want to mention that the provided Id (a Guid value) doesn’t exist in our database. So, our API should return not found result (404).

Before we test it, we have to modify the Execute method:

public async Task Execute**()**

**{**

//await GetCompaniesWithStream();

//await CreateCompanyWithStream();

await GetNonExistentCompany**()**;

**}**

And also, we have to enable this service in the Program class:

private static void ConfigureServices**(**IServiceCollection services**)**

**{**

//services.AddScoped<IHttpClientServiceImplementation, HttpClientCrudService>();

//services.AddScoped<IHttpClientServiceImplementation, HttpClientPatchService>();

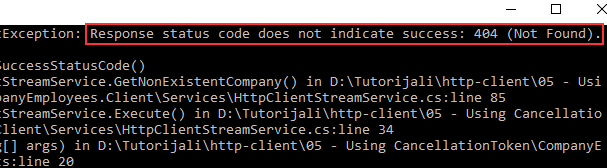
services.AddScoped**<**IHttpClientServiceImplementation, HttpClientStreamService**>()**;

//services.AddScoped<IHttpClientServiceImplementation, HttpClientCancellationService>();

**}**

Great.

Let’s start both applications and inspect the result:

**[](https://code-maze.com/wp-content/uploads/2021/02/24-Not-Found-response-with-an-exception..png)**

As you can see, we did get the 404 response but we still throw an exception. Well, we can change that.

**Working with Status Codes**

Let’s add a small modification to our method:

private async Task GetNonExistentCompany**()**

**{**

var uri = Path.Combine**(**"companies", "F8088E81-7EFA-4E49-F824-08D8C38D155C"**)**;

using **(**var response = await \_httpClient.GetAsync**(**uri, HttpCompletionOption.ResponseHeadersRead**))**

**{**

if**(**!response.IsSuccessStatusCode**)**

**{**

if **(**response.StatusCode.Equals**(**HttpStatusCode.NotFound**))**

**{**

Console.WriteLine**(**"The company you are searching for couldn't be found."**)**;

return;

**}**

response.EnsureSuccessStatusCode**()**;

**}**

var stream = await response.Content.ReadAsStreamAsync**()**;

var companies = await JsonSerializer.DeserializeAsync**<**List**<**CompanyDto**>>(**stream, \_options**)**;

**}**

**}**

We first check if the response doesn’t contain a successful status code with the IsSuccessStatusCode property. If it doesn’t, we explicitly check for the status code we want to handle, in this case, a NotFound status code. In that case, we just write an informative message to a console window. For all the other unsuccessful status codes, we throw an exception with the EnsureSuccessStatusCode method.

Of course, you can always expand this conditioning with other status codes as well, but in that case, it would be better to extract that logic into another method to make this method more readable.

Now, if we start our applications:

**[Better not found message](https://code-maze.com/wp-content/uploads/2021/02/25-Better-not-found-message.png)**

We can see our message on the screen.

Excellent.

Everything works as expected.

**Conclusion**

There we go.

Now, we know how to cancel our request using the CancellationToken and CancellationTokenSource and also how to use CancellationTokenSource to share the token between different requests. Furthermore, we know how to use different status codes from our response to prevent throwing exceptions for each unsuccessful response.

**Using HttpClientFactory in ASP.NET Core Applications**

Up until now, we’ve been using HttpClient directly in our services. In every service, we’ve created an HttpClient instance and all the required configurations. That lead to repeating code in all of our service classes. Well, in this article we are going to learn how to prevent that by using HttpClientFactory. Of course, this is not the only advantage of using HttpClientFactory. We are going to learn how HttpClientFactory prevents additional problems that HttpClient can cause. Additionally, we are going to show you how to create named and typed clients using HttpClientFactory.

To download a source code, you can visit our **[HttpClientFactory](https://github.com/CodeMazeBlog/httpclient-aspnetcore/tree/httpclientfactory-aspnetcore" \t "_blank)** repository.

You can also visit our **[HttpClient Tutorial page](https://code-maze.com/httpclient-with-asp-net-core-tutorial/" \t "_blank)**, to see all the articles from this tutorial.

We are going to divide this article into the following sections:

* [**Problems with HttpClient**](https://code-maze.com/using-httpclientfactory-in-asp-net-core-applications/#problems-with-httpclient)
* [**Adding HttpClientFactory in Our ASP.NET Core Application**](https://code-maze.com/using-httpclientfactory-in-asp-net-core-applications/#adding-http-client-factory)
* [**Using Named HttpClient Instances**](https://code-maze.com/using-httpclientfactory-in-asp-net-core-applications/#named-instances)
* [**Using Typed HttpClient Instances**](https://code-maze.com/using-httpclientfactory-in-asp-net-core-applications/#typed-instances)
* [**Encapsulation of the Logic Related to a Typed Client**](https://code-maze.com/using-httpclientfactory-in-asp-net-core-applications/#encapsulation)
* [**Conclusion**](https://code-maze.com/using-httpclientfactory-in-asp-net-core-applications/#conclusion)

Let’s dive right into it.

**Problems with HttpClient**

The HttpClient class implements the IDisposable interface. By seeing that, we can all be tempted to try using our HttpClient instance inside the using directive, thus disposing of it once it is out of the scope. But, this is not a good practice. If we dispose of the HttpClient, we are going to dispose of the underlying HttpClientHandler as well. Now, this means that for every new request we have to create a new HttpClient instance and thus the handler as well. Of course, that’s the problem. Reopening connections could lead to slow performance because these connections and HttpClientHandlers are pretty expensive while working with HttpClient.

Also, there is another problem. By creating too many connections, we can face socket exhaustion because we use too many sockets too fast, and we don’t have any available socket to create a new connection.

So, with all these in mind, we can conclude that we shouldn’t dispose of our HttpClient but share it throughout the requests. That’s what we’ve been doing in our previous articles with the static HttpClient instance. This also allows reusing of the underlying connections.

But, we have to pay attention, that using a static instance is not the ultimate solution. When we reuse our instance, we also reuse the connection until the socket is closed. Thus connections won’t get the update from DNS (for example switching between different environments). If our connection is not aware of switching from staging to the production environment, our requests would not go to the right environment as well.

To help us solve these problems, we can use HttpClientFactory to create HttpClient instances.

**How HttpClientFactory Helps Us Solving Mentioned Problems?**

Not only that HttpClientFactory can create and manage new HttpClient instances but also, it works with underlying handlers. Basically, when creating new HttpClient instances, it doesn’t recreate a new message handler but it takes one from a pool. Then, it uses that message handler to send the requests to the API. The default lifetime of a handler is set to two minutes, and during that time, any request for a new HttpClient can reuse an existing message handler and the connection as well. This means we don’t have to create a new message handler for every request and also we don’t have to open a new connection, thus preventing the socket exhausting issue.

Furthermore, since the handler’s lifetime is set to two minutes, after that time, HttpClientFactory uses a new message handler. Well, with that, we solve the DNS issues. When we use a new message handler, we take all the DNS changes into account.

In addition to solving these problems, with HttpClientHandler, we can centralize our HttpClient’s configuration. If you read our previous articles from this series, you saw that we had to repeat the same configuration in each service class. Well, with HttpClientHandler, we can prevent that.

That said, let’s see how we can use HttpClientFactory.

**Adding HttpClientFactory in Our ASP.NET Core Application**

To be able to use HttpClientFactory in our app, we have to install the Microsoft.Extensions.Http library in our client application:

Install-Package Microsoft.Extensions.Http -Version 5.0.0

Then, we have to add the IHttpClientFactory and other services to the service collection by using the AddHttpClient method in the Program class:

private static void ConfigureServices**(**IServiceCollection services**)**

**{**

services.AddHttpClient**()**;

//services.AddScoped<IHttpClientServiceImplementation, HttpClientCrudService>();

//services.AddScoped<IHttpClientServiceImplementation, HttpClientPatchService>();

//services.AddScoped<IHttpClientServiceImplementation, HttpClientStreamService>();

//services.AddScoped<IHttpClientServiceImplementation, HttpClientCancellationService>();

**}**

For now, this is enough. We are going to expand this method with additional configuration soon enough.

Now, let’s create a new service class as we did in our previous articles:

public class HttpClientFactoryService : IHttpClientServiceImplementation

**{**

private readonly IHttpClientFactory \_httpClientFactory;

private readonly JsonSerializerOptions \_options;

public HttpClientFactoryService**(**IHttpClientFactory httpClientFactory**)**

**{**

\_httpClientFactory = httpClientFactory;

\_options = new JsonSerializerOptions **{** PropertyNameCaseInsensitive = true **}**;

**}**

public async Task Execute**()**

**{**

throw new NotImplementedException**()**;

**}**

**}**

To be able to use HttpClientFactory in our HttpClientFactoryService class, we have to inject it with **[Dependency Injection](https://code-maze.com/dependency-injection-aspnet/" \t "_blank)**. And that’s exactly what we do. Additionally, we configure options for our JSON serialization. Since we don’t want to add the cancellation logic here, we are not using the CancellationTokenSource as we did in our **[previous article](https://code-maze.com/canceling-http-requests-in-asp-net-core-with-cancellationtoken" \t "_blank)**. Of course, if you want, you can add it here without a problem.

Now, let’s add a new method to fetch companies from the API:

private async Task GetCompaniesWithHttpClientFactory**()**

**{**

var httpClient = \_httpClientFactory.CreateClient**()**;

using **(**var response = await httpClient.GetAsync**(**"https://localhost:5001/api/companies", HttpCompletionOption.ResponseHeadersRead**))**

**{**

response.EnsureSuccessStatusCode**()**;

var stream = await response.Content.ReadAsStreamAsync**()**;

var companies = await JsonSerializer.DeserializeAsync**<**List**<**CompanyDto**>>(**stream, \_options**)**;

**}**

**}**

In this code, the only thing we are not familiar with is the part where we use the CreateClient method from HttpClientFactory to create a new HttpClient using the default configuration. Everything else is already explained in our previous articles from **[this series](https://code-maze.com/httpclient-with-asp-net-core-tutorial/" \t "_blank)**. Also, since we didn’t provide our custom configuration for the factory, we have to use an entire URI for the companies endpoint in the GetAsync method.

After this, we can modify the Execute method:

public async Task Execute**()**

**{**

await GetCompaniesWithHttpClientFactory**()**;

**}**

And also, let’s register this service in the Program class:

private static void ConfigureServices**(**IServiceCollection services**)**

**{**

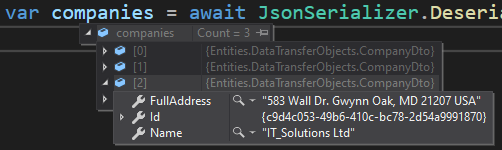
services.AddHttpClient**()**;

...

services.AddScoped**<**IHttpClientServiceImplementation, HttpClientFactoryService**>()**;

**}**

As soon as we do that, we can place the breakpoint in our new method and start both applications:

**[](https://code-maze.com/wp-content/uploads/2021/02/26-Using-HttpClientFactory-to-fetch-data-and-create-a-client.png)**

And there we go. We can see all three companies as a result. But, we can improve this solution by using Named and Typed HttpClient instances.

**Using Named HttpClient Instances**

In the Program class, we use the AddHttpClient method to register IHttpClientFactory without additional configuration. This means that every HttpClient instance we create with the CreateClient method will have the same configuration. But usually, that is not enough, since our client app often requires different HttpClient instances while communicating with a single or multiple APIs. Well, to support that, we can use the named HttpClient instances.

In our previous articles, we used the same configuration in each service to set up the base address, timeout, and to clear default request headers. Now, we can do the same, but only in one place:

private static void ConfigureServices**(**IServiceCollection services**)**

**{**

services.AddHttpClient**(**"CompaniesClient", config =**>**

**{**

config.BaseAddress = new Uri**(**"https://localhost:5001/api/"**)**;

config.Timeout = new TimeSpan**(**0, 0, 30**)**;

config.DefaultRequestHeaders.Clear**()**;

**})**;

...

services.AddScoped**<**IHttpClientServiceImplementation, HttpClientFactoryService**>()**;

**}**

With these modifications, the AddHttpClient method adds IHttpClientFactory to the service collection and also configures a named HttpClient instance. We provide a name for the instance and also a default configuration.

After this, we can modify the method in our new service:

private async Task GetCompaniesWithHttpClientFactory**()**

**{**

var httpClient = \_httpClientFactory.CreateClient**(**"CompaniesClient"**)**;

using **(**var response = await httpClient.GetAsync**(**"companies", HttpCompletionOption.ResponseHeadersRead**))**

**{**

response.EnsureSuccessStatusCode**()**;

var stream = await response.Content.ReadAsStreamAsync**()**;

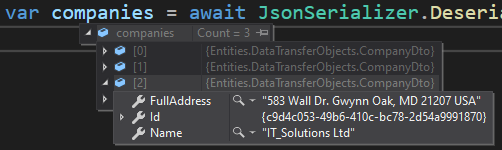
var companies = await JsonSerializer.DeserializeAsync**<**List**<**CompanyDto**>>(**stream, \_options**)**;

**}**

**}**

As you can see, we pass a name argument to the CreateClient method and also we don’t have to use a full URI in the GetAsync method. Since we are using the name of the client, the configuration that corresponds to this name applies as well.

Once we start both applications, we are going to get the same result as before:

**[](https://code-maze.com/wp-content/uploads/2021/02/26-Using-HttpClientFactory-to-fetch-data-and-create-a-client.png)**

Excellent.

Now, let’s see how to use a Typed client.

**Using Typed HttpClient Instances**

With the typed instances, we can achieve the same thing as with named instances, but we don’t have to use strings during the registration – we can use types.

Let’s start by creating a new Clients folder in a client application and a new CompaniesClient class inside that folder:

public class CompaniesClient

**{**

public HttpClient Client **{** get; **}**

public CompaniesClient**(**HttpClient client**)**

**{**

Client = client;

Client.BaseAddress = new Uri**(**"https://localhost:5001/api/"**)**;

Client.Timeout = new TimeSpan**(**0, 0, 30**)**;

Client.DefaultRequestHeaders.Clear**()**;

**}**

**}**

This is our typed client class with the default configuration, and we can register it in the Program class by calling AddHttpClient one more time in the ConfigureServices method:

services.AddHttpClient**<**CompaniesClient**>()**;

So, we are not using the name but the type of the client.

Now, in our HttpClientFactoryService, we have to inject our new client:

private readonly IHttpClientFactory \_httpClientFactory;

private readonly CompaniesClient \_companiesClient;

private readonly JsonSerializerOptions \_options;

public HttpClientFactoryService**(**IHttpClientFactory httpClientFactory, CompaniesClient companiesClient**)**

**{**

\_httpClientFactory = httpClientFactory;

\_companiesClient = companiesClient;

\_options = new JsonSerializerOptions **{** PropertyNameCaseInsensitive = true **}**;

**}**

Then, we are going to create a new method to make use of our typed client:

private async Task GetCompaniesWithTypedClient**()**

**{**

using **(**var response = await \_companiesClient.Client.GetAsync**(**"companies", HttpCompletionOption.ResponseHeadersRead**))**

**{**

response.EnsureSuccessStatusCode**()**;

var stream = await response.Content.ReadAsStreamAsync**()**;

var companies = await JsonSerializer.DeserializeAsync**<**List**<**CompanyDto**>>(**stream, \_options**)**;

**}**

**}**

As we can see, we are not creating a new client instance by using the CreateClient method. This time, we just use the injected typed client with its Client property.

Finally, let’s just execute this method:

public async Task Execute**()**

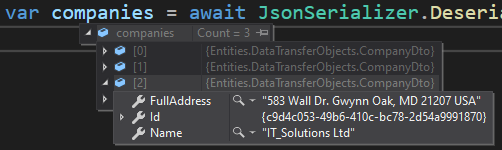
**{**

//await GetCompaniesWithHttpClientFactory();

await GetCompaniesWithTypedClient**()**;

**}**

Once we start both applications, we are going to get the same result as before:

**[](https://code-maze.com/wp-content/uploads/2021/02/26-Using-HttpClientFactory-to-fetch-data-and-create-a-client.png)**

This looks great.

Now let’s see how to extract the company related logic to the CompaniesClient class.

**Encapsulation of the Logic Related to a Typed Client**

Since we already have the typed client class, we can extract all the related logic from the service to this class. To do that, we are going to modify the CompaniesClient class:

public class CompaniesClient

**{**

private readonly HttpClient \_client;

private readonly JsonSerializerOptions \_options;

public CompaniesClient**(**HttpClient client**)**

**{**

\_client = client;

\_client.BaseAddress = new Uri**(**"https://localhost:5001/api/"**)**;

\_client.Timeout = new TimeSpan**(**0, 0, 30**)**;

\_client.DefaultRequestHeaders.Clear**()**;

\_options = new JsonSerializerOptions **{** PropertyNameCaseInsensitive = true **}**;

**}**

**}**

We don’t have the Client property anymore. Instead, we have a private readonly variable, which we are going to use in this class to execute the HttpClient’s logic. Additionally, we add the JsonSerializerOptions configuration.

Now, we can add a new method:

public async Task**<**List**<**CompanyDto**>>** GetCompanies**()**

**{**

using **(**var response = await \_client.GetAsync**(**"companies", HttpCompletionOption.ResponseHeadersRead**))**

**{**

response.EnsureSuccessStatusCode**()**;

var stream = await response.Content.ReadAsStreamAsync**()**;

var companies = await JsonSerializer.DeserializeAsync**<**List**<**CompanyDto**>>(**stream, \_options**)**;

return companies;

**}**

**}**

With this method, we are fetching the companies from the API and return a result.

Finally, we can modify the GetCompaniesWithTypedClientmethod in a service class:

private async Task GetCompaniesWithTypedClient**()** =**>** await \_companiesClient.GetCompanies**()**;

Excellent.

The logic related to the typed client is in a typed client’s class and our service is just calling that method.

You can test the execution, but you will get the same result as before.

**Conclusion**

To sum up, in this article, we have learned:

* What problems HttpClientFactory solves
* How to use HttpClientFactory in our application
* The way to use Named and Typed client instances
* And how to extract logic from a service to a client class