

Make HTTP requests using IHttpConnectionFactory in ASP.NET Core

Article • 04/11/2023

By [Kirk Larkin](#), [Steve Gordon](#), [Glenn Condrón](#), and [Ryan Nowak](#).

An [IHttpConnectionFactory](#) can be registered and used to configure and create [HttpClient](#) instances in an app. [IHttpConnectionFactory](#) offers the following benefits:

- Provides a central location for naming and configuring logical [HttpClient](#) instances. For example, a client named *github* could be registered and configured to access [GitHub](#). A default client can be registered for general access.
- Codifies the concept of outgoing middleware via delegating handlers in [HttpClient](#). Provides extensions for Polly-based middleware to take advantage of delegating handlers in [HttpClient](#).
- Manages the pooling and lifetime of underlying [HttpClientMessageHandler](#) instances. Automatic management avoids common DNS (Domain Name System) problems that occur when manually managing [HttpClient](#) lifetimes.
- Adds a configurable logging experience (via [ILogger](#)) for all requests sent through clients created by the factory.

The sample code in this topic version uses [System.Text.Json](#) to deserialize JSON content returned in HTTP responses. For samples that use [Json.NET](#) and [ReadAsStringAsync](#), use the version selector to select a 2.x version of this topic.

Consumption patterns

There are several ways [IHttpConnectionFactory](#) can be used in an app:

- [Basic usage](#)
- [Named clients](#)
- [Typed clients](#)
- [Generated clients](#)

The best approach depends upon the app's requirements.

Basic usage

Register [IHttpConnectionFactory](#) by calling [AddHttpClient](#) in [Program.cs](#):

```
C#  
  
var builder = WebApplication.CreateBuilder(args);  
  
// Add services to the container.  
builder.Services.AddHttpClient();
```

An [IHttpConnectionFactory](#) can be requested using [dependency injection \(DI\)](#). The following code uses [IHttpConnectionFactory](#) to create an [HttpClient](#) instance:

```
C#  
  
public class BasicModel : PageModel  
{  
    private readonly IHttpConnectionFactory _httpClientFactory;  
  
    public BasicModel(IHttpConnectionFactory httpClientFactory) =>  
        _httpClientFactory = httpClientFactory;  
  
    public IEnumerable<GitHubBranch>? GitHubBranches { get; set; }  
  
    public async Task OnGet()  
    {  
        var httpRequestMessage = new HttpRequestMessage(  
            HttpMethod.Get,  
            "https://api.github.com/repos/dotnet/AspNetCore.Docs/branches")  
        {  
            Headers =  
            {  
                { HeaderNames.Accept, "application/vnd.github.v3+json" },  
                { HeaderNames.UserAgent, "HttpRequestsSample" }  
            }  
        };  
    }  
};
```

```

var httpClient = _httpClientFactory.CreateClient();
var httpResponseMessage = await httpClient.SendAsync(httpRequestMessage);

if (httpResponseMessage.IsSuccessStatusCode)
{
    using var contentStream =
        await httpResponseMessage.Content.ReadAsStreamAsync();

    GitHubBranches = await JsonSerializer.DeserializeAsync<
        IEnumerable<GitHubBranch>>(contentStream);
}
}
}

```

Using `IHttpClientFactory` like in the preceding example is a good way to refactor an existing app. It has no impact on how `HttpClient` is used. In places where `HttpClient` instances are created in an existing app, replace those occurrences with calls to [CreateClient](#).

Named clients

Named clients are a good choice when:

- The app requires many distinct uses of `HttpClient`.
- Many `HttpClient`s have different configuration.

Specify configuration for a named `HttpClient` during its registration in `Program.cs`:

```

C#

builder.Services.AddHttpClient("GitHub", httpClient =>
{
    httpClient.BaseAddress = new Uri("https://api.github.com/");

    // using Microsoft.Net.Http.Headers;
    // The GitHub API requires two headers.
    httpClient.DefaultRequestHeaders.Add(
        HeaderNames.Accept, "application/vnd.github.v3+json");
    httpClient.DefaultRequestHeaders.Add(
        HeaderNames.UserAgent, "HttpRequestsSample");
});

```

In the preceding code the client is configured with:

- The base address `https://api.github.com/`.
- Two headers required to work with the GitHub API.

CreateClient

Each time [CreateClient](#) is called:

- A new instance of `HttpClient` is created.
- The configuration action is called.

To create a named client, pass its name into `CreateClient`:

```

C#

public class NamedClientModel : PageModel
{
    private readonly IHttpClientFactory _httpClientFactory;

    public NamedClientModel(IHttpClientFactory httpClientFactory) =>
        _httpClientFactory = httpClientFactory;

    public IEnumerable<GitHubBranch>? GitHubBranches { get; set; }

    public async Task OnGet()
    {
        var httpClient = _httpClientFactory.CreateClient("GitHub");
        var httpResponseMessage = await httpClient.GetAsync(
            "repos/dotnet/AspNetCore.Docs/branches");

        if (httpResponseMessage.IsSuccessStatusCode)

```

```

    {
        using var contentStream =
            await httpResponseMessage.Content.ReadAsStreamAsync();

        GitHubBranches = await JsonSerializer.DeserializeAsync<
            IEnumerable<GitHubBranch>>(contentStream);
    }
}

```

In the preceding code, the request doesn't need to specify a hostname. The code can pass just the path, since the base address configured for the client is used.

Typed clients

Typed clients:

- Provide the same capabilities as named clients without the need to use strings as keys.
- Provides IntelliSense and compiler help when consuming clients.
- Provide a single location to configure and interact with a particular `HttpClient`. For example, a single typed client might be used:
 - For a single backend endpoint.
 - To encapsulate all logic dealing with the endpoint.
- Work with DI and can be injected where required in the app.

A typed client accepts an `HttpClient` parameter in its constructor:

```

C#

public class GitHubService
{
    private readonly HttpClient _httpClient;

    public GitHubService(HttpClient httpClient)
    {
        _httpClient = httpClient;

        _httpClient.BaseAddress = new Uri("https://api.github.com/");

        // using Microsoft.Net.Http.Headers;
        // The GitHub API requires two headers.
        _httpClient.DefaultRequestHeaders.Add(
            HeaderNames.Accept, "application/vnd.github.v3+json");
        _httpClient.DefaultRequestHeaders.Add(
            HeaderNames.UserAgent, "HttpRequestsSample");
    }

    public async Task<IEnumerable<GitHubBranch>?> GetAspNetCoreDocsBranchesAsync() =>
        await _httpClient.GetFromJsonAsync<IEnumerable<GitHubBranch>>(
            "repos/dotnet/AspNetCore.Docs/branches");
    }
}

```

In the preceding code:

- The configuration is moved into the typed client.
- The provided `HttpClient` instance is stored as a private field.

API-specific methods can be created that expose `HttpClient` functionality. For example, the `GetAspNetCoreDocsBranches` method encapsulates code to retrieve docs GitHub branches.

The following code calls `AddHttpClient` in `Program.cs` to register the `GitHubService` typed client class:

```

C#

builder.Services.AddHttpClient<GitHubService>();

```

The typed client is registered as transient with DI. In the preceding code, `AddHttpClient` registers `GitHubService` as a transient service. This registration uses a factory method to:

1. Create an instance of `HttpClient`.
2. Create an instance of `GitHubService`, passing in the instance of `HttpClient` to its constructor.

The typed client can be injected and consumed directly:

```
C#

public class TypedClientModel : PageModel
{
    private readonly GitHubService _githubService;

    public TypedClientModel(GitHubService githubService) =>
        _githubService = githubService;

    public IEnumerable<GitHubBranch>? GitHubBranches { get; set; }

    public async Task OnGet()
    {
        try
        {
            GitHubBranches = await _githubService.GetAspNetCoreDocsBranchesAsync();
        }
        catch (HttpRequestException)
        {
            // ...
        }
    }
}
```

The configuration for a typed client can also be specified during its registration in `Program.cs`, rather than in the typed client's constructor:

```
C#

builder.Services.AddHttpClient<GitHubService>(httpClient =>
{
    httpClient.BaseAddress = new Uri("https://api.github.com/");

    // ...
});
```

Generated clients

`IHttpClientFactory` can be used in combination with third-party libraries such as [Refit](#). Refit is a REST library for .NET. It converts REST APIs into live interfaces. Call `AddRefitClient` to generate a dynamic implementation of an interface, which uses `HttpClient` to make the external HTTP calls.

A custom interface represents the external API:

```
C#

public interface IGitHubClient
{
    [Get("/repos/dotnet/AspNetCore.Docs/branches")]
    Task<IEnumerable<GitHubBranch>> GetAspNetCoreDocsBranchesAsync();
}
```

Call `AddRefitClient` to generate the dynamic implementation and then call `ConfigureHttpClient` to configure the underlying `HttpClient`:

```
C#

builder.Services.AddRefitClient<IGitHubClient>()
    .ConfigureHttpClient(httpClient =>
    {
        httpClient.BaseAddress = new Uri("https://api.github.com/");

        // using Microsoft.Net.Http.Headers;
        // The GitHub API requires two headers.
        httpClient.DefaultRequestHeaders.Add(
            HeaderNames.Accept, "application/vnd.github.v3+json");
        httpClient.DefaultRequestHeaders.Add(
            HeaderNames.UserAgent, "HttpRequestsSample");
    });
```

Use DI to access the dynamic implementation of `IGitHubClient`:

```
C#
```

```

public class RefitModel : PageModel
{
    private readonly IGitHubClient _gitHubClient;

    public RefitModel(IGitHubClient gitHubClient) =>
        _gitHubClient = gitHubClient;

    public IEnumerable<GitHubBranch>? GitHubBranches { get; set; }

    public async Task OnGet()
    {
        try
        {
            GitHubBranches = await _gitHubClient.GetAspNetCoreDocsBranchesAsync();
        }
        catch (ApiException)
        {
            // ...
        }
    }
}

```

Make POST, PUT, and DELETE requests

In the preceding examples, all HTTP requests use the GET HTTP verb. `HttpClient` also supports other HTTP verbs, including:

- POST
- PUT
- DELETE
- PATCH

For a complete list of supported HTTP verbs, see [HttpMethod](#).

The following example shows how to make an HTTP POST request:

```

C#

public async Task CreateItemAsync(TodoItem todoItem)
{
    var todoItemJson = new StringContent(
        JsonSerializer.Serialize(todoItem),
        Encoding.UTF8,
        Application.Json); // using static System.Net.Mime.MediaTypeNames;

    using var httpResponseMessage =
        await _httpClient.PostAsync("/api/TodoItems", todoItemJson);

    httpResponseMessage.EnsureSuccessStatusCode();
}

```

In the preceding code, the `CreateItemAsync` method:

- Serializes the `TodoItem` parameter to JSON using `System.Text.Json`.
- Creates an instance of `StringContent` to package the serialized JSON for sending in the HTTP request's body.
- Calls `PostAsync` to send the JSON content to the specified URL. This is a relative URL that gets added to the `HttpClient.BaseAddress`.
- Calls `EnsureSuccessStatusCode` to throw an exception if the response status code doesn't indicate success.

`HttpClient` also supports other types of content. For example, `MultipartContent` and `StreamContent`. For a complete list of supported content, see [HttpContent](#).

The following example shows an HTTP PUT request:

```

C#

public async Task SaveItemAsync(TodoItem todoItem)
{
    var todoItemJson = new StringContent(
        JsonSerializer.Serialize(todoItem),
        Encoding.UTF8,
        Application.Json);

    using var httpResponseMessage =

```

```

        await _httpClient.PutAsync($"api/TodoItems/{todoItem.Id}", todoItemJson);

        httpResponseMessage.EnsureSuccessStatusCode();
    }
}

```

The preceding code is similar to the POST example. The `SaveItemAsync` method calls `PutAsync` instead of `PostAsync`.

The following example shows an HTTP DELETE request:

```

C#

public async Task DeleteItemAsync(long itemId)
{
    using var httpResponseMessage =
        await _httpClient.DeleteAsync($"api/TodoItems/{itemId}");

    httpResponseMessage.EnsureSuccessStatusCode();
}

```

In the preceding code, the `DeleteItemAsync` method calls `DeleteAsync`. Because HTTP DELETE requests typically contain no body, the `DeleteAsync` method doesn't provide an overload that accepts an instance of `HttpContent`.

To learn more about using different HTTP verbs with `HttpClient`, see [HttpClient](#).

Outgoing request middleware

`HttpClient` has the concept of delegating handlers that can be linked together for outgoing HTTP requests. `IHttpClientFactory`:

- Simplifies defining the handlers to apply for each named client.
- Supports registration and chaining of multiple handlers to build an outgoing request middleware pipeline. Each of these handlers is able to perform work before and after the outgoing request. This pattern:
 - Is similar to the inbound middleware pipeline in ASP.NET Core.
 - Provides a mechanism to manage cross-cutting concerns around HTTP requests, such as:
 - caching
 - error handling
 - serialization
 - logging

To create a delegating handler:

- Derive from [DelegatingHandler](#).
- Override `SendAsync`. Execute code before passing the request to the next handler in the pipeline:

```

C#

public class ValidateHeaderHandler : DelegatingHandler
{
    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        if (!request.Headers.Contains("X-API-KEY"))
        {
            return new HttpResponseMessage(HttpStatusCode.BadRequest)
            {
                Content = new StringContent(
                    "The API key header X-API-KEY is required.")
            };
        }

        return await base.SendAsync(request, cancellationToken);
    }
}

```

The preceding code checks if the `X-API-KEY` header is in the request. If `X-API-KEY` is missing, `BadRequest` is returned.

More than one handler can be added to the configuration for an `HttpClient` with [Microsoft.Extensions.DependencyInjection.HttpClientBuilderExtensions.AddHttpMessageHandler](#):

```

C#

```

```
builder.Services.AddTransient<ValidateHeaderHandler>();

builder.Services.AddHttpClient("HttpMessageHandler")
    .AddHttpMessageHandler<ValidateHeaderHandler>();
```

In the preceding code, the `ValidateHeaderHandler` is registered with DI. Once registered, `AddHttpMessageHandler` can be called, passing in the type for the handler.

Multiple handlers can be registered in the order that they should execute. Each handler wraps the next handler until the final `HttpClientHandler` executes the request:

```
C#

builder.Services.AddTransient<SampleHandler1>();
builder.Services.AddTransient<SampleHandler2>();

builder.Services.AddHttpClient("MultipleHttpMessageHandlers")
    .AddHttpMessageHandler<SampleHandler1>()
    .AddHttpMessageHandler<SampleHandler2>();
```

In the preceding code, `SampleHandler1` runs first, before `SampleHandler2`.

Use DI in outgoing request middleware

When `IHttpClientFactory` creates a new delegating handler, it uses DI to fulfill the handler's constructor parameters. `IHttpClientFactory` creates a **separate** DI scope for each handler, which can lead to surprising behavior when a handler consumes a *scoped* service.

For example, consider the following interface and its implementation, which represents a task as an operation with an identifier, `OperationId`:

```
C#

public interface IOperationScoped
{
    string OperationId { get; }
}

public class OperationScoped : IOperationScoped
{
    public string OperationId { get; } = Guid.NewGuid().ToString()[^4..];
}
```

As its name suggests, `IOperationScoped` is registered with DI using a *scoped* lifetime:

```
C#

builder.Services.AddScoped<IOperationScoped, OperationScoped>();
```

The following delegating handler consumes and uses `IOperationScoped` to set the `X-OPERATION-ID` header for the outgoing request:

```
C#

public class OperationHandler : DelegatingHandler
{
    private readonly IOperationScoped _operationScoped;

    public OperationHandler(IOperationScoped operationScoped) =>
        _operationScoped = operationScoped;

    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        request.Headers.Add("X-OPERATION-ID", _operationScoped.OperationId);

        return await base.SendAsync(request, cancellationToken);
    }
}
```

In the [HttpRequestsSample download](#), navigate to `/operation` and refresh the page. The request scope value changes for each request, but the handler scope value only changes every 5 seconds.

Handlers can depend upon services of any scope. Services that handlers depend upon are disposed when the handler is disposed.

Use one of the following approaches to share per-request state with message handlers:

- Pass data into the handler using [HttpRequestMessage.Options](#).
- Use [IHttpContextAccessor](#) to access the current request.
- Create a custom [AsyncLocal<T>](#) storage object to pass the data.

Use Polly-based handlers

`IHttpClientFactory` integrates with the third-party library [Polly](#). Polly is a comprehensive resilience and transient fault-handling library for .NET. It allows developers to express policies such as Retry, Circuit Breaker, Timeout, Bulkhead Isolation, and Fallback in a fluent and thread-safe manner.

Extension methods are provided to enable the use of Polly policies with configured `HttpClient` instances. The Polly extensions support adding Polly-based handlers to clients. Polly requires the [Microsoft.Extensions.Http.Polly](#) NuGet package.

Handle transient faults

Faults typically occur when external HTTP calls are transient. [AddTransientHttpErrorPolicy](#) allows a policy to be defined to handle transient errors. Policies configured with `AddTransientHttpErrorPolicy` handle the following responses:

- [HttpRequestException](#)
- HTTP 5xx
- HTTP 408

`AddTransientHttpErrorPolicy` provides access to a `PolicyBuilder` object configured to handle errors representing a possible transient fault:

```
C#

builder.Services.AddHttpClient("PollyWaitAndRetry")
    .AddTransientHttpErrorPolicy(policyBuilder =>
        policyBuilder.WaitAndRetryAsync(
            3, retryNumber => TimeSpan.FromMilliseconds(600)));
```

In the preceding code, a `WaitAndRetryAsync` policy is defined. Failed requests are retried up to three times with a delay of 600 ms between attempts.

Dynamically select policies

Extension methods are provided to add Polly-based handlers, for example, [AddPolicyHandler](#). The following `AddPolicyHandler` overload inspects the request to decide which policy to apply:

```
C#

var timeoutPolicy = Policy.TimeoutAsync<HttpResponseMessage>(
    TimeSpan.FromSeconds(10));
var longTimeoutPolicy = Policy.TimeoutAsync<HttpResponseMessage>(
    TimeSpan.FromSeconds(30));

builder.Services.AddHttpClient("PollyDynamic")
    .AddPolicyHandler(httpRequestMessage =>
        httpRequestMessage.Method == HttpMethod.Get ? timeoutPolicy : longTimeoutPolicy);
```

In the preceding code, if the outgoing request is an HTTP GET, a 10-second timeout is applied. For any other HTTP method, a 30-second timeout is used.

Add multiple Polly handlers

It's common to nest Polly policies:

```
C#

builder.Services.AddHttpClient("PollyMultiple")
    .AddTransientHttpErrorPolicy(policyBuilder =>
        policyBuilder.RetryAsync(3))
    .AddTransientHttpErrorPolicy(policyBuilder =>
        policyBuilder.CircuitBreakerAsync(5, TimeSpan.FromSeconds(30)));
```


In the preceding example:

- Two handlers are added.
- The first handler uses [AddTransientHttpErrorPolicy](#) to add a retry policy. Failed requests are retried up to three times.
- The second [AddTransientHttpErrorPolicy](#) call adds a circuit breaker policy. Further external requests are blocked for 30 seconds if 5 failed attempts occur sequentially. Circuit breaker policies are stateful. All calls through this client share the same circuit state.

Add policies from the Polly registry

An approach to managing regularly used policies is to define them once and register them with a [PolicyRegistry](#). For example:

```
C#

var timeoutPolicy = Policy.TimeoutAsync<HttpResponseMessage>(
    TimeSpan.FromSeconds(10));
var longTimeoutPolicy = Policy.TimeoutAsync<HttpResponseMessage>(
    TimeSpan.FromSeconds(30));

var policyRegistry = builder.Services.AddPolicyRegistry();

policyRegistry.Add("Regular", timeoutPolicy);
policyRegistry.Add("Long", longTimeoutPolicy);

builder.Services.AddHttpClient("PollyRegistryRegular")
    .AddPolicyHandlerFromRegistry("Regular");

builder.Services.AddHttpClient("PollyRegistryLong")
    .AddPolicyHandlerFromRegistry("Long");
```

In the preceding code:

- Two policies, [Regular](#) and [Long](#), are added to the Polly registry.
- [AddPolicyHandlerFromRegistry](#) configures individual named clients to use these policies from the Polly registry.

For more information on [IHttpClientFactory](#) and Polly integrations, see the [Polly wiki](#).

HttpClient and lifetime management

A new [HttpClient](#) instance is returned each time [CreateClient](#) is called on the [IHttpClientFactory](#). An [HttpMessageHandler](#) is created per named client. The factory manages the lifetimes of the [HttpMessageHandler](#) instances.

[IHttpClientFactory](#) pools the [HttpMessageHandler](#) instances created by the factory to reduce resource consumption. An [HttpMessageHandler](#) instance may be reused from the pool when creating a new [HttpClient](#) instance if its lifetime hasn't expired.

Pooling of handlers is desirable as each handler typically manages its own underlying HTTP connections. Creating more handlers than necessary can result in connection delays. Some handlers also keep connections open indefinitely, which can prevent the handler from reacting to DNS (Domain Name System) changes.

The default handler lifetime is two minutes. The default value can be overridden on a per named client basis:

```
C#

builder.Services.AddHttpClient("HandlerLifetime")
    .SetHandlerLifetime(TimeSpan.FromMinutes(5));
```

[HttpClient](#) instances can generally be treated as .NET objects **not** requiring disposal. Disposal cancels outgoing requests and guarantees the given [HttpClient](#) instance can't be used after calling [Dispose](#). [IHttpClientFactory](#) tracks and disposes resources used by [HttpClient](#) instances.

Keeping a single [HttpClient](#) instance alive for a long duration is a common pattern used before the inception of [IHttpClientFactory](#). This pattern becomes unnecessary after migrating to [IHttpClientFactory](#).

Alternatives to IHttpClientFactory

Using [IHttpClientFactory](#) in a DI-enabled app avoids:

- Resource exhaustion problems by pooling [HttpMessageHandler](#) instances.

- Stale DNS problems by cycling `HttpMessageHandler` instances at regular intervals.

There are alternative ways to solve the preceding problems using a long-lived `SocketsHttpHandler` instance.

- Create an instance of `SocketsHttpHandler` when the app starts and use it for the life of the app.
- Configure `PooledConnectionLifetime` to an appropriate value based on DNS refresh times.
- Create `HttpClient` instances using `new HttpClient(handler, disposeHandler: false)` as needed.

The preceding approaches solve the resource management problems that `IHttpClientFactory` solves in a similar way.

- The `SocketsHttpHandler` shares connections across `HttpClient` instances. This sharing prevents socket exhaustion.
- The `SocketsHttpHandler` cycles connections according to `PooledConnectionLifetime` to avoid stale DNS problems.

Logging

Clients created via `IHttpClientFactory` record log messages for all requests. Enable the appropriate information level in the logging configuration to see the default log messages. Additional logging, such as the logging of request headers, is only included at trace level.

The log category used for each client includes the name of the client. A client named *MyNamedClient*, for example, logs messages with a category of "System.Net.Http.HttpClient.**MyNamedClient**.LogicalHandler". Messages suffixed with *LogicalHandler* occur outside the request handler pipeline. On the request, messages are logged before any other handlers in the pipeline have processed it. On the response, messages are logged after any other pipeline handlers have received the response.

Logging also occurs inside the request handler pipeline. In the *MyNamedClient* example, those messages are logged with the log category "System.Net.Http.HttpClient.**MyNamedClient**.ClientHandler". For the request, this occurs after all other handlers have run and immediately before the request is sent. On the response, this logging includes the state of the response before it passes back through the handler pipeline.

Enabling logging outside and inside the pipeline enables inspection of the changes made by the other pipeline handlers. This may include changes to request headers or to the response status code.

Including the name of the client in the log category enables log filtering for specific named clients.

Configure the HttpMessageHandler

It may be necessary to control the configuration of the inner `HttpMessageHandler` used by a client.

An `IHttpClientBuilder` is returned when adding named or typed clients. The `ConfigurePrimaryHttpMessageHandler` extension method can be used to define a delegate. The delegate is used to create and configure the primary `HttpMessageHandler` used by that client:

```
C#

builder.Services.AddHttpClient("ConfiguredHttpMessageHandler")
    .ConfigurePrimaryHttpMessageHandler(() =>
        new HttpClientHandler
        {
            AllowAutoRedirect = true,
            UseDefaultCredentials = true
        });
```

Cookies

The pooled `HttpMessageHandler` instances results in `CookieContainer` objects being shared. Unanticipated `CookieContainer` object sharing often results in incorrect code. For apps that require cookies, consider either:

- Disabling automatic cookie handling
- Avoiding `IHttpClientFactory`

Call `ConfigurePrimaryHttpMessageHandler` to disable automatic cookie handling:

```
C#

builder.Services.AddHttpClient("NoAutomaticCookies")
    .ConfigurePrimaryHttpMessageHandler(() =>
        new HttpClientHandler
        {
```

```
UseCookies = false
```

```
});
```

Use IHttpConnectionFactory in a console app

In a console app, add the following package references to the project:

- [Microsoft.Extensions.Hosting](#) [↗](#)
- [Microsoft.Extensions.Http](#) [↗](#)

In the following example:

- [IHttpClientFactory](#) and `GitHubService` are registered in the [Generic Host's](#) service container.
- `GitHubService` is requested from DI, which in-turn requests an instance of `IHttpClientFactory`.
- `GitHubService` uses `IHttpClientFactory` to create an instance of `HttpClient`, which it uses to retrieve docs GitHub branches.

C#

```
using System.Text.Json;
using System.Text.Json.Serialization;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

var host = new HostBuilder()
    .ConfigureServices(services =>
    {
        services.AddHttpClient();
        services.AddTransient<GitHubService>();
    })
    .Build();

try
{
    var gitHubService = host.Services.GetRequiredService<GitHubService>();
    var gitHubBranches = await gitHubService.GetAspNetCoreDocsBranchesAsync();

    Console.WriteLine($"{gitHubBranches?.Count() ?? 0} GitHub Branches");

    if (gitHubBranches is not null)
    {
        foreach (var gitHubBranch in gitHubBranches)
        {
            Console.WriteLine($"{gitHubBranch.Name}");
        }
    }
}
catch (Exception ex)
{
    host.Services.GetRequiredService<ILogger<Program>>()
        .LogError(ex, "Unable to load branches from GitHub.");
}

public class GitHubService
{
    private readonly IHttpConnectionFactory _httpClientFactory;

    public GitHubService(IHttpConnectionFactory httpClientFactory) =>
        _httpClientFactory = httpClientFactory;

    public async Task<IEnumerable<GitHubBranch>> GetAspNetCoreDocsBranchesAsync()
    {
        var httpRequestMessage = new HttpRequestMessage(
            HttpMethod.Get,
            "https://api.github.com/repos/dotnet/AspNetCore.Docs/branches")
        {
            Headers =
            {
                { "Accept", "application/vnd.github.v3+json" },
                { "User-Agent", "HttpRequestsConsoleSample" }
            }
        };

        var httpClient = _httpClientFactory.CreateClient();
        var httpResponseMessage = await httpClient.SendAsync(httpRequestMessage);

        httpResponseMessage.EnsureSuccessStatusCode();
    }
}
```

```

using var contentStream =
    await httpResponseMessage.Content.ReadAsStreamAsync();

return await JsonSerializer.DeserializeAsync
    <IEnumerable<GitHubBranch>>(contentStream);
}


}

public record GitHubBranch(
    [property: JsonPropertyName("name")] string Name);

```

Header propagation middleware

Header propagation is an ASP.NET Core middleware to propagate HTTP headers from the incoming request to the outgoing `HttpClient` requests. To use header propagation:

- Install the [Microsoft.AspNetCore.HeaderPropagation](#)  package.
- Configure the `HttpClient` and middleware pipeline in `Program.cs`:

```

C#

// Add services to the container.
builder.Services.AddControllers();

builder.Services.AddHttpClient("PropagateHeaders")
    .AddHeaderPropagation();

builder.Services.AddHeaderPropagation(options =>
{
    options.Headers.Add("X-TraceId");
});

var app = builder.Build();

// Configure the HTTP request pipeline.
app.UseHttpsRedirection();


app.UseHeaderPropagation();

app.MapControllers();

```

- Make outbound requests using the configured `HttpClient` instance, which includes the added headers.

Additional resources

- [View or download sample code](#)  (how to download)
- [Use HttpClientFactory to implement resilient HTTP requests](#)
- [Implement HTTP call retries with exponential backoff with HttpClientFactory and Polly policies](#)
- [Implement the Circuit Breaker pattern](#)
- [How to serialize and deserialize JSON in .NET](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)