**C# HttpClient**

*last modified February 27, 2022*

C# HttpClient tutorial shows how to create HTTP requests with HttpClient in C#. In the examples, we create simple GET and POST requests.

The Hypertext Transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web.

HttpClient is a base class for sending HTTP requests and receiving HTTP responses from a resource identified by a URI.

**HTTP request methods**

HTTP defines a set of request methods to indicate the desired action to be performed for a given resource.

* GET - requests a representation of the specified resource
* HEAD - identical to a GET request, but without the response body
* POST - sends data to a resource, often causing state change or side effects
* PUT - creates a resource or updates an existing resource
* DELETE - deletes the specified resource
* CONNECT - starts two-way communications with the requested resource
* OPTION - describes the communication options for the target resource
* TRACE - returns the full HTTP request back for debugging purposes
* PATCH - performs partial modifications to the resource

**C# HttpClient status code**

HTTP response status codes indicate whether a specific HTTP request has been successfully completed. Responses are grouped in five classes:

* Informational responses (100–199)
* Successful responses (200–299)
* Redirects (300–399)
* Client errors (400–499)
* Server errors (500–599)

**Program.cs**

namespace HttpClientStatus;

class Program

{

static async Task Main(string[] args)

{

using var client = new HttpClient();

var result = await client.GetAsync("http://webcode.me");

Console.WriteLine(result.StatusCode);

}

}

The example creates a GET request to a small website. We get the status code of the request.

using var client = new HttpClient();

A new HttpClient is created.

var result = await client.GetAsync("http://webcode.me");

The GetAsync method sends a GET request to the specified Uri as an asynchronous operation. The await operator suspends the evaluation of the enclosing async method until the asynchronous operation completes. When the asynchronous operation completes, the await operator returns the result of the operation, if any.

$ dotnet run

OK

We get the 200 OK status code; the website is up.

**C# HttpClient GET request**

The GET method requests a representation of the specified resource.

**Program.cs**

using var client = new HttpClient();

var content = await client.GetStringAsync("http://webcode.me");

Console.WriteLine(content);

The example issues a GET request to the webcode.me website. It outputs the simple HTML code of the home page.

var content = await client.GetStringAsync("http://webcode.me");

The GetStringAsync sends a GET request to the specified Uri and returns the response body as a string in an asynchronous operation.

$ dotnet run

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<link rel="stylesheet" href="format.css">

<title>My html page</title>

</head>

<body>

<p>

Today is a beautiful day. We go swimming and fishing.

</p>

<p>

Hello there. How are you?

</p>

</body>

</html>

**C# HttpClient HEAD request**

The HTTP HEAD method requests the headers that are returned if the specified resource would be requested with an HTTP GET method.

**Program.cs**

var url = "[http://webcode.me](http://webcode.me/)";

using var client = new HttpClient();

var result = await client.SendAsync(new HttpRequestMessage(HttpMethod.Head, url));

Console.WriteLine(result);

The example issues a HEAD request.

$ dotnet run

StatusCode: 200, ReasonPhrase: 'OK', Version: 1.1,

Content: System.Net.Http.HttpConnectionResponseContent, Headers:

{

Server: nginx/1.6.2

Date: Tue, 12 Jan 2021 12:01:07 GMT

Connection: keep-alive

ETag: "5d32ffc5-15c"

Accept-Ranges: bytes

Content-Type: text/html

Content-Length: 348

Last-Modified: Sat, 20 Jul 2019 11:49:25 GMT

}

These are the header fields of the response.

**C# HttpClient User-Agent**

The User-Agent request header is a string that lets servers and network peers identify the application, operating system, vendor, and/or version of the requesting user agent.

**Program.cs**

var url = "<http://webcode.me/ua.php>";

using var client = new HttpClient();

client.DefaultRequestHeaders.Add("User-Agent", "C# program");

var res = await client.GetStringAsync(url);

Console.WriteLine(res);

The example sets a User-Agent header for its GET request. The requested resource simply returns the client's User-Agent string.

**C# HttpClient HttpRequestMessage**

The HttpRequestMessage represents a request message.

**Program.cs**

var url = "[http://webcode.me](http://webcode.me/)";

using var client = new HttpClient();

var msg = new HttpRequestMessage(HttpMethod.Get, url);

msg.Headers.Add("User-Agent", "C# Program");

var res = await client.SendAsync(msg);

var content = await res.Content.ReadAsStringAsync();

Console.WriteLine(content);

In the example, we manually build the request message.

var msg = new HttpRequestMessage(HttpMethod.Get, url);

msg.Headers.Add("User-Agent", "C# Program");

var res = await client.SendAsync(msg);

A GET request message is created with HttpRequestMessage and sent with SendAsync.

var content = await res.Content.ReadAsStringAsync();

We read the content of the response with ReadAsStringAsync.

**C# HttpClient query strings**

Query string is a part of the URL which is used to add some data to the request for the resource. It is often a sequence of key/value pairs. It follows the path and starts with the ? character.

**Program.cs**

var u = "<http://webcode.me/qs.php>";

using var client = new HttpClient();

var builder = new UriBuilder(u);

builder.Query = "name=John Doe&occupation=gardener";

var url = builder.ToString();

var res = await client.GetAsync(url);

var content = await res.Content.ReadAsStringAsync();

Console.WriteLine(content);

The query string is built with the UriBuilder.

$ dotnet run

{"name":"John Doe","occupation":"gardener"}

**C# HttpClient multiple async requests**

In the following example, we generate multiple asynchronous GET requests.

**Program.cs**

using System.Text.RegularExpressions;

var urls = new string[] { "[http://webcode.me](http://webcode.me/)", "[http://example.com](http://example.com/)",

"[http://httpbin.org](http://httpbin.org/)", "[https://ifconfig.me](https://ifconfig.me/)", "[http://termbin.com](http://termbin.com/)",

"[https://github.com](https://github.com/)"

};

var rx = new Regex(@"<title>\s\*(.+?)\s\*</title>",

RegexOptions.Compiled);

using var client = new HttpClient();

var tasks = new List<Task<string>>();

foreach (var url in urls)

{

tasks.Add(client.GetStringAsync(url));

}

Task.WaitAll(tasks.ToArray());

var data = new List<string>();

foreach (var task in tasks)

{

data.Add(await task);

}

foreach (var content in data)

{

var matches = rx.Matches(content);

foreach (var match in matches)

{

Console.WriteLine(match);

}

}

We download the given web pages asynchronously and print their HTML title tags.

tasks.Add(client.GetStringAsync(url));

The GetStringAsync sends a GET request to the specified url and returns the response body as a string in an asynchronous operation. It returns a new task; in C# a task represents an asynchronous operation.

Task.WaitAll(tasks.ToArray());

The Task.WaitAll waits for all of the provided tasks to complete execution.

data.Add(await task);

The await unwraps the result value.

$ dotnet run

<title>My html page</title>

<title>Example Domain</title>

<title>httpbin.org</title>

<title>termbin.com - terminal pastebin</title>

<title>GitHub: Where the world builds software · GitHub</title>

**C# HttpClient POST request**

The HTTP POST method sends data to the server. The type of the body of the request is indicated by the Content-Type header.

$ dotnet add package Newtonsoft.Json

We need to add the Newtonsoft.Json package to process JSON data.

**Program.cs**

using System.Text;

using Newtonsoft.Json;

var person = new Person("John Doe", "gardener");

var json = JsonConvert.SerializeObject(person);

var data = new StringContent(json, Encoding.UTF8, "application/json");

var url = "<https://httpbin.org/post>";

using var client = new HttpClient();

var response = await client.PostAsync(url, data);

var result = await response.Content.ReadAsStringAsync();

Console.WriteLine(result);

record Person(string Name, string Occupation);

In the example, we send a POST request to <https://httpbin.org/post> website, which is an online testing service for developers.

var person = new Person("John Doe", "gardener");

var json = JsonConvert.SerializeObject(person);

var data = new StringContent(json, Encoding.UTF8, "application/json");

We turn an object into a JSON data with the help of the Newtonsoft.Json package.

var response = await client.PostAsync(url, data);

We send an asynchronous POST request with the PostAsync method.

var result = await response.Content.ReadAsStringAsync();

Console.WriteLine(result);

We read the returned data and print it to the console.

$ dotnet run

{

"args": {},

"data": "{\"Name\":\"John Doe\",\"Occupation\":\"gardener\"}",

"files": {},

"form": {},

"headers": {

"Content-Length": "43",

"Content-Type": "application/json; charset=utf-8",

"Host": "httpbin.org",

"X-Amzn-Trace-Id": "Root=1-5ffd917e-349220186065913c2544d3ba"

},

"json": {

"Name": "John Doe",

"Occupation": "gardener"

},

...

"url": "<https://httpbin.org/post>"

}

**C# HttpClient JSON request**

JSON (JavaScript Object Notation) is a lightweight data-interchange format. This format is easy for humans to read and write and for machines to parse and generate. It is a less verbose and more readable alternative to XML. The official Internet media type for JSON is application/json.

**Program.cs**

using System.Net.Http.Headers;

using Newtonsoft.Json;

using var client = new HttpClient();

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

client.DefaultRequestHeaders.Add("User-Agent", "C# console program");

client.DefaultRequestHeaders.Accept.Add(

new MediaTypeWithQualityHeaderValue("application/json"));

var url = "repos/symfony/symfony/contributors";

HttpResponseMessage response = await client.GetAsync(url);

response.EnsureSuccessStatusCode();

var resp = await response.Content.ReadAsStringAsync();

List<Contributor> contributors = JsonConvert.DeserializeObject<List<Contributor>>(resp);

contributors.ForEach(Console.WriteLine);

record Contributor(string Login, short Contributions);

The example generates a GET request to to Github. It finds out the top contributors of the Symfony framework. It uses the Newtonsoft.Json to work with JSON.

client.DefaultRequestHeaders.Add("User-Agent", "C# console program");

In the request header, we specify the user agent.

client.DefaultRequestHeaders.Accept.Add(

new MediaTypeWithQualityHeaderValue("application/json"));

In the accept header value, we tell that JSON is an acceptable response type.

var url = "repos/symfony/symfony/contributors";

HttpResponseMessage response = await client.GetAsync(url);

var resp = await response.Content.ReadAsStringAsync();

We generate a request and read the content asynchronously.

List<Contributor> contributors = JsonConvert.DeserializeObject<List<Contributor>>(resp);

contributors.ForEach(Console.WriteLine);

We transform the JSON response into a list of Contributor objects with the JsonConvert.DeserializeObject method.

**C# HttpClient POST form data**

POST requests are often sent via a post form. The type of the body of the request is indicated by the Content-Type header. The FormUrlEncodedContent is a container for name/value tuples encoded using application/x-www-form-urlencoded MIME type.

**Program.cs**

var url = "<https://httpbin.org/post>";

using var client = new HttpClient();

var data = new Dictionary<string, string>

{

{"name", "John Doe"},

{"occupation", "gardener"}

};

var res = await client.PostAsync(url, new FormUrlEncodedContent(data));

var content = await res.Content.ReadAsStringAsync();

Console.WriteLine(content);

The example sends a form POST requests using FormUrlEncodedContent.

$ dotnet run

{

"args": {},

"data": "",

"files": {},

"form": {

"name": "John Doe",

"occupation": "gardener"

},

"headers": {

"Content-Length": "33",

"Content-Type": "application/x-www-form-urlencoded",

"Host": "httpbin.org",

"X-Amzn-Trace-Id": "Root=1-6218d376-03e6c2f004520f2f2e504f37"

},

"json": null,

...

"url": "<https://httpbin.org/post>"

}

**C# HttpClient proxy**

A proxy is an intermediary between a client requesting a resource and the server providing that resource.

**Program.cs**

using System.Net;

var port = 7302;

var proxy = "example.com";

var url = "[http://webcode.me](http://webcode.me/)";

var handler = new HttpClientHandler()

{

Proxy = new WebProxy(new Uri($"socks5://{proxy}:{port}")),

UseProxy = true,

};

using var client = new HttpClient(handler);

var res = await client.GetAsync(url);

var content = await res.Content.ReadAsStringAsync();

Console.WriteLine(content);

The example creates a web request through a proxy. C# uses WebProxy to set up a proxy server.

**C# HttpClient download image**

The GetByteArrayAsync sends a GET request to the specified Uri and returns the response body as a byte array in an asynchronous operation.

**Program.cs**

using var httpClient = new HttpClient();

var url = "<http://webcode.me/favicon.ico>";

byte[] imageBytes = await httpClient.GetByteArrayAsync(url);

string documentsPath = System.Environment.GetFolderPath(

System.Environment.SpecialFolder.Personal);

string localFilename = "favicon.ico";

string localPath = Path.Combine(documentsPath, localFilename);

Console.WriteLine(localPath);

File.WriteAllBytes(localPath, imageBytes);

In the example, we download an image from the webcode.me website. The image is written to the user's Documents folder.

byte[] imageBytes = await httpClient.GetByteArrayAsync(url);

The GetByteArrayAsync returns the image as an array of bytes.

string documentsPath = System.Environment.GetFolderPath(

System.Environment.SpecialFolder.Personal);

We determine the Documents folder with the GetFolderPath method.

File.WriteAllBytes(localPath, imageBytes);

The bytes are written to the disk with the File.WriteAllBytes method.

**C# HttpClient Basic authentication**

In HTTP protocol, basic access authentication is a method for an HTTP user agent (such as a web browser or a console application) to provide a user name and password when making a request. In basic HTTP authentication, a request contains a header field in the form of Authorization: Basic <credentials>, where credentials is the base64 encoding of id and password joined by a single colon :.

**Note:** The credentials are not encrypted; therefore, HTTP basic authentication must be used with the HTTPS protocol.

HTTP Basic authentication is the simplest technique for enforcing access controls to web resources. It does not require cookies, session identifiers, or login pages; rather, HTTP Basic authentication uses standard fields in the HTTP header.

**Program.cs**

using System.Text;

using System.Net.Http.Headers;

var userName = "user7";

var passwd = "passwd";

var url = "<https://httpbin.org/basic-auth/user7/passwd>";

using var client = new HttpClient();

var authToken = Encoding.ASCII.GetBytes($"{userName}:{passwd}");

client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Basic",

Convert.ToBase64String(authToken));

var result = await client.GetAsync(url);

var content = await result.Content.ReadAsStringAsync();

Console.WriteLine(content);

The example sends credentials to the httpbin.org website.

var authToken = Encoding.ASCII.GetBytes($"{userName}:{passwd}");

client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Basic",

Convert.ToBase64String(authToken));

Here we build the authentication header.

var url = "<https://httpbin.org/basic-auth/user7/passwd>";

The URL contains authentication details because we test it with the httpbin.org website. This way we don't need to set up our own server. Authentication details are never put into the URL, of course.

$ dotnet run

{

"authenticated": true,

"user": "user7"

}

In this tutorial, we have used C# HttpClient to create HTTP requests.