**Reference URLs:**

<https://www.c-sharpcorner.com/article/create-api-with-asp-net-core-day-three-working-with-http-status-codes-in-asp/>

<https://www.c-sharpcorner.com/article/frequently-used-status-code-and-how-to-return-them-from-asp-net-core-web-api/>

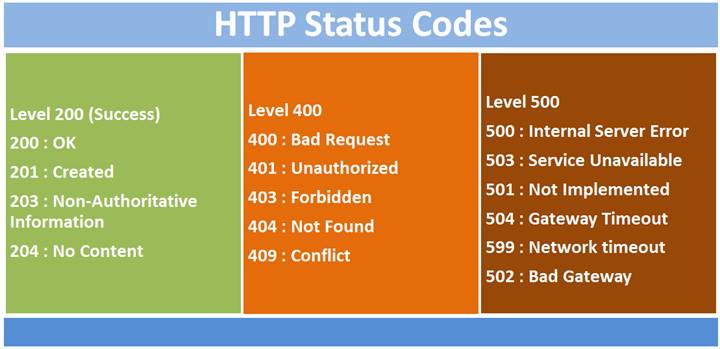
<https://docs.microsoft.com/en-us/aspnet/core/web-api/advanced/formatting?view=aspnetcore-3.1>

<https://medium.com/cheranga/calling-web-apis-using-typed-httpclients-net-core-20d3d5ce980>

<https://ochzhen.com/blog/created-createdataction-createdatroute-methods-explained-aspnet-core#:~:text=The%20main%20difference%20is%20how,and%20CreatedAtRoute%20gives%20more%20safety>.

**Working with HttpStatusCodes:**

While consuming an API, an HTTP Request is sent and in return, the response is sent along with the return data and a HTTP. The HTTP code Status Codes are important because they tell consumers about what exactly happened to their request; A wrong HTTP code can confuse the consumer. A consumer should know via the response if his/her request has been well taken care of or not, and if the response is not as expected, the Status Code should tell the consumer where the problem is - if it is at consumer level or at API level?

Suppose there is a situation where consumer gets response as status code 200 but at the service level, there is some problem or issue. In that case, consumer will get a false assumption of everything being fine, whereas that won’t be a case. So if there is something wrong at service or there occurs some error on the server, the status code, so that the consumer knows that there actually is something wrong with the request that it sent. In general, there are a lot of access codes. One can find the complete list [here](http://www.restapitutorial.com/httpstatuscodes.html), but not all are so important 500 should be sent to consumer except the few. Few status codes are very frequently used with the normal CRUD operations that a service performs, so service does not necessarily have to support all of them. Let’s have a glance over a few of the important status codes.  
  
****  
  
When we talk about the levels of status codes, there are 5 levels. Level 100 status codes are more of informal nature. Level 200 status codes are specifically for request being sent well. We get 200 codes for success of a GET request, 201 if a new resource has been successfully created. 204 status codes is also for success but in return it does not returns anything, just like if consumer has performed delete operation and in return doesn’t really expect something back. Level 300 http status codes are basically used for redirection, for example to tell a consumer that the requested resource like page, image has been moved to another location. Level 400 status codes are meant to state errors or client error for e.g. status code 400 means Bad Request, 401 is Unauthorized that is invalid authentication credentials or details have been provided by the consumer, 403 means that authentication is a success, but the user is not authorized. 404 are very common and we often encounter which mean that the requested resource is not available. Level 500 are for server errors. Internal Server Error exception is very common, that contains the code 500. This error means that there is some unexpected error on the server and client cannot do anything about it. We’ll cover how we can use these HTTP status codes in our application.

Request and response are the backbones of any RESTful Web API. Every status code has a specific meaning and during the development, we must make sure that we are returning a valid response with a valid status code.

**What is the Status Code?**

Status code is a numeric value, and it is the main component of HTTP Response. The status code is issued by the server based on the operation, input data, and some other parameters.

The status code gives some useful information about the behavior of the response.

**Status code categories**

All status codes are divided into the following 5 categories,

**1xx – Informational**

**2xx – Successful**

**3xx – Redirection**

**4xx – Client Error**

**5xx – Server Error**

**Commonly Used Status Codes:**

**200 status code**

This is the most common status code that is returned from Web API.

This 200 status code belongs to the Successful category. It means everything about the operation is successful.

**Example**

Get all employees data

Get single employee data

Asp.Net Core has **Ok() method** to **return 200 status code.**

**201 Status code**

201 status code indicates that the new resource has been created successfully and the server will return the link to get that newly created resource.

Asp.Net Core has **Created() method** to return **201 status code.**

**400 status code**

400 status code indicated the bad request. It means there is something wrong in the request data.

In asp.net core we can return **400 status** using the **BadRequest()** method.

**404 status code**

If we are looking for a resource that does not exist, then the server returns a 404 status code.

In asp.net core we can return **404 status** using the **NotFound() method.**

**CreatedAtRoute vs CreatedAtAction**

Now we know a lot about these two methods and, therefore, can discuss them together. Here are some ***key points*** to consider:

* Both of them provide different ways of achieving the same thing - returning 201 Created response with a **Location** header
* Both of them check correctness and existence of the target action, thus, ensuring validness of the returned URI
* **CreatedAtAction** finds target action by method and controller names **CreatedAtRoute** finds target action by a route name
* **CreatedAtAction** requires action name, default is controller method name but can be assigned with **ActionName** attribute **CreatedAtRoute** requires a name of the target route, it can be assigned to particular route by us or declared in Startup
* **CreatedAtRoute** covers functionality of **CreatedAtAction** when using overload that doesn’t require **routeName** parameter

As always, which one to use depends on the use case. In general, **CreatedAtRoute** gives more options and includes functionality of **CreatedAtAction**. However, in some cases **CreatedAtAction** is more convenient, for example, when handling actions inside of the same controller.

* ***The main difference*** is how you specify the URI of the created resource which will be included in the Location header.
* **Created** gives you ***more control*** over URI creation, whereas **CreatedAtAction** and **CreatedAtRoute** gives ***more safety***.
* If your server is behind ***load balancer*** or ***reverse proxy***, you might want to use **Created** method

## Created Explained

This method is very straightforward and has only two overloaded versions. Detailed signatures of both methods can be found [here](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase.created?view=aspnetcore-5.0).

Since **Created** method is very simple, it needs only these two pieces of information:

* **uri** - simply the URI that should be returned in the **Location** header
* **value** - content to return in a response body

Basically, the main difference between them is how you pass the value of the location header - as **string** or **Uri**.

**Important points** about passing location header using these methods:

* We are responsible for ensuring URI correctness and existence on the server at all times, for example, when we update paths or arguments.
* Using **string** means that value will be returned as is, without any validation.
* **Uri** [class](https://docs.microsoft.com/en-us/dotnet/api/system.uri?view=net-5.0) helps ensure that the URI format is correct. However, it doesn’t check that this path exists on the server.
* Value can be either absolute or relative.

## CreatedAtAction Explained

This method provides more support in generating URI for the Location header.

As the name suggests, this method allows us to set Location URI of the newly created resource by **specifying the name of an action** where we can retrieve our resource.

**Content Negotiation in ASP.NET Core Web API:**

ASP.NET Core MVC supports formatting response data, using specified formats or in response to a client's request.

## Format-specific Action Results

Some action result types are specific to a particular format, such as [JsonResult](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.jsonresult) and [ContentResult](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.contentresult). Actions can return results that always use a specified format, ignoring a client's request for a different format. For example, returning JsonResult returns JSON-formatted data and returning ContentResult returns plain-text-formatted string data.

An action isn't required to return any specific type. ASP.NET Core supports any object return value. Results from actions that return objects that aren't [IActionResult](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.iactionresult) types are serialized using the appropriate [IOutputFormatter](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.formatters.ioutputformatter) implementation. For more information, see [Controller action return types in ASP.NET Core web API](https://docs.microsoft.com/en-us/aspnet/core/web-api/action-return-types?view=aspnetcore-6.0).

By default, the built-in helper method [ControllerBase.Ok](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase.ok) returns JSON-formatted data:

[HttpGet]

public IActionResult Get() =>

Ok(\_todoItemStore.GetList());

The sample code returns a list of todo items. Using the F12 browser developer tools or [Postman](https://www.getpostman.com/tools) with the previous code displays:

* The response header containing **content-type:** application/json; charset=utf-8.
* The request headers. For example, the Accept header. The Accept header is ignored by the preceding code.

To return plain text formatted data, use [ContentResult](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.contentresult) and the [Content](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase.content) helper:

[HttpGet("Version")]

public ContentResult GetVersion() =>

Content("v1.0.0");

In the preceding code, the Content-Type returned is text/plain.

For actions with multiple return types, return IActionResult. For example, when returning different HTTP status codes based on the result of the operation.

## Content negotiation

Content negotiation occurs when the client specifies an [Accept header](https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html). The default format used by ASP.NET Core is [JSON](https://json.org/). Content negotiation is:

* Implemented by [ObjectResult](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.objectresult).
* Built into the status code-specific action results returned from the helper methods. The action results helper methods are based on ObjectResult.

When a model type is returned, the return type is ObjectResult.

The following action method uses the Ok and NotFound helper methods:

[HttpGet("{id:long}")]

public IActionResult GetById(long id)

{

var todo = \_todoItemStore.GetById(id);

if (todo is null)

{

return NotFound();

}

return Ok(todo);

}

By default, ASP.NET Core supports the following media types:

* application/json
* text/json
* text/plain

Tools such as [Fiddler](https://www.telerik.com/fiddler) or [Postman](https://www.getpostman.com/tools) can set the Accept request header to specify the return format. When the Accept header contains a type the server supports, that type is returned. The next section shows how to add additional formatters.

Controller actions can return POCOs (Plain Old CLR Objects). When a POCO is returned, the runtime automatically creates an ObjectResult that wraps the object. The client gets the formatted serialized object. If the object being returned is null, a 204 No Content response is returned.

The following example returns an object type:

[HttpGet("{id:long}")]

public TodoItem? GetById(long id) =>

\_todoItemStore.GetById(id);

In the preceding code, a request for a valid todo item returns a 200 OK response. A request for an invalid todo item returns a 204 No Content response.

### The Accept header

Content negotiation takes place when an Accept header appears in the request. When a request contains an accept header, ASP.NET Core:

* Enumerates the media types in the accept header in preference order.
* Tries to find a formatter that can produce a response in one of the formats specified.

If no formatter is found that can satisfy the client's request, ASP.NET Core:

* Returns 406 Not Acceptable if [MvcOptions.ReturnHttpNotAcceptable](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.mvcoptions.returnhttpnotacceptable" \l "microsoft-aspnetcore-mvc-mvcoptions-returnhttpnotacceptable) is set to true, or -
* Tries to find the first formatter that can produce a response.

If no formatter is configured for the requested format, the first formatter that can format the object is used. If no Accept header appears in the request:

* The first formatter that can handle the object is used to serialize the response.
* There isn't any negotiation taking place. The server is determining what format to return.

If the Accept header contains \*/\*, the Header is ignored unless RespectBrowserAcceptHeader is set to true on [MvcOptions](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.mvcoptions).

### Browsers and content negotiation

Unlike typical API clients, web browsers supply Accept headers. Web browsers specify many formats, including wildcards. By default, when the framework detects that the request is coming from a browser:

* The Accept header is ignored.
* The content is returned in JSON, unless otherwise configured.

This approach provides a more consistent experience across browsers when consuming APIs.

To configure an app to respect browser accept headers, set the [RespectBrowserAcceptHeader](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.mvcoptions.respectbrowseracceptheader" \l "microsoft-aspnetcore-mvc-mvcoptions-respectbrowseracceptheader) property to true:

C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers(options =>

{

options.RespectBrowserAcceptHeader = true;

});

### Configure formatters

Apps that need to support extra formats can add the appropriate NuGet packages and configure support. There are separate formatters for input and output. Input formatters are used by [Model Binding](https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-6.0). Output formatters are used to format responses. For information on creating a custom formatter, see [Custom Formatters](https://docs.microsoft.com/en-us/aspnet/core/web-api/advanced/custom-formatters?view=aspnetcore-6.0).

### Add XML format support

To configure XML formatters implemented using [XmlSerializer](https://docs.microsoft.com/en-us/dotnet/api/system.xml.serialization.xmlserializer), call [AddXmlSerializerFormatters](https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.dependencyinjection.mvcxmlmvcbuilderextensions.addxmlserializerformatters):

C#

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers()

.AddXmlSerializerFormatters();

When using the preceding code, controller methods return the appropriate format based on the request's Accept header.

**Using Strongly Typed HttpClients:**

In these days, it’s almost impossible to implement a decent system without implementing or consuming a web API.

Building APIs has always been a first-class citizen and it has been evolving ever since in any programming platform with any major platform update or how we deploy our software.

In Microsoft related technologies, we have evolved from implementing an ASMX web service to implementing and deploying an API using Azure functions (yes, WCF, MVC, Web API approaches are, still widely used).

In modern (well maybe not so modern) times consumption of a web service has never been easy with the introduction of response types such as JSON (application/json). In this post I am going to discuss about how to consume a REST based API where the responses are in JSON format. For that I’ll be using .NETs HttpClient .

Alright, so what’s the big deal?

Well, as developers we would like to write beautiful, functioning code. So, when we saw HttpClient was implementing IDisposable interface, we used it within an using statements because that’s the sensible (A.K.A — best practice) thing to do. But… boy we were wrong! 😆

Well that’s one object which you shouldn’t be messing with by wrapping inside an using statement.

By default the behavior of an HttpClient instance is it’s thread safe and can be reused. So if you have many HttpClient instances created; each instance will have its own connection pool (which basically means, there will be many sockets opened in a given time). The problem is, though you dispose the HttpClient instance, the underlying sockets will not be released that soon. So, if you get a burst or if your API needs to serve many requests at the same time, it will fail to do at a certain point in time because the sockets will be exhausted.

**Behold..here comes .NET Core 2.1**

Well starting from .NET Core 2.1 Microsoft saw this becoming a real problem and introduced HttpClientFactory . As per the name, it’s a factory which we can use to create HttpClient instances without worrying about disposables, DNS changes and all that drama.

And, to be honest they did it in style! 👍 👍 👍

You can use the HttpClient in many ways.