Returning a 404 from an explicitly typed ASP.NET Core API controller (not IActionResult)

Asked 2 years, 9 months ago Active 2 months ago Viewed 35k times

47

8

ASP.NET Core API controllers typically return explicit types (and do so by default if you create a new project), something like:

```
[Route("api/[controller]")]
public class ThingsController : Controller
{
    // GET api/things
    [HttpGet]
    public async Task<IEnumerable<Thing>> GetAsync()
    {
        //...
    }

    // GET api/things/5
    [HttpGet("{id}")]
    public async Task<Thing> GetAsync(int id)
    {
        Thing thingFromDB = await GetThingFromDBAsync();
        if(thingFromDB == null)
            return null; // This returns HTTP 204

        // Process thingFromDB, blah blah blah
        return thing;
    }

    // POST api/things
    [HttpPost]
    public void Post([FromBody]Thing thing)
    {
        //..
    }

    //... and so on...
}
```

This is then regarded by a lot of client side Javascript components as success, so there's code like:

```
const response = await fetch('.../api/things/5', {method: 'GET' ...});
if(response.ok)
    return await response.json(); // Error, no content!
```

A search online (such as this question and this answer) points to helpful `return NotFound();` extension methods for the controller, but all these return `IActionResult`, which isn't compatible with my `Task<Thing>` return type. That design pattern looks like this:

```
// GET api/things/5
[HttpGet("{id}")]
public async Task<IActionResult> GetAsync(int id)
{
    var thingFromDB = await GetThingFromDBAsync();
    if (thingFromDB == null)
        return NotFound();

    // Process thingFromDB, blah blah blah
    return Ok(thing);
}
```

That works, but to use it the return type of `GetAsync` must be changed to `Task<IActionResult>` - the explicit typing is lost, and either all the return types on the controller have to change (i.e. not use explicit typing at all) or there will be a mix where some actions deal with explicit types while others. In addition unit tests now need to make assumptions about the serialisation and explicitly deserialise the content of the `IActionResult` where before they had a concrete type.

There are loads of ways around this, but it appears to be a confusing mishmash that could easily be designed out, so the real question is: **what is the correct way intended by the ASP.NET Core designers?**

It seems that the possible options are:

1. Have a weird (messy to test) mix of explicit types and `IActionResult` depending on expected type.
2. Forget about explicit types, they're not really supported by Core MVC, *always* use `IActionResult` (in which case why are they present at all?)
3. Write an implementation of `HttpResponseException` and use it like `ArgumentOutOfRangeException` (see this answer for an implementation). However, that does require using exceptions for program flow, which is generally a bad idea and also deprecated by the MVC Core team.

6. Or is there a reason why `204` is correct and `404` wrong for a failed GET request?

These all involve compromises and refactoring that lose something or add what seems to be unnecessary complexity at odds with the design of MVC Core. Which compromise is the correct one and why?

c# asp.net-core asp.net-core-mvc http-status-code-404 asp.net-core-webapi

edited Jul 19 at 21:18 — T.S. — 11.1k 10 38 57

asked Jan 4 '17 at 13:03 — Keith — 100k 63 249 365

1 @Hackerman hi, did you read the question? I am specifically aware of `StatusCode(500)` and it only works with actions that return `IActionResult`, which I then go into some detail on. – Keith Jan 4 '17 at 13:10

1 @Hackerman no, it specifically isn't. That *only* works with `IActionResult`. I'm asking about actions with *explicit types*. I go on to enquire about the use of `IActionResult` in the first bullet point, but I'm not asking how to call `StatusCode(404)` - I already know and cite it in the question. – Keith Jan 4 '17 at 13:15

1 For your scenario the solution could be something like `return new HttpResponseMessage(HttpStatusCode.NotFound);` ...also according to this: docs.microsoft.com/en-us/aspnet/core/mvc/models/formatting `For non-trivial actions with multiple return types or options (for example, different HTTP status codes based on the result of operations performed), prefer IActionResult as the return type.` – Hackerman Jan 4 '17 at 13:43

1 @Hackerman you voted to close my question as a dupe of a question that I had found, read and gone through *before I asked this question* and one that I addressed in the question as not the answer I was looking for. Obviously I went on the defensive - I want an answer to *my* question, not to be pointed back in a circle. Your final comment is actually useful and begins to address what I'm actually asking about - you should flesh it out to a full answer. – Keith Jan 4 '17 at 14:23

1 Ok, I got more info on the subject...in order to accomplish something like that(still I think that the best approach should be using `IActionResult`), you can follow this example `public Item Get(int id) { var item = _repo.FindById(id); if (item == null) throw new HttpResponseException(HttpStatusCode.NotFound); return item; }` where you can return an `HttpResponseException` if `thing` is `null` ... – Hackerman Jan 4 '17 at 17:02

4 Answers

This is addressed in ASP.NET Core 2.1 with `ActionResult<T>` :

```
    if (thing == null)
        return NotFound();

    return thing;
}
```

Or even:

```
public ActionResult<Thing> Get(int id) =>
    GetThingFromDB() ?? NotFound();
```

I'll update this answer with more detail once I've implemented it.

Original Answer

In ASP.NET Web API 5 there was an `HttpResponseException` (as pointed out by Hackerman) but it's been removed from Core and there's no middleware to handle it.

I think this change is due to .NET Core - where ASP.NET tries to do everything out of the box, ASP.NET Core only does what you specifically tell it to (which is a big part of why it's so much quicker and portable).

I can't find a an existing library that does this, so I've written it myself. First we need a custom exception to check for:

```
public class StatusCodeException : Exception
{
    public StatusCodeException(HttpStatusCode statusCode)
    {
        StatusCode = statusCode;
    }

    public HttpStatusCode StatusCode { get; set; }
}
```

Then we need a `RequestDelegate` handler that checks for the new exception and converts it to the HTTP response status code:

```
public class StatusCodeExceptionHandler
{
    private readonly RequestDelegate request;
```

```
        this.request = pipeline;
    }

    public Task Invoke(HttpContext context) => this.InvokeAsync(context); // Stops VS
from nagging about async method without ...Async suffix.

    async Task InvokeAsync(HttpContext context)
    {
        try
        {
            await this.request(context);
        }
        catch (StatusCodeException exception)
        {
            context.Response.StatusCode = (int)exception.StatusCode;
            context.Response.Headers.Clear();
        }
    }
}
```

Then we register this middleware in our `Startup.Configure`:

```
public class Startup
{
    ...

    public void Configure(IApplicationBuilder app)
    {
        ...
        app.UseMiddleware<StatusCodeExceptionHandler>();
```

Finally actions can throw the HTTP status code exception, while still returning an explicit type that can easily be unit tested without conversion from `IActionResult`:

```
public Thing Get(int id) {
    Thing thing = GetThingFromDB();

    if (thing == null)
        throw new StatusCodeException(HttpStatusCode.NotFound);

    return thing;
}
```

This keeps the explicit types for the return values and allows easy distinction between successful empty results (`return null;`) and an error because something can't be found (I think of it like throwing an `ArgumentOutOfRangeException`).

While this is a solution to the problem it still doesn't really answer my question - the designers of the Web API build support for explicit types with the expectation that they would be used, added specific handling for `return null;` so that it would produce a 204 rather than a 200, and then didn't add any way to deal with 404? It seems like a lot of work to add something so basic.

edited Feb 2 '18 at 18:24

answered Jan 5 '17 at 11:45

Keith

100k 63 249 365

I think that if the route-resource is valid, but doesn't return anything, the proper response should be 204(No Content)....if the route doesn't exist should return a 404 response(not found)...makes sense, right? – Hackerman Jan 5 '17 at 13:04

1 @Hackerman I suspect that might well be an option, but plenty of client side APIs generalise (1xx on it... 2xx ok, 3xx go here, 4xx you got it wrong, 5xx we got it wrong) - 2xx implies that all is OK, when actually the user requested a resource that isn't there (like the example in my question, the Fetch API considers 204 as OK to continue). I suppose 204 could mean right resource path, wrong resource ID, but that not been my understanding. Any citation on that as the desired pattern? – Keith Jan 5 '17 at 14:15

1 Take a look at this article racksburg.com/choosing-an-http-status-code ...I think that the flow chart for status codes 2xx/3xx explains it very well...you can look at the others too :)\ – Hackerman Jan 5 '17 at 14:27

@Hackerman That still points to 404 being correct in this context. – Keith Jan 5 '17 at 15:10

1 @Hackerman I'm really not sure on that - it seems like trying to convey information about the API in the API. If we're doing that shouldn't we also implement 405 (rather than 404) for valid controllers without the requested action and so on? In REST the thing being returned is the resource, not the API itself. I think that's why the convention is for names to be plural (rather than singular like they'd be in a DB) - `api/things/5` is the resource that expects to be a single *thing*. – Keith Jan 9 '17 at 9:23

14

You can actually use `IActionResult` or `Task<IActionResult>` instead of `Thing` or `Task<Thing>` or even `Task<IEnumerable<Thing>>` . If you have an API that returns **JSON** then you can simply do the following:

```
[Route("api/[controller]")]
public class ThingsController : Controller
{
    // GET api/things
    [HttpGet]
    public async Task<IActionResult> GetAsync()
```

```
    // GET api/things/5
    [HttpGet("{id}")]
    public async Task<IActionResult> GetAsync(int id)
    {
        var thingFromDB = await GetThingFromDBAsync();
        if (thingFromDB == null)
            return NotFound();

        // Process thingFromDB, blah blah blah
        return Ok(thing); // This will be JSON by default
    }

    // POST api/things
    [HttpPost]
    public void Post([FromBody] Thing thing)
    {
    }
}
```

Update

It seems as though the concern is that being *explicit* in the return of an API is somehow helpful, while it is possible to be *explicit* it is in fact not very useful. If you're writing unit tests that exercise the request / response pipeline you are typically going to verify the raw return (which would most likely be *JSON*, i.e.; a string in **C#**). You could simply take the returned string and convert it back to the strongly typed equivalent for comparisons using `Assert`.

This seems to be the only shortcoming with using `IActionResult` or `Task<IActionResult>`. If you really, really want to be explicit and still want to set the status code there are several ways to do this - but it is frowned upon as the framework already has a built-in mechanism for this, i.e.; using the `IActionResult` returning method wrappers in the `Controller` class. You could write some custom middleware to handle this however you'd like, however.

Finally, I would like to point out that if an API call returns `null` according to **W3** a status code of *204* is actually accurate. Why on earth would you want a *404*?

204

> The server has fulfilled the request but does not need to return an entity-body, and might want to return updated metainformation. The response MAY include new or updated metainformation in the form of entity-headers, which if present SHOULD be associated with the requested variant.
>
> If the client is a user agent, it SHOULD NOT change its document view from that which caused the request to be sent. This response

> The 204 response MUST NOT include a message-body, and thus is always terminated by the first empty line after the header fields.

I think the first sentence of the second paragraph says it best, "If the client is a user agent, it SHOULD NOT change its document view from that which caused the request to be sent". This is the case with an API. As compared to a **404**:

> The server has not found anything matching the Request-URI. No indication is given of whether the condition is temporary or permanent. The 410 (Gone) status code SHOULD be used if the server knows, through some internally configurable mechanism, that an old resource is permanently unavailable and has no forwarding address. This status code is commonly used when the server does not wish to reveal exactly why the request has been refused, or when no other response is applicable.

The primary difference being one is more applicable for an API and the other for the document view, i.e.; the page displayed.

edited Jan 9 '17 at 13:29

answered Jan 4 '17 at 13:45

David Pine

17.4k 5 57 90

Hi David, cheers. I realise that I can switch to return `IActionResult` - *but if this is the answer why does the ASP.NET Core API Controller support implicit type conversions at all if IActionResult is really required?* – Keith Jan 4 '17 at 14:19

If you really think about it, an API that claims to return a `Thing` that actually returns `null` is actually the non-body 204. If you want to return something that is more flexible the supported framework approach is to use `IActionResult` . It shouldn't be the framework's responsibility to assume anything, especially 404. But again, you have the ability to do it this way. As an alternative you could add custom middleware to set the `Response.StatusCode` if you'd like instead, but why go through the trouble? – David Pine Jan 4 '17 at 14:41

1 This should be the right answer...and about the status codes, there are different implementations(some uses one code, some uses another) based on what you want to accomplish and how...if you want to use a `404` on some cases and as long as your api is well documented, it shouldn't be a problem. – Hackerman Jan 11 '17 at 15:55

7 This shouldn't be the right answer. If you want to /get/{id} and there is no element of that id on the server, a 404 - not found is the correct answer to that. And regarding being explicitly typed - the type information of the method is actually used in tools like swagger who rely on the controllers declaration to be able to generate the correct swagger file. So IActionResult is missing a lot information in that case. – Sebastian P.R. Gingter Mar 3 '17 at 17:45

1 Good thing this wasn't accepted as the right answer. Thanks for commenting. :) – David Pine Mar 3 '17 at 17:48

In order to accomplish something like that(still, I think that the best approach should be using `IActionResult`), you can follow, where you can `throw` an `HttpResponseException` if your `Thing` is `null` :

```
public async Task<Thing> GetAsync(int id)
{
    Thing thingFromDB = await GetThingFromDBAsync();
    if(thingFromDB == null){
        throw new HttpResponseException(HttpStatusCode.NotFound); // This returns HTTP
404
    }
    // Process thingFromDB, blah blah blah
    return thing;
}
```

answered Jan 4 '17 at 20:17

Hackerman
11k 1 25 36

Cheers (+1) - I think this is a step in the right direction, but (on testing) `HttpResponseException` and the middleware to handle it don't appear to be in .NET Core 1.1. The next question is: should I roll my own middleware or is there an existing (ideally MS) package that already does this? – Keith Jan 5 '17 at 7:15

It looks like this was the way to do this in ASP.NET Web API 5, but it's been dropped in Core, which makes sense given's Core's manual approach. In most of the cases where Core has dropped a default ASP behaviour there's a new optional middleware that you can add in `Startup.Configure` , but there doesn't appear to be one here. In lieu of that it doesn't appear to be too difficult to write one from scratch. – Keith Jan 5 '17 at 11:17

Ok, I've fleshed out an answer based on this that does work, but still doesn't answer the original question: stackoverflow.com/a/41484262/905 – Keith Jan 5 '17 at 11:46

1 Take a look at this: github.com/aspnet/Mvc/issues/4311 – Hackerman Jan 5 '17 at 13:11

0

What you're doing with returning **"Explicit Types"** from the controller is isn't going to cooperate with your requirement to explicitly deal with your own **response code**. The simplest solution is to go with `IActionResult` (like others have suggested); However, you can also explicitly control your return type using the `[Produces]` filter.

Using `IActionResult` .

The way to get control over the status results, is you need to return a `IActionResult` which is where you can then take advantage of the `StatusCodeResult` type. However, now you've got your issue of wanting to force a partiular format...

The stuff below is taken from the Microsoft Document: Formatting Response Data -Forcing a Particular Format

> If you would like to restrict the response formats for a specific action you can, you can apply the `[Produces]` filter. The `[Produces]` filter specifies the response formats for a specific action (or controller). Like most Filters, this can be applied at the action, controller, or global scope.

Putting it all together

Here's an example of control over the `StatusCodeResult` in addition to control over the "explicit return type".

```
// GET: api/authors/search?namelike=foo
[Produces("application/json")]
[HttpGet("Search")]
public IActionResult Search(string namelike)
{
    var result = _authorRepository.GetByNameSubstring(namelike);
    if (!result.Any())
    {
        return NotFound(namelike);
    }
    return Ok(result);
}
```

I'm not a big supporter of this design pattern, but I have put those concerns in some additional answers to other people's questions. You will need to note that the `[Produces]` filter will require you to map it to the appropriate Formatter / Serializer / Explicit Type. You could take a look at this answer for more ideas or this one for implementing a more granular control over your ASP.NET Core Web API.

edited May 23 '17 at 12:10
Community ♦
1 1

answered Feb 25 '17 at 15:10
Svek
7,537 3 25 51

Cheers @Svek, that's interesting but a bit tangential. I'm not really concerned (in this question anyway) with the formatting possible with `IActionResult`, rather the need for it in the first place. – Keith Feb 27 '17 at 9:38