HttpClient.GetAsync(…) never returns when using await/async

Asked 8 years, 10 months ago Active 27 days ago Viewed 196k times

329

163

Edit: This question looks like it might be the same problem, but has no responses...

Edit: In test case 5 the task appears to be stuck in `WaitingForActivation` state.

I've encountered some odd behaviour using the System.Net.Http.HttpClient in .NET 4.5 - where "awaiting" the result of a call to (e.g.) `httpClient.GetAsync(...)` will never return.

This only occurs in certain circumstances when using the new async/await language functionality and Tasks API - the code always seems to work when using only continuations.

Here's some code which reproduces the problem - drop this into a new "MVC 4 WebApi project" in Visual Studio 11 to expose the following GET endpoints:

```
/api/test1
/api/test2
/api/test3
/api/test4
/api/test5 <--- never completes
/api/test6
```

Each of the endpoints here return the same data (the response headers from stackoverflow.com) except for `/api/test5` which never completes.

Have I encountered a bug in the HttpClient class, or am I misusing the API in some way?

Code to reproduce:

```
public class BaseApiController : ApiController
{
    /// <summary>
    /// Retrieves data using continuations
    /// </summary>
    protected Task<string> Continuations_GetSomeDataAsync()
    {
        var httpClient = new HttpClient();

        var t = httpClient.GetAsync("http://stackoverflow.com",
HttpCompletionOption.ResponseHeadersRead);

        return t.ContinueWith(t1 => t1.Result.Content.Headers.ToString());
    }

    /// <summary>
    /// Retrieves data using async/await
    /// </summary>
    protected async Task<string> AsyncAwait_GetSomeDataAsync()
    {
        var httpClient = new HttpClient();
        var result = await httpClient.GetAsync("http://stackoverflow.com",
HttpCompletionOption.ResponseHeadersRead);

        return result.Content.Headers.ToString();
    }
}

public class Test1Controller : BaseApiController
{
    /// <summary>
    /// Handles task using Async/Await
    /// </summary>
    public async Task<string> Get()
    {
        var data = await Continuations_GetSomeDataAsync();

        return data;
    }
}

public class Test2Controller : BaseApiController
{
    /// <summary>
    /// Handles task by blocking the thread until the task completes
    /// </summary>
    public string Get()
    {
        var task = Continuations_GetSomeDataAsync();

        var data = task.GetAwaiter().GetResult();

        return data;
    }
}

public class Test3Controller : BaseApiController
{
    /// <summary>
    /// Passes the task back to the controller host
    /// </summary>
    public Task<string> Get()
    {
        return Continuations_GetSomeDataAsync();
    }
}

public class Test4Controller : BaseApiController
{
    /// <summary>
    /// Handles task using Async/Await
    /// </summary>
    public async Task<string> Get()
    {
        var data = await AsyncAwait_GetSomeDataAsync();

        return data;
    }
}

public class Test5Controller : BaseApiController
{
    /// <summary>
    /// Handles task by blocking the thread until the task completes
    /// </summary>
    public string Get()
    {
        var task = AsyncAwait_GetSomeDataAsync();

        var data = task.GetAwaiter().GetResult();
```

```
            return data;
        }
    }

    public class Test6Controller : BaseApiController
    {
        /// <summary>
        /// Passes the task back to the controller host
        /// </summary>
        public Task<string> Get()
        {
            return AsyncAwait_GetSomeDataAsync();
        }
    }
```

c# .net asynchronous async-await dotnet-httpclient

Share Edit Follow

edited May 23 '17 at 12:18
Community ♦
1 1

asked Apr 27 '12 at 1:28
Benjamin Fox
4,914 5 17 19

2 It doesn't appear to be the same issue, but just to make sure you know about it, there's an MVC4 bug in the beta WRT async methods that complete synchronously - see stackoverflow.com/questions/9627329/… – James Manning Apr 27 '12 at 1:45

Thanks - I'll watch out for that. In this case I think that the method should always be asynchronous because of the call to `HttpClient.GetAsync(...)` ? – Benjamin Fox Apr 27 '12 at 1:59

7 Answers

Active Oldest Votes

481

You are misusing the API.

Here's the situation: in ASP.NET, only one thread can handle a request at a time. You can do some parallel processing if necessary (borrowing additional threads from the thread pool), but only one thread would have the request context (the additional threads do not have the request context).

This is managed by the ASP.NET `SynchronizationContext`.

By default, when you `await` a `Task`, the method resumes on a captured `SynchronizationContext` (or a captured `TaskScheduler`, if there is no `SynchronizationContext`). Normally, this is just what you want: an asynchronous controller action will `await` something, and when it resumes, it resumes with the request context.

So, here's why `test5` fails:

- `Test5Controller.Get` executes `AsyncAwait_GetSomeDataAsync` (within the ASP.NET request context).
- `AsyncAwait_GetSomeDataAsync` executes `HttpClient.GetAsync` (within the ASP.NET request context).

- The HTTP request is sent out, and `HttpClient.GetAsync` returns an uncompleted `Task`.
- `AsyncAwait_GetSomeDataAsync` awaits the `Task`; since it is not complete, `AsyncAwait_GetSomeDataAsync` returns an uncompleted `Task`.
- `Test5Controller.Get` **blocks** the current thread until that `Task` completes.
- The HTTP response comes in, and the `Task` returned by `HttpClient.GetAsync` is completed.
- `AsyncAwait_GetSomeDataAsync` attempts to resume within the ASP.NET request context. However, there is already a thread in that context: the thread blocked in `Test5Controller.Get`.
- Deadlock.

Here's why the other ones work:

- (`test1`, `test2`, and `test3`): `Continuations_GetSomeDataAsync` schedules the continuation to the thread pool, *outside* the ASP.NET request context. This allows the `Task` returned by `Continuations_GetSomeDataAsync` to complete without having to re-enter the request context.
- (`test4` and `test6`): Since the `Task` is *awaited*, the ASP.NET request thread is not blocked. This allows `AsyncAwait_GetSomeDataAsync` to use the ASP.NET request context when it is ready to continue.

And here's the best practices:

1. In your "library" `async` methods, use `ConfigureAwait(false)` whenever possible. In your case, this would change `AsyncAwait_GetSomeDataAsync` to be `var result = await httpClient.GetAsync("http://stackoverflow.com", HttpCompletionOption.ResponseHeadersRead).ConfigureAwait(false);`
2. Don't block on `Task`s; it's `async` all the way down. In other words, use `await` instead of `GetResult` (`Task.Result` and `Task.Wait` should also be replaced with `await`).

That way, you get both benefits: the continuation (the remainder of the `AsyncAwait_GetSomeDataAsync` method) is run on a basic thread pool thread that doesn't have to enter the ASP.NET request context; and the controller itself is `async` (which doesn't block a request thread).

More information:

- My `async` / `await` intro post, which includes a brief description of how `Task` awaiters use `SynchronizationContext`.
- The Async/Await FAQ, which goes into more detail on the contexts. Also see Await, and UI, and deadlocks! Oh, my! which *does* apply here even though you're in ASP.NET rather than a UI, because the ASP.NET `SynchronizationContext` restricts the request context to just one thread at a time.
- This MSDN forum post.
- Stephen Toub demos this deadlock (using a UI), and so does Lucian Wischik.

Update 2012-07-13: Incorporated this answer into a blog post.

Share Edit Follow

edited Feb 21 '20 at 21:43
Callum Watkins
2,415 2 26 42

answered Apr 27 '12 at 13:20
Stephen Cleary
366k 68 595 718

2 Is there some documentation for the ASP.NET `SynchroniztaionContext` that explains that there can be only one thread in the context for some request? If not, I think there should be. – svick Apr 27 '12 at 13:27

8 It is not documented anywhere AFAIK. – Stephen Cleary Apr 27 '12 at 13:50

10 Thanks - **awesome** response. The difference in behaviour between (apparently) functionally identical code is frustrating but makes sense with your explanation. It would be useful if the framework was able to detect such deadlocks and raise an exception somewhere. – Benjamin Fox Apr 28 '12 at 1:46

3 Are there situations where using .ConfigureAwait(false) in an asp.net context is NOT recommended? It would seem to me that it should always be used and that its only in a UI context that it should not be used since you need to sync to the UI. Or am I missing the point? – AlexGad May 9 '12 at 16:50

3 The ASP.NET `SynchronizationContext` does provide some important functionality: it flows the request context. This includes all kinds of stuff from authentication to cookies to culture. So in ASP.NET, instead of syncing back to the UI, you sync back to the request context. This may change shortly: the new `ApiController` does have an `HttpRequestMessage` context as a property - so it *may* not be required to flow the context through `SynchronizationContext` - but I don't yet know. – Stephen Cleary May 10 '12 at 1:56

63

Edit: Generally try to avoid doing the below except as a last ditch effort to avoid deadlocks. Read the first comment from Stephen Cleary.

Quick fix from here. Instead of writing:

```
Task tsk = AsyncOperation();
tsk.Wait();
```

Try:

```
Task.Run(() => AsyncOperation()).Wait();
```

Or if you need a result:

```
var result = Task.Run(() => AsyncOperation()).Result;
```

From the source (edited to match the above example):

> AsyncOperation will now be invoked on the ThreadPool, where there won't be a SynchronizationContext, and the continuations used inside of AsyncOperation won't be forced back to the invoking thread.

For me this looks like a useable option since I do not have the option of making it async all the way (which I would prefer).

From the source:

> Ensure that the await in the FooAsync method doesn't find a context to marshal back to. The simplest way to do that is to invoke the asynchronous work from the ThreadPool, such as by wrapping the invocation in a Task.Run, e.g.
>
> int Sync() { return Task.Run(() => Library.FooAsync()).Result; }
>
> FooAsync will now be invoked on the ThreadPool, where there won't be a SynchronizationContext, and the continuations used inside of FooAsync won't be forced back to the thread that's invoking Sync().

Share Edit Follow edited Feb 15 '18 at 10:07 answered Nov 1 '13 at 13:23

Ykok
971 10 14

7 Might want to re-read your source link; the author recommends *not* doing this. Does it work? Yes, but only in the sense that you avoid deadlock. This solution negates *all* the benefits of `async` code on ASP.NET, and in fact can cause problems at scale. BTW, `ConfigureAwait` does not "break proper async behavior" in any scenario; it's exactly what you should use in library code. – Stephen Cleary Nov 28 '13 at 19:15

2 It's the entire first section, titled in bold `Avoid Exposing Synchronous Wrappers for Asynchronous Implementations`. The entire rest of the post is explaining a few different ways to do it *if* you absolutely *need* to. – Stephen Cleary Nov 29 '13 at 21:24

1 Added the section I found in the source - I'll leave it up to future readers to decide. Note that you should generally try to avoid doing this and only do it as a last ditch resort (ie. when using async code you do not have control over). – Ykok Dec 5 '13 at 13:22

3 I like all the answers here and as always.... they are all based on context (pun intended lol). I am wrapping HttpClient's Async calls with a synchronous version so I can't change that code to add ConfigureAwait to that library. So to prevent the deadlocks in production, I am wrapping the Async calls in a Task.Run. So as I understand it, this is going to use 1 extra thread per request and avoids the deadlock. I assume that to be completely compliant, I need to use WebClient's sync methods. That is a lot of work to justify so I'll need a compelling reason not stick with my current approach. – samneric May 8 '14 at 2:20

1 I ended up creating an Extension Method to convert Async to Sync. I read on here somewhere its the same way the .Net framework does it: public static TResult RunSync<TResult>(this Func<Task<TResult>> func) { return _taskFactory .StartNew(func) .Unwrap() .GetAwaiter() .GetResult(); } – samneric May 19 '17 at 20:01

12

Since you are using `.Result` or `.Wait` or `await` this will end up causing a **deadlock** in your code.

you can use `ConfigureAwait(false)` in `async` methods for **preventing deadlock**

like this:

```
var result = await httpClient.GetAsync("http://stackoverflow.com",
HttpCompletionOption.ResponseHeadersRead)
                             .ConfigureAwait(false);
```

you can use `ConfigureAwait(false)` wherever possible for Don't Block Async Code .

Share Edit Follow

edited Jan 25 '19 at 19:49
ΩmegaMan
22.1k 8 73 94

answered Jan 20 '18 at 15:12
Hasan Fathi
3,818 2 30 44

2

These two schools are not really excluding.

Here is the scenario where you simply have to use

```
Task.Run(() => AsyncOperation()).Wait();
```

or something like

```
AsyncContext.Run(AsyncOperation);
```

I have a MVC action that is under database transaction attribute. The idea was (probably) to roll back everything done in the action if something goes wrong. This does not allow context switching, otherwise transaction rollback or commit is going to fail itself.

The library I need is async as it is expected to run async.

The only option. Run it as a normal sync call.

I am just saying to each its own.

Share Edit Follow

edited May 17 '17 at 17:18

answered Feb 8 '17 at 8:19
alex.peter
164 6

so you're suggesting the first option in your answer? – Don Cheadle May 16 '17 at 22:44

> "The library I need is async as it is expected to run async." That is the real problem. – Redwolf Jan 6 at 8:39

1

In my case 'await' never finished because of exception while executing the request, e.g. server not responding, etc. Surround it with try..catch to identify what happened, it'll also complete your 'await' gracefully.

```
public async Task<Stuff> GetStuff(string id)
{
    string path = $"/api/v2/stuff/{id}";
    try
    {
        HttpResponseMessage response = await client.GetAsync(path);
        if (response.StatusCode == HttpStatusCode.OK)
        {
            string json = await response.Content.ReadAsStringAsync();
            return JsonUtility.FromJson<Stuff>(json);
        }
        else
        {
            Debug.LogError($"Could not retrieve stuff {id}");
        }
    }
    catch (Exception exception)
    {
        Debug.LogError($"Exception when retrieving stuff {exception}");
    }
    return null;
}
```

Share Edit Follow edited Jan 8 at 17:12 answered Oct 26 '20 at 16:16
Alexey Podlasov
895 10 15

This is an overlooked answer. Sometimes the error isn't apparent. For example, if you initialize a variable in OnIntialised the render engine can throw an error if it tried to build from that variable, as the thread is still awaiting, then the whole thing falls over with no error anywhere. This answered a problem I was looking at for weeks. – Tod Dec 16 '20 at 10:14

1

I'm going to put this in here more for completeness than direct relevance to the OP. I spent nearly a day debugging an `HttpClient` request, wondering why I was never getting back a response.

Finally found that I had forgotten to `await` the `async` call further down the call stack.

Feels about as good as missing a semicolon.

Share Edit Follow answered May 21 '19 at 19:05
Bondolin
2,151 5 24 48

0

I was using to many await, so i was not getting response , i converted in to sync call its started working

```
                using (var client = new HttpClient())
                using (var request = new HttpRequestMessage())
                {
                    client.DefaultRequestHeaders.Accept.Add(new 
    MediaTypeWithQualityHeaderValue("application/json"));
                    request.Method = HttpMethod.Get;
                    request.RequestUri = new Uri(URL);
                    var response = client.GetAsync(URL).Result;
                    response.EnsureSuccessStatusCode();
                    string responseBody = response.Content.ReadAsStringAsync().Result;
```

Share Edit Follow answered Feb 3 at 12:35
Shahid Islam
400 2 6