How to safely call an async method in C# without await

Asked 6 years, 4 months ago Active today Viewed 145k times

265

75

I have an `async` method which returns no data:

```
public async Task MyAsyncMethod()
{
    // do some stuff async, don't return any data
}
```

I'm calling this from another method which returns some data:

```
public string GetStringData()
{
    MyAsyncMethod(); // this generates a warning and swallows exceptions
    return "hello world";
}
```

Calling `MyAsyncMethod()` without awaiting it causes a "Because this call is not awaited, the current method continues to run before the call is completed" warning in visual studio. On the page for that warning it states:

> You should consider suppressing the warning only if you're sure that you don't want to wait for the asynchronous call to complete and that **the called method won't raise any exceptions**.

I'm sure I don't want to wait for the call to complete; I don't need to or have the time to. **But the call *might* raise exceptions.**

I've stumbled into this problem a few times and I'm sure it's a common problem which must have a common solution.

How do I safely call an async method without awaiting the result?

Update:

For people suggesting that I just await the result, this is code that is responding to a web request on our web service (ASP.NET Web API). Awaiting in a UI context keeps the UI thread free, but awaiting in a web request call will wait for the Task to finish before responding to the request, thereby increasing response times with no reason.

c# exception-handling async-await

edited Mar 20 '13 at 13:17

asked Mar 20 '13 at 11:59

George Powell
3,821 6 27 38

Why not just create a completion method and just ignore it there? Because if it is running on background thread. Then it won't stop your program from terminating anyway. – Yahya Mar 20 '13 at 12:02

1 If you don't want to wait for the result, the only option is to ignore/suppress the warning. If you *do* want to wait for the result/exception then `MyAsyncMethod().Wait()` – Peter Ritchie Mar 20 '13 at 12:42

1 About your edit: that does not make sense to me. Say the response is sent to the client 1 sec after the request, and 2 secs later your async method throws an exception. What would you do with that exception? You cannot send it to the client, if your response is already sent. What else would you do with it? – user743382 Mar 20 '13 at 13:02

2 @Romoku Fair enough. Assuming someone looks at the log, anyway. :) – user743382 Mar 20 '13 at 13:11

2 A variation on the ASP.NET Web API scenario is a *self-hosted* Web API in a long-lived process (like, say, a Windows service), where a request creates a lengthy background task to do something expensive, but still wants to get a response quickly with an HTTP 202 (Accepted). – David Rubin Jun 23 '15 at 17:44

8 Answers

164

If you want to get the exception "asynchronously", you could do:

```
MyAsyncMethod().
  ContinueWith(t => Console.WriteLine(t.Exception),
      TaskContinuationOptions.OnlyOnFaulted);
```

This will allow you to deal with an exception on a thread other than the "main" thread. This means you don't have to "wait" for the call to `MyAsyncMethod()` from the thread that calls `MyAsyncMethod` ; but, still allows you to do something with an exception--but only if an exception occurs.

Update:

technically, you could do something similar with `await` :

```
try
{
    await MyAsyncMethod().ConfigureAwait(false);
}
catch (Exception ex)
{
    Trace.WriteLine(ex);
}
```

...which would be useful if you needed to specifically use `try` / `catch` (or `using`) but I find the `ContinueWith` to be a little more explicit because you have to know what `ConfigureAwait(false)` means.

edited Dec 9 '16 at 16:59

answered Mar 20 '13 at 12:59

Peter Ritchie
30.6k 9 71 92

I've accepted this as the answer but also see my answer below and Stephen Cleary's point specific to doing this in ASP.NET – George Powell Mar 20 '13 at 16:10

7 I turned this into an extension method on `Task` : public static class AsyncUtility { public static void PerformAsyncTaskWithoutAwait(this Task task, Action<Task> exceptionHandler) { var dummy = task.ContinueWith(t => exceptionHandler(t), TaskContinuationOptions.OnlyOnFaulted); } } Usage: MyAsyncMethod().PerformAsyncTaskWithoutAwait(t => log.ErrorFormat("An error occurred while calling MyAsyncMethod:\n{0}", t.Exception)); – Mark Avenius Jun 24 '15 at 14:13

1 downvoter. Comments? If there's something wrong in the answer, I'd love to know and/or fix. – Peter Ritchie Feb 1 '17 at 19:48

1 "wait" isn't the correct description in this context. The line after the `ConfiguratAwait(false)` doesn't execute until the task completes, but the current thread doesn't "wait" (i.e. block) for that, the next line is called asynchronously to the await invocation. Without the `ConfigureAwait(false)` that next line would be executed on the original web request context. With the `ConfigurateAwait(false)` it's executed in the same context as the async method (task), freeing the original context/thread to continue on... – Peter Ritchie May 18 '17 at 13:32

7 The ContinueWith version is not the same as the try{ await }catch{} version. In the first version, everything after ContinueWith() will execute immediately. The initial task is fired and forgotten. In the second version, everything after the catch{} will execute only after the initial task is completed. The second version is equivalent to "await MyAsyncMethod().ContinueWith(t => Console.WriteLine(t.Exception), TaskContinuationOptions.OnlyOnFaulted).ConfigureAwait(fals); – Thanasis Ioannidis Oct 12 '17 at 9:59

51

You should first consider making `GetStringData` an `async` method and have it `await` the task returned from `MyAsyncMethod`.

If you're absolutely sure that you don't need to handle exceptions from `MyAsyncMethod` *or* know when it completes, then you can do this:

```
public string GetStringData()
{
  var _ = MyAsyncMethod();
  return "hello world";
}
```

BTW, this is not a "common problem". It's very rare to want to execute some code and not care whether it completes *and* not care whether it completes successfully.

Update:

Since you're on ASP.NET and wanting to return early, you may find my blog post on the subject useful. However, ASP.NET was not designed for this, and there's no *guarantee* that your code will run after the response is returned. ASP.NET will do its best to let it run, but it can't guarantee it.

So, this is a fine solution for something simple like tossing an event into a log where it doesn't *really* matter if you lose a few here and there. It's not a good solution for any kind of business-critical operations. In those situations, you *must* adopt a more complex architecture, with a persistent way to save the operations (e.g., Azure Queues, MSMQ) and a separate background process (e.g., Azure Worker Role, Win32 Service) to process them.

edited Mar 20 '13 at 13:23

answered Mar 20 '13 at 12:39

Stephen Cleary
304k 50 499 631

2 I think you might have misunderstood. I **do** care if it throws exceptions and fails, but I don't want to have to await the method before returning my data. Also see my edit about the context I'm working in if that makes any difference. – George Powell Mar 20 '13 at 13:02

4 @GeorgePowell: It's very dangerous to have code running in an ASP.NET context without an active request. I have a blog post that may help you out, but without knowing more of your problem I can't say whether I'd recommend that approach or not. – Stephen Cleary Mar 20 '13 at 13:17

@StephenCleary I have a similar need. In my example, I have/need a batch processing engine to run in the cloud, I'll "ping" the end point to kick off batch processing, but I want to return immediately. Since pinging it gets it started, it can handle everything from there. If there are exceptions that are thrown, then they'd just be logged in my "BatchProcessLog/Error" tables... – ganders Aug 16 '16 at 18:31

2 In C#7, you can replace `var _ = MyAsyncMethod();` with `_ = MyAsyncMethod();` . This still avoids warning CS4014, but it makes it a bit more explicit that you're not using the variable. – Brian Jan 19 '18 at 23:09

The answer by Peter Ritchie was what I wanted, and Stephen Cleary's article about returning early in ASP.NET was very helpful.

44

As a more general problem however (not specific to an ASP.NET context) the following Console application demonstrates the usage and behavior of Peter's answer using `Task.ContinueWith(...)`

```
static void Main(string[] args)
{
  try
  {
    // output "hello world" as method returns early
    Console.WriteLine(GetStringData());
  }
  catch
  {
    // Exception is NOT caught here
  }
  Console.ReadLine();
}

public static string GetStringData()
{
  MyAsyncMethod().ContinueWith(OnMyAsyncMethodFailed,
TaskContinuationOptions.OnlyOnFaulted);
  return "hello world";
}

public static async Task MyAsyncMethod()
{
  await Task.Run(() => { throw new Exception("thrown on background thread"); });
}

public static void OnMyAsyncMethodFailed(Task task)
{
  Exception ex = task.Exception;
  // Deal with exceptions here however you want
}
```

`GetStringData()` returns early without awaiting `MyAsyncMethod()` and exceptions thrown in `MyAsyncMethod()` are dealt with in `OnMyAsyncMethodFailed(Task task)` and **not** in the `try` / `catch` around `GetStringData()`

edited Mar 20 '13 at 16:14

answered Mar 20 '13 at 15:49

George Powell

3,821 6 27 38

6 Remove `Console.ReadLine();` and add a little sleep/delay in `MyAsyncMethod` and you'll never see the exception. – tymtam Nov 23 '16 at 22:57

Like It @George! – sebu Apr 29 '18 at 7:42

I end up with this solution :

16

```
public async Task MyAsyncMethod()
{
    // do some stuff async, don't return any data
}

public string GetStringData()
{
    // Run async, no warning, exception are catched
    RunAsync(MyAsyncMethod());
    return "hello world";
}

private void RunAsync(Task task)
{
    task.ContinueWith(t =>
    {
        ILog log = ServiceLocator.Current.GetInstance<ILog>();
        log.Error("Unexpected Error", t.Exception);

    }, TaskContinuationOptions.OnlyOnFaulted);
}
```

answered Mar 15 '16 at 18:22

Filimindji

808 1 10 22

This is called fire and forget, and there is an extension for that.

4

> Consumes a task and doesn't do anything with it. Useful for fire-and-forget calls to async methods within async methods.

Install nuget package.

Use:

```
MyAsyncMethod().Forget();
```

edited Jul 1 at 9:46

answered Jan 21 at 9:51

wast

390 4 15

1 It doesn't do anything, it's just a trick to remove the warning. See github.com/microsoft/vs-threading/blob/master/src/… – Paolo Fulgoni Jun 28 at 12:25

2

I guess the question arises, why would you need to do this? The reason for `async` in C# 5.0 is so you can await a result. This method is not actually asynchronous, but simply called at a time so as not to interfere too much with the current thread.

Perhaps it may be better to start a thread and leave it to finish on its own.

edited Mar 20 '13 at 12:06

Drew Noakes

197k 125 543 637

answered Mar 20 '13 at 12:02

rhughes

5,951 6 46 72

2 `async` is a bit more than just "awaiting" a result. "await" implies that the lines following "await" are executed asynchronously on the same thread that invoked "await". This can be done without "await", of course, but you end up having a bunch of delegates and lose the sequential look-and-feel of the code (as well as the ability to use `using` and `try/catch` ... – Peter Ritchie Mar 20 '13 at 13:09

1 @PeterRitchie You can have a method that is functionally asynchronous, doesn't use the `await` keyword, and doesn't use the `async` keyword, but there is no point whatsoever in using the `async` keyword without also using `await` in the definition of that method. – Servy Mar 20 '13 at 15:56

1 @PeterRitchie From the statement: " `async` is a bit more than just "awaiting" a result." The `async` keyword (you implied it's the keyword by enclosing it in backticks) means *nothing* more than awaiting the result. It's asynchrony, as the general CS concepts, that means more than just awaiting a result. – Servy Mar 20 '13 at 16:35

1 @Servy `async` creates a state machine that manages any awaits within the async method. If there are no `await` s within the method it still creates that state machine--but the method is not asynchronous. And if the `async` method returns `void` , there's nothing to await. So, it's *more* than just awaiting a result. – Peter Ritchie Mar 20 '13 at 19:43

1 @PeterRitchie As long as the method returns a task you can await it. There is no need for the state machine, or the `async` keyword. All you'd need to do (and all that really happens in the end with a state machine in that special case) is that the method is run synchronously and then wrapped in a completed task. I suppose technically you don't just remove the state machine; you remove the state machine and then call `Task.FromResult` . I assumed you (and also the compiler writers) could add the addendum on your own. – Servy Mar 20 '13 at 23:25

The solution is start the HttpClient into another execution task without sincronization context:

1

```
var submit = httpClient.PostAsync(uri, new StringContent(body,
Encoding.UTF8,"application/json"));
var t = Task.Run(() => submit.ConfigureAwait(false));
await t.ConfigureAwait(false);
```

answered 13 hours ago

Ernesto Garcia

11 1

New contributor

0

On technologies with message loops (not sure if ASP is one of them), you can block the loop and process messages until the task is over, and use ContinueWith to unblock the code:

```
public void WaitForTask(Task task)
{
    DispatcherFrame frame = new DispatcherFrame();
    task.ContinueWith(t => frame.Continue = false));
    Dispatcher.PushFrame(frame);
}
```

This approach is similar to blocking on ShowDialog and still keeping the UI responsive.

answered Jan 16 at 6:34

haimb

161 3