async/await - when to return a Task vs void?

420

132

Under what scenarios would one want to use

```
public async Task AsyncMethod(int num)
```

instead of

```
public async void AsyncMethod(int num)
```

The only scenario that I can think of is if you need the task to be able to track its progress.

Additionally, in the following method, are the async and await keywords unnecessary?

```
public static async void AsyncMethod2(int num)
{
    await Task.Factory.StartNew(() => Thread.Sleep(num));
}
```

c# asynchronous .net-4.5

edited Sep 14 '18 at 15:21

French Boiethios
12.3k 4 44 86

asked Aug 27 '12 at 14:33

user981225
3,075 5 24 35

19 Note that async methods should **always** be suffixed with the name Async.Example `Foo()` would become `FooAsync()` . – Fred Apr 15 '16 at 20:27

25 @Fred Mostly, but not always. This is just the convention and the accepted exceptions to this convention are with event based classes or interface contracts, see MSDN. For example, you shouldn't rename common event handlers, such as Button1_Click. – Ben Jun 16 '16 at 12:09

11 Just a note you shouldn't using `Thread.Sleep` with your tasks you should `await Task.Delay(num)` instead – Bob Vale Apr 7 '17 at 11:01

Join Stack Overflow to learn, share knowledge, and build your career.

Sign up with email | Sign up with Google | Sign up with Facebook

because it was using an Entity Framework context object declared as a member of the controller was disposed before the method finished to execute. The framework disposed the controller before its method finished to execute. I changed the method to async Task and it worked. – costa May 26 '17 at 17:51

6 Answers

361

1) Normally, you would want to return a `Task`. The main exception should be when you *need* to have a `void` return type (for events). If there's no reason to disallow having the caller `await` your task, why disallow it?

2) `async` methods that return `void` are special in another aspect: they represent *top-level async operations*, and have additional rules that come into play when your task returns an exception. The easiest way is to show the difference is with an example:

```
static async void f()
{
    await h();
}

static async Task g()
{
    await h();
}

static async Task h()
{
    throw new NotImplementedException();
}

private void button1_Click(object sender, EventArgs e)
{
    f();
}

private void button2_Click(object sender, EventArgs e)
{
    g();
}
```

Join Stack Overflow to learn, share knowledge, and build your career.

Sign up with email | Sign up with Google | Sign up with Facebook

`f` 's exception is always "observed". An exception that leaves a top-level asynchronous method is simply treated like any other unhandled exception. `g` 's exception is never observed. When the garbage collector comes to clean up the task, it sees that the task resulted in an exception, and nobody handled the exception. When that happens, the `TaskScheduler.UnobservedTaskException` handler runs. You should never let this happen. To use your example,

```
public static async void AsyncMethod2(int num)
{
    await Task.Factory.StartNew(() => Thread.Sleep(num));
}
```

Yes, use `async` and `await` here, they make sure your method still works correctly if an exception is thrown.

for more information see: http://msdn.microsoft.com/en-us/magazine/jj991977.aspx

edited Jun 3 '14 at 12:07

suizo

319 5 20

answered Aug 27 '12 at 14:53

user743382

10 I meant `f` instead of `g` in my comment. The exception from `f` is passed to the `SynchronizationContext` . `g` will raise `UnobservedTaskException` , but `UTE` no longer crashes the process if it's not handled. There are some situations where it's acceptable to have "asynchronous exceptions" like this that are ignored. – Stephen Cleary Aug 28 '12 at 13:07

3 If you have `WhenAny` with multiple `Task` s resulting in exceptions. You often only have to handle the first one, and you often want to ignore the others. – Stephen Cleary Aug 28 '12 at 13:29

1 @StephenCleary Thanks, I guess that's a good example, although it depends on the reason you call `WhenAny` in the first place whether it's okay to ignore the other exceptions: the main use case I have for it still ends up awaiting the remaining tasks when any finishes, with or without an exception. – user743382 Aug 28 '12 at 13:41

7 I'm a little confused as to why you recommend returning a Task instead of void. Like you said, f() will throw and exception but g() will not. Isn't it best to be made aware of these background thread exceptions? – user981225 Aug 28 '12 at 17:41

2 @user981225 Indeed, but that then becomes the responsibility of g's caller: every method that calls g should be async and use await too. This is a guideline, not a hard rule, you can decide that in your specific program, it's easier to have g return void. – user743382 Aug 28 '12 at 20:12

Join Stack Overflow to learn, share knowledge, and build your career.

Sign up with email | Sign up with Google | Sign up with Facebook

The reason behind this is the Synchronization Context used by the AsyncVoidMethodBuilder, being none in this example. When there is no ambient Synchronization Context, any exception that is unhandled by the body of an async void method is rethrown on the ThreadPool. While there is seemingly no other logical place where that kind of unhandled exception could be thrown, the unfortunate effect is that the process is being terminated, because unhandled exceptions on the ThreadPool effectively terminate the process since .NET 2.0. You may intercept all unhandled exception using the AppDomain.UnhandledException event, but there is no way to recover the process from this event.

When writing UI event handlers, async void methods are somehow painless because exceptions are treated the same way found in non-async methods; they are thrown on the Dispatcher. There is a possibility to recover from such exceptions, with is more than correct for most cases. Outside of UI event handlers however, async void methods are somehow dangerous to use and may not that easy to find.

edited Apr 19 at 10:41

Community ♦
1 1

answered Apr 11 '14 at 15:40

Davide Icardi
8,857 6 43 61

24

I got clear idea from this statements.

1. Async void methods have different error-handling semantics. When an exception is thrown out of an async Task or async Task method, that exception is captured and placed on the Task object. With async void methods, there is no Task object, so any exceptions thrown out of an async void method will be raised directly on the SynchronizationContext(SynchronizationContext represents a location "where" code might be executed.) that was active when the async void method started

Exceptions from an Async Void Method Can’t Be Caught with Catch

```
private async void ThrowExceptionAsync()
{
  throw new InvalidOperationException();
}
public void AsyncVoidExceptions_CannotBeCaughtByCatch()
{
  try
  {
    ThrowExceptionAsync();
```

Join Stack Overflow to learn, share knowledge, and build your career.

Sign up with email | Sign up with Google | Sign up with Facebook

```
    }
}
```

These exceptions can be observed using AppDomain.UnhandledException or a similar catch-all event for GUI/ASP.NET applications, but using those events for regular exception handling is a recipe for unmaintainability(it crashes the application).

2. Async void methods have different composing semantics. Async methods returning Task or Task can be easily composed using await, Task.WhenAny, Task.WhenAll and so on. Async methods returning void don't provide an easy way to notify the calling code that they've completed. It's easy to start several async void methods, but it's not easy to determine when they've finished. Async void methods will notify their SynchronizationContext when they start and finish, but a custom SynchronizationContext is a complex solution for regular application code.
3. Async Void method useful when using synchronous event handler because they raise their exceptions directly on the SynchronizationContext, which is similar to how synchronous event handlers behave

For more details check this link https://msdn.microsoft.com/en-us/magazine/jj991977.aspx

edited May 25 '17 at 9:07

answered May 25 '17 at 8:39

Nayas Subramanian

1,372 12 20

10

The problem with calling async void is that you don't even get the task back, you have no way of knowing when the function's task has completed (see https://blogs.msdn.microsoft.com/oldnewthing/20170720-00/?p=96655)

Here are the three ways to call an async function:

```
async Task<T> SomethingAsync() { ... return t; }
async Task SomethingAsync() { ... }
async void SomethingAsync() { ... }
```

> In all the cases, the function is transformed into a chain of tasks. The difference is what the function returns.
>
> In the first case, the function returns a task that eventually produces the t.

Join Stack Overflow to learn, share knowledge, and build your career.

Sign up with email | Sign up with Google | Sign up with Facebook

> The third case is the nasty one. The third case is like the second case, except that you don't even get the task back. You have no way of knowing when the function's task has completed.
>
> The async void case is a "fire and forget": You start the task chain, but you don't care about when it's finished. When the function returns, all you know is that everything up to the first await has executed. Everything after the first await will run at some unspecified point in the future that you have no access to.

answered Jun 19 '18 at 15:31

user8128167
3,105 5 38 54

5

I think you can use `async void` for kicking off background operations as well, so long as you're careful to catch exceptions. Thoughts?

```
class Program {

    static bool isFinished = false;

    static void Main(string[] args) {

        // Kick off the background operation and don't care about when it completes
        BackgroundWork();

        Console.WriteLine("Press enter when you're ready to stop the background
operation.");
        Console.ReadLine();
        isFinished = true;
    }

    // Using async void to kickoff a background operation that nobody wants to be
notified about when it completes.
    static async void BackgroundWork() {
        // It's important to catch exceptions so we don't crash the appliation.
        try {
            // This operation will end after ten interations or when the app closes.
Whichever happens first.
            for (var count = 1; count <= 10 && !isFinished; count++) {
                await Task.Delay(1000);
```

Join Stack Overflow to learn, share knowledge, and build your career.

Sign up with email | Sign up with Google | Sign up with Facebook

```
            }
        }
    }
```

answered Feb 16 '18 at 12:43

bboyle1234
4,094 2 17 19

The problem with calling async void is that you don't even get the task back, you have no way of knowing when the function's task has completed – user8128167 Jun 19 '18 at 15:27

This does make sense for (rare) background operations where you don't care about the result and you don't want an exception to affect other operations. A typical use case would be sending a log event to a log server. You want this to happen in the background, and you don't want your service / request handler to fail if the log server is down. Of course, you have to catch all exceptions, or your process will be terminated. Or is there a better way to achieve this? – Florian Winter Aug 27 '18 at 8:57

Exceptions from an Async Void Method Can't Be Caught with Catch. msdn.microsoft.com/en-us/magazine/jj991977.aspx – Si Zi Oct 16 '18 at 11:43

1 @SiZi the catch is INSIDE the async void method as shown in the example and it is caught. – bboyle1234 Oct 16 '18 at 21:34

0

My answer is simple you cannon await void method Error CS4008 Cannot await 'void' TestAsync e:\test\TestAsync\TestAsync\Program.cs

So if the method is async it is better to be awaitable, because you can loose async advantage.

answered Sep 20 '18 at 16:21

Serg Sh
316 2 11

Join Stack Overflow to learn, share knowledge, and build your career.

Sign up with email | Sign up with Google | Sign up with Facebook