async void – How to Tame the Asynchronous Nightmare

Dev Leader
7 Feb 2023 CPOL

You're an intermediate dotnet programmer and you mostly know your way around using Tasks. You sprinkle async and await through your code, and everything is working just as expected.

You're an intermediate dotnet programmer and you mostly know your way around using Tasks. You sprinkle async and await through your code, and everything is working just as expected. You've heard time and time again that you always want the return types of your asynchronous methods to be a Task (or Task<T>) and that async void is essentially the root of all evil. No sweat.

One day you go to wire up an event handler using the syntax myObject.SomeEvent += SomeEventHandler, and your event handler needs to await some asynchronous code. You take all of the right steps and change your method signature to get that beautiful async Task added in, replacing void. But suddenly you get a compile error about your event handler not being compatible.

You feel trapped. You're scared. And then you do the unspeakable…

You change your method signature for your event handler to async void and suddenly all of your compilation problems disappear right before your eyes. You hear the voices of all of the legendary dotnet programmers you look up to echoing in your mind: "What are you doing?! You cannot commit such a heinous coding crime!". But it's too late. You've succumbed to the powers of async void and all of your problems have appeared to vanish.

That is, until one day it all falls apart. And that's when you did a search on the Internet and found this article.

Welcome, friend.

A Companion Video!

What’s Actually Wrong With async void?

Let’s start here with the basics. Here are a few of dangers of using async void in your C# code:

- Exceptions thrown in an “async void” method cannot be caught in the same way as they can in an “async Task” method. When an exception is thrown in an “async void” method, it will be raised on the Synchronization Context, which can cause the application to crash.
- Because “async void” methods cannot be awaited, they can lead to confusing and hard-to-debug code. For example, if an “async void” method is called multiple times, it can lead to multiple instances of the method running concurrently, which can cause race conditions and other unexpected behavior.

I’d be willing to bet you’re here because of the first point. Your application is experiencing weird crashes, and you’re having a heck of a time diagnosing the issue and getting traces. We are unable to (traditionally) wrap calls to async void methods in try/catch blocks and actually have them work as we’d expect.

I wanted to demonstrate this with a simple bit of code that you can literally try out in your browser. Let’s discuss the code below (which, by the way, is using the top level statements feature of .NET 7.0 so if you’re not used to seeing a C# program without a static void main… don’t be shocked!):

C#

```
using System;
using System.Threading;
using System.Threading.Tasks;
```

```
Console.WriteLine("Start");
try
{
    // NOTE: uncomment the single line for each one of the scenarios below one at a time to
try it out!

    // Scenario 1: we can await an async Task which allows us to catch exceptions
    //await AsyncTask();

    // Scenario 2: we cannot await an async void and as a result we cannot catch the
exception
    //AsyncVoid();

    // Scenario 3: we purposefully wrap the async void in a Task (which we can await), but
it still blows up
    //await Task.Run(AsyncVoid);
}
catch (Exception ex)
{
    Console.WriteLine("Look! We caught the exception here!");
    Console.WriteLine(ex);
}
Console.WriteLine("End");

async void AsyncVoid()
{
    Console.WriteLine("Entering async void method");
    await AsyncTask();

    // Pretend there's some super critical code right here
    // ...

    Console.WriteLine("Leaving async void method.");
}

async Task AsyncTask()
{
    Console.WriteLine("Entering async Task method");
    Console.WriteLine("About to throw...");
    throw new Exception("The expected exception");
}
```

As the code above shows, we have three examples scenarios to look at. And seriously, jump over to the link and try them each out by uncommenting one of those lines at a time.

Scenario 1

In scenario 1, we see our old faithful friend async Task. This task is going to throw an exception when we run the program, and since we are awaiting an async Task, the wrapping try/catch block is able to catch the exception just as we'd expect.

Scenario 2

In scenario 2, this might seem like some code that brought you here. The dreaded async void. When you run this one, you'll notice that we print information that we're about to throw the exception… But then we never see the indication that we're leaving the async void method! In fact, we just see the line indicating the program has ended. How spooky is that?

Scenario 3

In scenario 3, this might look like an attempt you have tried to solve your async void woes. Surely if we wrap the async void in a Task, we can await that, and then we'll go back to being safe… right? Right?! No. In fact, in .NET Fiddle you'll actually see that it prints "Unhandled exception". That's not something we have in our code, that's something scarier firing off *AFTER* our program is complete.

What's Actually Actually The Problem?

Fundamentally, no matter how you try massage code around, if you have an event handler the return type of it must be void to hook up to a C# event using the + operator.

Period.

This means that even if you try to refactor all of your code to avoid this, you're only "fixing" the issue by running away from it. And yes, while I agree if possible you should try and write code that doesn't trap you into crappy patterns, sometimes curiosity wins out.

Is there a way that we could still have our event handlers as async void (so that we can await things inside of them)? Yes, of course! And the added bonus of the method that we're going to dive into is that it's not restricted to just event handlers. You can leverage this pattern on any old async void method you have laying around.

Do I recommend it? Absolutely not. I often find myself designing objects with events on them and I try to restrict my usage to this exact scenario. You know, the one that all of the C# gurus say "Fine, if you MUST use async void then… Make sure it's for an event handler on a button or something".

So let's look at how we can make that a less painful experience going forward.

Waking Up From The Nightmare

You've been patient enough, so let's dive right into it. If you want to follow along with a working demo right in your browser, check out this link.

Here's the demo code, which we'll cover right after:

C#

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.ExceptionServices;

var someEventRaisingObject = new SomeEventRaisingObject();

// notice the async void. BEHOLD!!
someEventRaisingObject.TheEvent += async (s, e) =>
{
    Console.WriteLine("Entering the event handler...");
    await TaskThatIsReallyImportantAsync();
    Console.WriteLine("Exiting the event handler...");
};

try
{
    // forcefully fire our event!
    await someEventRaisingObject.RaiseEventAsync();
}
catch (Exception ex)
{
    Console.WriteLine($"Caught an exception in our handler: {ex.Message}");
}

// just block so we can wait for async operations to complete
Console.WriteLine("Press Enter to exit");
Console.ReadLine();

async Task TaskThatIsReallyImportantAsync()
{
    // switch between these two lines (comment one or the other out) to play with the
behavior
    throw new Exception("This is an expected exception");
    //await Task.Run(() => Console.WriteLine("Look at us writing to the console from a
task!"));
}

class SomeEventRaisingObject
{
    // you could in theory have your own event args here
    public event EventHandler<EventArgs> TheEvent;

    public Task RaiseEventAsync()
    {
        // the old way (if you toggle this way with the exception throwing, it will not hit
our handler!)
        //TheEvent?.Invoke(this, EventArgs.Empty);
        //return Task.CompletedTask;

        // the new way (if you toggle this way with the exception throwing, it WILL hit our
handler!)
```

```
        return InvokeAsync(TheEvent, true, true, this, EventArgs.Empty);
    }

    private async Task InvokeAsync(
            MulticastDelegate theDelegate,
            bool forceOrdering,
            bool stopOnFirstError,
            params object[] args)
        {
            if (theDelegate is null)
            {
                return;
            }

            var taskCompletionSource = new TaskCompletionSource<bool>();

            // this is used to try and ensure we do not try and set more
            // information on the TaskCompletionSource after it is complete
            // due to some out-of-ordering issues
            bool taskCompletionSourceCompleted = false;

            var delegates = theDelegate.GetInvocationList();
            var countOfDelegates = delegates.Length;

            // keep track of exceptions along the way and a separate collection
            // for exceptions we have assigned to the TCS
            var assignedExceptions = new List<Exception>();
            var trackedExceptions = new ConcurrentQueue<Exception>();

            foreach (var @delegate in theDelegate.GetInvocationList())
            {
                var async = @delegate.Method
                    .GetCustomAttributes(typeof(AsyncStateMachineAttribute), false)
                    .Any();

                bool waitFlag = false;
                var completed = new Action(() =>
                {
                    if (Interlocked.Decrement(ref countOfDelegates) == 0)
                    {
                        lock (taskCompletionSource)
                        {
                            if (taskCompletionSourceCompleted)
                            {
                                return;
                            }

                            assignedExceptions.AddRange(trackedExceptions);

                            if (!trackedExceptions.Any())
                            {
                                taskCompletionSource.SetResult(true);
                            }
                            else if (trackedExceptions.Count == 1)
                            {
                                taskCompletionSource.SetException(assignedExceptions[0]);
                            }
                            else
                            {
                                taskCompletionSource.SetException(new
AggregateException(assignedExceptions));
                            }

                            taskCompletionSourceCompleted = true;
                        }
                    }

                    waitFlag = true;
                });
                var failed = new Action<Exception>(e =>
                {
                    trackedExceptions.Enqueue(e);
                });

                if (async)
                {
                    var context = new EventHandlerSynchronizationContext(completed,
failed);
                    SynchronizationContext.SetSynchronizationContext(context);
                }

                try
                {
                    @delegate.DynamicInvoke(args);
                }
                catch (TargetParameterCountException e)
                {
                    throw;
                }
                catch (TargetInvocationException e) when (e.InnerException != null)
                {
                    // When exception occured inside Delegate.Invoke method all exceptions
are wrapped in
                    // TargetInvocationException.
                    failed(e.InnerException);
                }
                catch (Exception e)
                {
                    failed(e);
                }

                if (!async)
                {
                    completed();
                }

                while (forceOrdering && !waitFlag)
                {
                    await Task.Yield();
                }

                if (stopOnFirstError && trackedExceptions.Any() &&
!taskCompletionSourceCompleted)
                {
                    lock (taskCompletionSource)
                    {
                        if (!taskCompletionSourceCompleted && !assignedExceptions.Any())
                        {
                            assignedExceptions.AddRange(trackedExceptions);
                            if (trackedExceptions.Count == 1)
                            {
                                taskCompletionSource.SetException(assignedExceptions[0]);
                            }
                            else
                            {
                                taskCompletionSource.SetException(new
AggregateException(assignedExceptions));
                            }

                            taskCompletionSourceCompleted = true;
                        }
                    }

                    break;
                }
            }

            await taskCompletionSource.Task;
        }

        private class EventHandlerSynchronizationContext : SynchronizationContext
        {
            private readonly Action _completed;
            private readonly Action<Exception> _failed;

            public EventHandlerSynchronizationContext(
                Action completed,
                Action<Exception> failed)
            {
                _completed = completed;
                _failed = failed;
            }

            public override SynchronizationContext CreateCopy()
            {
                return new EventHandlerSynchronizationContext(
                    _completed,
                    _failed);
            }

            public override void Post(SendOrPostCallback d, object state)
            {
                if (state is ExceptionDispatchInfo edi)
                {
                    _failed(edi.SourceException);
                }
                else
                {
                    base.Post(d, state);
                }
            }
            public override void Send(SendOrPostCallback d, object state)
            {
                if (state is ExceptionDispatchInfo edi)
                {
                    _failed(edi.SourceException);
                }
                else
                {
                    base.Send(d, state);
                }
            }

            public override void OperationCompleted() => _completed();
        }
}
```

Understanding The Scenarios

There are two classes of things that we can play with in this code snippet:

- Toggle between what `TaskThatIsReallyImportantAsync` does. You can either safely print to the console, or have it throw an exception so you can experiment with the happy vs bad path. This alone isn't the focus of the article, but it allows you to try different situations.
- Toggle the behavior of `RaiseEventAsync`, which will show you the not-so-great behavior of async void compared to our solution! Mix this with the previous option to see how we can improve our ability to catch exceptions.

How It Works

This solution is something I heavily borrowed from Oleg Karasik. In fact, his original article does such a fantastic job explaining how he built out the algorithm feature at a time. So I give full credit to him because without his post, I never would have put together what I am showing here.

The main take aways regarding how this works are that:

- We can use a custom synchronization context, specifically for handling events, that take a custom completed/failed callback.
- We can use reflection to check for `AsyncStateMachineAttribute` to see if our handler is truly marked as async or not.
- `GetInvocationList` allows us to get the entire chain of handled registered to an event.
- `DynamicInvoke` allows us to invoke the delegate without the compiler screaming about types.

When you combine the above, we can also layer in a couple of other features:

- `forceOrdering`: This boolean flag can force each handler to run in the order that they were registered or if it's disabled, the handlers can run asynchronously with each other.

- `stopOnFirstError`: This boolean flag can prevent the code from firing off the subsequent event handlers if one of them raises an exception.

In the end, we are able to set information on a `TaskCompletionSource` instance that indicates completion or an exception being tracked.

Cool Stuff! What Did You Contribute?

Good question! Aside from just trying to raise awareness of this code that I think is super cool, I have made my own adjustments. While I really loved what the original code offered, it wasn't particularly how I'd like to have access to it in my own codebase.

Multicast Delegates

When I got to playing around with this code, I realized that the things I wanted to support are all essentially this type called a `MulticastDelegate`. This means `EventHandler`, `EventHandler<T>`, and even `Action`, `Action<T>`, `Action<T1, T2>`, etc… are all technically supported by this approach.

I wanted to offer this in the library support so if there was a situation outside of event handlers where this showed up, then I would have a solution to work with. When does that happen? Not very often, really. However, now if I ever come across a situation where I need an `Action` with an async void implementation, I can sleep easy at night.

Extension Methods

From a technical perspective, this isn't all that impressive. However, from a quality of life perspective I wanted to build out a library that had access to extension methods that were really obvious for me about their use case. Breaking these out further to separate `EventHandler<T>` support from `Action` support meant that I could have cleaner APIs. For example, every `EventHandler<T>` must have a sender and an `EventArgs` instance. However, an `Action` does not require either and therefore should have an API that is aligned to that. Similarly, the more type parameters on `Action` the more arguments I should allow.

Aggregate Exceptions

When we're dealing with a bunch of handlers that can throw exceptions, we can run into a similar situation where we're not tracking all of the things that blow up. For most of us, that was the primary motivator anyway. But if you're super unlucky, it wasn't just a single handler blowing up… it was multiple.

Tracking exceptions allows us to throw an `AggregateException` if we have multiple. There are even tests to illustrate that this behavior is covered as we'd expect!

Let’s See An Example!

Okay, one last example using my variation. You can see the code below by checking out this link:

C#

```
using System;
using System.Threading;
using System.Threading.Tasks;

var invoker = new GenericEventHandlerInvoker();
invoker.Event += async (s, e) =>
{
    await Task.Run(() => Console.WriteLine("Printing 1 from a task."));
    throw new InvalidOperationException("expected 1!");
};
invoker.Event += async (s, e) =>
{
    await Task.Run(() => Console.WriteLine("Printing 2 from a task."));
    throw new InvalidOperationException("expected 2!");
};

Console.WriteLine("Starting...");
try
{
    await invoker.InvokeAsync(
        ordered: false,
        stopOnFirstError: false);
}
catch (Exception ex)
{
    Console.WriteLine("We caught the exception!");
    Console.WriteLine(ex);
}

Console.WriteLine("Done!");

class GenericEventHandlerInvoker
{
    public event EventHandler<EventArgs> Event;

    public async Task InvokeAsync(
        bool ordered,
        bool stopOnFirstError)
    {
        // here is the extension method syntax we can see from NexusLabs.Framework
        await Event
            .InvokeAsync(
                this,
                EventArgs.Empty,
                ordered,
                stopOnFirstError)
            .ConfigureAwait(false);
    }
}
```

Like the earlier example, we have an object that's going to raise events for us that we can subscribe to. In this example though, we will register two async void methods. Each method will print to the console and then throw an exception. We wrap the invocation of the event itself in a try catch so we can prove the behavior we expect, which is we must be able to catch the exceptions from async void.

If you run this code as is, you will note that you get both lines printed to the console. You can change this by inverting the value for `stopOnFirstError`. You will also notice that by not stopping on the first error, we actually had an `AggregateException` get thrown that contains both of our exceptions.

Finally, if you look towards the end of the file you will see that I have this all wrapped up in an extension method called `InvokeAsync`, which you can mostly invoke like a normal event. There are other flavors of this that express the ordering directly in the name of the method instead of passing it as a parameter if you prefer it for verbosity purposes.

Conclusion

Hopefully this article will save you from the nightmare that is async void. I have this code up in a repository on GitHub, and to be clear, my intention is not to get you to go download and use my package/repo. It's nice if you'd like to, but I make this library for me to save time on my C# projects. If I have an angry mob of people chasing me down regularly for async/await issues that will probably lower my quality of life.

I hope this article serves as a reminder to folks that when we hear "It can't be done" or "Never do X" or "Always do Y", we shouldn't always stop in our tracks without trying to understand further. When we have constraints, this is when we come up with the most creative solutions!

The post async void – How to Tame the Asynchronous Nightmare appeared first on Dev Leader.

License

Written By

Dev Leader

Team Leader Microsoft

United States

I'm a software engineering professional with a decade of hands-on experience creating software and managing engineering teams. I graduated from the University of Waterloo in Honours Computer Engineering in 2012.

I started blogging at http://www.devleader.ca in order to share my experiences about leadership (especially in a startup environment) and development experience. Since then, I have been trying to create content on various platforms to be able to share information about programming and engineering leadership.

My Social:
YouTube: https://youtube.com/@DevLeader
TikTok: https://www.tiktok.com/@devleader
Blog: http://www.devleader.ca/
GitHub: https://github.com/ncosentino/
Twitch: https://www.twitch.tv/ncosentino
Twitter: https://twitter.com/DevLeaderCa
Facebook: https://www.facebook.com/DevLeaderCa
Instagram:
https://www.instagram.com/dev.leader
LinkedIn: https://www.linkedin.com/in/nickcosentino

Comments and Discussions

5 messages have been posted for this article Visit **https://www.codeproject.com/Articles/5353314/async-void-How-to-Tame-the-Asynchronous-Nightmare** to post and view comments on this article, or click **here** to get a print view with messages.