Whose Task is this?

Dominik Przywara

sopra steria

Agenda

- Introduction
- Run and forget pattern
- Wrapping events into Task
- Async operations in lambdas
- ConfigureAwait()
- ValueTask
- Tasks aggregation
- CancellationToken
- Dealing with exceptions

Lots of examples

Why even bother?

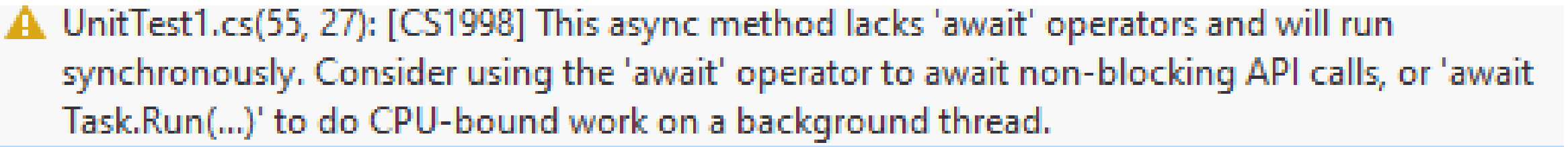
- To not waste time on things that don’t really matter…
- …or things that needs a lot of it
- To give better user experience – no more freezing UI!
- To prevent OS from killing your app

Evolution over the years in .NET

- APM (Asynchronous Programming Model) – IAsyncResult with Begin() & End() methods – **deprecated**
- EAP (Event-based Asynchronous Programming) – **deprecated***
- TAP (Task-based Asynchronous Programming) – since .NET 4.0

Let’s go with basics!

- There are two types of async operations – I/O bound and CPU bound
- It’s a promise that job will be done
- ***async*** keyword in method declaration does not make your methods asynchronous…
- …it’s the ***await*** that does!
- You return either *void/Task/Task*
- Using await yields control to it’s caller*
- Console app and **ASP.NET Core** by default has no SynchronizationContext!
- Task is reference type
- It uses state machine


UnitTest1.cs(55, 27): [CS1998] This async method lacks 'await' operators and will run synchronously. Consider using the 'await' operator to await non-blocking API calls, or 'await Task.Run(...)' to do CPU-bound work on a background thread.

Wait, what? Yielding control to caller?

DEMO

Await fast.

Sync over async (blocking)

- **It’s always a bad idea**
- Avoid using
  - ***.Result***
  - ***.GetAwaiter().GetResult()***
  - ***.Wait()***

  … Or wrapping one of above into Task.Run();
- **YOU SHOULD ALWAYS USE *await***

async != async

Threadpool work (CPU bounded)

```
private async Task HeavyBackgroundJob(string textToCrack)
{
    // some work
    result = await Task.Run(() => CrackRSA(textToCrack));
    // even more heavy work to do
}
```

I/O bound – no thread needed

```
private async Task VeryImportantTextToDownload()
{
    // some work
    using var httpClient = new HttpClient();
    result = await httpClient.GetStringAsync(uri);
    //even more to do
}
```

Sync with async result

```
public Task<Guid> SyncWithAsyncResult()
{
    //Very bad idea
    return Task.Run(Guid.NewGuid);
}
```

Sync with async result

```
public Task<Guid> SyncWithAsyncResult()
{
    //Better idea
    return Task.FromResult(Guid.NewGuid());
}
```

Sync with async result

```
public ValueTask<Guid> SyncWithAsyncResult()
{
    //Best idea when having synchronous result
    return new ValueTask<Guid>(Guid.NewGuid());
}
```

Avoid Task.Run in public API (e.g. NuGet)

Background jobs

Blocking the thread from thread-pool

```
public class BackgroundProcessor
{
    public void Start()
    {
        Task.Run(BackgroundJob);
    }

    private void BackgroundJob()
    {
    }
}
```

Dedicated thread

```
public class BackgroundProcessor
{
    public void Start()
    {
        var thread = new Thread(BackgroundJob)
        {
            // allows process to exit while this
            // thread is running
            IsBackground = true
        };

        thread.Start();
    }

    private void BackgroundJob()
    {
    }
}
```

Run and forget

```
private string _result;

private void Button1_Click(object sender, EventArgs args)
{
    DownloadText();
    ShowDialog(_result);
}

private async void DownloadText()
{
    using var httpClient = new HttpClient();
    _result = await httpClient.GetStringAsync(uri);
    // all exceptions thrown here will *CRASH PROCESS*
    // so your for example Web API will stop responding
}
```

Run and forget

```
private string _result;

private async void Button1_Click(object sender, EventArgs args)
{
    await DownloadText();
    ShowDialog(_result);
}

private async Task DownloadText()
{
    using var httpClient = new HttpClient();
    _result = await httpClient.GetStringAsync(uri);

    // all exceptions thrown here can be caught
    // using TaskScheduler.UnobservedTaskException
    // (event fires after GC.Collect and GC.WaitForPendingFinalizers)
}
```

Wrapping events into Task

```
public class FavoriteSongPlayer
{
    private Player _player;
    private Song _favoriteSong;

    public FavoriteSongPlayer(Song favoriteSong, Player player)
    {
        _favoriteSong = favoriteSong;
        _player = player;
    }

    public void PlayFavoriteSong()
    {
        _player.OnCompleted += ShowDialogOnSongCompleted;
        _player.Start(_favoriteSong);
    }

    private void ShowDialogOnSongCompleted(object sender, EventArgs e)
    {
        // Show dialog
    }
}
```

```
public Task PlayFavoriteSong(CancellationToken cancellationToken) {
    // It's super important to pass TaskCreationOptions.RunContinuationsAsynchronously
    // as a parameter. Thanks to that we maintain asynchronous calls later on!
    var tcs = new TaskCompletionSource<bool>(TaskCreationOptions.RunContinuationsAsynchronously);
    var registration = default(CancellationTokenRegistration);

    if (cancellationToken.CanBeCanceled) {
        registration = cancellationToken.Register(state => ((Player) state).ForceStop(), _player);
    }

    _player.OnCompleted += ShowDialogOnSongCompleted;
    _player.Start(_favoriteSong);

    return tcs.Task;

    void ShowDialogOnSongCompleted(object sender, EventArgs e) {
        registration.Dispose();
        _player.OnCompleted -= ShowDialogOnSongCompleted;

        if (_player.ForcelyStopped) {
            tcs.TrySetCanceled();
            return;
        }

        // Show dialog and wait for the result of it
        tcs.TrySetResult(true);
    }
}
```

Context matters

```
Action a1 = async () => await SomeAction();
Func<Task> t1 = async () => await SomeAction();

await Task.Run(async () => await SomeAction());
```

Context matters

```
public async Task ImportantMethod()
{
    try
    {
        await Dispatcher.RunAsync(async () => await SomeAction());
        throw new Exception("Exception have been thrown!");
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}

private async Task SomeAction()
{
    await Task.Delay(TimeSpan.FromSeconds(2));
    Console.WriteLine($"{nameof(SomeAction)} completed.");
}
```

```
Presentation.Tests.DispatchTest

Exception have been thrown!
```

Context matters

```
public static class Dispatcher
{
    public delegate void DispatcherHandler();

    public static Task RunAsync(DispatcherHandler handler)
    {
        return Task.Run(() => handler());
    }
}
```

Context matters

```
public static class Dispatcher
{
    public delegate Task DispatcherHandler();

    public static Task RunAsync(DispatcherHandler handler)
    {
        return Task.Run(() => handler());
    }
}
```

```
Presentation.Tests.DispatchTest

SomeAction completed.
Exception have been thrown!
```

Don’t come back – ConfigureAwait()

```
public async Task DownloadEverything()
{
    await DownloadConfig();
    await DownloadIssues();
    await DownloadOffers();
    await DownloadSomethingVeryImportant();
}
```

Don’t come back – ConfigureAwait()

```
var ctx = SynchronizationContext.Current;

DownloadConfig().ContinueWith(_ =>
{
    ctx.Post(__ =>
    {
        var ctx1 = SynchronizationContext.Current;
        DownloadIssues().ContinueWith(___ =>
        {
            ctx1.Post(____ =>
            {
                var ctx2 = SynchronizationContext.Current;
                DownloadOffers()
                    .ContinueWith(
                        _____ =>
                        {
                            ctx2.Post(______ =>
                            {
                                DownloadSomethingVeryImportant();
                            }, null);
                        },
                        null);
            }, null);
        }, null);
    }, null);
});
```

Don’t come back – ConfigureAwait()

```
public async Task DownloadEverything()
{
    await DownloadConfig().ConfigureAwait(false);
    await DownloadIssues().ConfigureAwait(false);
    await DownloadOffers().ConfigureAwait(false);
    await DownloadSomethingVeryImportant().ConfigureAwait(false);
}
```

Eliding await a.k.a. State machine short-cutting

```
public Task WithEliding()
{
    return Task.Delay(1000);
}

public async Task WithoutEliding()
{
    await Task.Delay(1000);
}
```

Eliding await a.k.a. State machine short-cutting

```
public async Task<string> GetKeywordsAsync(string url)
{
    using var client = new HttpClient();
    return await client.GetStringAsync(url);
}

public Task<string> GetKeywordsWithElidingAsync(string url)
{
    using var client = new HttpClient();
    return client.GetStringAsync(url);
}
```

.NET != .NET

```
588         protected override void Dispose(bool disposing)
589         {
590             if (disposing && !_disposed)
591             {
592                 _disposed = true;
593 
594                 // Cancel all pending requests (if any). Note that we don't call CancelPendingRequests() but cancel
595                 // the CTS directly. The reason is that CancelPendingRequests() would cancel the current CTS and create
596                 // a new CTS. We don't want a new CTS in this case.
597                 _pendingRequestsCts.Cancel();
598                 _pendingRequestsCts.Dispose();
599             }
600 
601             base.Dispose(disposing);
602         }
```

https://github.com/dotnet/corefx/blob/master/src/System.Net.Http/src/System/Net/Http/HttpClient.cs#L588

.NET != .NET

```
113                     protected override void Dispose (bool disposing)
114                     {
115                             if (disposing && !disposed) {
116                                     disposed = true;
117
118                                     cts.Dispose ();
119                             }
120
121                             base.Dispose (disposing);
122                     }
```

https://github.com/mono/mono/blob/master/mcs/class/System.Net.Http/System.Net.Http/HttpClient.cs#L113

How to deadlock your app

ValueTask

Make "async ValueTask/ValueTask<T>" methods ammortized allocation-free (Milestone 5.0)

https://github.com/dotnet/coreclr/pull/26310

Aggregating tasks

```
//async
Task.WhenAll();
Task.WhenAny();

//sync
Task.WaitAll();
Task.WaitAny();
```

Aggregating tasks

```
public async Task DownloadAllInParallel()
{
    var tasks = new List<Task>(capacity: 4);
    tasks.Add(DownloadIssues());
    tasks.Add(DownloadConfig());
    tasks.Add(DownloadOffers());
    tasks.Add(DownloadSomethingVeryImportant());

    await Task.WhenAll(tasks);
}
```

Let the fastest win

```
public async Task<StockExchange> GetStockExchangeFor(Company company)
{
    using var cts = new CancellationTokenSource();

    var tasks = new List<Task<StockExchange>>
    {
        FirstStockExchange(company, cts.Token),
        SecondStockExchange(company, cts.Token),
        ThirdStockExchange(company, cts.Token),
        FourthStockExchange(company, cts.Token)
    };

    var stock = await Task.WhenAny(tasks);
    cts.Cancel();

    return await stock;
}
```

Dealing with API without CancellationToken (hack)

```
public async Task<Config> DownloadConfigWithTimeout(int timeoutInSeconds = 5)
{
    var tcs = new TaskCompletionSource<Config>(
                                TaskCreationOptions.RunContinuationsAsynchronously);
    using var cts = new CancellationTokenSource();
    cts.CancelAfter(TimeSpan.FromSeconds(timeoutInSeconds));
    cts.Token.Register(() => tcs.TrySetCanceled());

    var tasks = new List<Task<Config>> {tcs.Task, DownloadConfig()};
    var completedTask = await Task.WhenAny(tasks);

    tcs.TrySetResult(null);
    return await completedTask;
}
```

```
try
{
    await DownloadConfig();
}
catch (Exception e)
{
   // Handle exception
}
try
{
    await DownloadIssues();
}
catch (Exception e)
{
   // Handle exception
}
try
{
    await DownloadOffers();
}
catch (Exception e)
{
   // Handle exception
}
try
{
    await DownloadSomethingVeryImportant();
}
catch (Exception e)
{
   // Handle exception
}
```

Houston, we have a problem!

Houston – we have some problems

```
var tasks = new List<Task>
{
    DownloadA(),
    DownloadB(),
    DownloadC()
};

try
{
    Task.WaitAll(tasks.ToArray());
}
catch (AggregateException ex)
{
    foreach (var innerException in ex.InnerExceptions)
    {
        Console.WriteLine(innerException.Message);
    }
}
```

Houston – we have some problems

```
var tasks = new List<Task>
{
    DownloadA(),
    DownloadB(),
    DownloadC()
};

Task task = Task.WhenAll(tasks);
try
{
    await task;
}
catch (ArgumentException firstThrownException)
{
    foreach (var innerException in (task.Exception as AggregateException).InnerExceptions)
    {
        Console.WriteLine(innerException.Message);
    }
}
```

Houston – we don’t know the problem

```
public Task WithEliding()
{
    return Task.Delay(1000);
}

public async Task WithoutEliding()
{
    await Task.Delay(1000);
}
```

Houston – we don’t know the problem

```
public Task WithEliding()
{
    throw new Exception($"{nameof(WithEliding)} method exception!");
    return Task.Delay(1000);
}

public async Task WithoutEliding()
{
    throw new Exception($"{nameof(WithoutEliding)} method exception!");
    await Task.Delay(1000);
}
```

Summary

- You have to know what you are doing while using Wait family
- Lack of CancellationToken is not a problem – e.g. use WhenAny with Task.Delay
- Context matters!
- Use ConfigureAwait(false) whenever possible
- Await fast
- Exceptions are aggregated
- Avoid Task.Run in public API
- **It’s awesome**

Bonus (if we have spare time)

More information & useful tools

- Stephen Cleary's blog
- C# docs
- https://github.com/StephenClearyArchive/AsyncEx.Tasks
- https://www.nuget.org/packages/MissingAwaitAnalyzer
- https://devblogs.microsoft.com/dotnet/understanding-the-whys-whats-and-whens-of-valuetask/
- https://github.com/davidfowl/AspNetCoreDiagnosticScenarios

About me

- Lead Software Engineer at Sopra Steria
- Co-Founder of Silesian Microsoft Group

https://github.com/dominikprzywara

sopra steria

Questions?

https://github.com/dominikprzywara/csharp-async-presentation

Thanks!