
ASYNC // AWAIT

- Software engineer here in Zurich working for Zühlke Engineering AG
- Originated from Germany
- I ♥ Blazor and C# / .NET
  - Even though these slides are written in Html / CSS / JavaScript
  - Wrote my own blog solely in Blazor, because I love that piece of technology
- ♥ NET User Group Zurich - Sharing knowledge / ideas is just great
- Contact: LinkedIn / My Blog / Github

DISCLAIMER

- As the nature of the topic can get quite complex, I will (over)simplify certain aspects. Also a basic prior knowledge to async/await is mandatory.
- Take these information with a grain of salt

AGENDA

1. Asynchronous Programming vs Parallel Programming
2. Deadlock and ConfigureAwait
3. State-Machine (very briefly and over-simplified)
4. Pitfalls & General Tips
5. ValueTask

THOUGHT-EXPERIMENT:

What do you think is the runtime of each of the code snippets?

Snippet A:

```
{
  var a = Task.Delay(1000);
  var b = Task.Delay(1000);
  var c = Task.Delay(1000);
  var d = Task.Delay(1000);
  var e = Task.Delay(1000);

  await a;
  await b;
  await c;
  await d;
  await e;
}
```

Snippet B:

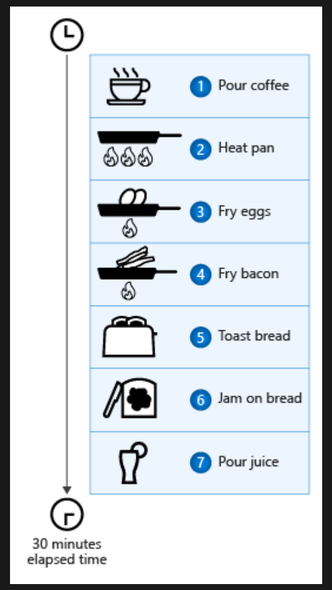
```
{
  await Task.Delay(1000);
  await Task.Delay(1000);
  await Task.Delay(1000);
  await Task.Delay(1000);
  await Task.Delay(1000);
}
```

ASYNCHRONOUS PROGRAMMING VS PARALLEL PROGRAMMING

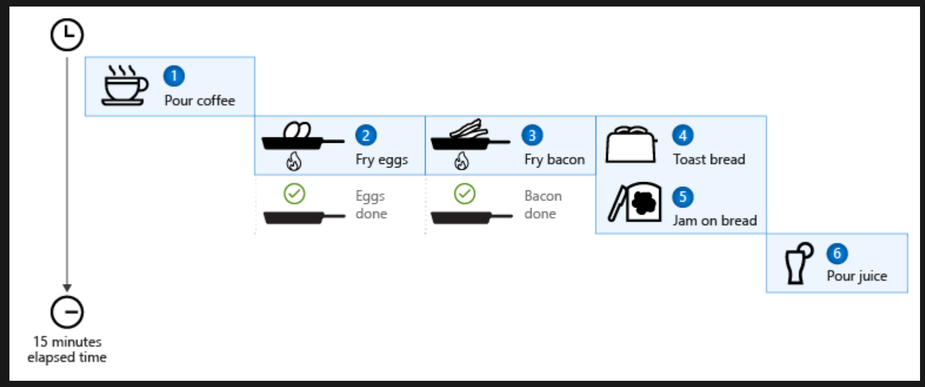
Imagine you want to cook breakfast, how do you do that?

1. Pour a cup of coffee
2. Heat up a pan, then fry two eggs
3. Fry three slices of bacon.
4. Toast two pieces of bread
5. Add butter and jam to the toast
6. Pour a glass of orange juice

SYNCHRONOUS:



ASYNCHRONOUS



ASYNCHRONOUS, CONCURRENCY, PARALLELISM???

- **Concurrency:** is when two or more tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant.
- **Parallelism:** is when tasks literally run at the same time, e.g., on a multicore processor. (Parallelism is a special form of concurrency)
- **Asynchronous:** is a mechanism to achieve concurrency (done via callbacks in C# and Javascript) but does not necessary involve multiple threads.

DEADLOCK / CONFIGUREAWAIT()

Imagine the following ASP.NET code in a controller:

```
1  public string GetSomething(string id)
2  {
3    var task = DoWorkAsync();
4    task.Wait();
5    return "done";
6  }
7
8  private async Task DoWorkAsync()
9  {
10   Debug.WriteLine("welcome");
11   await Task.Delay(500);
12   Debug.WriteLine("welcome back");
13 }
```

The request / main thread is waiting while the async / offloaded function is waiting to go to the same context as before (the main thread).

LET'S SEE THIS IN ACTION IN ASP.NET AND IN A CONSOLE APPLICATION

WHAT IS A SYNCHRONIZATIONCONTEXT? WHAT IS A TASK-SCHEDULER?

LET'S UNWRAP THAT A BIT

TASKSCHEDULER

Simplified:

- TaskScheduler is responsible for executing scheduled tasks
- Mainly there are two scheduler: *thread pool task scheduler* and *synchronization context task scheduler*
- Default is *thread pool task scheduler*. ASP.NET uses the later one

SYNCHRONIZATIONCONTEXT

Simplified:

- Different frameworks have different mechanism for communicating between tasks
- SynchronizationContext is the abstraction of exactly that - how to communicate
- It's a representation of the current environment **where** our code is running in
- It provides a way to queue a unit of work to a context
- Saved on the Task-object

WHY IS IT NEEDED?

Classic example: Windows Forms / WPF

- Windows Forms or WPF only allows the thread which created an UI-element to modify it, otherwise exception
- When you offload work and "come back" you still want to be able to update UI elements
- The synchronization context allows exactly that
- You can offload work to a background worker and come back to the exact same context (which is allowed to update/render on the UI-Element)

AND ASP.NET?

- It is mainly used for HttpContext.Current, Culture Information, Identity Information ...

WHAT CAN I DO TO PREVENT THE DEADLOCK?

CONFIGUREAWAIT TO THE RESCUE

- By default calls to an awaited task will capture the current context and attempt to resume execution on the context once complete
- Setting `ConfigureAwait(continueOnCapturedContext: false)` does the following:
    - When we re-enter from the asynchronous code to the awaiter, we basically tell: "I don't mind which context continues the execution"
        - The truth is, you will have no context at all. Even if you are on the same thread
    - Best suited for libraries. Maybe your code will run in a console application or in ASP.NET... no one knows 😉
    - Small performance improvement because you "miss out" the synchronization

THAT SOUNDS AWESOME. LET'S DO IT EVERYWHERE THEN!

There is a pitfall

- Earlier we discussed that the AspNetSynchronizationContext is responsible for HttpContext.Current, Culture Information, ...
- As you have no context, there is no information anymore present (for the continuation function)

```
public IEnumerable<string> Get()
{
  var httpContext = HttpContext.Current; // Is NOT null
  Debug.WriteLine("Before DoWorkAsync");
  var task = DoWorkAsync();
  task.Wait();
  httpContext = HttpContext.Current; // Is NOT null
  Debug.WriteLine("After DoWorkAsync");

  return new string[] { "value1", "value2" };
}

private async Task DoWorkAsync()
{
  var httpContext = HttpContext.Current; // Is NOT null
  Debug.WriteLine("Welcome");
  await Task.Delay(500).ConfigureAwait(false);
  httpContext = HttpContext.Current; // Is null
  Debug.WriteLine("We are done");
}
```

PROPER SOLUTION: ASYNC / AWAIT EVERYTHING

WHY IS IT DIFFERENT THAN THE SYNCHRONOUS VERSION?

- If the entire call stack is asynchronous there is no problem because, once await is reached the original thread is released, freeing the request context
- Therefore no deadlock

```
public async Task<IEnumerable<string>> Get()
{
  var httpContext = HttpContext.Current; // Is NOT null
  Debug.WriteLine("Before DoWorkAsync");
  var task = DoWorkAsync();
  await task;
  httpContext = HttpContext.Current; // Is NOT null
  Debug.WriteLine("After DoWorkAsync");

  return new string[] { "value1", "value2" };
}

private async Task DoWorkAsync()
{
  var httpContext = HttpContext.Current; // Is NOT null
  Debug.WriteLine("Welcome");
  await Task.Delay(500);
  httpContext = HttpContext.Current; // Is NOT null
  Debug.WriteLine("We are done");
}
```

ANOTHER ONE: USE ASP.NET CORE

- ASP.NET Core has no Synchronization Context
- Therefore you can't have the same deadlock as described above

	Custom User Thread	Console App	.NET Core Web App	Library Code	Desktop App	ASP.NET Web App
Do I have a SyncronizationContext by default?	no			inherited from caller	yes	yes
How is my task scheduled?	`Task.Start(TaskScheduler.Default);`			depends on caller	`SynchronizationContext.Current`	
By default code executes on	user thread	main thread	thread pool	caller thread	ui thread	request thread
By default tasks are executed on	thread pool			depends on caller	ui thread	request thread
`GetDataAsync().Result` deadlocks?	no		yes (if high load)	yes (depends on caller)	yes (instantly	
`Task.Run(async()=> await GetDataAsync()) .Result` can produce a deadlock?	no		yes (if high load)		no	
`Task.Run(()=>GetDataAsync() .Result).Result` can produce a deadlock?	yes (if the load is high enough)					

STATE-MACHINE

```
ass StockPrices

 private Dictionary _stockPrices;
 public async Task GetStockPriceForAsync(string companyId)
 {
     if (string.IsNullOrEmpty(_companyId)) { throw new ArgumentNullException();

     await InitializeMapIfNeededAsync();
     _stockPrices.TryGetValue(companyId, out var result);
     return result;
 }

 private async Task InitializeMapIfNeededAsync()
 {
     if (_stockPrices != null)
         return;

     await Task.Delay(42);
     // Getting the stock prices from the external source and cache in memory.
     _stockPrices = new Dictionary { { "MSFT", 42 } };
 }
```

```
1  class GetStockPriceForAsync_StateMachine
2  {
3    enum State { Start, Step1, }
4    private readonly StockPrices @this;
5    private readonly string _companyId;
6    private readonly TaskCompletionSource _tcs;
7    private Task _initializeMapIfNeededTask;
8    private State _state = State.Start;
9
10   public GetStockPriceForAsync_StateMachine(StockPrices @this, string com
11   {
12     this.@this = @this;
13     _companyId = companyId;
14   }
15
16   public void Start()
17   {
18     try
19     {
20       if (_state == State.Start)
21       {
22         // The code from the start of the method to the first 'await'.
23
24         if (string.IsNullOrEmpty(_companyId))
25           throw new ArgumentNullException();
26
27         _initializeMapIfNeededTask = @this.InitializeMapIfNeeded();
28
29         // Update state and schedule continuation
30         _state = State.Step1;
31         _initializeMapIfNeededTask.ContinueWith(_ => Start());
32       }
33       else if (_state == State.Step1)
34       {
35         // Need to check the error and the cancel case first
36         if (_initializeMapIfNeededTask.Status == TaskStatus.Canceled)
37           _tcs.SetCanceled();
```

Real Life Example


WON'T SOMEBODY PLEASE
THINK OF THE
Performance

PITFALLS

```
await MyLongRunningTask();
```

VERSUS

```
Task longRunningTask = MyLongRunningTask();
await longRunningTask;
```

AND WHAT ABOUT THREADS? ONE TASK = ONE THREAD?

THOUGHT-EXPERIMENT:

What do you think is the runtime of each of the code snippets?

Snippet A:

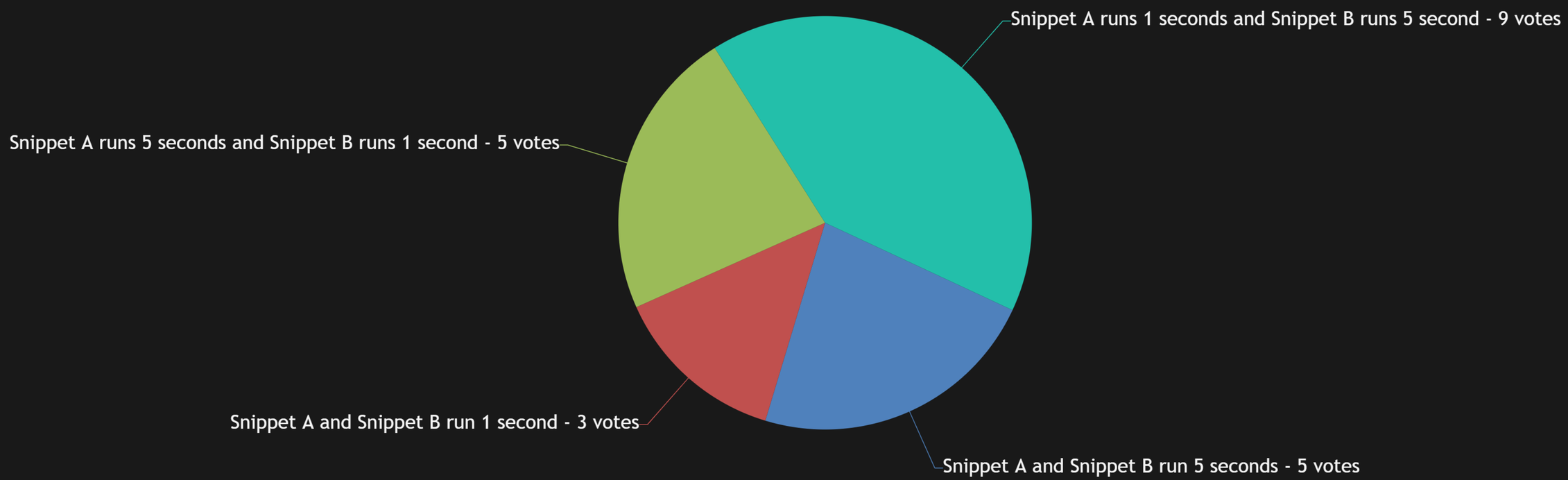
```
{
  var a = Task.Delay(1000);
  var b = Task.Delay(1000);
  var c = Task.Delay(1000);
  var d = Task.Delay(1000);
  var e = Task.Delay(1000);

  await a;
  await b;
  await c;
  await d;
  await e;
}
```

Snippet B:

```
{
  await Task.Delay(1000);
  await Task.Delay(1000);
  await Task.Delay(1000);
  await Task.Delay(1000);
  await Task.Delay(1000);
}
```

Runtime of async/await snippets

Total votes: 22



EXCEPTIONS

NOT PROPERLY AWAITED TASK

```
var ids = new List();
// ...
ids.ForEach(id => _myRepo.UpdateAsync(ids));
```

WHAT IS THE PROBLEM HERE? WE DO NOT AWAIT?

```
var ids = new List();
// ...
ids.ForEach(async id => await _myRepo.UpdateAsync(ids));
```

NOW BETTER?
DEFINITION OF FOREACH:

```
public void ForEach(Action<T> action)
```

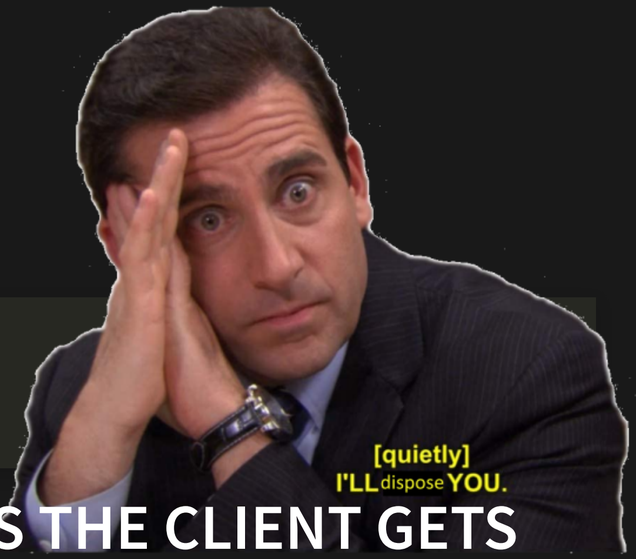
CAN NOT BE AWAITED. ACCEPTS ONLY ASYNC VOID.

THE SAME APPLIES FOR LINQ TO OBJECT (LIKE SELECT(), WHERE(), ...)

Another one without awaiting...

THIS SEEMS REASONABLE, OR?

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



WELL, NO. YOUR DOWNLOAD WILL MOST LIKELY BE ABORTED AS THE CLIENT GETS DISPOSED.

We saw earlier that only until the first await in GetStringAsync the function is called synchronously the rest is passed back as continuation-task. At this point the HttpClient gets disposed as we go out the using block

DO THIS INSTEAD

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

AWAIT IN FOREACH

Those calls in the foreach loop are not blocking each other, right?

```
public static async Task Main(string[] args)
{
  foreach (var x in new[] { 1, 2, 3 })
  {
    await DoSomethingAsync(x); // Could also be an HTTP call
  }
}

private static async Task DoSomethingAsync(int x)
{
  Console.WriteLine($"Doing {x}... ({DateTime.Now :hh:mm:ss})");
  await Task.Delay(2000);
  Console.WriteLine($"{x} done.    ({DateTime.Now :hh:mm:ss})");
}
```

Possible solution if you want to have them asynchronous

```
public static async Task Main(string[] args)
{
  var tasks = new List<Task>();

    foreach (var x in new[] { 1, 2, 3 })
    {
      var task = DoSomethingAsync(x);
      tasks.Add(task);
    }

    await Task.WhenAll();
}
```

- Be aware that now no order of execution is guaranteed

Some other pitfalls

- Don't use async void (it causes more havoc!). Only exception: Top-Level asynchronous eventhandler.
  - Exceptions can't be caught with catch (only inside the async void method)
  - Horrible to test
  - Can't be awaited

GENERAL TIPS

- Asynchronous and parallel programming are not the same
- That said: Task != Thread (even though this is sometimes true)
    - If you want to nitpick: Often times on Kernel-Level you have a thread for interrupts (I/O handling). Someone has to inform you when I/O is done. If you want to know more about that have a look at **Direct Memory Access (DMA)**.

- `Task.Run` is meant for CPU bound operations and should be used instead of `Task.Factory.StartNew`
  - Be cautious when using this in ASP.NET (Core) as you decrease your scalability
  - ASP.NET (Core): Each request is one thread, but with Task.Run you will remove one thread from the Threadpool
  - Your thread pool size is finite ;) and depends on various factors like virtual address space
  - Just because you can have hundreds of threads from the thread pool doesn't mean they can be executed "in parallel"

- Try not to mix synchronous code and asynchronous code (and vice versa)
    - Don't use a synchronous function when there is an asynchronous version available (DbContext.SaveChanges vs DbContext.SaveChangesAsync) in an async function
- ConfigureAwait(false) in libraries is in general a good idea
    - Exception: Methods that require context: GUI apps which modify UI-elements, ASP.NET which needs the HttpContext.Current
- How to call async function from sync function: MSDN
- Use CancellationToken to save some precious resources

TO SUM IT UP:

USE AWAIT BY DEFAULT (JUSTIFY IF YOU REALLY WANT TO OMIT IT). BE ASYNCHRONOUS IN THE WHOLE CHAIN. AND ALWAYS MEASURE FIRST, THEN ACT!

VALUETASK

Simplified version

```
public readonly struct ValueTask<TResult>
{
  internal readonly Task<TResult> _task;
  internal readonly TResult _result;
}
```

- Can be returned from an asynchronous function
- But also from a synchronous one
  - Here nothing has to be allocated
  - Only on completion of a asynchronous a Task object has to be allocated

CLASSIC USE CASE

```
1  public class WeatherService
2  {
3    private readonly Dictionary<string, WeatherData> _weatherCache;
4    private readonly IWeatherRepository weatherRepository;
5
6    public async ValueTask<WeatherData> GetWeatherForCityAsync(string city)
7    {
8      if (_weatherCache.ContainsKey(city))
9      {
10       return _weatherCache[city];
11     }
12
13     return await weatherRepository.GetForCityAsync(city);
14   }
15 }
```

- Not every code path is async
- Safe some time when the synchronous path is taken

BIG NO-NO'S

The following operations should never be performed on a ValueTask instance:

- Awaiting the instance multiple times
- Calling AsTask multiple times
- Using .Result or .GetAwaiter().GetResult() when the operation hasn't yet completed, or using them multiple times
- Using more than one of these techniques to consume the instance

If you do any of the above, the results are undefined.

SO SHOULD I TAKE IT?

Short answer: No

Long answer: It depends

ValueTask is designed for hot paths where every millisecond and every allocation matters. These are micro-optimizations. The pitfalls outweigh the gains in almost every case. As always measure first, act second.

Resources / Further Reading

- Repository with all the examples shown plus more (async void exceptions, benchmark, ...): here
- Material about async / await including SynchronizationContext: here, here and here
- Understanding the State-Machine: here
- ValueTask: here
- Async / await no threads? Why is it responsive? here