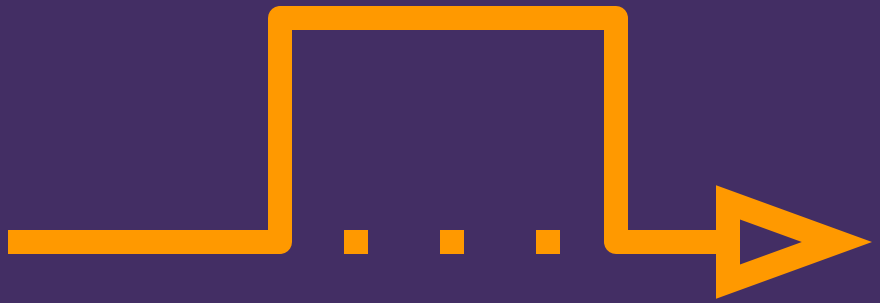
Async in C#

The good, the bad and the ugly

Hi, I'm Stu

- F# |> I ❤️
- 8 years of experience in .NET
- Organiser of Bristol F# meetup and DDD South West
- Passionate about Open Source

You can find me at @stuartblang



Why an Async talk?

History of Async

2011

Async/Await introduced
.NET 4.5
C# 5

You are here!

2018

It really troubles me how much async, bad async, really bad async we see in the wild

Kathleen Dollard - .NET Rocks! #1143

Async is an abstraction

Safe Abstractions

Dangerous Abstractions

Async/Await

“Powerful”

Leaky

Magic

The Good

- Non-blocking waiting
- There is no thread!

The Bad

- Not clear what is true async
- Minor performance overhead
- Duplicated code
- Async can’t go everywhere
- Risk of async deadlocks
- Doing it wrong is much worse that not doing it at all

How does it work?

```
1   string result = await MyAsyncMethod();
2
3   async Task<string> MyAsyncMethod()
4   {
5       Console.WriteLine("A");
6       await Task.Delay(100);
7       Console.WriteLine("B");
8       await Task.Delay(100);
9       Console.WriteLine("C");
10      return "Dunzo";
11  }
```

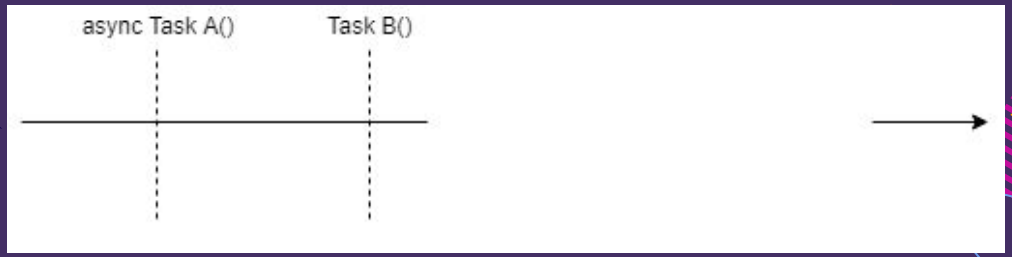
Compiler generated code

How does it work?

```
1   Task<string> myTask = MyAsyncMethod();
2   string result = await myTask;
3
4   async Task<string> MyAsyncMethod()
5   {
6       Console.WriteLine("A");
7       await Task.Delay(100);
8       Console.WriteLine("B");
9       await Task.Delay(100);
10      Console.WriteLine("C");
11      return "Dunzo";
12  }
```

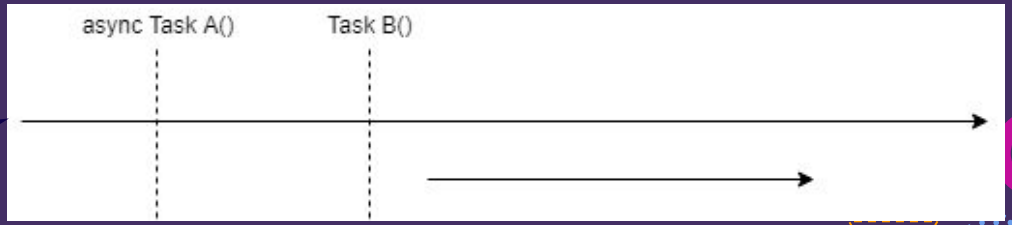
How does it work?

`True async`



`Task.FromResult(...)`

`Task.Run(() => ...)`



.ConfigureAwait(false)

What's that about?

Synchronization Context

What's that about?

How does it work?

```
1   string result = await MyAsyncMethod();
2
3   async Task<string> MyAsyncMethod()
4   {
5       Console.WriteLine("A");
6       await Task.Delay(100);
7       Console.WriteLine("B");
8       await Task.Delay(100);
9       Console.WriteLine("C");
10      return "Dunzo";
11  }
```

Continuing on Captured Context

```
1   string result = await MyAsyncMethod();
2
3   async Task<string> MyAsyncMethod()
4   {
5       Console.WriteLine("A");
6       await Task.Delay(100);
7       Console.WriteLine("B");
8       await Task.Delay(100).ConfigureAwait(false);
9       Console.WriteLine("C");
10      return "Dunzo";
11  }
```

Synchronization Context Cont.

```
public class SynchronizationContext
{
    // Dispatch work to the context.
    void Post(SendOrPostCallback d, object state);
    void Send(SendOrPostCallback d, object state);
    // Keep track of the number of asynchronous operations.
    void OperationCompleted();
    void OperationStarted();
    // Each thread has a current context.
    // If "Current" is null, then the thread's current context is
    // "new SynchronizationContext()", by convention.
    static SynchronizationContext Current { get; }
    static void SetSynchronizationContext(SynchronizationContext syncContext);
}
```

Synchronization Context Cont.

WindowsFormsSynchronizationContext

DispatcherSynchronizationContext

Default (ThreadPool) SynchronizationContext

AspNetSynchronizationContext

SynchronizationContext Behaviors

	Specific Thread Used to Execute Delegates	Exclusive (Delegates Execute One at a Time)	Ordered (Delegates Execute in Queue Order)	Send May Invoke Delegate Directly	Post May Invoke Delegate Directly
WinForms	Yes	Yes	Yes	If called from UI thread	Never
WPF	Yes	Yes	Yes	If called from UI thread	Never
Default	No	No	No	Always	Never
ASP.NET	No	Yes	No	Always	Always

Deadlock

```
1   string result = MyAsyncMethod().Result;
2
3   async Task<string> MyAsyncMethod()
4   {
5       Console.WriteLine("A");
6       await Task.Delay(100);
7       Console.WriteLine("B");
8       await Task.Delay(100);
9       Console.WriteLine("C");
10      return "Dunzo";
11  }
```

```
public async Task<ActionResult> About()
{
    async Task<int> CountdownAsync(int number)
    {
        using (new AsyncTrace(_trace, nameof(CountdownAsync)))
        {
            if (number > 1)
                number = await CountdownAsync(number - 1);
            await Task.Delay(100);
            return number;
        }
    }

    using (new AsyncTrace(_trace))
    {
        int result = await CountdownAsync(5);
    }

    ViewBag.Diagnostics = _trace;
    return View();
}
```

#	Method Name	Context Type	Thread ID
→	About	AspNetSynchronizationContext	33
→	CountdownAsync	AspNetSynchronizationContext	33
→	CountdownAsync	AspNetSynchronizationContext	33
→	CountdownAsync	AspNetSynchronizationContext	33
→	CountdownAsync	AspNetSynchronizationContext	33
→	CountdownAsync	AspNetSynchronizationContext	33
←	CountdownAsync	AspNetSynchronizationContext	116
←	CountdownAsync	AspNetSynchronizationContext	132
←	CountdownAsync	AspNetSynchronizationContext	127
←	CountdownAsync	AspNetSynchronizationContext	139
←	CountdownAsync	AspNetSynchronizationContext	118
←	About	AspNetSynchronizationContext	118

```
public async Task<ActionResult> About()
{
    async Task<int> CountdownAsync(int number)
    {
        using (new AsyncTrace(_trace, nameof(CountdownAsync)))
        {
            if (number > 1)
                number = await CountdownAsync(number - 1);
            await Task.Delay(100);
            return number;
        }
    }

    using (new AsyncTrace(_trace))
    {
        int result = CountdownAsync(5).Result;
    }

    ViewBag.Diagnostics = _trace;
    return View();
}
```

```
public async Task<ActionResult> About()
{
    async Task<int> CountdownAsync(int number)
    {
        using (new AsyncTrace(_trace, nameof(CountdownAsync)))
        {
            if (number > 1)
                number = await CountdownAsync(number - 1)
                                .ConfigureAwait(false);
            await Task.Delay(100).ConfigureAwait(false);
            return number;
        }
    }

    using (new AsyncTrace(_trace))
    {
        int result = CountdownAsync(5).Result;
    }

    ViewBag.Diagnostics = _trace;
    return View();
}
```

#	Method Name	Context Type	Thread ID
→	About	AspNetSynchronizationContext	9
→	CountdownAsync	AspNetSynchronizationContext	9
→	CountdownAsync	AspNetSynchronizationContext	9
→	CountdownAsync	AspNetSynchronizationContext	9
→	CountdownAsync	AspNetSynchronizationContext	9
→	CountdownAsync	AspNetSynchronizationContext	9
←	CountdownAsync	(null)	14
←	CountdownAsync	(null)	10
←	CountdownAsync	(null)	12
←	CountdownAsync	(null)	7
←	CountdownAsync	(null)	8
←	About	AspNetSynchronizationContext	9

```
public async Task<ActionResult> About()
{
    async Task<int> CountdownAsync(int number)
    {
        await Task.Delay(100).ConfigureAwait(false);
        using (new AsyncTrace(_trace, nameof(CountdownAsync)))
        {
            if (number > 1)
                number = await CountdownAsync(number - 1);
            await Task.Delay(100);
            return number;
        }
    }

    using (new AsyncTrace(_trace))
    {
        int result = CountdownAsync(5).Result;
    }

    ViewBag.Diagnostics = _trace;
    return View();
}
```

#	Method Name	Context Type	Thread ID
→	About	AspNetSynchronizationContext	7
→	CountdownAsync	(null)	13
→	CountdownAsync	(null)	12
→	CountdownAsync	(null)	13
→	CountdownAsync	(null)	8
→	CountdownAsync	(null)	5
←	CountdownAsync	(null)	10
←	CountdownAsync	(null)	12
←	CountdownAsync	(null)	13
←	CountdownAsync	(null)	8
←	CountdownAsync	(null)	5
←	About	AspNetSynchronizationContext	7

```
public async Task<ActionResult> About()
{
    async Task<int> CountdownAsync(int number)
    {
        await Task.Delay(0).ConfigureAwait(false);
        using (new AsyncTrace(_trace, nameof(CountdownAsync)))
        {
            if (number > 1)
                number = await CountdownAsync(number - 1);
            await Task.Delay(100);
            return number;
        }
    }

    using (new AsyncTrace(_trace))
    {
        int result = CountdownAsync(5).Result;
    }

    ViewBag.Diagnostics = _trace;
    return View();
}
```

```
public async Task<ActionResult> About()
{
    async Task<int> CountdownAsync(int number)
    {
        SynchronizationContext.SetSynchronizationContext(null);
        using (new AsyncTrace(_trace, nameof(CountdownAsync)))
        {
            if (number > 1)
                number = await CountdownAsync(number - 1);
            await Task.Delay(100);
            return number;
        }
    }

    using (new AsyncTrace(_trace))
    {
        int result = CountdownAsync(5).Result;
    }

    ViewBag.Diagnostics = _trace;
    return View();
}
```

#	Method Name	Context Type	Thread ID
→	About	AspNetSynchronizationContext	29
→	CountdownAsync	(null)	29
→	CountdownAsync	(null)	29
→	CountdownAsync	(null)	29
→	CountdownAsync	(null)	29
→	CountdownAsync	(null)	29
←	CountdownAsync	(null)	46
←	CountdownAsync	(null)	53
←	CountdownAsync	(null)	57
←	CountdownAsync	(null)	53
←	CountdownAsync	(null)	49
←	About	AspNetSynchronizationContext	29

```
public async Task<ActionResult> About()
{
    async Task<int> CountdownAsync(int number)
    {
        using (new AsyncTrace(_trace, nameof(CountdownAsync)))
        {
            if (number > 1)
                number = await CountdownAsync(number - 1);
            await Task.Delay(100);
            return number;
        }
    }

    using (new AsyncTrace(_trace))
    {
        int result = Task.Run(() => CountdownAsync(5).Result)
                         .Result;
    }

    ViewBag.Diagnostics = _trace;
    return View();
}
```

#	Method Name	Context Type	Thread ID
→	About	AspNetSynchronizationContext	14
→	CountdownAsync	(null)	6
→	CountdownAsync	(null)	6
→	CountdownAsync	(null)	6
→	CountdownAsync	(null)	6
→	CountdownAsync	(null)	6
←	CountdownAsync	(null)	7
←	CountdownAsync	(null)	8
←	CountdownAsync	(null)	7
←	CountdownAsync	(null)	8
←	CountdownAsync	(null)	7
←	About	AspNetSynchronizationContext	14

```
public async Task<ActionResult> About()
{
    async ContextFreeTask<int> CountdownAsync(int number)
    {
        using (new AsyncTrace(_trace, nameof(CountdownAsync)))
        {
            if (number > 1)
                number = await CountdownAsync(number - 1);
            await Task.Delay(100);
            return number;
        }
    }

    using (new AsyncTrace(_trace))
    {
        int result = CountdownAsync(5).Result;
    }

    ViewBag.Diagnostics = _trace;
    return View();
}
```

ContextFreeTask

#	Method Name	Context Type	Thread ID
→	About	AspNetSynchronizationContext	17
→	CountdownAsync	AspNetSynchronizationContext	17
→	CountdownAsync	AspNetSynchronizationContext	17
→	CountdownAsync	AspNetSynchronizationContext	17
→	CountdownAsync	AspNetSynchronizationContext	17
→	CountdownAsync	AspNetSynchronizationContext	17
←	CountdownAsync	(null)	19
←	CountdownAsync	(null)	11
←	CountdownAsync	(null)	20
←	CountdownAsync	(null)	13
←	CountdownAsync	(null)	9
←	About	AspNetSynchronizationContext	17

```
public async Task<ActionResult> About()
{
    async Task<int> CountdownAsync(int number)
    {
        using (new AsyncTrace(_trace, nameof(CountdownAsync)))
        {
            if (number > 1)
                number = await CountdownAsync(number - 1);
            await Task.Delay(100);
            return number;
        }
    }

    using (new AsyncTrace(_trace))
    {
        var context = new JoinableTaskContext();
        var jtf = new JoinableTaskFactory(context);
        int result = jtf.Run(() => CountdownAsync(5));
    }

    ViewBag.Diagnostics = _trace;
    return View();
}
```

vs-threading

#	Method Name	Context Type	Thread ID
→	About	AspNetSynchronizationContext	10
→	CountdownAsync	JoinableTaskSynchronizationContext	10
→	CountdownAsync	JoinableTaskSynchronizationContext	10
→	CountdownAsync	JoinableTaskSynchronizationContext	10
→	CountdownAsync	JoinableTaskSynchronizationContext	10
→	CountdownAsync	JoinableTaskSynchronizationContext	10
←	CountdownAsync	JoinableTaskSynchronizationContext	10
←	CountdownAsync	JoinableTaskSynchronizationContext	10
←	CountdownAsync	JoinableTaskSynchronizationContext	10
←	CountdownAsync	JoinableTaskSynchronizationContext	10
←	CountdownAsync	JoinableTaskSynchronizationContext	10
←	About	AspNetSynchronizationContext	10

Formal definition of async deadlocks

You are susceptible to deadlocks if:

1. You have a current SynchronizationContext that enforces exclusive access;
2. Some code further into your call stack has access to it and can/does:
    a. Synchronously block on some async code;
    b. Within that async code awaits an incomplete task that does not use .ConfigureAwait(false), or temporarily removes the context.

If you use .Result, .Wait(), .GetAwaiter().GetResult() you have done a dangerous thing, and you should be prepared to guard against naughty awaiters (you may not even control).

Resources

Prevention Methods

- .ConfigureAwait(false) all the things
- ContextFreeTask
- Comment the code
- null the SynchronizationContext (with handy helper functions)
- async top to bottom (replace our libraries and framework where we have to)
- Deadlock detection in QA & Prod
- Detect deadlocks in unit tests?
- Use ASP.NET Core!

Automated Tools

- Microsoft's AsyncPackage (Roslyn Analyzer) - Async006 - Detects more blocking calls than you can shake a stick at.
- ConfigureAwait.Fody
- ConfigureAwait Checker for ReSharper
- ConfigureAwaitChecker - Roslyn Analyzer + VSIX
- DeadlockDetection

Awesome async resources

- Stephen Cleary
  - https://blog.stephencleary.com/
- Anthony Steele
  - Avoiding basic mistakes in async await
  - Resynchronising async code

Async Guidance

- davidfowl/AspNetCoreDiagnosticScenarios - AsyncGuidance.md

Thanks!

Any questions?

You can find me at @stuartblang & stuart.b.lang@gmail.com

Thanks!

Any questions?

You can find me at @stuartblang & stuart.b.lang@gmail.com