Async in C#, the Old and the New

HELLO!

I am Stuart Lang

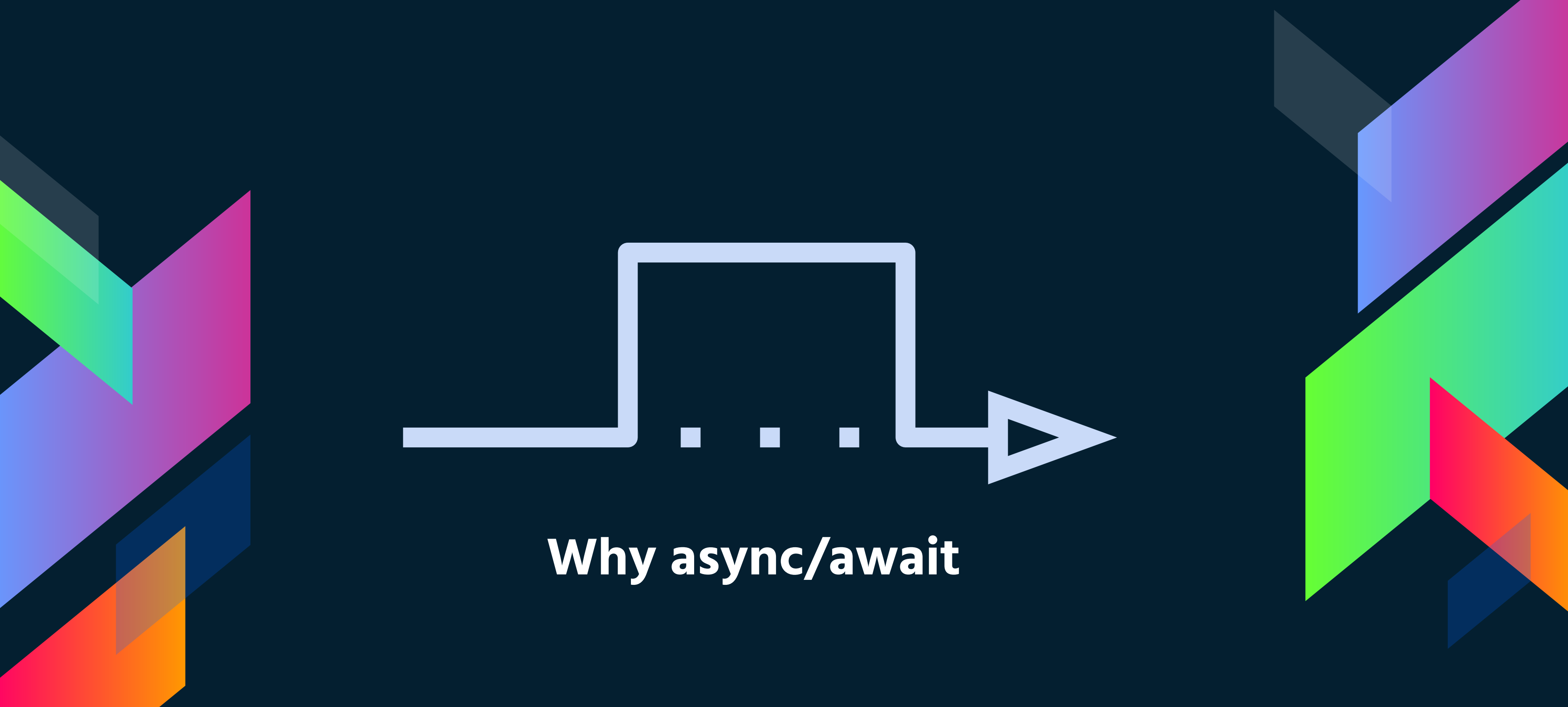
You can find me at

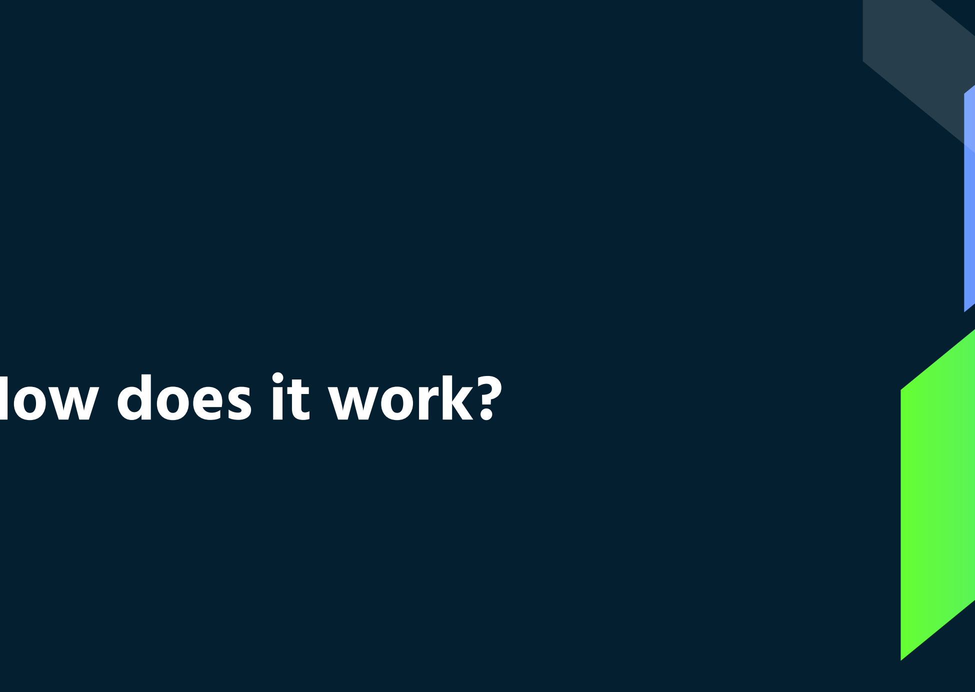
stu.dev

{ } NDC Conferences

Agenda

- Introduction to async and how it works
- A look at the new guidance - do’s & don'ts
- Some new stuff
  - Async Streams
  - Channels


Why async/await



How does it work?

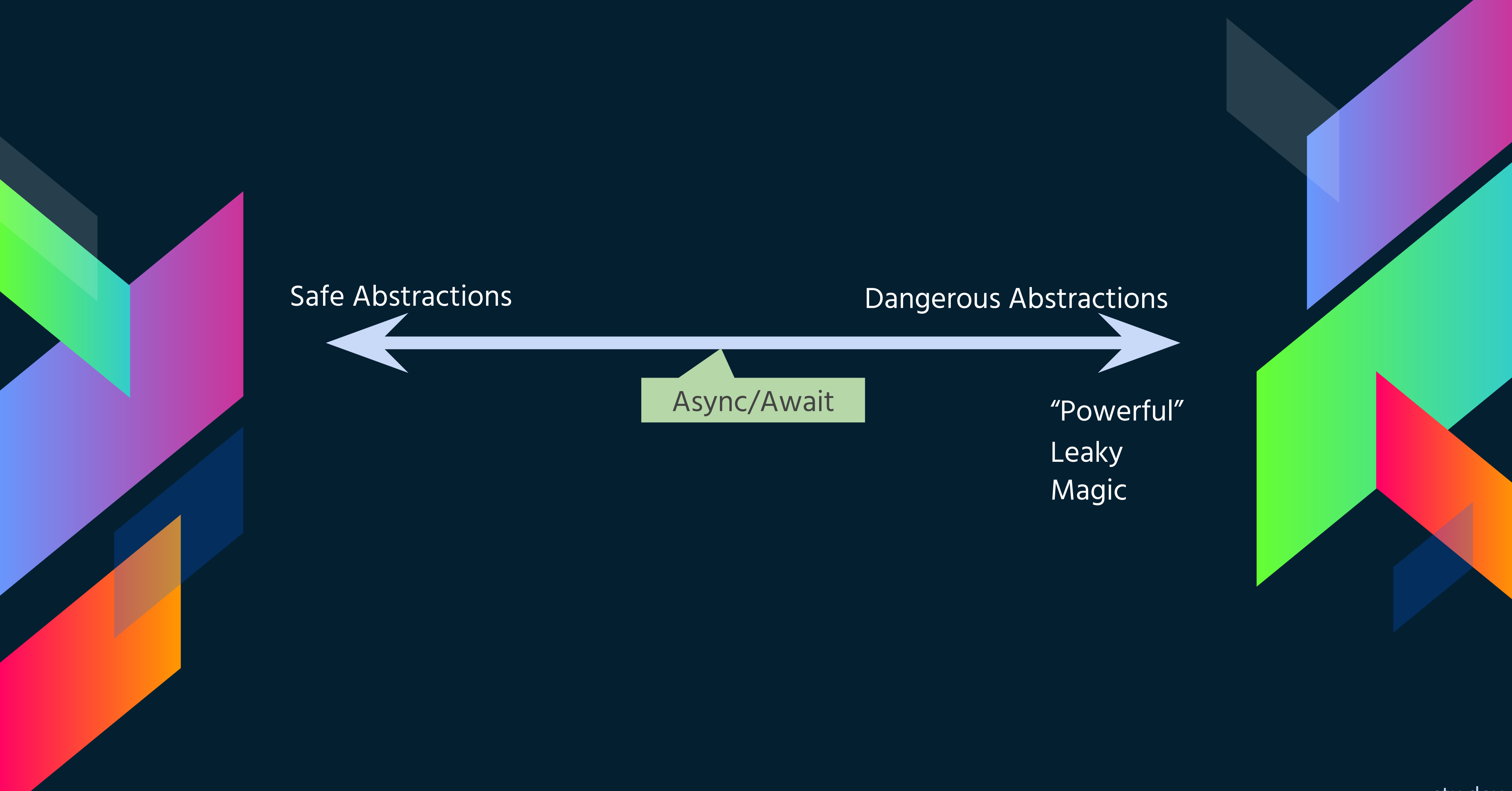
How does it work?

```
string result = await MyAsyncMethod();

async Task<string> MyAsyncMethod()
{
    Console.WriteLine("A");
    await Task.Delay(100);
    Console.WriteLine("B");
    await Task.Delay(100);
    Console.WriteLine("C");
    return "Dunzo";
}
```

[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  }
```


Safe Abstractions
Dangerous Abstractions
Async/Await
“Powerful”
Leaky
Magic

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

Kathleen Dollard - .NET Rocks! #1143

Link 🔗

AspNetCoreDiagnosticScenarios

github.com/davidfowl/AspNetCoreDiagnosticScenarios/blob/master/AsyncGuidance.md

davidfowl / AspNetCoreDiagnosticScenarios

Code | Issues 13 | Pull requests 0 | Projects 0 | Actions | Wiki | Security | Pulse | Community

Branch: master ▾ AspNetCoreDiagnosticScenarios / AsyncGuidance.md

Find file | Copy path

kimsey0 Update StreamWriter Dispose recommendations for C# 8 (#60) 872ac1c on 23 Jan

18 contributors

880 lines (686 sloc) | 32.7 KB

Raw | Blame | History

Table of contents

- Asynchronous Programming
  - Asynchrony is viral
  - Async void
  - Prefer Task.FromResult over Task.Run for pre-computed or trivially computed data
  - Avoid using Task.Run for long running work that blocks the thread
  - Avoid using Task.Result and Task.Wait
  - Prefer await over ContinueWith
  - Always create TaskCompletionSource<T> with TaskCreationOptions.RunContinuationsAsynchronously
  - Always dispose CancellationTokenSource(s) used for timeouts

stu.dev

Asynchrony is viral - ❌ Bad

```
public int DoSomethingAsync()
{
    var result = CallDependencyAsync().Result;
    return result + 1;
}
```

Asynchrony is viral - ✔ Good

```
public async Task<int> DoSomethingAsync()
{
    var result = await CallDependencyAsync();
    return result + 1;
}
```

Async void - ❌ Bad

```
public class MyController : Controller
{
    [HttpPost("/start")]
    public IActionResult Post()
    {
        BackgroundOperationAsync();
        return Accepted();
    }

    public async void BackgroundOperationAsync()
    {
        var result = await CallDependencyAsync();
        DoSomething(result);
    }
}
```

Async void - ✓ Good

```
public class MyController : Controller
{
    [HttpPost("/start")]
    public IActionResult Post()
    {
        Task.Run(BackgroundOperationAsync);
        return Accepted();
    }

    public async Task BackgroundOperationAsync()
    {
        var result = await CallDependencyAsync();
        DoSomething(result);
    }
}
```

CancellationTokens - ❌ Bad

```
public static async Task<T> WithCancellation<T>(this Task<T> task, CancellationToken cancellationToken)
{
    // There's no way to dispose the registration
    var delayTask = Task.Delay(-1, cancellationToken);

    var resultTask = await Task.WhenAny(task, delayTask);
    if (resultTask == delayTask)
    {
        // Operation cancelled
        throw new OperationCanceledException();
    }

    return await task
}
```

CancellationTokens - ✔️ Good

```
public static async Task<T> WithCancellation<T>(this Task<T> task, CancellationToken cancellationToken)
{
    var tcs = new TaskCompletionSource<object>(TaskCreationOptions.RunContinuationsAsynchronously);

    // This disposes the registration as soon as one of the tasks trigger
    using (cancellationToken.Register(state =>
    {
        ((TaskCompletionSource<object>)state).TrySetResult(null);
    },
    tcs))
    {
        var resultTask = await Task.WhenAny(task, tcs.Task);
        if (resultTask == tcs.Task)
        {
            // Operation cancelled
            throw new OperationCanceledException(cancellationToken);
        }

        return await task;
    }
}
```

Using a timeout - ❌ Bad

```
public static async Task<T> TimeoutAfter<T>(this Task<T> task, TimeSpan timeout)
{
    var delayTask = Task.Delay(timeout);

    var resultTask = await Task.WhenAny(task, delayTask);
    if (resultTask == delayTask)
    {
        // Operation cancelled
        throw new OperationCanceledException();
    }

    return await task;
}
```

Using a timeout - ✅ Good

```
public static async Task<T> TimeoutAfter<T>(this Task<T> task, TimeSpan timeout)
{
    using (var cts = new CancellationTokenSource())
    {
        var delayTask = Task.Delay(timeout, cts.Token);

        var resultTask = await Task.WhenAny(task, delayTask);
        if (resultTask == delayTask)
        {
            // Operation cancelled
            throw new OperationCanceledException();
        }
        else
        {
            // Cancel the timer task so that it does not fire
            cts.Cancel();
        }

        return await task;
    }
}
```

FlushAsync & Dispose - ❌ Bad

```
app.Run(async context =>
{
    // The implicit Dispose call will synchronously write to the response body
    using (var streamWriter = new StreamWriter(context.Response.Body))
    {
        await streamWriter.WriteAsync("Hello World");
    }
});
```

FlushAsync & Dispose - ✅ Good

```
app.Run(async context =>
{
    using (var streamWriter = new StreamWriter(context.Response.Body))
    {
        await streamWriter.WriteAsync("Hello World");

        // Force an asynchronous flush
        await streamWriter.FlushAsync();
    }
});
```

FlushAsync & Dispose - ✔️ Good

```
app.Run(async context =>
{
    // The implicit AsyncDispose call will flush asynchronously
    await using (var streamWriter = new StreamWriter(context.Response.Body))
    {
        await streamWriter.WriteAsync("Hello World");
    }
});
```

FlushAsync & Dispose - ✔️ Good

```
app.Run(async context =>
{
    // The implicit AsyncDispose call will flush asynchronously
    await using var streamWriter = new StreamWriter(context.Response.Body);
    await streamWriter.WriteAsync("Hello World");
});
```

async/await vs return Task- ❌ Bad

```
public Task<int> DoSomethingAsync()
{
    return CallDependencyAsync();
}
```

async/await vs return Task - ✅ Good

```
public async Task<int> DoSomethingAsync()
{
    return await CallDependencyAsync();
}
```

stu.dev

Prefer Task.FromResult over Task.Run - ❌ Bad

```
public class MyLibrary
{
    public Task<int> AddAsync(int a, int b)
    {
        return Task.Run(() => a + b);
    }
}
```

Prefer Task.FromResult over Task.Run - ✔️ Good

```
public class MyLibrary
{
    public Task<int> AddAsync(int a, int b)
    {
        return Task.FromResult(a + b);
    }
}
```

Prefer Task.FromResult over Task.Run - ✅ Good

```
public class MyLibrary
{
    public ValueTask<int> AddAsync(int a, int b)
    {
        return new ValueTask<int>(a + b);
    }
}
```

TaskCompletionSource - ❌ Bad

```
public Task<int> DoSomethingAsync()
{
    var tcs = new TaskCompletionSource<int>();

    var operation = new LegacyAsyncOperation();
    operation.Completed += result =>
    {
        // Code awaiting on this task will resume on this thread!
        tcs.SetResult(result);
    };

    return tcs.Task;
}
```

TaskCompletionSource - ✓ Good

```
public Task<int> DoSomethingAsync()
{
    var tcs = new TaskCompletionSource<int>(TaskCreationOptions.RunContinuationsAsynchronously);

    var operation = new LegacyAsyncOperation();
    operation.Completed += result =>
    {
        // Code awaiting on this task will resume on a different thread-pool thread
        tcs.SetResult(result);
    };

    return tcs.Task;
}
```

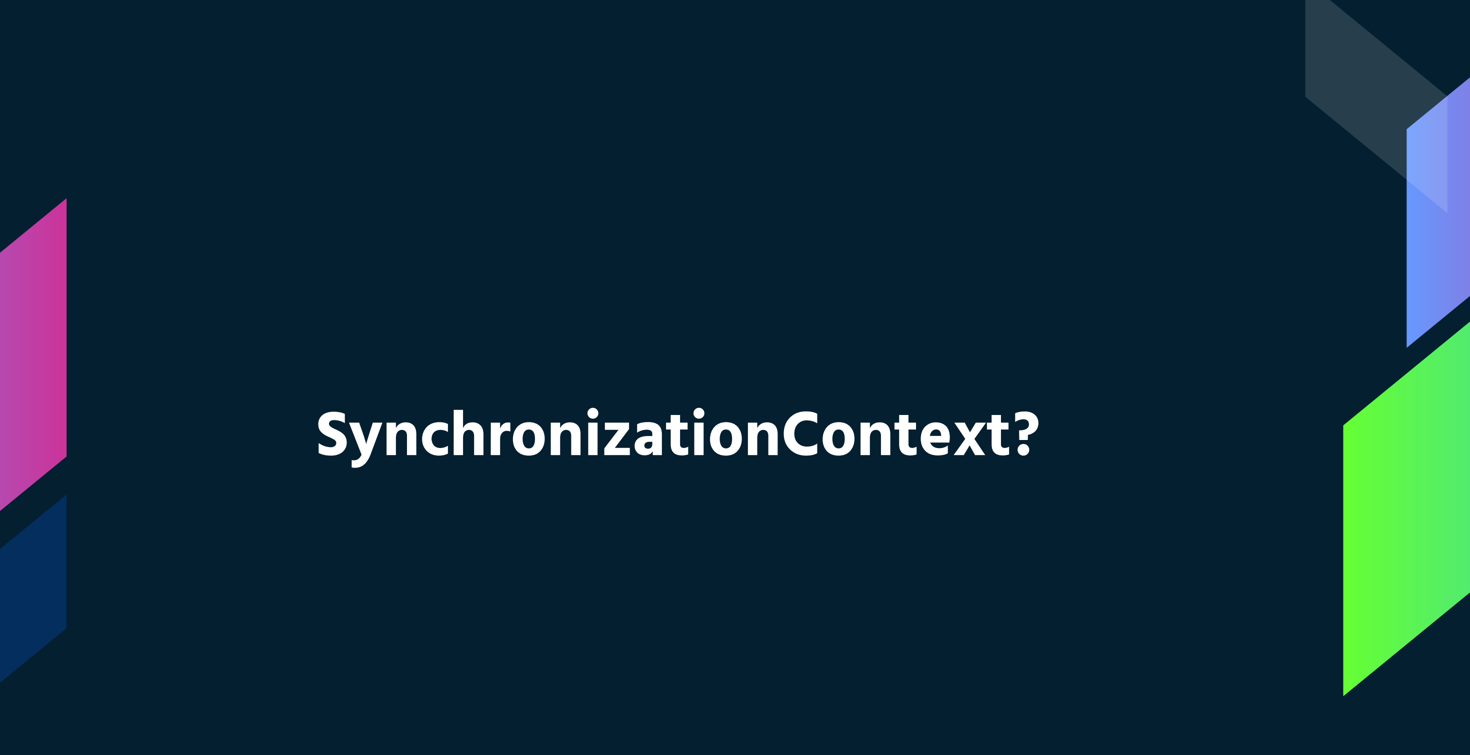
Avoid Task.Run for long blocking - ❌ Bad

```
public class QueueProcessor
{
    private readonly BlockingCollection<Message> _messageQueue = new BlockingCollection<Message>();

    public void StartProcessing()
    {
        Task.Run(ProcessQueue);
    }

    public void Enqueue(Message message)
    {
        _messageQueue.Add(message);
    }

    private void ProcessQueue()
    {
        foreach (var item in _messageQueue.GetConsumingEnumerable())
        {
            ProcessItem(item);
        }
    }

    private void ProcessItem(Message message) { }
}
```

Avoid Task.Run for long blocking - ✔️ Good

```
public class QueueProcessor
{
    private readonly BlockingCollection<Message> _messageQueue = new BlockingCollection<Message>();

    public void StartProcessing()
    {
        var thread = new Thread(ProcessQueue)
        {
            // This is important as it allows the process to exit while this thread is running
            IsBackground = true
        };
        thread.Start();
    }

    public void Enqueue(Message message)
    {
        _messageQueue.Add(message);
    }

    private void ProcessQueue()
    {
        foreach (var item in _messageQueue.GetConsumingEnumerable())
        {
             ProcessItem(item);
        }
    }

    private void ProcessItem(Message message) { }
}
```

Do not use .Wait() or
.Result ❌



SynchronizationContext?

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  }
```

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).ConfigureAwait(false);
9       Console.WriteLine("C");
10      return "Dunzo";
11  }
```

Synchronization Context

```
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

WindowsFormsSynchronizationContext

DispatcherSynchronizationContext

Default (ThreadPool) SynchronizationContext

AspNetSynchronizationContext

Synchronization Context

	Specific Thread Used to Execute Delegates	Exclusive (Delegates Execute One at a Time)
WinForms	Yes	Yes
WPF	Yes	Yes
Default	No	No
ASP.NET	No	Yes

```
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();
}
```

```csharp
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

```csharp
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();
}
```

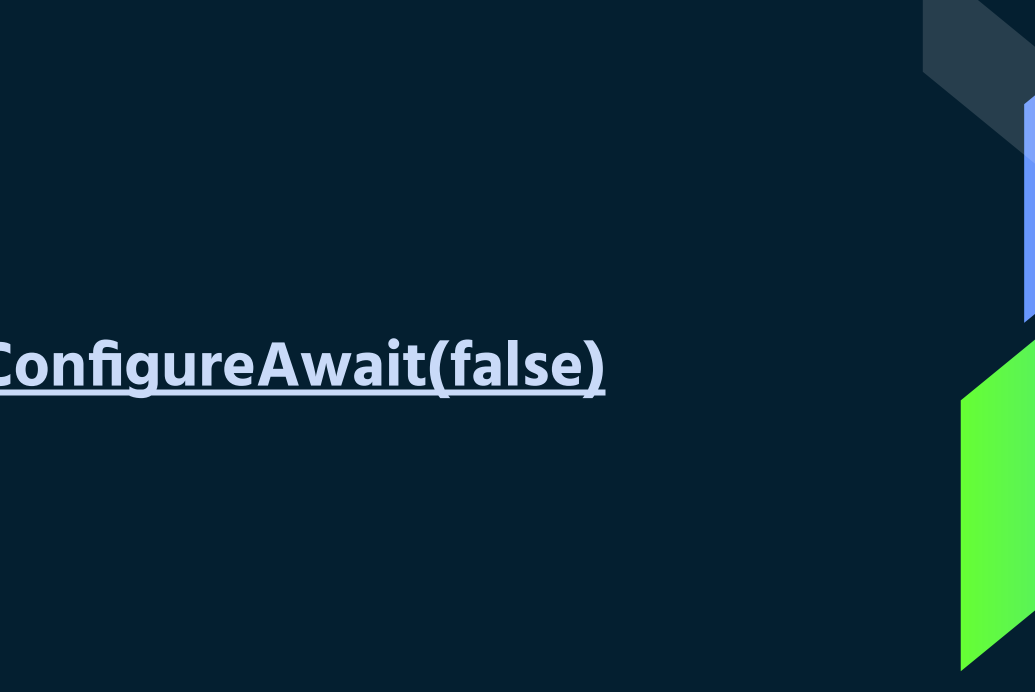
#	Method Name	Context Type	Thread ID
→	About	AspNetSynchronizationContext	17
→	CountdownAsync	AspNetSynchronizationContext	17
→	CountdownAsync	AspNetSynchronizationContext	17
→	CountdownAsync	AspNetSynchronizationContext	17
→	CountdownAsync	AspNetSynchronizationContext	17
→	CountdownAsync	AspNetSynchronizationContext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

ContextFreeTask

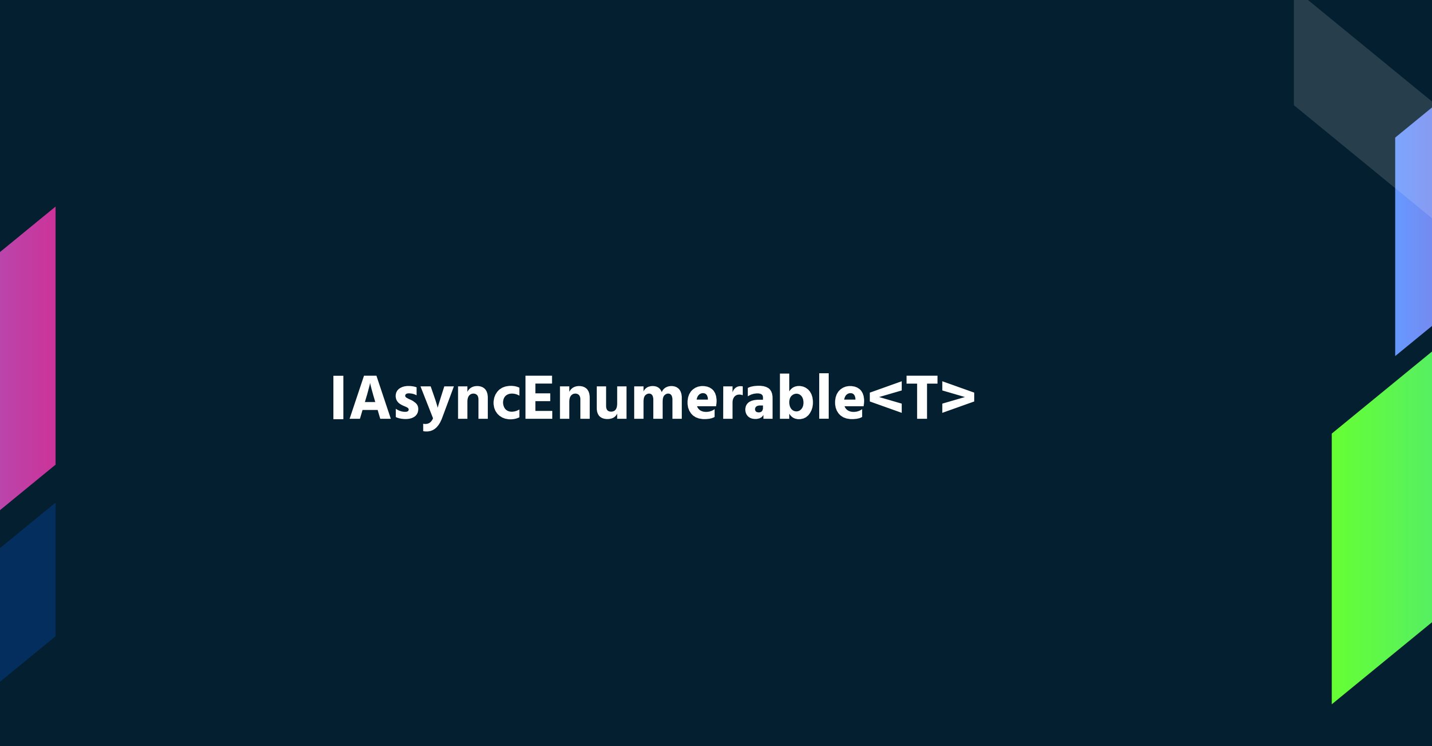
```csharp
public async Task<ActionResult> About()
{
    async Task<int> CountdownAsync(int number)
    {
        using (new AsyncTrace(_trace, nameof(CountdownAsync)))
        {
            if (number > 1)
                number = await CountdownAsync(number - 1);
            await Task.Delay(100);
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


.ConfigureAwait(false)

stu.dev


IAsyncEnumerable<T>

The problem

```
public async Task<IEnumerable<User>> GetUsers()
{
    var allResults = new List<User>();
    var nextUrl = "https://account.zendesk.com/api/v2/users.json";
    while (nextUrl != null)
    {
        var page = await _client.GetAsync(nextUrl)
                                .Content.ReadAsAsync<UsersListResponse>();

        allResults.AddRange(page.Users);
        nextUrl = page.NextPage;
        // eg "https://account.zendesk.com/api/v2/users.json?page=2"
    }
    return allResults;
}
```

Familiar definition

```
Next* (Error|Completed)?
```

Familiar definition

push

IObservable<T>

synchronous

asynchronous

IEnumerable<T>

IAsyncEnumerable<T>

pull

```csharp
public interface IAsyncEnumerable<out T>
{
    IAsyncEnumerator<T> GetAsyncEnumerator(CancellationToken cancellationToken
 = default (CancellationToken));
}

public interface IAsyncEnumerator<out T> : IAsyncDisposable
{
    ValueTask<bool> MoveNextAsync();

    T Current { get; }
}
```

Demo - IAsyncEnumerable<T> & Channels

THANKS!

Any questions?

You can find me at:

[stu.dev]