Turbocharged:

Measuring and Optimising Application Performance

@stevejgordon
https://stevejgordon.co.uk

http://bit.ly/TurbochargedWorkshop

MVP
Microsoft
Most Valuable Professional

.NET southeast
https://www.meetup.com/dotnetsoutheast

PLURALSIGHT

madgex

EXPECTATIONS

What we'll cover

- What is performance?
- Measuring application and code performance
- Span<T>, ReadOnlySpan<T> and Memory<T>
- ArrayPool
- System.IO.Pipelines and ReadOnlySequence<T>
- .NET Core 3.0 JSON APIs

Aspects of Performance

Execution Time

Throughput

Memory Allocations

"We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%."

Donald Knuth, 1974, Structured Programming with go to Statements

http://web.archive.org/web/20130731202547/http://pplab.snu.ac.kr/courses/adv_pl05/papers/p261-knuth.pdf

PERFORMANCE

IS

CONTEXTUAL

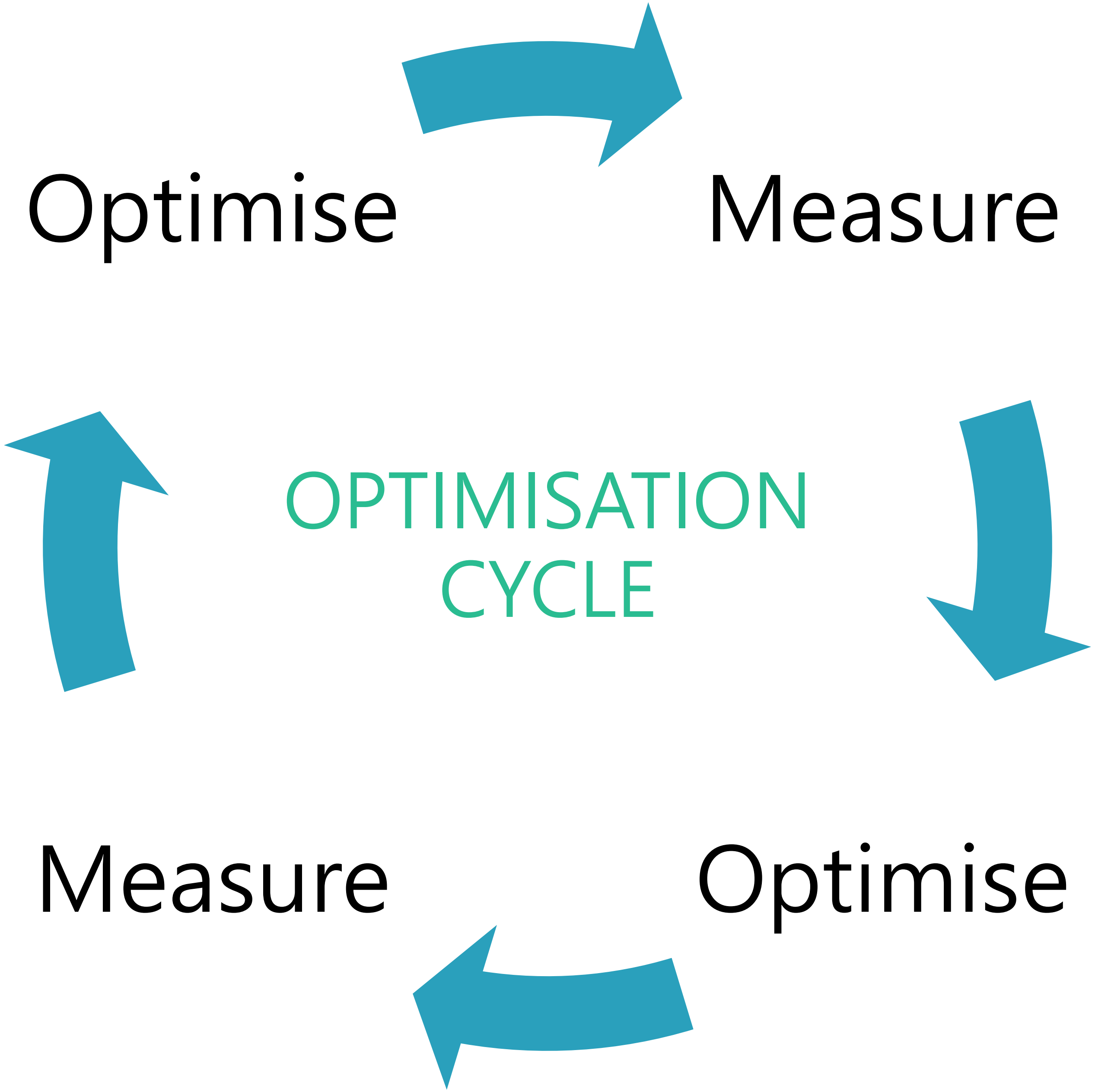
PERFORMANCE SHOULD

BE A PART OF

EVERY STORY

PERFORMANCE

READABILITY



#MOMO

Measuring Application Performance

- Visual Studio Diagnostic Tools (debugging)
- Visual Studio Profiling / PerfView / dotTrace / dotMemory
- ILSpy / JustDecompile / dotPeek

- Production metrics and monitoring

Benchmark .NET

- Library for .NET (micro)benchmarking
- High precision measurements
- Extra data and output available using diagnosers
- Compare performance on different platforms, architectures, JIT versions and GC Modes
- Used extensively in CoreFx, CoreClr and ASP.NET Core

- https://benchmarkdotnet.org
- https://github.com/dotnet/BenchmarkDotNet

```csharp
namespace BenchmarkExample
{
    public class Program
    {
        public static void Main(string[] args) =>
            _ = BenchmarkRunner.Run<NameParserBenchmarks>();
    }

    [MemoryDiagnoser]
    public class NameParserBenchmarks
    {
        private const string FullName = "Steve J Gordon";
        private static readonly NameParser Parser = new NameParser();

        [Benchmark]
        public void GetLastName()
        {
            Parser.GetLastName(FullName);
        }
    }
}
```

```csharp
namespace BenchmarkExample
{
    public class Program
    {
        public static void Main(string[] args) =>
            _ = BenchmarkRunner.Run<NameParserBenchmarks>();
    }

    [MemoryDiagnoser]
    public class NameParserBenchmarks
    {
        private const string FullName = "Steve J Gordon";
        private static readonly NameParser Parser = new NameParser();

        [Benchmark]
        public void GetLastName()
        {
            Parser.GetLastName(FullName);
        }
    }
}
```

```csharp
namespace BenchmarkExample
{
    public class Program
    {
        public static void Main(string[] args) =>
            _ = BenchmarkRunner.Run<NameParserBenchmarks>();
    }

    [MemoryDiagnoser]
    public class NameParserBenchmarks
    {
        private const string FullName = "Steve J Gordon";
        private static readonly NameParser Parser = new NameParser();

        [Benchmark]
        public void GetLastName()
        {
            Parser.GetLastName(FullName);
        }
    }
}
```

```csharp
namespace BenchmarkExample
{
    public class Program
    {
        public static void Main(string[] args) =>
            _ = BenchmarkRunner.Run<NameParserBenchmarks>();
    }

    [MemoryDiagnoser]
    public class NameParserBenchmarks
    {
        private const string FullName = "Steve J Gordon";
        private static readonly NameParser Parser = new NameParser();

        [Benchmark]
        public void GetLastName()
        {
            Parser.GetLastName(FullName);
        }
    }
}
```

```
namespace BenchmarkExample
{
    public class Program
    {
        public static void Main(string[] args) =>
            _ = BenchmarkRunner.Run<NameParserBenchmarks>();
    }

    [MemoryDiagnoser]
    public class NameParserBenchmarks
    {
        private const string FullName = "Steve J Gordon";
        private static readonly NameParser Parser = new NameParser();

        [Benchmark]
        public void GetLastName()
        {
            Parser.GetLastName(FullName);
        }
    }
}
```

```
// * Summary *

BenchmarkDotNet=v0.11.3, OS=Windows 10.0.17134.706 (1803/April2018Update/Redstone4)
Intel Core i7-6700 CPU 3.40GHz (Skylake), 1 CPU, 8 logical and 4 physical cores
.NET Core SDK=3.0.100-preview3-010410
  [Host]     : .NET Core 2.2.3 (CoreCLR 4.6.27414.05, CoreFX 4.6.27414.05), 64bit RyuJIT
  DefaultJob : .NET Core 2.2.3 (CoreCLR 4.6.27414.05, CoreFX 4.6.27414.05), 64bit RyuJIT


      Method |      Mean |     Error |    StdDev |    Median | Gen 0 | Gen 1 | Gen 2 | Allocated |
             |           |           |           |           |/1k Op |/1k Op |/1k Op | Memory/Op |
------------ |----------:|----------:|----------:|----------:|------:|------:|------:|----------:|
 GetLastName | 163.18 ns | 3.1903 ns | 4.2590 ns | 161.87 ns | 0.0379 |     - |     - |     160 B |
```

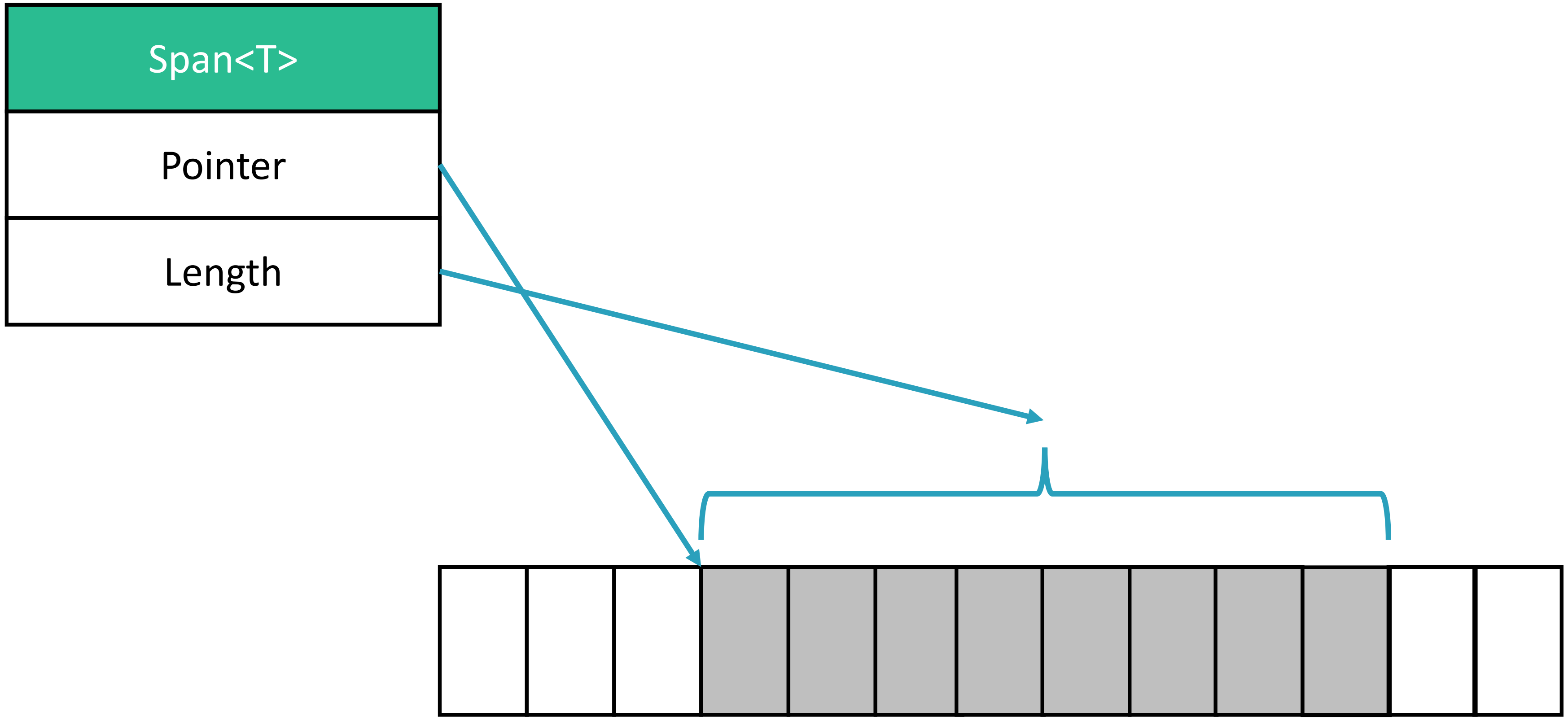
(1 / 0.0379) x 1000 = 26,385.2 operations before Gen 0 collection.

ACTIVITY 01
BENCHMARKING

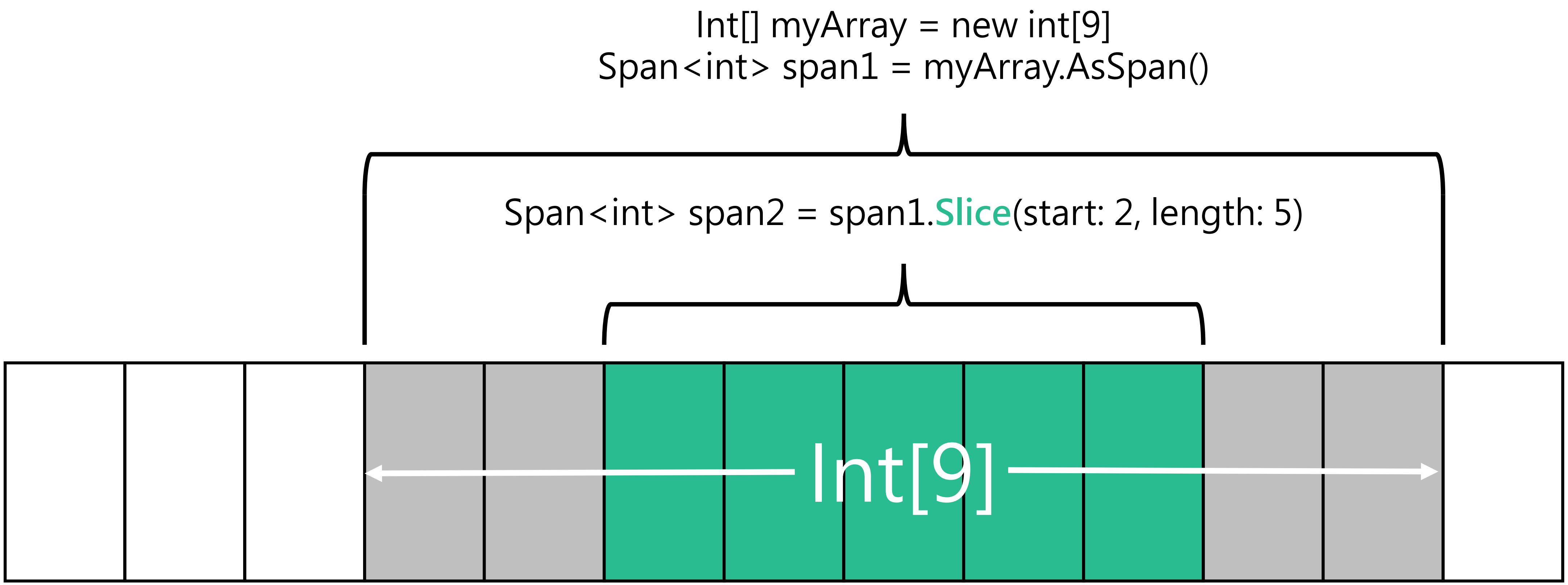
Span<T> and ReadOnlySpan<T>

- **System.Memory** package. Built into .NET Core 2.1.
- Provides a read/write 'view' onto a contiguous region of memory
  - Heap (Managed objects) – e.g. Arrays, Strings
  - Stack (via stackalloc)
  - Native/Unmanaged (P/Invoke)
- Index / Iterate to modify the memory within the Span
- Almost no overhead
- Thread safe – Only accessible by one thread at a time
- ReadOnlySpan<T> is used with strings

Span<T>



Span<T>.Slice



Slicing a Span is a constant time/cost operation – O(1)

OPTIMISING SOME CODE

```csharp
[MemoryDiagnoser]
public class ArrayBenchmarks
{
    private int[] _myArray;
    private static readonly Consumer Consumer = new Consumer();

    [Params(10, 1000, 10000)]
    public int Size { get; set; }

    [GlobalSetup]
    public void Setup()
    {
        _myArray = new int[Size];

        for (var i = 0; i < Size; i++)
            _myArray[i] = i;
    }

    …
}
```

```
[MemoryDiagnoser]
public class ArrayBenchmarks
{
    private int[] _myArray;
    private static readonly Consumer Consumer = new Consumer();

    [Params(10, 1000, 10000)]
    public int Size { get; set; }

    [GlobalSetup]
    public void Setup()
    {
        _myArray = new int[Size];

        for (var i = 0; i < Size; i++)
            _myArray[i] = i;
    }

    …
}
```

```csharp
[MemoryDiagnoser]
public class ArrayBenchmarks
{
    private int[] _myArray;
    private static readonly Consumer Consumer = new Consumer();

    [Params(10, 1000, 10000)]
    public int Size { get; set; }

    [GlobalSetup]
    public void Setup()
    {
        _myArray = new int[Size];

        for (var i = 0; i < Size; i++)
            _myArray[i] = i;
    }

    …
}
```

```
[MemoryDiagnoser]
public class ArrayBenchmarks
{
    private int[] _myArray;
    private static readonly Consumer Consumer = new Consumer();

    [Params(10, 1000, 10000)]
    public int Size { get; set; }

    [GlobalSetup]
    public void Setup()
    {
        _myArray = new int[Size];

        for (var i = 0; i < Size; i++)
            _myArray[i] = i;
    }

    …
}
```

```
[MemoryDiagnoser]
public class ArrayBenchmarks
{


    …

    [Benchmark(Baseline = true)]
    public void Original() =>
        _myArray.Skip(Size / 2).Take(Size / 4)

    …

}
```

```
[MemoryDiagnoser]
public class ArrayBenchmarks
{
    private static readonly Consumer Consumer = new Consumer();

    …

    [Benchmark(Baseline = true)]
    public void Original() =>
        _myArray.Skip(Size / 2).Take(Size / 4).Consume(Consumer);

    …

}
```

Method	Size	Mean	Ratio	Gen 0	Gen 1	Gen 2	Allocated
Original	10	72.8695 ns	1.00	0.0151	-	-	96 B
Original	1000	2,109.1965 ns	1.000	0.0114	-	-	96 B
Original	10000	20,220.9778 ns	1.000	-	-	-	96 B

```
[MemoryDiagnoser]
public class ArrayBenchmarks
{
    …

    [Benchmark]
    public int[] ArrayCopy()
    {
        var newArray = new int[Size / 4];
        Array.Copy(_myArray, Size / 2, newArray, 0, Size / 4);
        return newArray;
    }

    …

}
```

Method	Size	Mean	Ratio	Gen 0	Gen 1	Gen 2	Allocated
Original	10	72.8695 ns	1.00	0.0151	-	-	96 B
ArrayCopy	10	12.6507 ns	0.17	0.0051	-	-	32 B
Original	1000	2,109.1965 ns	1.000	0.0114	-	-	96 B
ArrayCopy	1000	75.1658 ns	0.036	0.1627	0.0005	-	1024 B
Original	10000	20,220.9778 ns	1.000	-	-	-	96 B
ArrayCopy	10000	662.3187 ns	0.033	1.5917	0.0505	-	10024 B

```
[MemoryDiagnoser]
public class ArrayBenchmarks
{
    …

    [Benchmark]
    public void NewArray()
    {
        var newArray = new int[Size / 4];

        for (var i = 0; i < Size / 4; i++)
            newArray[i] = _myArray[(Size / 2) + i];
    }

    …

}
```

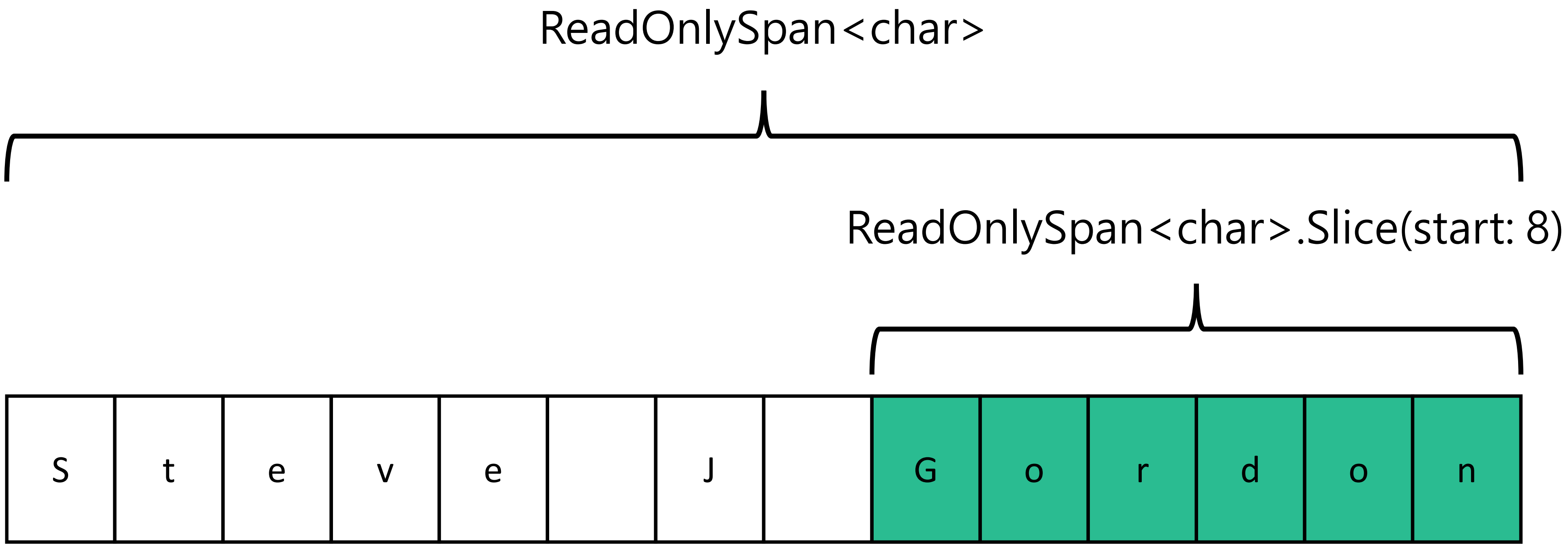
Method	Size	Mean	Ratio	Gen 0	Gen 1	Gen 2	Allocated
Original	10	72.8695 ns	1.00	0.0151	-	-	96 B
ArrayCopy	10	12.6507 ns	0.17	0.0051	-	-	32 B
NewArray	10	4.9848 ns	0.07	0.0051	-	-	32 B
Original	1000	2,109.1965 ns	1.000	0.0114	-	-	96 B
ArrayCopy	1000	75.1658 ns	0.036	0.1627	0.0005	-	1024 B
NewArray	1000	202.8316 ns	0.096	0.1626	-	-	1024 B
Original	10000	20,220.9778 ns	1.000	-	-	-	96 B
ArrayCopy	10000	662.3187 ns	0.033	1.5917	0.0505	-	10024 B
NewArray	10000	1,947.1885 ns	0.096	1.5907	-	-	10024 B

```
[MemoryDiagnoser]
public class ArrayBenchmarks
{
    …

    [Benchmark]
    public Span<int> Span() => _myArray.AsSpan().Slice(Size / 2, Size / 4);

    …

}
```

Method	Size	Mean	Ratio	Gen 0	Gen 1	Gen 2	Allocated
Original	10	72.8695 ns	1.00	0.0151	-	-	96 B
ArrayCopy	10	12.6507 ns	0.17	0.0051	-	-	32 B
NewArray	10	4.9848 ns	0.07	0.0051	-	-	32 B
Span	10	0.9999 ns	0.01	-	-	-	-
Original	1000	2,109.1965 ns	1.000	0.0114	-	-	96 B
ArrayCopy	1000	75.1658 ns	0.036	0.1627	0.0005	-	1024 B
NewArray	1000	202.8316 ns	0.096	0.1626	-	-	1024 B
Span	1000	1.0034 ns	0.000	-	-	-	-
Original	10000	20,220.9778 ns	1.000	-	-	-	96 B
ArrayCopy	10000	662.3187 ns	0.033	1.5917	0.0505	-	10024 B
NewArray	10000	1,947.1885 ns	0.096	1.5907	-	-	10024 B
Span	10000	1.0125 ns	0.000	-	-	-	-

Working with Strings

```
ReadOnlySpan<char> span = "Some string data".AsSpan();
```



Span<T> IndexOf

- A common operation when parsing is to slice through a Span<T>/ReadOnlySpan<T> looking for specific characters.
- IndexOf(char) can be used for this.
- Often this is done within a loop, slicing the input each time to continue for the rest of the string.

```
var myString = "A String";

ReadOnlySpan mySpan = myString.AsSpan();

var indexOfSpace = mySpan.IndexOf(' '); // value is 1
```

ACTIVITY 02
APPLYING SPAN

WORKING ON THE STACK

```csharp
public void CalculateFibonacci()
{
    const int arraySize = 20;
    Span<int> fib = stackalloc int[arraySize];

    fib[0] = fib[1] = 1; // Sequence starts with 1

    for (int i = 2; i < arraySize; ++i)
    {
        // Sum the previous two numbers.
        fib[i] = fib[i-1] + fib[i-2];
    }
}
```

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/stackalloc

```
public void CalculateFibonacci()
{
    const int arraySize = 20;
    Span<int> fib = stackalloc int[arraySize];

    fib[0] = fib[1] = 1; // Sequence starts with 1

    for (int i = 2; i < arraySize; ++i)
    {
        // Sum the previous two numbers.
        fib[i] = fib[i-1] + fib[i-2];
    }
}
```

```csharp
public void CalculateFibonacci()
{
    const int arraySize = 20;
    Span<int> fib = stackalloc int[arraySize];

    fib[0] = fib[1] = 1; // Sequence starts with 1

    for (int i = 2; i < arraySize; ++i)
    {
        // Sum the previous two numbers.
        fib[i] = fib[i-1] + fib[i-2];
    }
}
```

```csharp
public Span<int> CalculateFibonacci()
{
    const int arraySize = 20;
    Span<int> fib = stackalloc int[arraySize];

    fib[0] = fib[1] = 1; // Sequence starts with 1

    for (int i = 2; i < arraySize; ++i)
    {
        // Sum the previous two numbers.
        fib[i] = fib[i-1] + fib[i-2];
    }

    return fib;
}
```

```csharp
public Span<int> CalculateFibonacci()
{
    const int arraySize = 20;
    Span<int> fib = stackalloc int[arraySize];

    fib[0] = fib[1] = 1; // Sequence starts with 1

    for (int i = 2; i < arraySize; ++i)
    {
        // Sum the previous two numbers.
        fib[i] = fib[i-1] + fib[i-2];
    }

    return fib; // CS8325 - Cannot use local 'fib' in this context
                // because it may expose referenced variables
                // outside of their declaration scope
}
```

Parameterised Benchmarks

- Test more than one expected input
- You can mark one or several fields or properties in your class by the [Params] attribute
- Every value must be a compile-time constant
- A benchmark will run for each combination of param values

```csharp
public class IntroParams
{
    [Params(100, 200)]
    public int A { get; set; }

    [Params(10, 20)]
    public int B { get; set; }

    [Benchmark]
    public void Benchmark() => Thread.Sleep(A + B + 5);
}
```

```
public class IntroParams
{
    [Params(100, 200)]
    public int A { get; set; }

    [Params(10, 20)]
    public int B { get; set; }

    [Benchmark]
    public void Benchmark() => Thread.Sleep(A + B + 5);
}
```

```
public class IntroParams
{
      [Params(100, 200)]
      public int A { get; set; }

      [Params(10, 20)]
      public int B { get; set; }

      [Benchmark]
      public void Benchmark() => Thread.Sleep(A + B + 5);
}
```

```csharp
public class IntroParams
{
    [Params(100, 200)]
    public int A { get; set; }

    [Params(10, 20)]
    public int B { get; set; }

    [Benchmark]
    public void Benchmark() => Thread.Sleep(A + B + 5);
}
```

Method	A	B	Mean	Error	StdDev
Benchmark	100	10	115.9 ms	0.1497 ms	0.1401 ms
Benchmark	100	20	125.9 ms	0.0414 ms	0.0387 ms
Benchmark	200	10	215.8 ms	0.2248 ms	0.2103 ms
Benchmark	200	20	225.7 ms	0.2407 ms	0.2251 ms

ACTIVITY 03
PARAMATERISED BENCHMARKS

Span<T> Limitations

- It's a **stack only** Value Type (ref struct) – Cannot live on the heap
- Requires C# 7.2+ for ref struct feature
- Cannot be boxed
- Cannot be a field in a class or standard (non ref) struct
- Cannot be used as an argument or local variable inside async methods
- Cannot be captured by lambda expressions
- Cannot be used as a generic type argument

Memory<T>

- Similar to Span<T> but can live on the heap
- A readonly struct but not a ref struct
- Slightly slower to slice into Memory<T>
- Can call Span property to get a span within a method

```
private async Task SomethingAsync(Span<byte> data)
{
    ... // Would be nice to do something with the Span here

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

```csharp
// CS4012 Parameters or locals of type 'Span<byte>' cannot be declared
// in async methods or lambda expressions.
private async Task SomethingAsync(Span<byte> data)
{
    ... // Would be nice to do something with the Span here

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

```
private async Task SomethingAsync(Memory<byte> data)
{
    ...

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

```
private async Task SomethingAsync(Memory<byte> data)
{
    Memory<byte> dataSliced = data.Slice(0, 100);

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

```csharp
private async Task SomethingAsync(Memory<byte> data)
{
    Memory<byte> dataSliced = data.Slice(0, 100);

    await Task.Delay(1000);
}

private void SomethingNotAsync(Span<byte> data)
{
    // some code
}
```

```csharp
private async Task SomethingAsync(Memory<byte> data)
{
    // CS4012 Parameters or locals of type 'Span<byte>' cannot be declared
    // in async methods or lambda expressions.
    var span = data.Span.Slice(1);

    SomethingNotAsync(span);

    await Task.Delay(1000);
}

private void SomethingNotAsync(Span<byte> data)
{
    // some code
}
```

```csharp
private async Task SomethingAsync(Memory<byte> data)
{
    SomethingNotAsync(data.Span.Slice(1));

    await Task.Delay(1000);
}

private void SomethingNotAsync(Span<byte> data)
{
    // some code
}
```

String.Create

```
String.Create(int length, TState state, SpanAction<char, TState> action)
```

- Create a string by manipulating heap memory at creation
- You must know the character length in advance
- Provide state which will be transformed into the string to avoid closures and optimise caching
- For state from multiple sources, use a ValueTuple

STRING.CREATE DEMO

ACTIVITY 04
STRING.CREATE

Putting it into practice – Key Builder

Microservice which:

1. Reads SQS message
2. Deserialises the JSON message
3. Stores a copy of the message to S3 using an object key derived from properties of the message.

S3ObjectKeyGenerator

Object Key Builder Benchmarks

Method	Mean	Ratio	Gen 0	Gen 1	Gen 2	Allocated
Original	1,088.0 ns	1.00	0.1812	-	-	1144 B
SpanBased	449.0 ns	0.41	0.0305	-	-	192 B
StringCreate	442.9 ns	0.41	0.0305	-	-	192 B

~2.5x Faster
~6x Less Allocations

18 million messages:

Reduction of 17GB of allocations daily

Removes approx. 2711 Gen 0 collections (562 vs. 3273)

ArrayPool

- Pool of arrays for re-use
- Found in System.Buffers
- ArrayPool<T>.Shared.Rent(int length)
- You are likely to get an array larger than your minimum size
- ArrayPool<T>.Shared.Return(T[] array, bool clearArray = false)
- Warning: By default returned arrays are not cleared!
- https://adamsitnik.com/Array-Pool/

```csharp
public class Processor
{
    public void DoSomeWorkVeryOften()
    {
        var buffer = new byte[1000];  // allocates

        DoSomethingWithBuffer(buffer);
    }

    private void DoSomethingWithBuffer(byte[] buffer)
    {
        // use the array
    }
}
```

```csharp
public class Processor
{
    public void DoSomeWorkVeryOften()
    {
        var buffer = new byte[1000];  // allocates

        DoSomethingWithBuffer(buffer);
    }

    private void DoSomethingWithBuffer(byte[] buffer)
    {
        // use the array
    }
}
```

```
public class Processor
{
    public void DoSomeWorkVeryOften()
    {
        var arrayPool = ArrayPool<byte>.Shared;
        var buffer = arrayPool.Rent(1000);

        DoSomethingWithBuffer(buffer);
    }

    private void DoSomethingWithBuffer(byte[] buffer)
    {
        // use the array
    }
}
```

```csharp
public class Processor
{
    public void DoSomeWorkVeryOften()
    {
        var arrayPool = ArrayPool<byte>.Shared;
        var buffer = arrayPool.Rent(1000);

        try
        {
            DoSomethingWithBuffer(buffer);
        }
        finally
        {
            arrayPool.Return(buffer);
        }
    }

    private void DoSomethingWithBuffer(byte[] buffer)
    {
        // use the array
    }
}
```

ArrayPool Benchmarks

Method	SizeInBytes	Mean	Gen 0	Gen 1	Gen 2	Allocated
RentAndReturn	20	29.397 ns	-	-	-	-
Allocate	20	6.563 ns	0.0115	-	-	48 B
RentAndReturn	100	28.797 ns	-	-	-	-
Allocate	100	13.349 ns	0.0306	-	-	128 B
RentAndReturn	1000	33.807 ns	-	-	-	-
Allocate	1000	84.908 ns	0.2447	-	-	1024 B
RentAndReturn	10000	35.387 ns	-	-	-	-
Allocate	10000	978.090 ns	2.3918	-	-	10024 B
RentAndReturn	100000	31.615 ns	-	-	-	-
Allocate	100000	12,875.858 ns	31.2347	31.2347	31.2347	100024 B

ACTIVITY 05
ARRAY POOL

System.IO.Pipelines

- Created by ASP.NET team to improve Kestrel requests per second.
- Improves I/O performance scenarios (~2x vs. streams)
- Removes common hard to write, boilerplate code
- Unlike streams, pipelines manages buffers for you from ArrayPool
- Two sides to a pipe, **PipeWriter** and **PipeReader**
- Can be awaited multiple times without multiple Task allocations in .NET Core 2.1 - IValueTaskSource

Pipes

PipeWriter : IBufferWriter<byte>

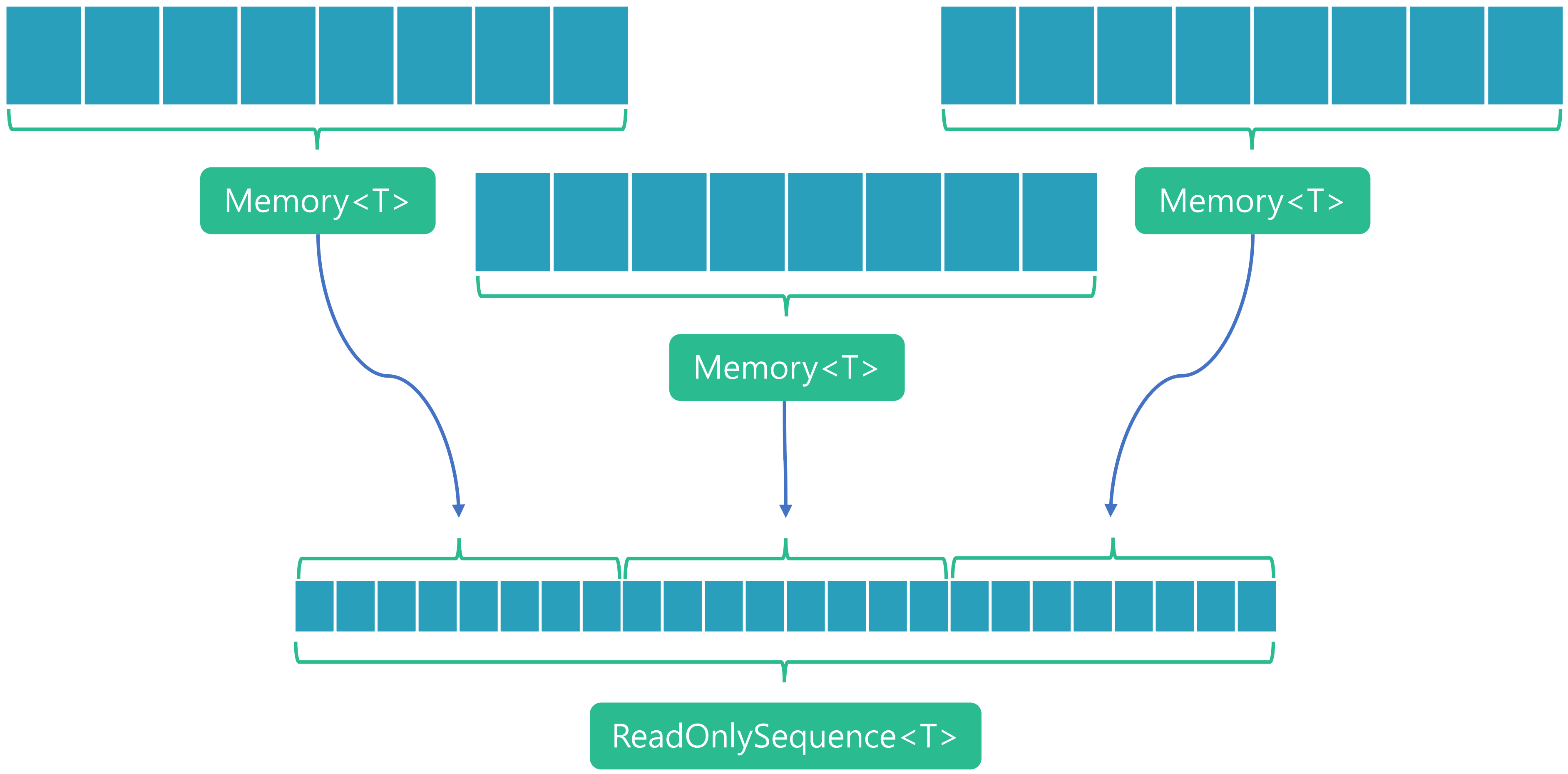
```
Memory<byte> m = pw.GetMemory();
…
pw.Advance(1000)
await pw.FlushAsync()
```

Pipe

```
ReadResult r = await reader.ReadAsync();
ReadOnlySequence<byte> b = r.Buffer;
```

PipeReader

ReadOnlySequence<T>



```csharp
private static async Task ProcessLinesAsync(Socket socket)
{
    var pipe = new Pipe();

    Task writing = FillPipeAsync(socket, pipe.Writer);
    Task reading = ReadPipeAsync(pipe.Reader);

    await Task.WhenAll(reading, writing);
 }
```

https://channel9.msdn.com/Shows/On-NET/High-performance-IO-with-SystemIOPipelines

```
private static async Task ProcessLinesAsync(Socket socket)
{
    var pipe = new Pipe();

    Task writing = FillPipeAsync(socket, pipe.Writer);
    Task reading = ReadPipeAsync(pipe.Reader);

    await Task.WhenAll(reading, writing);
 }
```

```csharp
private static async Task ProcessLinesAsync(Socket socket)
{
    var pipe = new Pipe();

    Task writing = FillPipeAsync(socket, pipe.Writer);
    Task reading = ReadPipeAsync(pipe.Reader);

    await Task.WhenAll(reading, writing);
 }
```

```
private static async Task ProcessLinesAsync(Socket socket)
{
    var pipe = new Pipe();

    Task writing = FillPipeAsync(socket, pipe.Writer);
    Task reading = ReadPipeAsync(pipe.Reader);

    await Task.WhenAll(reading, writing);
 }
```

```
private static async Task ProcessLinesAsync(Socket socket)
{
    var pipe = new Pipe();

    Task writing = FillPipeAsync(socket, pipe.Writer);
    Task reading = ReadPipeAsync(pipe.Reader);

    await Task.WhenAll(reading, writing);
 }
```

```csharp
private static async Task FillPipeAsync(Socket socket, PipeWriter writer)
{
    while (true)
    {
        try
        {
            Memory<byte> memory = writer.GetMemory();  // Request memory from the pipe
            int bytesRead = await socket.ReceiveAsync(memory, SocketFlags.None);

            if (bytesRead == 0)
                break;

            writer.Advance(bytesRead); // Tell the PipeWriter how much was read from the Socket
        }
        catch
        {
            break;
        }

        FlushResult result = await writer.FlushAsync(); // Make the data available to the PipeReader

        if (result.IsCompleted)
            break;
    }

    writer.Complete(); // Signal to the reader that we're done writing
}
```

```csharp
private static async Task FillPipeAsync(Socket socket, PipeWriter writer)
{
    while (true)
    {
        try
        {
            Memory<byte> memory = writer.GetMemory();  // Request memory from the pipe
            int bytesRead = await socket.ReceiveAsync(memory, SocketFlags.None);

            if (bytesRead == 0)
                break;

            writer.Advance(bytesRead); // Tell the PipeWriter how much was read from the Socket
        }
        catch
        {
            break;
        }

        FlushResult result = await writer.FlushAsync(); // Make the data available to the PipeReader

        if (result.IsCompleted)
            break;
    }

    writer.Complete(); // Signal to the reader that we're done writing
}
```

```
private static async Task FillPipeAsync(Socket socket, PipeWriter writer)
{
    while (true)
    {
        try
        {
            Memory<byte> memory = writer.GetMemory();  // Request memory from the pipe
            int bytesRead = await socket.ReceiveAsync(memory, SocketFlags.None);

            if (bytesRead == 0)
                break;

            writer.Advance(bytesRead); // Tell the PipeWriter how much was read from the Socket
        }
        catch
        {
            break;
        }

        FlushResult result = await writer.FlushAsync(); // Make the data available to the PipeReader

        if (result.IsCompleted)
            break;
    }

    writer.Complete(); // Signal to the reader that we're done writing
}
```

```csharp
private static async Task FillPipeAsync(Socket socket, PipeWriter writer)
{
    while (true)
    {
        try
        {
            Memory<byte> memory = writer.GetMemory();  // Request memory from the pipe
            int bytesRead = await socket.ReceiveAsync(memory, SocketFlags.None);

            if (bytesRead == 0)
                break;

            writer.Advance(bytesRead); // Tell the PipeWriter how much was read from the Socket
        }
        catch
        {
            break;
        }

        FlushResult result = await writer.FlushAsync(); // Make the data available to the PipeReader

        if (result.IsCompleted)
            break;
    }

    writer.Complete(); // Signal to the reader that we're done writing
}
```

```
private static async Task FillPipeAsync(Socket socket, PipeWriter writer)
{
    while (true)
    {
        try
        {
            Memory<byte> memory = writer.GetMemory();  // Request memory from the pipe
            int bytesRead = await socket.ReceiveAsync(memory, SocketFlags.None);

            if (bytesRead == 0)
                break;

            writer.Advance(bytesRead); // Tell the PipeWriter how much was read from the Socket
        }
        catch
        {
            break;
        }

        FlushResult result = await writer.FlushAsync(); // Make the data available to the PipeReader

        if (result.IsCompleted)
            break;
    }

    writer.Complete(); // Signal to the reader that we're done writing
}
```

```csharp
private static async Task FillPipeAsync(Socket socket, PipeWriter writer)
{
    while (true)
    {
        try
        {
            Memory<byte> memory = writer.GetMemory();  // Request memory from the pipe
            int bytesRead = await socket.ReceiveAsync(memory, SocketFlags.None);

            if (bytesRead == 0)
                break;

            writer.Advance(bytesRead); // Tell the PipeWriter how much was read from the Socket
        }
        catch
        {
            break;
        }

        FlushResult result = await writer.FlushAsync(); // Make the data available to the PipeReader

        if (result.IsCompleted)
            break;
    }

    writer.Complete(); // Signal to the reader that we're done writing
}
```

```
private static async Task FillPipeAsync(Socket socket, PipeWriter writer)
{
    while (true)
    {
        try
        {
            Memory<byte> memory = writer.GetMemory();  // Request memory from the pipe
            int bytesRead = await socket.ReceiveAsync(memory, SocketFlags.None);

            if (bytesRead == 0)
                break;

            writer.Advance(bytesRead); // Tell the PipeWriter how much was read from the Socket
        }
        catch
        {
            break;
        }

        FlushResult result = await writer.FlushAsync(); // Make the data available to the PipeReader

        if (result.IsCompleted)
            break;
    }

    writer.Complete(); // Signal to the reader that we're done writing
}
```

```
private static async Task ReadPipeAsync(PipeReader reader)
{
    while (true)
    {
        ReadResult result = await reader.ReadAsync(); // will await until the writer flushes
        ReadOnlySequence<byte> buffer = result.Buffer;
        SequencePosition? position = null;

        do
        {
            position = buffer.PositionOf((byte)'\n’); // Find the EOL

            if (position != null)
            {
                ProcessLine(buffer.Slice(0, position.Value));
                // Skip what we've already processed including \n
                buffer = buffer.Slice(buffer.GetPosition(1, position.Value));
            }
        } while (position != null);

        reader.AdvanceTo(buffer.Start, buffer.End); // Tell PipeReader how much we consumed

        if (result.IsCompleted) // Stop reading if there’s no more data coming
            break;
    }

    reader.Complete(); // Mark the PipeReader as complete
}
```

```csharp
private static async Task ReadPipeAsync(PipeReader reader)
{
    while (true)
    {
        ReadResult result = await reader.ReadAsync(); // will await until the writer flushes
        ReadOnlySequence<byte> buffer = result.Buffer;
        SequencePosition? position = null;

        do
        {
            position = buffer.PositionOf((byte)'\n’); // Find the EOL

            if (position != null)
            {
                ProcessLine(buffer.Slice(0, position.Value));
                // Skip what we've already processed including \n
                buffer = buffer.Slice(buffer.GetPosition(1, position.Value));
            }
        } while (position != null);

        reader.AdvanceTo(buffer.Start, buffer.End); // Tell PipeReader how much we consumed

        if (result.IsCompleted) // Stop reading if there’s no more data coming
            break;
    }

    reader.Complete(); // Mark the PipeReader as complete
}
```

```csharp
private static async Task ReadPipeAsync(PipeReader reader)
{
    while (true)
    {
        ReadResult result = await reader.ReadAsync(); // will await until the writer flushes
        ReadOnlySequence<byte> buffer = result.Buffer;
        SequencePosition? position = null;

        do
        {
            position = buffer.PositionOf((byte)'\n’); // Find the EOL

            if (position != null)
            {
                ProcessLine(buffer.Slice(0, position.Value));
                // Skip what we've already processed including \n
                buffer = buffer.Slice(buffer.GetPosition(1, position.Value));
            }
       } while (position != null);

       reader.AdvanceTo(buffer.Start, buffer.End); // Tell PipeReader how much we consumed

       if (result.IsCompleted) // Stop reading if there’s no more data coming
           break;
    }

    reader.Complete(); // Mark the PipeReader as complete
}
```

```csharp
private static async Task ReadPipeAsync(PipeReader reader)
{
    while (true)
    {
        ReadResult result = await reader.ReadAsync(); // will await until the writer flushes
        ReadOnlySequence<byte> buffer = result.Buffer;
        SequencePosition? position = null;

        do
        {
            position = buffer.PositionOf((byte)'\n’); // Find the EOL

            if (position != null)
            {
                ProcessLine(buffer.Slice(0, position.Value));
                // Skip what we've already processed including \n
                buffer = buffer.Slice(buffer.GetPosition(1, position.Value));
            }
        } while (position != null);

        reader.AdvanceTo(buffer.Start, buffer.End); // Tell PipeReader how much we consumed

        if (result.IsCompleted) // Stop reading if there’s no more data coming
            break;
    }

    reader.Complete(); // Mark the PipeReader as complete
}
```

```csharp
private static async Task ReadPipeAsync(PipeReader reader)
{
    while (true)
    {
        ReadResult result = await reader.ReadAsync(); // will await until the writer flushes
        ReadOnlySequence<byte> buffer = result.Buffer;
        SequencePosition? position = null;

        do
        {
            position = buffer.PositionOf((byte)'\n’); // Find the EOL

            if (position != null)
            {
                ProcessLine(buffer.Slice(0, position.Value));
                // Skip what we've already processed including \n
                buffer = buffer.Slice(buffer.GetPosition(1, position.Value));
            }
       } while (position != null);

       reader.AdvanceTo(buffer.Start, buffer.End); // Tell PipeReader how much we consumed

       if (result.IsCompleted) // Stop reading if there’s no more data coming
           break;
    }

    reader.Complete(); // Mark the PipeReader as complete
}
```

```
private static async Task ReadPipeAsync(PipeReader reader)
{
    while (true)
    {
        ReadResult result = await reader.ReadAsync(); // will await until the writer flushes
        ReadOnlySequence<byte> buffer = result.Buffer;
        SequencePosition? position = null;

        do
        {
            position = buffer.PositionOf((byte)'\n’); // Find the EOL

            if (position != null)
            {
                ProcessLine(buffer.Slice(0, position.Value));
                // Skip what we've already processed including \n
                buffer = buffer.Slice(buffer.GetPosition(1, position.Value));
            }
        } while (position != null);

        reader.AdvanceTo(buffer.Start, buffer.End); // Tell PipeReader how much we consumed

        if (result.IsCompleted) // Stop reading if there’s no more data coming
            break;
    }

    reader.Complete(); // Mark the PipeReader as complete
}
```

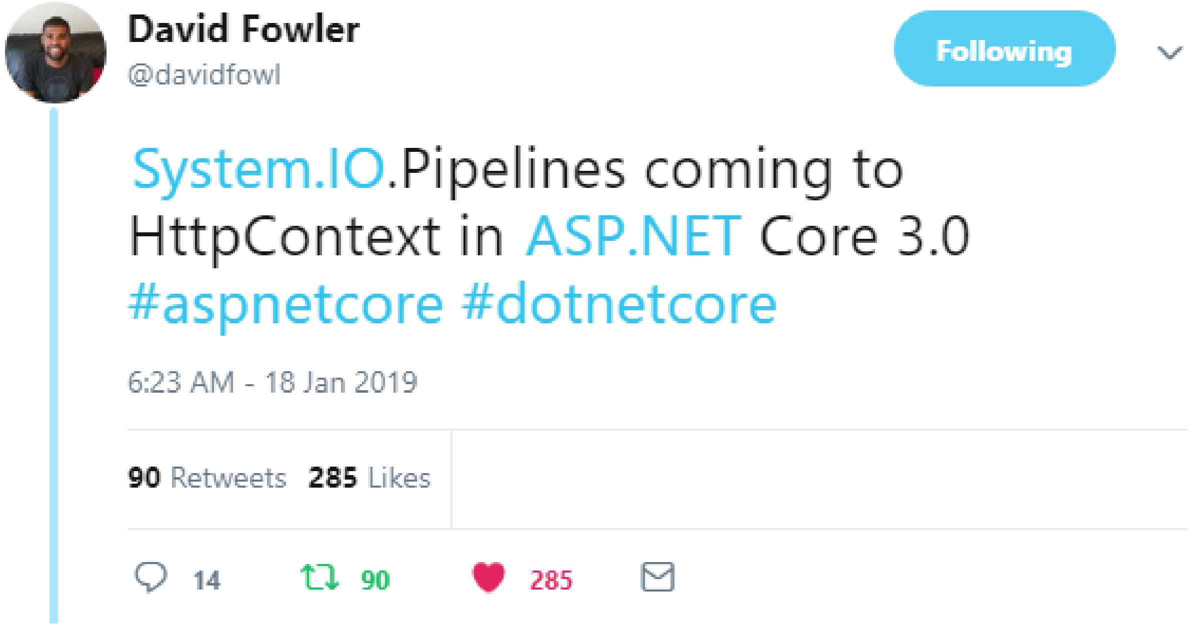
```
private static async Task ReadPipeAsync(PipeReader reader)
{
    while (true)
    {
        ReadResult result = await reader.ReadAsync(); // will await until the writer flushes
        ReadOnlySequence<byte> buffer = result.Buffer;
        SequencePosition? position = null;

        do
        {
            position = buffer.PositionOf((byte)'\n’); // Find the EOL

            if (position != null)
            {
                ProcessLine(buffer.Slice(0, position.Value));
                // Skip what we've already processed including \n
                buffer = buffer.Slice(buffer.GetPosition(1, position.Value));
            }
        } while (position != null);

        reader.AdvanceTo(buffer.Start, buffer.End); // Tell PipeReader how much we consumed

        if (result.IsCompleted) // Stop reading if there’s no more data coming
            break;
    }

    reader.Complete(); // Mark the PipeReader as complete
}
```

```csharp
private static async Task ReadPipeAsync(PipeReader reader)
{
    while (true)
    {
        ReadResult result = await reader.ReadAsync(); // will await until the writer flushes
        ReadOnlySequence<byte> buffer = result.Buffer;
        SequencePosition? position = null;

        do
        {
            position = buffer.PositionOf((byte)'\n’); // Find the EOL

            if (position != null)
            {
                ProcessLine(buffer.Slice(0, position.Value));
                // Skip what we've already processed including \n
                buffer = buffer.Slice(buffer.GetPosition(1, position.Value));
            }
       } while (position != null);

       reader.AdvanceTo(buffer.Start, buffer.End); // Tell PipeReader how much we consumed

       if (result.IsCompleted) // Stop reading if there’s no more data coming
           break;
    }

    reader.Complete(); // Mark the PipeReader as complete
}
```

Stream Connectors

- CoreFx APIs are being updated to expose pipes
- Connectors exists to transition from a stream to a pipe.
- PipeReader.Create(Stream, StreamPipeReaderOptions)
- PipeWriter.Create(Stream, StreamPipeWriterOptions)

Pipes and ASP.NET Core 3.0


David Fowler
@davidfowl

Following

System.IO.Pipelines coming to HttpContext in ASP.NET Core 3.0 #aspnetcore #dotnetcore

6:23 AM - 18 Jan 2019

90 Retweets **285** Likes

14 90 285

```csharp
public class Startup
{
    public void ConfigureServices(IServiceCollection services) {}

    public void Configure(IApplicationBuilder app)
    {
        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapPost("/", async context =>
            {
                while (true)
                {
                    var result = await context.Request.BodyReader.ReadAsync();
                    var buffer = result.Buffer;

                    // process the buffer

                    if (result.IsCompleted) break;

                    context.Request.BodyReader.AdvanceTo(buffer.End);
                }
            });
        });
    }
}
```

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services) {}

    public void Configure(IApplicationBuilder app)
    {
        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapPost("/", async context =>
            {
                while (true)
                {
                    var result = await context.Request.BodyReader.ReadAsync();
                    var buffer = result.Buffer;

                    // process the buffer

                    if (result.IsCompleted) break;

                    context.Request.BodyReader.AdvanceTo(buffer.End);
                }
            });
        });
    }
}
```

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services) {}

    public void Configure(IApplicationBuilder app)
    {
        app.UseRouting();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapPost("/", async context =>
            {
                while (true)
                {
                    var result = await context.Request.BodyReader.ReadAsync();
                    var buffer = result.Buffer;

                    // process the buffer

                    if (result.IsCompleted) break;

                    context.Request.BodyReader.AdvanceTo(buffer.End);
                }
            });
        });
    }
}
```

SequenceReader<T>

- Simplifies processing of ReadOnlySequence<T>
- Focuses on performance and minimal or zero heap allocs
- Unifies differences between single segment and multi segement sequences
- Read and advance through a sequence
- SequenceReader<T> is a ref struct and has same limitations as Span

SEQUENCE READER DEMO

Putting it into practice – Span<T> Parsing

Microservice which:

1. Retrieves S3 object (TSV file) from AWS
2. Decompresses file
3. Parses TSV to get 3 of 25 columns for each row
4. Indexes data to ElasticSearch

CloudFrontParser

TSV Parsing Optimisation - Results

Processing 75 files of 10,000 rows each

Method	Mean	Ratio	Gen 0	Gen 1	Gen 2	Allocated
Original	7,035.8 ms	1.00	1582000.0	242000.0	92000.0	1.46 KB
Optimised	923.1 ms	0.13	52000.0	23000.0	1000.0	1.84 KB

Over 7x Faster

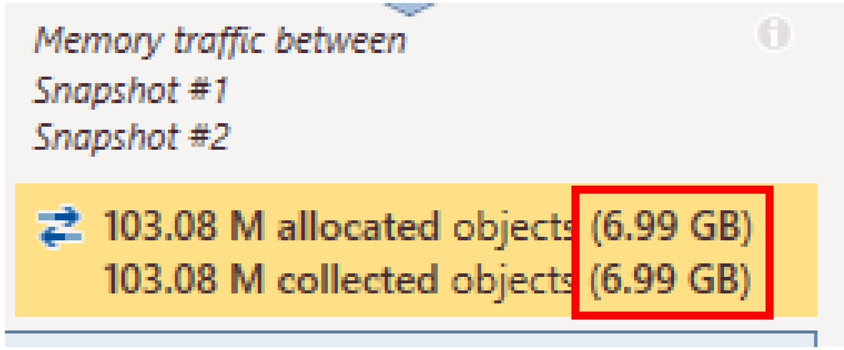
Allocations ??? ¯_(ツ)_/¯

DOTMEMORY DEMO

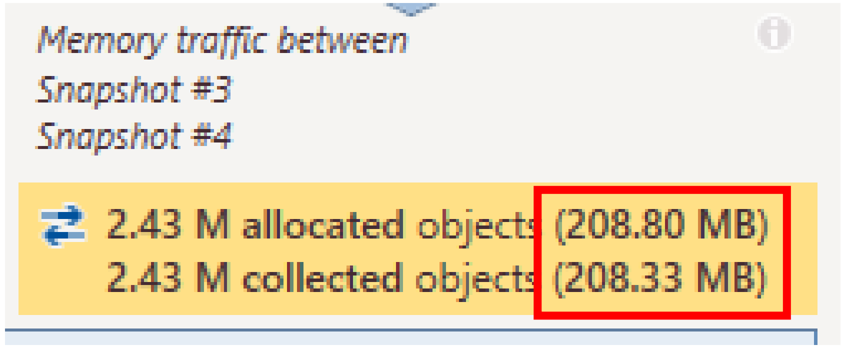
TSV Parsing Optimisation – dotMemory

Processing 75 files of 10,000 rows each

Original



Optimised



33.6x Less Heap Memory Allocated

NOTE: ~203.5Mb are the string allocations for the parsed data

Nightly of Benchmark .NET

Processing 75 files of 10,000 rows each

Method	Mean	Ratio	Gen 0	Gen 1	Gen 2	Allocated
Original	8,500.9 ms	1.00	1548000.0	267000.0	109000.0	7205.44 MB
Optimised	957.5 ms	0.11	43000.0	20000.0	2000.0	242.41 MB

WORKING WITH JSON

JSON APIs - .NET Core 3.0

- In the box JSON APIs - **System.Text.Json**
- Low-Level – **Utf8JsonReader** and **Utf8JsonWriter**
- Mid-Level – **JsonDocument**
- High-Level – **JsonSerializer** and **JsonDeserializer**
- Future – **JsonPipeWriter**, **JsonStreamWriter** etc.

SYSTEM.TEXT.JSON DEMOS

Method	Mean	Error	StdDev	Median	Gen 0	Gen 1	Gen 2	Allocated
HighLevel	865,114.5 ns	29,099.781 ns	85,801.360 ns	912,150.0 ns	6.8359	1.9531	-	31248 B
HighLevelEmpty	521,730.1 ns	18,397.991 ns	54,246.891 ns	554,526.6 ns	7.3242	3.4180	-	30540 B
MidLevel	3,192.9 ns	69.495 ns	74.359 ns	3,172.0 ns	0.1450	-	-	624 B
MidLevelEmpty	1,557.2 ns	20.582 ns	19.253 ns	1,558.3 ns	0.1068	-	-	448 B
LowLevel	1,667.3 ns	20.512 ns	18.183 ns	1,668.7 ns	0.1240	-	-	520 B
LowLevelEmpty	672.6 ns	6.150 ns	5.753 ns	672.5 ns	0.0820	-	-	344 B
WithoutPipe	1,192.4 ns	23.260 ns	33.359 ns	1,193.1 ns	0.0248	-	-	104 B
EmptyWithoutPipe	284.5 ns	5.683 ns	12.111 ns	284.2 ns	-	-	-	-

Method	Mean	Error	StdDev	Median	Gen 0	Gen 1	Gen 2	Allocated
HighLevel	865,114.5 ns	29,099.781 ns	85,801.360 ns	912,150.0 ns	6.8359	1.9531	-	31248 B
HighLevelEmpty	521,730.1 ns	18,397.991 ns	54,246.891 ns	554,526.6 ns	7.3242	3.4180	-	30540 B
MidLevel	3,192.9 ns	69.495 ns	74.359 ns	3,172.0 ns	0.1450	-	-	624 B
MidLevelEmpty	1,557.2 ns	20.582 ns	19.253 ns	1,558.3 ns	0.1068	-	-	448 B
LowLevel	1,667.3 ns	20.512 ns	18.183 ns	1,668.7 ns	0.1240	-	-	520 B
LowLevelEmpty	672.6 ns	6.150 ns	5.753 ns	672.5 ns	0.0820	-	-	344 B
WithoutPipe	1,192.4 ns	23.260 ns	33.359 ns	1,193.1 ns	0.0248	-	-	104 B
EmptyWithoutPipe	284.5 ns	5.683 ns	12.111 ns	284.2 ns	-	-	-	-

Compared to High-Level

With user: 271x faster with 98% fewer allocations

Empty: 334x faster with 98.5% fewer allocations

Method	Mean	Error	StdDev	Median	Gen 0	Gen 1	Gen 2	Allocated
HighLevel	865,114.5 ns	29,099.781 ns	85,801.360 ns	912,150.0 ns	6.8359	1.9531	-	31248 B
HighLevelEmpty	521,730.1 ns	18,397.991 ns	54,246.891 ns	554,526.6 ns	7.3242	3.4180	-	30540 B
MidLevel	3,192.9 ns	69.495 ns	74.359 ns	3,172.0 ns	0.1450	-	-	624 B
MidLevelEmpty	1,557.2 ns	20.582 ns	19.253 ns	1,558.3 ns	0.1068	-	-	448 B
LowLevel	1,667.3 ns	20.512 ns	18.183 ns	1,668.7 ns	0.1240	-	-	520 B
LowLevelEmpty	672.6 ns	6.150 ns	5.753 ns	672.5 ns	0.0820	-	-	344 B
WithoutPipe	1,192.4 ns	23.260 ns	33.359 ns	1,193.1 ns	0.0248	-	-	104 B
EmptyWithoutPipe	284.5 ns	5.683 ns	12.111 ns	284.2 ns	-	-	-	-

Compared to Mid-Level

With user: 1.9x faster with 28% fewer allocations

Empty: 2.3x faster with 23% fewer allocations

Method	Mean	Error	StdDev	Median	Gen 0	Gen 1	Gen 2	Allocated
HighLevel	865,114.5 ns	29,099.781 ns	85,801.360 ns	912,150.0 ns	6.8359	1.9531	-	31248 B
HighLevelEmpty	521,730.1 ns	18,397.991 ns	54,246.891 ns	554,526.6 ns	7.3242	3.4180	-	30540 B
MidLevel	3,192.9 ns	69.495 ns	74.359 ns	3,172.0 ns	0.1450	-	-	624 B
MidLevelEmpty	1,557.2 ns	20.582 ns	19.253 ns	1,558.3 ns	0.1068	-	-	448 B
LowLevel	1,667.3 ns	20.512 ns	18.183 ns	1,668.7 ns	0.1240	-	-	520 B
LowLevelEmpty	672.6 ns	6.150 ns	5.753 ns	672.5 ns	0.0820	-	-	344 B
WithoutPipe	1,192.4 ns	23.260 ns	33.359 ns	1,193.1 ns	0.0248	-	-	104 B
EmptyWithoutPipe	284.5 ns	5.683 ns	12.111 ns	284.2 ns	-	-	-	-

Compared to Low-Level with PipeReader

With user: 1.4x faster with 80% fewer allocations

Empty: 2.4x faster with 100% fewer allocations

ACTIVITY 06
JSON

Putting it into practice – Parsing JSON

Microservice which:

1. Perform ElasticSearch Bulk Index
2. Deserialise JSON response to check for errors
3. Return a list of the IDs which errored

WARNING: APIs have changed a little in latest previews!

BulkResponseParser

JSON API vs JSON.NET - Results

Processing Failure Response

Method	Mean	Ratio	Gen 0	Gen 1	Gen 2	Allocated
Original	428,500 ns	1.00	27.3428	0.4883	-	114.30 KB
Optimised	141,900 ns	0.33	3.6621	0.2441	-	15.77 KB

Processing Successful Response

Method	Mean	Ratio	Gen 0	Gen 1	Gen 2	Allocated
Original	386,514.8 ns	1.000	26.3672	0.4883	-	111408 B
Optimised	485.3 ns	0.001	0.0181	0.0010	-	80 B

OTHER QUICK PEEKS

ValueTask<T>

- Provides a value type that wraps a Task<TResult> and a TResult, only one of which is used
- Reduces Task allocations for methods that generally are expected to complete synchronously
- A ValueTask<TResult> instance may only be awaited once

https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.valuetask-1

System.Threading.Channels

- Provides a set of synchronization data structures for passing data between producers and consumers asynchronously
- Optimised for performance and reduced locks
- Supports various producer/consumer patterns

CHANNELS DEMO

FUTURE APIs TO WATCH

UTF8String

- Today, strings are UTF16
- UTF16 requires a minimum of 2 bytes per character
- The web works with UTF8 data
- UTF8 requires a minimum of 1 byte per character
- UTF8String proposes a type which is more efficient for web scenarios

https://github.com/dotnet/corefxlab/issues/2350

NEXT STEPS

Business Buy-In

- Identify a quick win
- Prototype some optimisations
- Use a scientific approach to measure gains
- Convert those gains into a monetary value
- Determine the cost to benefit ratio

Cost Saving Example: Input Processor

This work is a small part of a much bigger potential gain

For a single microservice handling
18 million messages per day

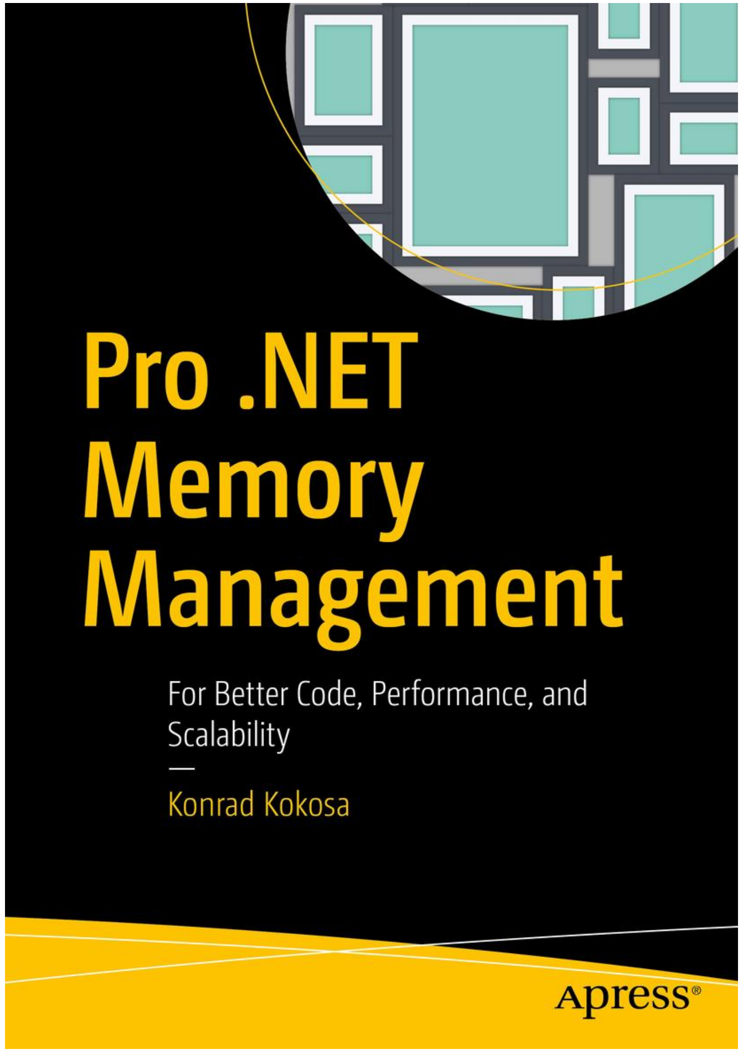
Reduction of at least 50% of allocations.
Potential to at least double per instance throughput
At least 1 less VM needed per year saving $1,700

Summary

- Measure, don't assume!
- Be scientific; make small changes each time and measure again
- Focus on hot paths
- Don't copy memory, slice it! Span<T> is less complex than it may first seem.
- Use ArrayPools where appropriate to reduce array allocations
- Consider Pipelines for I/O scenarios
- Consider new Utf8Json APIs for high-performance JSON parsing

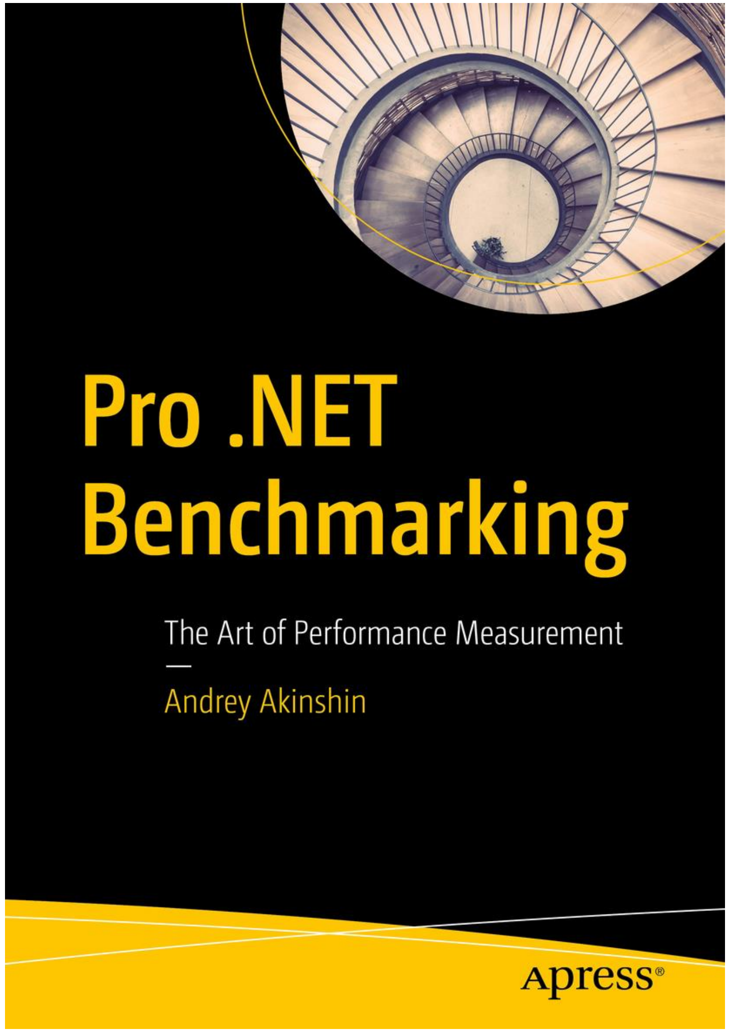
Pro .NET Memory Management

By Konrad Kokosa



Pro .NET Benchmarking

By Andrey Akinshin



Thanks for listening!

@stevejgordon

www.stevejgordon.co.uk

http://bit.ly/TurbochargedWorkshop

MVP
Microsoft®
Most Valuable
Professional

.NET
southeast

https://www.meetup.com/dotnetsoutheast

madgex