GOPAS

# TechEd

**23. - 25. 5. 2023** | PRAHA / ONLINE

Největší československá
**konference o IT**

GOLD PARTNER

<epam>

# 10 (and more) reasons to migrate from .NET Framework to .NET

Ákos Nagy | **Teched 2023**

GOLD PARTNER

Tech·Ed GOPAS ‹epam›

# Why?

"New is always better"

-Barney Stinson

# Why?

- The usual PR reasons
  - “Modular”
  - “Cross-platform”
  - “More performant”

- The “cop-out” reasons
  - New C# language features (C# 8 and above)
  - Re-imagined frameworks(ASP.NET Core, EF Core)
    - “Modular, cross-platform, more performant” ☺
  - New frameworks
    - Blazor, MAUI

# Why?

- Let’s look at concrete examples
  - Language features with high impact that go beyond the “reduce boilerplate” type of compiler smoke-and-mirrors
  - APIs that have been rewritten and improved to be more performant

- Note that these are just some personal examples
  - With every C# release, you can check out https://devblogs.microsoft.com and https://microsoft.com for a comprehensive list of new features
  - With every .NET release, you can check out https://devblogs.microsoft.com for a comprehensive list of performance related improvements (look for Stephen Toub)

# Disclaimer

# Reason 1 – Async streams

- Async streams are a C# 8 language feature (supported by new BCL interfaces)
- Allows to combine async methods and iterator methods
  - Async methods traditionally had to have a return type of `Task` or `Task<T>` (or void)
  - This was later expanded to “task-like types” (mostly to support `ValueTask`)
  - This made it possible to introduce new types for async method return types

- Methods can return `IAsyncEnumerable<T>`
  - Mirrors `IEnumerable<T>`
  - Methods can be marked async, so they can use await
  - Can be used with await foreach language keyword

- Typical use-case
  - Paged/batched/segmented I/O operations
    - I/O operations are always asynchronous
    - The paged/batched/segmented nature makes them good
    - candidates for iterators

# Reason 2 – Async dispose

- Async disposable are a new C# language feature (supported by new BCL interfaces)
- Classes can implement `IAsyncDisposable` that have a `ValueTask DiposeAsync()` method
  - Mirrors `IDisposable`
  - Methods can be marked async, so they can use await
  - Can be used with await using language keyword

- Typical use-case
  - Disposing I/O resources
    - This usually involves an I/O operation, so must be done asynchronously
    - Awaiting tasks (cancelled or not) started by a resource before disposing of it

# Reason 3 – Activator.CreateInstance<T>()

- `Activator.CreateInstance<T>()` was one of the APIs that have been greatly improved performance-wise
- Why should I care?
  - `Activator.CreateInstance<T>()` is the API that supports the new() generic constraint

# Reason 3 – Activator.CreateInstance<T>()

```
BenchmarkDotNet=v0.13.5, OS=Windows 11 (10.0.22621.1702/22H2/2022Update/SunValley2)
12th Gen Intel Core i7-1255U, 1 CPU, 12 logical and 10 physical cores
.NET SDK=7.0.203
  [Host]     : .NET 7.0.5 (7.0.523.17405), X64 RyuJIT AVX2
  Job-OXMGFL : .NET 7.0.5 (7.0.523.17405), X64 RyuJIT AVX2
  Job-DBOEAU : .NET Framework 4.8.1 (4.8.9139.0), X64 RyuJIT VectorSize=256
```

| Method | Runtime | Mean | Error | StdDev | Ratio | Gen0 | Allocated | Alloc Ratio |
|---|---|---|---|---|---|---|---|---|
| CreateNode | .NET 7.0 | 8.286 ns | 0.1879 ns | 0.1665 ns | 0.31 | 0.0038 | 24 B | 1.00 |
| CreateNode | .NET Framework 4.8 | 26.883 ns | 0.2934 ns | 0.2745 ns | 1.00 | 0.0038 | 24 B | 1.00 |

# Reason 4 – Attribute reflections

- Resolving attributes have been improved so they are again much faster

```
BenchmarkDotNet=v0.13.5, OS=Windows 11 (10.0.22621.1702/22H2/2022Update/SunValley2)
12th Gen Intel Core i7-1255U, 1 CPU, 12 logical and 10 physical cores
.NET SDK=7.0.203
  [Host]     : .NET 7.0.5 (7.0.523.17405), X64 RyuJIT AVX2
  Job-OXMGFL : .NET 7.0.5 (7.0.523.17405), X64 RyuJIT AVX2
  Job-DBOEAU : .NET Framework 4.8.1 (4.8.9139.0), X64 RyuJIT VectorSize=256
```

| Method | Runtime | Mean | Error | StdDev | Ratio | Gen0 | Allocated | Alloc Ratio |
|---|---|---|---|---|---|---|---|---|
| GetCustomAttributes | .NET 7.0 | 513.2 ns | 8.16 ns | 7.63 ns | 0.32 | 0.0391 | 248 B | 0.66 |
| GetCustomAttributes | .NET Framework 4.8 | 1,606.1 ns | 24.73 ns | 23.14 ns | 1.00 | 0.0591 | 377 B | 1.00 |

# Reason 5 – Cryptography performance

- The performance of `CryptoStream`, the core stream supporting encryption/decryption in .NET (Framework) have been greatly improved

```
BenchmarkDotNet=v0.13.5, OS=Windows 11 (10.0.22621.1702/22H2/2022Update/SunValley2)
12th Gen Intel Core i7-1255U, 1 CPU, 12 logical and 10 physical cores
.NET SDK=7.0.203
  [Host]     : .NET 7.0.5 (7.0.523.17405), X64 RyuJIT AVX2
  Job-OXMGFL : .NET 7.0.5 (7.0.523.17405), X64 RyuJIT AVX2
  Job-DBOEAU : .NET Framework 4.8.1 (4.8.9139.0), X64 RyuJIT VectorSize=256
```

| Method | Runtime | Mean | Error | StdDev | Ratio | Gen0 | Allocated | Alloc Ratio |
|---|---|---|---|---|---|---|---|---|
| EncodeBase64 | .NET 7.0 | 8.568 µs | 0.1671 µs | 0.1788 µs | 0.004 | 0.0458 | 296 B | 0.000 |
| EncodeBase64 | .NET Framework 4.8 | 2,153.219 µs | 27.5961 µs | 25.8134 µs | 1.000 | 339.8438 | 2145043 B | 1.000 |

# Reason 6 and 7 – System.Int32 handling

- `Int32.Parse(string)` is much faster

| Method | Runtime | Mean | Error | StdDev | Ratio | Allocated | Alloc Ratio |
|---|---|---|---|---|---|---|---|
| ParseInt | .NET 7.0 | 11.00 ns | 0.174 ns | 0.162 ns | 0.19 | - | NA |
| ParseInt | .NET Framework 4.8 | 59.22 ns | 1.069 ns | 1.000 ns | 1.00 | - | NA |

- `Int32.ToString()` is much faster

| Method | Runtime | Mean | Error | StdDev | Ratio | Gen0 | Allocated | Alloc Ratio |
|---|---|---|---|---|---|---|---|---|
| FormatInt | .NET 7.0 | 10.59 ns | 0.181 ns | 0.169 ns | 0.33 | 0.0064 | 40 B | 0.83 |
| FormatInt | .NET Framework 4.8 | 31.69 ns | 0.632 ns | 0.591 ns | 1.00 | 0.0076 | 48 B | 1.00 |

# Reason 8 – LInQ API performance

| Method | Runtime | Mean | Error | StdDev | Ratio | RatioSD | Gen0 | Gen1 | Allocated | Alloc Ratio |
|---|---|---|---|---|---|---|---|---|---|---|
| Min | .NET 7.0 | 643.7 ns | 11.02 ns | 9.77 ns | 0.02 | 0.00 | - | - | - | 0.00 |
| Min | .NET Framework 4.8 | 30,255.7 ns | 322.65 ns | 301.81 ns | 1.00 | 0.00 | - | - | 32 B | 1.00 |
| Max | .NET 7.0 | 649.2 ns | 12.73 ns | 11.29 ns | 0.02 | 0.00 | - | - | - | 0.00 |
| Max | .NET Framework 4.8 | 34,475.2 ns | 513.53 ns | 455.23 ns | 1.00 | 0.00 | - | - | 32 B | 1.00 |
| OrderBy | .NET 7.0 | 1,036,173.8 ns | 13,903.99 ns | 12,325.53 ns | 0.86 | 0.02 | 17.5781 | 1.9531 | 120313 B | 1.00 |
| OrderBy | .NET Framework 4.8 | 1,210,533.2 ns | 23,544.74 ns | 22,023.77 ns | 1.00 | 0.00 | 17.5781 | 1.9531 | 120448 B | 1.00 |

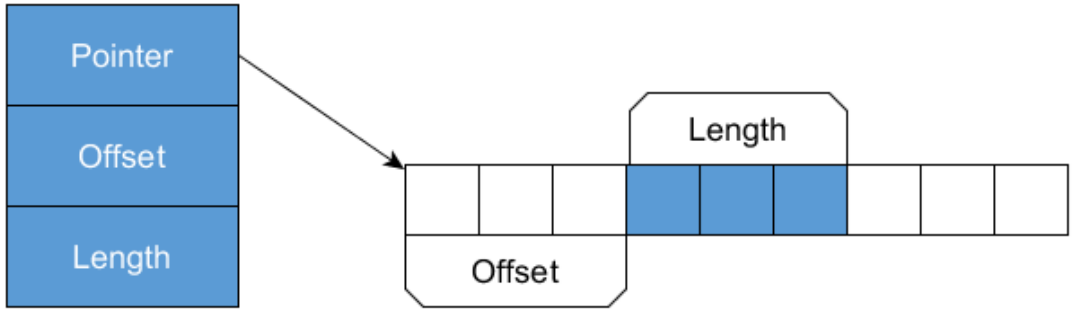
- Some LInQ operations are faster
  - `Max(), Min()`
  - `OrderBy()`

# Reason 9 – Span<T>

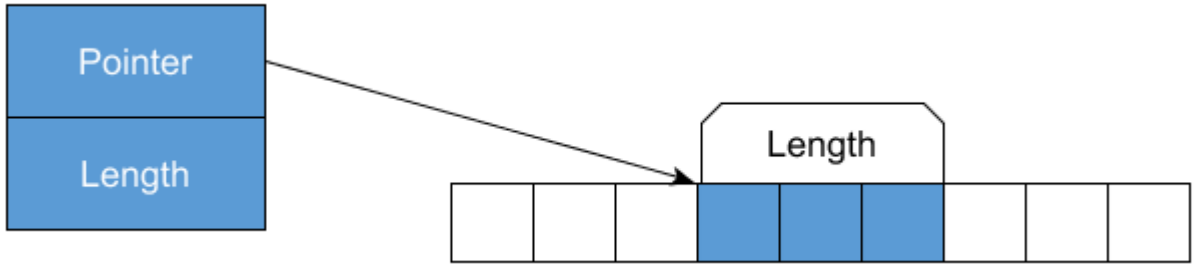
- `Span<T>` is a new data structure that was introduced to represent a continuous segment of allocated memory
  - Strings
  - Arrays
- Whenever strings are manipulated or a substring is referenced for some operation, a new string must be allocated (same for arrays)
  - i.e. every substring or subarray is a new memorysegment
- `Span<T>` aims to solve this problem

# Reason 9 – Span<T>

- Span<T> for .NET Framework is available as a NuGet package




- Span<T> is part of the .NET type system
- The .NET GC knows span and can update pointers when doing the “Compact” phase “natively”




https://adamsitnik.com/Span/

# Reason 9 – Span<T>

```
BenchmarkDotNet=v0.13.5, OS=Windows 11 (10.0.22621.1702/22H2/2022Update/SunValley2)
12th Gen Intel Core i7-1255U, 1 CPU, 12 logical and 10 physical cores
.NET SDK=7.0.203
  [Host]     : .NET 7.0.5 (7.0.523.17405), X64 RyuJIT AVX2
  Job-OXMGFL : .NET 7.0.5 (7.0.523.17405), X64 RyuJIT AVX2
  Job-DBOEAU : .NET Framework 4.8.1 (4.8.9139.0), X64 RyuJIT VectorSize=256
```

| Method | Runtime | Mean | Error | StdDev | Ratio | Allocated | Alloc Ratio |
|---|---|---|---|---|---|---|---|
| Slice | .NET 7.0 | 2.864 µs | 0.0462 µs | 0.0432 µs | 0.57 | - | NA |
| Slice | .NET Framework 4.8 | 5.020 µs | 0.0907 µs | 0.0849 µs | 1.00 | - | NA |

# Reason 10 (1x) - Improved JSON parsing

- With the emergence of HTTP based APIs, a big emphasis was put on this ecosystem
  - JSON serialization and parsing
  - Serialization is traditionally a complex problem
    - .NET supports JSON serialization with source generators (reason ☺)
    - Also serialization is now based on spans instead of strings (reason ☺)
  - HTTP communication is usually UTF-8
  - .NET strings are UTF-16
    - New APIs support UTF-8 based searching (reason ☺ )
    - C# 11 also supports UTF-8 strings

# Reason 10 (1x) - Improved JSON parsing

```
BenchmarkDotNet=v0.13.5, OS=Windows 11 (10.0.22621.1702/22H2/2022Update/SunValley2)
12th Gen Intel Core i7-1255U, 1 CPU, 12 logical and 10 physical cores
.NET SDK=7.0.203
  [Host]     : .NET 7.0.5 (7.0.523.17405), X64 RyuJIT AVX2
  DefaultJob : .NET 7.0.5 (7.0.523.17405), X64 RyuJIT AVX2
```

| Method | Mean | Error | StdDev | Ratio | RatioSD | Gen0 | Allocated | Alloc Ratio |
|---|---|---|---|---|---|---|---|---|
| RegularStringbenchmark | 872.5 ns | 13.72 ns | 12.16 ns | 1.00 | 0.00 | 0.0191 | 120 B | 1.00 |
| EncodedStringbenchmark | 818.0 ns | 16.18 ns | 15.13 ns | 0.94 | 0.02 | 0.0134 | 88 B | 0.73 |
| NativeUtf8Stringbenchmark | 819.4 ns | 9.30 ns | 8.70 ns | 0.94 | 0.01 | 0.0134 | 88 B | 0.73 |

# Reason 11 – ArrayPool<T>

- A pool of pre-allocated arrays that are managed by the framework, so they can be reused
- Less allocations and less cleanup means that less resources (time) is spent on memory management (same idea for the `Span<T>` time)

# Reason 11 – ArrayPool<T>

```
BenchmarkDotNet=v0.13.5, OS=Windows 11 (10.0.22621.1702/22H2/2022Update/SunValley2)
12th Gen Intel Core i7-1255U, 1 CPU, 12 logical and 10 physical cores
.NET SDK=7.0.203
  [Host]     : .NET 7.0.5 (7.0.523.17405), X64 RyuJIT AVX2
  DefaultJob : .NET 7.0.5 (7.0.523.17405), X64 RyuJIT AVX2
```

| Method | Mean | Error | StdDev | Ratio | Gen0 | Allocated | Alloc Ratio |
|---|---|---|---|---|---|---|---|
| UseArrays | 1,939.6 ns | 21.81 ns | 18.21 ns | 1.00 | 6.3286 | 40024 B | 1.00 |
| UseArrayPool | 697.0 ns | 13.47 ns | 13.23 ns | 0.36 | - | - | 0.00 |

# Reason 12 - RecyclableMemoryStream

- A pool of pre-allocated arrays that are managed by the framework, so they can be reused
- Same benefits as the `ArrayPool<T>`

# Reason 12 - RecyclableMemoryStream

```
BenchmarkDotNet=v0.13.5, OS=Windows 11 (10.0.22621.1702/22H2/2022Update/SunValley2)
12th Gen Intel Core i7-1255U, 1 CPU, 12 logical and 10 physical cores
.NET SDK=7.0.203
  [Host]     : .NET 7.0.5 (7.0.523.17405), X64 RyuJIT AVX2
  DefaultJob : .NET 7.0.5 (7.0.523.17405), X64 RyuJIT AVX2
```

| Method | Mean | Error | StdDev | Ratio | Gen0 | Gen1 | Gen2 | Allocated | Alloc Ratio |
|---|---|---|---|---|---|---|---|---|---|
| SerializerUsingMemoryStream | 127.03 ms | 1.571 ms | 1.312 ms | 1.00 | 24750.0000 | 24750.0000 | 24750.0000 | 90903.97 KB | 1.000 |
| SerializerUsingRecyclableMemoryStream | 87.34 ms | 1.378 ms | 1.222 ms | 0.69 | - | - | - | 83.86 KB | 0.001 |

# Thank you for your attention!

- Source code available at: https://github.com/conwid/dotnetreasons
- Read more from me at: https://dotnetfalcon.com
- Available for hire (consultation, training): https://hireme.dotnetfalcon.com
- Keep in touch: https://linkedin.com/in/azakosnagy

GOPAS

# Tech Ed

23. - 25. 5. 2023 | PRAHA / ONLINE