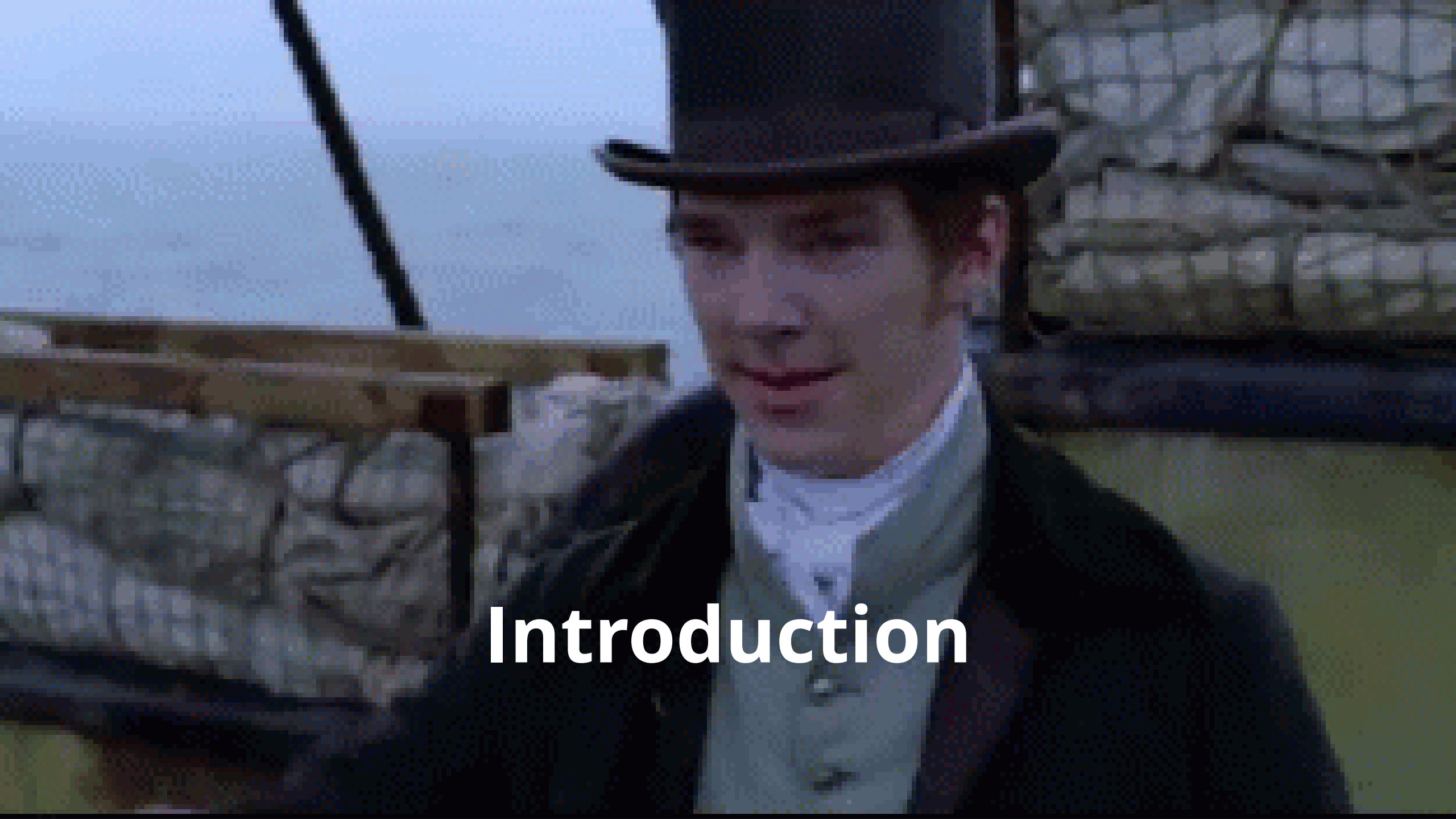
.NET 6

Rainer Stropek | @rstropek

.NET 6 Status

- LTS version
  - .NET 5 was not
- From now on 🔗
  - Every November a major version
  - Every second major version will have LTS



Introduction

Rainer Stropek

- Passionate software developers for 25+ years
- Microsoft MVP, Regional Director
- Trainer, Teacher, Mentor
- 💕 community

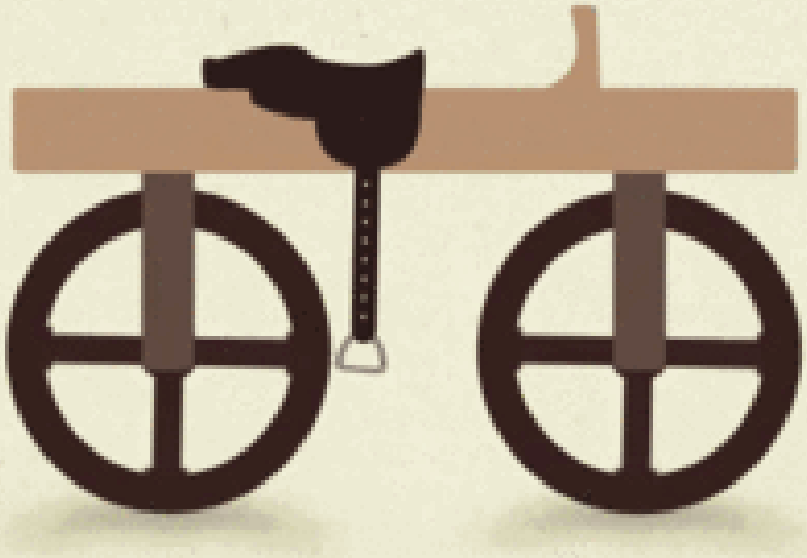
CODING CLUB LINZ

CoderDojo LINZ

https://rainerstropek.me

One .NET

A little bit of history...

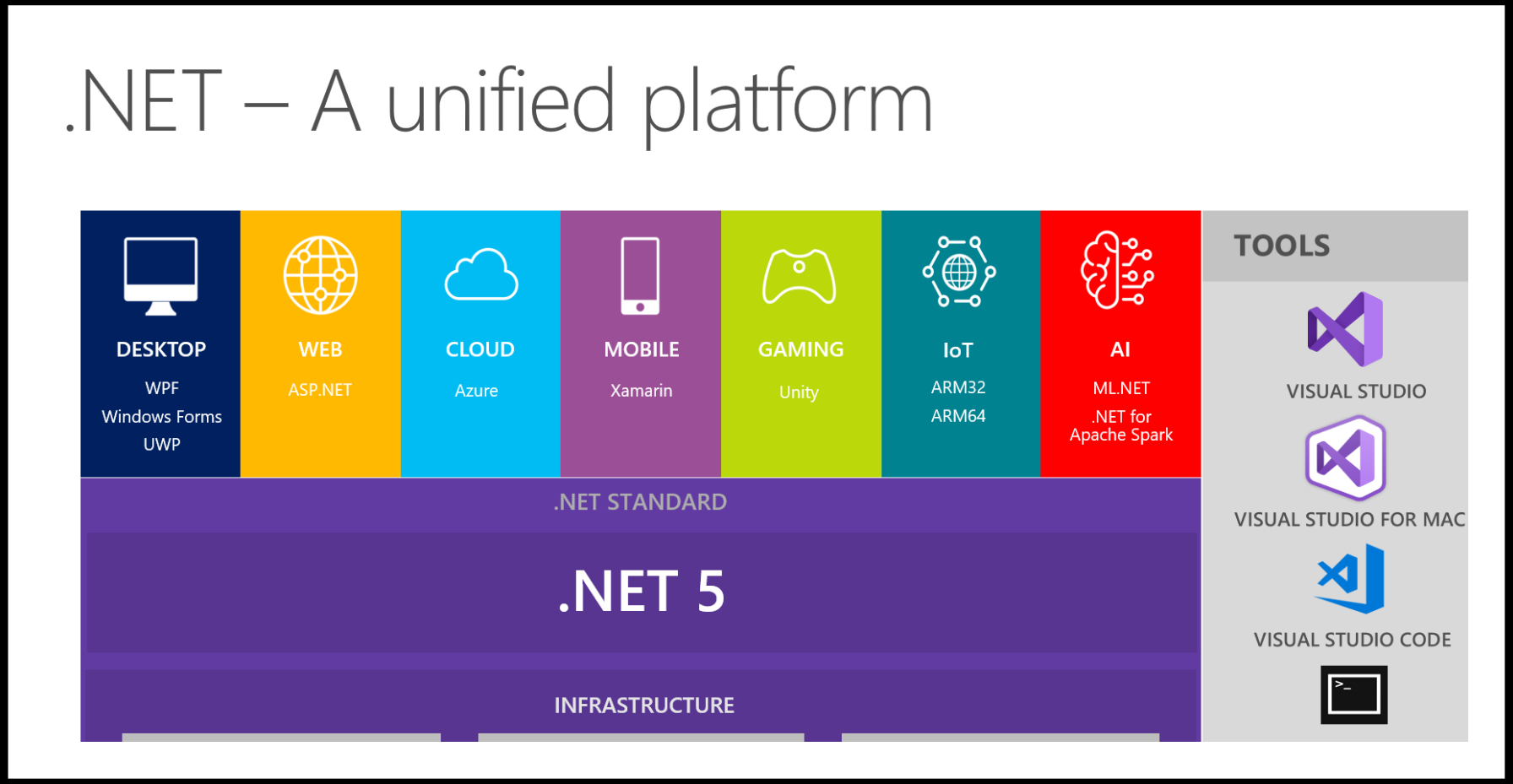


A little bit of history…

- .NET Framework at its beginnings
  - Windows only
  - Closed source
- Mono appeared
  - Separate implementation
  - Open-source, cross platform
- .NET Core
  - Lean, modern .NET
  - Open-source, cross platform
  - Grew bigger over time
- Back to one .NET

.NET Standard

.NET Unification Vision



.NET 6

- Plattform support 🔗
  - Nearly the same as .NET 5
  - New: macOS Arm64
  - No big migration hurdles to be expected
- LTS
  - Patches for three years
- No new features/APIs to .NET Framework and .NET Core
  - No need to run away
  - Have a plan for switching to the "new world"

Modular SDK

- .NET Core: Monolithik SDK
- .NET >= 5: Modular SDK
- .NET 6: *dotnet workload* extended

```
# list workloads available to install
dotnet workload search

# installs a workload
dotnet workload install

# lists installed workloads
dotnet workload list

# re-install all workloads you've previously installed
dotnet workload repair

# update workload
dotnet workload update

# remove the specified workload if you no longer it
dotnet workload uninstall
```

More about workloads

Crossgen2

aka *ReadyToRun* format

ReadyToRun File Format

- Goal: Binary file format for better startup performance
  - Form of *ahead of time* compilation (AOT)
  - Not full native AOT
  - "Run anywhere" code ranging from full JIT to full AOT (flexible)
  - Re-JITted after start for better steady-state performance
- Drawbacks
  - Larger file sizes
  - Specific for target architecture (e.g. Win x64)
- Publish as ReadyToRun 🔗
  - In .NET 6: *Crossgen2* replaces *Crossgen*
  - CLI (*dotnet publish ... -p:PublishReadyToRun=true*)
  - .csproj (*<PublishReadyToRun>true</PublishReadyToRun>*)

Conversation about ReadyToRun

Demo Time!

Source on GitHub

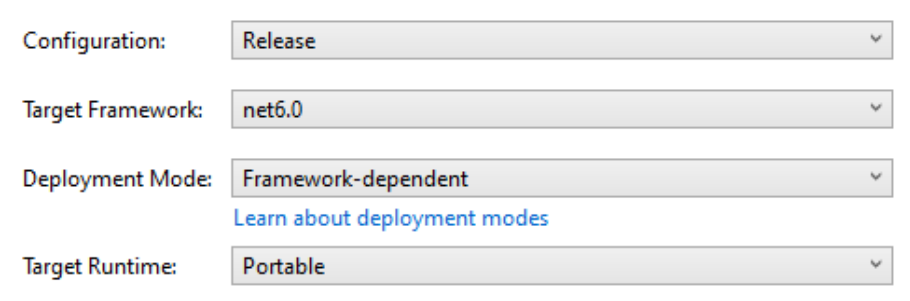
Crossgen2 Demo

Measure startup time of ASP.NET Core Web API with EF Core (in-memory provider) with and without *ReadyToRun*.

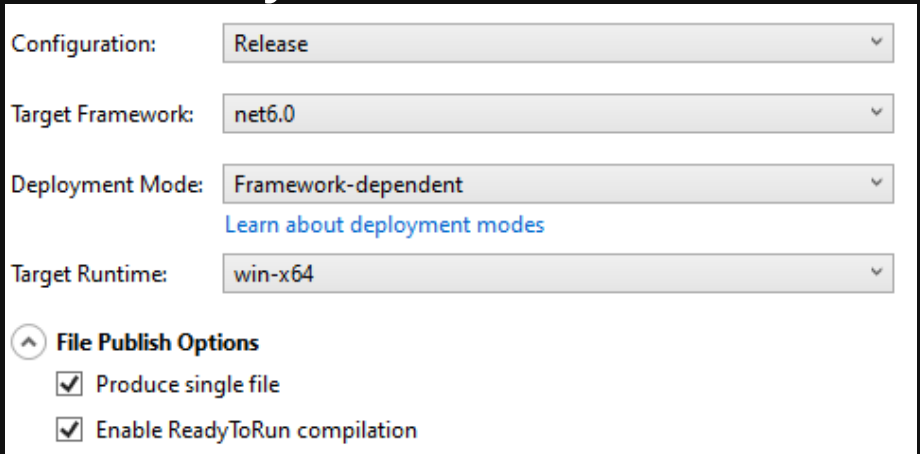
Much better startup time, more than double the file size

Category	Testrun	Process start	Configure starting	Configure finished	Startup	
Without	1	2.189,80	2.360,16	2.907,99	547,83	
Without	2	6.496,17	6.636,69	7.195,37	558,68	
Without	3	10.102,46	10.238,99	10.784,11	545,12	
				Average	550,54	
With	1	6.504,70	6.681,46	6.971,09	289,63	
With	2	9.215,79	9.374,64	9.661,51	286,87	
With	3	11.790,80	11.955,95	12.242,74	286,78	
				Average	287,76	48%

Without ReadyToRun:



With ReadyToRun:



Excursus: Measure startup time improvement with PerfView

Tip: could be analyzed with Benchmark.NET, too

Sync-over-async Improvements

Sync-over-async

- Pattern: Use async only in library, sync in API surface
  - Frequently found in legacy apps (e.g. modern lib, rather old UI)

```
// Note: Sync API, no async/await, no Task
static int DoSomethingSlow()
{
    // Note async in the background
    var result = Task.Run(async () =>
    {
        // Simulate work that takes 2 seconds (e.g. DB or net access)
        await Task.Delay(TimeSpan.FromSeconds(2));
        return 42;
    }).Result; // Note the blocking access to the Result property here

        return result;
}
```

More about sync-over-async

Sync-over-async Improvements

- Thread injection happens faster
  - If thread pool is busy just waiting for sync-over-async
- Consequence: Better performance
- Drawbacks
  - Larger thread pool
  - Uses more memory
- So?
  - Prefer modern async/await/Task programming model
  - Less problems when modernizing legacy apps

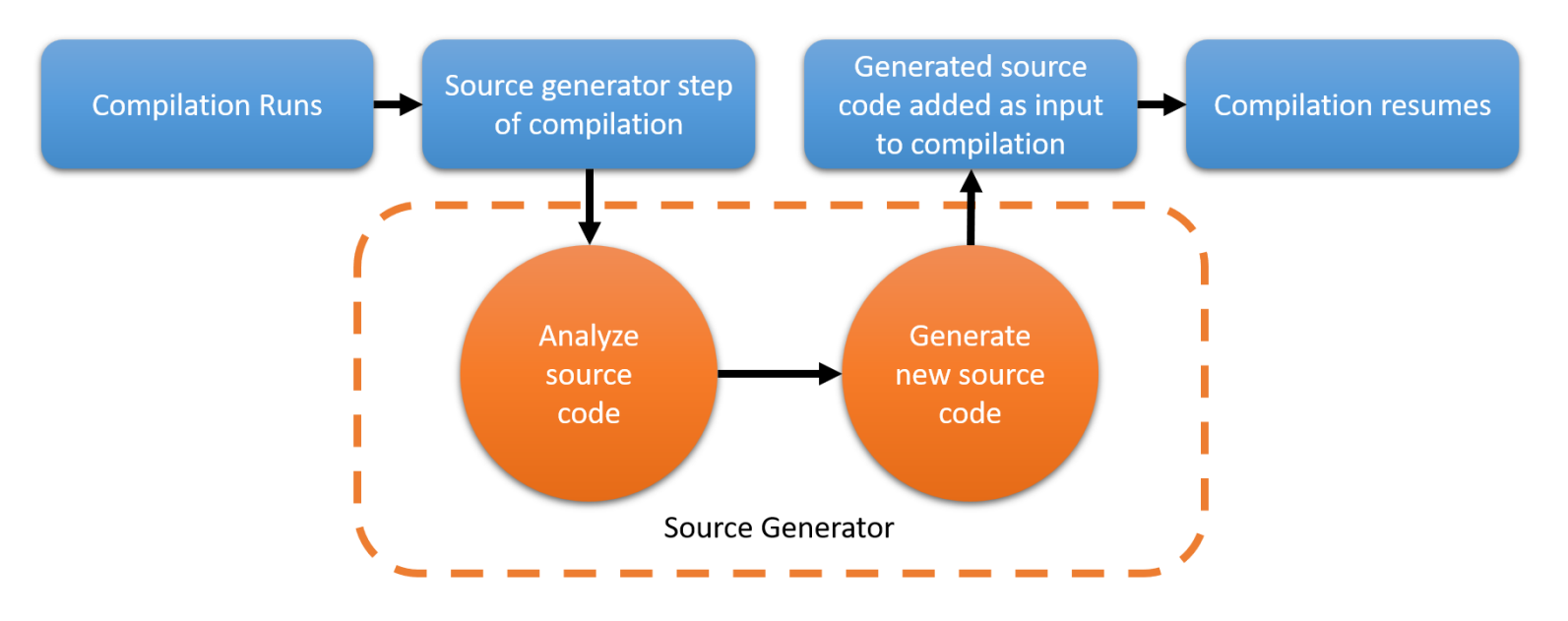
Demo Time!

Source on GitHub

JSON Serialization Code Generator

What's a Code Generator?

- Runs during compilation
- Inspects your program (using Roslyn)
- Produces additional files that are compiled together with your app



Code Generator Documentation

Why Code Generators?

- Currently:
  - Runtime reflection, e.g. ASP.NET Core DI (impacts performance)
  - Change IL after compilation ("IL weaving")
  - MSBuild tasks
- In the future:
  - Analyse code at compile time and generate code
    - Similar to Google's *Wire* DI framework (Go)
    - .NET examples: JSON Serialization (new in .NET 6), Logging Source Generation (net in .NET 6), ASP.NET Core Razor (new in .NET 6), MvvmGen (community)
  - Less reflection leads to...
    - ...better performance
    - ...smaller apps because AoT compiler (linker) can remove unused parts of your code

Demo Time!

Source on GitHub

Cold Start Run Strategy

Method	Mean	Error	StdDev
DeserializeJsonBlob	3.875 ms	0.8436 ms	2.487 ms
DeserializeJsonBlobGenerated	3.748 ms	0.6969 ms	2.055 ms

~3.3% improvement

Execution Without Cold Starts

Method	Mean	Error	StdDev
DeserializeJsonBlob	2.694 ms	0.0280 ms	0.0185 ms
DeserializeJsonBlobGenerated	2.773 ms	0.0193 ms	0.0128 ms

No improvement

Source Generators Behind the Scenes

- Generator
  - Goal: Learn how a source generator works
    - Build it
    - Running it in compiler and VS2022
  - Offer *Generate* attribute for partial methods
  - Generates a configurable amount of dummy code
    - No meaning, just generating lots of meaningless C#
    - Kind of *c-sharp-ipsum* 😉
- Generate code in a library
  - Makes lib pretty large
- Generate console app using the library
  - Goal: Learn how assembly trimming works
  - Build self-contained without/with assembly trimming
  - Try dynamic calls with assembly trimming

Source Generators Behind the Scenes

```
1  [Generator]
2  public class DummyGenerator : ISourceGenerator
3  {
4      /// <summary>
5      /// Visitor class that finds methods to generate
6      /// </summary>
7      class SyntaxReceiver : ISyntaxContextReceiver
8      {
9          ...
10     }
11
12     public void Initialize(GeneratorInitializationContext context)
13     {
14         // Register the attribute source
15         context.RegisterForPostInitialization((i) => i.AddSource("GenerateAttribute", attributeText));
16
17         // Register a syntax receiver that will be created for each generation pass
18         context.RegisterForSyntaxNotifications(() => new SyntaxReceiver());
19     }
20
21     public void Execute(GeneratorExecutionContext context)
22     {
23         ...
24         // Create string builder for source generation
25         var sourceBuilder = new StringBuilder();
26
27                 ... // Generate code
28
29         // Inject the created source into the users compilation
30         context.AddSource("dummy_generated.cs", SourceText.From(sourceBuilder.ToString(), Encoding.UTF8));
31     }
32 }
```

Source Code on GitHub

Source Generators Behind the Scenes

DummyClass.cs | DummyGenerator.cs | LinkerLibraryClient: Publish | LinkerLibraryClient.csproj

LinkerLibrary | LinkerLibrary.DummyClass | DoSomething()

```
using DummyCodeGenerator;

namespace LinkerLibrary
{
    2 references | Rainer Stropek, 19 minutes ago | 1 author, 1 change
    public partial class DummyClass
    {
        [Generate(Length = 3)]
        2 references | Rainer Stropek, 19 minutes ago | 1 author, 1 change
        public partial void DoSomething();
    }
}
```

```
<Project Sdk="Microsoft.NET.Sdk">

    <PropertyGroup>
        <TargetFramework>net6.0</TargetFramework>
        <Nullable>enable</Nullable>
        <IsTrimmable>true</IsTrimmable>
    </PropertyGroup>

    <ItemGroup>
        <ProjectReference Include="..\DummyCodeGenerator\DummyCodeGenerator.csproj"
                          OutputItemType="Analyzer" ReferenceOutputAssembly="false"/>
    </ItemGroup>

</Project>
```

dummy_generat...s [generated]

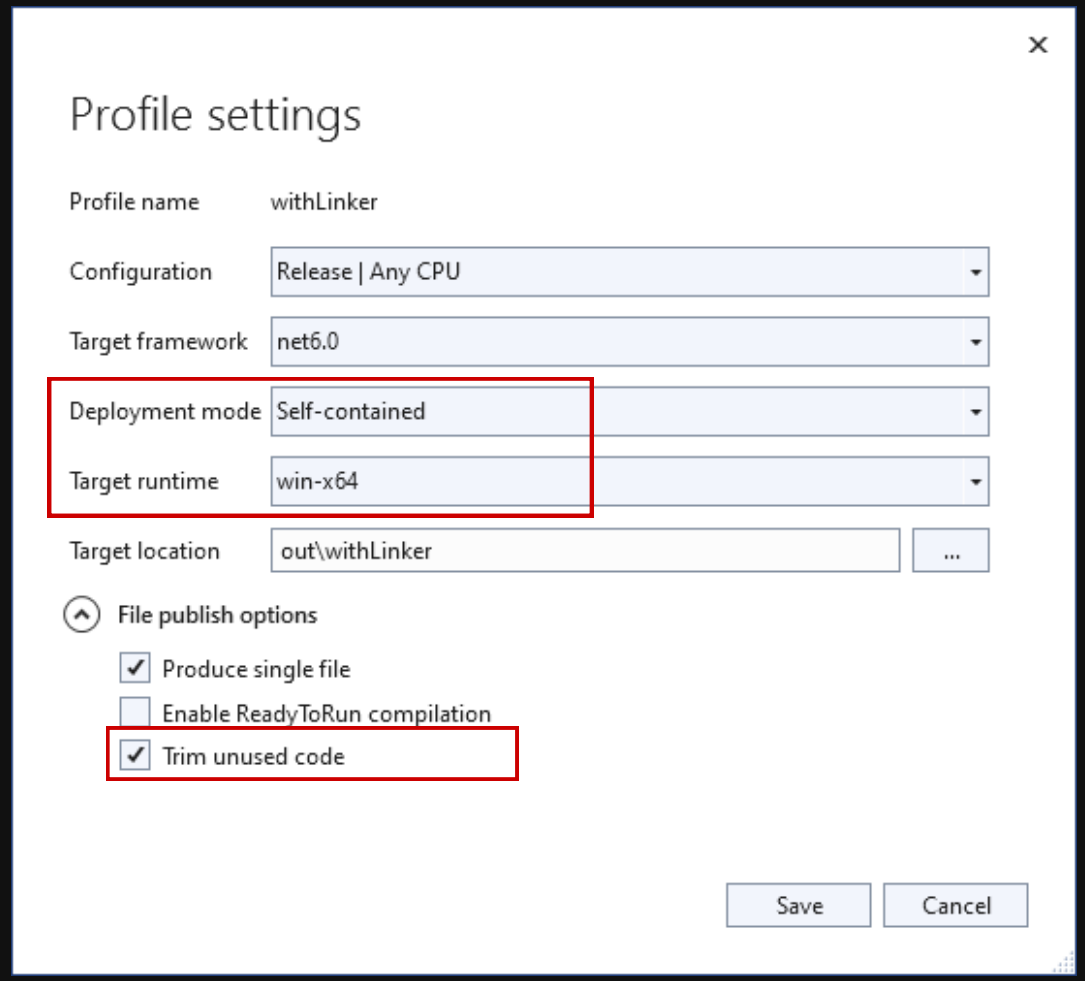
This file is auto-generated by the generator 'DummyCodeGenerator.DummyGenerator' and cannot be edited.

Miscellaneous Files | LinkerLibrary.DummyClass | DoSomething()

```
using System;
using static System.Diagnostics.Debug;
namespace LinkerLibrary
{
    partial class DummyClass
    {
        public partial void DoSomething()
        {
            WriteLine("Generating dummy code with length = 3");
            var rnd = new Random();

            var num0 = rnd.Next(100000);
            if (num0 == 45713) WriteLine($"Guess it correctly, it is {num0}");


            var num1 = rnd.Next(100000);
            if (num1 == 85076) WriteLine($"Guess it correctly, it is {num1}");


            var num2 = rnd.Next(100000);
            if (num2 == 79752) WriteLine($"Guess it correctly, it is {num2}");


            WriteLine("Done evaluating");
        }
    }
}
```

Source Code on GitHub

Trimmed Assemblies




```
1  <Project Sdk="Microsoft.NET.Sdk">
2
3      <PropertyGroup>
4          <OutputType>Exe</OutputType>
5          <TargetFramework>net6.0</TargetFramework>
6          <!--
7              Note that <TrimMode>link</TrimMode> is default in .NET 6.
8              See https://docs.microsoft.com/en-us/dotnet/core/deploying/trimming-optio
9          -->
10
11         <!-- Note new support for compressing single-file bundles (expanded in-memory
12         <EnableCompressionInSingleFile>true</EnableCompressionInSingleFile>
13     </PropertyGroup>
14
15     <ItemGroup>
16         <ProjectReference Include="..\LinkerLibrary\LinkerLibrary.csproj" />
17     </ItemGroup>
18
19     <!--
20         You can prevent trimming of assemblies. There are many additional options for
21         Read more about them at https://docs.microsoft.com/en-us/dotnet/core/deployin
22     -->
23     <ItemGroup>
24         <TrimmerRootAssembly Include="System.Text.Json" />
25     </ItemGroup>
26 </Project>
```

Source Code on GitHub

Demo Time!

JSON Serialization Enhancements

Yet Additional JSON APIs??

- Support for streaming through *IAsyncEnumerable* 🔗
  - Significant performance improvement in some scenarios
- JSON Writeable DOM API 🔗
  - For scenarios in which POCOs are not an option
    - Data structures not known at runtime
  - Potential for perf optimization
    - Deserialize only a subtree of large JSON in POCOs
    - Manipulate a subtree without deserializing everything into POCOs

Demo Time!

Source on GitHub

Linq Enhancements

Community 🤘🚀👏

Demo Time!

Source on GitHub

DateOnly 📅
TimeOnly ⏰

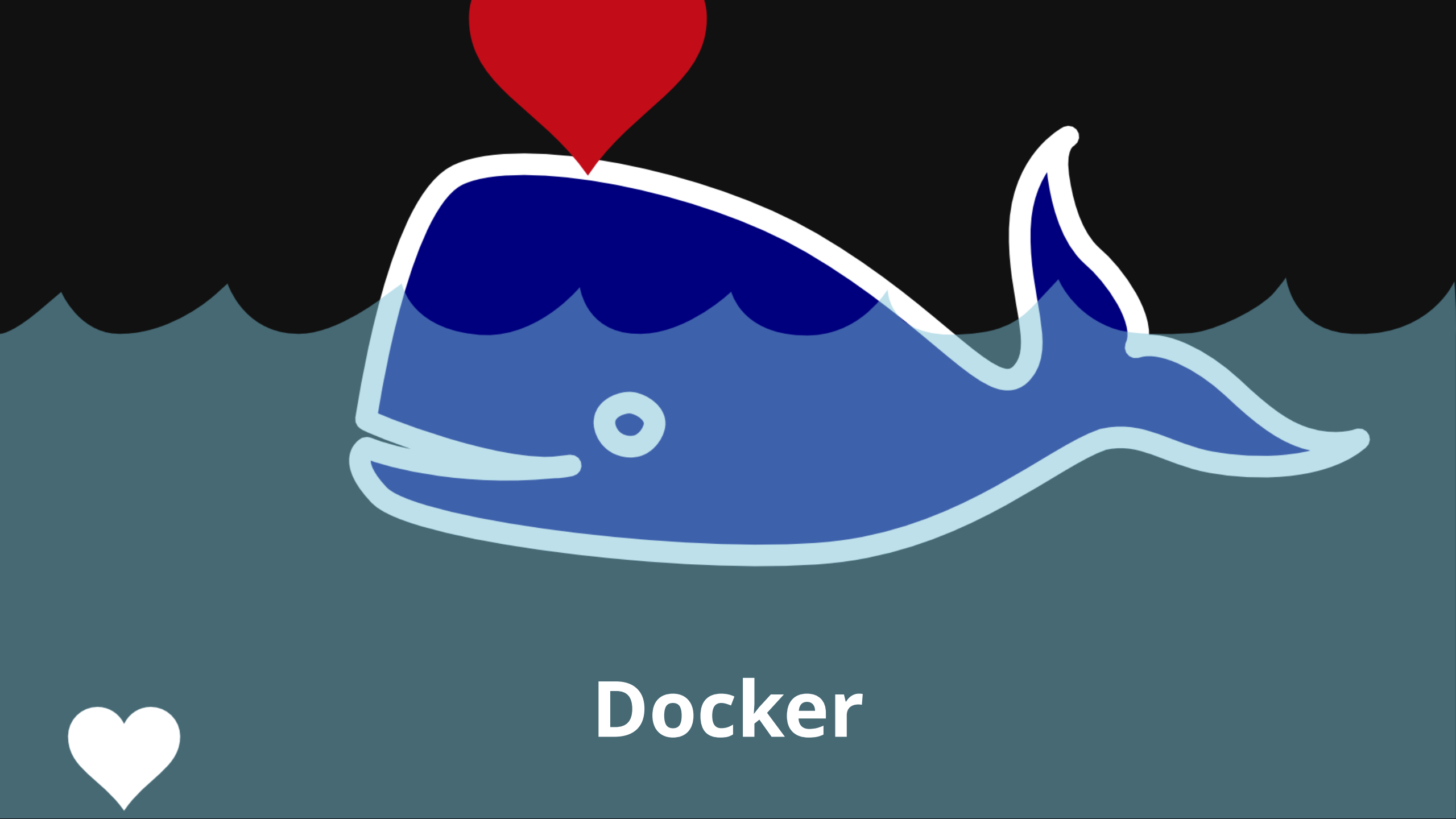
finally 🥳😅

Demo Time!

Source on GitHub

Additional Date/Time-Related Enhancements

- Better performance of *DateTime.UtcNow* 🔗
- Enhanced support for time zone names on all platforms 🔗


Docker

Demo Time!

Source on GitHub

What else?

- Lots of new analyzers 🔗
- File stream perf improvements on Windows 🔗
- *JsonSerializer* supports ignoring cycles 🔗
- New *PriorityQueue* = Queue with priorities 🔗
- More libraries (e.g. *SignalR*) are annotated for *Nullable*
- Enhance startup performance with *Profile-Guided Optimization* (PGO) 🔗

.NET 6 🤘

Rainer Stropek | @rstropek