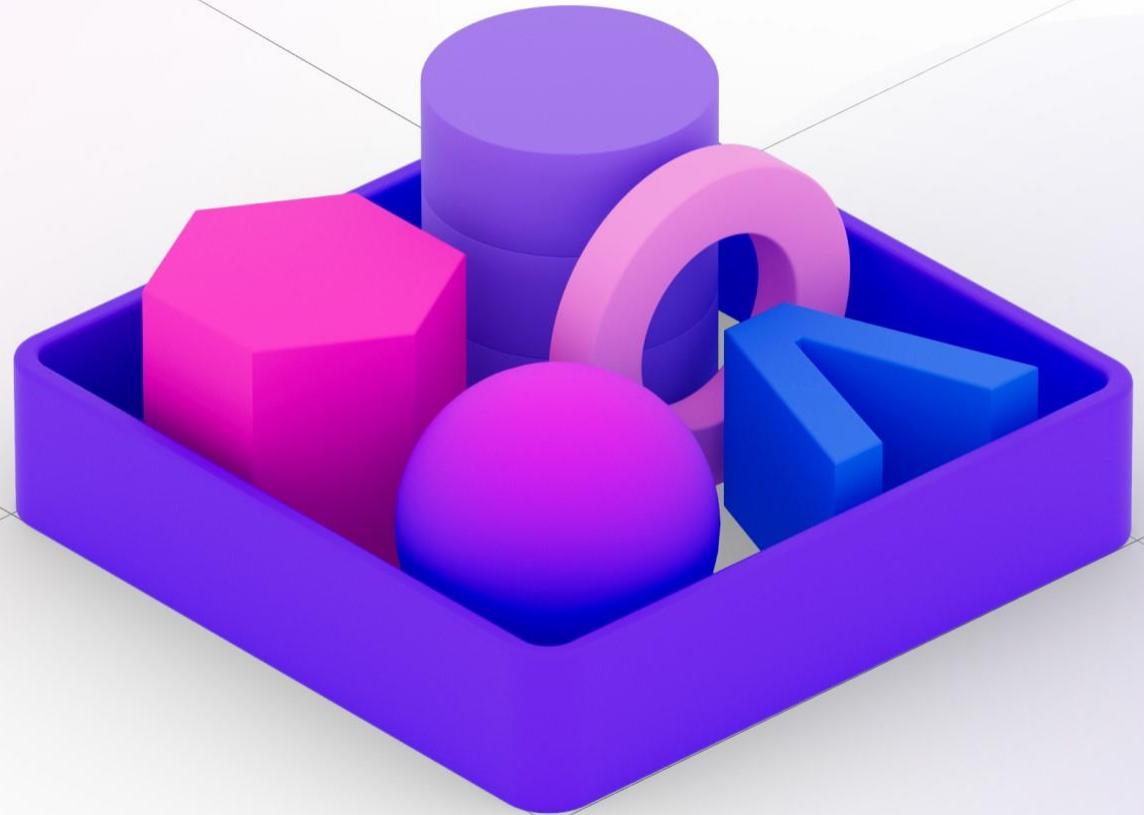


.NET Conf



.NET High Performance programming

Hai Nguyen Duy

Senior Architect

Azure Cybersecurity Expert

Parker Solar Probe

692.000 km/h



Agenda

How .NET manage memory

Tool to measure performance

Optimize performance
detail practices

Q&A



1.

How .NET manage memory



Levels of optimization

Level	Description
High level design (Architectural Level)	Make the best use of the resources (images, sounds, and so on), goals, any constraints, and the expected load (e.g. microservices, data caches, load balancing, database design, denormalize column, indexing, scalability vertical – horizontal, Monitoring and Analytics Log performance, parallelism in batch job ...)
Source code level (focus)	Depend on the source language, the target machine language C# 6,7,8 (changes continuously), the framework inside (e.g. EF, N+1 query) and the compiler. (e.g. dispose resource,...), Data model, Algorithms & Data structures efficiency

Levels of optimization

Level	Description
Compilation and Machine Code Optimization Level	Related to the process of compiling source code into machine code. Compilers may perform optimizations to improve execution time, efficiently use the CPU, and take advantage of specific characteristics of the computer architecture
Deployment and System Configuration Level	involves fine-tuning the deployment environment and system configuration to ensure the application operates efficiently in production conditions. This may include server configuration, database optimization, and managing system resources.

Memory Allocations

.NET is “Managed Memory Runtime”: the runtime environment (Common Language Runtime - CLR) takes care of memory allocation

Stack and Heap:

The stack is used for storing value types and method call information, while the heap is used for reference types and objects.

Garbage Collection:

.NET has a garbage collector that automatically reclaims memory from objects that are no longer in use, preventing memory leaks.



Generational Collection Process

Mark Phase:	<ul style="list-style-type: none">identifies and marks objects that are still in use and reachable from the root objects.
Compact Phase (optional):	<ul style="list-style-type: none">the marked objects are compacted to reduce memory fragmentation. This phase rearranges objects in memory, moving them closer together to create contiguous blocks of free memory.This is particularly useful in scenarios where memory fragmentation can be a problem, such as long-running applications.
Finalization Phase:	<ul style="list-style-type: none">call the finalizer methods of objects that have finalizers which are used for cleaning up resources like files, database connections, etc.The garbage collector postpones finalization until the objects are no longer accessible and marks them for finalization in the previous phase.
Completion Phase:	<ul style="list-style-type: none">the garbage collector performs a second pass to clean up objects that have been marked for finalization. These objects' finalizers are called to release unmanaged resources.
Reclamation Phase:	<ul style="list-style-type: none">the memory occupied by objects that are neither reachable nor have finalizers is reclaimed. The memory is returned to the heap, making it available for new object allocations.

Garbage Collection

.NET's heap memory is divided into three generations :

Generation 0:

1. Youngest generation.
2. Newly allocated objects.
3. Frequent garbage collection.

Generation 1:

1. Objects survived Generation 0 collections.
2. Less frequent garbage collection.
3. Objects may be promoted to Generation 2.

Generation 2:

1. Oldest generation.
2. Objects survived multiple collections.
3. Least frequent garbage collection.
4. Persistent objects for the application's lifetime.

Benefits of Generational GC

Efficiency:

Frequent collection of the Young Generation reduces overhead.

Faster Collections:

Smaller spaces are quicker to scan.

Improved Application Performance:

Reduced pause times for garbage collection.



Performance tips

1 Best way to reduce time taken
for GC is to not allocate at all

2 Try to keep objects
short-lived

3 Try to keep objects small to avoid
them ending up on the LOH.

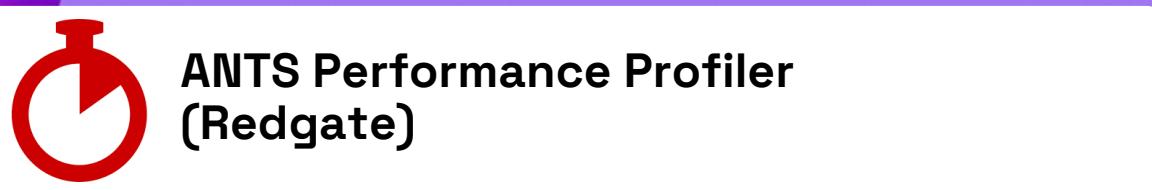


2.

NET performance tools



.NET Tools



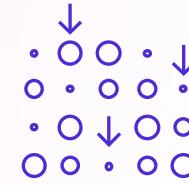
Benchmark .NET



Library for .NET
benchmarking



High precision
measurements



Extra data and
output available
using diagnoses



Compare performance
on different platform,
architectures, JIT
versions and GC models

BenchmarkDotNet

Transform methods into benchmarks,
track their performance, and share
reproducible measurement experiments

```
[SimpleJob(RuntimeMoniker.Net472, baseline: true)]
[SimpleJob(RuntimeMoniker.NetCoreApp30)]
[SimpleJob(RuntimeMoniker.NativeAot70)]
[SimpleJob(RuntimeMoniker.Mono)]
[RPlotExporter]
public class Md5VsSha256
{
    private SHA256 sha256 = SHA256.Create();
    private MD5 md5 = MD5.Create();
    private byte[] data;

    [Params(1000, 10000)]
    public int N;

    [GlobalSetup]
    public void Setup()
    {
        data = new byte[N];
        new Random(42).NextBytes(data);
    }

    [Benchmark]
    public byte[] Sha256() => sha256.ComputeHash(data);

    [Benchmark]
    public byte[] Md5() => md5.ComputeHash(data);
}
```

BenchmarkDotNet=v0.12.0, OS=Windows 10.0.17763.805 (1809/October2018Update/Redstone5)						
Intel Core i7-7700K CPU 4.20GHz (Kaby Lake), 1 CPU, 8 logical and 4 physical cores						
[Host]						
Net472	: .NET Framework 4.7.2 (4.7.3468.0), X64 RyuJIT					
NetCoreApp30	: .NET Core 3.0.0 (CoreCLR 4.700.19.46205, CoreFX 4.700.19.46214), X64 RyuJIT					
NativeAot70	: .NET 7.0.0-preview.4.22172.7, X64 NativeAOT					
Mono	: Mono 6.4.0 (Visual Studio), X64					
Method	Runtime	N	Mean	Error	StdDev	Ratio
Sha256	.NET 4.7.2	1000	7.735 us	0.1913 us	0.4034 us	1.00
Sha256	.NET Core 3.0	1000	3.989 us	0.0796 us	0.0745 us	0.50
Sha256	NativeAOT 7.0	1000	4.091 us	0.0811 us	0.1562 us	0.53
Sha256	Mono	1000	13.117 us	0.2485 us	0.5019 us	1.70
Md5	.NET 4.7.2	1000	2.872 us	0.0552 us	0.0737 us	1.00
Md5	.NET Core 3.0	1000	1.848 us	0.0348 us	0.0326 us	0.64
Md5	NativeAOT 7.0	1000	1.817 us	0.0359 us	0.0427 us	0.63
Md5	Mono	1000	3.574 us	0.0678 us	0.0753 us	1.24
Sha256	.NET 4.7.2	10000	74.509 us	1.5787 us	4.6052 us	1.00
Sha256	.NET Core 3.0	10000	36.049 us	0.7151 us	1.0025 us	0.49
Sha256	NativeAOT 7.0	10000	36.253 us	0.7076 us	0.7571 us	0.49
Sha256	Mono	10000	116.350 us	2.2555 us	3.0110 us	1.58
Md5	.NET 4.7.2	10000	17.308 us	0.3361 us	0.4250 us	1.00
Md5	.NET Core 3.0	10000	15.726 us	0.2064 us	0.1930 us	0.90
Md5	NativeAOT 7.0	10000	15.627 us	0.2631 us	0.2461 us	0.89
Md5	Mono	10000	30.205 us	0.5868 us	0.6522 us	1.74

DotNet Diagnostics tool

dotnet-counters && dotnet-dump

```
ca. Command Prompt - dotnet-counters monitor --refresh-interval 1 -p 24316
Press p to pause, r to resume, q to quit.
  Status: Running

[System.Runtime]
% Time in GC since last GC (%)          0
Allocation Rate (B / 1 sec)            8,168
CPU Usage (%)                         0
Exception Count (Count / 1 sec)        0
GC Committed Bytes (MB)              88.449
GC Fragmentation (%)                  0.771
GC Heap Size (MB)                     60.378
Gen 0 GC Count (Count / 1 sec)        0
Gen 0 Size (B)                        1,507,464
Gen 1 GC Count (Count / 1 sec)        0
Gen 1 Size (B)                        51,017,960
Gen 2 GC Count (Count / 1 sec)        0
Gen 2 Size (B)                        0
IL Bytes Jitted (B)                  209,625
LOH Size (B)                          4,616,800
Monitor Lock Contention Count (Count / 1 sec) 0
Number of Active Timers               0
Number of Assemblies Loaded           119
Number of Methods Jitted              2,372
POH (Pinned Object Heap) Size (B)    171,264
ThreadPool Completed Work Item Count (Count / 1 sec) 0
ThreadPool Queue Length                0
ThreadPool Thread Count                0
Time spent in JIT (ms / 1 sec)         0
Working Set (MB)                      283.623
```

```
> gcroot 023cb5011c90
Caching GC roots, this may take a while.
Subsequent runs of this command will be faster.

HandleTable:
  0000023c9fa011f8 (strong handle)
    -> 023ca4000020      System.Object[]
    -> 023cb28188f0      testwebapi.Controllers.Processor
    -> 023cb2818908     testwebapi.Controllers.CustomerCache
    -> 023cb2818920     System.Collections.Generic.List<testw
    -> 023cba04f218     testwebapi.Controllers.Customer[]
    -> 023cb5011cf0     testwebapi.Controllers.Customer
    -> 023cb5011c90     System.String

Thread 13ac:
  baa947f610 7fff45bdca87 System.Threading.PortableThreadPool+Gat
  iate.CoreLib/src/System/Threading/PortableThreadPool.GateThread.cs
    r14:
      -> 023ca4000020      System.Object[]
      -> 023cb28188f0      testwebapi.Controllers.Processor
      -> 023cb2818908     testwebapi.Controllers.CustomerCache
      -> 023cb2818920     System.Collections.Generic.List<testw
      -> 023cba04f218     testwebapi.Controllers.Customer[]
      -> 023cb5011cf0     testwebapi.Controllers.Customer
      -> 023cb5011c90     System.String

Found 2 unique roots.
```

DotNet Diagnostics tool

dotnet-trace and perfview

```
dhn@NB30305 ~
> dotnet-trace collect -p 35536 --providers Microsoft-DotNETCore-SampleProfiler

Provider Name          Keywords          Level          Enabled By
Microsoft-DotNETCore-SampleProfiler  0xFFFFFFFFFFFFFFF  Verbose(5)  --providers

Process      : D:\.NET\DiagnosticScenarios.exe
Output File   : C:\Users\dhn\DiagnosticScenarios.exe_20231103_145704.netrace
[00:00:00:36] Recording
Press <Enter> or <Ctrl+C>
Stopping the trace. This may take a few seconds...
Trace completed.
```

The screenshot shows the output of the `dotnet-trace collect` command and the interface of the DotNet Diagnostics tool. The terminal window displays the command and its execution details. The main window is a performance analysis tool with various tabs and filters, showing a detailed breakdown of the recorded metrics.

Name	Inc %	Inc	Inc Ct	Exc %	Exc	Exc Ct	Fold	Fold Ct	When	First	Last
✓ROOT	100.0	330,040.3	165,321	0.0	0	0	0	0	*****	0.372	36,671.137
+✓Process64 Process(35536) (35536) Arg	100.0	330,040.3	165,321	0.0	0	0	0	0	*****	0.372	36,671.137
+✓(Non-Activities)	100.0	330,040.3	165,321	0.0	0	0	0	0	*****	0.372	36,671.137
+✓Threads	100.0	330,040.3	165,321	0.0	0	0	0	0	*****	0.372	36,671.137
+✓Thread (34564)	11.1	36,670.9	18,369	0.0	0	0	0	0	99A99A9A9A9999AA99A9AAA9999	0.409	36,671.137
+✓module System.Private.CoreLib.i	11.1	36,670.9	18,369	11.1	36,671	18,369	36,671	18,369	99A99A9A9A9999AA99A9AAA9999	0.409	36,671.137
+✓Thread (35088)	11.1	36,670.8	18,369	0.0	0	0	0	0	99A99A9A9A99AAA9AAAAAA99999A	0.372	36,671.137
+✓module DiagnosticScenarios <<	11.1	36,670.8	18,369	0.0	0	0	0	0	99A99A9A9A99AAA9AAAAAA99999A	0.372	36,671.137
+✓module Microsoft.Extensions.H	11.1	36,670.8	18,369	0.0	0	0	0	0	99A99A9A9A99AAA9AAAAAA99999A	0.372	36,671.137
+✓module System.Private.CoreL	11.1	36,670.8	18,369	11.1	36,671	18,369	36,671	18,369	99A99A9A9A99AAA9AAAAAA99999A	0.372	36,671.137

VS Profiling (CPU usage)

The screenshot displays the Visual Studio Diagnostic Tools interface, specifically the CPU Usage and Call Stack windows.

CPU Usage window:

Function Name	Total CPU [unit, %]	Self CPU [unit, %]	Module
iisexpress (PID: 7280)	14452 (100.00%)	88 (0.61%)	iisexpress
[External Call] system.private.corelib.dll!0x000007ff...	14357 (99.34%)	20 (0.14%)	system.private.cor...
testwebapi.Controllers.DiagScenarioController.h...	14337 (99.20%)	1252 (8.66%)	diagnosticscenarios
[External Call] System.Diagnostics.Stopwatch.Stop()	5859 (40.54%)	5859 (40.54%)	system.private.cor...
[External Call] System.Diagnostics.Stopwatch.Start()	5702 (39.45%)	5702 (39.45%)	system.private.cor...
[External Call] System.Diagnostics.Stopwatch.get_...	1297 (8.97%)	1297 (8.97%)	system.private.cor...
[External Call] system.private.corelib.dll!0x000007ff...	83 (0.57%)	83 (0.57%)	system.private.cor...
[External Call] system.private.corelib.dll!0x000007ff...	80 (0.55%)	80 (0.55%)	system.private.cor...
[External Call] system.private.corelib.dll!0x000007ff...	15 (0.10%)	15 (0.10%)	system.private.cor...
[External Call] system.private.corelib.dll!0x000007ff...	8 (0.06%)	8 (0.06%)	system.private.cor...

Call Stack window:

```
121    {
122        var watch = new Stopwatch();
123        watch.Start();
124
125        while (true)
126        {
127            watch.Stop();
128            if (watch.ElapsedMilliseconds > milliseconds)
129                break;
130            watch.Start();
131        }
132
133        return "success:highcpu";

```

Diagnostic Tools window:

- Diagnostics session: 2:01 minutes (2:01 min selected)
- Events
- Process Memory (MB)
- CPU (% of all processors)

Solution Explorer window:

- Solution 'DiagnosticScenarios' (1 of 1 project)
 - External Sources
 - DiagnosticScenarios
 - Connected Services
 - Dependencies
 - Properties
 - launchSettings.json
 - Controllers
 - C# DiagnosticScenarios.cs
 - C# ValuesController.cs
 - appsettings.json
 - build.bat
 - investigateleak.md
 - Program.cs
 - readme.md
 - Startup.cs

VS Profiling (CPU usage)

The screenshot shows the Visual Studio Diagnostic Tools interface during a CPU usage profiling session. The main window displays the CPU Usage profile for the file `DiagnosticScenarios.cs`. The timeline shows a single event spanning 2:01 minutes. The CPU usage chart indicates high activity, peaking at approximately 100% usage. The `Top Functions` section highlights two external calls to `System.Diagnostics.Stopwatch` methods, which account for 99.9% of the CPU time.

CPU Usage (Caller/Callee view) details:

- Calling Functions:** `system.private.corelib.dll!0x...` (14337, 99.20%)
- Current Function:** `testwebapi.Controllers.DiagS...14337` (99.20%)
- Called Functions:**
 - `System.Diagnostics.Stopwatch...` 5859 (40.54%)
 - `System.Diagnostics.Stopwatch...` 5702 (39.45%)
 - `System.Diagnostics.Stopwatch...` 1297 (8.97%)
 - `system.private.corelib.dll!0x0000...` 83 (0.57%)
 - `system.private.corelib.dll!0x0000...` 80 (0.55%)
 - [Other] (0.44%)

Code Editor: Shows the `DiagnosticScenarios.cs` file with the following code snippet:

```
D:\.NET performance\dotnetsample\samples-main\core\diagnostics\DiagnosticScenarios\Controllers\DiagnosticScenarios.cs:121
    115     }
    116 }
    117 [HttpGet]
    118 [Route("highcpu/{milliseconds}")]
    119 public ActionResult<string> highcpu(int milliseconds)
    120 {
    121     var watch = new Stopwatch();
    122     watch.Start();
    123
    124     while (true)
    125     {
    126         watch.Stop();
    127     }
    128 }
```

Watch 1: No issues found.

Call Stack: Shows the stack trace for the current thread, indicating an await operation and the entry point `DiagnosticScenarios.dll!DiagnosticScenarios.Program.Main(string[] args) Line 16`.

Solution Explorer: Shows the project structure for `DiagnosticScenarios`, including files like `DiagnosticScenarios.cs`, `ValuesController.cs`, and `Program.cs`.

VS Profiling (CPU usage)

The screenshot displays the Visual Studio Diagnostic Tools interface, specifically the CPU Usage view, for a diagnostic scenarios project. The main window shows a call tree of CPU usage, event timelines, memory usage, and CPU usage over time. The Solution Explorer on the right shows the project structure.

Solution Explorer: Shows the project 'DiagnosticScenarios' with files like 'DiagnosticScenarios.cs', 'ValuesController.cs', 'Program.cs', and 'Startup.cs'.

Diagnostic Tools: The central area displays the following data:

- Call Tree:** Shows the hierarchy of CPU usage. Top items include 'iisexpress (PID: 7280)' (100.00%), '[External Call] system.private.corelib.dll!0x0...' (99.34%), and 'testwebapi.Controllers.DiagScenarioCon...' (99.20%).
- Events:** A timeline showing various events over 2:01 minutes.
- Process Memory (MB):** A chart showing memory usage in MB over time, with tabs for GC, Snapshot, and Private Bytes.
- CPU (% of all processors):** A chart showing CPU usage percentage over time.
- Top Insights:** No insights found.
- Top Functions:** Function names and their CPU usage:
 - [External Call] System.Diagnostics.Stopwatch.Stop() (Kernel : 99.9% (14441))
 - [External Call] System.Diagnostics.Stopwatch.Start() (Other : < 0.1% (8))
- Threads:** A pie chart showing thread usage: Kernel (99.9%), Other (< 0.1%), and ASP.NET (< 0.1%).

Watch 1: A watch window showing variables and their values.

Call Stack: A call stack window showing the current stack trace, indicating a waiting operation.

VS Profiling (memory usage)

The screenshot displays the Visual Studio Diagnostic Tools interface, specifically the Memory Usage tab, during a profiling session. The main window shows a table of managed memory objects, a timeline of events, and memory and CPU usage graphs.

Managed Memory (Table):

Object Type	Count	Size (Bytes)
testwebapi.Controllers.Customer	400,000	48,000,000
List<testwebapi.Controllers.Customer>	1	4,194,360
NativeRuntimeEventSource	1	276,832
RuntimeType+RuntimeTypeCache	253	178,984
Microsoft.Extensions.DependencyInjection.ServiceLookup.ConstructorCallSite	221	101,568
Signature	602	73,344
Dictionary<String, Object>	16	67,456
RuntimeType	1,532	61,280
ConcurrentDictionary+Tables<Microsoft.Extensions.DependencyInjection.ServiceLookup.ServiceCach	1	36,512
Dictionary<String, String>	31	34,592
TplEventSource	1	33,872
Total	407,154	53,566,840

Diagnostic Tools (Timeline):

Diagnostics session: 2:17 minutes

Events: 1:50min, 2:00min, 2:10min, 2:20min

Process Memory (MB): 158

CPU (% of all processors): 100

Solution Explorer:

- Solution 'DiagnosticScenarios' (1 of 1 project)
 - External Sources
 - DiagnosticScenarios
 - Properties
 - launchSettings.json
 - Controllers
 - DiagnosticScenarios.cs
 - ValuesController.cs
 - appsettings.json
 - build.bat
 - investigateleak.md
 - Program.cs
 - readme.md
 - Startup.cs

VS Profiling (memory usage)

The screenshot displays the Visual Studio Diagnostic Tools interface during a memory usage session. The main window shows three primary panels: 'Process Memory (MB)', 'CPU (% of all processors)', and a 'Summary' table.

Process Memory (MB) Panel: Shows memory usage over time. The Y-axis represents memory in MB, ranging from 0 to 158. The X-axis shows time points: 2:30min, 2:40min, 2:50min, and 3:00min. A blue bar indicates the private bytes usage, which remains relatively stable around 158 MB across the session.

CPU (% of all processors) Panel: Shows CPU usage percentage over time. The Y-axis ranges from 0 to 100%. The X-axis shows the same time points as the memory panel. The CPU usage is near 0% throughout the session.

Summary Table:

ID	Time	Objects (Diff)	Heap Size (Diff)	Label
1	11.25s	19,202	(n/a)	1,822.27 KB (n/a)
2	52.34s	819,445 (+800,243 ↑)	52,724.54 KB (+50,902.27 KB ↑)	

Left Side Panels:

- Instances of testwebapi.Controllers.Customer:** A list of customer instances with their memory sizes. All entries show "Not available. Values are only available when stopped in the debugger." and a size of 120 bytes.
- Watch 1:** A watch window showing the current values of variables.
- Call Stack:** A call stack window showing the current call hierarchy.

Solution Explorer: Shows the project structure for 'DiagnosticScenarios'.

- Solution:** DiagnosticScenarios (1 of 1 project)
 - External Sources
 - DiagnosticScenarios
 - Connected Services
 - Dependencies
 - Properties
 - launchSettings.json
 - Controllers
 - DiagnosticScenarios.cs
 - ValuesController.cs
 - appsettings.json
 - build.bat
 - investigateleak.md
 - Program.cs
 - readme.md
 - Startup.cs

dotMemory (license)



dump_20231103_141645_dmp – JetBrains dotMemory

Home + Analysis #1

Process Dump
#0 CLR 7.0.13

Snapshot #1
All objects
862.3 K objects
53.30 MB total size

Largest types
System.String
424.6 K objects
38.00 MB total size

Group by: Types Dominators Similar Retention Instances Call Tree Call Tree (Icicle Chart) Back Traces Generations

This view groups objects by similarity of their retention paths. The graph on the right shows two key paths to roots that are most different from each other. [?](#)

Filter: Ctrl+Shift+F: Filter by namespaces, types, arrays, and generic arguments. Type !g to exclude generic args from matching or !a to exclude arrays. [?](#) [Clear](#)

Type	Objects	Bytes
String System	420,480	39,525,120
String System	1,809	123,242
String System	20	62,310
String System	521	25,524
String System	118	10,866
String System	220	10,354
String System	138	8,566
String System	151	6,840
String System	138	6,180
String System	23	4,212
String System	105	3,670
String System	62	3,278
String System	36	1,844
String System	13	1,830

The retention graph on the right side of the interface illustrates the memory lifetime of various objects. It starts with an 'Interior local variable' at the bottom, which points to an 'Object[]'. This array points to a 'CustomerCache' object, which in turn points to a 'Processor' object. The 'Processor' object points to a 'cache' field, which points to a 'List<Customer>' object. This list points to a 'Customer[]' array, which contains multiple 'Customer' objects. One such customer object has an 'id' field. The graph uses arrows to show the flow of references between objects.

dotTrace (license)



dtsn_2023-11-03_083116819_.Huqymyf.tmp - 12/3/2023, 7:57:04 AM – JetBrains dotTrace Viewer

File Edit View Help

Filters ✖

Events ?

Not Sel... 36,672 ms

Subsystems ?

	ms	%
<input type="checkbox"/>	36,672 ms	100

Timeline ✖

1 filter applied (clear all): highcpu ✖

out in []

Filtered intervals 0

ID	Name	ms	%
33716	CLR Worker	36,672 ms	100
35088	CLR Worker		
35052	CLR Worker		
18520	CLR Worker		
29252	CLR Worker		
15244	CLR Worker		
34564	CLR Worker		
31220	CLR Worker		
34956	CLR Worker		

Hotspots under 'highcpu' ✖

Total / Own+System time Plain List

Search Functions

100 % highcpu • 36,672 ms / 301 ms • testwebapi.Controllers.DiagScenarioController.highcpu(int)
51.9 % Start • 19,037 ms / 19,037 ms • System.Diagnostics.Stopwatch.Start()
47.3 % Stop • 17,334 ms / 17,334 ms • System.Diagnostics.Stopwatch.Stop()

User code

Call Tree ✖

Backtraces Flame graph

100 % highcpu • 36,672 ms • testwebapi.Controllers.DiagScenarioController.highcpu(int)
51.9 % Start • 19,037 ms • System.Diagnostics.Stopwatch.Start()
47.3 % Stop • 17,334 ms • System.Diagnostics.Stopwatch.Stop()

dotMemory Unit test performance

Check for objects

```
dotMemory.Check(memory =>
{
    Assert.That(memory.GetObjects(where => where.Type
        .Is<Foo>())
        .ObjectsCount,
        Is.EqualTo(0)));
});
```

Compare snapshots

```
var memoryCheckPoint1 = dotMemory.Check();
foo.Bar();
var memoryCheckPoint2 = dotMemory.Check(memory =>
{
    Assert.That(memory.GetTrafficFrom(memoryCheckPoint1)
        .Where(obj => obj.Interface.Is<IFoo>())
        .AllocatedMemory.SizeInBytes, Is.LessThan(1000));});
bar.Foo();
dotMemory.Check(memory =>
{
    Assert.That(memory.GetTrafficFrom(memoryCheckPoint2)
        .Where( obj => obj.Type.Is<Bar>())
        .AllocatedMemory.ObjectsCount, Is.LessThan(10));});
```

Check memory traffic

```
var memoryCheckPoint = dotMemory.Check();

foo.Bar();

dotMemory.Check(memory =>
{
    Assert.That(memory.GetDifference(memoryCheckPoint)
        .GetSurvivedObjects()
        .GetObjects(where => where
            .Namespace.Like("MyApp"))
        .ObjectsCount, Is.EqualTo(0));
});
```

Continue analysis in dotMemory

```
[DotMemoryUnit(SavingStrategy = SavingStrategy.OnAnyFail,
    Directory = @"C:\tmp\Class",
    WorkspaceNumberLimit = 200,
    DiskSpaceLimit = 104857600)]
```

ILSpy

The screenshot shows the ILSpy interface with the following details:

- File Bar:** File, View, Window, Help.
- Toolbars:** Standard toolbar with icons for Open, Save, Undo, Redo, Cut, Copy, Paste, Find, and Search.
- Search Bar:** C# 11.0 / VS 2022, search icon.
- Assemblies Tree:** Shows the loaded assemblies:
 - System.Private.CoreLib (6.0.0.0, .NETCoreApp, v6.0)
 - System.Private.Uri (6.0.0.0, .NETCoreApp, v6.0)
 - System.Linq (6.0.0.0, .NETCoreApp, v6.0)
 - System.Private.Xml (6.0.0.0, .NETCoreApp, v6.0)
 - System.Xaml (6.0.2.0, .NETCoreApp, v6.0)
 - WindowsBase (6.0.2.0, .NETCoreApp, v6.0)
 - PresentationCore (6.0.2.0, .NETCoreApp, v6.0)
 - PresentationFramework (6.0.2.0, .NETCoreApp, v6.0)
 - ConsoleApp1 (1.0.0.0, .NETCoreApp, v6.0)
 - Metadata
 - Debug Metadata (From portable PDB)
 - References
 - {} -
 - ConsoleApp1
 - Program
 - Microsoft.CodeAnalysis
 - System.Runtime.CompilerServices
 - System.Runtime (6.0.0.0, .NETCoreApp, v6.0)
 - System.Net.Http (6.0.0.0, .NETCoreApp, v6.0)
 - System.Console (6.0.0.0, .NETCoreApp, v6.0)
 - System.Runtime.InteropServices (6.0.0.0, .NETCoreApp, v6.0)
 - System.Runtime.CompilerServices.Unsafe (6.0.0.0, .NETCoreApp, v6.0)
- Program Tab:** The current tab is Program. The code shown is:

```
// ConsoleApp1, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
// ConsoleApp1.Program
+ using ...

internal class Program
{
    [CompilerGenerated]
    private sealed class <AsyncDownload>d__1 : IAsyncStateMachine
    {
        public int <>1__state;
        public AsyncTaskMethodBuilder<string> <>t__builder;
        private HttpClient <client>5__1;
        private string <>s_2;
        private TaskAwaiter<string> <>u_1;
        private void MoveNext()
        {
            int num = <>1__state;
            string result;
            try
            {
                TaskAwaiter<string> awaier;
                if (num != 0)
                {
                    <client>5__1 = new HttpClient();
                    awaier = <client>5__1.GetStringAsync("https://msdn.microsoft.com").GetAwaiter();
                    if (!awaier.IsCompleted)
                    {
                        num = (<>1__state = 0);
                        <>u_1 = awaier;
                        <AsyncDownload>d__1 stateMachine = this;
                        <>t__builder.AwaitUnsafeOnCompleted(ref awaier, ref stateMachine);
                        return;
                    }
                }
                else
                {
                    awaier = <>u_1;
                    <>u_1 = default(TaskAwaiter<string>);
                    num = (<>1__state = -1);
                }
                <>s_2 = awaier.GetResult();
                result = <>s_2;
            }
        }
    }
}
```

dotPeek

The screenshot displays three panes of the dotPeek application:

- Assembly Explorer**: Shows the project structure for "ConsoleApp1". It includes two projects: "ConsoleApp1 (1.0.0.0, msil, .NETCoreApp v7.0, Debug)" and "ConsoleApp1 (1.0.0.0, msil, .NETCoreApp v6.0, Debug)". The "Program" class under the second project is selected.
- Program.cs**: Displays the C# source code for the "Program" class. The code defines a static void Main method and an asynchronous Task<string> AsyncDownload method. The AsyncDownload method uses an HttpClient to download a web page.
- IL Viewer**: Shows the Low-level C# IL for the same code. It includes generated assembly code for the AsyncDownload method, which involves creating a state machine and a task builder.

```
namespace ConsoleApp1
{
    internal class Program
    {
        static void Main(string[] args)
        {
            try
            {
                AsyncDownload().GetAwaiter().GetResult();
                Console.ReadLine();
            }
            catch (Exception e)
            {
                Console.WriteLine(e);
                throw;
            }
        }

        static async Task<string> AsyncDownload()
        {
            HttpClient client = new HttpClient();
            //Asynchronously download the contents of a web page
            return await client.GetStringAsync("https://msdn.microsoft.com");
        }
    }
}
```

```
namespace ConsoleApp1
{
    [NullableContext(1)]
    [Nullable(0)]
    internal class Program
    {
        private static void Main(string[] args)
        {
            try
            {
                Program.AsyncDownload().GetAwaiter().GetResult();
                Console.ReadLine();
            }
            catch (Exception e)
            {
                Console.WriteLine((object) e);
                throw;
            }
        }

        [AsyncStateMachine(typeof (Program.<>AsyncDownload>d_1))]
        [DebuggerStepThrough]
        private static Task<string> AsyncDownload()
        {
            Program.<>AsyncDownload>d_1 stateMachine = new Program.<>AsyncDownload>d_1();
            stateMachine.<>t__builder = AsyncTaskMethodBuilder<string>.Create();
            stateMachine.<>1__state = -1;
            stateMachine.<>t__builder.Start<Program.<>AsyncDownload>d_1>(<ref> stateMachine);
            return stateMachine.<>t__builder.Task;
        }

        public Program()
        {
            base..ctor();
        }

        [CompilerGenerated]
        private sealed class <>AsyncDownload>d_1 : IAsyncStateMachine
        {
            public int <>1__state;
            [Nullable(0)]
            public AsyncTaskMethodBuilder<string> <>t__builder;
            [Nullable(0)]
            private HttpClient <>client>s_1;
            [Nullable(0)]
            private string <>s_2;
            [Nullable(new byte[] { 0, 1 })]
            private TaskAwaiter<string> <>u_1;

            public <>AsyncDownload>d_1()
            {
            }
        }
    }
}
```

Production metrics and monitoring

- Load testing
- Stress Testing

Apache Jmeter

Gatling

Locust

NBomber

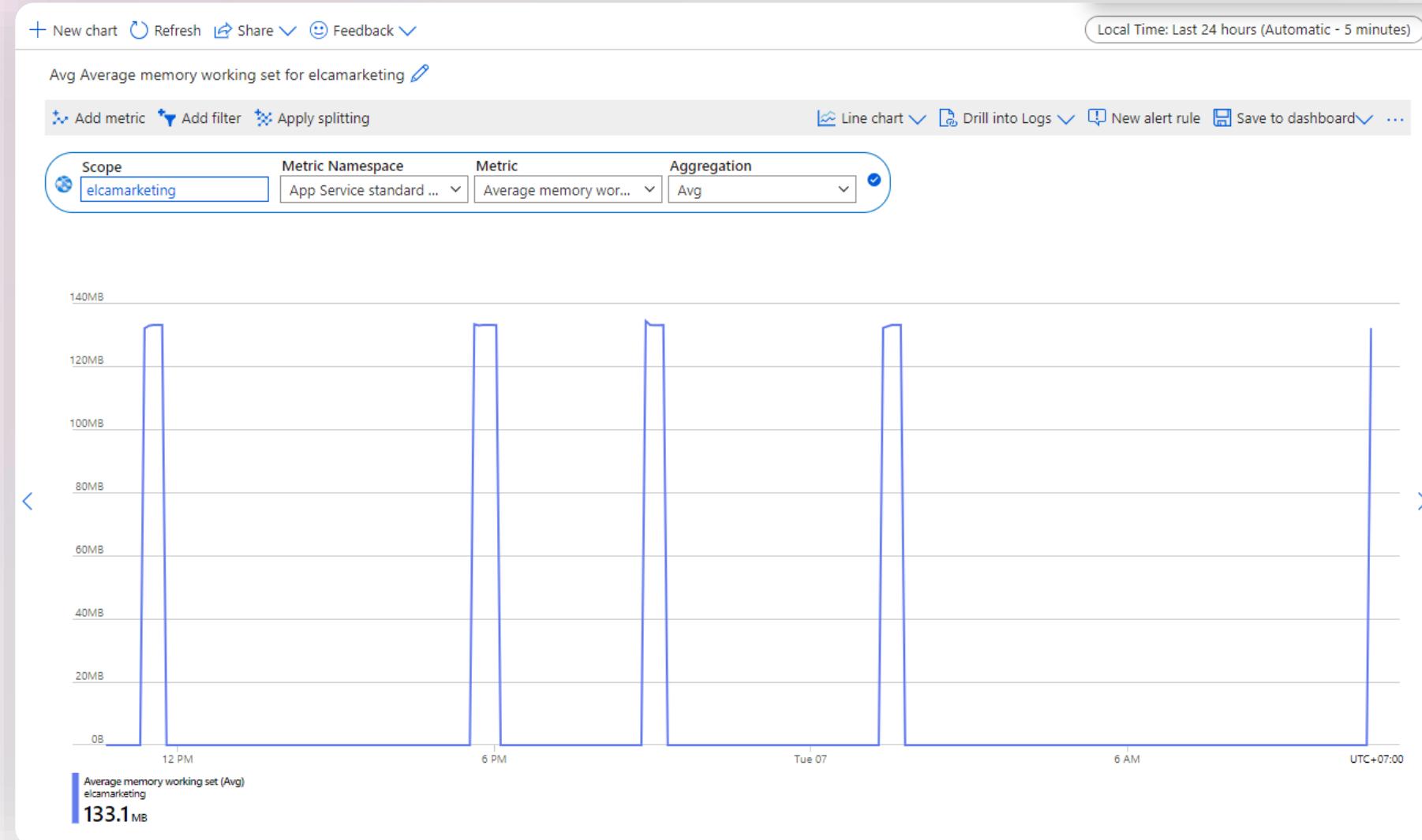
Visual Studio Load Test

Production metrics and monitoring

OnPremise/Cloud

- OpenTelemetry + Prometheus+ Grafana
- Elastic search + correlationId

Azure monitoring



Production metrics and monitoring

Home > DiagServiceEastUS

DiagServiceEastUS | Performance

Application Insights

Search Refresh Code Optimizations Profiler View in Logs Analyze with Workbooks Copy link Feedback

Overview Activity log Access control (IAM) Tags Diagnose and solve problems

Investigate Application map Smart detection Live metrics Transaction search Availability Failures

Performance Troubleshooting guides (preview)

Monitoring Alerts Metrics

Local Time: Last 24 hours Roles = All

Operations Dependencies Roles

Operation times: zoom into a range Avg 50th 95th 99th

Request count

15 sec
10 sec
5.0sec
0 sec

0 200 0 2.50k 0 1.50k 0 500 0 1.0ms 0 43ms 0 520ms 0 2.9sec 0 13sec 0 55sec

06:00 PM 09:00 PM Sat 4 03:00 AM 06:00 AM 09:00 AM 12:00 PM 03:00 PM 06:00 PM 09:00 PM Sat 4 03:00 AM 06:00 AM 09:00 AM 12:00 PM 03:00 PM

04:42 PM 04:42 PM

Overall

Distribution of durations: zoom into ... Scale

Profiler Request count Duration

2.50k 2k 1.50k 1k 500 0 2.50k 2k 1.50k 1k 500 0 1.0ms 43ms 520ms 2.9sec 13sec 55sec

1.0ms 1.0ms 1.0ms 1.3min

Insights (1)

Top 3 Dependencies

OPERATION NAME	DURATION (AVG)	COUNT	PIN
...			

Production metrics and monitoring

Github Copilot

Home > DiagServiceEastUS | Performance >

Code Optimizations

DiagServiceEastUS | PREVIEW

Code Optimizations analyzes profiler traces to help you improve the performance of your cloud applications. This AI-based service works in tandem with the Application Insights Profiler to detect CPU and memory bottlenecks at runtime and provides code-level recommendations on how to fix them. [Learn More](#).

Refresh View dismissed Learn how to use Code Optimizations

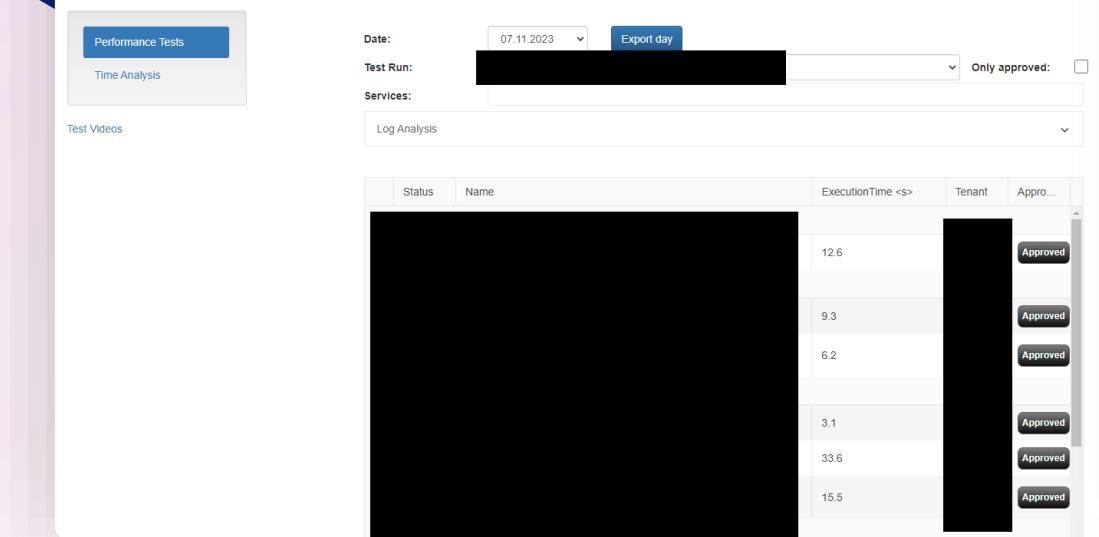
If you are interested in participating in an on-going paid user study on Code Optimizations, please complete this [short survey](#).

Type	Performance Issue	Method	Count	Impact	Role
Memory	Inefficient `String.Substring()`	DiagService.Runners.StringOperationRunner.Run	1	68%	diagservicecw
Memory	Excessive allocations due to String.Substring	DiagService.Runners.StringOperationRunner.Run	1	68%	diagservicecw
Memory	Inefficient String Concatenation	DiagService.Runners.StringOperationRunner.Run	1	9.2%	diagservicecw
Memory	String.ToLower() Taking Too Much CPU/Memory	DiagService.Runners.StringOperationRunner.Run	1	7.9%	diagservicecw
Memory	Unnecessary String Allocations and CPU Usage Due to 'String.Split()'	DiagService.Runners.StringOperationRunner.Run	1	7%	diagservicecw
CPU	Array.Sort is causing high CPU usage	DiagService.Runners.ArrayOperationRunner.Run	1	4.2%	diagservicecw
CPU	Stopwatch.GetElapsedDateTimeTicks is causing high CPU usage	DiagService.Controllers.PerfCPUController.HighCpuEnumerableCount	1	2.8%	diagservicecw
CPU	String.Substring is causing high CPU usage	DiagService.Runners.StringOperationRunner.Run	1	2.5%	diagservicecw
CPU	Inefficient `String.Substring()`	DiagService.Runners.StringOperationRunner.Run	1	2.5%	diagservicecw
CPU	Inefficient String Concatenation	DiagService.Runners.StringOperationRunner.Run	1	1.9%	diagservicecw
CPU	`IEnumerable<T>.Any()` Taking Too Much CPU/Memory	DiagService.Runners.EnumerableOperationRunner.Run	1	1.6%	diagservicecw
CPU	Excessive Allocations of Dictionaries/Lists/StringBuilder/HashSet Types	DiagService.Controllers.MemoryController.GarbageCollection	1	0.8%	diagservicecw

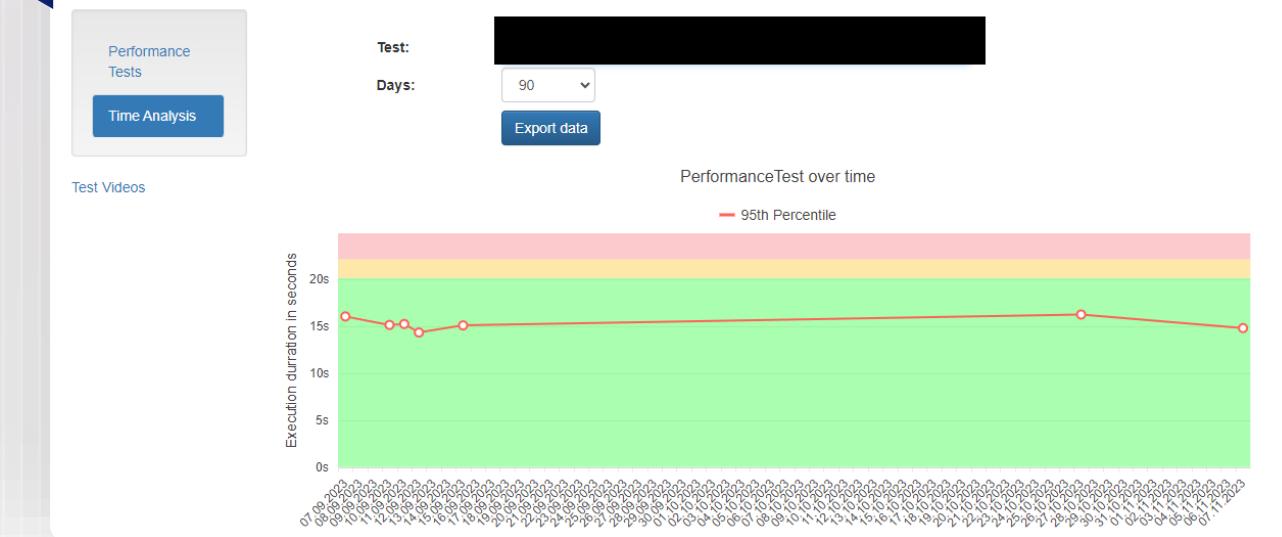
Production metrics and monitoring

Measure selenium from open to finish on UI

Performance Dashboard



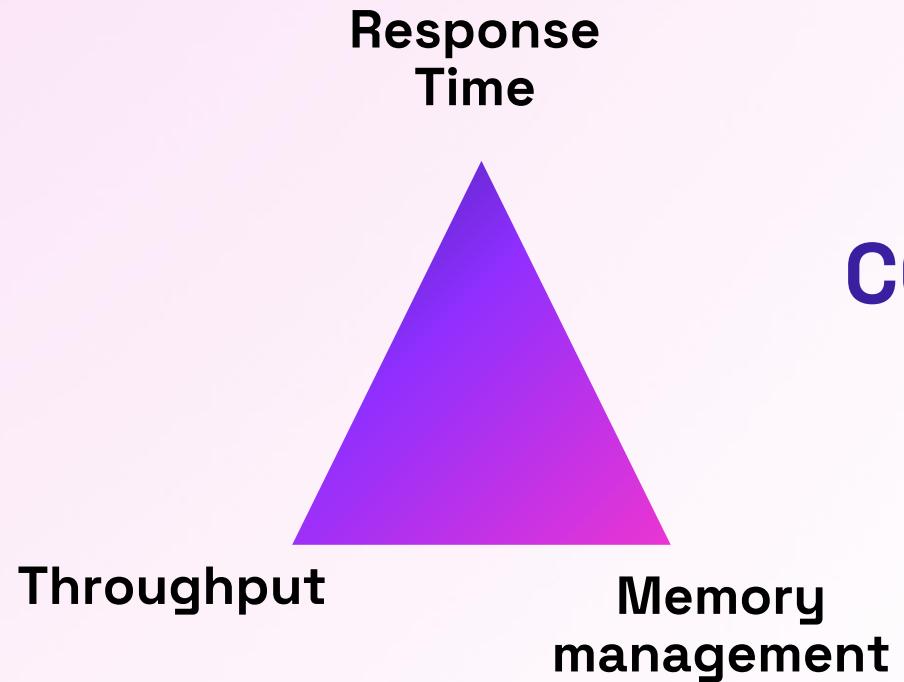
Performance Dashboard



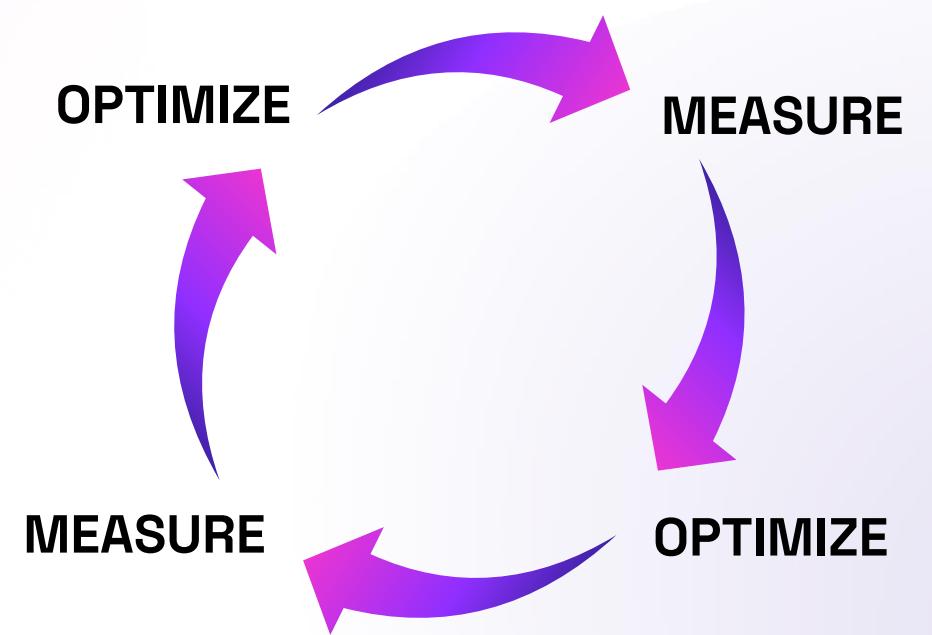
3.

Optimize detail lesson
learns in C# applications

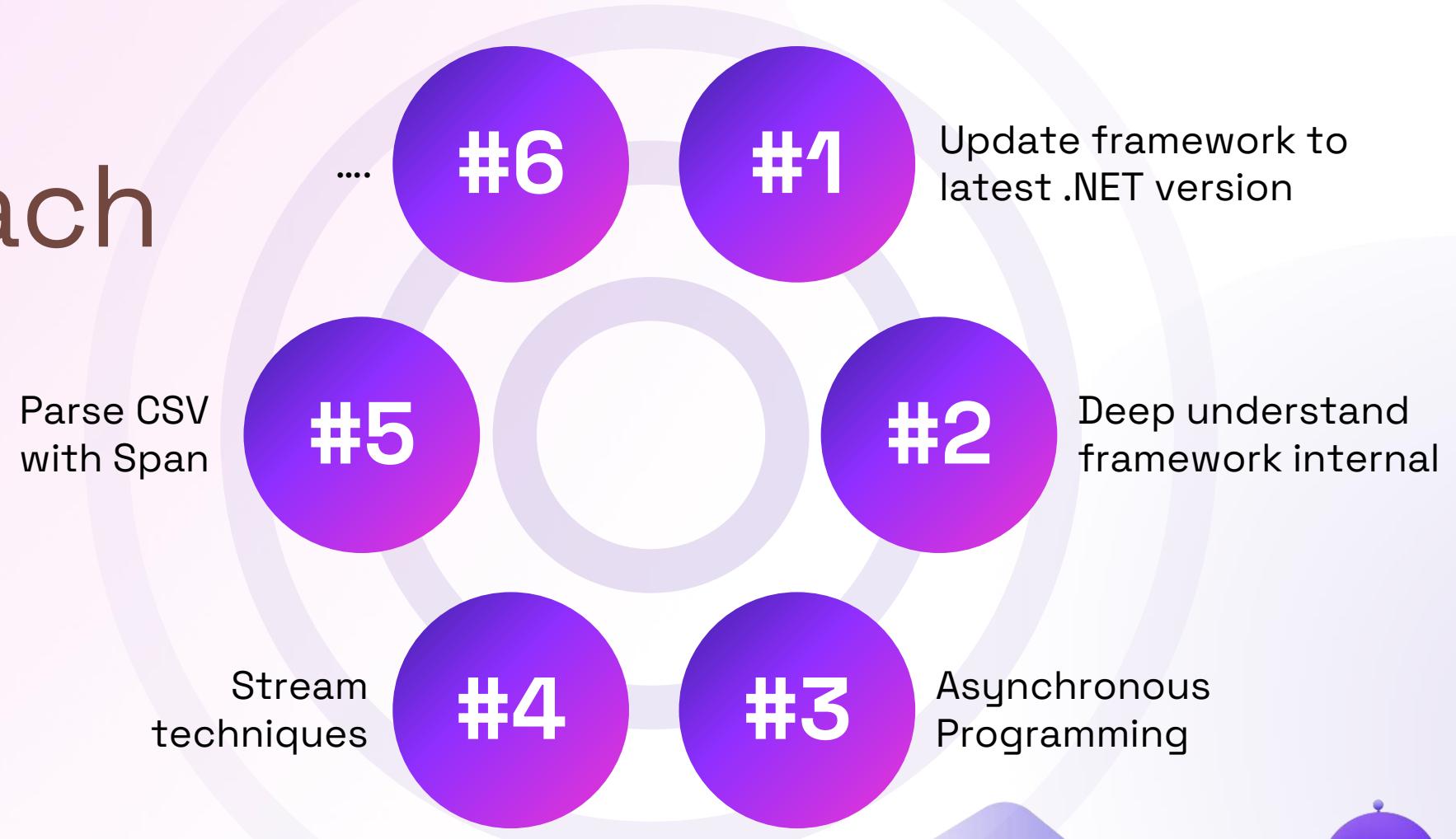
Different aspects of performance



CONTEXTUAL



Focus approach



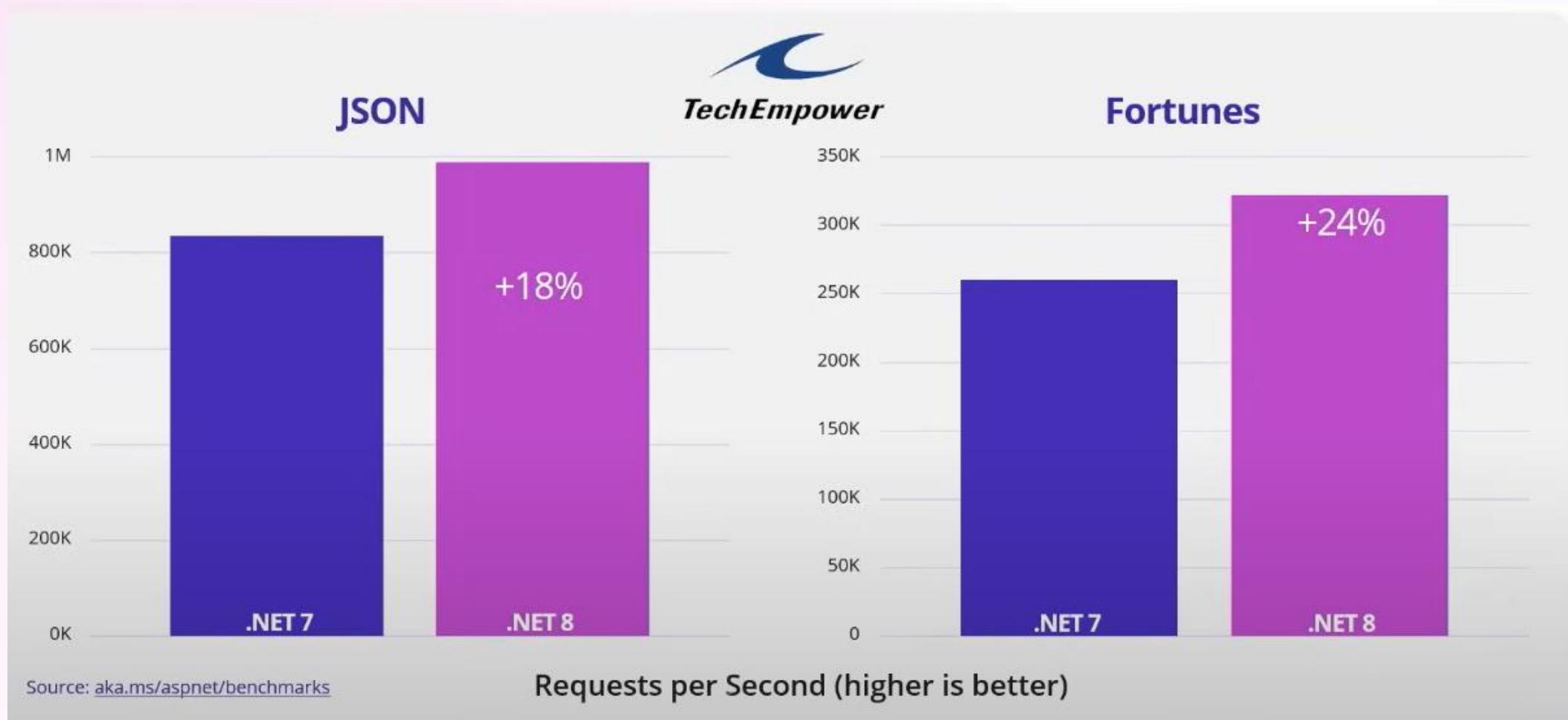
#1

Update framework to latest .NET version

<https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-8/>



Stephen Toub - MSFT



#1

Update framework to latest .NET version

Dynamic PGO with GDV cloning / Hoisting

```
private static int Sum(Func<int, int> func)
{
    int sum = 0;
    for (int i = 0; i < 10_000; i++)
    {
        sum += func(i); // delegate call
    }
    return sum;
}
```



```
private static int Sum(Func<int, int> func)
{
    int sum = 0;
    if (func.Method == KnownLambda)
    {
        for (int i = 0; i < 10_000; i++)
        {
            sum += i + 1;
        }
    }
    else
    {
        for (int i = 0; i < 10_000; i++)
        {
            sum += func(i);
        }
    }
    return sum;
}
```

- Tier0+Instr
 - Tracks the method invoked at the delegate call site
- Tier1
 - Sees that just one method is invoked at the call site, and adds a GDV check
 - Inlines the call on the path where the check succeeds
 - Sees that the GDV check is a loop invariant, and clones the loop to create a check-free version

#1

Update framework to latest .NET version

```
[GlobalSetup]
0 references
public void GlobalSetup()
    => source = Enumerable.Range(0, 100).ToList();

[Benchmark]
0 references
public int SumFor()
{
    var sum = 0;
    for (var index = 0; index < source.Count;
        sum += source[index]);
    return sum;
}

[Benchmark]
0 references
public int SumForEach()
{
    var sum = 0;
    foreach (var item in source)
        sum += item;
    return sum;
}

[Benchmark]
0 references
public int SumLINQ()
=> source.Sum();
```

.NET 6 internal code MS

```
public static int Sum(this IEnumerable<int> source)
{
    if (source == null)
    {
        ThrowHelper.ThrowArgumentNullException(ExceptionArgument.source);
    }

    int sum = 0;
    checked
    {
        foreach (int v in source)
        {
            sum += v;
        }
    }

    return sum;
}
```

.NET 8 internal code MS

```
private static TResult Sum<TSource, TResult>(this IEnumerable<TSource> source)
    where TSource : struct, INumber<TSource>
    where TResult : struct, INumber<TResult>
{
    if (source.TryGetSpan(out ReadOnlySpan<TSource> span))
    {
        return Sum<TSource, TResult>(span);
    }

    TResult sum = TResult.Zero;
    foreach (TSource value in source)
    {
        checked { sum += TResult.CreateChecked(value); }
    }

    return sum;
}
```

#1

Update framework to latest .NET version

// * Summary *										
BenchmarkDotNet v0.13.9+228a464e8be6c580ad9408e98f18813f6407fb5a, Windows 10 (10.0.19044.3448/21H2/November2021Update)										
11th Gen Intel Core i7-1165G7 2.80GHz, 1 CPU, 8 logical and 4 physical cores										
.NET SDK 8.0.100-rc.2.23502.2										
[Host] : .NET 6.0.24 (6.0.2423.51814), X64 RyuJIT AVX2										
.NET 6.0 : .NET 6.0.24 (6.0.2423.51814), X64 RyuJIT AVX2										
.NET 7.0 : .NET 7.0.13 (7.0.1323.51816), X64 RyuJIT AVX2										
.NET 8.0 : .NET 8.0.0 (8.0.23.47906), X64 RyuJIT AVX2										
Method	Job	Runtime	Count	Mean	Error	StdDev	Median	Ratio	RatioSD	
SumFor	.NET 6.0	.NET 6.0	100	79.89 ns	1.672 ns	4.850 ns	79.72 ns	1.00	0.00	
SumFor	.NET 7.0	.NET 7.0	100	84.71 ns	2.166 ns	5.780 ns	84.91 ns	1.06	0.10	
SumFor	.NET 8.0	.NET 8.0	100	82.28 ns	1.819 ns	5.305 ns	82.65 ns	1.03	0.08	
SumForEach	.NET 6.0	.NET 6.0	100	68.10 ns	1.873 ns	5.493 ns	67.11 ns	1.00	0.00	
SumForEach	.NET 7.0	.NET 7.0	100	81.64 ns	3.115 ns	9.184 ns	79.64 ns	1.21	0.18	
SumForEach	.NET 8.0	.NET 8.0	100	68.05 ns	1.482 ns	4.275 ns	68.25 ns	1.01	0.08	
SumLINQ	.NET 6.0	.NET 6.0	100	547.51 ns	10.867 ns	26.036 ns	548.33 ns	1.00	0.00	
SumLINQ	.NET 7.0	.NET 7.0	100	88.42 ns	2.549 ns	7.274 ns	88.00 ns	0.16	0.02	
SumLINQ	.NET 8.0	.NET 8.0	100	24.40 ns	0.973 ns	2.743 ns	23.94 ns	0.05	0.01	
SumFor	.NET 6.0	.NET 6.0	10000	6,807.33 ns	319.006 ns	915.287 ns	6,800.14 ns	1.00	0.00	
SumFor	.NET 7.0	.NET 7.0	10000	7,247.09 ns	147.534 ns	428.024 ns	7,289.73 ns	1.08	0.15	
SumFor	.NET 8.0	.NET 8.0	10000	7,294.18 ns	206.693 ns	586.354 ns	7,260.04 ns	1.09	0.17	
SumForEach	.NET 6.0	.NET 6.0	10000	6,470.56 ns	249.755 ns	736.410 ns	6,558.94 ns	1.00	0.00	
SumForEach	.NET 7.0	.NET 7.0	10000	6,237.65 ns	124.279 ns	307.187 ns	6,243.26 ns	0.99	0.16	
SumForEach	.NET 8.0	.NET 8.0	10000	5,431.94 ns	185.965 ns	548.321 ns	5,392.62 ns	0.85	0.13	
SumLINQ	.NET 6.0	.NET 6.0	10000	55,142.98 ns	1,763.178 ns	5,087.172 ns	53,709.78 ns	1.00	0.00	
SumLINQ	.NET 7.0	.NET 7.0	10000	7,727.01 ns	167.770 ns	481.364 ns	7,718.31 ns	0.14	0.02	
SumLINQ	.NET 8.0	.NET 8.0	10000	1,116.88 ns	39.878 ns	116.325 ns	1,085.77 ns	0.02	0.00	

#2

Deep understand framework internal

EF core optimize performance (60% performance issues)

- **Proper Indexing**
- **Efficient Querying**

- Where, Select, AsNoTracking, pagination with methods like Skip and Take
- Avoid lazy loading for performance-critical scenarios
- Consider using explicit loading, eager loading with include for related entities when needed, avoid N+1 query problem when loading related entities
- RawSQL - For complex and performance-critical queries, bulk CRUD, cautious about SQL injection and security.
- Projections, Batch Fetching, Compiled Queries (reduce the overhead of query compilation), denormalization

- **Database design:**
normalized and well-structured to avoid performance bottlenecks
- **Caching:**
caching for frequently accessed data to reduce database calls and improve response times, Use libraries like MemoryCache or Redis for caching
- **Profiling and Monitoring:**
identify specific bottlenecks and prioritize optimizations based on real-world performance data - Entity Framework Profiler and Application Insights

#2

Deep understand framework internal

ASP.NET core optimize performance

- **Web API Response Compression:**

Enable response compression to reduce the payload size and improve data transfer times.

- **Request and Response Size Limiting:**

Configure limits for the size of incoming requests and outgoing responses. This can help prevent performance issues related to handling excessively large payloads.

- **Caching:** Implement caching for frequently requested data. Leverage caching mechanisms provided by ASP.NET Core. Consider using token-based authentication and caching authorization results when appropriate.

- **Use Content Delivery Networks (CDN):** Offload static content to a CDN to reduce the latency for users by serving assets from geographically distributed servers.

- **WEB MVC Compression and Minification:** Compress and minify static assets (CSS, JavaScript, images) to reduce the load time of your web pages. This can be achieved using tools like Gzip and Brotli, and by configuring bundling and minification

#3

Asynchronous Programming

Asynchronous Programming

Advantages:

- Non-blocking code.
- Allows the program to perform other tasks during slow I/O operations (e.g., file reading, network requests, database queries).

Use Cases:

- Ideal for I/O-bound tasks.

Synchronous Programming

Advantages:

- Generally, more efficient for CPU-bound tasks.
- No context-switching or task overhead.

Use Cases:

- Suitable for operations that don't involve I/O.

#3

Asynchronous Programming

```
PREFERENCE
public class TPLAsyncBetter
{
    [Benchmark]
    0 references
    public int SynchronousOperation()
    {
        int result = 0;
        for (int i = 0; i < 10; i++)
        {
            // Simulate an I/O-bound operation by reading a file synchronously.
            result += ReadFileSynchronously();
        }
        return result;
    }

    [Benchmark]
    0 references
    public async Task<int> AsynchronousOperation()
    {
        int result = 0;
        for (int i = 0; i < 10; i++)
        {
            // Simulate an I/O-bound operation by reading a file asynchronously.
            result += await ReadFileAsynchronously();
        }
        return result;
    }

    1 reference
    private int ReadFileSynchronously()
    {
        byte[] data = File.ReadAllBytes("Detail.txt");
        Thread.Sleep(100);
        return data.Length;
    }

    1 reference
    private async Task<int> ReadFileAsynchronously()
    {
        byte[] data = await File.ReadAllBytesAsync("Detail.txt");
        await Task.Delay(100);
        return data.Length;
    }
}
```

IO bound Async better

// * Summary *					
BenchmarkDotNet v0.13.9+228a464e8be6c580ad9408e98f18813f6407 11th Gen Intel Core i7-1165G7 2.80GHz, 1 CPU, 8 logical and .NET SDK 8.0.100					
[Host]	:	.NET 8.0.0 (8.0.23.53103), X64 RyuJIT AVX2	[DefaultJob]	:	.NET 8.0.0 (8.0.23.53103), X64 RyuJIT AVX2
Method		Mean	Error	StdDev	
SynchronousOperation		1.347 s	0.0214 s	0.0190 s	
AsynchronousOperation		1.319 s	0.0258 s	0.0394 s	

#3

Asynchronous Programming

```
[SimpleJob(RuntimeMoniker.Net80)]
1 reference
public class TPL
{
    3 references
    public static void LengthyTask()
    {
        int y = 0;
        for (int x = 0; x < 10; x++)
            y++;
    }
    [Benchmark]
    0 references
    public void SynchronousMethod()
    {
        LengthyTask();
    }

    [Benchmark]
    0 references
    public void TaskMethod()
    {
        Task.Run(new Action(LengthyTask));
    }

    [Benchmark]
    0 references
    public void AsynchronousTaskMethod()
    {
        var data = async () => await Task.Run(new
            Action(LengthyTask));
        data.Invoke();
    }
}
```

Simple code sync better

BenchmarkDotNet v0.13.9+228a464e8be6c580ad9408e98f18813f6407fb5a, Windows 10 11th Gen Intel Core i7-1165G7 2.80GHz, 1 CPU, 8 logical and 4 physical cores .NET SDK 8.0.100-rc.2.23502.2 [Host] : .NET 6.0.24 (6.0.2423.51814), X64 RyuJIT AVX2 .NET 8.0 : .NET 8.0.0 (8.0.23.47906), X64 RyuJIT AVX2			
Method	Mean	Error	StdDev
SynchronousMethod	6.187 ns	0.2425 ns	0.6995 ns
TaskMethod	200.707 ns	4.0488 ns	9.8554 ns
AsynchronousTaskMethod	553.373 ns	11.0415 ns	15.4787 ns

Asynchronous Programming – Batching and parallelism with When All

- ❑ **Issue async many independent tasks:** awaiting tasks one by one → async execution takes time.
- ❑ **Solution:** Use "WhenAll" to wait for all at once.
- ❑ **Advantage:** Boosts performance, reduces time, faster execution with "WhenAll" compared to awaiting each task separately.

The screenshot shows a Developer PowerShell window with the following content:

```
// * Summary *
BenchmarkDotNet=v0.13.1, OS=Windows 10.0.19043.1415 (21H1/May2021Update)
Intel Core i5-3330 CPU 3.00GHz (Ivy Bridge), 1 CPU, 4 logical and 4 physical cores
.NET SDK=6.0.101
[Host]    : .NET 6.0.1 (6.0.121.56705), X64 RyuJIT
DefaultJob : .NET 6.0.1 (6.0.121.56705), X64 RyuJIT

| Method | Mean | Error | StdDev |
|-----:|-----:|-----:|-----:|
| SynchronousAwait | 624.4 ms | 6.42 ms | 5.69 ms |
| AsynchronousWhenAll | 311.3 ms | 2.23 ms | 1.87 ms |
```

```
REFERENCES
public class TPLWhenAll
{
    2 references
    private async Task<int> TaskOne()
    {
        await Task.Delay(300);
        return 100;
    }
    2 references
    private async Task<string> TaskTwo()
    {
        await Task.Delay(300);
        return "TaskTwo";
    }
    [Benchmark]
    0 references
    public async Task SynchronousAwait()
    {
        int intValue = await TaskOne();
        string stringValue = await TaskTwo();
    }
    [Benchmark]
    0 references
    public async Task AsynchronousWhenAll()
    {
        var taskOne = TaskOne();
        var taskTwo = TaskTwo();
        await Task.WhenAll(taskOne, taskTwo);
    }
}
```

#3

Asynchronous Programming

Optimizing Execution:

- Use ConfigureAwait(false) to optimize context handling and avoid potential deadlocks.

CPU-Bound Tasks:

- Offload heavy CPU tasks with Task.Run.
- Ideal for large dataset processing or complex calculations.

Efficient Caching:

- Cache results for time-consuming operations.
- Avoids redundant work, improving overall performance.

```
public async Task<List<Customer>> GetCustomersAsync()
{
    using (var context = new MyDbContext()){
        // Do some work on the current synchronization context
        var result = await context.Customers.ToListAsync().ConfigureAwait(false);
        // The following code runs without attempting to resume on the original context
        return result;
    }
}
```

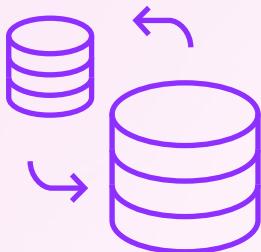
```
public async Task<int> CalculateResultAsync(int input)
{
    int result = await Task.Run(() => PerformComplexCalculation(input));
    return result;
}
```

```
private readonly ConcurrentDictionary<int, string> _cache = new ConcurrentDictionary<int, string>();

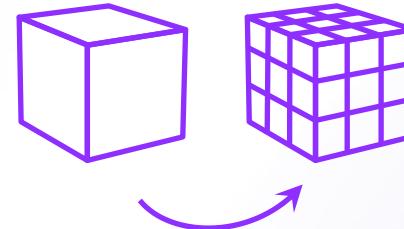
public async Task<string> GetCachedDataAsync(int key)
{
    return await _cache.GetOrAddAsync(key, async k => await FetchData(k));
}
```

#4

Stream techniques



- Streaming processes large XML files (or other data) with minimal memory usage, enabling sequential on-the-fly data handling.



- It's handy for very large XML files that don't fit in memory. Instead of loading the whole file at once, you process it bit by bit as it streams, keeping memory usage low and predictable.

#4

Stream techniques

Read big XML file partially

```
using (XmlReader reader = XmlReader.Create(xmlFilePath))
{
    while (reader.Read())
    {
        if (reader.NodeType == XmlNodeType.Element)
        {
            if (reader.Name == "NodeElement")
            {
                string elementValue =
                    reader.ReadElementContentAsString();
                Console.WriteLine($"NodeElement: {elementValue}");
            }
        }
    }
}
```

Download a Very Large File

```
string fileUrl = "https://verylargefile.zip";
string localFilePath = "downloadedfile.zip";

using (HttpClient httpClient = new HttpClient())
using (HttpResponseMessage response =
    await httpClient.GetAsync(fileUrl, HttpCompletionOption.ResponseHeadersRead))
using (Stream remoteStream =
    await response.Content.ReadAsStreamAsync())
using (FileStream fileStream =
    new FileStream(localFilePath, FileMode.Create,
    FileAccess.Write, FileShare.None, 8192, true))
{
    byte[] buffer = new byte[8192];
    int bytesRead;
    while ((bytesRead = await remoteStream.ReadAsync(buffer, 0, buffer.Length)) > 0)
    {
        await fileStream.WriteAsync(buffer, 0, bytesRead);
    }
    Console.WriteLine("File downloaded successfully.");
}
```

#4

Stream techniques

Event-Driven Processing	Move through the XML document element by element with a streaming reader, triggering events as needed.
Sequential Processing	Process data as you read it, avoiding storing the entire XML tree in memory. For example, extract specific data when encountering relevant elements without loading the entire document.
Low Memory Usage	Keep only a small part of the XML data in memory, ensuring a predictable and low memory footprint, regardless of file size.

#5

Parse CSV with Span

- **Problem:** Operations on "sub-data" require a temporary object to represent it, which introduces a lot of short-living objects. Typical examples are `string.Substring()` or sub-arrays/sub-lists.
- **Solution:** Introduce special types that allow slicing - representing sub-regions without a memory copy.
- **Benefits:** You can operate on subset of data without overhead of memory copy - especially useful in various "descendant" parsers.

#5

Parse CSV with Span

```
private static readonly string _dateString = "31 10 2023";

[Benchmark(Baseline = true)]
0 references
public DateTime Original()
{
    var day = _dateString.Substring(0, 2);
    var month = _dateString.Substring(3, 2);
    var year = _dateString.Substring(6);
    return new DateTime(int.Parse(year), int.Parse(month), int.Parse(day));
}

[Benchmark]
0 references
public DateTime Span()
{
    ReadOnlySpan<char> dateSpan = _dateString;
    var day = dateSpan.Slice(0, 2);
    var month = dateSpan.Slice(3, 2);
    var year = dateSpan.Slice(6);
    return new DateTime(int.Parse(year), int.Parse(month), int.Parse(day));
}
```

BenchmarkDotNet v0.13.9+228a464e8be6c580ad9408e98f18813f6407fb5a, Windows 10 (10.0.19041.1052)							
11th Gen Intel Core i7-1165G7 2.80GHz, 1 CPU, 8 logical and 4 physical cores							
.NET SDK 8.0.100-rc.2.23502.2							
[Host] : .NET 6.0.24 (6.0.2423.51814), X64 RyuJIT AVX2							
DefaultJob : .NET 6.0.24 (6.0.2423.51814), X64 RyuJIT AVX2							
Method	Mean	Error	StdDev	Ratio	RatioSD	Gen0	Allocated
Original	139.42 ns	2.861 ns	6.279 ns	1.00	0.00	0.0153	96 B
Span	80.31 ns	1.655 ns	4.122 ns	0.58	0.04	-	-

#6

TPL

- Provides a higher-level abstraction for parallelism, making it easier to write code that can efficiently utilize multiple processors and handle concurrent tasks.
- Boost performance with parallel loops by distributing work among multiple CPU cores. Swap regular `for` and `foreach` loops with their parallel `Parallel.For()` and `Parallel.ForEach()` equivalents when safe and feasible
- Use the `Partitioner` class to smartly split work into chunks, minimizing overhead and ensuring balanced parallel tasks. This optimization enhances both performance and workload distribution in C# applications

#6

TPL

```
0 references
public class TPLParallelForEach
{
    private int[] numbers;

    [Params(100)]
    public int N;

    [GlobalSetup]
    0 references
    public void Setup()
    {
        numbers = new int[N];
        Random rand = new Random();

        for (int i = 0; i < N; i++)
        {
            numbers[i] = rand.Next(1, 100);
        }
    }

    [Benchmark]
    0 references
    public void NormalSum()
    {
        long total = 0;
        numbers.ToList().ForEach(x => total += Sum(x));
    }

    [Benchmark]
    0 references
    public void ParallelSum()
    {
        long total = 0;
        Parallel.ForEach(numbers, (number) =>
        {
            total += Sum(number);
        });
    }
}
```

```
[Benchmark]
0 references
public void ParallelSumWithPartitioner()
{
    long total = 0;
    var partitioner = Partitioner.Create(numbers);
    Parallel.ForEach(numbers, (number) =>
    {
        total += Sum(number);
    });
}

5 references
private int Sum(int number)
{
    Thread.Sleep(1);
    return number;
}

[Benchmark]
0 references
public void ParallelSum4Core()
{
    long total = 0;
    int maxParallelism = 4; // Number of cores

    var semaphore = new SemaphoreSlim(maxParallelism);

    Parallel.ForEach(numbers, new ParallelOptions { MaxDegreeOfParallelism = maxParallelism }, (number) =>
    {
        semaphore.Wait();
        try
        {
            total += Sum(number);
        }
        finally
        {
            semaphore.Release();
        }
    });
}
```

#6

TPL

```
BenchmarkDotNet v0.13.9+228a464e8be6c580ad9408e98f18813f6407fb5a, Windows 10 (10.0.19044.3448/21H2/November2021Update)
11th Gen Intel Core i7-1165G7 2.80GHz, 1 CPU, 8 logical and 4 physical cores
.NET SDK 8.0.100-rc.2.23502.2
[Host]    : .NET 6.0.24 (6.0.2423.51814), X64 RyuJIT AVX2
DefaultJob : .NET 6.0.24 (6.0.2423.51814), X64 RyuJIT AVX2

| Method           | N   | Mean       | Error     | StdDev   |
| ----- |-----:|-----:|-----:|-----:|
| NormalSum       | 100 | 1,569.07 ms | 7.435 ms | 6.955 ms |
| ParallelSum     | 100 | 93.57 ms   | 11.824 ms | 32.959 ms |
| ParallelSumWithPartitioner | 100 | 115.99 ms   | 14.906 ms | 43.952 ms |
| ParallelSum4Core | 100 | 390.79 ms   | 2.969 ms  | 2.777 ms  |
| ParallelSum8Core | 100 | 249.43 ms   | 1.898 ms  | 1.682 ms  |

// * Warnings *
MultimodalDistribution
    TPLParallelForEach.ParallelSum: Default          -> It seems that the distribution is bimodal (mValue = 3.27)
    TPLParallelForEach.ParallelSumWithPartitioner: Default -> It seems that the distribution can have several modes (mValue = 3.17)

// * * * * *
```

#7

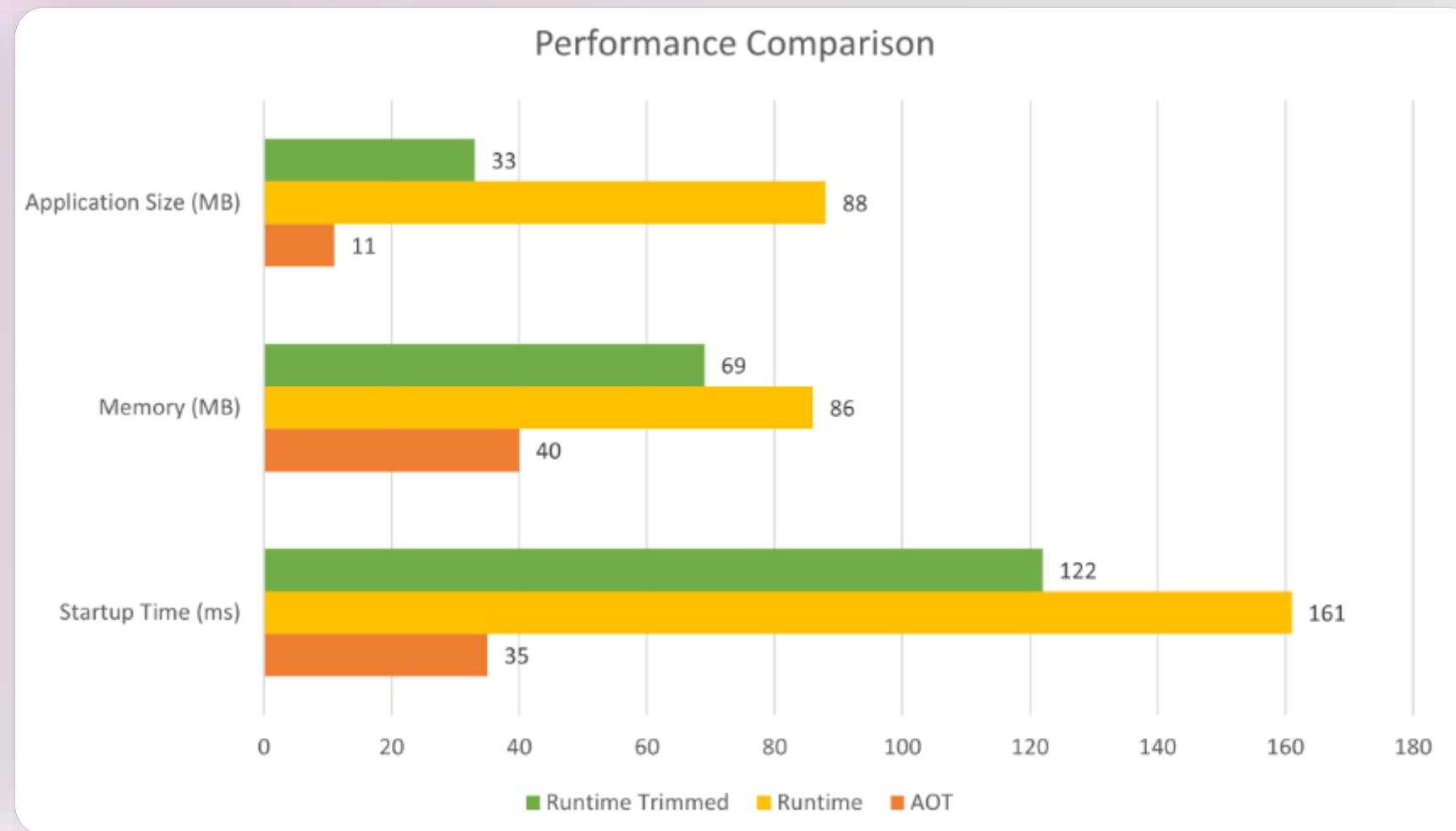
AOT NET 8 (not fully support)

Publishing and deploying a native AOT app provides the following benefits:

Minimized disk footprint	Produces a compact executable with only essential code from external dependencies. Benefits include smaller container images and faster deployment due to reduced executable size.
Reduced startup time	Faster start-up times in Native AOT apps mean quicker response to requests and smoother transitions in app version deployment for container orchestrators.
Reduced memory demand	Native AOT apps reduce memory usage, enhancing deployment density and scalability. CoreCRL's static PGO support enables optimizations, including performance gains based on a closed-world assumption, even without PGO.

#7

AOT.NET 8 (not fully support)



#7

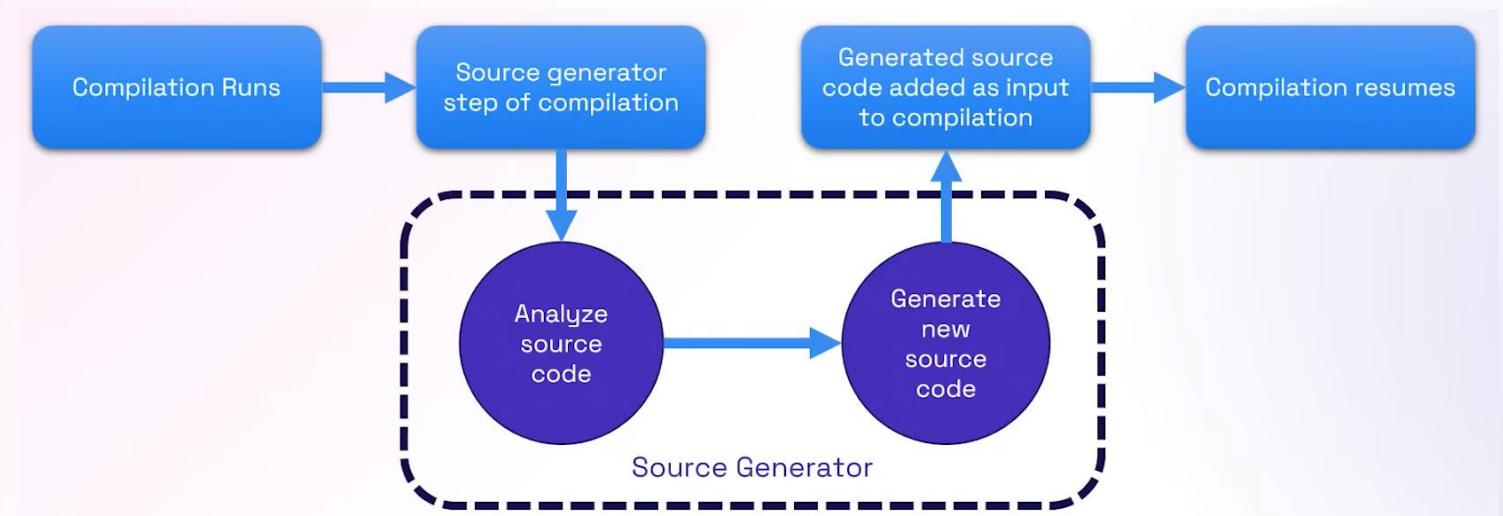
AOT.NET 8

Feature	Fully Supported	Partially Supported	Not Supported
gRPC	✓		
Minimal APIs		✓	
MVC			✗
Blazor Server			✗
SignalR			✗
Authentication			✗ (JWT soon)
CORS	✓		
HealthChecks	✓		
HttpLogging	✓		
Localization	✓		
OutputCaching	✓		
RateLimiting	✓		
RequestDecompression	✓		
ResponseCaching	✓		
ResponseCompression	✓		
Rewrite	✓		
Session			✗
Spa			✗
StaticFiles	✓		
WebSockets	✓		

#8

Source generators

- 1. Optimized Performance:** compile-time generation for efficiency.
- 2. Less Boilerplate code:** Simplified, readable, and easy to maintain.
- 3. Improved code quality** by generating code that is tailored to specific needs
- 4. Minimized risk of human errors**



#8

Source generators

The screenshot shows a code editor interface with the following details:

- Generated Regex(@"[abcxyz]")**: A tooltip or status message indicating the code was generated by a source generator.
- 2 references**: Shows the number of references to the generated code.
- No issues found**: A status message indicating there are no issues in the generated code.
- enerated] ✎ X**: A tab bar with the current tab being "enerated".
- Generated by the generator 'System.Text.RegularExpressions.Generator.RegexGenerator' and cannot be edited.**: A message explaining the nature of the generated code.
- _2023 (net8.0)**: The project name and .NET version.
- System.Text.RegularExpressions.Generated.Utilities**: The namespace of the generated code.
- s_ascii_E000007**: A variable name highlighted in blue.
- Generated Code (Visible Portions):**

```
file static class Utilities
{
    // <summary>Default timeout value set in <see cref=" ApplicationContext "/>, or <see cref=" Regex.InfiniteMatchTimeout "/>
    internal static readonly TimeSpan s_defaultTimeout = ApplicationContext.GetData("REGEX_DEFAULT_MATCH_TIMEOUT") is Time
        ? TimeSpan.FromMilliseconds(Regex.InfiniteMatchTimeout)
        : TimeSpan.FromMilliseconds(ApplicationContext.GetData("REGEX_DEFAULT_MATCH_TIMEOUT"));

    // <summary>Whether <see cref=" s_defaultTimeout "/> is non-infinite.</summary>
    internal static readonly bool s_hasTimeout = s_defaultTimeout != Regex.InfiniteMatchTimeout;

    // <summary>Supports searching for characters in or not in "abcxyz".</summary>
    internal static readonly SearchValues<char> s_ascii_E000007 = SearchValues.Create("abcxyz");
}
```
- Before Generated Code:**

```
// Before
private static readonly char[] s_values = new[] { 'a', 'b', 'c', 'x', 'y', 'z' };
...
int pos = source.IndexOfAny(s_values);
```
- After Generated Code:**

```
// After
private static readonly SearchValues s_values = SearchValues.Create("abcxyz");
...
int pos = source.IndexOfAny(s_values);
```

Efficient search for any set of bytes or chars

#8

Source generators

```
using BenchmarkDotNet.Running;

BenchmarkRunner.Run<RegexBenchmarks>();

public partial class RegexBenchmarks
{
    private const string CurrencyRegex = @"\p{Sc}+\s*\d+";
    private readonly Regex _currencyRegex = new(CurrencyRegex);
    private readonly Regex _compiledCurrencyRegex = new(CurrencyRegex, RegexOptions.Compiled);

    [GeneratedRegex(CurrencyRegex)]

    private static partial Regex NewGeneratedRegex();
    private readonly Regex _generatedCurrencyRegex = NewGeneratedRegex();

    [Params("$ 7", "7 €")]

    public string PotentialCurrency { get; set; } = default!;

    [Benchmark]
    public void Cached_Compiled_Instance()
    {
        _compiledCurrencyRegex.IsMatch(PotentialCurrency);
    }

    [Benchmark]
    public void New_Cached_Generated_Instance()
    {
        _generatedCurrencyRegex.IsMatch(PotentialCurrency);
    }
}
```

BenchmarkDotNet=v0.13.4, OS=Windows 10 (10.0.19044.3693/21H2/Nov 11th Gen Intel Core i7-1165G7 2.80GHz, 1 CPU, 8 logical and 4 physical cores).NET SDK=8.0.100

[Host] : .NET 8.0.0 (8.0.23.53103), X64 RyuJIT AVX2
DefaultJob : .NET 8.0.0 (8.0.23.53103), X64 RyuJIT AVX2

Method	PotentialCurrency	Mean
Cached_Compiled_Instance	\$ 7	55.45 ns
New_Cached_Generated_Instance	\$ 7	36.25 ns
Cached_Compiled_Instance	7 ?	51.36 ns
New_Cached_Generated_Instance	7 ?	30.62 ns

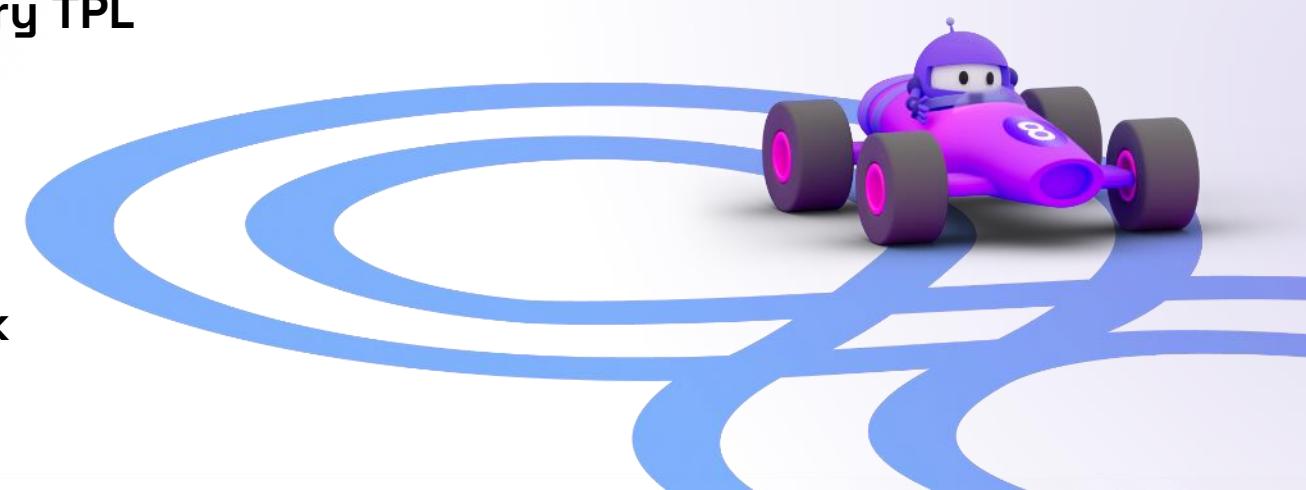
#9

Avoid memory leak

1. **Unreleased Resources:** Failing to release resources properly (such as database connections, file handles, or network sockets) can cause memory leaks. Always **dispose** of resources explicitly when they are no longer needed.
2. **Prefer StringBuilder:** over string concatenation using '+' in C# to improve performance. StringBuilder is mutable and more efficient for building strings, avoiding unnecessary overhead associated with immutable strings
3. **Event Handlers:** Subscribing to events without unsubscribing can prevent objects from being garbage collected. Make sure to **unsubscribe** from events when the object is no longer in use.
4. **Static Variables:** Static variables live for the entire lifetime of the application domain. If you store large objects in static variables, they won't be garbage collected until the application shuts down. Use static variables judiciously and consider using **dependency injection** instead.
5. **Large Objects:** Creating large objects that are held in memory for a long time can cause memory leaks. Consider using **object pooling** or other techniques to manage large objects efficiently. Optimize data structures and consider using **paging** or **lazy loading**
6. **Finalizers:** Incorrectly implemented finalizers (destructors) can delay garbage collection. Avoid using finalizers unless absolutely necessary.

Summary

- Measure, Optimize, Measure, Optimize
- Performance is contextual
- Made small changes each time and measure again
- Update framework to latest .NET version
- Deep understand framework internal (e.g. ORM, ...) for performance
- Asynchronous programming
- Streaming technique
- Parse CSV
- Task Parallel Library TPL
- AOT
- Source generator
- Avoid memory leak



References

Konrad Kokosa - Pro .NET Memory management book (2018)

Steve Gordon - Writing High-performance C# and .NET code (2022)

.NET Conf 2023

<https://learn.microsoft.com/en-us/dotnet/standard/linq/perform-streaming-transform-large-xml-documents>

<https://github.com/adamsitnik/awesome-dot-net-performance>

<https://www.thinktecture.com/en/net/dotnet-7-performance-regular-expressions/>



Questions?





THANK YOU VERY MUCH