# What’s New in .NET 7

.NET 7 is officially released and generally available for all major platforms (Windows, Linux, and macOS). This release of .NET 7 introduces many new features and continues to build on themes we introduced last year, including:

* **Performance**: .NET 7 introduces new performance gains while solidifying the unification of target platforms through investments in the Multi-platform App UI (MAUI) experience, leveraging exceptional performance and high-power efficiency on ARM64 devices and enhancements to the cloud-native developer experience, like building container images directly from the SDK without relying on third-party dependencies. .NET 7 is fast, the fastest .NET to date.
* **Simplifies choices for new developers** With the addition of C# 11 and API improvements to .NET libraries, you’re more productive than ever writing code. You can deploy your apps directly to Azure Container Apps for a distributed and scalable cloud-native experience. You can even query metadata stored in SQL JSON columns directly using Language Integrated Query (LINQ) and Entity Framework 7 while handling state-distributed across multiple microservices instances with Orleans, also known as Distributed .NET.
* **Build modern apps**: If you’re dealing with legacy codebases, you can incrementally modernize your legacy ASP.NET application to ASP.NET Core using an advanced migration experience that proxies user requests to legacy code and runs two separate websites behind the scenes, taking care of load-balancing.
* **.NET is for cloud-native apps:** .NET is great for apps built in the cloud that can run at hyper-scale. Build faster, more reliably, and easily deploy anywhere.
* **Microsoft is the best place for .NET developers**: Take a look at other articles in this issue to learn how you can build modern .NET apps with Microsoft platforms and services including Azure, Visual Studio, GitHub, and more.

.NET 7 planning began well before .NET 6 was released in November 2021. Once the release candidate milestone is reached, usually the team gathers feedback and starts the planning for the next major release.

.NET 6 proved to be the fastest .NET ever and unified various products to create a platform that runs everywhere from your mobile iOS or Android phone to Linux, macOS and, of course, Windows.

In this article, we'll explore what’s new in .NET 7. This release includes many of your requests and suggestions, so we’re excited for you to download .NET 7 and give us your feedback.

You can get started with .NET 7 in just three easy steps:

1. Download the .NET 7 SDK here <https://dotnet.microsoft.com/download/dotnet/7.0>. If you use Visual Studio or VS Code, the latest version will support with .NET 7 and the proper version will be listed on the download page. If you use another editor, please verify that it supports .NET 7.
2. Install the .NET 7 SDK.
3. Start a new project or upgrade an existing project. (Feel free to leave our team feedback on our open source repos: <https://github.com/dotnet/>).

This article is focused on the fundamentals of the release, including the runtime, libraries, and SDK. It’s these fundamental features that you interact with every day. The .NET 7 release includes new library APIs, language features, package management experiences, runtime plumbing, and SDK capabilities. This article provides a look at only a handful of improvements and new capabilities. Check out the .NET Team blog (<https://devblogs.microsoft.com/dotnet/>) to learn about the entire release.

## Targeting .NET 7

When you target a framework in an app or library, you’re specifying the set of APIs that you’d like to make available to the app or library. To target .NET 7, it’s as easy as changing the target framework in your project.

<TargetFramework>net7.0</TargetFramework>

Apps that target the net7.0 target framework moniker (TFM) will work on all supported operating systems and CPU architectures. They give you access to all the APIs in net7.0 plus a bunch of operating system-specific ones:

* net7.0-android
* net7.0-ios
* net7.0-maccatalyst
* net7.0-macos
* net7.0-tvos
* net7.0-windows

The APIs exposed through the net7.0 TFM are designed to work everywhere. If you’re ever in doubt whether an API is supported with net7.0, you can always check out <https://apisof.net/>. You can see an example of .NET 7 API support for System.IO in **Figure 1.**

Graphical user interface, text, application, chat or text message

Description automatically generated

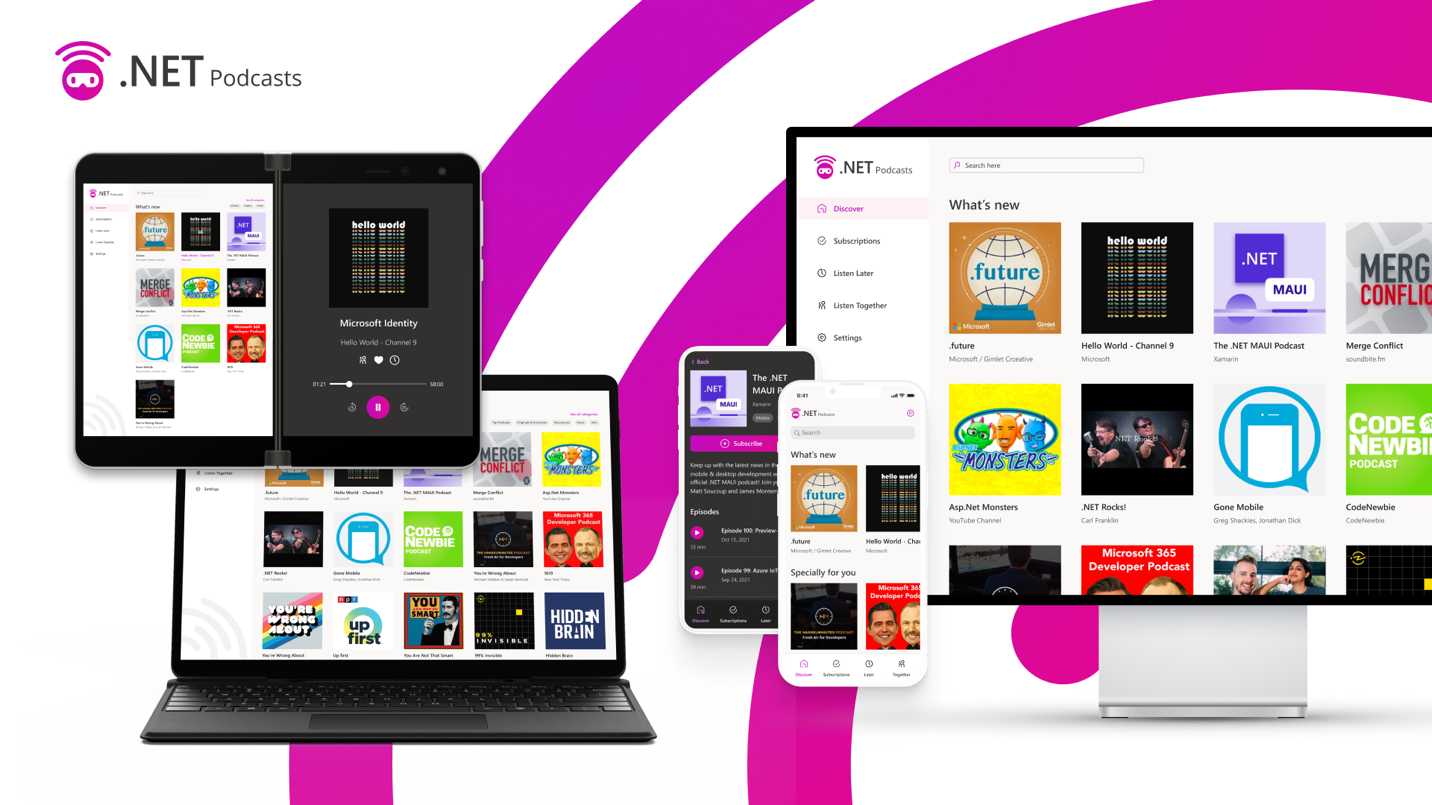
**Figure 1:** The .NET API catalog

## .NET MAUI

.NET Multi-platform App UI (MAUI) unifies Android, iOS, macOS, and Windows APIs into a single API so you can write one app that runs natively on many platforms. .NET MAUI enables you to deliver the best app experiences designed specifically by each platform (Android, iOS, macOS, Windows, and Tizen) while enabling you to craft consistent brand experience through rich styling and graphics. Out of the box, each platform looks and behaves the way it should without any additional widgets or styling required.

### New Features

Since the release of .NET MAUI, we’ve heard how much you appreciate the simplicity we’ve introduced to .NET for creating client applications from a single project. More and more libraries and services to support you are now available in .NET to help you quickly add features to your applications, such as Azure AD authentication, Bluetooth, printing, NFC, online/offline data sync, and more. In .NET 7, we bring you improved desktop features, mobile Maps, and a heavy dose of quality improvements to controls and layouts. You can see how that looks in **Figure 2.**



**Figure 2:** The .NET podcasts MAUI reference application

Whether you’re using Blazor hybrid or fully native controls with .NET MAUI for desktop, there are some native interactions you’ve told us would be very helpful. We’ve added context menus so you can reveal multi-level menus in line with your content. When the user hovers over a view and you want to display helpful text, you can now annotate your view with **TooltipProperties.Text** and a tooltip control automatically appears and disappears. You can also now add gestures for hover and right-clicking to any element using the new **PointerGesture** and we added masking properties on **TapGesture**.

Mobile applications can now take advantage of a new Map control for displaying and annotating native maps for Android, iOS, and iPadOS applications, which you can see in **Figure 3**. We’ve seen customers use maps to visualize farming scenarios, oil pipeline servicing jobs, tracking and displaying workouts, travel planning, and more. Map supports drawing shapes, placing pins, custom markers, and even geocoding street addresses, latitude, and longitude.



**Figure 3:** New Map control for displaying and annotating native maps for Android, iOS, and iPadOS applications

Maps can also be useful on the desktop. The same Apple control works for macOS, and we’re contributing a browser-based implementation of the map control for Windows to the .NET MAUI Community Toolkit.

### Performance

We’re focused on improving both your daily productivity as well as performance of your .NET MAUI applications. Gains in developer productivity, we believe, should not be at the cost of application performance. Our goal was for .NET MAUI to be faster than its predecessor, Xamarin.Forms, and it was clear that we had some work to do in .NET MAUI itself.

We improved areas like Microsoft.Extensions and DependencyInjection usage, AOT compilation, Java interop, XAML, code in .NET MAUI in general, and many more.

**Table 1** provides a performance chart of our journey thus far.

**Table 1:** Xamarin vs. .NET MAUI Startup Time Performance

| Application | Framework | Startup Time(ms) |
| --- | --- | --- |
| Xamarin.Android | Xamarin | 306.5 |
| Xamarin.Forms | Xamarin | 498.6 |
| Xamarin.Forms (Shell) | Xamarin | 817.7 |
| dotnet new android | MAUI GA | 182.8 |
| dotnet new maui (No Shell) | MAUI GA | 464.2 |
| dotnet new maui (Shell) | MAUI GA | 568.1 |
| .NET Podcast App (Shell) | MAUI GA | 814.2 |

## Cloud Native

What first started as the idea of “lifting and shifting” your applications from an on-premises environment to the cloud has now evolved into a set of best practices for building your applications for the cloud. These best practices include but aren’t limited to:

* Containerization
* CI/CD
* Orchestration and application definition
* Observability and analysis
* Service proxy, discovery, and mesh
* Networking, policy, and security
* Distributed database and storage
* Streaming and messaging
* Container registry and runtime
* Software distribution

There are three major reasons you’d go cloud native. The first is having resilience and scalability—using the cloud to reduce risk of outages and increasing availability. The second is efficiency. You can architect for the cloud to squeeze every ounce of performance and cost for your apps. The third is velocity. It’s much faster to convert your ideas to code by leveraging the vast resources of the cloud.

Next, we’ll talk about some of the major improvements building cloud native apps with .NET.

### Built-in Container Support

The popularity and practical usage of containers is rising, and for many companies, they represent the preferred way of deploying to the cloud. Working with containers adds new work to a team’s backlog, including building and publishing images, checking security and compliance, and optimizing the size and performance of said images. We believe there’s an opportunity to create a better, more streamlined experience with .NET containers.

You can now create containerized versions of your applications with just **dotnet publish**. Container images are now a supported output type of the .NET SDK:

# create a new project and move to its directory

dotnet new mvc -n my-awesome-container-app

cd my-awesome-container-app

# add a reference to a (temporary) package that creates the container

dotnet add package Microsoft.NET.Build.Containers

# publish your project for linux-x64

dotnet publish --os linux --arch x64 -p:PublishProfile=DefaultContainer

We built this solution with the following goals:

* Seamless integration with existing build logic; preventing context gaps
* Implemented in C# to take advantage of our own tooling and benefit from .NET runtime performance improvements
* Part of the .NET SDK, providing a streamlined process for updates and servicing

### Microsoft Orleans

Microsoft Orleans takes familiar concepts like objects, interfaces, async/await, and try/catch extends them to multi-server environments. It helps developers experienced with single-server applications transition to building resilient, scalable cloud services and other distributed applications. For this reason, Orleans has often been referred to as **Distributed .NET**.

Orleans was created by Microsoft Research and introduced the Virtual Actor Model as a novel approach to building a new generation of distributed systems for the Cloud era. The core contribution of Orleans is its programming model that tames the complexity inherent to highly parallel distributed systems without restricting capabilities or imposing onerous constraints on the developer.

Alongside .NET 7.0, we'll be shipping the latest update to Orleans. Orleans 4.0 gives developers better performance (as high as 50% in some lab performance tests), offers support for OpenTelemetry (<https://opentelemetry.io/>), and reduces the complexity developers face when building Orleans Grains, the distributed primitive complementing the ASP.NET object model and enabling distributed state persistence in cloud-native environments.

### Observability

The goal of observability is to help you better understand the state of your application as it scales and technical complexity increases. .NET has embraced OpenTelemetry (<https://opentelemetry.io/>) and the following improvements were made in .NET 7.

* Activity.Current change event
* Performant activity properties enumerator methods
* Performant ActivityEvent and ActivityLink tags enumerator methods

### Introducing Activity.Current Change Event

A typical implementation of distributed tracing uses an **AsyncLocal<T>** to track the “span context” of managed threads. Changes to the span context are tracked by using the **AsyncLocal<T>** constructor that takes the valueChangedHandler parameter. However, with [Activity](https://docs.microsoft.com/dotnet/api/system.diagnostics.activity) becoming the standard to represent spans as used by OpenTelemetry, it’s impossible to set the value-changed handler because the context is tracked via Activity.Current. The new change event can be used instead to receive the desired notifications.

Activity.CurrentChanged += CurrentChanged;

void CurrentChanged(object? sender,

ActivityChangedEventArgs e)

{

Console.WriteLine($"Activity.Current

value changed from Activity:

{e.Previous.OperationName} to

Activity: {e.Current.OperationName}");

}

### Expose Performant Activity Properties Enumerator Methods

The exposed methods can be used in performance-critical scenarios to enumerate the Activity Tags, Links, and Events properties without any extra allocations and with fast items access.

Activity a = new Activity("Root");

a.SetTag("key1", "value1");

a.SetTag("key2", "value2");

foreach (ref readonly

KeyValuePair<string, object?> tag in

a.EnumerateTagObjects())

{

Console.WriteLine($"{tag.Key},

{tag.Value}");

}

### Expose Performant ActivityEvent and ActivityLink Tags Enumerator Methods

The exposed methods can be used in performance-critical scenarios to enumerate the Tag objects without any extra allocations and with fast items access.

var tags = new List<KeyValuePair<string,

object?>>()

{

new KeyValuePair<string,

object?>("tag1", "value1"),

new KeyValuePair<string,

object?>("tag2", "value2"),

};

ActivityLink link = new

ActivityLink(default, new

ActivityTagsCollection(tags));

foreach (ref readonly KeyValuePair<string,

object?> tag in link.EnumerateTagObjects())

{

// Consume the link tags without any

extra allocations or value copying.

}

ActivityEvent e = new

ActivityEvent("SomeEvent", tags: new

ActivityTagsCollection(tags));

foreach (ref readonly KeyValuePair<string,

object?> tag in e.EnumerateTagObjects())

{

// Consume the event's tags without any

extra allocations or value copying.

}

## Modernization

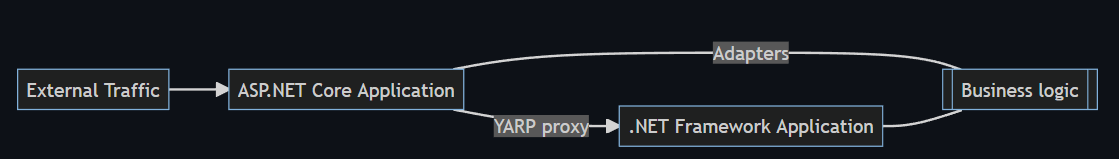
We’re focused on providing the best developer experience possible, regardless of what version of .NET you use. This includes moving off old versions to take advantage of new features. The .NET Upgrade Assistant was created to make it easier for developers to migrate legacy .NET apps to current .NET releases, including .NET 7. It supports multiple project types including:

* ASP.NET MVC
* Windows Forms
* Windows Presentation Foundation (WPF)
* Console apps
* Class libraries
* Xamarin.Forms
* Universal Windows Platform (UWP)

In addition to providing a guided step-by-step experience, the upgrade assistant now supports advanced scenarios, including bringing your UWP apps to the Windows Apps SDK (WinUI) and migrating from Xamarin to .NET MAUI.

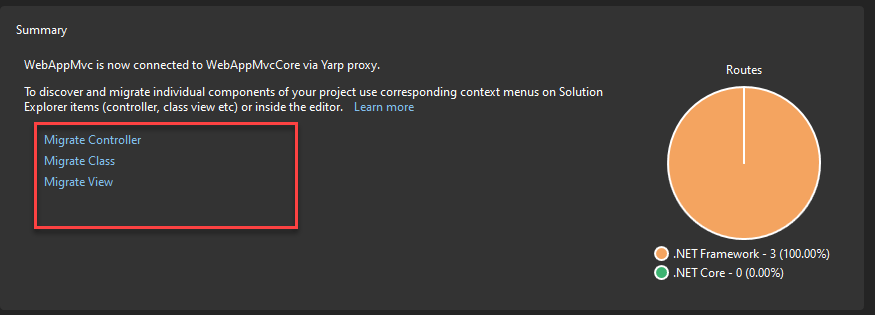
### ASP.NET to ASP.NET Core Migration

Due to the popularity and size of ASP.NET projects, there is also the option to upgrade “in place” using a special proxy. See **Figure 4** for a better understanding of how requests will flow using this proxy.



**Figure 4:** ASP.NET Core migration using a YARP proxy.

This means that you can migrate portions of your website so that it runs a hybrid combination of legacy and new code. The two versions are transparent to the end user because routes are mapped to the appropriate versions that use shared state. **Figure 5** shows how you can migrate your individual models, views, and controllers to .NET Core.



**Figure 5:** The Visual Studio ASP.NET MVC migration experience

## ARM64

We’re focused on making ARM a great platform to run .NET applications. Both x64 and ARM64 are based on different architectures (CISC vs. RISC) and each has different characteristics. The instruction set architecture (ISA) is thus different for each of them and this difference surfaces in the form of performance numbers. Although this variability exists between the two platforms, we wanted to understand how performant .NET is when running on ARM64 platforms compared to x64 and what can be done to improve its efficiency. Our continued goal is to match the parity of performance of x64 with ARM64 to help our customers move their .NET applications to ARM.

### Runtime Improvements

One challenge we had with our investigation of x64 and ARM64 was finding out that the L3 cache size wasn’t being correctly read from ARM64 machines. We changed our heuristics to return an approximate size if the L3 cache size could not be fetched from the OS or the machine’s BIOS. Now we can better approximate core counts per L3 cache sizes. See Table 2 to see a precise mapping.

**Table 2:** L3 cache size per machine core count

|  |  |
| --- | --- |
| Core count | L3 cache size |
| 1 ~ 4 | 4 MB |
| 5 ~ 16 | 8 MB |
| 17 ~ 64 | 16 MB |
| 65+ | 32 MB |

Next came our understanding of LSE atomics. Which, if you’re not familiar, provides atomic APIs to gain exclusive access to critical regions. In CISC architecture x86-x64 machines, read-modify-write (RMW) operations on memory can be performed by a single instruction by adding a lock prefix.

However, on RISC architecture machines, RMW operations are not permitted, and all operations are done through registers. Hence, for concurrency scenarios, they have pair of instructions. "Load Acquire" (ldaxr) gains exclusive access to the memory region such that no other core can access it and "Store Release" (stlxr) releases the access for other cores to access. Between these pairs, the critical operations are performed. If the stlxr operation failed because some other CPU operated on the memory after you load the contents using ldaxr, there’s a code to retry (cbnz  jumps back to retry) the operation.

ARM introduced LSE atomics instructions in v8.1. With these instructions, such operations can be done in less code and faster than the traditional version. When we enabled this for Linux and later extended it to Windows, we saw a performance win of around 45%. See **Figure 6** of LSE atomics performance enhancements on Windows for lock scenarios.

Chart, scatter chart

Description automatically generated

**Figure 6:** LSE atomics performance enhancements on Windows for lock scenarios

### Library Improvements

To optimize libraries for ARM64 using intrinsics, we added new cross-platform helpers to enable as good performance as x64. These include helpers for **Vector64, Vector128,** and **Vector256**. These vectorization algorithms are now unified by removing hardware-specific intrinsics and instead using hardware-agnostic intrinsics. This process is known as **Vectorization** in which operations are applied to whole elements instead of individual ones for performance benefits.

Rewriting APIs such as **EncodeToUtf8** and **DecodeFromUtf8** from a SSE3 implementation to a Vector-based one can provide up to 60% improvements. See **Figure 7** regarding text processing improvements with Vector-based implementations.

Chart, line chart

Description automatically generated

**Figure 7:** Text processing improvements with Vector-based implementations

Similarly converting other APIs such as **NarrowUtf16ToAscii()** and **GetIndexOfFirstNonAsciiChar()** can prove a performance win of up to 35%. See **Figure 8** regarding Span<Byte>.Reverse() improvements with Vector-based implementations.

Chart, scatter chart

Description automatically generated

**Figure 8:** Span<Byte>.Reverse() improvements with Vector-based implementations

### Performance Impact

With our work in .NET 7, many MicroBenchmarks improved by 10-60%. As we started .NET 7, the requests per second (RPS) was lower for ARM64, but slowly overcame parity of x64. See **Figure 9** for the TechEmpower benchmark.

Chart, line chart

Description automatically generated

**Figure 9:** TechEmpower RPS x64 vs. ARM64 (higher the better) benchmark

Similarly for latency (measured in milliseconds), we would exceed parity of x64. See **Figure 10** for the TechEmpower benchmark.

Chart, line chart

Description automatically generated

**Figure 10:** TechEmpower x64 vs. ARM64 latency (the lower the better)

## C# 11

The newest addition to the C# language is C# 11. C# 11 adds many features such as generic math, object initialization improvements, auto-default structs, numeric IntPtr, raw string literals, and many more. We’ll cover a few below, but invite you to read the What’s new in C# 11 documentation found at (<https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-11>).

### Generic Math

One long-requested feature in .NET is the ability to use mathematical operators on generic types. Using static abstracts in interfaces and the new interfaces now being exposed in .NET, you can now write mathematical operators on generic types of code, as shown in **Listing 1.**

This is made possible by exposing several new static abstract interfaces that correspond to the various operators available to the language and by providing a few other interfaces representing common functionality, such as parsing or handling number, integer, and floating-point types.

With Generic Math, you can take full advantage of operators and static APIs by combining static virtuals and the power of generics.

### Raw String Literals

There is now a new format for string literals. Raw string literals can contain arbitrary text, including whitespace, new lines, embedded quotes, and other special characters without requiring escape sequences. A raw string literal starts with at least three double-quote (“””) characters and ends with the same number of double-quote characters.

String longMessage = “””

This is a long message.

It has several lines.

Some are indented

more than others.

Some should start at the first column.

Some have “quoted text” in them.

“””;

### Numeric IntPtr and UintPtr

The nint and nuint types now alias [System.IntPtr](https://docs.microsoft.com/en-us/dotnet/api/system.intptr) and [System.UintPtr](https://docs.microsoft.com/en-us/dotnet/api/system.uintptr), respectively. These are two native-sized integers that depend on the platform and is computed at runtime. **Figure 11** shows the various C# types/keywords, including nint and nuint.

Graphical user interface, text, application, email

Description automatically generated

**Figure 11:** C# types/keywords. including nint and nuint

## .NET Libraries

Many of .NET’s first-party libraries have seen significant improvements in the .NET 7 release. There’s new support for nullable annotations for Microsoft.Extensions, polymorphism for System.Text.Json, new APIs for System.Composition.Hosting, adding Microseconds and Nanoseconds to date and time structures, and new Tar APIs, to name a few.

Next you can read about the many changes to .NET libraries.

### Nullable annotations for Microsoft.Extensions

All of the Microsoft.Extensions.\* libraries now contain the C# 8 opt-in feature that allows for the compiler to track reference type nullability in order to catch potential null dereferences. This helps you minimize the likelihood that your code causes the runtime to throw a **System.NullReferenceException**.

### System.Text.Json Polymorphism

System.Text.Json now supports serializing and deserializing polymorphic type hierarchies using attribute annotations:

[JsonDerivedType(typeof(Derived))]

public class Base

{

public int X { get; set; }

}

public class Derived : Base

{

public int Y { get; set; }

}

This configuration enables polymorphic serialization for **Base**, specifically when the runtime type is **Derived**:

Base value = new Derived();

JsonSerializer.Serialize<Base>(value); // { "X" :

0, "Y" : 0 }

Note that this does not enable polymorphic deserialization because the payload would be roundtripped as **Base**:

Base value = JsonSerializer.Deserialize<Base>(@"{

""X"" : 0, ""Y"" : 0 }");

value is Derived; // false

To enable polymorphic deserialization, users need to specify a type discriminator for the derived class:

[JsonDerivedType(typeof(Base), typeDiscriminator:

"base")]

[JsonDerivedType(typeof(Derived),

typeDiscriminator: "derived")]

public class Base

{

public int X { get; set; }

}

public class Derived : Base

{

public int Y { get; set; }

}

This now emits JSON along with type discriminator metadata:

Base value = new Derived();

JsonSerializer.Serialize<Base>(value); // {

"$type" : "derived", "X" : 0, "Y" : 0 }

That can be used to deserialize the value polymorphically:

Base value = JsonSerializer.Deserialize<Base>(@"{

""$type"" : ""derived"", ""X"" : 0, ""Y"" : 0 }");

value is Derived; // true

### System.Composition.Hosting

A new API has been added to allow a single object instance to the System.Composition.Hosting container providing similar functionality to the legacy interfaces as System.ComponentModel.Composition.Hosting through the API [**ComposeExportedValue(CompositionContainer, T)**](https://docs.microsoft.com/dotnet/api/system.componentmodel.composition.attributedmodelservices.composeexportedvalue?view=dotnet-plat-ext-6.0#system-componentmodel-composition-attributedmodelservices-composeexportedvalue-1(system-componentmodel-composition-hosting-compositioncontainer-0))**.**

namespace System.Composition.Hosting

{

public class ContainerConfiguration

{

public ContainerConfiguration

WithExport<TExport>(TExport

exportedInstance);

public ContainerConfiguration

WithExport<TExport>(TExport

exportedInstance, string contractName =

null, IDictionary<string, object> metadata =

null);

public ContainerConfiguration

WithExport(Type contractType, object

exportedInstance);

public ContainerConfiguration

WithExport(Type contractType, object

exportedInstance, string contractName =

null, IDictionary<string, object> metadata =

null);

}

}

### Adding Microseconds and Nanoseconds to TimeStamp, DateTime, DateTimeOffset, and TimeOnly

Before .NET 7, the lowest increment of time available in the various date and time structures was the “tick” available in the Ticks property. For reference, a single tick is 100ns. Developers have traditionally had to perform computations on the “tick” value to determine microsecond and nanosecond values. In .NET 7, we’ve introduced both microseconds and nanoseconds to the date and time implementations. **Listing 2** shows the new API structure.

### Microsoft.Extensions.Caching

We added metrics support for **IMemoryCache**, which is a new API of **MemoryCacheStatistics** that holds cache hits, misses, and estimated size for **IMemoryCache**. You can get an instance of **MemoryCacheStatistics** by calling **GetCurrentStatistics**() when the flag **TrackStatistics** is enabled.

The **GetCurrentStatistics()** API allows app developers to use event counters or metrics APIs to track statistics for one or more memory cache. **Listing 3** shows how you can get started using this API for one memory cache:

You can then view stats below with **dotnet-counters tool**:

Press p to pause, r to resume, q to quit.

Status: Running

[System.Runtime]

CPU Usage (%) 0

Working Set (MB) 28

[Microsoft-Extensions-Caching-MemoryCache]

cache-hits 269

### System.Formats.Tar APIs

We added a new **System.Formats.Tar** assembly that contains cross-platform APIs that allow reading, writing, archiving, and extracting of Tar archives. These APIs are even used by the SDK to create containers as a publishing target.

**Listing 4** provides a couple of examples of how you might use these APIs to generate and extract contents of a tar archive.

### Type Converters

There are now exposed type converters for the newly added primitive types **DateOnly,** **TimeOnly,** **Int128**, **UInt128**, and **Half**.

namespace System.ComponentModel

{

public class DateOnlyConverter :

System.ComponentModel.TypeConverter

{

public DateOnlyConverter() { }

}

Public class TimeOnlyConverter :

System.ComponentModel.TypeConverter

{

public TimeOnlyConverter() { }

}

public class Int128Converter :

System.ComponentModel.BaseNumberConverter

{

public Int128Converter() { }

}

public class UInt128Converter :

System.ComponentModel.BaseNumberConverter

{

public UInt128Converter() { }

}

public class HalfConverter :

System.ComponentModel.BaseNumberConverter

{

public HalfConverter() { }

}

}

These are helpful converters to easily convert to more primitive types.

TypeConverter dateOnlyConverter =

TypeDescriptor.GetConverter(typeof(DateOnly)

);

// produce DateOnly value of DateOnly(1940,

10, 9)

DateOnly? date =

dateOnlyConverter.ConvertFromString("1940-

10-09") as DateOnly?;

TypeConverter timeOnlyConverter =

TypeDescriptor.GetConverter(typeof(TimeOnly)

);

// produce TimeOnly value of TimeOnly(20,

30, 50)

TimeOnly? time =

timeOnlyConverter.ConvertFromString("20:30:5

0") as TimeOnly?;

TypeConverter halfConverter =

TypeDescriptor.GetConverter(typeof(Half));

// produce Half value of -1.2

Half? half =

halfConverter.ConvertFromString(((Half)(-

1.2)).ToString()) as Half?;

TypeConverter Int128Converter =

TypeDescriptor.GetConverter(typeof(Int128));

// produce Int128 value of Int128.MaxValue

which equal

170141183460469231731687303715884105727

Int128? int128 =

Int128Converter.ConvertFromString("170141183

460469231731687303715884105727") as Int128?;

TypeConverter UInt128Converter =

TypeDescriptor.GetConverter(typeof(UInt128))

;

// produce UInt128 value of UInt128.MaxValue

Which equal

340282366920938463463374607431768211455

UInt128? uint128 =

UInt128Converter.ConvertFromString("34028236

6920938463463374607431768211455") as

UInt128?;

### JSON Contract Customization

In certain situations, developers serializing or deserializing JSON find that they don’t want to or cannot change types because they either come from an external library or it would greatly pollute the code, but they may need to make some changes that influence serialization, like removing properties, changing how numbers get serialized, and how an object is created. They are frequently forced to either write wrappers or custom converters, which is not only a hassle but also makes serialization slower.

JSON contract customization gives users more control over what and how types get serialized or deserialized.

Developers can use**DefaultJsonTypeInfoResolver** and add their modifiers. All modifiers will then be called serially, like this:

JsonSerializerOptions options = new()

{

TypeInfoResolver = new

DefaultJsonTypeInfoResolver()

{

Modifiers =

{

(JsonTypeInfo jsonTypeInfo) =>

{

// your modifications here, i.e.:

if (jsonTypeInfo.Type == typeof(int))

{

jsonTypeInfo.NumberHandling =

JsonNumberHandling.AllowReadingFromString;

}

}

}

}

};

Point point =

JsonSerializer.Deserialize<Point>(@"{""X"":"

"12"",""Y"":""3""}", options);

Console.WriteLine($"({point.X},{point.Y})");

// (12,3)

public class Point

{

public int X { get; set; }

public int Y { get; set; }

}

## .NET SDK

The .NET SDK continues to add new features to make you more productive than ever. In .NET 7, we improve your experiences with the .NET CLI, authoring templates, and managing your packages in a central location.

### CLI Parser and Tab Completion

The **dotnet new** command has been given a more consistent and intuitive interface for many of the subcommands that users know and love. There’s also support for tab completion of template options and arguments. Now the CLI gives feedback on valid arguments and options as the user types.

Here’s the new help output as an example:

❯ dotnet new --help

Description:

Template Instantiation Commands for .NET

CLI.

Usage:

dotnet new [<template-short-name>

[<template-args>...]] [options]

dotnet new [command] [options]

Arguments:

<template-short-name> A short name of the

template to create.

<template-args> Template specific

options to use.

Options:

-?, -h, --help Show command line help.

Commands:

install <package> Installs a

template package.

uninstall <package> Uninstalls a

template package.

update Checks the

currently installed template packages for

update, and install the updates.

search <template-name> Searches for the

templates on NuGet.org.

list <template-name> Lists templates

containing the specified template name. If

no name is specified, lists all templates.

The **dotnet CLI** has supported tab completion for quite a while with popular shells like PowerShell, bash, zsh, and fish to name a few. It’s up to individual dotnet commands to implement meaningful completions, however. For .NET 7, the **dotnet new** command learned how to provide tab completion. You can see what that looks like in **Figure 12.**

Table

Description automatically generated with medium confidence

**Figure 12:** The dotnet new command provides tab completion.

This can be helpful for you to make choices when creating new .NET projects to know what options and arguments are available to you. You can see what that looks like in **Figure 13**.

A screenshot of a computer

Description automatically generated with medium confidence

**Figure 13:** The options and arguments that are available

And additionally, what common options and arguments are commonly mistaken or not supported for the given command. Instead, you are only shown what’s supported in the current version of the .NET CLI, as shown in **Figure 14**.



**Figure 14:** Supported options in the .NET CLI

### Template Authoring

.NET 7 adds the concept of constraints to .NET Templates. Constraints let you define the context in which your templates are allowed, which helps the template engine determine what templates it should show in commands like **dotnet new list**. For this release, we’ve added support for three kinds of constraints:

* **Operating System:** Limits templates based on the operating system of the user
* **Template Engine Host**: Limits templates based on which host is executing the template engine. This is usually the .NET CLI itself, or an embedded scenario like the New Project Dialog in Visual Studio or Visual Studio for Mac.
* **Installed Workloads**: Requires that the specified .NET SDK workload is installed before the template will become available.

In all cases, describing these constraints is as easy as adding a new **constraints** section to your template’s configuration file:

"constraints": {

"web-assembly": {

"type": "workload",

"args": "wasm-tools"

},

}

We’ve also added a new ability for **choice** parameters. This is the ability for a user to specify more than one value in a single selection. This can be used in the same way a **Flags**-style enum might be used. Common examples of this type of parameter might be:

* Opting into multiple forms of authentication on the **web** template
* Choosing multiple target platforms (iOS, Android, web) at once in the **maui** templates

Opting-in to this behavior is as simple as adding **"allowMultipleValues": true** to the parameter definition in your template’s configuration. Once you do, you’ll get access to several helper functions to use in your template’s content as well to help detect specific values that the user choses.

### Central Package Management

Dependency management is a core feature of NuGet. Managing dependencies for a single project can be easy. Managing dependencies for multi-project solutions can prove to be difficult as they start to scale in size and complexity. In situations where you manage common dependencies for many different projects, you can leverage NuGet’s central package management features to do all of this from the ease of a single location.

To get started with central package management, you can create a **Directory.Packages.props** file at the root of your solution and set the MSBuild property **ManagePackageVersionsCentrally** to **true**.

Inside, you can define each of the respective package versions required of your solution using **<PackageVersion />**elements that define the package ID and version.

<Project>

<PropertyGroup>

<ManagePackageVersionsCentrally>true</ManagePacka

geVersionsCentrally>

</PropertyGroup>

<ItemGroup>

<PackageVersion Include="Newtonsoft.Json"

Version="13.0.1" />

</ItemGroup>

</Project>

Within a project of the solution, you can then use the respective **<PackageReference />** syntax you know and love, but without a **Version** attribute to infer the centrally managed version instead.

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

<PropertyGroup>

<TargetFramework>net6.0</TargetFramework>

</PropertyGroup>

<ItemGroup>

<PackageReference Include="Newtonsoft.Json" />

</ItemGroup>

</Project>

## Performance

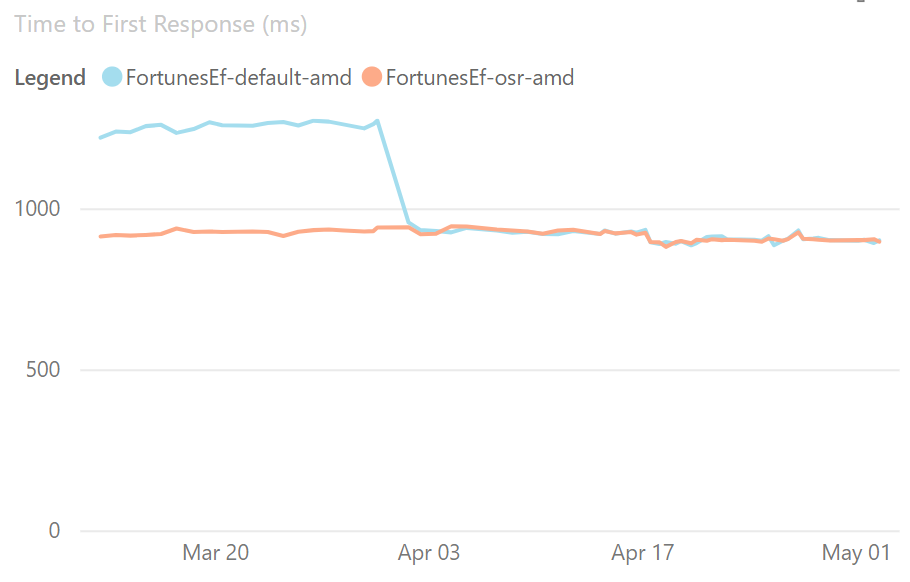
Performance has been a big part of every .NET release. Every year, the .NET team publishes a blog on the latest improvements. If you haven’t already, do check out “Performance improvements in .NET 7” post by Stephen Toub (<https://devblogs.microsoft.com/dotnet/performance_improvements_in_net_7/>). We’ll provide a short summary of some of the performance improvements to the JIT compiler from Stephen Toub’s article.

### On Stack Replacement (OSR)

On Stack Replacement (OSR) allows the runtime to change the code executed by currently running methods in the middle of method execution, although those methods are active “on stack.” It serves as a complement to tiered compilation.

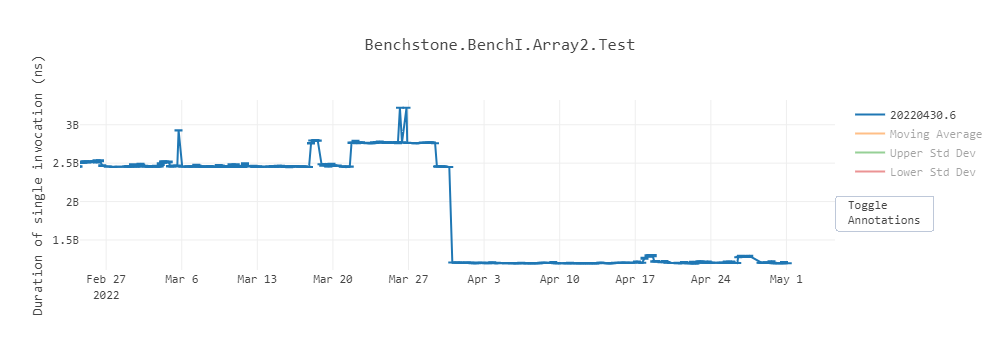
OSR allows long-running methods to switch to more optimized versions mid-execution, so the runtime can JIT all the methods quickly at first and then transition to more optimized versions when those methods are called frequently through tiered compilation or have long-running loops through OSR.

OSR improves startup time. Almost all methods are now initially jitted by the quick JIT. We have seen 25% improvement in startup time in jitting-heavy applications like Avalonia “IL” spy, and the various TechEmpower benchmarks we track show 10-30% improvements in time to first request. **Figure 15** is a chart showing when OSR was enabled by default.

[](https://devblogs.microsoft.com/dotnet/wp-content/uploads/sites/10/2022/05/166267802-37af5fca-527e-452a-8388-000f7222dd79.png)

**Figure 15:** Time to first response when OSR is enabled

OSR can also improve performance of applications, and, in particular, applications using Dynamic PGO, as methods with loops are now better optimized. For example, the **Array2** microbenchmark showed dramatic improvement when OSR was enabled. Refer to **Figure 16** to see an example of this benchmark.

[](https://devblogs.microsoft.com/dotnet/wp-content/uploads/sites/10/2022/05/166268309-420349d9-ed6a-4b5f-b4ae-fa2c664e4b50.png)

**Figure 16:** The Array2 microbenchmark with OSR enabled

### Profile-Guided Optimization (PGO)

Profile-Guided Optimization (PGO) has been around for a long time in a number of languages and compilers. The basic idea is that you compile your app, asking the compiler to inject instrumentation into the application to track various pieces of interesting information. You then put your app through its paces, running through various common scenarios, causing that instrumentation to “profile” what happens when the app is executed, and the results of that are then saved out. The app is then recompiled, feeding those instrumentation results back into the compiler, and allowing it to optimize the app for exactly how it’s expected to be used.

This approach to PGO is referred to as “static PGO,” as the information is all gleaned ahead of actual deployment, and it’s something .NET has been doing in various forms for years. The interesting development in .NET is “dynamic PGO,” which was introduced in .NET 6, but turned off by default.

Dynamic PGO takes advantage of tiered compilation. The JIT instruments the tier-0 code to track how many times the method is called, or in the case of loops, how many times the loop executes. Tiered compilation can instrument a variety of possibilities. For example, it can track exactly which concrete types are used as the target of an interface dispatch, and then, in tier-1, specialize the code to expect the most common types (this is referred to as “guarded devirtualization,” or GDV). You can see this in this little example. Set the **DOTNET\_TieredPGO** environment variable to **1**, and then run it on .NET 7:

class Program

{

static void Main()

{

IPrinter printer = new Printer();

for (int i = 0; ; i++)

{

DoWork(printer, i);

}

}

static void DoWork(IPrinter printer, int

i)

{

printer.PrintIfTrue(I == int.MaxValue);

}

interface iPrinter

{

void PrintIfTrue(bool condition);

}

class Printer : iPrinter

{

public void PrintIfTrue(bool condition)

{

if (condition)

Console.WriteLine“"Print”");

}

}

}

The tier-0 code for**DoWork** ends up looking like **Listing 5**. The tier-1 code for **DoWork**  ends up looking like **Listing 6.**

The main improvement you get with PGO is that it now works with OSR in .NET 7. This means that hot-running methods that do interface dispatch can get these devirtualization/inlining optimizations.

With PGO disabled, you get the same performance throughput for .NET 6 and .NET 7, as shown in **Table 3**.

**Table 3:** Dynamic PGO disabled in .NET 6 and .NET 7

| Method | Runtime | Mean | Ratio |
| --- | --- | --- | --- |
| DelegatePGO | .NET 6.0 | 1.665 us | 1.00 |
| DelegatePGO | .NET 7.0 | 1.659 us | 1.00 |

But the picture changes when you enable dynamic PGO in a .csproj via **<TieredPGO>true</TieredPGO> or environment variable of DOTNET\_TieredPGO=1**. .NET 6 gets ~14% faster, but .NET 7 gets ~3x faster, as shown in **Table 4**.

**Table 4:** Dynamic PGO enabled in .NET 6 and .NET 7

| Method | Runtime | Mean | Ratio |
| --- | --- | --- | --- |
| DelegatePGO | .NET 6.0 | 1,427.7 ns | 1.00 |
| DelegatePGO | .NET 7.0 | 539.0 ns | 0.38 |

### Native AOT

To many people, the word “performance” in the context of software is about throughput. How fast does something execute? How much data per second can it process? How many requests per second can it process? And so on. But there are many other facets to performance. How much memory does it consume? How fast does it start up and get to the point of doing something useful? How much space does it consume on disk? How long does it take to download?

And then there are related concerns. To achieve these goals, what dependencies are required? What kinds of operations does it need to perform to achieve these goals, and are all of those operations permitted in the target environment? If any of this paragraph resonates with you, you’re the target audience for the Native AOT support now shipping in .NET 7.

.NET has long had support for AOT code generation. For example, .NET Framework had it in the form of **ngen**, and .NET Core has it in the form of **crossgen**. Both of those solutions involve a standard .NET executable that has some of its IL already compiled to assembly code, but not all methods will have assembly code generated for them, various things can invalidate the assembly code that was generated, external .NET assemblies without any native assembly code can be loaded, and so on, and, in all those cases, the runtime continues to use a JIT compiler. Native AOT is different. It’s an evolution of CoreRT, which itself was an evolution of .NET Native, and it’s entirely free of a JIT.

The binary that results from publishing a build is a completely standalone executable in the target platform’s platform-specific file format (e.g., COFF on Windows, ELF on Linux, Mach-O on macOS) with no external dependencies other than that one is standard to that platform (e.g., libc). And it’s entirely native: no IL in sight, no JIT, no nothing. All required code is compiled and/or linked into the executable, including the same GC that’s used with standard .NET apps and services, and a minimal runtime that provides services around threading and the like.

All of that brings great benefits: super-fast startup time, small and entirely self-contained deployment, and the ability to run in places JIT compilers aren’t allowed (because memory pages that were writable can’t then be executable). It also brings limitations: No JIT means no dynamic loading of arbitrary assemblies (e.g., **Assembly.LoadFile)** and no reflection emit (e.g., **DynamicMethod**), and with everything compiled and linked into the app, that means more functionality is used (or might be used) and the larger your deployment can be. Even with those limitations, for a certain class of application, Native AOT is an incredibly exciting and welcome addition to .NET 7.

Today, Native AOT is focused on console applications, so let’s create a console app:

dotnet new console -o nativeaotexample

You now have a **“Hello World” console application.** To enable publishing the application with Native AOT, edit the .csproj to include the following in the existing **<PropertyGroup>:**

<PublishAot>true</PublishAot>

The app is now fully configured to be able to target Native AOT. All that’s left is to publish. If you wanted to publish to the Windows x64 runtime, you might use the following command:

dotnet publish -r win-x64 -c Release

This generates an executable in the output publish directory:

Directory:

C:\nativeaotexample\bin\Release\net7.0\win-

x64\publish

Mode LastWriteTime Length

Name

---- ------------- ------

----

-a--- 8/27/2022 6:18 PM 3648512

nativeaotexample.exe

-a--- 8/27/2022 6:18 PM 14290944

nativeaotexample.pdb

That ~3.5MB .exe is the executable, and the .pdb next to it is debug information, which isn’t needed deploying the app. You can now copy that **nativeaotexample.exe** to any 64-bit Windows machine, regardless of what .NET may or may not be installed anywhere on the box, and the app will run.

## Summary

As you can see above, .NET 7 includes performance gains, C# 11, improvements for the runtime, library, Native AOT, MAUI/Blazor enhancements and more.

It's a huge release that improves your .NET developer quality of life by improving fundamentals like performance, functionality, and usability. Our goal with .NET is to empower you to build any application, anywhere. Whether that’s mobile applications with .NET MAUI, high-power efficient apps on ARM64 devices, or best-in-class cloud native apps, .NET 7 has got you covered.

Download .NET 7 by visiting <https://dotnet.microsoft.com/download> and get started today building your first .NET 7 app!

Get .NET 7 Help for Free

How does a FREE hour-long CODE Consulting virtual meeting with our expert .NET consultants sound? Yes, FREE. No strings. No commitment. No credit cards. Nothing to buy. For more information, visit [www.codemag.com/consulting](http://www.codemag.com/consulting) or email us at [info@codemag.com](mailto:info@codemag.com).

Jon Douglas

@\_jondouglas

[jonathan.douglas@microsoft.com](mailto:jonathan.douglas@microsoft.com)

<https://devblogs.microsoft.com/dotnet/author/jondouglas/>

Jon Douglas is a Principal Product Manager for NuGet and .NET at Microsoft.

Jeremy Likness

@JeremyLikness

[jeremy.likness@microsoft.com](mailto:jeremy.likness@microsoft.com)

<https://devblogs.microsoft.com/dotnet/author/jeremy-likness/>

Jeremy Likness is a Principal Product Manager at Microsoft focused on .NET Web Frameworks.

Angelos Petropoulos

@apetrop

[angelos.petropoulos@microsoft.com](mailto:angelos.petropoulos@microsoft.com)

<https://devblogs.microsoft.com/dotnet/author/angelos-petropoulos/>

Angelos Petropoulos is a Principal Product Manager at Microsoft working on .NET, Azure, and Visual Studio.

Listing 1: Mathematical operators on generic types

public static TResult Sum<T, TResult>(IEnumerable<T> values)

where T : INumber<T>

where TResult : INumber<TResult>

{

TResult result = TResult.Zero;

foreach (var value in values)

{

result += TResult.Create(value);

}

return result;

}

public static TResult Average<T, TResult>(IEnumerable<T> values)

where T : INumber<T>

where TResult : INumber<TResult>

{

TResult sum = Sum<T, TResult>(values);

return TResult.Create(sum) / TResult.Create(values.Count());

}

public static TResult StandardDeviation<T, TResult>(IEnumerable<T> values)

where T : INumber<T>

where TResult : IFloatingPoint<TResult>

{

TResult standardDeviation = TResult.Zero;

if (values.Any())

{

TResult average = Average<T, TResult>(values);

TResult sum = Sum<TResult, TResult>(values.Select((value) => {

var deviation = TResult.Create(value) - average;

return deviation \* deviation;

}));

standardDeviation = TResult.Sqrt(sum

/ TResult.Create(values.Count() - 1));

}

return standardDeviation;

}

Listing 2: Microseconds and Nanoseconds in DateTime, DateTimeOffset, and TimeOnly

namespace System {

public struct DateTime {

public DateTime(int year, int month,

int day, int hour, int minute, int second,

int millisecond, int microsecond);

public DateTime(int year, int month,

int day, int hour, int minute, int second,

int millisecond, int microsecond,

System.DateTimeKind kind);

public DateTime(int year, int month,

int day, int hour, int minute, int second,

int millisecond, int microsecond,

System.Globalization.Calendar calendar);

public int Microsecond { get; }

public int Nanosecond { get; }

public DateTime

AddMicroseconds(double value);

}

public struct DateTimeOffset {

public DateTimeOffset(int year, int

month, int day, int hour, int minute, int

second, int millisecond, int microsecond,

System.TimeSpan offset);

public DateTimeOffset(int year, int

month, int day, int hour, int minute, int

second, int millisecond, int microsecond,

System.TimeSpan offset,

System.Globalization.Calendar calendar);

public int Microsecond { get; }

public int Nanosecond { get; }

public DateTimeOffset

AddMicroseconds(double microseconds);

}

public struct TimeSpan {

public const long

TicksPerMicrosecond = 10L;

public const long NanosecondsPerTick

= 100L;

public TimeSpan(int days, int hours,

int minutes, int seconds, int milliseconds,

int microseconds);

public int Microseconds { get; }

public int Nanoseconds { get; }

public double TotalMicroseconds {

get; }

public double TotalNanoseconds {

get; }

public static TimeSpan

FromMicroseconds(double microseconds);

}

public struct TimeOnly {

public TimeOnly(int hour, int

minute, int second, int millisecond, int

microsecond);

public int Microsecond { get; }

public int Nanosecond { get; }

}

}

Listing 3: Get started with MemoryCacheStatistics

// when using

`services.AddMemoryCache(options =>

options.TrackStatistics = true);` to

instantiate

[EventSource(Name = "Microsoft-Extensions-Caching-Memory")]

internal sealed class CachingEventSource

: EventSource

{

public CachingEventSource(IMemoryCache memoryCache)

{ \_memoryCache = memoryCache; }

protected override void

OnEventCommand(EventCommandEventArgs

command)

{

if (command.Command ==

EventCommand.Enable)

{

if (\_cacheHitsCounter ==

null)

{

\_cacheHitsCounter = new

PollingCounter("cache-hits", this, () =>

\_memoryCache.GetCurrentStatistics().CacheHits)

{

DisplayName = "Cache

hits",

};

}

}

}

}

Listing 4: Using System.Formats.Tar APIs

// Generates a tar archive where all the

entry names are prefixed by the root

directory 'SourceDirectory'

TarFile.CreateFromDirectory(sourceDirectoryN

ame: "/home/dotnet/SourceDirectory/",

destinationFileName:

"/home/dotnet/destination.tar",

includeBaseDirectory: true);

// Extracts the contents of a tar archive

into the specified directory, but avoids

overwriting anything found inside

TarFile.ExtractToDirectory(sourceFileName:

"/home/dotnet/destination.tar",

destinationDirectoryName:

"/home/dotnet/DestinationDirectory/",

overwriteFiles: false);

Listing 5: Tier-0 code for DoWork

G\_M000\_IG01: ;; offset=0000H

55 push rbp

4883EC30 sub rsp, 48

488D6C2430 lea rbp, [rsp+30H]

33C0 xor eax, eax

488945F8 mov qword ptr [rbp-08H], rax

488945F0 mov qword ptr [rbp-10H], rax

48894D10 mov gword ptr [rbp+10H], rcx

895518 mov dword ptr [rbp+18H], edx

G\_M000\_IG02: ;; offset=001BH

FF059F220F00 inc dword ptr [(reloc 0x7ffc3f1b2ea0)]

488B4D10 mov rcx, gword ptr [rbp+10H]

48894DF8 mov gword ptr [rbp-08H], rcx

488B4DF8 mov rcx, gword ptr [rbp-08H]

48BAA82E1B3FFC7F0000 mov rdx, 0x7FFC3F1B2EA8

E8B47EC55F call CORINFO\_HELP\_CLASSPROFILE32

488B4DF8 mov rcx, gword ptr [rbp-08H]

48894DF0 mov gword ptr [rbp-10H], rcx

488B4DF0 mov rcx, gword ptr [rbp-10H]

33D2 xor edx, edx

817D18FFFFFF7F cmp dword ptr [rbp+18H], 0x7FFFFFFF

0F94C2 sete dl

49BB0800F13EFC7F0000 mov r11, 0x7FFC3EF10008

41FF13 call [r11]iPrinter:PrintIfTrue(bool):this

90 nop

G\_M000\_IG03: ;; offset=0062H

4883C430 add rsp, 48

5D pop rbp

C3 ret

Listing 6: Tier-1 code for DoWork

G\_M000\_IG02: ;; offset=0020H

48B9982D1B3FFC7F0000 mov rcx,

0x7FFC3F1B2D98

48390F cmp qword

ptr [rdi], rcx

7521 jne SHORT

G\_M000\_IG05

81FEFFFFFF7F cmp esi,

0x7FFFFFFF

7404 je SHORT

G\_M000\_IG04

G\_M000\_IG03: ;; offset=0037H

FFC6 inc esi

EBE5 jmp SHORT

G\_M000\_IG02

G\_M000\_IG04: ;; offset=003BH

48B9D820801A24020000 mov rcx,

0x2241A8020D8

488B09 mov rcx,

gword ptr [rcx]

FF1572CD0D00 call

[Console:WriteLine(String)]

EBE7 jmp SHORT

G\_M000\_IG03