What's new in the .NET 8 runtime

05/07/2024

This article describes new features in the .NET runtime for .NET 8.

Performance improvements

.NET 8 includes improvements to code generation and just-in time (JIT) compilation:

- Arm64 performance improvements
- SIMD improvements
- Support for AVX-512 ISA extensions (see Vector512 and AVX-512)
- Cloud-native improvements
- JIT throughput improvements
- Loop and general optimizations
- Optimized access for fields marked with ThreadStaticAttribute
- Consecutive register allocation. Arm64 has two instructions for table vector lookup, which require that all entities in their tuple operands are present in consecutive registers.
- JIT/NativeAOT can now unroll and auto-vectorize some memory operations with SIMD, such as comparison, copying, and zeroing, if it can determine their sizes at compile time.

In addition, dynamic profile-guided optimization (PGO) has been improved and is now enabled by default. You no longer need to use a runtime configuration option to enable it. Dynamic PGO works hand-in-hand with tiered compilation to further optimize code based on additional instrumentation that's put in place during tier 0.

On average, dynamic PGO increases performance by about 15%. In a benchmark suite of ~4600 tests, 23% saw performance improvements of 20% or more.

Codegen struct promotion

.NET 8 includes a new physical promotion optimization pass for codegen that generalizes the JIT's ability to promote struct variables. This optimization (also called *scalar replacement of aggregates*) replaces the fields of struct variables by primitive variables that the JIT is then able to reason about and optimize more precisely.

The JIT already supported this optimization but with several large limitations including:

- It was only supported for structs with four or fewer fields.

- It was only supported if each field was a primitive type, or a simple struct wrapping a primitive type.

Physical promotion removes these limitations, which fixes a number of long-standing JIT issues.

Garbage collection

.NET 8 adds a capability to adjust the memory limit on the fly. This is useful in cloud-service scenarios, where demand comes and goes. To be cost-effective, services should scale up and down on resource consumption as the demand fluctuates. When a service detects a decrease in demand, it can scale down resource consumption by reducing its memory limit. Previously, this would fail because the garbage collector (GC) was unaware of the change and might allocate more memory than the new limit. With this change, you can call the RefreshMemoryLimit() API to update the GC with the new memory limit.

There are some limitations to be aware of:

- On 32-bit platforms (for example, Windows x86 and Linux ARM), .NET is unable to establish a new heap hard limit if there isn't already one.
- The API might return a non-zero status code indicating the refresh failed. This can happen if the scale-down is too aggressive and leaves no room for the GC to maneuver. In this case, consider calling `GC.Collect(2, GCCollectionMode.Aggressive)` to shrink the current memory usage, and then try again.
- If you scale up the memory limit beyond the size that the GC believes the process can handle during startup, the `RefreshMemoryLimit` call will succeed, but it won't be able to use more memory than what it perceives as the limit.

The following code snippet shows how to call the API.

C#

```
GC.RefreshMemoryLimit();
```

You can also refresh some of the GC configuration settings related to the memory limit. The following code snippet sets the heap hard limit to 100 mebibytes (MiB):

C#

```
AppContext.SetData("GCHeapHardLimit", (ulong)100 * 1_024 * 1_024);
```

```
GC.RefreshMemoryLimit();
```

The API can throw an InvalidOperationException if the hard limit is invalid, for example, in the case of negative heap hard limit percentages and if the hard limit is too low. This can happen if the heap hard limit that the refresh will set, either because of new AppData settings or implied by the container memory limit changes, is lower than what's already committed.

Globalization for mobile apps

Mobile apps on iOS, tvOS, and MacCatalyst can opt in to a new *hybrid* globalization mode that uses a lighter ICU bundle. In hybrid mode, globalization data is partially pulled from the ICU bundle and partially from calls into Native APIs. Hybrid mode serves all the locales supported by mobile .

Hybrid mode is most suitable for apps that can't work in invariant globalization mode and that use cultures that were trimmed from ICU data on mobile. You can also use it when you want to load a smaller ICU data file. (The *icudt_hybrid.dat* file is 34.5 % smaller than the default ICU data file *icudt.dat*.)

To use hybrid globalization mode, set the `HybridGlobalization` MSBuild property to true:

XML

```
<PropertyGroup>
  <HybridGlobalization>true</HybridGlobalization>
</PropertyGroup>
```

There are some limitations to be aware of:

- Due to limitations of Native API, not all globalization APIs are supported in hybrid mode.
- Some of the supported APIs have different behavior.

To check if your application is affected, see Behavioral differences .

Source-generated COM interop

.NET 8 includes a new source generator that supports interoperating with COM interfaces. You can use the GeneratedComInterfaceAttribute to mark an interface as a COM interface for the source generator. The source generator will then generate code to enable calling from C# code to

unmanaged code. It also generates code to enable calling from unmanaged code into C#. This source generator integrates with LibraryImportAttribute, and you can use types with the GeneratedComInterfaceAttribute as parameters and return types in `LibraryImport`-attributed methods.

C#

```
using System.Runtime.InteropServices;
using System.Runtime.InteropServices.Marshalling;

[GeneratedComInterface]
[Guid("5401c312-ab23-4dd3-aa40-3cb4b3a4683e")]
partial interface IComInterface
{
    void DoWork();
}

internal partial class MyNativeLib
{
    [LibraryImport(nameof(MyNativeLib))]
    public static partial void GetComInterface(out IComInterface comInterface);
}
```

The source generator also supports the new GeneratedComClassAttribute attribute to enable you to pass types that implement interfaces with the GeneratedComInterfaceAttribute attribute to unmanaged code. The source generator will generate the code necessary to expose a COM object that implements the interfaces and forwards calls to the managed implementation.

Methods on interfaces with the GeneratedComInterfaceAttribute attribute support all the same types as `LibraryImportAttribute`, and `LibraryImportAttribute` now supports `GeneratedComInterface`-attributed types and `GeneratedComClass`-attributed types.

If your C# code only uses a `GeneratedComInterface`-attributed interface to either wrap a COM object from unmanaged code or wrap a managed object from C# to expose to unmanaged code, you can use the options in the Options property to customize which code will be generated. These options mean you don't need to write marshallers for scenarios that you know won't be used.

The source generator uses the new StrategyBasedComWrappers type to create and manage the COM object wrappers and the managed object wrappers. This new type handles providing the expected .NET user experience for COM interop, while providing customization points for advanced users. If your application has its own mechanism for defining types from COM or if you need to support scenarios that source-generated COM doesn't currently support, consider using

the new StrategyBasedComWrappers type to add the missing features for your scenario and get the same .NET user experience for your COM types.

If you're using Visual Studio, new analyzers and code fixes make it easy to convert your existing COM interop code to use source-generated interop. Next to each interface that has the ComImportAttribute, a lightbulb offers an option to convert to source-generated interop. The fix changes the interface to use the GeneratedComInterfaceAttribute attribute. And next to every class that implements an interface with `GeneratedComInterfaceAttribute`, a lightbulb offers an option to add the GeneratedComClassAttribute attribute to the type. Once your types are converted, you can move your `DllImport` methods to use `LibraryImportAttribute`.

Limitations

The COM source generator doesn't support apartment affinity, using the `new` keyword to activate a COM CoClass, and the following APIs:

- IDispatch-based interfaces.
- IInspectable-based interfaces.
- COM properties and events.

Configuration-binding source generator

.NET 8 introduces a source generator to provide AOT and trim-friendly configuration in ASP.NET Core. The generator is an alternative to the pre-existing reflection-based implementation.

The source generator probes for Configure(TOptions), Bind, and Get calls to retrieve type info from. When the generator is enabled in a project, the compiler implicitly chooses generated methods over the pre-existing reflection-based framework implementations.

No source code changes are needed to use the generator. It's enabled by default in AOT-compiled web apps, and when PublishTrimmed is set to `true` (.NET 8+ apps). For other project types, the source generator is off by default, but you can opt in by setting the `EnableConfigurationBindingGenerator` property to `true` in your project file:

XML

```
<PropertyGroup>
    <EnableConfigurationBindingGenerator>true</EnableConfigurationBindingGenerator>
</PropertyGroup>
```

The following code shows an example of invoking the binder.

C#

```
public class ConfigBindingSG
{
    static void RunIt(params string[] args)
    {
        WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
        IConfigurationSection section =
builder.Configuration.GetSection("MyOptions");

        // !! Configure call - to be replaced with source-gen'd implementation
        builder.Services.Configure<MyOptions>(section);

        // !! Get call - to be replaced with source-gen'd implementation
        MyOptions? options0 = section.Get<MyOptions>();

        // !! Bind call - to be replaced with source-gen'd implementation
        MyOptions options1 = new();
        section.Bind(options1);

        WebApplication app = builder.Build();
        app.MapGet("/", () => "Hello World!");
        app.Run();
    }

    public class MyOptions
    {
        public int A { get; set; }
        public string S { get; set; }
        public byte[] Data { get; set; }
        public Dictionary<string, string> Values { get; set; }
        public List<MyClass> Values2 { get; set; }
    }

    public class MyClass
    {
        public int SomethingElse { get; set; }
    }
}
```

Core .NET libraries

This section contains the following subtopics:

- Reflection

- Serialization
- Time abstraction
- UTF8 improvements
- Methods for working with randomness
- Performance-focused types
- System.Numerics and System.Runtime.Intrinsics
- Data validation
- Metrics
- Cryptography
- Networking
- Stream-based ZipFile methods

Reflection

Function pointers were introduced in .NET 5, however, the corresponding support for reflection wasn't added at that time. When using `typeof` or reflection on a function pointer, for example, `typeof(delegate*<void>())` or `FieldInfo.FieldType` respectively, an IntPtr was returned. Starting in .NET 8, a System.Type object is returned instead. This type provides access to function pointer metadata, including the calling conventions, return type, and parameters.

> **Note**
>
> A function pointer instance, which is a physical address to a function, continues to be represented as an **IntPtr**. Only the reflection type has changed.

The new functionality is currently implemented only in the CoreCLR runtime and MetadataLoadContext.

New APIs have been added to System.Type, such as IsFunctionPointer, and to System.Reflection.PropertyInfo, System.Reflection.FieldInfo, and System.Reflection.ParameterInfo. The following code shows how to use some of the new APIs for reflection.

C#

```
using System;
using System.Reflection;

// Sample class that contains a function pointer field.
public unsafe class UClass
{
```

```
    public delegate* unmanaged[Cdecl, SuppressGCTransition]<in int, void> _fp;
}

internal class FunctionPointerReflection
{
    public static void RunIt()
    {
        FieldInfo? fieldInfo = typeof(UClass).GetField(nameof(UClass._fp));

        // Obtain the function pointer type from a field.
        Type? fpType = fieldInfo?.FieldType;

        // New methods to determine if a type is a function pointer.
        Console.WriteLine(
        $"IsFunctionPointer: {fpType?.IsFunctionPointer}");
        Console.WriteLine(
            $"IsUnmanagedFunctionPointer: {fpType?.IsUnmanagedFunctionPointer}");

        // New methods to obtain the return and parameter types.
        Console.WriteLine($"Return type: {fpType?.GetFunctionPointerReturnType()}");

        if (fpType is not null)
        {
            foreach (Type parameterType in fpType.GetFunctionPointerParameterTypes())
            {
                Console.WriteLine($"Parameter type: {parameterType}");
            }
        }

        // Access to custom modifiers and calling conventions requires a "modified
type".
        Type? modifiedType = fieldInfo?.GetModifiedFieldType();

        // A modified type forwards most members to its underlying type.
        Type? normalType = modifiedType?.UnderlyingSystemType;

        if (modifiedType is not null)
        {
            // New method to obtain the calling conventions.
            foreach (Type callConv in modifiedType.GetFunctionPointerCallingConven-
tions())
            {
                Console.WriteLine($"Calling convention: {callConv}");
            }
        }

        // New method to obtain the custom modifiers.
        Type[]? modifiers =
            modifiedType?.GetFunctionPointerParameterTypes()[0].GetRequiredCustomMod-
ifiers();
```

```
        if (modifiers is not null)
        {
            foreach (Type modreq in modifiers)
            {
                Console.WriteLine($"Required modifier for first parameter:
{modreq}");
            }
        }
    }
}
```

The previous example produces the following output:

Output

```
IsFunctionPointer: True
IsUnmanagedFunctionPointer: True
Return type: System.Void
Parameter type: System.Int32&
Calling convention: System.Runtime.CompilerServices.CallConvSuppressGCTransition
Calling convention: System.Runtime.CompilerServices.CallConvCdecl
Required modifier for first parameter: System.Runtime.InteropServices.InAttribute
```

Serialization

Many improvements have been made to System.Text.Json serialization and deserialization functionality in .NET 8. For example, you can customize handling of JSON properties that aren't in the POCO.

The following sections describe other serialization improvements:

- Built-in support for additional types
- Source generator
- Interface hierarchies
- Naming policies
- Read-only properties
- Disable reflection-based default
- New JsonNode API methods
- Non-public members
- Streaming deserialization APIs
- WithAddedModifier extension method
- New JsonContent.Create overloads

- Freeze a JsonSerializerOptions instance

For more information about JSON serialization in general, see JSON serialization and deserialization in .NET.

Built-in support for additional types

The serializer has built-in support for the following additional types.

- Half, Int128, and UInt128 numeric types.

```
C#
```

```
Console.WriteLine(JsonSerializer.Serialize(
    [ Half.MaxValue, Int128.MaxValue, UInt128.MaxValue ]
));
//
[65500,170141183460469231731687303715884105727,340282366920938463463374607431768
211455]
```

- Memory<T> and ReadOnlyMemory<T> values. `byte` values are serialized to Base64 strings, and other types to JSON arrays.

```
C#
```

```
JsonSerializer.Serialize<ReadOnlyMemory<byte>>(new byte[] { 1, 2, 3 }); //
"AQID"
JsonSerializer.Serialize<Memory<int>>(new int[] { 1, 2, 3 }); // [1,2,3]
```

Source generator

.NET 8 includes enhancements of the System.Text.Json source generator that are aimed at making the Native AOT experience on par with the reflection-based serializer. For example:

- The source generator now supports serializing types with required and init properties. These were both already supported in reflection-based serialization.
- Improved formatting of source-generated code.
- JsonSourceGenerationOptionsAttribute feature parity with JsonSerializerOptions. For more information, see Specify options (source generation).

- Additional diagnostics (such as SYSLIB1034 and SYSLIB1039).
- Don't include types of ignored or inaccessible properties.
- Support for nesting `JsonSerializerContext` declarations within arbitrary type kinds.
- Support for compiler-generated or *unspeakable* types in weakly typed source generation scenarios. Since compiler-generated types can't be explicitly specified by the source generator, System.Text.Json now performs nearest-ancestor resolution at run time. This resolution determines the most appropriate supertype with which to serialize the value.
- New converter type `JsonStringEnumConverter<TEnum>`. The existing JsonStringEnumConverter class isn't supported in Native AOT. You can annotate your enum types as follows:

  C#

  ```
  [JsonConverter(typeof(JsonStringEnumConverter<MyEnum>))]
  public enum MyEnum { Value1, Value2, Value3 }

  [JsonSerializable(typeof(MyEnum))]
  public partial class MyContext : JsonSerializerContext { }
  ```

  For more information, see Serialize enum fields as strings.

- New `JsonConverter.Type` property lets you look up the type of a non-generic `JsonConverter` instance:

  C#

  ```
  Dictionary<Type, JsonConverter> CreateDictionary(IEnumerable<JsonConverter> con-
  verters)
      => converters.Where(converter => converter.Type != null)
                   .ToDictionary(converter => converter.Type!);
  ```

  The property is nullable since it returns `null` for `JsonConverterFactory` instances and `typeof(T)` for `JsonConverter<T>` instances.

Chain source generators

The JsonSerializerOptions class includes a new TypeInfoResolverChain property that complements the existing TypeInfoResolver property. These properties are used in contract customization for chaining source generators. The addition of the new property means that you don't have to

specify all chained components at one call site—they can be added after the fact. TypeInfoResolverChain also lets you introspect the chain or remove components from it. For more information, see Combine source generators.

In addition, JsonSerializerOptions.AddContext<TContext>() is now obsolete. It's been superseded by the TypeInfoResolver and TypeInfoResolverChain properties. For more information, see SYSLIB0049.

Interface hierarchies

.NET 8 adds support for serializing properties from interface hierarchies.

The following code shows an example where the properties from both the immediately implemented interface and its base interface are serialized.

C#

```
public static void InterfaceHierarchies()
{
    IDerived value = new DerivedImplement { Base = 0, Derived = 1 };
    string json = JsonSerializer.Serialize(value);
    Console.WriteLine(json); // {"Derived":1,"Base":0}
}

public interface IBase
{
    public int Base { get; set; }
}

public interface IDerived : IBase
{
    public int Derived { get; set; }
}

public class DerivedImplement : IDerived
{
    public int Base { get; set; }
    public int Derived { get; set; }
}
```

Naming policies

JsonNamingPolicy includes new naming policies for `snake_case` (with an underscore) and `kebab-case` (with a hyphen) property name conversions. Use these policies similarly to the existing

JsonNamingPolicy.CamelCase policy:

C#

```
var options = new JsonSerializerOptions
{
    PropertyNamingPolicy = JsonNamingPolicy.SnakeCaseLower
};
JsonSerializer.Serialize(new { PropertyName = "value" }, options);
// { "property_name" : "value" }
```

For more information, see Use a built-in naming policy.

Read-only properties

You can now deserialize onto read-only fields or properties (that is, those that don't have a `set` accessor).

To opt in to this support globally, set a new option, PreferredObjectCreationHandling, to JsonObjectCreationHandling.Populate. If compatibility is a concern, you can also enable the functionality more granularly by placing the `[JsonObjectCreationHandling(JsonObjectCreationHandling.Populate)]` attribute on specific types whose properties are to be populated, or on individual properties.

For example, consider the following code that deserializes into a `CustomerInfo` type that has two read-only properties.

C#

```
public static void ReadOnlyProperties()
{
    CustomerInfo customer = JsonSerializer.Deserialize<CustomerInfo>("""
        { "Names":["John Doe"], "Company":{"Name":"Contoso"} }
        """)!;

    Console.WriteLine(JsonSerializer.Serialize(customer));
}

class CompanyInfo
{
    public required string Name { get; set; }
    public string? PhoneNumber { get; set; }
}
```

```
[JsonObjectCreationHandling(JsonObjectCreationHandling.Populate)]
class CustomerInfo
{
    // Both of these properties are read-only.
    public List<string> Names { get; } = new();
    public CompanyInfo Company { get; } = new()
    {
        Name = "N/A",
        PhoneNumber = "N/A"
    };
}
```

Prior to .NET 8, the input values were ignored and the `Names` and `Company` properties retained their default values.

Output

```
{"Names":[],"Company":{"Name":"N/A","PhoneNumber":"N/A"}}
```

Now, the input values are used to populate the read-only properties during deserialization.

Output

```
{"Names":["John Doe"],"Company":{"Name":"Contoso","PhoneNumber":"N/A"}}
```

For more information about the *populate* deserialization behavior, see Populate initialized properties.

Disable reflection-based default

You can now disable using the reflection-based serializer by default. This disablement is useful to avoid accidental rooting of reflection components that aren't even in use, especially in trimmed and Native AOT apps. To disable default reflection-based serialization by requiring that a JsonSerializerOptions argument be passed to the JsonSerializer serialization and deserialization methods, set the `JsonSerializerIsReflectionEnabledByDefault` MSBuild property to `false` in your project file.

Use the new IsReflectionEnabledByDefault API to check the value of the feature switch. If you're a library author building on top of System.Text.Json, you can rely on the property to configure your defaults without accidentally rooting reflection components.

For more information, see Disable reflection defaults.

New JsonNode API methods

The JsonNode and System.Text.Json.Nodes.JsonArray types include the following new methods.

C#

```
public partial class JsonNode
{
    // Creates a deep clone of the current node and all its descendants.
    public JsonNode DeepClone();

    // Returns true if the two nodes are equivalent JSON representations.
    public static bool DeepEquals(JsonNode? node1, JsonNode? node2);

    // Determines the JsonValueKind of the current node.
    public JsonValueKind GetValueKind(JsonSerializerOptions options = null);

    // If node is the value of a property in the parent
    // object, returns its name.
    // Throws InvalidOperationException otherwise.
    public string GetPropertyName();

    // If node is the element of a parent JsonArray,
    // returns its index.
    // Throws InvalidOperationException otherwise.
    public int GetElementIndex();

    // Replaces this instance with a new value,
    // updating the parent object/array accordingly.
    public void ReplaceWith<T>(T value);

    // Asynchronously parses a stream as UTF-8 encoded data
    // representing a single JSON value into a JsonNode.
    public static Task<JsonNode?> ParseAsync(
        Stream utf8Json,
        JsonNodeOptions? nodeOptions = null,
        JsonDocumentOptions documentOptions = default,
        CancellationToken cancellationToken = default);
}

public partial class JsonArray
{
    // Returns an IEnumerable<T> view of the current array.
    public IEnumerable<T> GetValues<T>();
}
```

Non-public members

You can opt non-public members into the serialization contract for a given type using JsonIncludeAttribute and JsonConstructorAttribute attribute annotations.

C#

```
public static void NonPublicMembers()
{
    string json = JsonSerializer.Serialize(new MyPoco(42));
    Console.WriteLine(json);
    // {"X":42}

    JsonSerializer.Deserialize<MyPoco>(json);
}

public class MyPoco
{
    [JsonConstructor]
    internal MyPoco(int x) => X = x;

    [JsonInclude]
    internal int X { get; }
}
```

For more information, see Use immutable types and non-public members and accessors.

Streaming deserialization APIs

.NET 8 includes new IAsyncEnumerable<T> streaming deserialization extension methods, for example GetFromJsonAsAsyncEnumerable. Similar methods have existed that return Task<TResult>, for example, HttpClientJsonExtensions.GetFromJsonAsync. The new extension methods invoke streaming APIs and return IAsyncEnumerable<T>.

The following code shows how you might use the new extension methods.

C#

```
public async static void StreamingDeserialization()
{
    const string RequestUri = "https://api.contoso.com/books";
    using var client = new HttpClient();
    IAsyncEnumerable<Book?> books = client.GetFromJsonAsAsyncEnumerable<Book>(Reques-
tUri);

    await foreach (Book? book in books)
    {
        Console.WriteLine($"Read book '{book?.title}'");
```

```
    }
}

public record Book(int id, string title, string author, int publishedYear);
```

WithAddedModifier extension method

The new WithAddedModifier(IJsonTypeInfoResolver, Action<JsonTypeInfo>) extension method lets you easily introduce modifications to the serialization contracts of arbitrary `IJsonTypeInfoResolver` instances.

C#

```
var options = new JsonSerializerOptions
{
    TypeInfoResolver = MyContext.Default
        .WithAddedModifier(static typeInfo =>
        {
            foreach (JsonPropertyInfo prop in typeInfo.Properties)
            {
                prop.Name = prop.Name.ToUpperInvariant();
            }
        })
};
```

New JsonContent.Create overloads

You can now create JsonContent instances using trim-safe or source-generated contracts. The new methods are:

- JsonContent.Create(Object, JsonTypeInfo, MediaTypeHeaderValue)
- JsonContent.Create<T>(T, JsonTypeInfo<T>, MediaTypeHeaderValue)

C#

```
var book = new Book(id: 42, "Title", "Author", publishedYear: 2023);
HttpContent content = JsonContent.Create(book, MyContext.Default.Book);

public record Book(int id, string title, string author, int publishedYear);

[JsonSerializable(typeof(Book))]
public partial class MyContext : JsonSerializerContext
```

```
{
}
```

Freeze a JsonSerializerOptions instance

The following new methods let you control when a JsonSerializerOptions instance is frozen:

- JsonSerializerOptions.MakeReadOnly()

  This overload is designed to be trim-safe and will therefore throw an exception in cases where the options instance hasn't been configured with a resolver.

- JsonSerializerOptions.MakeReadOnly(Boolean)

  If you pass `true` to this overload, it populates the options instance with the default reflection resolver if one is missing. This method is marked `RequiresUnreferenceCode` / `RequiresDynamicCode` and is therefore unsuitable for Native AOT applications.

The new IsReadOnly property lets you check if the options instance is frozen.

Time abstraction

The new TimeProvider class and ITimer interface add *time abstraction* functionality, which allows you to mock time in test scenarios. In addition, you can use the time abstraction to mock Task operations that rely on time progression using Task.Delay and Task.WaitAsync. The time abstraction supports the following essential time operations:

- Retrieve local and UTC time
- Obtain a timestamp for measuring performance
- Create a timer

The following code snippet shows some usage examples.

C#

```
// Get system time.
DateTimeOffset utcNow = TimeProvider.System.GetUtcNow();
DateTimeOffset localNow = TimeProvider.System.GetLocalNow();

TimerCallback callback = s => ((State)s!).Signal();
```

```
// Create a timer using the time provider.
ITimer timer = _timeProvider.CreateTimer(
    callback, null, TimeSpan.Zero, Timeout.InfiniteTimeSpan);

// Measure a period using the system time provider.
long providerTimestamp1 = TimeProvider.System.GetTimestamp();
long providerTimestamp2 = TimeProvider.System.GetTimestamp();

TimeSpan period = _timeProvider.GetElapsedTime(providerTimestamp1,
providerTimestamp2);
```

C#

```
// Create a time provider that works with a
// time zone that's different than the local time zone.
private class ZonedTimeProvider(TimeZoneInfo zoneInfo) : TimeProvider()
{
    private readonly TimeZoneInfo _zoneInfo = zoneInfo ?? TimeZoneInfo.Local;

    public override TimeZoneInfo LocalTimeZone => _zoneInfo;

    public static TimeProvider FromLocalTimeZone(TimeZoneInfo zoneInfo) =>
        new ZonedTimeProvider(zoneInfo);
}
```

UTF8 improvements

If you want to enable writing out a string-like representation of your type to a destination span, implement the new IUtf8SpanFormattable interface on your type. This new interface is closely related to ISpanFormattable, but targets UTF8 and `Span<byte>` instead of UTF16 and `Span<char>`.

IUtf8SpanFormattable has been implemented on all of the primitive types (plus others), with the exact same shared logic whether targeting `string`, `Span<char>`, or `Span<byte>`. It has full support for all formats (including the new "B" binary specifier) and all cultures. This means you can now format directly to UTF8 from `Byte`, `Complex`, `Char`, `DateOnly`, `DateTime`, `DateTimeOffset`, `Decimal`, `Double`, `Guid`, `Half`, `IPAddress`, `IPNetwork`, `Int16`, `Int32`, `Int64`, `Int128`, `IntPtr`, `NFloat`, `SByte`, `Single`, `Rune`, `TimeOnly`, `TimeSpan`, `UInt16`, `UInt32`, `UInt64`, `UInt128`, `UIntPtr`, and `Version`.

New Utf8.TryWrite methods provide a UTF8-based counterpart to the existing MemoryExtensions.TryWrite methods, which are UTF16-based. You can use interpolated string syntax to format a complex expression directly into a span of UTF8 bytes, for example:

C#

```
static bool FormatHexVersion(
    short major,
    short minor,
    short build,
    short revision,
    Span<byte> utf8Bytes,
    out int bytesWritten) =>
    Utf8.TryWrite(
        utf8Bytes,
        CultureInfo.InvariantCulture,
        $"{major:X4}.{minor:X4}.{build:X4}.{revision:X4}",
        out bytesWritten);
```

The implementation recognizes IUtf8SpanFormattable on the format values and uses their implementations to write their UTF8 representations directly to the destination span.

The implementation also utilizes the new Encoding.TryGetBytes(ReadOnlySpan<Char>, Span<Byte>, Int32) method, which together with its Encoding.TryGetChars(ReadOnlySpan<Byte>, Span<Char>, Int32) counterpart, supports encoding and decoding into a destination span. If the span isn't long enough to hold the resulting state, the methods return `false` rather than throwing an exception.

Methods for working with randomness

The System.Random and System.Security.Cryptography.RandomNumberGenerator types introduce two new methods for working with randomness.

GetItems<T>()

The new System.Random.GetItems and System.Security.Cryptography.RandomNumberGenerator.GetItems methods let you randomly choose a specified number of items from an input set. The following example shows how to use `System.Random.GetItems<T>()` (on the instance provided by the Random.Shared property) to randomly insert 31 items into an array. This example could be used in a game of "Simon" where players must remember a sequence of colored buttons.

C#

```
private static ReadOnlySpan<Button> s_allButtons = new[]
{
    Button.Red,
```

```
    Button.Green,
    Button.Blue,
    Button.Yellow,
};

// ...

Button[] thisRound = Random.Shared.GetItems(s_allButtons, 31);
// Rest of game goes here ...
```

Shuffle<T>()

The new Random.Shuffle and RandomNumberGenerator.Shuffle<T>(Span<T>) methods let you randomize the order of a span. These methods are useful for reducing training bias in machine learning (so the first thing isn't always training, and the last thing always test).

C#

```
YourType[] trainingData = LoadTrainingData();
Random.Shared.Shuffle(trainingData);

IDataView sourceData = mlContext.Data.LoadFromEnumerable(trainingData);

DataOperationsCatalog.TrainTestData split =
mlContext.Data.TrainTestSplit(sourceData);
model = chain.Fit(split.TrainSet);

IDataView predictions = model.Transform(split.TestSet);
// ...
```

Performance-focused types

.NET 8 introduces several new types aimed at improving app performance.

- The new System.Collections.Frozen namespace includes the collection types FrozenDictionary<TKey,TValue> and FrozenSet<T>. These types don't allow any changes to keys and values once a collection is created. That requirement allows faster read operations (for example, `TryGetValue()`). These types are particularly useful for collections that are populated on first use and then persisted for the duration of a long-lived service, for example:

  C#

```
private static readonly FrozenDictionary<string, bool> s_configurationData =
    LoadConfigurationData().ToFrozenDictionary();

// ...
if (s_configurationData.TryGetValue(key, out bool setting) && setting)
{
    Process();
}
```

- Methods like MemoryExtensions.IndexOfAny look for the first occurrence of *any value in the passed collection*. The new System.Buffers.SearchValues<T> type is designed to be passed to such methods. Correspondingly, .NET 8 adds new overloads of methods like MemoryExtensions.IndexOfAny that accept an instance of the new type. When you create an instance of SearchValues<T>, all the data that's necessary to optimize subsequent searches is derived *at that time*, meaning the work is done up front.

- The new System.Text.CompositeFormat type is useful for optimizing format strings that aren't known at compile time (for example, if the format string is loaded from a resource file). A little extra time is spent up front to do work such as parsing the string, but it saves the work from being done on each use.

C#

```
private static readonly CompositeFormat s_rangeMessage =
    CompositeFormat.Parse(LoadRangeMessageResource());

// ...
static string GetMessage(int min, int max) =>
    string.Format(CultureInfo.InvariantCulture, s_rangeMessage, min, max);
```

- New System.IO.Hashing.XxHash3 and System.IO.Hashing.XxHash128 types provide implementations of the fast XXH3 and XXH128 hash algorithms.

System.Numerics and System.Runtime.Intrinsics

This section covers improvements to the System.Numerics and System.Runtime.Intrinsics namespaces.

- Vector256<T>, Matrix3x2, and Matrix4x4 have improved hardware acceleration on .NET 8. For example, Vector256<T> was reimplemented to internally be `2x Vector128<T>` operations, where possible. This allows partial acceleration of some functions when

`Vector128.IsHardwareAccelerated == true` but `Vector256.IsHardwareAccelerated == false`, such as on Arm64.

- Hardware intrinsics are now annotated with the `ConstExpected` attribute. This ensures that users are aware when the underlying hardware expects a constant and therefore when a non-constant value may unexpectedly hurt performance.
- The Lerp(TSelf, TSelf, TSelf) `Lerp` API has been added to IFloatingPointIeee754<TSelf> and therefore to `float` (Single), `double` (Double), and Half. This API allows a linear interpolation between two values to be performed efficiently and correctly.

Vector512 and AVX-512

.NET Core 3.0 expanded SIMD support to include the platform-specific hardware intrinsics APIs for x86/x64. .NET 5 added support for Arm64 and .NET 7 added the cross-platform hardware intrinsics. .NET 8 furthers SIMD support by introducing Vector512<T> and support for Intel Advanced Vector Extensions 512 (AVX-512) instructions.

Specifically, .NET 8 includes support for the following key features of AVX-512:

- 512-bit vector operations
- Additional 16 SIMD registers
- Additional instructions available for 128-bit, 256-bit, and 512-bit vectors

If you have hardware that supports the functionality, then Vector512.IsHardwareAccelerated now reports `true`.

.NET 8 also adds several platform-specific classes under the System.Runtime.Intrinsics.X86 namespace:

- Avx512F (foundational)
- Avx512BW (byte and word)
- Avx512CD (conflict detection)
- Avx512DQ (doubleword and quadword)
- Avx512Vbmi (vector byte manipulation instructions)

These classes follow the same general shape as other instruction set architectures (ISAs) in that they expose an IsSupported property and a nested Avx512F.X64 class for instructions available only to 64-bit processes. Additionally, each class has a nested Avx512F.VL class that exposes the `Avx512VL` (vector length) extensions for the corresponding instruction set.

Even if you don't explicitly use `Vector512` -specific or `Avx512F` -specific instructions in your code, you'll likely still benefit from the new AVX-512 support. The JIT can take advantage of the additional registers and instructions implicitly when using Vector128<T> or Vector256<T>. The base class library uses these hardware intrinsics internally in most operations exposed by Span<T> and ReadOnlySpan<T> and in many of the math APIs exposed for the primitive types.

Data validation

The System.ComponentModel.DataAnnotations namespace includes new data validation attributes intended for validation scenarios in cloud-native services. While the pre-existing `DataAnnotations` validators are geared towards typical UI data-entry validation, such as fields on a form, the new attributes are designed to validate non-user-entry data, such as configuration options. In addition to the new attributes, new properties were added to the RangeAttribute type.

Expand table

New API	Description
RangeAttribute.MinimumIsExclusive RangeAttribute.MaximumIsExclusive	Specifies whether bounds are included in the allowable range.
System.ComponentModel.DataAnnotations.LengthAttribute	Specifies both lower and upper bounds for strings or collections. For example, `[Length(10, 20)]` requires at least 10 elements and at most 20 elements in a collection.
System.ComponentModel.DataAnnotations.Base64StringAttribute	Validates that a string is a valid Base64 representation.
System.ComponentModel.DataAnnotations.AllowedValuesAttribute System.ComponentModel.DataAnnotations.DeniedValuesAttribute	Specify allow lists and deny lists, respectively. For example, `[AllowedValues("apple", "banana", "mango")]`.

Metrics

New APIs let you attach key-value pair tags to Meter and Instrument objects when you create them. Aggregators of published metric measurements can use the tags to differentiate the aggregated values.

C#

```
var options = new MeterOptions("name")
{
    Version = "version",
    // Attach these tags to the created meter.
    Tags = new TagList()
    {
        { "MeterKey1", "MeterValue1" },
        { "MeterKey2", "MeterValue2" }
    }
};

Meter meter = meterFactory!.Create(options);

Counter<int> counterInstrument = meter.CreateCounter<int>(
    "counter", null, null, new TagList() { { "counterKey1", "counterValue1" } }
);
counterInstrument.Add(1);
```

The new APIs include:

- MeterOptions
- Meter(MeterOptions)
- CreateCounter<T>(String, String, String, IEnumerable<KeyValuePair<String,Object>>)

Cryptography

.NET 8 adds support for the SHA-3 hashing primitives. (SHA-3 is currently supported by Linux with OpenSSL 1.1.1 or later and Windows 11 Build 25324 or later.) APIs where SHA-2 is available now offer a SHA-3 compliment. This includes `SHA3_256`, `SHA3_384`, and `SHA3_512` for hashing; `HMACSHA3_256`, `HMACSHA3_384`, and `HMACSHA3_512` for HMAC; `HashAlgorithmName.SHA3_256`, `HashAlgorithmName.SHA3_384`, and `HashAlgorithmName.SHA3_512` for hashing where the algorithm is configurable; and `RSAEncryptionPadding.OaepSHA3_256`, `RSAEncryptionPadding.OaepSHA3_384`, and `RSAEncryptionPadding.OaepSHA3_512` for RSA OAEP encryption.

The following example shows how to use the APIs, including the `SHA3_256.IsSupported` property to determine if the platform supports SHA-3.

C#

```
// Hashing example
if (SHA3_256.IsSupported)
```

```
{
    byte[] hash = SHA3_256.HashData(dataToHash);
}
else
{
    // ...
}

// Signing example
if (SHA3_256.IsSupported)
{
     using ECDsa ec = ECDsa.Create(ECCurve.NamedCurves.nistP256);
     byte[] signature = ec.SignData(dataToBeSigned, HashAlgorithmName.SHA3_256);
}
else
{
    // ...
}
```

SHA-3 support is currently aimed at supporting cryptographic primitives. Higher-level constructions and protocols aren't expected to fully support SHA-3 initially. These protocols include X.509 certificates, SignedXml, and COSE.

Networking

Support for HTTPS proxy

Until now, the proxy types that HttpClient supported all allowed a "man-in-the-middle" to see which site the client is connecting to, even for HTTPS URIs. HttpClient now supports *HTTPS proxy*, which creates an encrypted channel between the client and the proxy so all requests can be handled with full privacy.

To enable HTTPS proxy, set the `all_proxy` environment variable, or use the WebProxy class to control the proxy programmatically.

Unix: `export all_proxy=https://x.x.x.x:3218` Windows: `set all_proxy=https://x.x.x.x:3218`

You can also use the WebProxy class to control the proxy programmatically.

Stream-based ZipFile methods

.NET 8 includes new overloads of ZipFile.CreateFromDirectory that allow you to collect all the files included in a directory and zip them, then store the resulting zip file into the provided stream.

Similarly, new ZipFile.ExtractToDirectory overloads let you provide a stream containing a zipped file and extract its contents into the filesystem. These are the new overloads:

C#

```
namespace System.IO.Compression;

public static partial class ZipFile
{
    public static void CreateFromDirectory(
        string sourceDirectoryName, Stream destination);

    public static void CreateFromDirectory(
        string sourceDirectoryName,
        Stream destination,
        CompressionLevel compressionLevel,
        bool includeBaseDirectory);

    public static void CreateFromDirectory(
        string sourceDirectoryName,
        Stream destination,
        CompressionLevel compressionLevel,
        bool includeBaseDirectory,
    Encoding? entryNameEncoding);

    public static void ExtractToDirectory(
        Stream source, string destinationDirectoryName) { }

    public static void ExtractToDirectory(
        Stream source, string destinationDirectoryName, bool overwriteFiles) { }

    public static void ExtractToDirectory(
        Stream source, string destinationDirectoryName, Encoding? entryNameEncoding)
{ }

    public static void ExtractToDirectory(
        Stream source, string destinationDirectoryName, Encoding? entryNameEncoding,
bool overwriteFiles) { }
}
```

These new APIs can be useful when disk space is constrained, because they avoid having to use the disk as an intermediate step.

Extension libraries

This section contains the following subtopics:

- Options validation
- LoggerMessageAttribute constructors
- Extensions metrics
- Hosted lifecycle services
- Keyed DI services
- System.Numerics.Tensors.TensorPrimitives

Keyed DI services

Keyed dependency injection (DI) services provide a means for registering and retrieving DI services using keys. By using keys, you can scope how you register and consume services. These are some of the new APIs:

- The IKeyedServiceProvider interface.
- The ServiceKeyAttribute attribute, which can be used to inject the key that was used for registration/resolution in the constructor.
- The FromKeyedServicesAttribute attribute, which can be used on service constructor parameters to specify which keyed service to use.
- Various new extension methods for IServiceCollection to support keyed services, for example, ServiceCollectionServiceExtensions.AddKeyedScoped.
- The ServiceProvider implementation of IKeyedServiceProvider.

The following example shows you how to use keyed DI services.

C#

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddSingleton<BigCacheConsumer>();
builder.Services.AddSingleton<SmallCacheConsumer>();
builder.Services.AddKeyedSingleton<ICache, BigCache>("big");
builder.Services.AddKeyedSingleton<ICache, SmallCache>("small");
WebApplication app = builder.Build();
app.MapGet("/big", (BigCacheConsumer data) => data.GetData());
app.MapGet("/small", (SmallCacheConsumer data) => data.GetData());
app.MapGet("/big-cache", ([FromKeyedServices("big")] ICache cache) =>
cache.Get("data"));
app.MapGet("/small-cache", (HttpContext httpContext) => httpContext.RequestSer-
vices.GetRequiredKeyedService<ICache>("small").Get("data"));
app.Run();

class BigCacheConsumer([FromKeyedServices("big")] ICache cache)
{
    public object? GetData() => cache.Get("data");
}
```

```
class SmallCacheConsumer(IServiceProvider serviceProvider)
{
    public object? GetData() => serviceProvider.GetRequiredKeyedService<ICache>
("small").Get("data");
}

public interface ICache
{
    object Get(string key);
}

public class BigCache : ICache
{
    public object Get(string key) => $"Resolving {key} from big cache.";
}

public class SmallCache : ICache
{
    public object Get(string key) => $"Resolving {key} from small cache.";
}
```

For more information, see dotnet/runtime#64427 .

Hosted lifecycle services

Hosted services now have more options for execution during the application lifecycle. IHostedService provided `StartAsync` and `StopAsync`, and now IHostedLifecycleService provides these additional methods:

- StartingAsync(CancellationToken)
- StartedAsync(CancellationToken)
- StoppingAsync(CancellationToken)
- StoppedAsync(CancellationToken)

These methods run before and after the existing points respectively.

The following example shows how to use the new APIs.

C#

```
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
```

```
internal class HostedLifecycleServices
{
    public async static void RunIt()
    {
        IHostBuilder hostBuilder = new HostBuilder();
        hostBuilder.ConfigureServices(services =>
        {
            services.AddHostedService<MyService>();
        });

        using (IHost host = hostBuilder.Build())
        {
            await host.StartAsync();
        }
    }

    public class MyService : IHostedLifecycleService
    {
        public Task StartingAsync(CancellationToken cancellationToken) => /* add log-
ic here */ Task.CompletedTask;
        public Task StartAsync(CancellationToken cancellationToken) => /* add logic
here */ Task.CompletedTask;
        public Task StartedAsync(CancellationToken cancellationToken) => /* add logic
here */ Task.CompletedTask;
        public Task StopAsync(CancellationToken cancellationToken) => /* add logic
here */ Task.CompletedTask;
        public Task StoppedAsync(CancellationToken cancellationToken) => /* add logic
here */ Task.CompletedTask;
        public Task StoppingAsync(CancellationToken cancellationToken) => /* add log-
ic here */ Task.CompletedTask;
    }
}
```

For more information, see dotnet/runtime#86511 .

Options validation

Source generator

To reduce startup overhead and improve the validation feature set, we've introduced a source-code generator that implements the validation logic. The following code shows example models and validator classes.

C#

```
public class FirstModelNoNamespace
{
    [Required]
    [MinLength(5)]
    public string P1 { get; set; } = string.Empty;

    [Microsoft.Extensions.Options.ValidateObjectMembers(
        typeof(SecondValidatorNoNamespace))]
    public SecondModelNoNamespace? P2 { get; set; }
}

public class SecondModelNoNamespace
{
    [Required]
    [MinLength(5)]
    public string P4 { get; set; } = string.Empty;
}

[OptionsValidator]
public partial class FirstValidatorNoNamespace
    : IValidateOptions<FirstModelNoNamespace>
{
}

[OptionsValidator]
public partial class SecondValidatorNoNamespace
    : IValidateOptions<SecondModelNoNamespace>
{
}
```

If your app uses dependency injection, you can inject the validation as shown in the following example code.

C#

```
WebApplicationBuilder builder = WebApplication.CreateBuilder(args);
builder.Services.AddControllersWithViews();
builder.Services.Configure<FirstModelNoNamespace>(
    builder.Configuration.GetSection("some string"));

builder.Services.AddSingleton<
    IValidateOptions<FirstModelNoNamespace>, FirstValidatorNoNamespace>();
builder.Services.AddSingleton<
    IValidateOptions<SecondModelNoNamespace>, SecondValidatorNoNamespace>();
```

ValidateOptionsResultBuilder type

.NET 8 introduces the ValidateOptionsResultBuilder type to facilitate the creation of a ValidateOptionsResult object. Importantly, this builder allows for the accumulation of multiple errors. Previously, creating the ValidateOptionsResult object that's required to implement IValidateOptions<TOptions>.Validate(String, TOptions) was difficult and sometimes resulted in layered validation errors. If there were multiple errors, the validation process often stopped at the first error.

The following code snippet shows an example usage of ValidateOptionsResultBuilder.

C#

```
ValidateOptionsResultBuilder builder = new();
builder.AddError("Error: invalid operation code");
builder.AddResult(ValidateOptionsResult.Fail("Invalid request parameters"));
builder.AddError("Malformed link", "Url");

// Build ValidateOptionsResult object has accumulating multiple errors.
ValidateOptionsResult result = builder.Build();

// Reset the builder to allow using it in new validation operation.
builder.Clear();
```

LoggerMessageAttribute constructors

LoggerMessageAttribute now offers additional constructor overloads. Previously, you had to choose either the parameterless constructor or the constructor that required all of the parameters (event ID, log level, and message). The new overloads offer greater flexibility in specifying the required parameters with reduced code. If you don't supply an event ID, the system generates one automatically.

C#

```
public LoggerMessageAttribute(LogLevel level, string message);
public LoggerMessageAttribute(LogLevel level);
public LoggerMessageAttribute(string message);
```

Extensions metrics

IMeterFactory interface

You can register the new IMeterFactory interface in dependency injection (DI) containers and use it to create Meter objects in an isolated manner.

Register the IMeterFactory to the DI container using the default meter factory implementation:

C#

```
// 'services' is the DI IServiceCollection.
services.AddMetrics();
```

Consumers can then obtain the meter factory and use it to create a new Meter object.

C#

```
IMeterFactory meterFactory = serviceProvider.GetRequiredService<IMeterFactory>();

MeterOptions options = new MeterOptions("MeterName")
{
    Version = "version",
};

Meter meter = meterFactory.Create(options);
```

MetricCollector<T> class

The new MetricCollector<T> class lets you record metric measurements along with timestamps. Additionally, the class offers the flexibility to use a time provider of your choice for accurate timestamp generation.

C#

```
const string CounterName = "MyCounter";
DateTimeOffset now = DateTimeOffset.Now;

var timeProvider = new FakeTimeProvider(now);
using var meter = new Meter(Guid.NewGuid().ToString());
Counter<long> counter = meter.CreateCounter<long>(CounterName);
using var collector = new MetricCollector<long>(counter, timeProvider);

Assert.IsNull(collector.LastMeasurement);

counter.Add(3);

// Verify the update was recorded.
Assert.AreEqual(counter, collector.Instrument);
```

```
Assert.IsNotNull(collector.LastMeasurement);

Assert.AreSame(collector.GetMeasurementSnapshot().Last(), collector.LastMeasurement);
Assert.AreEqual(3, collector.LastMeasurement.Value);
Assert.AreEqual(now, collector.LastMeasurement.Timestamp);
```

System.Numerics.Tensors.TensorPrimitives

The updated System.Numerics.Tensors NuGet package includes APIs in the new System.Numerics.Tensors.TensorPrimitives type that add support for tensor operations. The tensor primitives optimize data-intensive workloads like those of AI and machine learning.

AI workloads like semantic search and retrieval-augmented generation (RAG) extend the natural language capabilities of large language models, such as ChatGPT, by augmenting prompts with relevant data. For these workloads, operations on vectors—like *cosine similarity* to find the most relevant data to answer a question—are crucial. The TensorPrimitives type provides APIs for vector operations.

For more information, see the Announcing .NET 8 RC 2 blog post .

Native AOT support

The option to publish as Native AOT was first introduced in .NET 7. Publishing an app with Native AOT creates a fully self-contained version of your app that doesn't need a runtime—everything is included in a single file. .NET 8 brings the following improvements to Native AOT publishing:

- Adds support for the x64 and Arm64 architectures on *macOS*.
- Reduces the sizes of Native AOT apps on Linux by up to 50%. The following table shows the size of a "Hello World" app published with Native AOT that includes the entire .NET runtime on .NET 7 vs. .NET 8:

Expand table

Operating system	.NET 7	.NET 8
Linux x64 (with `-p:StripSymbols=true`)	3.76 MB	1.84 MB
Windows x64	2.85 MB	1.77 MB

- Lets you specify an optimization preference: size or speed. By default, the compiler chooses to generate fast code while being mindful of the size of the application. However, you can use the `<OptimizationPreference>` MSBuild property to optimize specifically for one or the other. For more information, see Optimize AOT deployments.

Target iOS-like platforms with Native AOT

.NET 8 starts the work to enable Native AOT support for iOS-like platforms. You can now build and run .NET iOS and .NET MAUI applications with Native AOT on the following platforms:

- `ios`
- `iossimulator`
- `maccatalyst`
- `tvos`
- `tvossimulator`

Preliminary testing shows that app size on disk decreases by about 35% for .NET iOS apps that use Native AOT instead of Mono. App size on disk for .NET MAUI iOS apps decreases up to 50%. Additionally, the startup time is also faster. .NET iOS apps have about 28% faster startup time, while .NET MAUI iOS apps have about 50% better startup performance compared to Mono. The .NET 8 support is experimental and only the first step for the feature as a whole. For more information, see the .NET 8 Performance Improvements in .NET MAUI blog post .

Native AOT support is available as an opt-in feature intended for app deployment; Mono is still the default runtime for app development and deployment. To build and run a .NET MAUI application with Native AOT on an iOS device, use `dotnet workload install maui` to install the .NET MAUI workload and `dotnet new maui -n HelloMaui` to create the app. Then, set the MSBuild property `PublishAot` to `true` in the project file.

XML

```xml
<PropertyGroup>
  <PublishAot>true</PublishAot>
</PropertyGroup>
```

When you set the required property and run `dotnet publish` as shown in the following example, the app will be deployed by using Native AOT.

.NET CLI

```
dotnet publish -f net8.0-ios -c Release -r ios-arm64  /t:Run
```

Limitations

Not all iOS features are compatible with Native AOT. Similarly, not all libraries commonly used in iOS are compatible with NativeAOT. And in addition to the existing limitations of Native AOT deployment, the following list shows some of the other limitations when targeting iOS-like platforms:

- Using Native AOT is only enabled during app deployment (`dotnet publish`).
- Managed code debugging is only supported with Mono.
- Compatibility with the .NET MAUI framework is limited.

AOT compilation for Android apps

To decrease app size, .NET and .NET MAUI apps that target Android use *profiled* ahead-of-time (AOT) compilation mode when they're built in Release mode. Profiled AOT compilation affects fewer methods than regular AOT compilation. .NET 8 introduces the `<AndroidStripILAfterAOT>` property that lets you opt-in to further AOT compilation for Android apps to decrease app size even more.

XML

```
<PropertyGroup>
  <AndroidStripILAfterAOT>true</AndroidStripILAfterAOT>
</PropertyGroup>
```

By default, setting `AndroidStripILAfterAOT` to `true` overrides the default `AndroidEnableProfiledAot` setting, allowing (nearly) all methods that were AOT-compiled to be trimmed. You can also use profiled AOT and IL stripping together by explicitly setting both properties to `true`:

XML

```
<PropertyGroup>
  <AndroidStripILAfterAOT>true</AndroidStripILAfterAOT>
  <AndroidEnableProfiledAot>true</AndroidEnableProfiledAot>
</PropertyGroup>
```

Cross-built Windows apps

When you build apps that target Windows on non-Windows platforms, the resulting executable is now updated with any specified Win32 resources—for example, application icon, manifest, version information.

Previously, applications had to be built on Windows in order to have such resources. Fixing this gap in cross-building support has been a popular request, as it was a significant pain point affecting both infrastructure complexity and resource usage.

See also

- What's new in .NET 8