What's new in .NET libraries for .NET 10

This article describes new features in the .NET libraries for .NET 10.

Cryptography

- Find certificates by thumbprints other than SHA-1
- Find PEM-encoded data in ASCII/UTF-8
- Encryption algorithm for PKCS#12/PFX export
- Post-quantum cryptography (PQC)

Find certificates by thumbprints other than SHA-1

Finding certificates uniquely by thumbprint is a fairly common operation, but the X509Certificate2Collection.Find(X509FindType, Object, Boolean) method (for the FindByThumbprint mode) only searches for the SHA-1 thumbprint value.

There's some risk to using the `Find` method for finding SHA-2-256 ("SHA256") and SHA-3-256 thumbprints since these hash algorithms have the same lengths.

Instead, .NET 10 introduces a new method that accepts the name of the hash algorithm to use for matching.

```C#
X509Certificate2Collection coll = store.Certificates.FindByThumbprint(HashAl-
gorithmName.SHA256, thumbprint);
Debug.Assert(coll.Count < 2, "Collection has too many matches, has SHA-2 been
broken?");
return coll.SingleOrDefault();
```

Find PEM-encoded data in ASCII/UTF-8

The PEM encoding (originally *Privacy Enhanced Mail*, but now used widely outside of email) is defined for "text", which means that the PemEncoding class was designed to run on String and `ReadOnlySpan<char>`. However, it's common (especially on Linux) to have something like a certificate written in a file that uses the ASCII (string) encoding. Historically, that meant you needed to open the file and convert the bytes to chars (or a string) before you could use `PemEncoding`.

The new PemEncoding.FindUtf8(ReadOnlySpan<Byte>) method takes advantage of the fact that PEM is only defined for 7-bit ASCII characters, and that 7-bit ASCII has a perfect overlap with single-byte UTF-8 values. By calling this new method, you can skip the UTF-8/ASCII-to-char conversion and read the file directly.

```diff
byte[] fileContents = File.ReadAllBytes(path);
-char[] text = Encoding.ASCII.GetString(fileContents);
-PemFields pemFields = PemEncoding.Find(text);
+PemFields pemFields = PemEncoding.FindUtf8(fileContents);

-byte[] contents = Base64.DecodeFromChars(text.AsSpan()
[pemFields.Base64Data]);
+byte[] contents = Base64.DecodeFromUtf8(fileContents.AsSpan()
[pemFields.Base64Data]);
```

Encryption algorithm for PKCS#12/PFX export

The new ExportPkcs12 methods on X509Certificate allow callers to choose what encryption and digest algorithms are used to produce the output:

- Pkcs12ExportPbeParameters.Pkcs12TripleDesSha1 indicates the Windows XP-era de facto standard. It produces an output supported by almost every library and platform that supports reading PKCS#12/PFX by choosing an older encryption algorithm.
- Pkcs12ExportPbeParameters.Pbes2Aes256Sha256 indicates that AES should be used instead of 3DES (and SHA-2-256 instead of SHA-1), but the output might not be understood by all readers (such as Windows XP).

If you want even more control, you can use the overload that accepts a PbeParameters.

Post-quantum cryptography (PQC)

.NET 10 includes support for three new asymmetric algorithms: ML-KEM (FIPS 203), ML-DSA (FIPS 204), and SLH-DSA (FIPS 205). The new types are:

- System.Security.Cryptography.MLKem
- System.Security.Cryptography.MLDsa
- System.Security.Cryptography.SlhDsa

Because it adds little benefit, these new types don't derive from AsymmetricAlgorithm. Rather than the `AsymmetricAlgorithm` approach of creating an object and then importing a key into it, or generating a fresh key, the new types all use static methods to generate or import a key:

C#

```csharp
using System;
using System.IO;
using System.Security.Cryptography;

private static bool ValidateMLDsaSignature(ReadOnlySpan<byte> data,
ReadOnlySpan<byte> signature, string publicKeyPath)
{
    string publicKeyPem = File.ReadAllText(publicKeyPath);

    using (MLDsa key = MLDsa.ImportFromPem(publicKeyPem))
    {
        return key.VerifyData(data, signature);
    }
}
```

And rather than setting object properties and having a key materialize, key generation on these new types takes in all of the options it needs.

C#

```csharp
using (MLKem key = MLKem.GenerateKey(MLKemAlgorithm.MLKem768))
{
    string publicKeyPem = key.ExportSubjectPublicKeyInfoPem();
    ...
}
```

These algorithms all continue with the pattern of having a static `IsSupported` property to indicate if the algorithm is supported on the current system.

.NET 10 includes Windows Cryptography API: Next Generation (CNG) support for Post-Quantum Cryptography (PQC), which makes these algorithms available on Windows systems with PQC support. For example:

C#

```csharp
using System;
using System.IO;
using System.Security.Cryptography;

private static bool ValidateMLDsaSignature(ReadOnlySpan<byte> data,
ReadOnlySpan<byte> signature, string publicKeyPath)
{
    string publicKeyPem = File.ReadAllText(publicKeyPath);

    using MLDsa key = MLDsa.ImportFromPem(publicKeyPem);
    return key.VerifyData(data, signature);
}
```

The PQC algorithms are available on systems where the system cryptographic libraries are OpenSSL 3.5 (or newer) or Windows CNG with PQC support. The MLKem type isn't marked as `[Experimental]`, but some of its methods are (and will be until the underlying standards are finalized). The MLDsa, SlhDsa, and CompositeMLDsa classes are marked as `[Experimental]` under diagnostic `SYSLIB5006` until development is complete.

ML-DSA

The MLDsa class includes ease-of-use features that simplify common code patterns:

diff

```diff
private static byte[] SignData(string privateKeyPath, ReadOnlySpan<byte>
data)
{
    using (MLDsa signingKey =
MLDsa.ImportFromPem(File.ReadAllBytes(privateKeyPath)))
    {
-       byte[] signature = new
byte[signingKey.Algorithm.SignatureSizeInBytes];
-       signingKey.SignData(data, signature);
+       return signingKey.SignData(data);
-       return signature;
    }
}
```

Additionally, .NET 10 adds support for HashML-DSA, which is called "PreHash" to help distinguish it from "pure" ML-DSA. As the underlying specification interacts with the Object Identifier (OID) value, the SignPreHash and VerifyPreHash methods on this `[Experimental]` type take the dotted-decimal OID as a string. This might evolve as more scenarios using HashML-DSA become well-defined.

C#

```csharp
private static byte[] SignPreHashSha3_256(MLDsa signingKey,
ReadOnlySpan<byte> data)
{
    const string Sha3_256Oid = "2.16.840.1.101.3.4.2.8";
    return signingKey.SignPreHash(SHA3_256.HashData(data), Sha3_256Oid);
}
```

Starting in RC 1, ML-DSA also supports signatures created and verified from an "external" mu value, which provides additional flexibility for advanced cryptographic scenarios:

C#

```
private static byte[] SignWithExternalMu(MLDsa signingKey, ReadOnlySpan<byte>
externalMu)
{
    return signingKey.SignMu(externalMu);
}

private static bool VerifyWithExternalMu(MLDsa verifyingKey,
ReadOnlySpan<byte> externalMu, ReadOnlySpan<byte> signature)
{
    return verifyingKey.VerifyMu(externalMu, signature);
}
```

Composite ML-DSA

.NET 10 introduces new types to support ietf-lamps-pq-composite-sigs (at draft 8 as of .NET 10 GA), including the CompositeMLDsa and CompositeMLDsaAlgorithm types, with implementation of the primitive methods for RSA variants.

C#

```csharp
var algorithm = CompositeMLDsaAlgorithm.MLDsa65WithRSA4096Pss;
using var privateKey = CompositeMLDsa.GenerateKey(algorithm);

byte[] data = [42];
byte[] signature = privateKey.SignData(data);

using var publicKey = CompositeMLDsa.ImportCompositeMLDsaPublicKey(algorithm,
privateKey.ExportCompositeMLDsaPublicKey());
Console.WriteLine(publicKey.VerifyData(data, signature)); // True

signature[0] ^= 1; // Tamper with signature
Console.WriteLine(publicKey.VerifyData(data, signature)); // False
```

AES KeyWrap with Padding (IETF RFC 5649)

AES-KWP is an algorithm that's occasionally used in constructions like Cryptographic Message Syntax (CMS) EnvelopedData, where content is encrypted once, but the decryption key needs to be distributed to multiple parties, each one in a distinct secret form.

.NET now supports the AES-KWP algorithm via instance methods on the Aes class:

C#

```csharp
private static byte[] DecryptContent(ReadOnlySpan<byte> kek,
ReadOnlySpan<byte> encryptedKey, ReadOnlySpan<byte> ciphertext)
{
    using (Aes aes = Aes.Create())
    {
        aes.SetKey(kek);

        Span<byte> dek = stackalloc byte[256 / 8];
        int length = aes.DecryptKeyWrapPadded(encryptedKey, dek);

        aes.SetKey(dek.Slice(0, length));
        return aes.DecryptCbc(ciphertext);
    }
}
```

Globalization and date/time

- New method overloads in ISOWeek for DateOnly type
- Numeric ordering for string comparison
- New TimeSpan.FromMilliseconds overload with single parameter

New method overloads in ISOWeek for DateOnly type

The ISOWeek class was originally designed to work exclusively with DateTime, as it was introduced before the DateOnly type existed. Now that `DateOnly` is available, it makes sense for `ISOWeek` to support it as well. The following overloads are new:

- GetWeekOfYear(DateOnly)
- GetYear(DateOnly)
- ToDateOnly(Int32, Int32, DayOfWeek)

Numeric ordering for string comparison

Numerical string comparison is a highly requested feature for comparing strings numerically instead of lexicographically. For example, `2` is less than `10`, so `"2"` should appear before `"10"` when ordered numerically. Similarly, `"2"` and `"02"` are equal numerically. With the new NumericOrdering option, it's now possible to do these types of comparisons:

```C#
StringComparer numericStringComparer =
StringComparer.Create(CultureInfo.CurrentCulture,
CompareOptions.NumericOrdering);

Console.WriteLine(numericStringComparer.Equals("02", "2"));
// Output: True

foreach (string os in new[] { "Windows 8", "Windows 10", "Windows 11"
}.Order(numericStringComparer))
{
    Console.WriteLine(os);
}

// Output:
// Windows 8
// Windows 10
// Windows 11

HashSet<string> set = new HashSet<string>(numericStringComparer) { "007" };
Console.WriteLine(set.Contains("7"));
// Output: True
```

This option isn't valid for the following index-based string operations: `IndexOf`, `LastIndexOf`, `StartsWith`, `EndsWith`, `IsPrefix`, and `IsSuffix`.

New `TimeSpan.FromMilliseconds` overload with single parameter

The TimeSpan.FromMilliseconds(Int64, Int64) method was introduced previously without adding an overload that takes a single parameter.

Although this works since the second parameter is optional, it causes a compilation error when used in a LINQ expression like:

```C#
Expression<Action> a = () => TimeSpan.FromMilliseconds(1000);
```

The issue arises because LINQ expressions can't handle optional parameters. To address this, .NET 10 introduces a new overload takes a single parameter. It also modifies the existing method to make the second parameter mandatory.

Strings

- String normalization APIs to work with span of characters
- UTF-8 support for hex-string conversion

String normalization APIs to work with span of characters

Unicode string normalization has been supported for a long time, but existing APIs only worked with the string type. This means that callers with data stored in different forms, such as character arrays or spans, must allocate a new string to use these APIs. Additionally, APIs that return a normalized string always allocate a new string to represent the normalized output.

.NET 10 introduces new APIs that work with spans of characters, which expand normalization beyond string types and help to avoid unnecessary allocations:

- StringNormalizationExtensions.GetNormalizedLength(ReadOnlySpan<Char>, NormalizationForm)
- StringNormalizationExtensions.IsNormalized(ReadOnlySpan<Char>, NormalizationForm)
- StringNormalizationExtensions.TryNormalize(ReadOnlySpan<Char>, Span<Char>, Int32, NormalizationForm)

UTF-8 support for hex-string conversion

.NET 10 adds UTF-8 support for hex-string conversion operations in the Convert class. These new methods provide efficient ways to convert between UTF-8 byte sequences and hexadecimal representations without requiring intermediate string allocations:

- Convert.FromHexString(ReadOnlySpan<Byte>)
- Convert.FromHexString(ReadOnlySpan<Byte>, Span<Byte>, Int32, Int32)
- Convert.TryToHexString(ReadOnlySpan<Byte>, Span<Byte>, Int32)
- Convert.TryToHexStringLower(ReadOnlySpan<Byte>, Span<Byte>, Int32)

These methods mirror the existing overloads that work with `string` and `ReadOnlySpan<char>`, but operate directly on UTF-8 encoded bytes for improved performance in scenarios where you're already working with UTF-8 data.

Collections

- Additional TryAdd and TryGetValue overloads for OrderedDictionary<TKey, TValue>

Additional `TryAdd` and `TryGetValue` overloads for `OrderedDictionary<TKey, TValue>`

OrderedDictionary<TKey,TValue> provides `TryAdd` and `TryGetValue` for addition and retrieval like any other `IDictionary<TKey, TValue>` implementation. However, there are scenarios where you might want to perform more operations, so new overloads are added that return an index to the entry:

- TryAdd(TKey, TValue, Int32)
- TryGetValue(TKey, TValue, Int32)

This index can then be used with GetAt and SetAt for fast access to the entry. An example usage of the new `TryAdd` overload is to add or update a key-value pair in the ordered dictionary:

```C#
// Try to add a new key with value 1.
if (!orderedDictionary.TryAdd(key, 1, out int index))
{
    // Key was present, so increment the existing value instead.
    int value = orderedDictionary.GetAt(index).Value;
    orderedDictionary.SetAt(index, value + 1);
}
```

This new API is already used in JsonObject and improves the performance of updating properties by 10-20%.

Serialization

- Allow specifying ReferenceHandler in JsonSourceGenerationOptions
- Option to disallow duplicate JSON properties
- Strict JSON serialization options
- PipeReader support for JSON serializer

Allow specifying ReferenceHandler in `JsonSourceGenerationOptions`

When you use source generators for JSON serialization, the generated context throws when cycles are serialized or deserialized. Now you can customize this behavior by specifying the ReferenceHandler in the JsonSourceGenerationOptionsAttribute. Here's an example using `JsonKnownReferenceHandler.Preserve`:

```C#
public static void MakeSelfRef()
{
    SelfReference selfRef = new SelfReference();
    selfRef.Me = selfRef;

    Console.WriteLine(JsonSerializer.Serialize(selfRef,
ContextWithPreserveReference.Default.SelfReference));
    // Output: {"$id":"1","Me":{"$ref":"1"}}
}

[JsonSourceGenerationOptions(ReferenceHandler =
JsonKnownReferenceHandler.Preserve)]
[JsonSerializable(typeof(SelfReference))]
internal partial class ContextWithPreserveReference : JsonSerializerContext
```

```
{
}

internal class SelfReference
{
    public SelfReference Me { get; set; } = null!;
}
```

Option to disallow duplicate JSON properties

The JSON specification doesn't specify how to handle duplicate properties when deserializing a JSON payload. This can lead to unexpected results and security vulnerabilities. .NET 10 introduces the JsonSerializerOptions.AllowDuplicateProperties option to disallow duplicate JSON properties:

C#

```
string json = """{ "Value": 1, "Value": -1 }""";
Console.WriteLine(JsonSerializer.Deserialize<MyRecord>(json).Value); // -1

JsonSerializerOptions options = new() { AllowDuplicateProperties = false };
JsonSerializer.Deserialize<MyRecord>(json, options);                // throws
JsonException
JsonSerializer.Deserialize<JsonObject>(json, options);              // throws
JsonException
JsonSerializer.Deserialize<Dictionary<string, int>>(json, options); // throws
JsonException

JsonDocumentOptions docOptions = new() { AllowDuplicateProperties = false };
JsonDocument.Parse(json, docOptions);   // throws JsonException

record MyRecord(int Value);
```

Duplicates are detected by checking if a value is assigned multiple times during deserialization, so it works as expected with other options like case-sensitivity and naming policy.

Strict JSON serialization options

The JSON serializer accepts many options to customize serialization and deserialization, but the defaults might be too relaxed for some applications. .NET 10 adds a new JsonSerializerOptions.Strict preset that follows best practices by including the following options:

- Applies the JsonUnmappedMemberHandling.Disallow policy.
- Disables JsonSerializerOptions.AllowDuplicateProperties.
- Preserves case sensitive property binding.

- Enables both JsonSerializerOptions.RespectNullableAnnotations and JsonSerializerOptions.RespectRequiredConstructorParameters settings.

These options are read-compatible with JsonSerializerOptions.Default - an object serialized with JsonSerializerOptions.Default can be deserialized with JsonSerializerOptions.Strict.

For more information about JSON serialization, see System.Text.Json overview.

PipeReader support for JSON serializer

JsonSerializer.Deserialize now supports PipeReader, complementing the existing PipeWriter support. Previously, deserializing from a `PipeReader` required converting it to a Stream, but the new overloads eliminate that step by integrating `PipeReader` directly into the serializer. As a bonus, not having to convert from what you're already holding can yield some efficiency benefits.

This shows the basic usage:

C#

```csharp
using System;
using System.IO.Pipelines;
using System.Text.Json;
using System.Threading.Tasks;

var pipe = new Pipe();

// Serialize to writer
await JsonSerializer.SerializeAsync(pipe.Writer, new Person("Alice"));
await pipe.Writer.CompleteAsync();

// Deserialize from reader
var result = await JsonSerializer.DeserializeAsync<Person>(pipe.Reader);
await pipe.Reader.CompleteAsync();

Console.WriteLine($"Your name is {result.Name}.");
// Output: Your name is Alice.

record Person(string Name);
```

Here is an example of a producer that produces tokens in chunks and a consumer that receives and displays them:

C#

```csharp
using System;
using System.Collections.Generic;
using System.IO.Pipelines;
using System.Text.Json;
using System.Threading.Tasks;

var pipe = new Pipe();

// Producer writes to the pipe in chunks.
var producerTask = Task.Run(async () =>
{
    async static IAsyncEnumerable<Chunk> GenerateResponse()
    {
        yield return new Chunk("The quick brown fox", DateTime.Now);
        await Task.Delay(500);
        yield return new Chunk(" jumps over", DateTime.Now);
        await Task.Delay(500);
        yield return new Chunk(" the lazy dog.", DateTime.Now);
    }

    await JsonSerializer.SerializeAsync<IAsyncEnumerable<Chunk>>(pipe.Writer,
GenerateResponse());
    await pipe.Writer.CompleteAsync();
});

// Consumer reads from the pipe and outputs to console.
var consumerTask = Task.Run(async () =>
{
    var thinkingString = "...";
    var clearThinkingString = new string("\b\b\b");
    var lastTimestamp = DateTime.MinValue;

    // Read response to end.
    Console.Write(thinkingString);
    await foreach (var chunk in
JsonSerializer.DeserializeAsyncEnumerable<Chunk>(pipe.Reader))
    {
        Console.Write(clearThinkingString);
        Console.Write(chunk.Message);
        Console.Write(thinkingString);
        lastTimestamp = DateTime.Now;
    }

    Console.Write(clearThinkingString);
    Console.WriteLine($" Last message sent at {lastTimestamp}.");

    await pipe.Reader.CompleteAsync();
});

await producerTask;
await consumerTask;

record Chunk(string Message, DateTime Timestamp);
```

All of this is serialized as JSON in the Pipe (formatted here for readability):

JSON

```json
[
    {
        "Message": "The quick brown fox",
        "Timestamp": "2025-08-01T18:37:27.2930151-07:00"
    },
    {
        "Message": " jumps over",
        "Timestamp": "2025-08-01T18:37:27.8594502-07:00"
    },
    {
        "Message": " the lazy dog.",
        "Timestamp": "2025-08-01T18:37:28.3753669-07:00"
    }
]
```

System.Numerics

- More left-handed matrix transformation methods
- Tensor enhancements

More left-handed matrix transformation methods

.NET 10 adds the remaining APIs for creating left-handed transformation matrices for billboard and constrained-billboard matrices. You can use these methods like their existing right-handed counterparts, for example, CreateBillboard(Vector3, Vector3, Vector3, Vector3), when using a left-handed coordinate system instead:

- Matrix4x4.CreateBillboardLeftHanded(Vector3, Vector3, Vector3, Vector3)
- Matrix4x4.CreateConstrainedBillboardLeftHanded(Vector3, Vector3, Vector3, Vector3, Vector3)

Tensor enhancements

The System.Numerics.Tensors namespace now includes a nongeneric interface, IReadOnlyTensor, for operations like accessing Lengths and Strides. Slice operations no longer copy data, which improves performance. Additionally, you can access data nongenerically by boxing to `object` when performance isn't critical.

The tensor APIs are now stable and no longer marked as experimental. While the APIs still require referencing the System.Numerics.Tensors NuGet package, they have been thoroughly reviewed and finalized for the .NET 10 release. The types take advantage of C# 14 extension operators to provide arithmetic operations when the underlying type `T` supports the operation. If `T` implements the relevant generic math interfaces, for example, `IAdditionOperators<TSelf, TOther, TResult>` or `INumber<TSelf>`, the operation is supported. For example, `tensor + tensor` is available for a `Tensor<int>`, but isn't available for a `Tensor<bool>`.

Options validation

- New AOT-safe constructor for ValidationContext

New AOT-safe constructor for `ValidationContext`

The ValidationContext class, used during options validation, includes a new constructor overload that explicitly accepts the `displayName` parameter:

ValidationContext(Object, String, IServiceProvider, IDictionary<Object,Object>)

The display name ensures AOT safety and enables its use in native builds without warnings.

Diagnostics

- Support for telemetry schema URLs in ActivitySource and Meter
- Out-of-proc trace support for Activity events and links
- Rate-limit trace-sampling support

Support for telemetry schema URLs in `ActivitySource` and `Meter`

ActivitySource and Meter now support specifying a telemetry schema URL during construction, which aligns with OpenTelemetry specifications. The telemetry schema ensures consistency and compatibility for tracing and metrics data. Additionally, .NET 10 introduces ActivitySourceOptions, which simplifies the creation of ActivitySource instances with multiple configuration options (including the telemetry schema URL).

The new APIs are:

- ActivitySource(ActivitySourceOptions)
- ActivitySource.TelemetrySchemaUrl
- Meter.TelemetrySchemaUrl
- ActivitySourceOptions

Out-of-proc trace support for Activity events and links

The Activity class enables distributed tracing by tracking the flow of operations across services or components. .NET supports serializing this tracing data out-of-process via the `Microsoft-Diagnostics-DiagnosticSource` event source provider. An `Activity` can include additional metadata such as ActivityLink and ActivityEvent. .NET 10 adds support for serializing these links and events, so out-of-proc trace data now includes that information. For example:

txt

```
Events->"[(TestEvent1,2025-03-27T23:34:10.6225721+00:00,[E11:EV1,E12:EV2]),
(TestEvent2,2025-03-27T23:34:11.6276895+00:00,[E21:EV21,E22:EV22])]"
Links->"[(19b6e8ea216cb2ba36dd5d957e126d9f,98f7abcb3418f217,Recorded,null,
false,[alk1:alv1,alk2:alv2]),(2d409549aadfdbdf5d1892584a5f2ab2,
4f3526086a350f50,None,null,false)]"
```

Rate-limit trace-sampling support

When distributed tracing data is serialized out-of-process via the `Microsoft-Diagnostics-DiagnosticSource` event source provider, all recorded activities can be emitted, or sampling can be applied based on a trace ratio.

A new sampling option called **Rate Limiting Sampling** restricts the number of *root activities* serialized per second. This helps control data volume more precisely.

Out-of-proc trace data aggregators can enable and configure this sampling by specifying the option in FilterAndPayloadSpecs . For example, the following setting limits serialization to 100 root activities per second across all `ActivitySource` instances:

txt

```
[AS]*/-ParentRateLimitingSampler(100)
```

ZIP files

- ZipArchive performance and memory improvements
- New async ZIP APIs
- Performance improvement in GZipStream for concatenated streams

ZipArchive performance and memory improvements

.NET 10 improves the performance and memory usage of ZipArchive.

First, the way entries are written to a `ZipArchive` when in `Update` mode has been optimized. Previously, all ZipArchiveEntry instances were loaded into memory and rewritten, which could lead to high memory usage and performance bottlenecks. The optimization reduces memory usage and improves performance by avoiding the need to load all entries into memory.

Second, the extraction of ZipArchive entries is now parallelized, and internal data structures are optimized for better memory usage. These improvements address issues related to performance bottlenecks and high memory usage, making `ZipArchive` more efficient and faster, especially when dealing with large archives.

New async ZIP APIs

.NET 10 introduces new asynchronous APIs that make it easier to perform non-blocking operations when reading from or writing to ZIP files. This feature was highly requested by the community.

New `async` methods are available for extracting, creating, and updating ZIP archives. These methods enable developers to efficiently handle large files and improve application responsiveness, especially in scenarios involving I/O-bound operations. These methods include:

- ZipArchive.CreateAsync(Stream, ZipArchiveMode, Boolean, Encoding, CancellationToken)
- ZipArchiveEntry.OpenAsync(CancellationToken)
- ZipFile.CreateFromDirectoryAsync
- ZipFile.ExtractToDirectoryAsync
- ZipFile.OpenAsync
- ZipFile.OpenReadAsync(String, CancellationToken)
- ZipFileExtensions.CreateEntryFromFileAsync
- ZipFileExtensions.ExtractToDirectoryAsync
- ZipFileExtensions.ExtractToFileAsync

For examples of using these APIs, see the Preview 4 blog post .

Performance improvement in GZipStream for concatenated streams

A community contribution improved the performance of GZipStream when processing concatenated GZip data streams. Previously, each new stream segment disposed and reallocated the internal `ZLibStreamHandle`, which resulted in additional memory allocations and initialization overhead. With this change, the handle is now reset and reused to reduce both managed and unmanaged memory allocations and improve execution time. The largest impact (~35% faster) is seen when processing a large number of small data streams. This change:

- Eliminates repeated allocation of ~64-80 bytes of memory per concatenated stream, with additional unmanaged memory savings.
- Reduces execution time by approximately 400 ns per concatenated stream.

Windows process management

Launch Windows processes in new process group

For Windows, you can now use ProcessStartInfo.CreateNewProcessGroup to launch a process in a separate process group. This allows you to send isolated signals to child processes that could otherwise take down the parent without proper handling. Sending signals is convenient to avoid forceful termination.

C#

```csharp
using System;
using System.Diagnostics;
using System.IO;
using System.Runtime.InteropServices;
using System.Threading;

class Program
{
    static void Main(string[] args)
    {
        bool isChildProcess = args.Length > 0 && args[0] == "child";
        if (!isChildProcess)
        {
            var psi = new ProcessStartInfo
            {
                FileName = Environment.ProcessPath,
                Arguments = "child",
                CreateNewProcessGroup = true,
            };

            using Process process = Process.Start(psi)!;
            Thread.Sleep(5_000);

            GenerateConsoleCtrlEvent(CTRL_C_EVENT, (uint)process.Id);
            process.WaitForExit();

            Console.WriteLine("Child process terminated gracefully, continue
with the parent process logic if needed.");
        }
        else
        {
            // If you need to send a CTRL+C, the child process needs to re-
enable CTRL+C handling, if you own the code, you can call
SetConsoleCtrlHandler(NULL, FALSE).
            // see https://learn.microsoft.com/windows/win32/api/pro-
cessthreadsapi/nf-processthreadsapi-createprocessw#remarks
            SetConsoleCtrlHandler((IntPtr)null, false);

            Console.WriteLine("Greetings from the child process!  I need to
be gracefully terminated, send me a signal!");

            bool stop = false;

            var registration =
PosixSignalRegistration.Create(PosixSignal.SIGINT, ctx =>
            {
                stop = true;
                ctx.Cancel = true;
                Console.WriteLine("Received CTRL+C, stopping...");
            });

            StreamWriter sw = File.AppendText("log.txt");
            int i = 0;
            while (!stop)
            {
                Thread.Sleep(1000);
                sw.WriteLine($"{++i}");
                Console.WriteLine($"Logging {i}...");
            }

            // Clean up
            sw.Dispose();
            registration.Dispose();

            Console.WriteLine("Thanks for not killing me!");
        }
    }

    private const int CTRL_C_EVENT = 0;
    private const int CTRL_BREAK_EVENT = 1;

    [DllImport("kernel32.dll", SetLastError = true)]
    [return: MarshalAs(UnmanagedType.Bool)]
    private static extern bool SetConsoleCtrlHandler(IntPtr handler, [Marsha-
lAs(UnmanagedType.Bool)] bool Add);

    [DllImport("kernel32.dll", SetLastError = true)]
    [return: MarshalAs(UnmanagedType.Bool)]
    private static extern bool GenerateConsoleCtrlEvent(uint dwCtrlEvent,
uint dwProcessGroupId);
}
```

WebSocket enhancements

WebSocketStream

.NET 10 introduces WebSocketStream, a new API designed to simplify some of the most common—and previously cumbersome—WebSocket scenarios in .NET.

Traditional `WebSocket` APIs are low-level and require significant boilerplate: handling buffering and framing, reconstructing messages, managing encoding/decoding, and writing custom wrappers to integrate with streams, channels, or other transport abstractions. These complexities make it difficult to use WebSockets as a transport, especially for apps with streaming or text-based protocols, or event-driven handlers.

`WebSocketStream` addresses these pain points by providing a Stream-based abstraction over a WebSocket. This enables seamless integration with existing APIs for reading, writing, and parsing data, whether binary or text, and reduces the need for manual plumbing.

`WebSocketStream` enables high-level, familiar APIs for common WebSocket consumption and production patterns. These APIs reduce friction and make advanced scenarios easier to implement.

Common usage patterns

Here are a few examples of how `WebSocketStream` simplifies typical `WebSocket` workflows:

Streaming text protocol (for example, STOMP)

C#

```csharp
using System.IO;
using System.Net.WebSockets;
using System.Threading;
using System.Threading.Tasks;

// Streaming text protocol (for example, STOMP).
using Stream transportStream = WebSocketStream.Create(
    connectedWebSocket,
    WebSocketMessageType.Text,
    ownsWebSocket: true);
// Integration with Stream-based APIs.
// Don't close the stream, as it's also used for writing.
using var transportReader = new StreamReader(transportStream, leaveOpen:
true);
var line = await transportReader.ReadLineAsync(cancellationToken); //
Automatic UTF-8 and new line handling.
transportStream.Dispose(); // Automatic closing handshake handling on
`Dispose`.
```

Streaming binary protocol (for example, AMQP)

C#

```
using System;
using System.IO;
using System.Net.WebSockets;
using System.Threading;
using System.Threading.Tasks;

// Streaming binary protocol (for example, AMQP).
Stream transportStream = WebSocketStream.Create(
    connectedWebSocket,
    WebSocketMessageType.Binary,
    closeTimeout: TimeSpan.FromSeconds(10));
await message.SerializeToStreamAsync(transportStream, cancellationToken);
var receivePayload = new byte[payloadLength];
await transportStream.ReadExactlyAsync(receivePayload, cancellationToken);
transportStream.Dispose();
// `Dispose` automatically handles closing handshake.
```

Read a single message as a stream (for example, JSON deserialization)

C#

```
using System.IO;
using System.Net.WebSockets;
using System.Text.Json;

// Reading a single message as a stream (for example, JSON deserialization).
using Stream messageStream =
WebSocketStream.CreateReadableMessageStream(connectedWebSocket,
WebSocketMessageType.Text);
// JsonSerializer.DeserializeAsync reads until the end of stream.
var appMessage = await JsonSerializer.DeserializeAsync<AppMessage>
(messageStream);
```

Write a single message as a stream (for example, binary serialization)

C#

```
using System;
using System.IO;
using System.Net.WebSockets;
using System.Threading;
using System.Threading.Tasks;

// Writing a single message as a stream (for example, binary serialization).
public async Task SendMessageAsync(AppMessage message, CancellationToken can-
cellationToken)
```

```
{
    using Stream messageStream =
WebSocketStream.CreateWritableMessageStream(_connectedWebSocket,
WebSocketMessageType.Binary);
    foreach (ReadOnlyMemory<byte> chunk in message.SplitToChunks())
    {
        await messageStream.WriteAsync(chunk, cancellationToken);
    }
} // EOM sent on messageStream.Dispose().
```

TLS enhancements

TLS 1.3 for macOS (client)

.NET 10 adds client-side TLS 1.3 support on macOS by integrating Apple's Network.framework into SslStream and HttpClient. Historically, macOS used Secure Transport which doesn't support TLS 1.3; opting into Network.framework enables TLS 1.3.

Scope and behavior

- macOS only, client-side in this release.
- Opt-in. Existing apps continue to use the current stack unless enabled.
- When enabled, older TLS versions (TLS 1.0 and 1.1) might no longer be available via Network.framework.

How to enable

Use an AppContext switch in code:

C#

```csharp
// Opt in to Network.framework-backed TLS on Apple platforms.
AppContext.SetSwitch("System.Net.Security.UseNetworkFramework", true);

using var client = new HttpClient();
var html = await client.GetStringAsync("https://example.com");
```

Or use an environment variable:

Bash

```bash
# Opt-in via environment variable (set for the process or machine as appro-
priate)
DOTNET_SYSTEM_NET_SECURITY_USENETWORKFRAMEWORK=1
```

```
# or
DOTNET_SYSTEM_NET_SECURITY_USENETWORKFRAMEWORK=true
```

Notes

- TLS 1.3 applies to SslStream and APIs built on it (for example, HttpClient/HttpMessageHandler).
- Cipher suites are controlled by macOS via Network.framework.
- Underlying stream behavior might differ when Network.framework is enabled (for example, buffering, read/write completion, cancellation semantics).
- Semantics might differ for zero-byte reads. Avoid relying on zero-length reads for detecting data availability.
- Certain internationalized domain names (IDN) hostnames might be rejected by Network.framework. Prefer ASCII/Punycode (A-label) hostnames or validate names against macOS/Network.framework constraints.
- If your app relies on specific SslStream edge-case behavior, validate it under Network.framework.

Note: The author created this article with assistance from AI. Learn more

Last updated on 11/07/2025