MAUI, EF 7, Minimal APIs, Blazor, C# 11

CODE

NOV 2022

codemag.com - THE LEADING INDEPENDENT DEVELOPER MAGAZINE - US $ 8.95 Can $ 11.95

CODE FOCUS

MAINTITLE HERE

C# Goes to 11

Sailing to MAUI

.NET 7 Performance Deep Dive

Azure is the best cloud for .NET

Build modern, scalable cloud apps on a cloud platform designed for .NET

More cloud services
Take your cloud app development farther using more than 100 Azure services that support .NET natively.

Visual Studio tools
Take advantage of integrated Visual Studio developer tools, get started faster with project templates, and be more productive with powerful debugging tools.

Faster, simpler development
Develop more easily with one-click deployment in Visual Studio or set up a CI/CD pipeline for your app in minutes.

Leverage a fully managed platform
Build, deploy, and scale your .NET app with Azure App Service or use powerful serverless compute with Azure Functions, without managing infrastructure, using the familiar Visual Studio IDE.

Build data-driven, intelligent apps
From image recognition to bot services, to databases, take advantage of Azure data services and artificial intelligence to create new experiences that scale.

Fix problems before your users notice
Intelligent application performance monitoring tools proactively alert you when there's a problem with your application. Built-in advanced diagnostics help you identify the root cause faster.

Start FREE at:

azure.microsoft.com/free/dotnet

Microsoft

Features

Departments

US subscriptions are US $29.99 for one year. Subscriptions outside the US pay $50.99 USD. Payments should be made in US dollars drawn on a US bank. American Express, MasterCard, Visa, and Discover credit cards are accepted. Bill Me option is available only for US subscriptions. Back issues are available. For subscription information, send e-mail to subscriptions@codemag.com or contact Customer Service at 832-717-4445 ext. 9.

Subscribe online at www.codemag.com

CODE Component Developer Magazine (ISSN # 1547-5166) is published bimonthly by EPS Software Corporation, 6605 Cypresswood Drive, Suite 425, Spring, TX 77379 U.S.A. POSTMASTER: Send address changes to CODE Component Developer Magazine, 6605 Cypresswood Drive, Suite 425, Spring, TX 77379 U.S.A.

LEAD Technologies

New in .NET 7

From the blazing beaches of MAUI, version 7 of .NET turns everything up to 11. How's that for an opening line? It seems to me only yesterday that we wrapped up the .NET 6 FOCUS issue of CODE Magazine and yet here we are, already spreading the news about .NET 7. And great news it is! For those of us who've been using Visual Studio and the .NET Framework for many years, it's never been better. The change to the Core platform has yielded results that many of us may still find surprising. Let's take a look at the wonderful gifts that .NET 7 gives us.

First, let talk about C#11. Each and every version of this language has presented developers with numerous productivity enhancements. One feature that I'm personally fond of is the addition of raw string literals. Channeling former CEO Steve Ballmer, developers in the year 2022 might chant: strings, strings, strings! JSON, YAML, XML, CSV strings are ever-present in today's development ecosystem and each version of C# makes this story that much better. You can now embed long strings into your code without resorting to loose text files on disk or embedded resources compiled into your application. This is just ONE of the features. Read the full article from Bill Wagner to find out about more wonderful additions to C#.

> Developers in the year 2022 might chant. "strings, strings, strings!"

Once you've whetted your whistle, set sail for the oasis of cross-platform development using the new MAUI features in .NET 7. MAUI was originally slated for shipment in .NET 6 but took more time to refine and improve than anticipated. The delay was worth the wait. The MAUI development platform helps .NET developers build applications capable of supporting macOS, Android, and Windows, all from a single code base. Check out David Ortinau's article on MAUI for details on building applications with this great tool.

.NET 7 continues "blazing" the trail of web assembly-compiled applications with updates to the Blazor framework. One great story is the capability of using components from other frameworks, like Angular, Vue, or React. Microsoft has adopted a philosophy of meeting developers where they are and your investment in these various tools is secure in the Blazor environment. You also get a better development experience with Hot Reload, better debugging, and improved interop. Blaze your own trail by looking at what Daniel Roth has written up for you!

> Microsoft has the philosophy of meeting developers where they are, to enhance their investments and ensure that their hard work is secure.

Performance, performance, performance... It seems that every version of .NET has radical performance improvements and .NET 7 continues this trend. In the software, every millisecond adds up and .NET 7 fights to remove every wasted cycle. Steven Toub presents the details on performance improvements and how they were achieved.

Data, data, data... The Entity Framework is 15 years old at this point and the EF team hasn't run out of cool features to surprise us with. Our resident EF expert Julie Lerman takes you through the numerous improvements to EF, including better performance, reduced rounds to the database server, enhancements to stored proc integration, changes to bulk updates, and many others. Building data-centric applications has never been better.

These are just a few highlights presented in this CODE FOCUS issue. Other topics include tools to help WCF developers as well as tips and tricks for converting your existing applications. Check them out—you won't be disappointed.

You may think that I'm being a bit over enthusiastic about this version of .NET. I don't think so. I've been a.NET developer since its first beta, and this version has done nothing but confirm my decision some 20 years back. Let me tell you the major reason why: .NET has a great backward-compatibility story along with a forward-thinking trajectory, ensuring that your sweat investment is preserved with minimal rework. This version is no different.

I hope you enjoy what we've created for you!

Rod Paddock
CODE

What's New in .NET 7

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

Jon Douglas
@_jondouglas
jonathan.douglas@microsoft.com
https://devblogs.microsoft.com/dotnet/author/jondouglas/

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

Jeremy Likness
@JeremyLikness
jeremy.likness@microsoft.com
https://devblogs.microsoft.com/dotnet/author/jeremy-likness/

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

Angelos Petropoulos
@apetrop
angelos.petropoulos@microsoft.com
https://devblogs.microsoft.com/dotnet/author/angelos-petropoulos/

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

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

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

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

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

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

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

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

Targeting .NET 7

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

```
<TargetFramework>net7.0</TargetFramework>
```

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

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

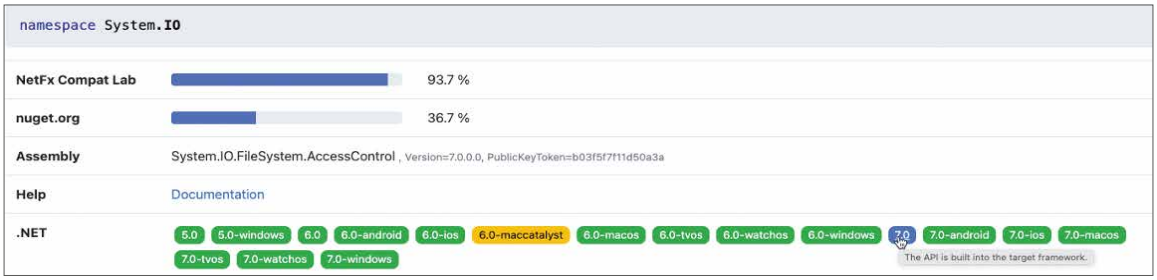

Figure 1: The .NET API catalog

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

.NET MAUI

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

New Features

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

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

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

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

Performance

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

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

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

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

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

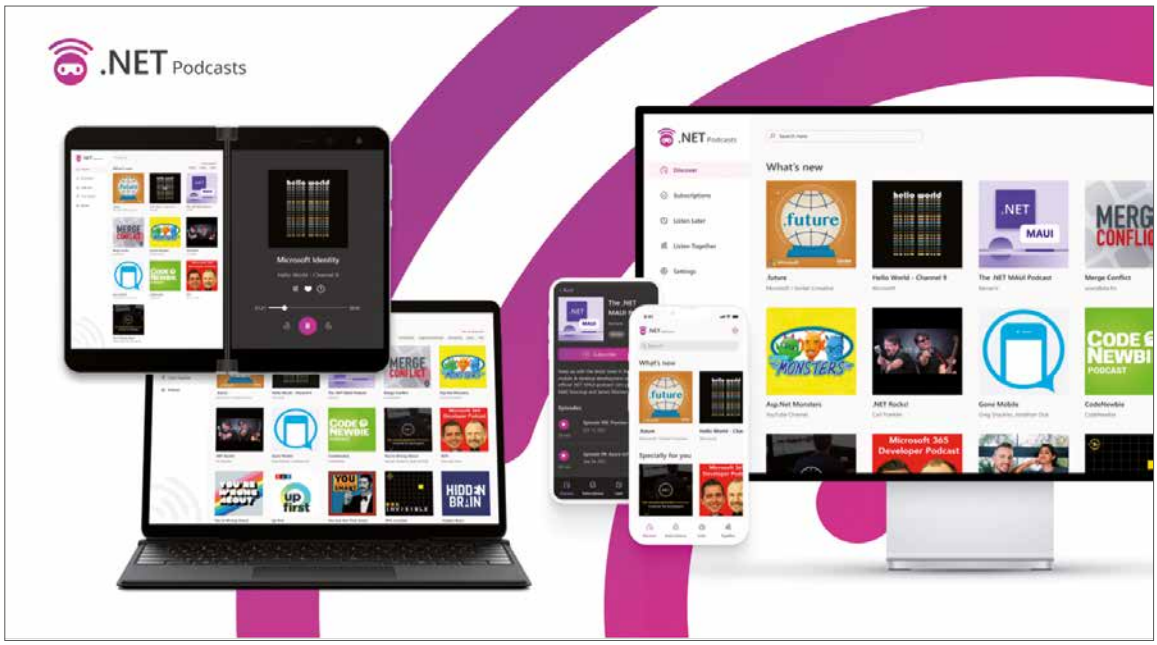

Figure 2: The .NET podcasts MAUI reference application

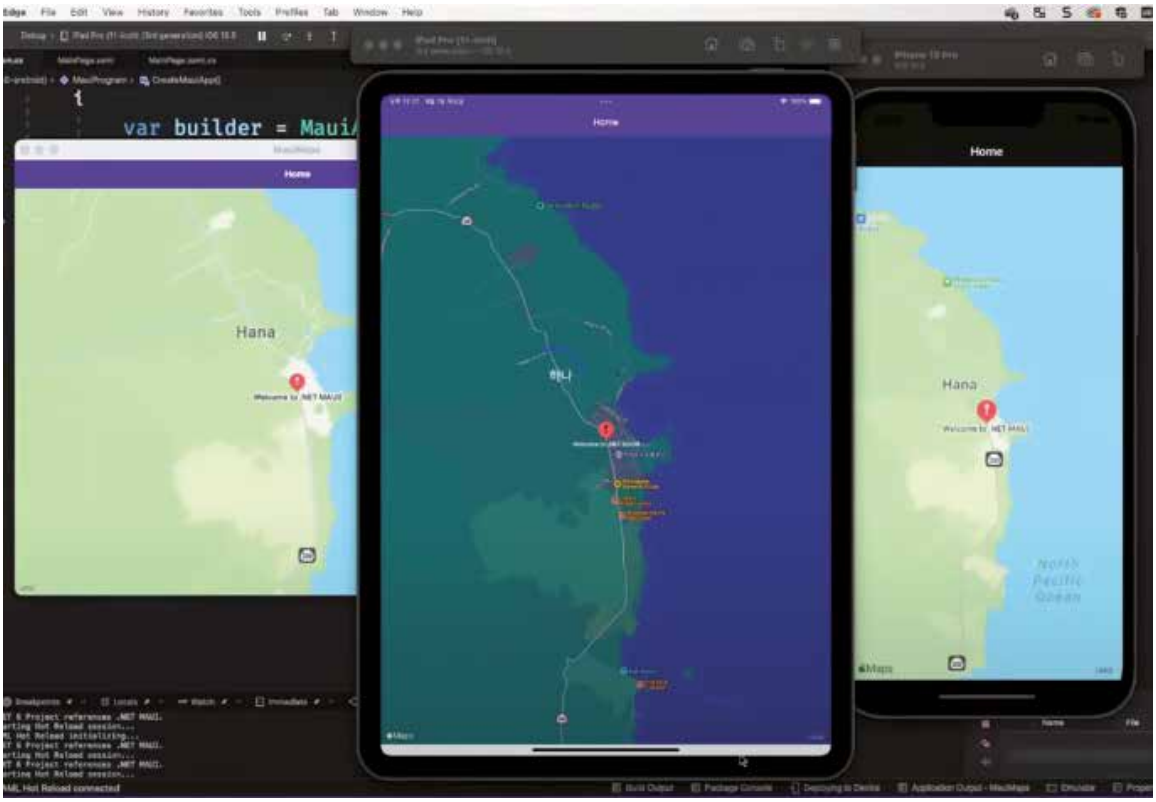

Figure 3: New Map control for displaying and annotating native maps for Android, iOS, and iPadOS applications

Cloud Native

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

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

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

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

Built-in Container Support

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

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

```
# create a new project and move to its directory
dotnet new mvc -n my-awesome-container-app
cd my-awesome-container-app

# add a reference to a (temporary) package that creates the
container
dotnet add package Microsoft.NET.Build.Containers

# publish your project for linux-x64
dotnet publish --os linux --arch x64 -p:PublishProfile=Defaul
tContainer
```

We built this solution with the following goals:

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

Microsoft Orleans

Microsoft Orleans takes familiar concepts like objects, interfaces, async/await, and try/catch extends them to multi-

server environments. It helps developers experienced with single-server applications transition to building resilient, scalable cloud services and other distributed applications. For this reason, Orleans has often been referred to as **Distributed .NET**.

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

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

Observability

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

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

Introducing Activity.Current Change Event

A typical implementation of distributed tracing uses an **AsyncLocal<T>** to track the "span context" of managed threads. Changes to the span context are tracked by using the **AsyncLocal<T>** constructor that takes the valueChangedHandler parameter. However, with Activity becoming the standard to represent spans as used by OpenTelemetry, it's impossible to set the value-changed handler because the context is tracked via Activity.Current. The new change event can be used instead to receive the desired notifications.

```
Activity.CurrentChanged += CurrentChanged;

void CurrentChanged(object? sender,
ActivityChangedEventArgs e)
    {
                        Console.WriteLine($"Activity.Current
                value changed from Activity:
                {e.Previous.OperationName} to
                Activity: {e.Current.OperationName}");
    }
```

Expose Performant Activity Properties Enumerator Methods

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

```
    Activity a = new Activity("Root");
    a.SetTag("key1", "value1");
    a.SetTag("key2", "value2");

    foreach (ref readonly
KeyValuePair<string, object?> tag in
 a.EnumerateTagObjects())
    {
        Console.WriteLine($"{tag.Key},
{tag.Value}");
    }
```

Expose Performant ActivityEvent and ActivityLink Tags Enumerator Methods

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

```
var tags = new List<KeyValuePair<string,
object?>>()
{
    new KeyValuePair<string,
object?>("tag1", "value1"),
    new KeyValuePair<string,
object?>("tag2", "value2"),
};

ActivityLink link = new
ActivityLink(default, new
ActivityTagsCollection(tags));

foreach (ref readonly KeyValuePair<string,
object?> tag in link.EnumerateTagObjects())
{
    // Consume the link tags without any
extra allocations or value copying.
}

ActivityEvent e = new
ActivityEvent("SomeEvent", tags: new
ActivityTagsCollection(tags));

foreach (ref readonly KeyValuePair<string,
object?> tag in e.EnumerateTagObjects())
{
    // Consume the event's tags without any
extra allocations or value copying.
}
```

Modernization

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

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

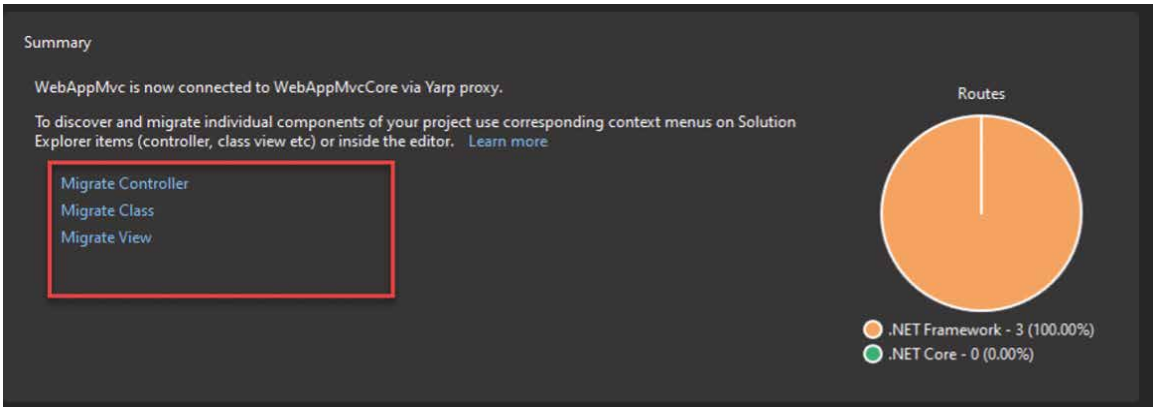

Figure 5: The Visual Studio ASP.NET MVC migration experience

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

ASP.NET to ASP.NET Core Migration

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

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

ARM64

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

Runtime Improvements

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

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

Table 2: L3 cache size per machine core count

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

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

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

Library Improvements

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

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

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

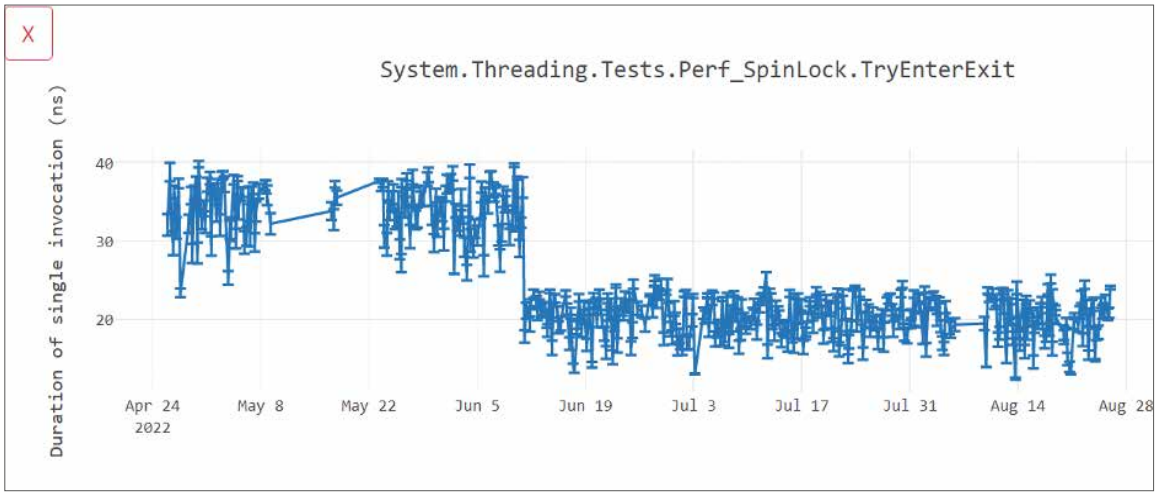

Figure 6: LSE atomics performance enhancements on Windows for lock scenarios

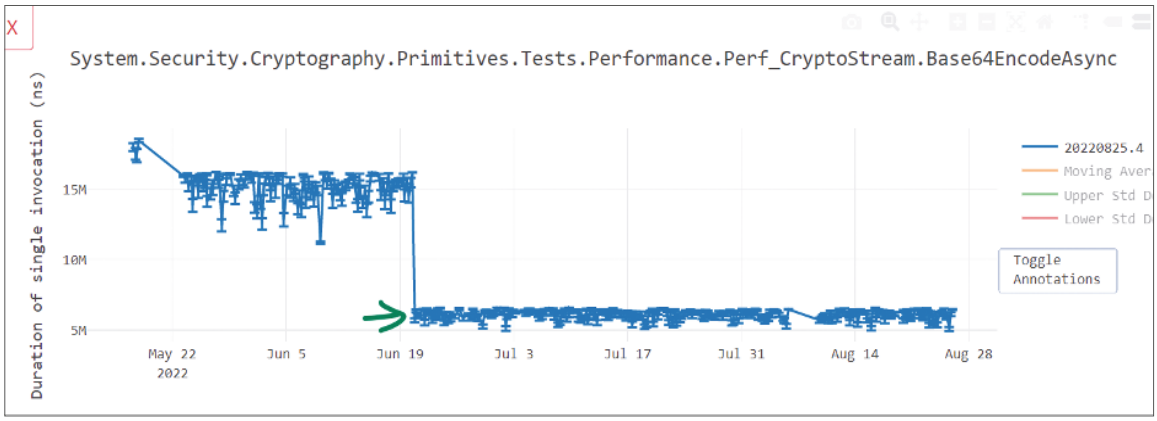

Figure 7: Text processing improvements with Vector-based implementations

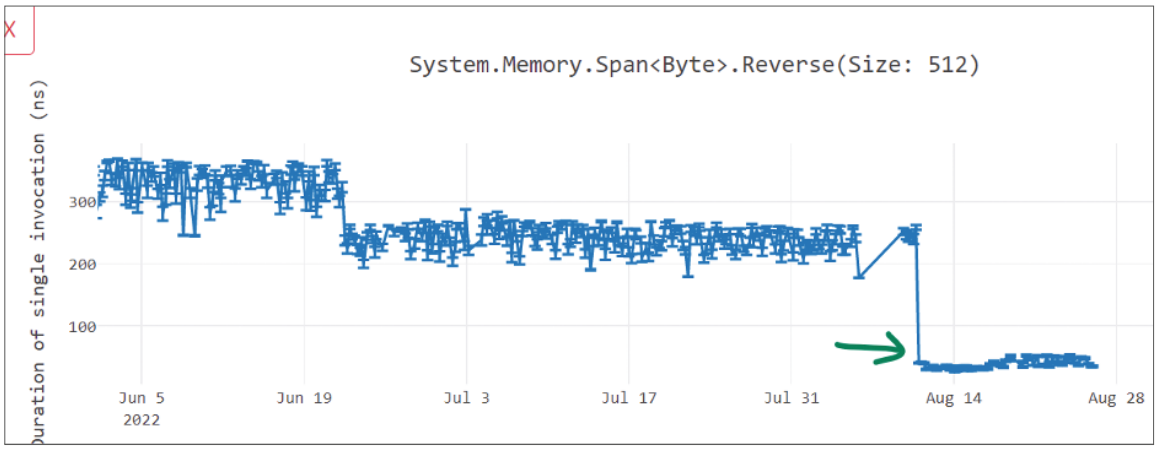

Figure 8: Span<Byte>.Reverse() improvements with Vector-based implementations

Performance Impact

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

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

C# 11

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

Generic Math

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

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

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

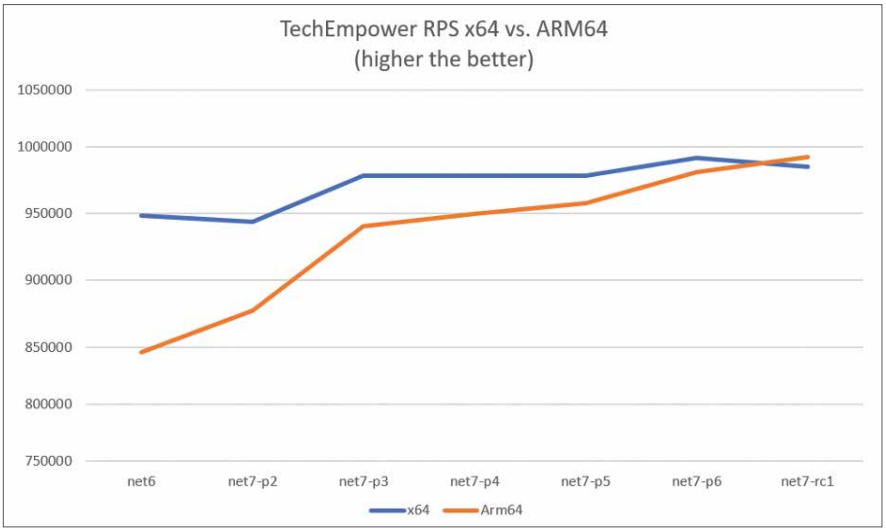

Figure 9: TechEmpower RPS x64 vs. ARM64 (higher the better) benchmark

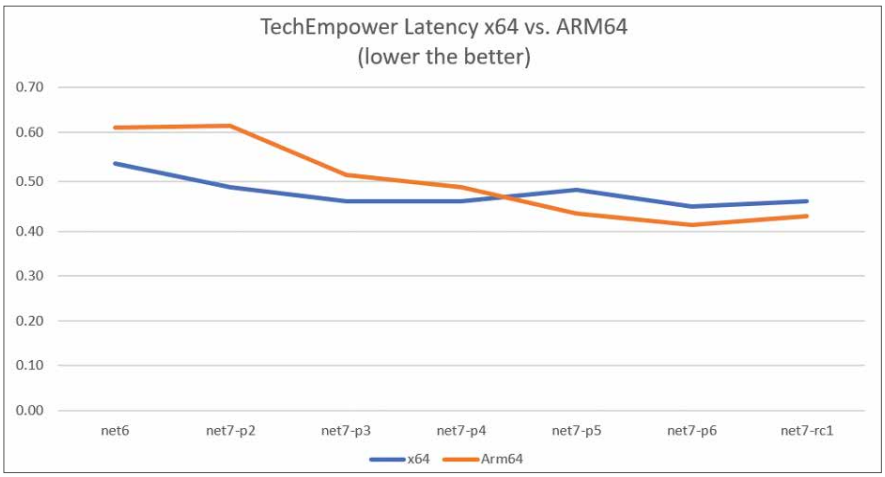

Figure 10: TechEmpower x64 vs. ARM64 latency (the lower the better)

C# type/keyword	Range	Size	.NET type
sbyte	-128 to 127	Signed 8-bit integer	System.SByte
byte	0 to 255	Unsigned 8-bit integer	System.Byte
short	-32,768 to 32,767	Signed 16-bit integer	System.Int16
ushort	0 to 65,535	Unsigned 16-bit integer	System.UInt16
int	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer	System.Int32
uint	0 to 4,294,967,295	Unsigned 32-bit integer	System.UInt32
long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	System.Int64
ulong	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer	System.UInt64
nint	Depends on platform (computed at runtime)	Signed 32-bit or 64-bit integer	System.IntPtr
nuint	Depends on platform (computed at runtime)	Unsigned 32-bit or 64-bit integer	System.UIntPtr

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

Raw String Literals

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

```
String longMessage = """
    This is a long message.
    It has several lines.
        Some are indented
                more than others.
    Some should start at the first column.
    Some have "quoted text" in them.
    """;
```

Numeric IntPtr and UintPtr

The nint and nuint types now alias System.IntPtr and System.UintPtr, respectively. These are two native-sized integers that depend on the platform and is computed at runtime. **Figure 11** shows the various C# types/keywords, including nint and nuint.

.NET Libraries

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

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

Nullable annotations for Microsoft.Extensions

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

System.Text.Json Polymorphism

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

```
[JsonDerivedType(typeof(Derived))]
public class Base
{
    public int X { get; set; }
}

public class Derived : Base
```

Listing 1: Mathematical operators on generic types

```
public static TResult Sum<T, TResult>(IEnumerable<T> values)
    where T : INumber<T>
    where TResult : INumber<TResult>
{
    TResult result = TResult.Zero;

    foreach (var value in values)
    {
        result += TResult.Create(value);
    }

    return result;
}

public static TResult Average<T, TResult>(IEnumerable<T> values)
    where T : INumber<T>
    where TResult : INumber<TResult>
{
    TResult sum = Sum<T, TResult>(values);
    return TResult.Create(sum) / TResult.Create(values.Count());
}
```

```
public static TResult StandardDeviation<T, TResult>(IEnumerable<T> values)
    where T : INumber<T>
    where TResult : IFloatingPoint<TResult>
{
    TResult standardDeviation = TResult.Zero;

    if (values.Any())
    {
        TResult average = Average<T, TResult>(values);
        TResult sum = Sum<TResult, TResult>(values.Select((value) => {
            var deviation = TResult.Create(value) - average;
            return deviation * deviation;
        }));
        standardDeviation = TResult.Sqrt(sum
/ TResult.Create(values.Count() - 1));
    }

    return standardDeviation;
}
```

```
{
    public int Y { get; set; }
}
```

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

```
Base value = new Derived();
JsonSerializer.Serialize<Base>(value); // { "X" :
0, "Y" : 0 }
```

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

```
Base value = JsonSerializer.Deserialize<Base>(@"{
""X"" : 0, ""Y"" : 0 }");
value is Derived; // false
```

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

```
[JsonDerivedType(typeof(Base), typeDiscriminator:
"base")]
[JsonDerivedType(typeof(Derived),
 typeDiscriminator: "derived")]
public class Base
{
    public int X { get; set; }
}

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

This now emits JSON along with type discriminator metadata:

```
Base value = new Derived();
JsonSerializer.Serialize<Base>(value); // {
"$type" : "derived", "X" : 0, "Y" : 0 }
```

That can be used to deserialize the value polymorphically:

```
Base value = JsonSerializer.Deserialize<Base>(@"{
 ""$type"" : ""derived"", ""X"" : 0, ""Y"" : 0 }");
value is Derived; // true
```

System.Composition.Hosting

A new API has been added to allow a single object instance to the System.Composition.Hosting container providing similar functionality to the legacy interfaces as System.ComponentModel.Composition.Hosting through the API **ComposeExportedValue(CompositionContainer, T).**

```
namespace System.Composition.Hosting
{
    public class ContainerConfiguration
    {
        public ContainerConfiguration
 WithExport<TExport>(TExport
exportedInstance);
        public ContainerConfiguration
 WithExport<TExport>(TExport
exportedInstance, string contractName =
null, IDictionary<string, object> metadata =
null);

        public ContainerConfiguration
WithExport(Type contractType, object
exportedInstance);
        public ContainerConfiguration
WithExport(Type contractType, object
exportedInstance, string contractName =
null, IDictionary<string, object> metadata =
null);
    }
}
```

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

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

Microsoft.Extensions.Caching

We added metrics support for **IMemoryCache**, which is a new API of **MemoryCacheStatistics** that holds cache hits, misses, and estimated size for **IMemoryCache**. You can get

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

```
namespace System {
    public struct DateTime {
        public DateTime(int year, int month,
int day, int hour, int minute, int second,
int millisecond, int microsecond);
        public DateTime(int year, int month,
int day, int hour, int minute, int second,
int millisecond, int microsecond,
System.DateTimeKind kind);
        public DateTime(int year, int month,
int day, int hour, int minute, int second,
int millisecond, int microsecond,
 System.Globalization.Calendar calendar);
        public int Microsecond { get; }
        public int Nanosecond { get; }
        public DateTime
AddMicroseconds(double value);
    }
    public struct DateTimeOffset {
        public DateTimeOffset(int year, int
month, int day, int hour, int minute, int
second, int millisecond, int microsecond,
 System.TimeSpan offset);
        public DateTimeOffset(int year, int
month, int day, int hour, int minute, int
second, int millisecond, int microsecond,
System.TimeSpan offset,
System.Globalization.Calendar calendar);
        public int Microsecond { get; }
        public int Nanosecond { get; }
        public DateTimeOffset
AddMicroseconds(double microseconds);
    }
    public struct TimeSpan {
        public const long
TicksPerMicrosecond = 10L;
        public const long NanosecondsPerTick
= 100L;
        public TimeSpan(int days, int hours,
int minutes, int seconds, int milliseconds,
int microseconds);
        public int Microseconds { get; }
        public int Nanoseconds { get; }
        public double TotalMicroseconds {
get; }
        public double TotalNanoseconds {
get; }
        public static TimeSpan
FromMicroseconds(double microseconds);
    }
    public struct TimeOnly {
        public TimeOnly(int hour, int
minute, int second, int millisecond, int
microsecond);
        public int Microsecond { get; }
        public int Nanosecond { get; }
    }
}
```

Listing 3: Get started with MemoryCacheStatistics

```
// when using
`services.AddMemoryCache(options =>
options.TrackStatistics = true);` to
instantiate

    [EventSource(Name = "Microsoft-Extensions-Caching-Memory")]
    internal sealed class CachingEventSource
: EventSource
    {
        public CachingEventSource(IMemoryCache memoryCache)
{ _memoryCache = memoryCache; }
        protected override void
OnEventCommand(EventCommandEventArgs
command)
        {
            if (command.Command ==
EventCommand.Enable)
            {
                if (_cacheHitsCounter ==
null)
                {
                    _cacheHitsCounter = new
PollingCounter("cache-hits", this, () =>
                        _memoryCache.GetCurrentStatistics().CacheHits)
                    {
                        DisplayName = "Cache
hits",
                    };
                }
            }
        }
    }
```

an instance of **MemoryCacheStatistics** by calling **GetCurrentStatistics**() when the flag **TrackStatistics** is enabled.

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

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

```
Press p to pause, r to resume, q to quit.
    Status: Running

[System.Runtime]
    CPU Usage (%)                                  0
    Working Set (MB)                              28
[Microsoft-Extensions-Caching-MemoryCache]
    cache-hits                                   269
```

System.Formats.Tar APIs

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

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

Type Converters

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

```
namespace System.ComponentModel
{
    public class DateOnlyConverter :
System.ComponentModel.TypeConverter
    {
        public DateOnlyConverter() { }
    }

    Public class TimeOnlyConverter :
System.ComponentModel.TypeConverter
    {
        public TimeOnlyConverter() { }
    }

    public class Int128Converter :
System.ComponentModel.BaseNumberConverter
    {
        public Int128Converter() { }
    }
```

```
    public class UInt128Converter :
System.ComponentModel.BaseNumberConverter
    {
        public UInt128Converter() { }
    }

    public class HalfConverter :
System.ComponentModel.BaseNumberConverter
    {
        public HalfConverter() { }
    }
}
```

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

```
TypeConverter dateOnlyConverter =
TypeDescriptor.GetConverter(typeof(DateOnly)
);
// produce DateOnly value of DateOnly(1940,
10, 9)
DateOnly? date =
dateOnlyConverter.ConvertFromString("1940-
10-09") as DateOnly?;

TypeConverter timeOnlyConverter =
TypeDescriptor.GetConverter(typeof(TimeOnly)
);
// produce TimeOnly value of TimeOnly(20,
30, 50)
TimeOnly? time =
timeOnlyConverter.ConvertFromString("20:30:5
0") as TimeOnly?;

TypeConverter halfConverter =
TypeDescriptor.GetConverter(typeof(Half));
// produce Half value of -1.2
Half? half =
halfConverter.ConvertFromString(((Half)(-
1.2)).ToString()) as Half?;

TypeConverter Int128Converter =
TypeDescriptor.GetConverter(typeof(Int128));
// produce Int128 value of Int128.MaxValue
which equal
170141183460469231731687303715884105727
Int128? int128 =
Int128Converter.ConvertFromString("170141183
460469231731687303715884105727") as Int128?;

TypeConverter UInt128Converter =
TypeDescriptor.GetConverter(typeof(UInt128))
;
// produce UInt128 value of UInt128.MaxValue
Which equal
340282366920938463463374607431768211455
UInt128? uint128 =
UInt128Converter.ConvertFromString("34028236
692093846346337460743176821145 5") as
UInt128?;
```

JSON Contract Customization

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

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

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

```
JsonSerializerOptions options = new()
{
    TypeInfoResolver = new
DefaultJsonTypeInfoResolver()
    {
        Modifiers =
        {
            (JsonTypeInfo jsonTypeInfo) =>
            {
                // your modifications here, i.e.:
                if (jsonTypeInfo.Type == typeof(int))
                {
                    jsonTypeInfo.NumberHandling =
JsonNumberHandling.AllowReadingFromString;
                }
            }
        }
    }
};

Point point =
JsonSerializer.Deserialize<Point>(@"{""X"":"
"12"",""Y"":""3""}", options);
Console.WriteLine($"({point.X},{point.Y})");
// (12,3)

public class Point
{
    public int X { get; set; }
    public int Y { get; set; }
}
```

Listing 4: Using System.Formats.Tar APIs

```
// Generates a tar archive where all the
entry names are prefixed by the root
directory 'SourceDirectory'
TarFile.CreateFromDirectory(sourceDirectoryN
ame: "/home/dotnet/SourceDirectory/",
destinationFileName:
"/home/dotnet/destination.tar",
includeBaseDirectory: true);

// Extracts the contents of a tar archive
into the specified directory, but avoids
overwriting anything found inside
TarFile.ExtractToDirectory(sourceFileName:
"/home/dotnet/destination.tar",
destinationDirectoryName:
"/home/dotnet/DestinationDirectory/",
overwriteFiles: false);
```

.NET SDK

The .NET SDK continues to add new features to make you more productive than ever. In .NET 7, we improve your expe-

riences with the .NET CLI, authoring templates, and managing your packages in a central location.

CLI Parser and Tab Completion

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

Here's the new help output as an example:

```
> dotnet new --help
Description:
  Template Instantiation Commands for .NET
CLI.

Usage:
  dotnet new [<template-short-name>
[<template-args>...]] [options]
  dotnet new [command] [options]

Arguments:
  <template-short-name>  A short name of the
template to create.
  <template-args>        Template specific
options to use.

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

Commands:
  install <package>       Installs a
template package.
  uninstall <package>     Uninstalls a
template package.
  update                  Checks the
currently installed template packages for
update, and install the updates.
  search <template-name>  Searches for the
templates on NuGet.org.
  list <template-name>    Lists templates
containing the specified template name. If
no name is specified, lists all templates.
```

```
❯ dotnet new angular
angular              grpc                 razor                viewstart
blazorserver         mstest               razorclasslib        web
blazorwasm           mvc                  razorcomponent       webapi
classlib             nugetconfig          react                webapp
console              nunit                reactredux           webconfig
editorconfig         nunit-test           sln                  winforms
gitignore            page                 tool-manifest        winformscontrollib
globaljson           proto                viewimports          winformslib
```

Figure 12: The dotnet new command provides tab completion.

```
❯ dotnet new web --dry-run
--dry-run                  --language               --output                 -la
--exclude-launch-settings  --name                   --type                   -n
--force                    --no-https               -?                       -o
--framework                --no-restore             -f                       /?
--help                     --no-update-check        -h                       /h
```

Figure 13: The options and arguments that are available

```
❯ dotnet new blazorserver --auth Individual
Individual     IndividualB2C  MultiOrg       None           SingleOrg      Windows
```

Figure 14: Supported options in the .NET CLI

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

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

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

Template Authoring

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

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

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

```
"constraints": {
      "web-assembly": {
          "type": "workload",
          "args": "wasm-tools"
      },
  }
```

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

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

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

Central Package Management

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

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

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

```
<Project>
  <PropertyGroup>
    <ManagePackageVersionsCentrally>true</ManagePackageVersionsCentrally>
  </PropertyGroup>

  <ItemGroup>
    <PackageVersion Include="Newtonsoft.Json" Version="13.0.1" />
  </ItemGroup>
</Project>
```

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

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

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Newtonsoft.Json" />
  </ItemGroup>
</Project>
```

Performance

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

On Stack Replacement (OSR)

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

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

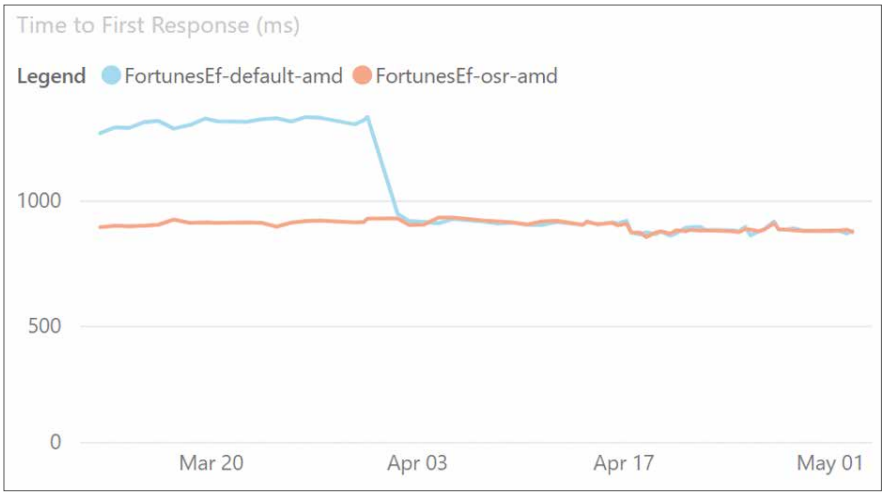

Figure 15: Time to first response when OSR is enabled

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

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

Profile-Guided Optimization (PGO)

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

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

Dynamic PGO takes advantage of tiered compilation. The JIT instruments the tier-0 code to track how many times the method is called, or in the case of loops, how many times the loop executes. Tiered compilation can instrument a variety of possibilities. For example, it can track exactly which concrete types are used as the target of an interface dispatch, and then, in tier-1, specialize the code to expect the most common types (this is referred to as "guarded devir-

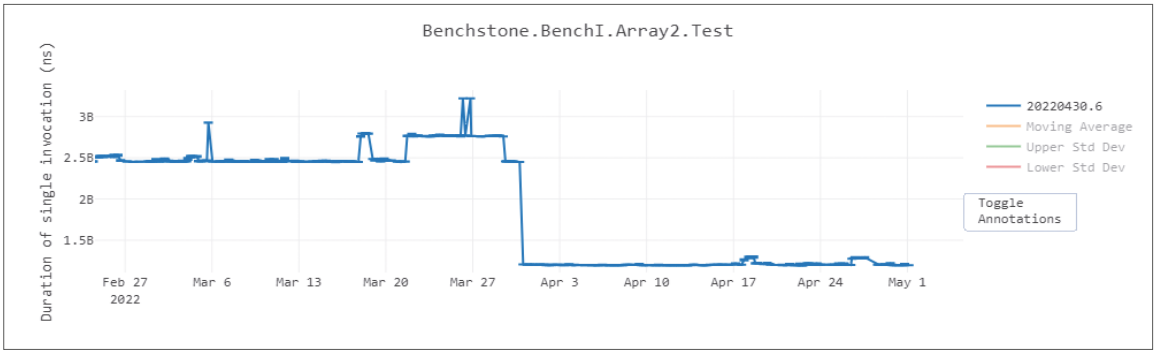

Figure 16: The Array2 microbenchmark with OSR enabled

tualization," or GDV). You can see this in this little example. Set the **DOTNET_TieredPGO** environment variable to **1**, and then run it on .NET 7:

```
class Program
{
    static void Main()
    {
        IPrinter printer = new Printer();
        for (int i = 0; ; i++)
        {
            DoWork(printer, i);
        }
    }

    static void DoWork(IPrinter printer, int
i)
    {
        printer.PrintIfTrue(I == int.MaxValue);
    }

    interface iPrinter
    {
        void PrintIfTrue(bool condition);
    }

    class Printer : iPrinter
    {
        public void PrintIfTrue(bool condition)
        {
            if (condition)
Console.WriteLine"Print"");
        }
    }
}
```

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

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

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

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

Native AOT

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

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

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

The binary that results from publishing a build is a completely standalone executable in the target platform's platform-specific file format (e.g., COFF on Windows, ELF on Linux, Mach-O on macOS) with no external dependencies other than that one is standard to that platform (e.g., libc). And it's entirely native: no IL in sight, no JIT, no nothing.

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

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

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

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

All required code is compiled and/or linked into the executable, including the same GC that's used with standard .NET apps and services, and a minimal runtime that provides services around threading and the like.

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

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

```
dotnet new console -o nativeaotexample
```

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

```
<PublishAot>true</PublishAot>
```

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

```
dotnet publish -r win-x64 -c Release
```

This generates an executable in the output publish directory:

```
    Directory:
C:\nativeaotexample\bin\Release\net7.0\win-
x64\publish

Mode                 LastWriteTime         Length
Name
----                 -------------         ------
----
-a---           8/27/2022  6:18 PM        3648512
nativeaotexample.exe
-a---           8/27/2022  6:18 PM       14290944
nativeaotexample.pdb
```

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

Summary

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

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

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

Jon Douglas, Jeremy Likness, and Angelos Petropoulos

Listing 5: Tier-0 code for DoWork

```
G_M000_IG01:                ;; offset=0000H
       55                   push     rbp
       4883EC30             sub      rsp, 48
       488D6C2430           lea      rbp, [rsp+30H]
       33C0                 xor      eax, eax
       488945F8             mov      qword ptr [rbp-08H], rax
       488945F0             mov      qword ptr [rbp-10H], rax
       48894D10             mov      gword ptr [rbp+10H], rcx
       895518               mov      dword ptr [rbp+18H], edx

G_M000_IG02:                ;; offset=001BH
       FF059F220F00         inc      dword ptr [(reloc 0x7ffc3f1b2ea0)]
       488B4D10             mov      rcx, gword ptr [rbp+10H]
       48894DF8             mov      gword ptr [rbp-08H], rcx
       488B4DF8             mov      rcx, gword ptr [rbp-08H]
       48BAA82E1B3FFC7F0000 mov      rdx, 0x7FFC3F1B2EA8
       E8B47EC55F           call     CORINFO_HELP_CLASSPROFILE32
       488B4DF8             mov      rcx, gword ptr [rbp-08H]
       48894DF0             mov      gword ptr [rbp-10H], rcx
       488B4DF0             mov      rcx, gword ptr [rbp-10H]
       33D2                 xor      edx, edx
       817D18FFFFFF7F       cmp      dword ptr [rbp+18H], 0x7FFFFFFF
       0F94C2               sete     dl
       49BB0800F13EFC7F0000 mov      r11, 0x7FFC3EF10008
       41FF13               call     [r11]iPrinter:PrintIfTrue(bool):this
       90                   nop

G_M000_IG03:                ;; offset=0062H
       4883C430             add      rsp, 48
       5D                   pop      rbp
       C3                   ret
```

Listing 6: Tier-1 code for DoWork

```
G_M000_IG02:                ;; offset=0020H
       48B9982D1B3FFC7F0000 mov      rcx,
0x7FFC3F1B2D98
       48390F               cmp      qword
ptr [rdi], rcx
       7521                 jne      SHORT
G_M000_IG05
       81FEFFFFFF7F         cmp      esi,
0x7FFFFFFF
       7404                 je       SHORT
G_M000_IG04

G_M000_IG03:                ;; offset=0037H
       FFC6                 inc      esi
       EBE5                 jmp      SHORT
G_M000_IG02

G_M000_IG04:                ;; offset=003BH
       48B9D820801A24020000 mov      rcx,
0x2241A8020D8
       488B09               mov      rcx,
gword ptr [rcx]
       FF1572CD0D00         call
[Console:WriteLine(String)]
       EBE7                 jmp      SHORT
G_M000_IG03
```

What's New in C# 11

Think about the new features in C# 11 as organized around a small set of themes: improved developer productivity, object initialization and creation, generic math support, and runtime performance. You'll benefit from all these features, even if you'll likely use the features in the first two themes far more often than the last two themes. This article discusses the reasons for the new features, provides examples of when you'll use them, and points to the benefits your programs get because the .NET runtime makes use of these features.

Bill Wagner
wiwagn@microsoft.com
@billwagner

Bill is a member of the .NET Content team at Microsoft, responsible for docs and learning content covering the C# language. He's also a member of the C# ECMA standards committee.

Improved Developer Productivity

C# 11 adds many features that improve your productivity. You'll use these features to write more concise and more readable code:

- Raw string literals
- Newlines in string interpolations
- UTF-8 string literals
- Pattern match Span<char> or ReadOnlySpan<char> on a constant struct
- List patterns

Let's start with making strings easier to manipulate. **Raw string literals** provide new syntax, making it easier to embed arbitrary text, including whitespace, new lines, embedded quotes, and other special characters without requiring escape sequences. A raw string literal starts with at least three double-quote (""") characters. It ends with the same number of double-quote characters. Typically, a raw string literal uses three double quotes on a single line to start the string, and three double quotes on a separate line to end the string. The **newlines** following the opening quotes and preceding the closing quotes aren't included in the final content:

```
string longMessage = """
    This is a long message.
    It has several lines.
        Some are indented
                more than others.
    Some should start at the first column.
    Some have "quoted text" in them.
    """;
```

> Raw string literals provide new syntax, making it easier to embed arbitrary text, including whitespace, new lines, embedded quotes, and other special characters, without requiring escape sequences.

Any whitespace to the left of the closing triple quotes will be removed from the string literal. Raw string literals can be combined with string interpolation to include braces in the output text. Multiple $ characters denote how many consecutive braces start and end the interpolation:

```
var location = $$"""
   You are at {{{Longitude}}, {{Latitude}}}
   """;
```

The preceding example specifies that two braces start and end an interpolation. The third repeated opening and closing brace are included in the output string.

You can use multiple $ characters in an interpolated raw string literal to embed { and } characters in the output string without escaping them. The following snippet shows an example where $$ indicates that two {{ and }} characters open and close an interpolated expression. The third consecutive { or } is added to the output string.

```
int X = 2;
int Y = 3;

var pointMessage = $$"""
    The point {{{X}}, {{Y}}} is {{Math.Sqrt(
    X * X + Y * Y)}} from the origin
    """;
Console.WriteLine(pointMessage);
// output:
// The point {2, 3} is 3.60555 from the origin.
```

The preceding example also demonstrates newlines in interpolated strings. The C# expressions embedded in an interpolated string can span multiple source code lines. You can format lengthy expressions such as LINQ queries or pattern matching switch expressions directly inside a string interpolation:

```
string message = $"The usage policy for
    {safetyScore} is {
    safetyScore switch
    {
        > 90 => "Unlimited usage",
        > 80 => "General usage, with safety check",
        > 70 => "Issues must be addressed < 1 week",
        > 50 => "Issues must be addressed today",
        _ => "Issues must be addressed now",
    }
    }";
```

Think of all the code you write where you format strings that include quote characters: LINQ query output, XML output, JSON output, and more. The new raw string literals will make it much easier to format these strings in a way that's easier for developers to read.

Developers who work directly with web standards or other data protocols that use UTF-8 string will appreciate **UTF-8 string literals**. Add the **u8** suffix on a string literal, and the

compiler interprets it as a UTF-8-encoded string. The string literal is stored as a ReadOnlySpan<byte>. The UTF-8 string literal can be used with .NET library APIs that require UTF-8 encoded strings. This feature creates a natural syntax for working with UTF-8 strings, providing more readable and more performant code constructs when you use UTF-8 strings.

Pattern matching expressions get an improvement for working with strings and string literals. You can now pattern-match a Span<char> or ReadOnlySpan<char> against a string literal. The compiler generates code to perform the match test without allocating and copying the span or the string.

Finally, **List patterns** extend pattern matching syntax to match the sequences of elements in a list or an array. Any pattern can be applied to any element in the list to check whether an individual element matches certain characteristics. The **discard** pattern **(_)** matches any single element. The **range** pattern **(..)** matches zero or more elements in the sequence. At most, one range pattern is allowed in a list pattern. The var pattern can capture any single element, or a range of elements. You can see several examples of list patterns in **Listing 1**.

List patterns provide a rich syntax to text the shape of sequences to determine whether a sequence contains elements that match required traits, and to ensure that those elements are in the proper location.

You'll often use these new features to write code that is more expressive, more concise, and more understandable. The new syntax makes working with strings and related data structures easier.

Object Initialization and Literals

Another goal for C# is to make it easier to initialize new objects or values correctly. These features enable using consistent syntax with both class types and struct types. Furthermore, when you make mistakes, the compiler surfaces the errors at the location where you can best correct the error. The features added for this goal are:

- Required members
- Auto-default struct
- Extended nameof scope
- Generic attributes

> Required members lets you annotate a member declaration to inform the compiler that it must be initialized either in a constructor or an object initializer.

Required members lets you annotate a member declaration to inform the compiler that it must be initialized either in a constructor or an object initializer. Consider this class:

```
public class Person
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
}
```

Callers are expected to set the values for the FirstName and LastName properties using object initializers. However, the compiler doesn't enforce that expectation. In C# 11, you add the required modifier to both property declaration to mandate that callers must initialize these properties:

```
public class Person
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
}
```

All callers must include object initializers for both properties. Otherwise, the compiler emits an error. The caller must make their code match the expectations set by the type author. If the type also has a constructor that sets the required properties, the type author adds the SetsRequiredMembers attribute to the constructor declaration. Then, the compiler forces that constructor to set an initial value for all required members. Callers using that constructor aren't required to add object initializers for those members. For example, the Person type might have the following declaration:

```
public class Person
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
```

Listing 1: List pattern examples

```
int[] one = { 1 };
int[] odd = { 1, 3, 5 };
int[] even = { 2, 4, 6 };
int[] fib = { 1, 1, 2, 3, 5 };

// You can match the entire sequence by specifying
// all the elements and using values:
Console.WriteLine(odd is [1, 3, 5]); // true
Console.WriteLine(even is [1, 3, 5]); // false (values)
Console.WriteLine(one is [1, 3, 5]); // false (length)

// You can match some elements in a sequence of
// a known length using the discard pattern (_) as a placeholder:

Console.WriteLine(odd is [1, _, _]); // true
Console.WriteLine(odd is [_, 3, _]); // true
Console.WriteLine(even is [_, _, 5]); // false (last value)

// You can supply any number of values or placeholders
// anywhere in the sequence. If you aren't concerned with the length,
// you can use the range pattern to match zero or more elements:
Console.WriteLine(odd is [1, .., 3, _]); // true
Console.WriteLine(fib is [1, .., 3, _]); // true

Console.WriteLine(odd is [1, _, 5, ..]); // true
Console.WriteLine(fib is [1, _, 5, ..]); // false


// The previous examples used the constant pattern
// to determine if an element is a given number.
// Any of those patterns could be replaced by a
// different pattern, such as a relational pattern:
Console.WriteLine(odd is [_, > 1, ..]); // true
Console.WriteLine(even is [_, > 1, ..]); // true
Console.WriteLine(fib is [_, > 1, ..]); // false
```

```
    public Person() { }

    [SetsRequiredMembers]
    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
}
```

Callers using the first constructor must set the required members. Callers using the second constructor rely on the constructor to set the required members.

Auto-default structs clarify the rules for definite assignment in struct constructors. The .NET runtime always initializes the storage for a struct to all 0s. Therefore, struct constructors don't need to set any member to the 0 value explicitly. The additional assignment is unnecessary. Constructors that don't explicitly set all member values now compile, and the compiler sets all members to definitely assigned. In general, when you create a struct by calling a constructor, that constructor initializes all values correctly. When you set a struct to the default value, all struct members are set to 0.

With **extended nameof scope**, type parameter names and parameter names are now in scope when used in a nameof expression in an attribute declaration on that method. This feature means that you can use the nameof operator to specify the name of a method parameter in an attribute on the method or parameter declaration. This feature is most often useful to add attributes for nullable analysis.

C# 11 adds **generic attributes**. A generic class can now declare System.Attribute as a based class. This provides a more convenient syntax for attributes that require a System.Type parameter.

These features make it easier for you to initialize objects correctly. They help the developers using your types to use them correctly.

Generic Math Support

The runtime support for generic math is covered in depth elsewhere. There are several new C# features that support this initiative:

- Static abstract and static virtual members in interfaces
- Checked user-defined operators
- Relaxed shift operators
- Unsigned right-shift operator
- Numeric IntPtr and UIntPtr

The largest set of changes are those necessary for **static abstract and static virtual members in interfaces.** The concept is easy to understand: You can declare operators or other static function as "virtual" or "abstract" in an interface. Any class that implements that interface must provide an implementation of those operators and static methods. Because these methods are static, the compiler must resolve the target method; there's no runtime dispatch as there is with instance virtual methods. That means that the compiler must be able to determine the correct type for each static method call.

In practice, that means that these interfaces are generally generic interfaces. Furthermore, one of the type parameters must be constrained to be a type that implements the interface. For example, the INumber interface covered in the generic math article declares this constraint, among many others:

```
public interface INumber<TSelf>
    where TSelf : INumber<TSelf>
```

Checked user defined operators enables developers to write different implementations of many overloaded operators for a checked and an unchecked context. In previous versions of C#, all overloaded operators used the same implementation in both a checked and unchecked context. This feature enables type authors to specify different behavior in each context.

The shift operators use to require that the right operand was an int. With **relaxed shift** operators, that restriction is now removed. Generic math algorithms can use the shift operators, and the right operand can be the type implementing the INumber interface.

The **unsigned right shift operator**, >>>, shifts an integral type to the right, always inserting 0 in the left-most bits. The arithmetic shift operator, >>, inserts 0 for positive numbers and 1 for negative numbers.

Finally, the keywords nint, and nuint are synonyms for the types System.IntPtr and System.UIntPtr, respectively. The native sized integer keywords were added in C# 10, but were not considered the same as the corresponding types. Now they are.

If you write types that represent numbers, you'll use these features and the corresponding generic math interfaces often. Otherwise, they may not directly affect your code, but you'll benefit from the consolidation of many numeric methods in the runtime library.

Runtime Performance

Finally, C# 11 adds features that can improve runtime performance. Most of these were requested by the .NET Runtime team. They are used in the runtime, so you'll get the performance benefits even if you don't use the feature in your code:

- Ref fields and scoped ref
- File local types
- Cached method group conversion to delegate

> Most of these were requested by the .NET Runtime team. They're used in the runtime, so you'll get the performance benefits even if you don't use the feature in your code.

Ref fields and scoped ref provide more syntax to enable passing parameters by reference. These features can reduce copying values or allocating new objects. **Ref fields** allows the ref modifier on a member of a ref struct. This feature minimizes copying when a ref struct needs to reference some storage in another object. The compiler uses static analysis to ensure that the ref struct doesn't have a lifetime that could extend beyond the source of the ref field. Ref parameters can include the scoped modifier to inform the compiler that the reference can't have a lifetime beyond the current method. That restricts how it can be passed, stored, or what storage could be assigned to the parameter. These language enhancements enable developers to write more performant code without using unsafe features. The compiler can enforce lifetime rules of reference variables.

File local types are classes where the definition includes the file modifier. These types can only be accessed within the same source file. They are primarily useful for code generators. A code generator can generate a class scoped to its output source file safely knowing that the type won't conflict with another type in the destination program.

Cached method group conversion to delegate is a compiler performance improvement that you'll benefit from without changing any of your code. This feature benefits from the updated work by the ECMA C# committee. Earlier standards forced the compiler to allocate a new delegate object when code converted a method group to a delegate. The committee updated the standard for version 6 to allow the compiler to cache the delegate object. C# 11, the next compiler released following that change, takes advantage of this new option. Your code makes fewer allocations just by compiling with C# 11.

Wrapping up

C# 11 adds quite a few features, some you're more likely to use than others. Some are for specific scenarios, like generic math; others are more general, like raw string literals. C# keeps evolving to support the applications that you're building today, while still being the language you know and love. You can find links to more details on each of these features at https://docs.microsoft.com/dotnet/csharp/whats-new/csharp-11.

Bill Wagner
CODE

Highlighted Performance Wins with .NET 7

Ever since .NET Core hit the scene more than seven years ago, performance has been an integral part of the culture of .NET. Developers are encouraged to find and improve all aspects of performance across the codebase, with changes ranging from removing a small allocation here, to overhauling an entire algorithm there, to brand new APIs intended to enable all developers

Stephen Toub
stoub@microsoft.com

Stephen Toub is a Partner Software Engineer at Microsoft. He spends most of his time focused on the libraries that comprise .NET and on performance across the .NET stack.

to further improve their own application and service performance. This, in turn, leads to immense quantities of improvements in any given release, and .NET 7 is no exception. The post at https://devblogs.microsoft.com/dotnet/performance_improvements_in_net_7 provides an in-depth exploration of hundreds of these.

Given the quantity and quality of these improvements, there's no handful of changes I can point to that are "the best," with changes each having their own unique impact in distinct areas of the platform. Instead, in this article, I've picked just three areas of improvement to highlight as ones that can have an instrumental impact on the performance of your code. This is a small taste, and I encourage you to read the (long) cited post to get a better understanding for the breadth and depth of performance improvements in this release.

On-Stack Replacement

There are many positives to just-in-time (JIT) compilation. A single binary can be run on any supported platform, with the code in the binary translated to the target computer's architecture on-demand, and in doing so, the JIT can specialize the generated code for the particulars of the target computer, such as choosing the best instructions it supports to implement a particular operation. The flip side of this is that on-demand compilation takes time during the execution of the program, and that time often shows up as delays in the startup of an application (or, for example, in the time for a service to complete a first request).

Tiered compilation was introduced in .NET Core 3.0 as a compromise between startup and steady-state throughput. With it, the JIT is able to compile methods multiple times. It first does so with minimal optimization, in order to minimize how long it takes to compile the method (because the cost of finding and applying optimizations is a significant percentage of the cost of compiling a method). It also equips that minimally optimized code with tracking for how many times the method is invoked. Once the method has been invoked enough times, the JIT recompiles the method, this time with all possible optimization, and redirects all future invocations of the method to that heavily optimized version.

That way, startup is faster while steady-state throughput remains efficient; in fact, steady-state throughput can even be faster because of additional information the JIT can learn about the method from its first compilation and then use when doing the subsequent compilation. However, methods with by-default loops previously were opted-out of tiered compilation because such methods can consume significant amounts of the app's execution time even without being invoked multiple times.

In .NET 7, even methods with loops benefit from tiered compilation. This is achieved via on-stack replacement (OSR). OSR results in the JIT not only equipping that initial compilation for number of invocations, but also equipping loops for the number of iterations processed. When the number of iterations exceeds a predetermined limit, just as with invocation count, the JIT compiles a new optimized version of the method. It then jumps to the new method, transferring over all relevant state so that execution can continue running seamlessly without the method needing to be invoked again.

You can see this in action in a few ways. Try compiling a simple console app:

```
using System;

for (int i = 0; ; i++)
{
    if (i == 0)
    {
        Console.WriteLine("Running...");
    }
}
```

Run it, but first set the **DOTNET_JitDisasmSummary** environment variable to 1. You should end up seeing output that includes lines like these:

```
   4: JIT compiled Program:<Main>$(ref)
      [Tier0, IL size=21, code size=94]
…
   6: JIT compiled Program:<Main>$(ref)
      [Tier1-OSR @0x2, IL size=21, code size=43]
```

The C# compiler generated a **<Main>$** method for your program containing top-level statements, and you see two entries for it in the JIT's recording of every method it compiled. The first is the initial compilation with minimal optimization, and the second is the OSR variant that's fully optimized. If you didn't have OSR such that tiered compilation didn't apply to this method, you would have instead seen a single entry for this method.

Or, try running this silly little benchmark:

```
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;
using System;

[DisassemblyDiagnoser]
public partial class Program
{
    static void Main(string[] args) =>
        BenchmarkSwitcher.
```

```
        FromAssembly(typeof(Program).Assembly).
        Run(args);

    private static readonly int s_year =
        DateTime.UtcNow.Year;

    [Benchmark]
    public int Compute()
    {
        int result = 0;
        for (int i = 0; i < 1_000_000; i++)
        {
            result += i;
            if (s_year == 2021)
            {
                result += i;
            }
        }
        return result;
    }
}
```

When I run it, I get numbers like .NET 6 taking 858.9us and .NET 7 taking 237.3us. Why is .NET 7 so much faster? Because of OSR.

In the .NET 6 version, tiered compilation isn't used and the loop is compiled such that the comparison in the body of the loop requires reading the value of **s_year**. In the .NET 7 version, tiered compilation is used, and when OSR kicks in and causes an optimized version of the code to be generated, by that point, the JIT knows that **s_year** was already initialized. As it's a **static readonly**, the JIT knows its value will never change, and the JIT can treat it like a constant. It'll see that the year isn't 2021 and so eliminate that **if** block as dead code. You're thus saving a static field read and a branch on every iteration of the loop when compared to the .NET 6 execution.

Regex

.NET 7 significantly improves the performance of regular expressions processing, so much so that in addition to the previously cited .NET 7 performance post, there's an entire post dedicated to **Regex** improvements (https://devblogs.microsoft.com/dotnet/regular-expression-improvements-in-dotnet-7). The performance improvements here broadly fit into four categories: ones that result in existing regexes being faster, new APIs for more efficiently working with regexes, a new regex source generator (which has not only startup and steady-state performance benefits but also pedagogical benefits), and the new **RegexOptions.NonBacktracking** option. Here, I'll take a look at the source generator, touching on a few of the other improvements, and you can see the blog posts for more details.

Let's say I wanted a regular expression for a specific formatting of phone numbers in the United States:

```
@"[0-9]{3}-[0-9]{3}-[0-9]{4}"
```

This looks for three digits, a dash, three more digits, another dash, and another four digits. And let's say my task was to count the number of phone numbers in a given piece of text. With .NET 6 and earlier, I might implement that as follows:

```
partial class Helpers
{
    private static readonly Regex s_pn =
        new Regex(@"[0-9]{3}-[0-9]{3}-[0-9]{4}",
            RegexOptions.Compiled);

    public static int CountPhoneNumbers(
        string text)
    {
        int count = 0;
        Match m = s_pn.Match(text);
        while (m.Success)
        {
            count++;
            m = m.NextMatch();
        }
        return count;
    }
}
```

This uses the **RegexOptions.Compiled** flag to ask the runtime to use reflection emit to generate a customized implementation of this regular expression at execution time. Doing so will make steady-state throughput much faster, at the expense of having to do all of the work to parse, analyze, optimize, and code gen this expression at execution time. It also won't be able to do that code generation in an environment that lacks a JIT compiler, like with Native AOT in .NET 7. What if all of that work could be done at compile-time instead? In .NET 7, it can. I can write the above in .NET 7 instead as the following:

```
partial class Helpers
{
    [GeneratedRegex(
        @"[0-9]{3}-[0-9]{3}-[0-9]{4}")]
    private static partial Regex PhoneNumber();

    public static int CountPhoneNumbers(
        ReadOnlySpan<char> text) =>
            PhoneNumber().Count(text);
}
```

A few things to notice here. First, my **CountPhoneNumbers** helper now accepts a **ReadOnlySpan<char>** instead of a **string** (which is implicitly convertible to **ReadOnlySpan<char>**). That's feasible because **Regex** now has multiple methods that accept **ReadOnlySpan<char>** as input and work efficiently over such spans, enabling my **CountPhoneNumbers** function to be used with a wider set of inputs, such as a **char[]**, a stackalloc'd **Span<char>**, a **ReadOnlySpan<char>** created around some native memory from interop, and so on.

Second, the body of my **CountPhoneNumbers** method has been condensed to a single line that just calls the **Count** method; this is a new method on **Regex** in .NET 7 that efficiently counts the number of occurrences in the input, in an amortized allocation-free manner.

Third and most importantly for this example, you'll notice that I'm no longer using the **Regex** constructor. Instead, I have a partial method that returns **Regex** and that's attributed with **[GeneratedRegex(...)]**. This new attribute triggers a source generator included in the .NET SDK to emit a C# implementation of the specified regex, as shown in the screenshot of Visual Studio in **Figure 1**. The generated code is logically equivalent to the IL that would be emitted by **RegexOptions.Compiled** at execution time but is instead C# emitted at compile time. This means that you get all the throughput benefits of the compiled approach, even in an

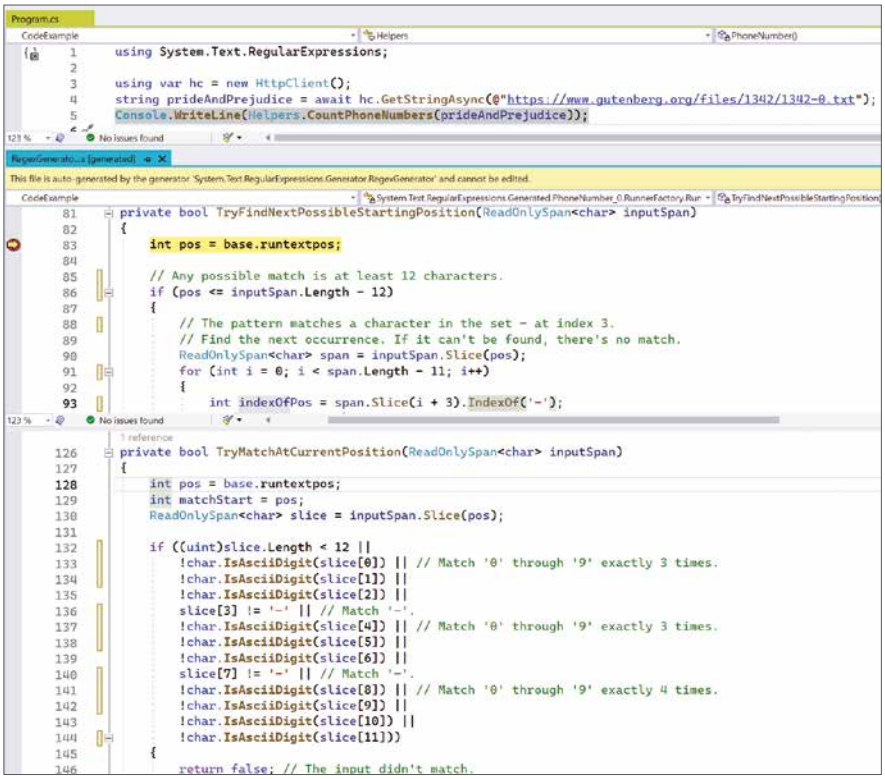

Figure 1: Stepping through C# code generated for a Regex

environment that doesn't support JIT'ing, and you get those benefits without having to pay for it in startup time. This code is also viewable and debuggable, which helps improve not only the correctness of an application but also your knowledge of how regexes are evaluated:

You can see in this example that the source generator has output very clean, very readable source code for the regular expression, tailored to exactly what the regex is, and very close to what you yourself might write if you were trying to write code to match this pattern. The code is even commented, and it uses new methods also introduced in .NET 7 for identifying categories of characters like ASCII digits.

It's also learned some new tricks in how to search efficiently for possible locations that might match the pattern. In .NET 6, it would have walked character by character looking for a digit that could start the pattern. Now in .NET 7, you can see it's searching for the dash, and if found, it'll be able to back up a few characters to try to match the pattern starting at that position. In a text like "Pride and Prejudice" that I'm searching here, which, as you might guess, has very few U.S. phone numbers in it, this leads to significant performance improvements.

LINQ

Language Integrated Query (LINQ) continues to be one of my favorite features in all of .NET. As a combination of over 200 methods for performing various manipulations of arbitrary collections of data, it's a succinct and flexible way to manipulate information. For example, during our work to optimize **Regex**, we'd frequently want to ask questions like "if I make this change to our loop auto-atomicity logic, how many loops could we now make atomic automatically," and with a database of around 20,000 real-world regular expressions in hand, we could write LINQ queries against the internal object model used to represent a parsed regular expression "node tree" in order to quickly get the answer, like this:

```
int count =
    (from pattern in patterns
     let tree = Parse(pattern)
     from node in tree.EnumerateAllNodes()
     where node.Kind is
         RegexNodeKind.Oneloopatomic or
         RegexNodeKind.Notoneloopatomic or
         RegexNodeKind.Setloopatomic
     select node)
    .Count();
```

As LINQ is used in so many applications by so many developers, we strive to make it as efficient as possible (even though in our core libraries we still avoid using it on hot paths due to overheads involved in things like enumerator allocation and delegate invocation). Previous releases of .NET saw some significant improvements, for example due to reducing the algorithmic complexity of various operations by passing additional information from one operator to another (such as enabling an **OrderBy(...).ElementAt** to perform a "quick select" rather than "quick sort" operation). In .NET 7, one of the larger improvements in LINQ relates to a much larger set of optimizations throughout .NET 7, that of vectorization.

Vectorization is the process of changing an implementation to use vector instructions, which are SIMD (single instruction multiple data) instructions capable of processing multiple pieces of data at the same time. Imagine that you wanted to determine whether an array of 1,000,000 bytes contained any zeros. You could loop over every byte in the array looking for 0, in which case you'd be performing 1,000,000 reads and comparisons. But what if you instead treated the array of 1,000,000 bytes as a span of 250,000 **Int32** values? You'd then only need to perform 250,000 read and comparison operations, and since the cost of reading and comparing an **Int32** is generally no more expensive than the cost of reading and comparing a byte, you'd have just quadrupled the throughput of your loop. What if you instead handled it as a span of 125,000 **Int64** values? What if you could process even more of the data at a time? That's vectorization.

Modern hardware provides the ability to process 128 bits, 256 bits, even 512 bits at a time (referred to as the width of the vector), with a plethora of instructions for performing various operations over a vector of data at a time. As you might guess, using these instructions can result in absolutely massive performance speedups. Many of these instructions were surfaced for direct consumption as "hardware intrinsics" in .NET Core 3.1 and .NET 5, but using those directly requires advanced know-how and is only recommended when absolutely necessary.

Higher level support has previously been exposed via the **Vector<T>** type, which enables you to write code in terms of **Vector<T>**, and the JIT then compiles that usage down to the best available instructions for the given system. **Vector<T>** is referred to as being "variable width," because, depending on the system, the code actually ends up running on, it might map to 128-bit or 256-bit instructions, and because of that variable nature, the operations you can perform with it are somewhat limited. .NET 7 sees the introduction of the new fixed-width **Vector128<T>** and **Vector256<T>** types, which are much more flexible. Many of the public APIs in .NET itself are now vectorized using one or more of these approaches.

Some of LINQ was previously vectorized. In .NET 6, the **Enumerable.SequenceEqual** method was augmented to special-case **T[]**, in which case, the implementation would use a vectorized implementation to compare the two arrays. In .NET 7, some of the overloads of Enumerable.Min, Enumerable.Max, Enumerable.Average, and Enumerable.Sum have all been improved.

First, these methods now all specialize for the very commonly used types **T[]** and **List<T>**, in order to optimize the processing of their contents. Both of these types make it easy to get a **ReadOnlySpan<T>** for their contents, which in turn means the contents of either can be processed with one shared routine that has fast access to each element rather than needing to go through an enumerator.

And then some of the implementations are able to take it further and vectorize that processing. Consider **public static double Enumerable.Average(this IEnumerable<int>)**, for example. Its behavior is to sum all of the **Int32** values in the source, accumulating the sum into an **Int64,** and then dividing that **Int64** by the number of summed elements (which needs to be at least 1). This can lead to huge speedups, for example:

```
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;
using System.Buffers;
using System.Linq;

public partial class Program
{
    static void Main(string[] args) =>
        BenchmarkSwitcher.
        FromAssembly(typeof(Program).Assembly).
        Run(args);

    private int[] _values =
        Enumerable.Range(0, 1_000_000).
        ToArray();

    [Benchmark]
    public double Average() =>
        _values.Average();
}
```

On my computer, this shows the benchmark on .NET 6 taking an average of 3661.6us and on .NET 7 taking an average of 141.2us—almost 26 times faster. How would you vectorize an operation like this if you wanted to do it by hand? Let's start with the sequential implementation, and for simplicity, let's assume the input has already been validated to be non-null and non-empty.

```
static double Average(ReadOnlySpan<int> values)
{
    long sum = 0;

    for (int i = 0; i < values.Length; i++)
    {
        sum += values[i];
    }

    return (double)sum / values.Length;
}
```

The basic idea is that you want to process as much as possible vectorized, and then use the same one-at-a-time loop to process any remaining items. Start by validating that you're running on hardware that can, in fact, accelerate the vectorized implementation and that you have enough data to vectorize at least something:

```
if (Vector.IsHardwareAccelerated &&
    values.Length >= Vector<int>.Count)
{
    ...
}
```

Once you know that, you can write your vectorized loop. The approach will be to maintain a **Vector<long>** of partial sums; for each **Vector<int>** you read from the input, you'll "widen" it into two **Vector<long>**s, meaning every **int** will be cast to a **long**, and because that doubles the size, you'll need two **Vector<long>**s to store the same data as the **Vector<int>.** Once you have those two **Vector<long>**s, you can just add them to your partial sums. And at the end, you can sum all of the partial sums together. That gives you this as your entire implementation:

```
static double Average(ReadOnlySpan<int> values)
{
    long sum = 0;
    int i = 0;

    if (Vector.IsHardwareAccelerated &&
        values.Length >= Vector<int>.Count)
    {
        Vector<long> sums = default;
        do
        {
            Vector.Widen(
              new Vector<int>(values.Slice(i)),
              out Vector<long> low,
              out Vector<long> high);
            sums += low;
            sums += high;
            i += Vector<int>.Count;
        }
        while (i <= values.Length -
                    Vector<int>.Count);
        sum += Vector.Sum(sums);
    }

    for (; i < values.Length; i++)
    {
        sum += values[i];
    }

    return (double)sum / values.Length;
}
```

If you look at the implementation of this overload in .NET 7 (it's open source, and you're encouraged to read and contribute), you'll see this is almost exactly what the official code currently does.

Call to Action

.NET 7 is full of these kinds of performance improvements, across the entirety of the release. Please download .NET 7, upgrade your apps and services, and try it out for yourself. We're excited to hear how these improvements contribute to the performance of your applications.

Stephen Toub

Use .NET MAUI for Native, No-Compromise Apps

Where your users are, that's where you want to be, where you need to be. On their phones, on their tablets, on their computers, and generally everywhere they interact with your app. This is reach—being present where your users are and where they're most effectively interacting with your app in order to help them to achieve their goals. The foundation of this omnipresence is being on both mobile and Web. The question is: How do you get there faster than your competition, with better quality, and at a lower cost?

David Ortinau
david.ortinau@microsoft.com
dev.to/davidortinau
twitter.com/davidortinau

David is a Principal Product Manager for .NET Client Apps at Microsoft, focused on .NET MAUI. A .NET developer since 2002, and versed in a range of programming languages, he has developed web, environmental, and mobile experiences for a wide variety of industries. After several successes with tech startups and running his own software company, David joined Microsoft to follow his passion: crafting tools that help developers create better app experiences.

Microsoft offers a variety of ways to create client applications for everything from games and virtual reality to forms over data. The catalog includes low-code solutions with Power Apps, Teams apps, C++ SDKs for desktop and mobile, and .NET, of course. It's great to have choice, and Microsoft knows first-hand from talking to many companies that each has a place. So, where does .NET fit in this mix and when should you use it?

We have shaped .NET Multi-platform App UI (MAUI) first and foremost for you to deliver client applications that feature rich user interactions, high performance, and native platform experiences. You can then leverage all a device and platform has to offer because .NET MAUI builds upon the native UI frameworks of Android, iOS, macOS, and Windows.

So that we start off on the same page, let me begin by defining how I'll be using some common and thus commonly fuzzy terms.

- **Native** means using the technology specifically designed and optimized for the best experience on a particular platform.
- A **client application** is any application a user interacts with that runs on devices such as mobile, tablet, desktop, etc., typically via a graphical interface.
- **Platform capabilities** refer to features such as camera, RFID, GPS, secure storage, accelerometer, file system, notifications, system tray, menus, Bluetooth, and so on.
- **Integrations** are hardware and software that can be leveraged by your application such as scanners, color sensors, printers, medical devices, and more.

One .NET

"If you are a .NET developer, then you are already a .NET MAUI developer," I heard a developer say recently on YouTube. This is exactly the confidence we want everyone to get from using .NET MAUI when applying their .NET experience to building mobile and desktop applications. We have designed this familiarity into .NET MAUI.

Your .NET code works the same in .NET MAUI as it does in Blazor, ASP.NET, or any other .NET app. It all uses the same base class library and runtime. The only difference is how the runtime on mobile is optimized for resource-constrained devices.

The startup and configuration of a .NET MAUI app should look very familiar to you. We employ the same builder pattern to create your app instance, configure dependencies, tap into lifecycle events, and more.

```
public static class MauiProgram {
    public static MauiApp CreateMauiApp(){
        var builder = MauiApp.CreateBuilder();
            builder
                .UseMauiApp<App>()
                .ConfigureFonts(fonts => {
                    fonts.AddFont("Regular.ttf",
                                  "OpenSansRegular");
                    fonts.AddFont("Semibold.ttf",
                                  "OpenSansSemibold");
            });
        return builder.Build();
    }
}
```

The project structure itself, as you see it in Visual Studio 2022 (see **Figure 1**), looks a lot like your other .NET solutions where your app is a single project. Historically, you'd have had a separate project for each platform you targeted and then a library project for any shared code. .NET MAUI gives you direct access to each platform from just one project by using .NET multi-targeting and unifies a bunch of common resource tasks. I'll explain more on that in the next section.

One Project, Many Platforms

You choose a development platform to build your app, not to wrangle with underlying platform idiosyncrasies, reconciling the differences between multiple platforms. In .NET

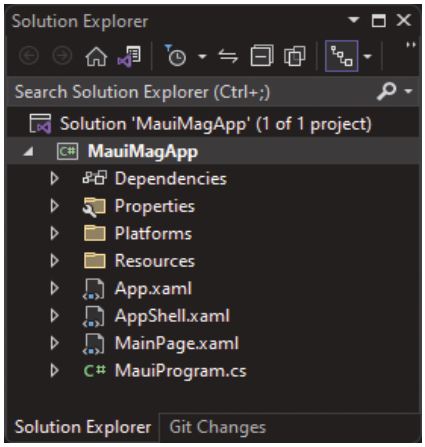

Figure 1: The Solution Explorer in Visual Studio

MAUI, we decided it didn't need to be so complicated to build cross-platform apps that remain native, so we've abstracted away the meaningless differences. Each platform is still close within reach. You should have to do as little as needed to get the most from each platform, including resources, styles, lifecycle events, and writing platform code.

Any Resource in One Place

Managing images and other visual assets can be a labor-intensive process and one that often involves both a developer and a designer. Each platform has different requirements for file formats, naming conventions, screen densities, and more. In .NET MAUI, you can place your source image in any of the supported formats in the **Resources/Images** folder, and .NET MAUI automatically generates the various artifacts needed.

Images

.NET MAUI supports source PNG, JPG, GIF, and SVG, which is a great vector file format that lends itself well to upscaling as well as downscaling. By default, all images are treated the same, starting from the original size. To get more control, open the **csproj** file and add some additional information for that image. For example, if you want to constrain an image to a different base size than the original, set a **BaseSize.**

```
<MauiImage
    Source="Resources/Images/global_map.svg"
    BaseSize="300,600"/>
```

The original size may be much larger, which isn't ideal for mobile. Or perhaps the image is exactly the size you want, so you can tell .NET MAUI to not resize anything by adding **Resize="false".** In this way, you can easily constrain the artifacts and keep your memory footprint low on mobile. Then, to use this image anywhere in your application, you need only reference the filename.

```
<Image
    Source="global_map.png"/>
```

Notice that here, the file extension changed from **svg** to **png**. This is because it references the resource optimized for runtime performance, which is a **png**, and not the source file, which, in this case, is an **svg**. To directly use **svg** at runtime, look to SkiaSharp or another library that supports that format.

You also configure your app icons and splash screen the same way in .NET MAUI by denoting which resources should be used to generate all the different sizes (see **Figure 2**) for the platforms you're targeting.

```
<MauiIcon
  Include="Resources\AppIcon\appicon.svg"
  ForegroundFile="Resources\AppIcon\appiconfg.svg"
  Color="#512BD4" />

<MauiSplashScreen
  Include="Resources\Splash\splash.svg"
  Color="#512BD4"
  BaseSize="128,128" />
```

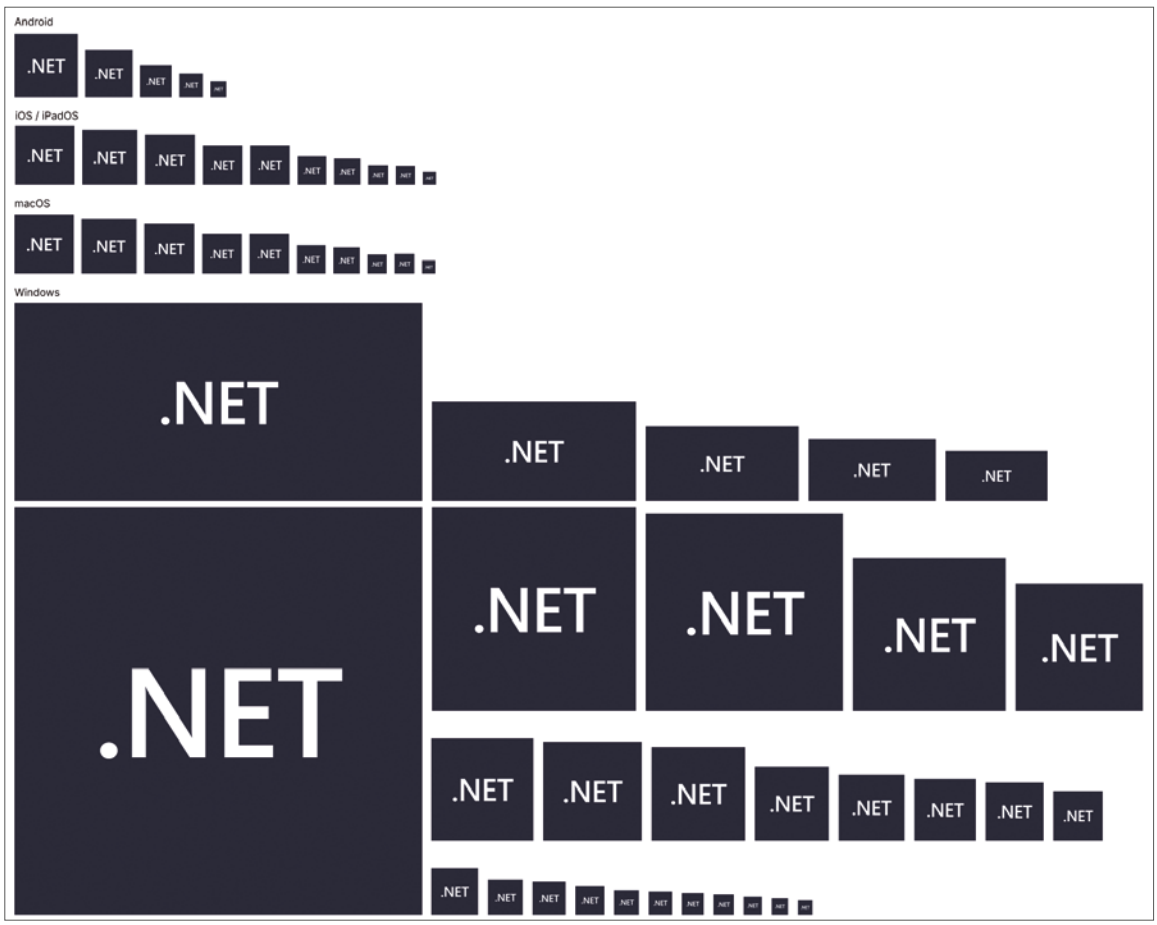

Figure 2: All the sizes of images needed for app icons

Fonts for Typography and Icons

Fonts are another notoriously bothersome thing to get right across multiple platforms and .NET MAUI makes this simple. Place any **TrueType** or **OpenType** in your **Resources/Fonts** folder, and then give the file a name in the **MauiProgram** that you'll use in your styles.

```
.ConfigureFonts(fonts => {
    fonts.AddFont("OpenSans.ttf", "OpenSans");
    fonts.AddFont("fabmdl2.ttf", "Fabric");
})
```

Apply the font family by that name to your styles or directly on any control, and if your font includes icon glyphs, you can use them for image sources as well. You can see this in **Figure 3**, where the down chevron glyph is beside the button label.

```
<Label Text="{x:Static a:FabIconFont.ChevronDown}"
       FontFamily="Fabric"/>
```

Styles

From a new project, .NET MAUI provides a complete stylesheet with a default .NET brand-inspired color palette, sensible defaults for all controls including light and dark themes, and visual states for interactive controls. You can use this as a starting point or as a reference guide to add your own global styles to your project (see **Listing 1**). The styles are implicit, which means any new control, such as a **Button** that you add to the screen, inherits those characteristics.

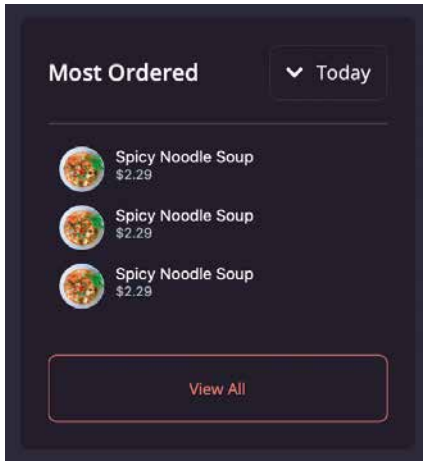

Figure 3: The font icon is used for the down chevron.

```
<Button Text="Save" />
```

In your application, you may wish to have different button styles for different uses (see **Figure 4**), such as primary action, secondary action, or by solid and outlined styles. For this, you can define a button style just like above and give it an **x:Key** to reference explicitly.

```
<Button Text="Save"
    Style="{StaticResource OutlineButton}" />

// Resources/Styles/Styles.xaml
<Style TargetType="Button" x:Key="OutlineButton">
    <Setter Property="CornerRadius" Value="8"/>
    <Setter Property="TextColor" Value="Black"/>
    <Setter Property="BorderWidth" Value="1"/>
    <Setter Property="BorderColor" Value="Black"/>
    <Setter Property="Background"
        Value="Transparent"/>
</Style>
```

App and View Lifecycle Events

Lifecycle events differ from platform to platform, not always firing in the same order. .NET MAUI provides a consistent cross-platform set of lifecycle events for your app itself and each view element, and platform-specific lifecycle events for when you need more control. App lifecycle events (as you see in **Figure 5**) include:

- **Created**: After the native window has been created
- **Activated**: When the created window becomes the focused window
- **Deactivated**: When the window is no longer the focused window
- **Stopped**: When the window is no longer visible
- **Resumed**: When the app resumes from being stopped
- **Destroying**: When the native window is being destroyed

Now in your app, you can access an event like **Created** from the window to perform cross-platform or even platform-specific work.

Listing 1: Default implicit Button style

```
code Listing (more than 15 lines)
        1         2         3         4         5         6
1234567890123456789012345678901234567890123456789012345678901234567

<Style TargetType="Button">
    <Setter Property="TextColor" Value="{AppThemeBinding
        Light={StaticResource White},
        Dark={StaticResource Primary}}" />
    <Setter Property="BackgroundColor" Value="{AppThemeBinding
        Light={StaticResource Primary},
        Dark={StaticResource White}}" />
    <Setter Property="FontFamily" Value="OpenSansRegular"/>
    <Setter Property="FontSize" Value="14"/>
    <Setter Property="CornerRadius" Value="8"/>
    <Setter Property="Padding" Value="14,10"/>
    <Setter Property="VisualStateManager.VisualStateGroups">
        <VisualStateGroupList>
            <VisualStateGroup x:Name="CommonStates">
                <VisualState x:Name="Normal" />
                <VisualState x:Name="Disabled">
                    <VisualState.Setters>
                        <Setter Property="TextColor"
                            Value="{AppThemeBinding
                            Light={StaticResource Gray950},
                            Dark={StaticResource Gray200}}" />
                        <Setter Property="BackgroundColor"
                            Value="{AppThemeBinding
                            Light={StaticResource Gray200},
                            Dark={StaticResource Gray600}}" />
                    </VisualState.Setters>
                </VisualState>
            </VisualStateGroup>
        </VisualStateGroupList>
    </Setter>
</Style>
```

```
protected override Window
    CreateWindow(IActivationState activationState)
    {
        Window window =
              base.CreateWindow(activationState);
        window.Created += (s, e) =>
        {
            var mauiWindow = (Window)s;
#if WINDOWS
            var nativeWindow =
                (MauiWinUIWindow)mauiWindow
                    .Handler.PlatformView;
            nativeWindow.GetAppWindow()
            .MoveAndResize(
            new RectInt32(x, y, width, height));
#endif
        };
        return window;
    }
```

From the cross-platform window control, you can access the platform view through the **handler**, the part of .NET MAUI responsible for mapping cross-platform to native platform APIs. Using multi-targeting compiler directives like **#if WINDOWS**, you can then use those platform-specific APIs to do things like centering the WinUI window on the user's screen.

Awesome for Mobile

.NET MAUI is the fastest .NET mobile we've ever shipped for Android and iOS. .NET 6 is 68% faster than Xamarin.Android, and 44% faster than Xamarin.Forms. Our full-featured .NET MAUI podcast app starts in under 240ms on modern iPhone and iPad devices. With each release of .NET, your client applications benefit from these improvements and many others made across the entire .NET stack.

Performance is only the beginning of what makes .NET MAUI great on mobile devices. Our product team is heavily focused on building a product that serves you, our customers, from small to enterprise-size companies. You have helped us and continue to help us become experts in what it takes to build apps that are successful for your businesses: a complete set of UI controls, a vibrant ecosystem of libraries, developer tooling, enterprise patterns and practices, and abundant learning resources.

UI Controls

Use C# or XAML to build rich UI with more than 40 cross-platform controls and layouts, plus all the platform-specific controls provided by each native framework. Compose and style them together to get infinitely more!

By default, each control is styled consistently while maintaining the characteristics that make it native to the platform. A perfect example is the **Entry** control for text input, as you can see in **Figure 6**. As you can see from the static image, this control looks quite different on all four platforms. The behavior also differs. Are these differences meaningful to your users? Often, they are, and this is the experience you want to ship.

When it's not the experience you were going for and you wish to unify the look and feel across all platforms, you can tap into the power of .NET MAUI handlers. Handlers are the implementations that drive each control, loosely coupled to the native platform. You can think of the architecture of controls this way: The cross-platform **Control** has common characteristics (properties, events, etc.) that a **Handler** maps to each platform control. Any of those mapped things can be overridden, or code added before or after by you.

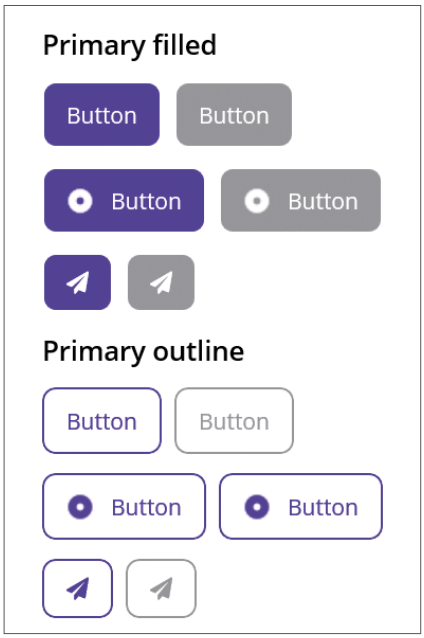

Figure 4: Buttons with various styles

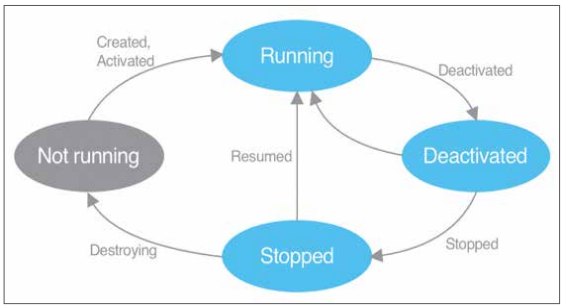

Figure 5: App lifecycle diagram

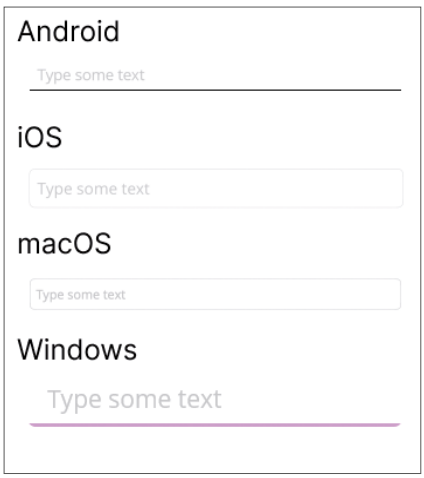

Figure 6: Platform default styling for Entry fields

For example, the first step to unifying the look of the **Entry** across platforms is to remove the underline and outlines.

```
EntryHandler.Mapper
  .AppendToMapping("MyCustomization",
    (handler, view) => {
#if ANDROID
      // Android specific code here
#elif IOS || MACCATALYST
      // iOS specific code here
#elif WINDOWS
      // Windows specific code here
#endif
});
```

By appending my own custom method to the handler map, my code will be run when any **Entry** is created. I just need to call this code somewhere in the startup of my application or before I put my first **Entry** on screen. Three methods are provided to customize handlers:

- **PrependToMapping:** Modifies the mapper for a handler before the .NET MAUI control mappings have been applied
- **ModifyMapping:** Modifies an existing mapping
- **AppendToMapping:** Modifies the mapper for a handler after the .NET MAUI control mappings have been applied

Listing 2: HomeViewModel without helpers

```
code Listing (more than 15 lines)
         1         2         3         4         5         6
1234567890123456789012345678901234567890123456789012345678901234567

public class HomeViewModel : INotifyPropertyChanged
{
        ObservableCollection<Item> _products;
        public ObservableCollection<Item> Products
        {
            get
            {
                return _products;
            }
            set
            {
                if (_products != value)
                {
                    _products = value;
                    OnPropertyChanged(nameOf(Products));
                }
            }
        }

        public Command AddProductCommand =>
                        new Command(AddProduct);

        async Task AddProduct()
    {
        await MessagingCenter.Send<HomeViewModel, string>
                                (this, "action", "add");
    }
.       #region INotifyPropertyChanged
        public event PropertyChangedEventHandler PropertyChanged;

        void OnPropertyChanged(
                 [CallerMemberName] string propertyName = null)
        {
            PropertyChanged?.Invoke(this,
                new PropertyChangedEventArgs(propertyName));
        }
        #endregion
}
```

With just a few lines of code, you can make deep platform customizations to your cross-platform app.

Amazing for Desktop

.NET MAUI is the first product at Microsoft for building native client applications that run on Android, iOS, macOS, and Windows. For desktop platforms, you get the most modern toolkit in WinUI with Windows App SDK and Mac Catalyst for macOS. This is more than just bringing .NET MAUI mobile apps to the desktop. We have added support for multiple app windows; New in .NET 7 are desktop-specific controls for app-level menus, context menus, and tooltips, plus gestures for mouse hover and right-click. The true innovation in .NET MAUI is enabled by the **BlazorWebView:** Blazor hybrid apps.

BlazorHybrid

If you've heard of hybrid apps before, withhold judgment; this is not the same. Why? Because in the end, it's all .NET. Here's how this works, as seen in our .NET Podcast open-source app.

You start with the usual .NET MAUI application and enable the Blazor integration in the **MauiProgram** builder.

```
public static MauiApp CreateMauiApp() {
  var builder = MauiApp.CreateBuilder();
    builder.UseMauiApp<App>();

    builder.Services.AddMauiBlazorWebView();
  return builder.Build();
}
```

Add a **BlazorWebView** to the page and reference your razor file.

```
<ContentPage>

  <b:BlazorWebView HostPage="wwwroot/index.html">
      <b:BlazorWebView.RootComponents>
          <b:RootComponent Selector="app"
            ComponentType="{x:Type local:Main}" />
      </b:BlazorWebView.RootComponents>
  </b:BlazorWebView>

</ContentPage>
```

> Any barrier between the code running inside the BlazorWebView and the rest of the app is imagined. There's no need for a JavaScript bridge.

From there, you build your UI using Blazor components and blend in native pages of UI components as needed. Because everything compiles to .NET, the Blazor components have all the same access to device services, file system, sensors, secure storage, app menus, and more, that you would nor-

Listing 3: DiscoverPage.razor

```
code Listing (more than 15 lines)
         1         2         3         4         5         6
1234567890123456789012345678901234567890123456789012345678901234567

<PageTitle>.NET Podcasts - Discover</PageTitle>
<SearchBar OnSearch="@FetchShows" />
    <div class="containerPage containerPage--featured">
        <div class="categories">
            <NavLink href="categories" class="appLink"
                     title="See all categories">
                See all categories
            </NavLink>
            <Tags Items="@topCategories" TItem="Category">
                <ItemTemplate Context="item">
                    <NavLink href="@($"category/{item.Id}")"
                             title="@item.Genre">
                        @item.Genre
                    </NavLink>
                </ItemTemplate>
            </Tags>
        </div>

            foreach (var group in groupedShows)
            {
                <div class="titleWrapper">
                    <TitlePage Label="@group.Key" />
                </div>

                <Grid Items="@group.Value" TItem="Show">
                    <ItemTemplate Context="item">
                        <NavLink @key="item.Id"
                                 href="@($"show/{item.Id}")">
                            <ShowCard Id="@item.Id"
                                      Title="@item.Title"
                                      Author="@item.Author"
                                      Image="@item.Image" />
                        </NavLink>
                    </ItemTemplate>
                    <EmptyResults></EmptyResults>
                </Grid>
            }
    </div>
```

mally access in a .NET MAUI app. The only difference is that the UI of a Blazor component is rendered via HTML. Any barrier between the code running inside the **BlazorWebView** (aka browser) and the rest of the app is imagined. There is no need for the JavaScript bridge that's common in other hybrid scenarios.

This excerpt from **Listing 3** displays a list of programs that are retrieved right in the razor file from a native **HttpClient** request.

```
protected override async Task OnInitializedAsync()
{
  persistingSubscription =
    ApplicationState
      .RegisterOnPersisting(PersistShows);
  if(!ApplicationState
    .TryTakeFromJson<Show[]>("shows",
      out var restored)){
        allShows = await PodcastService
          .GetShows(MaxShows, null);
  }else{
    allShows = restored!;
  }
  UpdateGroupedShowsAndCategories(allShows);
}
```

This code is 100% shared between the web app and the mobile and desktop .NET MAUI apps. You need write no JavaScript to make this work because it's all running on a device in the .NET process.

Visual Studio 2022 supports hot reload for this scenario as well, and you can create your Blazor components with CSS styling and share them across all your .NET apps in the browser, mobile, and desktop.

Desktop Specific Controls

For most of your desktop scenarios, **BlazorWebView** may be the best solution, and you get the bonus of 100% code share with your web apps. .NET MAUI also has you covered when you want deeper, desktop native features like app level menus. And, new in .NET 7, we've responded to the feedback from our customers targeting desktop with .NET MAUI to add context menus, tooltips, and mouse-related gestures for hover and right-click. .NET MAUI doesn't just give you mobile experiences on the desktop, but rich desktop client experiences to support your demanding business requirements.

To add app-level menus to your application, from any **ContentPage**, you can add a hierarchy of menu items and wire them to events or bind to commands. In the .NET Point of Sale sample app, for example, I added a menu (see **Figure 7**) for quick access to adding new products to the menu.

```
<ContentPage.MenuBarItems>
  <MenuBarItem Text="Products">
    <MenuFlyoutItem Text="Add Product"
        Command="{Binding AddProductCommand}"/>
    <MenuFlyoutItem Text="Add Product Category"/>
  </MenuBarItem>
</ContentPage.MenuBarItems>
```

Using a similar API, you can add menus to any control in .NET 7 and they'll be revealed in a context menu (see **Figure 8**) when you right-click.

```
<Editor>
    <FlyoutBase.ContextFlyout>
        <MenuFlyout>
            <MenuFlyoutItem Text="Bold"
                Clicked="OnBoldClicked"/>
            <MenuFlyoutItem Text="Italics"
                Clicked="OnItalicsClicked"/>
            <MenuFlyoutItem Text="Underline"
                Clicked="OnUnderlineClicked"/>
        </MenuFlyout>
    </FlyoutBase.ContextFlyout>
</Editor>
```

Tooltips can provide simple contextual clues in your UI, and then with **PointerGesture**, you can add other visual animations to enhance the UI, such as a color change or underline or scaling—anything you can imagine.

```
<Label
    Text="https://docs.microsoft.com/dotnet/maui"
    ToolTipProperties.Text="dotnet/maui">
```

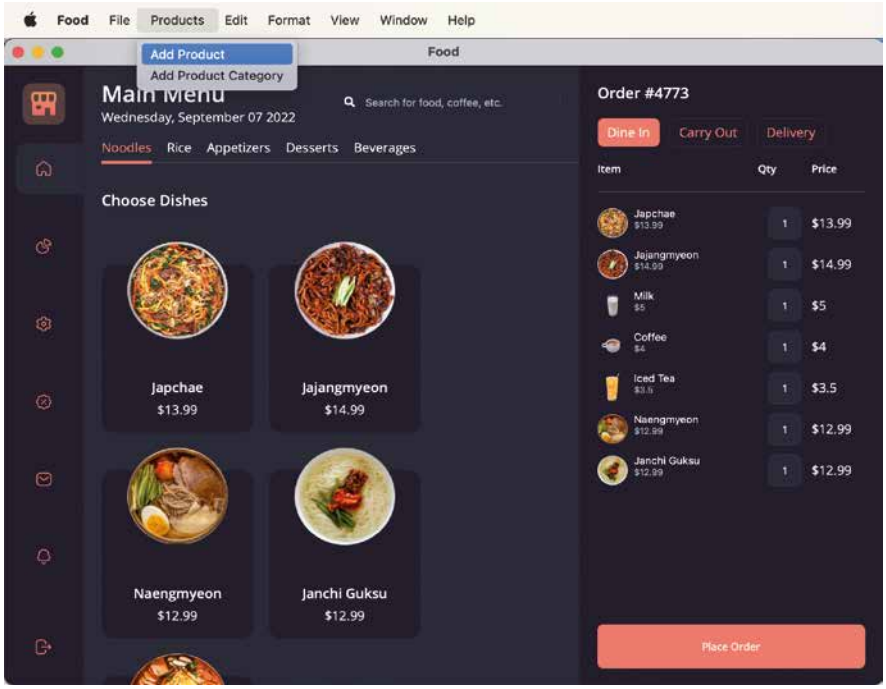

Figure 7: .NET MAUI can display app level menus like this on macOS.

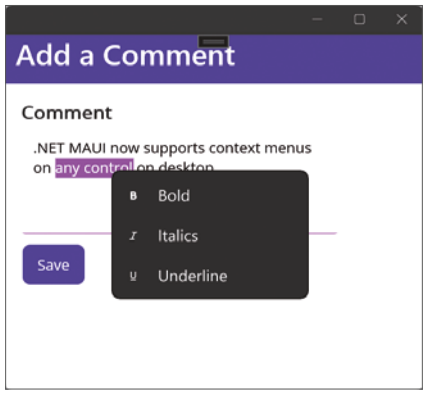

Figure 8: Display a context menu by attaching the new ContextFlyout control to a target control.

```
    <Label.GestureRecognizers>
        <PointerGestureRecognizer
            PointerEntered="HoverBegan"
            PointerExited="HoverEnded"
            PointerMoved="HoverMoved" />
    </Label.GestureRecognizers>
</Label>
```

Developer Tooling

Visual Studio 2022 on Windows and Mac provides the most productive developer experience for .NET MAUI, beginning with hot reload. Whether you use XAML for UI or C# for everything, as you make changes to your debugging app, you can apply those changes and see them live. This saves countless hours in stopping, rebuilding, and deploying changes just to see the differences made. To get started, you only need to start debugging your app (aka F5) from Visual Studio. Once debugging, the output indicates that hot reload is connected.

XAML Hot Reload

XAML changes are reloaded in your app as soon as they're valid, so no saving is required. You'll also notice that your UI and app state are preserved as you make changes, which is another massive time saver. Of course, editing a running application is tricky, and invalid code changes could cause the app to crash, so those won't be reloaded. Watch first for squiggles in your XAML indicating a syntax error to see when this happens. Then check the errors panel for any **XHR** errors that report why a XAML Hot Reload may not have been successful. And finally, the XAML Hot Reload output panel (see **Figure 9**) shows a running log of all activity related to your session.

.NET Hot Reload

C# changes are reloaded a bit differently. Once you have a change that you wish to apply, click the hot reload flame button near the debug button in the Visual Studio toolbar. Alternatively, you can tell Visual Studio to apply changes on file save. If the changes are valid and successful, all will be well, and you can proceed to retrigger the code path that you changed to see the new behavior. Visual Studio otherwise indicates if the changes could not be applied in a modal and give you the next steps to choose from.

Notice that I said you need to retrigger the code path? In a case where you edit a click handler, you click the button again. If you want to rerun the same method, like a UI build method, when you make a change, you can tap into a .NET hot reload handler event. To do this in your application, add a class that implements the **MetadataUpdateHandlerAttribute** like **Listing 5**.

In your **ContentPage**, listen for the **UpdateApplicationEvent** and then execute the method you want to retrigger, such as **Build()**.

```
HotReloadService.UpdateApplicationEvent += (obj)=>
{
    MainThread.BeginInvokeOnMainThread(() =>
    {
        Build();
    });
}
```

In this way, the **Build()** method is always called anytime a .NET hot reload is applied to the app, which is more convenient than navigating away and back or adding an extra button click to retrigger the method. When you're done working on that page and method, you can simply comment it out. It's also a good idea to wrap such things in **#if DEBUG** so you don't accidently release an app with this code.

Live Preview and Live Visual Tree

One of the hottest new features for developers in Visual Studio is the ability to see your app right inside the IDE, zooming in on the area you're actively working on, aligning to the very pixel, and even inspecting UI to navigate to code. This is another of the many massive time savers, especially when you might be working on a code base with which you are unfamiliar. Start debugging your .NET MAUI application and open the XAML Live Preview panel, as shown in **Figure 10**.

Click the target tool in the top toolbar of the panel and roll over the UI to see details about the controls. Find any con-

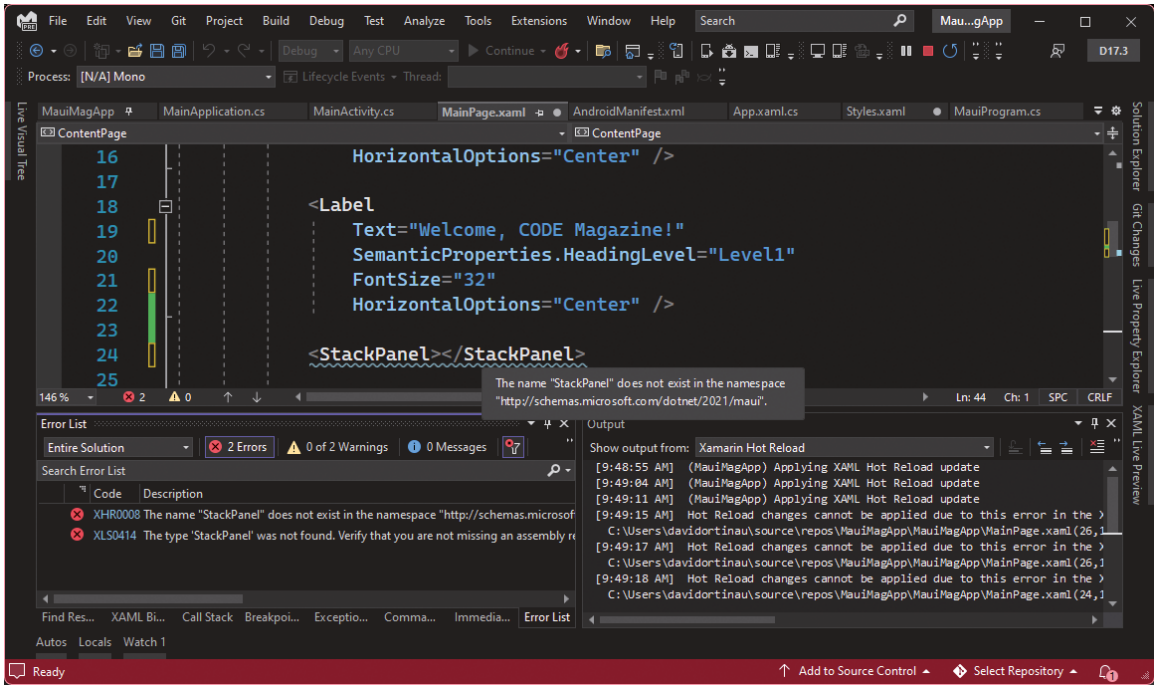

Figure 9: XAML Hot reload errors display in the error panel, with squiggles in the editor and status in the output panel

Listing 4: PodcastService.cs

```
code Listing (more than 15 lines)
        1         2         3         4         5         6
1234567890123456789012345678901234567890123456789012345678901234567

using System.Net.Http.Json;

namespace Podcast.Shared;

public class PodcastService
{
    private readonly HttpClient _httpClient;

    public PodcastService(HttpClient httpClient)
    {
        _httpClient = httpClient;
    }

    public Task<Category[]?> GetCategories() =>
        _httpClient.GetFromJsonAsync<Category[]>("v1/categories");

    public Task<Show[]?> GetShows(int limit, string? term = null) =>
        _httpClient.GetFromJsonAsync<Show[]>
                        ($"/v1/shows?limit={limit}&term={term}");

    public Task<Show[]?> GetShows(int limit, string? term = null,
                                  Guid? categoryId = null) =>
    _httpClient.GetFromJsonAsync<Show[]>
        ($"/v1/shows?limit={limit}
                    &term={term}
                    &categoryId={categoryId}");

    public Task<Show?> GetShow(Guid id) =>
        _httpClient.GetFromJsonAsync<Show>($"v1/shows/{id}");
}
```

trol and click it. This navigates you to the Live Visual Tree, a tree view representation of the UI hierarchy. To then go further and see the line of code where the control resides and click the eye icon beside it in the tree. From the code, you can hot reload and continue forward building your cross-platform application.

Enterprise Patterns and Practices

Writing robust yet performant code that can be readily tested is a must for most organizations. .NET MAUI fully supports the Model-View-ViewModel (MVVM) architectural pattern for separation of concerns, data binding, commanding, dependency injection, and a loosely coupled messaging bus. With this, you can roll your own app architecture or add on community libraries to make your development even nicer. If you're looking for a deep dive into enterprise development with .NET MAUI, check out the e-book "Enterprise Application Patterns Using .NET MAUI" available at https://docs.microsoft.com/en-us/dotnet/architecture/maui/ and downloadable from https://aka.ms/maui-ebook.

Listing 5: HotReloadService.cs

```
#if DEBUG
[assembly: MetadataUpdateHandlerAttribute(
  typeof(HotReloadService))]
namespace MauiApp {
  public static class HotReloadService {
    public static event Action<Type[]?>? UpdateApplicationEvent;

    internal static void ClearCache(Type[]? types) { }
    internal static void UpdateApplication(Type[]? types) {
      UpdateApplicationEvent?.Invoke(types);
    }
  }
}
#endif
```

The most popular MVVM libraries available work with .NET MAUI, and now the Microsoft Community Toolkit has shipped an MVVM library that includes some fantastic source generators to help you deliver better code faster. Consider a common property and command you might add to a **ViewModel** in your application that you would then bind to a control

4 page M
– Artwo

MS insert
rk coming

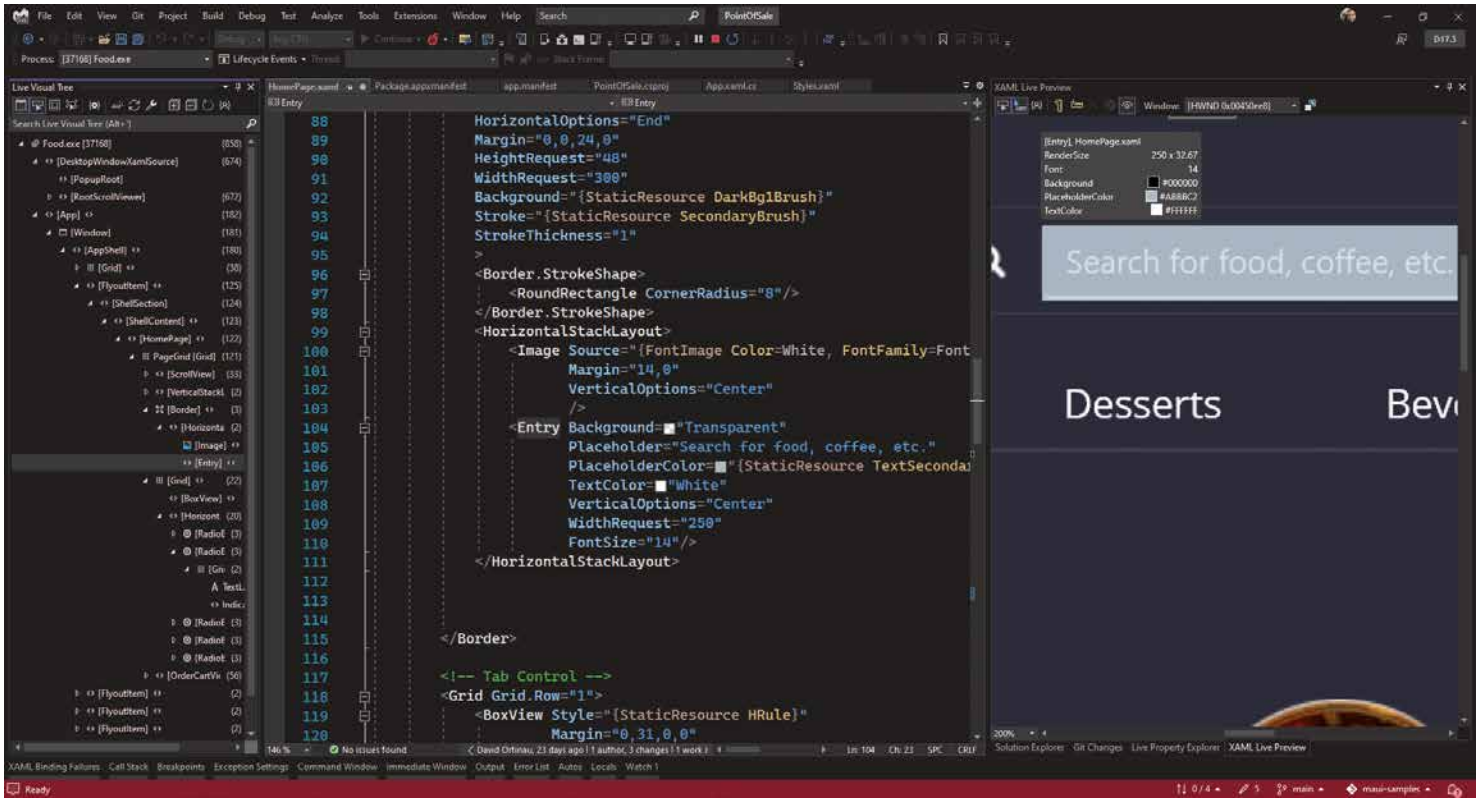

Figure 10: XAML Live Preview shows and inspects your running application.

Sample Code

You can get the code at https://github.com/microsoft/dotnet-podcasts and https://github.com/dotnet/maui-samples

for display. Typically, you'd need to manually implement the **INotifyPropertyChanged** interface and express the entire property with getter, setter, a backing property, and implementations. Let's compare the differences.

Compare **Listing 2** with the simplicity of the same **HomeViewModel** using the .NET Community Toolkit MVVM helpers:

```
[INotifyPropertyChanged]
public partial class HomeViewModel
{
    [ObservableProperty]
    ObservableCollection<Item> _products;

    [RelayCommand]
    async Task AddProduct()
    {
        await MessagingCenter
            .Send<HomeViewModel, string>
                (this, "action", "add");
    }
}
```

C# generators create a partial class (notice that this class is partial) to fill out the code that you don't have to write for the properties and commands making them bindable! This is wonderful for many reasons, not least of which is that you get IntelliSense in the UI for those generated methods. You can bind the list control to a public property called **Products** and then bind the menu item to a command called **AddProductCommand**. Even if you don't know or remember those naming conventions, the IntelliSense provided by Visual Studio based on the generated partial classes will easily guide you. This is less code for you to maintain!

The toolkit does much more than this, so be sure to check it out in-depth at https://docs.microsoft.com/dotnet/communitytoolkit/mvvm/.

Get Started Today

This is an all-new beginning for building powerful client applications for your businesses with .NET. Each year, you'll get the newest C# language features along with a faster, more secure .NET platform with which to build mobile and desktop applications using .NET MAUI. Begin your journey today at https://dot.net/maui.

.NET MAUI is the only first-party supported framework for building cross-platform apps, and I encourage you to bookmark our official support policy page at https://aka.ms/maui-support-policy. With each annual major release of .NET, we support .NET MAUI for 18 months in accordance with our Modern Lifecycle Policy. Every other release of the .NET SDK and runtime is covered by a three-year-long-term support policy. You'll never see a gap in coverage for the SDKs you depend upon.

Our .NET 7 release focus for .NET MAUI is to help developers upgrade their mobile applications from .NET Framework to .NET, and then enabling our five-million-plus and growing .NET developers to see that they too are .NET MAUI developers.

David Ortinau

Minimal APIs: Stuck in the Middleware Again

No matter how you build your APIs, you'll need certain functionality that's more than just in the body of a method. ASP.NET Core allows you to add this functionality through middleware. Let's see how that middleware can interact with Minimal APIs. Last year, I showed you how to write and structure your own Minimal APIs (https://www.codemag.com/Article/2201081/Minimal-APIs-in-.NET-6).

What was missing in that article is how to opt into middleware functionality. Let's dig into middleware and Minimal APIs.

Middleware in ASP.NET Core

Before I dig into using middleware with Minimal APIs, let me explain what middleware is. Sometimes you don't need a twenty-five-cent word to explain a simple concept. Middleware allows ASP.NET Core to execute a piece of code. That code could be part of ASP.NET Core, like Static File support, Authentication support, or even Routing. That code could be some third-party library or even your own code. For example, here's a simple top-level file that new .NET 6 projects use to listen for requests:

```
var bldr = WebApplication.CreateBuilder(args);

// Add services to the container.
bldr.Services.AddRazorPages();

// Get the web application object
var app = bldr.Build();

// Add Middleware
app.UseStaticFiles();
app.UseRouting();
app.MapRazorPages();

// Start Listening for requests
app.Run();
```

Once the app object is created by building the web application (to contain any required services), you can add middleware that will be called, in order, to handle the request. The order of this middleware matters, as this is the specific order that the request is passed into each part of the middleware. As you can see in **Figure 1**, the order of the middleware is stacked together:

As a request comes in, each middleware looks at the request and tries to determine if it can handle the request. In **Figure 2**, you'll see that the request can be handled by the Razor Pages middleware.

The request is passed to Static Files but it determines that it isn't a request that Static Files can handle, so it passes it to the Routing middleware. The Routing middleware determines that it can't handle it either, so it passes it to the next middleware: Razor Pages. Once the Razor Pages middleware realizes that it can handle the request, it fulfills the request and returns, which passes the request to the last middleware that called it. This continues the chain until the request is ultimately returned to the user.

At any point, middleware can fulfill the request and, in that case, it doesn't call the next piece of middleware. Instead, it just returns the chain back up, as seen in **Figure 3**.

Although these pieces of middleware may do a little or quite a lot, it's a black box for most of us developers. But what's really happening in the middleware? Let's write a tiny piece of middleware and find out.

If you call the Use method on the application, you can include a lambda to be called during a request:

```
app.Use(async (ctx, next) =>
{
  await next.Invoke(ctx); // Pass the context
});
```

You'll notice that you're passed two parameters when the lambda is called. The first parameter is the context object (**HttpContext**) that gives you access to the request and response objects. The second parameter is a **RequestDelegate**. This is how you will call the next piece of middleware. Notice that middleware has no idea of the order of middleware; it just gives you a bite at the apple of the request. Not all middleware is there to handle a request. For this example, let's write to the log with the speed of the request:

```
app.Use(async (ctx, next) =>
{
  // Get a starting time
  var start = DateTime.UtcNow;

  // Call the next piece of middleware
  await next.Invoke(ctx);

  // Execute code as the chain returns
  // back up the list of middleware.
  var totalMs = (DateTime.UtcNow - start)
                .TotalMilliseconds;
  app.Logger.LogInformation(
    @$"Request {ctx.Request.Path}: {totalMs}ms");

});
```

Now that you've had a brief introduction to middleware, let's talk about how it works with Minimal APIs.

Using Middleware with Minimal APIs

Although some middleware is about answering requests (e.g., Razor Pages, Controllers, and Static Files), other middleware is to provide services to other middleware. To

Shawn Wildermuth
shawn@wildermuth.com
wildermuth.com
@shawnwildermuth

Shawn Wildermuth has been tinkering with computers and software since he got a Vic-20 back in the early '80s. As a Microsoft MVP since 2003, he's also involved with Microsoft as an ASP.NET Insider and ClientDev Insider. He's the author of over twenty Pluralsight courses, written eight books, an international conference speaker, and one of the Wilder Minds. You can reach him at his blog at http://wildermuth.com. He's also making his first, feature-length documentary about software developers today called "Hello World: The Film." You can see more about it at http://helloworldfilm.com.

accomplish this, you need to opt-in or provide data to be used by the middleware. You can see this in one of the most common cases with Authorization. If you were writing a controller-based API, you might use an attribute to tell the Authorization middleware that authorization is required by a particular controller or action:

```
[Route("{moniker}/api/me")]
[Authorize]
[ApiController]
public class MeController : Controller
{
```

The use of the Authorize attribute allows the controller to opt into requiring authorization. But how do you accomplish this with Minimal APIs? If you have a Minimal API that requires authorization, you can still use the attribute:

```
app.MapGet("api/films",
  [Authorize]
  async (BechdelDataService ds,
         int? page,
         int? pageSize) =>
  {
    FilmResult data = await ds.LoadAllFilmsAsync();
    if (data.Results is null) {
      return Results.NotFound();
    }
    return Results.Ok(data);
  });
```

You should notice that the attribute is on the method (the anonymous method in this example). Unlike with controllers, there isn't a good way to specify the attribute for a group of Minimal APIs. I find the attribute method clunky and, evidently, so did Microsoft, as they recommend another way.

Instead of using an attribute, the more common way is to use a fluent syntax for the Minimal API:

```
app.MapGet("api/films",
  async (BechdelDataService ds,
         int? page,
         int? pageSize) =>
  {
    FilmResult data = await ds.LoadAllFilmsAsync();
    if (data.Results is null)
    {
      return Results.NotFound();
    }
    return Results.Ok(data);
  }).RequireAuthorization();
```

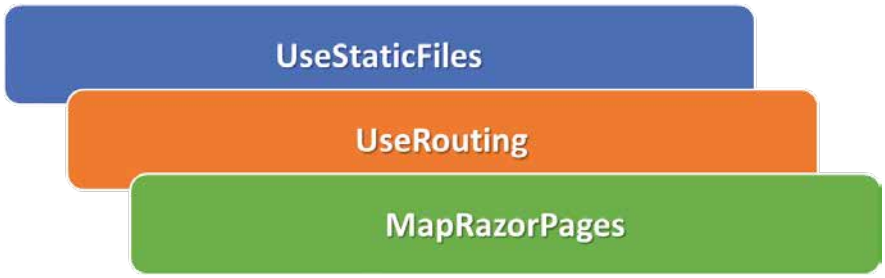

Figure 1: The Middleware

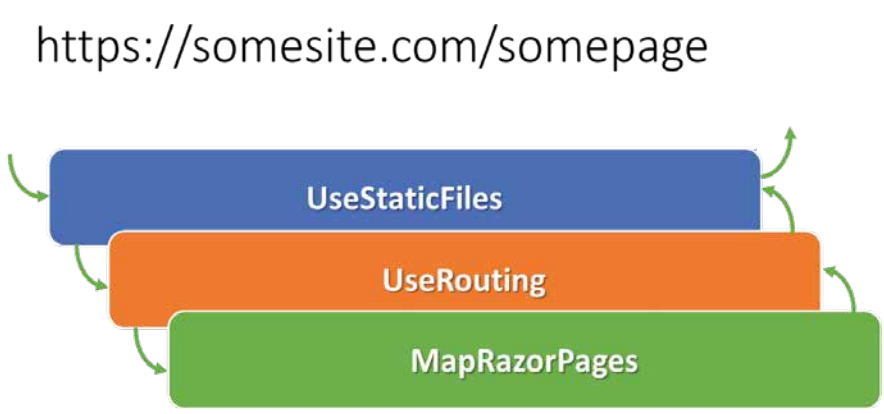

Figure 2: Chain of Middleware

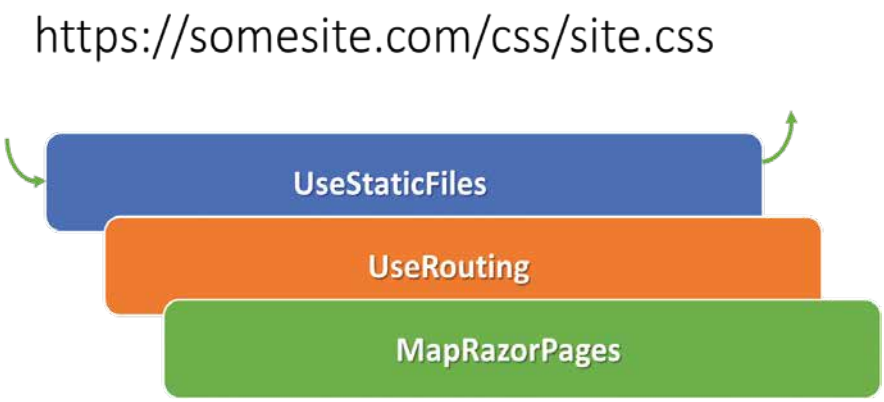

Figure 3: Short Circuiting Middleware

In this way, most middleware supports a fluent syntax (via extension methods) to the call to **MapXXX** to add information to the middleware. For example, you can also allow specific APIs to be accessible without authorization by using the **AllowAnonymous** method:

```
app.MapGet("api/years",
  async (BechdelDataService ds) =>
  {
    var data = await ds.LoadFilmYears();
    if (data is null) Results.NotFound();
    return Results.Ok(data);
  }).AllowAnonymous();
```

This is a fluent syntax so you can chain these together:

```
app.MapGet("api/years",
  async (BechdelDataService ds) =>
  {
    var data = await ds.LoadFilmYears();
    if (data is null) Results.NotFound();
    return Results.Ok(data);
  }).AllowAnonymous()
    .RequireHost("localhost");
```

Let's walk through some common middleware to see how it's used in minimal APIs.

CORS

In the case of CORS (or cross-origin resource sharing), often you'll only have a single policy defined:

```
app.UseCors(cfg =>
{
  cfg.WithMethods("GET");
  cfg.AllowAnyHeader();
  cfg.AllowAnyOrigin();
});
```

In some cases, you'll be using CORS policies and want to opt into individual ones. For example, if you configured CORS to have a partner policy for certain APIs, it would look like this:

```
builder.Services.AddCors(cfg =>
{
  cfg.AddDefaultPolicy(cfg =>
  {
    cfg.WithMethods("GET");
    cfg.AllowAnyHeader();
    cfg.AllowAnyOrigin();
  });

  cfg.AddPolicy("partners", cfg =>
  {
    cfg.WithOrigins("https://somepartnername.com");
    cfg.AllowAnyMethod();
  });
});
```

Although the default policy applies to all the **MapGet** calls, you might want to support all methods if you're from a partner website (i.e., "partners" policy). To opt into that, make a fluent call on the Minimal API:

```
app.MapGet("api/films/{year:int}",
  async (BechdelDataService ds,
         int? page,
         int? pageSize,
         int year) =>
{
  FilmResult data =
    await ds.LoadAllFilmsByYearAsync(year);
  if (data.Results is null)
  {
    return Results.NotFound();
  }
  return Results.Ok(data);
}).RequireCors("partners");
```

Swagger

Using **OpenAPI** (through Swagger) has become an important way to document and test your APIs. Minimal APIs support this with a set of fluent syntax methods. This allows you to annotate the minimal API with information that will be available to **OpenAPI** tooling as well as if you're supporting the Swagger UI plug-in.

First, there are methods for adding information about what the API produces. You could specify just the result codes like so:

```
app.MapGet("api/films/{year:int}",
  async (BechdelDataService ds,
         int? page,
         int? pageSize,
         int year) =>
{
  FilmResult data =
    await ds.LoadAllFilmsByYearAsync(year);
  if (data.Results is null)
  {
    return Results.NotFound();
  }
  return Results.Ok(data);
}).Produces(200);
```

You can specify the types it produces as well (**FilmResult** in this example):

```
app.MapGet("api/films/{year:int}",
  async (BechdelDataService ds,
         int? page,
         int? pageSize,
         int year) =>
{
  FilmResult data =
    await ds.LoadAllFilmsByYearAsync(year);
  if (data.Results is null)
  {
    return Results.NotFound();
  }
  return Results.Ok(data);
}).Produces<FilmResult>(200, "application/json");
```

Notice that you can specify the type, the result code, and the MIME type. In addition, you'll want to explain problem codes as well:

```
app.MapGet("api/films/{year:int}",
  async (BechdelDataService ds,
         int? page,
         int? pageSize,
         int year) =>
{
  FilmResult data =
    await ds.LoadAllFilmsByYearAsync(year);
  if (data.Results is null)
  {
    return Results.NotFound();
  }
  return Results.Ok(data);
}).Produces<FilmResult>(200, "application/json")
  .ProducesProblem(404);
```

This produces metadata so that callers can expect certain types of problem results. You can also add metadata for API names and tags (used for grouping):

```
app.MapGet("api/films/{year:int}",
  async (BechdelDataService ds,
         int? page,
         int? pageSize,
         int year) =>
{
  FilmResult data =
    await ds.LoadAllFilmsByYearAsync(year);
  if (data.Results is null)
  {
    return Results.NotFound();
  }
  return Results.Ok(data);
}).Produces<FilmResult>(200, "application/json")
  .ProducesProblem(404)
```

Source Code

The source code can be downloaded at https://github.com/wilder-minds/minimalapi-middleware.

```
  .WithName("GetAllFilms")
  .WithTags("films");
```

In this way, these methods can be used to add metadata about your API to let your users know what to expect. Although these work with Swagger, there are other methods (e.g., **WithMetadata** and **WithDisplayName**) that don't show up in the Swagger implementation.

OutputCaching

If you're familiar with ASP.NET before .NET Core, you might be used to using output caching. This allows you to cache the output of a request so that subsequent calls could just return the result. In .NET 7, this comes back and is supported via the **CacheOutput** method:

```
app.MapGet("api/films",
  async (BechdelDataService ds,
         int? page,
         int? pageSize) =>
  {
    FilmResult data = await ds.LoadAllFilmsAsync();
    if (data.Results is null)
    {
      return Results.NotFound();
    }
    return Results.Ok(data);
  }).CacheOutput();
    // Or .CacheOutput("somepolicy")
    // for specific output caching needs
```

You can see that you can either use the default output caching policy by providing no parameter, or you can supply a named policy. This allows you to handle output caching on an API-by-API basis. This is often only useful on MapGet APIs.

ResponseCaching

Like output caching, response caching is a way to help decrease the load on a server. It does this by using caching HTTP headers. If you've opted into using Response Caching, you'll need a way to specify the response caching policy for your Minimal APIs. Unfortunately, at least in the .NET 7 previews, there aren't methods for adding response caching. Instead, you'll have to use the attributes, but luckily this still works:

```
app.MapGet("api/films",
  [ResponseCache(Duration = 5)]
  async (BechdelDataService ds,
         int? page,
         int? pageSize) =>
  {
    FilmResult data = await ds.LoadAllFilmsAsync();
    if (data.Results is null)
    {
      return Results.NotFound();
    }
    return Results.Ok(data);
  });
```

In this case, you're telling the response caching to add a **Cache-control** header with a **max-age** of five seconds. So, to use Response Caching, you'll continue to operate like you did in controller-based APIs from earlier versions of ASP.NET Core.

Where Are We?

Although this article isn't a comprehensive list of how to opt into features of built-in middleware, I hope it provides a hint as to how the interaction of middleware and Minimal APIs can work together. For middleware authors, supporting extension methods to allow Minimal APIs to provide you with information about an API is something you should consider. For middleware users, you've seen that if a type of middleware doesn't directly support Minimal APIs, you can still fall back to using attributes to attain the same functionality.

Shawn Wildermuth

EF Core 7: It Just Keeps Getting Better

Lately, it seems that each iteration of EF Core brings fabulous new features and improvements. That has a lot do with the fact that the team has made a big investment in creating a stable base to build on. Although EF Core 7 is being released alongside .NET 7 and ASP.NET Core 7, it targets .NET 6, which is the long-term support version of .NET. So you can continue using it on a supported version of .NET for that longer term. I've been overwhelmed in trying to choose which of its features to share with you here. There are so many that are interesting. Not only does it mean writing about them, but I also get to test them all out, which is quite a lot of fun, thanks to the fact that I don't have to do so with the goal of releasing production code.

You'll find this article filled with some of the features that will be most impactful to the bulk of dev teams as well as a few that I personally found interesting.

Although I will always refer to this version as EF Core 7, keep in mind that much of the documentation and other resources will use **EF7** as its nickname. I still remember that first version of EF Core, just after EF6, which had a working name of **EF7** until it became **EF Core**. So, I may wait until EF Core 8 to use the new nickname.

Faster and Faster!

Back in 2021, one of the biggest stories for EF Core 6 was the dramatic performance improvement for non-tracking queries. At that time, the team committed to focusing on improving the performance of other workflows in EF Core 7. And true to their word, there was a lot of work done on updates that EF Core sends to the database.

Shay Rojansky, who has become "the performance guy" on the EF team, has explored the many nooks and crannies within SQL sent to the database and other related areas, discovering many points at which efficiencies could be applied. Some of the inefficiencies he discovered hailed back to the early days of Entity Framework. In an EF Core Community Standup earlier this year, Shay walked us through a fascinating look at the discoveries he'd made and the tunings he applied. Each tuning may have only sped things up a small amount, but they do add up!

Although most of these tweaks are under the covers and you will benefit from them without having to take any action, I would like to highlight some of them for you. However, if you do want to geek out on these changes, I highly recommend watching the standup video here on YouTube (https://youtu.be/EXbuRVqxn2o).

Reducing Round Trips to the Database

Some of Shay's discoveries were nuances that I hadn't paid attention to. An interesting one is a drawback of EF Core's default transaction behavior. As you may know, EF Core wraps every command sent in SaveChanges inside a database transaction so that if one fails, they'll all roll back. If you only have one command being sent, the calls for the transaction aren't needed because there aren't other commands involved. Therefore, when SaveChanges involves only a single change, rather than sending the three commands (BEGIN TRANSACTION, the change command, and then COMMIT), EF Core 7 only sends the change command, cutting the chattiness down from three commands to one. And when comparing EF Core 6 to EF Core 7 where the database was on a remote server, Shay measured a 45% improvement on the SaveChanges call. Granted this was only a change from about 8 ms to 4 ms, but those do add up in a production application. This is a great example of the types of tweaks made to the updates.

Another tweak is related to inserts. You may be aware of another pattern that's been around since the beginning of EF, and which continued through EF Core, and that's that INSERT commands have always been paired with a SELECT to return the database-generated value of any primary or foreign key. These new keys were then applied to the related objects that EF was inserting. Although this doesn't require EF Core to make an additional call to the database, it does force the database to execute an additional command. Now with EF Core 7, the SQL Server provider compresses all of those into a single command by using an OUTPUT in the INSERT command, rather than an extra command to SELECT.

In other words, instead of this multi-command message from EF Core 6:

```
INSERT INTO [People] ([Name])
      VALUES (@p0);
      SELECT [PersonId]
      FROM [People]
      WHERE @@ROWCOUNT = 1
      AND [PersonId] = scope_identity();
```

EF Core 7 sends this:

```
INSERT INTO [People] ([Name])
      OUTPUT INSERTED.[PersonId]
      VALUES (@p0);
```

In the case of a single INSERT being sent where EF Core 7 won't wrap this in a transaction, the OUTPUT clause also removes the need for a transaction that was needed around the composed INSERT plus SELECT.

The improvements are not limited to when SaveChanges only sends a single change. Batched commands are also streamlined. Not only do they also lose the explicit BEGIN and COMMIT for transactions, but they're also expressed in a more efficient way.

And if you've ever used EF Core's HiLo feature, parent/child inserts can really benefit from it. Wait what? HiLo? Yeah, me too. I had completely forgotten about this feature, introduced in EF Core 3, to ask SQL Server to pre-generate a bunch of primary keys that EF Core caches and pushes into INSERT commands as needed. Here's the documentation for the UseHiLo method: https://docs.microsoft.com/en-us/dotnet/api/microsoft.entityframeworkcore.sqlserverprop-

Julie Lerman
@julielerman
thedatafarm.com/contact

Julie Lerman is a Microsoft Regional director, Docker Captain, and a long-time Microsoft MVP who now counts her years as a coder in decades. She makes her living as a coach and consultant to software teams around the world. You can find Julie presenting on Entity Framework, Domain-Driven Design and other topics at user groups and conferences around the world. Julie blogs at thedatafarm.com/blog, is the author of the highly acclaimed "Programming Entity Framework" books, and many popular videos on Pluralsight.com.

ertybuilderextensions.usehilo. If you've cached keys with HiLo, EF Core can use them for parent child inserts without having to first send the parent INSERT command in order to get the new primary key for the parent to use as the foreign key for the child object(s). This means that these inserts can now be sent as a batch, which reduces the save from four round trips in EF Core 6 to a single round trip in EF Core 7.

In addition to the video I linked to above, Shay Rojansky's blog post about EF Core 7 Preview 4 details many of these improvements. I'm a big fan of how he builds a story around the changes asking, "what about this?" and "what about that?" The blog post is here: https://devblogs.microsoft.com/dotnet/announcing-ef-core-7-preview6-performance-optimizations/

Some Other Notable Performance Enhancements

Speaking of batched commands, you may recall that EF Core batches commands that are sent with SaveChanges. Based on performance analysis by the team when first designing the feature, the SQL Server provider only batched commands if there were at least four being sent. That's been changed so that the minimum number of commands to batch is now two.

If you use Lazy Loading via the proxy generation workflow, there are also major performance improvements here. A user reported that enabling lazy loading proxies on a context was creating a huge performance problem for the model builder. That meant that the very first database interaction performed for that context in an application instance was taking an inordinate amount of time. The team changed how the model responded to proxy generation, which resulted in a complete reduction of that extra time caused by UseLazyLoadingProxies. Arthur Vickers relays the EF Core 6 vs. EF Core 7 timings on a complex model in this GitHub issue comment: https://github.com/dotnet/efcore/issues/20135#issuecomment-1141085764. The chart shows that in EF Core 6, the model in question took 13 times longer to generate when UseLazyLoadingProxies was enabled. In EF Core 7, the time for model generation was equal with or without the proxy method.

Finally, Bulk Updates and Deletes

On the topic of updates, EF Core 7 brings another long-requested feature for pushing changes to the database: bulk updates. How long? The GitHub issue (https://github.com/dotnet/efcore/issues/795) was opened in 2014.

Since the beginning of EF time, if you wanted to update a row in the database, you first had to query it, apply the changes to the object, and then call SaveChanges. If you wanted to delete a row, it's a similar workflow: retrieve the object, change its state to deleted, and then call SaveChanges.

Users have long wanted to be able to express something similar to a LINQ query to push the changes directly to the database—something that's more like how you can express updates and deletes in SQL.

After a lot of conversations with the community, the team decided to design this via ExecuteDelete and ExecuteUpdate methods that are appended to LINQ queries in the same way that you'd apply a LINQ execution method. And these are executed immediately, not stored in the change tracker awaiting a call to SaveChanges.

You would write a delete method like this:

```
context.People
  .Where(p => p.PersonId == 1).ExecuteDelete();
```

Although I'm only deleting a single row, you can write your expression so that the delete affects multiple rows.

Update is a little more complicated because it enables you to specify multiple updates to occur. Each change is encapsulated in a SetProperty method. Therefore, ExecuteUpdate expects SetProperty expressions and each SetProperty expression expects an expression with the property and the value. Here, I'm applying perhaps not the most brilliant logic to assume that Lehrman is always a misspelling of my last name and my relatives'.

```
context.People
  .Where(p => p.LastName == "Lehrman")
  .ExecuteUpdate (s =>
    s.SetProperty(c =>c.LastName, c =>"Lerman"));
```

The resulting SQL is as clear as if you'd written it yourself.

```
UPDATE [p]
SET [p].[LastName] = N'Lerman'
FROM [People] AS [p]
WHERE [p].[LastName] = N'Lehrman'
```

These are simple examples, but the team has worked out variations to handle relationships, inheritance, and other more complex scenarios. Check the documentation for more detailed examples.

Mapping Entity Properties to Database JSON Columns

Storing JSON data in a relational database is usually a matter of storing the objects as some flavor of text or char in the database. For example, I may have an nvarchar Measurements column in my People table with JSON data that looks like this:

```
{"HeightCM":188,"ShoeUK":7}
```

Most RDBMs, including SQL Server, have a way to query JSON-formatted data as JSON, not as text. I can write the TSQL to query for specific elements: Here I only want the HeightCM data, within the Measurements column:

```
SELECT personid, firstname,lastname,
       json_value(measurements,'$.HeightCM')
       as HeightCM
FROM people
```

In my .NET solution, I can create a Measurements type:

```
public class Measurements
{
    public int HeightCM { get; set; }
    public int ShoeUK { get; set; }
```

I can use JSON conversion anytime I want to store the data from type as a JSON string in my Person class. That way I can work with a tidy class in C# and still have my JSON formatted text stored in the database.

```
Person.Measurements =
  JsonSerializer.Serialize
  (new Measurements{HeightCM=188,ShoeUK=7});
```

When I retrieve Measurements in any query, I'll have to deserialize it back into the Measurements type to work with it in my code.

It's already cumbersome to serialize and deserialize, but worse yet is querying with LINQ. You have to query the string, not the type. But how? How can you use a string query method to retrieve just the HeightCM, or worse, to retrieve all people whose HeightCM is greater than 180? You can't do that with LINQ. Either you have to retrieve more data than you want and then apply the filter in the client-side code, or you have to send raw SQL, or perhaps use views or stored procedures.

Because of this, direct support for JSON columns has been a highly requested feature for EF Core. Finally with EF Core 7, it had risen to the top of the to-do list and thanks to work done by Maurycy Markowski on the EF Core team, it's supported in this version. According to Markowski, the feature is pretty basic in this version, but it provides the framework for deeper implementation in the future.

The keys to this support lay in the combination of leveraging EF Core-owned types and the database providers translating queries into SQL that reflects how their database queries JSON data.

This also means that you now have another way of persisting value objects with EF Core. Owned entities have given you a path for storing value objects in a relational database where the properties of the value object get split out into additional columns in the table along with the type that "owns" that property. Now the value object can be more neatly encapsulated into a JSON object in a single database column.

You need to apply two mappings to the Person.Measurements property in OnModeling. The OwnsOne mapping has an overload that allows you to further specify the relationship of the owned property using an OwnedNavigationBuilder. This builder has new overload allowing you to specify that the property is a JSON column.

```
modelBuilder.Entity<Person>()
 .OwnsOne(p => p.Measurements,
          jb => { jb.ToJson(); });
```

Here I add two new person objects:

```
var personA = new Person
{
    FirstName = "Maurycy",
    LastName = "Markowski",
    Measurements = new Measurements
     { HeightCM = 188, ShoeUK = 8 }};
var personB = new Person
{
    FirstName = "Katrina",
    LastName = "Jones",
    Measurements = new Measurements
      { HeightCM = 170, ShoeUK = 7 }};
```

Once I've added them to the context and called SaveChanges with this mapping in place, the Measurements data is compressed into JSON and stored into the Measurements column (**Figure 1**).

And because Measurements is a type in my system, I can construct queries that are aware of its properties.

```
var tallpeople = context.People
.Where(p=>p.Measurements.HeightCM>180)
.ToList();
```

The real magic comes in EF Core and the provider's ability to transform this into SQL. In this case, TSQL:

```
SELECT [p].[PersonId], [p].[FirstName],
  [p].[LastName],
  JSON_QUERY([p].[Measurements],'$')
FROM [People] AS [p]
WHERE CAST(JSON_VALUE
  ([p].[Measurements],'$.HeightCM') AS int)>180
```

More patterns related to this are supported, including collections and layers of objects (e.g., grandchildren) that are stored as tiered JSON documents in the nvarchar column. Check the documentation for further examples.

Mapping Stored Procedures Just Like EF6

In the original Entity Framework, you had the ability to map stored procedures to entities. When you called SaveChanges, as long as you followed the basic rules, EF called your stored procedures, pushing in the parameters rather than generating its own SQL. Bringing this feature to EF Core has been on the back burner for quite some time but now, with more critical features out of the way, the team has implemented this capability into EF Core 7.

In EF, I recall some convoluted UI for doing this mapping, although I have zero impetus to pull out one of the old 1000-page EF books I wrote to remind myself how that worked.

It's much simpler in EF Core 7. There are simple and discoverable FluentAPI mappings called InsertUsingStoredProcedure, UpdateUsingStoredProcedure, and DeleteUsingStoredProcedure that you apply to an entity in OnModelCreating.

	PersonId	FirstNa...	LastName	Measurements
▶	1	Katrina	Jones	{"HeightCM":170,"ShoeUK":7}
	2	Maurycy	Markowski	{"HeightCM":188,"ShoeUK":8}

Figure 1: Measurements values are stored as JSON in the nvarchar column.

Each method takes a string to identity the procedure name and a StoredProcedureBuilder that's comprised of one or more parameters where you identify the entity properties that align with the parameters via matching names. Each method is constructed a bit differently based on its nature.

As an example, here is a simple insert stored procedure:

```
CREATE   PROCEDURE dbo.PeopleInsert
    @personid int OUT,
    @firstname nvarchar(100),
    @lastname nvarchar(100)
 AS
 BEGIN
   INSERT into [People] (FirstName, LastName)
      Values(@firstname, lastname);
   SELECT @personid = SCOPE_IDENTITY();
 END;
```

The StoredProcedureBuilder for an insert starts with the procedure name and then the lambda for the StoredProcedureBuilder. For the parameters, I used lambdas to express each property of Person that maps to the parameters and did so in the order expected by the procedure. Additionally, I used an overload to further specify the first parameter—that it's an output parameter and because the parameter name doesn't match the property name, I specify that the name is "id". You may already have noticed this:

```
modelBuilder.Entity<Person>()
 .InsertUsingStoredProcedure("PeopleInsert",
    spbuilder => spbuilder
     .HasParameter(p => p.PersonId,
        pb => pb.IsOutput().HasName("id"))
     .HasParameter(p => p.FirstName)
     .HasParameter(p => p.LastName)
  )
```

If you're also mapping the update and delete, you can compose them together. Another improvement over the way this was in EF is that you aren't required to supply all of the mappings in order for any of them to work. For example, if I only have a mapping for updates, that procedure is used and EF Core composes SQL for inserts and deletes.

There are additional methods that you can use with the StoredProcedureBuilder besides HasParameter: HasOriginalParameter, HasResultColumn, and HasRowsAffectedParameter.

Enabling Value Generation on Value Converters Used for Key Properties

This is an important change to EF Core 7 for developers following practices learned from Domain-Driven Design (like me!). Let me start by explaining what this means. By now you know what key properties are in EF Core. Most often you see a key defined as an int or a GUID.

```
public class Person
{
  public int PersonId { get; set; }
```

Int is commonly used for relational databases that can generate those integers for you. GUIDs give you more control over keys on the client side. You can generate them at the same time you create new objects without waiting on the database to provide those values for you. In the scenario above, EF Core creates a temporary value for PersonId (seen only by EF Core's internals) while awaiting that database-generated value. You can at least make the setter private to protect from developers accidentally setting the PersonId property to some random value, which could cause a conflict in the database. There are ways around that protection. Imagine that you have so many people in your database that you run out of ints and decide to switch to GUIDs. That's a difficult change to make so far into your application's history.

A common practice, especially among developers following guidance and practices from Domain-Driven Design, is to create a value object that you use as the type for the key property.

Here's an example of a new type I created and named EntityKey that's then used as the type for the Person class' PersonId:

```
public class EntityKey
{
  public EntityKey(int id) => Id = id;
  public int Id { get; private set; }
}
public class Person
{
  public EntityKey PersonId { get; set; }
  . . .
```

This gives some nice advantages. For example, if I need to change the EntityKey Id property to a GUID, it won't impact the Person type at all. The Person type doesn't care about how EntityKey is implemented. Read more about some advantages of using Value Objects for key properties at Nick Chamberlain's blog post here: https://buildplease.com/pages/vo-ids/.

Back to EF Core. EF Core knows how to handle ints and GUIDS as keys but it doesn't know how to store your custom-generated EntityKey type. Value converters, introduced in EF Core 3, provide what looks like a possible solution. You can tell EF Core that when it's time to save a Person object, it should use the **Id** property of the PersonId property as the value to persist. And when querying Person types, EF Core should take the int that's stored in the table and create an EntityKey from it (using that constructor defined in EntityKey) then set that as the value of PersonId. All this is defined in this HasConversion-fluent API method.

```
modelBuilder
    .Entity<Person>()
    .Property(c => c.PersonId)
    .HasConversion(
        v => v.Id,
        v => new EntityKey(v))
    .ValueGeneratedOnAdd();
```

It's a brilliant solution, but up through EF Core 6, EF Core couldn't combine its value-generation capabilities with the conversion. Steve Smith, my brainy co-conspirator on the Pluralsight Domain-Driven Design Fundamentals, brought

this up way back in 2018 in this GitHub issue: https://github.com/dotnet/efcore/issues/12135. EF Core complained, saying that it doesn't have a value generator for EntityKey and the error message suggested that you should set the key's value in code.

Perhaps you've figured out where this is leading: EF Core 7 now supports the combination of value converters with value generation. You must have that ValueGeneratedOnAdd() method or you'll get a runtime exception saying that the ChangeTracker isn't able to track an EntityKey type.

Back to the Future: Intercepting Object Materialization and Other New Interceptors, Too

Why "back to the future"? For you long-time users of Entity Framework, before EF4 gave you POCO support and the DbContext, you had a more tightly coupled way of implementing EF using an ObjectContext. Through that API, you had access to an ObjectMaterialized event handler that allowed you to inject your own rules and logic as the object was being created from query results. If you wanted to access it when using EF6 DbContext, you'd have to drill into the low-level ObjectContext. But since EF Core, there is no ObjectContest and you've never had a way to override the behavior.

Until now. Huzzah! You finally have this capability with EF Core 7 by way of interceptors. Interceptors were another great feature of EF6 that took some time to find their way into EF Core 3. Now EF Core 7 adds a slew of new interceptors to allow you to add your own logic to low-level actions. These interceptors allow you to:

- Override object materialization
- Modify the LINQ expression tree
- Affect how optimistic concurrency is handled
- Tap into additional points in the lifecycle of connections and commands
- Muck with query result sets

Arthur Vickers details the various new interceptors in his blog post at https://devblogs.microsoft.com/dotnet/announcing-ef7-preview7-entity-framework/.

I'll focus here on the object materialization interceptor, aka the IMaterializationInterceptor. This interceptor allows you to tap into the pipeline before and after materialization. In other words, once EF Core has instantiated the object but hasn't yet pushed the query result values into it. I'll dig a little deeper into this interceptor, which should also give you an idea of how you can do the same with the other interceptors.

There are four interception points in this interceptor; before and after the new instance is created and, once created, before and after the instance is initialized.

Let's take a look at each of these methods.

I've created a class that implements the IMaterializationInterceptor interface (see Listing 1) and implemented all four methods without adding any of my own logic, so each returns either the InterceptionResult or the entity by default.

The value of the HasResult property of the result returned by CreatingInstance is false but it's materializationData object has access to the DbContext instance and the EntityType, which gives you the ability to affect anything within those objects. All of the methods expose the materializationData object.

When CreatingInstance is hit, you have access to the instance of the object being created, although its properties have not yet been populated. And it's this entity that's returned by default from the method.

Next, the InitializingInstance method gets hit immediately after the property values have been created but not yet populated. It returns an InterceptionResult (note that this one isn't generic) that has one read-only property, IsSurpressed, which is false. Query results overwrite any changes you make to the entity properties here. It's best to make those changes in the next method. However, if you have unmapped properties, any values you apply to them here will remain. In his Preview 7 blog post referenced above, Vickers uses an example of a property for audit data to note when the data was retrieved from the database. He then populates that Retrieved property in the InitializingInstance method.

Finally, there's the InitializedInstance that receives the populated entity object as a parameter.

Keep in mind that if you have related objects or an owned type, such as the Measurements type from the JSON column support example above, those will be materialized separately. Therefore, if you affect the Measurements property of a Person as the Person is being materialized, that property will be overwritten when the Measurements object is being materialized.

This interceptor isn't solely for modifying results or entities. You might use it to trigger application events or other relevant actions. But like any tool, whether for coding or building a doghouse, take care in how you apply it. As Khalid Abuhakmeh warns in a blog post about EF Core 5 interceptors (https://khalidabuhakmeh.com/entity-framework-core-5-interceptors), you should be careful about adding resource-intensive logic in any of the interceptors, as well as triggering unwanted side effects.

Listing 1: The IMaterializationInterceptor interface's returns

```
public class MyMaterializationInterceptor
 : IMaterializationInterceptor
{
  public InterceptionResult<object> CreatingInstance(
       MaterializationInterceptionData materializationData,
       InterceptionResult<object> result)=>result;

    public object CreatedInstance(
       MaterializationInterceptionData materializationData,
       object entity) =>entity;

    public InterceptionResult InitializingInstance(
       MaterializationInterceptionData materializationData,
       object entity, InterceptionResult result)=> result;

    public object InitializedInstance(
       MaterializationInterceptionData materializationData,
       object entity) => entity;
```

Support for Database Specific Aggregate Functions

EF Core is designed to enable common features across databases. EF Core 7 now allows database providers to expose provider-specific aggregates that the provider knows how to translate into their own flavor of SQL. You'll find these in the EF Functions extension that has already been exposing methods such as SQL Server's CONTAINS, RANDOM, and a number of date functions.

Thanks to the change in EF Core 7, the SQL Server provider adds string.Join, string.Concat, and some methods for you statistics seekers, methods to translate to some TSQL functions I've never used in my very lengthy career: StandardDeviationSample (STDEV in TSQL), StandardDeviationPopulation (STDEVP), VarianceSample (VAR), VariancePopulation (VARP). SQLite also benefits from string.Join and string.Concat.

Shay Rojansky is not only a member of the EF Core team but also a long-time maintainer of PostgreSQL providers for .NET, EF, and EF Core. In addition to working on the other functions, he has added quite a few aggregate methods to the PostgreSQL provider for EF Core for strings such as filtering and ordering, JSON, arrays, ranges, and some statistics as well. For the curious, learn more in this GitHub pull request discussion: https://github.com/npgsql/efcore.pg/pull/2383.

You may wonder how string.Join is new. You've always been able to write a query like this:

```
context.People
.Select(p=>
   string.Join(",",p.FirstName, p.LastName))
.ToList();
```

That returns a list of joined names such as:

```
Julie, Lerman
Shay, Rojansky
```

What's new here is that you can use them in GroupBy expressions. Here, for example is a query where I want to group by LastName and then create a comma-delimited list of the first names in that group.

```
Var groupedpeople = context.People
  .GroupBy(p => p.LastName)
  .Select(surname => new
  {
    Last = surname.Key,
    firstnames =
       string.Join(",",
         surname.Select(p => p.FirstName))
  })
.ToList();
```

Given that I've seeded the database with three people, two with the last name of Jones, here are the results of this query:

```
Jones: Katrina,Serena
Markowski: Maurycy
```

Shay discusses and demonstrates a number of the new aggregate features for EF Core 7 in the August 25, 2022 Community Standup here on YouTube: https://www.youtube.com/watch?v=IfaURw5D1Qg.

More EF 6 Parity

In each iteration of EF Core, the team works toward bringing more parity with features we loved and relied on from EF6. Here are some that have been implemented for EF Core 7.

Entity Splitting

Entity splitting is a mapping that allows you to persist properties of a single entity across multiple tables or views.

There are Fluent API and data annotation mappings for this. Here's an example of mapping a few properties from the Person type into a separate table called PeopleLastNames using the new SplitToTable method. I'm letting convention take care of naming the core table. For views, there's a method called SplitToView.

```
modelBuilder.Entity<Person>()
  .SplitToTable(
    "PeopleLastNames",
    s => s.Property(p => p.LastName)
  );
```

This mapping creates a second table with its own PersonId column that's a primary key as well as a foreign key pointing back to PersonId in the People table (**Figure 2**).

You can specify multiple properties to split out by using an expression function in the lambda as follows:

```
modelBuilder.Entity<Person>()
  .SplitToTable(
    "PeopleNames",
     s => { s.Property(p => p.LastName);
            s.Property(p => p.FirstName);
          }
  );
```

With an eye always on persisting classes that are designed following Domain-Drive Design, entity splitting also means that you now have a variety of ways to persist value objects:

- As separate columns in the same table as the host entity
- As a JSON document in a single column of the host entity's table

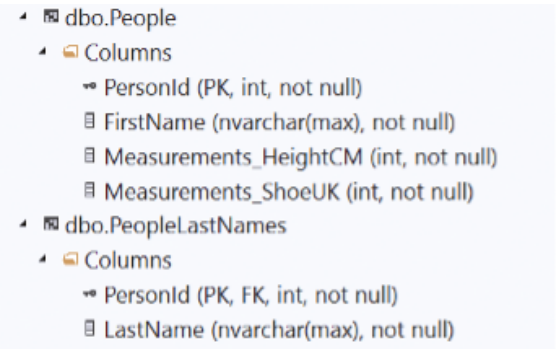

Figure 2: Entity Splitting creates a separate table for specified properties.

- As a separate table with individual columns for each property of the value object along with the primary key column
- As a separate table with the primary key column and a single column containing a representative JSON document

In the future, EF Core will let you more easily use value conversions to store value objects (https://github.com/dotnet/efcore/issues/13947) but the added flexibility in EF Core 7 via JSON column support and table splitting combined with owned-type support continue to allow you to persist value objects in relational stores.

EF Core 7 as a Part of Distributed Transactions

If you used EF6 or earlier versions, you may be familiar with the support for including EF's SaveChanges in Windows' distributed transactions. A limitation in .NET Core has prevented this for some time, but once again, Rojansky came to the rescue and fixed this in the dotnet runtime repository. You can read about this change at https://github.com/dotnet/runtime/issues/715. I haven't written about EF and distributed transactions in a long time (https://docs.microsoft.com/en-us/archive/msdn-magazine/2013/december/entity-framework-entity-framework-6-the-ninja-edition)! But now, with or without EF Core in the mix, you can combine transactions from a variety of systems in a single transaction.

Table Per Concrete Type (TPC) Mapping

TPC was always "the red-headed step-child" of inheritance mappings. In EF Core, as it was in EF, TPC was the last to be implemented and has been overlooked by many developers. Vickers reminds you that it's a much better strategy than the more popular Table Per Type (TPT). TPC was supported in EF6. EF Core arrived with Table per Hierarchy (all columns across an inheritance hierarchy in a single table). Then EF Core 5 brought you TPT (where the unique properties of each derived type are stored in their own tables). Now, finally, TPC has arrived with EF Core 7. TPC stores each complete derived type in its own table. To learn more, check out the EF Core 7 Preview 5 announcement blog post that reviews pros and cons of these various mappings (https://devblogs.microsoft.com/dotnet/announcing-ef7-preview5/).

Define Your Own Scaffolding Rules with T4 Templates

Do you remember T4 templates? They're yet another language syntax to learn but don't worry, it's not YAML. Templates are the underpinnings of how EF Core scaffolding is able to reverse-engineer databases into a DbContext and entity classes. Back in the days of yore when EF was not EF Core, you could use T4 templates to customize how to build your models from your databases. This is really handy when you find yourself adding (or removing) the same code time and time again after scaffolding a database.

Brice Lambson was then, as he is now, the go-to guy for T4 templating on the EF team. He and other team members showed off the work he's done on this feature in this April 2022 Community Standup (https://youtu.be/x2nh1vZ-BsHE), if you want to see some great demos of how you can generate your finely tuned DbContext and entity classes when scaffolding databases. There's tooling for doing this in Visual Studio 2022 (written by Brice) and command line tools where you can tell EF Core scaffolding to use your templates instead of the default. I wasn't surprised to see Arthur Vickers create a customization on DbSet. Arthur isn't a big fan of the null bang (!) used to satisfy null reference settings.

```
Public virtual DbSet<Person>
  People { get; set; } = null!;
```

Instead, he customized the template to use his preferred pattern by removing the getter and setter and just returning an instance of the DbSet.

```
Public virtual DbSet<Person> People
  => Set<Person>();
```

Override EF Core's Conventions with Your Own

EF Core 6 brought the ability to apply bulk configurations, which is wonderful. You can override the ConfigureConventions to apply things like HaveColumnType to all properties in the model that are strings, instead of doing it per property in each relevant entity.

But there was something else from EF6 that we've been hoping for: a more sweeping way to affect conventions. That's finally come to EF Core 7 and in fact, creating the ConfigureConventions method in EF Core 6 was part of the preparation for this feature, referred to as "public conventions" because the conventions are now publicly exposed.

With the conventions now public, you can now remove or replace built-in conventions as well as add completely new ones.

The set of built-in conventions is exposed in the ModelConfigurationBuilder that's passed into the Configure Conventions method. You can even take a look at them by drilling into the configurationBuilders Conventions property.

```
Protected override void ConfigureConventions
 (ModelConfigurationBuilder configurationBuilder)
{
  var cs = configurationBuilder.Conventions;
  base.ConfigureConventions(configurationBuilder);
}
```

Listing 2: The MaxStringLength200Convention class

```
public class MaxStringLength200Convention :
 IModelFinalizingConvention
{
  public void ProcessModelFinalizing(IConventionModelBuilder
    modelBuilder,
    IConventionContext<IConventionModelBuilder> context)
    {
      foreach (var property in
       modelBuilder.Metadata.GetEntityTypes()
       .SelectMany(entityType =>
        entityType.GetDeclaredProperties().Where(
         property => property.ClrType == typeof(string))))
        {
          property.Builder.HasMaxLength(200);
        }
    }
  }
}
```

The various conventions are grouped. For example, the ForeignKeyAddedConventions contains a collection of six conventions, one of which is the KeyDiscoveryConvention.

You can remove a convention, which also means that if you don't know what you're doing, you could really mess up your data model. But let's say you want database tables to match the names of the entity classes, not the DbSets. You could remove the TableNameFromDbSetConvention.

```
configurationBuilder.Conventions
 .Remove(typeof(TableNameFromDbSetConvention));
```

Or perhaps you have a configuration rule that needs to be applied in all of your DbContexts in all of your apps. Perhaps all strings for your SQL Server database should default to nvarchar(200), rather than nvarchar(max). You can create a convention for that in a class, add it to your project, and then add it to the configuration builder.

Adding conventions is a little more complicated because different conventions are applied at different stages of model building and these stages are identified via different interfaces. Quite often, it's simplest to add your conventions when the model is finished with applying conventions by implementing the IModelFinalizingConvention. Do keep in mind that mappings are always applied after conventions, so you might have mappings that override your custom conventions.

There's a new class to define a custom MaxLengthConvention limiting text-based data columns to 200. I can reuse this class across many DbContexts in one or more solutions.

My MaxStringLength200Convention class (**Listing 2**) employs what might be a familiar pattern of searching the metadata for strings, and then setting the HasMaxLength mapping for those strings.

With the class in place, you can now add the convention. The Add method takes a lambda but as you don't need to reference the lambda in the expression: You can simply use an underscore as the lambda variable.

Here's the updated ConfigureConventions method:

```
protected override void ConfigureConventions(
  ModelConfigurationBuilder configurationBuilder)
{
  configurationBuilder.Conventions
    .Remove(typeof(TableNameFromDbSetConvention));
  configurationBuilder.Conventions
    .Add(_ => new MaxStringLength200Convention());
  base.ConfigureConventions(configurationBuilder);
}
```

So Much More to Explore

It's never possible to include all of the changes in a single article and there are so many more improvements and new features in EF Core 7 that have caught my eye. The EF Core team has created a lot of great resources and documentation that I highly recommend checking out. In addition to the docs at https://docs.microsoft.com/en-us/ef/core/, there are a lot of great details to glean from their GitHub repository (https://github.com/dotnet/efcore). The bi-weekly updates (https://github.com/dotnet/efcore/issues/27185) have very detailed lists of changes and, of course, filtering on milestones is also very useful. But also be aware that you can move to EF Core 7 just for the performance benefits without having to worry much about breaking changes. The list of breaking changes is short and you can view it at https://docs.microsoft.com/en-us/ef/core/what-is-new/ef-core-7.0/breaking-changes.

Julie Lerman
CODE

Compare. Buy. Build.

Discover the best software components and tools

524	1,337	51,979	1,733	25
Commercial Products	Features Compared	Data Points Collected	Hours of Research	Years of Knowledge

www.componentsource.com/compare

Licensing Experts
available 24 hours Mon-Fri

Call **888.850.9911**
sales@componentsource.com

Specializing in

Perpetual Licenses
Timed Licenses
Subscriptions
Renewals
Upgrades
Old Versions
Lapsed Renewals
License Co-terms

ComponentSource®
www.componentsource.com

Upgrade Tooling for .NET 7

Last year, the .NET team introduced the .NET Upgrade Assistant tool to make migrating from .NET Framework to modern .NET targets easier. Since then, the team has been hard at work iterating on the upgrade tooling story to improve functionality and fill gaps. With .NET 7's release, there are now more tooling options for easing the transition from .NET Framework to .NET 7.

Mike Rousos
mikerou@microsoft.com

Mike Rousos is a Principal Software Engineer on the .NET Customer Engagement Team. A member of the .NET team since 2004, he has worked on a wide variety of feature areas and contributed content to the .NET team blog, .NET Conf sessions, Channel 9 videos, and .NET development e-books like ".NET Microservices: Architecture for Containerized .NET Applications." Outside of work, Mike is involved in his church and enjoys reading, writing, and games of all sorts. His primary hobby, though, is spending time with his four kids.

This article walks through those improvements—both the new capabilities in Upgrade Assistant and some new tooling focused specifically on web scenarios with the ASP.NET Incremental Migration Tooling and System.Web Adapters.

Although there are more tools to choose from now, each has a particular role in the upgrade process:

- Use **Upgrade Assistant** for analyzing the work required to upgrade, both with the existing analysis mode and the new binary analysis mode explained in more detail later in this article.
- Use **Upgrade Assistant** for upgrading class libraries or WPF, WinForms, console, or WCF apps in-place.
- Use **ASP.NET Incremental Migration Tooling** to upgrade ASP.NET applications by incrementally moving endpoints to a new ASP.NET Core project.
- Use **System.Web Adapters** in conjunction with ASP.NET Incremental Migration (to help the old and new projects interoperate) and to migrate class libraries with System.Web dependencies.

You can find the latest docs and news about Upgrade Assistant on GitHub at https://github.com/dotnet/upgrade-assistant.

You can find the latest docs and news about the ASP.NET Incremental Migration Tooling and System.Web Adapters on GitHub at https://github.com/dotnet/systemweb-adapters.

Upgrade Process Overview

Although there's new tooling to help, the overall process to migrate from .NET Framework to .NET 7 is still the same. The steps are shown in **Figure 1**.

Step 1: Preparation

The first step in upgrading a project from .NET Framework to .NET 7 is to get ready for the upgrade by understanding what work is ahead and possibly by making preliminary changes to the solution.

Specific actions during this step of migration include:

1. **Reviewing the project's dependencies**. This includes the .NET Framework APIs your app depends on, third-party packages, and other projects you own. While reviewing APIs or dependencies (like NuGet packages) that won't work on .NET 7, be on the lookout for any that will be difficult to replace. If missing APIs or NuGet packages are used in common or high priority code paths, this may be a red flag that an upgrade will require larger changes.
    a. Upgrade Assistant has a new binary analysis command in preview. Details on how to use this new command are in the Upgrade Assistant section of this article. Upgrade Assistant's binary analysis command will report on which .NET Framework APIs your project uses and whether those APIs are supported on .NET 7 or not.

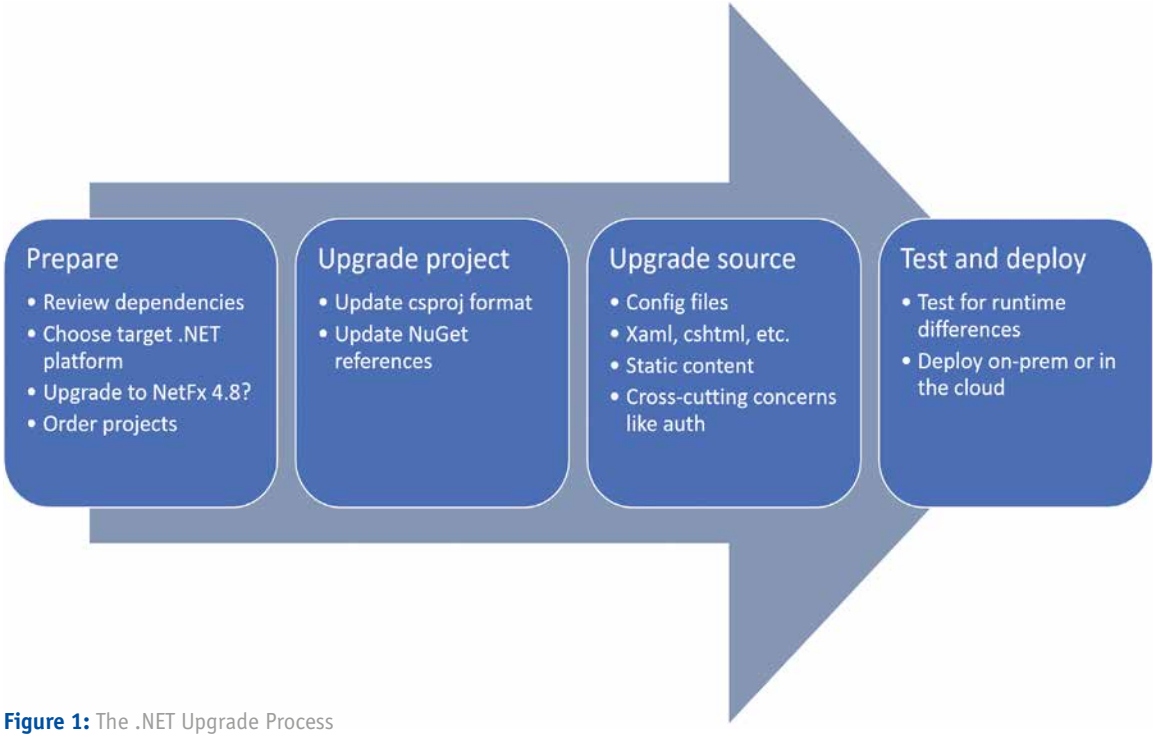

Figure 1: The .NET Upgrade Process

b. Upgrade Assistant's existing Analyze command is also useful while preparing for an upgrade because it highlights changes that Upgrade Assistant recommends, such as specific NuGet packages that will need to be updated or other changes that are required in the source or configuration of the project.

2. **Choosing a target framework**. When upgrading from .NET Framework, you might want to target .NET 7, .NET 6, or .NET Standard 2.0. If you'll be using Upgrade Assistant, it recommends an appropriate target based on your project and whether you prefer Long Term Support (LTS) or Current versions. More details on .NET support policy are available at https://dotnet.microsoft.com/platform/support/policy/dotnet-core. If you're using incremental migration tooling or updating the project file by hand, you need to choose the target framework yourself. Things to consider include:
   a. A .NET Standard 2.0 target allows libraries to be shared between .NET 6/7 callers and .NET Framework callers. Because of this valuable portability, it's recommended that class libraries should target .NET Standard 2.0 unless they have dependencies that require a more restrictive target framework. Executable projects (console apps, Windows desktop apps, and web apps like ASP.NET Core MVC or WebAPI projects) cannot target .NET Standard because it's only an API definition and not a runnable product.
   b. .NET 6 is the most recent LTS release of .NET. This means that it will be supported longer than .NET 7. LTS releases are supported for three years from their release date compared to 18 months for Current releases. If you need longer-term support for your project and don't need any of .NET 7's new features, .NET 6 is a great choice.
   c. .NET 7 is the most recent Current release of .NET. That means it has the latest features and best performance. If you intend to take advantage of .NET 7's improvements in your project, choose it as the target framework moniker.
3. **Upgrading to .NET Framework 4.8**. In some cases, it may make sense to retarget projects to .NET Framework 4.8 prior to upgrading to .NET 7. For very large projects or projects targeting old versions of .NET Framework (before .NET Framework 4.5, for example), this can be valuable because it allows you to address breaking changes between older and newer versions of .NET Framework separate from further breaking changes moving from .NET Framework to .NET 7. For small or medium-sized projects or those targeting more recent .NET Framework versions, upgrading to .NET Framework 4.8 isn't necessary because the number of breaking changes moving to .NET Framework 4.8 is likely to be small.
4. **Order projects**. Finally, if your solution contains many projects, consider in which order you will upgrade them to .NET 7 (or .NET Standard). It's usually best to upgrade lower-level dependencies ("leaf nodes" of the project graph) to .NET Standard first and then upgrade their callers. If you use Upgrade Assistant to upgrade a solution, it recommends an ordering based on the dependencies of projects in the solution and based on which project you ultimately want upgraded to .NET 7.

Step 2: Upgrading the Project File

Once you've prepared for the upgrade by understanding the project's dependencies and, possibly, by upgrading it to a newer version of .NET Framework, you're ready to begin modernizing the project's assets. The best place to start is the project file itself because .NET 7 uses new, simpler SDK-style project files.

Both Upgrade Assistant and the Incremental Migration Tooling can help with this step.

- If you're using Upgrade Assistant to upgrade a non-web project, it upgrades the project file in-place. In one of Upgrade Assistant's early steps, it updates the project's csproj or vbproj file so that it uses the new SDK-style format while being functionally equivalent to the old project file.
- If you're using Incremental Migration Tooling to upgrade an ASP.NET app, it creates a new ASP.NET Core project that you can gradually move endpoints into. The new project starts out empty with references and other items added as more of the project is migrated.

As part of upgrading the project file, you will also want to review your NuGet package references and make sure they're correct. The packages.config method of referencing NuGet packages that were probably used in the original project lists all NuGet packages needed, whereas, in an SDK-style project, only the packages your project uses directly need to be referenced (those packages' references do not need to be referenced). Again, tooling helps with this. Upgrade Assistant automatically removes unneeded package references and Incremental Migration Tooling won't even add unnecessary package references to the new project in the first place.

Of course, the version of NuGet references may also need to be updated in order to work with the new .NET target you'll be using (.NET 6 or .NET 7). Both Upgrade Assistant and Incremental Migration update package versions.

Finally, as part of upgrading the project file, the target framework moniker for the project should be set to the .NET platform that you're upgrading to. Upgrade Assistant automatically detects whether .NET 6, .NET 7, or .NET Standard makes the most sense for the project and updates it accordingly. With Incremental Migration, the UI for creating the new project allows the user to select .NET 6 or .NET 7.

Step 3: Upgrading Project Source

The third step of upgrading from .NET Framework to .NET 7 is upgrading the contents of the project: the source code, config files, and other assets. This is usually the bulk of the upgrade work.

Both Upgrade Assistant and Incremental Migration Tooling have the ability to automatically apply some fixes to source code while upgrading. For Upgrade Assistant, fixes are applied automatically across the whole project while, for Incremental Migration Tooling, fixes are made as the source code is copied over from the old project to the new one as needed to migrate specific vertical slices of the app.

Remember that no upgrade tooling can completely automate the changes needed to upgrade from .NET Framework to .NET 7. Especially for application models (like web apps) that have significant differences from .NET Framework, it will be necessary to make manual updates the tooling

wasn't able to automate. Although the upgrade tooling will address the "easy" changes, allowing you to focus on those that require more understanding of the project, the tooling won't address all necessary changes in any but the simplest of projects. In most cases, source code upgraded to .NET 7 won't immediately build and requires manual work to complete the upgrade before the project will successfully compile and run on the new target.

Step 4: Testing and Deploying

Once the project builds against the new .NET target, you're nearly done. But don't forget to test thoroughly before deploying the upgraded project. There are runtime behavioral differences between .NET Framework and .NET 7 that can cause apps to fail when run even if they build correctly. Make sure that any test suites are ported and passing in addition to exercising the upgraded project manually to confirm that everything's working as expected.

Once that's done, you're ready to deploy. With the upgraded app running on .NET 7, deployment may look a bit different—it's now an option to run on Linux or in Linux-based containers. You can also choose between framework-dependent deployment (which will use the .NET runtime on the user's machine) or self-contained deployment that includes the .NET runtime and libraries needed with the application. More details on .NET deployment models and options (including ahead-of-time compilation and single-file deployment) are available at https://docs.microsoft.com/dotnet/core/deploying.

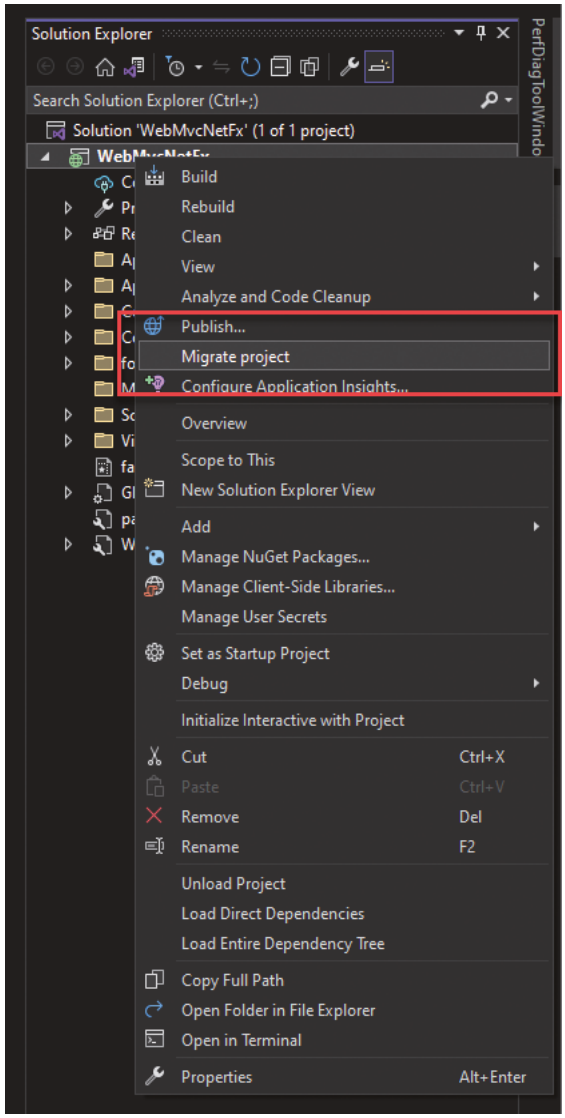

Figure 2: Migrating an ASP.NET app with Incremental Migration Tooling

ASP.NET Incremental Migration

One of the most exciting new options for upgrading is ASP.NET Incremental Migration Tooling. This tooling is a Visual Studio extension that allows developers to use a simple GUI interface to create new ASP.NET Core projects in parallel with existing ASP.NET apps. The tooling takes care of connecting the two apps (using a YARP proxy) so that requests coming into the ASP.NET Core app are first handled by that app and, if the ASP.NET Core app is unable to handle them, are then proxied to the original ASP.NET app. This applies the Strangler Fig pattern, allowing developers to gradually move functionality from their ASP.NET app into a new ASP.NET Core app one controller or action method at a time without changing their original ASP.NET app. From an end user's perspective, the experience of using the app will be unchanged. Behind the scenes, the app will be slowly moving to ASP.NET Core as more and more of its components are upgraded.

Although Upgrade Assistant is still the recommended tool for upgrading class libraries and non-web apps, ASP.NET Incremental Migration Tooling is recommended for ASP.NET scenarios.

Getting Started with Incremental Migration

To get started, install the Visual Studio extension from https://marketplace.visualstudio.com/items?itemName=WebToolsTeam.aspnetprojectmigrations.

Once the extension is installed, you can choose "Migrate project" in the context menu when right-clicking on an ASP.NET app, as shown in **Figure 2**.

A UI appears allowing the user to select the target framework (.NET 6 or .NET 7) and template to use (MVC or WebAPI) for the new project and a new ASP.NET Core project is created, configured with a YARP proxy to automatically forward any requests it can't handle to the original ASP.NET app. Project launch configuration is also updated so that pushing **F5** in Visual Studio begins debugging both the old and new web projects.

Incrementally Migrating Project Components

Once the new project is created, you can choose to migrate classes, controllers, or views. Simply right-click on the component to migrate (either in Visual Studio's editor or in the solution explorer) and choose the **Migrate** menu option. The UI will display a tree view of the selected component and its dependencies, as shown in **Figure 3**. Dependencies could include other classes, views, NuGet packages, or project-to-project references. From the migration UI, you can choose which of the dependencies to migrate (by default, all direct dependencies are selected).

Clicking the **Migrate selection** button copies the selected item along with all selected dependencies into the new

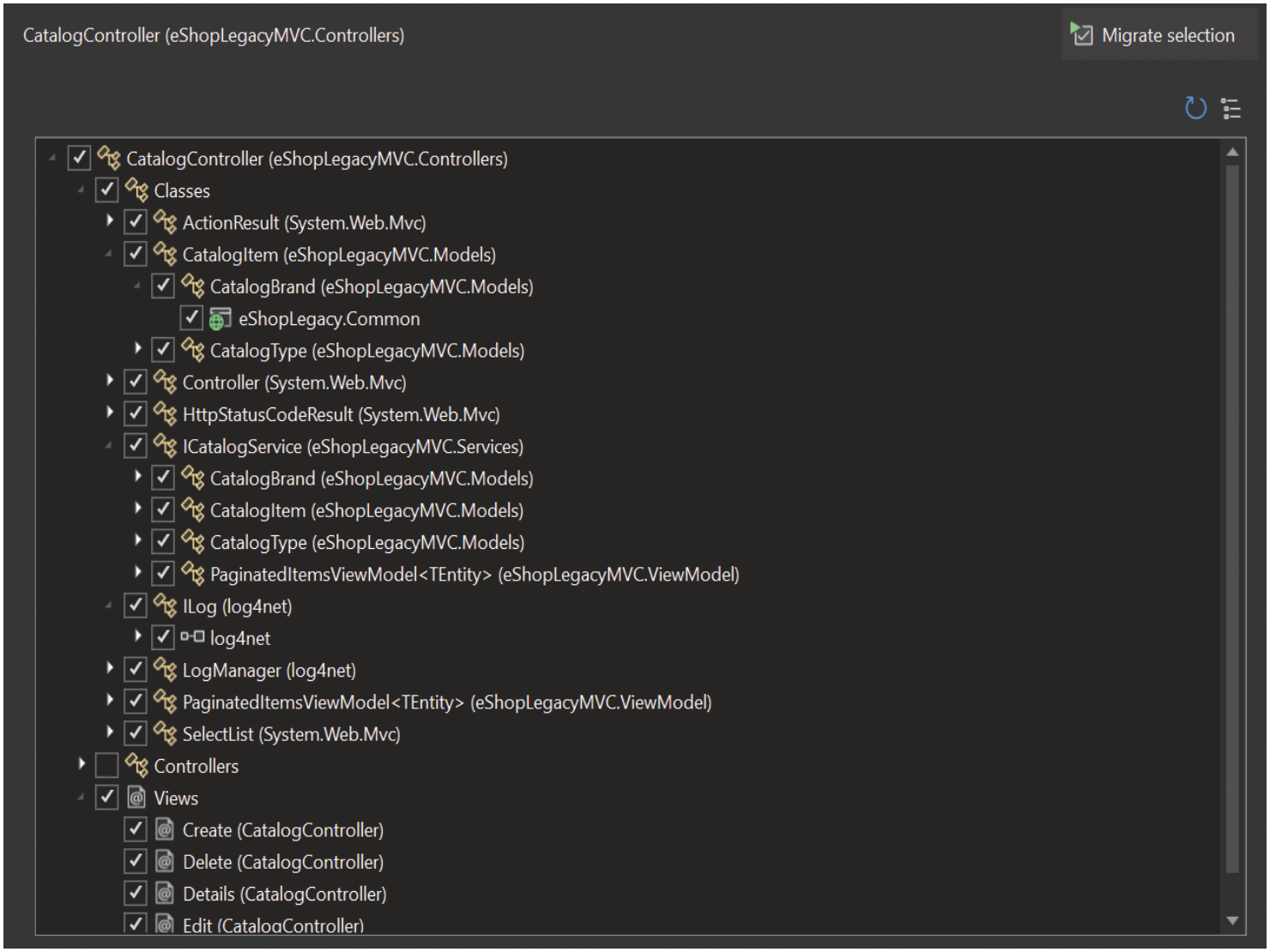

Figure 3: The incremental migration UI

project. While copying source files, the tooling updates source code with some transformations to get it closer to what's needed to run on ASP.NET Core (fixing up some common ASP.NET namespace changes, for example, or replacing types with near equivalents in ASP.NET Core). While copying NuGet dependencies, the tooling updates the packages to more recent versions that work with the newer .NET version in the new project. Of course, as mentioned previously, there will be additional manual fixups needed in migrated source files before they will build and work in the new project. But by using the Incremental Migration Tooling, you're able to tackle that challenge one small part of the app at a time.

Except for a couple properties in the csproj file to enable the migration process, the original ASP.NET app isn't changed, so it will keep working the same as always.

System.Web Adapters

The ASP.NET team has created an open-source project called System.Web Adapters that pairs with the Incremental Migration Tooling extension to make it easier to transition code from ASP.NET to ASP.NET Core. The project can be found at https://github.com/dotnet/systemweb-adapters. In that GitHub repository, you can learn more about the project, file issues, or even submit pull requests to contribute.

The System.Web Adapters project has two parts:

- A core package of adapters.
- Extensions that make it easier to work with an ASP.NET and ASP.NET Core project side-by-side, as in the case of incremental migration.

Adapters Package

The lowest-level component of the System.Web adapters is the Microsoft.AspNetCore.SystemWebAdapters package. This package targets .NET Standard and contains adapters for common System.Web APIs, such as HttpContext.Current and APIs on HttpRequest, HttpResponse, and many other types. When used by a .NET Framework caller, the adapters type-forwards references to these types to the actual implementations in System.Web. When used from a .NET 6 or .NET 7 caller, the package shims the calls to equivalent ASP.NET Core calls.

Using Microsoft.AspNetCore.SystemWebAdapters, developers can use code that depends on these common System.Web APIs while targeting .NET Standard 2.0 and have it work on either ASP.NET or ASP.NET Core. This has benefits in two scenarios:

- By using the adapters package, class libraries with System.Web dependencies can be upgraded to target .NET Standard 2.0 with minimal code changes. This makes upgrading a large web solution much simpler. By targeting class libraries to .NET standard, they can be used by upstream callers that are still targeting .NET Framework and by callers that are upgraded to ASP.NET Core and .NET 7. This is especially useful in incremental migration scenarios because it allows the original ASP.NET app and the new ASP.NET Core app in the migration scenario to share class libraries.
- Using the System.Web adapters can also help minimize code changes when initially migrating code to an ASP.NET Core project during incremental migration. Although the APIs supported by the adapters package are prioritized around what's most likely to be used in libraries, there's enough overlap with APIs used in MVC scenarios that they provide some value getting migrated classes working on ASP.NET Core, as well. It will be necessary to update instances of System.Web APIs being used from the migrated ASP.NET Core app eventually, but the System.Web adapters can be useful in getting things working initially.

Services Packages

In addition to the adapters package, the System.Web Adapters project includes new functionality that helps ASP.NET and ASP.NET Core apps work together better in incremental upgrade scenarios. This new functionality ships in two packages: one package for use in the ASP.NET app (Microsoft.AspNetCore.SystemWebAdapters.FrameworkServices) and one for use in the ASP.NET Core app (Microsoft.AspNetCore.SystemWebAdapters.CoreServices).

The services are enabled in the ASP.NET Core app by calling AddSystemWebAdapters and UseSystemWebAdapters in their main method, as shown in this code snippet. This makes SystemWebAdapters services available in the app's dependency injection container and registers SystemWebAdapters middleware.

```
using Microsoft.AspNetCore.SystemWebAdapters;
var builder = WebApplication.CreateBuilder(args);

// Add System.Web adapter services to the container
builder.Services.AddSystemWebAdapters();

builder.Services.AddReverseProxy()
 .LoadFromConfig(builder.Configuration
 .GetSection("ReverseProxy"));

builder.Services.AddControllersWithViews();

var app = builder.Build();

// Additional middleware goes here

// Add System.Web adapter middleware
app.UseSystemWebAdapters();

app.MapDefaultControllerRoute();
app.MapReverseProxy();
```

Once the System.Web adapter middleware is registered, endpoints in the ASP.NET Core app can opt into behaviors to emulate ASP.NET behaviors to match how they functioned in the old app. These optional ASP.NET-emulated behaviors include setting the value of Thread.CurrentPrincipal, forcing a request to be served by a single thread, pre-buffering requests, or buffering responses. Controllers and action methods can opt into these behaviors using attributes or they can be enabled by default using extension methods in the app's startup path. More information is available in the System.Web adapters documentation at https://github.com/dotnet/systemweb-adapters/blob/main/docs/usage_guidance.md.

On the ASP.NET side of things, System.Web adapter services are enabled by registering the SystemWebAdapters module in the app's web.config (this happens automatically when the FrameworkServices package is installed) and by calling Application.AddSystemWebAdapters() in the app's Application_Start method.

It's recommended to enable the System.Web adapter services' proxy support, which will cause values like the request URL to match the app's public entry point even though the requests have been proxied to the ASP.NET app. That support is enabled by calling AddProxySupport in the app's Application_Start method:

```
protected void Application_Start()
{
    SystemWebAdapterConfiguration
        .AddSystemWebAdapters(this)
        .AddProxySupport(options =>
            options.UseForwardedHeaders = true);
}
```

Full details on setting up the ASP.NET app to take advantage of new functionality in the System.Web adapters is available in documentation at https://github.com/dotnet/system-web-adapters/blob/main/docs/framework.md.

Remote App Authentication and Session

The System.Web adapters services packages also enable sharing authentication and session state between the ASP.NET and ASP.NET Core apps. This means that if a user signs in using an endpoint in the original ASP.NET app or sets session items on such an endpoint, the same identity and session items will be available when the user navigates to an endpoint served by the ASP.NET Core app. Similarly, session items written from the ASP.NET Core app will be available to the ASP.NET app.

Although session and authentication work differently in ASP.NET and ASP.NET Core, the System.Web adapters enable these features to work in incremental migration scenarios by allowing the ASP.NET Core app to make requests to the ASP.NET app behind the scenes to determine user identity and session items. Any changes to session state are similarly written to the ASP.NET app. So, the ASP.NET app serves as the source of truth for authentication and session information.

These features are enabled by calling AddRemoteAppClient on the ASP.NET Core app's ISystemWebAdapterBuilder (re-

turned by adding System.Web services to the app's services) and by calling AddRemoteAppServer on the ASP.NET app's ISystemWebAdapterBuilder (returned by the call to Application.AddSystemWebAdapters). These remote app calls allow the user to configure the connection between the ASP.NET Core and ASP.NET apps—specifying a security key so that the ASP.NET app knows requests for session or auth information are legitimate and specifying the base URL of the ASP.NET app for the ASP.NET Core app to communicate with.

To enable sharing session information between the two apps, the ASP.NET app must call AddSession and configure a serializer that will be used for writing and reading sessions state to/from the ASP.NET Core app. Serialization is typically done with the System.Web Adapters' JsonSessionSerializer, but users can implement their own serializers if needed. As part of configuring the serializer, the developer must register session item keys that will be used and specify the types of the corresponding session items. This registration with typing information is necessary to securely deserialize session items.

Similarly, to enable shared authentication to work, the ASP.NET app must add a call to AddAuthentication in order to enable endpoints that will deliver user identity to the ASP.NET Core app.

Altogether, with both remote authentication and remote session enabled, the code in global.asax.cs ends up looking like this:

```
SystemWebAdapterConfiguration
 .AddSystemWebAdapters(this)
    .AddProxySupport(options =>
        options.UseForwardedHeaders = true)
    .AddRemoteAppServer(remote => remote
        .Configure(options =>
            options.ApiKey = "MySecureKey")
        .AddAuthentication()
        .AddSession())
    .AddJsonSessionSerializer(o =>
    {
        o.KnownKeys
         .Add("myInt", typeof(int));
        o.KnownKeys
         .Add("mObject", typeof(DemoModel));
    });
```

On the ASP.NET Core side, sharing session state and shared authentication are enabled in the same way. AddSession and AddAuthentication are called while configuring the ISystemWebAdapterRemoteClientAppBuilder, and a serializer is registered with the ISystemWebAdapterBuilder along with session items that are expected.

```
builder.Services.AddSystemWebAdapters()
    .AddRemoteAppClient(remote => remote
        .Configure(ConfigureRemoteAppOptions)
        .AddAuthentication(true)
        .AddSession())
    .AddJsonSessionSerializer(o =>
    {
        o.KnownKeys
         .Add("myInt", typeof(int));
        o.KnownKeys
         .Add("mObject", typeof(DemoModel));
    });
```

Notice that on the ASP.NET Core side, the call to AddAuthentication takes a Boolean parameter. This parameter indicates whether remote app authentication should be the default authentication scheme for the app or not. If remote app authentication is the default scheme, user identity is retrieved from the ASP.NET app for every request to the ASP.NET Core app. If it's not the default, identity will only be retrieved from the ASP.NET app for requests to endpoints decorated with an authentication attribute specifically requesting authentication with this scheme:

```
[Authorize(AuthenticationSchemes = "Remote")]
```

Using remote app authentication as the default scheme avoids needing to add these attributes and makes the user's identity available everywhere but comes with the downside of having an HTTP call to the ASP.NET app to retrieve identity for every request the ASP.NET Core app serves. Not using remote app authentication as the default scheme allows the remote identity to be used more tactically only in cases where it's needed.

Similarly, because sharing session state requires HTTP requests to the ASP.NET app, it's not enabled by default. Instead, controllers or action methods that need to share session state with the ASP.NET app should be annotated with the [Session] attribute.

Because the System.Web adapters libraries are still in preview at the time this article is being written, some small changes may be made to the API (method names, etc.) prior to the article being published. Full details and all the latest details on configuring and using shared session and shared authentication in incremental migration scenarios are available in the System.Web adapter documentation at https://github.com/dotnet/systemweb-adapters/tree/main/docs.

.NET Upgrade Assistant

Although Incremental Migration Tooling and the System.Web Adapters offer migration assistance for ASP.NET scenarios, the .NET Upgrade Assistant is still the recommended tool for upgrading class libraries, console apps, Xamarin apps, and Windows Desktop apps in-place. Since Upgrade Assistant's release last year, a number of new features have been added, most notably the ability to scan binaries and compare the .NET Framework APIs used against the surface area of .NET 7 and the ability to upgrade a new type of project: WCF server projects.

Binary Analysis

Previously, when upgrading from .NET Framework to a more modern .NET target, users would use the .NET Portability Analyzer to gauge how many of the .NET Framework APIs they'd used were unavailable on the .NET platform they were migrating to. The .NET Portability Analyzer is in the process of being deprecated and this functionality has been added to Upgrade Assistant. Now you can use the same tool to analyze .NET API usage and upgrade your projects.

Upgrade Assistant's binary analysis feature differs from its existing analyze command in that the analyze command works by running the same analysis as the upgrade com-

mand and reports on items to be changed (source code to update, package versions to update, etc.) and the binary analysis feature instead looks at API usage compared to a catalog of which APIs are available on which .NET platforms.

Binary analysis is still in development, so until it's fully released, you need to set an environment variable to enable it in Upgrade Assistant.

```
Set UA_FEATURES=ANALYZE_BINARIES
```

Once the UA_FEATURES variable is set to ANALYZE_BINARIES, Upgrade Assistant has a new command available to it: analyzebinaries. There are several useful options that can be passed to the analyzebinaries command. Among them are:

- **-f <format>**: This option allows specifying the output format for the report that's generated. Options are HTML or Sarif. Sarif is a well-known JSON format used to store diagnostic information. Visual Studio can display data from Sarif files in its error list and extensions are available to visualize Sarif in Visual Studio Code.
- **-t <Current | LTS | Preview>**: This option allows specifying which .NET target you intend to upgrade to so APIs used in the app can be compared to that target. Specifying **Current** compares against .NET 7 and **LTS** compares against .NET 6. **Preview** also compares against .NET 7 for the time being but soon it will compare against .NET 8 previews.
- **-p <Linux | Windows>**: The platform option allows specifying whether the upgrade project will run on Windows or Linux so that the API surface area can be adjusted for the desired environment.

Unlike other Upgrade Assistant commands, the analyzebinaries command runs on compiled binaries rather than project or solution files. So, for input, you'll need to specify a dll or a folder containing one or more .NET assemblies. This means that you can run the analyzebinaries command even against dependencies you don't own. This can be useful if you need to gauge whether a binary dependency is likely to work on .NET 7, but be careful about running the command on NuGet binaries. NuGet packages often have different binaries for different targets, so seeing whether a specific assembly from NuGet uses unsupported APIs or not is typically not useful. There may be other assemblies available for the package for different targets or newer versions of the package available with more supported .NET platforms. NuGet dependencies are better analyzed with Upgrade Assistant's analyze command.

To use analyzebinaries, it's recommended to copy the assemblies you own source code for as well as any binary dependencies that cannot be upgraded to newer .NET 7-native versions through other means (not NuGet output or .NET Framework binaries) into a temporary folder. Then, run the analyzebinaries command using options appropriate for your scenario. For example:

```
upgrade-assistant analyzebinaries -t LTS
  -f html .\binaryoutput
```

Upgrade Assistant reads the assemblies and generates a report showing any APIs that aren't supported on the target

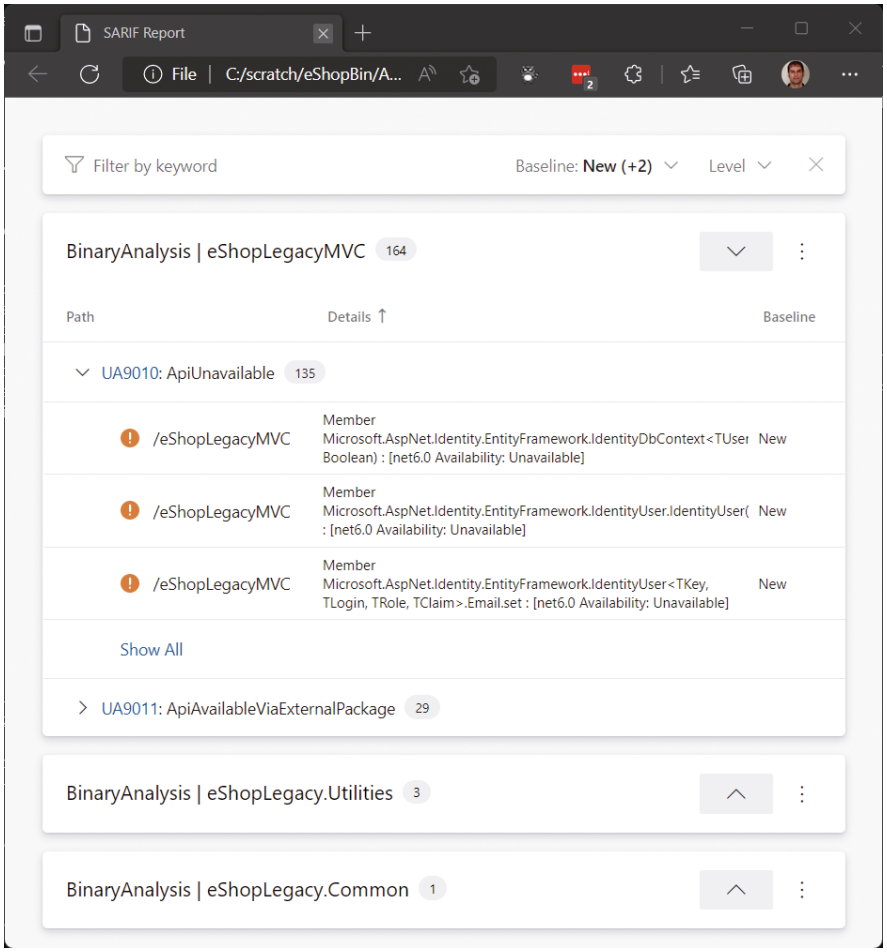

Figure 4: Upgrade Assistant analyzebinaries output

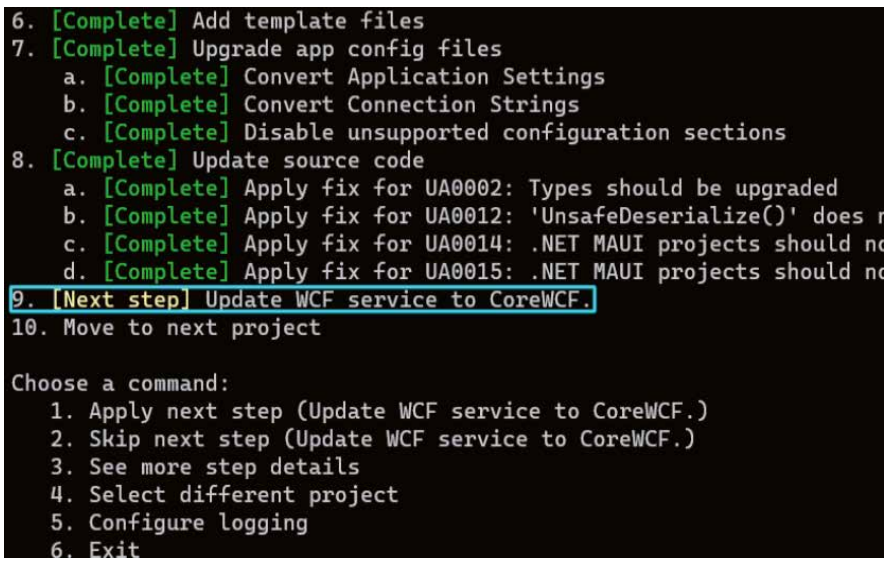

Figure 5: Upgrade Assistant's new CoreWCF Upgrade step

version of .NET, as well as APIs that are available but require additional NuGet references, as shown in **Figure 4**.

CoreWCF

In April of 2022, version 1.0 of the CoreWCF community project was released. This project makes many server-side WCF

APIs available for .NET 6 and .NET 7. At the same time as the 1.0 release, Microsoft announced that it would support CoreWCF usage in production scenarios and that it was a recommended path forward for customers with server-side WCF dependencies who wanted to upgrade to .NET 6 or .NET 7, but who could not easily remove the dependency on WCF. More details on CoreWCF and Microsoft support for the project is available at https://devblogs.microsoft.com/dotnet/corewcf-v1-released. To learn more about CoreWCF, be sure to check out Sam Spencer's article on CoreWCF elsewhere in this issue of CODE Magazine or go to https://github.com/corewcf/corewcf.

To help customers looking to use CoreWCF as a means of upgrading to .NET 7, Upgrade Assistant now supports upgrading self-hosted WCF scenarios to CoreWCF. No new commands are needed—just use a recent version of Upgrade Assistant and a new upgrade step (shown in **Figure 5**) will look for self-hosted WCF services and, if found, help to upgrade them to CoreWCF.

Most of the changes moving to CoreWCF are related to setting up the ServiceHost. Some WCF settings that were previously specified in configuration are now set up programmatically. Upgrade Assistant's new CoreWCF upgrade step will help to move the parts of the configuration that need to be changed into code while leaving the rest of the config file for CoreWCF to use. CoreWCF runs on top of ASP.NET Core, so the new code paths will look familiar to anyone who's configured an ASP.NET Core web host before. Because CoreWCF tries to be as compatible with WCF as possible, existing service contracts and implementations should work unchanged.

Upgrade Tooling Roadmap

This article has introduced a lot of new upgrade tooling options: new Upgrade Assistant features, including a binary analysis command, new Incremental Migration tooling for ASP.NET scenarios through a Visual Studio extension, and adapter libraries to make upgrading web scenarios easier.

Going forward, the teams working on these tools will be focused both on adding more functionality to the tools and aligning them with each other. It's great that there's now more tooling available for upgrading to .NET 7 but, in the future, these tools will begin to consolidate and support each other better. There will continue to be a command-line experience through Upgrade Assistant and a Visual Studio experience through the Incremental Migration extension, but planning is underway to share the internal logic of these two tools so that they will give similar experiences (through different user interfaces) and allow users to write extensions that will work with either toolset.

The upgrade tooling for .NET is still young and developing rapidly. As you work with the tools, please get involved with their development efforts. By sending feedback, filing issues, and creating pull requests, you can help to shape the future of .NET upgrade tooling. You can provide feedback and contribute to Upgrade Assistant at https://github.com/dotnet/upgrade-assistant.

Similarly, you can connect with the System.Web adapters project and create issues or pull requests at https://github.com/dotnet/systemweb-adapters. Although the Incremental Migration Visual Studio extension isn't currently open source, you can still provide feedback on it through the System.Web adapters GitHub repo.

Wrap Up

.NET 7 offers a host of new features and performance improvements. Targeting .NET Framework, while supported, keeps projects from using the latest runtime features, language features, performance improvements, and community libraries. Upgrading from .NET Framework to .NET 7 can be a challenge, especially for some app models like ASP.NET that have seen major changes, but tools like Upgrade Assistant and Incremental Migration Tooling for ASP.NET can help.

No tooling can completely automate the transition from .NET Framework to .NET 7, but Upgrade Assistant can help analyze the work that needs to be done and assist in making many of the changes needed to move libraries, console apps, and Windows Desktop or Xamarin apps to .NET 7. Meanwhile, the new Incremental Migration Tooling for ASP.NET provides a visual interface for gradually moving functionality in an ASP.NET app to ASP.NET Core on .NET 7 one component at a time. The System.Web adapters libraries enable sharing web-based code between .NET Framework and .NET 7 and enable important interoperability scenarios between ASP.NET and ASP.NET Core apps in incremental upgrade scenarios.

Please try the tools out and engage with us on GitHub to share feedback or any issues you run into!

Mike Rousos
CODE

Using CoreWCF to Move WCF Services to .NET Core

CoreWCF is a port of the functionality of the WCF Server libraries to the .NET [Core] platform. In this article, I'll talk about the objectives of the CoreWCF project and how it can be used to more easily modernize applications to .NET. CoreWCF is an open source project and can be found at https://github.com/CoreWCF/CoreWCF. If you're going to work with the code examples shown in this article, you'll need the following installed on your system:

- Visual Studio 2022
- .NET 6.0 or greater

Introduction to CoreWCF

Windows Communication Foundation (WCF) is an RPC mechanism that enables rich client/server communication between processes. The advantages WCF has over other RPC mechanisms such as WebAPI, REST, or gRPC are that the contracts and data types are defined using .NET classes and interfaces, it supports a high fidelity serialization of data types, has efficient binary serialization, and includes built-in security and rich communication paradigms such as callbacks and streaming. WCF supports WS-* standards-based communication using SOAP and proprietary bindings for more efficient communication between .NET on the client and server. For these reasons, WCF has been a very commonly used RPC stack in .NET Framework apps.

When the .NET Core project started in 2016, the goal was to take the best of .NET, but also to use it as an opportunity to pare down some of the bloat that had accrued with .NET Framework. The software industry has largely moved on from SOAP to restful Web API and gRPC. WCF has a very large and complex surface area that wasn't seen as fitting well with the "lean and mean" goals of .NET Core.

Microsoft had been recommending WCF as **the** RPC mechanism to use in .NET for some time, and customers have a large investment in WCF services. We realized that the lack of WCF capability was a major blocker to being able to modernize the apps to .NET Core.

The.NET team didn't initially intend to include WCF in the box, but there were a couple of devoted WCF developers who wanted to update WCF and bring it to modern .NET. So we funded the effort to seed a community project, which became CoreWCF. This gathered the attention of other developers in the community who needed WCF support, including the .NET team at AWS, who contributed toward the support of WS-* security protocols.

CoreWCF isn't a straight port of WCF to .NET—there were a couple of architectural changes that were needed as part of the port:

- Using ASP.NET Core as the service host, push pipeline, and the middleware pattern for extensibility.
- Removing the obsolete Asynchronous Programming Model (APM) programming pattern as it made the codebase incredibly hard to work with, which isn't desirable for a community project wanting to encourage external contributions.
- Removing platform-specific and IO code. Refactoring apps into microservices and Linux-based containers is a common requirement, and so CoreWCF needs to be able to run anywhere that .NET core can be run.

After three years of development, the feature set was deemed functional enough for mainstream usage and so the 1.0 of the project was released in April 2022. CoreWCF, like the .NET platfom it's built on, is cross-platform, so can be used as easily on Linux, in a Kubernetes container, or on MacOS as it can be on a Windows server.

When talking with WCF customers, particularly larger enterprise customers, a large concern with adopting a community project is what level of support would be available. Microsoft came to an internal agreement so that Microsoft Support is available for CoreWCF when used in production. If you've ported to CoreWCF and you find operational bugs (not missing features), Product Support coordinates a fix.

Using CoreWCF in Practice

There are a couple of parts to creating a WCF service:

- **Defining the service contracts:** C# interfaces with attributes
- **Implementing the contracts:** C# classes implementing the service interface
- **Creating data contracts for data that will be serialized as part of the service contracts:** POCO classes with optional attribution
- **Exposing the services:** Use WCF Bindings

The key advantage of CoreWCF is that in most cases, it doesn't involve any changes to the service definition, implementations, or data contracts. The same code you used for these items in WCF is supported in CoreWCF. The differences come with the ceremony as to how the services are hosted and the bindings that are supported.

What Does a CoreWCF Service Look Like?

The best way to understand CoreWCF is probably to create a new service from scratch. The easiest way to do that is using the new project template (a community contribution), which can be installed using the dotnet tool.

```
dotnet new --install CoreWCF.Templates
```

Sam Spencer
sam.spencer@microsoft.com

Sam is a Program Manager on the .NET team. He has worked on various developer technologies at Microsoft including YARP, CoreWCF, .NET, WinUI, WinJS, LightSwitch, Visual Web Developer and Visual Basic.

That installs a new template that can be used from the command line or via Visual Studio. Once installed, in the Visual Studio New Project dialog, type "corewcf" in the search box to find the template, as seen in **Figure 1**

Go through the dialogs to create a project for .NET 6.0 or 7.0. The template has support for the other versions, but with .NET 6+, you get the simplified ASP.NET starter code and top-level statements.

Opening IService.cs, you can see that the definition of the service is identical to how it's defined with WCF. You have an interface decorated with the ServiceContract attribute, and each of the methods to be exposed is decorated with **OperationContract.** The only difference is that the attributes are now from the **CoreWCF** namespace rather than **System.ServiceModel** namespaces.

```
using CoreWCF;
using System;
using System.Runtime.Serialization;

namespace CoreWCFService1
{
  [ServiceContract]
  public interface IService
  {
    [OperationContract]
    string GetData(int value);

    [OperationContract]
    CompositeType GetObject(CompositeType obj);
  }
```

The service implementation is in the form of a class that implements the service definition. There are no changes from how this is defined in WCF.

```
public class Service : IService
{
  public string GetData(int value)
  {
    return $"You entered: {value}";
  }

  public CompositeType GetObject(CompositeType obj)
  {
    if (obj == null)
    {
      throw new ArgumentNullException("composite");
    }
    if (obj.BoolValue)
    {
      obj.StringValue += "Suffix";
    }
    return obj;
  }
}
```

The last part of the file uses the **DataContract** and **DataMember** attributes to mark a class for serialization so that it can be used as part of service definition.

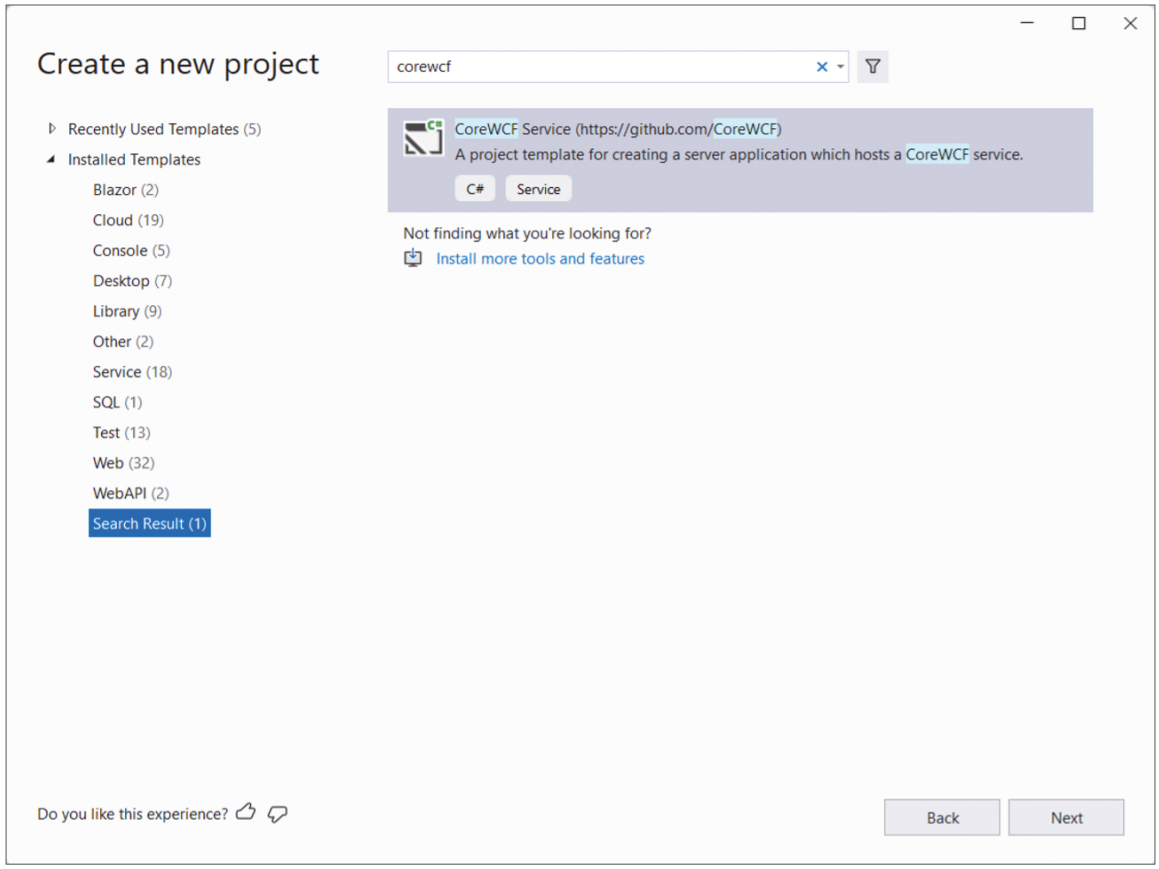

Figure 1: New project dialog showing CoreWCF template

```
[DataContract]
public class CompositeType
{
  bool boolValue = true;
  string stringValue = "Hello ";

  [DataMember]
  public bool BoolValue
  {
    get { return boolValue; }
    set { boolValue = value; }
  }

  [DataMember]
  public string StringValue
  {
    get { return stringValue; }
    set { stringValue = value; }
  }
}
```

Program.cs sets up the host and exposes the services via bindings. If you're familiar with ASP.NET Core, this will look somewhat familiar because CoreWCF uses the same host and builder model, but with extra middleware that implements WCF services and provides WSDL generation through metadata.

```
var builder = WebApplication.CreateBuilder();

builder.Services.AddServiceModelServices();
builder.Services.AddServiceModelMetadata();
builder.Services.AddSingleton<IServiceBehavior,
  UseRequestHeadersForMetadataAddressBehavior>();

var app = builder.Build();
```

The next block adds a service from our class Service and exposes its definition from IService with a BasicHttpBinding and sets the URL as /Service.svc.

```
app.UseServiceModel(bld =>
{
  bld.AddService<Service>();
  bld.AddServiceEndpoint<Service, IService>(
    new BasicHttpBinding(BasicHttpSecurityMode.Transport),
    "/Service.svc");
  var mb = app.Services.GetRequiredService<
    ServiceMetadataBehavior>();
  mb.HttpsGetEnabled = true;
});

app.Run();
```

By default, the ASP.NET hosting runs the process using the Kestrel web server, but it can also be hosted in IIS Express, depending on how the process is launched. That can be changed in Visual Studio using the launchSettings.json file.

Bindings in CoreWCF

Bindings in WCF define how the services will be exposed over the wire. For example, the BasicHttpBinding used above exposes a service over HTTP using the SOAP protocol. The NetTcp Binding uses a custom binary XML serialization to be more efficient over the wire than plain text XML. WCF supported a large range of bindings and only a subset are currently included in CoreWCF, listed in **Table 1**.

The WSFederationHttpBinding and related security functionality have been developed by AWS. The WebHttpBinding was implemented by a community member who had a large investment in services using the Binding, and they determined that it would be cheaper for them to port the binding than converting all the services to MVC or WebAPI.

The main bindings not yet available that we hear developers asking for are:

- NamedPipeBinding
- MSMQBinding

Both are currently under development. In line with the architectural changes to be cross-platform, the Message Queue binding will be changing to have a provider model for the actual queue implementation so that it isn't tied to the Windows MSMQ Server and can be used with queue implementations for cloud providers such as AWS and Azure.

Configuration in CoreWCF

WCF had strong support for configuration. The bindings that control the way that services are exposed could be specified in code, but more commonly was done through configuration. The idea was that you can create the service, and then Ops/IT can configure the binding and its properties without needing to rebuild the application. This resulted in long and complex configuration files, which proved to be a source of errors and complexity. Later versions of .NET Framework introduced a simplified form of configuration to reduce the verbosity and provide common defaults.

The WCF configuration model was based around the .NET Framework xml/web.config model for configuration that isn't fully supported on Core, and which now uses a JSON-based configuration format. The lack of configuration support was a problem for a couple of users, and their combined work has resulted in the package CoreWCF.Configuration. Currently, not all options are supported, but this should help when migrating WCF applications.

Here are the basic steps to use the configuration library:

Binding	Description
BasicHttpBinding	Uses SOAP to encode messages over HTTP and is compatible with ASMX-based services and clients using SOAP 1.1
NetHttpBinding	Uses a binary format over HTTP for smaller message encoding. If the service contract is duplex, it will use web sockets to create a bi-directional channel
WSFederationHttpBinding	Supports WS-Federation to enable federated security
WSHttpBinding	SOAP 1.2-based, only uses open standard WS-* protocols for interoperability with other frameworks
NetTcpBinding	Uses TCP for delivery with a binary encoding for the message format, with support for Windows Security
WebHttpBinding	Supports creating Restful web services using WCF contract definitions

Table 1: Bindings supported by CoreWCF

1. Copy the system.serviceModel section into a separate file, such as wcf.config, which should be placed in the application folder. Here's an example file:

```
<system.serviceModel>
  <bindings>
    <netTcpBinding>
      <binding name="netTcpBindingConfig"
        receiveTimeout="00:10:00" />
    </netTcpBinding>
  </bindings>
  <services>
    <service name="Services.ISomeContact">
      <endpoint address="net.tcp://localhost:8750/Service"
        binding="netTcpBinding"
        bindingConfiguration="netTcpBindingConfig"
        contract="ISomeContact" />
    </service>
  </services>
</system.serviceModel>
```

1. In your CoreWCF project, add the CoreWCF.ConfigurationManager NuGet package.
2. Tell CoreWCF to load configuration from your XML file:

```
builder.Services.AddServiceModelServices();
builder.Services.AddServiceModelConfigurationManagerFile(
  "wcf.config");
builder.Services.AddServiceModelMetadata();
builder.Services.AddSingleton<IServiceBehavior,
  UseRequestHeadersForMetadataAddressBehavior>();
```

If NetTcpBinding is used, the TCP port numbers of endpoints using this binding must be additionally specified when configuring the builder.

That's all there should be to it! If there are any configuration problems in the wcf.config file, they will be reported when the application starts.

The set of supported configuration elements are:

```
<bindings>
  <basicHttpBinding maxBufferSize="" transferMode=""
textEncoding="" />
  <netHttpBinding maxBufferSize="" transferMode=""
textEncoding=""
    messageEncoding="" />
  <netTcpBinding maxBufferSize="" maxBufferPoolSize=""
maxConnections=""
    transferMode="" hostNameComparisonMode="" />
  <wSHttpBinding maxBufferPoolSize="" />
</bindings>
```

All the bindings above also support these properties: name="", securityMode="", maxReceivedMessageSize="", receiveTimeout="", closeTimeout="", openTimeout="", sendTimeout="", and xmlReaderQuotas="".

```
<services>
  <endpoint name="" address=""
binding=""  bindingConfiguration=""
    contract="" />
</services>
```

The goal is to add more configuration support over time.

Client-Side WCF Support

.NET Core includes client-side support for calling WCF services with the System.ServiceModel.* NuGet packages. Client wrappers can be generated for services using the ConnectedServices and Service Reference UI in Visual Studio. From the command line, the **dotnet-svcutil** tool can be used to generate the same wrapper code.

```
dotnet tool install --global dotnet-svcutil
dotnet-svcutil --roll-forward LatestMajor
https://localhost:7173/Service.svc?wsdl
```

Using either of those tools requires that the ServiceModelMetadata feature is added as part of the app initialization. This is included in the project template and the example code further up. You may need to add the **?wsdl** query string to the URL to get the metadata.

The following console app code uses the service wrapper to call the service above:

```
using ServiceReference1;
// Instantiate the Service wrapper specifying the
// binding and optionally the Endpoint URL.
var client = new ServiceClient(
  ServiceClient.EndpointConfiguration.BasicHttpBinding_
IService,
  "https://localhost:7173/Service.svc");

var simpleResult = await client.GetDataAsync(10);
Console.WriteLine(simpleResult);

var msg = new CompositeType(){ StringValue = "A ",
  BoolValue=true};
var msgResult = await client.GetObjectAsync(msg);
Console.WriteLine(msgResult.StringValue);
```

How to Modernize a WCF Service to .NET Core

Assuming you have an existing WCF application, there are a couple of approaches that can be used to modernize the services using CoreWCF:

- In-place replacement while running on .NET Framework
- Copy individual services over
- In-place upgrade to .NET Core

Read on for some discussion about these options.

In-Place Replacement While Running on .NET Framework

The 1.x releases of CoreWCF have been designed so that it can be used on .NET Framework with ASP.NET Core 2.1. The services can be updated individually to be exposed using CoreWCF before changing the runtime and project file format. This can be an effective way to ensure that CoreWCF will meet your needs before diving into a larger migration.

The code shown in this article for .NET 6 uses top-level statements and the simplified ASP.NET initialization code. Samples for .NET Framework and earlier versions of .NET can be found at https://github.com/corewcf/samples.

Copy Individual Services Over

Some developers prefer to take the approach of creating a new project and then copying code across from their existing projects. You should find that only a few changes are required to the service classes and interfaces to be able to use them with CoreWCF. The CoreWCF project template includes the ceremony for setting up the host. You will need to specify the bindings for each service in the **app.UseServiceModel** block, or use the XML configuration support to set up the bindings.

In-Place Upgrade to .NET Core

If you're doing an in-place upgrade of the project from .NET Framework to Core, there are various steps that need to be performed, regardless of the use of WCF. The Upgrade Assistant tool discussed in other articles in this issue can be used to perform those migration steps, such as updating project file format, changing NuGet references, namespace upgrades, etc.

At the time of writing this article, one of our summer interns has been working on additions to the Upgrade Assistant to perform some of the CoreWCF migration actions for you. By the time you read this, they may have been included in the tool. For more information, see the repo at https://github.com/dotnet/upgrade-assistant.

The manual steps that need to be performed after the .NET Upgrade Assistant has updated a project to .NET 6 are:

- Add a NuGet package reference to CoreWCF.Primitives and the packages for the type of binding you will be using.
- Replace any using System.ServiceModel; import with using CoreWCF; as the namespace has been changed.
- CoreWCF is built on top of ASP.NET Core, so you need to update the project to start an ASP.NET Core host.
  - Update the project's SDK to Microsoft.NET.Sdk.Web (because it uses ASP.NET Core).
  - Make the app's Main method async.
  - Replace ServiceHost setup with the code shown above.
- The services need to be exposed via bindings.
  - That can be done with code such as:

```
app.UseServiceModel(bld =>
{
    bld.AddService<Service>();
    bld.AddServiceEndpoint<Service, IService>(
      new BasicHttpBinding(BasicHttpSecurityMode.Transport),
      "/Service.svc");
});
```

- Each service needs to be added and be exposed by at least one binding.
- Or it can be done via configuration. If using configuration, be aware that not all of what you could specify in WCF is currently supported in CoreWCF. See the section on configuration above.
- For example, the <host> element isn't supported in the service model configuration. Instead, the port that endpoints listen to is configured in code. So, you need to remove the <host> element from wcf.config and add the following line to the app's main method, before the call to builder.Build:

```
builder.WebHost.UseNetTcp(8090);
```

What's Next for CoreWCF

The project is being developed in the open at https://github.com/corewcf/corewcf. Its release schedule is independent from that of .NET or Visual Studio: It releases when new functionality is ready to be consumed. We have plans for supporting additional Bindings such as NamedPipe and MessageQueue. As a community-driven project, we focus on what's most important to customers and have a pinned issue (https://github.com/CoreWCF/CoreWCF/issues/234) for discussing what features should be added next. If you want a listed feature, add a thumbs up and if it's not on the list, please create a new entry for it.

We welcome contributions from the community, both large and small. If you have a small change or fix, feel free to create a Pull Request (PR) with the change. If you want to contribute something larger, we suggest that you create an issue to discuss the feature and design before submitting a PR—that generally results in a smoother process for all involved.

Conclusion

CoreWCF provides a smoother modernization path for applications that have taken a strong dependency on WCF. Using CoreWCF is often a quicker migration path than re-designing services to use a different RPC mechanism.

Sam Spencer

Blazor for the Web and Beyond in .NET 7

The web is everywhere. Web apps run on devices of all shapes and sizes from desktop computers to mobile phones, thanks to ubiquitous implementation of modern open web standards. For developers seeking to build apps for a broad range of devices and platforms, the web provides unmatched cross-platform reach. .NET has supported building web apps from its earliest days.

Daniel Roth
daroth@microsoft.com
@danroth27

Daniel Roth is a Principal Product Manager at Microsoft on the ASP.NET team. He has worked on various parts of .NET over the years, including WCF, XAML, ASP.NET Web API, ASP.NET MVC, and ASP.NET Core. His current passion is making Web UI development easy with ASP.NET Core and Blazor.

The new .NET 7 release has end-to-end support for building web apps including high-performance back-end services with ASP.NET Core, rich interactive web UI with Blazor, and middleware for everything in between. Many of the largest, most heavily used web apps on the planet are built using .NET technologies. .NET provides a full stack and cross-platform solution for building web apps along with great tooling in Visual Studio and an active open-source ecosystem.

Blazor enables client web UI development with .NET without the need to write JavaScript. With Blazor, you author reusable web UI components using a combination of HTML, CSS, and C# that can then be used in any modern web browser. Unlike earlier attempts to run .NET in browsers, like the ill-fated Silverlight, Blazor relies only on open web standards, like WebAssembly and WebSockets. Blazor is also fully open source and has an active community of contributors and independent project maintainers.

Blazor became part of .NET and ASP.NET Core with .NET Core 3.0 back in 2019 and is fully supported as part of the .NET 6 long-term support (LTS) release. .NET 7 includes many great new Blazor features that make implementing web apps easier and more productive. Let's look at what .NET 7 has to offer for Blazor development!

Blazor Custom Elements

Blazor provides a powerful component model for encapsulating reusable pieces of web UI. This makes it easy to build reusable libraries of web UI components and to quickly build apps using preexisting components.

With .NET 7, you can now use Blazor components from existing JavaScript apps, including apps built with popular front-end frameworks like Angular, React, or Vue. .NET 7 adds support for using Blazor components as custom HTML elements.

> With .NET 7, you can now use Blazor components from existing JavaScript apps, including apps built with popular front-end frameworks like Angular, React, or Vue.

Modern browsers provide APIs for defining custom elements that can encapsulate UI elements. These custom elements can then be used with any web UI. You can host custom elements implemented using Blazor with either Blazor Server or Blazor WebAssembly.

To get started building custom elements with Blazor, you first need to add a package reference to Microsoft.AspNetCore.Components.CustomElements.

To register a Blazor component as a custom element, use the RegisterCustomElement<TComponent>() extension method and specify a name for the element. Be sure to use an element name with at least one dash (-) in it, as this is required by the custom elements standard.

In Blazor WebAssembly, you register a custom element like this in Program.cs:

```
builder.RootComponents.RegisterCustomElement
<Counter>("blazor-counter");
```

In Blazor Server, registering a custom element looks like this:

```
builder.Services.AddServerSideBlazor(options =>
{
    options.RootComponents.RegisterCustomElement
<Counter>("blazor-counter");
});
```

You can now add a blazor-counter element to your app to create a Counter component.

```
<blazor-counter></blazor-counter>
```

Note that to use the custom element you'll need to specify explicitly both the open and close tags: Custom elements don't support XML-style self-closing elements.

To set up your JavaScript app to use a Blazor custom element, you'll first need to decide if you want to use Blazor Server or Blazor WebAssembly and add the corresponding script to your app (blazor.server.js or blazor.webassembly.js).

```
<script src="_framework/blazor.webassembly.js">
</script> <!—or blazor.server.js →
```

To use Blazor Server, you'll need an ASP.NET Core app to host the Blazor components. With Blazor WebAssembly, you'll need to publish the app together with your JavaScript app. During development, you'll want to set up your front-end development server to proxy requests to the ASP.NET Core back-end or Blazor WebAssembly app and run both apps simultaneously. Using ASP.NET Core to host both apps can simplify the development experience. ASP.NET Core provides templates for hosting Angular-, React-, and Blazor WebAssembly-based apps that you can use to get set up quickly.

You can pass parameters to your Blazor custom elements using HTML attributes or by setting properties on the element object using JavaScript. For example, let's say the Counter component has a parameter for specifying the increment amount:

```
[Parameter]
public int IncrementAmount { get; set; }
```

You can specify the parameter in HTML on the custom element like this:

```
<blazor-counter increment-amount="10">
</blazor-counter>
```

Or using JavaScript like this:

```
const elem = document.querySelector(
    "blazor-counter");
elem.incrementAmount = 10;
```

Note that the HTML attribute name for the property is kebab case (increment-amount) and the JavaScript property is camelCase (incrementAmount).

With the JavaScript syntax, you can pass complex parameter values using JavaScript objects, which will then get JSON serialized. With HTML attributes, you're limited to passing simple types like strings, Booleans, or numerical types. Passing child content or other templated content via render fragments is not supported.

Data Binding Get/Set/After Modifiers

.NET 7 includes some nice improvements for configuring data binding in Blazor.

In Blazor, you can create a two-way binding between UI elements and component state using the @bind directive attribute:

```
<input @bind="searchText" />

@code {
    string searchText = "";
}
```

When the value of the input changes, the searchText field is automatically updated accordingly. Also, when the component renders, the value of the input is set to the value of the searchText field.

In .NET 7, you can now easily run async logic after a binding event has completed using the new **@bind:after** modifier:

```
<input @bind="searchText" @bind:after=
"PerformSearch" />

@code {
    string searchText = "";

    async Task PerformSearch()
    {
        // Do something async with searchText
    }
}
```

In this example, the **PerformSearch** async method runs automatically after any changes to the search text are detected.

It's also now easier to set up binding for component parameters. Components can support two-way data binding by defining a pair of parameters for the value and for a callback that's called when the value changes. The new **@bind:get** and **@bind:set** modifiers now make it trivial to create a component that binds to an underlying UI element.

```
<input @bind:get="Value" @bind:set=
"ValueChanged" />

@code {
    [Parameter]
    public TValue Value { get; set; }

    [Parameter]
    public EventCallback<TValue> ValueChanged
{ get; set; }
}
```

The @bind:get and @bind:set modifiers are always used together. The @bind:get modifier specifies the value to bind to, and the @bind:set modifier specifies a callback that's called when the value changes.

A parent component using the above component (let's call it MyInput) can now bind to the value of the input using @bind-Value.

```
<MyInput @bind-Value="text" />

@code {
    string text = "Type something great!";
}
```

Show App Loading Progress

The Blazor WebAssembly project template has a new loading UI that shows the progress of loading the app (see **Figure 1**).

The new loading screen is implemented with some simple HTML and CSS in the Blazor WebAssembly template using two new CSS custom properties (variables) provided by Blazor WebAssembly:

- **--blazor-load-percentage:** The percentage of app files loaded
- **--blazor-load-percentage-text:** The percentage of app files loaded rounded to the nearest whole number

Using these new CSS variables, you can create a custom loading UI that matches the style of your own Blazor WebAssembly apps.

Empty Blazor Templates

The Blazor project templates provide a great starting point for learning how to build your first Blazor template. In addition to setting up the Blazor app for development, these templates include sample pages demonstrating commonly used features as well as Bootstrap for CSS styling.

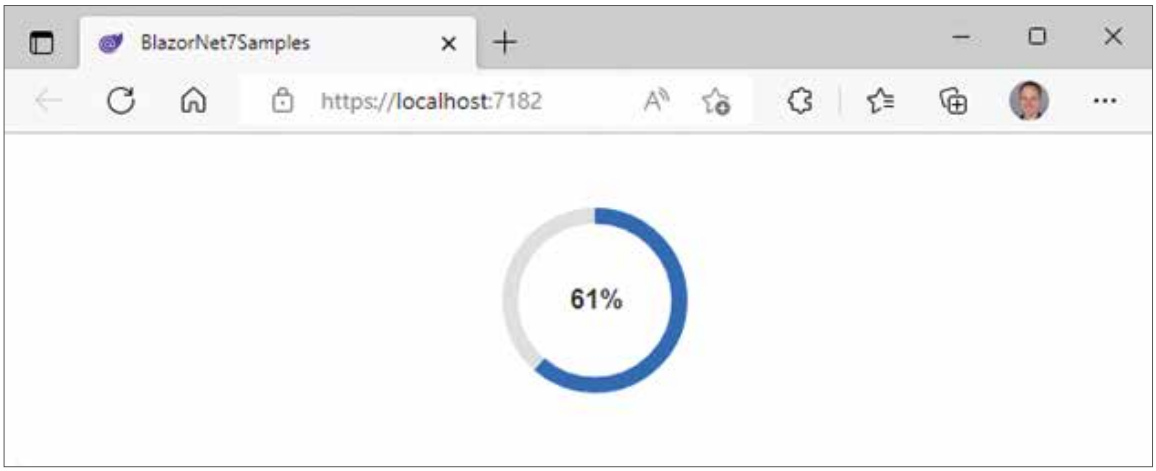

Figure 1: The new Blazor loading screen shows the app's loading progress.

After you've figured out the basics of using Blazor, the included demo code is no longer needed and having to repeatedly remove it can get tedious. So, in .NET 7, we've added new empty Blazor project templates that include only the bare necessities for creating a new Blazor app. You now have a completely blank canvas that you can make completely your own, including using whatever CSS design framework you want.

The new empty Blazor templates appear alongside the existing Blazor templates in the Visual Studio new project dialog (see **Figure 2**).

Handle Location Changing Events

Blazor in .NET 7 now has support for handling location changing events. This allows you to warn users about unsaved work or to perform related actions when the user performs a page navigation.

To handle location changing events, register a handler with the NavigationManager service using the RegisterLocationChangingHandler method. Your handler can then perform async work on a navigation or choose to cancel the navigation by calling PreventNavigation on the LocationChangingContext. RegisterLocationChangingHandler returns an IDisposable instance that, when disposed, removes the corresponding location changing handler.

> The new NavigationLock component makes common scenarios for handling location changing events simple.

For example, the following handler prevents navigation to the counter page:

```
var registration = NavigationManager
.RegisterLocationChangingHandler(async cxt =>
{
    if (cxt.TargetLocation.EndsWith("counter"))
    {
        cxt.PreventNavigation();
    }
});
```

Note that your handler will only be called for internal navigations within the app. External navigations can only be handled synchronously using the **beforeunload** event in JavaScript.

The new NavigationLock component makes common scenarios for handling location changing events simple.

```
<NavigationLock
    OnBeforeInternalNavigation="ConfirmNavigation"
    ConfirmExternalNavigation />
```

NavigationLock exposes an OnBeforeInternalNavigation callback that you can use to intercept and handle internal location changing events. If you want users to confirm external navigations too, you can use the ConfirmExternalNavigations property, which will hook the beforeunload event for you and trigger the browser-specific prompt. The NavigationLock component makes it simple to confirm user navigations when there's unsaved data. **Listing 1** shows using NavigationLock with a form that the user may have modified but not submitted.

Dynamic Authentication Requests

Blazor provides out-of-the-box support for authentication using OpenID Connect and a variety of identity providers including Azure Active Directory (Azure AD) and Azure AD B2C. In .NET 7, Blazor now supports creating dynamic authentication requests at runtime with custom parameters to handle more advanced authentication scenarios in Blazor WebAssembly apps.

To specify additional parameters, use the new InteractiveRequestOptions type and helper methods on NavigationManager. For example, you can specify a log-in hint to the identity provider to indicate who to authenticate, like this:

```
InteractiveRequestOptions requestOptions = new()
    {
        Interaction = InteractionType.SignIn,
        ReturnUrl = NavigationManager.Uri,
```

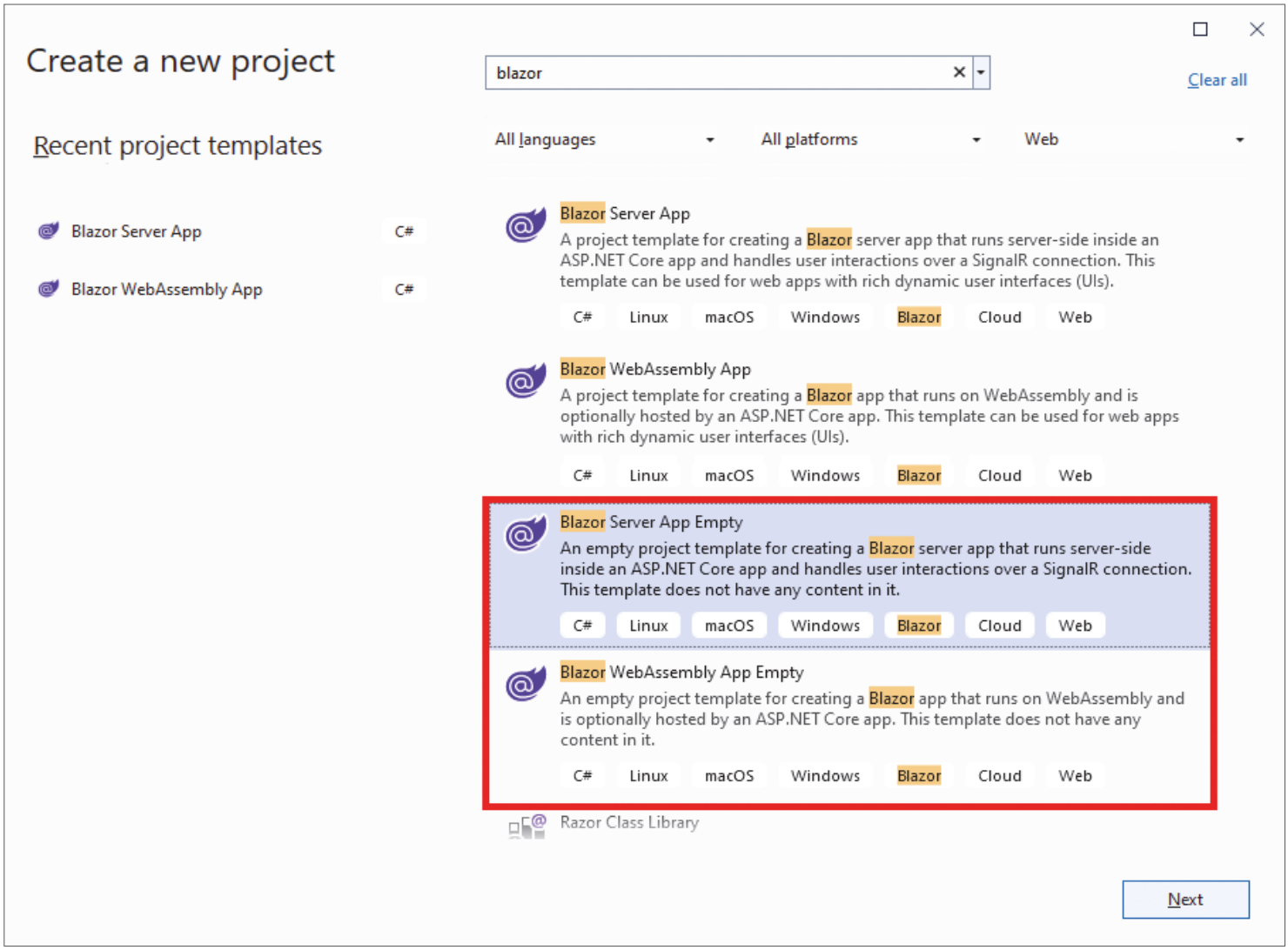

Figure 2: Start your next Blazor app with a blank canvas using the new empty Blazor project templates.

```
    };
requestOptions.TryAddAdditionalParameter(
    "login_hint", "user@example.com");
NavigationManager.NavigateToLogin(
    "authentication/login", requestOptions);
```

Similarly, you can specify the OpenID Connect prompt parameter, like when you want to force an interactive login:

```
requestOptions.TryAddAdditionalParameter(
    "prompt", "login");
```

You can specify these options when using IAccessTokenProvider directly to request tokens:

```
var result = await AccessTokenProvider
    .RequestAccessToken(
        new() { Scopes = new[] { "edit" } });

if (!result.TryGetToken(out var token))
{
    result.InteractionOptions
        .TryAddAdditionalParameter(
            "login_hint", "user@example.com");
```

Listing 1: Use the NavigationLock component to confirm user navigations when there is unsaved form data.

```
<EditForm EditContext="editContext" OnValidSubmit="Submit">
    ...
</EditForm>
<NavigationLock OnBeforeInternalNavigation="ConfirmNavigation"
ConfirmExternalNavigation />

@code {
    private readonly EditContext editContext;
    ...

    // Called only for internal navigations.
    // External navigations will trigger a browser specific
    // prompt.
    async Task ConfirmNavigation(LocationChangingContext context)
    {
        if (editContext.IsModified())
        {
            var isConfirmed = await JS.InvokeAsync<bool>(
                "window.confirm",
                "Are you sure you want to leave this page?");

            if (!isConfirmed)
            {
                context.PreventNavigation();
            }
        }
    }
}
```

```
    NavigationManager.NavigateToLogin(
        result.InteractiveRequestUrl,
        result.InteractionOptions);
}
```

You can also specify authentication request options when making HTTP requests and the token cannot be acquired by the AuthorizationMessageHandler without user interaction:

```
try
{
    await Http.GetAsync("/orders");
}
catch (AccessTokenNotAvailableException ex)
{
    ex.Redirect(requestOptions =>
    {
        requestOptions.TryAddAdditionalParameter(
            "login_hint", "user@example.com");
    });
}
```

Any additional parameters specified for the authentication request will be passed through to the underlying authentication library and on to the identity provider.

Hot Reload Improvements

.NET 6 introduced hot reload support, which is the ability to apply code changes to your running app during development without having to restart it. Both Blazor Server and Blazor WebAssembly apps support hot reloading changes, although hot reload for Blazor WebAssembly apps was significantly more limited.

.NET 7 improves hot reload support for Blazor WebAssembly apps by adding the following capabilities:

- Add new types
- Add nested classes
- Add static and instance methods to existing types
- Add static fields and methods to existing types
- Add static lambdas to existing methods
- Add lambdas that capture this to existing methods that already captured this previously

Work is ongoing to improve hot reload across all of .NET, so expect more hot reload improvements to come in future releases.

Blazor WebAssembly Debugging Improvements

Debugging Blazor WebAssembly apps presents unique challenges because the app runs within a web browser. To debug your Blazor WebAssembly code, Visual Studio connects to the browser via a .NET debugging proxy using the browser's JavaScript debugging protocol.

Blazor WebAssembly debugging in .NET 7 now has the following improvements:

- Support for the Just My Code setting to show or hide type members not from user code
- Support for inspecting multidimensional arrays
- The Call Stack window now shows the correct name for async methods
- Improved expression evaluation
- Correct handling of the "new" keyword on derived members
- Support for debugger-related attributes in System.Diagnostics

Expanded Crypto Support

.NET 7 includes expanded support for cryptographic algorithms when running on WebAssembly. The following algorithms are now also supported:

- SHA1, SHA256, SHA384, SHA512
- HMACSHA1, HMACSHA256, HMACSHA384, HMACSHA512
- Rfc2898DeriveBytes (PBKDF2)
- HKDF

Improved JavaScript Interop on WebAssembly

.NET 7 introduces a new low-level mechanism for using .NET in JavaScript-based apps. With this new JavaScript interop capability, you can invoke .NET code from JavaScript using the .NET WebAssembly runtime as well as call into JavaScript functionality from .NET without any dependency on the Blazor UI component model.

The easiest way to see the new JavaScript interop functionality in action is using the new experimental templates in the **wasm-experimental** workload. You can install the wasm-experimental workload using the following command:

```
dotnet workload install wasm-experimental
```

This workload contains two project templates: WebAssembly Browser App, and WebAssembly Console App (see **Figure 3**).

These templates are experimental, which means the developer workflow for them hasn't been fully sorted out yet. But the .NET and JavaScript APIs used in these templates are supported in .NET 7 and provide a foundation for using .NET on WebAssembly from JavaScript.

The WebAssembly Browser App template creates a simple web app that demonstrates using .NET directly from JavaScript in a browser. The WebAssembly Console App is similar, but runs as a Node.js console app instead of a browser-based web app.

The root HTML file in both templates imports a JavaScript module (main.js or main.mjs) that imports the relevant APIs from dotnet.js for working with .NET from JavaScript using WebAssembly.

```
import { dotnet } from './dotnet.js'

const {
    setModuleImports,
    getAssemblyExports,
    getConfig,
    runMainAndExit
} = await dotnet.create();
```

These APIs enable you to set up named modules that can be imported into your C# code.

Apex Data Solutions
– Artwork Coming

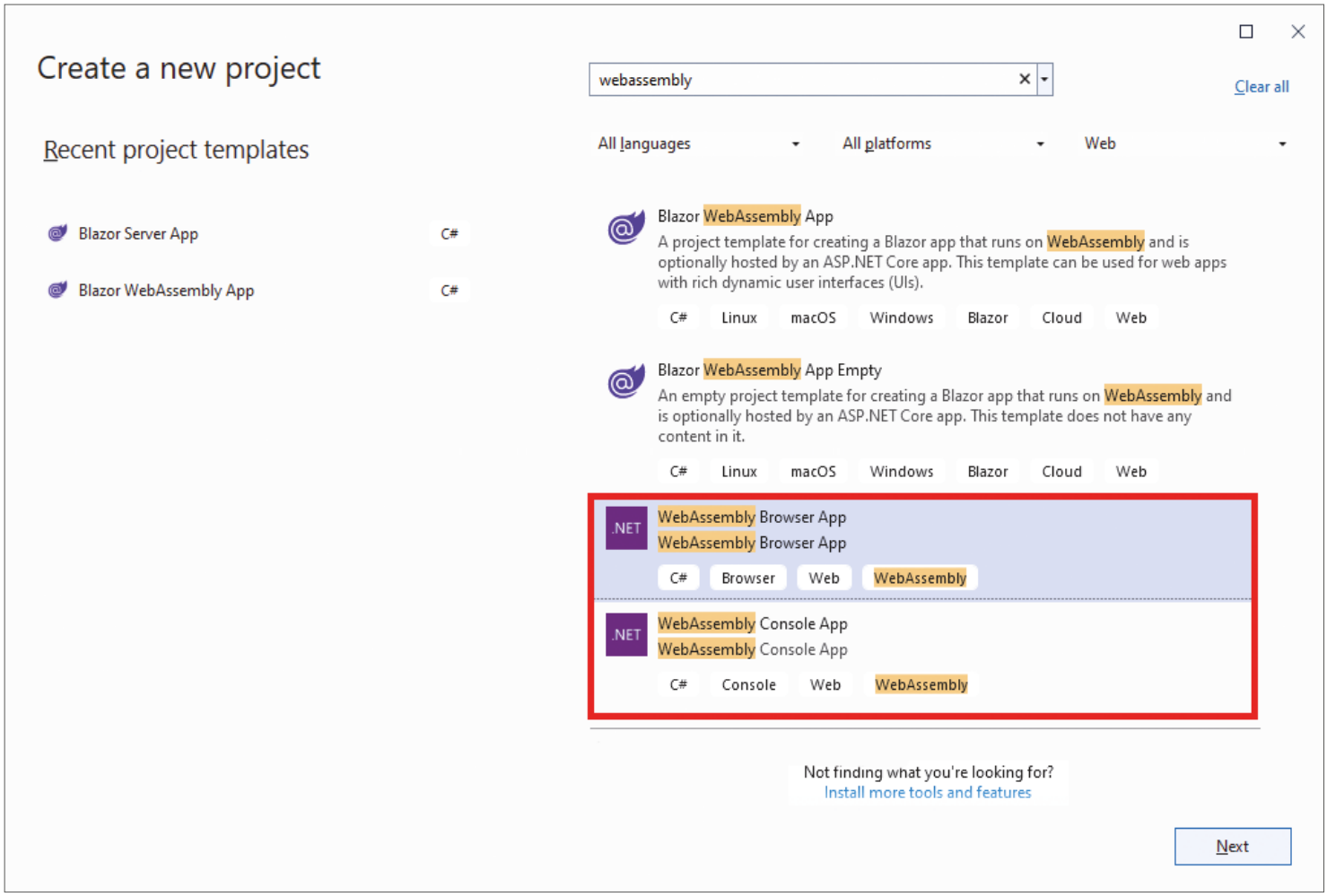

Figure 3: The experimental WebAssembly Browser App and WebAssembly Console App templates show how to execute .NET on WebAssembly from JavaScript.

Running .NET on WebAssembly isn't just for Blazor

Blazor WebAssembly apps can run in modern web browsers thanks to a .NET runtime implemented in WebAssembly. This runtime can be used for more than just Blazor apps.

The new JavaScript interop support in .NET 7 makes it straightforward for JavaScript apps to call into existing .NET libraries without any dependency on the Blazor component model.

The .NET WebAssembly runtime also enables other alternative UI frameworks based on .NET to run in web browsers, like Uno, OpenSilver, and Ooui.

```
setModuleImports('main.js', {
    window: {
        location: {
            href: () =>
                globalThis.window.location.href
        }
    }
});
```

To import a JavaScript function so it can be called from C#, use the new JSImportAttribute on a matching method signature:

```
[JSImport("window.location.href", "main.js")]
internal static partial string GetHRef();
```

The first parameter to the JSImportAttribute is the name of the JavaScript function to import and the second parameter is the name of the module, both of which were set up by the setModuleImports call.

In the imported method signature, you can use .NET types for parameters and return values, which will be marshaled for you. Use JSMarshalAsAttribute<T> to control how the imported method parameters are marshaled. For example, you might choose to marshal a long as JSType.Number or JSType.BitInt. You can pass Action/Func callbacks as parameters, which will be marshaled as callable JavaScript functions. You can pass both JavaScript and managed object references and they will be marshaled as proxy objects, keeping the object alive across the boundary until the proxy is garbage collected. You can also import and export asynchronous methods that return a Task, which will be marshaled as JavaScript promises. Most of the marshaled types work in both directions, as parameters and as return values, on both imported and exported methods.

To export a .NET method so it can be called from JavaScript, use the JSExportAttribute:

```
[JSExport]
internal static string Greeting()
{
    var text = $"Hello from {GetHRef()}";
    Console.WriteLine(text);
    return text;
}
```

You can then get the exported methods in JavaScript and invoke them.

```
const config = getConfig();
const exports = await getAssemblyExports(
```

```
    config.mainAssemblyName);
const text = exports.MyClass.Greeting();
```

You can also invoke the main entry point in a .NET app.

```
await runMainAndExit(
    config.mainAssemblyName,
    ['dotnet', 'is', 'great!']);
```

Blazor provides its own JavaScript interop mechanism based on the IJSRuntime interface, which is uniformly supported across all Blazor hosting models. This common asynchronous abstraction enables library authors to build JavaScript interop libraries that can be shared across the Blazor ecosystem and is still the recommend way to do JavaScript interop in Blazor.

In Blazor WebAssembly apps, you also had the option to make synchronous JavaScript interop calls using the **IJSInProcessRuntime** or even unmarshalled calls using the IJSUnmarshalledRuntime. IJSUnmarshalledRuntime was tricky to use and only partially supported. In .NET 7, IJSUnmarshalledRuntime is now obsolete and should be replaced with the [JSImport]/[JSExport] mechanism. Blazor doesn't directly expose the dotnet runtime instance it uses from JavaScript, but it can still be accessed by calling getDotnetRuntime(0). You can also import JavaScript modules from your C# code by calling JSHost.ImportAsync, which makes the module's exports visible to [JSImport].

Blazor Hybrid

Blazor isn't just for web apps! Blazor components can also be hosted in native client apps using the Blazor Hybrid hosting model. In a Blazor Hybrid app, your components run directly on the device, not from within a browser. WebAssembly isn't used and there's no need for a web server. The Blazor components render to an embedded web view control using a local interop channel. Blazor Hybrid apps run fast and have full access to the native device's capabilities through normal .NET APIs. You can reuse existing Blazor web UI components with Blazor Hybrid apps and share them with both your web and native client apps. With Blazor Hybrid you can build a single shared UI for mobile, desktop, and web.

Blazor Hybrid support is included with .NET MAUI in .NET 7. By hosting your Blazor components in a .NET MAUI app, you can build native mobile and desktop apps using your existing web development skills. .NET MAUI provides access to many native device capabilities through a common cross-platform API, including the ability to reuse native UI controls alongside your Blazor components. To create a new Blazor Hybrid app with .NET MAUI, simply use the included .NET MAUI Blazor App project template. Blazor Hybrid support is also available for WPF and Windows Forms apps on NuGet.

Blazor Just Keeps Getting Better!

Blazor has come a long way from its humble beginnings. Blazor today can be used to build a web-based UI for nearly any kind of app. With the addition of the many improvements to Blazor .NET 7, Blazor continues to provide itself as a mature and modern client web UI framework. We hope you enjoy working with Blazor in .NET 7 and we look forward to seeing what you build with these new capabilities!

Daniel Roth

CODE

CODE Focus Nov 2022
Volume 19 Issue 1

Group Publisher
Markus Egger

Associate Publisher
Rick Strahl

Editor-in-Chief
Rod Paddock

Managing Editor
Ellen Whitney

Content Editor
Melanie Spiller

Editorial Contributors
Otto Dobretsberger
Jim Duffy
Jeff Etter
Mike Yeager

Writers In This Issue
Jon Douglas
Jeremy Likness
Angelos Petropoulos
Mike Rousos
Stephen Toub
Shawn Wildermuth
Julie Lerman
David Ortinau
Daniel Roth
Sam Spencer
Bill Wagner

Technical Reviewers
Markus Egger
Rod Paddock

Production
Friedl Raffeiner Grafik Studio
www.frigraf.it

Graphic Layout
Friedl Raffeiner Grafik Studio in collaboration with onsight (www.onsightdesign.info)

Printing
Fry Communications, Inc.
800 West Church Rd.
Mechanicsburg, PA 17055

Advertising Sales
Tammy Ferguson
832-717-4445 ext 26
tammy@codemag.com

Circulation & Distribution
General Circulation: EPS Software Corp.
Newsstand: The NEWS Group (TNG)
Media Solutions
The Mail Group

Subscriptions
Subscription Manager
Colleen Cade
ccade@codemag.com

US subscriptions are US $29.99 for one year. Subscriptions outside the US are US $50.99. Payments should be made in US dollars drawn on a US bank. American Express, MasterCard, Visa, and Discover credit cards accepted. Bill me option is available only for US subscriptions. Back issues are available. For subscription information, e-mail subscriptions@codemag.com.

Subscribe online at
www.codemag.com

CODE Developer Magazine
6605 Cypresswood Drive, Ste 425, Spring, Texas 77379
Phone: 832-717-4445
Fax: 832-717-4460