C# 10, .NET MAUI, Blazor, Visual Studio 2022

CODE

NOV 2021

CODE FOCUS

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

.NET 6.0 VISUAL STUDIO

Explore What's New in ASP.NET Core

Changes to EF Core

Building Power Apps with .NET 6

Inside Front cover .N
from MS

Jet Artwork to come

Features

8 The Unified .NET 6

There were many lessons learned as the .NET team released .NET 5 during the lockdown with an all-remote team. Rich shows how those lessons carried into .NET 6 with major performance improvements, multiple operating system scenarios for building client apps, support for Apple Silicon chips, and faster and more responsive development tools.

Rich Lander

13 Bring Your .NET Apps Forward with the .NET Upgrade Assistant

Now that you're using all the shiny new tools in .NET 6, you need to make sure that the rest of your .NET Framework is keeping up. Mike shows you how the new Upgrade Assistant makes it easy.

Mike Rousos

19 Visual Studio 2022 Productivity

VS 2022 is finally 64-bit! Mika shows you how, with enhanced speed, AI coding assistance, expanded productivity tools, and streamlined team collaboration, you'll find this new version improving your workdays.

Mika Dumont

29 Essential C# 10.0: Making it Simpler

It's time for the annual release of C# vNext. Mark shows you how it's streamlined in some ways and tightened in others. In fact, he thinks it will mark a sea change in how C# devs write code.

Mark Michaelis

35 What's New in ASP.NET Core in .NET 6

You already know that ASP.NET Core provides everything you need to build great Web UIs and powerful back-end services. Daniel shows how you can build rich interactive client Web UIs using all your favorite interactivity tools, standards-based HTTP APIs, real-time services, and back-end services.

Daniel Roth

46 EF Core 6: Fulfilling the Bucket List

EF Core just gets better and better. Julie shows you how the development team listened to the community for this latest release as she explores all the cool new tools.

Julie Lerman

54 An Introduction to .NET MAUI

You've been using Xamarin for years. Steven shows how the .NET Multi-platform App UI (.NET MAUI) hasn't just kept up with everything, but how it compares with the old Xamarin.Forms.

Steven Thewissen

59 Blazor Hybrid Web Apps with .NET MAUI

You've been waiting for MAUI, and now it's here! Ed takes you on a tour and shows you how easy it is to code for the Web, desktops, and mobiles using the skills you already have.

Ed Charbeneau

68 Power Up Your Power Apps with .NET 6 and Azure

Power Apps help design and specify how a mobile app will function without having to know all those troublesome details of being a professional coder. Come along as Brady walks you through .NET 6's new ASP.NET Core Minimal APIs, then publishes the app to Azure App Service, imports it into Azure API Managements, and secures it with Microsoft Identity Platform.

Brady Gaster

Departments

6 .NET Focus Features Fabulous Features

Rod and the team here at CODE Magazine are pretty excited about the new .NET release.

Rod Paddock

32 Advertisers Index

74 Code Compilers

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.

.NET Focus Features Fabulous Features

I think this is the third .NET Focus issue we've shipped, and I can say without a doubt that these .NET issues are very fun to put together. Well, it's not all fun. LOL. Sometimes the deadlines are a bit crazy, but overall, the experience is quite rewarding. This issue is no exception. The part I find most rewarding is the privilege of working with the teams building these features, creating in-depth content that our readers will be able to take advantage of almost immediately. Another rewarding aspect is getting a personal tour of what's most important in the version of .NET we're creating content for from the people responsible for making those changes. As you're about to see, there's a lot to this tour.

At a high level, this version of .NET is a unification version. Rich Lander does a good job of conveying the overarching vision of .NET 6 in his aptly named article "The Unified .NET 6." This article is not just "fluff." There are a lot of details to this unification, including performance improvements. Check it out—you won't be disappointed.

There are a TON of exciting new features in this version of .NET and I believe my favorite one is the concept of Minimal APIs. Daniel Roth does a great job in his article "What's New in ASP.NET Core in .NET 6," demonstrating how to build a reasonably complex Web API with a single file. I love this new ability as we'll no longer need to build completely scaffolded Web projects to just "try something." In my humble opinion, this is one of the most important aspects of this release. And this is just one of the cool features being shipped in ASP.NET Core.

My next favorite feature is just a simple little thing. In C# 10, we can now create global Using statements. No more redundant Includes at the top of every program you create. You just put the common Using statements in a common file and cut down on the "cruft" in your programs. This's just one cool feature in C# 10. Check out Mark Michaelis' article, "Essential C# 10.0: Making It Simple" for more details.

This next one is a HUGE set of changes that every .NET developer will love. Visual Studio 2022 is now—drum roll please—64-bit. Yes! You read that correctly. In Mika Dumont's article, "Visual Studio 2022 Productivity," you'll learn that a major benefit of this release is SPEED, SPEED, and more SPEED. The shift to 64 bits will make VS compile faster, search faster, and make your everyday development work...well...faster! <g> As if this isn't enough already, there are a ton of other features for developers. No spoilers here: Check this article out for yourself. There's lots of great information here.

Another area of focus that you'll find fascinating are the various user interface frameworks that are in heavy development now. We have content on the .NET Multi-platform App UI (.NET MAUI), Blazor, and Power Apps. The work that the .NET team has been doing over the years is paying huge dividends in the .NET 6 release. There are now many ways to build applications for any platform you want. Want C# code in your Web applications? Blazor can help. Want to build mobile applications? Take a trip to .NET MAUI to make that interface painless for you and the users. Want to build Office 365 applications? It's Power Apps to the rescue.

I'm looking forward to seeing how these various platforms play out over the coming years.

Along with these features and tools, there are also some great articles on tools you can use to migrate your applications to the current version, as well as deep dives into performance updates in the overall .NET framework, .NET 6. And the beauty of this performance work is that it's essentially "free." There are no major changes to your code and yet there are huge performance gains.

And last but not least. The Entity Framework is making huge strides for us "gear heads" who love data. Audit tables, performance improvements, improvements to migrations and, finally, more work on the CosmoDB provider. For EF Core users, there's definitely a lot to love.

Like all releases, .NET 6.0 has something for everyone and we're just scratching the surface in this issue. I hope you like what we have curated for you.

Rod Paddock
CODE

The Unified .NET 6

The .NET 6 project started in late 2020 as Microsoft was finishing .NET 5. .NET 5 proved to be a very successful base to start from, having been the first release to tackle .NET platform unification, the first of the annual November releases, the first "all remote" team release (due to the pandemic), and it has been (most importantly) rapidly and broadly adopted. The.NET 5 release cycle

Rich Lander
rlander@ms
Twitter: @runfaster2000
GitHub: @richlander

Richard Lander is a Principal Program Manager on the .NET team at Microsoft. He works on making .NET work well in the cloud, in memory-limited Docker containers, and on ARM hardware like the Raspberry Pi. He's part of the design team that defines new .NET runtime capabilities and features. Richard also focuses on making the .NET open source project a safe inclusive place for people to learn, do interesting projects, and develop their skills. He also writes extensively for the .NET blog. Richard reported for work at Microsoft in 2000, having just graduated from the University of Waterloo (Canada) with an Honors English degree, with intensive study areas in Computer Science and SGML/XML Markup Languages. In his spare time, he swims, bikes, and runs, and enjoys using power tools. He grew up in Canada and New Zealand.

taught Microsoft how to better span major investments across multiple releases, which continues into .NET 6. The new release delivers major performance improvements, enables new scenarios for building client apps for multiple operating systems, adds support for Apple Silicon chips, and provides much faster and more responsive development tools with hot reload. At the same time, it improves on existing scenarios.

.NET users see keeping up with .NET innovation as a key ingredient of their business success, expanding their developer workforce to include the .NET team, and taking advantage of performance improvements, observability, and new language features. Microsoft thinks that .NET developers will be eager to convert to .NET 6.

This article is focused on the fundamentals of the release, including runtime, libraries, and SDK. It's these fundamental features that you experience and interact with most every day, with new libraries APIs, language features, runtime plumbing, and SDK capabilities. The article provides a look at only a handful of improvements and new capabilities. You'll want to check out the .NET Team blog (https://devblogs.microsoft.com/dotnet/) to learn about the whole release.

Unifying the .NET Platform as net6.0-everything

The top headline of the release (and this article) is unifying the .NET platform. Looking several years back, the .NET Framework with Windows was on one side, and Xamarin with Android and Apple operating systems was on the other. They were both ".NET" but were defined more by their differences than their commonality. .NET 6 unifies the experience and product into a single offering.

The following items unify the platform:

- Uniform runtime and library implementation and common APIs
- Symmetric model for targeting operating systems, like Android and Windows
- Support for all of the relevant operating systems and environments
- Tools that enable building all app types
- Opt-in to targeting of additional experiences, enabling a significant limit to the time and size it takes to use .NET on your computer
- New functionality is available to all .NET developers at the same time

Let's take a look at a number of templates to better demonstrate what you'll see in .NET 6.

Cross-Platform Model

I'll start with the Console template (class library is the same: https://docs.microsoft.com/en-us/dotnet/core/tutorials/library-with-visual-studio?pivots=dotnet-5-0) because it's the baseline by which you'll judge all others. You can think of net6.0 as the cross-platform target framework moniker (TFM).

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

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

</Project>
```

Note: All unrelated content has been removed in these examples. The actual templates are longer and include other configuration, like enabling nullability. Those changes are also important but aren't covered in this article.

Apps that target the net6.0 TFM will work on all supported operating systems and CPU architectures. The APIs exposed via the net6.0 TFM are designed to work everywhere, like HttpClient**. There are platform compat analyzers that warn you in the few cases where APIs are OS-specific.**

There's nothing surprising in this template. It has a reference to the base SDK: **Microsoft.NET.SDK**. As an aside, the SDK reference is the reason this project format is often called "SDK-style." The project also declares that it's a .NET 6 app by specifying a dependence on the net6.0 target framework.

As an aside, the net6.0 TFM, and net5.0 before it, satisfy the same purpose as .NET Standard. .NET Standard is still supported but Microsoft is no longer making new versions. You can think of net6.0 as your new .NET Standard, if you'd like. One of the major improvements over .NET Standard is that it works for apps, not only libraries.

ASP.NET Core apps are nearly identical but reference a different SDK, which is **Microsoft.NET.Sdk.Web**. That's the mechanism that provides Web apps with additional APIs and build-time functionality (like Razor page compilation) as compared to Console apps.

Operating System API Targeting

In terms of existing templates, Windows Forms and WPF apps introduce operating system targeting.

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

  <PropertyGroup>
    <TargetFramework>net6.0-windows
    </TargetFramework>
    <UseWindowsForms>true</UseWindowsForms>
  </PropertyGroup>

</Project>
```

There are two differences to call out. The first is that Microsoft has extended the target framework to describe and include operating system APIs. The change was first made in

.NET 5. This is apparent in net6.0-windows because Windows is an operating system. Although Windows Forms and WPF aren't Windows APIs, they're available only on Windows and rely heavily on Windows technologies. As a result, Microsoft chose to expose them with the Windows-specific TFM. Windows APIs, including Windows Forms and WPF, aren't available if you target the cross-platform net6.0 target framework.

The second change is that .NET 6 doesn't expose application-specific SDKs. You'll notice that the Windows Forms project uses the base Microsoft.NET.Sdk and also sets the **UseWindowsForms** property to true. WPF works the same way. The **UseXYZ** property tells the base SDK which additional SDKs should be imported as an implementation detail. There are all the same SDKs as before but they're not a formal part of the project file. This is the new model going forward. It may be applied to ASP.NET Core templates in a future release.

This new model was created to enable multi-targeting. SDKs don't play nicely with multi-targeting, at least not with the way they're currently exposed as a singular attribute value. They also don't work well for composing multiple technologies. For example, imagine that you want to expose a Web endpoint from a client app. Which SDK would you put at the top of the file? With the new model, that problem goes away.

Before I switch to looking at other operating systems, let's take a closer look at the Windows TFM. The new **net6.0-windows** has no version number, yet .NET 6 supports multiple Windows versions. The version-less TFM (as it relates to the operating system) targets the lowest-supported operating system version. In this case, that's Windows 7. If you want access to WinRT APIs, you need to target Windows 10. You can use **net6.0-windows10.0.17763.0** to target Windows 10, version 1809, for example.

Expanding Supported Operating Systems

Now that you've taken a look at the more familiar Windows experience, check out how the same model plays out for Android, macOS, and iOS. The spoiler is that it's the same.

The following are the TFMs for these OSes:

- net6.0-android
- net6.0-maccatalyst
- net6.0-ios

These TFMs are version-less, just like net6.0-windows. They are all equivalent to the lowest-supported versioned TFM for each of those operating systems. For example, net6.0-ios and net6.0-ios14 are equivalent. For .NET 7, perhaps net7.0-ios and net7.0-ios15 will similarly match.

You may not be familiar with Mac Catalyst. It's a newer macOS application type defined by Apple and a variant of iOS (including iOS UI APIs) that's optimized for desktop apps. Its primary purpose is to make source code sharing between iOS and macOS platforms easier and to provide macOS developers with access to the newest Apple APIs (which have historically only been available with iOS). For .NET 6, Microsoft decided to prioritize Mac Catalyst over Mac (classic). There's no support with .NET 6 for creating non-Mac Catalyst Mac apps and no net6.0-macos target framework.

You can see this all coming together with a .NET Multi-platform App UI (.NET MAUI) app.

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

  <PropertyGroup>
    <TargetFrameworks>net6.0-android;net6.0-ios;
    net6.0-maccatalyst</TargetFrameworks>
    <UseMaui>true</UseMaui>
  </PropertyGroup>

</Project>
```

This example is taken from the Weather `21 app that you can find on GitHub, here: https://github.com/davidortinau/WeatherTwentyOne.

You can see a few design points at play:

- The app multi-targets over three target frameworks.
- The SDK is uniform and coherent across all three because it's the base SDK.
- The app declares that it's a .NET MAUI app—with UseMaui—across all target frameworks, which results in MAUI-specific build tasks and other configuration.

You can see that there's added support for Android, iOS, and macOS with .NET 6 (previously all supported by Xamarin) and that they're modeled in the same way as Windows. These new operating systems have first-class support at the most fundamental levels of the .NET SDK.

macOS and Windows Arm64

Continuing with client operating systems, there's added support for Arm64 CPUs for macOS and Windows. For macOS, that's new with .NET 6 and for Windows, Microsoft is building on .NET 5 capabilities. Both Arm64 operating systems offer x64 emulation, which, on one hand, is zero cost for Microsoft but on the other hand, has caused Microsoft to significantly rethink the .NET installation model and the CLI support for architecture targeting.

macOS Arm64

Let's start with macOS. You've probably heard about Apple's move to Apple Silicon chips, called (in this timeframe) "M1" and "M2." They are essentially the desktop version of the A-series iPhone chips, which are all the way up to (in this timeframe) "A14" and "A15". Microsoft has had support for Arm64 (on Linux) since the .NET Core 3.0 release, and Arm32 before that. That all helped, but Apple required implementation of a couple of security-oriented features above and beyond the existing .NET Arm64 capability.

The primary requirement was adding support for the W^X memory feature, which was already on Microsoft's backlog. Memory pages (think virtual memory) can (in theory) be marked with any or all of three states: read, write, and execute. Think of these as permissions or capabilities. When running on Apple Silicon chips, macOS doesn't allow a memory page to be configured for both write and execute. This prevents an attacker from generating code at runtime and then causing the application to execute it. That's why the feature is called "write exclusive execute" or "write xor execute." Pages can be read-write or read-execute but never write-execute or read-write-execute. Some parts of the runtime, like the JIT, relied on r-w-x pages and have since been adapted to new approaches that only use the allowable memory page types.

For .NET 6, this memory-related feature is enabled by default for macOS on Apple Silicon computers, and is otherwise opt-in. Microsoft expects it to be enabled by default for all environments with .NET 7. It's a good security feature and will benefit all .NET developers and deployments. There's a roadmap of defense-in-depth features, and others are planned for .NET 7 and future releases to further secure applications.

X64 Emulation

The most significant Arm64-related change is x64 emulation, which is available on both macOS and Windows (on Arm64 computers). The primary issue is that x64 emulation (on both operating systems) is a very narrow capability (focused nearly exclusively on instruction set emulation), as compared to the broad WoW64 subsystem on Windows that supports 32-bit x86 apps including file and registry virtualization. That means that .NET and other development platforms are responsible for the bulk of the user experience for supporting x64 emulation.

First, the team needed to enable developers to install both Arm64 and x64 .NET builds on the same computer. At the start of the release, and at the time of writing, these builds collide (in multiple ways). That's not a workable model. Microsoft has been working on a plan—documented at dotnet/designs—for enabling Arm64 and x64 builds to coexist and to be insensitive to the order of install.

Going forward, it's expected that most developers (on Arm64 macOS and Windows computers) will exclusively install the Arm64 .NET SDK (which will also include Arm64 runtimes for that version) for building code and then install and use whichever additional Arm64 and x64 runtimes they want to use for running and testing it. For developers, x64 runtime usage (on Arm64 computers) will probably be limited to ensuring compatibility with x64 production targets (both cloud and client) and validating x64-specific bugs. Most x64 validation is expected to be performed by x64-capable continuous integration (CI). Microsoft expects this to be common for many years. A common need for the x64 SDK on Arm64 computers isn't expected, although it will be available.

Microsoft also expects end users to use x64-only apps on Arm64 as a popular scenario.

The .NET CLI syntax has been extended to make targeting x64 easier with the Arm64 SDK. The following is an example of that.

Here's the .NET 6 app.

```
using System.Runtime.InteropServices;

Console.WriteLine($"Hello,
  {RuntimeInformation.OSArchitecture}!");
```

Assuming the .NET 6 Arm64 SDK is installed, the app runs as Arm64 by default. Let's validate that.

```
rich@M1 % dotnet build
rich@M1 % ./bin/Debug/net6.0/yyzapp
Hello, Arm64!
```

Using the Arm64 SDK again, you can also target the app to x64 with the new **-a** (architecture) switch to produce an x64 app instead of the default native architecture. This assumes that the .NET 6 x64 runtime is installed, because otherwise the app wouldn't run.

```
rich@M1 % dotnet build -a x64
rich@M1 % ./bin/Debug/net6.0/osx-x64/yyzapp
Hello, X64!
```

The same thing works with dotnet run and dotnet test.

The goal with x64 emulation was to deliver an experience that was intuitive to use and could be driven entirely from the Arm64 SDK. Microsoft focused on the Arm64 SDK because most developers have that anyway and because it's faster, by definition, given that it isn't emulated. The .NET build system is a significant body of software and it's going to run much faster natively on Apple Silicon chips.

Effect on Containers?

You might be wondering how all of this affects containers. The answer is: Not a lot.

```
rich@M1 ~ % docker run --rm mcr.microsoft.com/dotnet/samples

Debian GNU/Linux 10 (buster)
OSArchitecture: Arm64
```

By default, Docker runs in Arm64 mode on Apple Silicon, the native architecture of the computer. Just like on Mac Intel computers, Docker uses Linux images, so no change there. You can also run x64 container images using QEMU-based emulation. Microsoft doesn't support .NET running in QEMU (on any operating system). That said, I'll at least show you how it works, using the **--platform** switch, so you can try it out.

```
rich@M1 ~ % docker run --rm --platform
linux/amd64 ubuntu bash -c "cat /etc/os-release
| grep PRETTY && uname -a"
PRETTY_NAME="Ubuntu 20.04.2 LTS"
Linux a881a5627af8 5.10.47-linuxkit
x86_64 x86_64 x86_64 GNU/Linux
```

System.Text.Json Source Generators

One of the goals, if not the most fundamental goal, of high-level programming languages is to compile human-centric abstractions down to machine-centric optimized (and safe) code. Aspects of .NET do just that, like the garbage collector, the thread pool, and async/await. Those features have a well-defined contract with the rest of the system. For the System.Text.Json serializer (and really any serializer), it's a lot harder to separate the human-centric API from the runtime execution model, in large part due to reflection. Reflection is both an incredibly enabling technology and a damned curse. Source generators, which were new in .NET 5, offer a way to break that formal coupling.

Reflection has at least two challenges. The first is that pervasive use is bad for performance (startup, throughput, and memory). It also makes assembly trimming difficult, which is another dimension of performance. The assembly trimmer—and any software like it—makes decisions statically based on what it can see and trust in assembly metadata. Reflection is inherently late-bound such that its complete operation is not recorded in metadata, which in turn limits the assembly trimmer from doing a great job.

With this new approach, you can write the same high-level serialization code as normal, and then opt into using the source generator, which generates a custom serialization

implementation with static (early bound) code using low-level primitives like Utf8JsonWriter and no reflection.

Zooming out, the System.Text.Json serializer is perhaps the best example of a relatively high-level .NET libraries component that takes advantage of and supports many new features while maintaining and improving performance. Recent examples are: IAsyncEnumerable, records, and nullability. These improvements make the serializer increasingly easier to use and more capable. They also inform these low-level features because the team itself is an important consumer.

Baseline Case

Let's start with the baseline case for using the System.Text.Json serializer. It's important to start with this case to demonstrate how easy it is to switch the new optimized patten.

```
using System.Text.Json;
using System.Text.Json.Serialization;

JsonMessage message = new("Hello, world!");

// baseline case for using JsonSerializer
string json = JsonSerializer.Serialize<JsonMessage>(message);
Console.WriteLine(json);

// Message type
internal record JsonMessage(string Message);
```

This code results in the following output.

```
{"Message":"Hello, world!"}
```

The serializer uses reflection to discover the **Message** property and then to extract its value from the associated getter. That works but it isn't optimal.

Optimized Serialization

The following code uses the source generator and produces much better results because it doesn't use reflection, but uses property accessor calls on JsonMessage and generates the JSON with Utf8JsonWriter directly.

```
// relies on source generation
string optimizedJson = JsonSerializer.Serialize(message,
    JsonContext.Default.JsonMessage);
Console.WriteLine(optimizedJson);

// Source generator definition
[JsonSerializable(typeof(JsonMessage))]
internal partial class JsonContext : JsonSerializerContext
{
}
```

I've shared just the changes to the program. The call to JsonSerializer.Serialize is switched to use a different signature and the partial JsonContext class is new. Otherwise, it's all the same. Note that the JsonContext name is arbitrary. You can choose any name for the class.

The magic is three-part:

- JsonContext is a partial class, which means the source generator can generate **.g.cs** files that fill out the rest of the class.
- The JsonContext class provides a place to hang an attribute that's global to the program (as opposed to a single serialization call) that links a type (in this case JsonMessage) and any serialization options (none of which are provided in this example) to the source-generated code.
- JsonSerializerContext defines and enforces (by virtue of inheritance) the shape that the serializer expects from (in this case) JsonContext.

That's pretty reasonable for a new scheme with so much benefit. You can see that it doesn't require much to adapt existing code. This new model is generally recommended, and is something you should strongly consider for performance-sensitive scenarios that process JSON content.

On the TechEmpower Caching Benchmark, a 40% increase in throughput solely was observed by moving to source generation for JSON serialization. **Table 1** gives you a sense of how much reflection can cost and how much computers love executing static code.

	Requests/sec	Requests
.NET 5	243,000	3,669,151
.NET 6	260,928	3,939,804
.NET 6 + JSON Source generation	364,224	5,499,468

Table 1: TechEmpower Caching Benchmark (with source generation)

Microsoft has also validated that IL trimming is improved when using source generation. In particular, trimming is able to cut the size of System.Text.Json.dll (for self-contained apps) in half. It also makes the assembly trimmer easier to use in more aggressive trimming modes because all code (at least as it relates to System.Text.Json) is statically reachable.

This description has been entirely focused on serialization. Deserialization has also been improved but not to the same degree. For deserialization (and you can do this with serialization, too), you can opt into using source generation to produce a metadata model that can be used at runtime. This is more like having a map, but not the route. Similar support for deserialization as serialization might be added in a future release.

JIT Compiler

Performance has been a big part of every .NET release. Microsoft publishes a post on the .NET Team blog every year on the latest improvements. Everyone is recommended to take a look at the "Performance improvements in .NET 6" post (https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-6/). I'll provide a short summary of some of the performance improvements in the JIT.

Inlining

One of the most effective performance optimization techniques in the just-in-time compiler is inlining. The runtime gets the JIT to compile one method and then calls into another that then needs to be JITed. Method calls are not free, particularly if they are virtual or (worse yet) interface calls (which is common). The JIT can erase method calls by pulling a method body (that would be called) into the current one as inline code. For methods that get called a lot, this performance optimization can help a lot.

The first example in the .NET 6 performance post (https://devblogs.microsoft.com/dotnet/performance-improve-

Method	Runtime	Mean	Ratio	Code Size
Format	.NET 5.0	13.21 ns	1.00	1,649 B
Format	.NET 6.0	10.37 ns	0.78	590 B

Table 2: TryFormat Performance

Method	Runtime	Mean	Ratio	Code Size	Allocated
GetLength	.NET Framework 4.8	6.3495 ns	1.000	106 B	32 B
GetLength	.NET Core 3.1	4.0185 ns	0.628	66 B	–
GetLength	.NET 5.0	0.1223 ns	0.019	27 B	–
GetLength	.NET 6.0	0.0204 ns	0.003	27 B	–

Table 3: Interface dispatch performance

Method	Mean	Code Size
PGO Disabled	1.905 ns	30 B
PGO Enabled	0.7071 ns	105 B

Table 4: Devirtualization performance with PGO

ments-in-net-6/) describes a case where a Utf8Formattter. TryFormat improved significantly in this release without any code changes. Surely that's impossible. A 22% improvement was seen in throughput and a 35% reduction in generated assembly code as seen in **Table 2**.

The Utf8Formattter.TryFormat method has a one-line implementation to the internal TryFormatInt64 method. In .NET 6, that method was marked with the MethodImplOptions.AggressiveInlining attribute, which greatly increases the chance that the method will be inlined. You can think of this attribute as the .NET performance optimization that's responsible for the double-digit improvement to TryFormat and likely other callers.

It gets better. As a result of inlining, the JIT is able to see through the method call and choose to copy the method body in full or in part. In this case, the JIT is able to see and process branches (if and switch statements) in the method implementation and choose to inline just a single method call that would have been the final and only observable result of actually running all the code. That's a huge benefit if this method is called a lot.

The JIT isn't really "running code" but it sure seems like it. It can reason about code and safely skip operations that are unnecessary but provably produce the same results. There are lots of compiler optimizations like this.

Devirtualization

Another big win from inlining is devirtualization, particularly for interfaces. Imagine a method is inlined that takes a collection interface like **IList<T>** or **IEnumerable<T>.** At this point, the code is now specific to the parent method and not subject to being called by arbitrary callers. As a result, the JIT may be able to reliably specialize the code to a single class and type of **T** resulting in much faster direct calls instead of interface dispatch.

Here is an example in the performance post that does this.

```
public int GetLength() => ((ITuple)(5, 6, 7))
   .Length;
```

Table 3 is the **ValueTuple'3** type being called on the **ITuple** interface that it implements.

The JIT inlines and devirtualizes the .Length property call in .NET 5 and .NET 6, respectively.

This improvement is absolutely impressive and demonstrates the value of this style of optimization. However, this particular optimization only applies when a method can be inlined and then specialized based on the narrow use of the code. Methods are generally not inlined (for good reason). As part of .NET 6, Microsoft has developed a completely different technology called dynamic PGO that has the capability to devirtualize any method (non-inlined). That enables much broader performance benefits.

If you have familiarity with devirtualization, you'll know that a code generator needs to be correct when it devirtualizes an interface or other virtual call. If not, the program will have unpredictable results or (more likely) crash because it might specialize, for example, an **ICollection<T>** argument as **List<T>** but then **IList<T>** or **ImmutableArray<T>** is passed in next. Clearly, you shouldn't risk crashing apps to get a performance win.

Dynamic PGO includes a new feature called guarded devirtualization. It's a sort of "zero risk gamble" performance feature. Based on observation, it can see that your code almost always passes List<T> to a method that takes an ICollection<T>. It then generates a fast path for List<T> and then a slow path for any other ICollection<T>. If dynamic PGO is right most of the time, it can provide a significant performance win. If the gamble proves wrong more than it expects, it can skip the preferred devirtualized call and go back to the normal virtualized call for all cases.

Let's see how this feature plays out with IEnumerable<int> with a call to MoveNext(), as captured by the benchmark shown in Table 4.

You can see that PGO results in bigger code size because it requires more machinery to work correctly and safely (the fast and slow paths), but wow! The drop in execution time is worth the price of admission. IEnumerable<T> is a particularly apt example because it's used everywhere.

Dynamic PGO is a fully supported opt-in feature in .NET 6 and worth trying out (by setting the DOTNET_TieredPGO environment variable to "1"). Microsoft plans to enable dynamic PGO by default with .NET 7. It's a very exciting feature with a lot of potential for improving performance.

Closing

.NET 6 is perhaps the most foundational release since .NET Core 1.0. It includes support for major new hardware platforms, broader use of source generation, another jump forward in performance, and tens of features not mentioned in this article. .NET 6 is a good reminder that Microsoft is investing in .NET for the long-term across both client and cloud. If you build cloud or client apps—and particularly if you build both—you've got a lot of strong options with .NET. Looking ahead, what comes next looks even better as new investments come to fruition. Like it's always been, it's a great time to be a .NET developer.

Rich Lander

Bring Your .NET Apps Forward with the .NET Upgrade Assistant

.NET 6 brings many exciting innovations to the .NET ecosystem. Compared to the .NET Framework, .NET 6 is faster and can work cross-platform on Windows, Mac, or Linux. .NET 6 and its successors are the future of .NET and going forward, all new features will come to .NET 6+ rather than to .NET Framework. Because of these benefits, it's valuable to upgrade existing .NET Framework

projects to build and run as .NET 6 apps instead. Transitioning from .NET Framework to .NET 6 can be challenging, though, because of the differences between the platforms, especially for some app models, like Web apps.

To help make upgrading to .NET 6 easier, Microsoft's .NET team has created a tool known as the .NET Upgrade Assistant (sometimes referred to as just Upgrade Assistant). Upgrade Assistant is an open-source command-line tool for automating some of the changes necessary to upgrade to .NET 6 and for highlighting other changes that need to be made manually. Upgrading from .NET Framework to .NET 6 is a complex process and, for many projects, can't be completely automated. Upgrade Assistant doesn't aim to completely upgrade projects automatically; instead, its goal is to automate away the many simple changes needed when upgrading to .NET 6 so that developers can focus on the most interesting changes. By partially automating the upgrade process, the .NET Upgrade Assistant can make adopting .NET 6 quicker. Upgrade Assistant currently supports upgrading class libraries, console apps, ASP.NET MVC, and WebAPI projects, plus WinForms and WPF projects. The tool works with either C# (csproj) or VB (vbproj) projects.

The .NET Upgrade Assistant is developed as an open-source project at https://github.com/dotnet/upgrade-assistant. This article introduces the basics of using the tool, and additional documentation and samples are available on the GitHub page. Users can also submit issues, questions, or pull requests via GitHub.

Installing the .NET Upgrade Assistant

Before installing the .NET Upgrade Assistant, make sure the following prerequisites are installed:

- The .NET SDK. Upgrade Assistant is a .NET SDK command-line tool, so the .NET SDK is required to install and run it. The SDK can be installed from https://dotnet.microsoft.com/download.
- The infrastructure needed to build the projects being upgraded. As part of execution, Upgrade Assistant builds the projects being upgraded. Therefore, any tooling required to build the projects needs to be present on the computer running Upgrade Assistant. At the very least, this means that MSBuild needs to be present and, in many cases, Visual Studio 2019 or above also needs to be installed.

As a .NET SDK tool, Upgrade Assistant can be easily installed and managed from the command line. To install the latest release of the tool from NuGet, run the command **dotnet tool install -g upgrade-assistant.** Similarly, Upgrade Assistant can be updated to the latest version by running **dotnet tool update -g upgrade-assistant.**

Because Upgrade Assistant is installed from NuGet, the installation process considers any NuGet configuration present on the computer. If you see errors about not being able to find Upgrade-Assistant on custom feeds or if you encounter issues related to authentication, it may be because custom NuGet configuration is interfering with the install process. You can work around these issues by ignoring inapplicable NuGet sources: **dotnet tool install -g upgrade-assistant --ignore-failed-sources.**

By default, the latest released version of Upgrade Assistant is installed. New versions are released approximately twice a month as new features become available in the tool. If you'd like to get even more frequent updates, nightly builds can be installed from the Upgrade Assistant team's CI feed. These builds are produced every time the code changes in the GitHub repo and provide immediate access to the latest features. The CI builds are not as well tested, though, so they should only be used until the features you need are incorporated into a regular bi-monthly release to NuGet. Installing from the CI feed can be done with: **dotnet tool install -g upgrade-assistant --add-source https://pkgs.dev.azure.com/dnceng/public/_packaging/dotnet-tools/nuget/v3/index.json.**

Using the .NET Upgrade Assistant

The .NET Upgrade Assistant can be executed by simply running **upgrade-assistant** from a command prompt. When running the tool, you need to specify either the **analyze** command or the **upgrade** command.

The Upgrade Command

The most common command for Upgrade Assistant is **upgrade**. This is the command that walks the user through different upgrade steps and begins getting the input project or solution ready to run on .NET 6. Upgrading with Upgrade Assistant is as simple as running **upgrade-assistant upgrade <Input File>,** where <Input File> is either a project file (csproj or vbproj) targeting an older version of .NET or a solution (sln) containing one or more projects to be upgraded.

Other optional parameters that can be used with the upgrade command include:

- **--target-tfm-support** allows the user to specify which .NET version Upgrade Assistant should upgrade to. They can choose LTS, Current, or Preview. With .NET 6's release, any of these options results in upgrading to .NET 6 (as it's both the most current and the latest LTS release of .NET). Once preview versions of .NET 7 are available, the Preview option upgrades to that target.

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.

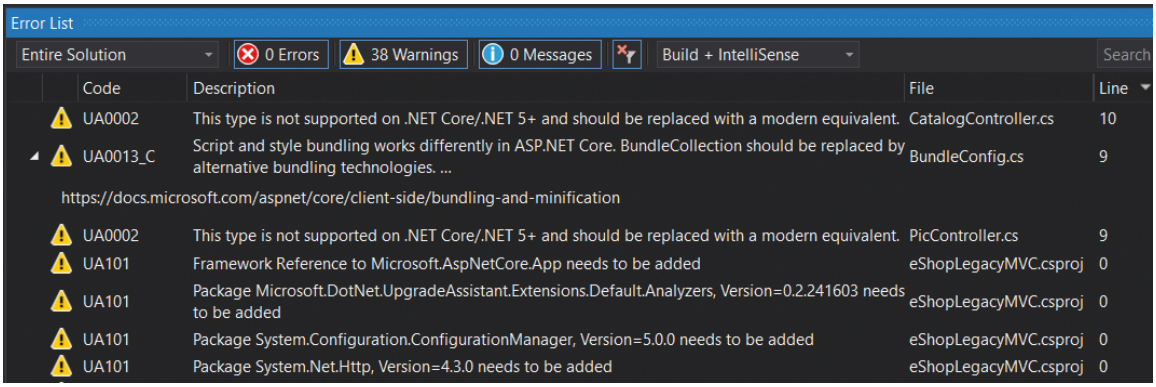

Figure 1: Example .NET Upgrade Assistant analyze output

- **-- vs-path** allows the user to specify which version of Visual Studio to use when loading/building the project in cases where multiple versions of Visual Studio are available.
- **--skip-backup** disables backing up the project before upgrading and can be useful when running on projects that are in source control, as backups may not be needed in that scenario.
- **--verbose** enables more verbose logging but is mostly used for diagnosing problems in the tool as the output includes much more detail and may be less useful for normal use cases than the typical tool output.

Once Upgrade Assistant has started the Upgrade command, it guides the user through applying a series of changes to their project with an interactive command line interface. This upgrade process is explored in more detail in the "Example Walkthrough" section of this article.

The Analyze Command

If you'd like to see what changes the .NET Upgrade Assistant recommends for a given project or solution without changing anything, you can use the **Analyze** command. Like the **Upgrade** command discussed earlier, the Analyze command takes a single required parameter—the path of the solution or project (csproj or vbproj) to analyze. The same optional parameters mentioned above also work for the Analyze command.

Unlike the Upgrade command, the Analyze command isn't interactive. Instead, it evaluates the projects using the same heuristics the Upgrade command uses to identify necessary changes based on the desired version of .NET (LTS, Current, or Preview) and outputs a report summarizing the findings. This process doesn't make any changes to the input projects. Instead, it provides analysis of what changes are required to upgrade the projects.

The output file is in the SARIF format, a well-known JSON-based OASIS standard used by analysis tools. An example output file is shown in **Figure 1**. SARIF files can be opened using extensions for either Visual Studio or VS Code (available from https://marketplace.visualstudio.com), or even in some Web-based viewing apps (https://microsoft.github.io/sarif-web-component, for example).

Example Walkthrough

To give more detail on how the .NET Upgrade Assistant works and what sorts of changes it can help with, let's run the tool on a sample app. For this exercise, I'll run the tool on the

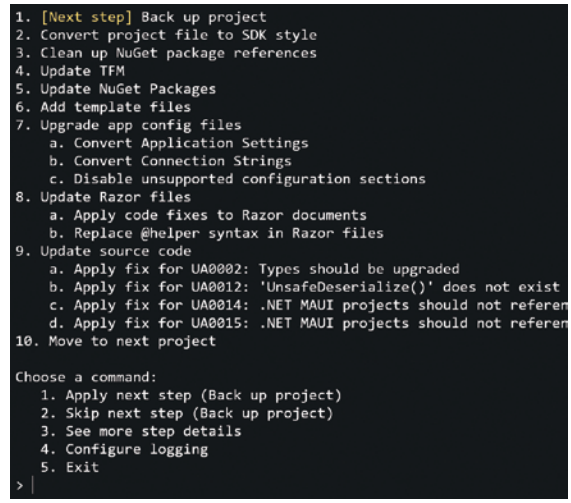

Figure 2: .NET Upgrade Assistant upgrade steps

eShop sample app, a simple ASP.NET MVC e-commerce sample available at https://github.com/dotnet-architecture/eShopModernizing/tree/master/eShopLegacyMVCSolution.

Because I want to upgrade the Web app to ASP.NET Core, I'll use Upgrade Assistant's **upgrade** command. From the eShopLegacyMVCSolution folder, execute this command: **upgrade-assistant upgrade eShopLegacyMVC.sln.**

Upgrade Assistant can run either on solutions (sln files) or projects (csproj or vbproj files). In the case of a solution with multiple projects, Upgrade Assistant asks the user which project is the solution's entry point. The entry point project is the top-level project the user wishes to upgrade. Typically, this is the project that's executed—usually an exe, Web app, or test project. Once Upgrade Assistant knows the solution's entry point, it analyzes project-to-project dependencies to determine which projects need to be upgraded to support ultimately upgrading the entry point project and will recommend in which order the projects should be upgraded (for example, beginning with moving the lower-level dependent projects first). It then walks the user through upgrading each of those projects in turn.

In the case of the eShop sample, the solution only contains a single project. So, even though I ran the tool on the eShop solution, Upgrade Assistant recognizes that there's only one project to upgrade and jumps directly to project-level upgrade steps, as shown in **Figure 2**. For each upgrade

step, the user can choose a command from a list of options. Typically, users choose the first action (**apply**) to apply the upgrade step to the project. In some cases, they may choose to learn more details about the step first, or to skip it. Commands are chosen by entering their corresponding number and pressing enter. As a shortcut, pressing enter without specifying a number executes the first command in the list.

Project Backup

The first step of the upgrade is to back the project up by copying it (and its source files) to another location. This is important because, by design, Upgrade Assistant leaves many projects in a non-building state after it's finished executing (because it applies **some** of the changes needed to upgrade to .NET 6 but others need to be applied manually). Backing up the project makes it possible for users to easily roll back to a version that successfully builds and works, in case they abandon the upgrade.

Project File Format Update

After backing up the project, the next step Upgrade Assistant applies is updating the project file to use the SDK style project file format. SDK style projects are more concise and more human-readable. They use globbing patterns and automatically include common source files. They also use the <PackageReference> format for referencing NuGet packages (instead of a packages.config file).

After applying this step, you can look at the eShopLegacyMVC.csproj file to see that it's considerably simpler and that package references have been updated to use the <PackageReference> format. Notice that the project still targets .NET Framework 4.7.2. Although the new project file format is required for .NET 6, it can also be used with older .NET Framework targets. Some users find it useful to run the first few steps of Upgrade Assistant to update to newer csproj and package reference formats without updating the .NET platform they're building against. In order to enable that scenario, the retargeting to .NET 6 occurs later in the upgrade process in a separate step.

NuGet Package Cleanup

The next upgrade step, after updating the project file format, is cleaning up NuGet package references. This is necessary because of differences between the old packages.config method of referencing NuGet packages and <PackageReference> references. A packages.config file lists all NuGet packages needed by a project regardless of whether those packages are referenced directly from the project or if they're only needed transitively by the project's dependencies. When using the <PacakgeReference> format for references, though, only top-level direct references need to be listed. When NuGet packages are restored, the NuGet client automatically finds other packages needed by the project's direct dependencies and downloads them also.

When Upgrade Assistant converted from the old-style project file to the new one, it copied over all NuGet references. During this step, Upgrade Assistant cleans up NuGet package references by removing those that are referenced transitively by other package references. It also performs other low-risk cleanup, such as replacing some assembly references with equivalent NuGet package references, which work for both .NET Framework and .NET 6.

After applying this step, you'll notice that the list of NuGet packages referenced from the eShop project is reduced to only those that're used directly by eShop. Applying this step to the eShop project also replaces the Framework reference to System.Configration with a package reference to System.Configuration.ConfigurationManager.

Update TFM

The next upgrade step is the one that retargets to .NET 6 by changing the target framework moniker (TFM) used in the project's <TargetFramework> property. Up until now, the project still targeted .NET Framework. The changes were only about modernizing the project file and NuGet package references. For non-Web scenarios, it's possible to exit the tool prior to applying the TFM change and have a project that still builds and works as before, except that the project file is updated to SDK-style.

Once the update TFM step is applied, the project is re-targeted to a new .NET platform. The specific target framework to which Upgrade Assistant chooses to upgrade a project depends on a set of heuristics that consider input parameters and characteristics of the project itself. For library projects, Upgrade Assistant attempts to retarget to .NET Standard 2.0 because that TFM makes it possible to use the library from the widest variety of consumers. In cases where the project can't target .NET Standard (because it's an exe or Web app, for example), Upgrade Assistant targets .NET 6. This step also considers dependencies because a project's TFM must be at least as restrictive as the targets of its dependencies. Because of this, Upgrade Assistant may choose a more specific TFM, like .NET 6.0 (represented by the TFM short name "net6.0" in the project file) or even an OS-specific TFM like **net6.0-windows** if that's necessary based on the dependencies the project has.

Console output from the TFM upgrade step explains which TFM will be used and why. In the case of the eShop sample, logging explains that the .NET 6 TFM will be used because the project builds to an executable (a Web app), so .NET Standard can't be used.

After applying this step, you can see that the <TargetFramework> property of the project file has been updated to net6.0 (representing .NET 6.0 as the target).

> The second NuGet update step is about updating NuGet package references based on the updated TFM.

Update NuGet Packages

The next step is a second round of updating NuGet package dependencies. Whereas the first NuGet package update step was about cleaning up package references after converting to the new project file format, this second NuGet update step is about updating NuGet package references based on the updated TFM. Some of the NuGet packages referenced by the project may not work with the new target. This upgrade step queries NuGet (and any additional NuGet sources configured for the project) to look for updated packages that work with the new TFM. The step also considers manually configured mappings of some NuGet packages to known .NET 6-compatible equivalents.

In the case of the eShop sample, logging tells us that this step replaces EntityFramework version 6.2 with version 6.4 because the latter version works with .NET 6. It also removes some packages like Antlr, WebGrease, and Autofac.Mvc5 that were useful in ASP.NET scenarios but are unnecessary for ASP.NET Core.

There are also some warnings in the console output for this step. These warnings occur if a NuGet package is referenced that doesn't support the new TFM and no newer version of the package that will work with the updated target can be found. In these cases, developers need to manually update the packages to more modern equivalents or restructure the project to not have these dependencies. For eShop, logging informs us that the Microsoft.ApplicationInsights.Web package reference could not be automatically updated and should be manually reviewed.

Add Template Files

Upgrade Assistant's next step is to add some additional files to the project that will be useful after upgrade. For Web scenarios like the eShop sample, Upgrade Assistant adds common files that are needed in ASP.NET Core apps. By using the **See more step details** command, you can see more information about the files to be added. They include: program.cs, startup.cs, appsettings.json, and appsettings.development.json. All of those are files that are expected to be part of an ASP.NET Core app but weren't present in the ASP.NET version of eShop. The files that are added provide a starting point for important ASP.NET Core components in the upgraded app and include comments explaining to the user what the files are for and where to go to learn more about them. This step is also useful because adding these files to the project makes it possible for future upgrade steps to further update them based on other project details, as will be shown in future steps.

If any of these files already exist in the project, Upgrade Assistant analyzes them to determine whether the files already present are likely to fill the same role as the template files and, if they were, skips adding the templates. If the files looked like different files that just shared a common name, however, the existing file is renamed and the new file added. After applying this step to the eShop sample, you'll see the four new files listed previously added to the project.

Update Config Files

ASP.NET Core apps don't use web.config files. The next step—upgrading config files—is all about migrating important data that would have existed in web.config or app.config files into new homes that are appropriate for .NET 6. This step has a number of sub-steps to perform different tasks with the project's web.config file:

- The first sub-step moves application settings from web.config into appsettings.json. After applying this to the eShop sample, you'll see that settings like UseCustomizationData are added to appsettings.json based on data in web.config.
- The next sub-step is similar but moves connection strings rather than app settings. Again, when run on eShop, you'll notice that the app's database connection string moved into appsettings.json.
- The next sub-step disables web.config sections that are no longer supported on .NET 6, such as System.ServiceModel or System.Diagnostics. In the case of the eShop sample, none of these sections are present in the config file, so the step is marked as complete even before you get to it, as no work is necessary.
- Finally, the system.web.webPages.razor/pages/namespaces conversion sub-step finds any namespaces that are automatically imported into Razor pages and moves those imports into a _ViewImports.cshtml file, which is the correct way to auto-import namespaces in ASP.NET Core. After running this step, notice the new _ViewImports.cshtml file added to the project's Views folder (shown in **Figure 3**).

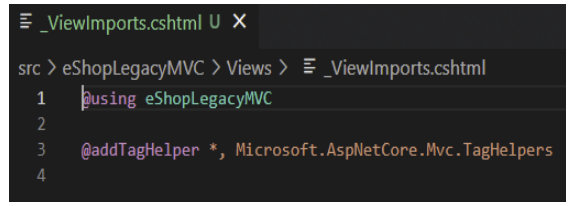

Figure 3: Auto-generated _ViewImports file with migrated import statement

Update Razor Files

The next upgrade step fixes up Razor (cshtml) files to work after the upgrade. It has two sub-steps. The first sub-step applies source-level fixes to C# code embedded in Razor files. The rules used to update the C# code are the same as are applied to .cs files in the next step. The second sub-step replaces usage of the @helper function syntax in Razor files with local methods because @helper isn't supported in ASP.NET Core.

The eShop sample doesn't use @helpers, so the second sub-step is already marked complete when you get to the Razor file update step. There are code fixes to be applied, however. Applying that step updates the _Layout.cshtml file in the eShop project to include a **@using Microsoft.AspNetCore.Http** import and replaces HttpContext.Current usage with HttpContextHelper.Current. HttpContextHelper is a class that Upgrade Assistant can insert into projects as a way to deal with HttpContext.Current usage in Web apps. HttpContext.Current was a commonly used static property in ASP.NET that returned the HTTP context for the current request. In ASP.NET Core, these sorts of global statics aren't used. The HttpContextHelper class is added to upgraded projects to act as a bridge that provides the ability to statically request the current HTTP context until call sites can be updated to not use that pattern.

Update Source Files

The final upgrade step before moving on to the next project (if there is one) is to update C# source files to fix up code patterns that won't work on .NET 6. This step has many sub-steps, one for each rule that will be applied. As with some of the previous steps, any sub-steps that are unnecessary (because the project doesn't contain the relevant code pattern) are marked as complete. You may notice that the list of sub-steps for this step (shown in **Figure 4**) isn't the same as the sub-steps that were listed here when Upgrade Assistant first started up. The reason for the difference is that some of the source updater's sub-steps are conditional on the type of project being upgraded and, prior to the project being converted to an ASP.NET Core project, some of the sub-steps didn't apply.

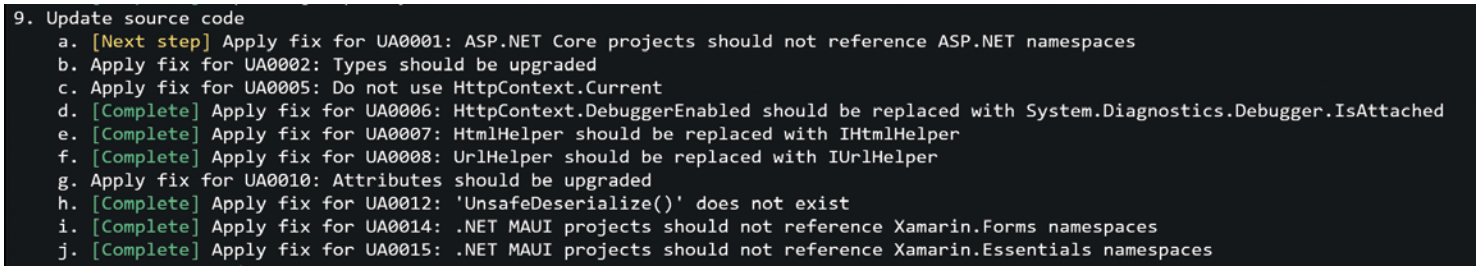

Figure 4: Source update steps for the eShop project

Figure 5: Output from the source updater step details changes made

Over time, the list of code fixes that are applied by the source code update step will grow. In the case of the eShop sample, the source update rules that are applied are:

- Removing imports for System.Web namespaces
- Replacing common ASP.NET Web types (like Controller or ActionResult) with equivalents that exist in ASP.NET Core in different namespaces
- Replacing HttpContext.Current usage with HttpContextHelper.Current, as discussed in the Update Razor Views section of this article
- Updating or removing ASP.NET attributes such as [Authorize] (which exists in a different namespace) or [AllowHtml] (which is no longer needed and can be removed)

Each of the code fixes is applied in turn and logs which files were updated so that users understand what code was changed in their project, as shown in **Figure 5**.

Finally, after all code fixes are applied, the source update step notifies the user (via warnings in the console) if there are any other code patterns found that couldn't be automatically addressed and require manual fix-up. In the case of eShop, after applying all the code fixes from this step, Upgrade Assistant informs you that manual changes are needed to address the project's use of bundling APIs in BundleConfig.cs that aren't supported on ASP.NET Core.

> Manual updates are still needed to get the project building and working, but many of the initial changes have been taken care of automatically by Upgrade Assistant.

After applying the source update step, upgrade of the eShop project is complete. There are still manual updates needed to get the project building and working, but many of the initial changes have been taken care of automatically by Upgrade Assistant and you've been alerted to some of the other changes that will be needed as follow-ups. Upgrade Assistant also adds a reference to a set of Roslyn Analyzers (Microsoft.DotNet.UpgradeAssistant.Extensions.Default.Analyzers) that causes any of the source issues identified by the source update step (including those like the bundling warning that couldn't be fixed automatically) to show up as warnings in the project. This makes it easy to identify next steps in the upgrade process when working with the project afterwards.

Advanced Usage

In addition to the basic workflow discussed above, there are a few other commands and command line options that can be useful for customizing the .NET Upgrade Assistant's behavior.

Automating Upgrade Assistant

By default, Upgrade Assistant runs in an interactive mode that requires the user to step through upgrade steps individually. It's important that users understand the changes Upgrade Assistant is making during the upgrade process because it's likely that the project won't build after the tool finishes and the user will need to understand what state the project's in and how to complete the upgrade manually.

After using Upgrade Assistant for a while, though, users may become familiar with the upgrade process and want the tool to run more autonomously, especially if they will be running the tool on many different solutions. To enable this, the **--non-interactive** option can be passed to the upgrade command to cause Upgrade Assistant to automatically apply all upgrade steps without pausing for user interaction. In the case of solution files with many projects, this can be combined with the **--entry-point** option to specify which project is the entry point when launching Upgrade Assistant. The entry point option can be specified multiple times or use wildcards to select multiple entry points.

Using Extensions

The .NET Upgrade Assistant includes first-class support for extensions so that users can fine-tune its behavior to meet their needs. Extensions can modify the behavior of existing upgrade steps or even add their own completely custom steps. There are a couple of ways to use extensions. To quickly use an extension, the **--extension** option can be used when running Upgrade Assistant to specify an extension to use. A more robust way to use extensions, though, is through the Upgrade Assistant extensions command.

The extensions command allows users to install extensions for use with specific projects. Running **upgrade-assistant extensions add <name>** installs the specified extension for use when Upgrade Assistant is run in the current directory. Extensions installed using this command are installed as NuGet packages. By default, Upgrade Assistant queries NuGet.org for the latest version of extension packages that are added. Optional **--source** and **--version** arguments can be used to indicate the NuGet source to use and the specific version of an extension to install.

Installed extensions are stored in a JSON file (**upgrade-assistant.json**) that can be checked into source control to ensure that everyone working with the project is using the same extensions (and the same versions of extensions) when running Upgrade Assistant.

Other useful extensions commands include:

- **Upgrade-assistant extensions list:** Lists currently installed extensions
- **Upgrade-assistant remove <name>:** Uninstalls the specified extension
- **Upgrade-assistant update <name>:** Updates the specified extension to the latest version
- **Upgrade-assistant restore:** Downloads all extensions listed in upgrade-assistant.json, which is useful if the extensions were originally installed on a different computer and the upgrade-assistant.json file has been shared.

Authoring Extensions

Besides using the .NET Upgrade Assistant's default steps or extensions published by others, developers and organizations may want to create their own Upgrade Assistant extensions to modify the tool's behavior.

A deep dive into how to create Upgrade Assistant extensions is out of scope for this article, but all the details are available at https://github.com/dotnet/upgrade-assistant/blob/main/docs/extensibility.md and several sample extensions can be found at https://github.com/dotnet/upgrade-assistant/tree/main/samples.

At a high level, the key components of an extension are:

- A manifest file (called ExtensionManifest.json) that lists all the files and config settings that comprise the extension.
- Assemblies (called **extension service providers**) that contain types implementing the **Microsoft.DotNet.UpgradeAssistant.Extensions.IExtensionServiceProvider** interface. This interface provides a mechanism for extensions to register types with Upgrade Assistant's dependency injection container.
    - The IExtensionServiceProvider interface and many other Upgrade Assistant APIs useful to extension authors are available in the Microsoft.DotNet.UpgradeAssistant.Abstractions NuGet package.

All services in Upgrade Assistant are resolved from its dependency injection container, so by using an extension service provider to register additional types, you can change the tool's behavior. Common types to register in the DI container include:

- Types derived from **Microsoft.DotNet.UpgradeAssistant.UpgradeStep**. These are the upgrade steps that show up in the Upgrade Assistants list of steps when the tool executes.
- Roslyn analyzers and code fix providers. Upgrade Assistant's source updater step uses these types to automatically address source-level changes (if both an analyzer and code fix provider are available) or to alert the user to a change that needs to be made manually (if only an analyzer is available).
- Implementations of **Microsoft.DotNet.UpgradeAssistant.IUpdater<T>** to be used by Upgrade Assistant steps, like the config updater or Razor file updater to modify their respective file types.

By implementing these types and creating associated configuration, extensions can either customize the behavior of the existing upgrade steps or add their own new ones.

Road Map

As Upgrade Assistant continues to evolve, additional features will be added based on customer input. Be sure to provide feedback through the GitHub project (https://github.com/dotnet/upgrade-assistant/issues) to weigh in on the features you'd like to see added. Once you've tested the tool out, you can also provide feedback directly to the Upgrade Assistant team through a survey at https://aka.ms/DotNetUASurvey.

Some of the areas the Upgrade Assistant team is already exploring for future investment are listed in the tool's roadmap document (https://github.com/dotnet/upgrade-assistant/blob/main/docs/roadmap.md). These areas include possible investments in:

- Additional analyzers and code fix providers to expand the set of source updates the tool can make
- Support for multi-targeting
- Support for migrating GC configuration
- Additional reporting options
- Support for WebForms projects
- Graphical user interface on top of the existing CLI
- Support for applying some upgrade steps (such as NuGet cleanup) to an entire solution at once rather than one project at a time

Wrap-Up

Upgrading projects from .NET Framework to .NET 6 is a non-trivial task and one that will probably never be completely automated. Hopefully the .NET Upgrade Assistant will make the process more approachable by providing guidance to users and by automating away some of the simpler but more numerous changes required so that developers can focus on the more interesting and challenging parts of the upgrade.

For additional information about the .NET Upgrade Assistant, you can look at the project's GitHub repository (https://github.com/dotnet/upgrade-assistant) and docs (https://docs.microsoft.com/dotnet/core/porting/upgrade-assistant-overview). Additional guidance on upgrading, in general, is available both in documentation (https://docs.microsoft.com/dotnet/core/porting) and in an ebook focused on ASP.NET to ASP.NET Core upgrades (https://docs.microsoft.com/dotnet/architecture/porting-existing-aspnet-apps).

Mike Rousos

Visual Studio 2022 Productivity

To improve your developing productivity, Microsoft made Visual Studio 2022 more intelligent, more approachable, and lighter weight. For the first time ever, Visual Studio is 64-bit. With enhanced speed, combined with AI coding assistance, ever expanding productivity tools, and streamlining team collaboration, this new version of Visual Studio has you set up for success.

In this article, I'll cover my favorite productivity enhancements in Visual Studio 2022 that will make your workflow more efficient.

Performance

Improving performance is always a top customer request. In Visual Studio 2022, Microsoft made significant progress, including making Visual Studio 64-bit and reducing the time it takes for several operations. Here are some of the immediate benefits:

- Faster search
- Faster incremental build
- Faster test execution
- Faster frameworks

Now that it's a 64-bit application, the devenv.exe process is no longer limited to ~4 GB of memory. For users with large or complex solutions, this is a game changer. You'll have more memory and avoid out of memory exceptions for every aspect of daily development: opening, editing, searching, running, and debugging. This move to 64-bit doesn't change the types or bitness of the applications you develop with Visual Studio. You can still build 32-bit apps with 64-bit Visual Studio with all the performance benefits of working with more memory available.

Faster Search

A performant search is key to a fast and productive developer inner loop. In Visual Studio 2022, searching for file names is faster. You'll see improvements, especially after the initial solution load, because Visual Studio can preserve more context in between opening and closing a solution. Go To (**Ctrl+T**), Visual Studio Search (**Ctrl+Q**), and Find in Files (**Ctrl+Shift+F**) have improved significantly. Additionally, Solution Explorer search is now 50% faster when tested on popular open-source repositories such as Orchard Core.

Faster Incremental Build

Incremental build allows you to avoid the extra overhead of rebuilding components that are still up to date despite all recent changes. This saves a massive amount of time and resources, so it's important that this system is as accurate and performant as possible. Microsoft improved the "Fast up to date" check to better detect file changes for .NET and .NET Core projects. This performance improvement speeds up any feature that depends on build execution like debugging and unit testing.

Faster Test Execution

With the introduction of Hot Reload, Microsoft has reduced the time it takes to execute tests in C# projects targeting .NET 6.0 or later. Additionally, by further optimizing Live Unit Testing start up processes, the time to start up Live Unit Testing is reduced. A 30% improvement was observed when tested on popular open-source repositories, such as Orchard Core.

Faster Frameworks

I would be remiss to cover Visual Studio performance without mentioning the amazing gains seen at the platform level with each new framework version. particularly regarding .NET 5.0 and .NET 6.0. Any app or developer tool using the latest major frameworks is going to instantly see performance gains in even the most fundamental operations. There's a multiplicative effect with any performance optimizations in the JIT (just-in-time compiler), the garbage collector, threading, types in the System namespace, etc. Visual Studio is one of the developer tools that gets to reap the rewards as features update to target the latest versions.

If you want an in-depth look at the .NET performance improvements (including benchmarks down to the nanosecond!), check out https://aka.ms/dotnetperf6.

Personal and Team Productivity

One of the first things developers do in Visual Studio is personalize it. Making your environment comfortable to work in is an essential part of getting into the zone. Both customizing your IDE and syncing your personal settings across devices have some new capabilities:

- Organize your workspace using color
- Match the Visual Studio theme to the Windows theme
- Improved theming flexibility

Visual Studio now has the capability to organize tabs by color, so you don't have to search for an open file. Your active document can be bolded so it's easier to find. You can also customize your tab width and appearance to suit your workflow. To find the new document management settings, navigate to **Tools > Options > Tabs & Windows,** as shown in **Figure 1**.

When there's little background light, shifting your theme to Night mode could improve reading. You now have the capa-

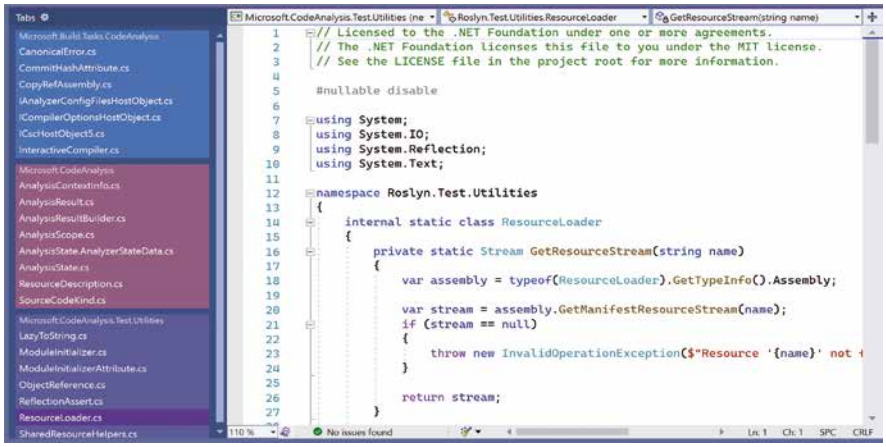

Figure 1: Personalize tabs

Mika Dumont
@mika_dumont

Mika is a Program Manager at Microsoft on the .NET and Visual Studio team. Her main focus is helping .NET developers be more productive in Visual Studio.

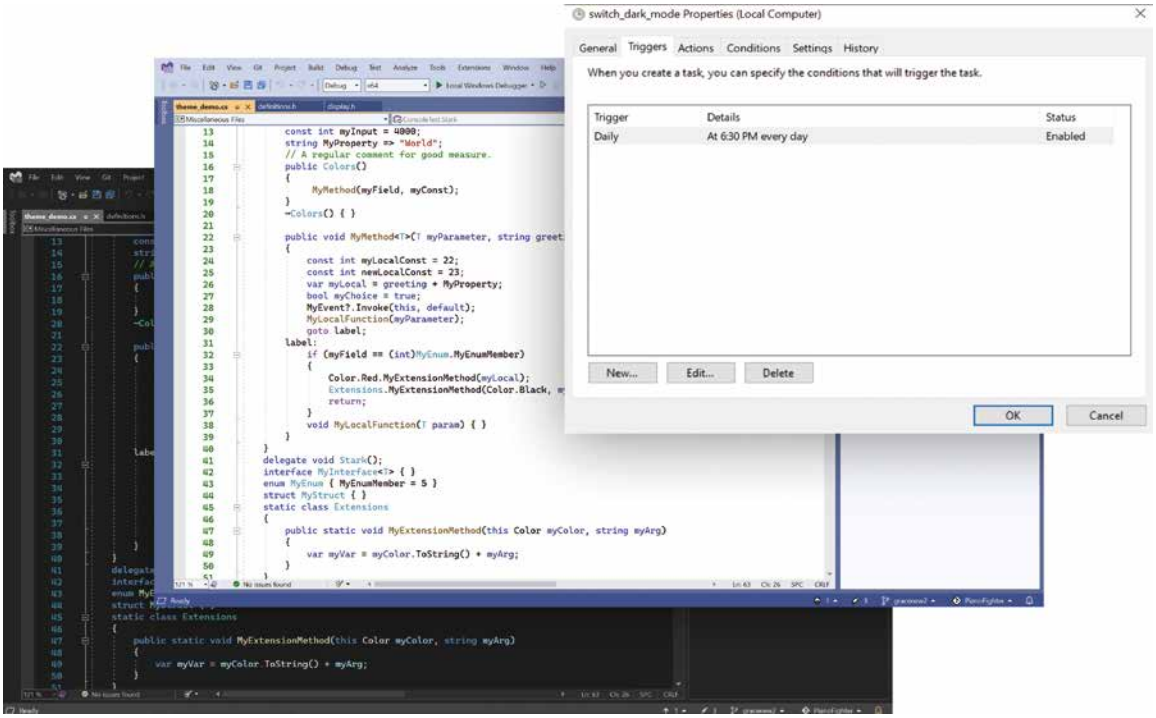

Figure 2: Use system-setting color themes

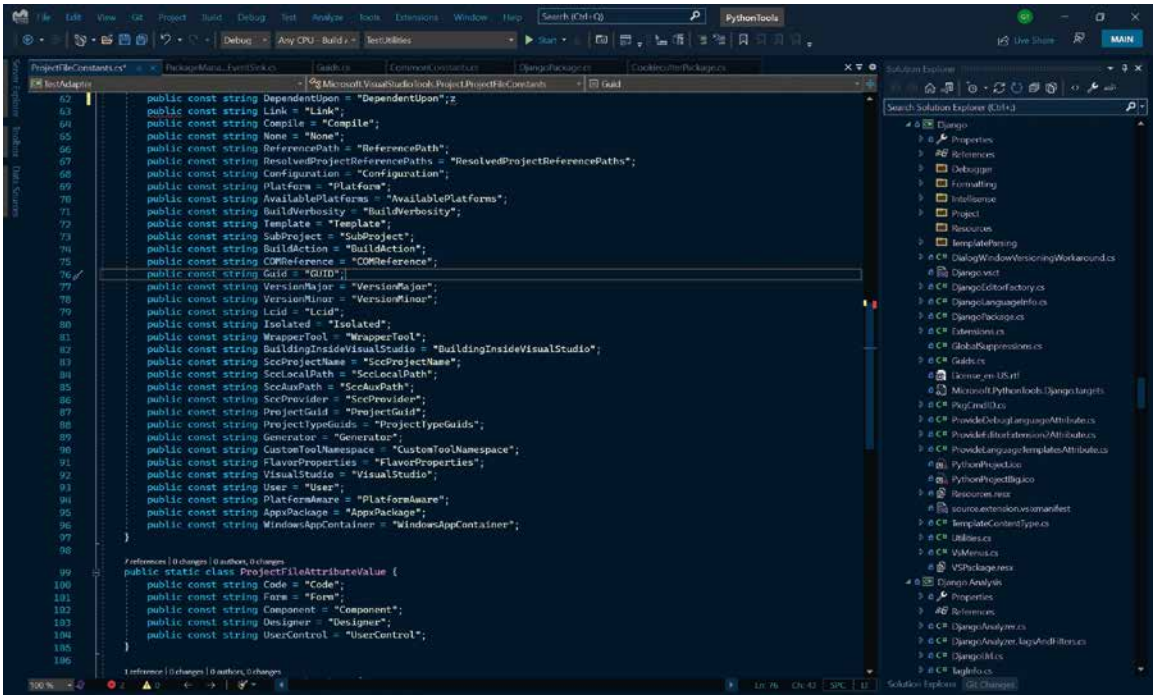

Figure 3: More themes

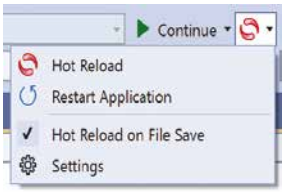

Figure 4: Hot Reload button and menu options

bility to match your Visual Studio theme to your Windows theme. For example, if you have a Dark theme enabled for Windows, you'll see a default Dark theme for Visual Studio 2022. You can enable this in **Tools > Options > Environment > General** and from the **Color Theme** drop-down, select **Use system setting,** as shown in **Figure 2**.

The journey is to expand the theming capabilities and make Visual Studio more inclusive. Microsoft has teamed up with the community of theme authors to convert a selection of Visual Studio Code themes to work in Visual Studio. To install a custom theme, visit the Visual Studio Marketplace, as shown in **Figure 3**.

Hot Reload

For many developers, one of the most time-consuming operations during the development and debugging phase is rebuilding and restarting the app in order to test a code change or complete a behavior that isn't yet implemented. The new Hot

Reload experience in Visual Studio 2022 reduces the number of restarts required by allowing you to modify your .NET application's source code while it's running. Unlike the existing Edit and Continue experience, with Hot Reload, you don't need to hit a breakpoint or pause your application to apply changes.

To use Hot Reload, simply make a supported change and use the new "Hot Reload" button to apply the changes to your running app. The next time the code is executed, the updated logic will be used, reducing the need for many restart cycles.

An option was also added for developers who prefer to use the Save operation to apply Hot Reload changes without the need to explicitly click the Hot Reload button. You can do so by expanding the Hot Reload menu and opting into this behavior by selecting "Hot Reload on File Save," as shown in **Figure 4.**

Hot Reload is available for developers who build apps powered by both .NET Framework and .NET Core for many types of apps, such as WPF, WinUI 3, Windows Forms, ASP.NET Core (for code-behind code changes), Console Apps, and even project types such as Azure Functions. Anywhere a modern supported version of .NET is available, Hot Reload is provided while under the debugger.

In addition, those developers who upgrade to .NET 6 get additional benefits, including powerful new features only available to the latest version of .NET, such as:

- Support for using Hot Reload when not using the debugger, such as launching your app in Visual Studio through the CTRL-F5 mechanism or using the .NET CLI **dotnet watch** tool
- Support for using Hot Reload with Razor files in both ASP.NET Core and Blazor projects
- Support for using Hot Reload with .NET MAUI apps across WinUI, iOS, and Android runtimes for both regular XAML-only projects and hybrid Blazor apps.

Hot Reload also works alongside other debugger experiences. This includes Edit and Continue for editing code during a breakpoint, as well as features that focus on the look and feel of an app while also changing code, such as XAML Hot Reload and CSS Hot Reload.

Finally, please be aware that some edits are not supported and when this situation is encountered, a "rude edit" dialog is displayed. Restarting the app at this point will be needed to apply your changes so you can move forward. This is just the start of the journey and in future releases of Visual Studio and .NET, Microsoft will be working to reduce the type of edits that aren't supported.

The New Razor Editor

In Visual Studio 2022, a new Razor editor was added for local development with MVC, Razor Pages, and Blazor. The new Razor editor has a ton of new tooling support surpassing the functionality of the old Razor editor. For instance, there's now a ton of C# code fixes and refactorings, improved syntax coloring, Go to Definition support, and item filtering in the IntelliSense completion list.

The design of the new Razor editor makes it much easier to enable C# code fixes and refactorings, and many more will be enabled in future releases. Here are just a few of my favorite code fixes and refactorings available in Razor files.

- Add missing Using directives, as shown in **Figure 5**.
- Renaming support, as shown in **Figure 6**.
- Razor syntax highlighting
- Improved colorization options

Razor syntax highlighting has also been updated to improve contrast, general look and feel, and usability. For example, you may notice that the C# background highlighting has been removed. This update improves contrast and makes it clearer when something has been selected or highlighted. The colorization options in **Tools > Options > Fonts and Colors** has been updated to be more descriptive and customizable.

Navigation and Code Exploration

Navigating and exploring code is an integral part of developer productivity. In Visual Studio 2022, Value Tracking has been added, allowing you to perform data flow analysis on your code to help you quickly determine how certain values might have passed at a given point. Value Tracking is available on any member in the context (right-click) menu by selecting the Track Value Source command. The Track Value Source command opens the Value Tracking window allowing you to analyze results, as shown in **Figure 7.**

It's also easier for you to visually inspect and navigate the inheritance chain with the new Inheritance Margin icons, as shown in **Figure 8**. The Inheritance Margin icons are located in the margins representing your code's implementations and overrides. Clicking on the Inheritance Margin icon displays inheritance options that you can select to navigate to. If you find Inheritance Margin distracting, you can disable them in **Tools > Options > Text Editor > [C# or Basic] > Advanced** and deselect Show inheritance margin.

Figure 5: Add missing Using directives

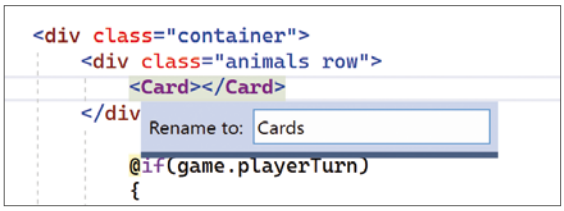

Figure 6: Renaming support for Blazor components

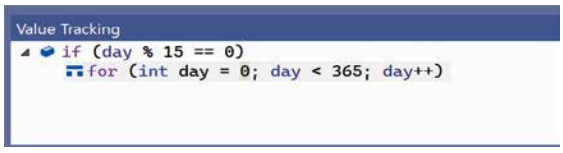

Figure 7: Value Tracking window showing changes to the "day" variable

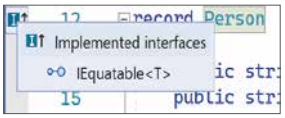

Figure 8: Inheritance Margin

```
public static void SimplifyLinq()
{
    var list = new List<string>();
    if (list.Where(x => x != null).Any());

}
```

Figure 9: Inline hints for C# and Visual Basic files

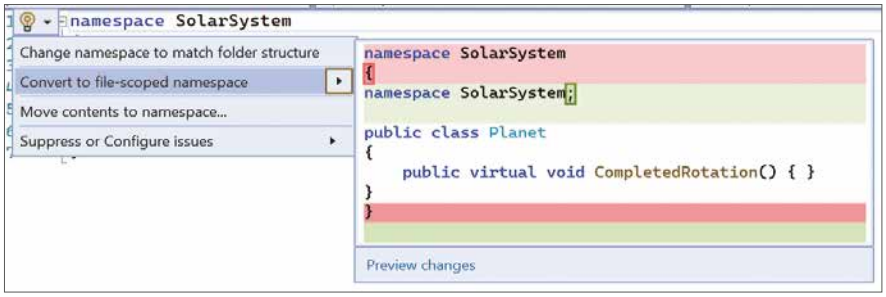

Figure 10: Convert namespace to the new C# 10.0 file-scoped namespace

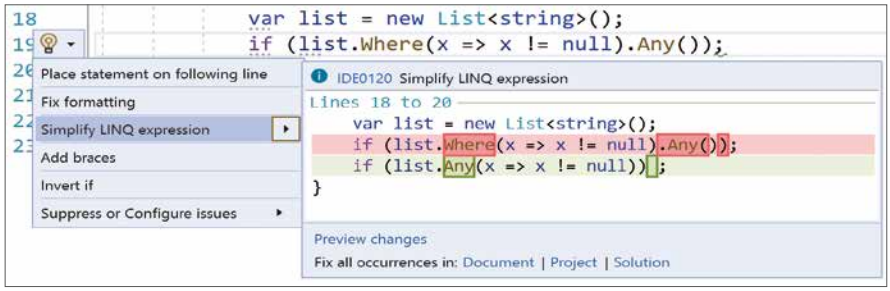

Figure 11: Simplify LINQ expressions to remove the unnecessary call to the Enumerable for the .Where() method to help improve performance and readability

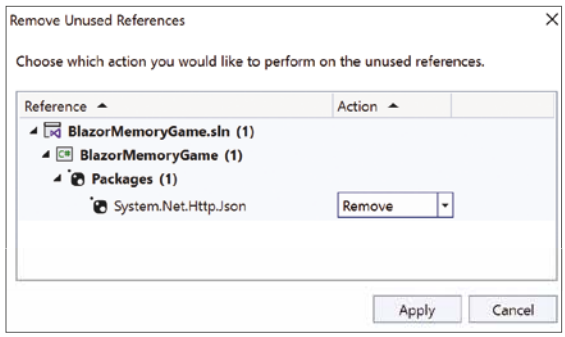

Figure 12: Remove Unused References window

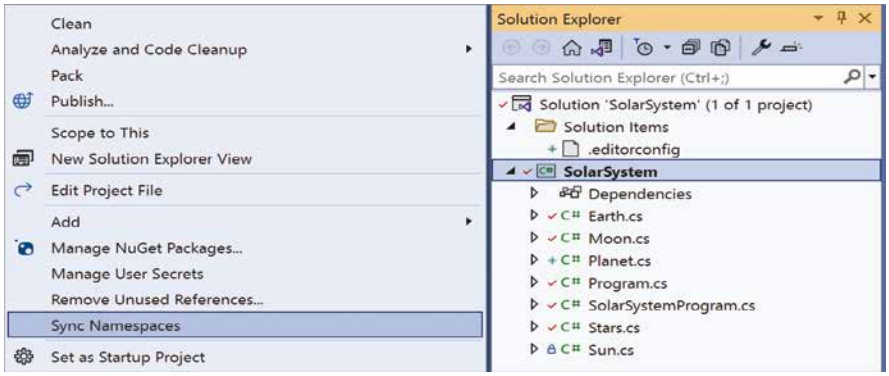

Figure 13: Sync Namespaces

A highly anticipated feature is inline hints. Inline hints display inline parameter name hints for literals, casted literals, and object instantiations prior to each argument in function calls, and inline type hints for variables with inferred types, lambda parameter types, and implicit object creation. To enable and customize inline hints go to **Tools > Options > Text Editor > [C# or Basic] > Advanced** and select **Inline Hints**. The inline hints then appear in C# or Visual Basic files, as shown in **Figure 9**.

Code Fixes and Refactorings

Visual Studio provides hints to help you maintain and modify your code in the form of code fixes and refactorings. These appear as lightbulbs and screwdrivers next to your code or in the margin. The hints can resolve warnings and errors as well as provide suggestions. You can open these suggestions by typing (**Ctrl+.**) or by clicking on the lightbulb or screwdriver icons.

You can check out the most popular refactorings that are built into Visual Studio at https://aka.ms/refactor. A bunch of new code fixes and refactorings have been added to Visual Studio 2022. Here are some of my favorites:

- Convert namespace to the new C# 10.0 file-scoped namespace, as shown in **Figure 10**.
- For improved performance and readability, simplify LINQ expressions to remove the unnecessary call to the Enumerable for the .Where() method, as shown in **Figure 11**.
- Extract base class
- Add explicit cast
- Extract local function
- Inline method
- Convert between verbatim string and regular string
- Generate comparison operators
- Use pattern matching
- Simplify conditional expression

You can also invoke code fixes and refactorings from the Solution Explorer (right-click) menu. One of our highly anticipated refactorings is **Remove Unused References**, which allows you to clean up project references and NuGet packages that have no usage. The Remove Unused References command is available in the (right-click) menu of a project name or dependencies node in Solution Explorer, as shown in **Figure 12**. Selecting Remove Unused References opens a dialog where you can view all references that will be removed, with an option to preserve the ones that you want to keep.

Another refactoring available in the Solution Explorer (right-click) menu is **Sync Namespaces**. Sync Namespaces allows you to synchronize namespaces with your folder structure. The Sync Namespaces command is available in the (right-click) menu of a project or folder in Solution Explorer, as shown in **Figure 13**. Selecting Sync Namespaces automatically synchronizes namespaces to match your folder structure.

.NET Analyzers

Code fixes and refactorings are powered by analyzers. An analyzer is a tool that inspects your code and reports diagnostics and errors. The .NET Compiler Platform (Roslyn) has several recommended analyzers that give you verbose feedback on code quality and code style. Starting in .NET 5.0, these analyzers are included with the .NET SDK and are enabled, by default, for projects that target .NET 5.0 or later. If you'd like to learn more about .NET analyzers, visit https://aka.ms/dotnetanalyzers.

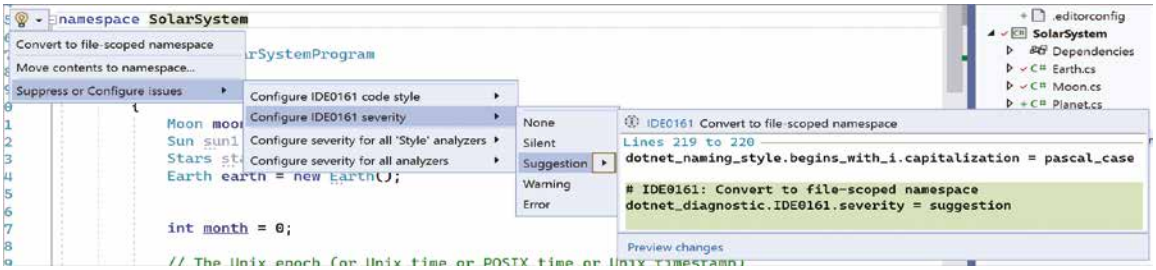

Figure 14: Configure code style severity for the new C# 10.0 file-scoped namespace

If you want a specific analyzer that you don't see included in the .NET SDK, you can create your own analyzer with the open source Roslyn APIs. Creating your own analyzer allows you to create a diagnostic and code fix for a scenario that's special to your code base. You can then share it with your team or anyone who depends on your library. For an example tutorial, visit https://aka.ms/diy-analyzer.

Code Style Enforcement

Enforcing consistent code style is important as developer teams and their code bases grow. Visual Studio allows you to configure analyzers to apply your preferred code style rules and customize the severity at which they appear in the editor, as shown in **Figure 14**. You can easily change your code style severity to display the rule violation as a suggestion, warning, or error in the editor.

You can configure code styles in the code style pages in **Tools > Options** or with EditorConfig. EditorConfig files help to keep your code consistent by defining code styles and formats. These files can live with your code in its repository and use the same source control. This way, the style guidance is the same for everyone on your team who clones from that repository. With the EditorConfig rules and syntax, you can enable or disable individual .NET coding conventions and configure the severity to which you want each rule enforced.

To add an EditorConfig file to a project or solution, right-click on the project or solution name within the Solution Explorer. Select **Add New Item**. In the Add New Item dialog, search for EditorConfig. Select the .NET EditorConfig template to add an EditorConfig file prepopulated with default options. A .editorconfig file appears in Solution Explorer, and it opens in the editor, as shown in **Figure 15**.

You can also add an EditorConfig file based on the code style settings you've chosen in the Visual Studio Options dialog. The options dialog is available at **Tools > Options > Text Editor > [C# or Basic] > Code Style > General.** Click **Generate .editorconfig file from settings** to automatically generate a coding style **.editorconfig** file based on the settings on this Options page.

If violations are found, they're reported in the code editor (as a squiggle under the offending code) and in the Error List window, as shown in **Figure 16**.

The .NET 6.0 SDK has a new command called **dotnet format** that you can run in the command line in-order to apply code styles from an EditorConfig file or from the Code Style options page. To use **dotnet format**, make sure your project targets the .NET 6.0 SDK or later. Next, open the Visual Studio integrated terminal by pressing **Ctrl+`** (that's an apostrophe). You can then run **dotnet format** to apply code style preferences across your entire solution, as shown in **Figure 17**. If you'd like to learn more about **dotnet format**, visit https://aka.ms/dotnet-format.

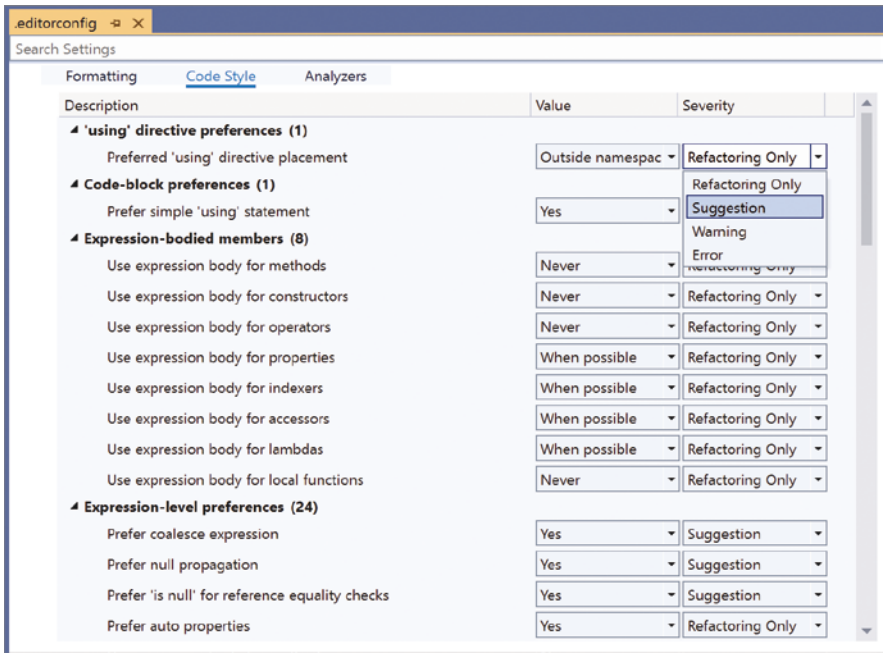

Figure 15: EditorConfig helps enforce code style

IntelliSense Completion

IntelliSense is a code-completion aid that includes a number of features, including List Members, Parameter Info, Quick Info, and Complete Word. These features help you to learn more about the code you're using, keep track of the parameters you're typing, and add calls to properties and methods with only a few keystrokes.

IntelliSense completion was recently added in DateTime and TimeSpan string literals for both C# and Visual Basic, as shown in **Figure 18**. Place your cursor inside the DateTime or TimeSpan string literal and press **Ctrl + Space** to open the completion list. You will then see completion options and an explanation as to what each character means.

Similarly, there's IntelliSense completion for regex strings, as shown in **Figure 19**. These completions also include an in-line description of what the suggestion does.

IntelliCode Context-Aware Completion

IntelliCode provides AI-assisted IntelliSense in having suggestions appear at the top of the completion list with a star icon next to them, as shown in **Figure 20**.

The completion list suggests the most likely correct API for a developer to use rather than presenting a simple alphabetical list of members and arguments. IntelliCode uses the developer's current code context as well as patterns based

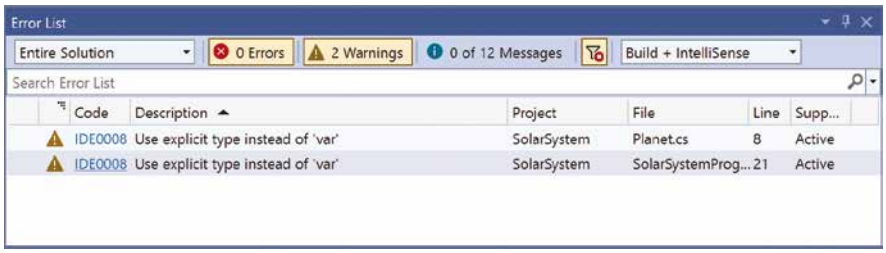

Figure 16: Error List shows warnings present in your code

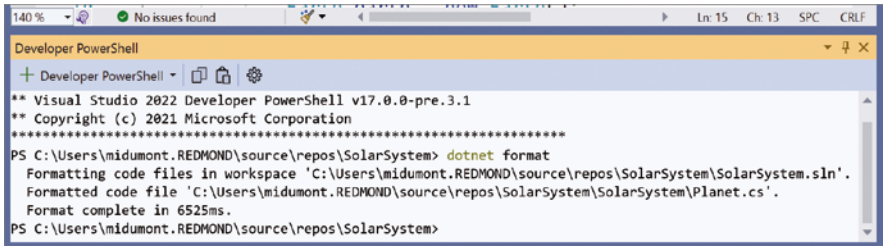

Figure 17: Run dotnet format in Visual Studio's integrated terminal to apply code style preferences to a solution

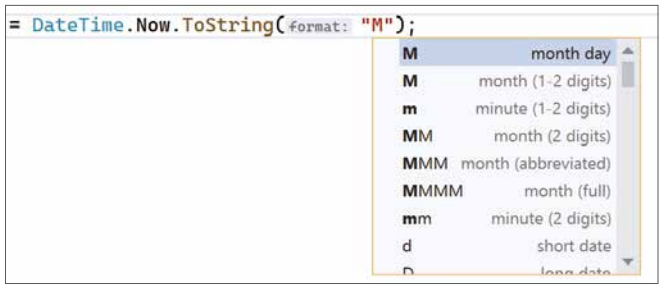

Figure 18: IntelliSense completion in DateTime and TimeSpan string literals for both C# and Visual Basic

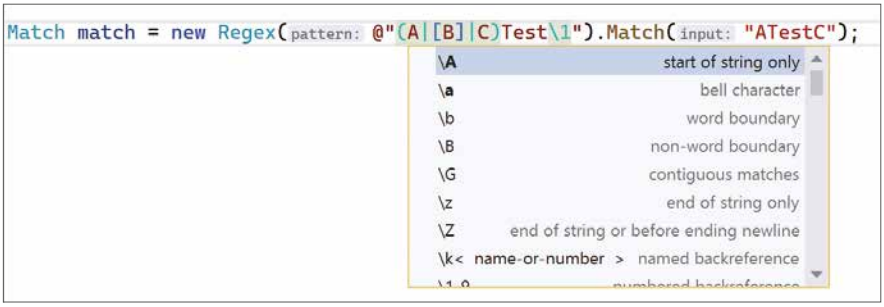

Figure 19: IntelliSense completion for regex strings

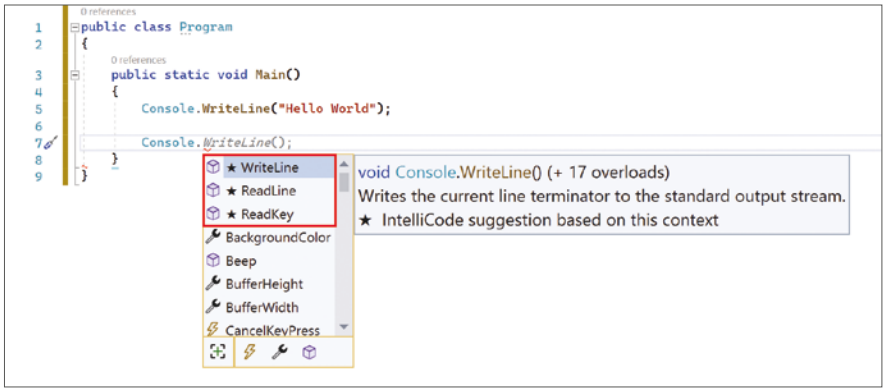

Figure 20: Starred IntelliCode suggestions in the IntelliSense completion list

on thousands of highly rated, open-source C# projects on GitHub. The results form a model that predicts the most likely and most relevant API calls.

In addition to providing AI-assisted IntelliSense completion, IntelliCode also provides context-aware inline completion. Inline completion predicts the next line of code and presents it as an inline suggestion to the right of your cursor, as shown in **Figure 21**. You can either accept the completion by pressing **tab-tab**; or keep typing to adjust the completion further.

IntelliCode can detect the manual code change that you're making and suggest an action to apply to your workflow. This is currently supported for two scenarios: generating a constructor and adding a new parameter to a constructor.

IntelliCode can also suggest edits to assist you while you're making similar changes in multiple places in your code. It tracks edits locally and detects repetition. It then offers to apply those same edits in other places where they might apply, as shown in **Figure 23**. For example, if a user has missed locations where a refactoring could be applied, IntelliCode suggestions help find those locations and fixes them.

Debugging

Visual Studio 2022 has added and improved upon features that enhance your productivity while debugging. The **Force Run to cursor** command on the right-click context menu (shown in **Figure 24**) lets you run directly to your cursor location in the source code by ignoring any breakpoints and any first-chance exception break conditions that may occur. Any breakpoints and first-chance exceptions encountered during execution are temporarily disabled.

When you're in an active debug session, a green glyph with the tooltip **Force run execution to here** appears next to the line of code where your mouse hovers (as shown in **Figure 25**), along with withholding a Shift key.

The breakpoint experience has also been improved with new UI gestures and functionalities to streamline the breakpoint debugging. The new temporary breakpoint lets you break the code only once. When debugging, Visual Studio debugger only pauses the application once for this breakpoint and deletes it automatically after it's been hit (as shown in **Figure 26**).

You can convert any breakpoint to a temporary breakpoint by enabling the "Remove breakpoint once hit" checkbox from the settings window or setting a new temporary breakpoint (as shown in **Figure 27**) with an advance breakpoint context menu on the right-click in the breakpoint gutter.

You can now also drag breakpoints from one location to another. This works for the advanced breakpoint as long as the actions/condition variables are within the Context.

There are plenty of new improvements to the Attach to Process dialog, shown in **Figure 28**, so you can identify the process that you want to attach much easier. With the new Command Line column and the app pool details in the Title column, you don't have to go back and forth with Task Manager to get the PID for those identical-looking processes. The **Show as parent/child processes** checkbox will give you a hierarchical parent-child process list view in attach to the process dialog itself.

The Select any window from desktop option lets you pick any running window from the desktop and attaches it to its associated process for debugging.

If you're working with applications that have multiple external libraries, and which have their components published to Source Servers, e.g., Newtonsoft.Json, CsvHelper, xUnit.net, etc., the new External Sources node in Solution Explorer (Figure 29) will give you an easier way to browse those sources and debug through them if needed. This node appears while debugging in the Solution Explorer and shows sources for managed modules with symbols loaded containing Source Link or Source Server information.

Remote Testing in Visual Studio

Run and debug tests on remote environments such as Linux containers, WSL, and over SSH connections from the comfort of Visual Studio. This idea of "remote testing" is now supported from the Visual Studio Test Explorer. Being able to target and debug Linux environments is crucial for cross-platform scenarios. Now you don't have to wait for feedback from CI to know how your code behaves on Linux or any other target OS. You can connect the Test Explorer directly to a remote environment, run tests there, view feedback in the Test Explorer, and even debug issues on the remote computer as they arise. Now, even testing on different operating systems can be a part of your developer inner loop!

For now, the feature is "bring your own compute." This means that you entirely leave the provisioning of the remote environments up to the user. This includes installing the necessary dependencies that your tests require in your target environment. For instance, if you want your tests targeting .NET 6.0 to run in a Linux container, you need to make sure that the container has .NET 6.0 installed via your DockerFile. Someday Microsoft might create an install experience that enables smooth acquisition of any of your test dependencies, but for now, the bulk of the provisioning of the environment is up to the user's specification. To understand the full setup details, visit the remote testing documentation at https://aka.ms/remotetesting.

Remote environments are specified using testenvironments.json in the root of your solution. An example testenvironment.json for a locally running Linux container would look something like this **Figure 30**.

Figure 21: Inline completions with IntelliCode

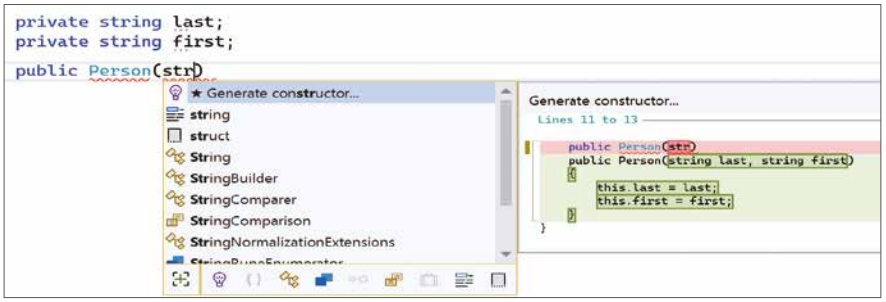

Figure 22: Suggested actions with IntelliCode

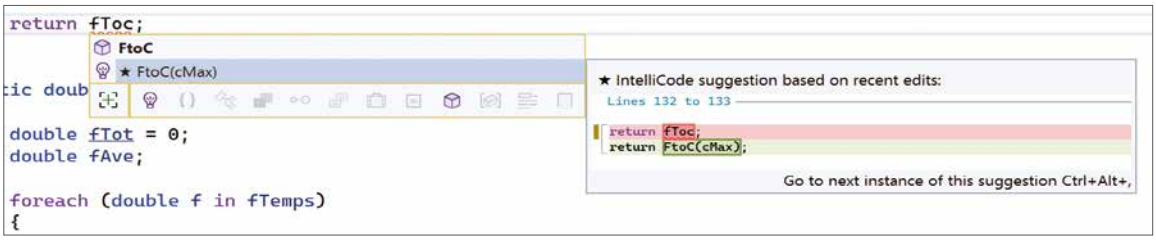

Figure 23: Suggested repeated edits with IntelliCode

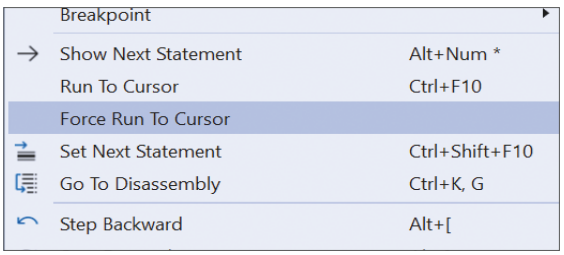

Figure 24: Force Run to Cursor command

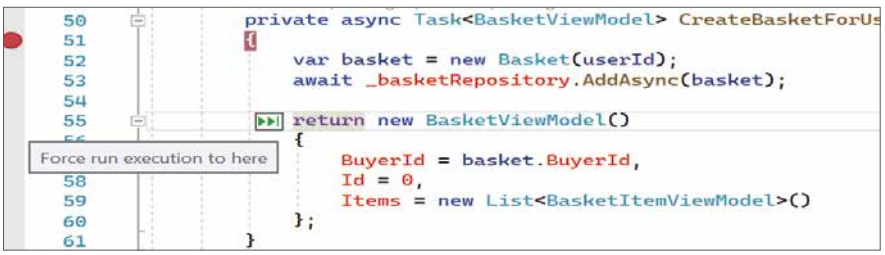

Figure 25: Force Run execution

Figure 26: Remove Temporary Breakpoint

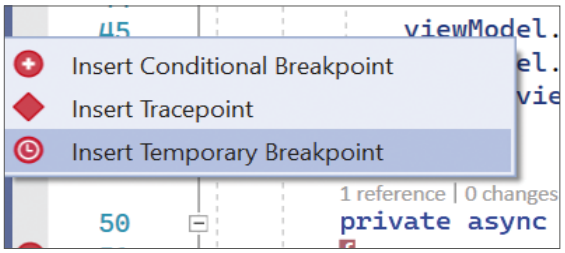

Figure 27: Add Temporary Breakpoint

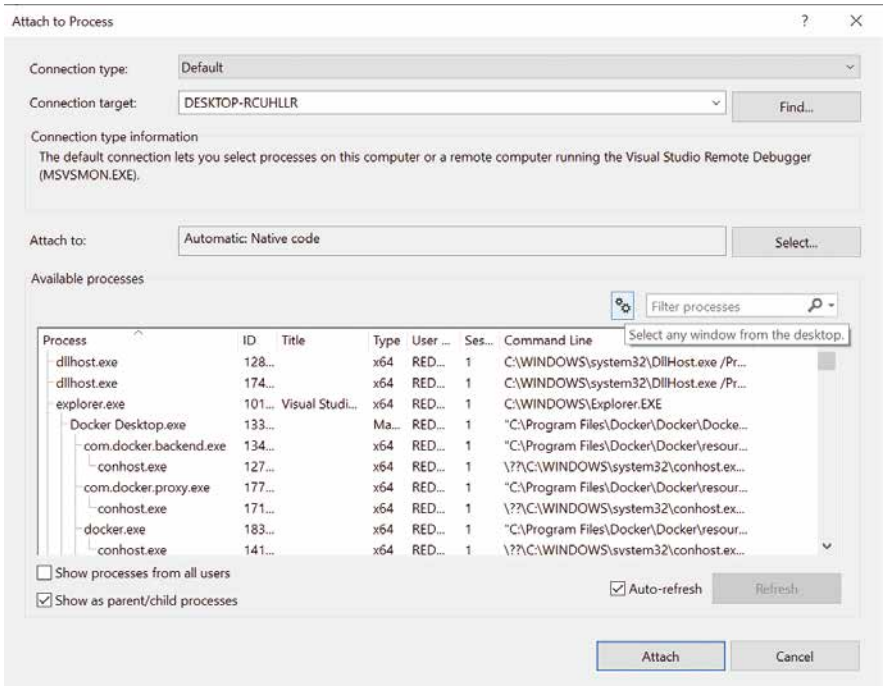

Figure 28: Attach to Process Dialog Improvements

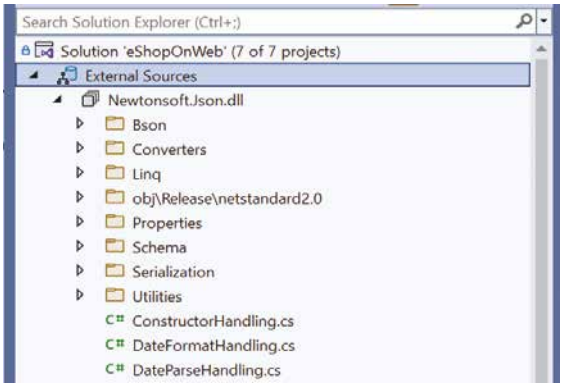

Figure 29: External Sources node in Solution Explorer

Figure 30: The testenvironment.json for local Linux container

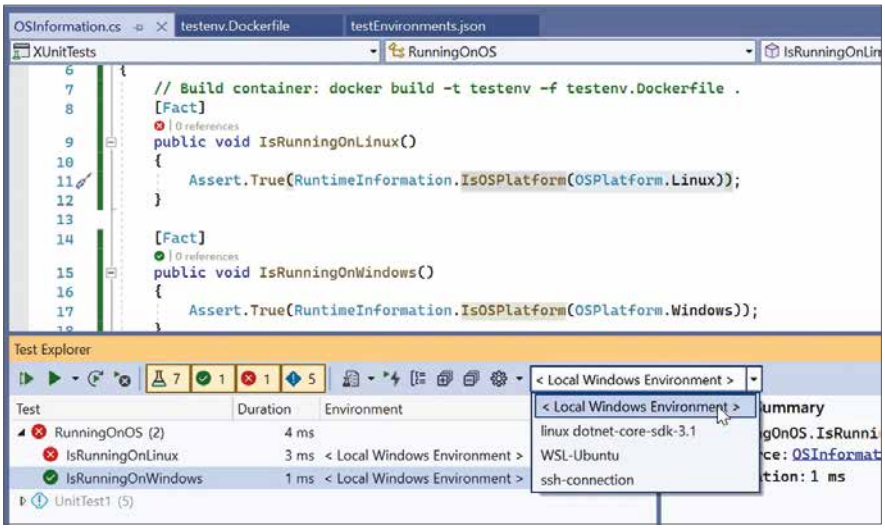

Figure 31: Remote testing drop-down in the Test Explorer

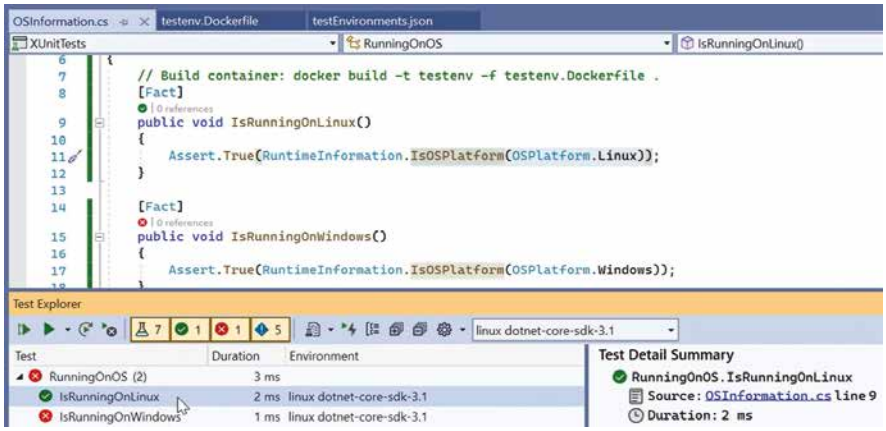

Figure 32: Linux test now passes in the Test Explorer

See the remote testing docs for full descriptions of the testenvironment.json schema and examples for containers, WSL, and SSH connections https://aka.ms/remotetesting.

Once the testenvironment.json is present, the Test Explorer loads a drop-down of remote environments, as shown in **Figure 31**.

When you select a remote environment, the Test Explorer begins discovering tests in the new environment. Once the tests are loaded, you can use the Test Explorer as you normally would for running, viewing output, grouping tests, and debugging your tests all with test results data streamed from the remote connection.

Let Visual Studio bring modern, cross-platform development to your inner loop with remote testing.

GitHub and Azure DevOps

In Visual Studio 2022, Microsoft made remote development easier than ever with better GitHub integration. The Git tooling in Visual Studio 2022 makes it easy to track changes you make to your code over time so you can both track your progress and revert to specific versions. Whether you're working

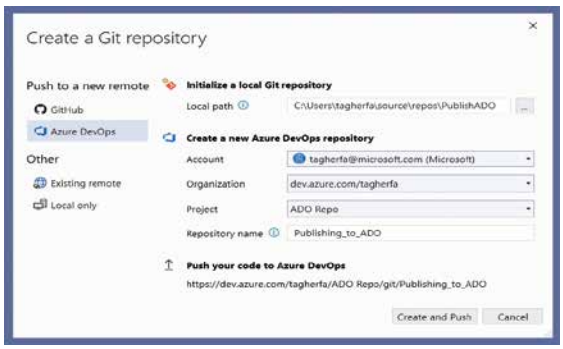

Figure 33: Create a Git repository

alone or working with a team of developers, the Git tooling in Visual Studio 2022 can be very useful to you and your team. GitHub offers free and secured cloud code storage where you can store your code and access it from any device, anywhere. Visual Studio 2022 comes with first-class GitHub and Azure DevOps functionality that makes it easy to use source control to manage your code and collaborate with others. Get started by adding your code to GitHub or Azure DevOps with the **Create a Git repository** dialog box, as shown in **Figure 33**. To do so, choose **Git** > **Create a Git repository** from the menu bar.

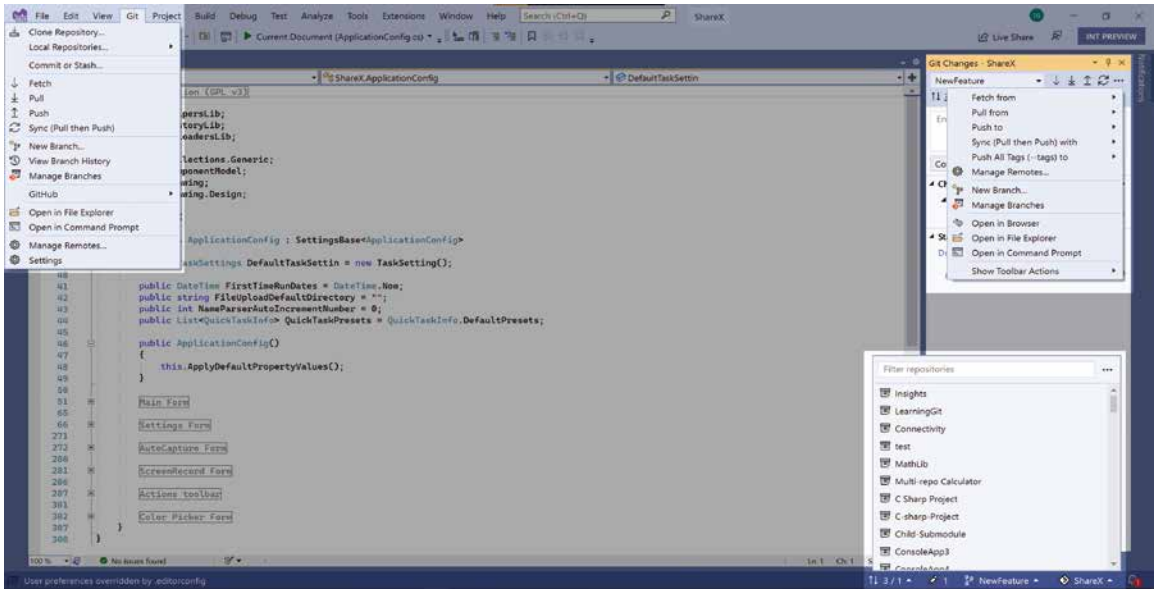

Figure 34: Git in Visual Studio 2022

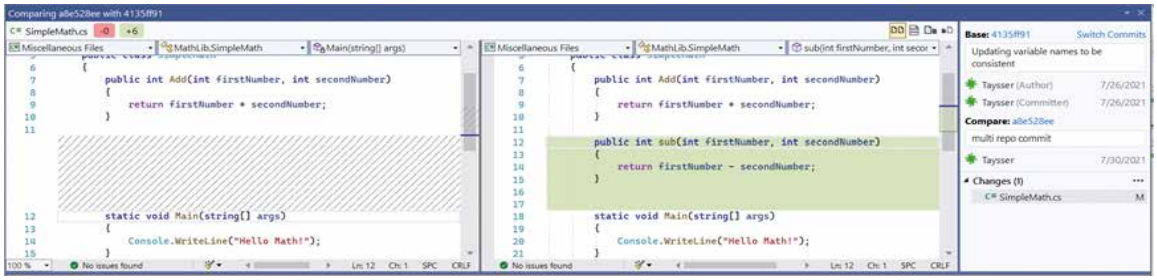

Figure 35: Comparing changes with Git

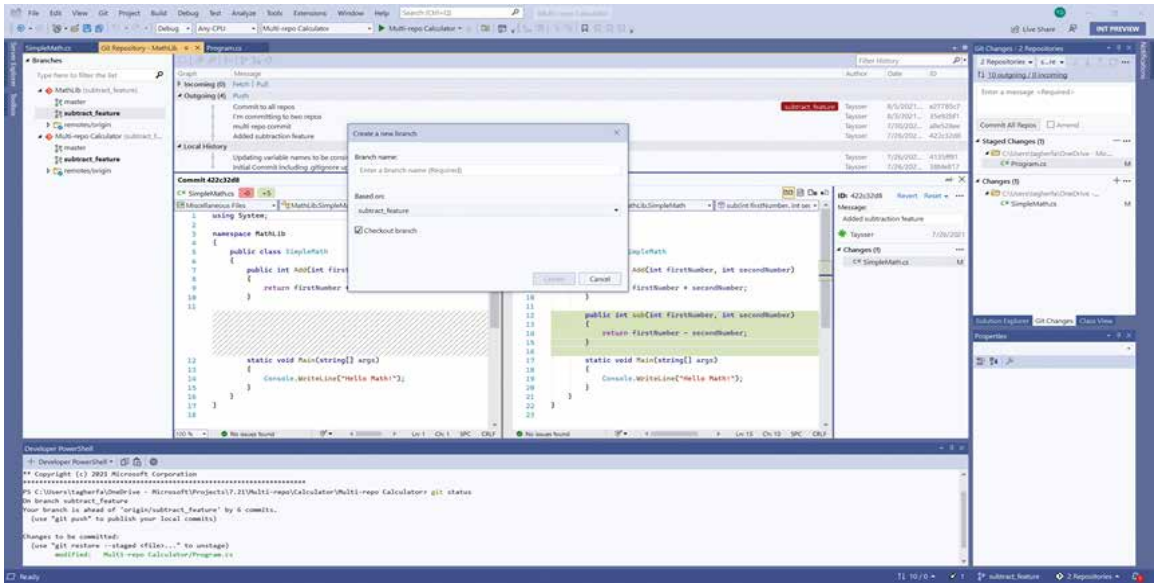

Figure 36: Managing your Git repository branches

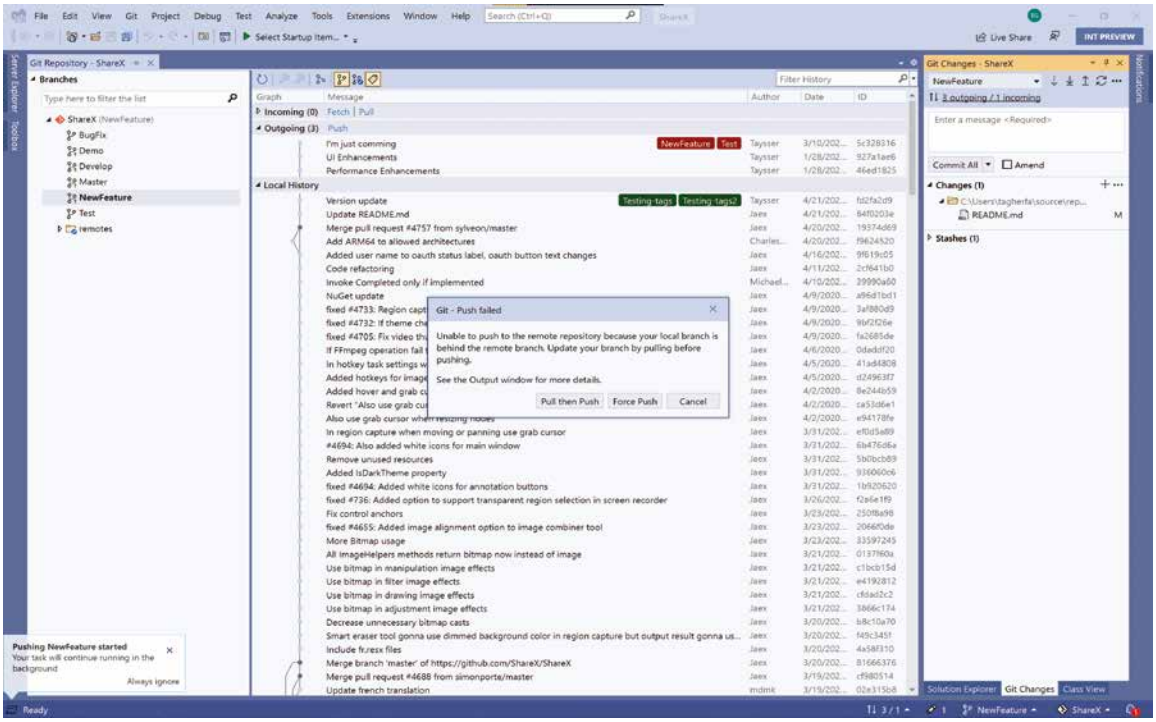

Figure 37: Git conflict resolution

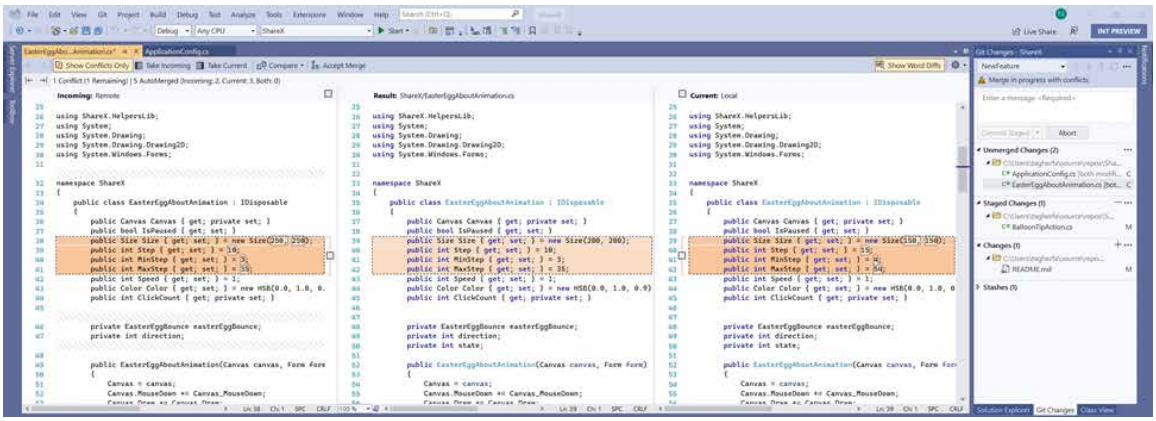

Figure 38: Git conflicts window

Resources

Hot Reload: https://aka.ms/dotnet-hotreload

Refactorings in Visual Studio 2022: https://aka.ms/refactor

Creating Roslyn analyzers: http://aka.ms/diy-analyzer

High-quality code base using .NET analyzers: https://aka.ms/dotnetanalyzers

Tips and tricks on Visual Studio Productivity: https://aka.ms/vs-productivity

Git tooling in Visual Studio 2022: https://aka.ms/vsgitdocs

Streamlined and Intuitive Git Experience

Visual Studio provides discoverable and intuitive Git features focused on maximizing the productivity of your daily workflow. You no longer need to move away from your code to commit your changes. These features include a top-level Git menu, a Git Changes window, and a Git focused Status bar. Git integrates with Visual Studio as a holistic experience; for example, both Solution Explorer and the Code Editor have a first-class Git integration, as you can see in **Figure 34**.

Repository Management and Collaboration

Visual Studio 2022 includes powerful multi repository browsing and collaboration features that eliminate the need to use other tools. Stay up to date with your repository by keeping an eye on your incoming/outgoing commits, previewing branches, and comparing commits (as shown in **Figure 35**).

And manage your repository by managing your branches (**Figure 36**) and squashing and cherry-picking commits.

The Git integration in Visual Studio promotes trust and confidence by providing contextual assistance and prompting you to do the right thing. It also includes a conflict resolution experience (**Figure 37**) that can show/hide word differences and navigate between conflicts and differences (**Figure 38**).

Get Involved

This is just a sneak peak of the latest productivity features in Visual Studio 2022. To tap into more productivity features, you can visit the Visual Studio blog site https://devblogs.microsoft.com/visualstudio/. To install Visual Studio 2022, you can visit https://visualstudio.microsoft.com/vs/preview/. As always, let Microsoft know what you think by providing feedback on the Developer Community website https://aka.ms/devcomm or using the Report a Problem tool in Visual Studio.

Mika Dumont

Essential C# 10.0: Making It Simpler

Now that C# is scheduled for an annual release, which generally occurs in November each year, it's time to review the upcoming targeted enhancements for C# vNext: C# 10.0. Although there aren't any mind-blowing new constructs (it's implausible to introduce something like LINQ every year), there's a steady stream of improvements. If I were to summarize C# 10.0, it would be removing the unneeded ceremony—such as extra curly braces or duplicate code—that doesn't add value. This synopsis shouldn't imply that the changes are irrelevant. In contrast, I think many of these changes are so strong that they will become the C# coding norm in the future. I suspect future C# developers will tend to forget the old syntax. In other words, several of these improvements are significant enough that you will likely never go back to the old way unless you require backward compatibility or need to code in an earlier version of C#.

Mark Michaelis
mark.michaelis.net
@MarkMichaelis

Mark is founder of IntelliTect, where he serves as its chief technical architect and trainer. For more than two decades, he's been a Microsoft MVP and a Microsoft Regional Director since 2007. He serves on several Microsoft software design review teams, including C#, Microsoft Azure, SharePoint, and Visual Studio ALM. He speaks at developer conferences and has written numerous books, including his latest, due in December 2019: "Essential C# 8.0 (7th Edition)".

File Scoped Namespace Declaration (#UseAlways)

To begin, consider an ever so simple feature called **file scoped namespace declaration**. Previously, namespace declaration involved placing everything within that namespace between curly brackets. With C# 10.0, you can declare the namespace before all other declarations (classes, structs, and the like) and not follow it with curly braces. As a result, the namespace automatically includes all the definitions that appear in the file.

For example, consider the following code snippet:

```
namespace EssentialCSharp10;

static class HelloWorld
{
    static void Main() { }
    // ...
}

// Additional namespace declarations are not
// allowed in the same file.
// namespace ScopedNamespaceDemo
// {
// }
// namespace AdditionalFileNamespace;
```

Here, the CSharp10 namespace is declared before any other declarations: a file scoped namespace syntax requirement. Additionally, a file scoped namespace declaration is an exclusive namespace declaration. No other namespaces, either traditionally curly-scoped or additional file-scoped, are allowed within the file.

Although not a significant change, I expect that I will always use this 10.0 feature going forward. Not only is the statement simpler without the curly braces, but it also means I no longer need to indent other declarations within the namespace. For this reason, I've tagged it with #UseAlways in the title. In addition, I think this feature warrants updating the C# coding guideline, assuming C# 10.0 or later: Do use file-based namespace declarations.

Global Using Directive (#UseAlways)

My #UseAlways recommendation might be surprising because namespace declarations haven't changed since C# 1.0, but C# 10.0 includes a second namespace-related change: global namespaces directives.

Good programmers refactor ruthlessly! Why is it, then, that C# forces us to re-declare a series of namespaces at the top of every file? For example, most files include a using System directive at the top. Similarly, a unit testing project virtually always imports the namespace for the target assembly under test and the test framework namespace. Why is it necessary to write the same using directive repeatedly for each new file? Wouldn't it make sense to write a single using directive that applies globally through the project?

Of course, the answer is yes. For namespaces that you have using directives throughout your project, you can now provide a global using directive that will import the namespace throughout the project. The syntax requires the new global contextual keyword to prefix a standard using directive, as shown in the following snippet within an XUnit test project:

```
global using EssentialCSharp10;
global using System;
global using Xunit;

global using static System.Console;
```

You can place the above snippet anywhere within your code. By convention, however, consider something like a GlobalUsings.cs or Usings.cs file. Furthermore, once the global using directives are in place, you can leverage them in all files within the project:

```
public class SampleUnitTest
{
    [Fact]
    public void Test()
    {
        // No using System needed.
        DateTime dateTime =
            DateTime.Now;

        // No using Xunit needed.
        Assert.True(
            dateTime <= DateTime.Now
        );

        WriteLine("...");
    }
}
```

Note, global using directives include support for using static. As a result, you can have a WriteLine() statement with no "System.Console" qualifier. Global aliases using directive syntax are also supported.

In addition to coding global using statements in C# explicitly, you can also declare them within MSBuild (as of 6.0.100-rc.1). A Using element within your CSPROJ file for example

(i.e., <Using Include="Microsoft.VisualStudio.TestTools.UnitTesting" />) generates an ImpklicitNamespaceImports.cs file that includes a corresponding global namespace declaration. Furthermore, adding a static attribute (e.g., Static="true") or an alias attribute, such as Alias=" UnitTesting," generates the corresponding static or alias directives. Furthermore, the some of the target frameworks include implicit global namespace directives. See https://docs.microsoft.com/en-us/dotnet/core/project-sdk/msbuild-props#disableimplicitnamespaceimports for a corresponding list. If, however, you prefer that no such default global namespaces get generated, you can turn them off with a ImplicitUsings element set to disable or false. Here's a sample PropertyGroup element from CSPROJ:

```
<PropertyGroup>
    <ImplicitUsings>disable</ImplicitUsings>
    <Using>EssentialCSharp10</Using>
    <Using>System</Using>
    <Using>Xunit</Using>
    <Using Static="true">System.Console</Using>
</PropertyGroup>
```

Note that this is not yet working with Visual Studio 2022 Preview 3.1.

You don't want to convert all your using directives to global using directives, or you're likely to run into ambiguities in the unqualified type names. However, I expect at least a few global declarations in most projects—such as the default ones from the target framework at a minimum, hence the #UseAlways tag.

Constant Interpolated Strings (#UsedFrequently)

One feature that until now has no doubt irritated you is the lack of a method to declare a constant interpolated string, even though the value is entirely determinable from other constants or compile-time determined values. C# 10 addresses this issue. Here are some examples to consider:

```
const string author =
    "Mother Theresa";
const string dontWorry =
    "Never worry about numbers.";
const string instead =
    "Help one person at a time and always " +
    "start with the person nearest you.";
const string quote =
    $"{ dontWorry } { instead } - { author }";
```

One case I particularly appreciate with constant interpolation is leveraging the nameof operator within attributes, as demonstrated in the following snippet:

```
[Obsolete($"Use {nameof(Thing2)} instead.")]
class Thing1 { }
class Thing2 { }
```

Prior to C# 10.0, the inability to use the nameof operator within a constant string literal was undoubtedly a source of consternation.

Lambda Improvements

C# 10.0 includes three improvements to lambda syntax support—both expressions and statements.

Attributes

With a host of new parameter attributes introduced with C# 8's nullable-references, the lack of attribute support in lambdas became even more noticeable. Fortunately, in C# 10, attributes on lambdas are now supported, including attributes on the return type:

```
Func<string?, string[]?>? Func =
    [return: NotNullIfNotNull("cityState")]
    static (string? cityState) =>
        cityState?.Split(", ");
```

Note that in order to have attributes on the lambda, it's also necessary to surround the parameter list with parenthesis. _ = [return: NotNullIfNotNull("cityState")] cityState => cityState?.Split(", ") is not allowed.

Explicit Return Type

If you're in the habit of using implicit type declaration with var, it's not that uncommon that the compiler is unable to figure out a method signature. Consider, for example, a method that returns null if it fails to parse text to a nullable integer:

```
var func = (string? text) =>
     int.TryParse(text, out number)?number:null;
```

The problem here is that both **int?** or **object** would be valid returns. And it isn't obvious which to use. Although it's possible to cast the result, the syntax for casting large expression is cumbersome. A preferable alternative, and the one available starting with C# 10.0, is to allow for declaring the return type as part of the lambda syntax:

```
Func<string?, int?> func = int? (string? text) =>
     int.TryParse(text, out int number)?number:null;
```

The added bonus for those using var less habitually is that the addition of the return type declaration enables quick actions to convert a var to the explicit lambda type, as demonstrated in the snippet above: Func<string?, int?>.

Note, lambdas declared with delegate { } syntax aren't supported: Func<int> func = delegate int { return 42; } won 't compile.

Natural Function Types

Lastly, it's possible to infer a natural delegate type for lambdas and statements. Essentially, the compiler will do a "best fit" attempt to determine the signature of the lambda expression or statement and thereby allowing the programmer to avoid specifying redundant types when the compiler can infer the type.

Caller Expression Attribute (#UsedRarely)

This feature has three different usage profiles. If you write perfect code that never needs debugging or triage, this next feature is probably useless for you. Alternatively, if you use debug classes, logging classes, or unit test assertions, this could be invaluable without you even knowing it exists. You essentially reap the benefits of the feature without having to make any adjustments to your code. If you're a library vendor where you provide validation or assertion logic, it's paramount that you use this feature where applicable. It will significantly improve your API functionality.

Working with Caller Attribute Methods

Imagine a function that validates a string parameter – verifying that it isn't null or empty. You could leverage such a function in a property as follows:

```
using static EssentialCSharp10.Tests.Verify;

class Person
{
    public string Name
    {
        get => _Name ?? "";
        set => _Name = AssertNotNullOrEmpty(value);
    }
    private string? _Name;
}
```

There's no C# 10.0-specific feature visible in this code snippet. **Name** is a non-nullable property with an assert method that throws an exception if the value is null or empty. So, what's the feature?

The difference shows up at runtime. In this case, when the value is null or empty, the AssertNotNullOrEmpty() method throws an ArgumentNull exception whose message includes the argument expression, "value." And, if method invocation was AssertNotNullOrEmpty("${firstName}{lastName}"), then the ArgumentNull exception message would include the exact text: "${firstName}{lastName}", because that was the argument expression specified when calling the method.

Logging is another area where this feature would prove very helpful. Rather than calling Logger.LogExpression($"Math.Sqrt(number) == { Math.Sqrt(number) }" you could instead call Logger.LogExpression(Math.Sqrt(number)) and have the Log() method include both the value and the expression in the output message.

Implementing Caller Attribute Methods

One of the big advantages of this **Caller Attribute Method** is added functionality without the caller knowing or making any source code changes. When you're implementing an assert, logging, or debug type method, you need to understand how to declare and leverage the feature. Here's a listing demonstrating the AssertNotNullOrEmpty() method implementation:

```
public static string AssertNotNullOrEmpty(
     string? argument,
     [CallerArgumentExpression("argument")]
         string argumentExpression = null!)
{
    if (string.IsNullOrEmpty(argument))
    {
        throw new ArgumentException(
            "Argument cannot be null or empty.",
            argumentExpression);
    }
    return argument;
}
```

The first thing to note is the CallerArgumentExpression attribute decorating the argumentExpression parameter. By adding this attribute, the C# compiler injects the expression specified as the argument into the argumentExpression. In other words, although the caller statement was coded as _Name = AssertNotNullOrEmpty(value), the C# compiler morphs the call into _Name = AssertNotNullOrEmpty(value, "value"). The result is that the AssertNotNullOrEmpty() method now has both the calculated value for the argument expression (in this case, it's value's value, as well as the expression itself). Thereby, when throwing the ArgumentException, not only can the message identify what was wrong, "Argument cannot be null or empty," but it can provide the text for the "value" expression.

Notice that the CallerArgumentExpression attribute includes a string parameter that identifies which parameter's expression in the implementing method will be injected into the CallerArgumentExpression's parameter. In this case, because "argument" is specified, the expression from the "argument" parameter is injected into the value of argumentExpression.

The result is that you're not limited to only using the CallerArgumentExpression on one parameter. You could, for example, have an AssertAreEqual(expected, actual, [CallerArgumentExpression("expected")] string expectedExpression = null!, [CallerArgumentExpression("actual")] string actualExpression = null!) and then provide an exception that shows the expressions, not just the end results.

There are a few coding guidelines to consider when implementing a CallerArgumentExpression method:

- Do declare the caller argument expression parameter as optional (using "=null!") so that invoking the method doesn't require the caller to identify the expression explicitly. In addition, it allows the feature to be added to existing APIs without the caller code changing.
- Consider declaring the caller argument expression parameter as non-nullable and assign null with the null-forgiveness operator (!). This allows the compiler to specify the value by default and imply that it's intended not to be null if an explicit value is set.

Parenthetically, it's unfortunate, but as of C# 10.0, you can't use the **nameof** operator to identify the parameter. For example, CallerArgumentExpression(nameof(argument)) won't work. That's because the argument parameter isn't in scope at the time the attribute is declared. However, such support is under consideration post-C# 10.0 (see Support for method parameter names in nameof(): https://github.com/dotnet/csharplang/issues/373).

Record Structs (#UsedOccasionally)

C# 9.0 added support for records. At the time, all records were reference types and potentially mutable. The advantage of adding the record types is that they provided a concise syntax for defining a new type with the primary purpose of encapsulating data (with less emphasis on providing behavior or a service). Now, records of both types have C# compiler-generated implementations of value equality, non-destructive mutation, and built-in display formatting.

Record Structs Versus Record Classes

In C# 10.0, the record feature was extended to allow for record value types (**record structs**) and record reference types (**record classes**). For example, consider the Angle declaration shown here:

```
record struct Angle(
    double Degrees, double Minutes, int Seconds)
{
    // By default, primary constructor
    // parameters are generated as read-write
```

```
    // properties:
    // public double Degrees {get; set;}

    // You can override the primary
    // constructor parameter implementation
    // Including making them read-only
    // (no setter) or init only.
    public double Minutes {
        get; init; } = Minutes;

    // Primary constructor parameters can be
    // overridden to be fields
    public int Seconds = Seconds;
}
```

Declaring the record as a value type involves adding the struct keyword between the contextual keyword record and the data type name. Like record classes, you can also declare a **primary constructor** immediately following the data type name. This declaration instructs the compiler to generate a public constructor (i.e., Angle [double Degrees, double Minutes, int Seconds]) that assigns the fields and properties (i.e., degrees, minutes, and seconds members) with the corresponding constructor parameter values. If not explicitly declared, the C# compiler generates properties corresponding to the primary constructor parameters (i.e., Degrees). Of course, you can add additional members to the record struct and override the generated properties with custom implementations even if the accessibility modifier isn't public or the property is read-only, init-only, or read-write.

Record structs include the same benefits of record classes except where behavior is inherent to C# value types. For example, equality methods (Equals(), !=, ==, and GetHashCode()) are all auto-generated at compile time. After all, perhaps these methods are the key feature justifying the record data type. Also, like record classes, records structs include a default implementation for ToString() that provides a formatted output of the property values. (The ToString() output for an instantiated Angle, for example, returns "Angle { Degrees = 30, Minutes = 18, Seconds = 0 }"). In addition, record structs include a deconstruct method that allows an instance of the type to convert into a set of variables corresponding to the primary constructor: i.e., (int degrees, int minutes, int seconds) = new Angle(30, 18,42). One last feature that both record types have in common is the with operator. It enables cloning the record into a new instance, optionally with modification on selected properties. Here's an example:

```
public static Angle operator +(
     Angle first, Angle second)
{
    (int degrees, int minutes, int seconds) = (
        (first.Degrees + second.Degrees),
        (first.Minutes + second.Minutes),
        (first.Seconds + second.Seconds));

    return first with
    {
        Seconds = seconds % 60,
        Minutes = (minutes +
            (int)(seconds / 60)) % 60,
        Degrees =
            (int)(degrees +
                (int)((minutes
                    + (int)(seconds / 60)) / 60)
            ) % 360
    };
}
```

And, as a bonus, with support was added to structs (not only record structs) in C# 10.0.

Unlike record classes, record structs always derive from System.ValueType because they're .NET value types and therefore no additional inheritance is supported. In addition, record structs can be qualified with the readonly keyword (readonly record struct Angle {}), thus rendering them immutable once the type is fully instantiated. Given the readonly modifier, the compiler verifies that no fields (including automated property-backing fields) are modified once object initialization is complete. As demonstrated with the Seconds primary constructor parameter, another difference with record structs is that you can override the primary constructor behavior to generate fields rather than properties.

Parenthetically, you can also now declare a record reference type with the class keyword (using simple record <TypeName> is still allowed), thus providing symmetry between the two types of record declaration:

```
record class FingerPrint(
    string CreatedBy, string? ModifiedBy = null)
        {}
```

Note: The readonly record modifier isn't allowed with record classes at this time.

Mutable Record Structs

One thing to be wary of is that, unlike a record class, record struct primary constructor parameters are read/write by de-

fault (at least in the Visual Studio Enterprise 2022 Preview [64-bit] Version 17.0.0 Preview 3.1 available at the time of writing). This default is surprising to me for two reasons:

- Historically, structs were declared immutable to avoid erroneously attempting to modify the struct by insidiously modifying a copy. This is primarily because passing a value type as a parameter, by definition, creates a copy. Modifying the copy is unexpectedly not reflected at the caller (if it wasn't obvious or well known that the type was a value type). Perhaps more insidious is a member that mutated the instance. Imagine a Rotate() method on an Angle instance that rotated the same Angle instance. Invoking said method from a collection, i.e., Angle[0].Rotate(42,42,42), unintentionally doesn't change the value stored in angle[0].
- Parenthetically, mutable value types were far more problematic when modifying them inside a collection was allowable. However, something like Angle[0].Degrees = 42 is now prevented by the compiler even if Degrees is writable, thus preventing unexpectedly not modifying Degrees.
- Mutable record structs, by default, would be inconsistent with record classes. Consistency behavior is a strong motivator when it comes to learning, understanding, and remembering.

Despite the considerations, there are important differences that distinguish the suitability of mutable record structs:

- Using a struct as a dictionary key does not carry the same risk of getting lost in a dictionary (assuming no self-modifying member is provided).
- There are reasonable scenarios where mutable fields are not problematic (like with System.ValueTuple).
- Supporting mutability and fields in record structs allows for tuples to easily be "upgraded" to record structs.
- Record structs include the readonly modifier, which renders it immutable with a single keyword.

Note: As of this writing in early September, the decision about mutability by default has not been finalized.

It's relatively rare for developers to need custom value types, so I'm tagging this feature as #UsedOccasionaly. Even so, I appreciate all the careful thought put into the ability to define value-type records. More importantly, almost all value types require implementation for equality behavior. For this reason, I suggest you consider a coding guideline: do use record structs when defining a struct.

Default (Parameterless) Struct Constructors

C# has never allowed a default constructor (a parameterless constructor) on a struct. Without it, there also isn't support for field initializers on structs because there's no place for the complier to inject the code. In C# 10.0, this gap between structs and classes is closed with the ability to define a default constructor on a struct (including a record struct) and allowing field (and property) initializers on structs. The following snippet provides an example.

```
public record struct Thing(string Name)
{
    public Thing() : this("<default>")
    {
        Name = Id.ToString();
    }

    public Guid Id { get; } = Guid.NewGuid();
}
```

In this example, you define an ID property that's assigned with a property initializer. In addition, the default constructor is defined and initializes the Name property to be the Id.ToString() value. It's important to note that the C# compiler injects the field/property initializer at the top of the default constructor. This location ensures that the C# compiler generates a primary constructor for the struct and ensures that it's invoked before the body of the default constructor is evaluated. Conceptually, this is very similar to how constructors on classes behave. Property and field initializers are also done inside of the generated primary constructor to ensure that those values are set before the body of the default constructor is executed. The effect is that the ID is sent by the time the user-defined portion of the constructor executes. Be aware that the compiler enforces the this() constructor invocation when a primary constructor is specified on the record struct.

Because default constructors were previously unavailable, guidelines dictated that default (zeroed out) values were valid even in an uninitialized state. Unfortunately, the same is true, even with default constructors, because zeroed-out blocks of memory are still used to initialize structs. For example, when instantiating an array of **n Things**—that is, var things = new Thing[42]—or when not setting member fields/properties in a containing type before accessing them.

Additional Improvements in C# 10.0

There are several other simplifications in C# 10.0. I put these in the category of you-didn't-know-this-wasn't-possible-until-you-tried. In other words, if you encountered these scenarios before, you were probably frustrated by the idiosyncrasy but worked around it and continued. With C# 10.0, the issue is eliminated. **Table 1** provides a list of such features along with code samples.

What's Not in C# 10.0

Several planned C# 10.0 features didn't make the cut.

- Nameof(parameter) inside an attribute constructor won't be supported.
- There won't be a parameter null-checking operator that decorates a parameter and then throws an exception if the null value is used.
- Generic attributes that include a type parameter when using the attribute.*
- Static abstracts members in interfaces forcing the implementing type to provide the member.*
- Required properties so that the value must be set in a compile time-verifiable way before construction completes
- Field keyword that virtually eliminates all need for a separate field to be declared.*

Asterisked (*) items are available in C# 10.0 preview, but are expected to be removed before general release.

Name	Sample Code	Description

Improved Definite Assignment Analysis	`string text = null;`
	`if (`
	`  text?.TryIntParse(`
	`  out int number) == true)`
	`{`
	`  number.ToString(); // Undefined error`
	`}`
		Occasionally, the scope of an out parameter declared inline of a method wasn't available within the statement block.
		This enhancement also improves the quality of null reference analysis for reference types. #UsedRarely

Record Classes with Sealed ToString()	`public record class Thing1(string Name)`
	`{`
	`  public sealed override string`
	`  ToString() => Name;`
	`}`
		In a record class, you can identify a ToString() method as sealed to prevent sub-types from overriding the implementation and potentially obscuring the intention of the technique. This isn't possible in record structs because inheritance isn't supported. #UsedRarely

Enhanced #Line Directive	`#line 42 "8. LineDirectiveTests.cs"`
	`  throw new Exception();`
	`//  ^`
	`//  |`
	`//  Column 13`
	`#line default`
		This identifies the starting character in a #line directive based on the first character of the following line.
		In this example, the first character 't' in throw identifies the column number. #UsedRarely

AsyncMethodBuilder override	`[AsyncMethodBuilder(`
	`  typeof(AsyncValueTaskMethodBuilder))]`
	`public readonly struct ValueTask :`
	`  IEquatable<ValueTask>`
	`{`
	`  //...`
	`}`
		This allows each async method to specify a custom AsyncMethodBuilder, rather than relying only on a class-specified builder. #UsedRarely

Extended property patterns	`// C# 8.0 syntax:`
	`// if(person is`
	`//  { Name: { Length: 0 } }) {}`
	`if (person is { Name.Length: 0 })`
	`{`
	`  throw new InvalidOperationException(`
	`  @$"Invalid {`
	`  nameof(Person.Name)}.");`
	`}`
		Rather than using curly braces to traverse a property chain, C# 10.0 allows "dot" notation, which is easier to understand. Going forward, a reasonable coding guideline would be: DO use the dot notation syntax for property pattern matching traversal. #UsedOccasionally

String Interpolation Improvements	N/A	Allowing for association with the constant interpolated string is a significant performance improvement in interpolated strings in general. In the past, interpolated string ultimately resulted in a call to string.Format(), which is an inefficient implementation given the rampant boxing, likely argument array allocations, string instantiation, and inability to leverage Span. Much of this was addressed in .NET 6.0 and C# compiler improvements. The details are available in Stephen Toub's excellent article String Interpolation in C# 10.0 and .NET 6 found at devblogs.microsoft.com/dotnet/string-interpolation-in-c-10-and-net-.6/. #UsedFrequently

Table 1: Additional Improvements

Of these features, I was most looking forward to the null-checking operator, but at the same time, I'm holding onto hope for a more generic solution to arrive that provides parameter checking for more than just null. Having support nameof(parameter) in method attributes will also be great, both for CallerArgumentExpression attributes as well as ASP.NET and Entity Framework development.

Summary

There are many relatively small "improvements" of C# 10.0; I don't really see them as new features. Rather, they are the sort of things that I previously assumed were already possible only to encounter compiler errors after coding. Now with the improvements in C# 10.0, I'll likely forget the time prior when they didn't work.

Beyond just the improvements, admittedly there isn't anything revolutionary in C# 10.0, but it certainly includes some features that will change the way I code: global using directives and file-based namespace declarations, to name a few. Although it's something I'll rarely code myself, I'm eager for logging, debugging, and unit testing library developers to update their APIs with support for caller argument expression attributes. Triage and diagnostics will be easier with the new APIs. And, although I think that defining custom value is rarely needed, the addition of record structs certainly makes it easier with all the equality support. For this reason alone, I suspect it's rare that someone would define a custom value type without using record struct.

Putting it all together, C# 10.0 is a welcome addition with a healthy set of features and improvements—enough enhancements, in fact, that I'll be disappointed whenever I have to program with an earlier version of C#.

Mark Michaelis

What's New in ASP.NET Core in .NET 6

ASP.NET Core is a modern Web framework for .NET and includes everything you need to build beautiful Web UIs and powerful back-end services. Unlike other development platforms that require you to piece together a Web application from multiple frameworks, ASP.NET Core offers a complete and cohesive Web development solution (see Figure 1).

With ASP.NET Core, you can build dynamic server-rendered UIs using MVC or Razor Pages. You can integrate ASP.NET Core with popular JavaScript frameworks or you can build rich interactive client Web UIs completely in .NET using Blazor. For services, you can use ASP.NET Core to build standards-based HTTP APIs, real-time services with SignalR, or high-performance back-end services with gRPC.

Under the hood, ASP.NET Core provides a flexible hosting model, high performance servers, and a rich set of built-in middleware to handle cross-cutting concerns like localization and security. No matter what kind of Web, server, or cloud app you're trying to build, ASP.NET Core offers a complete and fully integrated solution.

The next wave of new features and updates to ASP.NET Core is now available with .NET 6. .NET 6 is the latest major release of .NET, which now ships on a regular yearly cadence. .NET 6 is also a Long Term Support (LTS) release, which means that it will enjoy three full years of support.

.NET 6 includes improvements that cover a broad set of themes:

- **New developers**: Makes it easier for new developers to get started with .NET
- **Client apps**: Expands support for building cross-platform native client apps
- **Cloud native**: Ensures that .NET has everything you need to run natively in the cloud
- **Enterprise and LTS**: Ensures that enterprises relying on LTS releases have a smooth upgrade path
- **Ecosystem**: Strengthens the .NET ecosystem
- **Inner-loop performance**: Makes development with .NET faster and more productive
- **Meet developer expectations:** Continues to deliver on the promises of the .NET platform

You can browse and dive into each of these .NET 6 themes on the https://themesof.net site and on GitHub.

ASP.NET Core has contributed new functionality and improvements to almost all the .NET 6 themes. To make it easier to find all the ASP.NET Core related work, there's an "ASP.NET Core in .NET 6 roadmap" issue linked to from the themesof.net site.

How did the ASP.NET team decide on these themes and enhancements? They were collected based on feedback from .NET community members, like you! Every suggestion, issue report, pull request, comment, and thumbs up has contributed to making .NET 6 a great release. Thank you for all your feedback and contributions!

There's lots that's new in in ASP.NET Core in .NET 6. Let's dive in and see what ASP.NET in .NET 6 has to offer.

Getting Started with ASP.NET Core in .NET 6

Getting started with ASP.NET Core in .NET 6 is easy. Just go to https://dot.net and install the .NET 6 SDK for your platform of choice. To create and run your first app, simply run the following commands:

```
dotnet new web
dotnet watch
```

You did it! Your browser should pop up and navigate to the running app.

You might see a warning in the browser that the development HTTPS certificate isn't trusted. To set up the ASP.NET Core development certificate for local development, run the following command and then restart the app:

```
dotnet dev-certs https --trust
```

.NET 6 is included with Visual Studio 2022, so if you have that installed, you're all set to go. Just make sure you've got the "ASP.NET and Web development" Visual Studio workload installed to enable the Web related tooling. If you're using an older version of Visual Studio, you'll need to upgrade to Visual Studio 2022. .NET 6 development isn't supported in older Visual Studio versions, so go ahead and treat yourself to the latest and greatest Visual Studio version.

To update an existing project to .NET 6, first update the target framework in your projects to **net6.0** and then update any package references to the .NET 6 versions. .NET 6 is a highly compatible release with previous .NET versions, but double check the .NET 6 release notes for any breaking changes that might affect you. Most .NET 5 apps should work on .NET 6.

Now that you're all set up, let's check out all the new features.

Minimal APIs

Building your first ASP.NET Core app is easier than ever before with .NET 6. You can now build your first ASP.NET Core app with a single C# file and just a few lines of code.

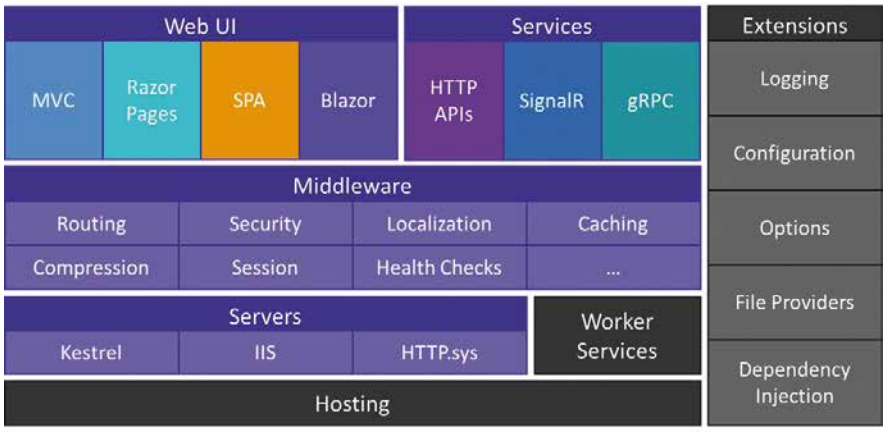

Figure 1: ASP.NET Core: A complete Web development solution

Daniel Roth
daroth@microsoft.com
@danroth27

Daniel Roth is a Principal Program 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 .NET and Blazor.

Here's a complete ASP.NET Core app with .NET 6:

```
var app = WebApplication.Create(args);

app.MapGet("/", () => "Hello World!");

app.Run();
```

That's it! Run the app and you get a single HTTP endpoint that returns the text "Hello World!"

If you've been coding in C# for a while, you might be wondering how this code even compiles. Where's Program.Main? Modern C# no longer requires you to define Program.Main to get your program started. Instead, you can define top-level statements for your app's entry point, which reduces the amount of boilerplate code.

Where are the namespaces and using directives? You don't need to add using directives for the most common namespaces in .NET 6 because they're defined implicitly for you using the new C# 10 support for global using directives.

ASP.NET Core in .NET 6 takes full advantage of modern C#. All of the ASP.NET Core project templates have been updated to use the latest C# features including top-level statements, implicit global using directives, file-scoped namespace declarations, and nullability checking.

Microsoft has also introduced a new minimal hosting API for ASP.NET Core. The new WebApplication API gives you all the flexibility of a traditional ASP.NET Core Startup class, but with much less ceremony. The existing pattern of using a Startup class is, of course, still supported, but the new API is much more convenient.

You can add middleware directly to your WebApplication using the normal middleware extension methods just like you would in Startup.Configure:

```
var app = WebApplication.Create(args);

// Configure the HTTP request pipeline.
app.UseStaticFiles();

app.Run();
```

To add services, create a WebApplicationBuilder and add services to the builder.Services property like you would in Startup.ConfigureServices:

```
var builder =
    WebApplication.CreateBuilder(args);
// Add services to the container.
builder.Servcies.AddSingleton<WeatherService>();

var app = builder.Build();

...

app.Run();
```

WebApplication sets up routing for you, so you're free to start adding endpoints right away. Improvements to routing in ASP.NET Core in .NET 6 makes building microservices and HTTP APIs trivial. You can create a minimal API by simply mapping a route to a method directly on your app. No need to define an entire API controller class (although doing so is still fully supported).

You've already seen that you can route to a method that returns a string to create minimal API that returns some plain text. If you return a complex object, it automatically gets serialized as JSON.

```
app.MapGet("/todos", () => new[] {
    new { Title = "Try .NET 6", IsDone = true},
    new { Title = "Eat veggies", IsDone = false},
});
```

With this minimal API, a GET request to **/todos** returns the following JSON response:

```
[
  {"title":"Try .NET 6","isDone":true},
  {"title":"Eat veggies","isDone":false}
]
```

You can also return an IResult instance from a route handler method. The Results static helper class provides many IResult implementations for common response types, like 404 Not Found or 201 Created.

Method parameters in route handlers can be bound to all sorts of useful stuff:

- HttpContext
- The HttpRequest or HttpResponse
- The ClaimsPrincipal for the current user
- Configured services

Simple type parameters get automatically bound to route value and query string parameters:

```
app.MapGet("/hello/{name}",
    (string name) => $"Hello {name}!");
```

Complex parameters get bound to JSON data in the request:

```
app.MapPost("/todos",
    async (Todo todo, TodoDbContext db) =>
{
    await db.Todos.AddAsync(todo);
    await db.SaveChangesAsync();

    return Results.Created(
        $"/todo/{todo.Id}", todo);
});
```

Minimal APIs require only a minimal amount of code to implement, but they can be used to define APIs both big and small. Minimal APIs benefit from all the great functionality in ASP.NET Core, including support for authentication, authorization, CORS, and OpenAPI support. You can find a complete Todo API implementation in **Listing 1** and tutorials for building minimal APIs in the ASP.NET Core docs. Also check out Brady Gaster's "Power Up your Power Apps with .NET 6 and Azure" elsewhere in this magazine for a complete end-to-end scenario based on minimal APIs.

More Developer Productivity

Building great Web apps requires rapid iteration. The faster you can make code changes and test them, the better. .NET 6 includes various improvements that make ASP.NET Core development faster, more iterative, and more fluid.

Listing 1: Minimal Todo API

```
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

var connectionString = builder.Configuration.GetConnectionString("Todos") ??
"Data Source=Todos.db";

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSqlite<TodoDbContext>(connectionString);
builder.Services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new() {
        Title = builder.Environment.ApplicationName,
        Version = "v1" });
});

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json",
$"{builder.Environment.ApplicationName} v1"));
}

app.MapFallback(() => Results.Redirect("/swagger"));

app.MapGet("/todos", async (TodoDbContext db) =>
{
    return await db.Todos.ToListAsync();
});

app.MapGet("/todos/{id}", async (TodoDbContext db, int id) =>
{
    return await db.Todos.FindAsync(id) is Todo todo ?
        Results.Ok(todo) :
        Results.NotFound();
});

app.MapPost("/todos", async (TodoDbContext db, Todo todo) =>
{
    await db.Todos.AddAsync(todo);
    await db.SaveChangesAsync();

    return Results.Created($"/todo/{todo.Id}", todo);
});

app.MapPut("/todos/{id}",
    async (TodoDbContext db, int id, Todo todo) =>
{
    if (id != todo.Id)
    {
        return Results.BadRequest();
    }

    if (!await db.Todos.AnyAsync(x => x.Id == id))
    {
        return Results.NotFound();
    }

    db.Update(todo);
    await db.SaveChangesAsync();

    return Results.Ok();
});


app.MapDelete("/todos/{id}",
    async (TodoDbContext db, int id) =>
{
    var todo = await db.Todos.FindAsync(id);
    if (todo is null)
    {
        return Results.NotFound();
    }

    db.Todos.Remove(todo);
    await db.SaveChangesAsync();

    return Results.Ok();
});

app.Run();
```

Faster Razor Compilation

.NET 6 includes optimizations across the .NET platform to speed up build times and improve app startup performance. For example, the Razor compiler was updated to use Roslyn Source Generators instead of its earlier two-phase approach, which makes building your MVC Views, Razor Pages, and Blazor components much faster. In many cases, compiling Razor files (.cshtml, .razor) is now over twice as fast as it was with .NET 5 (see **Figure 2**).

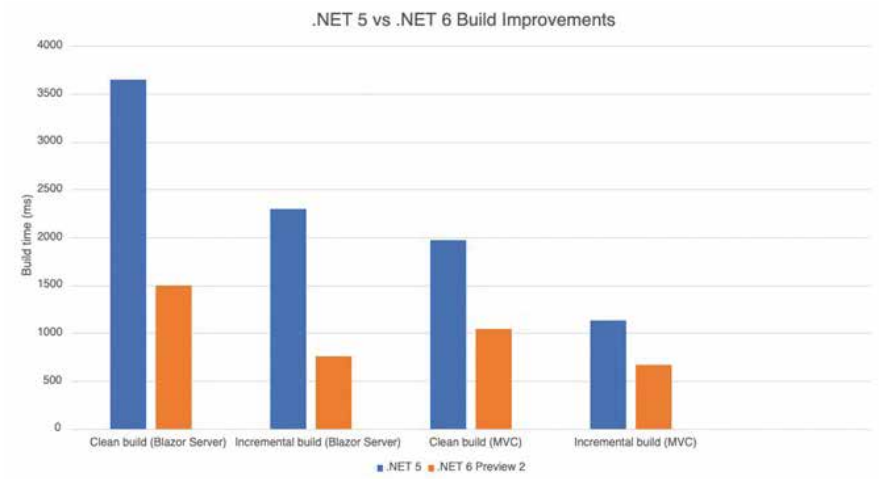

Figure 2: Faster Razor build times with .NET 6.

Hot Reload

.NET 6 adds support for .NET Hot Reload, which enables you to make changes to your app while it's running without having to restart it. The .NET tooling calculates the exact code delta that needs to be applied to the app based on your code changes and then applies the delta to the running app almost instantly. Because code changes are applied to the running app, any existing app state is preserved. .NET Hot Reload enables you to rapidly iterate on a specific part of the app with minimal disruption. .NET Hot Reload is designed to work with all the .NET 6 app models and it works great with all flavors of ASP.NET Core Web apps: MVC, Razor Pages, and Blazor.

.NET Hot Reload is enabled whenever you run an ASP.NET Core app using **dotnet watch**. Previously the **dotnet watch** command simply restarted your app and refreshed the browser whenever it detected a code file change. In .NET 6, **dotnet watch** first attempts to apply the changes to the running app using hot reload.

```
> dotnet watch
watch : Hot reload enabled.
For a list of supported edits, see https://aka.ms/dotnet/
hot-reload.
Press "Ctrl + Shift + R" to restart.
```

If the hot reload succeeds, you see the result of the changes in the app almost immediately:

```
watch : File changed:
C:\BlazorApp\Pages\Index.razor.
watch : Hot reload of changes succeeded.
```

Center spread .NET a
from MS

rtwork to come

Hot reload allows you to modify the markup for your views, pages, and components and see the visual impact in real time without losing any app state. You can quickly make updates to your Razor rendering logic, change the behavior of methods, and add new type members and attributes.

Not all changes can be safely applied at runtime. For example, you can't change the signature of a method. If a change can't be applied using hot reload, then the tooling gives you the option to restart the app to apply the change:

```
watch : File changed:
C:\BlazorApp\Pages\Counter.razor.
watch : Unable to apply hot reload because of
a rude edit. Rebuilding the app...
watch : Unable to handle changes using hot
reload.
watch : Do you want to restart your app -
Yes (y) / No (n) / Always (a) / Never (v)?
```

You can also hot reload CSS changes into the browser without having to refresh the page. CSS hot reload will detect when CSS changes have been made and then apply them to the live DOM. CSS hot reload works with normal CSS files and with scoped CSS files.

Hot Reload is deeply integrated into Visual Studio 2022. With Visual Studio 2022, you can easily apply changes to your running app while debugging and get great design-time feedback on what changes can be successfully applied. When making CSS changes, you can see the UI impact of your changes **as you type** with CSS auto sync. To learn all about the Visual Studio 2022 tooling for Hot Reload and all the other .NET productivity improvements be sure to check out Mika Dumont's article on "Visual Studio 2022 Productivity" elsewhere in this magazine.

MVC and Razor Pages Improvements

ASP.NET Core MVC has been the workhorse for ASP.NET Core Web apps for many years and it just keeps getting better. Here are some of the improvements coming to MVC and Razor Pages in .NET 6.

CSS Isolation for Pages and Views

CSS isolation was introduced in .NET 5 as a way to scope styles to a specific Blazor component. In .NET 6, the same functionality is enabled for MVC views and Razor pages.

View- and page-specific styles apply only to that view or page without polluting the global styles. Isolating styles to a specific page or view can make it easier to reason about the styles in your app and to avoid unintentional side effects as styles are added, updated, and composed from multiple sources.

You define view and page-specific styles using a .cshtml.css file that matches the name of the .cshtml file of the page or view. Any styles defined in the .cshtml.css file will only be applied to that specific view or page. ASP.NET Core achieves CSS isolation by rewriting the CSS selectors as part of the build so that they only match markup rendered by that view or page. ASP.NET Core then bundles together all the rewritten CSS files and makes the bundle available to the app as a static Web asset at the path [PROJECT NAME].styles.css.

IAsyncDisposable

You can now implement IAsyncDisposable on controllers, page models, and view components to asynchronously dispose of resources. By fully supporting async patterns, ASP.NET Core enables more efficient utilization of server resources.

Async Streaming

ASP.NET Core now supports using IAsyncEnumerable to asynchronously stream response data from controller actions. Returning an IAsyncEnumerable from an action no longer buffers the response content in memory before it gets sent. This helps reduce memory usage when returning large datasets that can be asynchronously enumerated.

Support for async streaming in ASP.NET Core in .NET 6 can make using Entity Framework Core with ASP.NET Core more efficient by fully leveraging the IAsyncEnumerable implementations in Entity Framework Core for querying the database. For example, the following code no longer buffers the product data into memory before sending the response:

```
public IActionResult GetProducts()
{
    return Ok(dbContext.Products);
}
```

Better Integration with JavaScript Frameworks

ASP.NET Core works great as a back-end for JavaScript-based apps written with popular frontend frameworks. The .NET SDK includes project templates for using ASP.NET Core with Angular and React. In .NET 6, Microsoft has improved how ASP.NET Core integrates with front-end JavaScript frameworks using a common pattern that can be applied to using ASP.NET Core with other JavaScript frameworks as well.

Most modern JavaScript frameworks come with command-line tooling for creating new apps and running them during development. In production, the built JavaScript app runs on a production Web server, like an ASP.NET Core app. In previous .NET releases, ASP.NET Core projects proxied requests for the JavaScript app to the JavaScript development server during development to preserve the JavaScript development experience. To enable this setup, ASP.NET Core apps had to include Angular- and React-specific components. Adding support for additional JavaScript frameworks meant building and maintaining additional integration components in ASP.NET Core as the JavaScript ecosystem evolves.

In .NET 6, things are flipped around. Now, the browser is pointed at the JavaScript development server and configures the development server to proxy API requests to the ASP.NET Core back-end. Most modern JavaScript development servers have built-in proxying support for precisely this sort of setup. The proxy configuration lives in the project instead of in framework code, which makes it trivial to adapt to other front-end JavaScript frameworks.

Microsoft has updated the built-in Angular and React templates to the latest versions (Angular 12 and React 17) and reconfigured the templates to use this new proxying setup. This should simplify single-page app development with ASP.NET Core and establish a pattern that can be used by the .NET community to integrate ASP.NET Core with more JavaScript frameworks.

Blazor Improvements

Of course, why write your Web app in JavaScript when you can build the entire thing with just .NET? Blazor is a

client-side Web UI framework included with ASP.NET Core that enables full stack Web development with .NET. .NET 6 includes support for both Blazor Server and Blazor WebAssembly apps, as well as new support for building hybrid native client apps using Blazor components. Regardless of how you decide to host your Blazor component, .NET 6 includes loads of new Blazor features that you can take advantage of.

WebAssembly Ahead-of-Time Compilation

Blazor WebAssembly now supports ahead-of-time (AOT) compilation, which compiles your .NET code directly to WebAssembly for a significant runtime performance boost.

Most Blazor WebAssemby apps today run using a .NET IL interpreter implemented in WebAssembly. Because the .NET code is interpreted, it generally runs much slower than it would on a normal .NET runtime. .NET WebAssembly AOT compilation addresses this runtime performance issue. The runtime performance improvement from WebAssembly AOT compilation can be quite dramatic for CPU intensive tasks. In some cases, code runs five to ten times faster than when interpreted!

.NET WebAssembly AOT compilation requires additional build tools that are installed as an optional .NET SDK workload. To install the .NET WebAssembly build tools, run the following command from an elevated command prompt:

```
dotnet workload install wasm-tools
```

To enable WebAssembly AOT compilation for your Blazor WebAssembly project, add the following property to your project file:

```
<RunAOTCompilation>true</RunAOTCompilation>
```

WebAssembly AOT compilation is performed when the project is published and generally takes a while: several minutes on small projects and potentially much longer for larger projects. Your app gets compiled to WebAssembly and bundled into the **dotnet.wasm** runtime bundle.

The download size of the published app with AOT compilation enabled is generally *larger* than the interpreted version, about two times bigger. .NET IL contains high-level instructions that must be expanded into native WebAssembly code. So, using WebAssembly AOT compilation trades off some load time performance for better runtime performance. Whether this tradeoff is worth it depends on your app. Many apps run interpreted just fine without the need for AOT compilation. Blazor WebAssembly apps that are particularly CPU-intensive will benefit the most from AOT compilation.

Smaller Download Size

Published Blazor WebAssembly apps are much smaller in .NET 6. In .NET 5, a minimal Blazor WebAssembly is about 1.7 MB when published. In .NET 6, it's now only 1.1 MB. This is thanks to some great new optimizations:

- **Smarter .NET IL trimming:** NET IL trimming is much improved in .NET 6, and the core framework libraries are more trimming friendly.
- **Runtime relinking**: In .NET 5, the .NET WebAssembly runtime was a fixed size. In .NET 6, the new .NET WebAssembly build tools support relinking the runtime to remove unused features. For example, if you use invariant globalization, tooling can remove a bunch of globalization code from the runtime.
- **Smaller JavaScript files**: Thanks to some great contributions for the .NET community, the JavaScript files that ship as part of ASP.NET Core and Blazor are much better optimized for size.

Smaller download sizes mean that your published Blazor WebAssembly apps load faster, especially on slower networks. It also mitigates the size impact of WebAssembly AOT compilation because you're already starting with a smaller app.

Error Boundaries

Blazor error boundaries provide a convenient way to handle exceptions within a component hierarchy. To define an error boundary, wrap the desired content with the new ErrorBoundary component. The ErrorBoundary component normally renders its child content, but when an unhandled exception is thrown, the ErrorBoundary renders some error UI instead.

For example, you can add a Counter component wrapped in an error boundary to the home page of a default Blazor app like this:

```
<ErrorBoundary>
    <Counter />
</ErrorBoundary>
```

The app continues to function as before but now the error boundary will handle unhandled exceptions. For example, you can update the Counter component to throw an exception if the count gets too big:

```
private void IncrementCount()
{
    if (currentCount >= 10)
    {
        throw new InvalidOperationException(
            "I've run out of fingers!");
    }
    currentCount++;
}
```

Now, if you click the counter too much, an unhandled exception is thrown, which gets handled by the error boundary by rendering some default error UI, as shown in **Figure 3**.

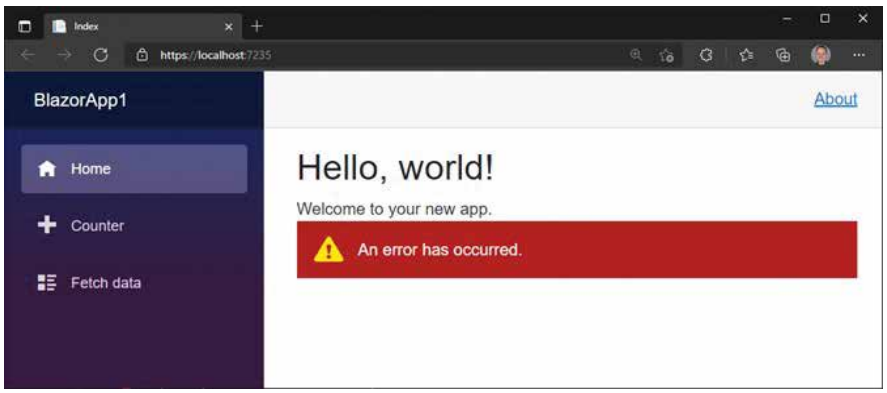

Figure 3: Default error UI provided by an error boundary.

By default, the ErrorBoundary component renders an empty div with a blazor-error-boundary CSS class for its error content. The colors, text, and icon for this default UI are all defined using CSS in the app, so you're free to customize them. You can also change the default error content by setting the ErrorContent property.

```
<ErrorBoundary>
    <ChildContent>
        <Counter />
    </ChildContent>
    <ErrorContent>
        <p class="my-error">
            Something broke. Sorry!
        </p>
    </ErrorContent>
</ErrorBoundary>
```

Preserve Prerendered State

Blazor apps can be prerendered from the server to speed up the perceived load time of the app. The prerendered HTML can immediately be rendered while the app is set up for interactivity in the background. Blazor is deeply integrated into ASP.NET Core, so Blazor components can be prerendered from any MVC View or Razor Page using the built-in component tag helper. The ability to run Blazor components on the client or the server is one of Blazor's key strengths.

However, any state that was used during prerendering on the server is generally lost and must be recreated when the app is loaded on the client. If any state is set up asynchronously on the client, the UI may flicker as the prerendered UI is replaced with temporary placeholders and then fully rendered again.

To solve this problem, you need a way to persist state used during prerendering and transfer it to the client for reuse. That's what the new preserve-component-state tag helper is for.

```
<component type="typeof(App)"
    render-mode="ServerPrerendered" />
<persist-component-state />
```

In your app, you control what state you want the preserve-component-state tag helper to persist by registering a callback with the new PersistentComponentState service. Your registered callback is called when state is about to be persisted into the prerendered page so that you can add to the persisted state. You then retrieve any persisted state when initializing your components. **Listing 2** shows an implementation of the FetchData component that uses persisted state for prerendering some weather data.

By initializing your components with the same state used during prerendering, any expensive initialization steps only need to be executed once. The newly rendered UI also matches the prerendered UI, so no flicker occurs.

Infer Generic Component Types from Ancestors

When using a generic Blazor component, like a Grid<TItem> or ListView<TItem>, Blazor can typically infer the generic type parameters based on the parameters passed to the component, so you don't have to explicitly specify them. In more sophisticated components, you might have multiple generic components that get used together where the type parameters are intended to match, like **Grid<TItem>** and **Column<TItem>.** In these composite scenarios, generic type parameters often need to be specified explicitly on child components, like this:

```
<Grid Items="@GetSales">
    <Column TItem="SaleRecord" Name="Product" />
    <Column TItem="SaleRecord" Name="Sales" />
</Grid>
```

In .NET 6, Blazor can now infer generic type parameters from ancestor components. Ancestor components opt-in to this behavior by cascading a named type parameter to descendants using the **[CascadingTypeParameter]** attribute. This attribute allows generic type inference to work automatically with descendants that have a type parameter with the same name.

For example, you can define Grid.razor components that look like this:

```
@typeparam TItem
@attribute [CascadingTypeParameter(
    nameof(TItem))]

...

@code {
    [Parameter]
    public IEnumerable<TItem>? Items
        { get; set; }

    [Parameter]
    public RenderFragment? ChildContent
        { get; set; }
}
```

And you can define Column.razor components that look like this:

```
@typeparam TItem

...

@code {
    [Parameter]
    public string? Title { get; set; }
}
```

You can then use the Grid and Column components without specifying any type parameters, like this:

```
<Grid Items="@GetSales()">
    <Column Title="Product" />
    <Column Title="Sales" />
</Grid>
```

Faster JavaScript Interoperability for Binary Data

Blazor is all about writing client Web apps using .NET. But sometimes you still need to use a little JavaScript. Maybe you want to reuse an existing JavaScript library or customize some low-level browser behavior. Blazor supports JavaScript interoperability (JS interop), and .NET 6 improves how JS interop works with binary data.

Blazor no longer base64-encodes binary data when doing JS interop, which makes transferring binary data across the interop boundary much more efficient. Blazor also now supports streaming binary data through JS interop using the new IJSStreamReference interface.

```
var dataRef = await JS
    .InvokeAsync<IJSStreamReference>("getData");
```

Listing 2: FetchData with persisted state from prerendering

```
@page "/fetchdata"

<PageTitle>Weather forecast</PageTitle>

@using BlazorApp1.Data
@inject WeatherForecastService ForecastService
@inject PersistentComponentState PersistentState

<h1>Weather forecast</h1>

<p>This component demonstrates fetching data from a service.</p>

@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <table class="table">
        <thead>
            <tr>
                <th>Date</th>
                <th>Temp. (C)</th>
                <th>Temp. (F)</th>
                <th>Summary</th>
            </tr>
        </thead>
        <tbody>
            @foreach (var forecast in forecasts)
            {
                <tr>
                    <td>@forecast.Date.ToShortDateString()</td>
                    <td>@forecast.TemperatureC</td>
                    <td>@forecast.TemperatureF</td>
                    <td>@forecast.Summary</td>
                </tr>
            }
        </tbody>
    </table>
}

@code {
    private WeatherForecast[]? forecasts;
    PersistingComponentStateSubscription persistingSubscription;

    protected override async Task OnInitializedAsync()
    {
        persistingSubscription = PersistentState
            .RegisterOnPersisting(PersistForecasts);
        if (!PersistentState.TryTakeFromJson<WeatherForecast[]>
("fetchdata", out forecasts))
        {
            forecasts = await ForecastService
                .GetForecastAsync(DateTime.Now);
        }
    }

    private Task PersistForecasts()
    {
        PersistentState.PersistAsJson("fetchdata", forecasts);
        return Task.CompletedTask;
    }
}
```

```
using var dataRefStream = await dataRef
    .OpenReadStreamAsync();

// Write JS Stream to disk
var outputPath = Path.Combine(
    Path.GetTempPath(), "file.txt");
using var outputFileStream =
    File.OpenWrite(outputPath);
await dataRefStream
    .CopyToAsync(outputFileStream);
```

Streaming data through JS interop is particularly useful in Blazor Server apps when you want to upload lots of data to the server. The InputFile component was updated to use these JS interop improvements to support much larger (>2GB) and faster file uploads.

Working with Query Strings

Blazor components can now receive parameters from the query string. To specify that a component parameter can come from the query string, apply the **[SupplyParameterFromQuery]** attribute in addition to the normal **[Parameter]** attribute:

```
[Parameter]
[SupplyParameterFromQuery]
public int? Page { get; set; }
```

You can also update the query string of the browser URL with new query string parameters using the new helper methods on NavigationManager:

```
var newUri = NavigationManager
    .GetUriWithQueryParameter("page", 3);
NavigationManager.NavigateTo(newUri);
```

Required Component Parameters

To indicate that a component parameter is required, apply the **[EditorRequired]** attribute:

```
[Parameter]
[EditorRequired]
public int IncrementAmount { get; set; }
```

If the user doesn't specify the required parameter when using the component, they'll get a build warning indicating that the parameter is missing. This isn't enforced at runtime, so you'll still need to deal with the possibility that the parameter wasn't set in your component implementation. But specifying required parameters does improve component usability.

Select Multiple

Blazor now provides the array of selected elements via ChangeEventArgs when handling the **onchange** event for a select element with the **multiple** attribute applied. Also, you can bind array values to the value attribute of a select element with the multiple attribute. The built-in InputSelect component also infers the multiple attribute when bound to an array.

DynamicComponent

DynamicComponent is a new built-in Blazor component that can be used to dynamically render a component specified by type.

```
<DynamicComponent Type="typeof(Counter)" />
```

Parameters can be passed to the rendered component using a dictionary:

```
<DynamicComponent Type="typeof(Counter)"
   Parameters="parameters" />

@code {
    Dictionary<string, object> parameters =
        new() { { "IncrementAmount", 10 } };
}
```

DynamicComponent is useful when you want to determine what component to render at runtime.

Render Root Components from JavaScript

What if you want to dynamically render a component at runtime from JavaScript? For example, you might want to add Blazor components to an existing JavaScript app. Top-level component in Blazor are called root components, and in earlier releases Blazor root components had to render when the app started up. In .NET 6, you can now dynamically render root components from JavaScript whenever you want.

To render a Blazor component from JavaScript, first register it for JavaScript rendering and assign it an identifier:

```
options.RootComponents
    .RegisterForJavaScript<Counter>(
        identifier: "counter");
```

You can then render the component from JavaScript into a container element using the registered identifier passing component parameters as needed:

```
let containerElement =
    document.getElementById('my-counter');
await Blazor.rootComponents.add(
    containerElement,
    'counter',
    { incrementAmount: 10 });
```

The ability to render root components from JavaScript enables all sorts of interesting scenarios, including building standards-based custom elements with Blazor. Experimental support for building custom elements is available using the Microsoft.AspNetCore.Components.CustomElements NuGet package. With this package installed, you can register root components as customer elements:

```
options.RootComponents
    .RegisterAsCustomElement<Counter>(
        "my-counter");
```

You can then use this custom element with any other Web framework you'd like. For example, here's how you would use this Blazor counter custom element in a React app:

```
<my-counter increment-amount={incrementAmount}>
</my-counter>
```

You can also now generate framework-specific JavaScript components from Blazor components for frameworks like Angular or React. The JavaScript component generation sample on GitHub shows how this can be done (see https://aka.ms/blazor-js-components). In this sample, you can attribute Blazor components to generate Angular or React component wrappers:

```
@*Generate an Angular component*@
@attribute [GenerateAngular]

@*Generate an React component*@
@attribute [GenerateReact]
```

You also register the Blazor components as Angular or React components:

```
options.RootComponents
    .RegisterForAngular<Counter>();

options.RootComponents
    .RegisterForReact<Counter>();
```

When the project gets built, it generates Angular and React components based on your Blazor components. You can use the generated Angular and React components like you would normally:

```
// Angular
<counter [incrementAmount]="incrementAmount">
</counter>

// React
<Counter incrementAmount={incrementAmount}>
</Counter>
```

Hopefully this captures your imagination about what's now possible with Blazor components in JavaScript. Microsoft is excited to see what the community does with this feature!

Modify HTML Head Content

Blazor now has built-in support for modifying HTML head-element content from components, including setting the title and adding meta elements.

To specify the page's title from a component, use the new PageTitle component.

```
<PageTitle>Counter</PageTitle>
```

To add other content to the head element, use the new HeadContent component:

```
<HeadContent>
    <meta name="description" content="@content">
</HeadContent>
```

To enable the functionality provided by PageTitle and HeadContent, you need to add a HeadOutlet root component to your app that appends to the head element. In Blazor WebAssembly, you can register the HeadOutlet component like this:

```
builder.RootComponents
    .Add<HeadOutlet>("head::after");
```

In Blazor Server, the setup is slightly more involved. To support prerendering, the App root component needs to be rendered **before** the HeadOutlet. This is typically accomplished using MVC-style layouts. Check out the updated Blazor Server template in .NET 6 to see how to set this up.

.NET MAUI Blazor Apps

Blazor enables building client-side Web UI with .NET, but sometimes you need more than what the Web platform offers. Sometimes you need full access to the native capabilities of the device. You can now host Blazor components in .NET MAUI apps to build cross-platform native apps using Web UI. The components run natively in the .NET process and render Web UI to an embedded Web view control using a local interop channel. This hybrid approach gives you the best of native

and the Web. Your components can access native functionality through the .NET platform, and they render standard Web UI. .NET MAUI Blazor apps can run anywhere .NET MAUI can (Windows, Mac, iOS, and Android). If you're not using .NET MAUI yet, you can also add Blazor components to your Windows Forms and WPF apps to start building UI in your Windows desktop apps that can be reused in .NET MAUI apps or on the Web. Check out Ed Charbeneau's article "Blazor Hybrid Web Apps with .NET MAUI" elsewhere in this magazine for all the details on how .NET MAUI and Blazor can be used together.

ASP.NET Core Runtime Improvements

Under the hood, ASP.NET Core apps are powered by versatile runtime that provides performance, reliability, and security. ASP.NET Core in .NET 6 includes many runtime improvements that will help to supercharge your Web apps.

Performance

Performance is an important part of every ASP.NET Core release. Better performance means better server resource utilization and reduced hosting costs. .NET 6 includes performance improvements at every level of the framework. ASP.NET Core in .NET 6 is the fastest Web framework Microsoft has ever shipped!

Here are some examples of the ASP.NET Core performance improvements in .NET 6.

- Middleware request throughput is ~5% faster in .NET 6.
- MVC on Linux is ~12%, thanks to faster logging.
- The new minimal APIs offer twice the throughput of API controllers without compromising on features.
- HTTPS connections use ~40% less memory, thanks to zero byte reads.
- Protobuf serialization is ~20% faster with .NET 6.

> ASP.NET Core in .NET 6 is the fastest Web framework Microsoft has ever shipped!

HTTP/3

HTTP/3 is the third and upcoming major version of HTTP. HTTP/3 has the same semantics as earlier HTTP versions, but introduces a new transport based on UDP called QUIC. HTTP/3 is still in the process of being standardized but has already gained significant adoption.

HTTP/3 and QUIC have several benefits compared to older HTTP versions:

- **Faster initial response time:** HTTP/3 and QUIC requires fewer roundtrips to establish a connection, so the first request reaches the server faster.
- **Avoid head-of-line blocking.** HTTP/2 multiplexes multiple requests on a TCP connection, so packet loss affects all requests on a given connection. QUIC provides native multiplexing, so lost packets only impact requests with lost data.
- **Transition between networks.** HTTP/3 allows the app or Web browser to seamlessly continue when a network changes.

.NET 6 introduces preview support for HTTP/3 in Kestrel, the built-in ASP.NET Core Web server. HTTP/3 support in ASP.NET Core is a preview feature because the specification is being standardized. Kestrel also doesn't support the network transitions feature of HTTP/3 in .NET 6, but Microsoft will explore adding it in a future .NET release.

To try out HTTP/3 with Kestrel, first enable support for preview features in your project by setting the following property:

```
<EnablePreviewFeatures>
  True
</EnablePreviewFeatures>
```

Then configure Kestrel to use HTTP/3:

```
using Microsoft.AspNetCore.Server.Kestrel.Core;

var builder =
    WebApplication.CreateBuilder(args);

builder.WebHost
    .ConfigureKestrel((context, options) =>
{
    options.ListenAnyIP(5001, listenOptions =>
    {
        listenOptions.UseHttps();
        listenOptions.Protocols =
            HttpProtocols.Http1AndHttp2AndHttp3;
    });
});
```

Browsers are finicky about connecting to localhost over HTTP/3, so you'll need to deploy the server to a separate computer to try it out.

And All the Rest

There are many more ASP.NET Core features and improvements in .NET 6:

- Support for Bootstrap 5
- Collocate JavaScript modules with views, pages, and components (.cshtml.js, .razor.js)
- Support for custom event arguments in Blazor
- Support for native dependencies in Blazor WebAssembly
- Blazor JavaScript initializers
- WebSocket compression
- HTTP and W3C logging
- Shadow-copying with IIS
- gRPC client support for load balancing and retries
- Etc.

Be sure to check the .NET 6 release notes for the full list of everything that's new. You can also find additional details on all these new features in the ASP.NET Core docs: https://docs.asp.net.

I hope you've enjoyed learning about all the great new functionality in ASP.NET Core now available with .NET 6. On behalf of the ASP.NET team, we look forward to hearing about your experiences with this momentous release.

Daniel Roth

What Are Custom Elements?

Custom elements are part of the HTML standard and "provide a way for authors to build their own fully-featured DOM elements." Because custom elements are based purely on open Web standards, they can be used with any modern Web framework. The ability to build custom elements with Blazor means you can reuse your Blazor and .NET investments across all your Web projects.

EF Core 6: Fulfilling the Bucket List

Ahh another year, another update to EF Core. Lucky us! Remember when Microsoft first released Entity Framework in 2008 and many worried that it would be yet another short-lived data access platform from Microsoft? (Note that ADO.NET is still widely used, maintained, and supported!) Well, it's been 13 years, including EF's transition to EF Core, and it just keeps getting better and better.

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.

You may have heard me refer to EF Core 3 as the "breaking changes edition." In reality, those breaking changes set EF Core up for the future, as I relayed in "Entity Framework Core 3.0: A Foundation for the Future" (http://codemag.com/Article/1911062), covering the highlights of those changes. The next release, EF Core 5 (following the numbering system of .NET 5), built on that foundation. I also wrote about this version in "EF Core 5: Building on the Foundation" (codemag.com/Article/2010042/EF-Core-5-Building-on-the-Foundation).

And now here comes EF Core 6. My perspective on it is that the team has been working on their (and your) bucket list! Digging into improvements to EF Core that they've been wanting to get to for quite a long time but there were more pressing features and fixes to focus on. But they didn't only work on their own goals. In advance of planning, the team put out a survey to gauge usage of existing versions of EF and EF Core and what they should focus on going forward. They presented the results from about 4000 developers in this January 2021 Community Standup: https://www.youtube.com/watch?v=IiAS61uVDqE. The survey was available before EF Core 5 was released and into only its first few months. So it was not surprising that EF Core 5 trailed behind EF Core 3 and EF 6. There were a substantial number of devs still using EF6. This makes a lot of sense to me for the many legacy apps out there: If it ain't broke, don't fix it.

The team has again been incredibly transparent about their goals and progress. In the docs (https://docs.microsoft.com/en-us/ef/core/what-is-new/ef-core-6.0/plan), they shared their high-level plan, updates of what's new for each preview, and their (very short) list of breaking changes. Late in the development cycle, it was sad to see the team come to grips with some plans they had to give up on for this version of EF Core 6 and change their GitHub milestones to "punted for EF Core 6." But as developers, we all understand how this goes. The GitHub repo tracks every issue with detail and tags which preview/version the issue is tied to. There are also detailed bi-weekly status updates at https://github.com/dotnet/efcore/issues/23884. And triggered by the lockdown, the team got online every few weeks with the EF Core Community Standups, showing us what they're working on and inviting guests to share their additions to the EF Core ecosystem. Here's a shortcut to the full playlist of their standups where you can also keep up with upcoming shows (https://bit.ly/EFCoreStandups).

There were monthly releases of preview builds available on NuGet (https://www.nuget.org/packages/Microsoft.EntityFrameworkCore) And GitHub repository's readme (https://github.com/dotnet/efcore#readme) led to detailed instructions on how to work with daily builds if you wanted to do that.

Although there have been a lot of small changes, this article will cover some of the most notable and most impactful changes that you'll find in EF Core 6.

Huge Gains in Query Performance

Top on the EF Core wish list for so many developers and the EF team was query performance. This has always been a criticized problem with EF and EF Core even though it has dramatically improved over the years. Jon P Smith, EF Core book author and a real expert, explains that the work he does for performance tuning is "not because EF Core is bad at performance, but because real-world application has a lot of data and lots of relationships (often hierarchical), and it takes some extra work to get the performance the client needs." Even with lots of great strategies for perf tuning EF, the team set out to really raise the bar on performance with EF Core 6 with laser-like focus and with Shay Rojansky leading up the effort.

An early step was to set a goal to improve EF Core's standing in a commonly known software industry standard benchmark for measuring the performance—the TechEmpower Web Framework Benchmarks (https://www.techempower.com/benchmarks/). The benchmark comes with tweakable source code and is used by hundreds of frameworks, including ASP.NET and EF Core. Among the comparisons are some that focus on data access and one in particular, called Fortunes, which "exercises the ORM, database connectivity, dynamic-size collections, sorting, server-side templates, XSS countermeasures, and character encoding." I was more than surprised to discover that there are almost 450 ORMs on the list—although only 30 are .NET ORMs! The TechEmpower benchmark provides a very specific set of standards (and source code) for how to set up and run the tests, including standard hardware requirements. The version that the ASP.NET team uses is on GitHub at https://github.com/aspnet/Benchmarks.

For EF Core 6, most of the performance improvements were aimed at non-tracking queries, although tracked queries certainly benefited. Rojansky tells me that they hope to deepen their focus on change-tracked queries in EF Core 7.

EF Core is often compared in performance to Dapper, a widely used micro-ORM for .NET built by the folks over at StackOverflow. At the start, there was a 55% gap between Dapper's rows returned per second and that of EF Core 5. There was a series of categories for improvements. Before even looking at the EF Core APIs, it turned out that the benchmark itself was begging for some tweaks to make comparisons more equitable. Tuning EF's DbContext pooling (23% improvement), PostgreSQL connection pooling (2.8% improvement), and requiring results rather than wasting time with null checks (1.7% improvement) already had a significant impact. Another interesting benefit was switching the benchmark app to use .NET 6 instead of .NET 5 (9.8% improvement). Finally, it was time to dig into fine tuning EF Core itself. Changes were made to how logging works, how related data is tracked, changing concurrency detection to Opt-In and a few other tweaks added up to another sig-

nificant gain. (Let's not forget shout outs to Nino Floris and Nikita Kazmin for contributions to this work.)

In the end, the gap between EF Core and Dapper was reduced from 55% to 4.5% and the overall speed of EF Core's queries based on the Fortunes benchmark improved by 70%.

This is truly commendable work and the team is thrilled to finally have had time to focus on this. You can read a detailed blog post by Rojansky at https://devblogs.microsoft.com/dotnet/announcing-entity-framework-core-6-0-preview-4-performance-edition/ and see how all of this was put together in this GitHub issue: https://github.com/dotnet/efcore/issues/23611.

Improved Startup Performance with Compiled Models

Another area for performance improvement that was highlighted by the survey was the need to pre-compile the models described by a DbContext. This goes all the way back to the very first version of Entity Framework; there were some mechanisms for this over the years in EF but not in EF Core. And "startup" is not exactly the correct word. The basic workflow about how EF gets going in an application hasn't changed much over the years. At runtime, EF has to read the DbContext and relevant entities along with any data annotations and fluent configurations to build an in-memory version of the data model. This doesn't happen when you first instantiate that context, but the first time you ask the context to do something. And that only happens once per application instance. How many people have "tested" EF performance by creating an app that instantiates a context, does ONE thing, and then exits? And then writes a rant about the terrible performance of EF? Well, that's because they're incurring the startup cost of that context every single time.

With a small model in an application—or perhaps a variety of small models—that initial startup cost will most likely be undetectable. But you may realize some benefit from compiled models in serverless apps with multiple instances, on devices with minimal resources or even when repeatedly debugging your app while you're working on it.

The EF team ran a variety of performance tests using a rather large and complex model with not only a lot of entities, but a lot of relationships and a lot of non-conventional configurations. In one of the EF Community Standups, Arthur Vickers demoed an app that leverages BenchmarkDotNet (https://benchmarkdotnet.org) to iterate the calls, then gather and report the timings. The startup time for that first use of a DbContext ran a little more than 10 times faster when using the compiled model than when simply allowing EF Core to work out the model at runtime.

I'll show you how easy it is to pre-compile a model and have your app use that. There are two steps: compiling the model and then using the compiled model.

Compiling your model is simply a matter of running a CLI command against your DbContext.

In the CLI, the command is:

```
dotnet ef dbcontext optimize
```

The PowerShell version of the command for use in Visual Studio's Package Manager Console is

```
Optimize-DbContext
```

By default, this creates a CompiledModels folder and generates the compiled model files in that folder. You can specify your own folder name with the **–outputdir** parameter, if you prefer. There are files for each entity describing all its configurations in one place, a file for the model itself, and a file that exposes the logic for building that particular model with its entities. In my case, those files are named AddressEntityType, PersonEntityType, PersonContextModel, and PersonContextModelBuilder.

Now, all of these new classes are part of your project. The last puzzle piece is to let the DbContext know to just use a compiled model instance rather than go through the process of reading all of the various sources of info it normally uses to build up its understanding of the model. You'll do this in the context's OnConfiguring class the DbContextOptionsBuilder. UseModel, passing in the type defined in those final two files.

```
optionsBuilder
  .UseModel(PeopleContextModel.Instance)
  .UseSqlite("Data Source=MyDatabase.db");
```

This essentially short circuits the OnModelCreating method. The parameter is the Instance property of the generated model file. Internally, EF Core calls the Initialize method and triggers the fast runtime creation of the model using the streamlined classes.

You can watch Vickers demo his own sample app (https://github.com/ajcvickers/CompiledModelsDemo) in the Community Standup from EF Core team (https://youtu.be/XdhX3iLXAPk). This is where I initiated my own education on compiled models. Additionally, you can hear team members Rojansky and Andriy Svyrid talk about additional benefits, starting at about 40 minutes, in the video.

Update Databases with Stand-Alone Executable Migrations

Migrations has been a key feature for EF's "code first" support since the early days. You define the domain models for your software and migrations and then determine how to apply those models and changes to the database schema. The API for migrations is dependent on EF and .NET APIs. Executing them in your CI/CD pipeline or just outside of development is tricky. You do have runtime methods to run migrations, such as the Database.Migrate() method. However, this comes with a critical caveat. If your app is a Web or serverless app (or API) where you may have multiple instances that point to a common database, there's a chance of hitting some damaging race conditions if multiple instances are concurrently attempting to migrate that database. The same problem exists if you're using Docker containers to manage application load.

A typical path for solving this problem is to let migrations generate SQL for you and use another mechanism for executing the SQL, for example, on production databases.

Migrations Bundles were introduced in EF Core 6 to provide another tool in the DevOps quiver to solve this problem.

A Migration Bundle is a self-contained executable that can be run on a variety of CLIs: PowerShell, Docker, SSH, and more. It only requires that the .NET Runtime be available, but you don't have to install the SDK or any of the EF Core packages. Further, you have the option of creating a totally standalone version. Therefore, you can run it outside of your application and have it as an explicit step in your pipelines.

Creating the bundles is fairly simple, and, in fact, is merely another command in the Migrations API.

In the **dotnet** CLI, the command extends from migrations.

```
dotnet ef migrations bundle
```

In Visual Studio's Package Manager Console it's:

```
Bundle-Migration
```

The command has additional parameters to force the runtime to be included in the exe (making it truly self-contained) as well as familiar parameters to specify the DbContext, project, and startup project to use.

The resulting file is an executable named efbundle (with an extension driven by the operating system you're running on) and is dropped into project's path. You can specify the name of the resulting file with the **-o/--output** parameter.

The bundle file is idempotent. It includes all of the migrations, but it checks the database and doesn't run any of the migrations that have already been applied.

Your first test will most likely be directly on your development computer after creating the bundle. In my case, that was initially in the CLI on my MacBook where I had just called the migrations bundle CLI command—in my project's folder. On macOS, in the directory where the bundle and the project live, you'll run the bundle by typing **./efbundle**. That tells the OS to run it from the current directory, not by traversing the system's PATH file. In Windows, you can just run bundle.

Bundle, in this case, will be able to find the connection string for the database as if I were running **dotnet ef database update** whether it's defined in a DbContext file, a startup configuration, or appsettings file. In fact, efbundle is running the database update command on your behalf.

Listing 1: TSQL created by a migration defining a temporal table

```
DECLARE @historyTableSchema sysname = SCHEMA_NAME()
EXEC(N'CREATE TABLE [People] (
    [Id] uniqueidentifier NOT NULL,
    [FirstName] nvarchar(max) NULL,
    [LastName] nvarchar(max) NULL,
    [MiddleName] nvarchar(max) NULL,
    [PeriodEnd] datetime2 GENERATED ALWAYS AS ROW END NOT NULL,
    [PeriodStart] datetime2 GENERATED ALWAYS AS ROW START
      NOT NULL,
    CONSTRAINT [PK_People] PRIMARY KEY ([Id]),
    PERIOD FOR SYSTEM_TIME([PeriodStart], [PeriodEnd])
) WITH (SYSTEM_VERSIONING = ON (HISTORY_TABLE = [' + @historyTableSchema + N'].
[PersonHistory]))');
GO
```

In your production or CI/CD pipeline you'll more likely want to pass in the **--connection** parameter.

```
./efbundle --connection "DataSource=xyz.db"
```

This example is still calling a build from the command line. But you can extrapolate to your DevOps tool of choice, like calling efbundle from, for example, a Dockerfile that's getting its connection string from a Docker env variable.

Support for Temporal Tables

Support for what? Yep, I'd literally never heard of (or perhaps remembered hearing of) temporal tables until this feature bubbled up in the EF Core 6 plans. But now that I'm aware of them, I can see why this was a highly requested feature for EF Core! Temporal tables are described in the SQL standards and have implementations in several databases, such as MariaDB, Oracle, PostgreSQL, and SQL Server. Interestingly, IBM DB2 implemented their own twist on temporal tables. SQL Server has supported them since SQL Server 2016 and refers to them as "system-versioned temporal tables."

Temporal tables are an automated way for a database to provide audit trails. When you designate a table as a temporal table, every time any data is changed in that table, the data that it's replacing is automatically stored in something that's akin to a sub-table, which is the temporal table's history component. The temporal table must include two date columns (in SQL Server, these must be datetime2) that signify the start and end moments when the values in that row were true. They are a demarcated as "System Time" columns. Not only will those always be tracked in the main table, but it's transferred to the history table as well. You can learn more about temporal tables from the SQL Server perspective at docs.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables. And thanks to EF Core, you don't have to worry about setting this up! Migrations will take care of it, as you'll read further on.

The main table always has the current state and is no different than any other table in that regard. But it's possible to query that table and include results from the historical data. This means you can find out what the state of data is at any given point in time. In SQL Server, there's a FOR_SYSTEM_TIME clause with a number of sub-clauses that triggers SQL Server to involve the history table.

EF Core 6 supports temporal tables in two ways. The first is for configuration. If you flag an entity as mapping to a temporal table, this triggers migrations to create the extra table columns and history table.

The mapping is configured as a parameter of the ToTable mapping with an IsTemporal method:

```
modelBuilder.Entity<Person>()
  .ToTable(tb => tb.IsTemporal());
```

The migration created based on this automatically adds in the datetime2 properties (PeriodEnd and PeriodStart) as shadow properties, along with a number of relevant annotations. There's also a set of annotations on the table, including one for specifying a history table. You can see the migration file in the article's downloads.

Listing 1 shows the TSQL for generated by the migration, which creates the People temporal table. Note that by the final release of EF Core 6, there may be a slight change to this, with the PeriodStart an PeriodEnd columns being flagged as HIDDEN, thereby a SELECT * query would not return those columns.

Letting migrations create the table, you can see how it works in **Figure 1**. The People table is tagged as a "System-Versioned" table and it contains a sub-table called PersonHistory, which is tagged as a History table. The columns (not shown) in the history table are an exact match of the People table columns. Having migrations set this all up for you is truly a convenience.

Also note that the names for the tracking columns and history table are defaults. There are additional mappings that let you configure the column and table names as well.

Now, onto the data. As you update or delete data from the People table, SQL Server sets the values of the PeriodStart and PeriodEnd columns accordingly and adds a new row to PeopleHistory with the state of the data before those changes. It also sets the PeriodStart and PeriodEnd columns in the history table.

Queries that involve temporal data are referred to as "time travel" queries. Note that you don't explicitly query that History table. Instead, SQL Server includes it as needed.

Here's a simple TSQL query example for you, where I'm also explicitly pulling in the temporal data columns because I'm a curious cat:

```
SELECT TOP (1000) [Id],[FirstName],[LastName],
   [MiddleName], [PeriodEnd],[PeriodStart]
FROM [TemporalTest].[dbo].[People]
FOR System_Time AS OF '2021-08-19 18:19:00'
```

When the main table data is more recent than the System_time I requested, SQL Server also looks in the history table. Fiddling with different System_Time predicate values, I can see different variations of the results data as I add and edit them along the way. I can even see data that was deleted after the AS OF date.

And now to EF Core's queries. There are new LINQ methods and expressions to support temporal tables.

- TemporalAsOf
- TemporalAll
- TemporalBetween
- TemporalFromTo
- TemporalContainedIn

The following query uses TemporalAsOf to return the People data as of the same time as I did using SQL, time traveling back to the state as of August 19, 2021 at 18:19:00 UTC.

```
var date=DateTime.Parse("2021-08-19 18:19:00");
var result=
  _context.People.TemporalAsOf(date).ToList();
```

The log shows that EF Core and the provider transformed this into the following TSQL, essentially the same that I had hand-coded.

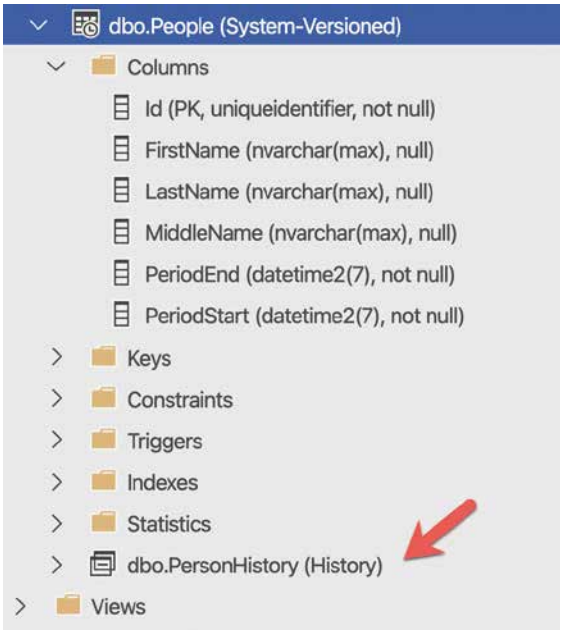

Figure 1: The schema of a temporal table as displayed in Azure Data Studio

```
SELECT [p].[Id], [p].[FirstName],
 [p].[LastName], [p].[MiddleName],
 [p].[PeriodEnd], [p].[PeriodStart]
FROM [People]
FOR SYSTEM_TIME
 AS OF '2021-08-19T18:19:00.0000000'
AS [p]
```

The result contained the same data I saw before, including the row that had subsequently been deleted and the original data prior to any edits.

This support makes it fairly easy to leverage temporal tables, thanks to migrations creating the proper schema and LINQ creating the relevant SQL. I can see why those who've been using temporal tables for years had surfaced this as one of the more highly requested features for EF Core.

There are a few important notes to keep in mind. First, to avoid side effects, some of EF Core's temporal LINQ methods inject AsNoTracking into the query because, as Rojansky explains, it's impossible to track multiple instances of the same data. However, the TemporalAsOf method is able to return tracked results. Second, these methods are only available on DbSet. Third, EF Core will treat the PeriodEnd and PeriodStart details as shadow properties allowing you to always force them to return using EF.Property() in a query. And finally, the TemporalAsOf method will propagate to any related entities (mapped to temporal tables) in a query for example if they are included or projected.

Bulk Conventions and Bulk Value Conversions

Specifying a convention for a particular type in your model has been possible but a little cumbersome and not at all discoverable. You had to drill into the model's metadata in the OnModelBuilding method.

```
foreach (var e in
 modelBuilder.Model.GetEntityTypes())
{
  foreach (var prop in e.GetProperties ()
   .Where (p => p.ClrType == typeof (string)))
  {
    prop.SetColumnType ("nvarchar(100)");
  }
}
```

The team revamped how models are configured under the covers, referred to as pre-convention model configuration (https://github.com/dotnet/efcore/issues/12229). In doing so, they were able to simplify your interaction with the APIs. With that change, you now have a way to define "bulk" conventions that's so much simpler that I literally found it heart-warming.

Note that custom conventions didn't make it into EF Core 6 but we're likely to get them in the next version. Read more in this GitHub issue: https://github.com/dotnet/efcore/issues/214.

The revamped model builder gives DbContext a new virtual method called ConfigureConventions along with a ModelConfiguratonBuilder class. This new class has methods for bulk configurations for properties, as well as some other methods such as IgnoreAny. ModelConfigurationBuilder.Properties exposes a number of configurations similar to ones you have via ModelBuilder.Entity().Properties.

You need to override the ConfigureConventions method, as you do with OnModelBuilding and OnConfiguring. In this case, I'll use Properties.HaveColumnType, which works like HasColumnType but more generically.

```
protected override void ConfigureConventions
 (ModelConfigurationBuilder configurationBuilder)
{
  configurationBuilder.Properties<string>()
    .HaveColumnType ("nvarchar(100)");
}
```

The other methods for configuring properties are AreFixedLength, AreUnicode, HaveAnnotation, HaveConverson, HaveMaxLength, HavePrecision and UseCollation.

From that list, I want to call out the HaveConversion method. This is related to Value Converters that were introduced in EF Core 2.1. In fact, I wrote about them in my CODE Focus article about EF Core 2.1 (https://www.codemag.com/article/1807071). Value converters allow you to configure mappings for CLR (or even your own) types that don't have direct translations to data types. There are a number of ways to express a conversion, but one frustrating drawback has been that you could only define a conversion for a single property. If you have many properties of the same type throughout your model, you have to write an explicit conversion for every single one. For comparison, here are two examples for converting individual properties.

```
modelBuilder.Entity<Address>()
 .Property(a=>a.AddressType)
 .HasConversion<string>();
modelBuilder.Entity<Address>()
 .Property(ad=>ad.StructureColor)
 .HasConversion(c=>c.ToString(),
               s=>Color.FromName(s));
```

The first persists an enum (AddressType) as a string rather than the conventional mapping of an integer, e.g., "Home" instead of 1. The second persists a property that's a System.Drawing.Color as the string name of the color and then transforms that string back to a Color type when it's read from the database. Having such an easy way to persist Color was a revelation when value converters were first introduced.

Now with configuration builders, you can be sure that any AddressType enum found throughout your model will get persisted as a string.

```
configurationBuilder.Properties<AddressType> ()
 .HaveConversion<string> ();
```

The signature I used for transforming the color, where I pass in an expression describing how to save the data and another for how to materialize the data from the database, isn't valid with HaveConversion. And that's for an interesting reason, as explained to me by EF team member, Andriy Svyryd. Even if the API exposed it, the precompiled model feature won't be able to read it. Instead, you need to use a more explicit path, which means building a custom ValueConverter. I'll include the code for my custom ColorToStringConverter class in the article's download.

With that class in play, I can now add another configuration to ConfigureConventions to leverage the new ColorToStringConverter class, ensuring that all Color properties throughout the model will be persisted as strings.

```
configurationBuilder.Properties<Color>()
 .HaveConversion<ColorToStringConverter>();
```

If you have a particular property that shouldn't follow the that bulk conversion rule, you can specify it as a null conversion in this way:

```
modelBuilder.Entity<Address>()
 .Property(ad=>ad.SomeProperty)
 .HasConversion((ValueConverter?)null);
```

Be sure not to do that in the case of properties like Color that have no conventional way to map to data types.

I'll share one last note about type mapping for a less common use case. The case is when you're building a query using a custom type that EF Core can't map to a data type. The DefaultTypeMapping method helps solve this. For an example of how to use this, check out the test named "Can_use_custom_converters_without_prop" in this functional test class (https://bit.ly/TypeMapTest) in the EF Core GitHub repository.

More EF6 Parity for Fans of GroupBy in Queries

The team is working to narrow the gap between EF6 and EF Core to make it easier for developers to transition old applications if needed or apply their existing knowledge to new ones. In the category of querying, GroupBy has not been given quite as much love since the early days of

EF Core. But there are now three additional query capabilities involving GroupBy and aggregates that are part of EF Core 6.

The first is that if you have navigations defined on our entities and you want to drill into those navigation objects after grouping, EF Core was unable to achieve that. The workaround was to break your query up and then pull the results together after the fact. Now it's possible to achieve that. For example, imagine that you have a model with authors and books with a one-to-many relationship between an author and their multiple books. (There are no co-authors in this case).

You might want to see if there's a pattern between author first names and their likelihood to write about .NET. Therefore, you want to write a query of authors, include books, and see how many of their books have the word ".NET" in the title.

The query:

```
var groupedAuthors=
 _context.Authors.Include(a=>a.Books)
 .GroupBy(a=>a.FirstName)
 .Select(g=>new
   {g.Key,AuthorCount=g.Count(),
    dotNetBooks=g.Sum(a=>a.Books
     .Count(b=>b.Title.Contains(".NET")))
   }).ToList();
```

This query fails in earlier versions of EF Core because the GroupBy would not have provided the properties of the related data (Books) to be used by the Select method. The exception suggests revising the query to perform some of the work in a client-side query. In EF Core 6, it will now succeed. And the results of this query may highlight a surprising number of authors named Scott who have quite a few books on .NET.

There are two other GroupBy features that were possible in EF6 but haven't been in EF Core until now.

- The ability to select Top N from a group.
- Using FirstOrDefault on groups.

You can read details of these in this GitHub issue if you're interested (https://github.com/dotnet/efcore/pull/25495). Or just go forth and group!

A More Intuitive CosmosDB Provider

I've saved the best for last because this was fun to explore. Improvements to the provider for accessing Cosmos DB were high on the wish list from the community. We've had the Cosmos provider since EF Core 3, so in case you're asking yourself "but why does an ORM need to work with a non-relational data store?" I answered that question in the EF Core 3 article referenced above. However, developers have asked why this non-relational provider got more love in this version than any of the relational providers. One reason, shared by Jeremy Likness, Sr. Program Manager for .NET Data at Microsoft, is that the Cosmos DB team gets "inundated with requests" for features in the Azure SDK from developers who prefer the EF Core APIs. In fact, if you look back at that EF Core 3 article, you'll see that my first reaction to the Cosmos provider even back then was "wow, it's so much easier to use than the SDK!". Another interesting data point from the survey was that MongoDB was the most requested provider. By investing in the Cosmos provider, the team is also paving the way for other non-relational providers. Yes, these are interesting points, but more interesting is that some of the new support for the provider in EF Core 6 better supports smart modeling for document database storage.

Listing 2: Logging output from EF Core 6's Cosmos provider

```
dbug: 08/25/2021 14:22:18.213 CoreEventId.QueryExecutionPlanned[10107] (Microsoft.
EntityFrameworkCore.Query)
      Generated query execution expression:
      'queryContext => new QueryingEnumerable<Person>(
          (CosmosQueryContext)queryContext,
          SqlExpressionFactory,
          QuerySqlGeneratorFactory,
          [Cosmos.Query.Internal.SelectExpression],
          Func<QueryContext, JObject, Person>,
          PeopleContext,
          null,
          False,
          True
      )'
info: 08/25/2021 14:22:18.273 CosmosEventId.ExecutingSqlQuery
 [30100] (Microsoft.EntityFrameworkCore.Database.Command)
      Executing SQL query for container 'PeopleContext' in
  partition '?' [Parameters=[]]
      SELECT c
      FROM root c
      WHERE (c["Discriminator"] = "Person")
info: 08/25/2021 14:22:19.292 CosmosEventId.ExecutedReadNext
 [30102] (Microsoft.EntityFrameworkCore.Database.Command)
      Executed ReadNext (983.9663 ms, 2.86 RU) ActivityId=
 '5b500af5-77eb-4513-a14c-bd5a00d45c4c',
 Container='PeopleContext', Partition='?', Parameters=[]
      SELECT c
      FROM root c
      WHERE (c["Discriminator"] = "Person")
dbug: 08/25/2021 14:22:22.535 CoreEventId.ContextDisposed[10407] (Microsoft.
EntityFrameworkCore.Infrastructure)
      'PeopleContext' disposed.
```

Richer Logging Details

First, I want to point out the new and improved support for logging events on the CosmosDB database. People requested details that would help them gain better insight into their resource use.

For example, **Listing 2** shows logging output (filtered on LogLevel.Informaton) in EF Core 6 for a query executed with the CosmosDB provider.

The first section showing the QueryExecutionPlanned event is the same as in EF Core 5. However, the following two sections showing the database command are new. EF Core 6 is more closely aligned with the logging we've come to expect from the other relational providers. But looking more closely, notice the ReadNext details. That includes not only the total round-trip time on your database but also the RUs (resource units) that are key to how your Azure bill is calculated.

The provider has additional new capabilities, such as raw queries with CosmosFromSQL, as well as some fixes to its behavior. And I happen to agree with Jeremy Likness about two other favorites: implicit ownership and support for primitive

collections and dictionaries. So I'll show you these two features and you can read about other Cosmos enhancements with this filtered GitHub search: https://bit.ly/EFC6Github.

Conventionally Nested Documents aka Implicit Ownership

In document databases such as Cosmos DB, nested types are expressed naturally in the stored JSON structure. EF Core 6 recognizes this for what is otherwise known as **owned entities,** as well as certain relationships in your model.

Owned entities are EF Core's mechanism for identifying and persisting complex data types that are properties of entities. What differentiates these classes is that they don't have their own key property. With relational databases, it's tricky to store and retrieve data that's shaped this way and you're required to explicitly configure this relationship via the OwnsOne and OwnsMany mappings. In doing so, EF Core knows to infer key properties for the sake of persistence. Otherwise, EF Core assumes that the types are related but you forgot to specify a key property.

In the case of Cosmos DB where it's easy to store nested objects, it feels redundant to have to configure this relationship when it's the obvious way to shape it. This is where the new "implicit ownership" feature comes into EF Core 6. However, it goes beyond the types that you currently define as owned types. In EF Core 6, related dependents will also be stored as sub documents in Cosmos DB. Let's first look at the complex types, which may also be value objects in your system design.

There's not much to it. Let's say I have a type called PersonName where I have defined First and Last properties and a method to concatenate these properties.

```
public class PersonName
{
    public string FirstName { get; private set;}
    public string LastName { get; private set; }
    public PersonName(string first, string last)
    {
        FirstName = first;
        LastName = last;
    }
    public string FullName
        => $"{FirstName.Trim()}
            {LastName.Trim()}";
}
```

There's no identity key and I have no reason at all to create a relationship between the two types. PersonName is a property of Person as well as some other classes. With a relational database provider, I'd have to configure PersonName as an Owned Entity of every type in which it's a property. When using the Cosmos provider (and let's just assume that this will resolve to future document database providers), I don't have to provide the configuration. PersonName will always be nested within a Person document.

```
{
  "Id": "18369f48-c9c9-41e0-a6c1-427dcca4816b",
  "Discriminator": "Person",
  "id": "Person|
        18369f48-c9c9-41e0-a6c1-427dcca4816b",
  "Name": {
    "FirstName": "Andriy",
    "LastName": "Svyryd"
  }
}
```

You can also get nested documents with related entities—related types that *do* have identity keys—but only in a particular scenario.

In my little model of people and addresses (where only one person can live at a given address), the Person class has an Addresses property:

```
public List<Address> Addresses { get; set; }
```

The address type is a true entity with a key property. Therefore, I have a one-to-many relationship between Person and Addresses.

In my first business rule scenario, I know that I'll never interact with addresses directly—only as part of a person object—and therefore I haven't defined a DbSet for Addresses in the DbContext.

Because of these two attributes of my model, I can only create or modify addresses in code as part of a Person object. And the Cosmos provider automatically stores all addresses as nested objects within person objects, as shown in **Figure 2**.

In my second business scenario, I want my app to be able to retrieve and persist addresses separately from their related Person documents. Therefore, I've defined a DbSet<Address> in the DbContext. In response to this, EF Core convention stores addresses as separate documents.

A third scenario is this: I don't want to have a DbSet<Address> defined in my DbContext, but I still prefer to have the Addresses persisted separately from the Person objects for other apps to access directly. In this case, you can explicitly configure the container for the Address type. But it doesn't have to get stored in a separate container. If you explicitly specify the same container, the Addresses


```
{
  "Id": "80a63f76-7624-42d4-9306-aa4a8a298853",
  "Discriminator": "Person",
  "id": "Person|80a63f76-7624-42d4-9306-aa4a8a298853",
  "Addresses": [
    {
      "Id": "de0edfb1-2062-4a8b-9b4c-e00e8ebf53a5",
      "AddressType": 2,
      "PostalCode": "12345",
      "Street": "Two Main",
      "StreetLine2": null
    }
  ],
  "Name": {
    "FirstName": "Shay",
    "LastName": "Rojansky"
  }
}
```


Figure 2: Addresses are nested within Person because there's no DbSet<Address> defined.

get stored alongside the Person objects in the same database container.

Here is the configuration I've used for that specific use case.

```
modelBuilder.Entity<Address> ()
  .ToContainer(this.GetType().Name);
```

With the addresses stored separately, you can still interact with them through the Set<Address> method, if needed.

Intelligently Storing Collections and Dictionaries of Primitives

This is another feature that speaks to how differently document databases store their data from relational databases. And this capability also enhances how you model your entities when persisting them in a document database.

Here are two new properties in my person class: a list of strings to store all of a person's nicknames and a dictionary where I can schedule what type of chocolate to eat on each weekday.

```
public List<String> Nicknames{get;set;}
public Dictionary<string,string>
 DailyChocolate {get;set;}
```

EF Core has no way of conventionally persisting these properties. You'll be told that the type List<String> and the type Dictionary<Datetime,int> need keys defined. Well, you can't really do that! They aren't entities! You can use a complicated value converter to do the trick. Check out how in this section of the EF Core docs (https://bit.ly/ConvertList-String). Surely there's a way to create a value converter for a Dictionary as well.

Storing these into a document database is very natural, so the new Cosmos provider handles it handily for you and by convention, so there are no mappings required.

I've set some nicknames and the chocolate schedule in code:

```
person.Nicknames = new List<string>
 { "Shay", "Roji", "Postgres Guy" };
person.DailyChocolate =
 new Dictionary<string, string>
 { { "Monday", "Dark Chocolate" },
   { "Tuesday", "Salted Milk Chocolate" }
 };
```

And then I call SaveChanges. Here's how that resolves in the stored document:

```
"Id": "429e66c1-f5e9-4750-98f9-c1de2b8a3758",
  "DailyChocolate": {
    "Monday": "Dark Chocolate",
    "Tuesday": "Salted Milk Chocolate"
  },
  "Discriminator": "Person",
  "Nicknames": [
    "Shay",
    "Roji",
    "Postgres Guy"
  ],
```

When querying for that data, it's resolved as the original List and Dictionary in the resulting objects.

There's So Much More to EF Core 6

It's never possible to share every new feature and improvement in an article like this. Therefore, I've stuck to some of the most impactful features of EF Core 6 that will be interesting to most users. But there are still so many interesting things brought to EF Core in this version. For example, in the Cosmos provider, some other improvements are raw SQL support, support for the Distinct operator (with limitations) in queries, and scaffolding many-to-many relationships from existing databases. Thanks to community member Willem Meints, there's now support for String.Concat in queries. Free text search in queries is now more flexible thanks to JSON value converters. The EF Core in-memory database now throws an exception if an attempt is made to save a null value for a property marked as required.

There are hundreds of improvements. Some you will never need, some are subtle and you may not notice, but each one will have a subset of developers who will experience a serious benefit from its existence.

The What's New document in the EF Core docs details more of these higher impact changes (https://docs.microsoft.com/en-us/ef/core/what-is-new/ef-core-6.0).

Although the collective bucket lists of desires for EF Core both from the EF Core team and the community may be vast, EF Core 6 goes a long way to checking off so many of those items.

Julie Lerman

An Introduction to .NET MAUI

Have you ever done any mobile development in the Microsoft ecosystem? Then you might've heard about Xamarin. The technique, at this point synonymous with the company that originally built it, goes back all the way to 2011. It's been Microsoft's main mobile development offering since 2016, when they acquired the company. Xamarin allows developers to use C# code

Steven Thewissen
www.thewissen.io
@devnl

Steven Thewissen is a freelance .NET developer from the Netherlands with a focus on mobile and Web development. He started working with Xamarin in 2014 and has been in love with it ever since. He also enjoys spending time on OSS projects, writing blogs, and kicking a ball around a soccer field.

to develop applications for iOS, Android, and UWP primarily, using Visual Studio. It does all of this from a shared codebase, meaning that unless you want to do something that's platform-specific, you can achieve most of what you want from a single shared library.

The advent of Xamarin.Forms provided an additional abstraction layer on top of that shared codebase with which you can define your user interface in a shared fashion through XAML. To improve the development experience, Microsoft created a lot of additional tooling over the years, making Xamarin a complete offering for mobile developers. The natural next step of that effort was introduced at Build 2020 in the form of the .NET Multi-platform App UI (.NET MAUI). In this article, I'll dive deeper into what it is, and what the biggest changes are compared to Xamarin.Forms.

What is .NET MAUI?

.NET MAUI is the evolution of what is currently Xamarin.Forms. There's now a single .NET 6 Base Class Library (BCL) where the different types of workloads, such as iOS and Android, are now all part of .NET. It effectively abstracts the details of the underlying platform away from your code. If you're running your app on iOS, macOS, or Android, you can now rely on that common BCL to deliver a consistent API and behavior. On Windows, CoreCLR is the .NET runtime that takes care of this.

Even though this BCL allows you to run the same shared code on different platforms, it doesn't allow you to share your user interface definitions. The need for an additional layer of UI abstraction is the problem that .NET MAUI will solve, while simultaneously branching out towards various additional desktop scenarios.

Looking at it from an architectural perspective, most of the code you write will interact with the upper two layers of the diagram shown in **Figure 1**. The .NET MAUI layer handles communication with the layers below it. However, it won't prevent you from calling into these layers if you need access to a platform-specific feature.

Figure 1: The architecture behind .NET MAUI

Making the move to .NET MAUI is also an opportunity for the Xamarin.Forms team to rebuild the eight-year-old toolkit from the ground up and tackle some of the issues that have been lingering at a lower level. Redesigning for performance and extensibility is an integral part of this effort. Companies all over the world use Xamarin extensively, so making these changes in the current toolkit quickly becomes nearly impossible. If you've previously used Xamarin.Forms to build cross-platform user interfaces, you'll notice many similarities when starting to look into .NET MAUI. There are a few differences worth exploring though.

The New Handlers Infrastructure

If you've ever done any Xamarin.Forms development, you might be aware of the concept of a **renderer**. This is a piece of code that takes care of rendering a specific control to the screen in a consistent way across each platform. As a developer, you can create a **custom renderer** that allows you to target a specific type of control on a specific platform and override its built-in behavior. For example, if you want to remove the underline beneath an Android input field, you could write a single custom renderer that would apply to all your **Entry** fields and do just that.

In .NET MAUI the concept of renderers becomes obsolete, but bringing your current renderers to .NET MAUI can be done through the compatibility APIs. Moving forward, handlers will replace renderers entirely. But why? There are a few underlying architectural issues within the current Xamarin.Forms implementation that have spurred the development of an alternative approach.

- The renderer and control are tightly coupled to one another from within Xamarin.Forms, which isn't ideal.
- You register your custom renderers on an assembly level. This means that for every control, the platform performs an assembly scan to find out if a custom renderer should be applied while starting up your app. This can be a rather slow process, relatively speaking. The Xamarin.Forms platform renderers also inject additional view elements that impact performance.
- Xamarin.Forms is an abstraction layer on top of multiple different platforms. Because of this abstraction, it can sometimes be quite difficult to reach the platform-specific code you're looking to change from within the confines of a renderer. Private methods could block your way to the thing you want to customize. The Xamarin.Forms team built additional constructs, such as the platform-specifics API to get around this, but its usage is typically not obvious to users.
- Creating a custom renderer isn't very intuitive. You need to inherit from a base renderer type that isn't well-known, and you can say the same for the methods that you need to override. When you only want your custom renderer to apply to a specific instance of a control, you need to create a custom type (e.g., a

CustomButton), target the renderer at that, and use that control instead of just a regular **Button**. This adds a lot of unnecessary code overhead.

Although those all sound like good reasons to improve, why change now? With this opportunity of reshaping the platform comes the chance for some fundamental rethinking of concepts like these that have been a bit of a sore spot. On the renderers side alone, the benefits are huge when it comes to performance, API simplification, and homogenization.

Reshaping the Underlying Infrastructure

The first step in reshaping the underlying infrastructure is to make sure to remove the current tight coupling with the controls. The .NET MAUI team achieved this by putting them behind an interface and having all the individual components interact with the interface. That way, it becomes easy to make different implementations of something like an **IButton**, while making sure the underlying infrastructure handles all these implementations in the same way. **Figure 2** shows how that looks from a conceptual perspective.

To prevent the need for assembly scanning with reflection, the team decided to change the way handlers are registered. Instead of registering them on an assembly level through attributes, handlers are explicitly registered by the platform, and you can now explicitly register any custom handlers in your startup code. I'll touch upon how to do this later in this article, but this eliminates the need for the assembly scanning penalty that you get on startup.

When it comes to making things hidden deep inside the native platform more easily reachable, the team has taken the approach of defining a mapper dictionary. The mapper is a dictionary with the properties (and actions) defined in the interface of a specific control that offers direct access to the native view object. Casting this native view object to the right type gives you instant access to platform-specific code from your shared code. The following sample shows how you can call into the mapper dictionary for a generic view and set its background color through a piece of platform-specific code. It also shows how to reach the native view.

```
#if __ANDROID__
ViewHandler
   .ViewMapper[nameof(IView.BackgroundColor)] =
      (h, v) => (h.NativeView as AView)
         .SetBackgroundColor(Color.Green);

var textView = label.Handler.NativeView;

#endif
```

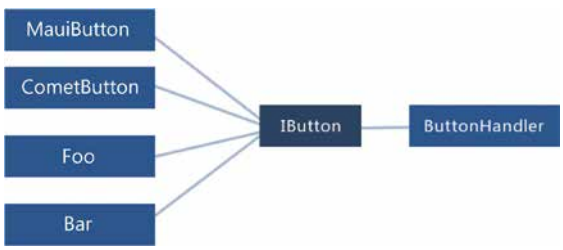

Figure 2: Abstracting away the tight coupling to the control implementations

In this sample, you use the generic **ViewHandler** to reach the background color, because each view has a background color property. Depending on the detail level you need, you can use a more specific handler, such as the **ButtonHandler**. This exposes the native button control directly, eliminating the need to cast it. The existing built-in platform-specifics API becomes obsolete because of this new mapper dictionary. Next, let's take a look at how you can change an existing custom renderer into a handler to see how the overhead that currently exists has been improved.

Differences Between Renderers and Handlers

The Xamarin.Forms renderer implementation is fundamentally a platform-specific implementation of a native control. To create a renderer, you perform the following steps:

- Subclass the control you want to target. Although not required, this is a good convention to adhere to.
- Create any public-facing properties you need in your control.
- Create a subclass of the **ViewRenderer** derived class responsible for creating the native control.
- Override **OnElementChanged** to customize the control. This method is called when the control is created on screen.
- Override **OnElementPropertyChanged** when wanting to target when a specific property changed its value.
- Add the **ExportRenderer** assembly attribute to make it scannable.
- Consume the new custom control in your XAML file.

Let's see how you can create something similar using .NET MAUI. The process to create a handler is as follows:

- Create a subclass of the **ViewHandler** class responsible for creating the native control.
- Override the **CreateNativeView** method that renders the native control.
- Create the *mapper* dictionary to respond to property changes.
- Register the handler in the *startup class*.

Although similarities exist between the two, the .NET MAUI implementation is a lot leaner. A lot of the technical baggage that came with the custom renderers has also been cleaned up, in part due to changes within the .NET MAUI internals. You can find the architecture for the handler infrastructure outlined in **Figure 3**. This sample indicates the layers a button goes through to render to the device screen.

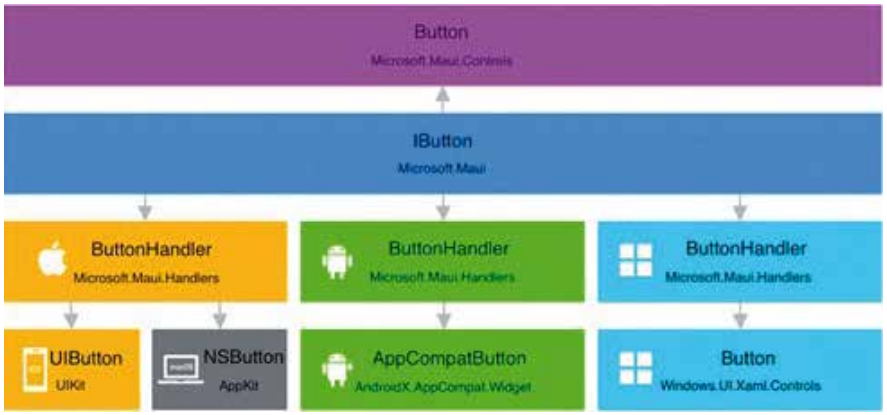

Figure 3: The .NET MAUI handler architecture

Implementing a Handler

To implement a custom handler, start by creating a control-specific interface. You want loose coupling between control and handlers, as mentioned earlier. To avoid holding references to cross-platform controls, you need an interface for your control.

```
public interface IMyButton : IView
{
    public string Text { get; }
    public Color TextColor { get; }
    void Completed();
}
```

By having your custom control implement this interface, you can target this specific type of control from your handler.

```
public class MyButton : View, IMyButton
{

}
```

Next, create a handler targeting the interface you defined earlier on each platform where you want to create platform-specific behavior. In this sample, you target Apple's **UIButton** control, which is the native button implementation on iOS.

```
public partial class MyButtonHandler :
   ViewHandler<IMyButton, UIButton>
{

}
```

Because this handler inherits from **ViewHandler,** you need to implement the **CreateNativeView** method.

```
protected override UIButton CreateNativeView()
{
    return new UIButton();
}
```

You can use this method to override default values and interact with the native control before it's created. That way, you can set a lot of the things that you would previously do in a custom renderer. Additional methods exist to tackle different scenarios, but I won't go into that in this article.

Working with the Mapper

I mentioned the mapper earlier in this article. It's the replacement of the **OnElementChanged** method in Xamarin.Forms, which makes it responsible for handling property changes in the handler. This is also the place where you can hook into these changes with your custom code. Here's what the property mapper for the **IMyButton** you created earlier would look like:

```
public static PropertyMapper<IMyButton,
   MyButtonHandler> MyButtonMapper =
   new PropertyMapper<IMyButton, MyButtonHandler>
      (ViewHandler.ViewMapper)
{
    [nameof(ICustomEntry.Text)] = MapText,
    [nameof(ICustomEntry.TextColor)] = MapTextColor
};
```

The dictionary maps properties to static methods that you can use to handle the property changes and customize the behavior:

```
public static void MapText(MyButtonHandler handler,
   IMyButton button)
{
    handler.NativeView?.SetTitle(button.Text,
       UIControlState.Normal);
}
```

The last thing left to do to make this handler provide custom behavior to your button is to register it. As you recall, .NET MAUI doesn't use assembly scanning anymore. You need to manually register the handler on startup. The next section covers how you can do that.

Adopting the .NET Generic Host

Coming from the ASP.NET Core space, you may already be aware of the .NET Generic Host model. It provides a clean way to configure and start up your apps. It does so by standardizing things like configuration, dependency injection, logging and more. The object is commonly referred to as encapsulating all of this as the *host*, and it's typically configured through the **Main** method in a **Program** class. Alternatively, a **Startup** class can also provide an entry point to configuring the host. This is what the out-of-the-box generic host in ASP.NET Core looks like:

```
public class Program
{
   public static void Main(string[] args)
   {
      CreateHostBuilder(args).Build().Run();
   }

   public static IHostBuilder
      CreateHostBuilder(string[] args) =>
         Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
               webBuilder.UseStartup<Startup>();
            });
}
```

Because .NET MAUI will also use the .NET Generic Host model, you'll be able to initialize your apps from a single location moving forward. It also provides the ability to configure fonts, services, and third-party libraries from a centralized location. You do this by creating a **MauiProgram** class with a **CreateMauiApp** method. Every platform invokes this method automatically when your app initializes.

```
using Microsoft.Maui;
using Microsoft.Maui.Hosting;

public static class MauiProgram
{
    public static MauiApp CreateMauiApp()
    {
        var builder = MauiApp.CreateBuilder();
        builder
           .UseMauiApp<App>()
           .ConfigureFonts(fonts =>
           {
              fonts.AddFont("ComicSans.ttf",
                 "ComicSans");
           });
```

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

The bare minimum this **MauiProgram** class needs to do is to build and return a **MauiApp**. The **Application** class, referenced as **App** in the **UseMauiApp** method, is the root object of your application. This defines the window in which it runs when startup has completed. The **App** is also where you define the starting page of your app.

```
using Microsoft.Maui;
using Microsoft.Maui.Controls;

public partial class App : Application
{
    public App()
    {
        InitializeComponent();
        MainPage = new MainPage();
    }
}
```

I covered the new concept of handlers earlier in this article. If you're looking to hook into this new handler architecture, the **MauiProgram** class is where you register them. You do this by calling the **ConfigureMauiHandlers** method and calling the **AddHandler** method on the current collection of handlers.

```
using Microsoft.Maui;
using Microsoft.Maui.Hosting;

public static class MauiProgram
{
    public static MauiApp CreateMauiApp()
    {
        var builder = MauiApp.CreateBuilder();

        builder
            .UseMauiApp<App>()
            .ConfigureMauiHandlers(handlers =>
            {
               handlers.AddHandler(typeof(MyEntry),
               typeof(MyEntryHandler));
            });

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

In this sample, you're applying the **MyEntryHandler** to all instances of **MyEntry**. The code in this handler will therefore run against any object of type **MyEntry** in your mobile app. This is the preferred solution for when you want to target a completely new control with your handler. If all you want to do is change a property on an out-of-the-box control, you can do this straight from the **MauiProgram** class as well, or really anywhere you know your code will run prior to the control being used.

```
#if __ANDROID__

Microsoft.Maui.Handlers.ButtonHandler
   .ButtonMapper["MyCustomization"] = (handler, view)
     => {
       handler.NativeView
          .SetBackgroundColor(Color.Green);
     };

#endif
```

This sample uses compiler directives to indicate that the handler code should only run on the Android platform because it uses APIs that are unavailable on other platforms. If you're doing a lot of platform-specific code, you might want to consider using other multi-targeting conventions instead of littering your code with compiler directives. This essentially means separating your platform-specific code into platform-specific files suffixed with the platform name. By using conditional statements in your project file, you can then ensure that only the platform-specific files are included when compiling for those specific platforms.

Using Existing Xamarin.Forms Custom Renderers

If you're looking to migrate an existing Xamarin.Forms app to .NET MAUI, you might already have written custom renderers to handle some of the functionality of your app. These are usable in .NET MAUI without too much adjustment, but it's advisable to port them over to the handler infrastructure. To use a Xamarin.Forms, custom renderer register it in the **MauiProgram** class.

```
var builder = MauiApp.CreateBuilder();

#if __ANDROID__

   builder.UseMauiApp<App>()
      .ConfigureMauiHandlers(handlers =>
      {
         handlers.AddCompatibilityRenderer(
         typeof(Microsoft.Maui.Controls.BoxView),
         typeof(MyBoxRenderer));
      });

#endif

return builder;
```

Using the **AddCompatibilityRenderer** method, you can hook up a custom renderer to a .NET MAUI control. You need to do this on a per-platform basis, so if you have multiple platforms, you'll need to add the renderer for each platform individually.

Moving Resources to a Single Project

One of the common pet peeves with Xamarin.Forms is the need to copy a lot of similar resources across multiple projects. If, for example, you have a specific image you want to use in your app, you must include it in all the separate platform projects, and preferably provide it in all the different device resolutions you'd like your app to support. Other types of resources, such as fonts and app icons suffer from a similar issue.

The new Single Project feature in .NET MAUI unifies all these resources into a shared head project that can target every supported platform. The .NET MAUI build tasks will then make sure that these resources end up in the right location when compiling down to the platform-specific artifacts. The single project approach will also improve experiences such as editing the app manifest and managing NuGet packages. **Figure 4** shows a mockup of what the single project experience could look like in Visual Studio. The same project that also contains your other shared logic will also contain the shared resources.

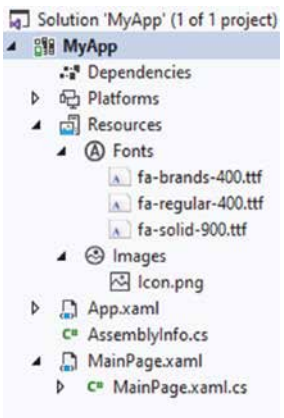

Figure 4: The new Single Project experience in Visual Studio

```
[10:49:05 INF] Applying upgrade step Add Project Properties for .NET MAUI Project
[10:49:05 INF] Removing Project Property: AndroidApplication Value : True
[10:49:05 INF] Removing Project Property: AndroidResgenFile Value : Resources\Resource.designer.cs
[10:49:05 INF] Removing Project Property: AndroidResgenClass Value : Resource
[10:49:05 INF] Removing Project Property: MonoAndroidAssetsPrefix Value : Assets
[10:49:05 INF] Removing Project Property: AndroidUseLatestPlatformSdk Value : false
[10:49:05 INF] Removing Project Property: AndroidEnableSGenConcurrent Value : true
[10:49:05 INF] Removing Project Property: AndroidHttpClientHandlerType Value : Xamarin.Android.Net.AndroidClientHandler
[10:49:05 INF] Removing Project Property: AndroidManagedSymbols Value : true
[10:49:05 INF] Removing Project Property: AndroidUseSharedRuntime Value : false
[10:49:05 INF] Removing Project Property: MonoAndroidResourcePrefix Value : Resources
[10:49:05 INF] Removing Project Property: AndroidUseAapt2 Value : true
[10:49:05 INF] Removing Project Property: AndroidLinkMode Value : None
[10:49:05 INF] Added .NET MAUI Project Properties successfully
[10:49:05 INF] Upgrade step Add Project Properties for .NET MAUI Project applied successfully
Please press enter to continue...
```

Figure 5: Informational output from the .NET Upgrade Assistant for .NET MAUI

What Will Happen to Other Popular Libraries?

A lot of publicly maintained libraries will need to be ported over to .NET MAUI by their creators. .NET Standard libraries without any Xamarin.Forms types will likely work without any updates. Other libraries will need to adopt the new interfaces and types and recompile as .NET 6-compatible NuGets. Some of these have already started this process by releasing early alpha/beta versions of their libraries. If you've ever developed a Xamarin application in the past, you've most likely also used Xamarin.Essentials and/or the Xamarin Community Toolkit.

Essentials now ships as part of .NET MAUI and resides in the Microsoft.Maui.Essentials namespace.

Just like Xamarin.Forms is evolving into .NET MAUI, the Xamarin Community Toolkit is evolving as well and will be known as the .NET MAUI Community Toolkit moving forward. It will still be the fully open-source, community-supported library that it is today, but it's merging with the Windows Community Toolkit, which allows more efficient code sharing and combining engineering efforts across both toolkits. The Xamarin Community Toolkit will also receive service updates on the same public schedule as Xamarin.Forms.

Check out the .NET MAUI Community Toolkit at: https://github.com/CommunityToolkit/Maui

Transitioning Your Existing App to .NET MAUI

Although Microsoft doesn't recommend porting your existing production apps to .NET MAUI right now, providing an upgrade path once .NET MAUI releases has always been a priority. Due to the existing similarities between Xamarin.Forms and .NET MAUI, the migration process can be straightforward. The .NET Upgrade Assistant is a tool that currently exists to help you upgrade from .NET Framework to .NET 5. With the help of an extension on top of the .NET Upgrade Assistant, you're able to automate migrating your Xamarin.Forms projects to a .NET MAUI SDK-style project while also performing some well-known namespace changes in your code. It does so by comparing your project files to what they need to be in order to be compatible with .NET MAUI. The .NET Upgrade Assistant then suggests the steps to take to automatically upgrade and convert your projects. It also maps specific project properties and attributes to their new versions, while stripping out obsoleted ones.

By using extensive logging, as shown in **Figure 5,** you'll be able to know all the steps the tool has taken to upgrade your project. This will also help you debug potential issues during your migration.

During the early days of .NET MAUI, there might not yet be adequate support for some of your NuGet packages. The .NET Upgrade Assistant works with analyzers to go through and validate whether these packages can be safely removed or upgraded to a different version.

Although it's not 100% able to upgrade your project, it does take away a lot of the tedious renaming and repeating steps. As a developer, you'll have to upgrade all your dependencies accordingly and manually register any of your compatibility services and renderers. Microsoft has stated that they will try to minimize the effort this takes as much as possible. Additional documentation on the exact process will be made available closer to release.

Check out the .NET Upgrade Assistant at: https://dotnet.microsoft.com/platform/upgrade-assistant

Conclusion

Developers who've worked with Xamarin.Forms in the past will find a lot of things in .NET MAUI to be familiar. The underlying changes to infrastructure, broader platform scope, and overall unification into .NET 6 also make it appealing to people new to the platform. Centralizing a lot more resources and code into the shared library using the single project feature greatly simplifies solution management. Additional performance improvements through using handlers gives the seasoned Xamarin developer something to explore.

Although the version of .NET MAUI in .NET 6 is highly anticipated, it's also only the first version of the platform. I personally expect a lot of additional features coming soon, and best of all; the entire platform is open source. This means that you and everyone else in the .NET ecosystem can contribute to improve and enhance the platform. I'm certainly curious to see what the future holds!

If you want to try out .NET MAUI for yourself, you can check out the GitHub repository (https://github.com/dotnet/maui) and the Microsoft Docs (https://docs.microsoft.com/dotnet/maui/), which already provide content on getting started.

Steven Thewissen

Blazor Hybrid Web Apps with .NET MAUI

Microsoft has introduced the highly anticipated Blazor framework in ASP.NET Core 3.0. In .NET 5.0, Blazor received significant updates to its component model, plus speed improvements and pre-rendering capabilities. Blazor's initial focus was to allow developers to target the browser using the .NET stack with little or no JavaScript required, all without a single browser plug-in.

The key to Blazor's success is its ability to enable .NET developers by leveraging their existing skills. Using Blazor, .NET developers can build a full-stack application using only .NET technologies.

In .NET 6.0, the Blazor framework finds yet another path for developer success with .NET MAUI. MAUI provides a set of technologies that enable apps to run on Web, desktop, and mobile. This new pattern is named Blazor Hybrid and, once again, developers are empowered to use their existing skills to reach even more ecosystems. With Blazor Hybrid, native desktop on Android, iOS, macOS, and Windows are now within reach.

> .NET MAUI stands for .NET Multi-platform App UI.

Bringing Blazor to the Desktop

Using Blazor for client-side Web UI with .NET is a fantastic solution, but sometimes full access to the native capabilities of the device is required and out of reach of Blazor on the Web. Blazor Hybrid combines Web technologies (HTML, CSS, and optionally JavaScript) with native in .NET MAUI Blazor. MAUI is a cross-platform framework for creating native mobile and desktop apps with C# and XAML. MAUI uses a single shared code-base to run on Android, iOS, macOS, and Windows, as illustrated by **Figure 1**.

Bringing Blazor to the desktop isn't a new or unique idea. It's how .NET MAUI Blazor targets cross-platform development that sets it apart from other solutions. To evaluate .NET MAUI Blazor properly, you need to first understand what the alternatives offer and what the tradeoffs are for each. There are two widely accepted methods of running Blazor on the desktop: Using Blazor as **Progressive Web Applications (PWAs)** or using it as an **Electron** shell.

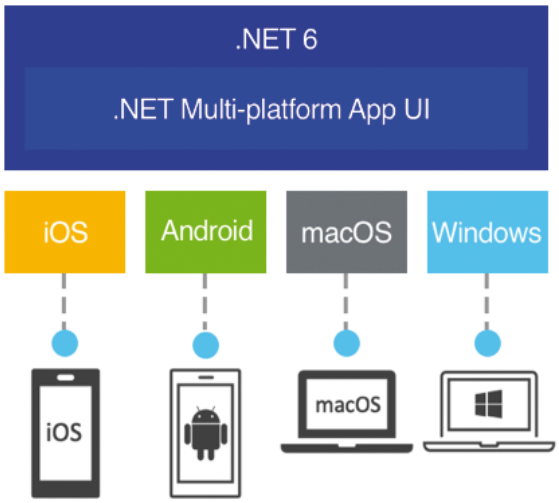

Figure 1: An expanded view of the MAUI platform

Progressive Web Applications

Progressive Web Applications (PWAs) are a type of Web application that can be installed on an operating system without the need for additional bundling and distribution systems. Publishing to an app store, such as the **Microsoft Store**, **Google Play**, or **Apple App Store** is optional and may require additional bundling. PWAs are built with Web-standard technologies including HTML, CSS, JavaScript, and WebAssembly, which work on a standards-compliant browser. PWA features are supported to varying degrees on both desktop and mobile with Apple lagging far behind in adoption.

Once a PWA is installed, the browser's address bar and buttons (Chrome) are stripped away. Just as in a native application, the PWA has a launch icon and interact natively with the Windows task bar. A PWA gains an important feature that further enhances the user experience: **Service workers**. Service workers are part of the PWA specification that's a type of Web worker. Service workers are JavaScript code that runs separately from the main browser thread, which can provide an **offline mode** (intercepting network requests, caching, or retrieving resources from the cache), and **deliver push messages**. Although PWAs can't access operating system-level APIs, they do feel much more native than a traditional Web application by mimicking their behavior. An example can be seen in **Figure 2** with the Blazing Coffee demo app from the Telerik website found at demos.Telerik.com (https://demos.telerik.com/blazor-coffee/).

Blazor WebAssembly applications can easily take advantage of PWA features by simply meeting the installation criteria of a PWA. Therefore, a PWA option is already available for Blazor when starting a new project from a template. Although Blazor PWA apps can easily be created, there are tradeoffs. Because there's no .NET API support for service workers, all functionalities must be done in JavaScript. And because one of Blazor's attractions is C#, this deters some developers from venturing too deep into service workers.

Electron

Electron is an open-source framework for building native desktop applications with Web technologies like JavaScript, HTML, and CSS. Electron uses an embedded **Chromium wrapper** that's powered by **Node.js**. Electron allows you to maintain **one JavaScript codebase** and create cross-platform apps that work on Windows, macOS, and Linux. Many popular desktop apps are essentially Electron Web apps. Visual Studio Code, Microsoft Teams, Slack, and Figma are all electron apps that developers use daily.

Electron is more than just a Web wrapper as the framework also provides a custom set of JavaScript APIs to interact with

Ed Charbeneau
Ed.charbeneau@progress.com
@EdCharbeneau

Ed is a Microsoft MVP and an international speaker, writer, online influencer, a Pr. Developer Advocate for Progress, and expert on all things Web development. Ed enjoys geeking out to cool new tech, brainstorming about future technology, and admiring great design.

Ed has shared his insights and experiences through training, live streaming, and podcasting at many industry events around the world including Microsoft Build, DevReach, Oredev, and Codemash. He's a community builder who regularly live streams on Twitch and shares knowledge at meetups. Ed has defined, architected, and implemented line-of-business solutions with a touch of style and UX best practices. He continues to bring this level of detail to the Telerik UI products.

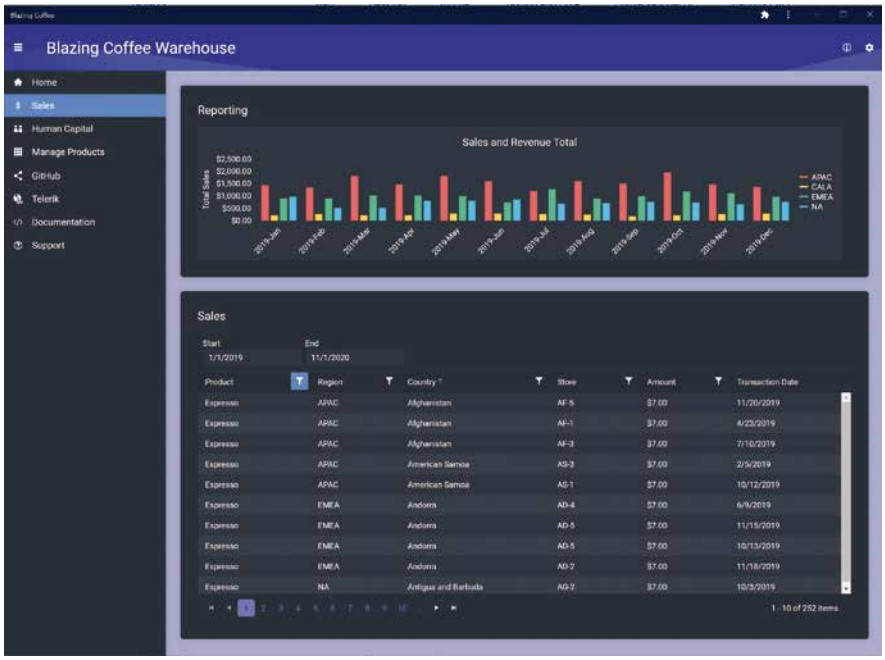

Figure 2: The Telerik Blazing Coffee demo app installed as a PWA

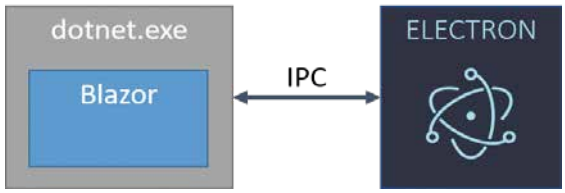

Figure 3: A block diagram of a Blazor Electron app

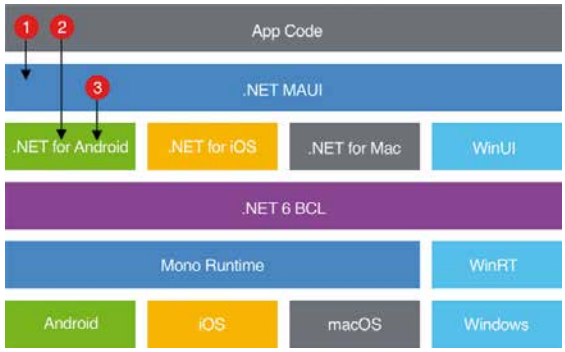

Figure 4: A block diagram of the MAUI platform

the host operating system. These modules control native desktop functionality, such as menus, dialogs, and tray icons. Although the API does surface some native desktop functionality, it isn't raw access to the full platform; instead, these are a select set of common features between platforms. APIs include access to the filesystem, running processes outside of the browser sandbox, kiosk mode, screen recording, system-tray support for minimized apps, and more.

Because Electron works by leveraging Web standard technologies via Chrome, it supports WebAssembly. This means that a Blazor WebAssembly app can be embedded in an Electron shell and transformed into a desktop application. Such applications use the .NET runtime in the context of Blazor WebAssembly running in interpreted mode, which is less performant than its native desktop equivalent. Using Electron's APIs using .NET is done through Electron.NET, a wrapper around a JavaScript-based Electron application with an embedded ASP.NET Core application. Via an Electron.NET IPC bridge, Electron APIs are invoked from the Blazor application. A block diagram of the architecture is shown in **Figure 3**.

Although Blazor apps can be successful using Electron and Electron.NET, there are tradeoffs. Electron is only for desktop applications, not mobile. Blazor apps rely on multiple frameworks from different vendors and communities and performance is not that of .NET running on the native operating system. In addition, API access is restricted to what's provided within the Electron and Electron.NET frameworks.

Introducing the Blazor Hybrid Architecture

MAUI is the **evolution of Xamarin.Forms,** which initially targeted iOS and Android, and with MAUI, expanded into desktop. Using MAUI, you'll write cross-platform applications in a single solution with the option of writing platform-specific code as needed. Because MAUI is full-stack .NET, sharing code, logic, testing, and tooling across the solution is possible. The Blazor Hybrid pattern is built upon MAUI and implemented through the **BlazorWebView**, a MAUI component used to render an embedded Blazor Web view using the WebView2 runtime.

.NET MAUI uses a single API to unify Android, iOS, macOS, and Windows APIs into a write-once run-anywhere developer experience. MAUI apps provide deep access into each native platform. .NET 6 introduces a series of platform-specific frameworks: .NET for Android, .NET for iOS, .NET for macOS, and Windows UI (WinUI) Library. The .NET 6 Base Class Library is shared among all the platforms while abstracting the individual characteristics of each platform from your code. The .NET runtime is used for the execution environment for MAUI applications, even though the underlying implementations of the runtime may be different, depending on the host. For example, on Android, iOS, and macOS, the environment is impended by Mono (a .NET runtime), and on Windows, WinRT provides the environment with optimizations for the Windows platform.

In a .NET MAUI app, you write code that primarily interacts with the .NET MAUI API, shown in **Figure 4 (1)**. .NET MAUI then directly consumes the native platform APIs, as in **Figure 4 (3)**. In addition, app code may directly exercise platform APIs, shown in **Figure 4 (2)**, if required.

MAUI is more than an abstract BCL to share common business logic on different platforms—it also unifies user interface (UI) development too. .NET MAUI provides a single framework for building the UIs for mobile and desktop apps. Because each platform has their own models and elements used to describe their UI, using individual platform-specific frameworks would be difficult to maintain. Instead, MAUI provides a common multi-platform framework for creating user interfaces, while having the flexibility to target specific platforms as needed. In addition to native UI frameworks, MAUI also introduces the **BlazorWebView**. Through a BlazorWebView component, MAUI apps can use the Blazor Web framework creating a .NET MAUI Blazor application.

BlazorWebView and .NET MAUI Blazor

The Blazor Hybrid pattern uses a BlazorWebView component that enables Blazor within a MAUI application, creating a .NET MAUI Blazor application. .NET MAUI Blazor enables both native and Web UI in a single application and they can co-exist in a single view. With .NET MAUI Blazor, applications can leverage Blazor's component model (Razor Components), which uses HTML, CSS, and the Razor syntax. The Blazor part of an app can reuse components, layouts, and styles that are used in an existing regular Web app. BlazorWebView can be composed alongside native elements; additionally, they leverage platform features and share state with their native counterparts.

In .NET MAUI Blazor apps, all code, both for the native UI parts and the Web UI parts, runs as .NET code on the platform's runtime using a single process. There's no local or remote Web server and no WebAssembly (WASM) in the Blazor Hybrid pattern. The native UI components run as the device's standard UI components (button, label, etc.) and the Web UI components are hosted in a Web view. The components can share state using standard .NET patterns, such as event handlers, dependency injection, or anything else you're already using in your apps today.

Creating a .NET MAUI Blazor Application

Creating your first .NET MAUI Blazor Application is familiar, yet new, territory for most developers. Although .NET technologies like Xamarin.Forms and Blazor have been around for some time, using them together is a new experience. The best way to ensure success is to install the latest updates for all the technologies involved. Thankfully, there are installers for the required SDKs and the version of Visual Studio 2022 with support for MAUI is just a few clicks away. Once the prerequisites are installed, I'll look at the .NET MAUI Blazor template and get an understanding of how the project is structured. Let's install everything now.

Installing .NET MAUI

On Windows, from the Visual Studio Installer for Visual Studio 2022, select the new .NET MAUI workload. This MAUI workload includes the .NET 6 SDK, the MAUI SDK, and MAUI templates. For non-Windows users, choose your development computer's .NET 6 SDK installer from https://dotnet.microsoft.com/, and then use the command line tool to install the MAUI workload by running **dotnet workload install maui**. When the installation is complete, MAUI will be available from Visual Studio or the command line.

With the prerequisites installed, a new .NET MAUI Blazor app is created using the MAUI-Blazor template. From Visual Studio, this is as simple as choosing the template from the **File > New** dialog. For all other platforms, use the CLI command **dotnet new maui-blazor.** Once the template has created a new project, you can see how a .NET MAUI Blazor app is structured.

A .NET MAUI Blazor app shares some similarities with a traditional Blazor app with the addition of MAUI features, such as a platform-specific feature folder and XAML files. Let's examine the project and identify the importance of each part. A newly constructed .NET MAUI Blazor app is shown in **Figure 5**.

At first glance, some familiar concepts may appear in **Table 1**, as .NET MAUI Blazor apps use patterns found in many .NET app types. Some key differences are the duality of the hybrid scenario where there's overlap between the concepts of root-level component views and routing. Let's examine some of the files to get an understanding of the purpose of each.

File or Folder	Purpose
/Pages	Razor component-based pages or features that will be rendered within a WebView component.
/Platforms	Platform-specific files, including resources, configurations, native business logic, or native UI components.
/Resources	Global application resources and static files.
/Shared	Common Razor components and Layout components used in Blazor WebViews.
/wwwroot	Web resources used in Blazor Webviews. Ex: CSS, fonts, and images.
_imports.razor	Global Using statements for Razor components and Pages.
App.xaml(.cs)	The root-level application view.
Main.razor	The root-level Blazor view and router.
MainPage.xaml	The default view rendered by the root (App.xaml).
Startup.cs	Application entry point, bootstrapping, and configuration.
/Pages/Counter.razor	A sample component that counts button click events.
/Pages/FetchData.razor	A sample component that fetches and displays data.

Table 1: A breakdown of .NET MAUI Blazor project folders and files

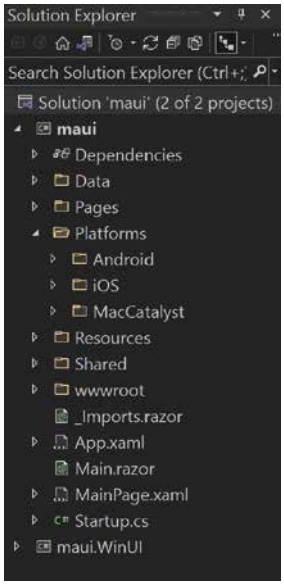

Figure 5: A newly created .NET MAUI Blazor app created from the template

At the time of writing, the project uses .NET 6 preview 7. Some of the project structure, files, and code may differ from the final .NET 6 release. The MAUI.WinUI project will likely consolidate under the /Platforms folder and isn't mentioned exclusively in the examples.

Startup.cs

Like most .NET apps, Startup is the entry point of the application. As Startup is initialized, the application is constructed and services are registered, the application is configured, and despondency injection is resolved. In the following code snippet, an application builder (appBuilder) is used to add application features. You can see the Blazor Web application registered in the first item with the method **RegisterBlazorMauiWebView**. Further down, the **UseMauiApp<App>** method initializes the App application root. The App class specifies the application's MainPage, which is initialized to the MainPage class. The remainder of the appBuilder registers and configures services that are used with **dependency injection** (DI) throughout the entire application.

```
public class Startup : IStartup
{
  public void Configure(IAppHostBuilder appBuilder)
  {
    appBuilder
    .RegisterBlazorMauiWebView()
    .UseMicrosoftExtensionsServiceProviderFactory()
    .UseMauiApp<App>()
    .ConfigureFonts(fonts =>
      {
                  fonts.AddFont("OpenSans-Regular.ttf",
          "OpenSansRegular");
          })
         .ConfigureServices(services =>
      {
        services.AddBlazorWebView();
        services
```

Listing 1: FetchData.razor

```
@inject WeatherForecastService ForecastService

<h1>Weather forecast</h1>

<p>This component demonstrates fetching data...</p>

@if (forecasts == null)
{
  <p><em>Loading...</em></p>
}
else
{
  <table class="table">
    <thead>
      <tr>
        <th>Date</th>
        <th>Temp. (C)</th>
        <th>Temp. (F)</th>
        <th>Summary</th>
      </tr>
    </thead>
    <tbody>
    @foreach (var forecast in forecasts)
      {
        <tr>
          <td>@forecast.Date.ToShortDateString()</td>
          <td>@forecast.TemperatureC</td>
          <td>@forecast.TemperatureF</td>
          <td>@forecast.Summary</td>
        </tr>
      }
    </tbody>
  </table>
}

@code {
  private WeatherForecast[] forecasts;
  protected override void OnInitialized()
  {
   forecasts = ForecastService.GetForecast();
  }
}
```

```
        .AddSingleton<WeatherForecastService>();
      });
  }
}
```

MainPage.xaml(.cs)

In the MainPage shown in the snippet below, you can see the first introduction of Blazor as a BlazorWebView. The MainPage wraps the BlazorWebView directly within a ContentPage, essentially creating a full-page Blazor view inside of the application's UI shell.

```
<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>
```

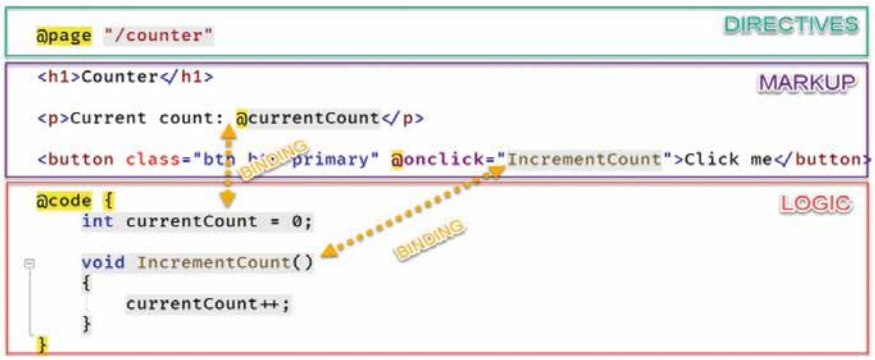

Figure 6: The composition of the counter component has directives, markup, and logic connected by data binding.

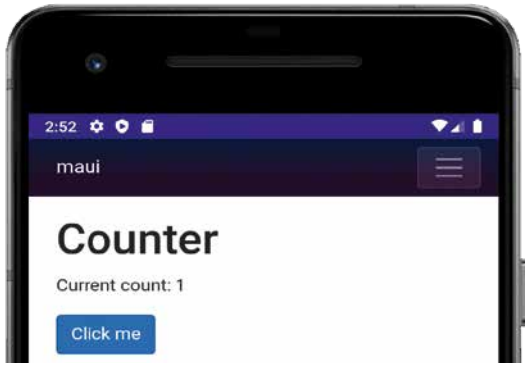

Figure 7: The Counter component rendered in .NET MAUI Blazor

The BlazorWebView uses a **HostPage** parameter to identify the HTML page, which will bootstrap the Blazor application. In the index.html shown in **Listing 3**, you'll find the root document that hosts the Blazor application within the view. Unlike Blazor WebAssembly, this HTML file initializes Blazor using **blazor.webview.js** instead of **blazor.webassembly.js**. The distinction here is that **Blazor isn't using WebAssembly**, but rather, the .NET runtime of the host application.

Counter.razor

The Counter page is a simple component decorated with the page directive. This component demonstrates the basic composition of a Razor Component (aka Blazor) including routing, data binding, and event binding/handling. Each portion of component composition is highlighted in **Figure 6**.

The counter component uses a basic HTML button to increment a counter field that's displayed within a paragraph tag. Updates to the BlazorWebView's DOM are handled by the Blazor framework though data binding. You can see the rendering of the counter component in **Figure 7.**

FetchData.razor

In the .NET MAUI Blazor project type, the Fetch Data page is a component that uses data from a service. The Fetch Data component demonstrates dependency injection and basic Razor template concepts. This version of Fetch Data is very similar to the example found in other Blazor application templates.

In **Listing 1**, at the top of the component following the routing directive, dependency injection is declared. The **@ inject** directive instructs Blazor to resolve an instance of **ForecastService**. The WeatherForecast is then used by the component's logic to fetch data and bind to an array of **WeatherForecast** objects. Displaying the **WeatherForecast** data is done by iterating over the **forecasts** collection and binding the values to an HTML table.

Running .NET MAUI Blazor

When running the application from Visual Studio, you immediately see the cross-platform nature of MAUI, as shown in **Figure 8**. Choosing to "run" from Visual Studio involves selecting from emulators and/or physical connected devices. Thankfully, making connections to emulators and devices is made easy through Visual Studio, a feature that has evolved from many years of supporting Xamarin. When the application launches, you can see the Blazor application running in a native shell, all without needing a Web browser, or plug-in. In the sample provided, the Blazor Web UI is used for all navigation, routing, and views in the application. Included are the familiar **counter** and **fetch data** examples, which are routine to Blazor Web projects.

Figure 8: The run dialog for a MAUI app in Visual Studio 2022

The template is just a glimpse into how a Blazor application is integrated with MAUI to form a .NET MAUI Blazor app. There is much more to Blazor Hybrid than just a BlazorWebView component.

Blazor Hybrid's Superpowers

When Blazor was introduced for the Web, one of the primary goals was to enable developers to build Web application UIs using .NET. Blazor has been successful in this regard. Blazor allows existing .NET code to work by using the .NET runtime via WebAssembly. With Blazor Hybrid, the primary goal has shifted slightly by extending the capabilities of .NET developers beyond the Web into the desktop and mobile space while reusing HTML and CSS skills in addition to .NET. The Blazor Hybrid pattern with .NET MAUI Blazor offers some unique abilities that are not available with Web centric development.

Minimizing Tradeoffs by Running Native .NET

Blazor WebAssembly and Blazor Server come with tradeoffs. You gain the capabilities of .NET in place of JavaScript at the cost of abstraction. When using WebAssembly, the .NET runtime operates in interpreted mode and isn't as performant as .NET running natively. Although advances to this technology are coming in .NET 6 with Ahead of Time Compilation (AOT), it remains a tradeoff when choosing WebAssembly. Similarly, Blazor server has tradeoffs as well. Although Blazor Server gains the ability to run .NET natively on the server, it requires a constant connection to the client and the performance is indicative of the client's latency.

As seen in **Table 2**, a .NET MAUI Blazor app has a unique position in the Blazor ecosystem where it can eliminate these tradeoffs by running the .NET runtime supplied by the native platform. Unlike Blazor WebAssembly, .NET MAUI Blazor **doesn't use interpreted mode** and performs as a device-native app. Because .NET MAUI Blazor is processed locally, the tradeoffs of Blazor Server are also circumvented. Although there are nuances to each platform's execution of the .NET runtime, as noted in **Table 2** footnotes 1-4, .NET MAUI Blazor's execution is "closer to the metal" than that of WebAssembly.

In addition to performance, .NET MAUI Blazor also has the greatest potential for sharing a single codebase while targeting cross platform development. This includes the ability to publish applications to all the major app stores with the added ability to receive a mobile home screen icon or desktop icon.

Feature	Blazor Hybrid	Blazor Electron	Blazor PWA
Installable	Yes	Yes	Yes
Publish to Store	Yes	Yes	No
Access Native APIs	via MAUI [1][2][3][4]	via Electron.NET	No
WebAssembly	No [1][2][3][4]	Yes	Yes
HTML/CSS	Yes	Yes	Yes
Desktop	Yes	Yes	Yes
Android	Yes	No	Yes
iOS	Yes	No	Limited
macOS	Yes	Yes	No
[1] Android apps built using .NET MAUI compile from C# into intermediate language (IL), which is then just-in-time (JIT) compiled to a native assembly when the app launches. [2] iOS apps built using .NET MAUI are fully ahead-of-time (AOT) compiled from C# into native ARM assembly code. [3] macOS apps built using .NET MAUI use Mac Catalyst, a solution from Apple that brings your iOS app built with UIKit to the desktop and augments it with additional AppKit and platform APIs, as required. [4] Windows apps built using .NET MAUI use Windows UI Library (WinUI) 3 and WinRT execution to create native apps that can target the Windows desktop and the Universal Windows Platform (UWP).			

Table 2: Comparing .NET MAUI Blazor capabilities with Electron and PWAs

Listing 2: WeatherForecastService.cs

```
public WeatherForecast[] GetForecast()
{
  string fileName = Path.Combine(
    Environment.GetFolderPath(
    Environment.SpecialFolder.LocalApplicationData)
     , "temp.csv");

  WeatherForecast[] weather;
  using (var reader = new StreamReader(fileName))
  using (var csv = new CsvReader(
    reader, CultureInfo.InvariantCulture))
    {
      weather = csv.GetRecords<WeatherForecast>()
      .ToArray();
    }
    return weather;
}
```

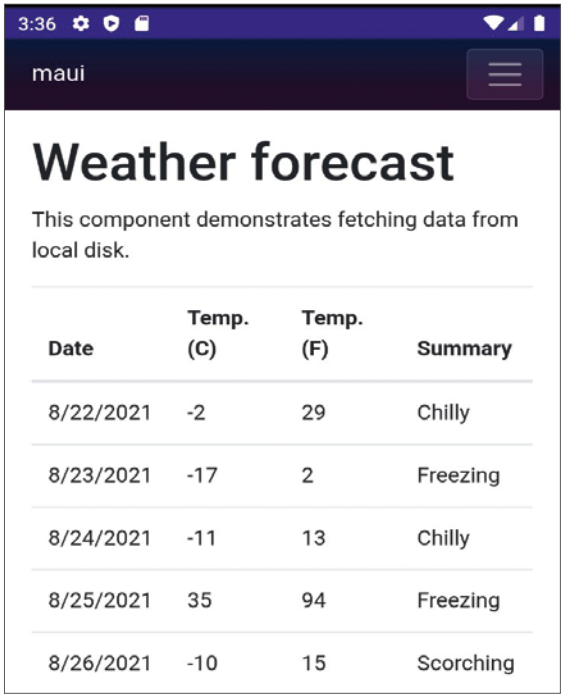

Figure 9: A .NET MAUI Blazor app fetching weather data from disk, and running on mobile

WebView2

The Microsoft Edge WebView2 control allows embedded Web technologies (HTML, CSS, and JavaScript) in native apps. The WebView2 control uses Microsoft Edge (Chromium) as the rendering engine to display the Web content in native apps.

With WebView2, Web code can be embedded in different parts of your native app, or a single WebView instance can be used to wrap the entirety of the app.

Distributing WebView2 with an application can be resolved in two ways. An "Evergreen distribution" approach can be used, which relies on an up-to-date version of Chromium with regular platform updates and security patches. Alternatively, a "Fixed Version" distribution can be used by packaging a specific version of the Chromium bits in the app.

WebView2 is supported by a wide range of Windows versions from 7 and up, and serves as a vehicle for migrating legacy applications. Through WebView2, an app can be transitioned from an existing technology to Web components using a piece-by-piece strategy.

Leveraging APIs

.NET MAUI Blazor applications aren't restricted to the same Web sandbox that Blazor WebAssembly is. .NET MAUI Blazor apps use the .NET 6 Base Class Library (BCL), which is implemented across all platforms. .NET MAUI unifies cross-platform APIs into a single API that allows a write-once run-anywhere developer experience, while additionally providing deep access to every aspect of each native platform. .NET MAUI also provides MAUI Essentials, a cross-platform API for native device features.

Examples of functionality provided by .NET MAUI essentials include:

- Access to sensors, such as the accelerometer, compass, and gyroscope on devices
- Ability to check the device's network connectivity state and detect changes
- Information is provided about the device the app is running on
- Copying and pasting text to the system clipboard between apps
- Picking single or multiple files from the device
- Storing data securely as key/value pairs
- Using built-in text-to-speech engines to read text from the device
- Initiating browser-based authentication flows that listen for a callback to a specific app registered URL

In the sample code of **Listing 2,** you can read a CSV file from disk using **System.IO.StreamReader** from the BCL. The WeatherForecastService is a class that can be injected into any MAUI view or a BlazorWebView. When the **GetForecasts** method is called from the WeatherForecastService, a **file from disk** is loaded into an array of WeatherForecast. The service uses a **cross-platform path** for files **Environment.SpecialFolder.LocalApplicationData** to ensure device compatibility. Once the path is established, the file is read into a stream reader and processed using the open-source CSV reader, **CsvHelper**. In the FetchData component, the WeatherForecastService is injected and the GetForecast method is called. Once the data has been read from disk, the Razor syntax is used to iterate over the data using HTML and CSS, as shown in **Listing 1**. The view code is identical to that of the Blazor Maui template because, only the service internals have changed.

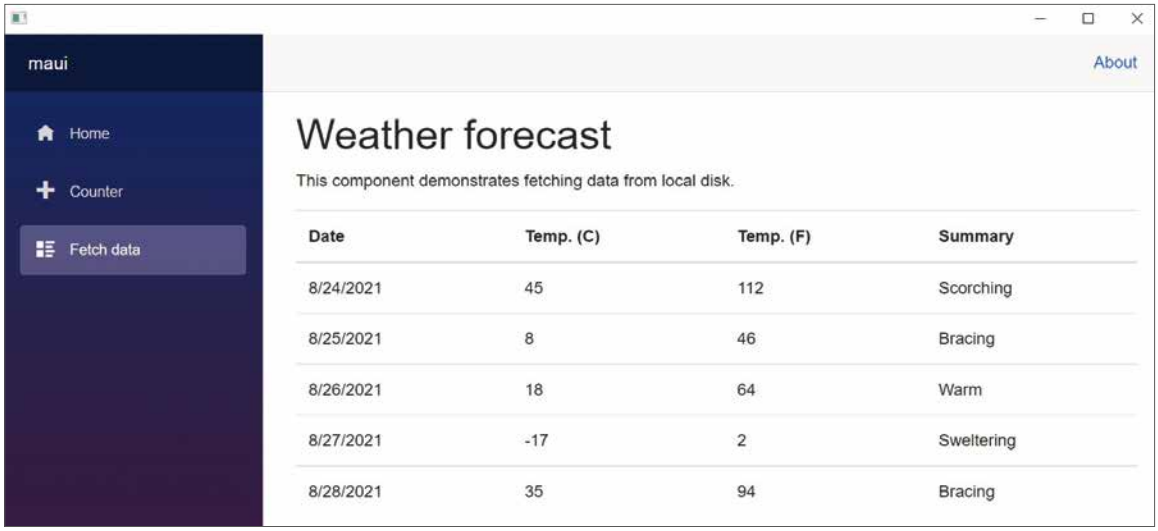

Figure 10: A .NET MAUI Blazor app fetching data weather from disk, running Windows

Listing 3: Index.html

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width,
initial-scale=1.0, maximum-scale=1.0, user-scalable=no" />
    <title>Blazor app</title>
    <base href="/" />
    <link href="{PROJECT NAME}.styles.css" rel="stylesheet" />
    <link href="app.css" rel="stylesheet" />
</head>

<body>
    <div id="app"></div>

    <div id="blazor-error-ui">
        An unhandled error has occurred.
        <a href="" class="reload">Reload</a>
        <a class="dismiss">✕</a>
    </div>

    <script src="_framework/blazor.webview.js"></script>
</body>
</html>
```

Listing 4: app.css

```
html, body {
    font-family: 'Helvetica Neue', Helvetica, Arial, sans-serif;
}

.valid.modified:not([type=checkbox]) {
    outline: 1px solid #26b050;
}

.invalid {
    outline: 1px solid red;
}

.validation-message {
    color: red;
}

#blazor-error-ui {
    background: lightyellow;
    bottom: 0;
    box-shadow: 0 -1px 2px rgba(0, 0, 0, 0.2);
    display: none;
    left: 0;
    padding: 0.6rem 1.25rem 0.7rem 1.25rem;
    position: fixed;
    width: 100%;
    z-index: 1000;
}

    #blazor-error-ui .dismiss {
        cursor: pointer;
        position: absolute;
        right: 0.75rem;
        top: 0.5rem;
    }
```

Running the app either on the desktop or mobile provides the same experience. The CSV data read from disk is displayed in the hybrid view, shown in **Figure 9** for mobile.

The CSV data read from disk is displayed in the hybrid view, shown in **Figure 10** for Windows desktop.

The examples provided are just one approach to using the BlazorWebView with MAUI. In this scenario, Blazor is used exclusively within the application shell providing the complete UI and navigation by the Main component, which is specified in the ComponentType parameter. Overtaking the entire app with a BlazorWebView component is optional; it can be added ad hoc to any XAML view and even mixed with native UI components. Mixed UIs can share application state and fully interact, as shown in **Figure 11**.

Hybrid Ecosystem

.NET 6 will further strengthen the developer ecosystem that surrounds MAUI and Blazor. Since the very beginning when Blazor was just an experiment, libraries supporting the app model started springing up. The same excitement can be seen with MAUI as well. Telerik, a brand synonymous with .NET developers for nearly 20 years, has already announced Telerik UI for MAUI (https://www.telerik.com/maui-ui). Telerik has a dedicated MAUI solution and a dedicated Blazor UI offering with 85+ components, embracing a future where developers have the choice of using platform-native UIs with MAUI and XAML, Blazor with HTML, or both UI component libraries working together seamlessly. The key takeaway here is that developers have choices as to how to implement UI that is unprecedented in .NET development.

As the Blazor ecosystem continues to grow and flourish, it will be further enhanced with packages enabled by MAUI via na-

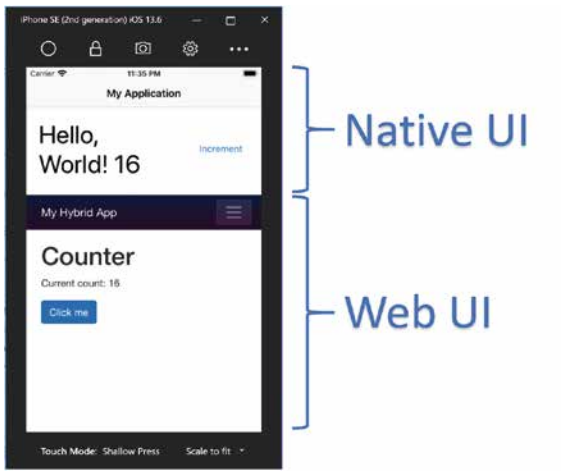

Figure 11: A .NET MAUI Blazor app with mixed Native and Web UI using a shared state.

Listing 5: Counter.razor

```
@using Microsoft.AspNetCore.Components.Web

<h1>Counter</h1>

<p>The current count is: @currentCount</p>
<button @onclick="IncrementCount">Count</button>

@code {
    int currentCount = 0;

    void IncrementCount()
    {
        currentCount++;
    }
}
```

tive platform integration and runtime execution. Libraries that may have been hampered by incompatibilities prior to MAUI find a new niche with Blazor Hybrid. For example, ML.NET, an open source, cross-platform machine learning (ML) framework for .NET devs, has seen limitations around WebAssembly. With Blazor Hybrid, the compatibility problem becomes less relevant as ML.NET and MAUI push to cover the same execution environments (https://devblogs.microsoft.com/dotnet/ml-net-june-updates-model-builder/#ml-net-release).

Listing 6: MyForm.cs

```
var serviceCollection = new ServiceCollection();
serviceCollection.AddBlazorWebView();
var blazor = new BlazorWebView()
{
    Dock = DockStyle.Fill,
    HostPage = "wwwroot/index.html",
    Services = serviceCollection.BuildServiceProvider(),
};
blazor.RootComponents.Add<Counter>("#app");
Controls.Add(blazor);
```

Listing 7: MainWindow.xaml

```
<Window x:Class="WpfApp1.MainWindow"
        xmlns="…"
        xmlns:local="clr-namespace:WpfApp1"
        xmlns:blazor="clr-namespace:
Microsoft.AspNetCore.
Components.WebView.Wpf;
assembly=Microsoft.AspNetCore.Components.WebView.Wpf"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>
        <blazor:BlazorWebView HostPage="wwwroot/index.html"
Services="{StaticResource services}">
            <blazor:BlazorWebView.RootComponents>
                <blazor:RootComponent Selector="#app"
ComponentType="{x:Type local:Counter}" />
            </blazor:BlazorWebView.RootComponents>
        </blazor:BlazorWebView>
    </Grid>
</Window>
```

Listing 8: MainWindow.xaml.cs

```
var serviceCollection = new ServiceCollection();
serviceCollection.AddBlazorWebView();
Resources.Add("services", serviceCollection.BuildServiceProvider());
```

Listing 9: Counter.razor.cs

```
public partial class Counter { }
```

Listing 10: Counter.razor (XML)

```
<StackLayout>
    <Label FontSize="30"
           Text="@("You pressed " + count + " times")" />
    <Button Text="+1"
            OnClick="@HandleClick" />
</StackLayout>

@code {
    int count;

    void HandleClick()
    {
        count++;
    }
}
```

Expectations are high here as developer options increase due to the expanding reach of MAUI and Blazor Hybrid, while new libraries and frameworks emerge to provide new and unique solutions.

Migration Paths

If the Blazor Hybrid pattern using .NET MAUI Blazor seems interesting but you're currently developing for **WPF** or **Windows Forms**, then hybrid feels out of reach. Traditionally, migrating an existing app to a new platform requires a lot of manually rewriting code, but the .NET team has provided a transitional pathway. In .NET 6, a BlazorWebView component was added to both WPF and Windows Forms. This means that existing projects using .NET can be upgraded to .NET 6.0, and, with a few steps, enable BlazorWebView.

By enabling BlazorWebView in WPF and Windows Forms, it's possible to start decoupling UI investments from WPF and Windows Forms. These Blazor-enabled projects only target the Windows platform, unlike MAUI. This is a great way to modernize existing desktop apps in a way that can be brought forward onto .NET MAUI or used on the Web. By using Blazor to modernize existing Windows Forms and WPF apps, existing investments can be leveraged with a transitional Blazor Hybrid application.

To use the new BlazorWebView controls, you first need to make sure that you have WebView2 runtime installed. This is the same WebView2 runtime used by Blazor Hybrid for .NET MAUI Blazor.

The following requirements must be met to add Blazor functionality to an existing Windows Forms app:

- Update the Windows Forms app to target .NET 6.
- Update the SDK used in the app's project (csproj) file to **Microsoft.NET.Sdk.Razor**.
- Add a package reference to **Microsoft.AspNetCore.Components.WebView.WindowsForms**.
- Add the wwwroot/index.html file from **Listing 3** to the project, replacing {PROJECT NAME} with the actual project name.
- Add the app.css file from **Listing 4**, with some basic styles to the wwwroot folder.
- For all files in the wwwroot folder, set the **Copy to Output Directory** property to **Copy if newer**.
- Add a root Blazor component **Counter.razor** from **Listing 5** to the project.
- Add a BlazorWebView control in **Listing 6** to the desired form to render the root Blazor component.
- Run the app to see your BlazorWebView in action, shown in **Figure 12**.

To add Blazor functionality to an existing WPF app, follow the same steps listed above for Windows Forms apps, except:

- Substitute a package reference for **Microsoft.AspNetCore.Components.WebView.Wpf**
- Add the BlazorWebView control in XAML, as shown in **Listing 7**.
- Set up the service provider as a static resource in the XAML code-behind file (such as MainWindow.xaml.cs), as shown in **Listing 8**.
- To satisfy tooling requirements for the WPF runtime, add an empty partial class from **Listing 9** for the component in **Counter.razor.cs.**
- Build and run your Blazor based WPF app, shown in **Figure 13.**

What to Expect Next

The Blazor Hybrid pattern and .NET MAUI Blazor marks a huge milestone for .NET 6 and the work doesn't end there. .NET 7 is expected in November 2022 and with it, even more Blazor is anticipated. Another Blazor experiment, Blazor Mobile Bindings, is likely to ship in the .NET 7 release. Blazor Mobile Bindings is extension of the Xamarin > MAUI evolution that uses Razor syntax to define UI components and behaviors. By enabling the Razor syntax as a replacement for XAML, context switching becomes almost trivial. In .NET 7 with Blazor Mobile Bindings, .NET MAUI Blazor apps will have nearly identical coding patterns whether it's a native view (XML and Razor) or Web-based view (HTML and Razor). In **Listing 10**, a Counter component is composed of XML using the Razor syntax to create a naïve UI. The Counter component looks very similar to the HTML-based Web component, except for platform-native StackLayout, Label, and Button components defined in XML.

Just as .NET MAUI Blazor apps using XAML, application UI logic can be mixed using the BlazorWebView component. With the Razor, the experience is even more seamless as Razor directives and code blocks can be used and follow the same conventions as HTML-based components. In **Listing 11**, a hybrid view uses both native UI and BlazorWebView to display a counter while sharing app state among all components. The @inject directive is safe to use in the context of Blazor Mobile Bindings and provides dependency injection for native components just as it does for Web components. The similarities continue with Razor data binding on the Label and Button components. The Web component code, which will be rendered by the BlazorWebView, looks very similar regarding syntax, as shown in **Listing 12**.

Blazor Mobile Bindings looks promising. It continues to blur the boundaries between native and Web programming, further extending the usefulness of existing .NET skills. Blazor has found an audience with Web developers by offering a solution to a .NET audience where JavaScript was the only player. Blazor Hybrid and MAUI carry that same tradition with cross-platform native app development via Blazor Mobile Bindings.

A New Era of Blazor Productivity, Again

When .NET 3.0 was released, it included Blazor for the first time. At the time, I wrote an article entitled A New Era of Blazor Productivity (https://codemag.com/Article/1911052/A-New-Era-of-Productivity-with-Blazor). The sentiment I expressed there holds true again: "As powerful as it is convenient, Blazor makes a great choice for new applications. By combining .NET technologies that you're already using with an intuitive component model, Blazor has created a new era of productivity."

With the Blazor Hybrid pattern, .NET and related tools shorten the learning curve and makes cross-platform development approachable. When using a Blazor Hybrid pattern, the tradeoffs with WebAssembly are reduced. Xamarin evolves to MAUI and brings native APIs to Android, iOS, macOS, and Windows, while BlazorWebViews enable Web architecture. As Blazor grows, the community and ecosystem will continue to grow with solutions for common problems. Blazor's roadmap shows commitment and promise that Blazor is here to stay by delivering innovative technologies to help you build modern applications.

Ed Charbeneau

CODE

Listing 11: Main.razor

```
@inject CounterState CounterState

<ContentView>
    <StackLayout>

        <StackLayout Margin="new Thickness(20)">
            <Label
           Text="@($"You pressed {CounterState.CurrentCount} times")"
           FontSize="30" />
            <Button Text="Increment from native"
           OnClick="@CounterState.IncrementCount" Padding="10" />
        </StackLayout>

        <BlazorWebView ContentRoot="WebUI/wwwroot"
            VerticalOptions="LayoutOptions.FillAndExpand">
            <FirstBlazorHybridApp.WebUI.App />
        </BlazorWebView>

    </StackLayout>
</ContentView>

@code {
    // initialization code
}
```

Listing 12: App.razor

```
@inject CounterState CounterState

<div style="text-align: center; background-color: lightblue;">
    <div>
        <span style="font-size: 30px; font-weight: bold;">
            You pressed @CounterState.CurrentCount times
        </span>
    </div>
    <div>
        <button style="margin: 20px;" @onclick="ClickMe">
             Increment from HTML
            </button>
    </div>
</div>

@code
{
    private void ClickMe()
    {
        CounterState.IncrementCount();
    }

    // initialization code
}
```

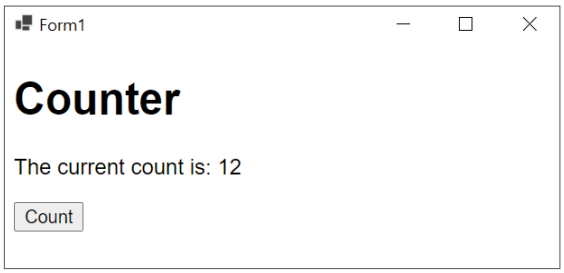

Figure 12: A Blazor Hybrid enabled WinForms application

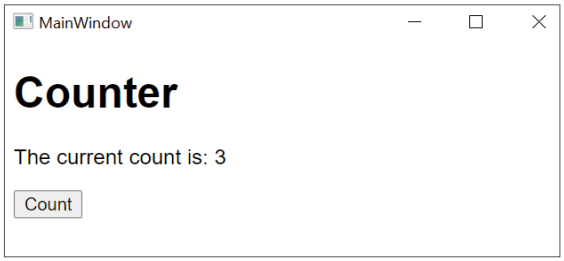

Figure 13: A Blazor Hybrid-enabled WPF application

Power Up Your Power Apps with .NET 6 and Azure

Power Apps are a great way to build business apps quickly and enable a citizen developer, who might be more familiar with understanding and solving business problems than the technical nuances associated with writing code, to quickly flesh out the design of an app and specify how it should function. Power Apps supports connectors that can integrate an app with a wide range of data sources and services that professional developers can author to enable these users to build specialized business apps quickly.

This article walks through how you can use .NET 6's new ASP.NET Core Minimal APIs to build an HTTP API and publish it to Azure App Service, then import the API into Azure API Management so you can integrate your API with multiple services and third parties, monitor it, and secure it with Microsoft Identity Platform. With the API imported into API Management and running nicely in Azure App Service, it makes a fantastic back-end API for a mobile Power App.

Brady Gaster
bradyg@microsoft.com
www.bradygaster.com
@bradygaster

Brady Gaster is a principal program manager in the Developer Division at Microsoft, where he works on SignalR, microservices, and HTTP APIs. He collaborates with teams across Microsoft in hopes to make it exciting for developers who work on .NET apps to party in the cloud. You can find Brady on Twitter when he's not learning with (or from) his two sons, tinkering with code, or making music in his home office using various synthesizers and guitars.

The Business Problem

Some local entrepreneurs I know work in the construction and civil engineering space, photographing complex work sites like cellular towers that need reconfiguration. Using Bing Maps or Google Earth to look at the pictures they get from their field photographers to ascertain the location in which they were standing when they took their site pictures is time consuming. After spending hours reviewing the work site, mapping software and the images, they can usually—within a certain degree of success—figure out how to redesign the site's configuration in their imaging software.

When I heard about this conundrum, I thought what any app developer thinks: It's time to build an app to solve this problem! After learning that most of their customers use Office 365 for their communication and productivity, a Power App solution seemed like a great idea, because Power Apps and Office 365 work so well together. Our team was nearing the release of ASP.NET Core 6, and I was so excited about minimal APIs, I thought I'd put the two together to do some "Fusion Development"—putting .NET, Azure, and Power Apps together to churn out a useful app quickly.

Prerequisites

To build a minimal API, all you need is .NET 6. If you plan on building Power Apps that use Azure compute resources, you'll need all the items listed below.

- **.NET 6:** You can download .NET at https://dot.net/download.
- **An Azure subscription:** You can try Azure for free by signing up at https://azure.microsoft.com.
- **The Azure CLI:** This can be downloaded from https://docs.microsoft.com/cli/azure/install-azure-cli.
- **A Power Apps subscription.** You can set up your own Power Apps subscription at https://powerapps.microsoft.com.
- **Visual Studio 2022:** The recommended development environment for the tasks in this article. You can download it from https://visualstudio.com/preview.

Minimal APIs with ASP.NET Core 6

Minimal APIs, new in .NET 6, are a low-ceremony way to build Web apps, small microservices, or HTTP APIs using ASP.NET Core. Minimal APIs hook into all the hosting and routing capabilities you know in ASP.NET Core, but they're based on C# top-level statements from a code-cleanliness perspective. There are few subtle changes between the process of creating a minimal API and a Web API, but for getting started with a new API or for experimentation, minimal can't be beat.

The first difference you'll notice is that a minimal API project is indeed just that: the bare minimum amount of code and configuration you need to get going, and that there's no **Startup.cs** file. **Figure 1** shows the simplicity of a minimal API project, or at least the minimum amount required to get going.

Minimal Means Less Ceremony Required to GET OK

Traditional Web API projects not only require you understand the project layout—**Controllers** folder, **Startup.cs** for some things, **Program.cs** for others—but there's a bit of ceremony involved to get an HTTP API outputting a string from a Web server. In a traditional Web API project's Controllers folder, you'd have a **HelloWorldController.cs** file. The **HelloWorldController.cs** file would represent a class in the compiled app, with a single method that responds when an HTTP request is made to the server's "/hello" endpoint.

```
using Microsoft.AspNetCore.Mvc;

namespace SideBySide.WebApi.Controllers
{
    [Route("hello")]
    [ApiController]
    public class HelloWorldController
      : ControllerBase
    {
        [HttpGet]
        public ActionResult<string> Get()
        {
            return new
              OkObjectResult("Hello World");
        }
    }
}
```

This Controller wouldn't even respond without the also-requisite wire-up in a separate file, **Startup.cs,** that enables MVC controllers, which is what Web API is built upon.

```
public void ConfigureServices(
    IServiceCollection services)
{
    services.AddControllers();
}
```

This means that you have two files to edit (at a minimum), a few concepts to have to understand: the project structure, controllers, routing attributes, and so on. With minimal APIs, Microsoft has reduced the amount of code you'd need to write to respond to an HTTP request. Rather than create a controller file (and a class in your assembly), you simply build the host and route the traffic to your code.

```
var builder =
    WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/hello", () => "hello world");

app.Run();
```

With minimal APIs, you get the same result **in four lines** versus 18 lines across multiple files, concepts, and more. With that introduction and summary of how awesome minimal APIs are in ASP.NET Core with .NET 6, let's dive into how it's going to be easier for local entrepreneurs—who we'll call Contoso Construction for the purpose of this article—to make it easier for construction and civil engineering site managers to figure out how to interpret job site photography.

The Contoso Construction API

This app's API will need to support very few elements of data; you'll record the precise geo-coordinates of a job site and the images taken of that job site from various angles, along with the compass heading of each photo. With those elements of data, the site manager can be confident that they know **exactly** where each photo was taken. The API you build as a back-end for the Power App will need to support the following features:

- View all job sites.
- Create new job sites.
- Upload new photos of job sites, complete with the geographic and compass metadata.
- View photos of a job site, along with the geo-location of the photographer at the time of the shot, because the site artists value not only knowing the address of the work site, but the actual geolocation of each photo.

For the purposes of simplicity, I'll forego the potential advantages of integrating with the Microsoft Identity Platform or Microsoft Graph API to augment the API with organizational data for the employees in the Azure Active Directory tenant or to distribute job sites by user, but that'd be a great follow-up topic.

Creating the Azure Resources Using Azure Bicep

The API allows app users to upload images, which are stored in Azure blob containers. Metadata about the images—the geo-coordinates of their location and their compass heading data, to be specific—will be stored in an Azure SQL Database using Entity Framework Core. Key Vault securely stores connection strings, so an Azure Web App, in which my .NET 6 minimal API code will be hosted, can connect to the storage and database securely (and so I'll never accidentally commit secrets to source control).

Bicep, a domain-specific language (DSL) that uses declarative syntax to describe Azure resources, their configuration, and their relationships with one another, offers developers a syntax more elegant than JSON for creating and configuring cloud applications. The GitHub repository containing the sample code for this article (located here: https://github.com/bradygaster/Contoso.Construction/) includes a Bicep file, **deploy.bicep.** It also contains a PowerShell script, **setup.ps1**, to run to create the entire Azure topology shown in **Figure 2**, then build the .NET source code, package it into a zip file, and publish it to Azure Web Apps.

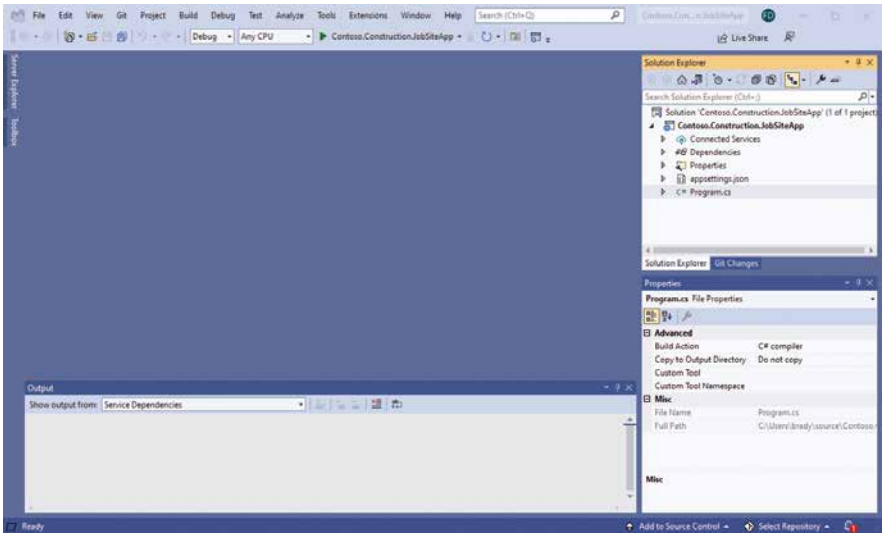

Figure 1: A minimal Web API project structure

Contoso's field app enables their field photographers to photograph job sites, so the essential technical requirements are to store uploaded images taken from a phone camera, along with some string or numeric data associated with each photo. From an app data flow perspective:

- The user uploads a photo taken by mobile phone camera.
- The image is stored in Azure Blob storage.
- A URL of image, geographic, and compass metadata associated with the image is stored in an Azure SQL Database.

The **deploy.bicep** file included with this article's sample code creates all the Azure resources required to host the minimal API code securely and enables use of it to the Job Site Power App, starting with the Azure SQL database and server, using the username and password parameters passed in as top-level parameters of the Bicep template file.

```
Resource sqlServer
  Microsoft.Sql/servers@2014-04-01'
={
  name: '${resourceBaseName}srv'
  location: resourceGroup().location
  properties: {
    administratorLogin: sqlUsername
    administratorLoginPassword: sqlPassword
  }
}

Resource sqlServerDatabase
  'Microsoft.Sql/servers/databases@2014-04-01'
= {
  parent: sqlServer
  name: '${resourceBaseName}db'
  location: resourceGroup().location
  properties: {
    collation: 'SQL_Latin1_General_CP1_CI_AS'
    edition: 'Basic'
```

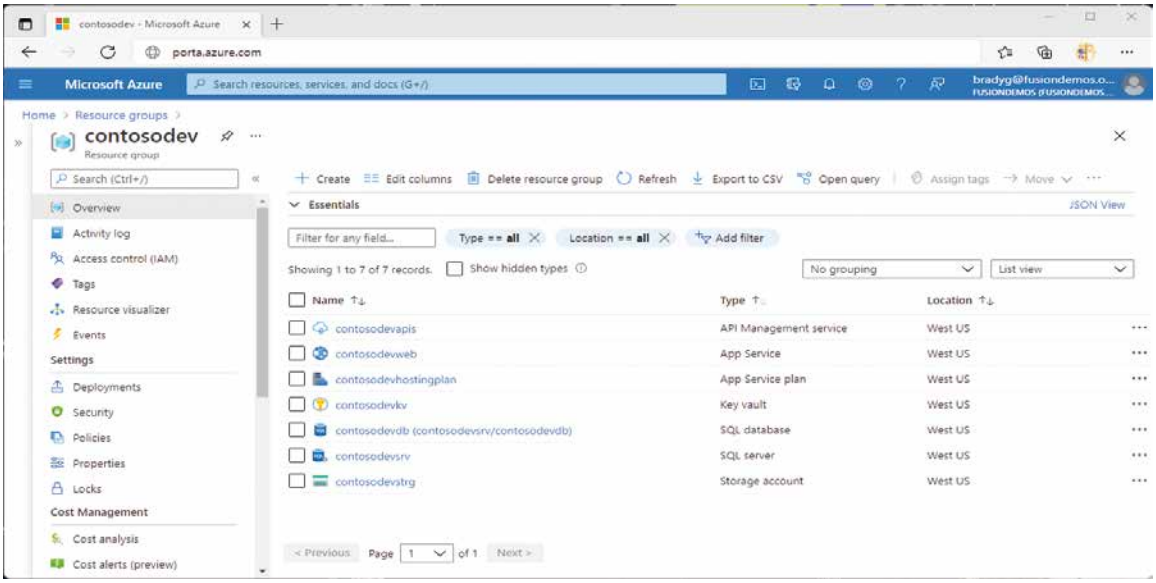

Figure 2: The Azure resources used by the Contoso Construction app

The Fusion Development Approach

Azure advocates and engineers across the Developer Division and Power Platform at Microsoft worked together to author an introductory ebook on Fusion Development. Download it from https://docs.microsoft.com/powerapps/guidance/fusion-dev-ebook/.

```
    maxSizeBytes: '2147483648'
    requestedServiceObjectiveName: 'Basic'
  }
}
```

The Azure Storage account and blob container get created once the SQL database server's creation completes.

```
Resource storage
  'Microsoft.Storage/storageAccounts@2021-02-01'
 = {
  name: '${resourceBaseName}strg'
  location: resourceGroup().location
  kind: 'StorageV2'
  sku: {
    name: 'Standard_LRS'
  }
}

Resource storageBlobContainer
  'Microsoft.Storage/storageAccounts
    /blobServices/containers@2021-04-01'
= {
  name: format('{0}/default/uploads',
   storage.name)
  dependsOn: [
    storage
  ]
  properties: {
    publicAccess: 'Blob'
  }
}
```

Stored as secure secrets in an Azure Key Vault, the SQL and Storage connection strings can then be read at runtime by the .NET minimal API code. Because those credentials are stored in Key Vault, the API connects to SQL and Storage securely, and you never have to worry about leaking a connection string.

```
Resource sqlSecret
'Microsoft.KeyVault/vaults/secrets
= {
  parent: keyVault
  name: 'ConnectionStrings--
AzureSqlConnectionString'
  properties: {
    value: 'Data Source=
tcp:${sqlServer.properties
        .fullyQualifiedDomainName},
   1433; Initial
   Catalog=${resourceBaseName}db;User
   Id=${sqlUsername};
   Password=${sqlPassword};'
  }
}

Resource storageSecret
  'Microsoft.KeyVault/vaults/secrets
    @2021-06-01-preview' = {
  parent: keyVault
  name: 'AzureStorageConnectionString'
  properties: {
    value: format('DefaultEndpointsProtocol=
      https;AccountName=${storage.name};
        AccountKey=${listKeys(storage.name,
            storage.apiVersion).keys[0].value};
            EndpointSuffix=core.windows.net')
  }
}
```

A setup script, **setup.ps1**, wraps up behind the environment creation and deployment process. Executing **setup.ps1** from the article's repository, providing the base resource name (in this article's case, **contosodev**), creates the Azure resources shown in **Figure 2**. It then compiles the .NET minimal API code, publishes it into a local self-contained build, and zips it up. The script then uploads the zip file to Azure Web Apps and starts up.

```
.\setup.ps1 -resourceBaseName
<your-desired-name>
```

Now I'll step through each part of the .NET minimal API code—all of which you'll find in the **Program.cs** file of the repository for this article—to talk about how each of those Azure resources will be used by the .NET code.

Connect to Azure Resources via the Azure SDK

The Bicep deployment template creates two secrets in Key Vault: one for the Azure SQL Database and the other for the Azure Storage account. The .NET Core Azure Key Vault configuration provider reads those two configuration elements that will be loaded in from the Azure Key Vault, not from **appsettings.json**. If you look at the Azure Key Vault secrets in the Azure portal after executing the Bicep template against your subscription, you'll see the two secrets shown in **Figure 3**.

The minimal API code starts with host build-up, configuration, and service registration. Much like my traditional Web API code began with wiring up services in **Startup.cs**, so too will I begin my development with minimal APIs. I'll go ahead and read the **VaultUri** environment variable and use it to connect the minimal API code to the Azure Key Vault.

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

// Add Key Vault provider
var uri = Environment
    .GetEnvironmentVariable("VaultUri");

builder.Configuration.AddAzureKeyVault(
            new Uri(uri),
            new DefaultAzureCredential());
```

Calls to the configuration system for the SQL database and Azure Storage connection strings are routed to Key Vault, and the minimal API code uses the traditional approach of injecting the Entity Framework Core **DbContext** class. See **Listing 1** for a look at the Entity Framework data objects. After the Entity Framework Core database context is injected into the minimal API's service collection it uses the Azure Extensions library's **AddAzureClients** method to add an Azure Storage client to the services collection, too.

```
// Add the Entity Framework Core DBContext
builder.Services.AddDbContext<JobSiteDb>(_ =>
{
    _.UseSqlServer(
        builder.Configuration
            .GetConnectionString(
                "AzureSqlConnectionString"));
});
```

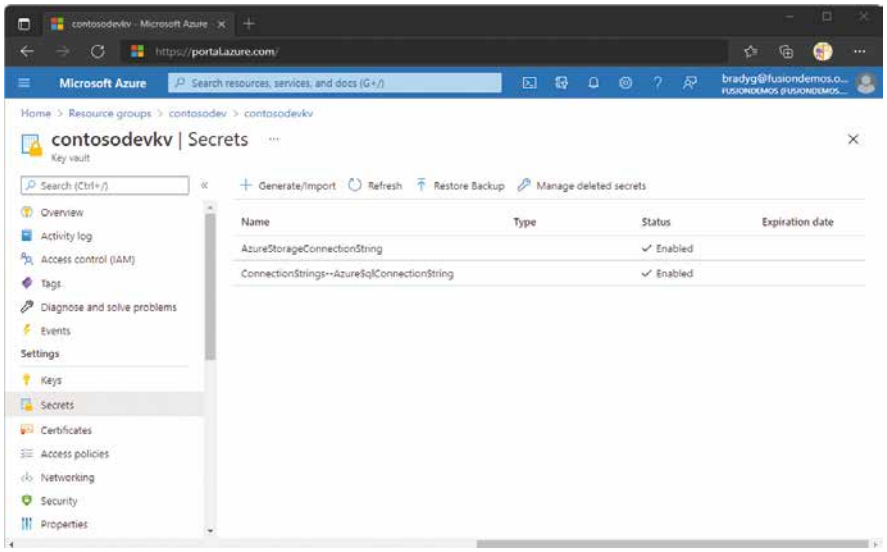

Figure 3: The SQL database and Azure Storage connection strings in Azure Key Vault

```
// Add Azure Storage services to the app
builder.Services.AddAzureClients(_ =>
{
    _.AddBlobServiceClient(
        builder.Configuration
            ["AzureStorageConnectionString"]
        );
});
```

Routing HTTP Traffic with Minimal API Methods

With the minimal API configured and connected to Azure resources, the last piece I'll explore is the routes. Each API endpoint is routed individually, so you have a 1:1 correlation between each HTTP endpoint in the API and the code that executes to satisfy the request. **MapGet** denotes that any HTTP GET request to the **/jobs** endpoint will be handled by this method.

Minimal is smart enough to know when each route method parameter should come from the path or from the services collection. You don't need to add **FromServices**—it just works. The **Produces** method is useful for specifying the shape of the API's

The OpenAPI Specification

The OpenAPI Specification exists as a language-agnostic interface to RESTful APIs that allows both humans and computers to understand a service's capabilities without the need for source code or documentation. Learn more about the OpenAPI Specification at https://swagger.io/specification/.

Listing 1: The Entity Framework Core data objects

```
public class JobSitePhoto
{
    [DatabaseGenerated(
        DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }
    public int Heading { get; set; }
    public int JobId { get; set; }
    public double Latitude { get; set; }
    public double Longitude { get; set; }
    public string PhotoUploadUrl { get; set; }
        = string.Empty;
}

public class Job
{
    [DatabaseGenerated(
        DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }
    public double Latitude { get; set; }
    public double Longitude { get; set; }
    public string Name { get; set; }
        = string.Empty;
    public List<JobSitePhoto> Photos
        { get; set; } = new List<JobSitePhoto>();
}

class JobSiteDb : DbContext
{
    public JobSiteDb(
        DbContextOptions<JobSiteDb> options)
        : base(options) { }

    public DbSet<Job> Jobs
        => Set<Job>();

    public DbSet<JobSitePhoto> JobSitePhotos
        => Set<JobSitePhoto>();

    protected override void OnModelCreating(
        ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Job>()
                .HasMany(s => s.Photos);

        base.OnModelCreating(modelBuilder);
    }
}
```

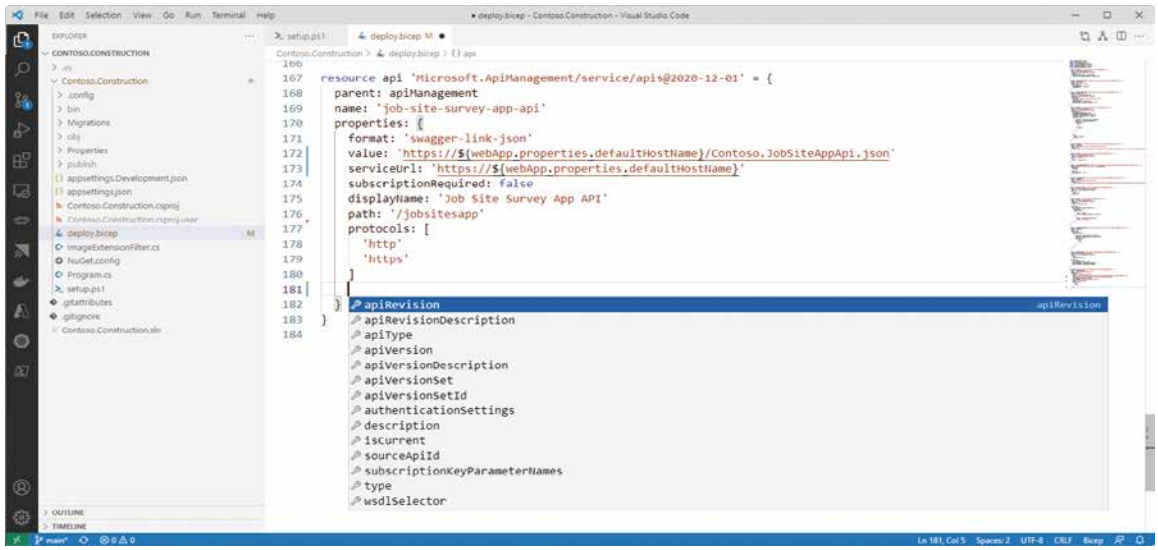

Figure 4: Importing an API into Azure API Management using Bicep and Visual Studio Code

response payload and output content type. Minimal API's **WithName** method is useful when you know you'll want to integrate your minimal API using OpenAPI descriptions. The **WithName** method sets the value of the OpenAPI **operationId** attribute, which is used heavily by code generators and API-consuming tools and services (like Power Apps, which I'll explore next).

```
// Enables GET of all jobs
app.MapGet("/jobs",
    async (JobSiteDb db) =>
        await db.Jobs.ToListAsync()
    )
    .Produces<List<Job>>(
       StatusCodes.Status200OK)
    .WithName("GetAllJobs");
```

Route methods support route arguments or parameters. The **GetJob** API method takes an input parameter and then uses it in a LINQ expression to query a SQL database using Entity Framework Core. In the **GetJob** API method code, the API either returns an HTTP 200 when the job site record is found, or an HTTP 404 when there's no matching record. This demonstrates a canonical example of how multiple **Produces** calls specify a variety of potential request/response scenarios based on the API's business logic.

```
// Enables GET of a specific job
app.MapGet("/jobs/{id}",
    async (int id, JobSiteDb db) =>
        await db.Jobs
                .Include("Photos")
                    .FirstOrDefaultAsync(_ =>
                        _.Id == id)
            is Job job
                ? Results.Ok(job)
                : Results.NotFound()
    )
    .Produces<Job>(StatusCodes.Status200OK)
    .Produces(StatusCodes.Status404NotFound)
    .WithName("GetJob");
```

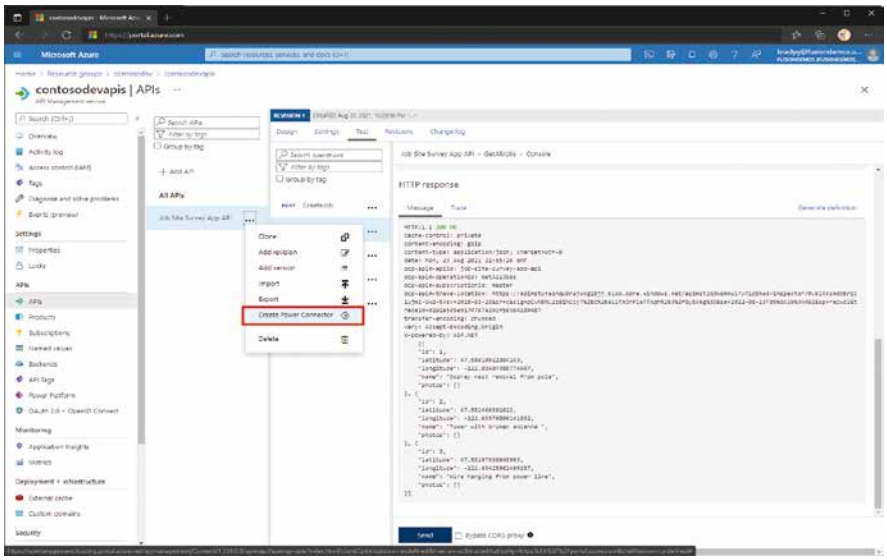

Figure 5: Creating a Power Platform Custom Connector from the API

The **CreateJob** API method accepts an incoming Job object and uses Entity Framework to create a new record in the Jobs table, demonstrating the simplicity of handling complex body payloads coming via HTTP POST requests.

```
// Enables creation of a new job
app.MapPost("/jobs/",
    async (Job job,
        JobSiteDb db) =>
    {
        db.Jobs.Add(job);
        await db.SaveChangesAsync();

        return Results.Created(
            $"/jobs/{job.Id}", job);
    })
    .Produces<Job>(StatusCodes.Status201Created)
    .WithName("CreateJob");
```

You can mix route parameters, body parameters, even accept form posts and file uploads. **Listing 2** shows the API's file-uploading route, which accepts image uploads taken by a Power App UI with a camera.

Listing 2: A complex API with route and HTTP file parameters

```
app.MapPost(
    "/jobs/{jobId}/photos/{lat}/{lng}/{heading}",
    async (HttpRequest req,
        int jobId,
        double lat,
        double lng,
        int heading,
        BlobServiceClient blobServiceClient,
        JobSiteDb db) =>
    {
        if (!req.HasFormContentType)
        {
            return Results.BadRequest();
        }

        var form = await req.ReadFormAsync();
        var file = form.Files["file"];

        if (file is null)
            return Results.BadRequest();

        using var upStream =
           file.OpenReadStream();

        var blobClient = blobServiceClient
              .GetBlobContainerClient("uploads")
                  .GetBlobClient(file.FileName);

        await blobClient.UploadAsync(upStream);

        db.JobSitePhotos.Add(new JobSitePhoto
        {
            JobId = jobId,
            Latitude = lat,
            Longitude = lng,
            Heading = heading,
            PhotoUploadUrl =
                blobClient.Uri.AbsoluteUri
        });

        await db.SaveChangesAsync();

        var job = await db.Jobs
                    .Include("Photos")
                    .FirstOrDefaultAsync(x =>
                        x.Id == jobId);

        return Results.Created(
            $"/jobs/{jobId}", job);
    })
    .Produces<Job>(StatusCodes.Status200OK,
        "application/json")
    .WithName("UploadSitePhoto");
```

Integrating the API in New Ways with OpenAPI

My favorite part of the set up script process in **deploy.bicep** is when you import the OpenAPI description from the newly-deployed app service into Azure API management. When this part of the Bicep script runs, the OpenAPI description automatically generated from the minimal API endpoints is imported into Azure API management. **Figure 4** shows this exciting final step in the Bicep deployment process, and how the Bicep extension for Visual Studio Code provides in-editor support and assistance. It also comes pre-packaged with dozens of fantastic snippets, so you don't have to guess as you're learning the Bicep template syntax.

Azure API Management enables you to take all your individual APIs running in App Service, in Azure Functions, in Kubernetes—or even on-premises—and have a single place to configure, secure, and monitor them. APIs imported into Azure API Management can be used in a variety of ways, like being exported to Power Platform as a Power Apps Custom Connector. Custom Connectors are the "glue" that connects the Power Platform—Power Automate, Power Apps, Power BI, and so on—to your APIs running in Azure. In **Figure 5** you'll see how, in the Azure portal, API management enables Power Platform export.

Use the API in a Low-Code Mobile Power App

The mobile app built for this article within Power Apps has five screens, two of which are simple maps that plot out job sites or photographs of those job sites and their point of origin. Field engineers click pins in the job site map screen shown in **Figure 6** to navigate to a second map showing clickable icons of each of the job site photos. When users click an icon in the photos map, photographs pop up in Info Boxes taken at that specific location. This way, the field engineers can make sure they've taken photos of the job sites from all possible angles.

The job site map screen binds to a Collection named **_jobSiteCollection.** In the Power App's **OnStart** handler, the app calls the minimal API, clearing and re-filling the collection. **Figure 7** shows how to use the Data tab in Power Apps to search for a custom connector, as well as the **OnStart** handler in which the API will be called to load the collection of job sites. As the app starts, **OnStart** fires, and the app calls the API via the **JobSiteSurveyAppAPI** connector's **GetAllJobs** method.

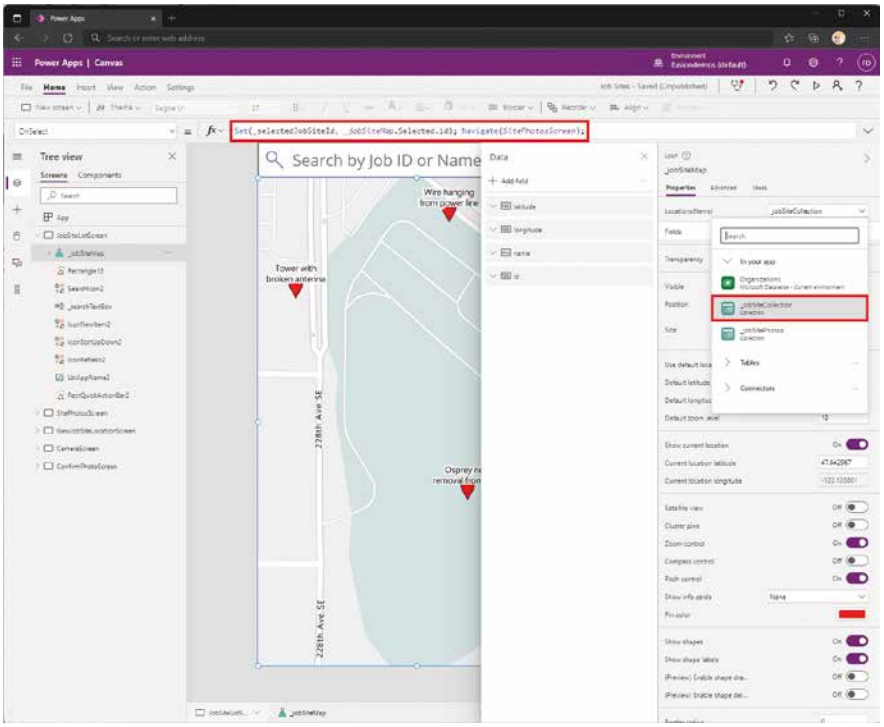

Figure 6: Binding a map to a collection and handling item selection

The Power Apps Monitor helps with API debugging, as it shows the requests and responses going to and from your API. The monitor, shown in **Figure 8**, is great to have open in another tab as build your app against your HTTP API.

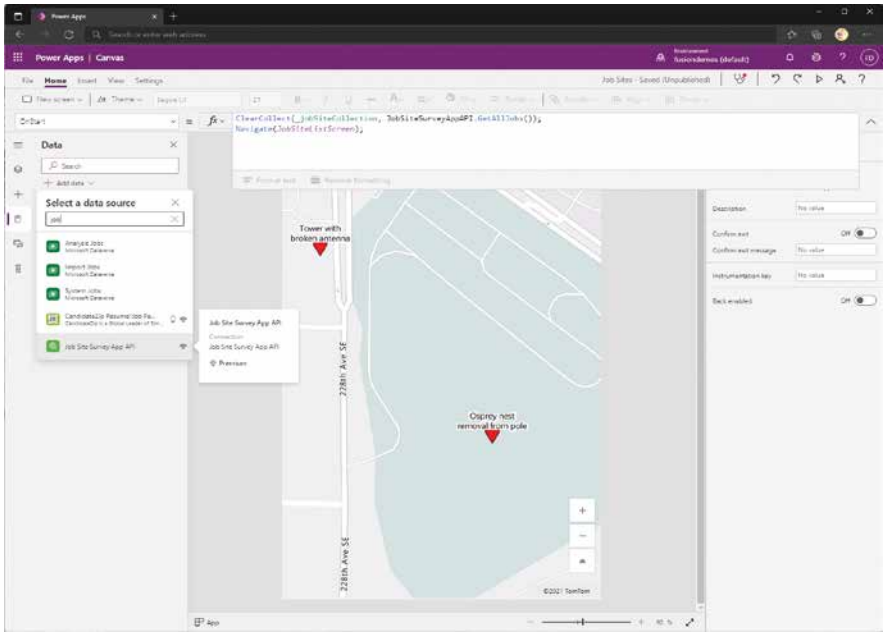

Figure 7: Adding an API's Custom Connector as a data source to a Power App

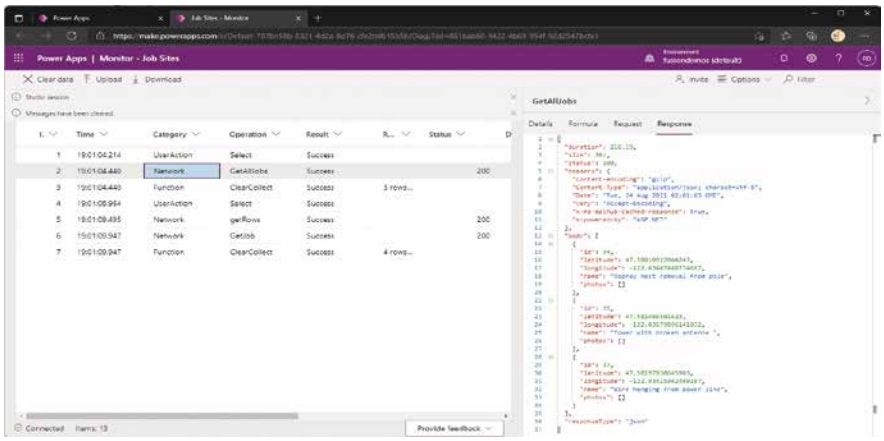

Figure 8: Debugging HTTP calls to an API using the Power Apps Monitor

Summary

If the idea of fusing together .NET, Azure, and Power Apps to get started on your low-code journey sounds appealing after reading this article, spend some time looking at the Microsoft Learn topics on fusion and low-code development (here: https://docs.microsoft.com/en-us/learn/paths/transform-business-applications-with-fusion-development/). There are a variety of other Learn paths available that go deeper on the topic of Fusion development.

If you're less interested in the UI parts of this end-to-end and mainly want to build APIs, you're sure to enjoy using .NET 6 minimal APIs with Bicep to make your API development and deployment much simpler than ever before, with less ceremony. I think experienced API developers are going to enjoy this new approach and that new API developers will find it a more graceful way to get started.

Happy coding! Be well!

Brady Gaster

CODE

CODE Focus Nov 2021
Volume 18 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
Ed Charbeneau
Brady Gaster
Julie Lerman
Daniel Roth
Steven Thewissen
Mika Dumont
Rich Lander
Mark Michaelis
Mike Rousos

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

Back inside cover .NET artwork to come from MS

Back outside cover .NET artwork to come from MS