Text, Text, Text, Text, Text, Text, Text, Text, Text, Text, Text

CODE

SEP
2020

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

CODE FOCUS

Title Text
TextText Title
Title Title

Title Text
TextText Title
Title Title

Title Text
TextText Title
Title Title

Azure is the best cloud for .NET

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



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



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



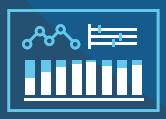
Faster, simpler development
Develop more easily with one-click deployment in Visual Studio or set up a CI/CD pipeline for your app in minutes.



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



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



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

Start FREE at:

azure.microsoft.com/free/dotnet

Microsoft

Microsoft



Innovate with Azure

Increase productivity and enable faster app development

Simplify your journey with the Azure Migration Program. Get proactive guidance and the right mix of expert help to ensure you can migrate with confidence.

www.azure.com/AMP

Features

8 From .NET Standard to .NET 5

One of the most exciting aspects of .NET 5 is the unification of implementation tools. Immo reveals how .NET 5 determines which implementation supports which APIs, cites warnings when you've written something unsupported, and reuses libraries for app components written for various versions of .NET.

Immo Landwerth

14 Introducing C# 9.0

If you liked C# before, you're really going to like C#9.0. Bill explains how the C# compiler that ships with the .NET 5 SDK has been updated and streamlined.

Bill Wagner

20 EF Core 5: Building on the Foundation

Julie's pretty excited about the new features in EF Core 5. You will be too when you read about the bugs fixed and over 200 new features and minor enhancements.

Julie Lerman

29 Project Tye: Creating Microservices in a .NET Way

Using the experimental developer's tool called Tye turns testing and deploying microservices into a sweet operation. .NET just works and Shayne shows you how.

Shayne Boyer

40 Big Data and Machine Learning in .NET 5

The theme of .NET 5 is unification. Jeremy and Bri show you how a uniform runtime and framework are just the beginning.

Jeremy Likness and Bri Achtman

48 F# 5: A New Era of Functional Programming with .NET

F# has been updated right along with .NET. But it's not just conveniently included. It marks the beginning of a new era and Phillip tells you why you'll be glad you've got it.

Phillip Carter

55 Xamarin.Forms 5: Dual Screens, Dark Modes, Designing with Shapes, and More

David takes an exciting look at the Surface Duo to show us the power of Xamarin.Forms 5.

David Ortinau

62 .NET 5.0 Runtime Highlights

Runtime got a lot of attention as the .NET platform became more unified. Richard tells you what's new and what to look forward to.

Richard Lander

68 Blazor Updates in .NET 5

You already know about the power of Blazor. Daniel shows you how the .NET 5 version uses .NET instead of JavaScript to build full-stack Web apps.

Daniel Roth

0 Windows Desktop Apps and .NET 5

The .NET collection of tools is finally all in the same place as the Windows development tools. Olia explores the changes and tells you how to get started migrating your existing applications to .NET 5.

Olia Gavrysh

0 Azure Tools for .NET in Visual Studio 2019

Visual Studio 2019 automatically discovers your app's Azure requirements and helps you configure them. Angelos shows you how to choose between live Azure services and local emulators.

Angelos Petropoulos

Departments

6 The Journey to One .NET

Beth talks about her coding journey and why she's so excited about .NET 5.

Beth Massi

15 Advertisers Index

74 Code Compilers

US subscriptions are US $29.99 for one year. Subscriptions outside the US pay $49.99 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.

The Journey to One .NET



When Rod Paddock, Editor in Chief of this magazine, asked me to write the editorial, I couldn't pass up the opportunity. The first time I was published in CODE Magazine was back in the fall of 2000, just after the unveiling of the .NET platform at Microsoft's Professional Developer's Conference. I have to say that I've come a long way in 20 years and so has .NET.

Back then, I was a FoxPro kid writing about "distributed Internet applications" and arguing with Visual Basic developers about what development environment was better. I then started developing with the very early versions of .NET and never looked back to FoxPro. It was the early days of the "programmable Web" and .NET was built for it. .NET allowed me to do things I never could before. Now I work at Microsoft as the Product Marketing Manager for .NET and am on the Board of Directors of the .NET Foundation. Yes, we've both come a long way.

The release of .NET 5 marks a pivot point in .NET's long history of enabling developers to be productive writing any type of application. Over the years, there have been multiple implementations and versions to cover all the app types starting with .NET Framework for Windows, Mono, and Xamarin for mobile, and of course the cross-platform .NET Core. All of these implementations have their own libraries and APIs, project systems, runtimes, and components. Luckily, the languages remain relatively consistent and the .NET standard API specification helps the ecosystem share libraries across the implementations. .NET 5 begins the journey to unify these. The goal is to simplify the choices and learning curve for new developers, at the same time making it easier for experienced developers to build anything.

.NET Core has taken the best of .NET Framework, adding support for Windows Forms and WPF, expanding support for more devices, chipsets, operating systems, and distros. It's got dramatically improved performance and memory usage. When .NET Core 3 released last year, it was the fastest adopted version of .NET **ever**. .NET Framework 4.8 for Windows was the final minor release last year. It will only get critical bug fixes from now on and will remain a component of Windows. As long as Windows is supported, .NET Framework is supported.

.NET 5 is the next version and future of .NET that releases November 10, 2020. .NET 6 will release in November 2021 and there will be subsequent major releases every year. We're continuing the journey of unifying the .NET platform, with a single framework that extends from cloud to desktop to mobile and beyond. The next step in the journey is to take .NET Core and Mono/Xamarin implementations and unify them into one base class library (BCL) and toolchain (SDK). You'll see this happen in the .NET 5 to 6 wave of releases. The unification will be completed with .NET 6, our Long-Term Support (LTS) release.

> .NET 5 releases
> November 10, 2020.

This special CODE FOCUS issue is all about .NET 5 and many of the improvements we've made with the open source community across the platform. The magazine is your chance to dig deep into the features of the release. .NET 5 has several enhancements, such as smaller, faster, single file applications that use less memory, which are appropriate for microservices and containerized applications across operating systems. It also includes significant performance improvements, adds support for Windows ARM64, and incorporates new releases of the C# 9.0 and F# 5.0 languages. It includes updates to Xamarin, Windows Forms, WPF, ASP.NET, and Blazor as well as significant new runtime features, tools, and libraries. Visual Studio, Visual Studio Code, and Visual Studio for Mac all support .NET 5.

Here are a few highlights in this issue.

For the C# language aficionado, we have details on how to take advantage of the new record declaration syntax, as well as an introduction of top-level statements.

Fans of F# will love the new built-in package management as well as the ability to integrate F# with Jupyter Notebooks.

.NET 5 has tooling that will be appreciated by user interface developers. The inclusion of and improvements to WPF and WinForms is a huge step for the .NET developer. There's also a great write-up discussing many improvements to Blazor technology. Mobile UI technologies are also discussed with details on updates to the Xamarin ecosystem.

There's also great content on many of the underlying tools used by .NET developers. This includes advances to EF Core, like better many-to-many support, logging, and better filter support. There's also a discussion of runtime changes including improvements to single file deployments, ARM support, and performance improvements in general.

You'll also find articles on Project Tye that helps build microservice-based applications, Improvements to Azure tooling in Visual Studio, as well as an update on what's been updated in the machine learning framework ML.NET

This issue is chock full of great details that will help you take advantage of these features right out of the gate. I'm truly excited for the future of .NET and hope you love .NET 5 as much as I do.

Beth Massi
CODE

From .NET Standard to .NET 5

The base class library (BCL) provides the fundamental APIs that you use to build all kinds of applications, no matter whether they are console apps, class libraries, desktop apps, mobile apps, websites or cloud services. One of the challenges we had at Microsoft was making the BCL easier to reason about. This includes questions like "which APIs are available on which .NET implementation," "do I use any APIs that aren't supported on all the operating systems that I want to run on," "do I use problematic APIs," and of course, "do I use the APIs correctly?"



Immo Landwerth
immol@microsoft.com
devblogs.microsoft.com/dotnet/
twitter.com/terrajobst

Immo Landwerth is a program manager on the .NET platform team at Microsoft. He works on the class libraries where he focuses on open source, cross-platform, and API design.

In this post, I'm going to tell you about how we're making it easier for you to answer these questions moving forward.

The Future of .NET Standard

.NET 5 will be a shared code base for .NET Core, Mono, Xamarin, and future .NET implementations.

To better reflect this, we've updated the target framework names (TFMs). TFMs are the strings you use to express which version of .NET you're targeting. You see them most often in project files and NuGet packages. Starting with .NET 5, we're using these values:

- **net5.0**. This is for code that runs everywhere. It combines and replaces the **netcoreapp** and **netstandard** names. This TFM will generally only include technologies that work cross-platform (except for pragmatic concessions, like we already did in .NET Standard).
- **net5.0-windows**. These kinds of TFMs represent OS specific flavors of .NET 5 that include net5.0 plus OS-specific bindings. In the case of **net5.0-windows,** these bindings include Windows Forms and WPF. In .NET 6, we'll also add TFMs to represent the mobile platforms, such as **net6.0-android** and **net6.0-ios**, which include .NET bindings for the Android and iOS SDKs.

There isn't going to be a new version of .NET Standard, but .NET 5 and all future versions will continue to support .NET Standard 2.1 and earlier. You should think of net5.0 (and future versions) as the foundation for sharing code moving forward.

Advantages of Merging .NET Core and .NET Standard

Before .NET 5, there were completely disjointed implementations of .NET (.NET Framework, .NET Core, Xamarin, etc.). In order to write a class library that can run on all of them, we had to give you a target that defines the set of shared APIs, which is exactly what .NET Standard is.

This means that every time we want to add a new API, we have to create a new version of .NET Standard and then work with all .NET platforms to ensure that they add support for this version of the standard. The first problem is that this process isn't very fast. The other problem is that this requires a decoder ring that tells you which version of which .NET platform support needs which version of the standard.

But with .NET 5, the situation is very different. We now have a shared code base for all .NET workloads, whether it's desktop apps, cloud services, or mobile apps. And in a world where all .NET workloads run on the same .NET stack, we don't need to artificially separate the work into API definition and implementation work. We just add the API to .NET and the next time the stack ships, the implementation is instantaneously available for all workloads.

This doesn't mean that all workloads will have the exact same API surface because that simply wouldn't work. For example, Android and iOS have a huge amount of OS APIs. You'll only be able to call those when you're running on those appropriate devices.

The new TFMs I mentioned earlier solve this problem: **net5.0** (and future version) represent the API set that's available to all platforms. On top of that, we added OS-specific TFMs (such as **net5.0-windows**) that have everything in **net5.0** plus all the APIs that are specific to Windows (such as Windows Forms and WPF). And in .NET 6, we're extending this to Android and iOS by adding **net6.0-android** and **net6.0-ios**. The naming convention solves the decoder ring problem: Just by looking at the names, it's easy to understand that an app targeting **net6.0-ios** can reference a library built for **net5.0** and **net6.0** but not a library that was built for **net6.0-android** or **net5.0-windows**.

What You Should Target

.NET 5 and all future versions will always support .NET Standard 2.1 and earlier. The only reason to retarget from .NET Standard to .NET 5 is to gain access to more APIs. So you can think of .NET 5 as .NET Standard vNext.

What about new code? Should you still start with .NET Standard 2.0 or should you go straight to .NET 5? It depends on what you're building:

- **App components**. If you're using libraries to break down your application into several components, my recommendation is to use **netX.Y** where **X.Y** is the lowest number of .NET that your application (or applications) are targeting. For simplicity, you probably want all projects that make up your application to be on the same version of .NET because it means you can assume the same BCL features everywhere.
- **Reusable libraries**. If you're building reusable libraries that you plan to ship on NuGet, you'll want to consider the trade-off between reach and API set. .NET Standard 2.0 is the highest version of .NET Standard that's supported by .NET Framework, so it will give you the most reach, while also giving you a fairly large API set to work with. We'd generally recommend against targeting .NET Standard 1.x as it's not worth the hassle anymore. If you don't need to support .NET Framework, you can go with either .NET Standard 2.1 or .NET 5. Most code can probably skip .NET Standard 2.1 and go straight to .NET 5.

So what should you do? My expectation is that widely used libraries will end up multi-targeting for both .NET Standard

2.0 and .NET 5: supporting .NET Standard 2.0 gives you the most reach while supporting .NET 5 ensures that you can leverage the latest platform features for customers that are already on .NET 5.

In a couple of years, the choice for reusable libraries will only involve the version number of **netX.Y**, which is basically how building libraries for .NET has always worked—you generally want to support some older version in order to ensure that you get the most reach.

To summarize:

- Use **netstandard2.0** to share code between .NET Framework and all other platforms.
- Use **netstandard2.1** to share code between Mono, Xamarin, and .NET Core 3.x.
- Use **net5.0** for code sharing moving forward.

Platform-Specific APIs

The goal of .NET Standard has always been to model the set of APIs that work everywhere. And we started with a very small set. Too small, as it turns out, which is why we ended up bringing back many .NET Framework APIs in .NET Standard 2.0. We did this to increase compatibility with existing code, especially NuGet packages. Although most of these APIs are general purpose, cross-platform APIs, we also included APIs that only work on Windows.

In some cases, we were able to make these APIs available as separate NuGet packages (such as **Microsoft.Win32.Registry**), but in some cases we couldn't because they were members on types that were already part of .NET Standard, for example APIs to set Windows file system permissions on the **File** and **Directory** classes. Moving forward, we'll try to avoid designing types where only parts of them work everywhere. But as always, there will be cases where we couldn't predict the future and are forced to throw a **PlatformNotSupportedException** for some operating system down the road.

Dealing with Windows-Specific APIs

Wouldn't it be nice if Visual Studio could make you aware when you accidentally call a platform-specific API? Enter the platform compatibility analyzer. It's a new feature in .NET 5 that checks your code for usages of APIs that aren't supported on all the platforms you care about. It's a Roslyn analyzer, which means that it's running live in the IDE as you're editing code but will also raise warnings when you're building on the command line or the CI computer, thus making sure that you don't miss it.

Let's look at an example. I have a logging library that I originally wrote for .NET Framework but now want to port to .NET 5. When I recompile, I get these warnings because **GetLoggingDirectory()** uses the Windows registry, as you can see in **Figure 1**.

Let's take a closer look at this method. It first checks the registry to see whether a logging directory is configured. If there isn't, it falls back to the application's directory:

```
private static string GetLoggingDirectory()
{
    using (var key = Registry
        .CurrentUser
        .OpenSubKey(@"Software\Fabrikam"))
    {
        var path = "LoggingDirectoryPath";
        if (key?.GetValue(path)
            is string configuredPath)
        {
            return configuredPath;
        }
    }

    var exePath = Process.GetCurrentProcess()
                         .MainModule.FileName;
    var folder = Path.GetDirectoryName(exePath);
    return Path.Combine(folder, "Logging");
    return "Logging";
}
```

To make this code ready for cross-platform use, you have several options:

- **Remove the Windows-only portions.** This isn't usually desirable because when you port existing code, you generally want to support your existing customers without losing features.
- **Multi-target the project** to build for both .NET Framework and .NET 5, and use conditional compilation with #if to only include the Windows-specific parts when building for .NET Framework. This makes sense when the Windows-only portion depends on large components (such as Windows Forms or WPF) but it means you're producing multiple binaries. This works best for cases where you're building NuGet packages because NuGet will ensure that consumers get the correct binary without them having to manually pick the correct one.
- **Guard the calls to Windows-only APIs** with an operating system check. This option is the least intrusive for all consumers and works best when the OS-specific component is small.

In the case above, the best option is to guard the call with an OS check. To make these super easy, we've added new methods on the existing **System.OperatingSystem** class. You only need to surround the code that uses the registry with **OperatingSystem.IsWindows()**:

```
private static string GetLoggingDirectory()
{
    // Only check registry on Windows
    if (OperatingSystem.IsWindows())
    {
        using (var key = Registry
            .CurrentUser
            .OpenSubKey(@"Software\Fabrikam"))
        {
```

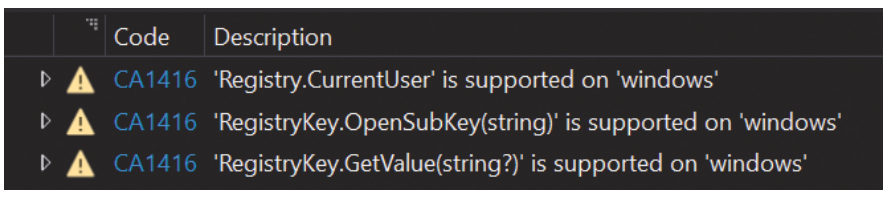


Figure 1: Warnings for using the Windows Registry

```
            var path = "LoggingDirectoryPath";
            if (key?.GetValue(path)
                is string configuredPath)
            {
                return configuredPath;
            }
        }
    }

    var exePath = Process.GetCurrentProcess()
                         .MainModule.FileName;
    var folder = Path.GetDirectoryName(exePath);
    return Path.Combine(folder, "Logging");
    return "Logging";
}
```

As soon as you do that, the warnings automatically disappear because the analyzer is smart enough to understand that these calls are only reachable when running on Windows.

Alternatively, you could have marked **GetLoggingDirectory()** as being Windows-specific:

```
[SupportedOSPlatform("windows")]
private static string GetLoggingDirectory()
{
    // ...
}
```

The analyzer will also no longer flag the use of the registry inside of **GetLoggingDirectory()** because it understands that it only gets called when running on Windows. However, it will now flag all callers of this method instead. This allows you to build your own platform-specific APIs and simply forward the requirement to your callers, as shown in **Listing 1**.

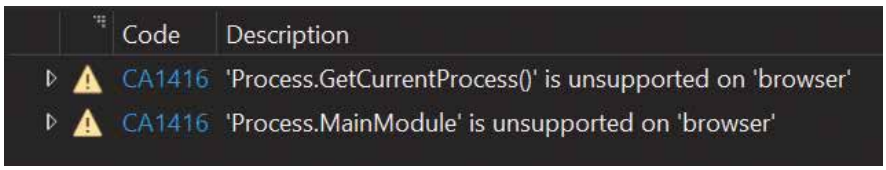


Figure 2: Warnings for using the Process APIs

Listing 1: Platform Guards

```
namespace System
{
    public sealed class OperatingSystem
    {
        public static bool IsOSPlatform(
            string platform);
        public static bool IsOSPlatformVersionAtLeast(
            string platform,
            int major, int minor = 0, int build = 0,
            int revision = 0);

        public static bool IsWindows();
        public static bool IsWindowsVersionAtLeast(
            int major, int minor = 0, int build = 0,
            int revision = 0);

        // Analogous APIs exist for Android, Browser,
        // FreeBSD, iOS, Linux, macOS, tvOS, and watchOS
    }
}
```

Dealing with Unsupported APIs

The previous case was an example of an API that only works on a specific set of operating systems. You can also mark APIs as unsupported for specific operating systems. This is useful for features that are generally cross-platform but can't be supported on some operating system due to some constraint. An example of this is Blazor WebAssembly. Because WebAssembly runs inside the browser's sandbox, you generally can't interact with the operating system or other processes. This means that some of the otherwise cross-platform APIs will throw a **PlatformNotSupportedException** when you try to call them from Blazor WebAssembly.

For instance, let's say I paste the **GetLoggingDirectory()** method into my Blazor app. Of course, the registry won't work, so let's delete that. This leaves us with just this:

```
private static string GetLoggingDirectory()
{
    var exePath = Process.GetCurrentProcess()
                         .MainModule.FileName;
    var folder = Path.GetDirectoryName(exePath);
    return Path.Combine(folder, "Logging");
}
```

Inside of a Blazor app this code generates two new warnings, seen in **Figure 2**.

This makes sense, given that you can't enumerate processes when running in the browser sandbox.

You may wonder why these methods weren't flagged earlier. The logging library targets net5.0, which can be consumed from a Blazor WebAssembly app as well. Shouldn't this warn you when you use APIs that won't work there?

Yes and no. On the one hand, net5.0 is indeed for code that's meant to run everywhere. But on the other hand, very few libraries need to run inside a browser sandbox and there are quite a few very widely used APIs, that can't be used there. If we flagged APIs that are unsupported by Blazor WebAssembly by default, a lot of developers would get warnings that never apply to their scenarios, and for them these warnings are just noise. However, when you build class libraries that are meant to be used by Blazor WebAssembly, you can enable this validation by adding a **<SupportedPlatform>** item to your project file:

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

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <SupportedPlatform Include="browser" />
  </ItemGroup>

</Project>
```

Blazor WebAssembly applications include this item by default. The hosted Blazor WebAssembly project template adds this line to the project that's shared between client and server as well.

Like the platform-specific APIs, you can also mark your own code as being unsupported by the browser sandbox by applying an attribute:

ID	Message	#APIs
SYSLIB0001	The UTF-7 encoding is insecure and should not be used. Consider using UTF-8 instead.	3
SYSLIB0002	PrincipalPermissionAttribute is not honored by the runtime and must not be used.	1
SYSLIB0003	Code Access Security is not supported or honored by the runtime.	144
SYSLIB0004	The Constrained Execution Region (CER) feature is not supported.	9
SYSLIB0005	The Global Assembly Cache is not supported.	2
SYSLIB0006	Thread.Abort is not supported and throws PlatformNotSupportedException.	2
SYSLIB0007	The default implementation of this cryptography algorithm is not supported	5
SYSLIB0008	The CreatePdbGenerator API is not supported and throws PlatformNotSupportedException.	1
SYSLIB0009	The AuthenticationManager Authenticate and PreAuthenticate methods are not supported and throw PlatformNotSupportedException.	2
SYSLIB0010	This Remoting API is not supported and throws PlatformNotSupportedException.	2
SYSLIB0011	BinaryFormatter serialization is obsolete and should not be used. See https://aka.ms/binaryformatter for more information.	6
SYSLIB0012	Assembly.CodeBase and Assembly.EscapedCodeBase are only included for .NET Framework compatibility. Use Assembly.Location instead.	3
SYSLIB0013	Uri.EscapeUriString can corrupt the URI string in some cases. Consider using Uri.EscapeDataString for query string components instead.	1
SYSLIB0014	Use HttpClient instead.	12

Table 1: List of obsoletions in .NET 5

```
[UnsupportedOSPlatform("browser")]
private static string GetLoggingDirectory()
{
    // ...
}
```

Better Obsoletions

A problem that's existed in the BCL for a long time is that it's not easy to obsolete APIs. One reason was that people compiled in production (for example, the ASP.NET websites compile on the Web server). When such a site compiles with warnings as errors, a Windows update can bring a new .NET Framework version with new obsoletions that might break the app.

The larger issue was that obsoletions can't be grouped; all obsoletions share the same diagnostic ID. This means you can either turn them all off or suppress every occurrence individually using **#pragma warning disable**. That means that obsoletions were usually only viable for methods; as soon as you obsolete a type (or worse, a set of types), you might quickly cause hundreds of warnings in your code base. At this point, most developers simply disable the warning for obsoleted APIs, which means that next time you obsolete an API, they won't notice anymore.

In .NET 5, we've addressed this via a simple trick: we added **DiagnosticId** property to **ObsoleteAttribute,** which the compilers use when reporting warnings. This allows you to give each obsoleted feature a separate ID. **Table 1** shows the list of features we've obsoleted in .NET 5. One of those features is Code Access Security (CAS). Although the attributes exist in .NET 5 to make porting easier, they aren't doing anything at runtime. This obsoletion applies to 144 APIs. If you happen to use CAS and port to .NET 5, you might get hundreds of warnings. To see the forest for the trees again, you might decide to suppress and ignore CAS-related obsoletion and file a bug to get rid of them later. You'd do this via the normal suppression mechanism, for example, by adding a **<NoWarn>** entry to your project:

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

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <!-- Ignore CAS obsoletions -->
    <NoWarn>SYSLIB0003</NoWarn>
  </PropertyGroup>

</Project>
```

The benefit of this is that other obsoletions for other features will still be raised. This allows you to declare bankruptcy in one area without accruing more debt in other areas.

In many cases, a single line of text is insufficient to describe why an API was obsoleted and what the replacement is. So another improvement we made is that we added a URL property to **ObsoleteAttribute**. All new obsoletions will have a documentation page with more details. You can access these URLs by clicking on the ID in the error list (**Figure 3**).

Analyzers

The platform compatibility analyzer is only one of about 250 analyzers that we included with the .NET 5 SDK (about 60 of them are on by default). These analyzers cover the use of the language (C#, VB) as well as the BCL APIs.

> There are about 250 analyzers in the .NET 5 SDK.

Moving forward, the idea is that as when we add new features to .NET, we're also adding corresponding analyzers and code fixers to help you use them correctly, right out of the gate.

Let's look at a simple example. I wrote this little Person class for one of my projects. Because equality is tricky, I made a mistake in the implementation of the **Equals** method. Can you spot it?

```
class Person : IEquatable<Person>
{
   public Person(string name, DateTime birthday)
   {
       Name = name;
       Birthday = birthday;
   }

   private string Name { get; }
   private DateTime Birthday { get; }

   public bool Equals(Person other)
   {
       return ReferenceEquals(Name,
                              other.Name) &&
              ReferenceEquals(Birthday,
                              other.Birthday);
   }
}
```

Fortunately, you don't have to. The new built-in analyzer has your back and flags the call to **ReferenceEquals** (**Figure 4**)

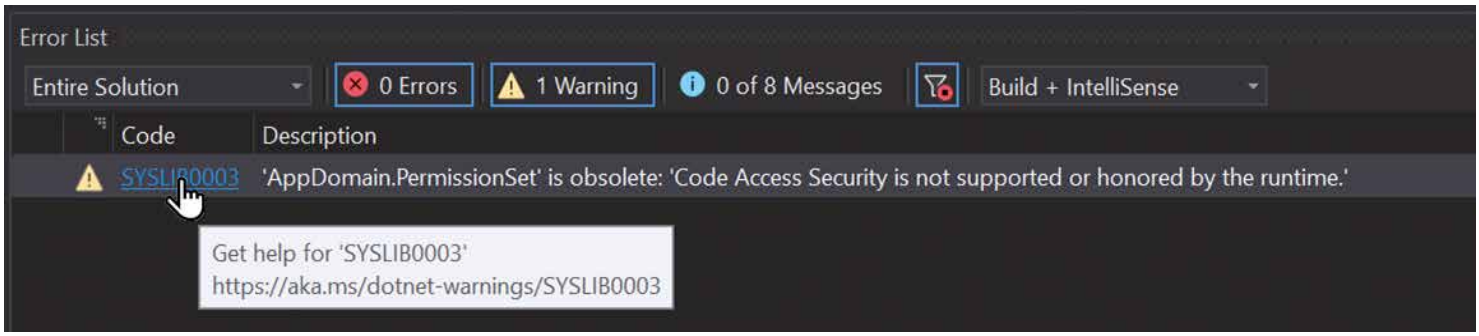


Figure 3: Using the URL of an obsoletion to find out more

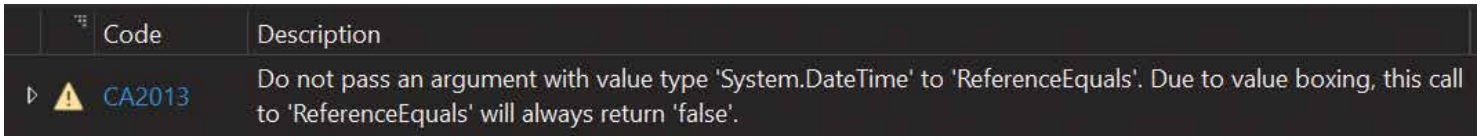


Figure 4: Warning when calling ReferenceEquals on value types

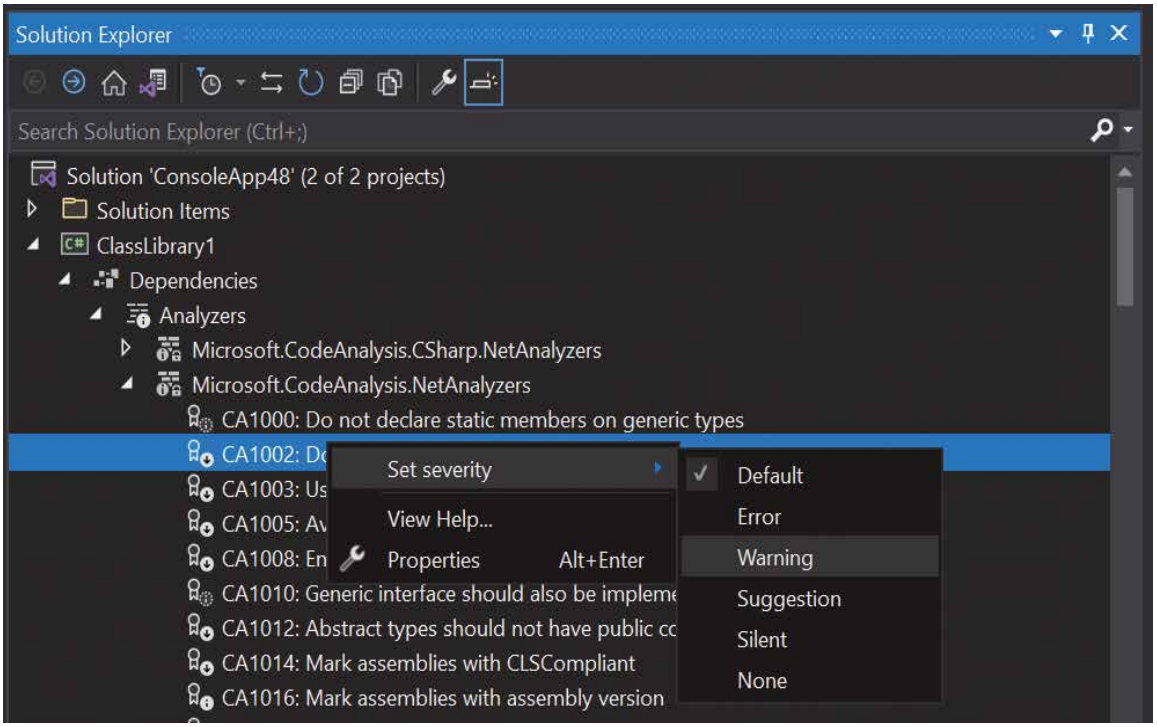


Figure 5: Customize rule severity via Solution Explorer



Figure 6: Warning when range-based indexers cause copies

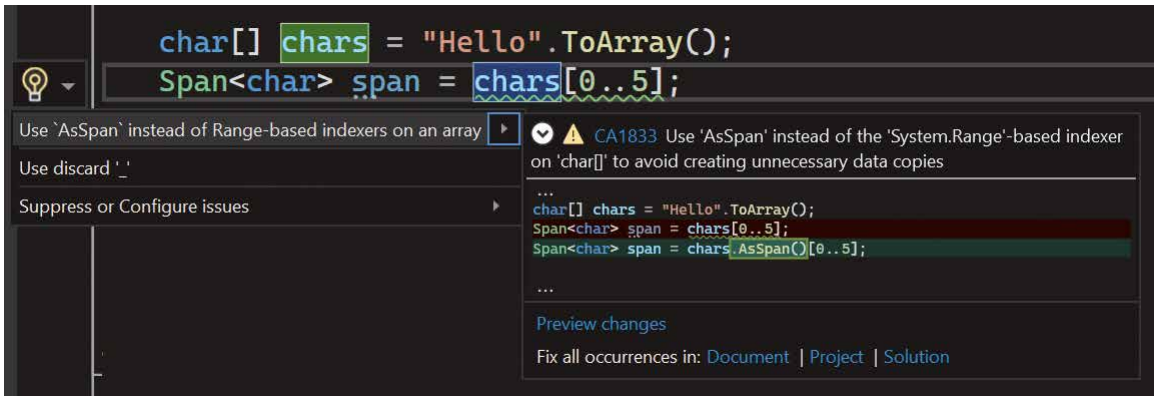


Figure 7: Invoking a code fixer

Not all analyzers are on by default because not every project has the same requirements. For example, not every project needs to be localized and not every project needs to operate in a Web service with high throughput demands.

For example, let's consider this code:

```
char[] chars = "Hello".ToArray();
Span<char> span = chars[0..2];
```

The second line uses the range-based indexer to create a new array that only has the two elements. Although there's no correctness issue with the code, it's not as efficient as it could be.

Let's turn on the analyzer by using the Solution Explorer (**Figure 5**). Open the **Dependencies** node, drill into **Analyzers** and under **Microsoft.CodeAnalysis.NetAnalyzers**, right-click on **CA1833** and select **Warning**.

This creates an **.editorconfig** with an entry for **CA1833**. Alternatively, you could also add a line to enable all rules in the performance category:

```
[*.cs]

# Setting severity of a specific rule:
dotnet_diagnostic.CA1833.severity = warning

# Bulk enable all performance rules:
dotnet_analyzer_diagnostic.category-
performance.severity = warning
```

After the rule is enabled, we get a warning for using the range-based indexer on the array (**Figure 6**).

You can invoke the lightbulb menu via **Ctrl+.**, which offers a fix for this issue, as seen in **Figure 7**.

The fixer changes the code to slice the array as a span, which doesn't need to copy the underlying array:

```
char[] chars = "Hello".ToArray();
Span<char> span = chars.AsSpan(0..5);
```

Sweet!

Closing

With .NET 5, we have heavily improved our support for static code analysis. This includes an analyzer for platform-specific code and a better mechanism to deal with obsoletions. The .NET 5 SDK includes over 230 analyzers!

.NET 5 is the successor of .NET Core and .NET Standard. As a result, the **net5.0** name unifies and replaces the **netcoreapp** and **netstandard** framework names. If you still need to target .NET Framework, you should continue to use **netstandard2.0**. Starting with .NET 5, we'll provide a unified implementation of .NET that can support all workloads, include console apps, Windows desktop apps, websites, and cloud services. And, with .NET 6, this will also include the iOS and Android platforms.

Immo Landwerth

Introducing C# 9.0

C# 9.0 adds many new features, and focuses on a few themes. C# 9.0 is part of .NET 5, which continues the journey toward a single .NET ecosystem. The new features focus on modern workloads, that is, the software applications and services you're building today. The C# 9.0 compiler ships with the .NET 5.0 SDK. Many of the C# 9.0 features rely on new features in the .NET 5.0



Bill Wagner
wiwagn@microsoft.com
twitter.com/billwagner

Bill Wagner is responsible for the C# area of https://docs.microsoft.com. He creates learning materials for developers interested in the C# language and .NET. He's a member of the ECMA C# Standards Committee and the .NET Foundation board of directors. He is President of the Humanitarian Toolbox. Before joining Microsoft, he was awarded Microsoft Regional Director and .NET MVP for 10+years. He is the author of *Effective C#* and *More Effective C#*.

libraries and updates to the .NET CLR that're part of .NET 5.0. Therefore, C# 9.0 is supported only on .NET 5.0. C# 9.0 focuses on features that support native cloud applications, modern software engineering practices, and more concise readable code. There are several new features that make up this release:

- Top-level statements
- Record types
- Init-only setters
- Enhancements to pattern matching
- Natural-sized integers
- Function pointers
- Omit localsinit
- Target type new
- Target type conditional
- Static anonymous methods
- Covariant return types
- Lambda discard parameters
- Attributes on local functions

This article explores those features and provides scenarios where you might use them.

Top-Level Statements

Let's start the exploration of C# 9.0 with top-level statements. This feature removes unnecessary ceremony from many applications. Consider the canonical "Hello World!" program:

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

There's only one line of code that does anything. With top-level statements, you can replace all that boilerplate with the using statement and the single line that does the work:

```
using System;

Console.WriteLine("Hello World!");
```

Top-level statements provide a way for you write programs with less ceremony. Only one file in your application may use top-level statements. If the compiler finds top-level statements in multiple source files, it's an error. It's also an error if you combine top-level statements with a declared program entry point method, typically a Main method. In a sense, you can think that one file contains the statements that would normally be in the Main method of a Program class.

One of the most common uses for this feature is creating teaching materials. Beginner C# developers can write the canonical "Hello World!" in one line of code. None of the extra ceremony is needed. Seasoned developers will find many uses for this feature, as well. Top-level statements enable a script-like experience for experimentation similar to what Jupyter Notebooks provides. Top-level statements are great for small console programs and utilities. In addition, Azure functions are an ideal use case for top-level statements.

Most importantly, top-level statements don't limit your application's scope or complexity. Those statements can access or use any .NET class. They also don't limit your use of command line arguments or return values. Top-level statements can access an array or strings named args. If the top-level statements return an integer value, that value becomes the integer return code from a synthesized Main method. The top-level statements may contain async expressions. In that case, the synthesized entry point returns a **Task**, or **Task<int>**. For example, the canonical Hello World example could be expanded to take an optional command line argument for a person's name. If the argument is present, the program prints the name. If not, it prints "Hello World!" like this:

```
using System;
using System.Linq;

if (args.Any())
{
  var msg =
    args.Aggregate((s1, s2) => $"{s1} {s2}");
    Console.WriteLine($"Hello {msg}");
}
else
    Console.WriteLine("Hello World!");
```

Record Types

Record types have a major impact on the code you write every day. There are a lot of smaller language enhancements that make up records. These enhancements are easier to understand by starting with the typical uses for records.

.NET types are largely classified as classes or anonymous types that are reference types and structs or tuples, which are value types. Although creating immutable value types is recommended, mutable value types don't often introduce errors. That's because value types are passed by value to methods, so any changes are made to a copy of the original data.

There are a lot of advantages to immutable reference types, as well. These advantages are more pronounced in concur-

rent programs with shared data. Unfortunately, C# forced you to write quite a bit of extra code to create immutable reference types. For that reason, busy developers, which is all developers, write mutable reference types. Records provide a type declaration for an immutable reference type that uses value semantics for equality. The synthesized methods for equality and hash codes considers two records to be equal if their properties are all equal. Consider this definition:

```
public record Person
{
    public string LastName { get; }
    public string FirstName { get; }

    public Person(string first, string last) =>
        (FirstName, LastName) = (first, last);
}
```

That record definition creates a Person type that contains two read-only properties: FirstName and LastName. The Person type is a reference type. If you looked at the IL, it's a class. It's immutable in that none of the properties can be modified once it's been created. When you define a record type, the compiler synthesizes several other methods for you:

- Methods for value-based equality comparisons
- Override for GetHashCode
- Copy and Clone members
- PrintMembers and ToString
- Deconstruct method

Records support inheritance. You can declare a new record derived from Person as follows:

```
public record Teacher : Person
{
    public string Subject { get; }

    public Teacher(string first, string last,
        string sub)
        : base(first, last)
        => Subject = sub;
}
```

You can also seal records to prevent further derivation:

```
public sealed record Student : Person
{
    public int Level { get; }

    public Student(string first, string last,
        int level)
        : base(first, last)
        => Level = level;
}
```

The compiler generates different versions of the methods mentioned above depending on whether or not the record type is sealed and whether or not the direct base class is an object. The compiler does this to ensure that equality for records means that the record types match and the values of each property of the records are equal. Consider the small hierarchy above. The compiler generates methods so that a Person could not be considered equal to a Student or a Teacher. In addition to the familiar Equals overloads, operator == and operator !=, the compiler generates a new EqualityContract property. The property returns a Type object that matches the type of the record. If the base type is object, the property is virtual. If the base type is another record type, the property is an override. If the record type is sealed, the property is sealed. The synthesized GetHashCode uses the GetHashCode from all the public properties declared in the base type and the record type. These synthesized methods enforce value-based equality throughout an inheritance hierarchy. That means that a Student will never be considered equal to a Person with the same name. The types of the two records must match, as well as all properties shared among the record types being equal.

Records also have a synthesized constructor and a "clone" method for creating copies. The synthesized constructor has one argument of the record type. It produces a new record with the same values for all properties of the record. This constructor is private if the record is sealed; otherwise it's protected. The synthesized "clone" method supports copy construction for record hierarchies. The term "clone" is in quotes because the actual name is compiler-generated. You can't create a method named "Clone" in a record type.

The synthesized clone method returns the type of record being copied using virtual dispatch. If a record type is abstract, the clone method is also abstract. If a record type is

ADVERTISERS INDEX

Advertisers Index

CODE Consulting	www.codemag.com/consulting	5
CODE Legacy	www.codemag.com/legacy	2
CODE Magazine	www.codemag.com/magazine	19
Microsoft	azure.microsoft.com/free/dotnet	2
Microsoft	www.azure.com/AMP	3
Microsoft	www.dot.net	38
Microsoft	live.dot.net	75
Microsoft	visualstudio.com/download	76

Advertising Sales:
Tammy Ferguson
832-717-4445 ext 026
tammy@codemag.com

This listing is provided as a courtesy to our readers and advertisers. The publisher assumes no responsibility for errors or omissions.

sealed, the clone method is sealed. If the base type of the record is object, the clone method is virtual. Otherwise, it's override. The result of all these rules is that you can create copies of a record ensuring that the copy is the same type. Furthermore, you can check the equality of any two records in an inheritance hierarchy and get the results that you intuitively expect.

```
Person p1 = new Person("Bill", "Wagner");
Student s1 = new Student("Bill", "Wagner", 11);

Console.WriteLine(s1 == p1); // false
```

The compiler synthesizes two methods that support printed output: a ToString override and PrintMembers. The PrintMembers method returns a comma-separated list of property names and values. The ToString() override returns the string produced by PrintMembers, surrounded by { and }. For example, the ToString method for Student generates a string like the following:

```
Student { LastName = Wagner, FirstName = Bill,
Level =  11 }
```

The examples shown so far use traditional syntax to declare properties. There's a more concise form called **positional records**. Here are the three record types defined earlier using positional record syntax:

```
public record Person(string FirstName,
    string LastName);

public record Teacher(string FirstName,
    string LastName,
    string Subject)
    : Person(FirstName, LastName);

public sealed record Student(string FirstName,
    string LastName, string Subject)
    : Person(FirstName, LastName);
```

These declarations create the same functionality as the earlier version (with a couple extra features that I'll cover in a bit). These declarations end with a semicolon instead of brackets because these records don't add additional methods. You can add a body and include any additional methods as well:

```
public record Pet(string Name)
{
  public void ShredTheFurniture() =>
    Console.WriteLine("Shredding furniture");
}

public record Dog(string Name) : Pet(Name)
{
  public void WagTail() =>
    Console.WriteLine("It's tail wagging time");

  public override string ToString()
  {
    StringBuilder s = new();
    base.PrintMembers(s);
    return $"{s.ToString()} is a dog";
  }
}
```

The compiler produces a Deconstruct method for positional records. The deconstruct method has parameters that match the names of all public properties in the record type. The Deconstruct method can be used to deconstruct the record into its component properties.

```
// Deconstruct is in the order
// declared in the record.
(string first, string last) = p1;
Console.WriteLine(first);
Console.WriteLine(last);
```

Finally, records support with-expressions. A **with** expression instructs the compiler to create a something like a copy of a record, but with specified properties modified.

```
Person brother = p1 with { FirstName = "Paul" };
```

The above line creates a new Person record where the LastName property is a copy of p1, and the first Name is "Paul". You can set any number of properties in a **with** expression.

Any of the synthesized members except the "clone" method may be written by you. If a record type has a method that matches the signature of any synthesized method, the compiler doesn't generate that method. The earlier Dog record example contains a hand-coded ToString method as an example.

Init-Only Setters

Init only setters provide consistent syntax to initialize members of an object. Property initializers make it clear which value is setting which property. The downside is that those properties must be settable. Starting with C# 9.0, you can create **init** accessors, instead of **set** accessors for properties and indexers. The main benefit is that callers can use property initializer syntax to set these values in creation expressions, but those properties are read-only once construction has completed. Init-only setters provide a window to change state. That window closes when the construction phase ends. The construction phase effectively ends after all initialization, including property initializers and **with** expressions, has completed. Consider this immutable Point structure:

```
public struct Point
{
    public double X { get; init; }
    public double Y { get; init; }

    public double Distance => Math.Sqrt(X * X +
        Y * Y);
}
```

You can initialize this structure using property initializer syntax, but you can't modify the values once construction and initialization has completed:

```
var pt = new Point { X = 3, Y = 4 };
pt.X = 7; // Error!
Console.WriteLine(pt.Distance);
```

In addition to using property initializers, init-only setters can be very useful to set base class properties from derived classes, or set derived properties through helpers in a base

class. Positional records declare properties using init-only setters. Those setters are used in **with** expressions. You can declare init-only setters for any class or struct you define.

Enhanced Pattern Matching

C# 9 includes new pattern-matching improvements:

- Type patterns match a variable as a type
- Parenthesized patterns enforce or emphasize the precedence of pattern combinations
- Conjunctive **and** patterns require both patterns to match
- Disjunctive **or** patterns require either pattern to match
- Negated **not** patterns require that a pattern doesn't match
- Relational patterns require that the input be less than, greater than, less than or equal, or greater than or equal to a given constant

These patterns enrich the syntax for patterns. Consider these examples:

```
public static bool IsLetter(this char c) =>
    c is >= 'a' and <= 'z'
    or >= 'A' and <= 'Z';
```

Alternatively, with optional parentheses to make it clear that **and** has higher precedence than **or**:

```
public static bool IsLetterIsSeparator(this char c) =>
    c is (>= 'a' and <= 'z')
    or (>= 'A' and <= 'Z')
    or '.'
    or ',';
```

One of the most common uses is a new clear syntax for a null check:

```
if (e is not null)
{
    // ...
}
```

Any of these patterns can be used in any context where patterns are allowed: **is** pattern expressions, **switch** expressions, nested patterns, and the pattern of a switch statement's case label.

Performance and Interop

Three new features improve support for native interop and low-level libraries that require high performance: native sized integers, function pointers, and omitting the localsinit flag.

Native sized integers nint and nuint are integer types. They are expressed by the underlying types System.IntPtr and System.UIntPtr. The compiler surfaces additional conversions and operations for these types as native ints. Native sized ints don't have constants for MaxValue or MinValue, except for nuint.MinValue, which has a MinValue of 0. Other values cannot be expressed as constants because it would depend on the native size of an integer on the target computer. You can use constant values for nint in the range **[int.MinValue .. int.MaxValue]**. You can use constant values for nuint in the range **[uint.MinValue .. uint.MaxValue]**. The compiler performs constant folding for all unary and binary operators using the Int32 and UInt32 types. If the result doesn't fit in 32-bits, the operation is executed at runtime and isn't considered a constant. Native-sized integers can increase performance in scenarios where integer math is used extensively and needs to have the faster performance possible.

Function pointers provide an easy syntax to access the IL opcodes ldftn and calli. You can declare function pointers using new delegate* syntax. A delegate* type is a pointer type. Invoking the delegate* type uses calli, in contrast to a delegate that uses callvirt on the Invoke method. Syntactically, the invocations are identical. Function pointer invocation uses the **managed** calling convention. You add the unmanaged keyword after the delegate* syntax to declare that you want the unmanaged calling convention. Other calling conventions can be specified using attributes on the delegate* declaration.

Finally, you can add the SkipLocalsInitAttribute to instruct the compiler not to emit the localsinit flag. This flag instructs the CLR to zero-initialize all local variables. This has been the default behavior for C# since 1.0. However, the extra zero-initialization may have measurable performance impact in some scenarios, in particular, when you use stackalloc. In those cases, you can add the SkipLocalsInitAttribute. You may add it to a single method or property, or to a class, struct, interface, or even a module. This attribute does not affect abstract methods; it affects the code generated for the implementation.

These features can improve performance in some scenarios. They should be used only after careful benchmarking both before and after adoption. Code involving native sized integers must be tested on multiple target platforms with different integer sizes. The other features require unsafe code.

Fit and Finish Features

Many of the other features help you write code more efficiently. In C# 9.0, you can omit the type in a new expression when the created object's type is already known. The most common use is in field declarations:

```
public class PropertyBag
{
    private Dictionary<string, object>
        properties = new();

    // elided
}
```

Target type new can also be used when you need to create a new object to pass as a parameter to a method. Consider a ParseJson() method with the following signature:

```
public JsonElement ParseJson(string text,
    JsonSerializerOptions opts)
```

You could call it as follows:

```
var result = ParseJson(text, new());
```

A similar feature improves the target type resolution of conditional expressions. With this change, the two expressions need not have an implicit conversion from one to the other but may both have implicit conversions to a common type. You likely won't notice this change. What you will notice is that some conditional expressions that previously required casts or wouldn't compile now just work.

Starting in C# 9.0, you can add the "static" modifier to lambda expressions or anonymous methods. That has the same effect as the "static" modifier on local functions: a static lambda or anonymous function can't capture local variables or instance state. This prevents accidentally capturing other variables.

Covariant return types provide flexibility for the return types of virtual functions. Previously, overrides had to return the same type as the base function. Now, overrides may return a type derived from the return type of the base function. This can be useful for Records and for other types that support virtual clone or factory methods.

Next, you can use discards as parameters to lambda expressions. This convenience enables you to avoid naming the argument, and the compiler may be able to avoid using it. You use the "_" for any argument.

Finally, you can now apply attributes to local functions. These are particularly useful to add nullable attribute annotations to local functions.

Support for Source Generators

Two final features support C# source generators. C# source generators are a component you can write that's similar to a Roslyn analyzer or code fix. The difference is that source generators analyze code and write new source code files as part of the compilation process. A typical source generator searches code for attributes or other conventions. Based on the information supplied by the attributes, the source generator writes new code that's added to the library or application.

You can read attributes or other code elements using the Roslyn analysis APIs. From that information, you can add new code to the compilation. Source generators can only add code; they're not allowed to modify any existing code in the compilation.

The two features added for source generators are extensions to partial method syntax and module initializers. First, there are fewer restrictions to partial methods. Before C# 9.0, partial methods were private but can't specify an access modifier, have a void return, or have out parameters. These restrictions meant that if no method implementation was provided, the compiler removed all calls to the partial method. C# 9.0 removes these restrictions but requires that partial method declarations have an implementation. Source generators can provide that implementation. To avoid introducing a breaking change, the compiler considers any partial method without an access modifier to follow the old rules. If the partial method includes the private access modifier, the new rules govern that partial method.

The second new feature for source generators is module initializers. These are methods that have the System.Runtime.CompilerServices.ModuleInitializer attribute attached to them. These methods are called by the runtime when the assembly loads. A module initializer method:

- Must be static
- Must be parameterless
- Must return void
- Must not be a generic method
- Must not be contained in a generic class
- Must be accessible from the containing module

That last bullet point effectively means the method and its containing class must be internal or public. The method cannot be a local function. Source generators may need to generate initialization code. Module initializers provide a standard place for that code to reside.

Summary

C# 9.0 continues the evolution of C# as a modern language. It's embracing new idioms and new programming paradigms while retaining its roots as an object-oriented component-based language. The new features make it efficient to build the modern programs that we're creating today. Try to adopt the new features today.

Bill Wagner

EF Core 5: Building on the Foundation

Hot on the tail of EF Core 3 is EF Core 5. Remember when we skipped from EF v1 to EF4 to align with .NET Framework 4.0? Like that time, this new gap in version numbers was designed to align with the successor to .NET Core 3. Except that successor is no longer called .NET Core, it's .NET 5. Yet EF Core 5 is keeping the "Core" (as is ASP.NET Core 5). This allows us to continue differentiating from the pre-Core versions of EF and ASP.NET. So, EF Core 5 it is. Happily for me, this article is about EF Core 5 and I'm not charged with further explaining the name and other changes to .NET 5, details you can read about in another article in this issue.

When EF Core 5 was still a few months from release, Arthur Vickers (from the EF team) tweeted some impressive stats about the 637 GitHub issues already closed for the EF Core 5 release:

- 299 bugs fixed
- 113 docs and cleanup
- 225 features and minor enhancements

Building on the EF Core 3 Foundation

You may have read my article introducing CODE Magazine readers to EF Core 3, entitled "Entity Framework Core 3.0, A Foundation for the Future" (https://codemag.com/Article/1911062/Entity-Framework-Core-3.0-A-Foundation-for-the-Future). Because the first iteration of EF Core was a complete rewrite of Entity Framework, it had been a very "v1" version. EF Core 2 goals were around tightening down the existing code base and adding critical features, making EF Core truly production ready. With that stable version in place (and widely used), the team felt that with EF Core 3, it was safe to make the kind of low-level changes that would result in breaking changes. There weren't many new features in EF Core 3, so teams could continue to use EF Core 2 if they wanted while EF Core 3 prepared the framework for a long future.

With EF Core 5.0, we're now witnessing that future. I want to assure you that the great number of breaking changes in EF Core 3.0 were really a one-time event. Some of you may be worried that this will happen again, but the EF Core team is now focused on building on that foundation and minimizing future breaking changes. EF Core 5 doesn't have very many breaking changes, and they're clearly laid out in the Breaking Changes page in the docs. Today (as EF Core 5 is in Preview 8) there is only one listed and it's very much an edge case related to SQLite and Geometric Dimensions...something I've never even heard of. (https://docs.microsoft.com/en-us/ef/core/what-is-new/ef-core-5.0/breaking-changes)

Unlike EF Core 3, which had only seven new features, EF Core 5 has many new features along with improvements to existing features and the usual bug fixes. There's no way to cover all of the new goodies in this article, so be sure to look at the docs to get a great overview. At the time of writing this article, there have been eight previews and you'll find a list of what's new for each of the different previews at https://docs.microsoft.com/en-us/ef/core/what-is-new/ef-core-5.0/whatsnew. I'll pick some that I find interesting and/or important and drill into them in this article.

More importantly, much of the groundwork laid in EF Core 3 has enabled some great additions to EF Core 5.0, including the return of features from EF6. I think the most notable of these is support for many-to-many relationships that doesn't require us to tangle with classes that represent the join—the solution we've been stuck with since EF Core first arrived.

And the reason many-to-many took so long to appear is a nice example of that EF Core 3 groundwork. The original many-to-many support was tied to the way Entity Framework was designed back in 2006. Although it was magical indeed in the way it inferred and persisted the join data, the source of that magic put a lot of limitations on what we developers could do with the data in that relationship. When building EF Core v1, the team didn't want to duplicate that limiting solution. However, re-thinking and re-devising many-to-many was going to be a lot of work that would have held up the release. In the meantime, we could at least leverage the workaround of using the explicit join entity. But now, with those necessary changes in place, the EF team was able to build a smarter, more flexible, and non-magical many-to-many feature in EF Core 5.

Platform Support for EF Core 5

EF Core 5 can be used with any platform that supports .NET Standard 2.1. This includes .NET Core 3.1 and .NET 5. But it won't run on .NET Standard 2.0, which means that starting with EF Core 5.0, you can no longer use EF Core with .NET Framework. You can still use .NET Framework with EF Core 1, 2, and 3.

Backward Compatible with EF Core 3

With the exception of the few breaking changes, you should find EF Core 5 to be backward compatible with your EF Core 3 code. I tested this out with various solutions of my own (e.g., the demos from the EF Core 3 article and some from my Pluralsight course, Getting Started with EF Core 3.1) and everything continued to run, with all tests passing. The icing on that compatibility cake however, is the opportunity to improve that code with new EF5 features.

Documentation and Community

There are a few other important themes to be aware of with EF Core 5. The team is focused on major improvements to the documentation. You might have noticed that earlier this year, the docs were reorganized. The home page for EF docs (**Figure 1**) at docs.microsoft.com/ef, has guided paths to more easily find the type of information you're seeking.

In addition to the already well documented features, they've been adding guidance and best practices. For example, there are now Platform Experience documents that provide guidance for using EF Core on different platforms.



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.

Some examples are Getting Started with WPF and EF Core (https://docs.microsoft.com/en-us/ef/core/get-started/wpf), Getting Started with Xamarin and EF Core (https://docs.microsoft.com/en-us/ef/core/get-started/xamarin) and guidance for Blazor and EF Core (https://docs.microsoft.com/en-us/aspnet/core/blazor/blazor-server-ef-core).

The EF team has also begun to have bi-weekly live community standups online. In a time when we're not able to travel and convene, it's wonderful to be able to not only visit with the team members (and some of their cats) but also to get such great insight into the work they and other members of the community are doing. These happen on YouTube with past and upcoming standups listed on this playlist: http://bit.ly/EFCoreStandups.

Another exciting aspect of EF Core 5 is the number of pull requests (PRs) that have come from the community and are now a part of EF Core. Although some of the PRs are very close to ready-for-primetime when submitted to the GitHub repository, the team has also worked closely with developers to help them spruce up PRs that weren't quite ready to be merged. What I truly love about this is how much attention and recognition the EF Core team has paid to these contributors. The weekly status updates that the team shares on GitHub (https://github.com/dotnet/efcore/issues/19549) include a list of the pull requests. For example, you can see PRs from six developers in the community listed in the August 6, 2020 update (https://github.com/dotnet/efcore/issues/19549#issuecomment-670225346). Even cooler is to see the long list of contributors including their photos and links to their GitHub accounts at the bottom of the announcement post for the first preview of EF Core 5 (https://devblogs.microsoft.com/dotnet/announcing-entity-framework-core-5-0-preview-1/#thank-you-to-our-contributors).

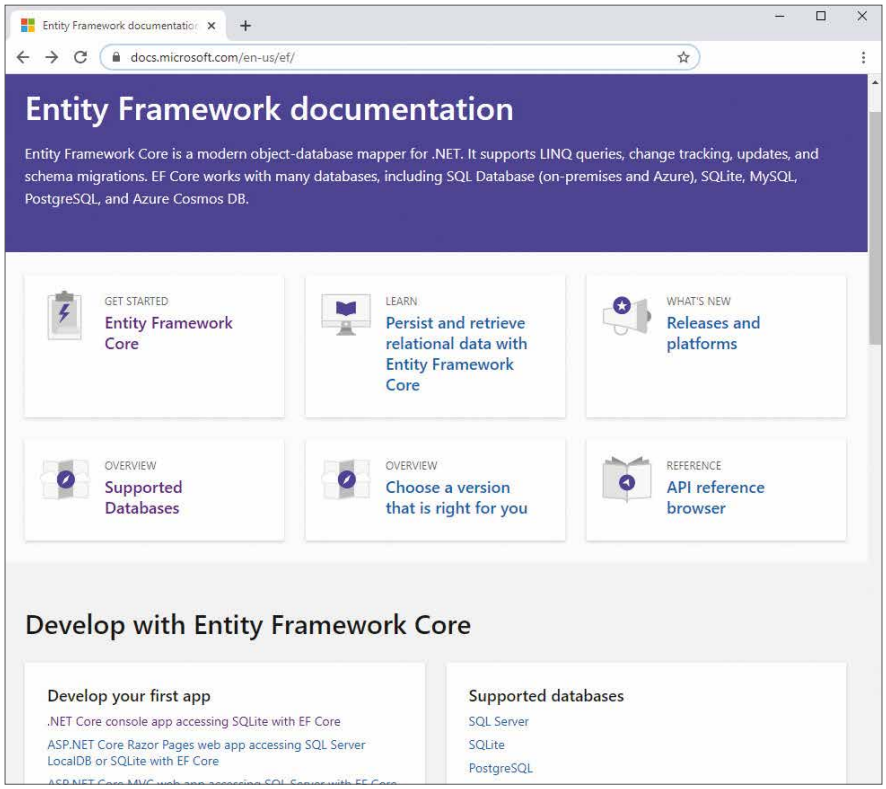


Figure 1: The redesigned EF Docs Home

The team is also excited about the community involvement. Speaking with Arthur Vickers, I could hear the joy in his voice as we talked about it. He tells me:

New and Improved Many-to-Many Support

As I mentioned earlier, and you may be quite familiar with, proper many-to-many support has existed since the outset of Entity Framework. What I mean by "proper" is where you aren't required to have a join entity in your code and EF infers the connection between the two ends. And as I explained above, it wasn't implemented in EF Core, EF Core 2, or EF Core 3, which left us tangling with a join entity. This was the most requested feature, as per the "thumbs up" on the relevant GitHub issue (https://github.com/dotnet/efcore/issues/1368).

EF Core 5 now has true many-to-many support so that you can write your classes in a natural way and EF Core will understand the mapping when it comes to creating and inferring the database schema, building SQL for queries and updates, and materializing results.

This is, to me, one of the more fascinating features, so I'll dig into this more deeply than any other feature in this article.

Here's an example of the natural way, in a relationship where people could reside or otherwise be tied to a variety of addresses, and an address might have a number of residents. Each Person has a list of Addresses and Address has a list of person types, called Residents.

```
public class Person
{
  public int Id { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public List<Address> Addresses { get; set; }
}

public class Address
{
  public int Id { get; set; }
  public string Street { get; set; }
  public string PostalCode { get; set; }
  public List<Person> Residents { get; set; }
}
```

The minimal requirement for EF Core 5 to recognize the relationship is that you must make the DbContext aware of one of the ends (e.g., create a DbSet for either entity). EF Core 5 can then infer the relationship in the model. For a simple, conventional many-to-many, there's no more configuration or effort needed. I've chosen to create DbSets for both types.

```
public DbSet<Person> People { get; set; }
public DbSet<Address> Addresses { get; set;}
```

When I add a migration to create the database, EF Core 5 infers a join table between People and Addresses in the database. The migration has CreateTable methods for Addresses, People, and AddressPerson. In the database created based on that migration (**Figure 2**), you can see that the join table has two columns, AddressesId and ResidentsId. EF Core

picked up the name of the Residents property when choosing the join table schema. The primary key is composed from both columns and each column is also a foreign key back to one of the other tables.

This mapping is the same as what we got by convention in earlier versions of Entity Framework based on the same set up.

A serious limitation about many-to-many support in EF6, however, is that there is no way to have additional columns in that join table. For example, the date on which a person took up residency at an address. In that case, you would have to create an explicit entity to map to the join table and manually manage every detail about the relationship in your code—just as we've had to do in EF Core. If you already had production code with the simpler many-to-many relationship, this also meant breaking a lot of code. This problem was a result of the house of cards (my words) on which many-to-many was originally built. The details are fairly interesting (to some of us) but I won't be delving into those internals in this article.

There are, however, two pieces of the EF Core 5 many-to-many support that you should know about. One is called **skip navigations,** which allows you to skip over the inferred join entity between Address and Person when writing your code. The other is **property-bag entity types**. These are new to EF Core 5 and are a type of dictionary. Property bag entities are used to solve a number of mapping quandaries, not just many-to-many. But for many-to-many, they allow you to store additional data into the join table while still benefiting from the skip navigation. The ability to have extra data in the join table is a very big difference from EF6 and, I hope you will agree, is one of the reasons that many-to-many support in EF Core was worth the wait.

When building up objects or querying, you don't have to make any reference to the join—just work with the direct navigation properties and EF Core will figure out how to persist and query. For example, here, I'm creating a new person and adding a few addresses:

```
var person = new Person
  {FirstName="Jeremy", LastName="Likness" };
person.Addresses.Add(new Address
    {Street="999 Main" });
person.Addresses.Add(new Address
    {Street="1000 Main" });
```

Then, I add the person to a context instance and save to the database:

```
context.People.Add(person);
context.SaveChanges();
```

EF Core first inserts the person and the addresses, then grabs the database-generated IDs from those new rows. Then it uses those values to insert the two relevant rows to the PersonAddress join table. By the way, I used the new streamlined logging capability in EF Core 5 to see the SQL. I'll show you how that works after I finish digging into many-to-many.

Also notable is how EF Core batches commands when performing these inserts. I found that if you reach the batch command threshold (at least four commands for SQL Server) when inserting or updating the ends (person and address), the person and address inserts are batched together. The join table inserts will be batched in their own command if there are at least four of those.

When querying from either end, there's no need to acknowledge the join thanks to skip navigations. You can eager load Addresses for People and eager load Residents (i.e., Person types) for Addresses, and EF Core will work out the correct queries and materialize the results.

```
context.People.Include(p=>p.Addresses)
```

Projection queries and explicit loading also take advantage of the many-to-many capability.

In the end, the bottom line for simple scenarios is that many-to-many works just like it did in EF6.

> The bottom line for simple scenarios is that many-to-many works just like it did in EF6.

Many-To-Many with Additional Columns

As mentioned earlier, the new property-bag entity type enables EF Core 5 to store additional data in the join table. And it avoids the detrimental side effect of trying to achieve that in EF6. In EF6, if you already had a many-to-many relationship and then decided to track additional data in the join table, it meant abandoning the many-to-many support and creating an explicit join entity (that maps directly to the join table) with a one-to-many relationship between that entity and each of the ends. This also meant the painstaking effort of navigating through that entity for every query and object interaction that involved the relationship. This is what carried through to EF Core as the only way to support many-to-many.

As an example, let's keep track of the date on which the person was associated with a particular address. Even though I have already implemented the simpler many-to-many, the existing code and mappings will continue to work. EF Core 5 lets you do this without impacting the existing relationship or about breaking a lot of code.

With EF Core 5, you do need to create an explicit entity that maps to the join table, because you need somewhere to

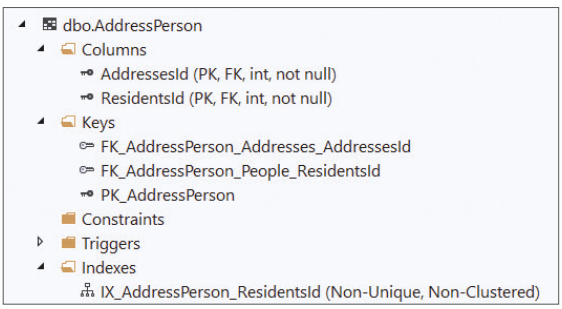


Figure 2: The schema of the Join table

capture that additional data (referred to as a payload). I'll do this with a class called Resident and add a LocatedDate property to track the new detail.

```
public class Resident
{
  public int PersonId { get; set; }
  public int AddressId { get; set; }
  public DateTime LocatedDate {
      get; private set; }
}
```

Unlike audit data, which you can configure using EF Core's shadow properties with no need to expose in your business logic, you'll be able to access this LocatedDate easily in your code.

I'll configure the mapping in DbContext.OnModelCreating.

```
modelBuilder.Entity<Person>()
  .HasMany(p => p.Addresses)
  .WithMany(a => a.Residents)
  .UsingEntity<Resident>(
    r => r.HasOne<Address>().WithMany(),
    r => r.HasOne<Person>().WithMany())
  .Property(r => r.LocatedDate)
    .HasDefaultValueSql("getdate()");
```

This mapping looked confusing to me at first but after applying it a few times, it now makes sense. Let me break it down for you so that you don't even need to wait until your second pass.

I start with the Person end, Entity<Person>, which has many addresses via its Addresses property. Then using WithMany, I explain that Addresses has many Residents (e.g. its List<Person> property). So far, this is an explicit way to map what EF is already determining through convention. And in fact, there is even a way to configure the many-to-many without even coding the navigations in the classes. Next comes the "how to do it" mapping: UsingEntity. This uses the join entity, Resident, and then specifies that every resident connects one Address and one Person. Each of those might have many resident properties, which is what the WithMany method represents. There's no need to specify the resident type in the WithMany method; EF Core will be able to infer that by convention.

Finally, I'm using a familiar mapping to specify that the database column that maps to the Resident.LocatedDate property should have a default value of the current date, using SQL Server's getdate() function.

There's one more critical change you must make in the mappings. By convention, EF Core originally named the join table AddressPerson. With the new Resident entity, EF Core convention will want that table to be named Residents. This causes the migration to drop the AddressPerson table and recreate it, causing you to lose data. To prevent this, I've included a mapping to ensure that the new Resident entity still maps to that AddressPerson table.

```
modelBuilder.Entity<Resident>()
  .ToTable("AddressPerson");
```

Do keep in mind that there are a lot of other ways you can configure the mappings for the many-to-many relationship.



Figure 3: The schema of the join table with a Payload column, LocatedDate

That's part of the beauty of this re-imagined many-to-many in EF Core 5.

When I add a new migration, EF Core creates logic to enhance the AddressPerson join table, modifying the column names and fixing up the names of foreign keys and indexes. Like the table name, you do have agency over these names using the fluent API, but I'll just let it use convention and make the changes. After calling update-database, **Figure 3** shows the schema of the modified table.

Now, when I create any new relationship between a Person and an Address, the new row in the join table will have the LocatedDate populated by SQL Server.

I can also query the Residents with a DbContext. Set<Resident> if I want to read that value. There are more ways to enhance the many-to-many relationship as well that you can learn about from the docs or perhaps a future article here in CODE Magazine. Let's look at some other EF Core 5 features now.

Simplified Logging Returns with More Smarts

Logging in Entity Framework and EF Core has had a long and lurid history. Well, not really lurid, but it sounded good, right? It was rough to get at the SQL created by EF in early days but then we got a super simple Database.Log method in EF6. This didn't make the transition to EF Core but instead, EF Core tied directly into the logging framework in .NET Core. This was a big advance. The pattern to tap into EF Core's logs was to configure the DbContext options to use a LoggingFactory. In ASP.NET, this is fairly simple because the LoggingFactory is pre-defined. In other apps, there's a bit more ceremony.

Now in EF Core 5, the team has merged the intelligence of the .NET Core logging framework and the simplicity of telling EF Core to "just do it." In fact, you don't even need to know that there's a LoggingFactory involved. Witness the loveliness of this code. In a console app where I'm explicitly specifying the optionsBuilder settings in OnConfiguring, I need only to append a LogTo method, specify the target as a delegate, e.g., Console.Writeline (notice it's not a method), and then specify how much detail to provide with a LogLevel enum.

```
optionsBuilder.UseSqlServer(myConnectionString)
  .LogTo(Console.WriteLine,LogLevel.Information);
```

This syntax is even simpler than EF6 (which required some additional settings in a config file) and much more flexible.

If you are working in ASP.NET Core, you can continue to take advantage of the built-in logging while filtering EF Core's output in the appsettings.json file.

Finally, Filtered Include!

Here's a treat we've been asking for since the beginning of time, well, of EF time: the ability to filter when eager loading with the Include method. The way it's always worked is that if you use Include to pull in related objects, it's an all or nothing grab. Include had no way to filter or sort. To get around that, we could instead use the Select method to create a projection query. Explicit (after the fact) loading also allows filtering and sorting.

Thanks again to some of the underlying capabilities recently brought into EF Core, EF Core 5 finally lets you use Where and other LINQ methods on the related properties specified in the Include method.

I've enhanced my domain to include wildlife sightings at each address and it's been a very busy morning at my house with bears, cubs, snakes, and squirrels all running about!

Here is a simple query with Include without the new capability:

```
context.Addresses
  .Include(a => a.WildlifeSightings)
  .FirstOrDefault();
```

This returns every sighting for the first address:

```
8/15/2020 10:27:31 AM: Bear
8/15/2020 10:27:31 AM: Bear Cub #1
8/15/2020 10:27:31 AM: Bear Cub #2
8/15/2020 10:27:31 AM: Bear Cub #3
8/15/2020 10:27:31 AM: Squirrel
8/15/2020 10:27:31 AM: Garter Snake
```

Now you can use LINQ methods inside the Include method. To filter to return only the bear sightings, I'll add the Where method to the related WildlifeSightings property.

```
context.Addresses.Include
  (a => a.WildlifeSightings
     .Where(w=>w.Description.Contains("Bear")))
 .FirstOrDefault();
```

If you follow me on Twitter, you may know that there has indeed been a mother bear with three cubs living in the woods in my neighborhood this past summer. I saw the momma from a distance one day, but I've not been lucky enough to see the cubs. Back to EF Core 5, which generated the correct SQL to filter on only the bear sightings. The SQL Statement is a single query and has added a filter using LEFT JOIN to access the related data. Here are the results—the squirrel and snake are gone.

```
8/15/2020 10:27:31 AM: Bear
8/15/2020 10:27:31 AM: Bear Cub #1
8/15/2020 10:27:31 AM: Bear Cub #2
8/15/2020 10:27:31 AM: Bear Cub #3
```

To filter and sort, it's just some more, familiar LINQ inside the Include method. Let's check out anything *but* bears now and sort them by name.

```
context.Addresses.Include
  (a => a.WildlifeSightings
  .OrderBy(w=>w.Description)
  .Where(w => !w.Description.Contains("Bear")))
.FirstOrDefault();
```

EF Core 5 generates SQL that returns only the snake and squirrel in alphabetical order.

```
8/15/2020 10:31:14 AM: Garter Snake
8/15/2020 10:31:14 AM: Squirrel
```

Tweak Performance with Split Queries

I mentioned the LEFT JOIN used for the filtered Include query above. The default for eager loaded queries is to build a single query. The more complexity you build into an eager loaded query, for example if you are piling on Include or ThenInclude methods when the Includes are pointing to collections, there's a good chance of the query performance degrading. This can be dramatic and I've worked with clients to solve this type of performance problem many times.

Earlier in EF Core's lifetime, the team tried to improve the performance by splitting up the query. However, users discovered that this path occasionally returned inconsistent results. In response, in EF Core 3, the team reverted back to a single query. Now in EF Core 5, although the single query remains the default, you have the option to force the query to be split up with the AsSplitQuery method. Like the AsNoTracking method, there's also a way to apply this to the context itself, not only particular queries. For example, here is the SQL from the simple Include above before I added the filter. Notice the LEFT JOIN to retrieve the related data.

```
SELECT [t].[Id], [t].[PostalCode], [t].[Street],
  [w].[Id], [w].[AddressId], [w].[DateTime],
  [w].[Description]
FROM (
     SELECT TOP(1) [a].[Id], [a].[PostalCode],
                   [a].[Street]
     FROM [Addresses] AS [a]
    ) AS [t]
LEFT JOIN [WildlifeSighting] AS [w]
ON [t].[Id] = [w].[AddressId]
ORDER BY [t].[Id], [w].[Id]
```

Adding AsSplitQuery into the LINQ query:

```
context.Addresses.AsSplitQuery()
  .Include(a => a.WildlifeSightings)
  .FirstOrDefault();
```

This results in a pair of select statements, eliminating the LEFT JOIN.

```
SELECT [t].[Id], [t].[PostalCode], [t].[Street]
FROM (
      SELECT TOP(1) [a].[Id], [a].[PostalCode],
```

```
                    [a].[Street]
      FROM [Addresses] AS [a]
    ) AS [t]
ORDER BY [t].[Id]

SELECT [w].[Id], [w].[AddressId],
       [w].[DateTime],
       [w].[Description], [t].[Id]
    FROM (
        SELECT TOP(1) [a].[Id]
        FROM [Addresses] AS [a]
    ) AS [t]
INNER JOIN [WildlifeSighting] AS [w]
    ON [t].[Id] = [w].[AddressId]
ORDER BY [t].[Id]
```

I'm not a DBA and don't have the skill to determine if this *particular* split query will result in better performance, but this is a very simple example. However, I believe that if you are diligent about profiling your queries and have someone on your team who can identify where performance will benefit from split queries, then you'll be in a position to leverage this feature. What the EF Core team has done is given you agency over this decision rather than forcing a single behavior on you.

I also tested AsSplitQuery() with filtered includes and across the many-to-many relationship with success.

Improving the Migration Experience for Production Apps

Creating a database and updating its schema as your model evolves is quite simple on your development computer with your local database. You can use PowerShell commands or the dotnet CLI to execute the migrations. EF Core also makes it easy to share migrations across a development team with migration files included in your source control and features such as idempotent migrations.

However, executing migrations in applications deployed to a server or to the cloud is a much different story. Microsoft hasn't yet landed on a great workflow for that. Some developers look to the Migrate method from the EF Core APIs during application start up to solve this problem. But if your application is deployed across multiple instances for load balancing, this could create some terrible conflicts if you hit a race condition between different instances attempting to execute the migration. Therefore, it's strongly recommended that you avoid that pattern.

The safest workflows have involved letting migrations create SQL and then using other tools to execute that SQL. For example, for SQL Server, there is tooling like dacpac or RedGate's SQL Change Automation. For a wider variety of relational databases, I've been leveraging RedGate's FlywayDB (http://flywaydb.org) in a Docker container to quickly execute migrations and then disappear. (See my article "Hybrid Database Migrations with EF Core and Flyway" at bit.ly/2E0VyiD for more on that pattern.)

The team is focused on improving this experience through better APIs and guidance along with, as per their docs, "longer-term collaboration with other teams to improve end-to-end experiences that go beyond just EF."

For the EF Core 5 timeframe, they were able to accomplish some specific tasks. For example, improvements have been made to idempotency in generated migration scripts and the EF Core CLI now allows you to pass in the database connection string when updating a database.

There have also been some changes for migrations in SQLite databases. SQLite has myriad limitations around modifying table schema, which has created a lot of problems for migrations in the past. Brice Lambson has been tackling this problem for many years in between working on many other features. He recently noted on GitHub that he originally created the relevant issue before his, now six-year old, child was born. EF Core 5 enables quite a number of previously unsupported migrations such as AlterColumn and AddForeignKey. You can see the list of migrations supported (followed by recommended workarounds) in the document, SQLite EF Core Database Provider Limitations at https://docs.microsoft.com/en-us/ef/core/providers/sqlite/limitations.

Transaction Support in Scripted Migrations

Have you ever run a migration that attempts to add a table that already exists? The migration fails and anything past that problem won't get run.

After much discussion and research, the team has added transactions to SQL generated by the **script-migration** and CLI **ef migration script** commands.

Although the BEGIN TRANSACTION and COMMIT statements are added by default, you can disable their inclusion with the NoTransactions parameter.

Transactions are used to wrap what should be natural transaction points throughout the script. You can read the discussion about how this was determined in this GitHub issue (https://github.com/dotnet/efcore/issues/7681). I would recommend that you (or in my case, someone with better DBA smarts) inspect the generated script to ensure that it's wrapping at the points that make sense. Personally, I always defer to the algorithm used to insert the commands if I don't have an expert available.

For you SQL Server gurus, here's a quick shout out to the new support for transaction savepoints. You can learn more about that at https://github.com/dotnet/EntityFramework.Docs/issues/2429.

Handy for Testing and Demos: ChangeTracker.Clear

In my years of researching the capabilities of Entity Framework and EF Core, I've often wanted to create a context and use it to try various things in one go, rather than creating and disposing short-lived contexts as I would do in a production app. However, one of the problems with that is that the context never forgets about entities that it's tracking and if I'm not careful, my results will be wrong and lead me astray.

For these scenarios—and I'll stress that this isn't recommended for production code—I've always wished that I could easily purge the change tracker's cache without having to instantiate a new context. EF Core 5 finally gives me the tool to fulfill that wish: the ChangeTracker.Clear() method. It just tells a DbContext's ChangeTracker to forget everything it knows and act as though it was newly instantiated. Thanks team!

New Support for DbContext and Dependency Injection in Blazor

ASP.NET Core has made it so easy to let your Web app spin up DbContext instances as needed without hard coding new instances all over your application, creating tight coupling and having to manage their lifetime.

But when Blazor, Microsoft's client-side UI framework, showed up on the scene, a lot of developers struggled to find a way to get the same benefit. Doing so involved creating your own class that implements IDbContextFactory. Not only was this more complex than the IServiceCollection.AddDbContext method in ASP.NET Core's startup class, it wasn't discoverable, although it's documented.

The Blazor team got together with the EF Core team to tackle this problem and now there's a nice solution that can be used in Blazor and other application types with the ease that we already experience with ASP.NET Core.

The new IServiceCollection.AddDbContextFactory is the magic potion. There's also a version for when you are pooling DbContexts: the AddPooledDbContextFactory.

Like ASP.NET Core, Blazor has a startup class with a ConfigureServices method. So, it's easy to use the new method to a factory into the DI services rather than a DbContext. The code looks similar to AddDbContext.

```
services.AddDbContextFactory<PeopleContext>
 (opt => opt.UseSqlServer
    ($"Data Source={myconnectionstring}")
 );
```

Like AddDbContext, you specify the options (e.g., provider, connection string, and things like EnableSensitiveLogging) for the DbContext instances that the factory will spin up for you.

Then, in code where you're using that injected factory, you only need to call its CreateContext method to get a ready-to-use instance of your context.

```
var context = _contextFactory.CreateDbContext()
```

Don't forget (as I often do) that your DbContext class will need a constructor that accepts the DbContextOptions. This isn't new, but something you've needed to do for using DbContext in dependency injection since the beginning of ASP.NET Core. As a reminder, this is what that constructor looks like:

```
public PeopleContext
  (DbContextOptions<PeopleContext> options)
  : base(options)
{
}
```

But there's a big difference between using AddDbContext and AddDbContextFactory. AddDbContext scopes the lifetime of the DbContexts that get created to the request that triggers its creation (e.g., a request to a controller method). But when you use the factory to create those contexts, you'll be responsible for the lifetime of each DbContext instance. In other words, you'll trigger the factory to create those instances as needed and then be responsible for disposing them.

In the Blazor app setup, EF Core migrations commands will be able to find the provider and connection string just as they do with ASP.NET Core.

The Return of Table-Per-Type Inheritance

TPT inheritance (where inherited types are stored in their own tables in the database) was a feature of EF from the start through to EF6. Like the many-to-many support, EF Core didn't have the needed building blocks to create a smarter implementation with better-performing SQL queries. However, it was one of the top requested features and now it's back in EF Core 5.

In the 300+ comment GitHub issue for TPT, EF team engineer Smit Patel shares (in https://github.com/dotnet/efcore/issues/2266#issuecomment-653661902) sample output comparing a TPT query in EF6 and EF Core 5 (**Figure 4**). It's surely more readable but there are also performance benefits as well, described in this same comment.

Setting up TPT works just as it did with EF6.

By default, you create an inherited type, e.g., ScaryWildlifeSighting that inherits from WildlifeSighting like this:

```
public class ScaryWildlifeSighting
 : WildlifeSighting
{
  public string Experience { get; set; }
}
```

The convention for EF Core is to treat this as a Table-Per-Hierarchy mapping, where the properties of ScaryWildlifeSighting are added to the WildlifeSIghtings table.

You can change this to TPT by letting EF Core know to put ScaryWildlifeSighting data into its own table using the [Table("ScaryWildlifeSightings")] data annotation or the ToTable mapping in OnModelCreating:

```
  modelBuilder.Entity<ScaryWildlifeSighting>()
          .ToTable("ScarySightings");
```

And you can create, update, and query the related data just as you would have in EF6. A query to find all of the scary sightings with the phrase "peed my pants," for example, would look like:

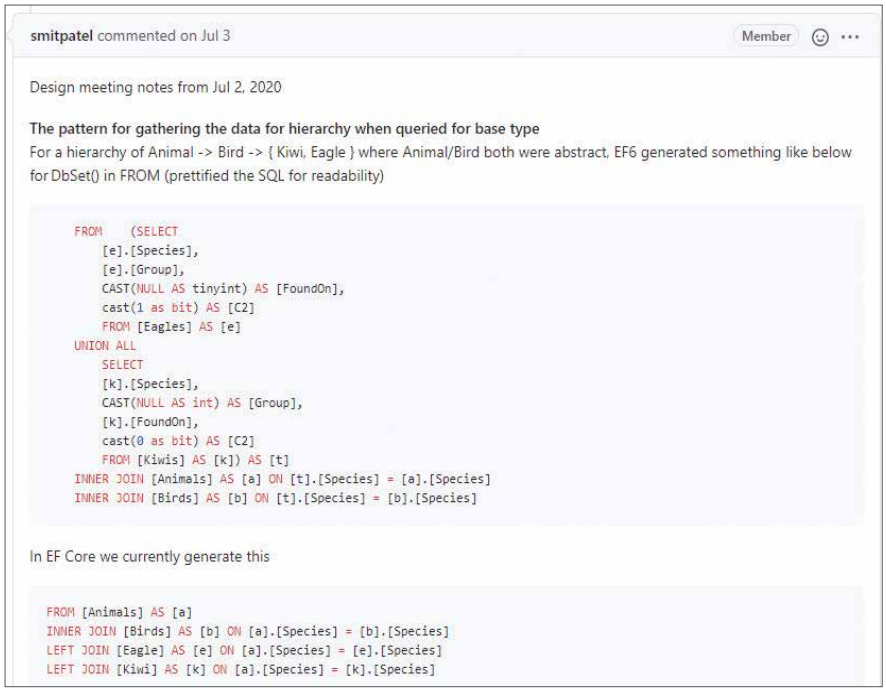
```
context.WildlifeSightings
  .OfType<ScaryWildlifeSighting>()
  .Where(s => s.Experience.Contains("peed"))
  .ToList();
```

So Many More Interesting and Useful Features

Obviously, I can't tell you about all 255 features and enhancements that have been added to EF Core 5. And it's so hard to pick which ones to list here in the final bit of this article.

I must give a shout out to a few things:

- Many enhancements have been made to the Cosmos DB provider.
- There are more ways to tap into SaveChanges besides overriding the method. For example, there are events and an interceptor.



smitpatel commented on Jul 3 — Member

Design meeting notes from Jul 2, 2020

The pattern for gathering the data for hierarchy when queried for base type

For a hierarchy of Animal -> Bird -> { Kiwi, Eagle } where Animal/Bird both were abstract, EF6 generated something like below for DbSet() in FROM (prettified the SQL for readability)

```
FROM    (SELECT
    [e].[Species],
    [e].[Group],
    CAST(NULL AS tinyint) AS [FoundOn],
    cast(1 as bit) AS [C2]
    FROM [Eagles] AS [e]
UNION ALL
    SELECT
    [k].[Species],
    CAST(NULL AS int) AS [Group],
    [k].[FoundOn],
    cast(0 as bit) AS [C2]
    FROM [Kiwis] AS [k]) AS [t]
INNER JOIN [Animals] AS [a] ON [t].[Species] = [a].[Species]
INNER JOIN [Birds] AS [b] ON [t].[Species] = [b].[Species]
```

In EF Core we currently generate this

```
FROM [Animals] AS [a]
INNER JOIN [Birds] AS [b] ON [a].[Species] = [b].[Species]
LEFT JOIN [Eagle] AS [e] ON [a].[Species] = [e].[Species]
LEFT JOIN [Kiwi] AS [k] ON [a].[Species] = [k].[Species]
```

Figure 4: Comparison of SQL generated from a TPT query in EF6 vs. EF Core 5

SPONSORED SIDEBAR:

Get .NET Core Help for Free

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

- If you were impacted by the regression in EF Core 3.0 that caused serious problems with the performance of queries involving owned entities, you'll be happy (as am I) to know that this was fixed for EF Core 5.
- Index properties are an interesting feature that are something like dynamic types but specific to EF Core. You might not use them directly (although you could) but they are also a building block that other features, such as many-to-many, rely upon.

And now I will let Brice Lambson from the EF team take over because he posted two great tweets about his favorite features in EF Core 5!

In his first tweet (https://twitter.com/bricelambs/status/1295789002876784645) he listed features that bring additional EF6 parity to EF Core 5. I've added notes to his list to provide some insight for you:

- Simple logging (covered above)
- Get the SQL of a query
  - Example: Console.WriteLine(someLinqQuery.ToQueryString();
- HasPrecision
  - Example: modelBuilder.Entity<Address>().Property(b=>b.SqFeet).HasPrecision(16,4)
- Defining query
  - EF Core 5 makes it easier to specify a defining query of an entity type as SQL (like EF6) and at the same time, they deprecated some of the APIs around using LINQ for defining queries.
- Pluralization while scaffolding
  - Enabled by the Humanizer project (http://github.com/Humanizr/Humanizer) now being used as the pluralization service.
- Specify connection string in tools
  - The migrations CLI now has a parameter called **--connection** to which you can pass the string
- Get-Migration
  - This, along with the CLI version (dotnet ef migrations list), will list the status of your migrations, e.g., what's been applied, what's pending, and if there are model changes that still need to be migrated.

In the second tweet (https://twitter.com/bricelambs/status/1295789904077582336), Brice listed some new features that he is especially fond of.

- Collation
- Indexer properties
- AddDbContextFactory (covered above)
- Ignore in Migrations
  - You can edit a migration and force it to ignore a table that the migration wants to create. There's an interesting discussion in the relevant GitHub issue. See https://github.com/dotnet/efcore/issues/2725.
- SQLite Computed columns
  - The SQLite provider now supports HasComputedColumn. But there's something else I want to point out. EF (and EF Core) have had computed column support for a long time for SQL Server. And if you wanted stored columns, you could use SQL Server's PERSISTED attribute so that searches don't need to build that value on the fly. Now there's a "stored" attribute on EF Core's HasComputedColumn mapping, which is a simpler and more discoverable way to force the database to store the value so that search and indexing can benefit.
- SQLite Table rebuilds (mentioned as part of the SQLite migrations above)
- Better SQL (the team is always improving the SQL translations)
- Simplifications for Discriminator, NULL and CASE
- Transactions and EXEC in migrations script
- Better LINQ translation errors (See https://docs.microsoft.com/en-us/ef/core/what-is-new/ef-core-5.0/whatsnew#improved-query-translation-exceptions for more information.)

Microsoft Continues to Invest in EF Core, So You Can, Too

In the old days of Entity Framework, there was a theme of "evolution, not revolution" when it came to new versions. Moving from EF to EF Core was certainly a big revolution, however that wasn't an update of EF. EF Core 3 was the first time that there was what we would call a revolution. But now that we're past that, EF Core is back on a path of evolving in great ways. EF Core 5 is a very sophisticated ORM and most importantly, the investment made into this version is more proof that Microsoft is dedicated to keeping EF Core around for a good long time to come. I hope you'll dig into the excellent documentation as well as the resources in GitHub to continue to discover what this version brings.

Remember also to check out the team's standups at http://bit.ly/EFCoreStandups and watch out for an update to my Getting Started with EF Core course (http://bit.ly/EFCore31) on Pluralsight sometime after EF Core 5 is released.

Julie Lerman

Project Tye: Creating Microservices in a .NET Way

Regardless of what you call your architecture or development practice, no doubt it's composed of many services built from many deployed projects and there's some pain in trying to make it all work. Microservice development, technology stack aside, can be a long list of tasks just in itself, including tools to use, how to get services set up, mapping ports, learning terminology, and deployment tasks. The idea of simply writing, running, and checking in the code is the ultimate goal toward which developers strive within the inner loop development cycle and Project Tye looks to push the pain away and get to just coding and debugging your services.

Tye is an experimental developer tool from the .NET team at Microsoft that's meant to make the experience of creating, testing and deploying microservices easier. Specifically, .NET microservices understands how .NET applications are built and work.

Getting Started with Tye

Tye is a .NET global tool, and you'll need to have the .NET Core SDK installed on your computer. Run the following command to install it globally.

```
dotnet tool install -g Microsoft.Tye
```

Although you could build, run, and debug your services without Tye, the goals of the project are the reasons you want it for developing many services together: running multiple service with a single command, a service discovery based on convention, and simplicity of deployment to Kubernetes with minimal configuration.

A Simple App

A very basic example of a simple app is a front-end application with a data service back-end. Create a front-end Razor Pages application within a **myweatherapp** folder.

```
mkdir myweatherapp
cd myweatherapp

dotnet new razor -n frontend
```

The standard dotnet commands could be used to run, build, and debug the application, but instead, you're going to use Tye to run the app.

Type the command for Tye run the front-end application.

```
tye run frontend
```

The Tye run command creates the output shown in **Figure 1,** that builds and starts all services in the application.

The run command identifies the applications and builds them, automatically setting the ports for bindings as these may change due to ports in use/conflicts. Finally, a dashboard is stated to view all of the services running in the application.

Opening the dashboard address reveals the service and some basic information. The Tye Dashboard is shown in **Figure 2**.

Looking further into the dashboard, the name of the service links to metrics, as shown in **Figure 3**.

For this application, the Bindings column displays the URIs for the front-end application and the Logs column links to a display for a console-style streaming logs display. This is

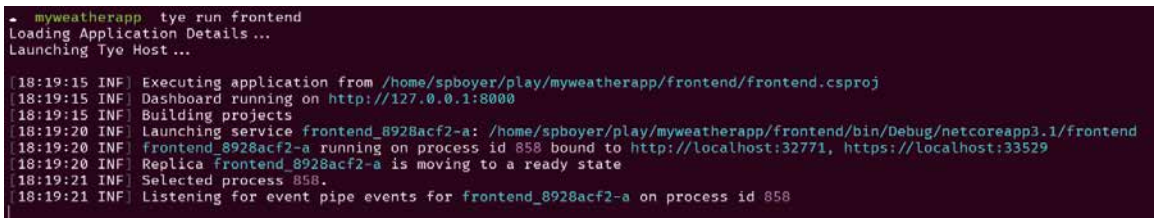


Figure 1: The Tye run command output.



Figure 2: The Tye Dashboard.



Shayne Boyer
shboyer@microsoft.com
tattoocoder.com
@spboyer

Shayne is a Principal Developer Advocate at Microsoft contributing to and working on ASP.NET, microservices, Azure, open source, and related tools.

similar to that typically produced if the dotnet command line were used to start the application, as shown in **Figure 4**.

Unless port bindings are explicitly defined in the bindings, each service is assigned a random port to avoid conflicts. A common issue when creating multiple services is tracking these ports in configuration files and mapping them from service to service.

Listing 1: Use GetServiceUri passing the service name

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddRazorPages();
   services.AddHttpClient<WeatherClient>(client =>
   {
      client.BaseAddress = Configuration.GetServiceUri("backend");

   });
}
```

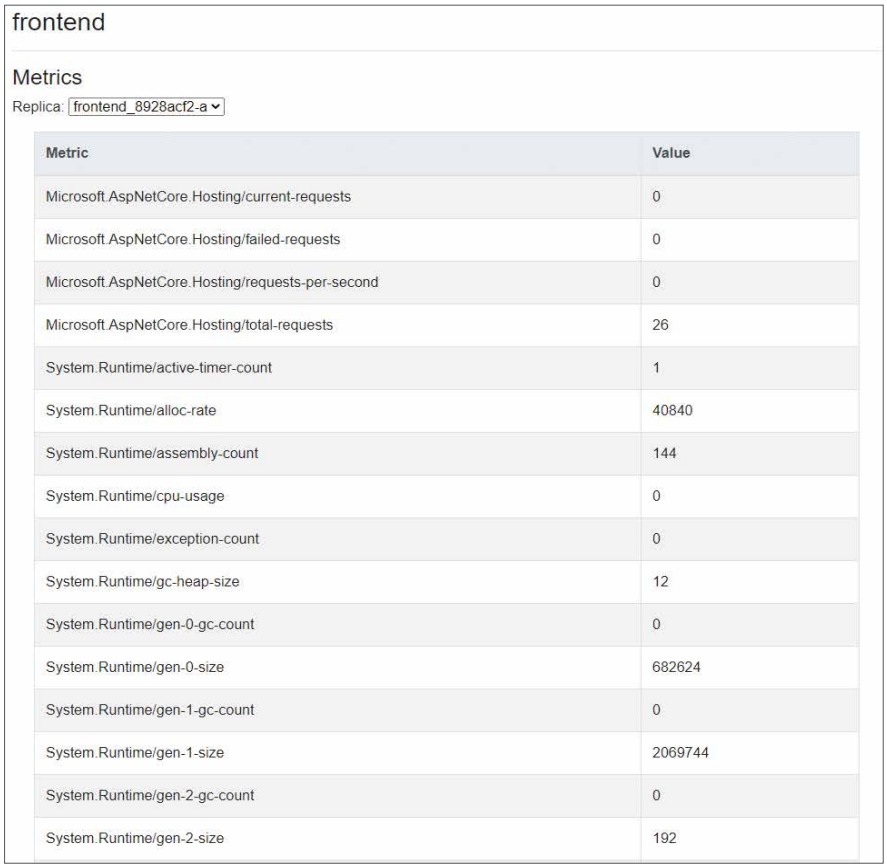


Figure 3: The Tye Dashboard Metrics section.



Figure 4: The Tye Dashboard Logs section.

Adding Multiple Services

Having a single application is the easy part. With microservices, applications are composed into small components and work together to accomplish the overall application's task. The orchestration of these components when developing code locally can involve starting up multiple debugging instances, IDEs, and/or consoles to get it all working.

Let's add a weather api service and see how Tye helps. First, create the new service with the .NET CLI and call it **backend**.

```
dotnet new webapi -n backend
```

Tye works with the understanding of .NET projects and solutions, in this case, creating a solution file. Adding the projects to it is the next step.

```
dotnet new sln -n myweatherapp
dotnet sln add frontend backend
```

You could use the run command from Tye and see the applications running in the dashboard, but at this point, there are just two applications running and no code has been added to wire them up. Usually, you might put the URI of the api in a configuration file or environment variable, noting that it may change when moving through environments while promoting the application.

As a part of this project, the Tye team has created an extension to the Configuration NuGet package called **Microsoft.Tye.Extensions.Configuration**. Adding a reference to this package allows your application to easily access the generated URI and ports needed for service discovery not only during local development, but there's also no need for you to change your code when deploying your applications to Kubernetes.

In the example of the front-end/back-end application, after adding a reference to the new package in the front-end; you can set up a new HttpClient in the ConfigureServices method, as shown in **Listing 1,** where the api endpoint is referred to as **backend**.

After adding the necessary code in the Razor page to render the data from the api, run the command to start the applications and view the dashboard.

```
tye run --dashboard
```

The dashboard in **Figure 5** shows both applications running with the random assigned ports for the front-end and back-end services.

Note that the code didn't define any port or specific URI for the weather api back-end. All you added was the configuration call for the **backend** service. Clicking on the URL for the front-end application shows the Web application running and retrieving the data, as shown in **Figure 6**.

Debugging Services

Adding breakpoints, stepping through code, and debugging statements are crucial processes in the inner development cycle. While Tye is running the services, it also exposes debugging hooks to attach to from editors and IDEs.

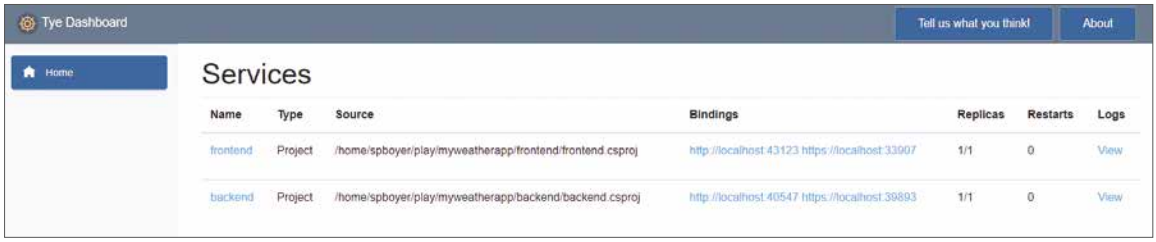


Figure 5: The Tye Dashboard displays services running and port bindings.

Each service can start in debug mode by passing in the **--debug** flag with the service name.

```
tye run --debug backend
```

This allows an editor like VS Code to attach to the back-end and hit a breakpoint. Alternatively, passing "*" instead of a specific service name exposes all the service's debug hooks, as shown in **Figure 7**.

Once the debugger is attached and the breakpoint is hit, the experience is the same as you'd expect. Locals, step through processes, call stacks, and other tools are there to use, as shown in **Figure 8**.

External Dependencies

Adding more .NET Core services to this project, running, and debugging them is pretty easy: The process is the same. However, in the microservice world, applications are built on many existing services, databases, and other processes.

Up to this point, you've depended on Tye's inherent understanding of the .NET project and solution system to build and start the applications along with a new configuration package. When adding in external services or needing more control over what happens for the services, a configuration file, **tye.yaml**, is used for these setting.

To generate this file, Tye has an **init** command to create the default file based on the projects that exist in the current working directory.

```
tye init
```

For the myweatherapp example, the tye.yaml file created looks like **Listing 2**.

In the file, each service is listed with configuration specifics. To add a database service like PostgreSQL, you could install that on your local computer and add it to appsettings.json as a connectionstring setting, or add a service configuration in tye.yaml and take advantage of the Docker container capabilities of Tye. When running the application, the lifecycle of all services is managed from Tye, starting and stopping the containers along with your application, as shown in **Listing 3.**

Adding a new section called "postgres" and defining the **image** instructs Tye to pull the Docker image when the run command is issued. In this same section, the password for the connection string is defined as a new environment variable. Finally, the port is hard-coded as 5432 as that is the default for PostgreSQL and you don't want a dynamic port set each time.

Tye Run Is a Single Command Running All Services

The run command builds and/or starts all of the services within the application. This command is inclusive of the dotnet restore and build, if needed for the services. If there were containers, the Docker commands are inclusive as well.

Listing 2: standard tye.yaml file

```
# tye application configuration file
# read all about it at https://github.com/dotnet/tye

name: myweatherapp
services:
- name: frontend
  project: frontend/frontend.csproj
- name: backend
  project: backend/backend.csproj
```

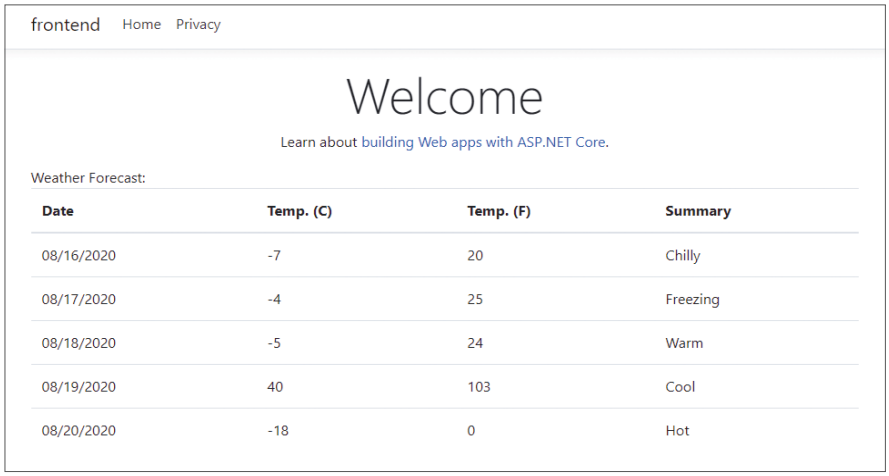


Figure 6: The front-end application runs on the random port it was assigned.

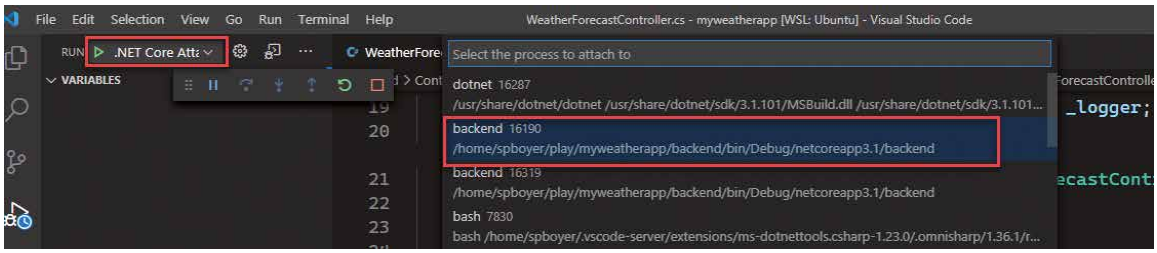


Figure 7: Attach the debugger to the running service with the –debug option flag.

Note the string interpolation formation for some of the values in the connection string for the database. The ${host} and ${port) values substitute for localhost:5432 in local development, but if the port was dynamically assigned; it would properly set as well. This is where using **Microsoft.Tye.Extensions.Configuration** to retrieve the values shows its value.

Listing 3: tye.yaml adding postgres service

```
name: myweatherapp
services:
- name: frontend
  project: frontend/frontend.csproj
- name: backend
  project: backend/backend.csproj
- name: postgres
  image:  postgres
  env:
  - name: POSTGRES_PASSWORD
    value: "pass@word1"
  bindings:
  - port: 5432
    connectionString: Server=${host};Port=${port};User Id=postgres;
Password=${env:POSTGRES_PASSWORD};
```

When **tye run** is executed and the dashboard is loaded, notice that the front-end and back-end services are noted as "Project" where postgres is a "Container", as shown in **Figure 9**.

Just as in the initial set up for getting the weather api URI, the postgres connection string is available via service discovery using the **GetConnectionString** method of the Configuration extension.

```
Configuration.GetConnectionString("postgres");
```

To this point, there has been no discussion about Dockerfiles, containers or base images other than for the PostgresSQL image. Yet another benefit in Tye is the ability to continually add dependencies to your application without writing the Docker files.

Running Apps as Containers

In **Figure 9,** the dashboard shows the two projects running and the postgres service running as a service. There are opinions in the community stating that development should get as close to the production state during development as possible.

Tye Offers an Extension for Configuration

Tracking URLs in settings or ports is cumbersome, often leading to file transforms or environment specific files; the NuGet package offers a no code change approach accessing these settings by service name.

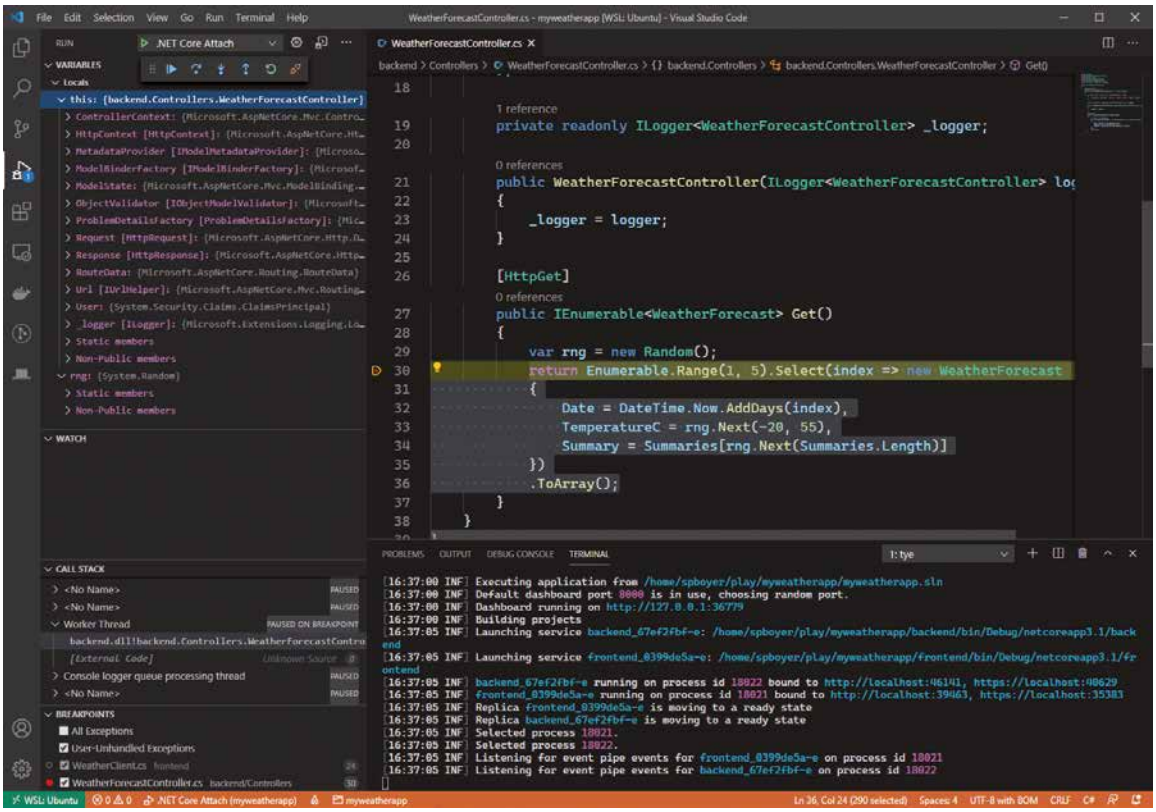


Figure 8: Debugging services in VS Code, using the WSL and editing on Ubuntu.



Figure 9: The dashboard displaying postgres service running as a container.

In this case, that means containers. Tye offers an option flag, **--docker**, where, when the services are started, they're all built in containers and run within Docker. Note that Docker for Windows or your OS is required to be installed and running.

```
tye run --docker
```

Tye pulls the base images for the projects based on the project type, builds the project and image, and subsequently runs the image(s), also creating any necessary networks for service discovery.

All of the services are now running in containers on Linux-based images. There was no need to learn the syntax to write a dockerfile or docker-compose nor understand how or where these files should be stored in your structure. It just works with Tye; all you need is a run command with the **--docker** flag.

```
➜ myweatherapp tye run --docker
Loading Application Details ...
Launching Tye Host ...

[10:47:52 INF] Executing application from /home/spboyer/play/myweatherapp/tye.yaml
[10:47:52 INF] Dashboard running on http://127.0.0.1:8000
[10:47:52 INF] Publishing project /home/spboyer/play/myweatherapp/frontend/frontend.csproj
[10:47:52 INF] Publishing project /home/spboyer/play/myweatherapp/backend/backend.csproj
[10:47:56 INF] Docker image postgres already installed
[10:47:56 INF] Docker image mcr.microsoft.com/dotnet/core/aspnet:3.1 already installed
[10:47:57 INF] removed container frontend-proxy_982e1698-a from previous run
[10:47:58 INF] removed container backend-proxy_1761d74d-1 from previous run
[10:47:59 INF] removed container postgres_5c127064-c from previous run
[10:47:59 INF] Creating docker network tye_network_193b8894-6
[10:47:59 INF] Running docker command network create --driver bridge tye_network_193b8894-6
[10:47:59 INF] Running image mcr.microsoft.com/dotnet/core/aspnet:3.1 for backend_9e6ed2c2-2
[10:47:59 INF] Running image postgres for postgres_27217055-b
[10:47:59 INF] Running image mcr.microsoft.com/dotnet/core/aspnet:3.1 for frontend_ccc1fb5a-5
```

Figure 10: The Tye run command with the --docker option pulling the needed container images.

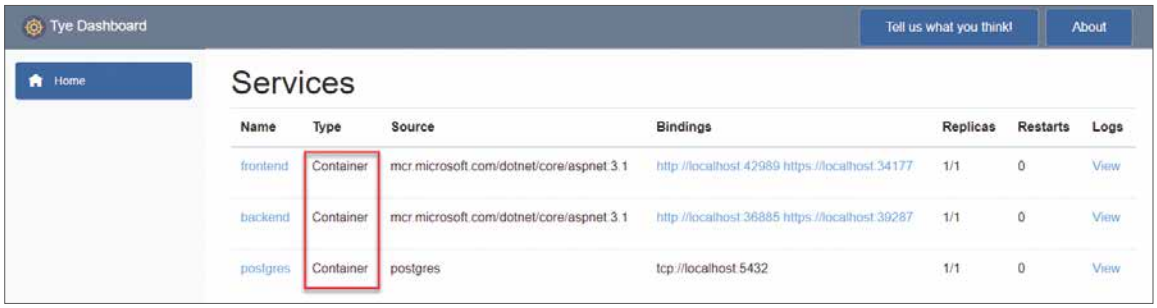


Figure 11: The dashboard running with --docker and all services running as containers.

Deploying Your Applications

Tye's deployment target is only to Kubernetes for a few reasons. First, Kubernetes is widely used in production for global companies across an array of industries. Second, it's vendor-neutral with support offered from many cloud providers? Finally, the open source community behind Kubernetes is strong.

Deploying microservice applications isn't always a simple task. Many of the common concerns include creating the containers, pushing the docker images to the specified repositories, understanding Kubernetes manifests, secrets for connection strings, and surely, depending on your app, something else.

Tye is here to help. The **deploy** command aims to address each of these items when run. Optionally, there's an interactive flag (**-i|--interactive**) that prompts you for your image repository among other detected dependencies such as external database connection string (setup in secrets) and ingress requirements.

Here's an example of the output for the myweatherapp project using the interactive flag, as shown in **Figure 12**.

```
tye deploy --interactive
```

For services to communicate inside of a Kubernetes cluster, Tye sets environment variables for service discovery, ports, and connection strings appropriately. For external services, Kubernetes secrets are used during deployments. There's no changing of configuration files, creating Helm charts, or chasing port numbers; Tye handles these for each deployment locally or when doing the deployment. The application is built and deployed to the Kubernetes cluster based on the .kubeconfig context set on your computer. Post deployment, using the Kubernetes command line tool **kubectl** to inspect the pods; the front-end and back-end applications are deployed, as shown in **Figure 13**.

```
➜ myweatherapp kubectl get pods
NAME                        READY   STATUS    RESTARTS   AGE
backend-5599fbf889-7jm6w    1/1     Running   0          148m
frontend-69fd9b7ddf-klccr   1/1     Running   0          148m
```

Figure 13: The kubectl get tool issues the pods command used to list the services running in the cluster.

```
➜ myweatherapp kubectl get secrets
NAME                                TYPE                                  DATA
binding-production-postgres-secret  Opaque                                1
```

Figure 14: The kubectl get secrets command is used to view the secrets, like the connection strings stored/set by the Tye deploy command.

The connection string for the postgres service is stored in secrets and, using the same utility, calls **get secrets.** You can see that the token exists in **Figure 14.**

The deploy command is an inclusive command, meaning the building and pushing of the containers to the registry happens inside the single gesture. Tye also offers doing this work using the **push** command. Or if just building the containers is the desired task, use the **build** command.

In short, Tye attempts to simplify the multiple gestures for all the services within an application to a single command.

The Tye Dashboard Is a Captive View into All Services

A single panel with links to all services logs, statistics, and diagnostics creates a simpler way to gain access to what is happening with your application.

Listing 4: Github Action for Tye CI/CD

```
name: Build and Deploy

on: [push]

env:
  AZURE_AKS_CLUSTER: myaksclustername
  AKS_RESOURCE_GROUP: myaksresourcegroup
  ACR_RESOURCE_URI: myregistry.azurecr.io

jobs:
  build:
    if: github.event_name == 'push' && \
contains(toJson(github.event.commits), '***NO_CI***') == false && \
 contains(toJson(github.event.commits), '[ci skip]') == false && \
contains(toJson(github.event.commits), '[skip ci]') == false
    name: tye deploy
    runs-on: ubuntu-latest
    steps:
    - name: Checkout
      uses: actions/checkout@v2

    - name: Setup .NET Core
      uses: actions/setup-dotnet@v1.5.0
      with:
        dotnet-version: 3.1.300

    - name: Install Tye tools
      run: |
        dotnet tool install -g Microsoft.Tye \
        --version "0.4.0-alpha.20371.1"

    - name: Login to ACR
      uses: Azure/docker-login@v1
      with:
        login-server: ${{ env.ACR_RESOURCE_URI }}
        username: ${{ secrets.ACR_USER }}
        password: ${{ secrets.ACR_PASSWORD }}

    - name: Set AKS context
      uses: azure/aks-set-context@v1
      with:
        creds: '${{ secrets.AZURE_CREDENTIALS }}'
        cluster-name: ${{ env.AZURE_AKS_CLUSTER }}
        resource-group: ${{ env.AKS_RESOURCE_GROUP }}

    - name: Install ingress-nginx
      run: |
        kubectl apply -f https://aka.ms/tye/ingress/deploy

    - name: tye deploy
      run: |
        tye deploy -v Debug
```

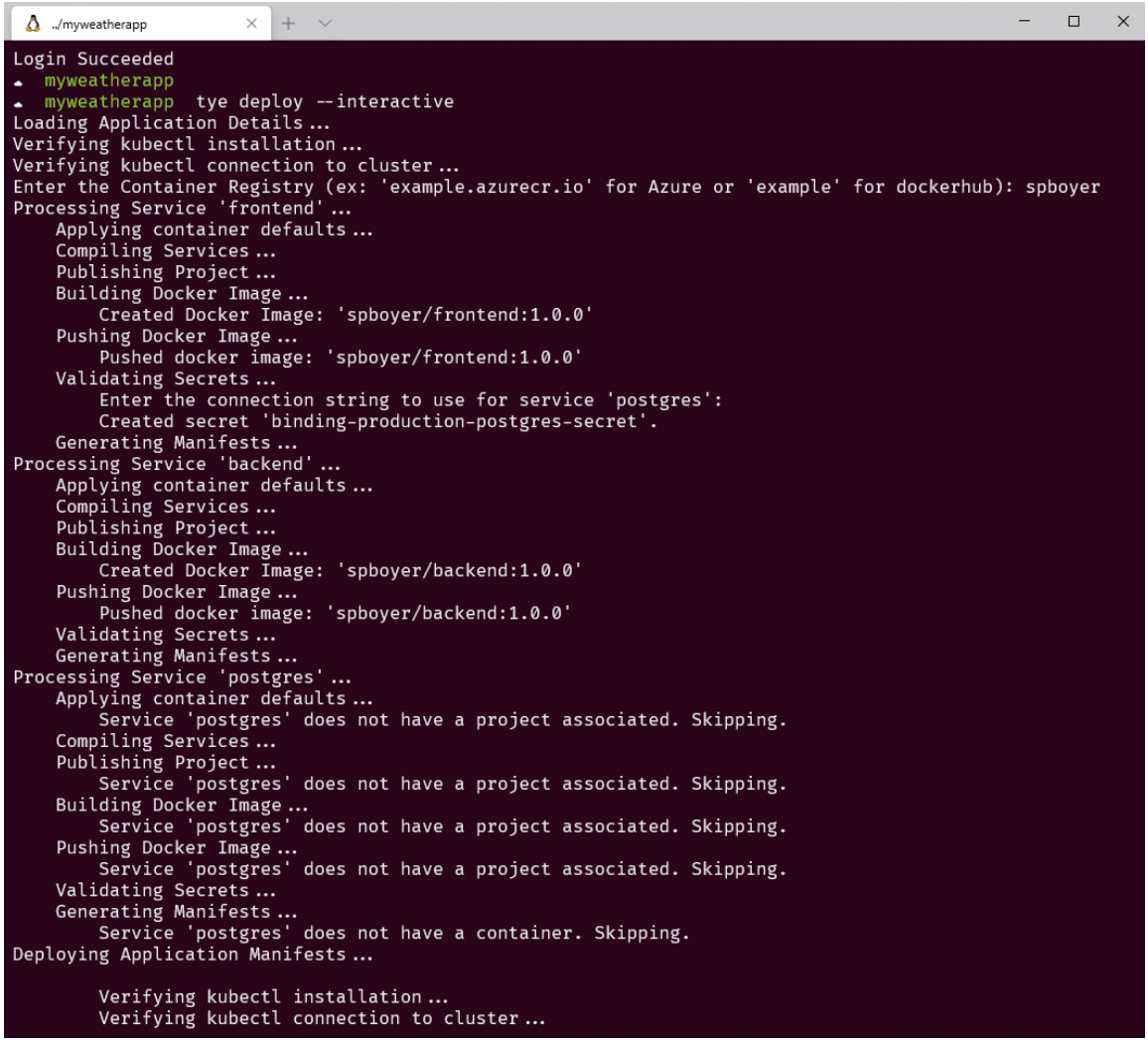


Figure 12: Output for deploying. The image is built for each project, the postgres connection is prompted, and the application is deployed to a Kubernetes cluster.

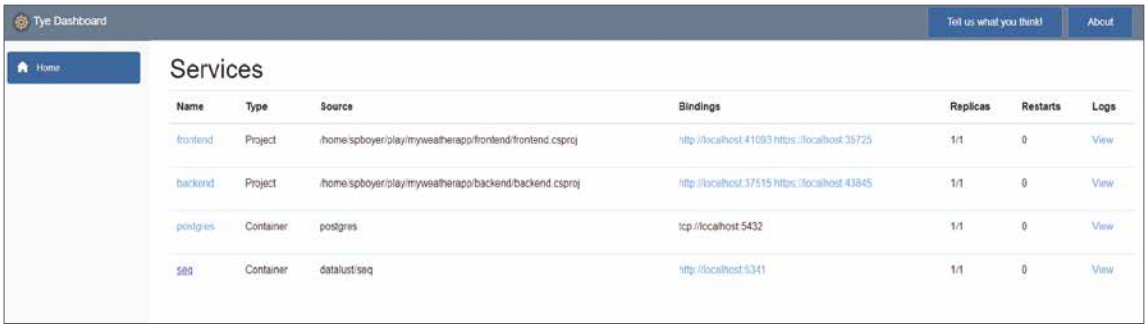


Figure 15: The dashboard displays seq as an additional service.

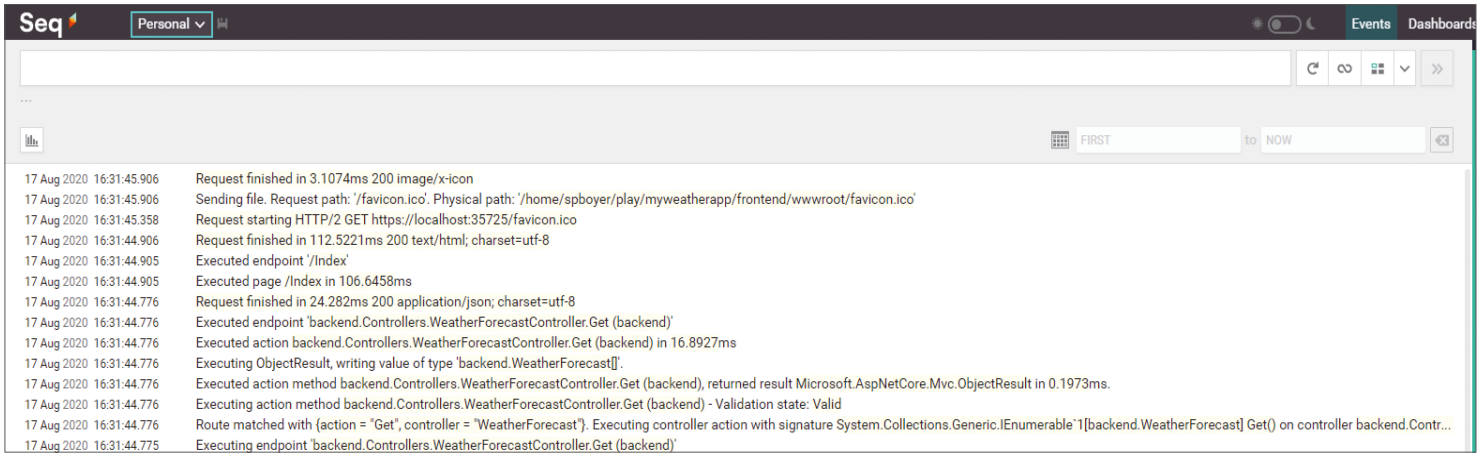


Figure 16: The Seq dashboard shows application logs.

Continuous Deployment and DevOps

Although using Tye in a "right click deploy" style fashion, calling deploy for each deployment is great for testing your code in an environment, but it isn't optimal.

Using something like Azure DevOps or GitHub Actions where checking in your code and the services can be deployed is a more likely path to success. Because Tye is a command line tool, setting this process up can be accomplished in any devops system. A recipe for using GitHub actions is documented in the repository at https://aka.ms/tye/recipes/githubaction and is shown in **Listing 4**.

Extensions

Beyond the built-in capabilities, there are extensions being added to support well-known logging services, such as Elastic and Seq, distributed tracing support for Zipkin, and an extension for Dapr, the event-driven runtime.

Here's an example of adding support for Seq, a simple addition to the tye.yaml file.

```
name: myweatherapp

extensions:
  - name: seq
    logPath: ./.logs

services:
- name: frontend
  project: frontend/frontend.csproj
```

When **tye run** is used, the proper Docker image is pulled and run and now, without any code changes, the logs are pushed to Seq, as shown in **Figure 15** and **Figure 16.**

As the project continues to solve for configuration, logging and diagnostic type services, and other popular microservice patterns and practices; more extensions are bound to surface.

Roadmap

Project Tye is an experimental tool. The goals of the project are solving for or easing the development pain points in service discovery, diagnostics, observability, configuration, and logging when it comes to microservices.

.NET has a rich ecosystem of tools, IDEs and it continues to improve with tools from team and community contributions like this project. All of the source code is available at https://github.com/dotnet/tye, and you'll also find samples, docs, and recipes. Look for monthly release cadence based on your feedback and contributions.

Shayne Boyer

Azure Functions Extension

Tye has local development support for running Azure Functions' serverless platform by setting up the service as an **azureFunction** in tye.yaml.

Big Data and Machine Learning in .NET 5

The theme for .NET 5 is a unified runtime and framework that can be used everywhere with uniform runtime behaviors and developer experiences. Microsoft released tools to work with big data (.NET for Spark) and machine learning (ML.NET) in .NET that work together to provide a productive end-to-end experience. In this article, we'll cover the basics of .NET for Spark, big data, ML.NET, and machine learning, and we'll dig into the APIs and capabilities to show you how straightforward it is to get started building and consuming your own Spark jobs and ML.NET models.



Jeremy Likness
jeremy.likness@microsoft.com
blog.jeremylikness.com/
@JeremyLikness

Jeremy Likness is a Program Manager at Microsoft on the .NET team. Jeremy works on the overall experience interacting with data from .NET with a focus on Entity Framework Core and .NET for Spark. Jeremy is an author and international speaker. He was diagnosed with young onset Parkinson's Disease in early 2020 and advocates for awareness and research for a cure. He spends most of his free time hiking the beautiful mountains of the Pacific Northwest.



Bri Achtman
briana.achtman@microsoft.com
@briacht

Bri is a Program Manager at Microsoft on the .NET team, currently working on ML.NET and Model Builder. She spends her time finding and sharing the many interesting ways .NET developers are using machine learning and improving the user experience for ML.NET. Outside of work, Bri is a Vanderbilt alumnus who loves taking her dog on long [illegible] the cream shop.

What's Big Data?

Big data is an industry term that is practically self-explanatory. The term refers to large data sets, typically involving terabytes or even petabytes of information, that are used as input for analysis to reveal patterns and trends in the data. The key differentiator between big data and traditional workloads is that big data tends to be too large, complex, or variable for traditional databases and applications to handle. A popular way to classify data is referred to as the "Three Vs."

- **Volume**: The amount of data, including number of items and size of documents or records
- **Variety:** The number of different structures, types, or schemas represented by the data set
- **Velocity:** How fast the data changes

Most big data solutions provide the means to store data in a data warehouse that's typically a distributed cluster optimized for rapid retrieval and parallel processing. Dealing with big data often involves multiple steps, as demonstrated in **Figure 1**.

.NET 5 developers who need to create analytics and insights based on large data sets can take advantage of a popular big data solution named Apache Spark™ by using .NET for Spark.

What's .NET for Spark?

.NET for Spark is based on Apache Spark, an open-source analytics engine for processing big data. It's designed to process large amounts of data in memory to provide better performance than other solutions that rely on persistent storage. It's a distributed system and processes workloads in parallel. It provides support for loading data, querying data (as part of a process and interactively), processing data, and outputting data.

Apache Spark supports Java, Scala, Python, R, and SQL out of the box. Microsoft created .NET for Spark to add support for .NET. The solution provides free, open-course, cross-platform tools for building big data applications using .NET-supported languages like C# and F# so that you can use existing .NET libraries while taking advantage of Spark features such as SparkSQL. **Figure 2** illustrates the high-level solution stack.

Listing 1 displays a small but complete .NET for Spark application that reads a text file and outputs the word count in descending order.

Setting up .NET for Apache Spark on a development computer involves installation of several dependencies, including the Java SDK and Apache Spark. You can access detailed step-by-step directions in the Getting Started guide at https://aka.ms/go-spark-net.

Spark for .NET is designed to run in multiple environments and can be deployed to run in the cloud. Possible deployment targets include Azure HDInsight, Azure Synapse, AWS EMR Spark, and Databricks. Cloud-based jobs connect to storage accounts for data. If the data is available as part of your project, you can submit it along with your other project files.

Big data is often used in conjunction with machine learning to gain insights about the data.

What's Machine Learning?

First, let's go over the basics of artificial intelligence and machine learning.

Artificial intelligence (AI) is the ability of a computer to imitate human intelligence and abilities, such as reasoning and finding meaning. Typical AI techniques often start as rule- or logic-based systems. As a simple example, think about the scenario where you want to classify something as "bread" or "not bread." When you start, it may seem like an easy problem, such as "if it has eyes, it's not bread." However, you will quickly start to realize that there are a lot of different features that can characterize something as bread vs. not bread, and the more features you have, the longer and more complex the series of if statements will get, as demonstrated in **Figure 3**.

As you can see from the example in **Figure 3**, traditional, rules-based AI techniques are often difficult to scale. This is where machine learning comes in. Machine learning (ML) is a subset of artificial intelligence that finds patterns in past data and learns from experience to act on new data. ML allows computers to make predictions without being explicitly programmed with logic rules. Thus, you can use ML when you have a problem that's difficult (or impossible) to solve with rules-based programming. You can think of ML as "programming the unprogrammable."

To solve the "bread" vs. "not bread" problem with ML, you provide examples of bread and examples of not bread (as seen in **Figure 4**) instead of implementing a long, complicated series of **if** statements. You pass these examples to an algorithm, which finds patterns in the data and returns a model that you can then use to predict whether images, which have not yet been seen by the model, are "bread" or "not bread."

Listing 1: Use .NET for Spark to generate a word count

```
using Microsoft.Spark.Sql;

namespace MySparkApp
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create a Spark session.
            SparkSession spark = SparkSession
                .Builder()
                .AppName("word_count_sample")
                .GetOrCreate();

            // Create initial DataFrame.
            DataFrame dataFrame = spark.Read()
                .Text("input.txt");

            // Count words.
            DataFrame words = dataFrame
                .Select(Functions.Split(Functions
                    .Col("value"),
                    " ").Alias("words"))
                .Select(Functions.Explode(Functions
                    .Col("words"))
                .Alias("word"))
                .GroupBy("word")
                .Count()
                .OrderBy(Functions.Col("count").Desc());

            // Show results.
            words.Show();

            // Stop Spark session.
            spark.Stop();
        }
    }
}
```

Figure 5 demonstrates another way to think about AI vs. ML. AI takes in rules and data as input with an expected output of the answers based on those rules. ML, on the other hand, takes in data and answers as the input and outputs the rules that can be used to generalize on new data.

What's ML.NET?

Microsoft released ML.NET, an open-source, cross-platform ML framework for .NET developers, at Build in May 2019. Teams at Microsoft have widely used internal versions of the framework for popular ML-powered features for the last nine years; some examples include Dynamics 365 Fraud Detection, PowerPoint Design Ideas, and Microsoft Defender Antivirus Threat Protection.

ML.NET allows you to stay in the .NET ecosystem to build, train, and consume ML models without requiring a background in ML or data science. ML.NET runs anywhere that .NET runs: Windows, Linux, macOS, on-prem, offline scenarios like WinForms or WPF desktop apps, or in any cloud, such as Azure. You can use ML.NET for a variety of scenarios, as described in **Table 1**.

ML.NET uses automated machine learning, or AutoML, to automate the process of building and training an ML model to find the best model based on the scenario and data provided. You can use ML.NET's AutoML via the AutoML.NET API or ML.NET tooling, which includes Model Builder in Visual Studio and the cross-platform ML.NET CLI, as seen in **Figure 6**. In addition to training the best model, ML.NET tooling also generates the files and C# code necessary for model consumption in the end-user .NET application, which can be any .NET app (desktop, Web, console, etc.). All AutoML scenarios offer a local training option, and image classification also allows you to take advantage of the cloud and train using Azure ML from Model Builder.

You can learn more about ML.NET in Microsoft Docs at https://aka.ms/mlnetdocs.

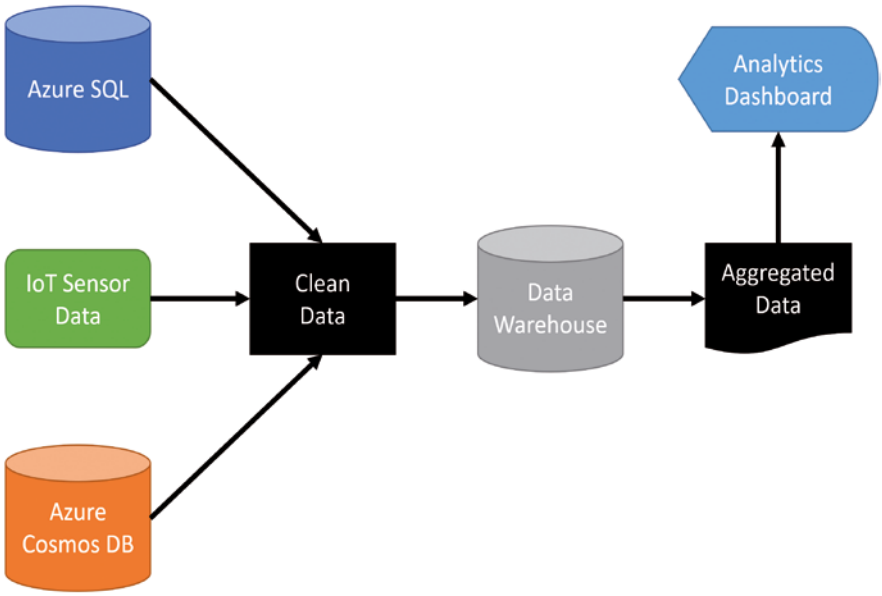


Figure 1: The big data process

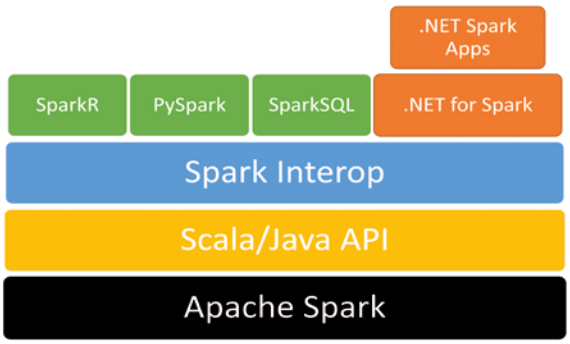


Figure 2: Architecture for .NET for Spark

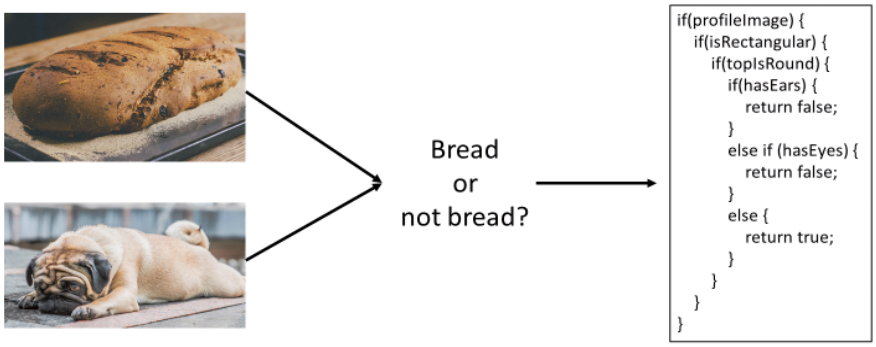


Figure 3: Determining "bread or not bread?" with AI if statements

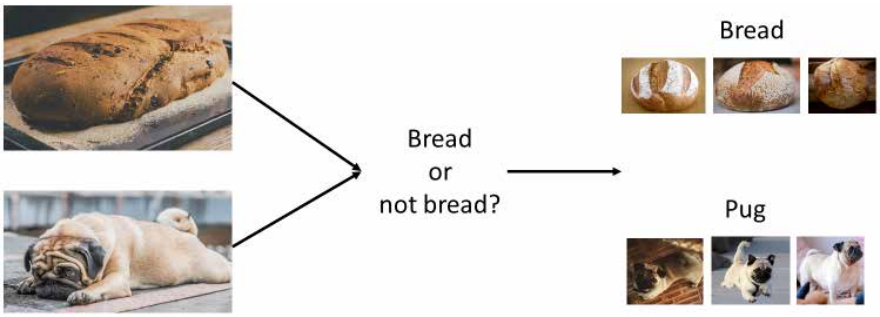


Figure 4: Determining "bread or not bread?" with ML

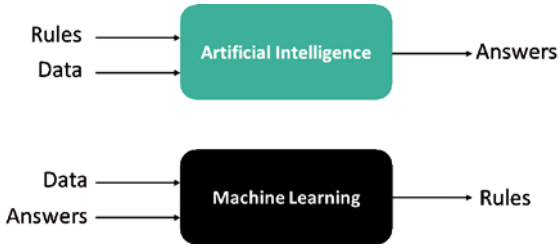


Figure 5: Artificial intelligence compared to machine learning

Combining ML and Big Data with ML.NET and Spark for .NET

Big data and ML go well together. Let's build a pipeline that uses both Spark for .NET and ML.NET to showcase how big data and ML work together. Markdown is a popular language for writing documentation and creating static websites that uses a less complicated syntax than HTML but provides more control over formatting than plain text. This is an excerpt from an example markdown file from the publicly available .NET documentation repo:

```
---
title: Welcome to .NET
description: Getting started with the .NET
family of technologies.
ms.date: 12/03/2019
ms.custom: "updateeachrelease"
---
# Welcome to .NET

See
[Get started with .NET Core]
(core/get-started.md)
to learn how to create .NET Core apps.

Build many types of apps with .NET, such as
cloud ,IoT, and games using free cross-
platform tools...
```

The section between the dashes is called **front matter** and provides metadata about the document using YAML. The section starting with a hash (#) is a top-level heading. Two hashes (##) indicate a sub-heading. The "Get started with .NET Core" is a link, followed by text.

The goal is to process a large collection of documents, add metadata such as word count and estimated reading time, and automatically group similar articles together.

Here is the pipeline you'll build:

1. Generate an input file from the markdown documents for Spark to process as part of the "data preparation/cleaning" step.
2. Use Spark for .NET to perform the following operations:
   - Build a word count for each document.
   - Estimate reading time for each document.
   - Create a list of top 20 words for each document based on "TF-IDF" or "term frequency/inverse document frequency" (this will be explained later).
3. Pass the processed data to ML.NET, which will use an approach called **k-means clustering** to automatically group the documents into categories.
4. Output the categorized list of metadata along with a sample-rollup showing the chosen categories and titles that fall underneath them.

The first step is to pull down a document repository and the reference application. You can use any repository or folder structure that contains markdown files. The examples used for this article are from the .NET documentation repository that is available to clone at https://aka.ms/dot-net-docs.

After you prepare your local environment for .NET and Spark, you can pull down the project from https://aka.ms/spark-ml-example.

The solution folder contains a batch command (provided in the repo) that you can use to run all the steps.

Process the Markdown

The **DocRepoParser** project recursively iterates through subfolders in a repository to gather metadata about various documents. The **Common** project contains several helper classes.

Task	Example scenarios
Classification (text-based)	Categorize e-mail messages as spam or not spam or classify survey comments into different groups based on the content.
Regression	Predicting the price of a used car based on its make, model, and mileage or predicting sales of products based on advertising budgets.
Forecasting	Predicting future product sales based on past sales or weather forecasting.
Anomaly detection	Detecting spikes in product sales over time or detecting power outages.
Ranking	Predicting the best order to display search engine results or ranking items for a user's newsfeed.
Clustering	Segmenting customers.
Recommendation	Recommending movies to a user based on their previously watched movies or recommending products that are frequently bought together.
Image classification	Categorizing images of machine parts.
Object detection	Detecting license plates on images of cars.

Table 1: ML.NET machine learning tasks and scenarios

For example, **FilesHelper** is used for all file I/O. It keeps track of the location to store files and filenames and provides services such as reading files to other projects. The constructor expects a tag (a number that uniquely identifies a workflow) and the path to the repo or top-level folder that contains the documentation. By default, it creates a folder under the user's local application data folder. This can be overridden if necessary.

MarkdownParser leverages a library called **Microsoft.Toolkit.Parsers** to parse the markdown. The library has two tasks: first, it must extract titles and subtitles, and second, it must extract words. The markdown file is exposed as "blocks" representing headers, links, and other markdown features. Blocks, in turn, contain "inlines" that host the text. For example, this code parses a **TableBlock** by iterating over rows and cells to find inlines:

```
case TableBlock table:
    table.Rows.SelectMany(r => r.Cells)
        .SelectMany(c => c.Inlines)
        .ForEach(i => candidate =
            RecurseInline(i, candidate, words,
                titles));
    break;
```

This code extracts the text part of an inline hyperlink:

```
case HyperlinkInline hyper:
    if (!string.IsNullOrWhiteSpace(hyper.Text))
    {
        words.Append(hyper.Text.ExtractWords());
    }
    break;
```

The result is a comma-delimited (CSV) file that looks like **Figure 7**.

The first step simply prepares the data for processing. The next step uses a Spark for .NET job to determine word count, reading time, and the top 20 terms for each document.

Build the .NET for Spark Job

The **SparkWordsProcessor** project hosts the .NET for the Spark job to run. Although the app is a console project, it requires Spark to run. The **runjob.cmd** batch command submits the job to run on a properly configured Windows computer. The pattern for a typical job is to create a session or "app," perform some logic, then stop the session.

```
var spark = SparkSession.Builder()
    .AppName(nameof(SparkWordsProcessor))
    .GetOrCreate();
RunJob();
spark.Stop();
```

The file from the previous step is easily read by passing its path to the Spark session:

```
var docs = spark.Read().HasHeader()
    .Csv(filesHelper.TempDataFile);
docs.CreateOrReplaceTempView(nameof(docs));
var totalDocs = docs.Count();
```

The **docs** variable resolves to a **DataFrame**. A data frame is essentially a table with a set of columns and a common interface to interact with data regardless of the underlying source. It's possible to reference a data frame from other data frames. SparkSQL can also be used to query data frames. You must create a temporary view that provides an alias to the data frame to reference it from SQL. The **CreateOrReplaceTempView** method makes it possible to select rows from the data frame like this:

```
SELECT * FROM docs
```

The **totalDocs** variable retrieves a count of all rows in the document. Spark provides a function called **Split** to break a string into an array. The **Explode** function turns each array item into a row:

```
var words = docs
    .Select(
        fileCol,
        Functions.Split(
            nameof(FileDataParse.Words)
                .AsColumn(), " ")
        .Alias(wordList))
    .Select(
        fileCol,
        Functions.Explode(
            wordList.AsColumn())
        .Alias(word));
```

The query generates one row per word or term. This data frame is the basis for generating the **term frequency** (TF), or the count of each word per document:

```
var termFrequency = words
    .GroupBy(fileCol, Functions.Lower(
        word.AsColumn()).Alias(word))
    .Count()
    .OrderBy(fileCol, count.AsColumn().Desc());
```

Spark has built-in models that can determine "term frequency/inverse document frequency." For this example, you'll determine term frequency manually to demonstrate how it is calculated. Terms occur with a specific frequency in each document. A document about wizards might have a high count of

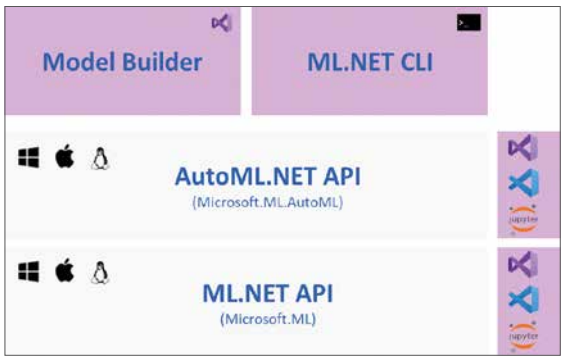


Figure 6: ML.NET tooling is built on top of the AutoML.NET API, which is on top of the ML.NET API

```
"File","Title","Subtitle1","Subtitle2","Subtitle3","Subtitle4","Subtitle5","Words"
"1-CODE_OF_CONDUCT.md","Code of Conduct","","","","","","Code of Conduct This project has adopted
the code of conduct defined by the Contributor Covenant to clarify expected behavior in our
community For more information see the NET Foundation Code of Conduct"
"1-CONTRIBUTING.md","Contributing","","","","","","Contributing Thank you for your interest in
contributing to the NET documentation We have moved our guidelines into a site-wide contribution
guide To see the guidance visit the Microsoft Docs contributor guide"
```

Figure 7: The generated CSV file

the term "wizards." The same document probably has a high count of the words "the" and "is" as well. To us, it's obvious that the term "wizard" is more important and provides more context. Spark, on the other hand, must be trained to recognize important terms. To determine what's truly important, we'll summarize the **document frequency**, or the count of how many times a word appears across all the documents in the repo. This is a "group by distinct occurrences":

```
var documentFrequency = words
    .GroupBy(Functions.Lower(word.AsColumn())
        .Alias(word))
    .Agg(Functions.CountDistinct(fileCol)
        .Alias(docFrequency));
```

Now it's time for math. A special equation computes what's known as the **inverse document frequency**, or IDF. The natural logarithm of total documents (plus one) is input into the equation and divided by the document frequency (plus one) for the word:

```
static double CalculateIdf(
int docFrequency, int totalDocuments) =>
Math.Log(totalDocuments + 1) /
    (docFrequency + 1);
```

Words that appear across all documents are assigned a lower value than words that appear less frequently. For example, given 1000 documents, a term that appears in every document has an IDF of 0.003 compared to a term that only appears in a few documents (~1). Spark supports user-defined functions which you can register like this:

```
spark.Udf().Register<int, int, double>(
    nameof(CalculateIdf), CalculateIdf);
```

Next, you can use the function to compute the IDF of all words in the data frame:

```
var idfPrep = documentFrequency.Select(
    word.AsColumn(),
    docFrequency.AsColumn())
    .WithColumn(total, Functions.Lit(totalDocs))
    .WithColumn(
    inverseDocFrequency,
    Functions.CallUDF(
        nameof(CalculateIdf),
        docFrequency.AsColumn(),
        total.AsColumn()));
```

Using the document frequency data frame, two columns are added. The first is the literal total number of documents, and the second is a call to your UDF to compute the IDF. There's just one more step to determine the "important words." Using TF-IDF, the important words are ones that don't appear often across all documents but do appear often in the current document. This is simply a product of the IDF and the TF. Consider the case of "is" with an IDF of 0.002 and a frequency of 50 in the document, versus "wizard" with an IDF of 1 and a frequency of 10. The TF-IDF for "is" computes to 0.1, compared to 10 for the term "wizard." This gives Spark a better notion of importance than just the raw word count.

So far, you've used code to define the data frames. Let's try some SparkSQL. To compute the TF-IDF, you join the document frequency data frame with the inverse document frequency data frame and create a new column named **termFreq_inverseDocFreq**. Here is the SparkSQL:

```
var idfJoin = spark.Sql($"SELECT t.File, d.word,
d.{docFrequency}, d.{inverseDocFrequency},
t.count, d.{inverseDocFrequency} * t.count as
{termFreq_inverseDocFreq} from
{nameof(documentFrequency)} d inner join
{nameof(termFrequency)} t on t.word = d.word");
```

Explore the code to see how the final steps are implemented. These steps include:

1. Create a data frame with a total word count per document.
2. Sort and partition the TF-IDF data to select the top 20 words per document.
3. Remove "stop words" that may have slipped through the cracks.
4. Use a UDF to divide total words by 225 to estimate the reading time (assuming 225 words per minute).

All of the steps described so far provide a template or definition for Spark. Like LINQ queries, the actual processing doesn't happen until the results are materialized (such as when the total document count was computed). The final step calls **Collect** to process and return the results and write them to another CSV. You can then use the new file as input for the ML model. A portion of the file is shown in **Figure 8**.

Spark for .NET enabled you to query and shape the data. You built multiple data frames over the same data source and then joined them to gain insights about important terms, word count, and read time. The next step is to apply ML to automatically generate categories.

Predict Categories

The last step is to categorize the documents. The **DocMLCategorization** project includes the **Microsoft.ML** package for ML.NET. Although Spark works with data frames, data views provide a similar concept in ML.NET.

This example uses a separate project for ML.NET so that the model can be trained as an independent step. For many scenarios, it's possible to reference ML.NET directly from your .NET for Spark project and perform ML as part of the same job.

First, you must tag the class so that ML.NET knows which columns in the source data map to properties in the class. The **FileData** class uses the **LoadColumn** annotation like this:

```
[LoadColumn(0)]
public string File { get; set; }
```

```
"File","Title","Subtitle1","Subtitle2","Subtitle3","Subtitle4","Subtitle5","Top20Words","WordCount","ReadingTime","Words"
"CODE_OF_CONDUCT.md","Code of Conduct","","","","","","conduct covenant contributor adopted clarify community expected
project foundation behavior code defined more information",33,"< 1 minute",""
"CONTRIBUTING.md","Contributing","","","","","","contributing contribution thank contributor visit interest docs guidance
site guidelines wide guide documentation microsoft",33,"< 1 minute",""
```

Figure 8: Processed metadata that's ready for ML training

Listing 2: Transforming text into features for machine learning

```
var pipeline = context.Transforms
    .Text.FeaturizeText(nameof(FileData.Title).Featurized(),
        nameof(FileData.Title))
    .Append(context.Transforms.Text.FeaturizeText(
        nameof(FileData.Subtitle1).Featurized(),
        nameof(FileData.Subtitle1)))
    .Append(context.Transforms.Text.FeaturizeText(
        nameof(FileData.Subtitle2).Featurized(),
        nameof(FileData.Subtitle2)))
    .Append(context.Transforms.Text.FeaturizeText(
        nameof(FileData.Subtitle3).Featurized(),
        nameof(FileData.Subtitle3)))
    .Append(context.Transforms.Text.FeaturizeText(
        nameof(FileData.Subtitle4).Featurized(),
        nameof(FileData.Subtitle4)))
    .Append(context.Transforms.Text.FeaturizeText(
        nameof(FileData.Subtitle5).Featurized(),
        nameof(FileData.Subtitle5)))
    .Append(context.Transforms.Text.FeaturizeText(
        nameof(FileData.Top20Words).Featurized(),
        nameof(FileData.Top20Words)))
    .Append(context.Transforms.Concatenate(
        features,
        nameof(FileData.Title).Featurized(),
        nameof(FileData.Subtitle1).Featurized(),
        nameof(FileData.Subtitle2).Featurized(),
        nameof(FileData.Subtitle3).Featurized(),
        nameof(FileData.Subtitle4).Featurized(),
        nameof(FileData.Subtitle5).Featurized(),
        nameof(FileData.Top20Words).Featurized()));
```

```
[LoadColumn(1)]
public string Title { get; set; }
```

You can then create a context for the model and load the data view from the file that was generated in the previous step:

```
var context = new MLContext(seed: 0);

var dataToTrain = context.Data
    .LoadFromTextFile<FileData>(
        path: filesHelper.ModelTrainingFile,
        hasHeader: true,
        allowQuoting: true,
        separatorChar: ',');
```

ML algorithms work best with numbers, so the text in the document must be converted to numeric vectors. ML.NET provides the **FeaturizeText** method for this. In one step, the model:

- Detects the language
- Tokenizes the text into individual words or tokens
- Normalizes the text so that variations of words are standardized and cased similarly
- Transforms the terms into consistent numerical values or "feature vectors" that are ready for processing

The code in **Listing 2** transforms columns into features and then creates a single "Features" column with the features combined.

At this point, the data is properly prepared to train the model. The training is **unsupervised**, which means it must infer information with an example. You're not inputting sample categories into the model, so the algorithm must figure out how the data is interrelated by analyzing how features cluster together. You will use the **k-means clustering** algorithm. This algorithm uses the features to compute the "distance" between documents and then "draws" bounds around the grouped documents. The algorithm involves randomization so no two runs will be alike. The main challenge is determining the optimal cluster size for training. Different documentation sets ideally have different optimal category counts, but the algorithm requires you to input the number of categories before training.

The code iterates between two and 20 clusters to determine the optimal size. For each run, it takes the feature data and applies the algorithm or trainer. It then transforms the existing data based on the prediction model. The result is evaluated to determine the average distance of documents in each cluster, and the result with the lowest average distance is selected.

```
var options = new KMeansTrainer.Options
{
    FeatureColumnName = features,
    NumberOfClusters = categories,
};
var clusterPipeline = pipeline.Append(
    context.Clustering.Trainers
    .KMeans(options));
var model = clusterPipeline.Fit(dataToTrain);
var predictions = model.Transform(dataToTrain);
var metrics = context.Clustering.Evaluate(
    predictions);
distances.Add(categories,
    metrics.AverageDistance);
```

After training and evaluation, you can then save the optimal model and use it to make predictions on the data set. An output file is generated along with a summary that shows some metadata about each category and lists the titles underneath. The title is only one of several features, so sometimes it requires looking into the details to make sense of the categories. In local tests, documents such as tutorials end up in one group, API documentation in another, and exceptions in their own group.

The machine learning model is saved as a single zip file. The file can be included in other projects to use with the Prediction Engine to make predictions on new data. For example, you could create a WPF application that allows users to browse to a directory and then loads and uses the trained model to categorize documents without having to train it first.

What's Next?

Spark for .NET is scheduled to GA around the same time as .NET 5. Read the roadmap and plans for upcoming features at https://aka.ms/spark-net-roadmap.

This walkthrough focused on a local development experience. To tap into the real power of big data, you can submit jobs to the cloud. There are a variety of cloud hosts that can accommodate petabytes of data and provide dozens of cores of computing power for your workloads. Azure Synapse Analytics is an Azure service that is designed to host large amounts of data, provide clusters for running big data jobs, and enable interactive exploration through notebooks and chart-based dashboards. To learn how to submit Spark for .NET jobs to Azure Synapse, visit https://aka.ms/spark-net-synapse.

Jeremy Likness and Bri Achtman

CODE

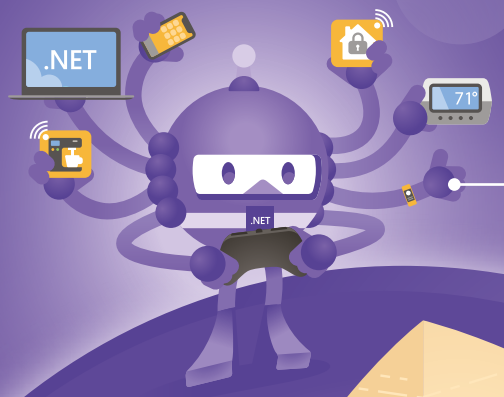
Explore Data Interactively with Notebooks

Batch jobs aren't the only way to explore big data. Synapse Studio Notebooks provide a Web interface to interact with data live.

In notebooks, you can provide code blocks in cells that define or show the contents of data frames, visualize data, and even share comments and narrative. It's a great way to explore data in real-time and gain insights.

Learn more about Synapse Studio notebooks by visiting: https://aka.ms/synapse-notebooks

Build your world with .NET





Any app, any platform

- Web
- Cloud
- Desktop
- Games
- Mobile
- Artificial Intelligence/Machine Learning (AI/ML)
- IoT (Internet of Things)

Scalable application architectures

- **Microservices**
- **Cloud Native**
- **Serverless**

Deployment to any point

Containerization and Hosting
- App Service, DockerHub, Kubernetes, AKS

Application Monitoring
- App Insights

Tools to get it done

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac
- .NET CLI (command line interface)
- Github
- Your favorite editor

Components to get you started

- .NET Libraries
- NuGet
- Identity and Security
- Data

A solid core

Multi-language
- C#
- F#
- VB.NET

Cross-Platform
- Windows
- Linux
- macOS
- iOS
- Android
- more

Performance and Reliability

Open Source

Any app, any platform

Web: Create scalable, high-performance websites and services that run on Windows, macOS, and Linux. https://aka.ms/startweb
Cloud: Create powerful, intelligent cloud apps with .NET using a fully managed platform. https://aka.ms/startcloud
Desktop: Create beautiful and compelling native desktop apps on Windows and macOS. https://aka.ms/startdesktop
Mobile: Use a single code base to build native mobile apps for iOS, Android, and Windows. https://aka.ms/startmobile
Games: Develop 2D and 3D games for the most popular desktops, phones, and consoles. https://aka.ms/startgaming
AI/ML (Artificial Intelligence and Machine Learning): Infuse AI and Machine Learning into your .NET apps such as vision algorithms, speech processing, predictive models, and more. https://aka.ms/startai
IoT (Internet of Things): Make IoT apps with native support for the Raspberry Pi and other single-board computers. https://aka.ms/startiothings

Scalable application architectures

Microservices: Microservices are highly scalable, resilient, and composable units of deployment for modern applications and .NET is a perfect platform for creating them. Get started learning and building them. https://aka.ms/startmicroservices
Cloud Native: Learn about the essential design patterns that are useful for building reliable, scalable, secure applications that are born in the cloud. https://aka.ms/startpatterns
Guides: Get started with modern .NET application architectures for building any type of application whether it's for web, mobile, desktop or the cloud. https://aka.ms/startarch

Deployment to any point

Containerization and Hosting: Containers simplify deployment and testing by bundling a service and its dependencies into a single unit, which is then run in an isolated environment. Orchestrators such as Kubernetes automate deployment, scaling, and management of containerized applications.

- Docker container images for .NET on Linux and Windows are available on Docker Hub. https://aka.ms/startdocker
- Simplify the deployment, management, and operations of Kubernetes with AKS. https://aka.ms/startaks
- Quickly create powerful cloud apps with or without containers using a fully managed platform with App Service. https://aka.ms/startappservice

Application Monitoring: Get actionable insights through application performance management and instant analytics with App Insights. https://aka.ms/startappinsights

Tools to get it done

Visual Studio Family: Visual Studio provides best-in-class tools for any developer on any operating system. From advanced IDEs and editors to agile tools, CI/CD, monitoring, and learning, they have you covered. Get started for free. www.visualstudio.com
.NET CLI: The .NET CLI (command-line-interface) is a command line tool you can use with any editor to build many types of .NET apps. Get it with the .NET SDK. dot.net/get-dotnet5

Components to get you started

.NET Libraries: .NET provides thousands of built-in APIs in base class libraries that help you build cross-platform code that can do just about anything, making it simple to share libraries across any application. https://aka.ms/netstandardapis
NuGet: From data components to UI controls and thousands more reusable libraries, the NuGet package manager helps you create .NET apps faster. www.nuget.org

A solid core

Multi-language: You can write your .NET apps in multiple programming languages.

- **C# (c-sharp**) is a simple, modern, object-oriented, and type-safe programming language with roots in the C family of languages, making it immediately familiar to C, C++, Java, and JavaScript programmers. https://aka.ms/start-csharp
- **F# (f-sharp)** is a functional programming language that also includes object-oriented and imperative programming. https://aka.ms/start-fsharp
- **Visual Basic** is an approachable language with a simple syntax for building type-safe, object-oriented programs. https://aka.ms/start-vb

Cross-platform: Your .NET apps will run on a variety of operating systems, depending on the app you're building. For instance, web apps can be hosted on Windows, macOS, or multiple distros of Linux. Or build mobile apps for Android and iOS all with .NET.
Performance: .NET is fast. Really fast! .NET performs faster than any other popular framework on TechEmpower benchmarks. From providing safer, faster memory access with Span<T> to a faster just-in-time compiler, great performance is at the core of .NET. https://aka.ms/dotnetperf
Open Source: .NET is open source under the .NET Foundation.
The .NET Foundation is an independent organization to foster open development and collaboration around the .NET ecosystem.
www.dotnetfoundation.org

.NET

F# 5: A New Era of Functional Programming with .NET

On the F# team at Microsoft, we're constantly improving the F# language to empower developers to do functional programming on .NET. Over the previous four releases, from 2017 until now, we've been on a long journey to make F# awesome on .NET Core. We've revamped the F# compiler and core library to run cross-platform, added support for Span<T> and low-level, cross-platform programming, and added the ability to preview language features that can ship with .NET preview releases.



Phillip Carter
phillipcarter.dev/
twitter.com/_cartermp

Phillip Carter is a Senior Program Manager on the .NET team at Microsoft, focusing on .NET languages and tools. He works on all things F#: language design, the compiler, the core library, and tooling. He also helps out with the C# compiler and .NET project tooling in Visual Studio.

With the .NET 5 release, we're releasing F# 5, the next major version of the F# language. But F# 5 isn't just a bundle of features that comes along for the ride with .NET 5. F# 5 marks the end of the current era—bringing up support for .NET Core—and the beginning of a new one. With F# 5, we're considering our journey to bring F# to .NET Core mostly complete. With F# 5, our focus shifts from .NET Core to three major areas:

- Interactive programming
- Making analytical-oriented programming convenient and fun
- Great fundamentals and performance for functional programming on .NET

In this article, I'll go through the F# language and tooling features we've implemented for F# 5 and explain how they align with our goals.

F# 5 Makes Interactive Programming a Joy

F# has a long history of being interactive. In fact, when F# 1.0 was developed, a tool called F# Interactive (FSI) was developed for the eventual release of F# 1.0 in 2006. This coincided with the first tooling integration into Visual Studio. FSI was used quite heavily in the initial marketing of F# (as shown in **Figure 1**) to demonstrate iterative and interactive development of Windows Forms applications, graphics scenes, and games on Windows.

The core experiences of FSI have largely remained the same in F# 5. These include:

- The ability to reference and call into assemblies on your computer
- The ability to load other F# scripts to execute as a collection of scripts
- Integration with Visual Studio
- The ability to customize output

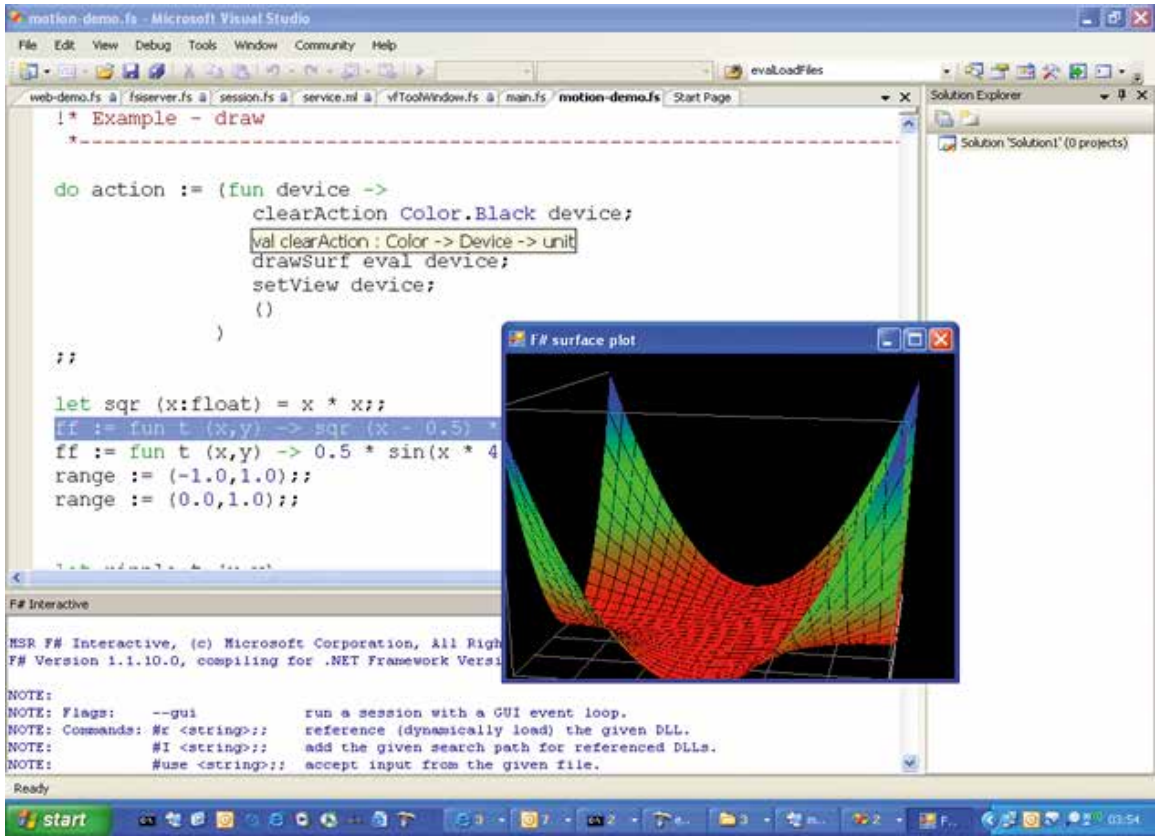


Figure 1: Initial prototype of F# Interactive in Visual Studio 2005

However, as F# and the .NET ecosystem moved from assemblies on a computer to packages installed via a package manager, many F# developers using FSI for various tasks found themselves annoyed by having to manually download a package and reference its assemblies manually. Additionally, as .NET's reach extended beyond Windows, developers on macOS or Linux found themselves missing features and relying on a Mono installation to use FSI in their environments.

Introducing Package Management Support in FSI

Using a package in an F# script has long been a source of frustration for F# programmers. They typically downloaded packages themselves and referenced assemblies in the path to the package manually. A smaller set of F# programmers used the Paket package manager and generated a "load script"—a feature in Paket that generates an F# script file with references to all the assemblies in the packages you want to reference—and loads this script into their working F# scripts. However, because Paket is an alternative to NuGet instead of a default tool, most F# programmers don't use it.

Now with F# 5, you can simply reference any NuGet package in an F# script. FSI restores this package with NuGet and automatically references all assemblies in the package. Here's an example:

```
#r "nuget: Newtonsoft.Json"

open Newtonsoft.Json

let o = {| X = 2; Y = "Hello" |}
printfn "%s" (JsonConvert.SerializeObject o)
```

When you execute the code in that snippet, you'll see the following output:

```
{"X":2,"Y":"Hello"}
val o : {| X: int; Y: string |} = { X = 2
                                    Y = "Hello" }
val it : unit = ()
```

The package management feature can handle just about anything you want to throw at it. It supports packages with native dependencies like ML.NET or Flips. It also supports packages like FParsec, which previously required that each assembly in the package is referenced in a specific order in FSI.

Introducing dotnet FSI

The second major frustration for F# programmers using FSI is that it was missing in .NET Core for a long time. Microsoft released an initial version of FSI for .NET Core with .NET Core 3.0, but it was only useful for F# scripts that didn't incorporate any dependencies. Now, in conjunction with package management, you can use FSI for all the same tasks on macOS or Linux as you would on Windows (except for launching WinForms and WPF apps, for obvious reasons). This is done with a single command: **dotnet fsi**.

Introducing F# Support in Jupyter Notebooks

There's no question that package management and making FSI available everywhere makes F# better for interactive programming. But Microsoft felt that we could do more than just that. Interactive programming has exploded in recent years in the Python community, thanks in large part to Jupyter Notebooks. The F# community had built initial support for F# in Jupyter many years ago, so we worked with its current maintainer to learn about what a good experience for Jupyter meant and built it.

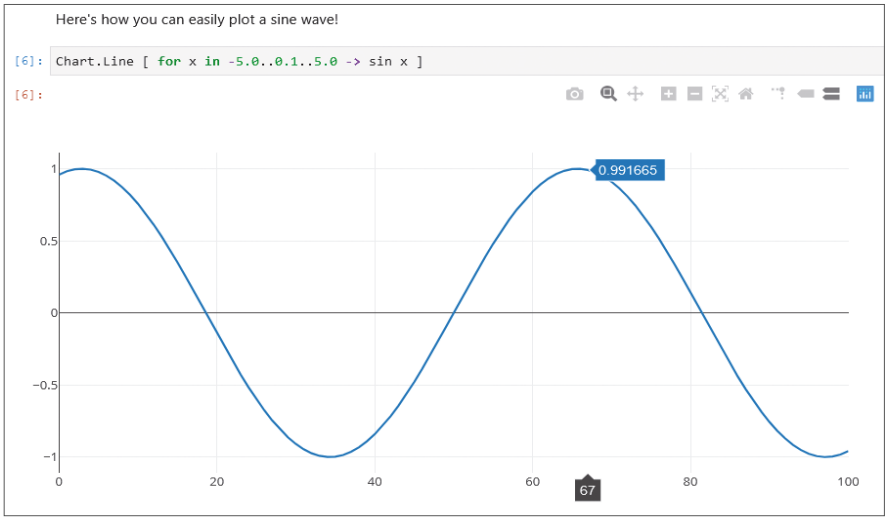


Figure 2: Charting in Jupyter notebooks with F#

Now, with F# 5, you can pull in packages, inspect data, and chart the results of your experimentation in a sharable, cross-platform notebook that anyone can read and adjust, as shown in **Figure 2**.

Another reason why we're very excited about F# support in Jupyter Notebooks is that the notebooks are easy to share with other people. Juputer Notebooks render as markdown documents in GitHub and other environments. Not only are they a programming tool, but they produce a document that can be used to instruct others how to perform tasks, share findings, learn a library, or even learn F# itself!

Introducing F# Support in Visual Studio Code Notebooks

F# support in Jupyter Notebooks brings interactivity to a whole new level. But Jupyter isn't the only way to program for a notebook. Visual Studio Code is also bringing notebook programming into the fold, with all the power of a language service that you would expect to find when editing code in a normal file. With F# support in Visual Studio Code Notebooks, you can enjoy language service integration when building a notebook, as shown in **Figure 3**.

Another benefit of Visual Studio Code notebooks is its file format, which is designed to be human-readable and easy to diff in source control. It supports importing Jupyter Notebooks and exporting Visual Studio Code notebooks as Jupyter Notebooks, as you can see in **Figure 4**.

You can do many things with F# in Visual Studio Code and Jupyter Notebooks, and we're looking to expand the capabilities beyond what's been described so far. Our roadmap includes integration with various other tools, more cohesive data visualization, and data interop with Python.

F# 5 Lays More Foundations for Analytical Programming

A paradigm of growing importance in the age of ubiquitous machine learning and data science is what I like to call "analytical programming." This paradigm isn't exactly new, although there

are new techniques, libraries, and frameworks coming out every day to further advance the space. Analytical programming is all about analyzing and manipulating data, usually applying numerical techniques to deliver insights. This ranges from importing a CSV and computing a linear regression on the data to the most advanced and compute-intensive neural networks coming out of AI research institutions.

F# 5 represents the beginning of our foray into this space. The team at Microsoft thinks that F# is already great for manipulating data, as countless F# users have demonstrated by using F# for exactly that purpose. F# also has great support for numeric programming with some built-in types and functions and a syntax that's approachable and succinct. So we kept that in mind and identified some more areas to improve.

Consistent Behavior with Slices

A very common operation performed in analytical programming is taking a slice of a data structure, particularly arrays. F# slices used to behave inconsistently, with some out-of-bounds behavior resulting in a runtime exception and others resulting in an empty slice. We've changed all slices for F# intrinsic types—arrays, lists, strings, 3D arrays, and 4D arrays—to return an empty slice for any slice you might specify that couldn't possibly exist:

```
let l = [ 1..10 ]
let a = [| 1..10 |]
let s = "hello!"

// Before: empty list
// F# 5: same
let emptyList = l.[-2..(-1)]

// Before: would throw exception
// F# 5: empty array
let emptyArray = a.[-2..(-1)]

// Before: would throw exception
// F# 5: empty string
let emptyString = s.[-2..(-1)]
```

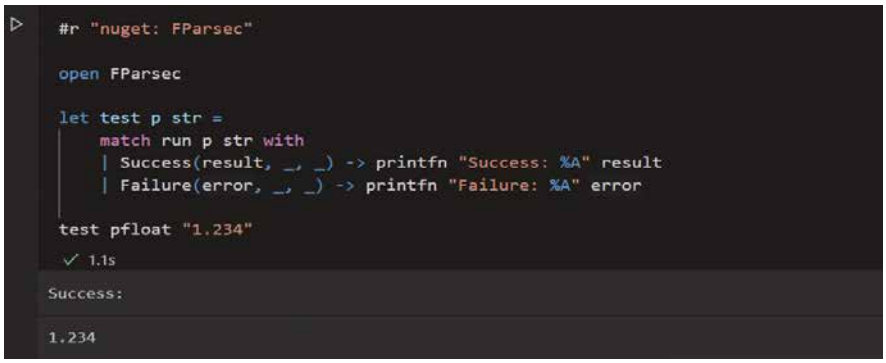


Figure 3: Installing a package and using it in Visual Studio Code Notebooks

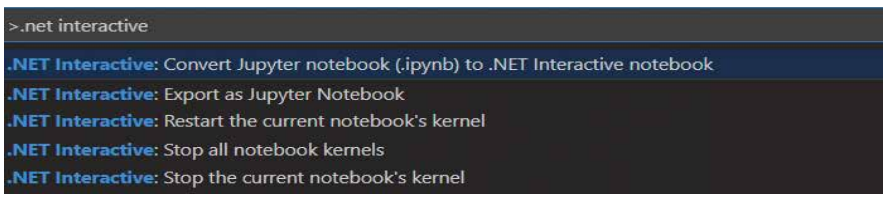


Figure 4: Converting a Jupyter Notebook in Visual Studio Code

The reasoning for this is largely because in F#, empty slices compose with nonempty slices. An empty string can be added to a nonempty string, empty arrays can be appended to nonempty arrays, etc. This change is non-breaking and allows for predictability in behavior.

Fixed Index Slicing for 3D and 4D Arrays

F# has built-in support for 3D and 4D arrays. These array types have always supported slicing and indexing, but never slicing based on a fixed index. With F# 5, this is now possible:

```
// First, create a 3D array
// with values from 0 to 7
let dim = 2
let m = Array3D.zeroCreate<int> dim dim dim

let mutable cnt = 0

for z in 0..dim-1 do
    for y in 0..dim-1 do
        for x in 0..dim-1 do
            m.[x,y,z] <- cnt
            cnt <- cnt + 1

// Now let's get the [4;5] slice!
m.[*, 0, 1]
```

This helps complete the picture for slicing scenarios with 3D and 4D arrays.

Preview: Reverse Indexes

Microsoft is also introducing the ability to use reverse indexes, which can be used with slices, as a preview in F# 5. To use it, simply place *<LangVersion>preview</LangVersion>* in your project file.

```
let xs = [1..10]

// Get element 1 from the end:
xs.[^1]

// Old way to get the last two elements
let lastTwoOldStyle = xs.[(xs.Length-2)..]

// New way to get the last two elements
let lastTwoNewStyle = xs.[^1..]

lastTwoOldStyle = lastTwoNewStyle // true
```

You can also define your own members via an F# type extension to augment these types to support F# slicing and reverse indexes. The following example does so with the **Span<'T>** type:

```
open System

type Span<'T> with
    member sp.GetSlice(startIdx, endIdx) =
        let s = defaultArg startIdx 0
        let e = defaultArg endIdx sp.Length
        sp.Slice(s, e - s)

    member sp.GetReverseIndex(_, offset: int) =
        sp.Length - offset

let sp = [| 1; 2; 3; 4; 5 |].AsSpan()
sp.[..^2] // [|1; 2; 3|]
```

F# intrinsic types have reverse indexes built in. In a future release of F#, we'll also support full interop with **System.Index** and **System.Range**, at which point, the feature will no longer be in preview.

Enhanced Code Quotations

F# Code Quotations are a metaprogramming feature that allows you to manipulate the structure of F# code and evaluate it in an environment of your choosing. This capability is essential for using F# as a model construction language for machine learning tasks, where the AI model may run on different hardware, such as a GPU. A critical piece missing in this puzzle has been the ability to faithfully represent F# type constraint information, such as those used in generic arithmetic, in the F# quotation so that an evaluator can know to apply those constraints in the environment it's evaluating in.

Starting with F# 5, constraints are now retained in code quotations, unlocking the ability for certain libraries such as DiffSharp to use this part of the F# type system to its advantage. A simple way to demonstrate this is the following code:

```
open FSharp.Linq.RuntimeHelpers

let eval q =
    LeafExpressionConverter
        .EvaluateQuotation q

let inline negate x = -x

// Crucially, 'negate' has
// the following signature:
//
// val inline negate:
//   x: ^a -> ^a
//        when  ^a:
//          (static member ( ~- ):  ^a ->  ^a)
//
// This constraint is critical to F# type safety
// and is now retained in quotations.
<@ negate 1.0 @>  |> eval
```

The use of an arithmetic operator implies a type constraint such that all types passed to *negate* must support the '–' operator. This code fails at runtime because the code quotation doesn't retain this constraint information, so evaluating it throws an exception.

Code quotations are the foundation for some more R&D-heavy work being done to use F# as a language for creating AI models, and so the ability to retain type constraint information in them helps make F# a compelling language for programmers in this space who seek a little more type safety in their lives.

F# 5 Has Great Fundamentals

F# 5 may be about making interactivity and analytical programming better, but at its core, F# 5 is still about making everyday coding in F# a joy. F# 5 includes several new features that both app developers and library authors can enjoy.

Support for nameof

First up is a feature that C# developers have come to love: **nameof**. The **nameof** operator takes an F# symbol as input and produces a string at compile-time that represents that symbol. It supports just about all F# constructs. The nameof operator is often used for logging diagnostics in a running application.

```
#r "nuget: FSharp.SystemTextJson"

open System.Text.Json
open System.Text.Json.Serialization
open System.Runtime.CompilerServices


module M =
    let f x = nameof x

printfn "%s" (M.f 12)
printfn "%s" (nameof M)
printfn "%s" (nameof M.f)

/// Simplified version of EventStore's API
type RecordedEvent =
    { EventType: string
      Data: byte[] }

/// My concrete type:
type MyEvent =
    | AData of int
    | BData of string

// use 'nameof' instead of the string literal in
// the match expression
let deserialize (e: RecordedEvent) : MyEvent =
    match e.EventType with
    | nameof AData ->
        JsonSerializer.Deserialize<AData> e.Data
        |> AData
    | nameof BData ->
        JsonSerializer.Deserialize<BData> e.Data
        |> BData
    | t -> failwithf "Invalid EventType: %s" t
```

Interpolated Strings

Next is a feature seen in languages such as C# and JavaScript: Interpolated Strings. Interpolated strings allow you to create interpolations or holes in a string that you can fill in with any F# expression. F# interpolated strings support typed interpolations synonymous with the same format specifies in **sprintf** and **printf** strings formats. F# interpolated strings also support triple-quotes strings. Just like in C#, all symbols in an F# interpolation are navigable, able to be renamed, and so on.

```
// Basic interpolated string
let name = "Phillip"
let age = 29
let message = $"Name: {name}, Age: {age}"

// Typed interpolation
// '%s' requires the interpolation to be a string
// '%d' requires the interpolation to be an int
let message2 = $"Name: %s{name}, Age: %d{age}"

// Verbatim interpolated strings
// Note the string quotes allowed inside the
// interpolated string
let messageJson = $"""
"Name": "{name}",
"Age": {age}"""
```

Additionally, you can write multiple expressions inside interpolated strings, producing a different value for the interpolated expression based on an input to the function. This is a more of a niche use of the feature, but because any interpolation can be a valid F# expression, it allows for a great deal of flexibility.

Open Type Declarations

F# has always allowed you to open a namespace or a module to expose its public constructs. Now, with F# 5, you can open any type to expose static constructs like static methods, static fields, static properties, and so on. F# union and records can also be opened. You can also open a generic type at a specific type instantiation.

```
open type System.Math

let x = Min(1.0, 2.0)

module M =
    type DU = A | B | C

    let someOtherFunction x = x + 1

// Open only the type inside the module
open type M.DU

printfn "%A" A
```

Enhanced Computation Expressions

Computation expressions are a well-loved set of features that allow library authors to write expressive code. For those versed in category theory, they are also the formal way to write Monadic and Monoidal computations. F# 5 extends computation expressions with two new features:

- Applicative forms for computation expressions via **let!..and!** keywords
- Proper support for overloading Custom Operations

"Applicative forms for computation expressions" is a bit of a mouthful. I'll avoid diving into category theory and instead work through an example:

```
// First, define a 'zip' function
module Result =
    let zip x1 x2 =
        match x1,x2 with
        | Ok x1res, Ok x2res ->
            Ok (x1res, x2res)
        | Error e, _ -> Error e
        | _, Error e -> Error e

// Next, define a builder with 'MergeSources'
// and 'BindReturn'
type ResultBuilder() =
    member _.MergeSources(t1: Result<'T,'U>,
                          t2: Result<'T1,'U>) =
        Result.zip t1 t2
    member _.BindReturn(x: Result<'T,'U>, f) =
        Result.map f x

let result = ResultBuilder()

let run r1 r2 r3 =
    // And here is our applicative!
    let res1: Result<int, string> =
        result {
            let! a = r1
            and! b = r2
            and! c = r3
            return a + b - c
        }

    match res1 with
    | Ok x ->
        printfn "%s is: %d" (nameof res1) x
    | Error e ->
        printfn "%s is: %s" (nameof res1) e
```

Listing 1: Computation Expressions can overload custom operations

```
type InputKind =
    | Text of placeholder:string option
    | Password of placeholder: string option

type InputOptions =
  { Label: string option
    Kind: InputKind
    Validators: (string -> bool) array }

type InputBuilder() =
    member t.Yield(_) =
      { Label = None
        Kind = Text None
        Validators = [||] }

    [<CustomOperation("text")>]
    member this.Text(io,?placeholder) =
        { io with Kind = Text placeholder }

    [<CustomOperation("password")>]
    member this.Password(io,?placeholder) =
        { io with Kind = Password placeholder }

    [<CustomOperation("label")>]
    member this.Label(io,label) =
        { io with Label = Some label }

    [<CustomOperation("with_validators")>]
    member this.Validators(io, [<System.ParamArray>] validators) =
        { io with Validators = validators }

let input = InputBuilder()

let name =
    input {
    label "Name"
    text
    with_validators
        (String.IsNullOrWhiteSpace >> not)
    }

let email =
    input {
    label "Email"
    text "Your email"
    with_validators
        (String.IsNullOrWhiteSpace >> not)
        (fun s -> s.Contains "@")
    }

let password =
    input {
    label "Password"
    password "Must contains at least 6 characters, one number and one uppercase"
    with_validators
        (String.exists Char.IsUpper)
        (String.exists Char.IsDigit)
        (fun s -> s.Length >= 6)
    }
```

Prior to F# 5, each of these **and!** keywords would have been **let!** keywords. The **and!** keyword differs in that the expression that follows it must be 100% independent. It cannot depend on the result of a previous **let!-**bound value. That means code like the following fails to compile:

```
let res1: Result<int, string> =
    result {
        let! a = r1
        and! b = r2 a // try to pass 'a'
        and! c = r3 b // try to pass 'b'
        return a + b - c
    }
```

So, why would we make that code fail to compile? A few reasons. First, it enforces computational independence at compile-time. Second, it does buy a little performance at runtime because it allows the compiler to build out the call graph statically. Third, because each computation is independent, they can be executed in parallel by whatever environment they're running in. Lastly, if a computation fails, such as in the previous example where one may return an **Error** value instead of an **Ok** value, the whole thing doesn't short-circuit on that failure. Applicative forms "gather" all resulting values and allow each computation to run before finishing. If you were to replace each **and!** with a **let!,** any that returned an **Error** short-circuits out of the function. This differing behavior allows library authors and users to choose the right behavior based on their scenario.

If this sounds like it's a little concept-heavy, that's fine! Applicative computations are a bit of an advanced concept from a library author's point of view, but they're a powerful tool for abstraction. As a user of them, you don't need to know all the ins and outs of how they work; you can simply know that each computation in a computation expression is guaranteed to be run independently of the others.

Another enhancement to computation expressions is the ability to properly support overloading for custom operations with the same keyword name, support for optional arguments, and support for **System.ParamArray** arguments. A custom operation is a way for a library author to specify a special keyword that represents their own kind of operation that can happen in a computation expression. This feature is used a lot in frameworks like Saturn to define an expressive DSL for building Web apps. Starting with F# 5, authors of components like Saturn can overload their custom operations without any caveats, as shown in **Listing 1**.

Proper support for overloads in Custom operations are developed entirely by two F# open source contributors Diego Esmerio and Ryan Riley.

With applicative forms for computation expressions and the ability to overload custom operations, we're excited to see what F# library authors can do next.

Interface Implementations at Different Generic Instantiations

Starting with F# 5, you can now implement the same interface at different generic instantiations. This feature was developed in partnership with Lukas Rieger, an F# open source contributor.

```
type IA<'T> =
    abstract member Get : unit -> 'T

type MyClass() =
    interface IA<int> with
        member x.Get() = 1
    interface IA<string> with
        member x.Get() = "hello"

let mc = MyClass()
let asInt = mc :> IA<int>
let asString = mc :> IA<string>

asInt.Get() // 1
asString.Get() // "hello"
```

More .NET Interop Improvements

.NET is an evolving platform, with new concepts introduced every release and thus, more opportunities to interoperate. Interfaces in .NET can now specify default implementations for methods and properties. F# 5 lets you consume these interfaces directly. Consider the following C# code:

```
using System;

namespace CSharpLibrary
{
    public interface MyDim
    {
        public int Z => 0;
    }
}
```

This interface can be consumed directly in F#:

```
open CSharp

// Create an object expression
// to implement the interface
let md = { new MyDim }
printfn $"DIM from C#: {md.Z}"
```

Another concept in .NET that's getting some more attention is nullable value types (formerly called Nullable Types). Initially created to better represent SQL data types, they are also foundational for core data manipulation libraries like the Data Frame abstraction in Microsoft.Data.Analysis. To make it a little easier to interop with these libraries, you apply a new type-directed rule for calling methods and assigning values to properties that are a nullable value type. Consider the following sample using this package with a package reference directive:

```
#r "nuget: Microsoft.Data.Analysis"

open System
open Microsoft.Data.Analysis

let dateTimes =
     "Datetimes"
    |> PrimitiveDataFrameColumn<DateTime>

// The following used to fail to compile
Let date = DateTime.Parse("2019/01/01")
dateTimes.Append(date)

// The previous is now equivalent to:
Let date = DateTime.Parse("2019/01/01")
Let data = Nullable<DateTime>(date)
dateTimes.Append(data)
```

F# and .NET SDK version	Time to compile (in seconds)
F# 5 and .NET 5 SDK	49.23 seconds
F# 4.7 and .NET Core 3.1 SDK	68.2 seconds
F# 4.5 and .NET Core 2.1 SDK	100.7 seconds

Table 1: Compile times for FSharpPlus.dll across recent F# versions

Jupyter Notebooks

Jupyter Notebooks are an interactive programming tool that lets you mix markdown and code in a document. The code can be executed in the notebook, often to produce structured data or charts that go hand-in-hand with an explanation.

Jupyter Notebooks started as IPython, an interactive programming tool for Python programs. It has grown to support many different languages and is now one of the primary tools used by data scientists in their work. It's also being used as an educational tool.

Learn more at: https://jupyter.org/

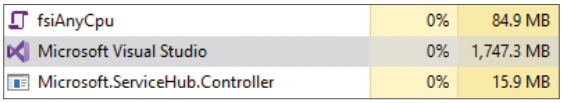


Figure 5: Peak memory usage running Find References on string in FAKE.sln in VS 16.5

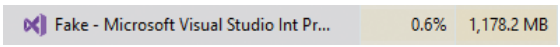


Figure 6: Peak memory usage running Find References on string in FAKE.sln in VS 16.6 and higher

These examples used to require that you explicitly construct a nullable value type with the *Nullable* type constructor as the example shows.

Better Performance

The Microsoft team has spent the past year improving F# compiler performance both in terms of throughput and tooling performance in IDEs like Visual Studio. These performance improvements have rolled out gradually rather than as part of one big release. The sum of this work that culminates in F# 5 can make a difference for everyday F# programming. As an example, I've compiled the same codebase—the core project in FSharpLus, a project that notoriously stresses the F# compiler—three times. Once for F# 5, once for the latest F# 4.7 with .NET Core, and once for the latest F# 4.5 in .NET Core, as shown in **Table 1.**

The results in **Table 1** come from running **dotnet build /clp:PerformanceSunnary** from the command-line and looking at the total time spent in the **Fsc** task, which is the F# compiler. Results might vary on your computer depending on things like process priority or background work, but you should see roughly the same decreases in compile times.

IDE performance is typically influenced by memory usage because IDEs, like Visual Studio, host a compiler within a language service as a long-lived process. As with other server processes, the less memory you use up, the less GC time is spent cleaning up old memory and the more time can be spent processing useful information. We focused on two major areas:

- Making use of memory-mapped files to back metadata read from the compiler
- Re-architecting operations that find symbols across a solution, like Find All References and Rename

The result is significantly less memory usage for larger solutions when using IDE features. **Figure 5** shows an example of memory usage when running Find References on the *string* type in FAKE, a very large open source codebase, prior to the changes we made.

This operation also takes one minute and 11 seconds to complete when run for the first time.

With F# 5 and the updated F# tools for Visual Studio, the same operation takes 43 seconds to complete and uses over 500MB less memory, as shown in **Figure 6**.

The example with results shown in **Figure 5** and **Figure 6** is extreme, since most developers aren't looking for usages of a base type like *string* in a very large codebase, but it goes to show how much better performance is when you're using F# 5 and the latest tooling for F# compared to just a year ago.

Performance is something that is constantly worked on, and improvements often come from our open source contributors. Some of them include Steffen Forkmann, Eugene Auduchinok, Chet Hust, Saul Rennison, Abel Braaksma, Isaac Abraham, and more. Every release features amazing work by open source contributorse're eternally grateful for their work.

The Continuing F# Journey and How to Get Involved

The Microsoft F# team is very excited to release F# 5 this year and we hope you'll love it as much as we do. F# 5 represents the start of a new journey for us. Looking forward, we're going to continually improve interactive experiences to make F# the best choice for notebooks and other interactive tooling. We're going to go deeper in language design and continue to support libraries like DiffSharp to make F# a compelling choice for machine learning. And as always, we're going to improve on F# compiler and tooling fundamentals and incorporate language features that everyone can enjoy.

We'd love to see you come along for the ride, too. F# is entirely open source, with language suggestions, language design, and core development all happening on GitHub. There are some excellent contributors today and we're seeking out more contributors who want to have a stake in how the F# language and tools evolve moving forward.

To get involved on a technical level, check out the following links:

- F# language suggestions: https://github.com/fsharp/fslang-suggestions
- F# language design: https://github.com/fsharp/fslang-design
- F# development: https://github.com/dotnet/fsharp
- F# running on JavaScript: https://fable.io/
- F# tooling for Visual Studio Code: http://ionide.io/
- F# running on Web Assembly: https://fsbolero.io/

The F# Software Foundation also hosts a large slack community, in addition to being a central point for various subcommunities to share information with one another. It's free to join, so head over to the website here to learn more: http://foundation.fsharp.org/join

Want to have a say in where F# goes next and how it does it? Come join us. We'd love to work together.

Phillip Carter

Xamarin.Forms 5: Dual Screens, Dark Modes, Designing with Shapes, and More

Beginning in early 2020, the Xamarin team started collaborating with the Surface and Windows developer teams to think about a new dual screen device that we were about to launch at Microsoft, and which you have no doubt heard about by now, the Surface Duo. This new device, which also makes phone calls, poses some unique opportunities to create new, productive mobile experiences. Our engineering team leapt at the chance to showcase how powerful Xamarin development can be for Android and cross-platform developers alike. How do you display controls on one screen which then spans to another screen? Where do you put navigation? Should content flow below the hinge, or space evenly? So many questions...

At our Developer Day launch event in early February, we delivered a fully functional native application built with Xamarin.Forms. The most remarkable thing about this application is that no core modifications to Xamarin.Forms were needed in order to achieve this. It's a strong validation to the design of the product. You can explore the source at https://aka.ms/app-xamarintv.

During six years in the market, Xamarin.Forms has spanned six versions of iOS and Android, run on Windows Phones, tablets, and desktops from WPF to UWP and now WinUI, and even been extended by contributors to macOS, Linux, and Samsung's Tizen platform. Thousands of companies of all sizes are using Xamarin.Forms to power mobile and desktop apps used in the consumer market as well as suites of line-of-business needs.

For the Surface Duo, we added a TwoPaneView layout (based on work by our Windows team), and a whole bunch of new state triggers to help you adapt to new screen sizes, orientations, and postures. Xamarin.Forms 5 also introduces drag-and-drop gestures. What we've done in Xamarin.Forms 5 to make your experience developing for dual-screens more delightful is just the beginning of what you can do with this release. Xamarin.Forms 5 contains the simplicity and productivity we've added based on your constant feedback to make it easier and faster for you to deliver beautiful cross-platform applications that share more code than ever before.

To give you the whirlwind tour of the development power you'll experience using Xamarin.Forms 5, I'll run you through the Fly Me sample app I built in the hopes that one day soon I'll again be able to board a plane and travel the globe. Xamarin.Forms 5 uses Shell to simplify the top-level things that every app needs. It also adds new design features such as shapes and brushes; combined with control templates; these new features speed up your UI development.

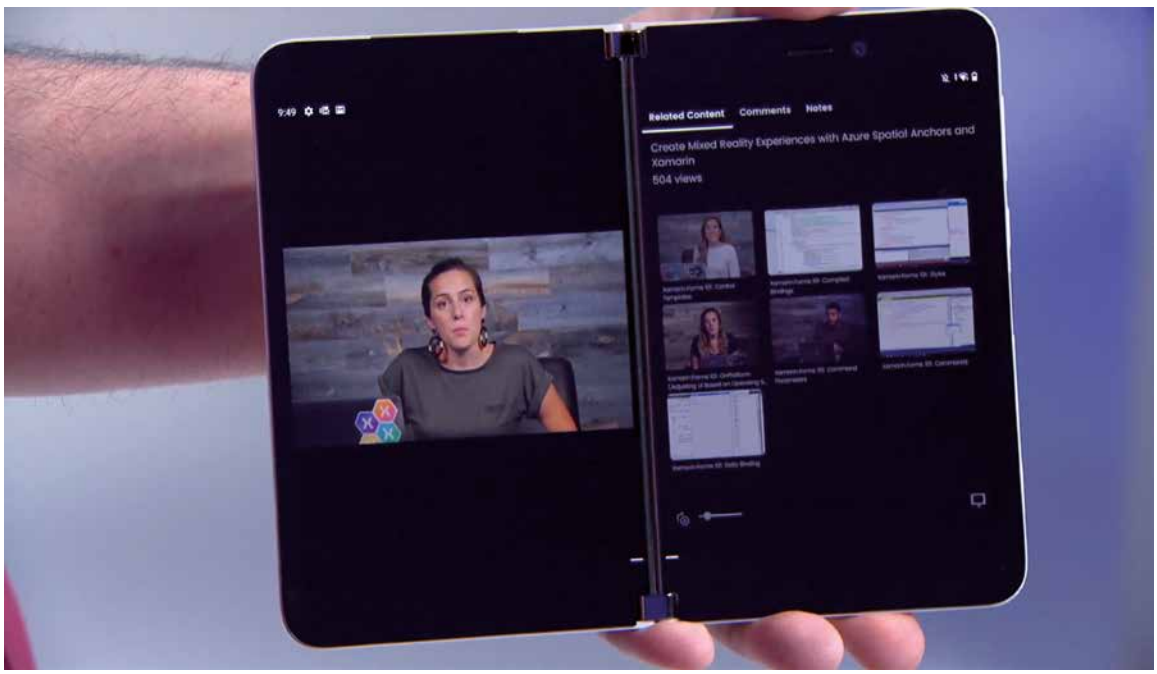


Figure 1: Xamarin.Forms was featured at the Surface Duo SDK launch event.



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

David is a Principal Program Manager on the .NET team at Microsoft, focused on Xamarin.Forms. A .NET developer since 2002, and versed in a range of programming languages, David has developed Web, environmental, and mobile experiences for a wide variety of industries. After several successes with tech startups and running his own software company, David joined Microsoft to follow his passion: crafting tools that help developers create better app experiences. When not at a computer or with his family, David is exploring trails through the woods.

Simplicity Is Primary

Simplicity starts at the container level of your application, commonly referred to as the app shell. In Xamarin.Forms 5, every app template starts with a simple **Shell** in a file named **AppShell.xaml**. This is where you describe the visible navigation structure for your application, whether using a flyout menu, tab menus, or any combination, as you can see in **Figure 2**.

Login and Flyout Navigation

Fly Me uses a flyout menu that flies out from the side of the UI over the app content. In **Figure 2** you can see a header, flyout items to navigate to pages throughout the app, and a logout menu item. The code begins simply with the Shell container.

```
<Shell
    xmlns="…"
    xmlns:x="…"
    xmlns:views="clr-namespace:FlyMe.Views"
    xmlns:vm="clr-namespace:FlyMe.ViewModels"
    FlyoutHeaderTemplate="{DataTemplate views:HeaderView}"
    x:Class="FlyMe.AppShell">

    <Shell.BindingContext>
        <vm:AppViewModel/>
    </Shell.BindingContext>
</Shell>
```

To begin populating the **Shell**, you add nodes for each item you wish to appear in the flyout. Each item can take a variety of properties including a named route for URI navigation, and the title and icon you wish to display. The content, of type **ContentPage**, that's displayed when the user selects the item, is contained in the **ShellContent**. The **DataTemplate** is used to make sure that the content is only created when you need it, thus keeping your application load time minimal.

```
<FlyoutItem
    Route="home"
    Title="My Flights"
    Icon="{StaticResource IconTabMyFlights}">
    <ShellContent
        ContentTemplate="{DataTemplate views:MyFlightsPage}" />
</FlyoutItem>

<FlyoutItem
    Title="Today"
    Icon="{StaticResource IconTabToday}">
    <ShellContent
        ContentTemplate="{DataTemplate views:TodayPage}" />
</FlyoutItem>
```

If all you have are a series of **FlyoutItems**, the application will load the first **ShellContent** and display the flyout menu icon in the upper left. In this application, you'll first display a log in page, and make sure that the rest of the application isn't accessible until the user authenticates. In order to do this, place a **ShellItem** before the **FlyoutItems** in the **AppShell.xaml.**

```
<ShellItem
    Route="login"
    IsVisible="{Binding IsNotLoggedIn}">
    <ShellContent
        ContentTemplate="{DataTemplate views:LoginPage}" />
</ShellItem>

<FlyoutItem
    Route="home"
    Title="My Flights"
    Icon="{StaticResource IconTabMyFlights}">
    <ShellContent
        ContentTemplate="{DataTemplate views:MyFlightsPage}" />
</FlyoutItem>
```

> If you've used XAML before, this mixing of visual and non-visual elements may seem a bit odd.

In XAML, the order of items matters, and you use that to your benefit here by making sure the **ShellItem** for the **LoginPage** appears **before** the **FlyoutItem**. This tells Shell to display just the first and not the second item. The same pattern is useful for displaying onboarding sequences and the like before the rest of your application is displayed. If you've used XAML before, this mixing of visual and non-visual elements may seem a bit odd. The AppShell is a unique use of XAML that both expresses the content of the application and some UI expectations (that you want **Shell** to render a flyout or tabs or both). A **ContentPage**, by comparison, explicitly describes UI elements.

In order to hide the log in page and let **Shell** proceed with displaying the rest of the application, you use the new

What is Xamarin.Forms?

Xamarin.Forms is a cross-platform toolkit for building native mobile and desktop apps with .NET and Visual Studio. You can use XAML and C# to declare your UI, and common architectural patterns like Model-View-ViewModel and Reactive UI.

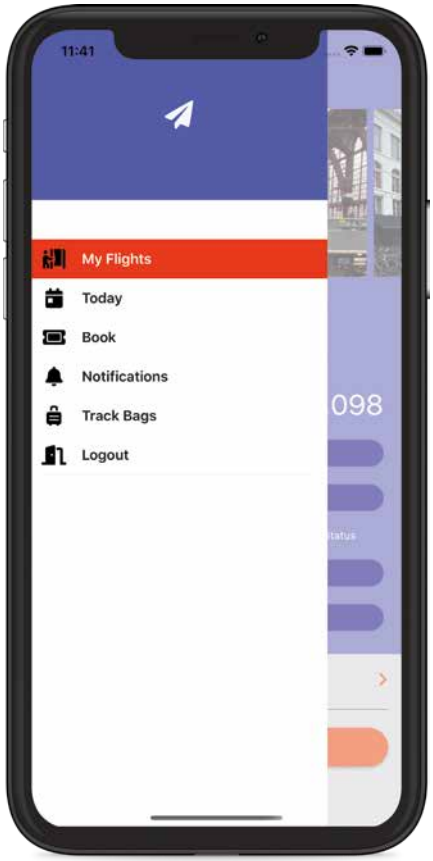


Figure 2: Shell flyout menu

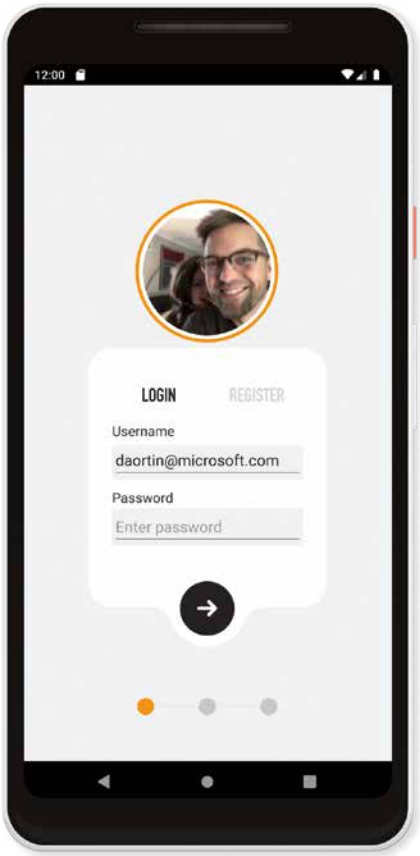


Figure 3: Log in with shapes, paths, and clipping

ShellItem.IsVisible property. Any item in your **Shell** that should not be navigable may be protected by setting that property to false.

Styling Your Flyout

You could create renderers to completely replace all the content of the flyout, but that tends to be a lot more code than you really need. Xamarin.Forms 5 simplifies adding content to the header, styling the flyout item templates, and even styling the backdrop that appears behind the flyout but over the page. You can instead add your own header content using the **Shell**'s **FlyoutHeaderTemplate** property, which in this example, is simply a grid and an image:

```
<Grid
    RowDefinitions="66,120"
    BackgroundColor="#5561F9">
    <Image
        Grid.RowSpan="2"
        VerticalOptions="Center"
        HorizontalOptions="Center"
        Source="{FontImage
            FontFamily=FontAwesome,
            Glyph=&#xf1d8;,
            Color=GhostWhite,
            Size=32}"/>
</Grid>
```

Each **FlyoutItem** has a title and icon. When you wish to style them further, changing the font size, colors, and selected states, you can now use new style classes to achieve the look you desire. Three classes and two element names are now available, as shown in **Table 1**.

Using these style classes, you can style the layout for each flyout item and use **VisualStateManager** with the new **TargetName** property to change the color of the label based on selection, as seen in **Listing 1**.

Before I finish discussing **Shell**, one last styling challenge to conquer is changing the color and opacity of the flyout's backdrop. The backdrop is the layer that sits behind the flyout and in front of the content page. The new **FlyoutBackdrop** takes a color or brush and is applied to any **ShellItem** via an attached property.

```
<Style
    TargetType="ShellItem"
    ApplyToDerivedTypes="true">
    <Setter Property="Shell.FlyoutBackdrop"
        Value="#CC333333"/>
</Style>
```

Limitless Design

Xamarin.Forms 5 introduces several new design-focused capabilities, including embedded fonts, font image support, shapes and paths, and brushes. These enhancements combine to bring any UI design within easy reach by using simple APIs shipping "in the box."

Embedded Fonts and Font Image Source

Adding fonts is now easier than ever, whether for beautifying your text or for using font icon glyphs. Add any mobile-supported font file such as TTF or OTF to your Xamarin.Forms .NET Standard Library and set the build type to "Embedded Resource." You can then add an assembly tag that tells the build tasks to make that font available by name or alias to your app, no matter which platform it's running on.

Fly Me uses Font Awesome, a popular free font with useful icons. In any ***.cs** file in the project, you can add the assembly attribute **ExportFont**. Although you could use the filename to refer to the font, it's helpful to provide an alias to declare your preferred name.

```
[assembly: ExportFont("fa-solid-900.ttf",
    Alias = "FontAwesome")]
```

To then display icons using the embedded font, assign your **FontImageSource** to any control that accepts an image source. You need only provide the font family, the glyph to be displayed, and styling information such as color and size.

Flyout Item Part	Style Class Name	Element Name
Text	FlyoutItemLabelStyle	FlyoutItemLabel
Icon	FlyoutItemIconStyle	FlyoutItemIcon
Container	FlyoutItemLayoutStyle	

Table 1: The available classes and elements

Listing 1: Styling the layout of the flyout items

```
<Style
    TargetType="Layout"
    ApplyToDerivedTypes="True"
    Class="FlyoutItemLayoutStyle">
    <Setter
        Property="HeightRequest"
        Value="44" />
    <Setter
        TargetName="FlyoutItemLabel"
        Property="Label.FontSize"
        Value="16" />
    <Setter
        TargetName="FlyoutItemLabel"
        Property="Label.TextColor"
        Value="{StaticResource TextOnLightColor}" />
    <Setter
        TargetName="FlyoutItemLabel"
        Property="Label.HeightRequest"
        Value="44" />
    <Setter
        Property="VisualStateManager.VisualStateGroups">
        <VisualStateGroupList>
            <VisualStateGroup
                x:Name="CommonStates">
                <VisualState
                    x:Name="Normal">
                    <VisualState.Setters>
                    </VisualState.Setters>
                </VisualState>
                <VisualState
                    x:Name="Selected">
                    <VisualState.Setters>
                        <Setter
                            Property="BackgroundColor"
                            Value="#FF3300" />
                        <Setter
                            TargetName="FlyoutItemLabel"
                            Property="Label.TextColor"
                            Value="White" />
                    </VisualState.Setters>
                </VisualState>
            </VisualStateGroup>
        </VisualStateGroupList>
    </Setter>
</Style>
```

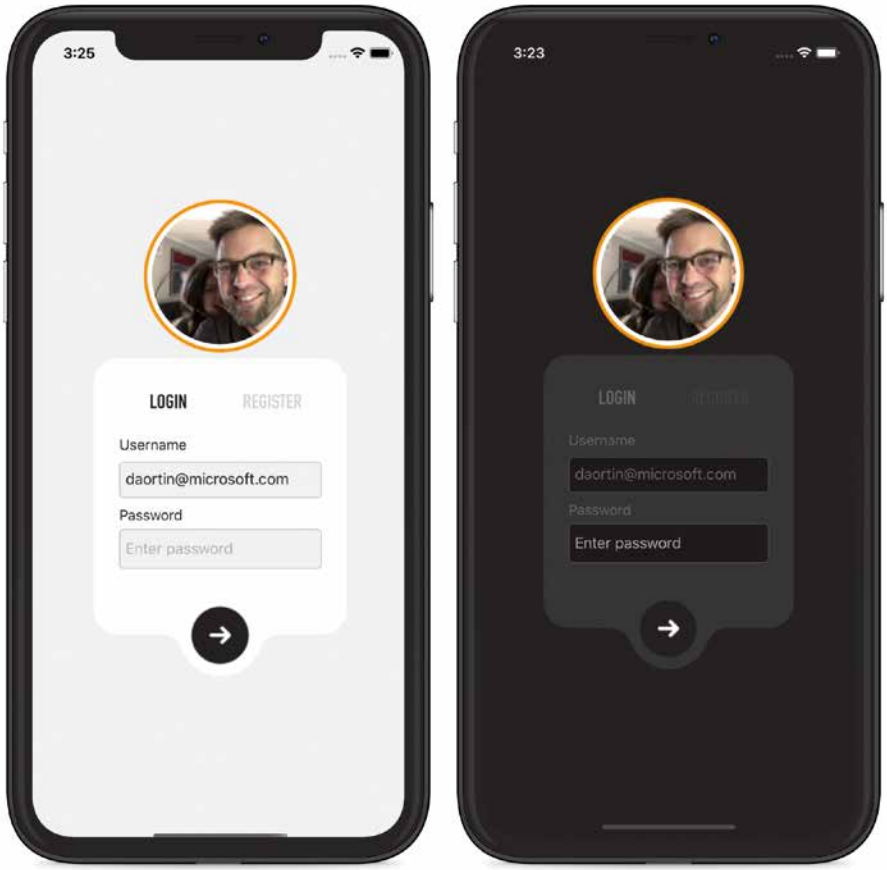


Figure 4: Appearance modes support light and dark

The flyout header demonstrates this using the handy **FontImage** markup extension.

Shapes, Paths, and Clipping

With shapes and paths, you have a whole new way to achieve your designs in Xamarin.Forms. Paths are a series of vector points and lines that can describe complex shapes. Due to their complexity, you won't really want to type the path data by hand. From almost any design tool (I like Figma) you can copy path data for a vector shape and copy it right into your code. In **Figure 3** you can see a different style of login screen that uses an interesting shape for the form background.

```
<Path
    Fill="#333333"
    Data="M251,0 C266.463973,-2.84068575e-15 279,12.536027
279,28 L279,276 C279,291.463973 266.463973,304 251,304
L214.607,304 L214.629319,304.009394 L202.570739,304.356889
C196.091582,304.5436 190.154631,308.020457
186.821897,313.579883 L186.821897,313.579883
L183.402481,319.283905 C177.100406,337.175023
160.04792,350 140,350 C119.890172,350 102.794306,337.095694
96.5412691,319.115947 L96.5273695,319.126964
L92.8752676,313.28194 C89.5084023,307.893423
83.6708508,304.544546 77.3197008,304.358047 L65.133,304
L28,304 C12.536027,304 1.8937905e-15,291.463973 0,276 L0,28
C-1.8937905e-15,12.536027 12.536027,2.84068575e-15 28,0
L251,0 Z"
/>
```

Be glad that you don't have to type that data string yourself, though with some practice you'll get pretty good understanding what's going on. SVG images also commonly use path data, and you can use your favorite text editor to read and copy the data string.

You can also draw primitive shapes like ellipse, line, polygon, polyline, and rectangle. Each of these shapes support common styling properties such as aspect, fill, and a variety of stroke options.

```
<StackLayout
    Orientation="Horizontal"
    HorizontalOptions="Center">
    <Ellipse Fill="#FF9900" />
    <Line />
    <Ellipse />
    <Line />
    <Ellipse />
</StackLayout>
```

One of the most powerful uses of shapes is the ability to clip other controls, also known as "masking." A square profile image can become a circle by applying an **EllipseGeometry** to the **Image.Clip** property. The same can be done with a path or any other shape, and clipping can be applied to any control or layout in Xamarin.Forms!

```
<Image
    HorizontalOptions="Center"
    VerticalOptions="Center"
    WidthRequest="150"
    HeightRequest="150"
    Source="profile.png">
    <Image.Clip>
        <EllipseGeometry
            Center="75,75"
            RadiusX="75"
            RadiusY="75"/>
    </Image.Clip>
</Image>
```

Dark Mode

Modern operating systems all now have some form of support for light and dark modes, like those seen in **Figure 4.** These modes may be triggered by ambient light sensors, time of day, or by user preference. You can make your colors and styles aware of appearance modes using a new binding extension appropriately called **AppThemeBinding**. If you use the default colors provided by the platform, and you make no customization, and then your application uses the platform default colors for light and dark appearance modes. To take more creative control, update your application styles and set the colors directly.

```
<Style
  TargetType="Page"
  ApplyToDerivedTypes="True">
  <Setter
    Property="BackgroundColor"
    Value="{AppThemeBinding Dark=#222222,
                            Light=#f1f1f1}" />
</Style>
```

Now when appearance mode changes on the devices for any reason, the application updates at runtime. How can you then opt-out of this behavior and allow the user to set their own appearance mode preference? Xamarin.Forms

Listing 2: Control template for a RadioButton

```
<ControlTemplate x:Key="CalendarRadioTemplate">
    <Frame HasShadow="False" HeightRequest="100" WidthRequest="100"
           HorizontalOptions="Start" VerticalOptions="Start"
           Padding="0">
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroupList>
                <VisualStateGroup x:Name="CommonStates">
                    <VisualState x:Name="Normal">
                        <VisualState.Setters>
                            <Setter Property="BackgroundColor"
                                    Value="#f3f2f1"/>
                            <Setter Property="BorderColor"
                                    Value="DarkBlue"/>
                            <Setter Property="Grid.IsVisible"
                                    TargetName="RadioIcon"
                                    Value="False"/>
                        </VisualState.Setters>
                    </VisualState>
                    <VisualState x:Name="Checked">
                        <VisualState.Setters>
                            <Setter Property="BorderColor"
                                    Value="DarkBlue"/>
                            <Setter Property="Grid.IsVisible"
                                    TargetName="RadioIcon"
                                    Value="False"/>
                        </VisualState.Setters>
                    </VisualState>
                </VisualStateGroup>
            </VisualStateGroupList>
        </VisualStateManager.VisualStateGroups>
        <Grid Margin="4" WidthRequest="100">
            <Grid x:Name="RadioIcon" WidthRequest="18"
                  HeightRequest="18" HorizontalOptions="End"
                  VerticalOptions="Start">
                <Ellipse Stroke="DarkBlue" WidthRequest="16"
                         HeightRequest="16" StrokeThickness="0.5"
                         VerticalOptions="Center"
                         HorizontalOptions="Center" Fill="White" />
                <Ellipse WidthRequest="8" HeightRequest="8"
                         Fill="DarkBlue" VerticalOptions="Center"
                         HorizontalOptions="Center" />
            </Grid>
            <ContentPresenter></ContentPresenter>
        </Grid>
    </Frame>
</ControlTemplate>
```

5 provides **App.Current.UserAppTheme** that you can set from anywhere in the application. You can choose from **OSAppTheme.Dark**, **OSAppTheme.Light**, or **OSAppTheme.Unspecified**. Choosing **Unspecified** gives control back to the platform to trigger appearing changes.

```
App.Current.UserAppTheme = OSAppTheme.Dark;
```

New Control Customizations

Windows desktop platforms have long enjoyed "lookless controls" that allow you to skin parts of a control by supplying a control template. You may not realize this, but Xamarin.Forms has supported control templates since version 2! In **Figure 5**, you can see the default **RadioButton** control without any styling applied. As you would expect the XAML is plain.

> We didn't provide the ability to supply a template that works seamlessly with core controls... until now

```
<StackLayout
                RadioButtonGroup.GroupName="SimpleRadios"
                Orientation="Horizontal">
                <RadioButton Content="Day"/>
                <RadioButton Content="Week"/>
                <RadioButton Content="Month"/>
</StackLayout>
```

The main difference between control templates in Xamarin.Forms versus other platforms is that, because the controls have adhered closely to the native experience, we didn't provide the ability to supply a template that works seamlessly with core controls...until now. In Xamarin.Forms 5, we're introducing support for control templating on the new **RadioButton** control.

You can apply a control template, such as **Listing 2,** directly to the **RadioButton.ControlTemplate** property, or better yet by setting a style. And because **RadioButton** takes any content, you can provide layout and controls that will be applied to the **ContentPresenter** in the template. Check out how much better this looks now in **Figure 6**!

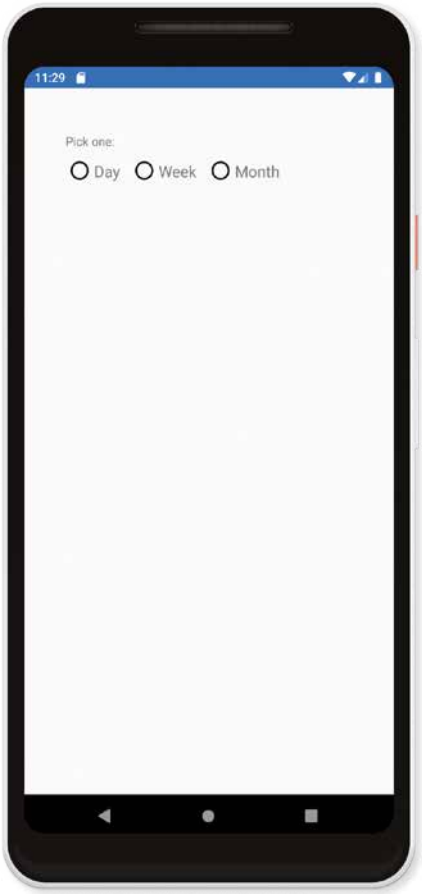


Figure 5: Basic RadioButton controls

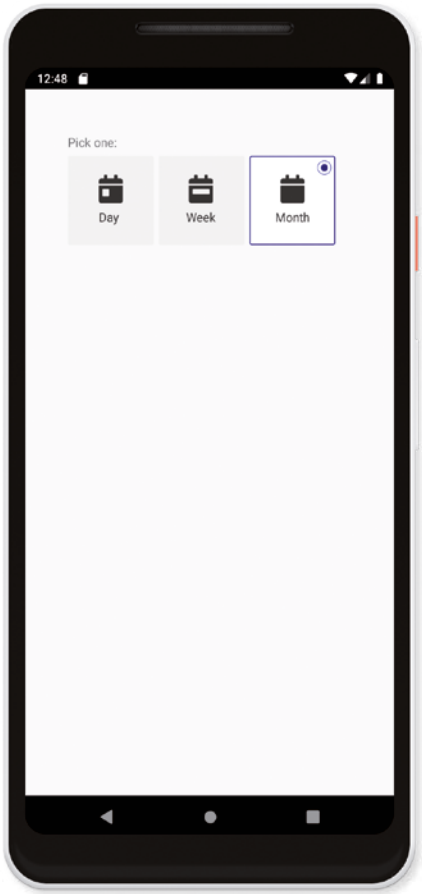


Figure 6: RadioButtons with content and control template

In Xamarin.Forms 5 you'll also find:		
Accessibility - TabIndex	FontImageSource	RefreshView
AdaptiveTrigger	GIF support	Shapes & Paths
AndroidX	Grid Row/Col simplificiation	Shell Modals
AppTheme (Dark Mode)	HTML Label	SourceLink
Brushes - gradient and solid	Image loading and error source	SpanModeStateTrigger
Bug fixes	IndicatorView	StateTrigger
CarouselView	Kerning / Character Spacing	SwipeView
CheckBox	Label padding	Switch VisualStates
Clipping	Label Transform	Transparent background modals
CollectionView	Map Shapes	TwoPaneView
CompareStateTrigger	MultiBindings	Various markup extensions
DeviceStateTrigger	MultiTriggers	Various Platform Specifics
Drag & Drop gestures	Native enhancements to Maps and WebView	VisualStateManager Target
DualScreen SDK	OrientationStateTrigger	WebView Cookies

Table 2: There are many more features to explore.

Faster Development in Visual Studio

The tooling experience continues to improve for Xamarin.Forms developers. Install the very latest release of Visual Studio 2019 to enjoy XAML Hot Reload for Android, iOS, macOS, and UWP, a new control toolbox and property panel with help for bindings and color selection, and a live visual tree inspector when debugging. For Windows developers, you can use Hot Restart to develop directly to your iOS device.

```
<RadioButton
  ControlTemplate="{StaticResource
                    CalendarRadioTemplate}">
  <RadioButton.Content>
    <StackLayout
        HorizontalOptions="Center"
        VerticalOptions="Center">
      <Image Source="{FontImage
                FontFamily=FontAwesome,
                Glyph=&#xf783;,
                Color=#323130, Size=32}"/>
      <Label Text="Day" TextColor="#323130"/>
    </StackLayout>
  </RadioButton.Content>
</RadioButton>
```

Although this is traditionally something you'd only think of doing in XAML, and I know that many of you do just that, control templates are equally powerful directly from C#. You just need to assign a custom template to the control's template property; described in C# or XAML, it doesn't matter.

Supplying a control template is a very powerful and convenient way to fully customize a control. When you do this, be aware that the control no longer uses the native platform control but only the cross-platform features. In most cases, this flexibility and control far outweighs that trade-off. For special cases, you can always adopt the custom renderer strategy.

What's Next

Xamarin.Forms 5 is a monumental release, and it wouldn't be that without amazing work from our many contributors. In this release, as compared to the last, contributions have increased nearly 34%. In 2020, Xamarin.Forms has twice set new high marks for usage, and you've reported to us the highest satisfaction ratings the platform has ever seen. This is really a celebration of you and your continued contributions and collaboration. Thank you!

Much of Xamarin.Forms 5 has been in preview for several months while we worked through your feedback to make it ready for a stable release. This article showcases only some the new features at your disposal. See the documentation at https://docs.microsoft.com/xamarin/whats-new/ for details about many more features, such as those shown in **Table 2.**

Our .NET team is already working on .NET 6 and the next release that will focus on surmounting significant challenges in app performance, improved desktop support, and advanced control customizations. It's encouraging that everything you invest today in .NET and Xamarin.Forms 5 has a future path for many years to come within .NET.

To get started today with Xamarin.Forms 5, you can quickly update your existing projects via your favorite NuGet package manager. The new project templates in Visual Studio will also be updated to use Xamarin.Forms 5 by default, so you get the very best experience. Continue to send us feedback and let us know how we're doing. Happy coding!

David Ortinau

.NET 5.0 Runtime Highlights

The .NET runtime is the foundation of the .NET platform and is therefore the source of many improvements and a key component of many new features and enabled scenarios. This is even more true since Microsoft started the .NET Core project. Many of the performance improvements and key changes we made to optimize scenarios (like Docker containers) have come from the runtime.

With each new .NET version, the .NET team chooses which new features and scenarios to enable. We listen to feedback from users, with most of it coming from GitHub issues. We also look at where the industry is headed next and try to predict the new ways that developers will want to use .NET. The features I want to tell you about are a direct outcome of those observations and predictions.

I'm going to tell you about two big .NET 5.0 projects: **single file apps**, and **ARM64**. There are many other improvements in .NET 5.0 that there simply isn't room to cover here, like P95 performance improvements, new diagnostic capabilities (like dotnet-monitor), and advances in native interop (like function pointers). If you're mostly interested in performance improvements, please check out the .NET 5.0 performance post at https://aka.ms/dotnet5-performance. Take a look at the .NET blog (https://aka.ms/dotnet5) to learn about the full set of improvements in this release and why you should consider adopting .NET 5.0 for your next project.

Single File Apps

Single file apps significantly expand .NET application deployment options with .NET 5.0. They enable you to create standalone, true xcopy, single-file executables. This capability is appealing for command-line tools, client applications, and Web applications. There's something truly simplifying and productive about launching a single file app from a network share or a USB drive, for example, and having it reliably just run on any computer without requiring installation pre-steps.

Single file apps are supported for all application types (ASP.NET Core, Windows Forms, etc.). There are some differences, depending on the operating system or application type. Those will be covered in the following sections.

All of This Has Happened Before

Many people have correctly observed and noted that .NET Core apps are not as simple as .NET Framework ones. For the longest time, .NET Framework has been part of Windows and .NET Framework executables have been very small single files. That's been really nice. You could put a console or Windows Forms app on a network share and expect it run. This has been possible because .NET Framework is integrated into Windows. The Windows Loader understands .NET executable files (like myapp.exe), and then hands execution off to the .NET Framework to take over.

When Microsoft built .NET Core, we had to start from scratch with many aspects of the platform, including how apps were launched. A driving goal was providing the same application behavior on all operating systems. We also didn't want to require operating system updates to change the behavior (like needing to run Windows Update to get a .NET Core app working). This led us to not replicate the approach we used for .NET Framework, even though we (very) briefly considered it. We needed to build a native launcher for discovering and loading the runtime for each supported operating system. That's how we ended up with multiple files: at least one for the launcher and another for the app. We're not alone; multiple other platforms have this too. For example, Java and Node.js apps have launchers.

Back to Basics

Over time, we heard more and more feedback that people wanted a single file application solution. Although it's common for language platforms to require launchers, other platforms like C++, Rust, and Go don't require them, and they offer single file as a default or an option you can use.

You wouldn't necessarily think of single file apps as a runtime feature. It's a publish option, right? In actuality, the work to enable single file apps was done almost exclusively in the runtime. There are two primary outcomes we needed to enable: include the runtime within the single file and load managed assemblies from within the single file. In short, we needed to adapt the runtime to being embedded in a single file configuration. It's sort of a new hosting model.

As I said, we use a launcher for .NET Core apps. It's responsible for being a native executable, discovering the runtime, and then loading the runtime and the managed app. The most obvious solution was to statically link the runtime into the launcher. That's what we did, starting with Linux for the .NET 5.0 release. We call the result the "super host." Native code runtime and library components are linked into the super host. Linux names are listed here (Windows names in brackets):

- libcoreclr.so (coreclr.dll)
- libclrjit.so (clrjit.dll)
- libmscordaccore.so (mscordaccore.dll)
- Library native components (for example, libSystem.IO.Compression.Native.so)

We focused on Linux for the single file experience with .NET 5.0. Some parts of the experience are only available on Linux, and other parts are also supported on Windows and macOS. These differences in capability will be called out throughout this document. There are critical challenges that we ran into building this feature. The combination of these issues led us to enable the broadest set of experiences with Linux, and then to wait to spend more time improving the other operating systems in upcoming releases. We also have work left to do to improve the Linux experience.

The first problem relates to the structure of the single file. The single file bundler copies managed assemblies to the end of the host (apphost or superhost) to create your app, a bit like adding ice cream to a cone. These assemblies can contain ready-to-run native code. Windows and macOS place extra requirements on executing native code that has been bundled in this way. We haven't done the work yet to satisfy



Richard Lander
rlander@microsoft.com

Richard Lander is a Principal Program Manager on the .NET team at Microsoft. He works on making .NET work great 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 and 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 he enjoys using power tools. He grew up in Canada and New Zealand.

these requirements. Continuing the analogy, these OSes require chocolate chips in the ice cream, but we only had time to get the basic vanilla and chocolate flavors ready. Fortunately, the runtime can work around this issue by copying and remapping assemblies in-memory. The workaround has a performance cost and applies to both self-contained and framework-dependent single file apps.

The second problem relates to diagnostics. We still need to teach the Visual Studio debugger to attach to and debug this executable type. This applies to other tools that use the diagnostics APIs, too. This problem only applies to single file apps that use the superhost, which is only Linux for .NET 5.0. You'll need to use LLDB to debug self-contained single file applications on Linux.

The last problem applies to digital signing on macOS. The macOS signing tool won't sign a file that's bundled (at least in the way we've approached bundling), the macOS app store won't accept .NET single file apps as a result. It also applies to macOS environments that require the "hardened runtime" mode, which is the case with the upcoming Apple Silicon computers. This restriction applies to both self-contained and framework-dependent single file apps.

We've significantly improved single file apps with .NET 5.0, but as you can likely tell, this release is just a stopping point on the .NET single file journey. We'll continue to improve single file apps in the next release, based on your feedback.

Now that we're through the theory, I'd like to show you the new experience. If you've been following .NET Core, you'll know that there are two deployment options: self-contained and framework dependent. Those same two options equally apply to single file apps. That's good. There are no new concepts to learn. Let's take a look.

Self-Contained Single File Apps

Self-contained single file apps include your app and a copy of the .NET runtime in one executable binary. You could launch one of these apps from a DVD or a USB stick and it would work. They don't rely on installing the .NET runtime ahead of time. In fact, self-contained apps (single file or otherwise) won't use a globally installed .NET runtime, even if it's there. Self-contained single file apps have a certain minimum size (by virtue of containing the runtime) and grow as you add dependencies on NuGet libraries. You can use the assembly trimmer to reduce the size of the binary.

Let's double check that we're on the same page. A self-contained single file app includes the following content:

- Native executable launcher
- .NET runtime
- .NET libraries
- Your app + dependencies (PackageRef and ProjectRef)

What you can expect:

- The apps will be larger because they're self-contained, so will take longer to download/copy.
- Startup is fast as it's unaffected by file size.
- Debugging is limited on Linux. You'll need to use LLDB.
- The native launcher is native code, so the app will only work in one environment (like Linux x64, Linux ARM64, or Windows x64). You need to publish for each environment you want to support.
- On Windows and macOS, native runtime binaries are copied beside your (not quite) single file app, by default. For WPF apps, you'll see additional WPF native binaries copied. You can opt to embed native runtime binaries instead, however, they'll be unpacked to a temporary directory on application launch.

Because I focused on Linux for this scenario, I'll demonstrate this experience on Linux and then show which parts work on Windows and macOS.

Let's start with the Linux experience. I'll do this in a Docker container, which may be easier for you to replicate. I'll start by building an app as framework-dependent (the default), then as a self-contained single file app, and then as an assembly-trimmed self-contained single file app. A lot of tool output text has been removed for brevity.

```
r@thundera ~ % docker pull mcr.microsoft.com/dotnet/sdk:5.0
r@thundera ~ % docker run --rm -it mcr.microsoft.com/dotnet/sdk:5.0
root@a255:/# dotnet new console -o app
"Console Application" was created
root@a255:/# cd app
root@a255:/app# dotnet build -c release
root@a255:/app# time ./bin/release/net5.0/app
Hello World!

real      0m0.040s
root@a255:/app# ls -s bin/release/net5.0/
total 232
200 app        8 app.dll
4 app.runtimeconfig.dev.json
4 app.deps.json   12 app.pdb
4 app.runtimeconfig.json
root@a255:/app# dotnet publish -c release
-r linux-x64 --self-contained true /p:PublishSingleFile=true
root@a255:/app# time ./bin/release/net5.0/
linux-x64/publish/app
Hello World!

real      0m0.039s
root@a255:/app# ls -s bin/release/net5.0/
linux-x64/publish
total 65848
65836 app     12 app.pdb
root@a255:/app# dotnet publish -c release
-r linux-x64 --self-contained true /p:PublishSingleFile=true
/p:PublishTrimmed=true /p:PublishReadyToRun=true
root@a255:/app# time ./bin/release/net5.0/
linux-x64/publish/app
Hello World!

real      0m0.040s
root@a255:/app# ls -s bin/release/net5.0/
linux-x64/publish
total 27820
27808 app     12 app.pdb
root@a255:/app# exit
r@thundera ~ %
```

This set of **dotnet** commands and the extra information shown for size and startup performance demonstrates how to publish single file apps and what you can expect from them. You'll see that apps using the assembly trimmer (via the **PublishTrimmed** property) are much smaller. You will also see **PublishReadyToRun** used. It isn't strictly neces-

sary but is available to make applications start faster. It will result in larger binaries. We recommend testing these features to see if they work well for your application and provide a benefit. Assembly trimming is known to break WPF apps, for example, by over-trimming. Check out https://aka.ms/dotnet5-assembly-trimming for more information on assembly trimming.

The following example shows how to set these same properties in a project file.

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <!-- Enable single file -->
    <PublishSingleFile>true</PublishSingleFile>
    <!-- Self-contained or
         framework-dependent -->
    <SelfContained>true</SelfContained>
    <!-- The OS and CPU type you are targeting -->
    <RuntimeIdentifier>linux-x64</RuntimeIdentifier>
    <!-- Enable assembly trimming - for
         self-contained apps -->
    <PublishTrimmed>true</PublishTrimmed>
    <!-- Enable AOT compilation -->
    <PublishReadyToRun>true</PublishReadyToRun>
  </PropertyGroup>

</Project>
```

The "Hello world" console app example demonstrates the baseline experience. Let's take a quick look at an ASP.NET Core Web service. You'll see that it's very similar. This time, I'll show the file sizes using ready-to-run and not.

```
r@thundera ~ % docker run --rm -it mcr.microsoft.com/dotnet/sdk:5.0
root@71c:/# dotnet new webapi -o webapi
"ASP.NET Core Web API" was created
root@71c:/# cd webapi/
root@71c:/webapi# dotnet publish -c release
-r linux-x64 --self-contained true /p:PublishSingleFile=true
/p:PublishTrimmed=true /p:PublishReadyToRun=true
root@71c:/webapi# ls -s bin/release/net5.0/
linux-x64/publish/
total 75508
 4 appsettings.Development.json  4 web.config
20 webapi.pdb    4 appsettings.json
75476 webapi
root@71c:/webapi# dotnet publish -c release
-r linux-x64 --self-contained true /p:PublishSingleFile=true
/p:PublishTrimmed=true
root@71c:/webapi# ls -s bin/release/net5.0/
linux-x64/publish/
total 44056
4 appsettings.Development.json 4 web.config
20 webapi.pdb   4 appsettings.json
44024 webapi
root@71c:/webapi# tmux
```

Next, I'll also show the app running by launching it and calling it with curl. In order to show the app and call it at the same time, I'll use a two-pane horizontally split tmux (installed via apt-get) session. Again, I'm using Docker.

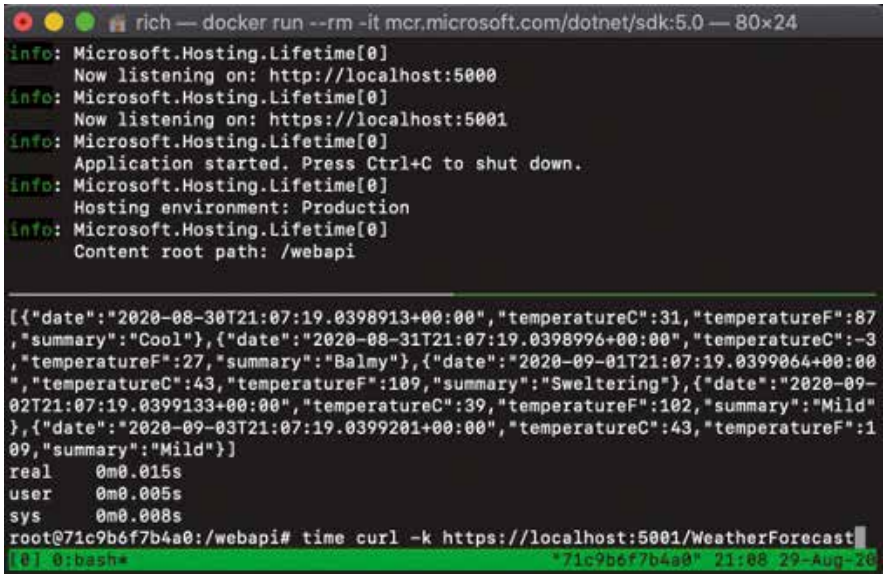


Figure 1: Calling single file ASP.NET Core Web API with curl

The command to launch the app: **./bin/release/net5.0/linux-x64/publish/webapi**.

The ASP.NET Core example is very similar to the console app. The big difference is that the size increases due to ready-to-run compilation is more apparent. Again, you should test your application in various configurations to see what's best.

If you publish a self-contained single-file app with containers, you should base it on a runtime-deps image. You don't need to use an aspnet image, because ASP.NET Core is already contained in the single-file app. See image URLs below.

- https://hub.docker.com/_/microsoft-dotnet-runtime-deps/
- https://hub.docker.com/_/microsoft-dotnet-aspnet

I'll now show you the experience on macOS, which matches the experience on Windows. As stated earlier, we didn't build a superhost for macOS or Windows in .NET 5.0. That means that the native runtime binaries are present **beside** the (not quite) single file. That's not the desired behavior, but it's what we have for .NET 5.0. We added a feature to embed these files and then unpack them upon application launch: **IncludeNativeLibrariesForSelfExtract.** That model isn't perfect. For example, the files can get deleted from the temp location (or are never deleted). This feature isn't generally recommended, but it may be the right choice in some cases.

```
r@thundera ~ % dotnet new console -o app
"Console Application" was created
r@thundera ~ % cd app
r@thundera app % dotnet publish -c release
-r osx-x64 --self-contained true
/p:PublishSingleFile=true /p:PublishTrimmed=true
/p:PublishReadyToRun=true
r@thundera app % ls -s bin/release/net5.0/
osx-x64/publish
total 47856
26640 app
   24 app.pdb
 1792 libSystem.IO.Compression.Native.dylib
```

```
  144 libSystem.Native.dylib
   32 libSystem.Net.Security.Native.dylib
  104 libSystem.Security.Cryptography.Native…
  304 libSystem.Security.Cryptography.Native…
 5232 libclrjit.dylib
13584 libcoreclr.dylib
r@thundera app % ./bin/release/net5.0/
osx-x64/publish/app
Hello World!
r@thundera app % dotnet publish -c release
-r osx-x64 --self-contained true /p:PublishSingleFile=true
/p:PublishTrimmed=true
/p:PublishReadyToRun=true /p:IncludeNativeLibrariesForSelfEx
tract=true
r@thundera app % ls -s bin/release/net5.0/osx-x64/publish
total 49264
49240 app    24 app.pdb
```

Zooming out, self-contained single-file apps are good in the same scenarios that self-contained apps generally are: computers that you don't control and where you can't count on a runtime being installed. We expect this deployment option to show up in a lot of new places, given the greatly improved ease to-use and convenience.

Framework-Dependent Single File Apps

Framework-dependent single file apps include only your app in one executable binary. They rely on a globally installed runtime (of the right version). They're really just a small step-function optimization on framework-dependent apps, as they already exist. For example, they don't use the superhost described earlier, but the regular apphost. If you deploy apps to an environment that's always guaranteed to have the right .NET version installed, framework-dependent single file apps are the way to go. They will be much smaller than self-contained single file apps.

Let's double check that we're on the same page. A framework-dependent single file app includes the following content:

- Native executable launcher (apphost, not superhost)
- Your app + dependencies (PackageRef and ProjectRef)

What you can expect:

- The apps will be smaller, so will be quick to download.
- Startup is fast, as it's unaffected by size.
- The native launcher is native code, so the app will only work in one environment (like Linux x64, Linux ARM64, or Windows x64). You'll need to publish for each environment you want to support.
- Unlike self-contained single file apps, there'll be no additional native runtime binaries copied beside your single file app.

I'll demonstrate this experience on Windows. The experience is the same on Linux and macOS.

```
C:\Users\rich>dotnet new console -o app
"Console Application" was created
C:\Users\rich>cd app
C:\Users\rich\app>dotnet publish -r win-x64
--self-contained false /p:PublishSingleFile=true
C:\Users\rich\app>dir bin\Debug\net5.0\
win-x64\publish
 Volume in drive C has no label.
 Volume Serial Number is 9E31-D4BD

 Directory of C:\Users\rich\app\bin\Debug\net5.0\
win-x64\publish

           148,236 app.exe
             9,432 app.pdb
           2 File(s)        157,668 bytes
C:\Users\rich\app>bin\Debug\net5.0\
win-x64\publish\app.exe
Hello World!
```

We can do the same thing with an ASP.NET Core application.

```
C:\Users\rich>dotnet new webapi -o webapi
"ASP.NET Core Web API" was created
C:\Users\rich>cd webapi
C:\Users\rich\webapi>dotnet publish -r win-x64
--self-contained false /p:PublishSingleFile=true
C:\Users\rich\webapi>dir bin\Debug\net5.0\
win-x64\publish
 Volume in drive C has no label.
 Volume Serial Number is 9E31-D4BD

 Directory of C:\Users\rich\webapi\bin\Debu…

              162 appsettings.Development.json
              192 appsettings.json
              473 web.config
          267,292 webapi.exe
           19,880 webapi.pdb
           5 File(s)        287,999 bytes
```

The difference between self-contained and framework-dependent apps becomes apparent. Framework-dependent single file apps are tiny. As stated at the start of this section, they're the clear winner for environments where you can count on the runtime being installed.

A great example of being able to depend on a runtime being available is Docker. If you publish a framework-dependent single-file ASP.NET Core app with containers, you should base it on an ASP.NET image, since you will need ASP.NET Core to be provided by a lower-level image layer. Image location: https://hub.docker.com/_/microsoft-dotnet-aspnet.

Next Steps for Single File Apps

We haven't defined our final plan for .NET 6.0 yet, but we do have some ideas, some of which I've already drawn attention to. The two most obvious focus areas are enabling the superhost model for Windows and macOS in addition to Linux, and enabling first-class debugging for self-contained single file apps. You may have noticed that ASP.NET Core apps have some extra files hanging around. Those should be cleaned up and made optional. That's likely a small work-item.

Assembly trimming is an important capability for making single file apps smaller. We have implemented both conservative and aggressive modes for assembly trimming but haven't yet landed a model where we'd feel confident enabling the aggressive mode by default. This will be a continued area of focus, likely for the next couple releases.

ARM64

ARM64 is a very popular family of CPUs, designed by Arm holdings. You have an Arm chip in your phone, and it's looking like

Resources

.NET blog:
https://devblogs.microsoft.com/dotnet/

ASP.NET blog:
https://devblogs.microsoft.com/aspnet/

.NET on GitHub:
https://github.com/dotnet/core

Download .NET:
https://dot.net/

.NET on Twitter:
https://twitter.com/dotnet

.NET on Facebook:
https://www.facebook.com/dotnet/

.NET on YouTube:
https://www.youtube.com/dotnet

Assembly trimming:
https://aka.ms/dotnet5-assembly-trimming

ARM64 performance:
https://aka.ms/dotnet5-arm64-performance

.NET 5.0 performance:
https://aka.ms/dotnet5-performance

.NET container images:
https://hub.docker.com/_/microsoft-dotnet

Arm chips will become popular in laptops, too. The Surface Pro X, The Samsung Galaxy Book S and the upcoming Apple Silicon-based Mac line all use ARM64 chips. On the .NET team, we're familiar with the various ARM instruction sets and have had support for ARM with .NET Core since the 2.0 version. More recently, we've been improving ARM64 performance to ensure that .NET apps perform well in environments that rely on ARM chips. ARM chips are also popular with IoT. The Raspberry Pi 3 and Raspberry Pi 4 single board computers use ARM64 chips. Wherever ARM chips end up being used, we want .NET apps to be a good option.

Take a look at https://aka.ms/dotnet5-arm64-performance to learn more about what we've done for ARM64 in this release.

What's Special about Arm Chips?

Arm chips aren't new. They've been used in embedded scenarios for years, along with chip families like MIPS. That's why, for most people, their first awareness of Arm chips is with their phones. Arm's two big advantages are low power and low cost. No one enjoys their phone running out of power. Arm chips help to prolong battery length. Electrical usage is important in plenty of other domains, and that's why we've seen Arm chips show up in laptops more recently.

The downside of Arm chips has been lower performance. Everyone knows that a Raspberry Pi as a desktop computer isn't going to be competitive with an Intel i7 or AMD Ryzen PC. More recently, we've seen Arm chips deliver higher performance. For example, most people think of the latest iPhone or iPad from Apple as high-performance. Even though Apple doesn't use the Arm branding, the "A" in their "A" series chips could equally apply to Apple as it could to Arm. The future looks bright for Arm technology.

Hardware Intrinsics

CPUs are big blocks of silicon and transistors. The great way to get higher levels of performance is to light up as many of those transistors as possible. Soon after starting the .NET Core project, we created a new set of APIs that enabled calling CPU instructions directly. This enables low-level code to tell the JIT compiler "hey, I know exactly what I want to do; please call instruction A, then B and C, and please don't try to guess that it's something else." We started out with hardware intrinsics for Intel and AMD processors for the x86 instruction set. That worked out very well, and we saw significant performance improvements on x86 and x86-64 processors.

We started the process of defining hardware intrinsic APIs for ARM processors in the .NET Core 3.0 project. Due to schedule issues, we weren't able to finish the project at that time. Fortunately, ARM intrinsics are included in the .NET 5.0 release. Even better, their usage has been sprinkled throughout the .NET libraries, in the places where hardware intrinsics were already used. As a result, .NET code is now much faster on ARM processors, and the gap with performance on x86-64 processors is now significantly less.

ARM64 Performance Improvements

Let's take a look at some performance improvements. The following are improvements to low-level APIs. You won't necessarily call these directly from your code, but it's likely that some APIs you already use call these APIs as an implementation detail.

You can see the improvements to **System.Numerics.BitOperations** in Table 1, measured in **nanoseconds**.

System.Collections.BitArray improvements are listed in **Table 2**, measured in nanoseconds**.**

Code-Size Improvements

By virtue of generating better code for ARM64, we found that code size dropped—a lot. This affected size in memory but also the size of ready-to-run code (which affects the

BitOperations method	Benchmark	.NET Core 3.1	.NET 5	Improvements
LeadingZeroCount(uint)	LeadingZeroCount_uint	10976.5	1155.85	-89%
Log2(ulong)	Log2_ulong	11550.03	1347.46	-88%
TrailingZeroCount(uint)	TrailingZeroCount_uint	7313.95	1164.10	-84%
PopCount(ulong)	PopCount_ulong	4234.18	1541.48	-64%
PopCount(uint)	PopCount_uint	4233.58	1733.83	-59%

Table 1: Hardware intrinsics performance improvements—BitOperations

BitArray method	Benchmark	.NET Core 3.1	.NET 5	Improvements
ctor(bool[])	BitArrayBoolArrayCtor(Size: 512)	1704.68	215.55	-87%
CopyTo(Array, int)	BitArrayCopyToBoolArray(Size: 4)	269.20	60.42	-78%
CopyTo(Array, int)	BitArrayCopyToIntArray(Size: 4)	87.83	22.24	-75%
And(BitArray)	BitArrayAnd(Size: 512)	212.33	65.17	-69%
Or(BitArray)	BitArrayOr(Size: 512)	208.82	64.24	-69%
Xor(BitArray)	BitArrayXor(Size: 512)	212.34	67.33	-68%
Not()	BitArrayNot(Size: 512)	152.55	54.47	-64%
SetAll(bool)	BitArraySetAll(Size: 512)	108.41	59.71	-45%
ctor(BitArray)	BitArrayBitArrayCtor(Size: 4)	113.39	74.63	-34%
ctor(byte[])	BitArrayByteArrayCtor(Size: 512)	395.87	356.61	-10%

Table 2: Hardware intrinsics performance improvements—BitArray

Nuget package	Package version	.NET Core 3.1	.NET 5.0	Code size improvement
Microsoft.EntityFrameworkCore	3.1.6	2414572	1944756	-19.46%
HtmlAgilityPack	1.11.24	255700	205944	-19.46%
WebDriver	3.141.0	330236	266116	-19.42%
System.Data.SqlClient	4.8.1	118588	96636	-18.51%
System.Web.Razor	3.2.7	474180	387296	-18.32%
Moq	4.14.5	307540	251264	-18.30%
MongoDB.Bson	2.11.0	863688	706152	-18.24%
AWSSDK.Core	3.3.107.32	889712	728000	-18.18%
AutoMapper	10.0.0	411132	338068	-17.77%
xunit.core	2.4.1	41488	34192	-17.59%
Google.Protobuf	3.12.4	643172	532372	-17.23%

Table 3: ARM64 code-size improvements: Top NuGet packages

TechEmpower Benchmark	.NET Core 3.1	.NET 5	Improvements
JSON RPS	484,256	542,463	+12.02%
Single Query RPS	49,663	53,392	+7.51%
20-Query RPS	10,730	11,114	+3.58%
Fortunes RPS	61,164	71,528	+16.95%
Updates RPS	9,154	10,217	+11.61%
Plaintext RPS	6,763,328	7,415,041	+9.64%
TechEmpower Performance Rating (TPR)	484	538	+11.16%

Table 4: ARM64 Web throughput performance improvements: TechEmpower

size of single file apps, for example). To test that, we compared the ARM64 code produced in .NET Core 3.1 vs. .NET 5.0 for the top 25 NuGet packages (subset shown in the **Table 3**). On average, we improved the code size of ready-to-run binaries by 16.61%. **Table 3** lists NuGet packages with the associated improvement. All the measurements are in bytes (lower is better).

Straight-Out Sprint

You might be wondering how these changes play out in an actual application. If you've been following .NET performance for a while, you'll know that we use the TechEmpower benchmark to measure performance, release after release. We compared the various TechEmpower benchmarks on ARM64, for .NET Core 3.1 vs .NET 5.0. Naturally, there have been other changes in .NET 5.0 that improve the TechEmpower results. The numbers in **Table 4** represent all product changes for .NET 5.0, not just those targeted ARM64. That's OK! We'll take whatever improvements are on offer. Higher numbers are better.

Next Steps for .NET and ARM64

To a large degree, we implemented the straightforward and obvious opportunities to improve .NET performance on ARM64. As part of .NET 6.0 and beyond, we'll need users to provide us with reports of ARM64 performance challenges that we can address, and to perform much deeper and more exotic analysis of ARM64 behavior. We will also be looking for new features in the Arm instruction set that we can take advantage of.

We are, at the time of writing, enabling very early .NET 6.0 builds on Apple Silicon, on Desktop Transition Kits that Apple made available to our team. It's exciting for us to see the Arm landscape expand within the Apple ecosystem. We look forward to seeing developers taking advantage of .NET ARM64 improvements on Mac desktops and laptops.

In Closing

.NET has proven to be a truly adaptable platform, in terms of application types, deployment models, and chip architectures. We've made technical choices that enable a uniform experience across application types while offering the best of what an underlying operating system or chip architecture has to offer, with few or any compromises. We'll continue expanding the capabilities of .NET and improving performance so that your applications can run in new places and find new markets.

Just over five years ago, we announced a plan to move to an open source development model on GitHub. Many of the improvements in .NET 5.0 have come from the .NET community. This includes individuals and corporations. Thanks! A sound architecture matters, but the care and technical capability of the .NET community is the source of forward progress for the platform.

You now know more about what we're delivering with .NET 5.0. Did we make good choices? We're always listening on the dotnet/runtime repo on GitHub. Tell us what you think.

Thanks to Kunal Pathak for ARM64 performance information.

Richard Lander

Blazor Updates in .NET 5

.NET 5 comes with Blazor included, so that you have everything you need to build rich, modern Web apps with.NET and C#. .NET has long supported building high-performance server applications with ASP.NET Core. Blazor in .NET 5 enables building rich, interactive, client-side Web UIs for single page apps using .NET instead of JavaScript. With ASP.NET Core and Blazor, you can build full-stack Web apps with just .NET. Blazor in .NET 5 includes many exciting updates and improvements that will make building your next Web app simple and productive. In this article, I'll show you what Blazor in .NET 5 has to offer.



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.

A Choice of Hosting Models

Blazor apps are made up of reusable UI components. You implement Blazor components using Razor syntax, a natural mixture of HTML and C#. Blazor components handle UI events, manage their own state, and render UI updates. Blazor does the clever work of keeping track of all the rendered updates and figuring out exactly what needs to be updated in the browser DOM.

A typical Blazor Counter component that updates a displayed count each time a button is pressed looks like this:

```
<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button @onclick="IncrementCount">
    Click me
</button>

@code {
    private int currentCount = 0;

    private void IncrementCount()
    {
        currentCount++;
    }
}
```

The markup in a Blazor component consists of standard HTML. The @onclick attribute specifies a C# event handler that gets called each time the user clicks the button. The IncrementCount method updates the value of the currentCount field, and then the component renders the updated value. The Web UI updates seamlessly without you having to write a single line of JavaScript.

How Blazor handles updating the UI depends on how your components are hosted. Blazor can execute your components either on the server or client-side in the browser via WebAssembly. .NET 5 includes support for both of these hosting models.

Blazor Server apps execute your UI components on the server from within an ASP.NET Core app. When a Blazor Server app is loaded in the browser, it sets up a real-time connection back to the server using SignalR. Blazor Server uses this connection to manage all UI interactions. Blazor sends all UI events from the browser to the server over the connection and the event is dispatched to the appropriate component to handle the event. The component renders its updates, and Blazor handles serializing the exact UI changes over the SignalR connection so they can then be applied in the browser. Blazor Server apps do all the hard work of managing the UI and app state on the server, while still giving you the rich interactivity of a single-page app.

Blazor WebAssembly apps download the .NET assemblies containing your component implementations to the browser along with a WebAssembly-based .NET runtime and then execute your components and .NET code directly in the browser. From the browser, Blazor dispatches UI events to the appropriate components and then applies the UI updates from the components. All of your .NET code is executed client-side without any required server process.

.NET 5 gives you the choice of two Blazor hosting models: Blazor Server and Blazor WebAssembly. Which model you choose depends on your app requirements. **Table 1** summarizes the advantages and disadvantages of each hosting model.

Regardless of which Blazor hosting model you choose, the way you write your components is **the same**. The same components can be used with either hosting model. By using components that are hosting model agnostic, you can easily convert a Blazor app from one hosting model to the other.

.NET 5 Core Libraries

Blazor WebAssembly apps in .NET 5 have access to all the .NET 5 APIs from with the browser; you're no longer constrained to .NET Standard 2.1. The functionality of the available APIs is still subject to the limitations imposed by the browser (same origin policy, networking and file system restrictions, etc.), but .NET 5 makes many more APIs available to you, like nullability annotations and Span-based APIs.

	Blazor Server	Blazor WebAssembly
Advantages	• Full access to server capabilities • Fast to startup • Code never leaves the server • Supports older browsers and thin clients	• Runs fully client-side • No required server component • Host as a static site • Can execute offline
Disadvantages	• Requires persistent connection and UI state • Higher UI latency	• Larger download size • Slower runtime performance

Table 1: Advantages and disadvantages of the different Blazor hosting models

Blazor WebAssembly projects include a compatibility analyzer to help you know if your Blazor WebAssembly app tries to use .NET 5 APIs that are not supported in the browser.

Blazor WebAssembly in .NET 5 also uses the same core libraries used for server workloads in .NET 5. Unifying on a single implementation of the .NET core libraries is part of the single .NET vision for the .NET 5 and 6 wave. Having a single implementation of the core framework libraries provides greater consistency for app developers and makes the platform much easier to maintain.

New Blazor WebAssembly SDK

All the logic for building, linking, and publishing a Blazor WebAssembly is now packaged in a Blazor WebAssembly SDK. This new SDK replaces the functionality provided previously by the Microsoft.AspNetCore.Components.WebAssembly.Build NuGet package.

Thanks to the new SDK, the project file for a Blazor WebAssembly app in .NET 5 is simpler. It looks like this:

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

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <!-- Package references -->
  </ItemGroup>

</Project>
```

Improved WebAssembly Runtime Performance

Blazor WebAssembly apps run .NET code directly in the browser using a WebAssembly-based .NET runtime. This runtime is a based on a .NET IL interpreter without any JIT compilation support, so it generally runs .NET code much slower than what you would see from the JIT-based .NET runtime used for native app and server scenarios. For .NET 5, we've improved Blazor WebAssembly performance significantly at multiple layers of the stack. The amount of performance improvement you'll see depends on the type of code you're running.

For arbitrary CPU-intensive code, Blazor WebAssembly in .NET 5 runs about 30% faster than Blazor WebAssembly 3.2. This performance boost is mainly due to optimizations in the core framework libraries, and improvements to the .NET IL interpreter. Things like string comparisons and dictionary lookups are generally much faster in .NET 5 on WebAssembly.

Microsoft did specific optimization work for JSON serialization and deserialization on WebAssembly to speed up all those Web API calls from the browser. JSON handling when running on WebAssembly is approximately two times faster in .NET 5.

Microsoft also made several optimizations to improve the performance of Blazor component rendering, particularly for UI involving lots of components, like when using high-density grids. Component rendering in Blazor WebAssembly is two-to-four times faster in .NET 5, depending on the specific scenario.

To test the performance of grid component rendering in .NET 5, Microsoft used three different grid component implementations, each rendering 300 rows with 20 columns:

- **Fast Grid:** A minimal, highly optimized implementation of a grid
- **Plain Table:** A minimal but not optimized implementation of a grid
- **Complex Grid:** A maximal, not optimized implementation of a grid, using a wide range of Blazor features at once

Table 2 shows the performance improvements for these grid rendering scenarios in .NET 5 at the time of this writing.

Virtualization

You can further optimize your Blazor Web UI by taking advantage of the new built-in support for virtualization. Virtualization is a technique for limiting the number of rendered component to just the ones that are currently visible, like when you have a long list or table with many items and only a small subset is visible at any given time. Blazor in .NET 5 adds a new Virtualize component that can be used to easily add virtualization to your components.

A typical list or table-based component might use a C# foreach loop to render each item in the list or each row in the table, like this:

```
@foreach (var employee in employees)
{
    <tr>
        <td>@employee.FirstName</td>
        <td>@employee.LastName</td>
        <td>@employee.JobTitle</td>
    </tr>
}
```

As the size of the list gets large (companies do grow!) rendering all the table rows this way may take a while, resulting in a noticeable UI delay.

Instead, you can replace the **foreach** loop with the Virtualize component, which only renders the rows that are currently visible.

```
<Virtualize Items="employees" ItemSize="40"
 Context="employee">
    <tr>
        <td>@employee.FirstName</td>
        <td>@employee.LastName</td>
        <td>@employee.JobTitle</td>
```

One .NET

Starting with .NET 5, the .NET team is working toward unifying the various .NET implementations for cloud, desktop, mobile, and devices into a single unified .NET platform. This work includes combining the best parts of .NET Core and Mono. Switching Blazor WebAssembly to use the .NET 5 core libraries instead of the Mono libraries is part of this effort.

You can learn more about the .NET team's efforts to unify the .NET Platform by watching Scott Hunter and Scott Hanselman's "The Journey to One .NET" presentation from Microsoft BUILD 2020: https://aka.ms/onedotnet

	Fast Grid	Plain Table	Complex Grid
3.2.0	162ms	490ms	1920ms
5.0 Preview 8	62ms	291ms	1050ms
5.0 RC1	52ms	255ms	780m
Improvement	**3.1x**	**1.9x**	**2.5x**

Table 2: Blazor WebAssembly performance improvements for different grid implementations.

```
    </tr>
</Virtualize>
```

The Virtualize component calculates how many items to render based on the height of the container and the size of the rendered items in pixels. You specify how to render each item using the ItemContent template or with child content. If the rendered items end up being slightly off from the specified size, the Virtualize component adjusts the number of items rendered based on the previously rendered output.

If you don't want to load all items into memory, you can specify an ItemsProvider, like this:

```
<Virtualize ItemsProvider="LoadEmployees"
 ItemSize="40" Context="employee">

    <tr>
        <td>@employee.FirstName</td>
        <td>@employee.LastName</td>
        <td>@employee.JobTitle</td>
    </tr>
</Virtualize>
```

An items provider is a delegate method that asynchronously retrieves the requested items on demand. The items provider receives an ItemsProviderRequest, which specifies the required number of items starting at a specific start index. The items provider then retrieves the requested items from a database or other service and returns them as an ItemsProviderResult<TItem> along with a count of the total number of items available. The items provider can choose to retrieve the items with each request, or cache them so they are readily available.

```
async ValueTask<ItemsProviderResult<Employee>>
 LoadEmployees(ItemsProviderRequest request)
{
    var numEmployees = Math.Min(request.Count,
 totalEmployees - request.StartIndex);
    var employees = await EmployeesService
        .GetEmployeesAsync(request.StartIndex,
 numEmployees, request.CancellationToken);
    return new ItemsProviderResult<Employee>(
 employees, totalEmployees);
}
```

Because requesting items from a data source might take a bit, you also have the option to render a placeholder until the item is available.

```
<Virtualize ItemsProvider="LoadEmployees"
 ItemSize="40" Context="employee">
    <ItemContent>
        <tr>
            <td>@employee.FirstName</td>
            <td>@employee.LastName</td>
            <td>@employee.JobTitle</td>
        </tr>
    </ItemContent>
    <Placeholder>
        <tr>
            <td>Loading...</td>
        </tr>
    </Placeholder>
</Virtualize>
```

Prerendering for Blazor WebAssembly

Prerendering your Blazor app on the server can significantly speed up the perceived load time of your app. Prerendering works by rendering the UI on the server in response to the first request. Prerendering is also great for search engine optimization (SEO), as it makes your app easier to crawl and index.

Blazor Server apps already have support for prerendering through the component tag helper. The Blazor Server project template is set up by default to prerender the entire app from the Pages/_Host.cshtml page using the component tag helper.

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

The component tag helper renders the specified Blazor component into the page or view. Previously, the component tag helper only supported the following rendering modes:

- **ServerPrerendered:** Prerenders the component into static HTML and includes a marker for a Blazor Server app to later use to make the component interactive when loaded in the browser.
- **Server:** Renders a marker for a Blazor Server app to use to include an interactive component when loaded in the browser. The component is not prerendered.
- **Static**: Renders the component into static HTML. The component is not interactive.

In .NET 5, the component tag helper now supports two additional render modes for prerendering a component from a Blazor WebAssembly app:

- **WebAssemblyPrerendered:** Prerenders the component into static HTML and includes a marker for a Blazor WebAssembly app to later use to make the component interactive when loaded in the browser.
- **WebAssembly:** Renders a marker for a Blazor WebAssembly app to use to include an interactive component when loaded in the browser. The component is not prerendered.

To set up prerendering in a Blazor WebAssembly app, you first need to host the app in an ASP.NET Core app. Then, replace the default static index.html file in the client project with a _Host.cshtml file in the server project and update the server startup logic to fallback to the new page instead of index.html (similar to how the Blazor Server template is set up). Once that's done, you can prerender the root App component like this:

```
<component type="typeof(App)"
 render-mode="WebAssemblyPrerendered" />
```

In addition to dramatically improving the perceived load time of a Blazor WebAssembly app, you can also use the component tag helper with the new render modes to add multiple components on different pages and views. You don't need to configure these components as root components in the app or add your own market tags on the page—the framework handles that for you.

You can also pass parameters to the component tag helper when using the WebAssembly-based render modes if the parameters are serializable.

```
<component type="typeof(Counter)"
 render-mode="WebAssemblyPrerendered"
 param-IncrementAmount="10" />
```

The parameters must be serializable so that they can be transferred to the client and used to initialize the component in the browser. You'll also need to be sure to author your components so that they can gracefully execute server-side without access to the browser.

CSS Isolation

Blazor now supports CSS isolation, where you can define styles that are scoped to a given component. Blazor applies component-specific styles to only that component without polluting the global styles and without affecting child components. Component-specific CSS styles 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 component-specific styles in a .razor.css file that matches the name of the .razor file for the component. For example, let's say you have a component MyComponent.razor file that looks like this:

```
<h1>My Component</h1>

<ul class="cool-list">
    <li>Item1</li>
    <li>Item2</li>
</ul>
```

You can then define a MyComponent.razor.css with the styles for MyComponent:

```
h1 {
    font-family: 'Comic Sans MS'
}

.cool-list li {
    color: red;
}
```

The styles in MyComponent.razor.css only get applied to the rendered output of MyComponent; the h1 elements rendered by other components, for example, are not affected.

To write a selector in component specific styles that affects child components, use the **::deep** combinator.

```
.parent ::deep .child {
    color: red;
}
```

By using the **::deep** combinator, only the .parent class selector is scoped to the component; the .child class selector isn't scoped, and matches content from child components.

Blazor achieves CSS isolation by rewriting the CSS selectors as part of the build so that they only match markup rendered by the component. Blazor adds component-specific attributes to the rendered output and updates the CSS selectors to require these attributes. Blazor then bundles together the rewritten CSS files and makes the bundle available to the app as a static Web asset at the path [LIBRARY NAME].styles.css. Although Blazor doesn't natively support CSS preprocessors like Sass or Less, you can still integrate CSS preprocessors with Blazor projects to generate component specific styles before they're rewritten as part of the Blazor build system.

Lazy Loading

Lazy loading enables you to improve the load time of your Blazor WebAssembly app by deferring the download of some assemblies until they are required. For many apps, lazy loading different parts of the app isn't necessary. The .NET IL that makes up .NET assemblies is very compact, especially when its compressed. You may be surprised by how much code you have to write in your app before it significantly impacts the app size. The download size of a Blazor WebAssembly app is typically dominated by the size of the runtime and core framework libraries, which Blazor aggressively trims to remove unused code. Lazy loading may be helpful if your Blazor WebAssembly app grows exceptionally large or you have parts of your app with large dependencies that aren't used elsewhere and can't be reasonably reduced through IL trimming.

Normally, Blazor downloads and loads all dependencies of the app when it's first loaded. To delay the loading of a .NET assembly, you add it to the BlazorWebAssemblyLazyLoad item group in your project file:

```
<BlazorWebAssemblyLazyLoad Include="Lib1.dll" />
```

Assemblies marked for lazy loading must be explicitly loaded by the app before they're used. To lazy load assemblies at runtime, use the LazyAssemblyLoader service:

```
@inject LazyAssemblyLoader LazyAssemblyLoader

@code {
    var assemblies = await LazyAssemblyLoader
        .LoadAssembliesAsync(new string[]
        {
            "Lib1.dll"
        });
}
```

Often, assembles need to be loaded when the user navigates to a particular page. The Router component has a new OnNavigateAsync event that's fired on every page navigation and can be used to lazy load assemblies for a particular route. You can also lazily load the entire page for a route by passing any loaded assemblies as additional assemblies to the Router.

You can see a full example of integrating lazy loading with the Router component in **Listing 1**.

Less JavaScript, More C#

Blazor's purpose is to enable building rich interactive Web UIs with .NET, but sometimes you still need some JavaScript. This might be because you want to reuse an existing library or because Blazor doesn't yet expose a native browser capability that you need. Blazor supports JavaScript interop, where you can call into any JavaScript code from your .NET code, but writing JavaScript interop code with Blazor should be rare. In .NET 5, Microsoft has added some new built-in features that reduce or eliminate the amount of JavaScript interop code required for some common scenarios.

Managing App State in Blazor

Blazor WebAssembly and Blazor Server have some important differences with how they handle app state. In a Blazor WebAssembly app, all of the app state lives in the browser on the client device. If the browser is running, the app state is available in memory.

Blazor Server apps, however, maintain all of the app state for all connected users on the server. If the browser connection to the server is lost, the UI can't function. If the server process goes down, all of the app state held in memory is lost unless it has otherwise been persisted. A common approach for persisting app state is to leverage the local storage available in the user's browser. The new protected local and session storage support in .NET 5 makes persisting app state from Blazor Server apps much easier.

Listing 1: Integrating lazy loading with the Blazor Router

```
@using System.Reflection
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.WebAssembly.Services
@inject LazyAssemblyLoader LazyAssemblyLoader

<Router AppAssembly="@typeof(Program).Assembly"
        AdditionalAssemblies="@lazyLoadedAssemblies"
        OnNavigateAsync="@OnNavigateAsync">
    <Navigating>
        <div>
            <p>Loading the requested page...</p>
        </div>
    </Navigating>
    <Found Context="routeData">
        <RouteView RouteData="@routeData"
                   DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <LayoutView Layout="@typeof(MainLayout)">
            <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>

@code {
    private List<Assembly> lazyLoadedAssemblies =
        new List<Assembly>();

    private async Task OnNavigateAsync(NavigationContext args)
    {
        if (args.Path.EndsWith("/page1"))
        {
            var assemblies = await LazyAssemblyLoader
                .LoadAssembliesAsync(new string[]
                {
                    "Lib1.dll"
                });
            lazyLoadedAssemblies.AddRange(assemblies);
        }
    }
}
```

Set UI Focus

Sometimes you need to set the focus on a UI element programmatically. Blazor in .NET 5 now has a convenient method on ElementReference for setting the UI focus on that element.

```
<button @onclick="() => textInput.FocusAsync()">
    Set focus
</button>
<input @ref="textInput"/>
```

File Upload

Blazor now offers an InputFile component for handling file uploads. The InputFile component is based on an HTML input of type "file". By default, you can upload single files, or you can add the "multiple" attribute to enable support for multiple files. When one or more files is selected for upload, the InputFile component fires an OnChange event and passes in an InputFileChangeEventArgs that provides access to the selected file list and details about each file.

```
<InputFile OnChange="OnInputFileChange"
 multiple />
```

To read a file, you call OpenReadStream on the file and read from the returned stream. In a Blazor WebAssembly app this reads the file into memory on the client. In a Blazor Server app, the file is transmitted to the server and read into memory on the server. Blazor also provides a ToImageFileAsync convenience method for resizing images files before they're uploaded.

The example code in **Listing 2** shows how to resize and reformat user selected images as 100x100 pixel PNG files. The example code then uploads the resized images and displays them as data URLs.

Influencing the HTML Head

Use the new Title, Link, and Meta components to programmatically set the title of a page and dynamically add link and meta tags to the HTML head in a Blazor app. To use these components, add a reference to the Microsoft.AspNetCore.Components.WebExtensions NuGet packages.

The following example programmatically sets the page title to show the number of unread user notifications, and updates the page icon a as well:

```
@if (unreadNotificationsCount > 0)
{
    var title =
 $"Notifications ({unreadNotificationsCount})";
    <Title Value="title"></Title>
    <Link rel="icon" href="icon-unread.ico" />
}
```

Protected Browser Storage

In Blazor Server apps, you may want to persist the app state in local or session storage so that the app can rehydrate it later if needed. When storing app state in the user's browser, you also need to ensure that it hasn't been tampered with. Blazor in .NET 5 helps solve this problem by providing two new services: ProtectedLocalStorage and ProtectedSessionStorage. These services help you store state in local and session storage respectively, and they take care of protecting the stored data using the ASP.NET Core data protection APIs.

To use the new services, simply inject either of them into your component implementations:

```
@inject ProtectedLocalStorage LocalStorage
@inject ProtectedSessionStorage SessionStorage
```

You can then get, set, and delete state asynchronously.

```
private async Task IncrementCount()
{
    await LocalStorage.SetAsync("count",
 ++currentCount);
}
```

JavaScript Isolation and Object References

When you do need to write some JavaScript for your Blazor app, Blazor now enables you to isolate your JavaScript as standard JavaScript modules. This has a couple of benefits: imported JavaScript no longer pollutes the global namespace, and consumers of your library and components no longer need to manually import the related JavaScript.

For example, the following JavaScript module exports a simple JavaScript function for showing a browser prompt:

Download the Code

You can download the sample code from this article at https://aka.ms/blazor-net5-samples.

Listing 2: Using InputFile to resize and upload multiple images

```
<div class="image-list">
    @foreach (var imageDataUrl in imageDataUrls)
    {
        <img src="@imageDataUrl" />
    }
</div>

@code {
    ElementReference textInput;

    IList<string> imageDataUrls = new List<string>();

    async Task OnInputFileChange(InputFileChangeEventArgs e)
    {
        var imageFiles = e.Files
            .Where(file => file.Type.StartsWith("image/"));
        var format = "image/png";
        foreach (var imageFile in imageFiles)
        {
            var resizedImageFile =
                await imageFile.ToImageFileAsync(format, 100, 100);
            var buffer = new byte[resizedImageFile.Size];
            await resizedImageFile.OpenReadStream()
                .ReadAsync(buffer);
            var imageDataUrl =
 $"data:{format}; base64,{Convert.ToBase64String(buffer)}";
            imageDataUrls.Add(imageDataUrl);
        }
    }
}
```


SPONSORED SIDEBAR:

Your Legacy Apps Stuck in the Past?

Get FREE advice on migrating your legacy application to today's modern platforms. Leverage CODE Consulting's years of experience migrating legacy applications by contacting us today to schedule your free hour of CODE consulting call. No strings. No commitment. Nothing to buy. For more information, visit www.codemag.com/consulting or email us at info@codemag.com.


```
export function showPrompt(message) {
    return prompt(message, 'Type anything here');
}
```

You can add this JavaScript module to your .NET library as a static Web asset (wwwroot/exampleJsInterop.js) using the Razor SDK and then import the module into your .NET code using the IJSRuntime service:

```
var module = await jsRuntime
    .InvokeAsync<IJSObjectReference>("import",
 "./_content/MyComponents/exampleJsInterop.js");
```

The "import" identifier is a special identifier used specifically for importing the specified JavaScript module. You specify the module using its stable static Web asset path: _content/[LIBRARY NAME]/[PATH UNDER WWWROOT].

The IJSRuntime imports the module as an IJSObjectReference, which represents a reference to a JavaScript object from .NET code. You can then use the IJSObjectReference to invoke exported JavaScript functions from the module:

```
public async ValueTask<string> Prompt(
 string message)
{
    return await module.InvokeAsync<string>(
 "showPrompt", message);
}
```

Debugging Improvements

Blazor WebAssembly debugging is significantly improved in .NET 5. Launching a Blazor WebAssembly app for debugging is now much faster and more reliable. The debugger now breaks on unhandled exceptions in your code. Microsoft has enabled support for debugging into external dependencies. The Blazor WebAssembly debug proxy has also been moved into its own process to enable future work to support debugging Blazor WebAssembly apps running in remote environments like Docker, Windows Subsystem for Linux, and Codespaces.

See UI Updates Faster with dotnet watch

When building UIs, you need to be able to see your changes as fast as possible. .NET 5 makes this easy with some nice improvements to the **dotnet watch** tool. When you run **dotnet watch run** on a project, it watches your files for code changes and then rebuilds and restarts the app so that you can see the results of your changes quickly. New in .NET 5, **dotnet watch** launches your default browser for you once the app is started and auto refreshes the browser as you make changes. This means that you can open your Blazor project (or any ASP.NET Core project) in your favorite text editor, run **dotnet watch run** once, and then focus on your code changes while the tooling handles rebuilding, restarting, and reloading your app. Microsoft expects to bring the auto refresh functionality to Visual Studio as well.

> Now, dotnet watch launches your default browser for you once the app is started and auto refreshes the browser as you make changes.

More Blazor Goodies

Although this article summarizes most of the major new Blazor features in .NET 5, there are still plenty of other smaller Blazor enhancements and improvements to check out. These improvements include:

- IAsyncDisposable support
- Control Blazor component instantiation
- New InputRadio component
- Support for catch-all route parameters
- Support for toggle events
- Parameterless InvokeAsync overload on EventCallback
- Blazor Server reconnection improvements
- Prerendering for Blazor WebAssembly apps

Many of these features and improvements came from the enthusiastic community of open source contributors. Microsoft greatly appreciates all of the preview feedback, bug reports, feature suggestions, doc updates, and design, and code reviews from countless individuals. Thank you to everyone who helped with this release. I hope you enjoy Blazor in .NET 5! Give Blazor a try today by going to https://blazor.net.

Daniel Roth

Windows Desktop Apps and .NET 5

This November, Microsoft is releasing .NET 5—the unified .NET platform that will be the only .NET platform moving forward. It has support for all kinds of .NET applications, including the Windows desktop platforms such as WPF, Windows Forms, and WinUI 3.0. In this article, I'll talk about what .NET 5 means for desktop developers and how to migrate your existing applications to .NET 5.

Why .NET 5 for Desktop Applications?

Let's start with "what is .NET 5?" Over the last few years, .NET developers have had a variety of .NET platforms on which to build their applications:

- **.NET Framework**: The oldest platform which, since its inception in 2002, evolved into a large codebase that worked only on Windows and had a support for various .NET technologies from desktop to Web. It was a great platform for developing .NET applications for many years. Since .NET Framework was first released, the tech world has changed a lot. What was once state-of-the-art became a limitation. Things like open source, cross-platform, improved performance, etc., became new "must-haves," which .NET Framework couldn't provide due to the way it was designed. That's why we released .NET Core.
- **.NET Core**: Created in 2016 as a new open source and cross-platform .NET. Initially, .NET Core supported only Web and microservice stacks until 2019, when all other technologies were brought to .NET Core as well.
- **Mono**: Initially started as a third-party implementation of the .NET Framework for Linux.
- **.NET Standard:** Isn't a platform but a specification, which has a common denominator for APIs of all platforms. It's another option for developers when it comes to target frameworks for libraries.

A variety of options isn't always a blessing but is sometimes a curse. Many people were confused and not sure what they should choose for their applications. Moving forward, there will be just one .NET, called .NET 5, that aggregates the best of all .NET platforms and supports all .NET technologies (**Figure 1**).

The preview version of .NET 5 is already available and in November 2020, the full .NET 5 version will be released! You can get it at https://dotnet.microsoft.com/download/dotnet/5.0.

Benefits of .NET 5 Compared to .NET Framework

There are many benefits in .NET 5 compared to .NET Framework. First, .NET 5 will be the future of .NET, which means that all new improvements, .NET APIs, runtime capabilities, and language features will go exclusively to .NET 5. .NET Framework will remain where it is at the highest version of 4.8 and receive only Windows essential (Windows and security) updates. All new feature development will happen in .NET 5.

Another very important feature of .NET 5 is that, like .NET Core, it allows "side-by-side" deployment of multiple versions of .NET 5 on the same computer. In your applications, you can specify which version of .NET you want to target. That means your application will never be broken by a random runtime update beyond your control. You can also package a specific version of .NET 5 with your application and be completely independent of your user's computer set up.

Because of this major improvement, you're able to innovate at much higher pace (without fear of breaking existing applications) and introduce many new features and improvements such as:

- Smaller applications sizes with Assembly Trimming feature
- Single-file executables
- Significant BCL performance improvements
- More choice on runtime experiences
- Better project files (SDK-style .csproj)
- New features in Windows Forms, WPF, C#, etc.
- Stable release schedule

You can read more about each area on Microsoft's .NET blog site at: https://devblogs.microsoft.com/dotnet/, where we post all the updates we make in .NET. As for the last point, from now on, .NET will be released once a year and every even version number will be a long-term support version. The schedule is shown in **Figure 2**.

Differences Between .NET 5 and .NET Core 3.x

.NET 5 is based on .NET Core 3.1, which means that all APIs, runtime, and performance improvemens that are in .NET Core 3.1 are also included in .NET 5. Plus, there are some additional features such as:

- New updates in Windows Forms, which will be covered later in this article
- Many improvements in WPF XAML tooling; also described below.
- Performance improvements that you can read about at https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-5/.
- Many runtime improvements, like usability features to the new JSON APIs, Garbage Collection improvements, better performance for assembly code generation RyuJIT, improvements to single-file deployment, and oth-



Figure 1: Structure of .NET 5



Olia Gavrysh
oliag@microsoft.com
devblogs.microsoft.com/dotnet/
@oliagavrysh

Olia Gavrysh is a program manager on the .NET team at Microsoft. She focuses on desktop developer tools. With the latest version of .NET Core and .NET 5 coming soon, a big part of her work centers around porting to .NET Core experience.

Olia has a background in software development and machine learning. Before becoming a PM, she spent 10 years writing .NET applications. When she joined Microsoft, she worked on the .NET Framework for machine learning called ML.NET. She's a frequent speaker at international developer conferences and a contributor to .NET Blog.

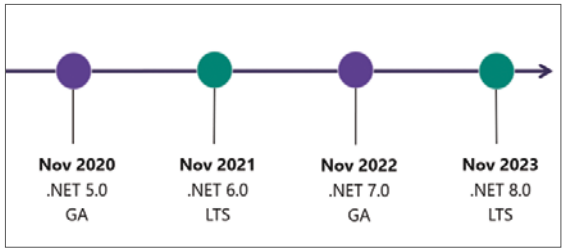


Figure 2: Schedule of .NET releases.

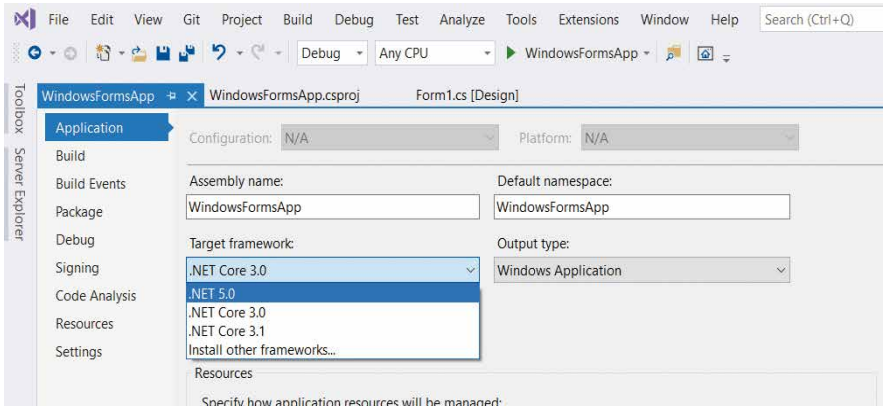


Figure 3: The Project Properties window where you can choose a target framework.

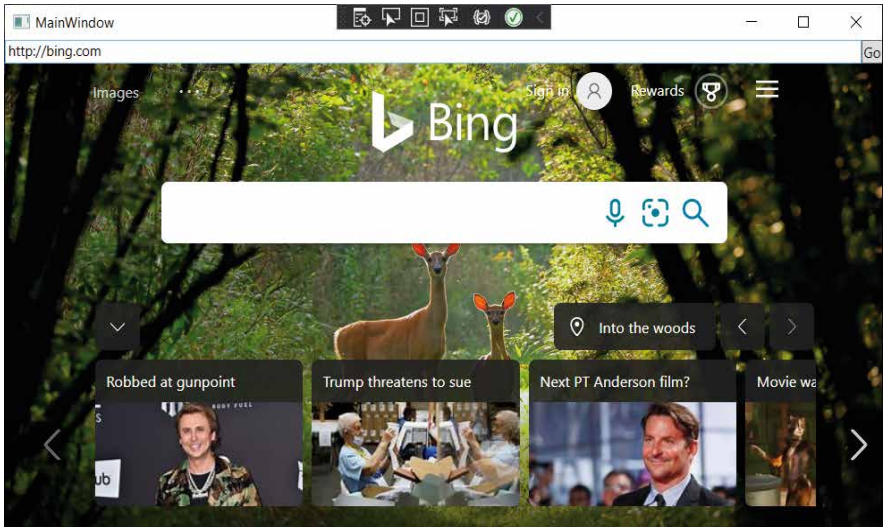


Figure 4: The WebView2 control in Windows Forms application.

ers can be read about at .NET Blog: https://devblogs.microsoft.com/dotnet.

There were some breaking changes in Windows Forms between .NET Core 3.0, .NET Core 3.1, and .NET 5 but most of them are very easy to fix by upgrading to a newer and better APIs. You can learn about all those changes here:

- Breaking changes between .NET Core 3.1 and .NET 5: https://docs.microsoft.com/en-us/dotnet/core/compatibility/3.1-5.0#windows-forms
- Breaking changes between .NET Core 3.0 and .NET Core 3.1: https://docs.microsoft.com/en-us/dotnet/core/compatibility/3.0-3.1#windows-forms

If you're creating new WPF or Windows Forms applications, build them on top of .NET 5! If your application was already created on .NET Core or .NET Framework, you can port it to the new platform. In the next section, I'll talk about porting.

Porting Existing WinForms and WPF Applications to .NET 5

Now's the time to port your Windows Forms and WPF applications to .NET 5. There are a few cases when staying on the .NET Framework is reasonable. For example, if your application is released, the development is completed, and you have no intention to touch the code (except maybe for bug fixes or servicing), in such a case, it's totally fine to leave your application targeting .NET Framework. But if you're doing any active development, I strongly recommend porting your application to .NET 5 so you'll be able to benefit from a huge amount of improvements introduced in the new platform. Let's see how to port your application from .NET Core and .NET Framework.

Porting from .NET Core

If your application targets .NET Core 3.0 or 3.1, porting it to .NET 5 should be easy. Right-click on the project file that you want to port in the Solution Explorer from the context menu select "Properties...". You'll see the Properties window shown in **Figure 3**.

The combo-box Target framework contains the name and version of the .NET platform you're targeting right now. In this case, it's .NET Core 3.0. Choose .NET 5.0 and save the project. You have ported to .NET 5.0!

There were some minor breaking changes between .NET Core 3.0, 3.1 and .NET 5. You can read about them at https://docs.microsoft.com/en-us/dotnet/core/compatibility/3.0-3.1#windows-forms and https://docs.microsoft.com/en-us/dotnet/core/compatibility/3.1-5.0#windows-forms. So if your application was using any of the old APIs that got changed in .NET 5, you'll need to do a minor refactoring to your code to update it with the latest APIs. Usually, it's straightforward.

Porting from .NET Framework

Unlike porting from .NET Core, which was the foundation for .NET 5, the difference between .NET Framework and .NET 5 is significant. So, in the case of porting from .NET Framework to .NET 5, you can't just change the target framework in the Project Properties window. We have a tool that can help you with the migration, called Try Convert.

Try Convert is a global tool that attempts to upgrade your project file from the old style to the new SDK style and retargets applicable projects to .NET 5. You can install the tool from here: https://github.com/dotnet/try-convert/releases. Once installed, in CLI, run the command:

```
try-convert -p "<path to your project file>"
```

Or:

```
try-convert -w "<path to your solution>"
```

After the tool completes the conversion, reload your files in Visual Studio. There's a possibility that Try Convert won't be able to perform the conversion due to specifics of your project. In that case, refer to our documentation here:

```
132    <Label Content="{Binding Name}" d:Content="Name!" />
133    <Button Content="{Binding Path=ButtonContent}" d:Content="Design Time Content" d:IsEnabled="false" Width="125" Heigh
```

Figure 5: Using Design-time Data for different control properties.

https://docs.microsoft.com/dotnet/core/porting/ and on the last step, set TargetFramework to **net5.0-windows.**

```
<TargetFramework>
          net5.0-windows
</TargetFramework>
```

Over time, some pieces of .NET Framework became obsolete and have been replaced with newer substitutions. We didn't port those old parts to .NET Core and .NET 5; some of them because they were no longer relevant, and some because they didn't fit into the new architecture of the platform. You can read more about it in the article from last year: https://www.codemag.com/Article/1911032/Upgrading-Windows-Desktop-Applications-with-.NET-Core-3, in the section "What Doesn't Work in .NET Core Out of the Box?"

What's New in .NET Desktop?

Once you've upgraded your application to the .NET 5 platform, you have access to all the latest improvements and updates. There are really a lot, and I want to highlight just a few of them related to desktop development.

Bring the Web to Your Desktop with WebView2

Microsoft is happy to announce a release of WebView2, the new browser control that allows you to render Web content (HTML/CSS/JavaScript) with the new Chromium-based Microsoft Edge in your desktop applications (shown on **Figure 4**). In November 2020, it becomes available for Windows Forms and WPF applications targeting .NET 5, .NET Core 3.x, and .NET Framework.

You're probably familiar with the predecessor of WebView2, the WebBrowser control that's been around for quite some time and is based on Internet Explorer. Then we built an Edge-based version of it called WebView. The new control was a big step forward but still had its limitations; it couldn't be embedded in your WPF and Windows Forms application and worked only on Windows 10. With the upgraded version of it, WebView2, we're removing those limitations. Now it's consistent across all Windows versions starting from Windows 7, has the latest improvements of the browser, and will receive updates every six weeks, allowing you to always stay on top of the most recent tech. For more details, check out the documentation and getting started guides at http://aka.ms/webview2.

WebView 2 is supported for the following platforms: Win32 C/C++, .NET Framework 4.6.2 or later, .NET Core 3.0 or later, and WinUI 3.0. And on the following Windows versions: Windows 10, Windows 8.1, Windows 8, Windows 7, Windows Server 2016, Windows Server 2012, Windows Server 2012 R2, and Windows Server 2008 R2.

Updates in XAML Designer

This year, we've been working on many new features for XAML Designer. Here are the most important ones.

Design-time Data. Very often, controls in WPF application are populated with data via data binding. This means that you don't see how your controls look with data in the designer before you run your application. In the early stages of the development, you might not even have the data source or your ViewModel yet. Thanks to the Design-time Data feature, your controls can be populated with "dummy" data visible only in the designer. This data won't be compiled into your binaries, so you don't have to worry about accidently shipping your applications with the test values.

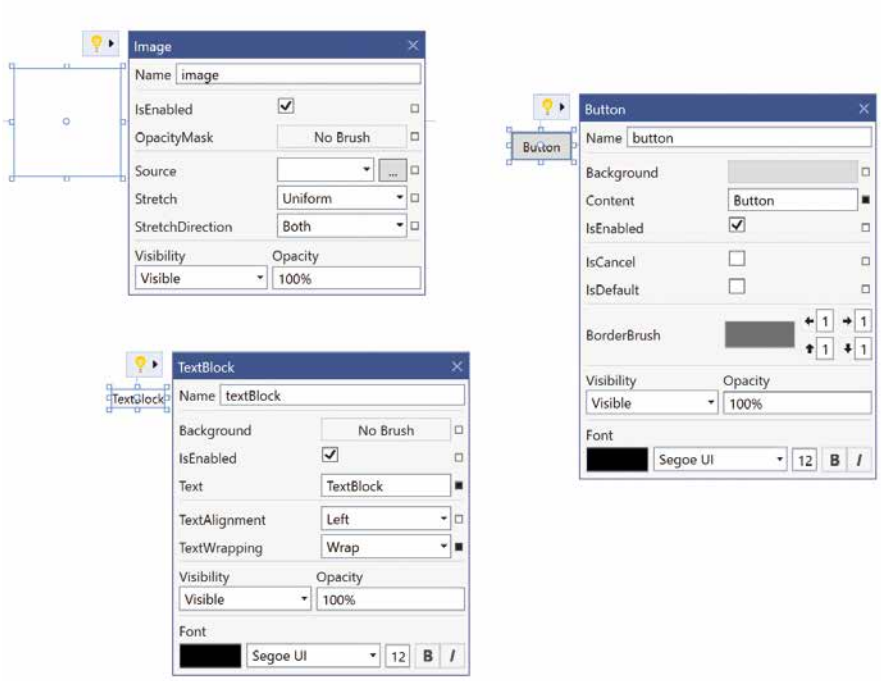


Figure 6: Suggested Actions for Image, Button, and TextBlock in XAML designer

For each XAML property for built-in controls, you can set a value using **d: prefix**, as shown in the **Figure 5**.

This way, you'll see values assigned to the properties with **d: prefix** in the design time, and without **d:** in the runtime (in your binaries).

Suggested Actions. Now when developing your UI for WPF (on .NET 5 and .NET Core) and UWP applications, you can quickly access the most commonly used properties and actions of a selected control in the designer by expanding the icon with a lightbulb next to it, as shown in **Figure 6**.

To use this feature, enable it in Visual Studio navigate to **Tools** > **Options** > **Preview Features** and check "**XAML Suggested Actions**". Currently, it works for standard built-in controls. We will keep working on improving this feature.

Binding Failures Troubleshooting Tools. We've heard your feedback about difficulties with troubleshooting binding failures. That's why we added a few things to improve your experience. First, you'll see right away if your application encountered any binding failures in the in-app toolbar. The binding failures icon will turn red and have a number of failures next to a red cross. In **Figure 7,** you can see 32 binding failures. Clicking the icon takes you to the new XAML Binding Failures window that has sorting, searching, grouping, and other useful features.

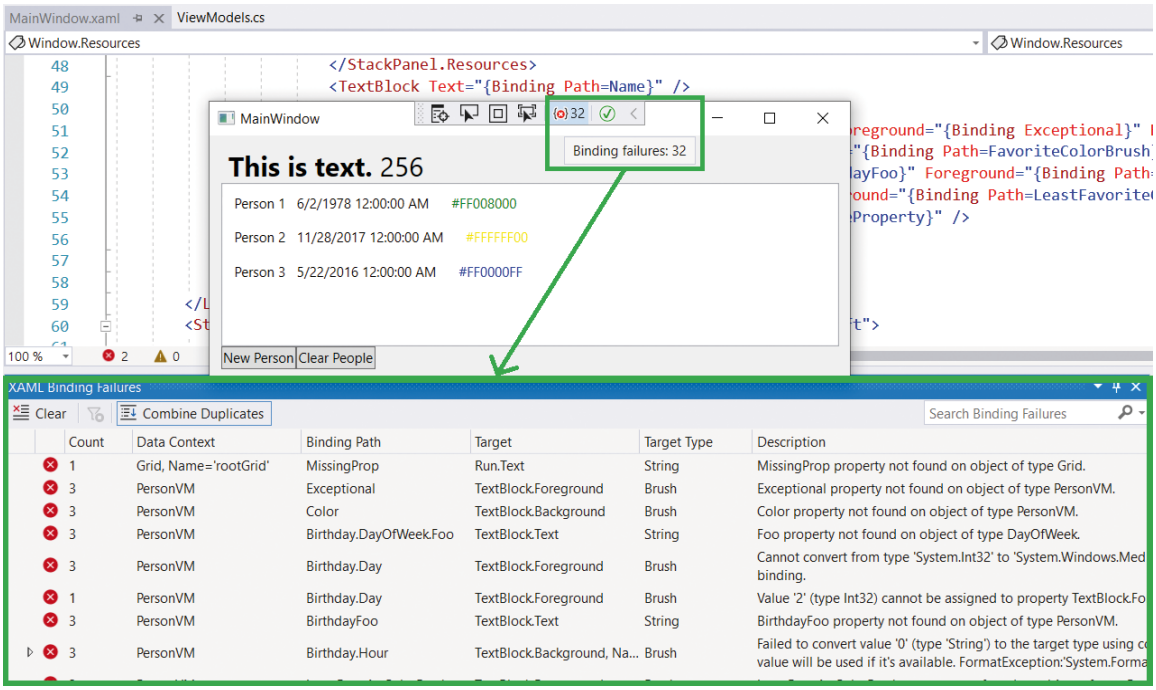


Figure 7: Binding Failures indicator and window

```
<Border BorderBrush='GreenYellow' HorizontalAlignment='Center' BorderThickness='1' >
    <Border.Background>
        <LinearGradientBrush EndPoint='0.5,1' StartPoint='0.5,0'>
            <GradientStop Offset='0' Color='#FFF1F104'/>
            <GradientStop Offset='1' Color='#FFF0C2EE'/>
```

Figure 8: Binding Failures indicator and window



Figure 9: XAML Designer Refresh button

This feature is also in the Preview, so to enable it, in Visual Studio navigate to **Tools** > **Options** > **Preview Features** and check **XAML Binding Failure Window**.

XAML code editor improvements. We added a few improvements in the code editor, such as:

- The ability to drag an image from Solution Explorer into the XAML editor, which generates a proper Image tag with the full path to the dragged image.
- An inline color preview right in your XAML code next to the color name or code (**Figure 8**).

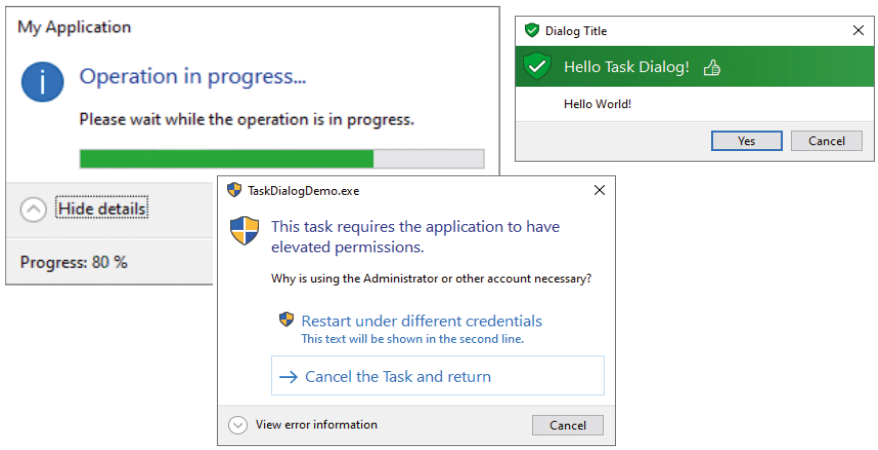


Figure 10: Windows Forms task Dialog

New XAML Designer for .NET Framework applications. We're bringing all the new features and extensibility support of XAML .NET Core designer for .NET Framework applications as well. Although the experience working with the designer will remain very similar, you'll notice faster load performance, 64-bit configuration support (custom controls will load normally just like they do in x86 configuration), bug fixes, and the new features described above.

To upgrade to the new designer for .NET Framework applications, in Visual Studio navigate to **Tools > Options > Preview Features** and check **New WPF XAML Designer for .NET Framework** and restart Visual Studio.

XAML Designer Refresh Button. If you face a rendering issue, there's no need to close and re-open the designer view. Now you can just click the Refresh button, as shown in **Figure 9**.

Updates in Windows Forms

By enabling Windows Forms on .NET Core and .NET 5, we manifest our commitment to innovate in Windows Forms! Even though most of the team's effort this year was focused around enabling Widows Forms designer for .NET 5 projects, we had a chance to add a few new features. And for that we owe a big "thank you" to our OSS community! One of the most exciting features was submitted by our open source contributor Konstantin Preisser. This was a Task Dialog that allows you to add customizable dialogs to your Windows Forms applications as shown in **Figure 10**.

We'd like to thank Tobias Käs, Hugh Bellamy, and many others who have been improving Windows Forms with us!

Explore New .NET desktop: WinUI 3

In addition to Windows Forms and WPF, .NET 5 includes support for WinUI 3—the evolution of UWP's XAML technology. It includes the entire presentation layer (XAML, composition, and input) decoupled from the Windows 10 OS, an ex-

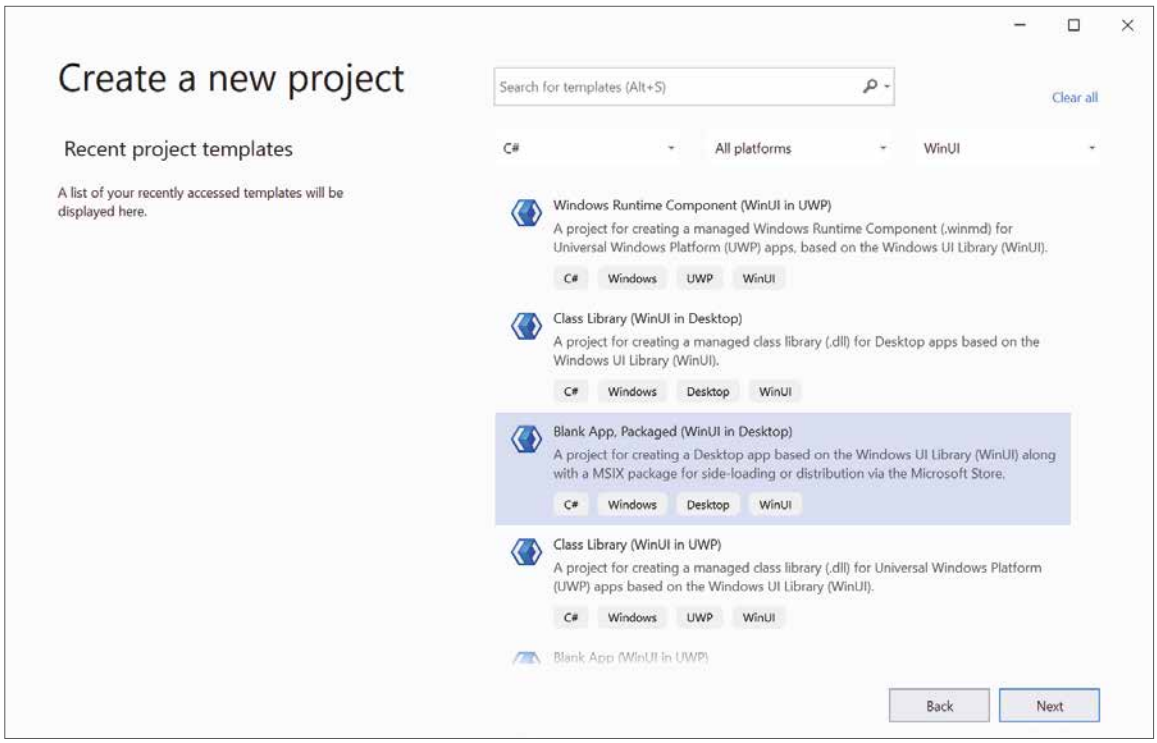


Figure 11: The WinUI template in the New Project dialog

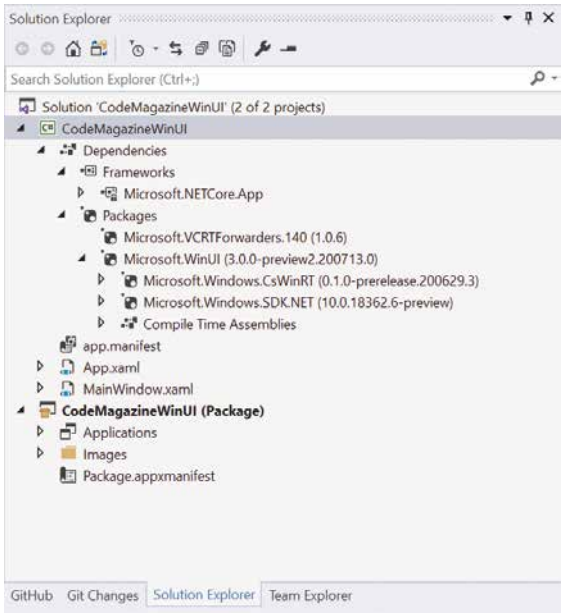


Figure 12: WinUI 3 default project in Solution Explorer

tensive controls library (including WebView2), and allows you to use the desktop application model in addition to the UWP app model.

Because WinUI 3 evolves from UWP, it supports various input types such as touch, mouse, keyboard, gamepad, and others, and allows different types of form factors and DPIs. Because accessibility was our top priority, WinUI 3 has very good integration with the Narrator tool. It also has good performance, and you can blend DirectX and XAML content to get even better performance.

WinUI 3 is still in development, but you can use the Preview version that's already available. Visit https://aka.ms/winui3 for more information and the Getting Started tutorial. Once you install the VSIX package with the project templates, you'll see the **Blank App, Packaged (WinUI in Desktop)** option in the Visual Studio New Project dialog (**Figure 11**).

On selecting this template, Visual Studio creates a default WinUI 3 project that's shown in **Figure 12**.

Looking at the default project, you can notice that it has similar structure to WPF project; it includes the App.xaml file that contains the application markup and MainWindow.xaml file with XAML markup of the MainWindow object that derives from the Window object. This way, WPF developers can take advantage of their existing skillset while working with WinUI 3.

Currently WinUI 3 has a temporary limitation: In order to deploy the application like a MISX package, the default solution includes a Windows Application Packaging project. This is a workaround that we'll remove in subsequent previews of WinUI 3.

Get Started

As you've seen there are a lot of improvements to the Windows desktop support in .NET 5. If you're looking for architecture guidance on moving your existing Windows desktop applications forward, see https://docs.microsoft.com/en-us/dotnet/architecture/modernize-desktop/. To get started with .NET 5, install the latest version of Visual Studio 2019 or head to https://dot.net/get-dotnet5.

Olia Gavrysh

Azure Tools for .NET in Visual Studio 2019

With the cloud being such a big part of IT these days, it will come as no surprise to you that Visual Studio includes tools to help you consume Azure services and deploy your app to Azure. What you might not already know, though, is how much Microsoft has invested in these tools in Visual Studio 2019 and the new experiences that are built on top of them.



Angelos Petropoulos
angelos.petropoulos@microsoft.com
devblogs.microsoft.com

Angelos Petropoulos is a Senior Program Manager at Microsoft working on .NET, Azure and Visual Studio. Before joining Microsoft, he spent 10 years designing and implementing enterprise applications as a .NET consultant. He was born in Greece, but he studied in the UK where he got a BSc in Software Engineering and a MSc in Object-Oriented Software Technology. He's one of those guys who still creates new WinForms projects.

Keep reading to learn more about how Visual Studio 2019 automatically discovers your app's dependencies on Azure services, helps you configure your local environment giving you a choice between accessing live Azure services or using local emulators, and helps you not just deploy your application in Azure, but more importantly make sure it **runs correctly.**

Microsoft always tries to light up new tools for as many .NET project types as possible. Even after the initial release of a new tool, we'll continue to add support for more project types in subsequent updates. Right now, the tools covered in this article are available for the following project types: ASP.NET, Azure Functions, WinForms, WPF, and Console.

In Brief: What It Takes to Consume Azure Services

If you already have a lot of experience consuming Azure services, you can safely skip this part. If it's been a while though, this is worth brushing up on. Here are the basics of what's required to consume Azure services from .NET apps:

- **The SDK, which is comprised of binaries required to talk to Azure services and are distributed via NuGet.** Today, each Azure service tends to have its own SDK, although keep in mind that it's not a hard requirement for every single Azure service to require an SDK.
- **A small amount of source code that enables the project to make use of the SDK.** Some use the term "bootstrapping" or "bootstrap code" to describe this. Sometimes this is where the authentication mode for communicating with the Azure service is configured.
- **The configuration artifacts required by the SDK.** The SDK may require certain configuration values to be read when the app runs. These configuration values are read from well-known configuration artifacts that must be added to the project for everything to work.
- **The configuration values for the configuration artifacts.** The value can be different for every environment the app runs in. For example, in a dev environment, you may want to connect to storageAccount1 and in a pre-production environment, you may want to connect to storageAccount2. This is achieved by having a different value in each environment.

Next, I'm going to cover how Visual Studio helps you manage all of the above with ease so you can focus on what matters the most: writing the business logic for your app.

Consume Azure Services Using Connected Services

I talked about what it takes to consume Azure services from a .NET app. You need an SDK in the form of NuGet packages, which you could acquire using Visual Studio's NuGet Package Manager, but what package names and versions do you need?

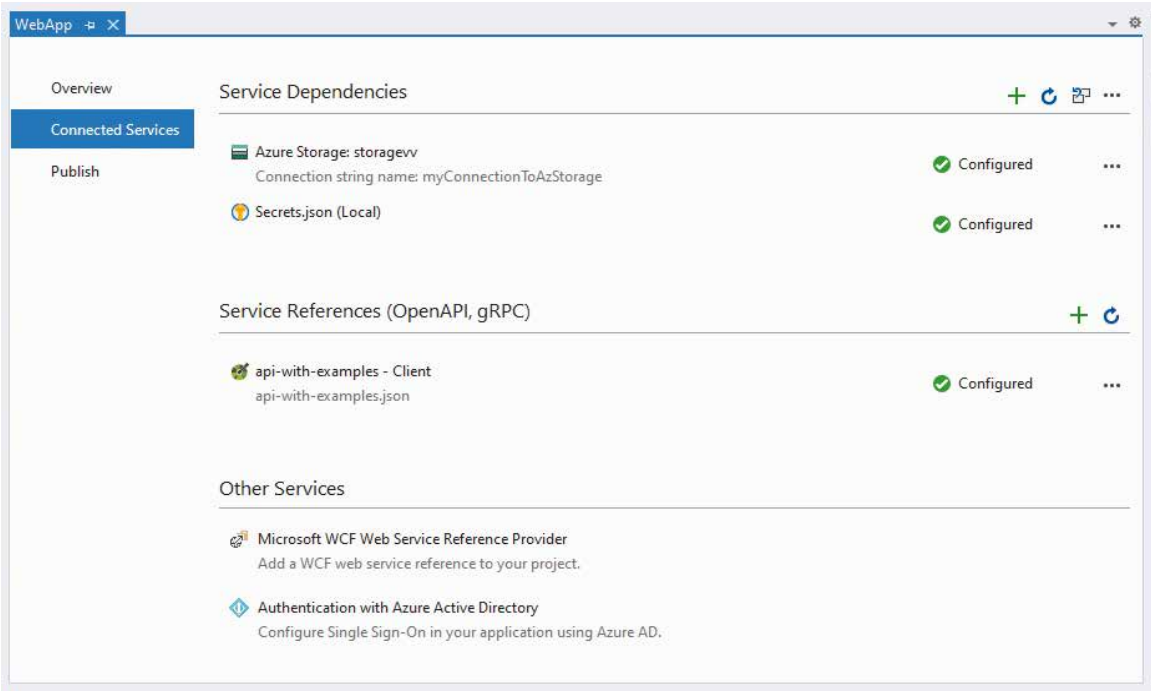


Figure 1: The Connected Services tab

Then you need some configuration artifacts, which you could add using Visual Studio's Add Item dialog, but which ones is this SDK going to look for? Also, when you put the configuration values in the configuration artifacts, what name should you use for each key-value pair?

You can get answers to all of these questions by going through the docs of each Azure service, but that takes time. On top of that, you must be careful to read the right version of the docs for the same framework version and project-type you're working with. Microsoft has observed customers in user studies making simple mistakes along the way, such as missing a step or misreading an instruction, which sometimes leads to frustration and loss of productivity. We realized that we could do more to help customers be successful, so in Visual Studio 2019, we came up with a revamped Connected Services experience (see **Figure 1**).

Let's quickly go over the UI changes to the Connected Services page. As you can see in **Figure 1,** the page contains three sections: *Service Dependencies*, *Service References*, and *Other Services*. Prior to Visual Studio 2019, the Connected Services page gave you a list of items to pick from, but the list wasn't uniform. Some items on the list help you consume Azure services while others help you stand-up and consume APIs by generating code. That made it challenging to add more functionality to the list without also adding confusion. We decided to split the old list up into *Service Dependencies* and *Service References*; leftover items that haven't been migrated but are still available under *Other Services*. Let's focus on the *Service Dependencies* section and talk about how it helps you consume Azure services and configure your local environment all in one go.

Get Started with Connected Services

Start by clicking the **+** icon on the top right corner of the Service Dependencies section (see **Figure 1**). A list of the supported Azure services (and their local emulators/alternatives) shows up, asking for you to pick one, as shown in **Figure 2**.

Table 1 has the complete list of what is currently supported, although keep in mind that we're continuously adding support for new Azure services with each update of Visual Studio. If you're looking for hosting Azure services such as Web Apps and Functions, you won't find them in this list; those are covered later in the section entitled **Deploy Your App to Azure**.

After picking an Azure service, you go through the following steps:

- Either provision a new instance of the service or search for an existing one, without leaving the IDE.
- Visual Studio reaches out to the instance selected and retrieves all information required to establish a successful connection. If any of the information is considered an application secret (e.g., connection strings, usernames, passwords, tokens, etc.) you also get the opportunity to either store it in Azure Key Vault or in a local secrets.json file that lives outside of your repo so that you never accidentally check it in.
- Get a summary of the actions/changes that are about to take place. You can see an example of that in **Figure 3** for adding Azure Storage.

With just a few clicks, you've enabled your application to consume an Azure service, configured how it should work locally, secured any related application secrets, and you're ready to start writing your business logic.

One Click to Restore Dependencies

If your local environment stops connecting to your Azure services successfully, Visual Studio can restore all the de-

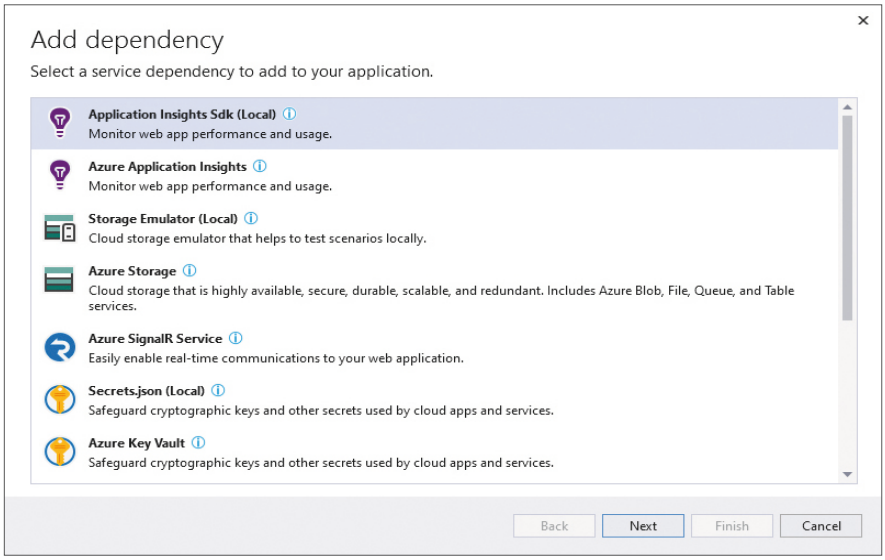


Figure 2: The *Add dependency* dialog in Connected Services

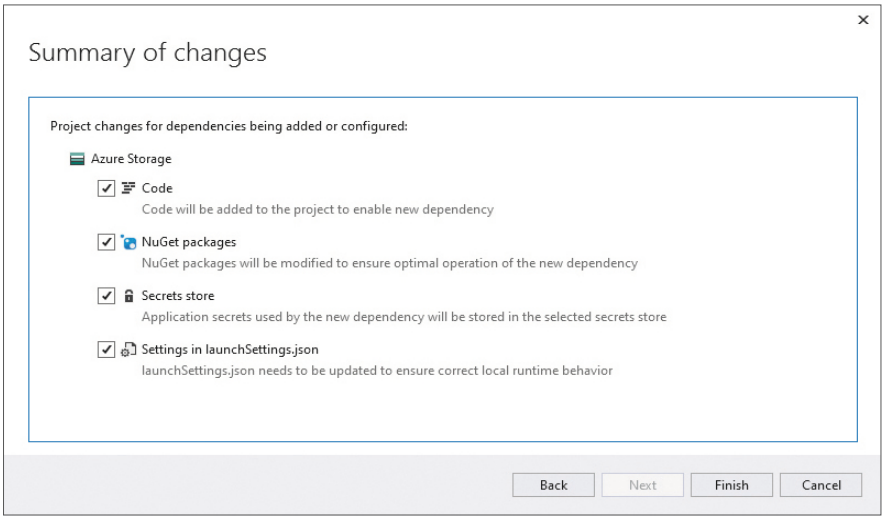


Figure 3: The *Summary of changes* tab in the *Add dependency* dialog

Azure service	Local emulators/alternatives that are also supported
Azure Application Insights	Application Insights SDK
Azure Storage	Azure Storage Emulator
Azure SignalR	
Azure Key Vault	Secrets.json
Azure SQL Server	SQL Server Express LocalDB, SQL Server On-Prem
Azure Cache for Redis	
Azure CosmosDB	

Table 1: List of Azure services you can configure using Service Dependencies

pendencies back to their working state by clicking the *Restore* icon at the top of the Service Dependencies table, two icons to the right of the **+** icon, as seen in **Figure 1**. Visual Studio checks to see if instances are missing and re-provisions them as well as checks whether configuration values are missing or incorrect and updates them with the correct values. If you really want to, you can even customize the restore operation by supplying optional parameters that let you overwrite things such as Azure resource group names and other details.

A Peek Under the Covers

To support all of this, Visual Studio creates two new files visible in Solution Explorer under Properties called *serviceDependencies.json* and *serviceDependencies.local.json*. Both files are safe to check into your repo, as they don't contain any secrets. Here's an example for *serviceDependencies.json*:

```
{
  "dependencies": {
    "secrets1": {
      "type": "secrets"
    },
    "storage1": {
      "type": "storage",
      "connectionId": "myConnectionToAzStorage"
    }
  }
}
```

Here's an example for *serviceDendencies.local.json*:

```
{
  "dependencies": {
    "secrets1": {
      "type": "secrets.user"
    },
    "storage1": {
      "resourceId": "/subscriptions/
        [parameters('subscriptionId')]
        /resourceGroups/
        [parameters('resourceGroupName')]
        /providers/Microsoft.Storage/
        storageAccounts/storagevv",
      "type": "storage.azure",
      "connectionId": "myConnectionToAzStorage",
      "secretStore": "LocalSecretsFile"
    }
  }
}
```

If you look closely at the contents of *serviceDendencies.local.json,* you'll notice that some values are parameterized. Visual Studio also creates a file called *serviceDependencies.local.json.user* that isn't visible in Solution Explorer by default. This file contains the value for all the parameters because some of them could be considered a secret (e.g., Azure subscription ID) and we don't recommend that you check it in unless you have fully understood the information that's being recorded in this file. Here's an example of *serviceDependencies.local.json.user:*

```
{
 "dependencies": {
  "storage1": {
    "restored": true,
    "restoreTime": "2020-08-21T16:48:10.4794127"
  },
  "secrets1": {
    "restored": true,
    "restoreTime": "2020-08-21T16:47:56.9363827"
  }
 },
 "parameters": {
  "storage1.resourceGroupName": {
    "Name": "storage1.resourceGroupName",
    "Type": "resourceGroup",
    "Value": "aapt821group"
   },
  "storage1.subscriptionId": {
    "Name": "storage1.subscriptionId",
    "Type": "subscription",
    "Value": "0000000-0000-0000-000-0000000000"
  }
 }
}
```

It's Never Too Late to Start Using It

If you've already configured your application to consume an Azure service without using Service Dependencies, it isn't too late to benefit from all the additional features that come with it. If you navigate to the Connected Services page, the Service Dependencies section automatically detects and lists Azure dependencies that already exist in your app. If you want to manage these existing dependencies through the Service Dependencies experience, all you have to do is click the "Configure" button next to each one of them. Give it a try; if you change your mind, you can always just delete the dependency afterward.

Deploy Your App to Azure Using Visual Studio Publish

Using Publish in Visual Studio is one of the easiest ways to deploy your application to Azure:

- It helps you navigate the hosting options available to you in Azure.
- It lets you provision new instances or search for existing ones without leaving the IDE.
- It suggests default settings/values based on what it knows about your project.
- It detects missing components that are required to make the deployment work and helps you acquire them.
- It detects your app's dependencies on Azure services and helps you to configure them correctly.
- It recommends supplementary Azure services to ensure that your app is secure and performs optimally in Azure.

Get Started with Visual Studio Publish

To get started, just right-click on your project in Solution Explorer and select *Publish* from the context menu. A dialog pops-up asking you to choose where to deploy to, with Azure listed at the top in alphabetical order. Clicking *Next* gives you a list of Azure hosting services to pick from, as shown in **Figure 4**. If you're interested in the complete listing of all hosting Azure services supported, check out **Table 2**.

After you pick the Azure hosting service you want, you'll get the opportunity to either create a new instance or search for an existing one without leaving the IDE. Completing all

Deep Integration with Azure Key Vault

Azure Key Vault is the recommended way for keeping app secrets secure in Azure and it can also be used for local development. Whether you're configuring dependencies for your local environment or for a Publish Profile, when dealing with secrets, Visual Studio offers to store them in Key Vault.

If the Key Vault instance already exists, all you need to do is change a radio button option. If you need to provision a new instance, you can do so without leaving the IDE.

Azure hosting service	Details
Azure App Service (Web Apps)	Linux and Windows are both supported
Azure Functions	Linux and Windows are both supported
Azure Container Registry	Deployment of the container image to App Service is also supported
Azure VMs	Requires Web Deploy to be already enabled on the VM

Table 2: List of Azure hosting services you can deploy to using Visual Studio Publish

the steps creates a new Publish Profile file (*.pubxml) and displays the Publish Profile summary page, as seen in **Figure 5** At this point, all that's left to do is hit the *Publish* button on the top right corner and the deployment begins. But what about those yellow warning icons? I'll address those in a moment.

Configure Your App's Dependencies

When deploying to Azure, you can think of each Publish Profile as a different environment in Azure. During the creation of the Publish Profile, you picked the Azure hosting service to deploy to, but you didn't configure any of the other dependencies your app may have on Azure services. To do that, you can use what should be a familiar experience by now, the Service Dependencies section (bottom section on **Figure 5**.

Visual Studio will do its best to automatically detect dependencies and ask you to configure them for each Publish Profile. If it has detected unconfigured dependencies, it will warn you to not start the deployment until you've addressed them. That's what those yellow warning icons are in **Figure 5**. The detection logic looks for NuGet packages, source code, and configuration values—whatever's appropriate for each Azure service. If you don't want Visual Studio to manage one of the dependencies and stop warning you about it, you can select the *Delete Dependency* option from the **...** menu.

It's also worth noting that you're not required to use the Service Dependencies section in Connected Services before you can use the Service Dependencies section in a Publish Profile. Naturally, they're designed to integrate with each other to give you the best possible end-to-end experience, but they also work great independently, giving you complete freedom.

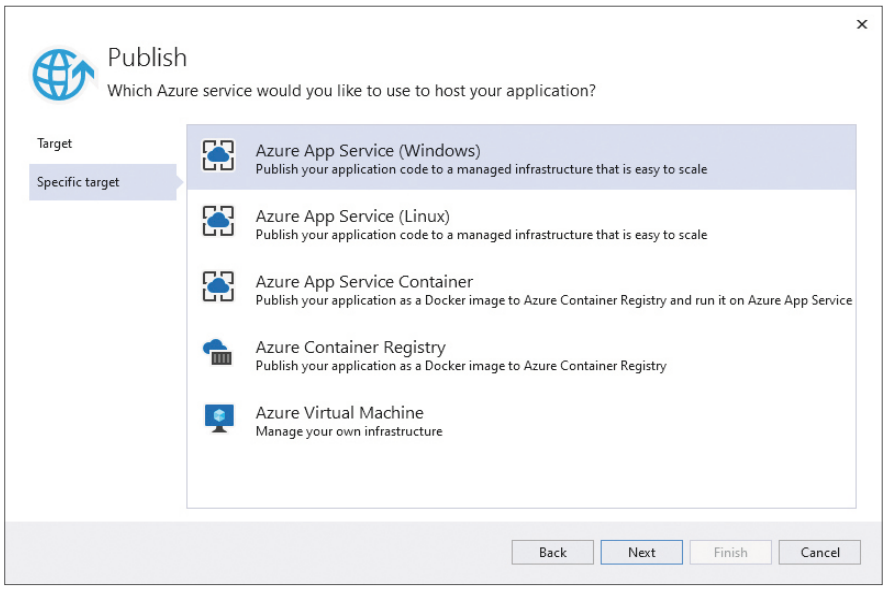


Figure 4: The Publish dialog after selecting Azure as the target

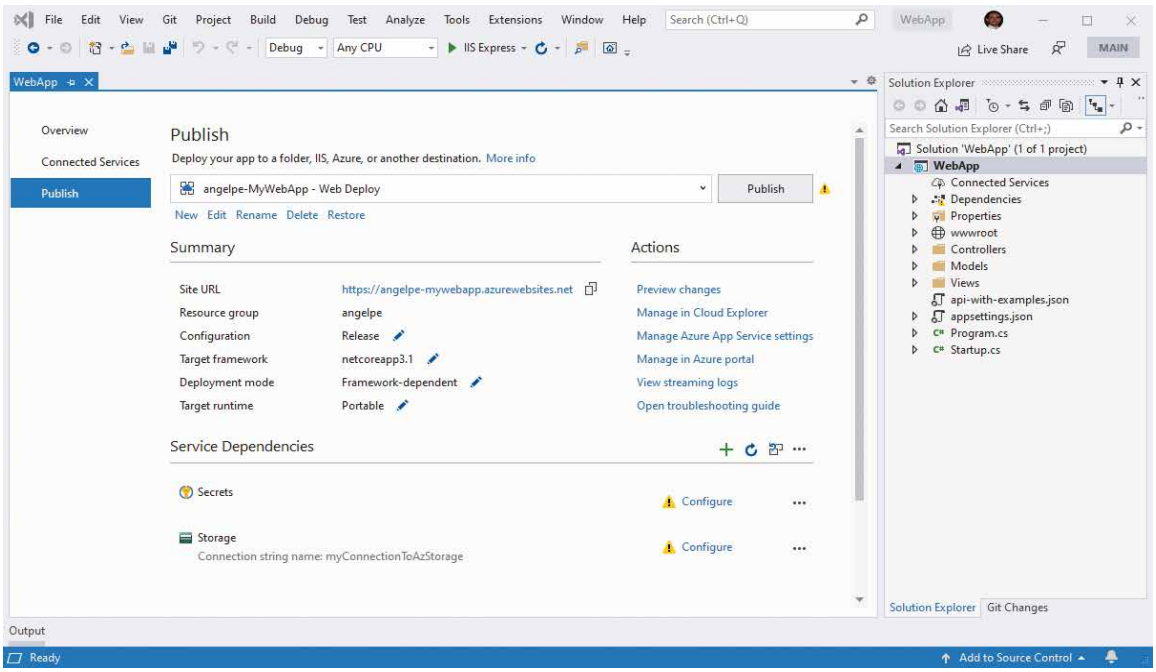


Figure 5: The Publish Profile summary page

Visual Studio's Generated ARM Templates Can Be a Great Starting Point for CICD

You can integrate templates into your continuous integration and continuous deployment (CI/CD) tools, which can automate your release pipelines for fast and reliable updates. Using Azure DevOps and Resource Manager template task, you can use Azure Pipelines to continuously build and deploy ARM templates.

With Visual Studio's generated ARM templates, you don't have to start from scratch!

Lsiting 1: Implementing infrastructure in the repo

```
{
 "$schema": "https://schema.management.azure.com/
    schemas/2018-05-01/
     subscriptionDeploymentTemplate.json#",
 "contentVersion": "1.0.0.0",
 "parameters": {
   "resourceGroupName": {
     "type": "string",
     "defaultValue": "aapt821group",
     "metadata": {
       "_parameterType": "resourceGroup",
       "description": "Name of the resource group
    for the resource. It is recommended to put
    resources under same resource group for better
    tracking."
     }
   },
   "resourceGroupLocation": {
     "type": "string",
     "defaultValue": "centralus",
     "metadata": {
       "_parameterType": "location",
       "description": "Location of the resource
     group. Resource groups could have different
     location than resources."
     }
   },
```

SPONSORED SIDEBAR:

Interested in Moving to the Azure Cloud? CODE Can Help!

Take advantage of a FREE hour-long CODE Consulting session (Yes, FREE!) to jumpstart your organization's plans to develop solutions on the Microsoft Azure cloud platform. No strings. No commitment. Nothing to buy. For more information, visit www.codemag.com/consulting or email us at info@codemag.com.

Get Specific Recommendations

The Service Dependencies section recommends additional Azure services based on the security and performance needs of your app. Visual Studio has a deep understanding of your app and uses this information to give you specific recommendations. A good example is the recommendation to use the Azure SignalR Service with specific ASP.NET apps when performance is important.

Check Out All of These ARM Templates

Azure Resource Manager (ARM) is the deployment and management service for Azure. It provides a management layer that enables you to create, update, and delete resources in your Azure account. ARM templates are basically JSON files with a declarative syntax that let you deploy entire environments. They're the way Azure recommends that you implement infrastructure as code for your Azure apps, as you can simply include them in your repo and version them. **Listing 1** includes the first 20 lines of such a file to give you an idea:

Visual Studio uses ARM to manage your app's dependencies and you can find the ARM templates that it uses in Solution Explorer under Properties > Service Dependencies followed by either *local* for your local environment or *the name of the Publish Profile*.

Final Thoughts

Give these Azure tools a try and let us know what you think. If you wish that we supported a feature or Azure service that we don't already, please let us know! You can submit your suggestions at https://developercommunity.visualstudio.com/.

Angelos Petropoulos

CODE Focus Nov 2020
Volume 21 Issue 6

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
Bri Achtman
Phillip Carter
Richard Lander
Julie Lerman
Beth Massi
Angelos Petropoulos
Bill Wagner
Shayne Boyer
Olia Gavrysh
Immo Landwerth
Jeremy Likness
David Ortinau
Daniel Roth

Technical Reviewers
Markus Egger
Rod Paddock

Production
Franz Wimmer
King Laurin GmbH
39057 St. Michael/Eppan, Italy

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

Subscriptions
Subscription Manager
Colleen Cade
ccade@codemag.com

US subscriptions are US $29.99 for one year. Subscriptions outside the US are US $49.99. 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

.NET Stream Team

Connect with experts

live.dot.net















Learn along with us

On .NET
.NET Community Standup
The Xamarin Show
Visual Studio Toolbox
Azure Friday

.NET

Microsoft | Visual Studio

Visual Studio, Visual Studio for Mac, Visual Studio Code support .NET 5

Download at:

visualstudio.com/download