

# .NET Core: Developing Cross-Platform Web Apps with ASP.NET Core – Workshop*PLUS*

Wael Kdounh - @waelkdounh

Senior Customer Engineer

v1.0

## Conditions and Terms of Use

### Microsoft Confidential

This training package is proprietary and confidential, and is intended only for uses described in the training materials. Content and software is provided to you under a Non-Disclosure Agreement and cannot be distributed. Copying or disclosing all or any portion of the content and/or software included in such packages is strictly prohibited.

The contents of this package are for informational and training purposes only and are provided "as is" without warranty of any kind, whether express or implied, including but not limited to the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Training package content, including URLs and other Internet Web site references, is subject to change without notice. Because Microsoft must respond to changing market conditions, the content should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

## Copyright and Trademarks

© 2013 Microsoft Corporation. All rights reserved.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

For more information, see Use of Microsoft Copyrighted Content at

<http://www.microsoft.com/about/legal/permissions/>

Active Directory, Azure, IntelliSense, Internet Explorer, Microsoft, Microsoft Corporate Logo, Silverlight, SharePoint, SQL Server, Visual Basic, Visual Studio, Windows, Windows Server, and Windows Vista are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other Microsoft products mentioned herein may be either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. All other trademarks are property of their respective owners.

# How to View This Presentation

- To switch to **Notes Page** view:
  - On the ribbon, click the **View** tab, and then click **Notes Page**
- To navigate through notes, use the Page Up and Page Down keys
  - Zoom in or zoom out, if required
- In the **Notes Page** view, you can:
  - Read any supporting text—now or after the delivery
  - Add notes to your copy of the presentation, if required
- Take the presentation files home with you

# Module 1: ASP.Net Core

## Module Overview

Module 1: ASP.Net Core

Section 1: Modern Web

Lesson: ASP.NET and Modern  
Web

What Is Modern Web?

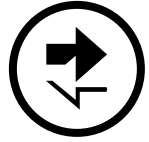
# Modern Web

- Web Frameworks
  - Mobile / Tablet First
  - Responsive Design
  - Client Frameworks
  - Cloud Ready
- Web Tooling
  - Standards Based
  - Tooling in Browser
  - Adopting Popular third-party Tools

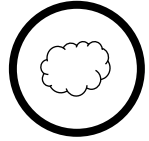
# ASP.NET Core for the Modern Web



Totally Modular



Faster Development Cycle



Seamless Transition From On-premises To Cloud



Fast



Choose Your Editors And Tools



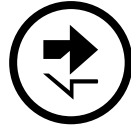

[Open Source with Contributions](#)



Cross-platform



# ASP.NET Core - Agility

-  **Faster Development Cycle**
  - Features are shipped as packages
  - Framework ships as part of the application
-  **More Control**
  - Zero day security bugs patched by Microsoft
  - Same code runs in development and production
  - Developer opts to new versions, allowing breaking changes

# ASP.NET Core - Fast



## Development Productivity And Low Friction

- Edit code and refresh browser
- Flexibility of dynamic environment with the power of .NET Framework
- Develop with Visual Studio, third-party and cloud editors



## Runtime Performance

- Faster startup times
- Lower memory / higher density (more than 90% reduction)
- Modular, opt into just features needed
- Use a raw socket, framework or both

# ASP.NET Core - Cloud

 Seamless transition from on-premises to Cloud and Cloud Ready


 Cloud Ready

- Configuration, Session and Cache

 Diagnostics

- Run/Debug in Cloud
- Tracing/Logging without re-deploying

# ASP.NET Core – Cross Platform

 Open Source with Contributions

 Runtime

- Windows, Mac, Linux (Debian, Ubuntu, CentOS, Fedora, and derivatives)

 Editors

- Visual Studio, Text, and Cloud editors
- No editors (command-line)

# Demo: ASP.NET Core and Visual Studio 2019

Module 1: ASP.Net Core

Section 1: Modern Web

Lesson: One ASP.NET

# ASP.NET Core

Sites

Services

Blazor

Razor  
Pages

MVC

gRPC  
Service

Web  
API

SignalR  
Core

ASP.NET Core

# Commonalities

- All programming models have the same Microsoft ASP.NET
  - Authentication/Authorization/Membership
  - Output Caching, Session State, and Configuration
  - AJAX, Deployment, etc.
- All programming models are fully supported and will continue to be supported
- All programming models solve real problems




# ASP.NET Core Project System


## Create a new ASP.NET Core web application

.NET Core


ASP.NET Core 3.1

 Empty


An empty project template for creating an ASP.NET Core application. This template does not have any content in it.

 API

A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.

 Web Application

A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.

 Web Application (Model-View-Controller)

A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.

Authentication

No Authentication

Change

Advanced


☒ Configure for HTTPS

☐ Enable Docker Support  
(Requires [Docker Desktop](#))


Linux

## Create a new Blazor app

ASP.NET Core 3.1

 Blazor Server App

A project template for creating a Blazor server app that runs server-side inside an ASP.NET Core app and handles user interactions over a SignalR connection. This template can be used for web apps with rich dynamic user interfaces (UIs).

 Blazor WebAssembly App

A project template for creating a Blazor app that runs on WebAssembly. This template can be used for web apps with rich dynamic user interfaces (UIs).

Authentication

No Authentication

Change

Advanced


☐ Enable Docker Support  
(Requires [Docker Desktop](#))

Linux

☐ ASP.NET Core hosted

## Create a new gRPC service

ASP.NET Core 3.1

 gRPC Service

A project template for creating a gRPC ASP.NET Core service.

Authentication

No Authentication

Change

Advanced

☐ Enable Docker Support  
(Requires [Docker Desktop](#))

Linux

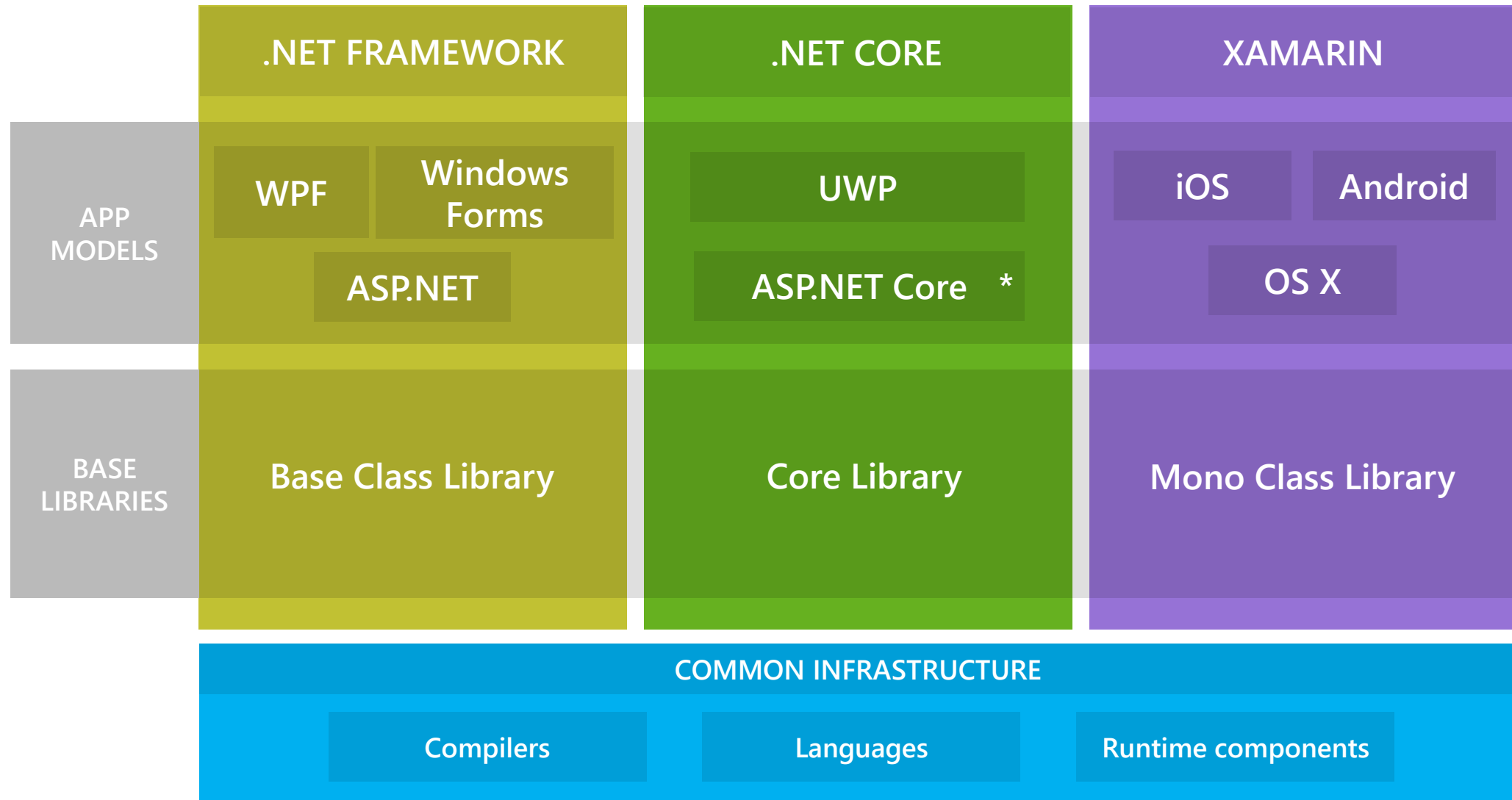
# Demo: One ASP.NET

Module 1: ASP.Net Core

Section 2: .NET Platform

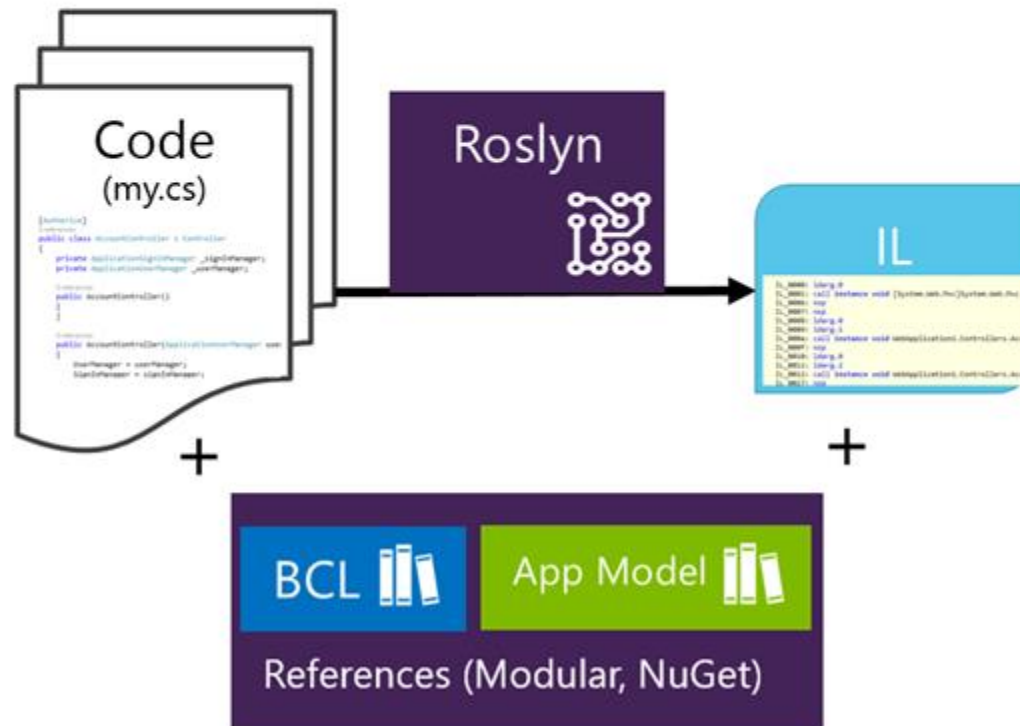
Lesson: Overview

# The Open .NET Ecosystem



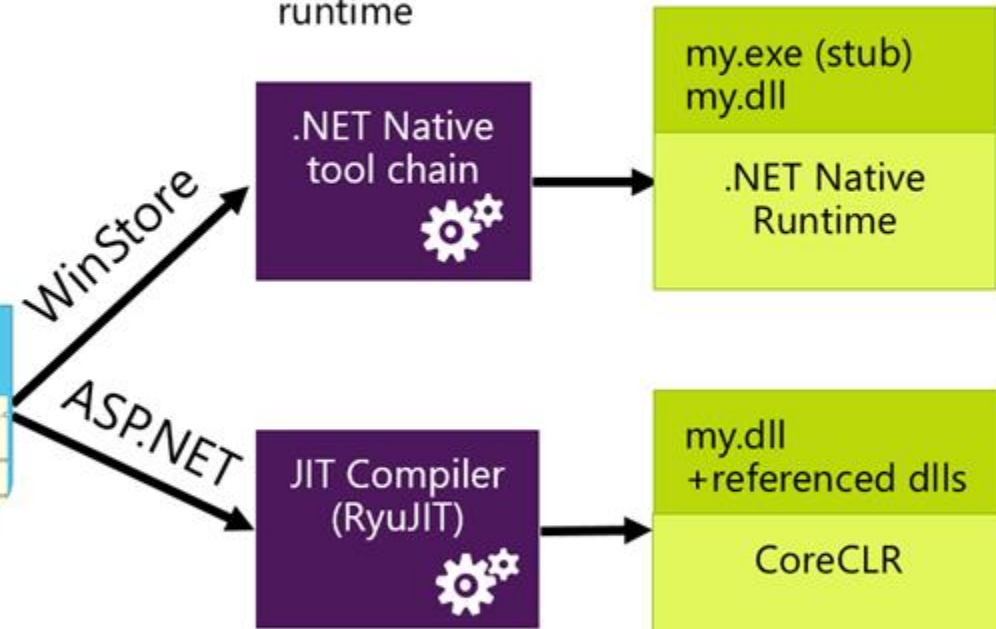
## Code / Build / Debug

Roslyn takes your code and compiles it to Intermediate language (IL). You have module references to the BCL and App Model that you are targeting.



## Deploy and Run

References are built with your app into one native dll deployed locally with runtime



References and CoreCLR are deployed with app locally, just-in-time (JIT) compilation on start up.

# ASP.NET vs. ASP.NET Core

MSBuild/CodeDOM > csc.exe

Loose, GAC, NuGet

FCL, GAC, NuGet

IIS

.NET BCL and FCL

.NET CLR

IIS: WebEngine4.dll; EXE: OS

Windows

Compilation

Libraries

Application Frameworks

Web Server

Platform Libraries

Runtime

Runtime Loader

Operating System

.Net CLI (Roslyn)

NuGet, npm, Bower

NuGet

IIS, HTTP.SYS, Kestrel

.NET BCL and FCL; .NET on NuGet

.NET CLR; .NET Core CLR

.Net CLI

Windows, OSX, Linux

# Which One is Right for Me?

## ASP.NET Core

Build for Windows, Mac, or Linux

Use [MVC](#), or [Web API](#)

Multiple versions per machine

Develop with Visual Studio or Visual Studio Code using C#

New platform

Ultra performance

[Choose .NET Framework or .NET Core runtime](#)

## ASP.NET

Build for Windows

Use [Web Forms](#), [SignalR](#), [MVC](#), [Web API](#), or [Web Pages](#)

One version per machine

Develop with Visual Studio using C#, VB or F#

Mature platform

High performance

Use .NET Framework runtime

# .NET Framework vs. .NET Core (Server Apps)

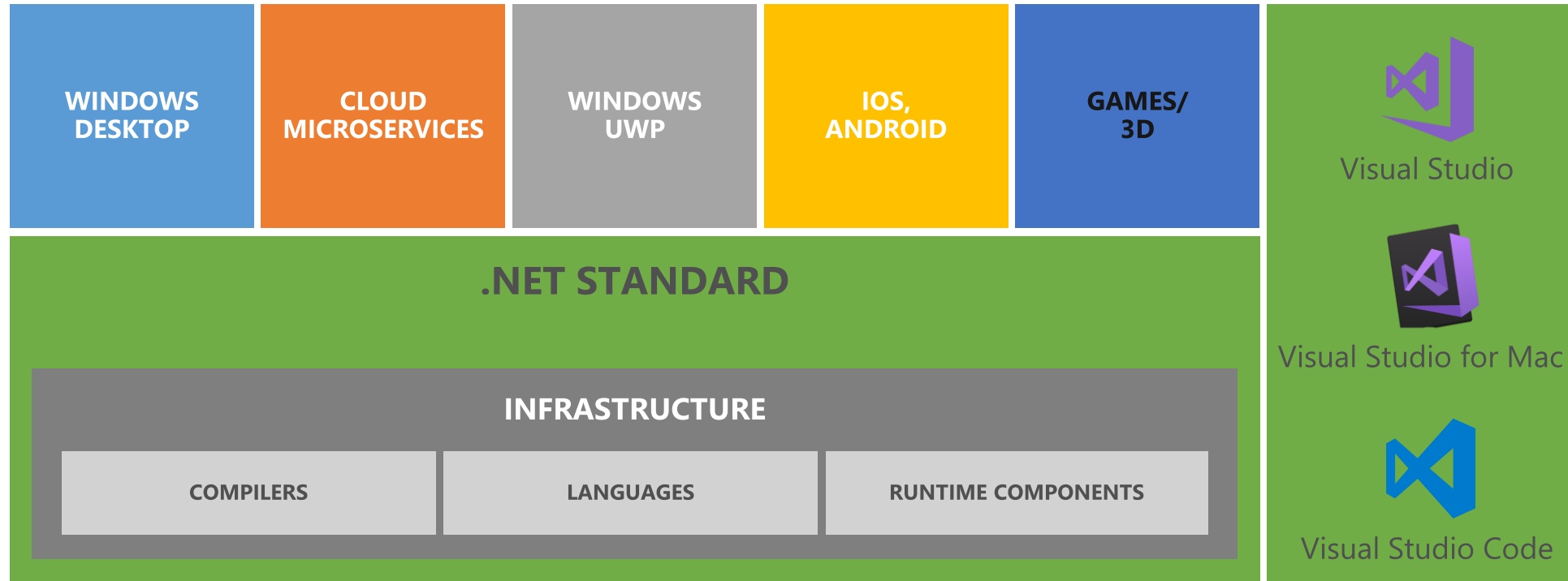
.NET Framework	.NET Core
Current application runs on .NET framework. Recommended to extend it instead of migrating	Cross-platform needs
Need 3 <sup>rd</sup> party libraries not available on .NET Core	Targeting microservices
Need .NET technologies not available on .NET Core	Using Docker containers
Need a platform not supported by .NET Core	Need high performance & scalable systems
	Side-by-side .NET versions by application
	Fully open-source



# .NET Standard Library

- Goal: Establish greater uniformity in the .NET ecosystem
- A set of APIs that all .NET platforms have to implement
- Unifies the .NET platform and prevents future fragmentation
- .NET Standard will replace Portable Class Libraries (PCLs) as the tooling story for building multi-platform .NET libraries.
- Addresses three main scenarios:
  - Defines uniform set of BCL APIs for all .NET implementations to implement, independent of workload.
  - Enables developers to produce portable libraries that are usable across .NET implementations, using this same set of APIs.
  - Reduces or even eliminates conditional compilation of shared source due to .NET APIs, only for OS APIs.

# .NET Standard



.NET Standard allows sharing code, binaries, and skills between .NET client, server, and all flavors

.NET Standard provides a specification for any platform to implement

All .NET runtimes provided by Microsoft implement the standard

# .NET Standard Library

<b>.NET Standard</b>	<b>1.0</b>	<b>1.1</b>	<b>1.2</b>	<b>1.3</b>	<b>1.4</b>	<b>1.5</b>	<b>1.6</b>	<b>2.0</b>	<b>2.1</b>
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	3.0
.NET Framework <sup>1</sup>	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1 <sup>2</sup>	4.6.1 <sup>2</sup>	4.6.1 <sup>2</sup>	N/A <sup>3</sup>
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4	6.4
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.14	12.16
Xamarin.Mac	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.8	5.16
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0	10.0
Universal Windows Platform	10.0	10.0	10.0	10.0	10.0	10.0.16299	10.0.16299	10.0.16299	TBD
Unity	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	TBD

<sup>3</sup> .NET Framework won't support .NET Standard 2.1 or later versions. For more details, see the [announcement of .NET Standard 2.1](#).

# APIs in .NET Standard 2.0

## **XML**

XLinq • XML Document • XPath • Schema • XSL

## **SERIALIZATION**

BinaryFormatter • Data Contract • XML

## **NETWORKING**

Sockets • HTTP • Mail • WebSockets

## **IO**

Files • Compression • MMF

## **THREADING**

Threads • Thread Pool • Tasks

## **CORE**

Primitives • Collections • Reflection • Interop • Linq

# .NET Standard 2.0 coverage and support

## **Much bigger API Surface**

We have more than doubled the set of available APIs from **13k** in [.NET Standard 1.6](#) to **32k** in [.NET Standard 2.0](#). Most of them are existing .NET Framework APIs.

## **.NET Framework compatibility mode**

The vast majority of NuGet packages are currently still targeting .NET Framework. Many projects are currently blocked from moving to .NET Standard because not all their dependencies are targeting .NET Standard yet. We added a compatibility mode that allows .NET Standard projects to reference .NET Framework libraries. Found that [70% of all NuGet packages on nuget.org are API compatible](#) with .NET Standard 2.0. So in practice it unblocks many projects.

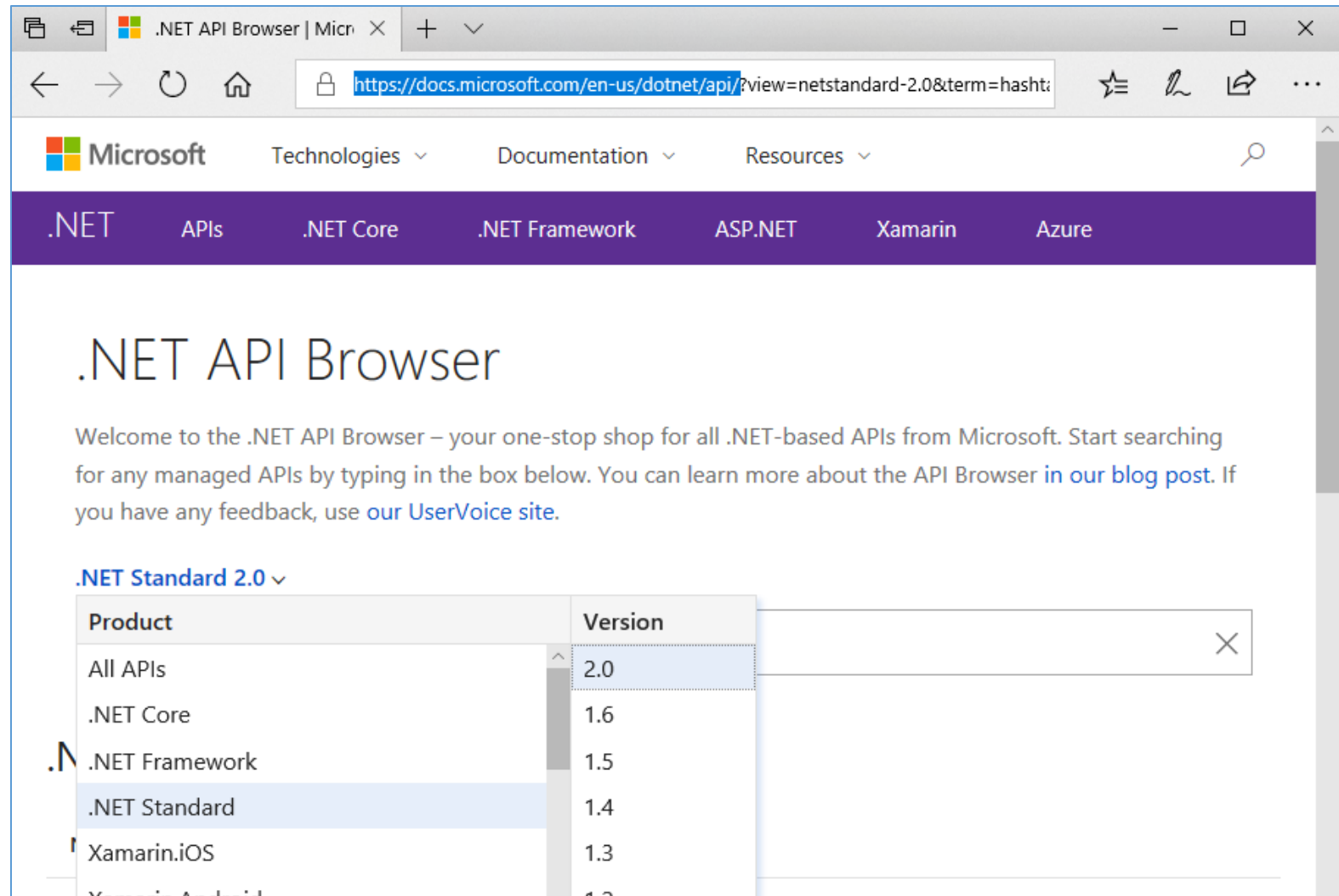
# Which Version Of .NET Standard Should I Target?

- When choosing a .NET Standard version you should consider this trade-off:
  - The higher the version, the more APIs are available to you.
  - The lower the version, the more platforms you can run on.
- So generally speaking, you should target the lowest version you get away with.

# .NET API Browser

Is one-stop shop for all .NET-based APIs from Microsoft. You can search for any managed APIs in it.

<https://docs.microsoft.com/en-us/dotnet/api/>



Demo:

.NET Standard



Module 1: ASP.Net Core

Section 3: ASP.NET Core

Lesson: ASP.NET Core Projects

# ASP.NET Core Project File

- **\*.csproj**
  - Simplified project file
  - Automatically includes all source files in/under the folder containing project.json
- All project folder files shown in Solution Explorer
  - Visual Studio automatically monitors the ASP.NET Core project directory files
- project.json no longer supported
  - Migrated to \*.csproj through Visual Studio Migration or through `dotnet migrate` on CLI

# ASP.NET Core Project File Contents

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

  <PropertyGroup>
    <TargetFramework>netcoreapp2.2</TargetFramework>
    <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
    <RootNamespace>netcore2._2</RootNamespace>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
    <PackageReference Include="Microsoft.AspNetCore.Razor.Design" Version="2.2.0" PrivateAssets="All" />
  </ItemGroup>

</Project>
```

Not required starting  
with netcoreapp3.0

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

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
  </ItemGroup>

</Project>
```

# ASP.NET Core Project File Contents

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

  <PropertyGroup>
    <TargetFramework>netcoreapp2.2</TargetFramework>
    <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
    <RootNamespace>netcore2._2</RootNamespace>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App" />
    <PackageReference Include="Microsoft.AspNetCore.Razor.Design" Version="2.2.0" PrivateAssets="All" />
  </ItemGroup>

</Project>
```

ASP.NET Core shared  
framework

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

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
  </ItemGroup>

</Project>
```

# What is Microsoft.AspNetCore.App Metapackage?

- **Microsoft.AspNetCore.App** is installed when the .NET Core 3.0 or later SDK is installed. The shared framework is the set of assemblies (.dll files) that are installed on the machine and includes a runtime component and a targeting pack
- **Projects that target the Microsoft.NET.Sdk.Web SDK implicitly reference the Microsoft.AspNetCore.App framework**  
<Project Sdk="Microsoft.NET.Sdk.Web">
- **ASP.NET Core 3.0 removes some assemblies that were previously part of the Microsoft.AspNetCore.App package reference. Most notable sub-components**
  - Json.NET (Newtonsoft.Json)
  - Entity Framework Core (Microsoft.EntityFrameworkCore.\*)
  - Microsoft.CodeAnalysis (Roslyn)

# Shared Framework – Deep Dive

- To put it simply, a .NET Core shared framework is a folder of assemblies (\*.dll files) that are not in the application folder
- These assemblies version and release together. This folder is one part of the “shared system-wide version of .NET Core”, and is usually found in **C:/Program Files/dotnet/shared**

# .NET Runtimes

- .NET Core apps run in one of two modes: **framework-dependent** or **self-contained**
- **Framework-dependent** deployment relies on the presence of a shared system-wide version of .NET Core
- **Self-contained** deployment doesn't rely on the presence of shared components on the target system. All components are included with the application
- You can produce both kinds of apps with these command line instructions:
  - **dotnet publish --configuration Release --runtime win10-x64 --output bin/self\_contained\_app/**
  - **dotnet publish --configuration Release --output bin/framework\_dependent\_app/**

# .NET Runtimes





- Build could be done for “common” platform or for specific platform

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

  <PropertyGroup>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <RuntimeIdentifiers>win10-x64</RuntimeIdentifiers>
  </PropertyGroup>

  <ItemGroup>
  </ItemGroup>

</Project>
```

<< BuildForSpecificRuntime > bin > Debug > netcoreapp1.1 >	
<input type="checkbox"/> Name	Date modified
 ubuntu.16.10-x64	11-12-16 18:50
 win10-x64	11-12-16 18:49
 BuildForSpecificRuntime.dll	11-12-16 18:50
 BuildForSpecificRuntime.pdb	11-12-16 18:50

```
C:\t\ASP.NETCore\Demos\Module 01 - Overview\BuildForSpecificRuntime>dotnet publish --runtime ubuntu.16.10-x64
Publishing BuildForSpecificRuntime for .NETCoreApp,Version=v1.1/ubuntu.16.10-x64
Project BuildForSpecificRuntime (.NETCoreApp,Version=v1.1) was previously compiled. Skipping compilation.
publish: Published to C:\t\ASP.NETCore\Demos\Module 01 - Overview\BuildForSpecificRuntime\bin\Debug\netcoreapp1.1\ubuntu.16.10-x64\publish
Published 1/1 projects successfully

C:\t\ASP.NETCore\Demos\Module 01 - Overview\BuildForSpecificRuntime>dotnet publish --runtime win10-x64
Publishing BuildForSpecificRuntime for .NETCoreApp,Version=v1.1/win10-x64
Project BuildForSpecificRuntime (.NETCoreApp,Version=v1.1) was previously compiled. Skipping compilation.
publish: Published to C:\t\ASP.NETCore\Demos\Module 01 - Overview\BuildForSpecificRuntime\bin\Debug\netcoreapp1.1\win10-x64\publish
Published 1/1 projects successfully
```



# Demo: Build On Windows For Different Runtimes

Module 1: ASP.Net Core

Section 3: ASP.NET Core

Lesson: Command Line Interface  
(CLI)

# .NET Core Command Line Interface (CLI)

- Cross-platform toolchain for developing .NET Core applications
- Primary layer built upon by Visual Studio, editors, build orchestrators, etc.
- Cross-platform with same surface area for supported platforms
- Language agnostic
- Target agnostic

```
dotnet new  
dotnet restore  
dotnet build --output /stuff  
dotnet run /stuff/new.dll
```

# CLI Command Examples

<code>dotnet restore</code>	Uses NuGet to restore dependencies as well as project-specific tools that are specified in the project file in parallel.
<code>dotnet build</code>	Restores any dependencies then builds the project and its dependencies into a set of binaries. The binaries include the project's code in Intermediate Language (IL) files with a .dll extension and symbol files used for debugging with a .pdb extension.
<code>dotnet run</code>	It allows you to run your application from the source code with one command. It's useful for fast iterative development from the command line. The command depends on the dotnet build command to build the code. Any requirements for the build, such as that the project must be restored first.
<code>dotnet clean</code>	Cleans the output of the previous build.
<code>dotnet new web</code>	Create a new Empty web application then restores the dependencies/packages for it.

## .NET Core Tooling

Visual Studio

VS Code

.NET Core  
Command Line  
tools

Shared SDK component

# CLI dotnet new templates

.NET Core 3 introduced many new templates from the CLI

Example:

```
dotnet new blazorwasm
```

This will create an ASP.NET Core Web Application which uses blazor WebAssembly

Template description	Template name	Languages
Console application	console	[C#], F#, VB
Class library	classlib	[C#], F#, VB
ASP.NET Core empty	web	[C#], F#
ASP.NET Core Web App (Model-View-Controller)	mvc	[C#], F#
ASP.NET Core Web App	razor	[C#]
ASP.NET Core with Angular	angular	[C#]
ASP.NET Core with React.js	react	[C#]
<b>Blazor Server App</b>	<b>blazorserver</b>	[C#]
<b>Blazor WebAssembly App</b>	<b>blazorwasm</b>	[C#]
...	...	...

# .NET Core CLI Extensibility

- .NET Core is built for extensibility, you extend the CLI with your own custom commands and tooling
- The CLI tools can be extended in three main ways:
  1. Via NuGet packages on a per-project basis  
Per-project tools are contained within the project's context, but they allow easy installation through restoration.
  2. Via NuGet packages with custom targets  
Custom targets allow you to easily extend the build process with custom tasks.
  3. Via the system's PATH  
PATH-based tools are good for general, cross-project tools that are usable on a single machine.

Example of extensibility is the EF Core commands

# Demo: .NET Core CLI & Visual Studio Code



Module 1: ASP.Net Core

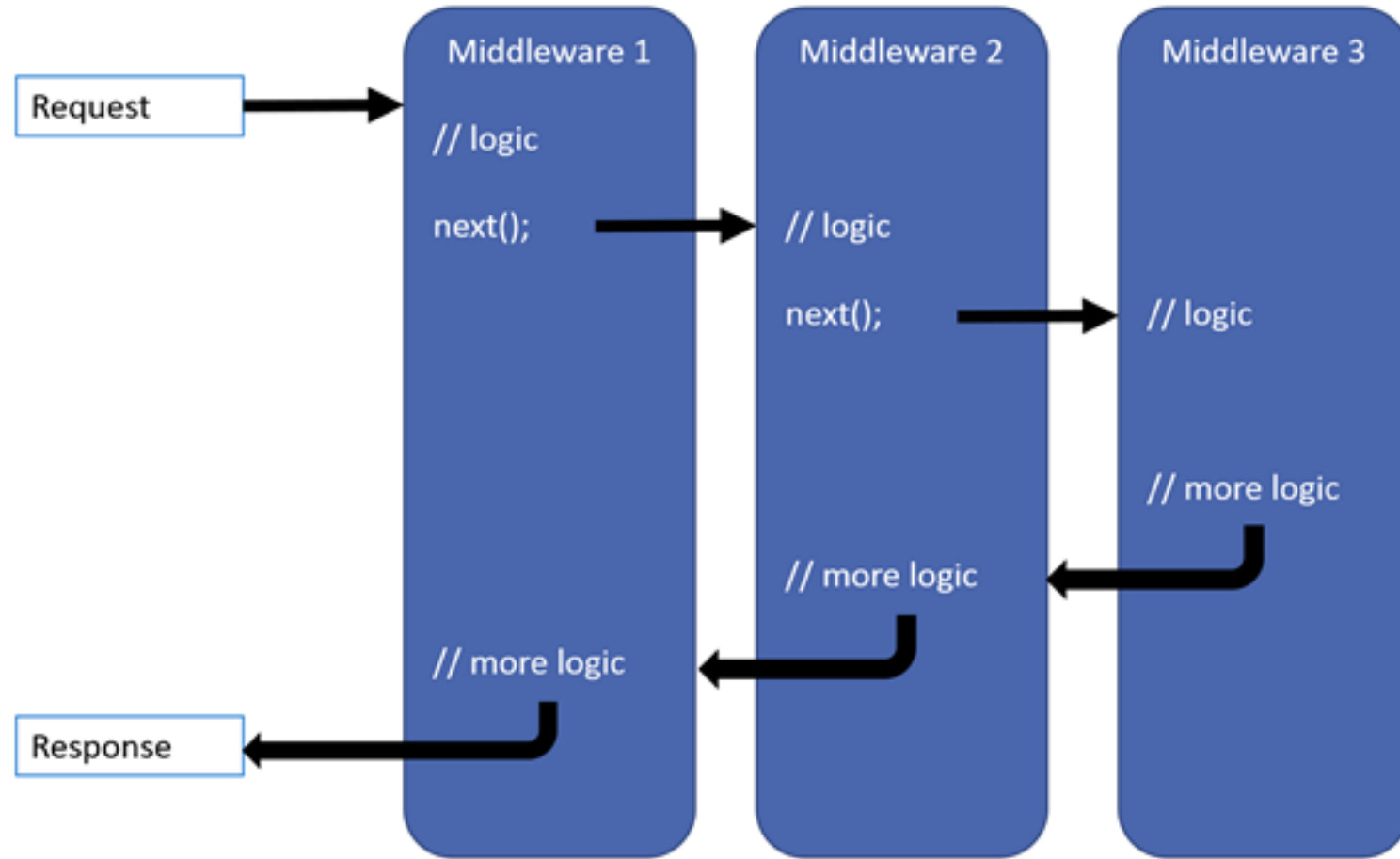
Section 3: ASP.NET Core

Lesson: Middleware

# Middleware

- Small application components assembled into an application pipeline to handle requests and responses
- Integrated support by ASP.NET Core
- Wired up in **Configure** method of **Startup** class
- Either invokes the next component in the chain or short-circuits it
- **Run**, **Map**, and **Use** extension methods
- Implemented in-line as anonymous method, or through a reusable class
- Order of **Use[Middleware]** statements in application's Configure method is very important

# Middleware Pipeline



# Simple ASP.NET Core Middleware

```
public class Startup {  
    public void Configure(IApplicationBuilder app) {  
        app.Run(async context =>  
        {  
            await context.Response.WriteAsync("Hello, World!");  
        });  
    }  
}
```

Run is a terminal  
middleware

The first Run delegate  
terminates the pipeline

# Chain Multiple Request Delegates Together With *Use*

```
public class Startup
```

```
{
```

```
    public void
```

```
    {
```

```
        app.Use
```

```
    {
```

```
        //
```

```
        await
```

```
        //
```

```
    });
```

```
    app.Run(async context =>
```

```
    {
```

```
        await context.Response.WriteAsync("Hello from 2nd delegate.");
```

```
    });
```

```
    }
```

```
}
```

next parameter

the next  
pipeline

## ⚠ Warning

Don't call `next.Invoke` after the response has been sent to the client. Changes to `HttpResponse` after the response has started throw an exception. For example, changes such as setting headers and a status code throw an exception. Writing to the response body after calling `next`:

- May cause a protocol violation. For example, writing more than the stated `Content-Length`.
- May corrupt the body format. For example, writing an HTML footer to a CSS file.

`HasStarted` is a useful hint to indicate if headers have been sent or the body has been written to.

circuit the  
t calling  
meter

# Built-in Middleware

Middleware	Description
Authentication	Provides authentication support
CORS	Configures Cross-Origin Resource Sharing
Diagnostics	Includes support for error pages and runtime information
Routing	Define and constrain request routes
Session	Provides support for managing user sessions
Static Files	Provides support for serving static files, and directory browsing

Full list can be found [here](#)

Module 1: ASP.Net Core

Section 3: ASP.NET Core

Lesson: Hosting

# Hosting in ASP.NET Core

- Host is responsible for app startup and lifetime management. At a minimum, the host configures a server and a request processing pipeline
- Many defaults encapsulated in new API: **WebHost.CreateDefaultBuilder**

```
public class Program
{
    0 references | 0 exceptions
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    1 reference | 0 exceptions
    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```



# Default Configurations for `WebHost.CreateDefaultBuilder`

- Configures Kestrel as the web server
- Sets the content root to **`Directory.GetCurrentDirectory`**
- Loads optional configuration from:
  - `appsettings.json`.
  - `appsettings.{Environment}.json`.
  - User secrets when the app runs in the **`Development`** environment
  - Environment variables
  - Command-line arguments
- Configures logging for console and debug output with log filtering rules specified in a Logging configuration section of an `appsettings.json` or `appsettings.{Environment}.json` file
- Enables IIS integration by configuring the base path and port the server should listen on when using the ASP.NET Core Module if you're running under IIS

# Host Configuration Values

- **Server URLs:** Indicates the IP addresses or host addresses with ports and protocols that the server should listen on for requests.  
`.UseUrls("http://*:5000;http://localhost:5001;https://hostname:5002")`
- **Startup Assembly:** Determines the assembly to search for the Startup class.  
`.UseStartup("StartupAssemblyName")`
- **Environment:** Sets the app's environment  
`.UseEnvironment("Development")`
- **Contents Root:** This setting determines where ASP.NET Core begins searching for content files, such as MVC views.  
`.UseContentRoot("c:\\mywebsite")`

# Host Configuration Values

- **Detailed Errors:** Determines if detailed errors should be captured.  
`.UseSetting(WebHostDefaults.DetailedErrorsKey, "true")`
- **Capture Startup Errors:** This setting controls the capture of startup errors.  
`.CaptureStartupErrors(true)`
- **Web Root:** Sets the relative path to the app's static assets.  
`.UseWebRoot("(Content Root)/wwwroot")`

Module 1: ASP.Net Core

Section 3: ASP.NET Core

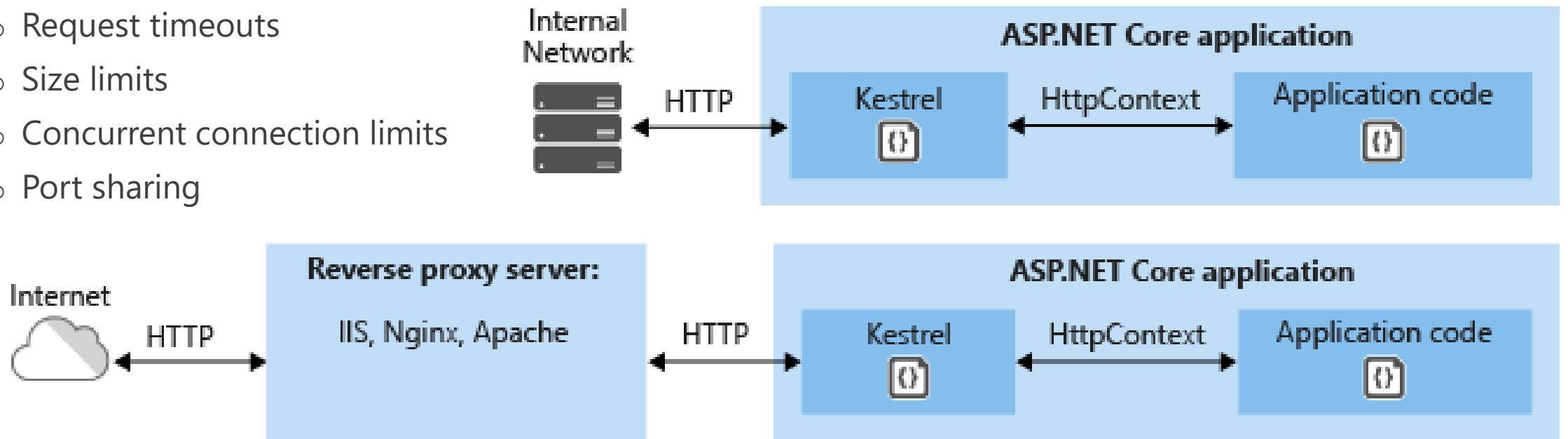
Lesson: Hosting Servers

# ASP.NET Core Hosting

- ASP.NET Core is completely decoupled from the web server environment that hosts the application
- ASP.NET Core ships with:
  - **Kestrel**: Cross-platform HTTP server based on libuv, a cross-platform asynchronous I/O library
  - **WebListener**: Windows-only HTTP server based on the Http.Sys kernel driver
- ASP.NET Core defines a number of HTTP Feature Interfaces
  - Used by web servers and middleware to identify supported features

# Kestrel

- Supported Features
  - HTTPS
  - WebSockets
  - Unix sockets for high performance behind Nginx
- Kestrel does not yet support:
  - Request timeouts
  - Size limits
  - Concurrent connection limits
  - Port sharing



# ASP.NET Core Module

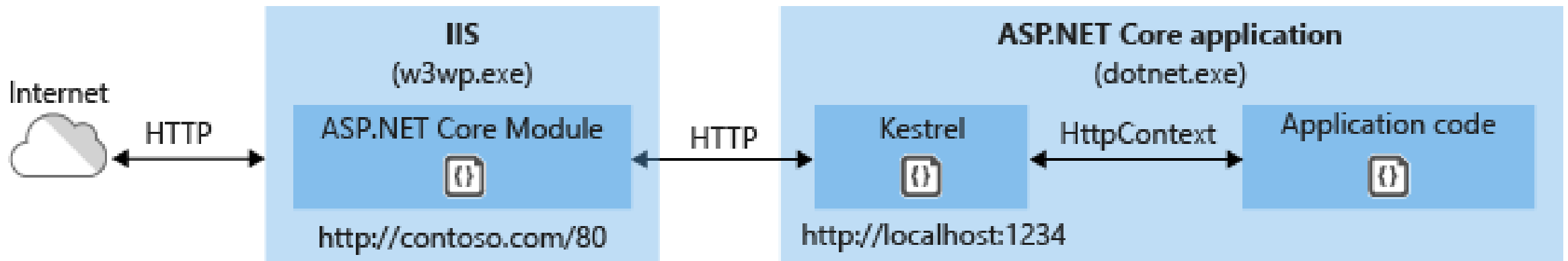
- Native IIS module hooked into IIS pipeline to redirect traffic to backend ASP.NET Core app
- Process management
  - Start dotnet.exe on first request
  - Restarts it when dotnet.exe crashes
- Advantages:
  - IIS App Pool does not run any managed code
  - Existing ASP.NET windows components are not required to be installed
  - Separate process for ASP.NET Core; existing ASP.NET modules can run alongside

```
var builder = new WebHostBuilder()
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseConfiguration(config)
    .UseStartup<Startup>()
    .UseUrls("http://localhost:5001")
    .UseIISIntegration()
    .UseKestrel(options =>
    {
        if (config["threadCount"] != null)
        {
            options.ThreadCount = int.Parse(config["threadCount"]);
        }
    });

var host = builder.Build();
host.Run();
```

# Request flow with ASP.NET Core Module (IIS)

1. Incoming web request is routed to primary port 80/443 through kernel model Http.Sys driver
2. Request forwarded to ASP.NET Core app (on non-80-443 port)
3. Kestrel picks up the request and pushes it into ASP.NET Core middleware pipeline
4. Middleware passes the request to application logic as HttpContext instance
5. Application HTTP response is eventually passed back to IIS





# Hosting Models

- **In-process hosting model**

- ASP.NET Core apps default to the in-process hosting model
- The following characteristics apply when hosting in-process:
  - IIS HTTP Server (IISHttpServer) is used instead of Kestrel server. For in-process, CreateDefaultBuilder calls UseIIS to:
    - Register the IISHttpServer
    - Configure the port and base path the server should listen on when running behind the ASP.NET Core Module
    - Configure the host to capture startup errors
    - Sharing an app pool among apps isn't supported. Use one app pool per app
    - ...

# Hosting Models

- **Out-of-process hosting model**

- To configure an app for out-of-process hosting, set the value of the <AspNetCoreHostingModel> property to OutOfProcess in the project file (.csproj):











```
<PropertyGroup>  
  <AspNetCoreHostingModel>OutOfProcess</AspNetCoreHostingModel>  
</PropertyGroup>
```

- The value of <AspNetCoreHostingModel> is case insensitive, so inprocess and outofprocess are valid values
- Kestrel server is used instead of IIS HTTP Server (IISHttpServer)
- For out-of-process, CreateDefaultBuilder calls UseIISIntegration to:
  - Configure the port and base path the server should listen on when running behind the ASP.NET Core Module.
  - Configure the host to capture startup errors.

# Demo: Hosting Model

Which Web Server Should You Use?

# Choosing Web Servers

	Windows	Linux/OSX	Development-Ready
IIS			
IIS Express			
WebListener			
Kestrel			
Apache/Nginx			

Module 1: ASP.Net Core

Section 3: ASP.NET Core

Lesson: Working Environments

# Working Environments

- ASP.NET Core configures app behavior based on the runtime environment using an environment variable
- ASP.NET Core reads the environment variable **ASPNETCORE\_ENVIRONMENT** at app startup and stores the value in `IWebHostEnvironment.EnvironmentName`.

ASPNETCORE\_ENVIRONMENT can be set to any value, but three values are provided by the framework:

- Development
- Staging
- Production (default)

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    if (env.IsProduction() || env.IsStaging() || env.IsEnvironment("Staging_2"))
    {
        app.UseExceptionHandler("/Error");
    }

    app.UseStaticFiles();
    app.UseMvc();
}
```

# Working Environments

- On Windows and macOS, environment variables and values aren't case sensitive. Linux environment variables and values are case sensitive by default



# Working Environments

- Startup Conventions
  - Startup ➔ Startup{EnvironmentName} – for example, *StartupDevelopment*
  - ConfigureServices( ) ➔ Configure[Environment]Services( )
  - Configure( ) ➔ Configure[Environment]( )
- Applies to Microsoft Azure as well through App Settings in Azure Portal

# Working Environment Configuration

Visual Studio configuration window for 'HelloMvc' project, showing the 'Debug' tab.

Application: Configuration: N/A

Build: Platform: N/A

Debug:

Profile: IIS Express [New... Delete]

Launch: IIS Express

☒ Launch URL: [ ]

☐ Use Specific Runtime:

Version	Platform	Architecture
1.0.0-rc1-update1	.NET Framework	x86

Environment Variables:

Name	Value
Hosting:Environment	Development

[Add Remove]

Module 1: ASP.Net Core

Section 4: .Net 5

Lesson: One .Net Vision

# One .NET Vision – .NET 5 to 6 "wave"

.NET Framework

Single SDK, one BCL, unified toolchain

Cross-platform native UI

.NET

Cross-platform web UI

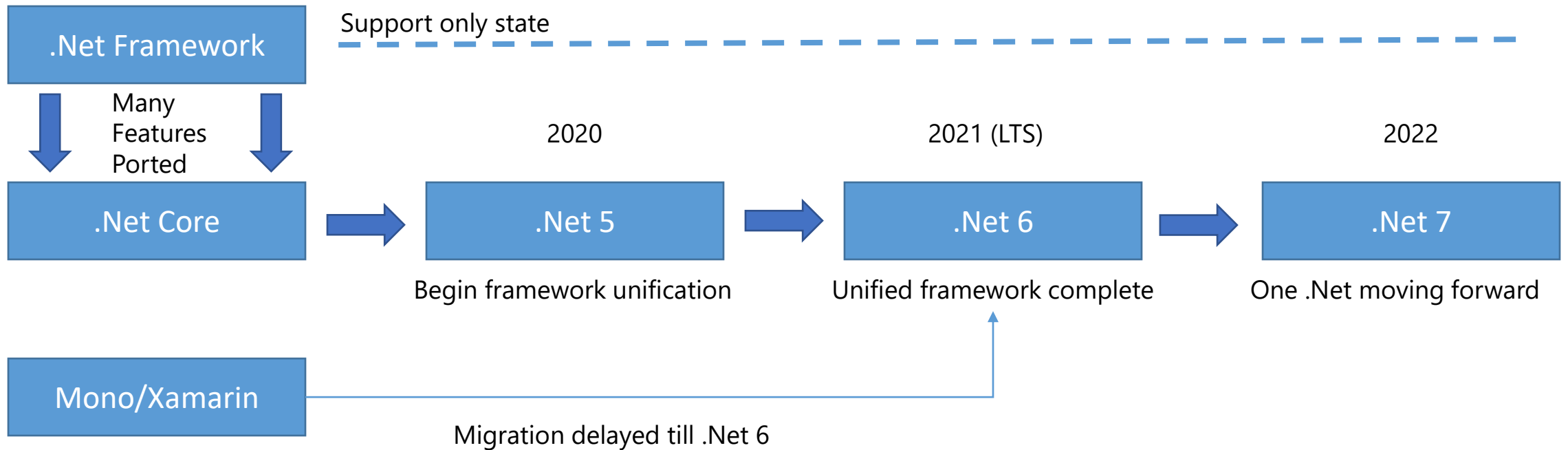
Cloud native investments

Mono / Xamarin

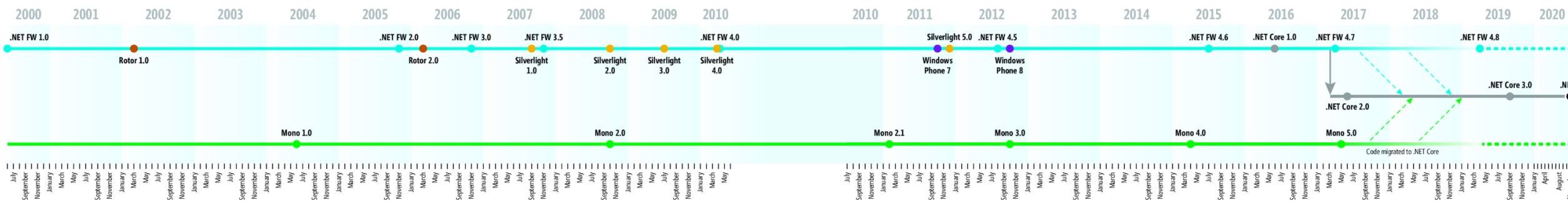
Continue improvements in speed, size, diagnostics, Azure services

**.NET has the best of breed solutions for all modern workloads**

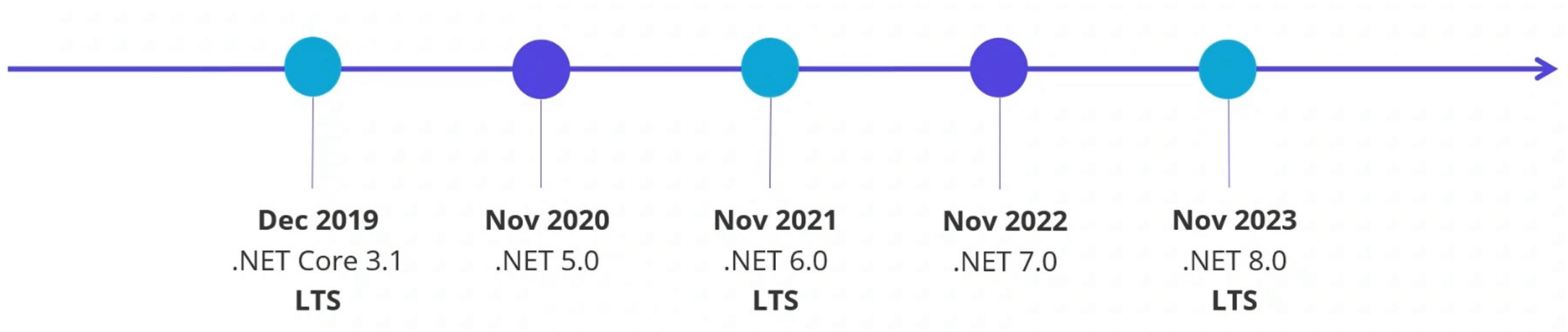
# One .NET Vision – .NET 5 to 6 "wave"



# One .NET Vision – .NET 5 to 6 "wave"



# The Future of One .Net Vision



- .Net 5 released in November 2020
- Major releases every year
- LTS for even numbered releases
- Predictable schedule, minor releases as needed

# What Is Not In .NET 5?

- Web Forms, WCF Server and Windows Workflow remain on .NET Framework 4.8 only. There are no plans to port these
- Recommendations
  - ASP.NET Blazor for ASP.NET Web Forms (we have a [migration guide](#))
  - gRPC for WCF Server and Remoting (we have a [migration guide](#), community port: [CoreWCF](#))
  - Open Source Core Workflow for Windows Workflow (WF): <https://github.com/UiPath/corewf>



# Returning and Current Features in .NET 5.0

Returning Features	New Features
Cross platform support	Significant performance improvements
Strong containerization features	C#9.0 and F#5.0
Extensible tooling	Simplified versioning and platform targets
Powerful cloud integration	New platform compatibility
	Single file programs

# Migrating to .Net 5

Version	ASP.NET Core 2.0- ASP.NET Core 3.1	ASP-NET Core 1.x	ASP.NET Framework
Level of Effort	<b>Low:</b> Simple configuration changes	<b>Medium:</b> Configuration and minor structural changes	<b>High:</b> Architectural, code and configuration changes

# Guidelines for .NET Packages Going Forward

Project Type	.Net Standard 2.0	.Net Standard 2.1	.Net 5
Sharing code between only future projects in .NET 5.0			✓
Sharing code between .NET 5.0, .NET Core 3+, recent versions of Xamarin, but not .NET Framework		✓	
Sharing code between .NET Framework and all other implementations	✓		

# New C# 9 Features - Record Keyword

- C# 9 Introduces a new keyword: record keyword
- record keyword makes an object immutable and behave like a value type
- To make the whole object immutable you have to set init keyword on each property if you are using an implicit parameterless constructor:

```
public record Product
{
    public string Name { get; init; }
    public int CategoryId { get; init; }
}
```

# New C# 9 Features - Top-level Statements

- Top-level statements remove 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!");
```

# New C# 9 Features - Pattern Matching Enhancements

- C# 9 includes new pattern matching improvements

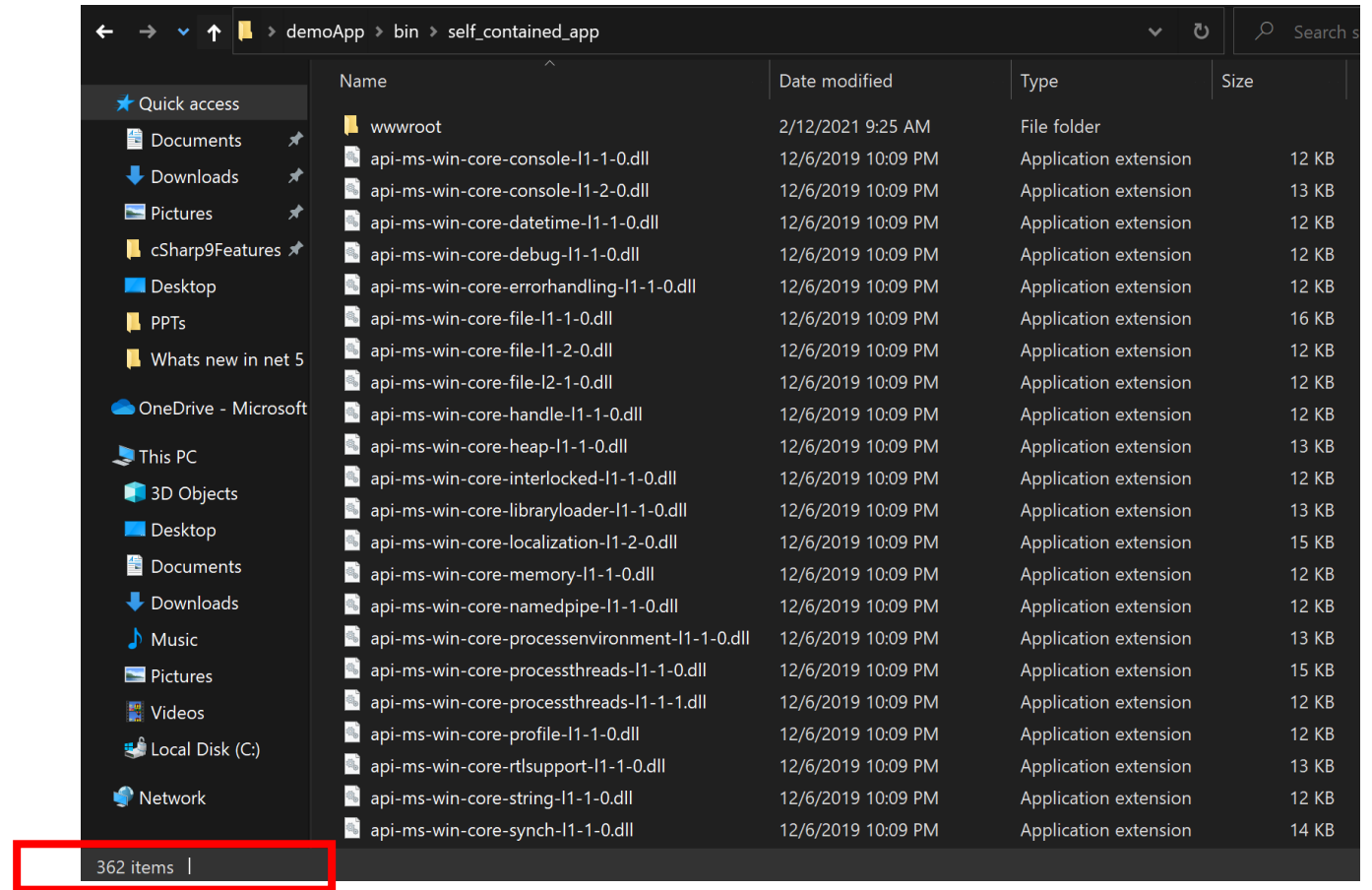
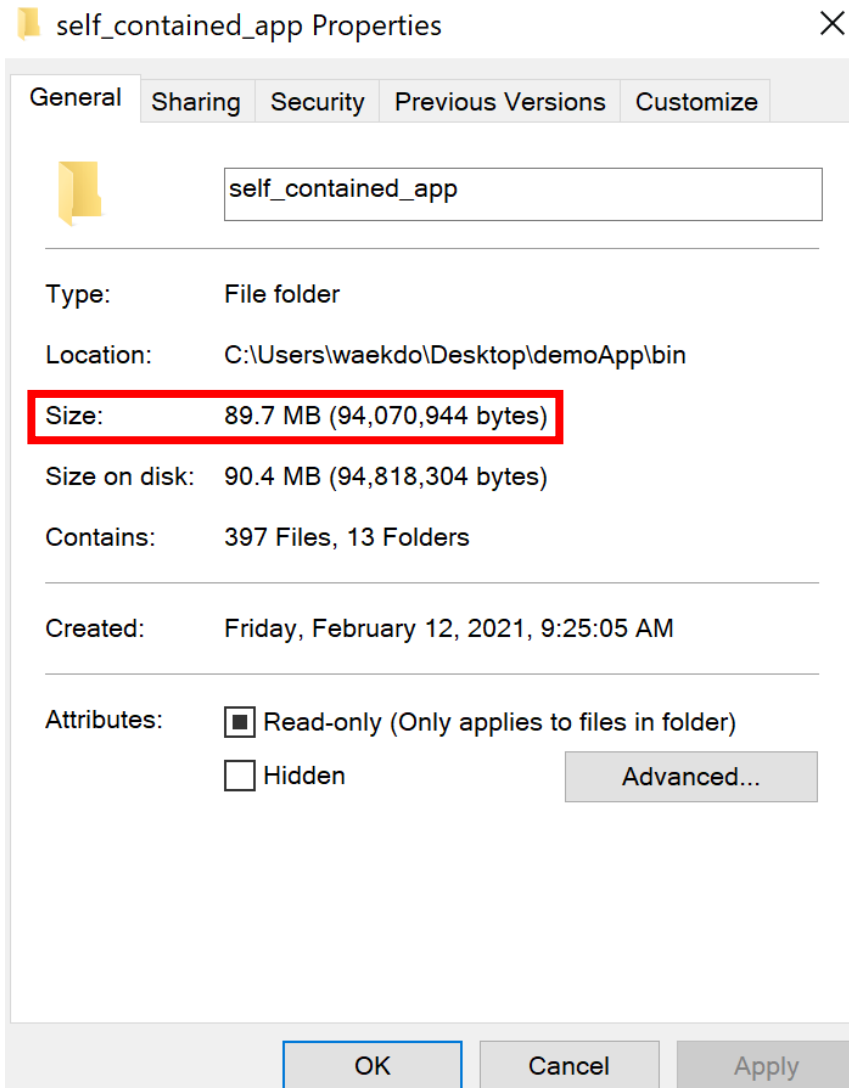
# Demo: C# 9 Features

# Single File App

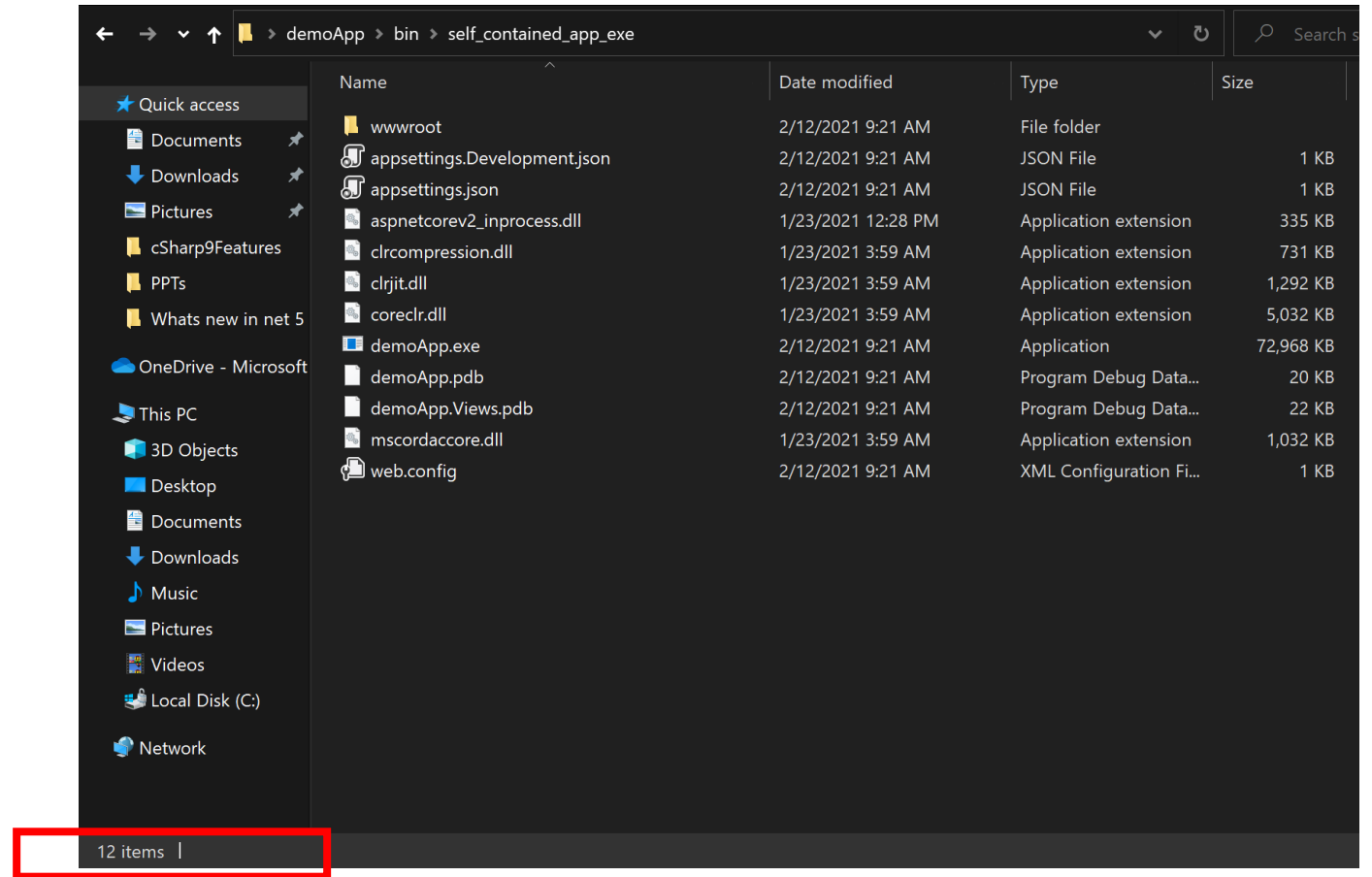
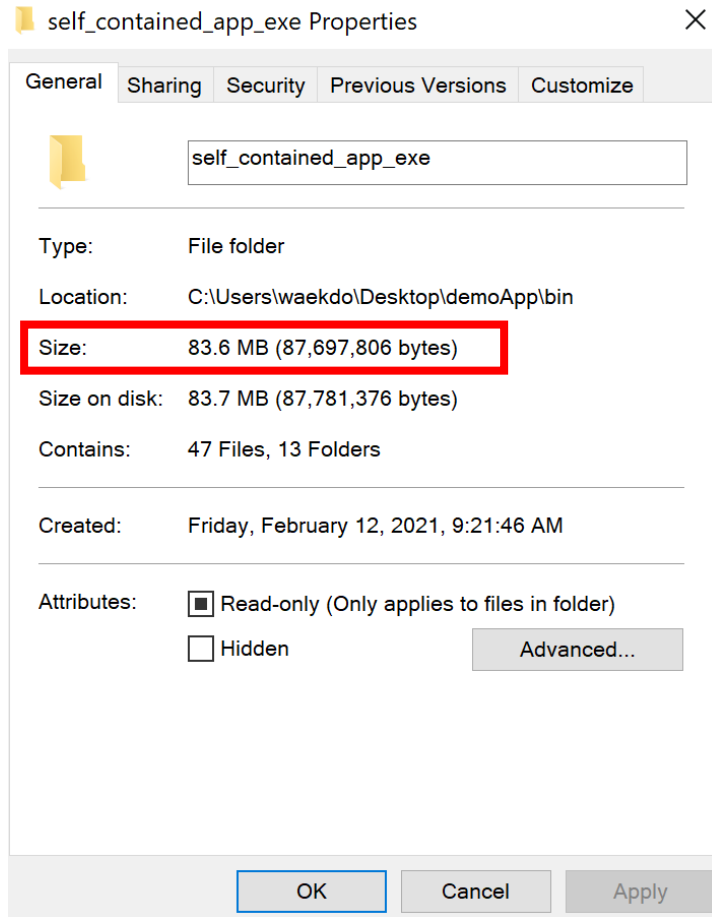
- Bundling all application-dependent files into a single binary provides an attractive option to deploy and distribute the application as a single file
- This deployment model has been available since .NET Core 3.0 and has been **enhanced in .NET 5.0**
- Previously in .NET Core 3.0, when a user runs your single-file app, .NET Core host first extracts all files to a temporary directory before running the application
- .NET 5.0 improves this experience by directly running the code without the need to extract the files from the app



# Publishing A Single EXE File In .NET Core 5.0



# Publishing A Single EXE File In .NET Core 5.0



# Demo: Publishing A Single EXE File In .NET 5.0

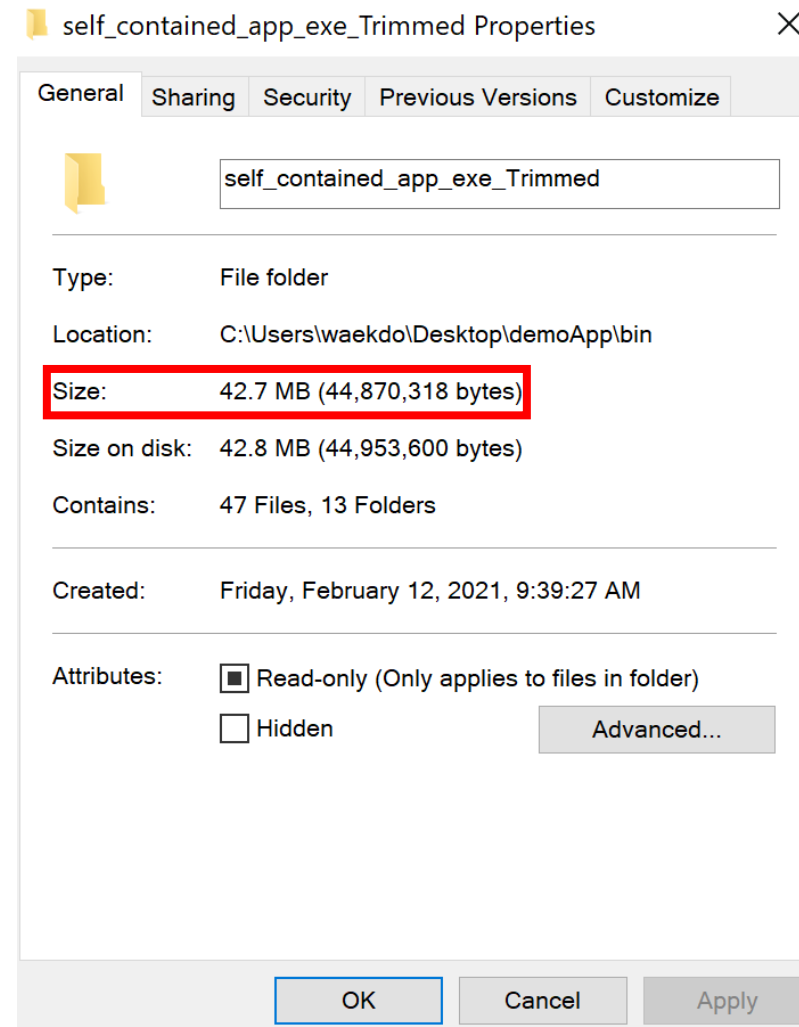
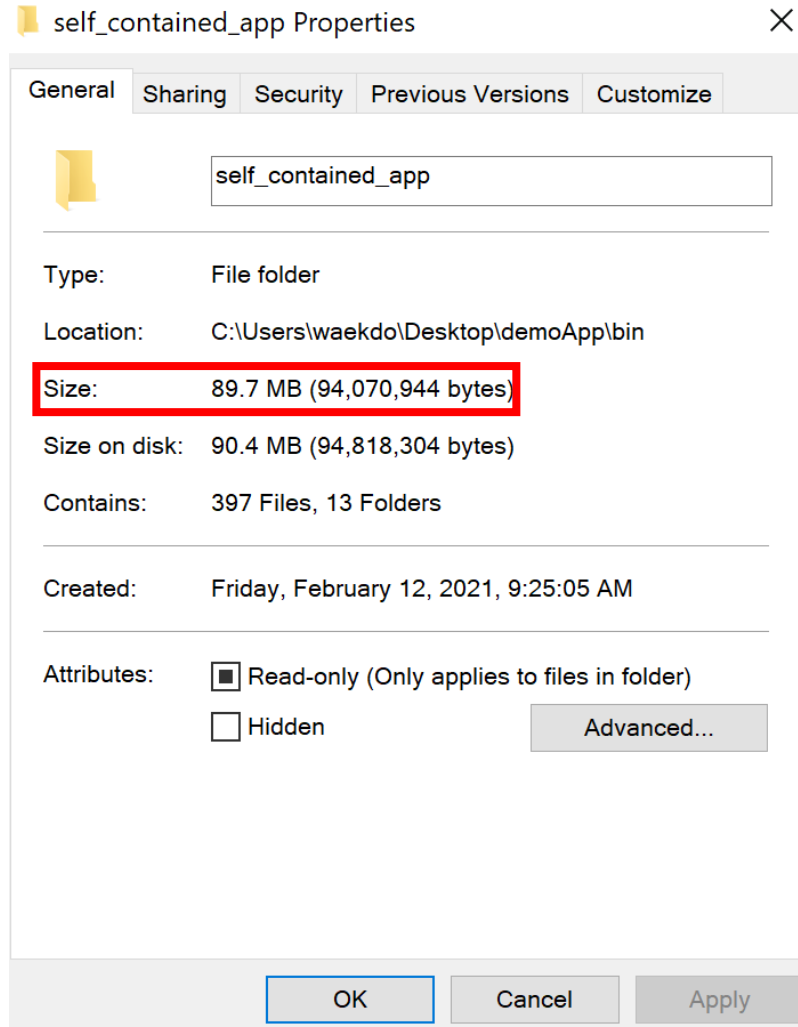
# Assembly Level Trimming/Member Level Trimming

- Assembly Level Trimming
  - Scans and Packages ONLY the used/referred assemblies by the code during Publishing
  - Reduced Application Size
    - `<RuntimeIdentifier>(Target OS such as win10-x64)</RuntimeIdentifier>`
    - `<PublishTrimmed>true</PublishTrimmed>`

# Assembly Level Trimming/Member Level Trimming

- Member Level Trimming (Experimental Feature)
  - Removes Members & Types that were not used by the App
  - Go To Member Level Trimming by adding below to Assembly Level Trimming
    - `<TrimMode>Link</TrimMode>`
- Require thorough Testing
- Not A Production Ready feature

# The PublishTrimmed Flag With IL Linker



## Using .csproj To Create A Reduced Single Executable

```
<PropertyGroup>  
  <TargetFramework>netcoreapp3.0</TargetFramework>  
  <RuntimeIdentifier>win10-x64</RuntimeIdentifier>  
  <PublishSingleFile>true</PublishSingleFile>  
  <PublishTrimmed>true</PublishTrimmed>  
</PropertyGroup>
```

# Demo: The PublishTrimmed Flag With IL Linker



# Reflected Assemblies

- Through various forms of reflection, we may end up loading assemblies at runtime that aren't direct references. Take this (very convoluted) example of loading an assembly at runtime :

```
static void Main(string[] args)
{
    Console.WriteLine(Assembly.Load("System.Security").FullName);
    Console.ReadLine();
}
```

- Now when debugging this locally, and we have .NET Core installed, we ask for System.Security and it knows what that is because we are using the installed .NET Core platform. So running it, we get :

```
System.Security, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a
```

# Reflected Assemblies

- But if we publish this using the PublishTrimmed flag from the command line, then run it :

Unhandled Exception: System.IO.FileNotFoundException: Could not load file or assembly 'System.Security, Culture=neutral, PublicKeyToken=null'.  
The system cannot find the file specified.

# Demo: Reflected Assemblies

# Module Summary

- In this module, you learned about:
  - Fundamentals Of ASP.Net Core
  - .Net Core And .Net Standard; .Net Framework Vs. .Net Core
  - Project Layout And Templates
  - CLI, Middleware, And Hosting Options & Configuration
  - What is New in .Net 5



