Contents

.NET Core Guide

9/24/2019 • 2 minutes to read • Edit Online

.NET Core is an open-source, general-purpose development platform maintained by Microsoft and the .NET community on GitHub. It's cross-platform (supporting Windows, macOS, and Linux) and can be used to build device, cloud, and IoT applications.

See About .NET Core to learn more about .NET Core, including its characteristics, supported languages and frameworks, and key APIs.

Check out .NET Core Tutorials to learn how to create a simple .NET Core application. It only takes a few minutes to get your first app up and running. If you want to try .NET Core in your browser, look at the Numbers in C# online tutorial.

Download .NET Core

Download the .NET Core SDK to try .NET Core on your Windows, macOS, or Linux machine. And if you prefer to use Docker containers, visit the .NET Core Docker Hub.

All .NET Core versions are available at .NET Core Downloads if you're looking for another .NET Core version.

.NET Core 3.0

The latest version is .NET Core 3.0. New features include Windows Desktop support with Windows Presentation Foundation (WPF) and Windows Forms, full stack C# web development with Blazor, new enhancements to SignalR and Azure SignalR Service, new C# language features with C# 8, and much more. For a full listing of the new features in .NET Core 3.0, see What's new in .NET Core 3.0.

Create your first application

After installing the .NET Core SDK, open a command prompt. Type the following `dotnet` commands to create and run a C# application:

```
dotnet new console
dotnet run
```

You should see the following output:

```
Hello World!
```

Support

.NET Core is supported by Microsoft, on Windows, macOS, and Linux. It's updated for security and quality several times a year, typically monthly.

.NET Core binary distributions are built and tested on Microsoft-maintained servers in Azure and supported just like any Microsoft product.

Red Hat supports .NET Core on Red Hat Enterprise Linux (RHEL). Red Hat builds .NET Core from source and makes it available in the Red Hat Software Collections. Red Hat and Microsoft collaborate to ensure that .NET Core works well on RHEL.

About .NET Core

11/1/2019 • 7 minutes to read • Edit Online

.NET Core has the following characteristics:

- **Cross-platform:** Runs on Windows, macOS, and Linux operating systems.
- **Consistent across architectures:** Runs your code with the same behavior on multiple architectures, including x64, x86, and ARM.
- **Command-line tools:** Includes easy-to-use command-line tools that can be used for local development and in continuous-integration scenarios.
- **Flexible deployment:** Can be included in your app or installed side-by-side (user-wide or system-wide installations). Can be used with Docker containers.
- **Compatible:** .NET Core is compatible with .NET Framework, Xamarin, and Mono, via .NET Standard.
- **Open source:** The .NET Core platform is open source, using MIT and Apache 2 licenses. .NET Core is a .NET Foundation project.
- **Supported by Microsoft:** .NET Core is supported by Microsoft, per .NET Core Support.

Languages

C#, Visual Basic, and F# languages can be used to write applications and libraries for .NET Core. These languages can be used in your favorite text editor or Integrated Development Environment (IDE), including:

- Visual Studio
- Visual Studio Code
- Sublime Text
- Vim

This integration is provided, in part, by the contributors of the OmniSharp and Ionide projects.

APIs

.NET Core exposes APIs for many scenarios, a few of which follow:

- Primitive types, such as bool and int.
- Collections, such as System.Collections.Generic.List<T> and System.Collections.Generic.Dictionary<TKey,TValue>.
- Utility types, such as System.Net.Http.HttpClient, and System.IO.FileStream.
- Data types, such as System.Data.DataSet, and DbSet.
- High-performance types, such as System.Numerics.Vector and Pipelines.

.NET Core provides compatibility with .NET Framework and Mono APIs by implementing the .NET Standard specification.

Frameworks

Multiple frameworks have been built on top of .NET Core:

- ASP.NET Core
- Windows 10 Universal Windows Platform (UWP)
- Tizen

Composition

.NET Core is composed of the following parts:

- The .NET Core runtime, which provides a type system, assembly loading, a garbage collector, native interop, and other basic services. .NET Core framework libraries provide primitive data types, app composition types, and fundamental utilities.
- The ASP.NET runtime, which provides a framework for building modern cloud-based internet connected applications, such as web apps, IoT apps, and mobile backends.
- The .NET Core CLI tools and language compilers (Roslyn and F#) that enable the .NET Core developer experience.
- The dotnet tool, which is used to launch .NET Core apps and CLI tools. It selects the runtime and hosts the runtime, provides an assembly loading policy, and launches apps and tools.

These components are distributed in the following ways:

- .NET Core Runtime -- includes the .NET Core runtime and framework libraries.
- ASP.NET Core Runtime -- includes ASP.NET Core and .NET Core runtime and framework libraries.
- .NET Core SDK -- includes the .NET CLI Tools, ASP.NET Core runtime, and .NET Core runtime and framework.

Open source

.NET Core is open source (MIT license) and was contributed to the .NET Foundation by Microsoft in 2014. It's now one of the most active .NET Foundation projects. It can be used by individuals and companies, including for personal, academic, or commercial purposes. Multiple companies use .NET Core as part of apps, tools, new platforms, and hosting services. Some of these companies make significant contributions to .NET Core on GitHub and provide guidance on the product direction as part of the .NET Foundation Technical Steering Group.

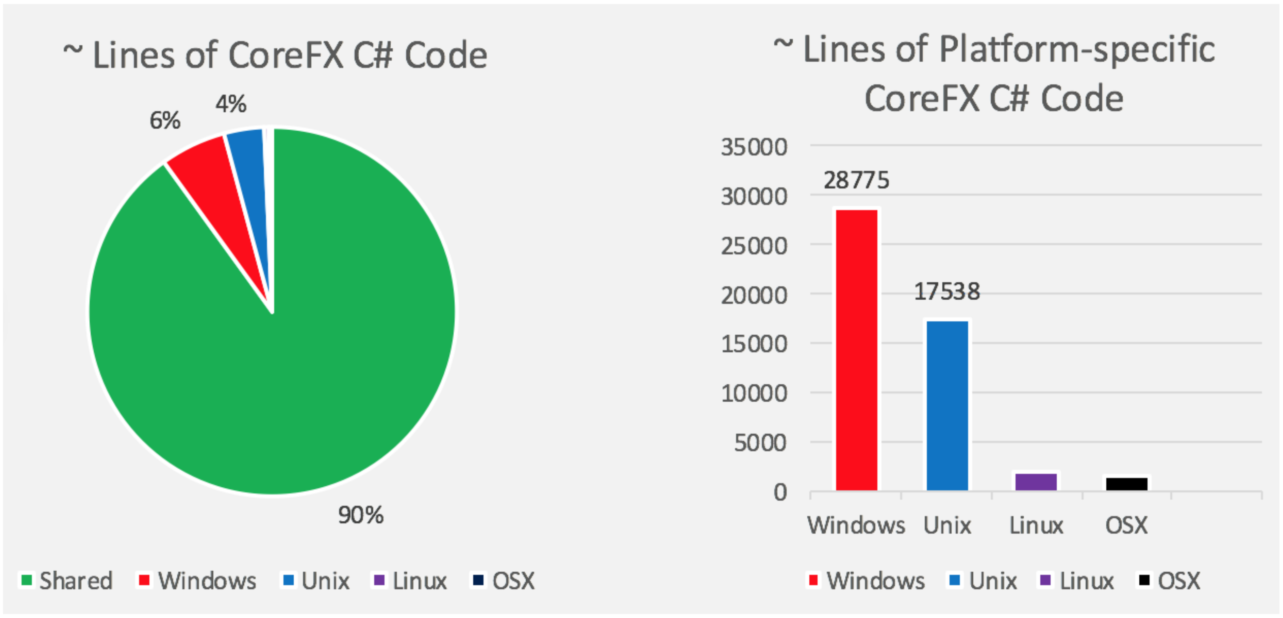
Designed for adaptability

.NET Core has been built as a very similar but unique product compared to other .NET products. It was designed to enable broad adaptability to new platforms and workloads and it has several OS and CPU ports available (and it may be ported to many more).

The product is broken into several pieces, enabling the various parts to be adapted to new platforms at different times. The runtime and platform-specific foundational libraries must be ported as a unit. Platform-agnostic libraries should work as-is on all platforms, by construction. There's a project bias towards reducing platform-specific implementations to increase developer efficiency, preferring platform-neutral C# code whenever an algorithm or API can be implemented in-full or in-part that way.

People commonly ask how .NET Core is implemented in order to support multiple operating systems. They typically ask if there are separate implementations or if conditional compilation is used. It's both, with a strong bias towards conditional compilation.

You can see in the following chart that the vast majority of CoreFX is platform-neutral code that is shared across all platforms. Platform-neutral code can be implemented as a single portable assembly that is used on all platforms.



Windows and Unix implementations are similar in size. Windows has a larger implementation since CoreFX implements some Windows-only features, such as Microsoft.Win32.Registry but doesn't yet implement many Unix-only concepts. You'll also see that the majority of the Linux and macOS implementations are shared across a Unix implementation, while the Linux and macOS-specific implementations are roughly similar in size.

There's a mix of platform-specific and platform-neutral libraries in .NET Core. You can see the pattern in a few examples:

- CoreCLR is platform-specific. It builds on top of OS subsystems, like the memory manager and thread scheduler.
- System.IO and System.Security.Cryptography.Algorithms are platform-specific, given that storage and cryptography APIs are different on each OS.
- System.Collections and System.Linq are platform-neutral, given that they create and operate over data structures.

Comparisons to other .NET implementations

It's probably easier to understand the size and shape of .NET Core by comparing it to existing .NET implementations.

Comparison with .NET Framework

.NET was first announced by Microsoft in 2000 and then evolved from there. The .NET Framework has been the primary .NET implementation produced by Microsoft during that nearly two decade period.

The major differences between .NET Core and the .NET Framework:

- **App-models** -- .NET Core doesn't support all the .NET Framework app-models. In particular, it doesn't support ASP.NET Web Forms and ASP.NET MVC, but it supports ASP.NET Core MVC. And starting with .NET Core 3.0, .NET Core also supports WPF and Windows Forms on Windows only.
- **APIs** -- .NET Core contains a large subset of .NET Framework Base Class Library, with a different factoring (assembly names are different; members exposed on types differ in key cases). In some cases, these differences require changes to port source to .NET Core. For more information, see The .NET Portability Analyzer. .NET Core implements the .NET Standard API specification.
- **Subsystems** -- .NET Core implements a subset of the subsystems in the .NET Framework, with the goal of a simpler implementation and programming model. For example, Code Access Security (CAS) isn't supported, while reflection is supported.
- **Platforms** -- The .NET Framework supports Windows and Windows Server while .NET Core also supports

macOS and Linux.
- **Open Source** -- .NET Core is open source, while a read-only subset of the .NET Framework is open source.

While .NET Core is unique and has significant differences to the .NET Framework and other .NET implementations, it's straightforward to share code between these implementations, using either source or binary sharing techniques.

Because .NET Core supports side-by-side installation and its runtime is completely independent of the .NET Framework, it can be installed on machines with .NET Framework installed without any issues.

Comparison with Mono

Mono is the original cross-platform implementation of .NET. It started out as an open-source alternative to .NET Framework and transitioned to targeting mobile devices as iOS and Android devices became popular. It can be thought of as a community clone of the .NET Framework. The Mono project team relied on the open .NET standards (notably ECMA 335) published by Microsoft to provide a compatible implementation.

The major differences between .NET Core and Mono:

- **App-models** -- Mono supports a subset of the .NET Framework app-models (for example, Windows Forms) and some additional ones for mobile development (for example, Xamarin.iOS) through the Xamarin product. .NET Core doesn't support Xamarin.
- **APIs** -- Mono supports a large subset of the .NET Framework APIs, using the same assembly names and factoring.
- **Platforms** -- Mono supports many platforms and CPUs.
- **Open Source** -- Mono and .NET Core both use the MIT license and are .NET Foundation projects.
- **Focus** -- The primary focus of Mono in recent years is mobile platforms, while .NET Core is focused on cloud and desktop workloads.

The future

It was announced that .NET 5 will be the next release of .NET Core and represents a unification of the platform. The project aims to improve .NET in a few key ways:

- Produce a single .NET runtime and framework that can be used everywhere and that has uniform runtime behaviors and developer experiences.
- Expand the capabilities of .NET by taking the best of .NET Core, .NET Framework, Xamarin and Mono.
- Build that product out of a single code-base that developers (Microsoft and the community) can work on and expand together and that improves all scenarios.

For more details about what's being planned for .NET 5, see Introducing .NET 5.

Get started with .NET Core

10/17/2019 • 2 minutes to read • Edit Online

This article provides information on getting started with .NET Core. .NET Core can be installed on Windows, Linux, and macOS. You can code in your favorite text editor and produce cross-platform libraries and applications.

If you're unsure what .NET Core is, or how it relates to other .NET technologies, start with the What is .NET overview. Put simply, .NET Core is an open-source, cross-platform implementation of .NET.

Create an application

First, download and install the .NET Core SDK on your computer.

Next, open a terminal such as **PowerShell**, **Command Prompt**, or **bash**. Type the following `dotnet` commands to create and run a C# application:

```
dotnet new console --output sample1
dotnet run --project sample1
```

You should see the following output:

```
Hello World!
```

Congratulations! You've created a simple .NET Core application. You can also use Visual Studio Code, Visual Studio (Windows only), or Visual Studio for Mac (macOS only), to create a .NET Core application.

Tutorials

You can get started developing .NET Core applications by following these step-by-step tutorials.

- Windows
- Linux
- macOS

- Build a C# "Hello World" Application with .NET Core in Visual Studio 2017.
- Build a C# class library with .NET Core in Visual Studio 2017.
- Build a Visual Basic "Hello World" application with .NET Core in Visual Studio 2017.
- Build a class library with Visual Basic and .NET Core in Visual Studio 2017.
- Watch a video on how to install and use Visual Studio Code and .NET Core.
- Watch a video on how to install and use Visual Studio 2017 and .NET Core.
- Getting started with .NET Core using the command-line.

See the Prerequisites for Windows development article for a list of the supported Windows versions.

Get started with C# and Visual Studio Code

10/7/2019 • 4 minutes to read • Edit Online

.NET Core gives you a fast and modular platform for creating applications that run on Windows, Linux, and macOS. Use Visual Studio Code with the C# extension to get a powerful editing experience with full support for C# IntelliSense (smart code completion) and debugging.

Prerequisites

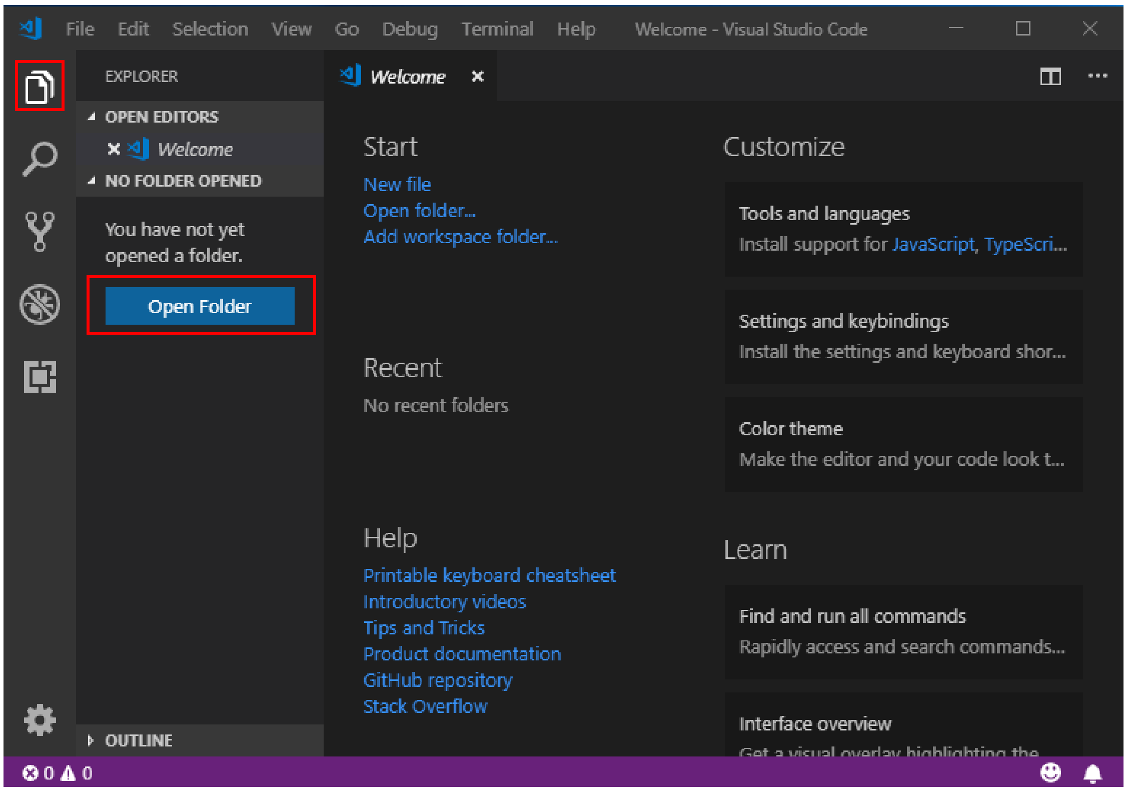
1. Install Visual Studio Code.
2. Install the .NET Core SDK.
3. Install the C# extension for Visual Studio Code. For more information about how to install extensions on Visual Studio Code, see VS Code Extension Marketplace.

Hello World

Let's get started with a simple "Hello World" program on .NET Core:

1. Open a project:

   - Open Visual Studio Code.
   - Click on the Explorer icon on the left menu and then click **Open Folder**.
   - Select **File** > **Open Folder** from the main menu to open the folder you want your C# project to be in and click **Select Folder**. For our example, we're creating a folder for our project named *HelloWorld*.



You have not yet opened a folder.

Install support for JavaScript, TypeScri...

Install the settings and keyboard shor...

Make the editor and your code look t...

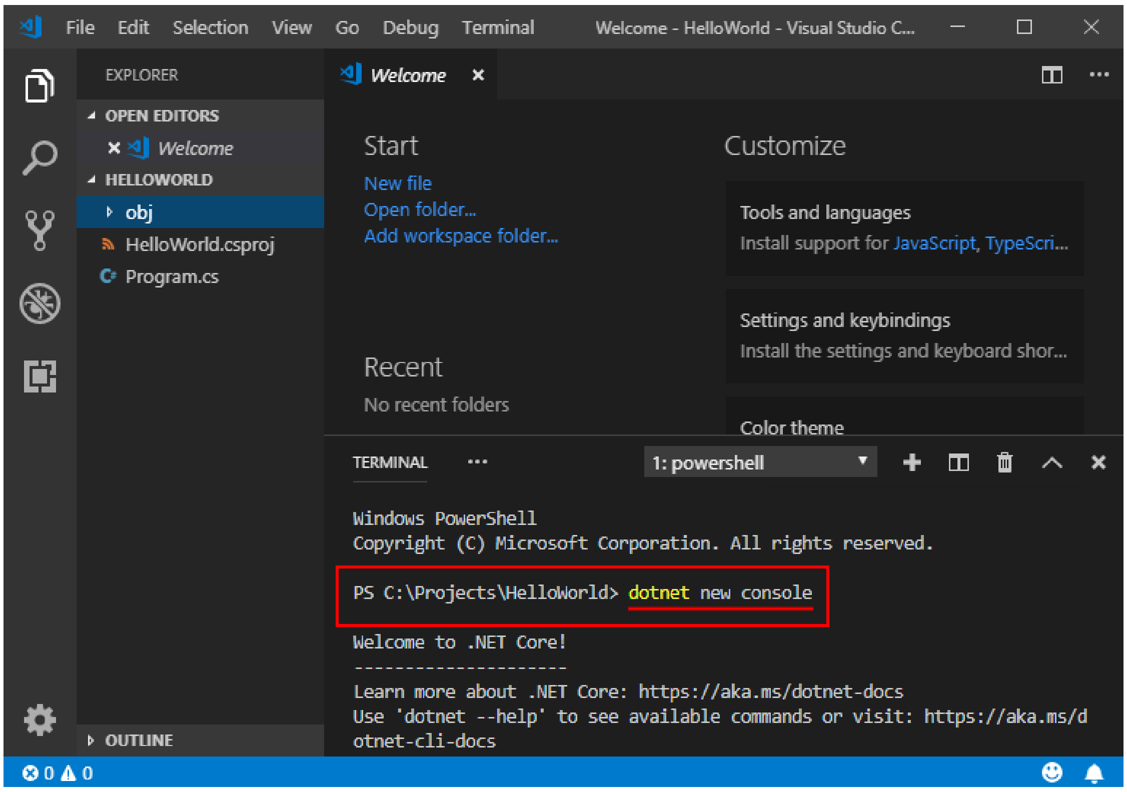
Rapidly access and search commands...

Get a visual overlay highlighting the...

2. Initialize a C# project:

   - Open the Integrated Terminal from Visual Studio Code by selecting **View** > **Integrated Terminal** from the main menu.

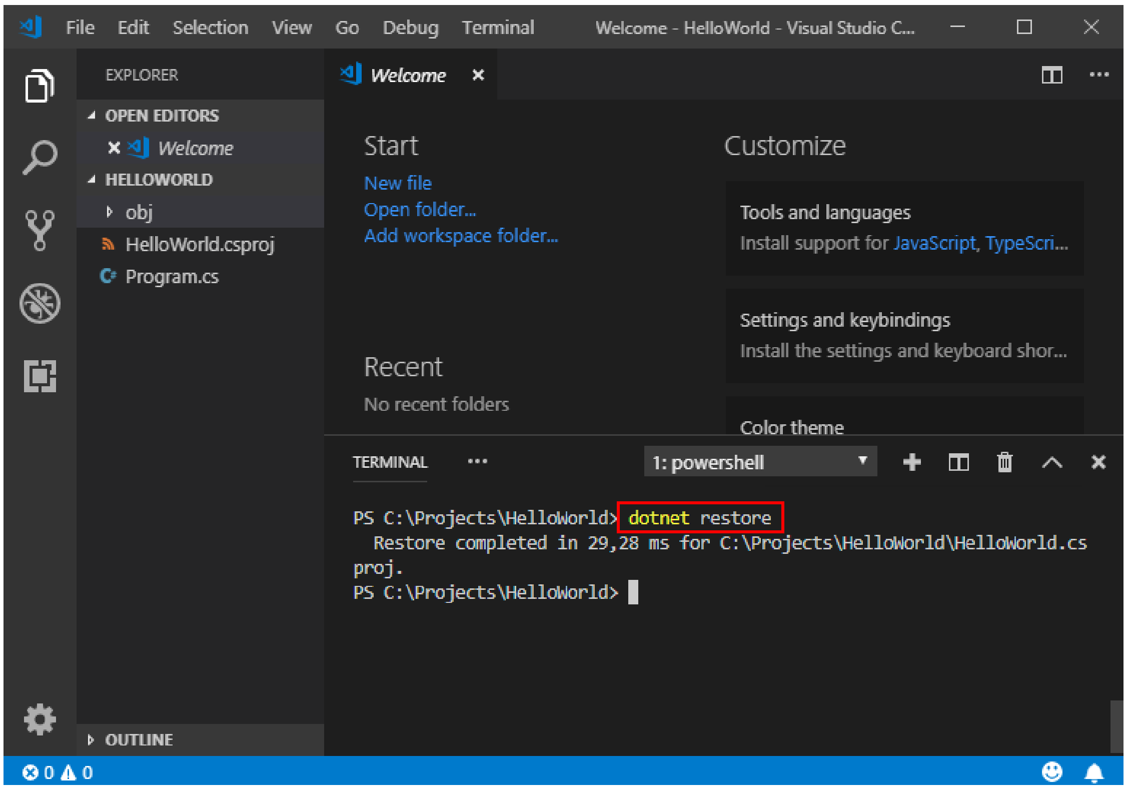
- In the terminal window, type `dotnet new console`.
- This command creates a *Program.cs* file in your folder with a simple "Hello World" program already written, along with a C# project file named *HelloWorld.csproj*.



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Projects\HelloWorld> dotnet new console

Welcome to .NET Core!
---------------------
Learn more about .NET Core: https://aka.ms/dotnet-docs
Use 'dotnet --help' to see available commands or visit: https://aka.ms/d
otnet-cli-docs
```

3. Resolve the build assets:

- For **.NET Core 1.x**, type `dotnet restore`. Running `dotnet restore` gives you access to the required .NET Core packages that are needed to build your project.



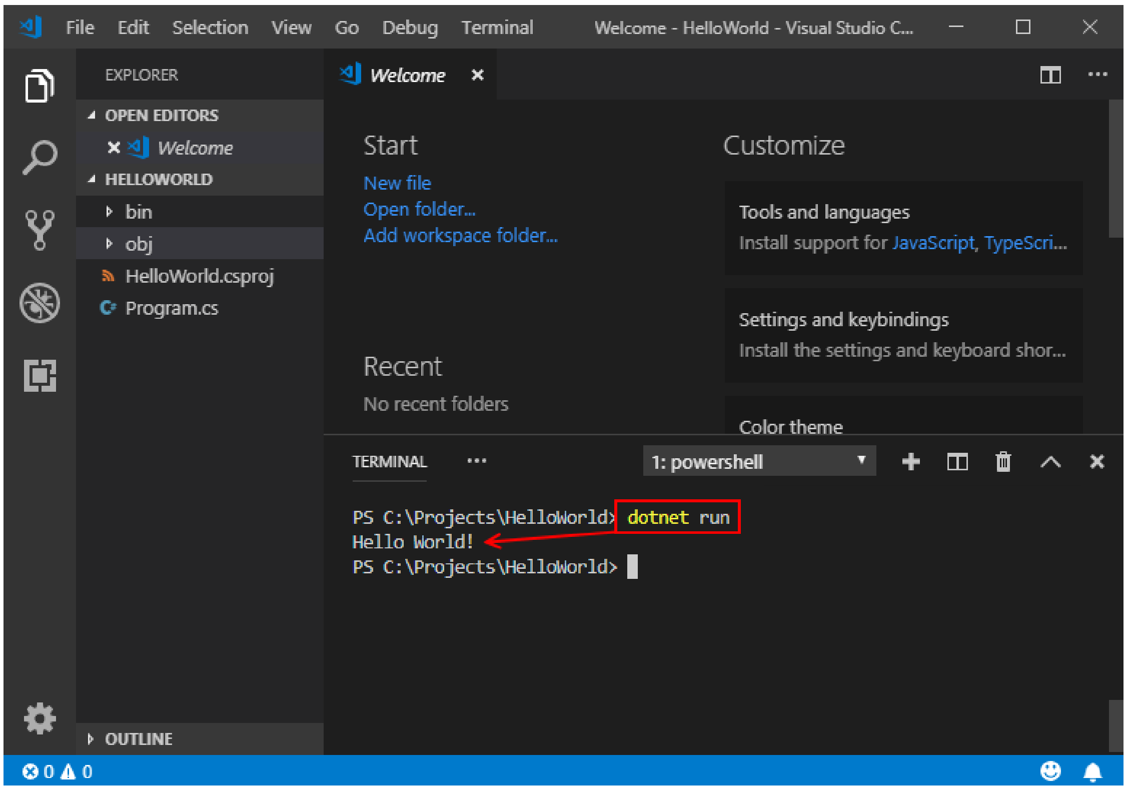
```
PS C:\Projects\HelloWorld> dotnet restore
  Restore completed in 29,28 ms for C:\Projects\HelloWorld\HelloWorld.cs
proj.
PS C:\Projects\HelloWorld>
```

NOTE

Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

4. Run the "Hello World" program:

   - Type `dotnet run`.

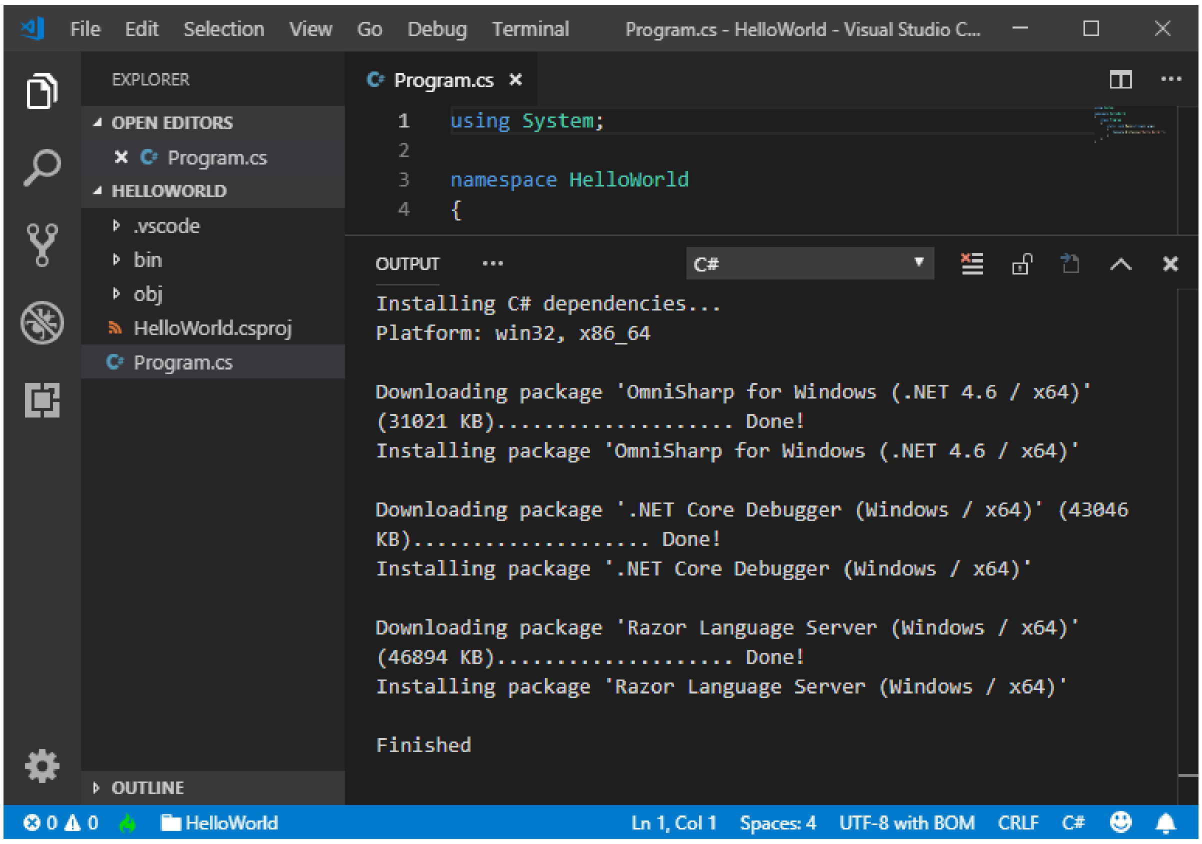


```
PS C:\Projects\HelloWorld> dotnet run
Hello World!
PS C:\Projects\HelloWorld>
```

You can also watch a short video tutorial for further setup help on Windows, macOS, or Linux.

Debug

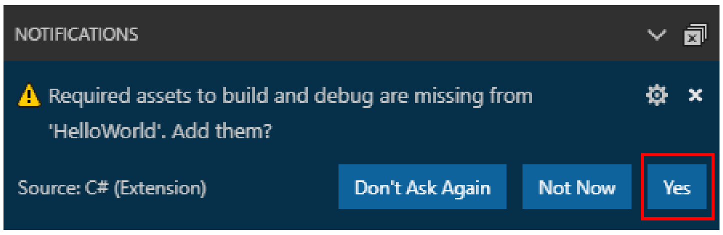
1. Open *Program.cs* by clicking on it. The first time you open a C# file in Visual Studio Code, OmniSharp loads in the editor.



```
using System;

namespace HelloWorld
{
```

```
Installing C# dependencies...
Platform: win32, x86_64

Downloading package 'OmniSharp for Windows (.NET 4.6 / x64)'
(31021 KB).................... Done!
Installing package 'OmniSharp for Windows (.NET 4.6 / x64)'

Downloading package '.NET Core Debugger (Windows / x64)' (43046
KB).................... Done!
Installing package '.NET Core Debugger (Windows / x64)'

Downloading package 'Razor Language Server (Windows / x64)'
(46894 KB).................... Done!
Installing package 'Razor Language Server (Windows / x64)'

Finished
```

2. Visual Studio Code should prompt you to add the missing assets to build and debug your app. Select **Yes**.

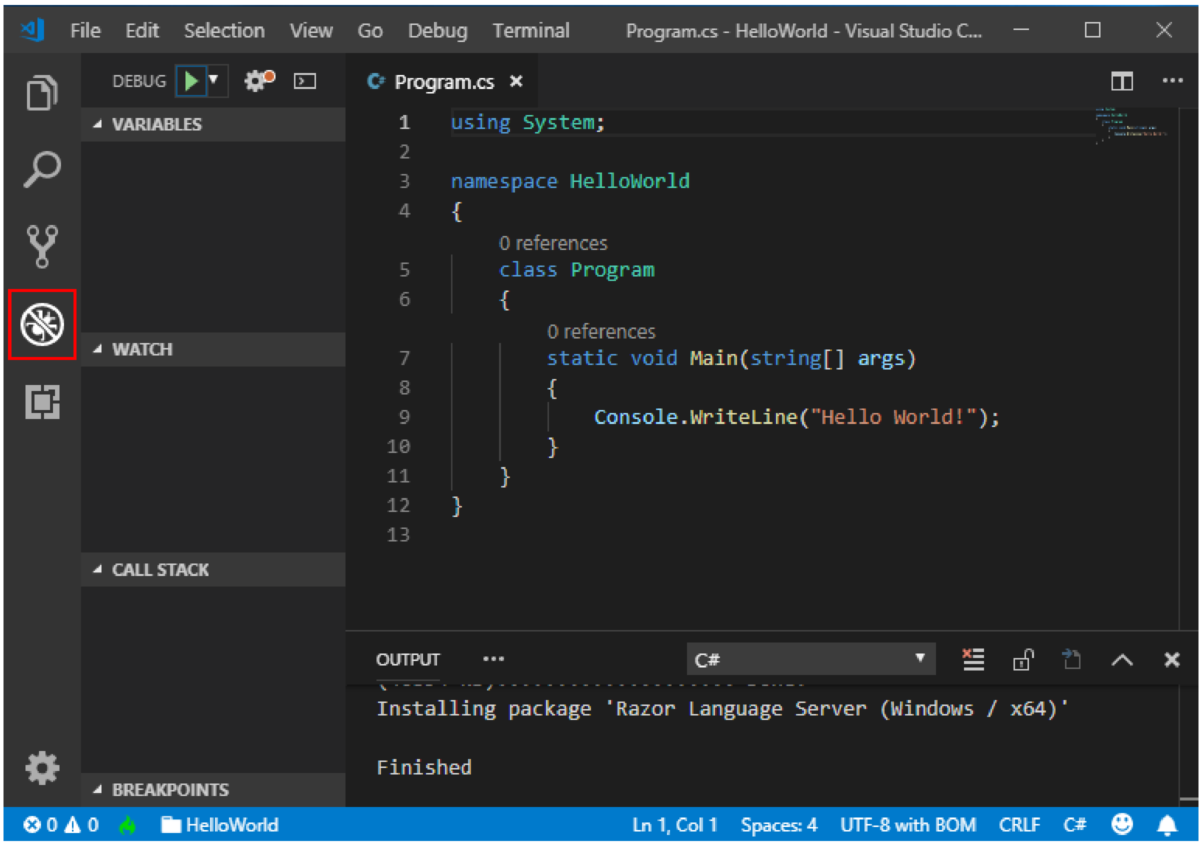


NOTIFICATIONS

Required assets to build and debug are missing from 'HelloWorld'. Add them?

Source: C# (Extension)

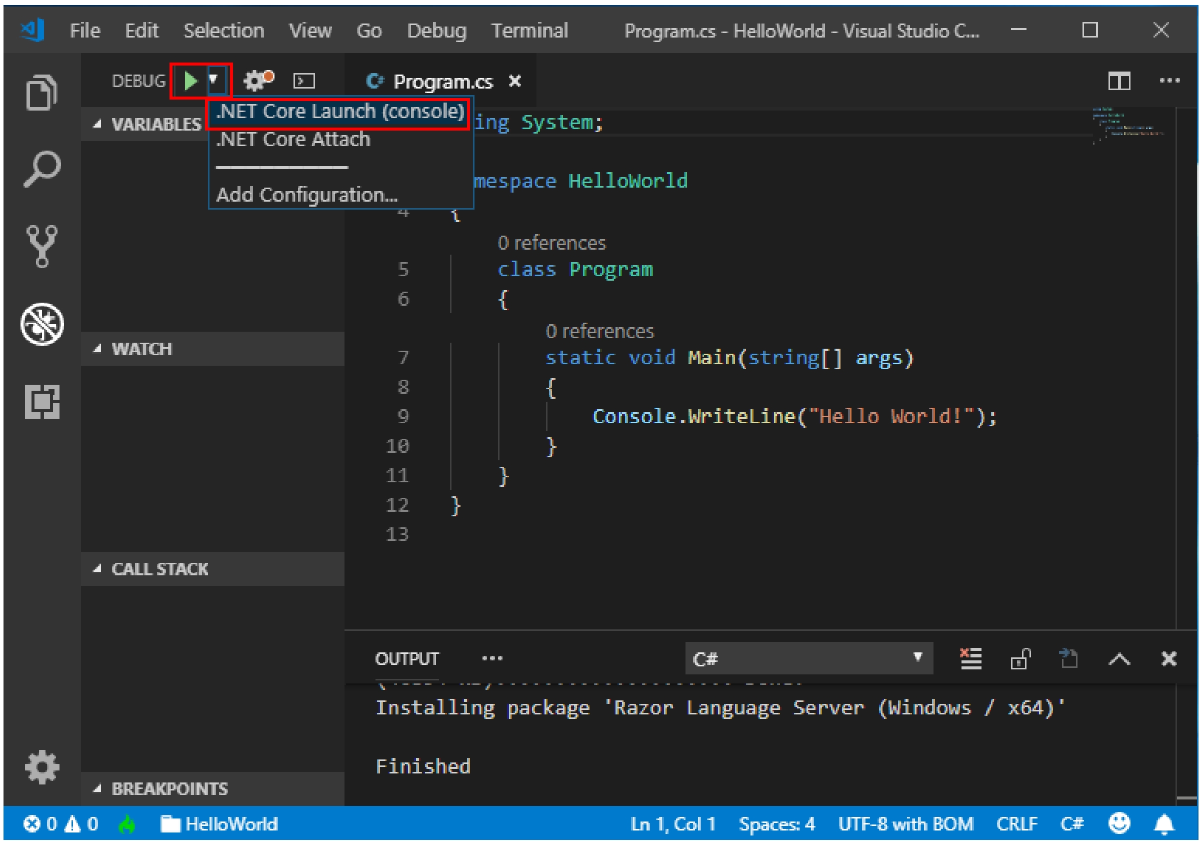
3. To open the Debug view, click on the Debugging icon on the left side menu.



```
using System;

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

```
Installing package 'Razor Language Server (Windows / x64)'

Finished
```

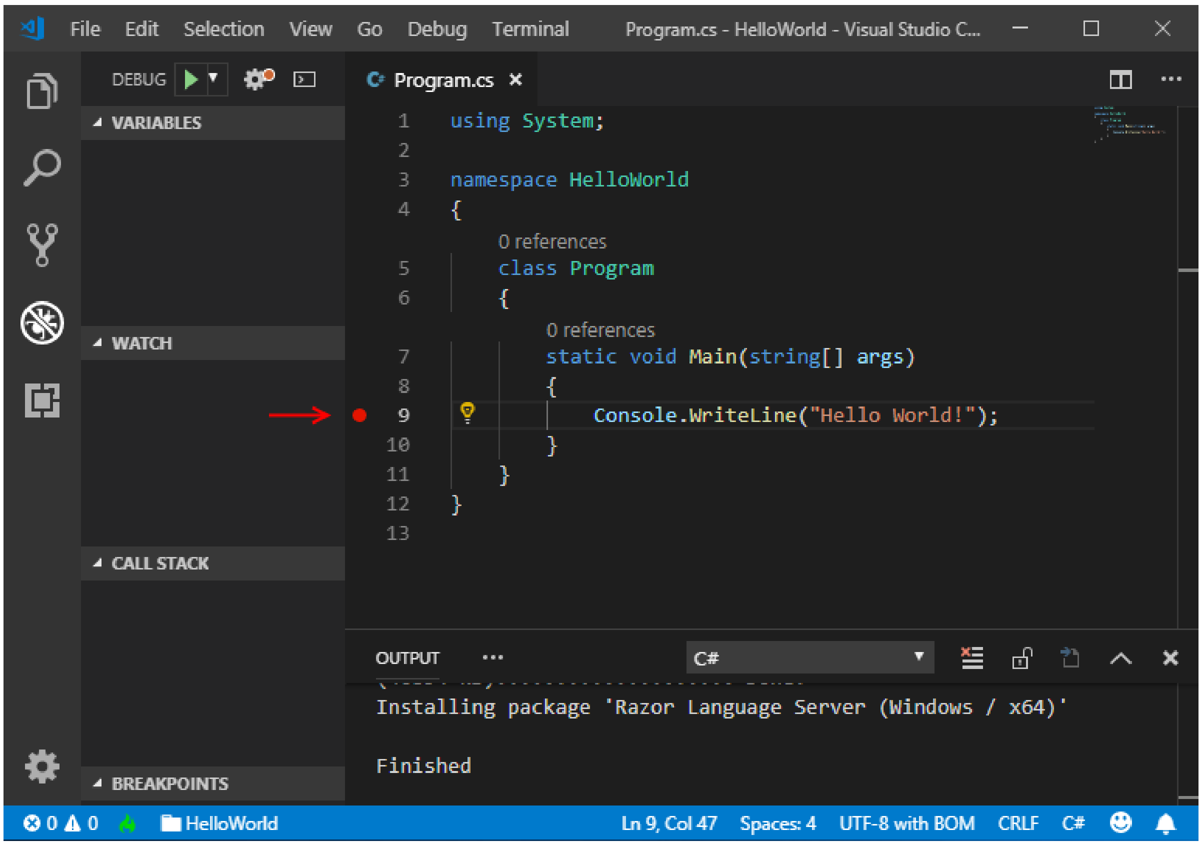
4. Locate the green arrow at the top of the pane. Make sure the drop-down next to it has **.NET Core Launch (console)** selected.



```
      ing System;

      mespace HelloWorld
 4    {
          0 references
 5        class Program
 6        {
              0 references
 7            static void Main(string[] args)
 8            {
 9                Console.WriteLine("Hello World!");
10            }
11        }
12    }
13
```

```
Installing package 'Razor Language Server (Windows / x64)'

Finished
```

5. Add a breakpoint to your project by clicking on the **editor margin**, which is the space on the left of the line numbers in the editor, next to line 9, or move the text cursor onto line 9 in the editor and press F9.



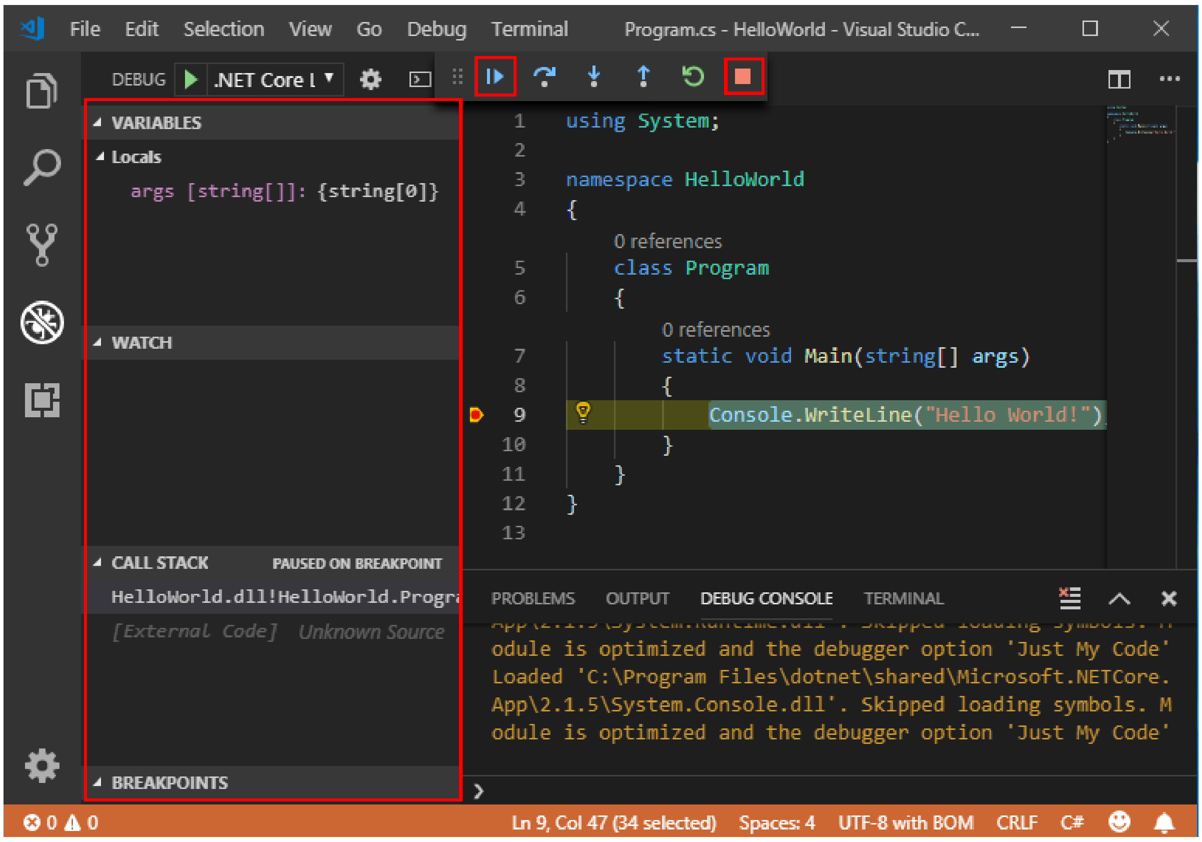
```
 1    using System;
 2
 3    namespace HelloWorld
 4    {
          0 references
 5        class Program
 6        {
              0 references
 7            static void Main(string[] args)
 8            {
 9                Console.WriteLine("Hello World!");
10            }
11        }
12    }
13
```

```
Installing package 'Razor Language Server (Windows / x64)'

Finished
```

6. To start debugging, press F5 or select the green arrow. The debugger stops execution of your program when it reaches the breakpoint you set in the previous step.

- While debugging, you can view your local variables in the top left pane or use the debug console.

7. Select the blue arrow at the top to continue debugging, or select the red square at the top to stop.



```
args [string[]]: {string[0]}
```

```
using System;

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

```
HelloWorld.dll!HelloWorld.Progra
[External Code]  Unknown Source
```

```
odule is optimized and the debugger option 'Just My Code'
Loaded 'C:\Program Files\dotnet\shared\Microsoft.NETCore.
App\2.1.5\System.Console.dll'. Skipped loading symbols. M
odule is optimized and the debugger option 'Just My Code'
```

⊗ 0 ⚠ 0   Ln 9, Col 47 (34 selected)   Spaces: 4   UTF-8 with BOM   CRLF   C#

TIP

For more information and troubleshooting tips on .NET Core debugging with OmniSharp in Visual Studio Code, see Instructions for setting up the .NET Core debugger.

Add a class

1. To add a new class, right click in the VSCode Explorer and select **New File**. This adds a new file to the folder you have open in VSCode.

2. Name your file *MyClass.cs*. You must save it with a `.cs` extension at the end for it to be recognized as a csharp file.

3. Add the code below to create your first class. Make sure to include the correct namespace so you can reference it from your *Program.cs* file:

```
using System;

namespace HelloWorld
{
    public class MyClass
    {
        public string ReturnMessage()
        {
            return "Happy coding!";
        }
    }
}
```

4. Call your new class from your main method in *Program.cs* by adding the code below:

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            var c1 = new MyClass();
            Console.WriteLine($"Hello World! {c1.ReturnMessage()}");
        }
    }
}
```

5. Save your changes and run your program again. The new message should appear with the appended string.

```
> dotnet run
Hello World! Happy coding!
```

FAQ

I'm missing required assets to build and debug C# in Visual Studio Code. My debugger says "No Configuration."

The Visual Studio Code C# extension can generate assets to build and debug for you. Visual Studio Code prompts you to generate these assets when you first open a C# project. If you didn't generate assets then, you can still run this command by opening the Command Palette (**View > Command Palette**) and typing ">.NET: Generate Assets for Build and Debug". Selecting this generates the *.vscode*, *launch.json*, and *tasks.json* configuration files that you need.

See also

- Setting up Visual Studio Code
- Debugging in Visual Studio Code

Build a C# Hello World application with the .NET Core SDK in Visual Studio 2017

11/3/2019 • 3 minutes to read • Edit Online

This article provides a step-by-step introduction to building, debugging, and publishing a simple .NET Core console application using C# in Visual Studio 2017. Visual Studio 2017 provides a full-featured development environment for building .NET Core applications. As long as the application doesn't have platform-specific dependencies, the application can run on any platform that .NET Core targets and on any system that has .NET Core installed.

Prerequisites

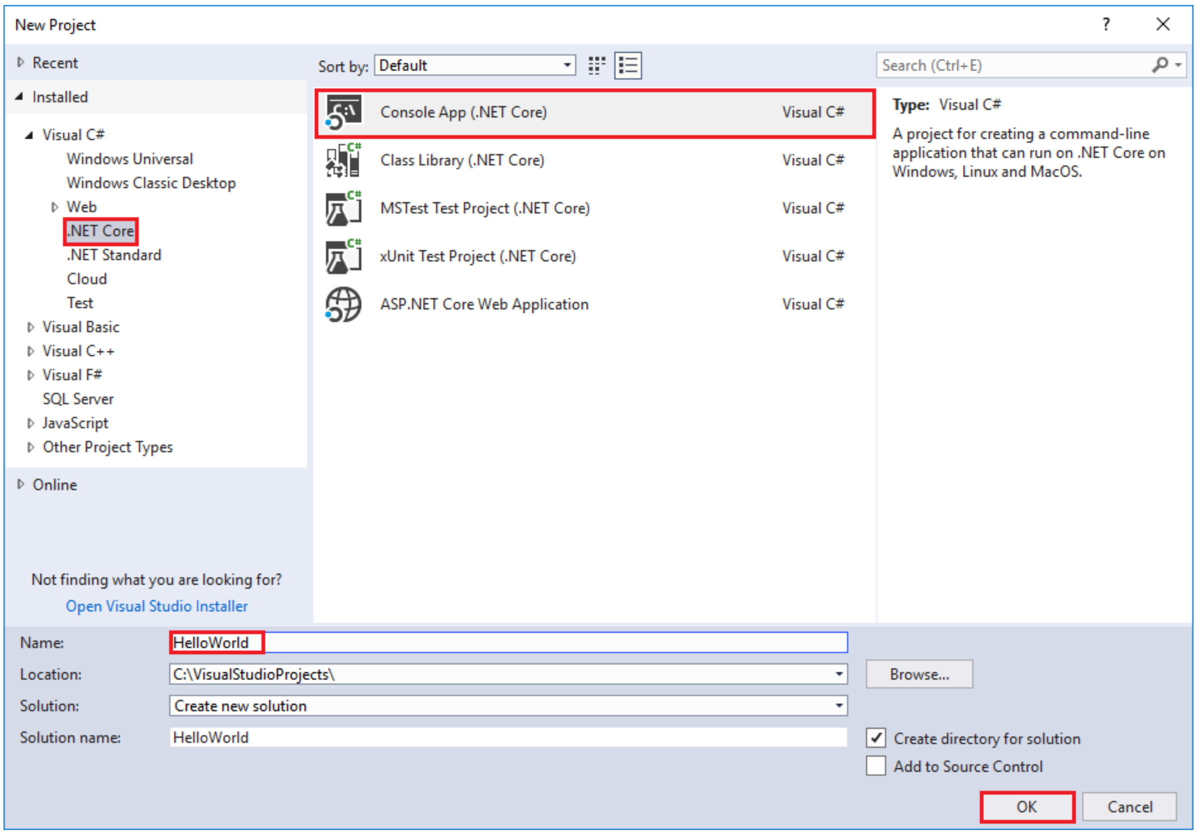
Visual Studio 2017 or later with the ".NET Core cross-platform development" workload installed. You can develop your app with .NET Core 2.1 or later versions.

For more information, see the Prerequisites for .NET Core on Windows article.

A simple Hello World application

Begin by creating a simple "Hello World" console application. Follow these steps:

1. Launch Visual Studio. Select **File** > **New** > **Project** from the menu bar. In the **New Project** dialog, select the **Visual C#** node followed by the **.NET Core** node. Then select the **Console App (.NET Core)** project template. In the **Name** text box, type "HelloWorld". Select the **OK** button.



Console App (.NET Core) — Visual C#
Class Library (.NET Core) — Visual C#
MSTest Test Project (.NET Core) — Visual C#
xUnit Test Project (.NET Core) — Visual C#
ASP.NET Core Web Application — Visual C#

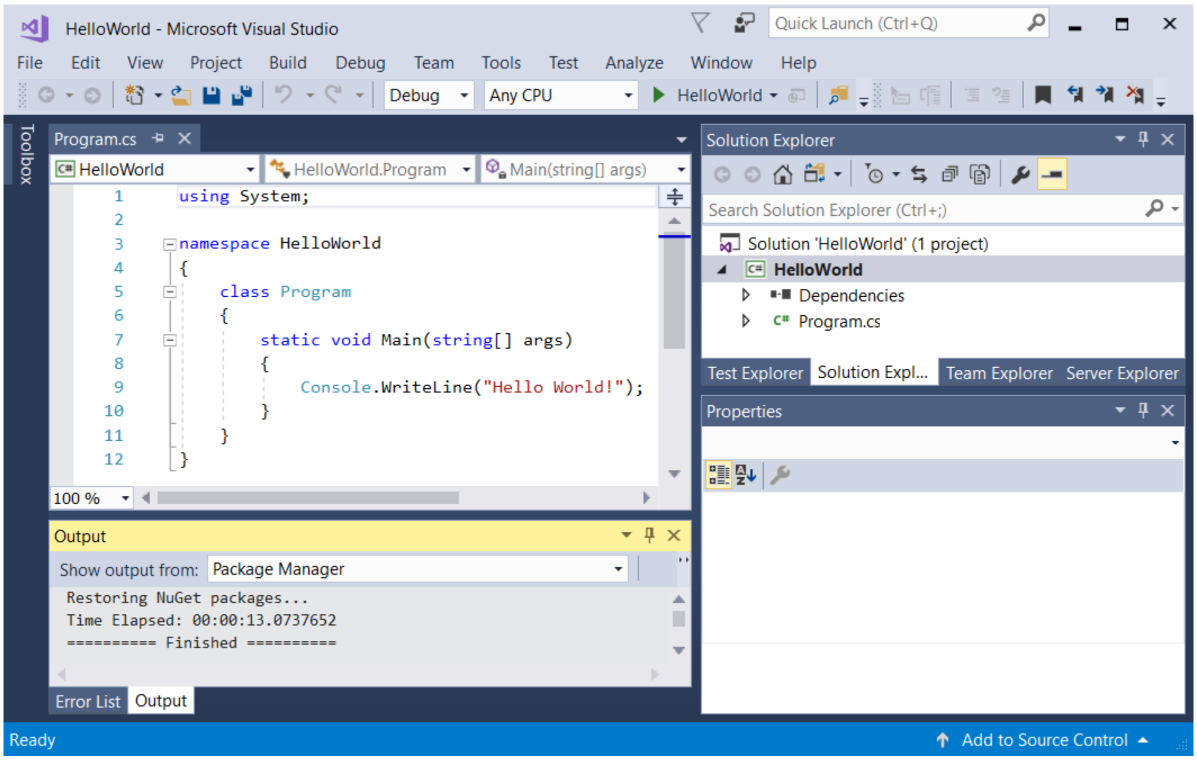
Type: Visual C#

A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.

Name: HelloWorld
Location: C:\VisualStudioProjects\
Solution: Create new solution
Solution name: HelloWorld
[✓] Create directory for solution
[ ] Add to Source Control

2. Visual Studio uses the template to create your project. The C# Console Application template for .NET Core automatically defines a class, `Program`, with a single method, `Main`, that takes a String array as an argument. `Main` is the application entry point, the method that's called automatically by the runtime when

it launches the application. Any command-line arguments supplied when the application is launched are available in the *args* array.



```
using System;

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

```
Restoring NuGet packages...
Time Elapsed: 00:00:13.0737652
========== Finished ==========
```

The template creates a simple "Hello World" application. It calls the Console.WriteLine(String) method to display the literal string "Hello World!" in the console window. By selecting the **HelloWorld** button with the green arrow on the toolbar, you can run the program in Debug mode. If you do, the console window is visible for only a brief time interval before it closes. This occurs because the `Main` method terminates and the application ends as soon as the single statement in the `Main` method executes.

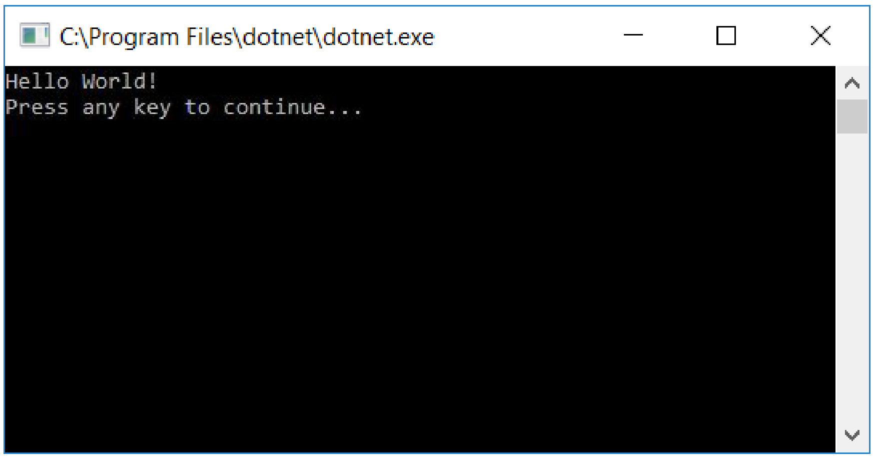
3. To cause the application to pause before it closes the console window, add the following code immediately after the call to the Console.WriteLine(String) method:

   ```
   Console.Write("Press any key to continue...");
   Console.ReadKey(true);
   ```

   This code prompts the user to press any key and then pauses the program until a key is pressed.

4. On the menu bar, select **Build** > **Build Solution**. This compiles your program into an intermediate language (IL) that's converted into binary code by a just-in-time (JIT) compiler.

5. Run the program by selecting the **HelloWorld** button with the green arrow on the toolbar.



```
Hello World!
Press any key to continue...
```

6. Press any key to close the console window.

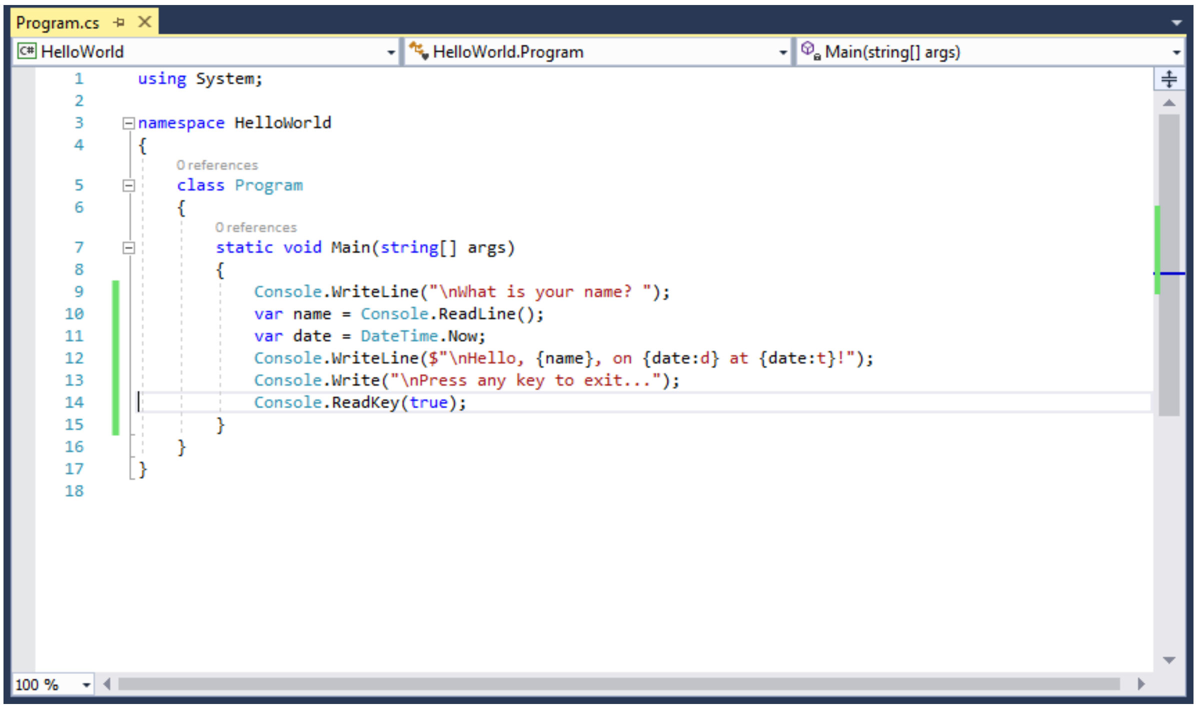
Enhancing the Hello World application

Enhance your application to prompt the user for their name and display it along with the date and time. To modify and test the program, do the following:

1. Enter the following C# code in the code window immediately after the opening bracket that follows the `static void Main(string[] args)` line and before the first closing curly bracket:

   ```
   Console.WriteLine("\nWhat is your name? ");
   var name = Console.ReadLine();
   var date = DateTime.Now;
   Console.WriteLine($"\nHello, {name}, on {date:d} at {date:t}!");
   Console.Write("\nPress any key to exit...");
   Console.ReadKey(true);
   ```

   This code replaces the contents of the `Main` method.

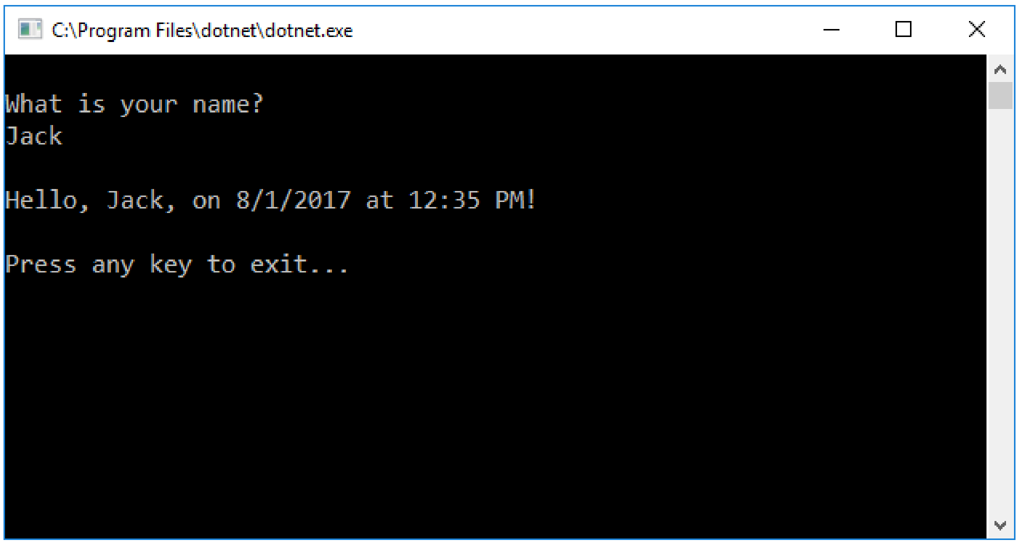
   

   ```
   using System;

   namespace HelloWorld
   {
       class Program
       {
           static void Main(string[] args)
           {
               Console.WriteLine("\nWhat is your name? ");
               var name = Console.ReadLine();
               var date = DateTime.Now;
               Console.WriteLine($"\nHello, {name}, on {date:d} at {date:t}!");
               Console.Write("\nPress any key to exit...");
               Console.ReadKey(true);
           }
       }
   }
   ```

   This code displays "What is your name?" in the console window and waits until the user enters a string followed by the Enter key. It stores this string into a variable named `name`. It also retrieves the value of the DateTime.Now property, which contains the current local time, and assigns it to a variable named `date`. Finally, it uses an interpolated string to display these values in the console window.

2. Compile the program by choosing **Build** > **Build Solution**.

3. Run the program in Debug mode in Visual Studio by selecting the green arrow on the toolbar, pressing F5, or choosing the **Debug** > **Start Debugging** menu item. Respond to the prompt by entering a name and pressing the Enter key.



```
What is your name?
Jack

Hello, Jack, on 8/1/2017 at 12:35 PM!

Press any key to exit...
```

4. Press any key to close the console window.

You've created and run your application. To develop a professional application, take some additional steps to make your application ready for release:

- For information on debugging your application, see Debug your .NET Core Hello World application using Visual Studio 2017.
- For information on developing and publishing a distributable version of your application, see Publish your .NET Core Hello World application with Visual Studio 2017.

Related articles

Instead of a console application, you can also build a class library with .NET Core and Visual Studio 2017. For a step-by-step introduction, see Building a class library with C# and .NET Core in Visual Studio 2017.

You can also develop a .NET Core console app on Mac, Linux, and Windows by using Visual Studio Code, a downloadable code editor. For a step-by-step tutorial, see Getting Started with Visual Studio Code.

Build a Visual Basic Hello World application with the .NET Core SDK in Visual Studio 2017

10/30/2019 • 3 minutes to read • Edit Online

This topic provides a step-by-step introduction to building, debugging, and publishing a simple .NET Core console application using Visual Basic in Visual Studio 2017. Visual Studio 2017 provides a full-featured development environment for building .NET Core applications. As long as the application doesn't have platform-specific dependencies, the application can run on any platform that .NET Core targets and on any system that has .NET Core installed.

Prerequisites

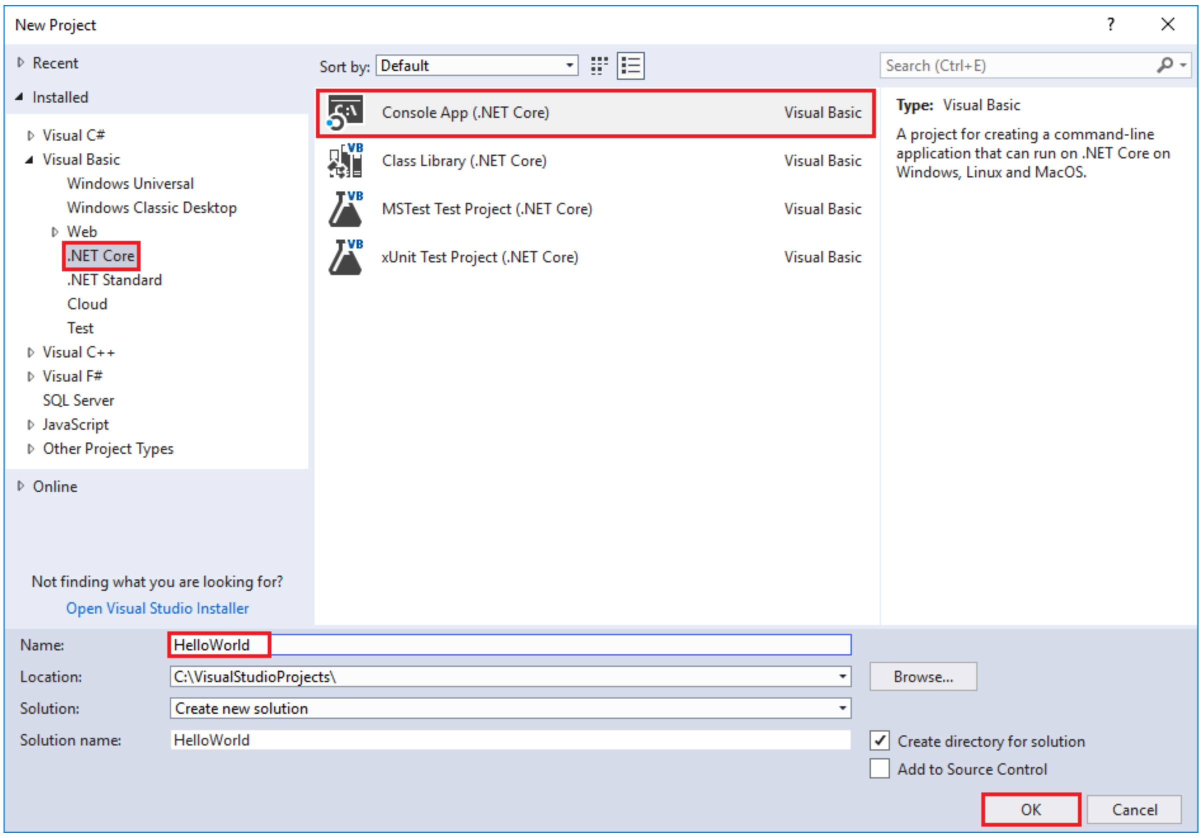
Visual Studio 2017 with the ".NET Core cross-platform development" workload installed. You can develop your app with .NET Core 2.1 or later versions.

For more information, see Prerequisites for .NET Core on Windows.

A simple Hello World application

Begin by creating a simple "Hello World" console application. Follow these steps:

1. Launch Visual Studio 2017. Select **File** > **New** > **Project** from the menu bar. In the *New Project** dialog, select the **Visual Basic** node followed by the **.NET Core** node. Then select the **Console App (.NET Core)** project template. In the **Name** text box, type "HelloWorld". Select the **OK** button.



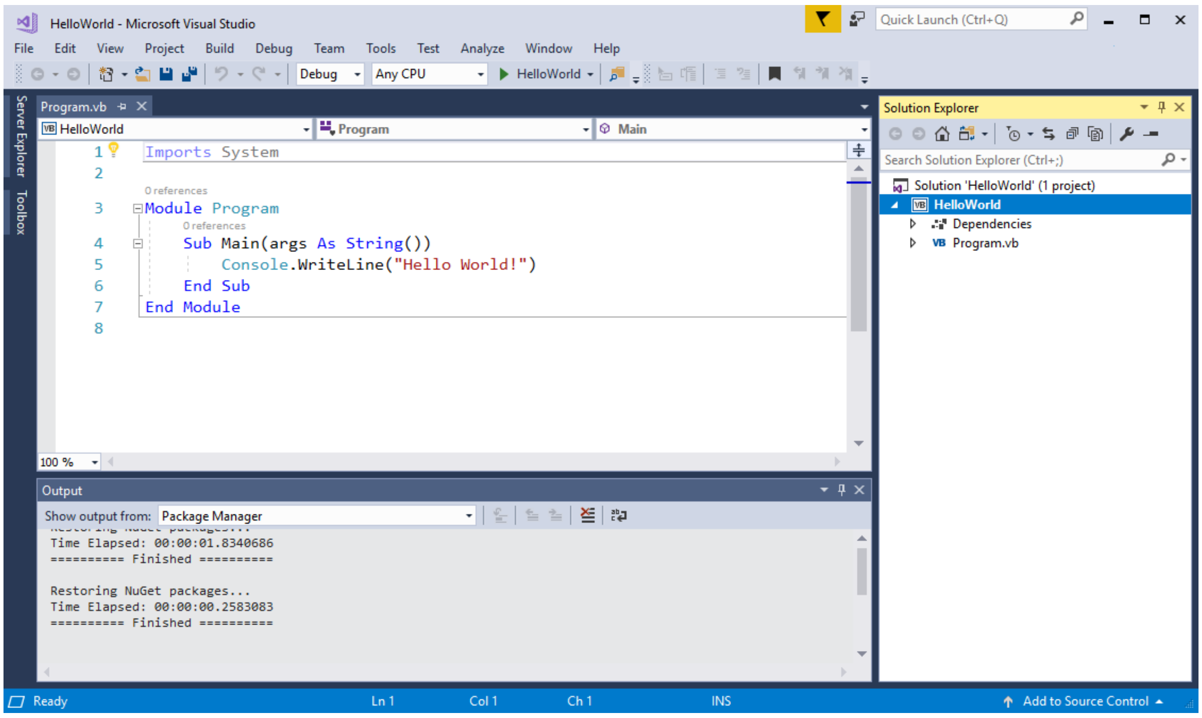
Type: Visual Basic

A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.

Name: HelloWorld
Location: C:\VisualStudioProjects\
Solution: Create new solution
Solution name: HelloWorld
[✓] Create directory for solution
[ ] Add to Source Control

2. Visual Studio uses the template to create your project. The Visual Basic Console Application template for .NET Core automatically defines a class, `Program`, with a single method, `Main`, that takes a String array as an argument. `Main` is the application entry point, the method that's called automatically by the runtime

when it launches the application. Any command-line arguments supplied when the application is launched are available in the *args* array.


HelloWorld - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Debug
Any CPU
HelloWorld
Program.vb
HelloWorld
Program
Main
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'HelloWorld' (1 project)
HelloWorld
Dependencies
Program.vb
100 %
Output
Show output from: Package Manager
Ready
Ln 1
Col 1
Ch 1
INS
Add to Source Control

```
Imports System

Module Program
    Sub Main(args As String())
        Console.WriteLine("Hello World!")
    End Sub
End Module
```

```
Time Elapsed: 00:00:01.8340686
========== Finished ==========

Restoring NuGet packages...
Time Elapsed: 00:00:00.2583083
========== Finished ==========
```

The template creates a simple "Hello World" application. It calls the Console.WriteLine(String) method to display the literal string "Hello World!" in the console window. By selecting the **HelloWorld** button with the green arrow on the toolbar, you can run the program in Debug mode. If you do, the console window is visible for only a brief time interval before it closes. This occurs because the `Main` method terminates and the application ends as soon as the single statement in the `Main` method executes.

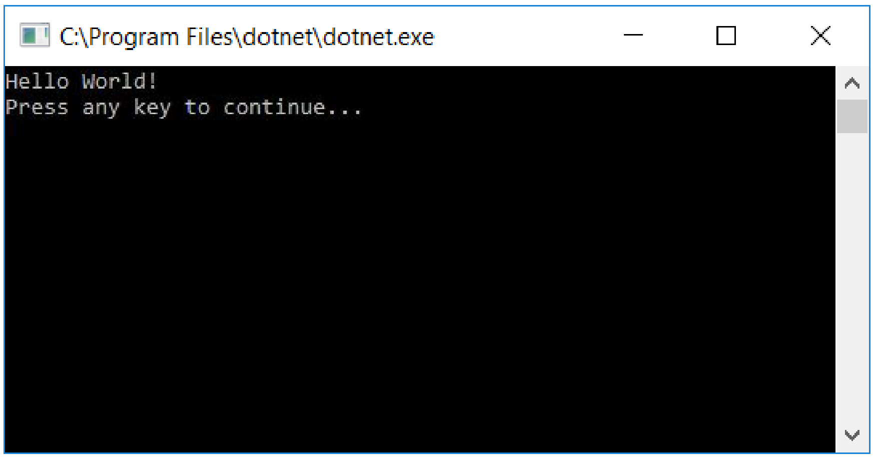
3. To cause the application to pause before it closes the console window, add the following code immediately after the call to the Console.WriteLine(String) method:

```
Console.Write("Press any key to continue...")
Console.ReadKey(true)
```

This code prompts the user to press any key and then pauses the program until a key is pressed.

4. On the menu bar, select **Build** > **Build Solution**. This compiles your program into an intermediate language (IL) that's converted into binary code by a just-in-time (JIT) compiler.

5. Run the program by selecting the **HelloWorld** button with the green arrow on the toolbar.



```
Hello World!
Press any key to continue...
```

6. Press any key to close the console window.

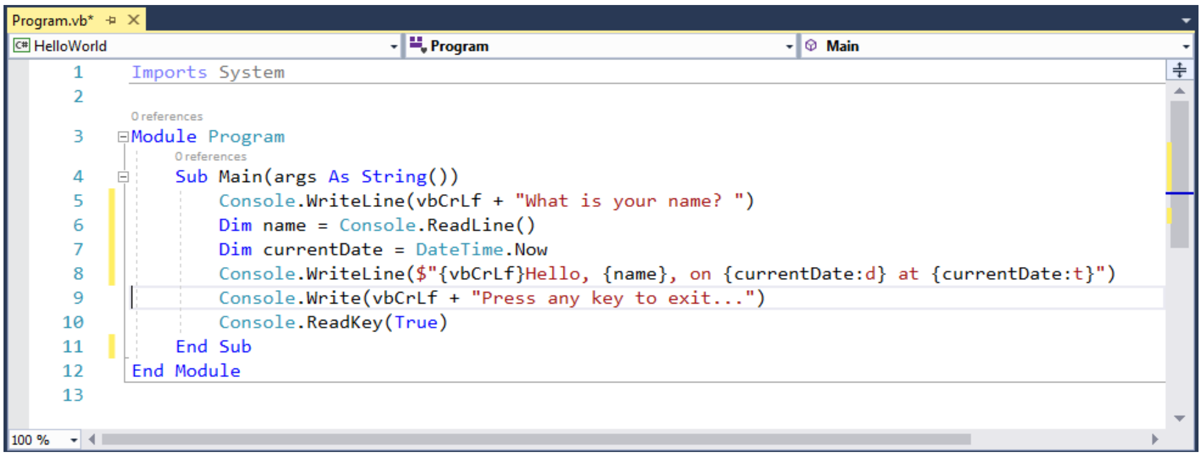
Enhancing the Hello World application

Enhance your application to prompt the user for his or her name and to display it along with the date and time. To modify and test the program, do the following:

1. Enter the following Visual Basic code in the code window immediately after the opening bracket that follows the `Sub Main(args As String())` line and before the first closing bracket:

   ```
   Console.WriteLine(vbCrLf + "What is your name? ")
   Dim name = Console.ReadLine()
   Dim currentDate = DateTime.Now
   Console.WriteLine($"{vbCrLf}Hello, {name}, on {currentDate:d} at {currentDate:t}")
   Console.Write(vbCrLf + "Press any key to exit... ")
   Console.ReadKey(True)
   ```

   This code replaces the contents of the `Main` method.

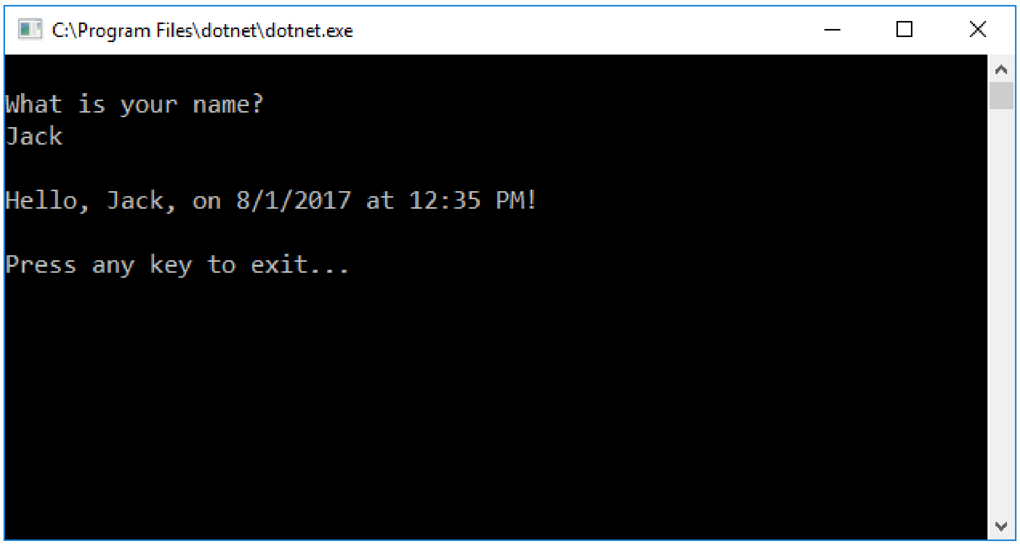
   

   ```
   Imports System

   0 references
   Module Program
       0 references
       Sub Main(args As String())
           Console.WriteLine(vbCrLf + "What is your name? ")
           Dim name = Console.ReadLine()
           Dim currentDate = DateTime.Now
           Console.WriteLine($"{vbCrLf}Hello, {name}, on {currentDate:d} at {currentDate:t}")
           Console.Write(vbCrLf + "Press any key to exit...")
           Console.ReadKey(True)
       End Sub
   End Module
   ```

   This code displays "What is your name?" in the console window and waits until the user enters a string followed by the Enter key. It stores this string into a variable named `name`. It also retrieves the value of the DateTime.Now property, which contains the current local time, and assigns it to a variable named `currentDate`. Finally, it uses an interpolated string to display these values in the console window.

2. Compile the program by choosing **Build** > **Build Solution**.

3. Run the program in Debug mode in Visual Studio by selecting the green arrow on the toolbar, pressing F5, or choosing the **Debug** > **Start Debugging** menu item. Respond to the prompt by entering a name and pressing the Enter key.

   ```

   What is your name?
   Jack

   Hello, Jack, on 8/1/2017 at 12:35 PM!

   Press any key to exit...
   ```

4. Press any key to close the console window.

You've created and run your application. To develop a professional application, take some additional steps to make your application ready for release:

- To debug your application, see Debug your .NET Core Hello World application using Visual Studio 2017.
- To publish a distributable version of your application, see Publish your .NET Core Hello World application with Visual Studio 2017.

Related topics

Instead of a console application, you can also build a .NET Standard class library with Visual Basic, .NET Core, and Visual Studio 2017. For a step-by-step introduction, see Build a .NET Standard library with Visual Basic and .NET Core SDK in Visual Studio 2017.

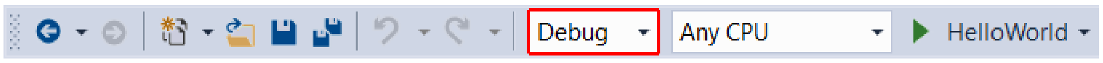
Debug your C# or Visual Basic .NET Core Hello World application using Visual Studio 2017

9/13/2019 • 11 minutes to read • Edit Online

So far, you've followed the steps in Build a C# Hello World Application with .NET Core in Visual Studio 2017 or Build a Visual Basic Hello World Application with .NET Core in Visual Studio 2017 to create and run a simple console application. Once you've written and compiled your application, you can begin testing it. Visual Studio includes a comprehensive set of debugging tools that you can use when testing and troubleshooting your application.

Debugging in Debug mode

Debug and *Release* are two of Visual Studio's default build configurations. The current build configuration is shown on the toolbar. The following toolbar image shows that Visual Studio is configured to compile your application in **Debug** mode.



You should always begin by testing your program in Debug mode. Debug mode turns off most compiler optimizations and provides richer information during the build process.

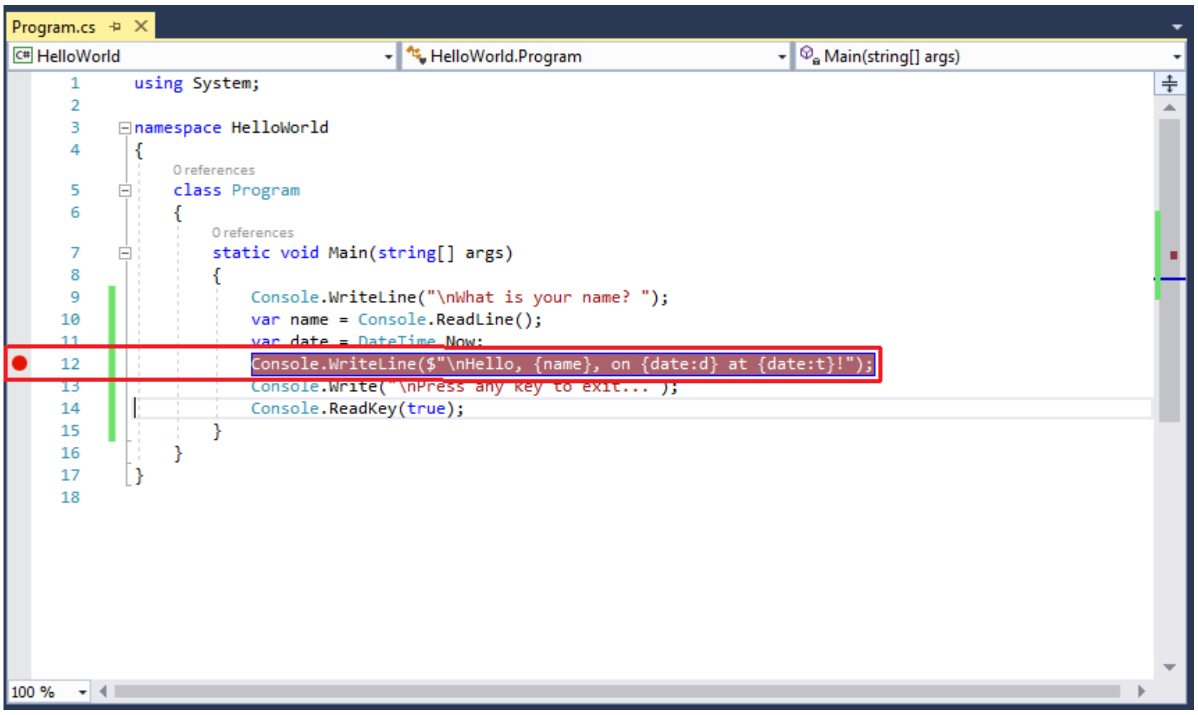
Setting a breakpoint

Run your program in Debug mode and try a few debugging features:

- C#
- Visual Basic

1. A *breakpoint* temporarily interrupts the execution of the application *before* the line with the breakpoint is executed.

   Set a breakpoint on the line that reads `Console.WriteLine($"\nHello, {name}, on {date:d} at {date:t}!");` by clicking in the left margin of the code window on that line or by choosing the **Debug** > **Toggle Breakpoint** menu item with the line selected. As the following figure shows, Visual Studio indicates the line on which the breakpoint is set by highlighting it and displaying a red circle in its left margin.



Program.cs

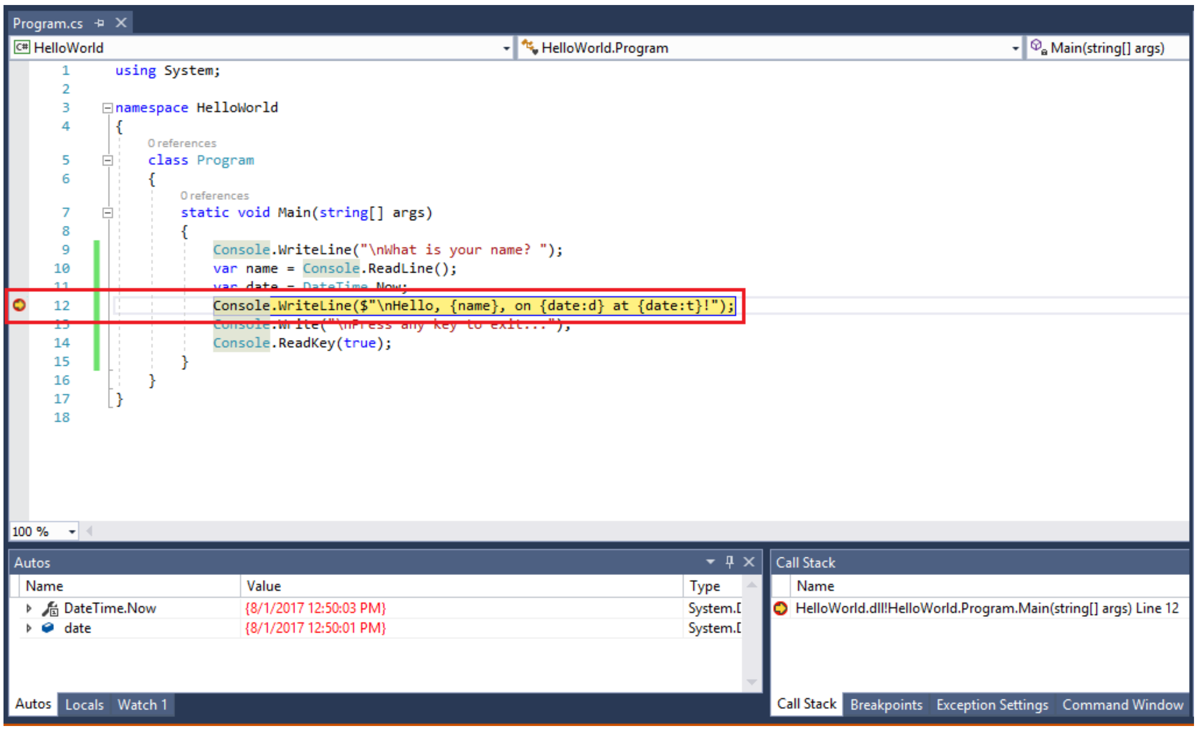
HelloWorld | HelloWorld.Program | Main(string[] args)

```
using System;

namespace HelloWorld
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("\nWhat is your name? ");
            var name = Console.ReadLine();
            var date = DateTime.Now;
            Console.WriteLine($"\nHello, {name}, on {date:d} at {date:t}!");
            Console.Write("\nPress any key to exit...");
            Console.ReadKey(true);
        }
    }
}
```

2. Run the program in Debug mode by selecting the **HelloWorld** button with the green arrow on the toolbar, pressing F5, or choosing **Debug** > **Start Debugging**.

3. Enter a string in the console window when the program prompts for a name and press Enter.

4. Program execution stops when it reaches the breakpoint and before the `Console.WriteLine` method executes. The **Autos** window displays the values of variables that are used around the current line. The **Locals** window (which you can view by clicking the **Locals** tab) displays the values of variables that are defined in the currently executing method.



Program.cs

HelloWorld | HelloWorld.Program | Main(string[] args)

```
using System;

namespace HelloWorld
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("\nWhat is your name? ");
            var name = Console.ReadLine();
            var date = DateTime.Now;
            Console.WriteLine($"\nHello, {name}, on {date:d} at {date:t}!");
            Console.Write("\nPress any key to exit...");
            Console.ReadKey(true);
        }
    }
}
```

Autos

Name	Value	Type
DateTime.Now	{8/1/2017 12:50:03 PM}	System.[
date	{8/1/2017 12:50:01 PM}	System.[

Call Stack

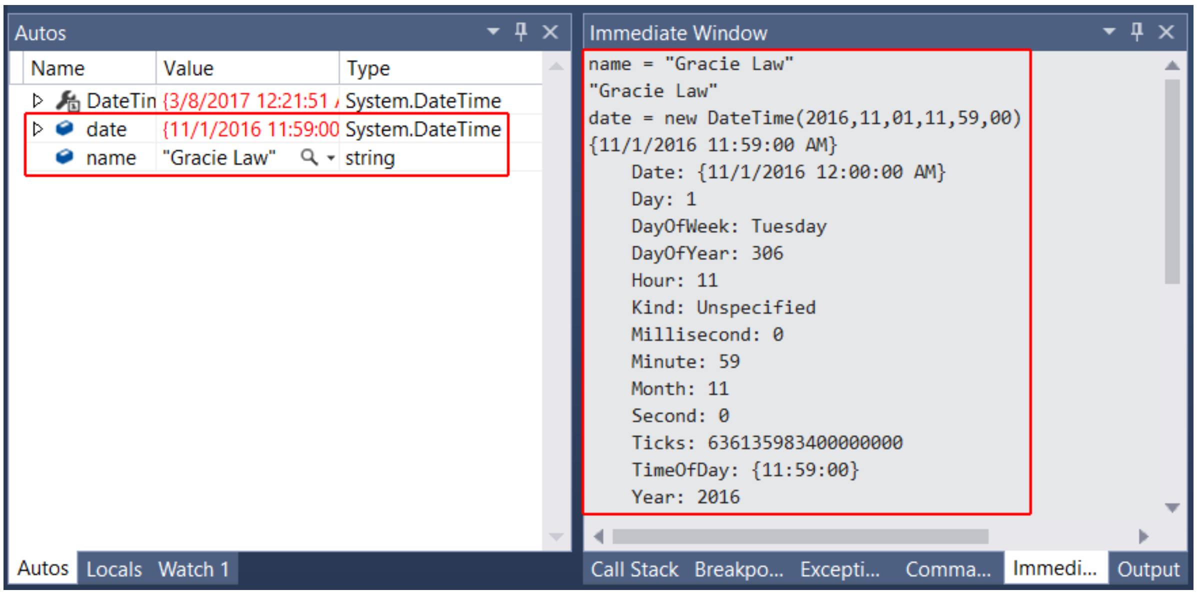
Name
HelloWorld.dll!HelloWorld.Program.Main(string[] args) Line 12

5. You can change the value of the variables to see how it affects your program. If the **Immediate Window** is not visible, display it by choosing the **Debug** > **Windows** > **Immediate** menu item. The **Immediate Window** lets you interact with the application you're debugging.

6. You can interactively change the values of variables. Enter `name = "Gracie"` in the **Immediate Window** and press the Enter key.

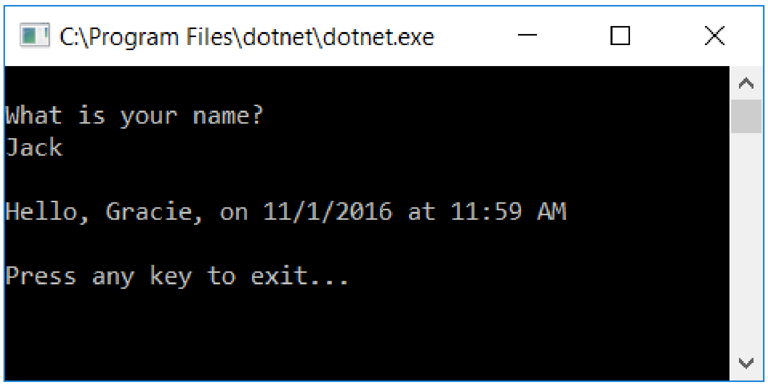
7. Enter `date = new DateTime(2016,11,01,11,59,00)` in the **Immediate Window** and press the Enter key.

   The **Immediate Window** displays the value of the string variable and the properties of the DateTime value. In addition, the value of the variables is updated in the **Autos** and **Locals** windows.



```
name = "Gracie Law"
"Gracie Law"
date = new DateTime(2016,11,01,11,59,00)
{11/1/2016 11:59:00 AM}
    Date: {11/1/2016 12:00:00 AM}
    Day: 1
    DayOfWeek: Tuesday
    DayOfYear: 306
    Hour: 11
    Kind: Unspecified
    Millisecond: 0
    Minute: 59
    Month: 11
    Second: 0
    Ticks: 636135983400000000
    TimeOfDay: {11:59:00}
    Year: 2016
```

8. Continue program execution by selecting the **Continue** button in the toolbar or by selecting the **Debug** > **Continue** menu item. The values displayed in the console window correspond to the changes you made in the **Immediate Window**.



```
What is your name?
Jack

Hello, Gracie, on 11/1/2016 at 11:59 AM

Press any key to exit...
```

9. Press any key to exit the application and end Debug mode.

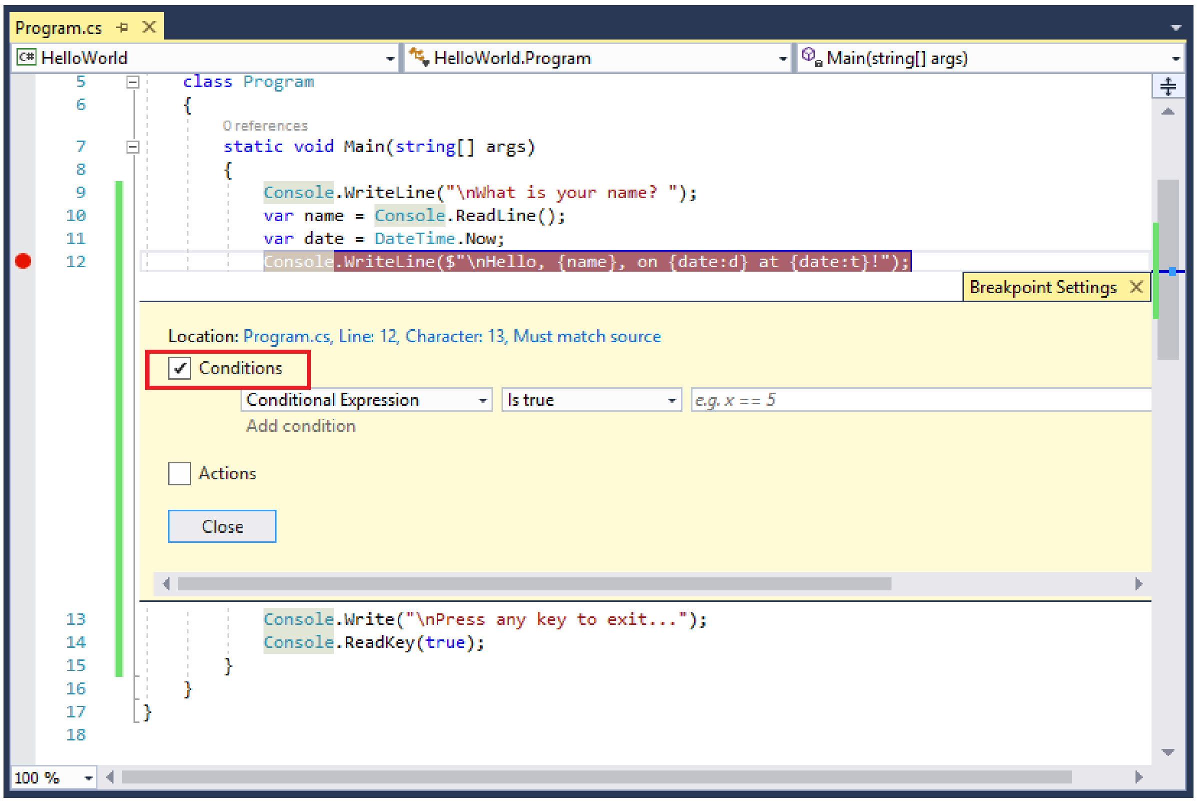
Setting a conditional breakpoint

Your program displays the string that the user enters. What happens if the user doesn't enter anything? You can test this with a useful debugging feature, the *conditional breakpoint*, which breaks program execution when one or more conditions are met.

To set a conditional breakpoint and test what happens when the user fails to enter a string, do the following:

- C#
- Visual Basic

1. Right-click on the red dot that represents the breakpoint. On the context menu, select **Conditions** to open the **Breakpoint Settings** dialog. Check the box for **Conditions**.



```
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("\nWhat is your name? ");
            var name = Console.ReadLine();
            var date = DateTime.Now;
            Console.WriteLine($"\nHello, {name}, on {date:d} at {date:t}!");
```

Location: Program.cs, Line: 12, Character: 13, Must match source

[✓] Conditions

```
            Console.Write("\nPress any key to exit...");
            Console.ReadKey(true);
        }
    }
}
```

2. For the **Conditional Expression** replace "e.g. x == 5" with the following:

```
String.IsNullOrEmpty(name)
```

You're testing for a code condition, that the `String.IsNullOrEmpty(name)` method call is `true` either because *name* has not been assigned a value or because its value is an empty string (""). You can also specify a *hit count*, which interrupts program execution before a statement is executed a specified number of times, or a *filter condition*, which interrupts program execution based on such attributes as a thread identifier, process name, or thread name.

3. Select the **Close** button to close the dialog.

4. Run the program in Debug mode.

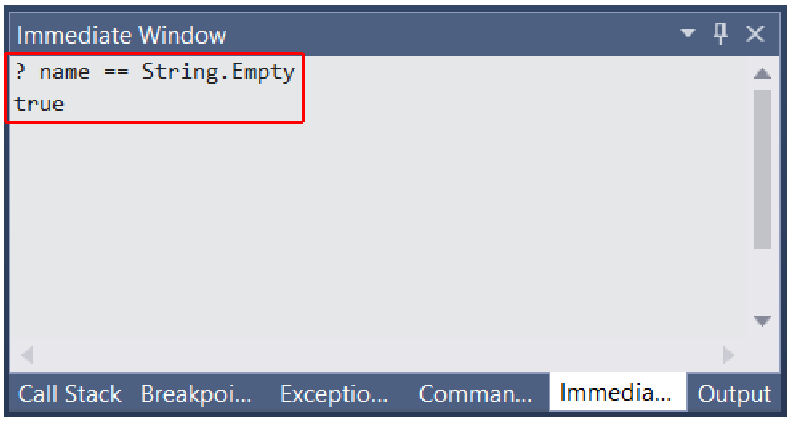
5. In the console window, press the Enter key when prompted to enter your name.

6. Because the condition we specified, `name` is either `null` or String.Empty, has been satisfied, program execution stops when it reaches the breakpoint and before the `Console.WriteLine` method executes.

7. Select the **Locals** window, which shows the values of variables that are local to the currently executing method, which is the `Main` method in your program. Observe that the value of the `name` variable is `""`, or String.Empty.

8. Confirm the value is an empty string by entering the following statement in the **Immediate Window**. The result is `true`.

```
? name == String.Empty
```



```
? name == String.Empty
true
```

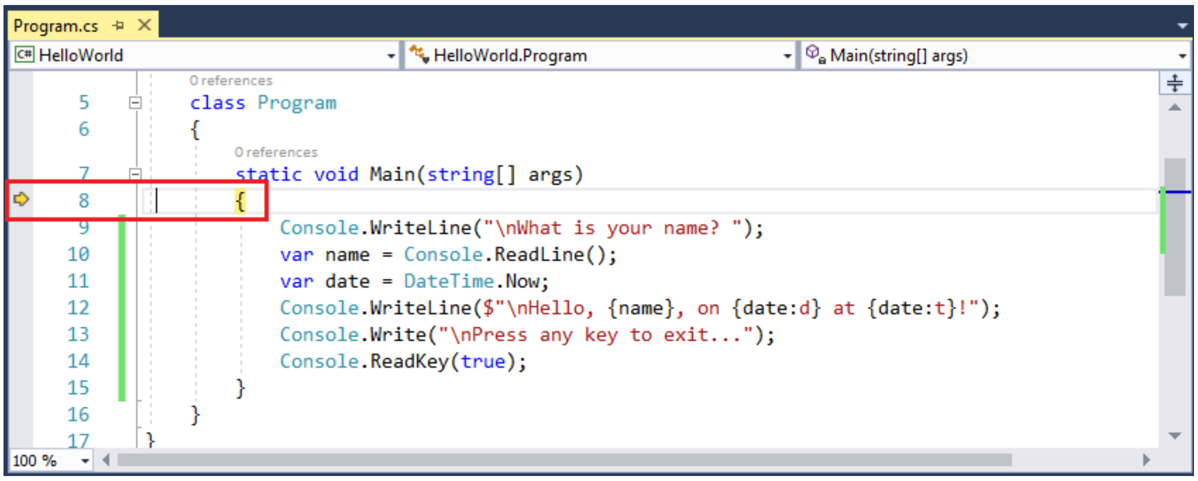
9. Select the **Continue** button on the toolbar to continue program execution.
10. Press any key to close the console window and exit Debug mode.
11. Clear the breakpoint by clicking on the dot in the left margin of the code window or by choosing the **Debug** > **Toggle Breakpoint** menu item with the row selected.

Stepping through a program

Visual Studio also allows you to step line by line through a program and monitor its execution. Ordinarily, you'd set a breakpoint and use this feature to follow program flow through a small part of your program code. Since your program is small, you can step through the entire program by doing the following:

- C#
- Visual Basic

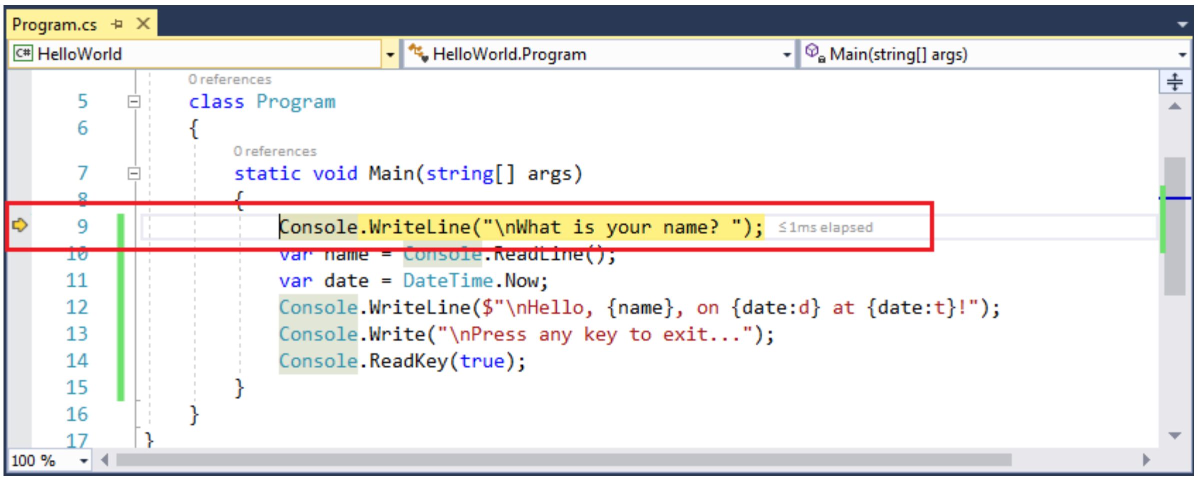
1. On the menu bar, choose **Debug** > **Step Into** or press the F11 key. Visual Studio highlights and displays an arrow beside the next line of execution.



```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Console.WriteLine("\nWhat is your name? ");
        var name = Console.ReadLine();
        var date = DateTime.Now;
        Console.WriteLine($"\nHello, {name}, on {date:d} at {date:t}!");
        Console.Write("\nPress any key to exit...");
        Console.ReadKey(true);
    }
}
}
```

At this point, the **Autos** window shows that your program has defined only one variable, `args`. Because you haven't passed any command-line arguments to the program, its value is an empty string array. In addition, Visual Studio has opened a blank console window.

2. Select **Debug** > **Step Into** or press the F11 key. Visual Studio now highlights the next line of execution. As the figure shows, it has taken less than one millisecond to execute the code between the last statement and this one. `args` remains the only declared variable, and the console window remains blank.



```
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("\nWhat is your name? "); ≤1ms elapsed
            var name = Console.ReadLine();
            var date = DateTime.Now;
            Console.WriteLine($"\nHello, {name}, on {date:d} at {date:t}!");
            Console.Write("\nPress any key to exit...");
            Console.ReadKey(true);
        }
    }
}
```

3. Select **Debug** > **Step Into** or press the F11 key. Visual Studio highlights the statement that includes the `name` variable assignment. The **Autos** window shows that `name` is `null`, and the console window displays the string "What is your name?".

4. Respond to the prompt by entering a string in the console window and pressing Enter. The console is unresponsive, and the string you enter isn't displayed in the console window, but the Console.ReadLine method will nevertheless capture your input.

5. Select **Debug** > **Step Into** or press the F11 key. Visual Studio highlights the statement that includes the `date` (in C#) or `currentDate` (in Visual Basic) variable assignment. The **Autos** window shows the DateTime.Now property value and the value returned by the call to the Console.ReadLine method. The console window also displays the string entered when the console prompted for input.

6. Select **Debug** > **Step Into** or press the F11 key. The **Autos** window shows the value of the `date` variable after the assignment from the DateTime.Now property. The console window is unchanged.

7. Select **Debug** > **Step Into** or press the F11 key. Visual Studio calls the Console.WriteLine(String, Object, Object) method. The values of the `date` (or `currentDate`) and `name` variables appear in the **Autos** window, and the console window displays the formatted string.

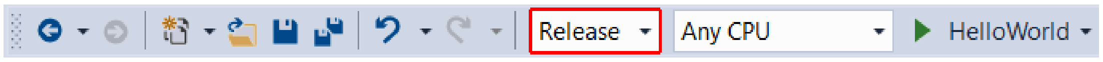
8. Select **Debug** > **Step Out** or press Shift and the F11 key. This stops step-by-step execution. The console window displays a message and waits for you to press a key.

9. Press any key to close the console window and exit Debug mode.

Building a Release version

Once you've tested the Debug build of your application, you should also compile and test the Release version. The Release version incorporates compiler optimizations that can sometimes negatively affect the behavior of an application. For example, compiler optimizations that are designed to improve performance can create race conditions in asynchronous or multithreaded applications.

To build and test the Release version of your console application, change the build configuration on the toolbar from **Debug** to **Release**.



When you press F5 or choose **Build Solution** from the **Build** menu, Visual Studio compiles the Release version of your console application. You can test it as you did the Debug version of the application.

Once you've finished debugging your application, the next step is to publish a deployable version of your application. For information on how to do this, see Publish the Hello World application with Visual Studio 2017.

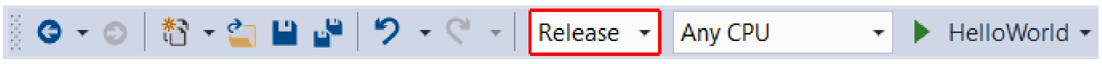
Publish your .NET Core Hello World application with Visual Studio 2017

8/20/2019 • 2 minutes to read • Edit Online

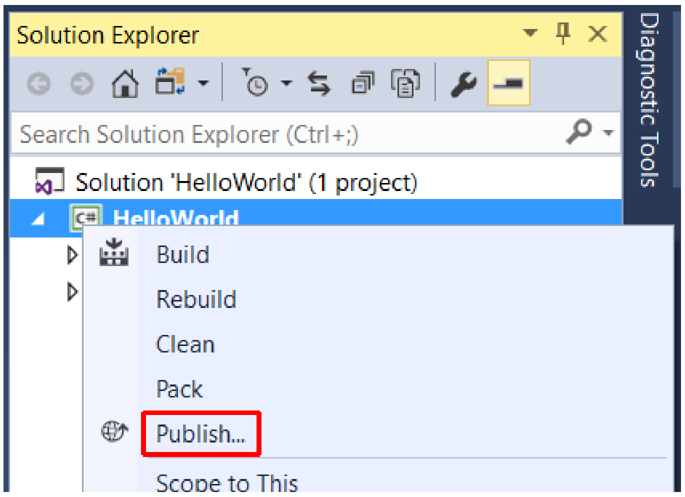
In Build a C# Hello World application with .NET Core in Visual Studio 2017 or Build a Visual Basic Hello World application with .NET Core in Visual Studio 2017, you built a Hello World console application. In Debug your C# Hello World application with Visual Studio 2017, you tested it using the Visual Studio debugger. Now that you're sure that it works as expected, you can publish it so that other users can run it. Publishing creates the set of files that are needed to run your application, and you can deploy the files by copying them to a target machine.

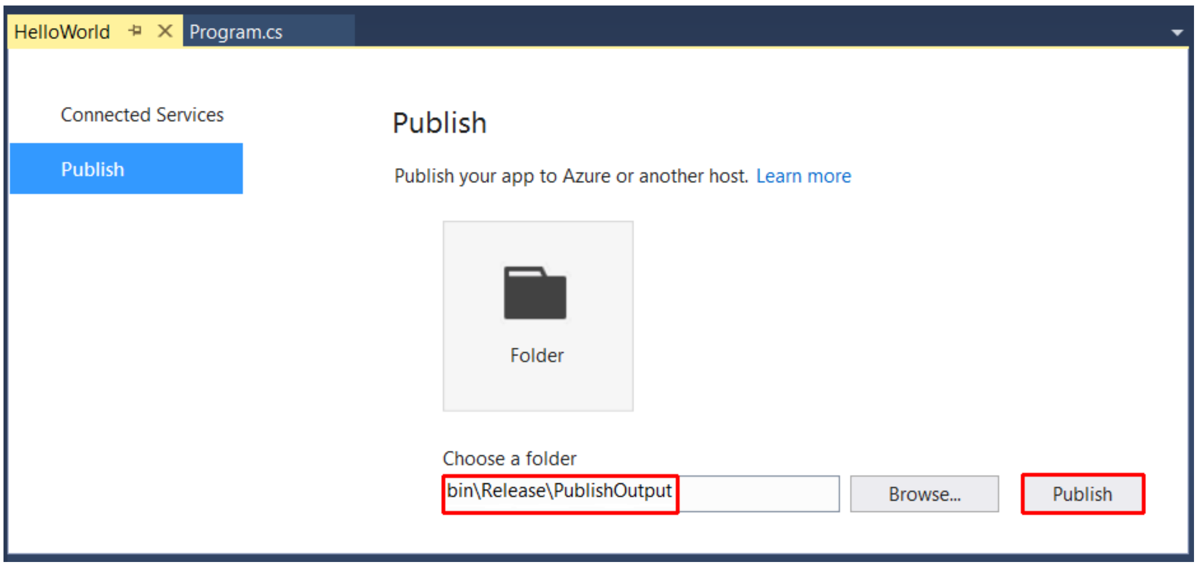
To publish and run your application:

1. Make sure that Visual Studio is building the Release version of your application. If necessary, change the build configuration setting on the toolbar from **Debug** to **Release**.



2. Right-click on the **HelloWorld** project (not the HelloWorld solution) and select **Publish** from the menu. You can also select **Publish HelloWorld** from the main Visual Studio **Build** menu.





3. Open a console window. For example in the **Type here to search** text box in the Windows taskbar, enter

`Command Prompt` (or `cmd` for short), and open a console window by either selecting the **Command Prompt** desktop app or pressing Enter if it's selected in the search results.

4. Navigate to the published application in the `bin\release\PublishOutput` subdirectory of your application's project directory. As the following figure shows, the published output includes the following four files:

   - *HelloWorld.deps.json*

     The application's runtime dependencies file. It defines the .NET Core components and the libraries (including the dynamic link library that contains your application) needed to run your application. For more information, see Runtime Configuration Files.

   - *HelloWorld.dll*

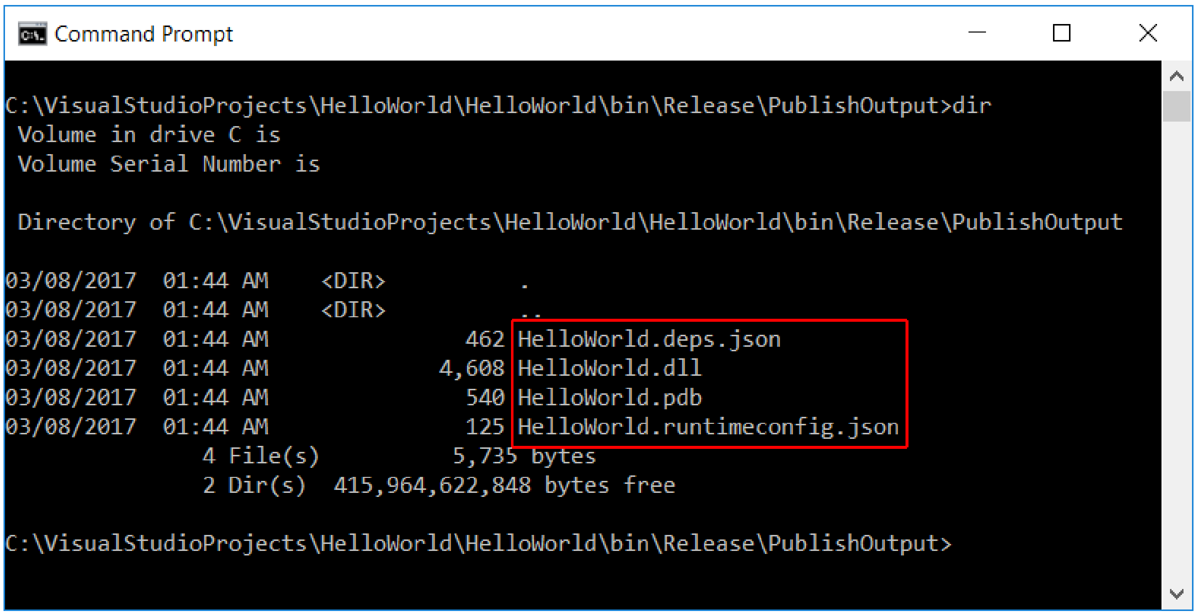
     The file that contains your application. It is a dynamic link library that can be executed by entering the `dotnet HelloWorld.dll` command in a console window.

   - *HelloWorld.pdb* (optional for deployment)

     A file that contains debug symbols. You aren't required to deploy this file along with your application, although you should save it in the event that you need to debug the published version of your application.

   - *HelloWorld.runtimeconfig.json*

     The application's runtime configuration file. It identifies the version of .NET Core that your application was built to run on. For more information, see Runtime Configuration Files.



```
C:\VisualStudioProjects\HelloWorld\HelloWorld\bin\Release\PublishOutput>dir
 Volume in drive C is
 Volume Serial Number is

 Directory of C:\VisualStudioProjects\HelloWorld\HelloWorld\bin\Release\PublishOutput

03/08/2017  01:44 AM    <DIR>          .
03/08/2017  01:44 AM    <DIR>          ..
03/08/2017  01:44 AM               462 HelloWorld.deps.json
03/08/2017  01:44 AM             4,608 HelloWorld.dll
03/08/2017  01:44 AM               540 HelloWorld.pdb
03/08/2017  01:44 AM               125 HelloWorld.runtimeconfig.json
               4 File(s)          5,735 bytes
               2 Dir(s)  415,964,622,848 bytes free

C:\VisualStudioProjects\HelloWorld\HelloWorld\bin\Release\PublishOutput>
```

The publishing process creates a framework-dependent deployment, which is a type of deployment where the published application will run on any platform supported by .NET Core with .NET Core installed on the system. Users can run your application by issuing the `dotnet HelloWorld.dll` command from a console window.

For more information on publishing and deploying .NET Core applications, see .NET Core Application Deployment.

Build a .NET Standard library with C# and the .NET Core SDK in Visual Studio 2017

5/15/2019 • 2 minutes to read • Edit Online

A *class library* defines types and methods that are called by an application. A class library that targets the .NET Standard 2.0 allows your library to be called by any .NET implementation that supports that version of the .NET Standard. When you finish your class library, you can decide whether you want to distribute it as a third-party component or whether you want to include it as a bundled component with one or more applications.

NOTE

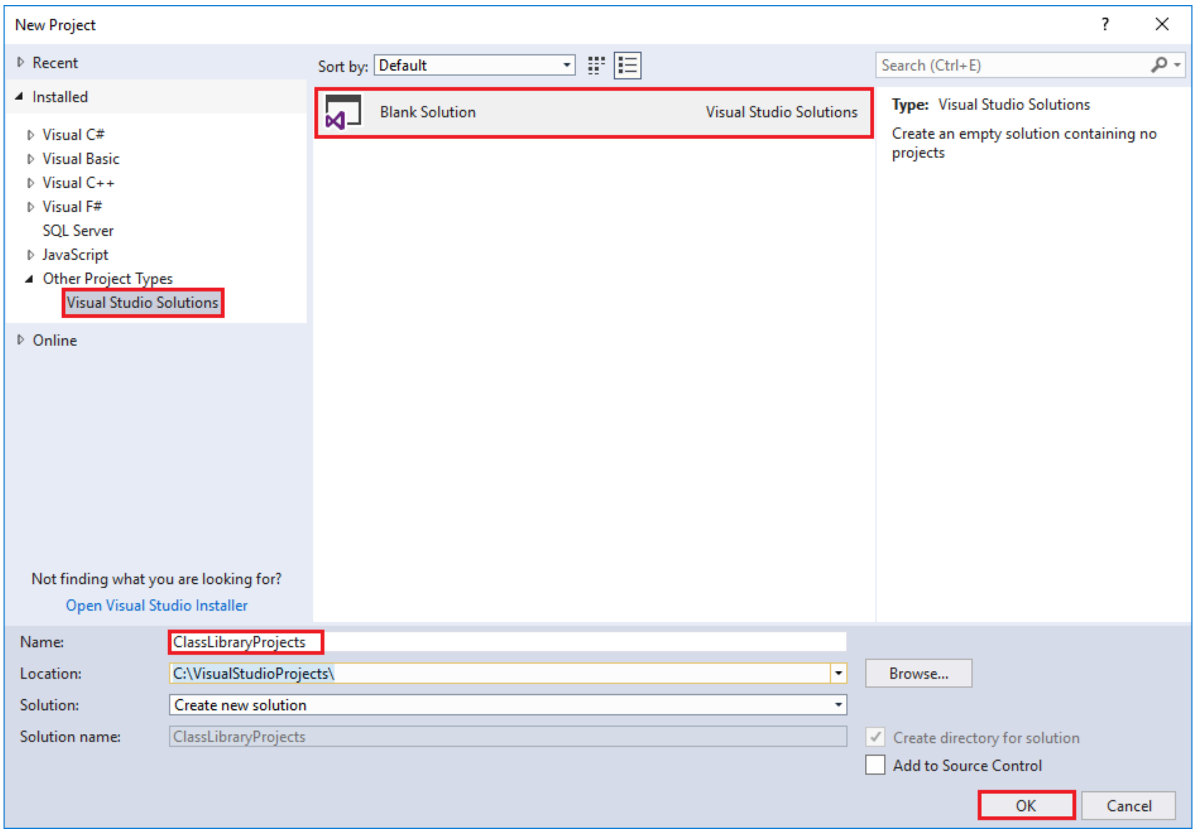
For a list of the .NET Standard versions and the platforms they support, see .NET Standard.

In this topic, you'll create a simple utility library that contains a single string-handling method. You'll implement it as an extension method so that you can call it as if it were a member of the String class.

Creating a class library solution

Start by creating a solution for your class library project and its related projects. A Visual Studio Solution just serves as a container for one or more projects. To create the solution:

1. On the Visual Studio menu bar, choose **File** > **New** > **Project**.
2. In the **New Project** dialog, expand the **Other Project Types** node, and select **Visual Studio Solutions**. Name the solution "ClassLibraryProjects" and select the **OK** button.



Type: Visual Studio Solutions

Create an empty solution containing no projects

Name: ClassLibraryProjects

Location: C:\VisualStudioProjects\

Solution: Create new solution

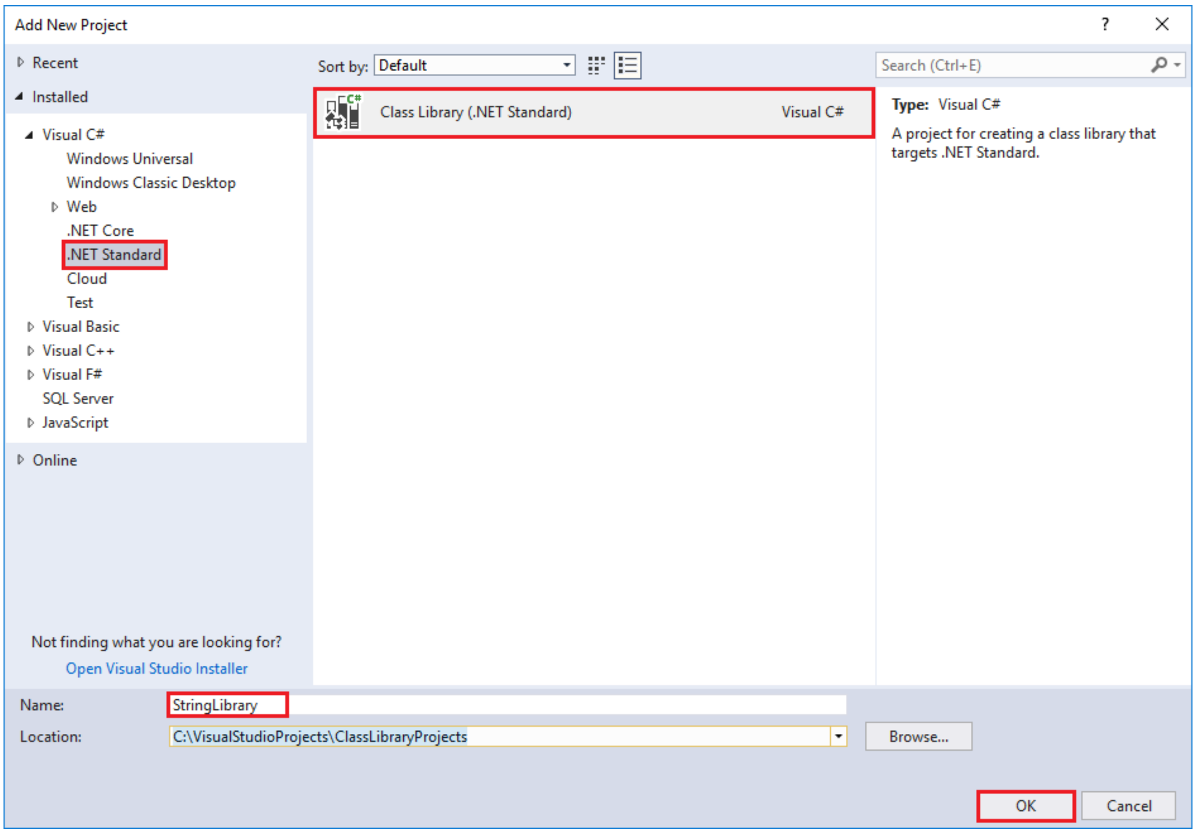
Solution name: ClassLibraryProjects

Creating the class library project

Create your class library project:

1. In **Solution Explorer**, right-click on the **ClassLibraryProjects** solution file and from the context menu, select **Add** > **New Project**.

2. In the **Add New Project** dialog, expand the **Visual C#** node, then select the **.NET Standard** node followed by the **Class Library (.NET Standard)** project template. In the **Name** text box, enter "StringLibrary" as the name of the project. Select **OK** to create the class library project.



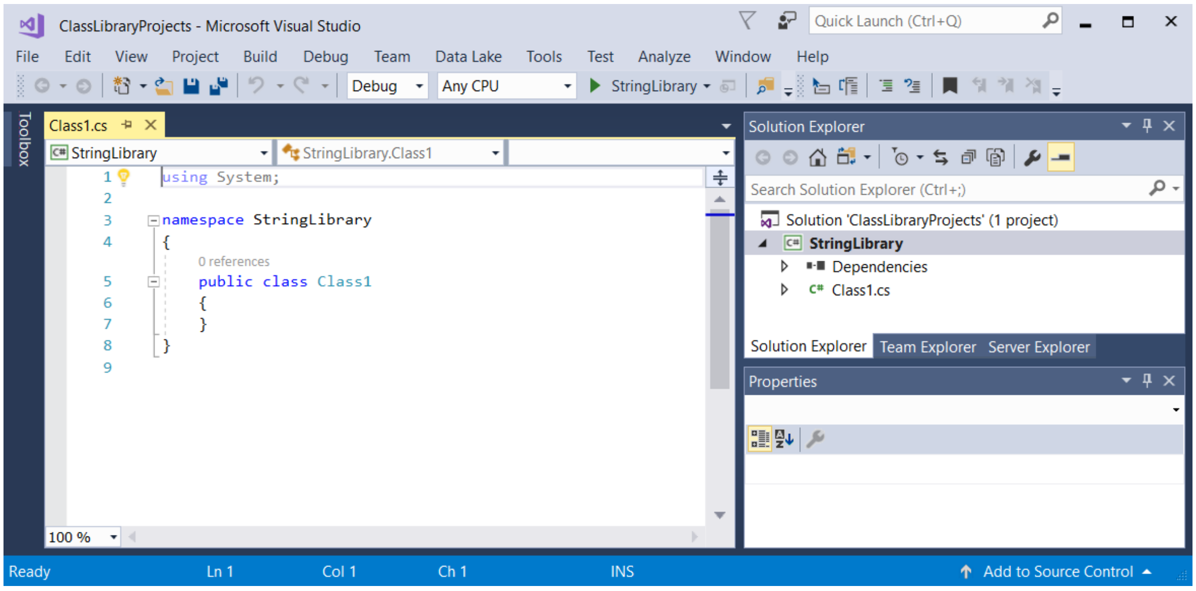
Type: Visual C#

A project for creating a class library that targets .NET Standard.

Name: StringLibrary

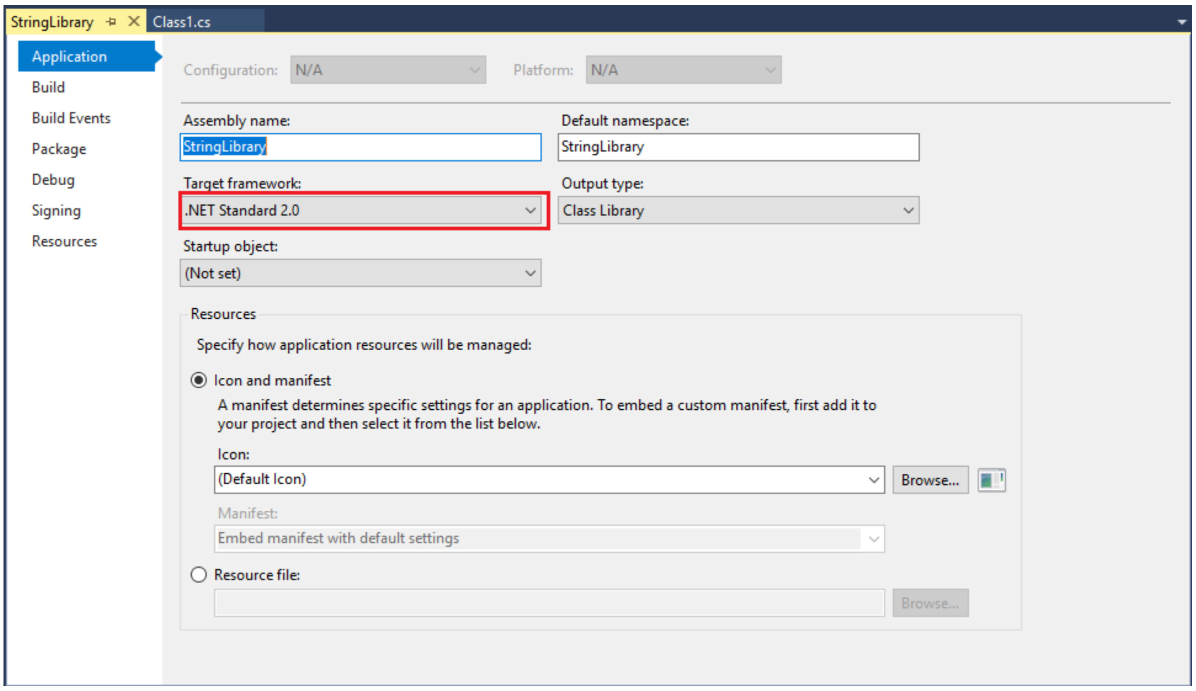
Location: C:\VisualStudioProjects\ClassLibraryProjects

The code window then opens in the Visual Studio development environment.


ClassLibraryProjects - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Project Build Debug Team Data Lake Tools Test Analyze Window Help
Debug
Any CPU
StringLibrary
Toolbox
Class1.cs
StringLibrary
StringLibrary.Class1
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'ClassLibraryProjects' (1 project)
StringLibrary
Dependencies
Class1.cs
Solution Explorer Team Explorer Server Explorer
Properties
100 %
Ready Ln 1 Col 1 Ch 1 INS
Add to Source Control

```
using System;

namespace StringLibrary
{
    0 references
    public class Class1
    {
    }
}
```

3. Check to make sure that our library targets the correct version of the .NET Standard. Right-click on the library project in the **Solution Explorer** windows, then select **Properties**. The **Target Framework** text box shows that we're targeting .NET Standard 2.0.



Configuration: N/A
Platform: N/A

Assembly name: StringLibrary
Default namespace: StringLibrary
Target framework: .NET Standard 2.0
Output type: Class Library
Startup object: (Not set)

Resources

Specify how application resources will be managed:

(●) Icon and manifest

A manifest determines specific settings for an application. To embed a custom manifest, first add it to your project and then select it from the list below.

Icon: (Default Icon)

Manifest: Embed manifest with default settings

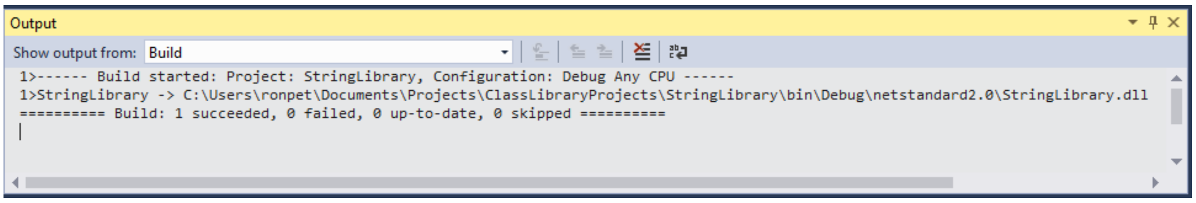
( ) Resource file:

4. Replace the code in the code window with the following code and save the file:

```
using System;

namespace UtilityLibraries
{
    public static class StringLibrary
    {
        public static bool StartsWithUpper(this String str)
        {
            if (String.IsNullOrWhiteSpace(str))
                return false;

            Char ch = str[0];
            return Char.IsUpper(ch);
        }
    }
}
```

The class library, `UtilityLibraries.StringLibrary`, contains a method named `StartsWithUpper`, which returns a Boolean value that indicates whether the current string instance begins with an uppercase character. The Unicode standard distinguishes uppercase characters from lowercase characters. The Char.IsUpper(Char) method returns `true` if a character is uppercase.

5. On the menu bar, select **Build** > **Build Solution**. The project should compile without error.



```
1>------ Build started: Project: StringLibrary, Configuration: Debug Any CPU ------
1>StringLibrary -> C:\Users\ronpet\Documents\Projects\ClassLibraryProjects\StringLibrary\bin\Debug\netstandard2.0\StringLibrary.dll
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Next step

You've successfully built the library. Because you haven't called any of its methods, you don't know whether it works as expected. The next step in developing your library is to test it by using a Unit Test Project.

Build a .NET Standard library with Visual Basic and the .NET Core SDK in Visual Studio 2017

10/30/2019 • 2 minutes to read • Edit Online

A *class library* defines types and methods that are called by an application. A class library that targets the .NET Standard 2.0 allows your library to be called by any .NET implementation that supports that version of the .NET Standard. When you finish your class library, you can decide whether you want to distribute it as a third-party component or whether you want to include it as a bundled component with one or more applications.

NOTE

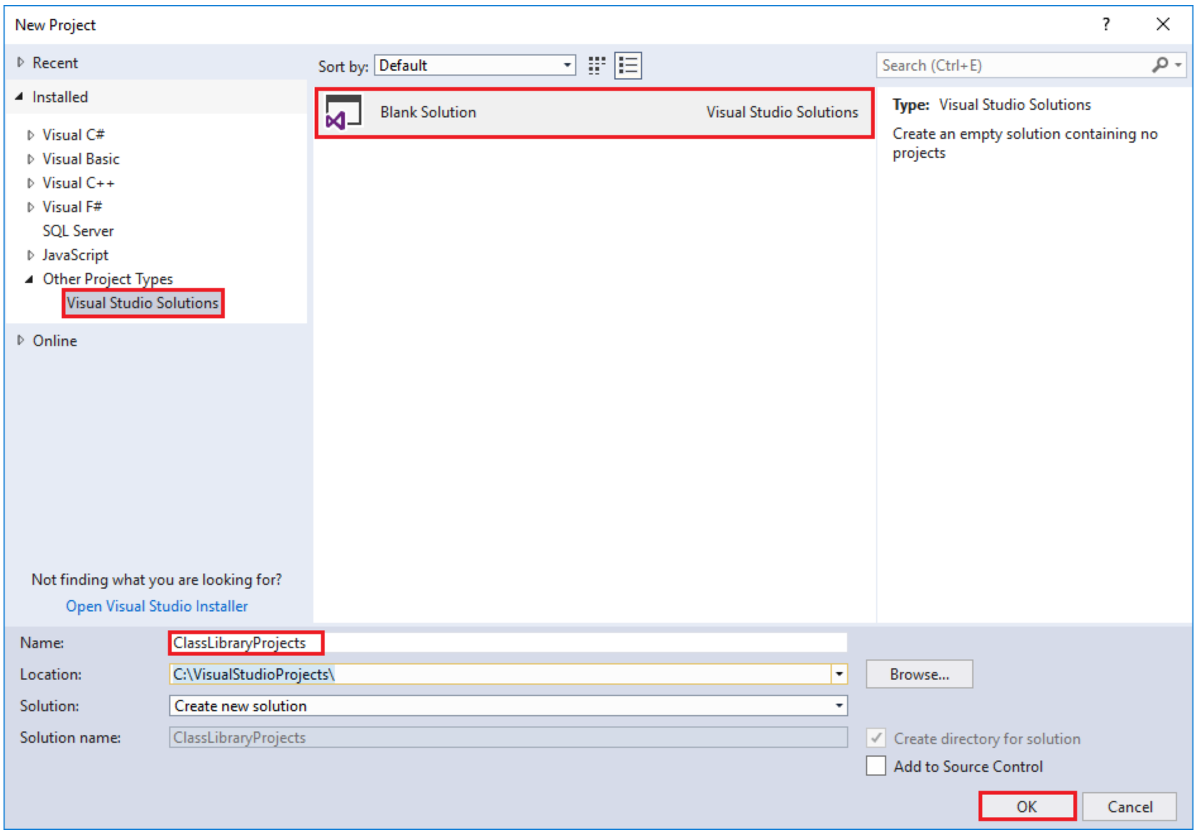
For a list of the .NET Standard versions and the platforms they support, see .NET Standard.

In this topic, you'll create a simple utility library that contains a single string-handling method. You'll implement it as an extension method so that you can call it as if it were a member of the String class.

Creating a class library solution

Start by creating a solution for your class library project and its related projects. A Visual Studio Solution just serves as a container for one or more projects. To create the solution:

1. On the Visual Studio menu bar, choose **File** > **New** > **Project**.
2. In the **New Project** dialog, expand the **Other Project Types** node, and select **Visual Studio Solutions**. Name the solution "ClassLibraryProjects" and select the **OK** button.



Sort by: Default

Type: Visual Studio Solutions

Create an empty solution containing no projects

Name: ClassLibraryProjects

Location: C:\VisualStudioProjects\

Solution: Create new solution

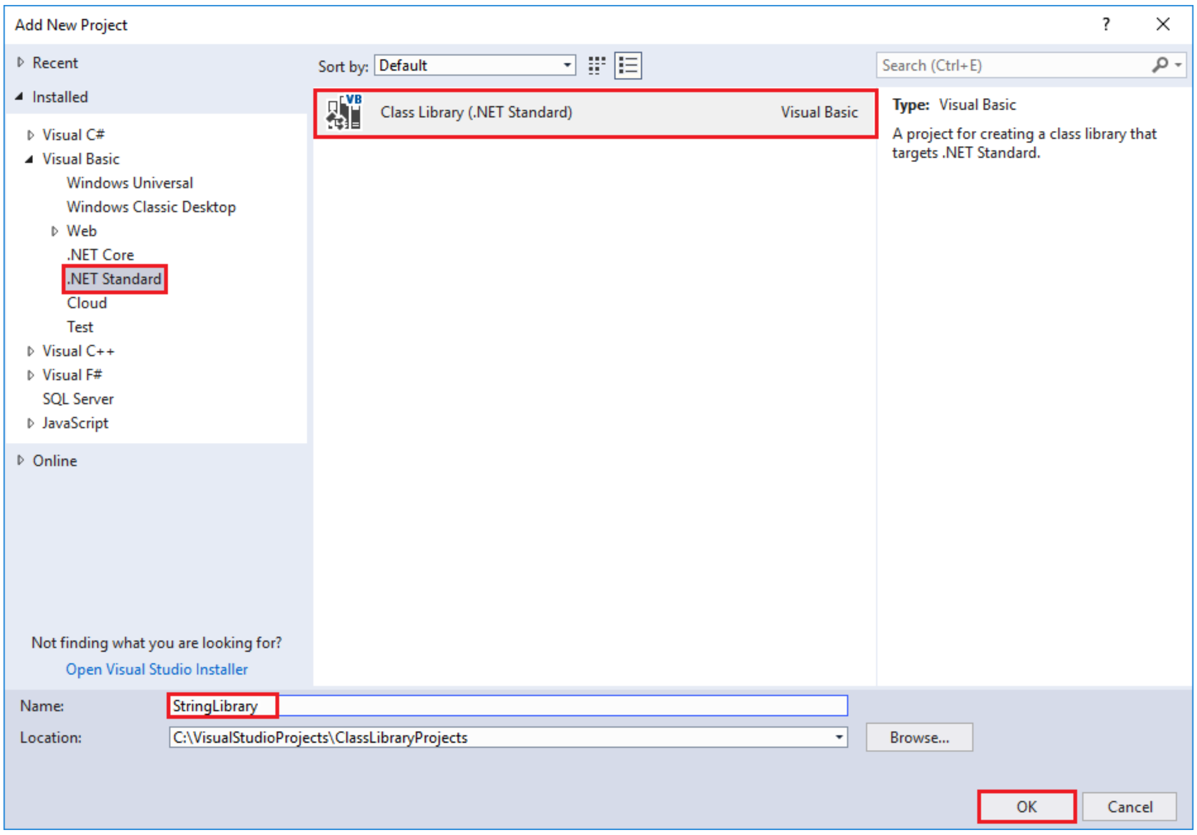
Solution name: ClassLibraryProjects

Creating the class library project

Create your class library project:

1. In **Solution Explorer**, right-click on the **ClassLibraryProjects** solution file and from the context menu, select **Add** > **New Project**.

2. In the **Add New Project** dialog, expand the **Visual Basic** node, then select the **.NET Standard** node followed by the **Class Library (.NET Standard)** project template. In the **Name** text box, enter "StringLibrary" as the name of the project. Select **OK** to create the class library project.



Add New Project

Recent
Installed
Visual C#
Visual Basic
Windows Universal
Windows Classic Desktop
Web
.NET Core
.NET Standard
Cloud
Test
Visual C++
Visual F#
SQL Server
JavaScript
Online

Not finding what you are looking for?
Open Visual Studio Installer

Sort by: Default

Class Library (.NET Standard) Visual Basic

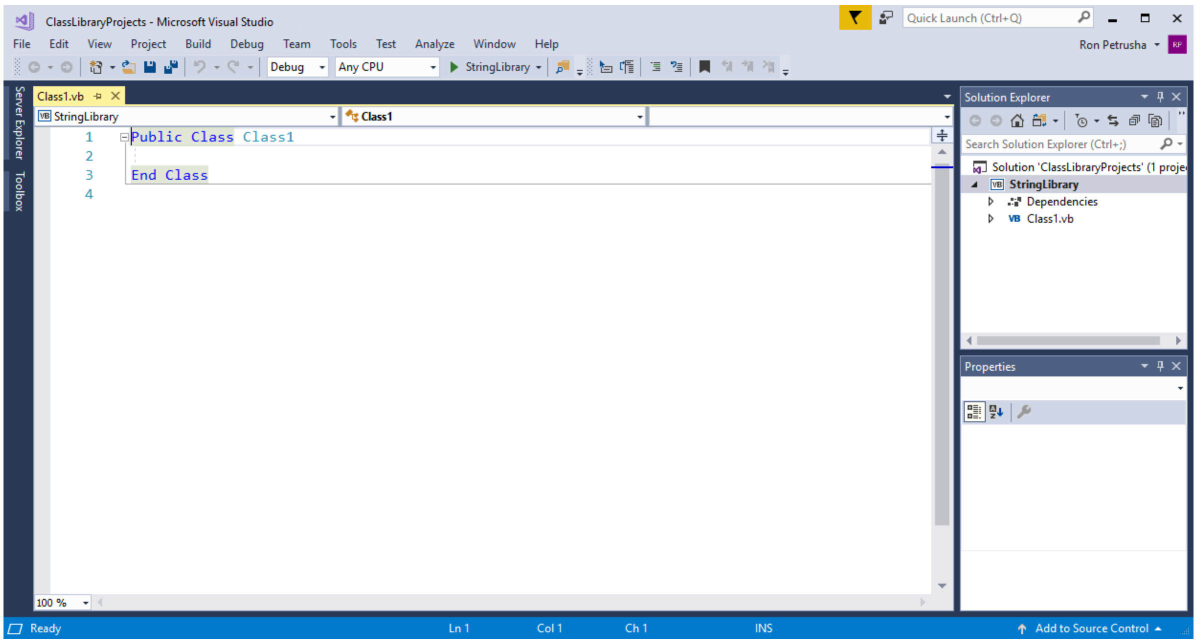
Search (Ctrl+E)

Type: Visual Basic
A project for creating a class library that targets .NET Standard.

Name: StringLibrary
Location: C:\VisualStudioProjects\ClassLibraryProjects

Browse...
OK Cancel

The code window then opens in the Visual Studio development environment.



ClassLibraryProjects - Microsoft Visual Studio

File Edit View Project Build Debug Team Tools Test Analyze Window Help

Class1.vb — StringLibrary — Class1

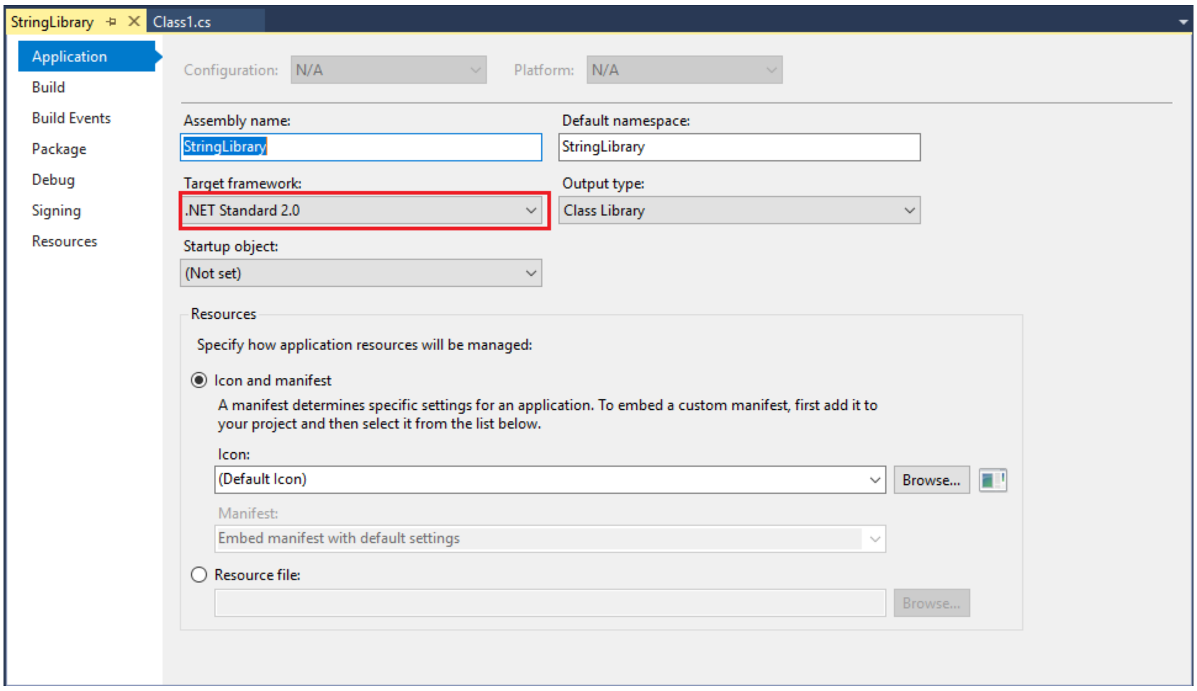
```
Public Class Class1

End Class
```

Solution Explorer
Solution 'ClassLibraryProjects' (1 proje
StringLibrary
Dependencies
Class1.vb

Properties

Ready Ln 1 Col 1 Ch 1 INS Add to Source Control

3. Check to make sure that the library targets the correct version of the .NET Standard. Right-click on the library project in the **Solution Explorer** windows, then select **Properties**. The **Target Framework** text box shows that we're targeting .NET Standard 2.0.



Configuration: N/A
Platform: N/A
Assembly name: StringLibrary
Default namespace: StringLibrary
Target framework: .NET Standard 2.0
Output type: Class Library
Startup object: (Not set)
Resources
Specify how application resources will be managed:
● Icon and manifest
A manifest determines specific settings for an application. To embed a custom manifest, first add it to your project and then select it from the list below.
Icon: (Default Icon)
Manifest: Embed manifest with default settings
○ Resource file:

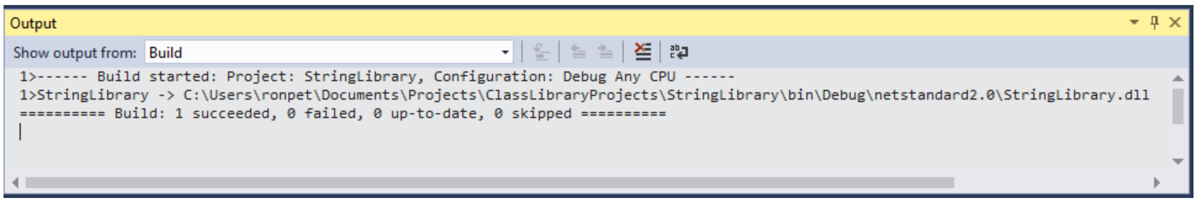
4. Also in the **Properties** dialog, clear the text in the **Root namespace** text box. For each project, Visual Basic automatically creates a namespace that corresponds to the project name, and any namespaces defined in source code files are parents of that namespace. We want to define a top-level namespace by using the `namespace` keyword.

5. Replace the code in the code window with the following code and save the file:

```vb
Imports System.Runtime.CompilerServices

Namespace UtilityLibraries
    Public Module StringLibrary
        <Extension>
        Public Function StartsWithUpper(str As String) As Boolean
            If String.IsNullOrWhiteSpace(str) Then
                Return False
            End If

            Dim ch As Char = str(0)
            Return Char.IsUpper(ch)
        End Function
    End Module
End Namespace
```

The class library, `UtilityLibraries.StringLibrary`, contains a method named `StartsWithUpper`, which returns a Boolean value that indicates whether the current string instance begins with an uppercase character. The Unicode standard distinguishes uppercase characters from lowercase characters. The Char.IsUpper(Char) method returns `true` if a character is uppercase.

1. On the menu bar, select **Build** > **Build Solution**. The project should compile without error.



```
1>------ Build started: Project: StringLibrary, Configuration: Debug Any CPU ------
1>StringLibrary -> C:\Users\ronpet\Documents\Projects\ClassLibraryProjects\StringLibrary\bin\Debug\netstandard2.0\StringLibrary.dll
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Next step

You've successfully built the library. Because you haven't called any of its methods, you don't know whether it works as expected. The next step in developing your library is to test it by using a Unit Test Project.

Test a .NET Standard library with .NET Core in Visual Studio 2017

10/29/2019 • 9 minutes to read • Edit Online

In Build a .NET Standard library with C# and .NET Core in Visual Studio 2017 or Build a .NET Standard library with Visual Basic and .NET Core in Visual Studio 2017, you created a simple class library that adds an extension method to the String class. Now, you'll create a unit test to make sure that it works as expected. You'll add your unit test project to the solution you created in the previous article.

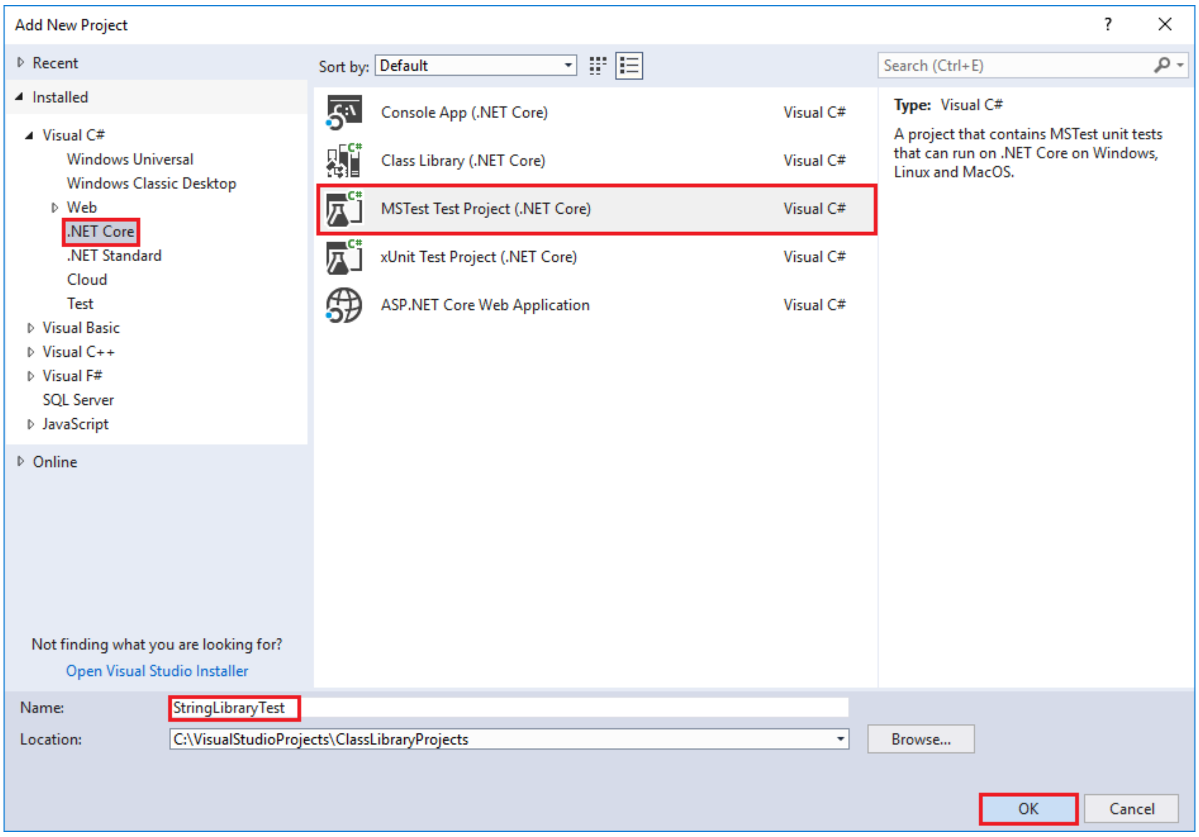
Creating a unit test project

To create the unit test project, do the following:

- C#
- Visual Basic

1. In **Solution Explorer**, open the context menu for the **ClassLibraryProjects** solution node and select **Add** > **New Project**.

2. In the **Add New Project** dialog, select the **Visual C#** node. Then select the **.NET Core** node followed by the **MSTest Test Project (.NET Core)** project template. In the **Name** text box, enter "StringLibraryTest" as the name of the project. Select **OK** to create the unit test project.


Add New Project
?
Recent
Installed
Visual C#
Windows Universal
Windows Classic Desktop
Web
.NET Core
.NET Standard
Cloud
Test
Visual Basic
Visual C++
Visual F#
SQL Server
JavaScript
Online
Sort by: Default
Search (Ctrl+E)
Console App (.NET Core) Visual C#
Class Library (.NET Core) Visual C#
MSTest Test Project (.NET Core) Visual C#
xUnit Test Project (.NET Core) Visual C#
ASP.NET Core Web Application Visual C#
Not finding what you are looking for?
Open Visual Studio Installer
Browse...
OK
Cancel

Type: Visual C#

A project that contains MSTest unit tests that can run on .NET Core on Windows, Linux and MacOS.

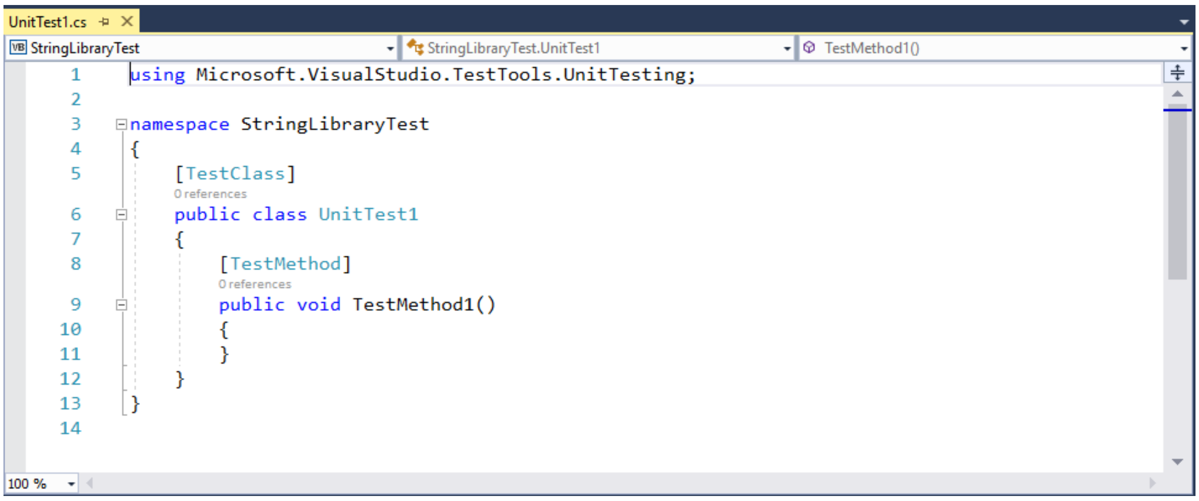
Name: StringLibraryTest

Location: C:\VisualStudioProjects\ClassLibraryProjects

NOTE

In addition to an MSTest Test project, you can also use Visual Studio to create an xUnit test project for .NET Core.

3. Visual Studio creates the project and opens the *UnitTest1.cs* file in the code window.

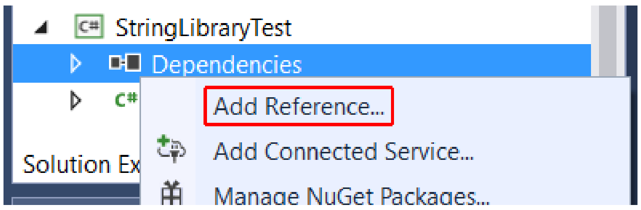


```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace StringLibraryTest
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

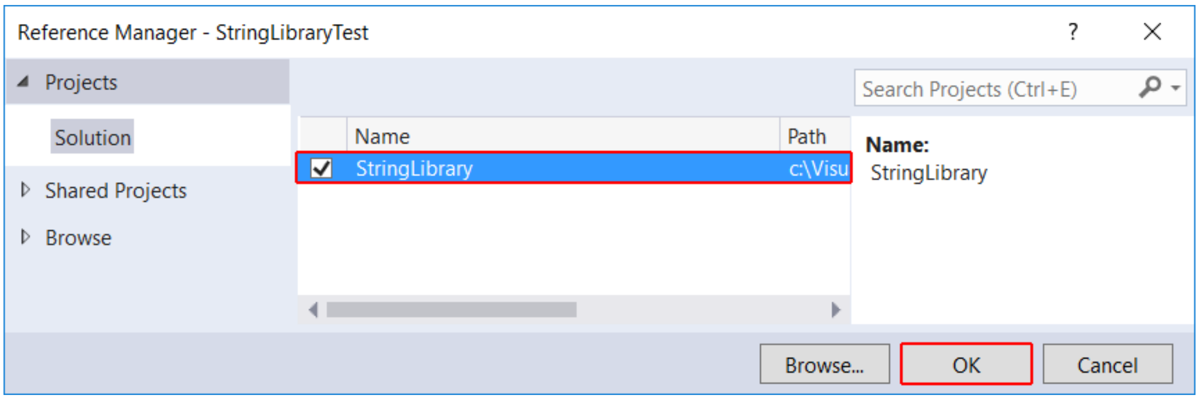
The source code created by the unit test template does the following:

- It imports the Microsoft.VisualStudio.TestTools.UnitTesting namespace, which contains the types used for unit testing.
- It applies the TestClassAttribute attribute to the `UnitTest1` class. Each test method in a test class tagged with the [TestMethod] attribute is executed automatically when the unit test is run.
- It applies the TestMethodAttribute attribute to define `TestMethod1` as a test method for automatic execution when the unit test is run.

4. In **Solution Explorer**, right-click the **Dependencies** node of the **StringLibraryTest** project and select **Add Reference** from the context menu.



5. In the **Reference Manager** dialog, expand the **Projects** node and check the box next to **StringLibrary**. Adding a reference to the `StringLibrary` assembly allows the compiler to find **StringLibrary** methods. Select the **OK** button. This adds a reference to your class library project, `StringLibrary`.



Adding and running unit test methods

When Visual Studio runs a unit test, it executes each method marked with the TestMethodAttribute attribute in a unit test class, the class to which the TestClassAttribute attribute is applied. A test method ends when the first failure is encountered or when all tests contained in the method have succeeded.

The most common tests call members of the Assert class. Many assert methods include at least two parameters, one of which is the expected test result and the other of which is the actual test result. Some of its most frequently called methods are shown in the following table:

ASSERT METHODS	FUNCTION
`Assert.AreEqual`	Verifies that two values or objects are equal. The assert fails if the values or objects are not equal.
`Assert.AreSame`	Verifies that two object variables refer to the same object. The assert fails if the variables refer to different objects.
`Assert.IsFalse`	Verifies that a condition is `false`. The assert fails if the condition is `true`.
`Assert.IsNotNull`	Verifies that an object is not `null`. The assert fails if the object is `null`.

You can also apply the ExpectedExceptionAttribute attribute to a test method. It indicates the type of exception a test method is expected to throw. The test fails if the specified exception is not thrown.

In testing the `StringLibrary.StartsWithUpper` method, you want to provide a number of strings that begin with an uppercase character. You expect the method to return `true` in these cases, so you can call the IsTrue method. Similarly, you want to provide a number of strings that begin with something other than an uppercase character. You expect the method to return `false` in these cases, so you can call the IsFalse method.

Since your library method handles strings, you also want to make sure that it successfully handles an empty string (`String.Empty`), a valid string that has no characters and whose Length is 0, and a `null` string that has not been initialized. If `StartsWithUpper` is called as an extension method on a String instance, it cannot be passed a `null` string. However, you can also call it directly as a static method and pass a single String argument.

You'll define three methods, each of which calls its Assert method repeatedly for each element in a string array. Because the test method fails as soon as it encounters the first failure, you'll call a method overload that allows you to pass a string that indicates the string value used in the method call.

To create the test methods:

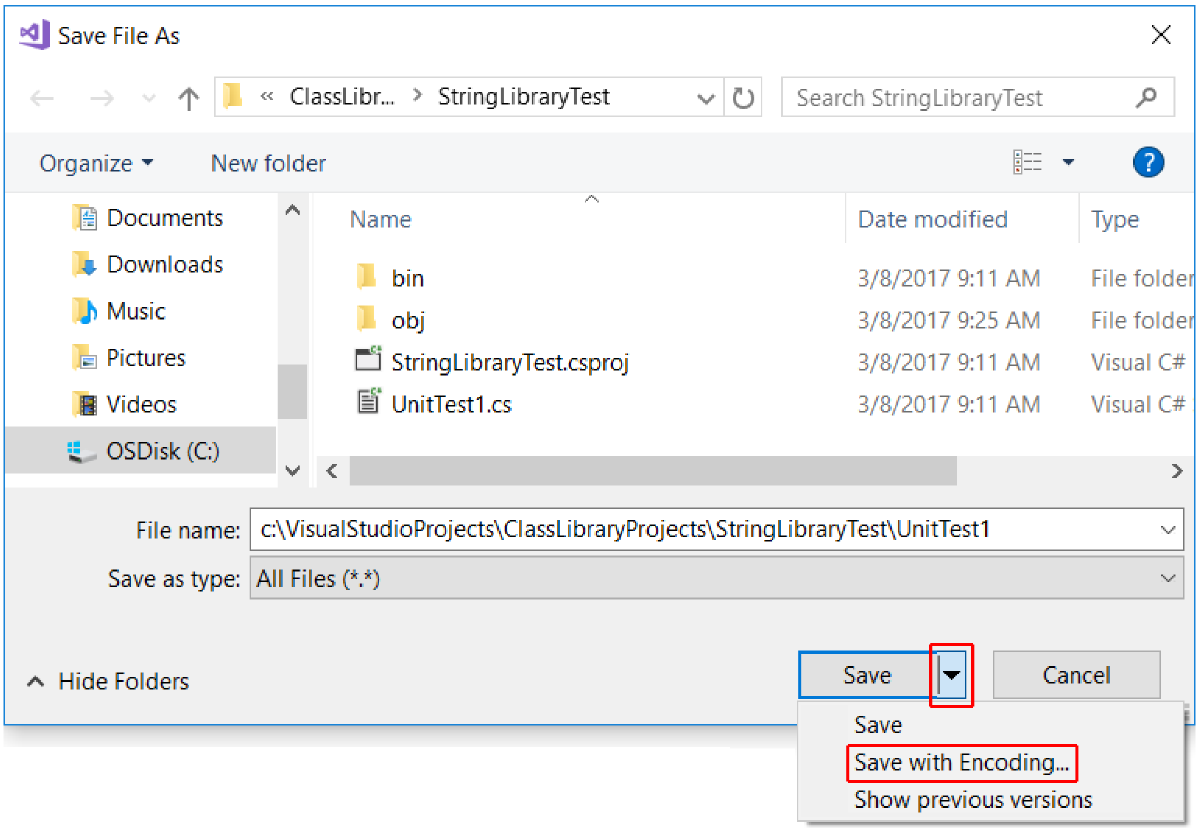
- C#
- Visual Basic

1. In the *UnitTest1.cs* code window, replace the code with the following code:

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using UtilityLibraries;

namespace StringLibraryTest
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestStartsWithUpper()
        {
            // Tests that we expect to return true.
            string[] words = { "Alphabet", "Zebra", "ABC", "Αθήνα", "Москва" };
            foreach (var word in words)
            {
                bool result = word.StartsWithUpper();
                Assert.IsTrue(result,
                       String.Format("Expected for '{0}': true; Actual: {1}",
                                     word, result));
            }
        }

        [TestMethod]
        public void TestDoesNotStartWithUpper()
        {
            // Tests that we expect to return false.
            string[] words = { "alphabet", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                               "1234", ".", ";", " " };
            foreach (var word in words)
            {
                bool result = word.StartsWithUpper();
                Assert.IsFalse(result,
                       String.Format("Expected for '{0}': false; Actual: {1}",
                                     word, result));
            }
        }

        [TestMethod]
        public void DirectCallWithNullOrEmpty()
        {
            // Tests that we expect to return false.
            string[] words = { string.Empty, null };
            foreach (var word in words)
            {
                bool result = StringLibrary.StartsWithUpper(word);
                Assert.IsFalse(result,
                       String.Format("Expected for '{0}': false; Actual: {1}",
                                     word == null ? "<null>" : word, result));
            }
        }
    }
}
```

Note that your test of uppercase characters in the `TestStartsWithUpper` method includes the Greek capital letter alpha (U+0391) and the Cyrillic capital letter EM (U+041C), and the test of lowercase characters in the `TestDoesNotStartWithUpper` method includes the Greek small letter alpha (U+03B1) and the Cyrillic small letter Ghe (U+0433).

2. On the menu bar, select **File** > **Save UnitTest1.cs As**. In the **Save File As** dialog, select the arrow beside the **Save** button, and select **Save with Encoding**.



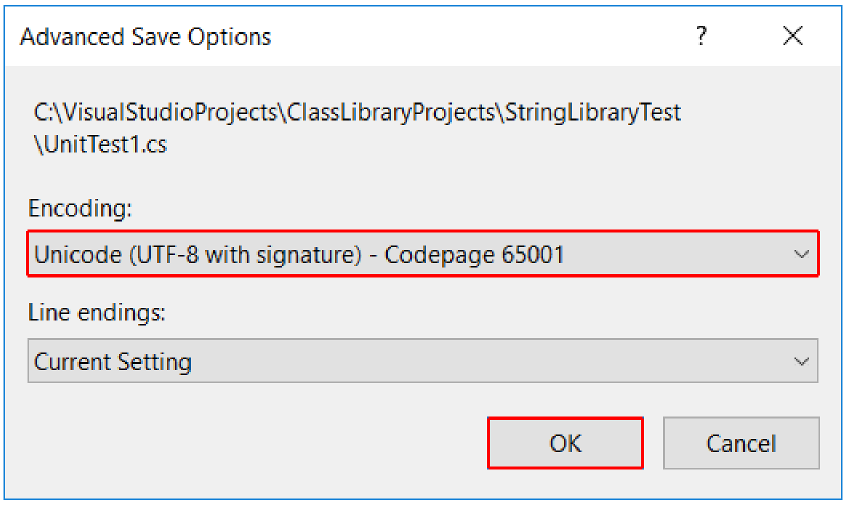
Name	Date modified	Type
bin	3/8/2017 9:11 AM	File folder
obj	3/8/2017 9:25 AM	File folder
StringLibraryTest.csproj	3/8/2017 9:11 AM	Visual C#
UnitTest1.cs	3/8/2017 9:11 AM	Visual C#

File name: c:\VisualStudioProjects\ClassLibraryProjects\StringLibraryTest\UnitTest1

Save as type: All Files (*.*)

1. In the **Confirm Save As** dialog, select the **Yes** button to save the file.

2. In the **Advanced Save Options** dialog, select **Unicode (UTF-8 with signature) - Codepage 65001** from the **Encoding** drop-down list and select **OK**.



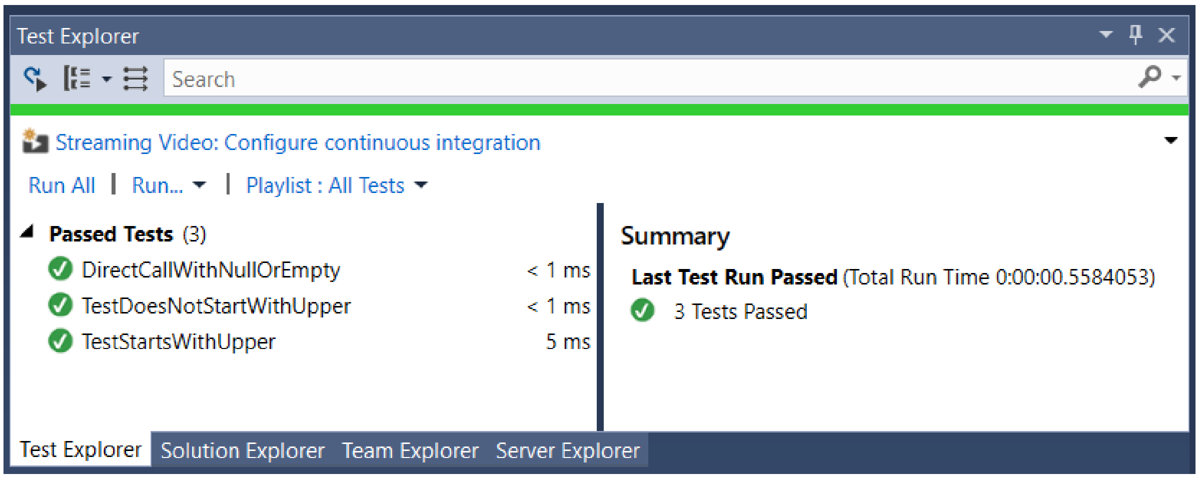
C:\VisualStudioProjects\ClassLibraryProjects\StringLibraryTest
\UnitTest1.cs

Encoding: Unicode (UTF-8 with signature) - Codepage 65001

Line endings: Current Setting

If you fail to save your source code as a UTF8-encoded file, Visual Studio may save it as an ASCII file. When that happens, the runtime doesn't accurately decode the UTF8 characters outside of the ASCII range, and the test results won't be accurate.

3. On the menu bar, select **Test** > **Run** > **All Tests**. The **Test Explorer** window opens and shows that the tests ran successfully. The three tests are listed in the **Passed Tests** section, and the **Summary** section reports the result of the test run.


Test Explorer

Streaming Video: Configure continuous integration

Run All | Run... | Playlist : All Tests

Passed Tests (3)
- DirectCallWithNullOrEmpty < 1 ms
- TestDoesNotStartWithUpper < 1 ms
- TestStartsWithUpper 5 ms

Summary

Last Test Run Passed (Total Run Time 0:00:00.5584053)

3 Tests Passed

Handling test failures

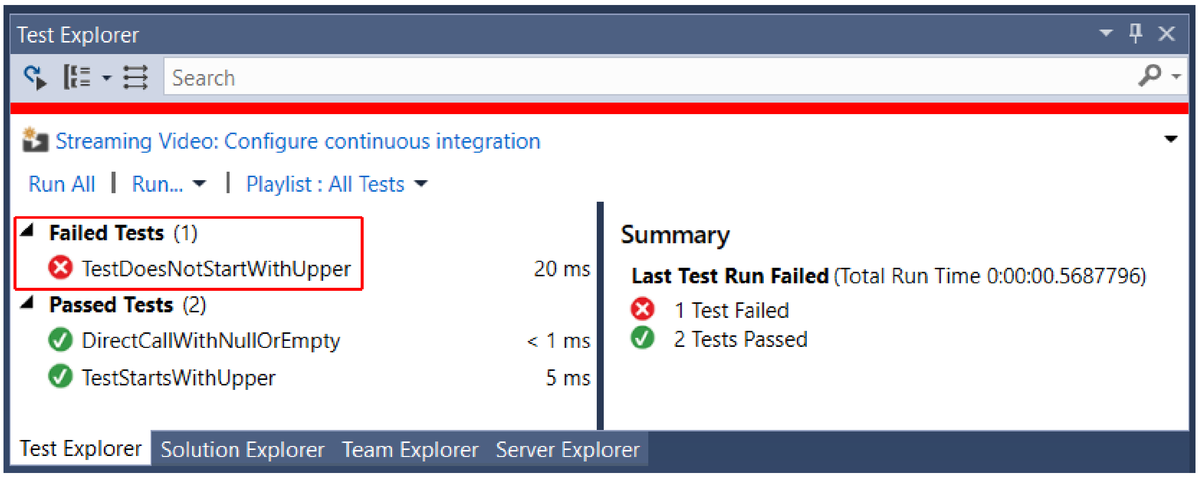
Your test run had no failures, but change it slightly so that one of the test methods fails:

1. Modify the `words` array in the `TestDoesNotStartWithUpper` method to include the string "Error". You don't need to save the file because Visual Studio automatically saves open files when a solution is built to run tests.

```
string[] words = { "alphabet", "Error", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                   "1234", ".", ";", " " };
```

```
Dim words() As String = { "alphabet", "Error", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                   "1234", ".", ";", " " }
```

2. Run the test by selecting **Test** > **Run** > **All Tests** from the menu bar. The **Test Explorer** window indicates that two tests succeeded and one failed.


Test Explorer

Streaming Video: Configure continuous integration

Run All | Run... | Playlist : All Tests

Failed Tests (1)
- TestDoesNotStartWithUpper 20 ms

Passed Tests (2)
- DirectCallWithNullOrEmpty < 1 ms
- TestStartsWithUpper 5 ms

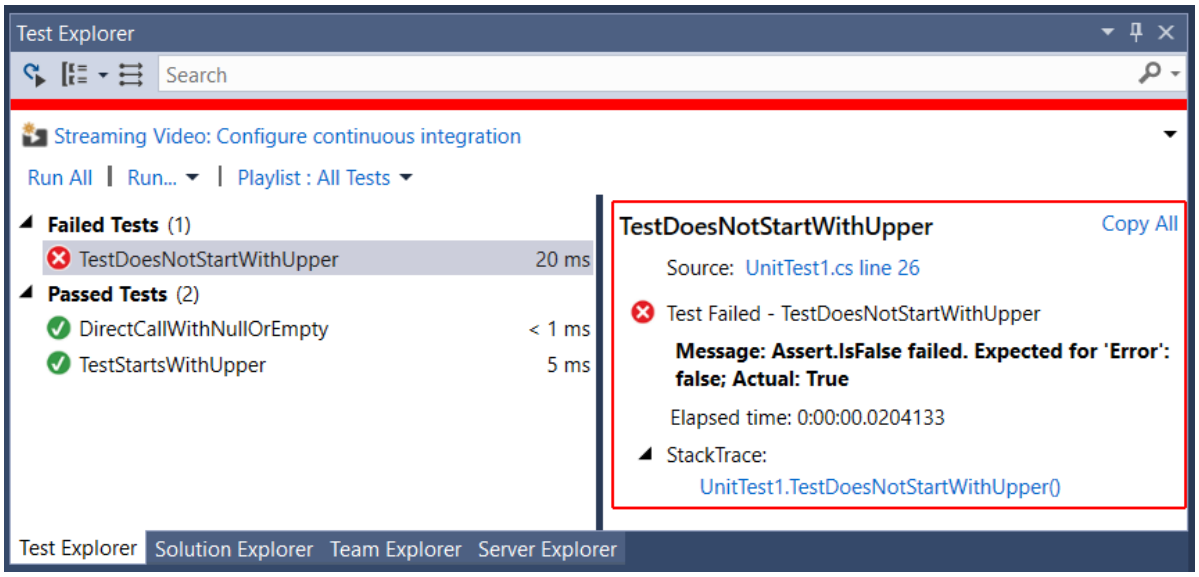
Summary

Last Test Run Failed (Total Run Time 0:00:00.5687796)

1 Test Failed

2 Tests Passed

3. In the **Failed Tests** section, select the failed test, `TestDoesNotStartWith`. The **Test Explorer** window displays the message produced by the assert: "Assert.IsFalse failed. Expected for 'Error': false; actual: True". Because of the failure, all strings in the array after "Error" were not tested.


TestDoesNotStartWithUpper

Source: UnitTest1.cs line 26

Test Failed - TestDoesNotStartWithUpper

Message: Assert.IsFalse failed. Expected for 'Error': false; Actual: True

Elapsed time: 0:00:00.0204133

StackTrace:

UnitTest1.TestDoesNotStartWithUpper()

Test Explorer
Search
Streaming Video: Configure continuous integration
Run All | Run... | Playlist : All Tests
Failed Tests (1)
TestDoesNotStartWithUpper 20 ms
Passed Tests (2)
DirectCallWithNullOrEmpty < 1 ms
TestStartsWithUpper 5 ms
Copy All
Test Explorer
Solution Explorer
Team Explorer
Server Explorer

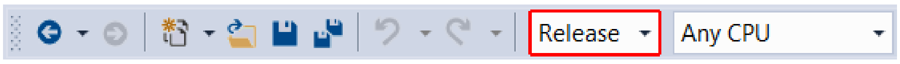
4. Undo the modification you did in step 1 and remove the string "Error". Rerun the test and the tests will pass.

Testing the Release version of the library

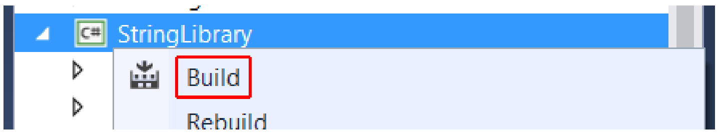
You've been running your tests against the Debug version of the library. Now that your tests have all passed and you've adequately tested your library, you should run the tests an additional time against the Release build of the library. A number of factors, including compiler optimizations, can sometimes produce different behavior between Debug and Release builds.

To test the Release build:

1. In the Visual Studio toolbar, change the build configuration from **Debug** to **Release**.



2. In **Solution Explorer**, right-click the **StringLibrary** project and select **Build** from the context menu to recompile the library.



3. Run the unit tests by choosing **Test** > **Run** > **All Tests** from the menu bar. The tests pass.

Now that you've finished testing your library, the next step is to make it available to callers. You can bundle it with one or more applications, or you can distribute it as a NuGet package. For more information, see Consuming a .NET Standard Class Library.

Consume a .NET Standard library in Visual Studio 2017

11/1/2019 • 5 minutes to read • Edit Online

Once you've created a .NET Standard class library by following the steps in Building a C# class library with .NET Core in Visual Studio 2017 or Building a Visual Basic class library with .NET Core in Visual Studio 2017, tested it in Testing a class library with .NET Core in Visual Studio 2017, and built a Release version of the library, the next step is to make it available to callers. You can do this in two ways:

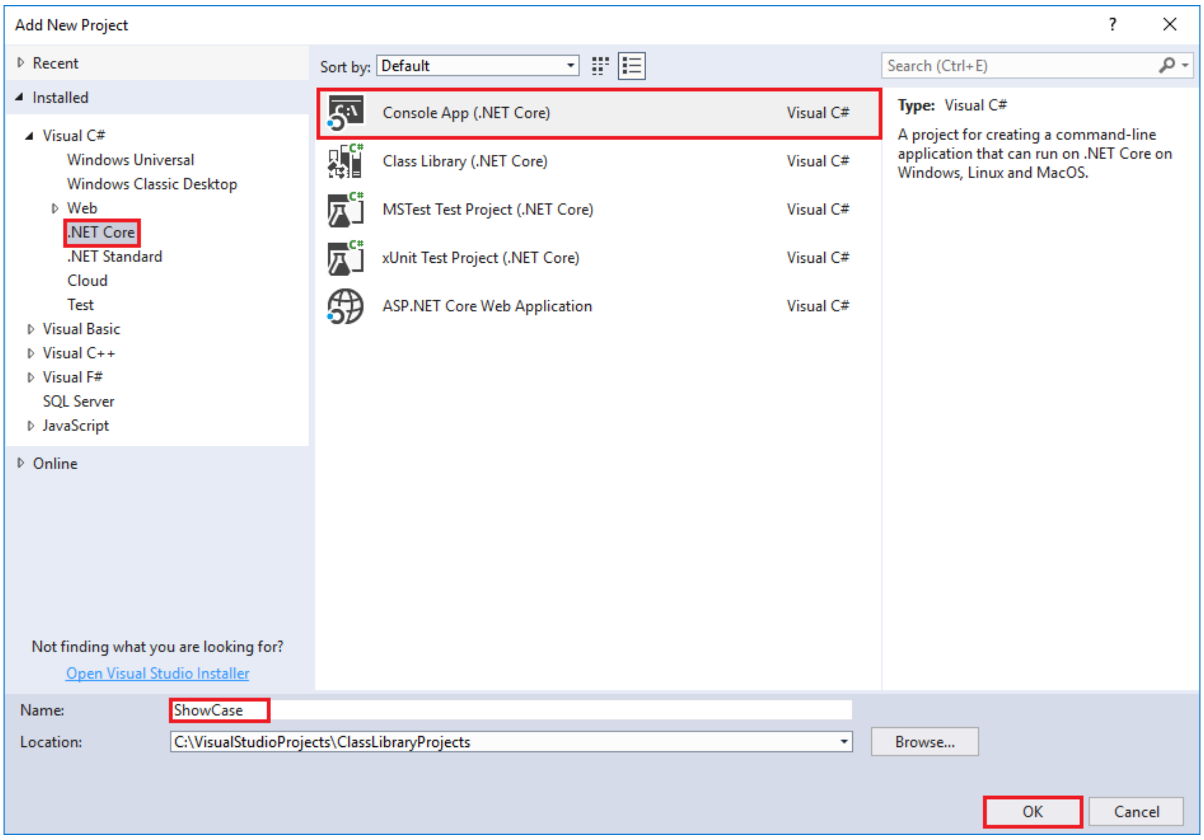
- If the library will be used by a single solution (for example, if it's a component in a single large application), you can include it as a project in your solution.
- If the library will be generally accessible, you can distribute it as a NuGet package.

Including a library as a project in a solution

Just as you included unit tests in the same solution as your class library, you can include your application as part of that solution. For example, you can use your class library in a console application that prompts the user to enter a string and reports whether its first character is uppercase:

- C#
- Visual Basic

1. Open the `ClassLibraryProjects` solution you created in the Building a C# Class Library with .NET Core in Visual Studio 2017 topic. In **Solution Explorer**, right-click the **ClassLibraryProjects** solution and select **Add** > **New Project** from the context menu.
2. In the **Add New Project** dialog, expand the **Visual C#** node and select the **.NET Core** node followed by the **Console App (.NET Core)** project template. In the **Name** text box, type "ShowCase", and select the **OK** button.


Add New Project
Recent
Installed
Visual C#
Windows Universal
Windows Classic Desktop
Web
.NET Core
.NET Standard
Cloud
Test
Visual Basic
Visual C++
Visual F#
SQL Server
JavaScript
Online
Not finding what you are looking for?
Open Visual Studio Installer
Sort by: Default
Search (Ctrl+E)
Console App (.NET Core) Visual C#
Class Library (.NET Core) Visual C#
MSTest Test Project (.NET Core) Visual C#
xUnit Test Project (.NET Core) Visual C#
ASP.NET Core Web Application Visual C#
Browse...
OK
Cancel

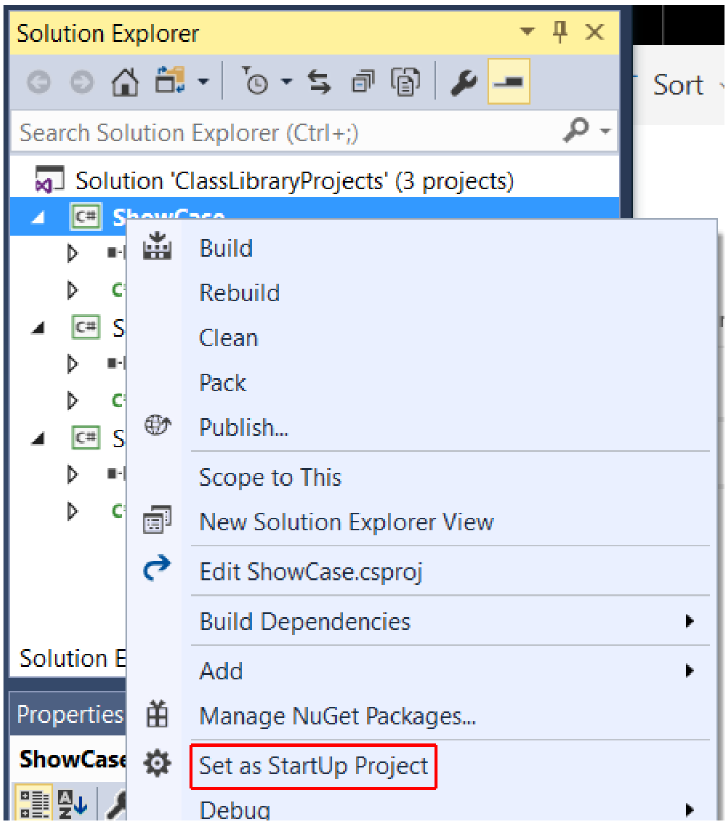
Type: Visual C#

A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.

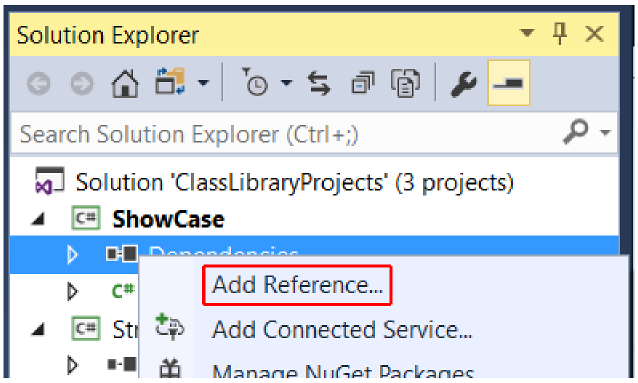
Name: ShowCase

Location: C:\VisualStudioProjects\ClassLibraryProjects

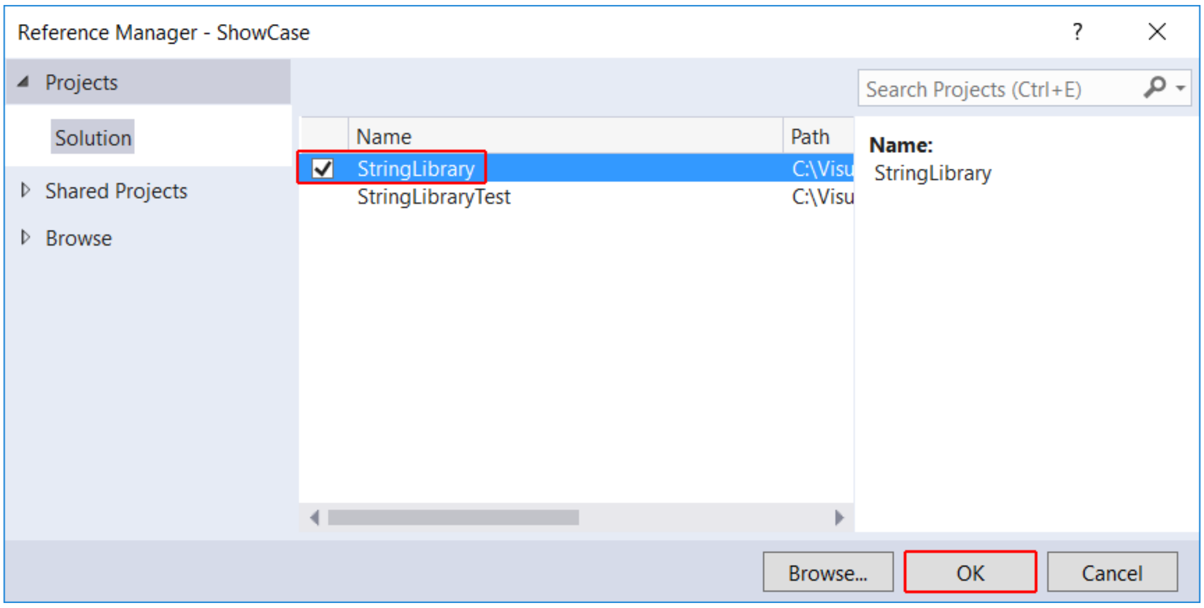
3. In **Solution Explorer**, right-click the **ShowCase** project and select **Set as StartUp Project** in the context menu.



4. Initially, your project doesn't have access to your class library. To allow it to call methods in your class library, you create a reference to the class library. In **Solution Explorer**, right-click the `ShowCase` project's **Dependencies** node and select **Add Reference**.



5. In the **Reference Manager** dialog, select **StringLibrary**, your class library project, and select the **OK** button.



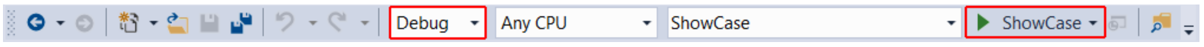
6. In the code window for the *Program.cs* file, replace all of the code with the following code:

```
using System;
using UtilityLibraries;

class Program
{
    static void Main(string[] args)
    {
        int row = 0;

        do
        {
            if (row == 0 || row >= 25)
               ResetConsole();

            string input = Console.ReadLine();
            if (String.IsNullOrEmpty(input)) break;
            Console.WriteLine($"Input: {input} {"Begins with uppercase? ",30}: " +
                              $"{(input.StartsWithUpper() ? "Yes" : "No")}\n");
            row += 3;
        } while (true);
        return;

        // Declare a ResetConsole local method
        void ResetConsole()
        {
            if (row > 0) {
               Console.WriteLine("Press any key to continue...");
               Console.ReadKey();
            }
            Console.Clear();
            Console.WriteLine("\nPress <Enter> only to exit; otherwise, enter a string and press
<Enter>:\n");
            row = 3;
        }
    }
}
```

The code uses the `row` variable to maintain a count of the number of rows of data written to the console window. Whenever it is greater than or equal to 25, the code clears the console window and displays a message to the user.

The program prompts the user to enter a string. It indicates whether the string starts with an uppercase character. If the user presses the Enter key without entering a string, the application terminates, and the console window closes.

7. If necessary, change the toolbar to compile the **Debug** release of the `ShowCase` project. Compile and run the program by selecting the green arrow on the **ShowCase** button.



You can debug and publish the application that uses this library by following the steps in Debugging your Hello World application with Visual Studio 2017 and Publishing your Hello World Application with Visual Studio 2017.

Distributing the library in a NuGet package

You can make your class library widely available by publishing it as a NuGet package. Visual Studio does not support the creation of NuGet packages. To create one, you use the `dotnet` command line utility:

1. Open a console window. For example in the **Ask me anything** text box in the Windows taskbar, enter `Command Prompt` (or `cmd` for short), and open a console window by either selecting the **Command Prompt** desktop app or pressing Enter if it's selected in the search results.

2. Navigate to your library's project directory. Unless you've reconfigured the typical file location, it's in the *Documents\Visual Studio 2017\Projects\ClassLibraryProjects\StringLibrary* directory. The directory contains your source code and a project file, *StringLibrary.csproj*.

3. Issue the command `dotnet pack --no-build`. The `dotnet` utility generates a package with a *.nupkg* extension.

   > **TIP**
   >
   > If the directory that contains *dotnet.exe* is not in your PATH, you can find its location by entering `where dotnet.exe` in the console window.

For more information on creating NuGet packages, see How to Create a NuGet Package with Cross Platform Tools.

Prerequisites for .NET Core on Windows

10/22/2019 • 4 minutes to read • Edit Online

This article shows the supported OS versions in order to run .NET Core applications on Windows. The supported OS versions and dependencies that follow apply to the three ways of developing .NET Core apps on Windows:

- Command line
- Visual Studio
- Visual Studio Code

Also, if you're developing on Windows using Visual Studio, the Prerequisites to develop .NET Core apps with Visual Studio section goes in more detail about minimum versions supported for .NET Core development.

.NET Core supported operating systems

The following articles have a complete list of .NET Core supported operating systems per version:

- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1

For download links and more information, see .NET downloads to download the latest version or .NET downloads archive for older versions.

.NET Core dependencies

Microsoft Visual C++ 2015 Redistributable Update 3 must be manually installed when:

- Installing .NET Core with the installer script.
- Deploying a self-contained .NET Core application.
- Building the product from source.
- Installing .NET Core via a *.zip* file. This can include build/CI/CD servers.

NOTE

For Windows 8.1 and earlier versions, or Windows Server 2012 R2 and earlier versions:

Make sure that your Windows installation is up-to-date and includes KB2999226, which can be installed through Windows Update. If you don't have this update installed, you'll see an error like the following when you launch a .NET Core application:

```
The program can't start because api-ms-win-crt-runtime-l1-1-0.dll is missing from your computer. Try
reinstalling the program to fix this problem.
```

For Windows 7 or Windows Server 2008 R2:

In addition to KB2999226, make sure you also have KB2533623 installed. If you don't have this update installed, you'll see an error similar to the following when you launch a .NET Core application:

`The library hostfxr.dll was found, but loading it from C:\<path_to_app>\hostfxr.dll failed`.

Prerequisites to develop .NET Core apps with Visual Studio

Even though you can use any editor to develop .NET Core applications using the .NET Core SDK, Visual Studio 2017 and later versions provide an integrated development environment for .NET Core apps on Windows.

Each .NET Core version has a minimum version of Visual Studio required. To verify your Visual Studio version:

- On the **Help** menu, choose **About Microsoft Visual Studio**.
- In the **About Microsoft Visual Studio** dialog, verify the version number.

The following table lists the minimum version for each SDK:

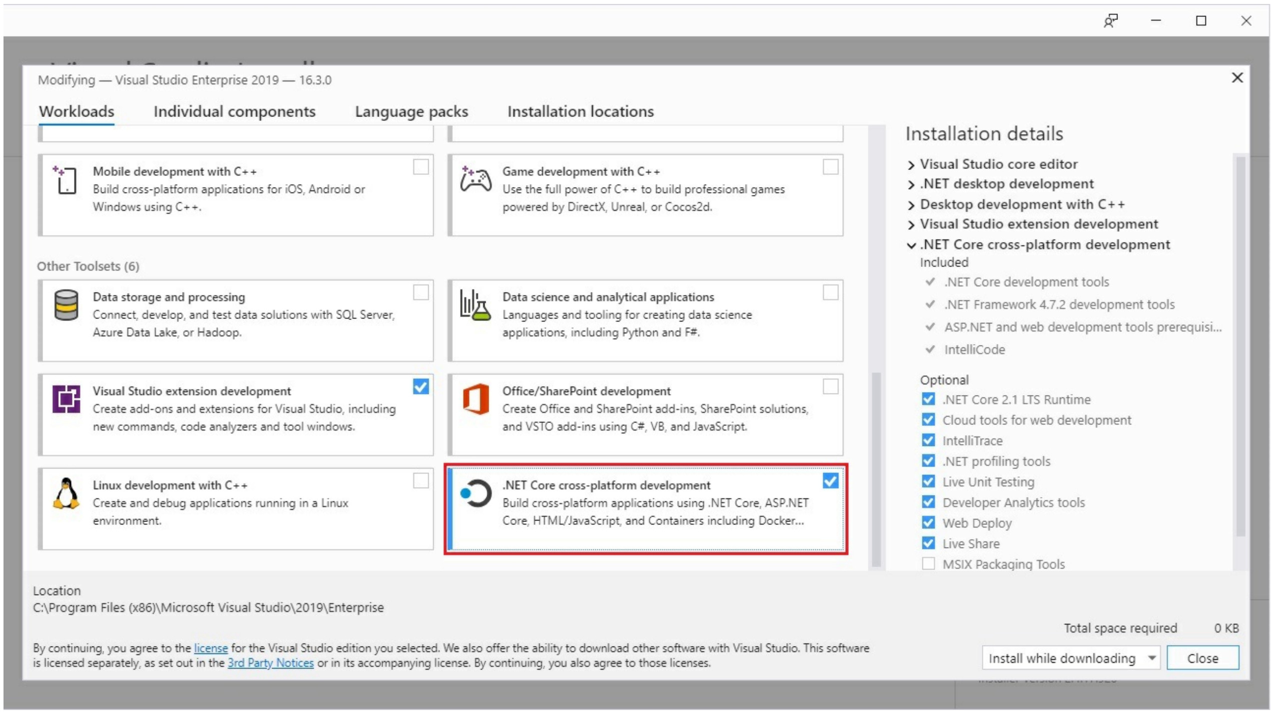
.NET CORE SDK VERSION	VISUAL STUDIO VERSION
3.0	Visual Studio 2019 version 16.3 or higher.
2.2	Visual Studio 2017 version 15.9 or higher.
2.1	Visual Studio 2017 version 15.7 or higher.
1.x	Visual Studio 2017 version 15.0 or higher.

- .NET Core 3.0
- .NET Core 2.x

To develop .NET Core apps in Visual Studio 2019 using the .NET Core 3.0 SDK:

- Download and install Visual Studio 2019 version 16.3 or higher and select one of the following workloads that includes the .NET Core SDK, depending on the kind of application you're building:
  - The **.NET Core cross-platform development** workload in the **Other Toolsets** section.
  - The **ASP.NET and web development** workload in the **Web & Cloud** section.
  - The **NET desktop development** workload in the **Windows** section.

The following image shows the **.NET Core cross-platform development** workload selected in the Visual Studio UI:



Modifying — Visual Studio Enterprise 2019 — 16.3.0

Workloads | Individual components | Language packs | Installation locations

Mobile development with C++ — Build cross-platform applications for iOS, Android or Windows using C++.

Game development with C++ — Use the full power of C++ to build professional games powered by DirectX, Unreal, or Cocos2d.

Other Toolsets (6)

Data storage and processing — Connect, develop, and test data solutions with SQL Server, Azure Data Lake, or Hadoop.

Data science and analytical applications — Languages and tooling for creating data science applications, including Python and F#.

[✓] Visual Studio extension development — Create add-ons and extensions for Visual Studio, including new commands, code analyzers and tool windows.

Office/SharePoint development — Create Office and SharePoint add-ins, SharePoint solutions, and VSTO add-ins using C#, VB, and JavaScript.

Linux development with C++ — Create and debug applications running in a Linux environment.

[✓] .NET Core cross-platform development — Build cross-platform applications using .NET Core, ASP.NET Core, HTML/JavaScript, and Containers including Docker...

Installation details
- Visual Studio core editor
- .NET desktop development
- Desktop development with C++
- Visual Studio extension development
- .NET Core cross-platform development
  - Included
    - ✓ .NET Core development tools
    - ✓ .NET Framework 4.7.2 development tools
    - ✓ ASP.NET and web development tools prerequisi...
    - ✓ IntelliCode
  - Optional
    - [✓] .NET Core 2.1 LTS Runtime
    - [✓] Cloud tools for web development
    - [✓] IntelliTrace
    - [✓] .NET profiling tools
    - [✓] Live Unit Testing
    - [✓] Developer Analytics tools
    - [✓] Web Deploy
    - [✓] Live Share
    - [ ] MSIX Packaging Tools

Location
C:\Program Files (x86)\Microsoft Visual Studio\2019\Enterprise

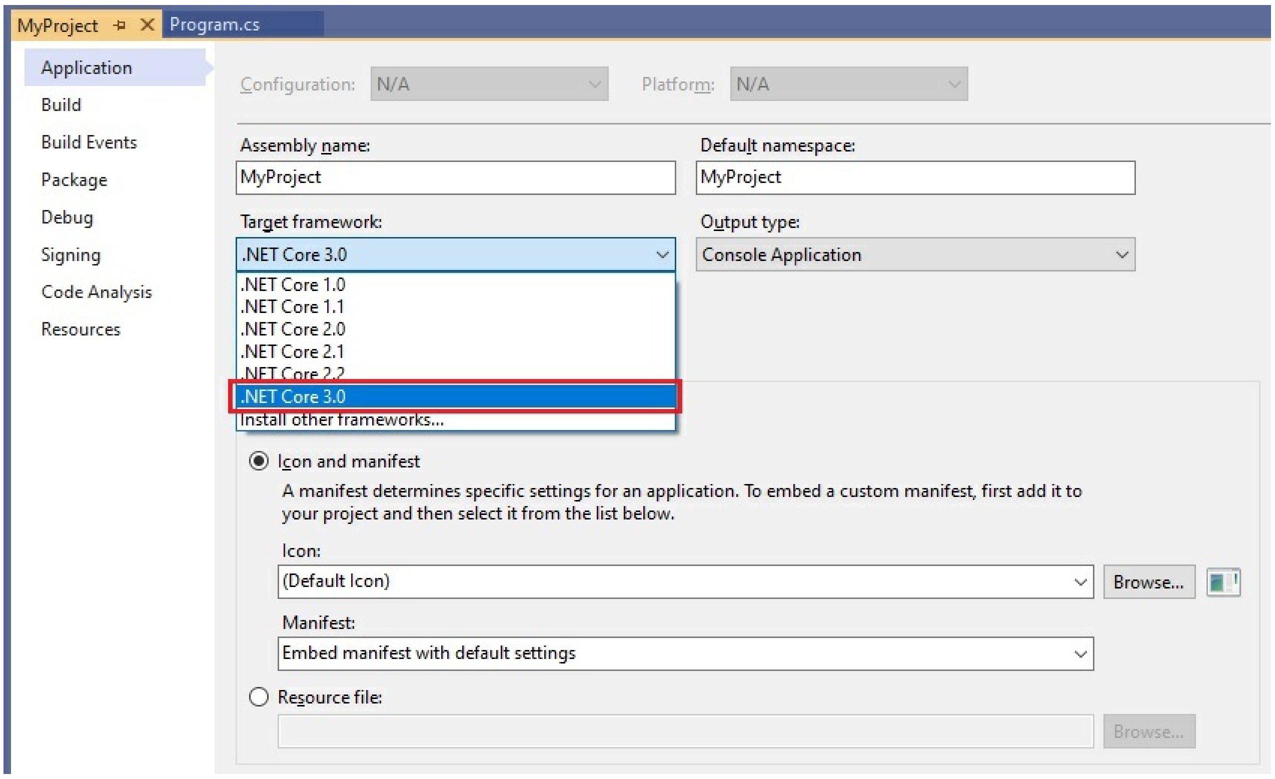
By continuing, you agree to the license for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.

Total space required 0 KB

Visual Studio 2019 version 16.3 uses .NET Core 3.0 SDK by default after any of these workloads are installed.

If you want your existing projects to use the latest .NET Core runtime, retarget each existing .NET Core project to .NET Core 3.0 using the following instructions:

- On the **Project** menu, choose **Properties**.
- In the **Target framework** selection menu, set the value to **.NET Core 3.0**.



Assembly name: MyProject

Default namespace: MyProject

Target framework: .NET Core 3.0

Output type: Console Application

Icon and manifest

A manifest determines specific settings for an application. To embed a custom manifest, first add it to your project and then select it from the list below.

Icon: (Default Icon)

Manifest: Embed manifest with default settings

Once you have Visual Studio configured with .NET Core 3.0 SDK, you can do the following actions:

- Open, build, and run existing .NET Core 1.x and 2.x projects.
- Retarget .NET Core 1.x and 2.x projects to .NET Core 3.0, build, and run.
- Create new .NET Core 3.0 projects.

Prerequisites for .NET Core on macOS

11/12/2019 • 2 minutes to read • Edit Online

This article shows you the supported macOS versions and .NET Core dependencies that you need to develop, deploy, and run .NET Core applications on macOS machines. The supported OS versions and dependencies that follow apply to the three ways of developing .NET Core apps on a Mac: via the command-line with your favorite editor, Visual Studio Code, and Visual Studio for Mac.

Downloads and dependencies

- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1

.NET Core 3.0 is supported on **macOS High Sierra (version 10.13)** and later versions. A **x64** CPU architecture is required.

Download and install the .NET Core SDK from the .NET Core 3.0 downloads page. .NET Core 3.0 Supported OS Versions for the complete list of .NET Core 3.0 supported operating systems, distributions and versions, out of support OS versions, and lifecycle policy links.

For a list of known issues, see .NET Core known issues.

libgdiplus

.NET Core applications that use the *System.Drawing.Common* assembly require libgdiplus to be installed.

An easy way to obtain libgdiplus is by using the Homebrew ("brew") package manager for macOS. After installing *brew*, install libgdiplus by executing the following commands at a Terminal (command) prompt:

```
brew update
brew install mono-libgdiplus
```

Visual Studio for Mac

You can use any editor to develop .NET Core applications using the .NET Core SDK. However, if you want to develop .NET Core applications on a Mac in an integrated development environment, you can use Visual Studio for Mac.

Prerequisites for .NET Core on Linux

10/17/2019 • 4 minutes to read • Edit Online

This article shows the dependencies needed to develop .NET Core applications on Linux. The supported Linux distributions/versions, and dependencies that follow apply to the two ways of developing .NET Core apps on Linux:

- Command-line with your favorite editor
- Visual Studio Code

NOTE

The .NET Core SDK package is not required for production servers/environments. Only the .NET Core runtime package is needed for apps deployed to production environments. The .NET Core runtime is deployed with apps as part of a self-contained deployment, however, it must be deployed for Framework-dependent deployed apps separately. For more information about framework-dependent and self-contained deployment types, see .NET Core application deployment. Also see Self-contained Linux applications for specific guidelines.

Supported Linux versions

- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1

.NET Core 3.0 treats Linux as a single operating system. There is a single Linux build (per chip architecture) for supported Linux distributions.

For download links and more information, see .NET Core 3.0 downloads.

.NET Core 3.0 is supported on the following Linux distributions/versions):

NOTE

A `+` symbol represents the minimum version.

OS	VERSION	ARCHITECTURES
Red Hat Enterprise Linux	6+, 7	x64
Oracle Linux	7	x64
CentOS	7	x64
Fedora	29+	x64
Debian	9+	x64, ARM32, ARM64
Ubuntu	16.04+	x64, ARM32, ARM64

OS	VERSION	ARCHITECTURES
Linux Mint	18+	x64
openSUSE	15+	x64
SUSE Enterprise Linux (SLES)	12 SP2+	x64
Alpine Linux	3.8+	x64, ARM64

See .NET Core 3.0 Supported OS Versions for the complete list of .NET Core 3.0 supported operating systems, distributions and versions, out of support OS versions, and lifecycle policy links.

For more information about how to install .NET Core 3.0 on ARM64, see Installing .NET Core 3.0 on Linux ARM64.

Linux distribution dependencies

The following are intended to be examples. The exact versions and names may vary slightly on your Linux distribution of choice.

Ubuntu

Ubuntu distributions require the following libraries installed:

- liblttng-ust0
- libcurl3 (for 14.x and 16.x)
- libcurl4 (for 18.x)
- libssl1.0.0
- libkrb5-3
- zlib1g
- libicu52 (for 14.x)
- libicu55 (for 16.x)
- libicu57 (for 17.x)
- libicu60 (for 18.x)

For versions earlier than .NET Core 2.1, following dependencies are also required:

- libunwind8
- libuuid1

For .NET Core applications that use the *System.Drawing.Common* assembly, you also need the following dependency:

- libgdiplus (version 6.0.1 or later)

NOTE

Most versions of Ubuntu include an earlier version of libgdiplus. You can install a recent version of libgdiplus by adding the Mono repository to your system. For more information, see https://www.mono-project.com/download/stable/.

CentOS and Fedora

CentOS distributions require the following libraries installed:

- lttng-ust

- libcurl
- openssl-libs
- krb5-libs
- libicu
- zlib

Fedora users: If your openssl's version >= 1.1, you'll need to install compat-openssl10.

For versions earlier than .NET Core 2.1, following dependencies are also required:

- libunwind
- libuuid

For more information about the dependencies, see Self-contained Linux applications.

For .NET Core applications that use the *System.Drawing.Common* assembly, you'll also need the following dependency:

- libgdiplus (version 6.0.1 or later)

NOTE

Most versions of CentOS and Fedora include an earlier version of libgdiplus. You can install a recent version of libgdiplus by adding the Mono repository to your system. For more information, see https://www.mono-project.com/download/stable/.

Installing .NET Core dependencies with the native installers

.NET Core native installers are available for supported Linux distributions/versions. The native installers require admin (sudo) access to the server. The advantage of using a native installer is that all of the .NET Core native dependencies are installed. Native installers also install the .NET Core SDK system-wide.

On Linux, there are two installer package choices:

- Using a feed-based package manager, such as apt-get for Ubuntu, or yum for CentOS/RHEL.
- Using the packages themselves, DEB or RPM.

Scripting Installs with the .NET Core installer script

The dotnet-install scripts are used to perform a non-admin install of the CLI toolchain and the shared runtime. You can download the script from https://dot.net/v1/dotnet-install.sh.

The script defaults to installing the latest "LTS" version, which is currently .NET Core 1.1. To install .NET Core 2.1, run the script with the following switch:

```
./dotnet-install.sh -c Current
```

The installer bash script is used in automation scenarios and non-admin installations. This script also reads PowerShell switches, so they can be used with the script on Linux/OS X systems.

Troubleshoot

If you have problems with a .NET Core installation on a supported Linux distribution/version, consult the following topics for your installed distributions/versions:

- .NET Core 3.0 known issues
- .NET Core 2.2 known issues

- .NET Core 2.1 known issues
- .NET Core 1.1 known issues
- .NET Core 1.0 known issues

What's new in .NET Core 3.0

11/12/2019 • 20 minutes to read • Edit Online

This article describes what is new in .NET Core 3.0. One of the biggest enhancements is support for Windows desktop applications (Windows only). By using the .NET Core 3.0 SDK component Windows Desktop, you can port your Windows Forms and Windows Presentation Foundation (WPF) applications. To be clear, the Windows Desktop component is only supported and included on Windows. For more information, see the Windows desktop section later in this article.

.NET Core 3.0 adds support for C# 8.0. It's highly recommended that you use Visual Studio 2019 version 16.3 or newer, Visual Studio for Mac 8.3 or newer, or Visual Studio Code with the latest **C# extension**.

Download and get started with .NET Core 3.0 right now on Windows, macOS, or Linux.

For more information about the release, see the .NET Core 3.0 announcement.

.NET Core RC1 was considered production ready by Microsoft and was fully supported. If you're using a preview release, you must move to the RTM version for continued support.

Language improvements C# 8.0

C# 8.0 is also part of this release, which includes the nullable reference types feature, async streams, and more patterns. For more information about C# 8.0 features, see What's new in C# 8.0.

Language enhancements were added to support the following API features detailed below:

- Ranges and indices
- Async streams

.NET Standard 2.1

.NET Core 3.0 implements **.NET Standard 2.1**. However, the default `dotnet new classlib` template generates a project that still targets **.NET Standard 2.0**. To target **.NET Standard 2.1**, edit your project file and change the `TargetFramework` property to `netstandard2.1`:

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

  <PropertyGroup>
    <TargetFramework>netstandard2.1</TargetFramework>
  </PropertyGroup>

</Project>
```

If you're using Visual Studio, you need Visual Studio 2019, as Visual Studio 2017 doesn't support **.NET Standard 2.1** or **.NET Core 3.0**.

Compile/Deploy

Default executables

.NET Core now builds framework-dependent executables by default. This behavior is new for applications that use a globally installed version of .NET Core. Previously, only self-contained deployments would produce an executable.

During `dotnet build` or `dotnet publish`, an executable is created that matches the environment and platform of the SDK you're using. You can expect the same things with these executables as you would other native executables, such as:

- You can double-click on the executable.
- You can launch the application from a command prompt directly, such as `myapp.exe` on Windows, and `./myapp` on Linux and macOS.

Single-file executables

The `dotnet publish` command supports packaging your app into a platform-specific single-file executable. The executable is self-extracting and contains all dependencies (including native) that are required to run your app. When the app is first run, the application is extracted to a directory based on the app name and build identifier. Startup is faster when the application is run again. The application doesn't need to extract itself a second time unless a new version was used.

To publish a single-file executable, set the `PublishSingleFile` in your project or on the command line with the `dotnet publish` command:

```
<PropertyGroup>
  <RuntimeIdentifier>win10-x64</RuntimeIdentifier>
  <PublishSingleFile>true</PublishSingleFile>
</PropertyGroup>
```

-or-

```
dotnet publish -r win10-x64 -p:PublishSingleFile=true
```

For more information about single-file publishing, see the single-file bundler design document.

Assembly linking

The .NET core 3.0 SDK comes with a tool that can reduce the size of apps by analyzing IL and trimming unused assemblies.

Self-contained apps include everything needed to run your code, without requiring .NET to be installed on the host computer. However, many times the app only requires a small subset of the framework to function, and other unused libraries could be removed.

.NET Core now includes a setting that will use the IL linker tool to scan the IL of your app. This tool detects what code is required, and then trims unused libraries. This tool can significantly reduce the deployment size of some apps.

To enable this tool, add the `<PublishTrimmed>` setting in your project and publish a self-contained app:

```
<PropertyGroup>
  <PublishTrimmed>true</PublishTrimmed>
</PropertyGroup>
```

```
dotnet publish -r <rid> -c Release
```

As an example, the basic "hello world" new console project template that is included, when published, hits about 70 MB in size. By using `<PublishTrimmed>`, that size is reduced to about 30 MB.

It's important to consider that applications or frameworks (including ASP.NET Core and WPF) that use reflection or related dynamic features, will often break when trimmed. This breakage occurs because the linker doesn't know about this dynamic behavior and can't determine which framework types are required for reflection. The IL Linker tool can be configured to be aware of this scenario.

Above all else, be sure to test your app after trimming.

For more information about the IL Linker tool, see the documentation or visit the mono/linker repo.

Tiered compilation

Tiered compilation (TC) is on by default with .NET Core 3.0. This feature enables the runtime to more adaptively use the Just-In-Time (JIT) compiler to get better performance.

The main benefit of TC is to enable (re-)jitting methods with a lower-quality-but-faster tier or a higher-quality-but-slower tier. This helps increase performance of an application as it goes through various stages of execution, from startup through steady-state. This contrasts with the non-TC approach, where every method is compiled a single way (the same as the high-quality tier), which is biased to steady-state over startup performance.

When TC is enabled, during startup for a method that is called:

- If the method has AOT-compiled code (ReadyToRun), the pregenerated code will be used.
- Otherwise, the method will be jitted. Typically, these methods currently are generics over value types.
  - Quick JIT produces lower-quality code more quickly. Quick JIT is enabled by default in .NET Core 3.0 for methods that do not contain loops and is preferred during startup.
  - The fully-optimizing JIT produces higher-quality code more slowly. For methods where Quick JIT would not be used (for example, if the method is attributed with `[MethodImpl(MethodImplOptions.AggressiveOptimization)]`), the fully-optimizing JIT is used.

Eventually, after methods are called a number of times, they are re-jitted with the fully-optimizing JIT in the background.

Code generated by Quick JIT may run slower, allocate more memory, or use more stack space. If there are issues, Quick JIT may be disabled using this setting in your project file:

```
<PropertyGroup>
  <TieredCompilationQuickJit>false</TieredCompilationQuickJit>
</PropertyGroup>
```

To disable TC completely, use this setting in your project file:

```
<PropertyGroup>
  <TieredCompilation>false</TieredCompilation>
</PropertyGroup>
```

Any changes to the above settings in the project file may require a clean build to be reflected (delete the `obj` and `bin` directories and rebuild).

ReadyToRun images

You can improve the startup time of your .NET Core application by compiling your application assemblies as ReadyToRun (R2R) format. R2R is a form of ahead-of-time (AOT) compilation.

R2R binaries improve startup performance by reducing the amount of work the just-in-time (JIT) compiler needs to do as your application loads. The binaries contain similar native code compared to what the JIT would produce. However, R2R binaries are larger because they contain both intermediate language (IL) code, which is still needed for some scenarios, and the native version of the same code. R2R is only available when you publish a self-contained app that targets specific runtime environments (RID) such as Linux x64 or Windows x64.

To compile your project as ReadyToRun, do the following:

1. Add the `<PublishReadyToRun>` setting to your project:

   ```
   <PropertyGroup>
     <PublishReadyToRun>true</PublishReadyToRun>
   </PropertyGroup>
   ```

2. Publish a self-contained app. For example, this command creates a self-contained app for the 64-bit version of Windows:

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

Cross platform/architecture restrictions

The ReadyToRun compiler doesn't currently support cross-targeting. You must compile on a given target. For example, if you want R2R images for Windows x64, you need to run the publish command on that environment.

Exceptions to cross-targeting:

- Windows x64 can be used to compile Windows ARM32, ARM64, and x86 images.
- Windows x86 can be used to compile Windows ARM32 images.
- Linux x64 can be used to compile Linux ARM32 and ARM64 images.

Runtime/SDK

Major-version Roll Forward

.NET Core 3.0 introduces an opt-in feature that allows your app to roll forward to the latest major version of .NET Core. Additionally, a new setting has been added to control how roll forward is applied to your app. This can be configured in the following ways:

- Project file property: `RollForward`
- Runtime configuration file property: `rollForward`
- Environment variable: `DOTNET_ROLL_FORWARD`
- Command-line argument: `--roll-forward`

One of the following values must be specified. If the setting is omitted, **Minor** is the default.

- **LatestPatch**
  Roll forward to the highest patch version. This disables minor version roll forward.
- **Minor**
  Roll forward to the lowest higher minor version, if requested minor version is missing. If the requested minor version is present, then the **LatestPatch** policy is used.
- **Major**
  Roll forward to lowest higher major version, and lowest minor version, if requested major version is missing. If the requested major version is present, then the **Minor** policy is used.
- **LatestMinor**
  Roll forward to highest minor version, even if requested minor version is present. Intended for component hosting scenarios.
- **LatestMajor**
  Roll forward to highest major and highest minor version, even if requested major is present. Intended for component hosting scenarios.
- **Disable**
  Don't roll forward. Only bind to specified version. This policy isn't recommended for general use because it disables the ability to roll forward to the latest patches. This value is only recommended for testing.

Besides the **Disable** setting, all settings will use the highest available patch version.

Build copies dependencies

The `dotnet build` command now copies NuGet dependencies for your application from the NuGet cache to the build output folder. Previously, dependencies were only copied as part of `dotnet publish`.

There are some operations, like linking and razor page publishing that will still require publishing.

Local tools

.NET Core 3.0 introduces local tools. Local tools are similar to global tools but are associated with a particular location on disk. Local tools aren't available globally and are distributed as NuGet packages.

WARNING

If you tried local tools in .NET Core 3.0 Preview 1, such as running `dotnet tool restore` or `dotnet tool install`, delete the local tools cache folder. Otherwise, local tools won't work on any newer release. This folder is located at:

On macOS, Linux: `rm -r $HOME/.dotnet/toolResolverCache`

On Windows: `rmdir /s %USERPROFILE%\.dotnet\toolResolverCache`

Local tools rely on a manifest file name `dotnet-tools.json` in your current directory. This manifest file defines the tools to be available at that folder and below. You can distribute the manifest file with your code to ensure that anyone who works with your code can restore and use the same tools.

For both global and local tools, a compatible version of the runtime is required. Many tools currently on NuGet.org target .NET Core Runtime 2.1. To install these tools globally or locally, you would still need to install the NET Core 2.1 Runtime.

Smaller Garbage Collection heap sizes

The Garbage Collector's default heap size has been reduced resulting in .NET Core using less memory. This change better aligns with the generation 0 allocation budget with modern processor cache sizes.

Garbage Collection Large Page support

Large Pages (also known as Huge Pages on Linux) is a feature where the operating system is able to establish memory regions larger than the native page size (often 4K) to improve performance of the application requesting these large pages.

The Garbage Collector can now be configured with the **GCLargePages** setting as an opt-in feature to choose to allocate large pages on Windows.

Windows Desktop & COM

.NET Core SDK Windows Installer

The MSI installer for Windows has changed starting with .NET Core 3.0. The SDK installers will now upgrade SDK feature-band releases in place. Feature bands are defined in the *hundreds* groups in the *patch* section of the version number. For example, **3.0.*101*** and **3.0.*201*** are versions in two different feature bands while **3.0.*101*** and **3.0.*199*** are in the same feature band. And, when .NET Core SDK **3.0.*101*** is installed, .NET Core SDK **3.0.*100*** will be removed from the machine if it exists. When .NET Core SDK **3.0.*200*** is installed on the same machine, .NET Core SDK **3.0.*101*** won't be removed.

For more information about versioning, see Overview of how .NET Core is versioned.

Windows desktop

.NET Core 3.0 supports Windows desktop applications using Windows Presentation Foundation (WPF) and Windows Forms. These frameworks also support using modern controls and Fluent styling from the Windows UI

XAML Library (WinUI) via XAML islands.

The Windows Desktop component is part of the Windows .NET Core 3.0 SDK.

You can create a new WPF or Windows Forms app with the following `dotnet` commands:

```
dotnet new wpf
dotnet new winforms
```

Visual Studio 2019 adds **New Project** templates for .NET Core 3.0 Windows Forms and WPF.

For more information about how to port an existing .NET Framework application, see Port WPF projects and Port Windows Forms projects.

WinForms high DPI

.NET Core Windows Forms applications can set high DPI mode with Application.SetHighDpiMode(HighDpiMode). The `SetHighDpiMode` method sets the corresponding high DPI mode unless the setting has been set by other means like `App.Manifest` or P/Invoke before `Application.Run`.

The possible `highDpiMode` values, as expressed by the System.Windows.Forms.HighDpiMode enum are:

- `DpiUnaware`
- `SystemAware`
- `PerMonitor`
- `PerMonitorV2`
- `DpiUnawareGdiScaled`

For more information about high DPI modes, see High DPI Desktop Application Development on Windows.

Create COM components

On Windows, you can now create COM-callable managed components. This capability is critical to use .NET Core with COM add-in models and also to provide parity with .NET Framework.

Unlike .NET Framework where the *mscoree.dll* was used as the COM server, .NET Core will add a native launcher dll to the *bin* directory when you build your COM component.

For an example of how to create a COM component and consume it, see the COM Demo.

Windows Native Interop

Windows offers a rich native API in the form of flat C APIs, COM, and WinRT. While .NET Core supports **P/Invoke**, .NET Core 3.0 adds the ability to **CoCreate COM APIs** and **Activate WinRT APIs**. For a code example, see the Excel Demo.

MSIX Deployment

MSIX is a new Windows application package format. It can be used to deploy .NET Core 3.0 desktop applications to Windows 10.

The Windows Application Packaging Project, available in Visual Studio 2019, allows you to create MSIX packages with self-contained .NET Core applications.

The .NET Core project file must specify the supported runtimes in the `<RuntimeIdentifiers>` property:

```
<RuntimeIdentifiers>win-x86;win-x64</RuntimeIdentifiers>
```

Linux improvements

SerialPort for Linux

.NET Core 3.0 provides basic support for System.IO.Ports.SerialPort on Linux.

Previously, .NET Core only supported using `SerialPort` on Windows.

For more information about the limited support for the serial port on Linux, see GitHub issue #33146.

Docker and cgroup memory Limits

Running .NET Core 3.0 on Linux with Docker works better with cgroup memory limits. Running a Docker container with memory limits, such as with `docker run -m`, changes how .NET Core behaves.

- Default Garbage Collector (GC) heap size: maximum of 20 mb or 75% of the memory limit on the container.
- Explicit size can be set as an absolute number or percentage of cgroup limit.
- Minimum reserved segment size per GC heap is 16 mb. This size reduces the number of heaps that are created on machines.

GPIO Support for Raspberry Pi

Two packages have been released to NuGet that you can use for GPIO programming:

- System.Device.Gpio
- Iot.Device.Bindings

The GPIO packages include APIs for *GPIO*, *SPI*, *I2C*, and *PWM* devices. The IoT bindings package includes device bindings. For more information, see the devices GitHub repo.

ARM64 Linux support

.NET Core 3.0 adds support for ARM64 for Linux. The primary use case for ARM64 is currently with IoT scenarios. For more information, see .NET Core ARM64 Status.

Docker images for .NET Core on ARM64 are available for Alpine, Debian, and Ubuntu.

NOTE

ARM64 Windows support isn't yet available.

Security

TLS 1.3 & OpenSSL 1.1.1 on Linux

.NET Core now takes advantage of TLS 1.3 support in OpenSSL 1.1.1, when it's available in a given environment. With TLS 1.3:

- Connection times are improved with reduced round trips required between the client and server.
- Improved security because of the removal of various obsolete and insecure cryptographic algorithms.

When available, .NET Core 3.0 uses **OpenSSL 1.1.1**, **OpenSSL 1.1.0**, or **OpenSSL 1.0.2** on a Linux system. When **OpenSSL 1.1.1** is available, both System.Net.Security.SslStream and System.Net.Http.HttpClient types will use **TLS 1.3** (assuming both the client and server support **TLS 1.3**).

IMPORTANT

Windows and macOS do not yet support **TLS 1.3**. .NET Core 3.0 will support **TLS 1.3** on these operating systems when support becomes available.

The following C# 8.0 example demonstrates .NET Core 3.0 on Ubuntu 18.10 connecting to https://www.cloudflare.com:

```
using System;
using System.Net.Security;
using System.Net.Sockets;
using System.Threading.Tasks;

namespace whats_new
{
    public static class TLS
    {
        public static async Task ConnectCloudFlare()
        {
            var targetHost = "www.cloudflare.com";

            using TcpClient tcpClient = new TcpClient();

            await tcpClient.ConnectAsync(targetHost, 443);

            using SslStream sslStream = new SslStream(tcpClient.GetStream());

            await sslStream.AuthenticateAsClientAsync(targetHost);
            await Console.Out.WriteLineAsync($"Connected to {targetHost} with {sslStream.SslProtocol}");
        }
    }
}
```

Cryptography ciphers

.NET 3.0 adds support for **AES-GCM** and **AES-CCM** ciphers, implemented with System.Security.Cryptography.AesGcm and System.Security.Cryptography.AesCcm respectively. These algorithms are both Authenticated Encryption with Association Data (AEAD) algorithms.

The following code demonstrates using `AesGcm` cipher to encrypt and decrypt random data.

```
using System;
using System.Linq;
using System.Security.Cryptography;

namespace whats_new
{
    public static class Cipher
    {
        public static void Run()
        {
            // key should be: pre-known, derived, or transported via another channel, such as RSA encryption
            byte[] key = new byte[16];
            RandomNumberGenerator.Fill(key);

            byte[] nonce = new byte[12];
            RandomNumberGenerator.Fill(nonce);

            // normally this would be your data
            byte[] dataToEncrypt = new byte[1234];
            byte[] associatedData = new byte[333];
            RandomNumberGenerator.Fill(dataToEncrypt);
            RandomNumberGenerator.Fill(associatedData);

            // these will be filled during the encryption
            byte[] tag = new byte[16];
            byte[] ciphertext = new byte[dataToEncrypt.Length];

            using (AesGcm aesGcm = new AesGcm(key))
            {
                aesGcm.Encrypt(nonce, dataToEncrypt, ciphertext, tag, associatedData);
            }

            // tag, nonce, ciphertext, associatedData should be sent to the other part

            byte[] decryptedData = new byte[ciphertext.Length];

            using (AesGcm aesGcm = new AesGcm(key))
            {
                aesGcm.Decrypt(nonce, ciphertext, tag, decryptedData, associatedData);
            }

            // do something with the data
            // this should always print that data is the same
            Console.WriteLine($"AES-GCM: Decrypted data is {(dataToEncrypt.SequenceEqual(decryptedData) ? "the
same as" : "different than")} original data.");
        }
    }
}
```

Cryptographic Key Import/Export

.NET Core 3.0 supports the import and export of asymmetric public and private keys from standard formats. You don't need to use an X.509 certificate.

All key types, such as *RSA*, *DSA*, *ECDsa*, and *ECDiffieHellman*, support the following formats:

- **Public Key**
  - X.509 SubjectPublicKeyInfo
- **Private key**
  - PKCS#8 PrivateKeyInfo
  - PKCS#8 EncryptedPrivateKeyInfo

RSA keys also support:

- **Public Key**
  - PKCS#1 RSAPublicKey
- **Private key**
  - PKCS#1 RSAPrivateKey

The export methods produce DER-encoded binary data, and the import methods expect the same. If a key is stored in the text-friendly PEM format, the caller will need to base64-decode the content before calling an import method.

```
using System;
using System.Security.Cryptography;

namespace whats_new
{
    public static class RSATest
    {
        public static void Run(string keyFile)
        {
            using var rsa = RSA.Create();

            byte[] keyBytes = System.IO.File.ReadAllBytes(keyFile);
            rsa.ImportRSAPrivateKey(keyBytes, out int bytesRead);

            Console.WriteLine($"Read {bytesRead} bytes, {keyBytes.Length - bytesRead} extra byte(s) in file.");
            RSAParameters rsaParameters = rsa.ExportParameters(true);
            Console.WriteLine(BitConverter.ToString(rsaParameters.D));
        }
    }
}
```

PKCS#8 files can be inspected with System.Security.Cryptography.Pkcs.Pkcs8PrivateKeyInfo and **PFX/PKCS#12** files can be inspected with System.Security.Cryptography.Pkcs.Pkcs12Info. **PFX/PKCS#12** files can be manipulated with System.Security.Cryptography.Pkcs.Pkcs12Builder.

.NET Core 3.0 API changes

Ranges and indices

The new System.Index type can be used for indexing. You can create one from an `int` that counts from the beginning, or with a prefix `^` operator (C#) that counts from the end:

```
Index i1 = 3;  // number 3 from beginning
Index i2 = ^4; // number 4 from end
int[] a = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
Console.WriteLine($"{a[i1]}, {a[i2]}"); // "3, 6"
```

There's also the System.Range type, which consists of two `Index` values, one for the start and one for the end, and can be written with a `x..y` range expression (C#). You can then index with a `Range`, which produces a slice:

```
var slice = a[i1..i2]; // { 3, 4, 5 }
```

For more information, see the ranges and indices tutorial.

Async streams

The IAsyncEnumerable<T> type is a new asynchronous version of IEnumerable<T>. The language lets you `await foreach` over `IAsyncEnumerable<T>` to consume their elements, and use `yield return` to them to produce elements.

The following example demonstrates both production and consumption of async streams. The `foreach` statement is async and itself uses `yield return` to produce an async stream for callers. This pattern (using `yield return`) is the recommended model for producing async streams.

```
async IAsyncEnumerable<int> GetBigResultsAsync()
{
    await foreach (var result in GetResultsAsync())
    {
        if (result > 20) yield return result;
    }
}
```

In addition to being able to `await foreach`, you can also create async iterators, for example, an iterator that returns an `IAsyncEnumerable/IAsyncEnumerator` that you can both `await` and `yield` in. For objects that need to be disposed, you can use `IAsyncDisposable`, which various BCL types implement, such as `Stream` and `Timer`.

For more information, see the async streams tutorial.

IEEE Floating-point

Floating point APIs are being updated to comply with IEEE 754-2008 revision. The goal of these changes is to expose all **required** operations and ensure that they're behaviorally compliant with the IEEE spec. For more information about floating-point improvements, see the Floating-Point Parsing and Formatting improvements in .NET Core 3.0 blog post.

Parsing and formatting fixes include:

- Correctly parse and round inputs of any length.
- Correctly parse and format negative zero.
- Correctly parse `Infinity` and `NaN` by doing a case-insensitive check and allowing an optional preceding `+` where applicable.

New System.Math APIs include:

- BitIncrement(Double) and BitDecrement(Double)
  Corresponds to the `nextUp` and `nextDown` IEEE operations. They return the smallest floating-point number that compares greater or lesser than the input (respectively). For example, `Math.BitIncrement(0.0)` would return `double.Epsilon`.

- MaxMagnitude(Double, Double) and MinMagnitude(Double, Double)
  Corresponds to the `maxNumMag` and `minNumMag` IEEE operations, they return the value that is greater or lesser in magnitude of the two inputs (respectively). For example, `Math.MaxMagnitude(2.0, -3.0)` would return `-3.0`.

- ILogB(Double)
  Corresponds to the `logB` IEEE operation that returns an integral value, it returns the integral base-2 log of the input parameter. This method is effectively the same as `floor(log2(x))`, but done with minimal rounding error.

- ScaleB(Double, Int32)
  Corresponds to the `scaleB` IEEE operation that takes an integral value, it returns effectively `x * pow(2, n)`, but is done with minimal rounding error.

- Log2(Double)
  Corresponds to the `log2` IEEE operation, it returns the base-2 logarithm. It minimizes rounding error.

- FusedMultiplyAdd(Double, Double, Double)
  Corresponds to the `fma` IEEE operation, it performs a fused multiply add. That is, it does `(x * y) + z` as a

single operation, thereby minimizing the rounding error. An example would be `FusedMultiplyAdd(1e308, 2.0, -1e308)` which returns `1e308`. The regular `(1e308 * 2.0) - 1e308` returns `double.PositiveInfinity`.

- CopySign(Double, Double)
  Corresponds to the `copySign` IEEE operation, it returns the value of `x`, but with the sign of `y`.

.NET Platform-Dependent Intrinsics

APIs have been added that allow access to certain perf-oriented CPU instructions, such as the **SIMD** or **Bit Manipulation instruction** sets. These instructions can help achieve significant performance improvements in certain scenarios, such as processing data efficiently in parallel.

Where appropriate, the .NET libraries have begun using these instructions to improve performance.

For more information, see .NET Platform Dependent Intrinsics.

Improved .NET Core Version APIs

Starting with .NET Core 3.0, the version APIs provided with .NET Core now return the information you expect. For example:

```
System.Console.WriteLine($"Environment.Version: {System.Environment.Version}");

// Old result
//   Environment.Version: 4.0.30319.42000
//
// New result
//   Environment.Version: 3.0.0
```

```
System.Console.WriteLine($"RuntimeInformation.FrameworkDescription:
{System.Runtime.InteropServices.RuntimeInformation.FrameworkDescription}");

// Old result
//   RuntimeInformation.FrameworkDescription: .NET Core 4.6.27415.71
//
// New result (notice the value includes any preview release information)
//   RuntimeInformation.FrameworkDescription: .NET Core 3.0.0-preview4-27615-11
```

WARNING

Breaking change. This is technically a breaking change because the versioning scheme has changed.

Fast built-in JSON support

.NET users have largely relied on **Json.NET** and other popular JSON libraries, which continue to be good choices. **Json.NET** uses .NET strings as its base datatype, which is UTF-16 under the hood.

The new built-in JSON support is high-performance, low allocation, and based on `Span<byte>`. For more information about the System.Text.Json namespace and types, see JSON serialization in .NET - overview. For tutorials on common JSON serialization scenarios, see How to serialize and deserialize JSON in .NET.

HTTP/2 support

The System.Net.Http.HttpClient type supports the HTTP/2 protocol. If HTTP/2 is enabled, the HTTP protocol version is negotiated via TLS/ALPN, and HTTP/2 is used if the server elects to use it.

The default protocol remains HTTP/1.1, but HTTP/2 can be enabled in two different ways. First, you can set the HTTP request message to use HTTP/2:

```
var client = new HttpClient() { BaseAddress = new Uri("https://localhost:5001") };

// HTTP/1.1 request
using (var response = await client.GetAsync("/"))
    Console.WriteLine(response.Content);

// HTTP/2 request
using (var request = new HttpRequestMessage(HttpMethod.Get, "/") { Version = new Version(2, 0) })
using (var response = await client.SendAsync(request))
    Console.WriteLine(response.Content);
```

Second, you can change HttpClient to use HTTP/2 by default:

```
var client = new HttpClient()
{
    BaseAddress = new Uri("https://localhost:5001"),
    DefaultRequestVersion = new Version(2, 0)
};

// HTTP/2 is default
using (var response = await client.GetAsync("/"))
    Console.WriteLine(response.Content);
```

Many times when you're developing an application, you want to use an unencrypted connection. If you know the target endpoint will be using HTTP/2, you can turn on unencrypted connections for HTTP/2. You can turn it on by setting the `DOTNET_SYSTEM_NET_HTTP_SOCKETSHTTPHANDLER_HTTP2UNENCRYPTEDSUPPORT` environment variable to `1` or by enabling it in the app context:

```
AppContext.SetSwitch("System.Net.Http.SocketsHttpHandler.Http2UnencryptedSupport", true);
```

Next steps

- Review the breaking changes between .NET Core 2.2 and 3.0.
- Review the breaking changes between .NET Framework and .NET Core 3.0.

What's new in .NET Core 2.2

10/30/2019 • 4 minutes to read • Edit Online

.NET Core 2.2 includes enhancements in application deployment, event handling for runtime services, authentication to Azure SQL databases, JIT compiler performance, and code injection prior to the execution of the `Main` method.

New deployment mode

Starting with .NET Core 2.2, you can deploy framework-dependent executables, which are **.exe** files instead of **.dll** files. Functionally similar to framework-dependent deployments, framework-dependent executables (FDE) still rely on the presence of a shared system-wide version of .NET Core to run. Your app contains only your code and any third-party dependencies. Unlike framework-dependent deployments, FDEs are platform-specific.

This new deployment mode has the distinct advantage of building an executable instead of a library, which means you can run your app directly without invoking `dotnet` first.

Core

Handling events in runtime services

You may often want to monitor your application's use of runtime services, such as the GC, JIT, and ThreadPool, to understand how they impact your application. On Windows systems, this is commonly done by monitoring the ETW events of the current process. While this continues to work well, it's not always possible to use ETW if you're running in a low-privilege environment or on Linux or macOS.

Starting with .NET Core 2.2, CoreCLR events can now be consumed using the System.Diagnostics.Tracing.EventListener class. These events describe the behavior of such runtime services as GC, JIT, ThreadPool, and interop. These are the same events that are exposed as part of the CoreCLR ETW provider. This allows for applications to consume these events or use a transport mechanism to send them to a telemetry aggregation service. You can see how to subscribe to events in the following code sample:

```
internal sealed class SimpleEventListener : EventListener
{
    // Called whenever an EventSource is created.
    protected override void OnEventSourceCreated(EventSource eventSource)
    {
        // Watch for the .NET runtime EventSource and enable all of its events.
        if (eventSource.Name.Equals("Microsoft-Windows-DotNETRuntime"))
        {
            EnableEvents(eventSource, EventLevel.Verbose, (EventKeywords)(-1));
        }
    }

    // Called whenever an event is written.
    protected override void OnEventWritten(EventWrittenEventArgs eventData)
    {
        // Write the contents of the event to the console.
        Console.WriteLine($"ThreadID = {eventData.OSThreadId} ID = {eventData.EventId} Name =
{eventData.EventName}");
        for (int i = 0; i < eventData.Payload.Count; i++)
        {
            string payloadString = eventData.Payload[i]?.ToString() ?? string.Empty;
            Console.WriteLine($"\tName = \"{eventData.PayloadNames[i]}\" Value = \"{payloadString}\"");
        }
        Console.WriteLine("\n");
    }
}
```

In addition, .NET Core 2.2 adds the following two properties to the EventWrittenEventArgs class to provide additional information about ETW events:

- EventWrittenEventArgs.OSThreadId
- EventWrittenEventArgs.TimeStamp

Data

AAD authentication to Azure SQL databases with the SqlConnection.AccessToken property

Starting with .NET Core 2.2, an access token issued by Azure Active Directory can be used to authenticate to an Azure SQL database. To support access tokens, the AccessToken property has been added to the SqlConnection class. To take advantage of AAD authentication, download version 4.6 of the System.Data.SqlClient NuGet package. In order to use the feature, you can obtain the access token value using the Active Directory Authentication Library for .NET contained in the `Microsoft.IdentityModel.Clients.ActiveDirectory` NuGet package.

JIT compiler improvements

Tiered compilation remains an opt-in feature

In .NET Core 2.1, the JIT compiler implemented a new compiler technology, *tiered compilation*, as an opt-in feature. The goal of tiered compilation is improved performance. One of the important tasks performed by the JIT compiler is optimizing code execution. For little-used code paths, however, the compiler may spend more time optimizing code than the runtime spends executing unoptimized code. Tiered compilation introduces two stages in JIT compilation:

- A **first tier**, which generates code as quickly as possible.
- A **second tier**, which generates optimized code for those methods that are executed frequently. The second tier of compilation is performed in parallel for enhanced performance.

For information on the performance improvement that can result from tiered compilation, see Announcing .NET

Core 2.2 Preview 2.

In .NET Core 2.2 Preview 2, tiered compilation was enabled by default. However, we've decided that we are still not ready to enable tiered compilation by default. So in .NET Core 2.2, tiered compilation continues to be an opt-in feature. For information on opting in to tiered compilation, see Jit compiler improvements in What's new in .NET Core 2.1.

Runtime

Injecting code prior to executing the Main method

Starting with .NET Core 2.2, you can use a startup hook to inject code prior to running an application's Main method. Startup hooks make it possible for a host to customize the behavior of applications after they have been deployed without needing to recompile or change the application.

We expect hosting providers to define custom configuration and policy, including settings that potentially influence the load behavior of the main entry point, such as the System.Runtime.Loader.AssemblyLoadContext behavior. The hook can be used to set up tracing or telemetry injection, to set up callbacks for handling, or to define other environment-dependent behavior. The hook is separate from the entry point, so that user code doesn't need to be modified.

See Host startup hook for more information.

See also

- What's new in .NET Core
- What's new in ASP.NET Core 2.2
- New features in EF Core 2.2

What's new in .NET Core 2.1

10/30/2019 • 10 minutes to read • Edit Online

.NET Core 2.1 includes enhancements and new features in the following areas:

- Tooling
- Roll forward
- Deployment
- Windows Compatibility Pack
- JIT compilation improvements
- API changes

Tooling

The .NET Core 2.1 SDK (v 2.1.300), the tooling included with .NET Core 2.1, includes the following changes and enhancements:

Build performance improvements

A major focus of .NET Core 2.1 is improving build-time performance, particularly for incremental builds. These performance improvements apply to both command-line builds using `dotnet build` and to builds in Visual Studio. Some individual areas of improvement include:

- For package asset resolution, resolving only assets used by a build rather than all assets.

- Caching of assembly references.

- Use of long-running SDK build servers, which are processes that span across individual `dotnet build` invocations. They eliminate the need to JIT-compile large blocks of code every time `dotnet build` is run. Build server processes can be automatically terminated with the following command:

  ```
  dotnet buildserver shutdown
  ```

New CLI commands

A number of tools that were available only on a per project basis using `DotnetCliToolReference` are now available as part of the .NET Core SDK. These tools include:

- `dotnet watch` provides a file system watcher that waits for a file to change before executing a designated set of commands. For example, the following command automatically rebuilds the current project and generates verbose output whenever a file in it changes:

  ```
  dotnet watch -- --verbose build
  ```

  Note the `--` option that precedes the `--verbose` option. It delimits the options passed directly to the `dotnet watch` command from the arguments that are passed to the child `dotnet` process. Without it, the `--verbose` option applies to the `dotnet watch` command, not the `dotnet build` command.

  For more information, see Develop ASP.NET Core apps using dotnet watch.

- `dotnet dev-certs` generates and manages certificates used during development in ASP.NET Core applications.

- `dotnet user-secrets` manages the secrets in a user secret store in ASP.NET Core applications.
- `dotnet sql-cache` creates a table and indexes in a Microsoft SQL Server database to be used for distributed caching.
- `dotnet ef` is a tool for managing databases, DbContext objects, and migrations in Entity Framework Core applications. For more information, see EF Core .NET Command-line Tools.

Global Tools

.NET Core 2.1 supports *Global Tools* -- that is, custom tools that are available globally from the command line. The extensibility model in previous versions of .NET Core made custom tools available on a per project basis only by using `DotnetCliToolReference`.

To install a Global Tool, you use the dotnet tool install command. For example:

```
dotnet tool install -g dotnetsay
```

Once installed, the tool can be run from the command line by specifying the tool name. For more information, see .NET Core Global Tools overview.

Tool management with the `dotnet tool` command

In .NET Core 2.1 SDK, all tools operations use the `dotnet tool` command. The following options are available:

- `dotnet tool install` to install a tool.
- `dotnet tool update` to uninstall and reinstall a tool, which effectively updates it.
- `dotnet tool list` to list currently installed tools.
- `dotnet tool uninstall` to uninstall currently installed tools.

Roll forward

All .NET Core applications starting with .NET Core 2.0 automatically roll forward to the latest *minor version* installed on a system.

Starting with .NET Core 2.0, if the version of .NET Core that an application was built with is not present at runtime, the application automatically runs against the latest installed *minor version* of .NET Core. In other words, if an application is built with .NET Core 2.0, and .NET Core 2.0 is not present on the host system but .NET Core 2.1 is, the application runs with .NET Core 2.1.

IMPORTANT

This roll-forward behavior doesn't apply to preview releases. By default, it also doesn't apply to major releases, but this can be changed with the settings below.

You can modify this behavior by changing the setting for the roll-forward on no candidate shared framework. The available settings are:

- `0` - disable minor version roll-forward behavior. With this setting, an application built for .NET Core 2.0.0 will roll forward to .NET Core 2.0.1, but not to .NET Core 2.2.0 or .NET Core 3.0.0.
- `1` - enable minor version roll-forward behavior. This is the default value for the setting. With this setting, an application built for .NET Core 2.0.0 will roll forward to either .NET Core 2.0.1 or .NET Core 2.2.0, depending on which one is installed, but it will not roll forward to .NET Core 3.0.0.
- `2` - enable minor and major version roll-forward behavior. If set, even different major versions are considered,

so an application built for .NET Core 2.0.0 will roll forward to .NET Core 3.0.0.

You can modify this setting in any of three ways:

- Set the `DOTNET_ROLL_FORWARD_ON_NO_CANDIDATE_FX` environment variable to the desired value.
- Add the following line with the desired value to the *.runtimeconfig.json* file:

  ```
  "rollForwardOnNoCandidateFx" : 0
  ```

- When using .NET Core CLI tools, add the following option with the desired value to a .NET Core command such as `run`:

  ```
  dotnet run --rollForwardOnNoCandidateFx=0
  ```

Patch version roll forward is independent of this setting and is done after any potential minor or major version roll forward is applied.

Deployment

Self-contained application servicing

`dotnet publish` now publishes self-contained applications with a serviced runtime version. When you publish a self-contained application with the .NET Core 2.1 SDK (v 2.1.300), your application includes the latest serviced runtime version known by that SDK. When you upgrade to the latest SDK, you'll publish with the latest .NET Core runtime version. This applies for .NET Core 1.0 runtimes and later.

Self-contained publishing relies on runtime versions on NuGet.org. You do not need to have the serviced runtime on your machine.

Using the .NET Core 2.0 SDK, self-contained applications are published with the .NET Core 2.0.0 runtime unless a different version is specified via the `RuntimeFrameworkVersion` property. With this new behavior, you'll no longer need to set this property to select a higher runtime version for a self-contained application. The easiest approach going forward is to always publish with .NET Core 2.1 SDK (v 2.1.300).

For more information, see Self-contained deployment runtime roll forward.

Windows Compatibility Pack

When you port existing code from the .NET Framework to .NET Core, you can use the Windows Compatibility Pack. It provides access to 20,000 more APIs than are available in .NET Core. These APIs include types in the System.Drawing namespace, the EventLog class, WMI, Performance Counters, Windows Services, and the Windows registry types and members.

JIT compiler improvements

.NET Core incorporates a new JIT compiler technology called *tiered compilation* (also known as *adaptive optimization*) that can significantly improve performance. Tiered compilation is an opt-in setting.

One of the important tasks performed by the JIT compiler is optimizing code execution. For little-used code paths, however, the compiler may spend more time optimizing code than the runtime spends running unoptimized code. Tiered compilation introduces two stages in JIT compilation:

- A **first tier**, which generates code as quickly as possible.
- A **second tier**, which generates optimized code for those methods that are executed frequently. The second

tier of compilation is performed in parallel for enhanced performance.

You can opt into tiered compilation in either of two ways.

- To use tiered compilation in all projects that use the .NET Core 2.1 SDK, set the following environment variable:

  ```
  COMPlus_TieredCompilation="1"
  ```

- To use tiered compilation on a per-project basis, add the `<TieredCompilation>` property to the `<PropertyGroup>` section of the MSBuild project file, as the following example shows:

  ```
  <PropertyGroup>
      <!-- other property definitions -->

      <TieredCompilation>true</TieredCompilation>
  </PropertyGroup>
  ```

API changes

`Span<T>` and `Memory<T>`

.NET Core 2.1 includes some new types that make working with arrays and other types of memory much more efficient. The new types include:

- System.Span<T> and System.ReadOnlySpan<T>.
- System.Memory<T> and System.ReadOnlyMemory<T>.

Without these types, when passing such items as a portion of an array or a section of a memory buffer, you have to make a copy of some portion of the data before passing it to a method. These types provide a virtual view of that data that eliminates the need for the additional memory allocation and copy operations.

The following example uses a Span<T> and Memory<T> instance to provide a virtual view of 10 elements of an array.

```
using System;

class Program
{
    static void Main()
    {
        int[] numbers = new int[100];
        for (int i = 0; i < 100; i++)
        {
            numbers[i] = i * 2;
        }

        var part = new Span<int>(numbers, start: 10, length: 10);
        foreach (var value in part)
            Console.Write($"{value}  ");
    }
}
// The example displays the following output:
//     20  22  24  26  28  30  32  34  36  38
```

```
Module Program
    Sub Main()
        Dim numbers As Integer() = New Integer(99) {}

        For i As Integer = 0 To 99
            numbers(i) = i * 2
        Next

        Dim part = New Memory(Of Integer)(numbers, start:=10, length:=10)

        For Each value In part.Span
            Console.Write($"{value}  ")
        Next
    End Sub
End Module
' The example displays the following output:
'     20  22  24  26  28  30  32  34  36  38
```

Brotli compression

.NET Core 2.1 adds support for Brotli compression and decompression. Brotli is a general-purpose lossless compression algorithm that is defined in RFC 7932 and is supported by most web browsers and major web servers. You can use the stream-based System.IO.Compression.BrotliStream class or the high-performance span-based System.IO.Compression.BrotliEncoder and System.IO.Compression.BrotliDecoder classes. The following example illustrates compression with the BrotliStream class:

```
public static Stream DecompressWithBrotli(Stream toDecompress)
{
    MemoryStream decompressedStream = new MemoryStream();
    using (BrotliStream decompressionStream = new BrotliStream(toDecompress, CompressionMode.Decompress))
    {
        decompressionStream.CopyTo(decompressedStream);
    }
    decompressedStream.Position = 0;
    return decompressedStream;
}
```

```
Public Function DecompressWithBrotli(toDecompress As Stream) As Stream
    Dim decompressedStream As New MemoryStream()
    Using decompressionStream As New BrotliStream(toDecompress, CompressionMode.Decompress)
        decompressionStream.CopyTo(decompressedStream)
    End Using
    decompressedStream.Position = 0
    Return decompressedStream
End Function
```

The BrotliStream behavior is the same as DeflateStream and GZipStream, which makes it easy to convert code that calls these APIs to BrotliStream.

New cryptography APIs and cryptography improvements

.NET Core 2.1 includes numerous enhancements to the cryptography APIs:

- System.Security.Cryptography.Pkcs.SignedCms is available in the System.Security.Cryptography.Pkcs package. The implementation is the same as the SignedCms class in the .NET Framework.
- New overloads of the X509Certificate.GetCertHash and X509Certificate.GetCertHashString methods accept a hash algorithm identifier to enable callers to get certificate thumbprint values using algorithms other than SHA-1.
- New Span<T>-based cryptography APIs are available for hashing, HMAC, cryptographic random number

generation, asymmetric signature generation, asymmetric signature processing, and RSA encryption.

- The performance of System.Security.Cryptography.Rfc2898DeriveBytes has improved by about 15% by using a Span<T>-based implementation.
- The new System.Security.Cryptography.CryptographicOperations class includes two new methods:
  - FixedTimeEquals takes a fixed amount of time to return for any two inputs of the same length, which makes it suitable for use in cryptographic verification to avoid contributing to timing side-channel information.
  - ZeroMemory is a memory-clearing routine that cannot be optimized.
- The static RandomNumberGenerator.Fill method fills a Span<T> with random values.
- The System.Security.Cryptography.Pkcs.EnvelopedCms is now supported on Linux and macOS.
- Elliptic-Curve Diffie-Hellman (ECDH) is now available in the System.Security.Cryptography.ECDiffieHellman class family. The surface area is the same as in the .NET Framework.
- The instance returned by RSA.Create can encrypt or decrypt with OAEP using a SHA-2 digest, as well as generate or validate signatures using RSA-PSS.

Sockets improvements

.NET Core includes a new type, System.Net.Http.SocketsHttpHandler, and a rewritten System.Net.Http.HttpMessageHandler, that form the basis of higher-level networking APIs. System.Net.Http.SocketsHttpHandler, for example, is the basis of the HttpClient implementation. In previous versions of .NET Core, higher-level APIs were based on native networking implementations.

The sockets implementation introduced in .NET Core 2.1 has a number of advantages:

- A significant performance improvement when compared with the previous implementation.
- Elimination of platform dependencies, which simplifies deployment and servicing.
- Consistent behavior across all .NET Core platforms.

SocketsHttpHandler is the default implementation in .NET Core 2.1. However, you can configure your application to use the older HttpClientHandler class by calling the AppContext.SetSwitch method:

```
AppContext.SetSwitch("System.Net.Http.UseSocketsHttpHandler", false);
```

```
AppContext.SetSwitch("System.Net.Http.UseSocketsHttpHandler", False)
```

You can also use an environment variable to opt out of using sockets implementations based on SocketsHttpHandler. To do this, set the `DOTNET_SYSTEM_NET_HTTP_USESOCKETSHTTPHANDLER` to either `false` or 0.

On Windows, you can also choose to use System.Net.Http.WinHttpHandler, which relies on a native implementation, or the SocketsHttpHandler class by passing an instance of the class to the HttpClient constructor.

On Linux and macOS, you can only configure HttpClient on a per-process basis. On Linux, you need to deploy libcurl if you want to use the old HttpClient implementation. (It is installed with .NET Core 2.0.)

See also

- What's new in .NET Core
- New features in EF Core 2.1

- What's new in ASP.NET Core 2.1

What's new in .NET Core 2.0

10/30/2019 • 6 minutes to read • Edit Online

.NET Core 2.0 includes enhancements and new features in the following areas:

- Tooling
- Language support
- Platform improvements
- API changes
- Visual Studio integration
- Documentation improvements

Tooling

dotnet restore runs implicitly

In previous versions of .NET Core, you had to run the dotnet restore command to download dependencies immediately after you created a new project with the dotnet new command, as well as whenever you added a new dependency to your project.

NOTE

Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

You can also disable the automatic invocation of `dotnet restore` by passing the `--no-restore` switch to the `new`, `run`, `build`, `publish`, `pack`, and `test` commands.

Retargeting to .NET Core 2.0

If the .NET Core 2.0 SDK is installed, projects that target .NET Core 1.x can be retargeted to .NET Core 2.0.

To retarget to .NET Core 2.0, edit your project file by changing the value of the `<TargetFramework>` element (or the `<TargetFrameworks>` element if you have more than one target in your project file) from 1.x to 2.0:

```
<PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
 </PropertyGroup>
```

You can also retarget .NET Standard libraries to .NET Standard 2.0 the same way:

```
<PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
 </PropertyGroup>
```

For more information about migrating your project to .NET Core 2.0, see Migrating from ASP.NET Core 1.x to ASP.NET Core 2.0.

Language support

In addition to supporting C# and F#, .NET Core 2.0 also supports Visual Basic.

Visual Basic

With version 2.0, .NET Core now supports Visual Basic 2017. You can use Visual Basic to create the following project types:

- .NET Core console apps
- .NET Core class libraries
- .NET Standard class libraries
- .NET Core unit test projects
- .NET Core xUnit test projects

For example, to create a Visual Basic "Hello World" application, do the following steps from the command line:

1. Open a console window, create a directory for your project, and make it the current directory.
2. Enter the command `dotnet new console -lang vb`.

   The command creates a project file with a `.vbproj` file extension, along with a Visual Basic source code file named *Program.vb*. This file contains the source code to write the string "Hello World!" to the console window.
3. Enter the command `dotnet run`. The .NET Core CLI automatically compiles and executes the application, which displays the message "Hello World!" in the console window.

Support for C# 7.1

.NET Core 2.0 supports C# 7.1, which adds a number of new features, including:

- The `Main` method, the application entry point, can be marked with the async keyword.
- Inferred tuple names.
- Default expressions.

Platform improvements

.NET Core 2.0 includes a number of features that make it easier to install .NET Core and to use it on supported operating systems.

.NET Core for Linux is a single implementation

.NET Core 2.0 offers a single Linux implementation that works on multiple Linux distributions. .NET Core 1.x required that you download a distribution-specific Linux implementation.

You can also develop apps that target Linux as a single operating system. .NET Core 1.x required that you target each Linux distribution separately.

Support for the Apple cryptographic libraries

.NET Core 1.x on macOS required the OpenSSL toolkit's cryptographic library. .NET Core 2.0 uses the Apple cryptographic libraries and doesn't require OpenSSL, so you no longer need to install it.

API changes and library support

Support for .NET Standard 2.0

The .NET Standard defines a versioned set of APIs that must be available on .NET implementations that comply with that version of the standard. The .NET Standard is targeted at library developers. It aims to guarantee the functionality that is available to a library that targets a version of the .NET Standard on each .NET implementation.

.NET Core 1.x supports the .NET Standard version 1.6; .NET Core 2.0 supports the latest version, .NET Standard 2.0. For more information, see .NET Standard.

.NET Standard 2.0 includes over 20,000 more APIs than were available in the .NET Standard 1.6. Much of this expanded surface area results from incorporating APIs that are common to the .NET Framework and Xamarin into .NET Standard.

.NET Standard 2.0 class libraries can also reference .NET Framework class libraries, provided that they call APIs that are present in the .NET Standard 2.0. No recompilation of the .NET Framework libraries is required.

For a list of the APIs that have been added to the .NET Standard since its last version, the .NET Standard 1.6, see .NET Standard 2.0 vs. 1.6.

Expanded surface area

The total number of APIs available on .NET Core 2.0 has more than doubled in comparison with .NET Core 1.1.

And with the Windows Compatibility Pack porting from .NET Framework has also become much simpler.

Support for .NET Framework libraries

.NET Core code can reference existing .NET Framework libraries, including existing NuGet packages. Note that the libraries must use APIs that are found in .NET Standard.

Visual Studio integration

Visual Studio 2017 version 15.3 and in some cases Visual Studio for Mac offer a number of significant enhancements for .NET Core developers.

Retargeting .NET Core apps and .NET Standard libraries

If the .NET Core 2.0 SDK is installed, you can retarget .NET Core 1.x projects to .NET Core 2.0 and .NET Standard 1.x libraries to .NET Standard 2.0.

To retarget your project in Visual Studio, you open the **Application** tab of the project's properties dialog and change the **Target framework** value to **.NET Core 2.0** or **.NET Standard 2.0**. You can also change it by right-clicking on the project and selecting the **Edit *.csproj file** option. For more information, see the Tooling section earlier in this topic.

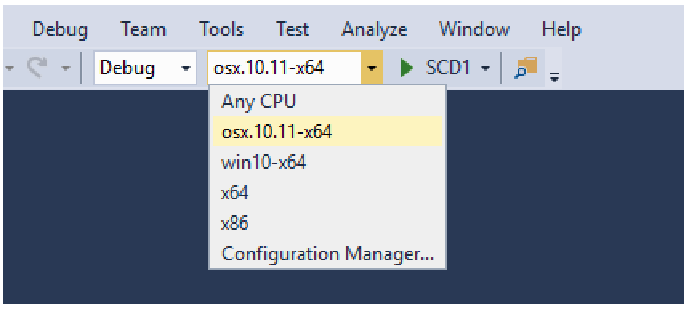
Live Unit Testing support for .NET Core

Whenever you modify your code, Live Unit Testing automatically runs any affected unit tests in the background and displays the results and code coverage live in the Visual Studio environment. .NET Core 2.0 now supports Live Unit Testing. Previously, Live Unit Testing was available only for .NET Framework applications.

For more information, see Live Unit Testing with Visual Studio 2017 and the Live Unit Testing FAQ.

Better support for multiple target frameworks

If you're building a project for multiple target frameworks, you can now select the target platform from the top-level menu. In the following figure, a project named SCD1 targets 64-bit macOS X 10.11 (`osx.10.11-x64`) and 64-bit Windows 10/Windows Server 2016 (`win10-x64`). You can select the target framework before selecting the project button, in this case to run a debug build.



Side-by-side support for .NET Core SDKs

You can now install the .NET Core SDK independently of Visual Studio. This makes it possible for a single version of Visual Studio to build projects that target different versions of .NET Core. Previously, Visual Studio and the .NET Core SDK were tightly coupled; a particular version of the SDK accompanied a particular version of Visual Studio.

Documentation improvements

.NET Application Architecture

.NET Application Architecture gives you access to a set of e-books that provide guidance, best practices, and sample applications when using .NET to build:

- Microservices and Docker containers
- Web applications with ASP.NET
- Mobile applications with Xamarin
- Applications that are deployed to the Cloud with Azure

See also

- What's new in ASP.NET Core 2.0

Evaluate breaking changes in .NET Core

11/12/2019 • 13 minutes to read • Edit Online

Throughout its history, .NET has attempted to maintain a high level of compatibility from version to version and across flavors of .NET. This continues to be true for .NET Core. Although .NET Core can be considered as a new technology that is independent of the .NET Framework, two major factors limit the ability of .NET Core to diverge from .NET Framework:

- A large number of developers either originally developed or continue to develop .NET Framework applications. They expect consistent behavior across .NET implementations.
- .NET Standard library projects allow developers to create libraries that target common APIs shared by .NET Core and .NET Framework. Developers expect that a library used in a .NET Core application should behave identically to the same library used in a .NET Framework application.

Along with compatibility across .NET implementations, developers expect a high level of compatibility across .NET Core versions. In particular, code written for an earlier version of .NET Core should run seamlessly on a later version of .NET Core. In fact, many developers expect that the new APIs found in newly released versions of .NET Core should also be compatible with the pre-release versions in which those APIs were introduced.

This article outlines the categories of compatibility changes (or breaking changes) and the way in which the .NET team evaluates changes in each of these categories. Understanding how the .NET team approaches possible breaking changes is particularly helpful for developers who are opening pull requests in the dotnet/corefx GitHub repository that modify the behavior of existing APIs.

> **NOTE**
> For a definition of compatibility categories, such as binary compatibility and backward compatibility, see Breaking change categories.

The following sections describes the categories of changes made to .NET Core APIs and their impact on application compatibility. The ✔️ icon indicates that a particular kind of change is allowed, ❌ indicates that it is disallowed, and ❓ indicates a change that may or may not be allowed. Changes in this last category require judgment and an evaluation of how predictable, obvious, and consistent the previous behavior was.

> **NOTE**
> In addition to serving as a guide to how changes to .NET Core libraries are evaluated, library developers can also use these criteria to evaluate changes to their libraries that target multiple .NET implementations and versions.

Modifications to the public contract

Changes in this category *modify* the public surface area of a type. Most of the changes in this category are disallowed since they violate *backwards compatibility* (the ability of an application that was developed with a previous version of an API to execute without recompilation on a later version).

Types

- **✔️ Removing an interface implementation from a type when the interface is already implemented by a base type**

- **□ Adding a new interface implementation to a type**

  This is an acceptable change because it does not adversely affect existing clients. Any changes to the type must work within the boundaries of acceptable changes defined here for the new implementation to remain acceptable. Extreme caution is necessary when adding interfaces that directly affect the ability of a designer or serializer to generate code or data that cannot be consumed down-level. An example is the ISerializable interface.

- **□ Introducing a new base class**

  A type can be introduced into an hierarchy between two existing types if it doesn't introduce any new abstract members or change the semantics or behavior of existing types. For example, in .NET Framework 2.0, the DbConnection class became a new base class for SqlConnection, which had previously derived directly from Component.

- **✔ □ Moving a type from one assembly to another**

  Note that the *old* assembly must be marked with the TypeForwardedToAttribute that points to the new assembly.

- **✔ □ Changing a struct type to a `readonly struct` type**

  Note that changing a `readonly struct` type to a `struct` type is not allowed.

- **✔ □ Adding the sealed or abstract keyword to a type when there are no *accessible* (public or protected) constructors**

- **✔ □ Expanding the visibility of a type**

- **□ Changing the namespace or name of a type**

- **□ Renaming or removing a public type**

  This breaks all code that uses the renamed or removed type.

- **□ Changing the underlying type of an enumeration**

  This is a compile-time and behavioral breaking change as well as a binary breaking change that can make attribute arguments unparsable.

- **□ Sealing a type that was previously unsealed**

- **□ Adding an interface to the set of base types of an interface**

  If an interface implements an interface that it previously did not implement, all types that implemented the original version of the interface are broken.

- **□ Removing a class from the set of base classes or an interface from the set of implemented interfaces**

  There is one exception to the rule for interface removal: you can add the implementation of an interface that derives from the removed interface. For example, you can remove IDisposable if the type or interface now implements IComponent, which implements IDisposable.

- **□ Changing a `readonly struct` type to a struct type**

  Note that the change of a `struct` type to a `readonly struct` type is allowed.

- **□ Changing a struct type to a `ref struct` type, and vice versa**

- **□ Reducing the visibility of a type**

However, increasing the visibility of a type is allowed.

Members

- **✔ ▯ Expanding the visibility of a member that is not virtual**
- **✔ ▯ Adding an abstract member to a public type that has no *accessible* (public or protected) constructors, or the type is sealed**

  However, adding an abstract member to a type that has accessible (public or protected) constructors and is not `sealed` is not allowed.

- **✔ ▯ Restricting the visibility of a protected member when the type has no accessible (public or protected) constructors, or the type is sealed**
- **✔ ▯ Moving a member into a class higher in the hierarchy than the type from which it was removed**
- **✔ ▯ Adding or removing an override**

  Note that introducing an override might cause previous consumers to skip over the override when calling base.

- **✔ ▯ Adding a constructor to a class, along with a parameterless constructor if the class previously had no constructors**

  However, adding a constructor to a class that previously had no constructors *without* adding the parameterless constructor is not allowed.

- **✔ ▯ Changing a member from abstract to virtual**
- **✔ ▯ Changing from a `ref readonly` to a `ref` return value (except for virtual methods or interfaces)**
- **✔ ▯ Removing readonly from a field, unless the static type of the field is a mutable value type**
- **✔ ▯ Calling a new event that wasn't previously defined**
- **▯ Adding a new instance field to a type**

  This change impacts serialization.

- **▯ Renaming or removing a public member or parameter**

  This breaks all code that uses the renamed or removed member, or parameter.

  Note that this includes removing or renaming a getter or setter from a property, as well as renaming or removing enumeration members.

- **▯ Adding a member to an interface**
- **▯ Changing the value of a public constant or enumeration member**
- **▯ Changing the type of a property, field, parameter, or return value**
- **▯ Adding, removing, or changing the order of parameters**
- **▯ Adding or removing the in, out , or ref keyword from a parameter**
- **▯ Renaming a parameter (including changing its case)**

  This is considered breaking for two reasons:

  - It breaks late-bound scenarios such as the late binding feature in Visual Basic and dynamic in C#.

- It breaks source compatibility when developers use named arguments.

- □ **Changing from a `ref` return value to a `ref readonly` return value**
- □□ **Changing from a `ref readonly` to a `ref` return value on a virtual method or interface**
- □ **Adding or removing abstract from a member**
- □ **Removing the virtual keyword from a member**

  While this often is not a breaking change because the C# compiler tends to emit callvirt Intermediate Language (IL) instructions to call non-virtual methods (`callvirt` performs a null check, while a normal call doesn't), this behavior is not invariable for several reasons:

  - C# is not the only language that .NET targets.
  - The C# compiler increasingly tries to optimize `callvirt` to a normal call whenever the target method is non-virtual and is probably not null (such as a method accessed through the ?. null propagation operator).

  Making a method virtual means that the consumer code would often end up calling it non-virtually.

- □ **Adding the virtual keyword to a member**
- □ **Making a virtual member abstract**

  A virtual member provides a method implementation that *can be* overridden by a derived class. An abstract member provides no implementation and *must be* overridden.

- □ **Adding an abstract member to a public type that has accessible (public or protected) constructors and that is not sealed**
- □ **Adding or removing the static keyword from a member**
- □ **Adding an overload that precludes an existing overload and defines a different behavior**

  This breaks existing clients that were bound to the previous overload. For example, if a class has a single version of a method that accepts a UInt32, an existing consumer will successfully bind to that overload when passing a Int32 value. However, if you add an overload that accepts an Int32, when recompiling or using late-binding, the compiler now binds to the new overload. If different behavior results, this is a breaking change.

- □ **Adding a constructor to a class that previously had no constructor without adding the parameterless constructor**
- □□ **Adding readonly to a field**
- □ **Reducing the visibility of a member**

  This includes reducing the visibility of a protected member when there are *accessible* (public or protected) constructors and the type is *not* sealed. If this is not the case, reducing the visibility of a protected member is allowed.

  Note that increasing the visibility of a member is allowed.

- □ **Changing the type of a member**

  The return value of a method or the type of a property or field cannot be modified. For example, the signature of a method that returns an Object cannot be changed to return a String, or vice versa.

- □ **Adding a field to a struct that previously had no state**

Definite assignment rules allow the use of uninitialized variables so long as the variable type is a stateless struct. If the struct is made stateful, code could end up with uninitialized data. This is both potentially a source breaking and a binary breaking change.

- ❌ **Firing an existing event when it was never fired before**

Behavioral changes

Assemblies

- ✔️ **Making an assembly portable when the same platforms are still supported**
- ❌ **Changing the name of an assembly**
- ❌ **Changing the public key of an assembly**

Properties, fields, parameters, and return values

- ✔️ **Changing the value of a property, field, return value, or out parameter to a more derived type**

  For example, a method that returns a type of Object can return a String instance. (However, the method signature cannot change.)

- ✔️ **Increasing the range of accepted values for a property or parameter if the member is not virtual**

  Note that while the range of values that can be passed to the method or are returned by the member can expand, the parameter or member type cannot. For example, while the values passed to a method can expand from 0-124 to 0-255, the parameter type cannot change from Byte to Int32.

- ❌ **Increasing the range of accepted values for a property or parameter if the member is virtual**

  This change breaks existing overridden members, which will not function correctly for the extended range of values.

- ❌ **Decreasing the range of accepted values for a property or parameter**
- ❌ **Increasing the range of returned values for a property, field, return value, or out parameter**
- ❌ **Changing the returned values for a property, field, method return value, or out parameter**
- ❌ **Changing the default value of a property, field, or parameter**
- ❌ **Changing the precision of a numeric return value**
- ❌ **A change in the parsing of input and throwing new exceptions (even if parsing behavior is not specified in the documentation**

Exceptions

- ✔️ **Throwing a more derived exception than an existing exception**

  Because the new exception is a subclass of an existing exception, previous exception handling code continues to handle the exception. For example, in .NET Framework 4, culture creation and retrieval methods began to throw a CultureNotFoundException instead of an ArgumentException if the culture could not be found. Because CultureNotFoundException derives from ArgumentException, this is an acceptable change.

- ✔️ **Throwing a more specific exception than NotSupportedException, NotImplementedException, NullReferenceException**
- ✔️ **Throwing an exception that is considered unrecoverable**

Unrecoverable exceptions should not be caught but instead should be handled by a high-level catch-all handler. Therefore, users are not expected to have code that catches these explicit exceptions. The unrecoverable exceptions are:

  - AccessViolationException
  - ExecutionEngineException
  - SEHException
  - StackOverflowException

- **✔️ Throwing a new exception in a new code path**

  The exception must apply only to a new code-path which is executed with new parameter values or state, and that can't be executed by existing code that targets the previous version.

- **✔️ Removing an exception to enable more robust behavior or new scenarios**

  For example, a `Divide` method that previously only handled positive values and threw an ArgumentOutOfRangeException otherwise can be changed to support both negative and positive values without throwing an exception.

- **✔️ Changing the text of an error message**

  Developers should not rely on the text of error messages, which also change based on the user's culture.

- **❌ Throwing an exception in any other case not listed above**

- **❌ Removing an exception in any other case not listed above**

Attributes

- **✔️ Changing the value of an attribute that is *not* observable**

- **❌ Changing the value of an attribute that *is* observable**

- **❌ Removing an attribute**

  In most cases, removing an attribute (such as NonSerializedAttribute) is a breaking change.

Platform support

- **✔️ Supporting an operation on a platform that was previously not supported**

- **❌ Not supporting or now requiring a specific service pack for an operation that was previously supported on a platform**

Internal implementation changes

- **❓ Changing the surface area of an internal type**

  Such changes are generally allowed, although they break private reflection. In some cases, where popular third-party libraries or a large number of developers depend on the internal APIs, such changes may not be allowed.

- **❓ Changing the internal implementation of a member**

  These changes are generally allowed, although they break private reflection. In some cases, where customer code frequently depends on private reflection or where the change introduces unintended side effects, these changes may not be allowed.

- **✔️ Improving the performance of an operation**

The ability to modify the performance of an operation is essential, but such changes can break code that relies upon the current speed of an operation. This is particularly true of code that depends on the timing of asynchronous operations. Note that the performance change should have no effect on other behavior of the API in question; otherwise, the change will be breaking.

- **✔ □ Indirectly (and often adversely) changing the performance of an operation**

  If the change in question is not categorized as breaking for some other reason, this is acceptable. Often, actions need to be taken that may include extra operations or that add new functionality. This will almost always affect performance but may be essential to make the API in question function as expected.

- **□ Changing a synchronous API to asynchronous (and vice versa)**

Code changes

- **✔ □ Adding params to a parameter**
- **□ Changing a struct to a class and vice versa**
- **□ Adding the checked keyword to a code block**

  This change may cause code that previously executed to throw an OverflowException and is unacceptable.

- **□ Removing params from a parameter**
- **□ Changing the order in which events are fired**

  Developers can reasonably expect events to fire in the same order, and developer code frequently depends on the order in which events are fired.

- **□ Removing the raising of an event on a given action**
- **□ Changing the number of times given events are called**
- **□ Adding the FlagsAttribute to an enumeration type**

Learn .NET Core and the .NET Core SDK tools by exploring these Tutorials

11/1/2019 • 2 minutes to read • Edit Online

The following tutorials are available for learning about .NET Core.

Building applications with Visual Studio 2017

- Building a C# Hello World application
- Debugging your C# Hello World application
- Publishing your C# Hello World application
- Building a C# class library
- Building a class library with Visual Basic
- Testing a class library
- Consuming a class library
- Azure Cosmos DB: Get started with the SQL API and .NET Core

Building applications with Visual Studio Code

- Get started with C# and Visual Studio Code
- Get started with .NET Core on macOS

Building applications with Visual Studio for Mac

- Get started with .NET Core on macOS using Visual Studio for Mac
- Building a complete .NET Core solution on macOS using Visual Studio for Mac

Building applications with the .NET Core CLI tools

- Get started with .NET Core on Windows/Linux/macOS using the .NET Core CLI tools
- Organizing and testing projects with the .NET Core CLI tools
- Get started with F#

Other

- Unit Testing in .NET Core using dotnet test
- Unit testing with MSTest and .NET Core
- Developing Libraries with Cross Platform Tools
- Hosting .NET Core from native code
- Create a custom template for dotnet new

For tutorials about developing ASP.NET Core web applications, see the ASP.NET Core documentation.

Tutorial: Create an item template

9/12/2019 • 5 minutes to read • Edit Online

With .NET Core, you can create and deploy templates that generate projects, files, even resources. This tutorial is part one of a series that teaches you how to create, install, and uninstall, templates for use with the `dotnet new` command.

In this part of the series, you'll learn how to:

- Create a class for an item template
- Create the template config folder and file
- Install a template from a file path
- Test an item template
- Uninstall an item template

Prerequisites

- .NET Core 2.2 SDK or later versions.
- Read the reference article Custom templates for dotnet new.

  The reference article explains the basics about templates and how they're put together. Some of this information will be reiterated here.
- Open a terminal and navigate to the *working\templates\* folder.

Create the required folders

This series uses a "working folder" where your template source is contained and a "testing folder" used to test your templates. The working folder and testing folder should be under the same parent folder.

First, create the parent folder, the name does not matter. Then, create a subfolder named*working*. Inside of the *working* folder, create a subfolder named *templates*.

Next, create a folder under the parent folder named *test*. The folder structure should look like the following:

```
parent_folder
├───test
└───working
    └───templates
```

Create an item template

An item template is a specific type of template that contains one or more files. These types of templates are useful when you want to generate something like a config, code, or solution file. In this example, you'll create a class that adds an extension method to the string type.

In your terminal, navigate to the *working\templates\* folder and create a new subfolder named *extensions*. Enter the folder.

```
working
└───templates
    └───extensions
```

Create a new file named *CommonExtensions.cs* and open it with your favorite text editor. This class will provide an extension method named `Reverse` that reverses the contents of a string. Paste in the following code and save the file:

```
using System;

namespace System
{
    public static class StringExtensions
    {
        public static string Reverse(this string value)
        {
            var tempArray = value.ToCharArray();
            Array.Reverse(tempArray);
            return new string(tempArray);
        }
    }
}
```

Now that you have the content of the template created, you need to create the template config at the root folder of the template.

Create the template config

Templates are recognized in .NET Core by a special folder and config file that exist at the root of your template. In this tutorial, your template folder is located at *working\templates\extensions\*.

When you create a template, all files and folders in the template folder are included as part of the template except for the special config folder. This config folder is named *.template.config*.

First, create a new subfolder named *.template.config*, enter it. Then, create a new file named *template.json*. Your folder structure should look like this:

```
working
└───templates
    └───extensions
        └───.template.config
                template.json
```

Open the *template.json* with your favorite text editor and paste in the following JSON code and save it:

```
{
  "$schema": "http://json.schemastore.org/template",
  "author": "Me",
  "classifications": [ "Common", "Code" ],
  "identity": "ExampleTemplate.StringExtensions",
  "name": "Example templates: string extensions",
  "shortName": "stringext",
  "tags": {
    "language": "C#",
    "type": "item"
  }
}
```

This config file contains all the settings for your template. You can see the basic settings, such as `name` and `shortName`, but there's also a `tags/type` value that is set to `item`. This categorizes your template as an item template. There's no restriction on the type of template you create. The `item` and `project` values are common names that .NET Core recommends so that users can easily filter the type of template they're searching for.

The `classifications` item represents the **tags** column you see when you run `dotnet new` and get a list of templates. Users can also search based on classification tags. Don't confuse the `tags` property in the *.json file with the `classifications` tags list. They're two different things unfortunately named similarly. The full schema for the *template.json* file is found at the JSON Schema Store. For more information about the *template.json* file, see the dotnet templating wiki.

Now that you have a valid *.template.config/template.json* file, your template is ready to be installed. In your terminal, navigate to the *extensions* folder and run the following command to install the template located at the current folder:

- **On Windows**: `dotnet new -i .\`
- **On Linux or macOS**: `dotnet new -i ./`

This command outputs the list of templates installed, which should include yours.

```
C:\working\templates\extensions> dotnet new -i .\
Usage: new [options]

Options:
  -h, --help          Displays help for this command.
  -l, --list          Lists templates containing the specified name. If no name is specified, lists all
templates.

... cut to save space ...

Templates                                         Short Name            Language          Tags
--------------------------------------------------------------------------------------------------------------
-----------------
Example templates: string extensions              stringext             [C#]              Common/Code
Console Application                               console               [C#], F#, VB      Common/Console
Class library                                     classlib              [C#], F#, VB      Common/Library
WPF Application                                   wpf                   [C#], VB          Common/WPF
Windows Forms (WinForms) Application              winforms              [C#], VB          Common/WinForms
Worker Service                                    worker                [C#]              Common/Worker/Web
```

Test the item template

Now that you have an item template installed, test it. Navigate to the *test/* folder and create a new console application with `dotnet new console`. This generates a working project you can easily test with the `dotnet run` command.

```
C:\test> dotnet new console
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on C:\test\test.csproj...
  Restore completed in 54.82 ms for C:\test\test.csproj.

Restore succeeded.
```

```
C:\test> dotnet run
Hello World!
```

Next, run `dotnet new stringext` to generate the *CommonExtensions.cs* from the template.

```
C:\test> dotnet new stringext
The template "Example templates: string extensions" was created successfully.
```

Change the code in *Program.cs* to reverse the `"Hello World"` string with the extension method provided by the template.

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

Run the program again and you'll see that the result is reversed.

```
C:\test> dotnet run
!dlroW olleH
```

Congratulations! You created and deployed an item template with .NET Core. In preparation for the next part of this tutorial series, you must uninstall the template you created. Make sure to delete all files from the *test* folder too. This will get you back to a clean state ready for the next major section of this tutorial.

Uninstall the template

Because you installed the template by file path, you must uninstall it with the **absolute** file path. You can see a list of templates installed by running the `dotnet new -u` command. Your template should be listed last. Use the path listed to uninstall your template with the `dotnet new -u <ABSOLUTE PATH TO TEMPLATE DIRECTORY>` command.

```
C:\working> dotnet new -u
Template Instantiation Commands for .NET Core CLI

Currently installed items:
  Microsoft.DotNet.Common.ItemTemplates
    Templates:
      dotnet gitignore file (gitignore)
      global.json file (globaljson)
      NuGet Config (nugetconfig)
      Solution File (sln)
      Dotnet local tool manifest file (tool-manifest)
      Web Config (webconfig)

... cut to save space ...

  NUnit3.DotNetNew.Template
    Templates:
      NUnit 3 Test Project (nunit) C#
      NUnit 3 Test Item (nunit-test) C#
      NUnit 3 Test Project (nunit) F#
      NUnit 3 Test Item (nunit-test) F#
      NUnit 3 Test Project (nunit) VB
      NUnit 3 Test Item (nunit-test) VB
  C:\working\templates\extensions
    Templates:
      Example templates: string extensions (stringext) C#
```

```
C:\working> dotnet new -u C:\working\templates\extensions
```

Next steps

In this tutorial, you created an item template. To learn how to create a project template, continue this tutorial series.

Create a project template

Tutorial: Create a project template

10/15/2019 • 5 minutes to read • Edit Online

With .NET Core, you can create and deploy templates that generate projects, files, even resources. This tutorial is part two of a series that teaches you how to create, install, and uninstall, templates for use with the `dotnet new` command.

In this part of the series you'll learn how to:

- Create the resources of a project template
- Create the template config folder and file
- Install a template from a file path
- Test an item template
- Uninstall an item template

Prerequisites

- Complete part 1 of this tutorial series.
- Open a terminal and navigate to the *working\templates\* folder.

Create a project template

Project templates produce ready-to-run projects that make it easy for users to start with a working set of code. .NET Core includes a few project templates such as a console application or a class library. In this example, you'll create a new console project that enables C# 8.0 and produces an `async main` entry point.

In your terminal, navigate to the *working\templates\* folder and create a new subfolder named *consoleasync*. Enter the subfolder and run `dotnet new console` to generate the standard console application. You'll be editing the files produced by this template to create a new template.

```
working
└───templates
    └───consoleasync
            consoleasync.csproj
            Program.cs
```

Modify Program.cs

Open up the *program.cs* file. The console project doesn't use an asynchronous entry point, so let's add that. Change your code to the following and save the file:

```
using System;
using System.Threading.Tasks;

namespace consoleasync
{
    class Program
    {
        static async Task Main(string[] args)
        {
            await Console.Out.WriteAsync("Hello World with C# 8.0!");
        }
    }
}
```

Modify consoleasync.csproj

Let's update the C# language version the project uses to version 8.0. Edit the *consoleasync.csproj* file and add the `<LangVersion>` setting to a `<PropertyGroup>` node.

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.2</TargetFramework>

    <LangVersion>8.0</LangVersion>

  </PropertyGroup>

</Project>
```

Build the project

Before you complete a project template, you should test it to make sure it compiles and runs correctly. In your terminal, run the `dotnet run` command and you should see the following output:

```
C:\working\templates\consoleasync> dotnet run
Hello World with C# 8.0!
```

You can delete the *obj* and *bin* folders created by using `dotnet run`. Deleting these files ensures your template only includes the files related to your template and not any files that result of a build action.

Now that you have the content of the template created, you need to create the template config at the root folder of the template.

Create the template config

Templates are recognized in .NET Core by a special folder and config file that exist at the root of your template. In this tutorial, your template folder is located at *working\templates\consoleasync\*.

When you create a template, all files and folders in the template folder are included as part of the template except for the special config folder. This config folder is named *.template.config*.

First, create a new subfolder named *.template.config*, enter it. Then, create a new file named *template.json*. Your folder structure should look like this:

```
working
└───templates
    └───consoleasync
        └───.template.config
                template.json
```

Open the *template.json* with your favorite text editor and paste in the following json code and save it:

```
{
  "$schema": "http://json.schemastore.org/template",
  "author": "Me",
  "classifications": [ "Common", "Console", "C#8" ],
  "identity": "ExampleTemplate.AsyncProject",
  "name": "Example templates: async project",
  "shortName": "consoleasync",
  "tags": {
    "language": "C#",
    "type": "project"
  }
}
```

This config file contains all of the settings for your template. You can see the basic settings such as `name` and `shortName` but also there's a `tags/type` value that's set to `project`. This designates your template as a project template. There's no restriction on the type of template you create. The `item` and `project` values are common names that .NET Core recommends so that users can easily filter the type of template they're searching for.

The `classifications` item represents the **tags** column you see when you run `dotnet new` and get a list of templates. Users can also search based on classification tags. Don't confuse the `tags` property in the json file with the `classifications` tags list. They're two different things unfortunately named similarly. The full schema for the *template.json* file is found at the JSON Schema Store. For more information about the *template.json* file, see the dotnet templating wiki.

Now that you have a valid *.template.config/template.json* file, your template is ready to be installed. Before you install the template, make sure that you delete any extra files folders and files you don't want included in your template, like the *bin* or *obj* folders. In your terminal, navigate to the *consoleasync* folder and run `dotnet new -i .\` to install the template located at the current folder. If you're using a Linux or MacOS operating system, use a forward slash: `dotnet new -i ./`.

This command outputs the list of templates installed, which should include yours.

```
C:\working\templates\consoleasync> dotnet new -i .\
Usage: new [options]

Options:
  -h, --help          Displays help for this command.
  -l, --list          Lists templates containing the specified name. If no name is specified, lists all
templates.

... cut to save space ...

Templates                                         Short Name            Language          Tags
--------------------------------------------------------------------------------------------------------------
-----------------
Console Application                               console               [C#], F#, VB      Common/Console
Example templates: async project                  consoleasync          [C#]              Common/Console/C#8
Class library                                     classlib              [C#], F#, VB      Common/Library
WPF Application                                   wpf                   [C#], VB          Common/WPF
Windows Forms (WinForms) Application              winforms              [C#], VB          Common/WinForms
Worker Service                                    worker                [C#]              Common/Worker/Web
```

Test the project template

Now that you have an item template installed, test it. Navigate to the *test* folder and create a new console application with `dotnet new consoleasync`. This generates a working project you can easily test with the `dotnet run` command.

```
C:\test> dotnet new consoleasync
The template "Example templates: async project" was created successfully.
```

```
C:\test> dotnet run
Hello World with C# 8.0!
```

Congratulations! You created and deployed a project template with .NET Core. In preparation for the next part of this tutorial series, you must uninstall the template you created. Make sure to delete all files from the *test* folder too. This will get you back to a clean state ready for the next major section of this tutorial.

Uninstall the template

Because you installed the template by using a file path, you must uninstall it with the **absolute** file path. You can see a list of templates installed by running the `dotnet new -u` command. Your template should be listed last. Use the path listed to uninstall your template with the `dotnet new -u <ABSOLUTE PATH TO TEMPLATE DIRECTORY>` command.

```
C:\working> dotnet new -u
Template Instantiation Commands for .NET Core CLI

Currently installed items:
  Microsoft.DotNet.Common.ItemTemplates
    Templates:
      dotnet gitignore file (gitignore)
      global.json file (globaljson)
      NuGet Config (nugetconfig)
      Solution File (sln)
      Dotnet local tool manifest file (tool-manifest)
      Web Config (webconfig)

... cut to save space ...

  NUnit3.DotNetNew.Template
    Templates:
      NUnit 3 Test Project (nunit) C#
      NUnit 3 Test Item (nunit-test) C#
      NUnit 3 Test Project (nunit) F#
      NUnit 3 Test Item (nunit-test) F#
      NUnit 3 Test Project (nunit) VB
      NUnit 3 Test Item (nunit-test) VB
  C:\working\templates\consoleasync
    Templates:
      Example templates: async project (consoleasync) C#
```

```
C:\working> dotnet new -u C:\working\templates\consoleasync
```

Next steps

In this tutorial, you created a project template. To learn how to package both the item and project templates into an easy-to-use file, continue this tutorial series.

Create a template pack

Tutorial: Create a template pack

9/19/2019 • 5 minutes to read • Edit Online

With .NET Core, you can create and deploy templates that generate projects, files, even resources. This tutorial is part three of a series that teaches you how to create, install, and uninstall, templates for use with the `dotnet new` command.

In this part of the series you'll learn how to:

- Create a *.csproj project to build a template pack
- Configure the project file for packing
- Install a template from a NuGet package file
- Uninstall a template by package ID

Prerequisites

- Complete part 1 and part 2 of this tutorial series.

  This tutorial uses the two templates created in the first two parts of this tutorial. It's possible you can use a different template as long as you copy the template as a folder into the *working\templates\* folder.

- Open a terminal and navigate to the *working\templates\* folder.

Create a template pack project

A template pack is one or more templates packaged into a file. When you install or uninstall a pack, all templates contained in the pack are added or removed, respectively. The previous parts of this tutorial series only worked with individual templates. To share a non-packed template, you have to copy the template folder and install via that folder. Because a template pack can have more than one template in it, and is a single file, sharing is easier.

Template packs are represented by a NuGet package (*.nupkg*) file. And, like any NuGet package, you can upload the template pack to a NuGet feed. The `dotnet new -i` command supports installing template pack from a NuGet package feed. Additionally, you can install a template pack from a *.nupkg* file directly.

Normally you use a C# project file to compile code and produce a binary. However, the project can also be used to generate a template pack. By changing the settings of the *.csproj*, you can prevent it from compiling any code and instead include all the assets of your templates as resources. When this project is built, it produces a template pack NuGet package.

The pack you'll create will include the item template and package template previously created. Because we grouped the two templates into the *working\templates\* folder, we can use the *working* folder for the *.csproj* file.

In your terminal, navigate to the *working* folder. Create a new project and set the name to `templatepack` and the output folder to the current folder.

```
dotnet new console -n templatepack -o .
```

The `-n` parameter sets the *.csproj* filename to *templatepack.csproj* and the `-o` parameters creates the files in the current directory. You should see a result similar to the following output.

```
C:\working> dotnet new console -n templatepack -o .
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on .\templatepack.csproj...
  Restore completed in 52.38 ms for C:\working\templatepack.csproj.

Restore succeeded.
```

Next, open the *templatepack.csproj* file in your favorite editor and replace the content with the following XML:

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

  <PropertyGroup>
    <PackageType>Template</PackageType>
    <PackageVersion>1.0</PackageVersion>
    <PackageId>AdatumCorporation.Utility.Templates</PackageId>
    <Title>AdatumCorporation Templates</Title>
    <Authors>Me</Authors>
    <Description>Templates to use when creating an application for Adatum Corporation.</Description>
    <PackageTags>dotnet-new;templates;contoso</PackageTags>

    <TargetFramework>netstandard2.0</TargetFramework>

    <IncludeContentInPack>true</IncludeContentInPack>
    <IncludeBuildOutput>false</IncludeBuildOutput>
    <ContentTargetFolders>content</ContentTargetFolders>
  </PropertyGroup>

  <ItemGroup>
    <Content Include="templates\**\*" Exclude="templates\**\bin\**;templates\**\obj\**" />
    <Compile Remove="**\*" />
  </ItemGroup>

</Project>
```

The `<PropertyGroup>` settings in the XML above is broken into three groups. The first group deals with properties required for a NuGet package. The three `<Package` settings have to do with the NuGet package properties to identify your package on a NuGet feed. Specifically the `<PacakgeId>` value is used to uninstall the template pack with a single name instead of a directory path. It can also be used to install the template pack from a NuGet feed. The remaining settings such as `<Title>` and `<Tags>` have to do with metadata displayed on the NuGet feed. For more information about NuGet settings, see NuGet and MSBuild properties.

The `<TargetFramework>` setting must be set so that MSBuild will run properly when you run the pack command to compile and pack the project.

The last three settings have to do with configuring the project correctly to include the templates in the appropriate folder in the NuGet pack when it's created.

The `<ItemGroup>` contains two settings. First, the `<Content>` setting includes everything in the *templates* folder as content. It's also set to exclude any *bin* folder or *obj* folder to prevent any compiled code (if you tested and compiled your templates) from being included. Second, the `<Compile>` setting excludes all code files from compiling no matter where they're located. This setting prevents the project being used to create a template pack from trying to compile the code in the *templates* folder hierarchy.

Build and install

Save this file and then run the pack command

```
dotnet pack
```

This command will build your project and create a NuGet package in This should be the *working\bin\Debug* folder.

```
C:\working> dotnet pack
Microsoft (R) Build Engine version 16.2.0-preview-19278-01+d635043bd for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

  Restore completed in 123.86 ms for C:\working\templatepack.csproj.

  templatepack -> C:\working\bin\Debug\netstandard2.0\templatepack.dll
  Successfully created package 'C:\working\bin\Debug\AdatumCorporation.Utility.Templates.1.0.0.nupkg'.
```

Next, install the template pack file with the `dotnet new -i PATH_TO_NUPKG_FILE` command.

```
C:\working> dotnet new -i C:\working\bin\Debug\AdatumCorporation.Utility.Templates.1.0.0.nupkg
Usage: new [options]

Options:
  -h, --help          Displays help for this command.
  -l, --list          Lists templates containing the specified name. If no name is specified, lists all
templates.

... cut to save space ...

Templates                                         Short Name            Language          Tags
--------------------------------------------------------------------------------------------------------------
-----------------
Example templates: string extensions              stringext             [C#]              Common/Code
Console Application                               console               [C#], F#, VB      Common/Console
Example templates: async project                  consoleasync          [C#]              Common/Console/C#8
Class library                                     classlib              [C#], F#, VB      Common/Library
```

If you uploaded the NuGet package to a NuGet feed, you can use the `dotnet new -i PACKAGEID` command where `PACKAGEID` is the same as the `<PackageId>` setting from the *.csproj* file. This package ID is the same as the NuGet package identifier.

Uninstall the template pack

No matter how you installed the template pack, either with the *.nupkg* file directly or by NuGet feed, removing a template pack is the same. Use the `<PackageId>` of the template you want to uninstall. You can get a list of templates that are installed by running the `dotnet new -u` command.

```
C:\working> dotnet new -u
Template Instantiation Commands for .NET Core CLI

Currently installed items:
  Microsoft.DotNet.Common.ItemTemplates
    Templates:
      dotnet gitignore file (gitignore)
      global.json file (globaljson)
      NuGet Config (nugetconfig)
      Solution File (sln)
      Dotnet local tool manifest file (tool-manifest)
      Web Config (webconfig)

... cut to save space ...

  NUnit3.DotNetNew.Template
    Templates:
      NUnit 3 Test Project (nunit) C#
      NUnit 3 Test Item (nunit-test) C#
      NUnit 3 Test Project (nunit) F#
      NUnit 3 Test Item (nunit-test) F#
      NUnit 3 Test Project (nunit) VB
      NUnit 3 Test Item (nunit-test) VB
  AdatumCorporation.Utility.Templates
    Templates:
      Example templates: async project (consoleasync) C#
      Example templates: string extensions (stringext) C#
```

Run `dotnet new -u AdatumCorporation.Utility.Templates` to uninstall the template. The `dotnet new` command will output help information that should omit the templates you previously installed.

Congratulations! you've installed and uninstalled a template pack.

Next steps

To learn more about templates, most of which you've already learned, see the Custom templates for dotnet new article.

- dotnet/templating GitHub repo Wiki
- dotnet/dotnet-template-samples GitHub repo
- *template.json* schema at the JSON Schema Store

Tutorial: Create a .NET Core solution in macOS using Visual Studio Code

9/19/2019 • 6 minutes to read • Edit Online

This document provides the steps and workflow to create a .NET Core solution for macOS. Learn how to create projects, unit tests, use the debugging tools, and incorporate third-party libraries via NuGet.

NOTE

This article uses Visual Studio Code on macOS.

Prerequisites

Install the .NET Core SDK. The .NET Core SDK includes the latest release of the .NET Core framework and runtime.

Install Visual Studio Code. During the course of this article, you also install Visual Studio Code extensions that improve the .NET Core development experience.

Install the Visual Studio Code C# extension by opening Visual Studio Code and pressing `F1` to open the Visual Studio Code palette. Type **ext install** to see the list of extensions. Select the C# extension. Restart Visual Studio Code to activate the extension. For more information, see the Visual Studio Code C# Extension documentation.

Get started

In this tutorial, you create three projects: a library project, tests for that library project, and a console application that makes use of the library. You can view or download the source for this topic at the dotnet/samples repository on GitHub. For download instructions, see Samples and Tutorials.

Start Visual Studio Code. Press `Ctrl+`` (the backquote or backtick character) or select **View > Integrated Terminal** from the menu to open an embedded terminal in Visual Studio Code. You can still open an external shell with the Explorer **Open in Command Prompt** command (**Open in Terminal** on Mac or Linux) if you prefer to work outside of Visual Studio Code.

Begin by creating a solution file, which serves as a container for one or more .NET Core projects. In the terminal, run the `dotnet new` command to create a new solution *golden.sln* inside a new folder named *golden*:

```
dotnet new sln -o golden
```

Navigate to the new *golden* folder and execute the following command to create a library project, which produces two files,*library.csproj* and *Class1.cs*, in the *library* folder:

```
dotnet new classlib -o library
```

Execute the `dotnet sln` command to add the newly created *library.csproj* project to the solution:

```
dotnet sln add library/library.csproj
```

The *library.csproj* file contains the following information:

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

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>

</Project>
```

Our library methods serialize and deserialize objects in JSON format. To support JSON serialization and deserialization, add a reference to the `Newtonsoft.Json` NuGet package. The `dotnet add` command adds new items to a project. To add a reference to a NuGet package, use the `dotnet add package` command and specify the name of the package:

```
dotnet add library package Newtonsoft.Json
```

This adds `Newtonsoft.Json` and its dependencies to the library project. Alternatively, manually edit the *library.csproj* file and add the following node:

```
<ItemGroup>
  <PackageReference Include="Newtonsoft.Json" Version="12.0.2" />
</ItemGroup>
```

Execute `dotnet restore`, (see note) which restores dependencies and creates an *obj* folder inside *library* with three files in it, including a *project.assets.json* file:

```
dotnet restore
```

In the *library* folder, rename the file *Class1.cs* to *Thing.cs*. Replace the code with the following:

```
using static Newtonsoft.Json.JsonConvert;

namespace Library
{
    public class Thing
    {
        public int Get(int left, int right) =>
            DeserializeObject<int>($"{left + right}");
    }
}
```

The `Thing` class contains one public method, `Get`, which returns the sum of two numbers but does so by converting the sum into a string and then deserializing it into an integer. This makes use of a number of modern C# features, such as `using static` directives, expression-bodied members, and string interpolation.

Build the library with the `dotnet build` command. This produces a *library.dll* file under *golden/library/bin/Debug/netstandard1.4*:

```
dotnet build
```

Create the test project

Build a test project for the library. From the *golden* folder, create a new test project:

```
dotnet new xunit -o test-library
```

Add the test project to the solution:

```
dotnet sln add test-library/test-library.csproj
```

Add a project reference the library you created in the previous section so that the compiler can find and use the library project. Use the `dotnet add reference` command:

```
dotnet add test-library/test-library.csproj reference library/library.csproj
```

Alternatively, manually edit the *test-library.csproj* file and add the following node:

```
<ItemGroup>
  <ProjectReference Include="..\library\library.csproj" />
</ItemGroup>
```

Now that the dependencies have been properly configured, create the tests for your library. Open *UnitTest1.cs* and replace its contents with the following code:

```
using Library;
using Xunit;

namespace TestApp
{
    public class LibraryTests
    {
        [Fact]
        public void TestThing() {
            Assert.NotEqual(42, new Thing().Get(19, 23));
        }
    }
}
```

Note that you assert the value 42 is not equal to 19+23 (or 42) when you first create the unit test (`Assert.NotEqual`), which will fail. An important step in building unit tests is to create the test to fail once first to confirm its logic.

From the *golden* folder, execute the following commands:

```
dotnet restore
dotnet test test-library/test-library.csproj
```

These commands will recursively find all projects to restore dependencies, build them, and activate the xUnit test runner to run the tests. The single test fails, as you expect.

Edit the *UnitTest1.cs* file and change the assertion from `Assert.NotEqual` to `Assert.Equal`. Execute the following command from the *golden* folder to re-run the test, which passes this time:

```
dotnet test test-library/test-library.csproj
```

Create the console app

The console app you create over the following steps takes a dependency on the library project you created earlier and calls its library method when it runs. Using this pattern of development, you see how to create reusable libraries for multiple projects.

Create a new console application from the *golden* folder:

```
dotnet new console -o app
```

Add the console app project to the solution:

```
dotnet sln add app/app.csproj
```

Create the dependency on the library by running the `dotnet add reference` command:

```
dotnet add app/app.csproj reference library/library.csproj
```

Run `dotnet restore` (see note) to restore the dependencies of the three projects in the solution. Open *Program.cs* and replace the contents of the `Main` method with the following line:

```
WriteLine($"The answer is {new Thing().Get(19, 23)}");
```

Add two `using` directives to the top of the *Program.cs* file:

```
using static System.Console;
using Library;
```

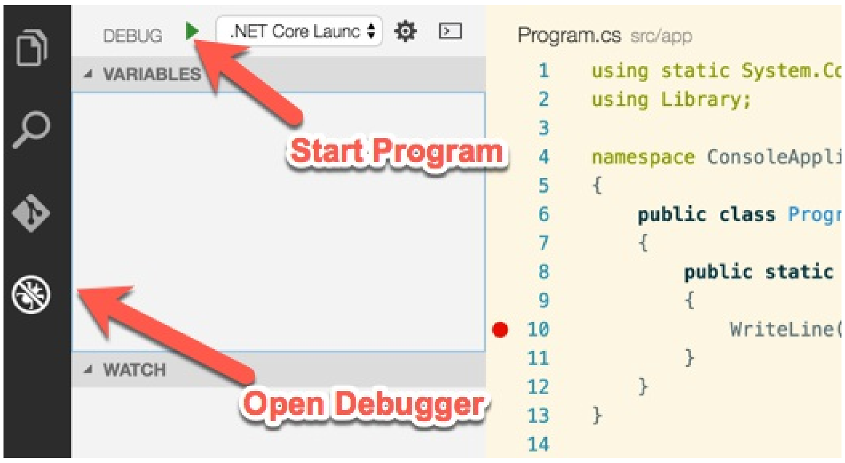
Execute the following `dotnet run` command to run the executable, where the `-p` option to `dotnet run` specifies the project for the main application. The app produces the string "The answer is 42".

```
dotnet run -p app/app.csproj
```

Debug the application

Set a breakpoint at the `WriteLine` statement in the `Main` method. Do this by either pressing the F9 key when the cursor is over the `WriteLine` line or by clicking the mouse in the left margin on the line where you want to set the breakpoint. A red circle will appear in the margin next to the line of code. When the breakpoint is reached, code execution will stop *before* the breakpoint line is executed.

Open the debugger tab by selecting the Debug icon in the Visual Studio Code toolbar, selecting **View > Debug** from the menu bar, or using the keyboard shortcut CTRL+SHIFT+D:



```
using static System.Co
using Library;

namespace ConsoleAppli
{
    public class Progr
    {
        public static
        {
            WriteLine(
        }
    }
}

```

Press the Play button to start the application under the debugger. The app begins execution and runs to the breakpoint, where it stops. Step into the `Get` method and make sure that you have passed in the correct arguments. Confirm that the answer is 42.

> **NOTE**
>
> Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

Get started with .NET Core on macOS using Visual Studio for Mac

9/23/2019 • 2 minutes to read • Edit Online

Visual Studio for Mac provides a full-featured Integrated Development Environment (IDE) for developing .NET Core applications. This topic walks you through building a simple console application using Visual Studio for Mac and .NET Core.

NOTE

Your feedback is highly valued. There are two ways you can provide feedback to the development team on Visual Studio for Mac:

- In Visual Studio for Mac, select **Help** > **Report a Problem** from the menu or **Report a Problem** from the Welcome screen, which will open a window for filing a bug report. You can track your feedback in the Developer Community portal.
- To make a suggestion, select **Help** > **Provide a Suggestion** from the menu or **Provide a Suggestion** from the Welcome screen, which will take you to the Visual Studio for Mac Developer Community webpage.

Prerequisites

See the Prerequisites for .NET Core on Mac topic.

Check the .NET Core Support article to ensure you're using a supported version of .NET Core.

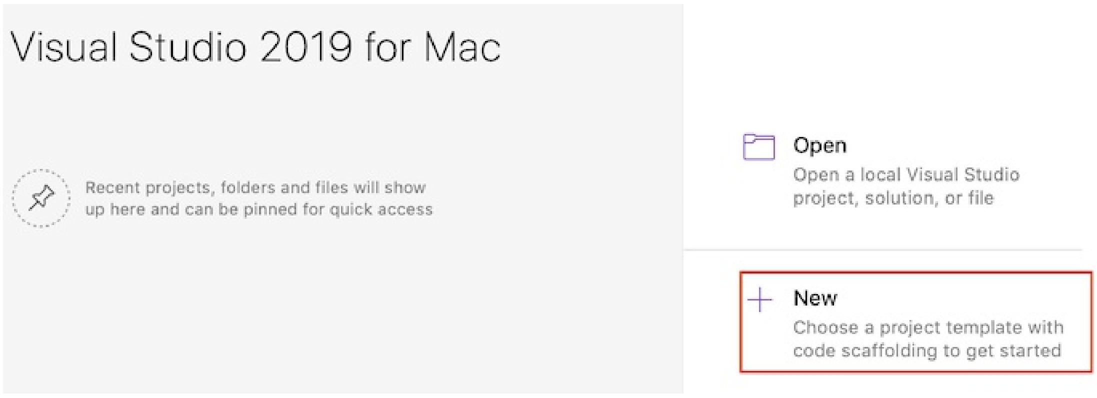
Get started

If you've already installed the prerequisites and Visual Studio for Mac, skip this section and proceed to Creating a project. Follow these steps to install the prerequisites and Visual Studio for Mac:

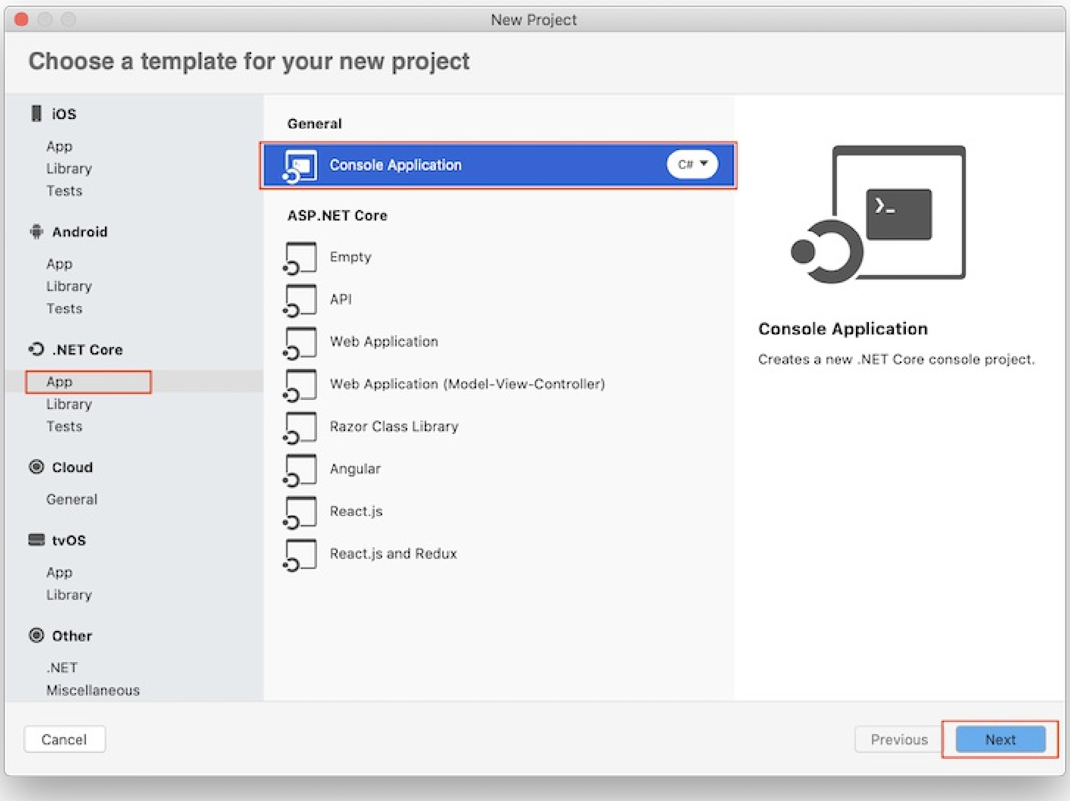
Download the Visual Studio for Mac installer. Run the installer. Read and accept the license agreement. During the install, select the option to install .NET Core. You're provided the opportunity to install Xamarin, a cross-platform mobile app development technology. Installing Xamarin and its related components is optional for .NET Core development. For a walk-through of the Visual Studio for Mac install process, see Visual Studio for Mac documentation. When the install is complete, start the Visual Studio for Mac IDE.

Creating a project

1. Select **New** on the Start Window.

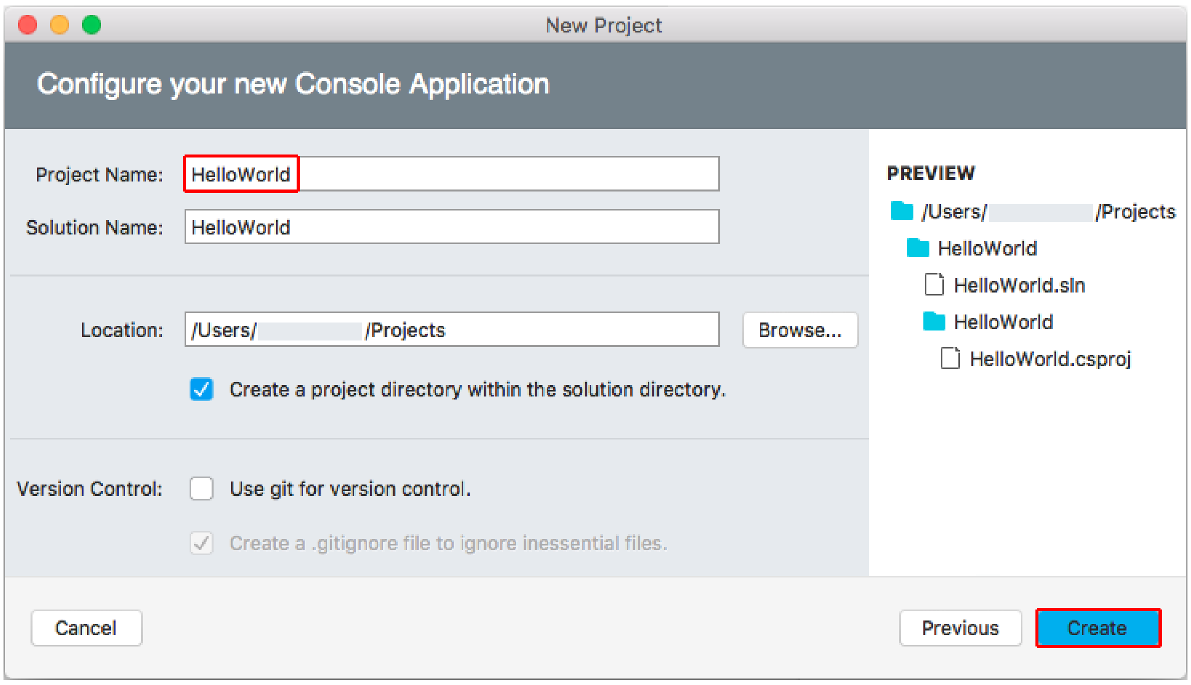

Visual Studio 2019 for Mac
Recent projects, folders and files will show up here and can be pinned for quick access
Open
Open a local Visual Studio project, solution, or file
New
Choose a project template with code scaffolding to get started

2. In the **New Project** dialog, select **App** under the **.NET Core** node. Select the **Console Application** template followed by **Next**.



3. If you have more than one version of .NET Core installed, select the target framework for your project.

4. Type "HelloWorld" for the **Project Name**. Select **Create**.



Project Name: HelloWorld

Solution Name: HelloWorld

Location: /Users/ /Projects

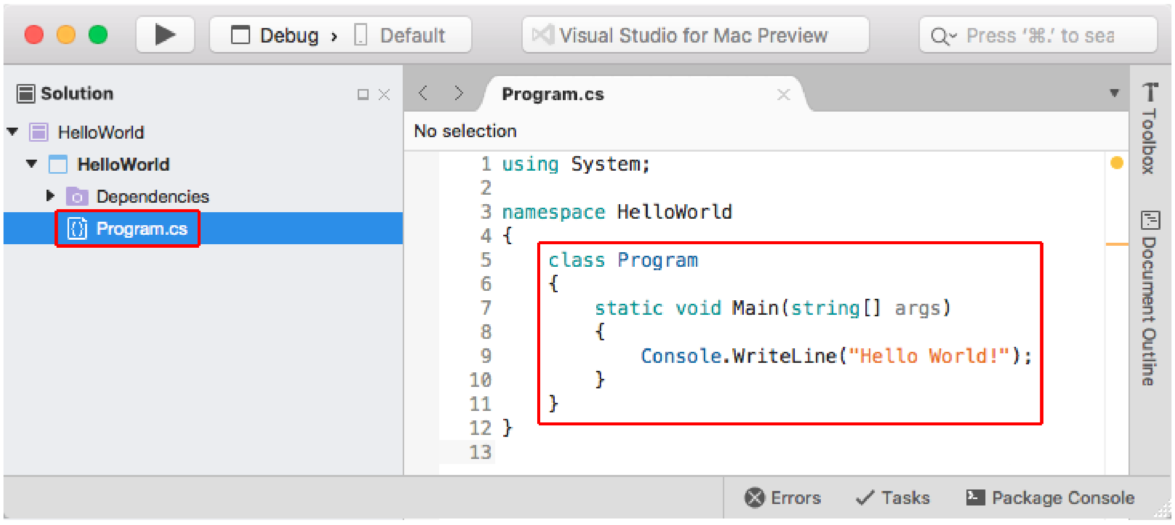
☑ Create a project directory within the solution directory.

Version Control: ☐ Use git for version control.

☑ Create a .gitignore file to ignore inessential files.

/Users/ /Projects
HelloWorld
HelloWorld.sln
HelloWorld
HelloWorld.csproj

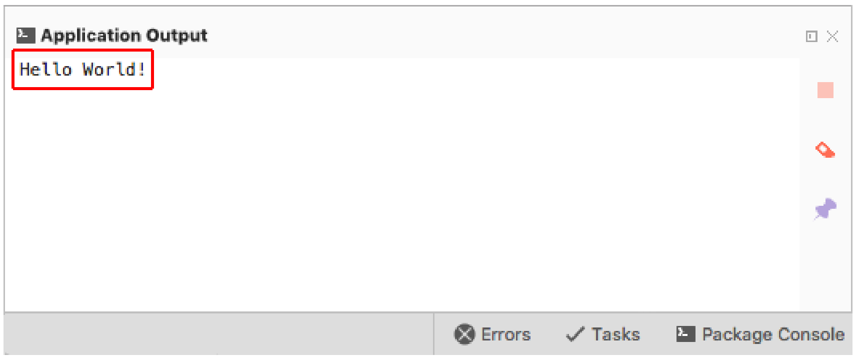
5. Wait while the project's dependencies are restored. The project has a single C# file, *Program.cs*, containing a `Program` class with a `Main` method. The `Console.WriteLine` statement will output "Hello World!" to the console when the app is run.



```
using System;

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

```

Run the application

Run the app in Debug mode using ⌘ ↵ (command + enter) or in Release mode using ⌥ ⌘ ↵ (option + command + enter).



```
Hello World!
```

Next step

The Building a complete .NET Core solution on macOS using Visual Studio for Mac topic shows you how to build a complete .NET Core solution that includes a reusable library and unit testing.

Building a complete .NET Core solution on macOS using Visual Studio for Mac

9/12/2019 • 9 minutes to read • Edit Online

Visual Studio for Mac provides a full-featured Integrated Development Environment (IDE) for developing .NET Core applications. This topic walks you through building a .NET Core solution that includes a reusable library and unit testing.

This tutorial shows you how to create an application that accepts a search word and a string of text from the user, counts the number of times the search word appears in the string using a method in a class library, and returns the result to the user. The solution also includes unit testing for the class library as an introduction to unit testing concepts. If you prefer to proceed through the tutorial with a complete sample, download the sample solution. For download instructions, see Samples and Tutorials.

NOTE

Your feedback is highly valued. There are two ways you can provide feedback to the development team on Visual Studio for Mac:

- In Visual Studio for Mac, select **Help** > **Report a Problem** from the menu or **Report a Problem** from the Welcome screen, which opens a window for filing a bug report. You can track your feedback in the Developer Community portal.
- To make a suggestion, select **Help** > **Provide a Suggestion** from the menu or **Provide a Suggestion** from the Welcome screen, which takes you to the Visual Studio for Mac Developer Community webpage.

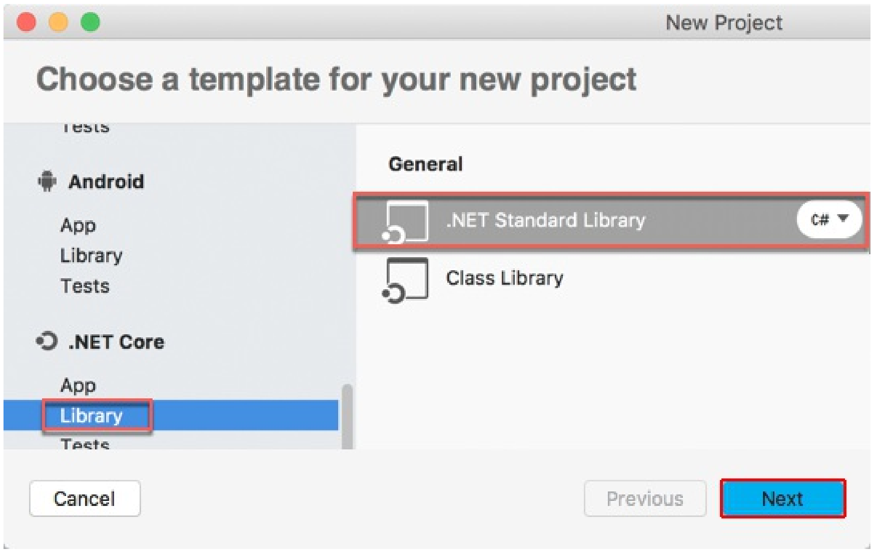
Prerequisites

- OpenSSL (if running .NET Core 1.1): See the Prerequisites for .NET Core on Mac topic.
- .NET Core SDK 1.1 or later
- Visual Studio 2017 for Mac

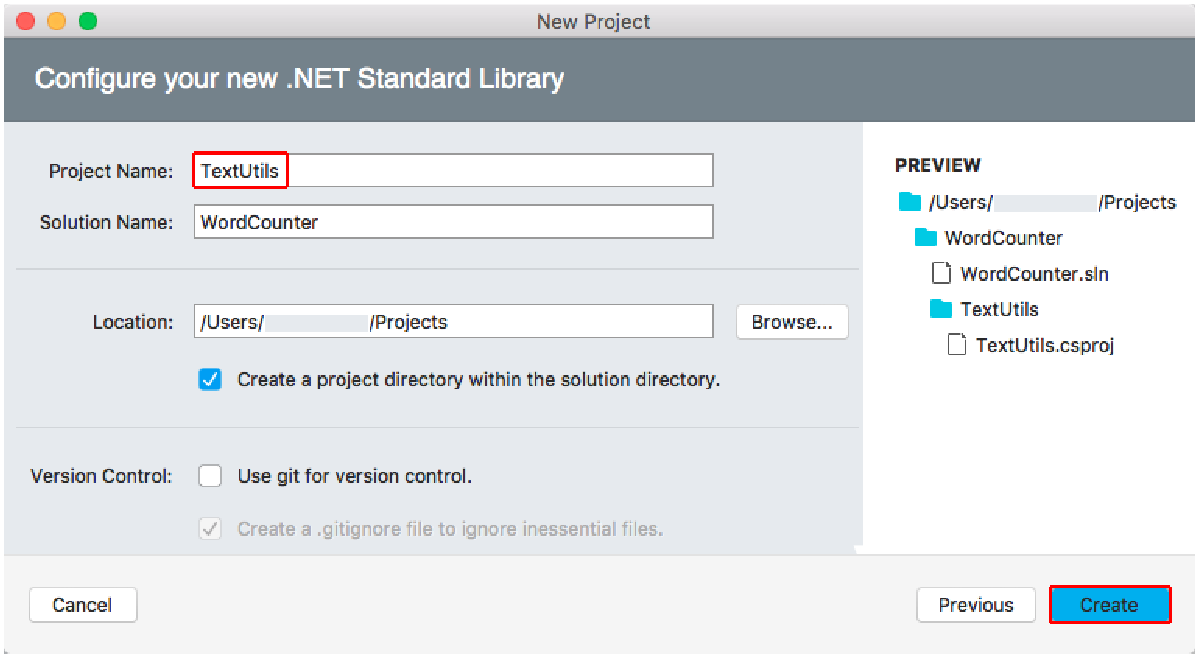
For more information on prerequisites, see the Prerequisites for .NET Core on Mac. For the full system requirements of Visual Studio 2017 for Mac, see Visual Studio 2017 for Mac Product Family System Requirements.

Building a library

1. On the Welcome screen, select **New Project**. In the **New Project** dialog under the **.NET Core** node, select the **.NET Standard Library** template. This creates a .NET Standard library that targets .NET Core as well as any other .NET implementation that supports version 2.0 of the .NET Standard. Select **Next**.



2. Name the project "TextUtils" (a short name for "Text Utilities") and the solution "WordCounter". Leave **Create a project directory within the solution directory** checked. Select **Create**.


New Project

Configure your new .NET Standard Library

Project Name: TextUtils

Solution Name: WordCounter

Location: /Users/ /Projects

Browse...

✓ Create a project directory within the solution directory.

Version Control: Use git for version control.

✓ Create a .gitignore file to ignore inessential files.

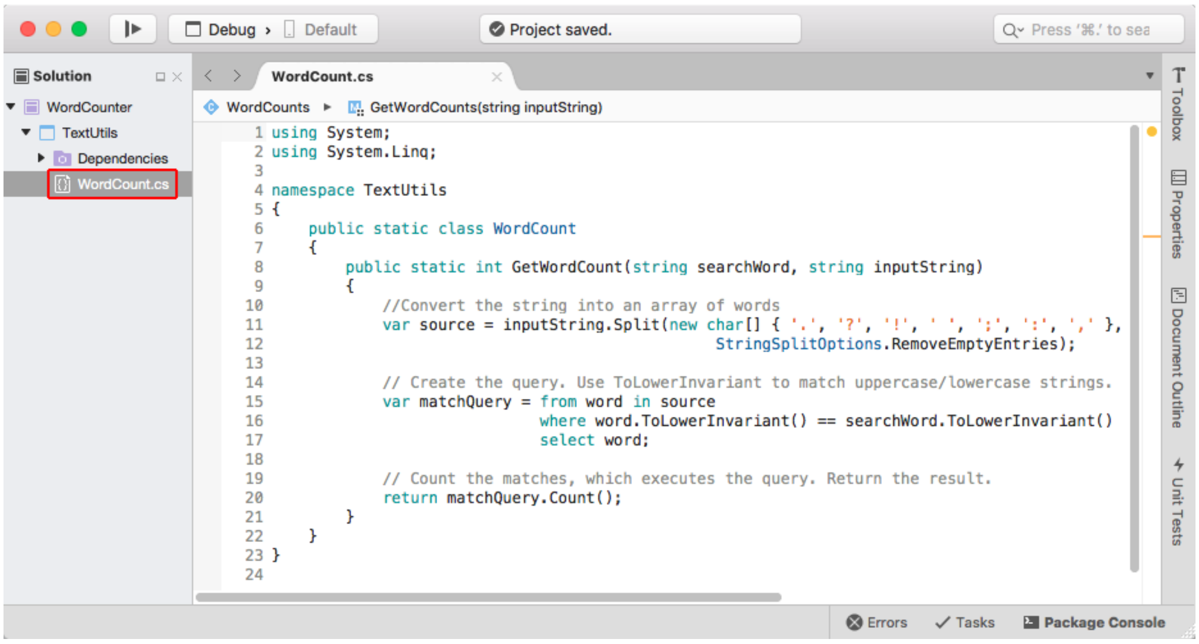
PREVIEW

/Users/ /Projects
WordCounter
WordCounter.sln
TextUtils
TextUtils.csproj

3. In the **Solution** sidebar, expand the `TextUtils` node to reveal the class file provided by the template, *Class1.cs*. Right-click the file, select **Rename** from the context menu, and rename the file to *WordCount.cs*. Open the file and replace the contents with the following code:

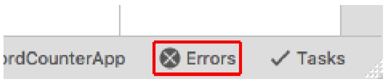
```
using System;
using System.Linq;

namespace TextUtils
{
    public static class WordCount
    {
        public static int GetWordCount(string searchWord, string inputString)
        {
            // Null check these variables and determine if they have values.
            if (string.IsNullOrEmpty(searchWord) || string.IsNullOrEmpty(inputString))
            {
                return 0;
            }

            // Convert the string into an array of words.
            var source = inputString.Split(new char[] { '.', '?', '!', ' ', ';', ':', ',' },
                                           StringSplitOptions.RemoveEmptyEntries);

            // Create the query. Use ToLowerInvariant to match uppercase/lowercase strings.
            var matchQuery = from word in source
                             where word.ToLowerInvariant() == searchWord.ToLowerInvariant()
                             select word;

            // Count the matches, which executes the query. Return the result.
            return matchQuery.Count();
        }
    }
}
```

4. Save the file by using any of three different methods: use the keyboard shortcut ⌘+s, select **File** > **Save** from the menu, or right-click on the file's tab and select **Save** from the contextual menu. The following image shows the IDE window:



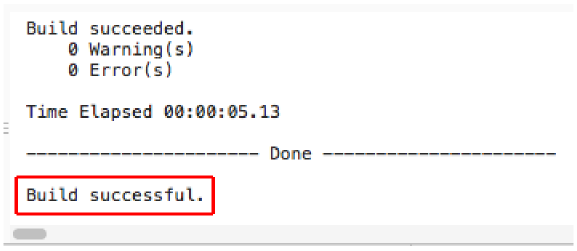
```
using System;
using System.Linq;

namespace TextUtils
{
    public static class WordCount
    {
        public static int GetWordCount(string searchWord, string inputString)
        {
            //Convert the string into an array of words
            var source = inputString.Split(new char[] { '.', '?', '!', ' ', ';', ':', ',' },
                                           StringSplitOptions.RemoveEmptyEntries);

            // Create the query. Use ToLowerInvariant to match uppercase/lowercase strings.
            var matchQuery = from word in source
                             where word.ToLowerInvariant() == searchWord.ToLowerInvariant()
                             select word;

            // Count the matches, which executes the query. Return the result.
            return matchQuery.Count();
        }
    }
}

```

5. Select **Errors** in the margin at the bottom of the IDE window to open the **Errors** panel. Select the **Build Output** button.



6. Select **Build** > **Build All** from the menu.

The solution builds. The build output panel shows that the build is successful.

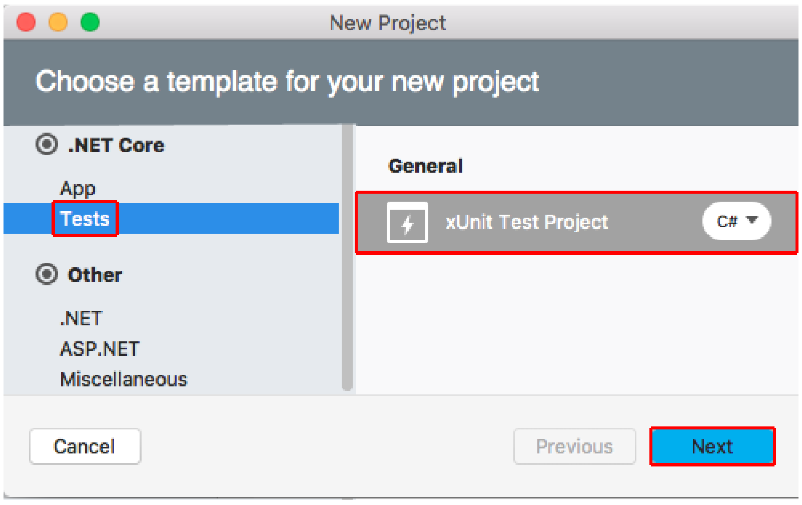


```
Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:05.13

---------------------- Done ----------------------

Build successful.
```

Creating a test project

Unit tests provide automated software testing during your development and publishing. The testing framework that you use in this tutorial is xUnit (version 2.2.0 or later), which is installed automatically when the xUnit test project is added to the solution in the following steps:

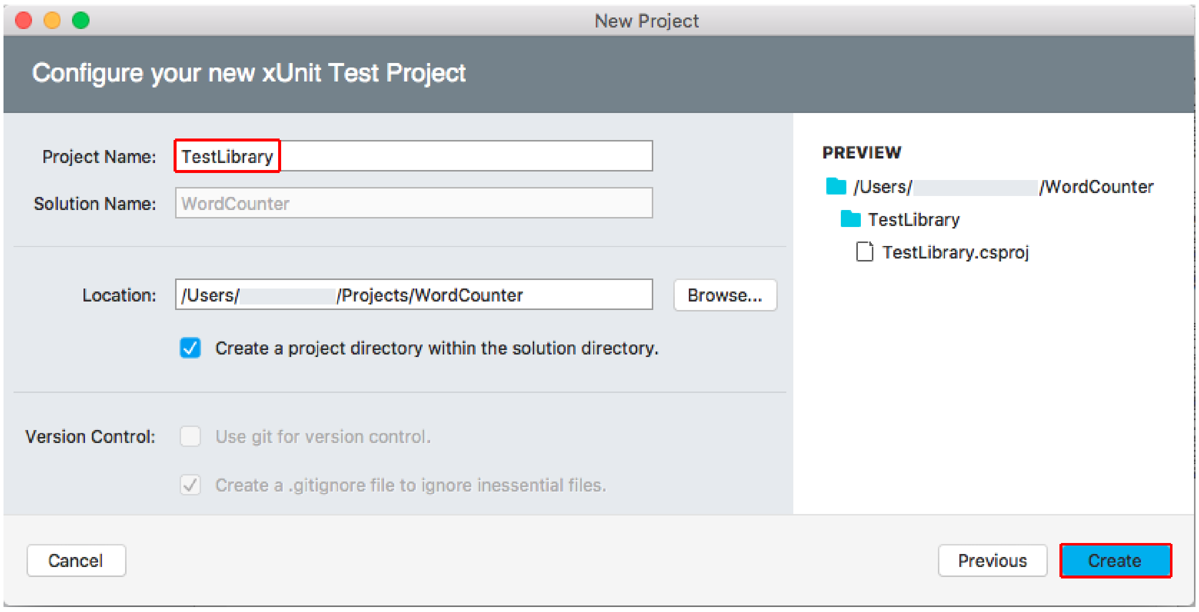
1. In the **Solution** sidebar, right-click the `WordCounter` solution and select **Add** > **Add New Project**.
2. In the **New Project** dialog, select **Tests** from the **.NET Core** node. Select the **xUnit Test Project** followed by **Next**.



New Project

Choose a template for your new project

3. Name the new project "TestLibrary" and select **Create**.



New Project

Configure your new xUnit Test Project

Project Name: TestLibrary

Solution Name: WordCounter

Location: /Users/ /Projects/WordCounter

☑ Create a project directory within the solution directory.

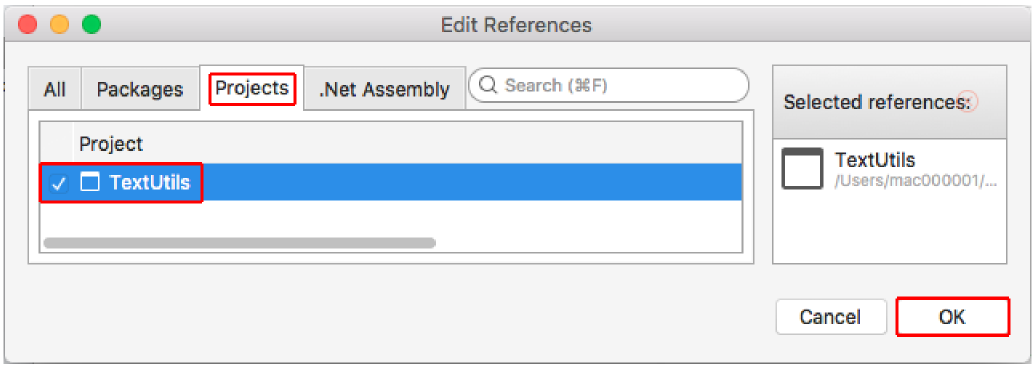
Version Control: ☐ Use git for version control.

☑ Create a .gitignore file to ignore inessential files.

PREVIEW

/Users/ /WordCounter
TestLibrary
TestLibrary.csproj

4. In order for the test library to work with the `WordCount` class, add a reference to the `TextUtils` project. In the **Solution** sidebar, right-click **Dependencies** under **TestLibrary**. Select **Edit References** from the context menu.
5. In the **Edit References** dialog, select the **TextUtils** project on the **Projects** tab. Select **OK**.

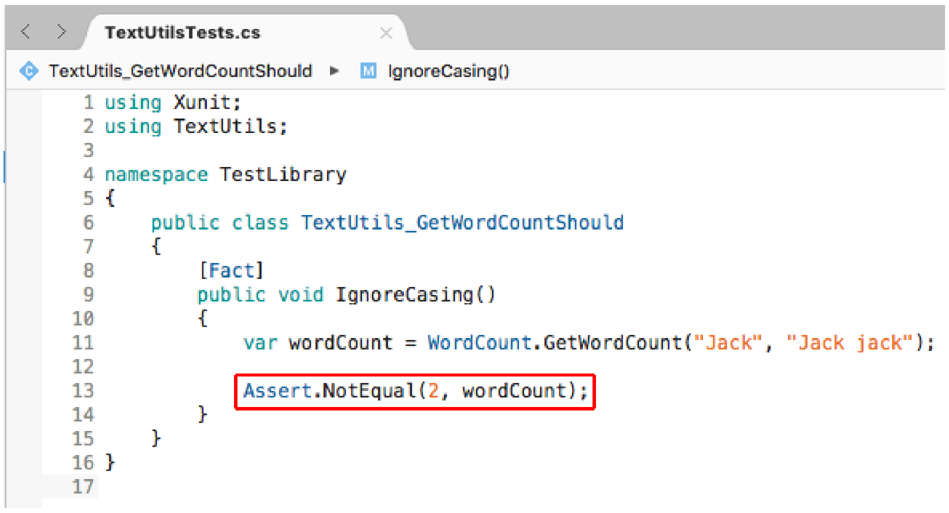


6. In the **TestLibrary** project, rename the *UnitTest1.cs* file to *TextUtilsTests.cs*.

7. Open the file and replace the code with the following:

```
using Xunit;
using TextUtils;
using System.Diagnostics;

namespace TestLibrary
{
    public class TextUtils_GetWordCountShould
    {
        [Fact]
        public void IgnoreCasing()
        {
            var wordCount = WordCount.GetWordCount("Jack", "Jack jack");

            Assert.NotEqual(2, wordCount);
        }
    }
}
```

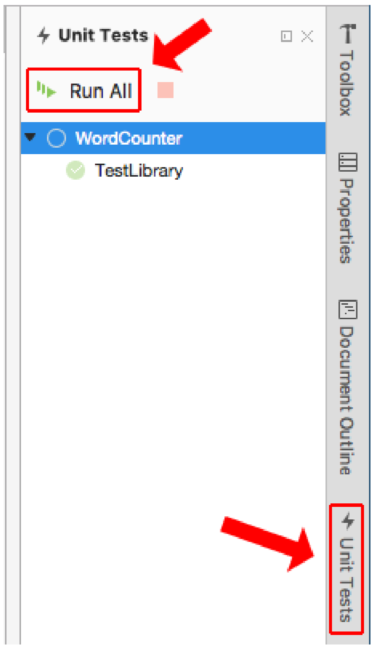
The following image shows the IDE with the unit test code in place. Pay attention to the `Assert.NotEqual` statement.



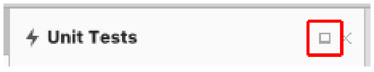
```
using Xunit;
using TextUtils;

namespace TestLibrary
{
    public class TextUtils_GetWordCountShould
    {
        [Fact]
        public void IgnoreCasing()
        {
            var wordCount = WordCount.GetWordCount("Jack", "Jack jack");

            Assert.NotEqual(2, wordCount);
        }
    }
}
```

It's important to make a new test fail once to confirm its testing logic is correct. The method passes in the name "Jack" (uppercase) and a string with "Jack" and "jack" (uppercase and lowercase). If the `GetWordCount` method is working properly, it returns a count of two instances of the search word. In order to fail this test on purpose, you first implement the test asserting that two instances of the search word "Jack" aren't returned by the `GetWordCount` method. Continue to the next step to fail the test on purpose.

8. Open the **Unit Tests** panel on the right side of the screen.

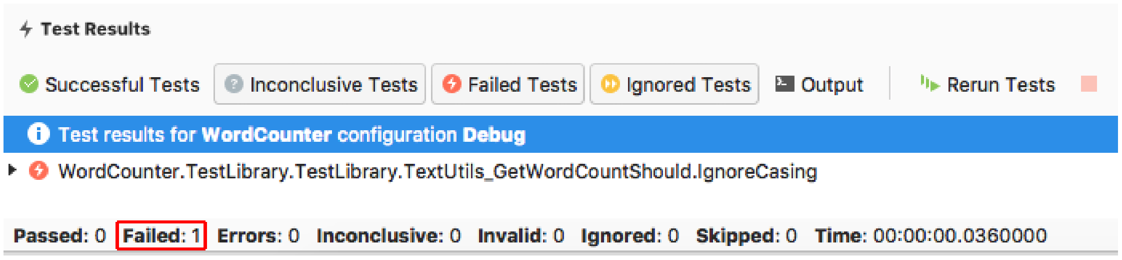


9. Click the **Dock** icon to keep the panel open.



10. Click the **Run All** button.

    The test fails, which is the correct result. The test method asserts that two instances of the `inputString`, "Jack," aren't returned from the string "Jack jack" provided to the `GetWordCount` method. Since word casing was factored out in the `GetWordCount` method, two instances are returned. The assertion that 2 *is not equal to* 2 fails. This is the correct outcome, and the logic of our test is good.



Test Results

Successful Tests | Inconclusive Tests | Failed Tests | Ignored Tests | Output | Rerun Tests

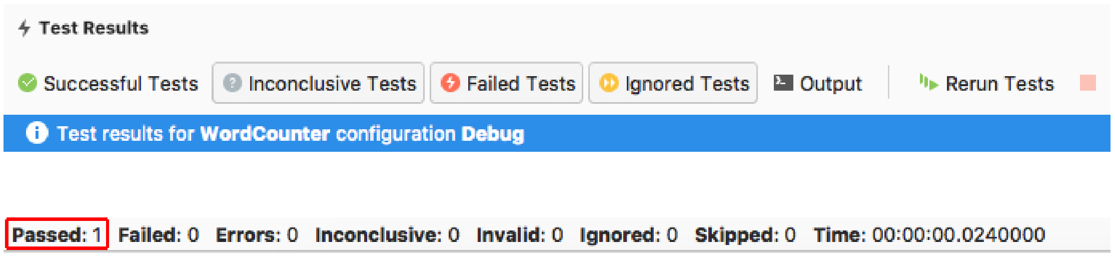
Test results for **WordCounter** configuration **Debug**

WordCounter.TestLibrary.TestLibrary.TextUtils_GetWordCountShould.IgnoreCasing

Passed: 0 **Failed:** 1 **Errors:** 0 **Inconclusive:** 0 **Invalid:** 0 **Ignored:** 0 **Skipped:** 0 **Time:** 00:00:00.0360000

11. Modify the `IgnoreCasing` test method by changing `Assert.NotEqual` to `Assert.Equal`. Save the file by using the keyboard shortcut ⌘+s, **File** > **Save** from the menu, or right-clicking on the file's tab and selecting **Save** from the context menu.

    You expect that the `searchWord` "Jack" returns two instances with `inputString` "Jack jack" passed into `GetWordCount`. Run the test again by clicking the **Run Tests** button in the **Unit Tests** panel or the **Rerun Tests** button in the **Test Results** panel at the bottom of the screen. The test passes. There are two instances of "Jack" in the string "Jack jack" (ignoring casing), and the test assertion is `true`.



Test Results

Successful Tests | Inconclusive Tests | Failed Tests | Ignored Tests | Output | Rerun Tests

Test results for **WordCounter** configuration **Debug**

Passed: 1 **Failed:** 0 **Errors:** 0 **Inconclusive:** 0 **Invalid:** 0 **Ignored:** 0 **Skipped:** 0 **Time:** 00:00:00.0240000

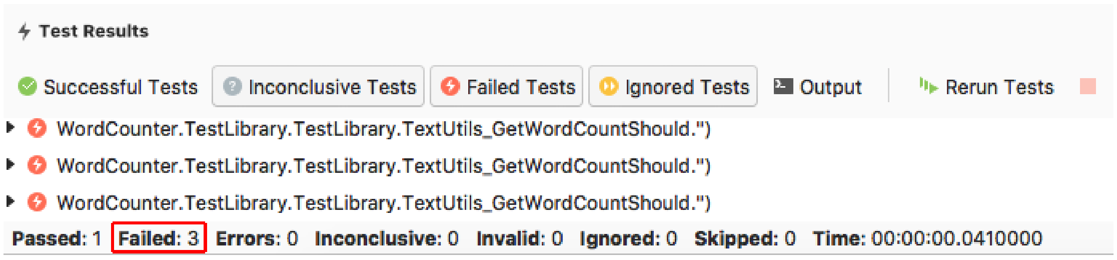
12. Testing individual return values with a `Fact` is only the beginning of what you can do with unit testing. Another powerful technique allows you to test several values at once using a `Theory`. Add the following

method to your `TextUtils_GetWordCountShould` class. You have two methods in the class after you add this method:

```
[Theory]
[InlineData(0, "Ting", "Does not appear in the string.")]
[InlineData(1, "Ting", "Ting appears once.")]
[InlineData(2, "Ting", "Ting appears twice with Ting.")]
public void CountInstancesCorrectly(int count,
                                    string searchWord,
                                    string inputString)
{
    Assert.NotEqual(count, WordCount.GetWordCount(searchWord,
                                                   inputString));
}
```

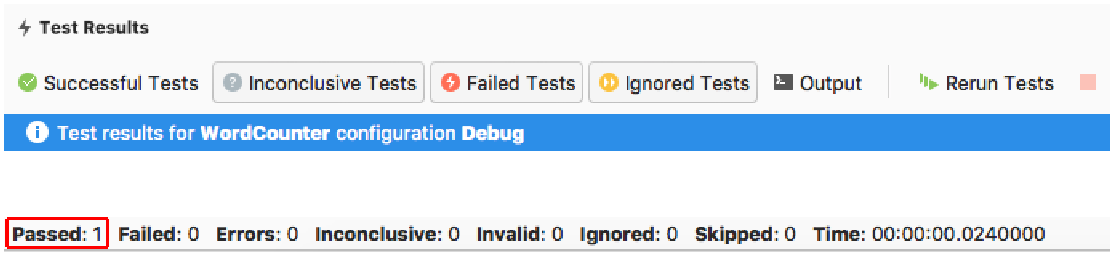
The `CountInstancesCorrectly` checks that the `GetWordCount` method counts correctly. The `InlineData` provides a count, a search word, and an input string to check. The test method runs once for each line of data. Note once again that you're asserting a failure first by using `Assert.NotEqual`, even when you know that the counts in the data are correct and that the values match the counts returned by the `GetWordCount` method. Performing the step of failing the test on purpose might seem like a waste of time at first, but checking the logic of the test by failing it first is an important check on the logic of your tests. When you come across a test method that passes when you expect it to fail, you've found a bug in the logic of the test. It's worth the effort to take this step every time you create a test method.

13. Save the file and run the tests again. The casing test passes but the three count tests fail. This is exactly what you expect to happen.



```
WordCounter.TestLibrary.TestLibrary.TextUtils_GetWordCountShould.")
WordCounter.TestLibrary.TestLibrary.TextUtils_GetWordCountShould.")
WordCounter.TestLibrary.TestLibrary.TextUtils_GetWordCountShould.")
Passed: 1  Failed: 3  Errors: 0  Inconclusive: 0  Invalid: 0  Ignored: 0  Skipped: 0  Time: 00:00:00.0410000
```

14. Modify the `CountInstancesCorrectly` test method by changing `Assert.NotEqual` to `Assert.Equal`. Save the file. Run the tests again. All tests pass.



```
Test results for WordCounter configuration Debug
Passed: 1  Failed: 0  Errors: 0  Inconclusive: 0  Invalid: 0  Ignored: 0  Skipped: 0  Time: 00:00:00.0240000
```

Adding a console app

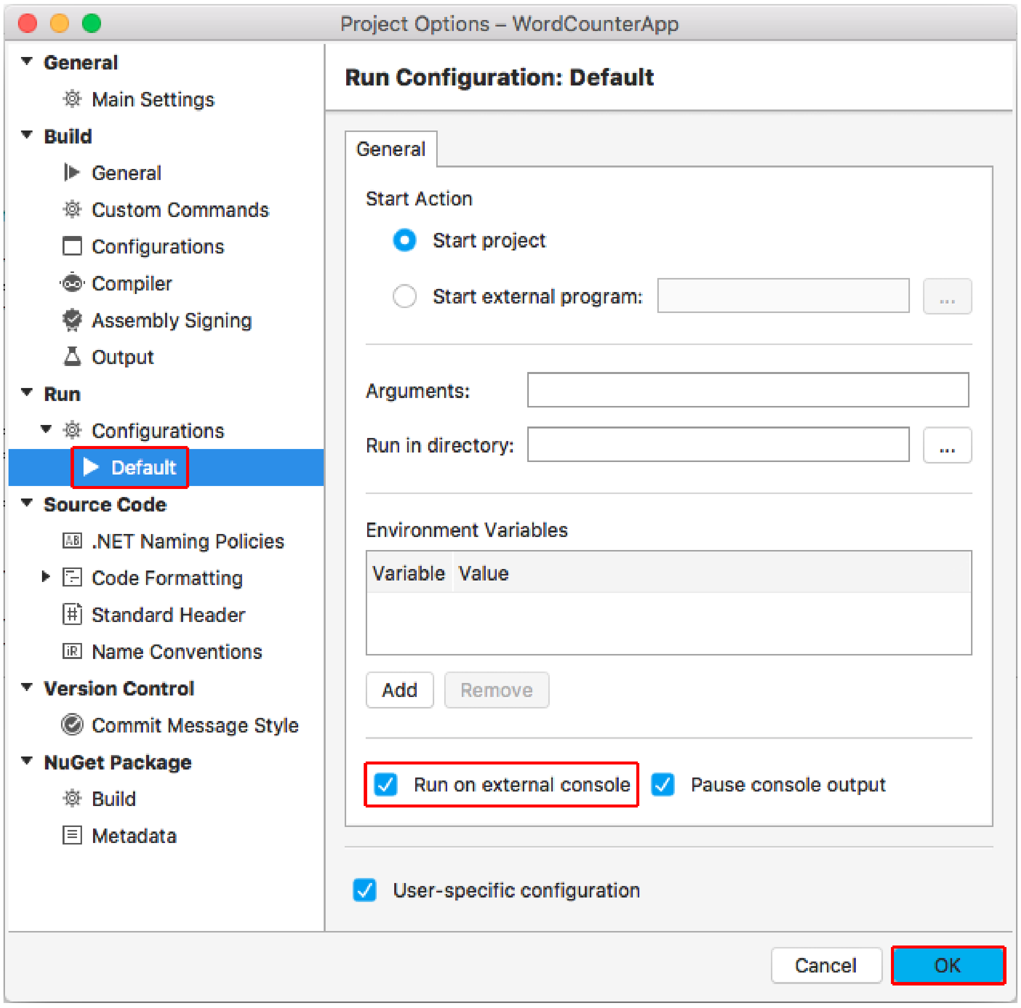
1. In the **Solution** sidebar, right-click the `WordCounter` solution. Add a new **Console Application** project by selecting the template from the **.NET Core** > **App** templates. Select **Next**. Name the project **WordCounterApp**. Select **Create** to create the project in the solution.

2. In the **Solutions** sidebar, right-click the **Dependencies** node of the new **WordCounterApp** project. In the **Edit References** dialog, check **TextUtils** and select **OK**.

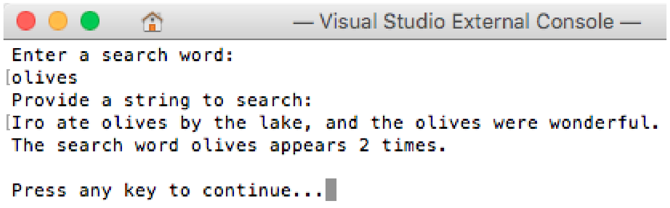
3. Open the *Program.cs* file. Replace the code with the following:

```
using System;
using TextUtils;

namespace WordCounterApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Enter a search word:");
            var searchWord = Console.ReadLine();
            Console.WriteLine("Provide a string to search:");
            var inputString = Console.ReadLine();

            var wordCount = WordCount.GetWordCount(searchWord, inputString);

            var pluralChar = "s";
            if (wordCount == 1)
            {
                pluralChar = string.Empty;
            }

            Console.WriteLine($"The search word {searchWord} appears " +
                              $"{wordCount} time{pluralChar}.");
        }
    }
}
```

4. To run the app in a console window instead of the IDE, right-click the `WordCounterApp` project, select **Options**, and open the **Default** node under **Configurations**. Check the box for **Run on external console**. Leave the **Pause console output** option checked. This setting causes the app to spawn in a console window so that you can type input for the `Console.ReadLine` statements. If you leave the app to run in the IDE, you can only see the output of `Console.WriteLine` statements. `Console.ReadLine` statements do not work in the IDE's **Application Output** panel.

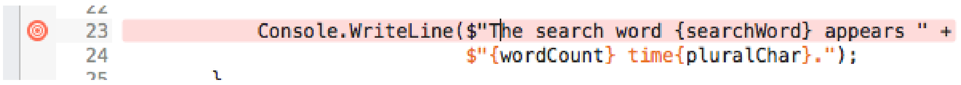


5. Because the current version of Visual Studio for Mac cannot run the tests when the solution is run, you run the console app directly. Right-click on the `WordCounterApp` project and select **Run item** from the context menu. If you attempt to run the app with the Play button, the test runner and app fail to run. For more information on the status of the work on this issue, see xunit/xamarinstudio.xunit (#60). When you run the app, provide values for the search word and input string at the prompts in the console window. The app indicates the number of times the search word appears in the string.



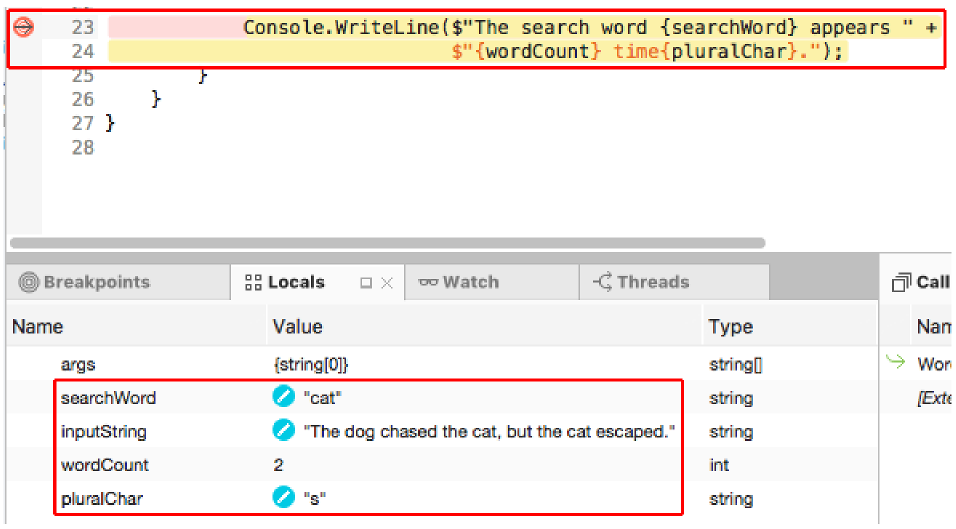
```
Enter a search word:
olives
Provide a string to search:
Iro ate olives by the lake, and the olives were wonderful.
The search word olives appears 2 times.

Press any key to continue...
```

6. The last feature to explore is debugging with Visual Studio for Mac. Set a breakpoint on the `Console.WriteLine` statement: Select in the left margin of line 23, and you see a red circle appear next to the line of code. Alternatively, select anywhere on the line of code and select **Run** > **Toggle Breakpoint** from the menu.



```
23            Console.WriteLine($"The search word {searchWord} appears " +
24                              $"{wordCount} time{pluralChar}.");
```

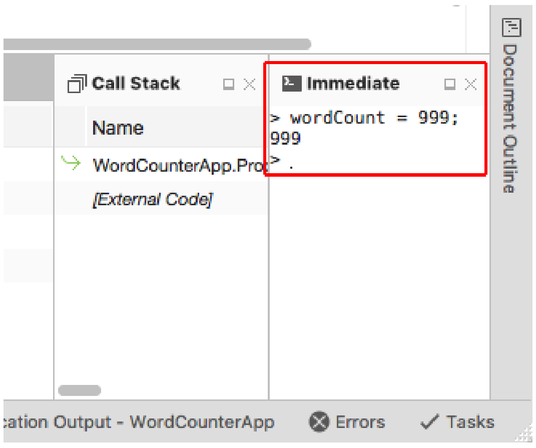
7. Right-click the `WordCounterApp` project. Select **Start Debugging item** from the context menu. When the app runs, enter the search word "cat" and "The dog chased the cat, but the cat escaped." for the string to search. When the `Console.WriteLine` statement is reached, program execution halts before the statement is executed. In the **Locals** tab, you can see the `searchWord`, `inputString`, `wordCount`, and `pluralChar` values.



```
23                Console.WriteLine($"The search word {searchWord} appears " +
24                                  $"{wordCount} time{pluralChar}.");
25            }
26        }
27 }
28
```

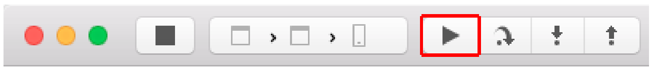
Name	Value	Type
args	{string[0]}	string[]
searchWord	"cat"	string
inputString	"The dog chased the cat, but the cat escaped."	string
wordCount	2	int
pluralChar	"s"	string

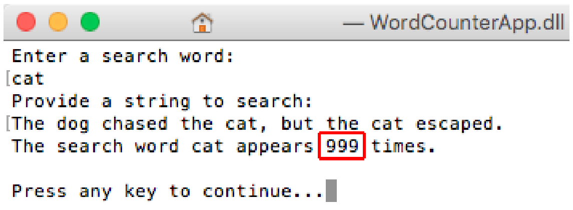
8. In the **Immediate** pane, type "wordCount = 999;" and press Enter. This assigns a nonsense value of 999 to the `wordCount` variable showing that you can replace variable values while debugging.



```
> wordCount = 999;
999
> .
```

9. In the toolbar, click the*continue* arrow. Look at the output in the console window. It reports the incorrect value of 999 that you set when you were debugging the app.





```
Enter a search word:
cat
Provide a string to search:
The dog chased the cat, but the cat escaped.
The search word cat appears 999 times.

Press any key to continue...
```

See also

- Visual Studio 2017 for Mac Release Notes

Get started with .NET Core on Windows/Linux/macOS using the command line

11/7/2019 • 5 minutes to read • Edit Online

This topic will show you how to start developing cross-platforms apps in your machine using the .NET Core CLI tools.

If you're unfamiliar with the .NET Core CLI toolset, read the .NET Core SDK overview.

Prerequisites

- .NET Core SDK 2.1 or later versions.
- A text editor or code editor of your choice.

Hello, Console App!

You can view or download the sample code from the dotnet/samples GitHub repository. For download instructions, see Samples and Tutorials.

Open a command prompt and create a folder named *Hello*. Navigate to the folder you created and type the following:

```
dotnet new console
dotnet run
```

Let's do a quick walkthrough:

1. `dotnet new console`

   `dotnet new` creates an up-to-date *Hello.csproj* project file with the dependencies necessary to build a console app. It also creates a *Program.cs*, a basic file containing the entry point for the application.

   Hello.csproj:

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

     <PropertyGroup>
       <OutputType>Exe</OutputType>
       <TargetFramework>netcoreapp2.2</TargetFramework>
     </PropertyGroup>

   </Project>
   ```

   The project file specifies everything that's needed to restore dependencies and build the program.

   - The `OutputType` tag specifies that we're building an executable, in other words a console application.
   - The `TargetFramework` tag specifies what .NET implementation we're targeting. In an advanced scenario, you can specify multiple target frameworks and build to all those in a single operation. In this tutorial, we'll stick to building only for .NET Core 2.1.

   Program.cs:

```
using System;

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

The program starts by `using System`, which means "bring everything in the `System` namespace into scope for this file". The `System` namespace includes the `Console` class.

We then define a namespace called `Hello`. You can change this to anything you want. A class named `Program` is defined within that namespace, with a `Main` method that takes an array of strings as its argument. This array contains the list of arguments passed in when the compiled program is called. As it is, this array is not used: all the program is doing is to write "Hello World!" to the console. Later, we'll make changes to the code that will make use of this argument.

NOTE

Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

`dotnet new` calls `dotnet restore` implicitly. `dotnet restore` calls into NuGet (.NET package manager) to restore the tree of dependencies. NuGet analyzes the *Hello.csproj* file, downloads the dependencies defined in the file (or grabs them from a cache on your machine), and writes the *obj/project.assets.json* file, which is necessary to compile and run the sample.

IMPORTANT

If you're using a .NET Core 1.x version of the SDK, you'll have to call `dotnet restore` yourself after calling `dotnet new`.

2. `dotnet run`

   `dotnet run` calls `dotnet build` to ensure that the build targets have been built, and then calls `dotnet <assembly.dll>` to run the target application.

   ```
   $ dotnet run
   Hello World!
   ```

   Alternatively, you can also execute `dotnet build` to compile the code without running the build console applications. This results in a compiled application as a DLL file that can be run with `dotnet bin\Debug\netcoreapp2.1\Hello.dll` on Windows (use `/` for non-Windows systems). You may also specify arguments to the application as you'll see later on the topic.

```
$ dotnet bin\Debug\netcoreapp2.1\Hello.dll
Hello World!
```

As an advanced scenario, it's possible to build the application as a self-contained set of platform-specific files that can be deployed and run to a machine that doesn't necessarily have .NET Core installed. See .NET Core Application Deployment for details.

Augmenting the program

Let's change the program a bit. Fibonacci numbers are fun, so let's add that in addition to use the argument to greet the person running the app.

1. Replace the contents of your *Program.cs* file with the following code:

```
using System;

namespace Hello
{
    class Program
    {
        static void Main(string[] args)
        {
            if (args.Length > 0)
            {
                Console.WriteLine($"Hello {args[0]}!");
            }
            else
            {
                Console.WriteLine("Hello!");
            }

            Console.WriteLine("Fibonacci Numbers 1-15:");

            for (int i = 0; i < 15; i++)
            {
                Console.WriteLine($"{i + 1}: {FibonacciNumber(i)}");
            }
        }

        static int FibonacciNumber(int n)
        {
            int a = 0;
            int b = 1;
            int tmp;

            for (int i = 0; i < n; i++)
            {
                tmp = a;
                a = b;
                b += tmp;
            }

            return a;
        }

    }
}
```

2. Execute `dotnet build` to compile the changes.

3. Run the program passing a parameter to the app:

```
$ dotnet run -- John
Hello John!
Fibonacci Numbers 1-15:
1: 0
2: 1
3: 1
4: 2
5: 3
6: 5
7: 8
8: 13
9: 21
10: 34
11: 55
12: 89
13: 144
14: 233
15: 377
```

And that's it! You can augment *Program.cs* any way you like.

Working with multiple files

Single files are fine for simple one-off programs, but if you're building a more complex app, you're probably going to have multiple source files on your project. Let's build off of the previous Fibonacci example by caching some Fibonacci values and add some recursive features.

1. Add a new file inside the *Hello* directory named *FibonacciGenerator.cs* with the following code:

```
using System;
using System.Collections.Generic;

namespace Hello
{
    public class FibonacciGenerator
    {
        private Dictionary<int, int> _cache = new Dictionary<int, int>();

        private int Fib(int n) => n < 2 ? n : FibValue(n - 1) + FibValue(n - 2);

        private int FibValue(int n)
        {
            if (!_cache.ContainsKey(n))
            {
                _cache.Add(n, Fib(n));
            }

            return _cache[n];
        }

        public IEnumerable<int> Generate(int n)
        {
            for (int i = 0; i < n; i++)
            {
                yield return FibValue(i);
            }
        }
    }
}
```

2. Change the `Main` method in your *Program.cs* file to instantiate the new class and call its method as in the following example:

```
using System;

namespace Hello
{
    class Program
    {
        static void Main(string[] args)
        {
            var generator = new FibonacciGenerator();
            foreach (var digit in generator.Generate(15))
            {
                Console.WriteLine(digit);
            }
        }
    }
}
```

3. Execute `dotnet build` to compile the changes.
4. Run your app by executing `dotnet run`. The following shows the program output:

```
$ dotnet run
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
```

Publish your app

Once you're ready to distribute your app, use the `dotnet publish` command to generate the *publish* folder at *bin\debug\netcoreapp2.1\publish\* (use `/` for non-Windows systems). You can distribute the contents of the *publish* folder to other platforms as long as they've already installed the dotnet runtime.

You can run your published app with the dotnet command:

```
$ dotnet bin\Debug\netcoreapp2.1\publish\Hello.dll
Hello World!
```

Conclusion

And that's it! Now, you can start using the basic concepts learned here to create your own programs.

See also

- Organizing and testing projects with the .NET Core CLI tools
- Publish .NET Core apps with the CLI
- Learn more about app deployment

Organizing and testing projects with the .NET Core command line

9/19/2019 • 6 minutes to read • Edit Online

This tutorial follows Get started with .NET Core on Windows/Linux/macOS using the command line, taking you beyond the creation of a simple console app to develop advanced and well-organized applications. After showing you how to use folders to organize your code, this tutorial shows you how to extend a console application with the xUnit testing framework.

Using folders to organize code

If you want to introduce new types into a console app, you can do so by adding files containing the types to the app. For example if you add files containing `AccountInformation` and `MonthlyReportRecords` types to your project, the project file structure is flat and easy to navigate:

```
/MyProject
|__AccountInformation.cs
|__MonthlyReportRecords.cs
|__MyProject.csproj
|__Program.cs
```

However, this only works well when the size of your project is relatively small. Can you imagine what will happen if you add 20 types to the project? The project definitely wouldn't be easy to navigate and maintain with that many files littering the project's root directory.

To organize the project, create a new folder and name it *Models* to hold the type files. Place the type files into the *Models* folder:

```
/MyProject
|__/Models
   |__AccountInformation.cs
   |__MonthlyReportRecords.cs
|__MyProject.csproj
|__Program.cs
```

Projects that logically group files into folders are easy to navigate and maintain. In the next section, you create a more complex sample with folders and unit testing.

Organizing and testing using the NewTypes Pets Sample

Building the sample

For the following steps, you can either follow along using the NewTypes Pets Sample or create your own files and folders. The types are logically organized into a folder structure that permits the addition of more types later, and tests are also logically placed in folders permitting the addition of more tests later.

The sample contains two types, `Dog` and `Cat`, and has them implement a common interface, `IPet`. For the `NewTypes` project, your goal is to organize the pet-related types into a *Pets* folder. If another set of types is added later, *WildAnimals* for example, they're placed in the *NewTypes* folder alongside the *Pets* folder. The *WildAnimals* folder may contain types for animals that aren't pets, such as `Squirrel` and `Rabbit` types. In this way as types are added, the project remains well organized.

Create the following folder structure with file content indicated:

```
/NewTypes
|__/src
   |__/NewTypes
      |__/Pets
         |__Dog.cs
         |__Cat.cs
         |__IPet.cs
      |__Program.cs
      |__NewTypes.csproj
```

IPet.cs:

```
using System;

namespace Pets
{
    public interface IPet
    {
        string TalkToOwner();
    }
}
```

Dog.cs:

```
using System;

namespace Pets
{
    public class Dog : IPet
    {
        public string TalkToOwner() => "Woof!";
    }
}
```

Cat.cs:

```
using System;

namespace Pets
{
    public class Cat : IPet
    {
        public string TalkToOwner() => "Meow!";
    }
}
```

Program.cs:

```
using System;
using Pets;
using System.Collections.Generic;

namespace ConsoleApplication
{
    public class Program
    {
        public static void Main(string[] args)
        {
            List<IPet> pets = new List<IPet>
            {
                new Dog(),
                new Cat()
            };

            foreach (var pet in pets)
            {
                Console.WriteLine(pet.TalkToOwner());
            }
        }
    }
}
```

NewTypes.csproj:

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.2</TargetFramework>
  </PropertyGroup>

</Project>
```

Execute the following command:

```
dotnet run
```

Obtain the following output:

```
Woof!
Meow!
```

Optional exercise: You can add a new pet type, such as a `Bird`, by extending this project. Make the bird's `TalkToOwner` method give a `Tweet!` to the owner. Run the app again. The output will include `Tweet!`

Testing the sample

The `NewTypes` project is in place, and you've organized it by keeping the pets-related types in a folder. Next, create your test project and start writing tests with the xUnit test framework. Unit testing allows you to automatically check the behavior of your pet types to confirm that they're operating properly.

Navigate back to the *src* folder and create a *test* folder with a *NewTypesTests* folder within it. At a command prompt from the *NewTypesTests* folder, execute `dotnet new xunit`. This produces two files: *NewTypesTests.csproj* and *UnitTest1.cs*.

The test project cannot currently test the types in `NewTypes` and requires a project reference to the `NewTypes` project. To add a project reference, use the `dotnet add reference` command:

```
dotnet add reference ../../src/NewTypes/NewTypes.csproj
```

Or, you also have the option of manually adding the project reference by adding an `<ItemGroup>` node to the *NewTypesTests.csproj* file:

```
<ItemGroup>
  <ProjectReference Include="../../src/NewTypes/NewTypes.csproj" />
</ItemGroup>
```

NewTypesTests.csproj:

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

  <PropertyGroup>
    <TargetFramework>netcoreapp2.2</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.4.0" />
    <PackageReference Include="xunit" Version="2.4.1" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.4.1" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="../../src/NewTypes/NewTypes.csproj"/>
  </ItemGroup>

</Project>
```

The *NewTypesTests.csproj* file contains the following:

- Package reference to `Microsoft.NET.Test.Sdk`, the .NET testing infrastructure
- Package reference to `xunit`, the xUnit testing framework
- Package reference to `xunit.runner.visualstudio`, the test runner
- Project reference to `NewTypes`, the code to test

Change the name of *UnitTest1.cs* to *PetTests.cs* and replace the code in the file with the following:

```
using System;
using Xunit;
using Pets;

public class PetTests
{
    [Fact]
    public void DogTalkToOwnerReturnsWoof()
    {
        string expected = "Woof!";
        string actual = new Dog().TalkToOwner();

        Assert.NotEqual(expected, actual);
    }

    [Fact]
    public void CatTalkToOwnerReturnsMeow()
    {
        string expected = "Meow!";
        string actual = new Cat().TalkToOwner();

        Assert.NotEqual(expected, actual);
    }
}
```

Optional exercise: If you added a `Bird` type earlier that yields a `Tweet!` to the owner, add a test method to the *PetTests.cs* file, `BirdTalkToOwnerReturnsTweet`, to check that the `TalkToOwner` method works correctly for the `Bird` type.

NOTE

Although you expect that the `expected` and `actual` values are equal, an initial assertion with the `Assert.NotEqual` check specifies that these values are *not equal*. Always initially create a test to fail in order to check the logic of the test. After you confirm that the test fails, adjust the assertion to allow the test to pass.

The following shows the complete project structure:

```
/NewTypes
|__/src
   |__/NewTypes
      |__/Pets
         |__Dog.cs
         |__Cat.cs
         |__IPet.cs
      |__Program.cs
      |__NewTypes.csproj
|__/test
   |__NewTypesTests
      |__PetTests.cs
      |__NewTypesTests.csproj
```

Start in the *test/NewTypesTests* directory. Restore the test project with the `dotnet restore` command. Run the tests with the `dotnet test` command. This command starts the test runner specified in the project file.

NOTE

Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

As expected, testing fails, and the console displays the following output:

```
Test run for c:\Users\ronpet\repos\samples\core\console-
apps\NewTypesMsBuild\test\NewTypesTests\bin\Debug\netcoreapp2.1\NewTypesTests.dll(.NETCoreApp,Version=v2.1)
Microsoft (R) Test Execution Command Line Tool Version 15.8.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
[xUnit.net 00:00:00.77]     PetTests.DogTalkToOwnerReturnsWoof [FAIL]
[xUnit.net 00:00:00.78]     PetTests.CatTalkToOwnerReturnsMeow [FAIL]
Failed   PetTests.DogTalkToOwnerReturnsWoof
Error Message:
 Assert.NotEqual() Failure
Expected: Not "Woof!"
Actual:   "Woof!"
Stack Trace:
   at PetTests.DogTalkToOwnerReturnsWoof() in c:\Users\ronpet\repos\samples\core\console-
apps\NewTypesMsBuild\test\NewTypesTests\PetTests.cs:line 13
Failed   PetTests.CatTalkToOwnerReturnsMeow
Error Message:
 Assert.NotEqual() Failure
Expected: Not "Meow!"
Actual:   "Meow!"
Stack Trace:
   at PetTests.CatTalkToOwnerReturnsMeow() in c:\Users\ronpet\repos\samples\core\console-
apps\NewTypesMsBuild\test\NewTypesTests\PetTests.cs:line 22

Total tests: 2. Passed: 0. Failed: 2. Skipped: 0.
Test Run Failed.
Test execution time: 1.7000 Seconds
```

Change the assertions of your tests from `Assert.NotEqual` to `Assert.Equal`:

```
using System;
using Xunit;
using Pets;

public class PetTests
{
    [Fact]
    public void DogTalkToOwnerReturnsWoof()
    {
        string expected = "Woof!";
        string actual = new Dog().TalkToOwner();

        Assert.Equal(expected, actual);
    }

    [Fact]
    public void CatTalkToOwnerReturnsMeow()
    {
        string expected = "Meow!";
        string actual = new Cat().TalkToOwner();

        Assert.Equal(expected, actual);
    }
}
```

Re-run the tests with the `dotnet test` command and obtain the following output:

```
Test run for c:\Users\ronpet\repos\samples\core\console-
apps\NewTypesMsBuild\test\NewTypesTests\bin\Debug\netcoreapp2.1\NewTypesTests.dll(.NETCoreApp,Version=v2.1)
Microsoft (R) Test Execution Command Line Tool Version 15.8.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...

Total tests: 2. Passed: 2. Failed: 0. Skipped: 0.
Test Run Successful.
Test execution time: 1.6029 Seconds
```

Testing passes. The pet types' methods return the correct values when talking to the owner.

You've learned techniques for organizing and testing projects using xUnit. Go forward with these techniques applying them in your own projects. *Happy coding!*

Developing Libraries with Cross Platform Tools

11/1/2019 • 11 minutes to read • Edit Online

This article covers how to write libraries for .NET using cross-platform CLI tools. The CLI provides an efficient and low-level experience that works across any supported OS. You can still build libraries with Visual Studio, and if that is your preferred experience refer to the Visual Studio guide.

Prerequisites

You need the .NET Core SDK and CLI installed on your machine.

For the sections of this document dealing with .NET Framework versions, you need the .NET Framework installed on a Windows machine.

Additionally, if you wish to support older .NET Framework targets, you need to install targeting/developer packs for older framework versions from the .NET download archives page. Refer to this table:

.NET FRAMEWORK VERSION	WHAT TO DOWNLOAD
4.6.1	.NET Framework 4.6.1 Targeting Pack
4.6	.NET Framework 4.6 Targeting Pack
4.5.2	.NET Framework 4.5.2 Developer Pack
4.5.1	.NET Framework 4.5.1 Developer Pack
4.5	Windows Software Development Kit for Windows 8
4.0	Windows SDK for Windows 7 and .NET Framework 4
2.0, 3.0, and 3.5	.NET Framework 3.5 SP1 Runtime (or Windows 8+ version)

How to target the .NET Standard

If you're not quite familiar with the .NET Standard, refer to the .NET Standard to learn more.

In that article, there is a table which maps .NET Standard versions to various implementations:

.NET STANDARD	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0	2.1
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	3.0
.NET Framework [1]	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1 [2]	4.6.1 [2]	4.6.1 [2]	N/A[3]
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4	6.4

.NET STANDARD	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0	2.1
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

1 The versions listed for .NET Framework apply to .NET Core 2.0 SDK and later versions of the tooling. Older versions used a different mapping for .NET Standard 1.5 and higher. You can download tooling for .NET Core tools for Visual Studio 2015 if you cannot upgrade to Visual Studio 2017.

2 The versions listed here represent the rules that NuGet uses to determine whether a given .NET Standard library is applicable. While NuGet considers .NET Framework 4.6.1 as supporting .NET Standard 1.5 through 2.0, there are several issues with consuming .NET Standard libraries that were built for those versions from .NET Framework 4.6.1 projects. For .NET Framework projects that need to use such libraries, we recommend that you upgrade the project to target .NET Framework 4.7.2 or higher.

3 .NET Framework won't support .NET Standard 2.1 or later versions. For more details, see the announcement of .NET Standard 2.1.

- The columns represent .NET Standard versions. Each header cell is a link to a document that shows which APIs got added in that version of .NET Standard.
- The rows represent the different .NET implementations.
- The version number in each cell indicates the *minimum* version of the implementation you'll need in order to target that .NET Standard version.
- For an interactive table, see .NET Standard versions.

Here's what this table means for the purposes of creating a library:

The version of the .NET Standard you pick will be a tradeoff between access to the newest APIs and the ability to target more .NET implementations and .NET Standard versions. You control the range of targetable platforms and versions by picking a version of `netstandardX.X` (Where `X.X` is a version number) and adding it to your project file (`.csproj` or `.fsproj`).

You have three primary options when targeting the .NET Standard, depending on your needs.

1. You can use the default version of the .NET Standard supplied by templates - `netstandard1.4` - which gives you access to most APIs on .NET Standard while still being compatible with UWP, .NET Framework 4.6.1, and the forthcoming .NET Standard 2.0.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard1.4</TargetFramework>
  </PropertyGroup>
</Project>
```

2. You can use a lower or higher version of the .NET Standard by modifying the value in the `TargetFramework` node of your project file.

   .NET Standard versions are backward compatible. That means that `netstandard1.0` libraries run on `netstandard1.1` platforms and higher. However, there is no forward compatibility - lower .NET Standard platforms cannot reference higher ones. This means that `netstandard1.0` libraries cannot reference libraries targeting `netstandard1.1` or higher. Select the Standard version that has the right mix of APIs and platform support for your needs. We recommend `netstandard1.4` for now.

3. If you want to target the .NET Framework versions 4.0 or below, or you wish to use an API available in the .NET Framework but not in the .NET Standard (for example, `System.Drawing`), read the following sections and learn how to multitarget.

How to target the .NET Framework

NOTE

These instructions assume you have the .NET Framework installed on your machine. Refer to the Prerequisites to get dependencies installed.

Keep in mind that some of the .NET Framework versions used here are no longer in support. Refer to the .NET Framework Support Lifecycle Policy FAQ about unsupported versions.

If you want to reach the maximum number of developers and projects, use the .NET Framework 4.0 as your baseline target. To target the .NET Framework, you will need to begin by using the correct Target Framework Moniker (TFM) that corresponds to the .NET Framework version you wish to support.

.NET FRAMEWORK VERSION	TFM
.NET Framework 2.0	`net20`
.NET Framework 3.0	`net30`
.NET Standard 3.5	`net35`
.NET Framework 4.0	`net40`
.NET Framework 4.5	`net45`
.NET Framework 4.5.1	`net451`
.NET Framework 4.5.2	`net452`
.NET Framework 4.6	`net46`
.NET Framework 4.6.1	`net461`
.NET Framework 4.6.2	`net462`
.NET Framework 4.7	`net47`
.NET Framework 4.8	`net48`

You then insert this TFM into the `TargetFramework` section of your project file. For example, here's how you would write a library which targets the .NET Framework 4.0:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net40</TargetFramework>
  </PropertyGroup>
</Project>
```

And that's it! Although this compiled only for the .NET Framework 4, you can use the library on newer versions of the .NET Framework.

How to Multitarget

NOTE

The following instructions assume you have the .NET Framework installed on your machine. Refer to the Prerequisites section to learn which dependencies you need to install and where to download them from.

You may need to target older versions of the .NET Framework when your project supports both the .NET Framework and .NET Core. In this scenario, if you want to use newer APIs and language constructs for the newer targets, use `#if` directives in your code. You also might need to add different packages and dependencies for each platform you're targeting to include the different APIs needed for each case.

For example, let's say you have a library that performs networking operations over HTTP. For .NET Standard and the .NET Framework versions 4.5 or higher, you can use the `HttpClient` class from the `System.Net.Http` namespace. However, earlier versions of the .NET Framework don't have the `HttpClient` class, so you could use the `WebClient` class from the `System.Net` namespace for those instead.

Your project file could look like this:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFrameworks>netstandard1.4;net40;net45</TargetFrameworks>
  </PropertyGroup>

  <!-- Need to conditionally bring in references for the .NET Framework 4.0 target -->
  <ItemGroup Condition="'$(TargetFramework)' == 'net40'">
    <Reference Include="System.Net" />
  </ItemGroup>

  <!-- Need to conditionally bring in references for the .NET Framework 4.5 target -->
  <ItemGroup Condition="'$(TargetFramework)' == 'net45'">
    <Reference Include="System.Net.Http" />
    <Reference Include="System.Threading.Tasks" />
  </ItemGroup>
</Project>
```

You'll notice three major changes here:

1. The `TargetFramework` node has been replaced by `TargetFrameworks`, and three TFMs are expressed inside.
2. There is an `<ItemGroup>` node for the `net40` target pulling in one .NET Framework reference.
3. There is an `<ItemGroup>` node for the `net45` target pulling in two .NET Framework references.

The build system is aware of the following preprocessor symbols used in `#if` directives:

TARGET FRAMEWORKS	SYMBOLS
.NET Framework	`NETFRAMEWORK`, `NET20`, `NET35`, `NET40`, `NET45`, `NET451`, `NET452`, `NET46`, `NET461`, `NET462`, `NET47`, `NET471`, `NET472`, `NET48`
.NET Standard	`NETSTANDARD`, `NETSTANDARD1_0`, `NETSTANDARD1_1`, `NETSTANDARD1_2`, `NETSTANDARD1_3`, `NETSTANDARD1_4`, `NETSTANDARD1_5`, `NETSTANDARD1_6`, `NETSTANDARD2_0`, `NETSTANDARD2_1`
.NET Core	`NETCOREAPP`, `NETCOREAPP1_0`, `NETCOREAPP1_1`, `NETCOREAPP2_0`, `NETCOREAPP2_1`, `NETCOREAPP2_2`, `NETCOREAPP3_0`

Here is an example making use of conditional compilation per-target:

```
using System;
using System.Text.RegularExpressions;
#if NET40
// This only compiles for the .NET Framework 4 targets
using System.Net;
#else
 // This compiles for all other targets
using System.Net.Http;
using System.Threading.Tasks;
#endif

namespace MultitargetLib
{
    public class Library
    {
#if NET40
        private readonly WebClient _client = new WebClient();
        private readonly object _locker = new object();
#else
        private readonly HttpClient _client = new HttpClient();
#endif

#if NET40
        // .NET Framework 4.0 does not have async/await
        public string GetDotNetCount()
        {
            string url = "https://www.dotnetfoundation.org/";

            var uri = new Uri(url);

            string result = "";

            // Lock here to provide thread-safety.
            lock(_locker)
            {
                result = _client.DownloadString(uri);
            }

            int dotNetCount = Regex.Matches(result, ".NET").Count;

            return $"Dotnet Foundation mentions .NET {dotNetCount} times!";
        }
#else
        // .NET 4.5+ can use async/await!
        public async Task<string> GetDotNetCountAsync()
        {
            string url = "https://www.dotnetfoundation.org/";

            // HttpClient is thread-safe, so no need to explicitly lock here
            var result = await _client.GetStringAsync(url);

            int dotNetCount = Regex.Matches(result, ".NET").Count;

            return $"dotnetfoundation.org mentions .NET {dotNetCount} times in its HTML!";
        }
#endif
    }
}
```

If you build this project with `dotnet build`, you'll notice three directories under the `bin/` folder:

```
net40/
net45/
netstandard1.4/
```

Each of these contain the `.dll` files for each target.

How to test libraries on .NET Core

It's important to be able to test across platforms. You can use either xUnit or MSTest out of the box. Both are perfectly suitable for unit testing your library on .NET Core. How you set up your solution with test projects will depend on the structure of your solution. The following example assumes that the test and source directories live in the same top-level directory.

> **NOTE**
>
> This uses some .NET Core CLI commands. See dotnet new and dotnet sln for more information.

1. Set up your solution. You can do so with the following commands:

   ```
   mkdir SolutionWithSrcAndTest
   cd SolutionWithSrcAndTest
   dotnet new sln
   dotnet new classlib -o MyProject
   dotnet new xunit -o MyProject.Test
   dotnet sln add MyProject/MyProject.csproj
   dotnet sln add MyProject.Test/MyProject.Test.csproj
   ```

   This will create projects and link them together in a solution. Your directory for `SolutionWithSrcAndTest` should look like this:

   ```
   /SolutionWithSrcAndTest
   |__SolutionWithSrcAndTest.sln
   |__MyProject/
   |__MyProject.Test/
   ```

2. Navigate to the test project's directory and add a reference to `MyProject.Test` from `MyProject`.

   ```
   cd MyProject.Test
   dotnet add reference ../MyProject/MyProject.csproj
   ```

3. Restore packages and build projects:

   ```
   dotnet restore
   dotnet build
   ```

   > **NOTE**
   >
   > Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

4. Verify that xUnit runs by executing the `dotnet test` command. If you chose to use MSTest, then the MSTest console runner should run instead.

And that's it! You can now test your library across all platforms using command line tools. To continue testing now that you have everything set up, testing your library is very simple:

1. Make changes to your library.
2. Run tests from the command line, in your test directory, with `dotnet test` command.

Your code will be automatically rebuilt when you invoke `dotnet test` command.

How to use multiple projects

A common need for larger libraries is to place functionality in different projects.

Imagine you wished to build a library which could be consumed in idiomatic C# and F#. That would mean that consumers of your library consume them in ways which are natural to C# or F#. For example, in C# you might consume the library like this:

```
using AwesomeLibrary.CSharp;

public Task DoThings(Data data)
{
    var convertResult = await AwesomeLibrary.ConvertAsync(data);
    var result = AwesomeLibrary.Process(convertResult);
    // do something with result
}
```

In F#, it might look like this:

```
open AwesomeLibrary.FSharp

let doWork data = async {
    let! result = AwesomeLibrary.AsyncConvert data // Uses an F# async function rather than C# async method
    // do something with result
}
```

Consumption scenarios like this mean that the APIs being accessed have to have a different structure for C# and F#. A common approach to accomplishing this is to factor all of the logic of a library into a core project, with C# and F# projects defining the API layers that call into that core project. The rest of the section will use the following names:

- **AwesomeLibrary.Core** - A core project which contains all logic for the library
- **AwesomeLibrary.CSharp** - A project with public APIs intended for consumption in C#
- **AwesomeLibrary.FSharp** - A project with public APIs intended for consumption in F#

You can run the following commands in your terminal to produce the same structure as this guide:

```
mkdir AwesomeLibrary && cd AwesomeLibrary
dotnet new sln
mkdir AwesomeLibrary.Core && cd AwesomeLibrary.Core && dotnet new classlib
cd ..
mkdir AwesomeLibrary.CSharp && cd AwesomeLibrary.CSharp && dotnet new classlib
cd ..
mkdir AwesomeLibrary.FSharp && cd AwesomeLibrary.FSharp && dotnet new classlib -lang F#
cd ..
dotnet sln add AwesomeLibrary.Core/AwesomeLibrary.Core.csproj
dotnet sln add AwesomeLibrary.CSharp/AwesomeLibrary.CSharp.csproj
dotnet sln add AwesomeLibrary.FSharp/AwesomeLibrary.FSharp.fsproj
```

This will add the three projects above and a solution file which links them together. Creating the solution file and linking projects will allow you to restore and build projects from a top-level.

Project-to-project referencing

The best way to reference a project is to use the .NET Core CLI to add a project reference. From the **AwesomeLibrary.CSharp** and **AwesomeLibrary.FSharp** project directories, you can run the following command:

```
dotnet add reference ../AwesomeLibrary.Core/AwesomeLibrary.Core.csproj
```

The project files for both **AwesomeLibrary.CSharp** and **AwesomeLibrary.FSharp** will now reference **AwesomeLibrary.Core** as a `ProjectReference` target. You can verify this by inspecting the project files and seeing the following in them:

```
<ItemGroup>
  <ProjectReference Include="..\AwesomeLibrary.Core\AwesomeLibrary.Core.csproj" />
</ItemGroup>
```

You can add this section to each project file manually if you prefer not to use the .NET Core CLI.

Structuring a solution

Another important aspect of multi-project solutions is establishing a good overall project structure. You can organize code however you like, and as long as you link each project to your solution file with `dotnet sln add`, you will be able to run `dotnet restore` and `dotnet build` at the solution level.

Create a .NET Core application with plugins

11/7/2019 • 8 minutes to read • Edit Online

This tutorial shows you how to create a custom AssemblyLoadContext to load plugins. An AssemblyDependencyResolver is used to resolve the dependencies of the plugin. The tutorial correctly isolates the plugin's dependencies from the hosting application. You'll learn how to:

- Structure a project to support plugins.
- Create a custom AssemblyLoadContext to load each plugin.
- Use the System.Runtime.Loader.AssemblyDependencyResolver type to allow plugins to have dependencies.
- Author plugins that can be easily deployed by just copying the build artifacts.

Prerequisites

- Install the .NET Core 3.0 SDK or a newer version.

Create the application

The first step is to create the application:

1. Create a new folder, and in that folder run the following command:

   ```
   dotnet new console -o AppWithPlugin
   ```

2. To make building the project easier, create a Visual Studio solution file in the same folder. Run the following command:

   ```
   dotnet new sln
   ```

3. Run the following command to add the app project to the solution:

   ```
   dotnet sln add AppWithPlugin/AppWithPlugin.csproj
   ```

Now we can fill in the skeleton of our application. Replace the code in the *AppWithPlugin/Program.cs* file with the following code:

```
using PluginBase;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Reflection;

namespace AppWithPlugin
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                if (args.Length == 1 && args[0] == "/d")
                {
                    Console.WriteLine("Waiting for any key...");
                    Console.ReadLine();
                }

                // Load commands from plugins.

                if (args.Length == 0)
                {
                    Console.WriteLine("Commands: ");
                    // Output the loaded commands.
                }
                else
                {
                    foreach (string commandName in args)
                    {
                        Console.WriteLine($"-- {commandName} --");

                        // Execute the command with the name passed as an argument.

                        Console.WriteLine();
                    }
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex);
            }
        }
    }
}
```

Create the plugin interfaces

The next step in building an app with plugins is defining the interface the plugins need to implement. We suggest that you make a class library that contains any types that you plan to use for communicating between your app and plugins. This division allows you to publish your plugin interface as a package without having to ship your full application.

In the root folder of the project, run `dotnet new classlib -o PluginBase`. Also, run `dotnet sln add PluginBase/PluginBase.csproj` to add the project to the solution file. Delete the `PluginBase/Class1.cs` file, and create a new file in the `PluginBase` folder named `ICommand.cs` with the following interface definition:

```
namespace PluginBase
{
    public interface ICommand
    {
        string Name { get; }
        string Description { get; }

        int Execute();
    }
}
```

This `ICommand` interface is the interface that all of the plugins will implement.

Now that the `ICommand` interface is defined, the application project can be filled in a little more. Add a reference from the `AppWithPlugin` project to the `PluginBase` project with the `dotnet add AppWithPlugin\AppWithPlugin.csproj reference PluginBase\PluginBase.csproj` command from the root folder.

Replace the `// Load commands from plugins` comment with the following code snippet to enable it to load plugins from given file paths:

```
string[] pluginPaths = new string[]
{
    // Paths to plugins to load.
};

IEnumerable<ICommand> commands = pluginPaths.SelectMany(pluginPath =>
{
    Assembly pluginAssembly = LoadPlugin(pluginPath);
    return CreateCommands(pluginAssembly);
}).ToList();
```

Then replace the `// Output the loaded commands` comment with the following code snippet:

```
foreach (ICommand command in commands)
{
    Console.WriteLine($"{command.Name}\t - {command.Description}");
}
```

Replace the `// Execute the command with the name passed as an argument` comment with the following snippet:

```
ICommand command = commands.FirstOrDefault(c => c.Name == commandName);
if (command == null)
{
    Console.WriteLine("No such command is known.");
    return;
}

command.Execute();
```

And finally, add static methods to the `Program` class named `LoadPlugin` and `CreateCommands`, as shown here:

```
static Assembly LoadPlugin(string relativePath)
{
    throw new NotImplementedException();
}

static IEnumerable<ICommand> CreateCommands(Assembly assembly)
{
    int count = 0;

    foreach (Type type in assembly.GetTypes())
    {
        if (typeof(ICommand).IsAssignableFrom(type))
        {
            ICommand result = Activator.CreateInstance(type) as ICommand;
            if (result != null)
            {
                count++;
                yield return result;
            }
        }
    }

    if (count == 0)
    {
        string availableTypes = string.Join(",", assembly.GetTypes().Select(t => t.FullName));
        throw new ApplicationException(
            $"Can't find any type which implements ICommand in {assembly} from {assembly.Location}.\n" +
            $"Available types: {availableTypes}");
    }
}
```

Load plugins

Now the application can correctly load and instantiate commands from loaded plugin assemblies, but it's still unable to load the plugin assemblies. Create a file named *PluginLoadContext.cs* in the *AppWithPlugin* folder with the following contents:

```
using System;
using System.Reflection;
using System.Runtime.Loader;

namespace AppWithPlugin
{
    class PluginLoadContext : AssemblyLoadContext
    {
        private AssemblyDependencyResolver _resolver;

        public PluginLoadContext(string pluginPath)
        {
            _resolver = new AssemblyDependencyResolver(pluginPath);
        }

        protected override Assembly Load(AssemblyName assemblyName)
        {
            string assemblyPath = _resolver.ResolveAssemblyToPath(assemblyName);
            if (assemblyPath != null)
            {
                return LoadFromAssemblyPath(assemblyPath);
            }

            return null;
        }

        protected override IntPtr LoadUnmanagedDll(string unmanagedDllName)
        {
            string libraryPath = _resolver.ResolveUnmanagedDllToPath(unmanagedDllName);
            if (libraryPath != null)
            {
                return LoadUnmanagedDllFromPath(libraryPath);
            }

            return IntPtr.Zero;
        }
    }
}
```

The `PluginLoadContext` type derives from AssemblyLoadContext. The `AssemblyLoadContext` type is a special type in the runtime that allows developers to isolate loaded assemblies into different groups to ensure that assembly versions don't conflict. Additionally, a custom `AssemblyLoadContext` can choose different paths to load assemblies from and override the default behavior. The `PluginLoadContext` uses an instance of the `AssemblyDependencyResolver` type introduced in .NET Core 3.0 to resolve assembly names to paths. The `AssemblyDependencyResolver` object is constructed with the path to a .NET class library. It resolves assemblies and native libraries to their relative paths based on the *.deps.json* file for the class library whose path was passed to the `AssemblyDependencyResolver` constructor. The custom `AssemblyLoadContext` enables plugins to have their own dependencies, and the `AssemblyDependencyResolver` makes it easy to correctly load the dependencies.

Now that the `AppWithPlugin` project has the `PluginLoadContext` type, update the `Program.LoadPlugin` method with the following body:

```
static Assembly LoadPlugin(string relativePath)
{
    // Navigate up to the solution root
    string root = Path.GetFullPath(Path.Combine(
        Path.GetDirectoryName(
            Path.GetDirectoryName(
                Path.GetDirectoryName(
                    Path.GetDirectoryName(
                        Path.GetDirectoryName(typeof(Program).Assembly.Location)))))));

    string pluginLocation = Path.GetFullPath(Path.Combine(root, relativePath.Replace('\\',
Path.DirectorySeparatorChar)));
    Console.WriteLine($"Loading commands from: {pluginLocation}");
    PluginLoadContext loadContext = new PluginLoadContext(pluginLocation);
    return loadContext.LoadFromAssemblyName(new
AssemblyName(Path.GetFileNameWithoutExtension(pluginLocation)));
}
```

By using a different `PluginLoadContext` instance for each plugin, the plugins can have different or even conflicting dependencies without issue.

Simple plugin with no dependencies

Back in the root folder, do the following:

1. Run the following command to create a new class library project named `HelloPlugin`:

   ```
   dotnet new classlib -o HelloPlugin
   ```

2. Run the following command to add the project to the `AppWithPlugin` solution:

   ```
   dotnet sln add HelloPlugin/HelloPlugin.csproj
   ```

3. Replace the *HelloPlugin/Class1.cs* file with a file named *HelloCommand.cs* with the following contents:

```
using PluginBase;
using System;

namespace HelloPlugin
{
    public class HelloCommand : ICommand
    {
        public string Name { get => "hello"; }
        public string Description { get => "Displays hello message."; }

        public int Execute()
        {
            Console.WriteLine("Hello !!!");
            return 0;
        }
    }
}
```

Now, open the *HelloPlugin.csproj* file. It should look similar to the following:

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

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

</Project>
```

In between the `<Project>` tags, add the following elements:

```
<ItemGroup>
<ProjectReference Include="..\PluginBase\PluginBase.csproj">
    <Private>false</Private>
</ProjectReference>
</ItemGroup>
```

The `<Private>false</Private>` element is important. This tells MSBuild to not copy *PluginBase.dll* to the output directory for HelloPlugin. If the *PluginBase.dll* assembly is present in the output directory, `PluginLoadContext` will find the assembly there and load it when it loads the *HelloPlugin.dll* assembly. At this point, the `HelloPlugin.HelloCommand` type will implement the `ICommand` interface from the *PluginBase.dll* in the output directory of the `HelloPlugin` project, not the `ICommand` interface that is loaded into the default load context. Since the runtime sees these two types as different types from different assemblies, the `AppWithPlugin.Program.CreateCommands` method won't find the commands. As a result, the `<Private>false</Private>` metadata is required for the reference to the assembly containing the plugin interfaces.

Now that the `HelloPlugin` project is complete, we should update the `AppWithPlugin` project to know where the `HelloPlugin` plugin can be found. After the `// Paths to plugins to load` comment, add `@"HelloPlugin\bin\Debug\netcoreapp3.0\HelloPlugin.dll"` as an element of the `pluginPaths` array.

Plugin with library dependencies

Almost all plugins are more complex than a simple "Hello World", and many plugins have dependencies on other libraries. The `JsonPlugin` and `OldJson` plugin projects in the sample show two examples of plugins with NuGet package dependencies on `Newtonsoft.Json`. The project files themselves don't have any special information for the project references, and (after adding the plugin paths to the `pluginPaths` array) the plugins run perfectly, even if run in the same execution of the AppWithPlugin app. However, these projects don't copy the referenced assemblies to their output directory, so the assemblies need to be present on the user's machine for the plugins to work. There are two ways to work around this problem. The first option is to use the `dotnet publish` command to publish the class library. Alternatively, if you want to be able to use the output of `dotnet build` for your plugin, you can add the `<CopyLocalLockFileAssemblies>true</CopyLocalLockFileAssemblies>` property between the `<PropertyGroup>` tags in the plugin's project file. See the `XcopyablePlugin` plugin project for an example.

Other examples in the sample

The complete source code for this tutorial can be found in the dotnet/samples repository. The completed sample includes a few other examples of `AssemblyDependencyResolver` behavior. For example, the `AssemblyDependencyResolver` object can also resolve native libraries as well as localized satellite assemblies included in NuGet packages. The `UVPlugin` and `FrenchPlugin` in the samples repository demonstrate these scenarios.

Reference a plugin from a NuGet package

Let's say that there is an app A that has a plugin interface defined in the NuGet package named `A.PluginBase`. How do you reference the package correctly in your plugin project? For project references, using the

`<Private>false</Private>` metadata on the `ProjectReference` element in the project file prevented the dll from being copied to the output.

To correctly reference the `A.PluginBase` package, you want to change the `<PackageReference>` element in the project file to the following:

```
<PackageReference Include="A.PluginBase" Version="1.0.0">
    <ExcludeAssets>runtime</ExcludeAssets>
</PackageReference>
```

This prevents the `A.PluginBase` assemblies from being copied to the output directory of your plugin and ensures that your plugin will use A's version of `A.PluginBase`.

Plugin target framework recommendations

Because plugin dependency loading uses the *.deps.json* file, there is a gotcha related to the plugin's target framework. Specifically, your plugins should target a runtime, such as .NET Core 3.0, instead of a version of .NET Standard. The *.deps.json* file is generated based on which framework the project targets, and since many .NET Standard-compatible packages ship reference assemblies for building against .NET Standard and implementation assemblies for specific runtimes, the *.deps.json* may not correctly see implementation assemblies, or it may grab the .NET Standard version of an assembly instead of the .NET Core version you expect.

Plugin framework references

Currently, plugins can't introduce new frameworks into the process. For example, you can't load a plugin that uses the `Microsoft.AspNetCore.App` framework into an application that only uses the root `Microsoft.NETCore.App` framework. The host application must declare references to all frameworks needed by plugins.

Get started with ASP.NET Core

3/12/2019 • 2 minutes to read • Edit Online

For tutorials about developing ASP.NET Core web applications, see ASP.NET Core Tutorials.

Write a custom .NET Core host to control the .NET runtime from your native code

9/13/2019 • 21 minutes to read • Edit Online

Like all managed code, .NET Core applications are executed by a host. The host is responsible for starting the runtime (including components like the JIT and garbage collector) and invoking managed entry points.

Hosting the .NET Core runtime is an advanced scenario and, in most cases, .NET Core developers don't need to worry about hosting because .NET Core build processes provide a default host to run .NET Core applications. In some specialized circumstances, though, it can be useful to explicitly host the .NET Core runtime, either as a means of invoking managed code in a native process or in order to gain more control over how the runtime works.

This article gives an overview of the steps necessary to start the .NET Core runtime from native code and execute managed code in it.

Prerequisites

Because hosts are native applications, this tutorial will cover constructing a C++ application to host .NET Core. You will need a C++ development environment (such as that provided by Visual Studio).

You will also want a simple .NET Core application to test the host with, so you should install the .NET Core SDK and build a small .NET Core test app (such as a 'Hello World' app). The 'Hello World' app created by the new .NET Core console project template is sufficient.

Hosting APIs

There are three different APIs that can be used to host .NET Core. This document (and its associated samples) cover all options.

- The preferred method of hosting the .NET Core runtime in .NET Core 3.0 and above is with the `nethost` and `hostfxr` libraries' APIs. These entry points handle the complexity of finding and setting up the runtime for initialization and allow both launching a managed application and calling into a static managed method.
- The preferred method of hosting the .NET Core runtime prior to .NET Core 3.0 is with the CoreClrHost.h API. This API exposes functions for easily starting and stopping the runtime and invoking managed code (either by launching a managed exe or by calling static managed methods).
- .NET Core can also be hosted with the `ICLRRuntimeHost4` interface in mscoree.h. This API has been around longer than CoreClrHost.h, so you may have seen older hosts using it. It still works and allows a bit more control over the hosting process than CoreClrHost. For most scenarios, though, CoreClrHost.h is preferred now because of its simpler APIs.

Sample Hosts

Sample hosts demonstrating the steps outlined in the tutorials below are available in the dotnet/samples GitHub repository. Comments in the samples clearly associate the numbered steps from these tutorials with where they're performed in the sample. For download instructions, see Samples and Tutorials.

Keep in mind that the sample hosts are meant to be used for learning purposes, so they are light on error checking and are designed to emphasize readability over efficiency.

Create a host using NetHost.h and HostFxr.h

The following steps detail how to use the `nethost` and `hostfxr` libraries to start the .NET Core runtime in a native application and call into a managed static method. The sample uses the `nethost` header and library installed with the .NET SDK and copies of the `coreclr_delegates.h` and `hostfxr.h` files from the dotnet/core-setup repository.

Step 1 - Load HostFxr and get exported hosting functions

The `nethost` library provides the `get_hostfxr_path` function for locating the `hostfxr` library. The `hostfxr` library exposes functions for hosting the .NET Core runtime. The full list of functions can be found in `hostfxr.h` and the native hosting design document. The sample and this tutorial use the following:

- `hostfxr_initialize_for_runtime_config`: Initializes a host context and prepares for initialization of the .NET Core runtime using the specified runtime configuration.
- `hostfxr_get_runtime_delegate`: Gets a delegate for runtime functionality.
- `hostfxr_close`: Closes a host context.

The `hostfxr` library is found using `get_hostfxr_path`. It is then loaded and its exports are retrieved.

```
// Using the nethost library, discover the location of hostfxr and get exports
bool load_hostfxr()
{
    // Pre-allocate a large buffer for the path to hostfxr
    char_t buffer[MAX_PATH];
    size_t buffer_size = sizeof(buffer) / sizeof(char_t);
    int rc = get_hostfxr_path(buffer, &buffer_size, nullptr);
    if (rc != 0)
        return false;

    // Load hostfxr and get desired exports
    void *lib = load_library(buffer);
    init_fptr = (hostfxr_initialize_for_runtime_config_fn)get_export(lib,
"hostfxr_initialize_for_runtime_config");
    get_delegate_fptr = (hostfxr_get_runtime_delegate_fn)get_export(lib, "hostfxr_get_runtime_delegate");
    close_fptr = (hostfxr_close_fn)get_export(lib, "hostfxr_close");

    return (init_fptr && get_delegate_fptr && close_fptr);
}
```

Step 2 - Initialize and start the .NET Core runtime

The `hostfxr_initialize_for_runtime_config` and `hostfxr_get_runtime_delegate` functions initialize and start the .NET Core runtime using the runtime configuration for the managed component that will be loaded. The `hostfxr_get_runtime_delegate` function is used to get a runtime delegate that allows loading a managed assembly and getting a function pointer to a static method in that assembly.

```
// Load and initialize .NET Core and get desired function pointer for scenario
load_assembly_and_get_function_pointer_fn get_dotnet_load_assembly(const char_t *config_path)
{
    // Load .NET Core
    void *load_assembly_and_get_function_pointer = nullptr;
    hostfxr_handle cxt = nullptr;
    int rc = init_fptr(config_path, nullptr, &cxt);
    if (rc != 0 || cxt == nullptr)
    {
        std::cerr << "Init failed: " << std::hex << std::showbase << rc << std::endl;
        close_fptr(cxt);
        return nullptr;
    }

    // Get the load assembly function pointer
    rc = get_delegate_fptr(
        cxt,
        hdt_load_assembly_and_get_function_pointer,
        &load_assembly_and_get_function_pointer);
    if (rc != 0 || load_assembly_and_get_function_pointer == nullptr)
        std::cerr << "Get delegate failed: " << std::hex << std::showbase << rc << std::endl;

    close_fptr(cxt);
    return (load_assembly_and_get_function_pointer_fn)load_assembly_and_get_function_pointer;
}
```

Step 3 - Load managed assembly and get function pointer to a managed method

The runtime delegate is called to load the managed assembly and get a function pointer to a managed method. The delegate requires the assembly path, type name, and method name as inputs and returns a function pointer that can be used to invoke the managed method.

```
// Function pointer to managed delegate
component_entry_point_fn hello = nullptr;
int rc = load_assembly_and_get_function_pointer(
    dotnetlib_path.c_str(),
    dotnet_type,
    dotnet_type_method,
    nullptr /*delegate_type_name*/,
    nullptr,
    (void**)&hello);
```

By passing `nullptr` as the delegate type name when calling the runtime delegate, the sample uses a default signature for the managed method:

```
public delegate int ComponentEntryPoint(IntPtr args, int sizeBytes);
```

A different signature can be used by specifying the delegate type name when calling the runtime delegate.

Step 4 - Run managed code!

The native host can now call the managed method and pass it the desired parameters.

```
lib_args args
{
    STR("from host!"),
    i
};

hello(&args, sizeof(args));
```

Create a host using CoreClrHost.h

The following steps detail how to use the CoreClrHost.h API to start the .NET Core runtime in a native application and call into a managed static method. The code snippets in this document use some Windows-specific APIs, but the full sample host shows both Windows and Linux code paths.

The Unix CoreRun host shows a more complex, real-world example of hosting using coreclrhost.h.

Step 1 - Find and load CoreCLR

The .NET Core runtime APIs are in *coreclr.dll* (on Windows), in *libcoreclr.so* (on Linux), or in *libcoreclr.dylib* (on macOS). The first step to hosting .NET Core is to load the CoreCLR library. Some hosts probe different paths or use input parameters to find the library while others know to load it from a certain path (next to the host, for example, or from a machine-wide location).

Once found, the library is loaded with `LoadLibraryEx` (on Windows) or `dlopen` (on Linux/Mac).

```
HMODULE coreClr = LoadLibraryExA(coreClrPath.c_str(), NULL, 0);
```

Step 2 - Get .NET Core hosting functions

CoreClrHost has several important methods useful for hosting .NET Core:

- `coreclr_initialize`: Starts the .NET Core runtime and sets up the default (and only) AppDomain.
- `coreclr_execute_assembly`: Executes a managed assembly.
- `coreclr_create_delegate`: Creates a function pointer to a managed method.
- `coreclr_shutdown`: Shuts down the .NET Core runtime.
- `coreclr_shutdown_2`: Like `coreclr_shutdown`, but also retrieves the managed code's exit code.

After loading the CoreCLR library, the next step is to get references to these functions using `GetProcAddress` (on Windows) or `dlsym` (on Linux/Mac).

```
coreclr_initialize_ptr initializeCoreClr = (coreclr_initialize_ptr)GetProcAddress(coreClr,
"coreclr_initialize");
coreclr_create_delegate_ptr createManagedDelegate = (coreclr_create_delegate_ptr)GetProcAddress(coreClr,
"coreclr_create_delegate");
coreclr_shutdown_ptr shutdownCoreClr = (coreclr_shutdown_ptr)GetProcAddress(coreClr, "coreclr_shutdown");
```

Step 3 - Prepare runtime properties

Before starting the runtime, it is necessary to prepare some properties to specify behavior (especially concerning the assembly loader).

Common properties include:

- `TRUSTED_PLATFORM_ASSEMBLIES` This is a list of assembly paths (delimited by ';' on Windows and ':' on Linux) which the runtime will be able to resolve by default. Some hosts have hard-coded manifests listing assemblies they can load. Others will put any library in certain locations (next to *coreclr.dll*, for example) on this list.
- `APP_PATHS` This is a list of paths to probe in for an assembly if it can't be found in the trusted platform assemblies (TPA) list. Because the host has more control over which assemblies are loaded using the TPA list, it is a best practice for hosts to determine which assemblies they expect to load and list them explicitly. If probing at runtime is needed, however, this property can enable that scenario.
- `APP_NI_PATHS` This list is similar to APP_PATHS except that it's meant to be paths that will be probed for native images.
- `NATIVE_DLL_SEARCH_DIRECTORIES` This property is a list of paths the loader should probe when looking for native libraries called via p/invoke.

- `PLATFORM_RESOURCE_ROOTS` This list includes paths to probe in for resource satellite assemblies (in culture-specific sub-directories).

In this sample host, the TPA list is constructed by simply listing all libraries in the current directory:

```
void BuildTpaList(const char* directory, const char* extension, std::string& tpaList)
{
    // This will add all files with a .dll extension to the TPA list.
    // This will include unmanaged assemblies (coreclr.dll, for example) that don't
    // belong on the TPA list. In a real host, only managed assemblies that the host
    // expects to load should be included. Having extra unmanaged assemblies doesn't
    // cause anything to fail, though, so this function just enumerates all dll's in
    // order to keep this sample concise.
    std::string searchPath(directory);
    searchPath.append(FS_SEPARATOR);
    searchPath.append("*");
    searchPath.append(extension);

    WIN32_FIND_DATAA findData;
    HANDLE fileHandle = FindFirstFileA(searchPath.c_str(), &findData);

    if (fileHandle != INVALID_HANDLE_VALUE)
    {
        do
        {
            // Append the assembly to the list
            tpaList.append(directory);
            tpaList.append(FS_SEPARATOR);
            tpaList.append(findData.cFileName);
            tpaList.append(PATH_DELIMITER);

            // Note that the CLR does not guarantee which assembly will be loaded if an assembly
            // is in the TPA list multiple times (perhaps from different paths or perhaps with different
NI/NI.dll
            // extensions. Therefore, a real host should probably add items to the list in priority order and
only
            // add a file if it's not already present on the list.
            //
            // For this simple sample, though, and because we're only loading TPA assemblies from a single
path,
            // and have no native images, we can ignore that complication.
        }
        while (FindNextFileA(fileHandle, &findData));
        FindClose(fileHandle);
    }
}
```

Because the sample is simple, it only needs the `TRUSTED_PLATFORM_ASSEMBLIES` property:

```
// Define CoreCLR properties
// Other properties related to assembly loading are common here,
// but for this simple sample, TRUSTED_PLATFORM_ASSEMBLIES is all
// that is needed. Check hosting documentation for other common properties.
const char* propertyKeys[] = {
    "TRUSTED_PLATFORM_ASSEMBLIES"      // Trusted assemblies
};

const char* propertyValues[] = {
    tpaList.c_str()
};
```

Step 4 - Start the runtime

Unlike the mscoree.h hosting API (described below), CoreCLRHost.h APIs start the runtime and create the default

AppDomain all with a single call. The `coreclr_initialize` function takes a base path, name, and the properties described earlier and returns back a handle to the host via the `hostHandle` parameter.

```
void* hostHandle;
unsigned int domainId;

// This function both starts the .NET Core runtime and creates
// the default (and only) AppDomain
int hr = initializeCoreClr(
                runtimePath,        // App base path
                "SampleHost",       // AppDomain friendly name
                sizeof(propertyKeys) / sizeof(char*),   // Property count
                propertyKeys,       // Property names
                propertyValues,     // Property values
                &hostHandle,        // Host handle
                &domainId);         // AppDomain ID
```

Step 5 - Run managed code!

With the runtime started, the host can call managed code. This can be done in a couple of different ways. The sample code linked to this tutorial uses the `coreclr_create_delegate` function to create a delegate to a static managed method. This API takes the assembly name, namespace-qualified type name, and method name as inputs and returns a delegate that can be used to invoke the method.

```
doWork_ptr managedDelegate;

// The assembly name passed in the third parameter is a managed assembly name
// as described at https://docs.microsoft.com/dotnet/framework/app-domains/assembly-names
hr = createManagedDelegate(
        hostHandle,
        domainId,
        "ManagedLibrary, Version=1.0.0.0",
        "ManagedLibrary.ManagedWorker",
        "DoWork",
        (void**)&managedDelegate);
```

In this sample, the host can now call `managedDelegate` to run the `ManagedWorker.DoWork` method.

Alternatively, the `coreclr_execute_assembly` function can be used to launch a managed executable. This API takes an assembly path and array of arguments as input parameters. It loads the assembly at that path and invokes its main method.

```
int hr = executeAssembly(
        hostHandle,
        domainId,
        argumentCount,
        arguments,
        "HelloWorld.exe",
        (unsigned int*)&exitCode);
```

Step 6 - Shutdown and clean up

Finally, when the host is done running managed code, the .NET Core runtime is shut down with `coreclr_shutdown` or `coreclr_shutdown_2`.

```
hr = shutdownCoreClr(hostHandle, domainId);
```

CoreCLR does not support reinitialization or unloading. Do not call `coreclr_initialize` again or unload the CoreCLR library.

Create a host using Mscoree.h

As mentioned previously, CoreClrHost.h is now the preferred method of hosting the .NET Core runtime. The `ICLRRuntimeHost4` interface can still be used, though, if the CoreClrHost.h interfaces aren't sufficient (if non-standard startup flags are needed, for example, or if an AppDomainManager is needed on the default domain). These instructions will guide you through hosting .NET Core using mscoree.h.

The CoreRun host shows a more complex, real-world example of hosting using mscoree.h.

A note about mscoree.h

The `ICLRRuntimeHost4` .NET Core hosting interface is defined in MSCOREE.IDL. A header version of this file (mscoree.h), which your host will need to reference, is produced via MIDL when the .NET Core runtime is built. If you do not want to build the .NET Core runtime, mscoree.h is also available as a pre-built header in the dotnet/coreclr repository. Instructions on building the .NET Core runtime can be found in its GitHub repository.

Step 1 - Identify the managed entry point

After referencing necessary headers (mscoree.h and stdio.h, for example), one of the first things a .NET Core host must do is locate the managed entry point it will be using. In our sample host, this is done by just taking the first command line argument to our host as the path to a managed binary whose `main` method will be executed.

```
// The managed application to run should be the first command-line parameter.
// Subsequent command line parameters will be passed to the managed app later in this host.
wchar_t targetApp[MAX_PATH];
GetFullPathNameW(argv[1], MAX_PATH, targetApp, NULL);
```

Step 2 - Find and load CoreCLR

The .NET Core runtime APIs are in *CoreCLR.dll* (on Windows). To get our hosting interface (`ICLRRuntimeHost4`), it's necessary to find and load *CoreCLR.dll*. It is up to the host to define a convention for how it will locate *CoreCLR.dll*. Some hosts expect the file to be present in a well-known machine-wide location (such as *%programfiles%\dotnet\shared\Microsoft.NETCore.App\2.1.6*). Others expect that *CoreCLR.dll* will be loaded from a location next to either the host itself or the app to be hosted. Still others might consult an environment variable to find the library.

On Linux or Mac, the core runtime library is *libcoreclr.so* or *libcoreclr.dylib*, respectively.

Our sample host probes a few common locations for *CoreCLR.dll*. Once found, it must be loaded via `LoadLibrary` (or `dlopen` on Linux/Mac).

```
HMODULE ret = LoadLibraryExW(coreDllPath, NULL, 0);
```

Step 3 - Get an ICLRRuntimeHost4 Instance

The `ICLRRuntimeHost4` hosting interface is retrieved by calling `GetProcAddress` (or `dlsym` on Linux/Mac) on `GetCLRRuntimeHost`, and then invoking that function.

```
ICLRRuntimeHost4* runtimeHost;

FnGetCLRRuntimeHost pfnGetCLRRuntimeHost =
    (FnGetCLRRuntimeHost)::GetProcAddress(coreCLRModule, "GetCLRRuntimeHost");

if (!pfnGetCLRRuntimeHost)
{
    printf("ERROR - GetCLRRuntimeHost not found");
    return -1;
}

// Get the hosting interface
HRESULT hr = pfnGetCLRRuntimeHost(IID_ICLRRuntimeHost4, (IUnknown**)&runtimeHost);
```

Step 4 - Set startup flags and start the runtime

With an `ICLRRuntimeHost4` in-hand, we can now specify runtime-wide startup flags and start the runtime. Startup flags determine which garbage collector (GC) to use (concurrent or server), whether we will use a single AppDomain or multiple AppDomains, and what loader optimization policy to use (for domain-neutral loading of assemblies).

```
hr = runtimeHost->SetStartupFlags(
    // These startup flags control runtime-wide behaviors.
    // A complete list of STARTUP_FLAGS can be found in mscoree.h,
    // but some of the more common ones are listed below.
    static_cast<STARTUP_FLAGS>(
        // STARTUP_FLAGS::STARTUP_SERVER_GC |        // Use server GC
        // STARTUP_FLAGS::STARTUP_LOADER_OPTIMIZATION_MULTI_DOMAIN |  // Maximize domain-neutral loading
        // STARTUP_FLAGS::STARTUP_LOADER_OPTIMIZATION_MULTI_DOMAIN_HOST | // Domain-neutral loading for
strongly-named assemblies
        STARTUP_FLAGS::STARTUP_CONCURRENT_GC |      // Use concurrent GC
        STARTUP_FLAGS::STARTUP_SINGLE_APPDOMAIN |     // All code executes in the default AppDomain
                                                                    // (required to use the runtimeHost-
>ExecuteAssembly helper function)
        STARTUP_FLAGS::STARTUP_LOADER_OPTIMIZATION_SINGLE_DOMAIN // Prevents domain-neutral loading
    )
);
```

The runtime is started with a call to the `Start` function.

```
hr = runtimeHost->Start();
```

Step 5 - Preparing AppDomain settings

Once the runtime is started, we will want to set up an AppDomain. There are a number of options that must be specified when creating a .NET AppDomain, however, so it's necessary to prepare those first.

AppDomain flags specify AppDomain behaviors related to security and interop. Older Silverlight hosts used these settings to sandbox user code, but most modern .NET Core hosts run user code as full trust and enable interop.

```
int appDomainFlags =
    // APPDOMAIN_FORCE_TRIVIAL_WAIT_OPERATIONS |  // Do not pump messages during wait
    // APPDOMAIN_SECURITY_SANDBOXED |     // Causes assemblies not from the TPA list to be loaded as partially
trusted
    APPDOMAIN_ENABLE_PLATFORM_SPECIFIC_APPS |   // Enable platform-specific assemblies to run
    APPDOMAIN_ENABLE_PINVOKE_AND_CLASSIC_COMINTEROP | // Allow PInvoking from non-TPA assemblies
    APPDOMAIN_DISABLE_TRANSPARENCY_ENFORCEMENT;   // Entirely disables transparency checks
```

After deciding which AppDomain flags to use, AppDomain properties must be defined. The properties are key/value pairs of strings. Many of the properties relate to how the AppDomain will load assemblies.

Common AppDomain properties include:

- `TRUSTED_PLATFORM_ASSEMBLIES` This is a list of assembly paths (delimited by `;` on Windows and `:` on Linux/Mac) which the AppDomain should prioritize loading and give full trust to (even in partially-trusted domains). This list is meant to contain 'Framework' assemblies and other trusted modules, similar to the GAC in .NET Framework scenarios. Some hosts will put any library next to *coreclr.dll* on this list, others have hard-coded manifests listing trusted assemblies for their purposes.
- `APP_PATHS` This is a list of paths to probe in for an assembly if it can't be found in the trusted platform assemblies (TPA) list. Because the host has more control over which assemblies are loaded using the TPA list, it is a best practice for hosts to determine which assemblies they expect to load and list them explicitly. If probing at runtime is needed, however, this property can enable that scenario.
- `APP_NI_PATHS` This list is very similar to APP_PATHS except that it's meant to be paths that will be probed for native images.
- `NATIVE_DLL_SEARCH_DIRECTORIES` This property is a list of paths the loader should probe when looking for native DLLs called via p/invoke.
- `PLATFORM_RESOURCE_ROOTS` This list includes paths to probe in for resource satellite assemblies (in culture-specific sub-directories).

In our simple sample host, these properties are set up as follows:

```
// TRUSTED_PLATFORM_ASSEMBLIES
// "Trusted Platform Assemblies" are prioritized by the loader and always loaded with full trust.
// A common pattern is to include any assemblies next to CoreCLR.dll as platform assemblies.
// More sophisticated hosts may also include their own Framework extensions (such as AppDomain managers)
// in this list.
size_t tpaSize = 100 * MAX_PATH; // Starting size for our TPA (Trusted Platform Assemblies) list
wchar_t* trustedPlatformAssemblies = new wchar_t[tpaSize];
trustedPlatformAssemblies[0] = L'\0';

// Extensions to probe for when finding TPA list files
const wchar_t *tpaExtensions[] = {
    L"*.dll",
    L"*.exe",
    L"*.winmd"
};

// Probe next to CoreCLR.dll for any files matching the extensions from tpaExtensions and
// add them to the TPA list. In a real host, this would likely be extracted into a separate function
// and perhaps also run on other directories of interest.
for (int i = 0; i < _countof(tpaExtensions); i++)
{
    // Construct the file name search pattern
    wchar_t searchPath[MAX_PATH];
    wcscpy_s(searchPath, MAX_PATH, coreRoot);
    wcscat_s(searchPath, MAX_PATH, L"\\");
    wcscat_s(searchPath, MAX_PATH, tpaExtensions[i]);

    // Find files matching the search pattern
    WIN32_FIND_DATAW findData;
    HANDLE fileHandle = FindFirstFileW(searchPath, &findData);

    if (fileHandle != INVALID_HANDLE_VALUE)
    {
        do
        {
            // Construct the full path of the trusted assembly
            wchar_t pathToAdd[MAX_PATH];
            wcscpy_s(pathToAdd, MAX_PATH, coreRoot);
            wcscat_s(pathToAdd, MAX_PATH, L"\\");
            wcscat_s(pathToAdd, MAX_PATH, findData.cFileName);

            // Check to see if TPA list needs expanded
            if (wcsnlen(pathToAdd, MAX_PATH) + (3) + wcsnlen(trustedPlatformAssemblies, tpaSize) >= tpaSize)
            {
                // Expand, if needed
                tpaSize *= 2;
                wchar_t* newTPAList = new wchar_t[tpaSize];
                wcscpy_s(newTPAList, tpaSize, trustedPlatformAssemblies);
                trustedPlatformAssemblies = newTPAList;
            }

            // Add the assembly to the list and delimited with a semi-colon
            wcscat_s(trustedPlatformAssemblies, tpaSize, pathToAdd);
            wcscat_s(trustedPlatformAssemblies, tpaSize, L";");

            // Note that the CLR does not guarantee which assembly will be loaded if an assembly
            // is in the TPA list multiple times (perhaps from different paths or perhaps with different
NI/NI.dll
            // extensions. Therefore, a real host should probably add items to the list in priority order and
only
            // add a file if it's not already present on the list.
            //
            // For this simple sample, though, and because we're only loading TPA assemblies from a single
path,
            // we can ignore that complication.
        }
        while (FindNextFileW(fileHandle, &findData));
        FindClose(fileHandle);
    }
}


// APP_PATHS
// App paths are directories to probe in for assemblies which are not one of the well-known Framework
assemblies
// included in the TPA list.
//
// For this simple sample, we just include the directory the target application is in.
// More complex hosts may want to also check the current working directory or other
// locations known to contain application assets.
wchar_t appPaths[MAX_PATH * 50];

// Just use the targetApp provided by the user and remove the file name
wcscpy_s(appPaths, targetAppPath);


// APP_NI_PATHS
// App (NI) paths are the paths that will be probed for native images not found on the TPA list.
// It will typically be similar to the app paths.
// For this sample, we probe next to the app and in a hypothetical directory of the same name with 'NI'
suffixed to the end.
wchar_t appNiPaths[MAX_PATH * 50];
wcscpy_s(appNiPaths, targetAppPath);
wcscat_s(appNiPaths, MAX_PATH * 50, L";");
wcscat_s(appNiPaths, MAX_PATH * 50, targetAppPath);
wcscat_s(appNiPaths, MAX_PATH * 50, L"NI");


// NATIVE_DLL_SEARCH_DIRECTORIES
// Native dll search directories are paths that the runtime will probe for native DLLs called via PInvoke
wchar_t nativeDllSearchDirectories[MAX_PATH * 50];
wcscpy_s(nativeDllSearchDirectories, appPaths);
wcscat_s(nativeDllSearchDirectories, MAX_PATH * 50, L";");
wcscat_s(nativeDllSearchDirectories, MAX_PATH * 50, coreRoot);


// PLATFORM_RESOURCE_ROOTS
// Platform resource roots are paths to probe in for resource assemblies (in culture-specific sub-directories)
wchar_t platformResourceRoots[MAX_PATH * 50];
wcscpy_s(platformResourceRoots, appPaths);
```

Step 6 - Create the AppDomain

Once all AppDomain flags and properties are prepared, `ICLRRuntimeHost4::CreateAppDomainWithManager` can be used to set up the AppDomain. This function optionally takes a fully qualified assembly name and type name to use as the domain's AppDomain manager. An AppDomain manager can allow a host to control some aspects of AppDomain behavior and may provide entry points for launching managed code if the host doesn't intend to invoke user code directly.

```
DWORD domainId;

// Setup key/value pairs for AppDomain  properties
const wchar_t* propertyKeys[] = {
    L"TRUSTED_PLATFORM_ASSEMBLIES",
    L"APP_PATHS",
    L"APP_NI_PATHS",
    L"NATIVE_DLL_SEARCH_DIRECTORIES",
    L"PLATFORM_RESOURCE_ROOTS"
};

// Property values which were constructed in step 5
const wchar_t* propertyValues[] = {
    trustedPlatformAssemblies,
    appPaths,
    appNiPaths,
    nativeDllSearchDirectories,
    platformResourceRoots
};

// Create the AppDomain
hr = runtimeHost->CreateAppDomainWithManager(
    L"Sample Host AppDomain",  // Friendly AD name
    appDomainFlags,
    NULL,       // Optional AppDomain manager assembly name
    NULL,       // Optional AppDomain manager type (including namespace)
    sizeof(propertyKeys) / sizeof(wchar_t*),
    propertyKeys,
    propertyValues,
    &domainId);
```

Step 7 - Run managed code!

With an AppDomain up and running, the host can now start executing managed code. The easiest way to do this is to use `ICLRRuntimeHost4::ExecuteAssembly` to invoke a managed assembly's entry point method. Note that this function only works in single-domain scenarios.

```
DWORD exitCode = -1;
hr = runtimeHost->ExecuteAssembly(domainId, targetApp, argc - 1, (LPCWSTR*)(argc > 1 ? &argv[1] : NULL),
&exitCode);
```

Another option, if `ExecuteAssembly` doesn't meet your host's needs, is to use `CreateDelegate` to create a function pointer to a static managed method. This requires the host to know the signature of the method it is calling into (in order to create the function pointer type) but allows hosts the flexibility to invoke code other than an assembly's entry point. The assembly name provided in the second parameter is the full managed assembly name of the library to load.

```
void *pfnDelegate = NULL;
hr = runtimeHost->CreateDelegate(
    domainId,
    L"HW, Version=1.0.0.0, Culture=neutral", // Target managed assembly
    L"ConsoleApplication.Program",           // Target managed type
    L"Main",                                 // Target entry point (static method)
    (INT_PTR*)&pfnDelegate);

((MainMethodFp*)pfnDelegate)(NULL);
```

Step 8 - Clean up

Finally, the host should clean up after itself by unloading AppDomains, stopping the runtime, and releasing the `ICLRRuntimeHost4` reference.

```
runtimeHost->UnloadAppDomain(domainId, true /* Wait until unload complete */);
runtimeHost->Stop();
runtimeHost->Release();
```

CoreCLR does not support unloading. Do not unload the CoreCLR library.

Conclusion

Once your host is built, it can be tested by running it from the command line and passing any arguments the host expects (like the managed app to run for the mscoree example host). When specifying the .NET Core app for the host to run, be sure to use the .dll that is produced by `dotnet build`. Executables (.exe files) produced by `dotnet publish` for self-contained applications are actually the default .NET Core host (so that the app can be launched directly from the command line in mainline scenarios); user code is compiled into a dll of the same name.

If things don't work initially, double-check that*coreclr.dll* is available in the location expected by the host, that all necessary Framework libraries are in the TPA list, and that CoreCLR's bitness (32- or 64-bit) matches how the host was built.

Hosting the .NET Core runtime is an advanced scenario that many developers won't require, but for those who need to launch managed code from a native process, or who need more control over the .NET Core runtime's behavior, it can be very useful.

Exposing .NET Core components to COM

11/12/2019 • 2 minutes to read • Edit Online

In .NET Core, the process for exposing your .NET objects to COM has been significantly streamlined in comparison to .NET Framework. The following process will walk you through how to expose a class to COM. This tutorial shows you how to:

- Expose a class to COM from .NET Core.
- Generate a COM server as part of building your .NET Core library.
- Automatically generate a side-by-side server manifest for Registry-Free COM.

Prerequisites

- Install .NET Core 3.0 SDK or a newer version.

Create the library

The first step is to create the library.

1. Create a new folder, and in that folder run the following command:

   ```
   dotnet new classlib
   ```

2. Open `Class1.cs`.

3. Add `using System.Runtime.InteropServices;` to the top of the file.

4. Create an interface named `IServer`. For example:

   ```
   using System;
   using System.Runtime.InteropServices;

   [ComVisible(true)]
   [Guid(ContractGuids.ServerInterface)]
   [InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
   public interface IServer
   {
       /// <summary>
       /// Compute the value of the constant Pi.
       /// </summary>
       double ComputePi();
   }
   ```

5. Add the `[Guid("<IID>")]` attribute to the interface, with the interface GUID for the COM interface you're implementing. For example, `[Guid("fe103d6e-e71b-414c-80bf-982f18f6c1c7")]`. Note that this GUID needs to be unique since it is the only identifier of this interface for COM. In Visual Studio, you can generate a GUID by going to Tools > Create GUID to open the Create GUID tool.

6. Add the `[InterfaceType]` attribute to the interface and specify what base COM interfaces your interface should implement.

7. Create a class named `Server` that implements `IServer`.

8. Add the `[Guid("<CLSID>")]` attribute to the class, with the class identifier GUID for the COM class you're implementing. For example, `[Guid("9f35b6f5-2c05-4e7f-93aa-ee087f6e7ab6")]`. As with the interface GUID, this GUID must be unique since it is the only identifier of this interface to COM.
9. Add the `[ComVisible(true)]` attribute to both the interface and the class.

> **IMPORTANT**
> Unlike in .NET Framework, .NET Core requires you to specify the CLSID of any class you want to be activatable via COM.

Generate the COM host

1. Open the `.csproj` project file and add `<EnableComHosting>true</EnableComHosting>` inside a `<PropertyGroup></PropertyGroup>` tag.
2. Build the project.

The resulting output will have a `ProjectName.dll`, `ProjectName.deps.json`, `ProjectName.runtimeconfig.json` and `ProjectName.comhost.dll` file.

Register the COM host for COM

Open an elevated command prompt and run `regsvr32 ProjectName.comhost.dll`. That will register all of your exposed .NET objects with COM.

Enabling RegFree COM

1. Open the `.csproj` project file and add `<EnableRegFreeCom>true</EnableRegFreeCom>` inside a `<PropertyGroup></PropertyGroup>` tag.
2. Build the project.

The resulting output will now also have a `ProjectName.X.manifest` file. This file is the side-by-side manifest for use with Registry-Free COM.

Sample

There is a fully functional COM server sample in the dotnet/samples repository on GitHub.

Additional notes

Unlike in .NET Framework, there is no support in .NET Core for generating a COM Type Library (TLB) from a .NET Core assembly. The guidance is to either manually write an IDL file or a C/C++ header for the native declarations of the COM interfaces.

Additionally, loading both .NET Framework and .NET Core into the same process does have diagnostic limitations. The primary limitation is the debugging of managed components as it is not possible to debug both .NET Framework and .NET Core at the same time. In addition, the two runtime instances don't share managed assemblies. This means that it isn't possible to share actual .NET types across the two runtimes and instead all interactions must be restricted to the exposed COM interface contracts.

Packages, metapackages and frameworks

10/11/2019 • 7 minutes to read • Edit Online

.NET Core is a platform made of NuGet packages. Some product experiences benefit from fine-grained definition of packages while others from coarse-grained. To accommodate this duality, the product is distributed as a fine-grained set of packages and in coarser chunks with a package type informally called a metapackage.

Each of the .NET Core packages support being run on multiple .NET implementations, represented as frameworks. Some of those frameworks are traditional frameworks, like `net46`, representing the .NET Framework. Another set is new frameworks that can be thought of as "package-based frameworks", which establish a new model for defining frameworks. These package-based frameworks are entirely formed and defined as packages, forming a strong relationship between packages and frameworks.

Packages

.NET Core is split into a set of packages, which provide primitives, higher-level data types, app composition types and common utilities. Each of these packages represent a single assembly of the same name. For example, System.Runtime contains System.Runtime.dll.

There are advantages to defining packages in a fine-grained manner:

- Fine-grained packages can ship on their own schedule with relatively limited testing of other packages.
- Fine-grained packages can provide differing OS and CPU support.
- Fine-grained packages can have dependencies specific to only one library.
- Apps are smaller because unreferenced packages don't become part of the app distribution.

Some of these benefits are only used in certain circumstances. For example, NET Core packages will typically ship on the same schedule with the same platform support. In the case of servicing, fixes can be distributed and installed as small single package updates. Due to the narrow scope of change, the validation and time to make a fix available is limited to what is needed for a single library.

The following is a list of the key NuGet packages for .NET Core:

- System.Runtime - The most fundamental .NET Core package, including Object, String, Array, Action, and IList<T>.
- System.Collections - A set of (primarily) generic collections, including List<T> and Dictionary<TKey,TValue>.
- System.Net.Http - A set of types for HTTP network communication, including HttpClient and HttpResponseMessage.
- System.IO.FileSystem - A set of types for reading and writing to local or networked disk-based storage, including File and Directory.
- System.Linq - A set of types for querying objects, including `Enumerable` and ILookup<TKey,TElement>.
- System.Reflection - A set of types for loading, inspecting, and activating types, including Assembly, TypeInfo and MethodInfo.

Typically, rather than including each package, it's easier and more robust to include a metapackage. However, when you need a single package, you can include it as in the following example, which references the System.Runtime package.

```xml
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard1.6</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="System.Runtime" Version="4.3.0" />
  </ItemGroup>
</Project>
```

Metapackages

Metapackages are a NuGet package convention for describing a set of packages that are meaningful together. They represent this set of packages by making them dependencies. They can optionally establish a framework for this set of packages by specifying a framework.

Previous versions of the .NET Core tools (both project.json and csproj-based tools) by default specified both a framework and a metapackage. Currently, however, the metapackage is implicitly referenced by the target framework, so that each metapackage is tied to a target framework. For example, the `netstandard1.6` framework references the NetStandard.Library version 1.6.0 metapackage. Similarly, the `netcoreapp2.1` framework references the Microsoft.NETCore.App Version 2.1.0 metapackage. For more information, see Implicit metapackage package reference in the .NET Core SDK.

Targeting a framework and implicitly referencing a metapackage means that you in effect are adding a reference to each of its dependent packages as a single gesture. That makes all of the libraries in those packages available for IntelliSense (or similar experience) and for publishing your app.

There are advantages to using metapackages:

- Provides a convenient user experience to reference a large set of fine-grained packages.
- Defines a set of packages (including specific versions) that are tested and work well together.

The .NET Standard metapackage is:

- NETStandard.Library - Describes the libraries that are part of the ".NET Standard". Applies to all .NET implementations (for example, .NET Framework, .NET Core and Mono) that support .NET Standard. Establishes the 'netstandard' framework.

The key .NET Core metapackages are:

- Microsoft.NETCore.App - Describes the libraries that are part of the .NET Core distribution. Establishes the `.NETCoreApp` framework. Depends on the smaller `NETStandard.Library`.
- Microsoft.AspNetCore.App - Includes all the supported packages from ASP.NET Core and Entity Framework Core except those that contain third-party dependencies. See Microsoft.AspNetCore.App metapackage for ASP.NET Core for more information.
- Microsoft.AspNetCore.All - Includes all the supported packages from ASP.NET Core, Entity Framework Core, and internal and third-party dependencies used by ASP.NET Core and Entity Framework Core. See Microsoft.AspNetCore.All metapackage for ASP.NET Core 2.x for more information.
- Microsoft.NETCore.Portable.Compatibility - A set of compatibility facades that enable mscorlib-based Portable Class Libraries (PCLs) to run on .NET Core.

Frameworks

.NET Core packages each support a set of runtime frameworks. Frameworks describe an available API set (and potentially other characteristics) that you can rely on when you target a given framework. They are versioned as new APIs are added.

For example, System.IO.FileSystem supports the following frameworks:

- .NETFramework,Version=4.6
- .NETStandard,Version=1.3
- 6 Xamarin platforms (for example, xamarinios10)

It is useful to contrast the first two of these frameworks, since they are examples of the two different ways that frameworks are defined.

The `.NETFramework,Version=4.6` framework represents the available APIs in the .NET Framework 4.6. You can produce libraries compiled with the .NET Framework 4.6 reference assemblies and then distribute those libraries in NuGet packages in a net46 lib folder. It will be used for apps that target the .NET Framework 4.6 or that are compatible with it. This is how all frameworks have traditionally worked.

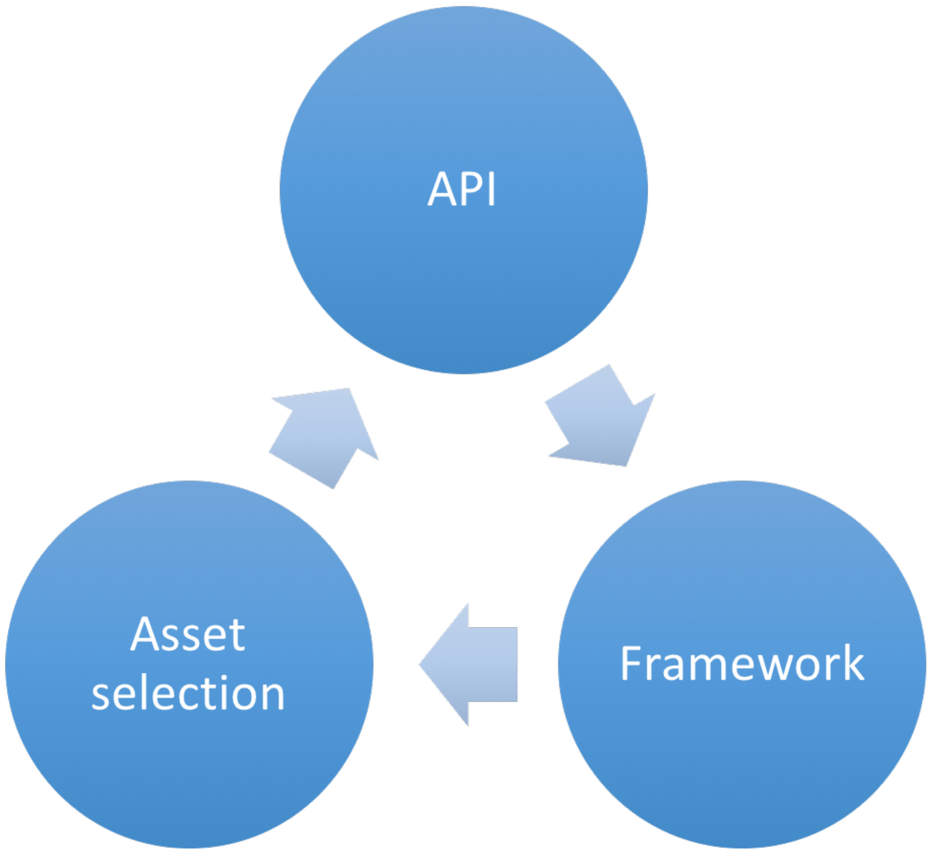
The `.NETStandard,Version=1.3` framework is a package-based framework. It relies on packages that target the framework to define and expose APIs in terms of the framework.

Package-based frameworks

There is a two-way relationship between frameworks and packages. The first part is defining the APIs available for a given framework, for example `netstandard1.3`. Packages that target `netstandard1.3` (or compatible frameworks, like `netstandard1.0`) define the APIs available for `netstandard1.3`. That may sound like a circular definition, but it isn't. By virtue of being "package-based", the API definition for the framework comes from packages. The framework itself doesn't define any APIs.

The second part of the relationship is asset selection. Packages can contain assets for multiple frameworks. Given a reference to a set of packages and/or metapackages, the framework is needed to determine which asset should be selected, for example `net46` or `netstandard1.3`. It is important to select the correct asset. For example, a `net46` asset is not likely to be compatible with .NET Framework 4.0 or .NET Core 1.0.

You can see this relationship in the following image. The *API* targets and defines the *framework*. The *framework* is used for *asset selection*. The *asset* gives you the API.



The two primary package-based frameworks used with .NET Core are:

- `netstandard`
- `netcoreapp`

.NET Standard

The .NET Standard (Target Framework Moniker: `netstandard`) framework represents the APIs defined by and built on top of the .NET Standard. Libraries that are intended to run on multiple runtimes should target this framework. They will be supported on any .NET Standard compliant runtime, such as .NET Core, .NET Framework and Mono/Xamarin. Each of these runtimes supports a set of .NET Standard versions, depending on which APIs they implement.

The `netstandard` framework implicitly references the `NETStandard.Library` metapackage. For example, the following MSBuild project file indicates that the project targets `netstandard1.6`, which references the `NETStandard.Library` version 1.6 metapackage.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard1.6</TargetFramework>
  </PropertyGroup>
</Project>
```

However, the framework and metapackage references in the project file do not need to match, and you can use the `<NetStandardImplicitPackageVersion>` element in your project file to specify a framework version that is lower than the metapackage version. For example, the following project file is valid.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard1.3</TargetFramework>
    <NetStandardImplicitPackageVersion>1.6.0</NetStandardImplicitPackageVersion>
  </PropertyGroup>
</Project>
```

It may seem strange to target `netstandard1.3` but use the 1.6.0 version of `NETStandard.Library`. It is a valid use-case, since the metapackage maintains support for older `netstandard` versions. It could be the case you've standardized on the 1.6.0 version of the metapackage and use it for all your libraries, which target a variety of `netstandard` versions. With this approach, you only need to restore `NETStandard.Library` 1.6.0 and not earlier versions.

The reverse would not be valid: targeting `netstandard1.6` with the 1.3.0 version of `NETStandard.Library`. You cannot target a higher framework with a lower metapackage, since the lower version metapackage will not expose any assets for that higher framework. The versioning scheme for metapackages asserts that metapackages match the highest version of the framework they describe. By virtue of the versioning scheme, the first version of `NETStandard.Library` is v1.6.0 given that it contains `netstandard1.6` assets. v1.3.0 is used in the example above, for symmetry with the example above, but does not actually exist.

.NET Core application

The .NET Core (Target Framework Moniker: `netcoreapp`) framework represents the packages and associated APIs that come with the .NET Core distribution and the console application model that it provides. .NET Core apps must use this framework, due to targeting the console application model, as should libraries that intended to run only on .NET Core. Using this framework restricts apps and libraries to running only on .NET Core.

The `Microsoft.NETCore.App` metapackage targets the `netcoreapp` framework. It provides access to ~60 libraries, ~40 provided by the `NETStandard.Library` package and ~20 more in addition. You can reference additional libraries that target `netcoreapp` or compatible frameworks, such as `netstandard`, to get access to additional APIs.

Most of the additional libraries provided by `Microsoft.NETCore.App` also target `netstandard` given that their dependencies are satisfied by other `netstandard` libraries. That means that `netstandard` libraries can also reference those packages as dependencies.

High-level overview of changes in the .NET Core tools

10/17/2019 • 4 minutes to read • Edit Online

This document describes the changes associated with moving from *project.json* to MSBuild and the *csproj* project system with information on the changes to the layering of the .NET Core tooling and the implementation of the CLI commands. These changes occurred with the release of .NET Core SDK 1.0 and Visual Studio 2017 on March 7, 2017 (see the announcement) but were initially implemented with the release of the .NET Core SDK Preview 3.

Moving away from project.json

The biggest change in the tooling for .NET Core is certainly the move away from project.json to csproj as the project system. The latest versions of the command-line tools don't support *project.json* files. That means that it cannot be used to build, run or publish project.json based applications and libraries. In order to use this version of the tools, you will need to migrate your existing projects or start new ones.

As part of this move, the custom build engine that was developed to build project.json projects was replaced with a mature and fully capable build engine called MSBuild. MSBuild is a well-known engine in the .NET community, since it has been a key technology since the platform's first release. Of course, because it needs to build .NET Core applications, MSBuild has been ported to .NET Core and can be used on any platform that .NET Core runs on. One of the main promises of .NET Core is that of a cross-platform development stack, and we have made sure that this move does not break that promise.

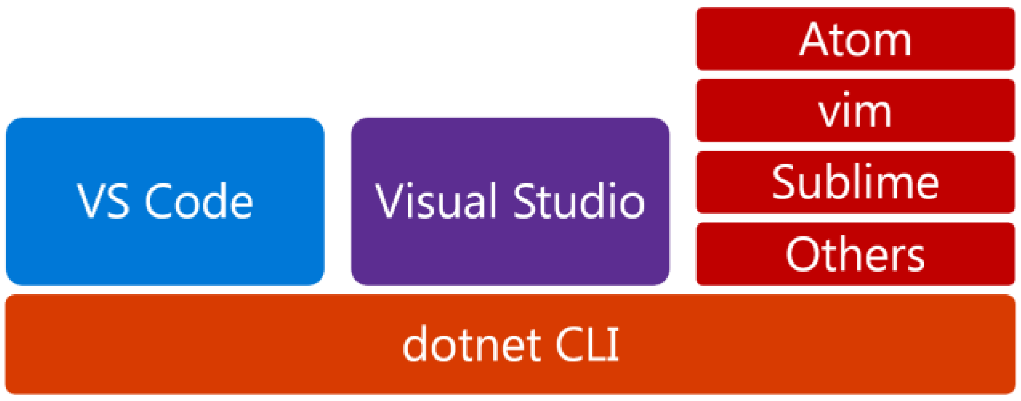
NOTE

If you are new to MSBuild and would like to learn more about it, you can start by reading the MSBuild Concepts article.

The tooling layers

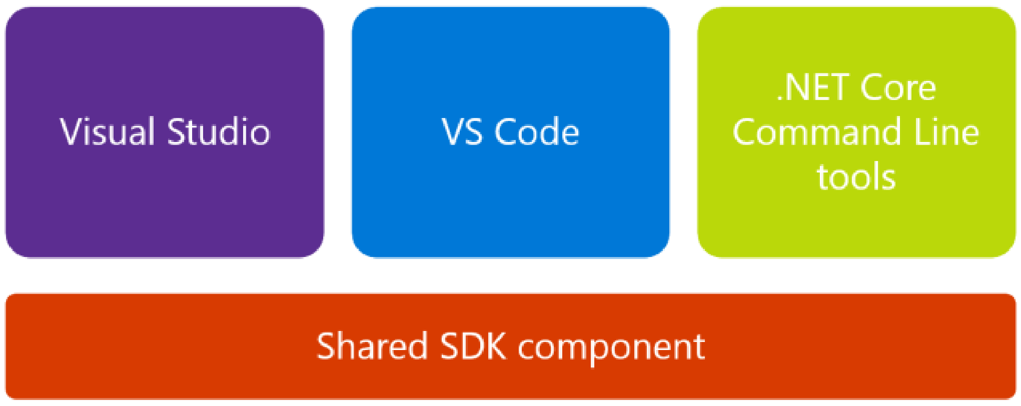
With the move away from the existing project system as well as with building engine switches, the question that naturally follows is do any of these changes change the overall "layering" of the whole .NET Core tooling ecosystem? Are there new bits and components?

Let's start with a quick refresher on Preview 2 layering as shown in the following picture:



The layering of the tools is quite simple. At the bottom we have the .NET Core Command Line tools as a foundation. All other, higher-level tools such as Visual Studio or Visual Studio Code, depend and rely on the CLI to build projects, restore dependencies and so on. This meant that, for example, if Visual Studio wanted to perform a restore operation, it would call into `dotnet restore` (see note) command in the CLI.

With the move to the new project system, the previous diagram changes:



The main difference is that the CLI is not the foundational layer anymore; this role is now filled by the "shared SDK component". This shared SDK component is a set of targets and associated tasks that are responsible for compiling your code, publishing it, packing NuGet packages etc. The SDK itself is open-source and is available on GitHub on the SDK repo.

NOTE

A "target" is a MSBuild term that indicates a named operation that MSBuild can invoke. It is usually coupled with one or more tasks that execute some logic that the target is supposed to do. MSBuild supports many ready-made targets such as `Copy` or `Execute`; it also allows users to write their own tasks using managed code and define targets to execute those tasks. For more information, see MSBuild tasks.

All the toolsets now consume the shared SDK component and its targets, CLI included. For example, the next version of Visual Studio will not call into `dotnet restore` (see note) command to restore dependencies for .NET Core projects, it will use the "Restore" target directly. Since these are MSBuild targets, you can also use raw MSBuild to execute them using the dotnet msbuild command.

CLI commands

The shared SDK component means that the majority of existing CLI commands have been re-implemented as MSBuild tasks and targets. What does this mean for the CLI commands and your usage of the toolset?

From an usage perspective, it doesn't change the way you use the CLI. The CLI still has the core commands that exist in Preview 2 release:

- `new`
- `restore`
- `run`
- `build`
- `publish`
- `test`
- `pack`

These commands still do what you expect them to do (new up a project, build it, publish it, pack it and so on). Majority of the options are not changed, and are still there, and you can consult either the commands' help screens (using `dotnet <command> --help`) or documentation on this site to get familiar with any changes.

From an execution perspective, the CLI commands will take their parameters and construct a call to "raw" MSBuild that will set the needed properties and run the desired target. To better illustrate this, consider the following command:

```
dotnet publish -o pub -c Release
```

This command is publishing an application into a `pub` folder using the "Release" configuration. Internally, this command gets translated into the following MSBuild invocation:

```
dotnet msbuild -t:Publish -p:OutputPath=pub -p:Configuration=Release
```

The notable exception to this rule are `new` and `run` commands, as they have not been implemented as MSBuild targets.

NOTE

Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

Managing dependencies with .NET Core SDK 1.0

4/16/2019 • 2 minutes to read • Edit Online

With the move of .NET Core projects from project.json to csproj and MSBuild, a significant investment also happened that resulted in unification of the project file and assets that allow tracking of dependencies. For .NET Core projects this is similar to what project.json did. There is no separate JSON or XML file that tracks NuGet dependencies. With this change, we've also introduced another type of *reference* into the csproj syntax called the `<PackageReference>`.

This document describes the new reference type. It also shows how to add a package dependency using this new reference type to your project.

The new <PackageReference> element

The `<PackageReference>` has the following basic structure:

```
<PackageReference Include="PACKAGE_ID" Version="PACKAGE_VERSION" />
```

If you are familiar with MSBuild, it will look familiar to the other reference types that already exist. The key is the `Include` statement which specifies the package id that you wish to add to the project. The `<Version>` child element specifies the version to get. The versions are specified as per NuGet version rules.

NOTE

If you are not familiar with the overall `csproj` syntax, see the MSBuild project reference documentation for more information.

Adding a dependency that is available only in a specific target is done using conditions like in the following example:

```
<PackageReference Include="PACKAGE_ID" Version="PACKAGE_VERSION" Condition="'$(TargetFramework)' ==
'netcoreapp2.1'" />
```

The above means that the dependency will only be valid if the build is happening for that given target. The `$(TargetFramework)` in the condition is a MSBuild property that is being set in the project. For most common .NET Core applications, you will not need to do this.

Adding a dependency to your project

Adding a dependency to your project is straightforward. Here is an example of how to add Json.NET version `9.0.1` to your project. Of course, it is applicable to any other NuGet dependency.

When you open your project file, you will see two or more `<ItemGroup>` nodes. You will notice that one of the nodes already has `<PackageReference>` elements in it. You can add your new dependency to this node, or create a new one; it is completely up to you as the result will be the same.

In this example we will use the default template that is dropped by `dotnet new console`. This is a simple console application. When we open up the project, we first find the `<ItemGroup>` with already existing `<PackageReference>` in it. We then add the following to it:

```
<PackageReference Include="Newtonsoft.Json" Version="9.0.1" />
```

After this, we save the project and run the `dotnet restore` command to install the dependency.

NOTE

Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

The full project looks like this:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Newtonsoft.Json" Version="9.0.1" />
  </ItemGroup>
</Project>
```

Removing a dependency from the project

Removing a dependency from the project file involves simply removing the `<PackageReference>` from the project file.

Additions to the csproj format for .NET Core

11/7/2019 • 15 minutes to read • Edit Online

This document outlines the changes that were added to the project files as part of the move from *project.json* to *csproj* and MSBuild. For more information about general project file syntax and reference, see the MSBuild project file documentation.

Implicit package references

Metapackages are implicitly referenced based on the target framework(s) specified in the `<TargetFramework>` or `<TargetFrameworks>` property of your project file. `<TargetFrameworks>` is ignored if `<TargetFramework>` is specified, independent of order. For more information, see Packages, metapackages and frameworks.

```
 <PropertyGroup>
   <TargetFramework>netcoreapp2.1</TargetFramework>
 </PropertyGroup>
```

```
<PropertyGroup>
  <TargetFrameworks>netcoreapp2.1;net462</TargetFrameworks>
</PropertyGroup>
```

Recommendations

Since `Microsoft.NETCore.App` or `NETStandard.Library` metapackages are implicitly referenced, the following are our recommended best practices:

- When targeting .NET Core or .NET Standard, never have an explicit reference to the `Microsoft.NETCore.App` or `NETStandard.Library` metapackages via a `<PackageReference>` item in your project file.
- If you need a specific version of the runtime when targeting .NET Core, you should use the `<RuntimeFrameworkVersion>` property in your project (for example, `1.0.4`) instead of referencing the metapackage.
  - This might happen if you are using self-contained deployments and you need a specific patch version of 1.0.0 LTS runtime, for example.
- If you need a specific version of the `NETStandard.Library` metapackage when targeting .NET Standard, you can use the `<NetStandardImplicitPackageVersion>` property and set the version you need.
- Don't explicitly add or update references to either the `Microsoft.NETCore.App` or `NETStandard.Library` metapackage in .NET Framework projects. If any version of `NETStandard.Library` is needed when using a .NET Standard-based NuGet package, NuGet automatically installs that version.

Implicit version for some package references

Most usages of `<PackageReference>` require setting the `Version` attribute to specify the NuGet package version to be used. When using .NET Core 2.1 or 2.2 and referencing Microsoft.AspNetCore.App or Microsoft.AspNetCore.All, however, the attribute is unnecessary. The .NET Core SDK can automatically select the version of these packages that should be used.

Recommendation

When referencing the `Microsoft.AspNetCore.App` or `Microsoft.AspNetCore.All` packages, do not specify their version. If a version is specified, the SDK may produce warning NETSDK1071. To fix this warning, remove the package version like in the following example:

```
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore.App" />
</ItemGroup>
```

> Known issue: the .NET Core 2.1 SDK only supported this syntax when the project also uses Microsoft.NET.Sdk.Web. This is resolved in the .NET Core 2.2 SDK.

These references to ASP.NET Core metapackages have a slightly different behavior from most normal NuGet packages. Framework-dependent deployments of applications that use these metapackages automatically take advantage of the ASP.NET Core shared framework. When you use the metapackages, **no** assets from the referenced ASP.NET Core NuGet packages are deployed with the application—the ASP.NET Core shared framework contains these assets. The assets in the shared framework are optimized for the target platform to improve application startup time. For more information about shared framework, see .NET Core distribution packaging.

If a version *is* specified, it's treated as the *minimum* version of ASP.NET Core shared framework for framework-dependent deployments and as an *exact* version for self-contained deployments. This can have the following consequences:

- If the version of ASP.NET Core installed on the server is less than the version specified on the PackageReference, the .NET Core process fails to launch. Updates to the metapackage are often available on NuGet.org before updates have been made available in hosting environments such as Azure. Updating the version on the PackageReference to ASP.NET Core could cause a deployed application to fail.
- If the application is deployed as a self-contained deployment, the application may not contain the latest security updates to .NET Core. When a version isn't specified, the SDK can automatically include the newest version of ASP.NET Core in the self-contained deployment.

Default compilation includes in .NET Core projects

With the move to the *csproj* format in the latest SDK versions, we've moved the default includes and excludes for compile items and embedded resources to the SDK properties files. This means that you no longer need to specify these items in your project file.

The main reason for doing this is to reduce the clutter in your project file. The defaults that are present in the SDK should cover most common use cases, so there is no need to repeat them in every project that you create. This leads to smaller project files that are much easier to understand as well as edit by hand, if needed.

The following table shows which element and which globs are both included and excluded in the SDK:

ELEMENT	INCLUDE GLOB	EXCLUDE GLOB	REMOVE GLOB
Compile	**/*.cs (or other language extensions)	**/*.user; **/*.*proj; **/*.sln; **/*.vssscc	N/A
EmbeddedResource	**/*.resx	**/*.user; **/*.*proj; **/*.sln; **/*.vssscc	N/A
None	**/*	**/*.user; **/*.*proj; **/*.sln; **/*.vssscc	**/*.cs; **/*.resx

> **NOTE**
>
> **Exclude glob** always excludes the `./bin` and `./obj` folders, which are represented by the `$(BaseOutputPath)` and `$(BaseIntermediateOutputPath)` MSBuild properties, respectively. As a whole, all excludes are represented by `$(DefaultItemExcludes)`.

If you have globs in your project and you try to build it using the newest SDK, you'll get the following error:

> Duplicate Compile items were included. The .NET SDK includes Compile items from your project directory by default. You can either remove these items from your project file, or set the 'EnableDefaultCompileItems' property to 'false' if you want to explicitly include them in your project file.

In order to get around this error, you can either remove the explicit `Compile` items that match the ones listed on the previous table, or you can set the `<EnableDefaultCompileItems>` property to `false`, like this:

```
<PropertyGroup>
    <EnableDefaultCompileItems>false</EnableDefaultCompileItems>
</PropertyGroup>
```

Setting this property to `false` will disable implicit inclusion, reverting to the behavior of previous SDKs where you had to specify the default globs in your project.

This change does not modify the main mechanics of other includes. However, if you wish to specify, for example, some files to get published with your app, you can still use the known mechanisms in *csproj* for that (for example, the `<Content>` element).

`<EnableDefaultCompileItems>` only disables `Compile` globs but doesn't affect other globs, like the implicit `None` glob, which also applies to *.cs items. Because of that, **Solution Explorer** will continue show *.cs items as part of the project, included as `None` items. In a similar way, you can set `<EnableDefaultNoneItems>` to false to disable the implicit `None` glob, like this:

```
<PropertyGroup>
    <EnableDefaultNoneItems>false</EnableDefaultNoneItems>
</PropertyGroup>
```

To disable **all implicit globs**, you can set the `<EnableDefaultItems>` property to `false` as in the following example:

```
<PropertyGroup>
    <EnableDefaultItems>false</EnableDefaultItems>
</PropertyGroup>
```

How to see the whole project as MSBuild sees it

While those csproj changes greatly simplify project files, you might want to see the fully expanded project as MSBuild sees it once the SDK and its targets are included. Preprocess the project with the `/pp` switch of the `dotnet msbuild` command, which shows which files are imported, their sources, and their contributions to the build without actually building the project:

```
dotnet msbuild -pp:fullproject.xml
```

If the project has multiple target frameworks, the results of the command should be focused on only one of them by specifying it as an MSBuild property:

```
dotnet msbuild -p:TargetFramework=netcoreapp2.0 -pp:fullproject.xml
```

Additions

Sdk attribute

The root `<Project>` element of the *.csproj* file has a new attribute called `Sdk`. `Sdk` specifies which SDK will be used by the project. The SDK, as the layering document describes, is a set of MSBuild tasks and targets that can build .NET Core code. The following SDKs are available for .NET Core:

1. The .NET Core SDK with the ID of `Microsoft.NET.Sdk`
2. The .NET Core web SDK with the ID of `Microsoft.NET.Sdk.Web`
3. The .NET Core Razor Class Library SDK with the ID of `Microsoft.NET.Sdk.Razor`
4. The .NET Core Worker Service with the ID of `Microsoft.NET.Sdk.Worker` (since .NET Core 3.0)
5. The .NET Core WinForms and WPF with the ID of `Microsoft.NET.Sdk.WindowsDesktop` (since .NET Core 3.0)

You need to have the `Sdk` attribute set to one of those IDs on the `<Project>` element in order to use the .NET Core tools and build your code.

PackageReference

A `<PackageReference>` item element specifies a NuGet dependency in the project. The `Include` attribute specifies the package ID.

```
<PackageReference Include="<package-id>" Version="" PrivateAssets="" IncludeAssets="" ExcludeAssets="" />
```

Version

The required `Version` attribute specifies the version of the package to restore. The attribute respects the rules of the NuGet versioning scheme. The default behavior is an exact version match. For example, specifying `Version="1.2.3"` is equivalent to NuGet notation `[1.2.3]` for the exact 1.2.3 version of the package.

IncludeAssets, ExcludeAssets and PrivateAssets

`IncludeAssets` attribute specifies which assets belonging to the package specified by `<PackageReference>` should be consumed. By default, all package assets are included.

`ExcludeAssets` attribute specifies which assets belonging to the package specified by `<PackageReference>` should not be consumed.

`PrivateAssets` attribute specifies which assets belonging to the package specified by `<PackageReference>` should be consumed but not flow to the next project. The `Analyzers`, `Build` and `ContentFiles` assets are private by default when this attribute is not present.

NOTE

`PrivateAssets` is equivalent to the *project.json/xproj* `SuppressParent` element.

These attributes can contain one or more of the following items, separated by the semicolon `;` character if more than one is listed:

- `Compile` – the contents of the *lib* folder are available to compile against.
- `Runtime` – the contents of the *runtime* folder are distributed.
- `ContentFiles` – the contents of the *contentfiles* folder are used.
- `Build` – the props/targets in the *build* folder are used.
- `Native` – the contents from native assets are copied to the *output* folder for runtime.
- `Analyzers` – the analyzers are used.

Alternatively, the attribute can contain:

- `None` – none of the assets are used.
- `All` – all assets are used.

DotNetCliToolReference

A `<DotNetCliToolReference>` item element specifies the CLI tool that the user wants to restore in the context of the project. It's a replacement for the `tools` node in *project.json*.

```
<DotNetCliToolReference Include="<package-id>" Version="" />
```

Version

`Version` specifies the version of the package to restore. The attribute respects the rules of the NuGet versioning scheme. The default behavior is an exact version match. For example, specifying `Version="1.2.3"` is equivalent to NuGet notation `[1.2.3]` for the exact 1.2.3 version of the package.

RuntimeIdentifiers

The `<RuntimeIdentifiers>` property element lets you specify a semicolon-delimited list of Runtime Identifiers (RIDs) for the project. RIDs enable publishing self-contained deployments.

```
<RuntimeIdentifiers>win10-x64;osx.10.11-x64;ubuntu.16.04-x64</RuntimeIdentifiers>
```

RuntimeIdentifier

The `<RuntimeIdentifier>` property element allows you to specify only one Runtime Identifier (RID) for the project. The RID enables publishing a self-contained deployment.

```
<RuntimeIdentifier>ubuntu.16.04-x64</RuntimeIdentifier>
```

Use `<RuntimeIdentifiers>` (plural) instead if you need to publish for multiple runtimes. `<RuntimeIdentifier>` can provide faster builds when only a single runtime is required.

PackageTargetFallback

The `<PackageTargetFallback>` property element allows you to specify a set of compatible targets to be used when restoring packages. It's designed to allow packages that use the dotnet TxM (Target x Moniker) to operate with packages that don't declare a dotnet TxM. If your project uses the dotnet TxM, then all the packages it depends on must also have a dotnet TxM, unless you add the `<PackageTargetFallback>` to your project in order to allow non-dotnet platforms to be compatible with dotnet.

The following example provides the fallbacks for all targets in your project:

```
<PackageTargetFallback>
    $(PackageTargetFallback);portable-net45+win8+wpa81+wp8
</PackageTargetFallback >
```

The following example specifies the fallbacks only for the `netcoreapp2.1` target:

```
<PackageTargetFallback Condition="'$(TargetFramework)'=='netcoreapp2.1'">
    $(PackageTargetFallback);portable-net45+win8+wpa81+wp8
</PackageTargetFallback >
```

Build events

The way that pre-build and post-build events are specified in the project file has changed. The properties PreBuildEvent and PostBuildEvent are not recommended in the SDK-style project format, because macros such as $(ProjectDir) are not resolved. For example, the following code is no longer supported:

```
<PropertyGroup>
    <PreBuildEvent>"$(ProjectDir)PreBuildEvent.bat" "$(ProjectDir)..\" "$(ProjectDir)" "$(TargetDir)" />
</PropertyGroup>
```

In SDK-style projects, use an MSBuild target named `PreBuild` or `PostBuild` and set the `BeforeTargets` property for

`PreBuild` or the `AfterTargets` property for `PostBuild`. For the preceding example, use the following code:

```
<Target Name="PreBuild" BeforeTargets="PreBuildEvent">
    <Exec Command="&quot;$(ProjectDir)PreBuildEvent.bat&quot; &quot;$(ProjectDir)..\&quot;
&quot;$(ProjectDir)&quot; &quot;$(TargetDir)&quot;" />
</Target>

<Target Name="PostBuild" AfterTargets="PostBuildEvent">
   <Exec Command="echo Output written to $(TargetDir)" />
</Target>
```

NOTE

You can use any name for the MSBuild targets, but the Visual Studio IDE recognizes `PreBuild` and `PostBuild` targets, so we recommend using those names so that you can edit the commands in the Visual Studio IDE.

NuGet metadata properties

With the move to MSBuild, we have moved the input metadata that is used when packing a NuGet package from *project.json* to *.csproj* files. The inputs are MSBuild properties so they have to go within a `<PropertyGroup>` group. The following is the list of properties that are used as inputs to the packing process when using the `dotnet pack` command or the `Pack` MSBuild target that is part of the SDK:

IsPackable

A Boolean value that specifies whether the project can be packed. The default value is `true`.

PackageVersion

Specifies the version that the resulting package will have. Accepts all forms of NuGet version string. Default is the value of `$(Version)`, that is, of the property `Version` in the project.

PackageId

Specifies the name for the resulting package. If not specified, the `pack` operation will default to using the `AssemblyName` or directory name as the name of the package.

Title

A human-friendly title of the package, typically used in UI displays as on nuget.org and the Package Manager in Visual Studio. If not specified, the package ID is used instead.

Authors

A semicolon-separated list of packages authors, matching the profile names on nuget.org. These are displayed in the NuGet Gallery on nuget.org and are used to cross-reference packages by the same authors.

PackageDescription

A long description of the package for UI display.

Description

A long description for the assembly. If `PackageDescription` is not specified then this property is also used as the description of the package.

Copyright

Copyright details for the package.

PackageRequireLicenseAcceptance

A Boolean value that specifies whether the client must prompt the consumer to accept the package license before installing the package. The default is `false`.

PackageLicenseExpression

An SPDX license identifier or expression. For example, `Apache-2.0`.

Here is the complete list of SPDX license identifiers. NuGet.org accepts only OSI or FSF approved licenses when using license type expression.

The exact syntax of the license expressions is described below in ABNF.

```
license-id            = <short form license identifier from https://spdx.org/spdx-specification-21-web-
version#h.luq9dgcle9mo>

license-exception-id  = <short form license exception identifier from https://spdx.org/spdx-specification-21-web-
version#h.ruv3yl8g6czd>

simple-expression = license-id / license-id”+”

compound-expression =  1*1(simple-expression /
                simple-expression "WITH" license-exception-id /
                compound-expression "AND" compound-expression /
                compound-expression "OR" compound-expression ) /
                "(" compound-expression ")" )

license-expression =  1*1(simple-expression / compound-expression / UNLICENSED)
```

NOTE

Only one of `PackageLicenseExpression`, `PackageLicenseFile` and `PackageLicenseUrl` can be specified at a time.

PackageLicenseFile

Path to a license file within the package if you are using a license that hasn’t been assigned an SPDX identifier, or it is a custom license (Otherwise `PackageLicenseExpression` is preferred)

Replaces `PackageLicenseUrl`, can't be combined with `PackageLicenseExpression` and requires Visual Studio 15.9.4, .NET SDK 2.1.502 or 2.2.101, or newer.

You will need to ensure the license file is packed by adding it explicitly to the project, example usage:

```
<PropertyGroup>
  <PackageLicenseFile>LICENSE.txt</PackageLicenseFile>
</PropertyGroup>
<ItemGroup>
  <None Include="licenses\LICENSE.txt" Pack="true" PackagePath="$(PackageLicenseFile)"/>
</ItemGroup>
```

PackageLicenseUrl

An URL to the license that is applicable to the package. (*deprecated since Visual Studio 15.9.4, .NET SDK 2.1.502 and 2.2.101*)

PackageIconUrl

A URL for a 64x64 image with transparent background to use as the icon for the package in UI display.

PackageReleaseNotes

Release notes for the package.

PackageTags

A semicolon-delimited list of tags that designates the package.

PackageOutputPath

Determines the output path in which the packed package will be dropped. Default is `$(OutputPath)`.

IncludeSymbols

This Boolean value indicates whether the package should create an additional symbols package when the project is packed. The symbols package's format is controlled by the `SymbolPackageFormat` property.

SymbolPackageFormat

Specifies the format of the symbols package. If "symbols.nupkg", a legacy symbols package will be created with a *.symbols.nupkg* extension containing PDBs, DLLs, and other output files. If "snupkg", a snupkg symbol package will be created containing the portable PDBs. Default is "symbols.nupkg".

IncludeSource

This Boolean value indicates whether the pack process should create a source package. The source package contains the library's source code as well as PDB files. Source files are put under the `src/ProjectName` directory in the resulting package file.

IsTool

Specifies whether all output files are copied to the *tools* folder instead of the *lib* folder. Note that this is different from a `DotNetCliTool` which is specified by setting the `PackageType` in the *.csproj* file.

RepositoryUrl

Specifies the URL for the repository where the source code for the package resides and/or from which it's being built.

RepositoryType

Specifies the type of the repository. Default is "git".

RepositoryBranch

Specifies the name of the source branch in the repository. When the project is packaged in a NuGet package, it's added to the package metadata.

RepositoryCommit

Optional repository commit or changeset to indicate which source the package was built against. `RepositoryUrl` must also be specified for this property to be included. When the project is packaged in a NuGet package, this commit or changeset is added to the package metadata.

NoPackageAnalysis

Specifies that pack should not run package analysis after building the package.

MinClientVersion

Specifies the minimum version of the NuGet client that can install this package, enforced by nuget.exe and the Visual Studio Package Manager.

IncludeBuildOutput

This Boolean values specifies whether the build output assemblies should be packed into the *.nupkg* file or not.

IncludeContentInPack

This Boolean value specifies whether any items that have a type of `Content` will be included in the resulting package automatically. The default is `true`.

BuildOutputTargetFolder

Specifies the folder where to place the output assemblies. The output assemblies (and other output files) are copied into their respective framework folders.

ContentTargetFolders

This property specifies the default location of where all the content files should go if `PackagePath` is not specified for them. The default value is "content;contentFiles".

NuspecFile

Relative or absolute path to the *.nuspec* file being used for packing.

> **NOTE**
>
> If the *.nuspec* file is specified, it's used **exclusively** for packaging information and any information in the projects is not used.

NuspecBasePath

Base path for the *.nuspec* file.

NuspecProperties

Semicolon separated list of key=value pairs.

AssemblyInfo properties

Assembly attributes that were typically present in an *AssemblyInfo* file are now automatically generated from properties.

Properties per attribute

Each attribute has a property that control its content and another to disable its generation as shown in the following table:

ATTRIBUTE	PROPERTY	PROPERTY TO DISABLE
AssemblyCompanyAttribute	`Company`	`GenerateAssemblyCompanyAttribute`
AssemblyConfigurationAttribute	`Configuration`	`GenerateAssemblyConfigurationAttribute`
AssemblyCopyrightAttribute	`Copyright`	`GenerateAssemblyCopyrightAttribute`
AssemblyDescriptionAttribute	`Description`	`GenerateAssemblyDescriptionAttribute`
AssemblyFileVersionAttribute	`FileVersion`	`GenerateAssemblyFileVersionAttribute`
AssemblyInformationalVersionAttribute	`InformationalVersion`	`GenerateAssemblyInformationalVersionAttribu`
AssemblyProductAttribute	`Product`	`GenerateAssemblyProductAttribute`
AssemblyTitleAttribute	`AssemblyTitle`	`GenerateAssemblyTitleAttribute`
AssemblyVersionAttribute	`AssemblyVersion`	`GenerateAssemblyVersionAttribute`
NeutralResourcesLanguageAttribute	`NeutralLanguage`	`GenerateNeutralResourcesLanguageAttribute`

Notes:

- `AssemblyVersion` and `FileVersion` default is to take the value of `$(Version)` without suffix. For example, if `$(Version)` is `1.2.3-beta.4`, then the value would be `1.2.3`.
- `InformationalVersion` defaults to the value of `$(Version)`.
- `InformationalVersion` has `$(SourceRevisionId)` appended if the property is present. It can be disabled using `IncludeSourceRevisionInInformationalVersion`.
- `Copyright` and `Description` properties are also used for NuGet metadata.
- `Configuration` is shared with all the build process and set via the `--configuration` parameter of `dotnet` commands.

GenerateAssemblyInfo

A Boolean that enable or disable all the AssemblyInfo generation. The default value is `true`.

GeneratedAssemblyInfoFile

The path of the generated assembly info file. Default to a file in the `$(IntermediateOutputPath)` (obj) directory.

Migrate from .NET Core 2.0 to 2.1

8/28/2019 • 2 minutes to read • Edit Online

This article shows you the basic steps for migrating your .NET Core 2.0 app to 2.1. If you're looking to migrate your ASP.NET Core app to 2.1, see Migrate from ASP.NET Core 2.0 to 2.1.

For an overview of the new features in .NET Core 2.1, see What's new in .NET Core 2.1.

Update the project file to use 2.1 versions

- Open the project file (the *.csproj, *.vbproj, or *.fsproj file).
- Change the target framework value from `netcoreapp2.0` to `netcoreapp2.1`. The target framework is defined by the `<TargetFramework>` or `<TargetFrameworks>` element.

  For example, change `<TargetFramework>netcoreapp2.0</TargetFramework>` to `<TargetFramework>netcoreapp2.1</TargetFramework>`.
- Remove `<DotNetCliToolReference>` references for tools that are bundled in the .NET Core 2.1 SDK (v 2.1.300 or later). These references include:
  - dotnet-watch (Microsoft.DotNet.Watcher.Tools)
  - dotnet-user-secrets (Microsoft.Extensions.SecretManager.Tools)
  - dotnet-sql-cache (Microsoft.Extensions.Caching.SqlConfig.Tools)
  - dotnet-ef (Microsoft.EntityFrameworkCore.Tools.DotNet)

  In previous .NET Core SDK versions, the reference to one of these tools in your project file looks similar to the following example:

  ```
  <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />
  ```

  Since this entry isn't used by the .NET Core SDK any longer, you'll see a warning similar to the following if you still have references to one of these bundled tools in your project:

  ```
  The tool 'Microsoft.EntityFrameworkCore.Tools.DotNet' is now included in the .NET Core SDK. Here is
  information on resolving this warning.
  ```

  Removing the `<DotNetCliToolReference>` references for those tools from your project file fixes this issue.

See also

- Migrate from ASP.NET Core 2.0 to 2.1
- What's new in .NET Core 2.1

Migrating .NET Core projects from project.json

10/22/2019 • 4 minutes to read • Edit Online

This document covers the following three migration scenarios for .NET Core projects:

1. Migration from a valid latest schema of *project.json* to *csproj*
2. Migration from DNX to csproj
3. Migration from RC3 and previous .NET Core csproj projects to the final format

This document is only applicable to older .NET Core projects that use project.json. It does not apply to migrating from .NET Framework to .NET Core.

Migration from project.json to csproj

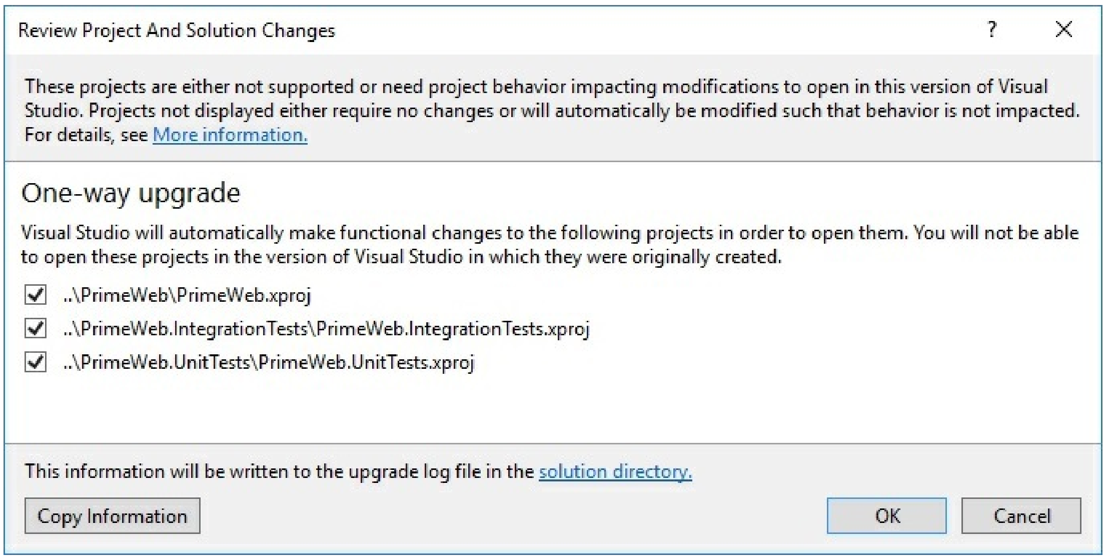
Migration from *project.json* to *.csproj* can be done using one of the following methods:

- Visual Studio
- dotnet migrate command-line tool

Both methods use the same underlying engine to migrate the projects, so the results will be the same for both. In most cases, using one of these two ways to migrate the *project.json* to *csproj* is the only thing that is needed, and no further manual editing of the project file is necessary. The resulting *.csproj* file will be named the same as the containing directory name.

Visual Studio

When you open an *.xproj* file or a solution file that references *.xproj* files in Visual Studio 2017 or Visual Studio 2019 version 16.2 and earlier, the **One-way upgrade** dialog appears. The dialog displays the projects to be migrated. If you open a solution file, all the projects specified in the solution file are listed. Review the list of projects to be migrated and select **OK**.


Review Project And Solution Changes

These projects are either not supported or need project behavior impacting modifications to open in this version of Visual Studio. Projects not displayed either require no changes or will automatically be modified such that behavior is not impacted. For details, see More information.

One-way upgrade

Visual Studio will automatically make functional changes to the following projects in order to open them. You will not be able to open these projects in the version of Visual Studio in which they were originally created.

- [✓] ..\PrimeWeb\PrimeWeb.xproj
- [✓] ..\PrimeWeb.IntegrationTests\PrimeWeb.IntegrationTests.xproj
- [✓] ..\PrimeWeb.UnitTests\PrimeWeb.UnitTests.xproj

This information will be written to the upgrade log file in the solution directory.

Visual Studio migrates the selected projects automatically. When migrating a solution, if you don't choose all projects, the same dialog appears asking you to upgrade the remaining projects from that solution. After the project is migrated, you can see and modify its contents by right-clicking the project in the **Solution Explorer** window and selecting **Edit <project name>.csproj**.

Files that were migrated (*project.json*, *global.json*, *.xproj*, and solution file) are moved to a *Backup* folder. The migrated solution file is upgraded to Visual Studio 2017 or Visual Studio 2019 and you won't be able to open that solution file in Visual Studio 2015 or earlier versions. A file named *UpgradeLog.htm* that contains a migration report is also saved and opened automatically.

IMPORTANT

In Visual Studio 2019 version 16.3 and later, you cannot load or migrate an *.xproj* file. Additionally, Visual Studio 2015 doesn't provide the ability to migrate an *.xproj* file. If you're using one of these Visual Studio versions, either install a suitable version of Visual Studio, or use the command line migration tool that's described next.

dotnet migrate

In the command-line scenario, you can use the `dotnet migrate` command. It migrates a project, a solution, or a set of folders in that order, depending on which ones were found. When you migrate a project, the project and all its dependencies are migrated.

Files that were migrated (*project.json*, *global.json*, and *.xproj*) are moved to a *backup* folder.

NOTE

If you are using Visual Studio Code, the `dotnet migrate` command does not modify Visual Studio Code-specific files such as *tasks.json*. These files need to be changed manually. This is also true if you are using an editor or Integrated Development Environment (IDE) other than Visual Studio.

See A mapping between project.json and csproj properties for a comparison of *project.json* and *.csproj* formats.

If you get an error:

> No executable found matching command dotnet-migrate

Run `dotnet --version` to see which version you are using. `dotnet migrate` was introduced in .NET Core SDK 1.0.0 and removed in version 3.0.100. You'll get this error if you have a *global.json* file in the current or parent directory, and the `sdk` version it specifies is outside this range.

Migration from DNX to csproj

If you are still using DNX for .NET Core development, your migration process should be done in two stages:

1. Use the existing DNX migration guidance to migrate from DNX to project-json enabled CLI.
2. Follow the steps from the previous section to migrate from *project.json* to *.csproj*.

NOTE

DNX has become officially deprecated during the Preview 1 release of the .NET Core CLI.

Migration from earlier .NET Core csproj formats to RTM csproj

The .NET Core csproj format has been changing and evolving with each new pre-release version of the tooling. There is no tool that will migrate your project file from earlier versions of csproj to the latest, so you need to manually edit the project file. The actual steps depend on the version of the project file you are migrating. The following is some guidance to consider based on the changes that happened between versions:

- Remove the tools version property from the `<Project>` element, if it exists.
- Remove the XML namespace (`xmlns`) from the `<Project>` element.

- If it doesn't exist, add the `Sdk` attribute to the `<Project>` element and set it to `Microsoft.NET.Sdk` or `Microsoft.NET.Sdk.Web`. This attribute specifies that the project uses the SDK to be used. `Microsoft.NET.Sdk.Web` is used for web apps.
- Remove the `<Import Project="$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\Microsoft.Common.props" />` and `<Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />` statements from the top and bottom of the project. These import statements are implied by the SDK, so there is no need for them to be in the project.
- If you have `Microsoft.NETCore.App` or `NETStandard.Library` `<PackageReference>` items in your project, you should remove them. These package references are implied by the SDK.
- Remove the `Microsoft.NET.Sdk` `<PackageReference>` element, if it exists. The SDK reference comes through the `Sdk` attribute on the `<Project>` element.
- Remove the globs that are implied by the SDK. Leaving these globs in your project will cause an error on build because compile items will be duplicated.

After these steps your project should be fully compatible with the RTM .NET Core csproj format.

For examples of before and after the migration from old csproj format to the new one, see the Updating Visual Studio 2017 RC – .NET Core Tooling improvements article on the .NET blog.

See also

- Port, Migrate, and Upgrade Visual Studio Projects

A mapping between project.json and csproj properties

2/23/2019 • 6 minutes to read • Edit Online

By Nate McMaster

During the development of the .NET Core tooling, an important design change was made to no longer support *project.json* files and instead move the .NET Core projects to the MSBuild/csproj format.

This article shows how the settings in *project.json* are represented in the MSBuild/csproj format so you can learn how to use the new format and understand the changes made by the migration tools when you're upgrading your project to the latest version of the tooling.

The csproj format

The new format, *.csproj, is an XML-based format. The following example shows the root node of a .NET Core project using the `Microsoft.NET.Sdk`. For web projects, the SDK used is `Microsoft.NET.Sdk.Web`.

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

Common top-level properties

name

```
{
  "name": "MyProjectName"
}
```

No longer supported. In csproj, this is determined by the project filename, which usually matches the directory name. For example, `MyProjectName.csproj`.

By default, the project filename also specifies the value of the `<AssemblyName>` and `<PackageId>` properties.

```
<PropertyGroup>
  <AssemblyName>MyProjectName</AssemblyName>
  <PackageId>MyProjectName</PackageId>
</PropertyGroup>
```

The `<AssemblyName>` will have a different value than `<PackageId>` if `buildOptions\outputName` property was defined in project.json. For more information, see Other common build options.

version

```
{
  "version": "1.0.0-alpha-*"
}
```

Use the `VersionPrefix` and `VersionSuffix` properties:

```xml
<PropertyGroup>
  <VersionPrefix>1.0.0</VersionPrefix>
  <VersionSuffix>alpha</VersionSuffix>
</PropertyGroup>
```

You can also use the `Version` property, but this may override version settings during packaging:

```xml
<PropertyGroup>
  <Version>1.0.0-alpha</Version>
</PropertyGroup>
```

Other common root-level options

```json
{
  "authors": [ "Anne", "Bob" ],
  "company": "Contoso",
  "language": "en-US",
  "title": "My library",
  "description": "This is my library.\r\nAnd it's really great!",
  "copyright": "Nugetizer 3000",
  "userSecretsId": "xyz123"
}
```

```xml
<PropertyGroup>
  <Authors>Anne;Bob</Authors>
  <Company>Contoso</Company>
  <NeutralLanguage>en-US</NeutralLanguage>
  <AssemblyTitle>My library</AssemblyTitle>
  <Description>This is my library.
And it's really great!</Description>
  <Copyright>Nugetizer 3000</Copyright>
  <UserSecretsId>xyz123</UserSecretsId>
</PropertyGroup>
```

frameworks

One target framework

```json
{
  "frameworks": {
    "netcoreapp1.0": {}
  }
}
```

```xml
<PropertyGroup>
  <TargetFramework>netcoreapp1.0</TargetFramework>
</PropertyGroup>
```

Multiple target frameworks

```
{
  "frameworks": {
    "netcoreapp1.0": {},
    "net451": {}
  }
}
```

Use the `TargetFrameworks` property to define your list of target frameworks. Use semi-colon to separate multiple framework values.

```
<PropertyGroup>
  <TargetFrameworks>netcoreapp1.0;net451</TargetFrameworks>
</PropertyGroup>
```

dependencies

IMPORTANT

If the dependency is a **project** and not a package, the format is different. For more information, see the dependency type section.

NETStandard.Library metapackage

```
{
  "dependencies": {
    "NETStandard.Library": "1.6.0"
  }
}
```

```
<PropertyGroup>
  <NetStandardImplicitPackageVersion>1.6.0</NetStandardImplicitPackageVersion>
</PropertyGroup>
```

Microsoft.NETCore.App metapackage

```
{
  "dependencies": {
    "Microsoft.NETCore.App": "1.0.0"
  }
}
```

```
<PropertyGroup>
  <RuntimeFrameworkVersion>1.0.3</RuntimeFrameworkVersion>
</PropertyGroup>
```

Note that the `<RuntimeFrameworkVersion>` value in the migrated project is determined by the version of the SDK you have installed.

Top-level dependencies

```json
{
  "dependencies": {
    "Microsoft.AspNetCore": "1.1.0"
  }
}
```

```xml
<ItemGroup>
  <PackageReference Include="Microsoft.AspNetCore" Version="1.1.0" />
</ItemGroup>
```

Per-framework dependencies

```json
{
  "framework": {
    "net451": {
      "dependencies": {
        "System.Collections.Immutable": "1.3.1"
      }
    },
    "netstandard1.5": {
      "dependencies": {
        "Newtonsoft.Json": "9.0.1"
      }
    }
  }
}
```

```xml
<ItemGroup Condition="'$(TargetFramework)'=='net451'">
  <PackageReference Include="System.Collections.Immutable" Version="1.3.1" />
</ItemGroup>

<ItemGroup Condition="'$(TargetFramework)'=='netstandard1.5'">
  <PackageReference Include="Newtonsoft.Json" Version="9.0.1" />
</ItemGroup>
```

imports

```json
{
  "dependencies": {
    "YamlDotNet": "4.0.1-pre309"
  },
  "frameworks": {
    "netcoreapp1.0": {
      "imports": [
        "dnxcore50",
        "dotnet"
      ]
    }
  }
}
```

```xml
<PropertyGroup>
  <PackageTargetFallback>dnxcore50;dotnet</PackageTargetFallback>
</PropertyGroup>
<ItemGroup>
  <PackageReference Include="YamlDotNet" Version="4.0.1-pre309" />
</ItemGroup>
```

dependency type

type: project

```
{
  "dependencies": {
    "MyOtherProject": "1.0.0-*",
    "AnotherProject": {
      "type": "project"
    }
  }
}
```

```
<ItemGroup>
  <ProjectReference Include="..\MyOtherProject\MyOtherProject.csproj" />
  <ProjectReference Include="..\AnotherProject\AnotherProject.csproj" />
</ItemGroup>
```

NOTE

This will break the way that `dotnet pack --version-suffix $suffix` determines the dependency version of a project reference.

type: build

```
{
  "dependencies": {
    "Microsoft.EntityFrameworkCore.Design": {
      "version": "1.1.0",
      "type": "build"
    }
  }
}
```

```
<ItemGroup>
  <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="1.1.0" PrivateAssets="All" />
</ItemGroup>
```

type: platform

```
{
  "dependencies": {
    "Microsoft.NETCore.App": {
      "version": "1.1.0",
      "type": "platform"
    }
  }
}
```

There is no equivalent in csproj.

runtimes

```
{
  "runtimes": {
    "win7-x64": {},
    "osx.10.11-x64": {},
    "ubuntu.16.04-x64": {}
  }
}
```

```
<PropertyGroup>
  <RuntimeIdentifiers>win7-x64;osx.10.11-x64;ubuntu.16.04-x64</RuntimeIdentifiers>
</PropertyGroup>
```

Standalone apps (self-contained deployment)

In project.json, defining a `runtimes` section means the app was standalone during build and publish. In MSBuild, all projects are *portable* during build, but can be published as standalone.

`dotnet publish --framework netcoreapp1.0 --runtime osx.10.11-x64`

For more information, see Self-contained deployments (SCD).

tools

```
{
  "tools": {
    "Microsoft.EntityFrameworkCore.Tools.DotNet": "1.0.0-*"
  }
}
```

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="1.0.0" />
</ItemGroup>
```

NOTE

`imports` on tools are not supported in csproj. Tools that need imports will not work with the new `Microsoft.NET.Sdk`.

buildOptions

See also Files.

emitEntryPoint

```
{
  "buildOptions": {
    "emitEntryPoint": true
  }
}
```

```
<PropertyGroup>
  <OutputType>Exe</OutputType>
</PropertyGroup>
```

If `emitEntryPoint` was `false`, the value of `OutputType` is converted to `Library`, which is the default value:

```
{
  "buildOptions": {
    "emitEntryPoint": false
  }
}
```

```
<PropertyGroup>
  <OutputType>Library</OutputType>
  <!-- or, omit altogether. It defaults to 'Library' -->
</PropertyGroup>
```

keyFile

```
{
  "buildOptions": {
    "keyFile": "MyKey.snk"
  }
}
```

The `keyFile` element expands to three properties in MSBuild:

```
<PropertyGroup>
  <AssemblyOriginatorKeyFile>MyKey.snk</AssemblyOriginatorKeyFile>
  <SignAssembly>true</SignAssembly>
  <PublicSign Condition="'$(OS)' != 'Windows_NT'">true</PublicSign>
</PropertyGroup>
```

Other common build options

```
{
  "buildOptions": {
    "warningsAsErrors": true,
    "nowarn": ["CS0168", "CS0219"],
    "xmlDoc": true,
    "preserveCompilationContext": true,
    "outputName": "Different.AssemblyName",
    "debugType": "portable",
    "allowUnsafe": true,
    "define": ["TEST", "OTHERCONDITION"]
  }
}
```

```
<PropertyGroup>
  <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
  <NoWarn>$(NoWarn);CS0168;CS0219</NoWarn>
  <GenerateDocumentationFile>true</GenerateDocumentationFile>
  <PreserveCompilationContext>true</PreserveCompilationContext>
  <AssemblyName>Different.AssemblyName</AssemblyName>
  <DebugType>portable</DebugType>
  <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
  <DefineConstants>$(DefineConstants);TEST;OTHERCONDITION</DefineConstants>
</PropertyGroup>
```

packOptions

See also Files.

Common pack options

```
{
  "packOptions": {
    "summary": "numl is a machine learning library intended to ease the use of using standard modeling techniques for both prediction and clustering.",
    "tags": ["machine learning", "framework"],
    "releaseNotes": "Version 0.9.12-beta",
    "iconUrl": "http://numl.net/images/ico.png",
    "projectUrl": "http://numl.net",
    "licenseUrl": "https://raw.githubusercontent.com/sethjuarez/numl/master/LICENSE.md",
    "requireLicenseAcceptance": false,
    "repository": {
      "type": "git",
      "url": "https://raw.githubusercontent.com/sethjuarez/numl"
    },
    "owners": ["Seth Juarez"]
  }
}
```

```
<PropertyGroup>
  <!-- summary is not migrated from project.json, but you can use the <Description> property for that if needed. -->
  <PackageTags>machine learning;framework</PackageTags>
  <PackageReleaseNotes>Version 0.9.12-beta</PackageReleaseNotes>
  <PackageIconUrl>http://numl.net/images/ico.png</PackageIconUrl>
  <PackageProjectUrl>http://numl.net</PackageProjectUrl>
  <PackageLicenseUrl>https://raw.githubusercontent.com/sethjuarez/numl/master/LICENSE.md</PackageLicenseUrl>
  <PackageRequireLicenseAcceptance>false</PackageRequireLicenseAcceptance>
  <RepositoryType>git</RepositoryType>
  <RepositoryUrl>https://raw.githubusercontent.com/sethjuarez/numl</RepositoryUrl>
  <!-- owners is not supported in MSBuild -->
</PropertyGroup>
```

There is no equivalent for the `owners` element in MSBuild. For `summary`, you can use the MSBuild `<Description>` property, even though the value of `summary` is not migrated automatically to that property, since that property is mapped to the `description` element.

scripts

```
{
  "scripts": {
    "precompile": "generateCode.cmd",
    "postpublish": [ "obfuscate.cmd", "removeTempFiles.cmd" ]
  }
}
```

Their equivalent in MSBuild are targets:

```
<Target Name="MyPreCompileTarget" BeforeTargets="Build">
  <Exec Command="generateCode.cmd" />
</Target>

<Target Name="MyPostCompileTarget" AfterTargets="Publish">
  <Exec Command="obfuscate.cmd" />
  <Exec Command="removeTempFiles.cmd" />
</Target>
```

runtimeOptions

```
{
  "runtimeOptions": {
    "configProperties": {
      "System.GC.Server": true,
      "System.GC.Concurrent": true,
      "System.GC.RetainVM": true,
      "System.Threading.ThreadPool.MinThreads": 4,
      "System.Threading.ThreadPool.MaxThreads": 25
    }
  }
}
```

All settings in this group, except for the "System.GC.Server" property, are placed into a file called *runtimeconfig.template.json* in the project folder, with options lifted to the root object during the migration process:

```
{
  "configProperties": {
    "System.GC.Concurrent": true,
    "System.GC.RetainVM": true,
    "System.Threading.ThreadPool.MinThreads": 4,
    "System.Threading.ThreadPool.MaxThreads": 25
  }
}
```

The "System.GC.Server" property is migrated into the csproj file:

```
<PropertyGroup>
  <ServerGarbageCollection>true</ServerGarbageCollection>
</PropertyGroup>
```

However, you can set all those values in the csproj as well as MSBuild properties:

```
<PropertyGroup>
  <ServerGarbageCollection>true</ServerGarbageCollection>
  <ConcurrentGarbageCollection>true</ConcurrentGarbageCollection>
  <RetainVMGarbageCollection>true</RetainVMGarbageCollection>
  <ThreadPoolMinThreads>4</ThreadPoolMinThreads>
  <ThreadPoolMaxThreads>25</ThreadPoolMaxThreads>
</PropertyGroup>
```

shared

```
{
  "shared": "shared/**/*.cs"
}
```

Not supported in csproj. You must instead create include content files in your *.nuspec* file. For more information, see Including content files.

files

In *project.json*, build and pack could be extended to compile and embed from different folders. In MSBuild, this is done using items. The following example is a common conversion:

```
{
  "buildOptions": {
    "compile": {
      "copyToOutput": "notes.txt",
      "include": "../Shared/*.cs",
      "exclude": "../Shared/Not/*.cs"
    },
    "embed": {
      "include": "../Shared/*.resx"
    }
  },
  "packOptions": {
    "include": "Views/",
    "mappings": {
      "some/path/in/project.txt": "in/package.txt"
    }
  },
  "publishOptions": {
    "include": [
      "files/",
      "publishnotes.txt"
    ]
  }
}
```

```
<ItemGroup>
  <Compile Include="..\Shared\*.cs" Exclude="..\Shared\Not\*.cs" />
  <EmbeddedResource Include="..\Shared\*.resx" />
  <Content Include="Views\**\*" PackagePath="%(Identity)" />
  <None Include="some/path/in/project.txt" Pack="true" PackagePath="in/package.txt" />

  <None Include="notes.txt" CopyToOutputDirectory="Always" />
  <!-- CopyToOutputDirectory = { Always, PreserveNewest, Never } -->

  <Content Include="files\**\*" CopyToPublishDirectory="PreserveNewest" />
  <None Include="publishnotes.txt" CopyToPublishDirectory="Always" />
  <!-- CopyToPublishDirectory = { Always, PreserveNewest, Never } -->
</ItemGroup>
```

NOTE

Many of the default globbing patterns are added automatically by the .NET Core SDK. For more information, see Default Compile Item Values.

All MSBuild `ItemGroup` elements support `Include`, `Exclude`, and `Remove`.

Package layout inside the .nupkg can be modified with `PackagePath="path"`.

Except for `Content`, most item groups require explicitly adding `Pack="true"` to be included in the package. `Content` will be put in the *content* folder in a package since the MSBuild `<IncludeContentInPack>` property is set to `true` by default. For more information, see Including content in a package.

`PackagePath="%(Identity)"` is a short way of setting package path to the project-relative file path.

testRunner

xUnit

```
{
  "testRunner": "xunit",
  "dependencies": {
    "dotnet-test-xunit": "<any>"
  }
}
```

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.0.0-*" />
  <PackageReference Include="xunit" Version="2.2.0-*" />
  <PackageReference Include="xunit.runner.visualstudio" Version="2.2.0-*" />
</ItemGroup>
```

MSTest

```
{
  "testRunner": "mstest",
  "dependencies": {
    "dotnet-test-mstest": "<any>"
  }
}
```

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.0.0-*" />
  <PackageReference Include="MSTest.TestAdapter" Version="1.1.12-*" />
  <PackageReference Include="MSTest.TestFramework" Version="1.1.11-*" />
</ItemGroup>
```

See also

- High-level overview of changes in CLI

Migrating from DNX to .NET Core CLI (project.json)

4/8/2019 • 8 minutes to read • Edit Online

Overview

The RC1 release of .NET Core and ASP.NET Core 1.0 introduced DNX tooling. The RC2 release of .NET Core and ASP.NET Core 1.0 moved from DNX to the .NET Core CLI.

As a slight refresher, let's recap what DNX was about. DNX was a runtime and a toolset used to build .NET Core and, more specifically, ASP.NET Core 1.0 applications. It consisted of 3 main pieces:

1. DNVM - an install script for obtaining DNX
2. DNX (Dotnet Execution Runtime) - the runtime that executes your code
3. DNU (Dotnet Developer Utility) - tooling for managing dependencies, building and publishing your applications

With the introduction of the CLI, all of the above are now part of a single toolset. However, since DNX was available in RC1 timeframe, you might have projects that were built using it that you would want to move off to the new CLI tooling.

This migration guide will cover the essentials on how to migrate projects off of DNX and onto .NET Core CLI. If you are just starting a project on .NET Core from scratch, you can freely skip this document.

Main changes in the tooling

There are some general changes in the tooling that should be outlined first.

No more DNVM

DNVM, short for *DotNet Version Manager* was a bash/PowerShell script used to install a DNX on your machine. It helped users get the DNX they need from the feed they specified (or default ones) as well as mark a certain DNX "active", which would put it on the $PATH for the given session. This would allow you to use the various tools.

DNVM was discontinued because its feature set was made redundant by changes coming in the .NET Core CLI tools.

The CLI tools come packaged in two main ways:

1. Native installers for a given platform
2. Install script for other situations (like CI servers)

Given this, the DNVM install features are not needed. But what about the runtime selection features?

You reference a runtime in your `project.json` by adding a package of a certain version to your dependencies. With this change, your application will be able to use the new runtime bits. Getting these bits to your machine is the same as with the CLI: you install the runtime via one of the native installers it supports or via its install script.

Different commands

If you were using DNX, you used some commands from one of its three parts (DNX, DNU or DNVM). With the CLI, some of these commands change, some are not available and some are the same but have slightly different semantics.

The table below shows the mapping between the DNX/DNU commands and their CLI counterparts.

DNX COMMAND	CLI COMMAND	DESCRIPTION
dnx run	dotnet run	Run code from source.
dnu build	dotnet build	Build an IL binary of your code.
dnu pack	dotnet pack	Package up a NuGet package of your code.
dnx [command] (for example, "dnx web")	N/A*	In DNX world, run a command as defined in the project.json.
dnu install	N/A*	In the DNX world, install a package as a dependency.
dnu restore	dotnet restore	Restore dependencies specified in your project.json. (see note)
dnu publish	dotnet publish	Publish your application for deployment in one of the three forms (portable, portable with native and standalone).
dnu wrap	N/A*	In DNX world, wrap a project.json in csproj.
dnu commands	N/A*	In DNX world, manage the globally installed commands.

(*) - these features are not supported in the CLI by design.

DNX features that are not supported

As the table above shows, there are features from the DNX world that we decided not to support in the CLI, at least for the time being. This section will go through the most important ones and outline the rationale behind not supporting them as well as workarounds if you do need them.

Global commands

DNU came with a concept called "global commands". These were, essentially, console applications packaged up as NuGet packages with a shell script that would invoke the DNX you specified to run the application.

The CLI does not support this concept. It does, however, support the concept of adding per-project commands that can be invoked using the familiar `dotnet <command>` syntax.

Installing dependencies

As of v1, the .NET Core CLI tools don't have an `install` command for installing dependencies. In order to install a package from NuGet, you would need to add it as a dependency to your `project.json` file and then run `dotnet restore` (see note).

Running your code

There are two main ways to run your code. One is from source, with `dotnet run`. Unlike `dnx run`, this will not do any in-memory compilation. It will actually invoke `dotnet build` to build your code and then run the built binary.

Another way is using the `dotnet` itself to run your code. This is done by providing a path to your assembly: `dotnet path/to/an/assembly.dll`.

Migrating your DNX project to .NET Core CLI

In addition to using new commands when working with your code, there are three major things left in migrating from DNX:

1. Migrate the `global.json` file if you have it to be able to use CLI.
2. Migrating the project file (`project.json`) itself to the CLI tooling.
3. Migrating off of any DNX APIs to their BCL counterparts.

Changing the global.json file

The `global.json` file acts like a solution file for both the RC1 and RC2 (or later) projects. In order for the CLI tools (as well as Visual Studio) to differentiate between RC1 and later versions, they use the `"sdk": { "version" }` property to make the distinction which project is RC1 or later. If `global.json` doesn't have this node at all, it is assumed to be the latest.

In order to update the `global.json` file, either remove the property or set it to the exact version of the tools that you wish to use, in this case **1.0.0-preview2-003121**:

```
{
    "sdk": {
        "version": "1.0.0-preview2-003121"
    }
}
```

Migrating the project file

The CLI and DNX both use the same basic project system based on `project.json` file. The syntax and the semantics of the project file are pretty much the same, with small differences based on the scenarios. There are also some changes to the schema which you can see in the schema file.

If you are building a console application, you need to add the following snippet to your project file:

```
"buildOptions": {
    "emitEntryPoint": true
}
```

This instructs `dotnet build` to emit an entry point for your application, effectively making your code runnable. If you are building a class library, simply omit the above section. Of course, once you add the above snippet to your `project.json` file, you need to add a static entry point. With the move off DNX, the DI services it provided are no longer available and thus this needs to be a basic .NET entry point: `static void Main()`.

If you have a "commands" section in your `project.json`, you can remove it. Some of the commands that used to exist as DNU commands, such as Entity Framework CLI commands, are being ported to be per-project extensions to the CLI. If you built your own commands that you are using in your projects, you need to replace them with CLI extensions. In this case, the `commands` node in `project.json` needs to be replaced by the `tools` node and it needs to list the tools dependencies.

After these things are done, you need to decide which type of portability you wish for you app. With .NET Core, we have invested into providing a spectrum of portability options that you can choose from. For instance, you may want to have a fully *portable* application or you may want to have a *self-contained* application. The portable application option is more like .NET Framework applications work: it needs a shared component to execute it on the target machine (.NET Core). The self-contained application doesn't require .NET Core to be installed on the target, but you have to produce one application for each OS you wish to support. These portability types and more are discussed in the application portability type document.

Once you make a call on what type of portability you want, you need to change your targeted framework(s). If you were writing applications for .NET Core, you were most likely using `dnxcore50` as your targeted framework. With the CLI and the changes that the new .NET Standard brought, the framework needs to be one of the following:

1. `netcoreapp1.0` - if you are writing applications on .NET Core (including ASP.NET Core applications)
2. `netstandard1.6` - if you are writing class libraries for .NET Core

If you are using other `dnx` targets, like `dnx451` you will need to change those as well. `dnx451` should be changed to `net451`. Please refer to the .NET Standard topic for more information.

Your `project.json` is now mostly ready. You need to go through your dependencies list and update the dependencies to their newer versions, especially if you are using ASP.NET Core dependencies. If you were using separate packages for BCL APIs, you can use the runtime package as explained in the application portability type document.

Once you are ready, you can try restoring with `dotnet restore` (see note). Depending on the version of your dependencies, you may encounter errors if NuGet cannot resolve the dependencies for one of the targeted frameworks above. This is a "point-in-time" problem; as time progresses, more and more packages will include support for these frameworks. For now, if you run into this, you can use the `imports` statement within the `framework` node to specify to NuGet that it can restore the packages targeting the framework within the "imports" statement. The restoring errors you get in this case should provide enough information to tell you which frameworks you need to import. If you are slightly lost or new to this, in general, specifying `dnxcore50` and `portable-net45+win8` in the `imports` statement should do the trick. The JSON snippet below shows how this looks like:

```
"frameworks": {
    "netcoreapp1.0": {
        "imports": ["dnxcore50", "portable-net45+win8"]
    }
}
```

Running `dotnet build` will show any eventual build errors, though there shouldn't be too many of them. After your code is building and running properly, you can test it out with the runner. Execute `dotnet <path-to-your-assembly>` and see it run.

NOTE

Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

.NET Core application deployment

10/30/2019 • 5 minutes to read • Edit Online

You can create three types of deployments for .NET Core applications:

- Framework-dependent deployment. As the name implies, framework-dependent deployment (FDD) relies on the presence of a shared system-wide version of .NET Core on the target system. Because .NET Core is already present, your app is also portable between installations of .NET Core. Your app contains only its own code and any third-party dependencies that are outside of the .NET Core libraries. FDDs contain *.dll* files that can be launched by using the dotnet utility from the command line. For example, `dotnet app.dll` runs an application named `app`.

- Self-contained deployment. Unlike FDD, a self-contained deployment (SCD) doesn't rely on the presence of shared components on the target system. All components, including both the .NET Core libraries and the .NET Core runtime, are included with the application and are isolated from other .NET Core applications. SCDs include an executable (such as *app.exe* on Windows platforms for an application named `app`), which is a renamed version of the platform-specific .NET Core host, and a *.dll* file (such as *app.dll*), which is the actual application.

- Framework-dependent executables. Produces an executable that runs on a target platform. Similar to FDDs, framework-dependent executables (FDE) are platform-specific and aren't self-contained. These deployments still rely on the presence of a shared system-wide version of .NET Core to run. Unlike an SCD, your app only contains your code and any third-party dependencies that are outside of the .NET Core libraries. FDEs produce an executable that runs on the target platform.

Framework-dependent deployments (FDD)

For an FDD, you deploy only your app and third-party dependencies. Your app will use the version of .NET Core that's present on the target system. This is the default deployment model for .NET Core and ASP.NET Core apps that target .NET Core.

Why create a framework-dependent deployment?

Deploying an FDD has a number of advantages:

- You don't have to define the target operating systems that your .NET Core app will run on in advance. Because .NET Core uses a common PE file format for executables and libraries regardless of operating system, .NET Core can execute your app regardless of the underlying operating system. For more information on the PE file format, see .NET Assembly File Format.

- The size of your deployment package is small. You only deploy your app and its dependencies, not .NET Core itself.

- Unless overridden, FDDs will use the latest serviced runtime installed on the target system. This allows your application to use the latest patched version of the .NET Core runtime.

- Multiple apps use the same .NET Core installation, which reduces both disk space and memory usage on host systems.

There are also a few disadvantages:

- Your app can run only if the version of .NET Core your app targets, or a later version, is already installed on the host system.

- It's possible for the .NET Core runtime and libraries to change without your knowledge in future releases. In rare cases, this may change the behavior of your app.

Self-contained deployments (SCD)

For a self-contained deployment, you deploy your app and any required third-party dependencies along with the version of .NET Core that you used to build the app. Creating an SCD doesn't include the native dependencies of .NET Core on various platforms, so these must be present before the app runs. For more information on version binding at runtime, see the article on version binding in .NET Core.

Starting with NET Core 2.1 SDK (version 2.1.300), .NET Core supports *patch version roll forward*. When you create a self-contained deployment, .NET Core tools automatically include the latest serviced runtime of the .NET Core version that your application targets. (The latest serviced runtime includes security patches and other bug fixes.) The serviced runtime does not have to be present on your build system; it is downloaded automatically from NuGet.org. For more information, including instructions on how to opt out of patch version roll forward, see Self-contained deployment runtime roll forward.

FDD and SCD deployments use separate host executables, so you can sign a host executable for an SCD with your publisher signature.

Why deploy a self-contained deployment?

Deploying a Self-contained deployment has two major advantages:

- You have sole control of the version of .NET Core that is deployed with your app. .NET Core can be serviced only by you.

- You can be assured that the target system can run your .NET Core app, since you're providing the version of .NET Core that it will run on.

It also has a number of disadvantages:

- Because .NET Core is included in your deployment package, you must select the target platforms for which you build deployment packages in advance.

- The size of your deployment package is relatively large, since you have to include .NET Core as well as your app and its third-party dependencies.

  Starting with .NET Core 2.0, you can reduce the size of your deployment on Linux systems by approximately 28 MB by using .NET Core *globalization invariant mode*. Ordinarily, .NET Core on Linux relies on the ICU libraries for globalization support. In invariant mode, the libraries are not included with your deployment, and all cultures behave like the invariant culture.

- Deploying numerous self-contained .NET Core apps to a system can consume significant amounts of disk space, since each app duplicates .NET Core files.

Framework-dependent executables (FDE)

Starting with .NET Core 2.2, you can deploy your app as an FDE, along with any required third-party dependencies. Your app will use the version of .NET Core that's installed on the target system.

Why deploy a framework-dependent executable?

Deploying an FDE has a number of advantages:

- The size of your deployment package is small. You only deploy your app and its dependencies, not .NET Core itself.

- Multiple apps use the same .NET Core installation, which reduces both disk space and memory usage on host systems.

- Your app can be run by calling the published executable without invoking the `dotnet` utility directly.

There are also a few disadvantages:

- Your app can run only if the version of .NET Core your app targets, or a later version, is already installed on the host system.
- It's possible for the .NET Core runtime and libraries to change without your knowledge in future releases. In rare cases, this may change the behavior of your app.
- You must publish your app for each target platform.

Step-by-step examples

For step-by-step examples of deploying .NET Core apps with CLI tools, see Deploying .NET Core Apps with CLI Tools. For step-by-step examples of deploying .NET Core apps with Visual Studio, see Deploying .NET Core Apps with Visual Studio.

See also

- Deploying .NET Core Apps with CLI Tools
- Deploying .NET Core Apps with Visual Studio
- Packages, Metapackages and Frameworks
- .NET Core Runtime IDentifier (RID) catalog

Publish .NET Core apps with the CLI

10/3/2019 • 7 minutes to read • Edit Online

This article demonstrates how you can publish your .NET Core application from the command line. .NET Core provides three ways to publish your applications. Framework-dependent deployment produces a cross-platform .dll file that uses the locally installed .NET Core runtime. Framework-dependent executable produces a platform-specific executable that uses the locally installed .NET Core runtime. Self-contained executable produces a platform-specific executable and includes a local copy of the .NET Core runtime.

For an overview of these publishing modes, see .NET Core Application Deployment.

Looking for some quick help on using the CLI? The following table shows some examples of how to publish your app. You can specify the target framework with the `-f <TFM>` parameter or by editing the project file. For more information, see Publishing basics.

PUBLISH MODE	SDK VERSION	COMMAND
Framework-dependent deployment	2.x	`dotnet publish -c Release`
Framework-dependent executable	2.2	`dotnet publish -c Release -r <RID> --self-contained false`
	3.0	`dotnet publish -c Release -r <RID> --self-contained false`
	3.0*	`dotnet publish -c Release`
Self-contained deployment	2.1	`dotnet publish -c Release -r <RID> --self-contained true`
	2.2	`dotnet publish -c Release -r <RID> --self-contained true`
	3.0	`dotnet publish -c Release -r <RID> --self-contained true`

* When using SDK version 3.0, framework-dependent executable is the default publishing mode when running the basic `dotnet publish` command. This only applies when the project targets either **.NET Core 2.1** or **.NET Core 3.0**.

Publishing basics

The `<TargetFramework>` setting of the project file specifies the default target framework when you publish your app. You can change the target framework to any valid Target Framework Moniker (TFM). For example, if your project uses `<TargetFramework>netcoreapp2.2</TargetFramework>`, a binary that targets .NET Core 2.2 is created. The TFM specified in this setting is the default target used by the `dotnet publish` command.

If you want to target more than one framework, you can set the `<TargetFrameworks>` setting to more than one TFM value separated by a semicolon. You can publish one of the frameworks with the `dotnet publish -f <TFM>` command. For example, if you have `<TargetFrameworks>netcoreapp2.1;netcoreapp2.2</TargetFrameworks>` and run `dotnet publish -f netcoreapp2.1`, a binary that targets .NET Core 2.1 is created.

Unless otherwise set, the output directory of the `dotnet publish` command is `./bin/<BUILD-CONFIGURATION>/<TFM>/publish/`. The default **BUILD-CONFIGURATION** mode is **Debug** unless changed with the `-c` parameter. For example, `dotnet publish -c Release -f netcoreapp2.1` publishes to `myfolder/bin/Release/netcoreapp2.1/publish/`.

If you use .NET Core SDK 3.0, the default publish mode for apps that target .NET Core versions 2.1, 2.2, or 3.0 is framework-dependent executable.

If you use .NET Core SDK 2.1, the default publish mode for apps that target .NET Core versions 2.1, 2.2 is framework-dependent deployment.

Native dependencies

If your app has native dependencies, it may not run on a different operating system. For example, if your app uses the native Windows API, it won't run on macOS or Linux. You would need to provide platform-specific code and compile an executable for each platform.

Consider also, if a library you referenced has a native dependency, your app may not run on every platform. However, it's possible a NuGet package you're referencing has included platform-specific versions to handle the required native dependencies for you.

When distributing an app with native dependencies, you may need to use the `dotnet publish -r <RID>` switch to specify the target platform you want to publish for. For a list of runtime identifiers, see Runtime Identifier (RID) catalog.

More information about platform-specific binaries is covered in the Framework-dependent executable and Self-contained deployment sections.

Sample app

You can use the following app to explore the publishing commands. The app is created by running the following commands in your terminal:

```
mkdir apptest1
cd apptest1
dotnet new console
dotnet add package Figgle
```

The `Program.cs` or `Program.vb` file that is generated by the console template needs to be changed to the following:

```
using System;

namespace apptest1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(Figgle.FiggleFonts.Standard.Render("Hello, World!"));
        }
    }
}
```

```
Imports System

Module Program
    Sub Main(args As String())
        Console.WriteLine(Figgle.FiggleFonts.Standard.Render("Hello, World!"))
    End Sub
End Module
```

When you run the app (`dotnet run`), the following output is displayed:

```
 _   _      _ _         __        __         _     _ _
| | | | ___| | | ___    \ \      / /__  _ __| | __| | |
| |_| |/ _ \ | |/ _ \    \ \ /\ / / _ \| '__| |/ _` | |
|  _  |  __/ | | (_) |    \ V  V / (_) | |  | | (_| |_|
|_| |_|\___|_|_|\___( )    \_/\_/ \___/|_|  |_|\__,_(_)
                    |/
```

Framework-dependent deployment

For the .NET Core SDK 2.x CLI, framework-dependent deployment (FDD) is the default mode for the basic `dotnet publish` command.

When you publish your app as an FDD, a `<PROJECT-NAME>.dll` file is created in the `./bin/<BUILD-CONFIGURATION>/<TFM>/publish/` folder. To run your app, navigate to the output folder and use the `dotnet <PROJECT-NAME>.dll` command.

Your app is configured to target a specific version of .NET Core. That targeted .NET Core runtime is required to be on the machine where you want to run your app. For example, if your app targets .NET Core 2.2, any machine that your app runs on must have the .NET Core 2.2 runtime installed. As stated in the Publishing basics section, you can edit your project file to change the default target framework or to target more than one framework.

Publishing an FDD creates an app that automatically rolls-forward to the latest .NET Core security patch available on the system that runs the app. For more information on version binding at compile time, see Select the .NET Core version to use.

Framework-dependent executable

For the .NET Core SDK 3.x CLI, framework-dependent executable (FDE) is the default mode for the basic `dotnet publish` command. You don't need to specify any other parameters as long as you want to target the current operating system.

In this mode, a platform-specific executable host is created to host your cross-platform app. This mode is similar to FDD as FDD requires a host in the form of the `dotnet` command. The host executable filename varies per platform, and is named something similar to `<PROJECT-FILE>.exe`. You can run this executable directly instead of calling `dotnet <PROJECT-FILE>.dll` which is still an acceptable way to run the app.

Your app is configured to target a specific version of .NET Core. That targeted .NET Core runtime is required to be on the machine where you want to run your app. For example, if your app targets .NET Core 2.2, any machine that your app runs on must have the .NET Core 2.2 runtime installed. As stated in the Publishing basics section, you can edit your project file to change the default target framework or to target more than one framework.

Publishing an FDE creates an app that automatically rolls-forward to the latest .NET Core security patch available on the system that runs the app. For more information on version binding at compile time, see Select the .NET Core version to use.

You must (except for .NET Core 3.x when you target the current platform) use the following switches with the `dotnet publish` command to publish an FDE:

- `-r <RID>` This switch uses an identifier (RID) to specify the target platform. For a list of runtime identifiers, see Runtime Identifier (RID) catalog.
- `--self-contained false` This switch tells the .NET Core SDK to create an executable as an FDE.

Whenever you use the `-r` switch, the output folder path changes to: `./bin/<BUILD-CONFIGURATION>/<TFM>/<RID>/publish/`

If you use the example app, run `dotnet publish -f netcoreapp2.2 -r win10-x64 --self-contained false`. This command creates the following executable: `./bin/Debug/netcoreapp2.2/win10-x64/publish/apptest1.exe`

NOTE

You can reduce the total size of your deployment by enabling **globalization invariant mode**. This mode is useful for applications that are not globally aware and that can use the formatting conventions, casing conventions, and string comparison and sort order of the invariant culture. For more information about **globalization invariant mode** and how to enable it, see .NET Core Globalization Invariant Mode.

Self-contained deployment

When you publish a self-contained deployment (SCD), the .NET Core SDK creates a platform-specific executable. Publishing an SCD includes all required .NET Core files to run your app but it doesn't include the native dependencies of .NET Core. These dependencies must be present on the system before the app runs.

Publishing an SCD creates an app that doesn't roll-forward to the latest available .NET Core security patch. For more information on version binding at compile time, see Select the .NET Core version to use.

You must use the following switches with the `dotnet publish` command to publish an SCD:

- `-r <RID>` This switch uses an identifier (RID) to specify the target platform. For a list of runtime identifiers, see Runtime Identifier (RID) catalog.
- `--self-contained true` This switch tells the .NET Core SDK to create an executable as an SCD.

NOTE

You can reduce the total size of your deployment by enabling **globalization invariant mode**. This mode is useful for applications that are not globally aware and that can use the formatting conventions, casing conventions, and string comparison and sort order of the invariant culture. For more information about **globalization invariant mode** and how to enable it, see .NET Core Globalization Invariant Mode.

See also

- .NET Core Application Deployment Overview
- .NET Core Runtime IDentifier (RID) catalog

Deploy .NET Core apps with Visual Studio

10/30/2019 • 15 minutes to read • Edit Online

You can deploy a .NET Core application either as a *framework-dependent deployment*, which includes your application binaries but depends on the presence of .NET Core on the target system, or as a *self-contained deployment*, which includes both your application and .NET Core binaries. For an overview of .NET Core application deployment, see .NET Core Application Deployment.

The following sections show how to use Microsoft Visual Studio to create the following kinds of deployments:

- Framework-dependent deployment
- Framework-dependent deployment with third-party dependencies
- Self-contained deployment
- Self-contained deployment with third-party dependencies

For information on using Visual Studio to develop .NET Core applications, see Prerequisites for .NET Core on Windows.

Framework-dependent deployment

Deploying a framework-dependent deployment with no third-party dependencies simply involves building, testing, and publishing the app. A simple example written in C# illustrates the process.

1. Create the project.

   Select **File** > **New** > **Project**. In the **New Project** dialog, expand your language's (C# or Visual Basic) project categories in the **Installed** project types pane, choose **.NET Core**, and then select the **Console App (.NET Core)** template in the center pane. Enter a project name, such as "FDD", in the **Name** text box. Select the **OK** button.

2. Add the application's source code.

   Open the *Program.cs* or *Program.vb* file in the editor and replace the auto-generated code with the following code. It prompts the user to enter text and displays the individual words entered by the user. It uses the regular expression `\w+` to separate the words in the input text.

```
using System;
using System.Text.RegularExpressions;

namespace Applications.ConsoleApps
{
    public class ConsoleParser
    {
        public static void Main()
        {
            Console.WriteLine("Enter any text, followed by <Enter>:\n");
            String s = Console.ReadLine();
            ShowWords(s);
            Console.Write("\nPress any key to continue... ");
            Console.ReadKey();
        }

        private static void ShowWords(String s)
        {
            String pattern = @"\w+";
            var matches = Regex.Matches(s, pattern);
            if (matches.Count == 0)
            {
                Console.WriteLine("\nNo words were identified in your input.");
            }
            else
            {
                Console.WriteLine($"\nThere are {matches.Count} words in your string:");
                for (int ctr = 0; ctr < matches.Count; ctr++)
                {
                    Console.WriteLine($"   #{ctr,2}: '{matches[ctr].Value}' at position
{matches[ctr].Index}");
                }
            }
        }
    }
}
```

```
Imports System.Text.RegularExpressions

Namespace Applications.ConsoleApps
    Public Module ConsoleParser
        Public Sub Main()
            Console.WriteLine("Enter any text, followed by <Enter>:")
            Console.WriteLine()
            Dim s = Console.ReadLine()
            ShowWords(s)
            Console.Write($"{vbCrLf}Press any key to continue... ")
            Console.ReadKey()
        End Sub

        Private Sub ShowWords(s As String)
            Dim pattern = "\w+"
            Dim matches = Regex.Matches(s, pattern)
            Console.WriteLine()
            If matches.Count = 0 Then
                Console.WriteLine("No words were identified in your input.")
            Else
                Console.WriteLine($"There are {matches.Count} words in your string:")
                For ctr = 0 To matches.Count - 1
                    Console.WriteLine($"   #{ctr,2}: '{matches(ctr).Value}' at position
{matches(ctr).Index}")
                Next
            End If
            Console.WriteLine()
        End Sub
    End Module
End Namespace
```

3. Create a Debug build of your app.

   Select **Build** > **Build Solution**. You can also compile and run the Debug build of your application by selecting **Debug** > **Start Debugging**.

4. Deploy your app.

   After you've debugged and tested the program, create the files to be deployed with your app. To publish from Visual Studio, do the following:

   a. Change the solution configuration from **Debug** to **Release** on the toolbar to build a Release (rather than a Debug) version of your app.

   b. Right-click on the project (not the solution) in **Solution Explorer** and select **Publish**.

   c. In the **Publish** tab, select **Publish**. Visual Studio writes the files that comprise your application to the local file system.

   d. The **Publish** tab now shows a single profile, **FolderProfile**. The profile's configuration settings are shown in the **Summary** section of the tab.

   The resulting files are placed in a directory named `Publish` on Windows and `publish` on Unix systems that is in a subdirectory of your project's *.\bin\release\netcoreapp2.1* subdirectory.

Along with your application's files, the publishing process emits a program database (.pdb) file that contains debugging information about your app. The file is useful primarily for debugging exceptions. You can choose not to package it with your application's files. You should, however, save it in the event that you want to debug the Release build of your app.

Deploy the complete set of application files in any way you like. For example, you can package them in a Zip file, use a simple `copy` command, or deploy them with any installation package of your choice. Once installed, users can then execute your application by using the `dotnet` command and providing the application filename, such as `dotnet fdd.dll`.

In addition to the application binaries, your installer should also either bundle the shared framework installer or check for it as a prerequisite as part of the application installation. Installation of the shared framework requires Administrator/root access since it is machine-wide.

Framework-dependent deployment with third-party dependencies

Deploying a framework-dependent deployment with one or more third-party dependencies requires that any dependencies be available to your project. The following additional steps are required before you can build your app:

1. Use the **NuGet Package Manager** to add a reference to a NuGet package to your project; and if the package is not already available on your system, install it. To open the package manager, select **Tools** > **NuGet Package Manager** > **Manage NuGet Packages for Solution**.

2. Confirm that `Newtonsoft.Json` is installed on your system and, if it is not, install it. The **Installed** tab lists NuGet packages installed on your system. If `Newtonsoft.Json` is not listed there, select the **Browse** tab and enter "Newtonsoft.Json" in the search box. Select `Newtonsoft.Json` and, in the right pane, select your project before selecting **Install**.

3. If `Newtonsoft.Json` is already installed on your system, add it to your project by selecting your project in the right pane of the **Manage Packages for Solution** tab.

Note that a framework-dependent deployment with third-party dependencies is only as portable as its third-party dependencies. For example, if a third-party library only supports macOS, the app isn't portable to Windows systems. This happens if the third-party dependency itself depends on native code. A good example of this is Kestrel server, which requires a native dependency on libuv. When an FDD is created for an application with this kind of third-party dependency, the published output contains a folder for each Runtime Identifier (RID) that the native dependency supports (and that exists in its NuGet package).

Self-contained deployment without third-party dependencies

Deploying a self-contained deployment with no third-party dependencies involves creating the project, modifying the *csproj* file, building, testing, and publishing the app. A simple example written in C# illustrates the process. You begin by creating, coding, and testing your project just as you would a framework-dependent deployment:

1. Create the project.

   Select **File** > **New** > **Project**. In the **New Project** dialog, expand your language's (C# or Visual Basic) project categories in the **Installed** project types pane, choose **.NET Core**, and then select the **Console App (.NET Core)** template in the center pane. Enter a project name, such as "SCD", in the **Name** text box, and select the **OK** button.

2. Add the application's source code.

   Open the *Program.cs* or *Program.vb* file in your editor, and replace the auto-generated code with the following code. It prompts the user to enter text and displays the individual words entered by the user. It uses the regular expression `\w+` to separate the words in the input text.

```
using System;
using System.Text.RegularExpressions;

namespace Applications.ConsoleApps
{
    public class ConsoleParser
    {
        public static void Main()
        {
            Console.WriteLine("Enter any text, followed by <Enter>:\n");
            String s = Console.ReadLine();
            ShowWords(s);
            Console.Write("\nPress any key to continue... ");
            Console.ReadKey();
        }

        private static void ShowWords(String s)
        {
            String pattern = @"\w+";
            var matches = Regex.Matches(s, pattern);
            if (matches.Count == 0)
            {
                Console.WriteLine("\nNo words were identified in your input.");
            }
            else
            {
                Console.WriteLine($"\nThere are {matches.Count} words in your string:");
                for (int ctr = 0; ctr < matches.Count; ctr++)
                {
                    Console.WriteLine($"   #{ctr,2}: '{matches[ctr].Value}' at position
{matches[ctr].Index}");
                }
            }
        }
    }
}
```

```vb
Imports System.Text.RegularExpressions

Namespace Applications.ConsoleApps
    Public Module ConsoleParser
        Public Sub Main()
            Console.WriteLine("Enter any text, followed by <Enter>:")
            Console.WriteLine()
            Dim s = Console.ReadLine()
            ShowWords(s)
            Console.Write($"{vbCrLf}Press any key to continue... ")
            Console.ReadKey()
        End Sub

        Private Sub ShowWords(s As String)
            Dim pattern = "\w+"
            Dim matches = Regex.Matches(s, pattern)
            Console.WriteLine()
            If matches.Count = 0 Then
                Console.WriteLine("No words were identified in your input.")
            Else
                Console.WriteLine($"There are {matches.Count} words in your string:")
                For ctr = 0 To matches.Count - 1
                    Console.WriteLine($"   #{ctr,2}: '{matches(ctr).Value}' at position
{matches(ctr).Index}")
                Next
            End If
            Console.WriteLine()
        End Sub
    End Module
End Namespace
```

3. Determine whether you want to use globalization invariant mode.

   Particularly if your app targets Linux, you can reduce the total size of your deployment by taking advantage of globalization invariant mode. Globalization invariant mode is useful for applications that are not globally aware and that can use the formatting conventions, casing conventions, and string comparison and sort order of the invariant culture.

   To enable invariant mode, right-click on your project (not the solution) in **Solution Explorer**, and select **Edit SCD.csproj** or **Edit SCD.vbproj**. Then add the following highlighted lines to the file:

```xml
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.2</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <RuntimeHostConfigurationOption Include="System.Globalization.Invariant" Value="true" />
  </ItemGroup>
</Project>
```

1. Create a Debug build of your application.

   Select **Build** > **Build Solution**. You can also compile and run the Debug build of your application by selecting **Debug** > **Start Debugging**. This debugging step lets you identify problems with your application when it's running on your host platform. You still will have to test it on each of your target platforms.

   If you've enabled globalization invariant mode, be particularly sure to test whether the absence of culture-sensitive data is suitable for your application.

Once you've finished debugging, you can publish your self-contained deployment:

- Visual Studio 15.6 and earlier
- Visual Studio 15.7 and later

After you've debugged and tested the program, create the files to be deployed with your app for each platform that it targets.

To publish your app from Visual Studio, do the following:

1. Define the platforms that your app will target.

   a. Right-click on your project (not the solution) in **Solution Explorer** and select **Edit SCD.csproj**.

   b. Create a `<RuntimeIdentifiers>` tag in the `<PropertyGroup>` section of your *csproj* file that defines the platforms your app targets, and specify the runtime identifier (RID) of each platform that you target. Note that you also need to add a semicolon to separate the RIDs. See Runtime IDentifier catalog for a list of runtime identifiers.

   For example, the following example indicates that the app runs on 64-bit Windows 10 operating systems and the 64-bit OS X Version 10.11 operating system.

   ```
   <PropertyGroup>
      <RuntimeIdentifiers>win10-x64;osx.10.11-x64</RuntimeIdentifiers>
   </PropertyGroup>
   ```

   Note that the `<RuntimeIdentifiers>` element can go into any `<PropertyGroup>` that you have in your *csproj* file. A complete sample *csproj* file appears later in this section.

2. Publish your app.

   After you've debugged and tested the program, create the files to be deployed with your app for each platform that it targets.

   To publish your app from Visual Studio, do the following:

   a. Change the solution configuration from **Debug** to **Release** on the toolbar to build a Release (rather than a Debug) version of your app.

   b. Right-click on the project (not the solution) in **Solution Explorer** and select **Publish**.

   c. In the **Publish** tab, select **Publish**. Visual Studio writes the files that comprise your application to the local file system.

   d. The **Publish** tab now shows a single profile, **FolderProfile**. The profile's configuration settings are shown in the **Summary** section of the tab. **Target Runtime** identifies which runtime has been published, and **Target Location** identifies where the files for the self-contained deployment were written.

   e. Visual Studio by default writes all published files to a single directory. For convenience, it's best to create separate profiles for each target runtime and to place published files in a platform-specific directory. This involves creating a separate publishing profile for each target platform. So now rebuild the application for each platform by doing the following:

      a. Select **Create new profile** in the **Publish** dialog.

      b. In the **Pick a publish target** dialog, change the **Choose a folder** location to *bin\Release\PublishOutput\win10-x64*. Select **OK**.

      c. Select the new profile (**FolderProfile1**) in the list of profiles, and make sure that the **Target Runtime** is `win10-x64`. If it isn't, select **Settings**. In the **Profile Settings** dialog, change the

Target Runtime to `win10-x64` and select **Save**. Otherwise, select **Cancel**.

d. Select **Publish** to publish your app for 64-bit Windows 10 platforms.

e. Follow the previous steps again to create a profile for the `osx.10.11-x64` platform. The **Target Location** is *bin\Release\PublishOutput\osx.10.11-x64*, and the **Target Runtime** is `osx.10.11-x64`. The name that Visual Studio assigns to this profile is **FolderProfile2**.

Note that each target location contains the complete set of files (both your app files and all .NET Core files) needed to launch your app.

Along with your application's files, the publishing process emits a program database (.pdb) file that contains debugging information about your app. The file is useful primarily for debugging exceptions. You can choose not to package it with your application's files. You should, however, save it in the event that you want to debug the Release build of your app.

Deploy the published files in any way you like. For example, you can package them in a Zip file, use a simple `copy` command, or deploy them with any installation package of your choice.

The following is the complete *csproj* file for this project.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.1</TargetFramework>
    <RuntimeIdentifiers>win10-x64;osx.10.11-x64</RuntimeIdentifiers>
  </PropertyGroup>
</Project>
```

Self-contained deployment with third-party dependencies

Deploying a self-contained deployment with one or more third-party dependencies involves adding the dependencies. The following additional steps are required before you can build your app:

1. Use the **NuGet Package Manager** to add a reference to a NuGet package to your project; and if the package is not already available on your system, install it. To open the package manager, select **Tools** > **NuGet Package Manager** > **Manage NuGet Packages for Solution**.

2. Confirm that `Newtonsoft.Json` is installed on your system and, if it is not, install it. The **Installed** tab lists NuGet packages installed on your system. If `Newtonsoft.Json` is not listed there, select the **Browse** tab and enter "Newtonsoft.Json" in the search box. Select `Newtonsoft.Json` and, in the right pane, select your project before selecting **Install**.

3. If `Newtonsoft.Json` is already installed on your system, add it to your project by selecting your project in the right pane of the **Manage Packages for Solution** tab.

The following is the complete *csproj* file for this project:

- Visual Studio 15.6 and earlier
- Visual Studio 15.7 and later

```xml
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.1</TargetFramework>
    <RuntimeIdentifiers>win10-x64;osx.10.11-x64</RuntimeIdentifiers>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Newtonsoft.Json" Version="10.0.2" />
  </ItemGroup>
</Project>
```

When you deploy your application, any third-party dependencies used in your app are also contained with your application files. Third-party libraries aren't required on the system on which the app is running.

Note that you can only deploy a self-contained deployment with a third-party library to platforms supported by that library. This is similar to having third-party dependencies with native dependencies in your framework-dependent deployment, where the native dependencies won't exist on the target platform unless they were previously installed there.

See also

- .NET Core Application Deployment
- .NET Core Runtime IDentifier (RID) catalog

How to create a NuGet package with .NET Core command-line interface (CLI) tools

10/22/2019 • 2 minutes to read • Edit Online

NOTE

The following shows command-line samples using Unix. The `dotnet pack` command as shown here works the same way on Windows.

.NET Standard and .NET Core libraries are expected to be distributed as NuGet packages. This is in fact how all of the .NET Standard libraries are distributed and consumed. This is most easily done with the `dotnet pack` command.

Imagine that you just wrote an awesome new library that you would like to distribute over NuGet. You can create a NuGet package with cross platform tools to do exactly that! The following example assumes a library called **SuperAwesomeLibrary** which targets `netstandard1.0`.

If you have transitive dependencies; that is, a project which depends on another package, you'll need to make sure to restore packages for your entire solution with the `dotnet restore` command before creating a NuGet package. Failing to do so will result in the `dotnet pack` command to not work properly.

NOTE

Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

After ensuring packages are restored, you can navigate to the directory where a library lives:

```
cd src/SuperAwesomeLibrary
```

Then it's just a single command from the command line:

```
dotnet pack
```

Your */bin/Debug* folder will now look like this:

```
$ ls bin/Debug
netstandard1.0/
SuperAwesomeLibrary.1.0.0.nupkg
SuperAwesomeLibrary.1.0.0.symbols.nupkg
```

Note that this will produce a package which is capable of being debugged. If you want to build a NuGet package with release binaries, all you need to do is add the `--configuration` (or `-c`) switch and use `release` as the argument.

```
dotnet pack --configuration release
```

Your */bin* folder will now have a *release* folder containing your NuGet package with release binaries:

```
$ ls bin/release
netstandard1.0/
SuperAwesomeLibrary.1.0.0.nupkg
SuperAwesomeLibrary.1.0.0.symbols.nupkg
```

And now you have the necessary files to publish a NuGet package!

Don't confuse `dotnet pack` with `dotnet publish`

It is important to note that at no point is the `dotnet publish` command involved. The `dotnet publish` command is for deploying applications with all of their dependencies in the same bundle -- not for generating a NuGet package to be distributed and consumed via NuGet.

See also

- Quickstart: Create and publish a package

Self-contained deployment runtime roll forward

10/17/2019 • 2 minutes to read • Edit Online

.NET Core self-contained application deployments include both the .NET Core libraries and the .NET Core runtime. Starting in .NET Core 2.1 SDK (version 2.1.300), a self-contained application deployment publishes the highest patch runtime on your machine. By default, `dotnet publish` for a self-contained deployment selects the latest version installed as part of the SDK on the publishing machine. This enables your deployed application to run with security fixes (and other fixes) available during `publish`. The application must be re-published to obtain a new patch. Self-contained applications are created by specifying `-r <RID>` on the `dotnet publish` command or by specifying the runtime identifier (RID) in the project file (csproj / vbproj) or on the command line.

Patch version roll forward overview

`restore`, `build` and `publish` are `dotnet` commands that can run separately. The runtime choice is part of the `restore` operation, not `publish` or `build`. If you call `publish`, the latest patch version will be chosen. If you call `publish` with the `--no-restore` argument, then you may not get the desired patch version because a prior `restore` may not have been executed with the new self-contained application publishing policy. In this case, a build error is generated with text similar to the following:

"The project was restored using Microsoft.NETCore.App version 2.0.0, but with current settings, version 2.0.6 would be used instead. To resolve this issue, make sure the same settings are used for restore and for subsequent operations such as build or publish. Typically this issue can occur if the RuntimeIdentifier property is set during build or publish but not during restore."

NOTE

`restore` and `build` can be run implicitly as part of another command, like `publish`. When run implicitly as part of another command, they are provided with additional context so that the right artifacts are produced. When you `publish` with a runtime (for example, `dotnet publish -r linux-x64`), the implicit `restore` restores packages for the linux-x64 runtime. If you call `restore` explicitly, it does not restore runtime packages by default, because it doesn't have that context.

How to avoid restore during publish

Running `restore` as part of the `publish` operation may be undesirable for your scenario. To avoid `restore` during `publish` while creating self-contained applications, do the following:

- Set the `RuntimeIdentifiers` property to a semicolon-separated list of all the RIDs to be published.
- Set the `TargetLatestRuntimePatch` property to `true`.

No-restore argument with dotnet publish options

If you want to create both self-contained applications and framework-dependent applications with the same project file, and you want to use the `--no-restore` argument with `dotnet publish`, then choose one of the following:

1. Prefer the framework-dependent behavior. If the application is framework-dependent, this is the default behavior. If the application is self-contained, and can use an unpatched 2.1.0 local runtime, set the `TargetLatestRuntimePatch` to `false` in the project file.

2. Prefer the self-contained behavior. If the application is self-contained, this is the default behavior. If the

application is framework-dependent, and requires the latest patch installed, set `TargetLatestRuntimePatch` to `true` in the project file.

3. Take explicit control of the runtime framework version by setting `RuntimeFrameworkVersion` to the specific patch version in the project file.

Runtime package store

9/19/2019 • 5 minutes to read • Edit Online

Starting with .NET Core 2.0, it's possible to package and deploy apps against a known set of packages that exist in the target environment. The benefits are faster deployments, lower disk space usage, and improved startup performance in some cases.

This feature is implemented as a *runtime package store*, which is a directory on disk where packages are stored (typically at */usr/local/share/dotnet/store* on macOS/Linux and *C:/Program Files/dotnet/store* on Windows). Under this directory, there are subdirectories for architectures and target frameworks. The file layout is similar to the way that NuGet assets are laid out on disk:

```
\dotnet
    \store
        \x64
            \netcoreapp2.0
                \microsoft.applicationinsights
                \microsoft.aspnetcore
                ...
        \x86
            \netcoreapp2.0
                \microsoft.applicationinsights
                \microsoft.aspnetcore
                ...
```

A *target manifest* file lists the packages in the runtime package store. Developers can target this manifest when publishing their app. The target manifest is typically provided by the owner of the targeted production environment.

Preparing a runtime environment

The administrator of a runtime environment can optimize apps for faster deployments and lower disk space use by building a runtime package store and the corresponding target manifest.

The first step is to create a *package store manifest* that lists the packages that compose the runtime package store. This file format is compatible with the project file format (*csproj*).

```
<Project Sdk="Microsoft.NET.Sdk">
  <ItemGroup>
    <PackageReference Include="<NUGET_PACKAGE>" Version="<VERSION>" />
    <!-- Include additional packages here -->
  </ItemGroup>
</Project>
```

Example

The following example package store manifest (*packages.csproj*) is used to add `Newtonsoft.Json` and `Moq` to a runtime package store:

```
<Project Sdk="Microsoft.NET.Sdk">
  <ItemGroup>
    <PackageReference Include="Newtonsoft.Json" Version="10.0.3" />
    <PackageReference Include="Moq" Version="4.7.63" />
  </ItemGroup>
</Project>
```

Provision the runtime package store by executing `dotnet store` with the package store manifest, runtime, and framework:

```
dotnet store --manifest <PATH_TO_MANIFEST_FILE> --runtime <RUNTIME_IDENTIFIER> --framework <FRAMEWORK>
```

Example

```
dotnet store --manifest packages.csproj --runtime win10-x64 --framework netcoreapp2.0 --framework-version
2.0.0
```

You can pass multiple target package store manifest paths to a single `dotnet store` command by repeating the option and path in the command.

By default, the output of the command is a package store under the *.dotnet/store* subdirectory of the user's profile. You can specify a different location using the `--output <OUTPUT_DIRECTORY>` option. The root directory of the store contains a target manifest *artifact.xml* file. This file can be made available for download and be used by app authors who want to target this store when publishing.

Example

The following *artifact.xml* file is produced after running the previous example. Note that `Castle.Core` is a dependency of `Moq`, so it's included automatically and appears in the *artifacts.xml* manifest file.

```
<StoreArtifacts>
  <Package Id="Newtonsoft.Json" Version="10.0.3" />
  <Package Id="Castle.Core" Version="4.1.0" />
  <Package Id="Moq" Version="4.7.63" />
</StoreArtifacts>
```

Publishing an app against a target manifest

If you have a target manifest file on disk, you specify the path to the file when publishing your app with the `dotnet publish` command:

```
dotnet publish --manifest <PATH_TO_MANIFEST_FILE>
```

Example

```
dotnet publish --manifest manifest.xml
```

You deploy the resulting published app to an environment that has the packages described in the target manifest. Failing to do so results in the app failing to start.

Specify multiple target manifests when publishing an app by repeating the option and path (for example, `--manifest manifest1.xml --manifest manifest2.xml`). When you do so, the app is trimmed for the union of packages specified in the target manifest files provided to the command.

Specifying target manifests in the project file

An alternative to specifying target manifests with the `dotnet publish` command is to specify them in the project file as a semicolon-separated list of paths under a **<TargetManifestFiles>** tag.

```
<PropertyGroup>
  <TargetManifestFiles>manifest1.xml;manifest2.xml</TargetManifestFiles>
</PropertyGroup>
```

Specify the target manifests in the project file only when the target environment for the app is well-known, such as for .NET Core projects. This isn't the case for open-source projects. The users of an open-source project typically deploy it to different production environments. These production environments generally have different sets of packages pre-installed. You can't make assumptions about the target manifest in such environments, so you should use the `--manifest` option of `dotnet publish`.

ASP.NET Core implicit store

The ASP.NET Core implicit store applies only to ASP.NET Core 2.0. We strongly recommend applications use ASP.NET Core 2.1 and later, which does **not** use the implicit store. ASP.NET Core 2.1 and later use the shared framework.

The runtime package store feature is used implicitly by an ASP.NET Core app when the app is deployed as a framework-dependent deployment (FDD) app. The targets in `Microsoft.NET.Sdk.Web` include manifests referencing the implicit package store on the target system. Additionally, any FDD app that depends on the `Microsoft.AspNetCore.All` package results in a published app that contains only the app and its assets and not the packages listed in the `Microsoft.AspNetCore.All` metapackage. It's assumed that those packages are present on the target system.

The runtime package store is installed on the host when the .NET Core SDK is installed. Other installers may provide the runtime package store, including Zip/tarball installations of the .NET Core SDK, `apt-get`, Red Hat Yum, the .NET Core Windows Server Hosting bundle, and manual runtime package store installations.

When deploying a framework-dependent deployment (FDD) app, make sure that the target environment has the .NET Core SDK installed. If the app is deployed to an environment that doesn't include ASP.NET Core, you can opt out of the implicit store by specifying **<PublishWithAspNetCoreTargetManifest>** set to `false` in the project file as in the following example:

```
<PropertyGroup>
  <PublishWithAspNetCoreTargetManifest>false</PublishWithAspNetCoreTargetManifest>
</PropertyGroup>
```

NOTE

For self-contained deployment (SCD) apps, it's assumed that the target system doesn't necessarily contain the required manifest packages. Therefore, **<PublishWithAspNetCoreTargetManifest>** cannot be set to `true` for an SCD app.

If you deploy an application with a manifest dependency that's present in the deployment (the assembly is present in the *bin* folder), the runtime package store *isn't used* on the host for that assembly. The *bin* folder assembly is used regardless of its presence in the runtime package store on the host.

The version of the dependency indicated in the manifest must match the version of the dependency in the runtime package store. If you have a version mismatch between the dependency in the target manifest and the version that exists in the runtime package store and the app doesn't include the required version of the package in its deployment, the app fails to start. The exception includes the name of the target manifest that called for the runtime package store assembly, which helps you troubleshoot the mismatch.

When the deployment is *trimmed* on publish, only the specific versions of the manifest packages you indicate are withheld from the published output. The packages at the versions indicated must be present on the host for the app to start.

See also

- dotnet-publish
- dotnet-store

Introduction to .NET and Docker

11/14/2019 • 3 minutes to read • Edit Online

.NET Core can easily run in a Docker container. Containers provide a lightweight way to isolate your application from the rest of the host system, sharing just the kernel, and using resources given to your application. If you're unfamiliar with Docker, it's highly recommended that you read through Docker's overview documentation.

For more information about how to install Docker, see the download page for Docker Desktop: Community Edition.

Docker basics

There are a few concepts you should be familiar with. The Docker client has a command line interface program you use to manage images and containers. As previously stated, you should take the time to read through the Docker overview documentation.

Images

An image is an ordered collection of filesystem changes that form the basis of a container. The image doesn't have a state and is read-only. Much the time an image is based on another image, but with some customization. For example, when you create an new image for your application, you would base it on an existing image that already contains the .NET Core runtime.

Because containers are created from images, images have a set of run parameters (such as a starting executable) that run when the container starts.

Containers

A container is a runnable instance of an image. As you build your image, you deploy your application and dependencies. Then, multiple containers can be instantiated, each isolated from one another. Each container instance has its own filesystem, memory, and network interface.

Registries

Container registries are a collection of image repositories. You can base your images on a registry image. You can create containers directly from an image in a registry. The relationship between Docker containers, images, and registries is an important concept when architecting and building containerized applications or microservices. This approach greatly shortens the time between development and deployment.

Docker has a public registry hosted at the Docker Hub that you can use. .NET Core related images are listed at the Docker Hub.

The Microsoft Container Registry (MCR) is the official source of Microsoft-provided container images. The MCR is built on Azure CDN to provide globally-replicated images. However, the MCR does not have a public-facing website and the primary way to learn about Microsoft-provided container images is through the Microsoft Docker Hub pages.

Dockerfile

A **Dockerfile** is a file that defines a set of instructions that creates an image. Each instruction in the **Dockerfile** creates a layer in the image. For the most part, when you rebuild the image, only the layers that have changed are rebuilt. The **Dockerfile** can be distributed to others and allows them to recreate a new image in the same manner you created it. While this allows you to distribute the *instructions* on how to create the image, the main way to distribute your image is to publish it to a registry.

.NET Core images

Official .NET Core Docker images are published to the Microsoft Container Registry (MCR) and are discoverable at the Microsoft .NET Core Docker Hub repository. Each repository contains images for different combinations of the .NET (SDK or Runtime) and OS that you can use.

Microsoft provides images that are tailored for specific scenarios. For example, the ASP.NET Core repository provides images that are built for running ASP.NET Core apps in production.

Azure services

Various Azure services support containers. You create a Docker image for your application and deploy it to one of the following services:

- Azure Kubernetes Service (AKS)
  Scale and orchestrate Linux containers using Kubernetes.
- Azure App Service
  Deploy web apps or APIs using Linux containers in a PaaS environment.
- Azure Container Instances
  Host your container in the cloud without any higher-level management services.
- Azure Batch
  Run repetitive compute jobs using containers.
- Azure Service Fabric
  Lift, shift, and modernize .NET applications to microservices using Windows Server containers.
- Azure Container Registry
  Store and manage container images across all types of Azure deployments.

Next steps

- Learn how to containerize a .NET Core application.
- Learn how to containerize an ASP.NET Core application.
- Try the Learn ASP.NET Core Microservice tutorial.
- Learn about Container Tools in Visual Studio

Tutorial: Containerize a .NET Core app

10/17/2019 • 11 minutes to read • Edit Online

This tutorial teaches you how to build a Docker image that contains your .NET Core application. The image can be used to create containers for your local development environment, private cloud, or public cloud.

You'll learn to:

- Create and publish a simple .NET Core app
- Create and configure a Dockerfile for .NET Core
- Build a Docker image
- Create and run a Docker container

You'll understand the Docker container build and deploy tasks for a .NET Core application. The *Docker platform* uses the *Docker engine* to quickly build and package apps as *Docker images*. These images are written in the *Dockerfile* format to be deployed and run in a layered container.

Prerequisites

Install the following prerequisites:

- .NET Core 2.2 SDK
  If you have .NET Core installed, use the `dotnet --info` command to determine which SDK you're using.
- Docker Community Edition
- A temporary working folder for the *Dockerfile* and .NET Core example app. In this tutorial, the name `docker-working` is used as the working folder.

Use SDK version 2.2

If you're using an SDK that is newer, like 3.0, make sure that your app is forced to use the 2.2 SDK. Create a file named *global.json* in your working folder and paste in the following JSON code:

```
{
  "sdk": {
    "version": "2.2.100"
  }
}
```

Save this file. The presence of file will force .NET Core to use version 2.2 for any `dotnet` command called from this folder and below.

Create .NET Core app

You need a .NET Core app that the Docker container will run. Open your terminal, create a working folder if you haven't already, and enter it. In the working folder, run the following command to create a new project in a subdirectory named *app*:

```
dotnet new console -o app -n myapp
```

Your folder tree will look like the following:

```
docker-working
│   global.json
│
└───app
    │   myapp.csproj
    │   Program.cs
    │
    └───obj
            myapp.csproj.nuget.cache
            myapp.csproj.nuget.g.props
            myapp.csproj.nuget.g.targets
            project.assets.json
```

The `dotnet new` command creates a new folder named *app* and generates a "Hello World" app. Enter the *app* folder and run the command `dotnet run`. You'll see the following output:

```
> dotnet run
Hello World!
```

The default template creates an app that prints to the terminal and then exits. For this tutorial, you'll use an app that loops indefinitely. Open the *Program.cs* file in a text editor. It should currently look like the following code:

```
using System;

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

Replace the file with the following code that counts numbers every second:

```
using System;

namespace myapp
{
    class Program
    {
        static void Main(string[] args)
        {
            var counter = 0;
            var max = args.Length != 0 ? Convert.ToInt32(args[0]) : -1;
            while (max == -1 || counter < max)
            {
                counter++;
                Console.WriteLine($"Counter: {counter}");
                System.Threading.Tasks.Task.Delay(1000).Wait();
            }
        }
    }
}
```

Save the file and test the program again with `dotnet run`. Remember that this app runs indefinitely. Use the cancel command CTRL+C to stop it. You'll see the following output:

```
> dotnet run
Counter: 1
Counter: 2
Counter: 3
Counter: 4
^C
```

If you pass a number on the command line to the app, it will only count up to that amount and then exit. Try it with `dotnet run -- 5` to count to five.

> **NOTE**
>
> Any parameters after `--` are not passed to the `dotnet run` command and instead are passed to your application.

Publish .NET Core app

Before you add your .NET Core app to the Docker image, publish it. You want to make sure that the container runs the published version of the app when it's started.

From the working folder, enter the *app* folder with the example source code and run the following command:

```
dotnet publish -c Release
```

This command compiles your app to the *publish* folder. The path to the *publish* folder from the working folder should be `.\app\bin\Release\netcoreapp2.2\publish\`

Get a directory listing of the publish folder to verify that the *myapp.dll* was created. From the *app* folder, run one of the following commands:

```
> dir bin\Release\netcoreapp2.2\publish
 Directory of C:\docker-working\app\bin\Release\netcoreapp2.2\publish

04/05/2019  11:00 AM    <DIR>          .
04/05/2019  11:00 AM    <DIR>          ..
04/05/2019  11:00 AM               447 myapp.deps.json
04/05/2019  11:00 AM             4,608 myapp.dll
04/05/2019  11:00 AM               448 myapp.pdb
04/05/2019  11:00 AM               154 myapp.runtimeconfig.json
```

```
me@DESKTOP:/docker-working/app$ ls bin/Release/netcoreapp2.2/publish
myapp.deps.json  myapp.dll  myapp.pdb  myapp.runtimeconfig.json
```

Create the Dockerfile

The *Dockerfile* file is used by the `docker build` command to create a container image. This file is a plaintext file named *Dockerfile* that does not have an extension.

In your terminal, navigate up a directory to the working folder you created at the start. Create a file named *Dockerfile* in your working folder and open it in a text editor. Add the following command as the first line of the file:

```
FROM mcr.microsoft.com/dotnet/core/runtime:2.2
```

The `FROM` command tells Docker to pull down the image tagged **2.2** from the

mcr.microsoft.com/dotnet/core/runtime repository. Make sure that you pull the .NET Core runtime that matches the runtime targeted by your SDK. For example, the app created in the previous section used the .NET Core 2.2 SDK and created an app that targeted .NET Core 2.2. So the base image referred to in the *Dockerfile* is tagged with **2.2**.

Save the *Dockerfile* file. The directory structure of the working folder should look like the following. Some of the deeper-level files and folders have been cut to save space in the article:

```
docker-working
│   Dockerfile
│   global.json
│
└───app
    │   myapp.csproj
    │   Program.cs
    │
    ├───bin
    │   └───Release
    │       └───netcoreapp2.2
    │           └───publish
    │                   myapp.deps.json
    │                   myapp.dll
    │                   myapp.pdb
    │                   myapp.runtimeconfig.json
    │
    └───obj
```

From your terminal, run the following command:

```
docker build -t myimage -f Dockerfile .
```

Docker will process each line in the *Dockerfile*. The `.` in the `docker build` command tells Docker to use the current folder to find a *Dockerfile*. This command builds the image and creates a local repository named **myimage** that points to that image. After this command finishes, run `docker images` to see a list of images installed:

```
> docker images
REPOSITORY                              TAG                 IMAGE ID            CREATED             SIZE
mcr.microsoft.com/dotnet/core/runtime   2.2                 d51bb4452469        2 days ago          314MB
myimage                                 latest              d51bb4452469        2 days ago          314MB
```

Notice that the two images share the same **IMAGE ID** value. The value is the same between both images because the only command in the *Dockerfile* was to base the new image on an existing image. Let's add two commands to the *Dockerfile*. Each command creates a new image layer with the final command representing the image the **myimage** repository will point to.

```
COPY app/bin/Release/netcoreapp2.2/publish/ app/

ENTRYPOINT ["dotnet", "app/myapp.dll"]
```

The `COPY` command tells Docker to copy the specified folder on your computer to a folder in the container. In this example, the *publish* folder is copied to a folder named *app* in the container.

The next command, `ENTRYPOINT`, tells Docker to configure the container to run as an executable. When the container starts, the `ENTRYPOINT` command runs. When this command ends, the container will automatically stop.

From your terminal, run `docker build -t myimage -f Dockerfile .` and when that command finishes, run `docker images`.

```
> docker build -t myimage -f Dockerfile .
Sending build context to Docker daemon  819.7kB
Step 1/3 : FROM mcr.microsoft.com/dotnet/core/runtime:2.2
 ---> d51bb4452469
Step 2/3 : COPY app/bin/Release/netcoreapp2.2/publish/ app/
 ---> a1e98ac62017
Step 3/3 : ENTRYPOINT ["dotnet", "app/myapp.dll"]
 ---> Running in f34da5c18e7c
Removing intermediate container f34da5c18e7c
 ---> ddcc6646461b
Successfully built ddcc6646461b
Successfully tagged myimage:latest

> docker images
REPOSITORY                         TAG                 IMAGE ID            CREATED             SIZE
myimage                            latest              ddcc6646461b        10 seconds ago      314MB
mcr.microsoft.com/dotnet/core/runtime   2.2            d51bb4452469        2 days ago          314MB
```

Each command in the *Dockerfile* generated a layer and created an **IMAGE ID**. The final **IMAGE ID** (yours will be different) is **ddcc6646461b** and next you'll create a container based on this image.

Create a container

Now that you have an image that contains your app, you can create a container. You can create a container in two ways. First, create a new container that is stopped.

```
> docker create myimage
0e8f3c2ca32ce773712a5cca38750f41259a4e54e04bdf0946087e230ad7066c
```

The `docker create` command from above will create a container based on the **myimage** image. The output of that command shows you the **CONTAINER ID** (yours will be different) of the created container. To see a list of *all* containers, use the `docker ps -a` command:

```
> docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS        PORTS
NAMES
0e8f3c2ca32c        myimage             "dotnet app/myapp.dll"   4 seconds ago       Created
boring_matsumoto
```

Manage the container

Each container is assigned a random name that you can use to refer to that container instance. For example, the container that was created automatically chose the name **boring_matsumoto** (yours will be different) and that name can be used to start the container. You override the automatic name with a specific one by using the `docker create --name` parameter.

The following example uses the `docker start` command to start the container, and then uses the `docker ps` command to only show containers that are running:

```
> docker start boring_matsumoto
boring_matsumoto

> docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS         PORTS
NAMES
0e8f3c2ca32c        myimage             "dotnet app/myapp.dll"   7 minutes ago       Up 8 seconds
boring_matsumoto
```

Similarly, the `docker stop` command will stop the container. The following example uses the `docker stop` command to stop the container, and then uses the `docker ps` command to show that no containers are running:

```
> docker stop boring_matsumoto
boring_matsumoto

> docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS      PORTS   NAMES
```

Connect to a container

After a container is running, you can connect to it to see the output. Use the `docker start` and `docker attach` commands to start the container and peek at the output stream. In this example, the `CTRL + C` command is used to detach from the running container. This may actually end the process in the container, which will stop the container. The `--sig-proxy=false` parameter ensures that `CTRL + C` won't stop the process in the container.

After you detach from the container, reattach to verify that it's still running and counting.

```
> docker start boring_matsumoto
boring_matsumoto

> docker attach --sig-proxy=false boring_matsumoto
Counter: 7
Counter: 8
Counter: 9
^C

> docker attach --sig-proxy=false boring_matsumoto
Counter: 17
Counter: 18
Counter: 19
^C
```

Delete a container

For the purposes of this article you don't want containers just hanging around doing nothing. Delete the container you previously created. If the container is running, stop it.

```
> docker stop boring_matsumoto
```

The following example lists all containers. It then uses the `docker rm` command to delete the container, and then checks a second time for any running containers.

```
> docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS      PORTS   NAMES
0e8f3c2ca32c        myimage             "dotnet app/myapp.dll"   19 minutes ago      Exited
boring_matsumoto

> docker rm boring_matsumoto
boring_matsumoto

> docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS      PORTS    NAMES
```

Single run

Docker provides the `docker run` command to create and run the container as a single command. This command eliminates the need to run `docker create` and then `docker start`. You can also set this command to automatically delete the container when the container stops. For example, use `docker run -it --rm` to do two things, first, automatically use the current terminal to connect to the container, and then when the container finishes, remove it:

```
> docker run -it --rm myimage
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
^C
```

With `docker run -it`, the CTRL + C command will stop process that is running in the container, which in turn, stops the container. Since the `--rm` parameter was provided, the container is automatically deleted when the process is stopped. Verify that it does not exist:

```
> docker ps -a
CONTAINER ID        IMAGE               COMMAND                   CREATED             STATUS     PORTS   NAMES
```

Change the ENTRYPOINT

The `docker run` command also lets you modify the `ENTRYPOINT` command from the *Dockerfile* and run something else, but only for that container. For example, use the following command to run `bash` or `cmd.exe`. Edit the command as necessary.

Windows

In this example, `ENTRYPOINT` is changed to `cmd.exe`. CTRL+C is pressed to end the process and stop the container.

```
> docker run -it --rm --entrypoint "cmd.exe" myimage

Microsoft Windows [Version 10.0.17763.379]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\>dir
 Volume in drive C has no label.
 Volume Serial Number is 3005-1E84

 Directory of C:\

04/09/2019  08:46 AM    <DIR>          app
03/07/2019  10:25 AM             5,510 License.txt
04/02/2019  01:35 PM    <DIR>          Program Files
04/09/2019  01:06 PM    <DIR>          Users
04/02/2019  01:35 PM    <DIR>          Windows
               1 File(s)          5,510 bytes
               4 Dir(s)  21,246,517,248 bytes free

C:\>^C
```

Linux

In this example, `ENTRYPOINT` is changed to `bash`. The `quit` command is run which ends the process and stop the container.

```
root@user:~# docker run -it --rm --entrypoint "bash" myimage
root@8515e897c893:/# ls app
myapp.deps.json  myapp.dll  myapp.pdb  myapp.runtimeconfig.json
root@8515e897c893:/# exit
exit
```

Essential commands

Docker has many different commands that cover what you want to do with your container and images. These Docker commands are essential to managing your containers:

- docker build
- docker run
- docker ps
- docker stop
- docker rm
- docker rmi
- docker image

Clean up resources

During this tutorial you created containers and images. If you want, delete these resources. Use the following commands to

1. List all containers

   ```
   > docker ps -a
   ```

2. Stop containers that are running. The `CONTAINER_NAME` represents the name automatically assigned to the container.

   ```
   > docker stop CONTAINER_NAME
   ```

3. Delete the container

   ```
   > docker rm CONTAINER_NAME
   ```

Next, delete any images that you no longer want on your machine. Delete the image created by your *Dockerfile* and then delete the .NET Core image the *Dockerfile* was based on. You can use the **IMAGE ID** or the **REPOSITORY:TAG** formatted string.

```
docker rmi myimage:latest
docker rmi mcr.microsoft.com/dotnet/core/runtime:2.2
```

Use the `docker images` command to see a list of images installed.

NOTE

Image files can be large. Typically, you would remove temporary containers you created while testing and developing your app. You usually keep the base images with the runtime installed if you plan on building other images based on that runtime.

Next steps

- Try the ASP.NET Core Microservice Tutorial.
- Review the Azure services that support containers.
- Read about Dockerfile commands.
- Explore the Container Tools for Visual Studio

What diagnostic tools are available in .NET Core?

10/15/2019 • 2 minutes to read • Edit Online

Software doesn't always behave as you would expect, but .NET Core has tools and APIs that will help you diagnose these issues quickly and effectively.

This article helps you find the various tools you need.

Managed debuggers

Managed debuggers allow you to interact with your program. Pausing, incrementally executing, examining, and resuming gives you insight into the behavior of your code. A debugger is the first choice for diagnosing functional problems that can be easily reproduced.

Logging and tracing

Logging and tracing are related techniques. They refer to instrumenting code to create log files. The files record the details of what a program does. These details can be used to diagnose the most complex problems. When combined with time stamps, these techniques are also valuable in performance investigations.

Unit testing

Unit testing is a key component of continuous integration and deployment of high-quality software. Unit tests are designed to give you an early warning when you break something.

.NET Core dotnet diagnostic Global Tools

dotnet-counters

dotnet-counters is a performance monitoring tool for first-level health monitoring and performance investigation. It observes performance counter values published via the EventCounter API. For example, you can quickly monitor things like the CPU usage or the rate of exceptions being thrown in your .NET Core application.

dotnet-dump

The dotnet-dump tool is a way to collect and analyze Windows and Linux core dumps without a native debugger.

dotnet-trace

.NET Core includes what is called the `EventPipe` through which diagnostics data is exposed. The dotnet-trace tool allows you to consume interesting profiling data from your app that can help in scenarios where you need to root cause apps running slow.

.NET Core managed debuggers

9/12/2019 • 2 minutes to read • Edit Online

Debuggers allow programs to be paused or executed step-by-step. When paused, the current state of the process can be viewed. By stepping through key sections, you gain understanding of your code and why it produces the result that it does.

Microsoft provides debuggers for managed code in **Visual Studio** and **Visual Studio Code**.

Visual Studio managed debugger

Visual Studio is an integrated development environment with the most comprehensive debugger available. Visual Studio is an excellent choice for developers working on Windows.

- Tutorial - Debugging a .NET Core application on Windows with Visual Studio

While Visual Studio is a Windows application, it can still be used to debug Linux and macOS apps remotely.

- Debugging a .NET Core application on Linux/OSX with Visual Studio

Debugging ASP.NET Core apps require slightly different instructions.

- Debug ASP.NET Core apps in Visual Studio

Visual Studio Code managed debugger

Visual Studio Code is a lightweight cross-platform code editor. It uses the same .NET Core debugger implementation as Visual Studio, but with a simplified user interface.

- Tutorial - Debugging a .NET Core application with Visual Studio Code
- Debugging in Visual Studio Code

.NET Core logging and tracing

9/12/2019 • 3 minutes to read • Edit Online

Logging and tracing are really two names for the same technique. The simple technique has been used since the early days of computers. It simply involves instrumenting an application to write output to be consumed later.

Reasons to use logging and tracing

This simple technique is surprisingly powerful. It can be used in situations where a debugger fails:

- Issues occurring over long periods of time, can be difficult to debug with a traditional debugger. Logs allow for detailed post-mortem review spanning long periods of time. In contrast, debuggers are constrained to real-time analysis.
- Multi-threaded applications and distributed applications are often difficult to debug. Attaching a debugger tends to modify behaviors. Detailed logs can be analyzed as needed to understand complex systems.
- Issues in distributed applications may arise from a complex interaction between many components and it may not be reasonable to connect a debugger to every part of the system.
- Many services shouldn't be stalled. Attaching a debugger often causes timeout failures.
- Issues aren't always foreseen. Logging and tracing are designed for low overhead so that programs can always be recording in case an issue occurs.

.NET Core APIs

Print style APIs

The System.Console, System.Diagnostics.Trace, and System.Diagnostics.Debug classes each provide similar print style APIs convenient for logging.

The choice of which print style API to use is up to you. The key differences are:

- System.Console
  - Always enabled and always writes to the console.
  - Useful for information that your customer may need to see in the release.
  - Because it's the simplest approach, it's often used for ad-hoc temporary debugging. This debug code is often never checked in to source control.
- System.Diagnostics.Trace
  - Only enabled when `TRACE` is defined.
  - Writes to attached Listeners, by default the DefaultTraceListener.
  - Use this API when creating logs that will be enabled in most builds.
- System.Diagnostics.Debug
  - Only enabled when `DEBUG` is defined.
  - Writes to an attached debugger.
  - On `*nix` writes to stderr if `COMPlus_DebugWriteToStdErr` is set.
  - Use this API when creating logs that will be enabled only in debug builds.

Logging events

The following APIs are more event oriented. Rather than logging simple strings they log event objects.

- System.Diagnostics.Tracing.EventSource

  - EventSource is the primary root .NET Core tracing API.
  - Available in all .NET Standard versions.
  - Only allows tracing serializable objects.
  - Writes to the attached event listeners.
  - .NET Core provides listeners for:
    - .NET Core's EventPipe on all platforms
    - Event Tracing for Windows (ETW)
    - LTTng tracing framework for Linux
- System.Diagnostics.DiagnosticSource
  - Included in .NET Core and as a NuGet package for .NET Framework.
  - Allows in-process tracing of non-serializable objects.
  - Includes a bridge to allow selected fields of logged objects to be written to an EventSource.
- System.Diagnostics.Activity
  - Provides a definitive way to identify log messages resulting from a specific activity or transaction. This object can be used to correlate logs across different services.
- System.Diagnostics.EventLog
  - Windows only.
  - Writes messages to the Windows Event Log.
  - System administrators expect fatal application error messages to appear in the Windows Event Log.

ILogger and logging frameworks

The low-level APIs may not be the right choice for your logging needs. You may want to consider a logging framework.

The ILogger interface has been used to create a common logging interface where the loggers can be inserted through dependency injection.

For instance, to allow you to make the best choice for your application `ASP.NET` offers support for a selection of built-in and third-party frameworks:

- ASP.NET built in logging providers
- ASP.NET Third-party logging providers

Logging related references

- How to: Compile Conditionally with Trace and Debug
- How to: Add Trace Statements to Application Code
- ASP.NET Logging provides an overview of the logging techniques it supports.
- C# String Interpolation can simplify writing logging code.
- The Exception.Message property is useful for logging exceptions.
- The System.Diagnostics.StackTrace class can be useful to provide stack info in your logs.

Performance considerations

String formatting can take noticeable CPU processing time.

In performance critical applications, it's recommended that you:

- Avoid lots of logging when no one is listening. Avoid constructing costly logging messages by checking if logging is enabled first.
- Only log what's useful.
- Defer fancy formatting to the analysis stage.

dotnet-counters

10/15/2019 • 2 minutes to read • Edit Online

This article applies to: ✓ .NET Core 3.0 SDK and later versions

Install dotnet-counters

To install the latest release version of the `dotnet-counters` NuGet package, use the dotnet tool install command:

```
dotnet tool install --global dotnet-counters
```

Synopsis

```
dotnet-counters [-h|--help] [--version] <command>
```

Description

`dotnet-counters` is a performance monitoring tool for ad-hoc health monitoring and first-level performance investigation. It can observe performance counter values that are published via the EventCounter API. For example, you can quickly monitor things like the CPU usage or the rate of exceptions being thrown in your .NET Core application to see if there's anything suspicious before diving into more serious performance investigation using `PerfView` or `dotnet-trace`.

Options

- `--version`

  Displays the version of the dotnet-counters utility.

- `-h|--help`

  Shows command-line help.

Commands

COMMAND
dotnet-counters list
dotnet-counters monitor

dotnet-counters list

Displays a list of counter names and descriptions, grouped by provider.

Synopsis

```
dotnet-counters list [-h|--help]
```

Example

```
> dotnet-counters list

    Showing well-known counters only. Specific processes may support additional counters.
    System.Runtime
        cpu-usage                    Amount of time the process has utilized the CPU (ms)
        working-set                  Amount of working set used by the process (MB)
        gc-heap-size                 Total heap size reported by the GC (MB)
        gen-0-gc-count               Number of Gen 0 GCs / sec
        gen-1-gc-count               Number of Gen 1 GCs / sec
        gen-2-gc-count               Number of Gen 2 GCs / sec
        exception-count              Number of Exceptions / sec
```

dotnet-counters monitor

Displays periodically refreshing values of selected counters.

Synopsis

```
dotnet-counters monitor [-h|--help] [-p|--process-id] [--refreshInterval] [counter_list]
```

Options

- `-p|--process-id <PID>`

  The ID of the process to be monitored.

- `--refresh-interval <SECONDS>`

  The number of seconds to delay between updating the displayed counters

- `counter_list <COUNTERS>`

  A space separated list of counters. Counters can be specified `provider_name[:counter_name]`. If the `provider_name` is used without a qualifying `counter_name`, then all counters are shown. To discover provider and counter names, use the dotnet-counters list command.

Examples

- Monitor all counters from `System.Runtime` at a refresh interval of 3 seconds:

  ```
  > dotnet-counters monitor --process-id 1902  --refresh-interval 3 System.Runtime

  Press p to pause, r to resume, q to quit.
    System.Runtime:
      CPU Usage (%)                                 24
      Working Set (MB)                            1982
      GC Heap Size (MB)                            811
      Gen 0 GC / second                             20
      Gen 1 GC / second                              4
      Gen 2 GC / second                              1
      Number of Exceptions / sec                     4
  ```

- Monitor just CPU usage and GC heap size from `System.Runtime`:

```
> dotnet-counters monitor --process-id 1902 System.Runtime[cpu-usage,gc-heap-size]

Press p to pause, r to resume, q to quit.
  System.Runtime:
    CPU Usage (%)                                  24
    GC Heap Size (MB)                             811
```

- Monitor `EventCounter` values from user-defined `EventSource`. For more information, see Tutorial: How to measure performance for very frequent events using EventCounters.

```
> dotnet-counters monitor --process-id 1902 Samples-EventCounterDemos-Minimal

Press p to pause, r to resume, q to quit.
    request                                       100
```

Dump collection and analysis utility (`dotnet-dump`)

11/12/2019 • 5 minutes to read • Edit Online

This article applies to: ✓ .NET Core 3.0 SDK and later versions

NOTE
`dotnet-dump` isn't supported on macOS.

Installing `dotnet-dump`

To install the latest release version of the `dotnet-dump` NuGet package, use the dotnet tool install command:

```
dotnet tool install -g dotnet-dump
```

Synopsis

```
dotnet-dump [-h|--help] [--version] <command>
```

Description

The `dotnet-dump` global tool is a way to collect and analyze Windows and Linux dumps without any native debugger involved like `lldb` on Linux. This tool is important on platforms like Alpine Linux where a fully working `lldb` isn't available. The `dotnet-dump` tool allows you to run SOS commands to analyze crashes and the garbage collector (GC), but it isn't a native debugger so things like displaying native stack frames aren't supported.

Options

- `--version`

  Displays the version of the dotnet-counters utility.

- `-h|--help`

  Shows command-line help.

Commands

COMMAND
dotnet-dump collect
dotnet-dump analyze

dotnet-dump collect

Captures a dump from a process.

Synopsis

```
dotnet-dump collect [-h|--help] [-p|--process-id] [--type] [-o|--output] [--diag]
```

Options

- `-h|--help`

  Shows command-line help.

- `-p|--process-id <PID>`

  Specifies the process ID number to collect a memory dump from.

- `--type <Heap|Mini>`

  Specifies the dump type, which determines the kinds of information that are collected from the process. There are two types:

  - `Heap` - A large and relatively comprehensive dump containing module lists, thread lists, all stacks, exception information, handle information, and all memory except for mapped images.
  - `Mini` - A small dump containing module lists, thread lists, exception information, and all stacks.

  If not specified, `Heap` is the default.

- `-o|--output <output_dump_path>`

  The full path and file name where the collected dump should be written.

  If not specified:

  - Defaults to *.\dump_YYYYMMDD_HHMMSS.dmp* on Windows.
  - Defaults to *./core_YYYYMMDD_HHMMSS* on Linux.

  YYYYMMDD is Year/Month/Day and HHMMSS is Hour/Minute/Second.

- `--diag`

  Enables dump collection diagnostic logging.

dotnet-dump analyze

Starts an interactive shell to explore a dump. The shell accepts various SOS commands.

Synopsis

```
dotnet-dump analyze <dump_path> [-h|--help] [-c|--command]
```

Arguments

- `<dump_path>`

  Specifies the path to the dump file to analyze.

Options

- `-c|--command <debug_command>`

  Specifies the command to run in the shell on start.

Analyze SOS commands

COMMAND	FUNCTION
`soshelp`	Displays all available commands
`soshelp|help <command>`	Displays the specified command.
`exit|quit`	Exits interactive mode.
`clrstack <arguments>`	Provides a stack trace of managed code only.
`clrthreads <arguments>`	Lists the managed threads running.
`dumpasync <arguments>`	Displays information about async state machines on the garbage-collected heap.
`dumpassembly <arguments>`	Displays details about an assembly.
`dumpclass <arguments>`	Displays information about a EE class structure at the specified address.
`dumpdelegate <arguments>`	Displays information about a delegate.
`dumpdomain <arguments>`	Displays information all the AppDomains and all assemblies within the domains.
`dumpheap <arguments>`	Displays info about the garbage-collected heap and collection statistics about objects.
`dumpil <arguments>`	Displays the Microsoft intermediate language (MSIL) that is associated with a managed method.
`dumplog <arguments>`	Writes the contents of an in-memory stress log to the specified file.
`dumpmd <arguments>`	Displays information about a MethodDesc structure at the specified address.
`dumpmodule <arguments>`	Displays information about a EE module structure at the specified address.
`dumpmt <arguments>`	Displays information about a method table at the specified address.
`dumpobj <arguments>`	Displays info about an object at the specified address.
`dso|dumpstackobjects <arguments>`	Displays all managed objects found within the bounds of the current stack.
`eeheap <arguments>`	Displays info about process memory consumed by internal runtime data structures.
`finalizequeue <arguments>`	Displays all objects registered for finalization.

COMMAND	FUNCTION
`gcroot <arguments>`	Displays info about references (or roots) to an object at the specified address.
`gcwhere <arguments>`	Displays the location in the GC heap of the argument passed in.
`ip2md <arguments>`	Displays the MethodDesc structure at the specified address in JIT code.
`histclear <arguments>`	Releases any resources used by the family of `hist*` commands.
`histinit <arguments>`	Initializes the SOS structures from the stress log saved in the debuggee.
`histobj <arguments>`	Displays the garbage collection stress log relocations related to `<arguments>`.
`histobjfind <arguments>`	Displays all the log entries that reference an object at the specified address.
`histroot <arguments>`	Displays information related to both promotions and relocations of the specified root.
`lm|modules`	Displays the native modules in the process.
`name2ee <arguments>`	Displays the MethodTable structure and EEClass structure for the `<argument>`.
`pe|printexception <arguments>`	Displays any object derived from the Exception class at the address `<argument>`.
`setsymbolserver <arguments>`	Enables the symbol server support
`syncblk <arguments>`	Displays the SyncBlock holder info.
`threads|setthread <threadid>`	Sets or displays the current thread ID for the SOS commands.

Using `dotnet-dump`

The first step is to collect a dump. This step can be skipped if a core dump has already been generated. The operating system or the .NET Core runtime's built-in dump generation feature can each create core dumps.

```
$ dotnet-dump collect --process-id 1902
Writing minidump to file ./core_20190226_135837
Written 98983936 bytes (24166 pages) to core file
Complete
```

Now analyze the core dump with the `analyze` command:

```
$ dotnet-dump analyze ./core_20190226_135850
Loading core dump: ./core_20190226_135850
Ready to process analysis commands. Type 'help' to list available commands or 'help [command]' to get detailed
help on a command.
Type 'quit' or 'exit' to exit the session.
>
```

This action brings up an interactive session that accepts commands like:

```
> clrstack
OS Thread Id: 0x573d (0)
    Child SP               IP Call Site
00007FFD28B42C58 00007fb22c1a8ed9 [HelperMethodFrame_PROTECTOBJ: 00007ffd28b42c58]
System.RuntimeMethodHandle.InvokeMethod(System.Object, System.Object[], System.Signature, Boolean, Boolean)
00007FFD28B42DD0 00007FB1B1334F67 System.Reflection.RuntimeMethodInfo.Invoke(System.Object,
System.Reflection.BindingFlags, System.Reflection.Binder, System.Object[], System.Globalization.CultureInfo)
[/root/coreclr/src/mscorlib/src/System/Reflection/RuntimeMethodInfo.cs @ 472]
00007FFD28B42E20 00007FB1B18D33ED SymbolTestApp.Program.Foo4(System.String)
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 54]
00007FFD28B42ED0 00007FB1B18D2FC4 SymbolTestApp.Program.Foo2(Int32, System.String)
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 29]
00007FFD28B42F00 00007FB1B18D2F5A SymbolTestApp.Program.Foo1(Int32, System.String)
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 24]
00007FFD28B42F30 00007FB1B18D168E SymbolTestApp.Program.Main(System.String[])
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 19]
00007FFD28B43210 00007fb22aa9cedf [GCFrame: 00007ffd28b43210]
00007FFD28B43610 00007fb22aa9cedf [GCFrame: 00007ffd28b43610]
```

To see an unhandled exception that killed your app:

```
> pe -lines
Exception object: 00007fb18c038590
Exception type:   System.Reflection.TargetInvocationException
Message:          Exception has been thrown by the target of an invocation.
InnerException:   System.Exception, Use !PrintException 00007FB18C038368 to see more.
StackTrace (generated):
SP               IP               Function
00007FFD28B42DD0 0000000000000000
System.Private.CoreLib.dll!System.RuntimeMethodHandle.InvokeMethod(System.Object, System.Object[],
System.Signature, Boolean, Boolean)
00007FFD28B42DD0 00007FB1B1334F67
System.Private.CoreLib.dll!System.Reflection.RuntimeMethodInfo.Invoke(System.Object,
System.Reflection.BindingFlags, System.Reflection.Binder, System.Object[],
System.Globalization.CultureInfo)+0xa7 [/root/coreclr/src/mscorlib/src/System/Reflection/RuntimeMethodInfo.cs
@ 472]
00007FFD28B42E20 00007FB1B18D33ED SymbolTestApp.dll!SymbolTestApp.Program.Foo4(System.String)+0x15d
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 54]
00007FFD28B42ED0 00007FB1B18D2FC4 SymbolTestApp.dll!SymbolTestApp.Program.Foo2(Int32, System.String)+0x34
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 29]
00007FFD28B42F00 00007FB1B18D2F5A SymbolTestApp.dll!SymbolTestApp.Program.Foo1(Int32, System.String)+0x3a
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 24]
00007FFD28B42F30 00007FB1B18D168E SymbolTestApp.dll!SymbolTestApp.Program.Main(System.String[])+0x6e
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 19]

StackTraceString: <none>
HResult: 80131604
```

Special instructions for Docker

If you're running under Docker, dump collection requires `SYS_PTRACE` capabilities (`--cap-add=SYS_PTRACE` or `--privileged`).

On Microsoft .NET Core SDK Linux Docker images, some `dotnet-dump` commands can throw the following exception:

> Unhandled exception: System.DllNotFoundException: Unable to load shared library 'libdl.so' or one of its dependencies' exception.

To work around this problem, install the "libc6-dev" package.

Trace for performance analysis utility (`dotnet-trace`)

10/15/2019 • 4 minutes to read • Edit Online

This article applies to: .NET Core 3.0 SDK and later versions

Installing `dotnet-trace`

To install the latest release version of the `dotnet-trace` NuGet package, use the dotnet tool install command:

```
dotnet tool install --global dotnet-trace
```

Synopsis

```
dotnet-trace [-h, --help] [--version] <command>
```

Description

The `dotnet-trace` tool is a cross-platform CLI global tool that enables the collection of .NET Core traces of a running process without any native profiler involved. It's built around the cross-platform `EventPipe` technology of the .NET Core runtime. `dotnet-trace` delivers the same experience on Windows, Linux, or macOS.

Options

- `--version`

Display the version of the dotnet-counters utility.

- `-h|--help`

Show command-line help.

Commands

COMMAND
dotnet-trace collect
dotnet-trace convert
dotnet-trace list-processes
dotnet-trace list-profiles

dotnet-trace collect

Collects a diagnostic trace from a running process.

Synopsis

```
dotnet-trace collect [-h|--help] [-p|--process-id] [--buffersize <size>] [-o|--output]
    [--providers] [--profile <profile-name>] [--format]
```

Options

- `-p|--process-id <PID>`

  The process to collect the trace from.

- `--buffersize <size>`

  Sets the size of the in-memory circular buffer in megabytes. Default 256 MB.

- `-o|--output <trace-file-path>`

  The output path for the collected trace data. If not specified it defaults to `trace.nettrace`.

- `--providers <list-of-comma-separated-providers>`

  A comma-separated list of `EventPipe` providers to be enabled. These providers supplement any providers implied by `--profile <profile-name>`. If there's any inconsistency for a particular provider, the configuration here takes precedence over the implicit configuration from the profile.

  This list of providers is in the form:

  - `Provider[,Provider]`
  - `Provider` is in the form: `KnownProviderName[:Flags[:Level][:KeyValueArgs]]`.
  - `KeyValueArgs` is in the form: `[key1=value1][;key2=value2]`.

- `--profile <profile-name>`

  A named pre-defined set of provider configurations that allows common tracing scenarios to be specified succinctly.

- `--format <NetTrace|Speedscope>`

  Sets the output format for the trace file conversion.

dotnet-trace convert

Converts `nettrace` traces to alternate formats for use with alternate trace analysis tools.

Synopsis

```
dotnet-trace convert [<input-filename>] [-h|--help] [--format] [-o|--output]
```

Arguments

- `<input-filename>`

  Input trace file to be converted. Defaults to *trace.nettrace*.

Options

- `--format <NetTrace|Speedscope>`

  Sets the output format for the trace file conversion.

- `-o|--output <output-filename>`

  Output filename. Extension of target format will be added.

dotnet-trace list-processes

Lists dotnet processes that can be traced.

Synopsis

```
dotnet-trace list-processes [-h|--help]
```

dotnet-trace list-profiles

Lists pre-built tracing profiles with a description of what providers and filters are in each profile.

Synopsis

```
dotnet-trace list-profiles [-h|--help]
```

Collect a trace with `dotnet-trace`

- To collect traces using `dotnet-trace`, you'll need to first, find out the process identifier (PID) of the .NET Core application to collect traces from.
  - On Windows, there are options such as using the task manager or the `tasklist` command.
  - On Linux, the trivial option could be using `ps` command.

You may also use the dotnet-trace list-processes command to find out what .NET Core processes are running, along with their PIDs.

- Then, run the following command:

```
> dotnet-trace collect --process-id <PID>

Press <Enter> to exit...
Connecting to process: <Full-Path-To-Process-Being-Profiled>/dotnet.exe
Collecting to file: <Full-Path-To-Trace>/trace.nettrace
  Session Id: <SessionId>
  Recording trace 721.025 (KB)
```

- Finally, stop collection by pressing the `<Enter>` key, and `dotnet-trace` will finish logging events to `trace.nettrace` file.

Viewing the trace captured from `dotnet-trace`

On Windows, `.nettrace` files can be viewed on PerfView for analysis, just like traces collected with ETW or LTTng. For traces collected on Linux, you can move the trace to a Windows machine to be viewed on PerfView.

You may also view the trace on a Linux machine by changing the output format of `dotnet-trace` to `speedscope`. You can change the output file format using the `-f|--format` option - `-f speedscope` will make `dotnet-trace` to produce a `speedscope` file. You can currently choose between `nettrace` (the default option) and `speedscope`. `Speedscope` files can be opened at https://www.speedscope.app.

NOTE

The .NET Core runtime generates traces in the `nettrace` format, and they're converted to speedscope (if specified) after the trace is completed. Since some conversions may result in loss of data, the original `nettrace` file is preserved next to the converted file.

Using `dotnet-trace` to collect counter values over time

If you're trying to use `EventCounter` for basic health monitoring in performance-sensitive settings like production environments and you want to collect traces instead of watching them in real time, you can do that with `dotnet-trace` as well.

For example, if you want to collect runtime performance counter values, you can use the following command:

```
dotnet-trace collect --process-id <PID> --providers System.Runtime:0:1:EventCounterIntervalSec=1
```

This command tells the runtime counters to be reported once every second for lightweight health monitoring. Replacing `EventCounterIntervalSec=1` with a higher value (for example, 60) allows you to collect a smaller trace with less granularity in the counter data.

If you want to disable runtime events to reduce the overhead (and trace size) even further, you can use the following command to disable runtime events and managed stack profiler.

```
dotnet-trace collect --process-id <PID> --providers System.Runtime:0:1:EventCounterIntervalSec=1,Microsoft-
Windows-DotNETRuntime:0:1,Microsoft-DotNETCore-SampleProfiler:0:1
```

.NET Providers

The .NET Core runtime supports the following .NET providers. .NET Core uses the same keywords to enable both `Event Tracing for Windows (ETW)` and `EventPipe` traces.

PROVIDER NAME	INFORMATION
`Microsoft-Windows-DotNETRuntime`	The Runtime Provider CLR Runtime Keywords
`Microsoft-Windows-DotNETRuntimeRundown`	The Rundown Provider CLR Rundown Keywords
`Microsoft-DotNETCore-SampleProfiler`	Enables the sample profiler.

Unit testing in .NET Core and .NET Standard

10/17/2019 • 2 minutes to read • Edit Online

.NET Core makes it easy to create unit tests. This article introduces unit tests and illustrates how they differ from other kinds of tests. The linked resources near the bottom of the page show you how to add a test project to your solution. After you set up your test project, you will be able to run your unit tests using the command line or Visual Studio.

If you're testing an **ASP.NET Core** project, see Integration tests in ASP.NET Core.

.NET Core 2.0 and later supports .NET Standard 2.0, and we will use its libraries to demonstrate unit tests.

You are able to use built-in .NET Core 2.0 and later unit test project templates for C#, F# and Visual Basic as a starting point for your personal project.

What are unit tests?

Having automated tests is a great way to ensure a software application does what its authors intend it to do. There are multiple types of tests for software applications. These include integration tests, web tests, load tests, and others. **Unit tests** test individual software components and methods. Unit tests should only test code within the developer's control. They should not test infrastructure concerns. Infrastructure concerns include databases, file systems, and network resources.

Also, keep in mind there are best practices for writing tests. For example, Test Driven Development (TDD) is when a unit test is written before the code it is meant to check. TDD is like creating an outline for a book before we write it. It is meant to help developers write simpler, more readable, and efficient code.

NOTE

The ASP.NET team follows these conventions to help developers come up with good names for test classes and methods.

Try not to introduce dependencies on infrastructure when writing unit tests. These make the tests slow and brittle, and should be reserved for integration tests. You can avoid these dependencies in your application by following the Explicit Dependencies Principle and using Dependency Injection. You can also keep your unit tests in a separate project from your integration tests. This ensures your unit test project doesn't have references to or dependencies on infrastructure packages.

Next steps

More information on unit testing in .NET Core projects:

.NET Core unit test projects are supported for:

- C#
- F#
- Visual Basic

You can also choose between:

- xUnit
- NUnit
- MSTest

You can learn more in the following walkthroughs:

- Create unit tests using *xUnit* and *C#* with the .NET Core CLI.
- Create unit tests using *NUnit* and *C#* with the .NET Core CLI.
- Create unit tests using *MSTest* and *C#* with the .NET Core CLI.
- Create unit tests using *xUnit* and *F#* with the .NET Core CLI.
- Create unit tests using *NUnit* and *F#* with the .NET Core CLI.
- Create unit tests using *MSTest* and *F#* with the .NET Core CLI.
- Create unit tests using *xUnit* and *Visual Basic* with the .NET Core CLI.
- Create unit tests using *NUnit* and *Visual Basic* with the .NET Core CLI.
- Create unit tests using *MSTest* and *Visual Basic* with the .NET Core CLI.

You can learn more in the following articles:

- Visual Studio Enterprise offers great testing tools for .NET Core. Check out Live Unit Testing or code coverage to learn more.
- For more information on how to run selective unit tests, see Running selective unit tests, or including and excluding tests with Visual Studio.
- How to use xUnit with .NET Core and Visual Studio.

Unit testing best practices with .NET Core and .NET Standard

9/12/2019 • 14 minutes to read • Edit Online

There are numerous benefits to writing unit tests; they help with regression, provide documentation, and facilitate good design. However, hard to read and brittle unit tests can wreak havoc on your code base. This article describes some best practices regarding unit test design for your .NET Core and .NET Standard projects.

In this guide, you'll learn some best practices when writing unit tests to keep your tests resilient and easy to understand.

By John Reese with special thanks to Roy Osherove

Why unit test?

Less time performing functional tests

Functional tests are expensive. They typically involve opening up the application and performing a series of steps that you (or someone else), must follow in order to validate the expected behavior. These steps may not always be known to the tester, which means they will have to reach out to someone more knowledgeable in the area in order to carry out the test. Testing itself could take seconds for trivial changes, or minutes for larger changes. Lastly, this process must be repeated for every change that you make in the system.

Unit tests, on the other hand, take milliseconds, can be run at the press of a button and do not necessarily require any knowledge of the system at large. Whether or not the test passes or fails is up to the test runner, not the individual.

Protection against regression

Regression defects are defects that are introduced when a change is made to the application. It is common for testers to not only test their new feature but also features that existed beforehand in order to verify that previously implemented features still function as expected.

With unit testing, it's possible to rerun your entire suite of tests after every build or even after you change a line of code. Giving you confidence that your new code does not break existing functionality.

Executable documentation

It may not always be obvious what a particular method does or how it behaves given a certain input. You may ask yourself: How does this method behave if I pass it a blank string? Null?

When you have a suite of well-named unit tests, each test should be able to clearly explain the expected output for a given input. In addition, it should be able to verify that it actually works.

Less coupled code

When code is tightly coupled, it can be difficult to unit test. Without creating unit tests for the code that you're writing, coupling may be less apparent.

Writing tests for your code will naturally decouple your code, because it would be more difficult to test otherwise.

Characteristics of a good unit test

- **Fast**. It is not uncommon for mature projects to have thousands of unit tests. Unit tests should take very little time to run. Milliseconds.

- **Isolated**. Unit tests are standalone, can be run in isolation, and have no dependencies on any outside factors such as a file system or database.
- **Repeatable**. Running a unit test should be consistent with its results, that is, it always returns the same result if you do not change anything in between runs.
- **Self-Checking**. The test should be able to automatically detect if it passed or failed without any human interaction.
- **Timely**. A unit test should not take a disproportionately long time to write compared to the code being tested. If you find testing the code taking a large amount of time compared to writing the code, consider a design that is more testable.

Let's speak the same language

The term *mock* is unfortunately very misused when talking about testing. The following defines the most common types of *fakes* when writing unit tests:

Fake - A fake is a generic term which can be used to describe either a stub or a mock object. Whether it is a stub or a mock depends on the context in which it's used. So in other words, a fake can be a stub or a mock.

Mock - A mock object is a fake object in the system that decides whether or not a unit test has passed or failed. A mock starts out as a Fake until it is asserted against.

Stub - A stub is a controllable replacement for an existing dependency (or collaborator) in the system. By using a stub, you can test your code without dealing with the dependency directly. By default, a fake starts out as a stub.

Consider the following code snippet:

```
var mockOrder = new MockOrder();
var purchase = new Purchase(mockOrder);

purchase.ValidateOrders();

Assert.True(purchase.CanBeShipped);
```

This would be an example of stub being referred to as a mock. In this case, it is a stub. You're just passing in the Order as a means to be able to instantiate `Purchase` (the system under test). The name `MockOrder` is also very misleading because again, the order is not a mock.

A better approach would be

```
var stubOrder = new FakeOrder();
var purchase = new Purchase(stubOrder);

purchase.ValidateOrders();

Assert.True(purchase.CanBeShipped);
```

By renaming the class to `FakeOrder`, you've made the class a lot more generic, the class can be used as a mock or a stub. Whichever is better for the test case. In the above example, `FakeOrder` is used as a stub. You're not using the `FakeOrder` in any shape or form during the assert. `FakeOrder` was just passed into the `Purchase` class to satisfy the requirements of the constructor.

To use it as a Mock, you could do something like this

```
var mockOrder = new FakeOrder();
var purchase = new Purchase(mockOrder);

purchase.ValidateOrders();

Assert.True(mockOrder.Validated);
```

In this case, you are checking a property on the Fake (asserting against it), so in the above code snippet, the `mockOrder` is a Mock.

IMPORTANT

It's important to get this terminology correct. If you call your stubs "mocks", other developers are going to make false assumptions about your intent.

The main thing to remember about mocks versus stubs is that mocks are just like stubs, but you assert against the mock object, whereas you do not assert against a stub.

Best practices

Naming your tests

The name of your test should consist of three parts:

- The name of the method being tested.
- The scenario under which it's being tested.
- The expected behavior when the scenario is invoked.

Why?

- Naming standards are important because they explicitly express the intent of the test.

Tests are more than just making sure your code works, they also provide documentation. Just by looking at the suite of unit tests, you should be able to infer the behavior of your code without even looking at the code itself. Additionally, when tests fail, you can see exactly which scenarios do not meet your expectations.

Bad:

```
[Fact]
public void Test_Single()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("0");

    Assert.Equal(0, actual);
}
```

Better:

```
[Fact]
public void Add_SingleNumber_ReturnsSameNumber()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("0");

    Assert.Equal(0, actual);
}
```

Arranging your tests

Arrange, Act, Assert is a common pattern when unit testing. As the name implies, it consists of three main actions:

- *Arrange* your objects, creating and setting them up as necessary.
- *Act* on an object.
- *Assert* that something is as expected.

Why?

- Clearly separates what is being tested from the *arrange* and *assert* steps.
- Less chance to intermix assertions with "Act" code.

Readability is one of the most important aspects when writing a test. Separating each of these actions within the test clearly highlight the dependencies required to call your code, how your code is being called, and what you are trying to assert. While it may be possible to combine some steps and reduce the size of your test, the primary goal is to make the test as readable as possible.

Bad:

```
[Fact]
public void Add_EmptyString_ReturnsZero()
{
    // Arrange
    var stringCalculator = new StringCalculator();

    // Assert
    Assert.Equal(0, stringCalculator.Add(""));
}
```

Better:

```
[Fact]
public void Add_EmptyString_ReturnsZero()
{
    // Arrange
    var stringCalculator = new StringCalculator();

    // Act
    var actual = stringCalculator.Add("");

    // Assert
    Assert.Equal(0, actual);
}
```

Write minimally passing tests

The input to be used in a unit test should be the simplest possible in order to verify the behavior that you are currently testing.

Why?

- Tests become more resilient to future changes in the codebase.
- Closer to testing behavior over implementation.

Tests that include more information than required to pass the test have a higher chance of introducing errors into the test and can make the intent of the test less clear. When writing tests you want to focus on the behavior. Setting extra properties on models or using non-zero values when not required, only detracts from what you are trying to prove.

Bad:

```
[Fact]
public void Add_SingleNumber_ReturnsSameNumber()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("42");

    Assert.Equal(42, actual);
}
```

Better:

```
[Fact]
public void Add_SingleNumber_ReturnsSameNumber()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("0");

    Assert.Equal(0, actual);
}
```

Avoid magic strings

Naming variables in unit tests is as important, if not more important, than naming variables in production code. Unit tests should not contain magic strings.

Why?

- Prevents the need for the reader of the test to inspect the production code in order to figure out what makes the value special.
- Explicitly shows what you're trying to *prove* rather than trying to *accomplish*.

Magic strings can cause confusion to the reader of your tests. If a string looks out of the ordinary, they may wonder why a certain value was chosen for a parameter or return value. This may lead them to take a closer look at the implementation details, rather than focus on the test.

TIP

When writing tests, you should aim to express as much intent as possible. In the case of magic strings, a good approach is to assign these values to constants.

Bad:

```
[Fact]
public void Add_BigNumber_ThrowsException()
{
    var stringCalculator = new StringCalculator();

    Action actual = () => stringCalculator.Add("1001");

    Assert.Throws<OverflowException>(actual);
}
```

Better:

```
[Fact]
void Add_MaximumSumResult_ThrowsOverflowException()
{
    var stringCalculator = new StringCalculator();
    const string MAXIMUM_RESULT = "1001";

    Action actual = () => stringCalculator.Add(MAXIMUM_RESULT);

    Assert.Throws<OverflowException>(actual);
}
```

Avoid logic in tests

When writing your unit tests avoid manual string concatenation and logical conditions such as `if`, `while`, `for`, `switch`, etc.

Why?

- Less chance to introduce a bug inside of your tests.
- Focus on the end result, rather than implementation details.

When you introduce logic into your test suite, the chance of introducing a bug into it increases dramatically. The last place that you want to find a bug is within your test suite. You should have a high level of confidence that your tests work, otherwise, you will not trust them. Tests that you do not trust, do not provide any value. When a test fails, you want to have a sense that something is actually wrong with your code and that it cannot be ignored.

> **TIP**
>
> If logic in your test seems unavoidable, consider splitting the test up into two or more different tests.

Bad:

```
[Fact]
public void Add_MultipleNumbers_ReturnsCorrectResults()
{
    var stringCalculator = new StringCalculator();
    var expected = 0;
    var testCases = new[]
    {
        "0,0,0",
        "0,1,2",
        "1,2,3"
    };

    foreach (var test in testCases)
    {
        Assert.Equal(expected, stringCalculator.Add(test));
        expected += 3;
    }

}
```

Better:

```
[Theory]
[InlineData("0,0,0", 0)]
[InlineData("0,1,2", 3)]
[InlineData("1,2,3", 6)]
public void Add_MultipleNumbers_ReturnsSumOfNumbers(string input, int expected)
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add(input);

    Assert.Equal(expected, actual);
}
```

Prefer helper methods to setup and teardown

If you require a similar object or state for your tests, prefer a helper method than leveraging Setup and Teardown attributes if they exist.

Why?

- Less confusion when reading the tests since all of the code is visible from within each test.
- Less chance of setting up too much or too little for the given test.
- Less chance of sharing state between tests which creates unwanted dependencies between them.

In unit testing frameworks, `Setup` is called before each and every unit test within your test suite. While some may see this as a useful tool, it generally ends up leading to bloated and hard to read tests. Each test will generally have different requirements in order to get the test up and running. Unfortunately, `Setup` forces you to use the exact same requirements for each test.

NOTE

xUnit has removed both SetUp and TearDown as of version 2.x

Bad:

```
private readonly StringCalculator stringCalculator;
public StringCalculatorTests()
{
    stringCalculator = new StringCalculator();
}
```

```
// more tests...
```

```
[Fact]
public void Add_TwoNumbers_ReturnsSumOfNumbers()
{
    var result = stringCalculator.Add("0,1");

    Assert.Equal(1, result);
}
```

Better:

```
[Fact]
public void Add_TwoNumbers_ReturnsSumOfNumbers()
{
    var stringCalculator = CreateDefaultStringCalculator();

    var actual = stringCalculator.Add("0,1");

    Assert.Equal(1, actual);
}
```

```
// more tests...
```

```
private StringCalculator CreateDefaultStringCalculator()
{
    return new StringCalculator();
}
```

Avoid multiple asserts

When writing your tests, try to only include one Assert per test. Common approaches to using only one assert include:

- Create a separate test for each assert.
- Use parameterized tests.

Why?

- If one Assert fails, the subsequent Asserts will not be evaluated.
- Ensures you are not asserting multiple cases in your tests.
- Gives you the entire picture as to why your tests are failing.

When introducing multiple asserts into a test case, it is not guaranteed that all of the asserts will be executed. In most unit testing frameworks, once an assertion fails in a unit test, the proceeding tests are automatically considered to be failing. This can be confusing as functionality that is actually working, will be shown as failing.

> **NOTE**
>
> A common exception to this rule is when asserting against an object. In this case, it is generally acceptable to have multiple asserts against each property to ensure the object is in the state that you expect it to be in.

Bad:

```
[Fact]
public void Add_EdgeCases_ThrowsArgumentExceptions()
{
    Assert.Throws<ArgumentException>(() => stringCalculator.Add(null));
    Assert.Throws<ArgumentException>(() => stringCalculator.Add("a"));
}
```

Better:

```
[Theory]
[InlineData(null)]
[InlineData("a")]
public void Add_InputNullOrAlphabetic_ThrowsArgumentException(string input)
{
    var stringCalculator = new StringCalculator();

    Action actual = () => stringCalculator.Add(input);

    Assert.Throws<ArgumentException>(actual);
}
```

Validate private methods by unit testing public methods

In most cases, there should not be a need to test a private method. Private methods are an implementation detail. You can think of it this way: private methods never exist in isolation. At some point, there is going to be a public facing method that calls the private method as part of its implementation. What you should care about is the end result of the public method that calls into the private one.

Consider the following case

```
public string ParseLogLine(string input)
{
    var sanitizedInput = TrimInput(input);
    return sanitizedInput;
}

private string TrimInput(string input)
{
    return input.Trim();
}
```

Your first reaction may be to start writing a test for `TrimInput` because you want to make sure that the method is working as expected. However, it is entirely possible that `ParseLogLine` manipulates `sanitizedInput` in such a way that you do not expect, rendering a test against `TrimInput` useless.

The real test should be done against the public facing method `ParseLogLine` because that is what you should ultimately care about.

```
public void ParseLogLine_ByDefault_ReturnsTrimmedResult()
{
    var parser = new Parser();

    var result = parser.ParseLogLine(" a ");

    Assert.Equals("a", result);
}
```

With this viewpoint, if you see a private method, find the public method and write your tests against that method. Just because a private method returns the expected result, does not mean the system that eventually calls the private method uses the result correctly.

Stub static references

One of the principles of a unit test is that it must have full control of the system under test. This can be problematic when production code includes calls to static references (e.g. `DateTime.Now`). Consider the following code

```
public int GetDiscountedPrice(int price)
{
    if(DateTime.Now.DayOfWeek == DayOfWeek.Tuesday)
    {
        return price / 2;
    }
    else
    {
        return price;
    }
}
```

How can this code possibly be unit tested? You may try an approach such as

```
public void GetDiscountedPrice_ByDefault_ReturnsFullPrice()
{
    var priceCalculator = new PriceCalculator();

    var actual = priceCalculator.GetDiscountedPrice(2);

    Assert.Equals(2, actual)
}

public void GetDiscountedPrice_OnTuesday_ReturnsHalfPrice()
{
    var priceCalculator = new PriceCalculator();

    var actual = priceCalculator.GetDiscountedPrice(2);

    Assert.Equals(1, actual);
}
```

Unfortunately, you will quickly realize that there are a couple problems with your tests.

- If the test suite is run on a Tuesday, the second test will pass, but the first test will fail.
- If the test suite is run on any other day, the first test will pass, but the second test will fail.

To solve these problems, you'll need to introduce a *seam* into your production code. One approach is to wrap the code that you need to control in an interface and have the production code depend on that interface.

```
public interface IDateTimeProvider
{
    DayOfWeek DayOfWeek();
}

public int GetDiscountedPrice(int price, IDateTimeProvider dateTimeProvider)
{
    if(dateTimeProvider.DayOfWeek() == DayOfWeek.Tuesday)
    {
        return price / 2;
    }
    else
    {
        return price;
    }
}
```

Your test suite now becomes

```
public void GetDiscountedPrice_ByDefault_ReturnsFullPrice()
{
    var priceCalculator = new PriceCalculator();
    var dateTimeProviderStub = new Mock<IDateTimeProvider>();
    dateTimeProviderStub.Setup(dtp => dtp.DayOfWeek()).Returns(DayOfWeek.Monday);

    var actual = priceCalculator.GetDiscountedPrice(2, dateTimeProviderStub);

    Assert.Equals(2, actual);
}

public void GetDiscountedPrice_OnTuesday_ReturnsHalfPrice()
{
    var priceCalculator = new PriceCalculator();
    var dateTimeProviderStub = new Mock<IDateTimeProvider>();
    dateTimeProviderStub.Setup(dtp => dtp.DayOfWeek()).Returns(DayOfWeek.Tuesday);

    var actual = priceCalculator.GetDiscountedPrice(2, dateTimeProviderStub);

    Assert.Equals(1, actual);
}
```

Now the test suite has full control over `DateTime.Now` and can stub any value when calling into the method.

Unit testing C# in .NET Core using dotnet test and xUnit

9/19/2019 • 4 minutes to read • Edit Online

This tutorial takes you through an interactive experience building a sample solution step-by-step to learn unit testing concepts. If you prefer to follow the tutorial using a pre-built solution, view or download the sample code before you begin. For download instructions, see Samples and Tutorials.

This article is about testing a .NET Core project. If you're testing an **ASP.NET Core** project, see Integration tests in ASP.NET Core.

Creating the source project

Open a shell window. Create a directory called *unit-testing-using-dotnet-test* to hold the solution. Inside this new directory, run `dotnet new sln` to create a new solution. Having a solution makes it easier to manage both the class library and the unit test project. Inside the solution directory, create a *PrimeService* directory. The directory and file structure thus far should be as follows:

```
/unit-testing-using-dotnet-test
    unit-testing-using-dotnet-test.sln
    /PrimeService
```

Make *PrimeService* the current directory and run `dotnet new classlib` to create the source project. Rename *Class1.cs* to *PrimeService.cs*. You first create a failing implementation of the `PrimeService` class:

```
using System;

namespace Prime.Services
{
    public class PrimeService
    {
        public bool IsPrime(int candidate)
        {
            throw new NotImplementedException("Please create a test first.");
        }
    }
}
```

Change the directory back to the *unit-testing-using-dotnet-test* directory.

Run the dotnet sln command to add the class library project to the solution:

```
dotnet sln add ./PrimeService/PrimeService.csproj
```

Creating the test project

Next, create the *PrimeService.Tests* directory. The following outline shows the directory structure:

```
/unit-testing-using-dotnet-test
    unit-testing-using-dotnet-test.sln
    /PrimeService
        Source Files
        PrimeService.csproj
    /PrimeService.Tests
```

Make the *PrimeService.Tests* directory the current directory and create a new project using `dotnet new xunit`. This command creates a test project that uses xUnit as the test library. The generated template configures the test runner in the *PrimeServiceTests.csproj* file similar to the following code:

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.3.0" />
  <PackageReference Include="xunit" Version="2.2.0" />
  <PackageReference Include="xunit.runner.visualstudio" Version="2.2.0" />
</ItemGroup>
```

The test project requires other packages to create and run unit tests. `dotnet new` in the previous step added xUnit and the xUnit runner. Now, add the `PrimeService` class library as another dependency to the project. Use the `dotnet add reference` command:

```
dotnet add reference ../PrimeService/PrimeService.csproj
```

You can see the entire file in the samples repository on GitHub.

The following shows the final solution layout:

```
/unit-testing-using-dotnet-test
    unit-testing-using-dotnet-test.sln
    /PrimeService
        Source Files
        PrimeService.csproj
    /PrimeService.Tests
        Test Source Files
        PrimeServiceTests.csproj
```

To add the test project to the solution, run the dotnet sln command in the *unit-testing-using-dotnet-test* directory:

```
dotnet sln add ./PrimeService.Tests/PrimeService.Tests.csproj
```

Creating the first test

You write one failing test, make it pass, then repeat the process. Remove *UnitTest1.cs* from the *PrimeService.Tests* directory and create a new C# file named *PrimeService_IsPrimeShould.cs*. Add the following code:

```
using Xunit;
using Prime.Services;

namespace Prime.UnitTests.Services
{
    public class PrimeService_IsPrimeShould
    {
        private readonly PrimeService _primeService;

        public PrimeService_IsPrimeShould()
        {
            _primeService = new PrimeService();
        }

        [Fact]
        public void IsPrime_InputIs1_ReturnFalse()
        {
            var result = _primeService.IsPrime(1);

            Assert.False(result, "1 should not be prime");
        }
    }
}
```

The `[Fact]` attribute indicates a test method that is run by the test runner. From the *PrimeService.Tests* folder, execute `dotnet test` to build the tests and the class library and then run the tests. The xUnit test runner contains the program entry point to run your tests. `dotnet test` starts the test runner using the unit test project you've created.

Your test fails. You haven't created the implementation yet. Make this test pass by writing the simplest code in the `PrimeService` class that works. Replace the existing `IsPrime` method implementation with the following code:

```
public bool IsPrime(int candidate)
{
    if (candidate == 1)
    {
        return false;
    }
    throw new NotImplementedException("Please create a test first.");
}
```

In the *PrimeService.Tests* directory, run `dotnet test` again. The `dotnet test` command runs a build for the `PrimeService` project and then for the `PrimeService.Tests` project. After building both projects, it runs this single test. It passes.

Adding more features

Now that you've made one test pass, it's time to write more. There are a few other simple cases for prime numbers: 0, -1. You could add those cases as new tests with the `[Fact]` attribute, but that quickly becomes tedious. There are other xUnit attributes that enable you to write a suite of similar tests:

- `[Theory]` represents a suite of tests that execute the same code but have different input arguments.
- `[InlineData]` attribute specifies values for those inputs.

Instead of creating new tests, apply these two attributes, `[Theory]` and `[InlineData]`, to create a single theory in the *PrimeService_IsPrimeShould.cs* file. The theory is a method that tests several values less than two, which is the lowest prime number:

```
[Theory]
[InlineData(-1)]
[InlineData(0)]
[InlineData(1)]
public void IsPrime_ValuesLessThan2_ReturnFalse(int value)
{
    var result = _primeService.IsPrime(value);

    Assert.False(result, $"{value} should not be prime");
}
```

Run `dotnet test` again, and two of these tests should fail. To make all of the tests pass, change the `if` clause at the beginning of the `IsPrime` method in the *PrimeService.cs* file:

```
if (candidate < 2)
```

Continue to iterate by adding more tests, more theories, and more code in the main library. You have the finished version of the tests and the complete implementation of the library.

Additional resources

- xUnit.net official site
- Testing controller logic in ASP.NET Core

Unit testing C# with NUnit and .NET Core

9/19/2019 • 4 minutes to read • Edit Online

This tutorial takes you through an interactive experience building a sample solution step-by-step to learn unit testing concepts. If you prefer to follow the tutorial using a pre-built solution, view or download the sample code before you begin. For download instructions, see Samples and Tutorials.

This article is about testing a .NET Core project. If you're testing an **ASP.NET Core** project, see Integration tests in ASP.NET Core.

Prerequisites

- .NET Core 2.1 SDK or later versions.
- A text editor or code editor of your choice.

Creating the source project

Open a shell window. Create a directory called *unit-testing-using-nunit* to hold the solution. Inside this new directory, run the following command to create a new solution file for the class library and the test project:

```
dotnet new sln
```

Next, create a *PrimeService* directory. The following outline shows the directory and file structure so far:

```
/unit-testing-using-nunit
    unit-testing-using-nunit.sln
    /PrimeService
```

Make *PrimeService* the current directory and run the following command to create the source project:

```
dotnet new classlib
```

Rename *Class1.cs* to *PrimeService.cs*. You create a failing implementation of the `PrimeService` class:

```
using System;

namespace Prime.Services
{
    public class PrimeService
    {
        public bool IsPrime(int candidate)
        {
            throw new NotImplementedException("Please create a test first.");
        }
    }
}
```

Change the directory back to the *unit-testing-using-nunit* directory. Run the following command to add the class library project to the solution:

```
dotnet sln add PrimeService/PrimeService.csproj
```

Creating the test project

Next, create the *PrimeService.Tests* directory. The following outline shows the directory structure:

```
/unit-testing-using-nunit
    unit-testing-using-nunit.sln
    /PrimeService
        Source Files
        PrimeService.csproj
    /PrimeService.Tests
```

Make the *PrimeService.Tests* directory the current directory and create a new project using the following command:

```
dotnet new nunit
```

The dotnet new command creates a test project that uses NUnit as the test library. The generated template configures the test runner in the *PrimeService.Tests.csproj* file:

```
<ItemGroup>
  <PackageReference Include="nunit" Version="3.12.0" />
  <PackageReference Include="NUnit3TestAdapter" Version="3.15.1" />
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.4.0" />
</ItemGroup>
```

The test project requires other packages to create and run unit tests. `dotnet new` in the previous step added the Microsoft test SDK, the NUnit test framework, and the NUnit test adapter. Now, add the `PrimeService` class library as another dependency to the project. Use the `dotnet add reference` command:

```
dotnet add reference ../PrimeService/PrimeService.csproj
```

You can see the entire file in the samples repository on GitHub.

The following outline shows the final solution layout:

```
/unit-testing-using-nunit
    unit-testing-using-nunit.sln
    /PrimeService
        Source Files
        PrimeService.csproj
    /PrimeService.Tests
        Test Source Files
        PrimeService.Tests.csproj
```

Execute the following command in the *unit-testing-using-nunit* directory:

```
dotnet sln add ./PrimeService.Tests/PrimeService.Tests.csproj
```

Creating the first test

You write one failing test, make it pass, then repeat the process. In the *PrimeService.Tests* directory, rename the *UnitTest1.cs* file to *PrimeService_IsPrimeShould.cs* and replace its entire contents with the following code:

```
using NUnit.Framework;
using Prime.Services;

namespace Prime.UnitTests.Services
{
    [TestFixture]
    public class PrimeService_IsPrimeShould
    {
        [Test]
        public void IsPrime_InputIs1_ReturnFalse()
        {
            PrimeService primeService = CreatePrimeService();
            var result = primeService.IsPrime(1);

            Assert.IsFalse(result, "1 should not be prime");
        }

        /*
        More tests
        */

        private PrimeService CreatePrimeService()
        {
             return new PrimeService();
        }
    }
}
```

The `[TestFixture]` attribute denotes a class that contains unit tests. The `[Test]` attribute indicates a method is a test method.

Save this file and execute `dotnet test` to build the tests and the class library and then run the tests. The NUnit test runner contains the program entry point to run your tests. `dotnet test` starts the test runner using the unit test project you've created.

Your test fails. You haven't created the implementation yet. Make this test pass by writing the simplest code in the `PrimeService` class that works:

```
public bool IsPrime(int candidate)
{
    if (candidate == 1)
    {
        return false;
    }
    throw new NotImplementedException("Please create a test first.");
}
```

In the *unit-testing-using-nunit* directory, run `dotnet test` again. The `dotnet test` command runs a build for the `PrimeService` project and then for the `PrimeService.Tests` project. After building both projects, it runs this single test. It passes.

Adding more features

Now that you've made one test pass, it's time to write more. There are a few other simple cases for prime numbers: 0, -1. You could add new tests with the `[Test]` attribute, but that quickly becomes tedious. There are other NUnit attributes that enable you to write a suite of similar tests. A `[TestCase]` attribute is used to create a suite of tests that execute the same code but have different input arguments. You can use the `[TestCase]` attribute to specify values for those inputs.

Instead of creating new tests, apply this attribute to create a single data driven test. The data driven test is a method that tests several values less than two, which is the lowest prime number:

```
[TestCase(-1)]
[TestCase(0)]
[TestCase(1)]
public void IsPrime_ValuesLessThan2_ReturnFalse(int value)
{
    var result = _primeService.IsPrime(value);

    Assert.IsFalse(result, $"{value} should not be prime");
}
```

Run `dotnet test`, and two of these tests fail. To make all of the tests pass, change the `if` clause at the beginning of the `Main` method in the *PrimeService.cs* file:

```
if (candidate < 2)
```

Continue to iterate by adding more tests, more theories, and more code in the main library. You have the finished version of the tests and the complete implementation of the library.

You've built a small library and a set of unit tests for that library. You've structured the solution so that adding new packages and tests is part of the normal workflow. You've concentrated most of your time and effort on solving the goals of the application.

Unit testing C# with MSTest and .NET Core

9/19/2019 • 4 minutes to read • Edit Online

This tutorial takes you through an interactive experience building a sample solution step-by-step to learn unit testing concepts. If you prefer to follow the tutorial using a pre-built solution, view or download the sample code before you begin. For download instructions, see Samples and Tutorials.

This article is about testing a .NET Core project. If you're testing an **ASP.NET Core** project, see Integration tests in ASP.NET Core.

Create the source project

Open a shell window. Create a directory called *unit-testing-using-mstest* to hold the solution. Inside this new directory, run `dotnet new sln` to create a new solution file for the class library and the test project. Next, create a *PrimeService* directory. The following outline shows the directory and file structure thus far:

```
/unit-testing-using-mstest
    unit-testing-using-mstest.sln
    /PrimeService
```

Make *PrimeService* the current directory and run `dotnet new classlib` to create the source project. Rename *Class1.cs* to *PrimeService.cs*. You create a failing implementation of the `PrimeService` class:

```
using System;

namespace Prime.Services
{
    public class PrimeService
    {
        public bool IsPrime(int candidate)
        {
            throw new NotImplementedException("Please create a test first.");
        }
    }
}
```

Change the directory back to the *unit-testing-using-mstest* directory. Run `dotnet sln add PrimeService/PrimeService.csproj` to add the class library project to the solution.

Create the test project

Next, create the *PrimeService.Tests* directory. The following outline shows the directory structure:

```
/unit-testing-using-mstest
    unit-testing-using-mstest.sln
    /PrimeService
        Source Files
        PrimeService.csproj
    /PrimeService.Tests
```

Make the *PrimeService.Tests* directory the current directory and create a new project using `dotnet new mstest`. The dotnet new command creates a test project that uses MSTest as the test library. The generated template configures the test runner in the *PrimeServiceTests.csproj* file:

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.3.0" />
  <PackageReference Include="MSTest.TestAdapter" Version="1.1.18" />
  <PackageReference Include="MSTest.TestFramework" Version="1.1.18" />
</ItemGroup>
```

The test project requires other packages to create and run unit tests. `dotnet new` in the previous step added the MSTest SDK, the MSTest test framework, and the MSTest runner. Now, add the `PrimeService` class library as another dependency to the project. Use the `dotnet add reference` command:

```
dotnet add reference ../PrimeService/PrimeService.csproj
```

You can see the entire file in the samples repository on GitHub.

The following outline shows the final solution layout:

```
/unit-testing-using-mstest
    unit-testing-using-mstest.sln
    /PrimeService
        Source Files
        PrimeService.csproj
    /PrimeService.Tests
        Test Source Files
        PrimeServiceTests.csproj
```

Execute `dotnet sln add .\PrimeService.Tests\PrimeService.Tests.csproj` in the *unit-testing-using-mstest* directory.

Create the first test

You write one failing test, make it pass, then repeat the process. Remove *UnitTest1.cs* from the *PrimeService.Tests* directory and create a new C# file named *PrimeService_IsPrimeShould.cs* with the following content:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Prime.Services;

namespace Prime.UnitTests.Services
{
    [TestClass]
    public class PrimeService_IsPrimeShould
    {
        private readonly PrimeService _primeService;

        public PrimeService_IsPrimeShould()
        {
            _primeService = new PrimeService();
        }

        [TestMethod]
        public void IsPrime_InputIs1_ReturnFalse()
        {
            var result = _primeService.IsPrime(1);

            Assert.IsFalse(result, "1 should not be prime");
        }
    }
}
```

The TestClass attribute denotes a class that contains unit tests. The TestMethod attribute indicates a method is a test method.

Save this file and execute `dotnet test` to build the tests and the class library and then run the tests. The MSTest test runner contains the program entry point to run your tests. `dotnet test` starts the test runner using the unit test project you've created.

Your test fails. You haven't created the implementation yet. Make this test pass by writing the simplest code in the `PrimeService` class that works:

```
public bool IsPrime(int candidate)
{
    if (candidate == 1)
    {
        return false;
    }
    throw new NotImplementedException("Please create a test first.");
}
```

In the *unit-testing-using-mstest* directory, run `dotnet test` again. The `dotnet test` command runs a build for the `PrimeService` project and then for the `PrimeService.Tests` project. After building both projects, it runs this single test. It passes.

Add more features

Now that you've made one test pass, it's time to write more. There are a few other simple cases for prime numbers: 0, -1. You could add new tests with the TestMethod attribute, but that quickly becomes tedious. There are other MSTest attributes that enable you to write a suite of similar tests. A DataTestMethod attribute represents a suite of tests that execute the same code but have different input arguments. You can use the DataRow attribute to specify values for those inputs.

Instead of creating new tests, apply these two attributes to create a single data driven test. The data driven test is a method that tests several values less than two, which is the lowest prime number:

```
[DataTestMethod]
[DataRow(-1)]
[DataRow(0)]
[DataRow(1)]
public void IsPrime_ValuesLessThan2_ReturnFalse(int value)
{
    var result = _primeService.IsPrime(value);

    Assert.IsFalse(result, $"{value} should not be prime");
}
```

Run `dotnet test`, and two of these tests fail. To make all of the tests pass, change the `if` clause at the beginning of the method:

```
if (candidate < 2)
```

Continue to iterate by adding more tests, more theories, and more code in the main library. You have the finished version of the tests and the complete implementation of the library.

You've built a small library and a set of unit tests for that library. You've structured the solution so that adding new packages and tests is part of the normal workflow. You've concentrated most of your time and effort on solving the goals of the application.

See also

- Microsoft.VisualStudio.TestTools.UnitTesting
- Use the MSTest framework in unit tests
- MSTest V2 test framework docs

Unit testing F# libraries in .NET Core using dotnet test and xUnit

9/19/2019 • 4 minutes to read • Edit Online

This tutorial takes you through an interactive experience building a sample solution step-by-step to learn unit testing concepts. If you prefer to follow the tutorial using a pre-built solution, view or download the sample code before you begin. For download instructions, see Samples and Tutorials.

This article is about testing a .NET Core project. If you're testing an **ASP.NET Core** project, see Integration tests in ASP.NET Core.

Creating the source project

Open a shell window. Create a directory called *unit-testing-with-fsharp* to hold the solution. Inside this new directory, run `dotnet new sln` to create a new solution. This makes it easier to manage both the class library and the unit test project. Inside the solution directory, create a *MathService* directory. The directory and file structure thus far is shown below:

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
```

Make *MathService* the current directory and run `dotnet new classlib -lang F#` to create the source project. You'll create a failing implementation of the math service:

```
module MyMath =
    let squaresOfOdds xs = raise (System.NotImplementedException("You haven't written a test yet!"))
```

Change the directory back to the *unit-testing-with-fsharp* directory. Run `dotnet sln add .\MathService\MathService.fsproj` to add the class library project to the solution.

Creating the test project

Next, create the *MathService.Tests* directory. The following outline shows the directory structure:

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
        Source Files
        MathService.fsproj
    /MathService.Tests
```

Make the *MathService.Tests* directory the current directory and create a new project using `dotnet new xunit -lang F#`. This creates a test project that uses xUnit as the test library. The generated template configures the test runner in the *MathServiceTests.fsproj*:

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.3.0-preview-20170628-02" />
  <PackageReference Include="xunit" Version="2.2.0" />
  <PackageReference Include="xunit.runner.visualstudio" Version="2.2.0" />
</ItemGroup>
```

The test project requires other packages to create and run unit tests. `dotnet new` in the previous step added xUnit and the xUnit runner. Now, add the `MathService` class library as another dependency to the project. Use the `dotnet add reference` command:

```
dotnet add reference ../MathService/MathService.fsproj
```

You can see the entire file in the samples repository on GitHub.

You have the following final solution layout:

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
        Source Files
        MathService.fsproj
    /MathService.Tests
        Test Source Files
        MathServiceTests.fsproj
```

Execute `dotnet sln add .\MathService.Tests\MathService.Tests.fsproj` in the *unit-testing-with-fsharp* directory.

Creating the first test

You write one failing test, make it pass, then repeat the process. Open *Tests.fs* and add the following code:

```
[<Fact>]
let ``My test`` () =
    Assert.True(true)

[<Fact>]
let ``Fail every time`` () = Assert.True(false)
```

The `[<Fact>]` attribute denotes a test method that is run by the test runner. From the*unit-testing-with-fsharp*, execute `dotnet test` to build the tests and the class library and then run the tests. The xUnit test runner contains the program entry point to run your tests. `dotnet test` starts the test runner using the unit test project you've created.

These two tests show the most basic passing and failing tests. `My test` passes, and `Fail every time` fails. Now, create a test for the `squaresOfOdds` method. The `squaresOfOdds` method returns a sequence of the squares of all odd integer values that are part of the input sequence. Rather than trying to write all of those functions at once, you can iteratively create tests that validate the functionality. Making each test pass means creating the necessary functionality for the method.

The simplest test we can write is to call `squaresOfOdds` with all even numbers, where the result should be an empty sequence of integers. Here's that test:

```
[<Fact>]
let ``Sequence of Evens returns empty collection`` () =
    let expected = Seq.empty<int>
    let actual = MyMath.squaresOfOdds [2; 4; 6; 8; 10]
    Assert.Equal<Collections.Generic.IEnumerable<int>>(expected, actual)
```

Your test fails. You haven't created the implementation yet. Make this test pass by writing the simplest code in the `MathService` class that works:

```
let squaresOfOdds xs =
    Seq.empty<int>
```

In the *unit-testing-with-fsharp* directory, run `dotnet test` again. The `dotnet test` command runs a build for the `MathService` project and then for the `MathService.Tests` project. After building both projects, it runs this single test. It passes.

Completing the requirements

Now that you've made one test pass, it's time to write more. The next simple case works with a sequence whose only odd number is `1`. The number 1 is easier because the square of 1 is 1. Here's that next test:

```
[<Fact>]
let ``Sequences of Ones and Evens returns Ones`` () =
    let expected = [1; 1; 1; 1]
    let actual = MyMath.squaresOfOdds [2; 1; 4; 1; 6; 1; 8; 1; 10]
    Assert.Equal<Collections.Generic.IEnumerable<int>>(expected, actual)
```

Executing `dotnet test` runs your tests and shows you that the new test fails. Now, update the `squaresOfOdds` method to handle this new test. You filter all the even numbers out of the sequence to make this test pass. You can do that by writing a small filter function and using `Seq.filter`:

```
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd
```

There's one more step to go: square each of the odd numbers. Start by writing a new test:

```
[<Fact>]
let ``SquaresOfOdds works`` () =
    let expected = [1; 9; 25; 49; 81]
    let actual = MyMath.squaresOfOdds [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
    Assert.Equal(expected, actual)
```

You can fix the test by piping the filtered sequence through a map operation to compute the square of each odd number:

```
let private square x = x * x
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd
    |> Seq.map square
```

You've built a small library and a set of unit tests for that library. You've structured the solution so that adding new packages and tests is part of the normal workflow. You've concentrated most of your time and effort on solving the goals of the application.

Unit testing F# libraries in .NET Core using dotnet test and NUnit

9/19/2019 • 5 minutes to read • Edit Online

This tutorial takes you through an interactive experience building a sample solution step-by-step to learn unit testing concepts. If you prefer to follow the tutorial using a pre-built solution, view or download the sample code before you begin. For download instructions, see Samples and Tutorials.

This article is about testing a .NET Core project. If you're testing an **ASP.NET Core** project, see Integration tests in ASP.NET Core.

Prerequisites

- .NET Core 2.1 SDK or later versions.
- A text editor or code editor of your choice.

Creating the source project

Open a shell window. Create a directory called *unit-testing-with-fsharp* to hold the solution. Inside this new directory, run the following command to create a new solution file for the class library and the test project:

```
dotnet new sln
```

Next, create a *MathService* directory. The following outline shows the directory and file structure so far:

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
```

Make *MathService* the current directory and run the following command to create the source project:

```
dotnet new classlib -lang F#
```

You create a failing implementation of the math service:

```
module MyMath =
    let squaresOfOdds xs = raise (System.NotImplementedException("You haven't written a test yet!"))
```

Change the directory back to the *unit-testing-with-fsharp* directory. Run the following command to add the class library project to the solution:

```
dotnet sln add .\MathService\MathService.fsproj
```

Creating the test project

Next, create the *MathService.Tests* directory. The following outline shows the directory structure:

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
        Source Files
        MathService.fsproj
    /MathService.Tests
```

Make the *MathService.Tests* directory the current directory and create a new project using the following command:

```
dotnet new nunit -lang F#
```

This creates a test project that uses NUnit as the test framework. The generated template configures the test runner in the *MathServiceTests.fsproj*:

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.5.0" />
  <PackageReference Include="NUnit" Version="3.9.0" />
  <PackageReference Include="NUnit3TestAdapter" Version="3.9.0" />
</ItemGroup>
```

The test project requires other packages to create and run unit tests. `dotnet new` in the previous step added NUnit and the NUnit test adapter. Now, add the `MathService` class library as another dependency to the project. Use the `dotnet add reference` command:

```
dotnet add reference ../MathService/MathService.fsproj
```

You can see the entire file in the samples repository on GitHub.

You have the following final solution layout:

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
        Source Files
        MathService.fsproj
    /MathService.Tests
        Test Source Files
        MathService.Tests.fsproj
```

Execute the following command in the *unit-testing-with-fsharp* directory:

```
dotnet sln add .\MathService.Tests\MathService.Tests.fsproj
```

Creating the first test

You write one failing test, make it pass, then repeat the process. Open *UnitTest1.fs* and add the following code:

```
namespace MathService.Tests

open System
open NUnit.Framework
open MathService

[<TestFixture>]
type TestClass () =

    [<Test>]
    member this.TestMethodPassing() =
        Assert.True(true)

    [<Test>]
     member this.FailEveryTime() = Assert.True(false)
```

The `[<TestFixture>]` attribute denotes a class that contains tests. The `[<Test>]` attribute denotes a test method that is run by the test runner. From the *unit-testing-with-fsharp* directory, execute `dotnet test` to build the tests and the class library and then run the tests. The NUnit test runner contains the program entry point to run your tests. `dotnet test` starts the test runner using the unit test project you've created.

These two tests show the most basic passing and failing tests. `My test` passes, and `Fail every time` fails. Now, create a test for the `squaresOfOdds` method. The `squaresOfOdds` method returns a sequence of the squares of all odd integer values that are part of the input sequence. Rather than trying to write all of those functions at once, you can iteratively create tests that validate the functionality. Making each test pass means creating the necessary functionality for the method.

The simplest test we can write is to call `squaresOfOdds` with all even numbers, where the result should be an empty sequence of integers. Here's that test:

```
[<Test>]
member this.TestEvenSequence() =
    let expected = Seq.empty<int>
    let actual = MyMath.squaresOfOdds [2; 4; 6; 8; 10]
    Assert.That(actual, Is.EqualTo(expected))
```

Notice that the `expected` sequence has been converted to a list. The NUnit framework relies on many standard .NET types. That dependency means that your public interface and expected results support ICollection rather than IEnumerable.

When you run the test, you see that your test fails. You haven't created the implementation yet. Make this test pass by writing the simplest code in the *Library.fs* class in your MathService project that works:

```
let squaresOfOdds xs =
    Seq.empty<int>
```

In the *unit-testing-with-fsharp* directory, run `dotnet test` again. The `dotnet test` command runs a build for the `MathService` project and then for the `MathService.Tests` project. After building both projects, it runs your tests. Two tests pass now.

Completing the requirements

Now that you've made one test pass, it's time to write more. The next simple case works with a sequence whose only odd number is `1`. The number 1 is easier because the square of 1 is 1. Here's that next test:

```
[<Test>]
member public this.TestOnesAndEvens() =
    let expected = [1; 1; 1; 1]
    let actual = MyMath.squaresOfOdds [2; 1; 4; 1; 6; 1; 8; 1; 10]
    Assert.That(actual, Is.EqualTo(expected))
```

Executing `dotnet test` fails the new test. You must update the `squaresOfOdds` method to handle this new test. You must filter all the even numbers out of the sequence to make this test pass. You can do that by writing a small filter function and using `Seq.filter`:

```
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd
```

Notice the call to `Seq.toList`. That creates a list, which implements the ICollection interface.

There's one more step to go: square each of the odd numbers. Start by writing a new test:

```
[<Test>]
member public this.TestSquaresOfOdds() =
    let expected = [1; 9; 25; 49; 81]
    let actual = MyMath.squaresOfOdds [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
    Assert.That(actual, Is.EqualTo(expected))
```

You can fix the test by piping the filtered sequence through a map operation to compute the square of each odd number:

```
let private square x = x * x
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd
    |> Seq.map square
```

You've built a small library and a set of unit tests for that library. You've structured the solution so that adding new packages and tests is part of the normal workflow. You've concentrated most of your time and effort on solving the goals of the application.

Unit testing F# libraries in .NET Core using dotnet test and MSTest

9/19/2019 • 5 minutes to read • Edit Online

This tutorial takes you through an interactive experience building a sample solution step-by-step to learn unit testing concepts. If you prefer to follow the tutorial using a pre-built solution, view or download the sample code before you begin. For download instructions, see Samples and Tutorials.

This article is about testing a .NET Core project. If you're testing an **ASP.NET Core** project, see Integration tests in ASP.NET Core.

Creating the source project

Open a shell window. Create a directory called *unit-testing-with-fsharp* to hold the solution. Inside this new directory, run `dotnet new sln` to create a new solution. This makes it easier to manage both the class library and the unit test project. Inside the solution directory, create a *MathService* directory. The directory and file structure thus far is shown below:

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
```

Make *MathService* the current directory and run `dotnet new classlib -lang F#` to create the source project. You'll create a failing implementation of the math service:

```
module MyMath =
    let squaresOfOdds xs = raise (System.NotImplementedException("You haven't written a test yet!"))
```

Change the directory back to the *unit-testing-with-fsharp* directory. Run `dotnet sln add .\MathService\MathService.fsproj` to add the class library project to the solution.

Creating the test project

Next, create the *MathService.Tests* directory. The following outline shows the directory structure:

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
        Source Files
        MathService.fsproj
    /MathService.Tests
```

Make the *MathService.Tests* directory the current directory and create a new project using `dotnet new mstest -lang F#`. This creates a test project that uses MSTest as the test framework. The generated template configures the test runner in the *MathServiceTests.fsproj*:

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.3.0-preview-20170628-02" />
  <PackageReference Include="MSTest.TestAdapter" Version="1.1.18" />
  <PackageReference Include="MSTest.TestFramework" Version="1.1.18" />
</ItemGroup>
```

The test project requires other packages to create and run unit tests. `dotnet new` in the previous step added MSTest and the MSTest runner. Now, add the `MathService` class library as another dependency to the project. Use the `dotnet add reference` command:

```
dotnet add reference ../MathService/MathService.fsproj
```

You can see the entire file in the samples repository on GitHub.

You have the following final solution layout:

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
        Source Files
        MathService.fsproj
    /MathService.Tests
        Test Source Files
        MathServiceTests.fsproj
```

Execute `dotnet sln add .\MathService.Tests\MathService.Tests.fsproj` in the *unit-testing-with-fsharp* directory.

Creating the first test

You write one failing test, make it pass, then repeat the process. Open *Tests.fs* and add the following code:

```
namespace MathService.Tests

open System
open Microsoft.VisualStudio.TestTools.UnitTesting
open MathService

[<TestClass>]
type TestClass () =

    [<TestMethod>]
    member this.TestMethodPassing() =
        Assert.IsTrue(true)

    [<TestMethod>]
     member this.FailEveryTime() = Assert.IsTrue(false)
```

The `[<TestClass>]` attribute denotes a class that contains tests. The `[<TestMethod>]` attribute denotes a test method that is run by the test runner. From the *unit-testing-with-fsharp* directory, execute `dotnet test` to build the tests and the class library and then run the tests. The MSTest test runner contains the program entry point to run your tests. `dotnet test` starts the test runner using the unit test project you've created.

These two tests show the most basic passing and failing tests. `My test` passes, and `Fail every time` fails. Now, create a test for the `squaresOfOdds` method. The `squaresOfOdds` method returns a list of the squares of all odd integer values that are part of the input sequence. Rather than trying to write all of those functions at once, you can iteratively create tests that validate the functionality. Making each test pass means creating the necessary functionality for the method.

The simplest test we can write is to call `squaresOfOdds` with all even numbers, where the result should be an empty sequence of integers. Here's that test:

```
[<TestMethod>]
member this.TestEvenSequence() =
    let expected = Seq.empty<int> |> Seq.toList
    let actual = MyMath.squaresOfOdds [2; 4; 6; 8; 10]
    Assert.AreEqual(expected, actual)
```

Notice that the `expected` sequence has been converted to a list. The MSTest library relies on many standard .NET types. That dependency means that your public interface and expected results support ICollection rather than IEnumerable.

When you run the test, you see that your test fails. You haven't created the implementation yet. Make this test pass by writing the simplest code in the `Mathservice` class that works:

```
let squaresOfOdds xs =
    Seq.empty<int> |> Seq.toList
```

In the *unit-testing-with-fsharp* directory, run `dotnet test` again. The `dotnet test` command runs a build for the `MathService` project and then for the `MathService.Tests` project. After building both projects, it runs this single test. It passes.

Completing the requirements

Now that you've made one test pass, it's time to write more. The next simple case works with a sequence whose only odd number is `1`. The number 1 is easier because the square of 1 is 1. Here's that next test:

```
[<TestMethod>]
member public this.TestOnesAndEvens() =
    let expected = [1; 1; 1; 1]
    let actual = MyMath.squaresOfOdds [2; 1; 4; 1; 6; 1; 8; 1; 10]
    Assert.AreEqual(expected, actual)
```

Executing `dotnet test` fails the new test. You must update the `squaresOfOdds` method to handle this new test. You must filter all the even numbers out of the sequence to make this test pass. You can do that by writing a small filter function and using `Seq.filter`:

```
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd |> Seq.toList
```

Notice the call to `Seq.toList`. That creates a list, which implements the ICollection interface.

There's one more step to go: square each of the odd numbers. Start by writing a new test:

```
[<TestMethod>]
member public this.TestSquaresOfOdds() =
    let expected = [1; 9; 25; 49; 81]
    let actual = MyMath.squaresOfOdds [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
    Assert.AreEqual(expected, actual)
```

You can fix the test by piping the filtered sequence through a map operation to compute the square of each odd number:

```
let private square x = x * x
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd
    |> Seq.map square
    |> Seq.toList
```

You've built a small library and a set of unit tests for that library. You've structured the solution so that adding new packages and tests is part of the normal workflow. You've concentrated most of your time and effort on solving the goals of the application.

Unit testing Visual Basic .NET Core libraries using dotnet test and xUnit

9/19/2019 • 4 minutes to read • Edit Online

This tutorial takes you through an interactive experience building a sample solution step-by-step to learn unit testing concepts. If you prefer to follow the tutorial using a pre-built solution, view or download the sample code before you begin. For download instructions, see Samples and Tutorials.

This article is about testing a .NET Core project. If you're testing an **ASP.NET Core** project, see Integration tests in ASP.NET Core.

Creating the source project

Open a shell window. Create a directory called *unit-testing-vb-using-dotnet-test* to hold the solution. Inside this new directory, run `dotnet new sln` to create a new solution. This practice makes it easier to manage both the class library and the unit test project. Inside the solution directory, create a *PrimeService* directory. You have the following directory and file structure thus far:

```
/unit-testing-using-dotnet-test
    unit-testing-using-dotnet-test.sln
    /PrimeService
```

Make *PrimeService* the current directory and run `dotnet new classlib -lang VB` to create the source project. Rename *Class1.VB* to *PrimeService.VB*. You create a failing implementation of the `PrimeService` class:

```
Namespace Prime.Services
    Public Class PrimeService
        Public Function IsPrime(candidate As Integer) As Boolean
            Throw New NotImplementedException("Please create a test first")
        End Function
    End Class
End Namespace
```

Change the directory back to the *unit-testing-vb-using-dotnet-test* directory. Run `dotnet sln add .\PrimeService\PrimeService.vbproj` to add the class library project to the solution.

Creating the test project

Next, create the *PrimeService.Tests* directory. The following outline shows the directory structure:

```
/unit-testing-vb-using-dotnet-test
    unit-testing-vb-using-dotnet-test.sln
    /PrimeService
        Source Files
        PrimeService.vbproj
    /PrimeService.Tests
```

Make the *PrimeService.Tests* directory the current directory and create a new project using `dotnet new xunit -lang VB`. This command creates a test project that uses xUnit as the test library. The generated template configures the test runner in the *PrimeServiceTests.vbproj*:

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.3.0-preview-20170628-02" />
  <PackageReference Include="xunit" Version="2.2.0" />
  <PackageReference Include="xunit.runner.visualstudio" Version="2.2.0" />
</ItemGroup>
```

The test project requires other packages to create and run unit tests. `dotnet new` in the previous step added xUnit and the xUnit runner. Now, add the `PrimeService` class library as another dependency to the project. Use the `dotnet add reference` command:

```
dotnet add reference ../PrimeService/PrimeService.vbproj
```

You can see the entire file in the samples repository on GitHub.

You have the following final folder layout:

```
/unit-testing-using-dotnet-test
    unit-testing-using-dotnet-test.sln
    /PrimeService
        Source Files
        PrimeService.vbproj
    /PrimeService.Tests
        Test Source Files
        PrimeServiceTests.vbproj
```

Execute `dotnet sln add .\PrimeService.Tests\PrimeService.Tests.vbproj` in the *unit-testing-vb-using-dotnet-test* directory.

Creating the first test

You write one failing test, make it pass, then repeat the process. Remove *UnitTest1.vb* from the *PrimeService.Tests* directory and create a new Visual Basic file named *PrimeService_IsPrimeShould.VB*. Add the following code:

```
Imports Xunit

Namespace PrimeService.Tests
    Public Class PrimeService_IsPrimeShould
        Private _primeService As Prime.Services.PrimeService = New Prime.Services.PrimeService()

        <Fact>
        Sub IsPrime_InputIs1_ReturnFalse()
            Dim result As Boolean = _primeService.IsPrime(1)

            Assert.False(result, "1 should not be prime")
        End Sub

    End Class
End Namespace
```

The `<Fact>` attribute denotes a test method that is run by the test runner. From the*unit-testing-using-dotnet-test*, execute `dotnet test` to build the tests and the class library and then run the tests. The xUnit test runner contains the program entry point to run your tests. `dotnet test` starts the test runner using the unit test project you've created.

Your test fails. You haven't created the implementation yet. Make this test pass by writing the simplest code in the `PrimeService` class that works:

```
Public Function IsPrime(candidate As Integer) As Boolean
    If candidate = 1 Then
        Return False
    End If
    Throw New NotImplementedException("Please create a test first.")
End Function
```

In the *unit-testing-vb-using-dotnet-test* directory, run `dotnet test` again. The `dotnet test` command runs a build for the `PrimeService` project and then for the `PrimeService.Tests` project. After building both projects, it runs this single test. It passes.

Adding more features

Now that you've made one test pass, it's time to write more. There are a few other simple cases for prime numbers: 0, -1. You could add those cases as new tests with the `<Fact>` attribute, but that quickly becomes tedious. There are other xUnit attributes that enable you to write a suite of similar tests. A `<Theory>` attribute represents a suite of tests that execute the same code but have different input arguments. You can use the `<InlineData>` attribute to specify values for those inputs.

Instead of creating new tests, apply these two attributes to create a single theory. The theory is a method that tests several values less than two, which is the lowest prime number:

```
    Public Class PrimeService_IsPrimeShould
        Private _primeService As Prime.Services.PrimeService = New Prime.Services.PrimeService()

        <Theory>
        <InlineData(-1)>
        <InlineData(0)>
        <InlineData(1)>
        Sub IsPrime_ValueLessThan2_ReturnFalse(value As Integer)
            Dim result As Boolean = _primeService.IsPrime(value)

            Assert.False(result, $"{value} should not be prime")
        End Sub

        <Theory>
        <InlineData(2)>
        <InlineData(3)>
        <InlineData(5)>
        <InlineData(7)>
        Public Sub IsPrime_PrimesLessThan10_ReturnTrue(value As Integer)
            Dim result As Boolean = _primeService.IsPrime(value)

            Assert.True(result, $"{value} should be prime")
        End Sub

        <Theory>
        <InlineData(4)>
        <InlineData(6)>
        <InlineData(8)>
        <InlineData(9)>
        Public Sub IsPrime_NonPrimesLessThan10_ReturnFalse(value As Integer)
            Dim result As Boolean = _primeService.IsPrime(value)

            Assert.False(result, $"{value} should not be prime")
        End Sub
    End Class
```

Run `dotnet test`, and two of these tests fail. To make all of the tests pass, change the `if` clause at the beginning of the method:

```
if candidate < 2
```

Continue to iterate by adding more tests, more theories, and more code in the main library. You have the finished version of the tests and the complete implementation of the library.

You've built a small library and a set of unit tests for that library. You've structured the solution so that adding new packages and tests is part of the normal workflow. You've concentrated most of your time and effort on solving the goals of the application.

Unit testing Visual Basic .NET Core libraries using dotnet test and NUnit

9/19/2019 • 4 minutes to read • Edit Online

This tutorial takes you through an interactive experience building a sample solution step-by-step to learn unit testing concepts. If you prefer to follow the tutorial using a pre-built solution, view or download the sample code before you begin. For download instructions, see Samples and Tutorials.

This article is about testing a .NET Core project. If you're testing an **ASP.NET Core** project, see Integration tests in ASP.NET Core.

Prerequisites

- .NET Core 2.1 SDK or later versions.
- A text editor or code editor of your choice.

Creating the source project

Open a shell window. Create a directory called *unit-testing-vb-nunit* to hold the solution. Inside this new directory, run the following command to create a new solution file for the class library and the test project:

```
dotnet new sln
```

Next, create a *PrimeService* directory. The following outline shows the file structure so far:

```
/unit-testing-vb-nunit
    unit-testing-vb-nunit.sln
    /PrimeService
```

Make *PrimeService* the current directory and run the following command to create the source project:

```
dotnet new classlib -lang VB
```

Rename *Class1.VB* to *PrimeService.VB*. You create a failing implementation of the `PrimeService` class:

```
Imports System

Namespace Prime.Services
    Public Class PrimeService
        Public Function IsPrime(candidate As Integer) As Boolean
            Throw New NotImplementedException("Please create a test first.")
        End Function
    End Class
End Namespace
```

Change the directory back to the *unit-testing-vb-using-mstest* directory. Run the following command to add the class library project to the solution:

```
dotnet sln add .\PrimeService\PrimeService.vbproj
```

Creating the test project

Next, create the *PrimeService.Tests* directory. The following outline shows the directory structure:

```
/unit-testing-vb-nunit
    unit-testing-vb-nunit.sln
    /PrimeService
        Source Files
        PrimeService.vbproj
    /PrimeService.Tests
```

Make the *PrimeService.Tests* directory the current directory and create a new project using the following command:

```
dotnet new nunit -lang VB
```

The dotnet new command creates a test project that uses NUnit as the test library. The generated template configures the test runner in the *PrimeServiceTests.vbproj* file:

```
<ItemGroup>
  <PackageReference Include="nunit" Version="3.12.0" />
  <PackageReference Include="NUnit3TestAdapter" Version="3.15.1" />
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.4.0" />
</ItemGroup>
```

The test project requires other packages to create and run unit tests. `dotnet new` in the previous step added NUnit and the NUnit test adapter. Now, add the `PrimeService` class library as another dependency to the project. Use the `dotnet add reference` command:

```
dotnet add reference ../PrimeService/PrimeService.vbproj
```

You can see the entire file in the samples repository on GitHub.

You have the following final solution layout:

```
/unit-testing-vb-nunit
    unit-testing-vb-nunit.sln
    /PrimeService
        Source Files
        PrimeService.vbproj
    /PrimeService.Tests
        Test Source Files
        PrimeService.Tests.vbproj
```

Execute the following command in the *unit-testing-vb-nunit* directory:

```
dotnet sln add .\PrimeService.Tests\PrimeService.Tests.vbproj
```

Creating the first test

You write one failing test, make it pass, then repeat the process. In the *PrimeService.Tests* directory, rename the *UnitTest1.vb* file to *PrimeService_IsPrimeShould.VB* and replace its entire contents with the following code:

```
Imports NUnit.Framework

Namespace PrimeService.Tests
    <TestFixture>
    Public Class PrimeService_IsPrimeShould
        Private _primeService As Prime.Services.PrimeService = New Prime.Services.PrimeService()

        <Test>
        Sub IsPrime_InputIs1_ReturnFalse()
            Dim result As Boolean = _primeService.IsPrime(1)

            Assert.False(result, "1 should not be prime")
        End Sub

    End Class
End Namespace
```

The `<TestFixture>` attribute indicates a class that contains tests. The `<Test>` attribute denotes a method that is run by the test runner. From the *unit-testing-vb-nunit*, execute `dotnet test` to build the tests and the class library and then run the tests. The NUnit test runner contains the program entry point to run your tests. `dotnet test` starts the test runner using the unit test project you've created.

Your test fails. You haven't created the implementation yet. Make this test pass by writing the simplest code in the `PrimeService` class that works:

```
Public Function IsPrime(candidate As Integer) As Boolean
    If candidate = 1 Then
        Return False
    End If
    Throw New NotImplementedException("Please create a test first.")
End Function
```

In the *unit-testing-vb-nunit* directory, run `dotnet test` again. The `dotnet test` command runs a build for the `PrimeService` project and then for the `PrimeService.Tests` project. After building both projects, it runs this single test. It passes.

Adding more features

Now that you've made one test pass, it's time to write more. There are a few other simple cases for prime numbers: 0, -1. You could add those cases as new tests with the `<Test>` attribute, but that quickly becomes tedious. There are other xUnit attributes that enable you to write a suite of similar tests. A `<TestCase>` attribute represents a suite of tests that execute the same code but have different input arguments. You can use the `<TestCase>` attribute to specify values for those inputs.

Instead of creating new tests, apply these two attributes to create a series of tests that test several values less than two, which is the lowest prime number:

```
<TestFixture>
Public Class PrimeService_IsPrimeShould
    Private _primeService As Prime.Services.PrimeService = New Prime.Services.PrimeService()

    <TestCase(-1)>
    <TestCase(0)>
    <TestCase(1)>
    Sub IsPrime_ValuesLessThan2_ReturnFalse(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.IsFalse(result, $"{value} should not be prime")
    End Sub

    <TestCase(2)>
    <TestCase(3)>
    <TestCase(5)>
    <TestCase(7)>
    Public Sub IsPrime_PrimesLessThan10_ReturnTrue(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.IsTrue(result, $"{value} should be prime")
    End Sub

    <TestCase(4)>
    <TestCase(6)>
    <TestCase(8)>
    <TestCase(9)>
    Public Sub IsPrime_NonPrimesLessThan10_ReturnFalse(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.IsFalse(result, $"{value} should not be prime")
    End Sub
End Class
```

Run `dotnet test`, and two of these tests fail. To make all of the tests pass, change the `if` clause at the beginning of the `Main` method in the *PrimeServices.cs* file:

```
if candidate < 2
```

Continue to iterate by adding more tests, more theories, and more code in the main library. You have the finished version of the tests and the complete implementation of the library.

You've built a small library and a set of unit tests for that library. You've structured the solution so that adding new packages and tests is part of the normal workflow. You've concentrated most of your time and effort on solving the goals of the application.

Unit testing Visual Basic .NET Core libraries using dotnet test and MSTest

9/19/2019 • 4 minutes to read • Edit Online

This tutorial takes you through an interactive experience building a sample solution step-by-step to learn unit testing concepts. If you prefer to follow the tutorial using a pre-built solution, view or download the sample code before you begin. For download instructions, see Samples and Tutorials.

This article is about testing a .NET Core project. If you're testing an **ASP.NET Core** project, see Integration tests in ASP.NET Core.

Creating the source project

Open a shell window. Create a directory called *unit-testing-vb-mstest* to hold the solution. Inside this new directory, run `dotnet new sln` to create a new solution. This practice makes it easier to manage both the class library and the unit test project. Inside the solution directory, create a *PrimeService* directory. You have the following directory and file structure thus far:

```
/unit-testing-vb-mstest
    unit-testing-vb-mstest.sln
    /PrimeService
```

Make *PrimeService* the current directory and run `dotnet new classlib -lang VB` to create the source project. Rename *Class1.VB* to *PrimeService.VB*. You create a failing implementation of the `PrimeService` class:

```
Imports System

Namespace Prime.Services
    Public Class PrimeService
        Public Function IsPrime(candidate As Integer) As Boolean
            Throw New NotImplementedException("Please create a test first")
        End Function
    End Class
End Namespace
```

Change the directory back to the *unit-testing-vb-using-mstest* directory. Run `dotnet sln add .\PrimeService\PrimeService.vbproj` to add the class library project to the solution.

Creating the test project

Next, create the *PrimeService.Tests* directory. The following outline shows the directory structure:

```
/unit-testing-vb-mstest
    unit-testing-vb-mstest.sln
    /PrimeService
        Source Files
        PrimeService.vbproj
    /PrimeService.Tests
```

Make the *PrimeService.Tests* directory the current directory and create a new project using `dotnet new mstest -lang VB`. This command creates a test project that uses MSTest as the test library. The generated template configures the test runner in the *PrimeServiceTests.vbproj*:

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.5.0" />
  <PackageReference Include="MSTest.TestAdapter" Version="1.1.18" />
  <PackageReference Include="MSTest.TestFramework" Version="1.1.18" />
</ItemGroup>
```

The test project requires other packages to create and run unit tests. `dotnet new` in the previous step added MSTest and the MSTest runner. Now, add the `PrimeService` class library as another dependency to the project. Use the `dotnet add reference` command:

```
dotnet add reference ../PrimeService/PrimeService.vbproj
```

You can see the entire file in the samples repository on GitHub.

You have the following final solution layout:

```
/unit-testing-vb-mstest
    unit-testing-vb-mstest.sln
    /PrimeService
        Source Files
        PrimeService.vbproj
    /PrimeService.Tests
        Test Source Files
        PrimeServiceTests.vbproj
```

Execute `dotnet sln add .\PrimeService.Tests\PrimeService.Tests.vbproj` in the *unit-testing-vb-mstest* directory.

Creating the first test

You write one failing test, make it pass, then repeat the process. Remove *UnitTest1.vb* from the *PrimeService.Tests* directory and create a new Visual Basic file named *PrimeService_IsPrimeShould.VB*. Add the following code:

```
Imports Microsoft.VisualStudio.TestTools.UnitTesting

Namespace PrimeService.Tests
    <TestClass>
    Public Class PrimeService_IsPrimeShould
        Private _primeService As Prime.Services.PrimeService = New Prime.Services.PrimeService()

        <TestMethod>
        Sub IsPrime_InputIs1_ReturnFalse()
            Dim result As Boolean = _primeService.IsPrime(1)

            Assert.IsFalse(result, "1 should not be prime")
        End Sub

    End Class
End Namespace
```

The `<TestClass>` attribute indicates a class that contains tests. The `<TestMethod>` attribute denotes a method that is run by the test runner. From the *unit-testing-vb-mstest*, execute `dotnet test` to build the tests and the class library and then run the tests. The MSTest test runner contains the program entry point to run your tests. `dotnet test` starts the test runner using the unit test project you've created.

Your test fails. You haven't created the implementation yet. Make this test pass by writing the simplest code in the

`PrimeService` class that works:

```
Public Function IsPrime(candidate As Integer) As Boolean
    If candidate = 1 Then
        Return False
    End If
    Throw New NotImplementedException("Please create a test first.")
End Function
```

In the *unit-testing-vb-mstest* directory, run `dotnet test` again. The `dotnet test` command runs a build for the `PrimeService` project and then for the `PrimeService.Tests` project. After building both projects, it runs this single test. It passes.

Adding more features

Now that you've made one test pass, it's time to write more. There are a few other simple cases for prime numbers: 0, -1. You could add those cases as new tests with the `<TestMethod>` attribute, but that quickly becomes tedious. There are other MSTest attributes that enable you to write a suite of similar tests. A `<DataTestMethod>` attribute represents a suite of tests that execute the same code but have different input arguments. You can use the `<DataRow>` attribute to specify values for those inputs.

Instead of creating new tests, apply these two attributes to create a single theory. The theory is a method that tests several values less than two, which is the lowest prime number:

```
    <TestClass>
    Public Class PrimeService_IsPrimeShould
        Private _primeService As Prime.Services.PrimeService = New Prime.Services.PrimeService()

        <DataTestMethod>
        <DataRow(-1)>
        <DataRow(0)>
        <DataRow(1)>
        Sub IsPrime_ValuesLessThan2_ReturnFalse(value As Integer)
            Dim result As Boolean = _primeService.IsPrime(value)

            Assert.IsFalse(result, $"{value} should not be prime")
        End Sub

        <DataTestMethod>
        <DataRow(2)>
        <DataRow(3)>
        <DataRow(5)>
        <DataRow(7)>
        Public Sub IsPrime_PrimesLessThan10_ReturnTrue(value As Integer)
            Dim result As Boolean = _primeService.IsPrime(value)

            Assert.IsTrue(result, $"{value} should be prime")
        End Sub

        <DataTestMethod>
        <DataRow(4)>
        <DataRow(6)>
        <DataRow(8)>
        <DataRow(9)>
        Public Sub IsPrime_NonPrimesLessThan10_ReturnFalse(value As Integer)
            Dim result As Boolean = _primeService.IsPrime(value)

            Assert.IsFalse(result, $"{value} should not be prime")
        End Sub
    End Class
```

Run `dotnet test`, and two of these tests fail. To make all of the tests pass, change the `if` clause at the beginning of the method:

```
if candidate < 2
```

Continue to iterate by adding more tests, more theories, and more code in the main library. You have the finished version of the tests and the complete implementation of the library.

You've built a small library and a set of unit tests for that library. You've structured the solution so that adding new packages and tests is part of the normal workflow. You've concentrated most of your time and effort on solving the goals of the application.

Running selective unit tests

3/1/2019 • 2 minutes to read • Edit Online

With the `dotnet test` command in .NET Core, you can use a filter expression to run selective tests. This article demonstrates how to filter which test are run. The following examples use `dotnet test`. If you're using `vstest.console.exe`, replace `--filter` with `--testcasefilter:`.

MSTest

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace MSTestNamespace
{
    [TestClass]
    public class UnitTest1
    {
        [TestCategory("CategoryA")]
        [Priority(1)]
        [TestMethod]
        public void TestMethod1()
        {
        }

        [Priority(2)]
        [TestMethod]
        public void TestMethod2()
        {
        }
    }
}
```

EXPRESSION	RESULT
`dotnet test --filter Method`	Runs tests whose `FullyQualifiedName` contains `Method`. Available in `vstest 15.1+`.
`dotnet test --filter Name~TestMethod1`	Runs tests whose name contains `TestMethod1`.
`dotnet test --filter ClassName=MSTestNamespace.UnitTest1`	Runs tests which are in class `MSTestNamespace.UnitTest1`. **Note:** The `ClassName` value should have a namespace, so `ClassName=UnitTest1` won't work.
`dotnet test --filter FullyQualifiedName!=MSTestNamespace.UnitTest1.TestMethod1`	Runs all tests except `MSTestNamespace.UnitTest1.TestMethod1`.
`dotnet test --filter TestCategory=CategoryA`	Runs tests which are annotated with `[TestCategory("CategoryA")]`.
`dotnet test --filter Priority=2`	Runs tests which are annotated with `[Priority(2)]`.

Using conditional operators | and &

EXPRESSION	RESULT
`dotnet test --filter "FullyQualifiedName~UnitTest1|TestCategory=CategoryA"`	Runs tests which have `UnitTest1` in `FullyQualifiedName` **or** `TestCategory` is `CategoryA`.
`dotnet test --filter "FullyQualifiedName~UnitTest1&TestCategory=CategoryA"`	Runs tests which have `UnitTest1` in `FullyQualifiedName` **and** `TestCategory` is `CategoryA`.
`dotnet test --filter "(FullyQualifiedName~UnitTest1&TestCategory=CategoryA)|Priority=1"`	Runs tests which have either `FullyQualifiedName` containing `UnitTest1` **and** `TestCategory` is `CategoryA` **or** `Priority` is 1.

xUnit

```
using Xunit;

namespace XUnitNamespace
{
    public class TestClass1
    {
        [Trait("Category", "CategoryA")]
        [Trait("Priority", "1")]
        [Fact]
        public void Test1()
        {
        }

        [Trait("Priority", "2")]
        [Fact]
        public void Test2()
        {
        }
    }
}
```

EXPRESSION	RESULT
`dotnet test --filter DisplayName=XUnitNamespace.TestClass1.Test1`	Runs only one test, `XUnitNamespace.TestClass1.Test1`.
`dotnet test --filter FullyQualifiedName!=XUnitNamespace.TestClass1.Test1`	Runs all tests except `XUnitNamespace.TestClass1.Test1`.
`dotnet test --filter DisplayName~TestClass1`	Runs tests whose display name contains `TestClass1`.

In the code example, the defined traits with keys `Category` and `Priority` can be used for filtering.

EXPRESSION	RESULT
`dotnet test --filter XUnit`	Runs tests whose `FullyQualifiedName` contains `XUnit`. Available in `vstest 15.1+`.
`dotnet test --filter Category=CategoryA`	Runs tests which have `[Trait("Category", "CategoryA")]`.

Using conditional operators | and &

EXPRESSION	RESULT
`dotnet test --filter "FullyQualifiedName~TestClass1|Category=CategoryA"`	Runs tests which has `TestClass1` in `FullyQualifiedName` **or** `Category` is `CategoryA`.
`dotnet test --filter "FullyQualifiedName~TestClass1&Category=CategoryA"`	Runs tests which has `TestClass1` in `FullyQualifiedName` **and** `Category` is `CategoryA`.
`dotnet test --filter "(FullyQualifiedName~TestClass1&Category=CategoryA)|Priority=1"`	Runs tests which have either `FullyQualifiedName` containing `TestClass1` **and** `Category` is `CategoryA` **or** `Priority` is 1.

NUnit

```
using NUnit.Framework;

namespace NUnitNamespace
{
    public class UnitTest1
    {
        [Category("CategoryA")]
        [Property("Priority", 1)]
        [Test]
        public void TestMethod1()
        {
        }

        [Property("Priority", 2)]
        [Test]
        public void TestMethod2()
        {
        }
    }
}
```

EXPRESSION	RESULT
`dotnet test --filter Method`	Runs tests whose `FullyQualifiedName` contains `Method`. Available in `vstest 15.1+`.
`dotnet test --filter Name~TestMethod1`	Runs tests whose name contains `TestMethod1`.
`dotnet test --filter FullyQualifiedName~NUnitNamespace.UnitTest1`	Runs tests which are in class `NUnitNamespace.UnitTest1`.
`dotnet test --filter FullyQualifiedName!=NUnitNamespace.UnitTest1.TestMethod1`	Runs all tests except `NUnitNamespace.UnitTest1.TestMethod1`.
`dotnet test --filter TestCategory=CategoryA`	Runs tests which are annotated with `[Category("CategoryA")]`.
`dotnet test --filter Priority=2`	Runs tests which are annotated with `[Priority(2)]`.

Using conditional operators | and &

EXPRESSION	RESULT
`dotnet test --filter "FullyQualifiedName~UnitTest1|TestCategory=CategoryA"`	Runs tests which have `UnitTest1` in `FullyQualifiedName` **or** `TestCategory` is `CategoryA`.
`dotnet test --filter "FullyQualifiedName~UnitTest1&TestCategory=CategoryA"`	Runs tests which have `UnitTest1` in `FullyQualifiedName` **and** `TestCategory` is `CategoryA`.
`dotnet test --filter "(FullyQualifiedName~UnitTest1&TestCategory=CategoryA)|Priority=1"`	Runs tests which have either `FullyQualifiedName` containing `UnitTest1` **and** `TestCategory` is `CategoryA` **or** `Priority` is 1.

Test published output with dotnet vstest

10/22/2019 • 2 minutes to read • Edit Online

You can run tests on already published output by using the `dotnet vstest` command. This will work on xUnit, MSTest, and NUnit tests. Simply locate the DLL file that was part of your published output and run:

```
dotnet vstest <MyPublishedTests>.dll
```

Where `<MyPublishedTests>` is the name of your published test project.

Example

The commands below demonstrate running tests on a published DLL.

```
dotnet new mstest -o MyProject.Tests
cd MyProject.Tests
dotnet publish -o out
dotnet vstest out/MyProject.Tests.dll
```

NOTE

Note: If your app targets a framework other than `netcoreapp`, you can still run the `dotnet vstest` command by passing in the targeted framework with a framework flag. For example, `dotnet vstest <MyPublishedTests>.dll --Framework:".NETFramework,Version=v4.6"`. In Visual Studio 2017 Update 5 and later, the desired framework is automatically detected.

See also

- Unit Testing with dotnet test and xUnit
- Unit Testing with dotnet test and NUnit
- Unit Testing with dotnet test and MSTest

Using .NET Core SDK and tools in Continuous Integration (CI)

5/29/2019 • 8 minutes to read • Edit Online

This document outlines using the .NET Core SDK and its tools on a build server. The .NET Core toolset works both interactively, where a developer types commands at a command prompt, and automatically, where a Continuous Integration (CI) server runs a build script. The commands, options, inputs, and outputs are the same, and the only things you supply are a way to acquire the tooling and a system to build your app. This document focuses on scenarios of tool acquisition for CI with recommendations on how to design and structure your build scripts.

Installation options for CI build servers

Using the native installers

Native installers are available for macOS, Linux, and Windows. The installers require admin (sudo) access to the build server. The advantage of using a native installer is that it installs all of the native dependencies required for the tooling to run. Native installers also provide a system-wide installation of the SDK.

macOS users should use the PKG installers. On Linux, there's a choice of using a feed-based package manager, such as apt-get for Ubuntu or yum for CentOS, or using the packages themselves, DEB or RPM. On Windows, use the MSI installer.

The latest stable binaries are found at .NET downloads. If you wish to use the latest (and potentially unstable) pre-release tooling, use the links provided at the dotnet/core-sdk GitHub repository. For Linux distributions, `tar.gz` archives (also known as `tarballs`) are available; use the installation scripts within the archives to install .NET Core.

Using the installer script

Using the installer script allows for non-administrative installation on your build server and easy automation for obtaining the tooling. The script takes care of downloading the tooling and extracting it into a default or specified location for use. You can also specify a version of the tooling that you wish to install and whether you want to install the entire SDK or only the shared runtime.

The installer script is automated to run at the start of the build to fetch and install the desired version of the SDK. The *desired version* is whatever version of the SDK your projects require to build. The script allows you to install the SDK in a local directory on the server, run the tools from the installed location, and then clean up (or let the CI service clean up) after the build. This provides encapsulation and isolation to your entire build process. The installation script reference is found in the dotnet-install article.

NOTE

Azure DevOps Services

When using the installer script, native dependencies aren't installed automatically. You must install the native dependencies if the operating system doesn't have them. For more information, see Prerequisites for .NET Core on Linux.

CI setup examples

This section describes a manual setup using a PowerShell or bash script, along with a description of several software as a service (SaaS) CI solutions. The SaaS CI solutions covered are Travis CI, AppVeyor, and Azure Pipelines.

Manual setup

Each SaaS service has its own methods for creating and configuring a build process. If you use different SaaS solution than those listed or require customization beyond the pre-packaged support, you must perform at least some manual configuration.

In general, a manual setup requires you to acquire a version of the tools (or the latest nightly builds of the tools) and run your build script. You can use a PowerShell or bash script to orchestrate the .NET Core commands or use a project file that outlines the build process. The orchestration section provides more detail on these options.

After you create a script that performs a manual CI build server setup, use it on your dev machine to build your code locally for testing purposes. Once you confirm that the script is running well locally, deploy it to your CI build server. A relatively simple PowerShell script demonstrates how to obtain the .NET Core SDK and install it on a Windows build server:

```
$ErrorActionPreference="Stop"
$ProgressPreference="SilentlyContinue"

# $LocalDotnet is the path to the locally-installed SDK to ensure the
#   correct version of the tools are executed.
$LocalDotnet=""
# $InstallDir and $CliVersion variables can come from options to the
#   script.
$InstallDir = "./cli-tools"
$CliVersion = "1.0.1"

# Test the path provided by $InstallDir to confirm it exists. If it
#   does, it's removed. This is not strictly required, but it's a
#   good way to reset the environment.
if (Test-Path $InstallDir)
{
    rm -Recurse $InstallDir
}
New-Item -Type "directory" -Path $InstallDir

Write-Host "Downloading the CLI installer..."

# Use the Invoke-WebRequest PowerShell cmdlet to obtain the
#   installation script and save it into the installation directory.
Invoke-WebRequest `
    -Uri "https://dot.net/v1/dotnet-install.ps1" `
    -OutFile "$InstallDir/dotnet-install.ps1"

Write-Host "Installing the CLI requested version ($CliVersion) ..."

# Install the SDK of the version specified in $CliVersion into the
#   specified location ($InstallDir).
& $InstallDir/dotnet-install.ps1 -Version $CliVersion `
    -InstallDir $InstallDir

Write-Host "Downloading and installation of the SDK is complete."

# $LocalDotnet holds the path to dotnet.exe for future use by the
#   script.
$LocalDotnet = "$InstallDir/dotnet"

# Run the build process now. Implement your build script here.
```

You provide the implementation for your build process at the end of the script. The script acquires the tools and then executes your build process. For UNIX machines, the following bash script performs the actions described in the PowerShell script in a similar manner:

```
#!/bin/bash
INSTALLDIR="cli-tools"
CLI_VERSION=1.0.1
DOWNLOADER=$(which curl)
if [ -d "$INSTALLDIR" ]
then
    rm -rf "$INSTALLDIR"
fi
mkdir -p "$INSTALLDIR"
echo Downloading the CLI installer.
$DOWNLOADER https://dot.net/v1/dotnet-install.sh > "$INSTALLDIR/dotnet-install.sh"
chmod +x "$INSTALLDIR/dotnet-install.sh"
echo Installing the CLI requested version $CLI_VERSION. Please wait, installation may take a few minutes.
"$INSTALLDIR/dotnet-install.sh" --install-dir "$INSTALLDIR" --version $CLI_VERSION
if [ $? -ne 0 ]
then
    echo Download of $CLI_VERSION version of the CLI failed. Exiting now.
    exit 0
fi
echo The CLI has been installed.
LOCALDOTNET="$INSTALLDIR/dotnet"
# Run the build process now. Implement your build script here.
```

Travis CI

You can configure Travis CI to install the .NET Core SDK using the `csharp` language and the `dotnet` key. For more information, see the official Travis CI docs on Building a C#, F#, or Visual Basic Project. Note as you access the Travis CI information that the community-maintained `language: csharp` language identifier works for all .NET languages, including F#, and Mono.

Travis CI runs both macOS and Linux jobs in a *build matrix*, where you specify a combination of runtime, environment, and exclusions/inclusions to cover your build combinations for your app. For more information, see the Customizing the Build article in the Travis CI documentation. The MSBuild-based tools include the LTS (1.0.x) and Current (1.1.x) runtimes in the package; so by installing the SDK, you receive everything you need to build.

AppVeyor

AppVeyor installs the .NET Core 1.0.1 SDK with the `Visual Studio 2017` build worker image. Other build images with different versions of the .NET Core SDK are available. For more information, see the appveyor.yml example and the Build worker images article in the AppVeyor docs.

The .NET Core SDK binaries are downloaded and unzipped in a subdirectory using the install script, and then they're added to the `PATH` environment variable. Add a build matrix to run integration tests with multiple versions of the .NET Core SDK:

```
environment:
  matrix:
    - CLI_VERSION: 1.0.1
    - CLI_VERSION: Latest

install:
  # See appveyor.yml example for install script
```

Azure DevOps Services

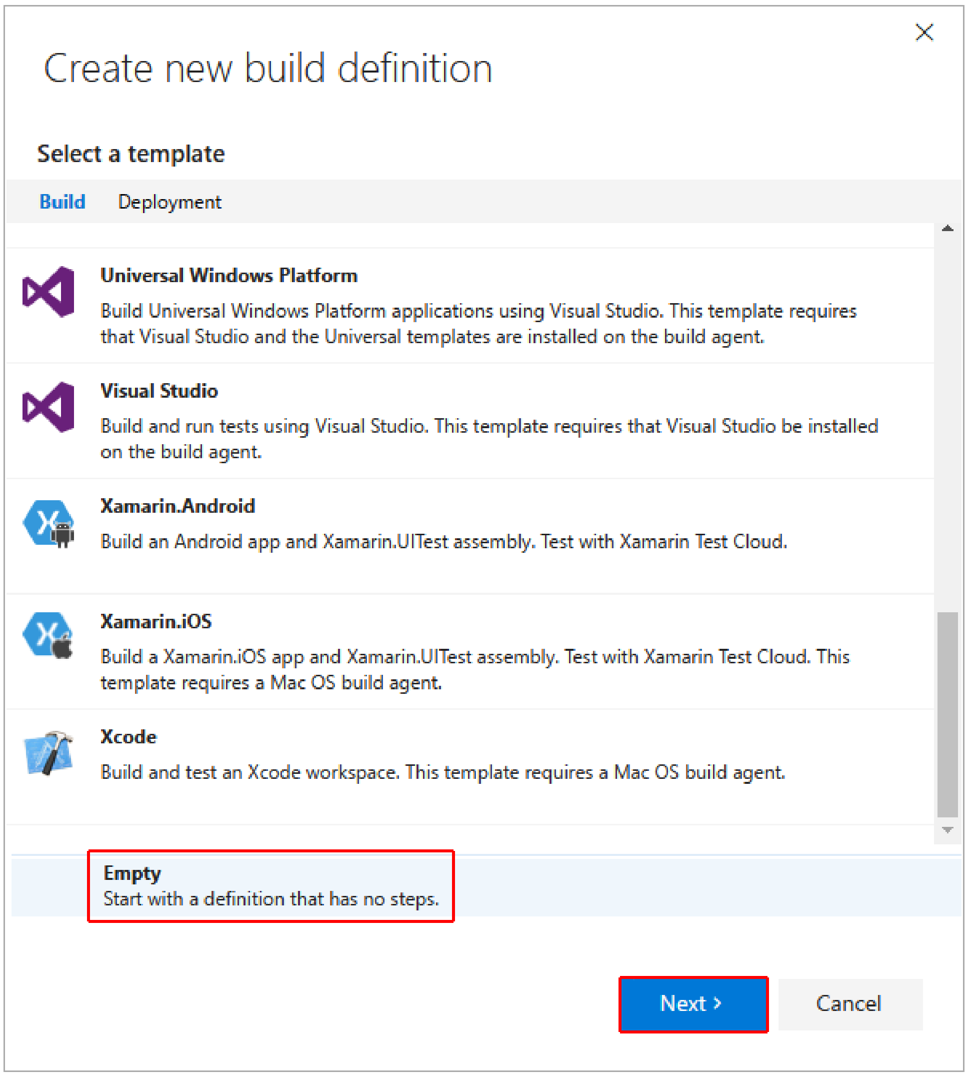
Configure Azure DevOps Services to build .NET Core projects using one of these approaches:

1. Run the script from the manual setup step using your commands.
2. Create a build composed of several Azure DevOps Services built-in build tasks that are configured to use .NET Core tools.

Both solutions are valid. Using a manual setup script, you control the version of the tools that you receive, since you download them as part of the build. The build is run from a script that you must create. This article only covers the manual option. For more information on composing a build with Azure DevOps Services build tasks, see the Azure Pipelines documentation.

To use a manual setup script in Azure DevOps Services, create a new build definition and specify the script to run for the build step. This is accomplished using the Azure DevOps Services user interface:

1. Start by creating a new build definition. Once you reach the screen that provides you an option to define what kind of a build you wish to create, select the **Empty** option.



Create new build definition

Select a template

Build | Deployment

Universal Windows Platform
Build Universal Windows Platform applications using Visual Studio. This template requires that Visual Studio and the Universal templates are installed on the build agent.

Visual Studio
Build and run tests using Visual Studio. This template requires that Visual Studio be installed on the build agent.

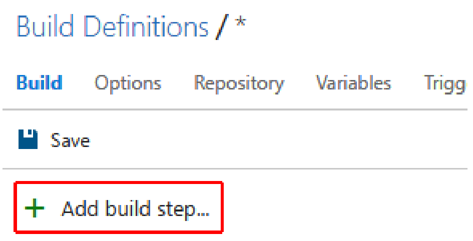
Xamarin.Android
Build an Android app and Xamarin.UITest assembly. Test with Xamarin Test Cloud.

Xamarin.iOS
Build a Xamarin.iOS app and Xamarin.UITest assembly. Test with Xamarin Test Cloud. This template requires a Mac OS build agent.

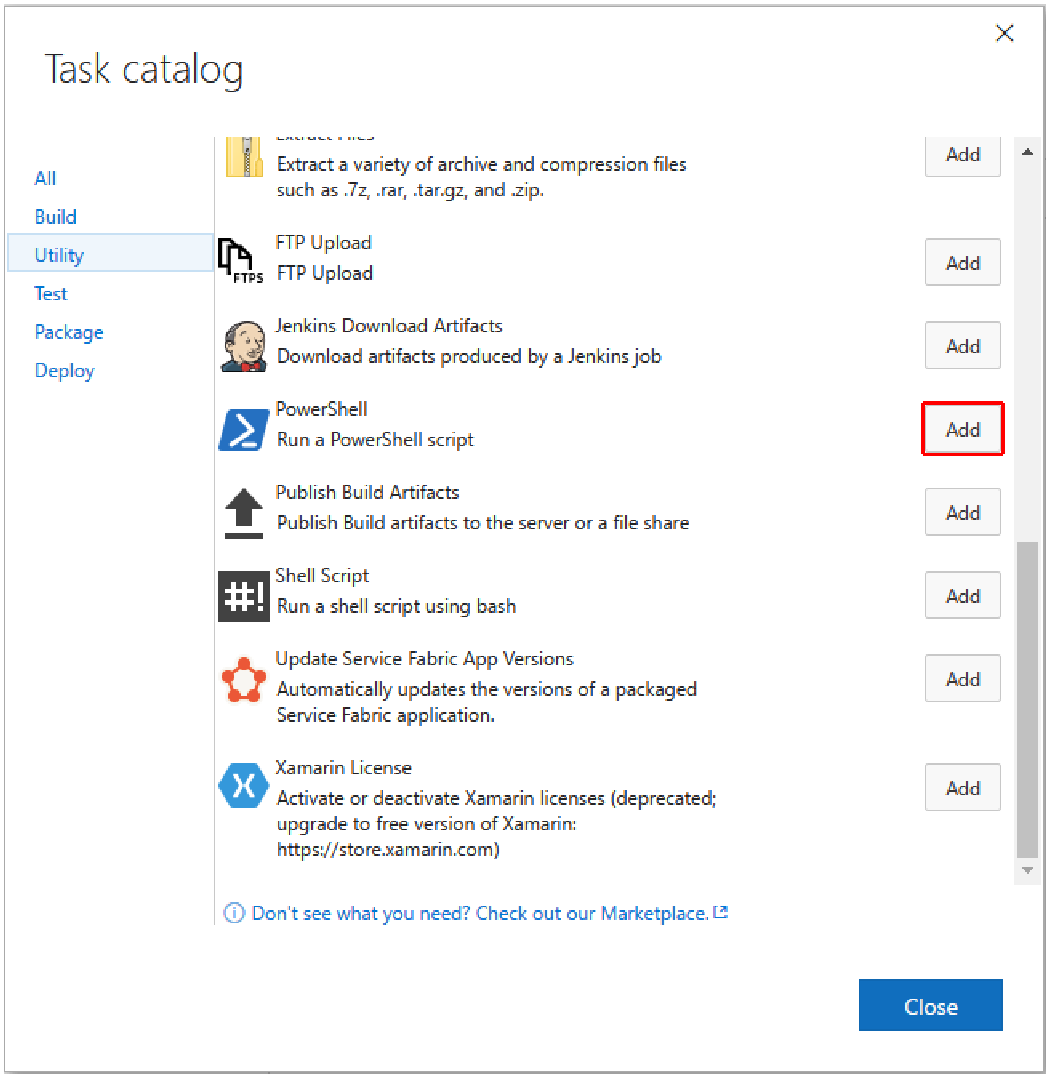
Xcode
Build and test an Xcode workspace. This template requires a Mac OS build agent.

Empty
Start with a definition that has no steps.

2. After configuring the repository to build, you're directed to the build definitions. Select **Add build step**:



3. You're presented with the **Task catalog**. The catalog contains tasks that you use in the build. Since you have a script, select the **Add** button for **PowerShell: Run a PowerShell script**.


Task catalog

All
Build
Utility
Test
Package
Deploy

Extract Files
Extract a variety of archive and compression files such as .7z, .rar, .tar.gz, and .zip.

FTP Upload
FTP Upload

Jenkins Download Artifacts
Download artifacts produced by a Jenkins job

PowerShell
Run a PowerShell script

Publish Build Artifacts
Publish Build artifacts to the server or a file share

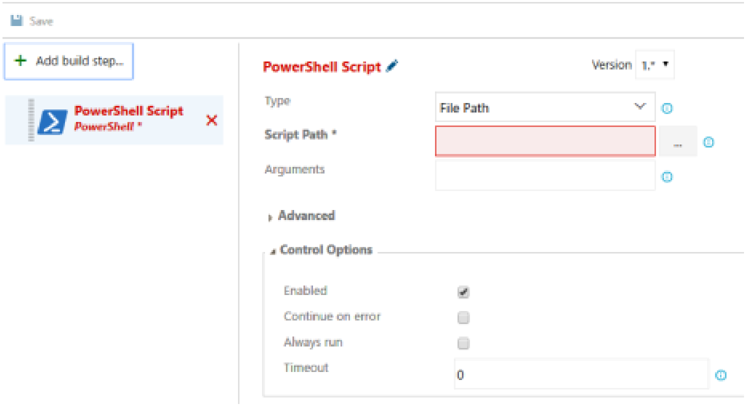
Shell Script
Run a shell script using bash

Update Service Fabric App Versions
Automatically updates the versions of a packaged Service Fabric application.

Xamarin License
Activate or deactivate Xamarin licenses (deprecated; upgrade to free version of Xamarin: https://store.xamarin.com)

Don't see what you need? Check out our Marketplace.

4. Configure the build step. Add the script from the repository that you're building:



PowerShell Script
Version 1.*
Type: File Path
Script Path *:
Arguments:
Enabled ✓
Continue on error
Always run
Timeout 0

Orchestrating the build

Most of this document describes how to acquire the .NET Core tools and configure various CI services without providing information on how to orchestrate, or *actually build*, your code with .NET Core. The choices on how to structure the build process depend on many factors that can't be covered in a general way here. For more information on orchestrating your builds with each technology, explore the resources and samples provided in the documentation sets of Travis CI, AppVeyor, and Azure Pipelines.

Two general approaches that you take in structuring the build process for .NET Core code using the .NET Core tools are using MSBuild directly or using the .NET Core command-line commands. Which approach you should take is determined by your comfort level with the approaches and trade-offs in complexity. MSBuild provides you the ability to express your build process as tasks and targets, but it comes with the added complexity of learning

MSBuild project file syntax. Using the .NET Core command-line tools is perhaps simpler, but it requires you to write orchestration logic in a scripting language like `bash` or PowerShell.

See also

- .NET downloads - Linux

Overview of how .NET Core is versioned

10/9/2019 • 4 minutes to read • Edit Online

.NET Core refers to the .NET Core Runtime and the .NET Core SDK, which contains the tools you need to develop applications. .NET Core SDKs are designed to work with any previous version of the .NET Core Runtime. This article explains the runtime and the SDK version strategy. An explanation of version numbers for .NET Standard can be found in the article introducing .NET Standard.

The .NET Core Runtime and .NET Core SDK add new features at a different rate - in general the .NET Core SDK provides updated tools more quickly than the .NET Core Runtime changes the runtime you use in production.

Versioning details

".NET Core 2.1" refers to the .NET Core Runtime version number. The .NET Core Runtime has a major/minor/patch approach to versioning that follows semantic versioning.

The .NET Core SDK doesn't follow semantic versioning. The .NET Core SDK releases faster and its versions must communicate both the aligned runtime and the SDK's own minor and patch releases. The first two positions of the .NET Core SDK version are locked to the .NET Core Runtime it released with. Each version of the SDK can create applications for this runtime or any lower version.

The third position of the SDK version number communicates both the minor and patch number. The minor version is multiplied by 100. Minor version 1, patch version 2 would be represented as 102. The final two digits represent the patch number. For example, the release of .NET Core 2.2 may create releases like the following table:

CHANGE	.NET CORE RUNTIME	.NET CORE SDK (*)
Initial release	2.2.0	2.2.100
SDK Patch	2.2.0	2.2.101
Runtime and SDK Patch	2.2.1	2.2.102
SDK Feature change	2.2.1	2.2.200

(*) This chart uses a future 2.2 .NET Core Runtime as the example because a historic artifact meant the first SDK for .NET Core 2.1 is 2.1.300. For more information, See the .NET Core version selection.

NOTES:

- If the SDK has 10 feature updates before a runtime feature update, version numbers roll into the 1000 series with numbers like 2.2.1000 as the feature release following 2.2.900. This situation isn't expected to occur.
- 99 patch releases without a feature release won't occur. If a release approaches this number, it forces a feature release.

You can see more details in the initial proposal at the dotnet/designs repository.

Semantic versioning

The .NET Core *Runtime* roughly adheres to Semantic Versioning (SemVer), adopting the use of `MAJOR.MINOR.PATCH` versioning, using the various parts of the version number to describe the degree and type of change.

```
MAJOR.MINOR.PATCH[-PRERELEASE-BUILDNUMBER]
```

The optional `PRERELEASE` and `BUILDNUMBER` parts are never part of supported releases and only exist on nightly builds, local builds from source targets, and unsupported preview releases.

Understand runtime version number changes

`MAJOR` is incremented when:

- Significant changes occur to the product, or a new product direction.
- Breaking changes were taken. There's a high bar to accepting breaking changes.
- An old version is no longer supported.
- A newer `MAJOR` version of an existing dependency is adopted.

`MINOR` is incremented when:

- Public API surface area is added.
- A new behavior is added.
- A newer `MINOR` version of an existing dependency is adopted.
- A new dependency is introduced.

`PATCH` is incremented when:

- Bug fixes are made.
- Support for a newer platform is added.
- A newer `PATCH` version of an existing dependency is adopted.
- Any other change doesn't fit one of the previous cases.

When there are multiple changes, the highest element affected by individual changes is incremented, and the following ones are reset to zero. For example, when `MAJOR` is incremented, `MINOR` and `PATCH` are reset to zero. When `MINOR` is incremented, `PATCH` is reset to zero while `MAJOR` is left untouched.

Version numbers in file names

The files downloaded for .NET Core carry the version, for example, `dotnet-sdk-2.1.300-win10-x64.exe`.

Preview versions

Preview versions have a `-preview[number]-([build]|"final")` appended to the version. For example, `2.0.0-preview1-final`.

Servicing versions

After a release goes out, the release branches generally stop producing daily builds and instead start producing servicing builds. Servicing versions have a `-servicing-[number]` appended to the version. For example, `2.0.1-servicing-006924`.

Relationship to .NET Standard versions

.NET Standard consists of a .NET reference assembly. There are multiple implementations specific to each platform. The reference assembly contains the definition of .NET APIs which are part of a given .NET Standard version. Each implementation fulfills the .NET Standard contract on the specific platform. You can learn more about .NET Standard in the article on .NET Standard in the .NET Guide.

The .NET Standard reference assembly uses a `MAJOR.MINOR` versioning scheme. `PATCH` level isn't useful for .NET Standard because it exposes only an API specification (no implementation) and by definition any change to the

API would represent a change in the feature set, and thus a new `MINOR` version.

The implementations on each platform may be updated, typically as part of the platform release, and thus not evident to the programmers using .NET Standard on that platform.

Each version of .NET Core implements a version of .NET Standard. Implementing a version of .NET Standard implies support for previous versions of .NET Standard. .NET Standard and .NET Core version independently. It's a coincidence that .NET Core 2.0 implements .NET Standard 2.0. .NET Core 2.1 also implements .NET Standard 2.0. .NET Core will support future versions of .NET Standard as they become available.

.NET CORE	.NET STANDARD
1.0	up to 1.6
2.0	up to 2.0
2.1	up to 2.0
2.2	up to 2.0
3.0	up to 2.1

See also

- Target frameworks
- .NET Core distribution packaging
- .NET Core Support Lifecycle Fact Sheet
- .NET Core 2+ Version Binding
- Docker images for .NET Core

Select the .NET Core version to use

11/3/2019 • 5 minutes to read • Edit Online

This article explains the policies used by the .NET Core tools, SDK, and runtime for selecting versions. These policies provide a balance between running applications using the specified versions and enabling ease of upgrading both developer and end-user machines. These policies perform the following actions:

- Easy and efficient deployment of .NET Core, including security and reliability updates.
- Use the latest tools and commands independent of target runtime.

Version selection occurs:

- When you run an SDK command, the SDK uses the latest installed version.
- When you build an assembly, target framework monikers define build time APIs.
- When you run a .NET Core application, target framework dependent apps roll forward.
- When you publish a self-contained application, self-contained deployments include the selected runtime.

The rest of this document examines those four scenarios.

The SDK uses the latest installed version

SDK commands include `dotnet new` and `dotnet run`. The .NET Core CLI must choose an SDK version for every `dotnet` command. It uses the latest SDK installed on the machine by default, even if:

- The project targets an earlier version of the .NET Core runtime.
- The latest version of the .NET Core SDK is a preview version.

You can take advantage of the latest SDK features and improvements while targeting earlier .NET Core runtime versions. You can target multiple runtime versions of .NET Core on different projects, using the same SDK tools for all projects.

On rare occasions, you may need to use an earlier version of the SDK. You specify that version in a *global.json* file. The "use latest" policy means you only use *global.json* to specify a .NET Core SDK version earlier than the latest installed version.

global.json can be placed anywhere in the file hierarchy. The CLI searches upward from the project directory for the first *global.json* it finds. You control which projects a given *global.json* applies to by its place in the file system. The .NET CLI searches for a *global.json* file iteratively navigating the path upward from the current working directory. The first *global.json* file found specifies the version used. If that SDK version is installed, that version is used. If the SDK specified in the *global.json* is not found, the .NET CLI uses matching rules to select a compatible SDK, or fails if none is found.

The following example shows the *global.json* syntax:

```
{
  "sdk": {
    "version": "2.0.0"
  }
}
```

The process for selecting an SDK version is:

1. `dotnet` searches for a *global.json* file iteratively reverse-navigating the path upward from the current working

directory.
2. `dotnet` uses the SDK specified in the first *global.json* found.
3. `dotnet` uses the latest installed SDK if no *global.json* is found.

You can learn more about selecting an SDK version in the Matching rules section of the article on *global.json*.

Target Framework Monikers define build time APIs

You build your project against APIs defined in a **Target Framework Moniker** (TFM). You specify the target framework in the project file. Set the `TargetFramework` element in your project file as shown in the following example:

```
<TargetFramework>netcoreapp2.0</TargetFramework>
```

You may build your project against multiple TFMs. Setting multiple target frameworks is more common for libraries but can be done with applications as well. You specify a `TargetFrameworks` property (plural of `TargetFramework`). The target frameworks are semicolon-delimited as shown in the following example:

```
<TargetFrameworks>netcoreapp2.0;net47</TargetFrameworks>
```

A given SDK supports a fixed set of frameworks, capped to the target framework of the runtime it ships with. For example, the .NET Core 2.0 SDK includes the .NET Core 2.0 runtime, which is an implementation of the `netcoreapp2.0` target framework. The .NET Core 2.0 SDK supports `netcoreapp1.0`, `netcoreapp1.1`, and `netcoreapp2.0` but not `netcoreapp2.1` (or higher). You install the .NET Core 2.1 SDK to build for `netcoreapp2.1`.

.NET Standard target frameworks are also capped to the target framework of the runtime the SDK ships with. The .NET Core 2.0 SDK is capped to `netstandard2.0`.

Framework-dependent apps roll forward

When you run an application from source with `dotnet run`, from a **framework-dependent deployment** with `dotnet myapp.dll`, or from a **framework-dependent executable** with `myapp.exe`, the `dotnet` executable is the **host** for the application.

The host chooses the latest patch version installed on the machine. For example, if you specified `netcoreapp2.0` in your project file, and `2.0.4` is the latest .NET runtime installed, the `2.0.4` runtime is used.

If no acceptable `2.0.*` version is found, a new `2.*` version is used. For example, if you specified `netcoreapp2.0` and only `2.1.0` is installed, the application runs using the `2.1.0` runtime. This behavior is referred to as "minor version roll-forward." Lower versions also won't be considered. When no acceptable runtime is installed, the application won't run.

A few usage examples demonstrate the behavior, if you target 2.0:

- 2.0 is specified. 2.0.5 is the highest patch version installed. 2.0.5 is used.
- 2.0 is specified. No 2.0.* versions are installed. 1.1.1 is the highest runtime installed. An error message is displayed.
- 2.0 is specified. No 2.0.* versions are installed. 2.2.2 is the highest 2.x runtime version installed. 2.2.2 is used.
- 2.0 is specified. No 2.x versions are installed. 3.0.0 is installed. An error message is displayed.

Minor version roll-forward has one side-effect that may affect end users. Consider the following scenario:

1. The application specifies that 2.0 is required.
2. When run, version 2.0.* is not installed, however, 2.2.2 is. Version 2.2.2 will be used.

3. Later, the user installs 2.0.5 and runs the application again, 2.0.5 will now be used.

It's possible that 2.0.5 and 2.2.2 behave differently, particularly for scenarios like serializing binary data.

Self-contained deployments include the selected runtime

You can publish an application as a **self-contained distribution**. This approach bundles the .NET Core runtime and libraries with your application. Self-contained deployments don't have a dependency on runtime environments. Runtime version selection occurs at publishing time, not run time.

The publishing process selects the latest patch version of the given runtime family. For example, `dotnet publish` will select .NET Core 2.0.4 if it is the latest patch version in the .NET Core 2.0 runtime family. The target framework (including the latest installed security patches) is packaged with the application.

It's an error if the minimum version specified for an application isn't satisfied. `dotnet publish` binds to the latest runtime patch version (within a given major.minor version family). `dotnet publish` doesn't support the roll-forward semantics of `dotnet run`. For more information about patches and self-contained deployments, see the article on runtime patch selection in deploying .NET Core applications.

Self-contained deployments may require a specific patch version. You can override the minimum runtime patch version (to higher or lower versions) in the project file, as shown in the following example:

```
<RuntimeFrameworkVersion>2.0.4</RuntimeFrameworkVersion>
```

The `RuntimeFrameworkVersion` element overrides the default version policy. For self-contained deployments, the `RuntimeFrameworkVersion` specifies the *exact* runtime framework version. For framework-dependent applications, the `RuntimeFrameworkVersion` specifies the *minimum* required runtime framework version.

How to remove the .NET Core Runtime and SDK

9/13/2019 • 5 minutes to read • Edit Online

Over time, as you install updated versions of the .NET Core runtime and SDK, you may want to remove outdated versions of .NET Core from your machine. Removing older versions of the runtime may change the runtime chosen to run shared framework applications, as detailed in the article on .NET Core version selection.

Should I remove a version?

The .NET Core version selection behaviors and the runtime compatibility of .NET Core across updates enables safe removal of previous versions. .NET Core runtime updates are compatible within a major version 'band' such as 1.x and 2.x. Additionally, newer releases of the .NET Core SDK generally maintain the ability to build applications that target previous versions of the runtime in a compatible manner.

In general, you only need the latest SDK and latest patch version of the runtimes required for your application. Instances where retaining older SDK or Runtime versions include maintaining **project.json**-based applications. Unless your application has specific reasons for earlier SDKs or runtimes, you may safely remove older versions.

Determine what is installed

Starting with .NET Core 2.1, the .NET CLI has options you can use to list the versions of the SDK and runtime that are installed on your machine. Use `dotnet --list-sdks` to see the list of SDKs installed on your machine. Use `dotnet --list-runtimes` to see the list of runtimes installed on your machine. The following text shows typical output for Windows, macOS, or Linux:

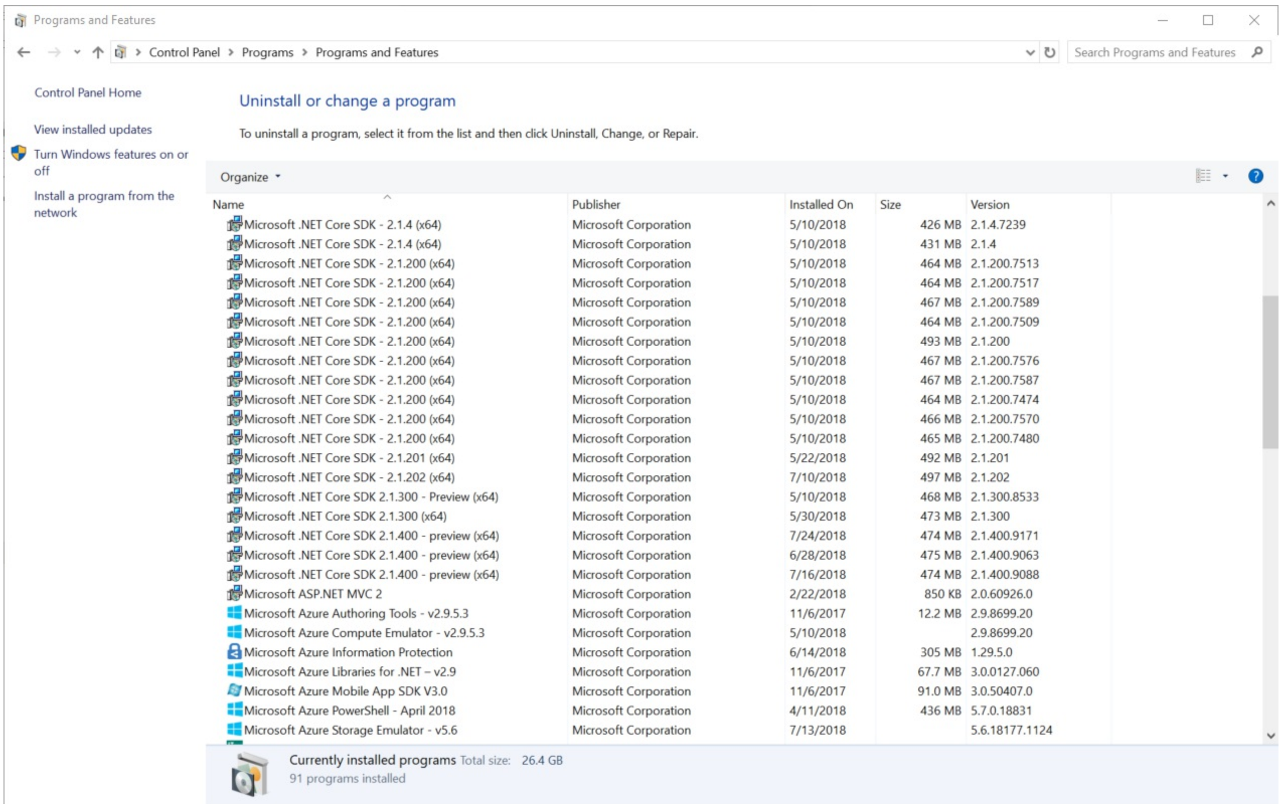
- Windows
- Linux
- macOS

```
C:\> dotnet --list-sdks
2.1.200-preview-007474 [C:\Program Files\dotnet\sdk]
2.1.200-preview-007480 [C:\Program Files\dotnet\sdk]
2.1.200-preview-007509 [C:\Program Files\dotnet\sdk]
2.1.200-preview-007570 [C:\Program Files\dotnet\sdk]
2.1.200-preview-007576 [C:\Program Files\dotnet\sdk]
2.1.200-preview-007587 [C:\Program Files\dotnet\sdk]
2.1.200-preview-007589 [C:\Program Files\dotnet\sdk]
2.1.200 [C:\Program Files\dotnet\sdk]
2.1.201 [C:\Program Files\dotnet\sdk]
2.1.202 [C:\Program Files\dotnet\sdk]
2.1.300-preview2-008533 [C:\Program Files\dotnet\sdk]
2.1.300 [C:\Program Files\dotnet\sdk]
2.1.400-preview-009063 [C:\Program Files\dotnet\sdk]
2.1.400-preview-009088 [C:\Program Files\dotnet\sdk]
2.1.400-preview-009171 [C:\Program Files\dotnet\sdk]

C:\> dotnet --list-runtimes
Microsoft.AspNetCore.All 2.1.0-preview2-final [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.All]
Microsoft.AspNetCore.All 2.1.0 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.All]
Microsoft.AspNetCore.All 2.1.1 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.All]
Microsoft.AspNetCore.All 2.1.2 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.All]
Microsoft.AspNetCore.App 2.1.0-preview2-final [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 2.1.0 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 2.1.1 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 2.1.2 [C:\Program Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.NETCore.App 2.0.6 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 2.0.7 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 2.0.9 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 2.1.0-preview2-26406-04 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 2.1.0 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 2.1.1 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 2.1.2 [C:\Program Files\dotnet\shared\Microsoft.NETCore.App]
```

Uninstalling .NET Core

- Windows
- Linux
- macOS

.NET Core uses the Windows **Add/Remove Programs** dialog to remove versions of the .NET Core runtime and SDK. The following figure shows the **Add/Remove Programs** dialog with several versions of the .NET runtime and SDK installed.


Programs and Features
Control Panel > Programs > Programs and Features
Search Programs and Features
Control Panel Home
View installed updates
Turn Windows features on or off
Install a program from the network
Organize

Uninstall or change a program

To uninstall a program, select it from the list and then click Uninstall, Change, or Repair.

Name	Publisher	Installed On	Size	Version
Microsoft .NET Core SDK - 2.1.4 (x64)	Microsoft Corporation	5/10/2018	426 MB	2.1.4.7239
Microsoft .NET Core SDK - 2.1.4 (x64)	Microsoft Corporation	5/10/2018	431 MB	2.1.4
Microsoft .NET Core SDK - 2.1.200 (x64)	Microsoft Corporation	5/10/2018	464 MB	2.1.200.7513
Microsoft .NET Core SDK - 2.1.200 (x64)	Microsoft Corporation	5/10/2018	464 MB	2.1.200.7517
Microsoft .NET Core SDK - 2.1.200 (x64)	Microsoft Corporation	5/10/2018	467 MB	2.1.200.7589
Microsoft .NET Core SDK - 2.1.200 (x64)	Microsoft Corporation	5/10/2018	464 MB	2.1.200.7509
Microsoft .NET Core SDK - 2.1.200 (x64)	Microsoft Corporation	5/10/2018	493 MB	2.1.200
Microsoft .NET Core SDK - 2.1.200 (x64)	Microsoft Corporation	5/10/2018	467 MB	2.1.200.7576
Microsoft .NET Core SDK - 2.1.200 (x64)	Microsoft Corporation	5/10/2018	467 MB	2.1.200.7587
Microsoft .NET Core SDK - 2.1.200 (x64)	Microsoft Corporation	5/10/2018	464 MB	2.1.200.7474
Microsoft .NET Core SDK - 2.1.200 (x64)	Microsoft Corporation	5/10/2018	466 MB	2.1.200.7570
Microsoft .NET Core SDK - 2.1.200 (x64)	Microsoft Corporation	5/10/2018	465 MB	2.1.200.7480
Microsoft .NET Core SDK - 2.1.201 (x64)	Microsoft Corporation	5/22/2018	492 MB	2.1.201
Microsoft .NET Core SDK - 2.1.202 (x64)	Microsoft Corporation	7/10/2018	497 MB	2.1.202
Microsoft .NET Core SDK 2.1.300 - Preview (x64)	Microsoft Corporation	5/10/2018	468 MB	2.1.300.8533
Microsoft .NET Core SDK 2.1.300 (x64)	Microsoft Corporation	5/30/2018	473 MB	2.1.300
Microsoft .NET Core SDK 2.1.400 - preview (x64)	Microsoft Corporation	7/24/2018	474 MB	2.1.400.9171
Microsoft .NET Core SDK 2.1.400 - preview (x64)	Microsoft Corporation	6/28/2018	475 MB	2.1.400.9063
Microsoft .NET Core SDK 2.1.400 - preview (x64)	Microsoft Corporation	7/16/2018	474 MB	2.1.400.9088
Microsoft ASP.NET MVC 2	Microsoft Corporation	2/22/2018	850 KB	2.0.60926.0
Microsoft Azure Authoring Tools - v2.9.5.3	Microsoft Corporation	11/6/2017	12.2 MB	2.9.8699.20
Microsoft Azure Compute Emulator - v2.9.5.3	Microsoft Corporation	5/10/2018		2.9.8699.20
Microsoft Azure Information Protection	Microsoft Corporation	6/14/2018	305 MB	1.29.5.0
Microsoft Azure Libraries for .NET – v2.9	Microsoft Corporation	11/6/2017	67.7 MB	3.0.0127.060
Microsoft Azure Mobile App SDK V3.0	Microsoft Corporation	11/6/2017	91.0 MB	3.0.50407.0
Microsoft Azure PowerShell - April 2018	Microsoft Corporation	4/11/2018	436 MB	5.7.0.18831
Microsoft Azure Storage Emulator - v5.6	Microsoft Corporation	7/13/2018		5.6.18177.1124

Currently installed programs Total size: 26.4 GB
91 programs installed

Select any versions you want to remove from your machine and click **Uninstall**.

.NET Core RID Catalog

10/30/2019 • 4 minutes to read • Edit Online

RID is short for *Runtime IDentifier*. RID values are used to identify target platforms where the application runs. They're used by .NET packages to represent platform-specific assets in NuGet packages. The following values are examples of RIDs: `linux-x64`, `ubuntu.14.04-x64`, `win7-x64`, or `osx.10.12-x64`. For the packages with native dependencies, the RID designates on which platforms the package can be restored.

A single RID can be set in the `<RuntimeIdentifier>` element of your project file. Multiple RIDs can be defined as a semicolon-delimited list in the project file's `<RuntimeIdentifiers>` element. They're also used via the `--runtime` option with the following .NET Core CLI commands:

- dotnet build
- dotnet clean
- dotnet pack
- dotnet publish
- dotnet restore
- dotnet run
- dotnet store

RIDs that represent concrete operating systems usually follow this pattern: `[os].[version]-[architecture]-[additional qualifiers]` where:

- `[os]` is the operating/platform system moniker. For example, `ubuntu`.
- `[version]` is the operating system version in the form of a dot-separated (`.`) version number. For example, `15.10`.
  - The version **shouldn't** be marketing versions, as they often represent multiple discrete versions of the operating system with varying platform API surface area.
- `[architecture]` is the processor architecture. For example: `x86`, `x64`, `arm`, or `arm64`.
- `[additional qualifiers]` further differentiate different platforms. For example: `aot`.

RID graph

The RID graph or runtime fallback graph is a list of RIDs that are compatible with each other. The RIDs are defined in the Microsoft.NETCore.Platforms package. You can see the list of supported RIDs and the RID graph in the *runtime.json* file, which is located at the CoreFX repo. In this file, you can see that all RIDs, except for the base one, contain an `"#import"` statement. These statements indicate compatible RIDs.

When NuGet restores packages, it tries to find an exact match for the specified runtime. If an exact match is not found, NuGet walks back the graph until it finds the closest compatible system according to the RID graph.

The following example is the actual entry for the `osx.10.12-x64` RID:

```
"osx.10.12-x64": {
    "#import": [ "osx.10.12", "osx.10.11-x64" ]
}
```

The above RID specifies that `osx.10.12-x64` imports `osx.10.11-x64`. So, when NuGet restores packages, it tries to find an exact match for `osx.10.12-x64` in the package. If NuGet cannot find the specific runtime, it can restore packages that specify `osx.10.11-x64` runtimes, for example.

The following example shows a slightly bigger RID graph also defined in the *runtime.json* file:

```
    win7-x64    win7-x86
       |   \   /    |
       |   win7     |
       |     |      |
    win-x64  |  win-x86
          \  |  /
            win
             |
            any
```

All RIDs eventually map back to the root `any` RID.

There are some considerations about RIDs that you have to keep in mind when working with them:

- RIDs are **opaque strings** and should be treated as black boxes.
- Don't build RIDs programmatically.
- Use RIDs that are already defined for the platform.
- The RIDs need to be specific, so don't assume anything from the actual RID value.

Using RIDs

To be able to use RIDs, you have to know which RIDs exist. New values are added regularly to the platform. For the latest and complete version, see the runtime.json file on CoreFX repo.

.NET Core 2.0 SDK introduces the concept of portable RIDs. They are new values added to the RID graph that aren't tied to a specific version or OS distribution and are the preferred choice when using .NET Core 2.0 and higher. They're particularly useful when dealing with multiple Linux distros since most distribution RIDs are mapped to the portable RIDs.

The following list shows a small subset of the most common RIDs used for each OS.

Windows RIDs

Only common values are listed. For the latest and complete version, see the runtime.json file on CoreFX repo.

- Portable (.NET Core 2.0 or later versions)
  - `win-x64`
  - `win-x86`
  - `win-arm`
  - `win-arm64`
- Windows 7 / Windows Server 2008 R2
  - `win7-x64`
  - `win7-x86`
- Windows 8.1 / Windows Server 2012 R2
  - `win81-x64`
  - `win81-x86`
  - `win81-arm`
- Windows 10 / Windows Server 2016

- `win10-x64`
- `win10-x86`
- `win10-arm`
- `win10-arm64`

See Prerequisites for .NET Core on Windows for more information.

Linux RIDs

Only common values are listed. For the latest and complete version, see the runtime.json file on CoreFX repo. Devices running a distribution not listed below may work with one of the Portable RIDs. For example, Raspberry Pi devices running a Linux distribution not listed can be targeted with `linux-arm`.

- Portable (.NET Core 2.0 or later versions)
  - `linux-x64` (Most desktop distributions like CentOS, Debian, Fedora, Ubuntu and derivatives)
  - `linux-musl-x64` (Lightweight distributions using musl like Alpine Linux)
  - `linux-arm` (Linux distributions running on ARM like Raspberry Pi)
- Red Hat Enterprise Linux
  - `rhel-x64` (Superseded by `linux-x64` for RHEL above version 6)
  - `rhel.6-x64` (.NET Core 2.0 or later versions)
- Tizen (.NET Core 2.0 or later versions)
  - `tizen`
  - `tizen.4.0.0`
  - `tizen.5.0.0`

See Prerequisites for .NET Core on Linux for more information.

macOS RIDs

macOS RIDs use the older "OSX" branding. Only common values are listed. For the latest and complete version, see the runtime.json file on CoreFX repo.

- Portable (.NET Core 2.0 or later versions)
  - `osx-x64` (Minimum OS version is macOS 10.12 Sierra)
- macOS 10.10 Yosemite
  - `osx.10.10-x64`
- macOS 10.11 El Capitan
  - `osx.10.11-x64`
- macOS 10.12 Sierra (.NET Core 1.1 or later versions)
  - `osx.10.12-x64`
- macOS 10.13 High Sierra (.NET Core 1.1 or later versions)
  - `osx.10.13-x64`
- macOS 10.14 Mojave (.NET Core 1.1 or later versions)
  - `osx.10.14-x64`

See Prerequisites for .NET Core on macOS for more information.

See also

- Runtime IDs

.NET Core SDK overview

8/17/2019 • 2 minutes to read • Edit Online

The .NET Core SDK is a set of libraries and tools that allow developers to create .NET Core applications and libraries. It contains the following components that are used to build and run applications:

- The .NET Core CLI tools.
- .NET Core libraries and runtime.
- The `dotnet` driver.

Acquiring the .NET Core SDK

As with any tooling, the first thing is to get the tools to your machine. Depending on your scenario, you can install the SDK using one of the following methods:

- Use the native installers.
- Use the installation shell script.

The native installers are primarily meant for developer's machines. The SDK is distributed using each supported platform's native install mechanism, such as DEB packages on Ubuntu or MSI bundles on Windows. These installers install and set up the environment as needed for the user to use the SDK immediately after the install. However, they also require administrative privileges on the machine. You can find the SDK to install on the .NET downloads page.

Install scripts, on the other hand, don't require administrative privileges. However, they also don't install any prerequisites on the machine; you need to install all of the prerequisites manually. The scripts are meant mostly for setting up build servers or when you wish to install the tools without admin privileges (do note the prerequisites caveat above). You can find more information in the install script reference article. If you're interested in how to set up the SDK on your CI build server, see the Using .NET Core SDK and tools in Continuous Integration (CI) article.

By default, the SDK installs in a "side-by-side" (SxS) manner, which means multiple versions of the CLI tools can coexist at any given time on a single machine. How the version gets picked when you're running CLI commands is explained in more detail in the Select the .NET Core version to use article.

See also

- .NET Core CLI
- .NET Core versioning overview
- How to remove the .NET Core runtime and SDK
- Select the .NET Core version to use

.NET Core command-line interface (CLI) tools

9/19/2019 • 3 minutes to read • Edit Online

The .NET Core command-line interface (CLI) is a new cross-platform toolchain for developing .NET applications. The CLI is a foundation upon which higher-level tools, such as Integrated Development Environments (IDEs), editors, and build orchestrators, can rest.

Installation

Either use the native installers or use the installation shell scripts:

- The native installers are primarily used on developer's machines and use each supported platform's native install mechanism, for instance, DEB packages on Ubuntu or MSI bundles on Windows. These installers install and configure the environment for immediate use by the developer but require administrative privileges on the machine. You can view the installation instructions in the .NET Core installation guide.
- Shell scripts are primarily used for setting up build servers or when you wish to install the tools without administrative privileges. Install scripts don't install prerequisites on the machine, which must be installed manually. For more information, see the install script reference topic. For information on how to set up CLI on your continuous integration (CI) build server, see Using .NET Core SDK and tools in Continuous Integration (CI).

By default, the CLI installs in a side-by-side (SxS) manner, so multiple versions of the CLI tools can coexist on a single machine. Determining which version is used on a machine where multiple versions are installed is explained in more detail in the Driver section.

CLI commands

The following commands are installed by default:

- .NET Core 2.x
- .NET Core 1.x

Basic commands

- new
- restore
- build
- publish
- run
- test
- vstest
- pack
- migrate
- clean
- sln
- help
- store

Project modification commands

- add package
- add reference
- remove package
- remove reference
- list reference

Advanced commands

- nuget delete
- nuget locals
- nuget push
- msbuild
- dotnet install script

The CLI adopts an extensibility model that allows you to specify additional tools for your projects. For more information, see the .NET Core CLI extensibility model topic.

Command structure

CLI command structure consists of the driver ("dotnet"), the command, and possibly command arguments and options. You see this pattern in most CLI operations, such as creating a new console app and running it from the command line as the following commands show when executed from a directory named *my_app*:

- .NET Core 2.x
- .NET Core 1.x

```
dotnet new console
dotnet build --output /build_output
dotnet /build_output/my_app.dll
```

Driver

The driver is named dotnet and has two responsibilities, either running a framework-dependent app or executing a command.

To run a framework-dependent app, specify the app after the driver, for example, `dotnet /path/to/my_app.dll`. When executing the command from the folder where the app's DLL resides, simply execute `dotnet my_app.dll`. If you want to use a specific version of the .NET Core Runtime, use the `--fx-version <VERSION>` option (see the dotnet command reference).

When you supply a command to the driver, `dotnet.exe` starts the CLI command execution process. For example:

```
dotnet build
```

First, the driver determines the version of the SDK to use. If there is no 'global.json', the latest version of the SDK available is used. This might be either a preview or stable version, depending on what is latest on the machine. Once the SDK version is determined, it executes the command.

Command

The command performs an action. For example, `dotnet build` builds code. `dotnet publish` publishes code. The commands are implemented as a console application using a `dotnet {command}` convention.

Arguments

The arguments you pass on the command line are the arguments to the command invoked. For example when you execute `dotnet publish my_app.csproj`, the `my_app.csproj` argument indicates the project to publish and is passed to the `publish` command.

Options

The options you pass on the command line are the options to the command invoked. For example when you execute `dotnet publish --output /build_output`, the `--output` option and its value are passed to the `publish` command.

Migration from project.json

If you used Preview 2 tooling to produce *project.json*-based projects, consult the dotnet migrate topic for information on migrating your project to MSBuild/*.csproj* for use with release tooling. For .NET Core projects created prior to the release of Preview 2 tooling, either manually update the project following the guidance in Migrating from DNX to .NET Core CLI (project.json) and then use `dotnet migrate` or directly upgrade your projects.

See also

- dotnet/CLI GitHub Repository
- .NET Core installation guide

.NET Core Global Tools overview

10/15/2019 • 4 minutes to read • Edit Online

This article applies to: ✓ .NET Core 2.1 SDK

A .NET Core Global Tool is a special NuGet package that contains a console application. A Global Tool can be installed on your machine on a default location that is included in the PATH environment variable or on a custom location.

If you want to use a .NET Core Global Tool:

- Find information about the tool (usually a website or GitHub page).
- Check the author and statistics in the home for the feed (usually NuGet.org).
- Install the tool.
- Call the tool.
- Update the tool.
- Uninstall the tool.

IMPORTANT
.NET Core Global Tools appear on your path and run in full trust. Do not install .NET Core Global Tools unless you trust the author.

Find a .NET Core Global Tool

Currently, there isn't a Global Tool search feature in the .NET Core Command-line Interface (CLI). The following are some recommendations on how to find tools:

- You can find .NET Core Global Tools on NuGet. However, NuGet doesn't yet allow you to search specifically for .NET Core Global Tools.
- You may find tool recommendations in blog posts or in the natemcmaster/dotnet-tools GitHub repository.
- You can see the source code for the Global Tools created by the ASP.NET team at the aspnet/DotNetTools GitHub repository.
- You can learn about diagnostic tools at .NET Core dotnet diagnostic Global Tools.

Check the author and statistics

Since .NET Core Global Tools run in full trust and are generally installed on your path, they can be very powerful. Don't download tools from people you don't trust.

If the tool is hosted on NuGet, you can check the author and statistics by searching for the tool.

Install a Global Tool

To install a Global Tool, you use the dotnet tool install .NET Core CLI command. The following example shows how to install a Global Tool in the default location:

```
dotnet tool install -g dotnetsay
```

If the tool can't be installed, error messages are displayed. Check that the feeds you expected are being checked.

If you're trying to install a pre-release version or a specific version of the tool, you can specify the version number using the following format:

```
dotnet tool install -g <package-name> --version <version-number>
```

If installation is successful, a message is displayed showing the command used to call the tool and the version installed, similar to the following example:

```
You can invoke the tool using the following command: dotnetsay
Tool 'dotnetsay' (version '2.0.0') was successfully installed.
```

Global Tools can be installed in the default directory or in a specific location. The default directories are:

OS	PATH
Linux/macOS	`$HOME/.dotnet/tools`
Windows	`%USERPROFILE%\.dotnet\tools`

These locations are added to the user's path when the SDK is first run, so Global Tools installed there can be called directly.

Note that the Global Tools are user-specific, not machine global. Being user-specific means you cannot install a Global Tool that is available to all users of the machine. The tool is only available for each user profile where the tool was installed.

Global Tools can also be installed in a specific directory. When installed in a specific directory, the user must ensure the command is available, by including that directory in the path, by calling the command with the directory specified, or calling the tool from within the specified directory. In this case, the .NET Core CLI doesn't add this location automatically to the PATH environment variable.

Use the tool

Once the tool is installed, you can call it by using its command. Note that the command may not be the same as the package name.

If the command is `dotnetsay`, you call it with:

```
dotnetsay
```

If the tool author wanted the tool to appear in the context of the `dotnet` prompt, they may have written it in a way that you call it as `dotnet <command>`, such as:

```
dotnet doc
```

You can find which tools are included in an installed Global Tool package by listing the installed packages using the dotnet tool list command.

You can also look for usage instructions at the tool's website or by typing one of the following commands:

```
<command> --help
dotnet <command> --help
```

Other CLI commands

The .NET Core SDK contains other commands that support .NET Core Global Tools. Use any of the `dotnet tool` commands with one of the following options:

- `--global` or `-g` specifies that the command is applicable to user-wide Global Tools.
- `--tool-path` specifies a custom location for Global Tools.

To find out which commands are available for Global Tools:

```
dotnet tool --help
```

Updating a Global Tool involves uninstalling and reinstalling it with the latest stable version. To update a Global Tool, use the dotnet tool update command:

```
dotnet tool update -g <packagename>
```

Remove a Global Tool using the dotnet tool uninstall:

```
dotnet tool uninstall -g <packagename>
```

To display all of the Global Tools currently installed on the machine, along with their version and commands, use the dotnet tool list command:

```
dotnet tool list -g
```

See also

- Troubleshoot .NET Core tool usage issues

Create a .NET Core Global Tool using the .NET Core CLI

9/19/2019 • 3 minutes to read • Edit Online

This article teaches you how to create and package a .NET Core Global Tool. The .NET Core CLI allows you to create a console application as a Global Tool, which others can easily install and run. .NET Core Global Tools are NuGet packages that are installed from the .NET Core CLI. For more information about Global Tools, see .NET Core Global Tools overview.

This article applies to: ✓ .NET Core 2.1 SDK

Create a project

This article uses the .NET Core CLI to create and manage a project.

Our example tool will be a console application that generates an ASCII bot and prints a message. First, create a new .NET Core Console Application.

```
dotnet new console -o botsay
```

Navigate to the `botsay` directory created by the previous command.

Add the code

Open the `Program.cs` file with your favorite text editor, such as `vim` or Visual Studio Code.

Add the following `using` directive to the top of the file, this helps shorten the code to display the version information of the application.

```
using System.Reflection;
```

Next, move down to the `Main` method. Replace the method with the following code to process the command-line arguments for your application. If no arguments were passed, a short help message is displayed. Otherwise, all of those arguments are transformed into a string and printed with the bot.

```
static void Main(string[] args)
{
    if (args.Length == 0)
    {
        var versionString = Assembly.GetEntryAssembly()
                                .GetCustomAttribute<AssemblyInformationalVersionAttribute>()
                                .InformationalVersion
                                .ToString();

        Console.WriteLine($"botsay v{versionString}");
        Console.WriteLine("-------------");
        Console.WriteLine("\nUsage:");
        Console.WriteLine("  botsay <message>");
        return;
    }

    ShowBot(string.Join(' ', args));
}
```

Create the bot

Next, add a new method named `ShowBot` that takes a string parameter. This method prints out the message and the ASCII bot. The ASCII bot code was taken from the dotnetbot sample.

```
static void ShowBot(string message)
{
    string bot = $"\n        {message}";
    bot += @"
    __________________
                      \
                       \
                          ....
                          ....'
                           ....
                        ..........
                    .............'..'..
                 ................'..'.....
               .......'..........'..'..'....
              ........'..........'..'..'.....
             .'....'..'..........'..'.......'.
             .'..................'...   ......
             .  ......'.........         .....
             .    _            __        ......
            ..    #            ##        ......
           ....       .                 .......
           ......  .......          ............
            ................  ......................
            ........................'................
           ......................'..'......    .......
        .........................'..'.....       .......
     ........    ..'.............'..'....      ..........
   ..'..'...      ...............'.......      ..........
  ...'......     ...... ..........  ......         .......
 ...........   .......              ........        ......
.......        '...'.'.              '.'.'.'         ....
.......       .....'..               ..'.....
   ..       ..........               ..'........
          ............               ..............
         .............               '..............
        ...........'..              .'.'............
       ...............              .'.'.............
      .............'..               ..'..'...........
      ...............                 .'..............
       .........                        ..............
        .....
";
    Console.WriteLine(bot);
}
```

Test the tool

Run the project and see the output. Try these variations of the command-line to see different results:

```
dotnet run
dotnet run -- "Hello from the bot"
dotnet run -- hello from the bot
```

All arguments after the `--` delimiter are passed to your application.

Setup the global tool

Before you can pack and distribute the application as a Global Tool, you need to modify the project file. Open the `botsay.csproj` file and add three new XML nodes to the `<Project><PropertyGroup>` node:

- `<PackAsTool>`
  [REQUIRED] Indicates that the application will be packaged for install as a Global Tool.

- `<ToolCommandName>`
  [OPTIONAL] An alternative name for the tool, otherwise the command name for the tool will be named after the project file. You can have multiple tools in a package, choosing a unique and friendly name helps differentiate from other tools in the same package.
- `<PackageOutputPath>`
  [OPTIONAL] Where the NuGet package will be produced. The NuGet package is what the .NET Core CLI Global Tools uses to install your tool.

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

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.1</TargetFramework>

    <PackAsTool>true</PackAsTool>
    <ToolCommandName>botsay</ToolCommandName>
    <PackageOutputPath>./nupkg</PackageOutputPath>

  </PropertyGroup>

</Project>
```

Even though `<PackageOutputPath>` is optional, use it in this example. Make sure you set it: `<PackageOutputPath>./nupkg</PackageOutputPath>`.

Next, create a NuGet package for your application.

```
dotnet pack
```

The `botsay.1.0.0.nupkg` file is created in the folder identified by the `<PackageOutputPath>` XML value from the `botsay.csproj` file, which in this example is the `./nupkg` folder. This makes it easy to install and test. When you want to release a tool publicly, upload it to https://www.nuget.org. Once the tool is available on NuGet, developers can perform a user-wide installation of the tool using the `--global` option of the dotnet tool install command.

Now that you have a package, install the tool from that package:

```
dotnet tool install --global --add-source ./nupkg botsay
```

The `--add-source` parameter tells the .NET Core CLI to temporarily use the `./nupkg` folder (our `<PackageOutputPath>` folder) as an additional source feed for NuGet packages. For more information about installing Global Tools, see .NET Core Global Tools overview.

If installation is successful, a message is displayed showing the command used to call the tool and the version installed, similar to the following example:

```
You can invoke the tool using the following command: botsay
Tool 'botsay' (version '1.0.0') was successfully installed.
```

You should now be able to type `botsay` and get a response from the tool.

NOTE

If the install was successful, but you cannot use the `botsay` command, you may need to open a new terminal to refresh the PATH.

Remove the tool

Once you're done experimenting with the tool, you can remove it with the following command:

```
dotnet tool uninstall -g botsay
```

Troubleshoot .NET Core tool usage issues

10/15/2019 • 6 minutes to read • Edit Online

You might come across issues when trying to install or run a .NET Core tool, which can be a global tool or a local tool. This article describes the common root causes and some possible solutions.

Installed .NET Core tool fails to run

When a .NET Core tool fails to run, most likely you ran into one of the following issues:

- The executable file for the tool wasn't found.
- The correct version of the .NET Core runtime wasn't found.

Executable file not found

If the executable file isn't found, you'll see a message similar to the following:

```
Could not execute because the specified command or file was not found.
Possible reasons for this include:
  * You misspelled a built-in dotnet command.
  * You intended to execute a .NET Core program, but dotnet-xyz does not exist.
  * You intended to run a global tool, but a dotnet-prefixed executable with this name could not be found on
the PATH.
```

The name of the executable determines how you invoke the tool. The following table describes the format:

EXECUTABLE NAME FORMAT	INVOCATION FORMAT
`dotnet-<toolName>.exe`	`dotnet <toolName>`
`<toolName>.exe`	`<toolName>`

- Global tools

  Global tools can be installed in the default directory or in a specific location. The default directories are:

OS	PATH
Linux/macOS	`$HOME/.dotnet/tools`
Windows	`%USERPROFILE%\.dotnet\tools`

  If you're trying to run a global tool, check that the `PATH` environment variable on your machine contains the path where you installed the global tool and that the executable is in that path.

  The .NET Core CLI tries to add the default locations to the PATH environment variable on its first usage. However, there are a couple of scenarios where the location might not be added to PATH automatically, so you'll have to edit PATH to configure it for the following cases:

  - If you're using Linux and you've installed the .NET Core SDK using *.tar.gz* files and not apt-get or rpm.
  - If you're using macOS 10.15 "Catalina" or later versions.
  - If you're using macOS 10.14 "Mojave" or earlier versions, and you've installed the .NET Core SDK using

  .tar.gz files and not *.pkg*.
  - If you've installed the .NET Core 3.0 SDK and you've set the `DOTNET_ADD_GLOBAL_TOOLS_TO_PATH` environment variable to `false`.
  - If you've installed .NET Core 2.2 SDK or earlier versions, and you've set the `DOTNET_SKIP_FIRST_TIME_EXPERIENCE` environment variable to `true`.

  For more information about global tools, see .NET Core Global Tools overview.

- Local tools

  If you're trying to run a local tool, verify that there's a manifest file called*dotnet-tools.json* in the current directory or any of its parent directories. This file can also live under a folder named *.config* anywhere in the project folder hierarchy, instead of the root folder. If *dotnet-tools.json* exists, open it and check for the tool you're trying to run. If the file doesn't contain an entry for `"isRoot": true`, then also check further up the file hierarchy for additional tool manifest files.

  If you're trying to run a .NET Core tool that was installed with a specified path, you need to include that path when using the tool. An example of using a tool-path installed tool is:

  ```
  ..\<toolDirectory>\dotnet-<toolName>
  ```

Runtime not found

.NET Core tools are framework-dependent applications, which means they rely on a .NET Core runtime installed on your machine. If the expected runtime isn't found, they follow normal .NET Core runtime roll-forward rules such as:

- An application rolls forward to the highest patch release of the specified major and minor version.
- If there's no matching runtime with a matching major and minor version number, the next higher minor version is used.
- Roll forward doesn't occur between preview versions of the runtime or between preview versions and release versions. So, .NET Core tools created using preview versions must be rebuilt and republished by the author and reinstalled.

Roll-forward won't occur by default in two common scenarios:

- Only lower versions of the runtime are available. Roll-forward only selects later versions of the runtime.
- Only higher major versions of the runtime are available. Roll-forward doesn't cross major version boundaries.

If an application can't find an appropriate runtime, it fails to run and reports an error.

You can find out which .NET Core runtimes are installed on your machine using one of the following commands:

```
dotnet --list-runtimes
dotnet --info
```

If you think the tool should support the runtime version you currently have installed, you can contact the tool author and see if they can update the version number or multi-target. Once they've recompiled and republished their tool package to NuGet with an updated version number, you can update your copy. While that doesn't happen, the quickest solution for you is to install a version of the runtime that would work with the tool you're trying to run. To download a specific .NET Core runtime version, visit the .NET Core download page.

If you install the .NET Core SDK to a non-default location, you need to set the environment variable `DOTNET_ROOT` to the directory that contains the `dotnet` executable.

.NET Core tool installation fails

There are a number of reasons the installation of a .NET Core global or local tool may fail. When the tool installation fails, you'll see a message similar to following one:

```
Tool '{0}' failed to install. This failure may have been caused by:

* You are attempting to install a preview release and did not use the --version option to specify the version.
* A package by this name was found, but it was not a .NET Core tool.
* The required NuGet feed cannot be accessed, perhaps because of an Internet connection problem.
* You mistyped the name of the tool.

For more reasons, including package naming enforcement, visit https://aka.ms/failure-installing-tool
```

To help diagnose these failures, NuGet messages are shown directly to the user, along with the previous message. The NuGet message may help you identify the problem.

Package naming enforcement

Microsoft has changed its guidance on the Package ID for tools, resulting in a number of tools not being found with the predicted name. The new guidance is that all Microsoft tools be prefixed with "Microsoft." This prefix is reserved and can only be used for packages signed with a Microsoft authorized certificate.

During the transition, some Microsoft tools will have the old form of the package ID, while others will have the new form:

```
dotnet tool install -g Microsoft.<toolName>
dotnet tool install -g <toolName>
```

As package IDs are updated, you'll need to change to the new package ID to get the latest updates. Packages with the simplified tool name will be deprecated.

Preview releases

- You're attempting to install a preview release and didn't use the `--version` option to specify the version.

.NET Core tools that are in preview must be specified with a portion of the name to indicate that they are in preview. You don't need to include the entire preview. Assuming the version numbers are in the expected format, you can use something like the following example:

```
dotnet tool install -g --version 1.1.0-pre <toolName>
```

NOTE

The .NET Core CLI team is planning to add a `--preview` switch in a future release to make this easier.

Package isn't a .NET Core tool

- A NuGet package by this name was found, but it wasn't a .NET Core tool.

If you try to install a NuGet package that is a regular NuGet package and not a .NET Core tool, you'll see an error similar to the following:

> NU1212: Invalid project-package combination for `<ToolName>`. DotnetToolReference project style can only contain references of the DotnetTool type.

NuGet feed can't be accessed

- The required NuGet feed can't be accessed, perhaps because of an Internet connection problem.

Tool installation requires access to the NuGet feed that contains the tool package. It fails if the feed isn't available. You can alter feeds with `nuget.config`, request a specific `nuget.config` file, or specify additional feeds with the `--add-source` switch. By default, NuGet throws an error for any feed that can't connect. The flag `--ignore-failed-sources` can skip these non-reachable sources.

Package ID incorrect

- You mistyped the name of the tool.

A common reason for failure is that the tool name isn't correct. This can happen because of mistyping, or because the tool has moved or been deprecated. For tools on NuGet.org, one way to ensure you have the name correct is to search for the tool at NuGet.org and copy the installation command.

See also

- .NET Core Global Tools overview

Elevated access for dotnet commands

9/24/2019 • 4 minutes to read • Edit Online

Software development best practices guide developers to writing software that requires the least amount of privilege. However, some software, like performance monitoring tools, requires admin permission because of operating system rules. The following guidance describes supported scenarios for writing such software with .NET Core.

The following commands can be run elevated:

- `dotnet tool` commands, such as dotnet tool install.
- `dotnet run --no-build`

We don't recommend running other commands elevated. In particular, we don't recommend elevation with commands that use MSBuild, such as dotnet restore, dotnet build, and dotnet run. The primary issue is permission management problems when a user transitions back and forth between root and a restricted account after issuing dotnet commands. You may find as a restricted user that you don't have access to the file built by a root user. There are ways to resolve this situation, but they're unnecessary to get into in the first place.

You can run commands as root as long as you don’t transition back and forth between root and a restricted account. For example, Docker containers run as root by default, so they have this characteristic.

Global tool installation

The following instructions demonstrate the recommended way to install, run, and uninstall .NET Core tools that require elevated permissions to execute.

- Windows
- Linux
- macOS

Install the global tool

If the folder `%ProgramFiles%\dotnet-tools` already exists, do the following to check whether the "Users" group has permission to write or modify that directory:

- Right-click the `%ProgramFiles%\dotnet-tools` folder and select **Properties**. The **Common Properties** dialog box opens.
- Select the **Security** tab. Under **Group or user names**, check whether the “Users” group has permission to write or modify the directory.
- If the "Users" group can write or modify the directory, use a different directory name when installing the tools rather than *dotnet-tools*.

To install tools, run the following command in elevated prompt. It will create the *dotnet-tools* folder during the installation.

```
dotnet tool install PACKAGEID --tool-path "%ProgramFiles%\dotnet-tools".
```

Run the global tool

Option 1 Use the full path with elevated prompt:

```
"%ProgramFiles%\dotnet-tools\TOOLCOMMAND"
```

Option 2 Add the newly created folder to `%Path%`. You only need to do this operation once.

```
setx Path "%Path%;%ProgramFiles%\dotnet-tools\"
```

And run with:

```
TOOLCOMMAND
```

Uninstall the global tool

In an elevated prompt, type the following command:

```
dotnet tool uninstall PACKAGEID --tool-path "%ProgramFiles%\dotnet-tools"
```

Local tools

Local tools are scoped per subdirectory tree, per user. When run elevated, local tools share a restricted user environment to the elevated environment. In Linux and macOS, this results in files being set with root user-only access. If the user switches back to a restricted account, the user can no longer access or write to the files. So installing tools that require elevation as local tools isn't recommended. Instead, use the `--tool-path` option and the previous guidelines for global tools.

Elevation during development

During development, you may need elevated access to test your application. This scenario is common for IoT apps, for example. We recommend that you build the application without elevation and then run it with elevation. There are a few patterns, as follows:

- Using generated executable (it provides the best startup performance):

  ```
  dotnet build
  sudo ./bin/Debug/netcoreapp3.0/APPLICATIONNAME
  ```

- Using the dotnet run command with the `—no-build` flag to avoid generating new binaries:

  ```
  dotnet build
  sudo dotnet run --no-build
  ```

See also

- .NET Core Global Tools overview

.NET Core CLI tools extensibility model

8/14/2019 • 8 minutes to read • Edit Online

This document covers the different ways you can extend the .NET Core Command-line Interface (CLI) tools and explain the scenarios that drive each one of them. You'll see how to consume the tools as well as how to build the different types of tools.

How to extend CLI tools

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.

The three extensibility mechanisms outlined above are not exclusive. You can use one, or all, or a combination of them. Which one to pick depends largely on the goal you are trying to achieve with your extension.

Per-project based extensibility

Per-project tools are framework-dependent deployments that are distributed as NuGet packages. Tools are only available in the context of the project that references them and for which they are restored. Invocation outside of the context of the project (for example, outside of the directory that contains the project) will fail because the command cannot be found.

These tools are perfect for build servers, since nothing outside of the project file is needed. The build process runs restore for the project it builds and tools will be available. Language projects, such as F#, are also in this category since each project can only be written in one specific language.

Finally, this extensibility model provides support for creation of tools that need access to the built output of the project. For instance, various Razor view tools in ASP.NET MVC applications fall into this category.

Consuming per-project tools

Consuming these tools requires you to add a `<DotNetCliToolReference>` element to your project file for each tool you want to use. Inside the `<DotNetCliToolReference>` element, you reference the package in which the tool resides and specify the version you need. After running `dotnet restore`, the tool and its dependencies are restored.

> **NOTE**
>
> Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

For tools that need to load the build output of the project for execution, there is usually another dependency which is listed under the regular dependencies in the project file. Since CLI uses MSBuild as its build engine, we recommend that these parts of the tool be written as custom MSBuild targets and tasks, since they can then take part in the overall build process. Also, they can get any and all data easily that is produced via the build, such as the location of the output files, the current configuration being built, and so on. All this information becomes a set of MSBuild properties that can be read from any target. You can see how to add a custom target using NuGet later in this document.

Let's review an example of adding a simple tools-only tool to a simple project. Given an example command called `dotnet-api-search` that allows you to search through the NuGet packages for the specified API, here is a console application's project file that uses that tool:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>

  <!-- The tools reference -->
  <ItemGroup>
    <DotNetCliToolReference Include="dotnet-api-search" Version="1.0.0" />
  </ItemGroup>
</Project>
```

The `<DotNetCliToolReference>` element is structured in a similar way as the `<PackageReference>` element. It needs the package ID of the package containing the tool and its version to be able to restore.

Building tools

As mentioned, tools are just portable console applications. You build tools as you would build any other console application. After you build it, you use the `dotnet pack` command to create a NuGet package (.nupkg file) that contains your code, information about its dependencies, and so on. You can give any name to the package, but the application inside, the actual tool binary, has to conform to the convention of `dotnet-<command>` in order for `dotnet` to be able to invoke it.

NOTE

In pre-RC3 versions of the .NET Core command-line tools, the `dotnet pack` command had a bug that caused the *.runtimeconfig.json* to not be packed with the tool. Lacking that file results in errors at runtime. If you encounter this behavior, be sure to update to the latest tooling and try the `dotnet pack` again.

Since tools are portable applications, the user consuming the tool must have the version of the .NET Core libraries that the tool was built against in order to run the tool. Any other dependency that the tool uses and that is not contained within the .NET Core libraries is restored and placed in the NuGet cache. The entire tool is, therefore, run using the assemblies from the .NET Core libraries as well as assemblies from the NuGet cache.

These kinds of tools have a dependency graph that is completely separate from the dependency graph of the project that uses them. The restore process first restores the project's dependencies and then restores each of the tools and their dependencies.

You can find richer examples and different combinations of this in the .NET Core CLI repo. You can also see the implementation of tools used in the same repo.

Custom targets

NuGet has the capability to package custom MSBuild targets and props files. With the move of the .NET Core CLI tools to use MSBuild, the same mechanism of extensibility now applies to .NET Core projects. You would use this type of extensibility when you want to extend the build process, or when you want to access any of the artifacts in the build process, such as generated files, or you want to inspect the configuration under which the build is invoked, and so on.

In the following example, you can see the target's project file using the `csproj` syntax. This instructs the `dotnet pack` command what to package, placing the targets files as well as the assemblies into the *build* folder inside the package. Notice the `<ItemGroup>` element that has the `Label` property set to `dotnet pack instructions`, and the Target defined beneath it.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <Description>Sample Packer</Description>
    <VersionPrefix>0.1.0-preview</VersionPrefix>
    <TargetFramework>netstandard1.3</TargetFramework>
    <DebugType>portable</DebugType>
    <AssemblyName>SampleTargets.PackerTarget</AssemblyName>
  </PropertyGroup>
  <ItemGroup>
    <EmbeddedResource Include="Resources\Pkg\dist-template.xml;compiler\resources\**\*"
Exclude="bin\**;obj\**;**\*.xproj;packages\**" />
    <None Include="build\SampleTargets.PackerTarget.targets" />
  </ItemGroup>
  <ItemGroup Label="dotnet pack instructions">
    <Content Include="build\*.targets">
      <Pack>true</Pack>
      <PackagePath>build\</PackagePath>
    </Content>
  </ItemGroup>
  <Target Name="CollectRuntimeOutputs" BeforeTargets="_GetPackageFiles">
    <!-- Collect these items inside a target that runs after build but before packaging. -->
    <ItemGroup>
      <Content Include="$(OutputPath)\*.dll;$(OutputPath)\*.json">
        <Pack>true</Pack>
        <PackagePath>build\</PackagePath>
      </Content>
    </ItemGroup>
  </Target>
  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.DependencyModel" Version="1.0.1-beta-000933"/>
    <PackageReference Include="Microsoft.Build.Framework" Version="0.1.0-preview-00028-160627" />
    <PackageReference Include="Microsoft.Build.Utilities.Core" Version="0.1.0-preview-00028-160627" />
    <PackageReference Include="Newtonsoft.Json" Version="9.0.1" />
  </ItemGroup>
  <ItemGroup />
  <PropertyGroup Label="Globals">
    <ProjectGuid>463c66f0-921d-4d34-8bde-7c9d0bffaf7b</ProjectGuid>
  </PropertyGroup>
  <PropertyGroup Condition=" '$(TargetFramework)' == 'netstandard1.3' ">
    <DefineConstants>$(DefineConstants);NETSTANDARD1_3</DefineConstants>
  </PropertyGroup>
  <PropertyGroup Condition=" '$(Configuration)' == 'Release' ">
    <DefineConstants>$(DefineConstants);RELEASE</DefineConstants>
  </PropertyGroup>
</Project>
```

Consuming custom targets is done by providing a `<PackageReference>` that points to the package and its version inside the project that is being extended. Unlike the tools, the custom targets package does get included into the consuming project's dependency closure.

Using the custom target depends solely on how you configure it. Since it's an MSBuild target, it can depend on a given target, run after another target and can also be manually invoked using the `dotnet msbuild -t:<target-name>` command.

However, if you want to provide a better user experience to your users, you can combine per-project tools and custom targets. In this scenario, the per-project tool would essentially just accept whatever needed parameters and would translate that into the required `dotnet msbuild` invocation that would execute the target. You can see a sample of this kind of synergy on the MVP Summit 2016 Hackathon samples repo in the `dotnet-packer` project.

PATH-based extensibility

PATH-based extensibility is usually used for development machines where you need a tool that conceptually covers more than a single project. The main drawback of this extension mechanism is that it's tied to the machine where the tool exists. If you need it on another machine, you would have to deploy it.

This pattern of CLI toolset extensibility is very simple. As covered in the .NET Core CLI overview, `dotnet` driver can run any command that is named after the `dotnet-<command>` convention. The default resolution logic first probes several locations and finally falls back to the system PATH. If the requested command exists in the system PATH and is a binary that can be invoked, `dotnet` driver will invoke it.

The file must be executable. On Unix systems, this means anything that has the execute bit set via `chmod +x`. On Windows, you can use *cmd* files.

Let's take a look at the very simple implementation of a "Hello World" tool. We will use both `bash` and `cmd` on Windows. The following command will simply echo "Hello World" to the console.

```
#!/bin/bash

echo "Hello World!"
```

```
echo "Hello World"
```

On macOS, we can save this script as `dotnet-hello` and set its executable bit with `chmod +x dotnet-hello`. We can then create a symbolic link to it in `/usr/local/bin` using the command `ln -s <full_path>/dotnet-hello /usr/local/bin/`. This will make it possible to invoke the command using the `dotnet hello` syntax.

On Windows, we can save this script as `dotnet-hello.cmd` and put it in a location that is in a system path (or you can add it to a folder that is already in the path). After this, you can just use `dotnet hello` to run this example.

Custom templates for dotnet new

11/1/2019 • 9 minutes to read • Edit Online

The .NET Core SDK comes with many templates already installed and ready for you to use. The `dotnet new` command isn't only the way to use a template, but also how to install and uninstall templates. Starting with .NET Core 2.0, you can create your own custom templates for any type of project, such as an app, service, tool, or class library. You can even create a template that outputs one or more independent files, such as a configuration file.

You can install custom templates from a NuGet package on any NuGet feed, by referencing a NuGet *.nupkg* file directly, or by specifying a file system directory that contains the template. The template engine offers features that allow you to replace values, include and exclude files, and execute custom processing operations when your template is used.

The template engine is open source, and the online code repository is at dotnet/templating on GitHub. Visit the dotnet/dotnet-template-samples repo for samples of templates. More templates, including templates from third parties, are found at Available templates for dotnet new on GitHub. For more information about creating and using custom templates, see How to create your own templates for dotnet new and the dotnet/templating GitHub repo Wiki.

To follow a walkthrough and create a template, see the Create a custom template for dotnet new tutorial.

.NET default templates

When you install the .NET Core SDK, you receive over a dozen built-in templates for creating projects and files, including console apps, class libraries, unit test projects, ASP.NET Core apps (including Angular and React projects), and configuration files. To list the built-in templates, run the `dotnet new` command with the `-l|--list` option:

```
dotnet new --list
```

Configuration

A template is composed of the following parts:

- Source files and folders.
- A configuration file (*template.json*).

Source files and folders

The source files and folders include whatever files and folders you want the template engine to use when the `dotnet new <TEMPLATE>` command is run. The template engine is designed to use *runnable projects* as source code to produce projects. This has several benefits:

- The template engine doesn't require you to inject special tokens into your project's source code.
- The code files aren't special files or modified in any way to work with the template engine. So, the tools you normally use when working with projects also work with template content.
- You build, run, and debug your template projects just like you do for any of your other projects.
- You can quickly create a template from an existing project just by adding a *./.template.config/template.json* configuration file to the project.

Files and folders stored in the template aren't limited to formal .NET project types. Source files and folders may consist of any content that you wish to create when the template is used, even if the template engine produces just one file as its output.

Files generated by the template can be modified based on logic and settings you've provided in the *template.json* configuration file. The user can override these settings by passing options to the `dotnet new <TEMPLATE>` command. A common example of custom logic is providing a name for a class or variable in the code file that's deployed by a template.

template.json

The *template.json* file is placed in a *.template.config* folder in the root directory of the template. The file provides configuration information to the template engine. The minimum configuration requires the members shown in the following table, which is sufficient to create a functional template.

MEMBER	TYPE	DESCRIPTION
`$schema`	URI	The JSON schema for the *template.json* file. Editors that support JSON schemas enable JSON-editing features when the schema is specified. For example, Visual Studio Code requires this member to enable IntelliSense. Use a value of `http://json.schemastore.org/template`.
`author`	string	The author of the template.
`classifications`	array(string)	Zero or more characteristics of the template that a user might use to find the template when searching for it. The classifications also appear in the *Tags* column when it appears in a list of templates produced by using the `dotnet new -l|--list` command.
`identity`	string	A unique name for this template.
`name`	string	The name for the template that users should see.
`shortName`	string	A default shorthand name for selecting the template that applies to environments where the template name is specified by the user, not selected via a GUI. For example, the short name is useful when using templates from a command prompt with CLI commands.

The full schema for the *template.json* file is found at the JSON Schema Store. For more information about the *template.json* file, see the dotnet templating wiki.

Example

For example, here is a template folder that contains two content files: *console.cs* and *readme.txt*. Take notice that there is the required folder named *.template.config* that contains the *template.json* file.

```
└───mytemplate
    │   console.cs
    │   readme.txt
    │
    └───.template.config
            template.json
```

The *template.json* file looks like the following:

```
{
  "$schema": "http://json.schemastore.org/template",
  "author": "Travis Chau",
  "classifications": [ "Common", "Console" ],
  "identity": "AdatumCorporation.ConsoleTemplate.CSharp",
  "name": "Adatum Corporation Console Application",
  "shortName": "adatumconsole"
}
```

The *mytemplate* folder is an installable template pack. Once the pack is installed, the `shortName` can be used with the `dotnet new` command. For example, `dotnet new adatumconsole` would output the `console.cs` and `readme.txt` files to the current folder.

Packing a template into a NuGet package (nupkg file)

A custom template is packed with the dotnet pack command and a *.csproj* file. Alternatively, NuGet can be used with the nuget pack command along with a *.nuspec* file. However, NuGet requires the .NET Framework on Windows and Mono on Linux and MacOS.

The *.csproj* file is slightly different from a traditional code-project *.csproj* file. Note the following settings:

1. The `<PackageType>` setting is added and set to `Template`.
2. The `<PackageVersion>` setting is added and set to a valid NuGet version number.
3. The `<PackageId>` setting is added and set to a unique identifier. This identifier is used to uninstall the template pack and is used by NuGet feeds to register your template pack.
4. Generic metadata settings should be set: `<Title>`, `<Authors>`, `<Description>`, and `<PackageTags>`.
5. The `<TargetFramework>` setting must be set, even though the binary produced by the template process isn't used. In the example below it's set to `netstandard2.0`.

A template pack, in the form of a *.nupkg* NuGet package, requires that all templates be stored in the *content* folder within the package. There are a few more settings to add to a *.csproj* file to ensure that the generated *.nupkg* can be installed as a template pack:

1. The `<IncludeContentInPack>` setting is set to `true` to include any file the project sets as **content** in the NuGet package.
2. The `<IncludeBuildOutput>` setting is set to `false` to exclude all binaries generated by the compiler from the NuGet package.
3. The `<ContentTargetFolders>` setting is set to `content`. This makes sure that the files set as **content** are stored in the *content* folder in the NuGet package. This folder in the NuGet package is parsed by the dotnet template system.

An easy way to exclude all code files from being compiled by your template project is by using the `<Compile Remove="**\*" />` item in your project file, inside an `<ItemGroup>` element.

An easy way to structure your template pack is to put all templates in individual folders, and then each template folder inside of a *templates* folder that is located in the same directory as your *.csproj* file. This way, you can use a single project item to include all files and folders in the *templates* as **content**. Inside of an `<ItemGroup>` element, create a `<Content Include="templates\**\*" Exclude="templates\**\bin\**;templates\**\obj\**" />` item.

Here is an example *.csproj* file that follows all of the guidelines above. It packs the *templates* child folder to the *content* package folder and excludes any code file from being compiled.

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

  <PropertyGroup>
    <PackageType>Template</PackageType>
    <PackageVersion>1.0</PackageVersion>
    <PackageId>AdatumCorporation.Utility.Templates</PackageId>
    <Title>AdatumCorporation Templates</Title>
    <Authors>Me</Authors>
    <Description>Templates to use when creating an application for Adatum Corporation.</Description>
    <PackageTags>dotnet-new;templates;contoso</PackageTags>
    <TargetFramework>netstandard2.0</TargetFramework>

    <IncludeContentInPack>true</IncludeContentInPack>
    <IncludeBuildOutput>false</IncludeBuildOutput>
    <ContentTargetFolders>content</ContentTargetFolders>
  </PropertyGroup>

  <ItemGroup>
    <Content Include="templates\**\*" Exclude="templates\**\bin\**;templates\**\obj\**" />
    <Compile Remove="**\*" />
  </ItemGroup>

</Project>
```

The example below demonstrates the file and folder structure of using a *.csproj* to create a template pack. The *MyDotnetTemplates.csproj* file and *templates* folder are both located at the root of a directory named *project_folder*. The *templates* folder contains two templates, *mytemplate1* and *mytemplate2*. Each template has content files and a *.template.config* folder with a *template.json* config file.

```
project_folder
│   MyDotnetTemplates.csproj
│
└───templates
    ├───mytemplate1
    │   │   console.cs
    │   │   readme.txt
    │   │
    │   └───.template.config
    │           template.json
    │
    └───mytemplate2
        │   otherfile.cs
        │
        └───.template.config
                template.json
```

Installing a template

Use the dotnet new -i|--install command to install a package.

To install a template from a NuGet package stored at nuget.org

Use the NuGet package identifier to install a template package.

```
dotnet new -i <NUGET_PACKAGE_ID>
```

To install a template from a local nupkg file

Provide the path to a *.nupkg* NuGet package file.

```
dotnet new -i <PATH_TO_NUPKG_FILE>
```

To install a template from a file system directory

Templates can be installed from a template folder, such as the *mytemplate1* folder from the example above. Specify the folder path of the *.template.config* folder. The path to the template directory does not need to be absolute. However, an absolute path is required to uninstall a template that is installed from a folder.

```
dotnet new -i <FILE_SYSTEM_DIRECTORY>
```

Get a list of installed templates

The uninstall command, without any other parameters, will list all installed templates.

```
dotnet new -u
```

That command returns something similar to the following output:

```
Template Instantiation Commands for .NET Core CLI

Currently installed items:
  Microsoft.DotNet.Common.ItemTemplates
    Templates:
      global.json file (globaljson)
      NuGet Config (nugetconfig)
      Solution File (sln)
      Dotnet local tool manifest file (tool-manifest)
      Web Config (webconfig)
  Microsoft.DotNet.Common.ProjectTemplates.3.0
    Templates:
      Class library (classlib) C#
      Class library (classlib) F#
      Class library (classlib) VB
      Console Application (console) C#
      Console Application (console) F#
      Console Application (console) VB
...
```

The first level of items after `Currently installed items:` are the identifiers used in uninstalling a template. And in the example above, `Microsoft.DotNet.Common.ItemTemplates` and `Microsoft.DotNet.Common.ProjectTemplates.3.0` are listed. If the template was installed by using a file system path, this identifier will the folder path of the *.template.config* folder.

Uninstalling a template

Use the dotnet new -u|--uninstall command to uninstall a package.

If the package was installed by either a NuGet feed or by a *.nupkg* file directly, provide the identifier.

```
dotnet new -u <NUGET_PACKAGE_ID>
```

If the package was installed by specifying a path to the *.template.config* folder, use that **absolute** path to uninstall the package. You can see the absolute path of the template in the output provided by the `dotnet new -u` command. For more information, see the Get a list of installed templates section above.

```
dotnet new -u <ABSOLUTE_FILE_SYSTEM_DIRECTORY>
```

Create a project using a custom template

After a template is installed, use the template by executing the `dotnet new <TEMPLATE>` command as you would with any other pre-installed template. You can also specify options to the `dotnet new` command, including template-specific options you configured in the template settings. Supply the template's short name directly to the command:

```
dotnet new <TEMPLATE>
```

See also

- Create a custom template for dotnet new (tutorial)
- dotnet/templating GitHub repo Wiki
- dotnet/dotnet-template-samples GitHub repo
- How to create your own templates for dotnet new
- *template.json* schema at the JSON Schema Store

How to enable TAB completion for .NET Core CLI

11/7/2019 • 2 minutes to read • Edit Online

Starting with .NET Core 2.0 SDK, the .NET Core CLI supports tab completion. This article describes how to configure tab completion for three shells, PowerShell, Bash, and zsh. Other shells may have support for auto completion. Refer to their documentation on how to configure auto completion, the steps should be similar to the steps described in this article.

This article applies to: ✓ .NET Core 2.x SDK

Once setup, tab completion for the .NET Core CLI is triggered by typing a `dotnet` command in the shell, and then pressing the TAB key. The current command line is sent to the `dotnet complete` command, and the results are processed by your shell. You can test the results without enabling tab completion by sending something directly to the `dotnet complete` command. For example:

```
> dotnet complete "dotnet a"
add
clean
--diagnostics
migrate
pack
```

If that command doesn't work, make sure that .NET Core 2.0 SDK or above is installed. If it's installed, but that command still doesn't work, make sure that the `dotnet` command resolves to a version of .NET Core 2.0 SDK and above. Use the `dotnet --version` command to see what version of `dotnet` your current path is resolving to. For more information, see Select the .NET Core version to use.

Examples

Here are some examples of what tab completion provides:

INPUT	BECOMES	BECAUSE
`dotnet a⇥`	`dotnet add`	`add` is the first subcommand, alphabetically.
`dotnet add p⇥`	`dotnet add --help`	Tab completion matches substrings and `--help` comes first alphabetically.
`dotnet add p⇥⇥`	`dotnet add package`	Pressing tab a second time brings up the next suggestion.
`dotnet add package Microsoft⇥`	`dotnet add package Microsoft.ApplicationInsights.Web`	Results are returned alphabetically.
`dotnet remove reference ⇥`	`dotnet remove reference ..\..\src\OmniSharp.DotNet\OmniSharp.DotNet.csproj`	Tab completion is project file aware.

PowerShell

To add tab completion to **PowerShell** for the .NET Core CLI, create or edit the profile stored in the variable `$PROFILE`. For more information, see How to create your profile and Profiles and execution policy.

Add the following code to your profile:

```
# PowerShell parameter completion shim for the dotnet CLI
Register-ArgumentCompleter -Native -CommandName dotnet -ScriptBlock {
     param($commandName, $wordToComplete, $cursorPosition)
         dotnet complete --position $cursorPosition "$wordToComplete" | ForEach-Object {
            [System.Management.Automation.CompletionResult]::new($_, $_, 'ParameterValue', $_)
         }
 }
```

bash

To add tab completion to your **bash** shell for the .NET Core CLI, add the following code to your `.bashrc` file:

```
# bash parameter completion for the dotnet CLI

_dotnet_bash_complete()
{
  local word=${COMP_WORDS[COMP_CWORD]}

  local completions
  completions="$(dotnet complete --position "${COMP_POINT}" "${COMP_LINE}" 2>/dev/null)"
  if [ $? -ne 0 ]; then
    completions=""
  fi

  COMPREPLY=( $(compgen -W "$completions" -- "$word") )
}

complete -f -F _dotnet_bash_complete dotnet
```

zsh

To add tab completion to your **zsh** shell for the .NET Core CLI, add the following code to your `.zshrc` file:

```
# zsh parameter completion for the dotnet CLI

_dotnet_zsh_complete()
{
  local completions=("$(dotnet complete "$words")")

  reply=( "${(ps:\n:)completions}" )
}

compctl -K _dotnet_zsh_complete dotnet
```

.NET Core SDK telemetry

8/29/2019 • 5 minutes to read • Edit Online

The .NET Core SDK includes a telemetry feature that collects usage data and exception information when the .NET Core CLI crashes. The .NET Core CLI comes with the .NET Core SDK and is the set of verbs that enable you to build, test, and publish your .NET Core apps. It's important that the .NET team understands how the tools are used so they can be improved. Information on failures helps the team resolve problems and fix bugs.

The collected data is anonymous and published in aggregate under the Creative Commons Attribution License.

Scope

`dotnet` has two functions: to run apps, and to execute CLI commands. Telemetry *isn't collected* when using `dotnet` to start an application in the following format:

- `dotnet [path-to-app].dll`

Telemetry *is collected* when using any of the .NET Core CLI commands, such as:

- `dotnet build`
- `dotnet pack`
- `dotnet run`

How to opt out

The .NET Core SDK telemetry feature is enabled by default. To opt out of the telemetry feature, set the `DOTNET_CLI_TELEMETRY_OPTOUT` environment variable to `1` or `true`.

A single telemetry entry is also sent by the .NET Core SDK installer when a successful installation happens. To opt out, set the `DOTNET_CLI_TELEMETRY_OPTOUT` environment variable before you install the .NET Core SDK.

Disclosure

The .NET Core SDK displays text similar to the following when you first run one of the .NET Core CLI commands (for example, `dotnet build`). Text may vary slightly depending on the version of the SDK you're running. This "first run" experience is how Microsoft notifies you about data collection.

```
Telemetry
---------
The .NET Core tools collect usage data in order to help us improve your experience. The data is anonymous. It
is collected by Microsoft and shared with the community. You can opt-out of telemetry by setting the
DOTNET_CLI_TELEMETRY_OPTOUT environment variable to '1' or 'true' using your favorite shell.

Read more about .NET Core CLI Tools telemetry: https://aka.ms/dotnet-cli-telemetry
```

Data points

The telemetry feature doesn't collect personal data, such as usernames or email addresses. It doesn't scan your code and doesn't extract project-level data, such as name, repository, or author. The data is sent securely to Microsoft servers using Azure Monitor technology, held under restricted access, and published under strict security controls from secure Azure Storage systems.

Protecting your privacy is important to us. If you suspect the telemetry is collecting sensitive data or the data is being insecurely or inappropriately handled, file an issue in the dotnet/cli repository or send an email to dotnet@microsoft.com for investigation.

The telemetry feature collects the following data:

SDK VERSIONS	DATA
All	Timestamp of invocation.
All	Command invoked (for example, "build"), hashed starting in 2.1.
All	Three octet IP address used to determine the geographical location.
All	Operating system and version.
All	Runtime ID (RID) the SDK is running on.
All	.NET Core SDK version.
All	Telemetry profile: an optional value only used with explicit user opt-in and used internally at Microsoft.
>=2.0	Command arguments and options: several arguments and options are collected (not arbitrary strings). See collected options. Hashed after 2.1.300.
>=2.0	Whether the SDK is running in a container.
>=2.0	Target frameworks (from the `TargetFramework` event), hashed starting in 2.1.
>=2.0	Hashed Media Access Control (MAC) address: a cryptographically (SHA256) anonymous and unique ID for a machine.
>=2.0	Hashed current working directory.
>=2.0	Install success report, with hashed installer exe filename.
>=2.1.300	Kernel version.
>=2.1.300	Libc release/version.
>=3.0.100	Whether the output was redirected (true or false).
>=3.0.100	On a CLI/SDK crash, the exception type and its stack trace (only CLI/SDK code is included in the stack trace sent). For more information, see .NET Core CLI/SDK crash exception telemetry collected.

Collected options

Certain commands send additional data. A subset of commands sends the first argument:

COMMAND	FIRST ARGUMENT DATA SENT
`dotnet help <arg>`	The command help is being queried for.
`dotnet new <arg>`	The template name (hashed).
`dotnet add <arg>`	The word `package` or `reference`.
`dotnet remove <arg>`	The word `package` or `reference`.
`dotnet list <arg>`	The word `package` or `reference`.
`dotnet sln <arg>`	The word `add`, `list`, or `remove`.
`dotnet nuget <arg>`	The word `delete`, `locals`, or `push`.

A subset of commands sends selected options if they're used, along with their values:

OPTION	COMMANDS
`--verbosity`	All commands
`--language`	`dotnet new`
`--configuration`	`dotnet build`, `dotnet clean`, `dotnet publish`, `dotnet run`, `dotnet test`
`--framework`	`dotnet build`, `dotnet clean`, `dotnet publish`, `dotnet run`, `dotnet test`, `dotnet vstest`
`--runtime`	`dotnet build`, `dotnet publish`
`--platform`	`dotnet vstest`
`--logger`	`dotnet vstest`
`--sdk-package-version`	`dotnet migrate`

Except for `--verbosity` and `--sdk-package-version`, all the other values are hashed starting with .NET Core 2.1.100 SDK.

.NET Core CLI/SDK crash exception telemetry collected

If the .NET Core CLI/SDK crashes, it collects the name of the exception and stack trace of the CLI/SDK code. This information is collected to assess problems and improve the quality of the .NET Core SDK and CLI. This article provides information about the data we collect. It also provides tips on how users building their own version of the .NET Core SDK can avoid inadvertent disclosure of personal or sensitive information.

Types of collected data

.NET Core CLI collects information for CLI/SDK exceptions only, not exceptions in your application. The collected data contains the name of the exception and the stack trace. This stack trace is of CLI/SDK code.

The following example shows the kind of data that is collected:

```
System.IO.IOException
at System.ConsolePal.WindowsConsoleStream.Write(Byte[] buffer, Int32 offset, Int32 count)
at System.IO.StreamWriter.Flush(Boolean flushStream, Boolean flushEncoder)
at System.IO.StreamWriter.Write(Char[] buffer)
at System.IO.TextWriter.WriteLine()
at System.IO.TextWriter.SyncTextWriter.WriteLine()
at Microsoft.DotNet.Cli.Utils.Reporter.WriteLine()
at Microsoft.DotNet.Tools.Run.RunCommand.EnsureProjectIsBuilt()
at Microsoft.DotNet.Tools.Run.RunCommand.Execute()
at Microsoft.DotNet.Tools.Run.RunCommand.Run(String[] args)
at Microsoft.DotNet.Cli.Program.ProcessArgs(String[] args, ITelemetry telemetryClient)
at Microsoft.DotNet.Cli.Program.Main(String[] args)
```

Avoid inadvertent disclosure information

.NET Core contributors and anyone else running a version of the .NET Core SDK that they built themselves should consider the path to their SDK source code. If a crash occurs while using a .NET Core SDK that is a custom debug build or configured with custom build symbol files, the SDK source file path from the build machine is collected as part of the stack trace and isn't hashed.

Because of this, custom builds of the .NET Core SDK shouldn't be located in directories whose path names expose personal or sensitive information.

See also

- .NET Core CLI Telemetry - 2019 Q2 Data
- Telemetry reference source (dotnet/cli repository)

global.json overview

9/19/2019 • 4 minutes to read • Edit Online

This article applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

The *global.json* file allows you to define which .NET Core SDK version is used when you run .NET Core CLI commands. Selecting the .NET Core SDK is independent from specifying the runtime your project targets. The .NET Core SDK version indicates which versions of the .NET Core CLI tools are used. In general, you want to use the latest version of the tools, so no *global.json* file is needed.

For more information about specifying the runtime instead, see Target frameworks.

.NET Core SDK looks for a *global.json* file in the current working directory (which isn't necessarily the same as the project directory) or one of its parent directories.

global.json schema

sdk

Type: Object

Specifies information about the .NET Core SDK to select.

version

Type: String

The version of the .NET Core SDK to use.

Note that this field:

- Doesn't have globbing support, that is, the full version number has to be specified.
- Doesn't support version ranges.

The following example shows the contents of a *global.json* file:

```
{
  "sdk": {
    "version": "2.2.100"
  }
}
```

global.json and the .NET Core CLI

It's helpful to know which versions are available in order to set one in the *global.json* file. You can find the full list of supported available SDKs at the Download .NET Core page. Starting with .NET Core 2.1 SDK, you can run the following command to verify which SDK versions are already installed on your machine:

```
dotnet --list-sdks
```

To install additional .NET Core SDK versions on your machine, visit the Download .NET Core page.

You can create a new the *global.json* file in the current directory by executing the dotnet new command, similar to the following example:

```
dotnet new globaljson --sdk-version 2.2.100
```

Matching rules

NOTE

The matching rules are governed by the apphost, which is part of the .NET Core runtime. The latest version of the host is used when you have multiple runtimes installed side-by-side.

Starting with .NET Core 2.0, the following rules apply when determining which version of the SDK to use:

- If no *global.json* file is found or *global.json* doesn't specify an SDK version, the latest installed SDK version is used. Latest SDK version can be either release or pre-release - the highest version number wins.
- If *global.json* does specify an SDK version:
  - If the specified SDK version is found on the machine, that exact version is used.
  - If the specified SDK version can't be found on the machine, the latest installed SDK **patch version** of that version is used. Latest installed SDK **patch version** can be either release or pre-release - the highest version number wins. In .NET Core 2.1 and higher, the **patch versions** lower than the **patch version** specified are ignored in the SDK selection.
  - If the specified SDK version and an appropriate SDK **patch version** can't be found, an error is thrown.

The SDK version is currently composed of the following parts:

`[.NET Core major version].[.NET Core minor version].[xyz][-optional preview name]`

The **feature release** of the .NET Core SDK is represented by the first digit (`x`) in the last portion of the number (`xyz`) for SDK versions 2.1.100 and higher. In general, the .NET Core SDK has a faster release cycle than .NET Core.

The **patch version** is defined by the last two digits (`yz`) in the last portion of the number (`xyz`) for SDK versions 2.1.100 and higher. For example, if you specify `2.1.300` as the SDK version, SDK selection finds up to `2.1.399` but `2.1.400` isn't considered a patch version for `2.1.300`.

.NET Core SDK versions `2.1.100` through `2.1.201` were released during the transition between version number schemes and don't correctly handle the `xyz` notation. We highly recommend if you specify these versions in the *global.json* file, that you ensure the specified versions are on the target machines.

With .NET Core SDK 1.x, if you specified a version and no exact match was found, the latest installed SDK version was used. Latest SDK version can be either release or pre-release - the highest version number wins.

Troubleshooting build warnings

WARNING

You are working with a preview version of the .NET Core SDK. You can define the SDK version via a global.json file in the current project. More at https://go.microsoft.com/fwlink/?linkid=869452

This warning indicates that your project is being compiled using a preview version of the .NET Core SDK, as explained in the Matching rules section. .NET Core SDK versions have a history and commitment of being high quality. However, if you don't want to use a preview version, add a *global.json* file to your project hierarchy structure to specify which SDK version to use, and use `dotnet --list-sdks` to confirm that the version is installed on your machine. When a new version is released, to use the new version, either remove the *global.json* file or update it to use the newer version.

> **WARNING**
>
> Startup project '{startupProject}' targets framework '.NETCoreApp' version '{targetFrameworkVersion}'. This version of the Entity Framework Core .NET Command-line Tools only supports version 2.0 or higher. For information on using older versions of the tools, see https://go.microsoft.com/fwlink/?linkid=871254

Starting with .NET Core 2.1 SDK (version 2.1.300), the `dotnet ef` command comes included in the SDK. This warning indicates that your project targets EF Core 1.0 or 1.1, which isn't compatible with .NET Core 2.1 SDK and later versions. To compile your project, install .NET Core 2.0 SDK (version 2.1.201) and earlier on your machine and define the desired SDK version using the *global.json* file. For more information about the `dotnet ef` command, see EF Core .NET Command-line Tools.

See also

- How project SDKs are resolved

dotnet command

9/19/2019 • 9 minutes to read • Edit Online

This article applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

Name

`dotnet` - A tool for managing .NET source code and binaries.

Synopsis

- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

```
dotnet [command] [arguments] [--additional-deps] [--additionalprobingpath] [--depsfile]
    [-d|--diagnostics] [--fx-version] [-h|--help] [--info] [--list-runtimes] [--list-sdks] [--roll-forward-
on-no-candidate-fx] [--runtimeconfig] [-v|--verbosity] [--version]
```

Description

`dotnet` is a tool for managing .NET source code and binaries. It exposes commands that perform specific tasks, such as `dotnet build` and `dotnet run`. Each command defines its own arguments. Type `--help` after each command to access brief help documentation.

`dotnet` can be used to run applications, by specifying an application DLL, such as `dotnet myapp.dll`. See .NET Core application deployment for to learn about deployment options.

Options

- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

`--additional-deps <PATH>`

Path to an additional *.deps.json* file.

`--additionalprobingpath <PATH>`

Path containing probing policy and assemblies to probe.

`--depsfile`

Path to a *deps.json* file.

A *deps.json* file contains a list of dependencies, compilation dependencies and version information used to address assembly conflicts. For more information about this file, see Runtime Configuration Files on GitHub.

`-d|--diagnostics`

Enables diagnostic output.

`--fx-version <VERSION>`

Version of the .NET Core runtime to use to run the application.

`-h|--help`

Prints out documentation for a given command, such as `dotnet build --help`. `dotnet --help` prints a list of available commands.

`--info`

Prints out detailed information about a .NET Core installation and the machine environment, such as the current operating system, and commit SHA of the .NET Core version.

`--list-runtimes`

Displays the installed .NET Core runtimes.

`--list-sdks`

Displays the installed .NET Core SDKs.

`--roll-forward-on-no-candidate-fx <N>`

Defines behavior when the required shared framework is not available. `N` can be:

- `0` - Disable even minor version roll forward.
- `1` - Roll forward on minor version, but not on major version. This is the default behavior.
- `2` - Roll forward on minor and major versions.

For more information, see Roll forward.

`--runtimeconfig`

Path to a *runtimeconfig.json* file.

A *runtimeconfig.json* file is a configuration file containing runtime configuration settings. For more information, see Runtime Configuration Files on GitHub.

`-v|--verbosity <LEVEL>`

Sets the verbosity level of the command. Allowed values are `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`. Not supported in every command; see specific command page to determine if this option is available.

`--version`

Prints out the version of the .NET Core SDK in use.

dotnet commands

General

- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

COMMAND	FUNCTION
dotnet build	Builds a .NET Core application.

COMMAND	FUNCTION
dotnet build-server	Interacts with servers started by a build.
dotnet clean	Clean build outputs.
dotnet help	Shows more detailed documentation online for the command.
dotnet migrate	Migrates a valid Preview 2 project to a .NET Core SDK 1.0 project.
dotnet msbuild	Provides access to the MSBuild command line.
dotnet new	Initializes a C# or F# project for a given template.
dotnet pack	Creates a NuGet package of your code.
dotnet publish	Publishes a .NET framework-dependent or self-contained application.
dotnet restore	Restores the dependencies for a given application.
dotnet run	Runs the application from source.
dotnet sln	Options to add, remove, and list projects in a solution file.
dotnet store	Stores assemblies in the runtime package store.
dotnet test	Runs tests using a test runner.

Project references

COMMAND	FUNCTION
dotnet add reference	Adds a project reference.
dotnet list reference	Lists project references.
dotnet remove reference	Removes a project reference.

NuGet packages

COMMAND	FUNCTION
dotnet add package	Adds a NuGet package.
dotnet remove package	Removes a NuGet package.

NuGet commands

COMMAND	FUNCTION
dotnet nuget delete	Deletes or unlists a package from the server.

COMMAND	FUNCTION
dotnet nuget locals	Clears or lists local NuGet resources such as http-request cache, temporary cache, or machine-wide global packages folder.
dotnet nuget push	Pushes a package to the server and publishes it.

Global Tools commands

.NET Core Global Tools are available starting with .NET Core SDK 2.1.300:

COMMAND	FUNCTION
dotnet tool install	Installs a Global Tool on your machine.
dotnet tool list	Lists all Global Tools currently installed in the default directory on your machine or in the specified path.
dotnet tool uninstall	Uninstalls a Global Tool from your machine.
dotnet tool update	Updates a Global Tool on your machine.

Additional tools

Starting with .NET Core SDK 2.1.300, a number of tools that were available only on a per project basis using `DotnetCliToolReference` are now available as part of the .NET Core SDK. These tools are listed in the following table:

TOOL	FUNCTION
dev-certs	Creates and manages development certificates.
ef	Entity Framework Core command-line tools.
sql-cache	SQL Server cache command-line tools.
user-secrets	Manages development user secrets.
watch	Starts a file watcher that runs a command when files change.

For more information about each tool, type `dotnet <tool-name> --help`.

Examples

Creates a new .NET Core console application:

```
dotnet new console
```

Restore dependencies for a given application:

```
dotnet restore
```

NOTE

Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

Build a project and its dependencies in a given directory:

`dotnet build`

Run an application DLL, such as `myapp.dll`:

`dotnet myapp.dll`

Environment variables

- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

`DOTNET_PACKAGES`

The global packages folder. If not set, it defaults to `~/.nuget/packages` on Unix or `%userprofile%\.nuget\packages` on Windows.

`DOTNET_SERVICING`

Specifies the location of the servicing index to use by the shared host when loading the runtime.

`DOTNET_CLI_TELEMETRY_OPTOUT`

Specifies whether data about the .NET Core tools usage is collected and sent to Microsoft. Set to `true` to opt-out of the telemetry feature (values `true`, `1`, or `yes` accepted). Otherwise, set to `false` to opt into the telemetry features (values `false`, `0`, or `no` accepted). If not set, the default is `false` and the telemetry feature is active.

`DOTNET_MULTILEVEL_LOOKUP`

Specifies whether .NET Core runtime, shared framework, or SDK are resolved from the global location. If not set, it defaults to `true`. Set to `false` to not resolve from the global location and have isolated .NET Core installations (values `0` or `false` are accepted). For more information about multi-level lookup, see Multi-level SharedFX Lookup.

`DOTNET_ROLL_FORWARD_ON_NO_CANDIDATE_FX`

Disables minor version roll forward, if set to `0`. For more information, see Roll forward.

See also

- Runtime Configuration Files

dotnet build

11/3/2019 • 5 minutes to read • Edit Online

This article applies to: ✓ .NET Core 1.x SDK and later versions

Name

`dotnet build` - Builds a project and all of its dependencies.

Synopsis

```
dotnet build [<PROJECT>|<SOLUTION>] [-c|--configuration] [-f|--framework] [--force]
    [--interactive] [--no-dependencies] [--no-incremental] [--no-restore] [--nologo]
    [-o|--output] [-r|--runtime] [-v|--verbosity] [--version-suffix]

dotnet build [-h|--help]
```

Description

The `dotnet build` command 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. Depending on the project type and settings, other files may be included, such as:

- An executable that can be used to run the application, if the project type is an executable targeting .NET Core 3.0 or later.
- Symbol files used for debugging with a *.pdb* extension.
- A *.deps.json* file, which lists the dependencies of the application or library.
- A *.runtimeconfig.json* file, which specifies the shared runtime and its version for an application.
- Other libraries that the project depends on (via project references or NuGet package references).

For executable projects targeting versions earlier than .NET Core 3.0, library dependencies from NuGet are typically NOT copied to the output folder. They're resolved from the NuGet global packages folder at run time. With that in mind, the product of `dotnet build` isn't ready to be transferred to another machine to run. To create a version of the application that can be deployed, you need to publish it (for example, with the dotnet publish command). For more information, see .NET Core Application Deployment.

For executable projects targeting .NET Core 3.0 and later, library dependencies are copied to the output folder. This means that if there isn't any other publish-specific logic (such as Web projects have), the build output should be deployable.

Building requires the *project.assets.json* file, which lists the dependencies of your application. The file is created when `dotnet restore` is executed. Without the assets file in place, the tooling can't resolve reference assemblies, which results in errors. With .NET Core 1.x SDK, you needed to explicitly run `dotnet restore` before running `dotnet build`. Starting with .NET Core 2.0 SDK, `dotnet restore` runs implicitly when you run `dotnet build`. If you want to disable implicit restore when running the build command, you can pass the `--no-restore` option.

NOTE

Starting with .NET Core 2.0, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

This command also supports the `dotnet restore` options when passed in the long form (for example, `--source`). Short form options, such as `-s`, are not supported.

Whether the project is executable or not is determined by the `<OutputType>` property in the project file. The following example shows a project that produces executable code:

```
<PropertyGroup>
  <OutputType>Exe</OutputType>
</PropertyGroup>
```

To produce a library, omit the `<OutputType>` property or change its value to `Library`. The IL DLL for a library doesn't contain entry points and can't be executed.

MSBuild

`dotnet build` uses MSBuild to build the project, so it supports both parallel and incremental builds. For more information, see Incremental Builds.

In addition to its options, the `dotnet build` command accepts MSBuild options, such as `-p` for setting properties or `-l` to define a logger. For more information about these options, see the MSBuild Command-Line Reference. Or you can also use the dotnet msbuild command.

Running `dotnet build` is equivalent to running `dotnet msbuild -restore`; however, the default verbosity of the output is different.

Arguments

`PROJECT | SOLUTION`

The project or solution file to build. If a project or solution file isn't specified, MSBuild searches the current working directory for a file that has a file extension that ends in either *proj* or *sln* and uses that file.

Options

- `-c|--configuration {Debug|Release}`

  Defines the build configuration. The default for most projects is `Debug`, but you can override the build configuration settings in your project.

- `-f|--framework <FRAMEWORK>`

  Compiles for a specific framework. The framework must be defined in the project file.

- `--force`

  Forces all dependencies to be resolved even if the last restore was successful. Specifying this flag is the same as deleting the *project.assets.json* file. Available since .NET Core 2.0 SDK.

- `-h|--help`

  Prints out a short help for the command.

- `--interactive`

  Allows the command to stop and wait for user input or action. For example, to complete authentication. Available since .NET Core 3.0 SDK.

- `--no-dependencies`

  Ignores project-to-project (P2P) references and only builds the specified root project.

- `--no-incremental`

  Marks the build as unsafe for incremental build. This flag turns off incremental compilation and forces a clean rebuild of the project's dependency graph.

- `--no-restore`

  Doesn't execute an implicit restore during build. Available since .NET Core 2.0 SDK.

- `--nologo`

  Doesn't display the startup banner or the copyright message. Available since .NET Core 3.0 SDK.

- `-o|--output <OUTPUT_DIRECTORY>`

  Directory in which to place the built binaries. If not specified, the default path is `./bin/<configuration>/<framework>/`. For projects with multiple target frameworks (via the `TargetFrameworks` property), you also need to define `--framework` when you specify this option.

- `-r|--runtime <RUNTIME_IDENTIFIER>`

  Specifies the target runtime. For a list of Runtime Identifiers (RIDs), see the RID catalog.

- `-v|--verbosity <LEVEL>`

  Sets the MSBuild verbosity level. Allowed values are `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`. The default is `minimal`.

- `--version-suffix <VERSION_SUFFIX>`

  Sets the value of the `$(VersionSuffix)` property to use when building the project. This only works if the `$(Version)` property isn't set. Then, `$(Version)` is set to the `$(VersionPrefix)` combined with the `$(VersionSuffix)`, separated by a dash.

Examples

- Build a project and its dependencies:

  ```
  dotnet build
  ```

- Build a project and its dependencies using Release configuration:

  ```
  dotnet build --configuration Release
  ```

- Build a project and its dependencies for a specific runtime (in this example, Ubuntu 18.04):

  ```
  dotnet build --runtime ubuntu.18.04-x64
  ```

- Build the project and use the specified NuGet package source during the restore operation (.NET Core

2.0 SDK and later versions):

```
dotnet build --source c:\packages\mypackages
```

- Build the project and set version 1.2.3.4 as a build parameter using the `-p` MSBuild option:

```
dotnet build -p:Version=1.2.3.4
```

dotnet build-server

10/17/2019 • 2 minutes to read • Edit Online

This article applies to: ✓ .NET Core 2.1 SDK and later versions

Name

`dotnet build-server` - Interacts with servers started by a build.

Synopsis

```
dotnet build-server shutdown [--msbuild] [--razor] [--vbcscompiler]
dotnet build-server shutdown [-h|--help]
dotnet build-server [-h|--help]
```

Commands

- `shutdown`

  Shuts down build servers that are started from dotnet. By default, all servers are shut down.

Options

- `-h|--help`

  Prints out a short help for the command.

- `--msbuild`

  Shuts down the MSBuild build server.

- `--razor`

  Shuts down the Razor build server.

- `--vbcscompiler`

  Shuts down the VB/C# compiler build server.

dotnet clean

9/19/2019 • 2 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK and later versions

Name

`dotnet clean` - Cleans the output of a project.

Synopsis

```
dotnet clean [<PROJECT>|<SOLUTION>] [-c|--configuration] [-f|--framework] [--interactive]
    [--nologo] [-o|--output] [-r|--runtime] [-v|--verbosity]
dotnet clean [-h|--help]
```

Description

The `dotnet clean` command cleans the output of the previous build. It's implemented as an MSBuild target, so the project is evaluated when the command is run. Only the outputs created during the build are cleaned. Both intermediate (*obj*) and final output (*bin*) folders are cleaned.

Arguments

`PROJECT | SOLUTION`

The MSBuild project or solution to clean. If a project or solution file is not specified, MSBuild searches the current working directory for a file that has a file extension that ends in *proj* or *sln*, and uses that file.

Options

- `-c|--configuration {Debug|Release}`

  Defines the build configuration. The default value is `Debug`. This option is only required when cleaning if you specified it during build time.

- `-f|--framework <FRAMEWORK>`

  The framework that was specified at build time. The framework must be defined in the project file. If you specified the framework at build time, you must specify the framework when cleaning.

- `-h|--help`

  Prints out a short help for the command.

- `--interactive`

  Allows the command to stop and wait for user input or action. For example, to complete authentication. Available since .NET Core 3.0 SDK.

- `--nologo`

  Doesn't display the startup banner or the copyright message. Available since .NET Core 3.0 SDK.

- `-o|--output <OUTPUT_DIRECTORY>`

  The directory that contains the build artifacts to clean. Specify the `-f|--framework <FRAMEWORK>` switch with the output directory switch if you specified the framework when the project was built.

- `-r|--runtime <RUNTIME_IDENTIFIER>`

  Cleans the output folder of the specified runtime. This is used when a self-contained deployment was created. Option available since .NET Core 2.0 SDK.

- `-v|--verbosity <LEVEL>`

  Sets the MSBuild verbosity level. Allowed values are `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`. The default is `normal`.

Examples

- Clean a default build of the project:

  ```
  dotnet clean
  ```

- Clean a project built using the Release configuration:

  ```
  dotnet clean --configuration Release
  ```

dotnet help reference

9/19/2019 • 2 minutes to read • Edit Online

This article applies to: ✓ .NET Core 2.0 SDK and later versions

Name

`dotnet help` - Shows more detailed documentation online for the specified command.

Synopsis

`dotnet help <COMMAND_NAME> [-h|--help]`

Description

The `dotnet help` command opens up the reference page for more detailed information about the specified command at docs.microsoft.com.

Arguments

- `COMMAND_NAME`

  Name of the .NET Core CLI command. For a list of the valid CLI commands, see CLI commands.

Options

- `-h|--help`

  Prints out a short help for the command.

Examples

- Opens the documentation page for the dotnet new command:

  ```
  dotnet help new
  ```

dotnet migrate

9/19/2019 • 2 minutes to read • Edit Online

This article applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

Name

`dotnet migrate` - Migrates a Preview 2 .NET Core project to a .NET Core SDK-style project.

Synopsis

```
dotnet migrate [<SOLUTION_FILE|PROJECT_DIR>] [--format-report-file-json] [-r|--report-file] [-s|--skip-
project-references] [--skip-backup] [-t|--template-file] [-v|--sdk-package-version] [-x|--xproj-file]
dotnet migrate [-h|--help]
```

Description

The `dotnet migrate` command migrates a valid Preview 2 *project.json*-based project to a valid .NET Core SDK-style *csproj* project.

By default, the command migrates the root project and any project references that the root project contains. This behavior is disabled using the `--skip-project-references` option at runtime.

Migration can be performed on the following assets:

- A single project by specifying the *project.json* file to migrate.
- All of the directories specified in the *global.json* file by passing in a path to the *global.json* file.
- A *solution.sln* file, where it migrates the projects referenced in the solution.
- On all subdirectories of the given directory recursively.

The `dotnet migrate` command keeps the migrated *project.json* file inside a `backup` directory, which it creates if the directory doesn't exist. This behavior is overridden using the `--skip-backup` option.

By default, the migration operation outputs the state of the migration process to standard output (STDOUT). If you use the `--report-file <REPORT_FILE>` option, the output is saved to the file specify.

The `dotnet migrate` command only supports valid Preview 2 *project.json*-based projects. This means that you cannot use it to migrate DNX or Preview 1 *project.json*-based projects directly to MSBuild/csproj projects. You first need to manually migrate the project to a Preview 2 *project.json*-based project and then use the `dotnet migrate` command to migrate the project.

The `dotnet migrate` command is no longer available starting with .NET Core 3.0 SDK.

Arguments

`PROJECT_JSON/GLOBAL_JSON/SOLUTION_FILE/PROJECT_DIR`

The path to one of the following:

- a *project.json* file to migrate.
- a *global.json* file: the folders specified in *global.json* are migrated.

- a *solution.sln* file: the projects referenced in the solution are migrated.
- a directory to migrate: recursively searches for *project.json* files to migrate inside the specified directory.

Defaults to current directory if nothing is specified.

Options

`--format-report-file-json <REPORT_FILE>`

Output migration report file as JSON rather than user messages.

`-h|--help`

Prints out a short help for the command.

`-r|--report-file <REPORT_FILE>`

Output migration report to a file in addition to the console.

`-s|--skip-project-references [Debug|Release]`

Skip migrating project references. By default, project references are migrated recursively.

`--skip-backup`

Skip moving *project.json*, *global.json*, and **.xproj* to a `backup` directory after successful migration.

`-t|--template-file <TEMPLATE_FILE>`

Template csproj file to use for migration. By default, the same template as the one dropped by `dotnet new console` is used.

`-v|--sdk-package-version <VERSION>`

The version of the sdk package that's referenced in the migrated app. The default is the version of the SDK in `dotnet new`.

`-x|--xproj-file <FILE>`

The path to the xproj file to use. Required when there is more than one xproj in a project directory.

Examples

Migrate a project in the current directory and all of its project-to-project dependencies:

```
dotnet migrate
```

Migrate all projects that *global.json* file includes:

```
dotnet migrate path/to/global.json
```

Migrate only the current project and no project-to-project (P2P) dependencies. Also, use a specific SDK version:

```
dotnet migrate -s -v 1.0.0-preview4
```

dotnet msbuild

9/19/2019 • 2 minutes to read • Edit Online

This article applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

Name

`dotnet msbuild` - Builds a project and all of its dependencies.

Synopsis

`dotnet msbuild <msbuild_arguments> [-h]`

Description

The `dotnet msbuild` command allows access to a fully functional MSBuild.

The command has the exact same capabilities as the existing MSBuild command-line client for SDK-style project only. The options are all the same. For more information about the available options, see the MSBuild Command-Line Reference.

The dotnet build command is equivalent to `dotnet msbuild -restore -target:Build`. `dotnet build` is more commonly used for building projects, but `dotnet msbuild` gives you more control. For example, if you have a specific target you want to run (without running the build target), you probably want to use `dotnet msbuild`.

Examples

- Build a project and its dependencies:

  ```
  dotnet msbuild
  ```

- Build a project and its dependencies using Release configuration:

  ```
  dotnet msbuild -p:Configuration=Release
  ```

- Run the publish target and publish for the `osx.10.11-x64` RID:

  ```
  dotnet msbuild -t:Publish -p:RuntimeIdentifiers=osx.10.11-x64
  ```

- See the whole project with all targets included by the SDK:

  ```
  dotnet msbuild -pp
  ```

dotnet new

11/1/2019 • 23 minutes to read • Edit Online

This article applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

Name

`dotnet new` - Creates a new project, configuration file, or solution based on the specified template.

Synopsis

- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

```
dotnet new <TEMPLATE> [--dry-run] [--force] [-i|--install] [-lang|--language] [-n|--name] [--nuget-
source] [-o|--output] [-u|--uninstall] [Template options]
dotnet new <TEMPLATE> [-l|--list] [--type]
dotnet new [-h|--help]
```

Description

The `dotnet new` command provides a convenient way to initialize a valid .NET Core project.

The command calls the template engine to create the artifacts on disk based on the specified template and options.

Arguments

`TEMPLATE`

The template to instantiate when the command is invoked. Each template might have specific options you can pass. For more information, see Template options.

If the `TEMPLATE` value isn't an exact match on text in the **Templates** or **Short Name** column, a substring match is performed on those two columns.

- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

The command contains a default list of templates. Use `dotnet new -l` to obtain a list of the available templates. The following table shows the templates that come pre-installed with the .NET Core SDK 2.2.100. The default language for the template is shown inside the brackets.

TEMPLATES	SHORT NAME	LANGUAGE	TAGS
Console Application	`console`	[C#], F#, VB	Common/Console

TEMPLATES	SHORT NAME	LANGUAGE	TAGS
Class library	`classlib`	[C#], F#, VB	Common/Library
Unit Test Project	`mstest`	[C#], F#, VB	Test/MSTest
NUnit 3 Test Project	`nunit`	[C#], F#, VB	Test/NUnit
NUnit 3 Test Item	`nunit-test`	[C#], F#, VB	Test/NUnit
xUnit Test Project	`xunit`	[C#], F#, VB	Test/xUnit
Razor Page	`page`	[C#]	Web/ASP.NET
MVC ViewImports	`viewimports`	[C#]	Web/ASP.NET
MVC ViewStart	`viewstart`	[C#]	Web/ASP.NET
ASP.NET Core Empty	`web`	[C#], F#	Web/Empty
ASP.NET Core Web App (Model-View-Controller)	`mvc`	[C#], F#	Web/MVC
ASP.NET Core Web App	`webapp`, `razor`	[C#]	Web/MVC/Razor Pages
ASP.NET Core with Angular	`angular`	[C#]	Web/MVC/SPA
ASP.NET Core with React.js	`react`	[C#]	Web/MVC/SPA
ASP.NET Core with React.js and Redux	`reactredux`	[C#]	Web/MVC/SPA
Razor Class Library	`razorclasslib`	[C#]	Web/Razor/Library/Razor Class Library
ASP.NET Core Web API	`webapi`	[C#], F#	Web/WebAPI
global.json file	`globaljson`		Config
NuGet Config	`nugetconfig`		Config
Web Config	`webconfig`		Config
Solution File	`sln`		Solution

Options

- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

- .NET Core 1.x

`--dry-run`

Displays a summary of what would happen if the given command were run if it would result in a template creation.

`--force`

Forces content to be generated even if it would change existing files. This is required when the output directory already contains a project.

`-h|--help`

Prints out help for the command. It can be invoked for the `dotnet new` command itself or for any template, such as `dotnet new mvc --help`.

`-i|--install <PATH|NUGET_ID>`

Installs a source or template pack from the `PATH` or `NUGET_ID` provided. If you want to install a prerelease version of a template package, you need to specify the version in the format of `<package-name>::<package-version>`. By default, `dotnet new` passes * for the version, which represents the last stable package version. See an example at the Examples section.

For information on creating custom templates, see Custom templates for dotnet new.

`-l|--list`

Lists templates containing the specified name. If invoked for the `dotnet new` command, it lists the possible templates available for the given directory. For example if the directory already contains a project, it doesn't list all project templates.

`-lang|--language {C#|F#|VB}`

The language of the template to create. The language accepted varies by the template (see defaults in the arguments section). Not valid for some templates.

> **NOTE**
>
> Some shells interpret `#` as a special character. In those cases, you need to enclose the language parameter value, such as `dotnet new console -lang "F#"`.

`-n|--name <OUTPUT_NAME>`

The name for the created output. If no name is specified, the name of the current directory is used.

`--nuget-source`

Specifies a NuGet source to use during install.

`-o|--output <OUTPUT_DIRECTORY>`

Location to place the generated output. The default is the current directory.

`--type`

Filters templates based on available types. Predefined values are "project", "item", or "other".

`-u|--uninstall <PATH|NUGET_ID>`

Uninstalls a source or template pack at the `PATH` or `NUGET_ID` provided. When excluding the

`<PATH|NUGET_ID>` value, all currently installed template packs and their associated templates are displayed.

> **NOTE**
>
> To uninstall a template using a `PATH`, you need to fully qualify the path. For example, *C:/Users/<USER>/Documents/Templates/GarciaSoftware.ConsoleTemplate.CSharp* will work, but *./GarciaSoftware.ConsoleTemplate.CSharp* from the containing folder will not. Additionally, do not include a final terminating directory slash on your template path.

Template options

Each project template may have additional options available. The core templates have the following additional options:

- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

console

`--langVersion <VERSION_NUMBER>` - Sets the `LangVersion` property in the created project file. For example, use `--langVersion 7.3` to use C# 7.3. Not supported for F#.

`--no-restore` - Doesn't execute an implicit restore during project creation.

angular, react, reactredux

`--exclude-launch-settings` - Exclude *launchSettings.json* from the generated template.

`--no-restore` - Doesn't execute an implicit restore during project creation.

`--no-https` - Project doesn't require HTTPS. This option only applies if `IndividualAuth` or `OrganizationalAuth` are not being used.

razorclasslib

`--no-restore` - Doesn't execute an implicit restore during project creation.

classlib

`-f|--framework <FRAMEWORK>` - Specifies the framework to target. Values: `netcoreapp2.2` to create a .NET Core Class Library or `netstandard2.0` to create a .NET Standard Class Library. The default value is `netstandard2.0`.

`--langVersion <VERSION_NUMBER>` - Sets the `LangVersion` property in the created project file. For example, use `--langVersion 7.3` to use C# 7.3. Not supported for F#.

`--no-restore` - Doesn't execute an implicit restore during project creation.

mstest, xunit

`-p|--enable-pack` - Enables packaging for the project using dotnet pack.

`--no-restore` - Doesn't execute an implicit restore during project creation.

nunit

`-f|--framework <FRAMEWORK>` - Specifies the framework to target. The default value is `netcoreapp2.1`.

`-p|--enable-pack` - Enables packaging for the project using dotnet pack.

`--no-restore` - Doesn't execute an implicit restore during project creation.

page

`-na|--namespace <NAMESPACE_NAME>` - Namespace for the generated code. The default value is `MyApp.Namespace`.

`-np|--no-pagemodel` - Creates the page without a PageModel.

viewimports

`-na|--namespace <NAMESPACE_NAME>` - Namespace for the generated code. The default value is `MyApp.Namespace`.

web

`--exclude-launch-settings` - Exclude *launchSettings.json* from the generated template.

`--no-restore` - Doesn't execute an implicit restore during project creation.

`--no-https` - Project doesn't require HTTPS. This option only applies if `IndividualAuth` or `OrganizationalAuth` are not being used.

mvc, webapp

`-au|--auth <AUTHENTICATION_TYPE>` - The type of authentication to use. The possible values are:

- `None` - No authentication (Default).
- `Individual` - Individual authentication.
- `IndividualB2C` - Individual authentication with Azure AD B2C.
- `SingleOrg` - Organizational authentication for a single tenant.
- `MultiOrg` - Organizational authentication for multiple tenants.
- `Windows` - Windows authentication.

`--aad-b2c-instance <INSTANCE>` - The Azure Active Directory B2C instance to connect to. Use with `IndividualB2C` authentication. The default value is `https://login.microsoftonline.com/tfp/`.

`-ssp|--susi-policy-id <ID>` - The sign-in and sign-up policy ID for this project. Use with `IndividualB2C` authentication.

`-rp|--reset-password-policy-id <ID>` - The reset password policy ID for this project. Use with `IndividualB2C` authentication.

`-ep|--edit-profile-policy-id <ID>` - The edit profile policy ID for this project. Use with `IndividualB2C` authentication.

`--aad-instance <INSTANCE>` - The Azure Active Directory instance to connect to. Use with `SingleOrg` or `MultiOrg` authentication. The default value is `https://login.microsoftonline.com/`.

`--client-id <ID>` - The Client ID for this project. Use with `IndividualB2C`, `SingleOrg`, or `MultiOrg` authentication. The default value is `11111111-1111-1111-11111111111111111`.

`--domain <DOMAIN>` - The domain for the directory tenant. Use with `SingleOrg` or `IndividualB2C` authentication. The default value is `qualified.domain.name`.

`--tenant-id <ID>` - The TenantId ID of the directory to connect to. Use with `SingleOrg` authentication. The default value is `22222222-2222-2222-2222-222222222222`.

`--callback-path <PATH>` - The request path within the application's base path of the redirect URI. Use with `SingleOrg` or `IndividualB2C` authentication. The default value is `/signin-oidc`.

`-r|--org-read-access` - Allows this application read-access to the directory. Only applies to `SingleOrg` or `MultiOrg` authentication.

`--exclude-launch-settings` - Exclude *launchSettings.json* from the generated template.

`--no-https` - Project doesn't require HTTPS. `app.UseHsts` and `app.UseHttpsRedirection` aren't added to `Startup.Configure`. This option only applies if `Individual`, `IndividualB2C`, `SingleOrg`, or `MultiOrg` aren't being used.

`-uld|--use-local-db` - Specifies LocalDB should be used instead of SQLite. Only applies to `Individual` or `IndividualB2C` authentication.

`--no-restore` - Doesn't execute an implicit restore during project creation.

webapi

`-au|--auth <AUTHENTICATION_TYPE>` - The type of authentication to use. The possible values are:

- `None` - No authentication (Default).
- `IndividualB2C` - Individual authentication with Azure AD B2C.
- `SingleOrg` - Organizational authentication for a single tenant.
- `Windows` - Windows authentication.

`--aad-b2c-instance <INSTANCE>` - The Azure Active Directory B2C instance to connect to. Use with `IndividualB2C` authentication. The default value is `https://login.microsoftonline.com/tfp/`.

`-ssp|--susi-policy-id <ID>` - The sign-in and sign-up policy ID for this project. Use with `IndividualB2C` authentication.

`--aad-instance <INSTANCE>` - The Azure Active Directory instance to connect to. Use with `SingleOrg` authentication. The default value is `https://login.microsoftonline.com/`.

`--client-id <ID>` - The Client ID for this project. Use with `IndividualB2C` or `SingleOrg` authentication. The default value is `11111111-1111-1111-11111111111111111`.

`--domain <DOMAIN>` - The domain for the directory tenant. Use with `SingleOrg` or `IndividualB2C` authentication. The default value is `qualified.domain.name`.

`--tenant-id <ID>` - The TenantId ID of the directory to connect to. Use with `SingleOrg` authentication. The default value is `22222222-2222-2222-2222-222222222222`.

`-r|--org-read-access` - Allows this application read-access to the directory. Only applies to `SingleOrg` or `MultiOrg` authentication.

`--exclude-launch-settings` - Exclude *launchSettings.json* from the generated template.

`--no-https` - Project doesn't require HTTPS. `app.UseHsts` and `app.UseHttpsRedirection` aren't added to `Startup.Configure`. This option only applies if `Individual`, `IndividualB2C`, `SingleOrg`, or `MultiOrg` aren't being used.

`-uld|--use-local-db` - Specifies LocalDB should be used instead of SQLite. Only applies to `Individual` or `IndividualB2C` authentication.

`--no-restore` - Doesn't execute an implicit restore during project creation.

globaljson

`--sdk-version <VERSION_NUMBER>` - Specifies the version of the .NET Core SDK to use in the *global.json* file.

Examples

Create a C# console application project by specifying the template name:

```
dotnet new "Console Application"
```

Create an F# console application project in the current directory:

```
dotnet new console -lang F#
```

Create a .NET Standard class library project in the specified directory (available only with .NET Core SDK 2.0 or later versions):

```
dotnet new classlib -lang VB -o MyLibrary
```

Create a new ASP.NET Core C# MVC project in the current directory with no authentication:

```
dotnet new mvc -au None
```

Create a new xUnit project:

```
dotnet new xunit
```

List all templates available for MVC:

```
dotnet new mvc -l
```

List all templates matching the *we* substring. No exact match is found, so substring matching runs against both the short name and name columns.

```
dotnet new we -l
```

Attempt to invoke the template matching *ng*. If a single match can't be determined, list the templates that are partial matches.

```
dotnet new ng
```

Install version 2.0 of the Single Page Application templates for ASP.NET Core (command option available for .NET Core SDK 1.1 and later versions only):

```
dotnet new -i Microsoft.DotNet.Web.Spa.ProjectTemplates::2.0.0
```

Create a *global.json* in the current directory setting the SDK version to 2.0.0 (available only with .NET Core SDK 2.0 or later versions):

```
dotnet new globaljson --sdk-version 2.0.0
```

See also

- Custom templates for dotnet new
- Create a custom template for dotnet new
- dotnet/dotnet-template-samples GitHub repo
- Available templates for dotnet new

dotnet nuget delete

9/19/2019 • 2 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK and later versions

Name

`dotnet nuget delete` - Deletes or unlists a package from the server.

Synopsis

```
dotnet nuget delete [<PACKAGE_NAME> <PACKAGE_VERSION>] [--force-english-output] [--interactive] [-k|--api-key]
[--no-service-endpoint]
    [--non-interactive] [-s|--source]
dotnet nuget delete [-h|--help]
```

Description

The `dotnet nuget delete` command deletes or unlists a package from the server. For nuget.org, the action is to unlist the package.

Arguments

- `PACKAGE_NAME`

  Name/ID of the package to delete.

- `PACKAGE_VERSION`

  Version of the package to delete.

Options

- `--force-english-output`

  Forces the application to run using an invariant, English-based culture.

- `-h|--help`

  Prints out a short help for the command.

- `--interactive`

  Allows the command to block and requires manual action for operations like authentication. Option available since .NET Core 2.2 SDK.

- `-k|--api-key <API_KEY>`

  The API key for the server.

- `--no-service-endpoint`

  Doesn't append "api/v2/package" to the source URL. Option available since .NET Core 2.1 SDK.

- `--non-interactive`

  Doesn't prompt for user input or confirmations.

- `-s|--source <SOURCE>`

  Specifies the server URL. Supported URLs for nuget.org include `https://www.nuget.org`, `https://www.nuget.org/api/v3`, and `https://www.nuget.org/api/v2/package`. For private feeds, replace the host name (for example, `%hostname%/api/v3`).

Examples

- Deletes version 1.0 of package `Microsoft.AspNetCore.Mvc`:

  ```
  dotnet nuget delete Microsoft.AspNetCore.Mvc 1.0
  ```

- Deletes version 1.0 of package `Microsoft.AspNetCore.Mvc`, not prompting user for credentials or other input:

  ```
  dotnet nuget delete Microsoft.AspNetCore.Mvc 1.0 --non-interactive
  ```

dotnet nuget locals

11/14/2019 • 2 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK and later versions

Name

`dotnet nuget locals` - Clears or lists local NuGet resources.

Synopsis

```
dotnet nuget locals <CACHE_LOCATION> [(-c|--clear)|(-l|--list)] [--force-english-output]
dotnet nuget locals [-h|--help]
```

Description

The `dotnet nuget locals` command clears or lists local NuGet resources in the http-request cache, temporary cache, or machine-wide global packages folder.

Arguments

- `CACHE_LOCATION`

  The cache location to list or clear. It accepts one of the following values:

  - `all` - Indicates that the specified operation is applied to all cache types: http-request cache, global packages cache, and the temporary cache.
  - `http-cache` - Indicates that the specified operation is applied only to the http-request cache. The other cache locations aren't affected.
  - `global-packages` - Indicates that the specified operation is applied only to the global packages cache. The other cache locations aren't affected.
  - `temp` - Indicates that the specified operation is applied only to the temporary cache. The other cache locations aren't affected.

Options

- `--force-english-output`

  Forces the application to run using an invariant, English-based culture.

- `-h|--help`

  Prints out a short help for the command.

- `-c|--clear`

  The clear option executes a clear operation on the specified cache type. The contents of the cache directories are deleted recursively. The executing user/group must have permission to the files in the cache directories. If not, an error is displayed indicating the files/folders that weren't cleared.

- `-l|--list`

The list option is used to display the location of the specified cache type.

Examples

- Displays the paths of all the local cache directories (http-cache directory, global-packages cache directory, and temporary cache directory):

  ```
  dotnet nuget locals all –l
  ```

- Displays the path for the local http-cache directory:

  ```
  dotnet nuget locals http-cache --list
  ```

- Clears all files from all local cache directories (http-cache directory, global-packages cache directory, and temporary cache directory):

  ```
  dotnet nuget locals all --clear
  ```

- Clears all files in local global-packages cache directory:

  ```
  dotnet nuget locals global-packages -c
  ```

- Clears all files in local temporary cache directory:

  ```
  dotnet nuget locals temp -c
  ```

Troubleshooting

For information on common problems and errors while using the `dotnet nuget locals` command, see Managing the NuGet cache.

dotnet nuget push

9/19/2019 • 2 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK and later versions

Name

`dotnet nuget push` - Pushes a package to the server and publishes it.

Synopsis

```
dotnet nuget push [<ROOT>] [-d|--disable-buffering] [--force-english-output] [--interactive] [-k|--api-key] [-
n|--no-symbols]
    [--no-service-endpoint] [-s|--source] [-sk|--symbol-api-key] [-ss|--symbol-source] [-t|--timeout]
dotnet nuget push [-h|--help]
```

Description

The `dotnet nuget push` command pushes a package to the server and publishes it. The push command uses server and credential details found in the system's NuGet config file or chain of config files. For more information on config files, see Configuring NuGet Behavior. NuGet's default configuration is obtained by loading *%AppData%\NuGet\NuGet.config* (Windows) or *$HOME/.local/share* (Linux/macOS), then loading any *nuget.config* or *.nuget\nuget.config* starting from the root of drive and ending in the current directory.

Arguments

- `ROOT`

  Specifies the file path to the package to be pushed.

Options

- `-d|--disable-buffering`

  Disables buffering when pushing to an HTTP(S) server to reduce memory usage.

- `--force-english-output`

  Forces the application to run using an invariant, English-based culture.

- `-h|--help`

Prints out a short help for the command.

- `--interactive`

  Allows the command to block and requires manual action for operations like authentication. Option available since .NET Core 2.2 SDK.

- `-k|--api-key <API_KEY>`

  The API key for the server.

- `-n|--no-symbols`

  Doesn't push symbols (even if present).

- `--no-service-endpoint`

  Doesn't append "api/v2/package" to the source URL. Option available since .NET Core 2.1 SDK.

- `-s|--source <SOURCE>`

  Specifies the server URL. This option is required unless `DefaultPushSource` config value is set in the NuGet config file.

- `-sk|--symbol-api-key <API_KEY>`

  The API key for the symbol server.

- `-ss|--symbol-source <SOURCE>`

  Specifies the symbol server URL.

- `-t|--timeout <TIMEOUT>`

  Specifies the timeout for pushing to a server in seconds. Defaults to 300 seconds (5 minutes). Specifying 0 (zero seconds) applies the default value.

Examples

- Pushes *foo.nupkg* to the default push source, specifying an API key:

  ```
  dotnet nuget push foo.nupkg -k 4003d786-cc37-4004-bfdf-c4f3e8ef9b3a
  ```

- Push *foo.nupkg* to the custom push source `https://customsource`, specifying an API key:

  ```
  dotnet nuget push foo.nupkg -k 4003d786-cc37-4004-bfdf-c4f3e8ef9b3a -s https://customsource/
  ```

- Pushes *foo.nupkg* to the default push source:

  ```
  dotnet nuget push foo.nupkg
  ```

- Pushes *foo.symbols.nupkg* to the default symbols source:

  ```
  dotnet nuget push foo.symbols.nupkg
  ```

- Pushes *foo.nupkg* to the default push source, specifying a 360-second timeout:

  ```
  dotnet nuget push foo.nupkg --timeout 360
  ```

- Pushes all *.nupkg* files in the current directory to the default push source:

  ```
  dotnet nuget push *.nupkg
  ```

NOTE

If this command doesn't work, it might be due to a bug that existed in older versions of the SDK (.NET Core 2.1 SDK and earlier versions). To fix this, upgrade your SDK version or run the following command instead: `dotnet nuget push **/*.nupkg`

dotnet pack

9/19/2019 • 4 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK and later versions

Name

`dotnet pack` - Packs the code into a NuGet package.

Synopsis

```
dotnet pack [<PROJECT>|<SOLUTION>] [-c|--configuration] [--force] [--include-source] [--include-symbols] [--interactive]
    [--no-build] [--no-dependencies] [--no-restore] [--nologo] [-o|--output] [--runtime] [-s|--serviceable]
    [-v|--verbosity] [--version-suffix]
dotnet pack [-h|--help]
```

Description

The `dotnet pack` command builds the project and creates NuGet packages. The result of this command is a NuGet package (that is, a *.nupkg* file).

If you want to generate a package that contains the debug symbols, you have two options available:

- `--include-symbols` - it creates the symbols package.
- `--include-source` - it creates the symbols package with a `src` folder inside containing the source files.

NuGet dependencies of the packed project are added to the *.nuspec* file, so they're properly resolved when the package is installed. Project-to-project references aren't packaged inside the project. Currently, you must have a package per project if you have project-to-project dependencies.

By default, `dotnet pack` builds the project first. If you wish to avoid this behavior, pass the `--no-build` option. This option is often useful in Continuous Integration (CI) build scenarios where you know the code was previously built.

You can provide MSBuild properties to the `dotnet pack` command for the packing process. For more information, see NuGet metadata properties and the MSBuild Command-Line Reference. The Examples section shows how to use the MSBuild -p switch for a couple of different scenarios.

Web projects aren't packable by default. To override the default behavior, add the following property to your *.csproj* file:

```
<PropertyGroup>
   <IsPackable>true</IsPackable>
</PropertyGroup>
```

NOTE

Starting with .NET Core 2.0, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

This command also supports the `dotnet restore` options when passed in the long form (for example, `--source`). Short form options, such as `-s`, are not supported.

Arguments

`PROJECT | SOLUTION`

The project or solution to pack. It's either a path to a csproj file, a solution file, or to a directory. If not specified, the command searches the current directory for a project or solution file.

Options

- `-c|--configuration {Debug|Release}`

  Defines the build configuration. The default value is `Debug`.

- `--force`

  Forces all dependencies to be resolved even if the last restore was successful. Specifying this flag is the same as deleting the *project.assets.json* file. Option available since .NET Core 2.0 SDK.

- `-h|--help`

  Prints out a short help for the command.

- `--include-source`

  Includes the debug symbols NuGet packages in addition to the regular NuGet packages in the output directory. The sources files are included in the `src` folder within the symbols package.

- `--include-symbols`

  Includes the debug symbols NuGet packages in addition to the regular NuGet packages in the output directory.

- `--interactive`

  Allows the command to stop and wait for user input or action (for example, to complete authentication). Available since .NET Core 3.0 SDK.

- `--no-build`

  Doesn't build the project before packing. It also implicitly sets the `--no-restore` flag.

- `--no-dependencies`

  Ignores project-to-project references and only restores the root project. Option available since .NET Core 2.0 SDK.

- `--no-restore`

  Doesn't execute an implicit restore when running the command. Option available since .NET Core 2.0 SDK.

- `--nologo`

  Doesn't display the startup banner or the copyright message. Available since .NET Core 3.0 SDK.

- `-o|--output <OUTPUT_DIRECTORY>`

  Places the built packages in the directory specified.

- `--runtime <RUNTIME_IDENTIFIER>`

  Specifies the target runtime to restore packages for. For a list of Runtime Identifiers (RIDs), see the RID catalog. Option available since .NET Core 2.0 SDK.

- `-s|--serviceable`

  Sets the serviceable flag in the package. For more information, see .NET Blog: .NET 4.5.1 Supports Microsoft Security Updates for .NET NuGet Libraries.

- `--version-suffix <VERSION_SUFFIX>`

  Defines the value for the `$(VersionSuffix)` MSBuild property in the project.

- `-v|--verbosity <LEVEL>`

  Sets the verbosity level of the command. Allowed values are `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`.

Examples

- Pack the project in the current directory:

  ```
  dotnet pack
  ```

- Pack the `app1` project:

  ```
  dotnet pack ~/projects/app1/project.csproj
  ```

- Pack the project in the current directory and place the resulting packages into the `nupkgs` folder:

  ```
  dotnet pack --output nupkgs
  ```

- Pack the project in the current directory into the `nupkgs` folder and skip the build step:

  ```
  dotnet pack --no-build --output nupkgs
  ```

- With the project's version suffix configured as `<VersionSuffix>$(VersionSuffix)</VersionSuffix>` in the *.csproj* file, pack the current project and update the resulting package version with the given suffix:

  ```
  dotnet pack --version-suffix "ci-1234"
  ```

- Set the package version to `2.1.0` with the `PackageVersion` MSBuild property:

  ```
  dotnet pack -p:PackageVersion=2.1.0
  ```

- Pack the project for a specific target framework:

  ```
  dotnet pack -p:TargetFrameworks=net45
  ```

- Pack the project and use a specific runtime (Windows 10) for the restore operation (.NET Core SDK 2.0 and later versions):

  ```
  dotnet pack --runtime win10-x64
  ```

- Pack the project using a .nuspec file:

  ```
  dotnet pack ~/projects/app1/project.csproj -p:NuspecFile=~/projects/app1/project.nuspec -
  p:NuspecBasePath=~/projects/app1/nuget
  ```

dotnet publish

9/19/2019 • 6 minutes to read • Edit Online

This article applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

Name

`dotnet publish` - Packs the application and its dependencies into a folder for deployment to a hosting system.

Synopsis

- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

```
dotnet publish [<PROJECT>] [-c|--configuration] [-f|--framework] [--force] [--manifest] [--no-build] [--
no-dependencies]
    [--no-restore] [-o|--output] [-r|--runtime] [--self-contained] [-v|--verbosity] [--version-suffix]
dotnet publish [-h|--help]
```

Description

`dotnet publish` compiles the application, reads through its dependencies specified in the project file, and publishes the resulting set of files to a directory. The output includes the following assets:

- Intermediate Language (IL) code in an assembly with a *dll* extension.
- *.deps.json* file that includes all of the dependencies of the project.
- *.runtimeconfig.json* file that specifies the shared runtime that the application expects, as well as other configuration options for the runtime (for example, garbage collection type).
- The application's dependencies, which are copied from the NuGet cache into the output folder.

The `dotnet publish` command's output is ready for deployment to a hosting system (for example, a server, PC, Mac, laptop) for execution. It's the only officially supported way to prepare the application for deployment. Depending on the type of deployment that the project specifies, the hosting system may or may not have the .NET Core shared runtime installed on it. For more information, see .NET Core Application Deployment. For the directory structure of a published application, see Directory structure.

NOTE

Starting with .NET Core 2.0, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

This command also supports the `dotnet restore` options when passed in the long form (for example, `--source`). Short form options, such as `-s`, are not supported.

Arguments

`PROJECT`

The project to publish. It's either the path and filename of a C#, F#, or Visual Basic project file, or the path to a directory that contains a C#, F#, or Visual Basic project file. If not specified, it defaults to the current directory.

Options

- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

`-c|--configuration {Debug|Release}`

Defines the build configuration. The default value is `Debug`.

`-f|--framework <FRAMEWORK>`

Publishes the application for the specified target framework. You must specify the target framework in the project file.

`--force`

Forces all dependencies to be resolved even if the last restore was successful. Specifying this flag is the same as deleting the *project.assets.json* file.

`-h|--help`

Prints out a short help for the command.

`--manifest <PATH_TO_MANIFEST_FILE>`

Specifies one or several target manifests to use to trim the set of packages published with the app. The manifest file is part of the output of the `dotnet store` command. To specify multiple manifests, add a `--manifest` option for each manifest. This option is available starting with .NET Core 2.0 SDK.

`--no-build`

Doesn't build the project before publishing. It also implicitly sets the `--no-restore` flag.

`--no-dependencies`

Ignores project-to-project references and only restores the root project.

`--no-restore`

Doesn't execute an implicit restore when running the command.

`-o|--output <OUTPUT_DIRECTORY>`

Specifies the path for the output directory. If not specified, it defaults to *./bin/[configuration]/[framework]/publish/* for a framework-dependent deployment or *./bin/[configuration]/[framework]/[runtime]/publish/* for a self-contained deployment. If the path is relative, the output directory generated is relative to the project file location, not to the current working directory.

`--self-contained`

Publishes the .NET Core runtime with your application so the runtime doesn't need to be installed on the target machine. If a runtime identifier is specified, its default value is `true`. For more information about the different deployment types, see .NET Core application deployment.

`-r|--runtime <RUNTIME_IDENTIFIER>`

Publishes the application for a given runtime. This is used when creating a self-contained deployment (SCD). For a list of Runtime Identifiers (RIDs), see the RID catalog. Default is to publish a framework-dependent deployment (FDD).

`-v|--verbosity <LEVEL>`

Sets the verbosity level of the command. Allowed values are `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`.

`--version-suffix <VERSION_SUFFIX>`

Defines the version suffix to replace the asterisk (`*`) in the version field of the project file.

Examples

Publish the project in the current directory:

```
dotnet publish
```

Publish the application using the specified project file:

```
dotnet publish ~/projects/app1/app1.csproj
```

Publish the project in the current directory using the `netcoreapp1.1` framework:

```
dotnet publish --framework netcoreapp1.1
```

Publish the current application using the `netcoreapp1.1` framework and the runtime for `OS X 10.10` (you must list this RID in the project file).

```
dotnet publish --framework netcoreapp1.1 --runtime osx.10.11-x64
```

Publish the current application but don't restore project-to-project (P2P) references, just the root project during the restore operation (.NET Core SDK 2.0 and later versions):

```
dotnet publish --no-dependencies
```

See also

- Target frameworks
- Runtime IDentifier (RID) catalog

dotnet restore

10/11/2019 • 5 minutes to read • Edit Online

This article applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

Name

`dotnet restore` - Restores the dependencies and tools of a project.

Synopsis

- .NET Core 2.x
- .NET Core 1.x

```
dotnet restore [<ROOT>] [--configfile] [--disable-parallel] [--force] [--ignore-failed-sources] [--no-cache]
    [--no-dependencies] [--packages] [-r|--runtime] [-s|--source] [-v|--verbosity] [--interactive]
dotnet restore [-h|--help]
```

Description

The `dotnet restore` command uses NuGet to restore dependencies as well as project-specific tools that are specified in the project file. By default, the restoration of dependencies and tools are executed in parallel.

NOTE

Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

To restore the dependencies, NuGet needs the feeds where the packages are located. Feeds are usually provided via the *nuget.config* configuration file. A default configuration file is provided when the CLI tools are installed. You specify additional feeds by creating your own *nuget.config* file in the project directory. You also specify additional feeds per invocation at a command prompt.

For dependencies, you specify where the restored packages are placed during the restore operation using the `--packages` argument. If not specified, the default NuGet package cache is used, which is found in the `.nuget/packages` directory in the user's home directory on all operating systems. For example, */home/user1* on Linux or *C:\Users\user1* on Windows.

For project-specific tooling, `dotnet restore` first restores the package in which the tool is packed, and then proceeds to restore the tool's dependencies as specified in its project file.

nuget.config differences

The behavior of the `dotnet restore` command is affected by the settings in the *nuget.config* file, if present. For example, setting the `globalPackagesFolder` in *nuget.config* places the restored NuGet packages in the specified folder. This is an alternative to specifying the `--packages` option on the `dotnet restore` command. For more information, see the nuget.config reference.

There are three specific settings that `dotnet restore` ignores:

- bindingRedirects

  Binding redirects don't work with `<PackageReference>` elements and .NET Core only supports `<PackageReference>` elements for NuGet packages.

- solution

  This setting is Visual Studio specific and doesn't apply to .NET Core. .NET Core doesn't use a `packages.config` file and instead uses `<PackageReference>` elements for NuGet packages.

- trustedSigners

  This setting isn't applicable as NuGet doesn't yet support cross-platform verification of trusted packages.

Implicit `dotnet restore`

Starting with .NET Core 2.0, `dotnet restore` is run implicitly if necessary when you issue the following commands:

- `dotnet new`
- `dotnet build`
- `dotnet build-server`
- `dotnet run`
- `dotnet test`
- `dotnet publish`
- `dotnet pack`

In most cases, you no longer need to explicitly use the `dotnet restore` command.

Sometimes, it might be inconvenient to run `dotnet restore` implicitly. For example, some automated systems, such as build systems, need to call `dotnet restore` explicitly to control when the restore occurs so that they can control network usage. To prevent `dotnet restore` from running implicitly, you can use the `--no-restore` flag with any of these commands to disable implicit restore.

Arguments

`ROOT`

Optional path to the project file to restore.

Options

- .NET Core 2.x
- .NET Core 1.x

`--configfile <FILE>`

The NuGet configuration file (*nuget.config*) to use for the restore operation.

`--disable-parallel`

Disables restoring multiple projects in parallel.

`--force`

Forces all dependencies to be resolved even if the last restore was successful. Specifying this flag is the same as deleting the *project.assets.json* file.

`-h|--help`

Prints out a short help for the command.

`--ignore-failed-sources`

Only warn about failed sources if there are packages meeting the version requirement.

`--no-cache`

Specifies to not cache packages and HTTP requests.

`--no-dependencies`

When restoring a project with project-to-project (P2P) references, restores the root project and not the references.

`--packages <PACKAGES_DIRECTORY>`

Specifies the directory for restored packages.

`-r|--runtime <RUNTIME_IDENTIFIER>`

Specifies a runtime for the package restore. This is used to restore packages for runtimes not explicitly listed in the `<RuntimeIdentifiers>` tag in the *.csproj* file. For a list of Runtime Identifiers (RIDs), see the RID catalog. Provide multiple RIDs by specifying this option multiple times.

`-s|--source <SOURCE>`

Specifies a NuGet package source to use during the restore operation. This setting overrides all of the sources specified in the *nuget.config* files. Multiple sources can be provided by specifying this option multiple times.

`--verbosity <LEVEL>`

Sets the verbosity level of the command. Allowed values are `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`. Default value is `minimal`.

`--interactive`

Allows the command to stop and wait for user input or action (for example to complete authentication). Since .NET Core 2.1.400.

Examples

Restore dependencies and tools for the project in the current directory:

```
dotnet restore
```

Restore dependencies and tools for the `app1` project found in the given path:

```
dotnet restore ~/projects/app1/app1.csproj
```

Restore the dependencies and tools for the project in the current directory using the file path provided as the source:

```
dotnet restore -s c:\packages\mypackages
```

Restore the dependencies and tools for the project in the current directory using the two file paths provided as sources:

```
dotnet restore -s c:\packages\mypackages -s c:\packages\myotherpackages
```

Restore dependencies and tools for the project in the current directory showing detailed output:

```
dotnet restore --verbosity detailed
```

dotnet run

11/3/2019 • 7 minutes to read • Edit Online

This article applies to: ✓ .NET Core 1.x SDK and later versions

Name

`dotnet run` - Runs source code without any explicit compile or launch commands.

Synopsis

- .NET Core 3.0
- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

```
dotnet run [-c|--configuration] [-f|--framework] [--force] [--interactive] [--launch-profile] [--no-build]
[--no-dependencies]
    [--no-launch-profile] [--no-restore] [-p|--project] [-r|--runtime] [-v|--verbosity] [[--] [application
arguments]]
dotnet run [-h|--help]
```

Description

The `dotnet run` command provides a convenient option 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, apply to `dotnet run` as well.

Output files are written into the default location, which is `bin/<configuration>/<target>`. For example if you have a `netcoreapp2.1` application and you run `dotnet run`, the output is placed in `bin/Debug/netcoreapp2.1`. Files are overwritten as needed. Temporary files are placed in the `obj` directory.

If the project specifies multiple frameworks, executing `dotnet run` results in an error unless the `-f|--framework <FRAMEWORK>` option is used to specify the framework.

The `dotnet run` command is used in the context of projects, not built assemblies. If you're trying to run a framework-dependent application DLL instead, you must use dotnet without a command. For example, to run `myapp.dll`, use:

```
dotnet myapp.dll
```

For more information on the `dotnet` driver, see the .NET Core Command Line Tools (CLI) topic.

To run the application, the `dotnet run` command resolves the dependencies of the application that are outside of the shared runtime from the NuGet cache. Because it uses cached dependencies, it's not recommended to use `dotnet run` to run applications in production. Instead, create a deployment using the `dotnet publish` command and deploy the published output.

NOTE

Starting with .NET Core 2.0, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

This command also supports the `dotnet restore` options when passed in the long form (for example, `--source`). Short form options, such as `-s`, are not supported.

Options

- .NET Core 3.0
- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

`--`

Delimits arguments to `dotnet run` from arguments for the application being run. All arguments after this delimiter are passed to the application run.

`-c|--configuration {Debug|Release}`

Defines the build configuration. The default value for most projects is `Debug`.

`-f|--framework <FRAMEWORK>`

Builds and runs the app using the specified framework. The framework must be specified in the project file.

`--force`

Forces all dependencies to be resolved even if the last restore was successful. Specifying this flag is the same as deleting the *project.assets.json* file.

`-h|--help`

Prints out a short help for the command.

`--interactive`

Allows the command to stop and wait for user input or action (for example, to complete authentication).

`--launch-profile <NAME>`

The name of the launch profile (if any) to use when launching the application. Launch profiles are defined in the *launchSettings.json* file and are typically called `Development`, `Staging`, and `Production`. For more information, see Working with multiple environments.

`--no-build`

Doesn't build the project before running. It also implicit sets the `--no-restore` flag.

`--no-dependencies`

When restoring a project with project-to-project (P2P) references, restores the root project and not the references.

`--no-launch-profile`

Doesn't try to use *launchSettings.json* to configure the application.

`--no-restore`

Doesn't execute an implicit restore when running the command.

`-p|--project <PATH>`

Specifies the path of the project file to run (folder name or full path). If not specified, it defaults to the current directory.

`--runtime <RUNTIME_IDENTIFIER>`

Specifies the target runtime to restore packages for. For a list of Runtime Identifiers (RIDs), see the RID catalog.

`-v|--verbosity <LEVEL>`

Sets the verbosity level of the command. Allowed values are `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`.

Examples

Run the project in the current directory:

```
dotnet run
```

Run the specified project:

```
dotnet run --project ./projects/proj1/proj1.csproj
```

Run the project in the current directory (the `--help` argument in this example is passed to the application, since the blank `--` option is used):

```
dotnet run --configuration Release -- --help
```

Restore dependencies and tools for the project in the current directory only showing minimal output and then run the project: (.NET Core SDK 2.0 and later versions):

```
dotnet run --verbosity m
```

dotnet sln

10/31/2019 • 2 minutes to read • Edit Online

This article applies to: ✓ .NET Core 1.x SDK and later versions

Name

`dotnet sln` - Modifies a .NET Core solution file.

Synopsis

```
dotnet sln [<SOLUTION_FILE>] [command] [-h|--help]
```

Description

The `dotnet sln` command provides a convenient way to add, remove, and list projects in a solution file.

To use the `dotnet sln` command, the solution file must already exist. If you need to create one, use the dotnet new command, like in the following example:

```
dotnet new sln
```

Arguments

- `SOLUTION_FILE`

  The solution file to use. If not specified, the command searches the current directory for one. If there are multiple solution files in the directory, one must be specified.

Options

- `-h|--help`

  Prints out a short help for the command.

Commands

`add`

Adds a project or multiple projects to the solution file.

Synopsis

```
dotnet sln [<SOLUTION_FILE>] add [--in-root] [-s|--solution-folder] <PROJECT_PATH>
dotnet sln add [-h|--help]
```

Arguments

- `SOLUTION_FILE`

  The solution file to use. If not specified, the command searches the current directory for one. If there are multiple solution files in the directory, one must be specified.

- `PROJECT_PATH`

  The path to the project to add to the solution. Add multiple projects by adding one after the other separated by spaces. Unix/Linux shell globbing pattern expansions are processed correctly by the `dotnet sln` command.

Options

- `-h|--help`

  Prints out a short help for the command.

- `--in-root`

  Places the projects in the root of the solution, rather than creating a solution folder. Available since .NET Core 3.0 SDK.

- `-s|--solution-folder`

  The destination solution folder path to add the projects to. Available since .NET Core 3.0 SDK.

`remove`

Removes a project or multiple projects from the solution file.

Synopsis

```
dotnet sln [<SOLUTION_FILE>] remove <PROJECT_PATH>
dotnet sln [<SOLUTION_FILE>] remove [-h|--help]
```

Arguments

- `SOLUTION_FILE`

  The solution file to use. If not specified, the command searches the current directory for one. If there are multiple solution files in the directory, one must be specified.

- `PROJECT_PATH`

  The path to the project to remove from the solution. Remove multiple projects by adding one after the other separated by spaces. Unix/Linux shell globbing pattern expansions are processed correctly by the `dotnet sln` command.

Options

- `-h|--help`

  Prints out a short help for the command.

`list`

Lists all projects in a solution file.

Synopsis

```
dotnet sln list [-h|--help]
```

Arguments

- `SOLUTION_FILE`

  The solution file to use. If not specified, the command searches the current directory for one. If there are multiple solution files in the directory, one must be specified.

Options

- `-h|--help`

  Prints out a short help for the command.

Examples

Add a C# project to a solution:

```
dotnet sln todo.sln add todo-app/todo-app.csproj
```

Remove a C# project from a solution:

```
dotnet sln todo.sln remove todo-app/todo-app.csproj
```

Add multiple C# projects to a solution:

```
dotnet sln todo.sln add todo-app/todo-app.csproj back-end/back-end.csproj
```

Remove multiple C# projects from a solution:

```
dotnet sln todo.sln remove todo-app/todo-app.csproj back-end/back-end.csproj
```

Add multiple C# projects to a solution using a globbing pattern (Unix/Linux only):

```
dotnet sln todo.sln add **/*.csproj
```

Remove multiple C# projects from a solution using a globbing pattern (Unix/Linux only):

```
dotnet sln todo.sln remove **/*.csproj
```

dotnet store

1/23/2019 • 2 minutes to read • Edit Online

This article applies to: ✓ .NET Core 2.x SDK

Name

`dotnet store` - Stores the specified assemblies in the runtime package store.

Synopsis

```
dotnet store -m|--manifest -f|--framework -r|--runtime [--framework-version] [-h|--help] [--output] [--skip-optimization] [--skip-symbols] [-v|--verbosity] [--working-dir]
```

Description

`dotnet store` stores the specified assemblies in the runtime package store. By default, assemblies are optimized for the target runtime and framework. For more information, see the runtime package store topic.

Required options

`-f|--framework <FRAMEWORK>`

Specifies the target framework.

`-m|--manifest <PATH_TO_MANIFEST_FILE>`

The *package store manifest file* is an XML file that contains the list of packages to store. The format of the manifest file is compatible with the SDK-style project format. So, a project file that references the desired packages can be used with the `-m|--manifest` option to store assemblies in the runtime package store. To specify multiple manifest files, repeat the option and path for each file. For example: `--manifest packages1.csproj --manifest packages2.csproj`.

`-r|--runtime <RUNTIME_IDENTIFIER>`

The runtime identifier to target.

Optional options

`--framework-version <FRAMEWORK_VERSION>`

Specifies the .NET Core SDK version. This option enables you to select a specific framework version beyond the framework specified by the `-f|--framework` option.

`-h|--help`

Shows help information.

`-o|--output <OUTPUT_DIRECTORY>`

Specifies the path to the runtime package store. If not specified, it defaults to the *store* subdirectory of the user profile .NET Core installation directory.

`--skip-optimization`

Skips the optimization phase.

`--skip-symbols`

Skips symbol generation. Currently, you can only generate symbols on Windows and Linux.

`-v|--verbosity <LEVEL>`

Sets the verbosity level of the command. Allowed values are `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`.

`-w|--working-dir <INTERMEDIATE_WORKING_DIRECTORY>`

The working directory used by the command. If not specified, it uses the *obj* subdirectory of the current directory.

Examples

Store the packages specified in the *packages.csproj* project file for .NET Core 2.0.0:

```
dotnet store --manifest packages.csproj --framework-version 2.0.0
```

Store the packages specified in the *packages.csproj* without optimization:

```
dotnet store --manifest packages.csproj --skip-optimization
```

See also

- Runtime package store

dotnet test

9/23/2019 • 6 minutes to read • Edit Online

This article applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

Name

`dotnet test` - .NET test driver used to execute unit tests.

Synopsis

- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

```
dotnet test [<PROJECT>] [-a|--test-adapter-path] [--blame] [-c|--configuration] [--collect] [-d|--diag] [-
f|--framework] [--filter]
    [-l|--logger] [--no-build] [--no-restore] [-o|--output] [-r|--results-directory] [-s|--settings] [-t|--
list-tests]
    [-v|--verbosity] [-- <RunSettings arguments>]

dotnet test [-h|--help]
```

Description

The `dotnet test` command is used to execute unit tests in a given project. The `dotnet test` command launches the test runner console application specified for a project. The test runner executes the tests defined for a unit test framework (for example, MSTest, NUnit, or xUnit) and reports the success or failure of each test. If all tests are successful, the test runner returns 0 as an exit code; otherwise if any test fails, it returns 1. The test runner and the unit test library are packaged as NuGet packages and are restored as ordinary dependencies for the project.

Test projects specify the test runner using an ordinary `<PackageReference>` element, as seen in the following sample project file:

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

  <PropertyGroup>
    <TargetFramework>netcoreapp2.2</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.4.0" />
    <PackageReference Include="xunit" Version="2.4.1" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.4.1" />
  </ItemGroup>

</Project>
```

Arguments

`PROJECT`

Path to the test project. If not specified, it defaults to current directory.

Options

- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

`-a|--test-adapter-path <PATH_TO_ADAPTER>`

Use the custom test adapters from the specified path in the test run.

`--blame`

Runs the tests in blame mode. This option is helpful in isolating the problematic tests causing test host to crash. It creates an output file in the current directory as *Sequence.xml* that captures the order of tests execution before the crash.

`-c|--configuration {Debug|Release}`

Defines the build configuration. The default value is `Debug`, but your project's configuration could override this default SDK setting.

`--collect <DATA_COLLECTOR_FRIENDLY_NAME>`

Enables data collector for the test run. For more information, see Monitor and analyze test run.

`-d|--diag <PATH_TO_DIAGNOSTICS_FILE>`

Enables diagnostic mode for the test platform and write diagnostic messages to the specified file.

`-f|--framework <FRAMEWORK>`

Looks for test binaries for a specific framework.

`--filter <EXPRESSION>`

Filters out tests in the current project using the given expression. For more information, see the Filter option details section. For more information and examples on how to use selective unit test filtering, see Running selective unit tests.

`-h|--help`

Prints out a short help for the command.

`-l|--logger <LoggerUri/FriendlyName>`

Specifies a logger for test results.

`--no-build`

Doesn't build the test project before running it. It also implicit sets the `--no-restore` flag.

`--no-restore`

Doesn't execute an implicit restore when running the command.

`-o|--output <OUTPUT_DIRECTORY>`

Directory in which to find the binaries to run.

`-r|--results-directory <PATH>`

The directory where the test results are going to be placed. If the specified directory doesn't exist, it's created.

`-s|--settings <SETTINGS_FILE>`

The `.runsettings` file to use for running the tests. Configure unit tests by using a `.runsettings` file.

`-t|--list-tests`

List all of the discovered tests in the current project.

`-v|--verbosity <LEVEL>`

Sets the verbosity level of the command. Allowed values are `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`.

`RunSettings arguments`

Arguments passed as RunSettings configurations for the test. Arguments are specified as `[name]=[value]` pairs after "-- " (note the space after --). A space is used to separate multiple `[name]=[value]` pairs.

Example: `dotnet test -- MSTest.DeploymentEnabled=false MSTest.MapInconclusiveToFailed=True`

For more information about RunSettings, see vstest.console.exe: Passing RunSettings args.

Examples

Run the tests in the project in the current directory:

`dotnet test`

Run the tests in the `test1` project:

`dotnet test ~/projects/test1/test1.csproj`

Run the tests in the project in the current directory and generate a test results file in the trx format:

`dotnet test --logger trx`

Filter option details

`--filter <EXPRESSION>`

`<Expression>` has the format `<property><operator><value>[|&<Expression>]`.

`<property>` is an attribute of the `Test Case`. The following are the properties supported by popular unit test frameworks:

TEST FRAMEWORK	SUPPORTED PROPERTIES
MSTest	• FullyQualifiedName • Name • ClassName • Priority • TestCategory
xUnit	• FullyQualifiedName • DisplayName • Traits

The `<operator>` describes the relationship between the property and the value:

OPERATOR	FUNCTION
`=`	Exact match
`!=`	Not exact match
`~`	Contains

`<value>` is a string. All the lookups are case insensitive.

An expression without an `<operator>` is automatically considered as a `contains` on `FullyQualifiedName` property (for example, `dotnet test --filter xyz` is same as `dotnet test --filter FullyQualifiedName~xyz`).

Expressions can be joined with conditional operators:

OPERATOR	FUNCTION
`|`	OR
`&`	AND

You can enclose expressions in parenthesis when using conditional operators (for example, `(Name~TestMethod1) | (Name~TestMethod2)`).

For more information and examples on how to use selective unit test filtering, see Running selective unit tests.

See also

- Frameworks and Targets
- .NET Core Runtime IDentifier (RID) catalog

dotnet tool install

9/19/2019 • 2 minutes to read • Edit Online

This article applies to: ✓ .NET Core 2.1 SDK

Name

`dotnet tool install` - Installs the specified .NET Core Global Tool on your machine.

Synopsis

```
dotnet tool install <PACKAGE_NAME> <-g|--global> [--add-source] [--configfile] [--framework] [-v|--
verbosity] [--version]
dotnet tool install <PACKAGE_NAME> <--tool-path> [--add-source] [--configfile] [--framework] [-v|--
verbosity] [--version]
dotnet tool install <-h|--help>
```

Description

The `dotnet tool install` command provides a way for you to install .NET Core Global Tools on your machine. To use the command, you either have to specify that you want a user-wide installation using the `--global` option or you specify a path to install it using the `--tool-path` option.

Global Tools are installed in the following directories by default when you specify the `-g` (or `--global`) option:

OS	PATH
Linux/macOS	`$HOME/.dotnet/tools`
Windows	`%USERPROFILE%\.dotnet\tools`

Arguments

`PACKAGE_NAME`

Name/ID of the NuGet package that contains the .NET Core Global Tool to install.

Options

`--add-source <SOURCE>`

Adds an additional NuGet package source to use during installation.

`--configfile <FILE>`

The NuGet configuration (*nuget.config*) file to use.

`--framework <FRAMEWORK>`

Specifies the target framework to install the tool for. By default, the .NET Core SDK tries to choose the most appropriate target framework.

`-g|--global`

Specifies that the installation is user wide. Can't be combined with the `--tool-path` option. If you don't specify this option, you must specify the `--tool-path` option.

`-h|--help`

Prints out a short help for the command.

`--tool-path <PATH>`

Specifies the location where to install the Global Tool. PATH can be absolute or relative. If PATH doesn't exist, the command tries to create it. Can't be combined with the `--global` option. If you don't specify this option, you must specify the `--global` option.

`-v|--verbosity <LEVEL>`

Sets the verbosity level of the command. Allowed values are `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`.

`--version <VERSION_NUMBER>`

The version of the tool to install. By default, the latest stable package version is installed. Use this option to install preview or older versions of the tool.

Examples

Installs the dotnetsay Global Tool in the default location:

```
dotnet tool install -g dotnetsay
```

Installs the dotnetsay Global Tool on a specific Windows folder:

```
dotnet tool install dotnetsay --tool-path c:\global-tools
```

Installs the dotnetsay Global Tool on a specific Linux/macOS folder:

```
dotnet tool install dotnetsay --tool-path ~/bin
```

Installs version 2.0.0 of the dotnetsay Global Tool:

```
dotnet tool install -g dotnetsay --version 2.0.0
```

See also

- .NET Core Global Tools

dotnet tool list

9/19/2019 • 2 minutes to read • Edit Online

This article applies to: ✓ .NET Core 2.1 SDK

Name

`dotnet tool list` - Lists all .NET Core Global Tools currently installed in the default directory on your machine or in the specified path.

Synopsis

```
dotnet tool list <-g|--global>
dotnet tool list <--tool-path>
dotnet tool list <-h|--help>
```

Description

The `dotnet tool list` command provides a way for you to list all .NET Core Global Tools installed user-wide on your machine (current user profile) or in the specified path. The command lists the package name, version installed, and the Global Tool command. To use the list command, you either have to specify that you want to see all user-wide tools using the `--global` option or specify a custom path using the `--tool-path` option.

Options

`-g|--global`

Lists user-wide Global Tools. Can't be combined with the `--tool-path` option. If you don't specify this option, you must specify the `--tool-path` option.

`-h|--help`

Prints out a short help for the command.

`--tool-path <PATH>`

Specifies a custom location where to find Global Tools. PATH can be absolute or relative. Can't be combined with the `--global` option. If you don't specify this option, you must specify the `--global` option.

Examples

Lists all Global Tools installed user-wide on your machine (current user profile):

`dotnet tool list -g`

Lists the Global Tools from a specific Windows folder:

`dotnet tool list --tool-path c:\global-tools`

Lists the Global Tools from a specific Linux/macOS folder:

`dotnet tool list --tool-path ~/bin`

See also

- .NET Core Global Tools

dotnet tool uninstall

9/19/2019 • 2 minutes to read • Edit Online

This article applies to: ✓ .NET Core 2.1 SDK

Name

`dotnet tool uninstall` - Uninstalls the specified .NET Core Global Tool from your machine.

Synopsis

```
dotnet tool uninstall <PACKAGE_NAME> <-g|--global>
dotnet tool uninstall <PACKAGE_NAME> <--tool-path>
dotnet tool uninstall <-h|--help>
```

Description

The `dotnet tool uninstall` command provides a way for you to uninstall .NET Core Global Tools from your machine. To use the command, you either have to specify that you want to remove a user-wide tool using the `--global` option or specify a path to where the tool is installed using the `--tool-path` option.

Arguments

`PACKAGE_NAME`

Name/ID of the NuGet package that contains the .NET Core Global Tool to uninstall. You can find the package name using the dotnet tool list command.

Options

`-g|--global`

Specifies that the tool to be removed is from a user-wide installation. Can't be combined with the `--tool-path` option. If you don't specify this option, you must specify the `--tool-path` option.

`-h|--help`

Prints out a short help for the command.

`--tool-path <PATH>`

Specifies the location where to uninstall the Global Tool. PATH can be absolute or relative. Can't be combined with the `--global` option. If you don't specify this option, you must specify the `--global` option.

Examples

Uninstalls the dotnetsay Global Tool:

`dotnet tool uninstall -g dotnetsay`

Uninstalls the dotnetsay Global Tool from a specific Windows folder:

```
dotnet tool uninstall dotnetsay --tool-path c:\global-tools
```

Uninstalls the dotnetsay Global Tool from a specific Linux/macOS folder:

```
dotnet tool uninstall dotnetsay --tool-path ~/bin
```

See also

- .NET Core Global Tools

dotnet tool update

9/19/2019 • 2 minutes to read • Edit Online

This article applies to: ✓ .NET Core 2.1 SDK

Name

`dotnet tool update` - Updates the specified .NET Core Global Tool on your machine.

Synopsis

```
dotnet tool update <PACKAGE_NAME> <-g|--global> [--configfile] [--framework] [-v|--verbosity]
dotnet tool update <PACKAGE_NAME> <--tool-path> [--configfile] [--framework] [-v|--verbosity]
dotnet tool update <-h|--help>
```

Description

The `dotnet tool update` command provides a way for you to update .NET Core Global Tools on your machine to the latest stable version of the package. The command uninstalls and reinstalls a tool, effectively updating it. To use the command, you either have to specify that you want to update a tool from a user-wide installation using the `--global` option or specify a path to where the tool is installed using the `--tool-path` option.

Arguments

`PACKAGE_NAME`

Name/ID of the NuGet package that contains the .NET Core Global Tool to update. You can find the package name using the dotnet tool list command.

Options

`--add-source <SOURCE>`

Adds an additional NuGet package source to use during installation.

`--configfile <FILE>`

The NuGet configuration (*nuget.config*) file to use.

`--framework <FRAMEWORK>`

Specifies the target framework to update the tool for.

`-g|--global`

Specifies that the update is for a user-wide tool. Can't be combined with the `--tool-path` option. If you don't specify this option, you must specify the `--tool-path` option.

`-h|--help`

Prints out a short help for the command.

`--tool-path <PATH>`

Specifies the location where the Global Tool is installed. PATH can be absolute or relative. Can't be combined with the `--global` option. If you don't specify this option, you must specify the `--global` option.

`-v|--verbosity <LEVEL>`

Sets the verbosity level of the command. Allowed values are `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`.

Examples

Updates the dotnetsay Global Tool:

```
dotnet tool update -g dotnetsay
```

Updates the dotnetsay Global Tool located on a specific Windows folder:

```
dotnet tool update dotnetsay --tool-path c:\global-tools
```

Updates the dotnetsay Global Tool located on a specific Linux/macOS folder:

```
dotnet tool update dotnetsay --tool-path ~/bin
```

See also

- .NET Core Global Tools

dotnet vstest

9/19/2019 • 6 minutes to read • Edit Online

This article applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

Name

`dotnet-vstest` - Runs tests from the specified files.

Synopsis

- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

```
dotnet vstest [<TEST_FILE_NAMES>] [--Settings|/Settings] [--Tests|/Tests] [--TestAdapterPath|/TestAdapterPath]
    [--Platform|/Platform] [--Framework|/Framework] [--Parallel|/Parallel] [--TestCaseFilter|/TestCaseFilter]
[--logger|/logger]
    [-lt|--ListTests|/lt|/ListTests] [--ParentProcessId|/ParentProcessId] [--Port|/Port] [--Diag|/Diag] [--
Blame|/Blame] [--InIsolation|/InIsolation]
    [[--] <args>...]] [-?|--Help|/?|/Help]
```

Description

The `dotnet-vstest` command runs the `VSTest.Console` command-line application to run automated unit tests.

Arguments

`TEST_FILE_NAMES`

Run tests from the specified assemblies. Separate multiple test assembly names with spaces.

Options

- .NET Core 2.1
- .NET Core 2.0
- .NET Core 1.x

`--Settings|/Settings:<Settings File>`

Settings to use when running tests.

`--Tests|/Tests:<Test Names>`

Run tests with names that match the provided values. Separate multiple values with commas.

`--TestAdapterPath|/TestAdapterPath`

Use custom test adapters from a given path (if any) in the test run.

`--Platform|/Platform:<Platform type>`

Target platform architecture used for test execution. Valid values are `x86`, `x64`, and `ARM`.

`--Framework|/Framework:<Framework Version>`

Target .NET Framework version used for test execution. Examples of valid values are `.NETFramework,Version=v4.6` or `.NETCoreApp,Version=v1.0`. Other supported values are `Framework40`, `Framework45`, `FrameworkCore10`, and `FrameworkUap10`.

`--Parallel|/Parallel`

Execute tests in parallel. By default, all available cores on the machine are available for use. Specify an explicit number of cores by setting the MaxCpuCount property under the RunConfiguration node in the runsettings file.

`--TestCaseFilter|/TestCaseFilter:<Expression>`

Run tests that match the given expression. `<Expression>` is of the format `<property>Operator<value>[|&<Expression>]`, where Operator is one of `=`, `!=`, or `~`. Operator `~` has 'contains' semantics and is applicable for string properties like `DisplayName`. Parenthesis `()` are used to group sub-expressions.

`-?|--Help|/?|/Help`

Prints out a short help for the command.

`--logger|/logger:<Logger Uri/FriendlyName>`

Specify a logger for test results.

- To publish test results to Team Foundation Server, use the `TfsPublisher` logger provider:

  ```
  /logger:TfsPublisher;
      Collection=<team project collection url>;
      BuildName=<build name>;
      TeamProject=<team project name>
      [;Platform=<Defaults to "Any CPU">]
      [;Flavor=<Defaults to "Debug">]
      [;RunTitle=<title>]
  ```

- To log results to a Visual Studio Test Results File (TRX), use the `trx` logger provider. This switch creates a file in the test results directory with given log file name. If `LogFileName` isn't provided, a unique file name is created to hold the test results.

  ```
  /logger:trx [;LogFileName=<Defaults to unique file name>]
  ```

`-lt|--ListTests|/lt|/ListTests:<File Name>`

Lists all discovered tests from the given test container.

`--ParentProcessId|/ParentProcessId:<ParentProcessId>`

Process ID of the parent process responsible for launching the current process.

`--Port|/Port:<Port>`

Specifies the port for the socket connection and receiving the event messages.

`--Diag|/Diag:<Path to log file>`

Enables verbose logs for the test platform. Logs are written to the provided file.

`--Blame|/Blame`

Runs the tests in blame mode. This option is helpful in isolating the problematic tests causing test host to crash. It creates an output file in the current directory as *Sequence.xml* that captures the order of tests execution before the crash.

`--InIsolation|/InIsolation`

Runs the tests in an isolated process. This makes *vstest.console.exe* process less likely to be stopped on an error in the tests, but tests may run slower.

`@<file>`

Reads response file for more options.

`args`

Specifies extra arguments to pass to the adapter. Arguments are specified as name-value pairs of the form `<n>=<v>`, where `<n>` is the argument name and `<v>` is the argument value. Use a space to separate multiple arguments.

Examples

Run tests in `mytestproject.dll`:

```
dotnet vstest mytestproject.dll
```

Run tests in `mytestproject.dll`, exporting to custom folder with custom name:

```
dotnet vstest mytestproject.dll --logger:"trx;LogFileName=custom_file_name.trx" --
ResultsDirectory:custom/file/path
```

Run tests in `mytestproject.dll` and `myothertestproject.exe`:

```
dotnet vstest mytestproject.dll myothertestproject.exe
```

Run `TestMethod1` tests:

```
dotnet vstest /Tests:TestMethod1
```

Run `TestMethod1` and `TestMethod2` tests:

```
dotnet vstest /Tests:TestMethod1,TestMethod2
```

dotnet-install scripts reference

9/10/2019 • 4 minutes to read • Edit Online

Name

`dotnet-install.ps1` | `dotnet-install.sh` - Script used to install the .NET Core CLI tools and the shared runtime.

Synopsis

Windows:

```
dotnet-install.ps1 [-Channel] [-Version] [-InstallDir] [-Architecture] [-SharedRuntime] [-Runtime] [-DryRun]
[-NoPath] [-Verbose] [-AzureFeed] [-UncachedFeed] [-NoCdn] [-FeedCredential] [-ProxyAddress] [-
ProxyUseDefaultCredentials] [-SkipNonVersionedFiles] [-Help]
```

macOS/Linux:

```
dotnet-install.sh [--channel] [--version] [--install-dir] [--architecture] [--runtime] [--dry-run] [--no-path]
[--verbose] [--azure-feed] [--uncached-feed] [--no-cdn] [--feed-credential] [--runtime-id] [--skip-non-
versioned-files] [--help]
```

Description

The `dotnet-install` scripts are used to perform a non-admin installation of the .NET Core SDK, which includes the .NET Core CLI tools and the shared runtime.

We recommend that you use the stable version that is hosted on .NET Core main website. The direct paths to the scripts are:

- https://dot.net/v1/dotnet-install.sh (bash, UNIX)
- https://dot.net/v1/dotnet-install.ps1 (Powershell, Windows)

The main usefulness of these scripts is in automation scenarios and non-admin installations. There are two scripts: one is a PowerShell script that works on Windows, and the other is a bash script that works on Linux/macOS. Both scripts have the same behavior. The bash script also reads PowerShell switches, so you can use PowerShell switches with the script on Linux/macOS systems.

The installation scripts download the ZIP/tarball file from the CLI build drops and proceed to install it in either the default location or in a location specified by `-InstallDir|--install-dir`. By default, the installation scripts download the SDK and install it. If you wish to only obtain the shared runtime, specify the `--runtime` argument.

By default, the script adds the install location to the $PATH for the current session. Override this default behavior by specifying the `--no-path` argument.

Before running the script, install the required dependencies.

You can install a specific version using the `--version` argument. The version must be specified as a three-part version (for example, 1.0.0-13232). If not provided, it uses the `latest` version.

Options

- `-Channel <CHANNEL>`

  Specifies the source channel for the installation. The possible values are:

  - `Current` - Most current release.
  - `LTS` - Long-Term Support channel (most current supported release).
  - Two-part version in X.Y format representing a specific release (for example, `2.0` or `1.0`).
  - Branch name. For example, `release/2.0.0`, `release/2.0.0-preview2`, or `master` (for nightly releases).

  The default value is `LTS`. For more information on .NET support channels, see the .NET Support Policy page.

- `-Version <VERSION>`

  Represents a specific build version. The possible values are:

  - `latest` - Latest build on the channel (used with the `-Channel` option).
  - `coherent` - Latest coherent build on the channel; uses the latest stable package combination (used with Branch name `-Channel` options).
  - Three-part version in X.Y.Z format representing a specific build version; supersedes the `-Channel` option. For example: `2.0.0-preview2-006120`.

  If not specified, `-Version` defaults to `latest`.

- `-InstallDir <DIRECTORY>`

  Specifies the installation path. The directory is created if it doesn't exist. The default value is *%LocalAppData%\Microsoft\dotnet*. Binaries are placed directly in this directory.

- `-Architecture <ARCHITECTURE>`

  Architecture of the .NET Core binaries to install. Possible values are `<auto>`, `amd64`, `x64`, `x86`, `arm64`, and `arm`. The default value is `<auto>`, which represents the currently running OS architecture.

- `-SharedRuntime`

  > **NOTE**
  >
  > This parameter is obsolete and may be removed in a future version of the script. The recommended alternative is the `Runtime` option.

  Installs just the shared runtime bits, not the entire SDK. This is equivalent to specifying `-Runtime dotnet`.

- `-Runtime <RUNTIME>`

  Installs just the shared runtime, not the entire SDK. The possible values are:

  - `dotnet` - the `Microsoft.NETCore.App` shared runtime.
  - `aspnetcore` - the `Microsoft.AspNetCore.App` shared runtime.

- `-DryRun`

  If set, the script won't perform the installation. Instead, it displays what command line to use to consistently install the currently requested version of the .NET Core CLI. For example, if you specify version `latest`, it displays a link with the specific version so that this command can be used deterministically in a build script. It also displays the binary's location if you prefer to install or download it yourself.

- `-NoPath`

  If set, the installation folder isn't exported to the path for the current session. By default, the script modifies the PATH, which makes the CLI tools available immediately after install.

- `-Verbose`

  Displays diagnostics information.

- `-AzureFeed`

  Specifies the URL for the Azure feed to the installer. We recommended that you don't change this value. The default value is `https://dotnetcli.azureedge.net/dotnet`.

- `-UncachedFeed`

  Allows changing the URL for the uncached feed used by this installer. We recommended that you don't change this value.

- `-NoCdn`

  Disables downloading from the Azure Content Delivery Network (CDN) and uses the uncached feed directly.

- `-FeedCredential`

  Used as a query string to append to the Azure feed. It allows changing the URL to use non-public blob storage accounts.

- `-ProxyAddress`

  If set, the installer uses the proxy when making web requests. (Only valid for Windows)

- `ProxyUseDefaultCredentials`

  If set, the installer uses the credentials of the current user when using proxy address. (Only valid for Windows)

- `-SkipNonVersionedFiles`

  Skips installing non-versioned files, such as *dotnet.exe*, if they already exist.

- `-Help`

  Prints out help for the script.

Examples

- Install the latest long-term supported (LTS) version to the default location:

  Windows:

  ```
  ./dotnet-install.ps1 -Channel LTS
  ```

  macOS/Linux:

  ```
  ./dotnet-install.sh --channel LTS
  ```

- Install the latest version from 2.0 channel to the specified location:

  Windows:

  ```
  ./dotnet-install.ps1 -Channel 2.0 -InstallDir C:\cli
  ```

macOS/Linux:

```
./dotnet-install.sh --channel 2.0 --install-dir ~/cli
```

- Install the 1.1.0 version of the shared runtime:

  Windows:

  ```
  ./dotnet-install.ps1 -Runtime dotnet -Version 1.1.0
  ```

  macOS/Linux:

  ```
  ./dotnet-install.sh --runtime dotnet --version 1.1.0
  ```

- Obtain script and install the 2.1.2 version behind a corporate proxy (Windows only):

  ```
  Invoke-WebRequest 'https://dot.net/v1/dotnet-install.ps1' -Proxy $env:HTTP_PROXY -
  ProxyUseDefaultCredentials -OutFile 'dotnet-install.ps1';
  ./dotnet-install.ps1 -InstallDir '~/.dotnet' -Version '2.1.2' -ProxyAddress $env:HTTP_PROXY -
  ProxyUseDefaultCredentials;
  ```

- Obtain script and install .NET Core CLI one-liner examples:

  Windows:

  ```
  @powershell -NoProfile -ExecutionPolicy unrestricted -Command "
  [Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12; &
  ([scriptblock]::Create((Invoke-WebRequest -UseBasicParsing 'https://dot.net/v1/dotnet-install.ps1')))
  <additional install-script args>"
  ```

  macOS/Linux:

  ```
  curl -sSL https://dot.net/v1/dotnet-install.sh | bash /dev/stdin <additional install-script args>
  ```

See also

- .NET Core releases
- .NET Core Runtime and SDK download archive

dotnet add reference

10/30/2019 • 2 minutes to read • Edit Online

This article applies to: ✓ .NET Core 1.x SDK and later versions

Name

`dotnet add reference` - Adds project-to-project (P2P) references.

Synopsis

```
dotnet add [<PROJECT>] reference [-f|--framework] <PROJECT_REFERENCES> [-h|--help] [--interactive]
```

Description

The `dotnet add reference` command provides a convenient option to add project references to a project. After running the command, the `<ProjectReference>` elements are added to the project file.

```
<ItemGroup>
  <ProjectReference Include="app.csproj" />
  <ProjectReference Include="..\lib2\lib2.csproj" />
  <ProjectReference Include="..\lib1\lib1.csproj" />
</ItemGroup>
```

Arguments

- `PROJECT`

  Specifies the project file. If not specified, the command searches the current directory for one.

- `PROJECT_REFERENCES`

  Project-to-project (P2P) references to add. Specify one or more projects. Glob patterns are supported on Unix/Linux-based systems.

Options

- `-h|--help`

  Prints out a short help for the command.

- `-f|--framework <FRAMEWORK>`

  Adds project references only when targeting a specific framework.

- `--interactive`

  Allows the command to stop and wait for user input or action (for example, to complete authentication). Available since .NET Core 3.0 SDK.

Examples

- Add a project reference:

```
dotnet add app/app.csproj reference lib/lib.csproj
```

- Add multiple project references to the project in the current directory:

```
dotnet add reference lib1/lib1.csproj lib2/lib2.csproj
```

- Add multiple project references using a globbing pattern on Linux/Unix:

```
dotnet add app/app.csproj reference **/*.csproj
```

dotnet list reference

9/19/2019 • 2 minutes to read • Edit Online

This topic applies to: ✓ .NET Core 1.x SDK and later versions

Name

`dotnet list reference` - Lists project-to-project references.

Synopsis

`dotnet list [<PROJECT>|<SOLUTION>] reference [-h|--help]`

Description

The `dotnet list reference` command provides a convenient option to list project references for a given project or solution.

Arguments

- `PROJECT | SOLUTION`

  Specifies the project or solution file to use for listing references. If not specified, the command searches the current directory for a project file.

Options

- `-h|--help`

  Prints out a short help for the command.

Examples

- List the project references for the specified project:

  ```
  dotnet list app/app.csproj reference
  ```

- List the project references for the project in the current directory:

  ```
  dotnet list reference
  ```

dotnet remove reference

12/10/2018 • 2 minutes to read • Edit Online

This article applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

Name

`dotnet remove reference` - Removes project-to-project references.

Synopsis

```
dotnet remove [<PROJECT>] reference [-f|--framework] <PROJECT_REFERENCES> [-h|--help]
```

Description

The `dotnet remove reference` command provides a convenient option to remove project references from a project.

Arguments

`PROJECT`

Target project file. If not specified, the command searches the current directory for one.

`PROJECT_REFERENCES`

Project-to-project (P2P) references to remove. You can specify one or multiple projects. Glob patterns are supported on Unix/Linux based terminals.

Options

`-h|--help`

Prints out a short help for the command.

`-f|--framework <FRAMEWORK>`

Removes the reference only when targeting a specific framework.

Examples

Remove a project reference from the specified project:

```
dotnet remove app/app.csproj reference lib/lib.csproj
```

Remove multiple project references from the project in the current directory:

```
dotnet remove reference lib1/lib1.csproj lib2/lib2.csproj
```

Remove multiple project references using a glob pattern on Unix/Linux:

```
dotnet remove app/app.csproj reference **/*.csproj
```

dotnet add package

9/19/2019 • 2 minutes to read • Edit Online

This article applies to: ✓ .NET Core 1.x SDK and later versions

Name

`dotnet add package` - Adds a package reference to a project file.

Synopsis

```
dotnet add [<PROJECT>] package <PACKAGE_NAME> [-h|--help] [-f|--framework] [--interactive] [-n|--no-restore]
[--package-directory] [-s|--source] [-v|--version]
```

Description

The `dotnet add package` command provides a convenient option to add a package reference to a project file. After running the command, there's a compatibility check to ensure the package is compatible with the frameworks in the project. If the check passes, a `<PackageReference>` element is added to the project file and dotnet restore is run.

NOTE

Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control the time at which the restore occurs.

For example, adding `Newtonsoft.Json` to *ToDo.csproj* produces output similar to the following example:

```
  Writing C:\Users\mairaw\AppData\Local\Temp\tmp95A8.tmp
info : Adding PackageReference for package 'Newtonsoft.Json' into project 'C:\projects\ToDo\ToDo.csproj'.
log  : Restoring packages for C:\Temp\projects\consoleproj\consoleproj.csproj...
info :   GET https://api.nuget.org/v3-flatcontainer/newtonsoft.json/index.json
info :   OK https://api.nuget.org/v3-flatcontainer/newtonsoft.json/index.json 79ms
info :   GET https://api.nuget.org/v3-flatcontainer/newtonsoft.json/12.0.1/newtonsoft.json.12.0.1.nupkg
info :   OK https://api.nuget.org/v3-flatcontainer/newtonsoft.json/12.0.1/newtonsoft.json.12.0.1.nupkg 232ms
log  : Installing Newtonsoft.Json 12.0.1.
info : Package 'Newtonsoft.Json' is compatible with all the specified frameworks in project
'C:\projects\ToDo\ToDo.csproj'.
info : PackageReference for package 'Newtonsoft.Json' version '12.0.1' added to file
'C:\projects\ToDo\ToDo.csproj'.
```

The *ToDo.csproj* file now contains a `<PackageReference>` element for the referenced package.

```
<PackageReference Include="Newtonsoft.Json" Version="12.0.1" />
```

Arguments

- `PROJECT`

  Specifies the project file. If not specified, the command searches the current directory for one.

- `PACKAGE_NAME`

  The package reference to add.

Options

- `-f|--framework <FRAMEWORK>`

  Adds a package reference only when targeting a specific framework.

- `-h|--help`

  Prints out a short help for the command.

- `--interactive`

  Allows the command to stop and wait for user input or action (for example, to complete authentication). Available since .NET Core 2.1 SDK, version 2.1.400 or later.

- `-n|--no-restore`

  Adds a package reference without performing a restore preview and compatibility check.

- `--package-directory <PACKAGE_DIRECTORY>`

  The directory where to restore the packages. The default package restore location is `%userprofile%\.nuget\packages` on Windows and `~/.nuget/packages` on macOS and Linux. For more information, see Managing the global packages, cache, and temp folders in NuGet.

- `-s|--source <SOURCE>`

  The NuGet package source to use during the restore operation.

- `-v|--version <VERSION>`

  Version of the package. See NuGet package versioning.

Examples

- Add `Newtonsoft.Json` NuGet package to a project:

  ```
  dotnet add package Newtonsoft.Json
  ```

- Add a specific version of a package to a project:

  ```
  dotnet add ToDo.csproj package Microsoft.Azure.DocumentDB.Core -v 1.0.0
  ```

- Add a package using a specific NuGet source:

  ```
  dotnet add package Microsoft.AspNetCore.StaticFiles -s https://dotnet.myget.org/F/dotnet-
  core/api/v3/index.json
  ```

See also

- Managing the global packages, cache, and temp folders in NuGet
- NuGet package versioning

dotnet list package

9/19/2019 • 3 minutes to read • Edit Online

This article applies to: ✓ .NET Core 2.2 SDK and later versions

Name

`dotnet list package` - Lists the package references for a project or solution.

Synopsis

```
dotnet list [<PROJECT>|<SOLUTION>] package [--config] [--framework] [--highest-minor] [--highest-patch]
   [--include-prerelease] [--include-transitive] [--interactive] [--outdated] [--source]
dotnet list package [-h|--help]
```

Description

The `dotnet list package` command provides a convenient option to list all NuGet package references for a specific project or a solution. You first need to build the project in order to have the assets needed for this command to process. The following example shows the output of the `dotnet list package` command for the SentimentAnalysis project:

```
Project 'SentimentAnalysis' has the following package references
   [netcoreapp2.1]:
   Top-level Package                Requested   Resolved
   > Microsoft.ML                   0.11.0      0.11.0
   > Microsoft.NETCore.App    (A)   [2.1.0, )   2.1.0

(A) : Auto-referenced package.
```

The **Requested** column refers to the package version specified in the project file and can be a range. The **Resolved** column lists the version that the project is currently using and is always a single value. The packages displaying an `(A)` right next to their names represent implicit package references that are inferred from your project settings (`Sdk` type, `<TargetFramework>` or `<TargetFrameworks>` property, etc.)

Use the `--outdated` option to find out if there are newer versions available of the packages you're using in your projects. By default, `--outdated` lists the latest stable packages unless the resolved version is also a prerelease version. To include prerelease versions when listing newer versions, also specify the `--include-prerelease` option. The following examples shows the output of the `dotnet list package --outdated --include-prerelease` command for the same project as the previous example:

```
The following sources were used:
   https://api.nuget.org/v3/index.json

Project `SentimentAnalysis` has the following updates to its packages
   [netcoreapp2.1]:
   Top-level Package      Requested   Resolved   Latest
   > Microsoft.ML         0.11.0      0.11.0     1.0.0-preview
```

If you need to find out whether your project has transitive dependencies, use the `--include-transitive` option.

Transitive dependencies occur when you add a package to your project that in turn relies on another package. The following example shows the output from running the `dotnet list package --include-transitive` command for the HelloPlugin project, which displays top-level packages and the packages they depend on:

```
Project 'HelloPlugin' has the following package references
   [netcoreapp3.0]:
   Top-level Package                      Requested                   Resolved
   > Microsoft.NETCore.Platforms    (A)   [3.0.0-preview3.19128.7, )   3.0.0-preview3.19128.7
   > Microsoft.WindowsDesktop.App   (A)   [3.0.0-preview3-27504-2, )   3.0.0-preview3-27504-2

   Transitive Package               Resolved
   > Microsoft.NETCore.Targets      2.0.0
   > PluginBase                     1.0.0

(A) : Auto-referenced package.
```

Arguments

`PROJECT | SOLUTION`

The project or solution file to operate on. If not specified, the command searches the current directory for one. If more than one solution or project is found, an error is thrown.

Options

- `--config <SOURCE>`

  The NuGet sources to use when searching for newer packages. Requires the `--outdated` option.

- `--framework <FRAMEWORK>`

  Displays only the packages applicable for the specified target framework. To specify multiple frameworks, repeat the option multiple times. For example: `--framework netcoreapp2.2 --framework netstandard2.0`.

- `-h|--help`

  Prints out a short help for the command.

- `--highest-minor`

  Considers only the packages with a matching major version number when searching for newer packages. Requires the `--outdated` option.

- `--highest-patch`

  Considers only the packages with a matching major and minor version numbers when searching for newer packages. Requires the `--outdated` option.

- `--include-prerelease`

  Considers packages with prerelease versions when searching for newer packages. Requires the `--outdated` option.

- `--include-transitive`

  Lists transitive packages, in addition to the top-level packages. When specifying this option, you get a list of packages that the top-level packages depend on.

- `--interactive`

Allows the command to stop and wait for user input or action. For example, to complete authentication. Available since .NET Core 3.0 SDK.

- `--outdated`

  Lists packages that have newer versions available.

- `-s|--source <SOURCE>`

  The NuGet sources to use when searching for newer packages. Requires the `--outdated` option.

Examples

- List package references of a specific project:

  ```
  dotnet list SentimentAnalysis.csproj package
  ```

- List package references that have newer versions available, including prerelease versions:

  ```
  dotnet list package --outdated --include-prerelease
  ```

- List package references for a specific target framework:

  ```
  dotnet list package --framework netcoreapp3.0
  ```

dotnet remove package

5/15/2019 • 2 minutes to read • Edit Online

This article applies to: ✓ .NET Core 1.x SDK ✓ .NET Core 2.x SDK

Name

`dotnet remove package` - Removes package reference from a project file.

Synopsis

```
dotnet remove [<PROJECT>] package <PACKAGE_NAME> [-h|--help]
```

Description

The `dotnet remove package` command provides a convenient option to remove a NuGet package reference from a project.

Arguments

`PROJECT`

Specifies the project file. If not specified, the command searches the current directory for one.

`PACKAGE_NAME`

The package reference to remove.

Options

`-h|--help`

Prints out a short help for the command.

Examples

Removes `Newtonsoft.Json` NuGet package from a project in the current directory:

```
dotnet remove package Newtonsoft.Json
```

.NET Core additional tools overview

10/22/2019 • 2 minutes to read • Edit Online

This section compiles a list of tools that support and extend the .NET Core functionality, in addition to the .NET Core command-line interface (CLI) tools.

WCF Web Service Reference tool

The WCF (Windows Communication Foundation) Web Service Reference is a Visual Studio connected service provider that made its debut in Visual Studio 2017 version 15.5. This tool retrieves metadata from a web service in the current solution, on a network location, or from a WSDL file, and generates a source file compatible with .NET Core, defining a WCF proxy class with methods that you can use to access the web service operations.

WCF dotnet-svcutil tool

The WCF (Windows Communication Foundation) dotnet-svcutil tool is a .NET Core CLI tool that retrieves metadata from a web service on a network location or from a WSDL file, and generates a source file compatible with .NET Core, defining a WCF proxy class with methods that you can use to access the web service operations. The **dotnet-svcutil** tool is an alternative option to the **WCF Web Service Reference** Visual Studio connected service provider, which first shipped with Visual Studio 2017 version 15.5. The **dotnet-svcutil** tool as a .NET Core CLI tool, is available cross-platform on Linux, macOS, and Windows.

WCF dotnet-svcutil.xmlserializer tool

On the .NET Framework, you can pre-generate a serialization assembly using the svcutil tool. The dotnet-svcutil.xmlserializer NuGet package provides similar functionality on .NET Core. It pre-generates C# serialization code for the types in the client application that are used by the WCF Service Contract and that can be serialized by the XmlSerializer. This improves the startup performance of XML serialization when serializing or deserializing objects of those types.

XML Serializer Generator

Like the Xml Serializer Generator (sgen.exe) for the .NET Framework, the Microsoft.XmlSerializer.Generator NuGet package is the solution for .NET Core and .NET Standard libraries. It creates an XML serialization assembly for types contained in an assembly to improve the startup performance of XML serialization when serializing or de-serializing objects of those types using XmlSerializer.

Use the WCF Web Service Reference Provider Tool

10/31/2019 • 3 minutes to read • Edit Online

Over the years, many Visual Studio developers have enjoyed the productivity that the **Add Service Reference** tool provided when their .NET Framework projects needed to access web services. The **WCF Web Service Reference** tool is a Visual Studio connected service extension that provides an experience like the Add Service Reference functionality for .NET Core and ASP.NET Core projects. This tool retrieves metadata from a web service in the current solution, on a network location, or from a WSDL file, and generates a .NET Core compatible source file containing Windows Communication Foundation (WCF) client proxy code that you can use to access the web service.

IMPORTANT

You should only reference services from a trusted source. Adding references from an untrusted source may compromise security.

Prerequisites

- Visual Studio 2017 version 15.5 or later versions

How to use the extension

NOTE

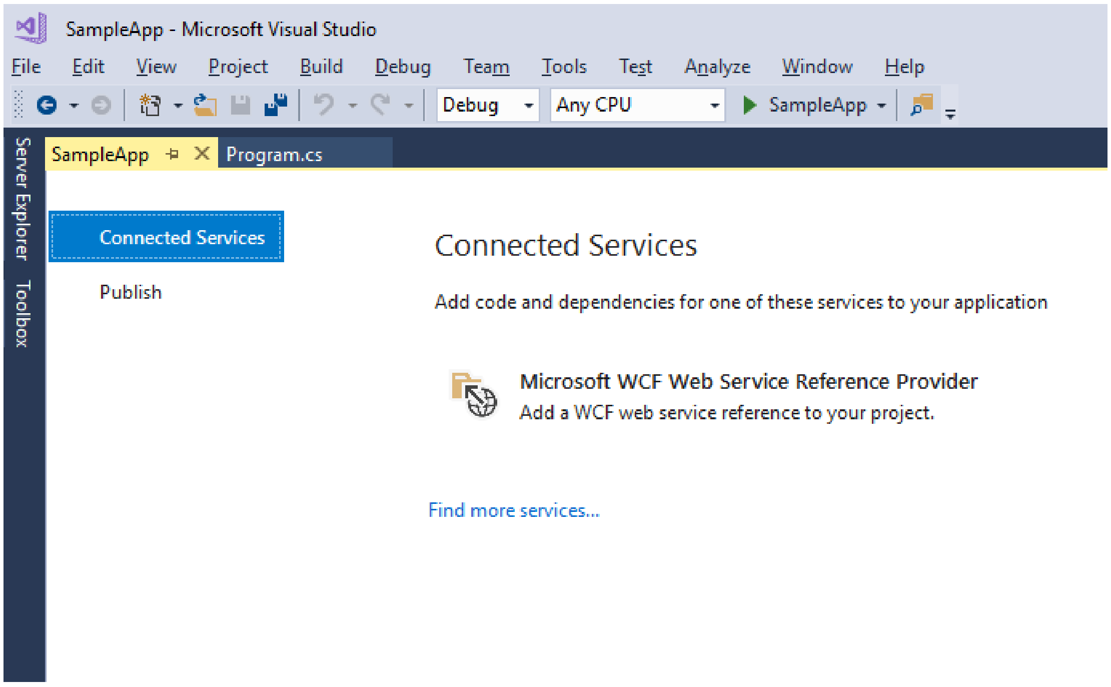
The **WCF Web Service Reference** option is applicable to projects created using the following project templates:

- **Visual C#** > **.NET Core**
- **Visual C#** > **.NET Standard**
- **Visual C#** > **Web** > **ASP.NET Core Web Application**

Using the **ASP.NET Core Web Application** project template as an example, this article walks you through adding a WCF service reference to the project:

1. In Solution Explorer, double-click the **Connected Services** node of the project (for a .NET Core or .NET Standard project this option is available when you right-click on the **Dependencies** node of the project in Solution Explorer).

   The **Connected Services** page appears as shown in the following image:


SampleApp - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Debug Any CPU SampleApp
Server Explorer Toolbox
SampleApp Program.cs
Connected Services
Publish

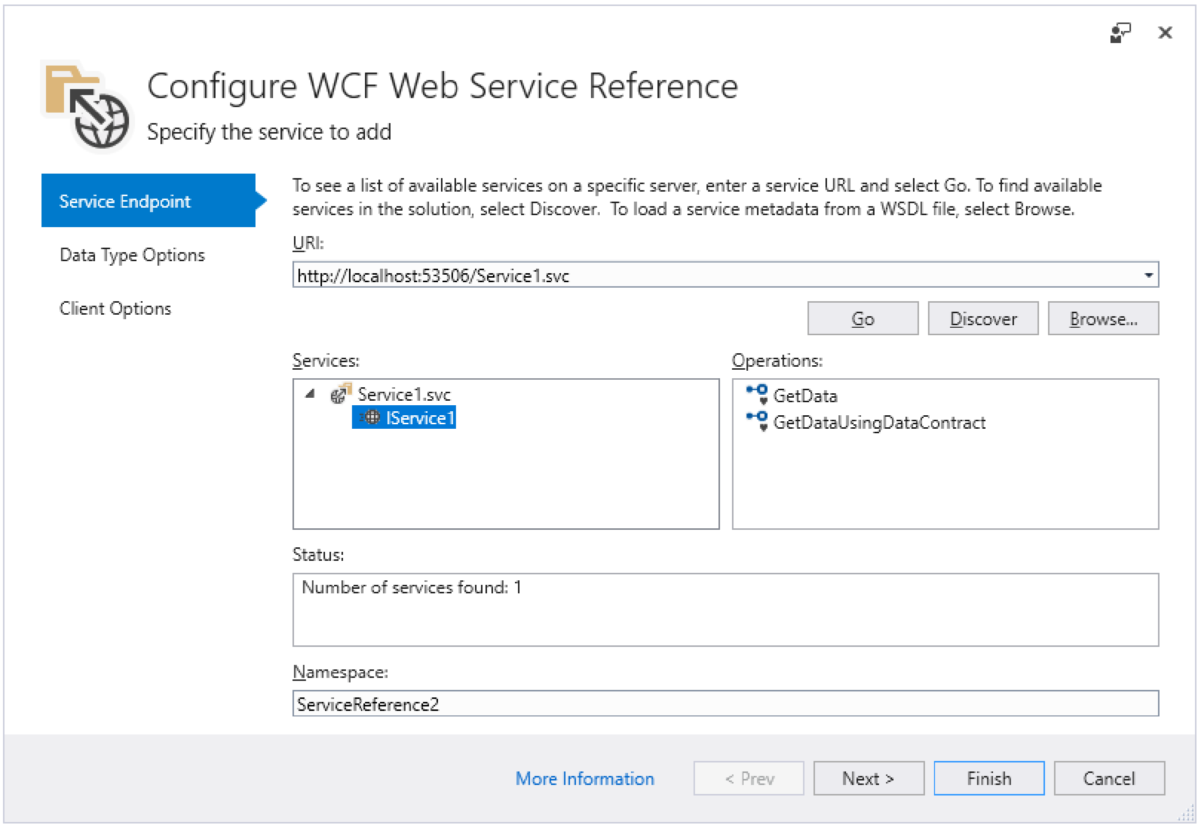
Connected Services

Add code and dependencies for one of these services to your application

Microsoft WCF Web Service Reference Provider
Add a WCF web service reference to your project.

Find more services...

2. On the **Connected Services** page, click **Microsoft WCF Web Service Reference Provider**. This brings up the **Configure WCF Web Service Reference** wizard:



Configure WCF Web Service Reference
Specify the service to add

To see a list of available services on a specific server, enter a service URL and select Go. To find available services in the solution, select Discover. To load a service metadata from a WSDL file, select Browse.

URI: http://localhost:53506/Service1.svc

Services: Service1.svc, IService1

Operations: GetData, GetDataUsingDataContract

Status: Number of services found: 1

Namespace: ServiceReference2

3. Select a service.

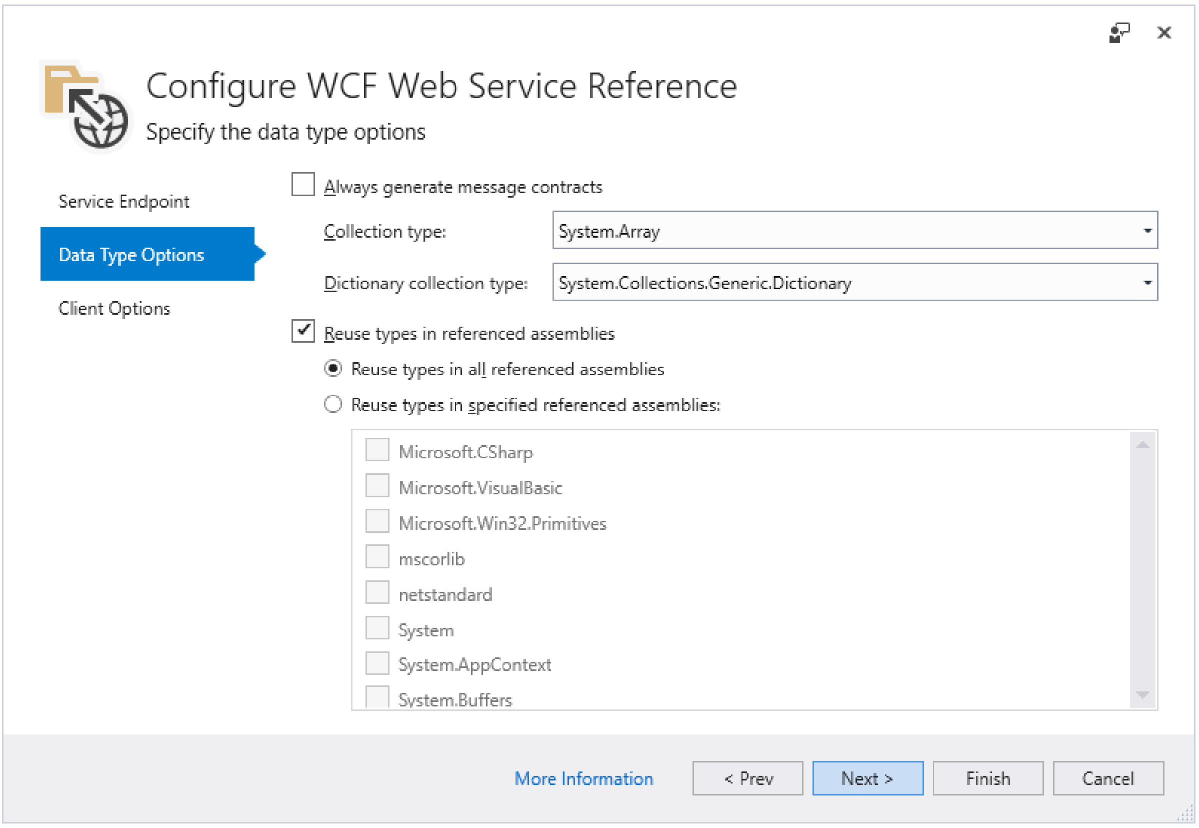
   3a. There are several services search options available within the **Configure WCF Web Service Reference** wizard:

   - To search for services defined in the current solution, click the **Discover** button.
   - To search for services hosted at a specified address, enter a service URL in the **Address** box and click the **Go** button.
   - To select a WSDL file that contains the web service metadata information, click the **Browse** button.

   3b. Select the service from the search results list in the **Services** box. If needed, enter the namespace for the generated code in the corresponding **Namespace** text box.

3c. Click the **Next** button to open the **Data Type Options** and the **Client Options** pages. Alternatively, click the **Finish** button to use the default options.

4. The **Data Type Options** form enables you to refine the generated service reference configuration settings:



Configure WCF Web Service Reference

Specify the data type options

- Service Endpoint
- Data Type Options
- Client Options

☐ Always generate message contracts

Collection type: System.Array

Dictionary collection type: System.Collections.Generic.Dictionary

☑ Reuse types in referenced assemblies

● Reuse types in all referenced assemblies

○ Reuse types in specified referenced assemblies:

☐ Microsoft.CSharp
☐ Microsoft.VisualBasic
☐ Microsoft.Win32.Primitives
☐ mscorlib
☐ netstandard
☐ System
☐ System.AppContext
☐ System.Buffers

NOTE

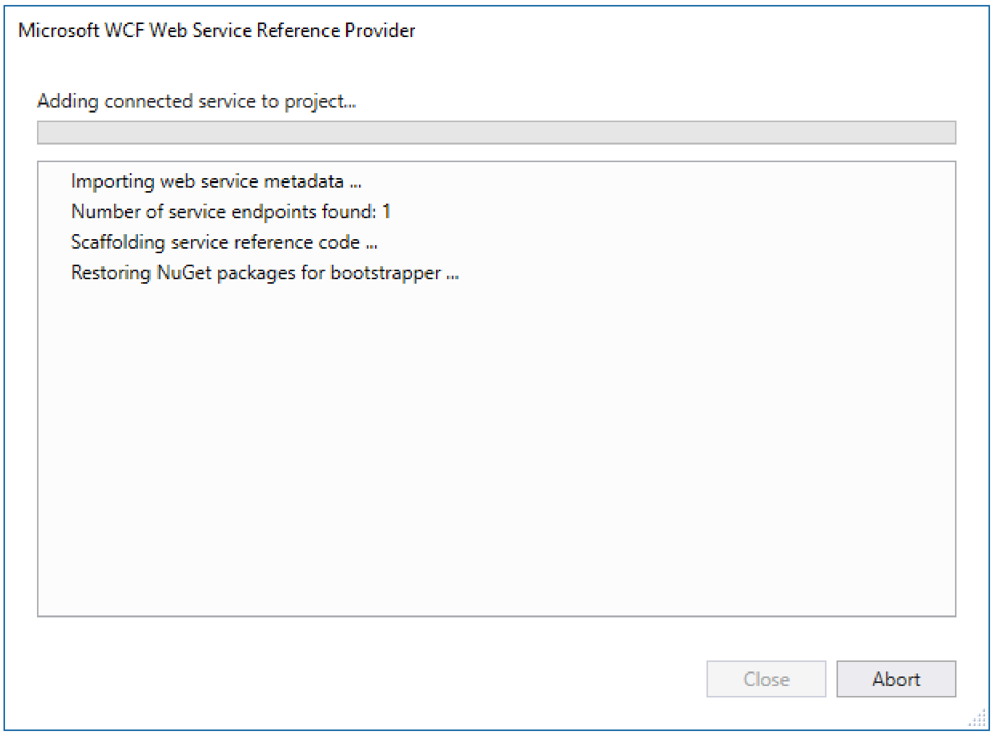
The **Reuse types in referenced assemblies** check box option is useful when data types needed for service reference code generation are defined in one of your project's referenced assemblies. It's important to reuse those existing data types to avoid compile-time type clash or runtime issues.

There may be a delay while type information is loaded, depending on the number of project dependencies and other system performance factors. The **Finish** button is disabled during loading unless the **Reuse types in referenced assemblies** check box is unchecked.

5. Click **Finish** when you are done.

While displaying progress, the tool:

- Downloads metadata from the WCF service.
- Generates the service reference code in a file named *reference.cs*, and adds it to your project under the **Connected Services** node.
- Updates the project file (.csproj) with NuGet package references required to compile and run on the target platform.


Microsoft WCF Web Service Reference Provider

Adding connected service to project...

```
Importing web service metadata ...
Number of service endpoints found: 1
Scaffolding service reference code ...
Restoring NuGet packages for bootstrapper ...
```

When these processes complete, you can create an instance of the generated WCF client type and invoke the service operations.

See also

- Get started with Windows Communication Foundation applications
- Windows Communication Foundation services and WCF data services in Visual Studio
- WCF supported features on .NET Core

Feedback & questions

If you have any questions or feedback, report it at Developer Community using the Report a problem tool.

Release notes

- Refer to the Release notes for updated release information, including known issues.

WCF dotnet-svcutil tool for .NET Core

10/22/2019 • 3 minutes to read • Edit Online

The Windows Communication Foundation (WCF) **dotnet-svcutil** tool is a .NET Core CLI tool that retrieves metadata from a web service on a network location or from a WSDL file, and generates a WCF class containing client proxy methods that access the web service operations.

Similar to the **Service Model Metadata - svcutil** tool for .NET Framework projects, the **dotnet-svcutil** is a command-line tool for generating a web service reference compatible with .NET Core and .NET Standard projects.

The **dotnet-svcutil** tool is an alternative option to the **WCF Web Service Reference** Visual Studio connected service provider that first shipped with Visual Studio 2017 version 15.5. The **dotnet-svcutil** tool as a .NET Core CLI tool, is available cross-platform on Linux, macOS, and Windows.

IMPORTANT

You should only reference services from a trusted source. Adding references from an untrusted source may compromise security.

Prerequisites

- dotnet-svcutil 2.x
- dotnet-svcutil 1.x

- .NET Core 2.1 SDK or later versions
- Your favorite code editor

Getting started

The following example walks you through the steps required to add a web service reference to a .NET Core web project and invoke the service. You'll create a .NET Core web application named *HelloSvcutil* and add a reference to a web service that implements the following contract:

```
[ServiceContract]
public interface ISayHello
{
    [OperationContract]
    string Hello(string name);
}
```

For this example, let's assume the web service will be hosted at the following address: `http://contoso.com/SayHello.svc`

From a Windows, macOS, or Linux command window perform the following steps:

1. Create a directory named *HelloSvcutil* for your project and make it your current directory, as in the following example:

   ```
   mkdir HelloSvcutil
   cd HelloSvcutil
   ```

2. Create a new C# web project in that directory using the `dotnet new` command as follows:

   ```
   dotnet new web
   ```

3. Install the `dotnet-svcutil` NuGet package as a CLI tool:

   - dotnet-svcutil 2.x
   - dotnet-svcutil 1.x

   ```
   dotnet tool install --global dotnet-svcutil
   ```

4. Run the *dotnet-svcutil* command to generate the web service reference file as follows:

   - dotnet-svcutil 2.x
   - dotnet-svcutil 1.x

   ```
   dotnet-svcutil http://contoso.com/SayHello.svc
   ```

The generated file is saved as *HelloSvcutil/ServiceReference/Reference.cs*. The *dotnet-svcutil* tool also adds to the project the appropriate WCF packages required by the proxy code as package references.

Using the Service Reference

1. Restore the WCF packages using the `dotnet restore` command as follows:

   ```
   dotnet restore
   ```

2. Find the name of the client class and operation you want to use. `Reference.cs` will contain a class that inherits from `System.ServiceModel.ClientBase`, with methods that can be used to call operations on the service. In this example, you want to call the *SayHello* service's *Hello* operation. `ServiceReference.SayHelloClient` is the name of the client class, and has a method called `HelloAsync` that can be used to call the operation.

3. Open the `Startup.cs` file in your editor, and add a using statement for the service reference namespace at the top:

   ```
   using ServiceReference;
   ```

4. Edit the `Configure` method to invoke the web service. You do this by creating an instance of the class that inherits from `ClientBase` and calling the method on the client object:

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

    app.Run(async (context) =>
    {
        var client = new SayHelloClient();
        var response = await client.HelloAsync();
        await context.Response.WriteAsync(response);
    });
}
```

5. Run the application using the `dotnet run` command as follows:

```
dotnet run
```

6. Navigate to the URL listed in the console (for example, `http://localhost:5000`) in your web browser.

You should see the following output: "Hello dotnet-svcutil!"

For a detailed description of the `dotnet-svcutil` tool parameters, invoke the tool passing the help parameter as follows:

- dotnet-svcutil 2.x
- dotnet-svcutil 1.x

```
dotnet-svcutil --help
```

Feedback & questions

If you have any questions or feedback, open an issue on GitHub. You can also review any existing questions or issues at the WCF repo on GitHub.

Release notes

- Refer to the Release notes for updated release information, including known issues.

Information

- dotnet-svcutil NuGet Package

Using dotnet-svcutil.xmlserializer on .NET Core

9/19/2019 • 2 minutes to read • Edit Online

The `dotnet-svcutil.xmlserializer` NuGet package can pre-generate a serialization assembly for .NET Core projects. It pre-generates C# serialization code for the types in the client application that are used by the WCF Service Contract and that can be serialized by the XmlSerializer. This improves the startup performance of XML serialization when serializing or deserializing objects of those types.

Prerequisites

- .NET Core 2.1 SDK or later
- Your favorite code editor

You can use the command `dotnet --info` to check which versions of .NET Core SDK and runtime you already have installed.

Getting started

To use `dotnet-svcutil.xmlserializer` in a .NET Core console application:

1. Create a WCF Service named 'MyWCFService' using the default template 'WCF Service Application' in .NET Framework. Add `[XmlSerializerFormat]` attribute on the service method like the following:

   ```
    [ServiceContract]
    public interface IService1
    {
        [XmlSerializerFormat]
        [OperationContract(Action = "http://tempuri.org/IService1/GetData", ReplyAction =
   "http://tempuri.org/IService1/GetDataResponse")]
        string GetData(int value);
    }
   ```

2. Create a .NET Core console application as WCF client application that targets at .NET Core 2.1 or later versions. For example, create an app named 'MyWCFClient' with the following command:

   ```
   dotnet new console --name MyWCFClient
   ```

   To ensure your project is targeting .NET Core 2.1 or later, inspect the `TargetFramework` XML element in your project file:

   ```
   <TargetFramework>netcoreapp2.1</TargetFramework>
   ```

3. Add a package reference to `System.ServiceModel.Http` by running the following command:

   ```
   dotnet add package System.ServiceModel.Http
   ```

4. Add the WCF Client code:

```
using System.ServiceModel;

    class Program
    {
        static void Main(string[] args)
        {
            var myBinding = new BasicHttpBinding();
            var myEndpoint = new EndpointAddress("http://localhost:2561/Service1.svc"); //Fill your
service url here
            var myChannelFactory = new ChannelFactory<IService1>(myBinding, myEndpoint);
            IService1 client = myChannelFactory.CreateChannel();
            string s = client.GetData(1);
            ((ICommunicationObject)client).Close();
        }
    }

[ServiceContract]
public interface IService1
{
    [XmlSerializerFormat]
    [OperationContract(Action = "http://tempuri.org/IService1/GetData", ReplyAction =
"http://tempuri.org/IService1/GetDataResponse")]
    string GetData(int value);
}
```

5. Add a reference to the `dotnet-svcutil.xmlserializer` package by running the following command:

   ```
   dotnet add package dotnet-svcutil.xmlserializer
   ```

   Running the command should add an entry to your project file similar to this:

   ```
   <ItemGroup>
     <DotNetCliToolReference Include="dotnet-svcutil.xmlserializer" Version="1.0.0" />
   </ItemGroup>
   ```

6. Build the application by running `dotnet build`. If everything succeeds, an assembly named *MyWCFClient.XmlSerializers.dll* is generated in the output folder. If the tool failed to generate the assembly, you'll see warnings in the build output.

7. Start the WCF service by, for example, running `http://localhost:2561/Service1.svc` in the browser. Then start the client application, and it will automatically load and use the pre-generated serializers at runtime.

Using Microsoft XML Serializer Generator on .NET Core

10/17/2019 • 2 minutes to read • Edit Online

This tutorial teaches you how to use the Microsoft XML Serializer Generator in a C# .NET Core application. During the course of this tutorial, you learn:

- How to create a .NET Core app
- How to add a reference to the Microsoft.XmlSerializer.Generator package
- How to edit your MyApp.csproj to add dependencies
- How to add a class and an XmlSerializer
- How to build and run the application

Like the Xml Serializer Generator (sgen.exe) for the .NET Framework, the Microsoft.XmlSerializer.Generator NuGet package is the equivalent for .NET Core and .NET Standard projects. It creates an XML serialization assembly for types contained in an assembly to improve the startup performance of XML serialization when serializing or de-serializing objects of those types using XmlSerializer.

Prerequisites

To complete this tutorial:

- .NET Core 2.1 SDK or later.
- Your favorite code editor.

TIP

Need to install a code editor? Try Visual Studio!

Use Microsoft XML Serializer Generator in a .NET Core console application

The following instructions show you how to use XML Serializer Generator in a .NET Core console application.

Create a .NET Core console application

Open a command prompt and create a folder named *MyApp*. Navigate to the folder you created and type the following command:

```
dotnet new console
```

Add a reference to the Microsoft.XmlSerializer.Generator package in the MyApp project

Use the `dotnet add package` command to add the reference in your project.

Type:

```
dotnet add package Microsoft.XmlSerializer.Generator -v 1.0.0
```

Verify changes to MyApp.csproj after adding the package

Open your code editor and let's get started! We're still working from the *MyApp* directory we built the app in.

Open *MyApp.csproj* in your text editor.

After running the `dotnet add package` command, the following lines are added to your *MyApp.csproj* project file:

```
<ItemGroup>
   <PackageReference Include="Microsoft.XmlSerializer.Generator" Version="1.0.0" />
</ItemGroup>
```

Add another ItemGroup section for .NET Core CLI Tool support

Add the following lines after the `ItemGroup` section that we inspected:

```
<ItemGroup>
   <DotNetCliToolReference Include="Microsoft.XmlSerializer.Generator" Version="1.0.0" />
</ItemGroup>
```

Add a class in the application

Open *Program.cs* in your text editor. Add the class named *MyClass* in *Program.cs*.

```
public class MyClass
{
   public int Value;
}
```

Create an `XmlSerializer` for MyClass

Add the following line inside *Main* to create an `XmlSerializer` for MyClass:

```
var serializer = new System.Xml.Serialization.XmlSerializer(typeof(MyClass));
```

Build and run the application

Still within the *MyApp* folder, run the application via `dotnet run` and it automatically loads and uses the pre-generated serializers at runtime.

Type the following command in your console window:

```
dotnet run
```

NOTE

`dotnet run` calls `dotnet build` to ensure that the build targets have been built, and then calls `dotnet <assembly.dll>` to run the target application.

IMPORTANT

The commands and steps shown in this tutorial to run your application are used during development time only. Once you're ready to deploy your app, take a look at the different deployment strategies for .NET Core apps and the `dotnet publish` command.

If everything succeeds, an assembly named *MyApp.XmlSerializers.dll* is generated in the output folder.

Congratulations! You have just:

- Created a .NET Core app.
- Added a reference to the Microsoft.XmlSerializer.Generator package.
- Edited your MyApp.csproj to add dependencies.
- Added a class and an XmlSerializer.
- Built and ran the application.

Related resources

- Introducing XML Serialization
- How to: Serialize Using XmlSerializer (C#)
- How to: Serialize Using XmlSerializer (Visual Basic)

Overview of the porting process from .NET Framework to .NET Core

10/29/2019 • 2 minutes to read • Edit Online

You might have code that currently runs on the .NET Framework that you're interested in porting to .NET Core. This article provides:

- An overview of the porting process.
- A list of the tools you may find helpful when you're porting your code to .NET Core.

Overview of the porting process

We recommend you to use the following process when porting your project to .NET Core:

1. Retarget all projects you wish to port to target the .NET Framework 4.7.2 or higher.

   This step ensures that you can use API alternatives for .NET Framework-specific targets when .NET Core doesn't support a particular API.

2. Use the .NET Portability Analyzer to analyze your assemblies and see if they're portable to .NET Core.

   The API Portability Analyzer tool analyzes your compiled assemblies and generates a report. This report shows a high-level portability summary and a breakdown of each API you're using that isn't available on NET Core.

3. Install the .NET API analyzer into your projects to identify APIs throwing PlatformNotSupportedException on some platforms and some other potential compatibility issues.

   This tool is similar to the portability analyzer, but instead of analyzing if things can build on .NET Core, it will analyze if you're using an API in a way that will throw the PlatformNotSupportedException at runtime. Although this isn't common if you're moving from .NET Framework 4.7.2 or higher, it's good to check.

4. Convert all of your `packages.config` dependencies to the PackageReference format with the conversion tool in Visual Studio.

   This step involves converting your dependencies from the legacy `packages.config` format. `packages.config` doesn't work on .NET Core, so this conversion is required if you have package dependencies.

5. Create new projects for .NET Core and copy over source files, or attempt to convert your existing project file with a tool.

   .NET Core uses a simplified (and different) project file format than .NET Framework. You'll need to convert your project files into this format to continue.

6. Port your test code.

   Because porting to .NET Core is such a significant change to your codebase, it's highly recommended to get your tests ported, so that you can run tests as you port your code over. MSTest, xUnit, and NUnit all work on .NET Core.

Additionally, you can attempt to port smaller solutions or individual projects to the .NET Core project file format with the dotnet try-convert tool in one operation. `dotnet try-convert` is not guaranteed to work for all your projects, and it may cause subtle changes in behavior that you may find that you depended on. It should be used as a *starting point* that automates the basic things that can be automated. It isn't a guaranteed solution to migrating a project.

NEXT

.NET Framework technologies unavailable on .NET Core

11/7/2019 • 2 minutes to read • Edit Online

Several technologies available to .NET Framework libraries aren't available for use with .NET Core, such as AppDomains, Remoting, Code Access Security (CAS), Security Transparency, and System.EnterpriseServices. If your libraries rely on one or more of these technologies, consider the alternative approaches outlined below. For more information on API compatibility, see the .NET Core breaking changes article.

Just because an API or technology isn't currently implemented doesn't imply it's intentionally unsupported. You should first search the GitHub repositories for .NET Core to see if a particular issue you encounter is by design, but if you cannot find such an indicator, please file an issue in the dotnet/corefx repository issues at GitHub to ask for specific APIs and technologies. Porting requests in the issues are marked with the `port-to-core` label.

AppDomains

Application domains (AppDomains) isolate apps from one another. AppDomains require runtime support and are generally quite expensive. Creating additional app domains is not supported. We don't plan on adding this capability in future. For code isolation, we recommend separate processes or using containers as an alternative. For the dynamic loading of assemblies, we recommend the new AssemblyLoadContext class.

To make code migration from .NET Framework easier, .NET Core exposes some of the AppDomain API surface. Some of the APIs function normally (for example, AppDomain.UnhandledException), some members do nothing (for example, SetCachePath), and some of them throw PlatformNotSupportedException (for example, CreateDomain). Check the types you use against the `System.AppDomain` reference source in the dotnet/corefx GitHub repository, making sure to select the branch that matches your implemented version.

Remoting

.NET Remoting was identified as a problematic architecture. It's used for cross-AppDomain communication, which is no longer supported. Also, Remoting requires runtime support, which is expensive to maintain. For these reasons, .NET Remoting isn't supported on .NET Core, and we don't plan on adding support for it in the future.

For communication across processes, consider inter-process communication (IPC) mechanisms as an alternative to Remoting, such as the System.IO.Pipes or the MemoryMappedFile class.

Across machines, use a network-based solution as an alternative. Preferably, use a low-overhead plain text protocol, such as HTTP. The Kestrel web server, the web server used by ASP.NET Core, is an option here. Also consider using System.Net.Sockets for network-based, cross-machine scenarios. For more options, see .NET Open Source Developer Projects: Messaging.

Code Access Security (CAS)

Sandboxing, which relies on the runtime or the framework to constrain which resources a managed application or library uses or runs, isn't supported on .NET Framework and therefore is also not supported on .NET Core. There are too many cases in the .NET Framework and the runtime where an elevation of privileges occurs to continue treating CAS as a security boundary. In addition, CAS makes the implementation more complicated and often has correctness-performance implications for applications that don't intend to use it.

Use security boundaries provided by the operating system, such as virtualization, containers, or user accounts for running processes with the minimum set of privileges.

Security Transparency

Similar to CAS, Security Transparency separates sandboxed code from security critical code in a declarative fashion but is no longer supported as a security boundary. This feature is heavily used by Silverlight.

Use security boundaries provided by the operating system, such as virtualization, containers, or user accounts for running processes with the least set of privileges.

System.EnterpriseServices

System.EnterpriseServices (COM+) is not supported by .NET Core.

NEXT

Analyze your dependencies to port code to .NET Core

10/23/2019 • 5 minutes to read • Edit Online

To port your code to .NET Core or .NET Standard, you must understand your dependencies. External dependencies are the NuGet packages or `.dll`s you reference in your project, but that you don't build yourself.

Migrate your NuGet packages to `PackageReference`

.NET Core uses PackageReference to specify package dependencies. If you're using packages.config to specify your packages in your project, you need to convert it to the `PackageReference` format because `packages.config` isn't supported in .NET Core.

To learn how to migrate, see the Migrate from packages.config to PackageReference article.

Upgrade your NuGet packages

After your migrating your project to the `PackageReference` format, you need to verify if your packages are compatible with .NET Core.

First, upgrade your packages to the latest version that you can. This can be done with the NuGet Package Manager UI in Visual Studio. It's likely that newer versions of your package dependencies are already compatible with .NET Core.

Analyze your package dependencies

If you haven't already verified that your converted and upgraded package dependencies work on .NET Core, there are a few ways that you can achieve that:

Analyze NuGet packages using nuget.org

You can see the Target Framework Monikers (TFMs) that each package supports on nuget.org under the **Dependencies** section of the package page.

Although using the site is an easier method to verify the compatibility, **Dependencies** information isn't available on the site for all packages.

Analyze NuGet packages using NuGet Package Explorer

A NuGet package is itself a set of folders that contain platform-specific assemblies. So you need to check if there's a folder that contains a compatible assembly inside the package.

The easiest way to inspect NuGet Package folders is to use the NuGet Package Explorer tool. After installing it, use the following steps to see the folder names:

1. Open the NuGet Package Explorer.
2. Click **Open package from online feed**.
3. Search for the name of the package.
4. Select the package name from the search results and click **open**.
5. Expand the *lib* folder on the right-hand side and look at folder names.

Look for a folder with names using one the following patterns: `netstandardX.Y` or `netcoreappX.Y`.

These values are the Target Framework Monikers (TFMs) that map to versions of the .NET Standard, .NET Core, and traditional Portable Class Library (PCL) profiles that are compatible with .NET Core.

IMPORTANT

When looking at the TFMs that a package supports, note that `netcoreapp*`, while compatible, is for .NET Core projects only and not for .NET Standard projects. A library that only targets `netcoreapp*` and not `netstandard*` can only be consumed by other .NET Core apps.

.NET Framework compatibility mode

After analyzing the NuGet packages, you might find that they only target the .NET Framework.

Starting with .NET Standard 2.0, the .NET Framework compatibility mode was introduced. This compatibility mode allows .NET Standard and .NET Core projects to reference .NET Framework libraries. Referencing .NET Framework libraries doesn't work for all projects, such as if the library uses Windows Presentation Foundation (WPF) APIs, but it does unblock many porting scenarios.

When you reference NuGet packages that target the .NET Framework in your project, such as Huitian.PowerCollections, you get a package fallback warning (NU1701) similar to the following example:

```
NU1701: Package ‘Huitian.PowerCollections 1.0.0’ was restored using ‘.NETFramework,Version=v4.6.1’ instead of
the project target framework ‘.NETStandard,Version=v2.0’. This package may not be fully compatible with your
project.
```

That warning is displayed when you add the package and every time you build to make sure you test that package with your project. If your project is working as expected, you can suppress that warning by editing the package properties in Visual Studio or by manually editing the project file in your favorite code editor.

To suppress the warning by editing the project file, find the `PackageReference` entry for the package you want to suppress the warning for and add the `NoWarn` attribute. The `NoWarn` attribute accepts a comma-separated list of all the warning IDs. The following example shows how to suppress the `NU1701` warning for the `Huitian.PowerCollections` package by editing your project file manually:

```
<ItemGroup>
  <PackageReference Include="Huitian.PowerCollections" Version="1.0.0" NoWarn="NU1701" />
</ItemGroup>
```

For more information on how to suppress compiler warnings in Visual Studio, see Suppressing warnings for NuGet packages.

What to do when your NuGet package dependency doesn't run on .NET Core

There are a few things you can do if a NuGet package you depend on doesn't run on .NET Core:

1. If the project is open source and hosted somewhere like GitHub, you can engage the developers directly.
2. You can contact the author directly on nuget.org. Search for the package and click **Contact Owners** on the left-hand side of the package's page.
3. You can search for another package that runs on .NET Core that accomplishes the same task as the package you were using.
4. You can attempt to write the code the package was doing yourself.
5. You could eliminate the dependency on the package by changing the functionality of your app, at least until a compatible version of the package becomes available.

Remember that open-source project maintainers and NuGet package publishers are often volunteers. They contribute because they care about a given domain, do it for free, and often have a different daytime job. So be mindful of that when contacting them to ask for .NET Core support.

If you can't resolve your issue with any of the above, you may have to port to .NET Core at a later date.

The .NET Team would like to know which libraries are the most important to support with .NET Core. You can send an email to dotnet@microsoft.com about the libraries you'd like to use.

Analyze dependencies that aren't NuGet packages

You may have a dependency that isn't a NuGet package, such as a DLL in the file system. The only way to determine the portability of that dependency is to run the .NET Portability Analyzer tool. The tool can analyze assemblies that target the .NET Framework and identify APIs that aren't portable to other .NET platforms such as .NET Core. You can run the tool as a console application or as a Visual Studio extension.

NEXT

Port .NET Framework libraries to .NET Core

8/28/2019 • 6 minutes to read • Edit Online

Learn how to port .NET Framework library code to .NET Core, to run cross-platform and expand the reach of the apps that use it.

Prerequisites

This article assumes that you:

- Are using Visual Studio 2017 or later.
  - .NET Core isn't supported on earlier versions of Visual Studio
- Understand the recommended porting process.
- Have resolved any issues with third-party dependencies.

You should also become familiar with the content of the following topics:

.NET Standard
This topic describes the formal specification of .NET APIs that are intended to be available on all .NET implementations.

Packages, Metapackages and Frameworks
This article discusses how .NET Core defines and uses packages and how packages support code running on multiple .NET implementations.

Developing Libraries with Cross Platform Tools
This topic explains how to write libraries for .NET using cross-platform CLI tools.

Additions to the *csproj* format for .NET Core
This article outlines the changes that were added to the project file as part of the move to *csproj* and MSBuild.

Porting to .NET Core - Analyzing your Third-Party Party Dependencies
This topic discusses the portability of third-party dependencies and what to do when a NuGet package dependency doesn't run on .NET Core.

Retargeting your .NET Framework code to .NET Framework 4.7.2

If your code isn't targeting .NET Framework 4.7.2, we recommended that you retarget to .NET Framework 4.7.2. This ensures the availability of the latest API alternatives for cases where the .NET Standard doesn't support existing APIs.

For each of your projects in Visual Studio you wish to port, do the following:

1. Right-click on the project and select **Properties**.
2. In the **Target Framework** dropdown, select **.NET Framework 4.7.2**.
3. Recompile your projects.

Because your projects now target .NET Framework 4.7.2, use that version of the .NET Framework as your base for porting code.

Determining the portability of your code

The next step is to run the API Portability Analyzer (ApiPort) to generate a portability report for analysis.

Make sure you understand the API Portability Analyzer (ApiPort) and how to generate portability reports for targeting .NET Core. How you do this likely varies based on your needs and personal tastes. What follows are a few different approaches. You may find yourself mixing steps of these approaches depending on how your code is structured.

Dealing primarily with the compiler

This approach may be the best for small projects or projects which don't use many .NET Framework APIs. The approach is simple:

1. Optionally, run ApiPort on your project. If you run ApiPort, gain knowledge from the report on issues you'll need to address.
2. Copy all of your code over into a new .NET Core project.
3. While referring to the portability report (if generated), solve compiler errors until the project fully compiles.

Although this approach is unstructured, the code-focused approach often leads to resolving issues quickly and might be the best approach for smaller projects or libraries. A project that contains only data models might be an ideal candidate for this approach.

Staying on the .NET Framework until portability issues are resolved

This approach might be the best if you prefer to have code that compiles during the entire process. The approach is as follows:

1. Run ApiPort on a project.
2. Address issues by using different APIs that are portable.
3. Take note of any areas where you're prevented from using a direct alternative.
4. Repeat the prior steps for all projects you're porting until you're confident each is ready to be copied over into a new .NET Core project.
5. Copy the code into a new .NET Core project.
6. Work out any issues where you noted that a direct alternative doesn't exist.

This careful approach is more structured than simply working out compiler errors, but it's still relatively code-focused and has the benefit of always having code that compiles. The way you resolve certain issues that couldn't be addressed by just using another API varies greatly. You may find that you need to develop a more comprehensive plan for certain projects, which is covered as the next approach.

Developing a comprehensive plan of attack

This approach might be best for larger and more complex projects, where restructuring code or completely rewriting certain areas of code might be necessary to support .NET Core. The approach is as follows:

1. Run ApiPort on a project.
2. Understand where each non-portable type is used and how that affects overall portability.
   - Understand the nature of those types. Are they small in number but used frequently? Are they large in number but used infrequently? Is their use concentrated, or is it spread throughout your code?
   - Is it easy to isolate code that isn't portable so that you can deal with it more effectively?
   - Do you need to refactor your code?
   - For those types which aren't portable, are there alternative APIs that accomplish the same task? For example if you're using the WebClient class, you might be able to use the HttpClient class instead.
   - Are there different portable APIs available to accomplish a task, even if it's not a drop-in replacement? For example if you're using XmlSchema to parse XML but don't require XML schema discovery, you could use System.Xml.Linq APIs and implement parsing yourself as opposed to relying on an API.
3. If you have assemblies that are difficult to port, is it worth leaving them on .NET Framework for now? Here are some things to consider:
   - You may have some functionality in your library that's incompatible with .NET Core because it relies too

heavily on .NET Framework or Windows-specific functionality. Is it worth leaving that functionality behind for now and releasing a .NET Core version of your library with less features on a temporary basis until resources are available to port the features?
   - Would a refactor help?
4. Is it reasonable to write your own implementation of an unavailable .NET Framework API? You could consider copying, modifying, and using code from the .NET Framework Reference Source. The reference source code is licensed under the MIT License, so you have significant freedom to use the source as a basis for your own code. Just be sure to properly attribute Microsoft in your code.
5. Repeat this process as needed for different projects.

The analysis phase could take some time depending on the size of your codebase. Spending time in this phase to thoroughly understand the scope of changes needed and to develop a plan usually saves you time in the long run, particularly if you have a complex codebase.

Your plan could involve making significant changes to your codebase while still targeting .NET Framework 4.7.2, making this a more structured version of the previous approach. How you go about executing your plan is dependent on your codebase.

Mixing approaches

It's likely that you'll mix the above approaches on a per-project basis. You should do what makes the most sense to you and for your codebase.

Porting your tests

The best way to make sure everything works when you've ported your code is to test your code as you port it to .NET Core. To do this, you'll need to use a testing framework that builds and runs tests for .NET Core. Currently, you have three options:

- xUnit
  - Getting Started
  - Tool to convert an MSTest project to xUnit
- NUnit
  - Getting Started
  - Blog post about migrating from MSTest to NUnit
- MSTest

Recommended approach to porting

Ultimately, the porting effort depends heavily on how your .NET Framework code is structured. A good way to port your code is to begin with the *base* of your library, which are the foundational components of your code. This might be data models or some other foundational classes and methods that everything else uses directly or indirectly.

1. Port the test project that tests the layer of your library that you're currently porting.
2. Copy over the base of your library into a new .NET Core project and select the version of the .NET Standard you wish to support.
3. Make any changes needed to get the code to compile. Much of this may require adding NuGet package dependencies to your *csproj* file.
4. Run the tests and make any needed adjustments.
5. Pick the next layer of code to port over and repeat the prior steps.

If you start with the base of your library and move outward from the base and test each layer as needed, porting is a systematic process where problems are isolated to one layer of code at a time.

NEXT

Organize your project to support both .NET Framework and .NET Core

10/17/2019 • 2 minutes to read • Edit Online

Learn how to create a solution that compiles for both .NET Framework and .NET Core side-by-side. See several options to organize projects to help you achieve this goal. Here are some typical scenarios to consider when you're deciding how to setup your project layout with .NET Core. The list may not cover everything you want; prioritize based on your project's needs.

- **Combine existing projects and .NET Core projects into single projects**

  What this is good for:

  - Simplifying your build process by compiling a single project rather than compiling multiple projects, each targeting a different .NET Framework version or platform.
  - Simplifying source file management for multi-targeted projects because you must manage a single project file. When adding/removing source files, the alternatives require you to manually sync these with your other projects.
  - Easily generating a NuGet package for consumption.
  - Allows you to write code for a specific .NET Framework version in your libraries through the use of compiler directives.

  Unsupported scenarios:

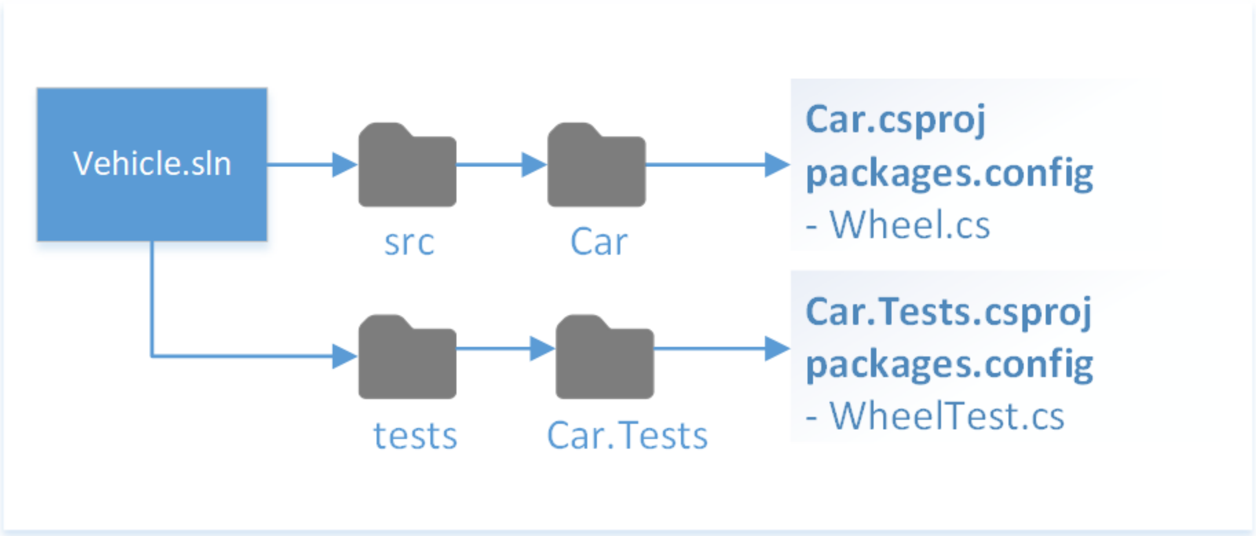
  - Requires developers to use Visual Studio 2017 to open existing projects. To support older versions of Visual Studio, keeping your project files in different folders is a better option.
- **Keep existing projects and new .NET Core projects separate**

  What this is good for:

  - Continuing to support development on existing projects without having to upgrade for developers/contributors who may not have Visual Studio 2017.
  - Decreasing the possibility of creating new bugs in existing projects because no code churn is required in those projects.

Example

Consider the repository below:

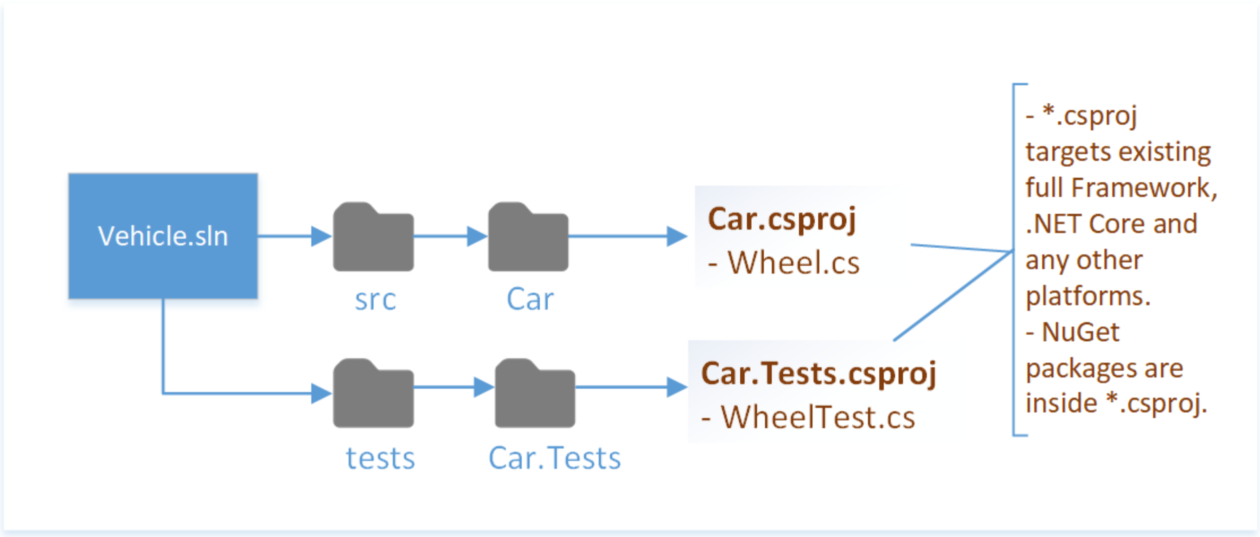


Source Code

The following describes several ways to add support for .NET Core for this repository depending on the constraints and complexity of the existing projects.

Replace existing projects with a multi-targeted .NET Core project

Reorganize the repository so that any existing **.csproj* files are removed and a single **.csproj* file is created that targets multiple frameworks. This is a great option because a single project is able to compile for different frameworks. It also has the power to handle different compilation options and dependencies per targeted framework.


Vehicle.sln
src
Car
Car.csproj
- Wheel.cs
tests
Car.Tests
Car.Tests.csproj
- WheelTest.cs
- *.csproj targets existing full Framework, .NET Core and any other platforms.
- NuGet packages are inside *.csproj.

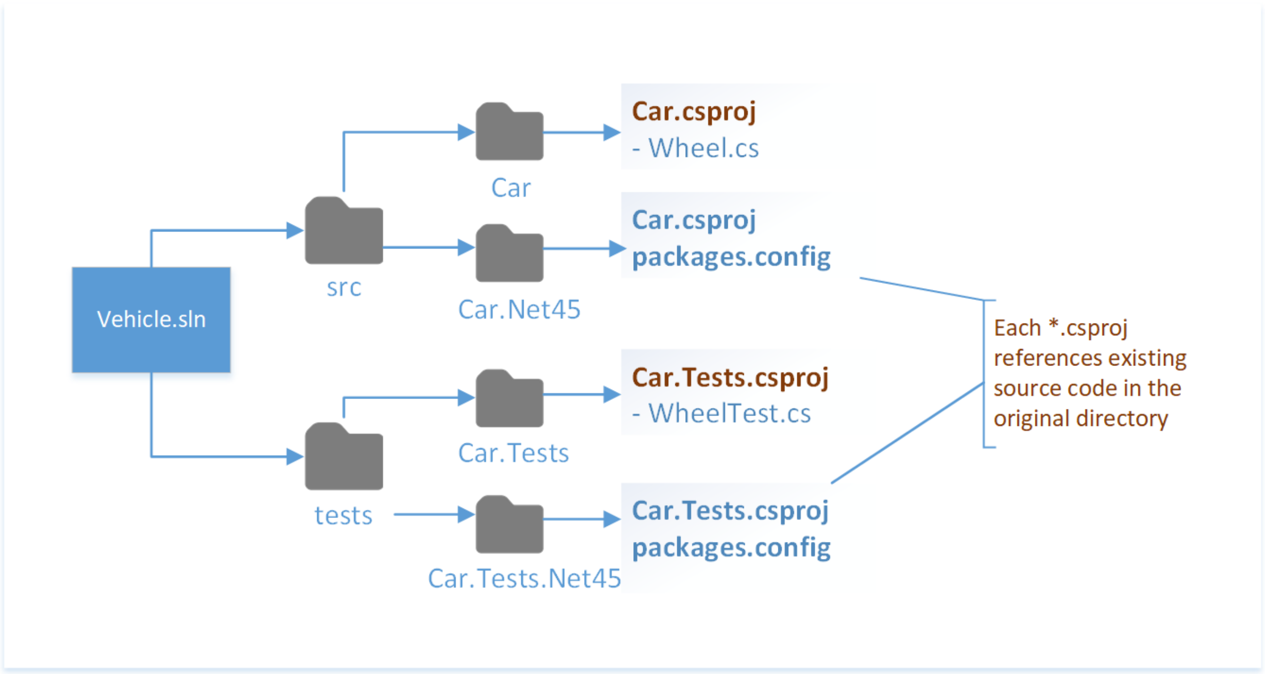
Source Code

Changes to note are:

- Replacement of *packages.config* and **.csproj* with a new .NET Core **.csproj*. NuGet packages are specified with `<PackageReference> ItemGroup`.

Keep existing projects and create a .NET Core project

If there are existing projects that target older frameworks, you may want to leave these projects untouched and use a .NET Core project to target future frameworks.



Source Code

Changes to note are:

- The .NET Core and existing projects are kept in separate folders.
  - Keeping projects in separate folders avoids forcing you to have Visual Studio 2017 or later versions. You can create a separate solution that only opens the old projects.

See also

- .NET Core porting documentation

Tools to help with porting to .NET Core

10/17/2019 • 2 minutes to read • Edit Online

You may find the tools listed in this article helpful when porting:

- .NET Portability Analyzer - A toolchain that can generate a report of how portable your code is between .NET Framework and .NET Core: As a command-line tool As a Visual Studio Extension
- .NET API analyzer - A Roslyn analyzer that discovers potential compatibility risks for C# APIs on different platforms and detects calls to deprecated APIs.

Additionally, you can attempt to port smaller solutions or individual projects to the .NET Core project file format with the CsprojToVs2017 tool.

WARNING

CsprojToVs2017 is a third-party tool. There is no guarantee that it will work for all of your projects, and it may cause subtle changes in behavior that you depend on. CsprojToVs2017 should be used as a *starting point* that automates the basic things that can be automated. It is not a guaranteed solution to migrating project file formats.

Use the Windows Compatibility Pack to port code to .NET Core

10/17/2019 • 2 minutes to read • Edit Online

Some of the most common issues found when porting existing code to .NET Core are dependencies on APIs and technologies that are only found in the .NET Framework. The *Windows Compatibility Pack* provides many of these technologies, so it's much easier to build .NET Core applications and .NET Standard libraries.

This package is a logical extension of .NET Standard 2.0 that significantly increases API set and existing code compiles with almost no modifications. But in order to keep the promise of .NET Standard ("it is the set of APIs that all .NET implementations provide"), this didn't include technologies that can't work across all platforms, such as registry, Windows Management Instrumentation (WMI), or reflection emit APIs.

The *Windows Compatibility Pack* sits on top of .NET Standard and provides access to technologies that are Windows only. It's especially useful for customers that want to move to .NET Core but plan to stay on Windows as a first step. In that scenario, not being able to use Windows-only technologies is only a migration hurdle with zero architectural benefits.

Package contents

The *Windows Compatibility Pack* is provided via the NuGet Package Microsoft.Windows.Compatibility and can be referenced from projects targeting .NET Core or .NET Standard.

It provides about 20,000 APIs, including Windows-only as well as cross-platform APIs from the following technology areas:

- Code Pages
- CodeDom
- Configuration
- Directory Services
- Drawing
- ODBC
- Permissions
- Ports
- Windows Access Control Lists (ACL)
- Windows Communication Foundation (WCF)
- Windows Cryptography
- Windows EventLog
- Windows Management Instrumentation (WMI)
- Windows Performance Counters
- Windows Registry
- Windows Runtime Caching
- Windows Services

For more information, see the specification of the compatibility pack.

Get started

1. Before porting, make sure to take a look at the Porting Process.
2. When porting existing code to .NET Core or .NET Standard, install the NuGet package Microsoft.Windows.Compatibility.
3. If you want to stay on Windows, you're all set.
4. If you want to run the .NET Core application or .NET Standard library on Linux or macOS, use the API Analyzer to find usage of APIs that won't work cross-platform.
5. Either remove the usages of those APIs, replace them with cross-platform alternatives, or guard them using a platform check, like:

```
private static string GetLoggingPath()
{
    // Verify the code is running on Windows.
    if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
    {
        using (var key = Registry.CurrentUser.OpenSubKey(@"Software\Fabrikam\AssetManagement"))
        {
            if (key?.GetValue("LoggingDirectoryPath") is string configuredPath)
                return configuredPath;
        }
    }

    // This is either not running on Windows or no logging path was configured,
    // so just use the path for non-roaming user-specific data files.
    var appDataPath = Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData);
    return Path.Combine(appDataPath, "Fabrikam", "AssetManagement", "Logging");
}
```

For a demo, check out the Channel 9 video of the Windows Compatibility Pack.

How to port a Windows Forms desktop app to .NET Core

11/12/2019 • 8 minutes to read • Edit Online

This article describes how to port your Windows Forms-based desktop app from .NET Framework to .NET Core 3.0. The .NET Core 3.0 SDK includes support for Windows Forms applications. Windows Forms is still a Windows-only framework and only runs on Windows. This example uses the .NET Core SDK CLI to create and manage your project.

In this article, various names are used to identify types of files used for migration. When migrating your project, your files will be named differently, so mentally match them to the ones listed below:

FILE	DESCRIPTION
MyApps.sln	The name of the solution file.
MyForms.csproj	The name of the .NET Framework Windows Forms project to port.
MyFormsCore.csproj	The name of the new .NET Core project you create.
MyAppCore.exe	The .NET Core Windows Forms app executable.

Prerequisites

- Visual Studio 2019 for any designer work you want to do.

  Install the following Visual Studio workloads:

  - .NET desktop development
  - .NET cross-platform development
- A working Windows Forms project in a solution that builds and runs without issue.
- Your project must be coded in C#.
- Install the latest .NET Core 3.0 preview.

NOTE
Visual Studio 2017 doesn't support .NET Core 3.0 projects. **Visual Studio 2019** supports .NET Core 3.0 projects but doesn't yet support the visual designer for .NET Core 3.0 Windows Forms projects. To use the visual designer, you must have a .NET Windows Forms project in your solution that shares the forms files with the .NET Core project.

Consider

When porting a .NET Framework Windows Forms application, there are a few things you must consider.

1. Check that your application is a good candidate for migration.

   Use the .NET Portability Analyzer to determine if your project will migrate to .NET Core 3.0. If your project has issues with .NET Core 3.0, the analyzer helps you identify those problems.

2. You're using a different version of Windows Forms.

   When .NET Core 3.0 Preview 1 was released, Windows Forms went open source on GitHub. The code for .NET Core Windows Forms is a fork of the .NET Framework Windows Forms codebase. It's possible some differences exist and your app won't port.

3. The Windows Compatibility Pack may help you migrate.

   Some APIs that are available in .NET Framework aren't available in .NET Core 3.0. The Windows Compatibility Pack adds many of these APIs and may help your Windows Forms app become compatible with .NET Core.

4. Update the NuGet packages used by your project.

   It's always a good practice to use the latest versions of NuGet packages before any migration. If your application is referencing any NuGet packages, update them to the latest version. Ensure your application builds successfully. After upgrading, if there are any package errors, downgrade the package to the latest version that doesn't break your code.

5. Visual Studio 2019 doesn't yet support the Forms Designer for .NET Core 3.0

   Currently, you need to keep your existing .NET Framework Windows Forms project file if you want to use the Forms Designer from Visual Studio.

Create a new SDK project

The new .NET Core 3.0 project you create must be in a different directory from your .NET Framework project. If they're both in the same directory, you may run into conflicts with the files that are generated in the **obj** directory. In this example, we'll create a directory named **MyFormsAppCore** in the **SolutionFolder** directory:

```
SolutionFolder
├───MyApps.sln
├───MyFormsApp
│   └───MyForms.csproj
└───MyFormsAppCore      <--- New folder for core project
```

Next, you need to create the **MyFormsCore.csproj** project in the **MyFormsAppCore** directory. You can create this file manually by using the text editor of choice. Paste in the following XML:

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

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <UseWindowsForms>true</UseWindowsForms>
  </PropertyGroup>

</Project>
```

If you don't want to create the project file manually, you can use Visual Studio or the .NET Core SDK to generate the project. However, you must delete all other files generated by the project template except for the project file. To use the SDK, run the following command from the **SolutionFolder** directory:

```
dotnet new winforms -o MyFormsAppCore -n MyFormsCore
```

After you create the **MyFormsCore.csproj**, your directory structure should look like the following:

```
SolutionFolder
├───MyApps.sln
├───MyFormsApp
│   └───MyForms.csproj
└───MyFormsAppCore
    └───MyFormsCore.csproj
```

You'll want to add the **MyFormsCore.csproj** project to **MyApps.sln** with either Visual Studio or the .NET Core CLI from the **SolutionFolder** directory:

```
dotnet sln add .\MyFormsAppCore\MyFormsCore.csproj
```

Fix assembly info generation

Windows Forms projects that were created with .NET Framework include an `AssemblyInfo.cs` file, which contains assembly attributes such as the version of the assembly to be generated. SDK-style projects automatically generate this information for you based on the SDK project file. Having both types of "assembly info" creates a conflict. Resolve this problem by disabling automatic generation, which forces the project to use your existing `AssemblyInfo.cs` file.

There are three settings to add to the main `<PropertyGroup>` node.

- **GenerateAssemblyInfo**
  When you set this property to `false`, it won't generate the assembly attributes. This avoids the conflict with the existing `AssemblyInfo.cs` file from the .NET Framework project.

- **AssemblyName**
  The value of this property is the output binary created when you compile. The name doesn't need an extension added to it. For example, using `MyCoreApp` produces `MyCoreApp.exe`.

- **RootNamespace**
  The default namespace used by your project. This should match the default namespace of the .NET Framework project.

Add these three elements to the `<PropertyGroup>` node in the `MyFormsCore.csproj` file:

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

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <UseWindowsForms>true</UseWindowsForms>

    <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
    <AssemblyName>MyCoreApp</AssemblyName>
    <RootNamespace>WindowsFormsApp1</RootNamespace>
  </PropertyGroup>

</Project>
```

Add source code

Right now, the **MyFormsCore.csproj** project doesn't compile any code. By default, .NET Core projects automatically include all source code in the current directory and any child directories. You must configure the project to include code from the .NET Framework project using a relative path. If your .NET Framework project used **.resx** files for icons and resources for your forms, you'll need to include those too.

Add the following `<ItemGroup>` node to your project. Each statement includes a file glob pattern that includes child directories.

```
<ItemGroup>
  <Compile Include="..\MyFormsApp\**\*.cs" />
  <EmbeddedResource Include="..\MyFormsApp\**\*.resx" />
</ItemGroup>
```

Alternatively, you can create a `<Compile>` or `<EmbeddedResource>` entry for each file in your .NET Framework project.

Add NuGet packages

Add each NuGet package referenced by the .NET Framework project to the .NET Core project.

Most likely your .NET Framework Windows Forms app has a **packages.config** file that contains a list of all of the NuGet packages that are referenced by your project. You can look at this list to determine which NuGet packages to add to the .NET Core project. For example, if the .NET Framework project referenced the `MetroFramework`, `MetroFramework.Design`, and `MetroFramework.Fonts` NuGet packages, add each to the project with either Visual Studio or the .NET Core CLI from the **SolutionFolder** directory:

```
dotnet add .\MyFormsAppCore\MyFormsCore.csproj package MetroFramework
dotnet add .\MyFormsAppCore\MyFormsCore.csproj package MetroFramework.Design
dotnet add .\MyFormsAppCore\MyFormsCore.csproj package MetroFramework.Fonts
```

The previous commands would add the following NuGet references to the **MyFormsCore.csproj** project:

```
<ItemGroup>
  <PackageReference Include="MetroFramework" Version="1.2.0.3" />
  <PackageReference Include="MetroFramework.Design" Version="1.2.0.3" />
  <PackageReference Include="MetroFramework.Fonts" Version="1.2.0.3" />
</ItemGroup>
```

Port control libraries

If you have a Windows Forms Controls library project to port, the directions are the same as porting a .NET Framework Windows Forms app project, except for a few settings. And instead of compiling to an executable, you compile to a library. The difference between the executable project and the library project, besides paths for the file globs that include your source code, is minimal.

Using the previous step's example, lets expand what projects and files we're working with.

FILE	DESCRIPTION
MyApps.sln	The name of the solution file.
MyControls.csproj	The name of the .NET Framework Windows Forms Controls project to port.
MyControlsCore.csproj	The name of the new .NET Core library project you create.
MyCoreControls.dll	The .NET Core Windows Forms Controls library.

```
SolutionFolder
├──MyApps.sln
├──MyFormsApp
│   └──MyForms.csproj
├──MyFormsAppCore
│   └──MyFormsCore.csproj
│
├──MyFormsControls
│   └──MyControls.csproj
└──MyFormsControlsCore
    └──MyControlsCore.csproj   <--- New project for core controls
```

Consider the differences between the `MyControlsCore.csproj` project and the previously created `MyFormsCore.csproj` project.

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

   <PropertyGroup>
-    <OutputType>WinExe</OutputType>
     <TargetFramework>netcoreapp3.0</TargetFramework>
     <UseWindowsForms>true</UseWindowsForms>

     <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
-    <AssemblyName>MyCoreApp</AssemblyName>
-    <RootNamespace>WindowsFormsApp1</RootNamespace>
+    <AssemblyName>MyControlsCore</AssemblyName>
+    <RootNamespace>WindowsFormsControlLibrary1</RootNamespace>
   </PropertyGroup>

   <ItemGroup>
-    <Compile Include="..\MyFormsApp\**\*.cs" />
-    <EmbeddedResource Include="..\MyFormsApp\**\*.resx" />
+    <Compile Include="..\MyFormsControls\**\*.cs" />
+    <EmbeddedResource Include="..\MyFormsControls\**\*.resx" />
   </ItemGroup>

 </Project>
```

Here is an example of what the .NET Core Windows Forms Controls library project file would look like:

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

  <PropertyGroup>

    <TargetFramework>netcoreapp3.0</TargetFramework>
    <UseWindowsForms>true</UseWindowsForms>

    <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
    <AssemblyName>MyCoreControls</AssemblyName>
    <RootNamespace>WindowsFormsControlLibrary1</RootNamespace>
  </PropertyGroup>

  <ItemGroup>
    <Compile Include="..\MyFormsControls\**\*.cs" />
    <EmbeddedResource Include="..\MyFormsControls\**\*.resx" />
  </ItemGroup>

</Project>
```

As you can see, the `<OutputType>` node was removed, which defaults the compiler to produce a library instead of an executable. The `<AssemblyName>` and `<RootNamespace>` were changed. Specifically the `<RootNamespace>` should match the namespace of the Windows Forms Controls library you are porting. And finally, the `<Compile>` and

`<EmbeddedResource>` nodes were adjusted to point to the folder of the Windows Forms Controls library you are porting.

Next, in the main .NET Core **MyFormsCore.csproj** project add reference to the new .NET Core Windows Forms Control library. Add a reference with either Visual Studio or the .NET Core CLI from the **SolutionFolder** directory:

```
dotnet add .\MyFormsAppCore\MyFormsCore.csproj reference .\MyFormsControlsCore\MyControlsCore.csproj
```

The previous command adds the following to the **MyFormsCore.csproj** project:

```
  <ItemGroup>
    <ProjectReference Include="..\MyFormsControlsCore\MyControlsCore.csproj" />
  </ItemGroup>
```

Problems compiling

If you have problems compiling your projects, you may be using some Windows-only APIs that are available in .NET Framework but not available in .NET Core. You can try adding the Windows Compatibility Pack NuGet package to your project. This package only runs on Windows and adds about 20,000 Windows APIs to .NET Core and .NET Standard projects.

```
dotnet add .\MyFormsAppCore\MyFormsCore.csproj package Microsoft.Windows.Compatibility
```

The previous command adds the following to the **MyFormsCore.csproj** project:

```
  <ItemGroup>
    <PackageReference Include="Microsoft.Windows.Compatibility" Version="2.0.1" />
  </ItemGroup>
```

Windows Forms Designer

As detailed in this article, Visual Studio 2019 only supports the Forms Designer in .NET Framework projects. By creating a side-by-side .NET Core project, you can test your project with .NET Core while you use the .NET Framework project to design forms. Your solution file includes both the .NET Framework and .NET Core projects. Add and design your forms and controls in the .NET Framework project, and based on the file glob patterns we added to the .NET Core projects, any new or changed files will automatically be included in the .NET Core projects.

Once Visual Studio 2019 supports the Windows Forms Designer, you can copy/paste the content of your .NET Core project file into the .NET Framework project file. Then delete the file glob patterns added with the `<Source>` and `<EmbeddedResource>` items. Fix the paths to any project reference used by your app. This effectively upgrades the .NET Framework project to a .NET Core project.

Next steps

- Read more about the Windows Compatibility Pack.
- Watch a video on porting your .NET Framework Windows Forms project to .NET Core.

2 minutes to read

Dependency loading in .NET Core

11/1/2019 • 2 minutes to read • Edit Online

Every .NET Core application has dependencies. Even the simple `hello world` app has dependencies on portions of the .NET Core class libraries.

Understanding .NET Core default assembly loading logic can help understanding and debugging typical deployment issues.

In some applications, dependencies are dynamically determined at runtime. In these situations, it's critical to understand how managed assemblies and unmanaged dependencies are loaded.

Understanding AssemblyLoadContext

The AssemblyLoadContext API is central to the .NET Core loading design. The Understanding AssemblyLoadContext article provides a conceptual overview to the design.

Loading details

The loading algorithm details are covered briefly in several articles:

- Managed assembly loading algorithm
- Satellite assembly loading algorithm
- Unmanaged (native) library loading algorithm
- Default probing

Create a .NET Core application with plugins

The tutorial Create a .NET Core application with plugins describes how to create a custom AssemblyLoadContext. It uses an AssemblyDependencyResolver to resolve the dependencies of the plugin. The tutorial correctly isolates the plugin's dependencies from the hosting application.

How to use and debug assembly unloadability in .NET Core

The How to use and debug assembly unloadability in .NET Core article is a step-by-step tutorial. It shows how to load a .NET Core application, execute, and then unload it. The article also provides debugging tips.

Understanding System.Runtime.Loader.AssemblyLoadContext

10/12/2019 • 4 minutes to read • Edit Online

The AssemblyLoadContext class is unique to .NET Core. This article attempts to supplement the AssemblyLoadContext API documentation with conceptual information.

This article is relevant to developers implementing dynamic loading, especially dynamic loading framework developers.

What is the AssemblyLoadContext?

Every .NET Core application implicitly uses the AssemblyLoadContext. It's the runtime's provider for locating and loading dependencies. Whenever a dependency is loaded, an AssemblyLoadContext instance is invoked to locate it.

- It provides a service of locating, loading, and caching managed assemblies and other dependencies.
- To support dynamic code loading and unloading, it creates an isolated context for loading code and its dependencies in their own AssemblyLoadContext instance.

When do you need multiple AssemblyLoadContext instances?

A single AssemblyLoadContext instance is limited to loading exactly one version of an Assembly per simple assembly name, AssemblyName.Name.

This restriction can become a problem when loading code modules dynamically. Each module is independently compiled and may depend on different versions of an Assembly. This problem commonly occurs when different modules depend on different versions of a commonly used library.

To support dynamically loading code, the AssemblyLoadContext API provides for loading conflicting versions of an Assembly in the same application. Each AssemblyLoadContext instance provides a unique dictionary mapping each AssemblyName.Name to a specific Assembly instance.

It also provides a convenient mechanism for grouping dependencies related to a code module for later unload.

What is special about the AssemblyLoadContext.Default instance?

The AssemblyLoadContext.Default instance is automatically populated by the runtime at startup. It uses default probing to locate and find all static dependencies.

It solves the most common dependency loading scenarios.

How does AssemblyLoadContext support dynamic dependencies?

AssemblyLoadContext has various events and virtual functions that can be overridden.

The AssemblyLoadContext.Default instance only supports overriding the events.

The articles Managed assembly loading algorithm, Satellite assembly loading algorithm, and Unmanaged (native) library loading algorithm refer to all the available events and virtual functions. The articles show each event and function's relative position in the loading algorithms. This article doesn't reproduce that information.

This section covers the general principles for the relevant events and functions.

- **Be repeatable**. A query for a specific dependency must always result in the same response. The same loaded dependency instance must be returned. This requirement is fundamental for cache consistency. For managed assemblies in particular, we're creating a Assembly cache. The cache key is a simple assembly name, AssemblyName.Name.
- **Typically don't throw**. It's expected that these functions return `null` rather than throw when unable to find the requested dependency. Throwing will prematurely end the search and be propagate an exception to the caller. Throwing should be restricted to unexpected errors like a corrupted assembly or an out of memory condition.
- **Avoid recursion**. Be aware that these functions and handlers implement the loading rules for locating dependencies. Your implementation shouldn't call APIs that trigger recursion. Your code should typically call **AssemblyLoadContext** load functions that require a specific path or memory reference argument.
- **Load into the correct AssemblyLoadContext**. The choice of where to load dependencies is application-specific. The choice is implemented by these events and functions. When your code calls **AssemblyLoadContext** load-by-path functions call them on the instance where you want the code loaded. Sometime returning `null` and letting the AssemblyLoadContext.Default handle the load may be the simplest option.
- **Be aware of thread races**. Loading can be triggered by multiple threads. The AssemblyLoadContext handles thread races by atomically adding assemblies to its cache. The race loser's instance is discarded. In your implementation logic, don't add extra logic that doesn't handle multiple threads properly.

How are dynamic dependencies isolated?

Each AssemblyLoadContext instance represents a unique scope for Assembly instances and Type definitions.

There's no binary isolation between these dependencies. They're only isolated by not finding each other by name.

In each AssemblyLoadContext:

- AssemblyName.Name may refer to a different Assembly instance.
- Type.GetType may return a different type instance for the same type `name`.

How are dependencies shared?

Dependencies can easily be shared between AssemblyLoadContext instances. The general model is for one AssemblyLoadContext to load a dependency. The other shares the dependency by using a reference to the loaded assembly.

This sharing is required of the runtime assemblies. These assemblies can only be loaded into the AssemblyLoadContext.Default. The same is required for frameworks like `ASP.NET`, `WPF`, or `WinForms`.

It's recommended that shared dependencies should be loaded into AssemblyLoadContext.Default. This sharing is the common design pattern.

Sharing is implemented in the coding of the custom AssemblyLoadContext instance. AssemblyLoadContext has various events and virtual functions that can be overridden. When any of these functions return a reference to an Assembly instance that was loaded in another AssemblyLoadContext instance, the Assembly instance is shared. The standard load algorithm defers to AssemblyLoadContext.Default for loading to simplify the common sharing pattern. See Managed assembly loading algorithm.

Complications

Type conversion issues

When two AssemblyLoadContext instances contain type definitions with the same `name`, they're not the same type. They're the same type if and only if they come from the same Assembly instance.

To complicate matters, exception messages about these mismatched types can be confusing. The types are referred to in the exception messages by their simple type names. The common exception message in this case would be of the form:

> Object of type 'IsolatedType' cannot be converted to type 'IsolatedType'.

Debugging type conversion issues

Given a pair of mismatched types it's important to also know:

- Each type's Type.Assembly
- Each type's AssemblyLoadContext, which can be obtained via the AssemblyLoadContext.GetLoadContext(Assembly) function.

Given two objects `a` and `b`, evaluating the following in the debugger will be helpful:

```
// In debugger look at each assembly's instance, Location, and FullName
a.GetType().Assembly
b.GetType().Assembly
// In debugger look at each AssemblyLoadContext's instance and name
System.Runtime.AssemblyLoadContext.GetLoadContext(a.GetType().Assembly)
System.Runtime.AssemblyLoadContext.GetLoadContext(b.GetType().Assembly)
```

Resolving type conversion issues

There are two design patterns for solving these type conversion issues.

1. Use common shared types. This shared type can either be a primitive runtime type, or it can involve creating a new shared type in a shared assembly. Often the shared type is an interface defined in an application assembly. See also: How are dependencies shared?.
2. Use marshaling techniques to convert from one type to another.

Default probing

10/8/2019 • 2 minutes to read • Edit Online

The AssemblyLoadContext.Default instance is responsible for locating an assembly's dependencies. This article describes the AssemblyLoadContext.Default instance's probing logic.

Host configured probing properties

When the runtime is started, the runtime host provides a set of named probing properties that configure AssemblyLoadContext.Default probe paths.

Each probing property is optional. If present, each property is a string value that contains a delimited list of absolute paths. The delimiter is ';' on Windows and ':' on all other platforms.

PROPERTY NAME	DESCRIPTION
`TRUSTED_PLATFORM_ASSEMBLIES`	List of platform and application assembly file paths.
`PLATFORM_RESOURCE_ROOTS`	List of directory paths to search for satellite resource assemblies.
`NATIVE_DLL_SEARCH_DIRECTORIES`	List of directory paths to search for unmanaged (native) libraries.
`APP_PATHS`	List of directory paths to search for managed assemblies.
`APP_NI_PATHS`	List of directory paths to search for native images of managed assemblies.

How are the properties populated?

There are two main scenarios for populating the properties depending on whether the *<myapp>.deps.json* file exists.

- When the **.deps.json* file is present, it's parsed to populate the probing properties.
- When the **.deps.json* file isn't present, the application's directory is assumed to contain all the dependencies. The directory's contents are used to populate the probing properties.

Additionally, the **.deps.json* files for any referenced frameworks are similarly parsed.

Finally the environment variable `ADDITIONAL_DEPS` can be used to add additional dependencies.

How do I see the probing properties from managed code?

Each property is available by calling the AppContext.GetData(String) function with the property name from the table above.

How do I debug the probing properties' construction?

The .NET Core runtime host will output useful trace messages when certain environment variables are enabled:

ENVIRONMENT VARIABLE	DESCRIPTION
`COREHOST_TRACE=1`	Enables tracing.

ENVIRONMENT VARIABLE	DESCRIPTION
`COREHOST_TRACEFILE=<path>`	Traces to a file path instead of the default `stderr`.
`COREHOST_TRACE_VERBOSITY`	Sets the verbosity from 1 (lowest) to 4 (highest).

Managed assembly default probing

When probing to locate a managed assembly, the AssemblyLoadContext.Default looks in order at:

- Files matching the AssemblyName.Name in `TRUSTED_PLATFORM_ASSEMBLIES` (after removing file extensions).
- Native image assembly files in `APP_NI_PATHS` with common file extensions.
- Assembly files in `APP_PATHS` with common file extensions.

Satellite (resource) assembly probing

To find a satellite assembly for a specific culture, construct a set of file paths.

For each path in `PLATFORM_RESOURCE_ROOTS` and then `APP_PATHS`, append the CultureInfo.Name string, a directory separator, the AssemblyName.Name string, and the extension '.dll'.

If any matching file exists, attempt to load and return it.

Unmanaged (native) library probing

When probing to locate an unmanaged library, the `NATIVE_DLL_SEARCH_DIRECTORIES` are searched looking for a matching library.

Managed assembly loading algorithm

11/12/2019 • 2 minutes to read • Edit Online

Managed assemblies are located and loaded with an algorithm involving various stages.

All managed assemblies except satellite assemblies and `WinRT` assemblies use the same algorithm.

When are managed assemblies loaded?

The most common mechanism to trigger a managed assembly load is a static assembly reference. These references are inserted by the compiler whenever code uses a type defined in another assembly. These assemblies are loaded (`load-by-name`) as needed by the runtime.

The direct use of specific APIs will also trigger loads:

API	DESCRIPTION	`ACTIVE` ASSEMBLYLOADCONTEXT
AssemblyLoadContext.LoadFromAssemblyName	`Load-by-name`	The this instance.
AssemblyLoadContext.LoadFromAssemblyPath AssemblyLoadContext.LoadFromNativeImagePath	Load from path.	The this instance.
AssemblyLoadContext.LoadFromStream	Load from object.	The this instance.
Assembly.LoadFile	Load from path in a new AssemblyLoadContext instance	The new AssemblyLoadContext instance.
Assembly.LoadFrom	Load from path in the AssemblyLoadContext.Default instance. Adds a Resolving handler to AssemblyLoadContext.Default. The handler will load the assembly's dependencies from its directory.	The AssemblyLoadContext.Default instance.

Assembly.Load(AssemblyName)
Assembly.Load(String)
Assembly.LoadWithPartialName
	`Load-by-name`.	Inferred from caller.
		Prefer AssemblyLoadContext methods.

Assembly.Load(Byte[])
Assembly.Load(Byte[], Byte[])
	Load from object in a new AssemblyLoadContext instance.	The new AssemblyLoadContext instance.

Type.GetType(String)
Type.GetType(String, Boolean)
Type.GetType(String, Boolean, Boolean)
	`Load-by-name`.	Inferred from caller.
		Prefer Type.GetType methods with an `assemblyResolver` argument.

API	DESCRIPTION	`ACTIVE` ASSEMBLYLOADCONTEXT
Assembly.GetType	If type `name` describes an assembly qualified generic type, trigger a `Load-by-name`.	Inferred from caller. Prefer Type.GetType when using assembly qualified type names.

Activator.CreateInstance(String, String)
Activator.CreateInstance(String, String, Object[])
Activator.CreateInstance(String, String, Boolean, BindingFlags, Binder, Object[], CultureInfo, Object[])
	`Load-by-name`.	Inferred from caller.
		Prefer Activator.CreateInstance methods taking a Type argument.

Algorithm

The following algorithm describes how the runtime loads a managed assembly.

1. Determine the `active` AssemblyLoadContext.
   - For a static assembly reference, the `active` AssemblyLoadContext is the instance that loaded the referring assembly.
   - Preferred APIs make the `active` AssemblyLoadContext explicit.
   - Other APIs infer the `active` AssemblyLoadContext. For these APIs, the AssemblyLoadContext.CurrentContextualReflectionContext property is used. If its value is `null`, then the inferred AssemblyLoadContext instance is used.
   - See table above.
2. For the `Load-by-name` methods, the active AssemblyLoadContext loads the assembly. In priority order by:
   - Checking its `cache-by-name`.
   - Calling the AssemblyLoadContext.Load function.
   - Checking the AssemblyLoadContext.Default instances' cache and running managed assembly default probing logic.
   - Raising the AssemblyLoadContext.Resolving event for the active AssemblyLoadContext.
   - Raising the AppDomain.AssemblyResolve event.
3. For the other types of loads, the `active` AssemblyLoadContext loads the assembly. In priority order by:
   - Checking its `cache-by-name`.
   - Loading from the specified path or raw assembly object.
4. In either case, if an assembly is newly loaded, then:
   - The AppDomain.AssemblyLoad event is raised.
   - A reference is added to the assembly's AssemblyLoadContext instance's `cache-by-name`.
5. If the assembly is found, a reference is added as needed to the `active` AssemblyLoadContext instance's `cache-by-name`.

Satellite assembly loading algorithm

8/28/2019 • 2 minutes to read • Edit Online

Satellite assemblies are used to store localized resources customized for language and culture.

Satellite assemblies use a different loading algorithm than general managed assemblies.

When are satellite assemblies loaded?

Satellite assemblies are loaded when loading a localized resource.

The basic API to load localized resources is the System.Resources.ResourceManager class. Ultimately the ResourceManager class will call the GetSatelliteAssembly method for each CultureInfo.Name.

Higher-level APIs may abstract the low-level API.

Algorithm

The .NET Core resource fallback process involves the following steps:

1. Determine the `active` AssemblyLoadContext instance. In all cases, the `active` instance is the executing assembly's AssemblyLoadContext.
2. The `active` instance attempts to load a satellite assembly for the requested culture in priority order by:
   - Checking its cache.
   - Checking the directory of the currently executing assembly for a subdirectory that matches the requested CultureInfo.Name (for example `es-MX`).

     > **NOTE**
     >
     > This feature was not implemented in .NET Core before 3.0.

     > **NOTE**
     >
     > On Linux and macOS, the subdirectory is case-sensitive and must either:
     >
     > - Exactly match case.
     > - Be in lower case.

   - If `active` is the AssemblyLoadContext.Default instance, by running the default satellite (resource) assembly probing logic.
   - Calling the AssemblyLoadContext.Load function.
   - Raising the AssemblyLoadContext.Resolving event.
   - Raising the AppDomain.AssemblyResolve event.
3. If a satellite assembly is loaded:
   - The AppDomain.AssemblyLoad event is raised.
   - The assembly is searched it for the requested resource. If the runtime finds the resource in the assembly,

it uses it. If it doesn't find the resource, it continues the search.

> **NOTE**
>
> To find a resource within the satellite assembly, the runtime searches for the resource file requested by the ResourceManager for the current CultureInfo.Name. Within the resource file, it searches for the requested resource name. If either is not found, the resource is treated as not found.

4. The runtime next searches the parent culture assemblies through many potential levels, each time repeating steps 2 & 3.

   Each culture has only one parent, which is defined by the CultureInfo.Parent property.

   The search for parent cultures stops when a culture's Parent property is CultureInfo.InvariantCulture.

   For the InvariantCulture, we don't return to steps 2 & 3, but rather continue with step 5.

5. If the resource is still not found, the resource for the default (fallback) culture is used.

   Typically, the resources for the default culture are included in the main application assembly. However, you can specify UltimateResourceFallbackLocation.Satellite for the NeutralResourcesLanguageAttribute.Location property. This value indicates that the ultimate fallback location for resources is a satellite assembly rather than the main assembly.

   > **NOTE**
   >
   > The default culture is the ultimate fallback. Therefore, we recommend that you always include an exhaustive set of resources in the default resource file. This helps prevent exceptions from being thrown. By having an exhaustive set, you provide a fallback for all resources and ensure that at least one resource is always present for the user, even if it is not culturally specific.

6. Finally,

   - If the runtime doesn't find a resource file for a default (fallback) culture, a MissingManifestResourceException or MissingSatelliteAssemblyException exception is thrown.
   - If the resource file is found but the requested resource isn't present, the request returns `null`.

Unmanaged (native) library loading algorithm

10/10/2019 • 2 minutes to read • Edit Online

Unmanaged libraries are located and loaded with an algorithm involving various stages.

The following algorithm describes how native libraries are loaded through `PInvoke`.

`PInvoke` load library algorithm

`PInvoke` uses the following algorithm when attempting to load an unmanaged assembly:

1. Determine the `active` AssemblyLoadContext. For an unmanaged load library, the `active` AssemblyLoadContext is the one with the assembly that defines the `PInvoke`.
2. For the `active` AssemblyLoadContext, try to find the assembly in priority order by:
   - Checking its cache.
   - Calling the current System.Runtime.InteropServices.DllImportResolver delegate set by the NativeLibrary.SetDllImportResolver(Assembly, DllImportResolver) function.
   - Calling the AssemblyLoadContext.LoadUnmanagedDll function on the `active` AssemblyLoadContext.
   - Checking the AppDomain instance's cache and running the Unmanaged (native) library probing logic.
   - Raising the AssemblyLoadContext.ResolvingUnmanagedDll event for the `active` AssemblyLoadContext.

Create a .NET Core application with plugins

11/7/2019 • 8 minutes to read • Edit Online

This tutorial shows you how to create a custom AssemblyLoadContext to load plugins. An AssemblyDependencyResolver is used to resolve the dependencies of the plugin. The tutorial correctly isolates the plugin's dependencies from the hosting application. You'll learn how to:

- Structure a project to support plugins.
- Create a custom AssemblyLoadContext to load each plugin.
- Use the System.Runtime.Loader.AssemblyDependencyResolver type to allow plugins to have dependencies.
- Author plugins that can be easily deployed by just copying the build artifacts.

Prerequisites

- Install the .NET Core 3.0 SDK or a newer version.

Create the application

The first step is to create the application:

1. Create a new folder, and in that folder run the following command:

   ```
   dotnet new console -o AppWithPlugin
   ```

2. To make building the project easier, create a Visual Studio solution file in the same folder. Run the following command:

   ```
   dotnet new sln
   ```

3. Run the following command to add the app project to the solution:

   ```
   dotnet sln add AppWithPlugin/AppWithPlugin.csproj
   ```

Now we can fill in the skeleton of our application. Replace the code in the *AppWithPlugin/Program.cs* file with the following code:

```
using PluginBase;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Reflection;

namespace AppWithPlugin
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                if (args.Length == 1 && args[0] == "/d")
                {
                    Console.WriteLine("Waiting for any key...");
                    Console.ReadLine();
                }

                // Load commands from plugins.

                if (args.Length == 0)
                {
                    Console.WriteLine("Commands: ");
                    // Output the loaded commands.
                }
                else
                {
                    foreach (string commandName in args)
                    {
                        Console.WriteLine($"-- {commandName} --");

                        // Execute the command with the name passed as an argument.

                        Console.WriteLine();
                    }
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex);
            }
        }
    }
}
```

Create the plugin interfaces

The next step in building an app with plugins is defining the interface the plugins need to implement. We suggest that you make a class library that contains any types that you plan to use for communicating between your app and plugins. This division allows you to publish your plugin interface as a package without having to ship your full application.

In the root folder of the project, run `dotnet new classlib -o PluginBase`. Also, run `dotnet sln add PluginBase/PluginBase.csproj` to add the project to the solution file. Delete the `PluginBase/Class1.cs` file, and create a new file in the `PluginBase` folder named `ICommand.cs` with the following interface definition:

```
namespace PluginBase
{
    public interface ICommand
    {
        string Name { get; }
        string Description { get; }

        int Execute();
    }
}
```

This `ICommand` interface is the interface that all of the plugins will implement.

Now that the `ICommand` interface is defined, the application project can be filled in a little more. Add a reference from the `AppWithPlugin` project to the `PluginBase` project with the `dotnet add AppWithPlugin\AppWithPlugin.csproj reference PluginBase\PluginBase.csproj` command from the root folder.

Replace the `// Load commands from plugins` comment with the following code snippet to enable it to load plugins from given file paths:

```
string[] pluginPaths = new string[]
{
    // Paths to plugins to load.
};

IEnumerable<ICommand> commands = pluginPaths.SelectMany(pluginPath =>
{
    Assembly pluginAssembly = LoadPlugin(pluginPath);
    return CreateCommands(pluginAssembly);
}).ToList();
```

Then replace the `// Output the loaded commands` comment with the following code snippet:

```
foreach (ICommand command in commands)
{
    Console.WriteLine($"{command.Name}\t - {command.Description}");
}
```

Replace the `// Execute the command with the name passed as an argument` comment with the following snippet:

```
ICommand command = commands.FirstOrDefault(c => c.Name == commandName);
if (command == null)
{
    Console.WriteLine("No such command is known.");
    return;
}

command.Execute();
```

And finally, add static methods to the `Program` class named `LoadPlugin` and `CreateCommands`, as shown here:

```
static Assembly LoadPlugin(string relativePath)
{
    throw new NotImplementedException();
}

static IEnumerable<ICommand> CreateCommands(Assembly assembly)
{
    int count = 0;

    foreach (Type type in assembly.GetTypes())
    {
        if (typeof(ICommand).IsAssignableFrom(type))
        {
            ICommand result = Activator.CreateInstance(type) as ICommand;
            if (result != null)
            {
                count++;
                yield return result;
            }
        }
    }

    if (count == 0)
    {
        string availableTypes = string.Join(",", assembly.GetTypes().Select(t => t.FullName));
        throw new ApplicationException(
            $"Can't find any type which implements ICommand in {assembly} from {assembly.Location}.\n" +
            $"Available types: {availableTypes}");
    }
}
```

Load plugins

Now the application can correctly load and instantiate commands from loaded plugin assemblies, but it's still unable to load the plugin assemblies. Create a file named *PluginLoadContext.cs* in the *AppWithPlugin* folder with the following contents:

```
using System;
using System.Reflection;
using System.Runtime.Loader;

namespace AppWithPlugin
{
    class PluginLoadContext : AssemblyLoadContext
    {
        private AssemblyDependencyResolver _resolver;

        public PluginLoadContext(string pluginPath)
        {
            _resolver = new AssemblyDependencyResolver(pluginPath);
        }

        protected override Assembly Load(AssemblyName assemblyName)
        {
            string assemblyPath = _resolver.ResolveAssemblyToPath(assemblyName);
            if (assemblyPath != null)
            {
                return LoadFromAssemblyPath(assemblyPath);
            }

            return null;
        }

        protected override IntPtr LoadUnmanagedDll(string unmanagedDllName)
        {
            string libraryPath = _resolver.ResolveUnmanagedDllToPath(unmanagedDllName);
            if (libraryPath != null)
            {
                return LoadUnmanagedDllFromPath(libraryPath);
            }

            return IntPtr.Zero;
        }
    }
}
```

The `PluginLoadContext` type derives from AssemblyLoadContext. The `AssemblyLoadContext` type is a special type in the runtime that allows developers to isolate loaded assemblies into different groups to ensure that assembly versions don't conflict. Additionally, a custom `AssemblyLoadContext` can choose different paths to load assemblies from and override the default behavior. The `PluginLoadContext` uses an instance of the `AssemblyDependencyResolver` type introduced in .NET Core 3.0 to resolve assembly names to paths. The `AssemblyDependencyResolver` object is constructed with the path to a .NET class library. It resolves assemblies and native libraries to their relative paths based on the *.deps.json* file for the class library whose path was passed to the `AssemblyDependencyResolver` constructor. The custom `AssemblyLoadContext` enables plugins to have their own dependencies, and the `AssemblyDependencyResolver` makes it easy to correctly load the dependencies.

Now that the `AppWithPlugin` project has the `PluginLoadContext` type, update the `Program.LoadPlugin` method with the following body:

```
static Assembly LoadPlugin(string relativePath)
{
    // Navigate up to the solution root
    string root = Path.GetFullPath(Path.Combine(
        Path.GetDirectoryName(
            Path.GetDirectoryName(
                Path.GetDirectoryName(
                    Path.GetDirectoryName(
                        Path.GetDirectoryName(typeof(Program).Assembly.Location)))))));

    string pluginLocation = Path.GetFullPath(Path.Combine(root, relativePath.Replace('\\',
Path.DirectorySeparatorChar)));
    Console.WriteLine($"Loading commands from: {pluginLocation}");
    PluginLoadContext loadContext = new PluginLoadContext(pluginLocation);
    return loadContext.LoadFromAssemblyName(new
AssemblyName(Path.GetFileNameWithoutExtension(pluginLocation)));
}
```

By using a different `PluginLoadContext` instance for each plugin, the plugins can have different or even conflicting dependencies without issue.

Simple plugin with no dependencies

Back in the root folder, do the following:

1. Run the following command to create a new class library project named `HelloPlugin`:

   ```
   dotnet new classlib -o HelloPlugin
   ```

2. Run the following command to add the project to the `AppWithPlugin` solution:

   ```
   dotnet sln add HelloPlugin/HelloPlugin.csproj
   ```

3. Replace the *HelloPlugin/Class1.cs* file with a file named *HelloCommand.cs* with the following contents:

```
using PluginBase;
using System;

namespace HelloPlugin
{
    public class HelloCommand : ICommand
    {
        public string Name { get => "hello"; }
        public string Description { get => "Displays hello message."; }

        public int Execute()
        {
            Console.WriteLine("Hello !!!");
            return 0;
        }
    }
}
```

Now, open the *HelloPlugin.csproj* file. It should look similar to the following:

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

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

</Project>
```

In between the `<Project>` tags, add the following elements:

```
<ItemGroup>
<ProjectReference Include="..\PluginBase\PluginBase.csproj">
    <Private>false</Private>
</ProjectReference>
</ItemGroup>
```

The `<Private>false</Private>` element is important. This tells MSBuild to not copy *PluginBase.dll* to the output directory for HelloPlugin. If the *PluginBase.dll* assembly is present in the output directory, `PluginLoadContext` will find the assembly there and load it when it loads the *HelloPlugin.dll* assembly. At this point, the `HelloPlugin.HelloCommand` type will implement the `ICommand` interface from the *PluginBase.dll* in the output directory of the `HelloPlugin` project, not the `ICommand` interface that is loaded into the default load context. Since the runtime sees these two types as different types from different assemblies, the `AppWithPlugin.Program.CreateCommands` method won't find the commands. As a result, the `<Private>false</Private>` metadata is required for the reference to the assembly containing the plugin interfaces.

Now that the `HelloPlugin` project is complete, we should update the `AppWithPlugin` project to know where the `HelloPlugin` plugin can be found. After the `// Paths to plugins to load` comment, add `@"HelloPlugin\bin\Debug\netcoreapp3.0\HelloPlugin.dll"` as an element of the `pluginPaths` array.

Plugin with library dependencies

Almost all plugins are more complex than a simple "Hello World", and many plugins have dependencies on other libraries. The `JsonPlugin` and `OldJson` plugin projects in the sample show two examples of plugins with NuGet package dependencies on `Newtonsoft.Json`. The project files themselves don't have any special information for the project references, and (after adding the plugin paths to the `pluginPaths` array) the plugins run perfectly, even if run in the same execution of the AppWithPlugin app. However, these projects don't copy the referenced assemblies to their output directory, so the assemblies need to be present on the user's machine for the plugins to work. There are two ways to work around this problem. The first option is to use the `dotnet publish` command to publish the class library. Alternatively, if you want to be able to use the output of `dotnet build` for your plugin, you can add the `<CopyLocalLockFileAssemblies>true</CopyLocalLockFileAssemblies>` property between the `<PropertyGroup>` tags in the plugin's project file. See the `XcopyablePlugin` plugin project for an example.

Other examples in the sample

The complete source code for this tutorial can be found in the dotnet/samples repository. The completed sample includes a few other examples of `AssemblyDependencyResolver` behavior. For example, the `AssemblyDependencyResolver` object can also resolve native libraries as well as localized satellite assemblies included in NuGet packages. The `UVPlugin` and `FrenchPlugin` in the samples repository demonstrate these scenarios.

Reference a plugin from a NuGet package

Let's say that there is an app A that has a plugin interface defined in the NuGet package named `A.PluginBase`. How do you reference the package correctly in your plugin project? For project references, using the `<Private>false</Private>` metadata on the `ProjectReference` element in the project file prevented the dll from being copied to the output.

To correctly reference the `A.PluginBase` package, you want to change the `<PackageReference>` element in the project file to the following:

```
<PackageReference Include="A.PluginBase" Version="1.0.0">
    <ExcludeAssets>runtime</ExcludeAssets>
</PackageReference>
```

This prevents the `A.PluginBase` assemblies from being copied to the output directory of your plugin and ensures that your plugin will use A's version of `A.PluginBase`.

Plugin target framework recommendations

Because plugin dependency loading uses the *.deps.json* file, there is a gotcha related to the plugin's target framework. Specifically, your plugins should target a runtime, such as .NET Core 3.0, instead of a version of .NET Standard. The *.deps.json* file is generated based on which framework the project targets, and since many .NET Standard-compatible packages ship reference assemblies for building against .NET Standard and implementation assemblies for specific runtimes, the *.deps.json* may not correctly see implementation assemblies, or it may grab the .NET Standard version of an assembly instead of the .NET Core version you expect.

Plugin framework references

Currently, plugins can't introduce new frameworks into the process. For example, you can't load a plugin that uses the `Microsoft.AspNetCore.App` framework into an application that only uses the root `Microsoft.NETCore.App` framework. The host application must declare references to all frameworks needed by plugins.

2 minutes to read

Build .NET Core from source

8/28/2019 • 3 minutes to read • Edit Online

The ability to build .NET Core from its source code is important in multiple ways: it makes it easier to port .NET Core to new platforms, it enables contributions and fixes to the product, and it enables the creation of custom versions of .NET. This article gives guidance to developers who want to build and distribute their own versions of .NET Core.

Build the CLR from source

The source code for the .NET CoreCLR can be found in the dotnet/coreclr repository on GitHub.

The build currently depends on the following prerequisites:

- Git
- CMake
- Python
- a C++ compiler.

After you've installed these prerequisites, you can build the CLR by invoking the build script (`build.cmd` on Windows, or `build.sh` on Linux and macOS) at the base of the dotnet/coreclr repository.

Installing the components differ depending on the operating system (OS). See the build instructions for your specific OS:

- Windows
- Linux
- macOS
- FreeBSD
- NetBSD

There is no cross-building across OS (only for ARM, which is built on X64).
You have to be on the particular platform to build that platform.

The build has two main `buildTypes`:

- Debug (default)- Compiles the runtime with minimal optimizations and additional runtime checks (asserts). This reduction in optimization level and the additional checks slow runtime execution but are valuable for debugging. This is the recommended setting for development and testing environments.
- Release - Compiles the runtime with full optimizations and without the additional runtime checks. This will yield much faster run time performance but it can take a bit longer to build and can be difficult to debug. Pass `release` to the build script to select this build type.

In addition, by default the build not only creates the runtime executables, but it also builds all the tests. There are quite a few tests, taking a significant amount of time that isn't necessary if you just want to experiment with changes. You can skip the tests builds by adding the `skiptests` argument to the build script, like in the following example (replace `.\build` with `./build.sh` on Unix machines):

```
.\build skiptests
```

The previous example showed how to build the `Debug` flavor, which has development time checks (asserts) enabled and optimizations disabled. To build the release (full speed) flavor, do the following:

```
.\build release skiptests
```

You can find more build options with build by using the -? or -help qualifier.

Using Your Build

The build places all of its generated files under the `bin` directory at the base of the repository. There is a *bin\Log* directory that contains log files generated during the build (Most useful when the build fails). The actual output is placed in a *bin\Product[platform].[CPU architecture].[build type]* directory, such as *bin\Product\Windows_NT.x64.Release*.

While the 'raw' output of the build is sometimes useful, normally you're only interested in the NuGet packages, which are placed in the `.nuget\pkg` subdirectory of the previous output directory.

There are two basic techniques for using your new runtime:

1. **Use dotnet.exe and NuGet to compose an application**. See Using Your Build for instructions on creating a program that uses your new runtime by using the NuGet packages you just created and the 'dotnet' command-line interface (CLI). This technique is the expected way non-runtime developers are likely to consume your new runtime.

2. **Use corerun.exe to run an application using unpackaged DLLs**. This repository also defines a simple host called corerun.exe that does NOT take any dependency on NuGet. You need to tell the host where to get the required DLLs you actually use, and you have to manually gather them together. This technique is used by all the tests in the dotnet/coreclr repo, and is useful for quick local 'edit-compile-debug' loop such as preliminary unit testing. See Executing .NET Core Apps with CoreRun.exe for details on using this technique.

Build the CLI from source

The source code for the .NET Core CLI can be found in the dotnet/cli repository on GitHub.

In order to build the .NET Core CLI, you need the following installed on your machine.

- Windows & Linux:
  - git on the PATH
- macOS:
  - git on the PATH
  - Xcode
  - OpenSSL

In order to build, run `build.cmd` on Windows, or `build.sh` on Linux and macOS from the root. If you don't want to execute tests, run `build.cmd -t:Compile` or `./build.sh -t:Compile`. To build the CLI in macOS Sierra, you need to set the DOTNET_RUNTIME_ID environment variable by running `export DOTNET_RUNTIME_ID=osx.10.11-x64`.

Using your build

Use the `dotnet` executable from *artifacts/{os}-{arch}/stage2* to try out the newly built CLI. If you want to use the build output when invoking `dotnet` from the current console, you can also add *artifacts/{os}-{arch}/stage2* to the PATH.

See also

- .NET Core Common Language Runtime (CoreCLR)

- .NET Core CLI Developer Guide
- .NET Core distribution packaging

.NET Core distribution packaging

10/16/2019 • 6 minutes to read • Edit Online

As .NET Core becomes available on more and more platforms, it's useful to learn how to package, name, and version it. This way, package maintainers can help ensure a consistent experience no matter where users choose to run .NET. This article is useful for users that are:

- Attempting to build .NET Core from source.
- Wanting to make changes to the .NET Core CLI that could impact the resulting layout or packages produced.

Disk layout

When installed, .NET Core consists of several components that are laid out as follows in the filesystem:

```
{dotnet_root}                                    (*)
├── dotnet                       (1)
├── LICENSE.txt                  (8)
├── ThirdPartyNotices.txt        (8)
├── host                                         (*)
│   └── fxr                                      (*)
│       └── <fxr version>        (2)
├── sdk                                          (*)
│   ├── <sdk version>            (3)
│   └── NuGetFallbackFolder      (4)             (*)
├── packs                                        (*)
│   ├── Microsoft.AspNetCore.App.Ref             (*)
│   │   └── <aspnetcore ref version>     (11)
│   ├── Microsoft.NETCore.App.Ref                (*)
│   │   └── <netcore ref version>        (12)
│   ├── Microsoft.NETCore.App.Host.<rid>         (*)
│   │   └── <apphost version>            (13)
│   ├── Microsoft.WindowsDesktop.App.Ref         (*)
│   │   └── <desktop ref version>        (14)
│   └── NETStandard.Library.Ref                  (*)
│       └── <netstandard version>        (15)
├── shared                                       (*)
│   ├── Microsoft.NETCore.App                    (*)
│   │   └── <runtime version>     (5)
│   ├── Microsoft.AspNetCore.App                 (*)
│   │   └── <aspnetcore version>  (6)
│   ├── Microsoft.AspNetCore.All                 (*)
│   │   └── <aspnetcore version>  (6)
│   └── Microsoft.WindowsDesktop.App             (*)
│       └── <desktop app version> (7)
└── templates                                    (*)
│   └── <templates version>      (17)
/
├── etc/dotnet
│       └── install_location     (16)
├── usr/share/man/man1
│       └── dotnet.1.gz          (9)
└── usr/bin
        └── dotnet               (10)
```

- (1) **dotnet** The host (also known as the "muxer") has two distinct roles: activate a runtime to launch an application, and activate an SDK to dispatch commands to it. The host is a native executable (`dotnet.exe`).

While there's a single host, most of the other components are in versioned directories (2,3,5,6). This means multiple versions can be present on the system since they're installed side by side.

- (2) **host/fxr/<fxr version>** contains the framework resolution logic used by the host. The host uses the latest hostfxr that is installed. The hostfxr is responsible for selecting the appropriate runtime when executing a .NET Core application. For example, an application built for .NET Core 2.0.0 uses the 2.0.5 runtime when it's available. Similarly, hostfxr selects the appropriate SDK during development.
- (3) **sdk/<sdk version>** The SDK (also known as "the tooling") is a set of managed tools that are used to write and build .NET Core libraries and applications. The SDK includes the .NET Core Command-line interface (CLI), the managed languages compilers, MSBuild, and associated build tasks and targets, NuGet, new project templates, and so on.
- (4) **sdk/NuGetFallbackFolder** contains a cache of NuGet packages used by an SDK during the restore operation, such as when running `dotnet restore` or `dotnet build`. This folder is only used prior to .NET Core 3.0. It can't be built from source, because it contains prebuilt binary assets from `nuget.org`.

The **shared** folder contains frameworks. A shared framework provides a set of libraries at a central location so they can be used by different applications.

- (5) **shared/Microsoft.NETCore.App/<runtime version>** This framework contains the .NET Core runtime and supporting managed libraries.
- (6) **shared/Microsoft.AspNetCore.{App,All}/<aspnetcore version>** contains the ASP.NET Core libraries. The libraries under `Microsoft.AspNetCore.App` are developed and supported as part of the .NET Core project. The libraries under `Microsoft.AspNetCore.All` are a superset that also contains third-party libraries.
- (7) **shared/Microsoft.Desktop.App/<desktop app version>** contains the Windows desktop libraries. This isn't included on non-Windows platforms.
- (8) **LICENSE.txt,ThirdPartyNotices.txt** are the .NET Core license and licenses of third-party libraries used in .NET Core, respectively.
- (9,10) **dotnet.1.gz, dotnet** `dotnet.1.gz` is the dotnet manual page. `dotnet` is a symlink to the dotnet host(1). These files are installed at well known locations for system integration.
- (11,12) **Microsoft.NETCore.App.Ref,Microsoft.AspNetCore.App.Ref** describe the API of an `x.y` version of .NET Core and ASP.NET Core respectively. These packs are used when compiling for those target versions.
- (13) **Microsoft.NETCore.App.Host.<rid>** contains a native binary for platform `rid`. This binary is a template when compiling a .NET Core application into a native binary for that platform.
- (14) **Microsoft.WindowsDesktop.App.Ref** describes the API of `x.y` version of Windows Desktop applications. These files are used when compiling for that target. This isn't provided on non-Windows platforms.
- (15) **NETStandard.Library.Ref** describes the netstandard `x.y` API. These files are used when compiling for that target.
- (16) **/etc/dotnet/install_location** is a file that contains the full path for `{dotnet_root}`. The path may end with a newline. It's not necessary to add this file when the root is `/usr/share/dotnet`.
- (17) **templates** contains the templates used by the SDK. For example, `dotnet new` finds project templates here.

The folders marked with `(*)` are used by multiple packages. Some package formats (for example, `rpm`) require special handling of such folders. The package maintainer must take care of this.

Recommended packages

.NET Core versioning is based on the runtime component `[major].[minor]` version numbers. The SDK version uses the same `[major].[minor]` and has an independent `[patch]` that combines feature and patch semantics for the SDK. For example: SDK version 2.2.302 is the second patch release of the third feature release of the SDK that supports the 2.2 runtime. For more information about how versioning works, see .NET Core versioning overview.

Some of the packages include part of the version number in their name. This allows you to install a specific version. The rest of the version isn't included in the version name. This allows the OS package manager to update the packages (for example, automatically installing security fixes). Supported package managers are Linux specific.

The following lists the recommended packages:

- `dotnet-sdk-[major].[minor]` - Installs the latest sdk for specific runtime
  - **Version:** <runtime version>
  - **Example:** dotnet-sdk-2.1
  - **Contains:** (3),(4)
  - **Dependencies:** `dotnet-runtime-[major].[minor]`, `aspnetcore-runtime-[major].[minor]`, `dotnet-targeting-pack-[major].[minor]`, `aspnetcore-targeting-pack-[major].[minor]`, `netstandard-targeting-pack-[netstandard_major].[netstandard_minor]`, `dotnet-apphost-pack-[major].[minor]`, `dotnet-templates-[major].[minor]`
- `aspnetcore-runtime-[major].[minor]` - Installs a specific ASP.NET Core runtime
  - **Version:** <aspnetcore runtime version>
  - **Example:** aspnetcore-runtime-2.1
  - **Contains:** (6)
  - **Dependencies:** `dotnet-runtime-[major].[minor]`
- `dotnet-runtime-deps-[major].[minor]` *(Optional)* - Installs the dependencies for running self-contained applications
  - **Version:** <runtime version>
  - **Example:** dotnet-runtime-deps-2.1
  - **Dependencies:** *distro specific dependencies*
- `dotnet-runtime-[major].[minor]` - Installs a specific runtime
  - **Version:** <runtime version>
  - **Example:** dotnet-runtime-2.1
  - **Contains:** (5)
  - **Dependencies:** `dotnet-hostfxr-[major].[minor]`, `dotnet-runtime-deps-[major].[minor]`
- `dotnet-hostfxr-[major].[minor]` - dependency
  - **Version:** <runtime version>
  - **Example:** dotnet-hostfxr-3.0
  - **Contains:** (2)
  - **Dependencies:** `dotnet-host`
- `dotnet-host` - dependency
  - **Version:** <runtime version>
  - **Example:** dotnet-host
  - **Contains:** (1),(8),(9),(10),(16)
- `dotnet-apphost-pack-[major].[minor]` - dependency

    - **Version:** <runtime version>
    - **Contains:** (13)
- `dotnet-targeting-pack-[major].[minor]` - Allows targeting a non-latest runtime
    - **Version:** <runtime version>
    - **Contains:** (12)
- `aspnetcore-targeting-pack-[major].[minor]` - Allows targeting a non-latest runtime
    - **Version:** <aspnetcore runtime version>
    - **Contains:** (11)
- `netstandard-targeting-pack-[netstandard_major].[netstandard_minor]` - Allows targeting a netstandard version
    - **Version:** <sdk version>
    - **Contains:** (15)
- `dotnet-templates-[major].[minor]`
    - **Version:** <sdk version>
    - **Contains:** (15)

The `dotnet-runtime-deps-[major].[minor]` requires understanding the *distro-specific dependencies*. Because the distro build system may be able to derive this automatically, the package is optional, in which case these dependencies are added directly to the `dotnet-runtime-[major].[minor]` package.

When package content is under a versioned folder, the package name `[major].[minor]` match the versioned folder name. For all packages, except the `netstandard-targeting-pack-[netstandard_major].[netstandard_minor]`, this also matches with the .NET Core version.

Dependencies between packages should use an *equal or greater than* version requirement. For example, `dotnet-sdk-2.2:2.2.401` requires `aspnetcore-runtime-2.2 >= 2.2.6`. This makes it possible for the user to upgrade their installation via a root package (for example, `dnf update dotnet-sdk-2.2`).

Most distributions require all artifacts to be built from source. This has some impact on the packages:

- The third-party libraries under `shared/Microsoft.AspNetCore.All` can't be easily built from source. So that folder is omitted from the `aspnetcore-runtime` package.
- The `NuGetFallbackFolder` is populated using binary artifacts from `nuget.org`. It should remain empty.

Multiple `dotnet-sdk` packages may provide the same files for the `NuGetFallbackFolder`. To avoid issues with the package manager, these files should be identical (checksum, modification date, and so on).

Building packages

The dotnet/source-build repository provides instructions on how to build a source tarball of the .NET Core SDK and all its components. The output of the source-build repository matches the layout described in the first section of this article.

project.json and Visual Studio 2015 with .NET Core

1/17/2019 • 2 minutes to read • Edit Online

On March 7, 2017, the .NET Core and ASP.NET Core documentation was updated for the release of Visual Studio 2017. The previous version of the documentation used Visual Studio 2015 and pre-release tooling based on the *project.json* file.

The documentation version from before the March 7 update is available in a PDF file and in a branch in the documentation repository.

PDF documentation

The best source of the earlier documentation is the .NET Core - PDF for project.json and Visual Studio 2015.

Documentation repository branch

You can view the earlier version of the documentation in the repository, but many links won't work and many code snippets are references that aren't expanded.

- .NET Core - project.json branch in the documentation repository

Current version of the documentation

- .NET Core documentation
- ASP.NET Core documentation