

.NET fundamentals documentation

Learn the fundamentals of .NET, an open-source developer platform for building many different types of applications.

Learn about .NET

DOWNLOAD

[Download .NET](#) ↗

OVERVIEW

[What is .NET?](#) ↗

[Introduction to .NET](#)

[.NET languages](#)

CONCEPT

[.NET Standard](#)

[Common Language Runtime \(CLR\)](#)

[.NET Core support policy](#) ↗

WHAT'S NEW

[What's new in .NET 9](#)

[What's new in .NET 8](#)

[What's new in .NET 7](#)

[What's new in .NET 6](#)

Install .NET

OVERVIEW

[Select which .NET version to use](#)

HOW-TO GUIDE

[Install .NET SDK](#)

[Install .NET runtime](#)

[Use a Linux package manager to install](#)

[Check installed versions](#)

REFERENCE

[.NET SDK and runtime dependencies](#)

Get started with .NET

GET STARTED

[Get started with .NET](#)

[Get started with ASP.NET Core](#)

[.NET on Q&A](#)

[.NET tech community forums](#) ↗

VIDEO

[Tutorial: Hello World in 10 minutes](#) ↗

TUTORIAL

[Create a Hello World app in Visual Studio Code](#)

[Create a Hello World app in Visual Studio](#)

[Containerize a .NET Core app](#)

CONCEPT

[Port from .NET Framework to .NET](#)

DEPLOY

[App publishing](#)

[Publish .NET apps with GitHub Actions](#)

Serialize data

CONCEPT

[Serialize and deserialize JSON](#)

HOW-TO GUIDE

[Serialize and deserialize JSON using C#](#)

[Migrate from Newtonsoft.Json to System.Text.Json](#)

[Write custom converters for JSON serialization](#)

SAMPLE

[Examples of XML serialization](#)

Runtime libraries

OVERVIEW

[Runtime libraries overview](#)

CONCEPT

[Dependency injection in .NET](#)

[Configuration in .NET](#)

[Logging in .NET](#)

[.NET generic host](#)

[Worker services in .NET](#)

[Caching in .NET](#)

[HTTP in .NET](#)

[Localization in .NET](#)

[File globbing in .NET](#)

TUTORIAL

[Implement a custom configuration provider](#)

[Compile-time logging source generation](#)

[Create a Windows Service using BackgroundService](#)

Format and convert dates, numbers, and strings

CONCEPT

[Numeric format strings](#)

[Date and time format strings](#)

[Composite formatting](#)

[Convert times between time zones](#)

[Trim and remove characters from strings](#)

[Regular expressions in .NET](#)

HOW-TO GUIDE

[Convert strings to DateTime](#)

[Pad a number with leading zeros](#)

[Display milliseconds in date and time values](#)

REFERENCE

[Regular expression language](#)

Use events and exceptions

CONCEPT

[Best practices for exceptions](#)

[Handle and raise events](#)

HOW-TO GUIDE

[Use a try-catch block to catch exceptions](#)

File and stream I/O

CONCEPT

[File and stream I/O](#)

[File path formats on Windows](#)

HOW-TO GUIDE

[Write text to a file](#)

[Read text from a file](#)

[Compress and extract files](#)

[Open and append to a log file](#)

Get started with .NET

Article • 08/12/2022

This article teaches you how to create and run a "Hello World!" app with [.NET](#).

Create an application

First, download and install the [.NET SDK](#) on your computer.

Next, open a terminal such as **PowerShell**, **Command Prompt**, or **bash**.

Type the following commands:

.NET CLI

```
dotnet new console -o sample1
cd sample1
dotnet run
```

You should see the following output:

Output

```
Hello World!
```

Congratulations! You've created a simple .NET application.

Next steps

Get started on developing .NET applications by following a [step-by-step tutorial](#) or by watching [.NET 101 videos](#) on YouTube.

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

Tutorials for getting started with .NET

Article • 09/08/2023

The following step-by-step tutorials run on Windows, Linux, or macOS, except as noted.

Tutorials for creating apps

- Create a console app
 - [using Visual Studio Code](#)
 - [using Visual Studio \(Windows\)](#)
- Create a web app
 - [with server-side web UI](#)
 - [with client-side web UI](#)
- Create a web API
- Create a remote procedure call web app
- Create a real-time web app
- Create a serverless function in the cloud
- [Create a mobile app for Android and iOS](#) ↗ (Windows)
- Create a Windows desktop app
 - [WPF](#)
 - [Windows Forms](#)
 - [Universal Windows Platform \(UWP\)](#)
- [Create a game using Unity](#) ↗
- Create a Windows service

Tutorials for creating class libraries

- Create a class library
 - [using Visual Studio Code](#)
 - [using Visual Studio \(Windows\)](#)

Resources for learning .NET languages

- [Get started with C#](#)
- [Get started with F#](#)
- [Get started with Visual Basic](#)

Other get-started resources

The following resources are for getting started with developing .NET apps but aren't step-by-step tutorials:

- [Internet of Things \(IoT\)](#)
- [Machine learning](#)

Next steps

To learn more about .NET, see [Introduction to .NET](#).

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install .NET on Windows, Linux, and macOS

Learn about installing .NET on Windows, Linux, and macOS. Discover the dependencies required to develop, deploy, and run .NET apps.

Windows

OVERVIEW

[Install on Windows](#)

[Supported Windows releases](#)

[Dependencies](#)

[Install with Visual Studio](#)

[Install alongside Visual Studio Code](#)

macOS

OVERVIEW

[Install on macOS](#)

[Supported macOS releases](#)

[Install alongside Visual Studio Code](#)

Linux

OVERVIEW

[Linux overview](#)

[Alpine](#)

[CentOS](#)

[Debian](#)

[Fedora](#)

[openSUSE](#)

[Red Hat Enterprise Linux and CentOS Stream](#)

[SUSE Linux Enterprise Server](#)

[Ubuntu](#)

Q&A

[GET STARTED](#)

[Standalone installers](#)

[Visual Studio installers](#)

[Linux feeds](#)

[Docker](#)

[Enterprise deployment](#)

Install .NET on Windows

Article • 12/19/2023

In this article, you learn how to install .NET on Windows. .NET is made up of the runtime and the SDK. The runtime is used to run a .NET app and might be included with the app. The SDK is used to create .NET apps and libraries. The .NET runtime is always installed with the SDK.

The latest version of .NET is 8.0.

[Download .NET](#)

There are two types of supported releases: Long Term Support (LTS) releases and Standard Term Support (STS) releases. The quality of all releases is the same. The only difference is the length of support. LTS releases get free support and patches for three years. STS releases get free support and patches for 18 months. For more information, see [.NET Support Policy](#).

The following table lists the support status of each version of .NET (and .NET Core):

[Expand table](#)

✓ Supported	✗ Unsupported
8 (LTS)	5
7 (STS)	3.1
6 (LTS)	3.0
	2.1
	2.0
	1.1
	1.0

Install with Windows Package Manager (winget)

You can install and manage .NET through the Windows Package Manager service, using the `winget` tool. For more information about how to install and use `winget`, see [Use the](#)

winget tool.

If you're installing .NET system-wide, install with administrative privileges.

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtimes. To install the .NET SDK, run the following command:

Windows Command Prompt

```
winget install Microsoft.DotNet.SDK.8
```

Install the runtime

There are three different .NET runtimes you can install, however, you should install both the .NET Desktop Runtime and the ASP.NET Core Runtime for maximum compatibility with all types of .NET apps. The following table describes what is included with each runtime:

Expand table

	Includes .NET Runtime	Includes .NET Desktop Runtime	Includes ASP.NET Core Runtime
.NET Runtime	Yes	No	No
.NET Desktop Runtime	Yes	Yes	No
ASP.NET Core Runtime	No	No	Yes

The following list provides details about each runtime along with the `winget` commands to install them:

- .NET Desktop Runtime

This runtime supports Windows Presentation Foundation (WPF) and Windows Forms apps that are built with .NET. This isn't the same as .NET Framework, which comes with Windows. This runtime includes .NET Runtime, but doesn't include ASP.NET Core Runtime, which must be installed separately.

```
Windows Command Prompt
```

```
winget install Microsoft.DotNet.DesktopRuntime.8
```

- **.NET Runtime**

This is the base runtime, and contains just the components needed to run a console app. Typically, you'd install both .NET Desktop Runtime and ASP.NET Core Runtime instead of this one.

```
Windows Command Prompt
```

```
winget install Microsoft.DotNet.Runtime.8
```

- **ASP.NET Core Runtime**

This runtime runs web server apps and provides many web-related APIs. ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. You must install .NET Runtime in addition to this runtime. The following commands install ASP.NET Core Runtime, In your terminal, run the following commands:

```
Windows Command Prompt
```

```
winget install Microsoft.DotNet.AspNetCore.8
```

You can install preview versions of the runtimes by substituting the version number, such as `6`, with the word `Preview`. The following example installs the preview release of the .NET Desktop Runtime:

```
Windows Command Prompt
```

```
winget install Microsoft.DotNet.DesktopRuntime.Preview
```

Install alongside Visual Studio Code

Visual Studio Code is a powerful and lightweight source code editor that runs on your desktop. Visual Studio Code is available for Windows, macOS, and Linux.

While Visual Studio Code doesn't come with an automated .NET Core installer like Visual Studio does, adding .NET Core support is simple.

1. [Download and install Visual Studio Code](#).

2. [Download and install the .NET SDK](#).
3. [Install the C# extension from the Visual Studio Code marketplace](#).

The **C# For Visual Studio Code** extension includes the latest .NET SDK, and you don't need to install any .NET runtime separately.

Install with Windows Installer

There are three different .NET runtimes you can install, however, you should install both the .NET Desktop Runtime and the ASP.NET Core Runtime for maximum compatibility with all types of .NET apps. The following table describes what is included with each runtime:

Expand table

	Includes .NET Runtime	Includes .NET Desktop Runtime	Includes ASP.NET Core Runtime
.NET Runtime	Yes	No	No
.NET Desktop Runtime	Yes	Yes	No
ASP.NET Core Runtime	No	No	Yes

.NET SDK allows you to create .NET apps, and includes all of the runtimes.

The [download page](#) for .NET provides Windows Installer executables.

If you want to install .NET silently, such as in a production environment or to support continuous integration, use the following switches:

- `/install`
Installs .NET.
- `/quiet`
Prevents any UI and prompts from displaying.
- `/norestart`
Suppresses any attempts to restart.

Console

```
dotnet-sdk-8.0.100-win-x64.exe /install /quiet /norestart
```

For more information, see [Standard Installer Command-Line Options](#).

Tip

The installer returns an exit code of **0** for success and an exit code of **3010** to indicate that a restart is required. Any other value is generally an error code.

Install with PowerShell automation

The [dotnet-install scripts](#) are used for CI automation and non-admin installs of the runtime. You can download the script from the [dotnet-install script reference page](#).

The script defaults to installing the latest [long term support \(LTS\)](#) version, which is .NET 8. You can choose a specific release by specifying the `Channel` switch. Include the `Runtime` switch to install a runtime. Otherwise, the script installs the SDK.

The following command installs both the Desktop and ASP.NET Core runtimes for maximum compatibility.

```
PowerShell
dotnet-install.ps1 -Channel 8.0 -Runtime windowsdesktop
dotnet-install.ps1 -Channel 8.0 -Runtime aspnetcore
```

Install the SDK by omitting the `-Runtime` switch. The `-Channel` switch is set in this example to `STS`, which installs the latest Standard Term Support version, which is .NET 7.

```
PowerShell
dotnet-install.ps1 -Channel STS
```

Install with Visual Studio

If you're using Visual Studio to develop .NET apps, the following table describes the minimum required version of Visual Studio based on the target .NET SDK version.

[] [Expand table](#)

.NET SDK version	Visual Studio version
8	Visual Studio 2022 version 17.8 or higher.

.NET SDK version	Visual Studio version
7	Visual Studio 2022 version 17.4 or higher.
6	Visual Studio 2022 version 17.0 or higher.
5	Visual Studio 2019 version 16.8 or higher.
3.1	Visual Studio 2019 version 16.4 or higher.
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.

If you already have Visual Studio installed, you can check your version with the following steps.

1. Open Visual Studio.
2. Select **Help > About Microsoft Visual Studio**.
3. Read the version number from the **About** dialog.

Visual Studio can install the latest .NET SDK and runtime.

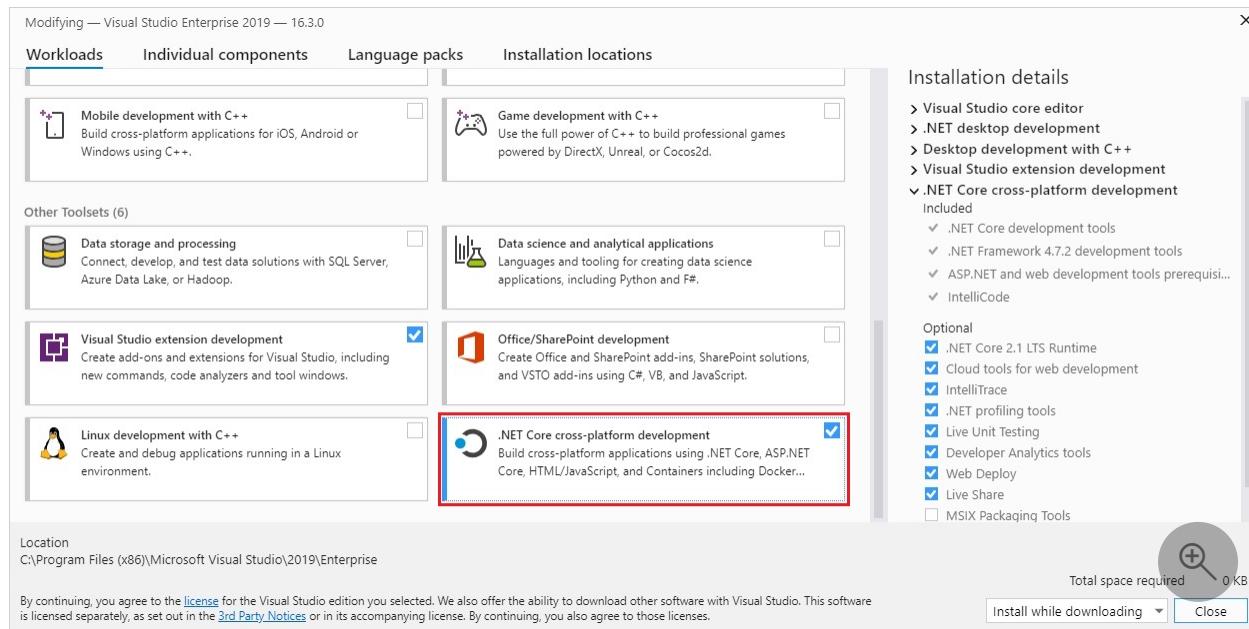
[Download Visual Studio](#)

For more information, see [.NET SDK, MSBuild, and Visual Studio versioning](#).

Select a workload

When installing or modifying Visual Studio, select one or more of the following workloads, 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 **Azure development** workload in the **Web & Cloud** section.
- The **.NET desktop development** workload in the **Desktop & Mobile** section.



Supported releases

The following table is a list of currently supported .NET releases and the versions of Windows they're supported on. These versions remain supported until either the version of [.NET reaches end-of-support](#) or the version of [Windows reaches end-of-life](#).

Windows 10 versions end-of-service dates are segmented by edition. Only **Home**, **Pro**, **Pro Education**, and **Pro for Workstations** editions are considered in the following table. Check the [Windows lifecycle fact sheet](#) for specific details.

Tip

A + symbol represents the minimum version.

[Expand table](#)

Operating System	.NET 8	.NET 7	.NET 6
Windows 11	✓	✓	✓
Windows Server 2022	✓	✓	✓
Windows Server, Version 1903 or later	✓	✓	✓
Windows 10, Version 1607 or later	✓	✓	✓
Windows 8.1	✗	✗	✓
Windows 7 SP1 ESU	✗	✗	✓

Operating System	.NET 8	.NET 7	.NET 6
Windows Server 2019	✓	✓	✓
Windows Server 2016			
Windows Server 2012 R2			
Windows Server 2012			
Windows Server Core 2012 R2	✓	✓	✓
Windows Server Core 2012	✓	✓	✓
Nano Server, Version 1809+	✓	✓	✓
Nano Server, Version 1803	✗	✗	✗

For more information about .NET 8 supported operating systems, distributions, and lifecycle policy, see [.NET 8 Supported OS Versions](#).

Unsupported releases

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Verify downloaded binaries

After downloading an installer, verify it to make sure that the file hasn't been changed or corrupted. You can verify the checksum on your computer and then compare it to what was reported on the download website.

When you download an installer or binary from an official download page, the checksum for the file is displayed. Select the **Copy** button to copy the checksum value to your clipboard.

Thanks for downloading .NET 8.0 SDK (v8.0.100) - Windows x64 Installer!

 **Using Visual Studio?** This release is only compatible with Visual Studio 2022 (v17.8). Using a different version? See [.NET SDKs for Visual Studio](#).

If your download doesn't start after 30 seconds, [click here to download manually](#).

Direct link <https://download.visualstudio.microsoft.com/download/pr/93961dfb-d1e0-49c8-9230-abcba1ebab5a/811ed1eb63d7652325727720edda26a8/dotnet-sdk-8.0.100-win-x64.exe>

Checksum (SHA512) 248acec95b381e5302255310fb9396267fd74a4a2dc2c3a5989031969cb31f8270cbd14bda1b0352ac90f8138bddad1a58e4af1e56cc4a1613b1cf2854b5

You can use **PowerShell** or **Command Prompt** to validate the checksum of the file you've downloaded. For example, the following command reports the checksum of the *dotnet-sdk-8.0.100-win-x64.exe* file:

Windows Command Prompt

```
> certutil -hashfile dotnet-sdk-8.0.100-win-x64.exe SHA512
SHA512 hash of dotnet-sdk-8.0.100-win-x64.exe:
248acec95b381e5302255310fb9396267fd74a4a2dc2c3a5989031969cb31f8270cbd14bda1b
c0352ac90f8138bddad1a58e4af1e56cc4a1613b1cf2854b518e
CertUtil: -hashfile command completed successfully.
```

PowerShell

```
> (Get-FileHash .\dotnet-sdk-8.0.100-win-x64.exe -Algorithm SHA512).Hash
248acec95b381e5302255310fb9396267fd74a4a2dc2c3a5989031969cb31f8270cbd14bda1b
c0352ac90f8138bddad1a58e4af1e56cc4a1613b1cf2854b518e
```

Compare the checksum with the value provided by the download site.

Use PowerShell and a checksum file to validate

The .NET release notes contain a link to a checksum file you can use to validate your downloaded file. The following steps describe how to download the checksum file and validate a .NET install binary:

1. The release notes page for .NET 8 on GitHub at <https://github.com/dotnet/core/tree/main/release-notes/8.0> contains a section named **Releases**. The table in that section links to the downloads and checksum files for each .NET 8 release:

Releases

Date	Release
2023/11/14	8.0.0
2023/10/10	8.0.0 RC 2
2023/09/12	8.0.0 RC 1
2023/08/08	8.0.0 Preview 7
2023/07/11	8.0.0 Preview 6

2. Select the link for the version of .NET that you downloaded. The previous section used .NET SDK 8.0.100, which is in the .NET 8.0.0 release.

Tip

If you're not sure which .NET release contains your checksum file, explore the links until you find it.

3. In the release page, you can see the .NET Runtime and .NET SDK version, and a link to the checksum file:

.NET 8.0.0 - November 14, 2023

The [.NET 8.0.0 and .NET SDK 8.0.100](#) releases are available for download. The latest 8.0 release is always listed at [.NET 8.0 Releases](#).

Downloads

	SDK Installer ¹	SDK Binaries ¹	Runtime Installer	Runtime Binaries	ASP.NET Core Runtime	Windows Desktop Runtime
Windows	x86 x64 Arm64	x86 x64 Arm64	x86 x64 Arm64	x86 x64 Arm64	x86 x64 Hosting Bundle²	x86 x64 Arm64
macOS	x64 ARM64	x64 ARM64	x64 ARM64	x64 ARM64	x64 ARM64	-
Linux	Snap and Package Manager	x64 Arm Arm64 Arm32 Alpine x64 Alpine	Packages (x64)	x64 Arm Arm64 Arm32 Alpine Arm64 Alpine x64 Alpine	x64¹ Arm¹ Arm64¹ x64 Alpine	-
	Checksums	Checksums	Checksums	Checksums	Checksums	Checksums

4. Copy the link to the checksum file.

5. Use the following script, but replace the link to download the appropriate checksum file:

```
PowerShell
```

```
Invoke-WebRequest  
https://dotnetcli.blob.core.windows.net/dotnet/checksums/8.0.0-sha.txt  
-OutFile 8.0.0-sha.txt
```

- With both the checksum file and the .NET release file downloaded to the same directory, search the checksum file for the checksum of the .NET download:

When validation passes, you see **True** printed:

PowerShell

```
> (Get-Content .\8.0.0-sha.txt | Select-String "dotnet-sdk-8.0.100-win-x64.exe").Line -like (Get-FileHash .\dotnet-sdk-8.0.100-win-x64.exe -Algorithm SHA512).Hash + "*"  
True
```

If you see **False** printed, the file you downloaded isn't valid and shouldn't be used.

Runtime information

The runtime is used to run apps created with .NET. When an app author publishes an app, they can include the runtime with their app. If they don't include the runtime, it's up to the user to install the runtime.

There are three different .NET runtimes you can install, however, you should install both the .NET Desktop Runtime and the ASP.NET Core Runtime for maximum compatibility with all types of .NET apps. The following table describes what is included with each runtime:

[+] Expand table

	Includes .NET Runtime	Includes .NET Desktop Runtime	Includes ASP.NET Core Runtime
.NET Runtime	Yes	No	No
.NET Desktop Runtime	Yes	Yes	No
ASP.NET Core Runtime	No	No	Yes

The following list provides details about each runtime:

- *Desktop Runtime*

Runs .NET WPF and Windows Forms desktop apps for Windows. Includes the .NET runtime.

- *ASP.NET Core Runtime*

Runs ASP.NET Core apps.

- *.NET Runtime*

This runtime is the simplest runtime and doesn't include any other runtime. Install both *ASP.NET Core Runtime* and *Desktop Runtime* for the best compatibility with .NET apps.

[Download .NET Runtime](#)

SDK information

The SDK is used to build and publish .NET apps and libraries. Installing the SDK includes all three [runtimes](#): ASP.NET Core, Desktop, and .NET.

[Download .NET SDK](#)

Arm-based Windows PCs

The following sections describe things you should consider when installing .NET on an Arm-based Windows PC.

What is supported

The following table describes which versions of .NET are supported on an Arm-based Windows PC:

[\[+\] Expand table](#)

.NET Version	Architecture	SDK	Runtime	Path conflict
8	Arm64	Yes	Yes	No
8	x64	Yes	Yes	No
7	Arm64	Yes	Yes	No
7	x64	Yes	Yes	No

.NET Version	Architecture	SDK	Runtime	Path conflict
6	Arm64	Yes	Yes	No
6	x64	Yes	Yes	No
5	Arm64	Yes	Yes	Yes
5	x64	No	Yes	Yes

The x64 and Arm64 versions of the .NET SDK exist independently from each other. If a new version is released, each architecture install needs to be upgraded.

Path differences

On an Arm-based Windows PC, all Arm64 versions of .NET are installed to the normal `C:\Program Files\dotnet\` folder. However, the **x64** version of the .NET SDK is installed to the `C:\Program Files\dotnet\x64\` folder.

Path conflicts

The **x64** .NET SDK installs to its own directory, as described in the previous section. This allows the Arm64 and x64 versions of the .NET SDK to exist on the same machine.

However, any **x64** SDK prior to 6 isn't supported and installs to the same location as the Arm64 version, the `C:\Program Files\dotnet\` folder. If you want to install an unsupported x64 SDK, you must uninstall the Arm64 version first. The opposite is also true, you must uninstall the unsupported x64 SDK to install the Arm64 version.

Path variables

Environment variables that add .NET to system path, such as the `PATH` variable, may need to be changed if you have both the x64 and Arm64 versions of the .NET SDK installed. Additionally, some tools rely on the `DOTNET_ROOT` environment variable, which would also need to be updated to point to the appropriate .NET SDK installation folder.

Dependencies

.NET 8

The following Windows versions are supported with .NET 8:

① Note

A + symbol represents the minimum version.

[+] [Expand table](#)

OS	Version	Architectures
Windows 11	22000+	x64, x86, Arm64
Windows 10 Client	1607+	x64, x86, Arm64
Windows Server	2012+	x64, x86
Windows Server Core	2012+	x64, x86
Nano Server	1809+	x64

For more information about .NET 8 supported operating systems, distributions, and lifecycle policy, see [.NET 8 Supported OS Versions](#).

Windows 7 / 8.1 / Server 2012

More dependencies are required if you're installing the .NET SDK or runtime on the following Windows versions:

[+] [Expand table](#)

Operating System	Prerequisites
Windows 7 SP1 ESU	<ul style="list-style-type: none">- Microsoft Visual C++ 2015-2019 Redistributable 64-bit / 32-bit- KB3063858 64-bit / 32-bit- Microsoft Root Certificate Authority 2011 (.NET Core 2.1 offline installer only)
Windows 8.1	Microsoft Visual C++ 2015-2019 Redistributable 64-bit / 32-bit
Windows Server 2012	Microsoft Visual C++ 2015-2019 Redistributable 64-bit / 32-bit
Windows Server 2012 R2	Microsoft Visual C++ 2015-2019 Redistributable 64-bit / 32-bit

The previous requirements are also required if you receive an error related to either of the following dlls:

- *api-ms-win-crt-runtime-l1-1-0.dll*
- *api-ms-win-cor-timezone-l1-1-0.dll*
- *hostfxr.dll*

Docker

Containers provide a lightweight way to isolate your application from the rest of the host system. Containers on the same machine share just the kernel and use resources given to your application.

.NET can run in a Docker container. Official .NET Docker images are published to the Microsoft Container Registry (MCR) and are discoverable at the [Microsoft .NET 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.

For more information about using .NET in a Docker container, see [Introduction to .NET and Docker](#) and [Samples](#).

Troubleshooting

After installing the .NET SDK, you may run into problems trying to run .NET CLI commands. This section collects those common problems and provides solutions.

- [No .NET SDK was found](#)
- [Building apps is slower than expected](#)

No .NET SDK was found

Most likely you installed both the x86 (32-bit) and x64 (64-bit) versions of the .NET SDK. This is causing a conflict because when you run the `dotnet` command it's resolving to the x86 version when it should resolve to the x64 version. This is usually fixed by adjusting the `%PATH%` variable to resolve the x64 version first.

1. Verify that you have both versions installed by running the `where.exe dotnet` command. If you do, you should see an entry for both the *Program Files*\ and *Program Files (x86)*\ folders. If the *Program Files (x86)*\ folder is first, as

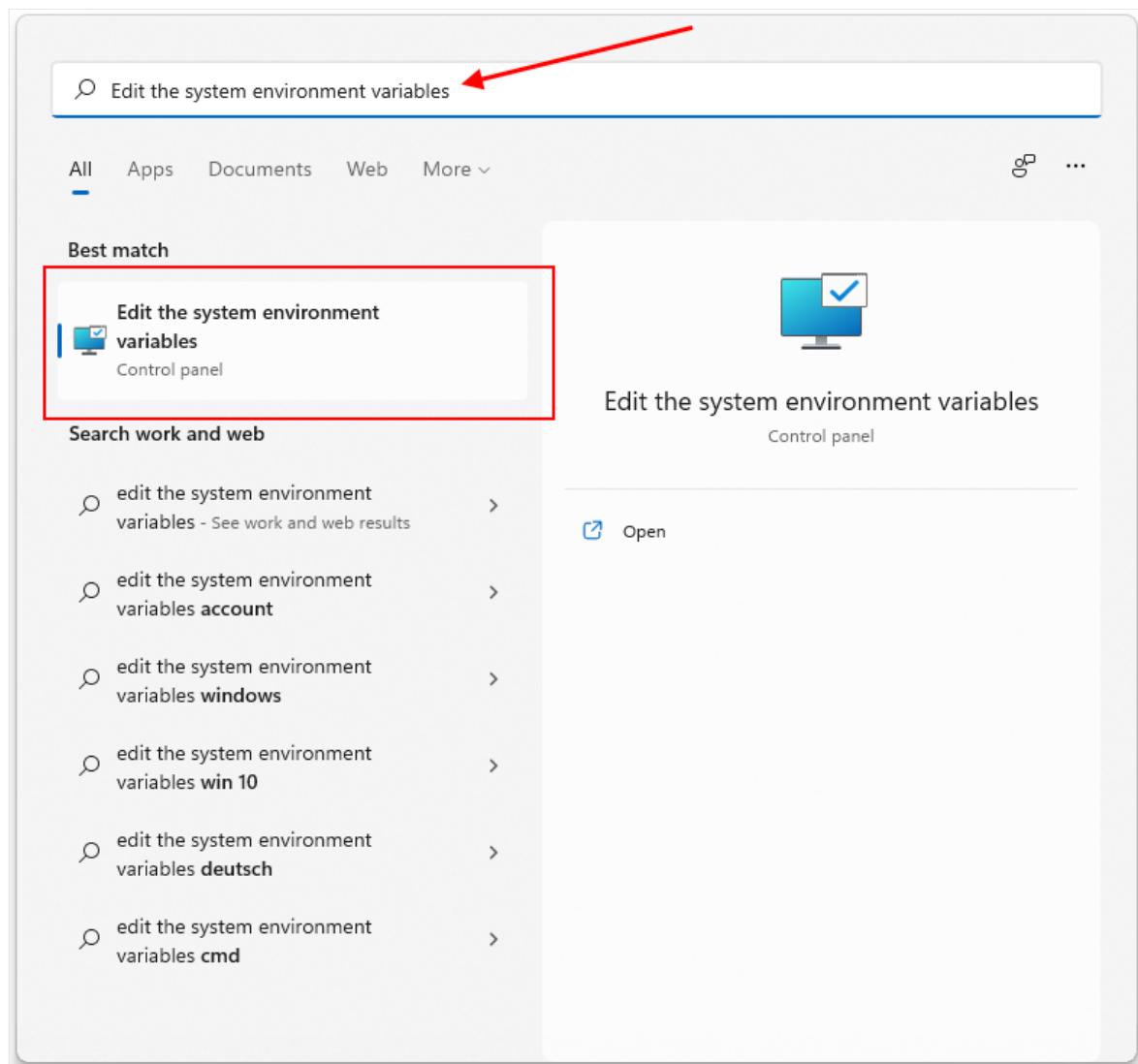
demonstrated by the following example, it's incorrect and you should continue on to the next step.

```
Windows Command Prompt

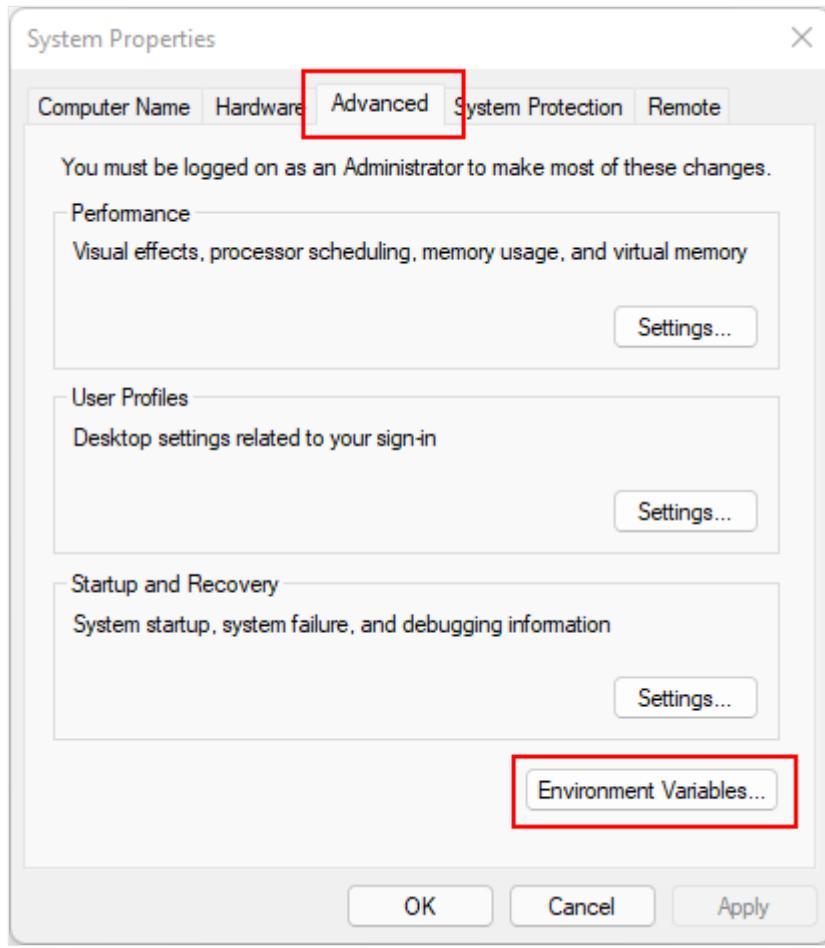
> where.exe dotnet
C:\Program Files (x86)\dotnet\dotnet.exe
C:\Program Files\dotnet\dotnet.exe
```

If it's correct and the *Program Files* is first, you don't have the problem this section is discussing and you should create a [.NET help request issue on GitHub](#) ↗

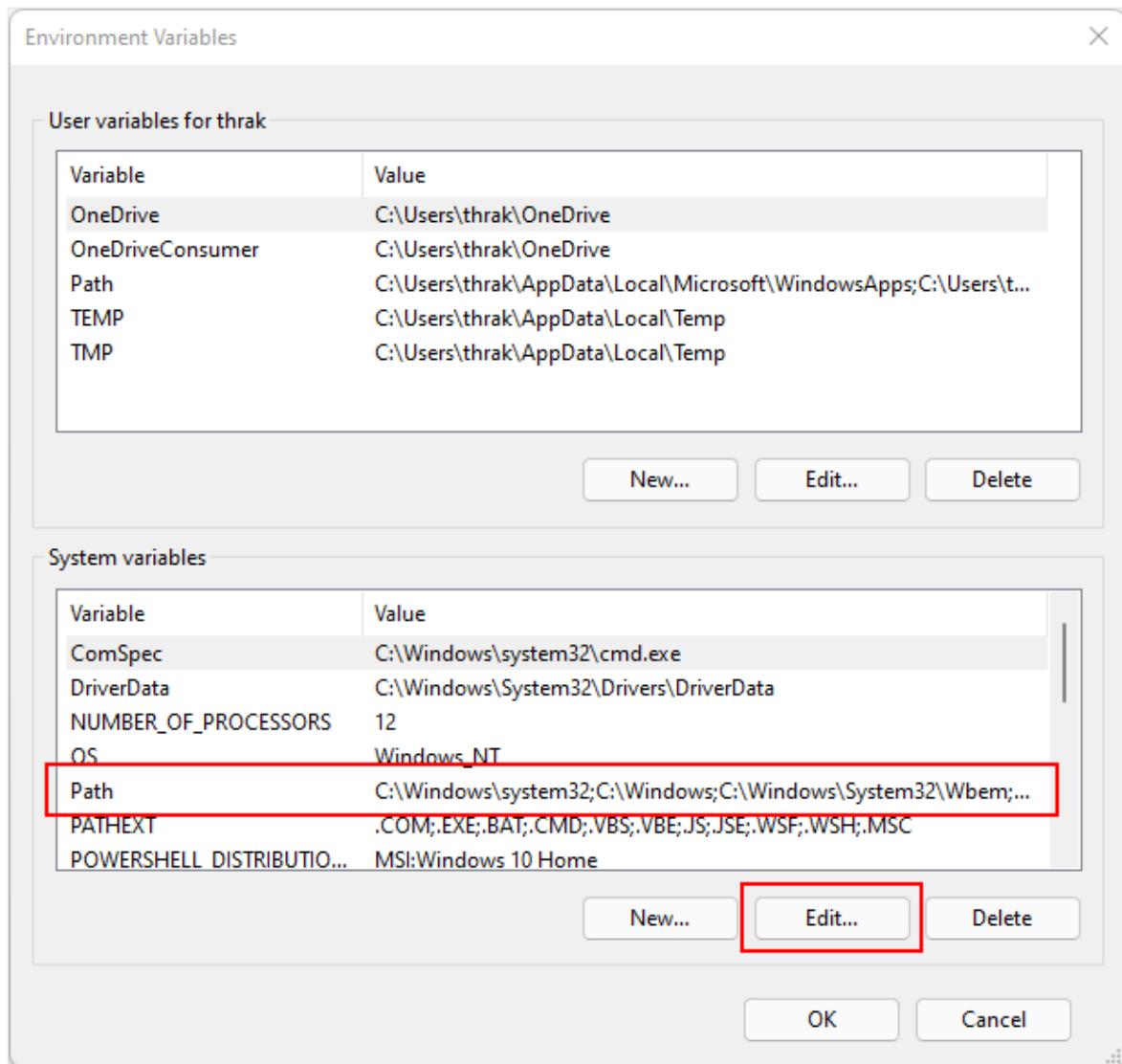
2. Press the Windows button and type "Edit the system environment variables" into search. Select **Edit the system environment variables**.



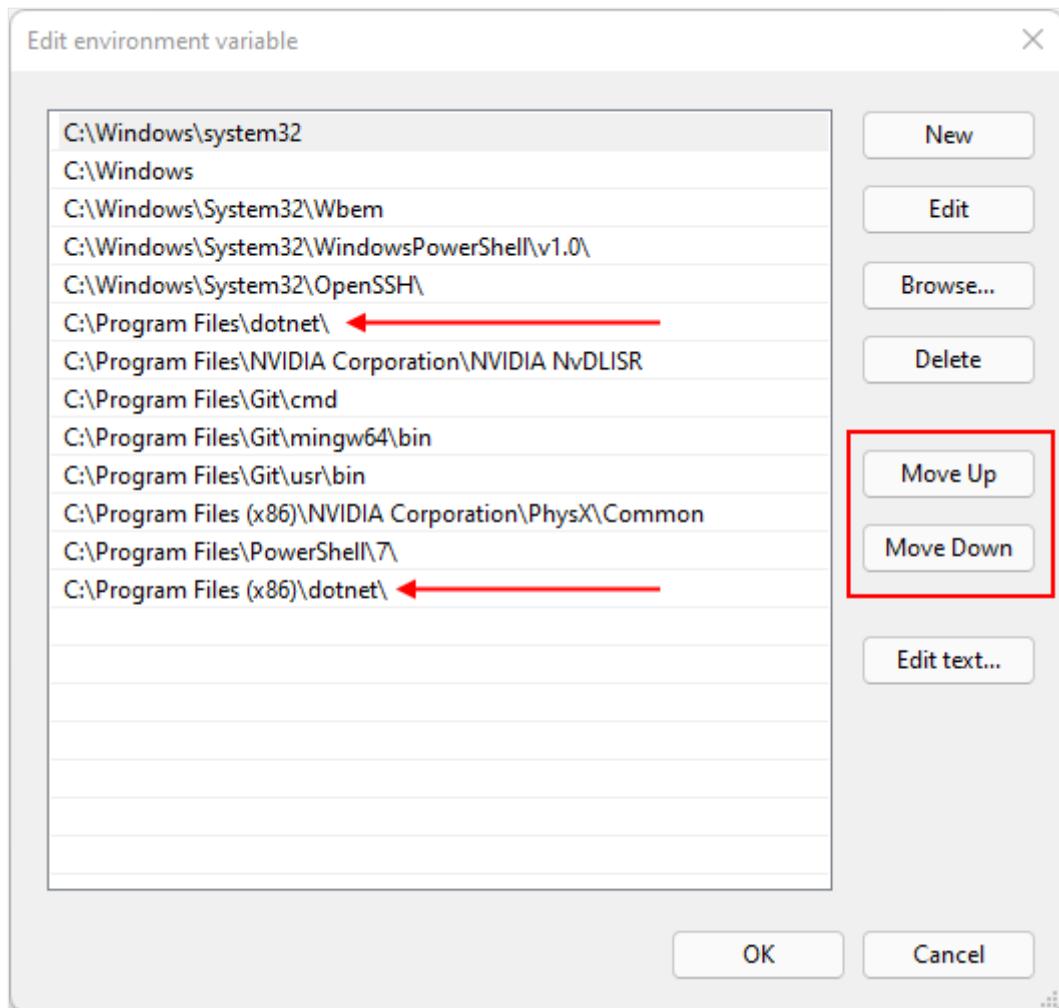
3. The **System Properties** window opens up to the **Advanced Tab**. Select **Environment Variables**.



4. On the **Environment Variables** window, under the **System variables** group, select the *Path** row and then select the **Edit** button.



5. Use the Move Up and Move Down buttons to move the **C:\Program Files\dotnet** entry above **C:\Program Files (x86)\dotnet**.



Building apps is slower than expected

Ensure that Smart App Control, a Windows feature, is off. Smart App Control isn't recommended to be enabled on machines used for development. Any setting other than "off" might negatively impact SDK performance.

Next steps

- [How to check if .NET is already installed.](#)
- [Tutorial: Hello World tutorial.](#)
- [Tutorial: Create a new app with Visual Studio Code.](#)
- [Tutorial: Containerize a .NET Core app.](#)

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you
can also create and review

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install .NET on macOS

Article • 12/30/2023

In this article, you learn how to install .NET on macOS. .NET is made up of the runtime and the SDK. The runtime is used to run a .NET app and might or might not be included with the app. The SDK is used to create .NET apps and libraries. The .NET runtime is always installed with the SDK.

The latest version of .NET is 8.

[Download .NET](#)

Supported releases

There are two types of supported releases: Long Term Support (LTS) releases and Standard Term Support (STS) releases. The quality of all releases is the same. The only difference is the length of support. LTS releases get free support and patches for three years. STS releases get free support and patches for 18 months. For more information, see [.NET Support Policy](#).

The following table is a list of currently supported .NET releases and the versions of macOS they're supported on:

[\[+\] Expand table](#)

Operating System	.NET 8 (LTS)	.NET 7 (STS)	.NET 6 (LTS)
macOS 14.0 "Sonoma"	✓ 8.0	✓ 7.0	✓ 6.0
macOS 13.0 "Ventura"	✓ 8.0	✓ 7.0	✓ 6.0
macOS 12.0 "Monterey"	✓ 8.0	✓ 7.0	✓ 6.0
macOS 11.0 "Big Sur"	✗	✓ 7.0	✓ 6.0
macOS 10.15 "Catalina"	✗	✓ 7.0	✓ 6.0

For a full list of .NET versions and their support life cycle, see [.NET Support Policy](#).

Unsupported releases

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Runtime information

The runtime is used to run apps created with .NET. When an app author publishes an app, they can include the runtime with their app. If they don't include the runtime, it's up to the user to install the runtime.

There are two different runtimes you can install on macOS:

- *ASP.NET Core runtime*
Runs ASP.NET Core apps. Includes the .NET runtime.
- *.NET runtime*
This runtime is the simplest runtime and doesn't include any other runtime. It's highly recommended that you install *ASP.NET Core runtime* for the best compatibility with .NET apps.

[Download .NET Runtime](#)

SDK information

The SDK is used to build and publish .NET apps and libraries. Installing the SDK includes both [runtimes](#): ASP.NET Core and .NET.

Notarization

Beginning with macOS Catalina (version 10.15), all software built after June 1, 2019 that's distributed with Developer ID must be notarized. This requirement applies to the .NET runtime, .NET SDK, and software created with .NET.

The runtime and SDK installers for .NET have been notarized since February 18, 2020. Prior released versions aren't notarized. If you run a non-notarized app, you'll see an error similar to the following image:



For more information about how enforced-notarization affects .NET (and your .NET apps), see [Working with macOS Catalina Notarization](#).

libgdiplus

.NET 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:

```
Console
```

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

Automated install

macOS has standalone installers that can be used to install .NET:

- [✓ .NET 8 downloads](#)
- [✓ .NET 7 downloads](#)
- [✓ .NET 6 downloads](#)

Manual install

As an alternative to the macOS installers for .NET, you can download and manually install the SDK and runtime. Manual installation is usually performed as part of continuous integration testing. For a developer or user, it's generally better to use an [installer](#).

Download a **binary** release for either the SDK or the runtime from one of the following sites. The .NET SDK includes the corresponding runtime:

- [.NET 8 downloads](#)
- [.NET 7 downloads](#)
- [.NET 6 downloads](#)
- [All .NET downloads](#)

Extract the downloaded file and use the `export` command to set `DOTNET_ROOT` to the extracted folder's location and then ensure .NET is in PATH. Exporting `DOTNET_ROOT` makes the .NET CLI commands available in the terminal. For more information about .NET environment variables, see [.NET SDK and CLI environment variables](#).

Different versions of .NET can be extracted to the same folder, which coexist side-by-side.

Example

The following commands use Bash to set the environment variable `DOTNET_ROOT` to the current working directory followed by `.dotnet`. That directory is created if it doesn't exist. The `DOTNET_FILE` environment variable is the filename of the .NET binary release you want to install. This file is extracted to the `DOTNET_ROOT` directory. Both the `DOTNET_ROOT` directory and its `tools` subdirectory are added to the `PATH` environment variable.

ⓘ Important

If you run these commands, remember to change the `DOTNET_FILE` value to the name of the .NET binary you downloaded.

Bash

```
DOTNET_FILE=dotnet-sdk-8.0.100-osx-x64.tar.gz
export DOTNET_ROOT=$(pwd)/.dotnet

mkdir -p "$DOTNET_ROOT" && tar zxf "$DOTNET_FILE" -C "$DOTNET_ROOT"
```

```
export PATH=$PATH:$DOTNET_ROOT
```

You can install more than one version of .NET in the same folder.

You can also install .NET to the home directory identified by the `HOME` variable or `~` path:

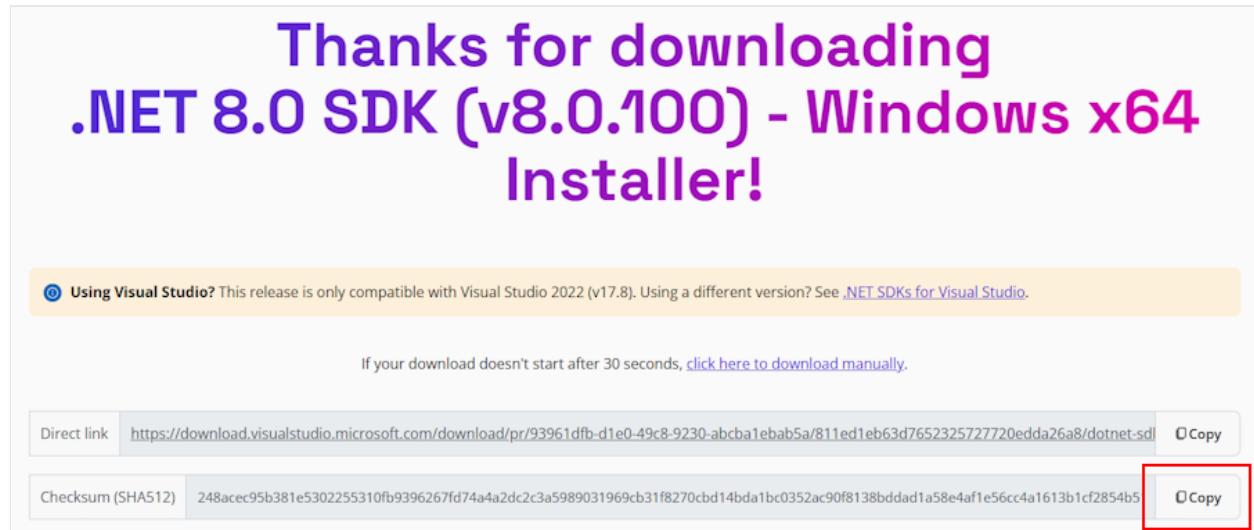
Bash

```
export DOTNET_ROOT=$HOME/.dotnet
```

Verify downloaded binaries

After downloading an installer, verify it to make sure that the file hasn't been changed or corrupted. You can verify the checksum on your computer and then compare it to what was reported on the download website.

When you download an installer or binary from an official download page, the checksum for the file is displayed. Select the **Copy** button to copy the checksum value to your clipboard.



Use the `sha512sum` command to print the checksum of the file you've downloaded. For example, the following command reports the checksum of the `dotnet-sdk-8.0.100-linux-x64.tar.gz` file:

Bash

```
$ sha512sum dotnet-sdk-8.0.100-linux-x64.tar.gz
13905ea20191e70baeba50b0e9bbe5f752a7c34587878ee104744f9fb453bfe439994d389697
22bdae7f60ee047d75dda8636f3ab62659450e9cd4024f38b2a5  dotnet-sdk-8.0.100-
linux-x64.tar.gz
```

Compare the checksum with the value provided by the download site.

ⓘ Important

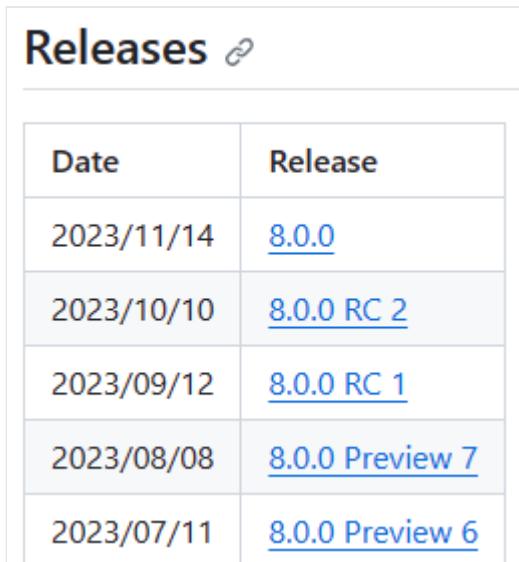
Even though a Linux file is shown in these examples, this information equally applies to macOS.

Use a checksum file to validate

The .NET release notes contain a link to a checksum file you can use to validate your downloaded file. The following steps describe how to download the checksum file and validate a .NET install binary:

1. The release notes page for .NET 8 on GitHub at

<https://github.com/dotnet/core/tree/main/release-notes/8.0> ↗ contains a section named **Releases**. The table in that section links to the downloads and checksum files for each .NET 8 release:



Date	Release
2023/11/14	8.0.0
2023/10/10	8.0.0 RC 2
2023/09/12	8.0.0 RC 1
2023/08/08	8.0.0 Preview 7
2023/07/11	8.0.0 Preview 6

2. Select the link for the version of .NET that you downloaded. The previous section used .NET SDK 8.0.100, which is in the .NET 8.0.0 release.
3. In the release page, you can see the .NET Runtime and .NET SDK version, and a link to the checksum file:

.NET 8.0.0 - November 14, 2023

The .NET 8.0.0 and .NET SDK 8.0.100 releases are available for download. The latest 8.0 release is always listed at [.NET 8.0 Releases](#).

Downloads

	SDK Installer ¹	SDK Binaries ¹	Runtime Installer	Runtime Binaries	ASP.NET Core Runtime	Windows Desktop Runtime
Windows	x86 x64 Arm64	x86 x64 Arm64	x86 x64 Arm64	x86 x64 Arm64	x86 x64 Hosting Bundle²	x86 x64 Arm64
macOS	x64 ARM64	x64 ARM64	x64 ARM64	x64 ARM64	x64 ARM64	-
Linux	Snap and Package Manager	x64 Arm Arm64 Arm32 Alpine x64 Alpine	Packages (x64)	x64 Arm Arm64 Arm32 Alpine Arm64 Alpine x64 Alpine	x64¹ Arm¹ Arm64¹ x64 Alpine	-
	Checksums	Checksums	Checksums	Checksums	Checksums	Checksums

4. Copy the link to the checksum file.

5. Use the following script, but replace the link to download the appropriate checksum file:

Bash

```
curl -O https://dotnetcli.blob.core.windows.net/dotnet/checksums/8.0.0-sha.txt
```

6. With both the checksum file and the .NET release file downloaded to the same directory, use the `sha512sum -c {file} --ignore-missing` command to validate the downloaded file.

When validation passes, you see the file printed with the **OK** status:

Bash

```
$ sha512sum -c 8.0.0-sha.txt --ignore-missing
dotnet-sdk-8.0.100-linux-x64.tar.gz: OK
```

If you see the file marked as **FAILED**, the file you downloaded isn't valid and shouldn't be used.

Bash

```
$ sha512sum -c 8.0.0-sha.txt --ignore-missing
dotnet-sdk-8.0.100-linux-x64.tar.gz: FAILED
sha512sum: WARNING: 1 computed checksum did NOT match
sha512sum: 8.0.0-sha.txt: no file was verified
```

Set environment variables system-wide

If you used the instructions in the [Manual install example](#) section, the variables set only apply to your current terminal session. Add them to your shell profile. There are many different shells available for macOS and each has a different profile. For example:

- **Bash Shell:** `~/.profile`, `/etc/profile`
- **Korn Shell:** `~/.kshrc` or `.profile`
- **Z Shell:** `~/.zshrc` or `.zprofile`

Set the following two environment variables in your shell profile:

- `DOTNET_ROOT`

This variable is set to the folder .NET was installed to, such as `$HOME/.dotnet`:

```
Bash
```

```
export DOTNET_ROOT=$HOME/.dotnet
```

- `PATH`

This variable should include both the `DOTNET_ROOT` folder and the `DOTNET_ROOT/tools` folder:

```
Bash
```

```
export PATH=$PATH:$DOTNET_ROOT:$DOTNET_ROOT/tools
```

Arm-based Macs

The following sections describe things you should consider when installing .NET on an Arm-based Mac.

What's supported

The following table describes which versions of .NET are supported on an Arm-based Mac:

 [Expand table](#)

.NET Version	Architecture	SDK	Runtime	Path conflict
8	Arm64	Yes	Yes	No
8	x64	Yes	Yes	No
7	Arm64	Yes	Yes	No
7	x64	Yes	Yes	No
6	Arm64	Yes	Yes	No
6	x64	Yes	Yes	No

Starting with .NET 6, the x64 and Arm64 versions of the .NET SDK exist independently from each other. If a new version is released, each install needs to be upgraded.

Path differences

On an Arm-based Mac, all Arm64 versions of .NET are installed to the normal `/usr/local/share/dotnet/` folder. However, when you install the **x64** version of .NET SDK, it's installed to the `/usr/local/share/dotnet/x64/dotnet/` folder.

Path conflicts

Starting with .NET 6, the **x64** .NET SDK installs to its own directory, as described in the previous section. This allows the Arm64 and x64 versions of the .NET SDK to exist on the same machine. However, any **x64** SDK prior to .NET 6 isn't supported and installs to the same location as the Arm64 version, the `/usr/local/share/dotnet/` folder. If you want to install an unsupported x64 SDK, you need to first uninstall the Arm64 version. The opposite is also true, you need to uninstall the unsupported x64 SDK to install the Arm64 version.

Path variables

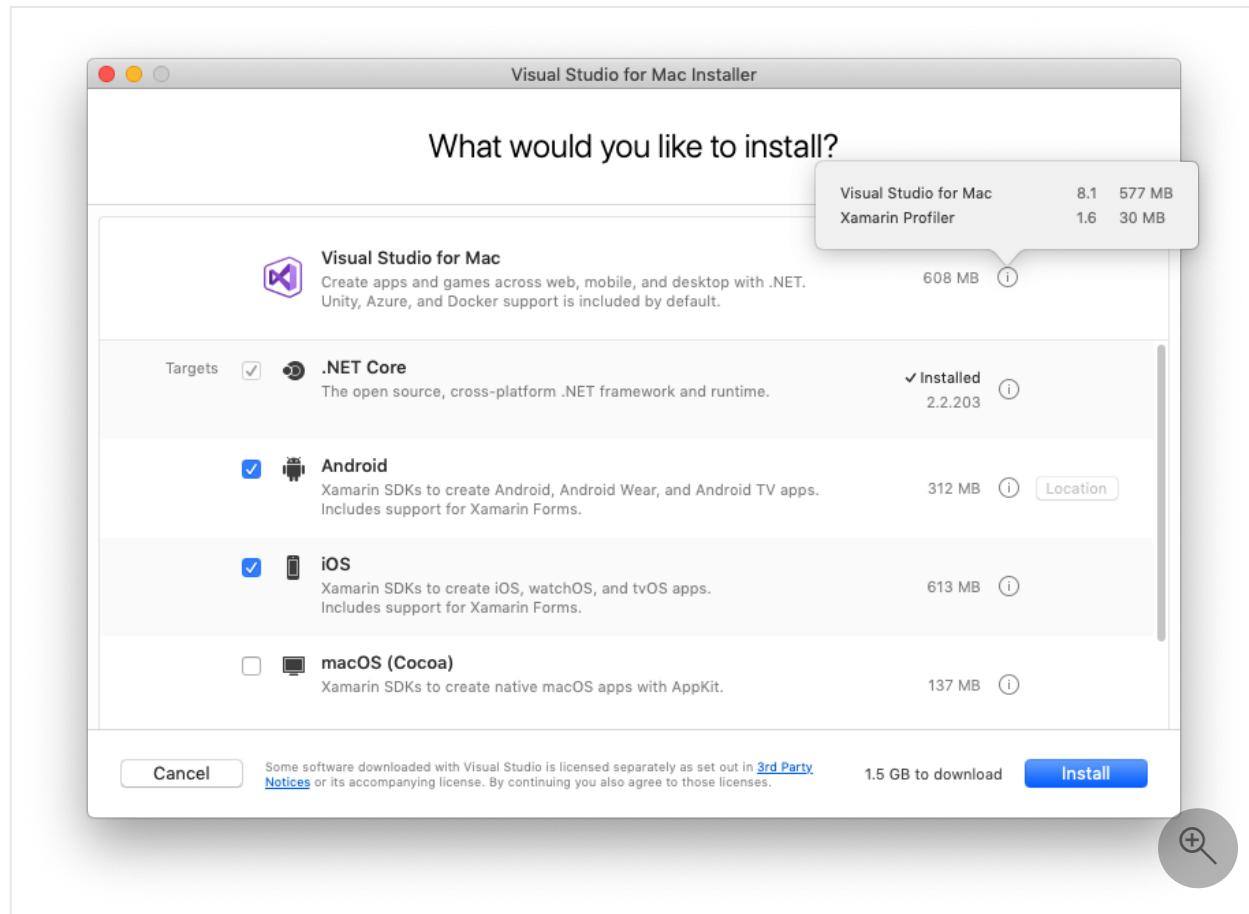
Environment variables that add .NET to system path, such as the `PATH` variable, might need to be changed if you have both the x64 and Arm64 versions of the .NET 6 SDK installed. Additionally, some tools rely on the `DOTNET_ROOT` environment variable, which would also need to be updated to point to the appropriate .NET 6 SDK installation folder.

Install with Visual Studio for Mac

Visual Studio for Mac installs the .NET SDK when the .NET workload is selected. To get started with .NET development on macOS, see [Install Visual Studio 2019 for Mac](#).

[+] [Expand table](#)

.NET SDK version	Visual Studio version
8.0	Visual Studio 2022 for Mac 17.8 or higher.
7.0	Visual Studio 2022 for Mac 17.4 or higher.
6.0	Visual Studio 2022 for Mac Preview 3 17.0 or higher.



ⓘ Important

Microsoft has announced the retirement of Visual Studio for Mac. Visual Studio for Mac will no longer be supported starting August 31, 2024. Alternatives include:

- Visual Studio Code with the [C# Dev Kit](#) and related extensions, such as [.NET MAUI](#) and [Unity](#).
- Visual Studio running on Windows in a VM on Mac.
- Visual Studio running on Windows in a [VM in the Cloud](#).

For more information, see [Visual Studio for Mac retirement announcement](#).

Install alongside Visual Studio Code

Visual Studio Code is a powerful and lightweight source code editor that runs on your desktop. Visual Studio Code is available for Windows, macOS, and Linux.

While Visual Studio Code doesn't come with an automated .NET installer like Visual Studio does, adding .NET support is simple.

1. [Download and install Visual Studio Code](#).
2. [Download and install the .NET SDK](#).
3. [Install the C# extension from the Visual Studio Code marketplace](#).

Install with bash automation

The [dotnet-install scripts](#) are used for automation and non-admin installs of the runtime. You can download the script from the [dotnet-install script reference page](#).

The script defaults to installing the latest [long term support \(LTS\)](#) version, which is .NET 8. You can choose a specific release by specifying the `channel` switch. Include the `runtime` switch to install a runtime. Otherwise, the script installs the SDK.

The following command installs the ASP.NET Core runtime for maximum compatibility. The ASP.NET Core runtime also includes the standard .NET runtime.

Bash

```
./dotnet-install.sh --channel 8.0 --runtime aspnetcore
```

Docker

Containers provide a lightweight way to isolate your application from the rest of the host system. Containers on the same machine share just the kernel and use resources given to your application.

.NET can run in a Docker container. Official .NET Docker images are published to the Microsoft Container Registry (MCR) and are discoverable at the [Microsoft .NET 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.

For more information about using .NET in a Docker container, see [Introduction to .NET and Docker](#) and [Samples](#).

Next steps

- [How to check if .NET is already installed.](#)
- [Working with macOS Catalina notarization.](#)
- [Tutorial: Get started on macOS.](#)
- [Tutorial: Create a new app with Visual Studio Code.](#)
- [Tutorial: Containerize a .NET app.](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install .NET on Linux

Article • 12/15/2023

This article details how to install .NET on various Linux distributions either manually, via a package manager, or via a [container](#).

Manual installation

You can install .NET manually in the following ways:

- [Manual install](#)
- [Scripted install](#)

You may need to install [.NET dependencies](#) if you install .NET manually.

Packages

.NET is available in [official package archives](#) for various Linux distributions and [packages.microsoft.com](#).

- [Alpine](#)
- [CentOS](#)
- [Debian](#)
- [Fedora](#)
- [openSUSE](#)
- [SLES](#)
- [Ubuntu](#)

.NET is [supported by Microsoft](#) when downloaded from a Microsoft source. Best effort support is offered from Microsoft when downloaded from elsewhere. You can open issues at [dotnet/core](#) if you run into problems.

Next steps

- [How to check if .NET is already installed.](#)
- [Tutorial: Create a new app with Visual Studio Code.](#)
- [Tutorial: Containerize a .NET app.](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install the .NET SDK or the .NET Runtime on Ubuntu

Article • 01/24/2024

This article describes how to install .NET on Ubuntu. The Microsoft package repository contains every version of .NET that is currently, or was previously, supported on Ubuntu. Starting with Ubuntu 22.04, some versions of .NET are available in the Ubuntu package feed. For more information about available versions, see the [Supported distributions](#) section.

⚠ Warning

It's recommended that you choose a single repository to source .NET packages. Don't mix .NET packages from multiple package repositories, as this leads to problems when apps try to resolve a specific version of .NET.

[] [Expand table](#)

Method	Pros	Cons
Package manager (Microsoft feed)	<ul style="list-style-type: none">Supported versions always available.Patches are available right way.Dependencies are included.Easy removal.	<ul style="list-style-type: none">Requires registering the Microsoft package repository.Preview releases aren't available.Only supports x64 Ubuntu.
Package manager (Ubuntu feed)	<ul style="list-style-type: none">Usually the latest version is available.Patches are available right way.Dependencies are included.Easy removal.	<ul style="list-style-type: none">.NET versions available vary by Ubuntu version.Preview releases aren't available.Only supports x64 Ubuntu. (Except for Ubuntu 23.04+, which also supports Arm64)
Script \ Manual extraction	<ul style="list-style-type: none">Control where .NET is installed.Preview releases are available.	<ul style="list-style-type: none">Manually install updates.Manually install dependencies.Manual removal.

Decide how to install .NET

When your version of Ubuntu supports .NET through the built-in Ubuntu feed, support for those builds of .NET is provided by Canonical and the builds might be optimized for different workloads. Microsoft provides support for packages in the Microsoft package repository feed.

Use the following sections to determine how you should install .NET:

- [I'm using Ubuntu 22.04 or later, and I only need .NET](#)
- [I'm using a version of Ubuntu prior to 22.04](#)
- [I'm using other Microsoft packages, such as powershell, mdatp, or mssql](#)
- [I want to create a .NET app](#)
- [I want to run a .NET app in a container, cloud, or continuous-integration scenario](#)
- [My Ubuntu distribution doesn't include the .NET version I want, or I need an out-of-support .NET version](#)
- [I want to install a preview version](#)
- [I don't want to use APT](#)
- [I'm using an Arm-based CPU](#)

I'm using Ubuntu 22.04 or later, and I only need .NET

Install .NET through the Ubuntu feed. For more information, see the following pages:

- [Install .NET on Ubuntu 22.04.](#)
- [Install .NET on Ubuntu 22.10.](#)
- [Install .NET on Ubuntu 23.04.](#)
- [Install .NET on Ubuntu 23.10.](#)

Important

.NET SDK versions offered by Canonical are always in the [.1xx feature band](#). If you want to use a newer feature band release, use the [Microsoft feed to install the SDK](#). Make sure you review the information in the [.NET package mix ups on Linux](#) article to understand the implications of switching between repository feeds.

If you're going to install the Microsoft repository to use other Microsoft packages, such as `powershell`, `mdatp`, or `mssql`, you need to deprioritize the .NET packages provided by the Microsoft repository. For instructions on how to deprioritize the packages, see [My Linux distribution provides .NET packages, and I want to use them](#).

I'm using a version of Ubuntu prior to 22.04

Use the instructions on the version-specific Ubuntu page.

- [20.04 \(LTS\)](#)
- [18.04 \(LTS\)](#)
- [16.04 \(LTS\)](#)

Review the [Supported distributions](#) section for more information about what versions of .NET are supported for your version of Ubuntu. If you're installing a version that isn't supported, see [Register the Microsoft package repository](#).

I'm using other Microsoft packages, such as `powershell`, `mdatp`, or `mssql`

If your Ubuntu version supports .NET through the built-in Ubuntu feed, you must decide which feed should install .NET. The [Supported distributions](#) section provides a table that lists which versions of .NET are available the package feeds.

If you want to source the .NET packages from the Ubuntu feed, you need to deprioritize the .NET packages provided by the Microsoft repository. For instructions on how to deprioritize the packages, see [My Linux distribution provides .NET packages, and I want to use them](#).

I want to create a .NET app

Use the same package sources for the SDK as you use for the runtime. For example, if you're using Ubuntu 22.04 and .NET 6, but not .NET 7, it's recommended that you install .NET through the built-in Ubuntu feed. If, however, you move to .NET 7, which isn't provided by Canonical for Ubuntu 22.04, you should uninstall .NET and reinstall it with the [Microsoft package repository](#). For more information, see [Register and install with the Microsoft package repository](#). Also, review the other suggestions in the [Decide how to install .NET](#) section.

I want to run a .NET app in a container, cloud, or continuous-integration scenario

If your Ubuntu version provides the .NET version you require, install it from the built-in feed. Otherwise, [register the Microsoft package repository](#) and install .NET from that repository. Review the information in the [Supported distributions](#) section.

If the version of .NET you want isn't available, try using the [dotnet-install](#) script.

My Ubuntu distribution doesn't include the .NET version I want, or I need an out-of-support .NET version

We recommend you use APT and the Microsoft package repository. For more information, see the [Register and install with the Microsoft package repository](#) section.

I want to install a preview version

Use one of the following ways to install .NET:

- [Install .NET with install-dotnet script.](#)
- [Manually install .NET](#)

I don't want to use APT

If you want an automated installation, use the [Linux installation script](#).

If you want full control over the .NET installation experience, download a tarball and manually install .NET. For more information, see [Manual install](#).

I'm using an Arm-based CPU

Use one of the following ways to install .NET:

- [Install .NET with install-dotnet script.](#)
- [Manually install .NET](#)

Supported distributions

The following table is a list of currently supported .NET releases and the versions of Ubuntu they're supported on. Each link goes to the specific Ubuntu version page with specific instructions on how to install .NET for that version of Ubuntu.

↔ [Expand table](#)

Ubuntu	Supported .NET versions	Available in Ubuntu feed	Available in Microsoft feed
23.10	8.0, 7.0, 6.0	8.0, 7.0, 6.0	8.0, 7.0, 6.0

Ubuntu	Supported .NET versions	Available in Ubuntu feed	Available in Microsoft feed
23.04	8.0, 7.0, 6.0	7.0, 6.0	8.0, 7.0, 6.0
22.10	7.0, 6.0	7.0, 6.0	7.0, 6.0, 3.1
22.04 (LTS)	8.0, 7.0, 6.0	7.0, 6.0	8.0, 7.0, 6.0, 3.1
20.04 (LTS)	8.0, 7.0, 6.0	None	8.0, 7.0, 6.0, 5.0, 3.1, 2.1
18.04 (LTS)	7.0, 6.0	None	7.0, 6.0, 5.0, 3.1, 2.2, 2.1
16.04 (LTS)	6.0	None	6.0, 5.0, 3.1, 3.0, 2.2, 2.1, 2.0

When an [Ubuntu version](#) falls out of support, .NET is no longer supported with that version.

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Register the Microsoft package repository

The Microsoft package repository contains all versions of .NET that were previously, or are currently, [supported with your version of Ubuntu](#). If your version of Ubuntu provides .NET packages, you'll need to deprioritize the Ubuntu packages and use the Microsoft repository. For instructions on how to deprioritize the packages, see [I need a version of .NET that isn't provided by my Linux distribution](#).

Important

Package manager installs are only supported on the **x64** architecture. Other architectures, such as **Arm**, must install .NET by some other means, such as with the [installer script](#) or by [manual installation](#).

Preview releases are **not** available in the Microsoft package repository. For more information, see [Install preview versions](#).

⊗ Caution

We recommend that you only use one repository to manage all of your .NET installs. If you've previously installed .NET with the Ubuntu repository, you must clean the system of .NET packages and configure the APT to ignore the Ubuntu feed. For more information about how to do this, see [I need a version of .NET that isn't provided by my Linux distribution](#).

Installing with APT can be done with a few commands. Before you install .NET, run the following commands to add the Microsoft package signing key to your list of trusted keys and add the package repository.

Open a terminal and run the following commands:

Bash

```
# Get Ubuntu version
declare repo_version=$(if command -v lsb_release &> /dev/null; then
  lsb_release -r -s; else grep -oP '(?=>^VERSION_ID=).+' /etc/os-release | tr
  -d '''; fi)

# Download Microsoft signing key and repository
wget https://packages.microsoft.com/config/ubuntu/$repo_version/packages-
microsoft-prod.deb -O packages-microsoft-prod.deb

# Install Microsoft signing key and repository
sudo dpkg -i packages-microsoft-prod.deb

# Clean up
rm packages-microsoft-prod.deb

# Update packages
sudo apt update
```

💡 Tip

The previous script was written for Ubuntu and it might not work if you're using a derived distribution, such as Linux Mint. It's likely that the `$repo_version` variable won't be assigned the correct value, making the URI for the `wget` command invalid. This variable maps to the specific Ubuntu version you want to get packages for, such as 22.10 or 23.04.

You can use a web browser and navigate to <https://packages.microsoft.com/config/ubuntu/> to see which versions of Ubuntu are available to use as the `$repo_version` value.

Install .NET

After you've [registered the Microsoft package repository](#), or if your version of Ubuntu's default feed supports the .NET package, you can install .NET through the package manager with the `sudo apt install <package-name>` command. Replace `<package-name>` with the name of the .NET package you want to install. For example, to install .NET SDK 8.0, use the command `sudo apt install dotnet-sdk-8.0`. The following table lists the currently supported .NET packages:

[] [Expand table](#)

Product	Type	Package
8.0	ASP.NET Core	aspnetcore-runtime-8.0
8.0	.NET	dotnet-runtime-8.0
8.0	.NET	dotnet-sdk-8.0
7.0	ASP.NET Core	aspnetcore-runtime-7.0
7.0	.NET	dotnet-runtime-7.0
7.0	.NET	dotnet-sdk-7.0
6.0	ASP.NET Core	aspnetcore-runtime-6.0
6.0	.NET	dotnet-runtime-6.0
6.0	.NET	dotnet-sdk-6.0

If you want to install an unsupported version of .NET, check the [Supported distributions](#) section to see if that version of .NET is available. Then, substitute the [version](#) of .NET you want to install. For example, to install ASP.NET Core 2.1, use the package name `aspnetcore-runtime-2.1`.

? **Tip**

If you're not creating .NET apps, install the ASP.NET Core runtime as it includes the .NET runtime and also supports ASP.NET Core apps.

Some environment variables affect how .NET is run after it's installed. For more information, see [.NET SDK and CLI environment variables](#).

Uninstall .NET

If you installed .NET through a package manager, uninstall in the same way with the `apt-get remove` command:

Bash

```
sudo apt-get remove dotnet-sdk-6.0
```

For more information, see [Uninstall .NET](#).

Install preview versions

Preview and release candidate versions of .NET aren't available in package repositories. You can install previews and release candidates of .NET in one of the following ways:

- [Scripted install with *install-dotnet.sh*](#)
- [Manual binary extraction](#)

Remove preview versions

When using a package manager to manage your installation of .NET, you may run into a conflict if you've previously installed a preview release. The package manager may interpret the non-preview release as an earlier version of .NET. To install the non-preview release, first uninstall the preview versions. For more information about uninstalling .NET, see [How to remove the .NET Runtime and SDK](#).

Use APT to update .NET

If you installed .NET through a package manager, you can upgrade the package with the `apt upgrade` command. For example, the following commands upgrade the `dotnet-sdk-7.0` package with the latest version:

Bash

```
sudo apt update
sudo apt upgrade dotnet-sdk-7.0
```

💡 Tip

If you've upgraded your Linux distribution since installing .NET, you may need to reconfigure the Microsoft package repository. Run the installation instructions for your current distribution version to upgrade to the appropriate package repository for .NET updates.

Troubleshooting

Starting with Ubuntu 22.04, you might run into a situation where it seems only a piece of .NET is available. For example, you've installed the runtime and the SDK, but when you run `dotnet --info` only the runtime is listed. This situation can be related to using two different package sources. The built-in Ubuntu 22.04 and Ubuntu 22.10 package feeds include some versions of .NET, but not all, and you might have also installed .NET from the Microsoft feeds. For more information about how to fix this problem, see [Troubleshoot .NET errors related to missing files on Linux](#).

APT problems

This section provides information on common errors you might get while using APT to install .NET.

Unable to find package

ⓘ Important

Using a package manager to install .NET from the **Microsoft package feed** only supports the **x64** architecture. Other architectures, such as **Arm**, aren't supported by the **Microsoft package feed**.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Use the install-dotnet script to install .NET.](#)
- [Manually install .NET.](#)

Unable to locate \ Some packages could not be installed

Note

This information only applies when .NET is installed from the Microsoft package feed.

If you receive an error message similar to **Unable to locate package {dotnet-package}** or **Some packages could not be installed**, run the following commands.

There are two placeholders in the following set of commands.

- `{dotnet-package}`

This represents the .NET package you're installing, such as `aspnetcore-runtime-8.0`. This is used in the following `sudo apt-get install` command.

- `{os-version}`

This represents the distribution version you're on. This is used in the `wget` command below. The distribution version is the numerical value, such as `20.04` on Ubuntu or `10` on Debian.

First, try purging the package list:

Bash

```
sudo dpkg --purge packages-microsoft-prod && sudo dpkg -i packages-microsoft-prod.deb
sudo apt-get update
```

Then, try to install .NET again. If that doesn't work, you can run a manual install with the following commands:

Bash

```
sudo apt-get install -y gpg
wget -O - https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor
-o microsoft.asc.gpg
sudo mv microsoft.asc.gpg /etc/apt/trusted.gpg.d/
wget https://packages.microsoft.com/config/ubuntu/{os-version}/prod.list
sudo mv prod.list /etc/apt/sources.list.d/microsoft-prod.list
sudo chown root:root /etc/apt/trusted.gpg.d/microsoft.asc.gpg
sudo chown root:root /etc/apt/sources.list.d/microsoft-prod.list
sudo apt-get update && \
    sudo apt-get install -y {dotnet-package}
```

Failed to fetch

While installing the .NET package, you may see an error similar to `Failed to fetch ... File has unexpected size ... Mirror sync in progress?`. This error could mean that the package feed for .NET is being upgraded with newer package versions, and that you should try again later. During an upgrade, the package feed shouldn't be unavailable for more than 30 minutes. If you continually receive this error for more than 30 minutes, please file an issue at <https://github.com/dotnet/core/issues>.

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you must install these dependencies to run your app:

- libc6
- libgcc1 (for 16.x and 18.x)
- libgcc-s1 (for 20.x or later)
- libgssapi-krb5-2
- libicu55 (for 16.x)
- libicu60 (for 18.x)
- libicu66 (for 20.x)
- libicu70 (for 22.04)
- libicu71 (for 22.10)
- libicu72 (for 23.04)
- liblttng-ust1 (for 22.x)
- libssl1.0.0 (for 16.x)
- libssl1.1 (for 18.x, 20.x)
- libssl3 (for 22.x)
- libstdc++6
- libunwind8 (for 22.x)
- zlib1g

Dependencies can be installed with the `apt install` command. The following snippet demonstrates installing the `zlib1g` library:

Bash

```
sudo apt install zlib1g
```

If the .NET app uses the `System.Drawing.Common` assembly, libgdiplus will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of *libgdiplus* by [adding the Mono repository to your system](#).

Next steps

- [How to enable Tab completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

Install .NET SDK or .NET Runtime on Ubuntu 23.10

Article • 12/30/2023

This article discusses how to install .NET on Ubuntu 23.10; .NET 8, .NET 7, and .NET 6, are supported.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

Important

Using a package manager to install .NET from the **Microsoft pacakge feed** only supports the **x64** architecture. Other architectures, such as **Arm**, aren't supported by the **Microsoft package feed**. Use the Ubuntu feed or manually install .NET. Be cautious of package mix up problems. For more information, see [.NET package mix ups on Linux](#).

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Use the install-dotnet script to install .NET](#).
- [Manually install .NET](#).

.NET is available in the Ubuntu package manager feeds, as well as the Microsoft package repository. However, you should only use one or the other to install .NET. If you want to use the Microsoft package repository, see [How to register the Microsoft package repository](#).

Supported versions

The following versions of .NET are supported or available for Ubuntu 23.10:

Supported .NET versions	Available in Ubuntu feed	Available in Microsoft feed
8.0, 7.0, 6.0	8.0, 7.0, 6.0	8.0, 7.0, 6.0

ⓘ Important

.NET SDK versions offered by Canonical are always in the **.1xx feature band**. If you want to use a newer feature band release, use the [Microsoft feed to install the SDK](#). Make sure you review the information in the [.NET package mix ups on Linux](#) article to understand the implications of switching between repository feeds.

When an [Ubuntu version](#) falls out of support, .NET is no longer supported with that version.

The following versions of .NET are **✗** no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Install .NET

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following commands:

Bash

```
sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-8.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package dotnet-sdk-8.0**, see the [troubleshooting](#) section.

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following commands install the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following commands:

Bash

```
sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-8.0
```

Important

If you receive an error message similar to **Unable to locate package aspnetcore-runtime-8.0**, see the [troubleshooting](#) section.

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo apt-get install -y dotnet-runtime-8.0
```

How to install other versions

.NET package names are standardized across all Linux distributions. The following table lists the packages:

All versions of .NET are available for download at <https://dotnet.microsoft.com/download/dotnet>, but require [manual installation](#). You can try and use the package manager to install a different version of .NET. However, the requested version may not be available.

The packages added to package manager feeds are named in a hackable format, for example: `{product}-{type}-{version}`.

- **product**

The type of .NET product to install. Valid options are:

- dotnet
- aspnetcore

- **type**

Chooses the SDK or the runtime. Valid options are:

- sdk (only available for the **dotnet** product)
- runtime

- **version**

The version of the SDK or runtime to install. This article will always give the instructions for the latest supported version. Valid options are any released version, such as:

- 8.0
- 6.0
- 3.1
- 2.1

It's possible the SDK/runtime you're trying to download is not available for your Linux distribution. For a list of supported distributions, see [Install .NET on Linux](#).

Examples

- Install the ASP.NET Core 8.0 runtime: `aspnetcore-runtime-8.0`
- Install the .NET Core 2.1 runtime: `dotnet-runtime-2.1`
- Install the .NET 5 SDK: `dotnet-sdk-5.0`
- Install the .NET Core 3.1 SDK: `dotnet-sdk-3.1`

Package missing

If the package-version combination doesn't work, it's not available. For example, there isn't an ASP.NET Core SDK, the SDK components are included with the .NET SDK. The value `aspnetcore-sdk-8.0` is incorrect and should be `dotnet-sdk-8.0`. For a list of Linux distributions supported by .NET, see [.NET dependencies and requirements](#).

Troubleshooting

If you run into issues installing or even running .NET, see [Troubleshooting](#).

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- libc6
- libgcc-s1
- libgssapi-krb5-2
- libicu72
- liblttng-ust1
- libssl3
- libstdc++6
- libunwind8
- zlib1g

Dependencies can be installed with the `apt install` command. The following snippet demonstrates installing the `zlib1g` library:

Bash

```
sudo apt install zlib1g
```

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of `libgdiplus` by [adding the Mono repository to your system](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install .NET SDK or .NET Runtime on Ubuntu 23.04

Article • 12/30/2023

This article discusses how to install .NET on Ubuntu 23.04; .NET 8, .NET 7, and .NET 6, are supported.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

Important

Using a package manager to install .NET from the **Microsoft pacakge feed** only supports the **x64** architecture. Other architectures, such as **Arm**, aren't supported by the **Microsoft package feed**. Use the Ubuntu feed or manually install .NET. Be cautious of package mix up problems. For more information, see [.NET package mix ups on Linux](#).

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Use the install-dotnet script to install .NET](#).
- [Manually install .NET](#).

.NET is available in the Ubuntu package manager feeds, as well as the Microsoft package repository. However, you should only use one or the other to install .NET. If you want to use the Microsoft package repository, see [How to register the Microsoft package repository](#).

Supported versions

The following versions of .NET are supported or available for Ubuntu 23.04:

Supported .NET versions	Available in Ubuntu feed	Available in Microsoft feed
8.0, 7.0, 6.0	7.0, 6.0	8.0, 7.0, 6.0

ⓘ Important

.NET SDK versions offered by Canonical are always in the **.1xx feature band**. If you want to use a newer feature band release, use the [Microsoft feed to install the SDK](#). Make sure you review the information in the [.NET package mix ups on Linux](#) article to understand the implications of switching between repository feeds.

When an [Ubuntu version](#) falls out of support, .NET is no longer supported with that version.

The following versions of .NET are **✗** no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Install .NET

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following commands:

Bash

```
sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-8.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package dotnet-sdk-8.0**, see the [troubleshooting](#) section.

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following commands install the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following commands:

Bash

```
sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-8.0
```

Important

If you receive an error message similar to **Unable to locate package aspnetcore-runtime-8.0**, see the [troubleshooting](#) section.

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo apt-get install -y dotnet-runtime-8.0
```

How to install other versions

.NET package names are standardized across all Linux distributions. The following table lists the packages:

All versions of .NET are available for download at <https://dotnet.microsoft.com/download/dotnet>, but require [manual installation](#). You can try and use the package manager to install a different version of .NET. However, the requested version may not be available.

The packages added to package manager feeds are named in a hackable format, for example: `{product}-{type}-{version}`.

- **product**

The type of .NET product to install. Valid options are:

- dotnet
- aspnetcore

- **type**

Chooses the SDK or the runtime. Valid options are:

- sdk (only available for the **dotnet** product)
- runtime

- **version**

The version of the SDK or runtime to install. This article will always give the instructions for the latest supported version. Valid options are any released version, such as:

- 8.0
- 6.0
- 3.1
- 2.1

It's possible the SDK/runtime you're trying to download is not available for your Linux distribution. For a list of supported distributions, see [Install .NET on Linux](#).

Examples

- Install the ASP.NET Core 8.0 runtime: `aspnetcore-runtime-8.0`
- Install the .NET Core 2.1 runtime: `dotnet-runtime-2.1`
- Install the .NET 5 SDK: `dotnet-sdk-5.0`
- Install the .NET Core 3.1 SDK: `dotnet-sdk-3.1`

Package missing

If the package-version combination doesn't work, it's not available. For example, there isn't an ASP.NET Core SDK, the SDK components are included with the .NET SDK. The value `aspnetcore-sdk-8.0` is incorrect and should be `dotnet-sdk-8.0`. For a list of Linux distributions supported by .NET, see [.NET dependencies and requirements](#).

Troubleshooting

If you run into issues installing or even running .NET, see [Troubleshooting](#).

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- libc6
- libgcc-s1
- libgssapi-krb5-2
- libicu72
- liblttng-ust1
- libssl3
- libstdc++6
- libunwind8
- zlib1g

Dependencies can be installed with the `apt install` command. The following snippet demonstrates installing the `zlib1g` library:

Bash

```
sudo apt install zlib1g
```

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of `libgdiplus` by [adding the Mono repository to your system](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you

.NET

.NET feedback

.NET is an open source project.
Select a link to provide feedback:

can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install .NET SDK or .NET Runtime on Ubuntu 22.10

Article • 08/17/2023

This article discusses how to install .NET on Ubuntu 22.10; .NET 6 and .NET 7 are both supported.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

Important

Using a package manager to install .NET from the **Microsoft package feed** only supports the **x64** architecture. Other architectures, such as **Arm**, aren't supported by the **Microsoft package feed**.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Use the install-dotnet script to install .NET.](#)
- [Manually install .NET.](#)

.NET is available in the Ubuntu package manager feeds, as well as the Microsoft package repository. However, you should only use one or the other to install .NET. If you want to use the Microsoft package repository, see [How to register the Microsoft package repository](#).

Warning

Don't use both repositories to manage .NET. If you've previously installed .NET from the Ubuntu feed or the Microsoft feed, you'll run into issues using the other feed. .NET is installed to different locations and is resolved differently for both package feeds. It's recommended that you uninstall previously installed versions of .NET and

then install with the Microsoft package repository. For more information, see [How to register the Microsoft package repository](#).

Supported versions

The following versions of .NET are supported or available for Ubuntu 22.10:

[Expand table](#)

Supported .NET versions	Available in Ubuntu feed	Available in Microsoft feed
7.0, 6.0	7.0, 6.0	7.0, 6.0, 3.1

Important

.NET SDK versions offered by Canonical are always in the **.1xx feature band**. If you want to use a newer feature band release, use the [Microsoft feed to install the SDK](#). Make sure you review the information in the [.NET package mix ups on Linux](#) article to understand the implications of switching between repository feeds.

When an [Ubuntu version](#) falls out of support, .NET is no longer supported with that version.

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Install .NET

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following commands:

Bash

```
sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-7.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package dotnet-sdk-7.0**, see the [troubleshooting](#) section.

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following commands install the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following commands:

Bash

```
sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-7.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package aspnetcore-runtime-7.0**, see the [troubleshooting](#) section.

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-7.0` in the previous command with `dotnet-runtime-7.0`:

Bash

```
sudo apt-get install -y dotnet-runtime-7.0
```

How to install other versions

.NET package names are standardized across all Linux distributions. The following table lists the packages:

All versions of .NET are available for download at <https://dotnet.microsoft.com/download/dotnet>, but require [manual installation](#). You can try and use the package manager to install a different version of .NET. However, the requested version may not be available.

The packages added to package manager feeds are named in a hackable format, for example: `{product}-{type}-{version}`.

- **product**

The type of .NET product to install. Valid options are:

- dotnet
- aspnetcore

- **type**

Chooses the SDK or the runtime. Valid options are:

- sdk (only available for the **dotnet** product)
- runtime

- **version**

The version of the SDK or runtime to install. This article will always give the instructions for the latest supported version. Valid options are any released version, such as:

- 8.0
- 6.0
- 3.1
- 2.1

It's possible the SDK/runtime you're trying to download is not available for your Linux distribution. For a list of supported distributions, see [Install .NET on Linux](#).

Examples

- Install the ASP.NET Core 8.0 runtime: `aspnetcore-runtime-8.0`
- Install the .NET Core 2.1 runtime: `dotnet-runtime-2.1`
- Install the .NET 5 SDK: `dotnet-sdk-5.0`
- Install the .NET Core 3.1 SDK: `dotnet-sdk-3.1`

Package missing

If the package-version combination doesn't work, it's not available. For example, there isn't an ASP.NET Core SDK, the SDK components are included with the .NET SDK. The

value `aspnetcore-sdk-8.0` is incorrect and should be `dotnet-sdk-8.0`. For a list of Linux distributions supported by .NET, see [.NET dependencies and requirements](#).

Troubleshooting

If you run into issues installing or even running .NET, see [Troubleshooting](#).

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- `libc6`
- `libgcc-s1`
- `libgssapi-krb5-2`
- `libicu71`
- `liblttng-ust1`
- `libssl3`
- `libstdc++6`
- `libunwind8`
- `zlib1g`

Dependencies can be installed with the `apt install` command. The following snippet demonstrates installing the `zlib1g` library:

```
Bash
```

```
sudo apt install zlib1g
```

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because `System.Drawing.Common` is no longer supported on Linux, this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of `libgdiplus` by [adding the Mono repository to your system](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)

- Tutorial: Create a console application with .NET SDK using Visual Studio Code

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install .NET SDK or .NET Runtime on Ubuntu 22.04

Article • 11/21/2023

This article discusses how to install .NET on Ubuntu 22.04; .NET 8, .NET 7, and .NET 6, are supported.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

Important

Using a package manager to install .NET from the **Microsoft package feed** only supports the **x64** architecture. Other architectures, such as **Arm**, aren't supported by the **Microsoft package feed**.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Use the install-dotnet script to install .NET.](#)
- [Manually install .NET.](#)

.NET is available in the [Ubuntu packages feed](#), as well as the Microsoft package repository. However, you should only use one or the other to install .NET. If you want to use the Microsoft package repository, see [How to register the Microsoft package repository](#).

Supported versions

The following versions of .NET are supported or available for Ubuntu 22.04:

 Expand table

Supported .NET versions	Available in Ubuntu feed	Available in Microsoft feed
8.0, 7.0, 6.0	7.0, 6.0	8.0, 7.0, 6.0, 3.1

ⓘ Important

.NET SDK versions offered by Canonical are always in the **.1xx feature band**. If you want to use a newer feature band release, use the [Microsoft feed to install the SDK](#). Make sure you review the information in the [.NET package mix ups on Linux](#) article to understand the implications of switching between repository feeds.

When an [Ubuntu version](#) falls out of support, .NET is no longer supported with that version.

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Install .NET

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following commands:

Bash

```
sudo apt-get update && \
sudo apt-get install -y dotnet-sdk-8.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package dotnet-sdk-8.0**, see the [troubleshooting](#) section.

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following commands install the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following commands:

Bash

```
sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-8.0
```

Important

If you receive an error message similar to **Unable to locate package aspnetcore-runtime-8.0**, see the [troubleshooting](#) section.

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo apt-get install -y dotnet-runtime-8.0
```

How to install other versions

Other versions of .NET aren't supported in the Ubuntu feeds. Instead, use the Microsoft package repository.

Troubleshooting

If you run into issues installing or even running .NET, see [Troubleshooting](#).

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- libc6

- libgcc-s1
- libgssapi-krb5-2
- libicu70
- liblttng-ust1
- libssl3
- libstdc++6
- libunwind8
- zlib1g

Dependencies can be installed with the `apt install` command. The following snippet demonstrates installing the `zlib1g` library:

Bash

```
sudo apt install zlib1g
```

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of `libgdiplus` by [adding the Mono repository to your system](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)



Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

[Open a documentation issue](#)

[Provide product feedback](#)

Install .NET SDK or .NET Runtime on Ubuntu 20.04

Article • 11/21/2023

This article discusses how to install .NET on Ubuntu 20.04; .NET 8, .NET 7, and .NET 6, are supported.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

ⓘ Important

Using a package manager to install .NET from the **Microsoft package feed** only supports the **x64** architecture. Other architectures, such as **Arm**, aren't supported by the **Microsoft package feed**.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Use the install-dotnet script to install .NET.](#)
- [Manually install .NET.](#)

Supported versions

The following versions of .NET are supported or available for Ubuntu 20.04:

[\[+\] Expand table](#)

Supported .NET versions	Available in Ubuntu feed	Available in Microsoft feed
8.0, 7.0, 6.0	None	8.0, 7.0, 6.0, 5.0, 3.1, 2.1

When an [Ubuntu version](#) falls out of support, .NET is no longer supported with that version.

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Add the Microsoft package repository

Installing with APT can be done with a few commands. Before you install .NET, run the following commands to add the Microsoft package signing key to your list of trusted keys and add the package repository.

Open a terminal and run the following commands:

Bash

```
wget https://packages.microsoft.com/config/ubuntu/20.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following commands:

Bash

```
sudo apt-get update &&
sudo apt-get install -y dotnet-sdk-8.0
```

Important

If you receive an error message similar to **Unable to locate package dotnet-sdk-8.0**, see the [troubleshooting](#) section.

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following commands install the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following commands:

Bash

```
sudo apt-get update && \
sudo apt-get install -y aspnetcore-runtime-8.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package aspnetcore-runtime-8.0**, see the [troubleshooting](#) section.

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo apt-get install -y dotnet-runtime-8.0
```

How to install other versions

All versions of .NET are available for download at

<https://dotnet.microsoft.com/download/dotnet>, but require [manual installation](#). You can try and use the package manager to install a different version of .NET. However, the requested version may not be available.

The packages added to package manager feeds are named in a hackable format, for example: `{product}-{type}-{version}`.

- **product**

The type of .NET product to install. Valid options are:

- `dotnet`
- `aspnetcore`

- **type**

Chooses the SDK or the runtime. Valid options are:

- `sdk` (only available for the `dotnet` product)
- `runtime`

- **version**

The version of the SDK or runtime to install. This article will always give the instructions for the latest supported version. Valid options are any released version, such as:

- 8.0
- 6.0
- 3.1
- 2.1

It's possible the SDK/runtime you're trying to download is not available for your Linux distribution. For a list of supported distributions, see [Install .NET on Linux](#).

Examples

- Install the ASP.NET Core 8.0 runtime: `aspnetcore-runtime-8.0`
- Install the .NET Core 2.1 runtime: `dotnet-runtime-2.1`
- Install the .NET 5 SDK: `dotnet-sdk-5.0`
- Install the .NET Core 3.1 SDK: `dotnet-sdk-3.1`

Package missing

If the package-version combination doesn't work, it's not available. For example, there isn't an ASP.NET Core SDK, the SDK components are included with the .NET SDK. The value `aspnetcore-sdk-8.0` is incorrect and should be `dotnet-sdk-8.0`. For a list of Linux distributions supported by .NET, see [.NET dependencies and requirements](#).

Troubleshooting

If you run into issues installing or even running .NET, see [Troubleshooting](#).

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- libc6
- libgcc-s1
- libgssapi-krb5-2
- libicu66

- libssl1.1
- libstdc++6
- zlib1g

Dependencies can be installed with the `apt install` command. The following snippet demonstrates installing the `zlib1g` library:

Bash

```
sudo apt install zlib1g
```

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because `System.Drawing.Common` is no longer supported on Linux, this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of `libgdiplus` by [adding the Mono repository to your system](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install .NET SDK or .NET Runtime on Ubuntu 18.04

Article • 08/17/2023

This article discusses how to install .NET on Ubuntu 18.04; .NET 6 and .NET 7 are supported.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

Important

Package manager installs are only supported on the **x64** architecture. Other architectures, such as **Arm**, must install .NET by some other means such as with Snap, an installer script, or through a manual binary installation.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Alternatively install .NET with install-dotnet script](#).
- [Manually install .NET](#)

Supported versions

The following versions of .NET are supported or available for Ubuntu 18.04:

Supported .NET versions	Available in Ubuntu feed	Available in Microsoft feed
7.0, 6.0	None	7.0, 6.0, 5.0, 3.1, 2.2, 2.1

When an [Ubuntu version](#) falls out of support, .NET is no longer supported with that version.

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Add the Microsoft package repository

Installing with APT can be done with a few commands. Before you install .NET, run the following commands to add the Microsoft package signing key to your list of trusted keys and add the package repository.

Open a terminal and run the following commands:

Bash

```
wget https://packages.microsoft.com/config/ubuntu/18.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following commands:

Bash

```
sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-7.0
```

Important

If you receive an error message similar to **Unable to locate package dotnet-sdk-7.0**, see the [troubleshooting](#) section.

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following commands install the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following commands:

Bash

```
sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-7.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package aspnetcore-runtime-7.0**, see the [troubleshooting](#) section.

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-7.0` in the previous command with `dotnet-runtime-7.0`:

Bash

```
sudo apt-get install -y dotnet-runtime-7.0
```

How to install other versions

All versions of .NET are available for download at

<https://dotnet.microsoft.com/download/dotnet>, but require [manual installation](#). You can try and use the package manager to install a different version of .NET. However, the requested version may not be available.

The packages added to package manager feeds are named in a hackable format, for example: `{product}-{type}-{version}`.

- **product**

The type of .NET product to install. Valid options are:

- `dotnet`
- `aspnetcore`

- **type**

Chooses the SDK or the runtime. Valid options are:

- `sdk` (only available for the `dotnet` product)
- `runtime`

- **version**

The version of the SDK or runtime to install. This article will always give the instructions for the latest supported version. Valid options are any released version, such as:

- 8.0
- 6.0
- 3.1
- 2.1

It's possible the SDK/runtime you're trying to download is not available for your Linux distribution. For a list of supported distributions, see [Install .NET on Linux](#).

Examples

- Install the ASP.NET Core 8.0 runtime: `aspnetcore-runtime-8.0`
- Install the .NET Core 2.1 runtime: `dotnet-runtime-2.1`
- Install the .NET 5 SDK: `dotnet-sdk-5.0`
- Install the .NET Core 3.1 SDK: `dotnet-sdk-3.1`

Package missing

If the package-version combination doesn't work, it's not available. For example, there isn't an ASP.NET Core SDK, the SDK components are included with the .NET SDK. The value `aspnetcore-sdk-8.0` is incorrect and should be `dotnet-sdk-8.0`. For a list of Linux distributions supported by .NET, see [.NET dependencies and requirements](#).

Troubleshooting

If you run into issues installing or even running .NET, see [Troubleshooting](#).

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- libc6
- libgcc1
- libgssapi-krb5-2
- libicu60

- libssl1.1
- libstdc++6
- zlib1g

Dependencies can be installed with the `apt install` command. The following snippet demonstrates installing the `zlib1g` library:

Bash

```
sudo apt install zlib1g
```

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because `System.Drawing.Common` is no longer supported on Linux, this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of `libgdiplus` by [adding the Mono repository to your system](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install .NET SDK or .NET Runtime on Ubuntu 16.04

Article • 08/17/2023

This article discusses how to install .NET on Ubuntu 16.04; Only .NET 6 is supported.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

ⓘ Important

Package manager installs are only supported on the **x64** architecture. Other architectures, such as **Arm**, must install .NET by some other means such as with Snap, an installer script, or through a manual binary installation.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Alternatively install .NET with install-dotnet script](#).
- [Manually install .NET](#)

Supported versions

The following versions of .NET are supported or available for Ubuntu 16.04:

Supported .NET versions	Available in Ubuntu feed	Available in Microsoft feed
6.0	None	6.0, 5.0, 3.1, 3.0, 2.2, 2.1, 2.0

When an [Ubuntu version](#) falls out of support, .NET is no longer supported with that version.

The following versions of .NET are  no longer supported:

- .NET 5

- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Add the Microsoft package repository

Installing with APT can be done with a few commands. Before you install .NET, run the following commands to add the Microsoft package signing key to your list of trusted keys and add the package repository.

Open a terminal and run the following commands:

Bash

```
wget https://packages.microsoft.com/config/ubuntu/16.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

Install .NET 6 SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following commands:

Bash

```
sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-6.0
```

Important

If you receive an error message similar to **Unable to locate package dotnet-sdk-6.0**, see the [troubleshooting](#) section.

Install .NET 6 Runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which

is the most compatible runtime for .NET. In your terminal, run the following commands:

Bash

```
sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-6.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package aspnetcore-runtime-6.0**, see the [troubleshooting](#) section.

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-6.0` in the previous command with `dotnet-runtime-6.0`:

Bash

```
sudo apt-get install -y dotnet-runtime-6.0
```

How to install other versions

All versions of .NET are available for download at <https://dotnet.microsoft.com/download/dotnet>, but require [manual installation](#). You can try and use the package manager to install a different version of .NET. However, the requested version may not be available.

The packages added to package manager feeds are named in a hackable format, for example: `{product}-{type}-{version}`.

- **product**

The type of .NET product to install. Valid options are:

- `dotnet`
- `aspnetcore`

- **type**

Chooses the SDK or the runtime. Valid options are:

- `sdk` (only available for the `dotnet` product)
- `runtime`

- **version**

The version of the SDK or runtime to install. This article will always give the instructions for the latest supported version. Valid options are any released version, such as:

- 8.0
- 6.0
- 3.1
- 2.1

It's possible the SDK/runtime you're trying to download is not available for your Linux distribution. For a list of supported distributions, see [Install .NET on Linux](#).

Examples

- Install the ASP.NET Core 8.0 runtime: `aspnetcore-runtime-8.0`
- Install the .NET Core 2.1 runtime: `dotnet-runtime-2.1`
- Install the .NET 5 SDK: `dotnet-sdk-5.0`
- Install the .NET Core 3.1 SDK: `dotnet-sdk-3.1`

Package missing

If the package-version combination doesn't work, it's not available. For example, there isn't an ASP.NET Core SDK, the SDK components are included with the .NET SDK. The value `aspnetcore-sdk-8.0` is incorrect and should be `dotnet-sdk-8.0`. For a list of Linux distributions supported by .NET, see [.NET dependencies and requirements](#).

Troubleshooting

If you run into issues installing or even running .NET, see [Troubleshooting](#).

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- libc6
- libgcc1
- libgssapi-krb5-2
- libicu55

- libssl1.0.0
- libstdc++6
- zlib1g

Dependencies can be installed with the `apt install` command. The following snippet demonstrates installing the `zlib1g` library:

Bash

```
sudo apt install zlib1g
```

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because `System.Drawing.Common` is no longer supported on Linux, this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of `libgdiplus` by [adding the Mono repository to your system](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install the .NET SDK or the .NET Runtime on Alpine

Article • 01/10/2024

ⓘ Important

.NET 8 was released on November 14, 2023. It may take time for the packages to appear in the package manager feeds.

.NET is supported on Alpine and this article describes how to install .NET on Alpine. When an Alpine version falls out of support, .NET is no longer supported with that version.

If you're using Docker, consider using [official .NET Docker images](#) instead of installing .NET yourself.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

The Alpine package manager supports installing some versions of .NET. If the .NET package is unavailable, you'll need to install .NET in one of the following alternative ways:

- [Use the .NET install script.](#)
- [Download and install .NET manually.](#)

Install .NET 8

.NET 8 isn't yet available in the official Alpine package repository. Use one of the following ways to install .NET 8:

- [Use the .NET install script.](#)
- [Download and install .NET manually.](#)

Install .NET 7

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo apk add dotnet7-sdk
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo apk add aspnetcore7-runtime
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore7-runtime` in the previous command with `dotnet7-runtime`:

Bash

```
sudo apk add dotnet7-runtime
```

Install .NET 6

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo apk add dotnet6-sdk
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo apk add aspnetcore6-runtime
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support. To install it, replace `aspnetcore6-runtime` in the previous command with `dotnet6-runtime`:

Bash

```
sudo apk add dotnet6-runtime
```

Supported distributions

The following table is a list of currently supported .NET releases and the versions of Alpine they're supported on. These versions remain supported until either the version of [.NET reaches end-of-support](#) or the version of [Alpine reaches end-of-life](#).

expand Expand table

Alpine	Supported Version	Available in Package Manager
3.18	.NET 8.0, .NET 7.0, .NET 6.0	.NET 7.0, .NET 6.0
3.17	.NET 8.0, .NET 7.0, .NET 6.0	.NET 7.0, .NET 6.0
3.16	.NET 7.0, .NET 6.0	.NET 6.0
3.15	.NET 7.0, .NET 6.0	None

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1

- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Supported architectures

The following table is a list of currently supported .NET releases and the architecture of Alpine they're supported on. These versions remain supported until either the version of [.NET reaches end-of-support](#) or the architecture of [Alpine is supported](#). Note that only `x86_64`, `armv7`, `aarch64` is officially supported by Microsoft. Other architectures are supported by the distribution maintainers, and can be installed using the `apk` package manager.

[\[\] Expand table](#)

Architecture	.NET 6	.NET 7	.NET 8
x86_64	3.16, 3.17, 3.18	3.17, 3.18	3.17, 3.18
x86	None	None	None
aarch64	3.16, 3.17, 3.18	3.17, 3.18	3.17, 3.18
armv7	3.16, 3.17, 3.18	3.17, 3.18	3.17, 3.18
armhf	None	None	None
s390x	3.17	3.17	3.17
ppc64le	None	None	None
riscv64	None	None	None

Install preview versions

Preview and release candidate versions of .NET aren't available in package repositories. You can install previews and release candidates of .NET in one of the following ways:

- [Scripted install with `install-dotnet.sh`](#)
- [Manual binary extraction](#)

Remove preview versions

When using a package manager to manage your installation of .NET, you may run into a conflict if you've previously installed a preview release. The package manager may interpret the non-preview release as an earlier version of .NET. To install the non-preview release, first uninstall the preview versions. For more information about uninstalling .NET, see [How to remove the .NET Runtime and SDK](#).

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- icu-libs
- krb5-libs
- libgcc
- libgdiplus (if the .NET app requires the *System.Drawing.Common* assembly)
- libintl
- libssl1.1 (for 3.14.x and older)
- libssl3 (for 3.15.x and newer)
- libstdc++
- zlib

To install the needed requirements, run the following command:

```
Bash
```

```
apk add bash icu-libs krb5-libs libgcc libintl libssl1.1 libstdc++ zlib
```

If the .NET app uses the *System.Drawing.Common* assembly, libgdiplus will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

To install libgdiplus on Alpine 3.16 or later, run:

```
Bash
```

```
apk add libgdiplus
```

Next steps

- How to enable TAB completion for the .NET CLI
- Tutorial: Create a console application with .NET SDK using Visual Studio Code

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install the .NET SDK or the .NET Runtime on CentOS Linux

Article • 05/04/2023

.NET is supported on CentOS Linux. This article describes how to install .NET on CentOS Linux. If you need to install .NET On CentOS Stream, see [Install the .NET SDK or the .NET Runtime on RHEL and CentOS Stream](#).

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

Important

Package manager installs are only supported on the **x64** architecture. Other architectures, such as **Arm**, must install .NET by some other means such as with Snap, an installer script, or through a manual binary installation.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Alternatively install .NET with install-dotnet script](#).
- [Manually install .NET](#)

Supported distributions

The following table is a list of currently supported .NET releases on CentOS Linux 7. These versions remain supported until either the version of .NET reaches [end-of-support](#) or the version of CentOS Linux is no longer supported.

CentOS Linux	.NET
7	7, 6

⚠️ Warning

CentOS Linux 8 reached an early End Of Life (EOL) on December 31st, 2021. For more information, see the official [CentOS Linux EOL page](#). Because of this, .NET isn't supported on CentOS Linux 8.

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

ⓘ Important

Package manager installs are only supported on the **x64** architecture. Other architectures, such as **Arm**, must install .NET by some other means such as with Snap, an installer script, or through a manual binary installation.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Alternatively install .NET with *install-dotnet* script.](#)
- [Manually install .NET](#)

Install preview versions

Preview and release candidate versions of .NET aren't available in package repositories. You can install previews and release candidates of .NET in one of the following ways:

- [Scripted install with *install-dotnet.sh*](#)
- [Manual binary extraction](#)

Remove preview versions

When using a package manager to manage your installation of .NET, you may run into a conflict if you've previously installed a preview release. The package manager may interpret the non-preview release as an earlier version of .NET. To install the non-

preview release, first uninstall the preview versions. For more information about uninstalling .NET, see [How to remove the .NET Runtime and SDK](#).

CentOS Linux 7

Before you install .NET, run the following commands to add the Microsoft package signing key to your list of trusted keys and add the Microsoft package repository. Open a terminal and run the following commands:

Bash

```
sudo rpm -Uvh https://packages.microsoft.com/config/centos/7/packages-microsoft-prod.rpm
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo yum install dotnet-sdk-7.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo yum install aspnetcore-runtime-7.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-7.0` in the previous command with `dotnet-runtime-7.0`:

Bash

```
sudo yum install dotnet-runtime-7.0
```

How to install other versions

All versions of .NET are available for download at <https://dotnet.microsoft.com/download/dotnet>, but require [manual installation](#). You can try and use the package manager to install a different version of .NET. However, the requested version may not be available.

The packages added to package manager feeds are named in a hackable format, for example: `{product}-{type}-{version}`.

- **product**

The type of .NET product to install. Valid options are:

- dotnet
- aspnetcore

- **type**

Chooses the SDK or the runtime. Valid options are:

- sdk (only available for the **dotnet** product)
- runtime

- **version**

The version of the SDK or runtime to install. This article will always give the instructions for the latest supported version. Valid options are any released version, such as:

- 8.0
- 6.0
- 3.1
- 2.1

It's possible the SDK/runtime you're trying to download is not available for your Linux distribution. For a list of supported distributions, see [Install .NET on Linux](#).

Examples

- Install the ASP.NET Core 8.0 runtime: `aspnetcore-runtime-8.0`
- Install the .NET Core 2.1 runtime: `dotnet-runtime-2.1`
- Install the .NET 5 SDK: `dotnet-sdk-5.0`
- Install the .NET Core 3.1 SDK: `dotnet-sdk-3.1`

Package missing

If the package-version combination doesn't work, it's not available. For example, there isn't an ASP.NET Core SDK, the SDK components are included with the .NET SDK. The value `aspnetcore-sdk-8.0` is incorrect and should be `dotnet-sdk-8.0`. For a list of Linux distributions supported by .NET, see [.NET dependencies and requirements](#).

Troubleshoot the package manager

This section provides information on common errors you may get while using the package manager to install .NET.

Unable to find package

Important

Package manager installs are only supported on the **x64** architecture. Other architectures, such as **Arm**, must install .NET by some other means such as with Snap, an installer script, or through a manual binary installation.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Alternatively install .NET with install-dotnet script](#).
- [Manually install .NET](#)

Failed to fetch

While installing the .NET package, you may see an error similar to `signature verification failed for file 'repomd.xml' from repository 'packages-microsoft-com-prod'`. Generally speaking, this error means that the package feed for .NET is being upgraded with newer package versions, and that you should try again later. During an upgrade, the package feed should not be unavailable for more than 2 hours. If you continually receive this error for more than 2 hours, please file an issue at <https://github.com/dotnet/core/issues>.

Errors related to missing `fxr`, `libhostfxr.so`, or `FrameworkList.xml`

For more information about solving these problems, see [Troubleshoot fxr, libhostfxr.so, and FrameworkList.xml errors](#).

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

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

If the target runtime environment's OpenSSL version is 1.1 or newer, you'll need to install `compat-openssl110`.

Dependencies can be installed with the `yum install` command. The following snippet demonstrates installing the `libicu` library:

Bash

```
sudo yum install libicu
```

For more information about the dependencies, see [Self-contained Linux apps ↗](#).

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of `libgdiplus` by [adding the Mono repository to your system ↗](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)



Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you



.NET feedback

The .NET documentation is open
source. Provide feedback [here](#).

can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install the .NET SDK or the .NET Runtime on RHEL and CentOS Stream

Article • 11/15/2023

ⓘ Important

.NET 8 was released on November 14, 2023. It may take time for the packages to appear in the package manager feeds.

.NET is supported on Red Hat Enterprise Linux (RHEL). This article describes how to install .NET on RHEL and CentOS Stream.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

Register your Red Hat subscription

To install .NET from Red Hat on RHEL, you first need to register using the Red Hat Subscription Manager. If this hasn't been done on your system, or if you're unsure, see the [Red Hat Product Documentation for .NET](#).

ⓘ Important

This doesn't apply to CentOS Stream.

Supported distributions

The following table is a list of currently supported .NET releases on both RHEL and CentOS Stream. These versions remain supported until either the version of [.NET reaches end-of-support](#) or the Linux distribution is no longer supported.

Distribution	.NET
RHEL 9 (9.1)	8, 7, 6
RHEL 8 (8.7)	8, 7, 6
RHEL 7	6
CentOS Stream 9	8, 7, 6
CentOS Stream 8	8, 7, 6

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Install preview versions

Preview and release candidate versions of .NET aren't available in package repositories. You can install previews and release candidates of .NET in one of the following ways:

- Scripted install with [*install-dotnet.sh*](#)
- Manual binary extraction

Remove preview versions

When using a package manager to manage your installation of .NET, you may run into a conflict if you've previously installed a preview release. The package manager may interpret the non-preview release as an earlier version of .NET. To install the non-preview release, first uninstall the preview versions. For more information about uninstalling .NET, see [How to remove the .NET Runtime and SDK](#).

RHEL 9

.NET is included in the AppStream repositories for RHEL 9.

ⓘ Important

.NET 8 was released on November 14, 2023. It may take time for the packages to appear in the package manager feeds.

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo dnf install dotnet-sdk-8.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo dnf install aspnetcore-runtime-8.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo dnf install dotnet-runtime-8.0
```

RHEL 8

.NET is included in the AppStream repositories for RHEL 8.

ⓘ Important

.NET 8 was released on November 14, 2023. It may take time for the packages to appear in the package manager feeds.

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo dnf install dotnet-sdk-8.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo dnf install aspnetcore-runtime-8.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo dnf install dotnet-runtime-8.0
```

RHEL 7 .NET 8

.NET 8 isn't compatible with RHEL 7 and doesn't work.

RHEL 7 .NET 7

.NET 7 isn't officially supported on RHEL 7. To install .NET 7, see [Install .NET on Linux by using an install script or by extracting binaries](#).

RHEL 7 ✓ .NET 6

The following command installs the `scl-utils` package:

Bash

```
sudo yum install scl-utils
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install .NET SDK, run the following commands:

Bash

```
subscription-manager repos --enable=rhel-7-server-dotnet-rpms
yum install rh-dotnet60 -y
scl enable rh-dotnet60 bash
```

Red Hat does not recommend permanently enabling `rh-dotnet60` because it may affect other programs. If you want to enable `rh-dotnet` permanently, add the following line to your `~/.bashrc` file.

Bash

```
source scl_source enable rh-dotnet60
```

Install the runtime

The .NET Runtime allows you to run apps that were made with .NET that didn't include the runtime. The commands below install the ASP.NET Core Runtime, which is the most compatible runtime for .NET Core. In your terminal, run the following commands.

Bash

```
subscription-manager repos --enable=rhel-7-server-dotnet-rpms
yum install rh-dotnet60-aspnetcore-runtime-6.0 -y
scl enable rh-dotnet60 bash
```

Red Hat does not recommend permanently enabling `rh-dotnet60` because it may affect other programs. If you want to enable `rh-dotnet60` permanently, add the following line

to your `~/.bashrc` file.

Bash

```
source scl_source enable rh-dotnet60
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime that doesn't include ASP.NET Core support: replace `rh-dotnet60-aspnetcore-runtime-6.0` in the preceding command with `rh-dotnet60-dotnet-runtime-6.0`.

CentOS Stream 9 ✓

.NET is included in the AppStream repositories for CentOS Stream 9.

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo dnf install dotnet-sdk-8.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo dnf install aspnetcore-runtime-8.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo dnf install dotnet-runtime-8.0
```

CentOS Stream 8 ✓

Use the Microsoft repository to install .NET:

Bash

```
sudo rpm -Uvh https://packages.microsoft.com/config/centos/8/packages-microsoft-prod.rpm
sudo yum install dotnet-sdk-8.0
```

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

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

If the target runtime environment's OpenSSL version is 1.1 or newer, you'll need to install `compat-openssl110`.

Dependencies can be installed with the `yum install` command. The following snippet demonstrates installing the `libicu` library:

Bash

```
sudo yum install libicu
```

For more information about the dependencies, see [Self-contained Linux apps ↗](#).

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of *libgdiplus* by [adding the Mono repository to your system](#).

How to install other versions

Consult the [Red Hat documentation for .NET](#) on the steps required to install other releases of .NET.

Troubleshoot the package manager

This section provides information on common errors you may get while using the package manager to install .NET or .NET Core.

Errors related to missing `fxr`, `libhostfxr.so`, or `FrameworkList.xml`

For more information about solving these problems, see [Troubleshoot fxr, libhostfxr.so, and FrameworkList.xml errors](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install the .NET SDK or the .NET Runtime on Debian

Article • 11/15/2023

This article describes how to install .NET on Debian. When a Debian version falls out of support, .NET is no longer supported with that version. However, these instructions may help you to get .NET running on those versions, even though it isn't supported.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

ⓘ Important

Using a package manager to install .NET from the **Microsoft package feed** only supports the **x64** architecture. Other architectures, such as **Arm**, aren't supported by the **Microsoft package feed**.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Use the install-dotnet script to install .NET.](#)
- [Manually install .NET.](#)

Supported distributions

The following table is a list of currently supported .NET releases and the versions of Debian they're supported on. These versions remain supported until either the version of [.NET reaches end-of-support](#) or the version of [Debian reaches end-of-life](#).

ⓘ [Expand table](#)

Debian	.NET
12	8, 7, 6

Debian	.NET
11	8, 7, 6
10	7, 6

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Install preview versions

Preview and release candidate versions of .NET aren't available in package repositories. You can install previews and release candidates of .NET in one of the following ways:

- Scripted install with [*install-dotnet.sh*](#)
- Manual binary extraction

Remove preview versions

When using a package manager to manage your installation of .NET, you may run into a conflict if you've previously installed a preview release. The package manager may interpret the non-preview release as an earlier version of .NET. To install the non-preview release, first uninstall the preview versions. For more information about uninstalling .NET, see [How to remove the .NET Runtime and SDK](#).

Debian 12

Installing with APT can be done with a few commands. Before you install .NET, run the following commands to add the Microsoft package signing key to your list of trusted keys and add the package repository.

Open a terminal and run the following commands:

```
Bash
```

```
wget https://packages.microsoft.com/config/debian/12/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following commands:

Bash

```
sudo apt-get update && \
sudo apt-get install -y dotnet-sdk-8.0
```

Important

If you receive an error message similar to **Unable to locate package dotnet-sdk-8.0**, see the [troubleshooting](#) section.

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following commands install the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following commands:

Bash

```
sudo apt-get update && \
sudo apt-get install -y aspnetcore-runtime-8.0
```

Important

If you receive an error message similar to **Unable to locate package aspnetcore-runtime-8.0**, see the [troubleshooting](#) section.

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo apt-get install -y dotnet-runtime-8.0
```

Debian 11

Installing with APT can be done with a few commands. Before you install .NET, run the following commands to add the Microsoft package signing key to your list of trusted keys and add the package repository.

Open a terminal and run the following commands:

Bash

```
wget https://packages.microsoft.com/config/debian/11/packages-microsoft-
prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following commands:

Bash

```
sudo apt-get update && \
sudo apt-get install -y dotnet-sdk-8.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package dotnet-sdk-8.0**, see the [troubleshooting](#) section.

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following commands install the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following commands:

Bash

```
sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-8.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package aspnetcore-runtime-8.0**, see the [troubleshooting](#) section.

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo apt-get install -y dotnet-runtime-8.0
```

Debian 10

Installing with APT can be done with a few commands. Before you install .NET, run the following commands to add the Microsoft package signing key to your list of trusted keys and add the package repository.

Open a terminal and run the following commands:

Bash

```
wget https://packages.microsoft.com/config/debian/10/packages-microsoft-
prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following commands:

Bash

```
sudo apt-get update && \
sudo apt-get install -y dotnet-sdk-7.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package dotnet-sdk-7.0**, see the [troubleshooting](#) section.

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following commands install the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following commands:

Bash

```
sudo apt-get update && \
sudo apt-get install -y aspnetcore-runtime-7.0
```

ⓘ Important

If you receive an error message similar to **Unable to locate package aspnetcore-runtime-7.0**, see the [troubleshooting](#) section.

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-7.0` in the previous command with `dotnet-runtime-7.0`:

Bash

```
sudo apt-get install -y dotnet-runtime-7.0
```

How to install other versions

All versions of .NET are available for download at

<https://dotnet.microsoft.com/download/dotnet>, but require [manual installation](#). You can try and use the package manager to install a different version of .NET. However, the requested version may not be available.

The packages added to package manager feeds are named in a hackable format, for example: `{product}-{type}-{version}`.

- **product**

The type of .NET product to install. Valid options are:

- `dotnet`
- `aspnetcore`

- **type**

Chooses the SDK or the runtime. Valid options are:

- `sdk` (only available for the `dotnet` product)
- `runtime`

- **version**

The version of the SDK or runtime to install. This article will always give the instructions for the latest supported version. Valid options are any released version, such as:

- `8.0`
- `6.0`
- `3.1`
- `2.1`

It's possible the SDK/runtime you're trying to download is not available for your Linux distribution. For a list of supported distributions, see [Install .NET on Linux](#).

Examples

- Install the ASP.NET Core 8.0 runtime: `aspnetcore-runtime-8.0`
- Install the .NET Core 2.1 runtime: `dotnet-runtime-2.1`
- Install the .NET 5 SDK: `dotnet-sdk-5.0`
- Install the .NET Core 3.1 SDK: `dotnet-sdk-3.1`

Package missing

If the package-version combination doesn't work, it's not available. For example, there isn't an ASP.NET Core SDK, the SDK components are included with the .NET SDK. The value `aspnetcore-sdk-8.0` is incorrect and should be `dotnet-sdk-8.0`. For a list of Linux distributions supported by .NET, see [.NET dependencies and requirements](#).

Use APT to update .NET

When a new patch release is available for .NET, you can simply upgrade it through APT with the following commands:

Bash

```
sudo apt-get update
sudo apt-get upgrade
```

If you've upgraded your Linux distribution since installing .NET, you may need to reconfigure the Microsoft package repository. Run the installation instructions for your current distribution version to upgrade to the appropriate package repository for .NET updates.

Troubleshooting

This section provides information on common errors you may get while using APT to install .NET.

Unable to find package

 **Important**

Using a package manager to install .NET from the **Microsoft package feed** only supports the **x64** architecture. Other architectures, such as **Arm**, aren't supported by the **Microsoft package feed**.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Use the install-dotnet script to install .NET.](#)
- [Manually install .NET.](#)

Unable to locate \ Some packages could not be installed

If you receive an error message similar to **Unable to locate package {dotnet-package}** or **Some packages could not be installed**, run the following commands.

There are two placeholders in the following set of commands.

- `{dotnet-package}`

This represents the .NET package you're installing, such as `aspnetcore-runtime-`

8.0. This is used in the following `sudo apt-get install` command.

- `{os-version}`

This represents the distribution version you're on. This is used in the `wget` command below. The distribution version is the numerical value, such as `20.04` on Ubuntu or `10` on Debian.

First, try purging the package list:

Bash

```
sudo dpkg --purge packages-microsoft-prod && sudo dpkg -i packages-microsoft-prod.deb
sudo apt-get update
```

Then, try to install .NET again. If that doesn't work, you can run a manual install with the following commands:

Bash

```
sudo apt-get install -y gpg
wget -O - https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor
-o microsoft.asc.gpg
sudo mv microsoft.asc.gpg /etc/apt/trusted.gpg.d/
wget https://packages.microsoft.com/config/debian/{os-version}/prod.list
sudo mv prod.list /etc/apt/sources.list.d/microsoft-prod.list
sudo chown root:root /etc/apt/trusted.gpg.d/microsoft.asc.gpg
sudo chown root:root /etc/apt/sources.list.d/microsoft-prod.list
sudo apt-get update && \
    sudo apt-get install -y {dotnet-package}
```

Failed to fetch

While installing the .NET package, you may see an error similar to `Failed to fetch ...`

`File has unexpected size ... Mirror sync in progress?`. This error could mean that the package feed for .NET is being upgraded with newer package versions, and that you should try again later. During an upgrade, the package feed shouldn't be unavailable for more than 30 minutes. If you continually receive this error for more than 30 minutes, please file an issue at <https://github.com/dotnet/core/issues>.

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these

libraries are installed:

- libc6
- libgcc1 (for 10.x)
- libgcc-s1 (for 11.x and 12.x)
- libgssapi-krb5-2
- libicu63 (for 10.x)
- libicu67 (for 11.x)
- libicu72 (for 12.x)
- libssl1.1
- libstdc++6
- zlib1g

Dependencies can be installed with the `apt install` command. The following snippet demonstrates installing the `libc6` library:

Bash

```
sudo apt install libc6
```

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of `libgdiplus` by [adding the Mono repository to your system](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

Install the .NET SDK or the .NET Runtime on Fedora

Article • 11/15/2023

.NET is supported on Fedora and this article describes how to install .NET on Fedora. When a Fedora version falls out of support, .NET is no longer supported with that version.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

For more information on installing .NET without a package manager, see one of the following articles:

- [Install the .NET SDK or the .NET Runtime with a script](#).
- [Install the .NET SDK or the .NET Runtime manually](#).

Supported distributions

The following table is a list of currently supported .NET releases and the versions of Fedora they're supported on. These versions remain supported until either the version of [.NET reaches end-of-support](#) or the version of [Fedora reaches end-of-life](#).

[] [Expand table](#)

Fedora	.NET
39	8, 7, 6
38	8, 7, 6
37	8, 7, 6

The following versions of .NET are  no longer supported:

- .NET 5

- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Install .NET 8

Important

.NET 8 was released on November 14, 2023. It may take time for the packages to appear in the package manager feeds.

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo dnf install dotnet-sdk-8.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo dnf install aspnetcore-runtime-8.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo dnf install dotnet-runtime-8.0
```

Install .NET 7

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo dnf install dotnet-sdk-7.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo dnf install aspnetcore-runtime-7.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-7.0` in the previous command with `dotnet-runtime-7.0`:

Bash

```
sudo dnf install dotnet-runtime-7.0
```

Install .NET 6

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the

following command:

```
Bash
```

```
sudo dnf install dotnet-sdk-6.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

```
Bash
```

```
sudo dnf install aspnetcore-runtime-6.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-6.0` in the previous command with `dotnet-runtime-6.0`:

```
Bash
```

```
sudo dnf install dotnet-runtime-6.0
```

Install preview versions

Preview and release candidate versions of .NET aren't available in package repositories. You can install previews and release candidates of .NET in one of the following ways:

- [Scripted install with *install-dotnet.sh*](#)
- [Manual binary extraction](#)

Remove preview versions

When using a package manager to manage your installation of .NET, you may run into a conflict if you've previously installed a preview release. The package manager may interpret the non-preview release as an earlier version of .NET. To install the non-preview release, first uninstall the preview versions. For more information about uninstalling .NET, see [How to remove the .NET Runtime and SDK](#).

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

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

If the target runtime environment's OpenSSL version is 1.1 or newer, you'll need to install `compat-openssl110`.

Dependencies can be installed with the `yum install` command. The following snippet demonstrates installing the `libicu` library:

```
Bash
```

```
sudo yum install libicu
```

For more information about the dependencies, see [Self-contained Linux apps ↗](#).

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of `libgdiplus` by [adding the Mono repository to your system ↗](#).

Install on older distributions

Older versions of Fedora don't contain .NET Core in the default package repositories. You can install .NET with the [dotnet-install.sh](#) script, or use Microsoft's repository to install .NET:

1. First, add the Microsoft signing key to your list of trusted keys.

```
Bash
```

```
sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc
```

2. Next, add the Microsoft package repository. The source of the repository is based on your version of Fedora.

[] [Expand table](#)

Fedora Version	Package repository
36	https://packages.microsoft.com/config/fedora/36/prod.repo
35	https://packages.microsoft.com/config/fedora/35/prod.repo
34	https://packages.microsoft.com/config/fedora/34/prod.repo
33	https://packages.microsoft.com/config/fedora/33/prod.repo
32	https://packages.microsoft.com/config/fedora/32/prod.repo
31	https://packages.microsoft.com/config/fedora/31/prod.repo
30	https://packages.microsoft.com/config/fedora/30/prod.repo
29	https://packages.microsoft.com/config/fedora/29/prod.repo
28	https://packages.microsoft.com/config/fedora/28/prod.repo
27	https://packages.microsoft.com/config/fedora/27/prod.repo

Bash

```
sudo wget -O /etc/yum.repos.d/microsoft-prod.repo
https://packages.microsoft.com/config/fedora/31/prod.repo
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo dnf install dotnet-sdk-7.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo dnf install aspnetcore-runtime-7.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-7.0` in the previous command with `dotnet-runtime-7.0`:

Bash

```
sudo dnf install dotnet-runtime-7.0
```

How to install other versions

All versions of .NET are available for download at

<https://dotnet.microsoft.com/download/dotnet>, but require [manual installation](#). You can try and use the package manager to install a different version of .NET. However, the requested version may not be available.

The packages added to package manager feeds are named in a hackable format, for example: `{product}-{type}-{version}`.

- **product**

The type of .NET product to install. Valid options are:

- `dotnet`
- `aspnetcore`

- **type**

Chooses the SDK or the runtime. Valid options are:

- `sdk` (only available for the `dotnet` product)
- `runtime`

- **version**

The version of the SDK or runtime to install. This article will always give the instructions for the latest supported version. Valid options are any released version, such as:

- `8.0`
- `6.0`

- 3.1
- 2.1

It's possible the SDK/runtime you're trying to download is not available for your Linux distribution. For a list of supported distributions, see [Install .NET on Linux](#).

Examples

- Install the ASP.NET Core 8.0 runtime: `aspnetcore-runtime-8.0`
- Install the .NET Core 2.1 runtime: `dotnet-runtime-2.1`
- Install the .NET 5 SDK: `dotnet-sdk-5.0`
- Install the .NET Core 3.1 SDK: `dotnet-sdk-3.1`

Package missing

If the package-version combination doesn't work, it's not available. For example, there isn't an ASP.NET Core SDK, the SDK components are included with the .NET SDK. The value `aspnetcore-sdk-8.0` is incorrect and should be `dotnet-sdk-8.0`. For a list of Linux distributions supported by .NET, see [.NET dependencies and requirements](#).

Troubleshoot the package manager

This section provides information on common errors you may get while using the package manager to install .NET or .NET Core.

Unable to find package

For more information on installing .NET without a package manager, see one of the following articles:

- [Install the .NET SDK or the .NET Runtime with a script](#).
- [Install the .NET SDK or the .NET Runtime manually](#).

Failed to fetch

While installing the .NET package, you may see an error similar to `signature verification failed for file 'repomd.xml' from repository 'packages-microsoft-com-prod'`. Generally speaking, this error means that the package feed for .NET is being upgraded with newer package versions, and that you should try again later. During an upgrade, the package feed should not be unavailable for more than 2 hours. If you

continually receive this error for more than 2 hours, please file an issue at <https://github.com/dotnet/core/issues>.

Errors related to missing `fxr`, `libhostfxr.so`, `FrameworkList.xml`, or `/usr/share/dotnet`

For more information about solving these problems, see [Troubleshoot `fxr`, `libhostfxr.so`, and `FrameworkList.xml` errors](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install the .NET SDK or the .NET Runtime on openSUSE

Article • 11/14/2023

.NET is supported on openSUSE. This article describes how to install .NET on openSUSE.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

ⓘ Important

Package manager installs are only supported on the **x64** architecture. Other architectures, such as **Arm**, must install .NET by some other means such as with Snap, an installer script, or through a manual binary installation.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Alternatively install .NET with install-dotnet script](#).
- [Manually install .NET](#)

Supported distributions

The following table is a list of currently supported .NET releases on openSUSE 15. These versions remain supported until either the version of [.NET reaches end-of-support](#) or the version of openSUSE is no longer supported.

openSUSE	.NET
15.4+	8, 7, 6

The following versions of .NET are  no longer supported:

- .NET 5

- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Install preview versions

Preview and release candidate versions of .NET aren't available in package repositories. You can install previews and release candidates of .NET in one of the following ways:

- [Scripted install with *install-dotnet.sh*](#)
- [Manual binary extraction](#)

Remove preview versions

When using a package manager to manage your installation of .NET, you may run into a conflict if you've previously installed a preview release. The package manager may interpret the non-preview release as an earlier version of .NET. To install the non-preview release, first uninstall the preview versions. For more information about uninstalling .NET, see [How to remove the .NET Runtime and SDK](#).

openSUSE 15

Before you install .NET, run the following commands to add the Microsoft package signing key to your list of trusted keys and add the Microsoft package repository. Open a terminal and run the following commands:

Bash

```
sudo zypper install libicu
sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc
wget https://packages.microsoft.com/config/opensuse/15/prod.repo
sudo mv prod.repo /etc/zypp/repos.d/microsoft-prod.repo
sudo chown root:root /etc/zypp/repos.d/microsoft-prod.repo
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

```
Bash
```

```
sudo zypper install dotnet-sdk-8.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

```
Bash
```

```
sudo zypper install aspnetcore-runtime-8.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

```
Bash
```

```
sudo zypper install dotnet-runtime-8.0
```

How to install other versions

All versions of .NET are available for download at

<https://dotnet.microsoft.com/download/dotnet>, but require [manual installation](#). You can try and use the package manager to install a different version of .NET. However, the requested version may not be available.

The packages added to package manager feeds are named in a hackable format, for example: `{product}-{type}-{version}`.

- **product**

The type of .NET product to install. Valid options are:

- `dotnet`
- `aspnetcore`

- **type**

Chooses the SDK or the runtime. Valid options are:

- `sdk` (only available for the `dotnet` product)
- `runtime`

- **version**

The version of the SDK or runtime to install. This article will always give the instructions for the latest supported version. Valid options are any released version, such as:

- 8.0
- 6.0
- 3.1
- 2.1

It's possible the SDK/runtime you're trying to download is not available for your Linux distribution. For a list of supported distributions, see [Install .NET on Linux](#).

Examples

- Install the ASP.NET Core 8.0 runtime: `aspnetcore-runtime-8.0`
- Install the .NET Core 2.1 runtime: `dotnet-runtime-2.1`
- Install the .NET 5 SDK: `dotnet-sdk-5.0`
- Install the .NET Core 3.1 SDK: `dotnet-sdk-3.1`

Package missing

If the package-version combination doesn't work, it's not available. For example, there isn't an ASP.NET Core SDK, the SDK components are included with the .NET SDK. The value `aspnetcore-sdk-8.0` is incorrect and should be `dotnet-sdk-8.0`. For a list of Linux distributions supported by .NET, see [.NET dependencies and requirements](#).

Troubleshoot the package manager

This section provides information on common errors you may get while using the package manager to install .NET.

Unable to find package

 **Important**

Package manager installs are only supported on the **x64** architecture. Other architectures, such as **Arm**, must install .NET by some other means such as with Snap, an installer script, or through a manual binary installation.

For more information on installing .NET **without a package manager**, see one of the following articles:

- [Alternatively install .NET with install-dotnet script.](#)
- [Manually install .NET](#)

Failed to fetch

While installing the .NET package, you may see an error similar to `signature verification failed for file 'repomd.xml' from repository 'packages-microsoft-com-prod'`. Generally speaking, this error means that the package feed for .NET is being upgraded with newer package versions, and that you should try again later. During an upgrade, the package feed should not be unavailable for more than 2 hours. If you continually receive this error for more than 2 hours, please file an issue at <https://github.com/dotnet/core/issues>.

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- `krb5`
- `libicu`
- `libopenssl1_0_0`

If the target runtime environment's OpenSSL version is 1.1 or newer, you'll need to install `compat-openssl110`.

Dependencies can be installed with the `zypper install` command. The following snippet demonstrates installing the `krb5` library:

```
Bash
```

```
sudo zypper install krb5
```

For more information about the dependencies, see [Self-contained Linux apps](#).

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install the .NET SDK or the .NET Runtime on RHEL and CentOS Stream

Article • 11/15/2023

ⓘ Important

.NET 8 was released on November 14, 2023. It may take time for the packages to appear in the package manager feeds.

.NET is supported on Red Hat Enterprise Linux (RHEL). This article describes how to install .NET on RHEL and CentOS Stream.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

Register your Red Hat subscription

To install .NET from Red Hat on RHEL, you first need to register using the Red Hat Subscription Manager. If this hasn't been done on your system, or if you're unsure, see the [Red Hat Product Documentation for .NET](#).

ⓘ Important

This doesn't apply to CentOS Stream.

Supported distributions

The following table is a list of currently supported .NET releases on both RHEL and CentOS Stream. These versions remain supported until either the version of [.NET reaches end-of-support](#) or the Linux distribution is no longer supported.

Distribution	.NET
RHEL 9 (9.1)	8, 7, 6
RHEL 8 (8.7)	8, 7, 6
RHEL 7	6
CentOS Stream 9	8, 7, 6
CentOS Stream 8	8, 7, 6

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Install preview versions

Preview and release candidate versions of .NET aren't available in package repositories. You can install previews and release candidates of .NET in one of the following ways:

- Scripted install with [*install-dotnet.sh*](#)
- Manual binary extraction

Remove preview versions

When using a package manager to manage your installation of .NET, you may run into a conflict if you've previously installed a preview release. The package manager may interpret the non-preview release as an earlier version of .NET. To install the non-preview release, first uninstall the preview versions. For more information about uninstalling .NET, see [How to remove the .NET Runtime and SDK](#).

RHEL 9

.NET is included in the AppStream repositories for RHEL 9.

ⓘ Important

.NET 8 was released on November 14, 2023. It may take time for the packages to appear in the package manager feeds.

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo dnf install dotnet-sdk-8.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo dnf install aspnetcore-runtime-8.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo dnf install dotnet-runtime-8.0
```

RHEL 8

.NET is included in the AppStream repositories for RHEL 8.

ⓘ Important

.NET 8 was released on November 14, 2023. It may take time for the packages to appear in the package manager feeds.

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo dnf install dotnet-sdk-8.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo dnf install aspnetcore-runtime-8.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo dnf install dotnet-runtime-8.0
```

RHEL 7 .NET 8

.NET 8 isn't compatible with RHEL 7 and doesn't work.

RHEL 7 .NET 7

.NET 7 isn't officially supported on RHEL 7. To install .NET 7, see [Install .NET on Linux by using an install script or by extracting binaries](#).

RHEL 7 ✓ .NET 6

The following command installs the `scl-utils` package:

Bash

```
sudo yum install scl-utils
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install .NET SDK, run the following commands:

Bash

```
subscription-manager repos --enable=rhel-7-server-dotnet-rpms
yum install rh-dotnet60 -y
scl enable rh-dotnet60 bash
```

Red Hat does not recommend permanently enabling `rh-dotnet60` because it may affect other programs. If you want to enable `rh-dotnet` permanently, add the following line to your `~/.bashrc` file.

Bash

```
source scl_source enable rh-dotnet60
```

Install the runtime

The .NET Runtime allows you to run apps that were made with .NET that didn't include the runtime. The commands below install the ASP.NET Core Runtime, which is the most compatible runtime for .NET Core. In your terminal, run the following commands.

Bash

```
subscription-manager repos --enable=rhel-7-server-dotnet-rpms
yum install rh-dotnet60-aspnetcore-runtime-6.0 -y
scl enable rh-dotnet60 bash
```

Red Hat does not recommend permanently enabling `rh-dotnet60` because it may affect other programs. If you want to enable `rh-dotnet60` permanently, add the following line

to your `~/.bashrc` file.

Bash

```
source scl_source enable rh-dotnet60
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime that doesn't include ASP.NET Core support: replace `rh-dotnet60-aspnetcore-runtime-6.0` in the preceding command with `rh-dotnet60-dotnet-runtime-6.0`.

CentOS Stream 9 ✓

.NET is included in the AppStream repositories for CentOS Stream 9.

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo dnf install dotnet-sdk-8.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo dnf install aspnetcore-runtime-8.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo dnf install dotnet-runtime-8.0
```

CentOS Stream 8 ✓

Use the Microsoft repository to install .NET:

Bash

```
sudo rpm -Uvh https://packages.microsoft.com/config/centos/8/packages-microsoft-prod.rpm
sudo yum install dotnet-sdk-8.0
```

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

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

If the target runtime environment's OpenSSL version is 1.1 or newer, you'll need to install `compat-openssl110`.

Dependencies can be installed with the `yum install` command. The following snippet demonstrates installing the `libicu` library:

Bash

```
sudo yum install libicu
```

For more information about the dependencies, see [Self-contained Linux apps ↗](#).

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can install a recent version of *libgdiplus* by [adding the Mono repository to your system](#).

How to install other versions

Consult the [Red Hat documentation for .NET](#) on the steps required to install other releases of .NET.

Troubleshoot the package manager

This section provides information on common errors you may get while using the package manager to install .NET or .NET Core.

Errors related to missing `fxr`, `libhostfxr.so`, or `FrameworkList.xml`

For more information about solving these problems, see [Troubleshoot fxr, libhostfxr.so, and FrameworkList.xml errors](#).

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install the .NET SDK or the .NET Runtime on SLES

Article • 11/14/2023

.NET is supported on SLES. This article describes how to install .NET on SLES.

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

Supported distributions

The following table is a list of currently supported .NET releases on both SLES 12 SP2 and SLES 15. These versions remain supported until either the version of [.NET reaches end-of-support](#) or the version of SLES is no longer supported.

SLES	.NET
15	8, 7, 6
12 SP5	8, 7, 6

The following versions of .NET are  no longer supported:

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

Install preview versions

Preview and release candidate versions of .NET aren't available in package repositories. You can install previews and release candidates of .NET in one of the following ways:

- Scripted install with *install-dotnet.sh*
- Manual binary extraction

Remove preview versions

When using a package manager to manage your installation of .NET, you may run into a conflict if you've previously installed a preview release. The package manager may interpret the non-preview release as an earlier version of .NET. To install the non-preview release, first uninstall the preview versions. For more information about uninstalling .NET, see [How to remove the .NET Runtime and SDK](#).

SLES 15

Before you install .NET, run the following commands to add the Microsoft package signing key to your list of trusted keys and add the Microsoft package repository. Open a terminal and run the following commands:

Bash

```
sudo rpm -Uvh https://packages.microsoft.com/config/sles/15/packages-microsoft-prod.rpm
```

Currently, the SLES 15 Microsoft repository setup package installs the *microsoft-prod.repo* file to the wrong directory, preventing zypper from finding the .NET packages. To fix this problem, create a symlink in the correct directory.

Bash

```
sudo ln -s /etc/yum.repos.d/microsoft-prod.repo /etc/zypp/repos.d/microsoft-prod.repo
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo zypper install dotnet-sdk-8.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo zypper install aspnetcore-runtime-8.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo zypper install dotnet-runtime-8.0
```

SLES 12

.NET requires SP2 as a minimum for the SLES 12 family.

Before you install .NET, run the following commands to add the Microsoft package signing key to your list of trusted keys and add the Microsoft package repository. Open a terminal and run the following commands:

Bash

```
sudo rpm -Uvh https://packages.microsoft.com/config/sles/12/packages-microsoft-prod.rpm
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

Bash

```
sudo zypper install dotnet-sdk-8.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command installs the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

Bash

```
sudo zypper install aspnetcore-runtime-8.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-8.0` in the previous command with `dotnet-runtime-8.0`:

Bash

```
sudo zypper install dotnet-runtime-8.0
```

How to install other versions

All versions of .NET are available for download at

<https://dotnet.microsoft.com/download/dotnet> ↗, but require [manual installation](#). You can try and use the package manager to install a different version of .NET. However, the requested version may not be available.

The packages added to package manager feeds are named in a hackable format, for example: `{product}-{type}-{version}`.

- **product**

The type of .NET product to install. Valid options are:

- `dotnet`
- `aspnetcore`

- **type**

Chooses the SDK or the runtime. Valid options are:

- `sdk` (only available for the `dotnet` product)
- `runtime`

- **version**

The version of the SDK or runtime to install. This article will always give the instructions for the latest supported version. Valid options are any released version, such as:

- 8.0
- 6.0
- 3.1
- 2.1

It's possible the SDK/runtime you're trying to download is not available for your Linux distribution. For a list of supported distributions, see [Install .NET on Linux](#).

Examples

- Install the ASP.NET Core 8.0 runtime: `aspnetcore-runtime-8.0`
- Install the .NET Core 2.1 runtime: `dotnet-runtime-2.1`
- Install the .NET 5 SDK: `dotnet-sdk-5.0`
- Install the .NET Core 3.1 SDK: `dotnet-sdk-3.1`

Package missing

If the package-version combination doesn't work, it's not available. For example, there isn't an ASP.NET Core SDK, the SDK components are included with the .NET SDK. The value `aspnetcore-sdk-8.0` is incorrect and should be `dotnet-sdk-8.0`. For a list of Linux distributions supported by .NET, see [.NET dependencies and requirements](#).

Troubleshoot the package manager

This section provides information on common errors you may get while using the package manager to install .NET.

Failed to fetch

While installing the .NET package, you may see an error similar to `signature verification failed for file 'repomd.xml' from repository 'packages-microsoft-com-prod'`. Generally speaking, this error means that the package feed for .NET is being upgraded with newer package versions, and that you should try again later. During an upgrade, the package feed should not be unavailable for more than 2 hours. If you continually receive this error for more than 2 hours, please file an issue at <https://github.com/dotnet/core/issues>.

Dependencies

When you install with a package manager, these libraries are installed for you. But, if you manually install .NET or you publish a self-contained app, you'll need to make sure these libraries are installed:

- krb5
- libicu
- libopenssl1_1

If the target runtime environment's OpenSSL version is 1.1 or newer, you'll need to install `compat-openssl10`.

Dependencies can be installed with the `zypper install` command. The following snippet demonstrates installing the `krb5` library:

```
Bash
```

```
sudo zypper install krb5
```

For more information about the dependencies, see [Self-contained Linux apps](#).

If the .NET app uses the `System.Drawing.Common` assembly, `libgdiplus` will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Install .NET on Linux by using an install script or by extracting binaries

Article • 12/30/2023

This article demonstrates how to install the .NET SDK or the .NET Runtime on Linux by using the install script or by extracting the binaries. For a list of distributions that support the built-in package manager, see [Install .NET on Linux](#).

Install the SDK (which includes the runtime) if you want to develop .NET apps. Or, if you only need to run apps, install the Runtime. If you're installing the Runtime, we suggest you install the **ASP.NET Core Runtime** as it includes both .NET and ASP.NET Core runtimes.

Use the `dotnet --list-sdks` and `dotnet --list-runtimes` commands to see which versions are installed. For more information, see [How to check that .NET is already installed](#).

.NET releases

There are two types of supported releases, Long Term Support (LTS) releases or Standard Term Support (STS). The quality of all releases is the same. The only difference is the length of support. LTS releases get free support and patches for 3 years. STS releases get free support and patches for 18 months. For more information, see [.NET Support Policy](#).

The following table lists the support status of each version of .NET (and .NET Core):

⋮ [Expand table](#)

✓ Supported	✗ Unsupported
8 (LTS)	5
7 (STS)	3.1
6 (LTS)	3.0
	2.2
	2.1
	2.0

✓ Supported	✗ Unsupported
	1.1
	1.0

Dependencies

It's possible that when you install .NET, specific dependencies may not be installed, such as when [manually installing](#). The following list details Linux distributions that are supported by Microsoft and have dependencies you may need to install. Check the distribution page for more information:

- [Alpine](#)
- [Debian](#)
- [CentOS](#)
- [Fedora](#)
- [RHEL and CentOS Stream](#)
- [SLES](#)
- [Ubuntu](#)

For generic information about the dependencies, see [Self-contained Linux apps ↗](#).

RPM dependencies

If your distribution wasn't previously listed, and is RPM-based, you may need the following dependencies:

- `krb5-libs`
- `libicu`
- `openssl-libs`

If the target runtime environment's OpenSSL version is 1.1 or newer, install `compat-openssl110`.

DEB dependencies

If your distribution wasn't previously listed, and is debian-based, you may need the following dependencies:

- `libc6`
- `libgcc1`

- libgssapi-krb5-2
- libicu67
- libssl1.1
- libstdc++6
- zlib1g

Common dependencies

If the .NET app uses the *System.Drawing.Common* assembly, libgdiplus will also need to be installed. Because [System.Drawing.Common is no longer supported on Linux](#), this only works on .NET 6 and requires setting the `System.Drawing.EnableUnixSupport` runtime configuration switch.

You can usually install a recent version of *libgdiplus* by [adding the Mono repository to your system](#).

Scripted install

The [dotnet-install scripts](#) are used for automation and non-admin installs of the **SDK** and **Runtime**. You can download the script from <https://dot.net/v1/dotnet-install.sh>. When .NET is installed in this way, you must install the dependencies required by your Linux distribution. Use the links in the [Install .NET on Linux](#) article for your specific Linux distribution.

ⓘ Important

Bash is required to run the script.

You can download the script with `wget`:

Bash

```
wget https://dot.net/v1/dotnet-install.sh -O dotnet-install.sh
```

Before running this script, make sure you grant permission for this script to run as an executable:

Bash

```
chmod +x ./dotnet-install.sh
```

The script defaults to installing the latest [long term support \(LTS\)](#) SDK version, which is .NET 8. To install the latest release, which might not be an (LTS) version, use the `--version latest` parameter.

Bash

```
./dotnet-install.sh --version latest
```

To install .NET Runtime instead of the SDK, use the `--runtime` parameter.

Bash

```
./dotnet-install.sh --version latest --runtime aspnetcore
```

You can install a specific major version with the `--channel` parameter to indicate the specific version. The following command installs .NET 8.0 SDK.

Bash

```
./dotnet-install.sh --channel 8.0
```

For more information, see [dotnet-install scripts reference](#).

To enable .NET on the command line, see [Set environment variables system-wide](#).

Manual install

As an alternative to the package managers, you can download and manually install the SDK and runtime. Manual installation is commonly used as part of continuous integration testing or on an unsupported Linux distribution. For a developer or user, it's better to use a package manager.

Download a **binary** release for either the SDK or the runtime from one of the following sites. The .NET SDK includes the corresponding runtime:

-  [.NET 8 downloads](#)
-  [.NET 7 downloads](#)
-  [.NET 6 downloads](#)
- [All .NET Core downloads](#)

Extract the downloaded file and use the `export` command to set `DOTNET_ROOT` to the extracted folder's location and then ensure .NET is in PATH. Exporting `DOTNET_ROOT`

makes the .NET CLI commands available in the terminal. For more information about .NET environment variables, see [.NET SDK and CLI environment variables](#).

Different versions of .NET can be extracted to the same folder, which coexist side-by-side.

Example

The following commands use Bash to set the environment variable `DOTNET_ROOT` to the current working directory followed by `.dotnet`. That directory is created if it doesn't exist. The `DOTNET_FILE` environment variable is the filename of the .NET binary release you want to install. This file is extracted to the `DOTNET_ROOT` directory. Both the `DOTNET_ROOT` directory and its `tools` subdirectory are added to the `PATH` environment variable.

ⓘ Important

If you run these commands, remember to change the `DOTNET_FILE` value to the name of the .NET binary you downloaded.

Bash

```
DOTNET_FILE=dotnet-sdk-8.0.100-linux-x64.tar.gz
export DOTNET_ROOT=$(pwd)/.dotnet

mkdir -p "$DOTNET_ROOT" && tar zxf "$DOTNET_FILE" -C "$DOTNET_ROOT"

export PATH=$PATH:$DOTNET_ROOT:$DOTNET_ROOT/tools
```

You can install more than one version of .NET in the same folder.

You can also install .NET to the home directory identified by the `HOME` variable or `~` path:

Bash

```
export DOTNET_ROOT=$HOME/.dotnet
```

Verify downloaded binaries

After downloading an installer, verify it to make sure that the file hasn't been changed or corrupted. You can verify the checksum on your computer and then compare it to

what was reported on the download website.

When you download an installer or binary from an official download page, the checksum for the file is displayed. Select the **Copy** button to copy the checksum value to your clipboard.



The screenshot shows the download page for the .NET 8.0 SDK (v8.0.100) - Windows x64 Installer. At the top, the text 'Thanks for downloading .NET 8.0 SDK (v8.0.100) - Windows x64 Installer!' is displayed. Below this, a note says 'Using Visual Studio? This release is only compatible with Visual Studio 2022 (v17.8). Using a different version? See [.NET SDKs for Visual Studio](#).'. A message below that says 'If your download doesn't start after 30 seconds, [click here to download manually](#)'. At the bottom, there are two buttons: 'Direct link' with a link to <https://download.visualstudio.microsoft.com/download/pr/93961dfb-d1e0-49c8-9230-abcba1ebab5a/811ed1eb63d7652325727720edda26a8/dotnet-sdk-8.0.100-win-x64.exe> and 'Checksum (SHA512)' with a value '248acec95b381e5302255310fb9396267fd74a4a2dc2c3a5989031969cb31f8270cbd14bda1bc0352ac90f8138bddad1a58e4af1e56cc4a1613b1cf2854b5'. The 'Copy' button next to the checksum value is highlighted with a red box.

Use the `sha512sum` command to print the checksum of the file you've downloaded. For example, the following command reports the checksum of the *dotnet-sdk-8.0.100-linux-x64.tar.gz* file:

Bash

```
$ sha512sum dotnet-sdk-8.0.100-linux-x64.tar.gz
13905ea20191e70baeba50b0e9bbe5f752a7c34587878ee104744f9fb453bfe439994d389697
22bdae7f60ee047d75dda8636f3ab62659450e9cd4024f38b2a5  dotnet-sdk-8.0.100-
linux-x64.tar.gz
```

Compare the checksum with the value provided by the download site.

ⓘ Important

Even though a Linux file is shown in these examples, this information equally applies to macOS.

Use a checksum file to validate

The .NET release notes contain a link to a checksum file you can use to validate your downloaded file. The following steps describe how to download the checksum file and validate a .NET install binary:

1. The release notes page for .NET 8 on GitHub at <https://github.com/dotnet/core/tree/main/release-notes/8.0> contains a section named **Releases**. The table in that section links to the downloads and checksum files for each .NET 8 release:

Releases 	
Date	Release
2023/11/14	8.0.0
2023/10/10	8.0.0 RC 2
2023/09/12	8.0.0 RC 1
2023/08/08	8.0.0 Preview 7
2023/07/11	8.0.0 Preview 6

2. Select the link for the version of .NET that you downloaded. The previous section used .NET SDK 8.0.100, which is in the .NET 8.0.0 release.
3. In the release page, you can see the .NET Runtime and .NET SDK version, and a link to the checksum file:

.NET 8.0.0 - November 14, 2023 						
The .NET 8.0.0 and .NET SDK 8.0.100 releases are available for download. The latest 8.0 release is always listed at .NET 8.0 Releases .						
Downloads 						
	SDK Installer ¹	SDK Binaries ¹	Runtime Installer	Runtime Binaries	ASP.NET Core Runtime	Windows Desktop Runtime
Windows	x86 x64 Arm64	x86 x64 Arm64	x86 x64 Arm64	x86 x64 Arm64	x86 x64 Hosting Bundle²	x86 x64 Arm64
macOS	x64 ARM64	x64 ARM64	x64 ARM64	x64 ARM64	x64 ARM64	-
Linux	Snap and Package Manager	x64 Arm Arm64 Arm32 Alpine x64 Alpine	Packages (x64)	x64 Arm Arm64 Arm32 Alpine Arm64 Alpine x64 Alpine	x64¹ Arm¹ Arm64¹ x64 Alpine	-
	Checksums	Checksums	Checksums	Checksums	Checksums	Checksums

4. Copy the link to the checksum file.
5. Use the following script, but replace the link to download the appropriate checksum file:

Bash

```
curl -0 https://dotnetcli.blob.core.windows.net/dotnet/checksums/8.0.0-
```

sha.txt

6. With both the checksum file and the .NET release file downloaded to the same directory, use the `sha512sum -c {file} --ignore-missing` command to validate the downloaded file.

When validation passes, you see the file printed with the **OK** status:

Bash

```
$ sha512sum -c 8.0.0-sha.txt --ignore-missing
dotnet-sdk-8.0.100-linux-x64.tar.gz: OK
```

If you see the file marked as **FAILED**, the file you downloaded isn't valid and shouldn't be used.

Bash

```
$ sha512sum -c 8.0.0-sha.txt --ignore-missing
dotnet-sdk-8.0.100-linux-x64.tar.gz: FAILED
sha512sum: WARNING: 1 computed checksum did NOT match
sha512sum: 8.0.0-sha.txt: no file was verified
```

Set environment variables system-wide

If you used the previous install script, the variables set only apply to your current terminal session. Add them to your shell profile. There are many different shells available for Linux and each has a different profile. For example:

- **Bash Shell:** `~/.bash_profile` or `~/.bashrc`
- **Korn Shell:** `~/.kshrc` or `.profile`
- **Z Shell:** `~/.zshrc` or `.zprofile`

Set the following two environment variables in your shell profile:

- `DOTNET_ROOT`

This variable is set to the folder .NET was installed to, such as `$HOME/.dotnet`:

Bash

```
export DOTNET_ROOT=$HOME/.dotnet
```

- PATH

This variable should include both the `DOTNET_ROOT` folder and the `DOTNET_ROOT/tools` folder:

Bash

```
export PATH=$PATH:$DOTNET_ROOT:$DOTNET_ROOT/tools
```

Next steps

- [How to enable TAB completion for the .NET CLI](#)
- [Tutorial: Create a console application with .NET SDK using Visual Studio Code](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to remove the .NET Runtime and SDK

Article • 11/29/2023

Over time, as you install updated versions of the .NET runtime and SDK, you may want to remove outdated versions of .NET from your machine. Uninstalling older versions of the runtime may change the runtime chosen to run shared framework applications, as detailed in the article on [.NET version selection](#).

Should I remove a version?

The [.NET version selection](#) behaviors and the runtime compatibility of .NET across updates enables safe removal of previous versions. .NET runtime updates are compatible within a major version **band** such as 7.x and 6.x. Additionally, newer releases of the .NET SDK generally maintain the ability to build applications that target previous versions of the runtime in a compatible manner.

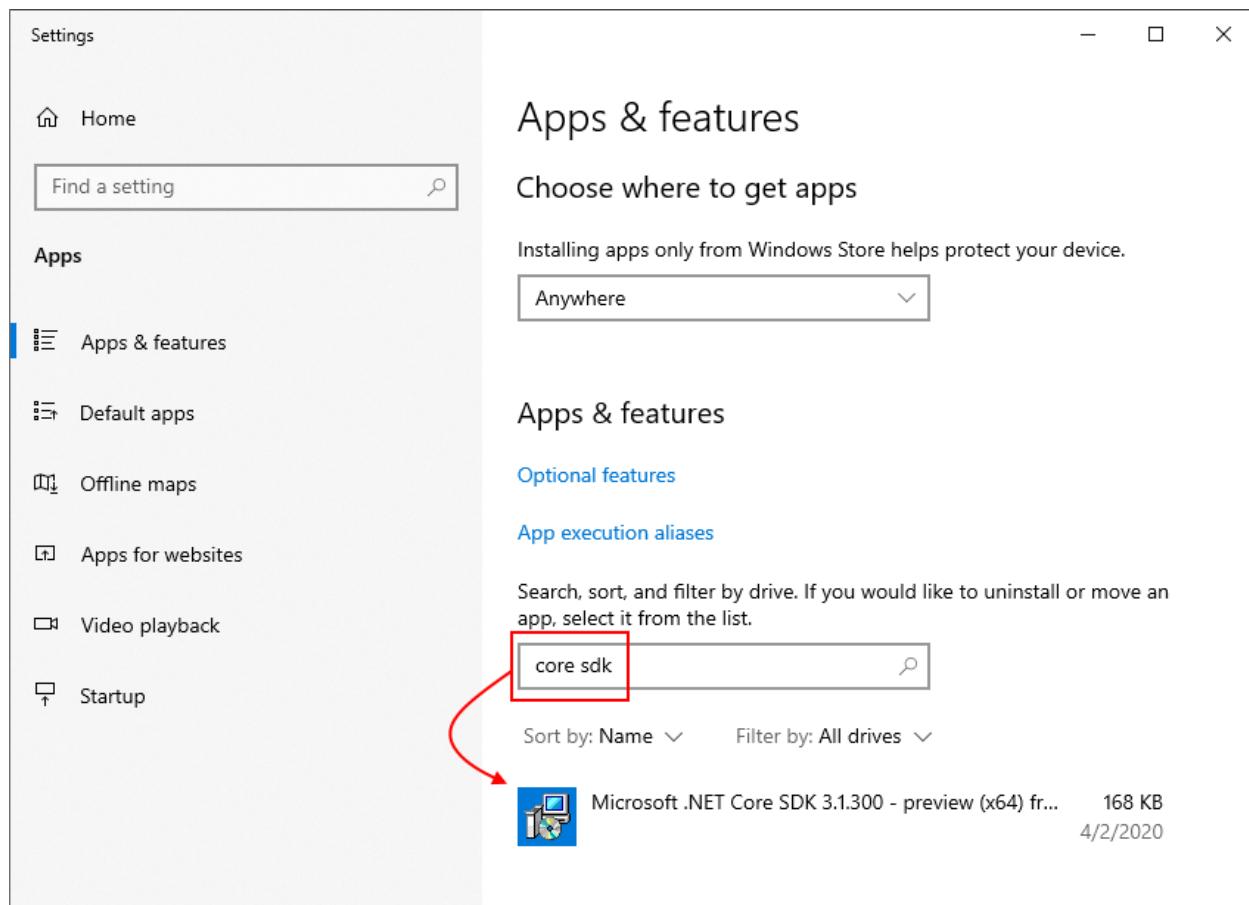
In general, you only need the latest SDK and latest patch version of the runtimes required for your application. Instances where you might want to keep older SDK or runtime versions include maintaining *project.json*-based applications. Unless your application has specific reasons for earlier SDKs or runtimes, you may safely remove older versions.

Determine what is installed

The .NET CLI has options you can use to list the versions of the SDK and runtime that are installed on your computer. Use `dotnet --list-sdks` to see the list of installed SDKs and `dotnet --list-runtimes` for the list of runtimes. For more information, see [How to check that .NET is already installed](#).

Uninstall .NET

.NET uses the Windows **Apps & features** dialog to remove versions of the .NET runtime and SDK. The following figure shows the **Apps & features** dialog. You can search for **core** or **.net** to filter and show installed versions of .NET.



Select any versions you want to remove from your computer and click **Uninstall**.

.NET Uninstall Tool

The [.NET Uninstall Tool](#) (`dotnet-core-uninstall`) lets you remove .NET SDKs and runtimes from a system. A collection of options is available to specify which versions should be uninstalled.

Visual Studio dependency on .NET SDK versions

Before Visual Studio 2019 version 16.3, Visual Studio installers called the standalone SDK installer for .NET Core version 2.1 or 2.2. As a result, the SDK versions appear in the Windows **Apps & features** dialog. Removing .NET SDKs that were installed by Visual Studio using the standalone installer may break Visual Studio. If Visual Studio has problems after you uninstall SDKs, run Repair on that specific version of Visual Studio. The following table shows some of the Visual Studio dependencies on .NET Core SDK versions:

Expand table

Visual Studio version	.NET Core SDK version
Visual Studio 2019 version 16.2	.NET Core SDK 2.2.4xx, 2.1.8xx
Visual Studio 2019 version 16.1	.NET Core SDK 2.2.3xx, 2.1.7xx
Visual Studio 2019 version 16.0	.NET Core SDK 2.2.2xx, 2.1.6xx
Visual Studio 2017 version 15.9	.NET Core SDK 2.2.1xx, 2.1.5xx
Visual Studio 2017 version 15.8	.NET Core SDK 2.1.4xx

Starting with Visual Studio 2019 version 16.3, Visual Studio is in charge of its own copy of the .NET SDK. For that reason, you no longer see those SDK versions in the **Apps & features** dialog.

Remove the NuGet fallback directory

Before .NET Core 3.0 SDK, the .NET Core SDK installers used a directory named *NuGetFallbackFolder* to store a cache of NuGet packages. This cache was used during operations such as `dotnet restore` or `dotnet build /t:Restore`. The *NuGetFallbackFolder* was located under the *sdk* folder where .NET is installed. For example it could be at *C:\Program Files\dotnet\sdk\NuGetFallbackFolder* on Windows and at */usr/local/share/dotnet/sdk/NuGetFallbackFolder* on macOS.

You may want to remove this directory, if:

- You're only developing using .NET Core 3.0 SDK or .NET 5 or later versions.
- You're developing using .NET Core SDK versions earlier than 3.0, but you can work online.

If you want to remove the NuGet fallback directory, you can delete it, but you'll need administrative privileges to do so.

It's not recommended to delete the *dotnet* directory. Doing so would remove any global tools you've previously installed. Also, on Windows:

- You'll break Visual Studio 2019 version 16.3 and later versions. You can run **Repair** to recover.
- If there are .NET Core SDK entries in the **Apps & features** dialog, they'll be orphaned.

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Manage .NET project and item templates

Article • 12/29/2022

.NET provides a template system that enables users to install or uninstall packages containing templates from NuGet, a NuGet package file, or a file system directory. This article describes how to manage .NET templates through the .NET SDK CLI.

For more information about creating templates, see [Tutorial: Create templates](#).

Install template

Template packages are installed through the `dotnet new install` SDK command. You can either provide the NuGet package identifier of a template package, or a folder that contains the template files.

NuGet hosted package

.NET CLI template packages are uploaded to [NuGet](#) for wide distribution. Template packages can also be installed from a private feed. Instead of uploading a template package to a NuGet feed, *nupkg* template files can be distributed and manually installed, as described in the [Local NuGet package](#) section.

For more information about configuring NuGet feeds, see [dotnet nuget add source](#).

To install a template package from the default NuGet feed, use the `dotnet new install {package-id}` command:

.NET CLI

```
dotnet new install Microsoft.DotNet.Web.Spa.ProjectTemplates
```

To install a template package from the default NuGet feed with a specific version, use the `dotnet new install {package-id}::{version}` command:

.NET CLI

```
dotnet new install Microsoft.DotNet.Web.Spa.ProjectTemplates::2.2.6
```

Local NuGet package

When a template package is created, a *nupkg* file is generated. If you have a *nupkg* file containing templates, you can install it with the `dotnet new install {path-to-package}` command:

.NET CLI

```
dotnet new install c:\code\NuGet-Packages\Some.Templates.1.0.0.nupkg
```

Folder

As an alternative to installing template from a *nupkg* file, you can also install templates from a folder directly with the `dotnet new install {folder-path}` command. The folder specified is treated as the template package identifier for any template found. Any template found in the specified folder's hierarchy is installed.

.NET CLI

```
dotnet new install c:\code\NuGet-Packages\some-folder\
```

The `{folder-path}` specified on the command becomes the template package identifier for all templates found. As specified in the [List template packages](#) section, you can get a list of template packages installed with the `dotnet new uninstall` command. In this example, the template package identifier is shown as the folder used for install:

Console

```
dotnet new uninstall
Currently installed items:

... cut to save space ...

c:\code\NuGet-Packages\some-folder
Templates:
  A Template Console Class (templateconsole) C#
  Project for some technology (contosoproject) C#
  Uninstall Command:
    dotnet new uninstall c:\code\NuGet-Packages\some-folder
```

Uninstall template package

Template packages are uninstalled through the `dotnet new uninstall` SDK command. You can either provide the NuGet package identifier of a template package, or a folder that contains the template files.

NuGet package

After a NuGet template package is installed, either from a NuGet feed or a *nupkg* file, you can uninstall it by referencing the NuGet package identifier.

To uninstall a template package, use the `dotnet new uninstall {package-id}` command:

.NET CLI

```
dotnet new uninstall Microsoft.DotNet.Web.Spa.ProjectTemplates
```

Folder

When templates are installed through a [folder path](#), the folder path becomes the template package identifier.

To uninstall a template package, use the `dotnet new uninstall {package-folder-path}` command:

.NET CLI

```
dotnet new uninstall c:\code\NuGet-Packages\some-folder
```

List template packages

By using the standard `uninstall` command without a package identifier, you can see a list of installed template packages along with the command that uninstalls each template package.

Console

```
dotnet new uninstall
Currently installed items:
... cut to save space ...
c:\code\NuGet-Packages\some-folder
Templates:
  A Template Console Class (templateconsole) C#
```

```
Project for some technology (contosoproject) C#
Uninstall Command:
dotnet new uninstall c:\code\NuGet-Packages\some-folder
```

Install template packages from other SDKs

If you've installed each version of the SDK sequentially, for example you installed SDK 6.0, then SDK 7.0, and so on, you'll have every SDK's templates installed. However, if you start with a later SDK version, like 7.0, only the templates for this version are included. Templates for any other release aren't included.

The .NET templates are available on NuGet, and you can install them like any other template. For more information, see [Install NuGet hosted package](#).

SDK	NuGet Package Identifier
.NET Core 2.1	Microsoft.DotNet.Common.ProjectTemplates.2.1
.NET Core 2.2	Microsoft.DotNet.Common.ProjectTemplates.2.2
.NET Core 3.0	Microsoft.DotNet.Common.ProjectTemplates.3.0
.NET Core 3.1	Microsoft.DotNet.Common.ProjectTemplates.3.1
.NET 5.0	Microsoft.DotNet.Common.ProjectTemplates.5.0
.NET 6.0	Microsoft.DotNet.Common.ProjectTemplates.6.0
.NET 7.0	Microsoft.DotNet.Common.ProjectTemplates.7.0
ASP.NET Core 2.1	Microsoft.DotNet.Web.ProjectTemplates.2.1
ASP.NET Core 2.2	Microsoft.DotNet.Web.ProjectTemplates.2.2
ASP.NET Core 3.0	Microsoft.DotNet.Web.ProjectTemplates.3.0
ASP.NET Core 3.1	Microsoft.DotNet.Web.ProjectTemplates.3.1
ASP.NET Core 5.0	Microsoft.DotNet.Web.ProjectTemplates.5.0
ASP.NET Core 6.0	Microsoft.DotNet.Web.ProjectTemplates.6.0
ASP.NET Core 7.0	Microsoft.DotNet.Web.ProjectTemplates.7.0

For example, the .NET 7 SDK includes templates for a console app targeting .NET 7. If you wanted to target .NET Core 3.1, you would need to install the 3.1 template package.

1. Try creating an app that targets .NET Core 3.1.

.NET CLI

```
dotnet new console --framework netcoreapp3.1
```

If you see an error message, you need to install the templates.

2. Install the .NET Core 3.1 project templates.

.NET CLI

```
dotnet new install Microsoft.DotNet.Common.ProjectTemplates.3.1
```

3. Try creating the app a second time.

.NET CLI

```
dotnet new console --framework netcoreapp3.1
```

And you should see a message indicating the project was created.

The template "Console Application" was created successfully.

Processing post-creation actions... Running 'dotnet restore' on path-to-project-file.csproj... Determining projects to restore... Restore completed in 1.05 sec for path-to-project-file.csproj.

Restore succeeded.

See also

- [Tutorial: Create an item template](#)
- [dotnet new](#)
- [dotnet nuget add source](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

macOS Catalina Notarization and the impact on .NET downloads and projects

Article • 05/20/2022

Beginning with macOS Catalina (version 10.15), all software built after June 1, 2019, and distributed with Developer ID, must be notarized. This requirement applies to the .NET runtime, .NET SDK, and software created with .NET. This article describes the common scenarios you may face with .NET and macOS notarization.

Installing .NET

The installers for .NET (both runtime and SDK) have been notarized since February 18, 2020. Prior released versions aren't notarized. You can manually install a non-notarized version of .NET by first downloading the installer, and then using the `sudo installer` command. For more information, see [Download and manually install for macOS](#).

Native appHost

In .NET SDK 7 and later versions, an **appHost**, which is a native Mach-O executable, is produced for your app. This executable is usually invoked by .NET when your project compiles, publishes, or is run with the `dotnet run` command. The non-**appHost** version of your app is a *dll* file that can be invoked by the `dotnet <app.dll>` command.

When run locally, the SDK signs the apphost using [ad hoc signing](#), which allows the app to run locally. When distributing your app, you'll need to properly sign your app according to Apple guidance.

You can also distribute your app without the apphost and rely on users to run your app using `dotnet`. To turn off **appHost** generation, add the `UseAppHost` boolean setting in the project file and set it to `false`. You can also toggle the appHost with the `-p:UseAppHost` parameter on the command line for the specific `dotnet` command you run:

- Project file

XML

```
<PropertyGroup>
  <UseAppHost>false</UseAppHost>
```

```
</PropertyGroup>
```

- Command-line parameter

.NET CLI

```
dotnet run -p:UseAppHost=false
```

An **appHost** is required when you publish your app **self-contained** and you cannot disable it.

For more information about the `UseAppHost` setting, see [MSBuild properties for Microsoft.NET.Sdk](#).

Context of the appHost

When the **appHost** is enabled in your project, and you use the `dotnet run` command to run your app, the app is invoked in the context of the **appHost** and not the default host (the default host is the `dotnet` command). If the **appHost** is disabled in your project, the `dotnet run` command runs your app in the context of the default host. Even if the **appHost** is disabled, publishing your app as **self-contained** generates an **appHost** executable, and users use that executable to run your app. Running your app with `dotnet <filename.dll>` invokes the app with the default host, the shared runtime.

When an app using the **appHost** is invoked, the certificate partition accessed by the app is different from the notarized default host. If your app must access the certificates installed through the default host, use the `dotnet run` command to run your app from its project file, or use the `dotnet <filename.dll>` command to start the app directly.

More information about this scenario is provided in the [ASP.NET Core and macOS and certificates](#) section.

ASP.NET Core, macOS, and certificates

.NET provides the ability to manage certificates in the macOS Keychain with the [System.Security.Cryptography.X509Certificates](#) class. Access to the macOS Keychain uses the applications identity as the primary key when deciding which partition to consider. For example, unsigned applications store secrets in the unsigned partition, but signed applications store their secrets in partitions only they can access. The source of execution that invokes your app decides which partition to use.

.NET provides three sources of execution: [appHost](#), default host (the `dotnet` command), and a custom host. Each execution model may have different identities, either signed or unsigned, and has access to different partitions within the Keychain. Certificates imported by one mode may not be accessible from another. For example, the notarized versions of .NET have a default host that is signed. Certificates are imported into a secure partition based on its identity. These certificates aren't accessible from a generated appHost, as the appHost is ad-hoc signed.

Another example, by default, ASP.NET Core imports a default SSL certificate through the default host. ASP.NET Core applications that use an appHost won't have access to this certificate and will receive an error when .NET detects the certificate isn't accessible. The error message provides instructions on how to fix this problem.

If certificate sharing is required, macOS provides configuration options with the `security` utility.

For more information on how to troubleshoot ASP.NET Core certificate issues, see [Enforce HTTPS in ASP.NET Core](#).

Default entitlements

.NET's default host (the `dotnet` command) has a set of default entitlements. These entitlements are required for proper operation of .NET. It's possible that your application may need additional entitlements, in which case you'll need to generate and use an [appHost](#) and then add the necessary entitlements locally.

Default set of entitlements for .NET:

- `com.apple.security.cs.allow-jit`
- `com.apple.security.cs.allow-unsigned-executable-memory`
- `com.apple.security.cs.allow-dyld-environment-variables`
- `com.apple.security.cs.disable-library-validation`

Notarize a .NET app

If you want your application to run on macOS Catalina (version 10.15) or higher, you'll want to notarize your app. The appHost you submit with your application for notarization should be used with at least the same [default entitlements](#) for .NET Core.

Next steps

- Install .NET on macOS.

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Troubleshoot .NET errors related to missing files on Linux

Article • 05/16/2023

When you try to use .NET on Linux, commands such as `dotnet new` and `dotnet run` may fail with a message related to a file not being found, such as *fxr*, *libhostfxr.so*, or *FrameworkList.xml*. Some of the error messages may be similar to the following items:

- **System.IO.FileNotFoundException**

System.IO.FileNotFoundException: Could not find file
'/usr/share/dotnet/packs/Microsoft.NETCore.App.Ref/5.0.0/data/FrameworkList
.xml'.

- **A fatal error occurred.**

A fatal error occurred. The required library *libhostfxr.so* could not be found.

or

A fatal error occurred. The folder [/usr/share/dotnet/host/fxr] does not exist.

or

A fatal error occurred, the folder [/usr/share/dotnet/host/fxr] does not contain any version-numbered child folders.

- **Generic messages about dotnet not found**

A general message may appear that indicates the SDK isn't found, or that the package has already been installed.

One symptom of these problems is that both the `/usr/lib64/dotnet` and `/usr/share/dotnet` folders are on your system.

💡 Tip

Use the `dotnet --info` command to list which SDKs and Runtimes are installed. For more information, see [How to check that .NET is already installed](#).

What's going on

These errors usually occur when two Linux package repositories provide .NET packages. While Microsoft provides a Linux package repository to source .NET packages, some Linux distributions also provide .NET packages. These distributions include:

- Alpine Linux
- Arch
- CentOS
- CentOS Stream
- Fedora
- RHEL
- Ubuntu 22.04+

If you mix .NET packages from two different sources, you'll likely run into problems. The packages might place things at different paths and might be compiled differently.

Solutions

The solution to these problems is to use .NET from one package repository. Which repository to pick, and how to do it, varies by use-case and the Linux distribution.

- [My Linux distribution provides .NET packages, and I want to use them.](#)
- [I need a version of .NET that isn't provided by my Linux distribution.](#)

My Linux distribution provides .NET packages, and I want to use them

- Do you use the Microsoft repository for other packages, such as PowerShell and MSSQL?
 - Yes

Configure your package manager to ignore the .NET packages from the Microsoft repository. It's possible that you've installed .NET from both repositories, so you want to choose one or the other.

1. Remove the existing .NET packages from your distribution. You want to start over and ensure that you don't install them from the wrong repository.

```
sudo dnf remove 'dotnet*' 'aspnet*' 'netstandard*'
```

2. Configure the Microsoft repository to ignore .NET packages.

Bash

```
echo 'excludepkgs=dotnet*,aspnet*,netstandard*' | sudo tee -a /etc/yum.repos.d/microsoft-prod.repo
```

3. Reinstall .NET from the distribution's package feed. For more information, see [Install .NET on Linux](#).

- o **No**

1. Remove the existing .NET packages from your distribution. You want to start over and ensure that you don't install them from the wrong repository.

Bash

```
sudo dnf remove 'dotnet*' 'aspnet*' 'netstandard*'
```

2. Delete the Microsoft repository feed from your distribution.

Bash

```
sudo dnf remove packages-microsoft-prod
```

3. Reinstall .NET from the distribution's package feed. For more information, see [Install .NET on Linux](#).

I need a version of .NET that isn't provided by my Linux distribution

Configure your package manager to ignore the .NET packages from the distribution's repository. It's possible that you've installed .NET from both repositories, so you want to choose one or the other.

1. Remove the existing .NET packages from your distribution. You want to start over and ensure that you don't install them from the wrong repository.

Bash

```
sudo dnf remove 'dotnet*' 'aspnet*' 'netstandard*'
```

2. Configure the Linux repository to ignore .NET packages.

Bash

```
echo 'excludepkgs=dotnet*,aspnet*,netstandard*' | sudo tee -a  
/etc/yum.repos.d/<your-package-source>.repo
```

Make sure to replace `<your-package-source>` with your distribution's package source.

3. Reinstall .NET from the distribution's package feed. For more information, see [Install .NET on Linux](#).

Online references

Many other users have reported these problems. The following is a list of those issues. You can read through them for insights on what may be happening:

- System.IO.FileNotFoundException and
'/usr/share/dotnet/packs/Microsoft.NETCore.App.Ref/5.0.0/data/FrameworkList.xml'
 - [SDK #15785: unable to build brand new project after upgrading to 5.0.3](#)
 - [SDK #15863: "MSB4018 ResolveTargetingPackAssets task failed unexpectedly" after updating to 5.0.103](#)
 - [SDK #17411: dotnet build always throwing error](#)
 - [SDK #12075: dotnet 3.1.301 on Fedora 32 unable to find FrameworkList.xml because it doesn't exist](#)
- Fatal error: *libhostfxr.so* couldn't be found
 - [SDK #17570: After updating Fedora 33 to 34 and dotnet 5.0.5 to 5.0.6 I get error regarding libhostfxr.so](#)
- Fatal error: folder */host/fxr* doesn't exist
 - [Core #5746: The folder does not exist when installing 3.1 on CentOS 8 with packages.microsoft.com repo enabled](#)
 - [SDK #15476: A fatal error occurred. The folder '/usr/share/dotnet/host/fxr' does not exist](#)
- Fatal error: folder */host/fxr* doesn't contain any version-numbered child folders

- [Installer #9254: Error when install dotnet/core/aspnet:3.1 on CentOS 8 - Folder does not contain any version-numbered child folders ↗](#)
- [StackOverflow: Error when install dotnet/core/aspnet:3.1 on CentOS 8 - Folder does not contain any version-numbered child folders ↗](#)
- Generic errors without clear messages
 - [Core #4605: cannot run "dotnet new console" ↗](#)
 - [Core #4644: Cannot install .NET Core SDK 2.1 on Fedora 32 ↗](#)
 - [Runtime #49375: After updating to 5.0.200-1 using package manager, it appears that no sdks are installed ↗](#)

See also

- [How to check that .NET is already installed](#)
- [How to remove the .NET Runtime and SDK](#)
- [Install .NET on Linux](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

How to check that .NET is already installed

Article • 12/30/2023

This article teaches you how to check which versions of the .NET runtime and SDK are installed on your computer. If you have an integrated development environment, such as Visual Studio, .NET may have already been installed.

Installing an SDK installs the corresponding runtime.

If any command in this article fails, you don't have the runtime or SDK installed. For more information, see the install articles for [Windows](#), [macOS](#), or [Linux](#).

Check SDK versions

You can see which versions of the .NET SDK are currently installed with a terminal. Open a terminal and run the following command.

.NET CLI

```
dotnet --list-sdks
```

You get output similar to the following.

Console

```
3.1.424 [C:\program files\dotnet\sdk]
5.0.100 [C:\program files\dotnet\sdk]
6.0.402 [C:\program files\dotnet\sdk]
7.0.404 [C:\program files\dotnet\sdk]
8.0.100 [C:\program files\dotnet\sdk]
```

Check runtime versions

You can see which versions of the .NET runtime are currently installed with the following command.

.NET CLI

```
dotnet --list-runtimes
```

You get output similar to the following.

```
Console

Microsoft.AspNetCore.App 3.1.30 [C:\Program
Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 6.0.10 [C:\Program
Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 7.0.5 [C:\Program
Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 8.0.0 [C:\Program
Files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.NETCore.App 3.1.30 [C:\Program
Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 5.0.17 [C:\Program
Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 6.0.10 [C:\Program
Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 7.0.5 [C:\Program
Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 8.0.0 [C:\Program
Files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.WindowsDesktop.App 3.1.30 [C:\Program
Files\dotnet\shared\Microsoft.WindowsDesktop.App]
Microsoft.WindowsDesktop.App 6.0.10 [C:\Program
Files\dotnet\shared\Microsoft.WindowsDesktop.App]
Microsoft.WindowsDesktop.App 7.0.5 [C:\Program
Files\dotnet\shared\Microsoft.WindowsDesktop.App]
Microsoft.WindowsDesktop.App 8.0.0 [C:\Program
Files\dotnet\shared\Microsoft.WindowsDesktop.App]
```

Check for install folders

It's possible that .NET is installed but not added to the `PATH` variable for your operating system or user profile. In this case, the commands from the previous sections may not work. As an alternative, you can check that the .NET install folders exist.

When you install .NET from an installer or script, it's installed to a standard folder. Much of the time the installer or script you're using to install .NET gives you an option to install to a different folder. If you choose to install to a different folder, adjust the start of the folder path.

- **dotnet executable**

`C:\program files\dotnet\dotnet.exe`

- **.NET SDK**

`C:\program files\dotnet\sdk\{version}\`

- **.NET Runtime**

`C:\program files\dotnet\shared\{runtime-type}\{version}\`

More information

You can see both the SDK versions and runtime versions with the command `dotnet --info`. You'll also get other environmental related information, such as the operating system version and runtime identifier (RID).

Next steps

- [Install the .NET Runtime and SDK for Windows.](#)
- [Install the .NET Runtime and SDK for macOS.](#)
- [Install the .NET Runtime and SDK for Linux.](#)

See also

- [Determine which .NET Framework versions are installed](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

How to install localized IntelliSense files for .NET

Article • 12/29/2022

IntelliSense is a code-completion aid that's available in different integrated development environments (IDEs), such as Visual Studio. By default, when you're developing .NET projects, the SDK only includes the English version of the IntelliSense files. This article explains:

- How to install the localized version of those files.
- How to modify the Visual Studio installation to use a different language.

Note

Localized IntelliSense files are no longer available. The latest version they're available for is .NET 5. We recommend using the English IntelliSense files.

Prerequisites

- [.NET SDK](#).
- [Visual Studio 2019 version 16.3](#) or a later version.

Download and install the localized IntelliSense files

Important

This procedure requires that you have administrator permission to copy the IntelliSense files to the .NET installation folder.

1. Go to the [Download IntelliSense files](#) page.
2. Download the IntelliSense file for the language and version you'd like to use.
3. Extract the contents of the zip file.
4. Navigate to the .NET Intellisense folder.

- a. Navigate to the .NET installation folder. By default, it's under `%ProgramFiles%\dotnet\packs`.
- b. Choose which SDK you want to install the IntelliSense for, and navigate to the associated path. You have the following options:

SDK type	Path
.NET 6 and later	<code>Microsoft.NETCore.App.Ref</code>
Windows Desktop	<code>Microsoft.WindowsDesktop.App.Ref</code>
.NET Standard	<code>NETStandard.Library.Ref</code>

- c. Navigate to the version you want to install the localized IntelliSense for. For example, `5.0.0`.
- d. Open the `ref` folder.
- e. Open the moniker folder. For example, `net5.0`.

So, the full path that you'd navigate to would look similar to `C:\Program Files\dotnet\packs\Microsoft.NETCore.App.Ref\5.0.0\ref\net5.0`.

5. Create a subfolder inside the moniker folder you just opened. The name of the folder indicates which language you want to use. The following table specifies the different options:

Language	Folder name
Brazilian Portuguese	<code>pt-br</code>
Chinese (simplified)	<code>zh-hans</code>
Chinese (traditional)	<code>zh-hant</code>
French	<code>fr</code>
German	<code>de</code>
Italian	<code>it</code>
Japanese	<code>ja</code>
Korean	<code>ko</code>
Russian	<code>ru</code>
Spanish	<code>es</code>

6. Copy the *.xml* files you extracted in step 3 to this new folder. The *.xml* files are broken down by SDK folders, so copy them to the matching SDK you chose in step 4.

Modify Visual Studio language

For Visual Studio to use a different language for IntelliSense, install the appropriate language pack. This can be done [during installation](#) or at a later time by modifying the Visual Studio installation. If you already have Visual Studio configured to the language of your choice, your IntelliSense installation is ready.

Install the language pack

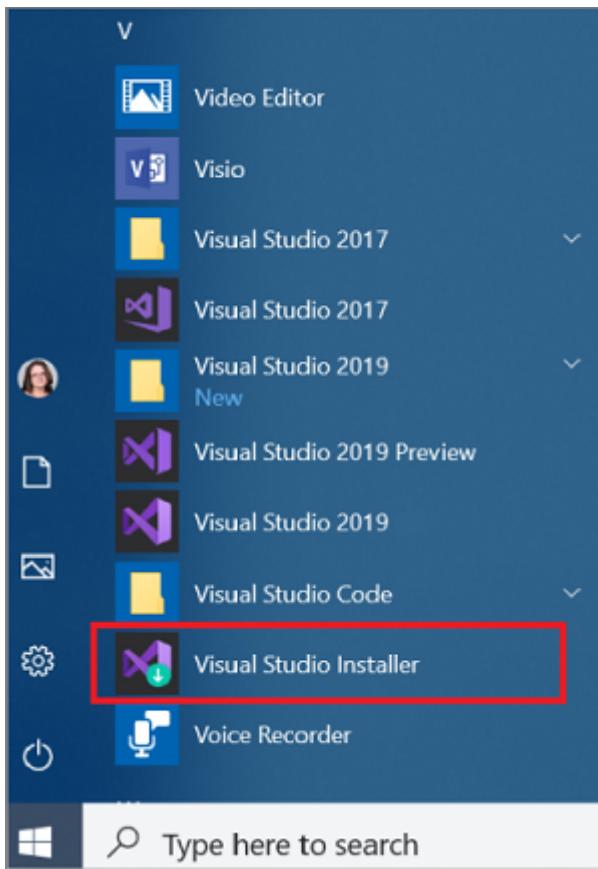
If you didn't install the desired language pack during setup, update Visual Studio as follows to install the language pack:

 **Important**

To install, update, or modify Visual Studio, you must log on with an account that has administrator permission. For more information, see [User permissions and Visual Studio](#).

1. Find the Visual Studio Installer on your computer.

For example, on a computer running Windows 10, select **Start**, and then scroll to the letter **V**, where it's listed as **Visual Studio Installer**.



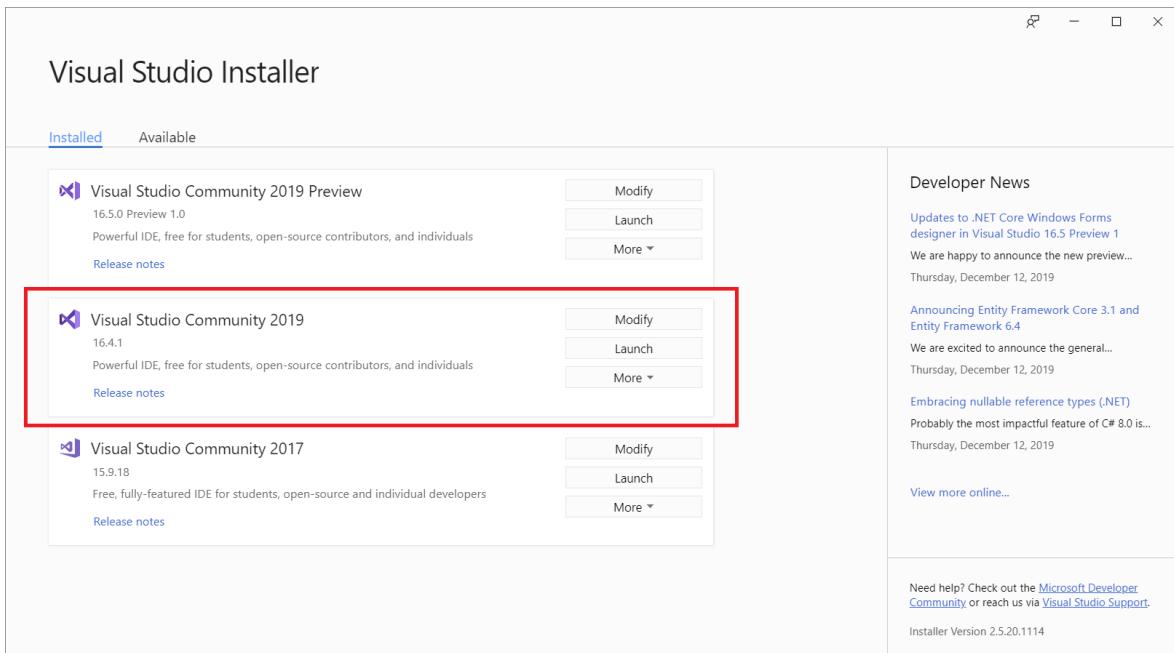
ⓘ Note

You can also find the Visual Studio Installer in the following location:

```
C:\Program Files (x86)\Microsoft Visual Studio\Installer\vs_installer.exe
```

You might have to update the installer before continuing. If so, follow the prompts.

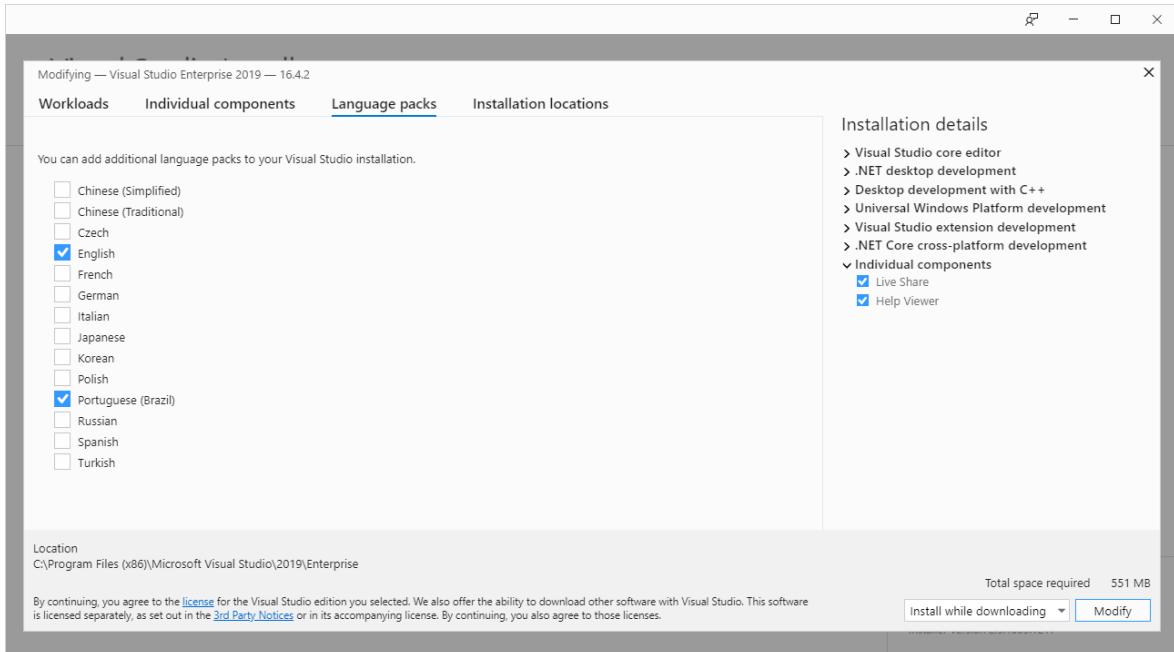
2. In the installer, look for the edition of Visual Studio that you want to add the language pack to, and then choose **Modify**.



ⓘ Important

If you don't see a **Modify** button but you see an **Update** one instead, you need to update your Visual Studio before you can modify your installation. After the update is finished, the **Modify** button should appear.

3. In the **Language packs** tab, select or deselect the languages you want to install or uninstall.



4. Choose **Modify**. The update starts.

Modify language settings in Visual Studio

Once you've installed the desired language packs, modify your Visual Studio settings to use a different language:

1. Open Visual Studio.
2. On the start window, choose **Continue without code**.
3. On the menu bar, select **Tools > Options**. The Options dialog opens.
4. Under the **Environment** node, choose **International Settings**.
5. On the **Language** drop-down, select the desired language. Choose **OK**.
6. A dialog informs you that you have to restart Visual Studio for the changes to take effect. Choose **OK**.
7. Restart Visual Studio.

After this, your IntelliSense should work as expected when you open a .NET project that targets the version of the IntelliSense files you just installed.

See also

- [IntelliSense in Visual Studio](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Introduction to .NET

Article • 01/10/2024

.NET is a free, cross-platform, [open-source developer platform](#) for building [many kinds of applications](#). It can run programs written in [multiple languages](#), with [C#](#) being the most popular. It relies on a [high-performance](#) runtime that is used in production by many [high-scale apps](#).

To learn how to [download .NET](#) and start writing your first app, see [Getting started](#).

The .NET platform has been designed to deliver productivity, performance, security, and reliability. It provides automatic memory management via a [garbage collector \(GC\)](#). It is type-safe and memory-safe, due to using a GC and strict language compilers. It offers [concurrency](#) via `async/await` and `Task` primitives. It includes a large set of libraries that have broad functionality and have been optimized for performance on multiple operating systems and chip architectures.

.NET has the following [design points](#):

- **Productivity is full-stack** with runtime, libraries, language, and tools all contributing to developer user experience.
- **Safe code** is the primary compute model, while [unsafe code](#) enables additional manual optimizations.
- **Static and dynamic code** are both supported, enabling a broad set of distinct scenarios.
- **Native code interop and hardware intrinsics** are low cost and high-fidelity (raw API and instruction access).
- **Code is portable across platforms** (OS and chip architecture), while platform targeting enables specialization and optimization.
- **Adaptability across programming domains** (cloud, client, gaming) is enabled with specialized implementations of the general-purpose programming model.
- **Industry standards** like OpenTelemetry and gRPC are favored over bespoke solutions.

.NET is maintained by Microsoft and the community. It is regularly updated to ensure users deploy secure and reliable applications to production.

Components

.NET includes the following components:

- Runtime -- executes application code.
- Libraries -- provides utility functionality like [JSON parsing](#).
- Compiler -- compiles C# (and other languages) source code into (runtime) executable code.
- SDK and other tools -- enable building and monitoring apps with modern workflows.
- App stacks -- like ASP.NET Core and Windows Forms, that enable writing apps.

The runtime, libraries, and languages are the pillars of the .NET stack. Higher-level components, like .NET tools, and app stacks, like ASP.NET Core, build on top of these pillars. C# is the primary programming language for .NET and much of .NET is written in C#.

C# is object-oriented and the runtime supports object orientation. C# requires garbage collection and the runtime provides a tracing garbage collector. The libraries (and also the app stacks) shape those capabilities into concepts and object models that enable developers to productively write algorithms in intuitive workflows.

The core libraries expose thousands of types, many of which integrate with and fuel the C# language. For example, C#'s `foreach` statement lets you enumerate arbitrary collections. Pattern-based optimizations enable collections like `List<T>` to be processed simply and efficiently. You can leave resource management up to garbage collection, but prompt cleanup is possible via `IDisposable` and direct language support in the `using` statement.

Support for doing multiple things at the same time is fundamental to practically all workloads. That could be client applications doing background processing while keeping the UI responsive, services handling many thousands of simultaneous requests, devices responding to a multitude of simultaneous stimuli, or high-powered machines parallelizing the processing of compute-intensive operations. Asynchronous programming support is a first-class feature of the C# programming language, which provides the `async` and `await` keywords that make it easy to write and compose asynchronous operations while still enjoying the full benefits of all the control flow constructs the language has to offer.

The [type system](#) offers significant breadth, catering somewhat equally to safety, descriptiveness, dynamism, and native interop. First and foremost, the type system enables an object-oriented programming model. It includes types, (single base class) inheritance, interfaces (including default method implementations), and virtual method dispatch to provide a sensible behavior for all the type layering that object orientation allows. [Generic types](#) are a pervasive feature that let you specialize classes to one or more types.

The .NET runtime provides automatic memory management via a garbage collector. For any language, its memory management model is likely its most defining characteristic. This is true for .NET languages. .NET has a self-tuning, tracing GC. It aims to deliver "hands off" operation in the general case while offering configuration options for more extreme workloads. The current GC is the result of many years of investment and learnings from a multitude of workloads.

Value types and stack-allocated memory blocks offer more direct, low-level control over data and native platform interop, in contrast to .NET's GC-managed types. Most of the primitive types in .NET, like integer types, are value types, and users can define their own types with similar semantics. Value types are fully supported through .NET's generics system, meaning that generic types like `List<T>` can provide flat, no-overhead memory representations of value type collections.

[Reflection](#) is a "programs as data" paradigm, allowing one part of a program to dynamically query and invoke another, in terms of assemblies, types, and members. It's particularly useful for late-bound programming models and tools.

Exceptions are the primary error handling model in .NET. Exceptions have the benefit that error information does not need to be represented in method signatures or handled by every method. Proper exception handling is essential for application reliability. To prevent your app from crashing, you can intentionally handle expected exceptions in your code. A crashed app is more reliable and diagnosable than an app with undefined behavior.

App stacks, like ASP.NET Core and Windows Forms, build on and take advantage of low-level libraries, language, and runtime. The app stacks define the way that apps are constructed and their lifecycle of execution.

The SDK and other tools enable a modern developer experience, both on a developer desktop and for continuous integration (CI). The modern developer experience includes being able to build, analyze, and test code. .NET projects can often be built by a single `dotnet build` command, which orchestrates restoring NuGet packages and building dependencies.

NuGet is the package manager for .NET. It contains hundreds of thousands of packages that implement functionality for many scenarios. A majority of apps rely on NuGet packages for some functionality. The [NuGet Gallery](#) is maintained by Microsoft.

Free and open source

.NET is free, open source, and is a [.NET Foundation](#) project. .NET is maintained by Microsoft and the community on GitHub in [several repositories](#).

.NET source and binaries are licensed with the [MIT license](#). Additional [licenses apply on Windows](#).

Support

.NET is [supported by multiple organizations](#) that work to ensure that .NET can run on [multiple operating systems](#) and is kept up to date. It can be used on Arm64, x64, and x86 architectures.

New versions of .NET are released annually in November, per our [releases and support policies](#). It is [updated monthly](#) on Patch Tuesday (second Tuesday), typically at 10AM Pacific time.

.NET ecosystem

There are multiple variants of .NET, each supporting a different type of app. The reason for multiple variants is part historical, part technical.

.NET implementations:

- **.NET Framework** -- The original .NET. It provides access to the broad capabilities of Windows and Windows Server. It is actively supported, in maintenance.
- **Mono** -- The original community and open source .NET. A cross-platform implementation of .NET Framework. Actively supported for Android, iOS, and WebAssembly.
- **.NET (Core)** -- Modern .NET. A cross-platform and open source implementation of .NET, rethought for the cloud age while remaining significantly compatible with .NET Framework. Actively supported for Linux, macOS, and Windows.

Next steps

- [Choose a .NET tutorial](#)
- [Try .NET in your browser](#)
- [Take a tour of C#](#)
- [Take a tour of F#](#)

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Build apps with .NET

Article • 01/10/2024

.NET has support for building many kinds of apps, including client, cloud, and gaming.

Cloud apps

- [.NET Aspire](#)
- [Serverless functions](#)
- [Web and microservices](#)

Client apps

- [Mobile](#)
- [Games ↗](#)
- [Desktop apps](#)

Other app types

- [Console apps](#)
- [Internet of Things \(IoT\)](#)
- [Machine learning](#)
- [Windows services](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Microsoft .NET language strategy

Article • 02/06/2023

Microsoft offers 3 languages on the .NET platform – C#, F# and Visual Basic. In this article, you'll learn about our strategy for each language. Look for links to additional articles on how these strategies guide us and ways to learn more about each language.

C#

C# is a cross-platform general purpose language that makes developers productive while writing highly performant code. With millions of developers, C# is the most popular .NET language. C# has broad support in the ecosystem and all .NET [workloads](#). Based on object-oriented principles, it incorporates many features from other paradigms, not least functional programming. Low-level features support high-efficiency scenarios without writing unsafe code. Most of the .NET runtime and libraries are written in C#, and advances in C# often benefit all .NET developers.

Our strategy for C#

We will keep evolving C# to meet the changing needs of developers and remain a state-of-the-art programming language. We will innovate eagerly and broadly in collaboration with the teams responsible for .NET libraries, developer tools, and workload support, while being careful to stay within the spirit of the language. Recognizing the diversity of domains where C# is being used, we will prefer language and performance improvements that benefit all or most developers and maintain a high commitment to backwards compatibility. We will continue to empower the broader .NET ecosystem and grow its role in C#'s future, while maintaining stewardship of design decisions.

You can read more about how this strategy guides us in the [C# guide](#).

F#

F# is a succinct, robust and performant language that is expression-based and immutable by default. It focuses on expressive power, simplicity and elegance and is used by many thousands of developers that appreciate its pragmatic function-first approach to .NET. F# offers the full power of .NET and works well with C# for mixed language solutions. The community makes significant contributions to the compiler and runtime, as well as a broad array of F# tools and frameworks.

Our strategy for F#

We will drive F# evolution and support the F# ecosystem with language leadership and governance. We will encourage community contributions to improve the F# language and developer experience. We will continue to rely on the community to provide important libraries, developer tools and [workload](#) support. As the language evolves, F# will support .NET platform improvements and maintain interoperability with new C# features. We will work across language, tooling, and documentation to lower the barrier to entry into F# for new developers and organizations as well as broadening its reach into new domains.

You can read more about how this strategy guides us in the [F# guide](#).

Visual Basic

Visual Basic (VB) has a long history as an approachable language favoring clarity over brevity. Its hundreds of thousands of developers are concentrated around the traditional Windows-based client [workloads](#) where VB has long pioneered great tooling and ease of use. Today's VB developers benefit from a stable and mature object-oriented language paired with a growing .NET ecosystem and ongoing tooling improvements. Some .NET workloads are not supported in VB, and it is common for VB developers to use C# for those scenarios.

Our strategy for Visual Basic

We will ensure Visual Basic remains a straightforward and approachable language with a stable design. The core libraries of .NET (such as the BCL) will support VB and many of the improvements to the .NET Runtime and libraries will automatically benefit VB. When C# or the .NET Runtime introduce new features that would require language support, VB will generally adopt a consumption-only approach and avoid new syntax. We do not plan to extend Visual Basic to new workloads. We will continue to invest in the experience in Visual Studio and interop with C#, especially in core VB scenarios such as Windows Forms and libraries.

You can read more about how this strategy guides us in the [Visual Basic guide](#).

 Collaborate with us on
GitHub



.NET feedback

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

.NET implementations

Article • 11/19/2023

A .NET app is developed for one or more *implementations* of .NET. Implementations of .NET include .NET Framework, .NET 5+ (and .NET Core), and Mono.

Each implementation of .NET includes the following components:

- One or more runtimes—for example, .NET Framework CLR and .NET 8 CLR.
- A class library—for example, .NET Framework Base Class Library and .NET 8 Base Class Library.
- Optionally, one or more application frameworks—for example, [ASP.NET](#), [Windows Forms](#), and [Windows Presentation Foundation \(WPF\)](#) are included in .NET Framework and .NET 5+.
- Optionally, development tools. Some development tools are shared among multiple implementations.

There are four .NET implementations that Microsoft supports:

- .NET 6 and later versions
- .NET Framework
- Mono
- UWP

.NET, previously referred to as .NET Core, is currently the primary implementation. .NET (8) is built on a single code base that supports multiple platforms and many workloads, such as Windows desktop apps and cross-platform console apps, cloud services, and websites. [Some workloads](#), such as .NET WebAssembly build tools, are available as optional installations.

.NET 5 and later versions

.NET, previously referred to as .NET Core, is a cross-platform implementation of .NET that's designed to handle server and cloud workloads at scale. It also supports other workloads, including desktop apps. It runs on Windows, macOS, and Linux. It implements .NET Standard, so code that targets .NET Standard can run on .NET. [ASP.NET Core](#), [Windows Forms](#), and [Windows Presentation Foundation \(WPF\)](#) all run on .NET.

.NET 8 is the latest version of this .NET implementation.

For more information, see the following resources:

- [.NET introduction](#)
- [.NET vs. .NET Framework for server apps](#)
- [.NET 5+ and .NET Standard](#)

.NET Framework

.NET Framework is the original .NET implementation that has existed since 2002. Versions 4.5 and later implement .NET Standard, so code that targets .NET Standard can run on those versions of .NET Framework. It contains additional Windows-specific APIs, such as APIs for Windows desktop development with Windows Forms and WPF. .NET Framework is optimized for building Windows desktop applications.

For more information, see the [.NET Framework guide](#).

Mono

Mono is a .NET implementation that is mainly used when a small runtime is required. It is the runtime that powers Xamarin applications on Android, macOS, iOS, tvOS, and watchOS and is focused primarily on a small footprint. Mono also powers games built using the Unity engine.

It supports all of the currently published .NET Standard versions.

Historically, Mono implemented the larger API of .NET Framework and emulated some of the most popular capabilities on Unix. It is sometimes used to run .NET applications that rely on those capabilities on Unix.

Mono is typically used with a just-in-time compiler, but it also features a full static compiler (ahead-of-time compilation) that is used on platforms like iOS.

For more information, see the [Mono documentation](#).

Universal Windows Platform (UWP)

UWP is an implementation of .NET that is used for building modern, touch-enabled Windows applications and software for the Internet of Things (IoT). It's designed to unify the different types of devices that you may want to target, including PCs, tablets, phones, and even the Xbox. UWP provides many services, such as a centralized app store, an execution environment (AppContainer), and a set of Windows APIs to use instead of Win32 (WinRT). Apps can be written in C++, C#, Visual Basic, and JavaScript.

For more information, see [Introduction to the Universal Windows Platform](#).

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

.NET class libraries

Article • 01/12/2022

Class libraries are the [shared library](#) concept for .NET. They enable you to componentize useful functionality into modules that can be used by multiple applications. They can also be used as a means of loading functionality that is not needed or not known at application startup. Class libraries are described using the [.NET Assembly file format](#).

There are three types of class libraries that you can use:

- **Platform-specific** class libraries have access to all the APIs in a given platform (for example, .NET Framework on Windows, Xamarin iOS), but can only be used by apps and libraries that target that platform.
- **Portable** class libraries have access to a subset of APIs, and can be used by apps and libraries that target multiple platforms.
- **.NET Standard** class libraries are a merger of the platform-specific and portable library concept into a single model that provides the best of both.

Platform-specific class libraries

Platform-specific libraries are bound to a single .NET platform (for example, .NET Framework on Windows) and can therefore take significant dependencies on a known execution environment. Such an environment exposes a known set of APIs (.NET and OS APIs) and maintains and exposes expected state (for example, Windows registry).

Developers who create platform-specific libraries can fully exploit the underlying platform. The libraries will only ever run on that given platform, making platform checks or other forms of conditional code unnecessary (modulo single sourcing code for multiple platforms).

Platform-specific libraries have been the primary class library type for the .NET Framework. Even as other .NET implementations emerged, platform-specific libraries remained the dominant library type.

Portable class libraries

Portable libraries are supported on multiple .NET implementations. They can still take dependencies on a known execution environment, however, the environment is a synthetic one that's generated by the intersection of a set of concrete .NET

implementations. Exposed APIs and platform assumptions are a subset of what would be available to a platform-specific library.

You choose a platform configuration when you create a portable library. The platform configuration is the set of platforms that you need to support (for example, .NET Framework 4.5+, Windows Phone 8.0+). The more platforms you opt to support, the fewer APIs and fewer platform assumptions you can make, the lowest common denominator. This characteristic can be confusing at first, since people often think "more is better" but find that more supported platforms results in fewer available APIs.

Many library developers have switched from producing multiple platform-specific libraries from one source (using conditional compilation directives) to portable libraries. There are [several approaches](#) for accessing platform-specific functionality within portable libraries, with bait-and-switch being the most widely accepted technique at this point.

.NET Standard class libraries

.NET Standard libraries are a replacement of the platform-specific and portable libraries concepts. They are platform-specific in the sense that they expose all functionality from the underlying platform (no synthetic platforms or platform intersections). They are portable in the sense that they work on all supporting platforms.

.NET Standard exposes a set of library *contracts*. .NET implementations must support each contract fully or not at all. Each implementation, therefore, supports a set of .NET Standard contracts. The corollary is that each .NET Standard class library is supported on the platforms that support its contract dependencies.

.NET Standard does not expose the entire functionality of .NET Framework (nor is that a goal), however, the libraries do expose many more APIs than Portable Class Libraries.

The following implementations support .NET Standard libraries:

- .NET Core
- .NET Framework
- Mono
- Universal Windows Platform (UWP)

For more information, see [.NET Standard](#).

Mono class libraries

Class libraries are supported on Mono, including the three types of libraries described previously. Mono is often viewed as a cross-platform implementation of .NET Framework. In part, this is because platform-specific .NET Framework libraries can run on the Mono runtime without modification or recompilation. This characteristic was in place before the creation of portable class libraries, so was an obvious choice to enable binary portability between .NET Framework and Mono (although it only worked in one direction).

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

.NET Standard

Article • 11/15/2023

.NET Standard is a formal specification of .NET APIs that are available on multiple .NET implementations. The motivation behind .NET Standard was to establish greater uniformity in the .NET ecosystem. .NET 5 and later versions adopt a different approach to establishing uniformity that eliminates the need for .NET Standard in most scenarios. However, if you want to share code between .NET Framework and any other .NET implementation, such as .NET Core, your library should target .NET Standard 2.0. [No new versions of .NET Standard will be released](#), but .NET 5 and all later versions will continue to support .NET Standard 2.1 and earlier.

For information about choosing between .NET 5+ and .NET Standard, see [.NET 5+ and .NET Standard](#) later in this article.

.NET Standard versions

.NET Standard is versioned. Each new version adds more APIs. When a library is built against a certain version of .NET Standard, it can run on any .NET implementation that implements that version of .NET Standard (or higher).

Targeting a higher version of .NET Standard allows a library to use more APIs but means it can only be used on more recent versions of .NET. Targeting a lower version reduces the available APIs but means the library can run in more places.

Select .NET Standard version

1.0

.NET Standard 1.0 has 7,949 of the 37,118 available APIs.

[Expand table](#)

.NET implementation	Version support
.NET and .NET Core	1.0, 1.1, 2.0, 2.1, 2.2, 3.0, 3.1, 5.0, 6.0, 7.0, 8.0
.NET Framework	4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
Mono	4.6, 5.4, 6.4
Xamarin.iOS	10.0, 10.14, 12.16

.NET implementation	Version support
Xamarin.Mac	3.0, 3.8, 5.16
Xamarin.Android	7.0, 8.0, 10.0
Universal Windows Platform	8.0, 8.1, 10.0, 10.0.16299, TBD
Unity	2018.1

For more information, see [.NET Standard 1.0](#). For an interactive table, see [.NET Standard versions](#).

Which .NET Standard version to target

We recommend you target .NET Standard 2.0, unless you need to support an earlier version. Most general-purpose libraries should not need APIs outside of .NET Standard 2.0. .NET Standard 2.0 is supported by all modern platforms and is the recommended way to support multiple platforms with one target.

If you need to support .NET Standard 1.x, we recommend that you *also* target .NET Standard 2.0. .NET Standard 1.x is distributed as a granular set of NuGet packages, which creates a large package dependency graph and results in developers downloading a lot of packages when building. For more information, see [Cross-platform targeting](#) and [.NET 5+ and .NET Standard](#) later in this article.

.NET Standard versioning rules

There are two primary versioning rules:

- **Additive:** .NET Standard versions are logically concentric circles: higher versions incorporate all APIs from previous versions. There are no breaking changes between versions.
- **Immutable:** Once shipped, .NET Standard versions are frozen.

There will be no new .NET Standard versions after 2.1. For more information, see [.NET 5+](#) and [.NET Standard](#) later in this article.

Specification

The .NET Standard specification is a standardized set of APIs. The specification is maintained by .NET implementers, specifically Microsoft (includes .NET Framework, .NET

Core, and Mono) and Unity.

Official artifacts

The official specification is a set of .cs files that define the APIs that are part of the standard. The [ref directory](#) in the (now archived) [dotnet/standard repository](#) defines the .NET Standard APIs.

The [NETStandard.Library](#) metapackage ([source](#)) describes the set of libraries that define (in part) one or more .NET Standard versions.

A given component, like `System.Runtime`, describes:

- Part of .NET Standard (just its scope).
- Multiple versions of .NET Standard, for that scope.

Derivative artifacts are provided to enable more convenient reading and to enable certain developer scenarios (for example, using a compiler).

- [API list in markdown](#).
- Reference assemblies, distributed as NuGet packages and referenced by the [NETStandard.Library](#) metapackage.

Package representation

The primary distribution vehicle for the .NET Standard reference assemblies is NuGet packages. Implementations are delivered in a variety of ways, appropriate for each .NET implementation.

NuGet packages target one or more [frameworks](#). .NET Standard packages target the ".NET Standard" framework. You can target the .NET Standard framework using the `netstandard` [compact TFM](#) (for example, `netstandard1.4`). Libraries that are intended to run on multiple implementations of .NET should target this framework. For the broadest set of APIs, target `netstandard2.0` since the number of available APIs more than doubled between .NET Standard 1.6 and 2.0.

The [NETStandard.Library](#) metapackage references the complete set of NuGet packages that define .NET Standard. The most common way to target `netstandard` is by referencing this metapackage. It describes and provides access to the ~40 .NET libraries and associated APIs that define .NET Standard. You can reference additional packages that target `netstandard` to get access to additional APIs.

Versioning

The specification is not singular, but a linearly versioned set of APIs. The first version of the standard establishes a baseline set of APIs. Subsequent versions add APIs and inherit APIs defined by previous versions. There is no established provision for removing APIs from the Standard.

.NET Standard is not specific to any one .NET implementation, nor does it match the versioning scheme of any of those implementations.

As noted earlier, there will be no new .NET Standard versions after 2.1.

Target .NET Standard

You can [build .NET Standard Libraries](#) using a combination of the `netstandard` framework and the `NETStandard.Library` metapackage.

.NET Framework compatibility mode

Starting with .NET Standard 2.0, the .NET Framework compatibility mode was introduced. This compatibility mode allows .NET Standard projects to reference .NET Framework libraries as if they were compiled for .NET Standard. Referencing .NET Framework libraries doesn't work for all projects, such as libraries that use Windows Presentation Foundation (WPF) APIs.

For more information, see [.NET Framework compatibility mode](#).

.NET Standard libraries and Visual Studio

To build .NET Standard libraries in Visual Studio, make sure you have [Visual Studio 2022](#), [Visual Studio 2019](#), or Visual Studio 2017 version 15.3 or later installed on Windows, or [Visual Studio for Mac version 7.1](#) or later installed on macOS.

If you only need to consume .NET Standard 2.0 libraries in your projects, you can also do that in Visual Studio 2015. However, you need NuGet client 3.6 or higher installed. You can download the NuGet client for Visual Studio 2015 from the [NuGet downloads](#) ↗ page.

.NET 5+ and .NET Standard

.NET 5, .NET 6, .NET 7, and .NET 8 are single products with a uniform set of capabilities and APIs that can be used for Windows desktop apps and cross-platform console apps, cloud services, and websites. The .NET 8 [TFMs](#), for example, reflect this broad range of scenarios:

- `net8.0`

This TFM is for code that runs everywhere. With a few exceptions, it includes only technologies that work cross-platform. For .NET 8 code, `net8.0` replaces both `netcoreapp` and `netstandard` TFMs.

- `net8.0-windows`

This is an example of an [OS-specific TFM](#) that add OS-specific functionality to everything that `net8.0` refers to.

When to target `net8.0` vs. `netstandard`

For existing code that targets `netstandard`, there's no need to change the TFM to `net8.0` or a later TFM. .NET 8 implements .NET Standard 2.1 and earlier. The only reason to retarget from .NET Standard to .NET 8+ would be to gain access to more runtime features, language features, or APIs. For example, in order to use C# 9, you need to target .NET 5 or a later version. You can multitarget .NET 8 and .NET Standard to get access to newer features and still have your library available to other .NET implementations.

Here are some guidelines for new code for .NET 5+:

- App components

If you're using libraries to break down an application into several components, we recommend you target `net8.0`. For simplicity, it's best to keep all projects that make up your application on the same version of .NET. Then you can assume the same BCL features everywhere.

- Reusable libraries

If you're building reusable libraries that you plan to ship on NuGet, consider the trade-off between reach and available feature set. .NET Standard 2.0 is the latest version that's supported by .NET Framework, so it gives good reach with a fairly large feature set. We don't recommend targeting .NET Standard 1.x, as you'd limit the available feature set for a minimal increase in reach.

If you don't need to support .NET Framework, you could go with .NET Standard 2.1 or .NET 8. We recommend you skip .NET Standard 2.1 and go straight to .NET 8. Most widely used libraries will multi-target for both .NET Standard 2.0 and .NET 5+. Supporting .NET Standard 2.0 gives you the most reach, while supporting .NET 5+ ensures you can leverage the latest platform features for customers that are already on .NET 5+.

.NET Standard problems

Here are some problems with .NET Standard that help explain why .NET 5 and later versions are the better way to share code across platforms and workloads:

- Slowness to add new APIs

.NET Standard was created as an API set that all .NET implementations would have to support, so there was a review process for proposals to add new APIs. The goal was to standardize only APIs that could be implemented in all current and future .NET platforms. The result was that if a feature missed a particular release, you might have to wait for a couple of years before it got added to a version of the Standard. Then you'd wait even longer for the new version of .NET Standard to be widely supported.

Solution in .NET 5+: When a feature is implemented, it's already available for every .NET 5+ app and library because the code base is shared. And since there's no difference between the API specification and its implementation, you're able to take advantage of new features much quicker than with .NET Standard.

- Complex versioning

The separation of the API specification from its implementations results in complex mapping between API specification versions and implementation versions. This complexity is evident in the table shown earlier in this article and the instructions for how to interpret it.

Solution in .NET 5+: There's no separation between a .NET 5+ API specification and its implementation. The result is a simplified TFM scheme. There's one TFM prefix for all workloads: `net8.0` is used for libraries, console apps, and web apps. The only variation is a [suffix that specifies platform-specific APIs](#) for a particular platform, such as `net8.0-windows`. Thanks to this TFM naming convention, you can easily tell whether a given app can use a given library. No version number equivalents table, like the one for .NET Standard, is needed.

- Platform-unsupported exceptions at run time

.NET Standard exposes platform-specific APIs. Your code might compile without errors and appear to be portable to any platform even if it isn't portable. When it runs on a platform that doesn't have an implementation for a given API, you get run-time errors.

Solution in .NET 5+: The .NET 5+ SDKs include code analyzers that are enabled by default. The platform compatibility analyzer detects unintentional use of APIs that aren't supported on the platforms you intend to run on. For more information, see [Platform compatibility analyzer](#).

.NET Standard not deprecated

.NET Standard is still needed for libraries that can be used by multiple .NET implementations. We recommend you target .NET Standard in the following scenarios:

- Use `netstandard2.0` to share code between .NET Framework and all other implementations of .NET.
- Use `netstandard2.1` to share code between Mono, Xamarin, and .NET Core 3.x.

See also

- [.NET Standard versions \(source\)](#)
- [.NET Standard versions \(interactive UI\)](#)
- [Build a .NET Standard library](#)
- [Cross-platform targeting](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

-  [Open a documentation issue](#)
-  [Provide product feedback](#)

Releases and support for .NET

Article • 10/14/2023

Microsoft ships major releases, minor releases, and servicing updates (patches) for .NET. This article explains release types, servicing updates, SDK feature bands, support periods, and support options.

ⓘ Note

For information about versioning and support for .NET Framework, see [.NET Framework Lifecycle](#).

Release types

Information about the type of each release is encoded in the version number in the form *major.minor.patch*.

For example:

- .NET 6 and .NET 7 are major releases.
- .NET Core 3.1 is the first minor release after the .NET Core 3.0 major release.
- .NET Core 5.0.15 is the fifteenth patch for .NET 5.

For a list of released versions of .NET and information about how often .NET ships, see the [Support Policy](#).

Major releases

Major releases include new features, new public API surface area, and bug fixes.

Examples include .NET 6 and .NET 7. Due to the nature of the changes, these releases are expected to have breaking changes. Major releases install side by side with previous major releases.

Minor releases

Minor releases also include new features, public API surface area, and bug fixes, and may also have breaking changes. An example is .NET Core 3.1. The difference between these and major releases is that the magnitude of the changes is smaller. An application upgrading from .NET Core 3.0 to 3.1 has a smaller jump to move forward. Minor releases install side by side with previous minor releases.

Servicing updates

Servicing updates (patches) ship almost every month, and these updates carry both security and non-security bug fixes. For example, .NET 5.0.8 was the eighth update for .NET 5. When these updates include security fixes, they're released on "patch Tuesday", which is always the second Tuesday of the month. Servicing updates are expected to maintain compatibility. Starting with .NET Core 3.1, servicing updates are upgrades that remove the preceding update. For example, the latest servicing update for 3.1 removes the previous 3.1 update upon successful installation.

Feature bands (SDK only)

Versioning for the .NET SDK works slightly differently from the .NET runtime. To align with new Visual Studio releases, .NET SDK updates sometimes include new features or new versions of components like MSBuild and NuGet. These new features or components may be incompatible with the versions that shipped in previous SDK updates for the same major or minor version.

To differentiate such updates, the .NET SDK uses the concept of feature bands. For example, the first .NET 5 SDK was 5.0.100. This release corresponds to the 5.0.1xx *feature band*. Feature bands are defined in the hundreds groups in the third section of the version number. For example, 5.0.101 and 5.0.201 are versions in two different feature bands while 5.0.101 and 5.0.199 are in the same feature band. When .NET SDK 5.0.101 is installed, .NET SDK 5.1.100 is removed from the machine if it exists. When .NET SDK 5.0.200 is installed on the same machine, .NET SDK 5.0.101 isn't removed.

For more information about the relationship between .NET SDK and Visual Studio versions, see [.NET SDK, MSBuild, and Visual Studio versioning](#).

Runtime roll forward and compatibility

Major and minor updates install side by side with previous versions. An application built to target a specific *major.minor* version continues to use that targeted runtime even if a newer version is installed. The app doesn't automatically roll forward to use a newer *major.minor* version of the runtime unless you opt in for this behavior. An application that was built to target .NET Core 3.0 doesn't automatically start running on .NET Core 3.1. We recommend rebuilding the app and testing against a newer major or minor runtime version before deploying to production. For more information, see [Framework-dependent apps roll forward](#) and [Self-contained deployment runtime roll forward](#).

Servicing updates are treated differently from major and minor releases. An application built to target .NET 7 runs on the 7.0.0 runtime by default. It automatically rolls forward

to use a newer 7.0.1 runtime when that servicing update is installed. This behavior is the default because we want security fixes to be used as soon as they're installed without any other action needed. You can opt out from this default roll forward behavior.

.NET version lifecycles

.NET versions adopt the [modern lifecycle](#) rather than the [fixed lifecycle](#) that was used for .NET Framework releases. Products that adopt a modern lifecycle have a service-like support model, with shorter support periods and more frequent releases.

Release tracks

There are two support tracks for releases:

- *Standard Term Support* (STS) releases

These versions are supported until 6 months after the next major or minor release ships.

Example:

- .NET 5 is an STS release and was released in November 2020. It was supported for 18 months, until May 2022.
- .NET 7 is an STS release and was released in November 2022. It's supported for 18 months, until May 2024.

- *Long Term Support* (LTS) releases

These versions are supported for a minimum of 3 years, or 1 year after the next LTS release ships if that date is later.

Example:

- .NET Core 3.1 is an LTS release and was released in December 2019. It was supported for 3 years, until December 2022.
- .NET 6 is an LTS release and was released in November, 2021. It's supported for 3 years, until November 2024.

Releases alternate between LTS and STS, so it's possible for an earlier release to be supported longer than a later release. For example, .NET Core 3.1 was an LTS release with support through December 2022. The .NET 5 release shipped almost a year later but went out of support earlier, in May 2022.

Servicing updates ship monthly and include both security and non-security (reliability, compatibility, and stability) fixes. Servicing updates are supported until the next

servicing update is released. Servicing updates have runtime roll forward behavior. That means that applications default to running on the latest installed runtime servicing update.

How to choose a release

If you're building a service and expect to continue updating it on a regular basis, then an STS release like the .NET 7 runtime may be your best option to stay up to date with the latest features .NET has to offer.

If you're building a client application that will be distributed to consumers, stability might be more important than access to the latest features. Your application might need to be supported for a certain period before the consumer can upgrade to the next version of the application. In that case, an LTS release like the .NET 6 runtime could be the right option.

 **Note**

We recommend upgrading to the latest SDK version, even if it's an STS release, as it can target all available runtimes.

Support for servicing updates

.NET servicing updates are supported until the next servicing update is released. The release cadence is monthly.

You need to regularly install servicing updates to ensure that your apps are in a secure and supported state. For example, if the latest servicing update for .NET 7 is 7.0.8 and we ship 7.0.9, then 7.0.8 is no longer the latest. The supported servicing level for .NET 7 is then 7.0.9.

For information about the latest servicing updates for each major and minor version, see the [.NET downloads page](#).

End of support

End of support refers to the date after which Microsoft no longer provides fixes, updates, or technical assistance for a product version. Before this date, make sure you have moved to using a supported version. Versions that are out of support no longer

receive security updates that protect your applications and data. For the supported date ranges for each version of .NET, see the [Support Policy](#).

Supported operating systems

.NET can be run on a range of operating systems. Each of these operating systems has a lifecycle defined by its sponsor organization (for example, Microsoft, Red Hat, or Apple). We take these lifecycle schedules into account when adding and removing support for operating system versions.

When an operating system version goes out of support, we stop testing that version and providing support for that version. Users need to move forward to a supported operating system version to get support.

For more information, see the [.NET OS Lifecycle Policy](#).

Get support

You have a choice between Microsoft assisted support and Community support.

Microsoft support

For assisted support, [contact a Microsoft Support Professional](#).

You need to be on a supported servicing level (the latest available servicing update) to be eligible for support. If a system is running .NET 7 and the 7.0.8 servicing update has been released, then 7.0.8 needs to be installed as a first step.

Community support

For community support, see the [Community page](#).

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

more information, see [our contributor guide](#).

Ecma standards

Article • 09/17/2022

The C# Language and the Common Language Infrastructure (CLI) specifications are standardized through [Ecma International®](#). The first editions of these standards were published by Ecma in December 2001.

Subsequent revisions to the standards have been developed by the TC49-TG2 (C#) and TC49-TG3 (CLI) task groups within the Programming Languages Technical Committee ([TC49](#)), and adopted by the Ecma General Assembly and subsequently by ISO/IEC JTC 1 via the ISO Fast-Track process.

Latest standards

The following official Ecma documents are available for [C#](#) and the [CLI](#) ([TR-84](#)):

- The C# Language Standard (version 7): [ECMA-334.pdf](#)
- The Common Language Infrastructure: [ECMA-335.pdf](#).
- Information Derived from the Partition IV XML File: [ECMA TR/84](#) format.

The official ISO/IEC documents are available from the ISO/IEC [Publicly Available Standards](#) page. These links are direct from that page:

- Information technology - Programming languages - C#: [ISO/IEC 23270:2018](#)
- Information technology — Common Language Infrastructure (CLI) Partitions I to VI: [ISO/IEC 23271:2012](#)
- Information technology — Common Language Infrastructure (CLI) — Technical Report on Information Derived from Partition IV XML File: [ISO/IEC TR 23272:2011](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

.NET glossary

Article • 05/04/2023

The primary goal of this glossary is to clarify meanings of selected terms and acronyms that appear frequently in the .NET documentation.

AOT

Ahead-of-time compiler.

Similar to [JIT](#), this compiler also translates [IL](#) to machine code. In contrast to JIT compilation, AOT compilation happens before the application is executed and is usually performed on a different machine. Because AOT tool chains don't compile at run time, they don't have to minimize time spent compiling. That means they can spend more time optimizing. Since the context of AOT is the entire application, the AOT compiler also performs cross-module linking and whole-program analysis, which means that all references are followed and a single executable is produced.

See [CoreRT](#) and [.NET Native](#).

app model

A [workload](#)-specific API. Here are some examples:

- [ASP.NET](#)
- [ASP.NET Web API](#)
- [Entity Framework \(EF\)](#)
- [Windows Presentation Foundation \(WPF\)](#)
- [Windows Communication Foundation \(WCF\)](#)
- [Windows Workflow Foundation \(WF\)](#)
- [Windows Forms \(WinForms\)](#)

ASP.NET

The original ASP.NET implementation that ships with the .NET Framework, also known as [ASP.NET 4.x](#).

Sometimes ASP.NET is an umbrella term that refers to both the original ASP.NET and [ASP.NET Core](#). The meaning that the term carries in any given instance is determined by

context. Refer to ASP.NET 4.x when you want to make it clear that you're not using ASP.NET to mean both implementations.

See [ASP.NET documentation](#).

ASP.NET Core

A cross-platform, high-performance, open-source implementation of ASP.NET.

See [ASP.NET Core documentation](#).

assembly

A `.dll` or `.exe` file that can contain a collection of APIs that can be called by applications or other assemblies.

An assembly may include types such as interfaces, classes, structures, enumerations, and delegates. Assemblies in a project's `bin` folder are sometimes referred to as *binaries*. See also [library](#).

BCL

Base Class Library.

A set of libraries that comprise the `System.*` (and to a limited extent `Microsoft.*`) namespaces. The BCL is a general purpose, lower-level framework that higher-level application frameworks, such as ASP.NET Core, build on.

The source code of the BCL for [.NET 5 \(and .NET Core\) and later versions](#) is contained in the [.NET runtime repository](#). Most of these BCL APIs are also available in .NET Framework, so you can think of this source code as a fork of the .NET Framework BCL source code.

The following terms often refer to the same collection of APIs that BCL refers to:

- [core .NET libraries](#)
- [framework libraries](#)
- [runtime libraries](#)
- [shared framework](#)

CLR

Common Language Runtime.

The exact meaning depends on the context. Common Language Runtime usually refers to the runtime of [.NET Framework](#) or the runtime of [.NET 5 \(and .NET Core\) and later versions](#).

A CLR handles memory allocation and management. A CLR is also a virtual machine that not only executes apps but also generates and compiles code on-the-fly using a [JIT](#) compiler.

The CLR implementation for .NET Framework is Windows only.

The CLR implementation for .NET 5 and later versions (also known as the Core CLR) is built from the same code base as the .NET Framework CLR. Originally, the Core CLR was the runtime of Silverlight and was designed to run on multiple platforms, specifically Windows and OS X. It's still a [cross-platform](#) runtime, now including support for many Linux distributions.

See also [runtime](#).

Core CLR

The Common Language Runtime for [.NET 5 \(and .NET Core\) and later versions](#).

See [CLR](#).

CoreRT

In contrast to the [CLR](#), CoreRT is not a virtual machine, which means it doesn't include the facilities to generate and run code on-the-fly because it doesn't include a [JIT](#). It does, however, include the [GC](#) and the ability for run-time type identification (RTTI) and reflection. However, its type system is designed so that metadata for reflection isn't required. Not requiring metadata enables having an [AOT](#) tool chain that can link away superfluous metadata and (more importantly) identify code that the app doesn't use. CoreRT is in development.

See [Intro to CoreRT](#) and [.NET Runtime Lab](#).

cross-platform

The ability to develop and execute an application that can be used on multiple different operating systems, such as Linux, Windows, and iOS, without having to rewrite

specifically for each one. This enables code reuse and consistency between applications on different platforms.

See [platform](#).

ecosystem

All of the runtime software, development tools, and community resources that are used to build and run applications for a given technology.

The term ".NET ecosystem" differs from similar terms such as ".NET stack" in its inclusion of third-party apps and libraries. Here's an example in a sentence:

- "The motivation behind [.NET Standard](#) was to establish greater uniformity in the .NET ecosystem."

framework

In general, a comprehensive collection of APIs that facilitates development and deployment of applications that are based on a particular technology. In this general sense, ASP.NET Core and Windows Forms are examples of application frameworks. The words **framework** and [library](#) are often used synonymously.

The word "framework" has a different meaning in the following terms:

- [framework libraries](#)
- [.NET Framework](#)
- [shared framework](#)
- [target framework](#)
- [TFM \(target framework moniker\)](#)
- [framework-dependent app](#)

Sometimes "framework" refers to an [implementation of .NET](#). For example, an article may call .NET 5+ a framework.

framework libraries

Meaning depends on context. May refer to the framework libraries for [.NET 5 \(and .NET Core\) and later versions](#), in which case it refers to the same libraries that [BCL](#) refers to. It may also refer to the [ASP.NET Core](#) framework libraries, which build on the BCL and provide additional APIs for web apps.

GC

Garbage collector.

The garbage collector is an implementation of automatic memory management. The GC frees memory occupied by objects that are no longer in use.

See [Garbage Collection](#).

IL

Intermediate language.

Higher-level .NET languages, such as C#, compile down to a hardware-agnostic instruction set, which is called Intermediate Language (IL). IL is sometimes referred to as MSIL (Microsoft IL) or CIL (Common IL).

JIT

Just-in-time compiler.

Similar to [AOT](#), this compiler translates [IL](#) to machine code that the processor understands. Unlike AOT, JIT compilation happens on demand and is performed on the same machine that the code needs to run on. Since JIT compilation occurs during execution of the application, the compile time is part of the run time. Thus, JIT compilers have to balance time spent optimizing code against the savings that the resulting code can produce. But a JIT knows the actual hardware and can free developers from having to ship different implementations.

implementation of .NET

An implementation of .NET includes:

- One or more runtimes. Examples: [CLR](#), [CoreRT](#).
- A class library that implements a version of .NET Standard and may include additional APIs. Examples: the [BCLs](#) for [.NET Framework](#) and [.NET 5 \(and .NET Core and later versions\)](#).
- Optionally, one or more application frameworks. Examples: [ASP.NET](#), Windows Forms, and WPF are included in .NET Framework and .NET 5+.
- Optionally, development tools. Some development tools are shared among multiple implementations.

Examples of .NET implementations:

- [.NET Framework](#)
- [.NET 5 \(and .NET Core\) and later versions](#)
- [Universal Windows Platform \(UWP\)](#)
- [Mono](#)

For more information, see [.NET implementations](#).

library

A collection of APIs that can be called by apps or other libraries. A .NET library is composed of one or more [assemblies](#).

The words **library** and **framework** are often used synonymously.

Mono

An open source, [cross-platform .NET implementation](#) that's mainly used when a small runtime is required. It is the runtime that powers Xamarin applications on Android, Mac, iOS, tvOS, and watchOS and is focused primarily on apps that require a small footprint.

It supports all of the currently published .NET Standard versions.

Historically, Mono implemented the larger API of the .NET Framework and emulated some of the most popular capabilities on Unix. It is sometimes used to run .NET applications that rely on those capabilities on Unix.

Mono is typically used with a [just-in-time compiler](#), but it also features a full [static compiler \(ahead-of-time compilation\)](#) that is used on platforms like iOS.

For more information, see the [Mono documentation](#).

Native AOT

A deployment mode where the app is self-contained and has been [ahead-of-time](#) compiled to native code at the time of publish. Native AOT apps don't use a [JIT](#) compiler at run time. They can run on machines that don't have the .NET runtime installed.

For more information, see [Native AOT deployment](#).

.NET

- In general, **.NET** is the umbrella term for [.NET Standard](#) and all [.NET implementations](#) and workloads.
- More specifically, **.NET** refers to the implementation of .NET that is recommended for all new development: [.NET 5 \(and .NET Core\) and later versions](#).

For example, the first meaning is intended in phrases such as "implementations of .NET" or "the .NET development platform." The second meaning is intended in names such as [.NET SDK](#) and [.NET CLI](#).

.NET is always fully capitalized, never ".Net".

See [.NET documentation](#)

.NET 5+

The plus sign after a version number means "and later versions." See [.NET 5 and later versions](#).

.NET 5 and later versions

A cross-platform, high-performance, open-source implementation of .NET. Also referred to as **.NET 5+**. Includes a Common Language Runtime ([CLR](#)), an [AOT](#) runtime ([CoreRT](#), in development), a Base Class Library ([BCL](#)), and the [.NET SDK](#).

Earlier versions of this .NET implementation are known as [.NET Core](#). .NET 5 is the next version following .NET Core 3.1. Version 4 was skipped to avoid confusing this newer implementation of .NET with the older implementation that is known as [.NET Framework](#). The current version of .NET Framework is 4.8.

See [.NET documentation](#).

.NET CLI

A cross-platform toolchain for developing applications and libraries for [.NET 5 \(and .NET Core\) and later versions](#). Also known as the .NET Core CLI.

See [.NET CLI](#).

.NET Core

See [.NET 5 and later versions](#).

.NET Framework

An [implementation of .NET](#) that runs only on Windows. Includes the Common Language Runtime ([CLR](#)), the Base Class Library ([BCL](#)), and application framework libraries such as [ASP.NET](#), Windows Forms, and WPF.

See [.NET Framework Guide](#).

.NET Native

A compiler tool chain that produces native code ahead-of-time ([AOT](#)), as opposed to just-in-time ([JIT](#)).

Compilation happens on the developer's machine similar to the way a C++ compiler and linker works. It removes unused code and spends more time optimizing it. It extracts code from libraries and merges them into the executable. The result is a single module that represents the entire app.

UWP is the application framework supported by .NET Native.

See [.NET Native documentation](#).

.NET SDK

A set of libraries and tools that allow developers to create .NET applications and libraries for [.NET 5 \(and .NET Core\) and later versions](#). Also known as the .NET Core SDK.

Includes the [.NET CLI](#) for building apps, .NET libraries and runtime for building and running apps, and the dotnet executable (*dotnet.exe*) that runs CLI commands and runs applications.

See [.NET SDK Overview](#).

.NET Standard

A formal specification of .NET APIs that are available in each [.NET implementation](#).

The .NET Standard specification is sometimes called a library. Because a library includes API implementations, not only specifications (interfaces), it's misleading to call .NET Standard a "library."

See [.NET Standard](#).

NGen

Native (image) generation.

You can think of this technology as a persistent [JIT](#) compiler. It usually compiles code on the machine where the code is executed, but compilation typically occurs at install time.

package

A NuGet package—or just a package—is a *.zip* file with one or more assemblies of the same name along with additional metadata such as the author name.

The *.zip* file has a *.nupkg* extension and may contain assets, such as *.dll* files and *.xml* files, for use with multiple [target frameworks](#) and versions. When installed in an app or library, the appropriate assets are selected based on the target framework specified by the app or library. The assets that define the interface are in the *ref* folder, and the assets that define the implementation are in the *lib* folder.

platform

An operating system and the hardware it runs on, such as Windows, macOS, Linux, iOS, and Android.

Here are examples of usage in sentences:

- ".NET Core is a cross-platform implementation of .NET."
- "PCL profiles represent Microsoft platforms, while .NET Standard is agnostic to platform."

Legacy .NET documentation sometimes uses ".NET platform" to mean either an [implementation of .NET](#) or the [.NET stack](#) including all implementations. Both of these usages tend to get confused with the primary (OS/hardware) meaning, so we try to avoid these usages.

"Platform" has a different meaning in the phrase "developer platform," which refers to software that provides tools and libraries for building and running apps. .NET is a cross-platform, open-source developer platform for building many different types of applications.

POCO

A POCO—or a plain old class/CLR object—is a .NET data structure that contains only public properties or fields. A POCO shouldn't contain any other members, such as:

- methods
- events
- delegates

These objects are used primarily as data transfer objects (DTOs). A pure POCO will not inherit another object, or implement an interface. It's common for POCOs to be used with serialization.

runtime

In general, the execution environment for a managed program. The OS is part of the runtime environment but is not part of the .NET runtime. Here are some examples of .NET runtimes in this sense of the word:

- Common Language Runtime (CLR)
- .NET Native (for UWP)
- Mono runtime

The word "runtime" has a different meaning in some contexts:

- *.NET runtime* on the [.NET 5 download page](#).

You can download the *.NET runtime* or other runtimes, such as the *ASP.NET Core runtime*. A *runtime* in this usage is the set of components that must be installed on a machine to run a [framework-dependent](#) app on the machine. The .NET runtime includes the [CLR](#) and the [.NET shared framework](#), which provides the [BCL](#).

- *.NET runtime libraries*

Refers to the same libraries that [BCL](#) refers to. However, other runtimes, such as the ASP.NET Core runtime, have different [shared frameworks](#), with additional libraries that build on the BCL.

- [Runtime Identifier \(RID\)](#).

Runtime here means the OS platform and CPU architecture that a .NET app runs on, for example: `linux-x64`.

- Sometimes "runtime" is used in the sense of an [implementation of .NET](#), as in the following examples:

- "The various .NET runtimes implement specific versions of .NET Standard. ... Each .NET runtime version advertises the highest .NET Standard version it supports ..."
- "Libraries that are intended to run on multiple runtimes should target this framework." (referring to .NET Standard)

shared framework

Meaning depends on context. The *.NET shared framework* refers to the libraries included in the [.NET runtime](#). In this case, the *shared framework* for [.NET 5](#) (and [.NET Core](#)) and [later versions](#) refers to the same libraries that [BCL](#) refers to.

There are other shared frameworks. The *ASP.NET Core shared framework* refers to the libraries included in the [ASP.NET Core runtime](#), which includes the BCL plus additional APIs for use by web apps.

For [framework-dependent apps](#), the shared framework consists of libraries that are contained in assemblies installed in a folder on the machine that runs the app. For [self-contained apps](#), the shared framework assemblies are included with the app.

For more information, see [Deep-dive into .NET Core primitives, part 2: the shared framework](#).

stack

A set of programming technologies that are used together to build and run applications.

"The .NET stack" refers to .NET Standard and all .NET implementations. The phrase "a .NET stack" may refer to one implementation of .NET.

target framework

The collection of APIs that a .NET app or library relies on.

An app or library can target a version of [.NET Standard](#) (for example, .NET Standard 2.0), which is a specification for a standardized set of APIs across all [.NET implementations](#). An app or library can also target a version of a specific .NET implementation, in which case it gets access to implementation-specific APIs. For example, an app that targets Xamarin.iOS gets access to Xamarin-provided iOS API wrappers.

For some target frameworks (for example, [.NET Framework](#)) the available APIs are defined by the assemblies that a .NET implementation installs on a system, which may

include application framework APIs (for example, ASP.NET, WinForms). For package-based target frameworks, the framework APIs are defined by the packages installed in the app or library.

See [Target Frameworks](#).

TFM

Target framework moniker.

A standardized token format for specifying the [target framework](#) of a .NET app or library. Target frameworks are typically referenced by a short name, such as `net462`. Long-form TFMs (such as `.NETFramework,Version=4.6.2`) exist but are not generally used to specify a target framework.

See [Target Frameworks](#).

UWP

Universal Windows Platform.

An [implementation of .NET](#) that is used for building touch-enabled Windows applications and software for the Internet of Things (IoT). It's designed to unify the different types of devices that you may want to target, including PCs, tablets, phones, and even the Xbox. UWP provides many services, such as a centralized app store, an execution environment (AppContainer), and a set of Windows APIs to use instead of Win32 (WinRT). Apps can be written in C++, C#, Visual Basic, and JavaScript. When using C# and Visual Basic, the .NET APIs are provided by [.NET 5 \(and .NET Core\) and later versions](#).

workload

A type of app someone is building. More generic than [app model](#). For example, at the top of every .NET documentation page, including this one, is a drop-down list for **Workloads**, which lets you switch to documentation for **Web, Mobile, Cloud, Desktop, and Machine Learning & Data**.

In some contexts, *workload* refers to a collection of Visual Studio features that you can choose to install to support a particular type of app. For an example, see [Select a workload](#).

See also

- [.NET fundamentals](#)
- [.NET Framework Guide](#)
- [ASP.NET Overview](#)
- [ASP.NET Core Overview](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

C# console app template generates top-level statements

Article • 02/15/2024

Starting with .NET 6, the project template for new C# console apps generates the following code in the *Program.cs* file:

C#

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

The new output uses recent C# features that simplify the code you need to write for a program. For .NET 5 and earlier versions, the console app template generates the following code:

C#

```
using System;

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

In the preceding code, the actual namespace depends on the project name.

These two forms represent the same program. Both are valid with C# 10.0. When you use the newer version, you only need to write the body of the `Main` method. The compiler generates a `Program` class with an entry point method and places all your top level statements in that method. The name of the generated method isn't `Main`, it's an implementation detail that your code can't reference directly. You don't need to include the other program elements, the compiler generates them for you. You can learn more about the code the compiler generates when you use top level statements in the article on [top level statements](#) in the C# Guide's fundamentals section.

You have two options to work with tutorials that haven't been updated to use .NET 6+ templates:

- Use the new program style, adding new top-level statements as you add features.
- Convert the new program style to the older style, with a `Program` class and a `Main` method.

If you want to use the old templates, see [Use the old program style](#) later in this article.

Use the new program style

The features that make the new program simpler are *top-level statements*, *global using directives*, and *implicit using directives*.

The term *top-level statements* means the compiler generates the class and method elements for your main program. The compiler generated class and entry point method are declared in the global namespace. You can look at the code for the new application and imagine that it contains the statements inside the `Main` method generated by earlier templates, but in the global namespace.

You can add more statements to the program, just like you can add more statements to your `Main` method in the traditional style. You can [access args \(command-line arguments\)](#), [use await](#), and [set the exit code](#). You can even add functions. They're created as local functions nested inside the generated entry point method. Local functions can't include any access modifiers (for example, `public` or `protected`).

Both top-level statements and *implicit using directives* simplify the code that makes up your application. To follow an existing tutorial, add any new statements to the `Program.cs` file generated by the template. You can imagine that the statements you write are between the open and closing braces in the `Main` method in the instructions of the tutorial.

If you'd prefer to use the older format, you can copy the code from the second example in this article, and continue the tutorial as before.

You can learn more about top-level statements in the tutorial exploration on [top-level statements](#).

Implicit using directives

The term *implicit using directives* means the compiler automatically adds a set of [using directives](#) based on the project type. For console applications, the following directives

are implicitly included in the application:

- `using System;`
- `using System.IO;`
- `using System.Collections.Generic;`
- `using System.Linq;`
- `using System.Net.Http;`
- `using System.Threading;`
- `using System.Threading.Tasks;`

Other application types include more namespaces that are common for those application types.

If you need `using` directives that aren't implicitly included, you can add them to the `.cs` file that contains top-level statements or to other `.cs` files. For `using` directives that you need in all of the `.cs` files in an application, use [global using directives](#).

Disable implicit `using` directives

If you want to remove this behavior and manually control all namespaces in your project, add `<ImplicitUsings>disable</ImplicitUsings>` to your project file in the `<PropertyGroup>` element, as shown in the following example:

XML

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

  <PropertyGroup>
    ...
    <ImplicitUsings>disable</ImplicitUsings>
  </PropertyGroup>

</Project>
```

Global `using` directives

A *global using directive* imports a namespace for your whole application instead of a single file. These global directives can be added either by adding a `<Using>` item to the project file, or by adding the `global using` directive to a code file.

You can also add a `<Using>` item with a `Remove` attribute to your project file to remove a specific [implicit using directive](#). For example, if the implicit `using` directives feature is

turned on with `<ImplicitUsings>enable</ImplicitUsings>`, adding the following `<using>` item removes the `System.Net.Http` namespace from those that are implicitly imported:

XML

```
<ItemGroup>
  <Using Remove="System.Net.Http" />
</ItemGroup>
```

Use the old program style

Starting with .NET SDK 6.0.300, the `console` template has a `--use-program-main` option. Use it to create a console project that doesn't use top-level statements and has a `Main` method.

.NET CLI

```
dotnet new console --use-program-main
```

The generated `Program.cs` is as follows:

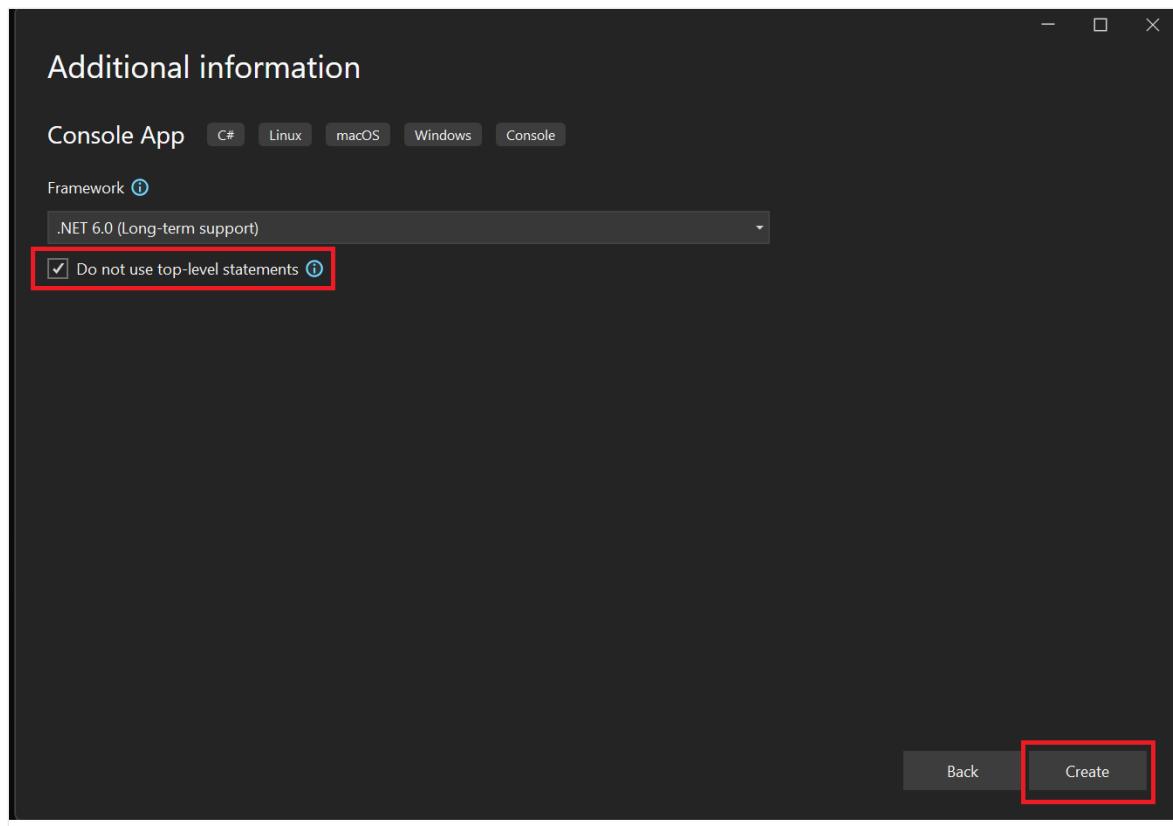
C#

```
namespace MyProject;

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

Use the old program style in Visual Studio

1. When you create a new project, the setup steps will navigate to the **Additional information** setup page. On this page, select the **Do not use top-level statements** check box.



2. After your project is created, the `Program.cs` content is as follows:

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

ⓘ Note

Visual Studio preserves the value for the options next time you create the project based on the same template, so by default when creating Console App project next time the "Do not use top-level statements" check box will be checked. The content of the `Program.cs` file might be different to match the code style defined in the global Visual Studio text editor settings or the `EditorConfig` file.

For more information, see [Create portable, custom editor settings with EditorConfig](#) and [Options, Text Editor, C#, Advanced](#).

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Create a .NET console application using Visual Studio

Article • 08/25/2023

This tutorial shows how to create and run a .NET console application in Visual Studio 2022.

Prerequisites

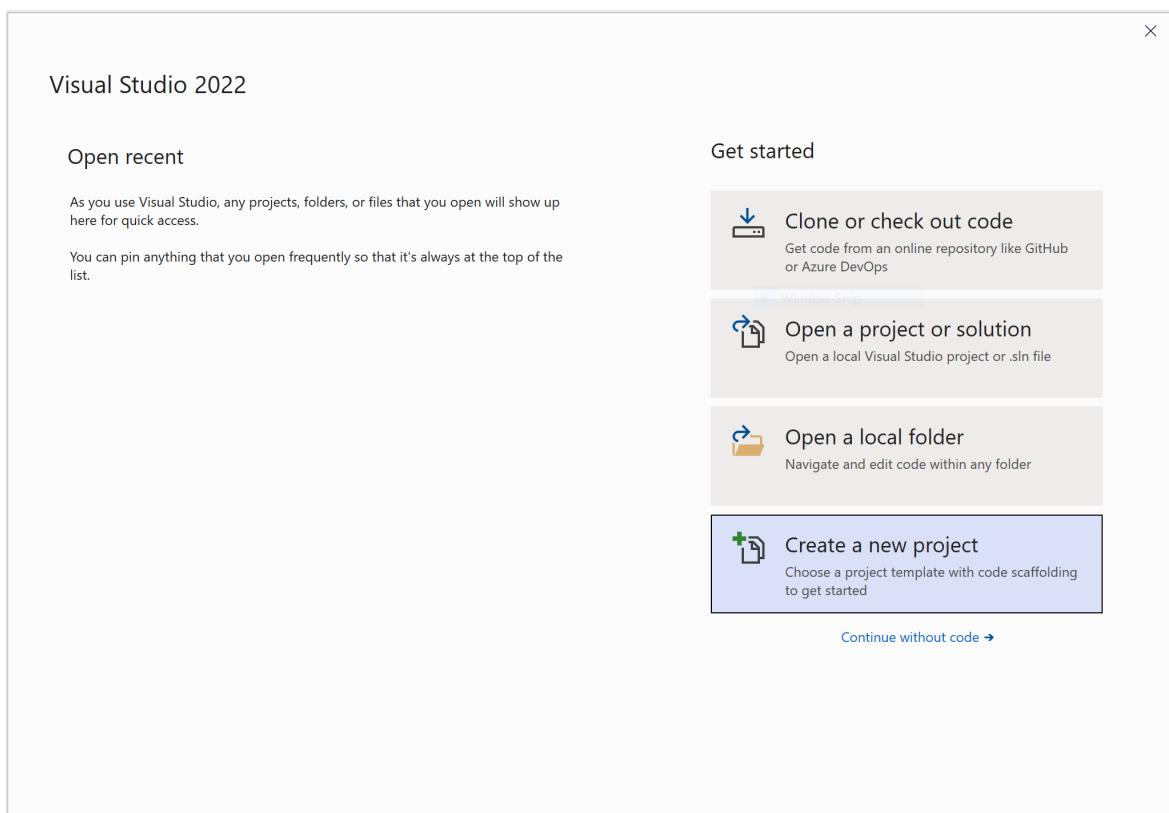
- [Visual Studio 2022 Preview](#) with the **.NET desktop development** workload installed. The .NET 8 SDK is automatically installed when you select this workload.

For more information, see [Install the .NET SDK with Visual Studio](#).

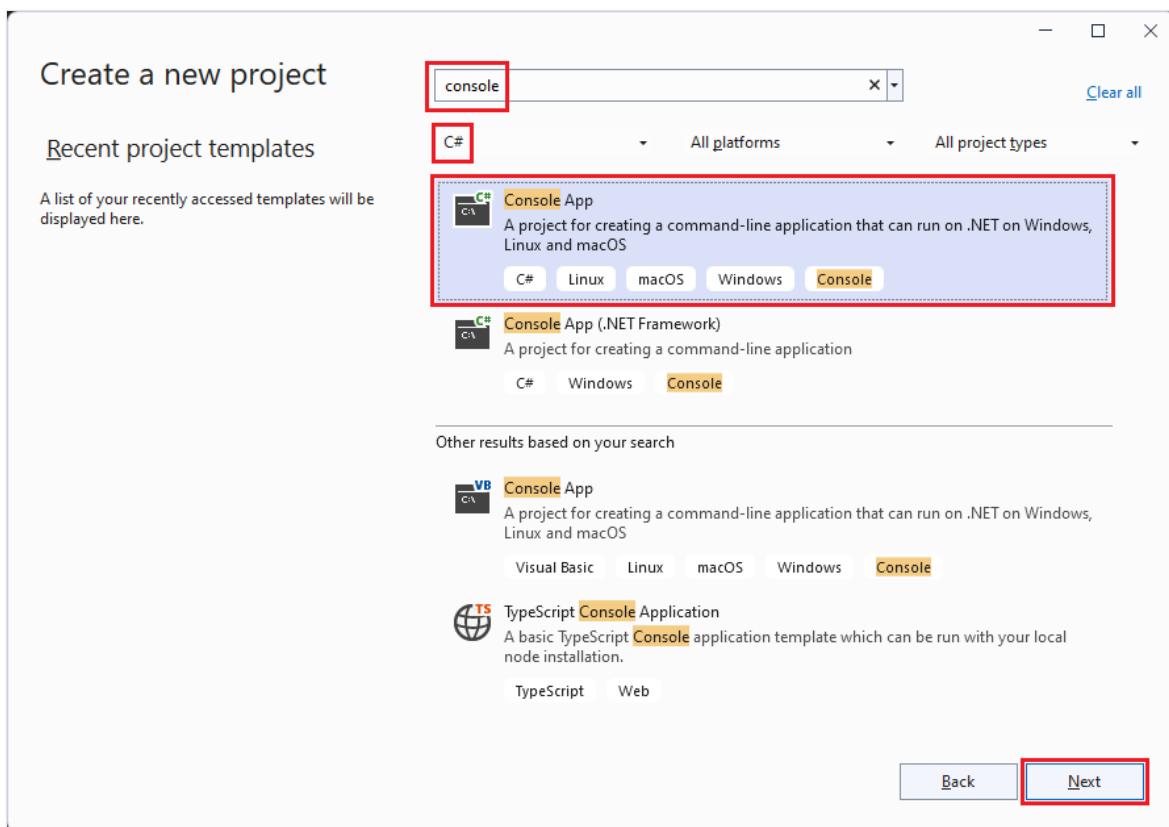
Create the app

Create a .NET console app project named "HelloWorld".

1. Start Visual Studio 2022.
2. On the start page, choose **Create a new project**.



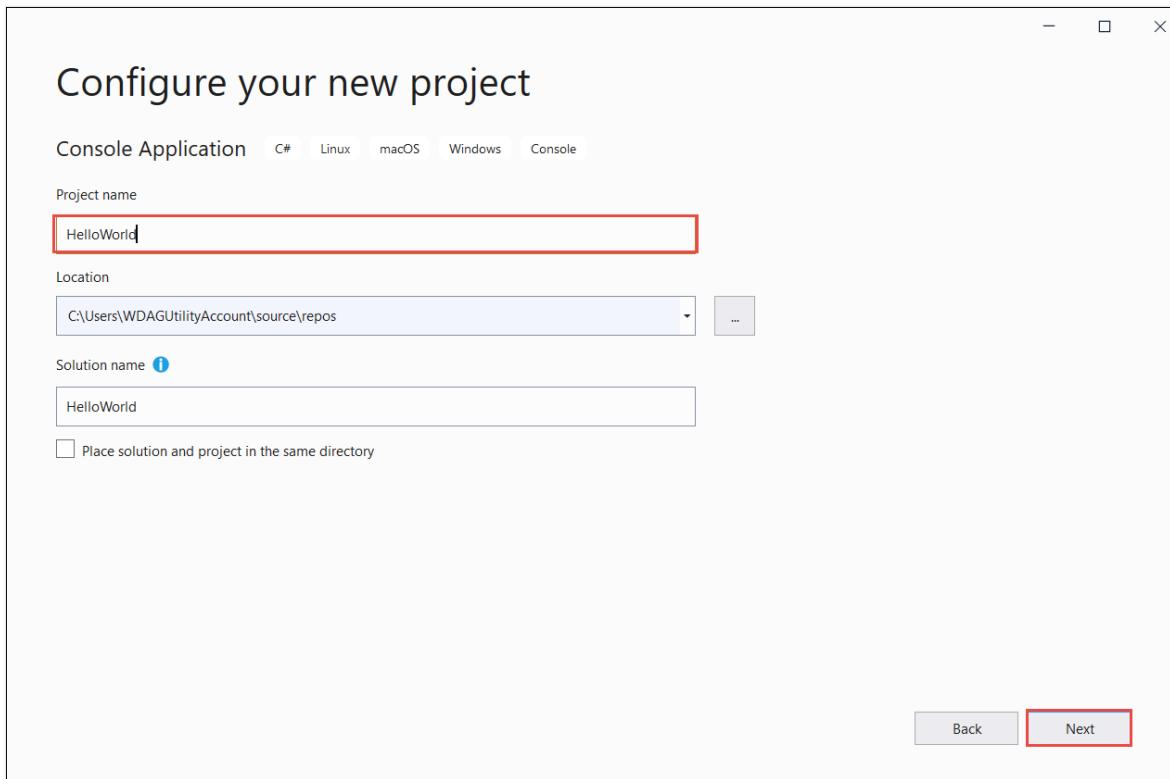
3. On the **Create a new project** page, enter **console** in the search box. Next, choose **C#** or **Visual Basic** from the language list, and then choose **All platforms** from the platform list. Choose the **Console App** template, and then choose **Next**.



💡 Tip

If you don't see the .NET templates, you're probably missing the required workload. Under the **Not finding what you're looking for?** message, choose the **Install more tools and features** link. The Visual Studio Installer opens. Make sure you have the **.NET desktop development** workload installed.

4. In the **Configure your new project** dialog, enter **HelloWorld** in the **Project name** box. Then choose **Next**.



5. In the Additional information dialog:

- Select **.NET 8 (Preview)**.
- Select **Do not use top-level statements**.
- Select **Create**.

The template creates a simple application that displays "Hello, World!" in the console window. The code is in the *Program.cs* or *Program.vb* file:

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

If the language you want to use is not shown, change the language selector at the top of the page.

The code 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

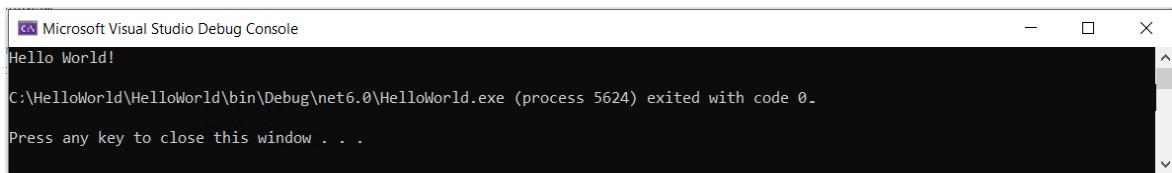
arguments supplied when the application is launched are available in the `args` array.

C# has a feature named [top-level statements](#) that lets you omit the `Program` class and the `Main` method. This tutorial doesn't use this feature. Whether you use it in your programs is a matter of style preference.

Run the app

1. Press `Ctrl + F5` to run the program without debugging.

A console window opens with the text "Hello, World!" printed on the screen. (Or "Hello World!" without a comma in the Visual Basic project template.)



2. Press any key to close the console window.

Enhance the app

Enhance the application to prompt the user for their name and display it along with the date and time.

1. In `Program.cs` or `Program.vb`, replace the contents of the `Main` method, which is the line that calls `Console.WriteLine`, with the following code:

```
C#  
  
Console.WriteLine("What is your name?");  
var name = Console.ReadLine();  
var currentDate = DateTime.Now;  
Console.WriteLine($"{Environment.NewLine}Hello, {name}, on  
{currentDate:d} at {currentDate:t}!");  
Console.WriteLine($"{Environment.NewLine}Press any key to exit...");  
Console.ReadKey(true);
```

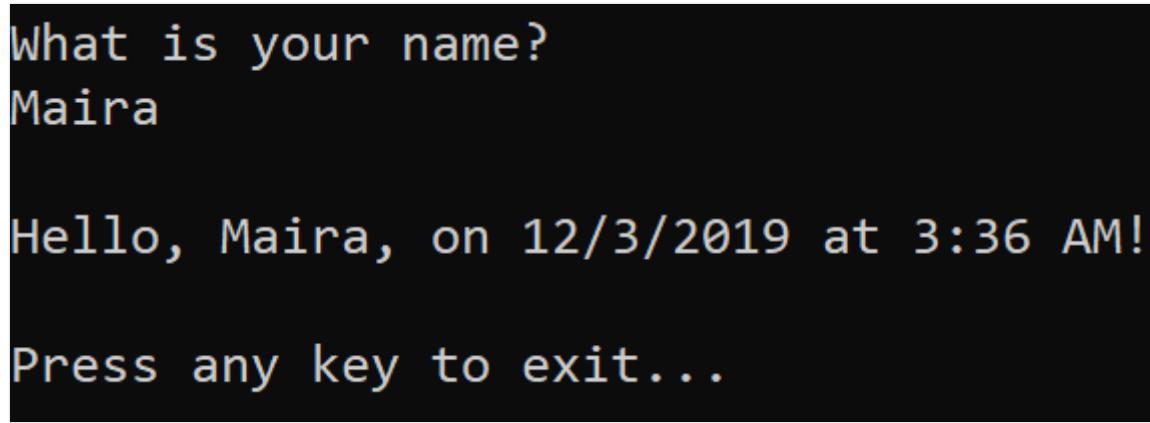
This code displays a prompt in the console window and waits until the user enters a string followed by the `Enter` key. It stores this string in a variable named `name`. It also retrieves the value of the `DateTime.Now` property, which contains the current local time, and assigns it to a variable named `currentDate`. And it displays these

values in the console window. Finally, it displays a prompt in the console window and calls the [Console.ReadKey\(Boolean\)](#) method to wait for user input.

[Environment.NewLine](#) is a platform-independent and language-independent way to represent a line break. Alternatives are `\n` in C# and `vbCrLf` in Visual Basic.

The dollar sign (\$) in front of a string lets you put expressions such as variable names in curly braces in the string. The expression value is inserted into the string in place of the expression. This syntax is referred to as [interpolated strings](#).

2. Press `Ctrl + F5` to run the program without debugging.
3. Respond to the prompt by entering a name and pressing the `Enter` key.



```
What is your name?  
Maira  
  
Hello, Maira, on 12/3/2019 at 3:36 AM!  
  
Press any key to exit...
```

4. Press any key to close the console window.

Additional resources

- [Standard-term support \(STS\) releases and long-term support \(LTS\) releases.](#)

Next steps

In this tutorial, you created a .NET console application. In the next tutorial, you debug the app.

[Debug a .NET console application using Visual Studio](#)

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you

.NET

[.NET feedback](#)

.NET is an open source project.
Select a link to provide feedback:

can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Debug a .NET console application using Visual Studio

Article • 08/25/2023

This tutorial introduces the debugging tools available in Visual Studio.

ⓘ Important

All of the keyboard shortcuts are based on the defaults from Visual Studio. Your keyboard shortcuts may vary, for more information see [Keyboard shortcuts in Visual Studio](#).

Prerequisites

- This tutorial works with the console app that you create in [Create a .NET console application using Visual Studio](#).

Use Debug build configuration

Debug and *Release* are Visual Studio's built-in build configurations. You use the *Debug* build configuration for debugging and the *Release* configuration for the final release distribution.

In the *Debug* configuration, a program compiles with full symbolic debug information and no optimization. Optimization complicates debugging, because the relationship between source code and generated instructions is more complex. The *Release* configuration of a program has no symbolic debug information and is fully optimized.

By default, Visual Studio uses the *Debug* build configuration, so you don't need to change it before debugging.

1. Start Visual Studio.
2. Open the project that you created in [Create a .NET console application using Visual Studio](#).

The current build configuration is shown on the toolbar. The following toolbar image shows that Visual Studio is configured to compile the *Debug* version of the app:

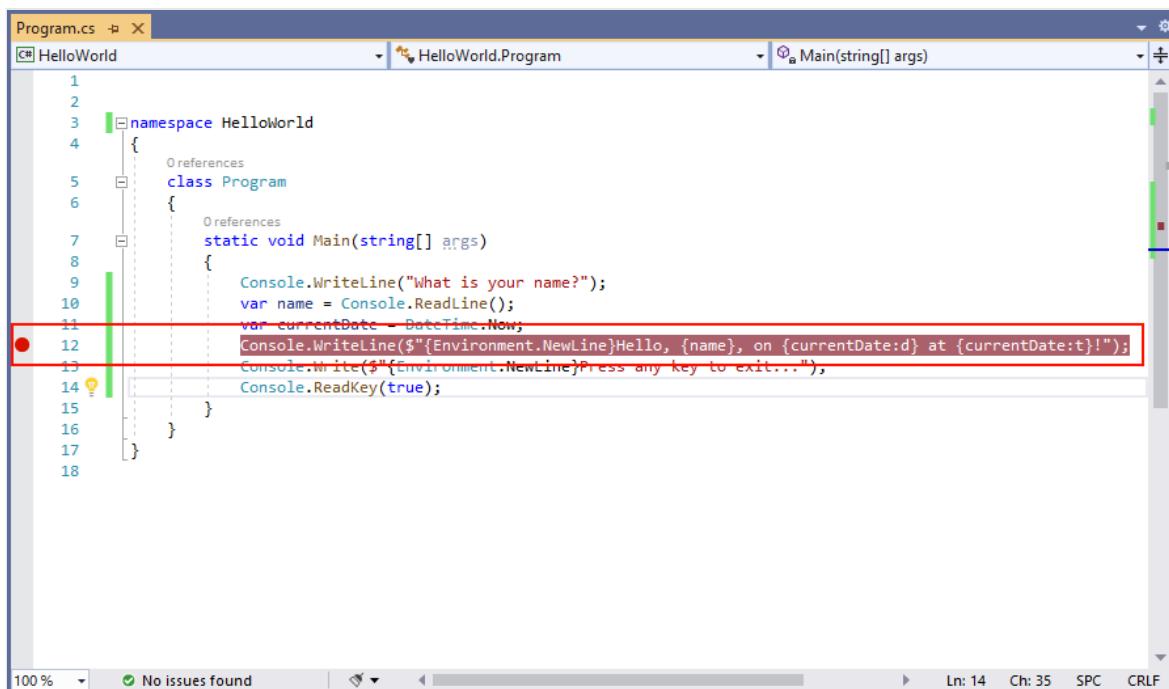


Set a breakpoint

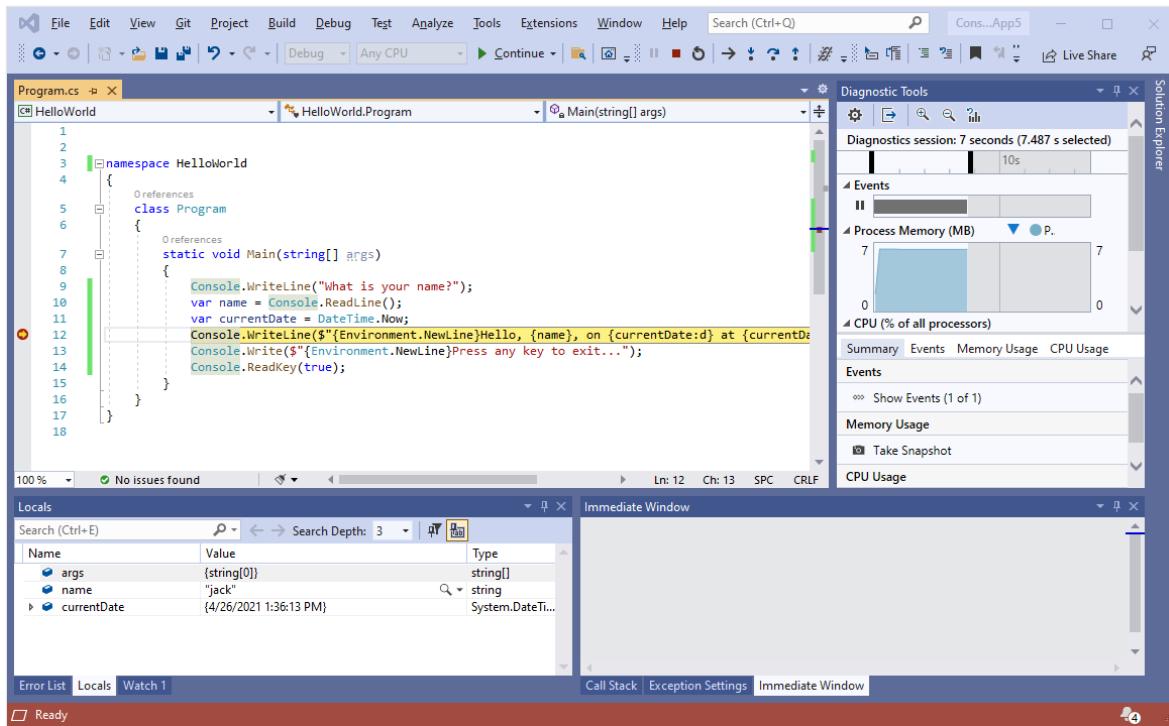
A *breakpoint* temporarily interrupts the execution of the application before the line with the breakpoint is executed.

1. Set a *breakpoint* on the line that displays the name, date, and time, by clicking in the left margin of the code window on that line. The left margin is to the left of the line numbers. Other ways to set a breakpoint are by placing the cursor in the line of code and then pressing **F9** or choosing **Debug > Toggle Breakpoint** from the menu bar.

As the following image shows, Visual Studio indicates the line on which the breakpoint is set by highlighting it and displaying a red dot in the left margin.



2. Press **F5** to run the program in Debug mode. Another way to start debugging is by choosing **Debug > Start Debugging** from the menu.
3. Enter a string in the console window when the program prompts for a name, and then press **Enter**.
4. Program execution stops when it reaches the breakpoint and before the `Console.WriteLine` method executes. The **Locals** window displays the values of variables that are defined in the currently executing method.

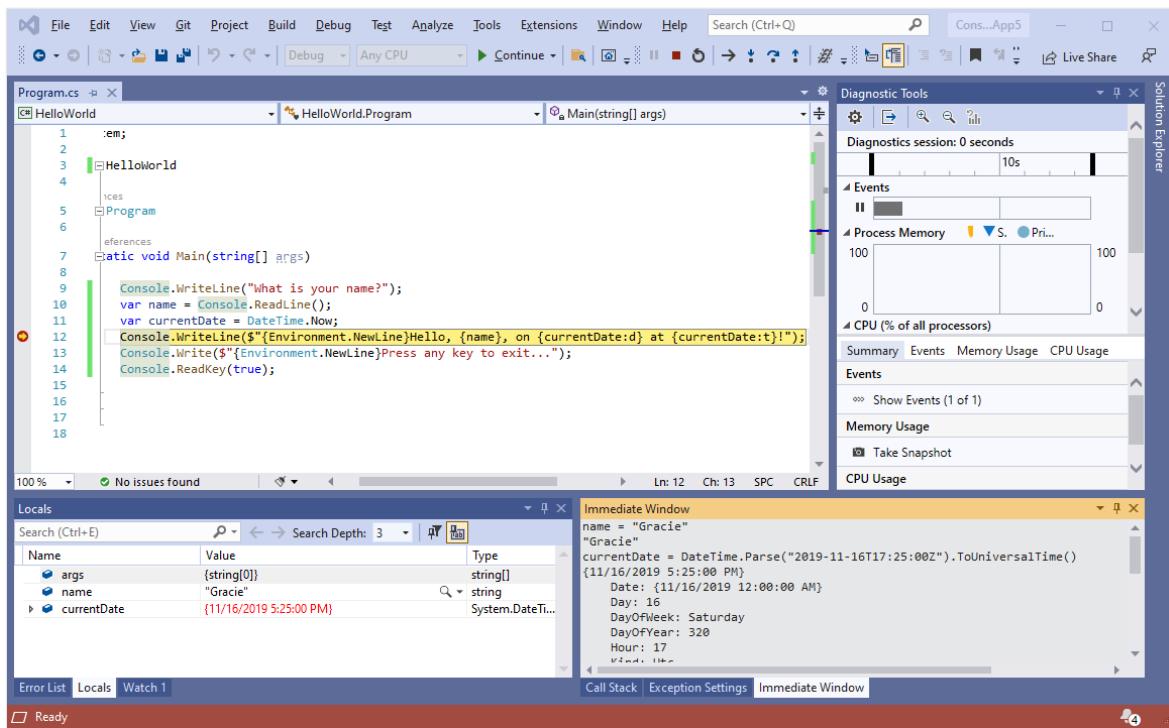


Use the Immediate window

The **Immediate** window lets you interact with the application you're debugging. You can interactively change the value of variables to see how it affects your program.

1. If the **Immediate** window is not visible, display it by choosing **Debug > Windows > Immediate**.
2. Enter `name = "Gracie"` in the **Immediate** window and press the **Enter** key.
3. Enter `currentDate = DateTime.Parse("2019-11-16T17:25:00Z").ToUniversalTime()` 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 values of the variables are updated in the **Locals** window.



4. Press **F5** to continue program execution. Another way to continue is by choosing **Debug > Continue** from the menu.

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 11/16/2019 at 5:25 PM!

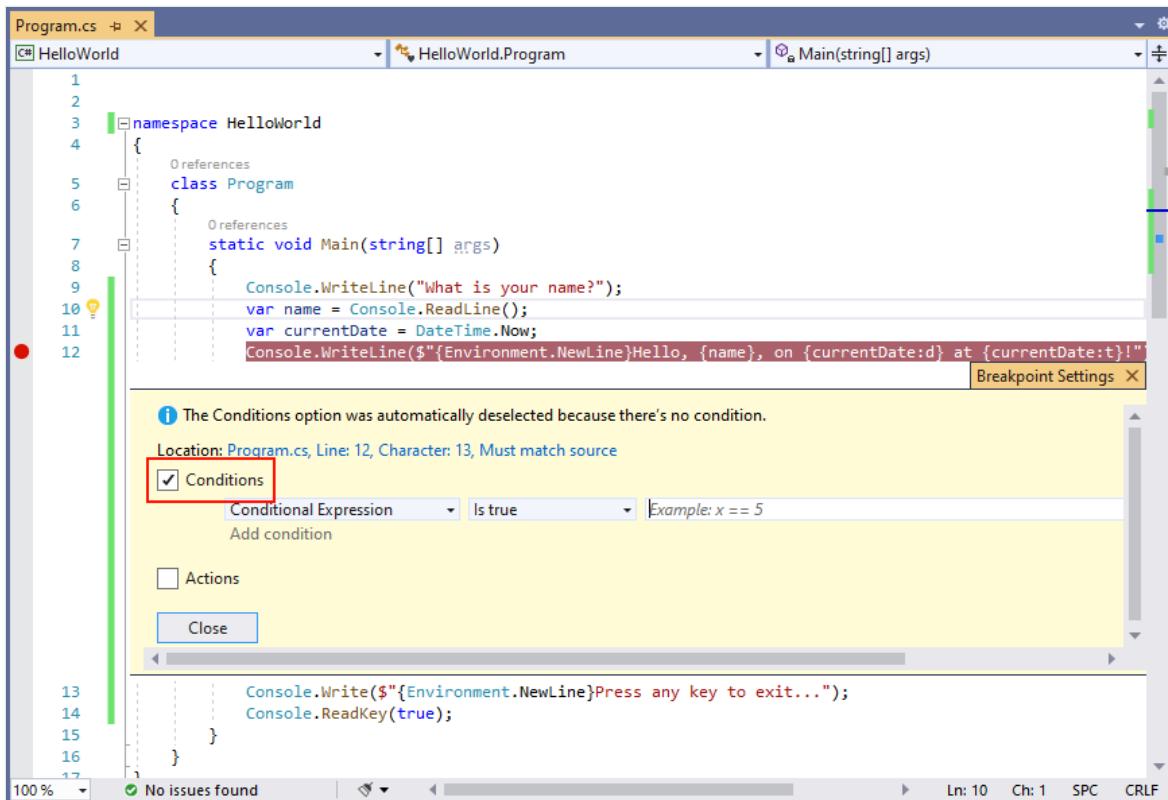
Press any key to exit...
```

5. Press any key to exit the application and stop debugging.

Set a conditional breakpoint

The 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 called a *conditional breakpoint*.

1. Right-click on the red dot that represents the breakpoint. In the context menu, select **Conditions** to open the **Breakpoint Settings** dialog. Select the box for **Conditions** if it's not already selected.



2. For the **Conditional Expression**, enter the following code in the field that shows example code that tests if `x` is 5.

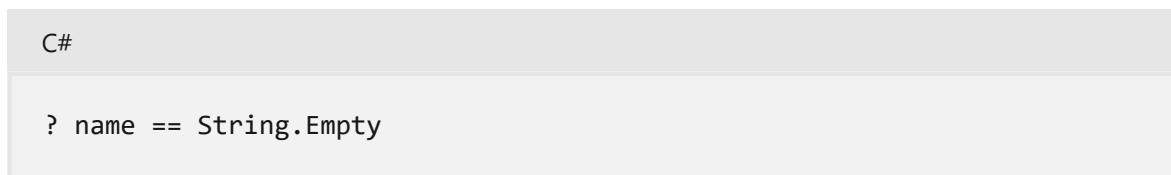
```
C#  
  
string.IsNullOrEmpty(name)
```

Each time the breakpoint is hit, the debugger calls the `String.IsNullOrEmpty(name)` method, and it breaks on this line only if the method call returns `true`.

Instead of a conditional expression, you can specify a *hit count*, which interrupts program execution before a statement is executed a specified number of times. Another option is to specify a *filter condition*, which interrupts program execution based on such attributes as a thread identifier, process name, or thread name.

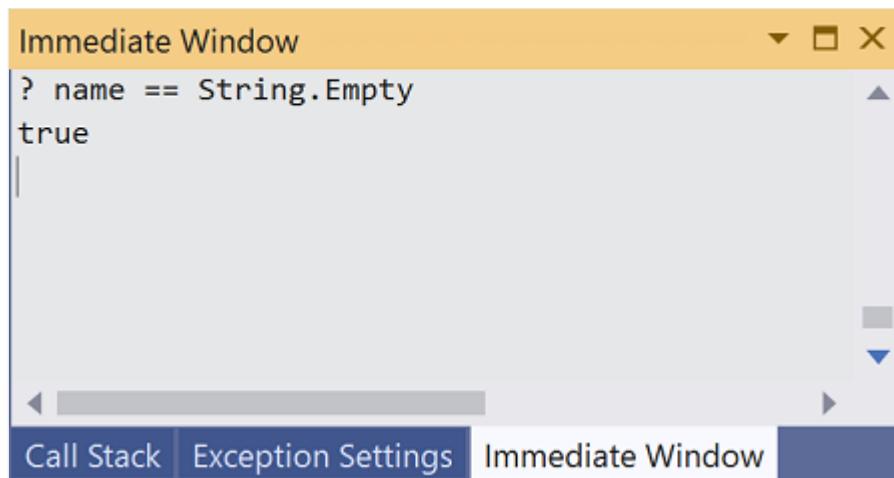
3. Select **Close** to close the dialog.
4. Start the program with debugging by pressing `F5`.
5. In the console window, press the `Enter` key when prompted to enter your name.
6. Because the condition you specified (`name` is either `null` or `String.Empty`) has been satisfied, program execution stops 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. In this case, `Main` is the currently executing method. 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 and pressing `Enter`. The result is `true`.



```
C#  
? name == String.Empty
```

The question mark directs the immediate window to [evaluate an expression](#).



9. Press `F5` to continue program execution.
10. Press any key to close the console window and stop debugging.
11. Clear the breakpoint by clicking on the dot in the left margin of the code window. Other ways to clear a breakpoint are by pressing `F9` or choosing **Debug > Toggle Breakpoint** while the line of code is selected.

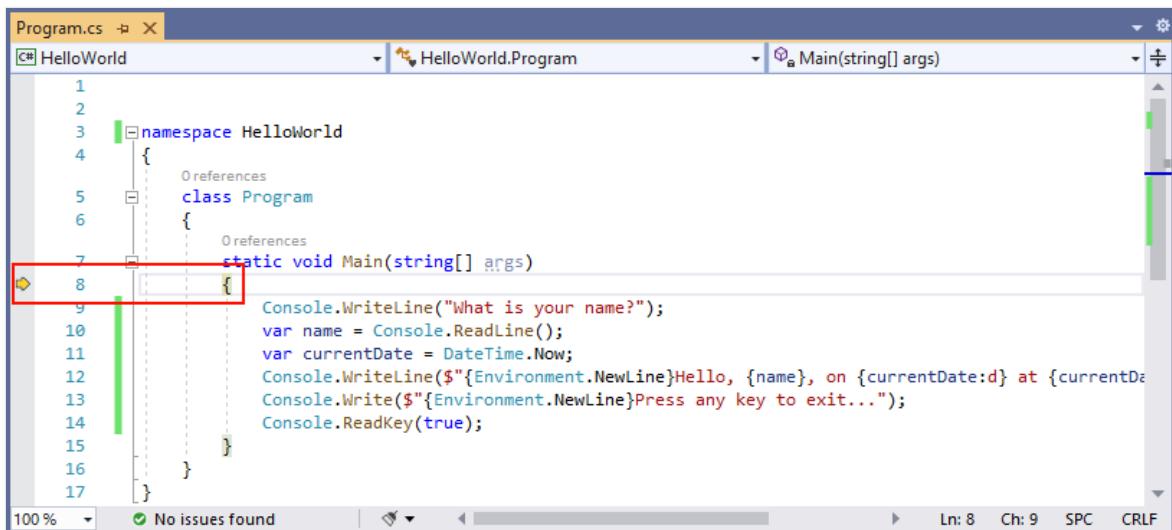
Step through a program

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

1. Choose **Debug > Step Into**. Another way to debug one statement at a time is by pressing `F11`.

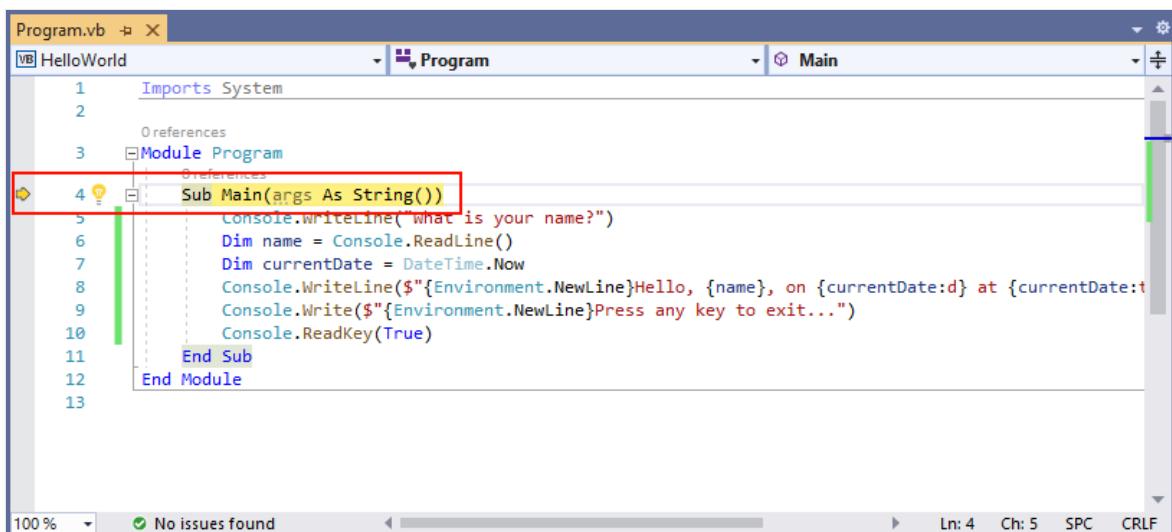
Visual Studio highlights and displays an arrow beside the next line of execution.

C#



```
1
2
3  namespace HelloWorld
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("What is your name?");
10             var name = Console.ReadLine();
11             var currentDate = DateTime.Now;
12             Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate:d} at {currentDate:t}");
13             Console.Write($"{Environment.NewLine}Press any key to exit...");
14             Console.ReadKey(true);
15         }
16     }
17 }
```

Visual Basic

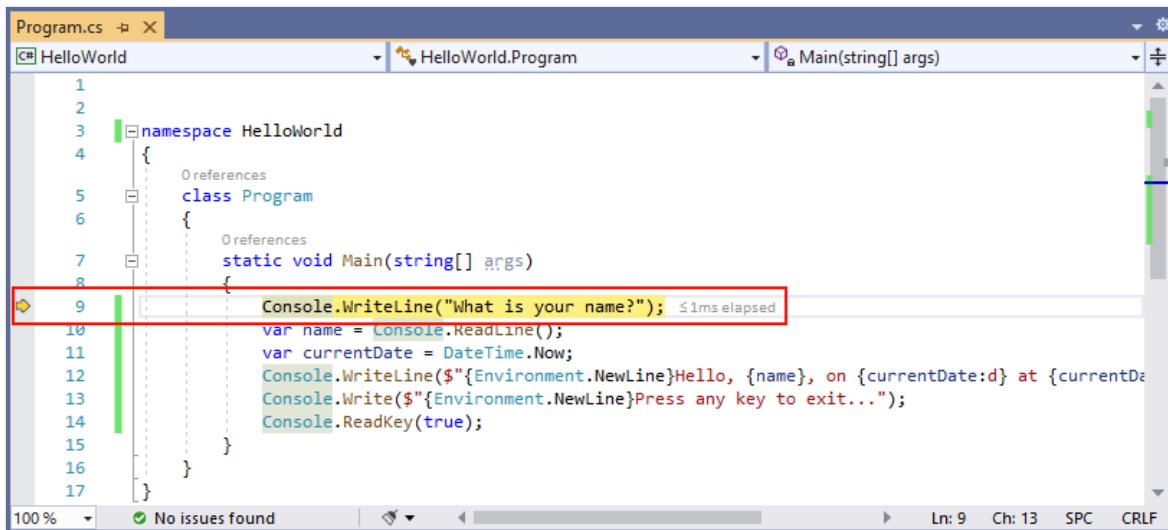


```
1 Imports System
2
3  Module Program
4      Sub Main(args As String())
5          Console.WriteLine("What is your name?")
6          Dim name = Console.ReadLine()
7          Dim currentDate = DateTime.Now
8          Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate:d} at {currentDate:t}");
9          Console.Write($"{Environment.NewLine}Press any key to exit...")
10         Console.ReadKey(True)
11     End Sub
12 End Module
```

At this point, the **Locals** window shows that the `args` array is empty, and `name` and `currentDate` have default values. In addition, Visual Studio has opened a blank console window.

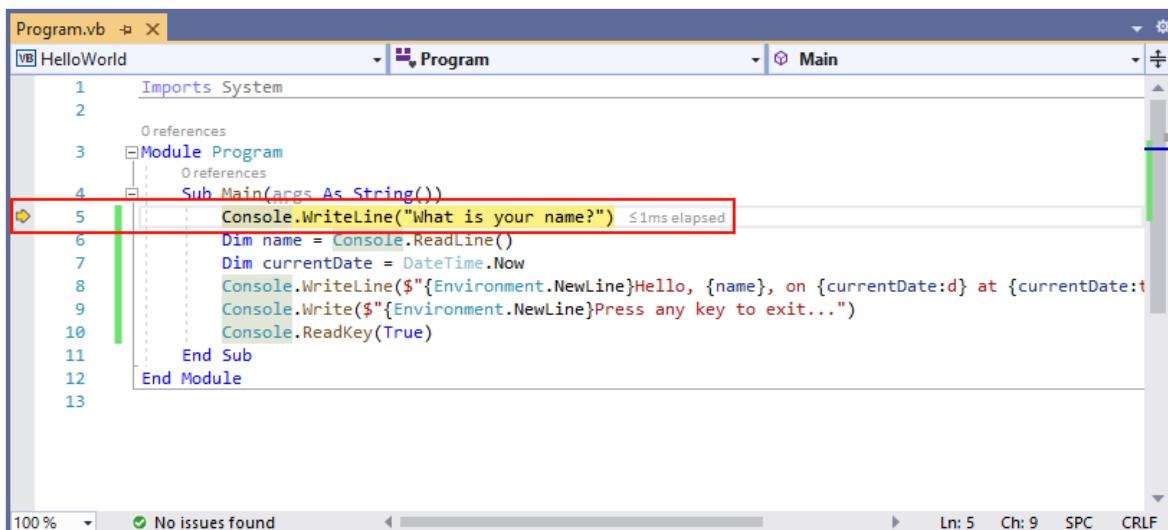
2. Press **F11**. Visual Studio now highlights the next line of execution. The **Locals** window is unchanged, and the console window remains blank.

C#



```
1
2
3  namespace HelloWorld
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("What is your name?"); // Line 9 is highlighted with a red box
10             var name = Console.ReadLine();
11             var currentDate = DateTime.Now;
12             Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate:d} at {currentDate:t}");
13             Console.Write($"{Environment.NewLine}Press any key to exit...");
14             Console.ReadKey(true);
15         }
16     }
17 }
```

Visual Basic



```
1  Imports System
2
3  Module Program
4      Sub Main(args As String())
5          Console.WriteLine("What is your name?") // Line 5 is highlighted with a red box
6          Dim name = Console.ReadLine()
7          Dim currentDate = DateTime.Now
8          Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate:d} at {currentDate:t}");
9          Console.Write($"{Environment.NewLine}Press any key to exit...");
10         Console.ReadKey(True)
11     End Sub
12  End Module
13
```

3. Press **F11**. Visual Studio highlights the statement that includes the `name` variable assignment. The **Locals** 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 entered isn't displayed in the console window, but the `Console.ReadLine` method will nevertheless capture your input.
5. Press **F11**. Visual Studio highlights the statement that includes the `currentDate` variable assignment. The **Locals** window shows the value returned by the call to the `Console.ReadLine` method. The console window also displays the string you entered at the prompt.
6. Press **F11**. The **Locals** window shows the value of the `currentDate` variable after the assignment from the `DateTime.Now` property. The console window is unchanged.

7. Press **F11**. Visual Studio calls the `Console.WriteLine(String, Object, Object)` method.

The console window displays the formatted string.

8. Choose **Debug > Step Out**. Another way to stop step-by-step execution is by pressing **Shift + F11**.

The console window displays a message and waits for you to press a key.

9. Press any key to close the console window and stop debugging.

Use Release build configuration

Once you've tested the **Debug** version 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 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 the application. You can test it as you did the **Debug** version.

Next steps

In this tutorial, you used Visual Studio debugging tools. In the next tutorial, you publish a deployable version of the app.

[Publish a .NET console application using Visual Studio](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For

.NET

 .NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

more information, see [our contributor guide](#).

 [Provide product feedback](#)

Tutorial: Publish a .NET console application using Visual Studio

Article • 08/25/2023

This tutorial shows how to publish a console app so that other users can run it. Publishing creates the set of files that are needed to run your application. To deploy the files, copy them to the target machine.

Prerequisites

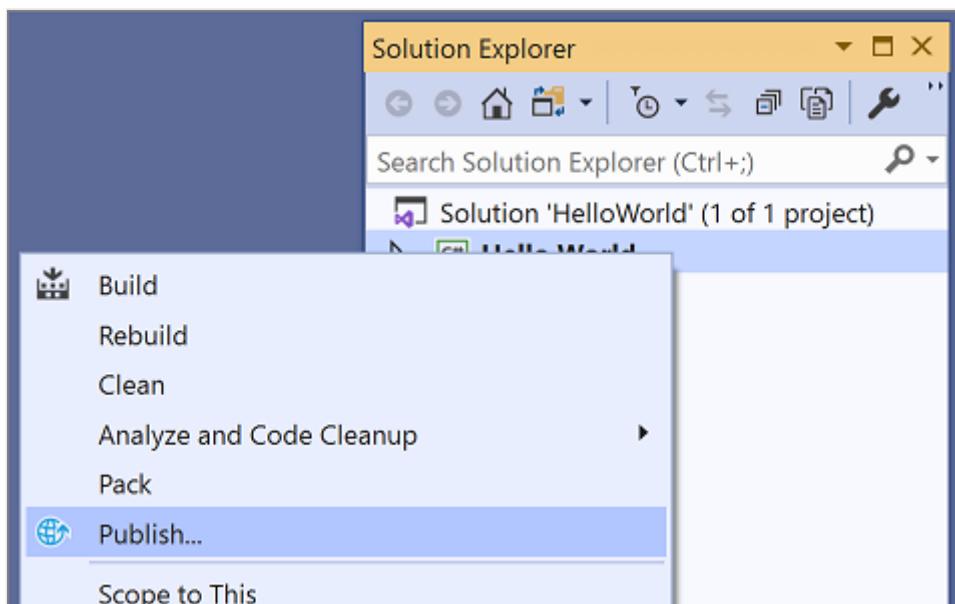
- This tutorial works with the console app that you create in [Create a .NET console application using Visual Studio](#).

Publish the app

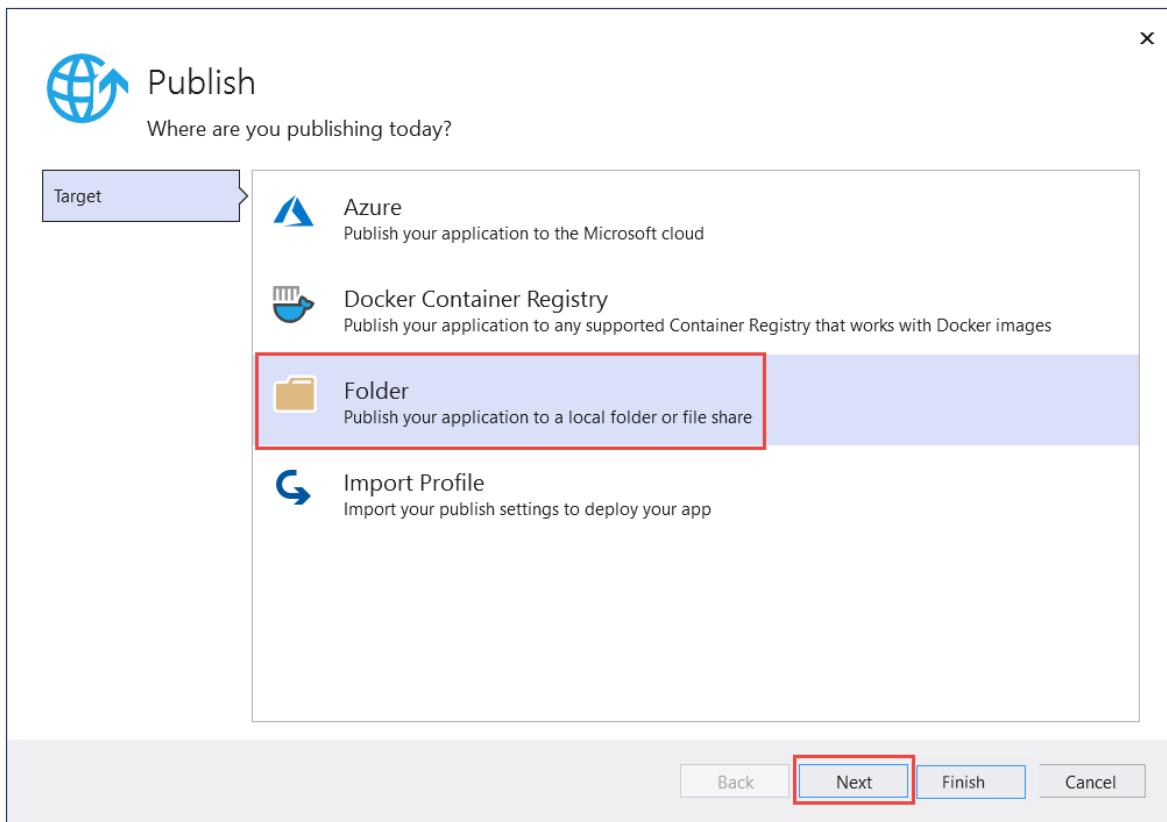
1. Start Visual Studio.
2. Open the *HelloWorld* project that you created in [Create a .NET console application using Visual Studio](#).
3. Make sure that Visual Studio is using the Release build configuration. If necessary, change the build configuration setting on the toolbar from **Debug** to **Release**.



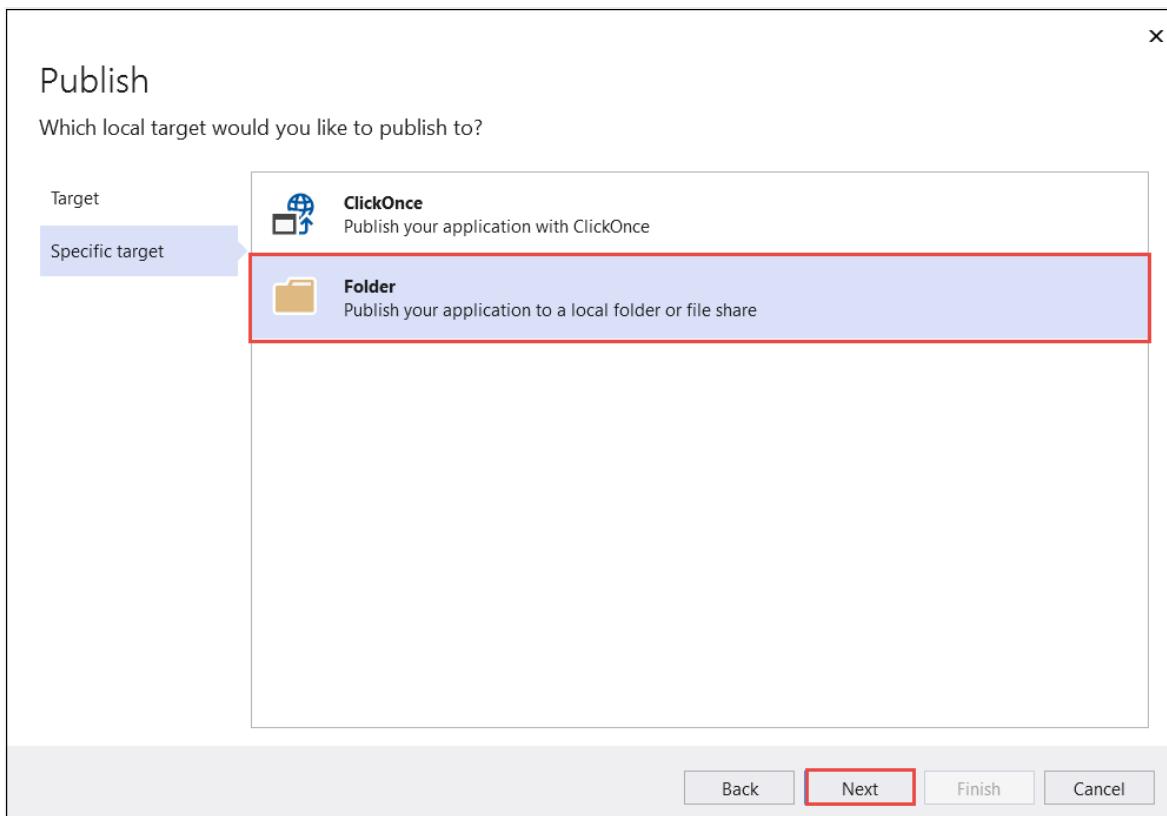
4. Right-click on the **HelloWorld** project (not the **HelloWorld** solution) and select **Publish** from the menu.



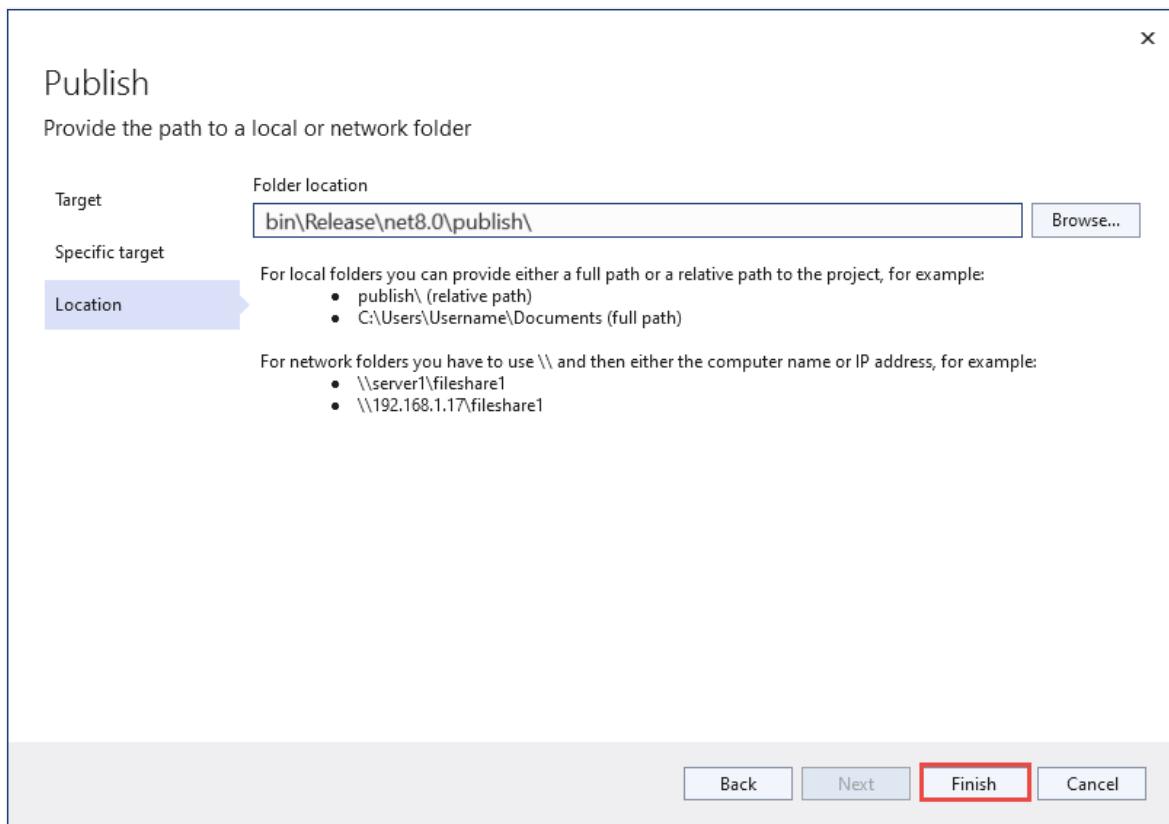
5. On the **Target** tab of the **Publish** page, select **Folder**, and then select **Next**.



6. On the **Specific Target** tab of the **Publish** page, select **Folder**, and then select **Next**.

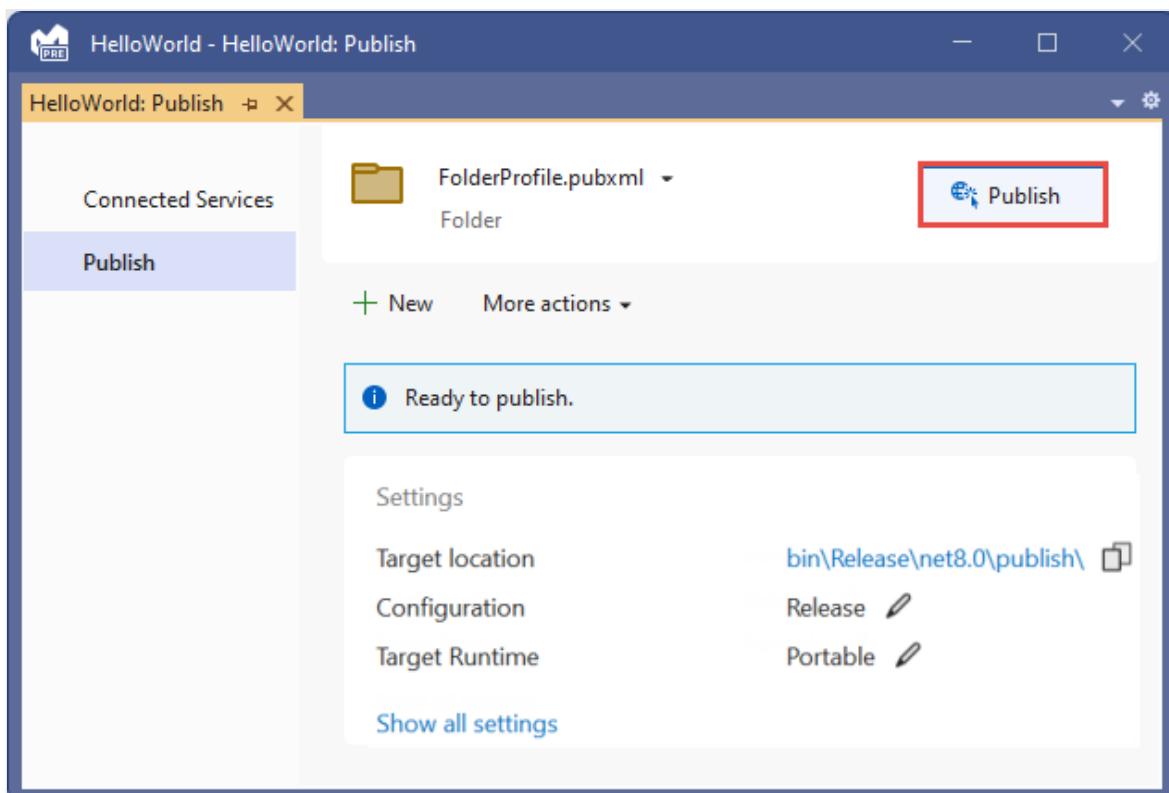


7. On the **Location** tab of the **Publish** page, select **Finish**.



8. On the **Publish profile creation progress** page, select **Close**.

9. On the **Publish** tab of the **Publish** window, select **Publish**.

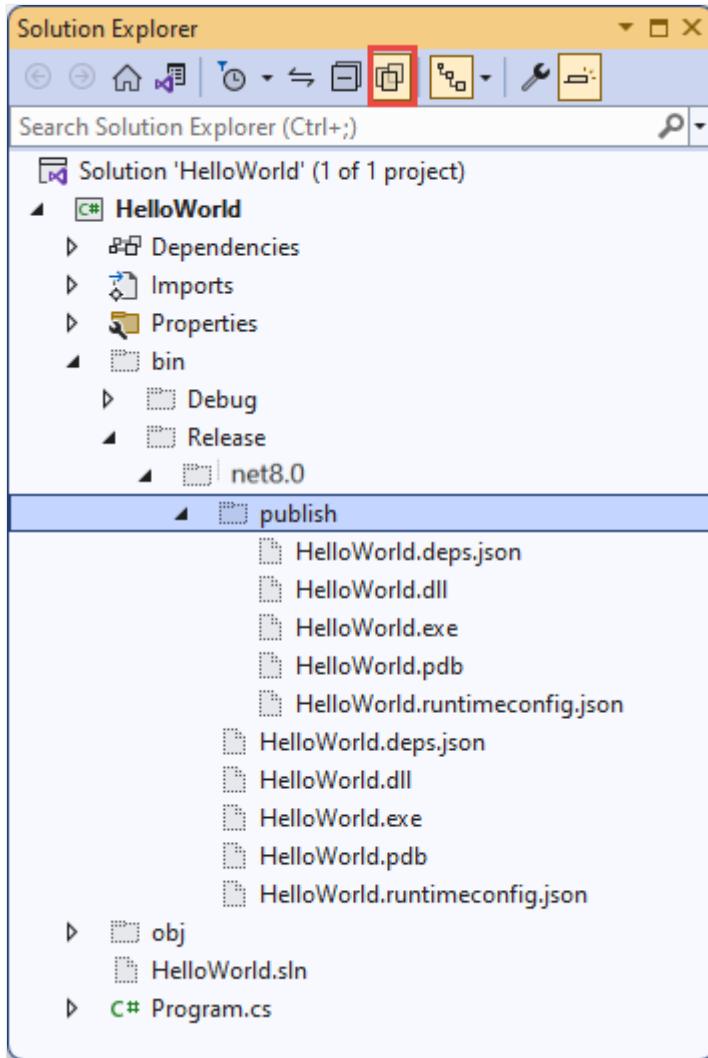


Inspect the files

By default, the publishing process creates a framework-dependent deployment, which is a type of deployment where the published application runs on a machine that has the .NET runtime installed. Users can run the published app by double-clicking the executable or issuing the `dotnet HelloWorld.dll` command from a command prompt.

In the following steps, you'll look at the files created by the publish process.

1. In **Solution Explorer**, select **Show all files**.
2. In the project folder, expand `bin/Release/net7.0/publish`.



As the image shows, the published output includes the following files:

- *HelloWorld.deps.json*

This is the application's runtime dependencies file. It defines the .NET components and the libraries (including the dynamic link library that contains your application) needed to run the app. For more information, see [Runtime configuration files](#).

- *HelloWorld.dll*

This is the [framework-dependent deployment](#) version of the application. To execute this dynamic link library, enter `dotnet HelloWorld.dll` at a command prompt. This method of running the app works on any platform that has the .NET runtime installed.

- *HelloWorld.exe*

This is the [framework-dependent executable](#) version of the application. To run it, enter `HelloWorld.exe` at a command prompt. The file is operating-system-specific.

- *HelloWorld.pdb* (optional for deployment)

This is the debug symbols file. 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*

This is the application's runtime configuration file. It identifies the version of .NET that your application was built to run on. You can also add configuration options to it. For more information, see [.NET runtime configuration settings](#).

Run the published app

1. In **Solution Explorer**, right-click the *publish* folder, and select **Copy Full Path**.
2. Open a command prompt and navigate to the *publish* folder. To do that, enter `cd` and then paste the full path. For example:

```
Console
```

```
cd C:\Projects\HelloWorld\bin\Release\net8.0\publish\
```

3. Run the app by using the executable:
 - a. Enter `HelloWorld.exe` and press `Enter`.
 - b. Enter a name in response to the prompt, and press any key to exit.
4. Run the app by using the `dotnet` command:
 - a. Enter `dotnet HelloWorld.dll` and press `Enter`.
 - b. Enter a name in response to the prompt, and press any key to exit.

Additional resources

- [.NET application deployment](#)
- [Publish .NET apps with the .NET CLI](#)
- [dotnet publish](#)
- [Tutorial: Publish a .NET console application using Visual Studio Code](#)
- [Use the .NET SDK in continuous integration \(CI\) environments](#)

Next steps

In this tutorial, you published a console app. In the next tutorial, you create a class library.

[Create a .NET class library using Visual Studio](#)

 [Collaborate with us on GitHub](#)

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

[.NET feedback](#)

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Create a .NET class library using Visual Studio

Article • 08/25/2023

In this tutorial, you create a simple class library that contains a single string-handling method.

A *class library* defines types and methods that are called by an application. If the library targets .NET Standard 2.0, it can be called by any .NET implementation (including .NET Framework) that supports .NET Standard 2.0. If the library targets .NET 8, it can be called by any application that targets .NET 8. This tutorial shows how to target .NET 8.

When you create a class library, you can distribute it as a NuGet package or as a component bundled with the application that uses it.

Prerequisites

- [Visual Studio 2022 Preview](#) with the **.NET desktop development** workload installed. The .NET 8 SDK is automatically installed when you select this workload.

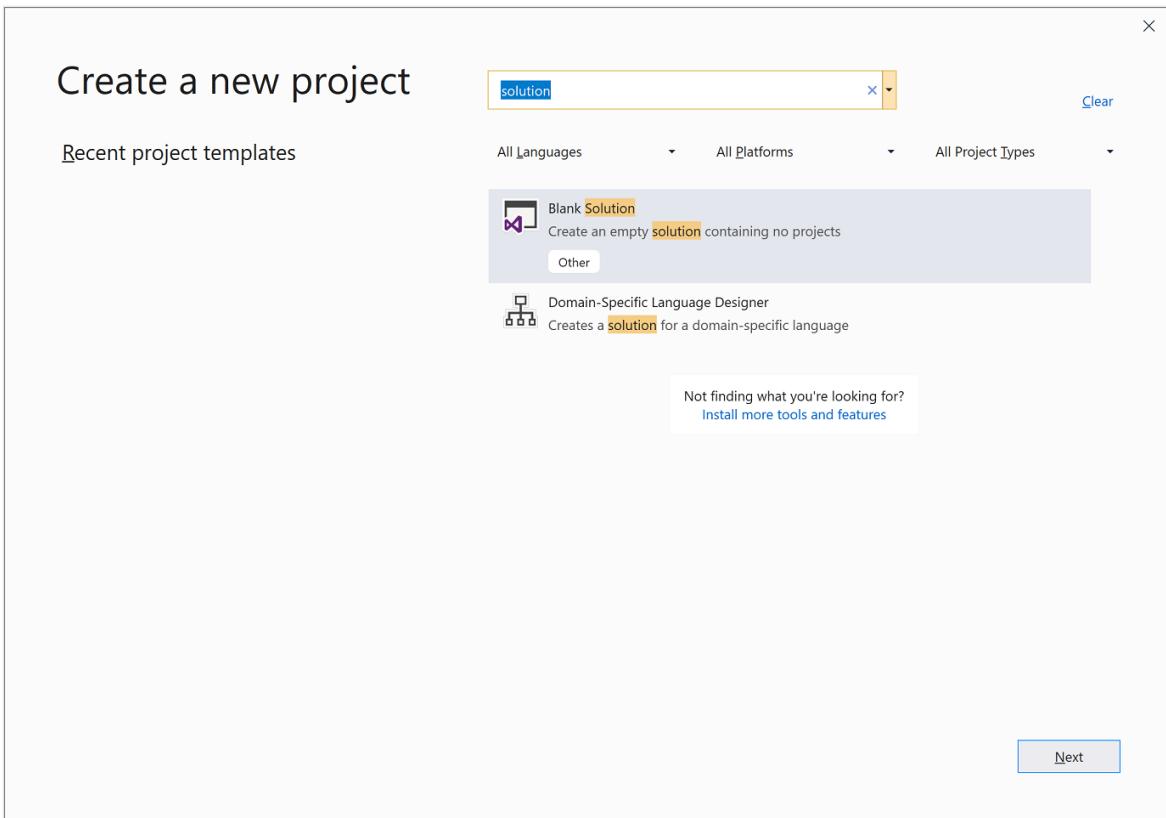
For more information, see [Install the .NET SDK with Visual Studio](#).

Create a solution

Start by creating a blank solution to put the class library project in. A Visual Studio solution serves as a container for one or more projects. You'll add additional, related projects to the same solution.

To create the blank solution:

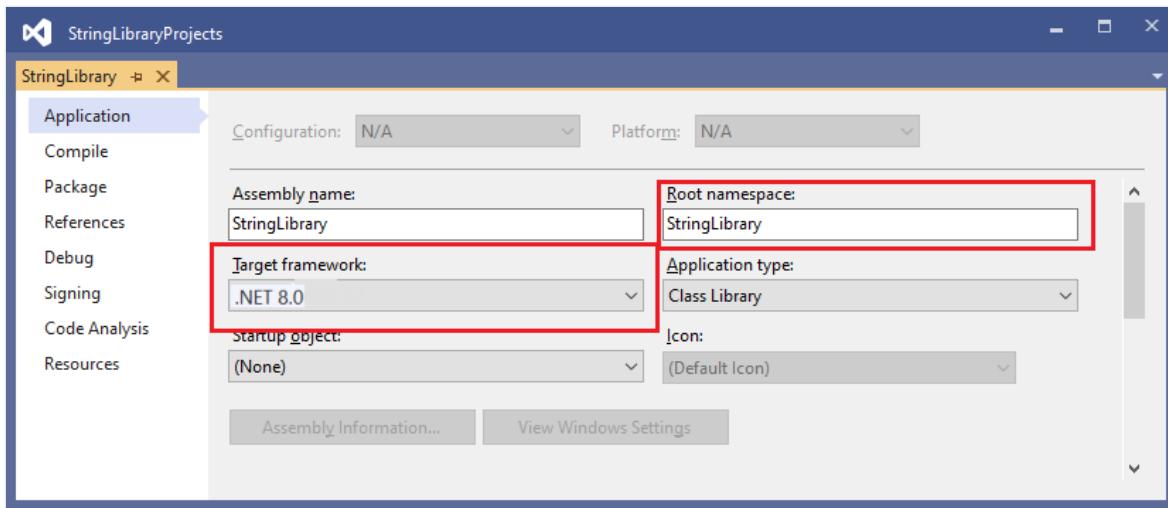
1. Start Visual Studio.
2. On the start window, choose **Create a new project**.
3. On the **Create a new project** page, enter **solution** in the search box. Choose the **Blank Solution** template, and then choose **Next**.



4. On the **Configure your new project** page, enter **ClassLibraryProjects** in the **Solution name** box. Then choose **Create**.

Create a class library project

1. Add a new .NET class library project named "StringLibrary" to the solution.
 - a. Right-click on the solution in **Solution Explorer** and select **Add > New Project**.
 - b. On the **Add a new project** page, enter **library** in the search box. Choose **C#** or **Visual Basic** from the **Language** list, and then choose **All platforms** from the **Platform** list. Choose the **Class Library** template, and then choose **Next**.
 - c. On the **Configure your new project** page, enter **StringLibrary** in the **Project name** box, and then choose **Next**.
 - d. On the **Additional information** page, select **.NET 8 (Preview)**, and then choose **Create**.
2. Check to make sure that the library targets the correct version of .NET. Right-click on the library project in **Solution Explorer**, and then select **Properties**. The **Target Framework** text box shows that the project targets .NET 7.0.
3. If you're using Visual Basic, clear the text in the **Root namespace** text box.



For each project, Visual Basic automatically creates a namespace that corresponds to the project name. In this tutorial, you define a top-level namespace by using the [namespace](#) keyword in the code file.

4. Replace the code in the code window for *Class1.cs* or *Class1.vb* with the following code, and save the file. If the language you want to use isn't shown, change the language selector at the top of the page.

```
C#  
  
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`. This method 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.

`StartsWithUpper` is implemented as an [extension method](#) so that you can call it as if it were a member of the `String` class. The question mark (?) after `string` in the C# code indicates that the string may be null.

5. On the menu bar, select **Build > Build Solution** or press **Ctrl + Shift + B** to verify that the project compiles without error.

Add a console app to the solution

Add a console application that uses the class library. The app will prompt the user to enter a string and report whether the string begins with an uppercase character.

1. Add a new .NET console application named "ShowCase" to the solution.
 - a. Right-click on the solution in **Solution Explorer** and select **Add > New project**.
 - b. On the **Add a new project** page, enter **console** in the search box. Choose **C#** or **Visual Basic** from the **Language** list, and then choose **All platforms** from the **Platform** list.
 - c. Choose the **Console Application** template, and then choose **Next**.
 - d. On the **Configure your new project** page, enter **ShowCase** in the **Project name** box. Then choose **Next**.
 - e. On the **Additional information** page, select **.NET 8 (Preview)** in the **Framework** box. Then choose **Create**.
2. In the code window for the *Program.cs* or *Program.vb* file, replace all of the code with the following code.

```
C#  
  
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}");  
            Console.WriteLine("Begins with uppercase? " +  
                $"{(input.StartsWithUpper() ? "Yes" : "No")});  
            Console.WriteLine();
```

```
        row += 4;
    } 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($"{Environment.NewLine}Press <Enter> only
to exit; otherwise, enter a string and press <Enter>:
{Environment.NewLine}");
        row = 3;
    }
}
```

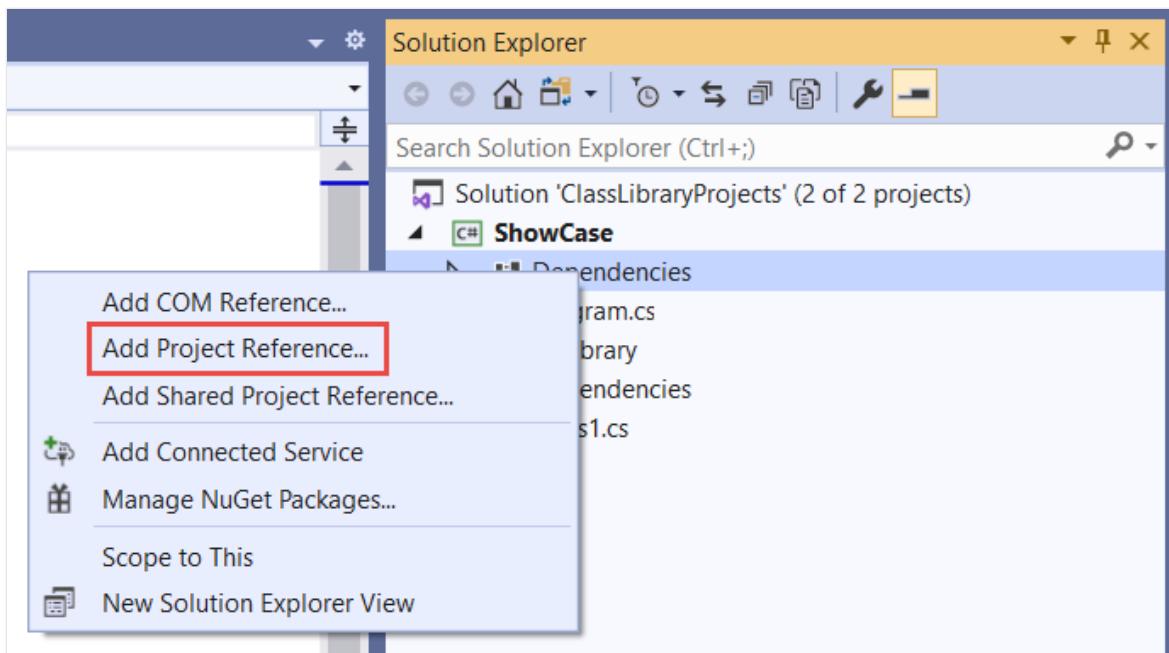
The code uses the `row` variable to maintain a count of the number of rows of data written to the console window. Whenever it's greater than or equal to 25, the code clears the console window and displays a message to the user.

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 ends, and the console window closes.

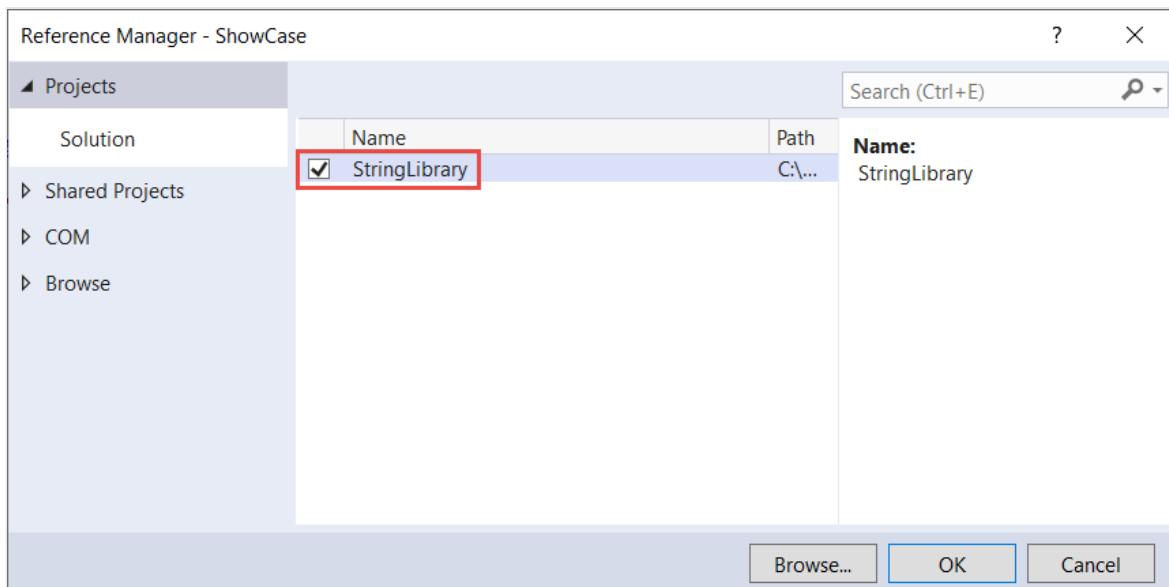
Add a project reference

Initially, the new console app project doesn't have access to the class library. To allow it to call methods in the class library, create a project reference to the class library project.

1. In **Solution Explorer**, right-click the `ShowCase` project's **Dependencies** node, and select **Add Project Reference**.

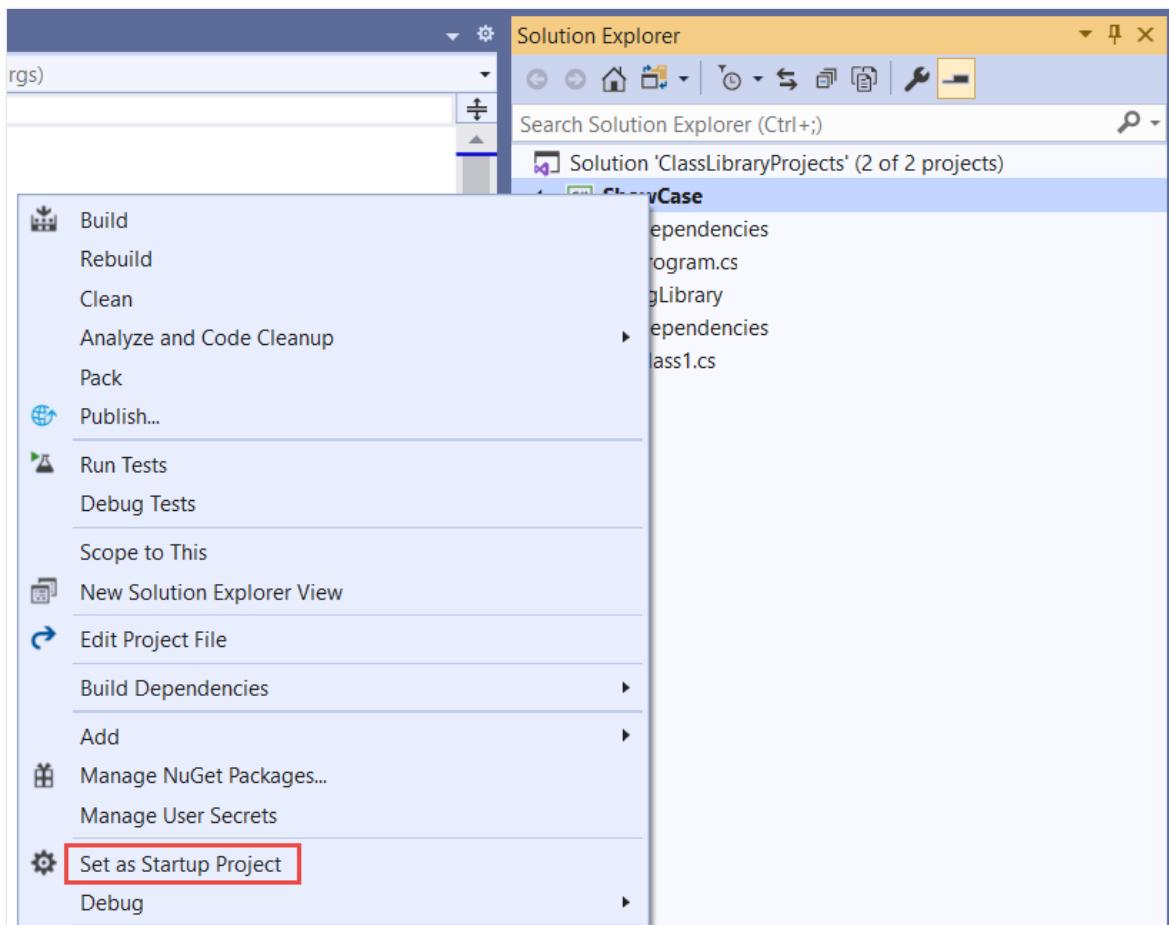


2. In the **Reference Manager** dialog, select the **StringLibrary** project, and select **OK**.



Run the app

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



2. Press **Ctrl + F5** to compile and run the program without debugging.
3. Try out the program by entering strings and pressing **Enter**, then press **Enter** to exit.

```
Press <Enter> only to exit; otherwise, enter a string and press <Enter>:  
Hello  
Input: Hello  
Begins with uppercase? Yes  
  
hello  
Input: hello  
Begins with uppercase? No
```

Additional resources

- Develop libraries with the .NET CLI
- .NET Standard versions and the platforms they support.

Next steps

In this tutorial, you created a class library. In the next tutorial, you learn how to unit test the class library.

Unit test a .NET class library using Visual Studio

Or you can skip automated unit testing and learn how to share the library by creating a NuGet package:

Create and publish a package using Visual Studio

Or learn how to publish a console app. If you publish the console app from the solution you created in this tutorial, the class library goes with it as a *.dll* file.

Publish a .NET console application using Visual Studio

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Test a .NET class library with .NET using Visual Studio

Article • 08/25/2023

This tutorial shows how to automate unit testing by adding a test project to a solution.

Prerequisites

- This tutorial works with the solution that you create in [Create a .NET class library using Visual Studio](#).

Create a unit test project

Unit tests provide automated software testing during your development and publishing. [MSTest](#) is one of three test frameworks you can choose from. The others are [xUnit](#) and [nUnit](#).

1. Start Visual Studio.
2. Open the `ClassLibraryProjects` solution you created in [Create a .NET class library using Visual Studio](#).
3. Add a new unit test project named "StringLibraryTest" to the solution.
 - a. Right-click on the solution in **Solution Explorer** and select **Add > New project**.
 - b. On the **Add a new project** page, enter **mstest** in the search box. Choose **C#** or **Visual Basic** from the **Language** list, and then choose **All platforms** from the **Platform** list.
 - c. Choose the **MSTest Test Project** template, and then choose **Next**.
 - d. On the **Configure your new project** page, enter **StringLibraryTest** in the **Project name** box. Then choose **Next**.
 - e. On the **Additional information** page, select **.NET 8 (Preview)** in the **Framework** box. Then choose **Create**.
4. Visual Studio creates the project and opens the class file in the code window with the following code. If the language you want to use is not shown, change the language selector at the top of the page.

C#

```
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. In C#, the namespace is imported via a `global using` directive in `GlobalUsings.cs`.
- It applies the `TestClassAttribute` attribute to the `UnitTest1` class.
- It applies the `TestMethodAttribute` attribute to define `TestMethod1` in C# or `TestSub` in Visual Basic.

Each method tagged with `[TestMethod]` in a test class tagged with `[TestClass]` is executed automatically when the unit test is run.

Add a project reference

For the test project to work with the `StringLibrary` class, add a reference in the `StringLibraryTest` project to the `StringLibrary` project.

1. In **Solution Explorer**, right-click the **Dependencies** node of the `StringLibraryTest` project and select **Add Project Reference** from the context menu.
2. In the **Reference Manager** dialog, expand the **Projects** node, and select the box next to `StringLibrary`. Adding a reference to the `StringLibrary` assembly allows the compiler to find `StringLibrary` methods while compiling the `StringLibraryTest` project.
3. Select **OK**.

Add and run unit test methods

When Visual Studio runs a unit test, it executes each method that is marked with the [TestMethodAttribute](#) attribute in a class that is marked with the [TestClassAttribute](#) attribute. A test method ends when the first failure is found 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 the [Assert](#) class's most frequently called methods are shown in the following table:

Assert methods	Function
<code>Assert.AreEqual</code>	Verifies that two values or objects are equal. The assert fails if the values or objects aren't equal.
<code>Assert.AreSame</code>	Verifies that two object variables refer to the same object. The assert fails if the variables refer to different objects.
<code>Assert.IsFalse</code>	Verifies that a condition is <code>false</code> . The assert fails if the condition is <code>true</code> .
<code>Assert.IsNotNull</code>	Verifies that an object isn't <code>null</code> . The assert fails if the object is <code>null</code> .

You can also use the [Assert.ThrowsException](#) method in a test method to indicate the type of exception it's expected to throw. The test fails if the specified exception isn't 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 [Assert.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 [Assert.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 hasn't been initialized. You can call `StartsWithUpper` directly as a static method and pass a single [String](#) argument. Or you can call `StartsWithUpper` as an extension method on a `string` variable assigned to `null`.

You'll define three methods, each of which calls an [Assert](#) method for each element in a string array. You'll call a method overload that lets you specify an error message to be displayed in case of test failure. The message identifies the string that caused the failure.

To create the test methods:

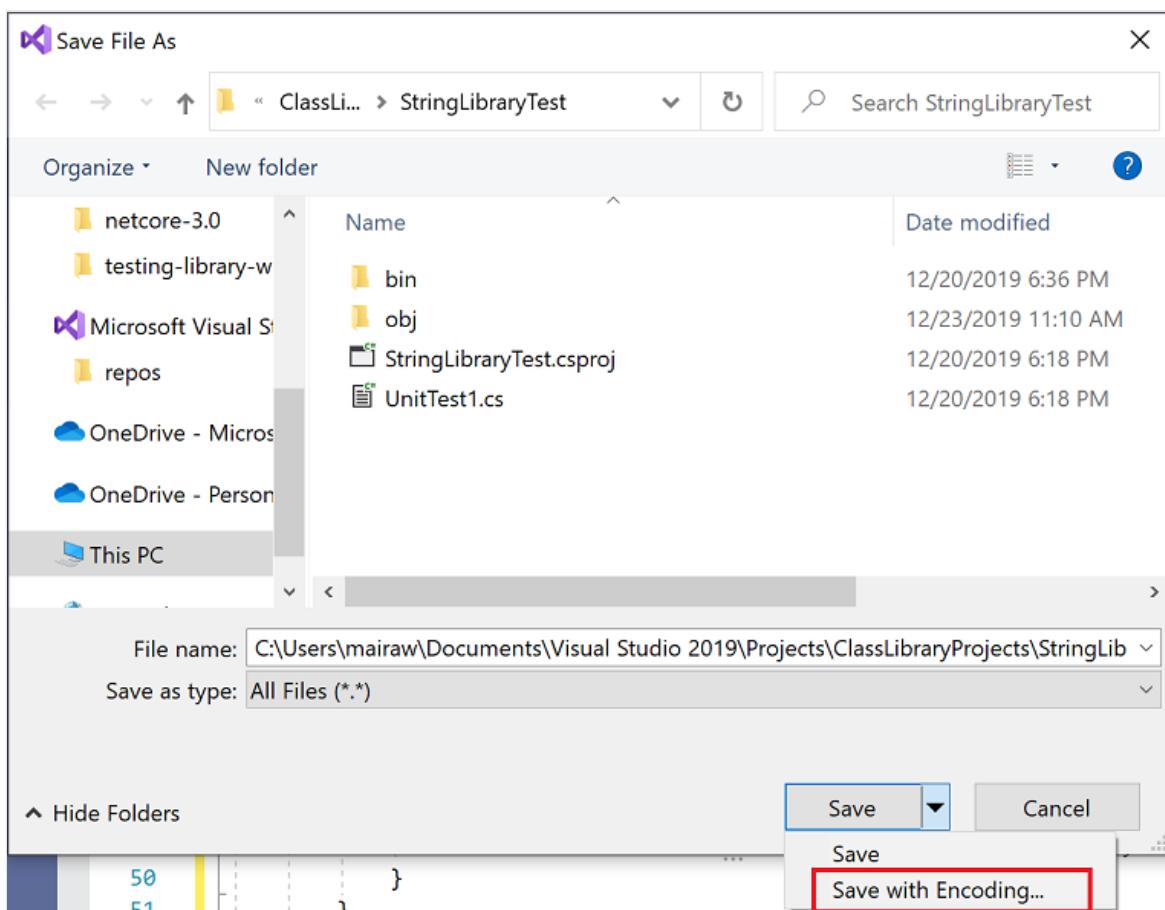
1. In the *UnitTest1.cs* or *UnitTest1.vb* code window, replace the code with the following code:

```
C#  
  
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)
```

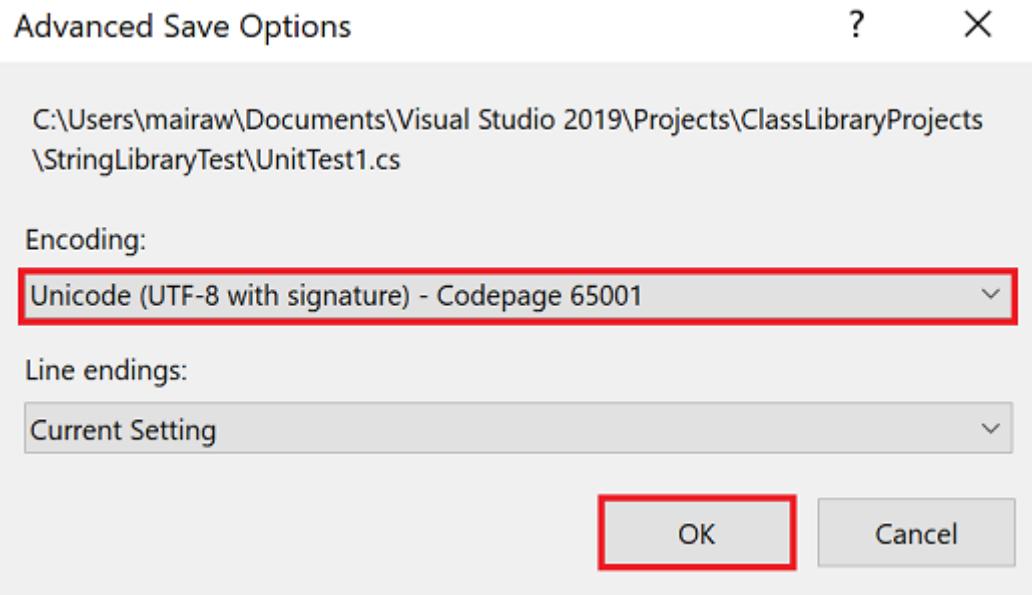
```
        {
            bool result = StringLibrary.StartsWithUpper(word);
            Assert.IsFalse(result,
                string.Format("Expected for '{0}': false;
Actual: {1}",
                word == null ? "<null>" : word,
result));
        }
    }
}
```

The test of uppercase characters in the `TestStartsWithUpper` method includes the Greek capital letter alpha (U+0391) and the Cyrillic capital letter EM (U+041C). 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** or **File > Save UnitTest1.vb As**. In the **Save File As** dialog, select the arrow beside the **Save** button, and select **Save with Encoding**.

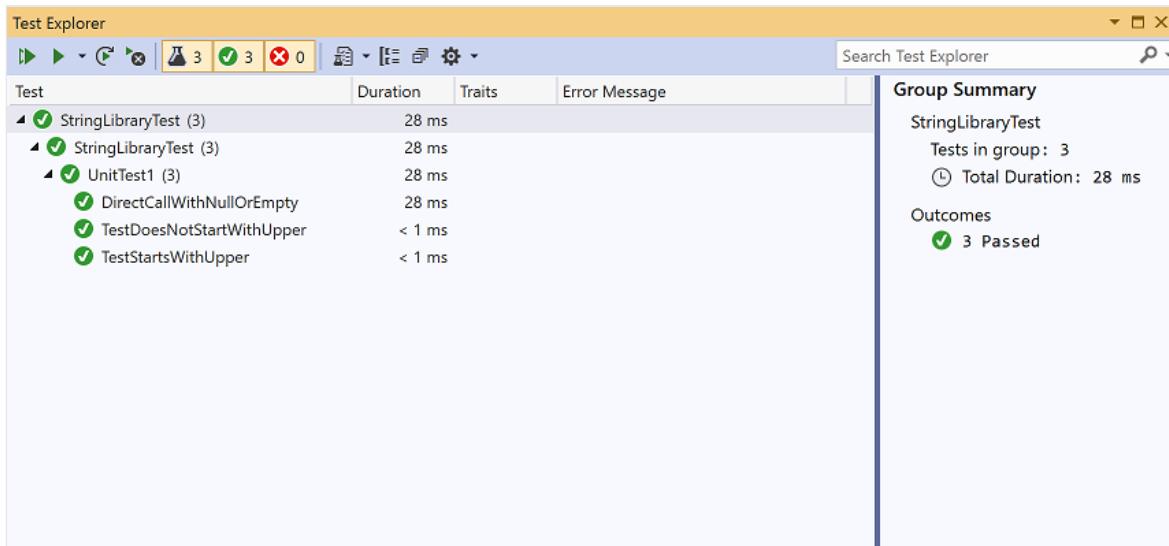


3. In the **Confirm Save As** dialog, select the **Yes** button to save the file.
 4. In the **Advanced Save Options** dialog, select **Unicode (UTF-8 with signature) - Codepage 65001** from the **Encoding** drop-down list and select **OK**.



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 correct.

5. On the menu bar, select **Test > Run All Tests**. If the **Test Explorer** window doesn't open, open it by choosing **Test > Test Explorer**. The three tests are listed in the **Passed Tests** section, and the **Summary** section reports the result of the test run.



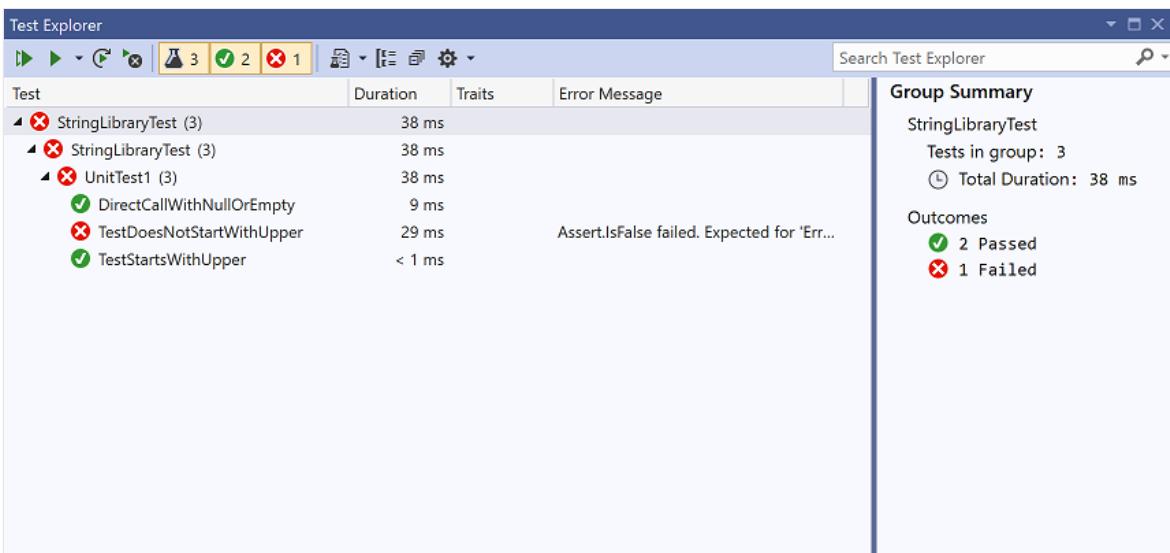
Handle test failures

If you're doing test-driven development (TDD), you write tests first and they fail the first time you run them. Then you add code to the app that makes the test succeed. For this tutorial, you created the test after writing the app code that it validates, so you haven't seen the test fail. To validate that a test fails when you expect it to fail, add an invalid value to the test input.

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.

```
C#  
  
string[] words = { "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.



3. Select the failed test, `TestDoesNotStartWithUpper`.

The **Test Explorer** window displays the message produced by the assert: "Assert.IsFalse failed. Expected for 'Error': false; actual: True". Because of the failure, no strings in the array after "Error" were tested.

```
Test Detail Summary  
  
✖ TestDoesNotStartWithUpper  
    Source: UnitTest1.cs line 25  
    Duration: 29 ms  
  
    Message:  
        Assert.IsFalse failed. Expected for 'Error': false; Actual: True  
  
    Stack Trace:  
        UnitTest1.TestDoesNotStartWithUpper\(\) line 34
```

4. Remove the string "Error" that you added in step 1. Rerun the test and the tests pass.

Test the Release version of the library

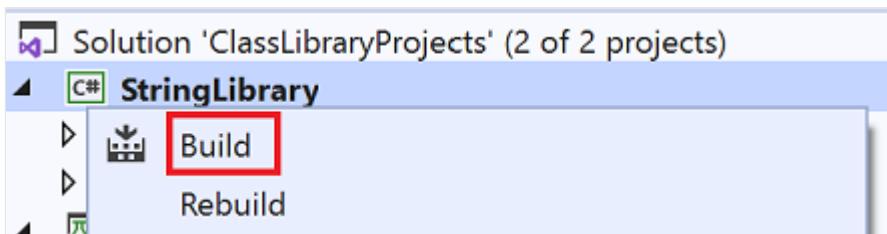
Now that the tests have all passed when running the Debug build of the library, run the tests an additional time against the Release build of the library. A number of factors, including compiler optimizations, can sometimes produce different behavior between Debug and Release builds.

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.

Debug tests

If you're using Visual Studio as your IDE, you can use the same process shown in [Tutorial: Debug a .NET console application using Visual Studio](#) to debug code using your unit test project. Instead of starting the *ShowCase* app project, right-click the **StringLibraryTests** project, and select **Debug Tests** from the context menu.

Visual Studio starts the test project with the debugger attached. Execution will stop at any breakpoint you've added to the test project or the underlying library code.

Additional resources

- [Unit test basics - Visual Studio](#)
- [Unit testing in .NET](#)

Next steps

In this tutorial, you unit tested a class library. You can make the library available to others by publishing it to [NuGet](#) as a package. To learn how, follow a NuGet tutorial:

[Create and publish a NuGet package using Visual Studio](#)

If you publish a library as a NuGet package, others can install and use it. To learn how, follow a NuGet tutorial:

[Install and use a package in Visual Studio](#)

A library doesn't have to be distributed as a package. It can be bundled with a console app that uses it. To learn how to publish a console app, see the earlier tutorial in this series:

[Publish a .NET console application using Visual Studio](#)

 Collaborate with us on
GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

 .NET feedback

The .NET documentation is open source. Provide feedback [here](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Quickstart: Install and use a NuGet package in Visual Studio (Windows only)

Article • 08/21/2023

A *NuGet package* contains reusable code that other developers have made available to you for use in your projects. You can install a NuGet package in a Microsoft Visual Studio project by using the [NuGet Package Manager](#), the [Package Manager Console](#), or the [.NET CLI](#). This article demonstrates how to create a Windows Presentation Foundation (WPF) project with the popular `Newtonsoft.Json` package. The same process applies to any other .NET or .NET Core project.

After you install a NuGet package, you can then make a reference to it in your code with the `using <namespace>` statement, where `<namespace>` is the name of package you're using. After you've made a reference, you can then call the package through its API.

The article is for Windows users only. If you're using Visual Studio for Mac, see [Install and use a package in Visual Studio for Mac](#).

Tip

To find a NuGet package, start with [nuget.org](#). Browsing [nuget.org](#) is how .NET developers typically find components they can reuse in their own applications. You can do a search of [nuget.org](#) directly or find and install packages within Visual Studio as shown in this article. For more information, see [Find and evaluate NuGet packages](#).

Prerequisites

- Install Visual Studio 2022 for Windows with the .NET desktop development workload.

You can install the 2022 Community edition for free from visualstudio.microsoft.com, or use the Professional or Enterprise edition.

Create a project

You can install a NuGet package into any .NET project if that package supports the same target framework as the project. However, for this quickstart you'll create a Windows Presentation Foundation (WPF) Application project.

Follow these steps:

1. In Visual Studio, select **File > New > Project**.
2. In the **Create a new project** window, enter **WPF** in the search box and select **C#** and **Windows** in the dropdown lists. In the resulting list of project templates, select **WPF Application**, and then select **Next**.
3. In the **Configure your new project** window, optionally update the **Project name** and the **Solution name**, and then select **Next**.
4. In the **Additional information** window, select **.NET 6.0** (or the latest version) for **Framework**, and then select **Create**.

Visual Studio creates the project, and it appears in [Solution Explorer](#).

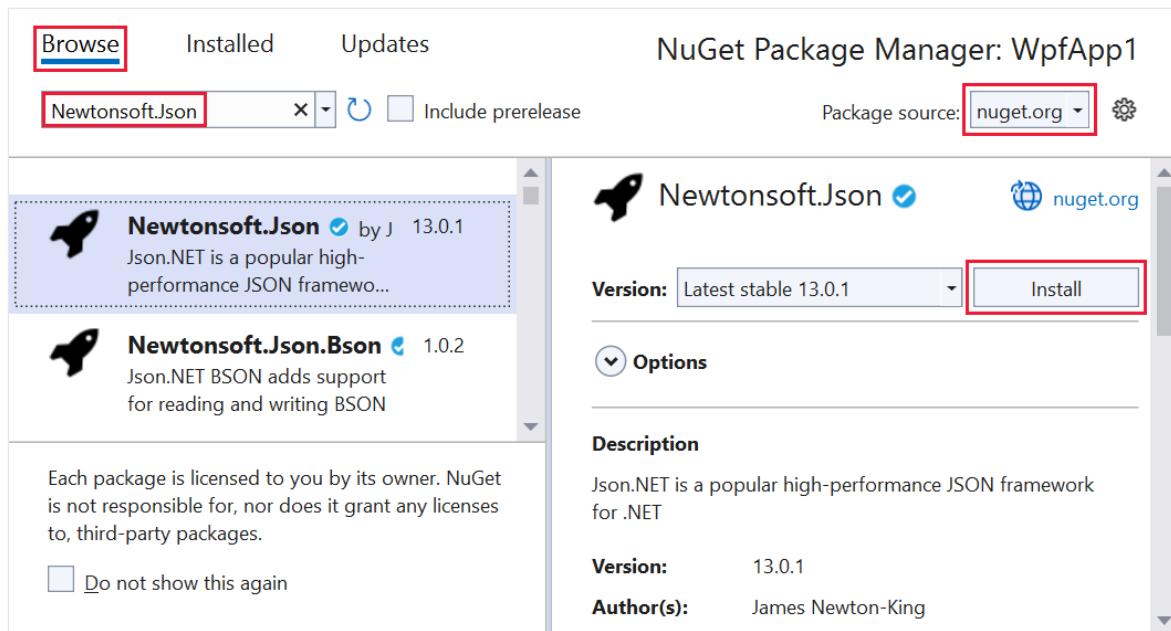
Add the `Newtonsoft.Json` NuGet package

To install a NuGet package in this quickstart, you can use either the NuGet Package Manager or the Package Manager Console. Depending on your project format, the installation of a NuGet package records the dependency in either your project file or a `packages.config` file. For more information, see [Package consumption workflow](#).

NuGet Package Manager

To use the [NuGet Package Manager](#) to install the `Newtonsoft.Json` package in Visual Studio, follow these steps:

1. Select **Project > Manage NuGet Packages**.
2. In the **NuGet Package Manager** page, choose **nuget.org** as the **Package source**.
3. From the **Browse** tab, search for `Newtonsoft.Json`, select **Newtonsoft.Json** in the list, and then select **Install**.

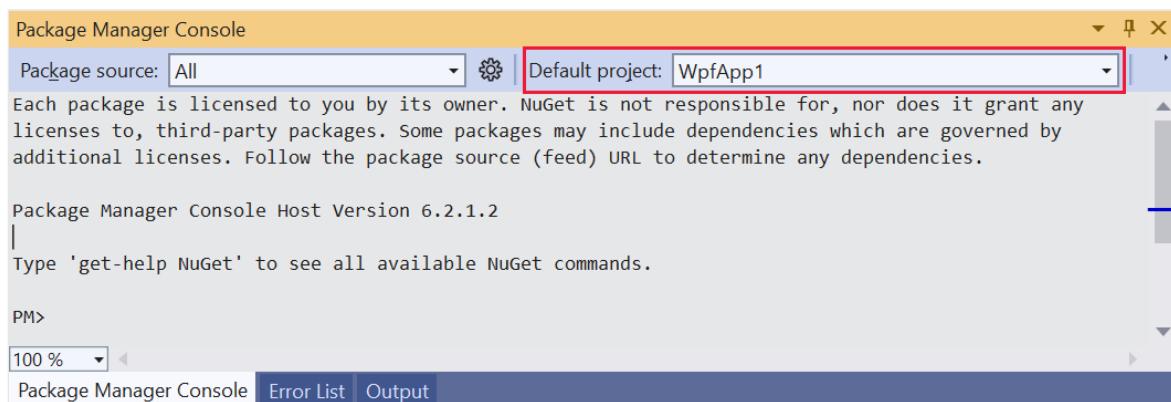


4. If you're prompted to verify the installation, select **OK**.

Package Manager Console

Alternatively, to use the [Package Manager Console](#) in Visual Studio to install the `Newtonsoft.Json` package, follow these steps:

1. From Visual Studio, select **Tools > NuGet Package Manager > Package Manager Console**.
2. After the **Package Manager Console** pane opens, verify that the **Default project** drop-down list shows the project in which you want to install the package. If you have a single project in the solution, it's preselected.



3. At the console prompt, enter the command `Install-Package Newtonsoft.Json`. For more information about this command, see [Install-Package](#).

The console window shows the output for the command. Errors typically indicate that the package isn't compatible with the project's target framework.

Use the Newtonsoft.Json API in the app

With the `Newtonsoft.Json` package in the project, call its `JsonConvert.SerializeObject` method to convert an object to a human-readable string:

1. From **Solution Explorer**, open *MainWindow.xaml* and replace the existing `<Grid>` element with the following code:

XAML

```
<Grid Background="White">
    <StackPanel VerticalAlignment="Center">
        <Button Click="Button_Click" Width="100px"
    HorizontalAlignment="Center" Content="Click Me" Margin="10"/>
        <TextBlock Name="TextBlock" HorizontalAlignment="Center"
    Text="TextBlock" Margin="10"/>
    </StackPanel>
</Grid>
```

2. Open the *MainWindow.xaml.cs* file under the *MainWindow.xaml* node, and insert the following code inside the `MainWindow` class after the constructor:

C#

```
public class Account
{
    public string Name { get; set; }
    public string Email { get; set; }
    public DateTime DOB { get; set; }
}

private void Button_Click(object sender, RoutedEventArgs e)
{
    Account account = new Account
    {
        Name = "John Doe",
        Email = "john@microsoft.com",
        DOB = new DateTime(1980, 2, 20, 0, 0, 0, DateTimeKind.Utc),
    };
    string json = JsonConvert.SerializeObject(account,
Newtonsoft.Json.Formatting.Indented);
    TextBlock.Text = json;
}
```

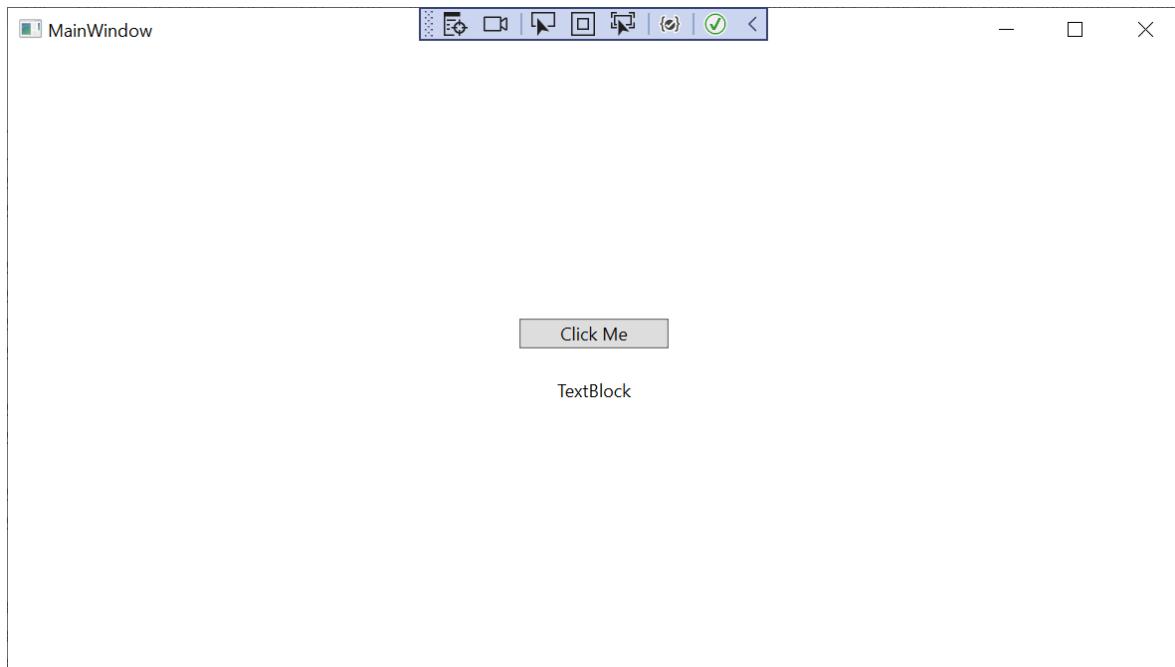
3. To avoid an error for the `JsonConvert` object in the code (a red squiggle line will appear), add the following statement at the beginning of the code file:

C#

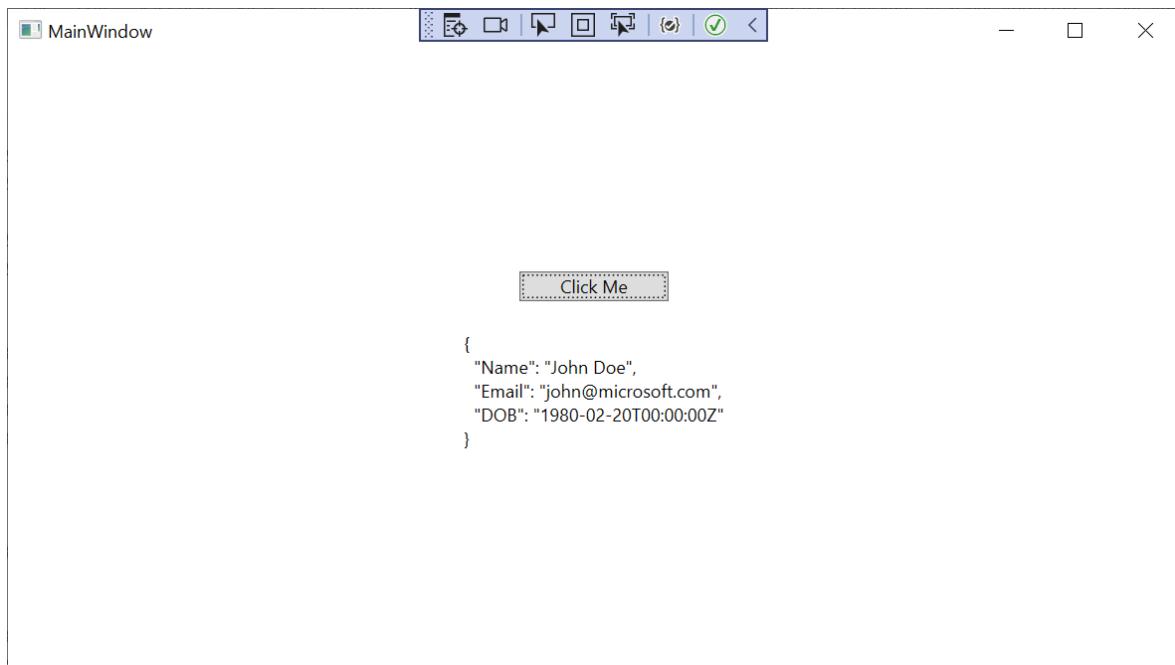
```
using Newtonsoft.Json;
```

4. To build and run the app, press F5 or select **Debug > Start Debugging**.

The following window appears:



5. Select the **Click Me** button to see the contents of the `TextBlock` object replaced with JSON text.



Related video

- [Install and Use a NuGet Package with Visual Studio](#)

- Find more NuGet videos on [Channel 9](#) and [YouTube](#).

See also

For more information about NuGet, see the following articles:

- [What is NuGet?](#)
- [Package consumption workflow](#)
- [Find and choose packages](#)
- [Package references in project files](#)
- [Install and use a package using the .NET CLI.](#)
- [Newtonsoft.Json package](#)

Next steps

Congratulations on installing and using your first NuGet package. Advance to the next article to learn more about installing and managing NuGet packages.

[Install and manage packages using the NuGet Package Manager](#)

[Install and manage packages using the Package Manager Console](#)

Quickstart: Create and publish a NuGet package using Visual Studio (Windows only)

Article • 08/21/2023

With Microsoft Visual Studio, you can create a NuGet package from a .NET class library, and then publish it to nuget.org using a CLI tool.

The quickstart is for Windows users only. If you're using Visual Studio for Mac, see [Create a NuGet package from existing library projects](#) or use the [.NET CLI](#).

Prerequisites

- Install Visual Studio 2022 for Windows with a .NET Core-related workload.

You can install the 2022 Community edition for free from visualstudio.microsoft.com, or use the Professional or Enterprise edition.

Visual Studio 2017 and later automatically includes NuGet capabilities when you install a .NET-related workload.

- Install the .NET CLI, if it's not already installed.

For Visual Studio 2017 and later, the .NET CLI is automatically installed with any .NET Core-related workload. Otherwise, install the [.NET Core SDK](#) to get the .NET CLI. The .NET CLI is required for .NET projects that use the [SDK-style format](#) (SDK attribute). The default .NET class library template in Visual Studio 2017 and later uses the SDK attribute.

Important

If you're working with a non-SDK-style project, follow the procedures in [Create and publish a .NET Framework package \(Visual Studio\)](#) instead to create and publish the package. For this article, the .NET CLI is recommended. Although you can publish any NuGet package using the NuGet CLI, some of the steps in this article are specific to SDK-style projects and the .NET CLI. The NuGet CLI is used for [non-SDK-style projects](#) (typically .NET Framework).

- Register for a free account on nuget.org if you don't have one already. You must register and confirm the account before you can upload a NuGet package.
- Install the NuGet CLI by downloading it from nuget.org. Add the `nuget.exe` file to a suitable folder, and add that folder to your PATH environment variable.

Create a class library project

You can use an existing .NET Class Library project for the code you want to package, or create one as follows:

1. In Visual Studio, select **File > New > Project**.
2. In the **Create a new project** window, select **C#, Windows, and Library** in the dropdown lists.
3. In the resulting list of project templates, select **Class Library** (with the description, *A project for creating a class library that targets .NET or .NET Standard*), and then select **Next**.
4. In the **Configure your new project** window, enter *AppLogger* for the **Project name**, and then select **Next**.
5. In the **Additional information** window, select an appropriate **Framework**, and then select **Create**.

If you're unsure which framework to select, the latest is a good choice, and can be easily changed later. For information about which framework to use, see [When to target .NET 5.0 or .NET 6.0 vs. .NET Standard](#).

6. To ensure the project was created properly, select **Build > Build Solution**. The DLL is found within the **Debug** folder (or **Release** if you build that configuration instead).
7. (Optional) For this quickstart, you don't need to write any additional code for the NuGet package because the template class library is sufficient to create a package. However, if you'd like some functional code for the package, include the following code:

```
C#  
  
namespace AppLogger  
{  
    public class Logger  
    {
```

```

public void Log(string text)
{
    Console.WriteLine(text);
}
}

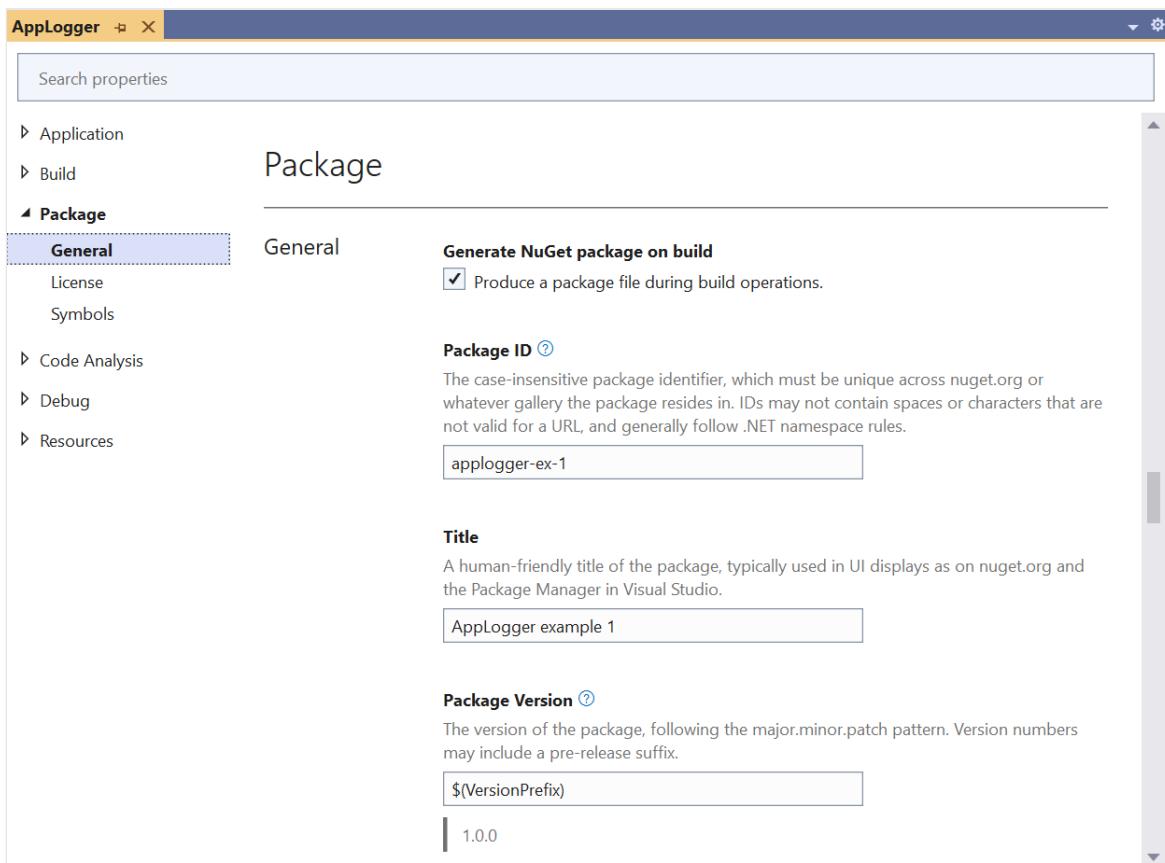
```

Configure package properties

After you've created your project, you can configure the NuGet package properties by following these steps:

1. Select your project in **Solution Explorer**, and then select **Project > <project name> Properties**, where <project name> is the name of your project.
2. Expand the **Package** node, and then select **General**.

The **Package** node appears only for SDK-style projects in Visual Studio. If you're targeting a non-SDK style project (typically .NET Framework), either [migrate the project](#), or see [Create and publish a .NET Framework package](#) for step-by-step instructions.



3. For packages built for public consumption, pay special attention to the **Tags** property, as tags help others find your package and understand what it does.

4. Give your package a unique **Package ID** and fill out any other desired properties.

For a table that shows how MSBuild properties (SDK-style projects) map to `.nuspec` file properties, see [pack targets](#). For a description of `.nuspec` file properties, see the [.nuspec file reference](#). All of these properties go into the `.nuspec` manifest that Visual Studio creates for the project.

Important

You must give the package an identifier that's unique across [nuget.org](#) or whatever host you're using. Otherwise, an error occurs. For this quickstart we recommend including *Sample* or *Test* in the name because the publishing step makes the package publicly visible.

5. (Optional) To see the properties directly in the `AppLogger.csproj` project file, select **Project > Edit Project File**.

The `AppLogger.csproj` tab loads.

This option is available starting in Visual Studio 2017 for projects that use the SDK-style attribute. For earlier Visual Studio versions, you must select **Project > Unload Project** before you can edit the project file.

Run the pack command

To create a NuGet package from your project, follow these steps:

1. Select **Build > Configuration Manager**, and then set the **Active solution configuration** to **Release**.

2. Select the `AppLogger` project in **Solution Explorer**, then select **Pack**.

Visual Studio builds the project and creates the `.nupkg` file.

3. Examine the **Output** window for details, which contains the path to the package file. In this example, the built assembly is in `bin\Release\net6.0` as befits a .NET 6.0 target:

Output

```
1>----- Build started: Project: AppLogger, Configuration: Release Any CPU -----
1>AppLogger ->
d:\proj\AppLogger\AppLogger\bin\Release\net6.0\AppLogger.dll
1>Successfully created package
```

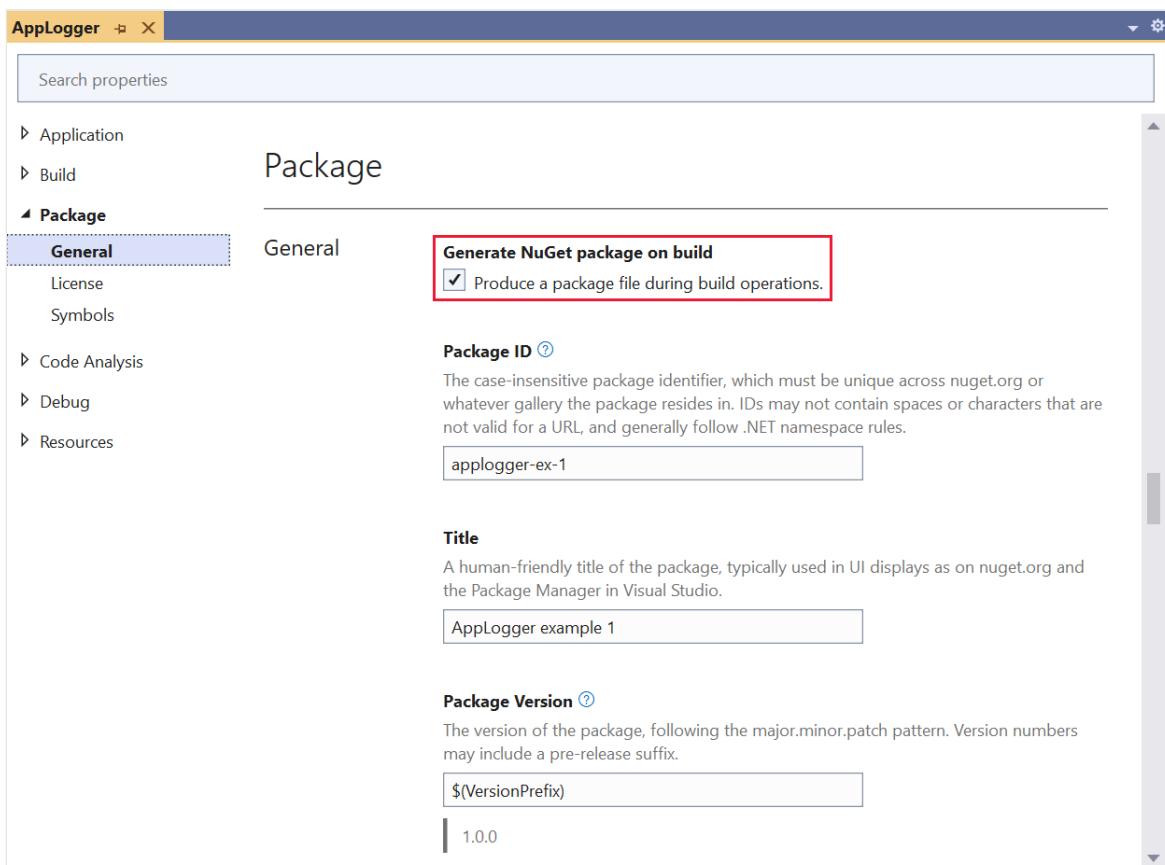
```
'd:\proj\AppLogger\AppLogger\bin\Release\AppLogger.1.0.0.nupkg'.
=====
Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped
=====
```

4. If you don't see the **Pack** command on the menu, your project is probably not an SDK-style project, and you need to use the NuGet CLI. Either [migrate the project](#) and use .NET CLI, or see [Create and publish a .NET Framework package](#) for step-by-step instructions.

(Optional) Generate package on build

You can configure Visual Studio to automatically generate the NuGet package when you build the project:

1. Select your project in **Solution Explorer**, and then select **Project > <project name> Properties**, where <project name> is the name of your project (AppLogger in this case).
2. Expand the **Package** node, select **General**, and then select **Generate NuGet package on build**.



ⓘ Note

When you automatically generate the package, the extra time to pack increases the overall build time for your project.

(Optional) Pack with MSBuild

As an alternative to using the **Pack** menu command, NuGet 4.x+ and MSBuild 15.1+ supports a `pack` target when the project contains the necessary package data:

1. With your project open in **Solution Explorer**, open a command prompt by selecting **Tools > Command Line > Developer Command Prompt**.

The command prompt opens in your project directory.

2. Run the following command: `msbuild -t:pack`.

For more information, see [Create a package using MSBuild](#).

Publish the package

After you've created a `.nupkg` file, publish it to [nuget.org](#) by using either the .NET CLI or the NuGet CLI, along with an API key acquired from [nuget.org](#).

Note

- Nuget.org scans all uploaded packages for viruses and rejects the packages if it finds any viruses. Nuget.org also scans all existing listed packages periodically.
- Packages you publish to [nuget.org](#) are publicly visible to other developers unless you unlist them. To host packages privately, see [Host your own NuGet feeds](#).

Acquire your API key

Before you publish your NuGet package, create an API key:

1. [Sign into your nuget.org account](#) or [create an account](#) if you don't have one already.
2. Select your user name at upper right, and then select **API Keys**.

3. Select **Create**, and provide a name for your key.

4. Under **Select Scopes**, select **Push**.

5. Under **Select Packages > Glob Pattern**, enter *****.

6. Select **Create**.

7. Select **Copy** to copy the new key.

⚠ Your API key has been regenerated. Make sure to copy your new API key now using the **Copy** button below. You will not be able to do so again.

 **test-key1**

⌚ Expires in a year | ⚡ Push new packages and package versions

Package owner: NuGet-test
Glob pattern: *

Copy   

Important

- Always keep your API key a secret. The API key is like a password that allows anyone to manage packages on your behalf. Delete or regenerate your API key if it's accidentally revealed.
- Save your key in a secure location, because you can't copy the key again later. If you return to the API key page, you need to regenerate the key to copy it. You can also remove the API key if you no longer want to push packages.

Scoping lets you create separate API keys for different purposes. Each key has an expiration timeframe, and you can scope the key to specific packages or glob patterns. You also scope each key to specific operations: Push new packages and package versions, push only new package versions, or unlist.

Through scoping, you can create API keys for different people who manage packages for your organization so they have only the permissions they need.

For more information, see [scoped API keys](#).

Publish with the .NET CLI or NuGet CLI

Each of the following CLI tools allows you to push a package to the server and publish it. Select the tab for your CLI tool, either **.NET CLI** or **NuGet CLI**.

.NET CLI

Using the .NET CLI (`dotnet.exe`) is the recommended alternative to using the NuGet CLI.

From the folder that contains the `.nupkg` file, run the following command. Specify your `.nupkg` filename, and replace the key value with your API key.

.NET CLI

```
dotnet nuget push Contoso.08.28.22.001.Test.1.0.0.nupkg --api-key
qz2jga8pl3dvn2akksyquwcs9ygggg4exypy3bhxy6w6x6 --source
https://api.nuget.org/v3/index.json
```

The output shows the results of the publishing process:

Output

```
Pushing Contoso.08.28.22.001.Test.1.0.0.nupkg to
'https://www.nuget.org/api/v2/package'...
PUT https://www.nuget.org/api/v2/package/
warn : All published packages should have license information specified.
Learn more: https://aka.ms/nuget/authoring-best-practices#licensing.
Created https://www.nuget.org/api/v2/package/ 1221ms
Your package was pushed.
```

For more information, see [dotnet nuget push](#).

ⓘ Note

If you want to avoid your test package being live on nuget.org, you can push to the nuget.org test site at <https://int.nugettest.org>. Note that packages uploaded to int.nugettest.org might not be preserved.

Publish errors

Errors from the `push` command typically indicate the problem. For example, you might have forgotten to update the version number in your project, so you're trying to publish a package that already exists.

You also see errors if your API key is invalid or expired, or if you try to publish a package using an identifier that already exists on the host. Suppose, for example, the identifier `AppLogger-test` already exists on nuget.org. If you try to publish a package with that identifier, the `push` command gives the following error:

Output

```
Response status code does not indicate success: 403 (The specified API key  
is invalid,  
has expired, or does not have permission to access the specified package.).
```

If you get this error, check that you're using a valid API key that hasn't expired. If you are, the error indicates the package identifier already exists on the host. To fix the error, change the package identifier to be unique, rebuild the project, recreate the `.nupkg` file, and retry the `push` command.

Manage the published package

When your package successfully publishes, you receive a confirmation email. To see the package you just published, on nuget.org, select your user name at upper right, and then select **Manage Packages**.

ⓘ Note

It might take awhile for your package to be indexed and appear in search results where others can find it. During that time, your package appears under **Unlisted Packages**, and the package page shows the following message:

⚠ This package has not been published yet. It will appear in search results and will be available for install/restore after both validation and indexing are complete. Package validation and indexing may take up to an hour. [Read more](#).

You've now published a NuGet package to nuget.org that other developers can use in their projects.

If you've created a package that isn't useful (such as this sample package that was created with an empty class library), or you decide you don't want the package to be visible, you can *unlist* the package to hide it from search results:

1. After the package appears under **Published Packages** on the **Manage Packages** page, select the pencil icon next to the package listing.

✓ Published Packages

1 package / 0 downloads

Package ID	Owners	Signing Owner	Downloads	Latest Version
 Contoso.08.28.22.001.Test	Test	username (0 certificates)	0	1.0.0 

2. On the next page, select **Listing**, deselect the **List in search results** checkbox, and then select **Save**.

✓ Listing

Select version

1.0.0 (Latest) 

List or unlist version

ⓘ You can control how your packages are listed using the checkbox below. As per [policy](#), permanent deletion is not supported as it would break every project depending on the availability of the package. For more assistance, [Contact Support](#).

List in search results.

Unlisted packages cannot be installed directly and do not show up in search results.

Save

The package now appears under **Unlisted Packages** in **Manage Packages** and no longer appears in search results.

ⓘ Note

To avoid your test package being live on nuget.org, you can push to the nuget.org test site at <https://int.nugettest.org>. Note that packages uploaded to int.nugettest.org might not be preserved.

Add a readme or another file

To directly specify files to include in the package, edit the project file and add the `content` property:

XML

```
<ItemGroup>
  <Content Include="readme.txt">
    <Pack>true</Pack>
    <PackagePath>\</PackagePath>
  </Content>
</ItemGroup>
```

In this example, the `Include` property specifies a file named `readme.txt` in the project root. Visual Studio displays the contents of that file as plain text immediately after it installs the package. Readme files aren't displayed for packages installed as dependencies. For example, here's the readme for the `HtmlAgilityPack` package:

Output

```
1 -----
2 ----- Html Agility Pack Nuget Readme -----
3 -----
4
5 ----Silverlight 4 and Windows Phone 7.1+ projects-----
6 To use XPATH features: System.Xml.XPath.dll from the 3 Silverlight 4 SDK
must be referenced.
7 This is normally found at
8 %ProgramFiles(x86)%\Microsoft SDKs\Microsoft
SDKs\Silverlight\v4.0\Libraries\Client
9 or
10 %ProgramFiles%\Microsoft SDKs\Microsoft
SDKs\Silverlight\v4.0\Libraries\Client
11
12 ----Silverlight 5 projects-----
13 To use XPATH features: System.Xml.XPath.dll from the Silverlight 5 SDK
must be referenced.
14 This is normally found at
15 %ProgramFiles(x86)%\Microsoft SDKs\Microsoft
SDKs\Silverlight\v5.0\Libraries\Client
16 or
17 %ProgramFiles%\Microsoft SDKs\Microsoft
SDKs\Silverlight\v5.0\Libraries\Client
```

Note

If you only add `readme.txt` at the project root without including it in the `content` property of the project file, it won't be included in the package.

Related video

<https://learn.microsoft.com/shows/NuGet-101/Create-and-Publish-a-NuGet-Package-with-Visual-Studio-4-of-5/player>

Find more NuGet videos on [Channel 9](#) and [YouTube](#).

Congratulations on creating a NuGet package by using a Visual Studio .NET class library. Advance to the next article to learn how to create a NuGet package with the Visual Studio .NET Framework.

Create a package using the NuGet CLI

To explore more that NuGet has to offer, see the following articles:

- [Create a NuGet package](#)
- [Publish a package](#)
- [Build a prerelease package](#)
- [Support multiple .NET Framework versions](#)
- [Package versioning](#)
- [Create localized NuGet packages](#)
- [Porting to .NET Core from .NET Framework](#)

Tutorial: Create a .NET console application using Visual Studio Code

Article • 09/30/2023

This tutorial shows how to create and run a .NET console application by using Visual Studio Code and the .NET CLI. Project tasks, such as creating, compiling, and running a project are done by using the .NET CLI. You can follow this tutorial with a different code editor and run commands in a terminal if you prefer.

Prerequisites

- [Visual Studio Code](#) with the [C# extension](#) installed.

If you have the [C# Dev Kit extension](#) installed, uninstall or disable it. It isn't used by this tutorial series.

For information about how to install extensions on Visual Studio Code, see [VS Code Extension Marketplace](#).

- The [.NET 8 SDK](#).

Create the app

Create a .NET console app project named "HelloWorld".

1. Start Visual Studio Code.
2. Select **File > Open Folder** (**File > Open...** on macOS) from the main menu.
3. In the **Open Folder** dialog, create a *HelloWorld* folder and select it. Then click **Select Folder (Open** on macOS).

The folder name becomes the project name and the namespace name by default. You'll add code later in the tutorial that assumes the project namespace is `HelloWorld`.

4. In the **Do you trust the authors of the files in this folder?** dialog, select **Yes, I trust the authors**. You can trust the authors because this folder only has files generated by .NET and added or modified by you.

5. Open the **Terminal** in Visual Studio Code by selecting **View > Terminal** from the main menu.

The **Terminal** opens with the command prompt in the *HelloWorld* folder.

6. In the **Terminal**, enter the following command:

```
.NET CLI  
dotnet new console --framework net8.0 --use-program-main
```

Open the *Program.cs* file to see the simple application created by the template:

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

The first time you open a *.cs* file, Visual Studio Code prompts you to add assets to build and debug your app. Select **Yes**, and Visual Studio Code creates a *.vscode* folder with *launch.json* and *tasks.json* files.

ⓘ Note

If you don't get the prompt, or if you accidentally dismiss it without selecting **Yes**, do the following steps to create *launch.json* and *tasks.json*:

- Select **Run > Add Configuration** from the menu.
- Select **.NET 5+ and .NET Core** at the **Select environment** prompt.

The code 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. The code in `Main` calls the `Console.WriteLine(String)` method to display a message in the console window.

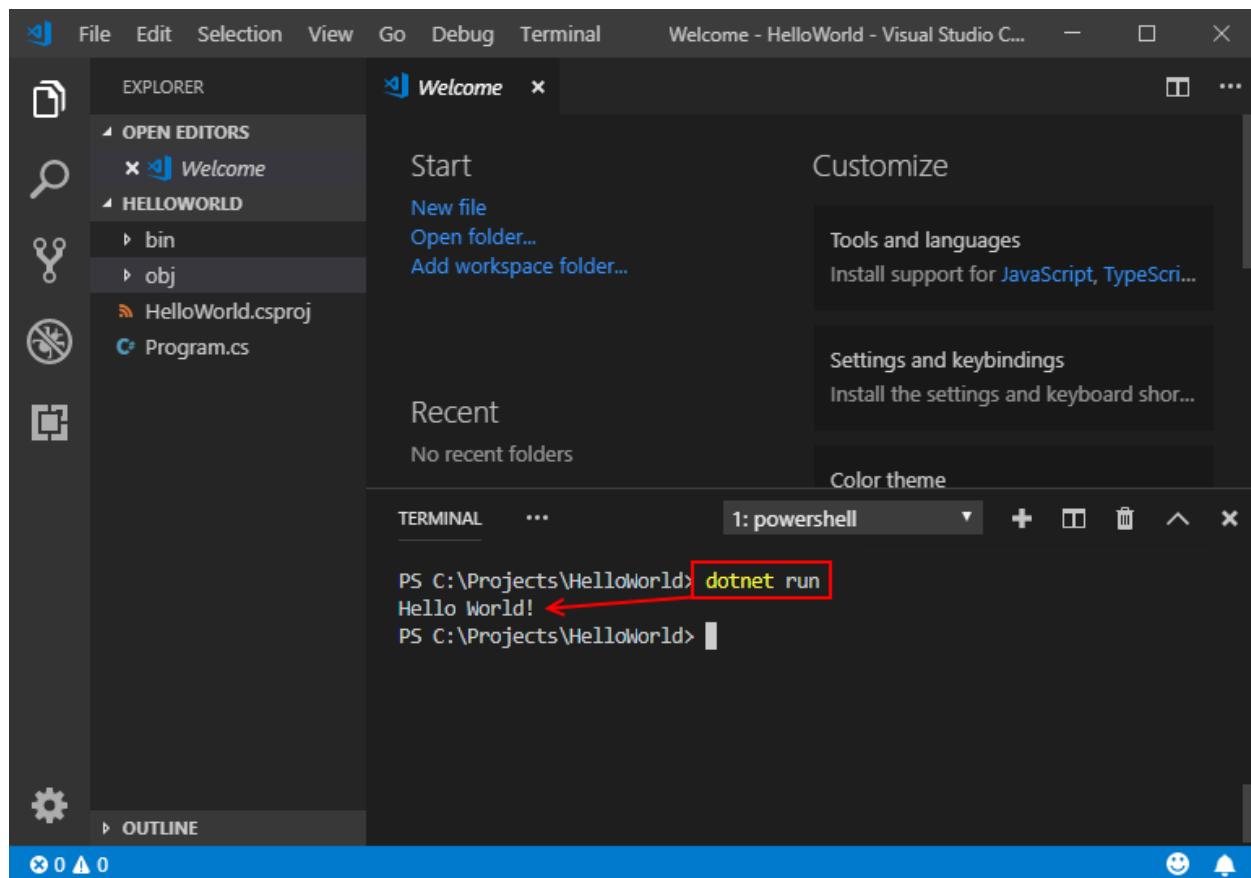
C# has a feature named [top-level statements](#) that lets you omit the `Program` class and the `Main` method. This tutorial doesn't use this feature. Whether you use it in your programs is a matter of style preference. In the `dotnet new` command that created the project, the `--use-program-main` option prevented top-level statements from being used.

Run the app

Run the following command in the **Terminal**:

```
.NET CLI
dotnet run
```

The program displays "Hello, World!" and ends.



Enhance the app

Enhance the application to prompt the user for their name and display it along with the date and time.

1. Open `Program.cs`.

2. Replace the contents of the `Main` method in `Program.cs`, which is the line that calls `Console.WriteLine`, with the following code:

C#

```
Console.WriteLine("What is your name?");
var name = Console.ReadLine();
var currentDate = DateTime.Now;
Console.WriteLine(${Environment.NewLine}Hello, {name}, on
{currentDate:d} at {currentDate:t}!");
Console.WriteLine(${Environment.NewLine}Press any key to exit...");
Console.ReadKey(true);
```

This code displays a prompt in the console window and waits until the user enters a string followed by the `Enter` key. It stores this string in 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`. And it displays these values in the console window. Finally, it displays a prompt in the console window and calls the `Console.ReadKey(Boolean)` method to wait for user input.

`NewLine` is a platform-independent and language-independent way to represent a line break. It's the same as `\n` in C#.

The dollar sign (\$) in front of a string lets you put expressions such as variable names in curly braces in the string. The expression value is inserted into the string in place of the expression. This syntax is referred to as [interpolated strings](#).

3. Save your changes.

 **Important**

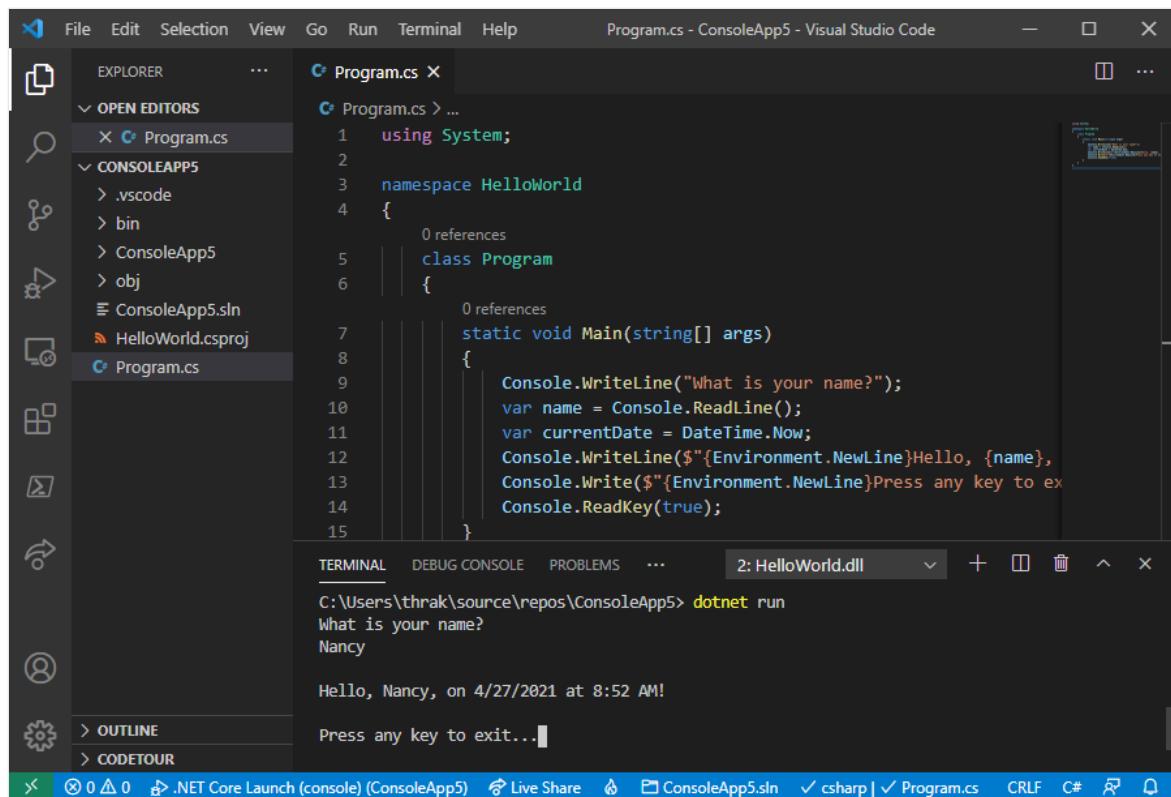
In Visual Studio Code, you have to explicitly save changes. Unlike Visual Studio, file changes are not automatically saved when you build and run an app.

4. Run the program again:

.NET CLI

```
dotnet run
```

5. Respond to the prompt by entering a name and pressing the `Enter` key.



```
1  using System;
2
3  namespace HelloWorld
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("What is your name?");
10             var name = Console.ReadLine();
11             var currentDate = DateTime.Now;
12             Console.WriteLine($"{Environment.NewLine}Hello, {name},");
13             Console.WriteLine($"{Environment.NewLine}Press any key to exit...");
14             Console.ReadKey(true);
15         }
16     }
17 }
```

C:\Users\thrak\source\repos\ConsoleApp5> dotnet run
What is your name?
Nancy

Hello, Nancy, on 4/27/2021 at 8:52 AM!

Press any key to exit...

6. Press any key to exit the program.

Additional resources

- [Setting up Visual Studio Code](#)

Next steps

In this tutorial, you created a .NET console application. In the next tutorial, you debug the app.

[Debug a .NET console application using Visual Studio Code](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Debug a .NET console application using Visual Studio Code

Article • 09/07/2023

This tutorial introduces the debugging tools available in Visual Studio Code for working with .NET apps.

Prerequisites

- This tutorial works with the console app that you create in [Create a .NET console application using Visual Studio Code](#).

Use Debug build configuration

Debug and *Release* are .NET's built-in build configurations. You use the *Debug* build configuration for debugging and the *Release* configuration for the final release distribution.

In the *Debug* configuration, a program compiles with full symbolic debug information and no optimization. Optimization complicates debugging, because the relationship between source code and generated instructions is more complex. The *Release* configuration of a program has no symbolic debug information and is fully optimized.

By default, Visual Studio Code launch settings use the *Debug* build configuration, so you don't need to change it before debugging.

1. Start Visual Studio Code.
2. Open the folder of the project that you created in [Create a .NET console application using Visual Studio Code](#).

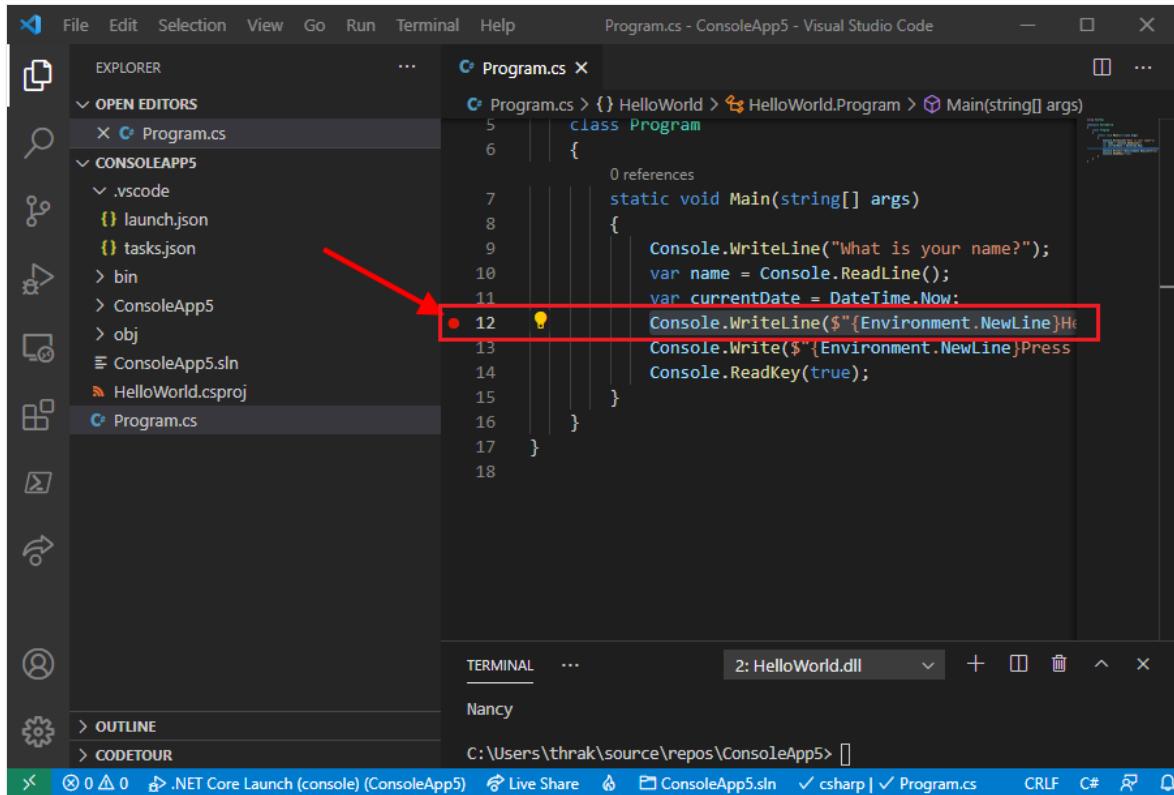
Set a breakpoint

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

1. Open the *Program.cs* file.
2. Set a *breakpoint* on the line that displays the name, date, and time, by clicking in the left margin of the code window. The left margin is to the left of the line

numbers. Other ways to set a breakpoint are by pressing **F9** or choosing **Run > Toggle Breakpoint** from the menu while the line of code is selected.

Visual Studio Code indicates the line on which the breakpoint is set by displaying a red dot in the left margin.



Set up for terminal input

The breakpoint is located after a `Console.ReadLine` method call. The **Debug Console** doesn't accept terminal input for a running program. To handle terminal input while debugging, you can use the integrated terminal (one of the Visual Studio Code windows) or an external terminal. For this tutorial, you use the integrated terminal.

1. The project folder contains a `.vscode` folder. Open the `launch.json` file that's in the `.vscode` folder.
2. In `launch.json`, change the `console` setting from `internalConsole` to `integratedTerminal`:

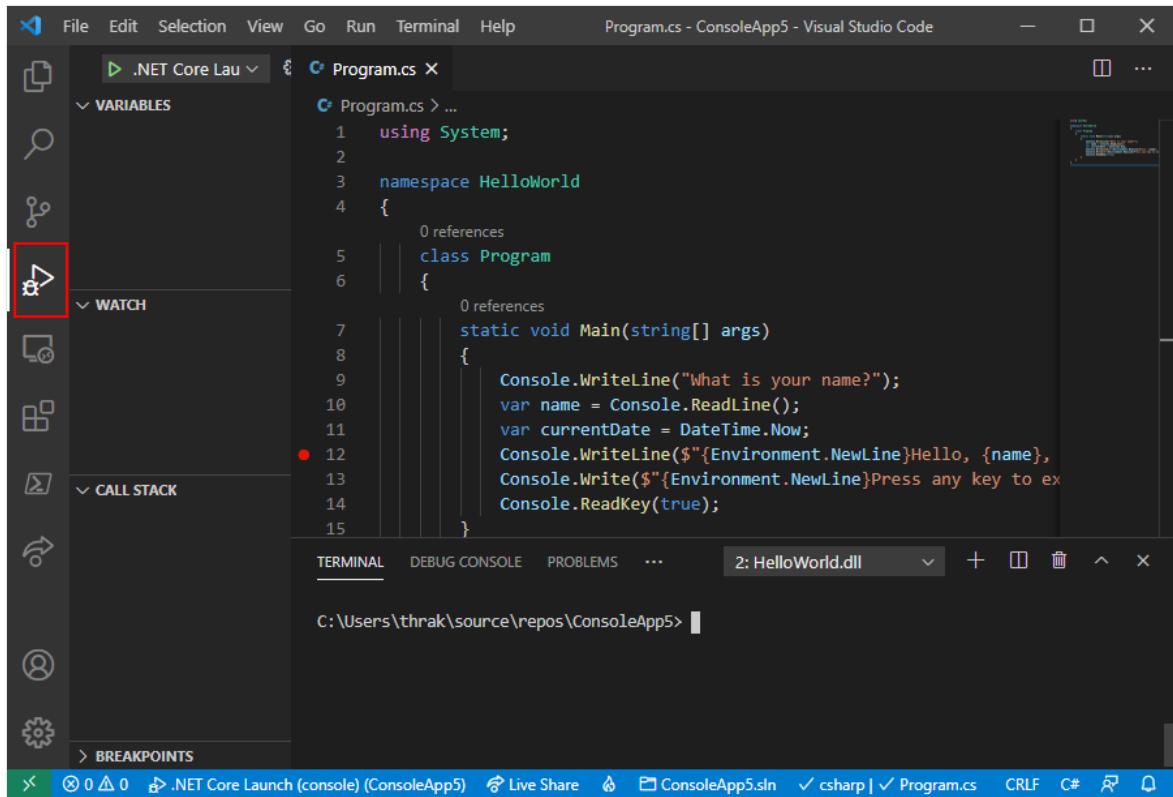
JSON

```
"console": "integratedTerminal",
```

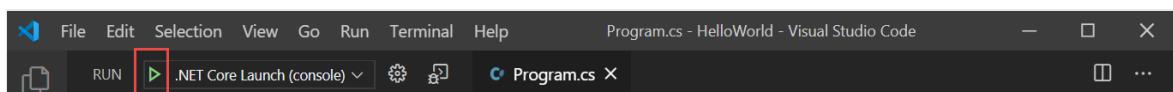
3. Save your changes.

Start debugging

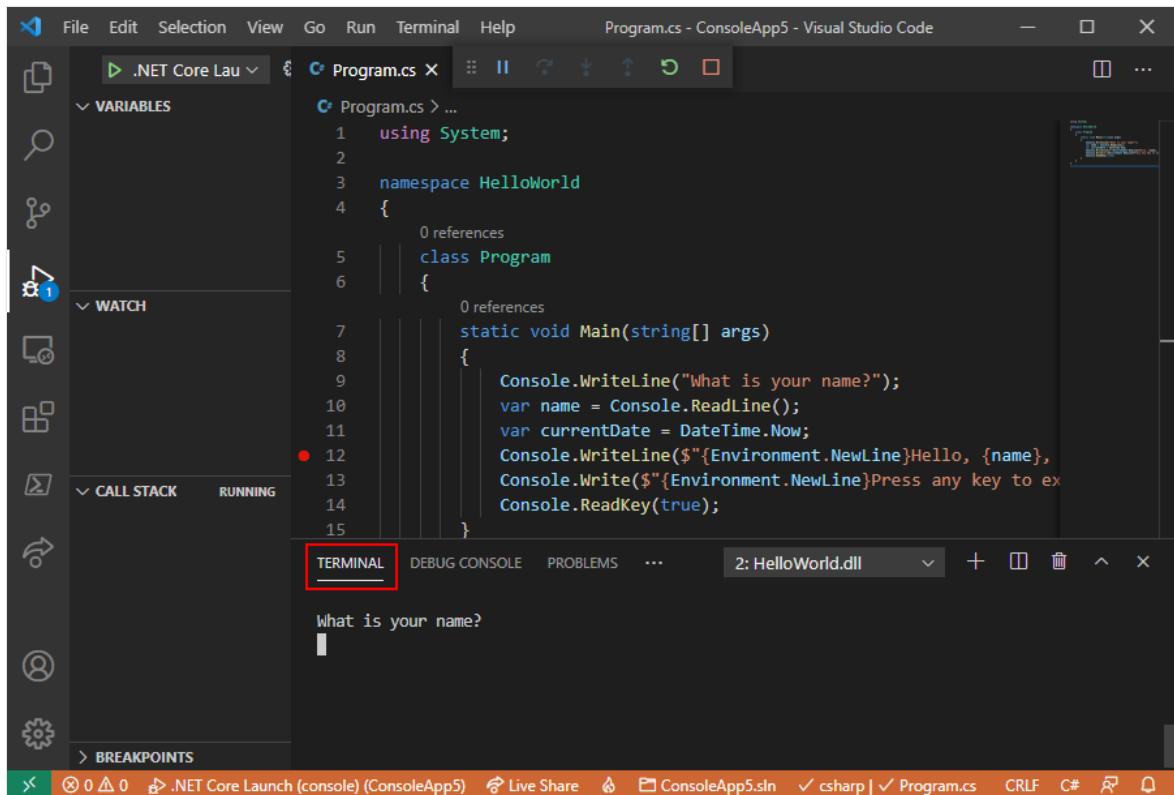
1. Open the Debug view by selecting the Debugging icon on the left side menu.



2. Select the green arrow at the top of the pane, next to **.NET Core Launch (console)**.
Other ways to start the program in debugging mode are by pressing **F5** or choosing **Run > Start Debugging** from the menu.



3. Select the **Terminal** tab to see the "What is your name?" prompt that the program displays before waiting for a response.



A screenshot of Visual Studio Code showing a .NET Core application named "ConsoleApp5". The code in `Program.cs` contains a `static void Main` method that prompts the user for their name and prints a greeting. A red box highlights the `TERMINAL` tab at the bottom of the interface. The terminal window shows the output of the application, which asks for the user's name and prints a greeting.

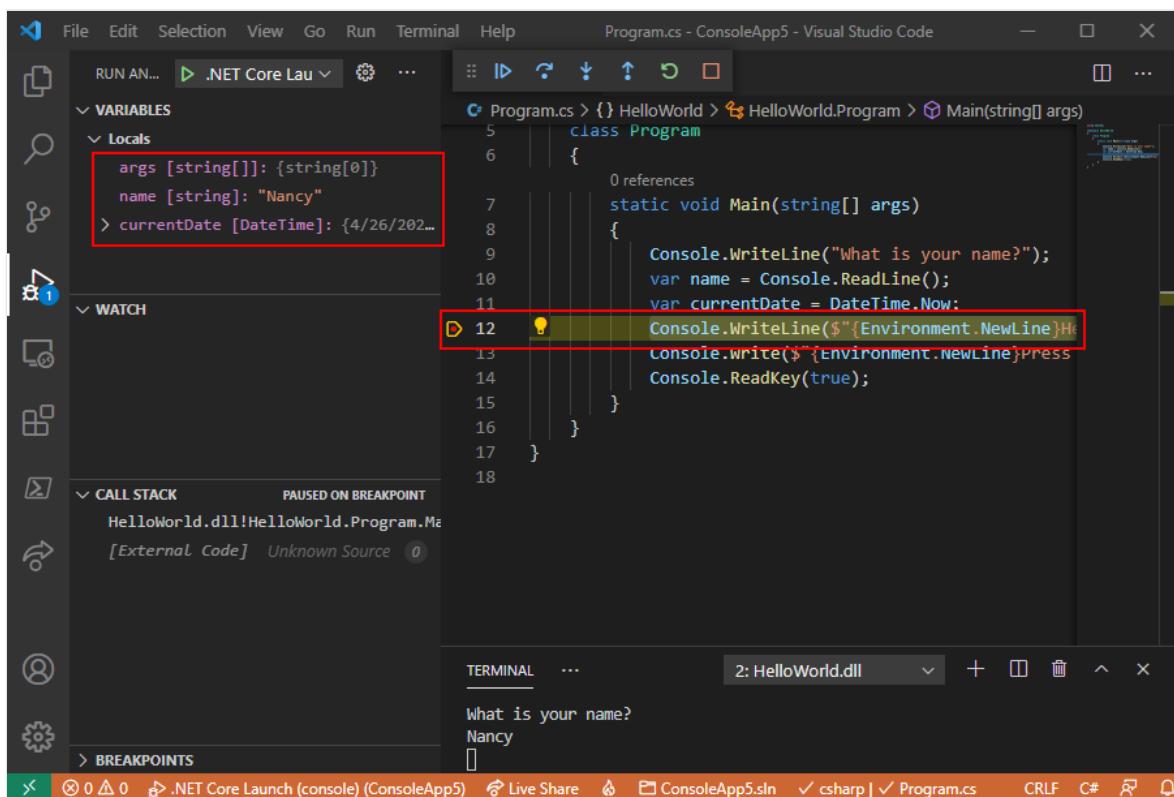
```
1  using System;
2
3  namespace HelloWorld
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("What is your name?");
10             var name = Console.ReadLine();
11             var currentDate = DateTime.Now;
12             Console.WriteLine($"{Environment.NewLine}Hello, {name},");
13             Console.WriteLine($"{Environment.NewLine}Press any key to exit");
14             Console.ReadKey(true);
15         }
16     }
17 }
```

TERMINAL DEBUG CONSOLE PROBLEMS ... 2: HelloWorld.dll + - x

What is your name?

4. Enter a string in the **Terminal** window in response to the prompt for a name, and then press **Enter**.

Program execution stops when it reaches the breakpoint and before the `Console.WriteLine` method runs. The **Locals** section of the **Variables** window displays the values of variables that are defined in the currently running method.



A screenshot of Visual Studio Code showing the application paused on a breakpoint. The **Locals** section of the **Variables** window is highlighted with a red box, showing the values of `args`, `name`, and `currentDate`. The current line of code, `Console.WriteLine("Hello, {name},");`, is also highlighted with a red box. The terminal window shows the application has printed "Hello, Nancy,".

RUN AN... .NET Core Lau ...

File Edit Selection View Go Run Terminal Help Program.cs - ConsoleApp5 - Visual Studio Code

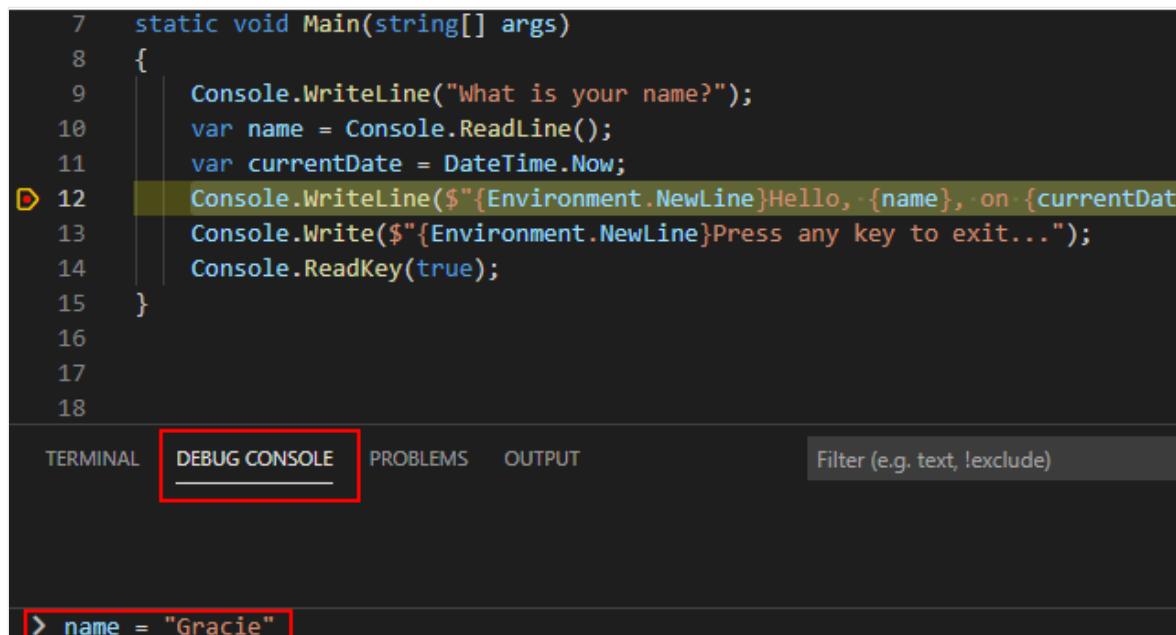
TERMINAL ... 2: HelloWorld.dll + - x

What is your name?
Nancy

Use the Debug Console

The **Debug Console** window lets you interact with the application you're debugging. You can change the value of variables to see how it affects your program.

1. Select the **Debug Console** tab.
2. Enter `name = "Gracie"` at the prompt at the bottom of the **Debug Console** window and press the **Enter** key.



A screenshot of the Visual Studio Code interface. The code editor shows a C# file with the following code:

```
7  static void Main(string[] args)
8  {
9      Console.WriteLine("What is your name?");
10     var name = Console.ReadLine();
11     var currentDate = DateTime.Now;
12     Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate}");
13     Console.WriteLine($"{Environment.NewLine}Press any key to exit...");
14     Console.ReadKey(true);
15 }
```

The line `Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate}");` is highlighted with a yellow background. The **DEBUG CONSOLE** tab is selected in the bottom navigation bar, and the status bar shows the filter `Filter (e.g. text, !exclude)`. In the terminal at the bottom, the command `> name = "Gracie"` is entered, with the entire line highlighted in red.

3. Enter `currentDate = DateTime.Parse("2019-11-16T17:25:00Z").ToUniversalTime()` at the bottom of the **Debug Console** window and press the **Enter** key.

The **Variables** window displays the new values of the `name` and `currentDate` variables.

4. Continue program execution by selecting the **Continue** button in the toolbar. Another way to continue is by pressing **F5**.



5. Select the **Terminal** tab again.

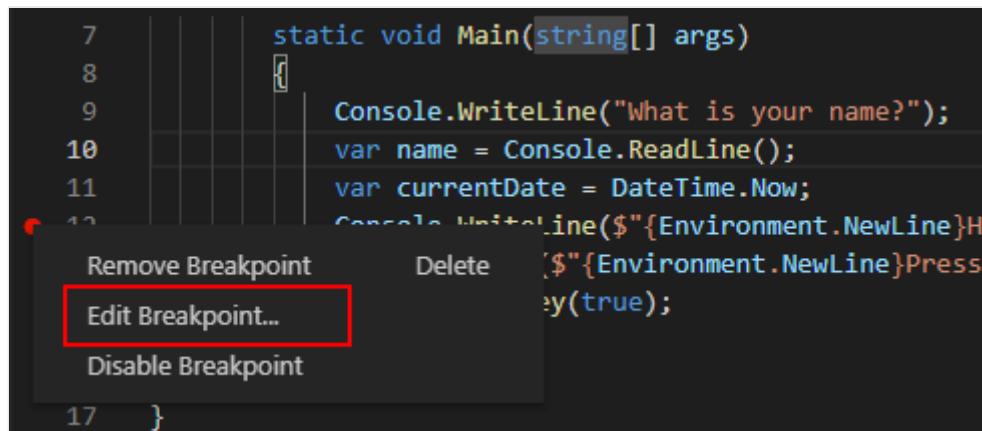
The values displayed in the console window correspond to the changes you made in the **Debug Console**.

6. Press any key to exit the application and stop debugging.

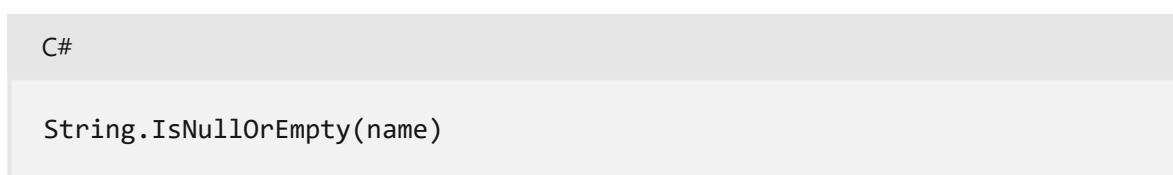
Set a conditional breakpoint

The 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 called a *conditional breakpoint*.

1. Right-click (`ctrl`-click on macOS) on the red dot that represents the breakpoint. In the context menu, select **Edit Breakpoint** to open a dialog that lets you enter a conditional expression.



2. Select **Expression** in the drop-down, enter the following conditional expression, and press **Enter**.





```
7  static void Main(string[] args)
8  {
9      Console.WriteLine("What is your name?");
10     var name = Console.ReadLine();
11     var currentDate = DateTime.Now;
12     Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentD
13     Console.Write($"{Environment.NewLine}Press any key to exit...");
14     Console.ReadKey(true);
15 }
```

Each time the breakpoint is hit, the debugger calls the `String.IsNullOrEmpty(name)` method, and it breaks on this line only if the method call returns `true`.

Instead of a conditional expression, you can specify a *hit count*, which interrupts program execution before a statement is run a specified number of times. Another option is to specify a *filter condition*, which interrupts program execution based on such attributes as a thread identifier, process name, or thread name.

3. Start the program with debugging by pressing `F5`.

4. In the **Terminal** tab, press the `Enter` key when prompted to enter your name.

Because the condition you 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 runs.

The **Variables** window shows that the value of the `name` variable is `""`, or `String.Empty`.

5. Confirm the value is an empty string by entering the following statement at the **Debug Console** prompt and pressing `Enter`. The result is `true`.

```
C#
name == String.Empty
```

6. Select the **Continue** button on the toolbar to continue program execution.

7. Select the **Terminal** tab, and press any key to exit the program and stop debugging.

8. Clear the breakpoint by clicking on the dot in the left margin of the code window. Other ways to clear a breakpoint are by pressing `F9` or choosing **Run > Toggle Breakpoint** from the menu while the line of code is selected.

9. If you get a warning that the breakpoint condition will be lost, select **Remove Breakpoint**.

Step through a program

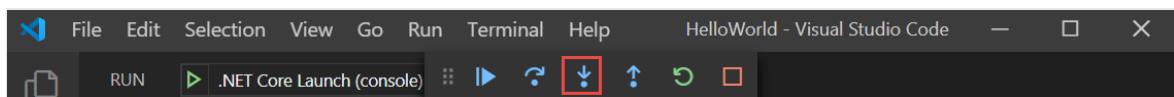
Visual Studio Code also allows you to step line by line through a program and monitor its execution. Ordinarily, you'd set a breakpoint and follow program flow through a small part of your program code. Since this program is small, you can step through the entire program.

1. Set a breakpoint on the opening curly brace of the `Main` method.
2. Press `F5` to start debugging.

Visual Studio Code highlights the breakpoint line.

At this point, the **Variables** window shows that the `args` array is empty, and `name` and `currentDate` have default values.

3. Select **Run > Step Into** or press `F11`.



Visual Studio Code highlights the next line.

4. Select **Run > Step Into** or press `F11`.

Visual Studio Code runs the `console.WriteLine` for the name prompt and highlights the next line of execution. The next line is the `Console.ReadLine` for the `name`. The **Variables** window is unchanged, and the **Terminal** tab shows the "What is your name?" prompt.

5. Select **Run > Step Into** or press `F11`.

Visual Studio highlights the `name` variable assignment. The **Variables** window shows that `name` is still `null`.

6. Respond to the prompt by entering a string in the Terminal tab and pressing `Enter`.

The **Terminal** tab might not display the string you enter while you're entering it, but the `Console.ReadLine` method will capture your input.

7. Select **Run > Step Into** or press `F11`.

Visual Studio Code highlights the `currentDate` variable assignment. The **Variables** window shows the value returned by the call to the `Console.ReadLine` method. The **Terminal** tab displays the string you entered at the prompt.

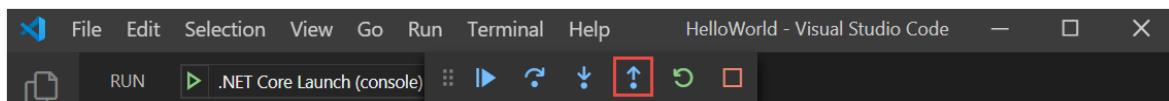
8. Select **Run > Step Into** or press `F11`.

The **Variables** window shows the value of the `currentDate` variable after the assignment from the `DateTime.Now` property.

9. Select **Run > Step Into** or press `F11`.

Visual Studio Code calls the `Console.WriteLine(String, Object, Object)` method. The console window displays the formatted string.

10. Select **Run > Step Out** or press `Shift + F11`.



11. Select the **Terminal** tab.

The terminal displays "Press any key to exit..."

12. Press any key to exit the program.

Use Release build configuration

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

To build and test the Release version of your console application, open the **Terminal** and run the following command:

```
.NET CLI
dotnet run --configuration Release
```

Additional resources

- [Debugging in Visual Studio Code](#)

Next steps

In this tutorial, you used Visual Studio Code debugging tools. In the next tutorial, you publish a deployable version of the app.

[Publish a .NET console application using Visual Studio Code](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Publish a .NET console application using Visual Studio Code

Article • 09/07/2023

This tutorial shows how to publish a console app so that other users can run it. Publishing creates the set of files that are needed to run an application. To deploy the files, copy them to the target machine.

The .NET CLI is used to publish the app, so you can follow this tutorial with a code editor other than Visual Studio Code if you prefer.

Prerequisites

- This tutorial works with the console app that you create in [Create a .NET console application using Visual Studio Code](#).

Publish the app

1. Start Visual Studio Code.
2. Open the *HelloWorld* project folder that you created in [Create a .NET console application using Visual Studio Code](#).
3. Choose **View > Terminal** from the main menu.

The terminal opens in the *HelloWorld* folder.

4. Run the following command:

```
.NET CLI  
dotnet publish --configuration Release
```

The default build configuration is *Debug*, so this command specifies the *Release* build configuration. The output from the *Release* build configuration has minimal symbolic debug information and is fully optimized.

The command output is similar to the following example:

```
Output
```

```
Microsoft (R) Build Engine version 17.8.0+b89cb5fde for .NET
Copyright (C) Microsoft Corporation. All rights reserved.

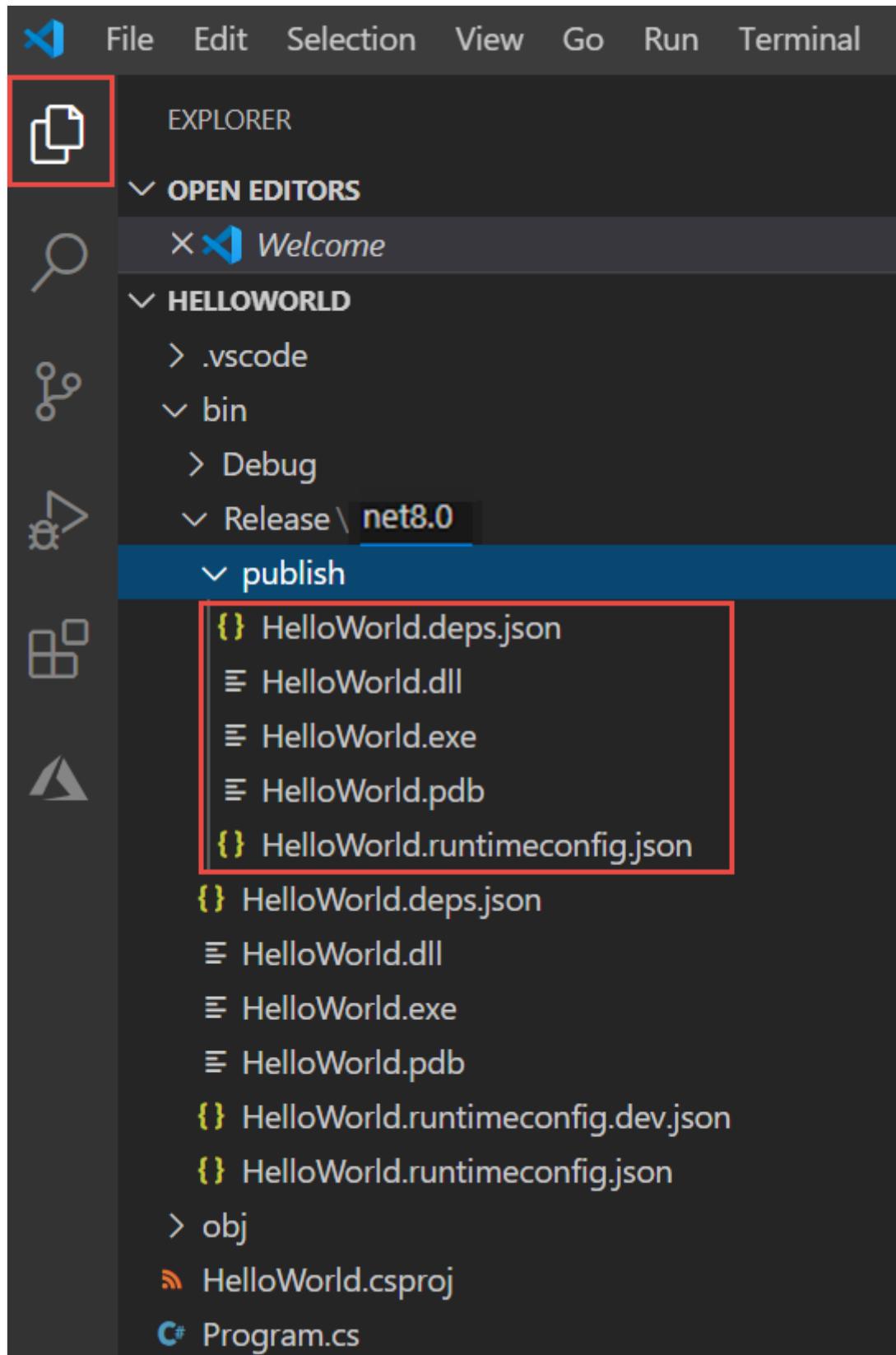
Determining projects to restore...
All projects are up-to-date for restore.
HelloWorld ->
C:\Projects\HelloWorld\bin\Release\net8.0\HelloWorld.dll
HelloWorld -> C:\Projects\HelloWorld\bin\Release\net8.0\publish\
```

Inspect the files

By default, the publishing process creates a framework-dependent deployment, which is a type of deployment where the published application runs on a machine that has the .NET runtime installed. To run the published app you can use the executable file or run the `dotnet HelloWorld.dll` command from a command prompt.

In the following steps, you'll look at the files created by the publish process.

1. Select the **Explorer** in the left navigation bar.
2. Expand *bin/Release/net7.0/publish*.



As the image shows, the published output includes the following files:

- *HelloWorld.deps.json*

This is the application's runtime dependencies file. It defines the .NET components and the libraries (including the dynamic link library that contains your application) needed to run the app. For more information, see [Runtime configuration files](#).

- *HelloWorld.dll*

This is the [framework-dependent deployment](#) version of the application. To run this dynamic link library, enter `dotnet HelloWorld.dll` at a command prompt. This method of running the app works on any platform that has the .NET runtime installed.

- *HelloWorld.exe* (*HelloWorld* on Linux or macOS.)

This is the [framework-dependent executable](#) version of the application. The file is operating-system-specific.

- *HelloWorld.pdb* (optional for deployment)

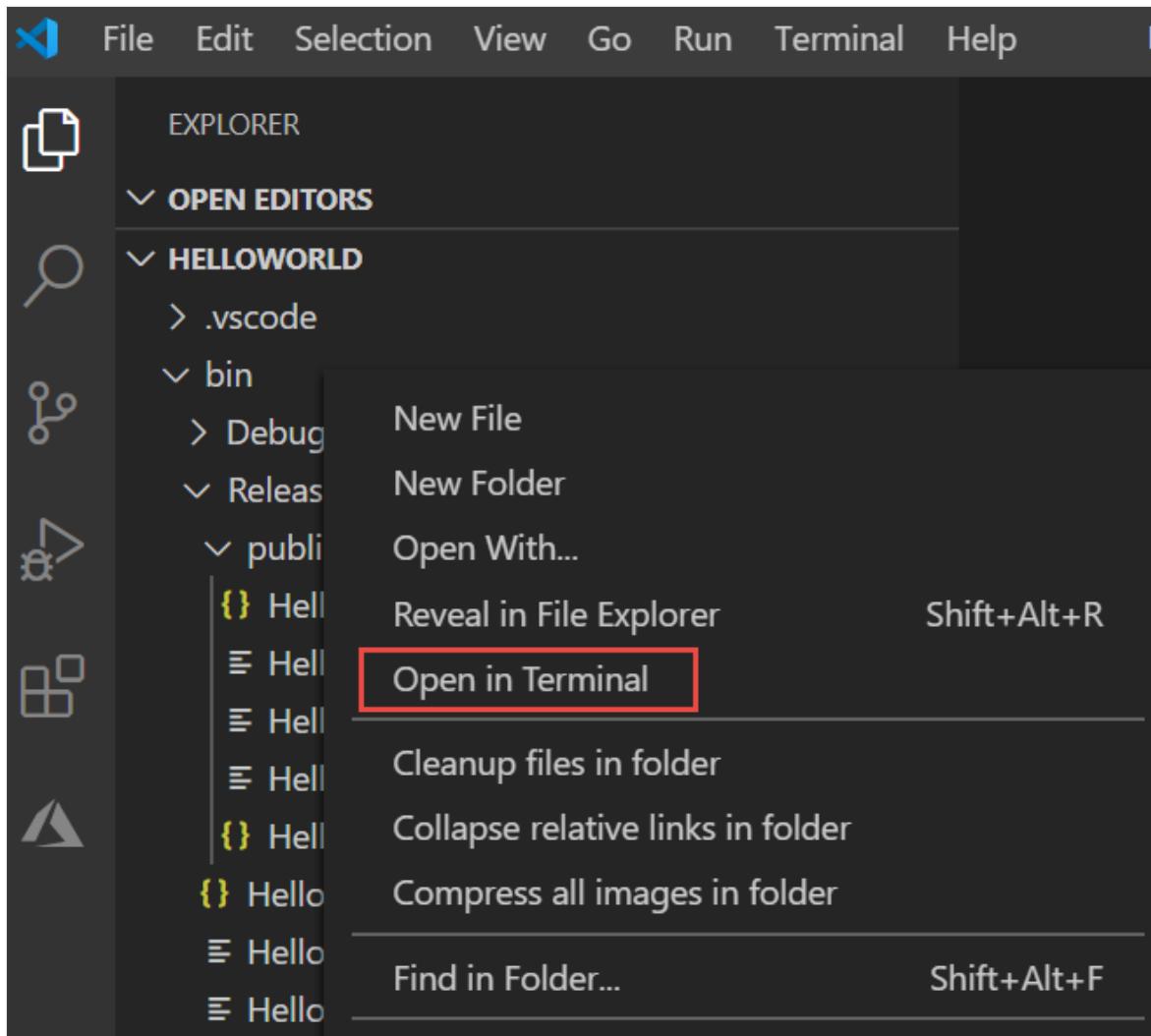
This is the debug symbols file. You aren't required to deploy this file along with your application, although you should save it in the event that you need to debug the published version of your application.

- *HelloWorld.runtimeconfig.json*

This is the application's runtime configuration file. It identifies the version of .NET that your application was built to run on. You can also add configuration options to it. For more information, see [.NET runtime configuration settings](#).

Run the published app

1. In **Explorer**, right-click the *publish* folder (`Ctrl`-click on macOS), and select **Open in Integrated Terminal**.



2. On Windows or Linux, run the app by using the executable.

a. On Windows, enter `.\HelloWorld.exe` and press `Enter`.

b. On Linux, enter `./HelloWorld` and press `Enter`.

c. Enter a name in response to the prompt, and press any key to exit.

3. On any platform, run the app by using the `dotnet` command:

a. Enter `dotnet HelloWorld.dll` and press `Enter`.

b. Enter a name in response to the prompt, and press any key to exit.

Additional resources

- [.NET application deployment](#)
- [Publish .NET apps with the .NET CLI](#)
- [dotnet publish](#)
- [Use the .NET SDK in continuous integration \(CI\) environments](#)

Next steps

In this tutorial, you published a console app. In the next tutorial, you create a class library.

[Create a .NET class library using Visual Studio Code](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

.NET is an open source project. Select a link to provide feedback:

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Create a .NET class library using Visual Studio Code

Article • 09/07/2023

In this tutorial, you create a simple utility library that contains a single string-handling method.

A *class library* defines types and methods that are called by an application. If the library targets .NET Standard 2.0, it can be called by any .NET implementation (including .NET Framework) that supports .NET Standard 2.0. If the library targets .NET 8, it can be called by any application that targets .NET 8. This tutorial shows how to target .NET 8.

When you create a class library, you can distribute it as a third-party component or as a bundled component with one or more applications.

Prerequisites

- [Visual Studio Code](#) with the [C# extension](#) installed.

If you have the [C# Dev Kit extension](#) installed, uninstall or disable it. It isn't used by this tutorial series.

For information about how to install extensions on Visual Studio Code, see [VS Code Extension Marketplace](#).

- The [.NET 8 SDK](#).

Create a solution

Start by creating a blank solution to put the class library project in. A solution serves as a container for one or more projects. You'll add additional, related projects to the same solution.

1. Start Visual Studio Code.
2. Select **File > Open Folder (Open... on macOS)** from the main menu
3. In the **Open Folder** dialog, create a *ClassLibraryProjects* folder and click **Select Folder (Open on macOS)**.

4. Open the **Terminal** in Visual Studio Code by selecting **View > Terminal** from the main menu.

The **Terminal** opens with the command prompt in the *ClassLibraryProjects* folder.

5. In the **Terminal**, enter the following command:

```
.NET CLI  
dotnet new sln
```

The terminal output looks like the following example:

```
Output  
The template "Solution File" was created successfully.
```

Create a class library project

Add a new .NET class library project named "StringLibrary" to the solution.

1. In the terminal, run the following command to create the library project:

```
.NET CLI  
dotnet new classlib -o StringLibrary
```

The `-o` or `--output` command specifies the location to place the generated output.

The terminal output looks like the following example:

```
Output  
The template "Class library" was created successfully.  
Processing post-creation actions...  
Running 'dotnet restore' on StringLibrary\StringLibrary.csproj...  
  Determining projects to restore...  
  Restored  
C:\Projects\ClassLibraryProjects\StringLibrary\StringLibrary.csproj (in  
328 ms).  
Restore succeeded.
```

2. Run the following command to add the library project to the solution:

.NET CLI

```
dotnet sln add StringLibrary/StringLibrary.csproj
```

The terminal output looks like the following example:

Output

```
Project `StringLibrary\StringLibrary.csproj` added to the solution.
```

3. Check to make sure that the library targets .NET 8. In Explorer, open *StringLibrary/StringLibrary.csproj*.

The `TargetFramework` element shows that the project targets .NET 8.0.

XML

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

  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
  </PropertyGroup>

</Project>
```

4. Open *Class1.cs* and replace the code with the following code.

C#

```
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`. This method 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.

`StartsWithUpper` is implemented as an [extension method](#) so that you can call it as if it were a member of the [String](#) class.

5. Save the file.
6. Run the following command to build the solution and verify that the project compiles without error.

```
.NET CLI
```

```
dotnet build
```

The terminal output looks like the following example:

```
Output
```

```
Microsoft (R) Build Engine version 17.8.0+b89cb5fde for .NET
Copyright (C) Microsoft Corporation. All rights reserved.

Determining projects to restore...
All projects are up-to-date for restore.

StringLibrary ->
C:\Projects\ClassLibraryProjects\StringLibrary\bin\Debug\net8.0\StringL
ibrary.dll
Build succeeded.

0 Warning(s)
0 Error(s)
Time Elapsed 00:00:02.78
```

Add a console app to the solution

Add a console application that uses the class library. The app will prompt the user to enter a string and report whether the string begins with an uppercase character.

1. In the terminal, run the following command to create the console app project:

```
.NET CLI
```

```
dotnet new console -o Showcase
```

The terminal output looks like the following example:

```
Output
```

```
The template "Console Application" was created successfully.  
Processing post-creation actions...  
Running 'dotnet restore' on Showcase\ShowCase.csproj...  
Determining projects to restore...  
Restored C:\Projects\ClassLibraryProjects>ShowCase>ShowCase.csproj  
(in 210 ms).  
Restore succeeded.
```

2. Run the following command to add the console app project to the solution:

.NET CLI

```
dotnet sln add Showcase/ShowCase.csproj
```

The terminal output looks like the following example:

Output

```
Project `ShowCase>ShowCase.csproj` added to the solution.
```

3. Open *ShowCase/Program.cs* and replace all of the code with the following code.

C#

```
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}");
            Console.WriteLine("Begins with uppercase? " +
                $"{(input.StartsWithUpper() ? "Yes" : "No")}");
            Console.WriteLine();
            row += 4;
        } 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($"{Environment.NewLine}Press <Enter> only
to exit; otherwise, enter a string and press <Enter>:
{Environment.NewLine}");
            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's 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 ends, and the console window closes.

4. Save your changes.

Add a project reference

Initially, the new console app project doesn't have access to the class library. To allow it to call methods in the class library, create a project reference to the class library project.

1. Run the following command:

```
.NET CLI
dotnet add ShowCase/ShowCase.csproj reference
StringLibrary/StringLibrary.csproj
```

The terminal output looks like the following example:

```
Output
Reference `..\StringLibrary\StringLibrary.csproj` added to the project.
```

Run the app

1. Run the following command in the terminal:

.NET CLI

```
dotnet run --project ShowCase/ShowCase.csproj
```

2. Try out the program by entering strings and pressing `Enter`, then press `Enter` to exit.

The terminal output looks like the following example:

Output

```
Press <Enter> only to exit; otherwise, enter a string and press <Enter>:
```

```
A string that starts with an uppercase letter
```

```
Input: A string that starts with an uppercase letter
```

```
Begins with uppercase? : Yes
```

```
a string that starts with a lowercase letter
```

```
Input: a string that starts with a lowercase letter
```

```
Begins with uppercase? : No
```

Additional resources

- [Develop libraries with the .NET CLI](#)
- [.NET Standard versions and the platforms they support.](#)

Next steps

In this tutorial, you created a solution, added a library project, and added a console app project that uses the library. In the next tutorial, you add a unit test project to the solution.

[Test a .NET class library with .NET using Visual Studio Code](#)

 Collaborate with us on
GitHub

The source for this content can
be found on GitHub, where you

.NET

[.NET feedback](#)

The .NET documentation is open
source. Provide feedback here.

can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Test a .NET class library using Visual Studio Code

Article • 09/07/2023

This tutorial shows how to automate unit testing by adding a test project to a solution.

Prerequisites

- This tutorial works with the solution that you create in [Create a .NET class library using Visual Studio Code](#).

Create a unit test project

Unit tests provide automated software testing during your development and publishing. The testing framework that you use in this tutorial is MSTest. [MSTest](#) is one of three test frameworks you can choose from. The others are [xUnit](#) and [nUnit](#).

1. Start Visual Studio Code.
2. Open the `ClassLibraryProjects` solution you created in [Create a .NET class library using Visual Studio Code](#).
3. Create a unit test project named "StringLibraryTest".

```
.NET CLI
dotnet new mstest -o StringLibraryTest
```

The project template creates a `UnitTest1.cs` file with the following code:

```
C#
namespace StringLibraryTest;

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

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

- It applies the `TestClassAttribute` attribute to the `UnitTest1` class.
- It applies the `TestMethodAttribute` attribute to define `TestMethod1`.
- It imports the `Microsoft.VisualStudio.TestTools.UnitTesting` namespace, which contains the types used for unit testing. The namespace is imported via a `global using` directive in `GlobalUsings.cs`.

Each method tagged with `[TestMethod]` in a test class tagged with `[TestClass]` is run automatically when the unit test is invoked.

4. Add the test project to the solution.

.NET CLI

```
dotnet sln add StringLibraryTest/StringLibraryTest.csproj
```

Add a project reference

For the test project to work with the `StringLibrary` class, add a reference in the `StringLibraryTest` project to the `StringLibrary` project.

1. Run the following command:

.NET CLI

```
dotnet add StringLibraryTest/StringLibraryTest.csproj reference
StringLibrary/StringLibrary.csproj
```

Add and run unit test methods

When Visual Studio invokes a unit test, it runs each method that is marked with the `TestMethodAttribute` attribute in a class that is marked with the `TestClassAttribute` attribute. A test method ends when the first failure is found 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 the `Assert` class's most frequently called methods are shown in the following table:

Assert methods	Function
<code>Assert.AreEqual</code>	Verifies that two values or objects are equal. The assert fails if the values or objects aren't equal.
<code>Assert.AreSame</code>	Verifies that two object variables refer to the same object. The assert fails if the variables refer to different objects.
<code>Assert.IsFalse</code>	Verifies that a condition is <code>false</code> . The assert fails if the condition is <code>true</code> .
<code>Assert.IsNotNull</code>	Verifies that an object isn't <code>null</code> . The assert fails if the object is <code>null</code> .

You can also use the `Assert.ThrowsException` method in a test method to indicate the type of exception it's expected to throw. The test fails if the specified exception isn't 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 `Assert.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 `Assert.IsFalse` method.

Since your library method handles strings, you also want to make sure that it successfully handles an `empty string (String.Empty)` and a `null` string. An empty string is one that has no characters and whose `Length` is 0. A `null` string is one that hasn't been initialized. You can call `StartsWithUpper` directly as a static method and pass a single `String` argument. Or you can call `StartsWithUpper` as an extension method on a `string` variable assigned to `null`.

You'll define three methods, each of which calls an `Assert` method for each element in a string array. You'll call a method overload that lets you specify an error message to be displayed in case of test failure. The message identifies the string that caused the failure.

To create the test methods:

1. Open `StringLibraryTest/UnitTest1.cs` and replace all of the code with the following code.

C#

```
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));
        }
    }
}
```

The test of uppercase characters in the `TestStartsWithUpper` method includes the Greek capital letter alpha (U+0391) and the Cyrillic capital letter EM (U+041C). 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. Save your changes.

3. Run the tests:

.NET CLI

```
dotnet test StringLibraryTest/StringLibraryTest.csproj
```

The terminal output shows that all tests passed.

Output

```
Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed! - Failed: 0, Passed: 3, Skipped: 0, Total: 3,
Duration: 3 ms - StringLibraryTest.dll (net8.0)
```

Handle test failures

If you're doing test-driven development (TDD), you write tests first and they fail the first time you run them. Then you add code to the app that makes the test succeed. For this tutorial, you created the test after writing the app code that it validates, so you haven't seen the test fail. To validate that a test fails when you expect it to fail, add an invalid value to the test input.

1. Modify the `words` array in the `TestDoesNotStartWithUpper` method to include the string "Error".

C#

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

2. Run the tests:

.NET CLI

```
dotnet test StringLibraryTest/StringLibraryTest.csproj
```

The terminal output shows that one test fails, and it provides an error message for the failed test: "Assert.IsFalse failed. Expected for 'Error': false; actual: True". Because of the failure, no strings in the array after "Error" were tested.

Output

```
Starting test execution, please wait...
A total of 1 test files matched the specified pattern.
  Failed TestDoesNotStartWithUpper [28 ms]
    Error Message:
      Assert.IsFalse failed. Expected for 'Error': false; Actual: True
    Stack Trace:
      at StringLibraryTest.UnitTest1.TestDoesNotStartWithUpper() in
C:\ClassLibraryProjects\StringLibraryTest\UnitTest1.cs:line 33

Failed!  - Failed:      1, Passed:      2, Skipped:      0, Total:      3,
Duration: 31 ms - StringLibraryTest.dll (net5.0)
```

3. Remove the string "Error" that you added in step 1. Rerun the test and the tests pass.

Test the Release version of the library

Now that the tests have all passed when running the Debug build of the library, 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.

1. Run the tests with the Release build configuration:

.NET CLI

```
dotnet test StringLibraryTest/StringLibraryTest.csproj --configuration
Release
```

The tests pass.

Debug tests

If you're using Visual Studio Code as your IDE, you can use the same process shown in [Debug a .NET console application using Visual Studio Code](#) to debug code using your

unit test project. Instead of starting the *ShowCase* app project, open *StringLibraryTest/UnitTest1.cs*, and select **Debug All Tests** between lines 7 and 8. If you're unable to find it, press **Ctrl + Shift + P** to open the command palette and enter **Reload Window**.

Visual Studio Code starts the test project with the debugger attached. Execution will stop at any breakpoint you've added to the test project or the underlying library code.

Additional resources

- [Unit testing in .NET](#)

Next steps

In this tutorial, you unit tested a class library. You can make the library available to others by publishing it to [NuGet](#) as a package. To learn how, follow a NuGet tutorial:

[Create and publish a package using the dotnet CLI](#)

If you publish a library as a NuGet package, others can install and use it. To learn how, follow a NuGet tutorial:

[Install and use a package using the dotnet CLI](#)

A library doesn't have to be distributed as a package. It can be bundled with a console app that uses it. To learn how to publish a console app, see the earlier tutorial in this series:

[Publish a .NET console application using Visual Studio Code](#)

The Visual Studio Code extension C# Dev Kit provides more tools for developing C# apps and libraries:

[C# Dev Kit](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

issues and pull requests. For more information, see [our contributor guide](#).

 [Open a documentation issue](#)

 [Provide product feedback](#)

Quickstart: Install and use a package with the dotnet CLI

Article • 08/21/2023

NuGet packages contain compiled binary code that developers make available for other developers to use in their projects. For more information, see [What is NuGet](#). This quickstart describes how to install the popular [Newtonsoft.Json](#) NuGet package into a .NET project by using the [dotnet add package](#) command.

You refer to installed packages in code with a `using <namespace>` directive, where `<namespace>` is often the package name. You can then use the package's API in your project.

Tip

Browse [nuget.org/packages](#) to find packages you can reuse in your own applications. You can search directly at <https://nuget.org>, or find and install packages from within Visual Studio. For more information, see [Find and evaluate NuGet packages for your project](#).

Prerequisites

- The [.NET SDK](#), which provides the `dotnet` command-line tool. Starting in Visual Studio 2017, the dotnet CLI automatically installs with any .NET or .NET Core related workloads.

Create a project

You can install NuGet packages into a .NET project. For this walkthrough, create a simple .NET console project by using the dotnet CLI, as follows:

1. Create a folder named *Nuget.Quickstart* for the project.
2. Open a command prompt and switch to the new folder.
3. Create the project by using the following command:

.NET CLI

```
dotnet new console
```

4. Use `dotnet run` to test the app. You should see the output `Hello, World!`.

Add the Newtonsoft.Json NuGet package

1. Use the following command to install the `Newtonsoft.json` package:

.NET CLI

```
dotnet add package Newtonsoft.Json
```

2. After the command completes, open the `Nuget.Quickstart.csproj` file in Visual Studio to see the added NuGet package reference:

XML

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

Use the Newtonsoft.Json API in the app

1. In Visual Studio, open the `Program.cs` file and add the following line at the top of the file:

CS

```
using Newtonsoft.Json;
```

2. Add the following code to replace the `Console.WriteLine("Hello, World!");` statement:

CS

```
namespace Nuget.Quickstart
{
    public class Account
    {
        public string? Name { get; set; }
        public string? Email { get; set; }
        public DateTime DOB { get; set; }
    }
}
```

```
internal class Program
{
    static void Main(string[] args)
    {
        Account account = new Account
        {
            Name = "John Doe",
            Email = "john@nuget.org",
            DOB = new DateTime(1980, 2, 20, 0, 0, 0,
DateTimeKind.Utc),
        };

        string json = JsonConvert.SerializeObject(account,
Formatting.Indented);
        Console.WriteLine(json);
    }
}
```

3. Save the file, then build and run the app by using the `dotnet run` command. The output is the JSON representation of the `Account` object in the code:

Output

```
{
  "Name": "John Doe",
  "Email": "john@nuget.org",
  "DOB": "1980-02-20T00:00:00Z"
}
```

Congratulations on installing and using your first NuGet package!

Related video

<https://learn.microsoft.com/shows/NuGet-101/Install-and-Use-a-NuGet-Package-with-the-NET-CLI-3-of-5/player>

Find more NuGet videos on [Channel 9](#) and [YouTube](#).

Next steps

Learn more about installing and using NuGet packages with the `dotnet` CLI:

[Install and use packages by using the `dotnet` CLI](#)

- Overview and workflow of package consumption

- Find and choose packages
- Package references in project files

Quickstart: Create and publish a package with the dotnet CLI

Article • 08/21/2023

This quickstart shows you how to quickly create a NuGet package from a .NET class library and publish it to nuget.org by using the .NET command-line interface, or [dotnet CLI](#).

Prerequisites

- The [.NET SDK](#), which provides the dotnet command-line tool. Starting in Visual Studio 2017, the dotnet CLI automatically installs with any .NET or .NET Core related workloads.
- A free account on nuget.org. Follow the instructions at [Add a new individual account](#).

Create a class library project

You can use an existing .NET Class Library project for the code you want to package, or create a simple project as follows:

1. Create a folder named *AppLogger*.
2. Open a command prompt and switch to the *AppLogger* folder. All the dotnet CLI commands in this quickstart run on the current folder by default.
3. Enter `dotnet new classlib`, which creates a project with the current folder name.

For more information, see [dotnet new](#).

Add package metadata to the project file

Every NuGet package has a manifest that describes the package's contents and dependencies. In the final package, the manifest is a *.nuspec* file, which uses the NuGet metadata properties you include in the project file.

Open the *.csproj*, *.fproj*, or *.vbproj* project file, and add the following properties inside the existing `<PropertyGroup>` tag. Use your own values for name and company, and replace the package identifier with a unique value.

XML

```
<PackageId>Contoso.08.28.22.001.Test</PackageId>
<Version>1.0.0</Version>
<Authors>your_name</Authors>
<Company>your_company</Company>
```

ⓘ Important

The package identifier must be unique across nuget.org and other package sources. Publishing makes the package publicly visible, so if you use the example AppLogger library or other test library, use a unique name that includes `Sample` or `Test`.

You can add any optional properties described in [NuGet metadata properties](#).

ⓘ Note

For packages you build for public consumption, pay special attention to the `PackageTags` property. Tags help others find your package and understand what it does.

Run the pack command

To build a NuGet package or `.nupkg` file from the project, run the [dotnet pack](#) command, which also builds the project automatically.

.NET CLI

```
dotnet pack
```

The output shows the path to the `.nupkg` file:

Output

```
MSBuild version 17.3.0+92e077650 for .NET
Determining projects to restore...
Restored C:\Users\myname\source\repos\AppLogger\AppLogger.csproj (in 64
ms).
AppLogger ->
C:\Users\myname\source\repos\AppLogger\bin\Debug\net6.0\AppLogger.dll
Successfully created package
```

```
'C:\Users\myname\source\repos\AppLogger\bin\Debug\Contoso.08.28.22.001.Test.  
1.0.0.nupkg'.
```

Automatically generate package on build

To automatically run `dotnet pack` whenever you run `dotnet build`, add the following line to your project file within `<PropertyGroup>`:

XML

```
<GeneratePackageOnBuild>true</GeneratePackageOnBuild>
```

Publish the package

Publish your `.nupkg` file to nuget.org by using the `dotnet nuget push` command with an API key you get from nuget.org.

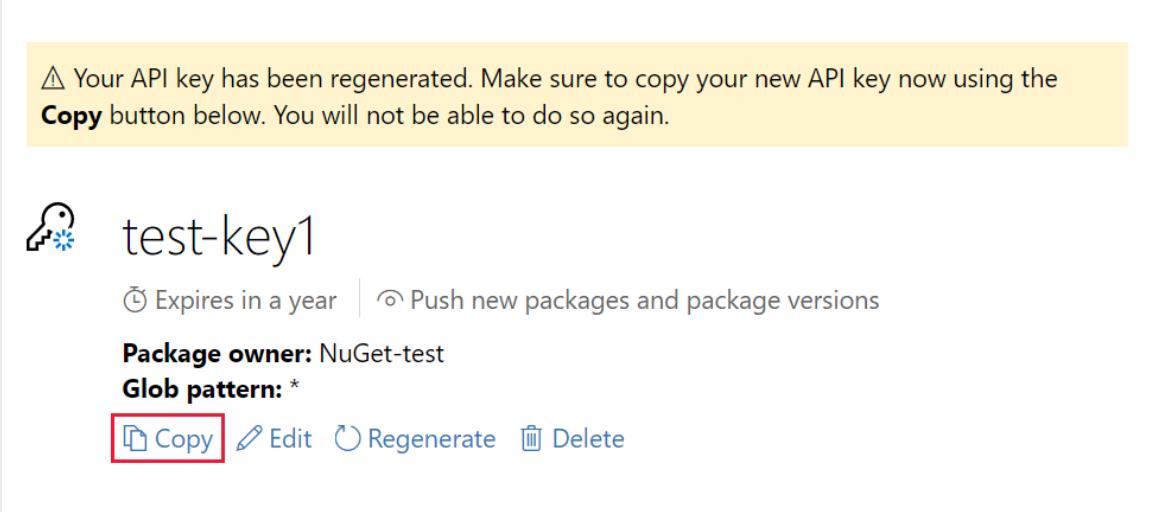
ⓘ Note

- Nuget.org scans all uploaded packages for viruses and rejects the packages if it finds any viruses. Nuget.org also scans all existing listed packages periodically.
- Packages you publish to nuget.org are publicly visible to other developers unless you unlist them. To host packages privately, see [Host your own NuGet feeds](#).

Get your API key

1. [Sign into your nuget.org account](#) or [create an account](#) if you don't have one already.
2. Select your user name at upper right, and then select **API Keys**.
3. Select **Create**, and provide a name for your key.
4. Under **Select Scopes**, select **Push**.
5. Under **Select Packages > Glob Pattern**, enter `*`.
6. Select **Create**.

7. Select **Copy** to copy the new key.



⚠ Your API key has been regenerated. Make sure to copy your new API key now using the **Copy** button below. You will not be able to do so again.

 **test-key1**

⌚ Expires in a year | ⚑ Push new packages and package versions

Package owner: NuGet-test
Glob pattern: *

ⓘ Important

- Always keep your API key a secret. The API key is like a password that allows anyone to manage packages on your behalf. Delete or regenerate your API key if it's accidentally revealed.
- Save your key in a secure location, because you can't copy the key again later. If you return to the API key page, you need to regenerate the key to copy it. You can also remove the API key if you no longer want to push packages.

Scoping lets you create separate API keys for different purposes. Each key has an expiration timeframe, and you can scope the key to specific packages or glob patterns. You also scope each key to specific operations: Push new packages and package versions, push only new package versions, or unlist.

Through scoping, you can create API keys for different people who manage packages for your organization so they have only the permissions they need.

For more information, see [scoped API keys](#).

Publish with `dotnet nuget push`

From the folder that contains the `.nupkg` file, run the following command. Specify your `.nupkg` filename, and replace the key value with your API key.

.NET CLI

```
dotnet nuget push Contoso.08.28.22.001.Test.1.0.0.nupkg --api-key  
qz2jga8p13dvn2akksyquwcs9ygggg4exypy3bhxy6w6x6 --source  
https://api.nuget.org/v3/index.json
```

The output shows the results of the publishing process:

Output

```
Pushing Contoso.08.28.22.001.Test.1.0.0.nupkg to  
'https://www.nuget.org/api/v2/package'...  
PUT https://www.nuget.org/api/v2/package/  
warn : All published packages should have license information specified.  
Learn more: https://aka.ms/nuget/authoring-best-practices#licensing.  
Created https://www.nuget.org/api/v2/package/ 1221ms  
Your package was pushed.
```

For more information, see [dotnet nuget push](#).

ⓘ Note

If you want to avoid your test package being live on nuget.org, you can push to the nuget.org test site at <https://int.nugettest.org>. Note that packages uploaded to int.nugettest.org might not be preserved.

Publish errors

Errors from the `push` command typically indicate the problem. For example, you might have forgotten to update the version number in your project, so you're trying to publish a package that already exists.

You also see errors if your API key is invalid or expired, or if you try to publish a package using an identifier that already exists on the host. Suppose, for example, the identifier `AppLogger-test` already exists on nuget.org. If you try to publish a package with that identifier, the `push` command gives the following error:

Output

```
Response status code does not indicate success: 403 (The specified API key  
is invalid,  
has expired, or does not have permission to access the specified package.).
```

If you get this error, check that you're using a valid API key that hasn't expired. If you are, the error indicates the package identifier already exists on the host. To fix the error,

change the package identifier to be unique, rebuild the project, recreate the `.nupkg` file, and retry the `push` command.

Manage the published package

When your package successfully publishes, you receive a confirmation email. To see the package you just published, on nuget.org, select your user name at upper right, and then select **Manage Packages**.

Note

It might take awhile for your package to be indexed and appear in search results where others can find it. During that time, your package appears under **Unlisted Packages**, and the package page shows the following message:

 **This package has not been published yet.** It will appear in search results and will be available for install/restore after both validation and indexing are complete. Package validation and indexing may take up to an hour. [Read more](#).

You've now published a NuGet package to nuget.org that other developers can use in their projects.

If you've created a package that isn't useful (such as this sample package that was created with an empty class library), or you decide you don't want the package to be visible, you can *unlist* the package to hide it from search results:

1. After the package appears under **Published Packages** on the **Manage Packages** page, select the pencil icon next to the package listing.

Published Packages					1 package / 0 downloads
Package ID	Owners	Signing Owner	Downloads	Latest Version	
 Contoso.08.28.22.001.Test	Test	username (0 certificates)	0	1.0.0	

2. On the next page, select **Listing**, deselect the **List in search results** checkbox, and then select **Save**.

✓ Listing

Select version

1.0.0 (Latest)

List or unlist version

ⓘ You can control how your packages are listed using the checkbox below. As per [policy](#), permanent deletion is not supported as it would break every project depending on the availability of the package. For more assistance, [Contact Support](#).

List in search results.

Unlisted packages cannot be installed directly and do not show up in search results.

Save

The package now appears under **Unlisted Packages** in **Manage Packages** and no longer appears in search results.

ⓘ Note

To avoid your test package being live on nuget.org, you can push to the nuget.org test site at <https://int.nugettest.org>. Note that packages uploaded to int.nugettest.org might not be preserved.

Congratulations on creating and publishing your first NuGet package!

Related video

<https://learn.microsoft.com/shows/NuGet-101/Create-and-Publish-a-NuGet-Package-with-the-NET-CLI-5-of-5/player>

Find more NuGet videos on [Channel 9](#) and [YouTube](#).

Next steps

See more details about how to create packages with the dotnet CLI:

Create a NuGet package with the dotnet CLI

Get more information about creating and publishing NuGet packages:

- [Publish a package](#)
- [Prerelease packages](#)
- [Support multiple target frameworks](#)
- [Package versioning](#)
- [Add a license expression or file](#)
- [Create localized packages](#)
- [Create symbol packages](#)
- [Sign packages](#)

Tutorial: Create a .NET console application using Visual Studio for Mac

Article • 09/08/2023

This tutorial shows how to create and run a .NET console application using Visual Studio for Mac.

Important

Microsoft has announced the retirement of Visual Studio for Mac. Visual Studio for Mac will no longer be supported starting August 31, 2024. Alternatives include:

- Visual Studio Code with the [C# Dev Kit](#) and related extensions, such as [.NET MAUI](#) and [Unity](#).
- Visual Studio running on Windows in a VM on Mac.
- Visual Studio running on Windows in a [VM in the Cloud](#).

For more information, see [Visual Studio for Mac retirement announcement](#).

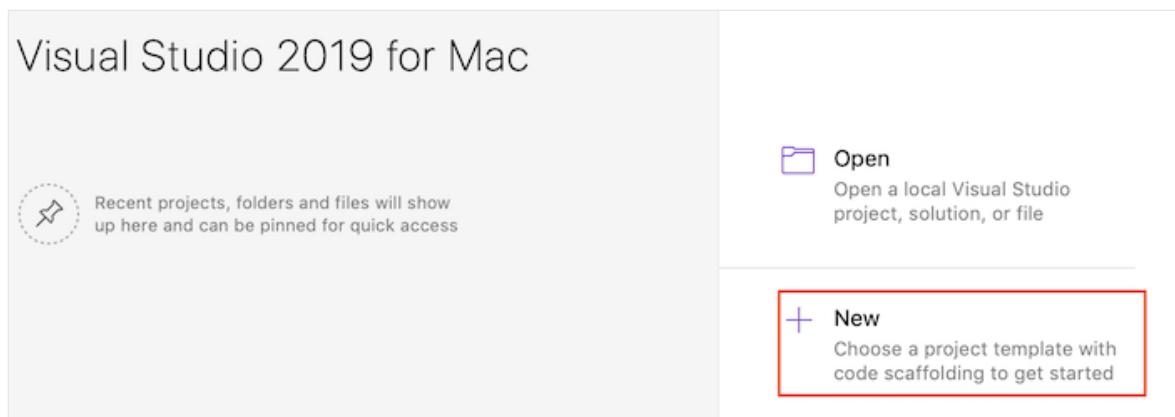
Prerequisites

- [Visual Studio for Mac version 8.8 or later](#). Select the option to install .NET Core. Installing Xamarin is optional for .NET development. For more information, see the following resources:
 - [Tutorial: Install Visual Studio for Mac](#).
 - [Supported macOS versions](#).
 - [.NET versions supported by Visual Studio for Mac](#).

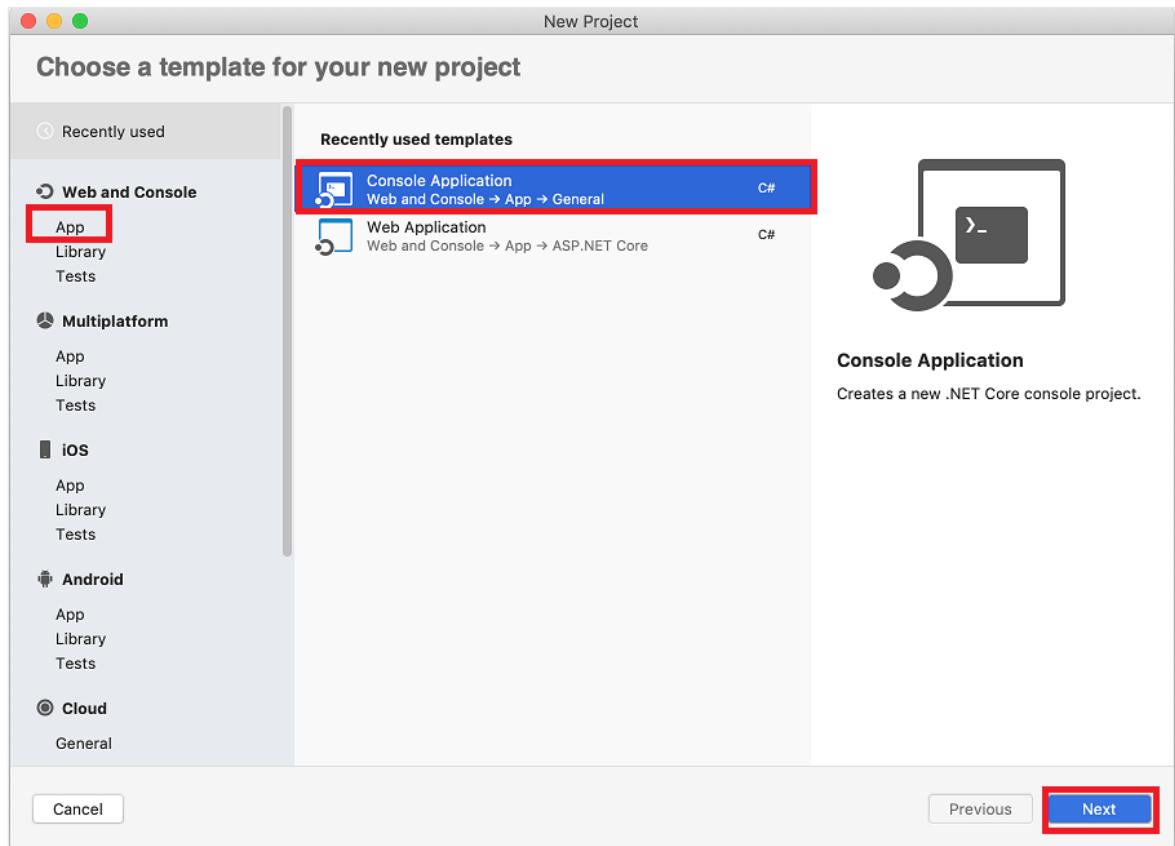
Create the app

1. Start Visual Studio for Mac.
2. Select **New** in the start window.

Visual Studio 2019 for Mac

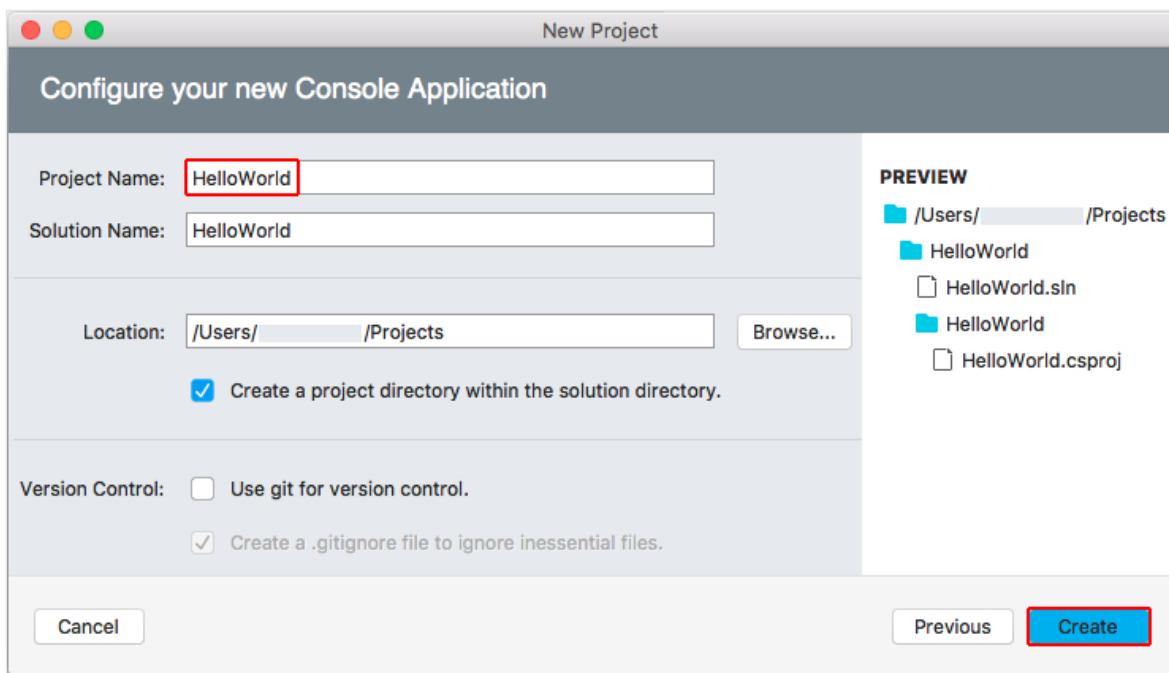


3. In the **New Project** dialog, select **App** under the **Web and Console** node. Select the **Console Application** template, and select **Next**.



4. In the **Target Framework** drop-down of the **Configure your new Console Application** dialog, select **.NET 5.0**, and select **Next**.

5. Type "HelloWorld" for the **Project Name**, and select **Create**.



The template creates a simple "Hello World" application. It calls the [Console.WriteLine\(String\)](#) method to display "Hello World!" in the terminal window.

The template code defines a class, `Program`, with a single method, `Main`, that takes a `String` array as an argument:

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

`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.

Run the app

1. Press  to run the app without debugging.



The screenshot shows the Visual Studio for Mac interface. The top menu bar includes File, Edit, View, Search, Project, Build, and Run. The toolbar below has buttons for Back, Forward, Stop, and Run (Debug > Default). A status bar at the bottom indicates 'No selection'.

The main window displays the code for `Program.cs`:

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

Below the code editor is a terminal window titled "Terminal - HelloWorld". It contains the output "Hello World!" which is highlighted with a red box.

2. Close the **Terminal** window.

Enhance the app

Enhance the application to prompt the user for their name and display it along with the date and time.

1. In *Program.cs*, replace the contents of the `Main` method, which is the line that calls `Console.WriteLine`, with the following code:

C#

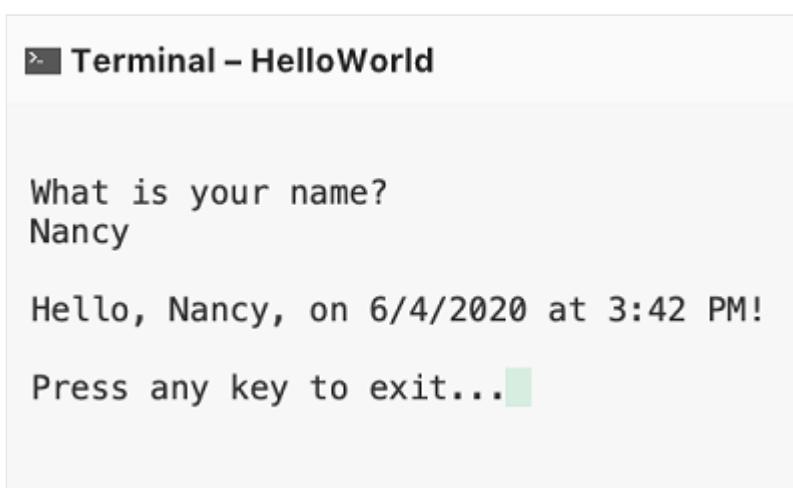
```
Console.WriteLine("What is your name?");
var name = Console.ReadLine();
var currentDate = DateTime.Now;
Console.WriteLine($"{Environment.NewLine}Hello, {name}, on
{currentDate:d} at {currentDate:t}!");
Console.WriteLine($"{Environment.NewLine}Press any key to exit...");
Console.ReadKey(true);
```

This code displays a prompt in the console window and waits until the user enters a string followed by the `enter` key. It stores this string in 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`. And it displays these values in the console window. Finally, it displays a prompt in the console window and calls the `Console.ReadKey(Boolean)` method to wait for user input.

`NewLine` is a platform-independent and language-independent way to represent a line break. Alternatives are `\n` in C# and `vbcrlf` in Visual Basic.

The dollar sign (\$) in front of a string lets you put expressions such as variable names in curly braces in the string. The expression value is inserted into the string in place of the expression. This syntax is referred to as [interpolated strings](#).

2. Press `⌘ ⌘ ⌘` (`option` + `command` + `enter`) to run the app.
3. Respond to the prompt by entering a name and pressing `enter`.



```
Terminal - HelloWorld

What is your name?
Nancy

Hello, Nancy, on 6/4/2020 at 3:42 PM!

Press any key to exit...
```

4. Close the terminal.

Next steps

In this tutorial, you created a .NET console application. In the next tutorial, you debug the app.

[Debug a .NET console application using Visual Studio for Mac](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Debug a .NET console application using Visual Studio for Mac

Article • 09/08/2023

ⓘ Important

Microsoft has announced the retirement of Visual Studio for Mac. Visual Studio for Mac will no longer be supported starting August 31, 2024. Alternatives include:

- Visual Studio Code with the [C# Dev Kit](#) and related extensions, such as [.NET MAUI](#) and [Unity](#).
- Visual Studio running on Windows in a VM on Mac.
- Visual Studio running on Windows in a [VM in the Cloud](#).

For more information, see [Visual Studio for Mac retirement announcement](#).

This tutorial introduces the debugging tools available in Visual Studio for Mac.

Prerequisites

- This tutorial works with the console app that you create in [Create a .NET console application using Visual Studio for Mac](#).

Use Debug build configuration

Debug and *Release* are Visual Studio's built-in build configurations. You use the *Debug* build configuration for debugging and the *Release* configuration for the final release distribution.

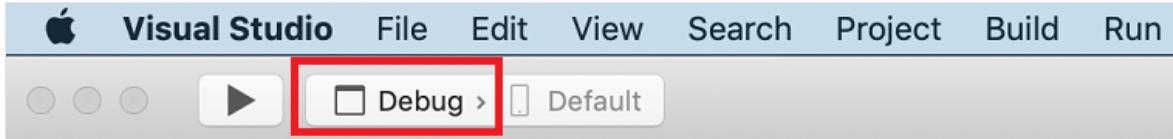
In the *Debug* configuration, a program compiles with full symbolic debug information and no optimization. Optimization complicates debugging, because the relationship between source code and generated instructions is more complex. The *Release* configuration of a program has no symbolic debug information and is fully optimized.

By default, Visual Studio for Mac uses the *Debug* build configuration, so you don't need to change it before debugging.

1. Start Visual Studio for Mac.

2. Open the project that you created in [Create a .NET console application using Visual Studio for Mac](#).

The current build configuration is shown on the toolbar. The following toolbar image shows that Visual Studio is configured to compile the Debug version of the app:

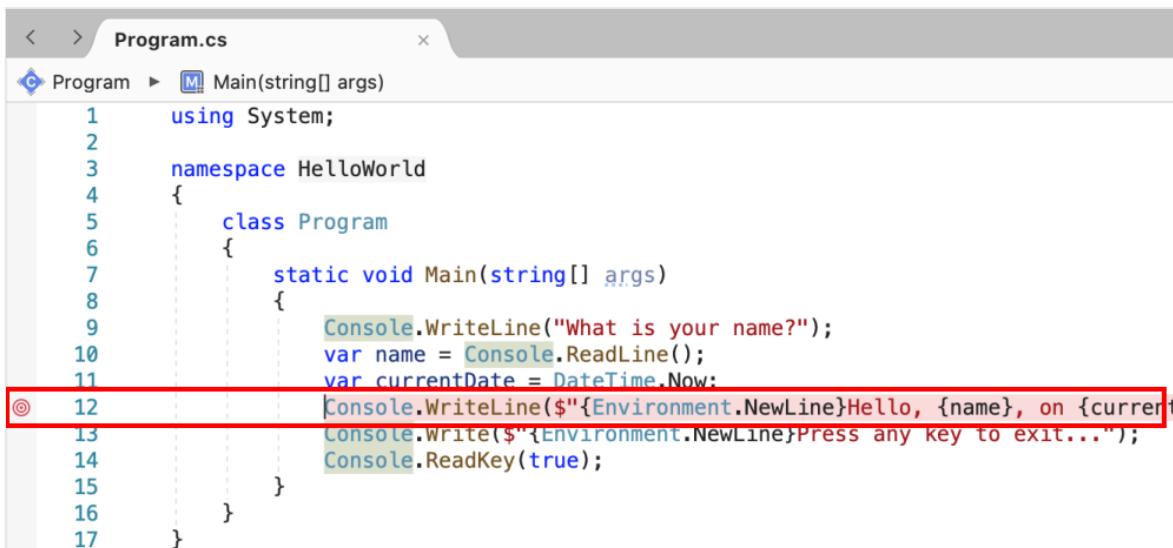


Set a breakpoint

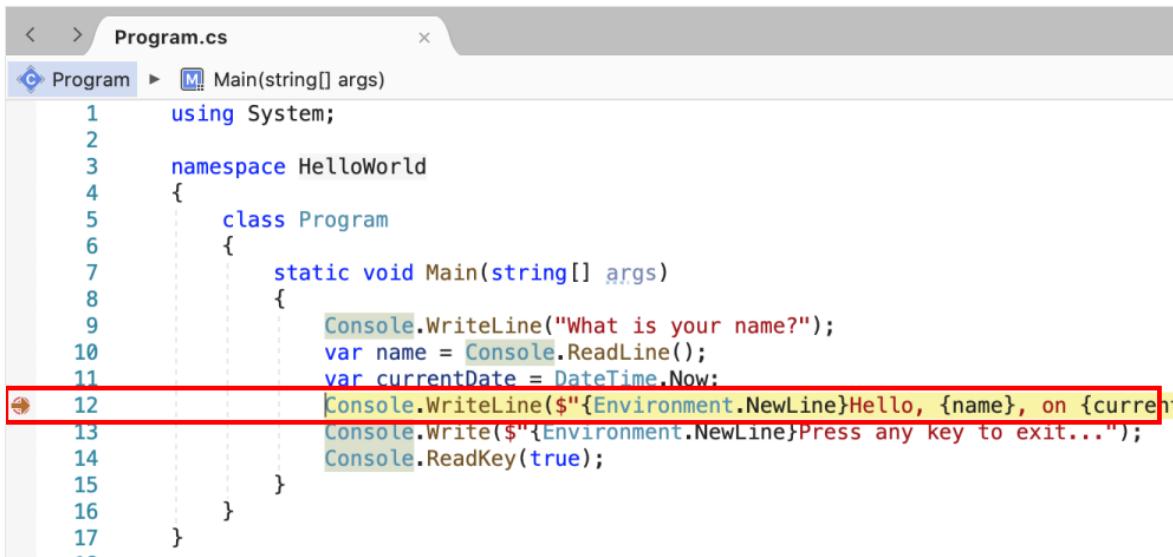
A *breakpoint* temporarily interrupts the execution of the application before the line with the breakpoint is executed.

1. Set a breakpoint on the line that displays the name, date, and time. To do that, place the cursor in the line of code and press   (`command` + `/`). Another way to set a breakpoint is by selecting **Debug > Toggle Breakpoint** from the menu.

Visual Studio indicates the line on which the breakpoint is set by highlighting it and displaying a red dot in the left margin.



2. Press    (`command + enter`) to start the program in debugging mode. Another way to start debugging is by choosing **Debug > Start Debugging** from the menu.
 3. Enter a string in the terminal window when the program prompts for a name, and then press .
 4. Program execution stops when it reaches the breakpoint, before the `Console.WriteLine` method executes.



```
1  using System;
2
3  namespace HelloWorld
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("What is your name?");
10             var name = Console.ReadLine();
11             var currentDate = DateTime.Now;
12             Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate}");
13             Console.WriteLine($"{Environment.NewLine}Press any key to exit...");
14             Console.ReadKey(true);
15         }
16     }
17 }
```

Use the Immediate window

The **Immediate** window lets you interact with the application you're debugging. You can interactively change the value of variables to see how it affects your program.

1. If the **Immediate** window is not visible, display it by choosing **View > Debug Windows > Immediate**.
2. Enter `name = "Gracie"` in the **Immediate** window and press `enter`.
3. Enter `currentDate = currentDate.AddDays(1)` in the **Immediate** window and press `enter`.

The **Immediate** window displays the new value of the string variable and the properties of the `DateTime` value.

```
[-] Immediate
name = "Gracie"
"Gracie"
currentDate = currentDate.AddDays(1)
{5/27/2021 11:15:29 AM}
    Date: {5/27/2021 12:00:00 AM}
    Day: 27
    DayOfWeek: System.DayOfWeek.Thursday
    DayOfYear: 147
    Hour: 11
    Kind: System.DateTimeKind.Local
    Millisecond: 799
    Minute: 15
    Month: 5
    Second: 29
    Ticks: 637577109297996840
    TimeOfDay: {11:15:29.7996840}
    Year: 2021
    Static members:
    Non-Public members:
```

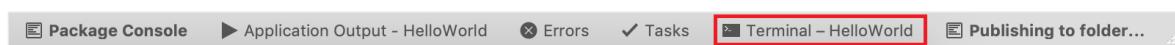
The **Locals** window displays the values of variables that are defined in the currently executing method. The values of the variables that you just changed are updated in the **Locals** window.

Breakpoints	Locals	Watch	Threads
Name	Value	Type	
	args	{string[0]}	string[]
	name	Gracie	string
▶	currentDate	{5/27/2021 11:15:29 AM}	System.DateTime

4. Press **⌘ ↵** (**command** + **enter**) to continue debugging.

The values displayed in the terminal correspond to the changes you made in the **Immediate** window.

If you don't see the Terminal, select **Terminal - HelloWorld** in the bottom navigation bar.



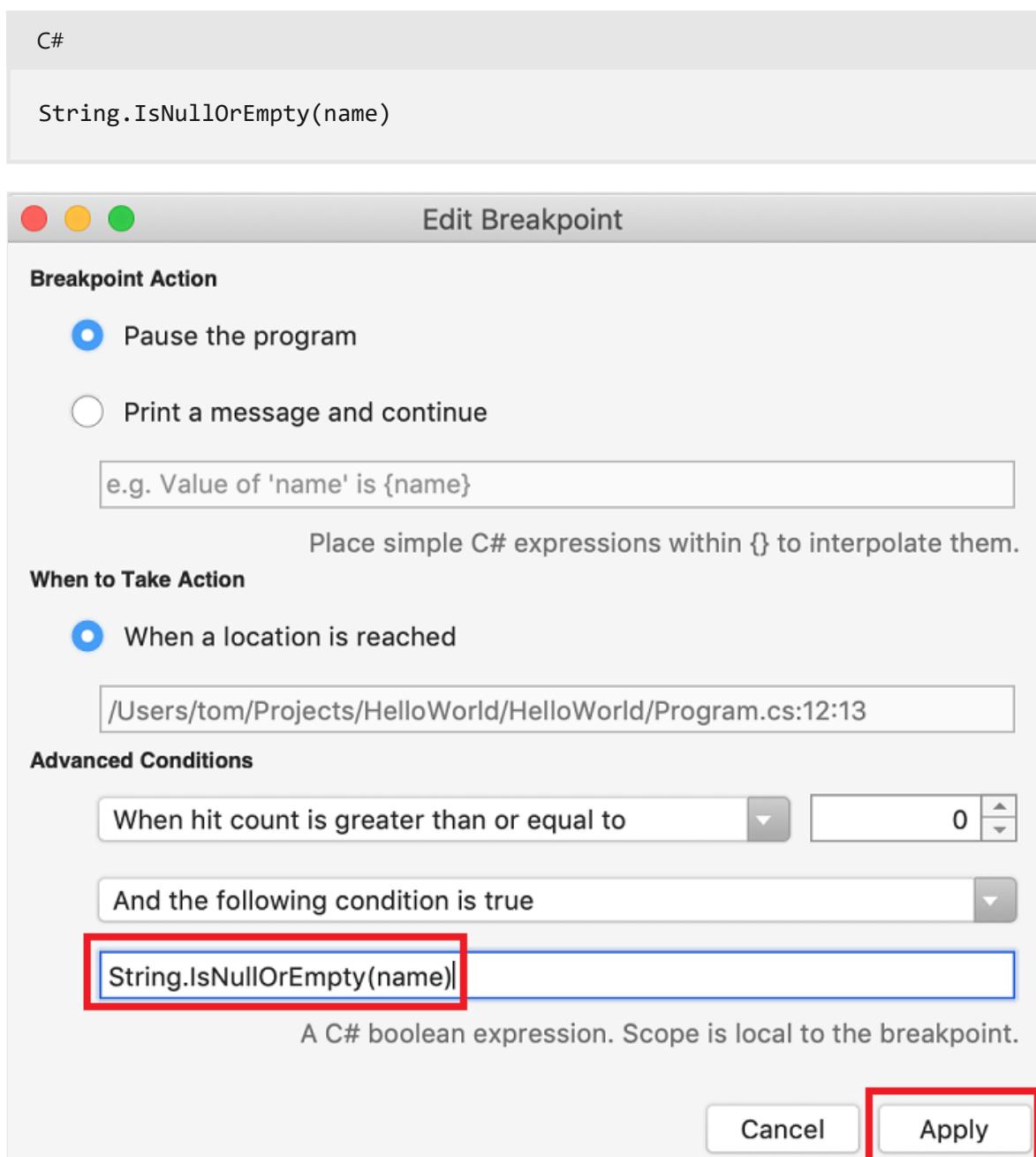
5. Press any key to exit the program.

6. Close the terminal window.

Set a conditional breakpoint

The program displays a string that the user enters. What happens if the user doesn't enter anything? You can test this with a useful debugging feature called a *conditional breakpoint*.

1. `ctrl`-click on the red dot that represents the breakpoint. In the context menu, select **Edit Breakpoint**.
2. In the **Edit Breakpoint** dialog, enter the following code in the field that follows **And the following condition is true**, and select **Apply**.



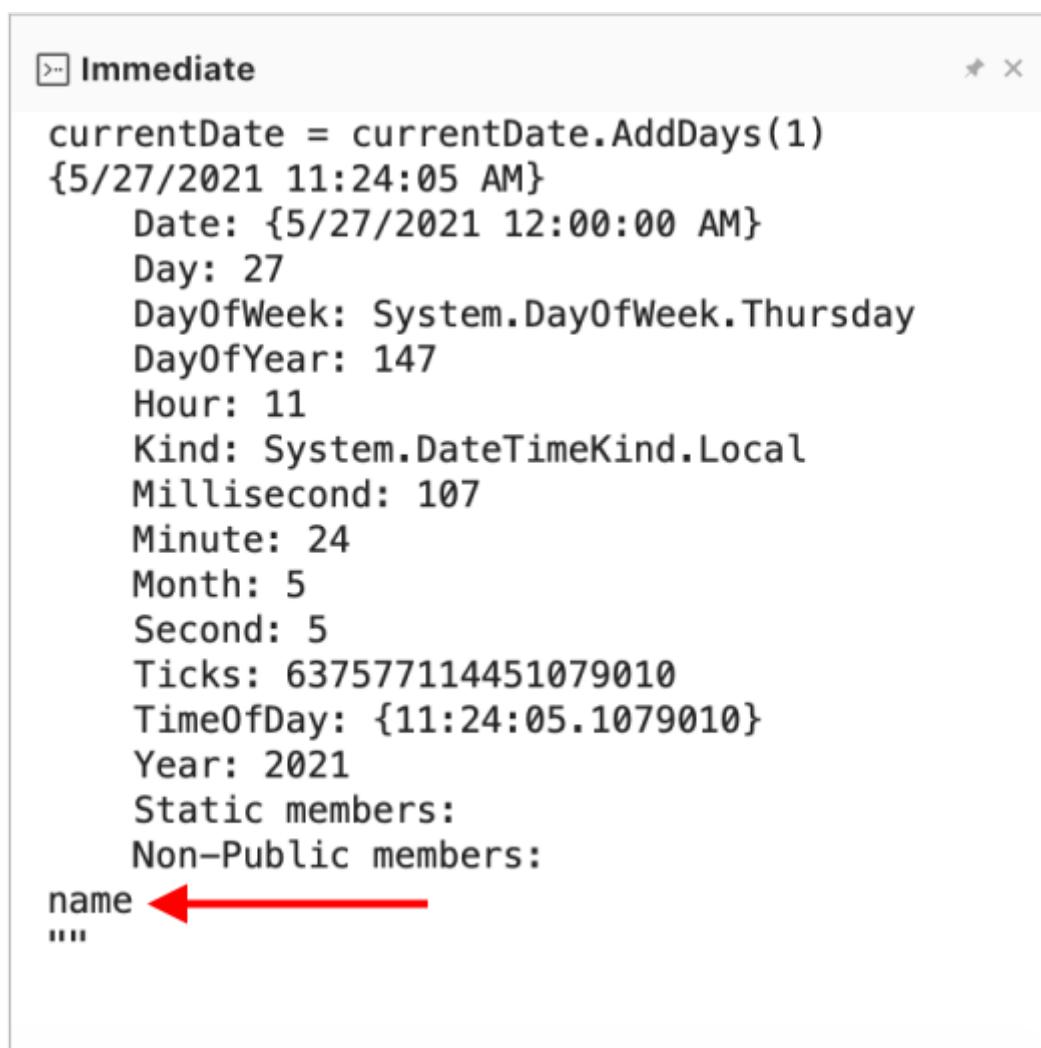
Each time the breakpoint is hit, the debugger calls the `String.IsNullOrEmpty(name)` method, and it breaks on this line only if the method call returns `true`.

Instead of a conditional expression, you can specify a *hit count*, which interrupts program execution before a statement is executed a specified number of times.

3. Press `⌘ ↵` (`command + enter`) to start debugging.
4. In the terminal window, press `enter` when prompted to enter your name.

Because the condition you specified (`name` is either `null` or `String.Empty`) has been satisfied, program execution stops when it reaches the breakpoint.

5. Select the **Locals** window, which shows the values of variables that are local to the currently executing method. In this case, `Main` is the currently executing method. Observe that the value of the `name` variable is `""`, that is, `String.Empty`.
6. You can also see that the value is an empty string by entering the `name` variable name in the **Immediate** window and pressing `enter`.



The screenshot shows the Immediate window with the following content:

```
currentDate = currentDate.AddDays(1)
{5/27/2021 11:24:05 AM}
  Date: {5/27/2021 12:00:00 AM}
  Day: 27
  DayOfWeek: System.DayOfWeek.Thursday
  DayOfYear: 147
  Hour: 11
  Kind: System.DateTimeKind.Local
  Millisecond: 107
  Minute: 24
  Month: 5
  Second: 5
  Ticks: 637577114451079010
  TimeOfDay: {11:24:05.1079010}
  Year: 2021
  Static members:
  Non-Public members:
name
```

A red arrow points to the word "name" in the list of non-public members.

7. Press `⌘ ↵` (`command + enter`) to continue debugging.

8. In the terminal window, press any key to exit the program.

9. Close the terminal window.

10. Clear the breakpoint by clicking on the red dot in the left margin of the code window. Another way to clear a breakpoint is by choosing **Debug > Toggle Breakpoint** while the line of code is selected.

Step 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 follow program flow through a small part of your program code. Since this program is small, you can step through the entire program.

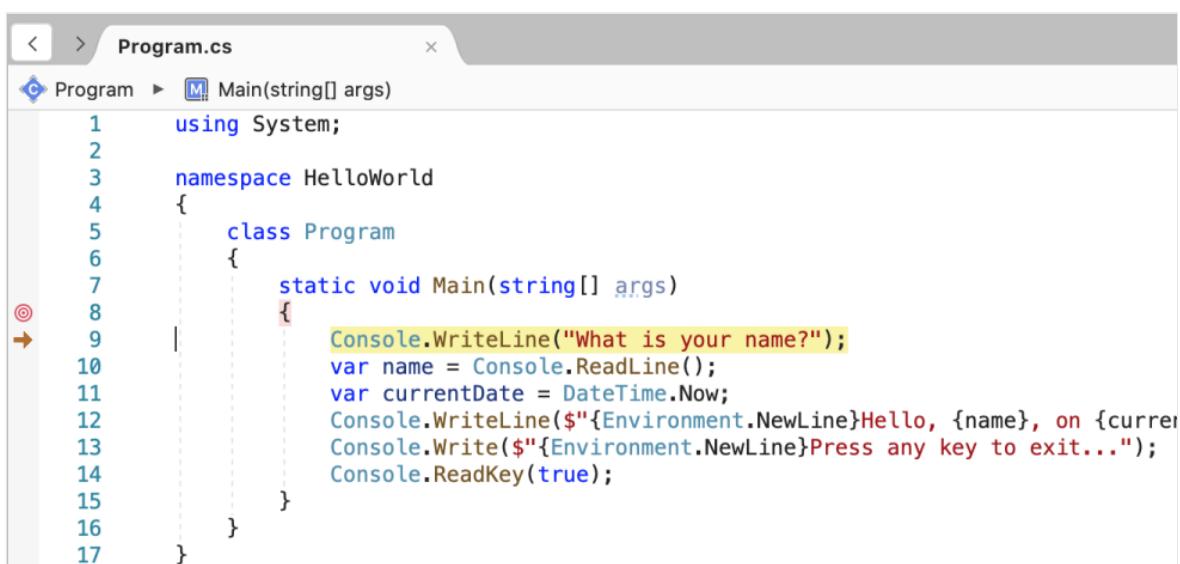
1. Set a breakpoint on the curly brace that marks the start of the `Main` method (press `command + \`).

2. Press `⌘ ↵` (`command + enter`) to start debugging.

Visual Studio stops on the line with the breakpoint.

3. Press `⇧ ⌘ I` (`shift + command + I`) or select **Debug > Step Into** to advance one line.

Visual Studio highlights and displays an arrow beside the next line of execution.



```
1  using System;
2
3  namespace HelloWorld
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine("What is your name?");
10             var name = Console.ReadLine();
11             var currentDate = DateTime.Now;
12             Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate}");
13             Console.WriteLine($"{Environment.NewLine}Press any key to exit...");
14             Console.ReadKey(true);
15         }
16     }
17 }
```

At this point, the **Locals** window shows that the `args` array is empty, and `name` and `currentDate` have default values. In addition, Visual Studio has opened a blank terminal.

4. Press    (`shift`+`command`+`I`).

Visual Studio highlights the statement that includes the `name` variable assignment. The **Locals** window shows that `name` is `null`, and the terminal displays the string "What is your name?".

5. Respond to the prompt by entering a string in the console window and pressing `enter`.

6. Press    (`shift`+`command`+`I`).

Visual Studio highlights the statement that includes the `currentDate` variable assignment. The **Locals** window shows the value returned by the call to the [Console.ReadLine](#) method. The terminal displays the string you entered at the prompt.

7. Press    (`shift`+`command`+`I`).

The **Locals** window shows the value of the `currentDate` variable after the assignment from the [DateTime.Now](#) property. The terminal is unchanged.

8. Press    (`shift`+`command`+`I`).

Visual Studio calls the [Console.WriteLine\(String, Object, Object\)](#) method. The terminal displays the formatted string.

9. Press    (`shift`+`command`+`U`) or select **Run > Step Out**.

The terminal displays a message and waits for you to press a key.

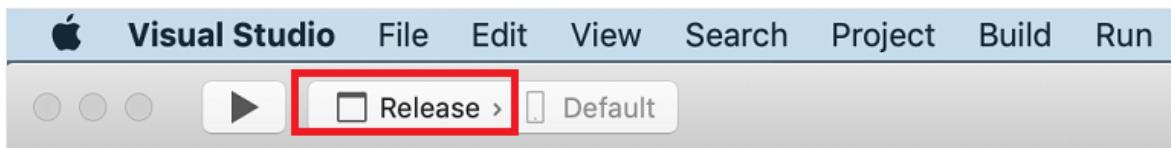
10. Press any key to exit the program.

Use Release build configuration

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

To build and test the Release version of the console application, do the following steps:

1. Change the build configuration on the toolbar from **Debug** to **Release**.



2. Press `⌥ ⌘ ⌘ ⌘` (`option + command + enter`) to run without debugging.

Next steps

In this tutorial, you used Visual Studio debugging tools. In the next tutorial, you publish a deployable version of the app.

[Publish a .NET console application using Visual Studio for Mac](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Publish a .NET console application using Visual Studio for Mac

Article • 09/08/2023

ⓘ Important

Microsoft has announced the retirement of Visual Studio for Mac. Visual Studio for Mac will no longer be supported starting August 31, 2024. Alternatives include:

- Visual Studio Code with the [C# Dev Kit](#) and related extensions, such as [.NET MAUI](#) and [Unity](#).
 - Visual Studio running on Windows in a VM on Mac.
 - Visual Studio running on Windows in a [VM in the Cloud](#).

For more information, see [Visual Studio for Mac retirement announcement](#).

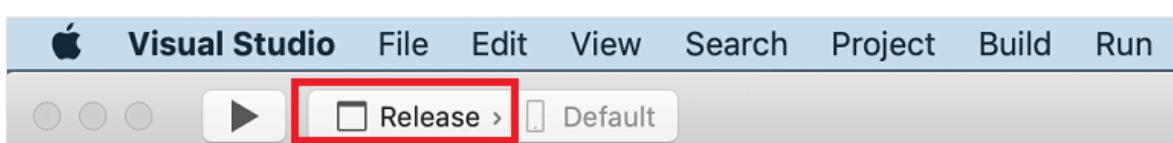
This tutorial shows how to publish a console app so that other users can run it. Publishing creates the set of files that are needed to run your application. To deploy the files, copy them to the target machine.

Prerequisites

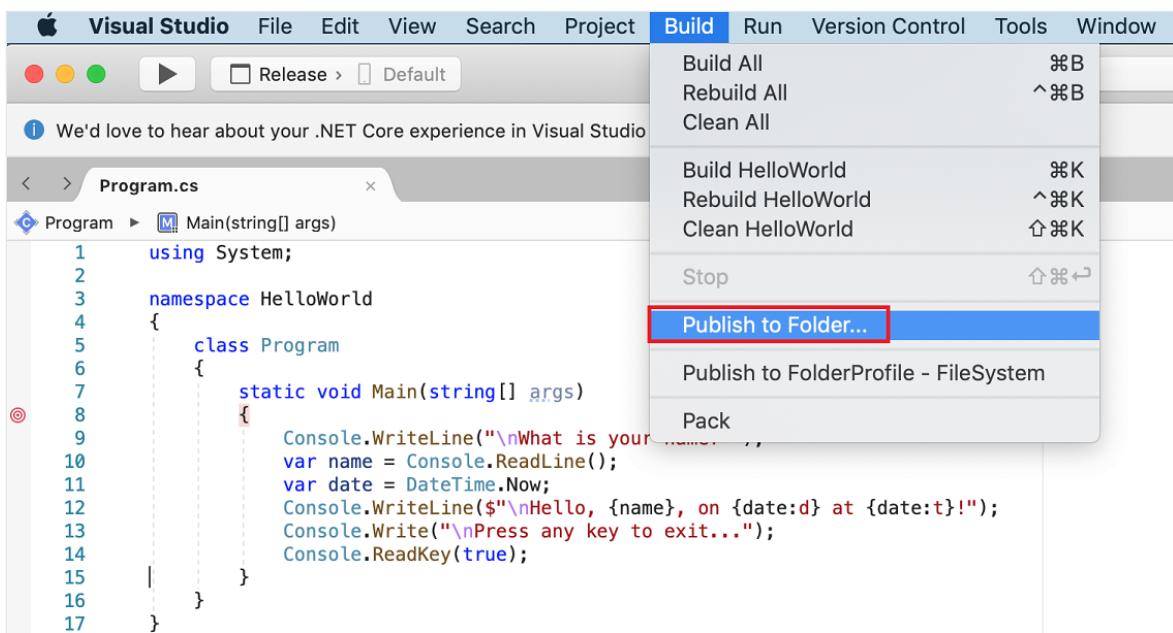
- This tutorial works with the console app that you create in [Create a .NET console application using Visual Studio for Mac](#).

Publish the app

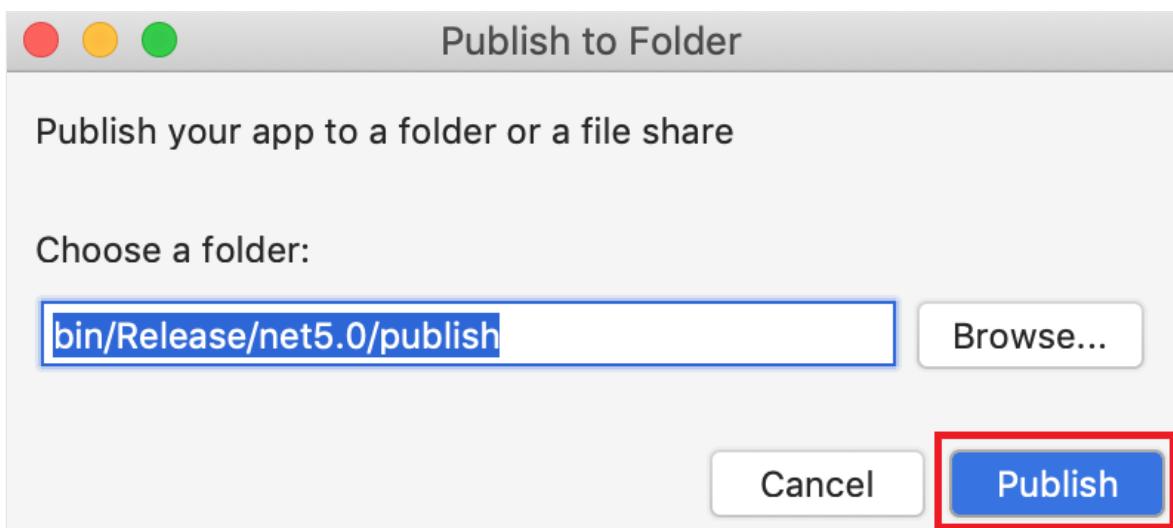
1. Start Visual Studio for Mac.
 2. Open the HelloWorld project that you created in [Create a .NET console application using Visual Studio for Mac](#).
 3. Make sure that Visual Studio is building the Release version of your application. If necessary, change the build configuration setting on the toolbar from **Debug** to **Release**.



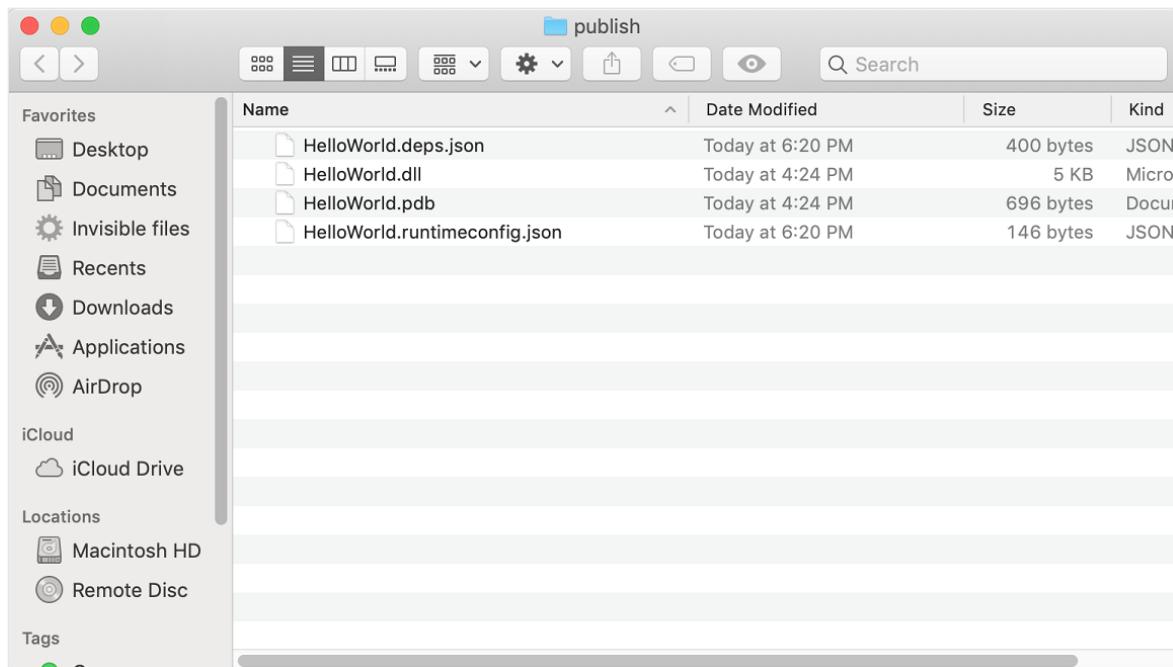
4. From the main menu, choose **Build > Publish to Folder...**



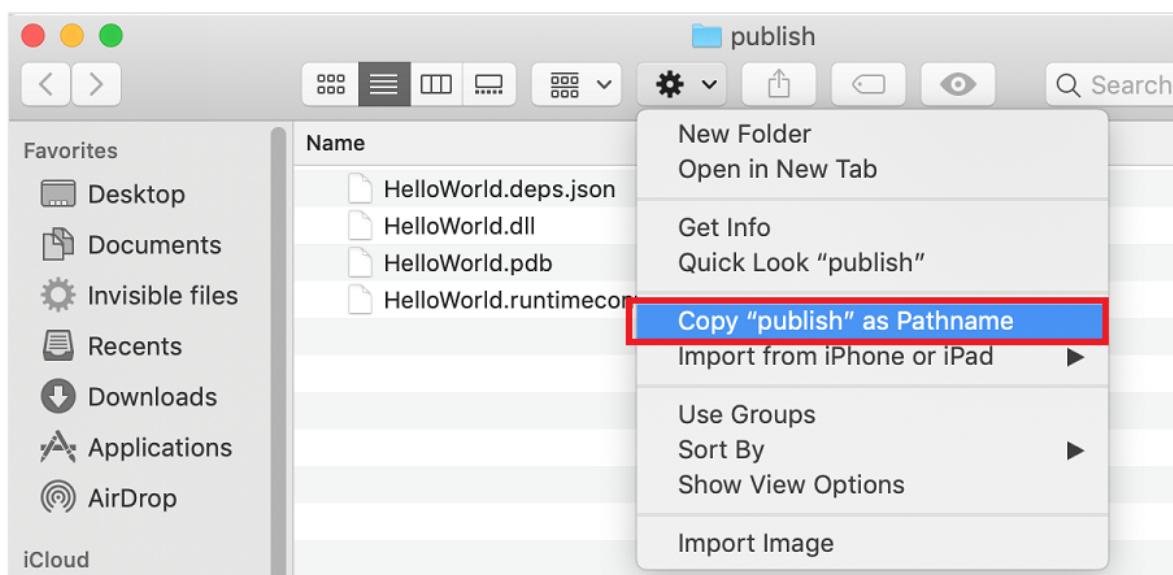
5. In the **Publish to Folder** dialog, select **Publish**.



The publish folder opens, showing the files that were created.



6. Select the gear icon, and select **Copy "publish" as Pathname** from the context menu.



Inspect the files

The publishing process creates a framework-dependent deployment, which is a type of deployment where the published application runs on a machine that has the .NET runtime installed. Users can run the published app by running the `dotnet HelloWorld.dll` command from a command prompt.

As the preceding image shows, the published output includes the following files:

- *HelloWorld.deps.json*

This is the application's runtime dependencies file. It defines the .NET components and the libraries (including the dynamic link library that contains your application) needed to run the app. For more information, see [Runtime configuration files](#).

- *HelloWorld.dll*

This is the [framework-dependent deployment](#) version of the application. To execute this dynamic link library, enter `dotnet HelloWorld.dll` at a command prompt. This method of running the app works on any platform that has the .NET runtime installed.

- *HelloWorld.pdb* (optional for deployment)

This is the debug symbols file. 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*

This is the application's runtime configuration file. It identifies the version of .NET that your application was built to run on. You can also add configuration options to it. For more information, see [.NET runtime configuration settings](#).

Run the published app

1. Open a terminal and navigate to the *publish* folder. To do that, enter `cd` and then paste the path that you copied earlier. For example:

```
Console
```

```
cd ~/Projects/HelloWorld/HelloWorld/bin/Release/net5.0/publish/
```

2. Run the app by using the `dotnet` command:

- a. Enter `dotnet HelloWorld.dll` and press `enter`.

- b. Enter a name in response to the prompt, and press any key to exit.

Additional resources

- [.NET application deployment](#)
- [Publish .NET apps with the .NET CLI](#)
- [dotnet publish](#)

- Tutorial: Publish a .NET console application using Visual Studio Code
- Use the .NET SDK in continuous integration (CI) environments

Next steps

In this tutorial, you published a console app. In the next tutorial, you create a class library.

[Create a .NET library using Visual Studio for Mac](#)

 Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Tutorial: Create a .NET class library using Visual Studio for Mac

Article • 09/08/2023

ⓘ Important

Microsoft has announced the retirement of Visual Studio for Mac. Visual Studio for Mac will no longer be supported starting August 31, 2024. Alternatives include:

- Visual Studio Code with the [C# Dev Kit](#) and related extensions, such as [.NET MAUI](#) and [Unity](#).
- Visual Studio running on Windows in a VM on Mac.
- Visual Studio running on Windows in a [VM in the Cloud](#).

For more information, see [Visual Studio for Mac retirement announcement](#).

In this tutorial, you create a class library that contains a single string-handling method.

A *class library* defines types and methods that are called by an application. If the library targets .NET Standard 2.0, it can be called by any .NET implementation (including .NET Framework) that supports .NET Standard 2.0. If the library targets .NET 5, it can be called by any application that targets .NET 5. This tutorial shows how to target .NET 5.

ⓘ 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](#).

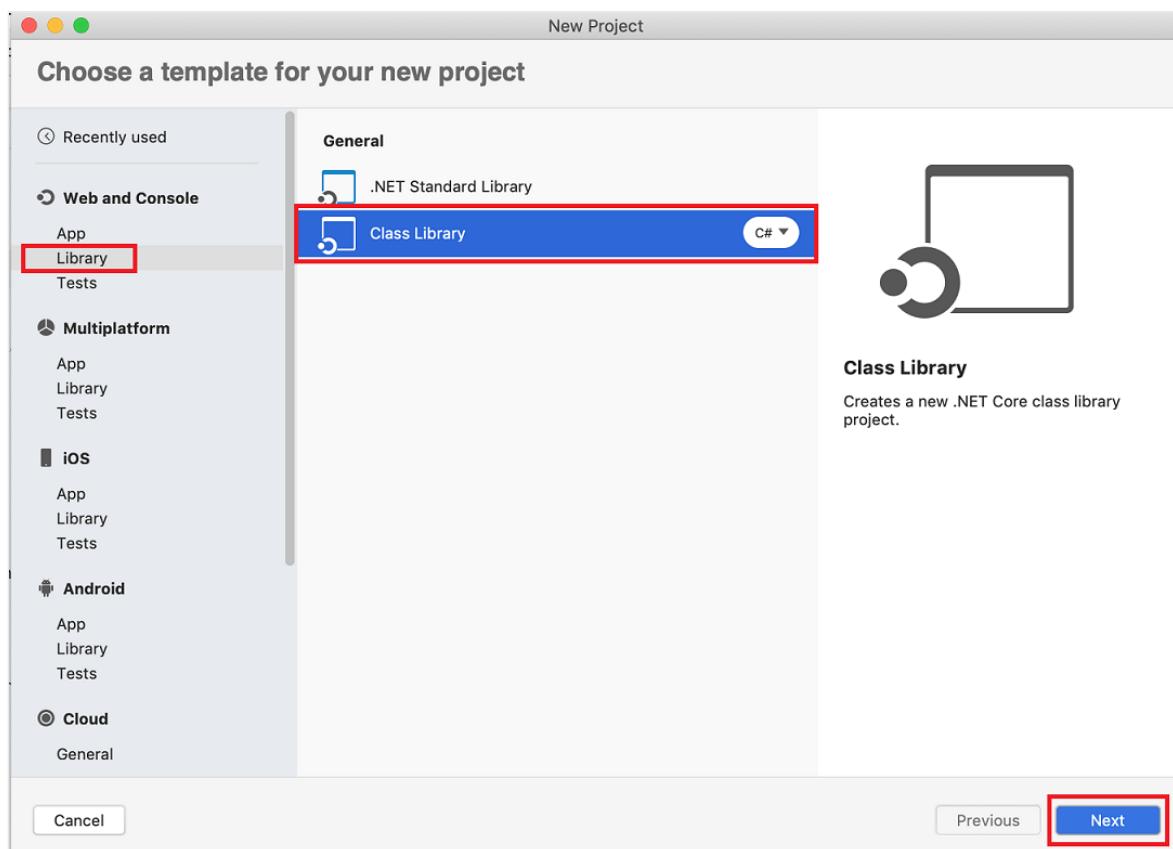
Prerequisites

- [Install Visual Studio for Mac version 8.8 or later](#). Select the option to install .NET Core. Installing Xamarin is optional for .NET development. For more information, see the following resources:
 - [Tutorial: Install Visual Studio for Mac](#).
 - [Supported macOS versions](#).
 - [.NET versions supported by Visual Studio for Mac](#).

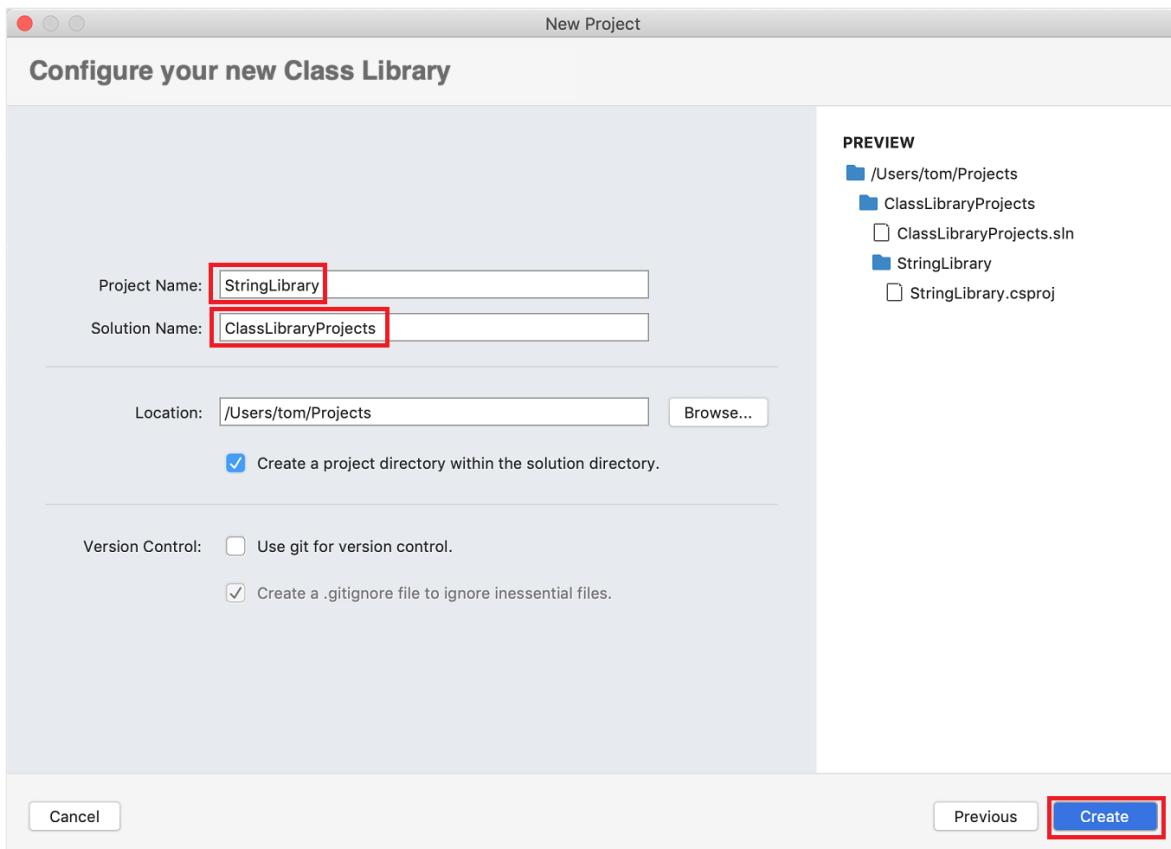
Create a solution with a class library project

A Visual Studio solution serves as a container for one or more projects. Create a solution and a class library project in the solution. You'll add additional, related projects to the same solution later.

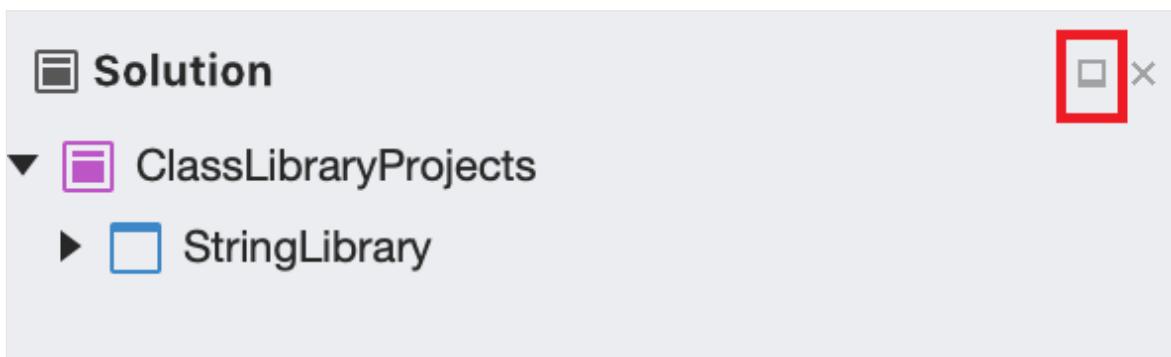
1. Start Visual Studio for Mac.
2. In the start window, select **New Project**.
3. In the **Choose a template for your new project** dialog select **Web and Console > Library > Class Library**, and then select **Next**.



4. In the **Configure your new Class Library** dialog, choose **.NET 5.0**, and select **Next**.
5. Name the project "StringLibrary" and the solution "ClassLibraryProjects". Leave **Create a project directory within the solution directory** selected. Select **Create**.



6. From the main menu, select **View > Solution**, and select the dock icon to keep the pad open.



7. In the **Solution** pad, expand the `StringLibrary` node to reveal the class file provided by the template, `Class1.cs`. `ctrl`-click the file, select **Rename** from the context menu, and rename the file to `StringLibrary.cs`. Open the file and replace the contents with the following code:

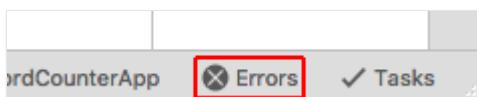
```
C#  
  
using System;  
  
namespace UtilityLibraries  
{  
    public static class StringLibrary  
    {  
        public static bool StartsWithUpper(this string str)  
        {
```

```
        if (string.IsNullOrWhiteSpace(str))
            return false;

        char ch = str[0];
        return char.ToUpper(ch);
    }
}
```

8. Press **⌘S** (**command** + **S**) to save the file.

9. Select **Errors** in the margin at the bottom of the IDE window to open the **Errors** panel. Select the **Build Output** button.



10. Select **Build > Build All** from the menu.

The solution builds. The build output panel shows that the build is successful.

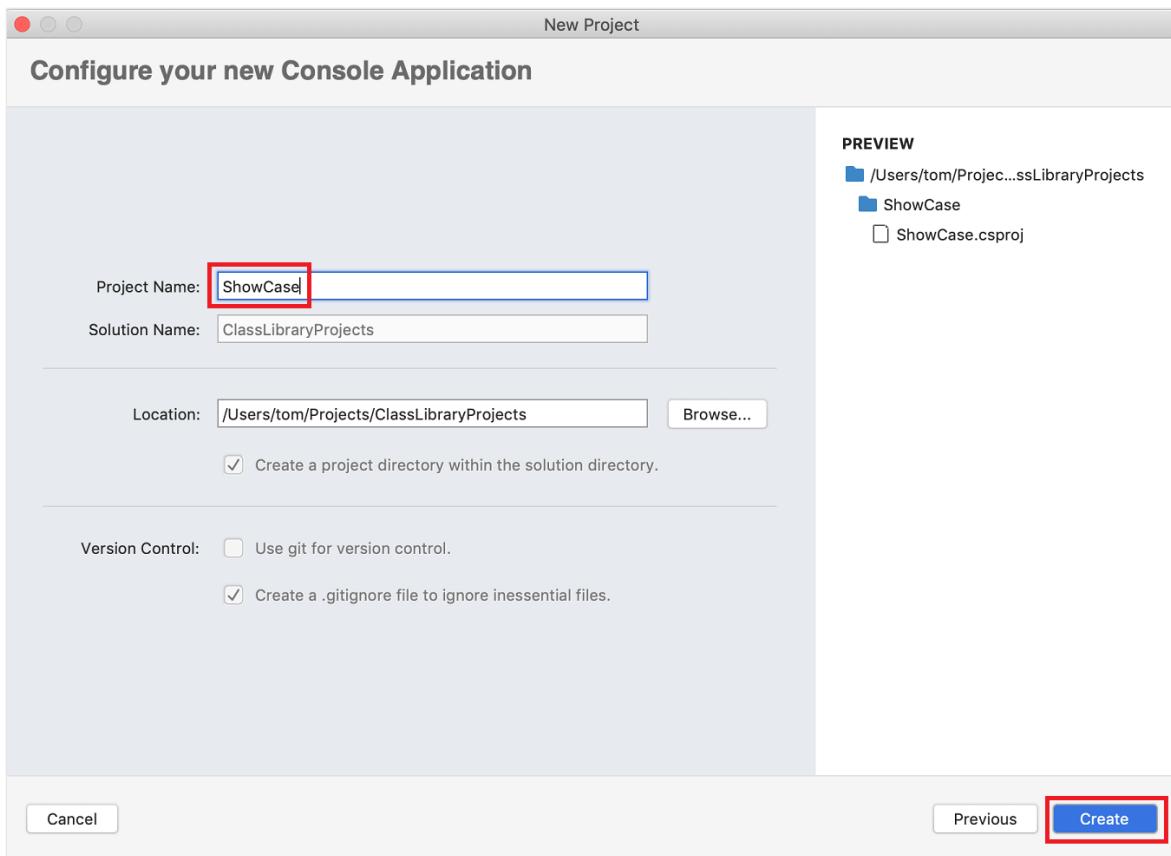
```
Build succeeded.
0 Warning(s)
0 Error(s)

Time Elapsed 00:00:01.50
=========
Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped
=========
Build successful.
```

Add a console app to the solution

Add a console application that uses the class library. The app will prompt the user to enter a string and report whether the string begins with an uppercase character.

1. In the **Solution** pad, **ctrl**-click the **ClassLibraryProjects** solution. Add a new **Console Application** project by selecting the template from the **Web and Console > App** templates, and select **Next**.
2. Select **.NET 5.0** as the **Target Framework** and select **Next**.
3. Name the project **ShowCase**. Select **Create** to create the project in the solution.



4. Open the *Program.cs* file. Replace the code with the following code:

```
C#  
  
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")}{Environment.NewLine}");  
            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($"{Environment.NewLine}Press <Enter> only  
to exit; otherwise, enter a string and press <Enter>:  
{Environment.NewLine}");
            row = 3;
        }
    }
}
```

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 ends, and the console window closes.

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

Add a project reference

Initially, the new console app project doesn't have access to the class library. To allow it to call methods in the class library, create a project reference to the class library project.

1. In the **Solutions** pad, `ctrl`-click the **Dependencies** node of the new **ShowCase** project. In the context menu, select **Add Reference**.
2. In the **References** dialog, select **StringLibrary** and select **OK**.

Run the app

1. `ctrl`-click the **ShowCase** project and select **Run project** from the context menu.
2. Try out the program by entering strings and pressing `enter`, then press `enter` to exit.

Terminal – ShowCase

```
Press <Enter> only to exit; otherwise, enter a string and press <Enter>:
```

```
Begins with uppercase
```

```
Input: Begins with uppercase
```

```
Begins with uppercase? : Yes
```

```
begins with lowercase
```

```
Input: begins with lowercase
```

```
Begins with uppercase? : No
```

Additional resources

- [Develop libraries with the .NET CLI](#)
- [Visual Studio 2019 for Mac Release Notes](#)
- [.NET Standard versions and the platforms they support.](#)

Next steps

In this tutorial, you created a solution and a library project, and added a console app project that uses the library. In the next tutorial, you add a unit test project to the solution.

[Test a .NET class library using Visual Studio for Mac](#)

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).



.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Test a .NET class library using Visual Studio

Article • 09/08/2023

ⓘ Important

Microsoft has announced the retirement of Visual Studio for Mac. Visual Studio for Mac will no longer be supported starting August 31, 2024. Alternatives include:

- Visual Studio Code with the [C# Dev Kit](#) and related extensions, such as [.NET MAUI](#) and [Unity](#).
- Visual Studio running on Windows in a VM on Mac.
- Visual Studio running on Windows in a [VM in the Cloud](#).

For more information, see [Visual Studio for Mac retirement announcement](#).

This tutorial shows how to automate unit testing by adding a test project to a solution.

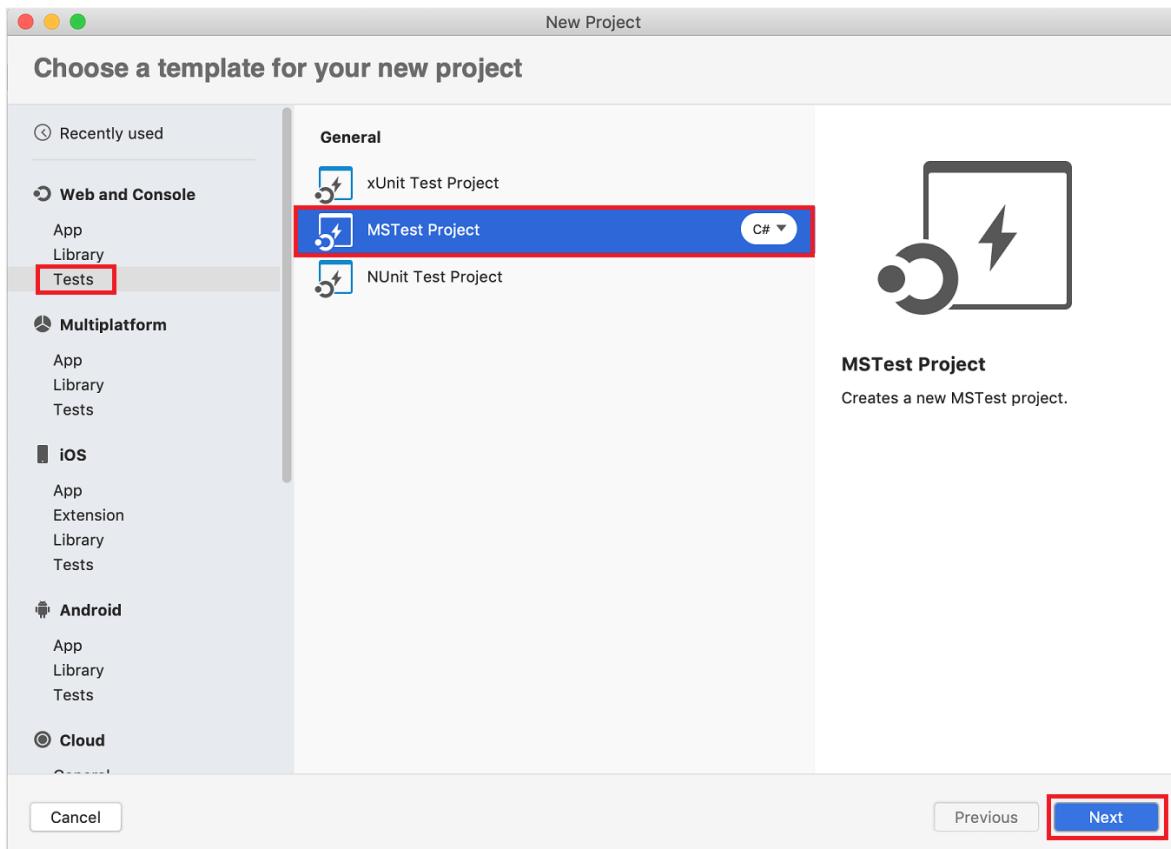
Prerequisites

- This tutorial works with the solution that you create in [Create a .NET class library using Visual Studio for Mac](#).

Create a unit test project

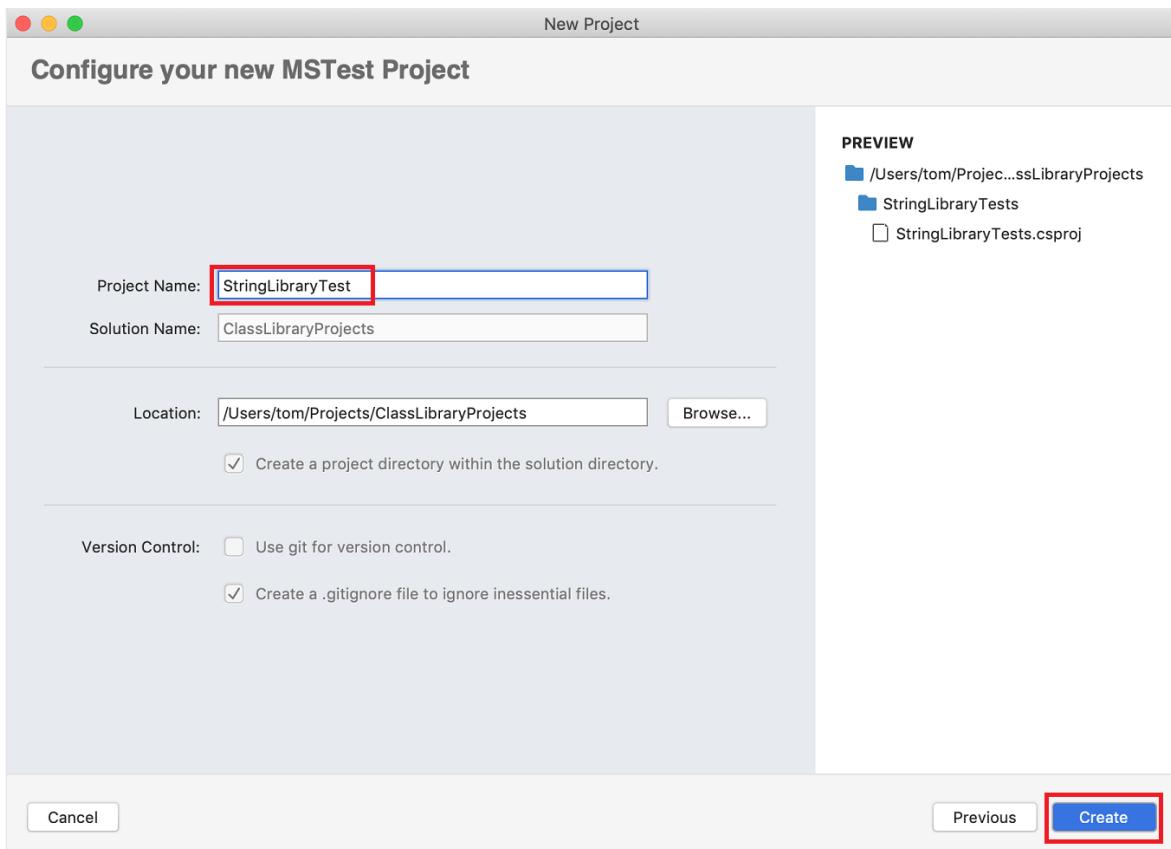
Unit tests provide automated software testing during your development and publishing. [MSTest](#) is one of three test frameworks you can choose from. The others are [xUnit](#) and [nUnit](#).

1. Start Visual Studio for Mac.
2. Open the `ClassLibraryProjects` solution you created in [Create a .NET class library using Visual Studio for Mac](#).
3. In the **Solution** pad, **ctrl**-click the `ClassLibraryProjects` solution and select **Add > New Project**.
4. In the **New Project** dialog, select **Tests** from the **Web and Console** node. Select the **MSTest Project** followed by **Next**.



5. Select .NET 5.0 as the Target Framework and select **Next**.

6. Name the new project "StringLibraryTest" and select **Create**.



Visual Studio creates a class file with the following code:

C#

```
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:

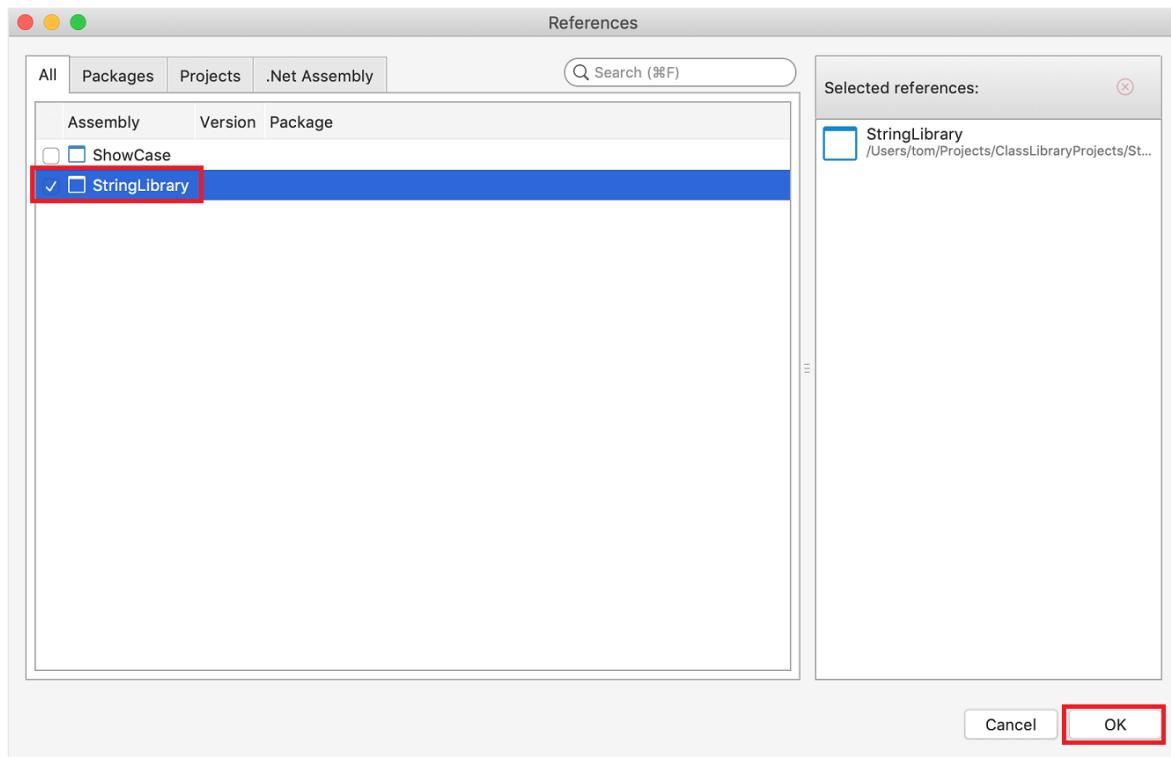
- It imports the `Microsoft.VisualStudio.TestTools.UnitTesting` namespace, which contains the types used for unit testing.
- It applies the `TestClassAttribute` attribute to the `UnitTest1` class.
- It applies the `TestMethodAttribute` attribute to `TestMethod1`.

Each method tagged with `[TestMethod]` in a test class tagged with `[TestClass]` is executed automatically when the unit test is run.

Add a project reference

For the test project to work with the `StringLibrary` class, add a reference to the `StringLibrary` project.

1. In the **Solution** pad, **ctrl**-click **Dependencies** under `StringLibraryTest`. Select **Add Reference** from the context menu.
2. In the **References** dialog, select the `StringLibrary` project. Select **OK**.



Add and run unit test methods

When Visual Studio runs a unit test, it executes each method that is marked with the [TestMethodAttribute](#) attribute in a class that is marked with the [TestClassAttribute](#) attribute. A test method ends when the first failure is found 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 the [Assert](#) class's most frequently called methods are shown in the following table:

Assert methods	Function
<code>Assert.AreEqual</code>	Verifies that two values or objects are equal. The assert fails if the values or objects aren't equal.
<code>Assert.AreSame</code>	Verifies that two object variables refer to the same object. The assert fails if the variables refer to different objects.
<code>Assert.IsFalse</code>	Verifies that a condition is <code>false</code> . The assert fails if the condition is <code>true</code> .
<code>Assert.IsNotNull</code>	Verifies that an object isn't <code>null</code> . The assert fails if the object is <code>null</code> .

You can also use the [Assert.ThrowsException](#) method in a test method to indicate the type of exception it's expected to throw. The test fails if the specified exception isn't 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 `Assert.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 `Assert.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 hasn't been initialized. You can call `StartsWithUpper` directly as a static method and pass a single `String` argument. Or you can call `StartsWithUpper` as an extension method on a `string` variable assigned to `null`.

You'll define three methods, each of which calls an `Assert` method for each element in a string array. You'll call a method overload that lets you specify an error message to be displayed in case of test failure. The message identifies the string that caused the failure.

To create the test methods:

1. Open the `UnitTest1.cs` file and replace the code with the following code:

```
C#  
  
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", "Aθήνα",  
"Москва" };  
            foreach (var word in words)  
            {  
                bool result = word.StartsWithUpper();  
                Assert.IsTrue(result,  
                    string.Format("Expected for '{0}': true; Actual:  
{1}",  
                                word, result));  
            }  
        }  
    }  
}
```

```

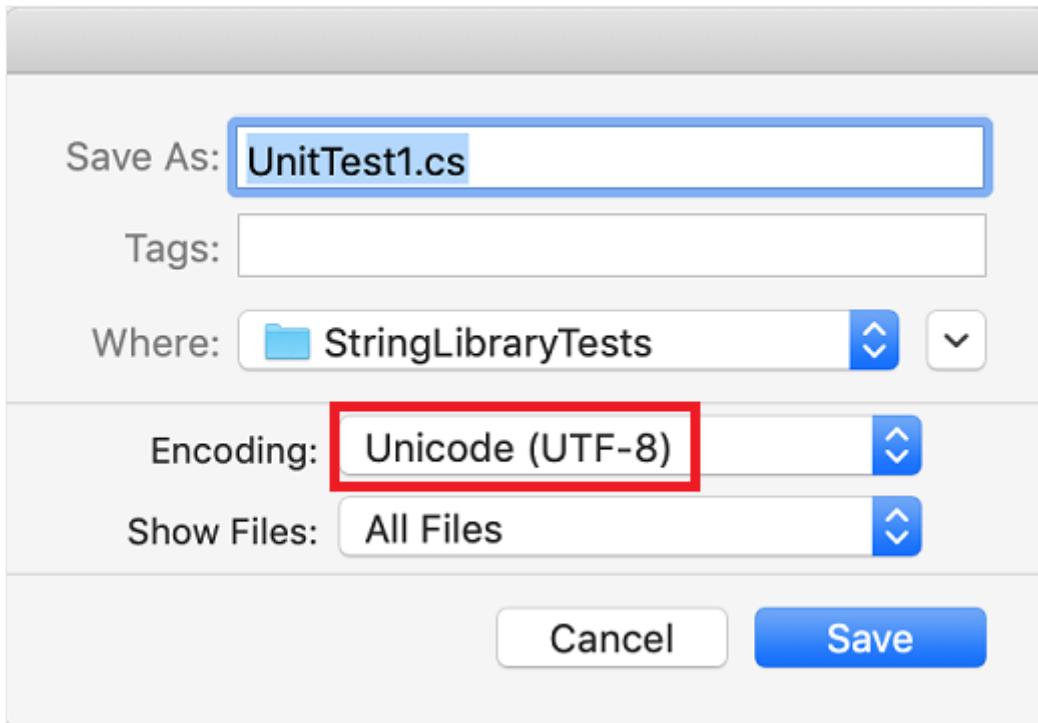
[TestMethod]
public void TestDoesNotStartWithUpper()
{
    // Tests that we expect to return false.
    string[] words = { "alphabet", "zebra", "abc",
"αυτοκινητοβιομηχανία", "государство",
"1234", ".", ";", " " };
    foreach (var word in words)
    {
        bool result = word.StartsWithUpper();
        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));
    }
}

```

The test of uppercase characters in the `TestStartsWithUpper` method includes the Greek capital letter alpha (U+0391) and the Cyrillic capital letter EM (U+041C). 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).

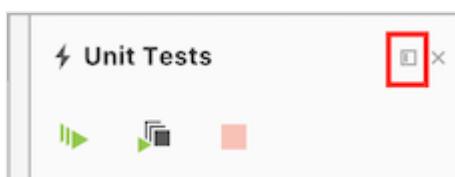
2. On the menu bar, select **File > Save As**. In the dialog, make sure that **Encoding** is set to **Unicode (UTF-8)**.



3. When you're asked if you want to replace the existing file, select **Replace**.

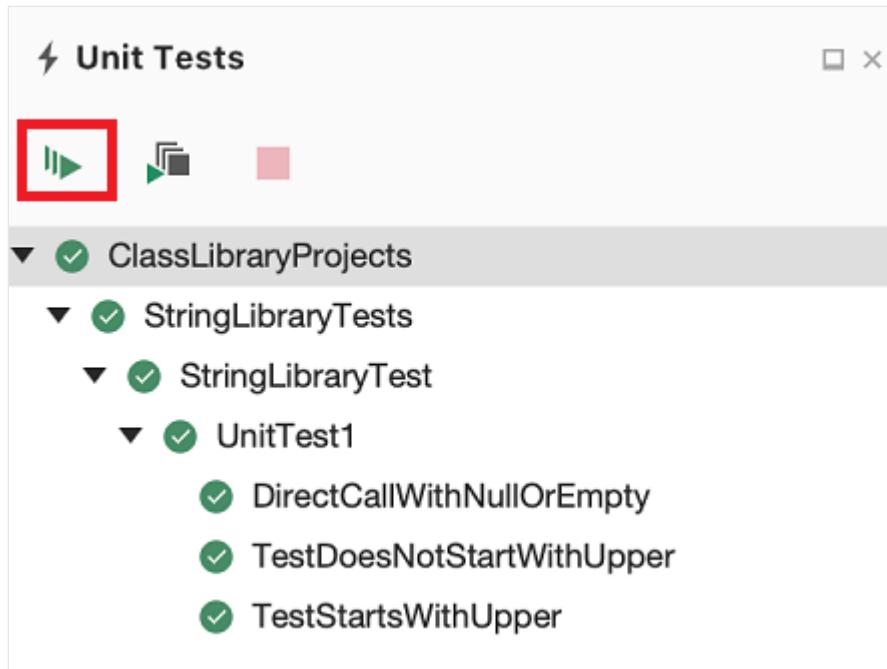
If you fail to save your source code as a UTF8-encoded file, Visual Studio may save it as an ASCII file. When that happens, the runtime doesn't accurately decode the UTF8 characters outside of the ASCII range, and the test results won't be correct.

4. Open the **Unit Tests** panel on the right side of the screen. Select **View > Tests** from the menu.
5. Click the **Dock** icon to keep the panel open.



6. Click the **Run All** button.

All tests pass.



Handle test failures

If you're doing test-driven development (TDD), you write tests first and they fail the first time you run them. Then you add code to the app that makes the test succeed. For this tutorial, you created the test after writing the app code that it validates, so you haven't seen the test fail. To validate that a test fails when you expect it to fail, add an invalid value to the test input.

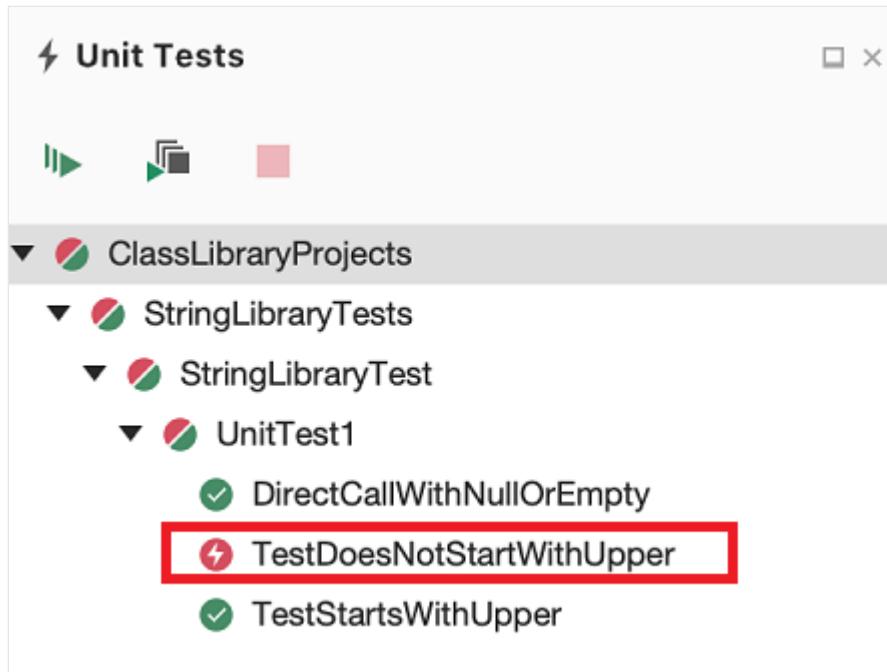
1. Modify the `words` array in the `TestDoesNotStartWithUpper` method to include the string "Error". You don't need to save the file because Visual Studio automatically saves open files when a solution is built to run tests.

C#

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

2. Run the tests again.

This time, the **Test Explorer** window indicates that two tests succeeded and one failed.



3. `ctrl`-click the failed test, `TestDoesNotStartWithUpper`, and select **Show Results Pad** from the context menu.

The **Results** pad displays the message produced by the assert: "Assert.IsFalse failed. Expected for 'Error': false; actual: True". Because of the failure, no strings in the array after "Error" were tested.

⚡ Unit Tests

ClassLibraryProjects

StringLibraryTests

StringLibraryTest

UnitTest1

DirectCallWithNullOrEmpty

TestDoesNotStartWithUpper

TestStartsWithUpper

TestDoesNotStartWithUpper

Result Output

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

at StringLibraryTest.UnitTest1.TestDoesNotStartWithUpper() in /

4/6

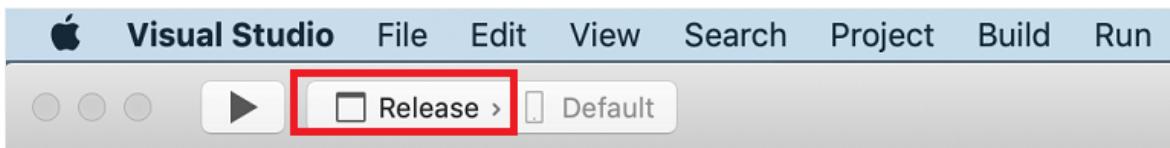
4. Remove the string "Error" that you added in step 1. Rerun the test and the tests pass.

Test the Release version of the library

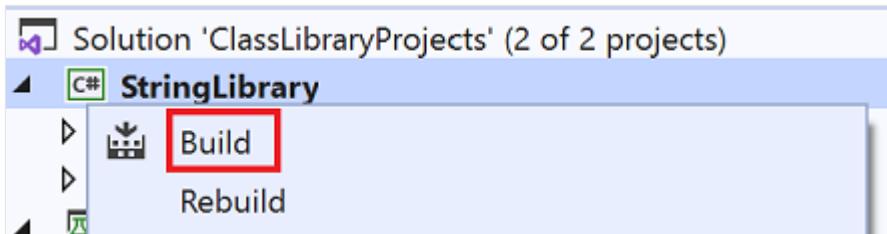
Now that the tests have all passed when running the Debug build of the library, run the tests an additional time against the Release build of the library. A number of factors, including compiler optimizations, can sometimes produce different behavior between Debug and Release builds.

To test the Release build:

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



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



3. Run the unit tests again.

The tests pass.

Debug tests

If you're using Visual Studio for Mac as your IDE, you can use the same process shown in [Tutorial: Debug a .NET console application using Visual Studio for Mac](#) to debug code using your unit test project. Instead of starting the *ShowCase* app project, **ctrl**-click the **StringLibraryTests** project, and select **Start Debugging Project** from the context menu.

Visual Studio starts the test project with the debugger attached. Execution will stop at any breakpoint you've added to the test project or the underlying library code.

Additional resources

- [Unit testing in .NET](#)

Next steps

In this tutorial, you unit tested a class library. You can make the library available to others by publishing it to [NuGet](#) as a package. To learn how, follow a NuGet tutorial:

[Create and publish a package \(dotnet CLI\)](#)

If you publish a library as a NuGet package, others can install and use it. To learn how, follow a NuGet tutorial:

[Install and use a package in Visual Studio for Mac](#)

A library doesn't have to be distributed as a package. It can be bundled with a console app that uses it. To learn how to publish a console app, see the earlier tutorial in this series:

Publish a .NET console application using Visual Studio for Mac

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

.NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)

Quickstart: Install and use a package in Visual Studio for Mac

Article • 09/03/2019

NuGet packages contain reusable code that other developers make available to you for use in your projects. See [What is NuGet?](#) for background. Packages are installed into a Visual Studio for Mac project using the NuGet Package Manager. This article demonstrates the process using the popular [Newtonsoft.Json](#) package and a .NET Core console project. The same process applies to any other Xamarin or .NET Core project.

Once installed, refer to the package in code with `using <namespace>` where `<namespace>` is specific to the package you're using. Once the reference is made, you can call the package through its API.

Tip

Start with nuget.org: Browsing *nuget.org* is how .NET developers typically find components they can reuse in their own applications. You can search *nuget.org* directly or find and install packages within Visual Studio as shown in this article. For general information, see [Find and evaluate NuGet packages](#).

Prerequisites

- Visual Studio 2019 for Mac.

You can install the 2019 Community edition for free from [visualstudio.com](#) or use the Professional or Enterprise editions.

If you're using Visual Studio on Windows, see [Install and use a package in Visual Studio \(Windows Only\)](#).

Create a project

NuGet packages can be installed into any .NET project, provided that the package supports the same target framework as the project.

For this walkthrough, use a simple .NET Core Console app. Create a project in Visual Studio for Mac using **File > New Solution...**, select the **.NET Core > App > Console**

Application template. Click **Next**. Accept the default values for **Target Framework** when prompted.

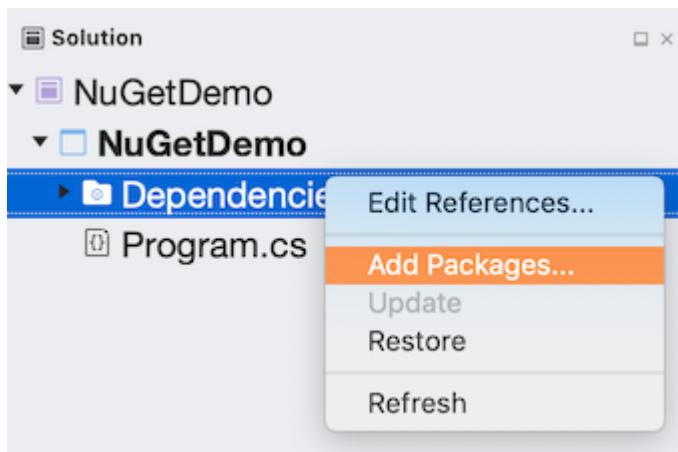
Visual Studio creates the project, which opens in Solution Explorer.

Add the Newtonsoft.Json NuGet package

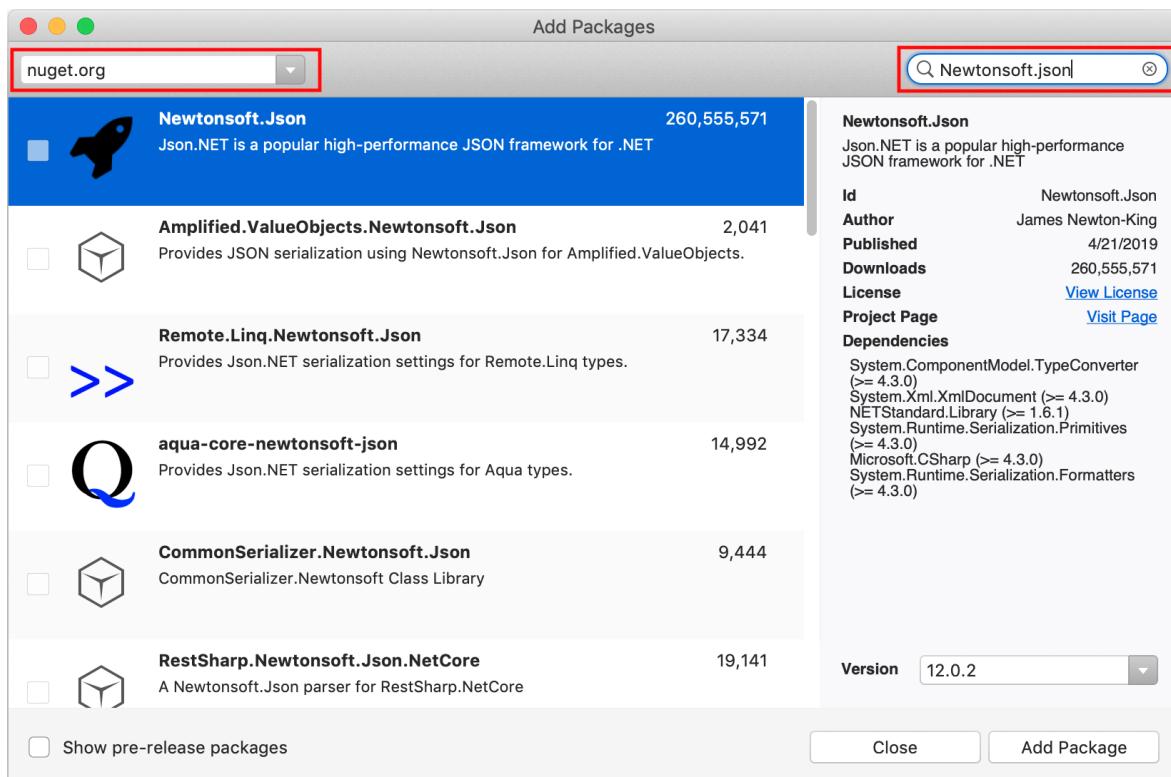
To install the package, you use the NuGet Package Manager. When you install a package, NuGet records the dependency in either your project file or a `packages.config` file (depending on the project format). For more information, see [Package consumption overview and workflow](#).

NuGet Package Manager

1. In Solution Explorer, right-click **Dependencies** and choose **Add Packages...**



2. Choose "nuget.org" as the **Package source** in the top left corner of the dialog, and search for **Newtonsoft.Json**, select that package in the list, and select **Add Packages...**:



If you want more information on the NuGet Package Manager, see [Install and manage packages using Visual Studio for Mac](#).

Use the Newtonsoft.Json API in the app

With the Newtonsoft.Json package in the project, you can call its `JsonConvert.SerializeObject` method to convert an object to a human-readable string.

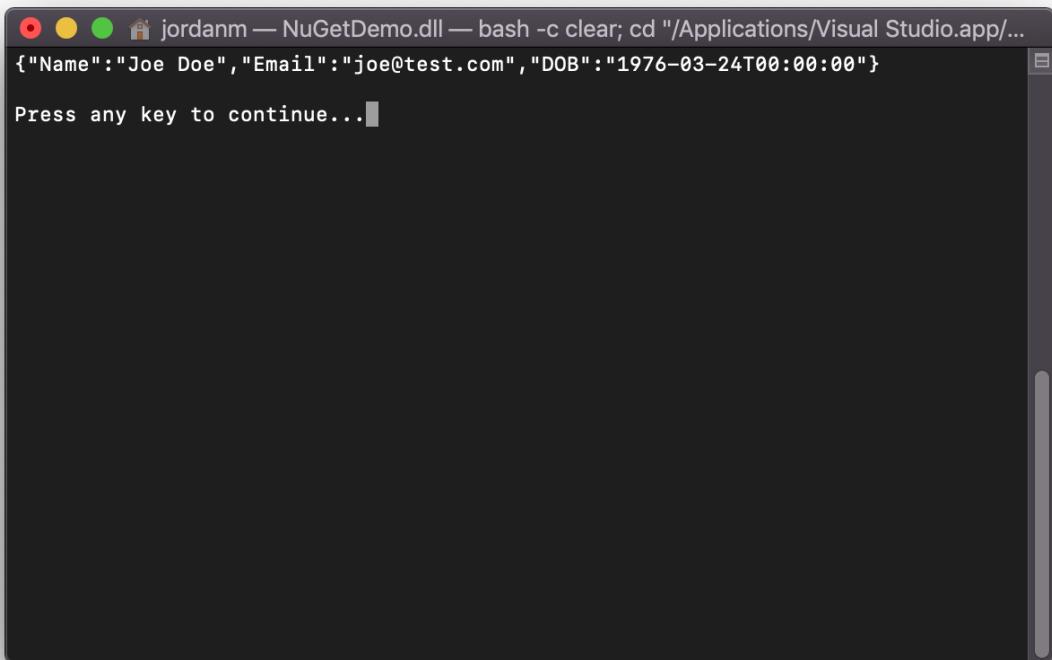
1. Open the `Program.cs` file (located in the Solution Pad) and replace the file contents with the following code:

```
C#  
  
using System;  
using Newtonsoft.Json;  
  
namespace NuGetDemo  
{  
    public class Account  
    {  
        public string Name { get; set; }  
        public string Email { get; set; }  
        public DateTime DOB { get; set; }  
    }  
  
    class Program  
    {  
        static void Main(string[] args)  
        {  
        }  
    }  
}
```

```
Account account = new Account()
{
    Name = "Joe Doe",
    Email = "joe@test.com",
    DOB = new DateTime(1976, 3, 24)
};
string json = JsonConvert.SerializeObject(account);
Console.WriteLine(json);
}
```

2. Build and run the app by selecting **Run > Start Debugging**:

3. Once the app runs, you'll see the serialized JSON output appear in the console:



```
{"Name":"Joe Doe", "Email":"joe@test.com", "DOB":"1976-03-24T00:00:00"}  
Press any key to continue...
```

Next steps

Congratulations on installing and using your first NuGet package!

[Install and manage packages using Visual Studio for Mac](#)

To explore more that NuGet has to offer, select the links below.

- [Overview and workflow of package consumption](#)
- [Package references in project files](#)

Learn .NET and the .NET SDK tools by exploring these tutorials

Article • 09/08/2023

The following tutorials show how to develop console apps and libraries for .NET Core, .NET 5, and later versions. For other types of applications, see [Tutorials for getting started with .NET](#).

Use Visual Studio

- [Create a console app](#)
- [Debug an app](#)
- [Publish an app](#)
- [Create a class library](#)
- [Unit test a class library](#)
- [Install and use a package](#)
- [Create and publish a package](#)
- [Create an F# console app](#)

Use Visual Studio Code

Choose these tutorials if you want to use Visual Studio Code or some other code editor. All of them use the CLI for .NET Core development tasks, so all except the debugging tutorial can be used with any code editor.

- [Create a console app](#)
- [Debug an app](#)
- [Publish an app](#)
- [Create a class library](#)
- [Unit test a class library](#)
- [Install and use a package](#)
- [Create and publish a package](#)
- [Create an F# console app](#)

Advanced articles

- [How to create libraries](#)
- [Unit test an app with xUnit](#)

- Unit test using C#/VB/F# with NUnit/xUnit/MSTest
- Live unit test with Visual Studio
- Create templates for the CLI
- Create and use tools for the CLI
- Create an app with plugins

 **Collaborate with us on GitHub**

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see [our contributor guide](#).

 .NET

.NET feedback

The .NET documentation is open source. Provide feedback here.

 [Open a documentation issue](#)

 [Provide product feedback](#)