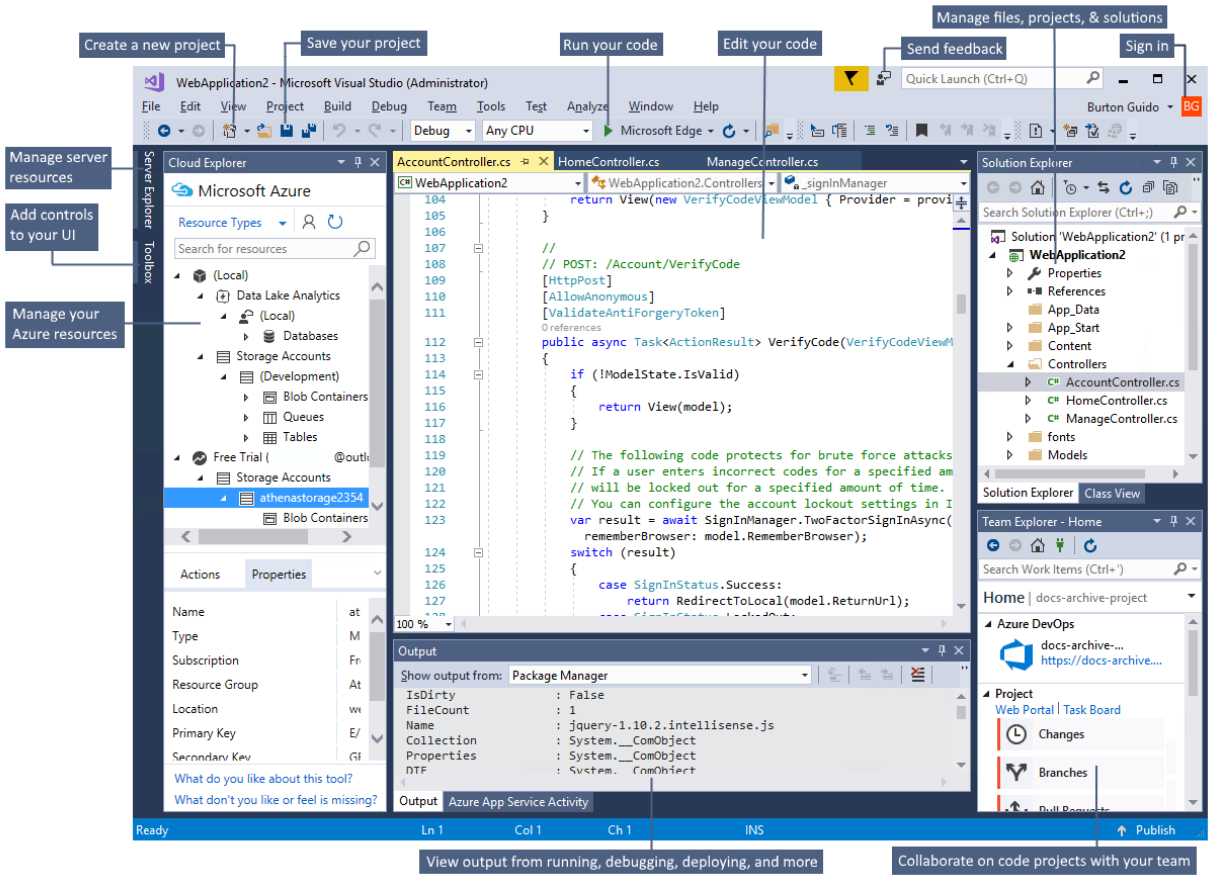
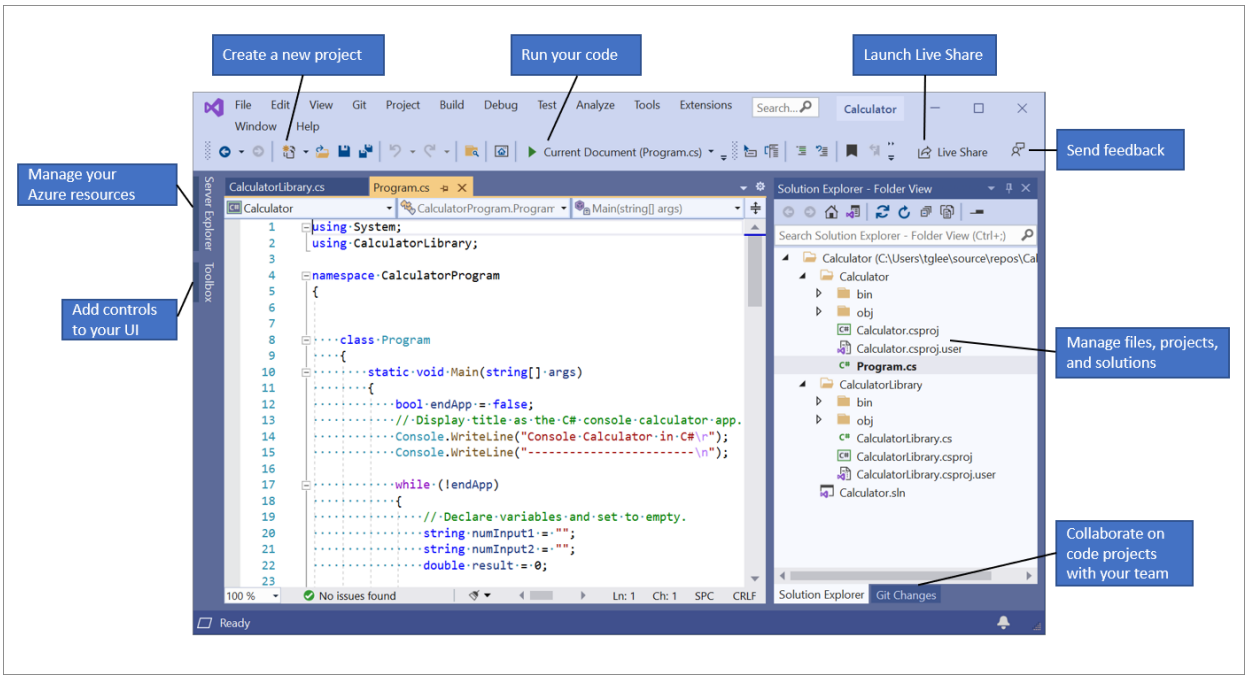
Contents

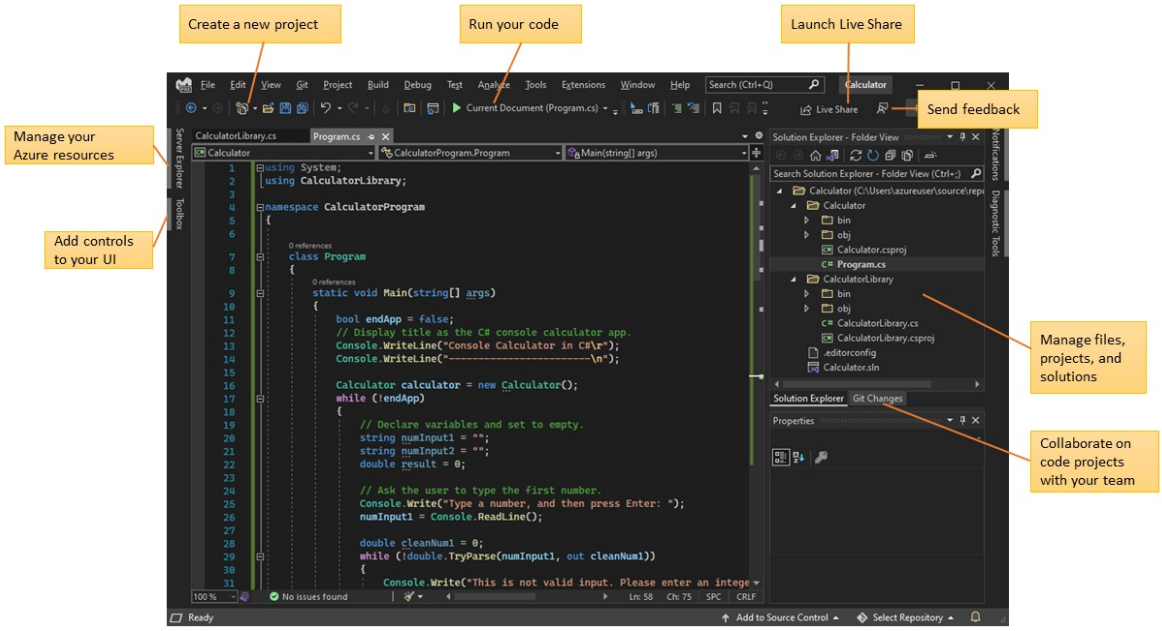
Welcome to the Visual Studio IDE | C#

10/28/2021 • 16 minutes to read • Edit Online

An *integrated development environment* (IDE) is a feature-rich program that supports many aspects of software development. The Visual Studio IDE is a creative launching pad that you can use to edit, debug, and build code, and then publish an app. Over and above the standard editor and debugger that most IDEs provide, Visual Studio includes compilers, code completion tools, graphical designers, and many more features to enhance the software development process.







The preceding image shows Visual Studio with an open project that shows key windows and their functionality:

- In Solution Explorer, at upper right, you can view, navigate, and manage your code files. **Solution Explorer** can help organize your code by grouping the files into solutions and projects.
- The central editor window, where you'll probably spend most of your time, displays file contents. In the editor window, you can edit code or design a user interface such as a window with buttons and text boxes.
- The Output window (bottom center) is where Visual Studio sends notifications such as debugging and error messages, compiler warnings, publishing status messages, and more. Each message source has its own tab.
- In Git Changes at lower right, you can track work items and share code with others by using version control technologies like Git and GitHub.

Editions

Visual Studio is available for Windows and Mac. Visual Studio for Mac has many of the same features as Visual

Studio for Windows, and is optimized for developing cross-platform and mobile apps. This article focuses on the Windows version of Visual Studio.

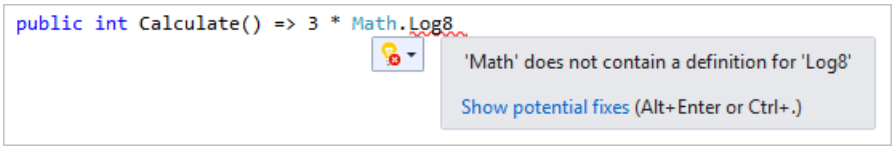
There are three editions of Visual Studio: Community, Professional, and Enterprise. See Compare Visual Studio editions to learn about which features are supported in each edition.

Popular productivity features

Some popular features in Visual Studio that improve your productivity when developing software include:

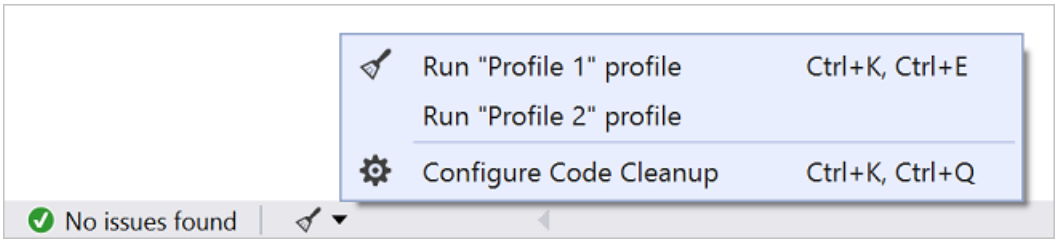
- Squiggles and Quick Actions

  Squiggles are wavy underlines that alert you to errors or potential problems in your code as you type. These visual clues help you fix problems immediately, without waiting to discover errors during build or runtime. If you hover over a squiggle, you see more information about the error. A lightbulb might also appear in the left margin showing *Quick Actions* you can take to fix the error.



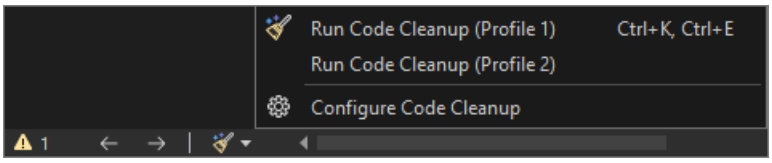
- Code Cleanup

  With the click of a button, you can format your code and apply any code fixes suggested by your code style settings, .editorconfig conventions, and Roslyn analyzers. **Code Cleanup**, currently available for C# code only, helps you resolve issues in your code before it goes to code review.



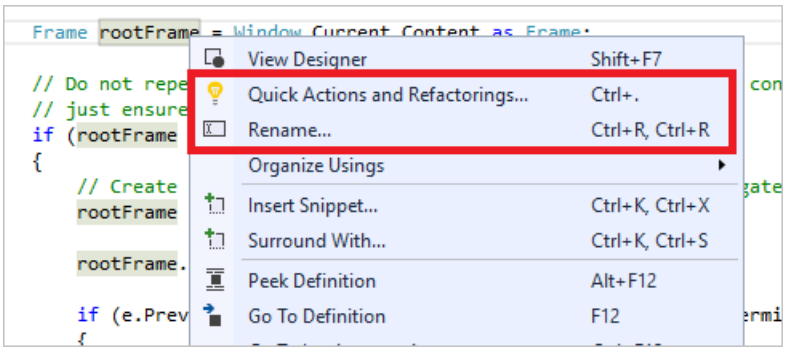
- Code Cleanup

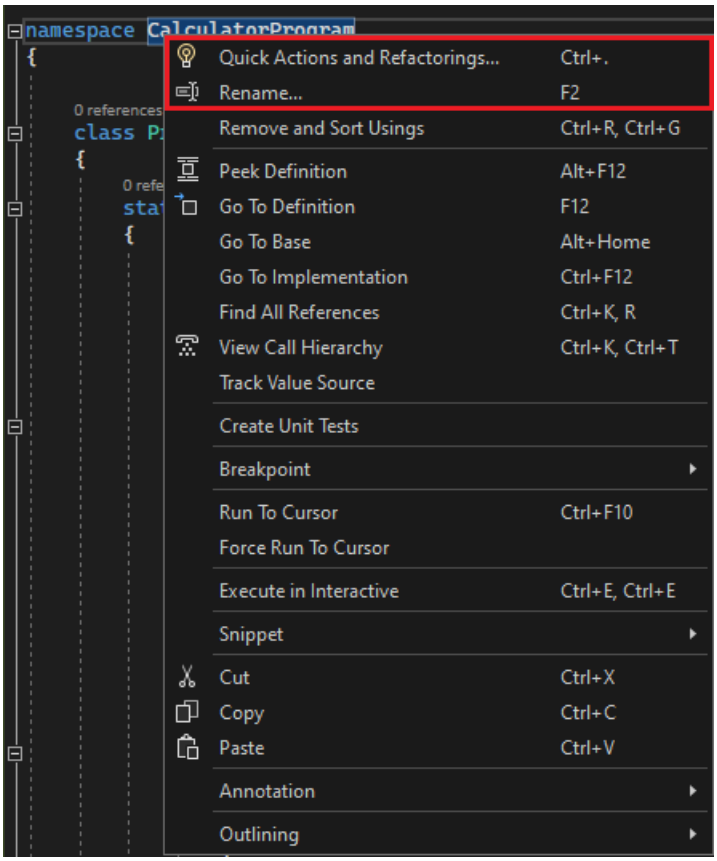
  With the click of a button, you can format your code and apply any code fixes suggested by your code style settings, .editorconfig conventions, and Roslyn analyzers. **Code Cleanup**, currently available for C# code only, helps you resolve issues in your code before it goes to code review.



- Refactoring

  Refactoring includes operations such as intelligent renaming of variables, extracting one or more lines of code into a new method, and changing the order of method parameters.

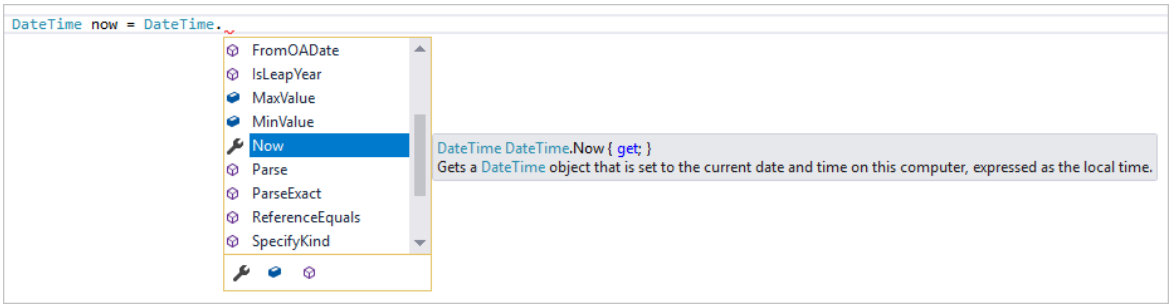


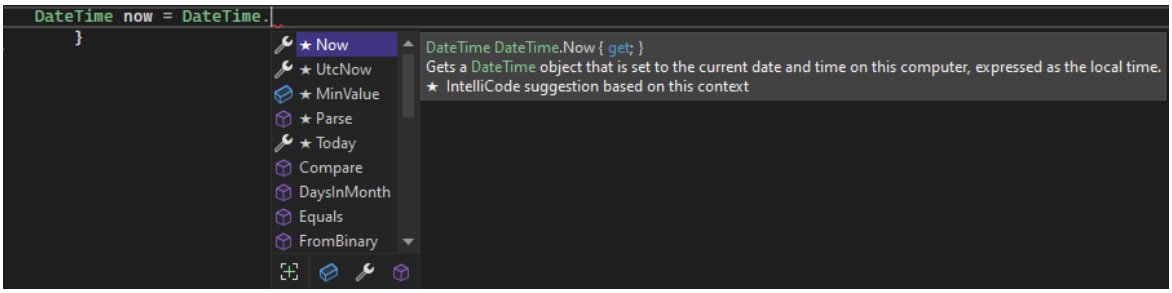


- IntelliSense

  IntelliSense is a set of features that display information about your code directly in the editor and, in some cases, write small bits of code for you. It's like having basic documentation inline in the editor, so you don't have to look up type information elsewhere.

  The following illustration shows how IntelliSense displays a member list for a type:

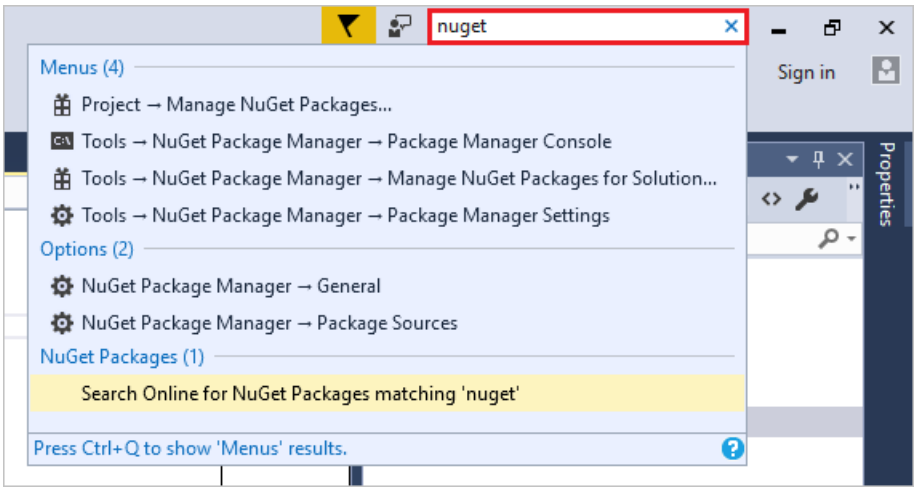




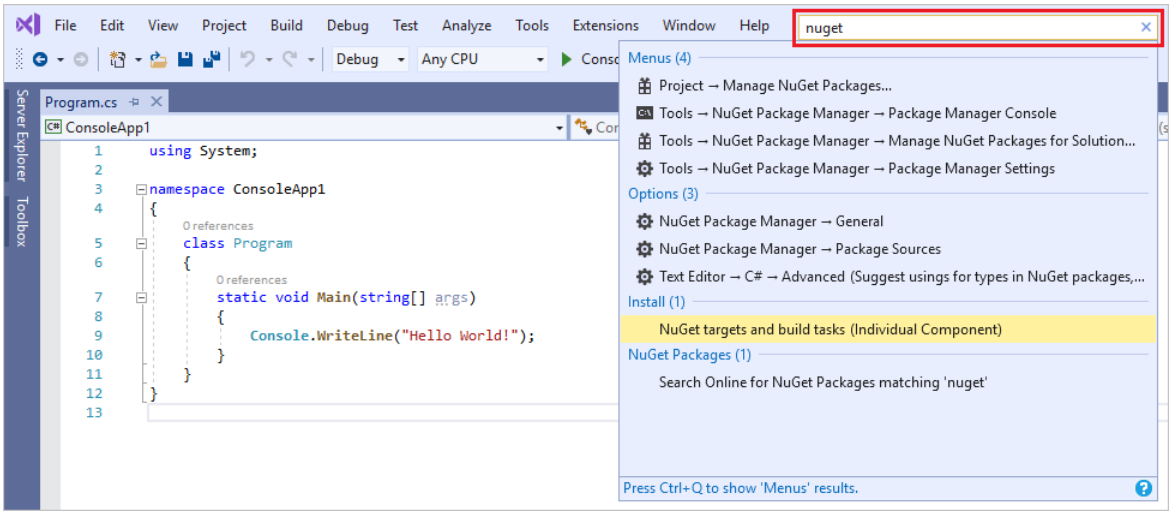
IntelliSense features vary by language. For more information, see C# IntelliSense, Visual C++ IntelliSense, JavaScript IntelliSense, and Visual Basic IntelliSense.

- Visual Studio search

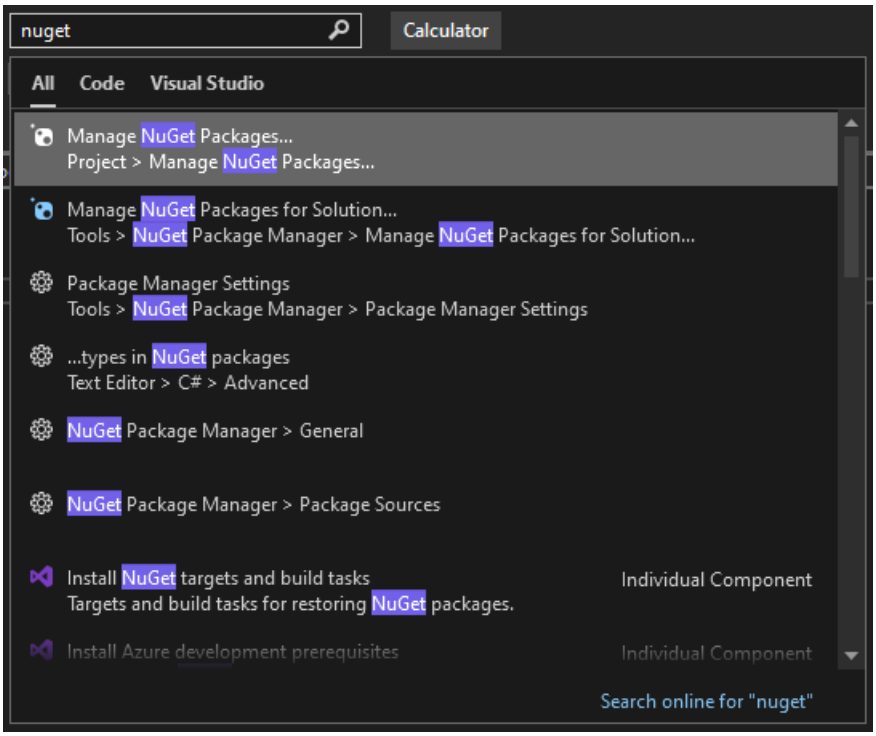
  Visual Studio menus, options, and properties can seem overwhelming at times. Visual Studio search, or **Ctrl+Q**, is a great way to rapidly find IDE features and code in one place.



For more information, see Quick Launch.



For information and productivity tips, see How to use Visual Studio search.



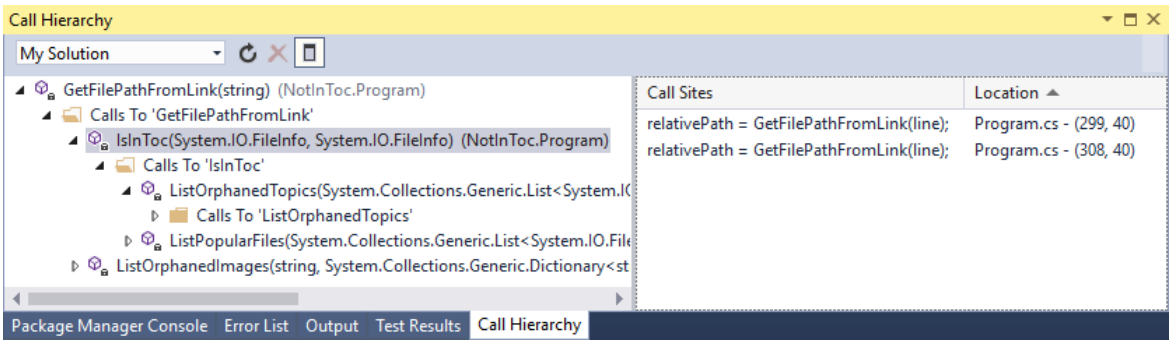
For information and productivity tips, see How to use Visual Studio search.

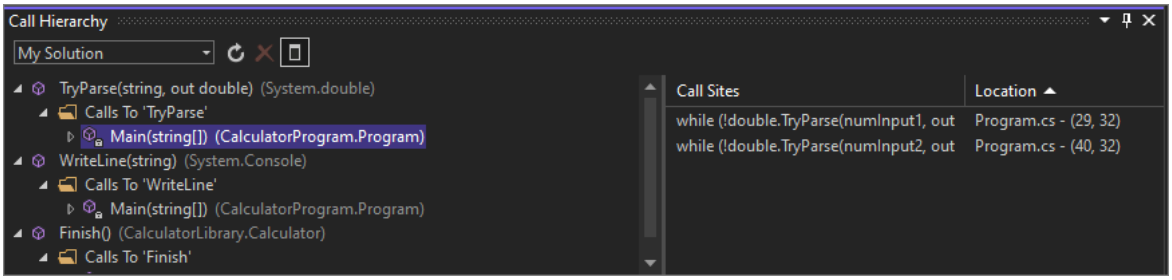
- Live Share

  Collaboratively edit and debug with others in real time, regardless of your app type or programming language. You can instantly and securely share your project. You can also share debugging sessions, terminal instances, localhost web apps, voice calls, and more.

- Call Hierarchy

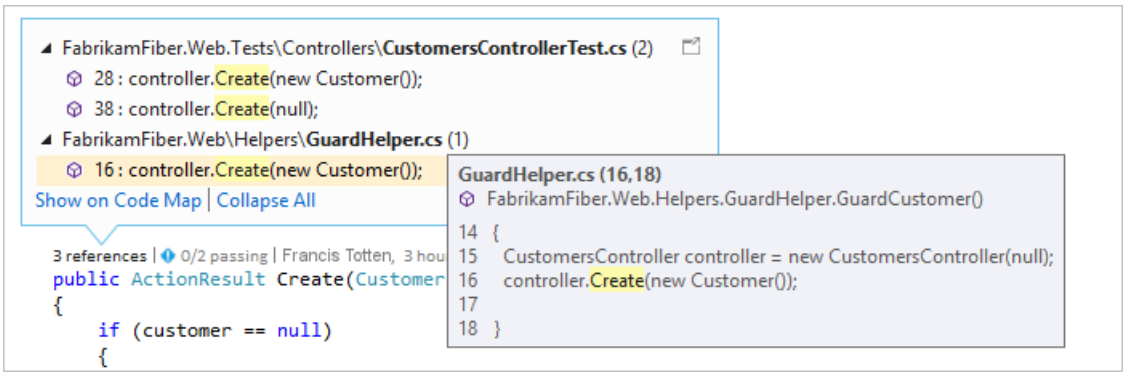
  The **Call Hierarchy** window shows the methods that call a selected method. This information can be useful when you're thinking about changing or removing the method, or when you're trying to track down a bug.

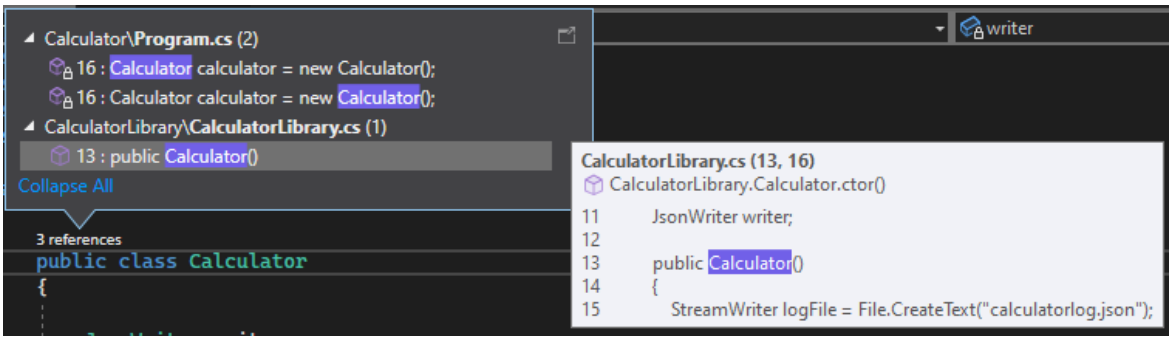




- CodeLens

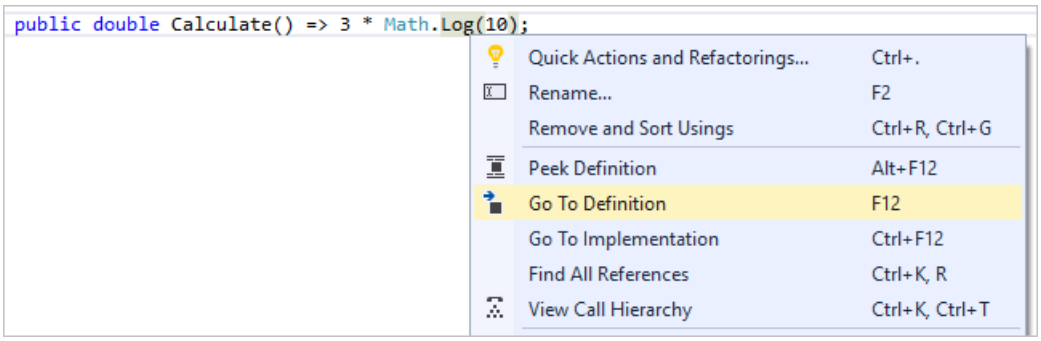
  CodeLens helps you find code references, code changes, linked bugs, work items, code reviews, and unit tests, without leaving the editor.

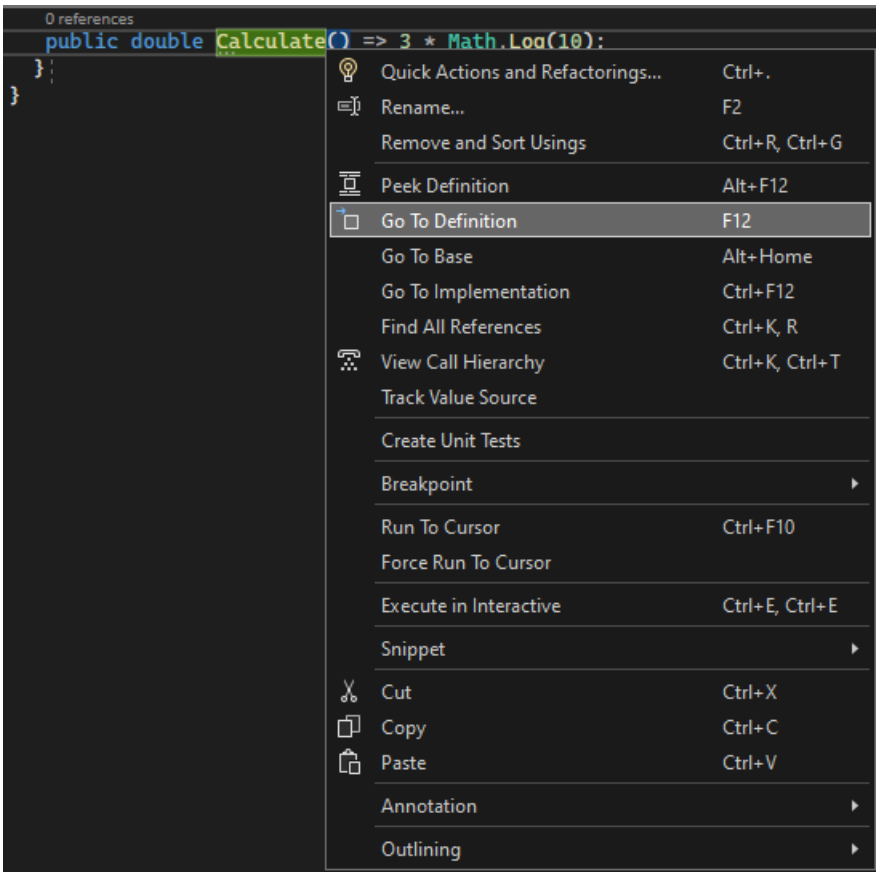




- Go To Definition

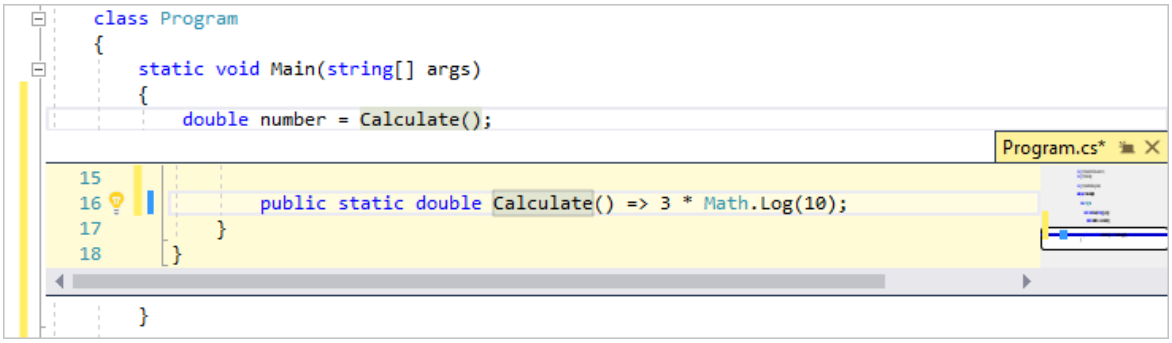
  The **Go To Definition** feature takes you directly to the location of a function or type definition.

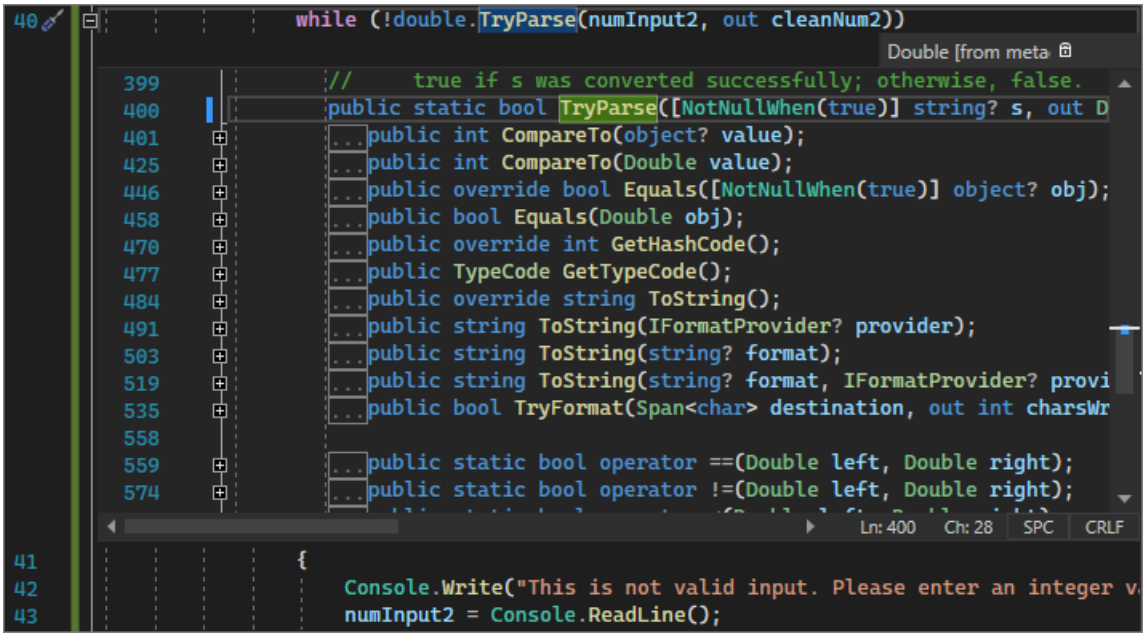




- Peek Definition

  The **Peek Definition** window shows a method or type definition without opening a separate file.



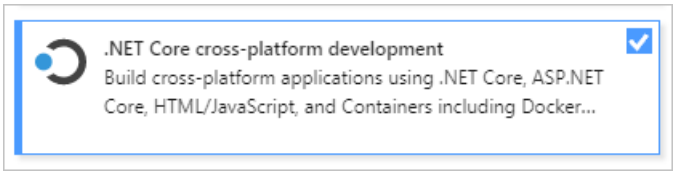


Install Visual Studio

In this section, you create a simple project to try out some of the things you can do with Visual Studio. You use IntelliSense as a coding aid, debug an app to see a variable value during app execution, and change the color theme.

To get started, download Visual Studio and install it on your system. The modular installer enables you to choose and install *workloads*, which are groups of features needed for the programming language or platform you prefer. To follow the steps for creating a program, be sure to select the **.NET Core cross-platform development** workload during installation.

To get started, download Visual Studio and install it on your system. The modular installer enables you to choose and install *workloads*, which are groups of features needed for the programming languages or platforms you want. To follow the steps to create a program, be sure to select the **.NET Core cross-platform development** workload during installation.



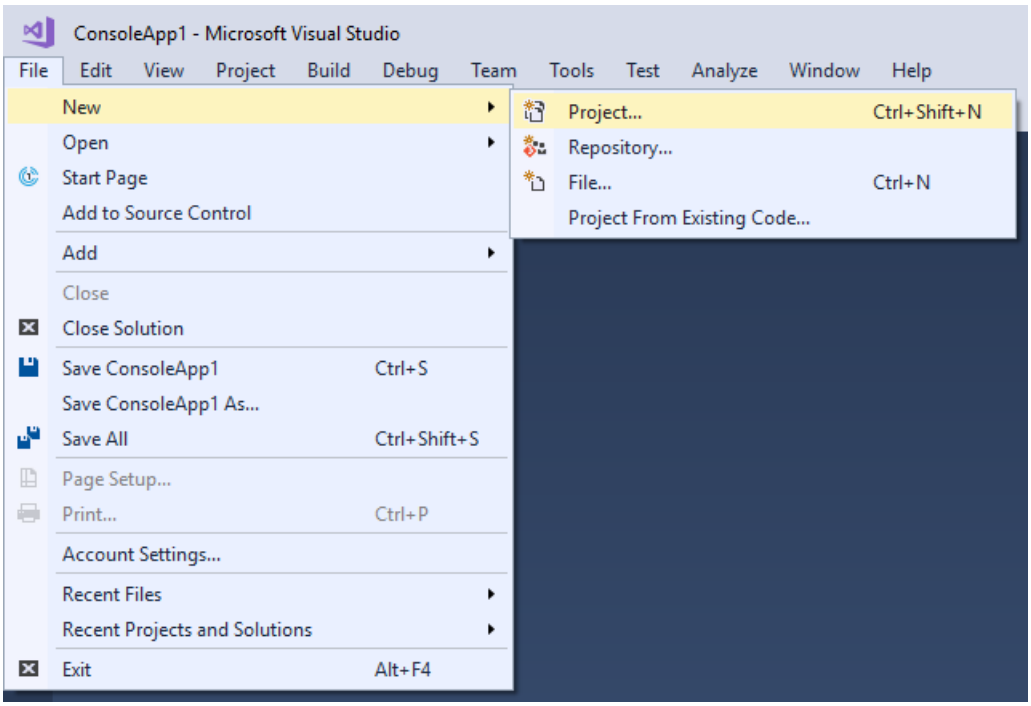
To get started, download Visual Studio and install it on your system. In the modular installer, you choose and install *workloads*, which are groups of features you need for the programming languages or platforms you want. To use the following steps to create a program, be sure to select the **.NET desktop development** workload during installation.

When you open Visual Studio for the first time, you can sign in by using your Microsoft account or your work or school account.

Create a program

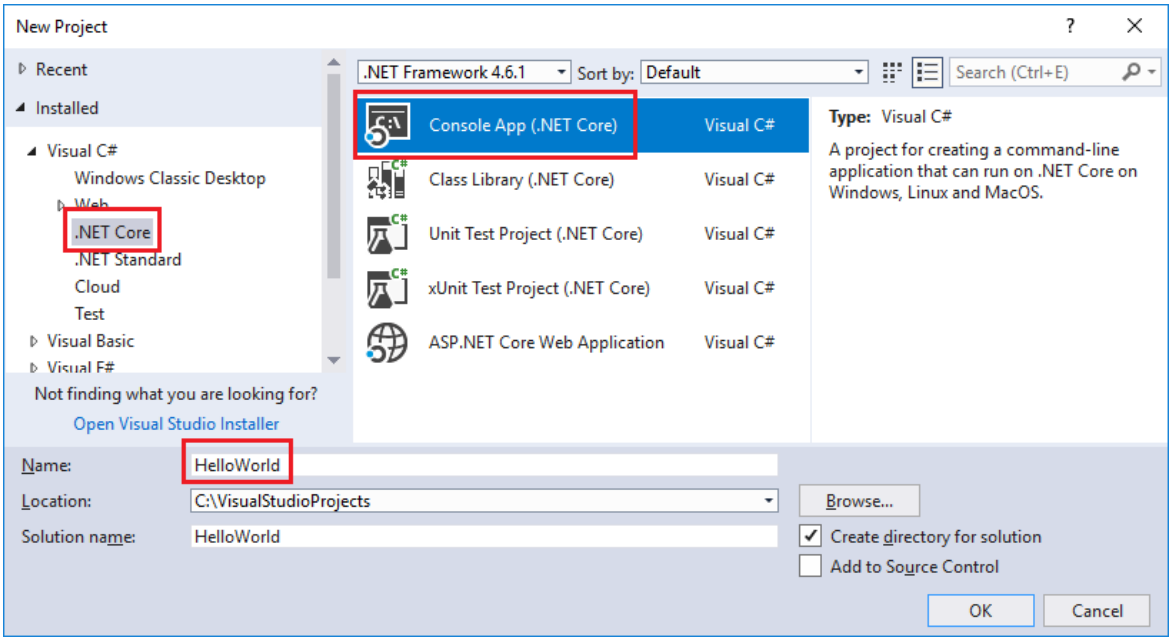
Dive in and create a simple program.

1. Open Visual Studio.
2. On the menu bar, choose **File** > **New** > **Project**.



The **New Project** dialog box shows several project *templates*. A template contains the basic files and settings needed for a given project type.

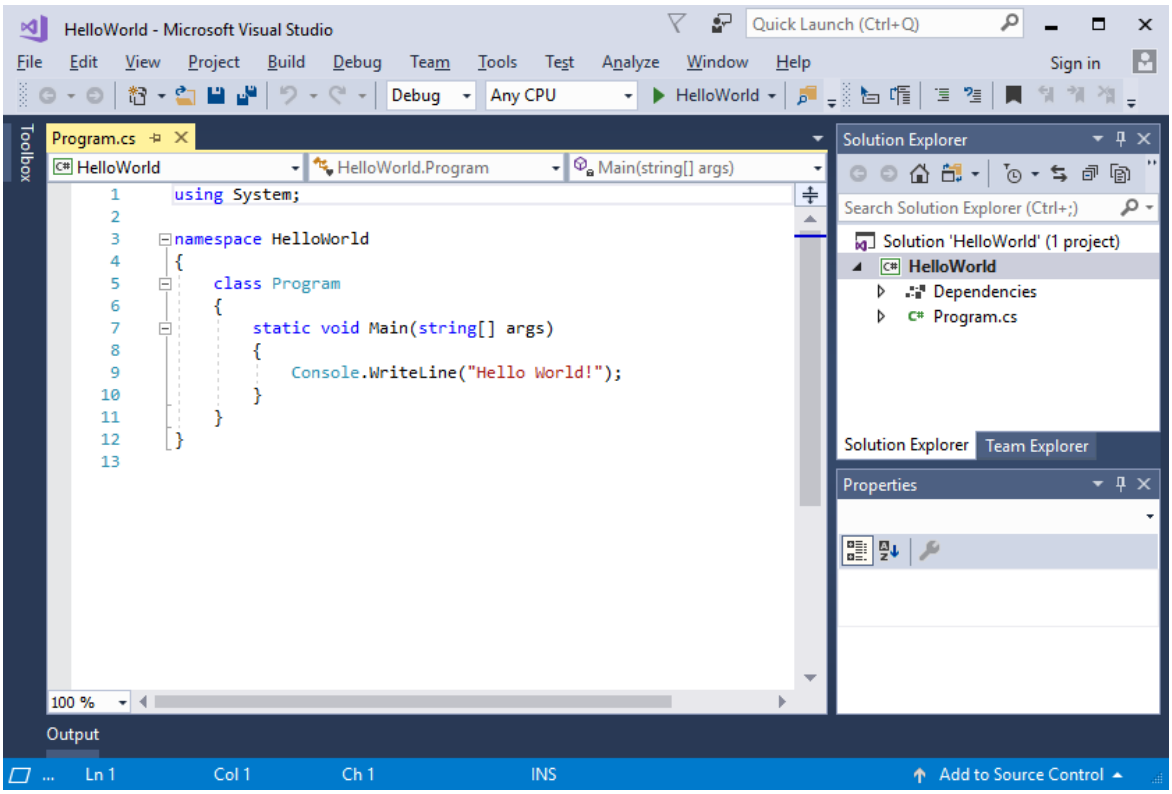
3. Choose the **.NET Core** template category under **Visual C#**, and then choose the **Console App (.NET Core)** template. In the **Name** text box, type **HelloWorld**, and then select the **OK** button.



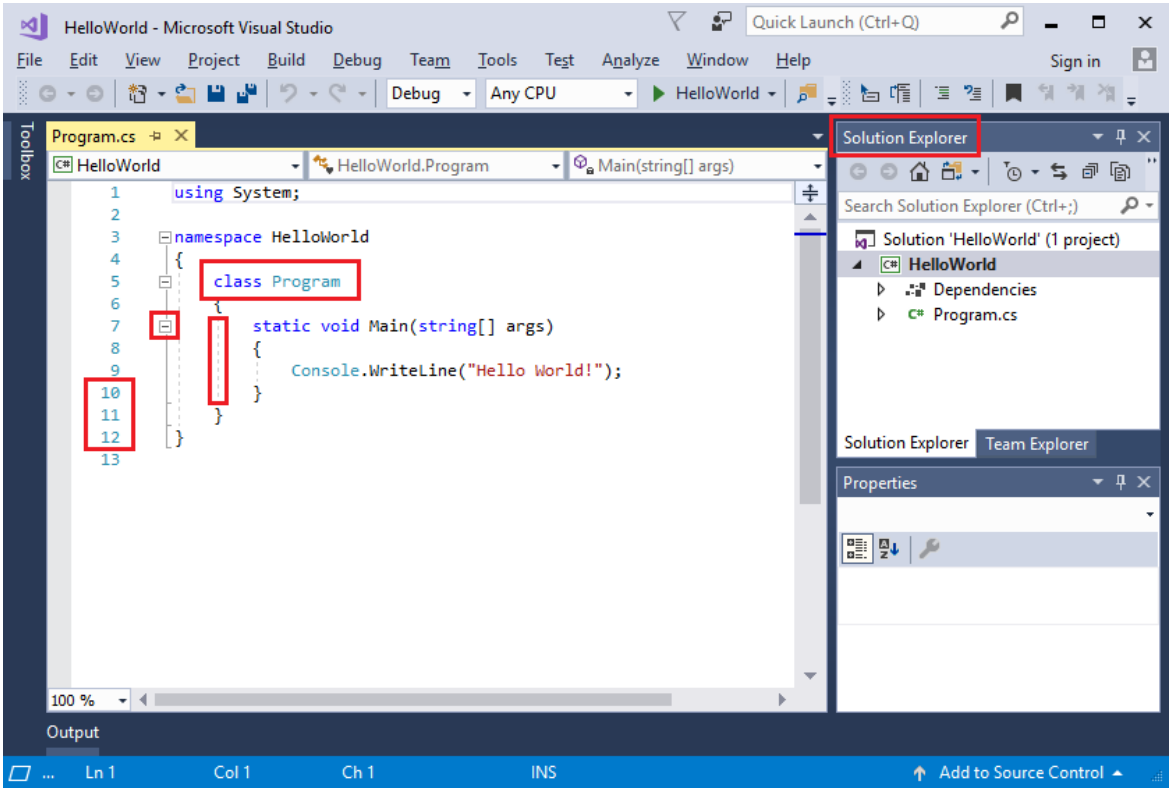
> **NOTE**
>
> If you don't see the **.NET Core** category, you need to install the **.NET Core cross-platform development** workload. To do this, choose the **Open Visual Studio Installer** link on the bottom left of the **New Project** dialog. After Visual Studio Installer opens, scroll down and select the **.NET Core cross-platform development** workload, and then select **Modify**.

Visual Studio creates the project. It's a simple "Hello World" application that calls the Console.WriteLine() method to display the literal string "Hello World!" in the console (program output) window.

Shortly, you should see something like the following screen:

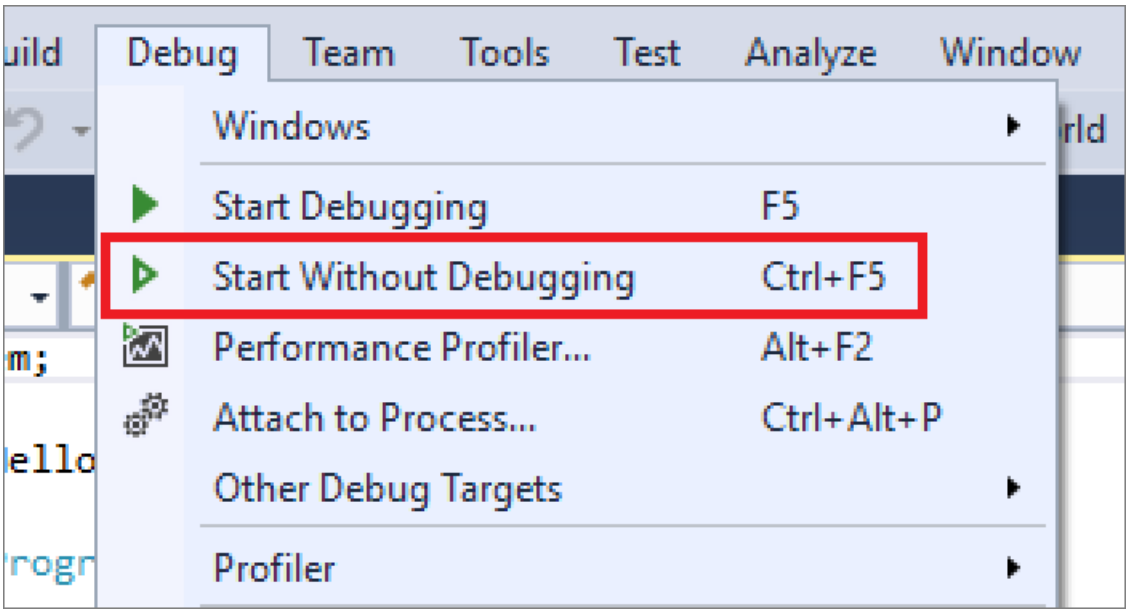


The C# code for your application shows in the editor window, which takes up most of the space. Notice that the text is automatically colorized to indicate different parts of the code, such as keywords and types. In addition, small, vertical dashed lines in the code indicate which braces match one another, and line numbers help you locate code later. You can choose the small, boxed minus signs to collapse or expand blocks of code. This code outlining feature lets you hide code you don't need, helping to minimize onscreen clutter. The project files are listed on the right side in a window called **Solution Explorer**.

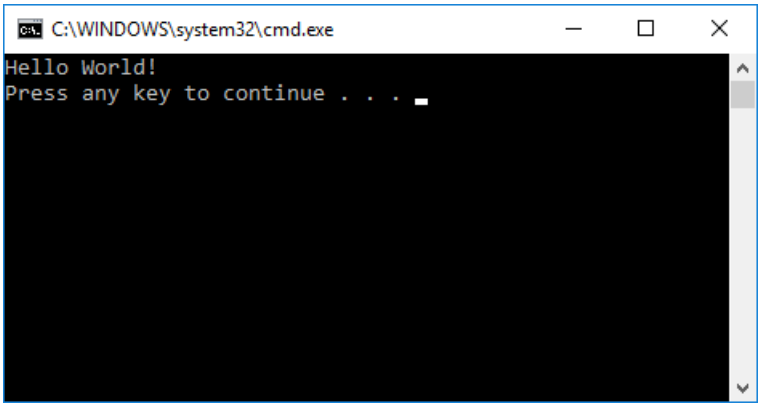


There are other menus and tool windows available, but let's move on for now.

4. Now, start the app. You can do this by choosing **Start Without Debugging** from the **Debug** menu on the menu bar. You can also press **Ctrl+F5**.



Visual Studio builds the app, and a console window opens with the message **Hello World!**. You now have a running app!



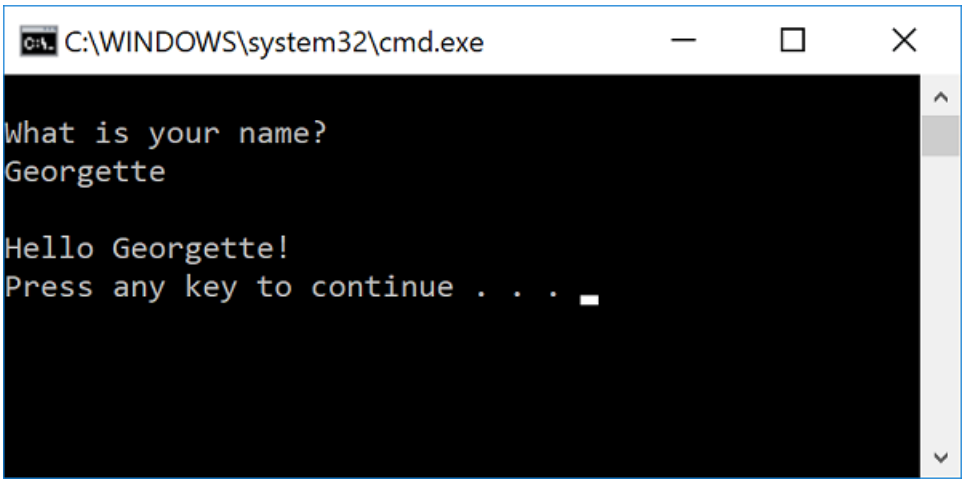
5. To close the console window, press any key on your keyboard.
6. Let's add some more code to the app. Add the following C# code before the line that says `Console.WriteLine("Hello World!");`:

   ```
   Console.WriteLine("\nWhat is your name?");
   var name = Console.ReadLine();
   ```

   This code displays **What is your name?** in the console window, and then waits until the user enters some text followed by the **Enter** key.
7. Change the line that says `Console.WriteLine("Hello World!");` to the following code:

   ```
   Console.WriteLine($"\nHello {name}!");
   ```
8. Run the app again by selecting **Debug** > **Start Without Debugging** or by pressing **Ctrl+F5**.

   Visual Studio rebuilds the app, and a console window opens and prompts you for your name.
9. Enter your name in the console window and press **Enter**.

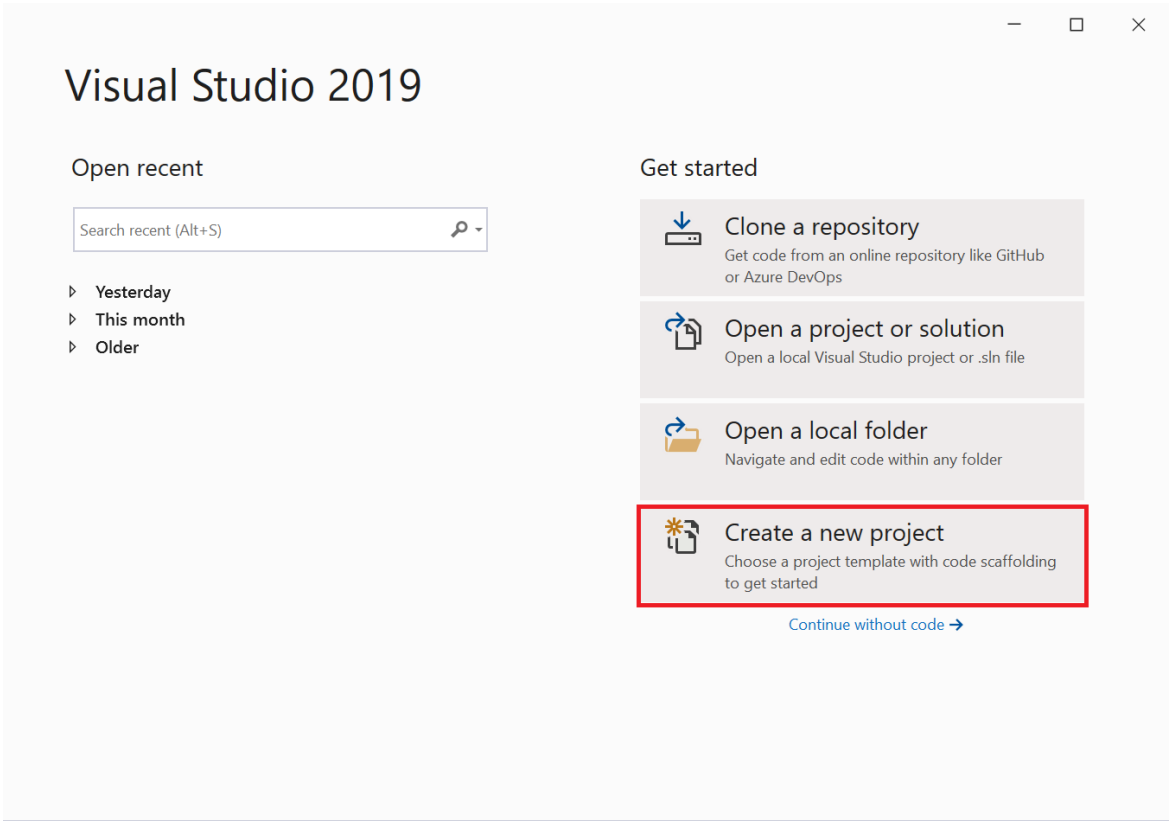


10. Press any key to close the console window and stop the running program.

1. Open Visual Studio.

   The start window appears with options for cloning a repo, opening a recent project, or creating a new project.

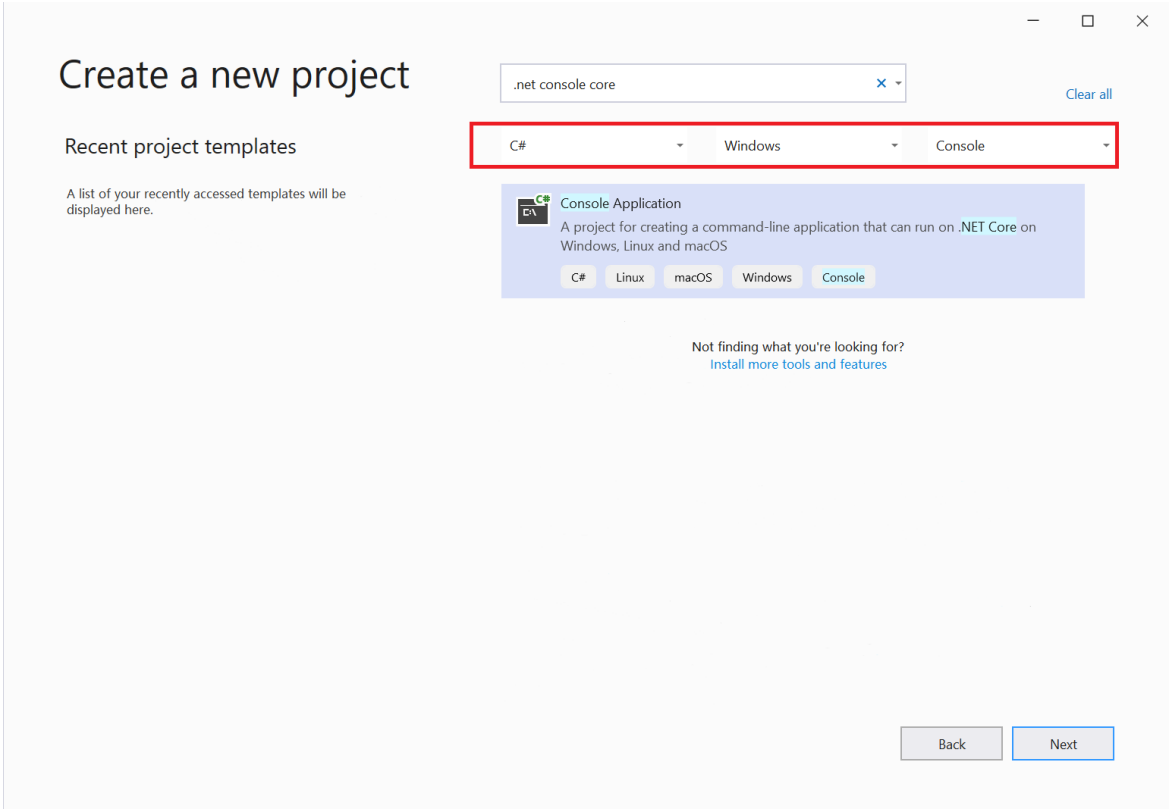
2. Choose **Create a new project**.



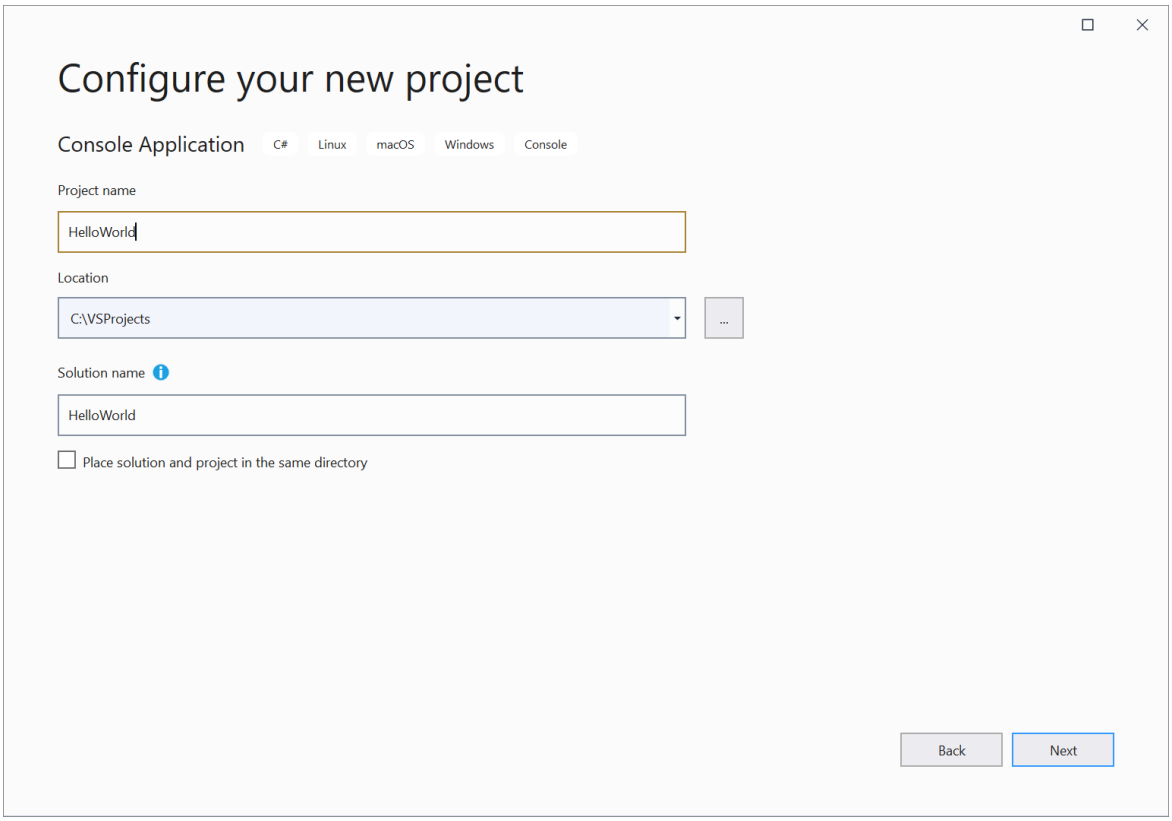
   The **Create a new project** window opens and shows several project *templates*. A template contains the basic files and settings required for a given project type.

3. To find the template we want, type or enter **.net core console** in the search box. The list of available templates is automatically filtered based on the keywords you entered. You can further filter the template results by choosing **C#** from the **All language** drop-down list, **Windows** from the **All platforms** list, and **Console** from the **All project types** list.

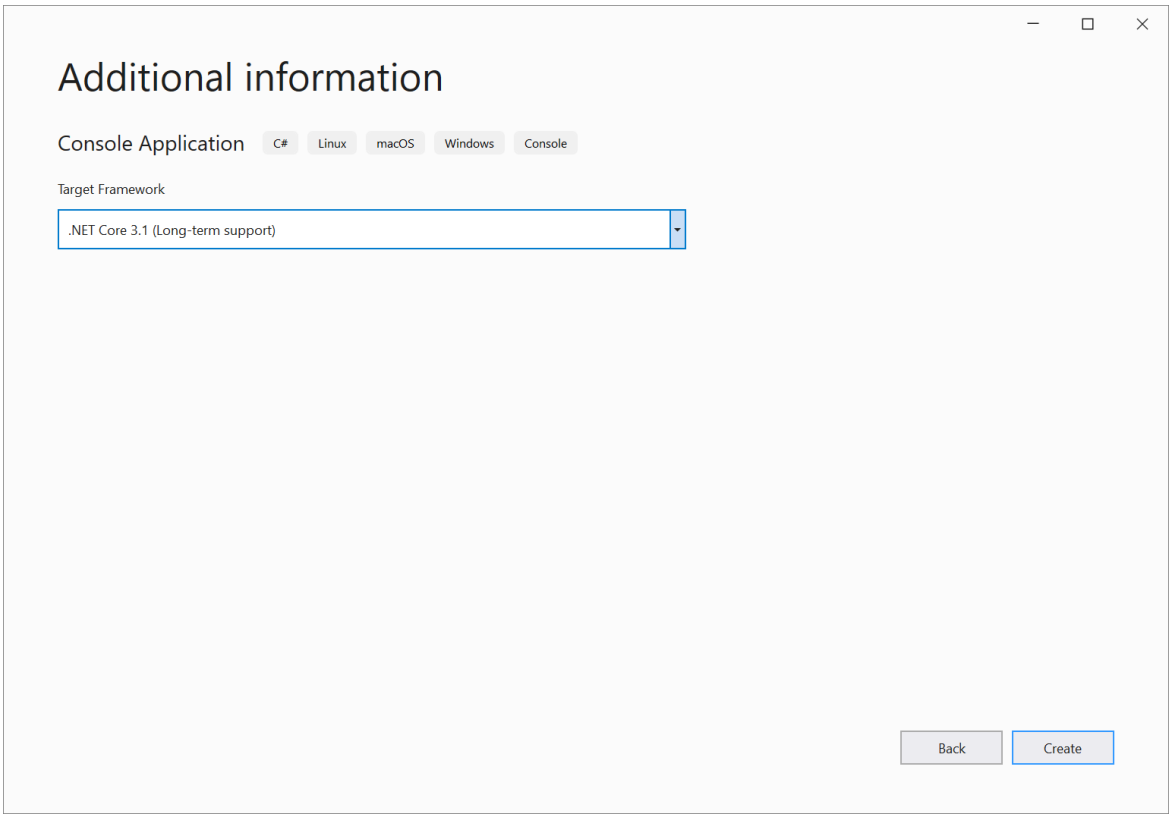
   Select the **Console Application** template, and then click **Next**.



4. In the **Configure your new project** window, enter **HelloWorld** in the **Project name** box, optionally change the directory location for your project files (the default locale is `C:\Users\<name>\source\repos`), and then click **Next**.

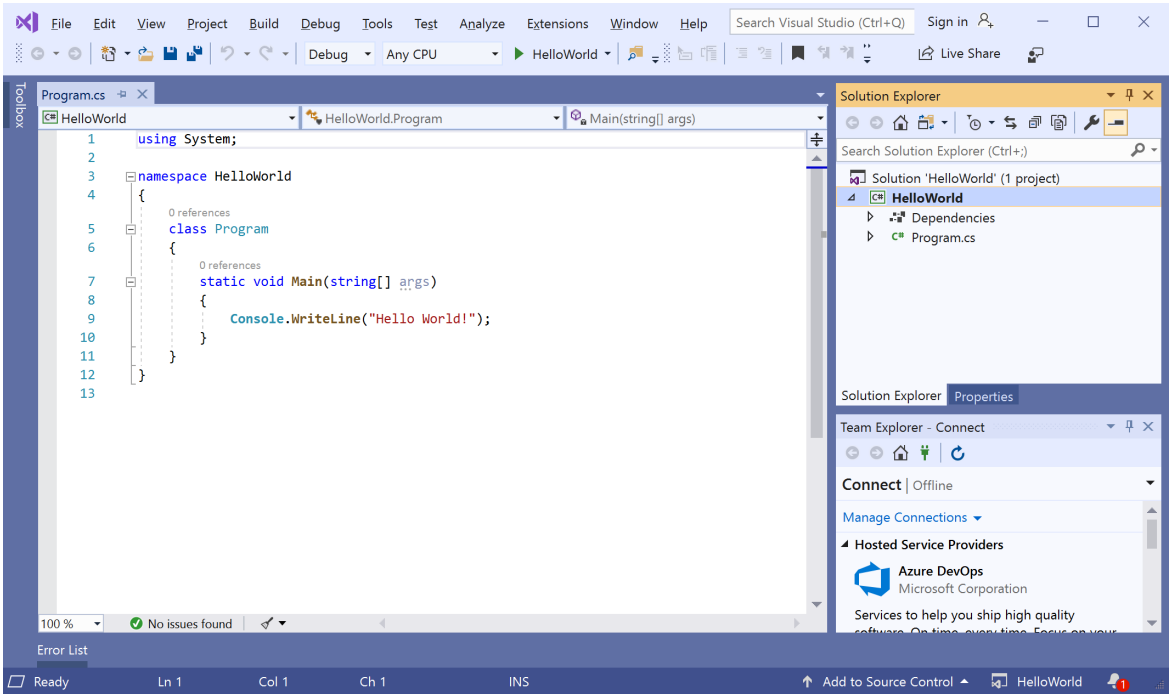


5. In the **Additional information** window, verify that **.NET Core 3.1** appears in the **Target Framework** drop-down menu, and then click **Create**.

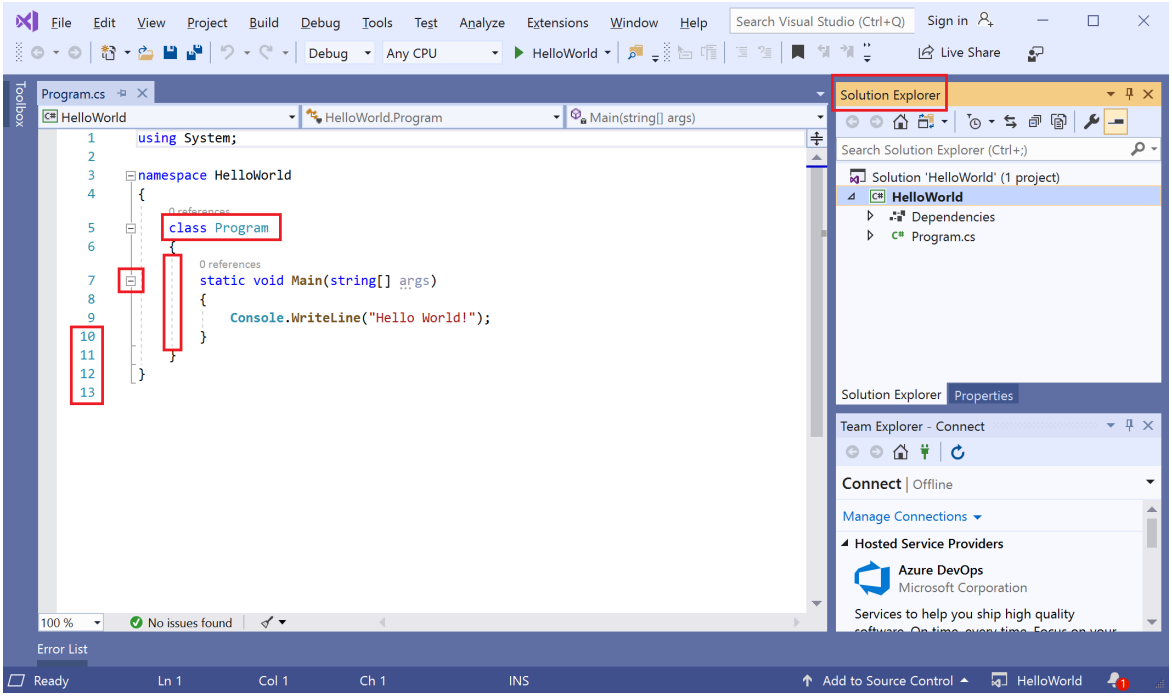


Visual Studio creates the project. It's a simple "Hello World" application that calls the Console.WriteLine() method to display the literal string "Hello World!" in the console (program output) window.

Shortly, you should see something like the following screen:

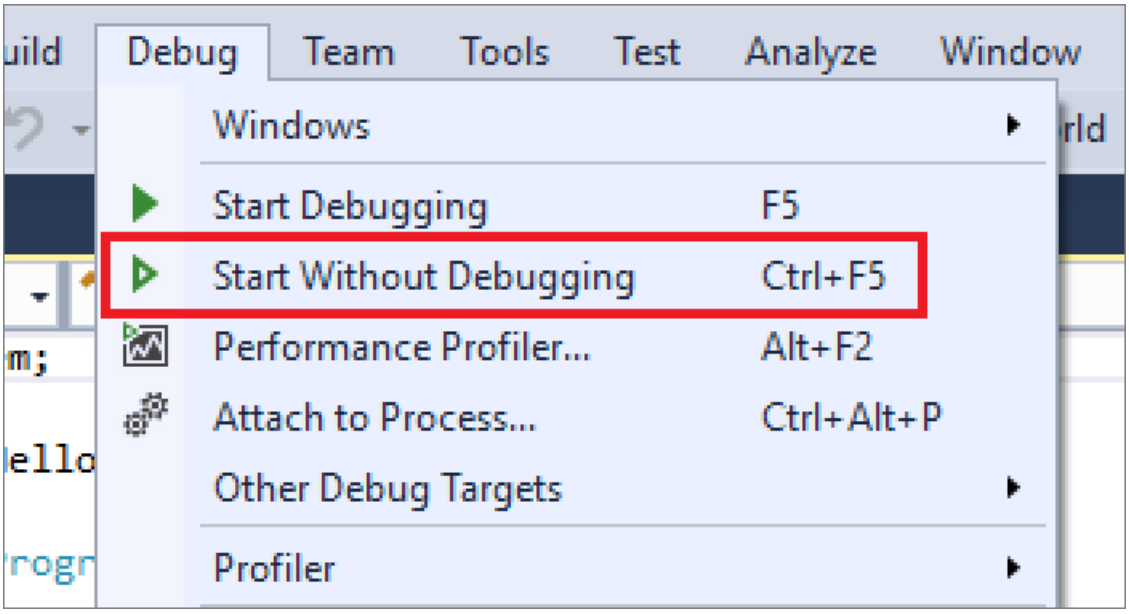


The C# code for your application shows in the editor window, which takes up most of the space. Notice that the text is automatically colorized to indicate different parts of the code, such as keywords and types. In addition, small, vertical dashed lines in the code indicate which braces match one another, and line numbers help you locate code later. You can choose the small, boxed minus signs to collapse or expand blocks of code. This code outlining feature lets you hide code you don't need, helping to minimize onscreen clutter. The project files are listed on the right side in a window called **Solution Explorer**.

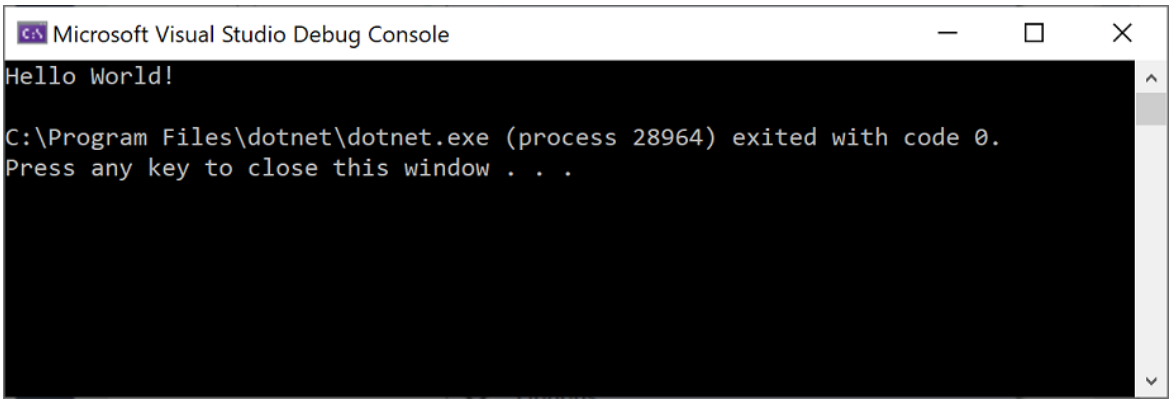


There are other menus and tool windows available, but let's move on for now.

6. Now, start the app. You can do this by choosing **Start Without Debugging** from the **Debug** menu on the menu bar. You can also press **Ctrl+F5**.



Visual Studio builds the app, and a console window opens with the message **Hello World!**. You now have a running app!



7. To close the console window, press any key on your keyboard.

8. Let's add some more code to the app. Add the following C# code before the line that says `Console.WriteLine("Hello World!");`:

   ```
   Console.WriteLine("\nWhat is your name?");
   var name = Console.ReadLine();
   ```

   This code displays **What is your name?** in the console window, and then waits until the user enters some text followed by the **Enter** key.

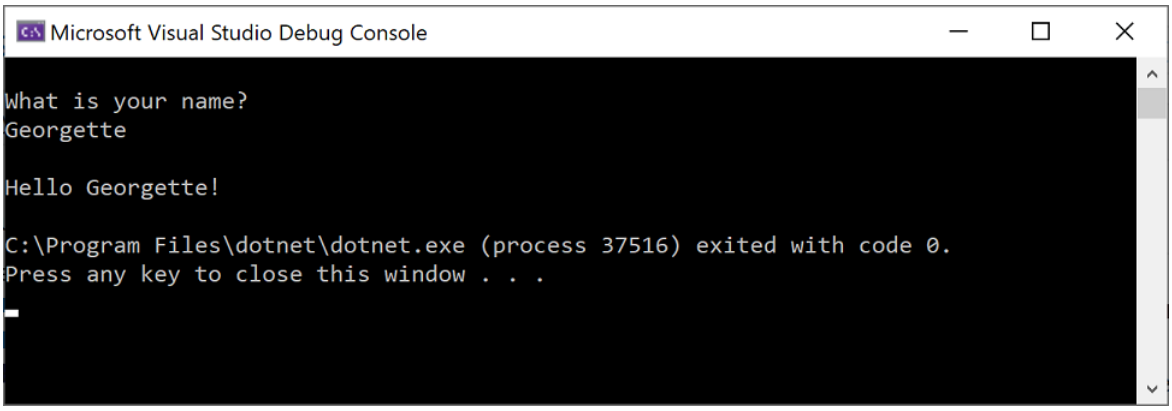
9. Change the line that says `Console.WriteLine("Hello World!");` to the following code:

   ```
   Console.WriteLine($"\nHello {name}!");
   ```

10. Run the app again by selecting **Debug** > **Start Without Debugging** or by pressing **Ctrl**+**F5**.

    Visual Studio rebuilds the app, and a console window opens and prompts you for your name.

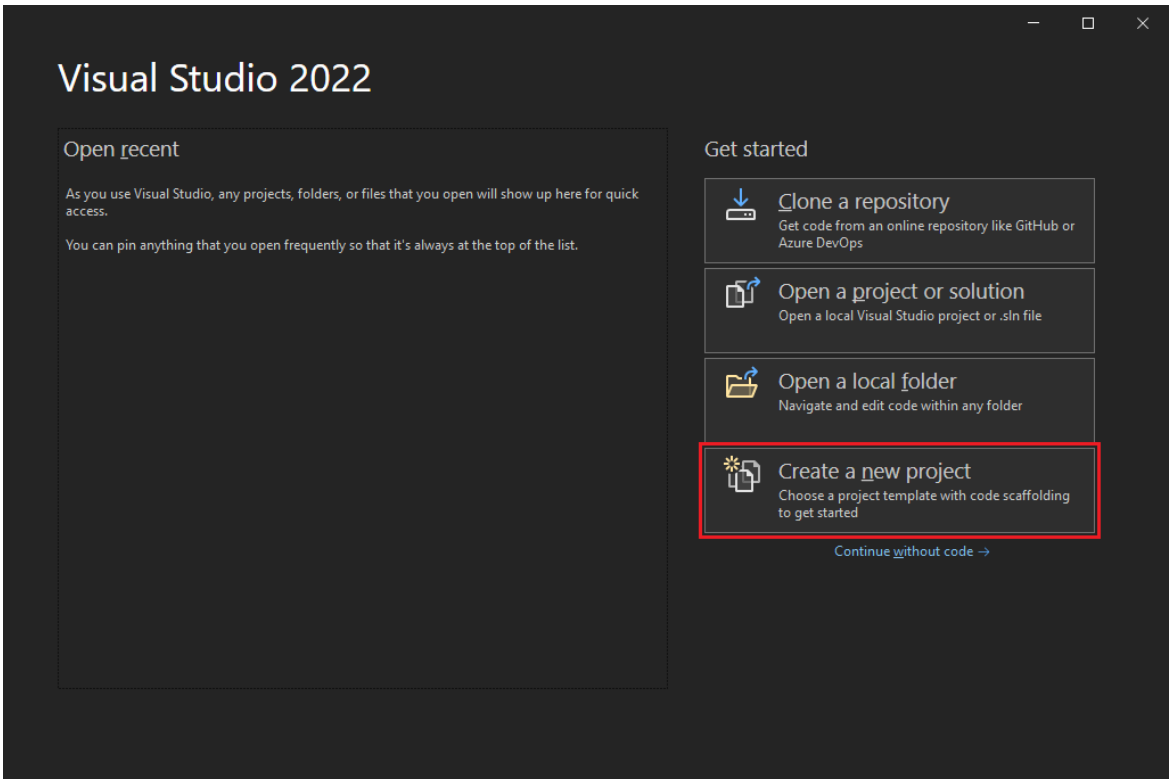
11. Enter your name in the console window and press **Enter**.

12. Press any key to close the console window and stop the running program.

1. Start Visual Studio. The start window appears with options for cloning a repo, opening a recent project, or creating a new project.

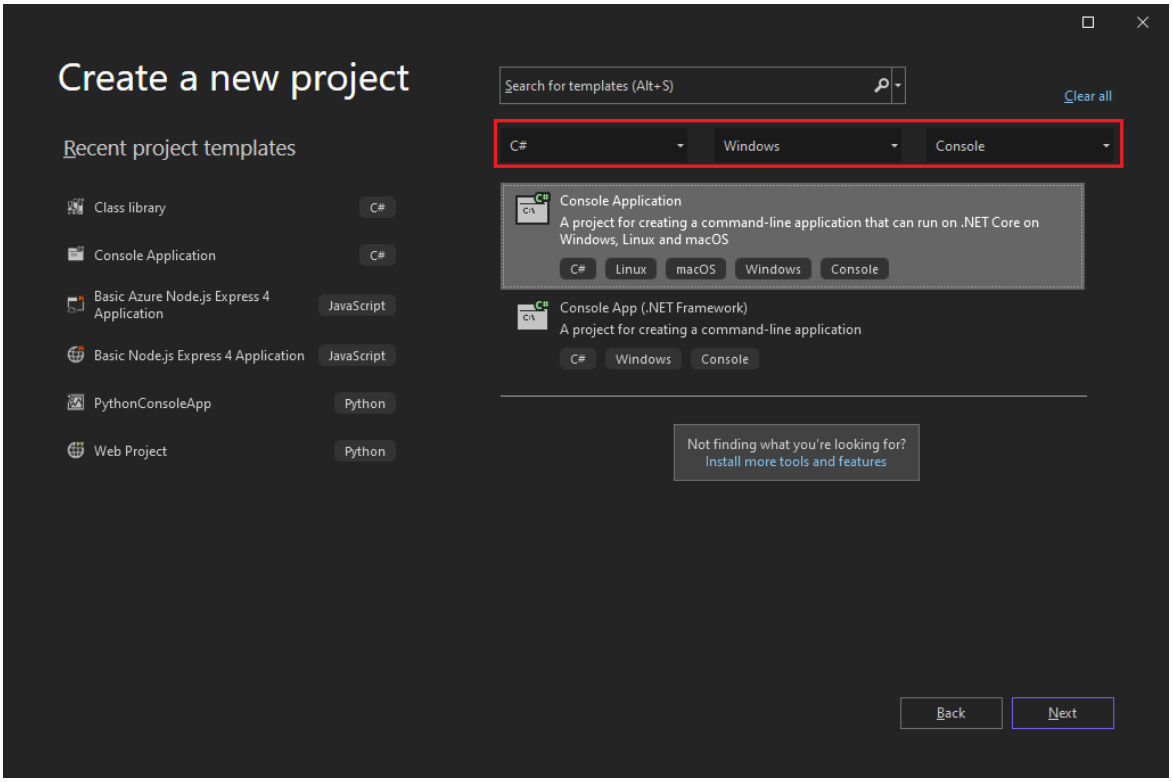
2. Choose **Create a new project**.



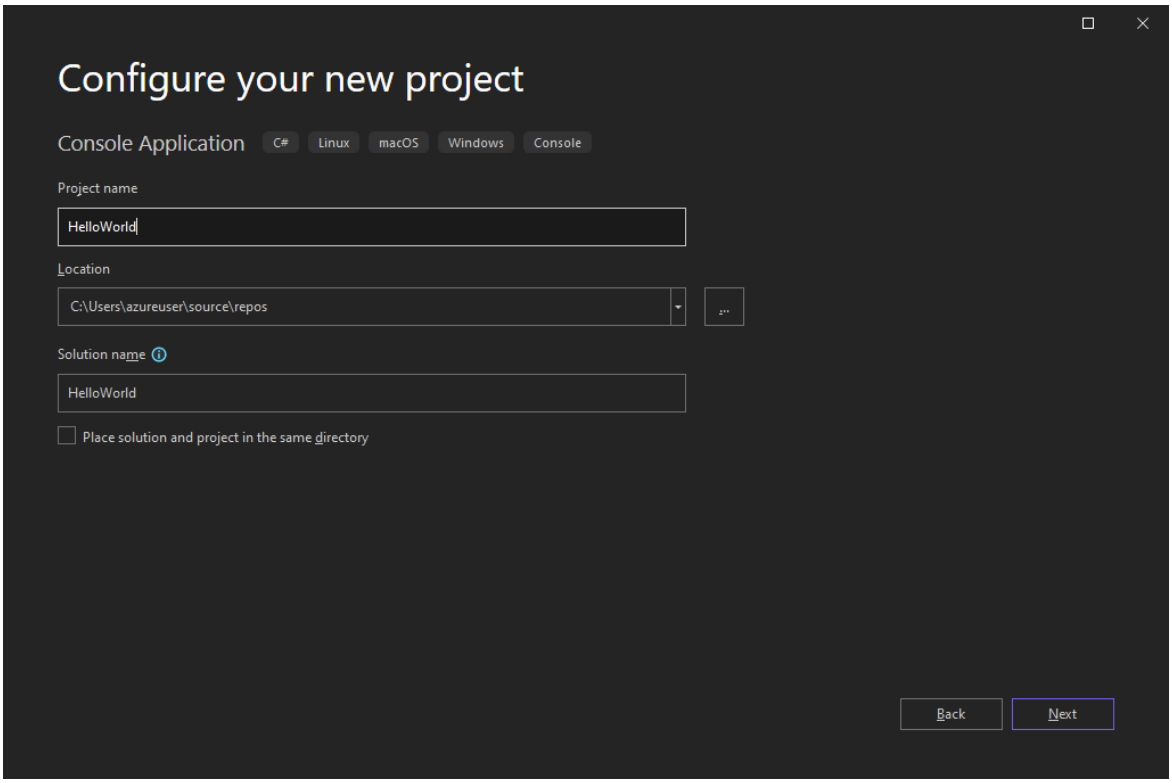
The **Create a new project** window opens and shows several project *templates*. A template contains the basic files and settings required for a given project type.

3. To find a template, you can type or enter keywords in the search box. The list of available templates filters based on the keywords you enter. You can further filter the template results by choosing **C#** from the **All languages** dropdown list, **Windows** from the **All platforms** list, and **Console** from the **All project types** list.

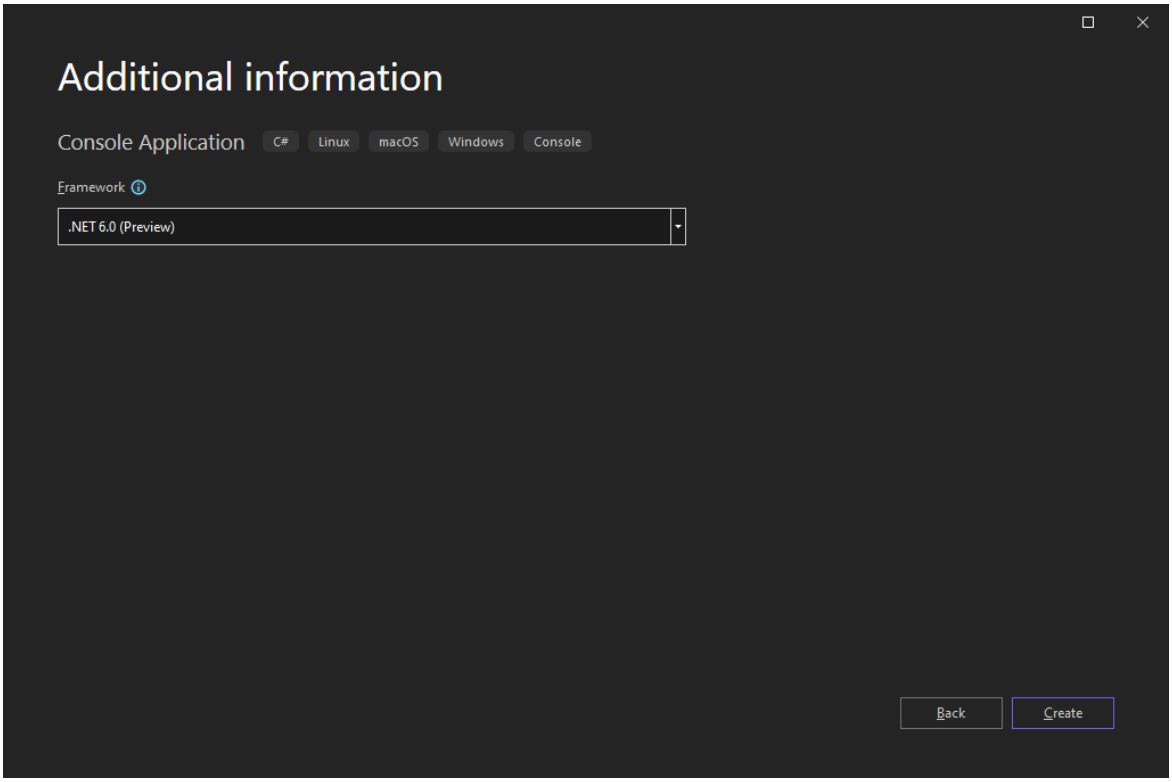
   Select the **Console Application** template, and then select **Next**.



4. In the **Configure your new project** window, enter **HelloWorld** in the **Project name** box. Optionally, change the project directory location from the default location of *C:\Users\<name>\source\repos*, and then select **Next**.

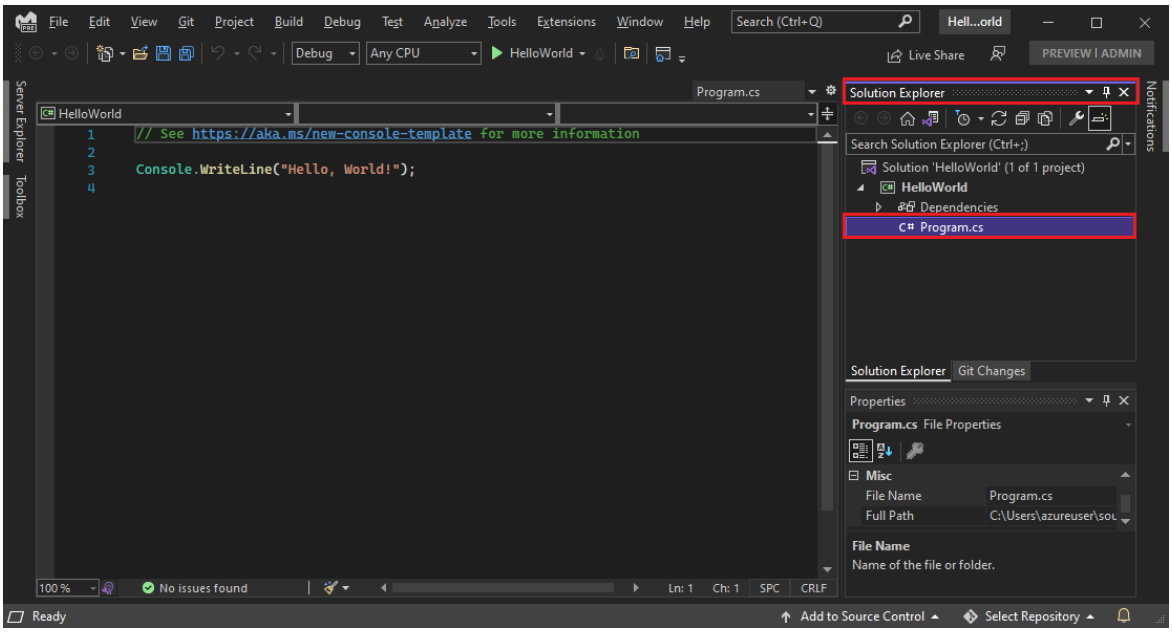


5. In the **Additional information** window, verify that **.NET 6.0** appears in the **Target Framework** drop-down menu, and then select **Create**.



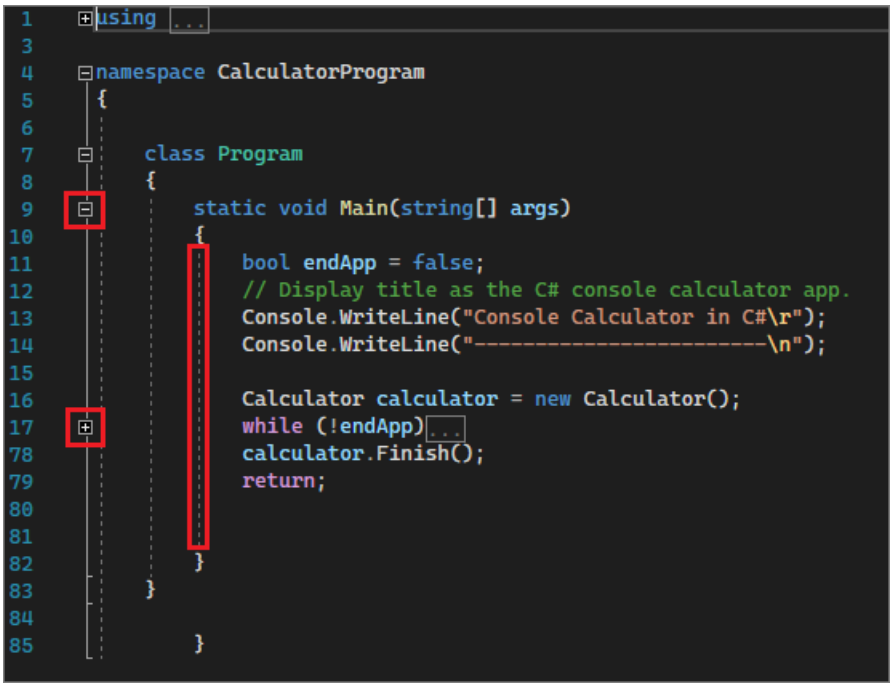
Visual Studio creates the project. The program is a simple "Hello World" application that calls the Console.WriteLine() method to display the string **Hello, World!** in a console window.

The project files appear on the right side of the Visual Studio IDE, in a window called the **Solution Explorer**. In the **Solution Explorer** window, select the **Program.cs** file. The C# code for your app opens in the central editor window, which takes up most of the space.



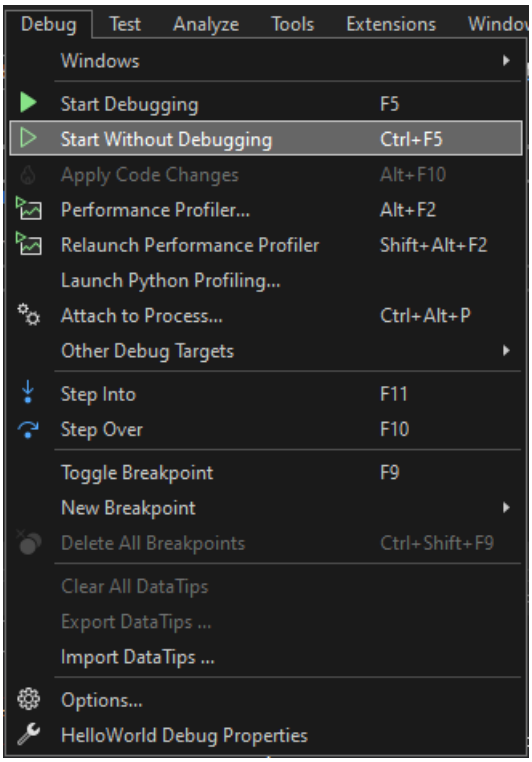
The code is automatically colorized to indicate different parts, such as keywords and types. Line numbers help you locate code.

Small, vertical dashed lines in the code indicate which braces match one another. You can also choose small, boxed minus or plus signs to collapse or expand blocks of code. This code outlining feature lets you hide code you don't need to see, helping to minimize onscreen clutter.

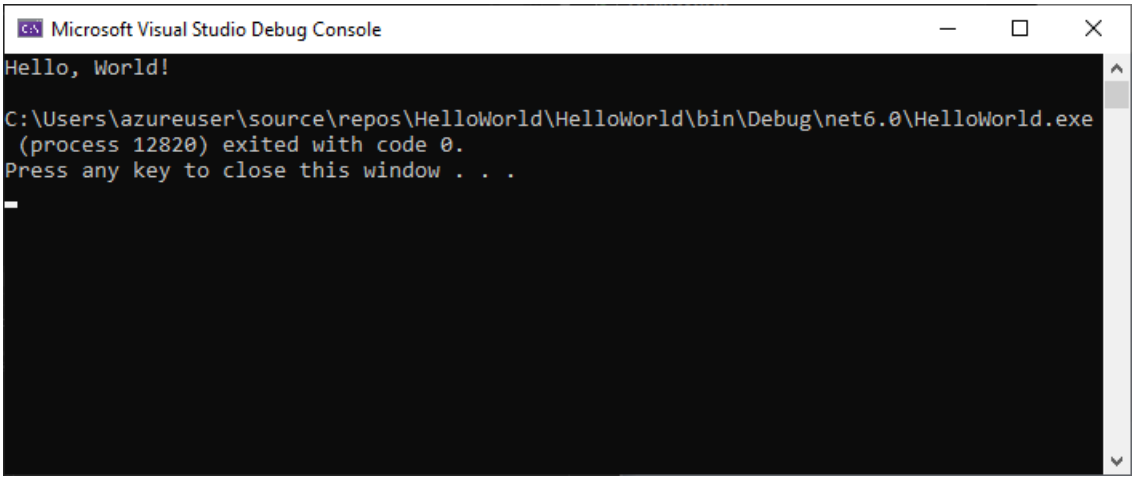


Many other menus and tool windows are available.

6. Start the app by choosing **Debug** > **Start Without Debugging** from the Visual Studio top menu. You can also press **Ctrl+F5**.



Visual Studio builds the app, and a console window opens with the message **Hello, World!**. You now have a running app!



7. To close the console window, press any key.
8. Let's add some more code to the app. Add the following C# code before the line that says `Console.WriteLine("Hello World!");`:

   ```
   Console.WriteLine("\nWhat is your name?");
   var name = Console.ReadLine();
   ```

   This code displays **What is your name?** in the console window, and then waits until the user enters some text.

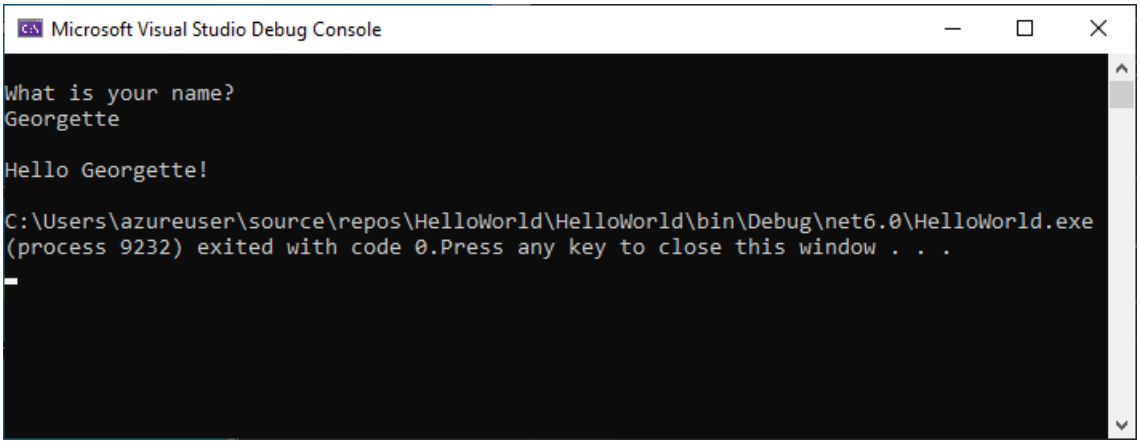
9. Change the line that says `Console.WriteLine("Hello World!");` to the following line:

   ```
   Console.WriteLine($"\nHello {name}!");
   ```

10. Run the app again by selecting **Debug** > **Start Without Debugging** or pressing **Ctrl**+**F5**.

    Visual Studio rebuilds the app, and a console window opens and prompts you for your name.

11. Type your name in the console window and press **Enter**.



12. Press any key to close the console window and stop the running program.

Use refactoring and IntelliSense

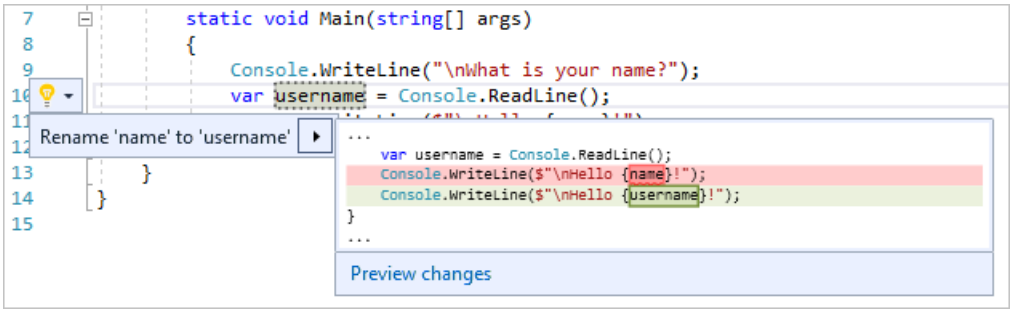
Let's look at a couple of the ways that refactoring and IntelliSense can help you code more efficiently.

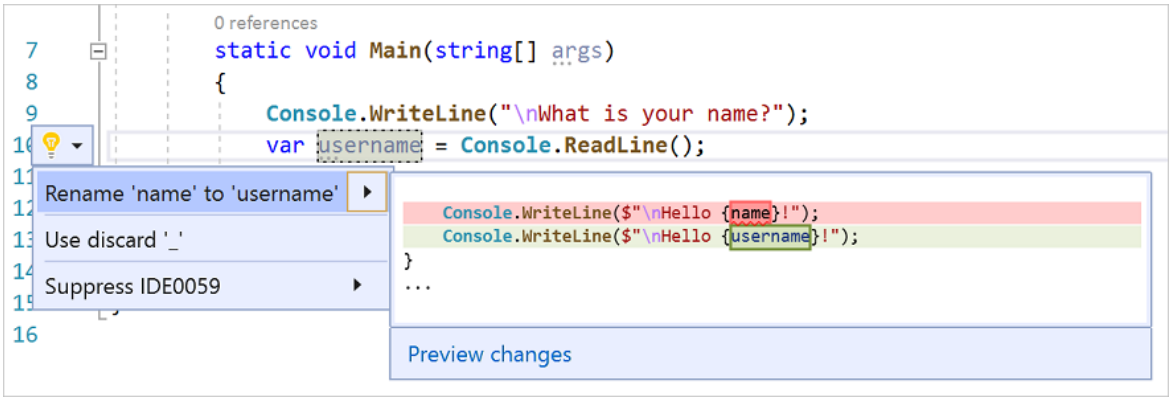
First, rename the `name` variable:

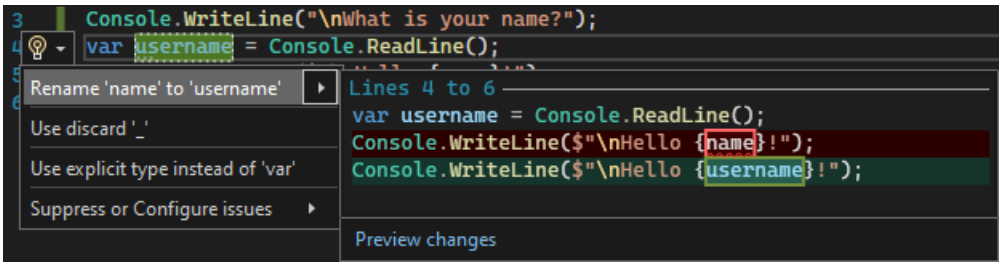
1. Double-click the `name` variable, and type the new name for the variable, *username*.

   A box appears around the variable, and a light bulb appears in the margin.

2. Select the light bulb icon to show the available Quick Actions. Select **Rename 'name' to 'username'**.





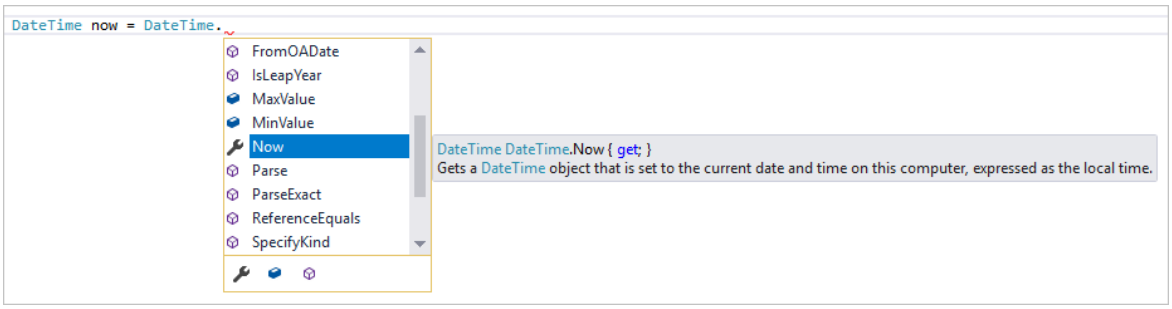


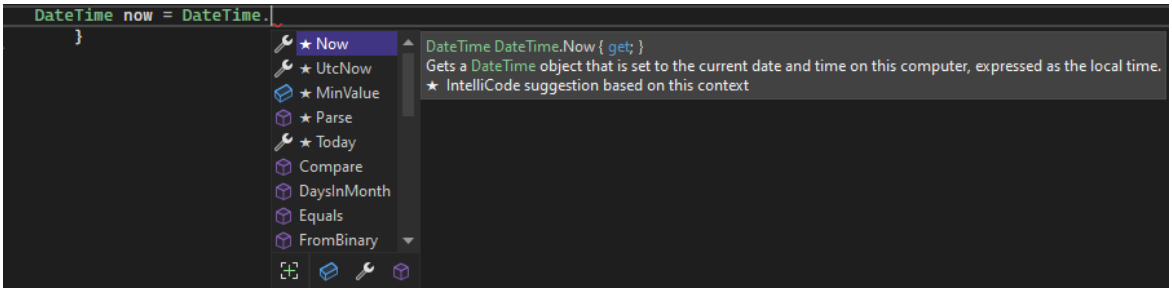
The variable is renamed across the project, which in our case is only two places.

```
namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("\nWhat is your name?");
            var name = Console.ReadLine();
            Console.WriteLine($"\nHello {name}!");
        }
    }
}
```

3. Now take a look at IntelliSense. Below the line that says `Console.WriteLine($"\nHello {username}!");`, type `DateTime now = DateTime.`.

   A box displays the members of the DateTime class. The description of the currently selected member also displays in a separate box.





4. Select the member named **Now**, which is a property of the class, by double-clicking it or pressing **Tab**. Complete the line of code by adding a semicolon to the end of the line: `DateTime now = DateTime.Now;`.

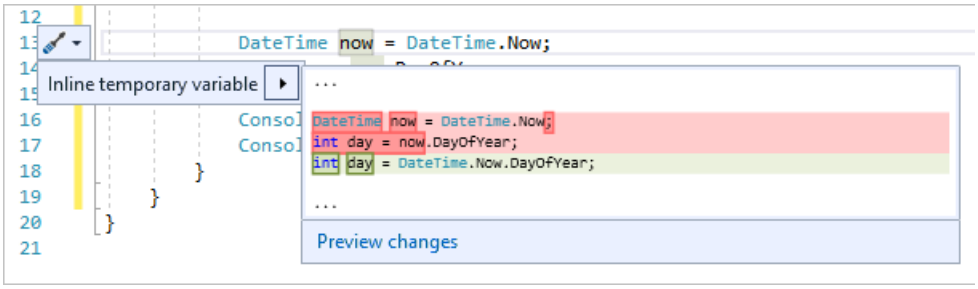
5. Below that line, enter the following lines of code:

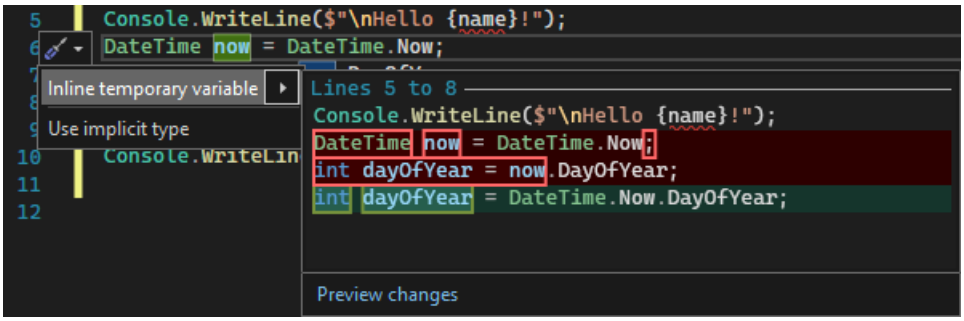
```
int dayOfYear = now.DayOfYear;

Console.Write("Day of year: ");
Console.WriteLine(dayOfYear);
```

> **TIP**
>
> Console.Write is different from Console.WriteLine in that it doesn't add a line terminator after it prints. That means that the next piece of text that's sent to the output will print on the same line. You can hover over each of these methods in your code to see their descriptions.

6. Next, use refactoring again to make the code a little more concise. Select the variable `now` in the line `DateTime now = DateTime.Now;`. A screwdriver icon appears in the margin on that line.

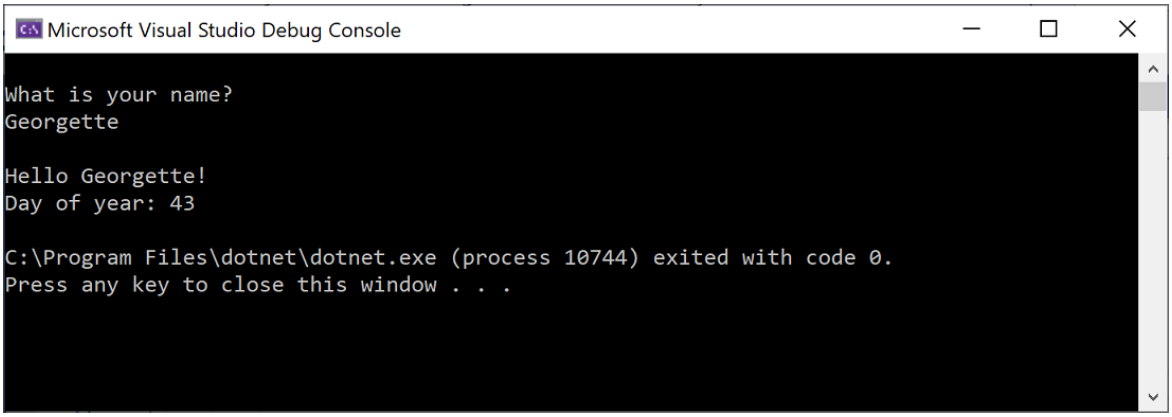
7. Select the screwdriver icon to see available suggestions from Visual Studio. This case shows the Inline temporary variable refactoring to remove a line of code without changing the overall code behavior.

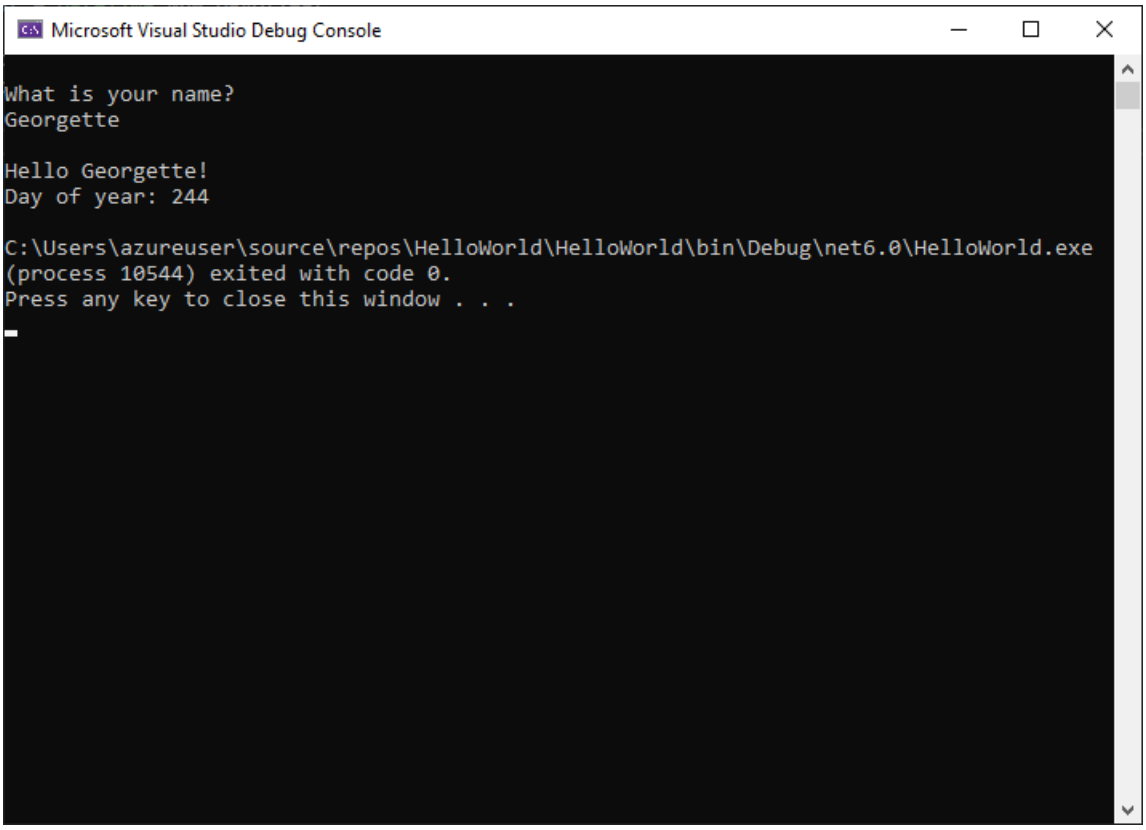




8. Select **Inline temporary variable** to refactor the code.

9. Run the program again by pressing **Ctrl+F5**. The output looks something like this:





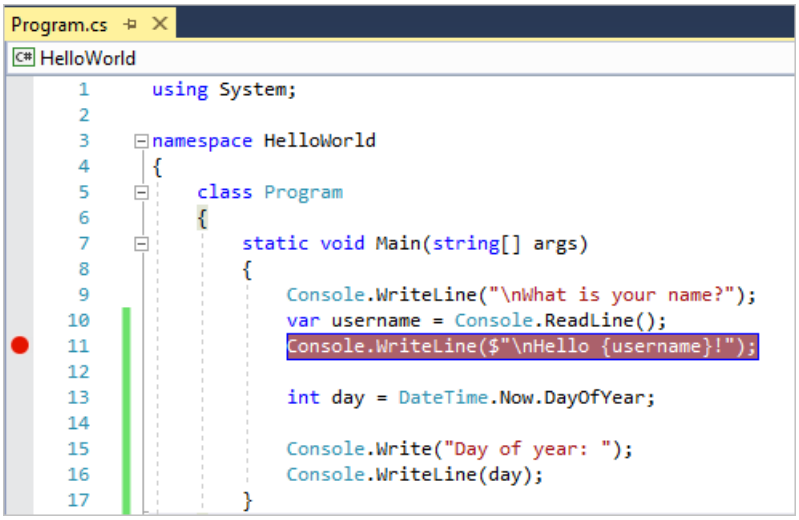
Debug code

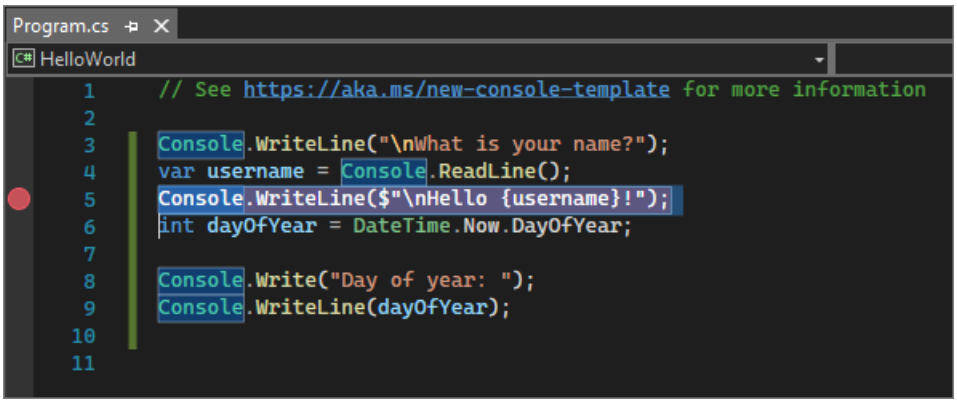
When you write code, you should run it and test it for bugs. Visual Studio's debugging system lets you step through code one statement at a time and inspect variables as you go. You can set *breakpoints* that stop execution of the code at a particular line, and observe how the variable value changes as the code runs.

Set a breakpoint to see the value of the `username` variable while the program is running.

1. Set a breakpoint on the line of code that says `Console.WriteLine($"\nHello {username}!");` by clicking in the far-left margin, or gutter, next to the line. You can also select the line of code and then press **F9**.

   A red circle appears in the gutter, and the line is highlighted.



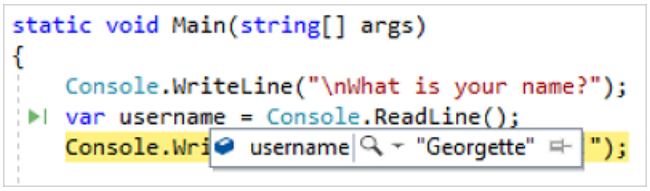


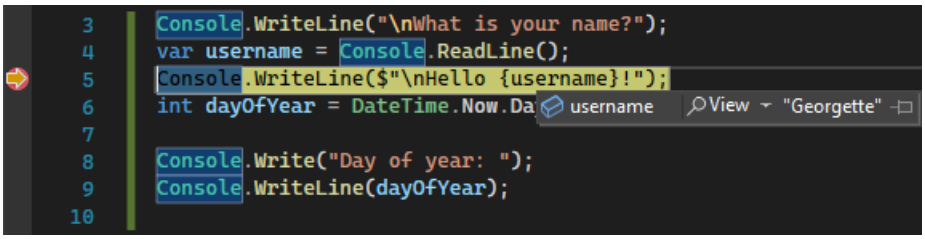
2. Start debugging by selecting **Debug** > **Start Debugging** or pressing **F5**.

3. When the console window appears and asks for your name, enter your name.

   The focus returns to the Visual Studio code editor, and the line of code with the breakpoint is highlighted in yellow. The yellow highlight means that this line of code will execute next. The breakpoint makes the app pause execution at this line.

4. Hover your mouse over the `username` variable to see its value. You can also right-click on `username` and select **Add Watch** to add the variable to the **Watch** window, where you can also see its value.





5. Press **F5** again to finish running the app.

For more information about debugging in Visual Studio, see the Debugger feature tour.

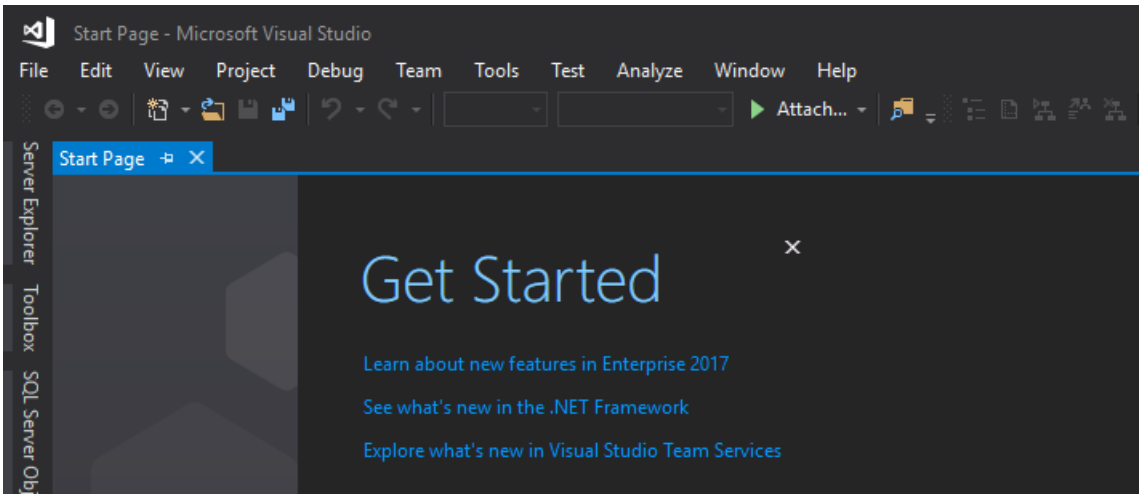
Customize Visual Studio

You can personalize the Visual Studio user interface, including changing the default color theme. To change the color theme:

1. On the menu bar, choose **Tools** > **Options** to open the **Options** dialog.

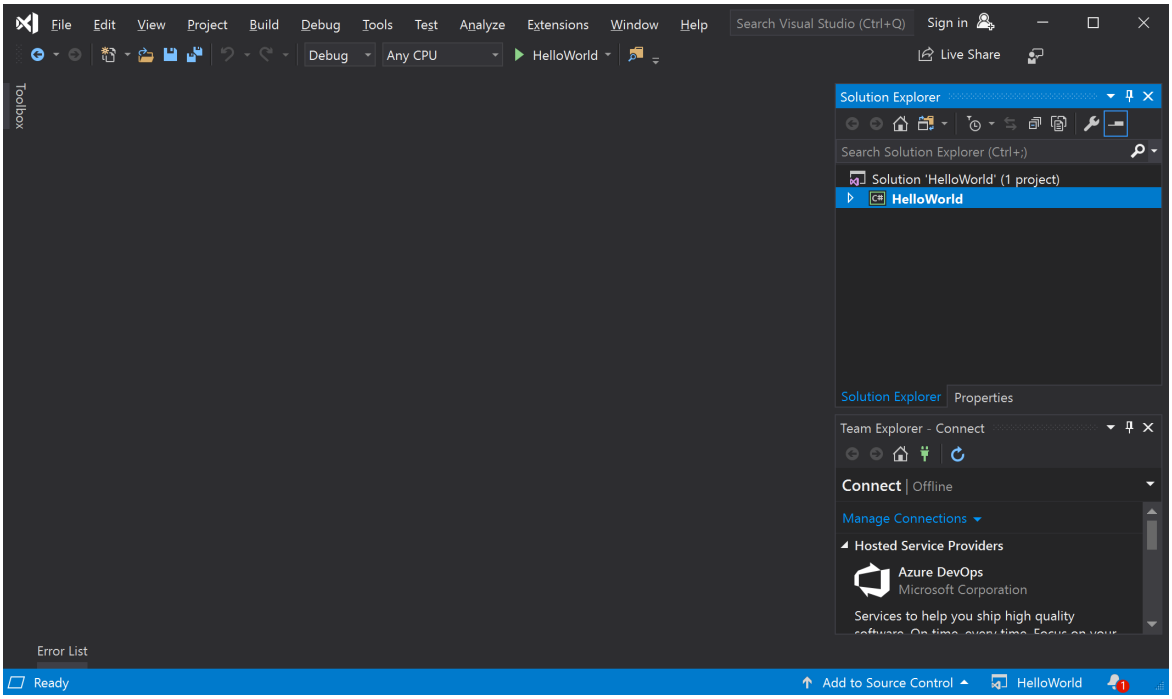
2. On the **Environment** > **General** options page, change the **Color theme** selection to **Dark**, and then choose **OK**.

   The color theme for the entire IDE changes to **Dark**.



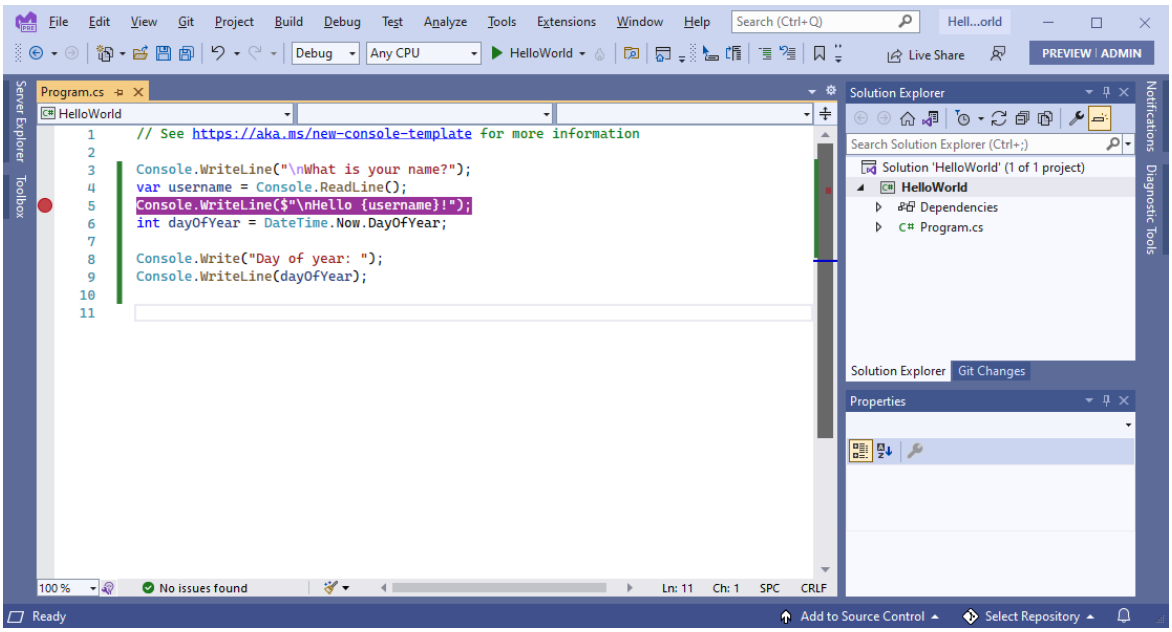
1. On the menu bar, choose **Tools** > **Options** to open the **Options** dialog.
2. On the **Environment** > **General** options page, change the **Color theme** selection to **Dark**, and then choose **OK**.

   The color theme for the entire IDE changes to **Dark**.



1. On the menu bar, choose **Tools** > **Options** to open the **Options** dialog.
2. On the **Environment** > **General** options page, change the **Color Theme** selection to **Blue** or **Light**, and then select **OK**.

   The color theme for the entire IDE changes accordingly. The following screenshot shows the Blue theme:



To learn about other ways you can personalize the IDE, see Personalize Visual Studio.

Select environment settings

You can configure Visual Studio to use environment settings tailored to C# developers:

1. On the menu bar, choose **Tools** > **Import and Export Settings**.
2. In the **Import and Export Settings Wizard**, select **Reset all settings**, and then select **Next**.
3. On the **Save Current Settings** page, choose whether to save your current settings before resetting. If you haven't customized any settings, select **No, just reset settings, overwriting my current settings**. Then select **Next**.
4. On the **Choose a Default Collection of Settings** page, choose **Visual C#**, and then select **Finish**.
5. On the **Reset Complete** page, select **Close**.

To learn about other ways you can personalize the IDE, see Personalize Visual Studio.

Next steps

Explore Visual Studio further by following along with one of these introductory articles:

Learn to use the code editor

Learn about projects and solutions

See also

- Discover more Visual Studio features.
- Visit visualstudio.microsoft.com.
- Read the Visual Studio blog.

Learn to use the code editor with C#

10/28/2021 • 11 minutes to read • Edit Online

In this 10-minute introduction to the code editor in Visual Studio, we'll add code to a file to look at some of the ways that Visual Studio makes writing, navigating, and understanding C# code easier.

> **TIP**
> If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

This article assumes you're already familiar with C#. If you aren't, we suggest you look at a tutorial such as Get started with C# and ASP.NET Core in Visual Studio first.

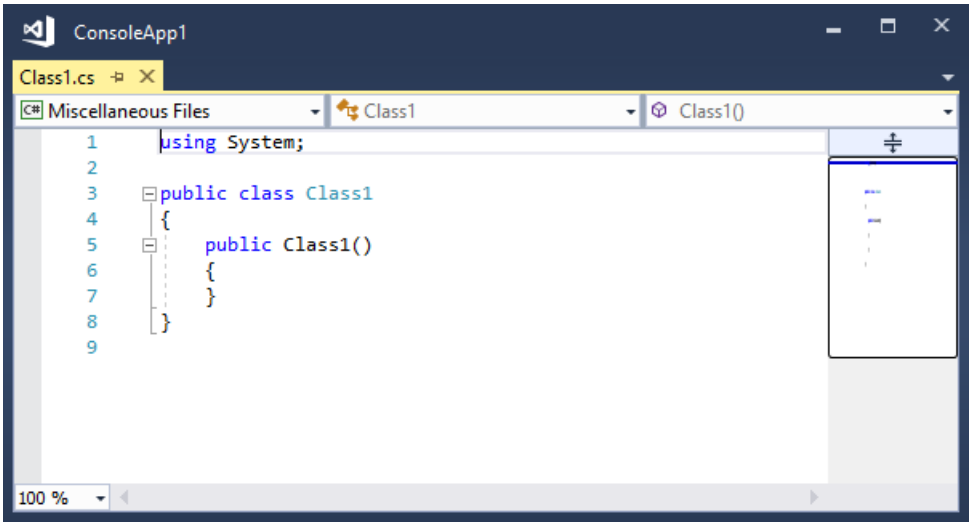
> **TIP**
> To follow along with this article, make sure you have the C# settings selected for Visual Studio. For information about selecting settings for the integrated development environment (IDE), see Select environment settings.

Create a new code file

Start by creating a new file and adding some code to it.

1. Open Visual Studio.
2. From the **File** menu on the menu bar, choose **New** > **File**, or press **Ctrl**+**N**.
3. In the **New File** dialog box, under the **General** category, choose **Visual C# Class**, and then choose **Open**.

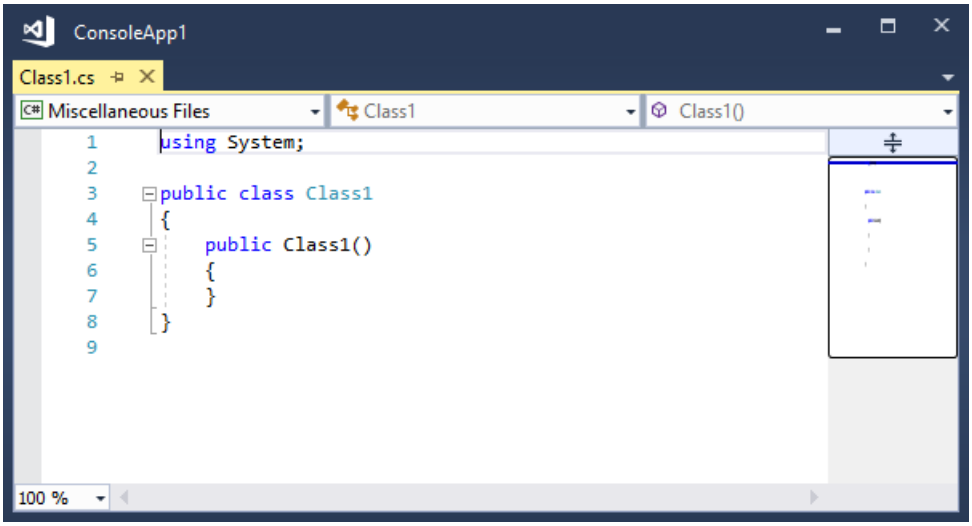
   A new file opens in the editor with the skeleton of a C# class. (Notice that we don't have to create a full Visual Studio project to gain some of the benefits that the code editor offers; all you need is a code file!)



1. Open Visual Studio. Press **Esc** or click **Continue without code** on the start window to open the development environment.

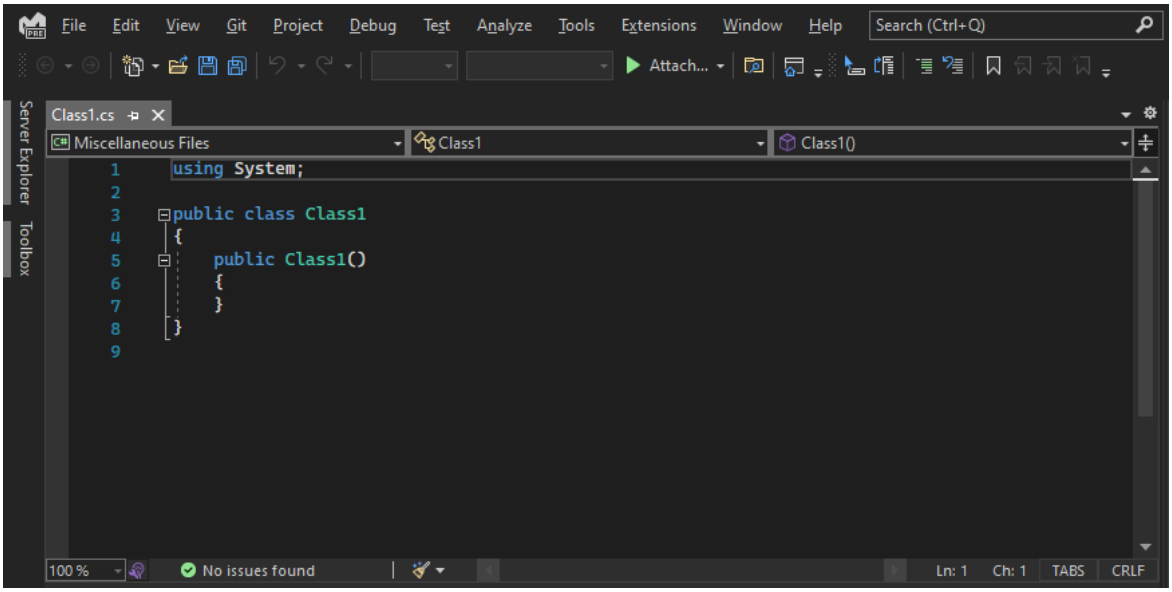
2. From the **File** menu on the menu bar, choose **New** > **File**, or press **Ctrl**+**N**.
3. In the **New File** dialog box, under the **General** category, choose **Visual C# Class**, and then choose **Open**.

   A new file opens in the editor with the skeleton of a C# class. (Notice that we don't have to create a full Visual Studio project to gain some of the benefits that the code editor offers; all you need is a code file!)



1. Open Visual Studio. Press **Esc**, or choose **Continue without code** on the start window, to open the development environment.
2. From the **File** menu on the menu bar, choose **New** > **File**, or press **Ctrl**+**N**.
3. In the **New File** dialog box, under the **General** category, choose **Visual C# Class**, and then choose **Open**.

   A new file opens in the editor with the skeleton of a C# class. You don't have to create a full Visual Studio project to gain some of the benefits that the code editor offers—all you need is a code file.



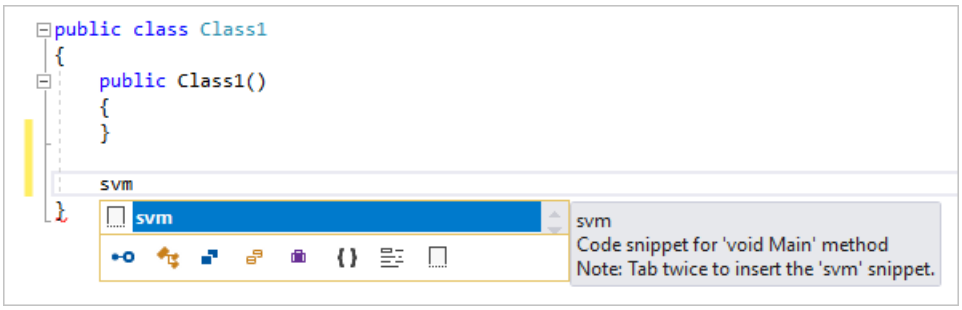
Use code snippets

Visual Studio provides useful *code snippets* that you can use to quickly and easily generate commonly used code blocks. Code snippets are available for different programming languages including C#, Visual Basic, and C++.

Let's add the C# `void Main` snippet to our file.

1. Place your cursor just above the final closing brace **}** in the file, and type the characters `svm` (which stands for `static void Main`—don't worry too much if you don't know what that means).

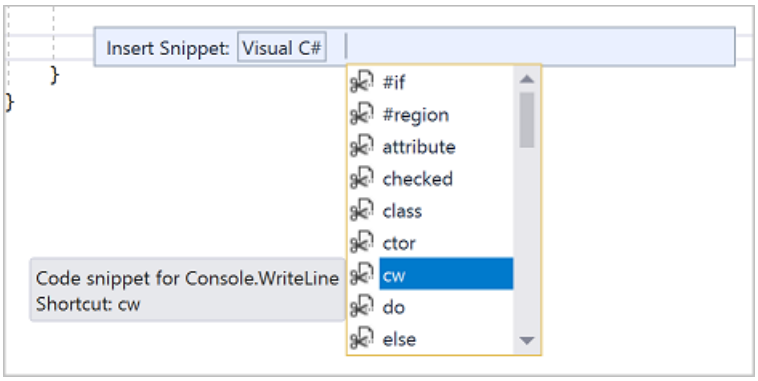
   A pop-up dialog box appears with information about the `svm` code snippet.



2. Press **Tab** twice to insert the code snippet.

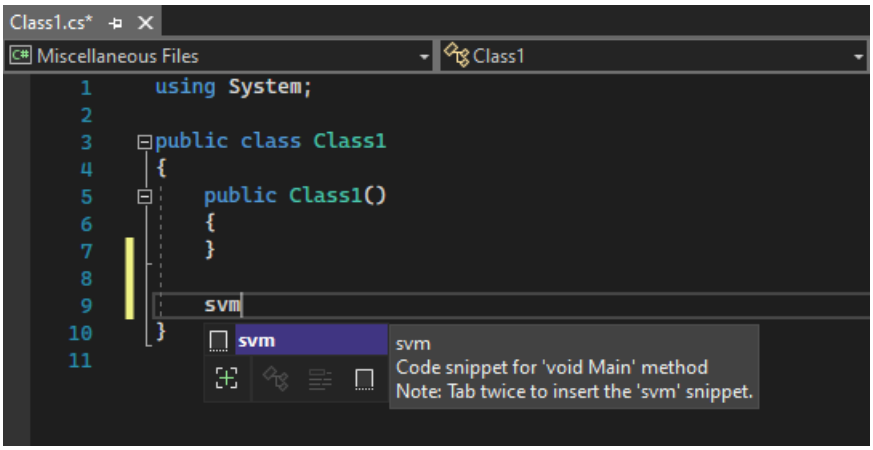
   You see the `static void Main()` method signature get added to the file. The Main() method is the entry point for C# applications.

The available code snippets vary for different programming languages. You can look at the available code snippets for your language by choosing **Edit** > **IntelliSense** > **Insert Snippet** or pressing **Ctrl**+**K**, **Ctrl**+**X**, and then choosing your language's folder. For C#, the list looks like this:



1. Place your cursor just above the final closing brace `}` in the file, and type the characters `svm`. `svm` stands for `static void Main`—don't worry if you don't know what that means yet.

   A pop-up dialog box appears with information about the `svm` code snippet.

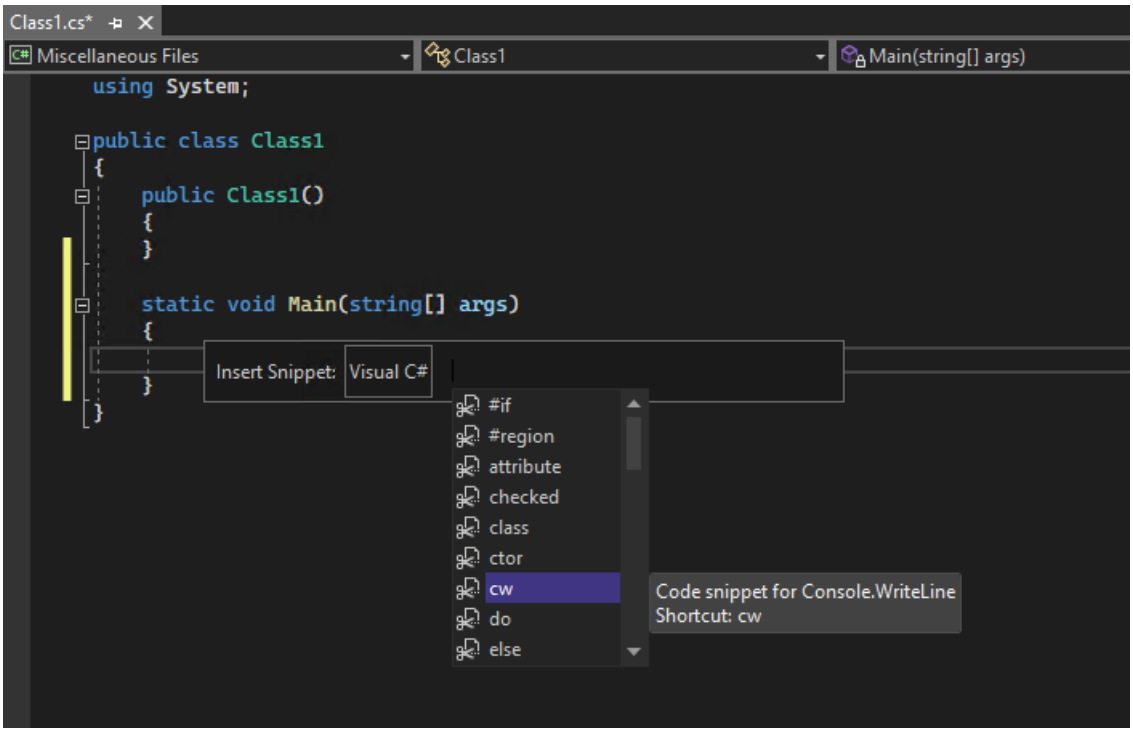


2. Press **Tab** twice to insert the code snippet.

   You'll see the `static void Main()` method signature get added to the file. The Main() method is the entry point for C# applications.

Available code snippets vary for different programming languages. You can look at the available code snippets

for your language by choosing **Edit** > **IntelliSense** > **Insert Snippet** or pressing **Ctrl**+**K**, **Ctrl**+**X**, and then choosing the folder for your programming language. For C#, the snippet list looks like this:



The list includes snippets for creating a class, a constructor, a for loop, an if or switch statement, and more.

Comment out code

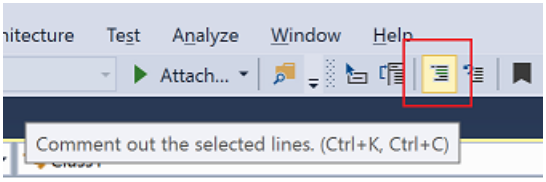
The toolbar, which is the row of buttons under the menu bar in Visual Studio, can help make you more productive as you code. For example, you can toggle IntelliSense completion mode (IntelliSense is a coding aid that displays a list of matching methods, amongst other things), increase or decrease a line indent, or comment out code that you don't want to compile. In this section, we'll comment out some code.

1. Paste the following code into the `Main()` method body.

```
// _words is a string array that we'll sort alphabetically
string[] _words = {
    "the",
    "quick",
    "brown",
    "fox",
    "jumps"
};

string[] morewords = {
    "over",
    "the",
    "lazy",
    "dog"
};

IEnumerable<string> query = from word in _words
                            orderby word.Length
                            select word;
```

2. We're not using the `morewords` variable, but we may use it later so we don't want to completely delete it.

Instead, let's comment out those lines. Select the entire definition of `morewords` to the closing semicolon, and then choose the **Comment out the selected lines** button on the toolbar. If you prefer to use the keyboard, press **Ctrl**+**K**, **Ctrl**+**C**.



The C# comment characters `//` are added to the beginning of each selected line to comment out the code.

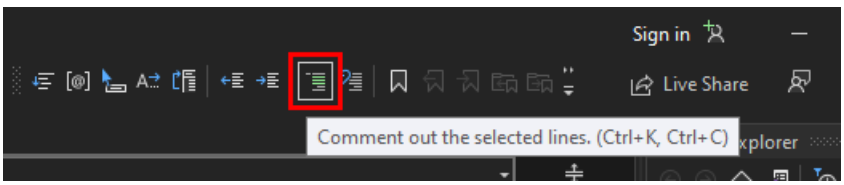
The toolbar, which is the row of buttons under the menu bar in Visual Studio, helps make you more productive as you code. For example, you can toggle IntelliSense completion mode, increase or decrease a line indent, or comment out code that you don't want to compile.

Let's comment out some code.

1. Paste the following code into the `Main()` method body.

```
// someWords is a string array.
string[] someWords = {
    "the",
    "quick",
    "brown",
    "fox",
    "jumps"
};

string[] moreWords = {
    "over",
    "the",
    "lazy",
    "dog"
};

// Alphabetically sort the words.
IEnumerable<string> query = from word in someWords
                            orderby word
                            select word;
```

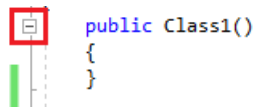
2. We're not using the `moreWords` variable, but we might use it later so we don't want to delete it. Instead, we'll comment out those lines. Select the entire definition of `moreWords` down to the closing semicolon, and then choose the **Comment out the selected lines** button on the toolbar. If you prefer to use the keyboard, press **Ctrl**+**E**, **Ctrl**+**C**.



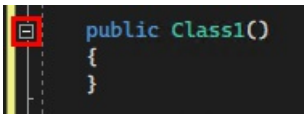
The C# comment characters `//` are added to the beginning of each selected line to comment out the code.

Collapse code blocks

We don't want to see the empty constructor that was generated for `Class1`, so to unclutter our view of the code, let's collapse it. Choose the small gray box with the minus sign inside it in the margin of the first line of the constructor. Or, if you prefer to use the keyboard, place the cursor anywhere in the constructor code and press **Ctrl+M**, **Ctrl+M**.



The code block collapses to just the first line, followed by an ellipsis (`...`). To expand the code block again, click the same gray box that now has a plus sign in it, or press **Ctrl+M**, **Ctrl+M** again. This feature is called Outlining and is especially useful when you're collapsing long methods or entire classes.



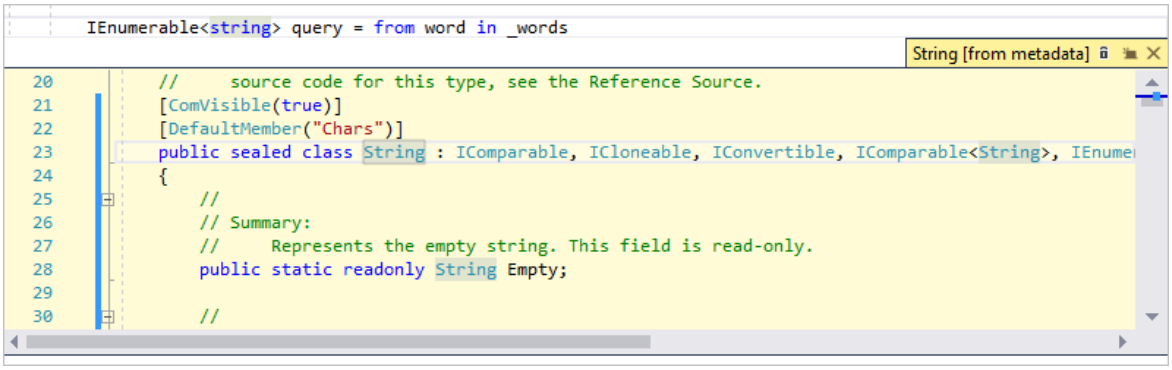
The code block collapses to just the first line, followed by an ellipsis (`...`). To expand the code block again, select the same gray box that now has a plus sign in it, or press **Ctrl+M**, **Ctrl+M** again. This feature is called Outlining and is especially useful when you're collapsing long methods or entire classes.

View symbol definitions

The Visual Studio editor makes it easy to inspect the definition of a type, method, etc. One way is to navigate to the file that contains the definition, for example by choosing **Go to Definition** or pressing **F12** anywhere the symbol is referenced. An even quicker way that doesn't move your focus away from the file you're working in is to use Peek Definition. Let's peek at the definition of the `string` type.

1. Right-click on any occurrence of `string` and choose **Peek Definition** from the content menu. Or, press **Alt+F12**.

   A pop-up window appears with the definition of the `String` class. You can scroll within the pop-up window, or even peek at the definition of another type from the peeked code.



2. Close the peeked definition window by choosing the small box with an "x" at the top right of the pop-up window.

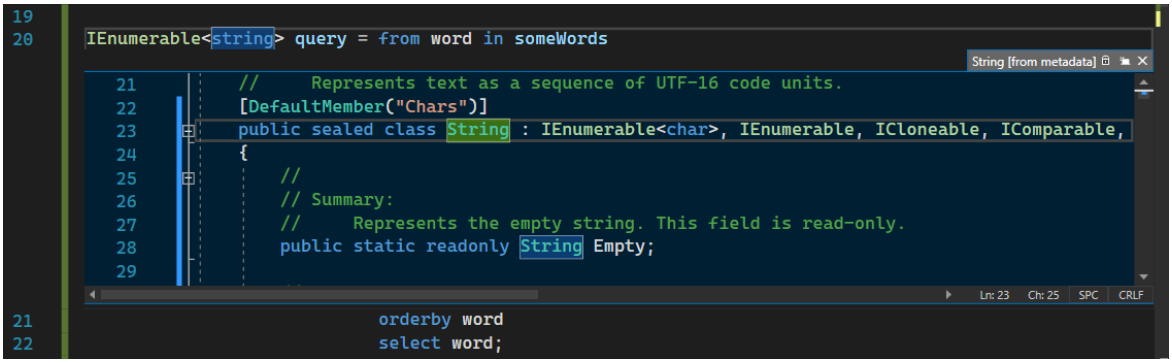
The Visual Studio editor makes it easy to inspect the definition of a type, method, or variable. One way is to go to the definition, in whichever file has it, by choosing **Go to Definition** or pressing **F12** anywhere a symbol is referenced. An even quicker way that doesn't move your focus away from the code you're working on is to use Peek Definition.

Let's peek at the definition of the `string` type.

1. Right-click on any occurrence of `string` and choose **Peek Definition** from the content menu. Or, press

Alt+F12.

A pop-up window appears with the definition of the `String` class. You can scroll within the pop-up window, or even peek at the definition of another type from the peeked code.



2. Close the peek definition window by choosing the small box with an "x" at the top right of the pop-up window.

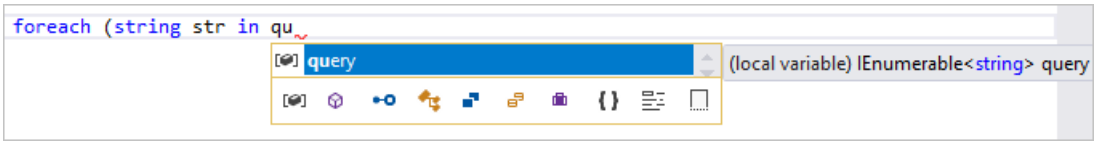
Use IntelliSense to complete words

IntelliSense is an invaluable resource when you're coding. It can show you information about available members of a type, or parameter details for different overloads of a method. You can also use IntelliSense to complete a word after you type enough characters to disambiguate it. Let's add a line of code to print out the ordered strings to the console window, which is the standard place for output from the program to go.

1. Below the `query` variable, start typing the following code:

```
foreach (string str in qu
```

You see IntelliSense show you **Quick Info** about the `query` symbol.



2. To insert the rest of the word `query` by using IntelliSense's word completion functionality, press **Tab**.

3. Finish off the code block to look like the following code. You can even practice using code snippets again by entering `cw` and then pressing **Tab** twice to generate the `Console.WriteLine` code.

```
foreach (string str in query)
{
   Console.WriteLine(str);
}
```

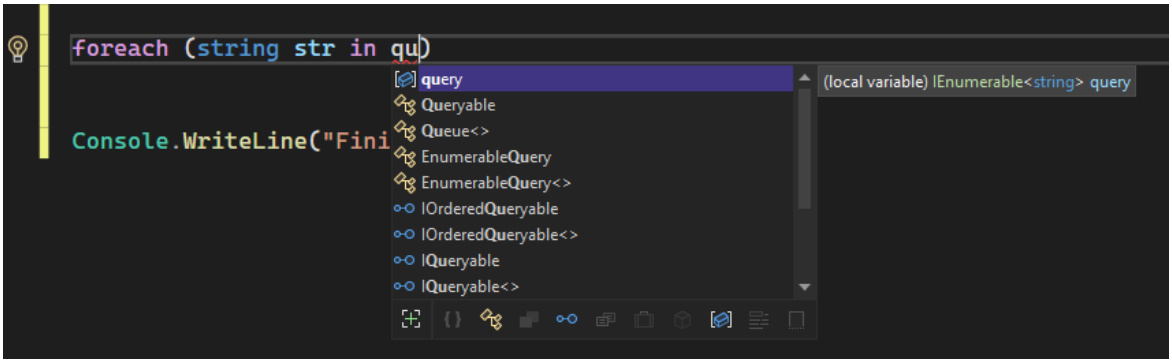
IntelliSense is an invaluable resource when you're coding. It can show you information about available members of a type, or parameter details for different overloads of a method. You can also use IntelliSense to complete a word after you type enough characters to disambiguate it.

Let's add a line of code to print out the ordered strings to the console window, which is the standard place for output from the program to go.

1. Below the `query` variable, start typing the following code:

```
foreach (string str in qu
```

You'll see an IntelliSense pop-up appear with information about the `query` symbol.



2. To insert the rest of the word `query` by using IntelliSense word completion, press **Tab**.

3. Finish off the code block to look like the following code. You can practice further with code snippets by entering `cw` and then pressing **Tab** twice to generate the `Console.WriteLine` statement.

```
foreach (string str in query)
{
   Console.WriteLine(str);
}
```

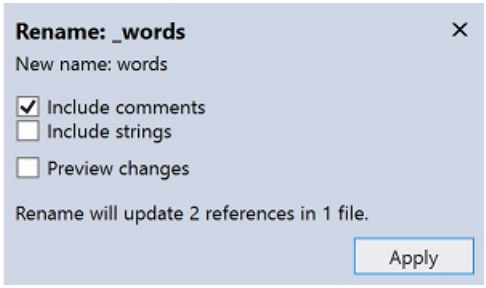
Refactor a name

Nobody gets code right the first time, and one of the things you might have to change is the name of a variable or method. Let's try out Visual Studio's refactor functionality to rename the `_words` variable to `words`.

1. Place your cursor over the definition of the `_words` variable, and choose **Rename** from the right-click or context menu, or press **Ctrl**+**R**, **Ctrl**+**R**.

   A pop-up **Rename** dialog box appears at the top right of the editor.

2. Enter the desired name **words**. Notice that the reference to `words` in the query is also automatically renamed. Before you press **Enter**, select the **Include comments** checkbox in the **Rename** pop-up box.



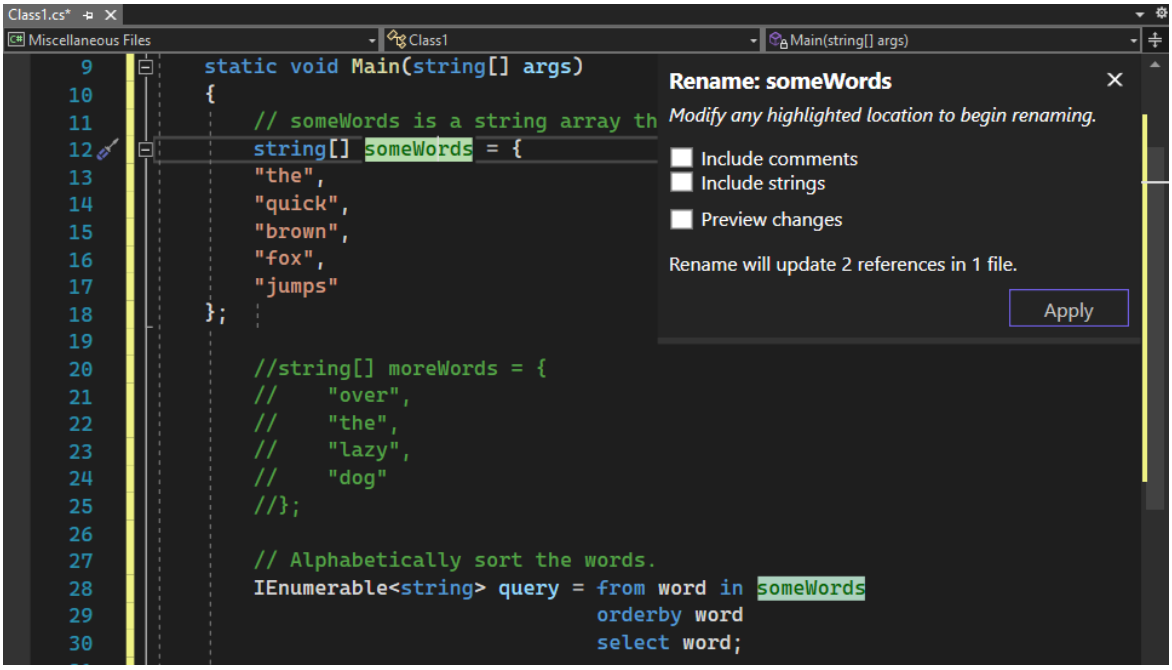
3. Press **Enter**.

   Both occurrences of `words` have been renamed, as well as the reference to `words` in the code comment.

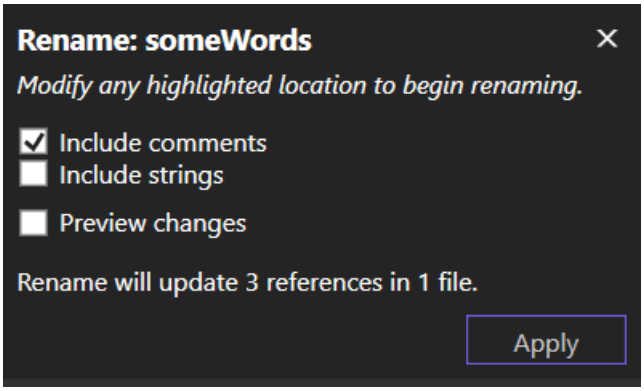
Nobody gets code right the first time, and one of the things you might have to change is the name of a variable or method. Let's try out Visual Studio's refactor functionality to rename the `someWords` variable to `unsortedWords`.

1. Place your cursor over the definition of the `someWords` variable, and choose **Rename** from the right-click or context menu, or press **F2**.

A **Rename** dialog box appears at the top right of the editor.



2. Enter the desired name **unsortedWords**. You'll see that the reference to `unsortedWords` in the `query` assignment statement is also automatically renamed. Before you press **Enter**, select the **Include comments** checkbox in the **Rename** pop-up box.



3. Press **Enter**, or choose **Apply** in the **Rename** dialog box.

   Both occurrences of `someWords` in your code have been renamed, as well as the text `someWords` in your code comment.

Next steps

Learn about projects and solutions

See also

- Code snippets
- Navigate code
- Outlining
- Go To Definition and Peek Definition
- Refactoring
- Use IntelliSense

Introduction to projects and solutions

10/28/2021 • 11 minutes to read • Edit Online

This introductory article explores what it means to create a *solution* and a *project* in Visual Studio. A solution is a container to organize one or more related code projects, like a class library project and a corresponding test project.

As an educational exercise to understand the concept of a project, you'll construct a solution and project from scratch. Ordinarily, you'd use Visual Studio project *templates* to create new projects. You'll also look at the properties of a project and some of the files it can contain, and create a reference from one project to another.

> **NOTE**
> Developing apps in Visual Studio doesn't require solutions and projects. You can just open a folder that contains code and start coding, building, and debugging. For example, a cloned GitHub repo might not contain Visual Studio projects and solutions. For more information, see Develop code in Visual Studio without projects or solutions.

If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

If you haven't already installed Visual Studio 2019, go to the Visual Studio downloads page to install it for free.

If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

Solutions and projects

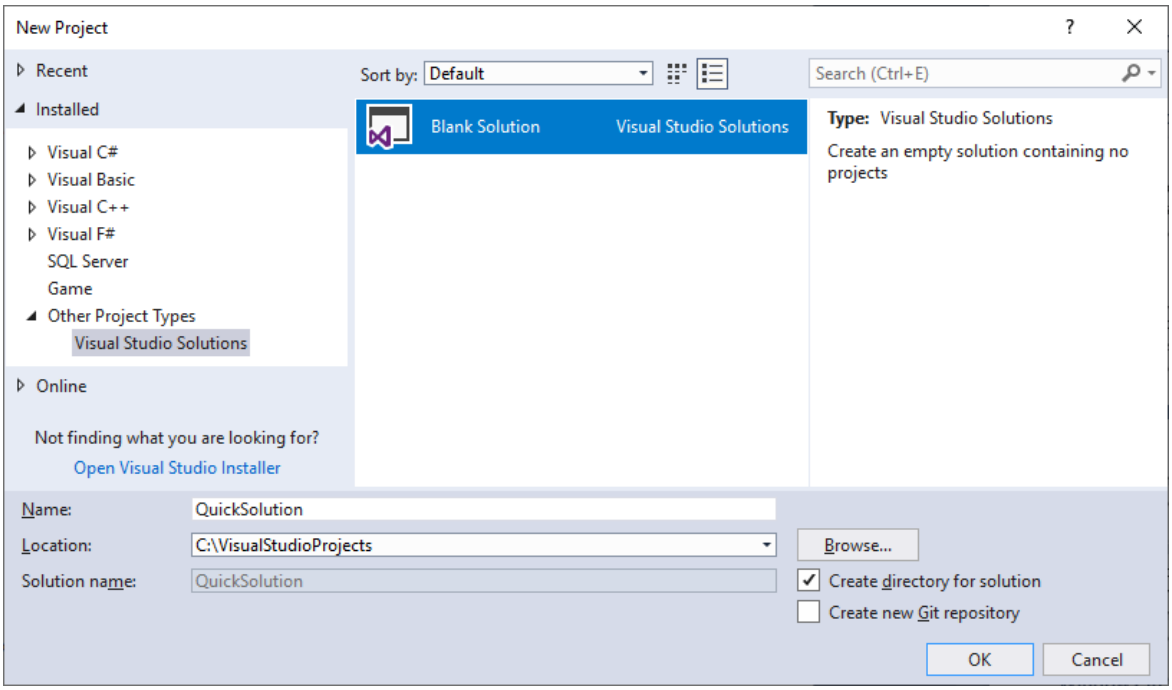
In Visual Studio, a solution isn't an "answer". A solution is simply a container Visual Studio uses to organize one or more related projects. When you open a solution, Visual Studio automatically loads all the projects that the solution contains.

Create a solution

Start your exploration by creating an empty solution. After you get to know Visual Studio, you probably won't create empty solutions very often. When you create a new project, Visual Studio automatically creates a solution for the project unless a solution is already open.

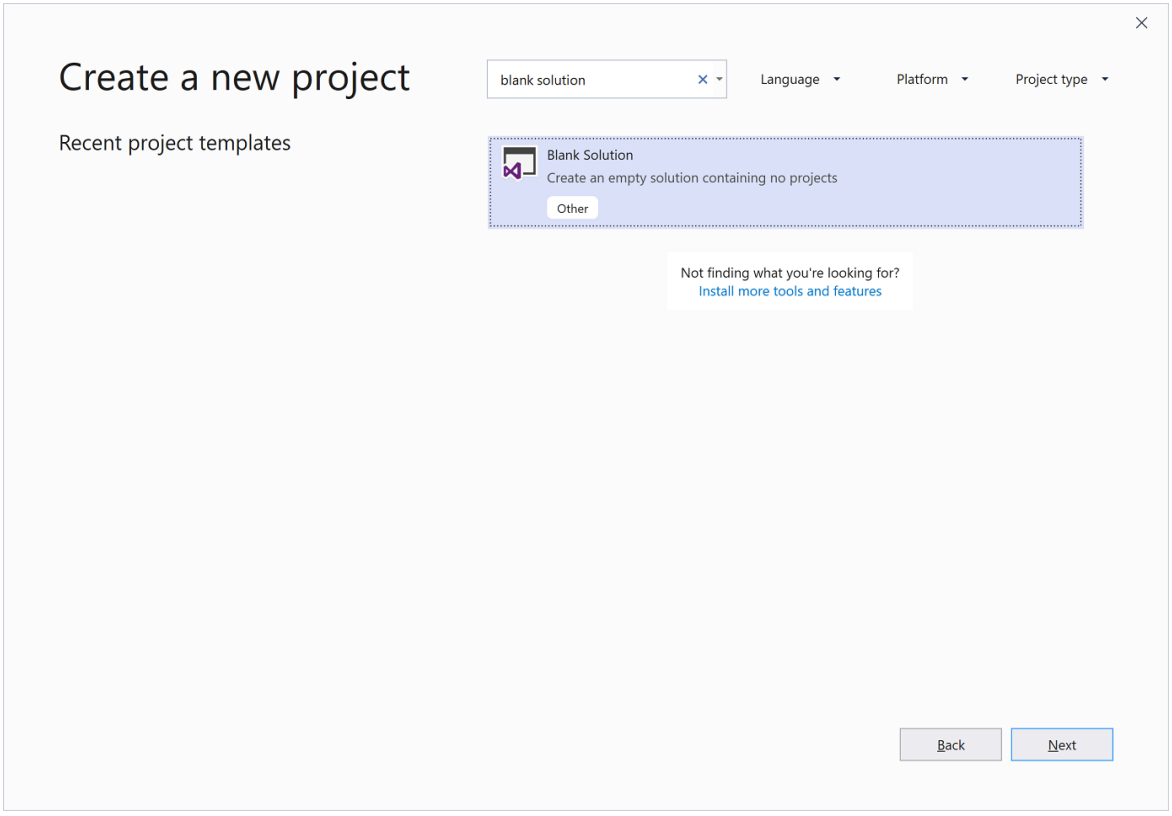
1. Open Visual Studio.
2. On the top menu bar, select **File** > **New** > **Project**.

   The **New Project** dialog box opens.
3. In the left pane, expand **Other Project Types**, then select **Visual Studio Solutions**. In the center pane, select the **Blank Solution** template. Name your solution **QuickSolution**, then select the **OK** button.



The **Start Page** closes, and a solution appears in **Solution Explorer** on the right-hand side of the Visual Studio window. You'll probably use **Solution Explorer** often, to browse the contents of your projects.

1. Open Visual Studio.
2. On the start window, select **Create a new project**.
3. On the **Create a new project** page, enter **blank solution** into the search box, select the **Blank Solution** template, and then select **Next**.

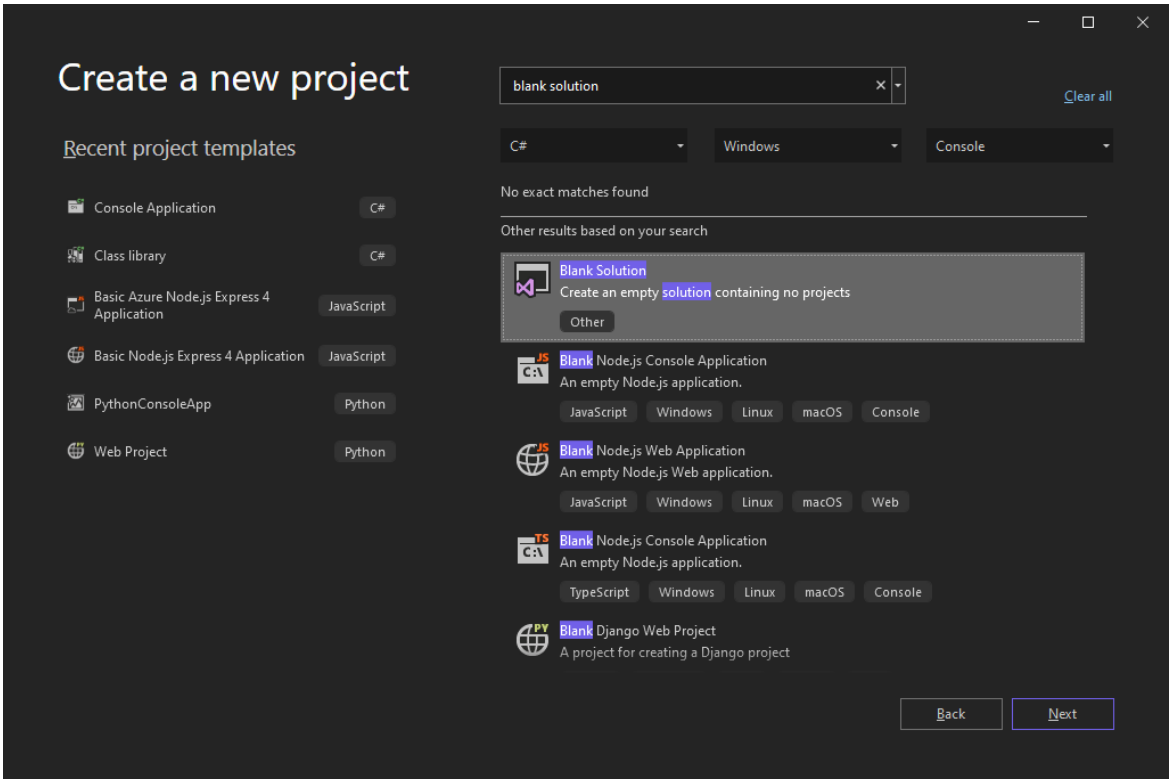


> **TIP**
> If you have several workloads installed, the **Blank Solution** template might not appear at the top of your list of search results. Try scrolling to the **Other results based on your search** section of the list. It should appear there.

4. Name the solution **QuickSolution**, and then select **Create**.

   A solution appears in **Solution Explorer** on the right-hand side of the Visual Studio window. You'll probably use **Solution Explorer** often, to browse the contents of your projects.

1. Open Visual Studio, and on the start window, select **Create a new project**.
2. On the **Create a new project** page, type *blank solution* into the search box, select the **Blank Solution** template, and then select **Next**.



> **TIP**
> If you have several workloads installed, the **Blank Solution** template might not appear at the top of your list of search results. Try scrolling through **Other results based on your search** to find the template.

3. On the **Configure your new project** page, name the solution **QuickSolution**, and then select **Create**.

   The **QuickSolution** solution appears in **Solution Explorer** on the right side of the Visual Studio window. You'll use **Solution Explorer** often to browse the contents of your projects.

Add a project

Now add your first project to the solution. Start with an empty project, and add the items you need.

1. From the right-click or context menu of **Solution 'QuickSolution'** in **Solution Explorer**, select **Add** > **New Project**.

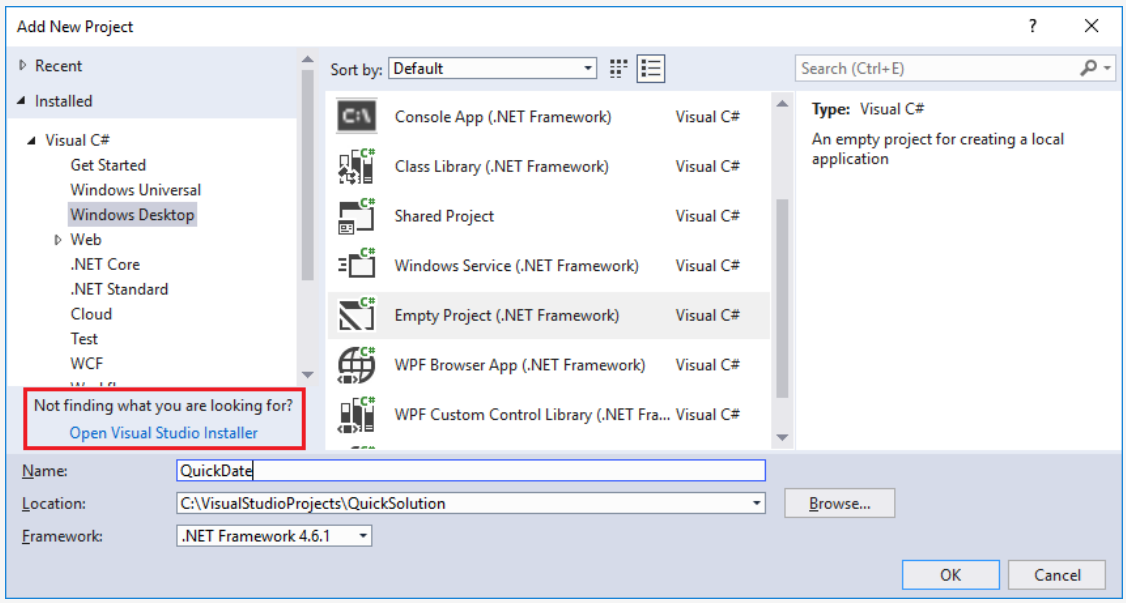
   The **Add New Project** dialog box opens.

2. In the left pane, expand **Visual C#** and select **Windows Desktop**. Then, in the middle pane, select the

Empty Project (.NET Framework) template. Name the project **QuickDate**, then select **OK**.

A project named QuickDate appears beneath the solution in **Solution Explorer**. Currently it contains a single file called *App.config*.

> **NOTE**
>
> If you don't see **Visual C#** in the left pane of the dialog box, you must install the **.NET desktop development** Visual Studio workload. Visual Studio uses workload-based installation to install only the components you need for the type of development you do. An easy way to install a new workload is to select the **Open Visual Studio Installer** link in the bottom left corner of the **Add New Project** dialog box. After Visual Studio Installer launches, select the **.NET desktop development** workload and then the **Modify** button.



1. From the right-click or context menu of **Solution 'QuickSolution'** in **Solution Explorer**, select **Add** > **New Project**.

   A dialog box opens that says **Add a new project**.

2. Enter the text **empty** into the search box at the top, and then select **C#** under **Language**.

3. Select the **Empty Project (.NET Framework)** template, and then select **Next**.

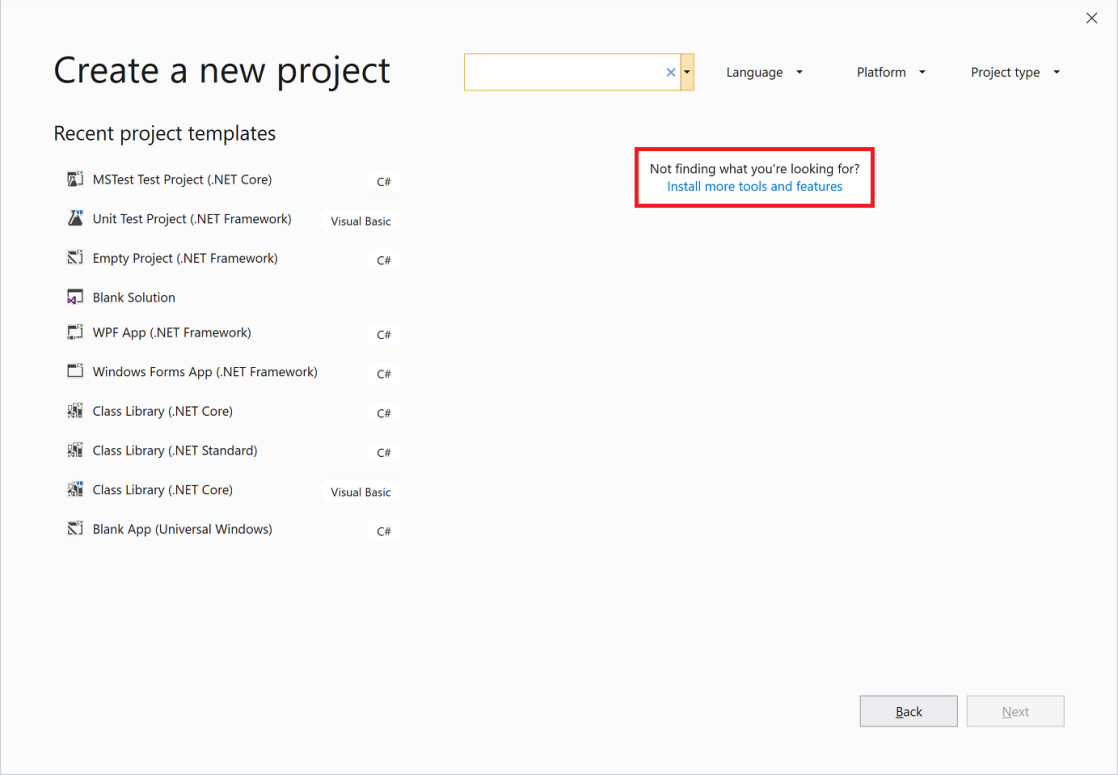
4. Name the project **QuickDate**, then select **Create**.

   A project named QuickDate appears beneath the solution in **Solution Explorer**. Currently it contains a single file called *App.config*.

NOTE

If you don't see the **Empty Project (.NET Framework)** template, you must install the **.NET desktop development** Visual Studio workload. Visual Studio uses workload-based installation to install only the components you need for the type of development you do.

An easy way to install a new workload when you're creating a new project is to select the **Install more tools and features** link under the text that says **Not finding what you're looking for?**. After Visual Studio Installer launches, select the **.NET desktop development** workload and then the **Modify** button.



1. Right-click **Solution 'QuickSolution'** in **Solution Explorer**, and select **Add** > **New Project** from the context menu.

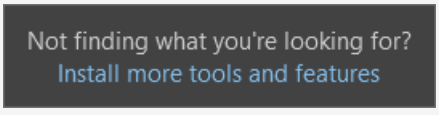
2. On the **Add a new project** page, type *empty* into the search box at the top, and select **C#** under **All languages**.

3. Select the C# **Empty Project (.NET Framework)** template, and then select **Next**.

NOTE

Visual Studio uses workload-based installation to install only the components you need for the type of development you do. If you don't see the **Empty Project (.NET Framework)** template, you need to install the **.NET desktop development** Visual Studio workload.

An easy way to install a new workload when you're creating a new project is to select the **Install more tools and features** link under the text that says **Not finding what you're looking for?**. In the Visual Studio Installer, select the **.NET desktop development** workload, and then select **Modify**.



4. On the **Configure your new project** page, name the project **QuickDate**, and then select **Create**.

   The **QuickDate** project appears under the solution in **Solution Explorer**. Currently the project contains

a single file called **App.config**.

Add an item to the project

Add a code file to your empty project.

1. From the right-click or context menu of the **QuickDate** project in **Solution Explorer**, select **Add** > **New Item**.

   The **Add New Item** dialog box opens.

2. Expand **Visual C# Items**, and then select **Code**. In the middle pane, select the **Class** item template. Under **Name**, type *Calendar*, and then select **Add**.

   Visual Studio adds a file named *Calendar.cs* to the project. The *.cs* on the end is the file extension for C# code files. The **Calendar.cs** file appears in the **Solution Explorer** visual project hierarchy, and the file opens in the editor.

3. Replace the contents of the *Calendar.cs* file with the following code:

```
using System;

namespace QuickDate
{
    internal class Calendar
    {
        static void Main(string[] args)
        {
            DateTime now = GetCurrentDate();
            Console.WriteLine($"Today's date is {now}");
            Console.ReadLine();
        }

        internal static DateTime GetCurrentDate()
        {
            return DateTime.Now.Date;
        }
    }
}
```

   You don't need to understand everything the code is doing yet. Run the app by pressing **Ctrl**+**F5**, and see that the app prints today's date to the *console*, or standard output, window.

Add a second project

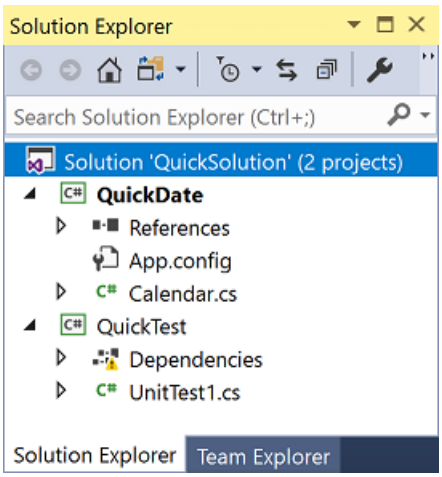
Solutions commonly contain more than one project, and these projects often reference each other. Some projects in a solution might be class libraries, some might be executable applications, and some might be unit test projects or websites.

To add a unit test project to your solution, start from a project template so you don't have to add another code file to the project.

1. From the right-click or context menu of **Solution 'QuickSolution'** in **Solution Explorer**, select **Add** > **New Project**.

2. In the left pane, expand **Visual C#** and select the **Test** category. In the middle pane, select the **MSTest Test Project (.NET Core)** project template. Name the project **QuickTest**, and then select **OK**.

   A second project is added to **Solution Explorer**, and a file named *UnitTest1.cs* opens in the editor.

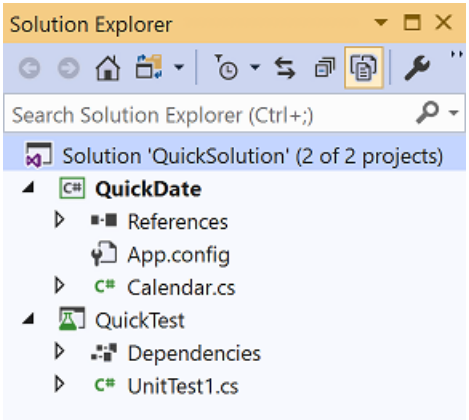


1. From the right-click or context menu of **Solution 'QuickSolution'** in **Solution Explorer**, select **Add** > **New Project**.
2. In the **Add a new project** dialog box, enter the text **unit test** into the search box at the top, and then select **C#** under **Language**.
3. Select the **Unit Test Project** project template for .NET Core, and then select **Next**.

   > **NOTE**
   > Starting in Visual Studio 2019 version 16.9, the MSTest project template name changed from **MSTest Unit Test Project (.NET Core)** to **Unit Test Project**. Several steps in the project creation changed in this update.

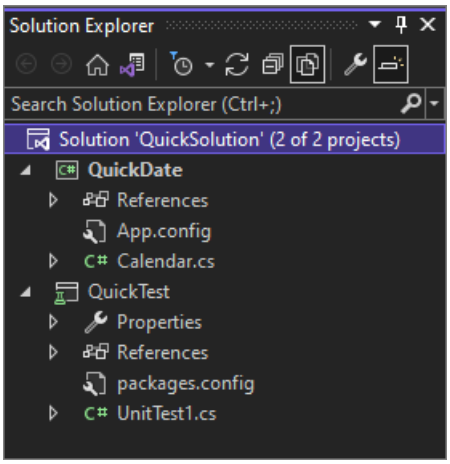
4. Name the project **QuickTest**, and then select **Next**.
5. Choose either the recommended target framework (.NET Core 3.1) or .NET 5, and then choose **Create**.

   A second project is added to **Solution Explorer**, and a file named *UnitTest1.cs* opens in the editor.



1. From the right-click or context menu of **Solution 'QuickSolution'** in **Solution Explorer**, select **Add** > **New Project**.
2. In the **Add a new project** dialog box, type *unit test* into the search box at the top, and then select **C#** under **All languages**.
3. Select the C# **Unit Test Project (.NET Framework)** project template, and then select **Next**.
4. On the **Configure your new project** page, name the project *QuickTest*, and then select **Create**.

   Visual Studio adds the **QuickTest** project to **Solution Explorer**, and the **UnitTest1.cs** file opens in the editor.



Add a project reference

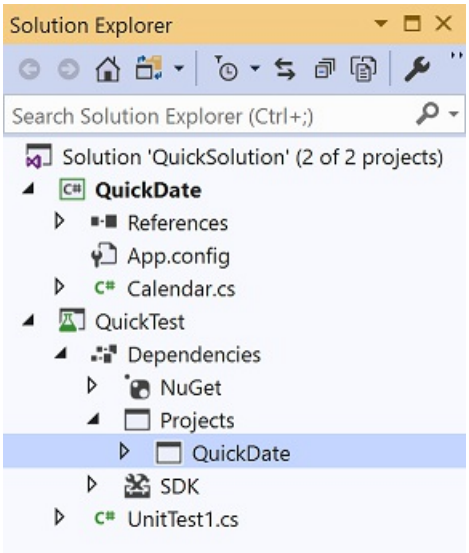
You'll use the new unit test project to test your method in the **QuickDate** project, so you need to add a reference to **QuickDate** to the **QuickTest** project. Adding the reference creates a *build dependency* between the two projects, meaning that when you build the solution, **QuickDate** builds before **QuickTest**.

1. Select the **Dependencies** node in the **QuickTest** project, and from the right-click or context menu, select **Add Reference**.

   The **Reference Manager** dialog box opens.

2. In the left pane, expand **Projects** and select **Solution**. In the middle pane, select the checkbox next to **QuickDate**, and then select **OK**.

   A reference to the **QuickDate** project is added.

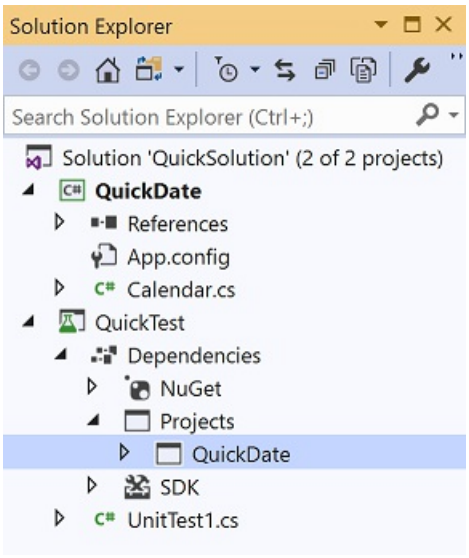


1. Select the **Dependencies** node in the **QuickTest** project, and from the right-click or context menu, select **Add Project Reference**.

   The **Reference Manager** dialog box opens.

2. In the left pane, expand **Projects**, and then select **Solution**. In the middle pane, select the checkbox next to **QuickDate**, and then select **OK**.

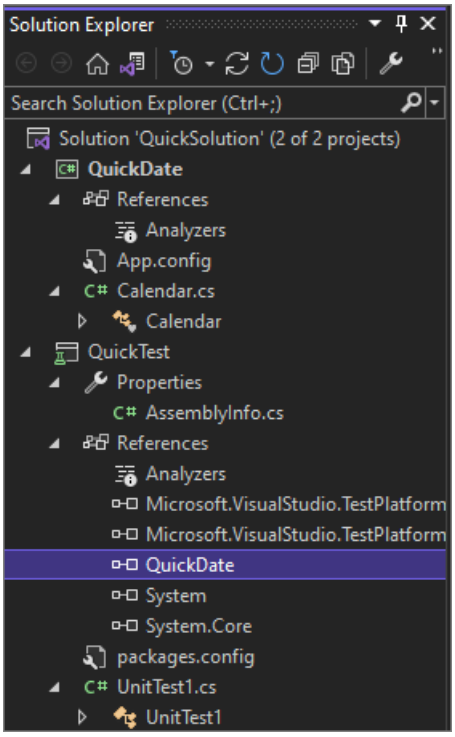
   A reference to the **QuickDate** project is added.



1. In **Solution Explorer**, right-click the **References** node of the **QuickTest** project, and select **Add Reference** from the context menu.

2. In the **Reference Manager** dialog box, under **Projects**, select the checkbox next to **QuickDate**, and then select **OK**.

   A reference to the **QuickDate** project appears under the **QuickTest** project in **Solution Explorer**.



Add test code

1. Now add test code to the C# test code file. Replace the contents of *UnitTest1.cs* with the following code:

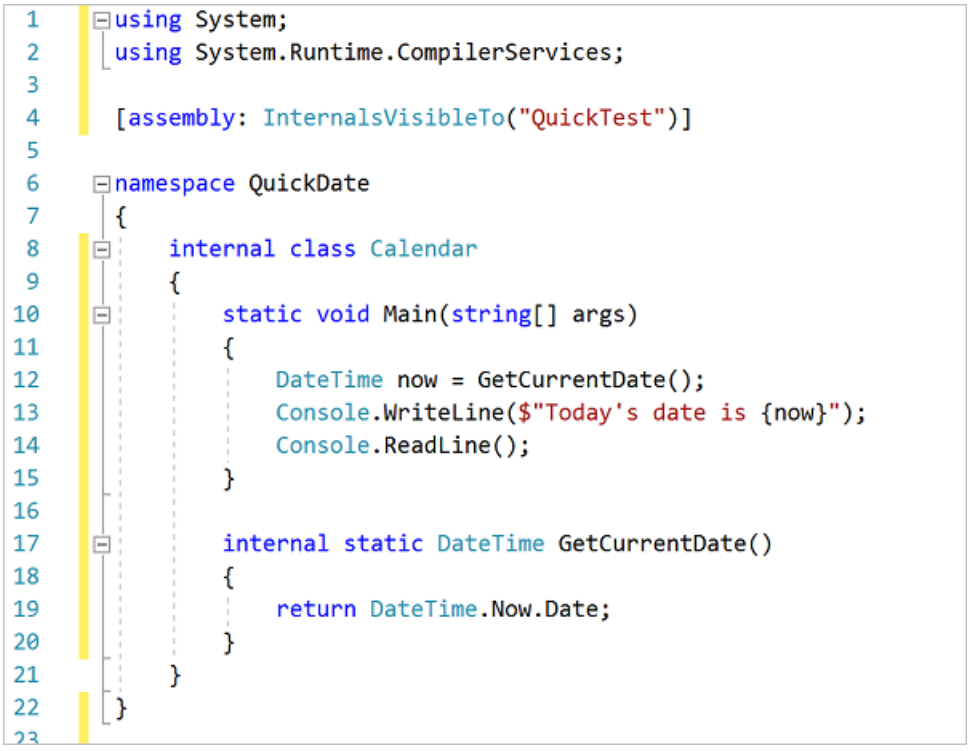
```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace QuickTest
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestGetCurrentDate()
        {
            Assert.AreEqual(DateTime.Now.Date, QuickDate.Calendar.GetCurrentDate());
        }
    }
}
```

A red squiggle appears under some of the code. You can fix this error by making the test project a friend assembly to the **QuickDate** project.

2. In the *Calendar.cs* file, add the following using statement and InternalsVisibleToAttribute attribute to the top of the file to resolve the error in the test project.

```
using System.Runtime.CompilerServices;

[assembly: InternalsVisibleTo("QuickTest")]
```

The **Calendar.cs** code should look like this screenshot:

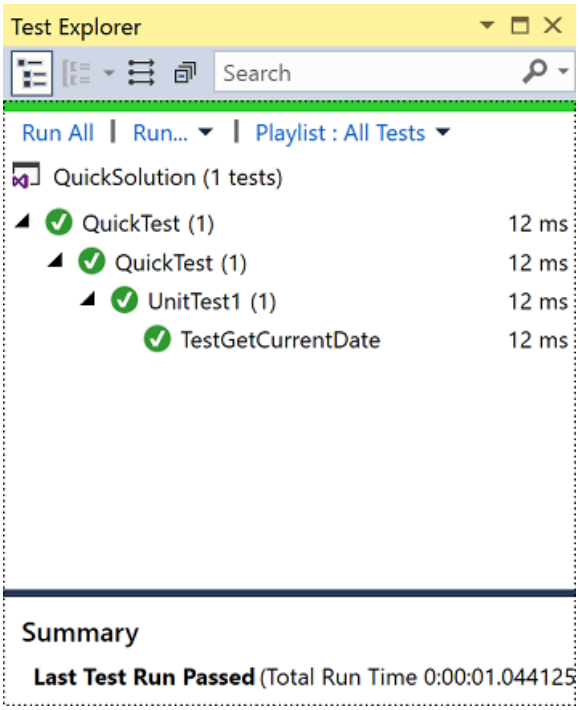


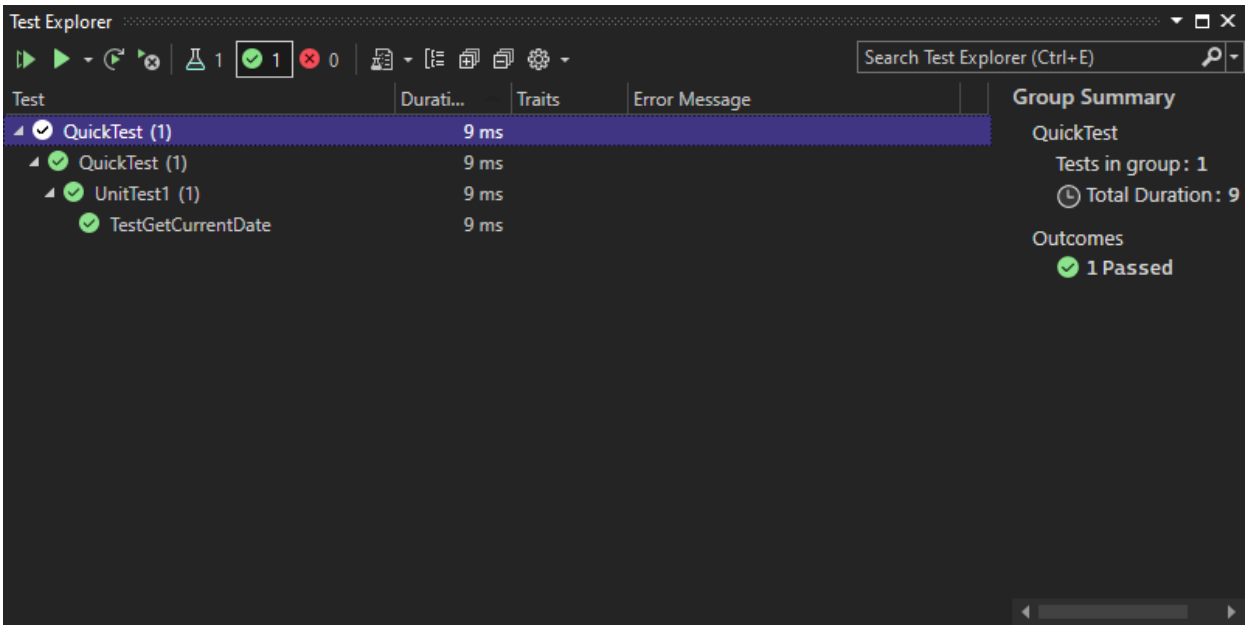
```
using System;
using System.Runtime.CompilerServices;


[assembly: InternalsVisibleTo("QuickTest")]


namespace QuickDate
{
    1 reference
    internal class Calendar
    {
        0 references
        static void Main(string[] args)
        {
            DateTime now = GetCurrentDate();
            Console.WriteLine($"Today's date is {now}");
            Console.ReadLine();
        }

        2 references
        internal static DateTime GetCurrentDate()
        {
            return DateTime.Now.Date;
        }
    }
}
```

Run the unit test

If you want to check that your unit test is working, choose **Test** > **Run** > **All Tests** from the menu bar. A window called **Test Explorer** opens, and you should see that the **TestGetCurrentDate** test passes.

To check that your unit test is working, choose **Test** > **Run All Tests** from the menu bar. The **Test Explorer** window opens, and you should see that the **TestGetCurrentDate** test passes.





> **TIP**
>
> If **Test Explorer** doesn't open automatically, open it by choosing **Test** > **Windows** > **Test Explorer** from the menu bar.

> **TIP**
>
> If **Test Explorer** doesn't open automatically, open it by choosing **Test** > **Test Explorer** from the menu bar.

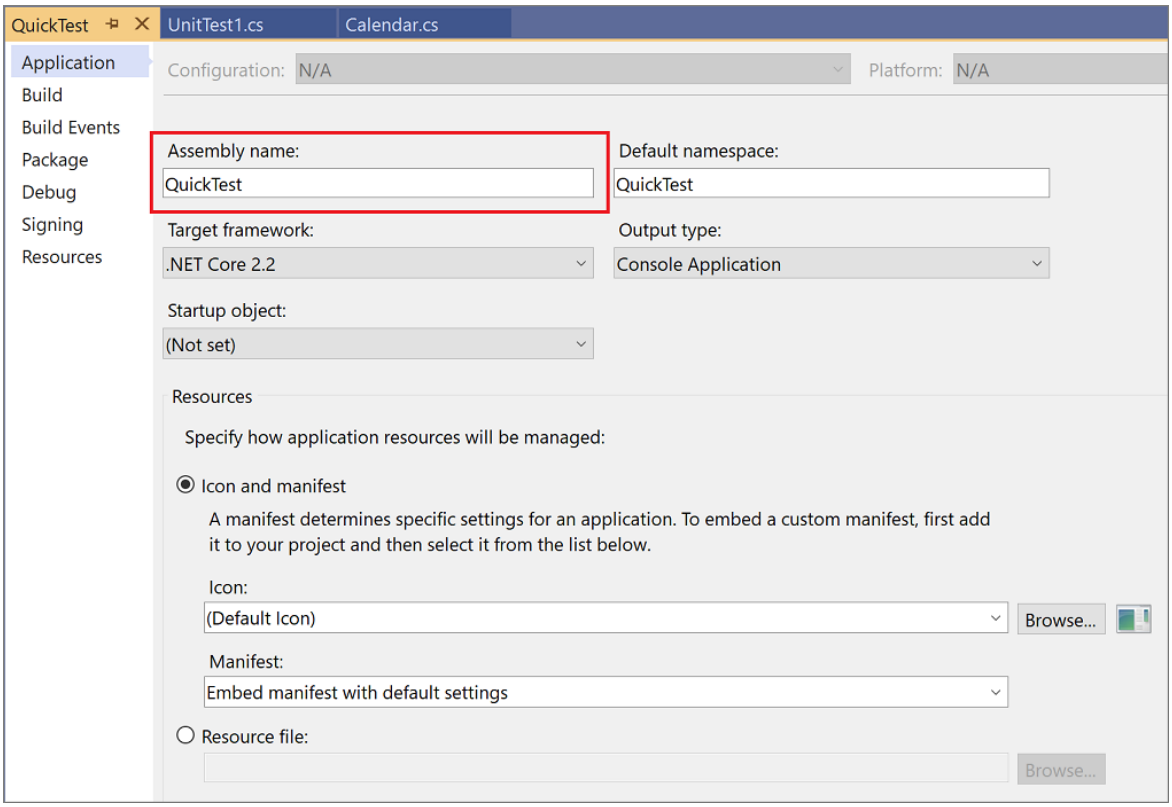
Project properties

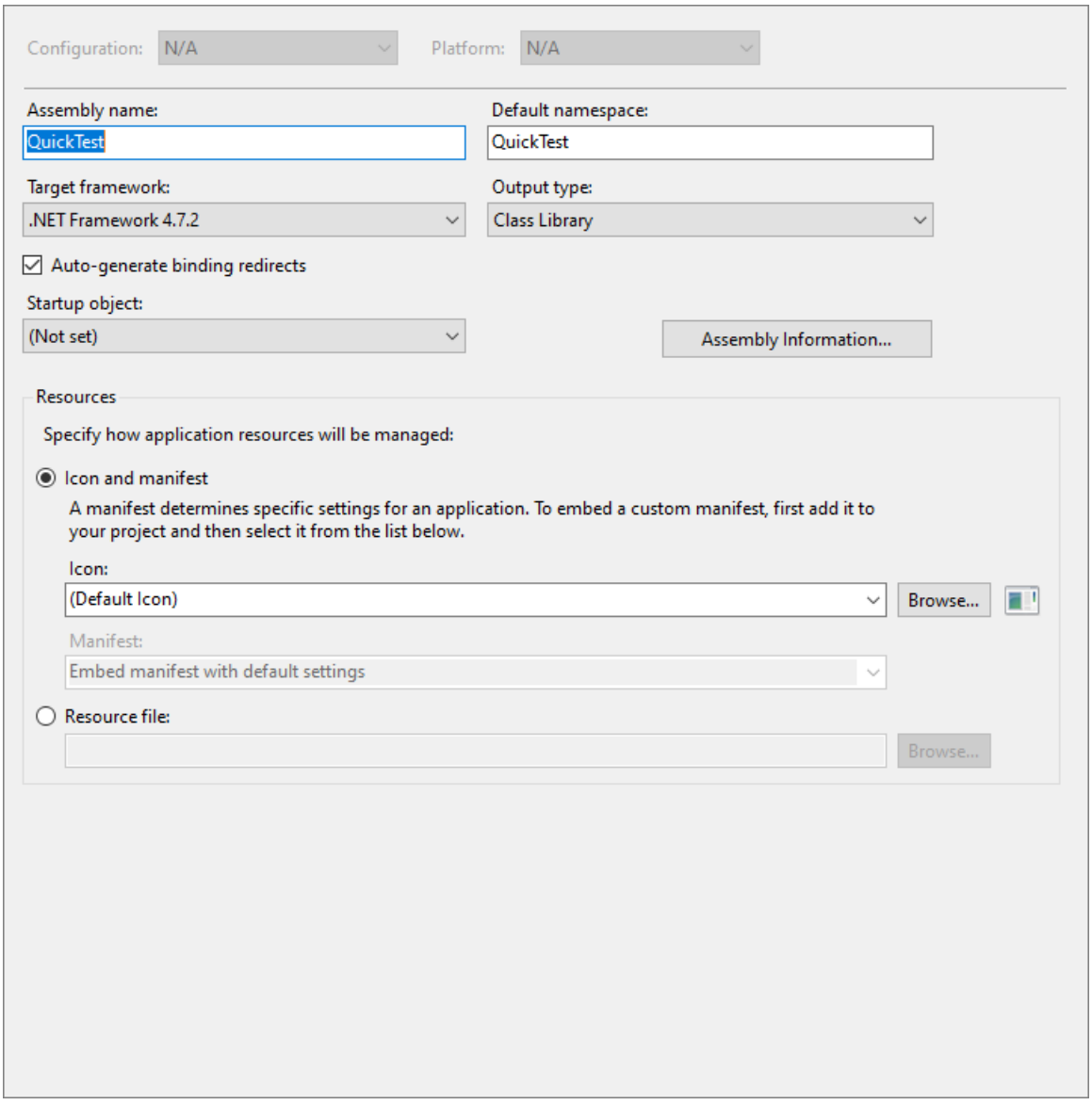
The line in the *Calendar.cs* file that contains the InternalsVisibleToAttribute attribute references the assembly name or file name of the **QuickTest** project. The assembly name might not always be the same as the project name. To find the assembly name of a project, use the project properties. The property pages contain various settings for the project.

1. In **Solution Explorer**, right-click the **QuickTest** project and select **Properties**, or select the project and press **Alt+Enter**.

   The *property pages* for the project open to the **Application** tab. The **Assembly name** of the QuickTest project is indeed **QuickTest**.

   If you want, you can change the name here. When you build the test project, the name of the resulting binary file then changes from *QuickTest.dll* to *<NewName>.dll*.





2. Explore some of the other tabs of the project's property pages, such as **Build** and **Debug**. These tabs are

different for different types of projects.

See also

- Work with projects and solutions
- Manage project and solution properties
- Manage references in a project
- Develop code in Visual Studio without projects or solutions
- Visual Studio IDE overview

Features of Visual Studio

10/28/2021 • 8 minutes to read • Edit Online

This article describes features for experienced developers, or developers who are already familiar with Visual Studio. For a basic introduction to Visual Studio, see the Visual Studio IDE overview.

Modular installation

In Visual Studio's modular installer, you choose and install the *workloads* you want. Workloads are groups of features that programming languages or platforms need to work. This modular strategy helps keep the Visual Studio installation footprint smaller, so it installs and updates faster.

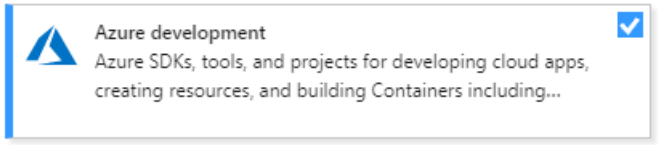
If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

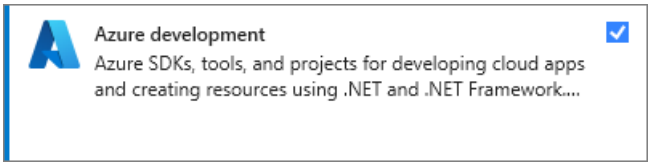
If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

To learn more about setting up Visual Studio on your system, see Install Visual Studio.

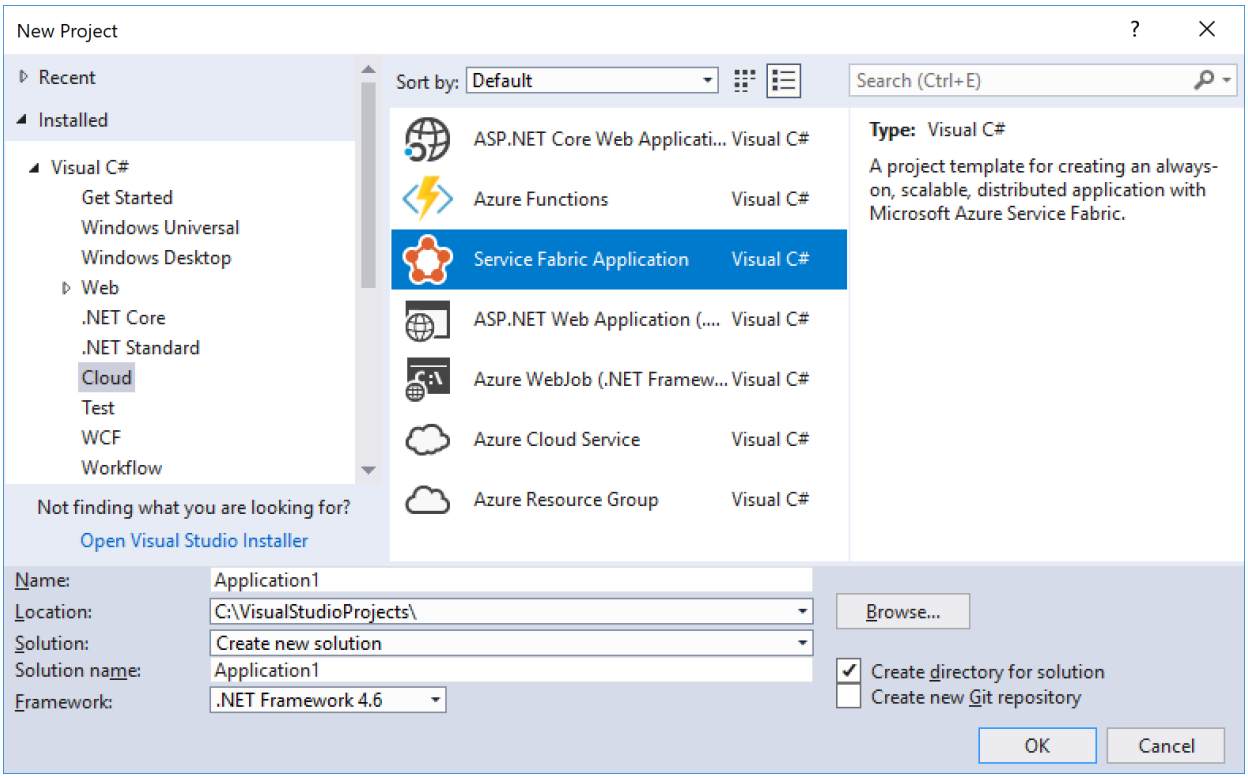
Create cloud-enabled Azure apps

Visual Studio has a suite of tools to easily create Microsoft Azure cloud-enabled applications. You can configure, build, debug, package, and deploy Azure apps and services directly from the Visual Studio integrated development environment (IDE). To get the Azure tools and project templates, select the **Azure development** workload when you install Visual Studio.

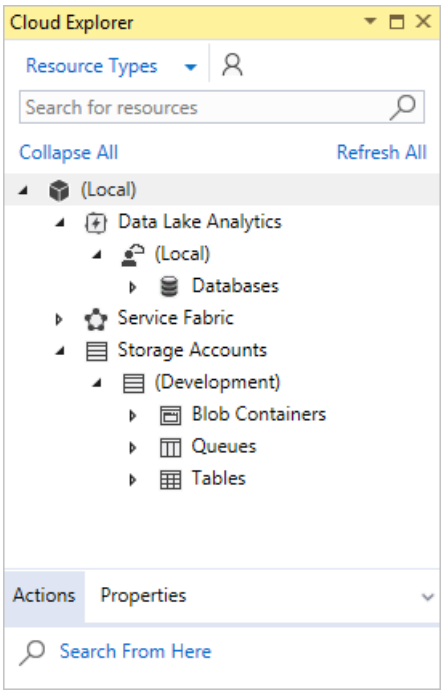




After you install the **Azure development** workload, the following **Cloud** templates for C# are available in the **New Project** dialog:



In Visual Studio, use Cloud Explorer to view and manage your Azure-based cloud resources. Cloud resources might include virtual machines (VMs), tables, and SQL databases. **Cloud Explorer** shows the Azure resources in all the accounts under the Azure subscription you're signed into. If an operation requires the Azure portal, **Cloud Explorer** has links to the place in the portal you need to go.

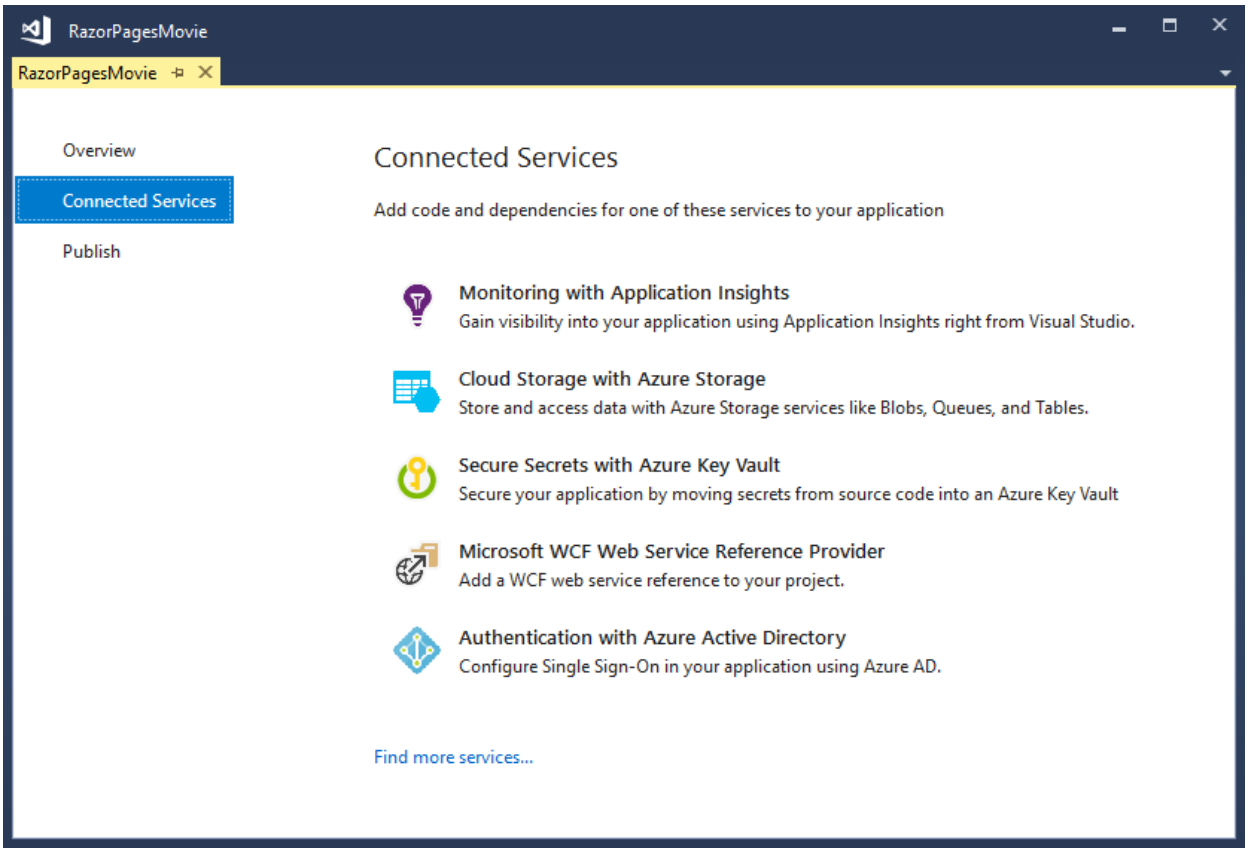


> **IMPORTANT**
>
> The Cloud Explorer window is retired in Visual Studio 2022. For more information, see Manage the resources associated with your Azure accounts in Visual Studio Cloud Explorer.
>
> Use the Azure portal to access Azure resources as necessary. You can continue to use the Azure node of Server Explorer in previous versions of Visual Studio.

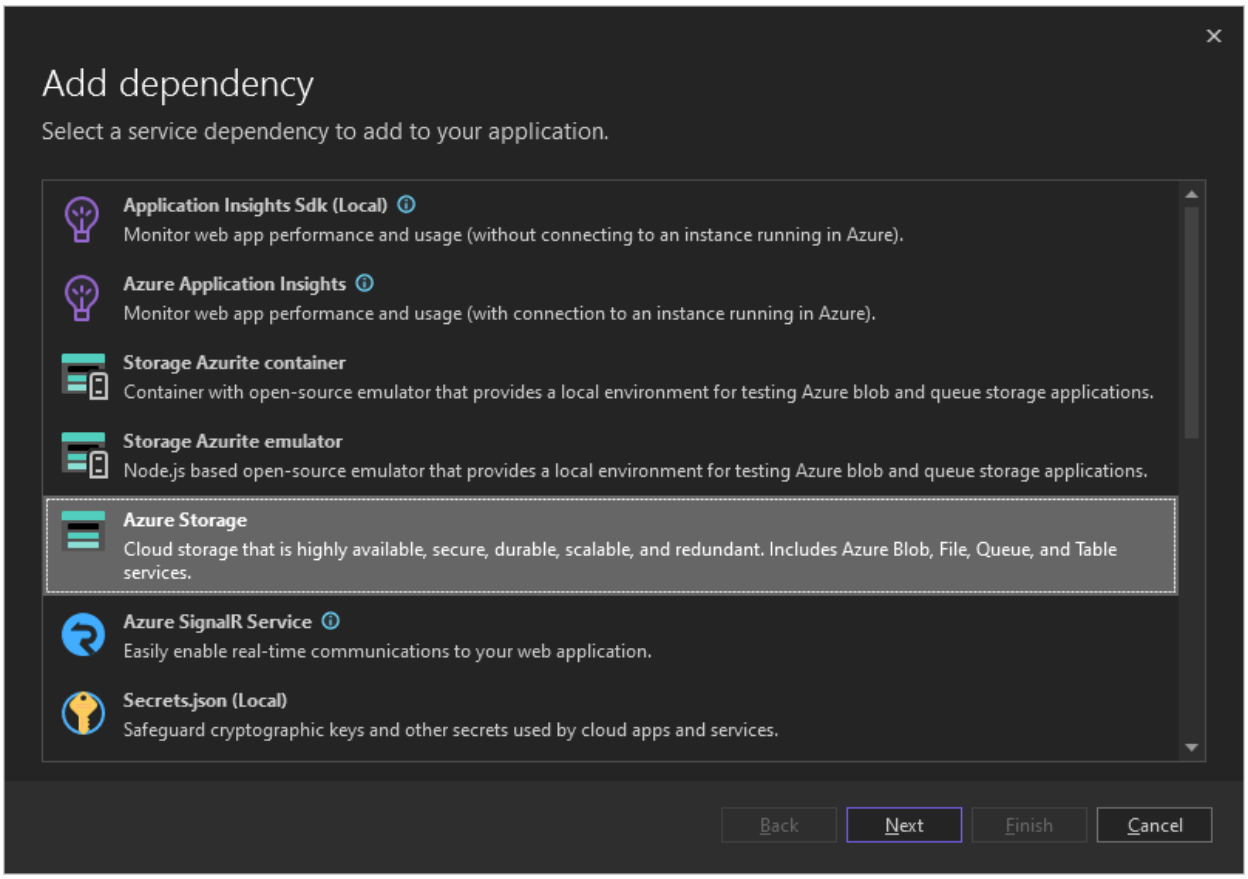
You can use Azure services for your apps by adding **Connected Services**, such as:

- Active Directory connected service, to use Azure Active Directory (Azure AD) accounts to connect to web apps
- Azure Storage connected service for blob storage, queues, and tables
- Key Vault connected service to manage secrets for web apps

The available **Connected Services** depend on your project type. Add a service by right-clicking the project in **Solution Explorer** and choosing **Add** > **Connected Service**.



On the **Connected Services** screen, select the link or the plus sign to **Add a service dependency**. On the **Add dependency** screen, select the service you want to add, and follow the screens to connect to your Azure subscription and service.



For more information, see Move to the cloud With Visual Studio and Azure.

Create web apps

Visual Studio can help you write apps for the web. You can create web apps by using ASP.NET, Node.js, Python, JavaScript, and TypeScript. Visual Studio supports many web frameworks, such as Angular, jQuery, and Express.

ASP.NET Core and .NET Core run on Windows, Mac, and Linux operating systems. ASP.NET Core is a major update to MVC, WebAPI, and SignalR. ASP.NET Core is designed from the ground up to provide a lean and composable .NET stack for building modern cloud-based web apps and services.

For more information, see Modern web tooling.

Build cross-platform apps and games

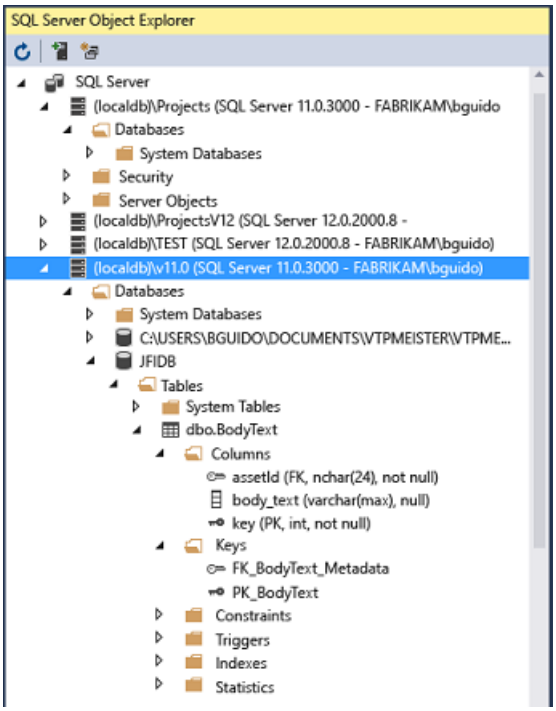
Visual Studio can build apps and games for macOS, Linux, and Windows, and for Android, iOS, and other mobile devices. With Visual Studio, you can build:

- .NET Core apps that run on Windows, macOS, and Linux.
- Mobile apps for iOS, Android, and Windows in C# and F# by using Xamarin.
- 2D and 3D games in C# by using Visual Studio Tools for Unity.
- Native C++ apps for iOS, Android, and Windows devices. Share common code in iOS, Android, and Windows libraries by using C++ for cross-platform development.

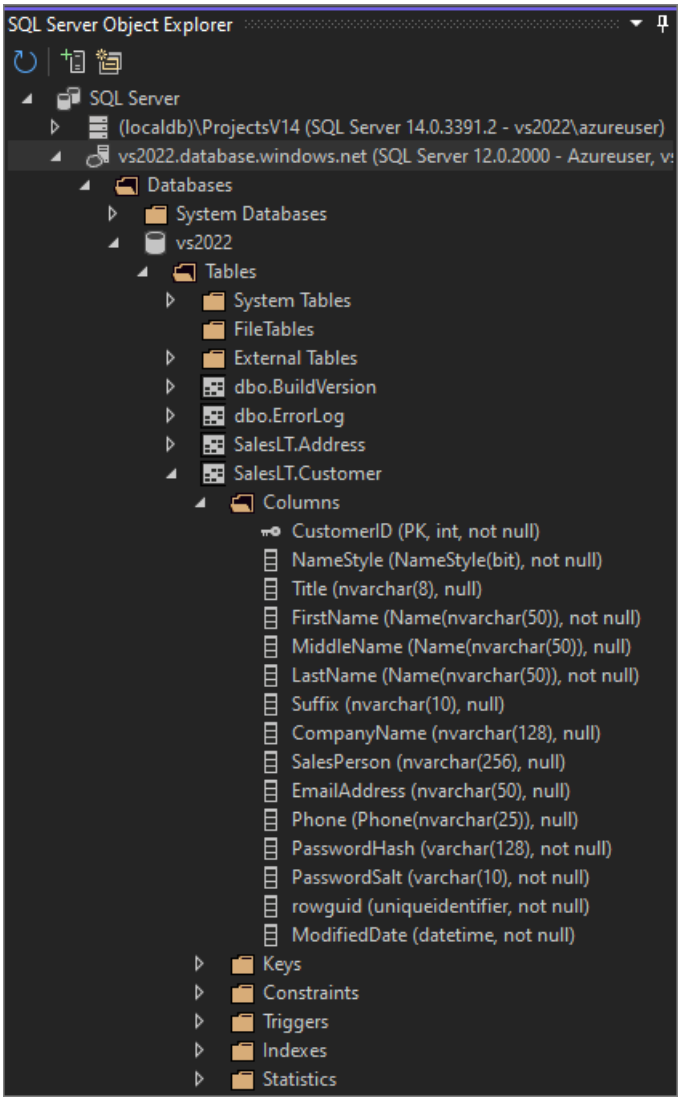
Connect to databases

Server Explorer helps you browse and manage server instances and assets locally, remotely, and on Azure, Microsoft 365, Salesforce.com, and websites. To open **Server Explorer**, choose **View** > **Server Explorer**. For more information on using Server Explorer, see Add new connections.

SQL Server Object Explorer provides a view of your database objects, similar to SQL Server Management Studio. With SQL Server Object Explorer, you can do light-duty database administration and design work. Examples include editing table data, comparing schemas, and executing queries by using contextual menus.



To open **SQL Server Object Explorer**, select its icon at the top of the **Server Explorer** window, or select **View** > **SQL Server Object Explorer** from the Visual Studio top menu.



SQL Server Data Tools (SSDT) is a powerful development environment for SQL Server, Azure SQL Database, and Azure SQL Data Warehouse. With SSDT, you can build, debug, maintain, and refactor databases. You can work with a database project, or directly with a connected database instance on- or off-premises. To get SSDT, use the Visual Studio Installer to install the **Data storage and processing** workload.

Debug, test, and improve your code

When you write code, you should run it and test it for bugs and performance. With Visual Studio's debugging system, you can debug code running in your local project, on a remote device, or on a device emulator. Step through code one statement at a time, and inspect variables as you go. Or set breakpoints that are only hit when a specified condition is true. You can manage debug options in the code editor itself, so you don't have to leave your code.

For more information about debugging in Visual Studio, see First look at the debugger.

To improve app performance, check out the Visual Studio profiling feature.

Visual Studio offers testing options like unit testing, Live Unit Testing, IntelliTest, and load and performance testing. Visual Studio also has advanced code analysis capabilities to find design, security, and other flaws.

Deploy your finished application

Visual Studio has tools to deploy your app to users or customers through the Microsoft Store, a SharePoint site, or InstallShield or Windows Installer technologies. You can access all these options through the Visual Studio IDE. For more information, see Deploy applications, services, and components.

Manage your source code and collaborate with others

In Visual Studio, you can manage your source code in Git repos hosted by any provider, including GitHub. You can also browse for an Azure DevOps Server to connect to, too.

For full details, see the Git experience in Visual Studio page and the Visual Studio version control documentation navigation page. And, for a step-by-step tutorial on how to connect to a Git or Azure DevOps repository by using Visual Studio, see the Open a project from a repo page.

> **TIP**
> We continue to build out the Git feature set and iterate on it based on your feedback. For more info about a recent feature update along with a link to survey where you can share your feedback on it, see the Multi-repo support in Visual Studio blog post.

How you open a project from a GitHub repo by using Visual Studio 2019 depends on which version you have. Specifically, if you've installed version **version 16.8** or later, there's a new, more fully integrated Git experience in Visual Studio available to you. For more information, see the Visual Studio version control documentation page.

And, for a step-by-step tutorial on how to connect to a Git or Azure DevOps repository by using Visual Studio, see the Open a project from a repo page.

To learn more about managing Git repos in Visual Studio by using **Team Explorer**, see Get started with Git and Azure Repos. To learn more the Visual Studio built-in source control features, see the Git features in Visual Studio blog post.

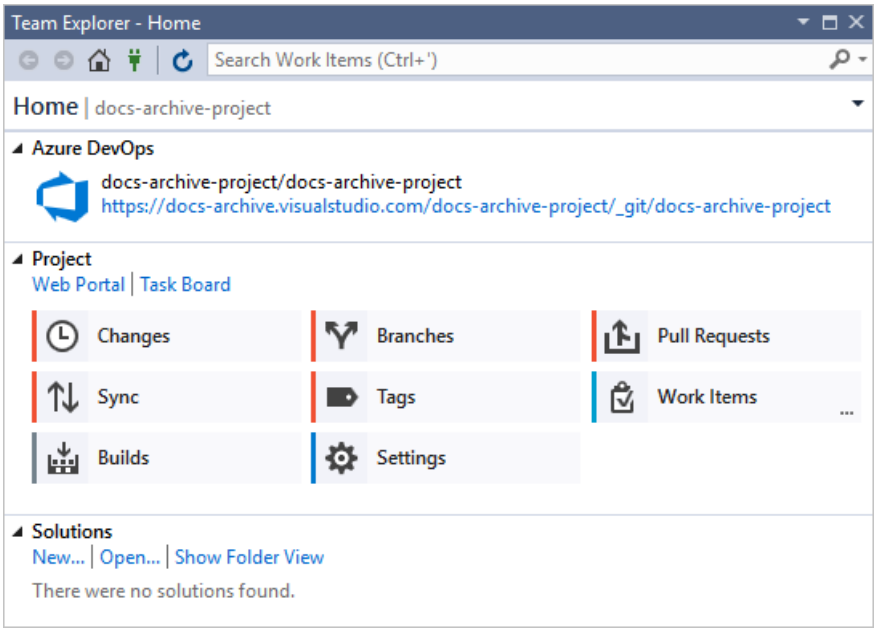
Azure DevOps Services is a suite of cloud-based services that plan, host, automate, and deploy software and promote team collaboration. DevOps Services supports both GitHub distributed version control and Team Foundation Version Control (TFVC) centralized version control. DevOps Services provides continuous build and

release (CI/CD) pipelines for code stored in version control systems. DevOps Services also supports Scrum, CMMI, and Agile development methodologies. You can use DevOps Services to manage code along with bugs and work items for your project.

Team Foundation Server (TFS) is the application lifecycle management hub for Visual Studio. It enables everyone involved with the development process to participate using a single solution. TFS is useful for managing heterogeneous teams and projects, too.

You can connect to an Azure DevOps organization or a Team Foundation Server on your network through the Visual Studio **Team Explorer** window. From the **Team Explorer** window, you can check code into or out of source control, manage work items, start builds, and access team rooms and workspaces. To open **Team Explorer**, use the search box, or select **View** > **Team Explorer**.

The following image shows the **Team Explorer** window for a solution that's hosted in Azure DevOps Services.



Azure DevOps is an application lifecycle management hub for Visual Studio. Azure DevOps enables everyone involved with the development process to participate using a single solution. Azure DevOps is also useful for managing heterogeneous teams and projects.

You can connect to an Azure DevOps organization or Azure DevOps Server on your network through the **Team Explorer** window in Visual Studio. From the **Team Explorer** window, you can check code into or out of source control, manage work items, start builds, and access team rooms and workspaces. To open **Team Explorer**, use the search box, or select **View** > **Team Explorer**.

You can also automate your build process to build code that developers check into version control. For example, you can build one or more projects nightly, or every time certain code is checked in. For more information, see Azure Pipelines.

Next steps

- If Visual Studio doesn't have the exact functionality you need, you can add it. Personalize the IDE based on your workflow and style, add support for external tools that aren't integrated with Visual Studio, and modify existing functionality to increase your productivity. For the latest version of the Visual Studio Extensibility Tools (VS SDK), see Visual Studio SDK.
- You can use the .NET Compiler Platform *Roslyn* to write your own code analyzers and code generators. Find everything you need at Roslyn.
- Find existing extensions for Visual Studio created by Microsoft developers and the Visual Studio

development community.

- To learn more about extending Visual Studio, see Extend Visual Studio IDE.

See also

- Visual Studio IDE overview
- What's new in Visual Studio 2017
- What's new in Visual Studio 2019
- What's new in Visual Studio 2022

Tutorial: Create a simple C# console app in Visual Studio (part 1 of 2)

10/28/2021 • 15 minutes to read • Edit Online

In this tutorial, you use Visual Studio to create and run a C# console app, and explore some features of the Visual Studio integrated development environment (IDE). This tutorial is part 1 of a two-part tutorial series.

In this tutorial, you:

- Create a Visual Studio project.
- Create a C# console app.
- Debug your app.
- Close your app.
- Inspect your complete code.

In part 2, you extend this app to add more projects, learn debugging tricks, and reference third-party packages.

Prerequisites

You must have Visual Studio installed.

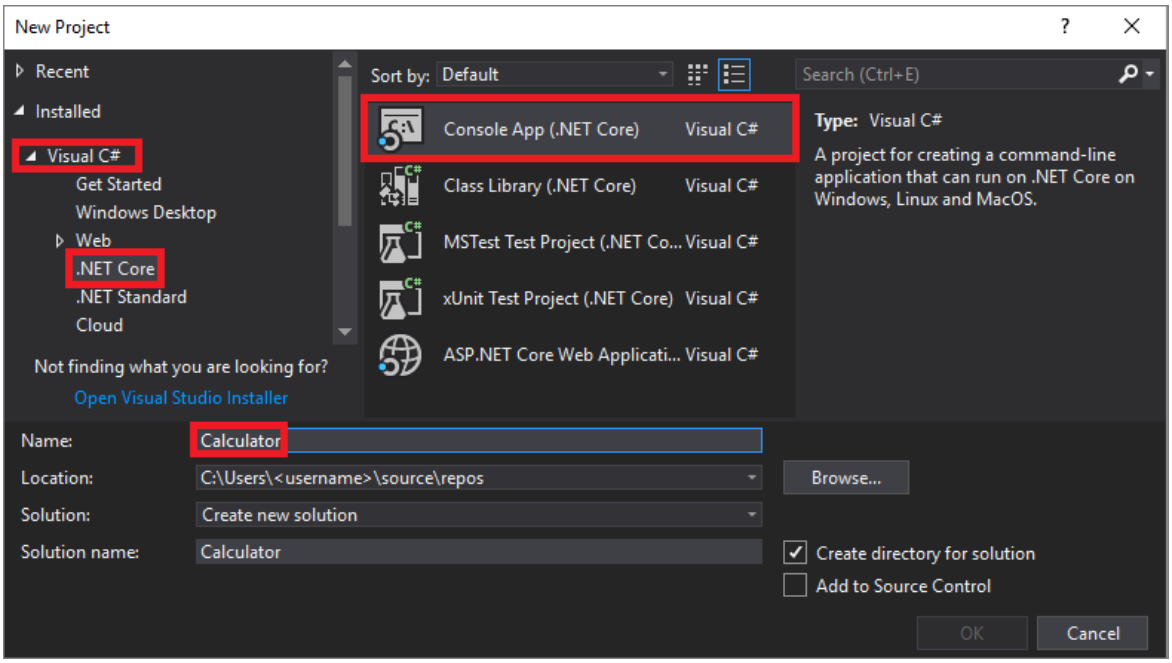
If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

Create a project

To start, create a C# application project. The project type comes with all the template files you need.

1. Open Visual Studio 2017.
2. From the top menu bar, choose **File** > **New** > **Project**. (Alternatively, press **Ctrl**+**Shift**+**N**).
3. In the left pane of the **New Project** dialog box, expand **C#**, and then choose **.NET Core**. In the middle pane, choose **Console App (.NET Core)**. Then name the file *Calculator*.

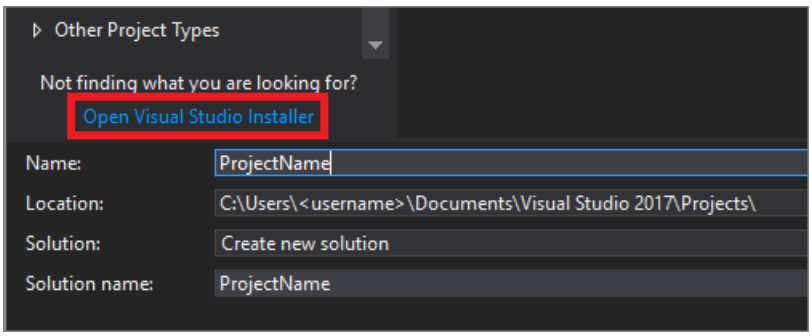


Add a workload (optional)

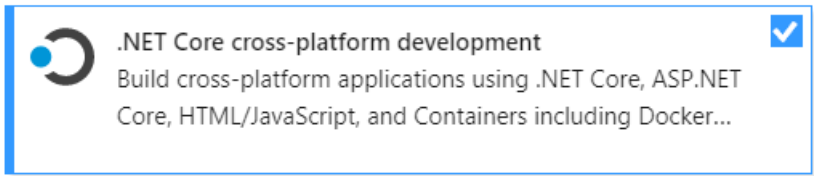
If you don't see the **Console App (.NET Core)** project template, you can get it by adding the **.NET Core cross-platform development** workload. Here's how.

Option 1: Use the New Project dialog box

1. Choose the **Open Visual Studio Installer** link in the left pane of the **New Project** dialog box.



2. The Visual Studio Installer launches. Choose the **.NET Core cross-platform development** workload, and then choose **Modify**.

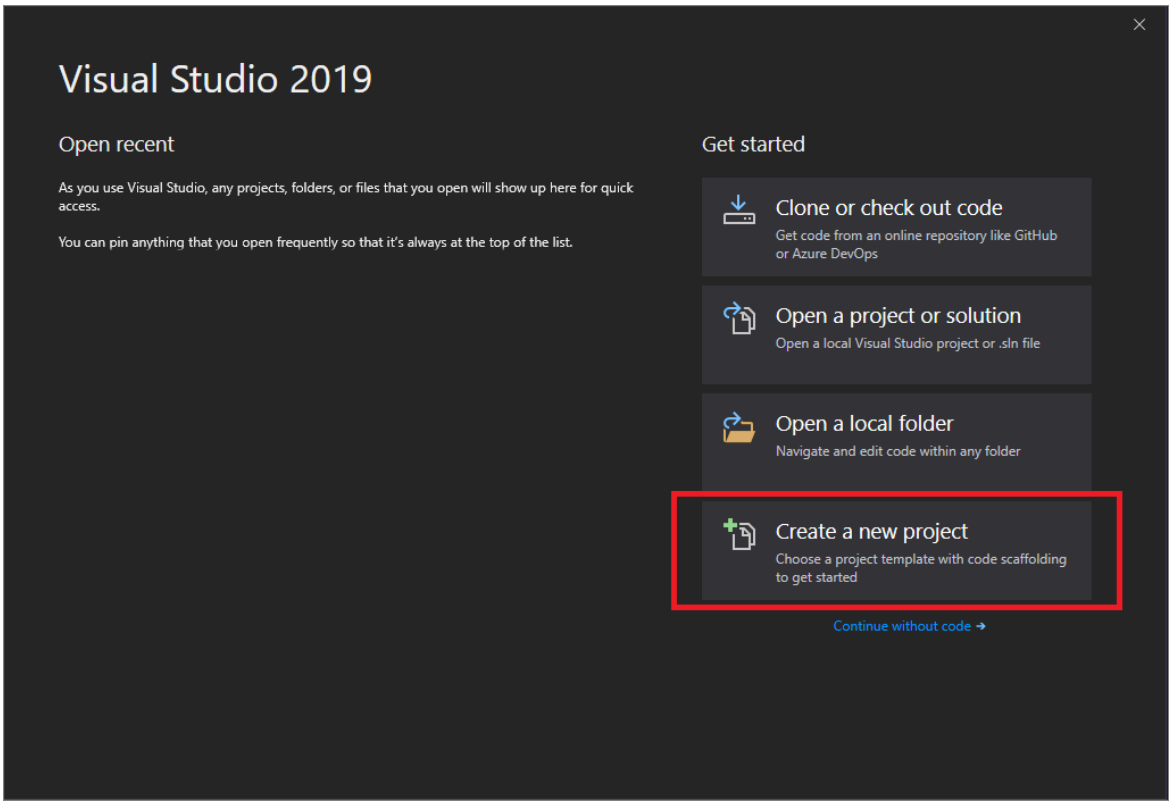


Option 2: Use the Tools menu bar

1. Cancel out of the **New Project** dialog box and from the top menu bar, choose **Tools** > **Get Tools and Features**.

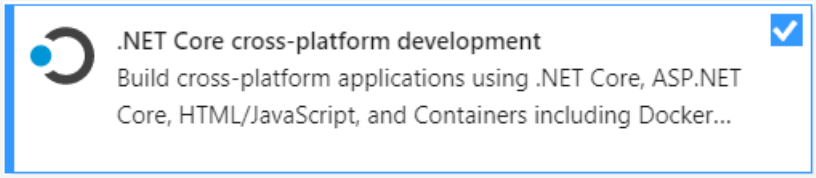
2. The Visual Studio Installer launches. Choose the **.NET Core cross-platform development** workload, and then choose **Modify**.

1. Open Visual Studio, and choose **Create a new project** in the Start window.

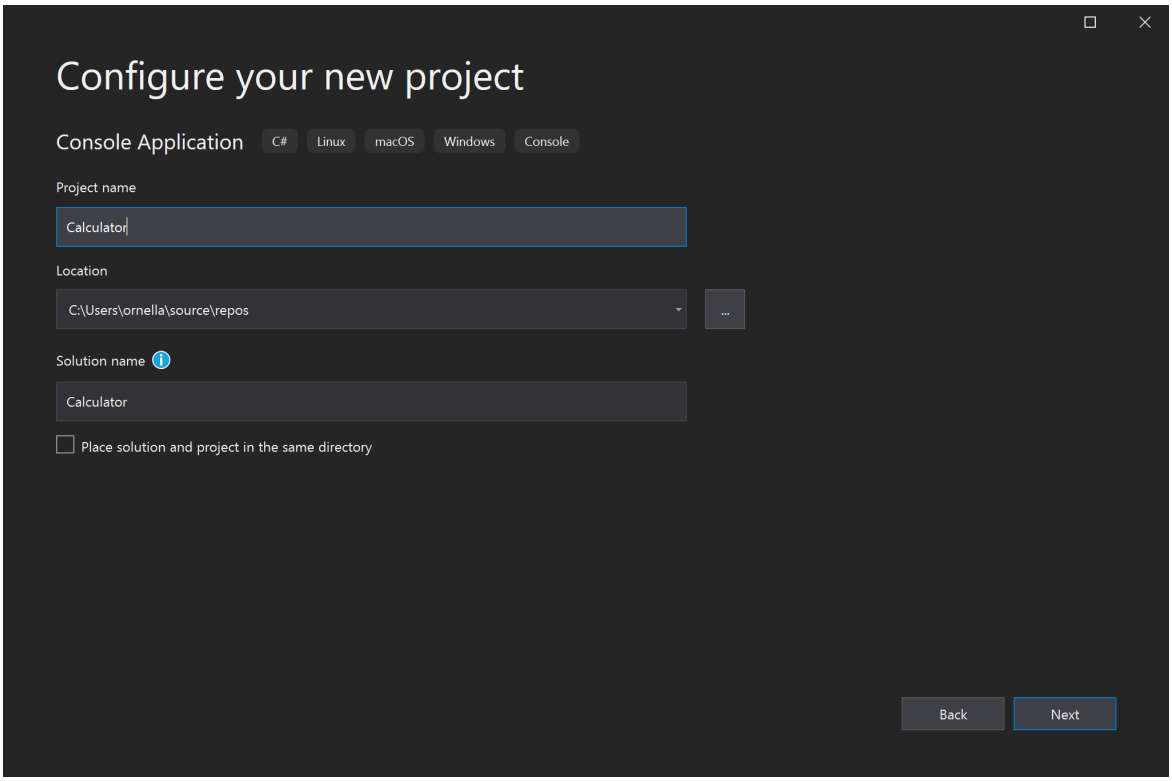


2. In the **Create a new project** window, choose **C#** from the Language list. Next, choose **Windows** from the Platform list and **Console** from the project types list.

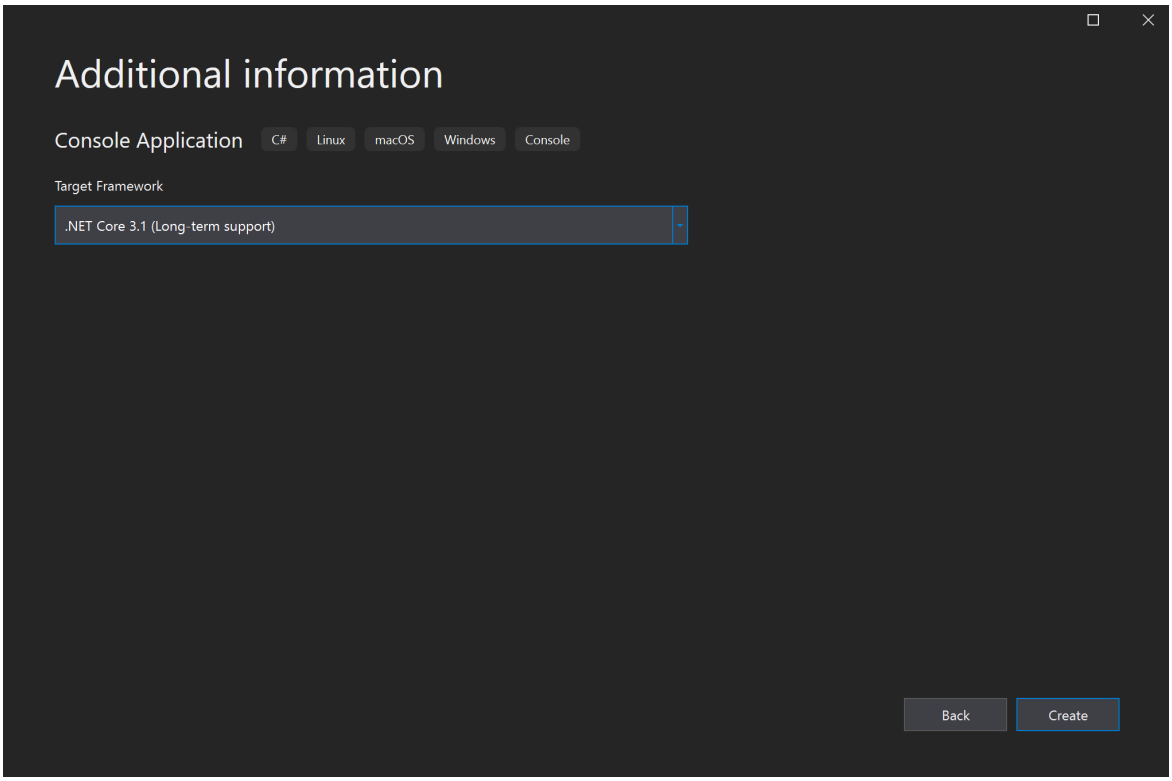
   After you apply the language, platform, and project type filters, choose the **Console Application** template, and then select **Next**.

> **NOTE**
>
> If you don't see the **Console Application** template, select **Install more tools and features**.
>
> 
>
>
> Then, in the Visual Studio Installer, choose the **.NET Core cross-platform development** workload.
>
> 
>
>
> After that, choose the **Modify** button in the Visual Studio Installer. You might be prompted to save your work; if so, do so. Next, choose **Continue** to install the workload. Then, return to step 2 in this "Create a project" procedure.

3. In the **Configure your new project** window, type or enter *Calculator* in the **Project name** box. Then, choose **Next**.

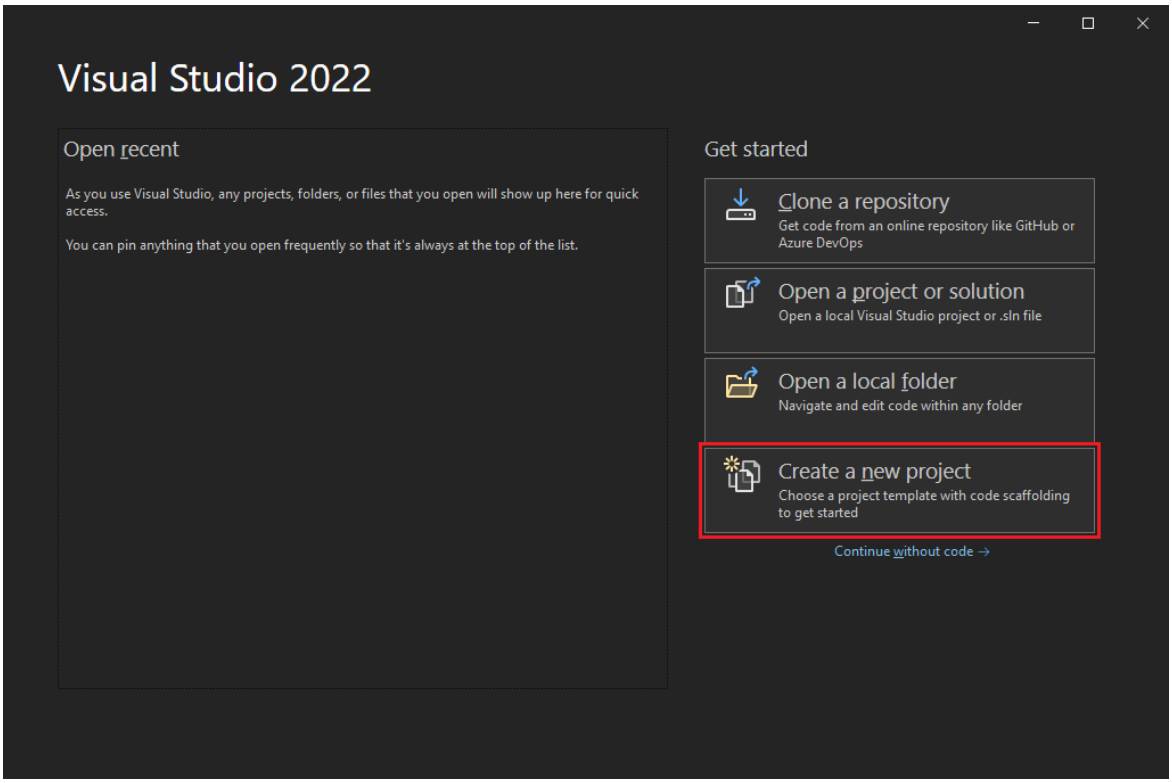


4. In the **Additional information** window, **.NET Core 3.1** should already be selected for your target framework. If not, select **.NET Core 3.1**. Then, choose **Create**.



Visual Studio opens your new project, which includes default "Hello World" code.

1. Open Visual Studio, and choose **Create a new project** in the Start window.

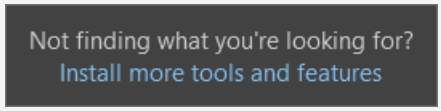


2. In the **Create a new project** window, select **All languages**, and then choose **C#** from the dropdown list. Choose **Windows** from the **All platforms** list, and choose **Console** from the **All project types** list.

   After you apply the language, platform, and project type filters, choose the **Console Application** template, and then select **Next**.

   NOTE

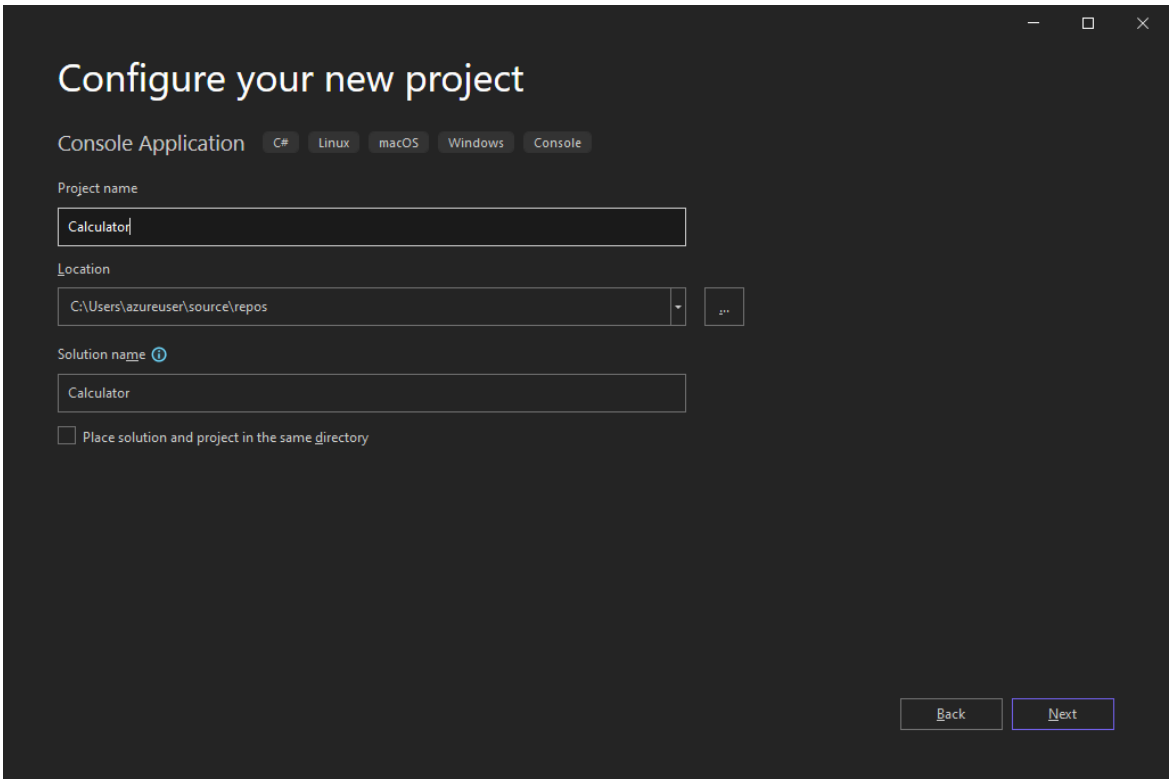
   If you don't see the **Console Application** template, select **Install more tools and features**.

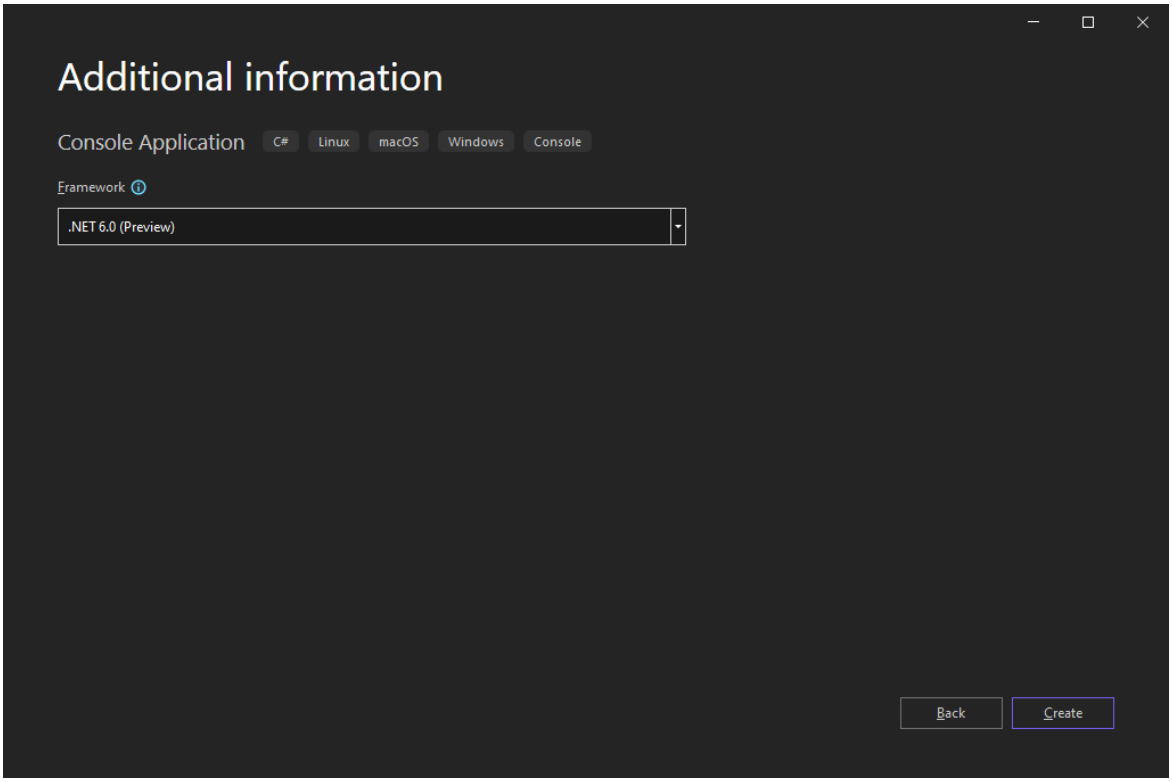
   In the Visual Studio Installer, choose the **.NET desktop development** workload, and then select **Modify**.

3. In the **Configure your new project** window, type or enter *Calculator* in the **Project name** box, and then select **Next**.



4. In the **Additional information** window, **.NET 6.0** should already be selected for your target framework. Select **Create**.



Visual Studio opens your new project, which includes default "Hello World" code.

Create the app

In this section, you:

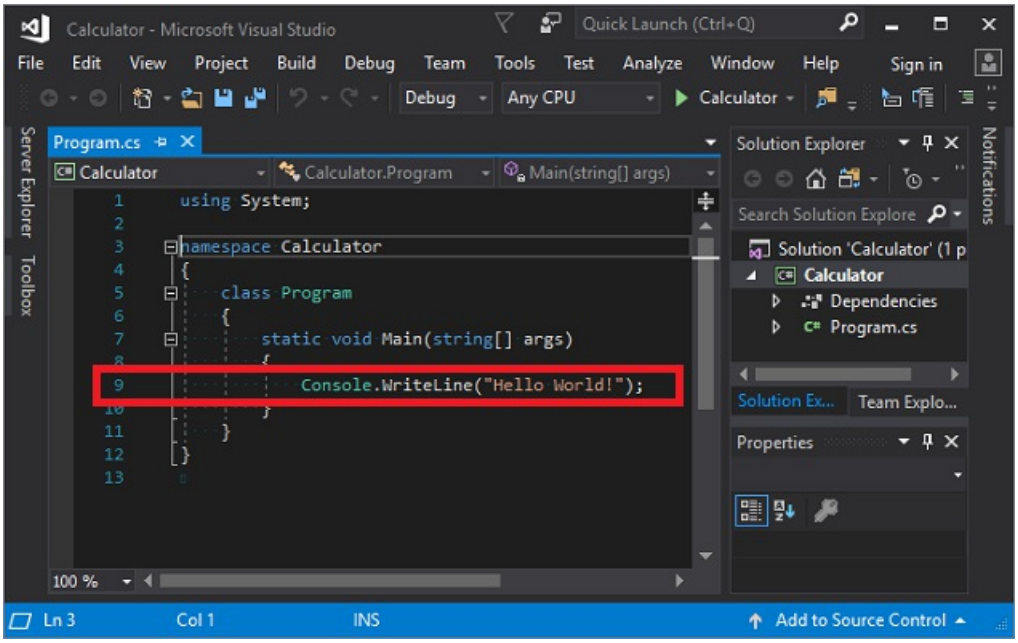
- Explore some basic integer math in C#.
- Add code to create a basic calculator app.
- Debug the app to find and fix errors.

- Refine the code to make it more efficient.

Explore integer math

Start with some basic integer math in C#.

1. In the code editor, delete the default "Hello World" code.



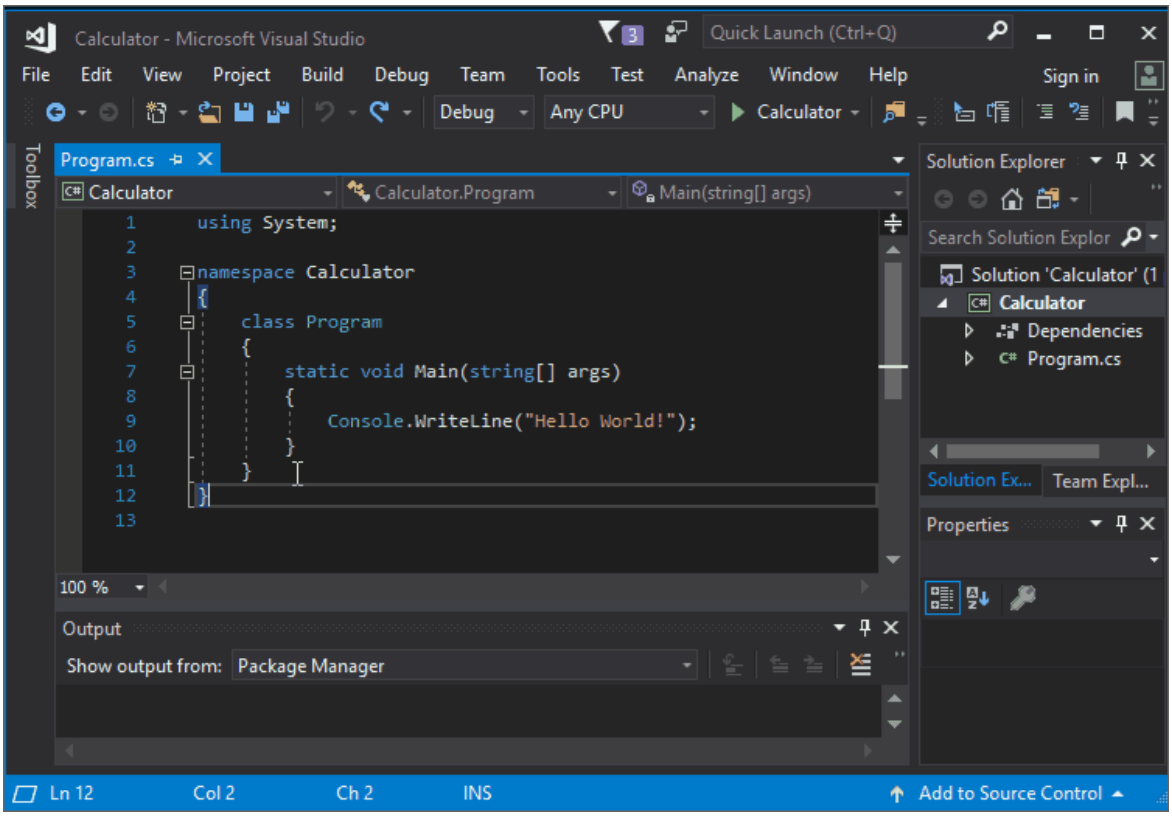
   Specifically, delete the line that says, `Console.WriteLine("Hello World!");`.

2. In its place, type the following code:

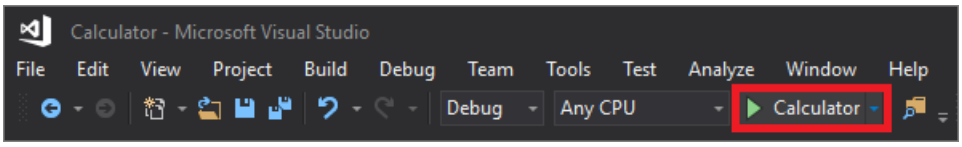
```
int a = 42;
int b = 119;
int c = a + b;
Console.WriteLine(c);
Console.ReadKey();
```

   Notice that when you do so, the IntelliSense feature in Visual Studio offers you the option to autocomplete the entry.

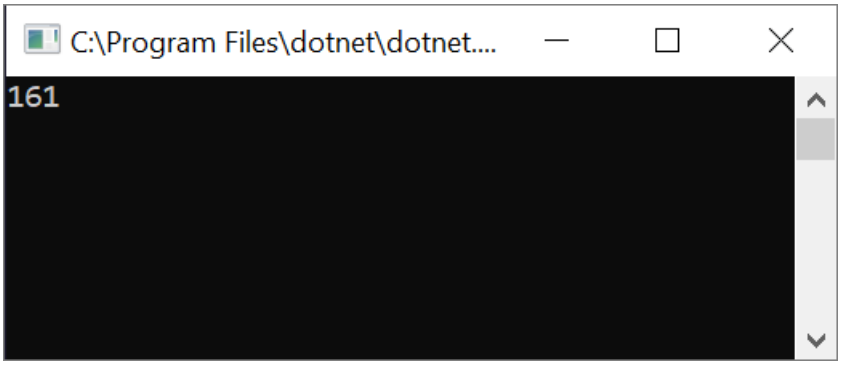
> **NOTE**
>
> The following animation isn't intended to duplicate the preceding code. It's intended only to show how the autocomplete feature works.



3. Choose the green **Start** button next to **Calculator** to build and run your program, or press **F5**.



A console window opens that reveals the sum of 42 + 119, which is **161**.

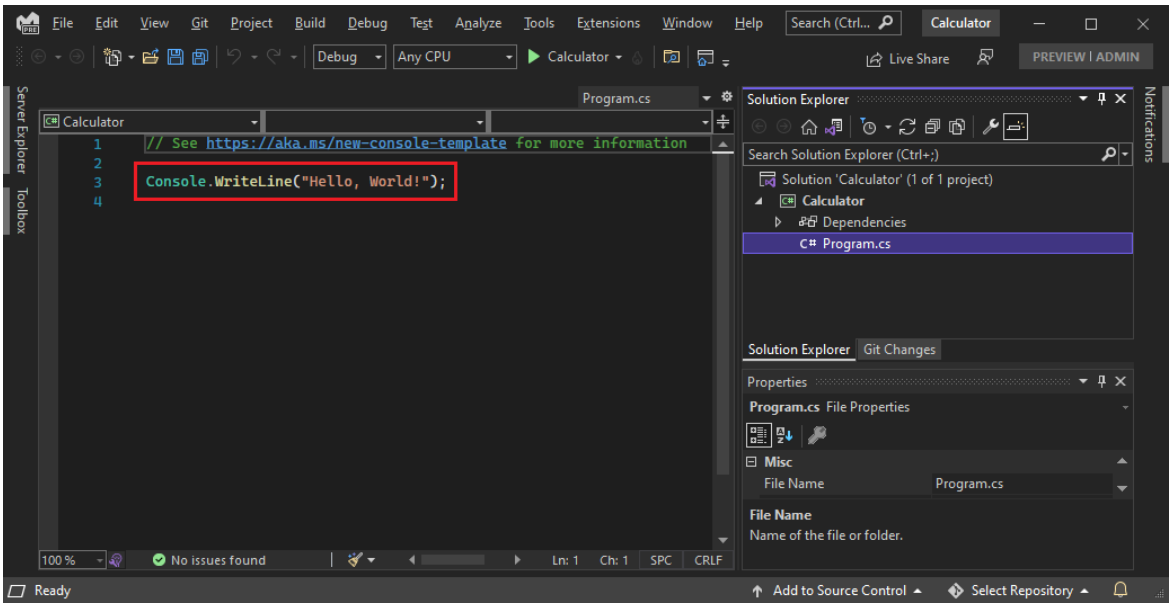


4. **(Optional)** You can change the operator to change the result. For example, you can change the `+` operator in the `int c = a + b;` line of code to `-` for subtraction, `*` for multiplication, or `/` for division. Then, when you run the program, the result changes, too.

5. Close the console window.

1. In **Solution Explorer**, in the right pane, select **Program.cs** to display the file in the code editor

2. In the code editor, replace the default "Hello World" code that says `Console.WriteLine("Hello World!");`.

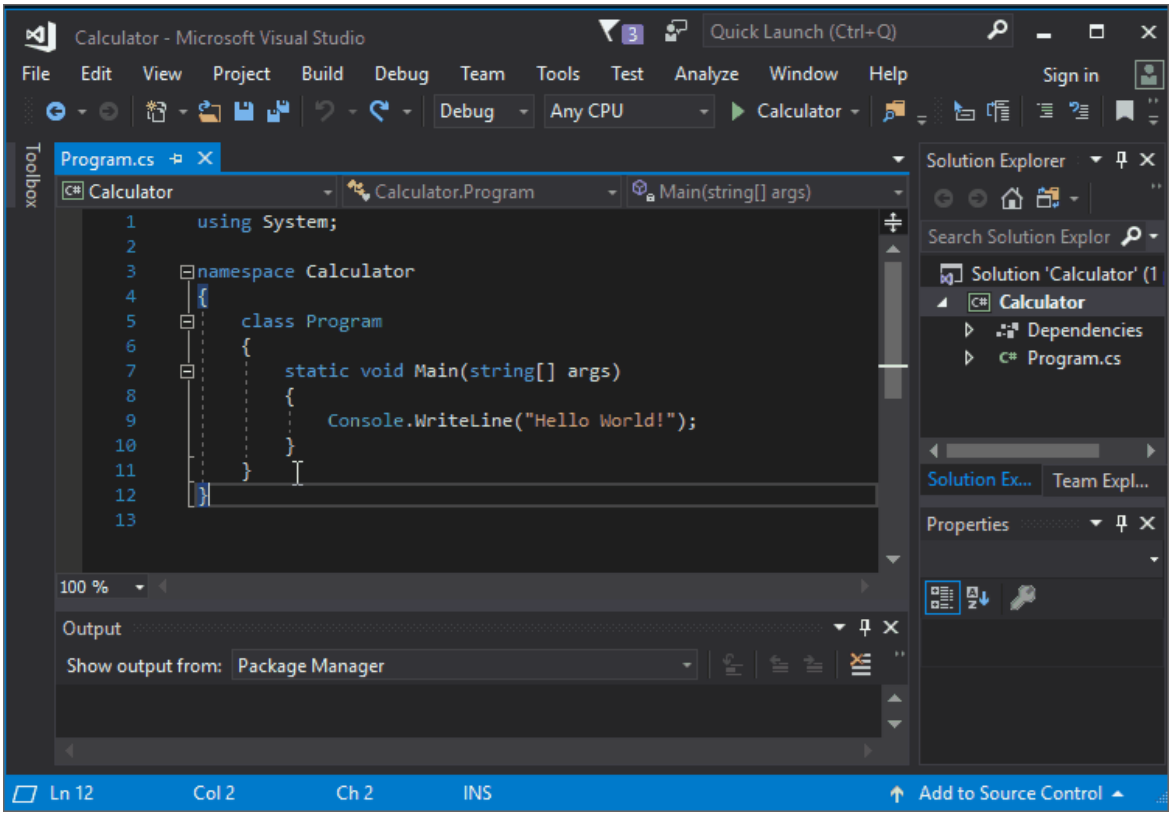


Replace the line with the following code:

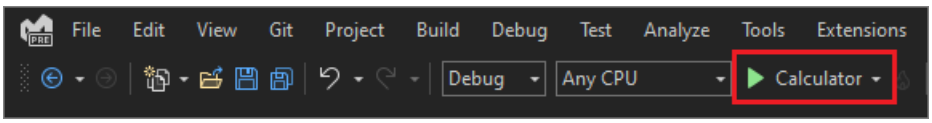
```
int a = 42;
int b = 119;
int c = a + b;
Console.WriteLine(c);
Console.ReadKey();
```

If you type the code, the Visual Studio IntelliSense feature offers you the option to autocomplete the entry.

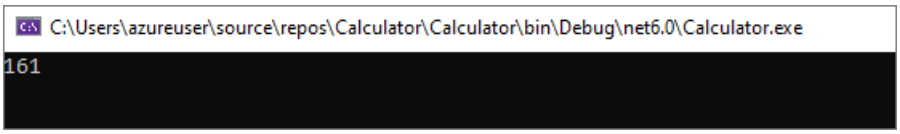
> **NOTE**
>
> The following animation isn't intended to demonstrate the preceding code, but only to show how IntelliSense works.



3. To build and run your app, press **F5**, or select the green arrow next to the name **Calculator** in the top toolbar.



A console window opens that shows the sum of 42 + 119, which is **161**.



4. Close the console window.

5. Optionally, you can change the operator to change the result. For example, you can change the `+` operator in the `int c = a + b;` line of code to `-` for subtraction, `*` for multiplication, or `/` for division. When you run the app, the result changes accordingly.

Add code to create a calculator

Continue by adding a more complex set of calculator code to your project.

1. In the code editor, replace all the code in *program.cs* with the following new code:

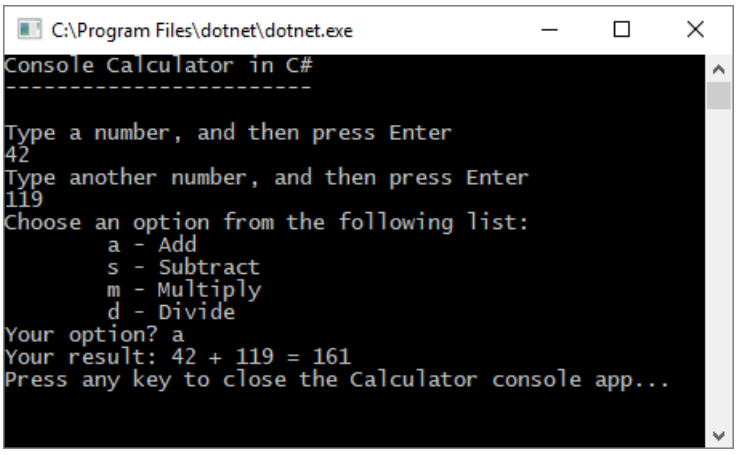
```
using System;

namespace Calculator
{
    class Program
    {
        static void Main(string[] args)
        {
            // Declare variables and then initialize to zero.
            int num1 = 0; int num2 = 0;

            // Display title as the C# console calculator app.
            Console.WriteLine("Console Calculator in C#\r");
            Console.WriteLine("------------------------\n");

            // Ask the user to type the first number.
            Console.WriteLine("Type a number, and then press Enter");
            num1 = Convert.ToInt32(Console.ReadLine());

            // Ask the user to type the second number.
            Console.WriteLine("Type another number, and then press Enter");
            num2 = Convert.ToInt32(Console.ReadLine());

            // Ask the user to choose an option.
            Console.WriteLine("Choose an option from the following list:");
            Console.WriteLine("\ta - Add");
            Console.WriteLine("\ts - Subtract");
            Console.WriteLine("\tm - Multiply");
            Console.WriteLine("\td - Divide");
            Console.Write("Your option? ");

            // Use a switch statement to do the math.
            switch (Console.ReadLine())
            {
                case "a":
                    Console.WriteLine($"Your result: {num1} + {num2} = " + (num1 + num2));
                    break;
                case "s":
                    Console.WriteLine($"Your result: {num1} - {num2} = " + (num1 - num2));
                    break;
                case "m":
                    Console.WriteLine($"Your result: {num1} * {num2} = " + (num1 * num2));
                    break;
                case "d":
                    Console.WriteLine($"Your result: {num1} / {num2} = " + (num1 / num2));
                    break;
            }
            // Wait for the user to respond before closing.
            Console.Write("Press any key to close the Calculator console app...");
            Console.ReadKey();
        }
    }
}
```

2. Select the **Calculator** button or press **F5** to run your app.

   A console window opens.

3. In the console window, follow the prompts to add the numbers **42** and **119** together.

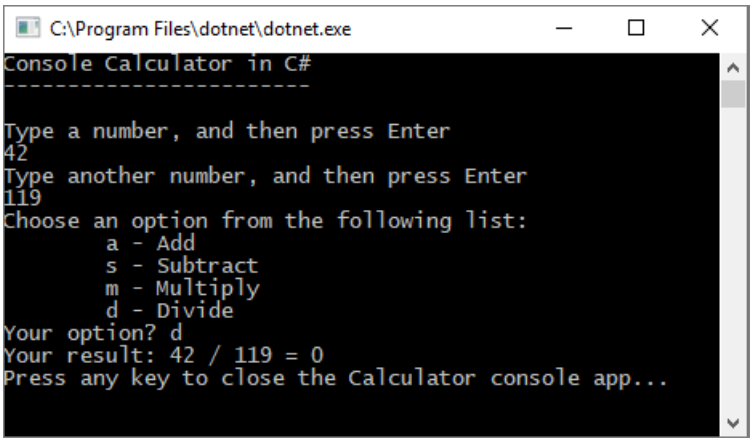
   Your app should look similar to the following screenshot:



Add decimal functionality

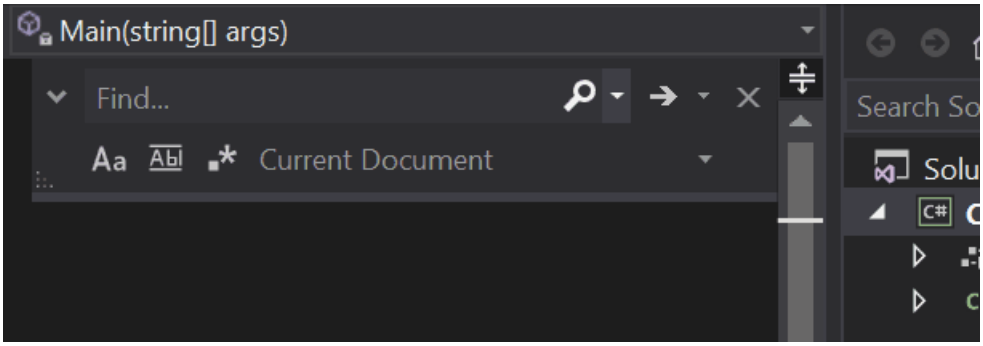
Now, tweak the code to add more functionality.

The current calculator app only accepts and returns whole numbers. For example, if you run the app and divide the number 42 by the number 119, your result is zero, which isn't exact.



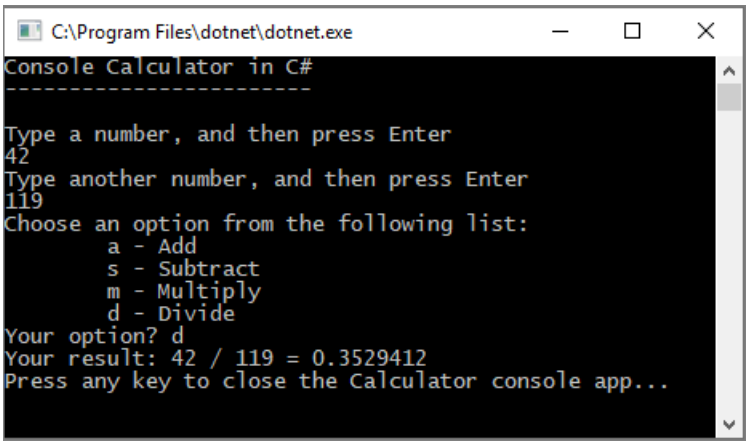
To fix the code to improve precision by handling decimals:

1. From *program.cs* in the Visual Studio editor, press **Ctrl**+**H** to open the **Find and Replace** control.
2. Type *int* in the control, and type *float* in the **Replace** field.
3. Select the icons for **Match case** and **Match whole word** in the control, or press **Alt**+**C** and **Alt**+**W**.
4. Select the **Replace all** icon or press **Alt**+**A** to run the search and replace.



5. Run your calculator app again, and divide the number **42** by the number **119**.

   The app now returns a decimal number instead of zero.

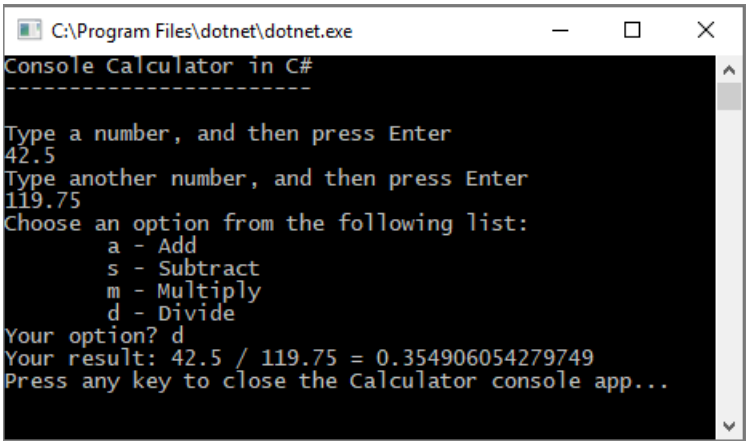


Now the app can produce decimal results. Make a few more tweaks to the code so the app can calculate decimals too.

1. Use the **Find and Replace** control to change each instance of the `float` variable to `double`, and to change each instance of the `Convert.ToInt32` method to `Convert.ToDouble`.

2. Run your calculator app, and divide the number **42.5** by the number **119.75**.

   The app now accepts decimal values, and returns a longer decimal numeral as its result.



   In the Revise the code section, you reduce the number of decimal places in the results.

Debug the app

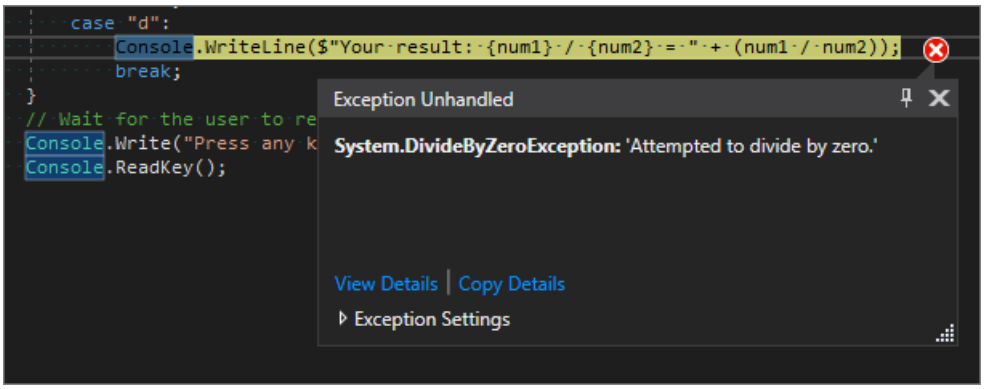
You've improved your basic calculator app, but your app doesn't yet handle exceptions, such as user input errors. For example, if users try to divide by zero, or enter an unexpected character, the app might stop working, return an error, or return an unexpected nonnumeric result.

Let's walk through a few common user input errors, locate them in the debugger if they appear there, and fix them in the code.

> **TIP**
> For more information about the debugger and how it works, see First look at the Visual Studio debugger.

Fix the "divide by zero" error

If you try to divide a number by zero, the console app might freeze, and then shows you what's wrong in the code editor.



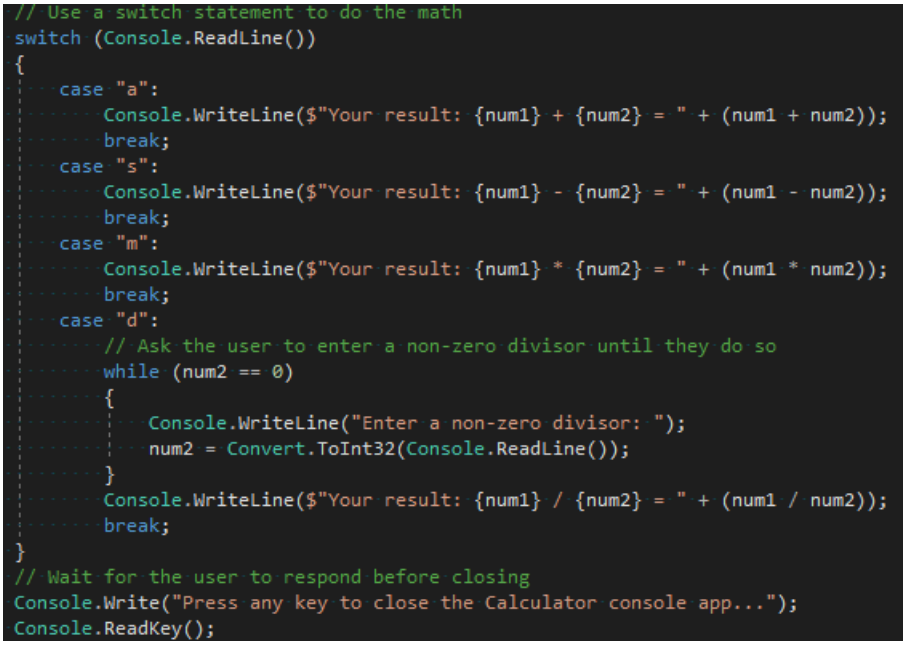
> **NOTE**
>
> Sometimes the app doesn't freeze, and the debugger doesn't show a divide-by-zero error. Instead, the app might return an unexpected nonnumeric result, such as an infinity symbol. The following code fix still applies.

To change the code to handle this error:

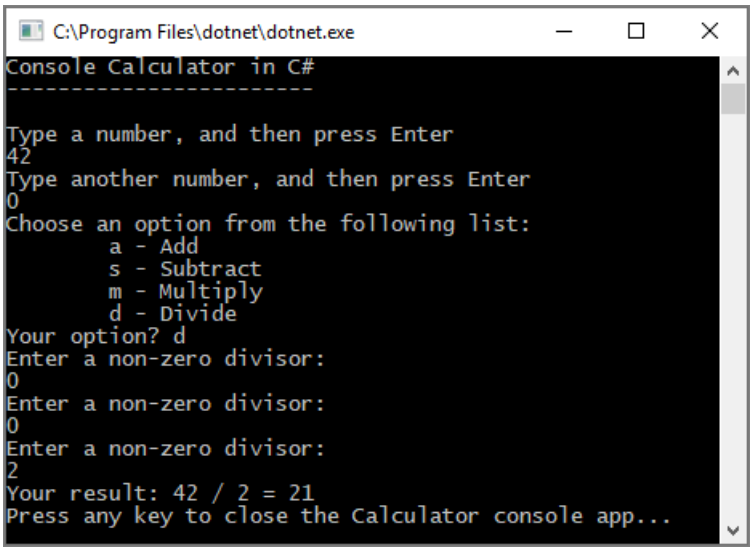
1. In *program.cs*, replace the code between `case "d":` and the comment that says `// Wait for the user to respond before closing` with the following code:

   ```
           // Ask the user to enter a non-zero divisor until they do so.
               while (num2 == 0)
               {
                   Console.WriteLine("Enter a non-zero divisor: ");
                   num2 = Convert.ToInt32(Console.ReadLine());
               }
               Console.WriteLine($"Your result: {num1} / {num2} = " + (num1 / num2));
               break;
       }
   ```

   After you replace the code, the section with the `switch` statement should look similar to the following screenshot:

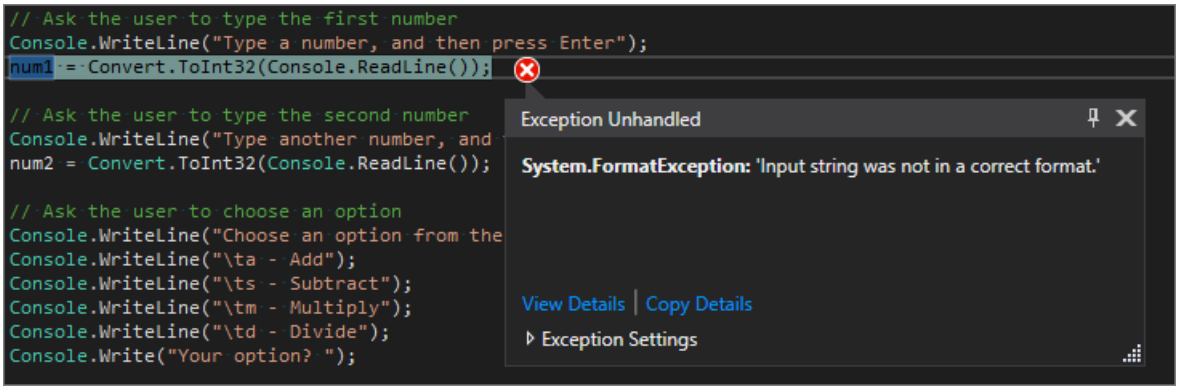


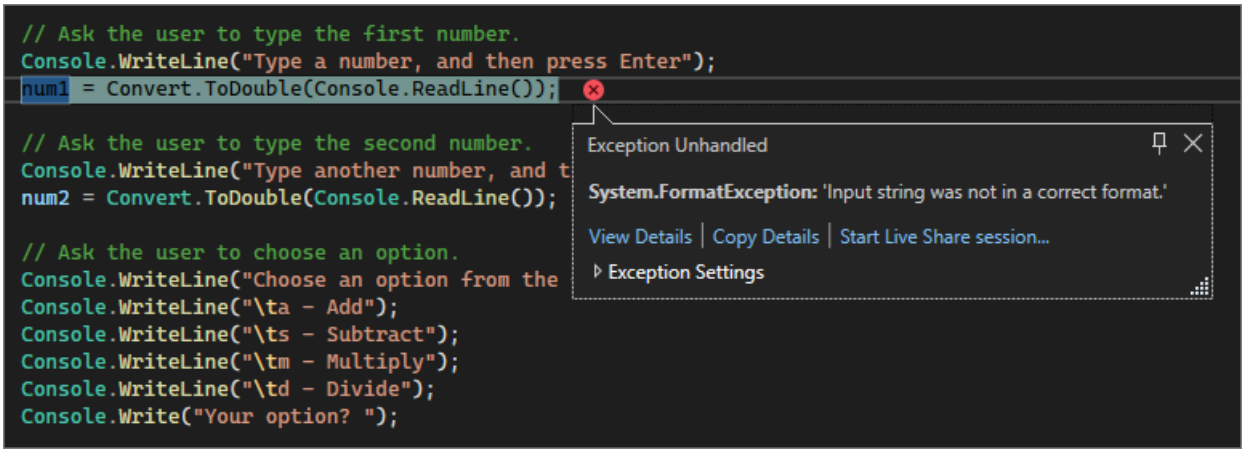
Now, when you divide any number by zero, the app asks for another number, and keeps asking until you provide a nonzero number.



Fix the "format" error

If you enter an alphabetic character when the app expects a numeric character, the app freezes. Visual Studio shows you what's wrong in the code editor.





To prevent this exception, you can refactor the code you've previously entered.

Revise the code

Rather than rely on the `program` class to handle all the code, you can divide your app into two classes: `Calculator` and `Program`.

The `Calculator` class handles the bulk of the calculation work, and the `Program` class handles the user interface and error-handling work.

Let's get started.

1. In *program.cs*, delete everything in the `Calculator` namespace between its opening and closing braces:

```
using System;

namespace Calculator
{

}
```

2. Between the braces, add the following new `Calculator` class:

```
class Calculator
{
    public static double DoOperation(double num1, double num2, string op)
    {
        double result = double.NaN; // Default value is "not-a-number" if an operation, such as
division, could result in an error.

        // Use a switch statement to do the math.
        switch (op)
        {
            case "a":
                result = num1 + num2;
                break;
            case "s":
                result = num1 - num2;
                break;
            case "m":
                result = num1 * num2;
                break;
            case "d":
                // Ask the user to enter a non-zero divisor.
                if (num2 != 0)
                {
                    result = num1 / num2;
                }
                break;
            // Return text for an incorrect option entry.
            default:
                break;
        }
        return result;
    }
}
```

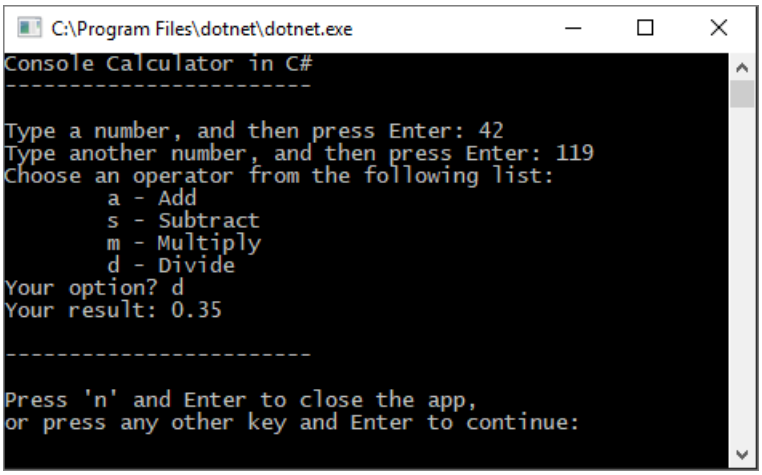
3. Also add a new `Program` class, as follows:

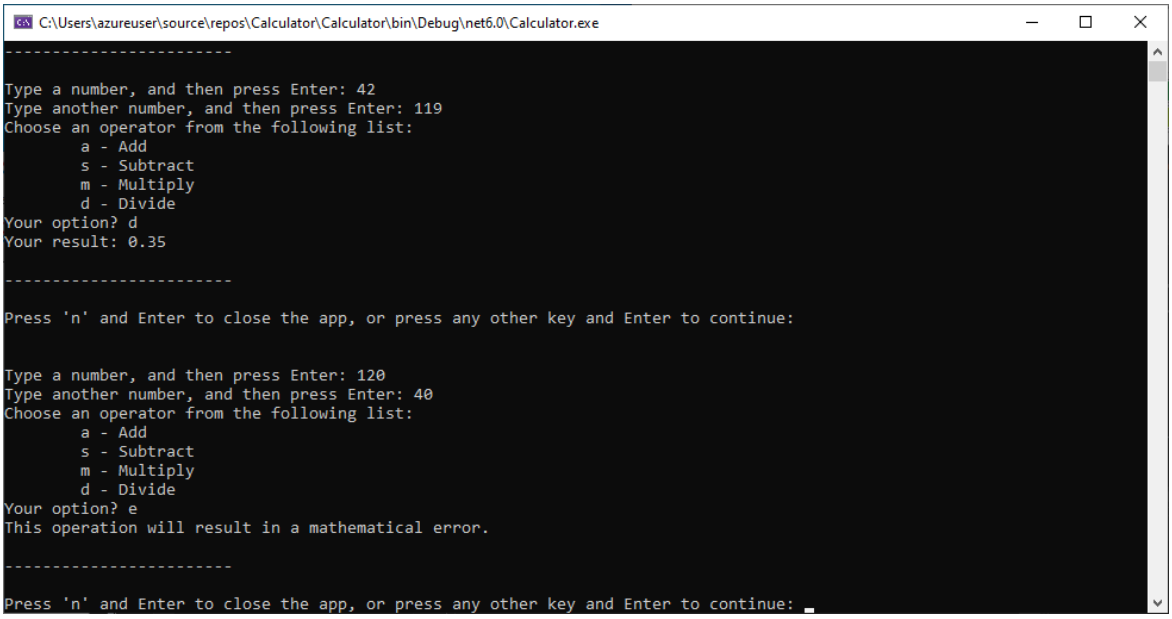
```
class Program
{
    static void Main(string[] args)
    {
        bool endApp = false;
        // Display title as the C# console calculator app.
        Console.WriteLine("Console Calculator in C#\r");
        Console.WriteLine("------------------------\n");

        while (!endApp)
        {
            // Declare variables and set to empty.
            string numInput1 = "";
            string numInput2 = "";
            double result = 0;

            // Ask the user to type the first number.
            Console.Write("Type a number, and then press Enter: ");
```

```
            numInput1 = Console.ReadLine();

            double cleanNum1 = 0;
            while (!double.TryParse(numInput1, out cleanNum1))
            {
                Console.Write("This is not valid input. Please enter an integer value: ");
                numInput1 = Console.ReadLine();
            }

            // Ask the user to type the second number.
            Console.Write("Type another number, and then press Enter: ");
            numInput2 = Console.ReadLine();

            double cleanNum2 = 0;
            while (!double.TryParse(numInput2, out cleanNum2))
            {
                Console.Write("This is not valid input. Please enter an integer value: ");
                numInput2 = Console.ReadLine();
            }

            // Ask the user to choose an operator.
            Console.WriteLine("Choose an operator from the following list:");
            Console.WriteLine("\ta - Add");
            Console.WriteLine("\ts - Subtract");
            Console.WriteLine("\tm - Multiply");
            Console.WriteLine("\td - Divide");
            Console.Write("Your option? ");

            string op = Console.ReadLine();

            try
            {
                result = Calculator.DoOperation(cleanNum1, cleanNum2, op);
                if (double.IsNaN(result))
                {
                    Console.WriteLine("This operation will result in a mathematical error.\n");
                }
                else Console.WriteLine("Your result: {0:0.##}\n", result);
            }
            catch (Exception e)
            {
                Console.WriteLine("Oh no! An exception occurred trying to do the math.\n - Details: "
+ e.Message);
            }

            Console.WriteLine("-----------------------\n");

            // Wait for the user to respond before closing.
            Console.Write("Press 'n' and Enter to close the app, or press any other key and Enter to
continue: ");
            if (Console.ReadLine() == "n") endApp = true;

            Console.WriteLine("\n"); // Friendly linespacing.
        }
        return;
    }
}
```

4. Select the **Calculator** button or press **F5** to run your app.

5. Follow the prompts and divide the number **42** by the number **119**. Your results should look similar to the following screenshot:

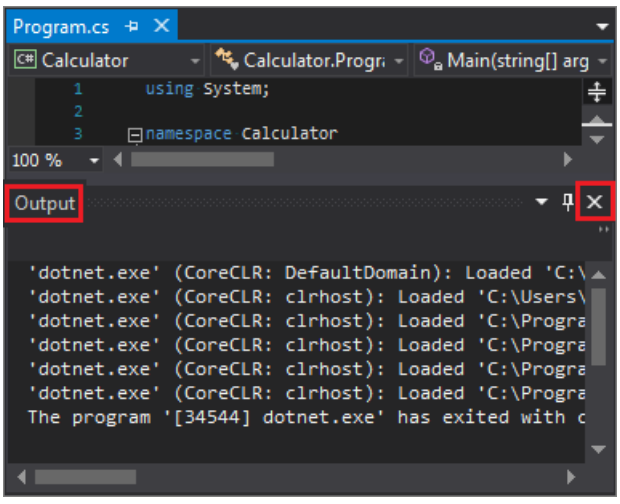




You can now enter more equations until you choose to close the console app. There are also fewer decimal places in the results. And if you enter an incorrect character, you get an appropriate error response.

Close the app

1. If you haven't already done so, close the Calculator app.
2. Close the **Output** pane in Visual Studio.



3. In Visual Studio, press **Ctrl**+**S** to save your app.

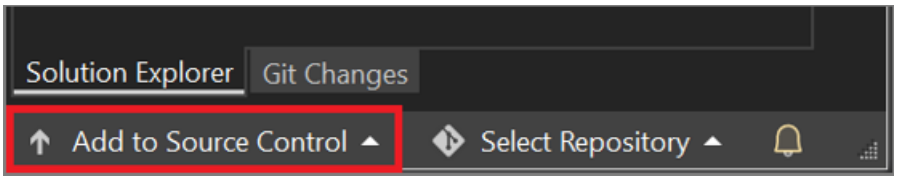
Add Git source control

Now that you've created an app, you might want to add it to a Git repository. We've got you covered. Visual Studio makes that process easy with Git tools you can use directly from the IDE.

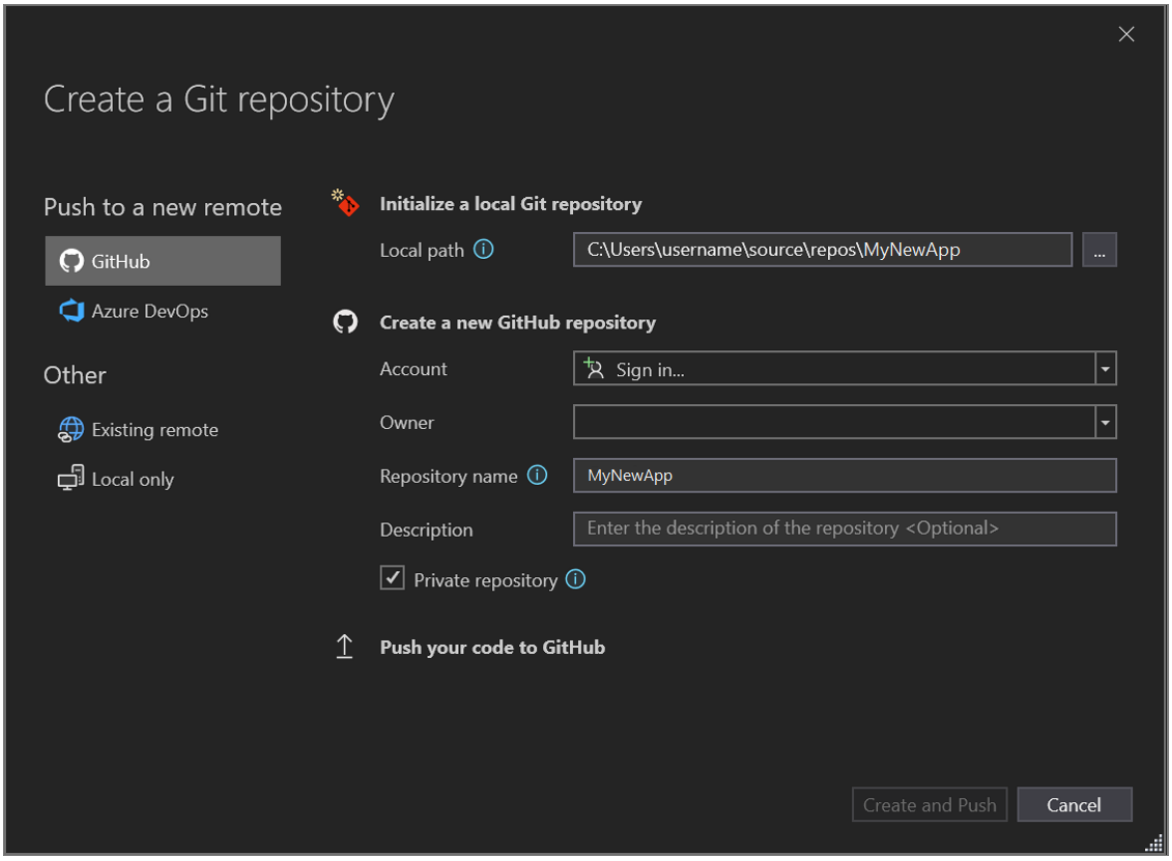
> **TIP**
> Git is the most widely used modern version control system, so whether you're a professional developer or you're learning how to code, Git can be very useful. If you're new to Git, the https://git-scm.com/ website is a good place to start. There, you can find cheat sheets, a popular online book, and Git Basics videos.

To associate your code with Git, you start by creating a new Git repository where your code is located. Here's how:

1. In the status bar at the bottom-right corner of Visual Studio, select **Add to Source Control**, and then select **Git**.



2. In the **Create a Git repository** dialog box, sign in to GitHub.

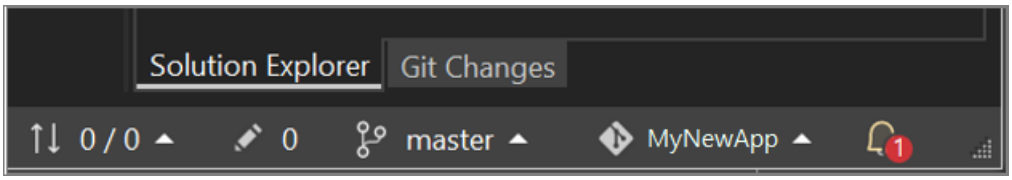


   The repository name auto-populates based on your folder location. By default, your new repository is private, which means you're the only one who can access it.

> **TIP**
> Whether your repository is public or private, it's best to have a remote backup of your code stored securely on GitHub. Even if you aren't working with a team, a remote repository makes your code available to you from any computer.

3. Select **Create and Push**.

   After you create your repository, you see status details in the status bar.



   The first icon with the arrows shows how many outgoing/incoming commits are in your current branch. You can use this icon to pull any incoming commits or push any outgoing commits. You can also choose to view these commits first. To do so, select the icon, and then select **View Outgoing/Incoming**.

   The second icon with the pencil shows the number of uncommitted changes to your code. You can select this icon to view those changes in the **Git Changes** window.

To learn more about how to use Git with your app, see the Visual Studio version control documentation.

Review: Code complete

In this tutorial, you made many changes to the calculator app. The app now handles computing resources more efficiently, and handles most user input errors.

Here's the complete code, all in one place:

```
using System;

namespace Calculator
{
    class Calculator
    {
        public static double DoOperation(double num1, double num2, string op)
        {
            double result = double.NaN; // Default value is "not-a-number" which we use if an operation,
such as division, could result in an error.

            // Use a switch statement to do the math.
            switch (op)
            {
                case "a":
                    result = num1 + num2;
                    break;
                case "s":
                    result = num1 - num2;
                    break;
                case "m":
                    result = num1 * num2;
                    break;
                case "d":
                    // Ask the user to enter a non-zero divisor.
                    if (num2 != 0)
                    {
                        result = num1 / num2;
                    }
```

```
                break;
            // Return text for an incorrect option entry.
            default:
                break;
        }
        return result;
    }
}

class Program
{
    static void Main(string[] args)
    {
        bool endApp = false;
        // Display title as the C# console calculator app.
        Console.WriteLine("Console Calculator in C#\r");
        Console.WriteLine("------------------------\n");

        while (!endApp)
        {
            // Declare variables and set to empty.
            string numInput1 = "";
            string numInput2 = "";
            double result = 0;

            // Ask the user to type the first number.
            Console.Write("Type a number, and then press Enter: ");
            numInput1 = Console.ReadLine();

            double cleanNum1 = 0;
            while (!double.TryParse(numInput1, out cleanNum1))
            {
                Console.Write("This is not valid input. Please enter an integer value: ");
                numInput1 = Console.ReadLine();
            }

            // Ask the user to type the second number.
            Console.Write("Type another number, and then press Enter: ");
            numInput2 = Console.ReadLine();

            double cleanNum2 = 0;
            while (!double.TryParse(numInput2, out cleanNum2))
            {
                Console.Write("This is not valid input. Please enter an integer value: ");
                numInput2 = Console.ReadLine();
            }

            // Ask the user to choose an operator.
            Console.WriteLine("Choose an operator from the following list:");
            Console.WriteLine("\ta - Add");
            Console.WriteLine("\ts - Subtract");
            Console.WriteLine("\tm - Multiply");
            Console.WriteLine("\td - Divide");
            Console.Write("Your option? ");

            string op = Console.ReadLine();

            try
            {
                result = Calculator.DoOperation(cleanNum1, cleanNum2, op);
                if (double.IsNaN(result))
                {
                    Console.WriteLine("This operation will result in a mathematical error.\n");
                }
                else Console.WriteLine("Your result: {0:0.##}\n", result);
            }
            catch (Exception e)
            {
                Console.WriteLine("Oh no! An exception occurred trying to do the math.\n - Details: " +
```

```
e.Message);
                }

                Console.WriteLine("------------------------\n");

                // Wait for the user to respond before closing.
                Console.Write("Press 'n' and Enter to close the app, or press any other key and Enter to
continue: ");
                if (Console.ReadLine() == "n") endApp = true;

                Console.WriteLine("\n"); // Friendly linespacing.
            }
            return;
        }
    }
}
```

Next steps

Continue with more tutorials:

C# tutorials

Tour the Visual Studio IDE

Continue with the second part of this tutorial:

Tutorial Part 2: Extend and debug your C# console app

Tutorial: Extend C# console app and debug in Visual Studio (part 2 of 2)

10/28/2021 • 18 minutes to read • Edit Online

In part 2 of this tutorial series, you dive a little deeper into the Visual Studio build and debug features you need for daily development. These features include managing multiple projects, debugging, and referencing third-party packages. You run the C# console app you created in Part 1 of this tutorial, and explore some features of the Visual Studio integrated development environment (IDE). This tutorial is part 2 of a two-part tutorial series.

In this tutorial, you:

- Add a second project.
- Reference libraries and add packages.
- Do more debugging.
- Inspect your completed code.

Prerequisites

To work through this article, you can use either of these Calculator apps:

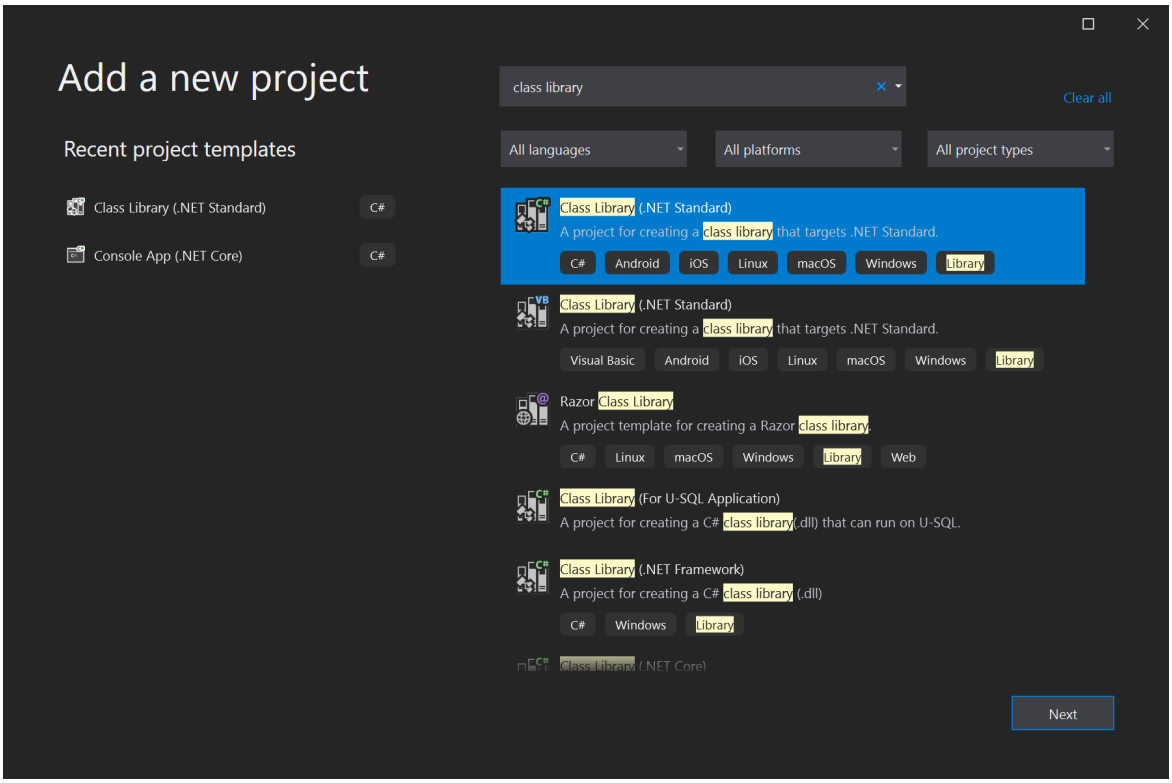
- The Calculator console app from part 1 of this tutorial.
- The C# Calculator app in the vs-tutorial-samples repo. To get started, open the app from the repo.

Add another project

Real-world code involves projects working together in a solution. You can add a class library project to your Calculator app that provides some calculator functions.

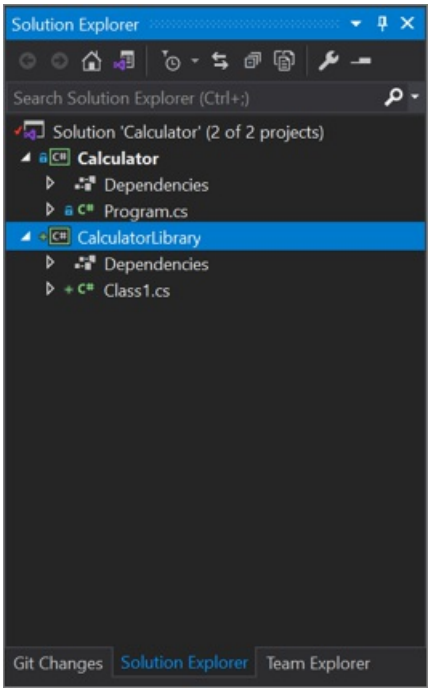
In Visual Studio, you use the menu command **File** > **Add** > **New Project** to add a new project. You can also right-click on the solution in **Solution Explorer** to add a project from the context menu.

1. In **Solution Explorer**, right-click the solution node and choose **Add** > **New Project**.
2. In the **Add a new project** window, type *class library* in the Search box. Choose the C# **Class library** project template, and then select **Next**.



3. On the **Configure your new project** screen, type the project name *CalculatorLibrary*, and then select **Next**.

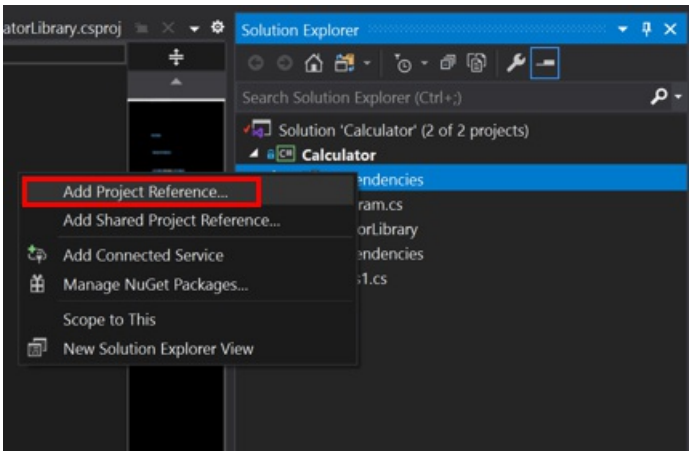
4. Choose .NET 3.1 when asked. Visual Studio creates the new project and adds it to the solution.



5. Rename the *Class1.cs* file to *CalculatorLibrary.cs*. To rename the file, you can right-click the name in **Solution Explorer** and choose **Rename**, select the name and press **F2**, or select the name and click again to type.

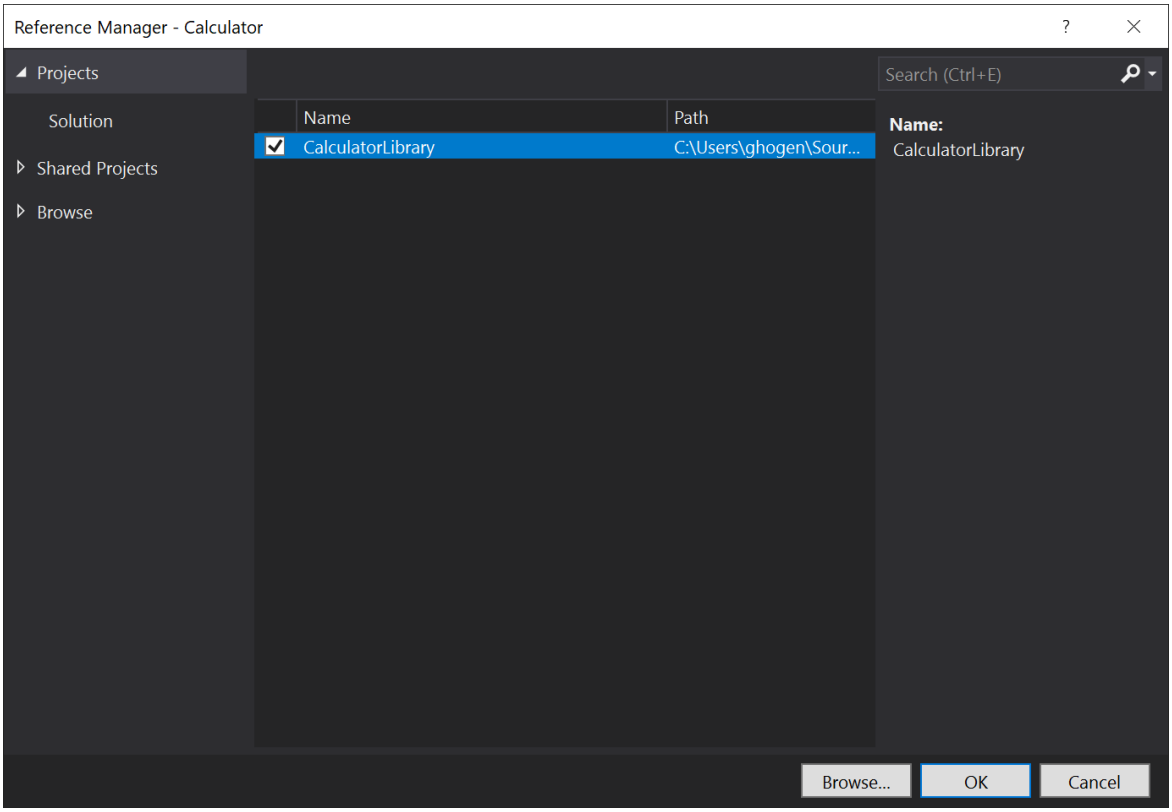
   A message might ask whether you want to rename references to `Class1` in the file. It doesn't matter how you answer, because you'll replace the code in a future step.

6. Now add a project reference, so the first project can use APIs that the new class library exposes. Right-click the **Dependencies** node in the **Calculator** project and choose **Add Project Reference**.

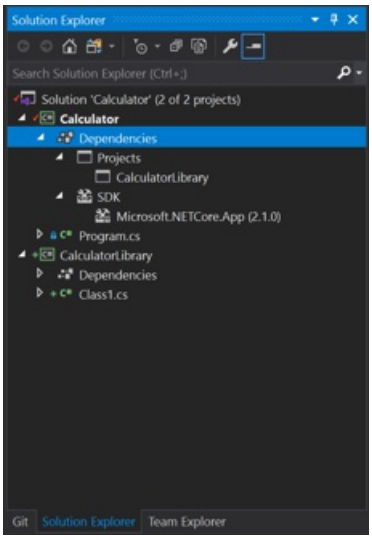


The **Reference Manager** dialog box appears. In this dialog box, you can add references to other projects, assemblies, and COM DLLs that your projects need.

7. In the **Reference Manager** dialog box, select the checkbox for the **CalculatorLibrary** project, and then select **OK**.



The project reference appears under a **Projects** node in **Solution Explorer**.



8. In *Program.cs*, select the `Calculator` class and all its code, and press **Ctrl**+**X** to cut it. Then, in *CalculatorLibrary.cs*, paste the code into the `CalculatorLibrary` namespace.

   Also add `public` before the Calculator class to expose it outside the library.

   CalculatorLibrary.cs should now resemble the following code:

```
using System;

 namespace CalculatorLibrary
 {
     public class Calculator
     {
         public static double DoOperation(double num1, double num2, string op)
         {
             double result = double.NaN; // Default value is "not-a-number" if an operation, such as
division, could result in an error.

             // Use a switch statement to do the math.
             switch (op)
             {
                 case "a":
                     result = num1 + num2;
                     break;
                 case "s":
                     result = num1 - num2;
                     break;
                 case "m":
                     result = num1 * num2;
                     break;
                 case "d":
                     // Ask the user to enter a non-zero divisor.
                     if (num2 != 0)
                     {
                         result = num1 / num2;
                     }
                     break;
                 // Return text for an incorrect option entry.
                 default:
                     break;
             }
             return result;
         }
     }
 }
```

9. *Program.cs* also has a reference, but an error says that the `Calculator.DoOperation` call doesn't resolve.

That's because `CalculatorLibrary` is in a different namespace. For a fully qualified reference, you could add the `CalculatorLibrary` namespace to the `Calculator.DoOperation` call:

```
result = CalculatorLibrary.Calculator.DoOperation(cleanNum1, cleanNum2, op);
```

Or, you could try adding a `using` directive to the beginning of the file:

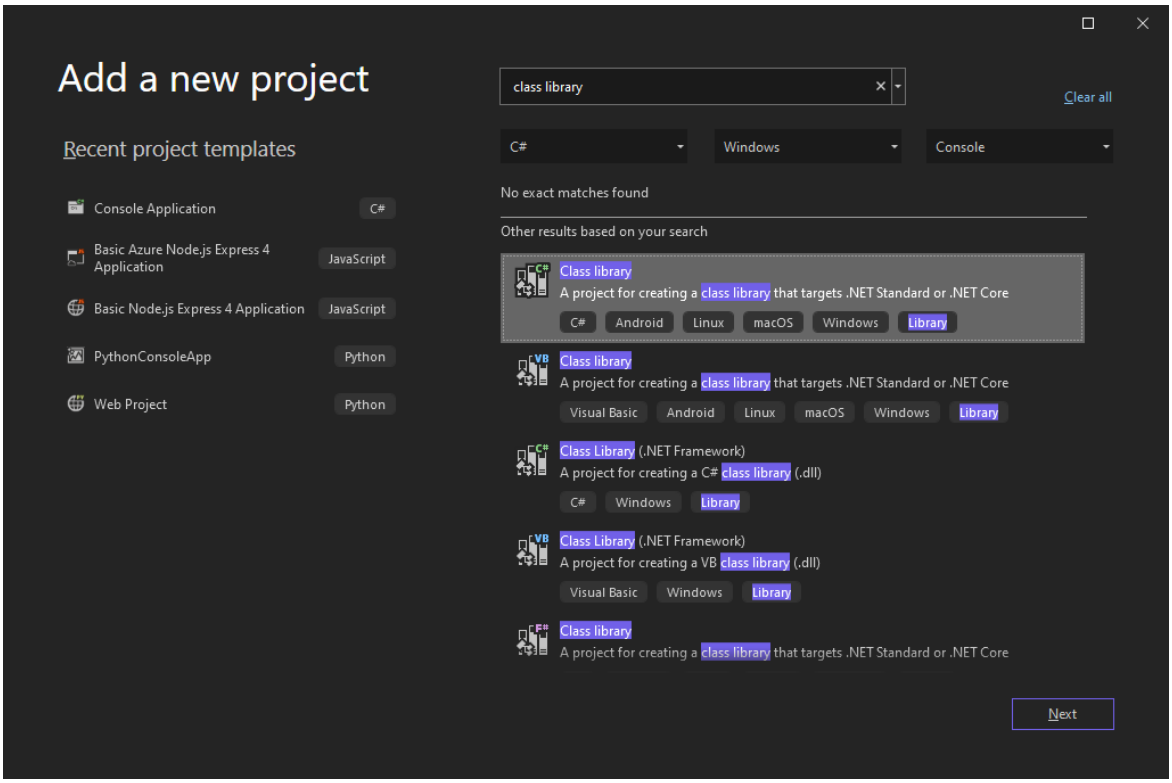
```
using CalculatorLibrary;
```

Adding the `using` directive should let you remove the `CalculatorLibrary` namespace from the call site, but now there's an ambiguity. Is `Calculator` the class in `CalculatorLibrary`, or is `Calculator` the namespace?

To resolve the ambiguity, rename the namespace from `Calculator` to `CalculatorProgram` in both *Program.cs* and *CalculatorLibrary.cs*.

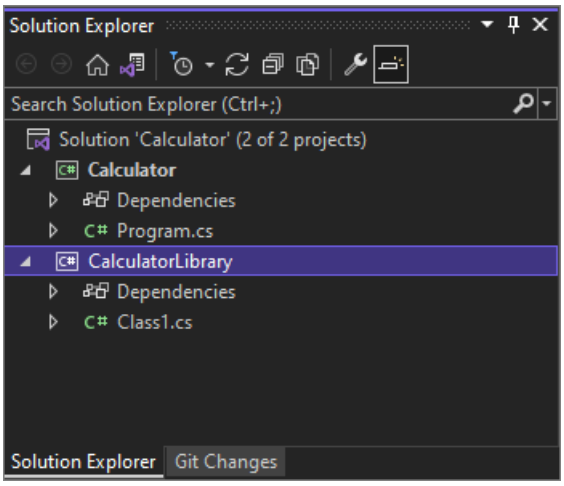
```
namespace CalculatorProgram
```

1. In **Solution Explorer**, right-click the solution node and choose **Add** > **New Project**.
2. In the **Add a new project** window, type *class library* in the Search box. Choose the C# **Class library** project template, and then select **Next**.



3. On the **Configure your new project** screen, type the project name *CalculatorLibrary*, and then select **Next**.
4. On the **Additional information** screen, .NET 6.0 is selected. Select **Create**.

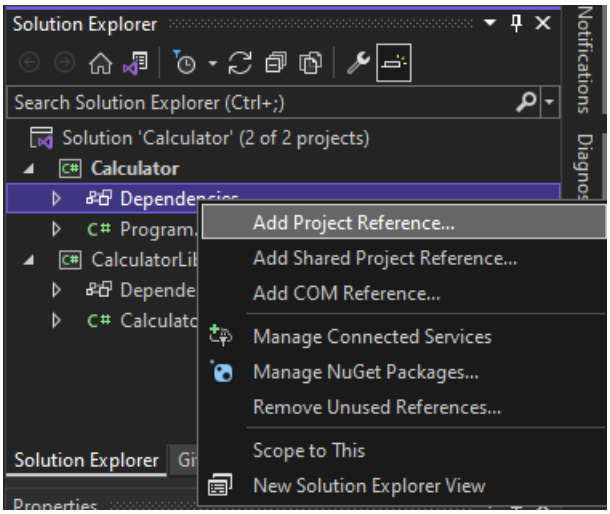
   Visual Studio creates the new project and adds it to the solution.



5. Rename the *Class1.cs* file to *CalculatorLibrary.cs*. To rename the file, you can right-click the name in **Solution Explorer** and choose **Rename**, select the name and press **F2**, or select the name and click again to type.

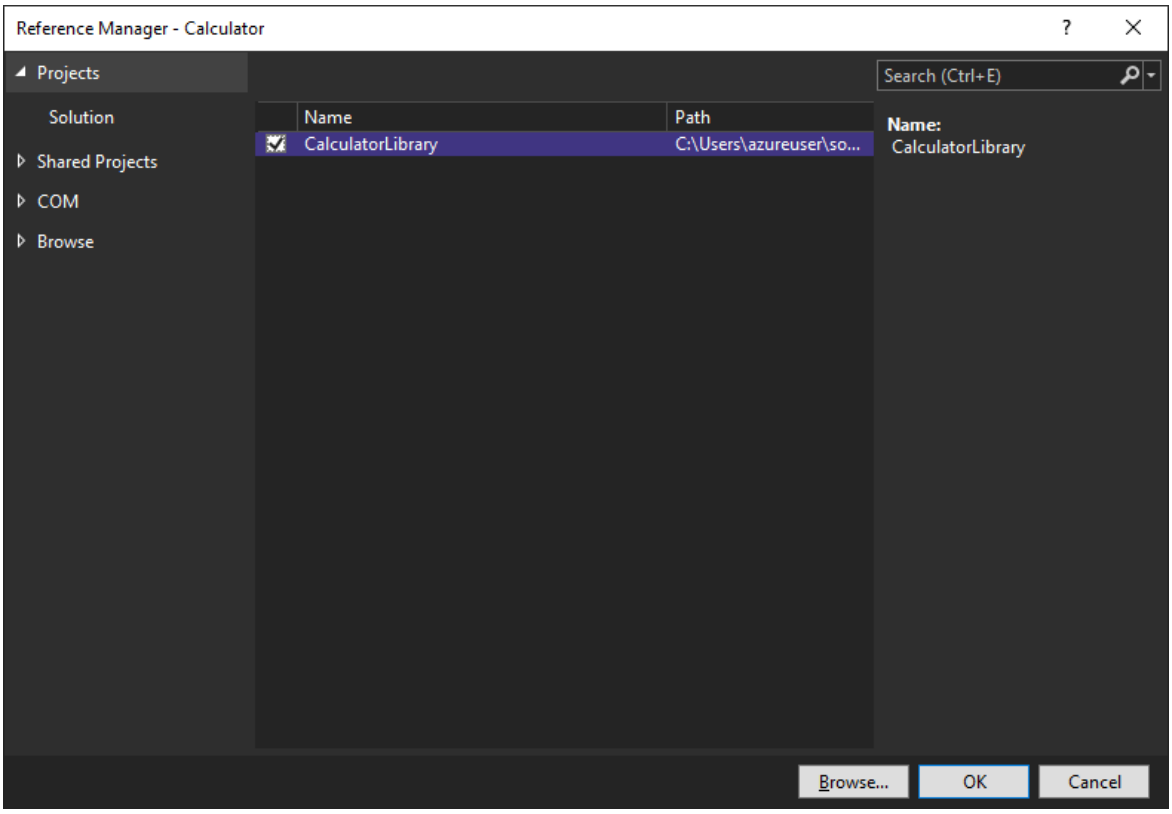
   A message might ask whether you want to rename references to `Class1` in the file. It doesn't matter how you answer, because you'll replace the code in a future step.

6. Now add a project reference, so the first project can use APIs that the new class library exposes. Right-click the **Dependencies** node in the **Calculator** project and choose **Add Project Reference**.

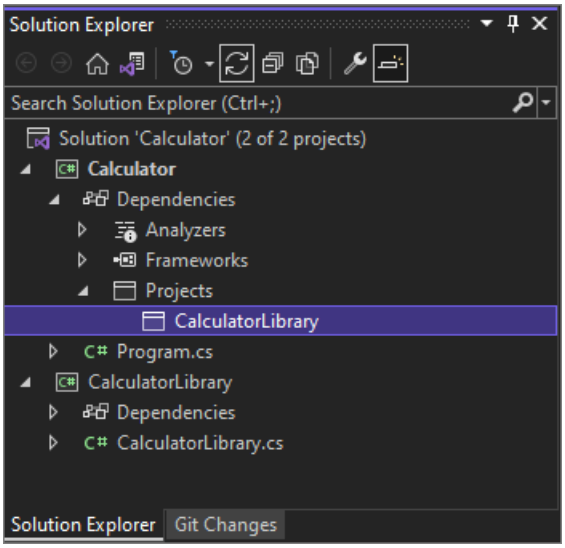


   The **Reference Manager** dialog box appears. In this dialog box, you can add references to other projects, assemblies, and COM DLLs that your projects need.

7. In the **Reference Manager** dialog box, select the checkbox for the **CalculatorLibrary** project, and then select **OK**.



The project reference appears under a **Projects** node in **Solution Explorer**.



8. In *Program.cs*, select the `Calculator` class and all its code, and press **Ctrl**+**X** to cut it. Then, in *CalculatorLibrary.cs*, paste the code into the `CalculatorLibrary` namespace.

   Also add `public` before the Calculator class to expose it outside the library.

   CalculatorLibrary.cs should now resemble the following code:

```
using System;

 namespace CalculatorLibrary
 {
     public class Calculator
     {
         public static double DoOperation(double num1, double num2, string op)
         {
             double result = double.NaN; // Default value is "not-a-number" if an operation, such as
division, could result in an error.

             // Use a switch statement to do the math.
             switch (op)
             {
                 case "a":
                     result = num1 + num2;
                     break;
                 case "s":
                     result = num1 - num2;
                     break;
                 case "m":
                     result = num1 * num2;
                     break;
                 case "d":
                     // Ask the user to enter a non-zero divisor.
                     if (num2 != 0)
                     {
                         result = num1 / num2;
                     }
                     break;
                 // Return text for an incorrect option entry.
                 default:
                     break;
             }
             return result;
         }
     }
 }
```

9. *Program.cs* also has a reference, but an error says that the `Calculator.DoOperation` call doesn't resolve. That's because `CalculatorLibrary` is in a different namespace. For a fully qualified reference, you could add the `CalculatorLibrary` namespace to the `Calculator.DoOperation` call:

   ```
   result = CalculatorLibrary.Calculator.DoOperation(cleanNum1, cleanNum2, op);
   ```

   Or, you could try adding a `using` directive to the beginning of the file:

   ```
   using CalculatorLibrary;
   ```

   Adding the `using` directive should let you remove the `CalculatorLibrary` namespace from the call site, but now there's an ambiguity. Is `Calculator` the class in `CalculatorLibrary`, or is `Calculator` the namespace?

   To resolve the ambiguity, rename the namespace from `Calculator` to `CalculatorProgram` in both *Program.cs* and *CalculatorLibrary.cs*.

   ```
   namespace CalculatorProgram
   ```

Reference .NET libraries: Write to a log

You can use the .NET `Trace` class to add a log of all operations, and write it to a text file. The `Trace` class is also useful for basic print debugging techniques. The `Trace` class is in `System.Diagnostics`, and uses `System.IO` classes like `StreamWriter`.

1. Start by adding the `using` directives at the top of *CalculatorLibrary.cs*:

   ```
   using System.IO;
   using System.Diagnostics;
   ```

2. This usage of the `Trace` class must hold onto a reference for the class, which it associates with a filestream. That requirement means the calculator works better as an object, so add a constructor at the beginning of the `Calculator` class in *CalculatorLibrary.cs*.

   Also remove the `static` keyword to change the static `DoOperation` method into a member method.

   ```
   public Calculator()
      {
          StreamWriter logFile = File.CreateText("calculator.log");
          Trace.Listeners.Add(new TextWriterTraceListener(logFile));
          Trace.AutoFlush = true;
          Trace.WriteLine("Starting Calculator Log");
          Trace.WriteLine(String.Format("Started {0}", System.DateTime.Now.ToString()));
      }

   public double DoOperation(double num1, double num2, string op)
      {
   ```

3. Add log output to each calculation. `DoOperation` should now look like the following code:

```
public double DoOperation(double num1, double num2, string op)
{
     double result = double.NaN; // Default value is "not-a-number" if an operation, such as
division, could result in an error.

     // Use a switch statement to do the math.
     switch (op)
     {
         case "a":
             result = num1 + num2;
             Trace.WriteLine(String.Format("{0} + {1} = {2}", num1, num2, result));
             break;
         case "s":
             result = num1 - num2;
             Trace.WriteLine(String.Format("{0} - {1} = {2}", num1, num2, result));
             break;
         case "m":
             result = num1 * num2;
             Trace.WriteLine(String.Format("{0} * {1} = {2}", num1, num2, result));
             break;
         case "d":
             // Ask the user to enter a non-zero divisor.
             if (num2 != 0)
             {
                 result = num1 / num2;
                 Trace.WriteLine(String.Format("{0} / {1} = {2}", num1, num2, result));
             }
                 break;
         // Return text for an incorrect option entry.
         default:
             break;
     }
     return result;
 }
```

4. Back in *Program.cs*, a red squiggly underline now flags the static call. To fix the error, create a `calculator` variable by adding the following code line just before the `while (!endApp)` loop:

   ```
   Calculator calculator = new Calculator();
   ```

   Also modify the `DoOperation` call site to reference the object named `calculator` in lowercase. The code is now a member invocation, rather than a call to a static method.

   ```
   result = calculator.DoOperation(cleanNum1, cleanNum2, op);
   ```

5. Run the app again. When you're done, right-click the **Calculator** project node and choose **Open Folder in File Explorer**.

6. In File Explorer, navigate to the output folder under *bin/Debug/*, and open the *calculator.log* file. The output should look something like this:

   ```
   Starting Calculator Log
   Started 7/9/2020 1:58:19 PM
   1 + 2 = 3
   3 * 3 = 9
   ```

At this point, *CalculatorLibrary.cs* should resemble this code:

```
using System;
using System.IO;
using System.Diagnostics;


namespace CalculatorLibrary
{
    public class Calculator
    {

        public Calculator()
        {
            StreamWriter logFile = File.CreateText("calculator.log");
            Trace.Listeners.Add(new TextWriterTraceListener(logFile));
            Trace.AutoFlush = true;
            Trace.WriteLine("Starting Calculator Log");
            Trace.WriteLine(String.Format("Started {0}", System.DateTime.Now.ToString()));
        }

        public double DoOperation(double num1, double num2, string op)
        {
            double result = double.NaN; // Default value is "not-a-number" if an operation, such as
division, could result in an error.

            // Use a switch statement to do the math.
            switch (op)
            {
                case "a":
                    result = num1 + num2;
                    Trace.WriteLine(String.Format("{0} + {1} = {2}", num1, num2, result));
                    break;
                case "s":
                    result = num1 - num2;
                    Trace.WriteLine(String.Format("{0} - {1} = {2}", num1, num2, result));
                    break;
                case "m":
                    result = num1 * num2;
                    Trace.WriteLine(String.Format("{0} * {1} = {2}", num1, num2, result));
                    break;
                case "d":
                    // Ask the user to enter a non-zero divisor.
                    if (num2 != 0)
                    {
                        result = num1 / num2;
                        Trace.WriteLine(String.Format("{0} / {1} = {2}", num1, num2, result));
                    }
                    break;
                // Return text for an incorrect option entry.
                default:
                    break;
            }
            return result;
        }
    }
}
```

Program.cs should look like the following code:

```
using System;
using CalculatorLibrary;

namespace CalculatorProgram
{

    class Program
    {
```

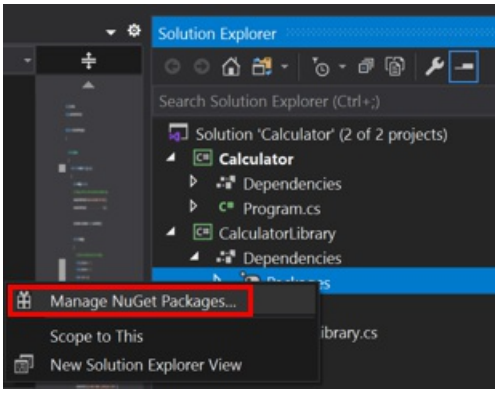
```
        static void Main(string[] args)
        {
            bool endApp = false;
            // Display title as the C# console calculator app.
            Console.WriteLine("Console Calculator in C#\r");
            Console.WriteLine("------------------------\n");

            Calculator calculator = new Calculator();
            while (!endApp)
            {
                // Declare variables and set to empty.
                string numInput1 = "";
                string numInput2 = "";
                double result = 0;

                // Ask the user to type the first number.
                Console.Write("Type a number, and then press Enter: ");
                numInput1 = Console.ReadLine();

                double cleanNum1 = 0;
                while (!double.TryParse(numInput1, out cleanNum1))
                {
                    Console.Write("This is not valid input. Please enter an integer value: ");
                    numInput1 = Console.ReadLine();
                }

                // Ask the user to type the second number.
                Console.Write("Type another number, and then press Enter: ");
                numInput2 = Console.ReadLine();

                double cleanNum2 = 0;
                while (!double.TryParse(numInput2, out cleanNum2))
                {
                    Console.Write("This is not valid input. Please enter an integer value: ");
                    numInput2 = Console.ReadLine();
                }

                // Ask the user to choose an operator.
                Console.WriteLine("Choose an operator from the following list:");
                Console.WriteLine("\ta - Add");
                Console.WriteLine("\ts - Subtract");
                Console.WriteLine("\tm - Multiply");
                Console.WriteLine("\td - Divide");
                Console.Write("Your option? ");

                string op = Console.ReadLine();

                try
                {
                    result = calculator.DoOperation(cleanNum1, cleanNum2, op);
                    if (double.IsNaN(result))
                    {
                        Console.WriteLine("This operation will result in a mathematical error.\n");
                    }
                    else Console.WriteLine("Your result: {0:0.##}\n", result);
                }
                catch (Exception e)
                {
                    Console.WriteLine("Oh no! An exception occurred trying to do the math.\n - Details: " +
e.Message);
                }

                Console.WriteLine("------------------------\n");

                // Wait for the user to respond before closing.
                Console.Write("Press 'n' and Enter to close the app, or press any other key and Enter to
continue: ");
                if (Console.ReadLine() == "n") endApp = true;
```

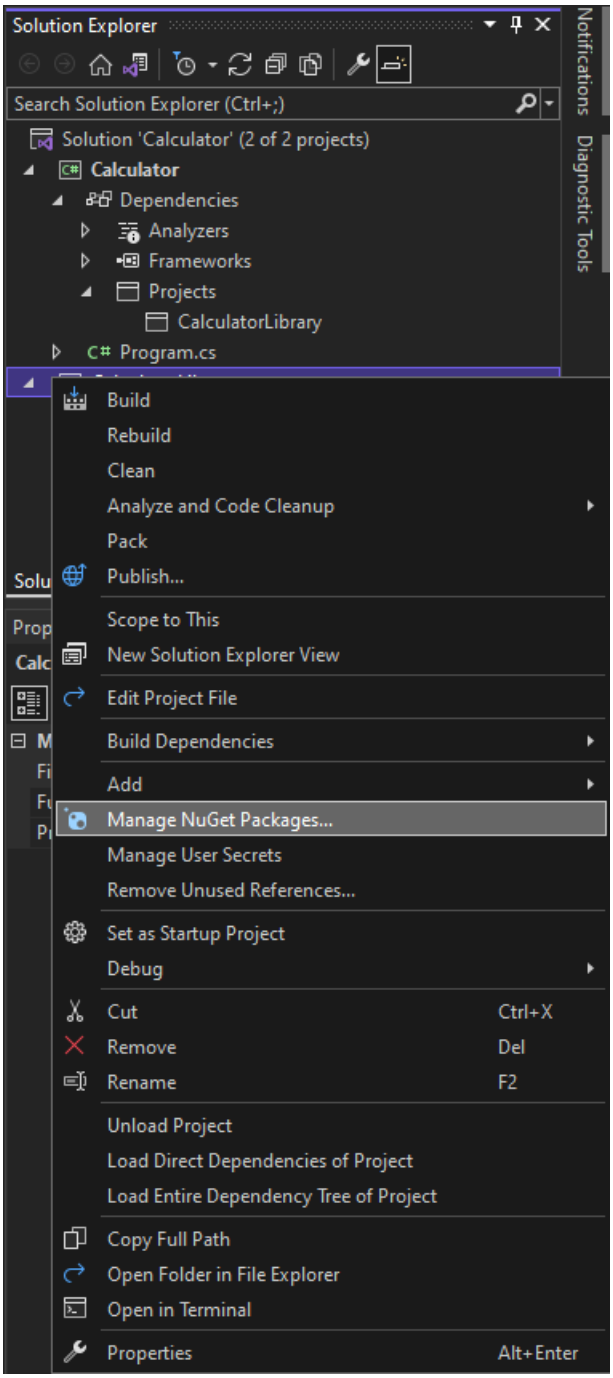
```
                Console.WriteLine("\n"); // Friendly linespacing.
            }
            return;
        }
    }
}
```

Add a NuGet Package: Write to a JSON file

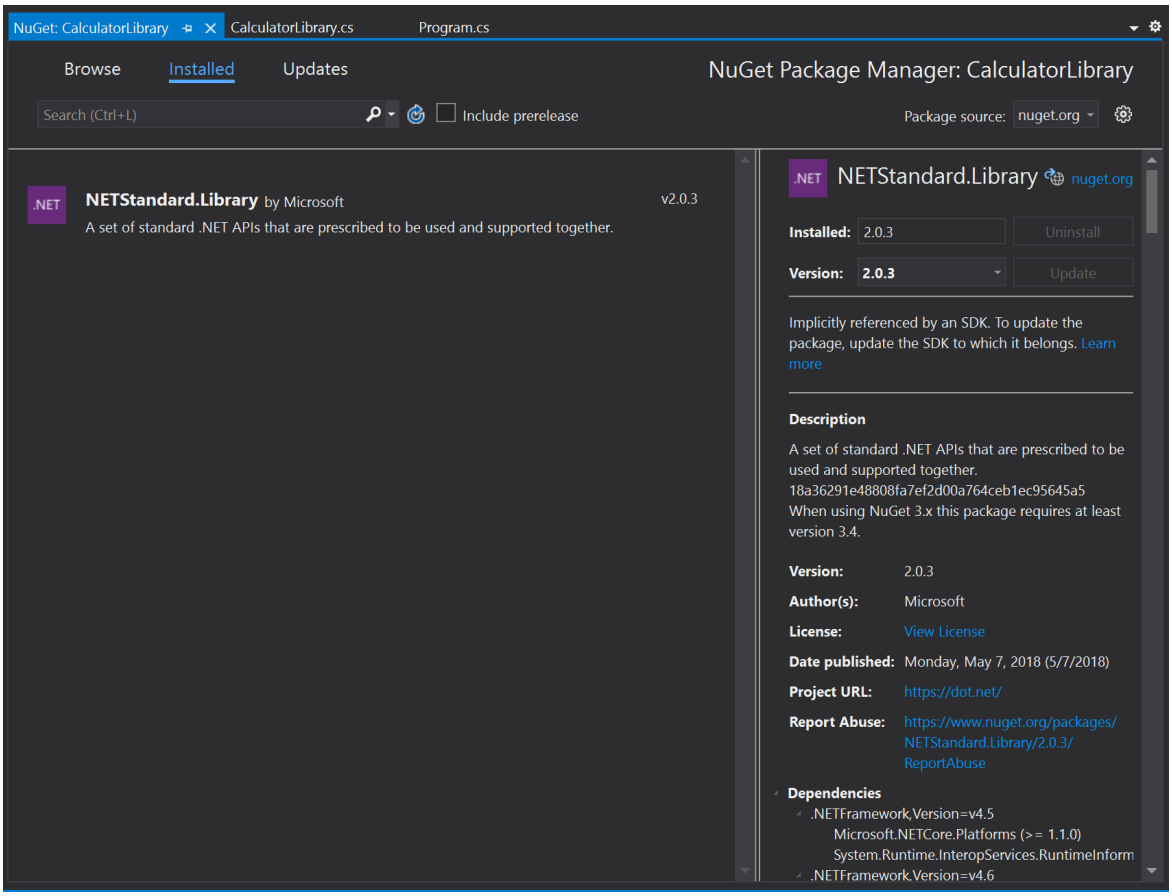
To output operations in JSON, a popular and portable format for storing object data, you can reference the *NewtonsoftJson* NuGet package. NuGet packages are the primary distribution method for .NET class libraries.

1. In **Solution Explorer**, right-click the **Dependencies** node for the **CalculatorLibrary** project, and choose **Manage NuGet Packages**.

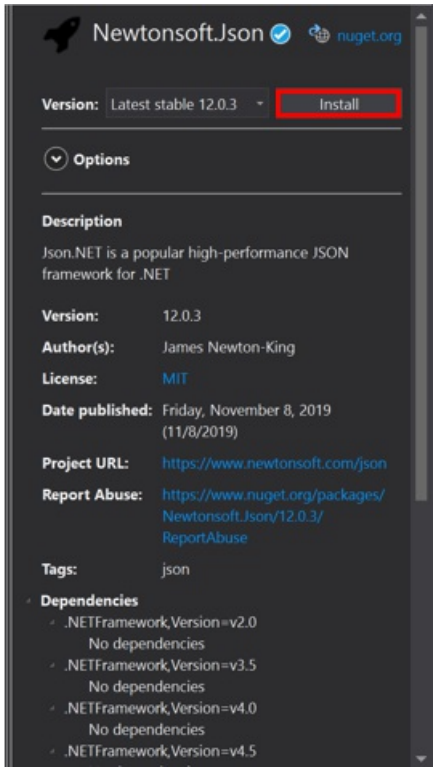




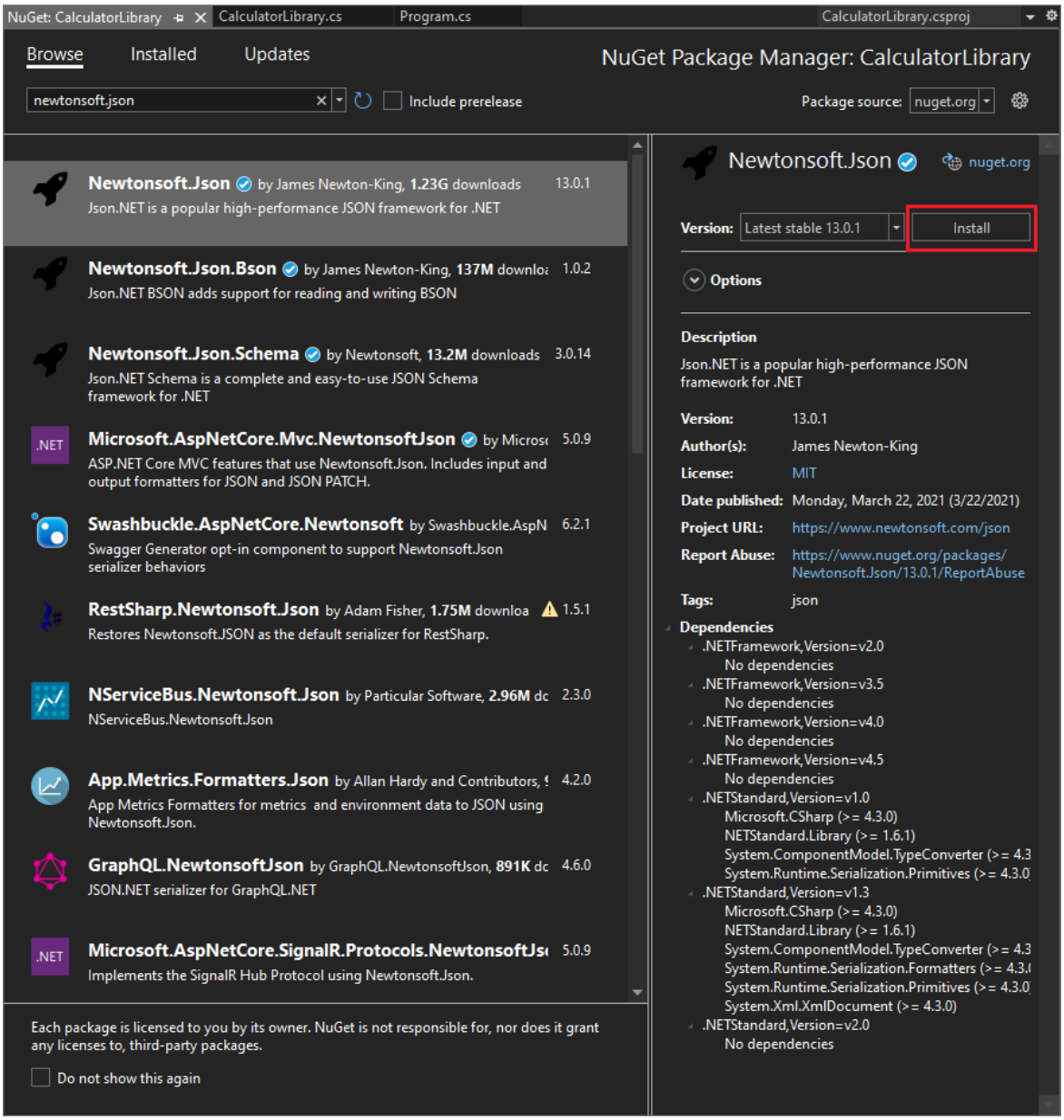
The NuGet Package Manager opens.



2. Search for and select the *Newtonsoft.Json* package, and select **Install**.



Visual Studio downloads the package and adds it to the project. A new entry appears in the References node in **Solution Explorer**.



If you're prompted whether to accept changes, select **OK**.

Visual Studio downloads the package and adds it to the project. A new entry appears in a **Packages** node in **Solution Explorer**.

3. Add a `using` directive for `Newtonsoft.Json` at the beginning of *CalculatorLibrary.cs*.

```
using Newtonsoft.Json;
```

4. Create the `JsonWriter` member object, and replace the `Calculator` constructor with the following code:

```
    JsonWriter writer;

    public Calculator()
    {
        StreamWriter logFile = File.CreateText("calculatorlog.json");
        logFile.AutoFlush = true;
        writer = new JsonTextWriter(logFile);
        writer.Formatting = Formatting.Indented;
        writer.WriteStartObject();
        writer.WritePropertyName("Operations");
        writer.WriteStartArray();
    }
```

5. Modify the `DoOperation` method to add the JSON `writer` code:

```
    public double DoOperation(double num1, double num2, string op)
    {
        double result = double.NaN; // Default value is "not-a-number" if an operation, such as
division, could result in an error.
        writer.WriteStartObject();
        writer.WritePropertyName("Operand1");
        writer.WriteValue(num1);
        writer.WritePropertyName("Operand2");
        writer.WriteValue(num2);
        writer.WritePropertyName("Operation");
        // Use a switch statement to do the math.
        switch (op)
        {
            case "a":
                result = num1 + num2;
                writer.WriteValue("Add");
                break;
            case "s":
                result = num1 - num2;
                writer.WriteValue("Subtract");
                break;
            case "m":
                result = num1 * num2;
                writer.WriteValue("Multiply");
                break;
            case "d":
                // Ask the user to enter a non-zero divisor.
                if (num2 != 0)
                {
                    result = num1 / num2;
                    writer.WriteValue("Divide");
                }
                break;
            // Return text for an incorrect option entry.
            default:
                break;
        }
        writer.WritePropertyName("Result");
        writer.WriteValue(result);
        writer.WriteEndObject();

        return result;
    }
```

6. Add a method to finish the JSON syntax once the user is done entering operation data.

```
public void Finish()
{
    writer.WriteEndArray();
    writer.WriteEndObject();
    writer.Close();
}
```

7. At the end of *Program.cs*, before the `return;`, add a call to `Finish`:

```
        // Add call to close the JSON writer before return
        calculator.Finish();
        return;
    }
```

8. Build and run the app, and after you're done entering a few operations, close the app by entering the *n*

command.

9. Open the *calculatorlog.json* file in File Explorer. You should see something like the following content:

```
{
 "Operations": [
     {
     "Operand1": 2.0,
     "Operand2": 3.0,
     "Operation": "Add",
     "Result": 5.0
     },
     {
     "Operand1": 3.0,
     "Operand2": 4.0,
     "Operation": "Multiply",
     "Result": 12.0
     }
 ]
}
```

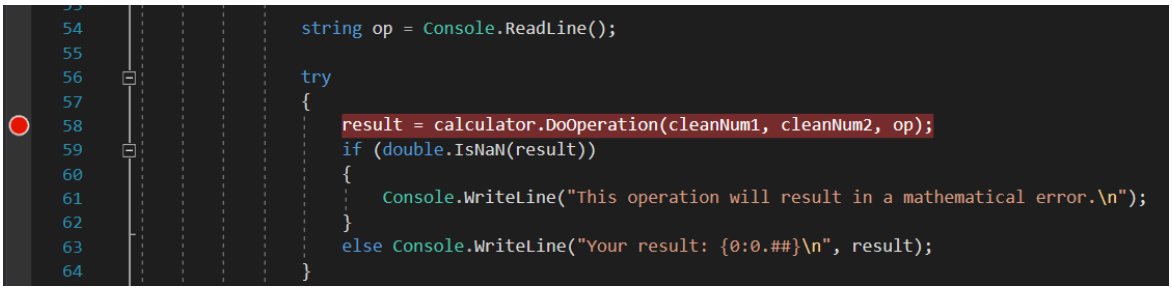
Debug: Set and hit a breakpoint

The Visual Studio debugger is a powerful tool. The debugger can step through your code to find the exact point where there's a programming mistake. You can then understand what corrections you need to make, and make temporary changes so you can continue running your app.

1. In *Program.cs*, click in the gutter to the left of the following code line. You can also click in the line and select **F9**, or right-click the line and select **Breakpoint** > **Insert Breakpoint**.

```
result = calculator.DoOperation(cleanNum1, cleanNum2, op);
```

The red dot that appears indicates a breakpoint. You can use breakpoints to pause your app and inspect code. You can set a breakpoint on any executable line of code.



2. Build and run the app. Enter the following values for the calculation:

   - For the first number, enter *8*.
   - For the second number, enter *0*.
   - For the operator, let's have some fun. Enter *d*.

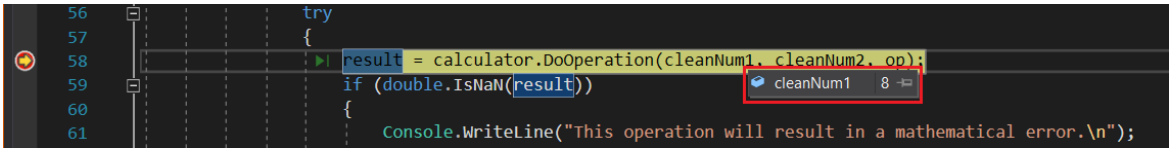
   The app suspends where you created the breakpoint, which is indicated by the yellow pointer on the left and the highlighted code. The highlighted code hasn't yet executed.

```
54          string op = Console.ReadLine();
55
56          try
57          {
58              result = calculator.DoOperation(cleanNum1, cleanNum2, op);
59              if (double.IsNaN(result))
60              {
61                  Console.WriteLine("This operation will result in a mathematical error.\n");
62              }
63              else Console.WriteLine("Your result: {0:0.##}\n", result);
64          }
```

Now, with the app suspended, you can inspect your application state.

Debug: View variables

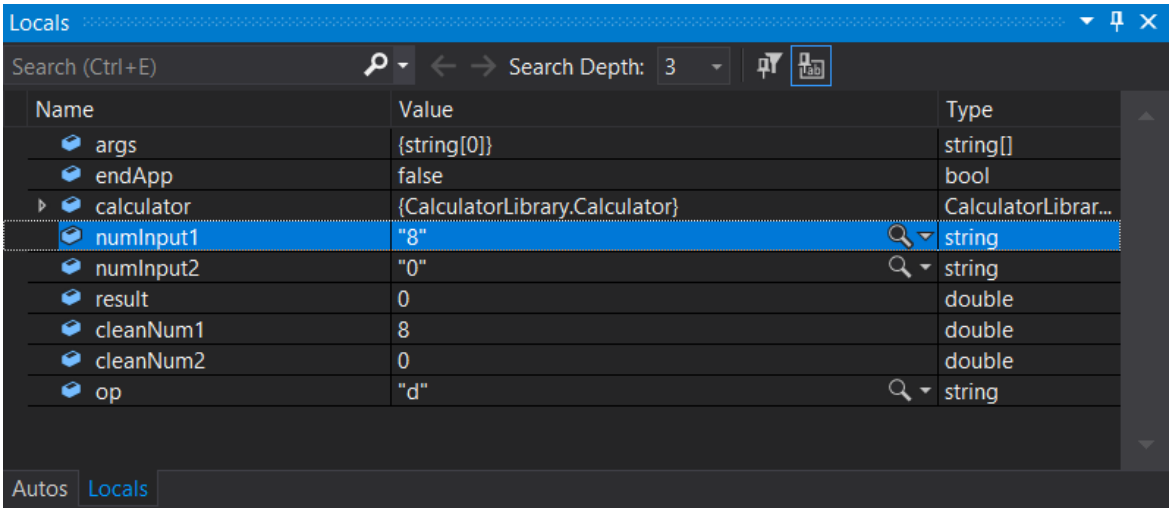
1. In the highlighted code, hover over variables such as `cleanNum1` and `op`. The current values for these variables, `8` and `d` respectively, appear in DataTips.

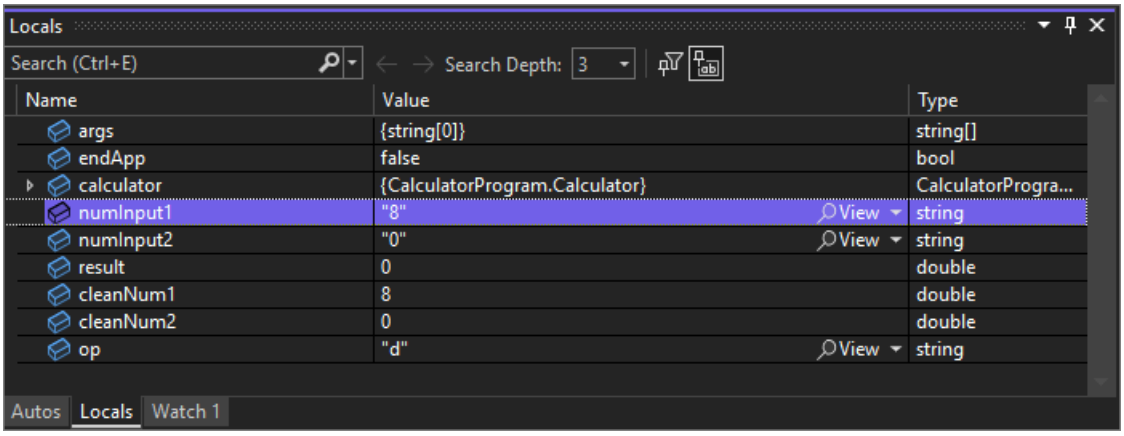


When debugging, checking to see whether variables hold the values you expect is often critical to fixing issues.

2. In the lower pane, look at the **Locals** window. If it's closed, select **Debug** > **Windows** > **Locals** to open it.

   The **Locals** window shows each variable that's currently in scope, along with its value and type.





3. Look at the **Autos** window.

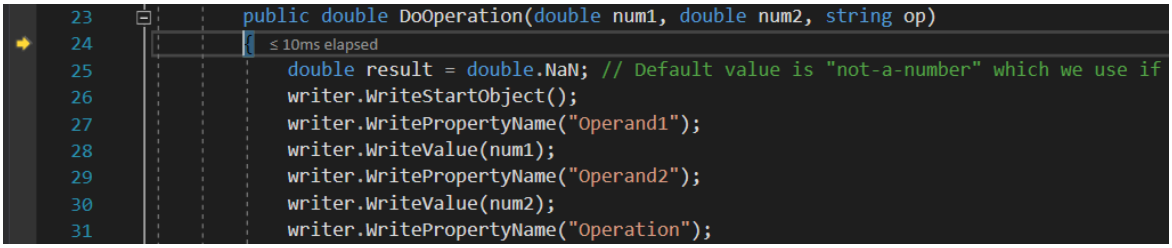
The Autos window is similar to the **Locals** window, but shows the variables immediately preceding and following the current line of code where your app is paused.

Next, execute code in the debugger one statement at a time, which is called *stepping*.

Debug: Step through code

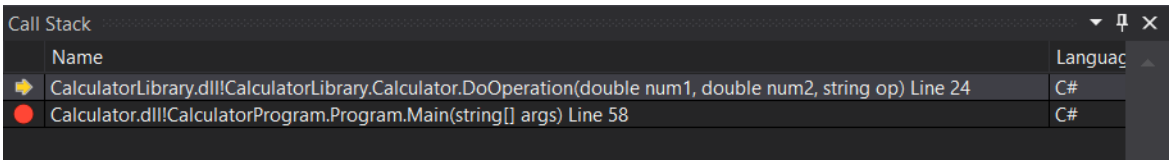
1. Press **F11**, or select **Debug** > **Step Into**.

   Using the Step Into command, the app executes the current statement and advances to the next executable statement, usually the next line of code. The yellow pointer on the left always indicates the current statement.

   You just stepped into the `DoOperation` method in the `Calculator` class.

2. To get a hierarchical look at your program flow, look at the **Call Stack** window. If it's closed, select **Debug** > **Windows** > **Call Stack** to open it.

   This view shows the current `Calculator.DoOperation` method, indicated by the yellow pointer. The second row shows the function that called the method, from the `Main` method in *Program.cs*.

   The **Call Stack** window shows the order in which methods and functions are getting called. This window also provides access to many debugger features, such as **Go to Source Code**, from its shortcut menu.

3. Press **F10**, or select **Debug** > **Step Over**, several times until the app pauses on the `switch` statement.

   ```
   switch (op)
   {
   ```

   The Step Over command is similar to the Step Into command, except that if the current statement calls a function, the debugger runs the code in the function, and doesn't suspend execution until the function returns. Step Over is faster than Step Into if you're not interested in a particular function.

4. Press **F10** one more time, so that the app pauses on the following line of code.

   ```
   if (num2 != 0)
   {
   ```

   This code checks for a divide-by-zero case. If the app continues, it will throw a general exception (an error), but you might want to try something else, like viewing the actual returned value in the console. One option is to use a debugger feature called *edit-and-continue* to make changes to the code and then continue debugging. However, there's a different trick to temporarily modify the execution flow.

Debug: Test a temporary change

1. Select the yellow pointer, currently paused on the `if (num2 != 0)` statement, and drag it to the following statement:

   ```
   result = num1 / num2;
   ```

   Dragging the pointer here causes the app to completely skip the `if` statement, so you can see what happens when you divide by zero.

2. Press **F10** to execute the line of code.

3. If you hover over the `result` variable, it shows a value of **Infinity**. In C#, Infinity is the result when you divide by zero.

4. Press **F5**, or select **Debug** > **Continue Debugging**.

   The infinity symbol appears in the console as the result of the math operation.

5. Close the app properly by entering the *n* command.

Code complete

Here's the complete code for the *CalculatorLibrary.cs* file, after you complete all the steps:

```
using System;
using System.IO;
using System.Diagnostics;
using Newtonsoft.Json;

namespace CalculatorLibrary
{
    public class Calculator
    {

        JsonWriter writer;

        public Calculator()
        {
            StreamWriter logFile = File.CreateText("calculatorlog.json");
            logFile.AutoFlush = true;
            writer = new JsonTextWriter(logFile);
            writer.Formatting = Formatting.Indented;
            writer.WriteStartObject();
            writer.WritePropertyName("Operations");
            writer.WriteStartArray();
        }

        public double DoOperation(double num1, double num2, string op)
        {
            double result = double.NaN; // Default value is "not-a-number" if an operation, such as
division, could result in an error.
            writer.WriteStartObject();
            writer.WritePropertyName("Operand1");
            writer.WriteValue(num1);
            writer.WritePropertyName("Operand2");
            writer.WriteValue(num2);
            writer.WritePropertyName("Operation");
            // Use a switch statement to do the math.
            switch (op)
            {
                case "a":
                    result = num1 + num2;
```

```
                    writer.WriteValue("Add");
                    break;
                case "s":
                    result = num1 - num2;
                    writer.WriteValue("Subtract");
                    break;
                case "m":
                    result = num1 * num2;
                    writer.WriteValue("Multiply");
                    break;
                case "d":
                    // Ask the user to enter a non-zero divisor.
                    if (num2 != 0)
                    {
                        result = num1 / num2;
                        writer.WriteValue("Divide");
                    }
                    break;
                // Return text for an incorrect option entry.
                default:
                    break;
            }
            writer.WritePropertyName("Result");
            writer.WriteValue(result);
            writer.WriteEndObject();

            return result;
        }

        public void Finish()
        {
            writer.WriteEndArray();
            writer.WriteEndObject();
            writer.Close();
        }
    }
}
```

And here's the code for *Program.cs*:

```
using System;
using CalculatorLibrary;

namespace CalculatorProgram
{

    class Program
    {
        static void Main(string[] args)
        {
            bool endApp = false;
            // Display title as the C# console calculator app.
            Console.WriteLine("Console Calculator in C#\r");
            Console.WriteLine("------------------------\n");

            Calculator calculator = new Calculator();
            while (!endApp)
            {
                // Declare variables and set to empty.
                string numInput1 = "";
                string numInput2 = "";
                double result = 0;

                // Ask the user to type the first number.
                Console.Write("Type a number, and then press Enter: ");
                numInput1 = Console.ReadLine();
```

```
                double cleanNum1 = 0;
                while (!double.TryParse(numInput1, out cleanNum1))
                {
                    Console.Write("This is not valid input. Please enter an integer value: ");
                    numInput1 = Console.ReadLine();
                }

                // Ask the user to type the second number.
                Console.Write("Type another number, and then press Enter: ");
                numInput2 = Console.ReadLine();

                double cleanNum2 = 0;
                while (!double.TryParse(numInput2, out cleanNum2))
                {
                    Console.Write("This is not valid input. Please enter an integer value: ");
                    numInput2 = Console.ReadLine();
                }

                // Ask the user to choose an operator.
                Console.WriteLine("Choose an operator from the following list:");
                Console.WriteLine("\ta - Add");
                Console.WriteLine("\ts - Subtract");
                Console.WriteLine("\tm - Multiply");
                Console.WriteLine("\td - Divide");
                Console.Write("Your option? ");

                string op = Console.ReadLine();

                try
                {
                    result = calculator.DoOperation(cleanNum1, cleanNum2, op);
                    if (double.IsNaN(result))
                    {
                        Console.WriteLine("This operation will result in a mathematical error.\n");
                    }
                    else Console.WriteLine("Your result: {0:0.##}\n", result);
                }
                catch (Exception e)
                {
                    Console.WriteLine("Oh no! An exception occurred trying to do the math.\n - Details: " +
e.Message);
                }

                Console.WriteLine("-----------------------\n");

                // Wait for the user to respond before closing.
                Console.Write("Press 'n' and Enter to close the app, or press any other key and Enter to
continue: ");
                if (Console.ReadLine() == "n") endApp = true;

                Console.WriteLine("\n"); // Friendly linespacing.
            }
            calculator.Finish();
            return;
        }
    }
}
```

Next steps

Congratulations on completing this tutorial! To learn more, continue with the following content:

- Continue with more C# tutorials.
- Quickstart: Create a ASP.NET Core web app.
- Learn to debug C# code in Visual Studio.

- Walk through how to create and run unit tests.
- Run a C# program.
- Learn about C# IntelliSense.
- Continue with the Visual Studio IDE overview.

Tutorial: Get started with C# and ASP.NET Core in Visual Studio

10/28/2021 • 15 minutes to read • Edit Online

In this tutorial for C# development with ASP.NET Core using Visual Studio, you'll create a C# ASP.NET Core web app, make changes to it, explore some features of the IDE, and then run the app.

Prerequisites

1. Install Visual Studio

   If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

   If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

2. Update Visual Studio - If you've already installed Visual Studio, make sure that you're running the most recent release. For more information about how to update your installation, see the Update Visual Studio to the most recent release page.

3. Choose your theme (optional) - This tutorial includes screenshots that use the dark theme. You can Personalize the Visual Studio IDE and Editor page to learn how.

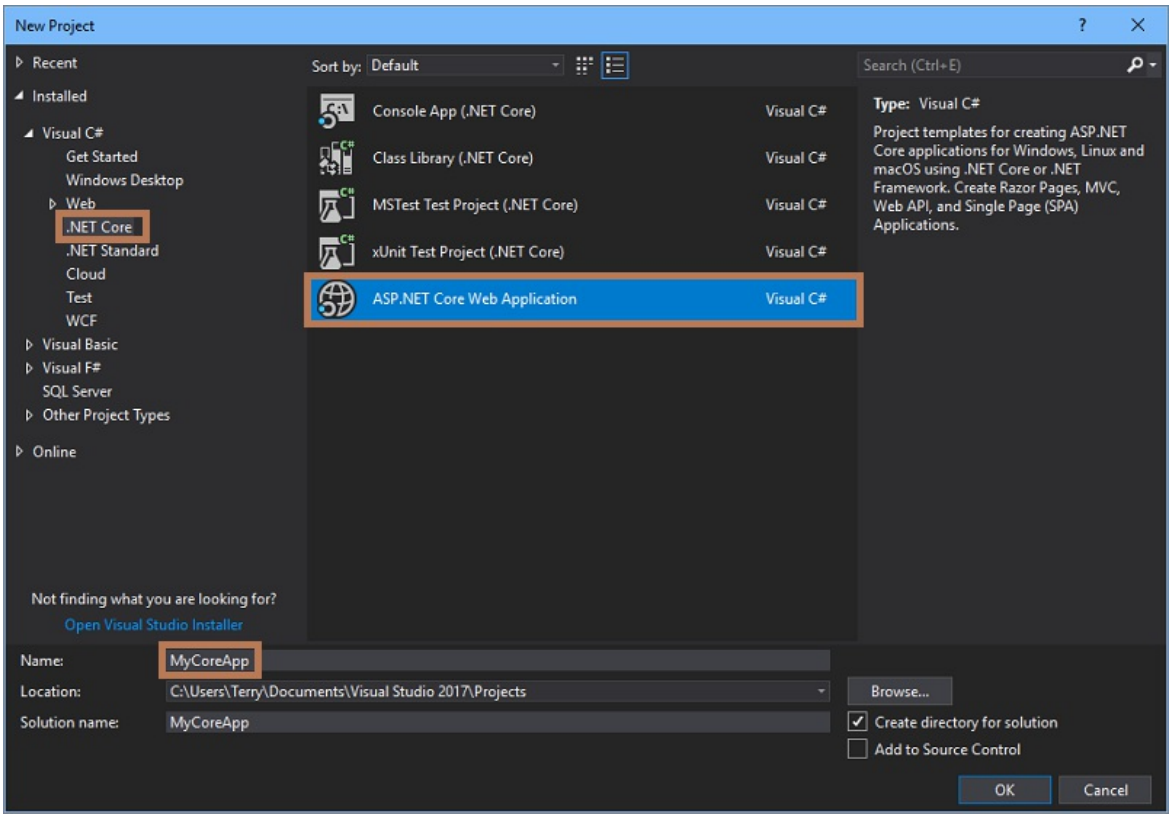
Create a project

First, you'll create a ASP.NET Core project. The project type comes with all the template files you'll need for a fully functional website, before you've even added anything!

1. Open Visual Studio 2017.

2. From the top menu bar, choose **File** > **New** > **Project**.

3. In the **New Project** dialog box in the left pane, expand **Visual C#**, expand **Web**, and then choose **.NET Core**. In the middle pane, choose **ASP.NET Core Web Application**. Then, name the file *MyCoreApp* and choose **OK**.

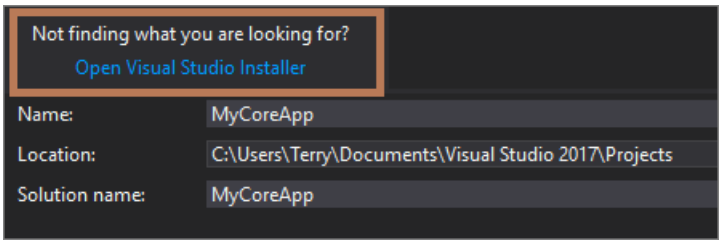


Add a workload (optional)

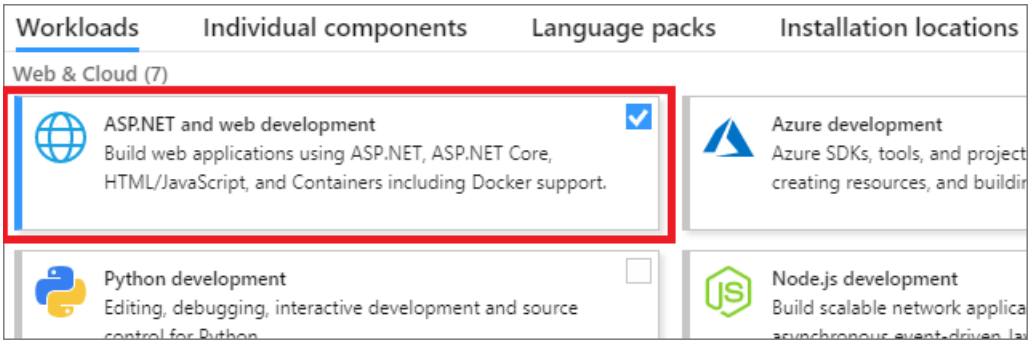
If you don't see the **ASP.NET Core Web Application** project template, you can get it by adding the **ASP.NET and web development** workload. You can add this workload in one of the two following ways, depending on which Visual Studio 2017 updates are installed on your machine.

Option 1: Use the New Project dialog box

1. Select the **Open Visual Studio Installer** link in the left pane of the **New Project** dialog box. (Depending on your display settings, you might have to scroll to see it.)



2. The Visual Studio Installer launches. Choose the **ASP.NET and web development** workload, and then choose **Modify**.



(You might have to close Visual Studio before you can continue installing the new workload.)

Option 2: Use the Tools menu bar

1. Cancel out of the **New Project** dialog box. Then, from the top menu bar, choose **Tools** > **Get Tools and Features**.

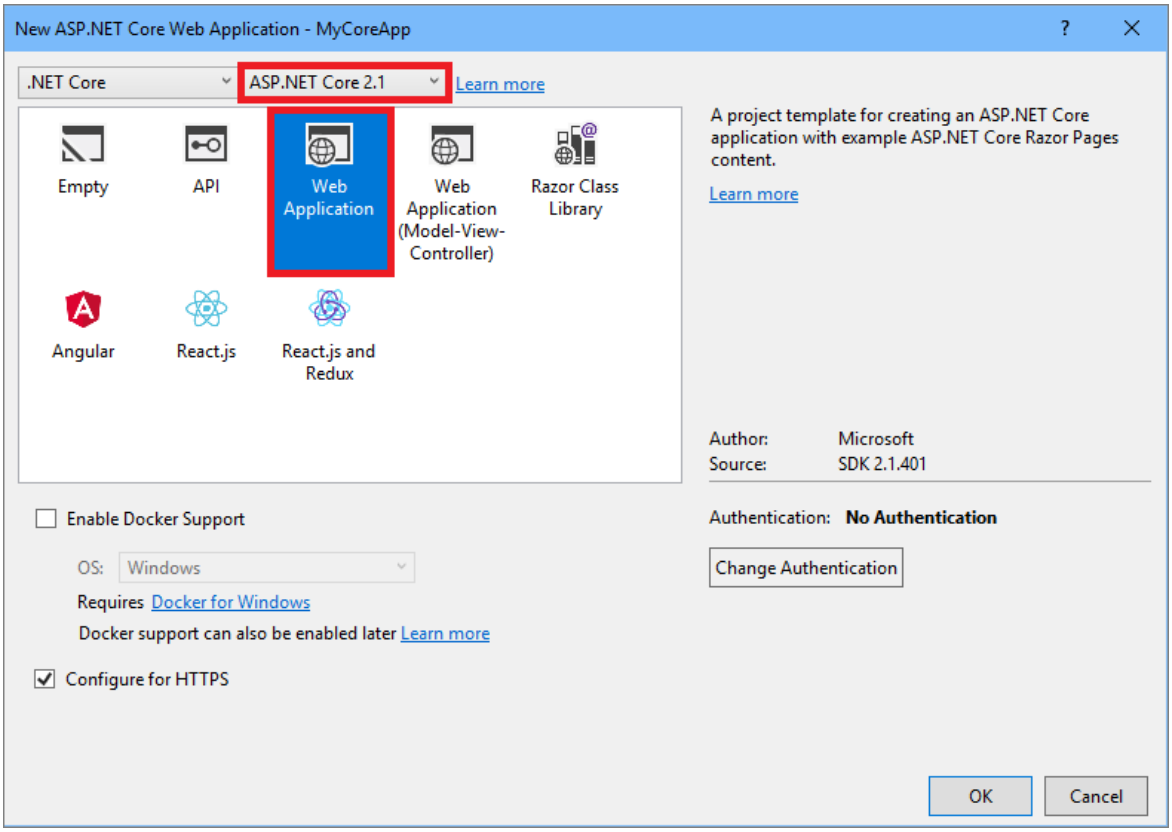
2. The Visual Studio Installer launches. Choose the **ASP.NET and web development** workload, and then choose **Modify**.

   (You might have to close Visual Studio before you can continue installing the new workload.)

Add a project template

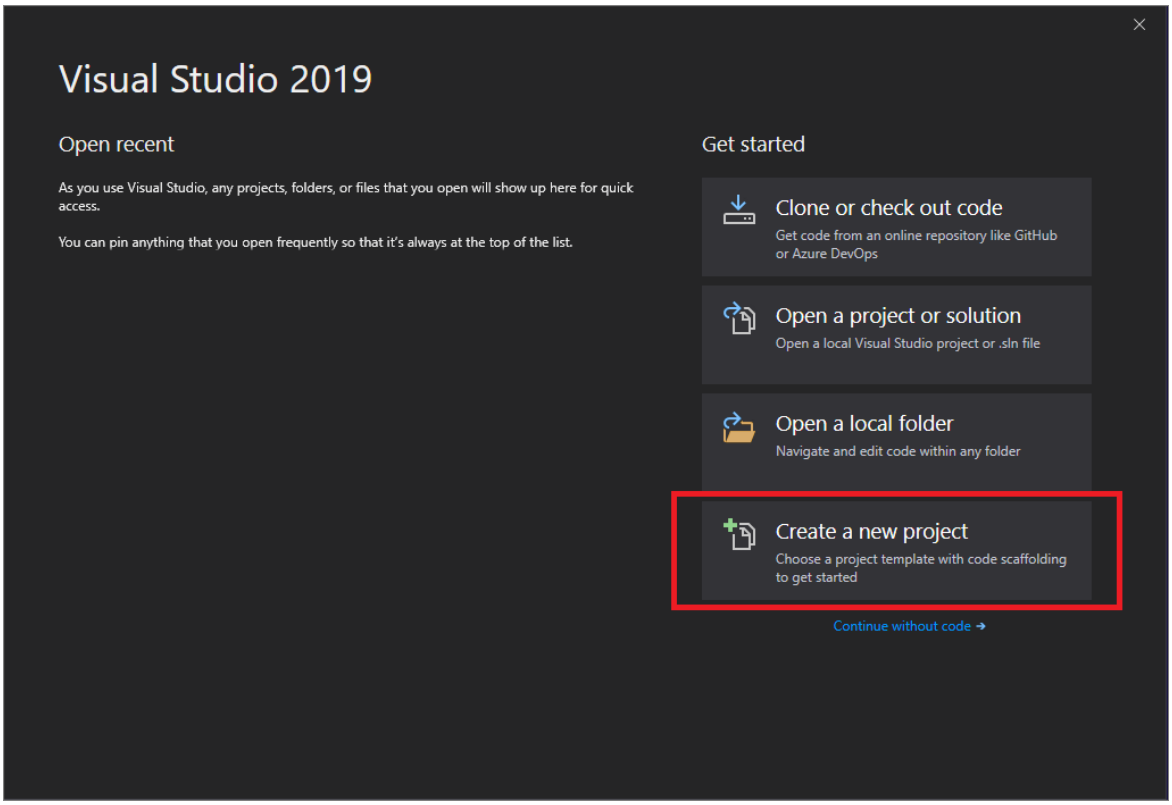
1. In the **New ASP.NET Core Web Application** dialog box, choose the **Web Application** project template.

2. Verify that **ASP.NET Core 2.1** appears in the top drop-down menu. Then, choose **OK**.



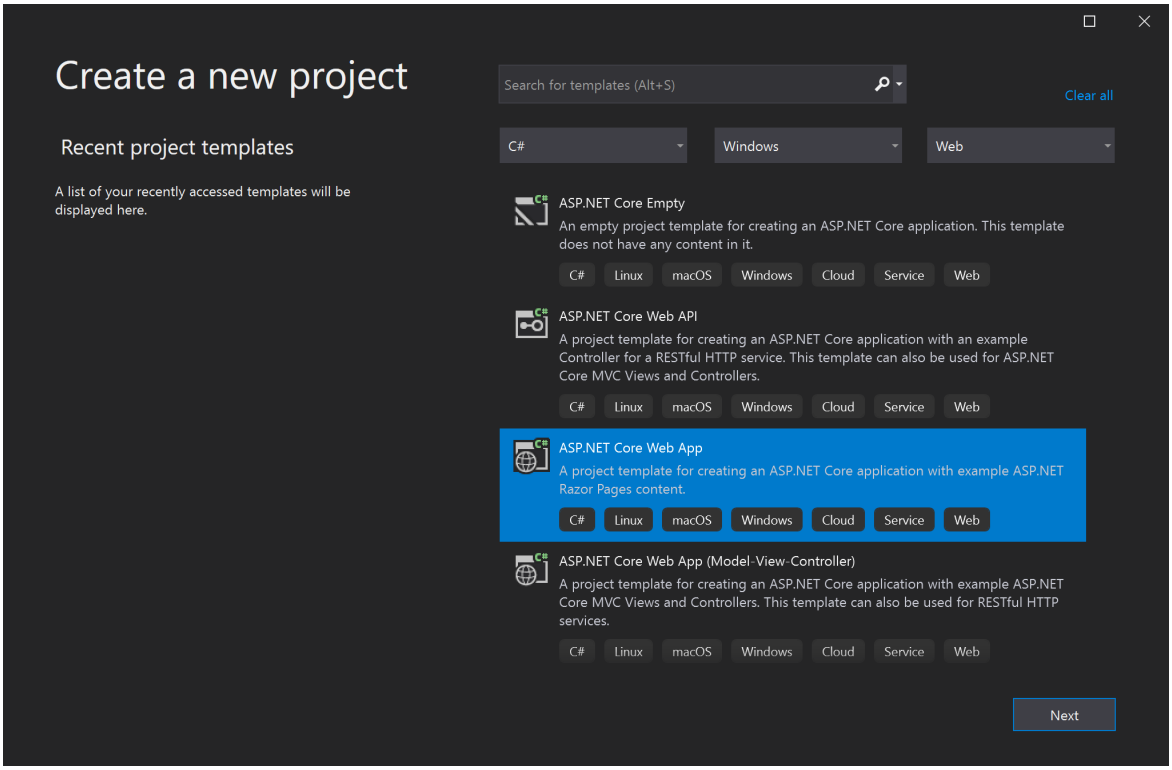
> **NOTE**
>
> If you don't see **ASP.NET Core 2.1** from the top drop-down menu, make sure that you are running the most recent release of Visual Studio. For more information about how to update your installation, see the Update Visual Studio to the most recent release page.

1. In the start window, choose **Create a new project**.



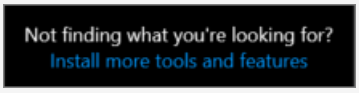
2. In the **Create a new project** window, choose **C#** from the Language list. Next, choose **Windows** from the Platform list, and **Web** from the project types list.

   After you apply the language, platform, and project type filters, choose the **ASP.NET Core Web App** template, and then choose **Next**.

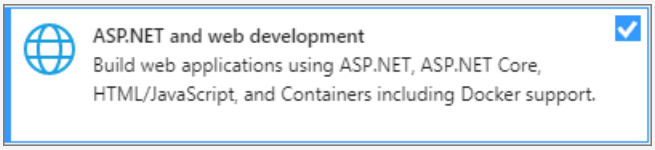


NOTE

If you don't see the **ASP.NET Core Web App** template, you can install it from the **Create a new project** window. In the **Not finding what you're looking for?** message, choose the **Install more tools and features** link.

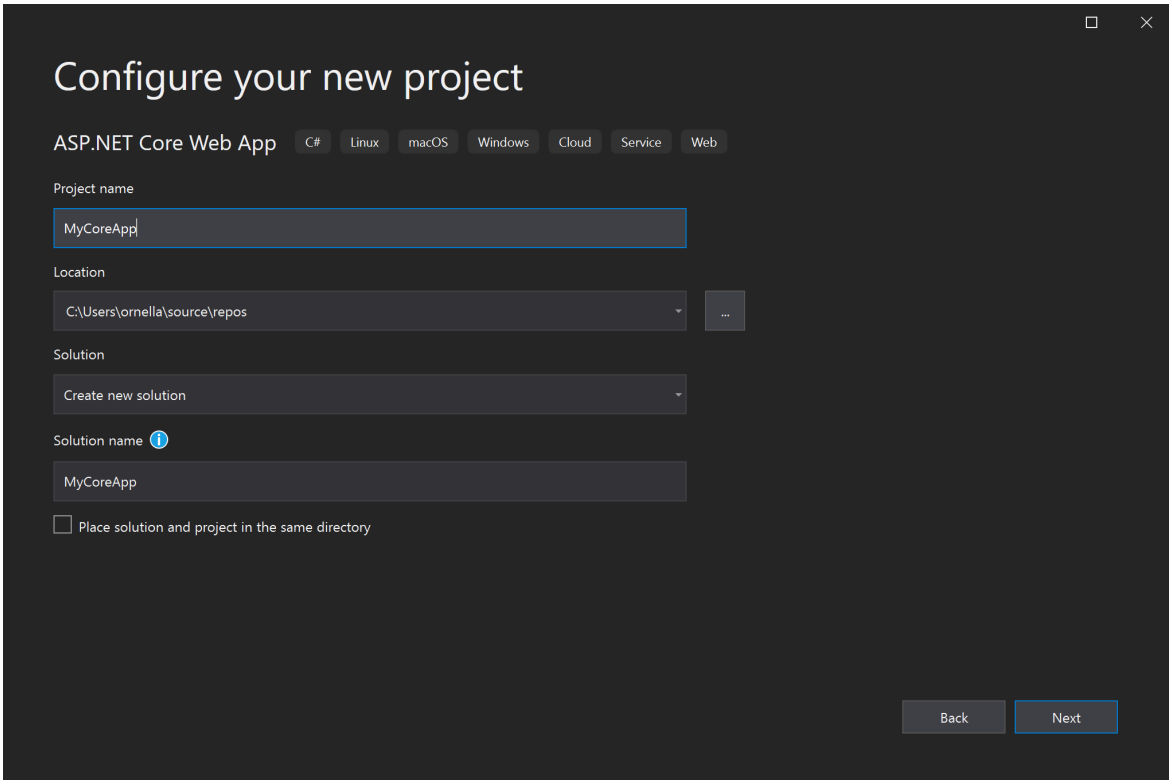


Then, in the Visual Studio Installer, choose the **ASP.NET and web development** workload.



After that, choose the **Modify** button in the Visual Studio Installer. If you're prompted to save your work, do so. Next, choose **Continue** to install the workload. Then, return to step 2 in this "Create a project" procedure.

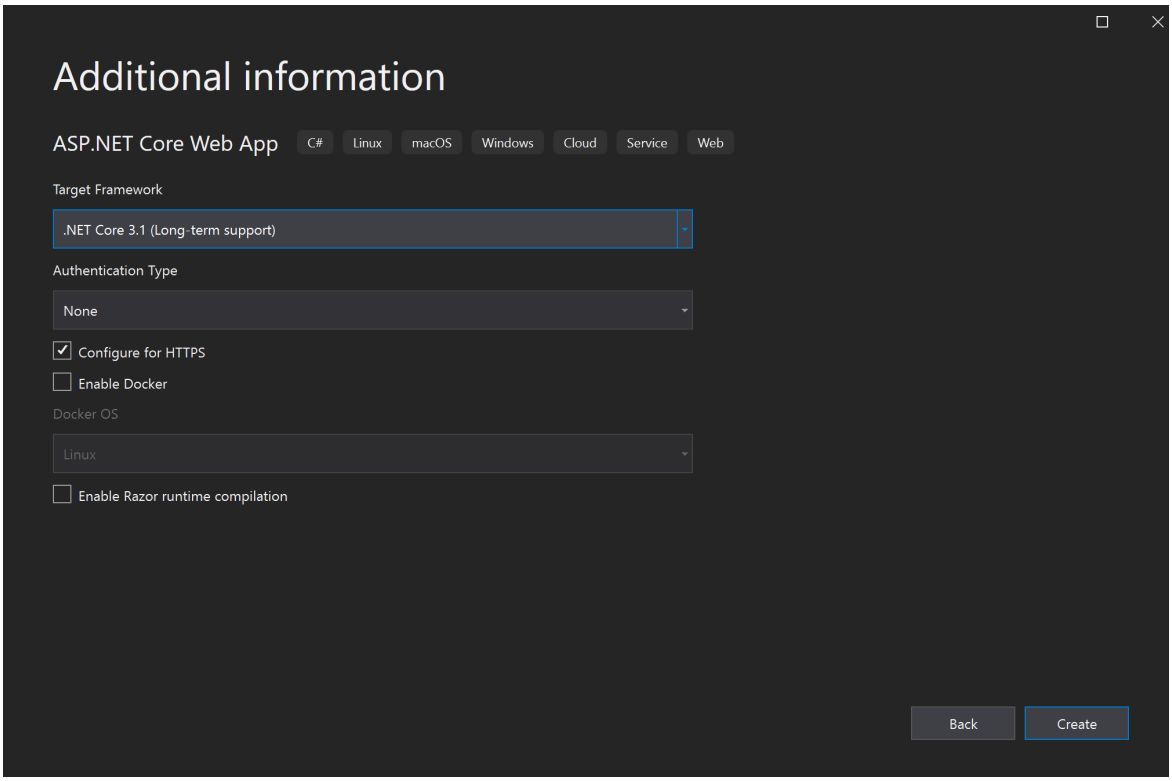
3. In the **Configure your new project** window, type or enter *MyCoreApp* in the **Project name** box. Then, choose **Next**.



4. In the **Additional information** window, verify that **.NET Core 3.1** appears in the top drop-down menu. Note that you can choose to enable Docker support by checking the box. You can also add authentication support by clicking the change Authentication button. From there you can choose from:

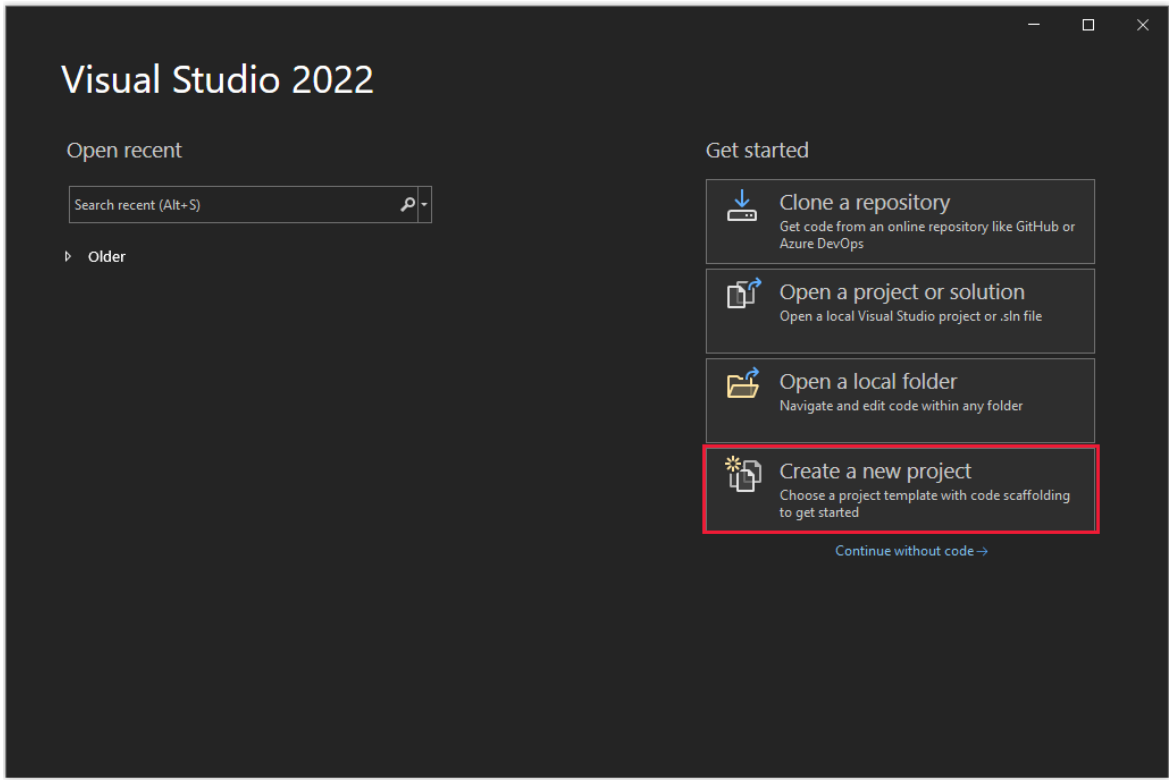
   - None: no authentication.
   - Individual accounts: these are stored in a local or Azure-based database.
   - Microsoft identity platform: this option uses Active Directory, Azure AD, or Microsoft 365 for authentication.
   - Windows: suitable for intranet applications.

   Leave the **Enable Docker** box unchecked, and select **None** for Authentication Type. Then, select **Create**.



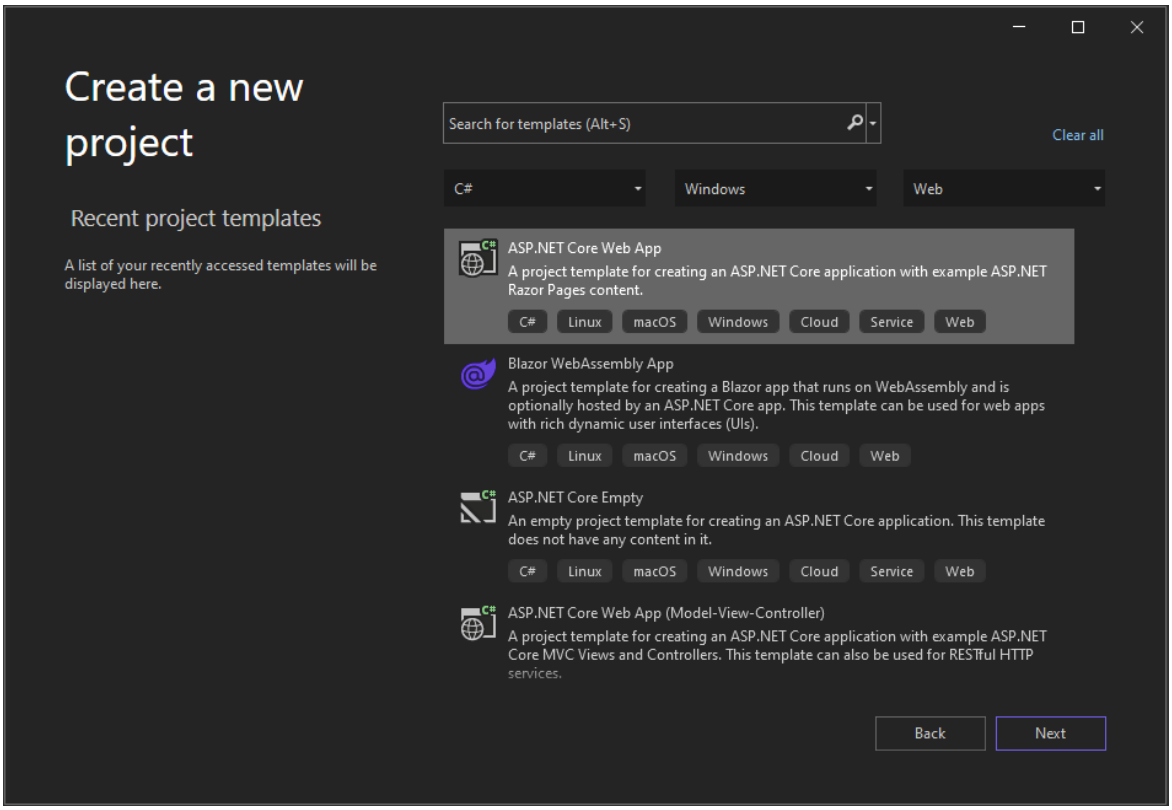
Visual Studio will open up your new project.

1. In the start window, choose **Create a new project**.



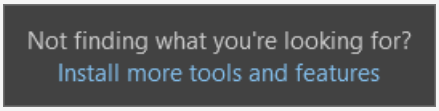
2. In the **Create a new project** window, choose **C#** from the Language list. Next, choose **Windows** from the Platform list, and **Web** from the project types list.

   After you apply the language, platform, and project type filters, choose the **ASP.NET Core Web App** template, and then choose **Next**.

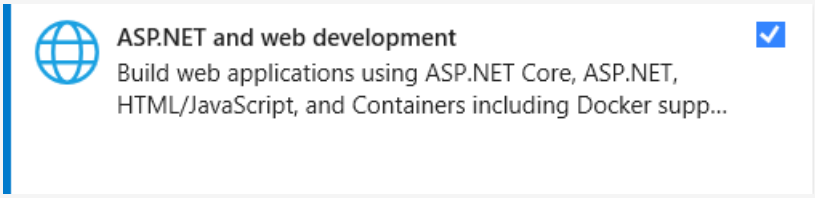


NOTE

If you don't see the **ASP.NET Core Web App** template, you can install it from the **Create a new project** window. In the **Not finding what you're looking for?** message, choose the **Install more tools and features** link.

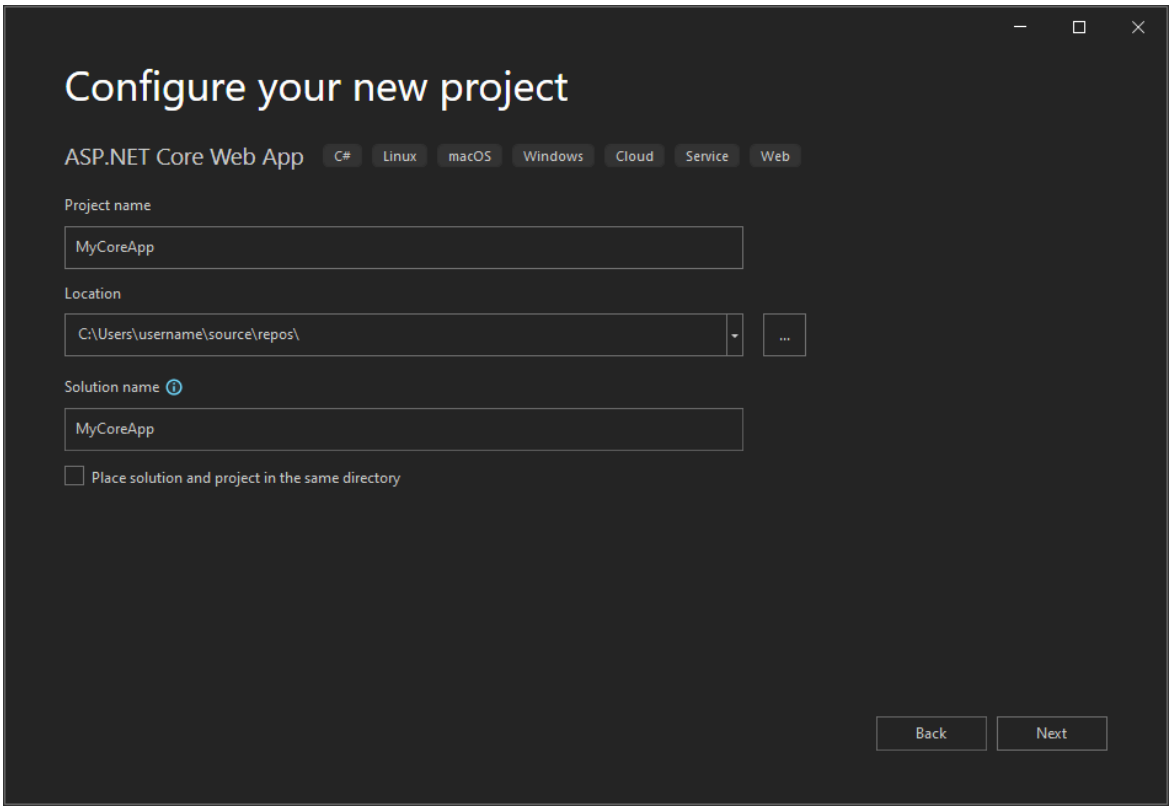


Then, in the Visual Studio Installer, choose the **ASP.NET and web development** workload.



After that, choose the **Modify** button in the Visual Studio Installer. If you're prompted to save your work, do so. Next, choose **Continue** to install the workload. Then, return to step 2 in this "Create a project" procedure.

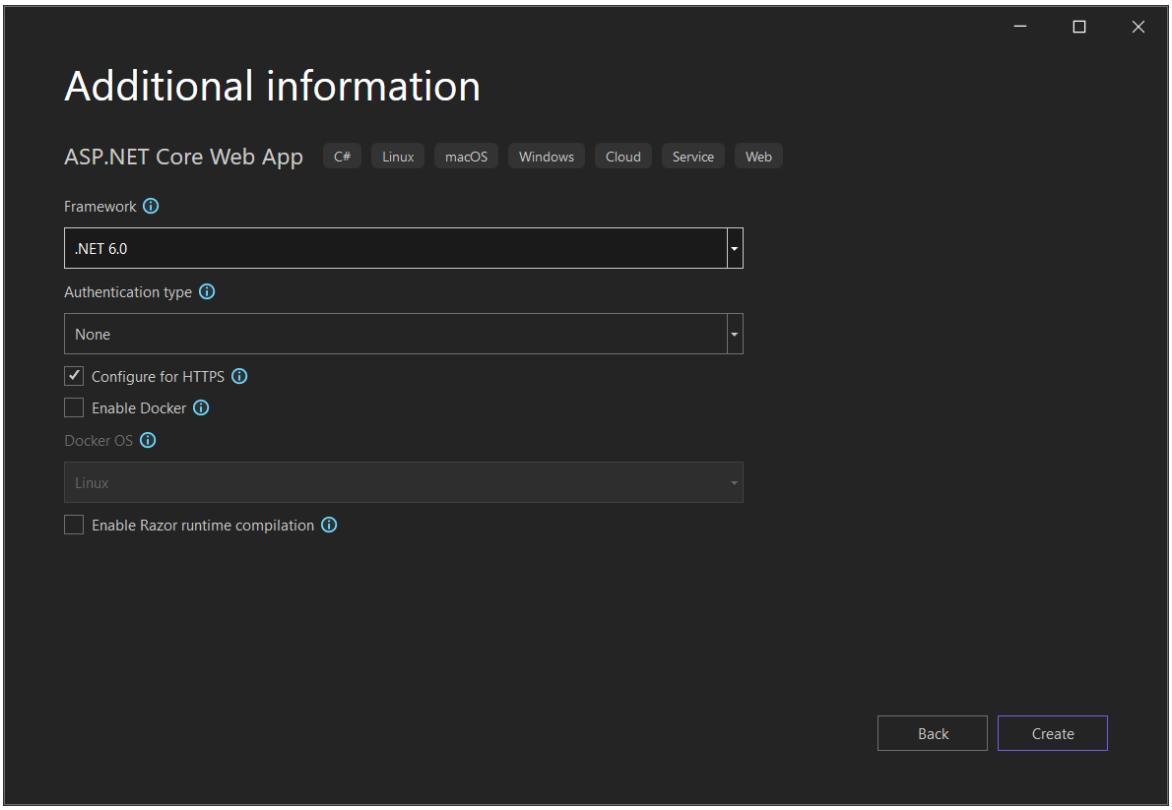
3. In the **Configure your new project** window, type or enter *MyCoreApp* in the **Project name** box. Then, choose **Next**.



4. In the **Additional information** window, verify that **.NET 6.0** appears in the **Framework** field. In this window, you can choose to enable Docker support by checking the box. You can also add authentication support by selecting a value from the **Authentication type** drop-down list. From there you can choose from:

   - None: no authentication.
   - Individual accounts: authentications that are stored in a local or Azure-based database.
   - Microsoft identity platform: this option uses Active Directory, Azure AD, or Microsoft 365 for authentication.
   - Windows: suitable for intranet applications.

   Leave the **Enable Docker** box unchecked, and select **None** for Authentication type. Then, select **Create**.



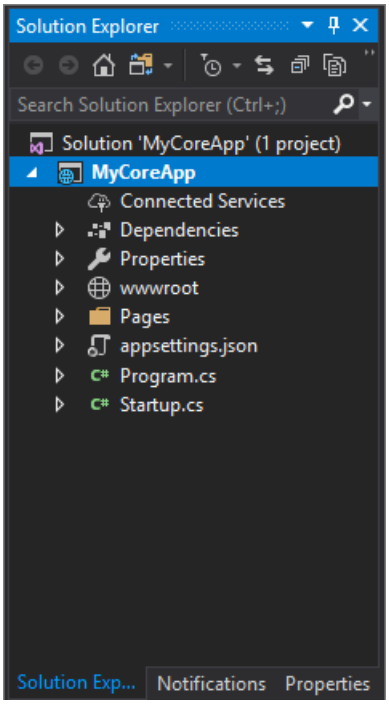
Visual Studio will open up your new project.

About your solution

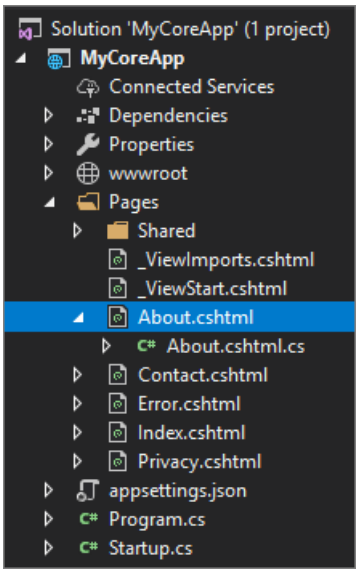
This solution follows the **Razor Page** design pattern. It's different than the Model-View-Controller (MVC) design pattern in that it's streamlined to include the model and controller code within the Razor Page itself.

Tour your solution

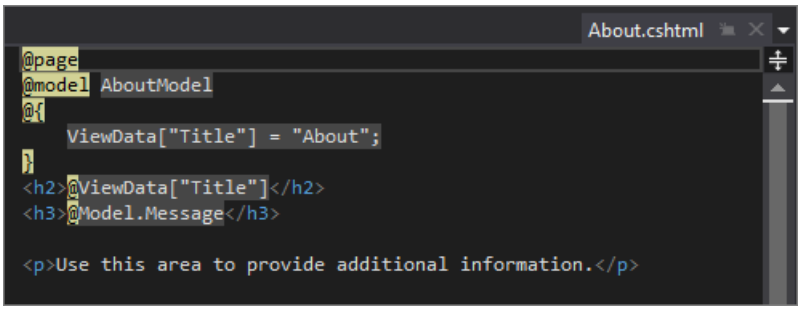
1. The project template creates a solution with a single ASP.NET Core project that is named *MyCoreApp*. Choose the **Solution Explorer** tab to view its contents.



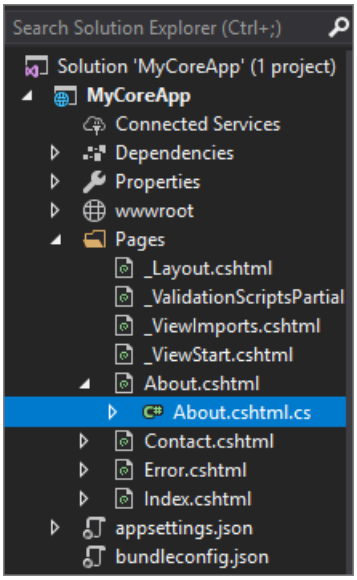
2. Expand the **Pages** folder, and then expand *About.cshtml*.



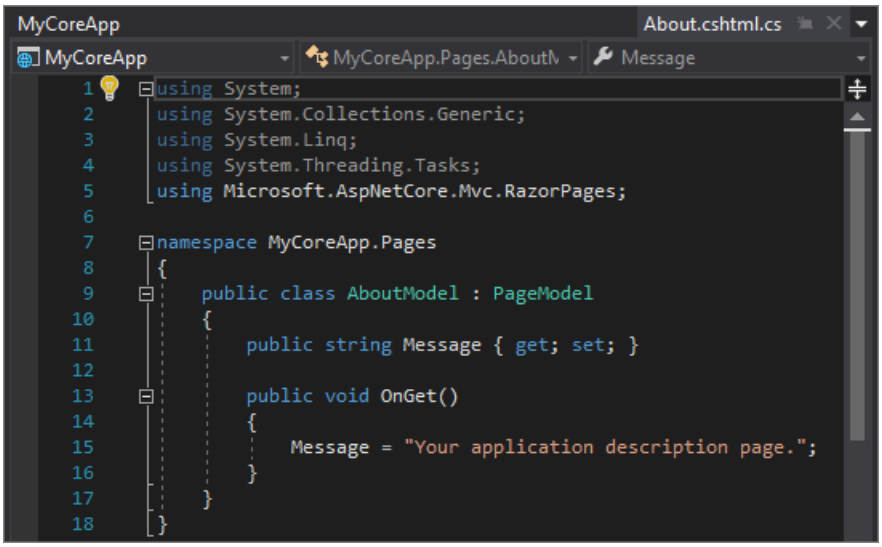
3. View the **About.cshtml** file in the code editor.



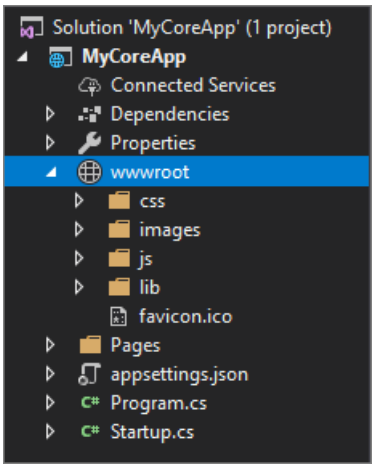
4. Choose the **About.cshtml.cs** file.



5. View the **About.cshtml.cs** file in the code editor.

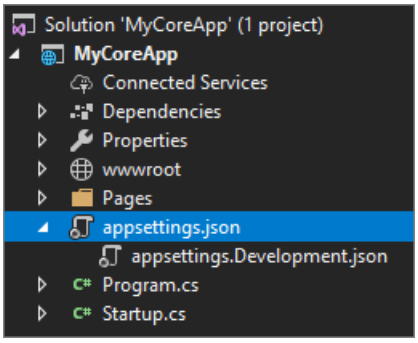


6. The project contains a **wwwroot** folder that is the root for your website. Expand the folder to view its contents.



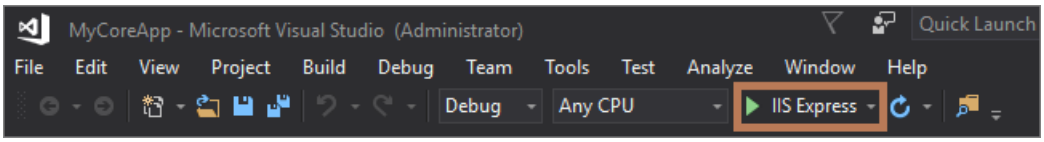
You can put static site content—such as CSS, images, and JavaScript libraries—directly in the paths where you want them.

7. The project also contains configuration files that manage the web app at run time. The default application configuration is stored in *appsettings.json*. However, you can override these settings by using *appsettings.Development.json*. Expand the **appsettings.json** file to view the **appsettings.Development.json** file.



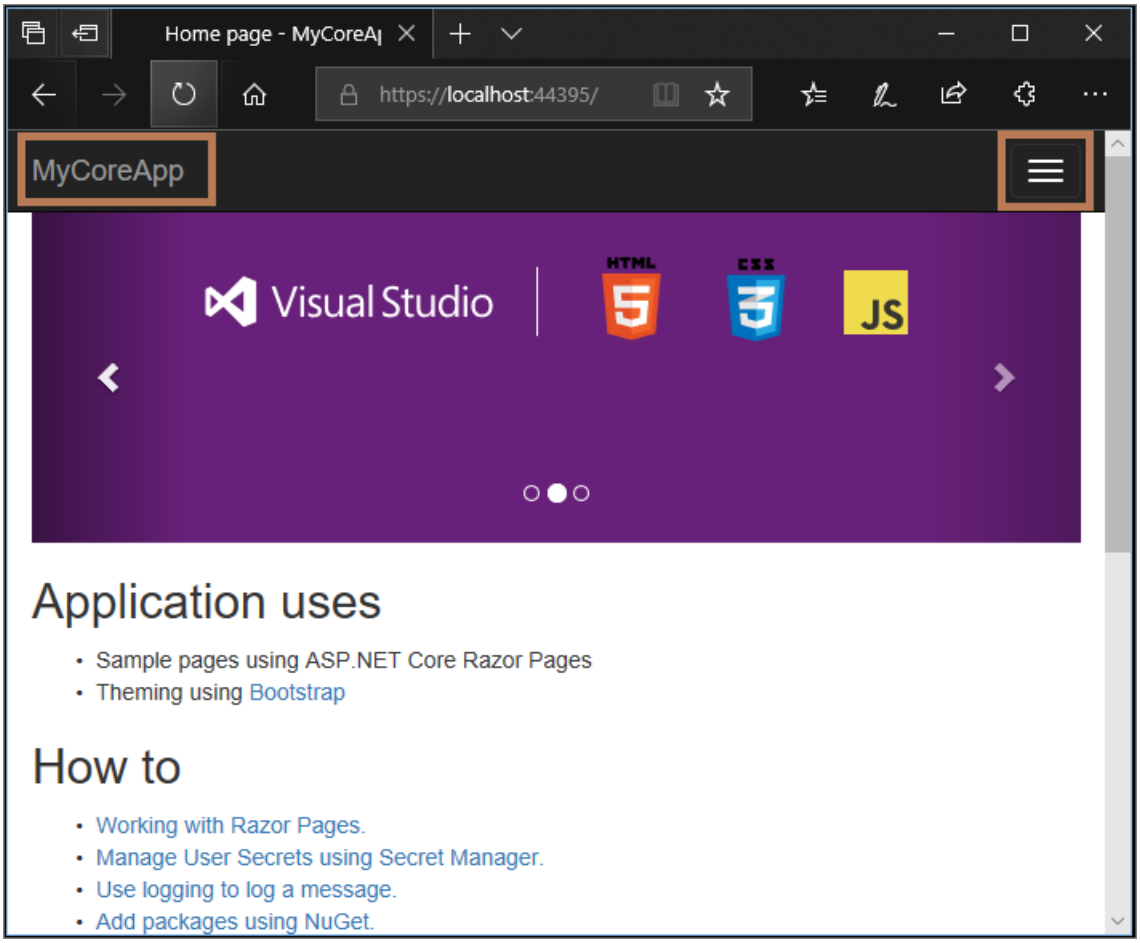
Run, debug, and make changes

1. Choose the **IIS Express** button in the IDE to build and run the app in Debug mode. (Alternatively, press **F5**, or choose **Debug** > **Start Debugging** from the menu bar.)

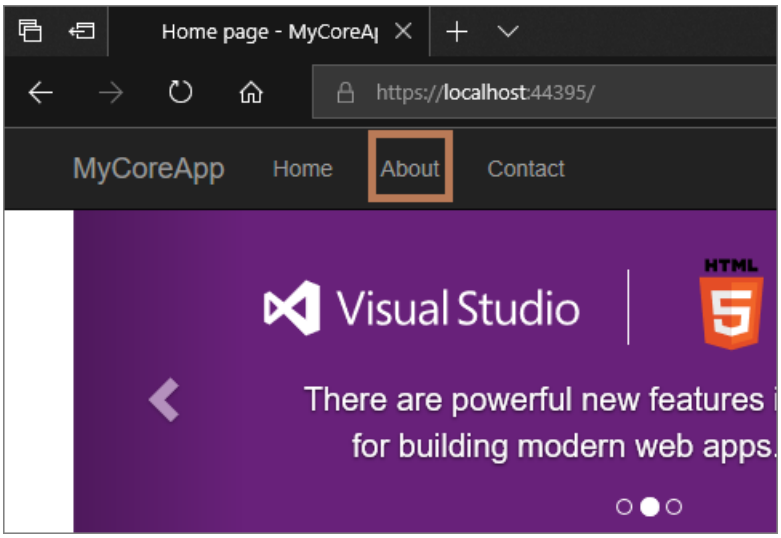


> **NOTE**
>
> If you get an error message that says **Unable to connect to web server 'IIS Express'**, close Visual Studio and then open it by using the **Run as administrator** option from the right-click or context menu. Then, run the application again.
>
> You might also get a message that asks if you want to accept an IIS SSL Express certificate. To view the code in a web browser, choose **Yes**, and then choose **Yes** if you receive a follow-up security warning message.

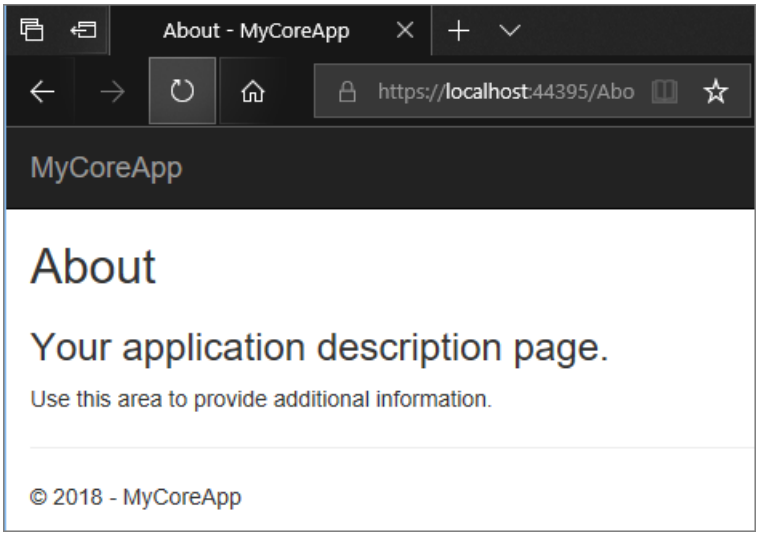
2. Visual Studio launches a browser window. You should then see **Home**, **About**, and **Contact** pages in the menu bar. (If you don't, choose the "hamburger" menu item to view them.)



3. Choose **About** from the menu bar.

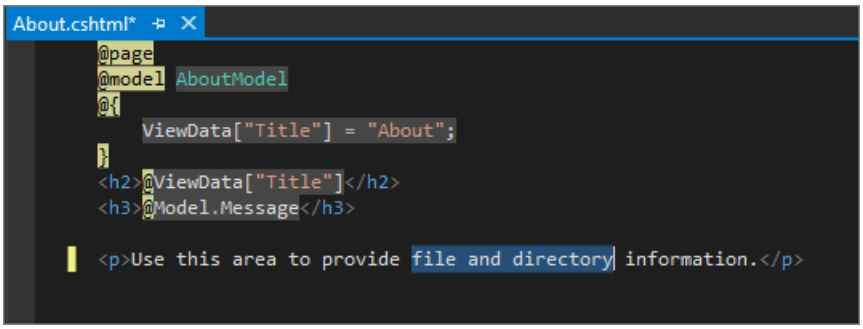


Among other things, the **About** page in the browser renders the text that is set in the *About.cshtml* file.



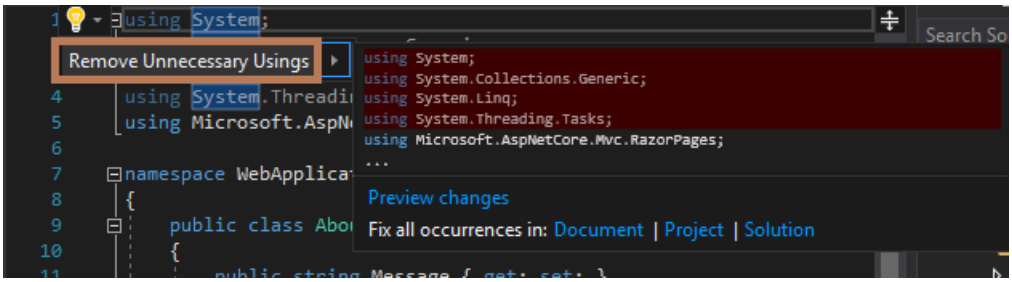
4. Return to Visual Studio, and then press **Shift+F5** to stop Debug mode. This also closes the project in the browser window.

5. In Visual Studio, choose **About.cshtml**. Then, delete the word *another* and in its place, add the words *file and directory*.



6. Choose **About.cshtml.cs**. Then, clean up the `using` directives at the top of the file by using the following shortcut:

   Choose any of the grayed-out `using` directives and a Quick Actions light bulb will appear just below the caret or in the left margin. Choose the light bulb, and then choose **Remove Unnecessary Usings**.

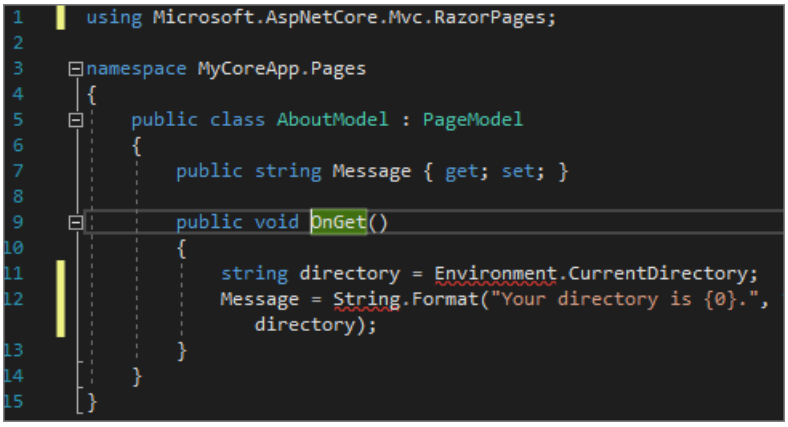


Visual Studio deletes the unnecessary `using` directives from the file.

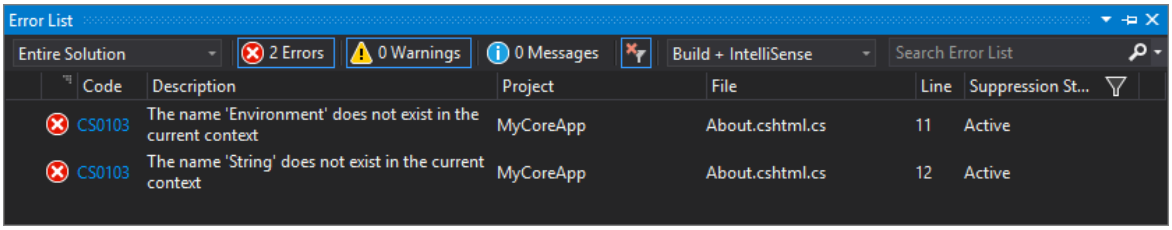
7. Next, in the `OnGet()` method, change the body to the following code:

```
public void OnGet()
{
    string directory = Environment.CurrentDirectory;
    Message = String.Format("Your directory is {0}.", directory);
}
```

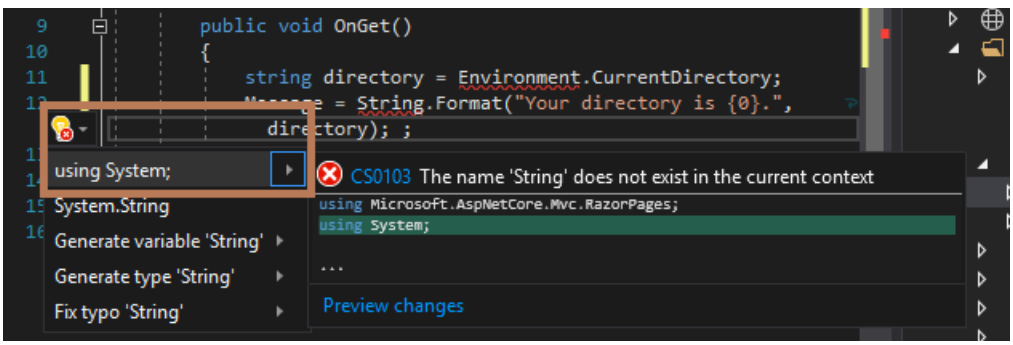
8. Notice that two wavy underlines appear under **Environment** and **String**. The wavy underlines appear because these types aren't in scope.



Open the **Error List** toolbar to see the same errors listed there. (If you don't see the **Error List** toolbar, choose **View** > **Error List** from the top menu bar.)

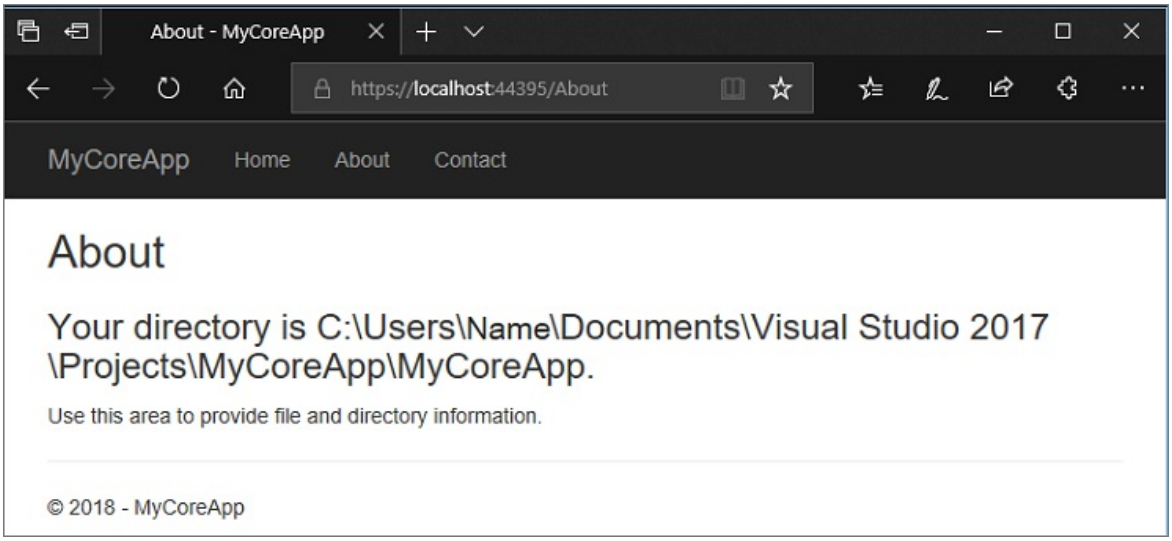


9. Let's fix this. In the code editor, place your cursor on either line that contains the error, and then choose the Quick Actions light bulb in the left margin. Then, from the drop-down menu, choose **using System;** to add this directive to the top of your file and resolve the errors.



10. Press **Ctrl+S** to save your changes, and then press **F5** to open your project in the web browser.

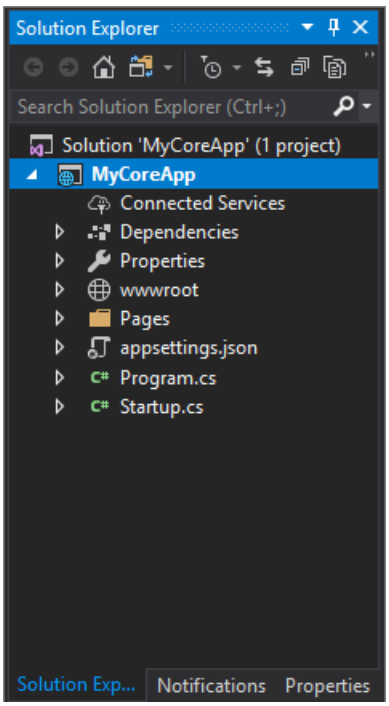
11. At the top of the web site, choose **About** to view your changes.



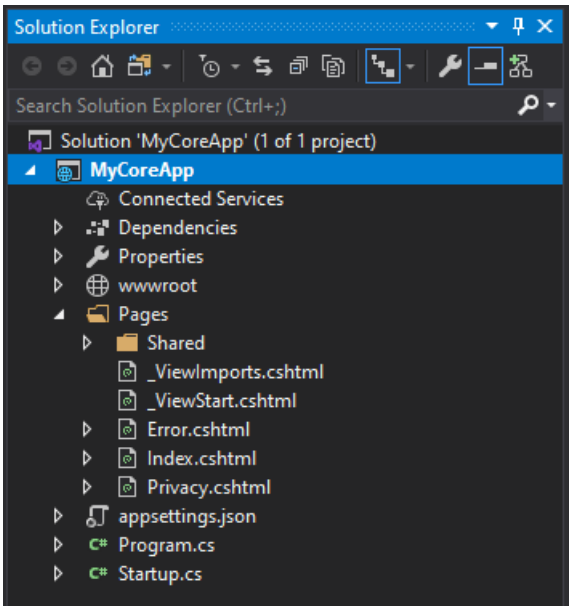
12. Close the web browser, press **Shift+F5** to stop Debug mode, and then close Visual Studio.

Tour your solution

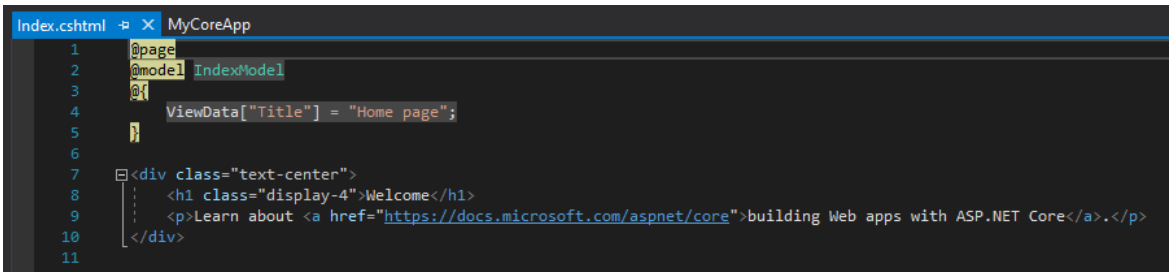
1. The project template creates a solution with a single ASP.NET Core project that is named *MyCoreApp*. Choose the **Solution Explorer** tab to view its contents.



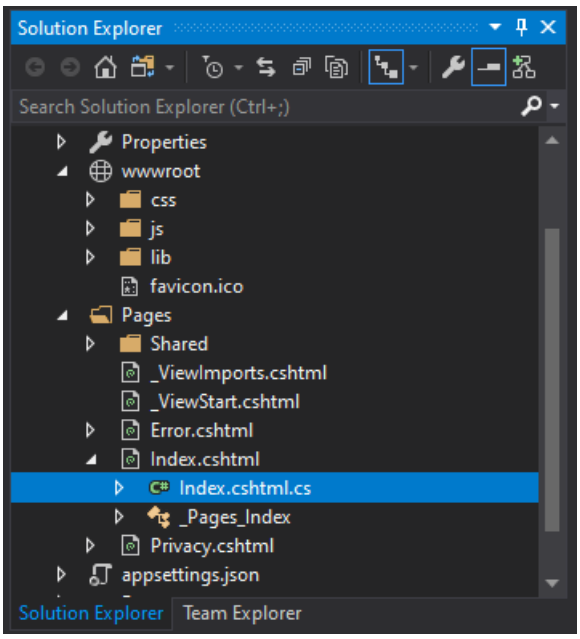
2. Expand the **Pages** folder.



3. View the **Index.cshtml** file in the code editor.



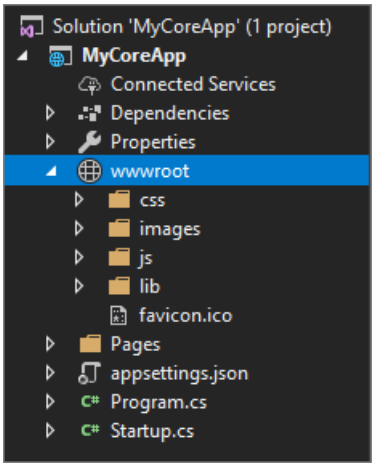
4. Each .cshtml file has an associated code file. To open the code file in the editor, expand the **Index.cshtml** node in Solution Explorer, and choose the **Index.cshtml.cs** file.



5. View the **Index.cshtml.cs** file in the code editor.

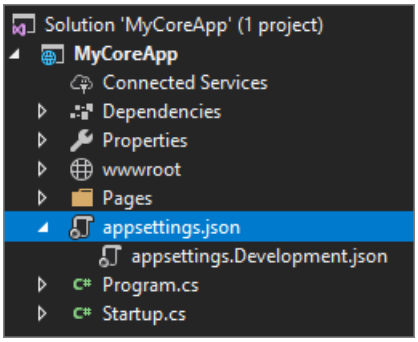
```
Index.cshtml.cs  Index.cshtml  MyCoreApp
MyCoreApp  MyCoreApp.Pages.IndexModel  _logger
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5  using Microsoft.AspNetCore.Mvc;
6  using Microsoft.AspNetCore.Mvc.RazorPages;
7  using Microsoft.Extensions.Logging;
8
9  namespace MyCoreApp.Pages
10 {
11     public class IndexModel : PageModel
12     {
13         private readonly ILogger<IndexModel> _logger;
14
15         public IndexModel(ILogger<IndexModel> logger)
16         {
17             _logger = logger;
18         }
19
20         public void OnGet()
21         {
22
23         }
24     }
25 }
26
```

6. The project contains a **wwwroot** folder that is the root for your website. Expand the folder to view its contents.



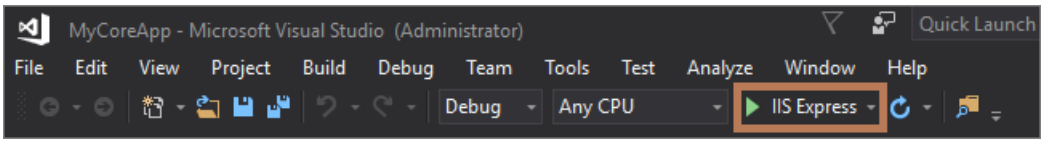
You can put static site content—such as CSS, images, and JavaScript libraries—directly in the paths where you want them.

7. The project also contains configuration files that manage the web app at run time. The default application configuration is stored in *appsettings.json*. However, you can override these settings by using *appsettings.Development.json*. Expand the **appsettings.json** file to view the **appsettings.Development.json** file.



Run, debug, and make changes

1. Choose the **IIS Express** button in the IDE to build and run the app in Debug mode. (Alternatively, press **F5**, or choose **Debug** > **Start Debugging** from the menu bar.)

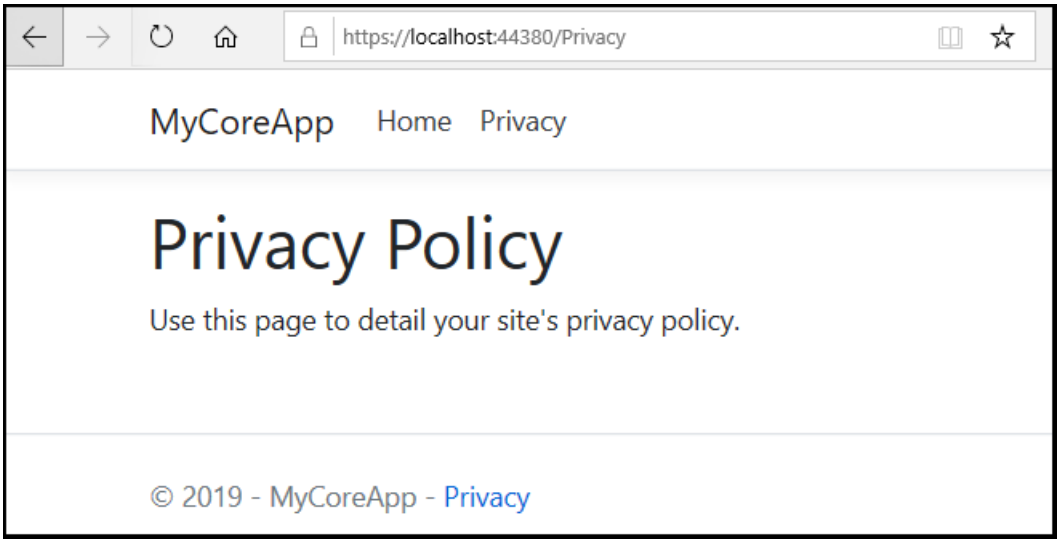


> **NOTE**
>
> If you get an error message that says **Unable to connect to web server 'IIS Express'**, close Visual Studio and then open it by using the **Run as administrator** option from the right-click or context menu. Then, run the application again.
>
> You might also get a message that asks if you want to accept an IIS SSL Express certificate. To view the code in a web browser, choose **Yes**, and then choose **Yes** if you receive a follow-up security warning message.

2. Visual Studio launches a browser window. You should then see **Home**, and **Privacy** pages in the menu bar.

3. Choose **Privacy** from the menu bar.

   The **Privacy** page in the browser renders the text that is set in the *Privacy.cshtml* file.



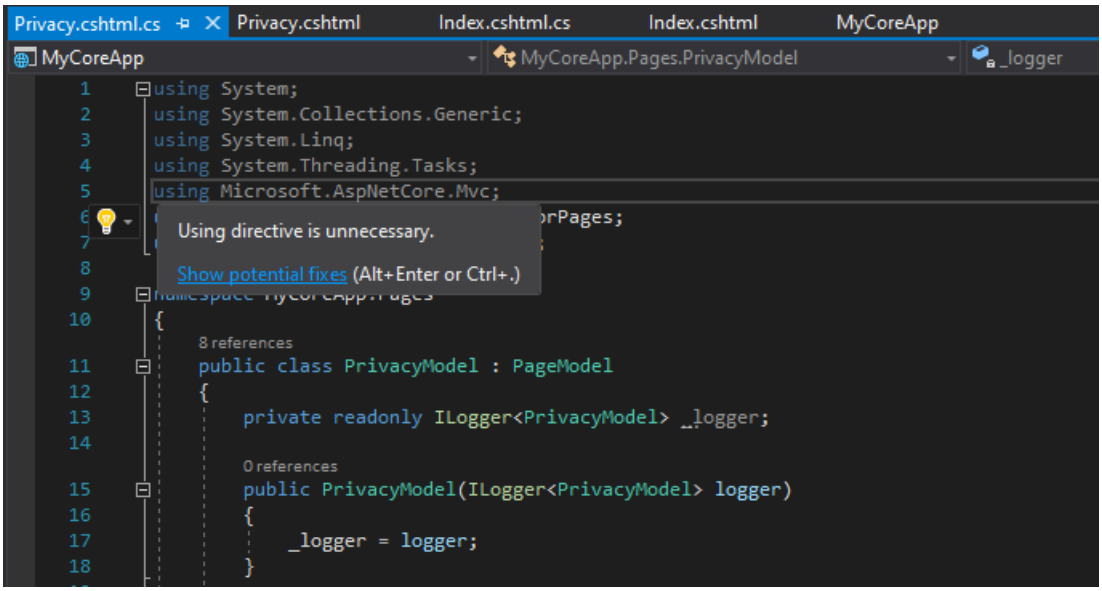
4. Return to Visual Studio, and then press **Shift+F5** to stop Debug mode. This also closes the project in the browser window.

5. In Visual Studio, open **Privacy.cshtml** for editing. Then, delete the words *Use this page to detail your site's privacy policy* and in its place, add the words *This page is under construction as of @ViewData["TimeStamp"]*.

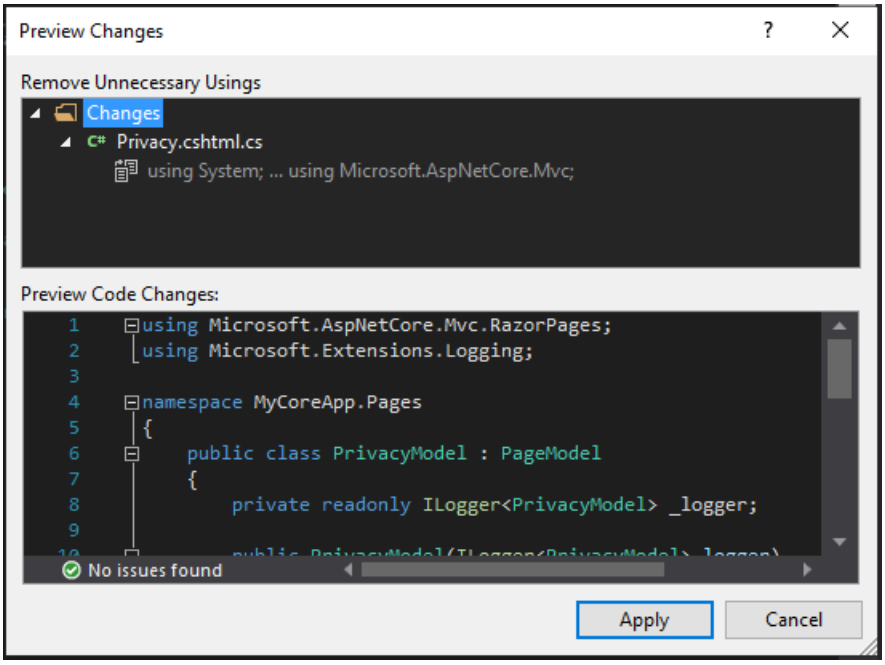
```
@page
@model PrivacyModel
@{
    ViewData["Title"] = "Privacy Policy";
}
<h1>@ViewData["Title"]</h1>

<p>This page is under construction as of @ViewData["TimeStamp"].</p>
```

6. Now, let's make a code change. Choose **Privacy.cshtml.cs**. Then, clean up the `using` directives at the top of the file by using the following shortcut:

Choose any of the grayed-out `using` directives and a Quick Actions light bulb will appear just below the caret or in the left margin. Choose the light bulb, and then hover over **Remove unnecessary usings**.



Now choose **Preview changes** to see what will change.

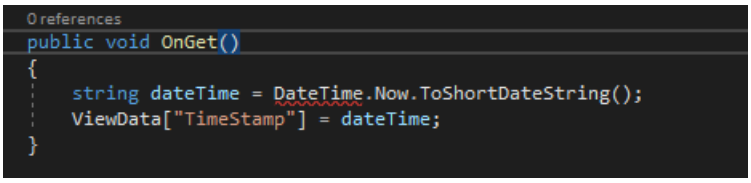


Choose **Apply**. Visual Studio deletes the unnecessary `using` directives from the file.

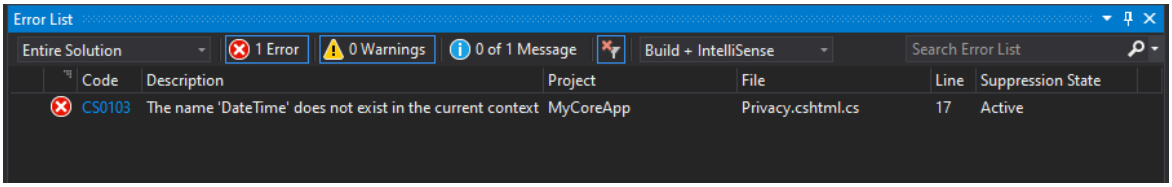
7. Next, in the `OnGet()` method, change the body to the following code:

```
public void OnGet()
{
   string dateTime = DateTime.Now.ToShortDateString();
   ViewData["TimeStamp"] = dateTime;
}
```

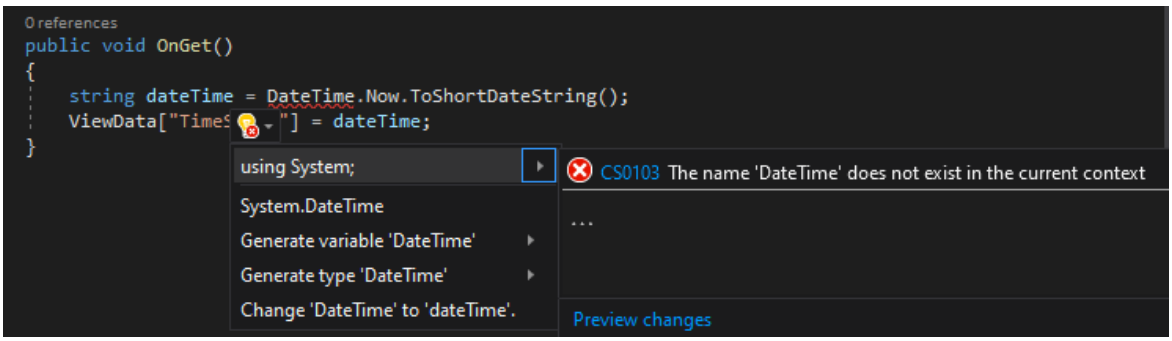
8. Notice that two wavy underlines appear under **DateTime**. The wavy underlines appear because this type isn't in scope.



Open the **Error List** toolbar to see the same errors listed there. (If you don't see the **Error List** toolbar, choose **View** > **Error List** from the top menu bar.)



9. Let's fix this. In the code editor, place your cursor on either line that contains the error, and then choose the Quick Actions light bulb in the left margin. Then, from the drop-down menu, choose **using System;** to add this directive to the top of your file and resolve the errors.

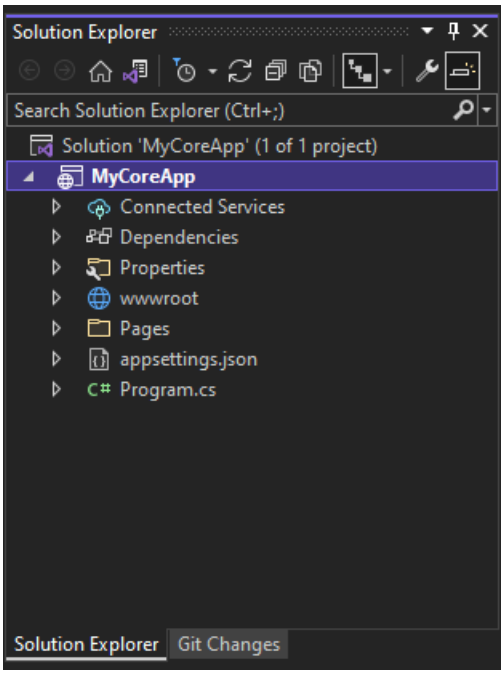


10. Press **F5** to open your project in the web browser.
11. At the top of the web site, choose **Privacy** to view your changes.

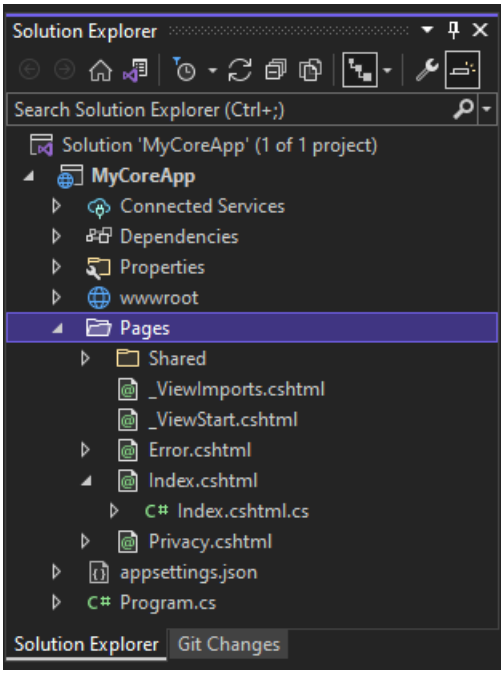
12. Close the web browser, press **Shift+F5** to stop Debug mode, and then close Visual Studio.

Tour your solution

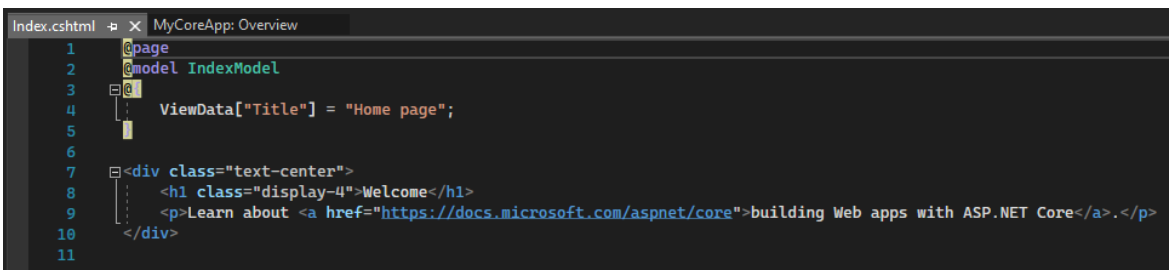
1. The project template creates a solution with a single ASP.NET Core project that is named *MyCoreApp*. Choose the **Solution Explorer** tab to view its contents.



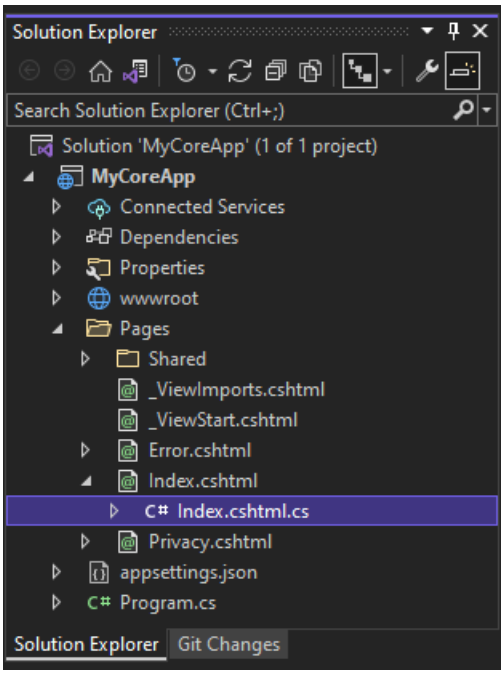
2. Expand the **Pages** folder.



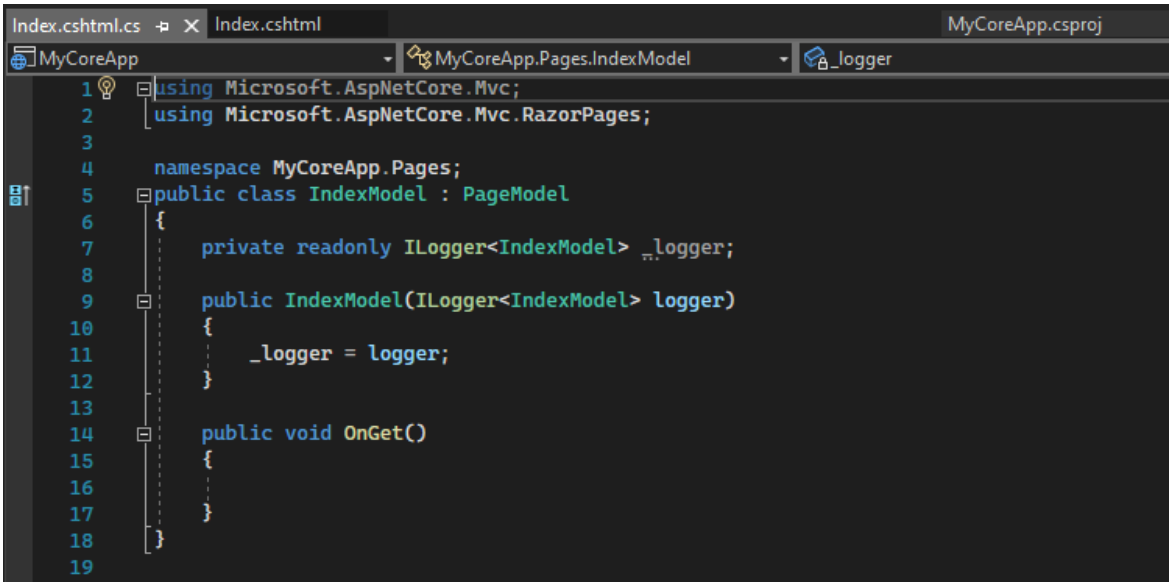
3. View the **Index.cshtml** file in the code editor.



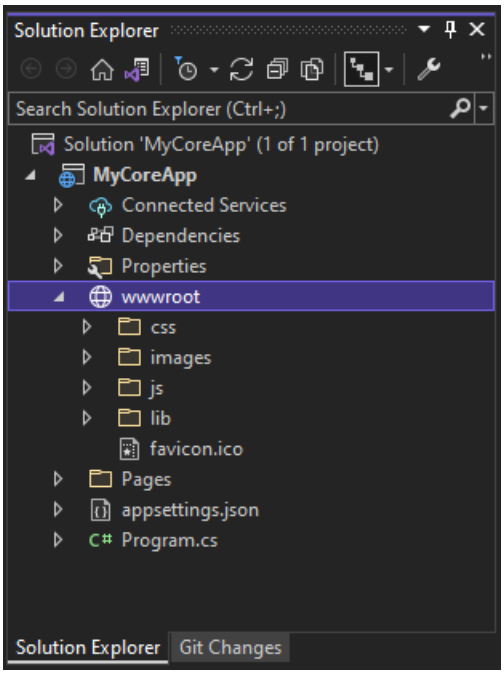
4. Each .cshtml file has an associated code file. To open the code file in the editor, expand the **Index.cshtml** node in Solution Explorer, and choose the **Index.cshtml.cs** file.



5. View the **Index.cshtml.cs** file in the code editor.

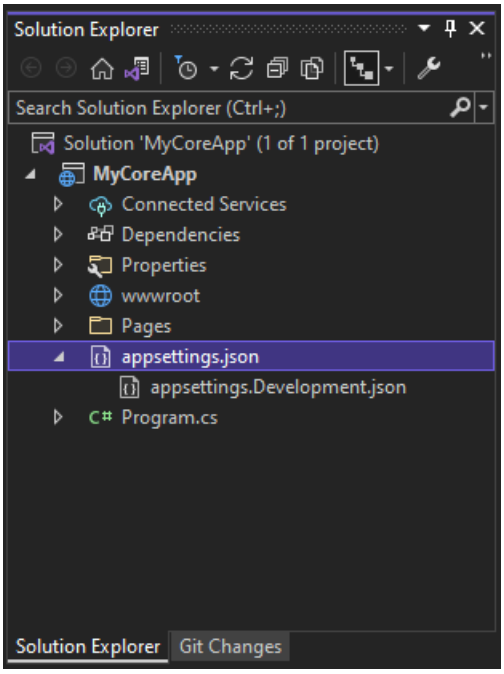


6. The project contains a **wwwroot** folder that is the root for your website. Expand the folder to view its contents.



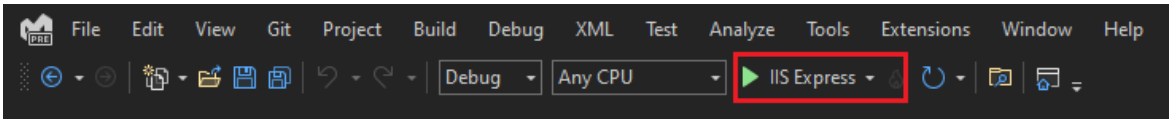
You can put static site content—such as CSS, images, and JavaScript libraries—directly in the paths where you want them.

7. The project also contains configuration files that manage the web app at run time. The default application configuration is stored in *appsettings.json*. However, you can override these settings by using *appsettings.Development.json*. Expand the **appsettings.json** file to view the **appsettings.Development.json** file.



Run, debug, and make changes

1. Choose the **IIS Express** button in the IDE to build and run the app in Debug mode. (Alternatively, press **F5**, or choose **Debug** > **Start Debugging** from the menu bar.)

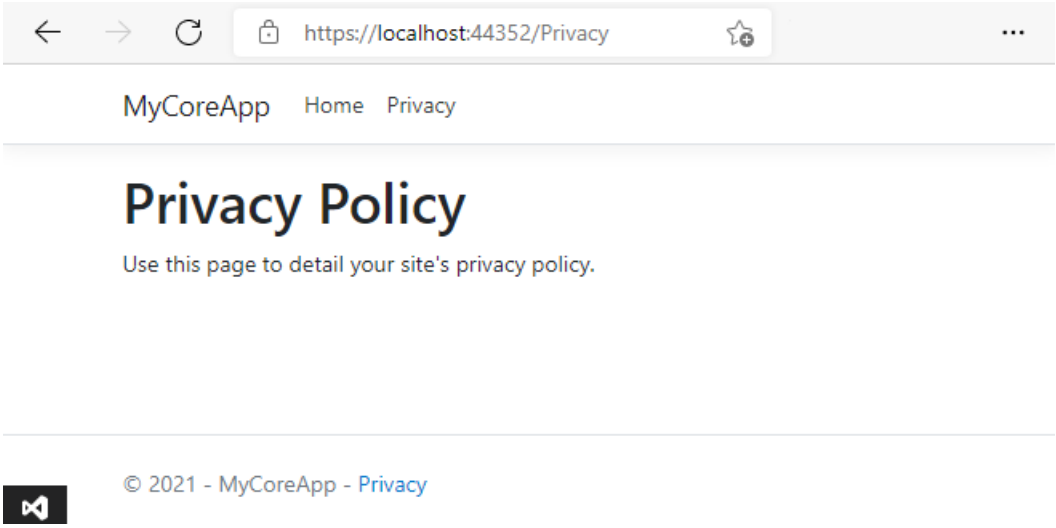


> **NOTE**
>
> If you get an error message that says **Unable to connect to web server 'IIS Express'**, close Visual Studio and then open it by using the **Run as administrator** option from the right-click or context menu. Then, run the application again.
>
> You might also get a message that asks if you want to accept an IIS SSL Express certificate. To view the code in a web browser, choose **Yes**, and then choose **Yes** if you receive a follow-up security warning message.

2. Visual Studio launches a browser window. You should then see **Home**, and **Privacy** pages in the menu bar.

3. Choose **Privacy** from the menu bar.

   The **Privacy** page in the browser renders the text that is set in the *Privacy.cshtml* file.



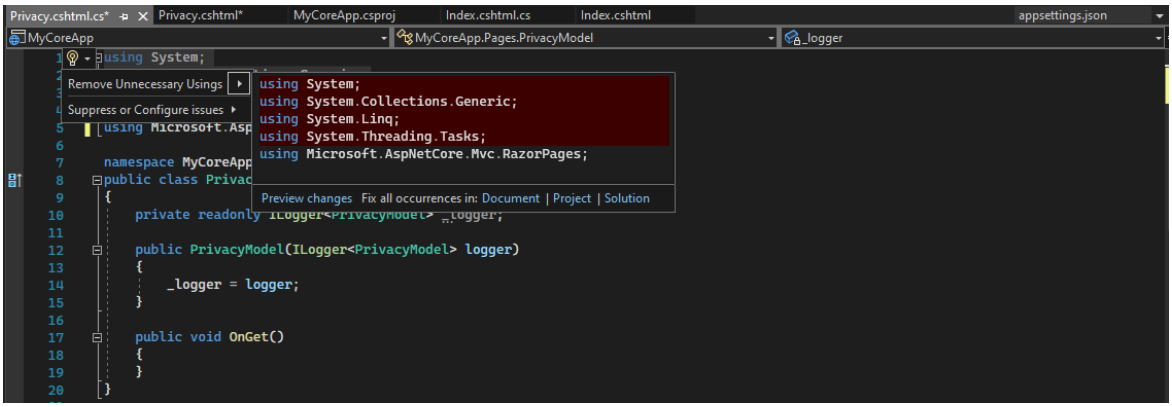
4. Return to Visual Studio, and then press **Shift+F5** to stop Debug mode. This also closes the project in the browser window.

5. In Visual Studio, open **Privacy.cshtml** for editing. Then, delete the words *Use this page to detail your site's privacy policy* and in its place, add the words *This page is under construction as of @ViewData["TimeStamp"]*.

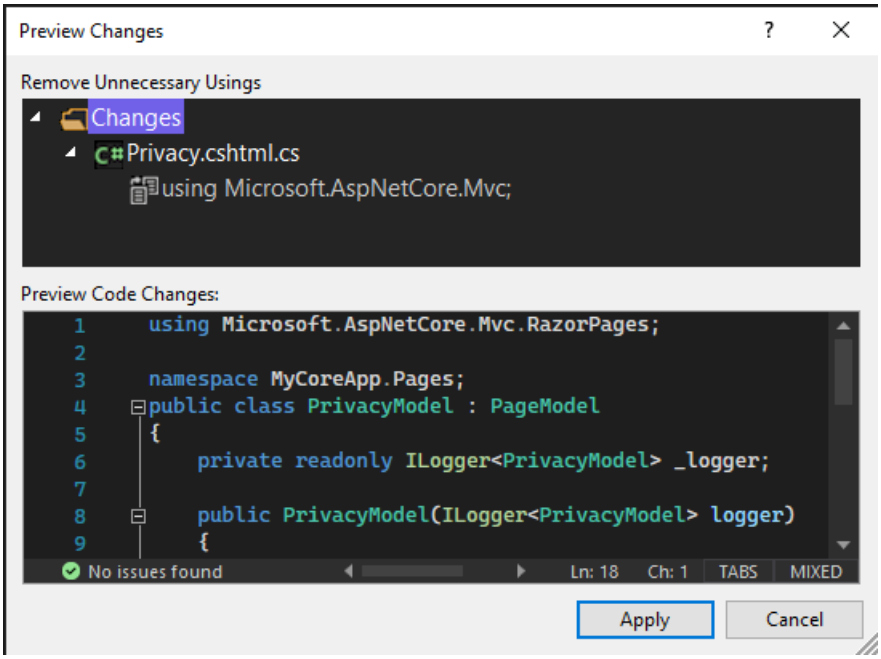
```
@page
@model PrivacyModel
@{
    ViewData["Title"] = "Privacy Policy";
}
<h1>@ViewData["Title"]</h1>

<p>This page is under construction as of @ViewData["Timestamp"].</p>
```

6. Now, let's make a code change. Choose **Privacy.cshtml.cs**. Then, clean up the `using` directives at the top of the file by selecting the following shortcut:

   Choose any of the grayed-out `using` directives and a Quick Actions light bulb will appear just below the caret or in the left margin. Choose the light bulb, and then hover over **Remove unnecessary usings**.



Now choose **Preview changes** to see what will change.



Choose **Apply**. Visual Studio deletes the unnecessary `using` directives from the file.

7. Next, create a string for the current date that is formatted for your culture or region by using the DateTime.ToString method.

   - The first argument for the method specifies how the the date should be displayed. This example uses the format specifier (`d`) which indicates the short date format.
   - The second argument is the CultureInfo object that specifies the culture or region for the date. This argument determines, among other things, the language of any words in the date, and the type of separators used.

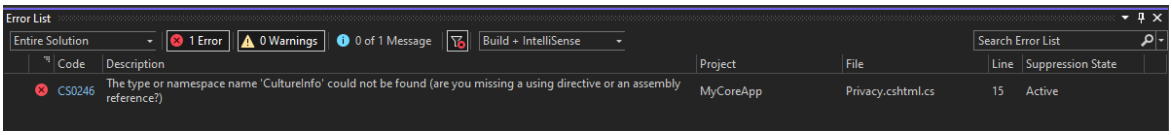
Change the body of the `OnGet()` method to the following code:

```
public void OnGet()
{
   string dateTime = DateTime.Now.ToString("d", new CultureInfo("en-US"));
   ViewData["TimeStamp"] = dateTime;
}
```

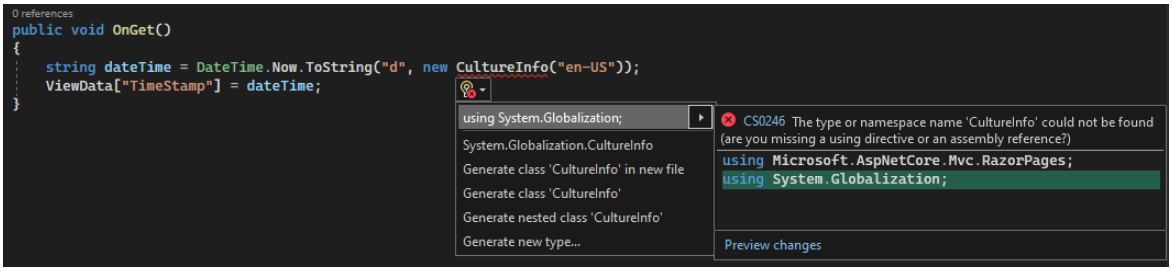
1. Notice that two wavy underlines appear under **CultureInfo**. The wavy underlines appear because this type isn't in scope.

```
0 references
public void OnGet()
{
    string dateTime = DateTime.Now.ToString("d", new CultureInfo("en-US"));
    ViewData["TimeStamp"] = dateTime;
}
```

Open the **Error List** toolbar to see the same errors listed there. (If you don't see the **Error List** toolbar, choose **View** > **Error List** from the top menu bar.)

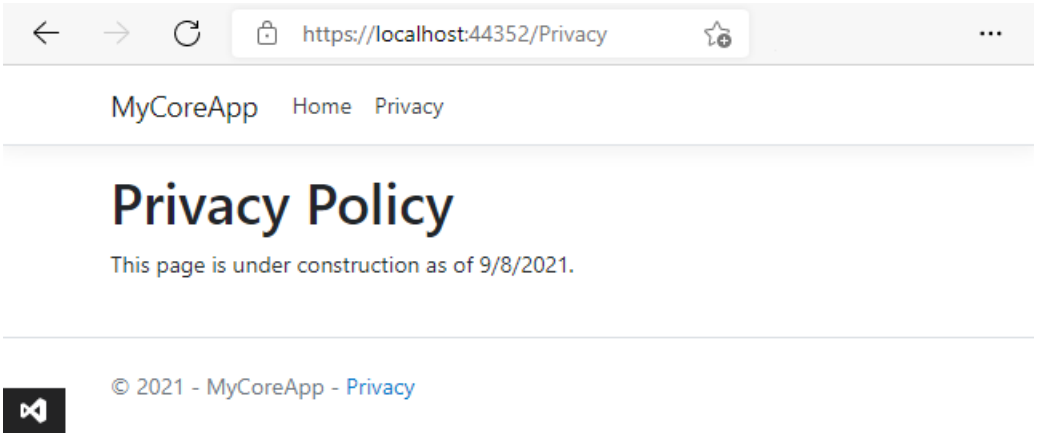


2. Let's fix this. In the code editor, place your cursor on either line that contains the error, and then choose the Quick Actions light bulb in the left margin. Then, from the drop-down menu, choose **using System.Globalization;** to add the directive to the top of your file and resolve the errors.



3. Press **F5** to open your project in the web browser.

4. At the top of the web site, choose **Privacy** to view your changes.



5. Close the web browser, press **Shift+F5** to stop Debug mode, and then close Visual Studio.

Quick answers FAQ

Here's a quick FAQ to highlight some key concepts.

What is C#?

C# is a type-safe and object-oriented programming language that's designed to be both robust and easy to learn.

What is ASP.NET Core?

ASP.NET Core is an open-source and cross-platform framework for building internet-connected applications, such as web apps and services. ASP.NET Core apps can run on either .NET Core or the .NET Framework. You can develop and run your ASP.NET Core apps cross-platform on Windows, Mac, and Linux. ASP.NET Core is open source at GitHub.

What is Visual Studio?

Visual Studio is an integrated development suite of productivity tools for developers. Think of it as a program you can use to create programs and applications.

Next steps

Congratulations on completing this tutorial! We hope you learned a little bit about C#, ASP.NET Core, and the Visual Studio IDE. To learn more about creating a web app or website with C# and ASP.NET, continue with the following tutorial:

Create a Razor Pages web app with ASP.NET Core

Or, learn how to containerize your web app with Docker:

Container Tools in Visual Studio

See also

Publish your web app to Azure App Service by using Visual Studio

Tutorial: Create your first Universal Windows Platform application in Visual Studio with XAML and C#

10/28/2021 • 7 minutes to read • Edit Online

In this introduction to the Visual Studio integrated development environment (IDE), you'll create a "Hello World" app that runs on any Windows 10 or later device. To do so, you'll use a Universal Windows Platform (UWP) project template, Extensible Application Markup Language (XAML), and the C# programming language.

If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

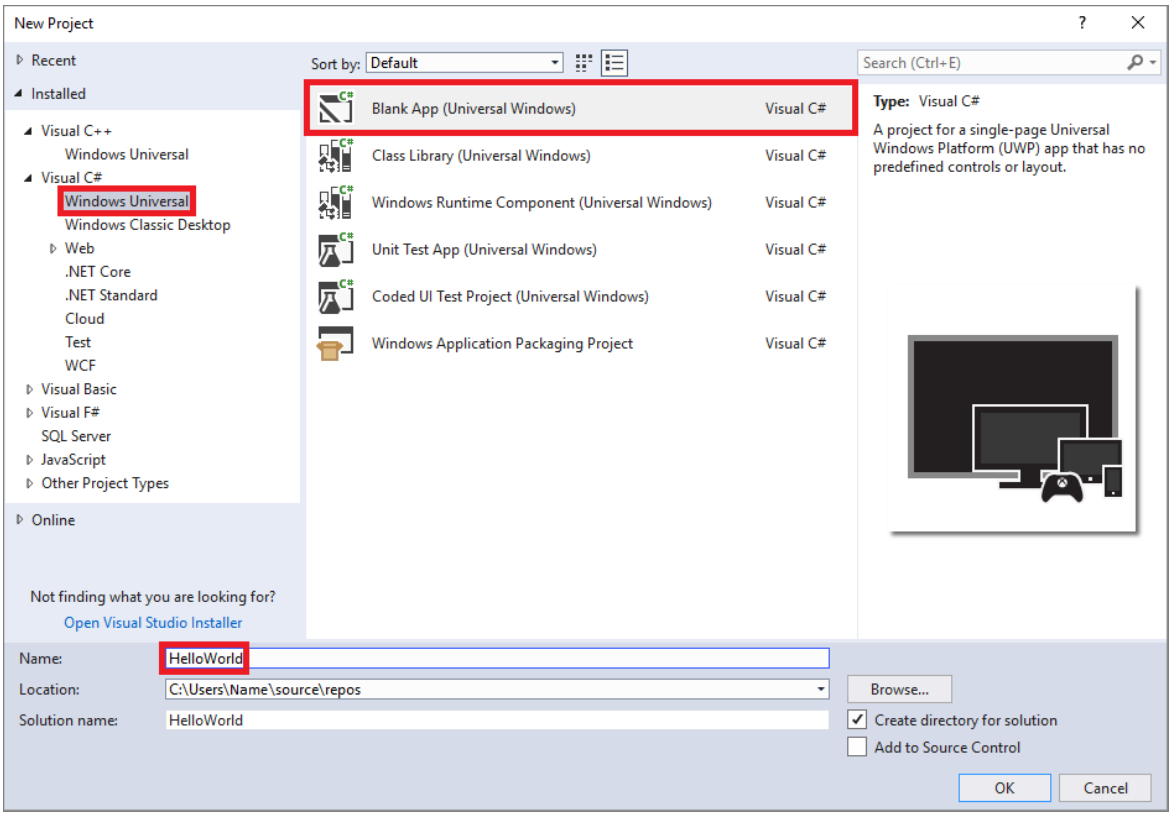
If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

Create a project

First, create a Universal Windows Platform project. The project type comes with all the template files you need, before you've even added anything!

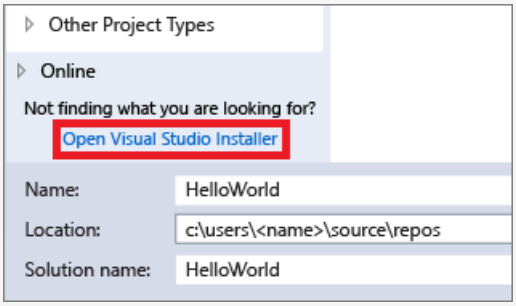
1. Open Visual Studio.
2. From the top menu bar, choose **File** > **New** > **Project**.
3. In the left pane of the **New Project** dialog box, expand **Visual C#**, and then choose **Windows Universal**. In the middle pane, choose **Blank App (Universal Windows)**. Then, name the project *HelloWorld* and choose **OK**.

> **NOTE**
>
> Make sure that the location of the source project is on a **New Technology File System (NTFS)** formatted drive, such as your Operating System (OS) drive. Otherwise, you might have trouble building and running your project.

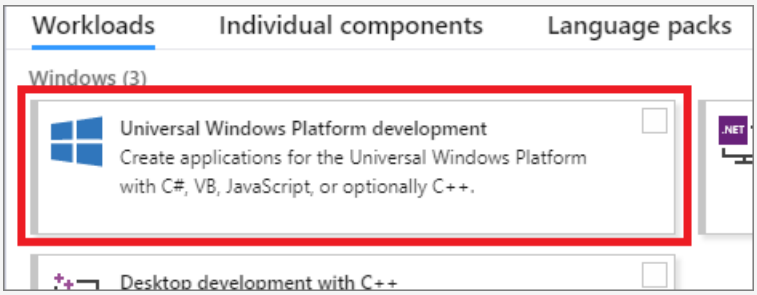


NOTE

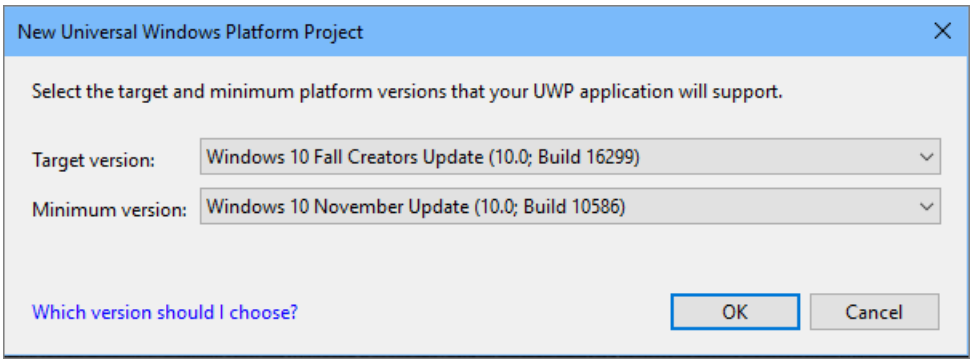
If you don't see the **Blank App (Universal Windows)** project template, click the **Open Visual Studio Installer** link in the left pane of the **New Project** dialog box.



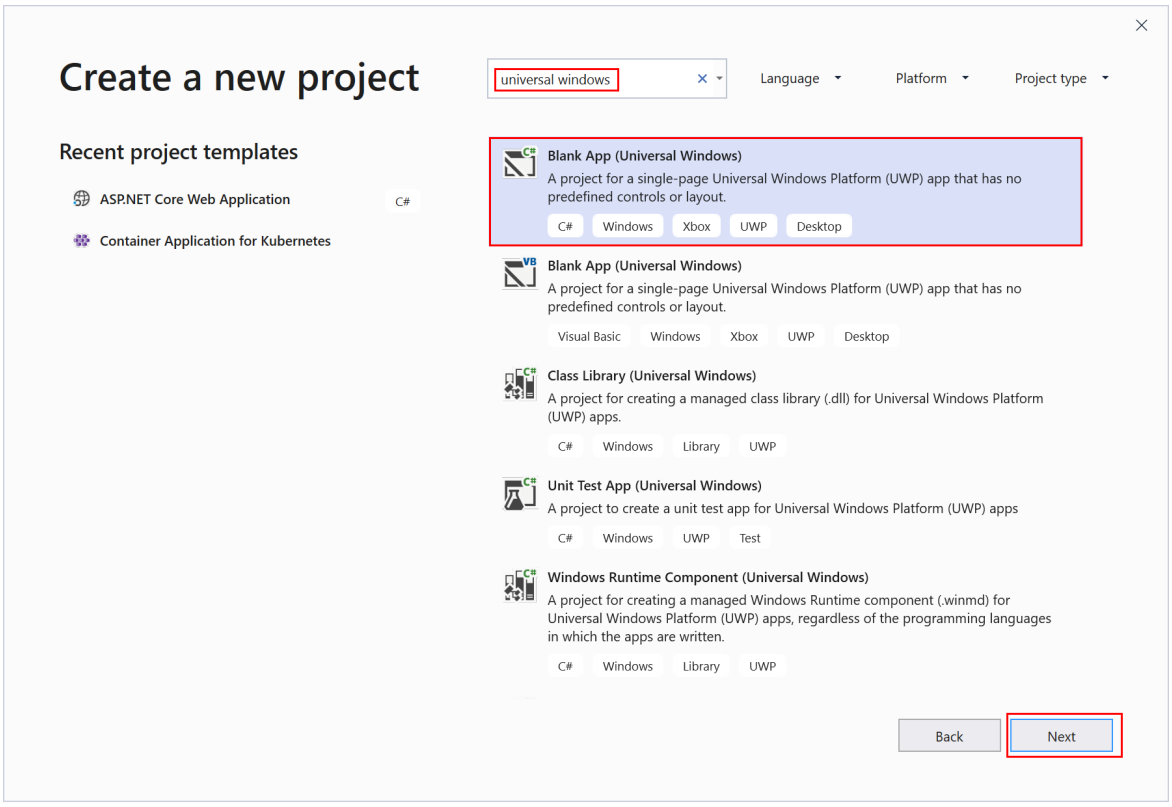
The Visual Studio Installer launches. Choose the **Universal Windows Platform development** workload, and then choose **Modify**.



4. Accept the default **Target version** and **Minimum version** settings in the **New Universal Windows Platform Project** dialog box.

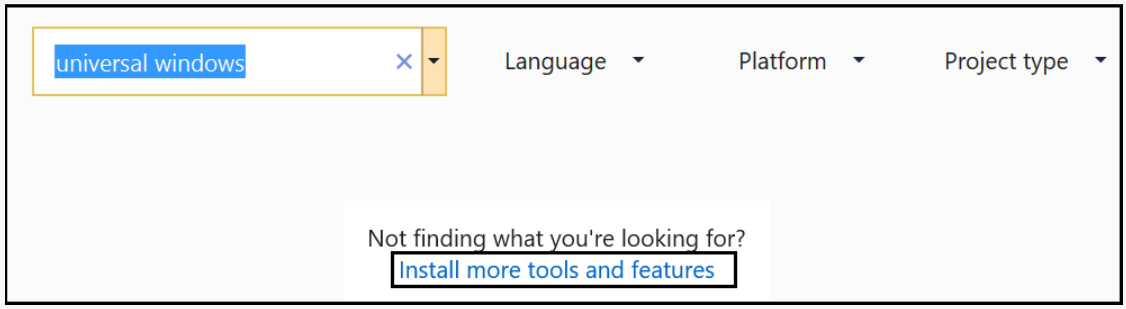


1. Open Visual Studio, and on the start window, choose **Create a new project**.
2. On the **Create a new project** screen, enter *Universal Windows* in the search box, choose the C# template for **Blank App (Universal Windows)**, and then choose **Next**.

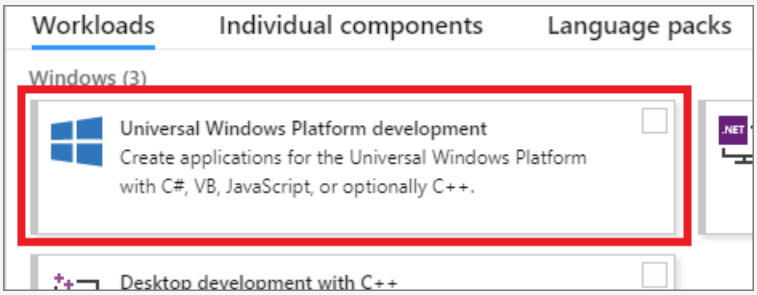


NOTE

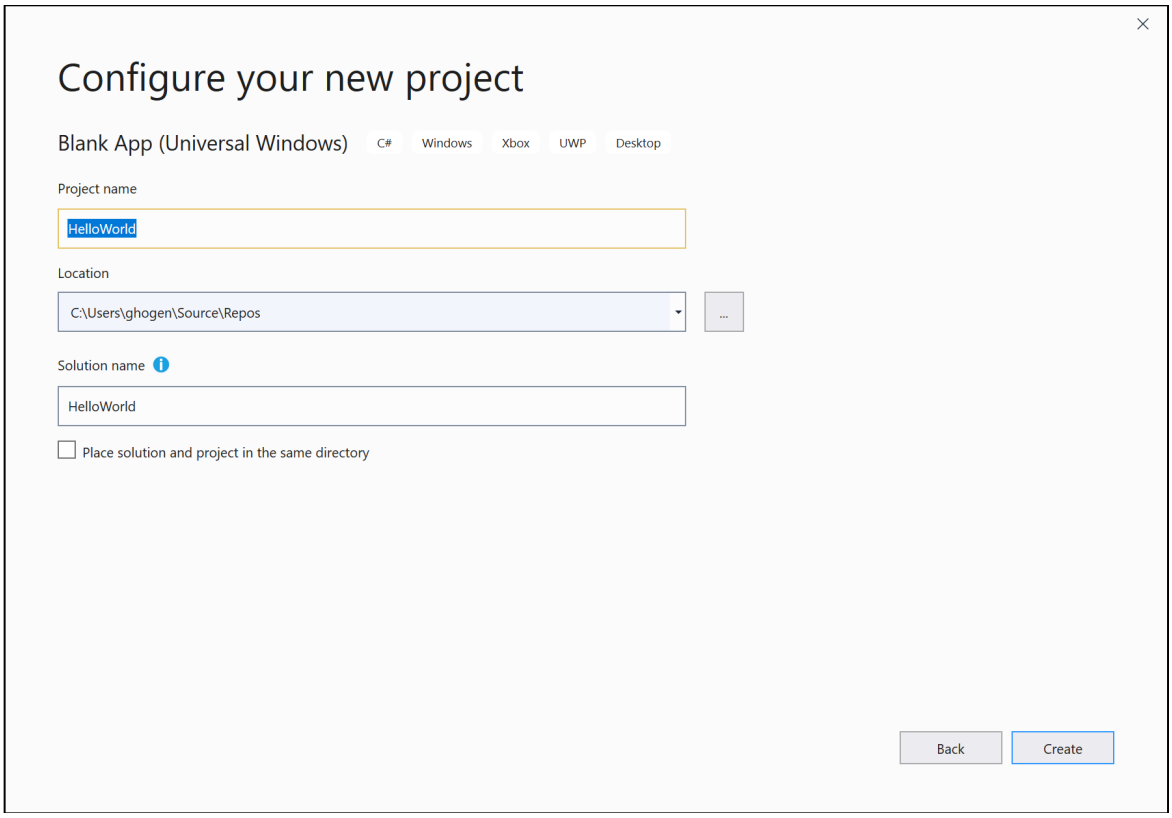
If you don't see the **Blank App (Universal Windows)** project template, click the **Install more tools and features** link.



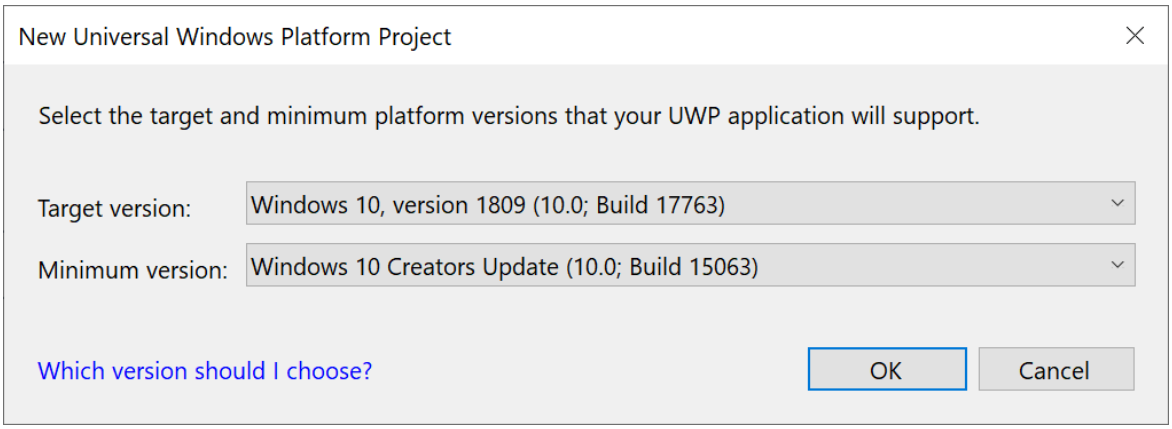
The Visual Studio Installer launches. Choose the **Universal Windows Platform development** workload, and then choose **Modify**.



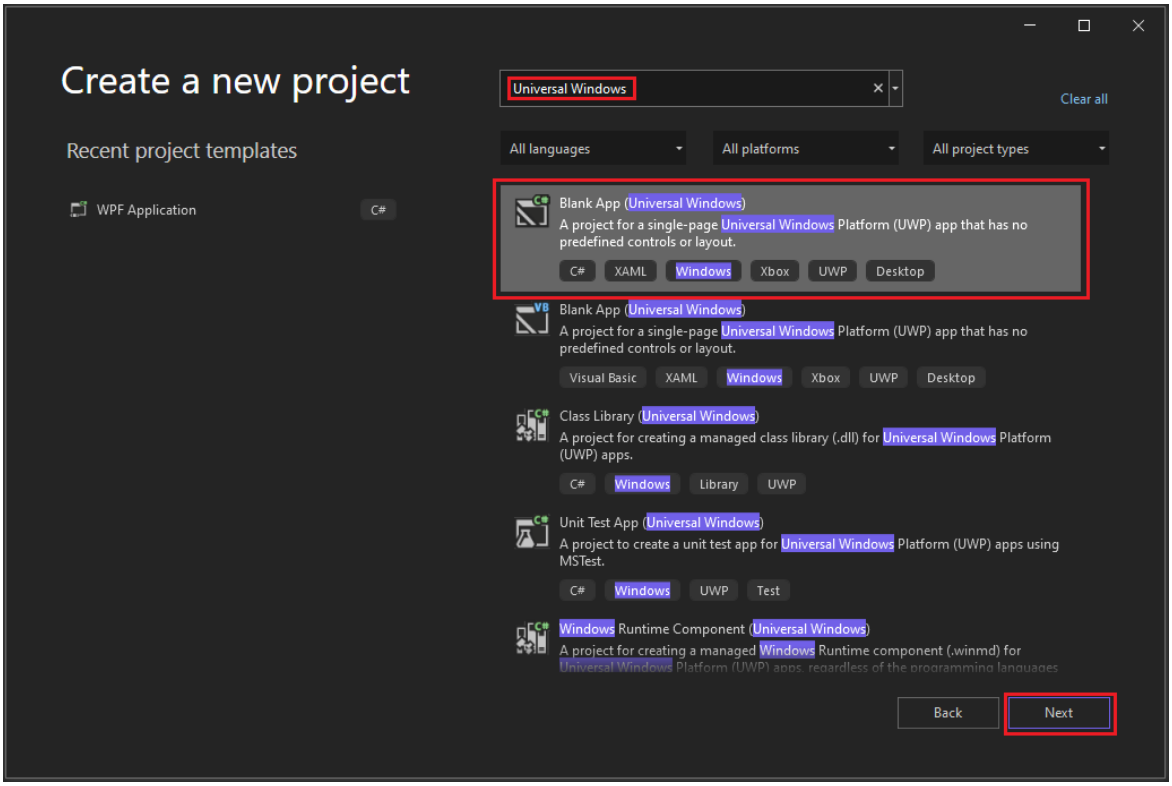
3. Give the project a name, *HelloWorld*, and choose **Create**.



4. Accept the default **Target version** and **Minimum version** settings in the **New Universal Windows Platform Project** dialog box.

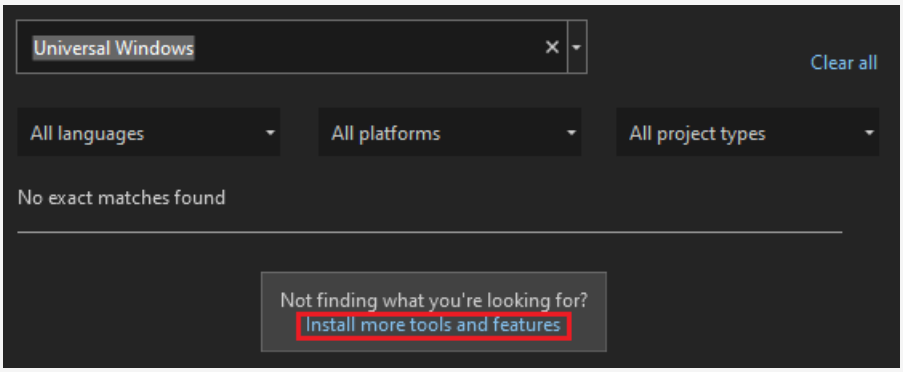


1. Open Visual Studio, and on the start window, choose **Create a new project**.
2. On the **Create a new project** screen, enter *Universal Windows* in the search box, choose the C# template for **Blank App (Universal Windows)**, and then choose **Next**.

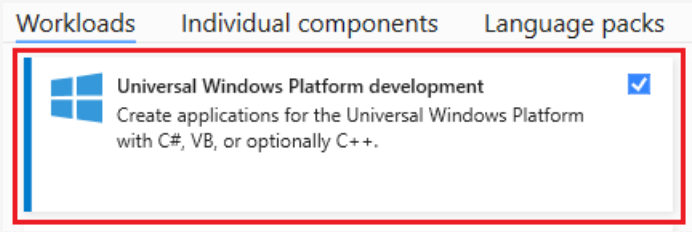


NOTE

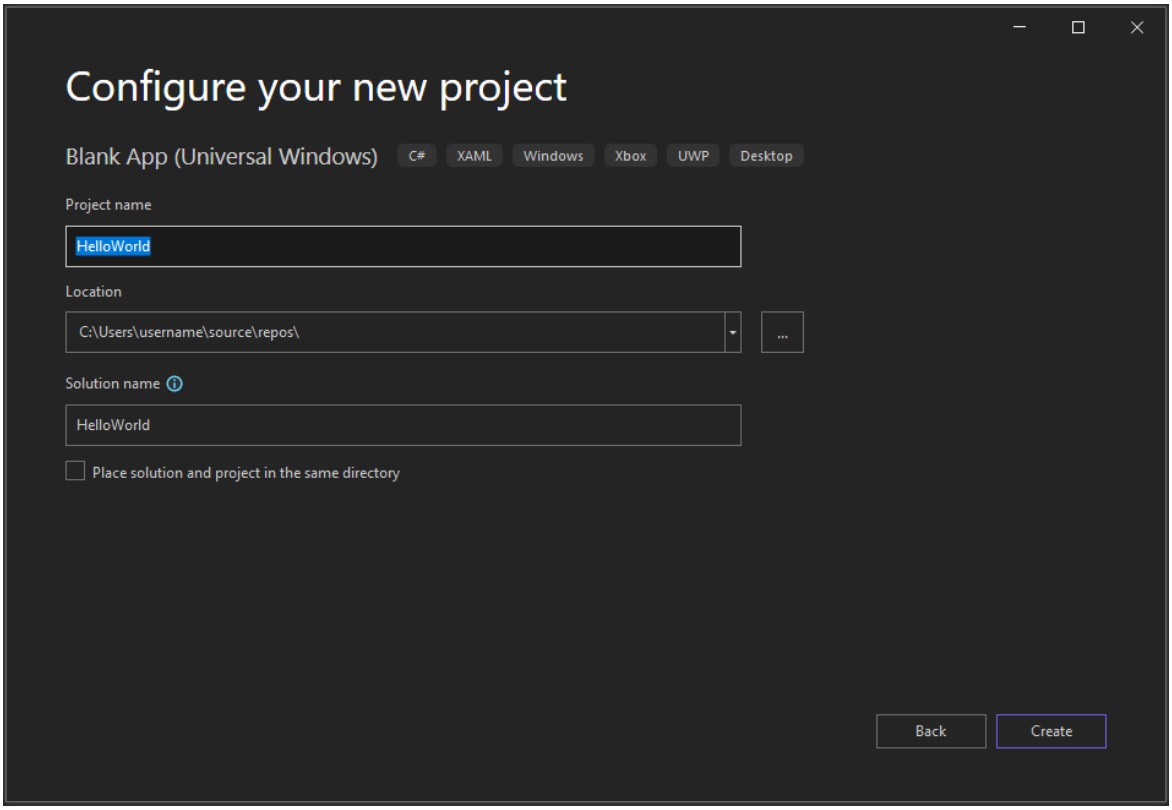
If you don't see the **Blank App (Universal Windows)** project template, click the **Install more tools and features** link.



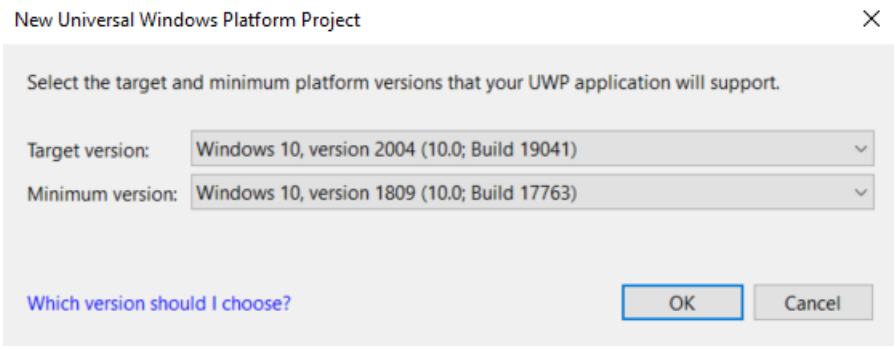
The Visual Studio Installer launches. Choose the **Universal Windows Platform development** workload, and then select **Modify**.



3. Give the project a name, *HelloWorld*, and choose **Create**.

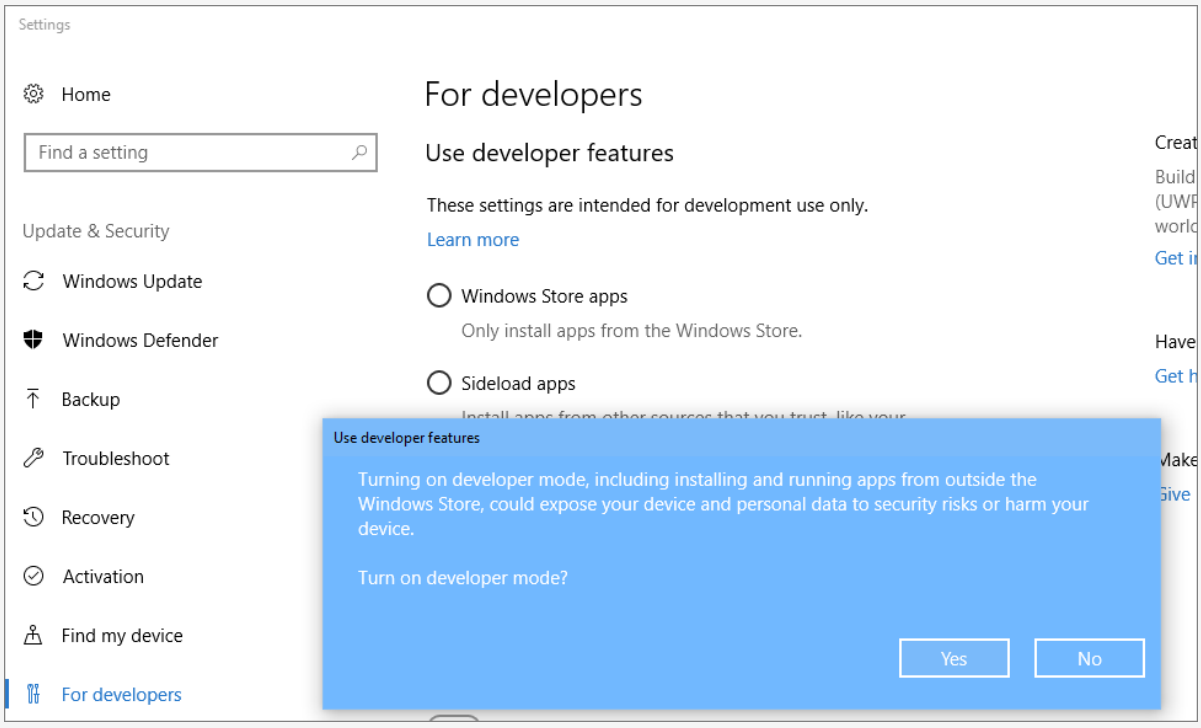


4. Accept the default **Target version** and **Minimum version** settings in the **New Universal Windows Platform Project** dialog box.



NOTE

If this is the first time you have used Visual Studio to create a UWP app, a **Settings** dialog box might appear. Choose **Developer mode**, and then choose **Yes**.



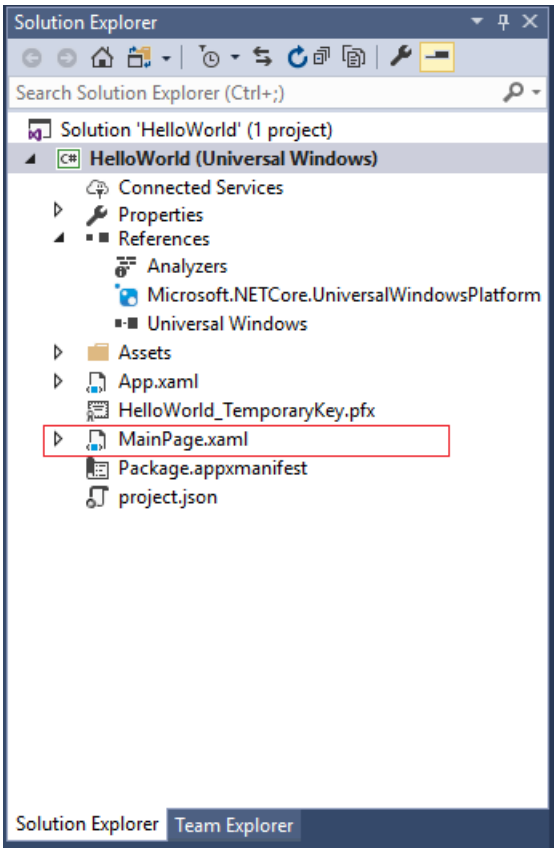
Visual Studio installs an additional Developer Mode package for you. When the package installation is complete, close the **Settings** dialog box.

Create the application

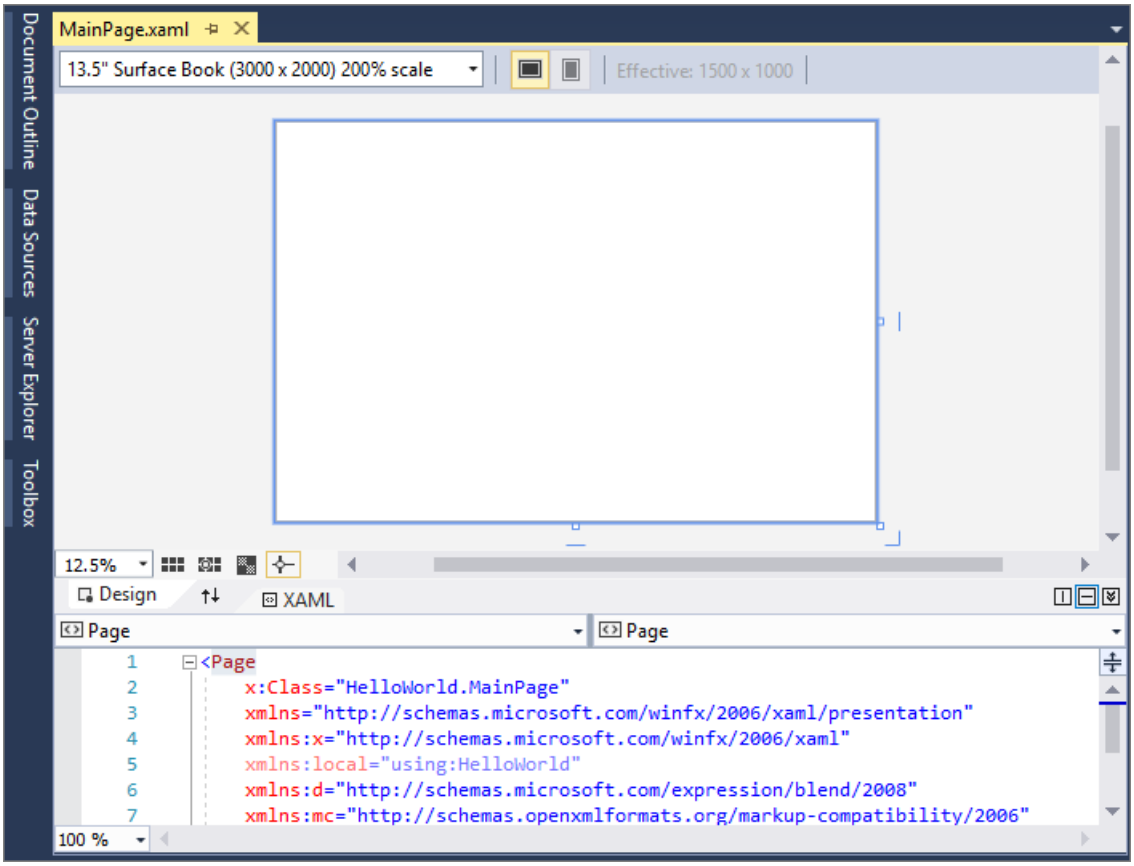
It's time to start developing. You'll add a button control, add an action to the button, and then start the "Hello World" app to see what it looks like.

Add a button to the Design canvas

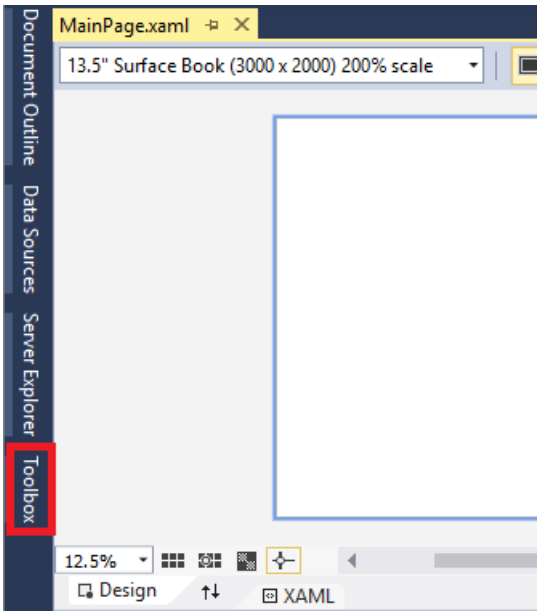
1. In the **Solution Explorer**, double-click *MainPage.xaml* to open a split view.



There are two panes: The **XAML Designer**, which includes a design canvas, and the **XAML Editor**, where you can add or change code.

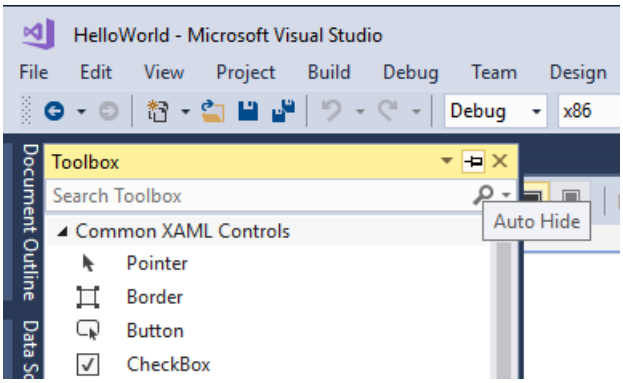


2. Choose **Toolbox** to open the Toolbox fly-out window.

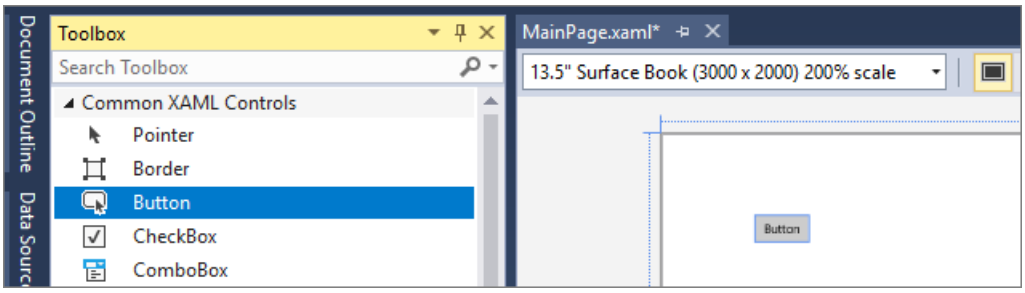


(If you don't see the **Toolbox** option, you can open it from the menu bar. To do so, choose **View** > **Toolbar**. Or, press **Ctrl**+**Alt**+**X**.)

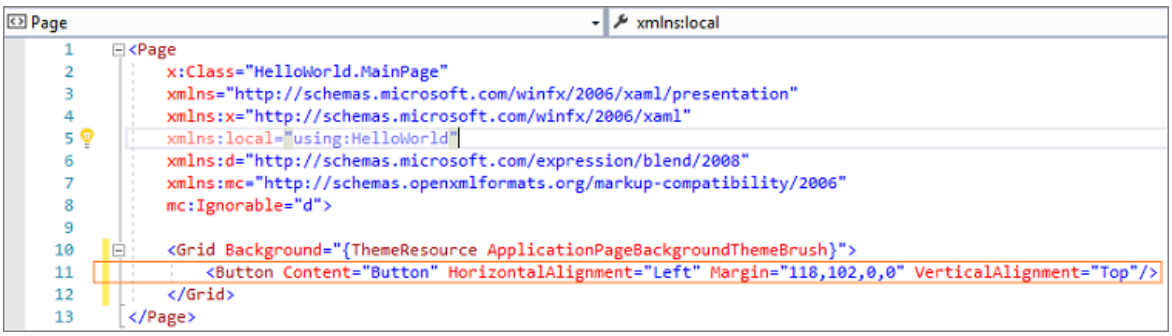
3. Click the **Pin** icon to dock the Toolbox window.



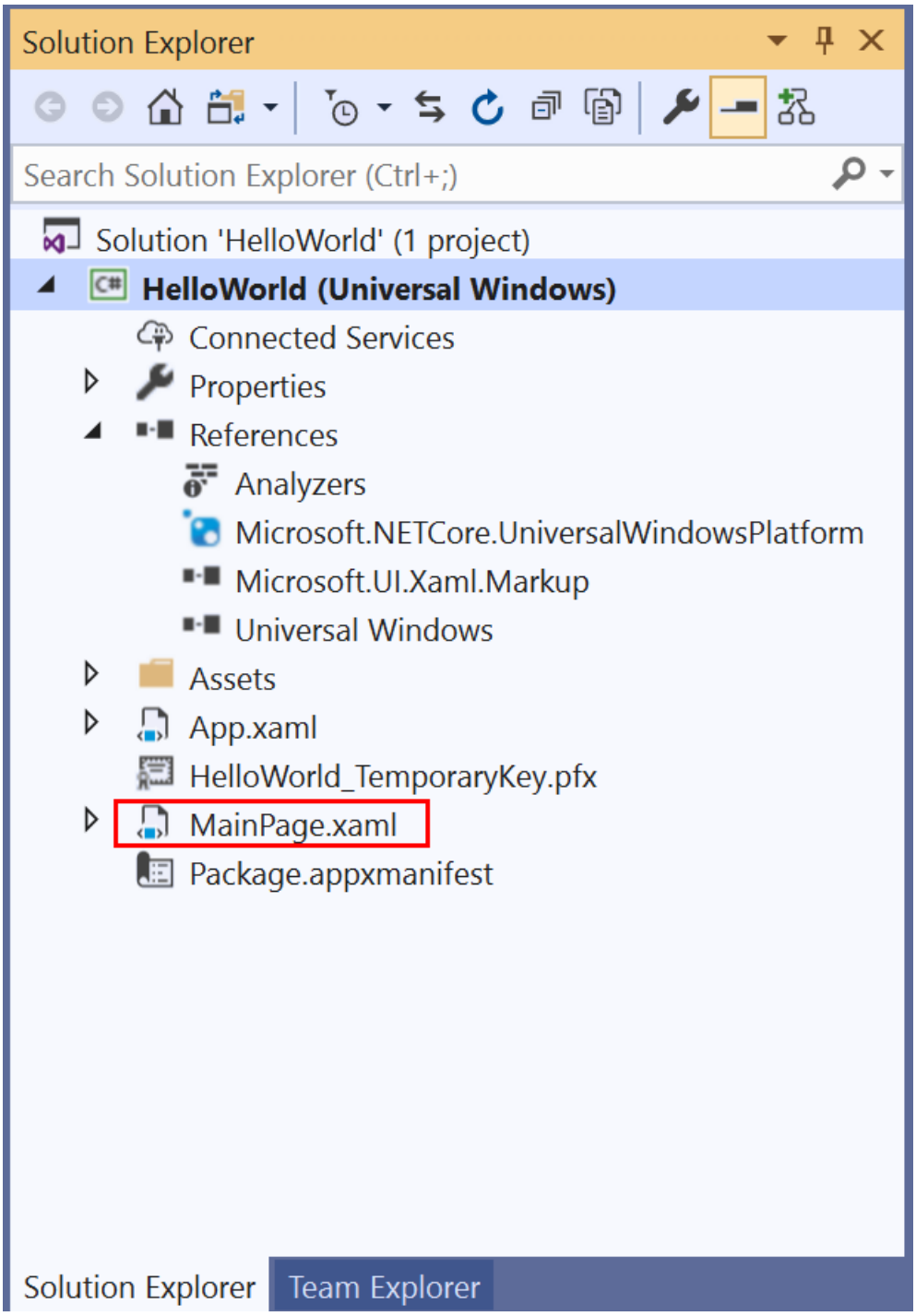
4. Click the **Button** control and then drag it onto the design canvas.



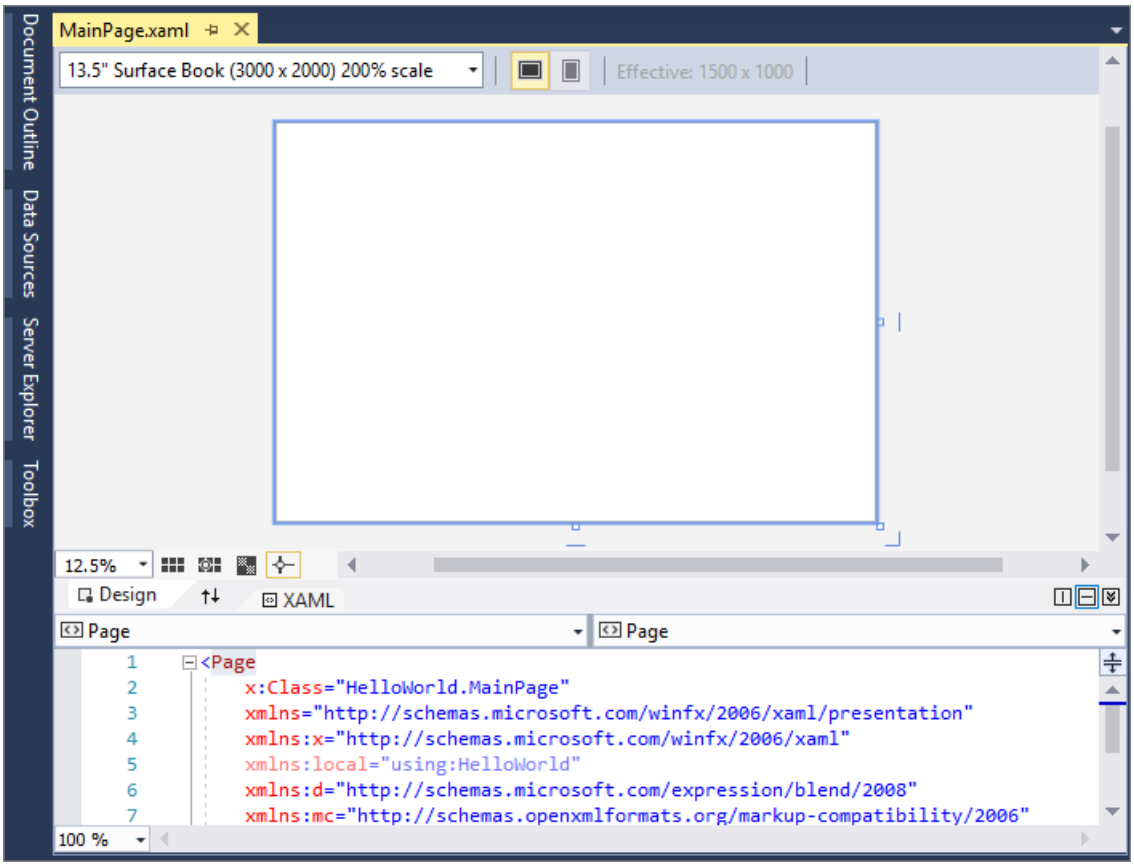
If you look at the code in the **XAML Editor**, you'll see that the Button has been added there, too:



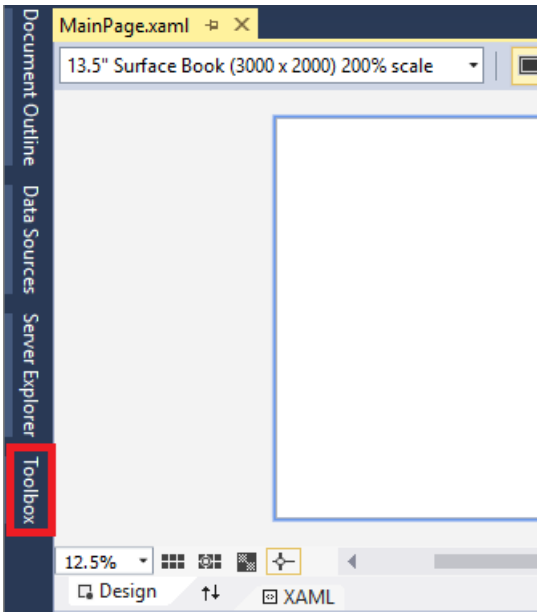
1. In the **Solution Explorer**, double-click *MainPage.xaml* to open a split view.



There are two panes: The **XAML Designer**, which includes a design canvas, and the **XAML Editor**, where you can add or change code.

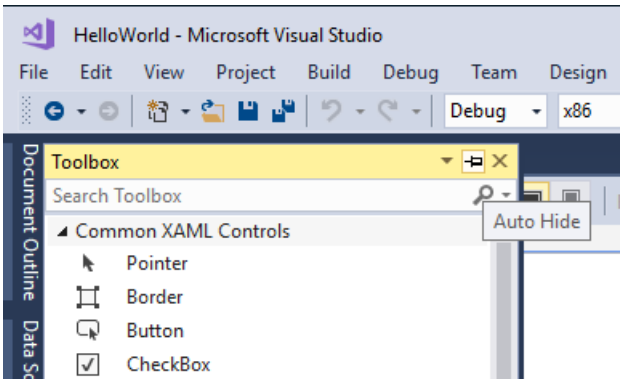


2. Choose **Toolbox** to open the Toolbox fly-out window.

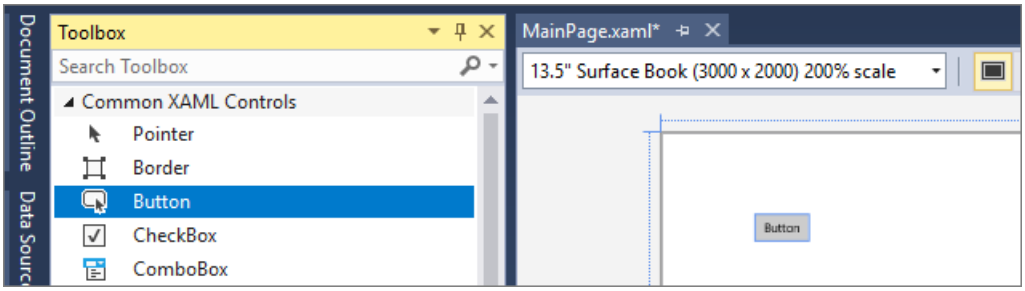


(If you don't see the **Toolbox** option, you can open it from the menu bar. To do so, choose **View** > **Toolbar**. Or, press **Ctrl**+**Alt**+**X**.)

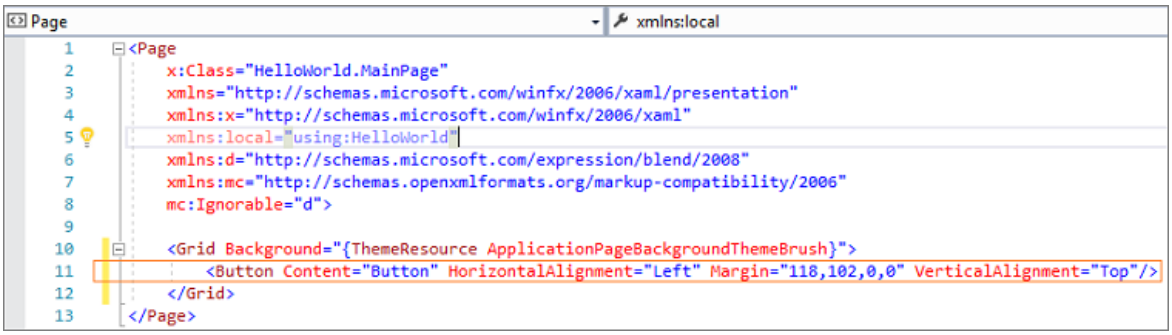
3. Click the **Pin** icon to dock the Toolbox window.



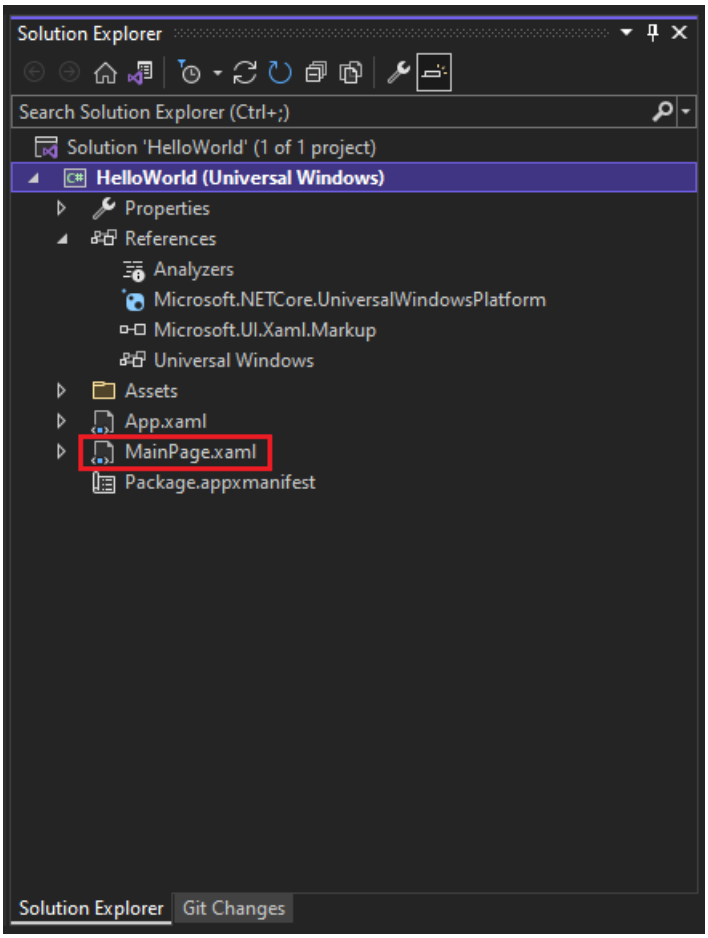
4. Click the **Button** control and then drag it onto the design canvas.



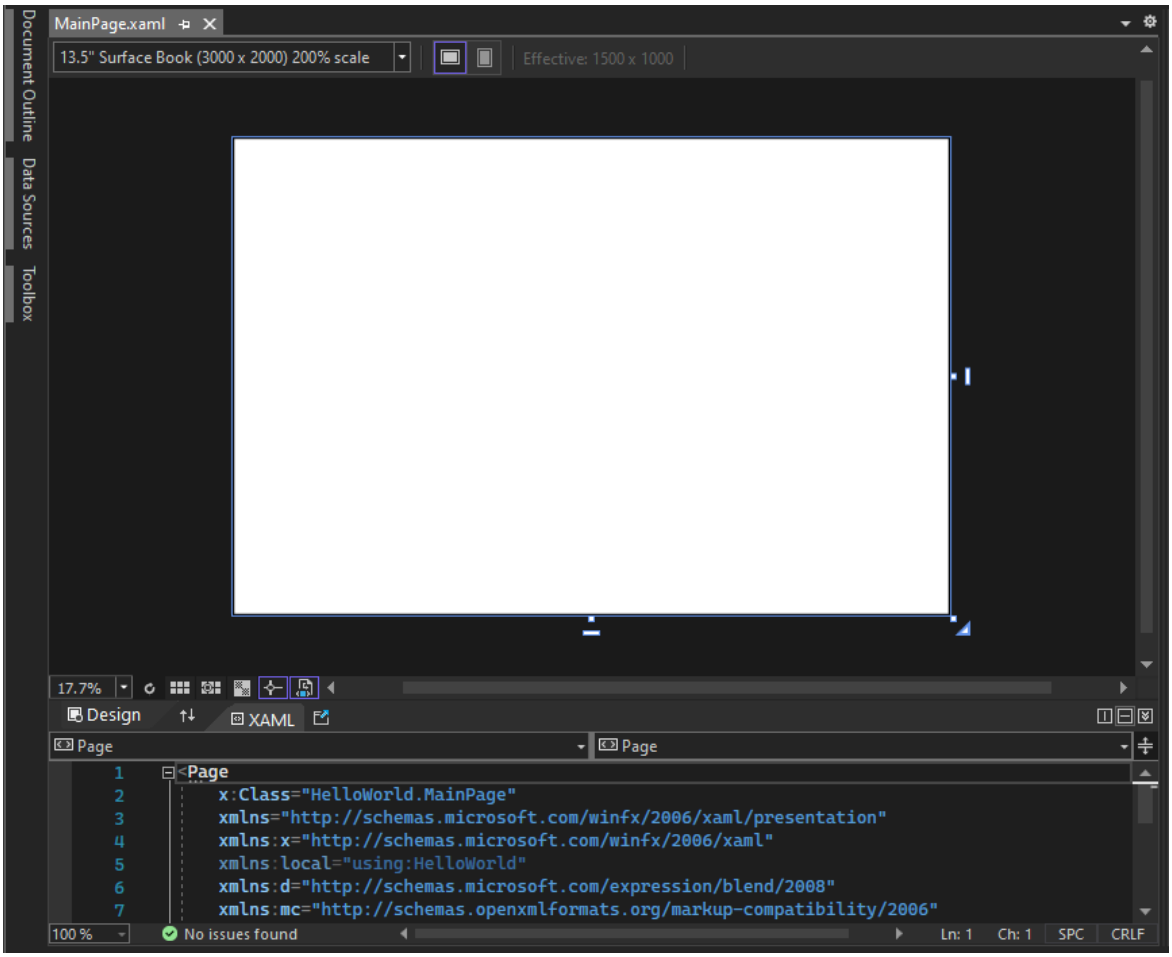
If you look at the code in the **XAML Editor**, you'll see that the Button has been added there, too:



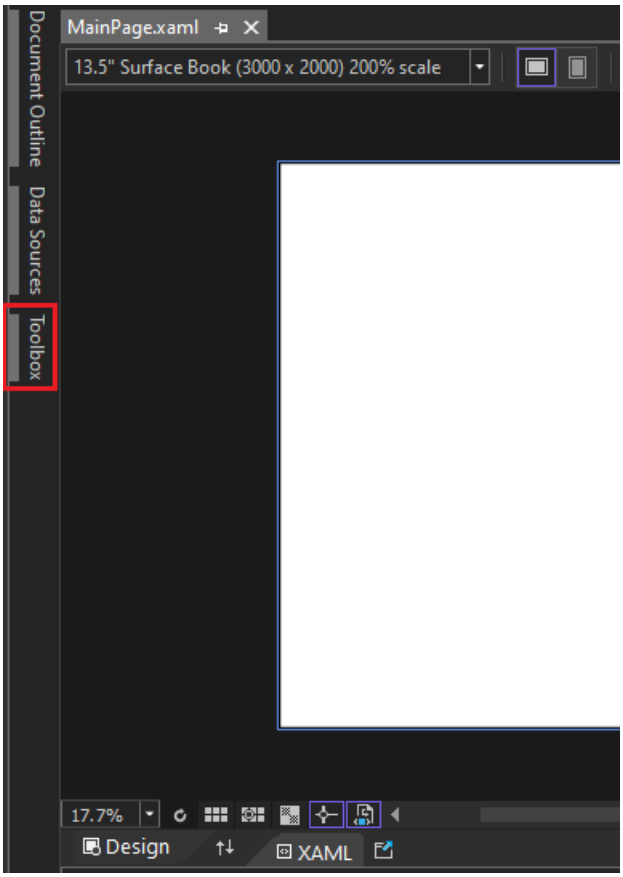
1. In the **Solution Explorer**, double-click *MainPage.xaml* to open a split view.



There are two panes: The **XAML Designer**, which includes a design canvas, and the **XAML Editor**, where you can add or change code.

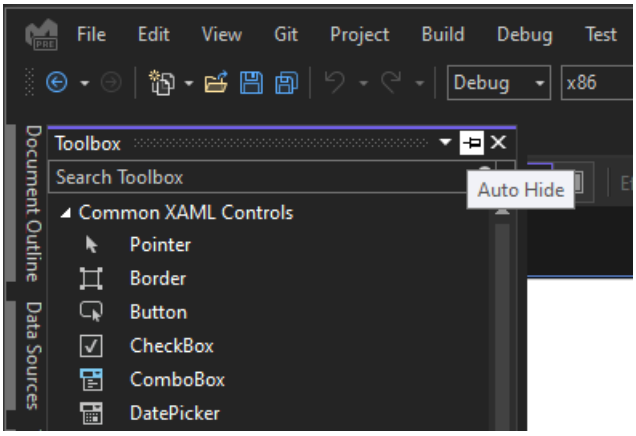


2. Choose **Toolbox** to open the Toolbox fly-out window.

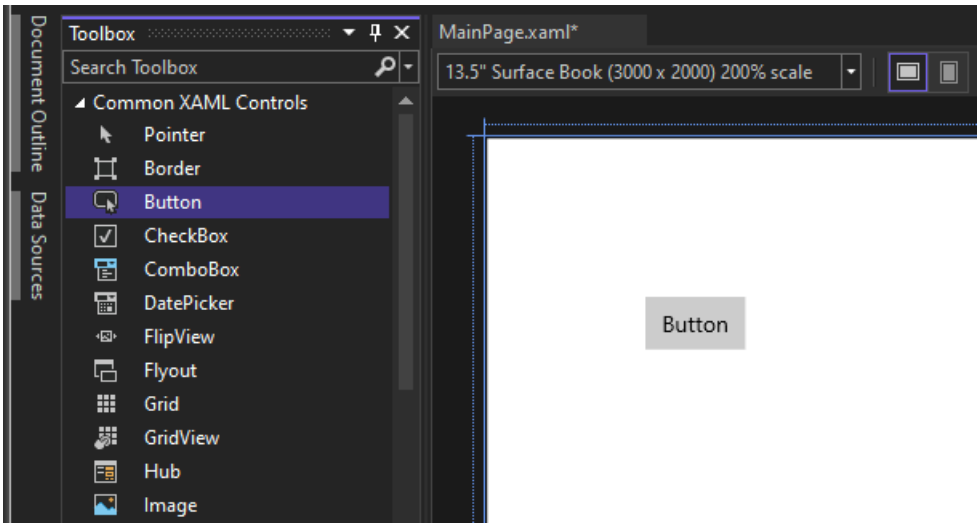


(If you don't see the **Toolbox** option, you can open it from the menu bar. To do so, choose **View** > **Toolbar**. Or, press **Ctrl**+**Alt**+**X**.)

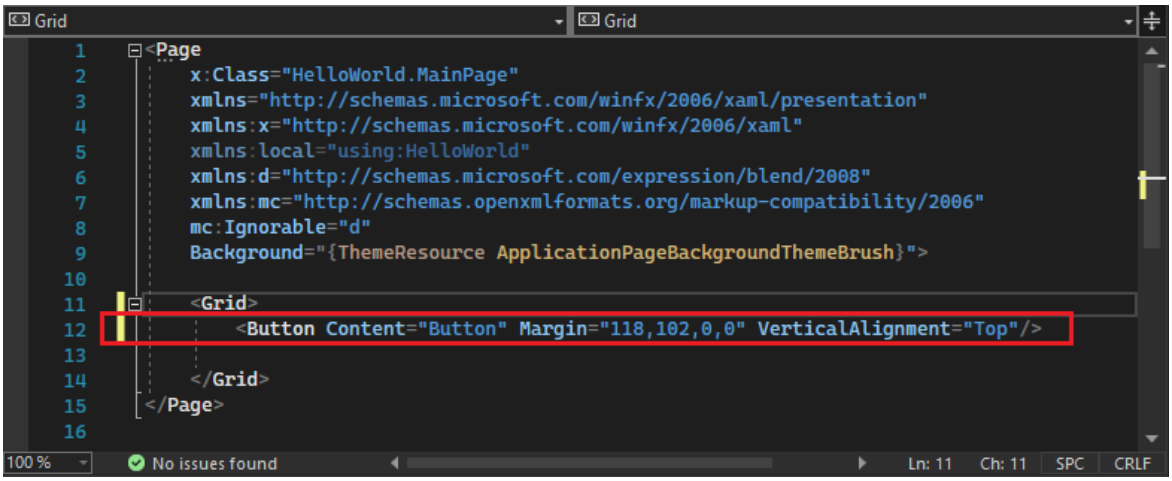
3. Select the **Pin** icon to dock the Toolbox window.



4. Select the **Button** control and then drag it onto the design canvas.

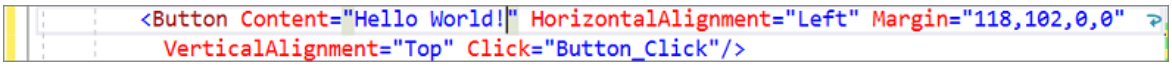


If you look at the code in the **XAML Editor**, you'll see that the Button has been added there, too:

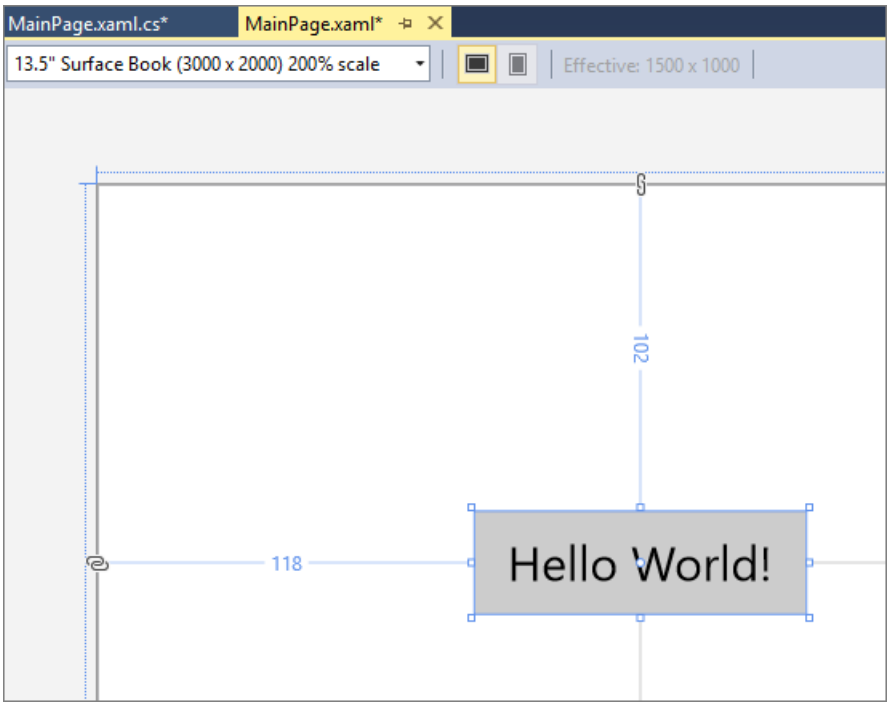


Add a label to the button

1. In the **XAML Editor**, change Button Content value from "Button" to "Hello World!".



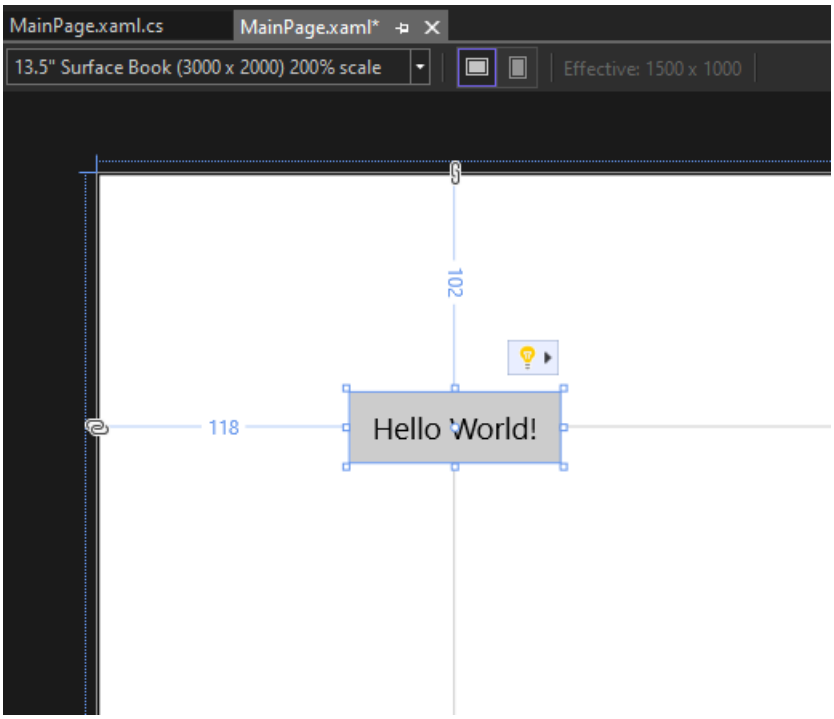
2. Notice that the button in the **XAML Designer** changes, too.



1. In the **XAML Editor**, change Button Content value from "Button" to "Hello World!".

```
<Button Content="Hello World!" Margin="118,102,0,0" VerticalAlignment="Top"/>
```

2. Notice that the button in the **XAML Designer** changes, too.



Add an event handler

An "event handler" sounds complicated, but it's just another name for code that is called when an event happens. In this case, it adds an action to the "Hello World!" button.

1. Double-click the button control on the design canvas.
2. Edit the event handler code in *MainPage.xaml.cs*, the code-behind page.

   Here is where things get interesting. The default event handler looks like this:

```
29
30          private void Button_Click(object sender, RoutedEventArgs e)
31          {
32
33          }
```

   Let's change it, so it looks like this:

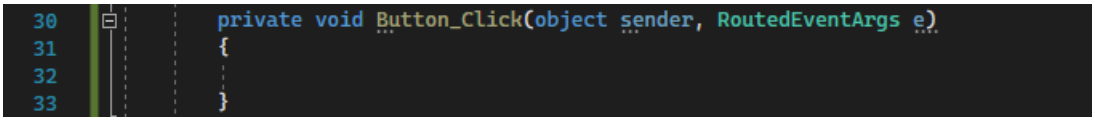
```
23      public sealed partial class MainPage : Page
24      {
25          public MainPage()
26          {
27              this.InitializeComponent();
28          }
29
30          private async void Button_Click(object sender, RoutedEventArgs e)
31          {
32                  MediaElement mediaElement = new MediaElement();
33                  var synth = new Windows.Media.SpeechSynthesis.SpeechSynthesizer();
34                  Windows.Media.SpeechSynthesis.SpeechSynthesisStream stream = await
                      synth.SynthesizeTextToStreamAsync("Hello, World!");
35                  mediaElement.SetSource(stream, stream.ContentType);
36                  mediaElement.Play();
37          }
38      }
39  }
```

Here's the code to copy and paste:

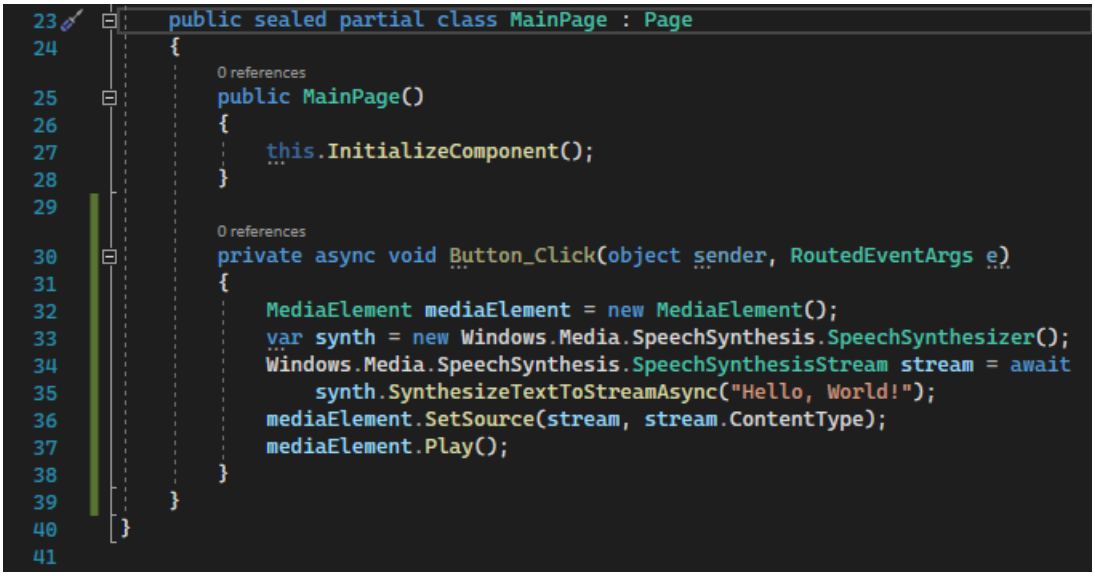
```
private async void Button_Click(object sender, RoutedEventArgs e)
      {
          MediaElement mediaElement = new MediaElement();
          var synth = new Windows.Media.SpeechSynthesis.SpeechSynthesizer();
          Windows.Media.SpeechSynthesis.SpeechSynthesisStream stream = await
synth.SynthesizeTextToStreamAsync("Hello, World!");
          mediaElement.SetSource(stream, stream.ContentType);
          mediaElement.Play();
      }
```

1. Double-click the button control on the design canvas.
2. Edit the event handler code in *MainPage.xaml.cs*, the code-behind page.

   Here is where things get interesting. The default event handler looks like this:



   Let's change it, so it looks like this:



   Here's the code to copy and paste:

```
private async void Button_Click(object sender, RoutedEventArgs e)
      {
          MediaElement mediaElement = new MediaElement();
          var synth = new Windows.Media.SpeechSynthesis.SpeechSynthesizer();
          Windows.Media.SpeechSynthesis.SpeechSynthesisStream stream = await
synth.SynthesizeTextToStreamAsync("Hello, World!");
          mediaElement.SetSource(stream, stream.ContentType);
          mediaElement.Play();
      }
```

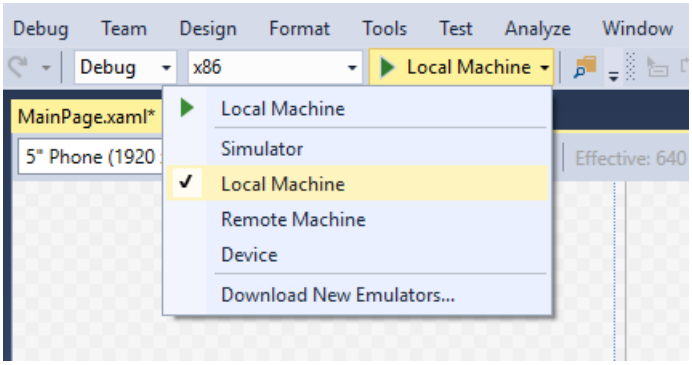
What did we just do?

The code uses some Windows APIs to create a speech synthesis object and then gives it some text to say. (For more information on using `SpeechSynthesis`, see System.Speech.Synthesis.)

Run the application

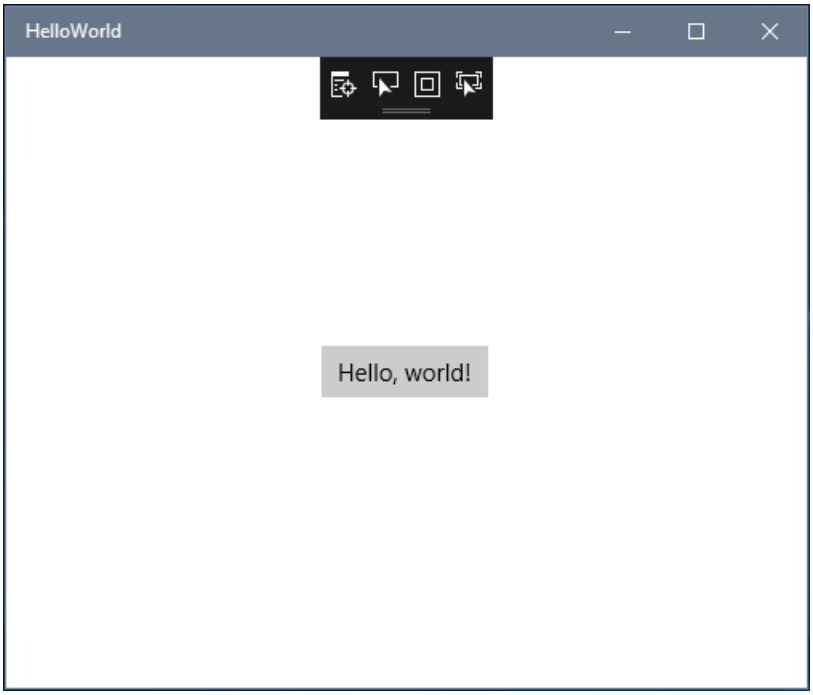
It's time to build, deploy, and launch the "Hello World" UWP app to see what it looks and sounds like. Here's how.

1. Use the Play button (it has the text **Local Machine**) to start the application on the local machine.



   (Alternatively, you can choose **Debug** > **Start Debugging** from the menu bar or press **F5** to start your app.)

2. View your app, which appears soon after a splash screen disappears. The app should look similar to this:



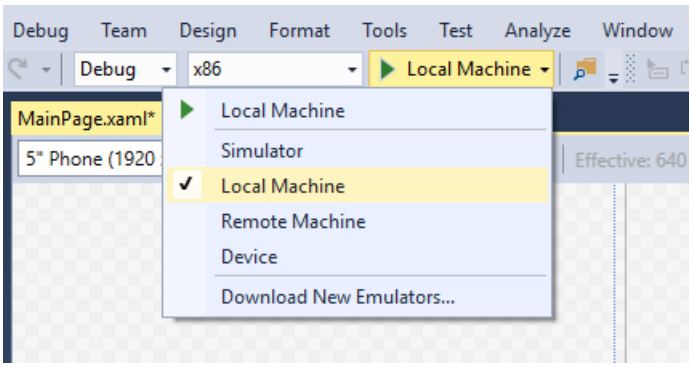
3. Click the **Hello World** button.

   Your Windows 10 or later device will literally say, "Hello, World!"

4. To close the app, click the **Stop Debugging** button in the toolbar. (Alternatively, choose **Debug** > **Stop debugging** from the menu bar, or press **Shift+F5**.)

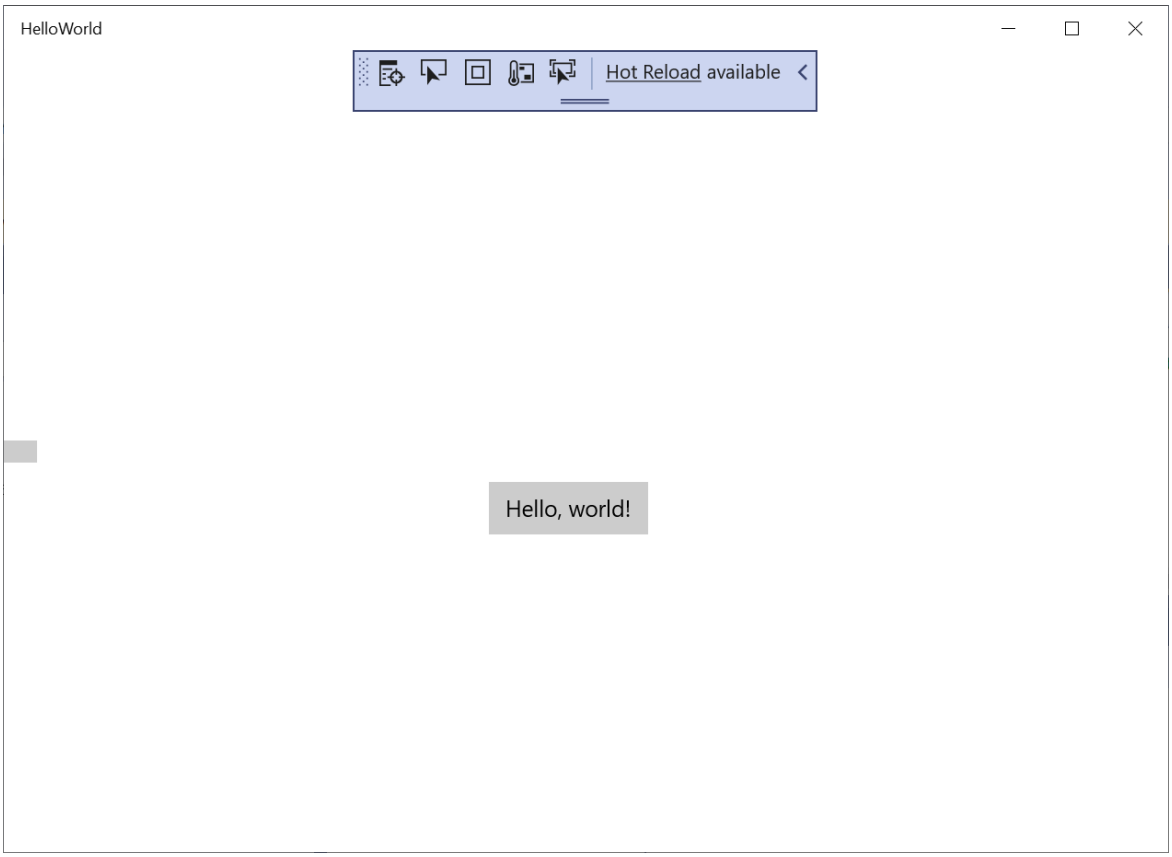
It's time to build, deploy, and launch the "Hello World" UWP app to see what it looks and sounds like. Here's how.

1. Use the Play button (it has the text **Local Machine**) to start the application on the local machine.



(Alternatively, you can choose **Debug** > **Start Debugging** from the menu bar or press **F5** to start your app.)

2. View your app, which appears soon after a splash screen disappears. The app should look similar to this:



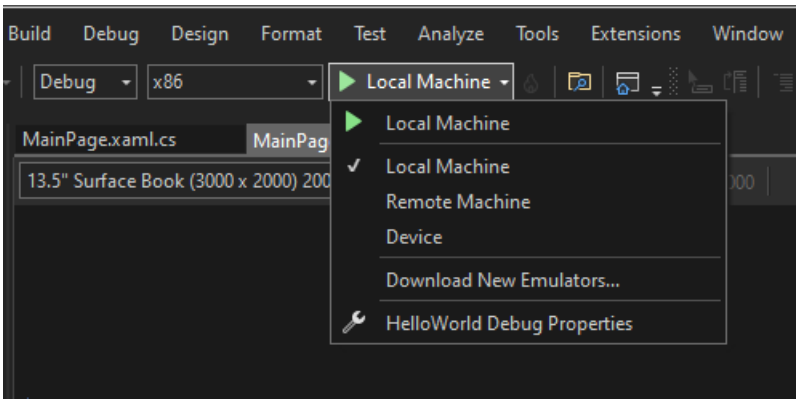
3. Click the **Hello World** button.

   Your Windows 10 or later device will literally say, "Hello, World!"

4. To close the app, click the **Stop Debugging** button in the toolbar. (Alternatively, choose **Debug** > **Stop debugging** from the menu bar, or press **Shift+F5**.)

It's time to build, deploy, and launch the "Hello World" UWP app to see what it looks and sounds like. Here's how.

1. Use the Play button (it has the text **Local Machine**) to start the application on the local machine.



(Alternatively, you can choose **Debug** > **Start Debugging** from the menu bar or press **F5** to start your app.)

2. View your app, which appears soon after a splash screen disappears. The app should look similar to this image:

3. Select the **Hello World** button.

   Your Windows 10 or later device will literally say, "Hello, World!".

4. To close the app, select the **Stop Debugging** button in the toolbar. (Alternatively, choose **Debug** > **Stop debugging** from the menu bar, or press **Shift+F5**.)

Next steps

Congratulations on completing this tutorial! We hope you learned some basics about UWP and the Visual Studio IDE. To learn more, continue with the following tutorial:

Create a user interface

See also

- UWP overview
- Get UWP app samples

Tutorial: Create a simple application with C#

10/28/2021 • 15 minutes to read • Edit Online

By completing this tutorial, you'll become familiar with many of the tools, dialog boxes, and designers that you can use when you develop applications with Visual Studio. You'll create a "Hello, World" application, design the UI, add code, and debug errors, while you learn about working in the integrated development environment (IDE).

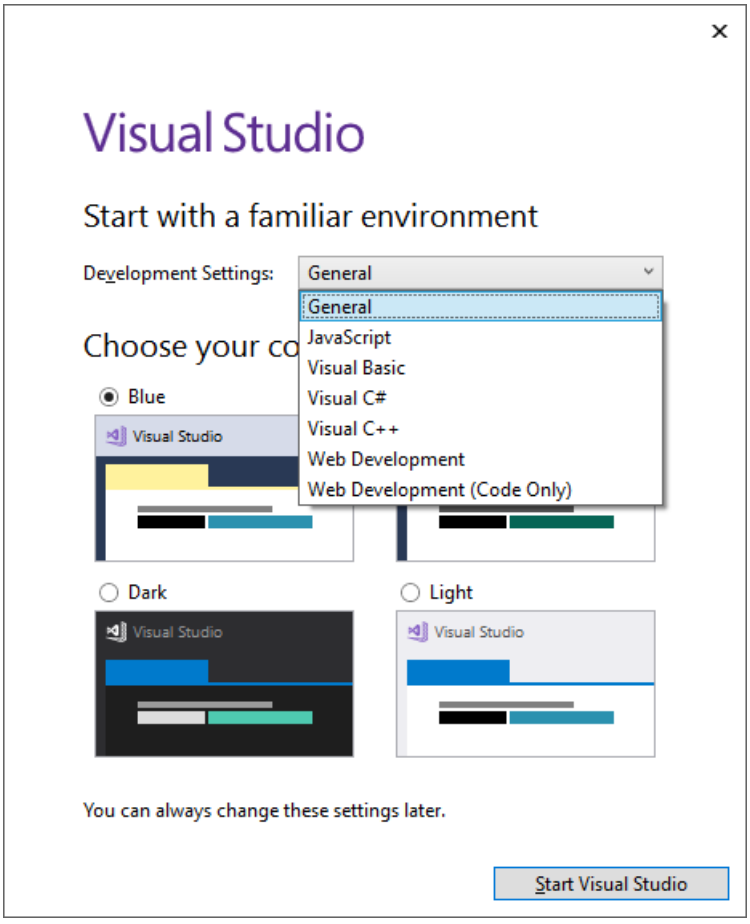
Prerequisites

If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

- If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.
- You can use either .NET Framework or .NET Core for this tutorial. .NET Core is the newer, more modern framework. .NET Core requires Visual Studio 2019 version 16.3 or later.

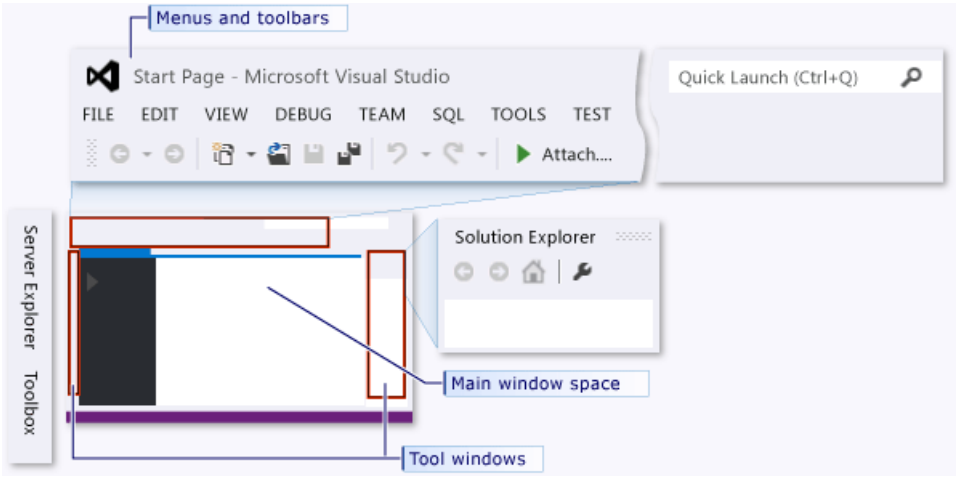
Configure the IDE

When you open Visual Studio for the first time, you'll be prompted to sign in. This step is optional for this tutorial. Next you may be shown a dialog box that asks you to choose your development settings and color theme. Keep the defaults and choose **Start Visual Studio**.



After Visual Studio launches, you'll see tool windows, the menus and toolbars, and the main window space. Tool windows are docked on the left and right sides of the application window, with **Quick Launch**, the menu bar, and the standard toolbar at the top. In the center of the application window is the **Start Page**. When you load a solution or project, editors and designers appear in the space where the **Start Page** is. When you develop an

application, you'll spend most of your time in this central area.

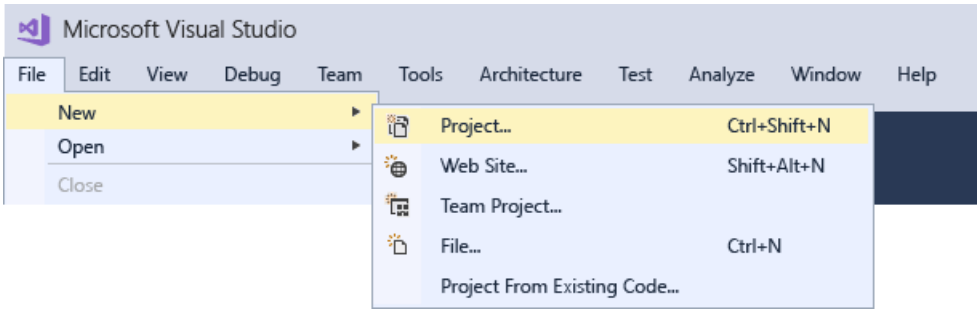


When you launch Visual Studio, the start window opens first. Select **Continue without code** to open the development environment. You'll see tool windows, the menus and toolbars, and the main window space. Tool windows are docked on the left and right sides of the application window. The search box, menu bar, and the standard toolbar are located at the top. When you load a solution or project, editors and designers appear in the central space of the application window. When you develop an application, you'll spend most of your time in this central area.

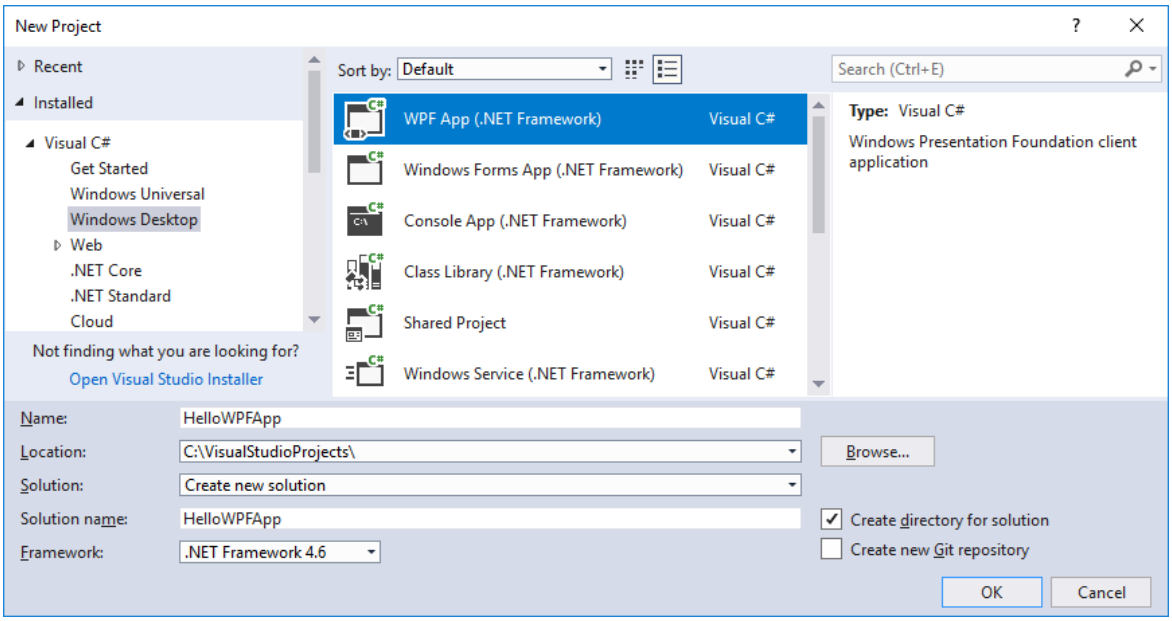
Create the project

When you create an application in Visual Studio, you first create a project and a solution. For this example, you'll create a Windows Presentation Foundation (WPF) project.

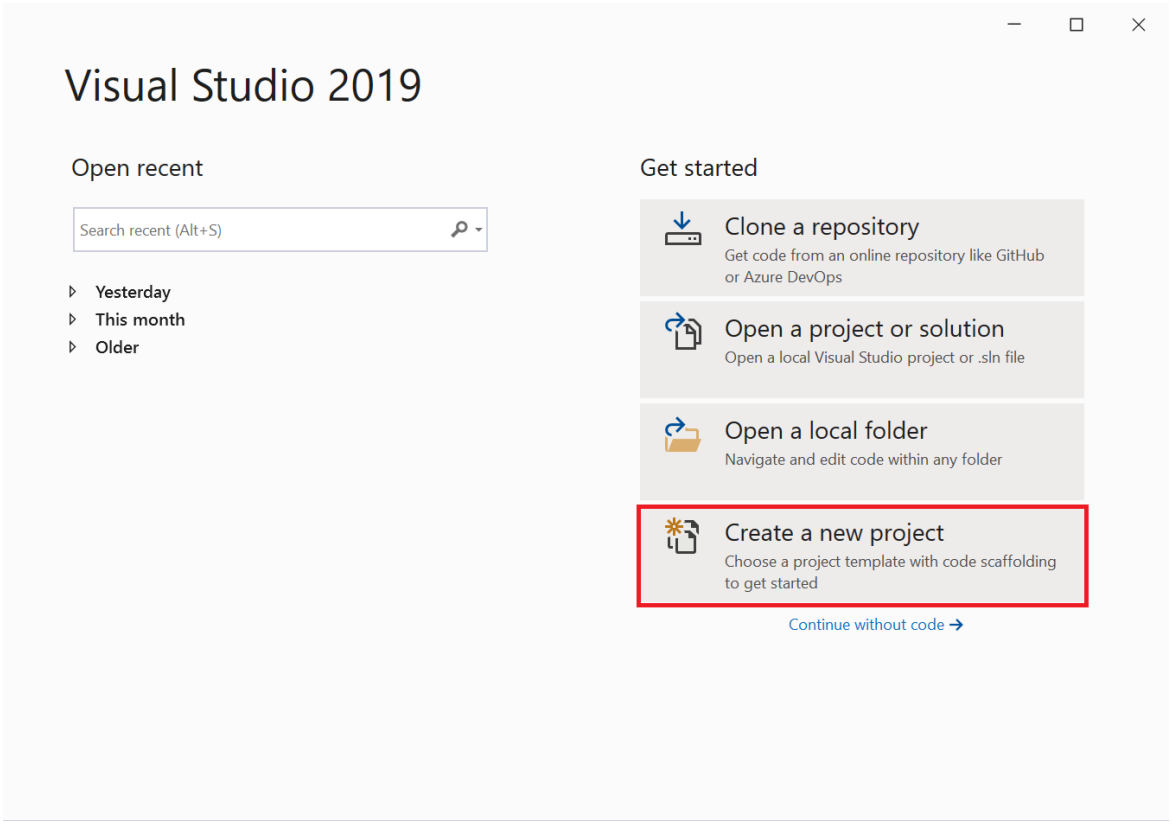
1. Create a new project. On the menu bar, select **File** > **New** > **Project**.



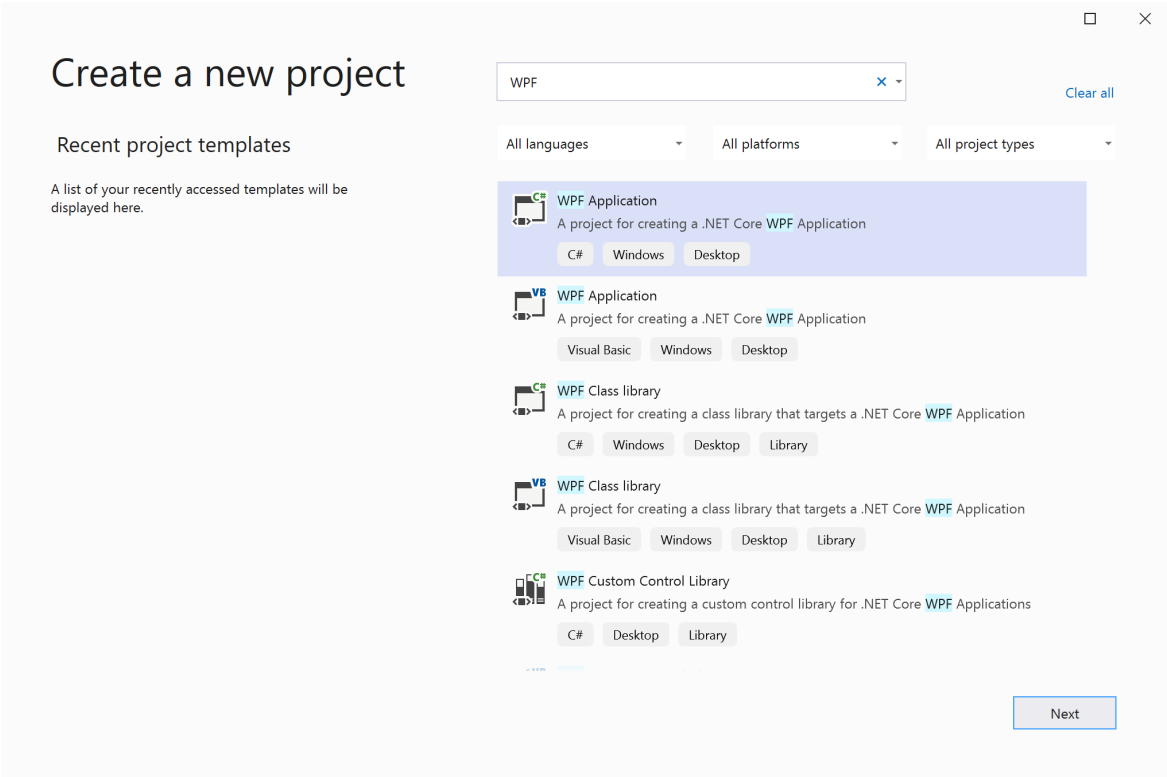
2. In the **New Project** dialog, select the **Installed** > **Visual C#** > **Windows Desktop** category, and then select the **WPF App (.NET Framework)** template. Name the project **HelloWPFApp**, and select **OK**.



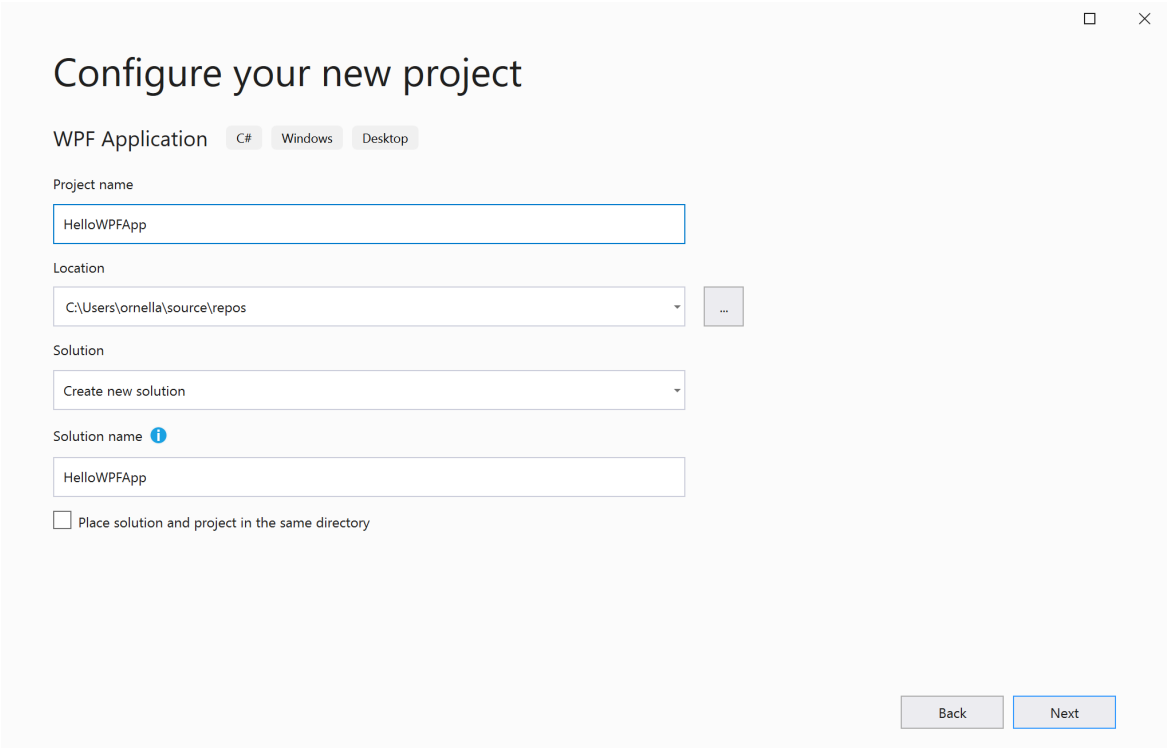
1. Open Visual Studio.
2. On the start window, choose **Create new project**.



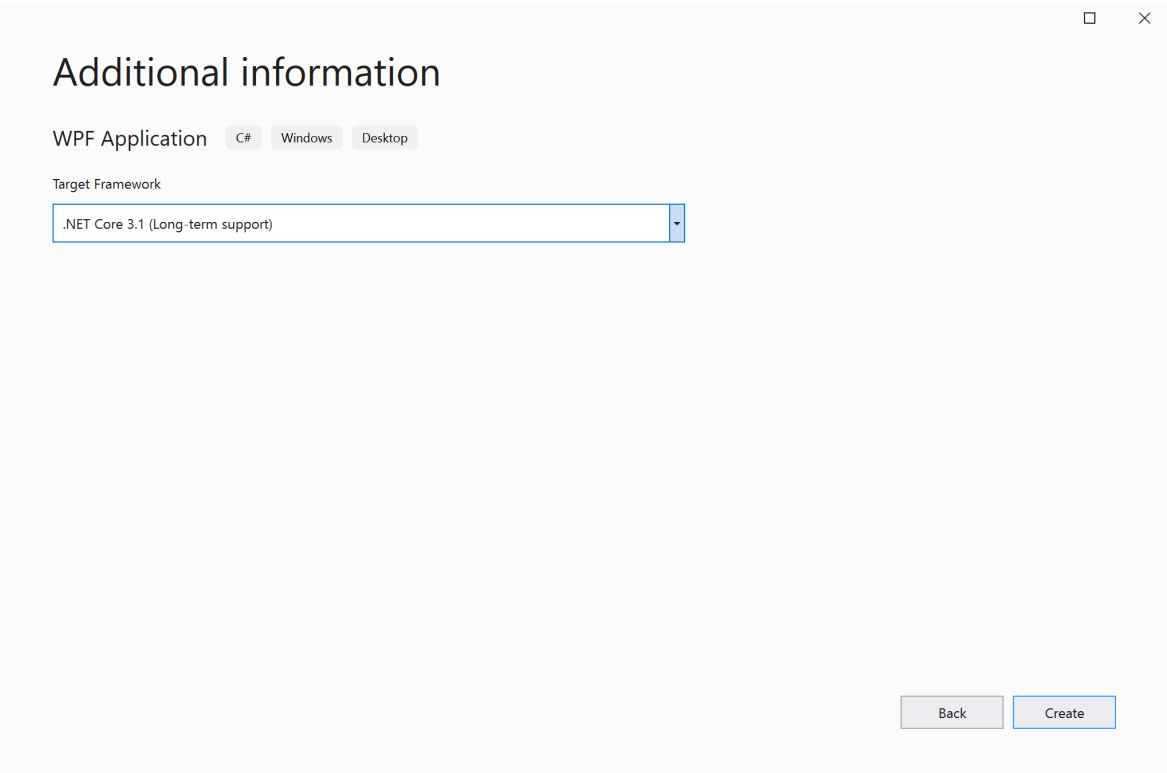
3. On the **Create a new project** screen, search for "WPF," choose **WPF Application**, and then choose **Next**.



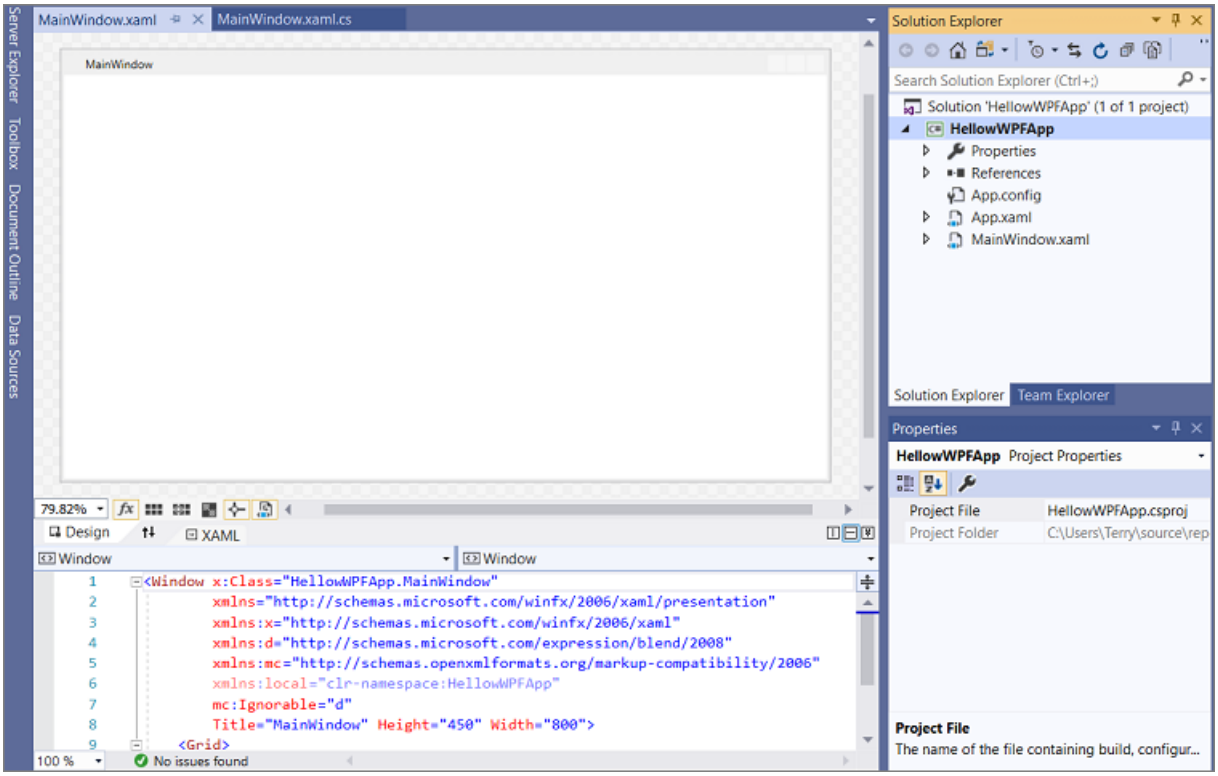
4. At the next screen, give the project a name, **HelloWPFApp**, and choose **Next**.



5. In the **Additional information** window, **.NET Core 3.1** should already be selected for your target framework. If not, select **.NET Core 3.1**. Then, choose **Create**.

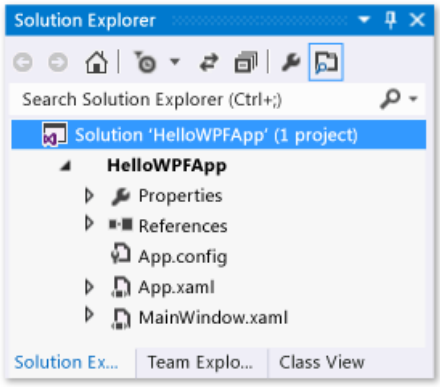


Visual Studio creates the HelloWPFApp project and solution, and **Solution Explorer** shows the various files. The **WPF Designer** shows a design view and a XAML view of *MainWindow.xaml* in a split view. You can slide the splitter to show more or less of either view. You can choose to see only the visual view or only the XAML view.

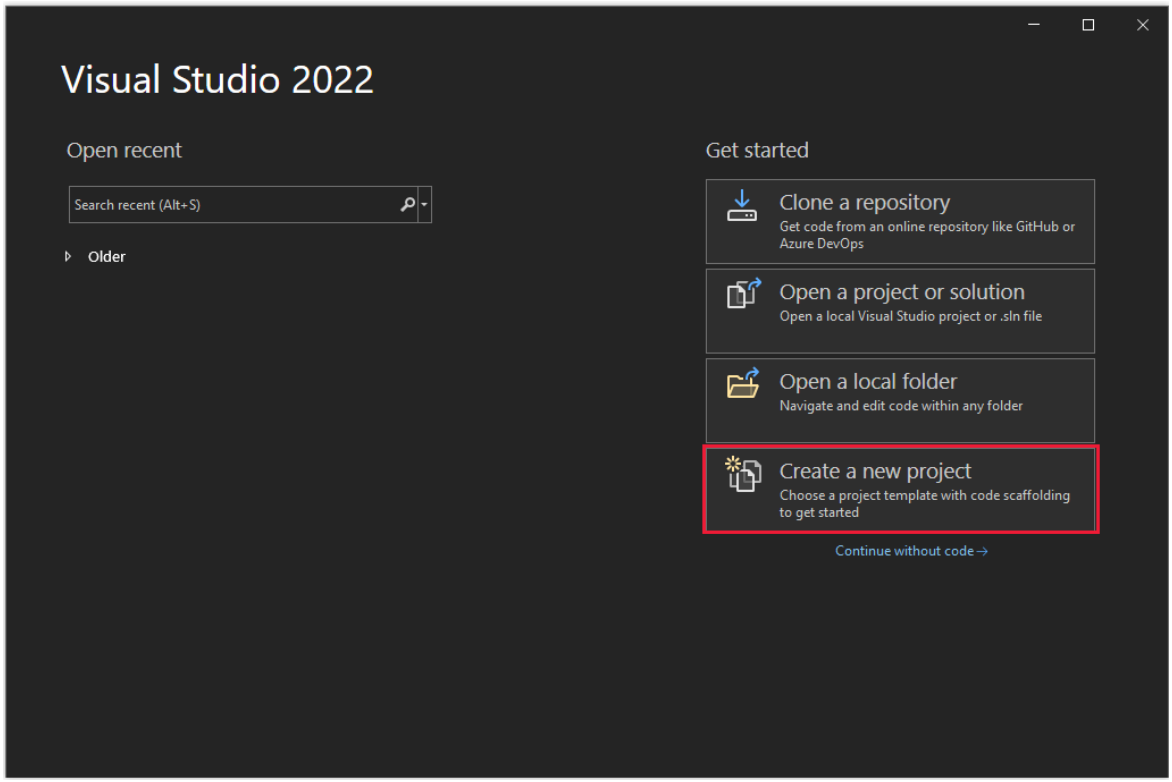


> **NOTE**
>
> For more information about XAML (eXtensible Application Markup Language), see the XAML overview for WPF page.

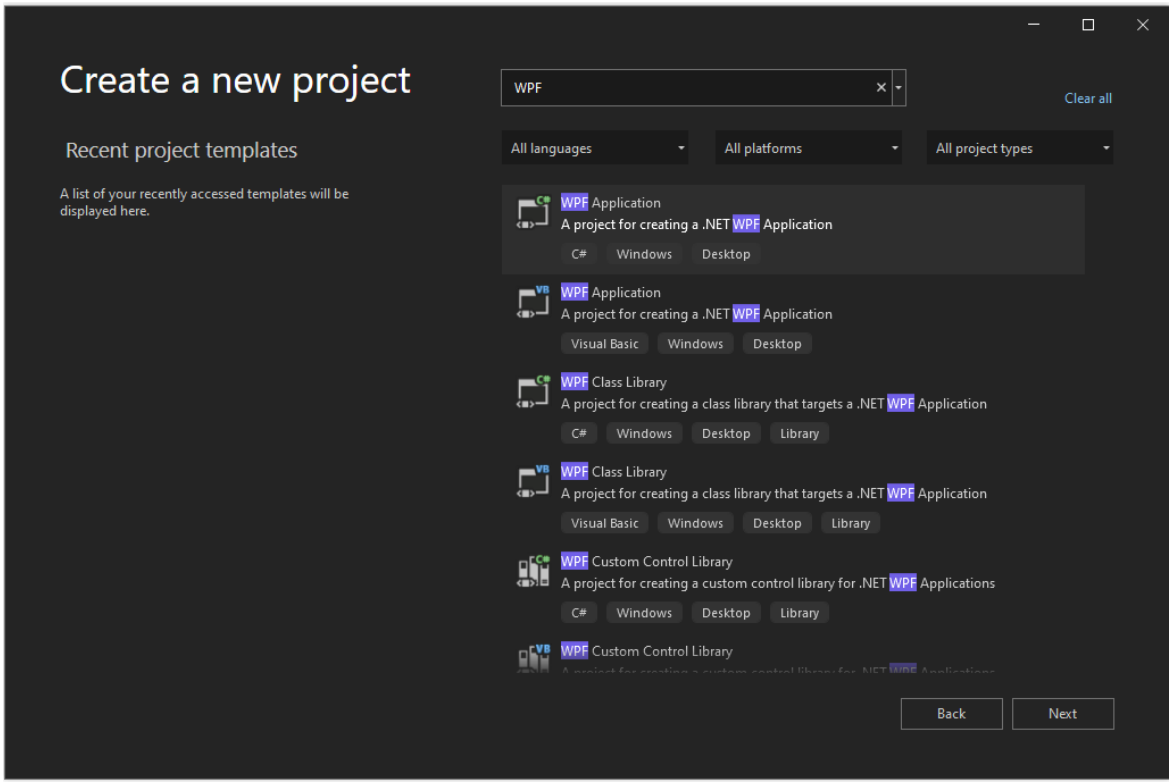
After you create the project, you can customize it. To do so, choose **Properties Window** from the **View** menu, or press **F4**. Then, you can display and change options for project items, controls, and other items in an application.



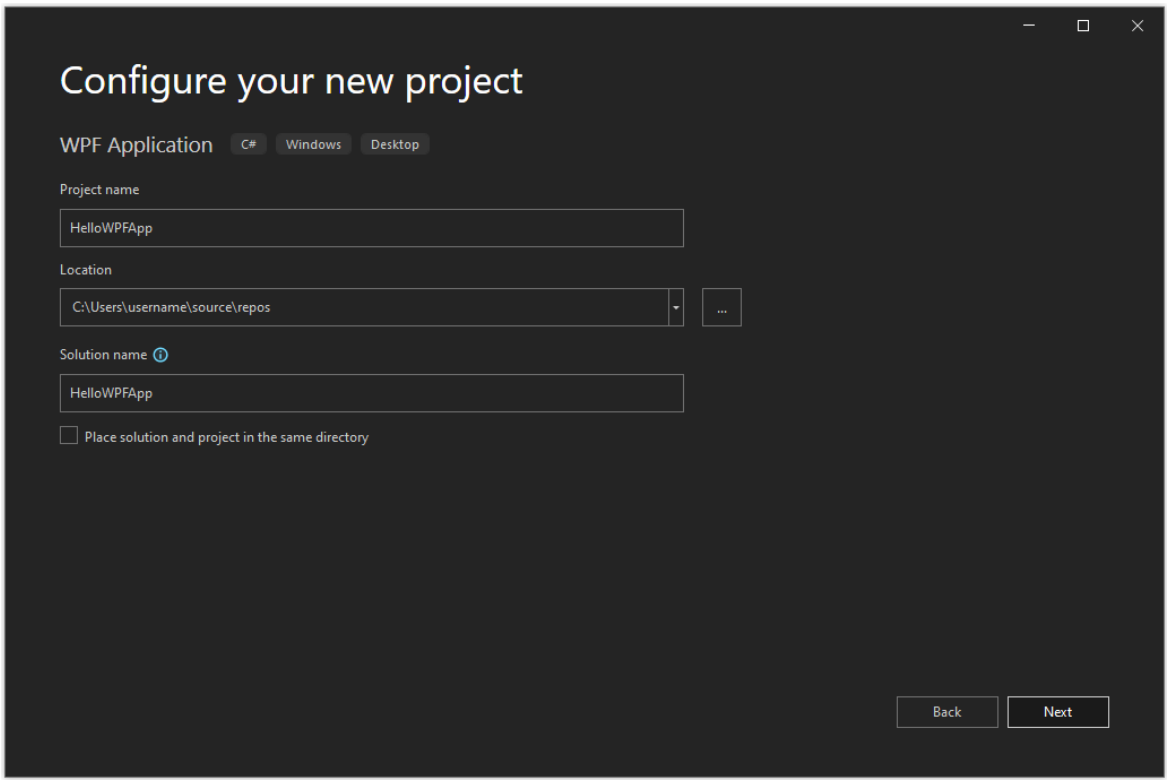
1. Open Visual Studio.
2. On the start window, choose **Create a new project**.



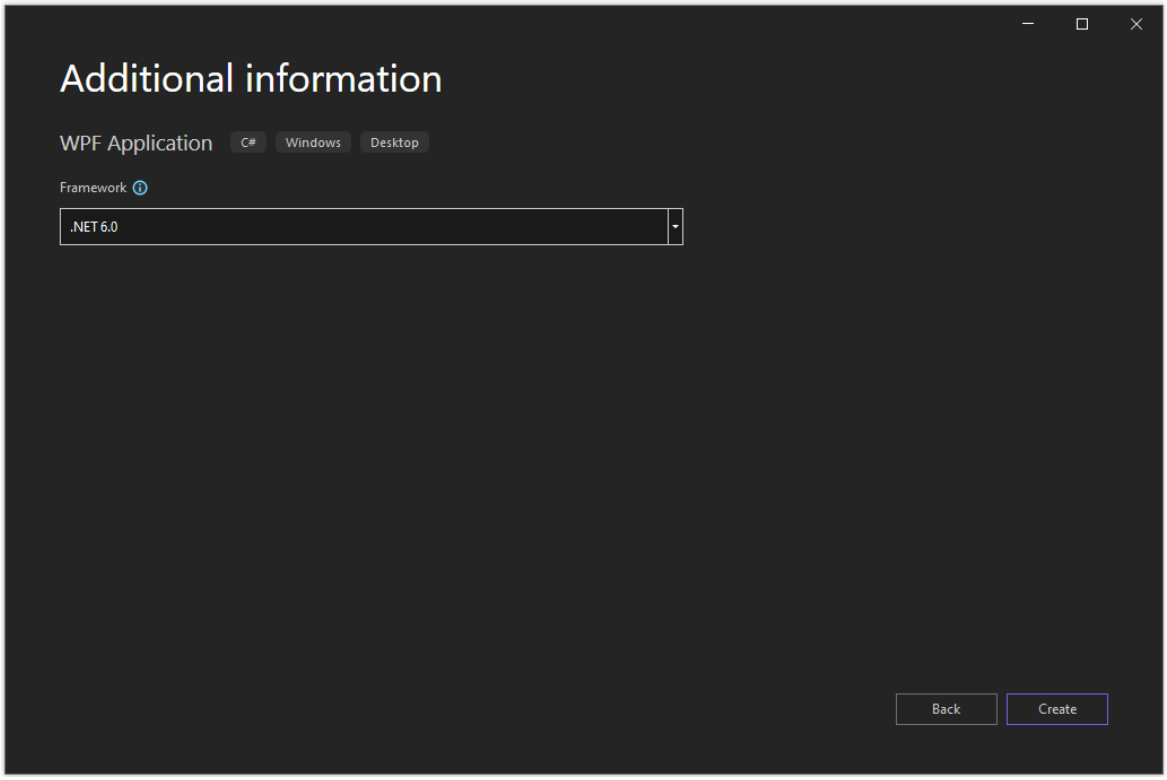
3. On the **Create a new project** screen, search for "WPF," choose **WPF Application**, and then choose **Next**.



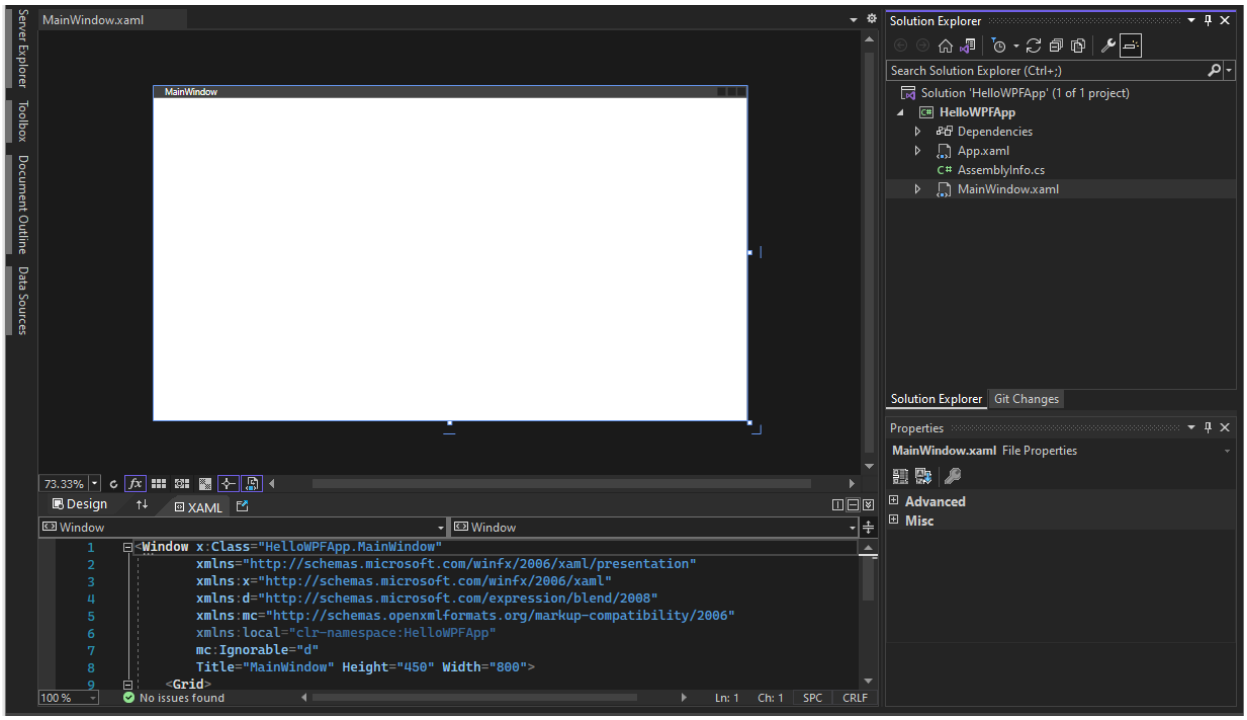
4. At the next screen, give the project a name, **HelloWPFApp**, and choose **Next**.



5. In the **Additional information** window, **.NET 6.0** should already be selected for your target framework. If not, select **.NET 6.0**. Then, choose **Create**.

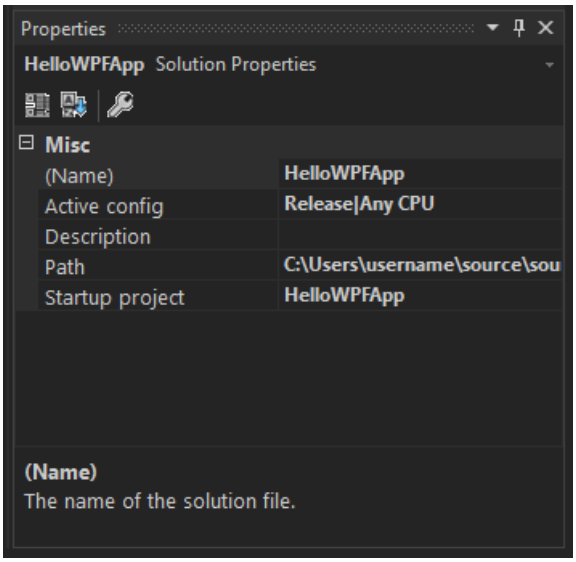


Visual Studio creates the HelloWPFApp project and solution, and **Solution Explorer** shows the various files. The **WPF Designer** shows a design view and a XAML view of *MainWindow.xaml* in a split view. You can slide the splitter to show more or less of either view. You can choose to see only the visual view or only the XAML view.



> **NOTE**
>
> For more information about XAML (eXtensible Application Markup Language), see the XAML overview for WPF page.

After you create the project, you can customize it. To do so, choose **Properties Window** from the **View** menu, or press **F4**. Then, you can display and change options for project items, controls, and other items in an application.



Change the name of MainWindow.xaml

Let's give MainWindow a more specific name. In **Solution Explorer**, right-click on *MainWindow.xaml* and choose **Rename**. Rename the file to *Greetings.xaml*.

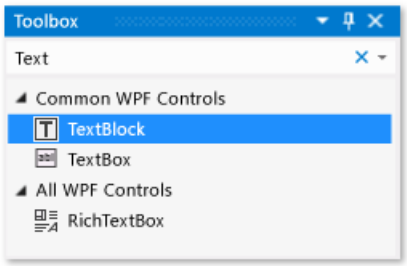
Design the user interface (UI)

If the designer is not open, select *Greetings.xaml* and press **Shift+F7** to open the designer.

We'll add three types of controls to this application: a TextBlock control, two RadioButton controls, and a Button control.

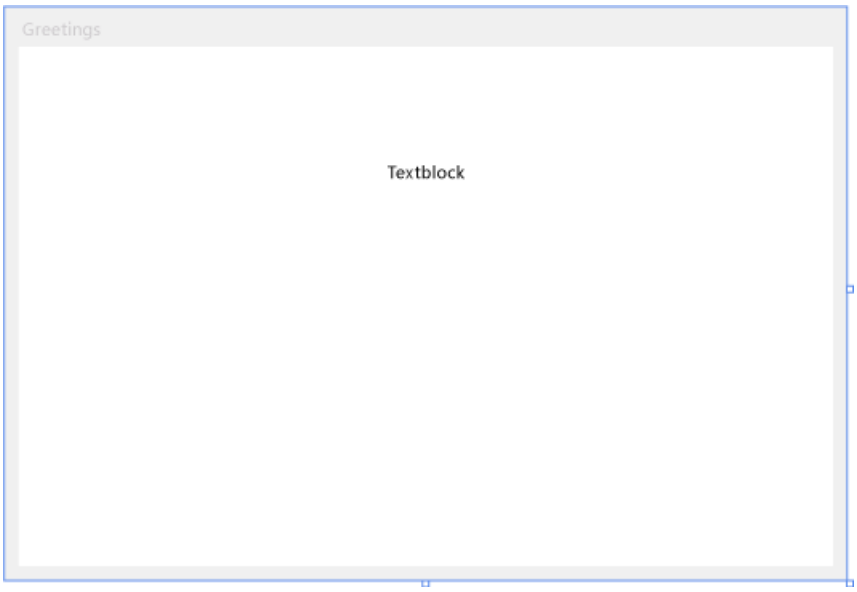
Add a TextBlock control

1. Press **Ctrl+Q** to activate the search box and type **Toolbox**. Choose **View > Toolbox** from the results list.
2. In the **Toolbox**, expand the **Common WPF Controls** node to see the TextBlock control.



3. Add a TextBlock control to the design surface by choosing the **TextBlock** item and dragging it to the window on the design surface. Center the control near the top of the window. In Visual Studio 2019 and later, you can use the red guidelines to center the control.

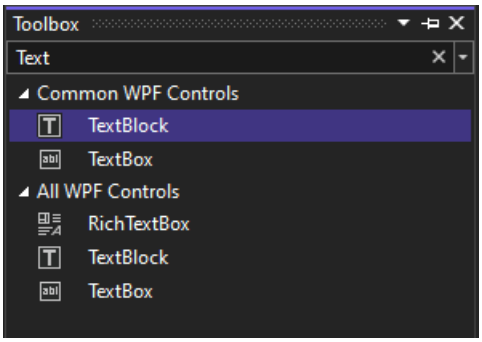
   Your window should resemble the following illustration:



The XAML markup should look something like the following example:

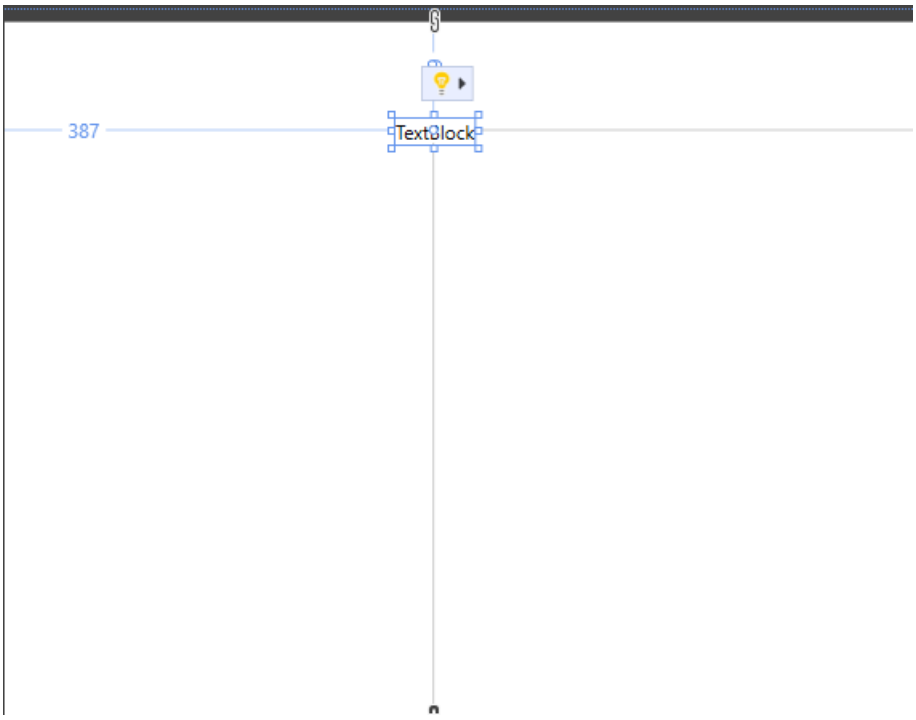
```
<Grid>
    <TextBlock HorizontalAlignment="Left" Margin="387,60,0,0" TextWrapping="Wrap" Text="TextBlock"
VerticalAlignment="Top"/>
</Grid>
```

1. Press **Ctrl+Q** to activate the search box and type **Toolbox**. Choose **View > Toolbox** from the results list.
2. In the **Toolbox**, expand the **Common WPF Controls** node to see the TextBlock control.



3. Add a TextBlock control to the design surface by choosing the **TextBlock** item and dragging it to the window on the design surface. Center the control near the top of the window. You can use the guidelines to center the control.

   Your window should resemble the following image:



The XAML markup should look something like the following example:

```
<Grid>
    <TextBlock HorizontalAlignment="Left" Margin="387,60,0,0" TextWrapping="Wrap" Text="TextBlock"
VerticalAlignment="Top"/>
</Grid>
```

Customize the text in the text block

1. In the XAML view, locate the markup for **TextBlock** and change the **Text** attribute from `TextBox` to `Select a message option and then choose the Display button.`

   The XAML markup should look something like the following example:

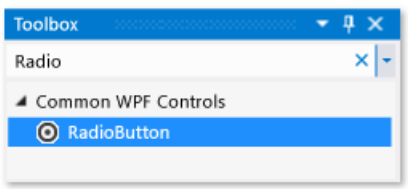
   ```
   <Grid>
       <TextBlock HorizontalAlignment="Left" Margin="387,60,0,0" TextWrapping="Wrap" Text="Select a
   message option and then choose the Display button." VerticalAlignment="Top"/>
   </Grid>
   ```

2. Center the TextBlock again if you like, and then save your changes by pressing **Ctrl+S** or using the **File** menu item.

Next, you'll add two RadioButton controls to the form.

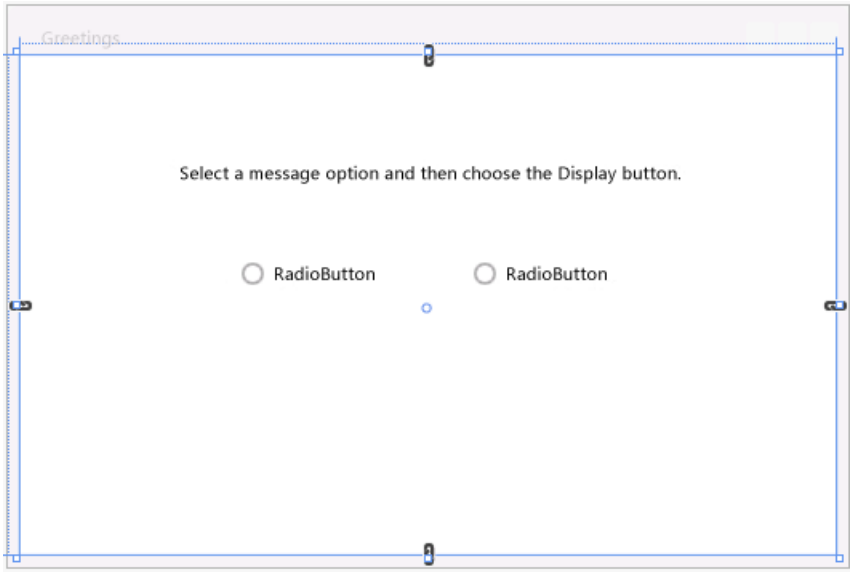
Add radio buttons

1. In the **Toolbox**, find the **RadioButton** control.

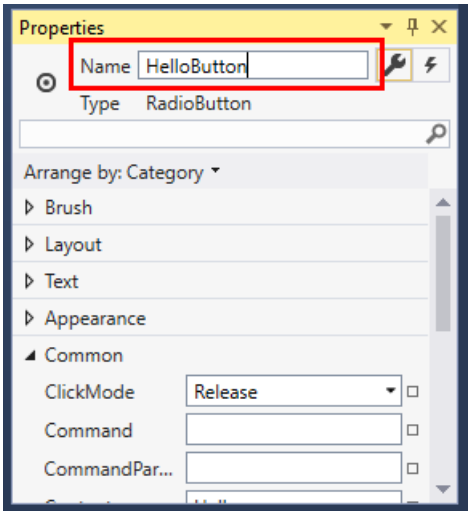
   

2. Add two RadioButton controls to the design surface by choosing the **RadioButton** item and dragging it to the window on the design surface. Move the buttons (by selecting them and using the arrow keys) so that the buttons appear side by side under the TextBlock control. Use the red guidelines to align the controls.

Your window should look like this:



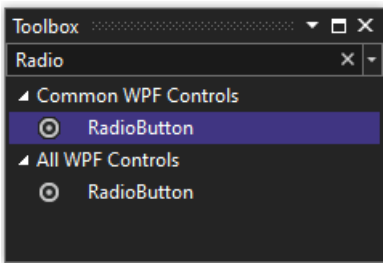
3. In the **Properties** window for the left RadioButton control, change the **Name** property (the property at the top of the **Properties** window) to `HelloButton`.



4. In the **Properties** window for the right RadioButton control, change the **Name** property to `GoodbyeButton`, and then save your changes.

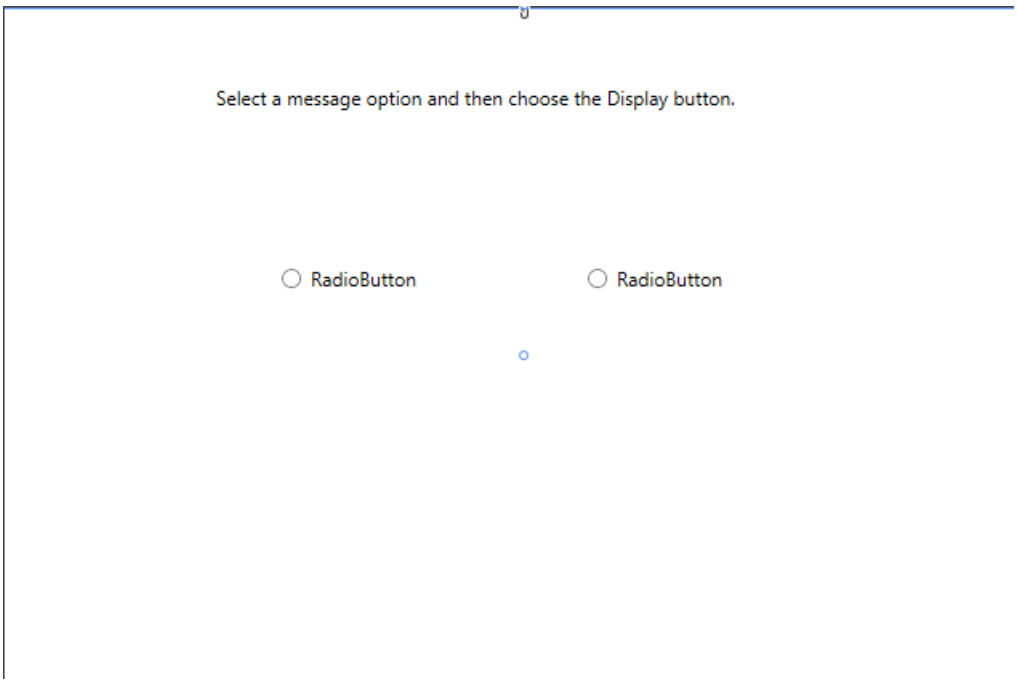
Next, you'll add display text for each RadioButton control. The following procedure updates the **Content** property for a RadioButton control.

1. In the **Toolbox**, find the **RadioButton** control.

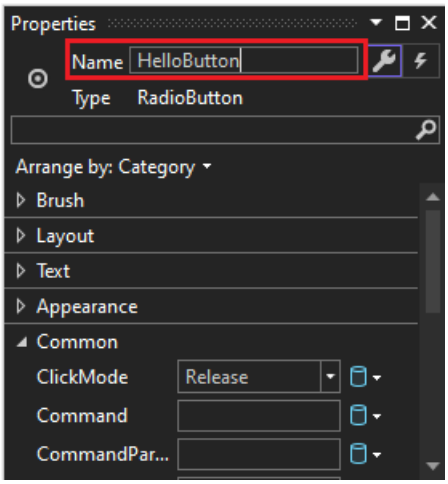


2. Add two RadioButton controls to the design surface by choosing the **RadioButton** item and dragging it to the window on the design surface. Move the buttons (by selecting them and using the arrow keys) so that the buttons appear side by side under the TextBlock control. You can use the guidelines to align the controls.

Your window should look like this:



3. In the **Properties** window for the left RadioButton control, change the **Name** property (the property at the top of the **Properties** window) to `HelloButton`.



4. In the **Properties** window for the right RadioButton control, change the **Name** property to `GoodbyeButton`, and then save your changes.

Next, you'll add display text for each RadioButton control. The following procedure updates the **Content** property for a RadioButton control.

Add display text for each radio button

1. Update the **Content** attribute for the `HelloButton` and `GoodbyeButton` to `"Hello"` and `"Goodbye"` in the XAML. The XAML markup should now look similar to the following example:

```
<Grid>
     <TextBlock HorizontalAlignment="Left" Margin="252,47,0,0" TextWrapping="Wrap" Text="Select a
message option and then choose the Display button." VerticalAlignment="Top"/>
     <RadioButton x:Name="HelloButton" Content="Hello" HorizontalAlignment="Left"
Margin="297,161,0,0" VerticalAlignment="Top"/>
     <RadioButton x:Name="GoodbyeButton" Content="Goodbye" HorizontalAlignment="Left"
Margin="488,161,0,0" VerticalAlignment="Top"/>
</Grid>
```

Set a radio button to be checked by default

In this step, we'll set HelloButton to be checked by default so that one of the two radio buttons is always selected.

1. In the XAML view, locate the markup for HelloButton.
2. Add an **IsChecked** attribute and set it to **True**. Specifically, add `IsChecked="True"`.

   The XAML markup should now look similar to the following example:

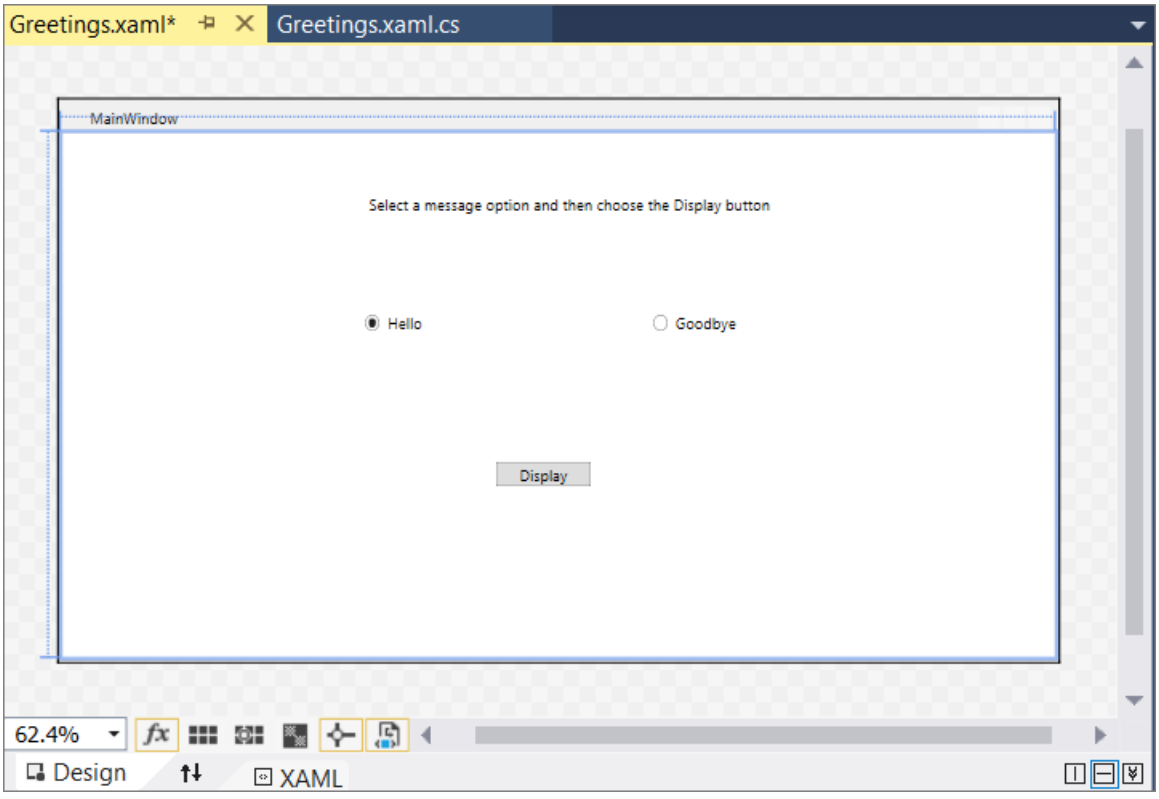
```
<Grid>
     <TextBlock HorizontalAlignment="Left" Margin="252,47,0,0" TextWrapping="Wrap" Text="Select a
message option and then choose the Display button." VerticalAlignment="Top"/>
     <RadioButton x:Name="HelloButton" Content="Hello" IsChecked="True" HorizontalAlignment="Left"
Margin="297,161,0,0" VerticalAlignment="Top"/>
     <RadioButton x:Name="GoodbyeButton" Content="Goodbye" HorizontalAlignment="Left"
Margin="488,161,0,0" VerticalAlignment="Top"/>
</Grid>
```

The final UI element that you'll add is a Button control.

Add the button control

1. In the **Toolbox**, find the **Button** control, and then add it to the design surface under the RadioButton controls by dragging it to the form in the design view. If you're using Visual Studio 2019 or later, a red line helps you center the control.
2. In the XAML view, change the value of **Content** for the Button control from `Content="Button"` to `Content="Display"`, and then save the changes.

   Your window should resemble the following illustration.

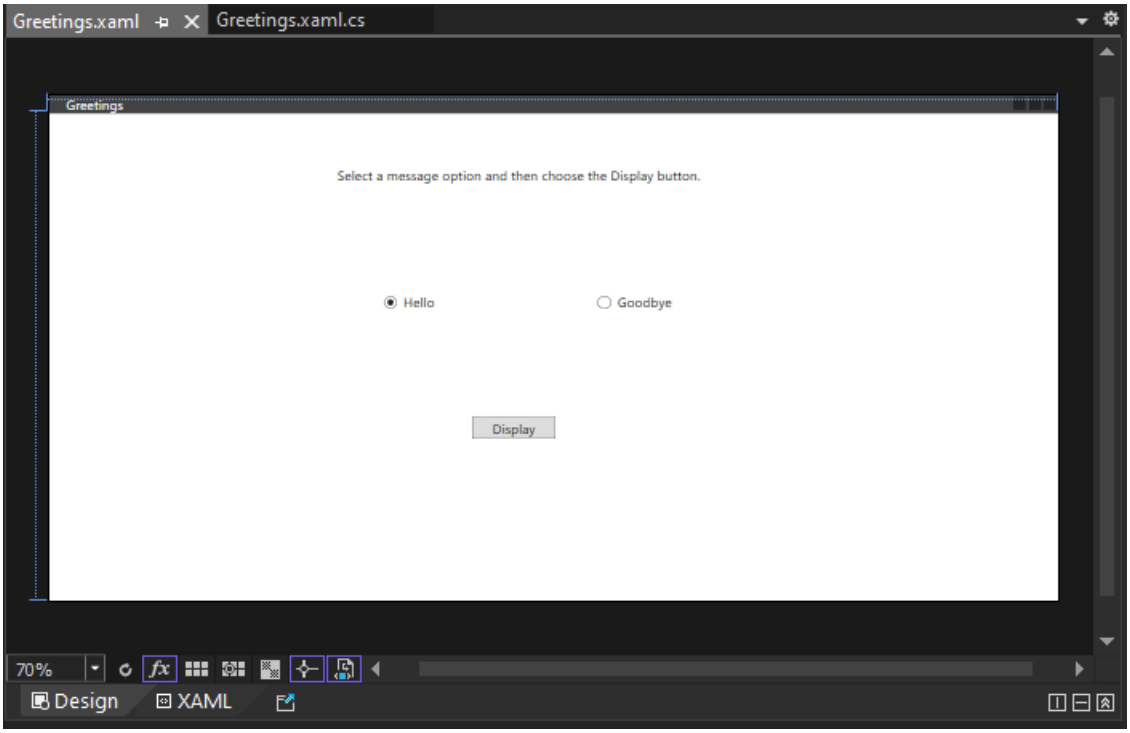


   The XAML markup should now look similar to the following example:

```
<Grid>
     <TextBlock HorizontalAlignment="Left" Margin="252,47,0,0" TextWrapping="Wrap" Text="Select a
message option and then choose the Display button." VerticalAlignment="Top"/>
     <RadioButton x:Name="HelloButton" Content="Hello" IsChecked="True" HorizontalAlignment="Left"
Margin="297,161,0,0" VerticalAlignment="Top"/>
     <RadioButton x:Name="GoodbyeButton" Content="Goodbye" HorizontalAlignment="Left"
Margin="488,161,0,0" VerticalAlignment="Top"/>
     <Button Content="Display" HorizontalAlignment="Left" Margin="377,270,0,0"
VerticalAlignment="Top" Width="75"/>
</Grid>
```

1. In the **Toolbox**, find the **Button** control, and then add it to the design surface under the RadioButton controls by dragging it to the form in the design view. The guidelines can help you center the control.
2. In the XAML view, change the value of **Content** for the Button control from `Content="Button"` to `Content="Display"`, and then save the changes.

   Your window should resemble the following screenshot.



   The XAML markup should now look similar to the following example:

```
<Grid>
     <TextBlock HorizontalAlignment="Left" Margin="252,47,0,0" TextWrapping="Wrap" Text="Select a
message option and then choose the Display button." VerticalAlignment="Top"/>
     <RadioButton x:Name="HelloButton" Content="Hello" IsChecked="True" HorizontalAlignment="Left"
Margin="297,161,0,0" VerticalAlignment="Top"/>
     <RadioButton x:Name="GoodbyeButton" Content="Goodbye" HorizontalAlignment="Left"
Margin="488,161,0,0" VerticalAlignment="Top"/>
     <Button Content="Display" HorizontalAlignment="Left" Margin="377,270,0,0"
VerticalAlignment="Top" Width="75"/>
</Grid>
```

Add code to the display button

When this application runs, a message box appears after a user chooses a radio button and then chooses the **Display** button. One message box will appear for Hello, and another will appear for Goodbye. To create this behavior, you'll add code to the `Button_Click` event in *Greetings.xaml.cs*.

1. On the design surface, double-click the **Display** button.

   Greetings.xaml.cs opens, with the cursor in the `Button_Click` event.

```
private void Button_Click(object sender, RoutedEventArgs e)
{

}
```

2. Enter the following code:

```
if (HelloButton.IsChecked == true)
{
     MessageBox.Show("Hello.");
}
else if (GoodbyeButton.IsChecked == true)
{
    MessageBox.Show("Goodbye.");
}
```

3. Save the application.

When this application runs, a message box appears after a user chooses a radio button and then chooses the **Display** button. One message box will appear for Hello, and another will appear for Goodbye. To create this behavior, you'll add code to the `Button_Click` event in *Greetings.xaml.cs*.

1. On the design surface, double-click the **Display** button.

   Greetings.xaml.cs opens, with the cursor in the `Button_Click` event.

```
private void Button_Click(object sender, RoutedEventArgs e)
{

}
```

2. Enter the following code:

```
if (HelloButton.IsChecked == true)
{
     MessageBox.Show("Hello.");
}
else if (GoodbyeButton.IsChecked == true)
{
    MessageBox.Show("Goodbye.");
}
```

3. Save the application.

Debug and test the application

Next, you'll debug the application to look for errors and test that both message boxes appear correctly. The following instructions tell you how to build and launch the debugger, but later you might read Build a WPF application (WPF) and Debug WPF for more information.

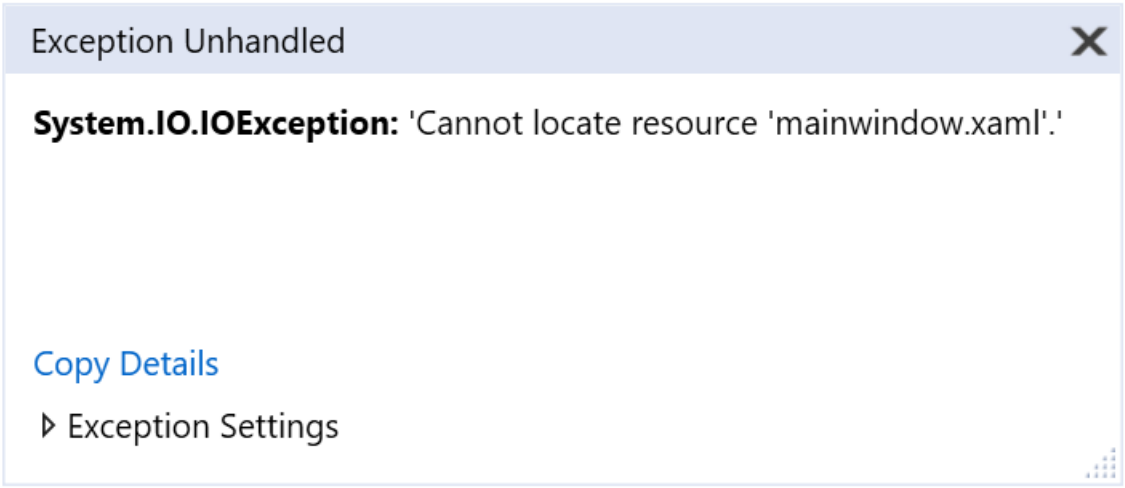
Find and fix errors

In this step, you'll find the error that we caused earlier by changing the name of the *MainWindow.xaml* file.

Start debugging and find the error

1. Start the debugger by pressing **F5** or selecting **Debug**, then **Start Debugging**.

   A **Break Mode** window appears, and the **Output** window indicates that an IOException has occurred: Cannot locate resource 'mainwindow.xaml'.

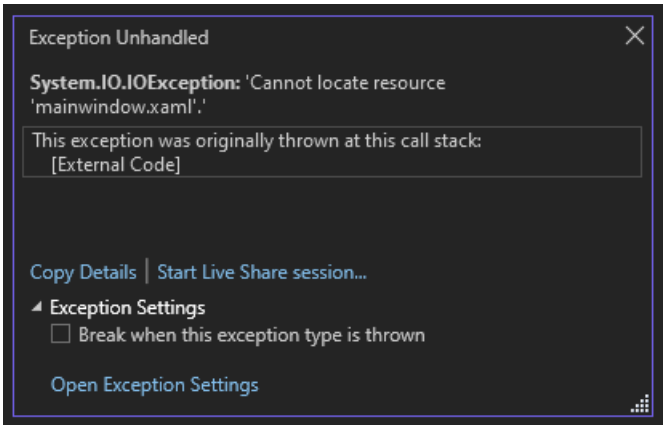


2. Stop the debugger by choosing **Debug** > **Stop Debugging**.

We renamed *MainWindow.xaml* to *Greetings.xaml* at the start of this tutorial, but the code still refers to *MainWindow.xaml* as the startup URI for the application, so the project can't start.

1. Start the debugger by pressing **F5** or selecting **Debug**, then **Start Debugging**.

   A **Break Mode** window appears, and the **Output** window indicates that an IOException has occurred: Cannot locate resource 'mainwindow.xaml'.



2. Stop the debugger by choosing **Debug** > **Stop Debugging**.

We renamed *MainWindow.xaml* to *Greetings.xaml* at the start of this tutorial, but the code still refers to *MainWindow.xaml* as the startup URI for the application, so the project can't start.

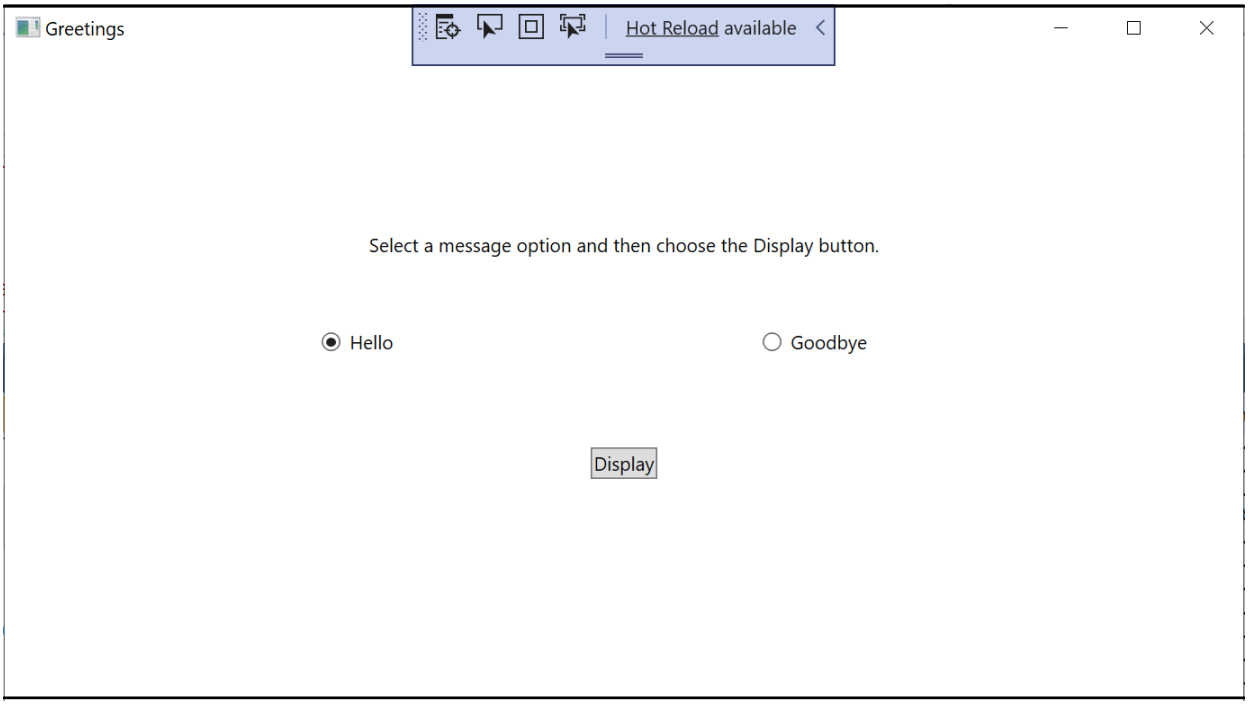
Specify Greetings.xaml as the startup URI

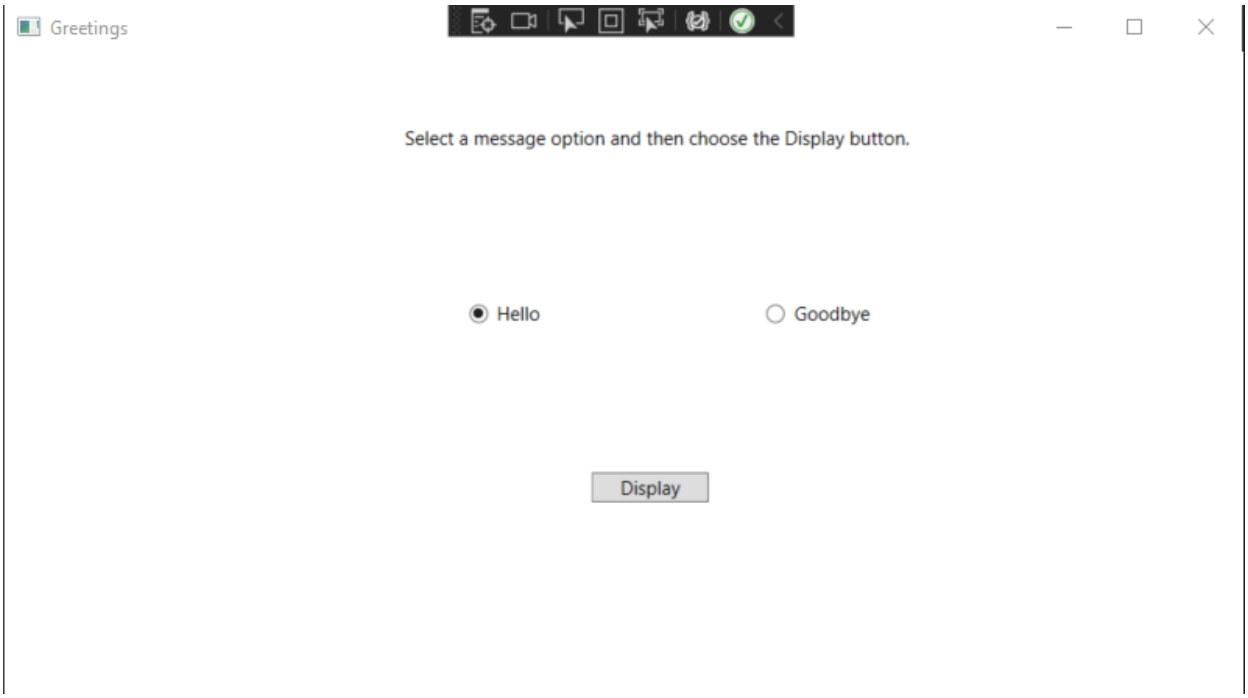
1. In **Solution Explorer**, open the *App.xaml* file.
2. Change `StartupUri="MainWindow.xaml"` to `StartupUri="Greetings.xaml"`, and save the changes.

As an optional step, it will avoid confusion to change the title of your application window to match this new name.

1. In **Solution Explorer**, open the *Greetings.xaml* file that you just renamed.
2. Change the value of the **Window.Title** property from `Title="MainWindow"` to `Title="Greetings"`, and save the changes.

Start the debugger again (press **F5**). You should now see the **Greetings** window of your application.





Now close the application window to stop debugging.

Debug with breakpoints

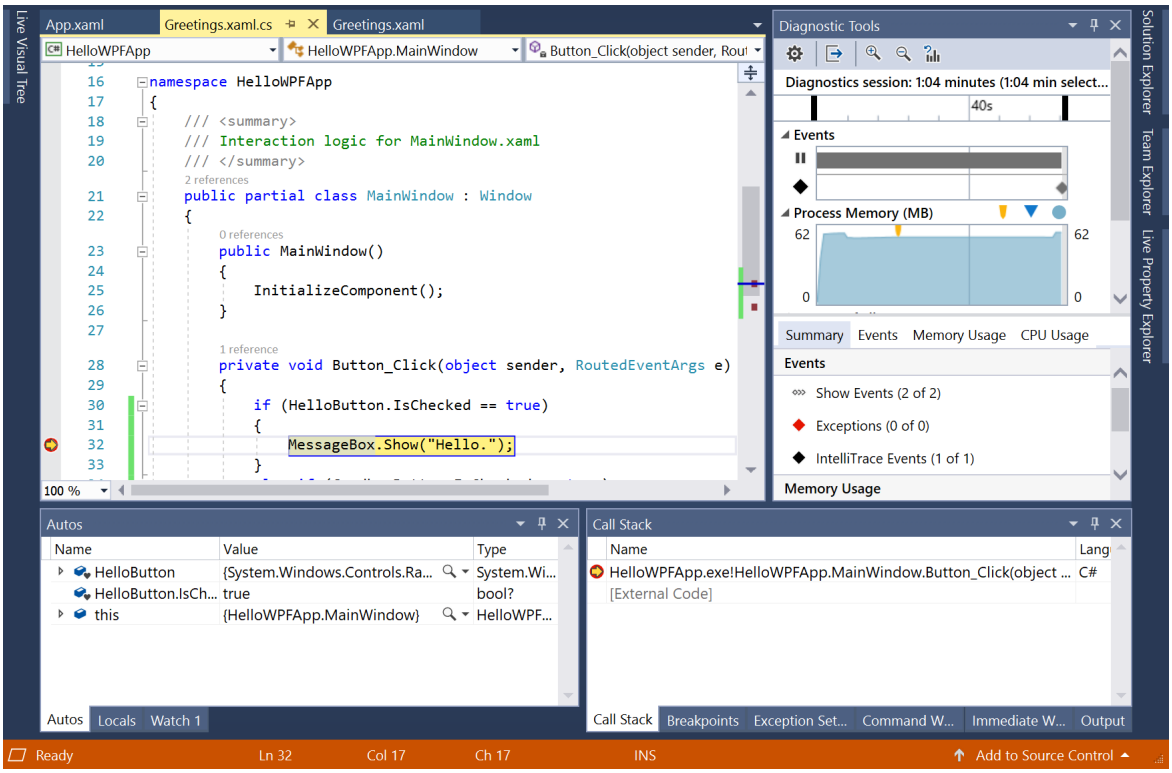
You can test the code during debugging by adding some breakpoints. You can add breakpoints by choosing **Debug** > **Toggle Breakpoint**, by clicking in the left margin of the editor next to the line of code where you want the break to occur, or by pressing **F9**.

Add breakpoints

1. Open *Greetings.xaml.cs*, and select the following line: `MessageBox.Show("Hello.")`
2. Add a breakpoint from the menu by selecting **Debug**, then **Toggle Breakpoint**.

   A red circle appears next to the line of code in the far left margin of the editor window.
3. Select the following line: `MessageBox.Show("Goodbye.")`.
4. Press the **F9** key to add a breakpoint, and then press **F5** to start debugging.
5. In the **Greetings** window, choose the **Hello** radio button, and then choose the **Display** button.

   The line `MessageBox.Show("Hello.")` is highlighted in yellow. At the bottom of the IDE, the Autos, Locals, and Watch windows are docked together on the left side, and the Call Stack, Breakpoints, Exception Settings, Command, Immediate, and Output windows are docked together on the right side.



6. On the menu bar, choose **Debug** > **Step Out**.

   The application resumes execution, and a message box with the word "Hello" appears.

7. Choose the **OK** button on the message box to close it.

8. In the **Greetings** window, choose the **Goodbye** radio button, and then choose the **Display** button.

   The line `MessageBox.Show("Goodbye.")` is highlighted in yellow.

9. Choose the **F5** key to continue debugging. When the message box appears, choose the **OK** button on the message box to close it.

10. Close the application window to stop debugging.

11. On the menu bar, choose **Debug** > **Disable All Breakpoints**.

1. Open *Greetings.xaml.cs*, and select the following line: `MessageBox.Show("Hello.")`

2. Add a breakpoint from the menu by selecting **Debug**, then **Toggle Breakpoint**.

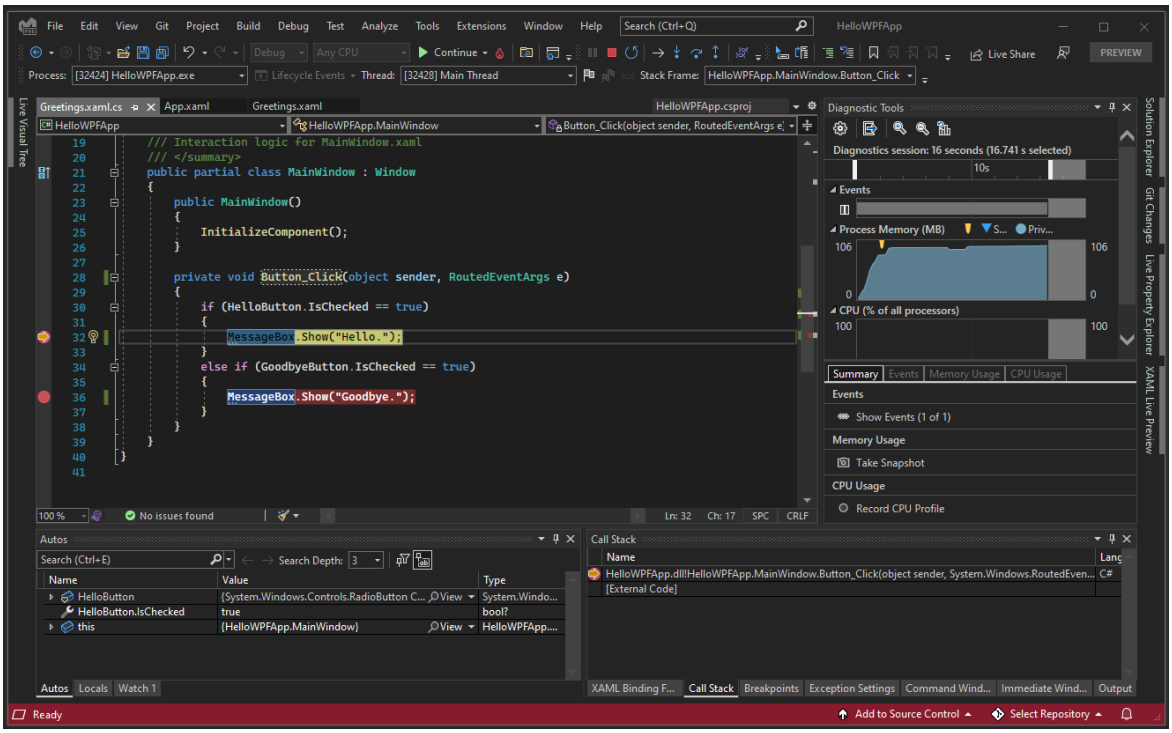
   A red circle appears next to the line of code in the far left margin of the editor window.

3. Select the following line: `MessageBox.Show("Goodbye.")`.

4. Press the **F9** key to add a breakpoint, and then press **F5** to start debugging.

5. In the **Greetings** window, choose the **Hello** radio button, and then choose the **Display** button.

   The line `MessageBox.Show("Hello.")` is highlighted in yellow. At the bottom of the IDE, the Autos, Locals, and Watch windows are docked together on the left side, and the Call Stack, Breakpoints, Exception Settings, Command, Immediate, and Output windows are docked together on the right side.



6. On the menu bar, choose **Debug** > **Step Out**.

   The application resumes execution, and a message box with the word "Hello" appears.

7. Choose the **OK** button on the message box to close it.

8. In the **Greetings** window, choose the **Goodbye** radio button, and then choose the **Display** button.

   The line `MessageBox.Show("Goodbye.")` is highlighted in yellow.

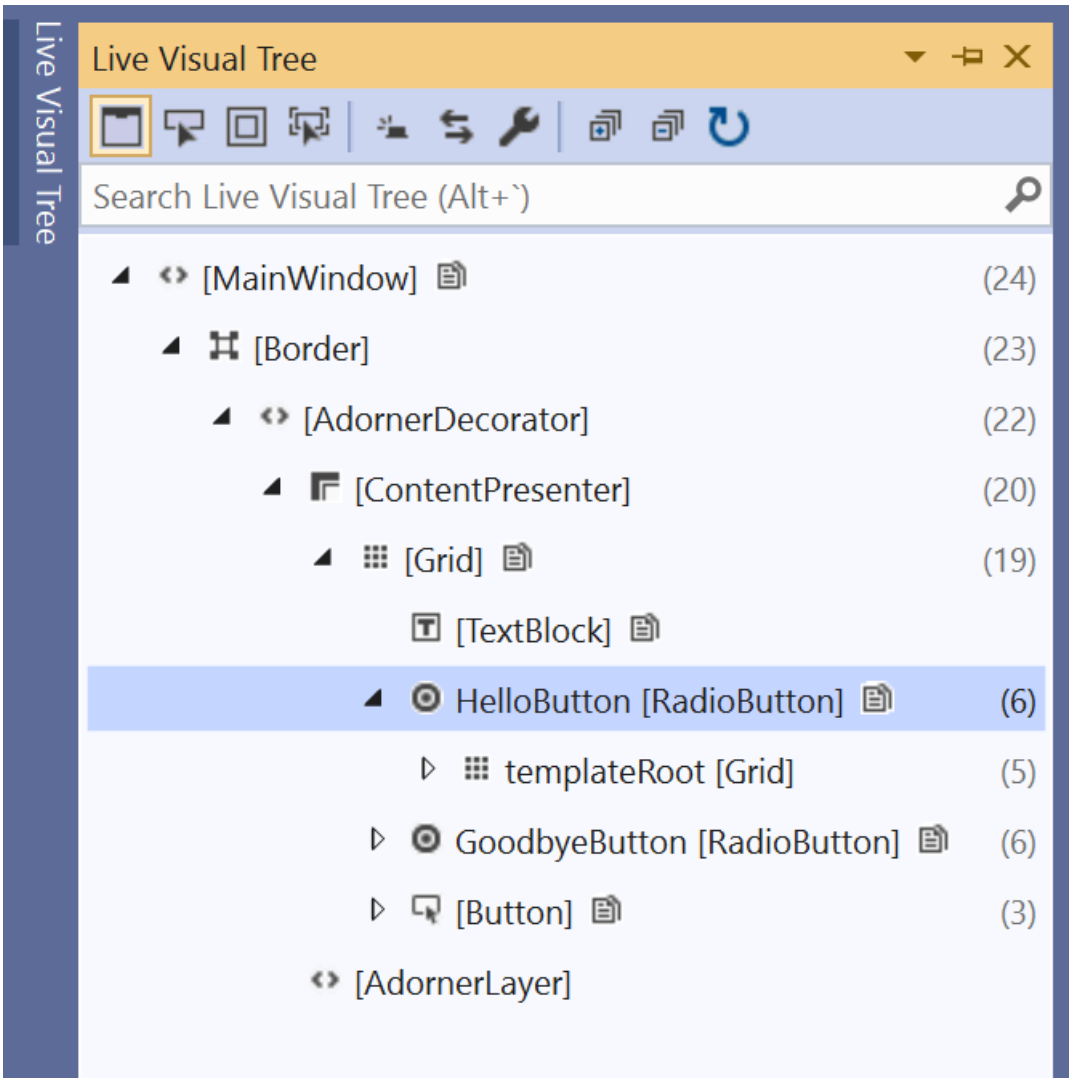
9. Choose the **F5** key to continue debugging. When the message box appears, choose the **OK** button on the message box to close it.

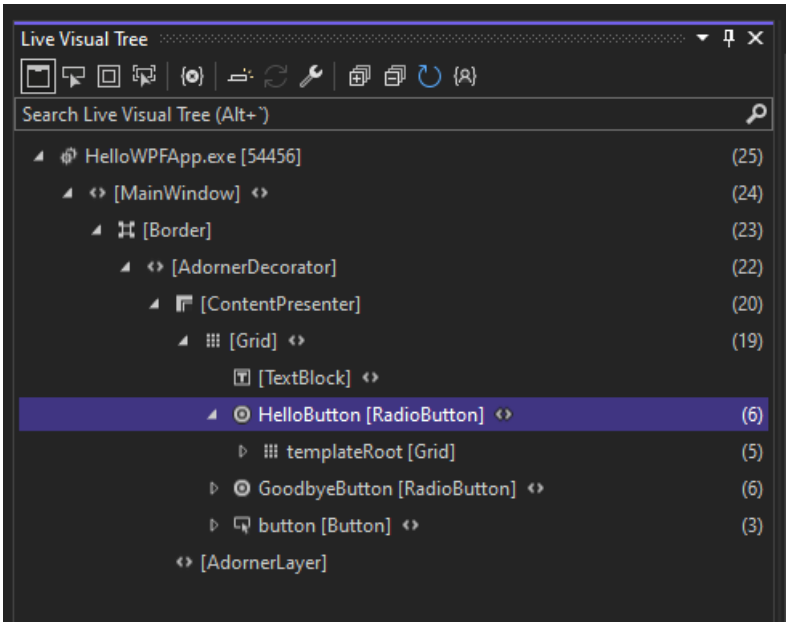
10. Close the application window to stop debugging.

11. On the menu bar, choose **Debug** > **Disable All Breakpoints**.

View a representation of the UI elements

In the running app, you should see a widget that appears at the top of your window. The widget is a runtime helper that provides quick access to some helpful debugging features. Select the first button, **Go to Live Visual Tree**. You should see a window with a tree that contains all the visual elements of your page. Expand the nodes to find the buttons you added.





Build a release version of the application

Now that you've verified that everything works, you can prepare a release build of the application.

1. On the main menu, select **Build** > **Clean solution** to delete intermediate files and output files that were created during previous builds. This step isn't required, but it cleans up the debug build outputs.

2. Change the build configuration for HelloWPFApp from **Debug** to **Release** by using the dropdown control on the toolbar (it says "Debug" currently).

3. Build the solution by choosing **Build** > **Build Solution**.

Congratulations on completing this tutorial! You can find the *.exe* you built under your solution and project directory (...*HelloWPFApp**HelloWPFApp**bin**Release*).

Next steps

Congratulations on completing this tutorial! To learn even more, continue with the following tutorials.

Continue with more WPF tutorials

See also

- Productivity tips

Create a Windows Forms app in Visual Studio with C#

10/28/2021 • 4 minutes to read • Edit Online

In this short introduction to the Visual Studio integrated development environment (IDE), you'll create a simple C# application that has a Windows-based user interface (UI).

If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

> **NOTE**
> Some of the screenshots in this tutorial use the dark theme. If you aren't using the dark theme but would like to, see the Personalize the Visual Studio IDE and Editor page to learn how.

If you haven't already installed Visual Studio, go to the Visual Studio 2022 Preview downloads page to install it for free.

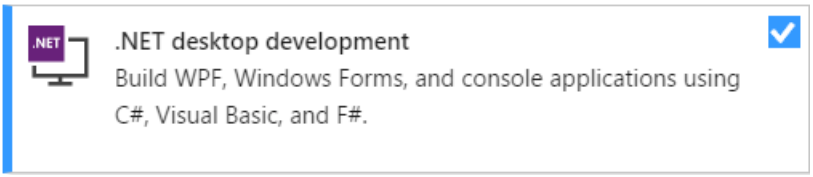
> **NOTE**
> Some of the screenshots in this tutorial use the dark theme. If you aren't using the dark theme but would like to, see the Personalize the Visual Studio IDE and Editor page to learn how.

Create a project

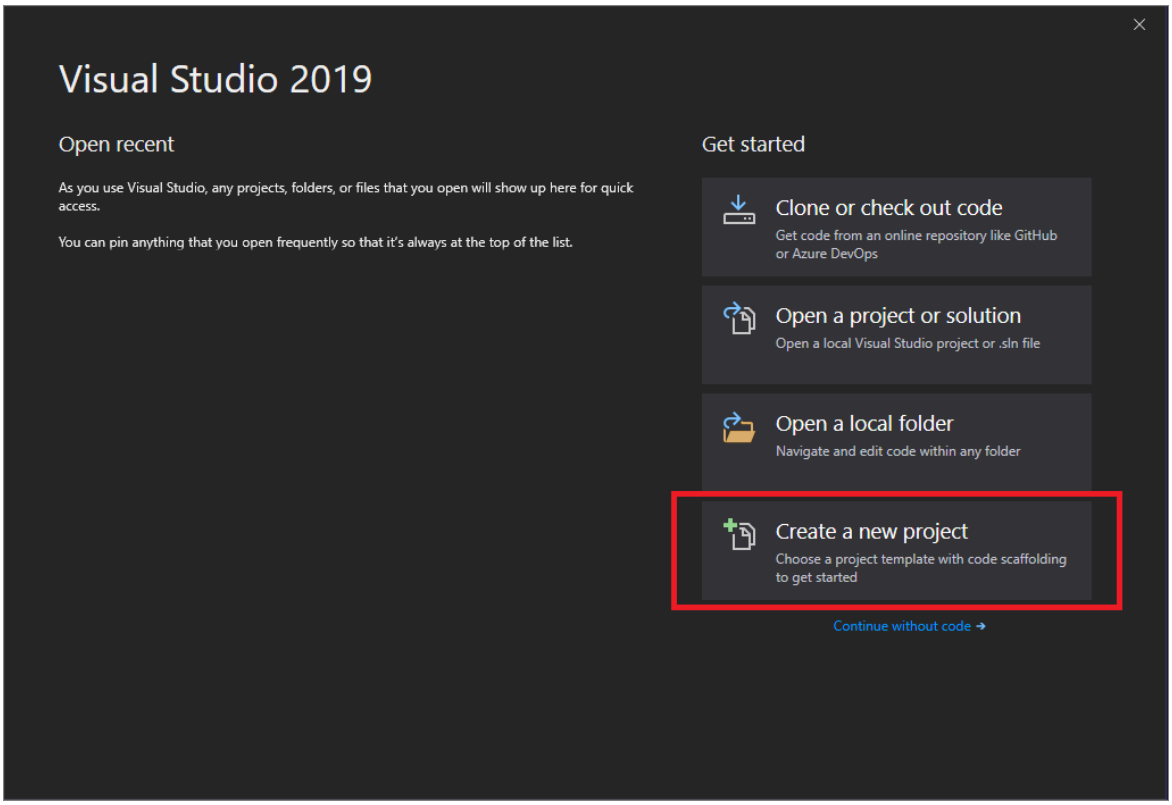
First, you'll create a C# application project. The project type comes with all the template files you'll need, before you've even added anything.

1. Open Visual Studio 2017.
2. From the top menu bar, choose **File** > **New** > **Project**.
3. In the **New Project** dialog box in the left pane, expand **Visual C#**, and then choose **Windows Desktop**. In the middle pane, choose **Windows Forms App (.NET Framework)**. Then name the file `HelloWorld`.

   If you don't see the **Windows Forms App (.NET Framework)** project template, cancel out of the **New Project** dialog box and from the top menu bar, choose **Tools** > **Get Tools and Features**. The Visual Studio Installer launches. Choose the **.NET desktop development** workload, then choose **Modify**.

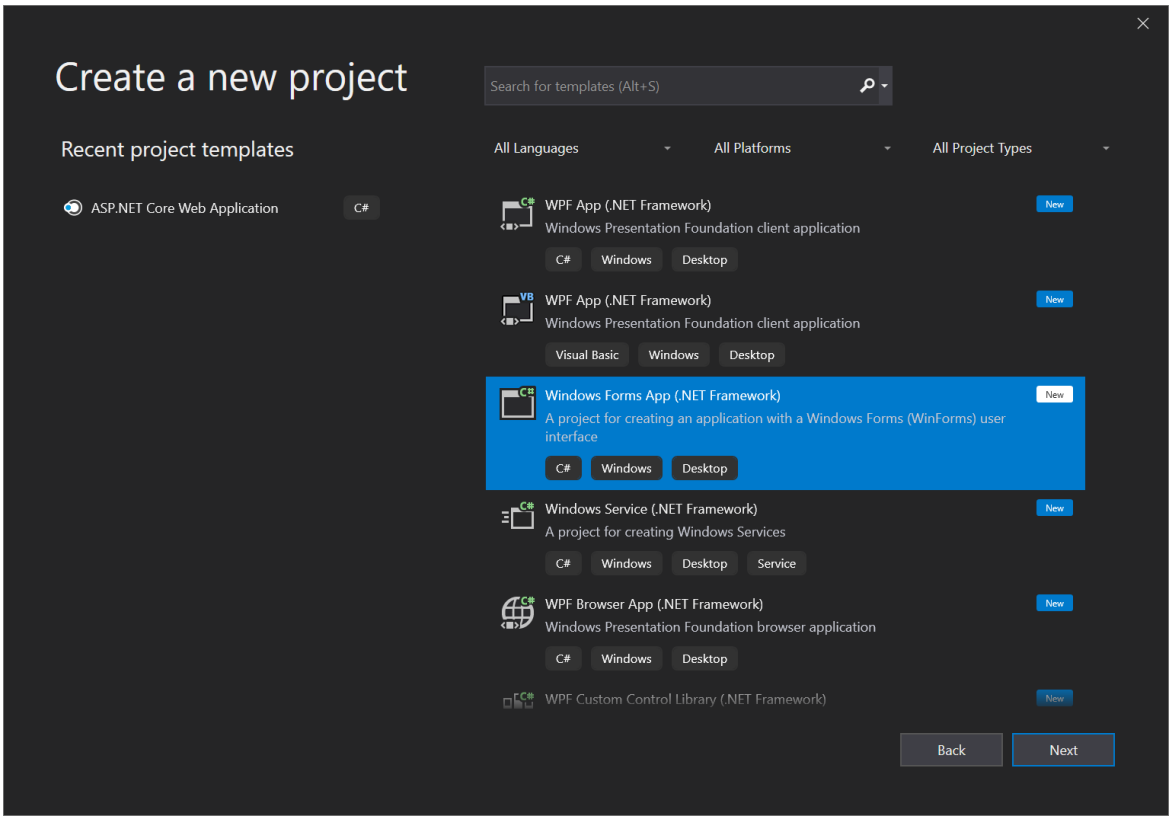


1. Open Visual Studio.
2. On the start window, choose **Create a new project**.



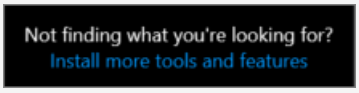
3. On the **Create a new project** window, choose the **Windows Forms App (.NET Framework)** template for C#.

   (If you prefer, you can refine your search to quickly get to the template you want. For example, enter or type *Windows Forms App* in the search box. Next, choose **C#** from the Language list, and then choose **Windows** from the Platform list.)

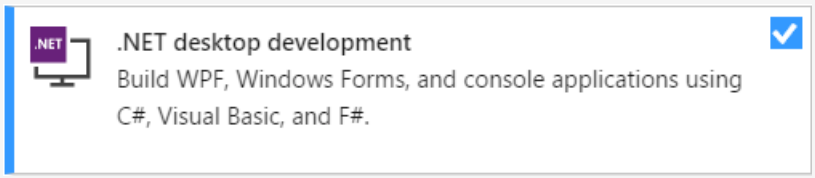


NOTE

If you do not see the **Windows Forms App (.NET Framework)** template, you can install it from the **Create a new project** window. In the **Not finding what you're looking for?** message, choose the **Install more tools and features** link.

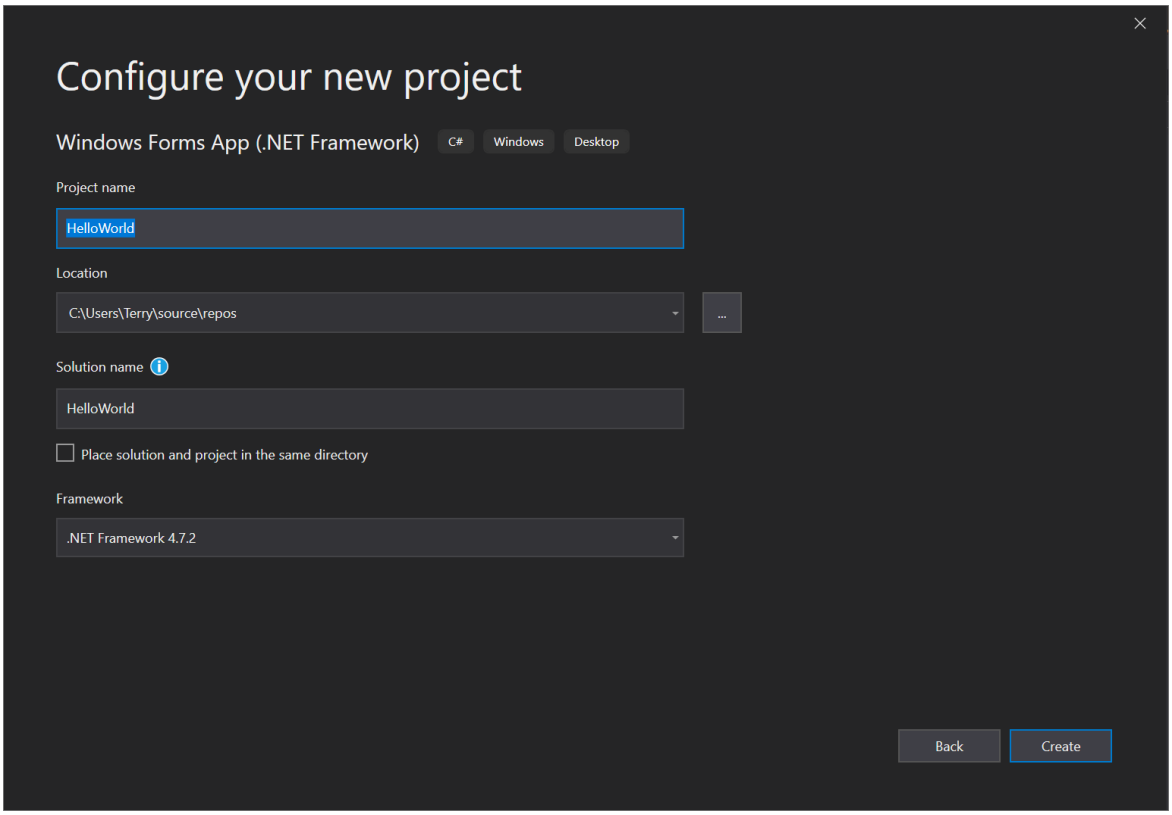


Next, in the Visual Studio Installer, choose the **.NET desktop development** workload.



After that, choose the **Modify** button in the Visual Studio Installer. You might be prompted to save your work; if so, do so. Next, choose **Continue** to install the workload. Then, return to step 2 in this "Create a project" procedure.

4. In the **Configure your new project** window, type or enter *HelloWorld* in the **Project name** box. Then, choose **Create**.



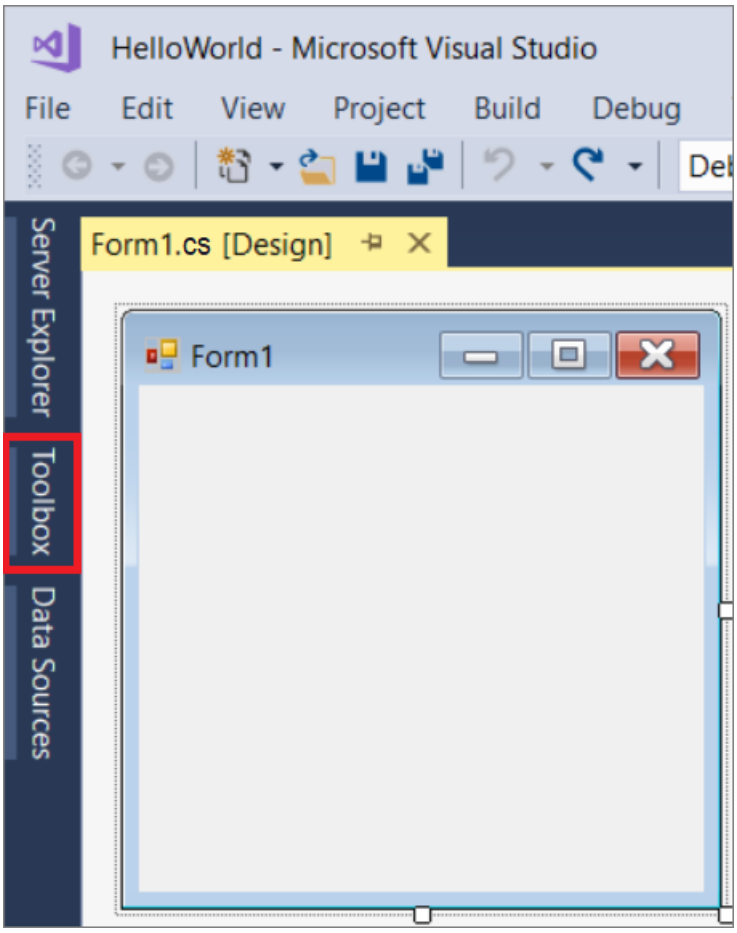
Visual Studio opens your new project.

Create the application

After you select your C# project template and name your file, Visual Studio opens a form for you. A form is a Windows user interface. We'll create a "Hello World" application by adding controls to the form, and then we'll run the app.

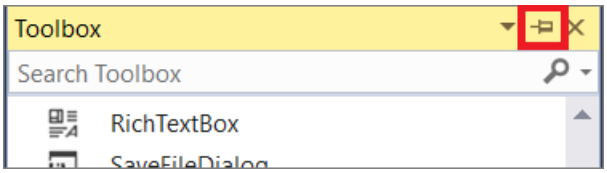
Add a button to the form

1. Choose **Toolbox** to open the Toolbox fly-out window.

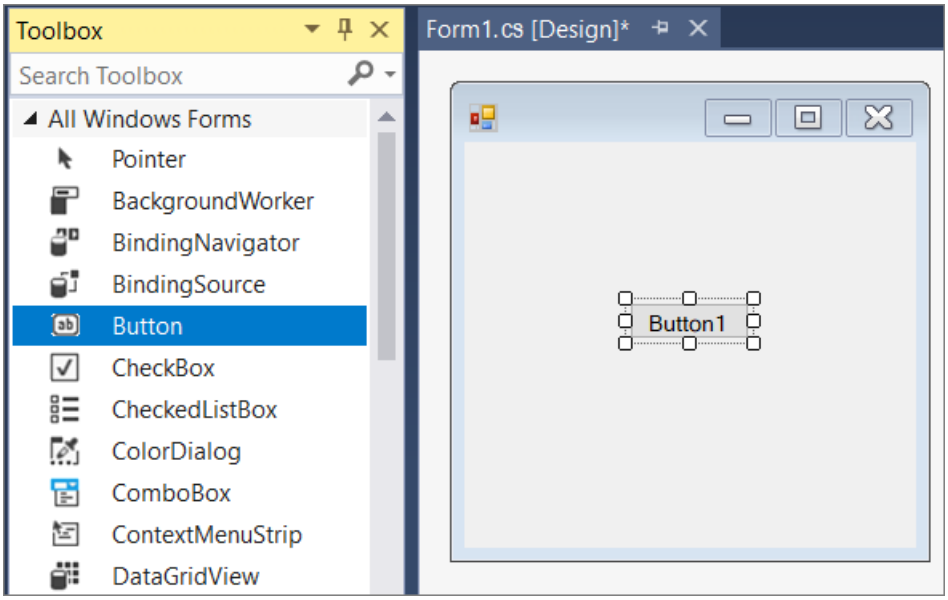


(If you don't see the **Toolbox** fly-out option, you can open it from the menu bar. To do so, **View** > **Toolbox**. Or, press **Ctrl**+**Alt**+**X**.)

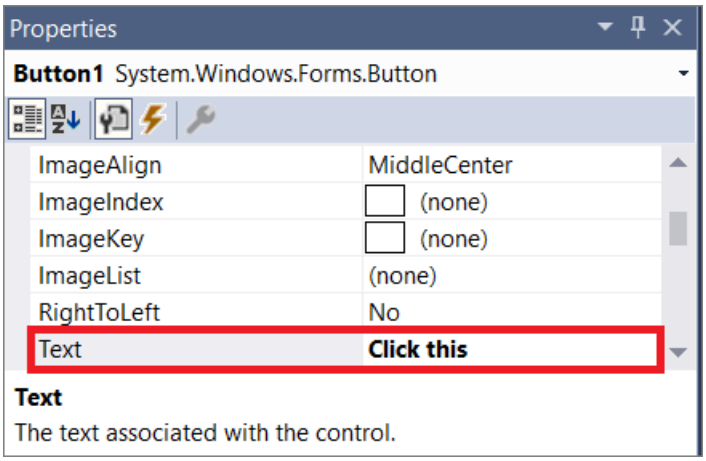
2. Choose the **Pin** icon to dock the **Toolbox** window.



3. Choose the **Button** control and then drag it onto the form.

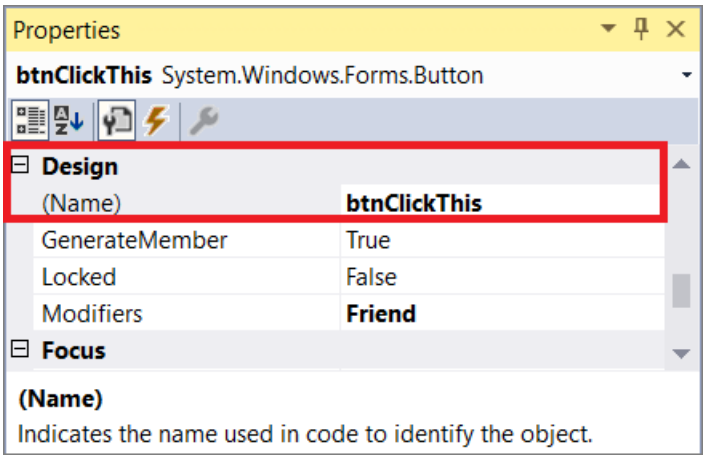


4. In the **Properties** window, locate **Text**, change the name from **Button1** to `Click this`, and then press **Enter**.



(If you don't see the **Properties** window, you can open it from the menu bar. To do so, choose **View** > **Properties Window**. Or, press **F4**.)

5. In the **Design** section of the **Properties** window, change the name from **Button1** to `btnClickThis`, and then press **Enter**.



> **NOTE**
>
> If you've alphabetized the list in the **Properties** window, **Button1** appears in the **(DataBindings)** section, instead.

Add a label to the form

Now that we've added a button control to create an action, let's add a label control to send text to.

1. Select the **Label** control from the **Toolbox** window, and then drag it onto the form and drop it beneath the **Click this** button.

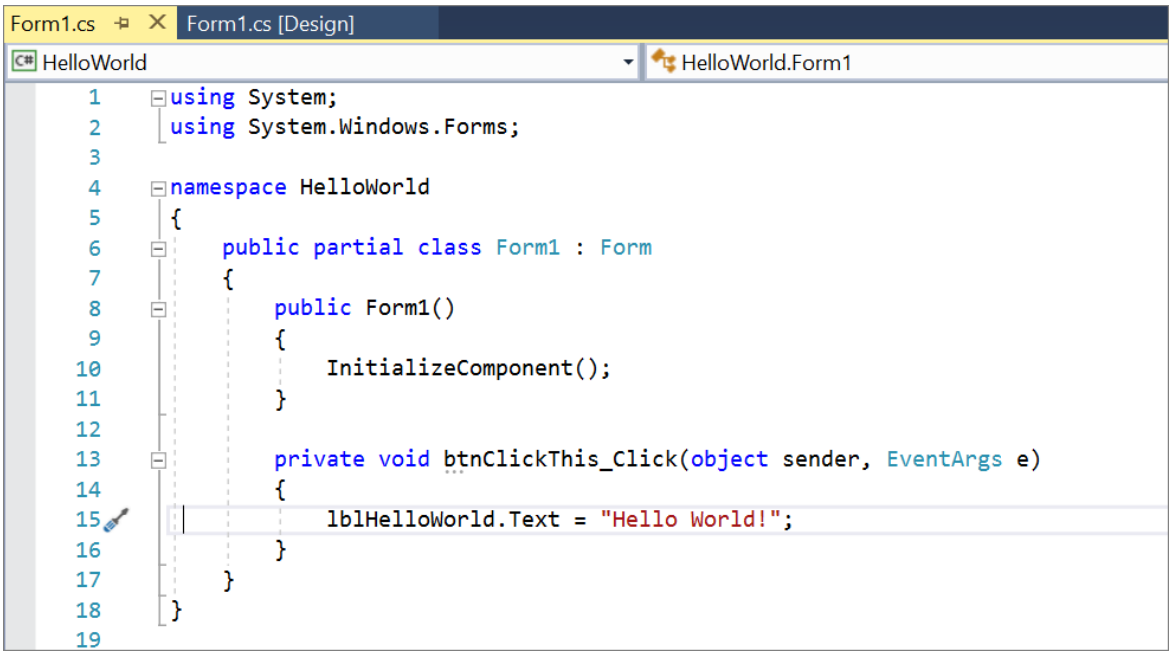
2. In either the **Design** section or the **(DataBindings)** section of the **Properties** window, change the name of **Label1** to `lblHelloWorld`, and then press **Enter**.

Add code to the form

1. In the **Form1.cs [Design]** window, double-click the **Click this** button to open the **Form1.cs** window.

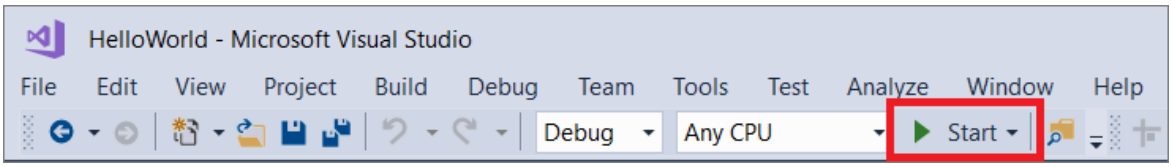
   (Alternatively, you can expand **Form1.cs** in **Solution Explorer**, and then choose **Form1**.)

2. In the **Form1.cs** window, after the **private void** line, type or enter `lblHelloWorld.Text = "Hello World!";` as shown in the following screenshot:



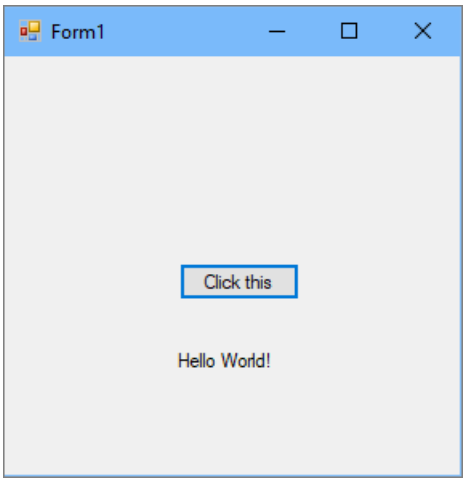
Run the application

1. Choose the **Start** button to run the application.



   Several things will happen. In the Visual Studio IDE, the **Diagnostics Tools** window will open, and an **Output** window will open, too. But outside of the IDE, a **Form1** dialog box appears. It will include your **Click this** button and text that says **Label1**.

2. Choose the **Click this** button in the **Form1** dialog box. Notice that the **Label1** text changes to **Hello World!**.



3. Close the **Form1** dialog box to stop running the app.

Next steps

To learn more, continue with the following tutorial:

Tutorial: Create a picture viewer

See also

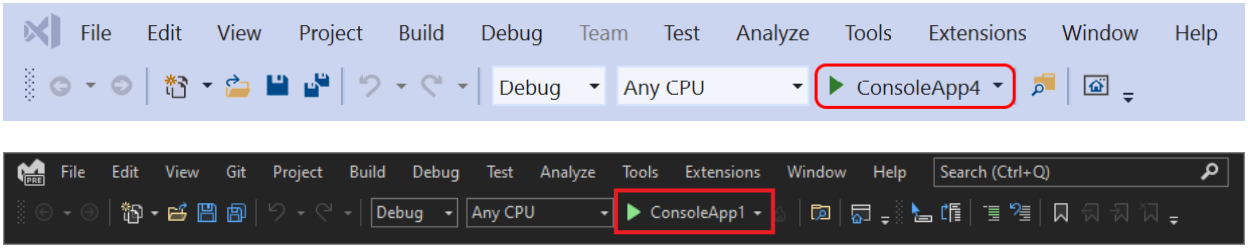
- More C# tutorials
- Visual Basic tutorials
- C++ tutorials

How to: Run a C# program in Visual Studio

10/28/2021 • 5 minutes to read • Edit Online

How to run a program depends on what you start from, the type of program, and whether you want to run under the debugger. In the simplest case, to build and run an open project in Visual Studio:

- Press **F5**, choose **Debug** > **Start with debugging** from the Visual Studio menu, or select the green **Start** arrow and project name on the Visual Studio toolbar.
- Or, to run without debugging, press **Ctrl+F5** or choose **Debug** > **Start without debugging** from the Visual Studio menu.

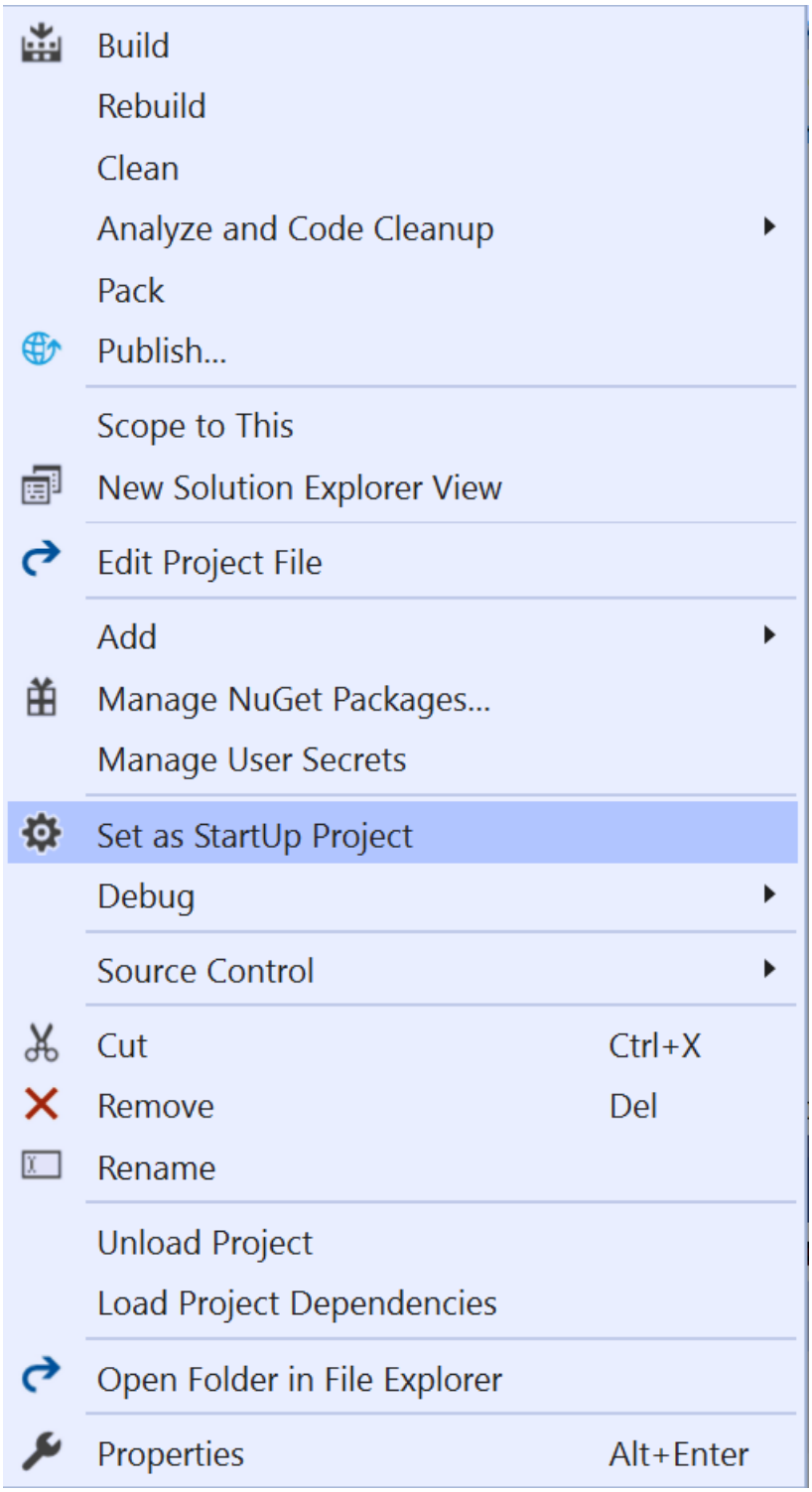


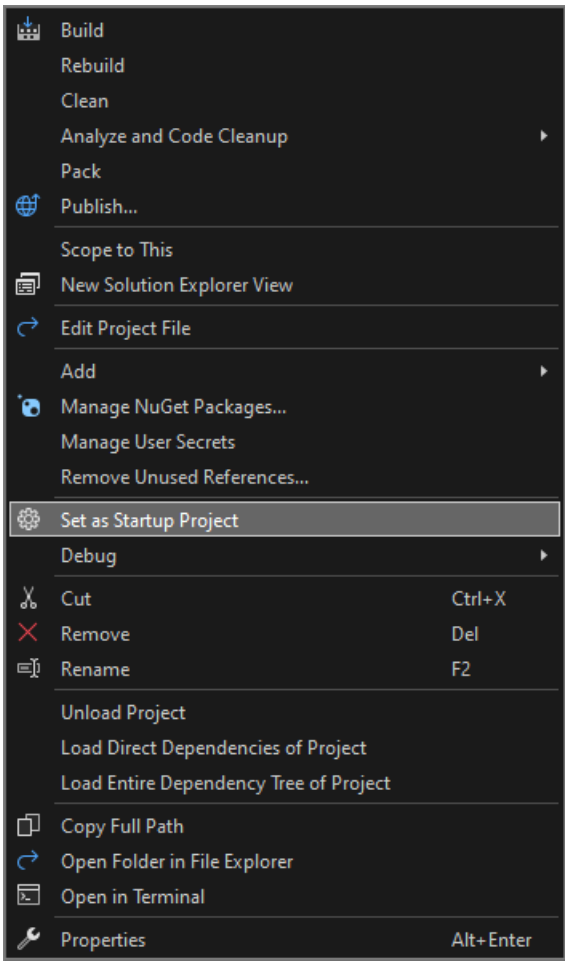
Start from a project

You can run a C# project or *.csproj* file if it's a runnable program. If the project contains a C# file with a `Main` method, and its output is an executable or *.exe* file, it will probably run if it builds successfully.

1. If your program code is already in a Visual Studio project, open the project. To do so, you can double-click or tap on the *.csproj* file in Windows File Explorer, or choose **Open a project** in Visual Studio, browse to find the *.csproj* file, and select the file.

2. After the project loads in Visual Studio, if your Visual Studio solution has more than one project, make sure to set the project with the `Main` method as the startup project. To set the startup project, right-click on the project name or node in **Solution Explorer** and choose **Set as Startup Project** from the context menu.


Build
Rebuild
Clean
Analyze and Code Cleanup
Pack
Publish...
Scope to This
New Solution Explorer View
Edit Project File
Add
Manage NuGet Packages...
Manage User Secrets
Set as StartUp Project
Debug
Source Control
Cut Ctrl+X
Remove Del
Rename
Unload Project
Load Project Dependencies
Open Folder in File Explorer
Properties Alt+Enter



3. To run the program, press **Ctrl+F5**, select **Debug** > **Start without debugging** from the top menu, or select the green **Start** button.

   Visual Studio tries to build and run your project. At the bottom of the Visual Studio screen, the build output appears in the **Output** window, and any build errors appear in the **Error List** window.

   If the build succeeds, the app runs as appropriate for the type of project. Console apps run in a terminal window, Windows desktop apps start in a new desktop window, and web apps run in a browser hosted by IIS Express.

Start from code

If you start from a code listing, code file, or small number of files, first make sure the code is a runnable program from a trusted source. Any app with a `Main` method is probably a runnable program. You can use the Console Application template to create a project to work with the app in Visual Studio.

Code listing for a single file

1. Start Visual Studio, and open an empty C# Console Application project.
2. Replace all the code in the project *.cs* file with the contents of your code listing or file.
3. Rename the project *.cs* file to match your code file name.

Several code listings or files on disk

1. Start Visual Studio, and create a new project of the appropriate type. Use the C# **Console Application** if you're not sure.
2. In the new project, replace all the code in the project code file with the contents of your first code listing or file.
3. Rename the project code file to match your code file name.
4. For each remaining code file:

a. Right-click the project node in **Solution Explorer** and choose **Add** > **Existing Item**, or select the project and press **Shift+Alt+A**.
b. Browse to and select the code file to import it into the project.

Several files in a folder

If you have a folder with many files, first check for a project or solution file. Programs that Visual Studio creates have project and solution files. In Windows File Explorer, look for files with the *.csproj* or *.sln* extension. Double-click the *.csproj* file to open it in Visual Studio. See Start from a Visual Studio solution or project.

If the code is from another development environment, there's no project file. Open the folder by choosing **Open** > **Folder** in Visual Studio. See Develop code without projects or solutions.

Start from a GitHub or Azure DevOps repo

If the code you want to run is in a GitHub or Azure DevOps repo, you can use Visual Studio to open the project directly from the repo. See Open a project from a repo.

Run the program

To start building the program, press the green **Start** button on the Visual Studio toolbar, or press **F5** or **Ctrl+F5**. Using the **Start** button or **F5** runs the program under the debugger.

Visual Studio attempts to build and run the code in your project. If a build doesn't succeed, see the following sections for some ideas on how to get the project to build successfully.

Troubleshooting

Your code might have errors. Or the code might be correct, but depends on missing assemblies or NuGet packages, or targets a different version of .NET. In those cases, you might be able to easily fix the build.

Add references

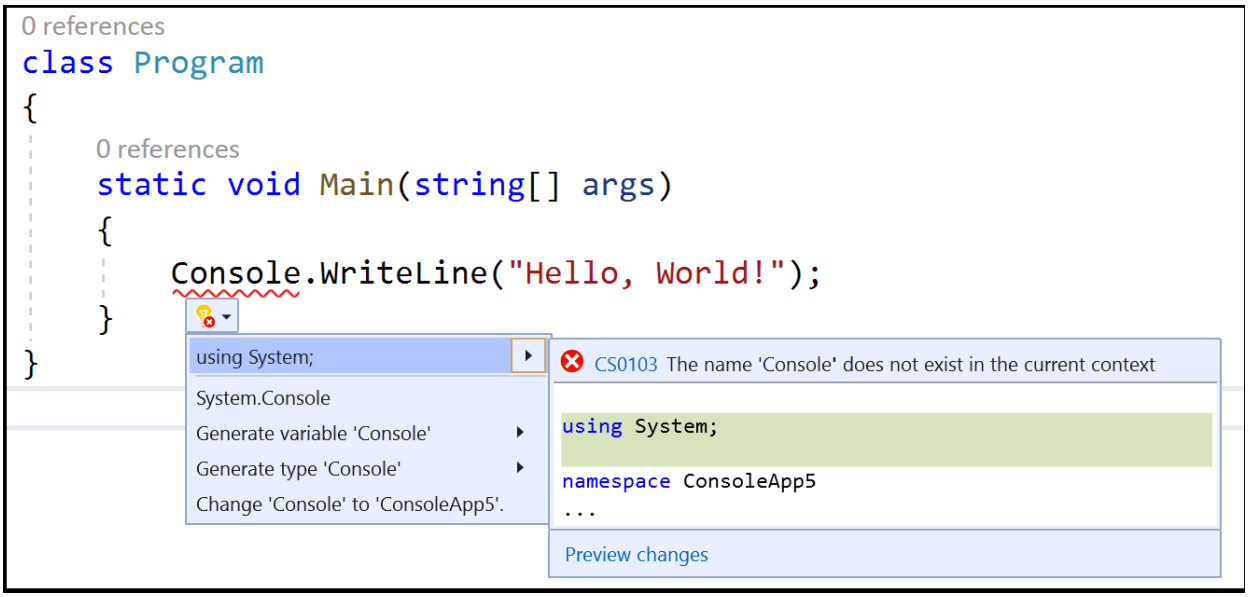
To build properly, the code must be correct and have the right references to libraries or other dependencies. Red squiggly underlines in code or entries in the **Error List** show errors even before you compile and run the program. If the errors relate to unresolved names, you probably need to add a reference or a `using` directive, or both. If the code references any missing assemblies or NuGet packages, you need to add those references to the project.

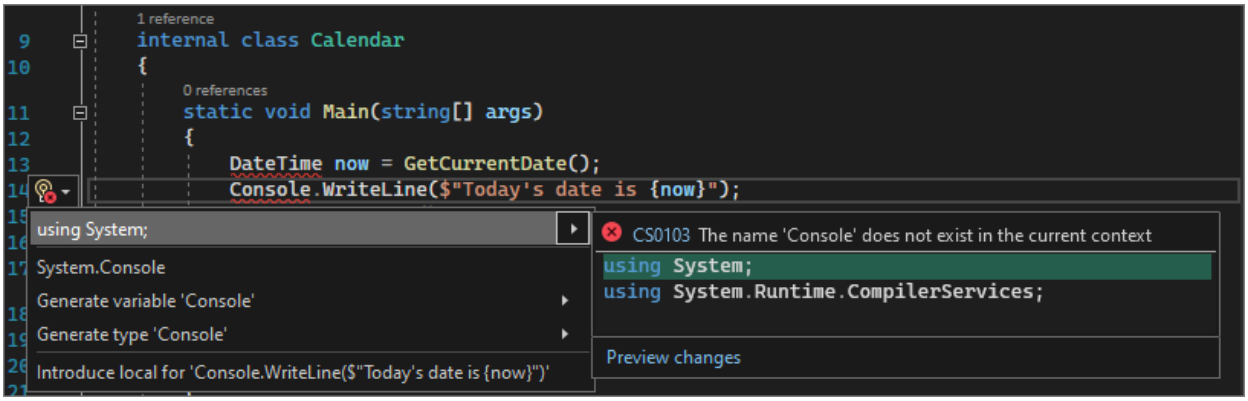
Visual Studio tries to help you identify missing references. When a name is unresolved, a light bulb icon appears in the editor. Select the light bulb to see suggestions on how to fix the issue. Fixes might be to:

- Add a using directive.
- Add a reference to an assembly.
- Install a NuGet package.

Add a using directive

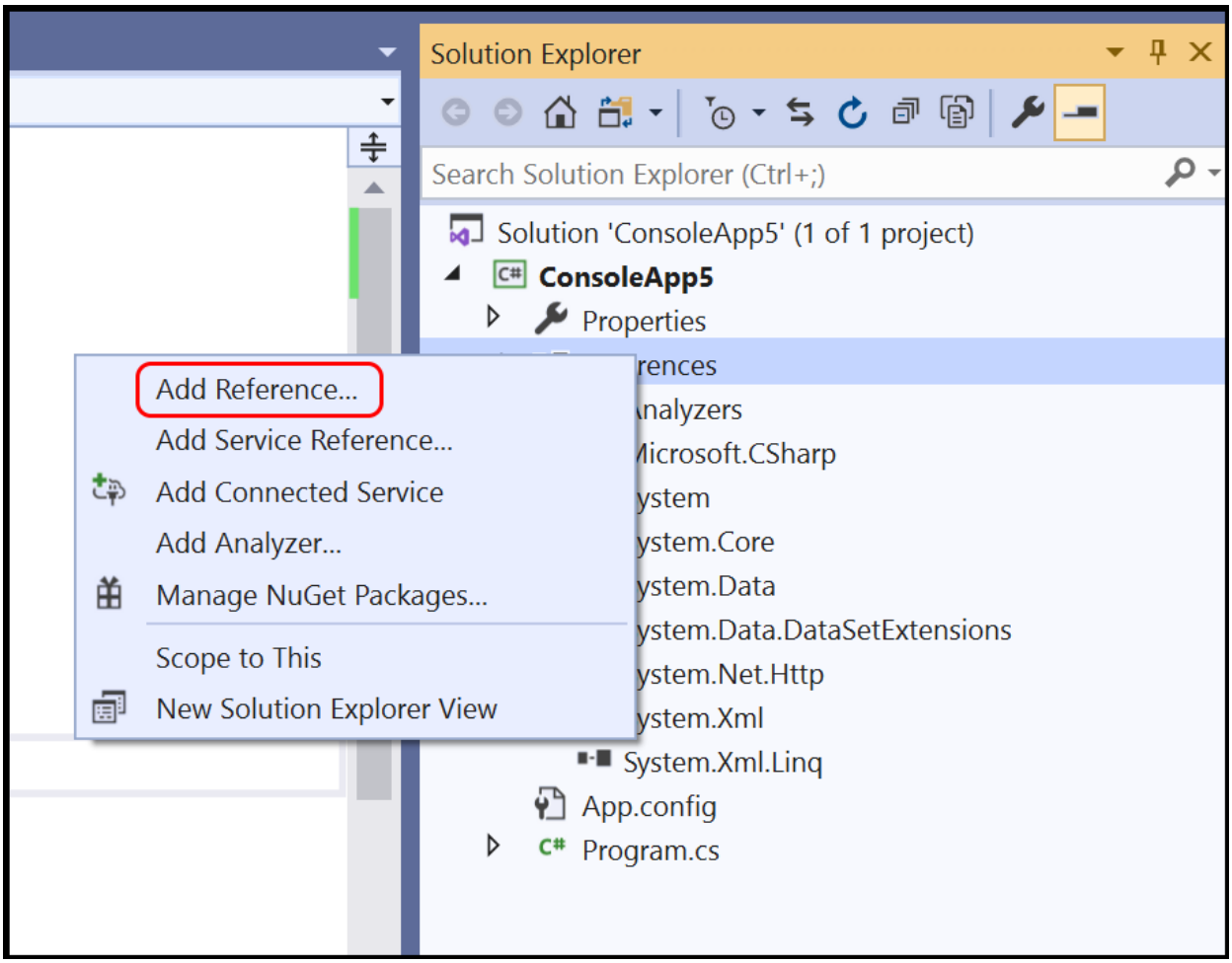
Here's an example of a missing `using` directive. You can add `using System;` to the start of the code file to resolve the unresolved name `Console`:

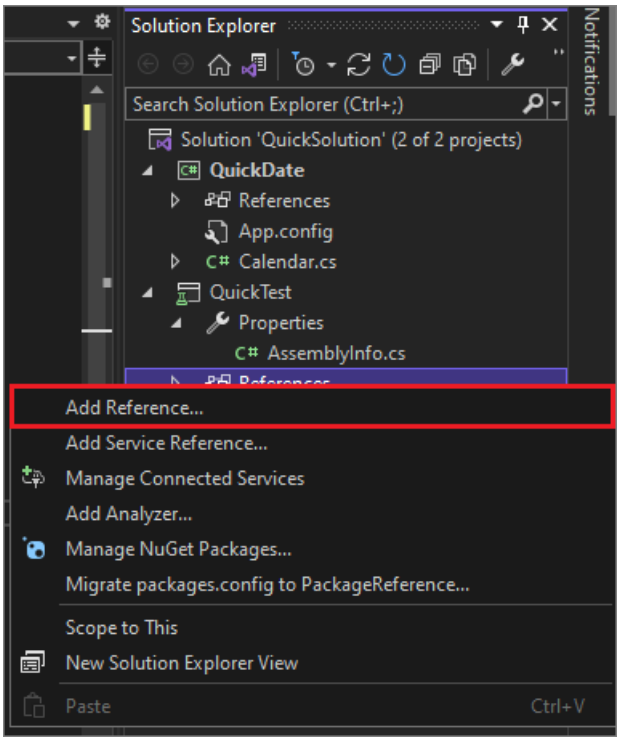




Add an assembly reference

.NET references can be assemblies or NuGet packages. In source code, the publisher or author usually explains what assemblies the code requires and what packages it depends on. To add a reference to a project manually, right-click the **References** node in **Solution Explorer** and choose **Add Reference**. In the **Reference Manager**, locate and add the required assembly.

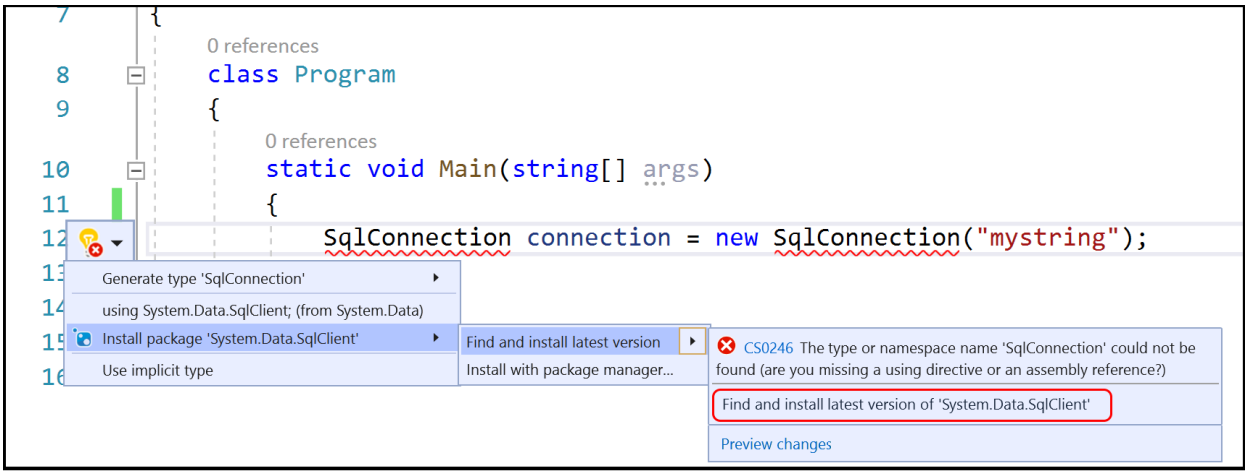


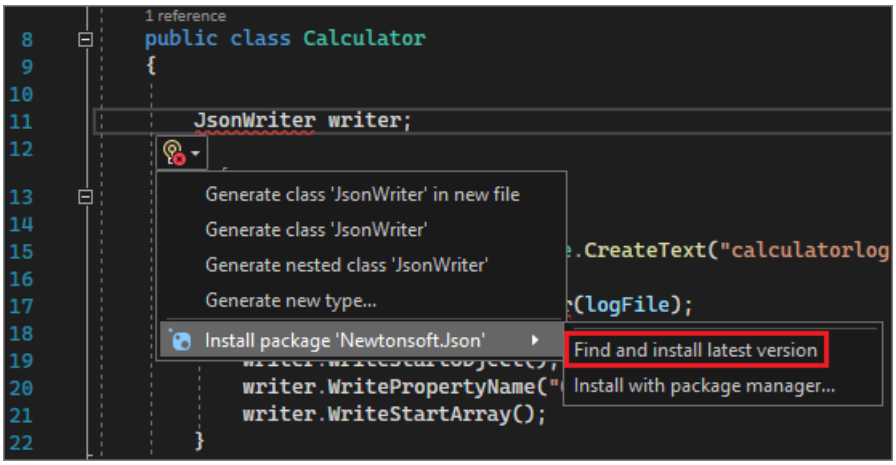


You can find assemblies and add references by following the instructions in Add or remove references by using the Reference Manager.

Add a NuGet package

If Visual Studio detects a missing NuGet package, a light bulb appears and gives you the option to install the package:





If that doesn't solve the issue or Visual Studio can't locate the package, try searching for the package online. See Install and use a NuGet package in Visual Studio.

Use the right version of .NET

Because different versions of the .NET Framework have some backward compatibility, a newer framework might run code written for an older framework without any changes. But sometimes you need to target a specific .NET framework version. You might need to install a specific version of the .NET Framework or .NET Core. See Modify Visual Studio.

To change the target .NET framework version, see Change the target framework. For more information, see Troubleshooting .NET Framework targeting errors.

Next steps

- Explore the Visual Studio development environment by reading Welcome to the Visual Studio IDE.
- Create your first C# app.

Tutorial: Open a project from a repo

10/28/2021 • 9 minutes to read • Edit Online

F In this tutorial, you'll use Visual Studio to connect to a repository for the first time and then open a project from it.

If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

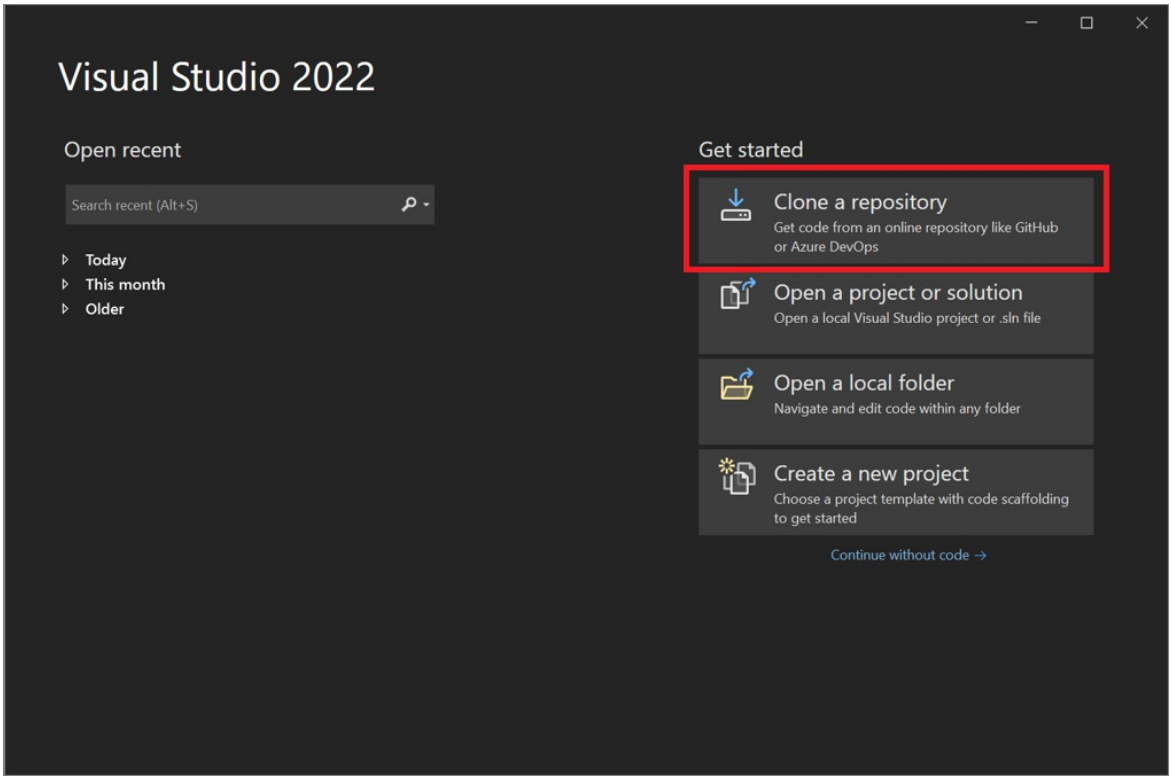
Open a project from a GitHub repo

Visual Studio makes it easy to open a project from a repo. You can do so when you start Visual Studio, or you can do so directly from within the Visual Studio IDE.

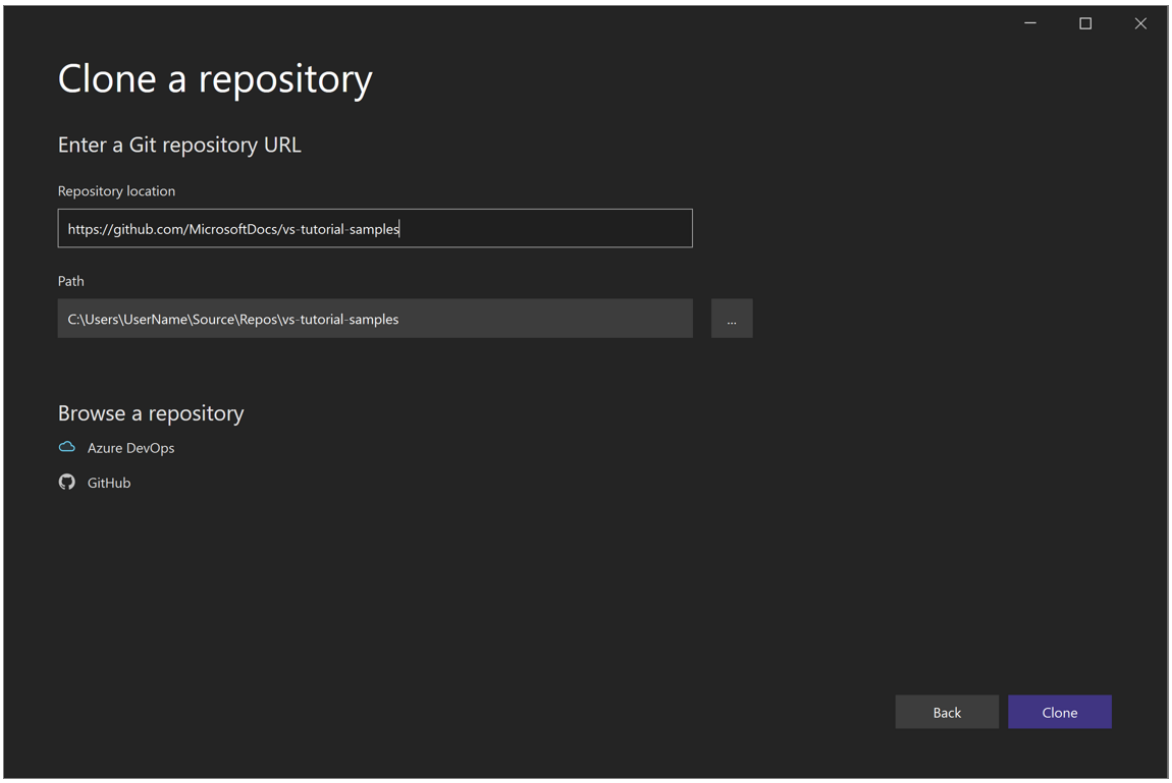
Here's how.

Use the start window

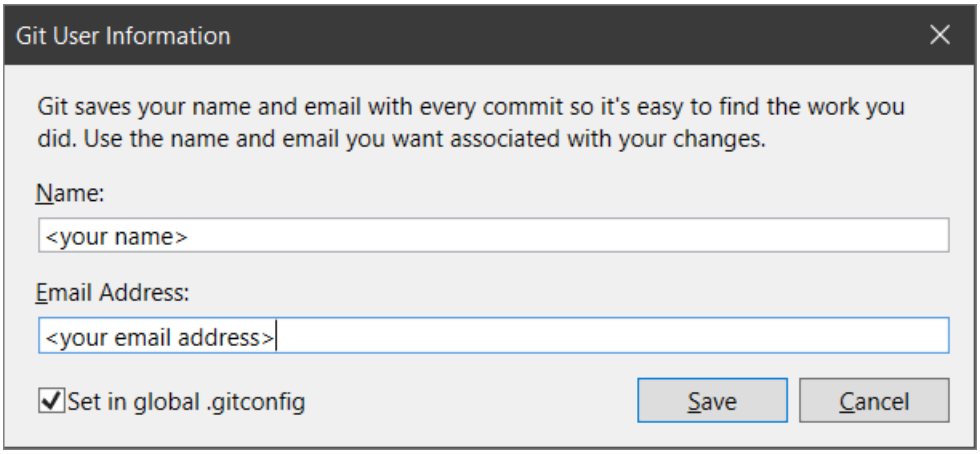
1. Open Visual Studio.
2. On the start window, select **Clone a repository**.



3. Enter or type the repository location, and then select the **Clone** button.



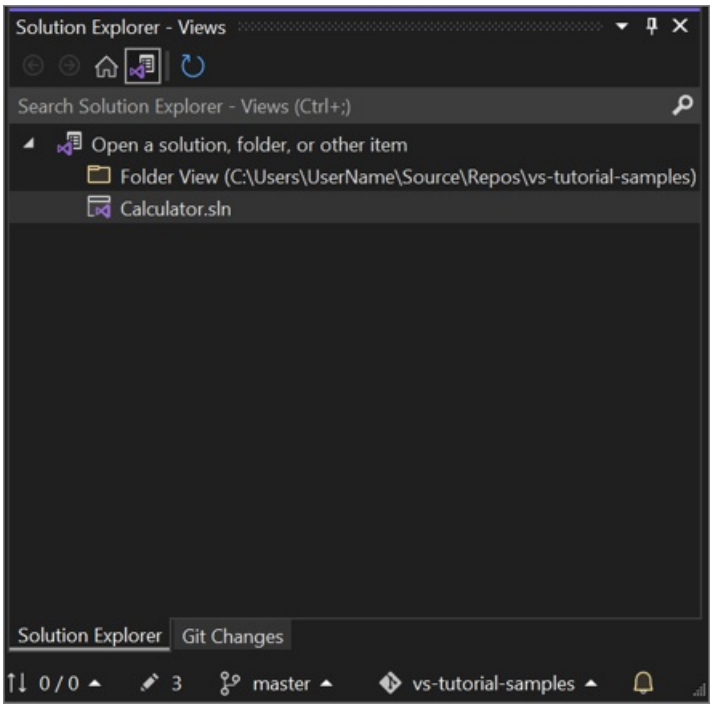
4. You might be asked for your user sign-in information in the **Git User Information** dialog box. You can either add your information or edit the default information it provides.



   Select **Save** to add the info to your global .gitconfig file. (Or, you can choose to do this later by selecting **Cancel**.)

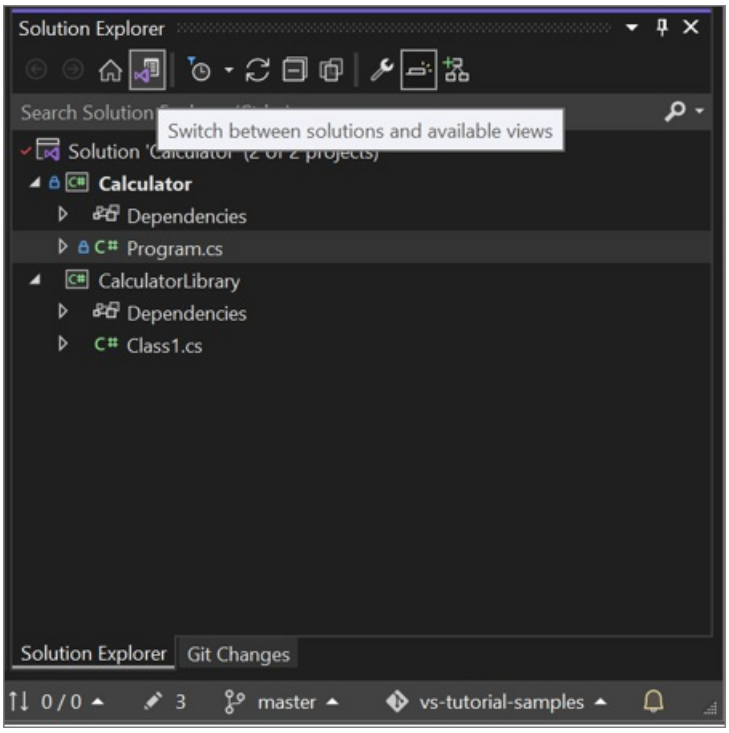
> **TIP**
>
> For more information about signing in to Visual Studio, see the **Sign in to Visual Studio** page. For specific information about how to use your GitHub account to sign in, see the **Work with GitHub accounts in Visual Studio** page. And if you receive a trust notification and want to know more about it, see the Configure trust settings for files and folders page.

5. Next, Visual Studio loads the solution(s) from the repository by using the **Folder View** in Solution Explorer.



You can view a solution in **Solution View** by double-clicking its .sln file.

Or, you can select the **Switch Views** button, and then select **Program.cs** to view a solution's code.

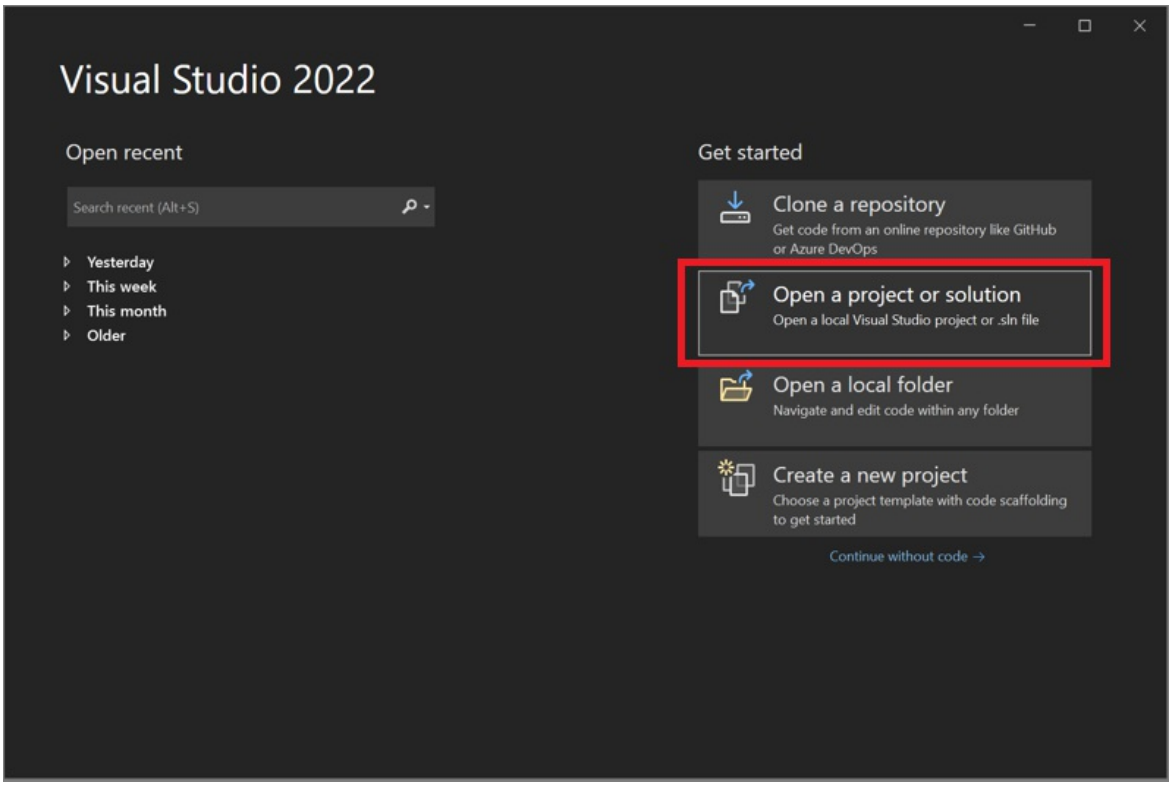


> **TIP**
> The default view is set to Folder View. You can change it to Solution View from the **Git** menu. Select **Settings** > **Source Control** > **Git Global Settings** > **Automatically load the solution when opening a Git repository** to do so.

Open a project locally from a previously cloned GitHub repo

1. Open Visual Studio.
2. On the start window, select **Open a project or solution**.

   Visual Studio opens an instance of File Explorer, where you can browse to your solution or project, and then select it to open it.



> **TIP**
>
> If you've opened the project or solution recently, select it from the **Open recent** section to quickly open it again.

Start coding!

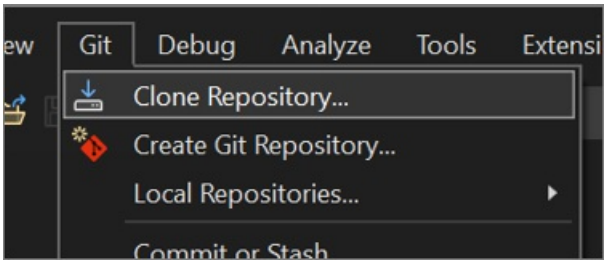
Use the IDE

You can also use the **Git** menu or the **Select Repository** control in the Visual Studio IDE to interact with a repository's folders and files.

Here's how.

To clone a repo and open a project

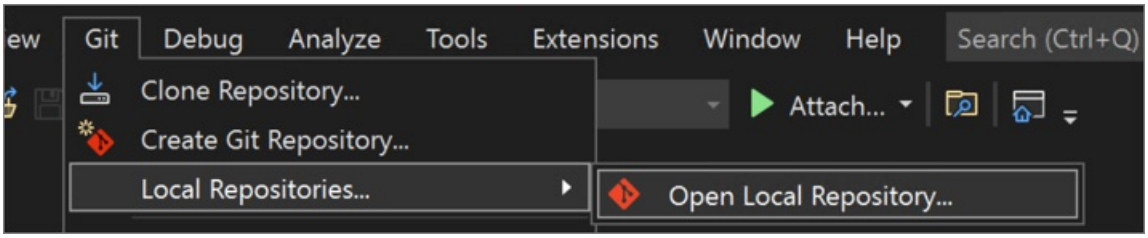
1. In the Visual Studio IDE, select the **Git** menu, and then select **Clone Repository**.



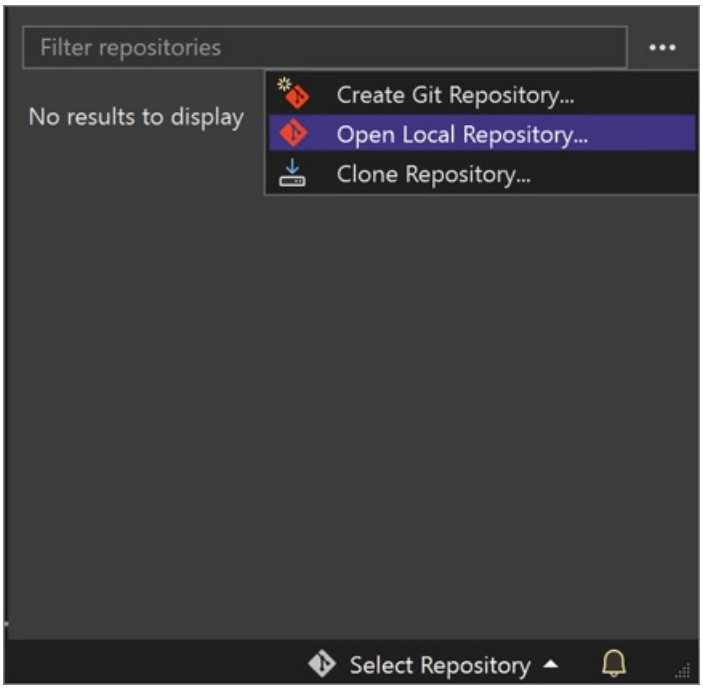
2. Follow the prompts to connect to the Git repository that includes the files you're looking for.

To open local folders and files

1. In the Visual Studio IDE, select the **Git** menu, select **Local Repositories**, and then select **Open Local Repository**.



Alternatively, you can perform the same task from **Solution Explorer**. To do so, choose the **Select Repository** control, select the **ellipsis** icon that's next to the **Filter repositories** box, and then select **Open Local Repository**.

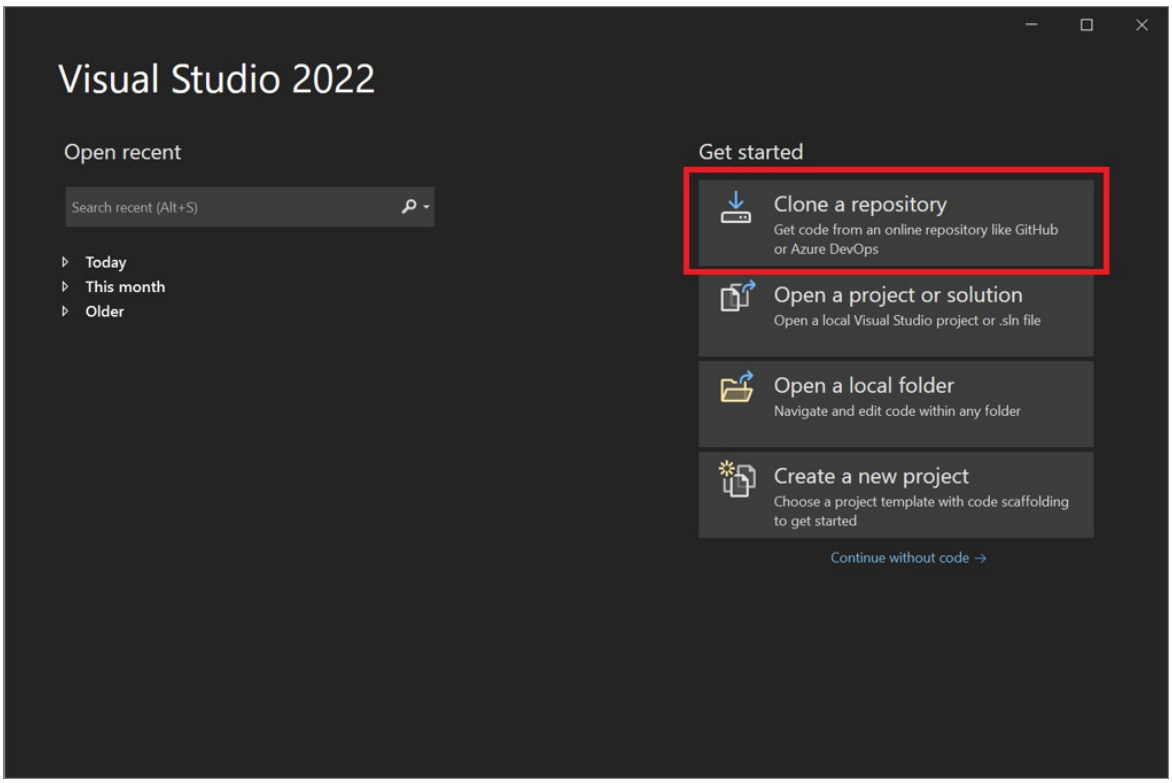


2. Follow the prompts to connect to the Git repository that has the files you're looking for.

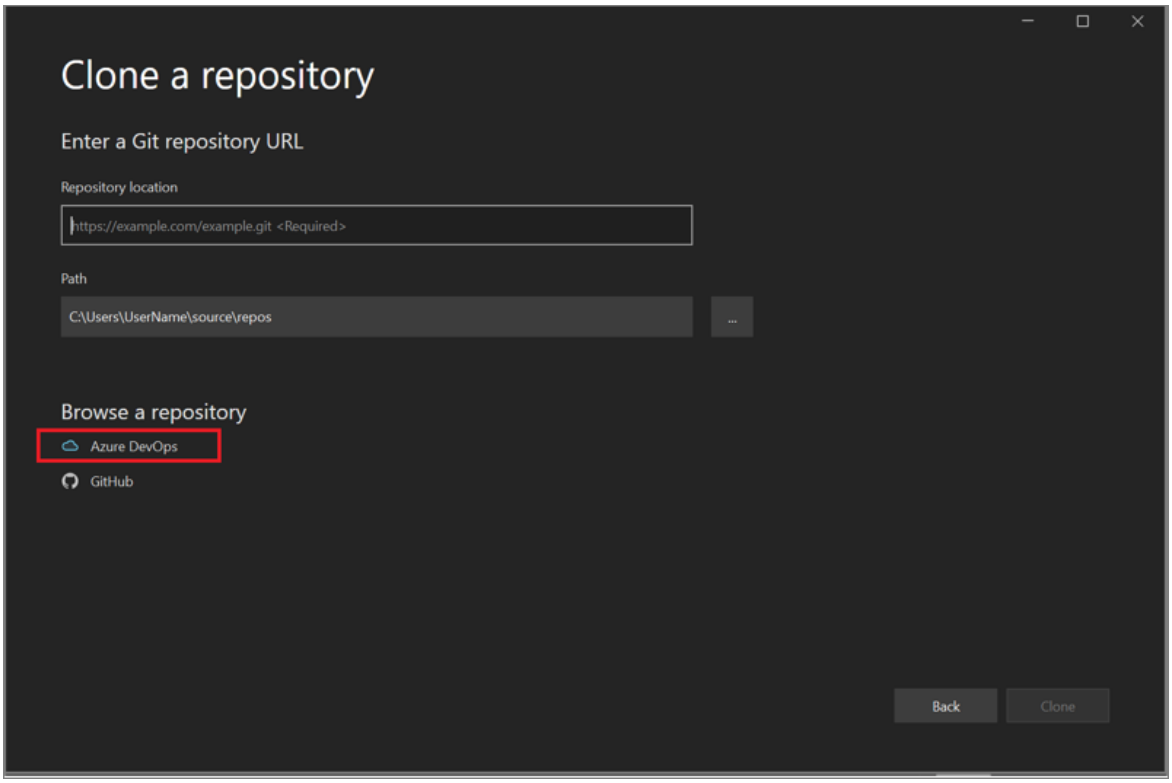
Browse to an Azure DevOps Server

Here's how to browse to an Azure DevOps Server by using Visual Studio.

1. Open Visual Studio.
2. On the start window, select **Clone a repository**.



3. In the **Browse a repository** section, select **Azure DevOps**.



4. Follow the prompts to connect to an Azure DevOps Server that hosts the files you're looking for.

Open a project from a GitHub repo with Visual Studio 2019

How you open a project from a GitHub repo by using Visual Studio depends on which version you have. Specifically, if you've installed version Visual Studio 2019 **version 16.8** or later, there's a new, more fully integrated Git experience in Visual Studio available to you.

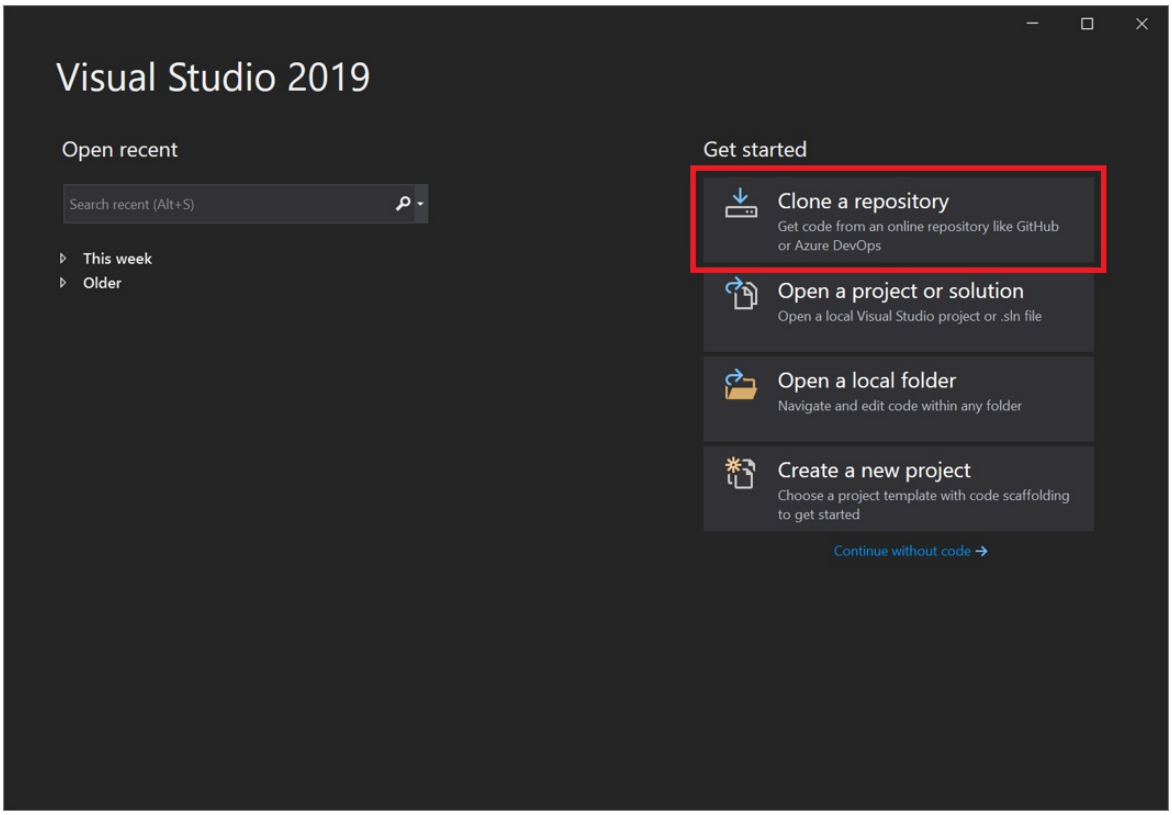
But no matter which version you've installed, you can always open a project from a GitHub repo with Visual Studio.

16.8 and later

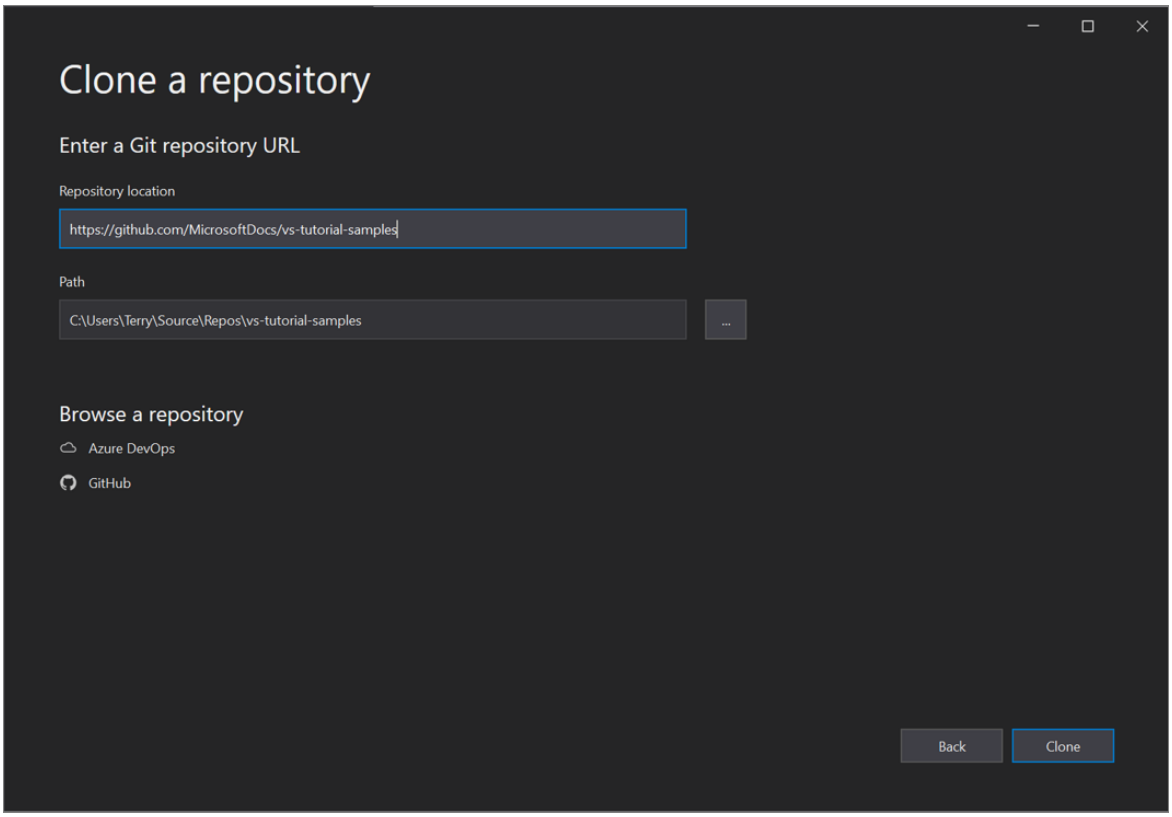
Here's how to use Git in Visual Studio 2019 **version 16.8** or later.

Clone a GitHub repo and then open a project

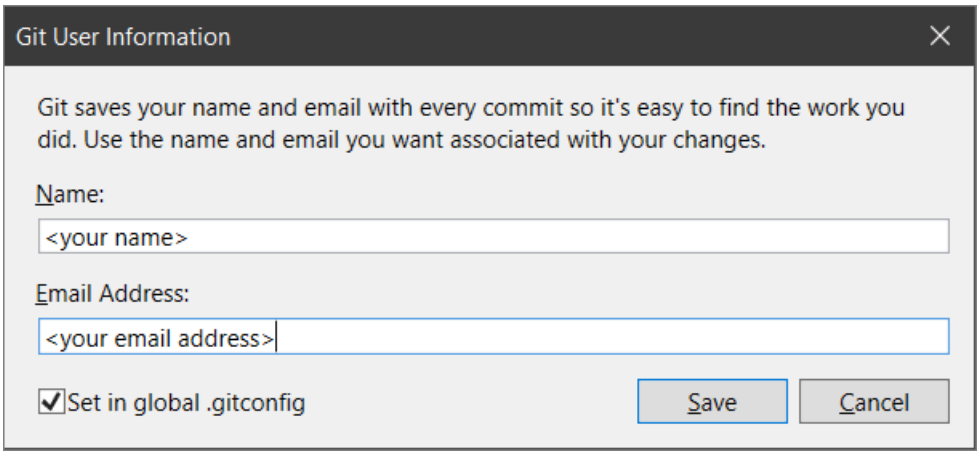
1. Open Visual Studio 2019.
2. On the start window, select **Clone a repository**.



3. Enter or type the repository location, and then select **Clone**.



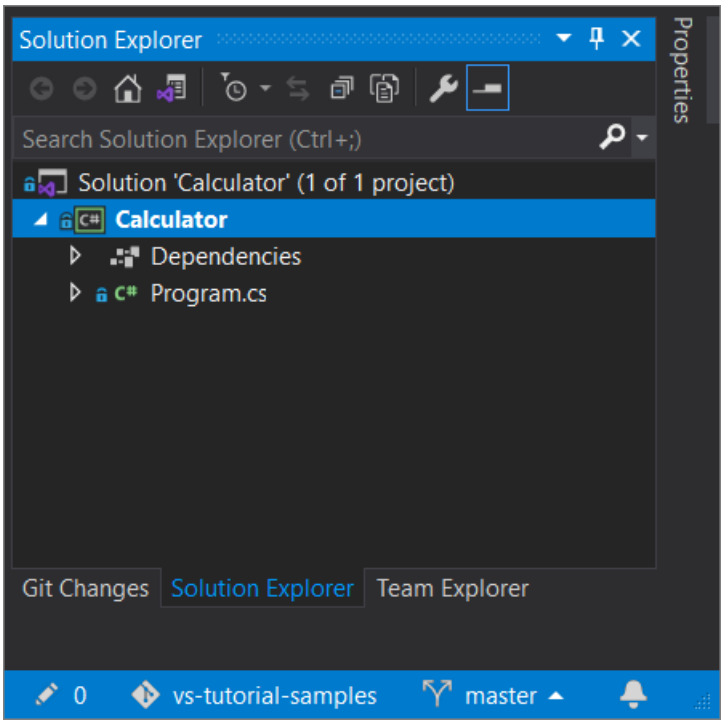
4. You might be asked for your user sign-in information in the **Git User Information** dialog box. You can either add your information or edit the default information it provides.



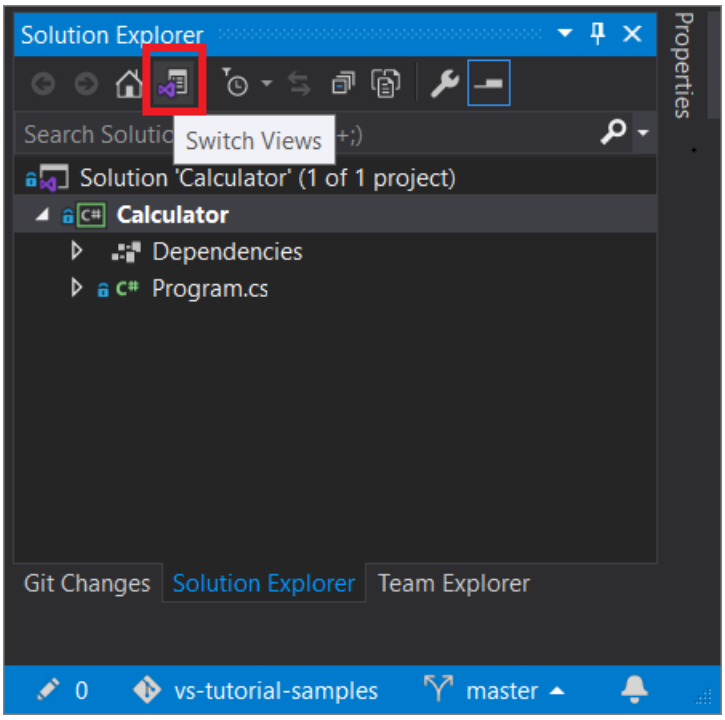
Select **Save** to add the info to your global .gitconfig file. (Or, you can choose to do this later by selecting **Cancel**.)

> **TIP**
>
> For more information about signing in to Visual Studio, see the Sign in to Visual Studio page. And for specific information about how to use your GitHub account to sign in, see the Work with GitHub accounts in Visual Studio page.

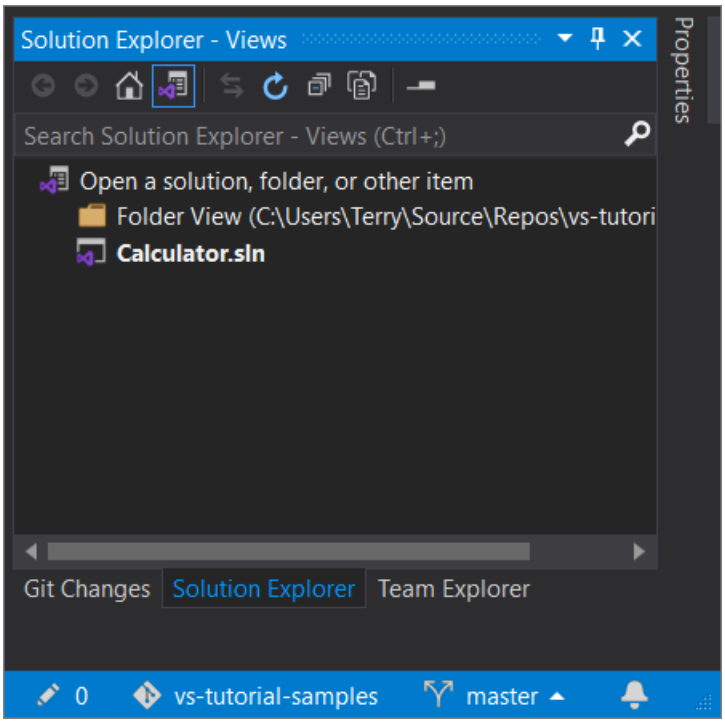
Next, Visual Studio automatically loads and opens the solution from the repository.



5. If your repository contains multiple solutions, you will see them in Solution Explorer. You can view the list of solutions by selecting the **Switch Views** button in Solution Explorer.



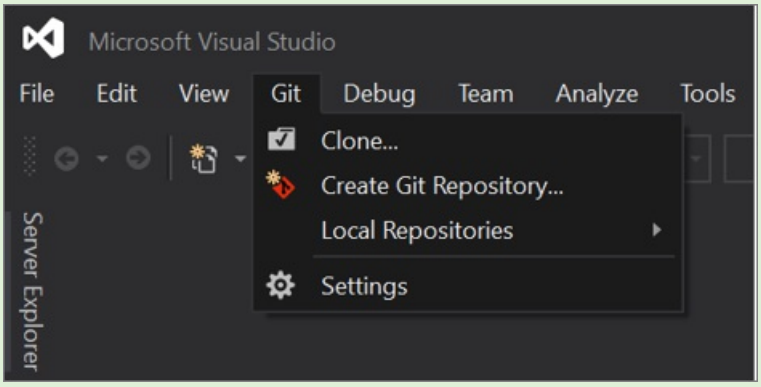
Solution Explorer then gives you the option to open the root folder in **Folder View** or to select a solution file to open.



To toggle the view, select the **Switch Views** button again.

TIP

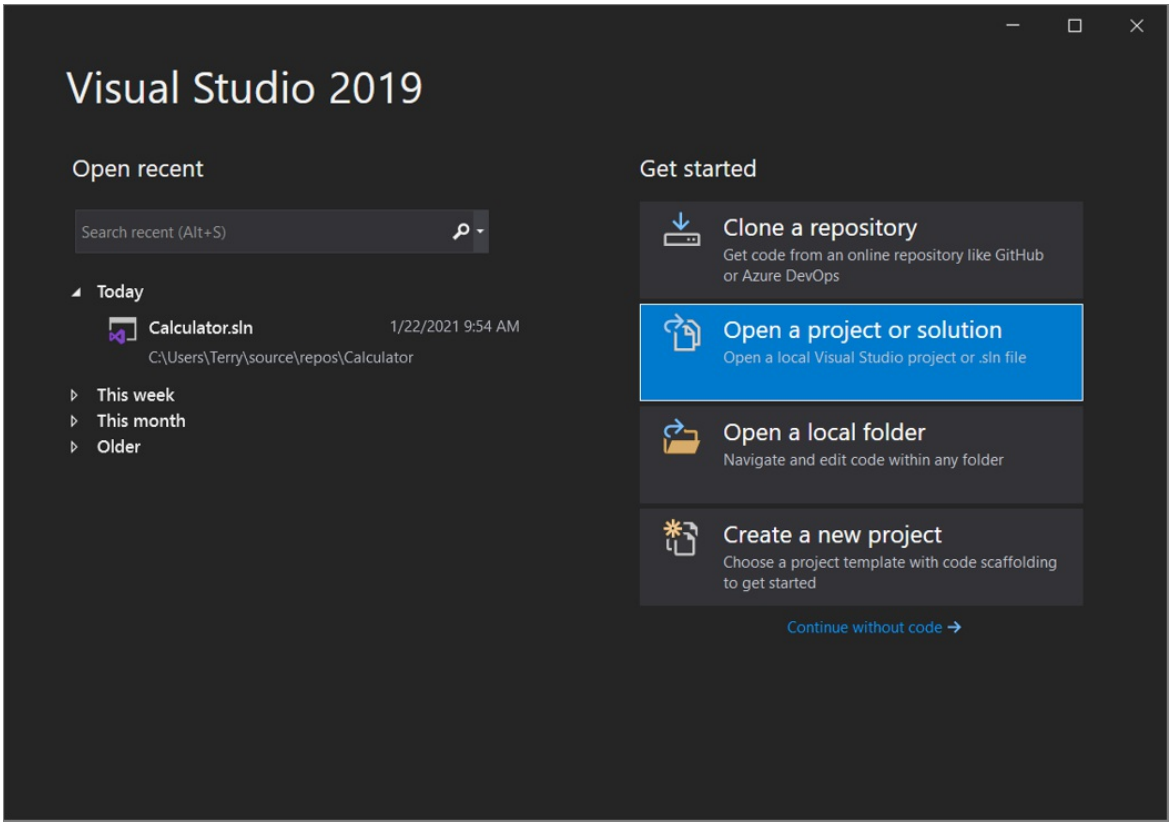
You can also use the **Git** menu in the Visual Studio IDE to clone a repo and open a project.



Open a project locally from a previously cloned GitHub repo

1. Open Visual Studio 2019 version 16.8 or later.
2. On the start window, select **Open a project or solution**.

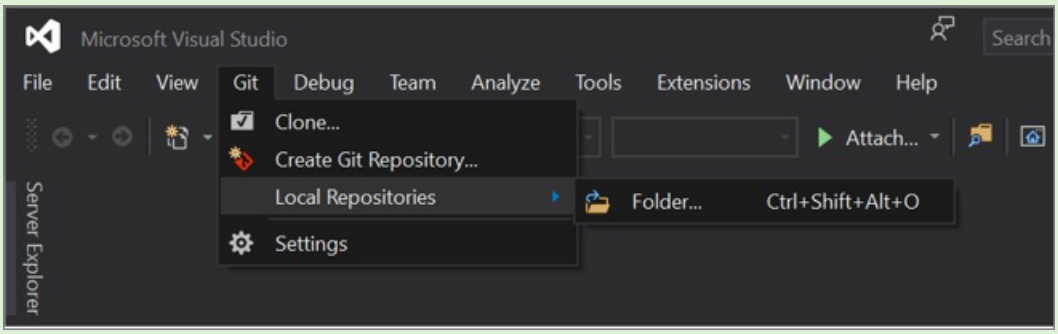
   Visual Studio opens an instance of File Explorer, where you can browse to your solution or project, and then select it to open it.



   If you've opened the project or solution recently, select it from the **Open recent** section to quickly open it again.

TIP

You can also use the **Git** menu in the Visual Studio IDE to open local folders and files from a repo that you've previously cloned.



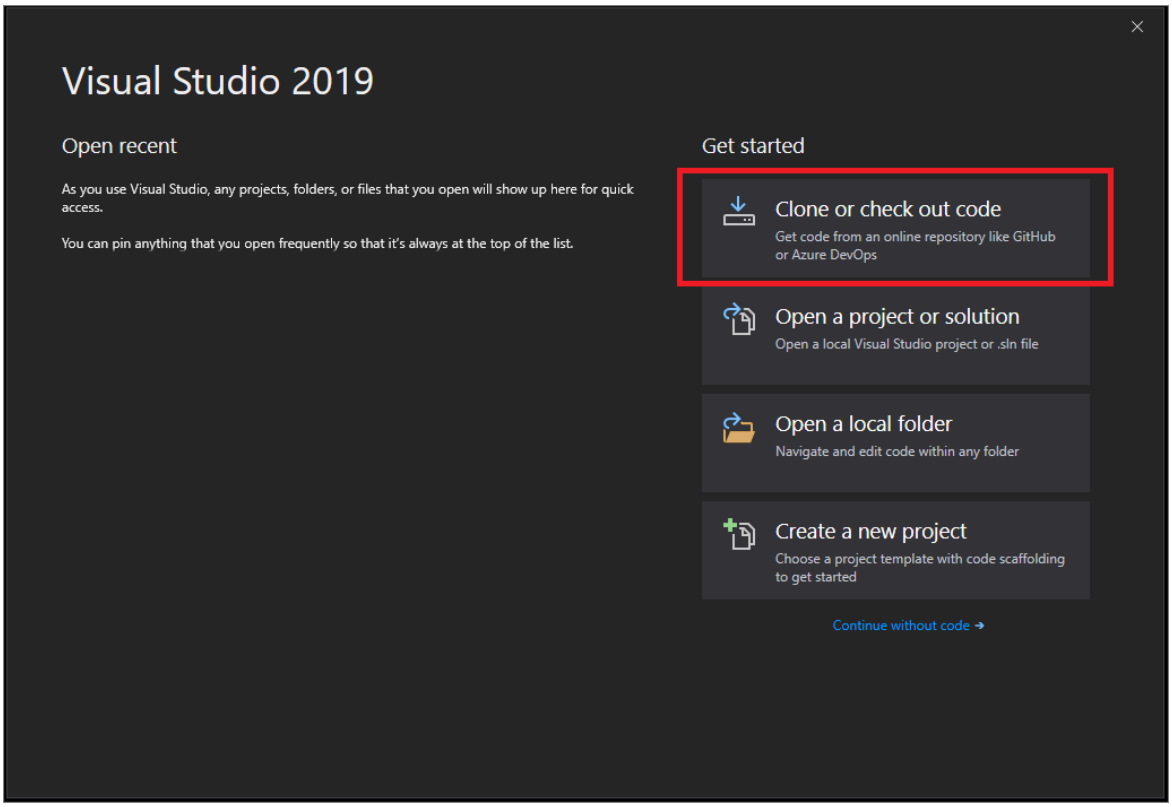
Start coding!

16.7 and earlier

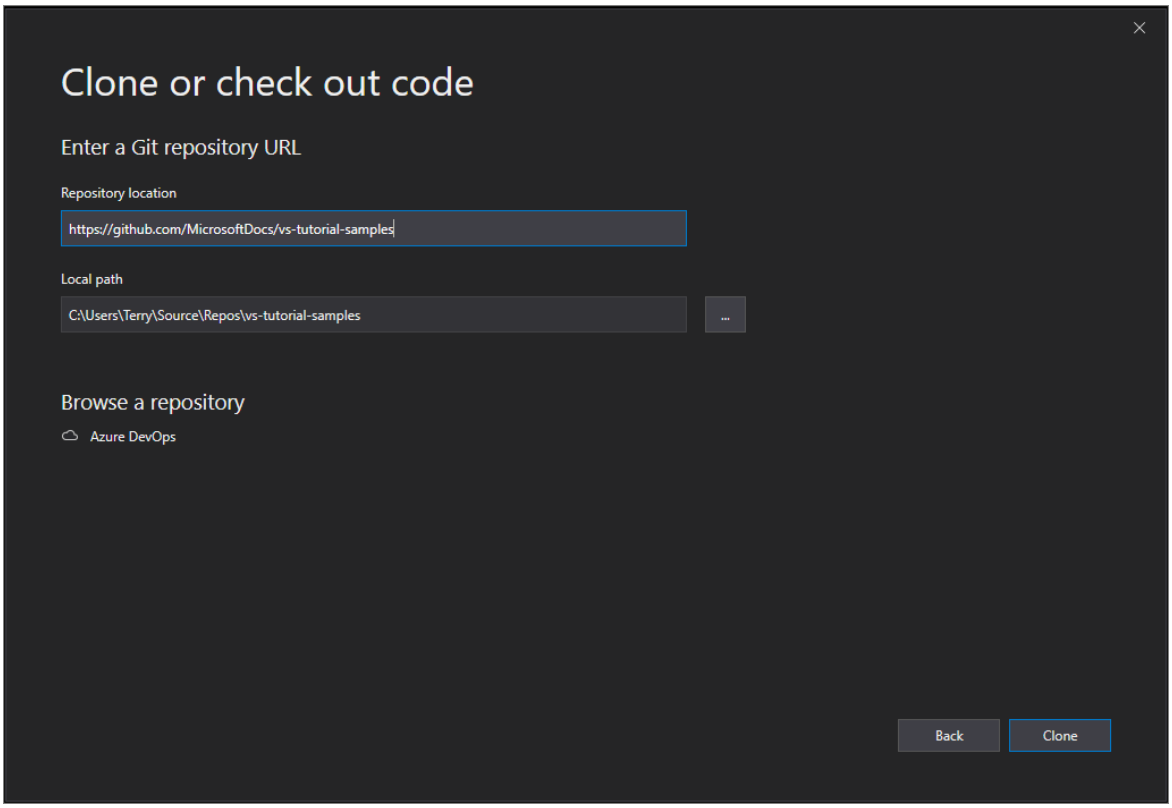
Here's how to use Git in Visual Studio 2019 **version 16.7** or earlier.

Clone a GitHub repo and then open a project

1. Open Visual Studio 2019 version 16.7 or earlier.
2. On the start window, select **Clone or check out code**.

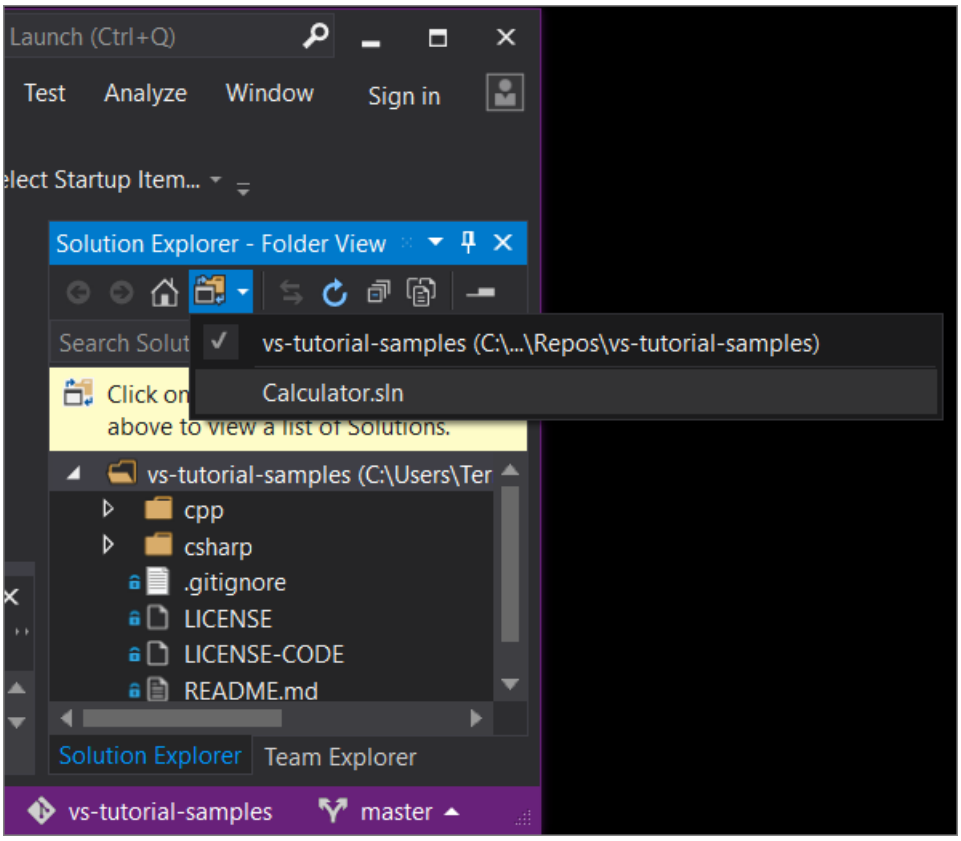


3. Enter or type the repository location, and then select **Clone**.



Visual Studio opens the project from the repo.

4. If you have a solution file available, it will appear in the "Solutions and Folders" fly-out menu. Select it, and Visual Studio opens your solution.



If you do not have a solution file (specifically, an .sln file) in your repo, the fly-out menu says "No Solutions Found." However, you can double-click any file from the folder menu to open it in the Visual Studio code editor.

Start coding!

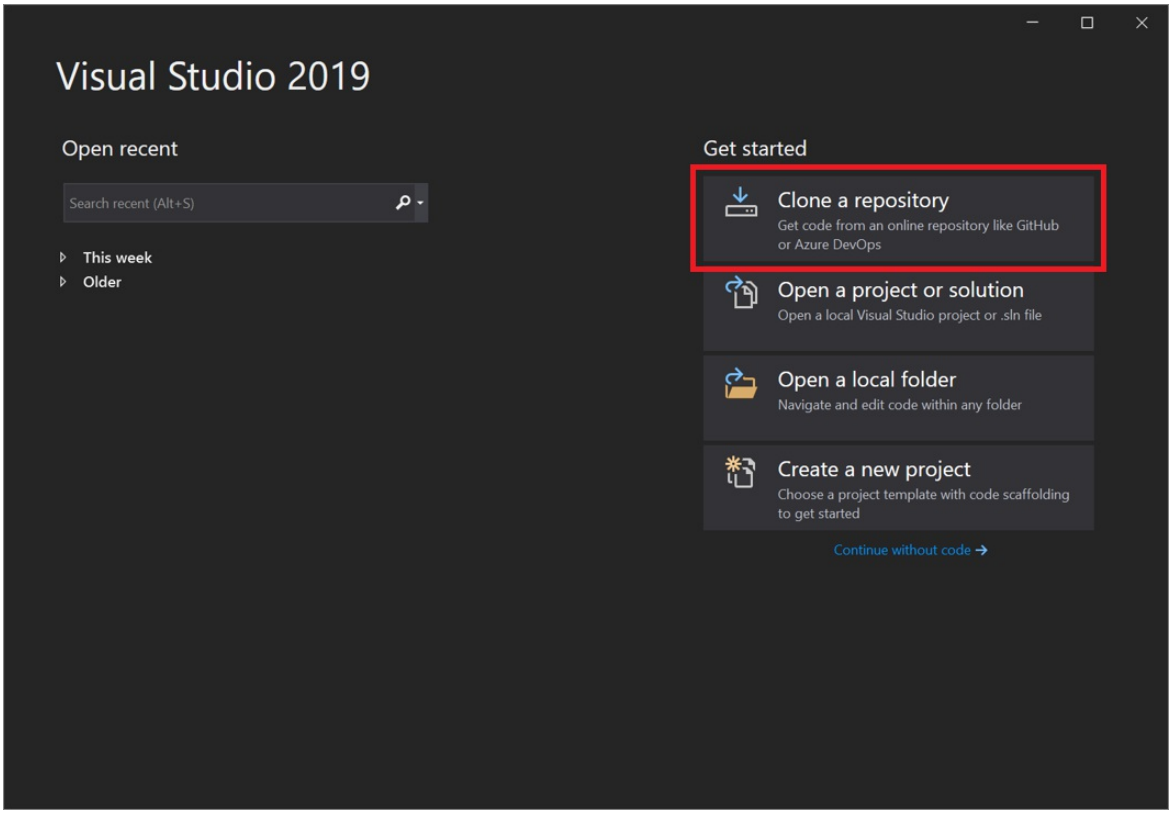
Connect to an Azure DevOps Server with Visual Studio 2019

What you see when you connect to an Azure DevOps Server by using Visual Studio 2019 depends on which version you have. Specifically, if you've installed version **version 16.8** or later, we've changed the UI to accommodate a new, more fully integrated Git experience in Visual Studio in Visual Studio.

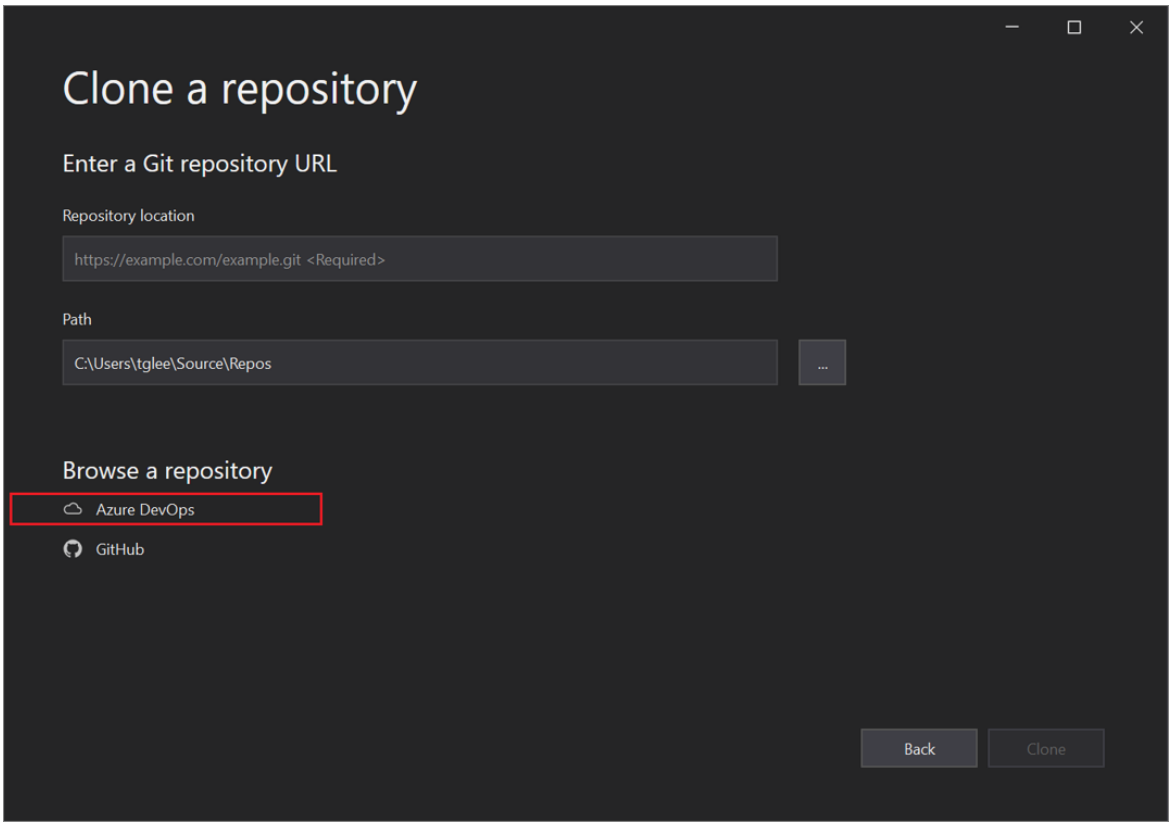
But no matter which version you have installed, you can always connect to an Azure DevOps Server with Visual Studio.

16.8 and later

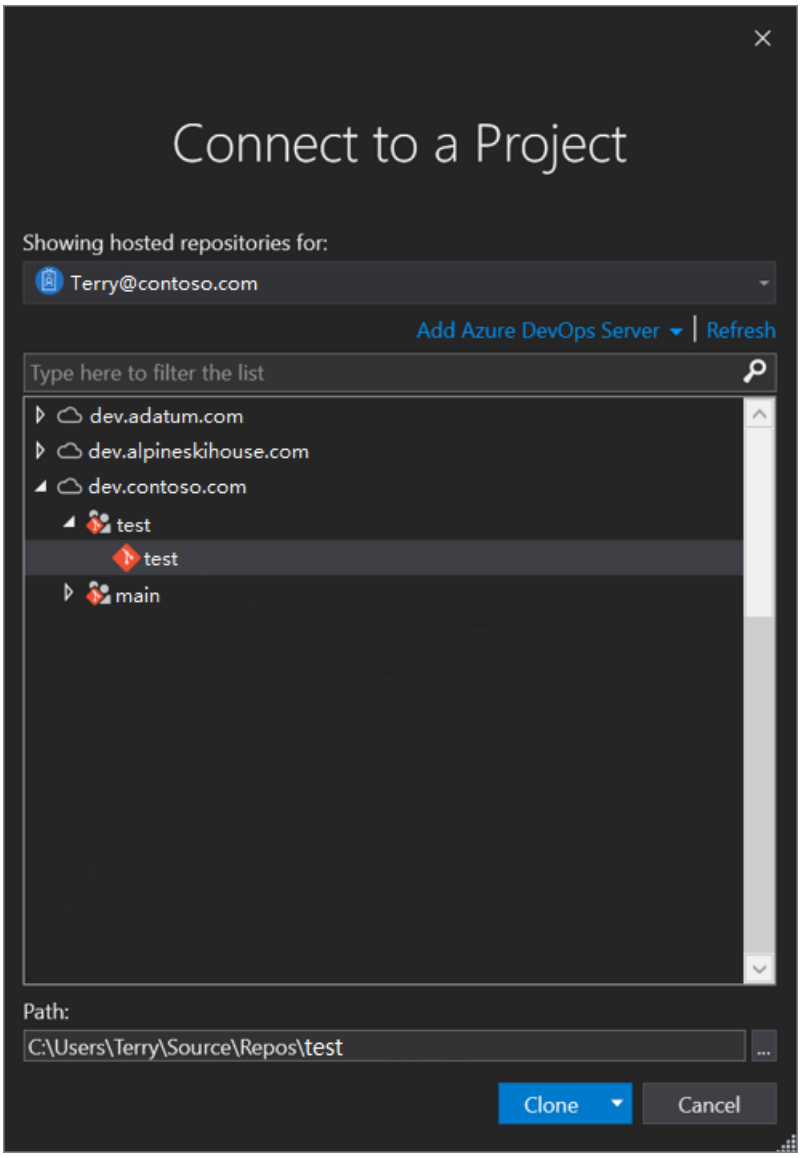
1. Open Visual Studio 2019 **version 16.8** or later.
2. On the start window, select **Clone a repository**.



3. In the **Browse a repository** section, select **Azure DevOps**.



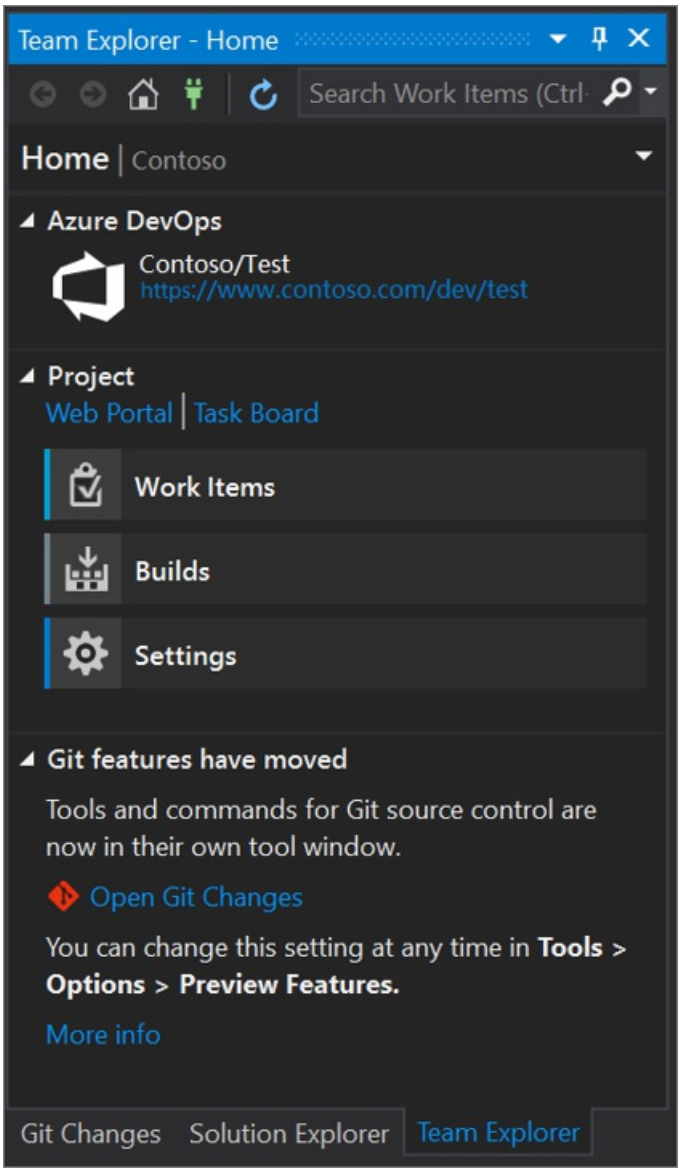
4. If you see a sign-in window, sign in to your account.
5. In the **Connect to a Project** dialog box, choose the repo that you want to connect to, and then select **Clone**.



> **TIP**
>
> If you don't see a pre-populated list of repos to connect to, select **Add Azure DevOps Server** to enter a server URL. (Alternatively, you might see a "No servers found" prompt that includes links to add an existing Azure DevOps Server or to create an Azure DevOps account.)

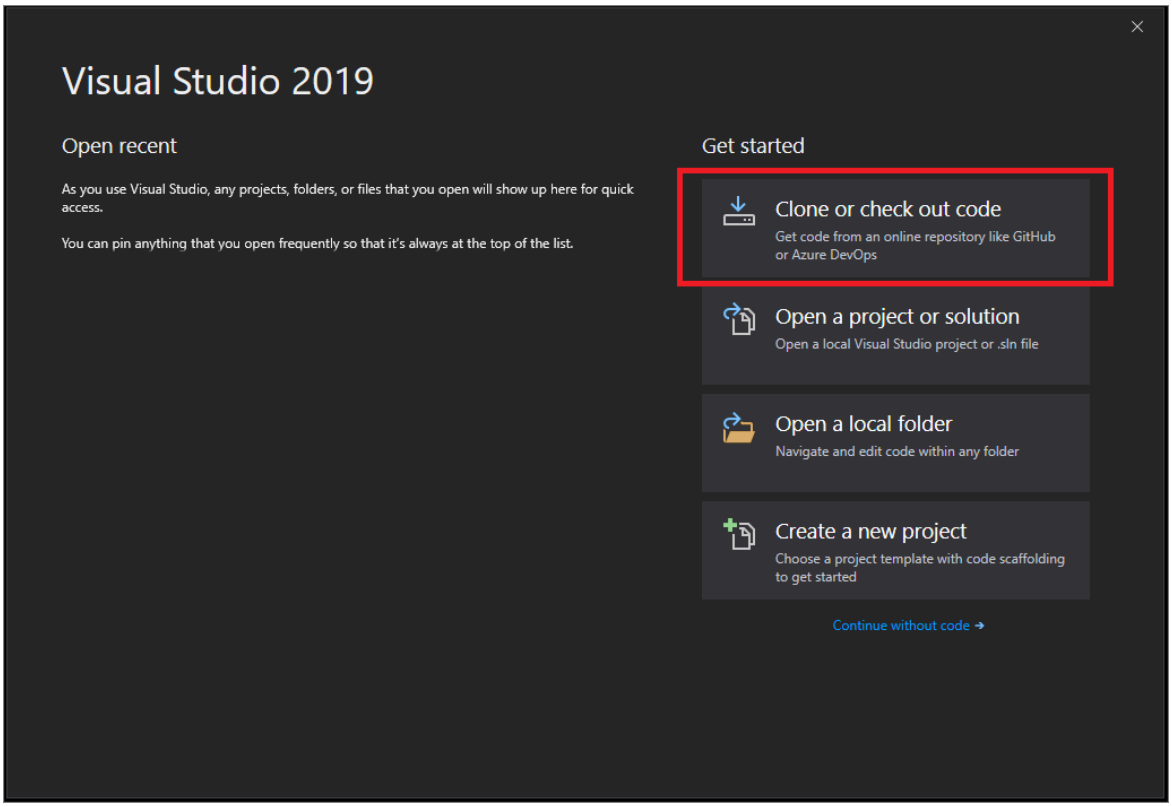
Next, Visual Studio opens **Solution Explorer** that shows the folders and files.

6. Select the **Team Explorer** tab to view the Azure DevOps actions.

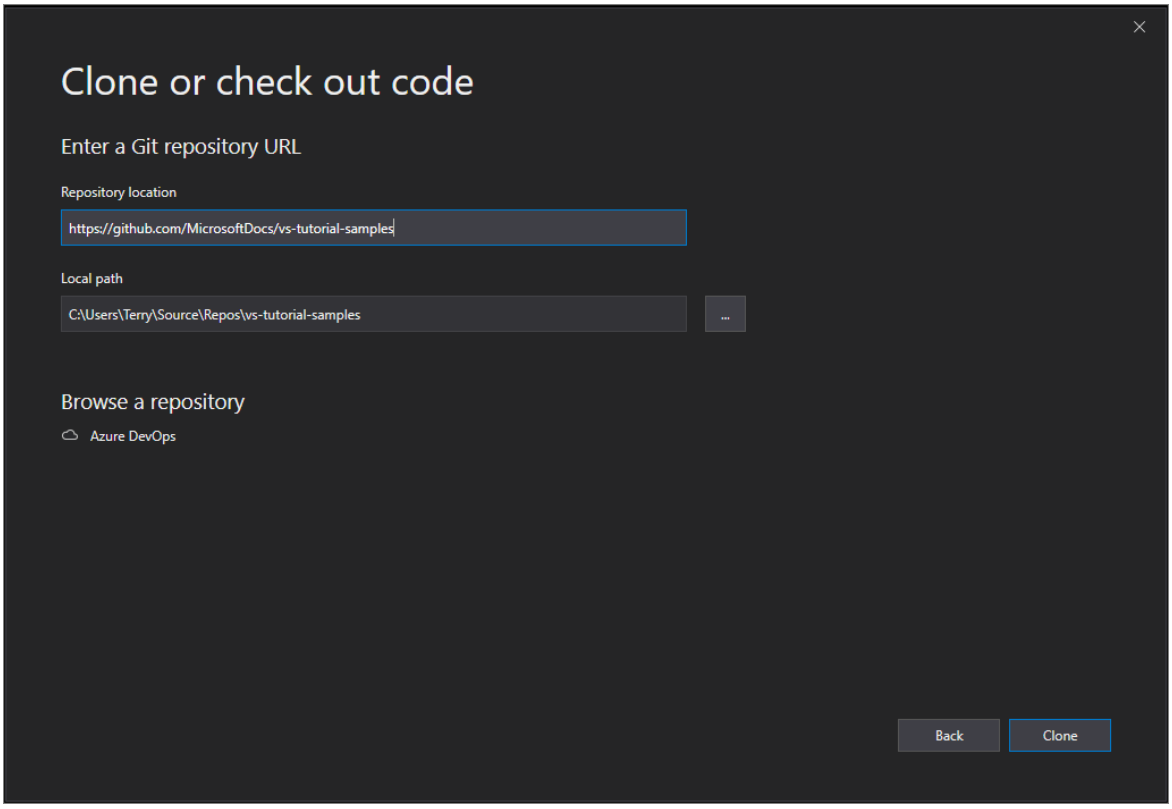


16.7 and earlier

1. Open Visual Studio 2019 version 16.7 or earlier.
2. On the start window, select **Clone or check out code**.

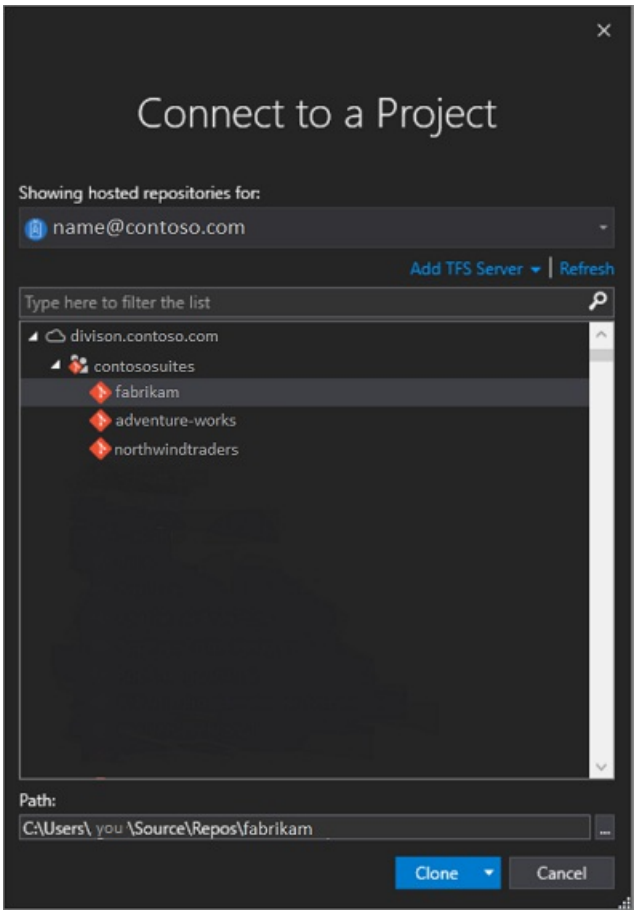


3. In the **Browse a repository** section, select **Azure DevOps**.



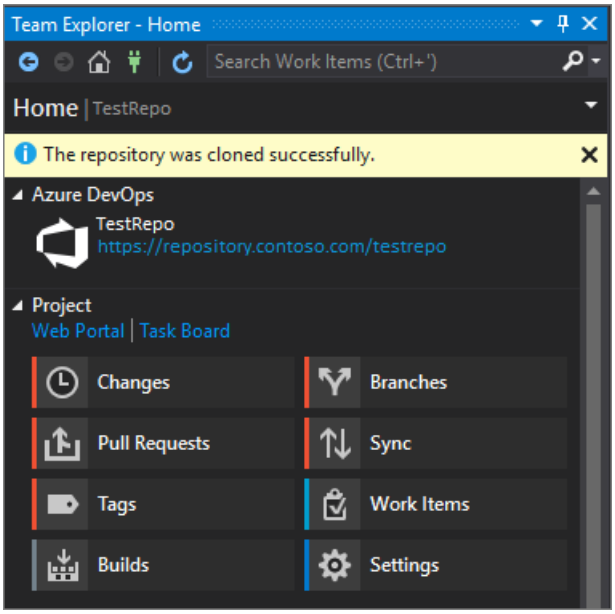
If you see a sign-in window, sign in to your account.

4. In the **Connect to a Project** dialog box, choose the repo that you want to connect to, and then select **Clone**.

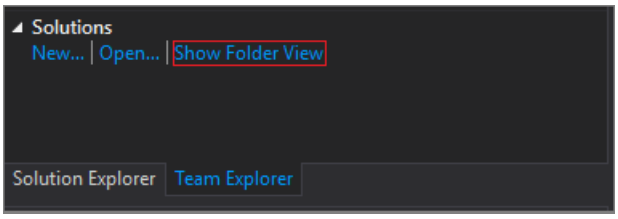


> **NOTE**
>
> What you see in the list box depends on the Azure DevOps repositories that you have access to.

Visual Studio opens **Team Explorer** and a notification appears when the clone is complete.

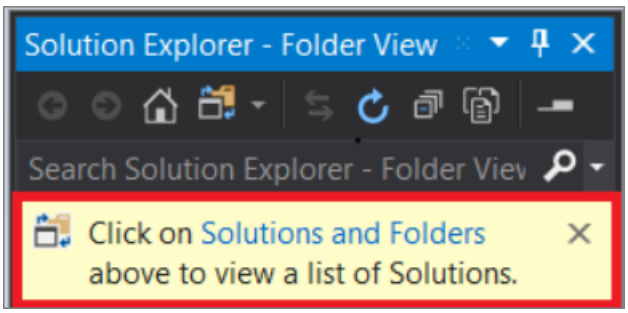


5. To view your folders and files, select the **Show Folder View** link.



Visual Studio opens **Solution Explorer**.

6. Choose the **Solutions and Folders** link to search for a solution file (specifically, an .sln file) to open.

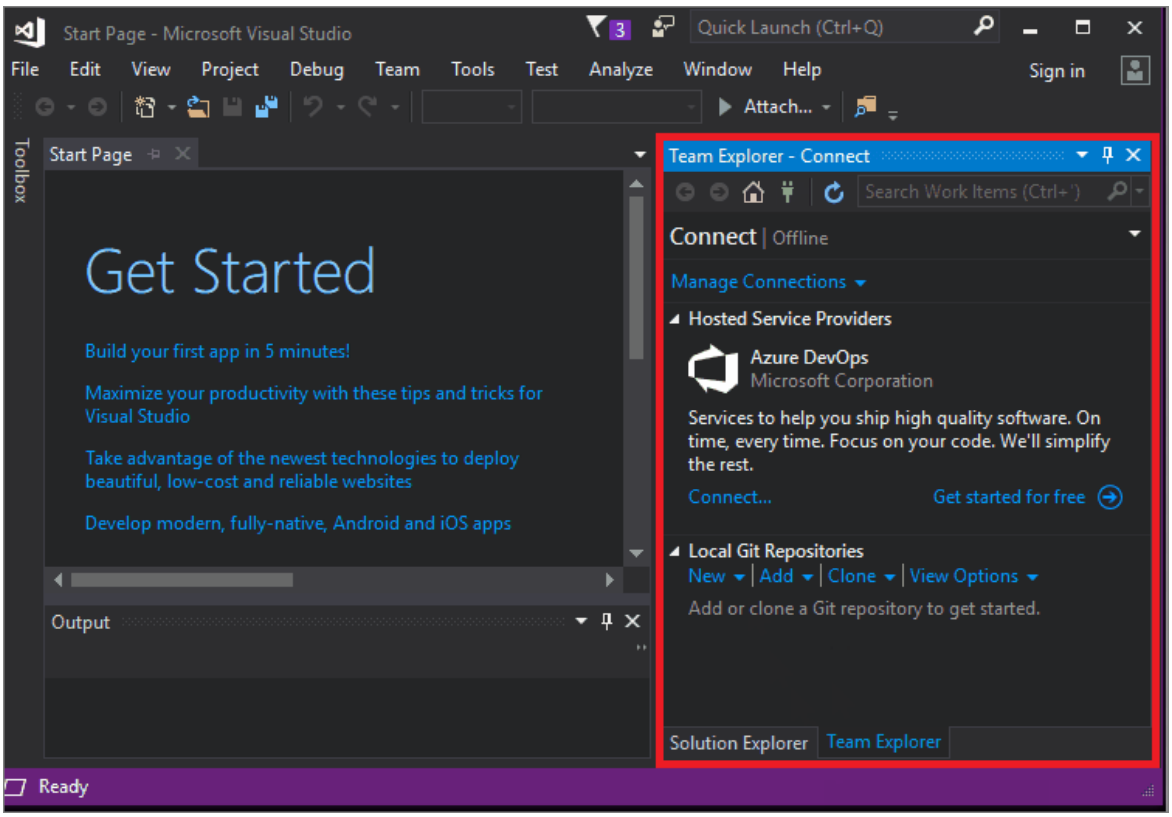


If you do not have a solution file in your repo, a 'No Solutions Found' message appears. However, you can double-click any file from the folder menu to open it in the Visual Studio code editor.

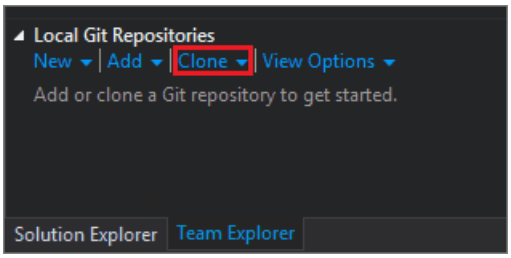
Open a project from a GitHub repo with Visual Studio 2017

1. Open Visual Studio 2017.
2. From the top menu bar, select **File** > **Open** > **Open from Source Control**.

   The **Team Explorer - Connect** pane opens.

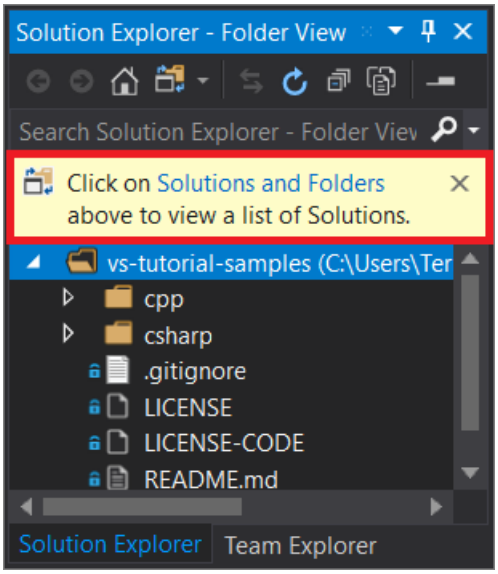


3. In the **Local Git Repositories** section, select **Clone**.

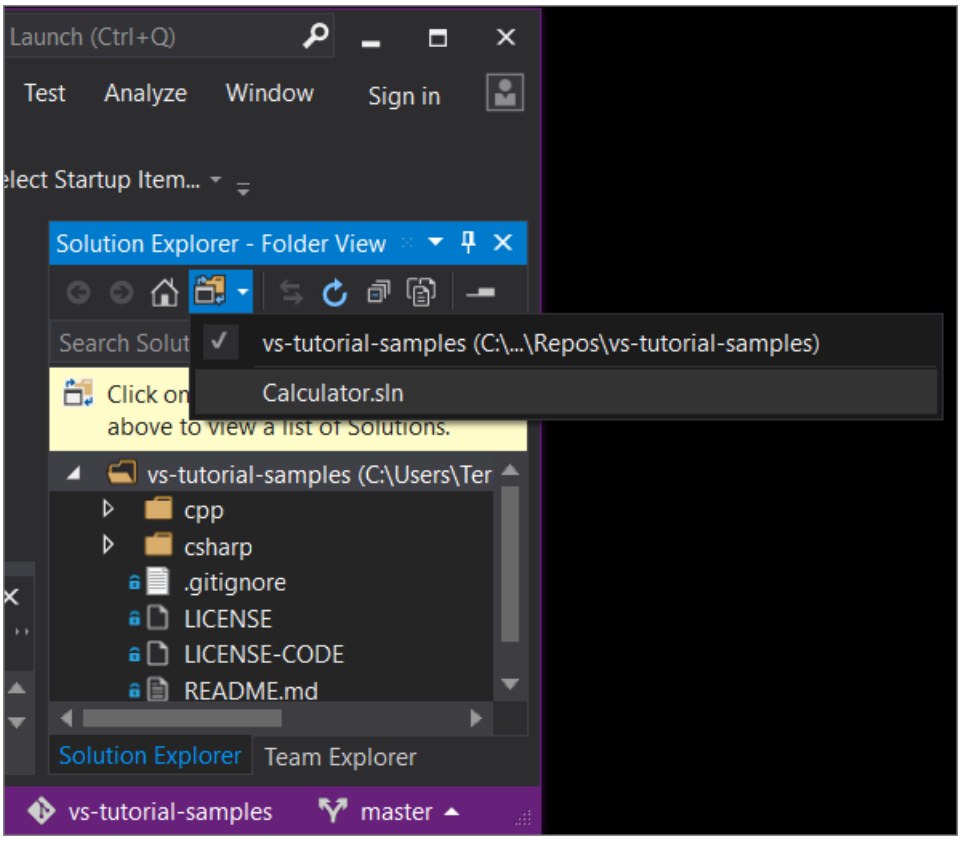


4. In the box that says ***Enter the URL of a Git repo to clone***, type or paste the URL for your repo, and then press **Enter**. (You might receive a prompt to sign in to GitHub; if so, do so.)

   After Visual Studio clones your repo, Team Explorer closes and Solution Explorer opens. A message appears that says *Click on Solutions and Folders above to view a list of Solutions.* Choose **Solutions and Folders**.



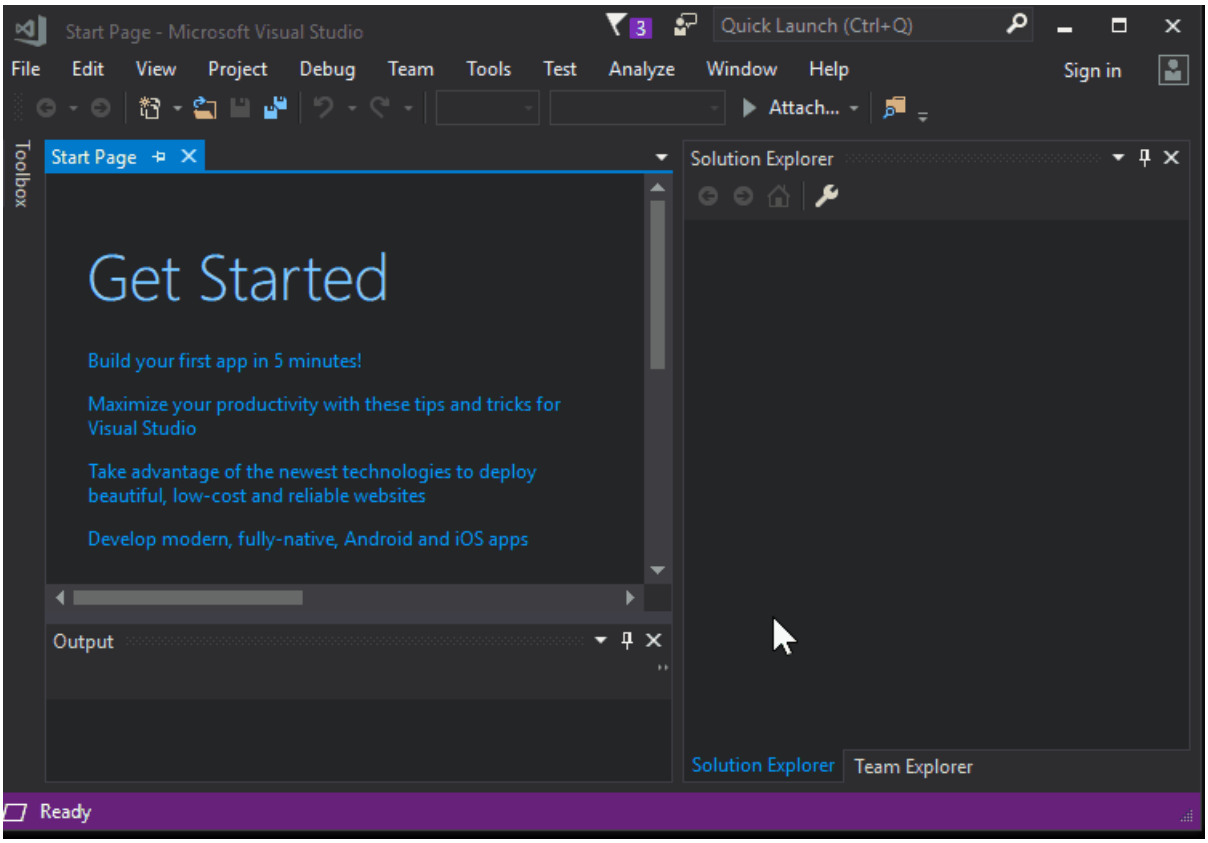
5. If you have a solution file available, it will appear in the "Solutions and Folders" fly-out menu. Choose it, and Visual Studio opens your solution.



If you do not have a solution file (specifically, a .sln file) in your repo, the fly-out menu will say "No Solutions Found." However, you can double-click any file from the folder menu to open it in the Visual Studio code editor.

Review your work

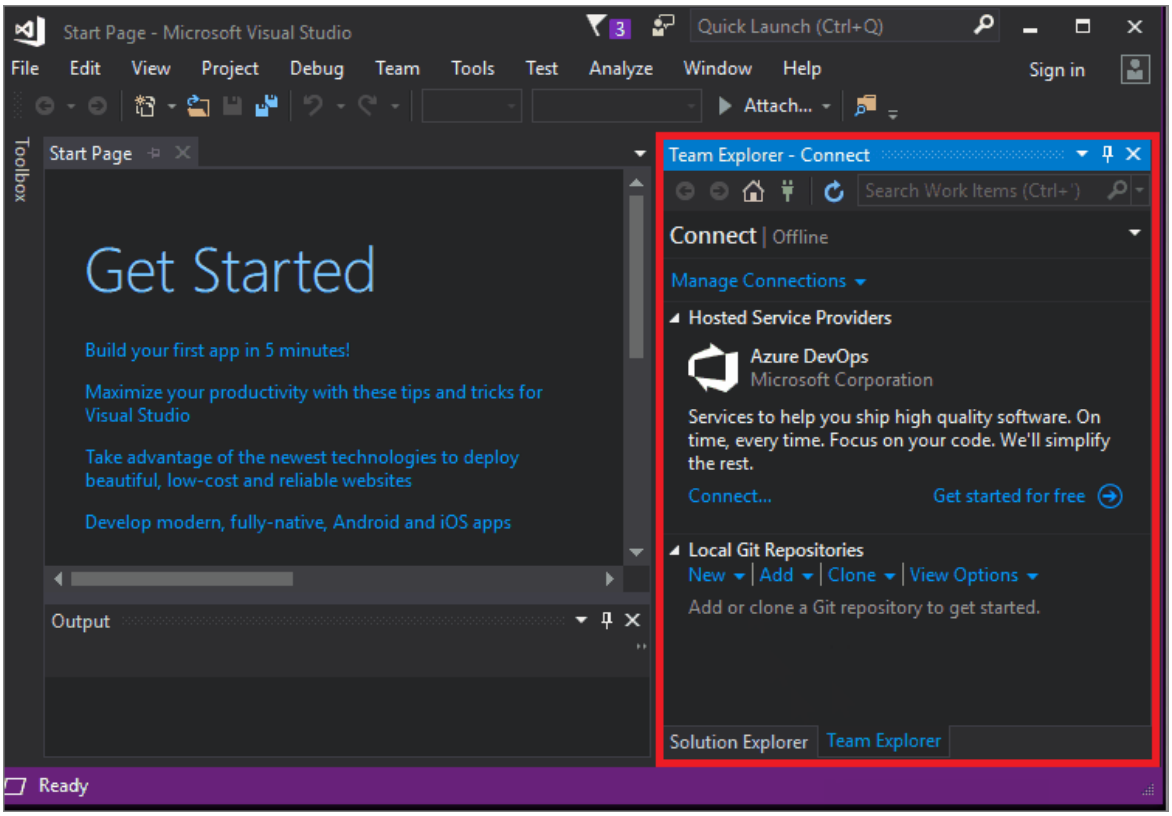
View the following animation to check the work that you completed in the previous section.



Open a project from an Azure DevOps repo with Visual Studio 2017

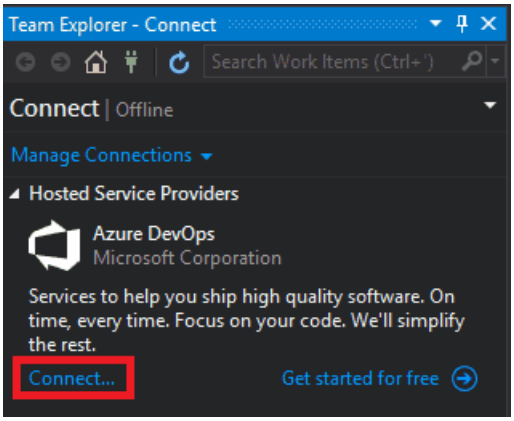
1. Open Visual Studio 2017.
2. From the top menu bar, select **File** > **Open** > **Open from Source Control**.

   The **Team Explorer - Connect** pane opens.

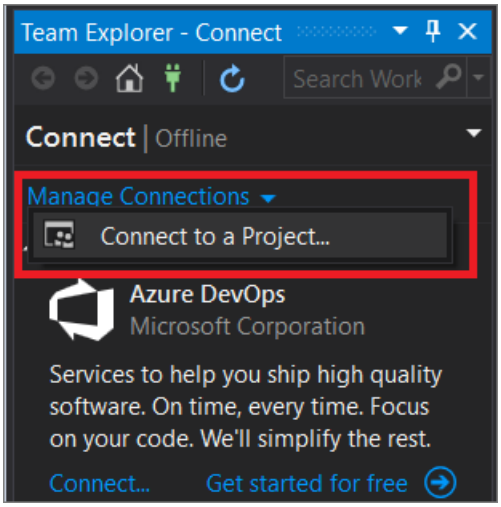


3. Here are two ways to connect to your Azure DevOps repo:

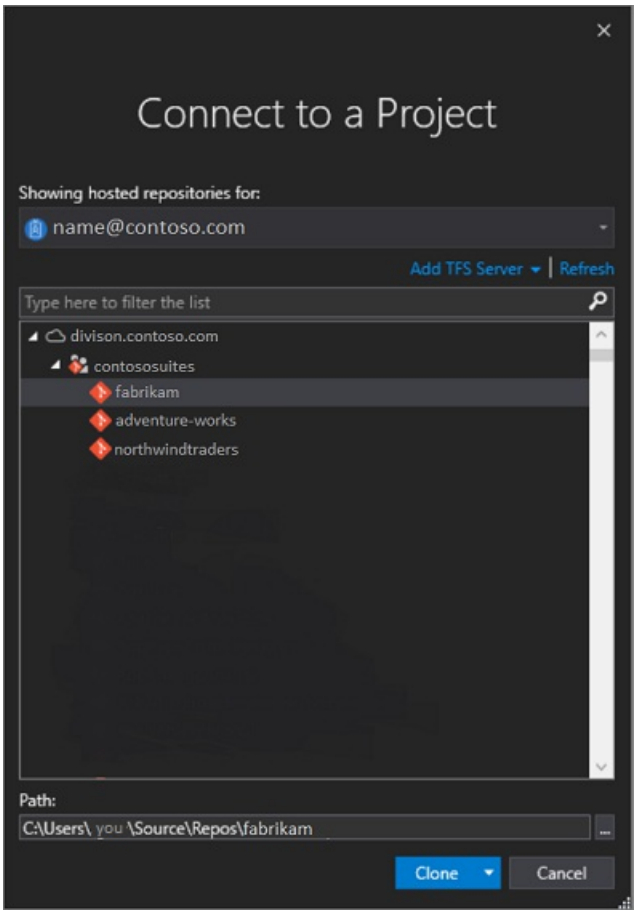
   - In the **Hosted Service Providers** section, select **Connect...**.



   - In the **Manage Connections** drop-down list, select **Connect to a Project...**.

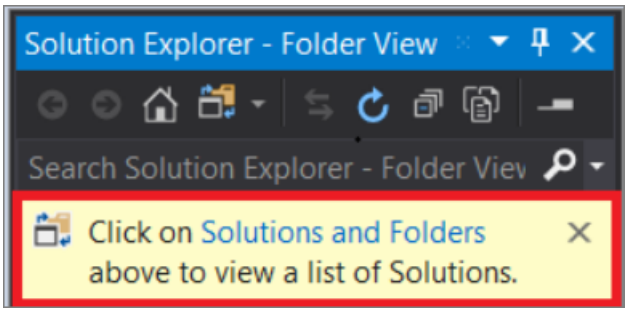


4. In the **Connect to a Project** dialog box, choose the repo that you want to connect to, and then select **Clone**.



> **NOTE**
>
> What you see in the list box depends on the Azure DevOps repositories that you have access to.

5. After Visual Studio clones your repo, Team Explorer closes and Solution Explorer opens. A message appears that says *Click on Solutions and Folders above to view a list of Solutions*. Choose **Solutions and Folders**.



   A solution file (specifically, a .sln file), will appear in the "Solutions and Folders" fly-out menu. Choose it, and Visual Studio opens your solution.

   If you do not have a solution file in your repo, the fly-out menu will say "No Solutions Found". However, you can double-click any file from the folder menu to open it in the Visual Studio code editor.

Next steps

Feel free to dive into any of the following language-specific tutorials:

- Visual Studio tutorials | C#
- Visual Studio tutorials | Visual Basic
- Visual Studio tutorials | C++

- Visual Studio tutorials | **Python**
- Visual Studio tutorials | **JavaScript**, **TypeScript**, and **Node.js**

See also

- The Git experience in Visual Studio
- Compare Git and Team Explorer side-by-side
- Microsoft Learn: Get started with Git and GitHub in Visual Studio
- Microsoft Learn: Get started with Azure DevOps
- Azure DevOps Services: Get started with Azure Repos and Visual Studio

Learn to use the code editor with C#

10/28/2021 • 11 minutes to read • Edit Online

In this 10-minute introduction to the code editor in Visual Studio, we'll add code to a file to look at some of the ways that Visual Studio makes writing, navigating, and understanding C# code easier.

> **TIP**
> If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

This article assumes you're already familiar with C#. If you aren't, we suggest you look at a tutorial such as Get started with C# and ASP.NET Core in Visual Studio first.

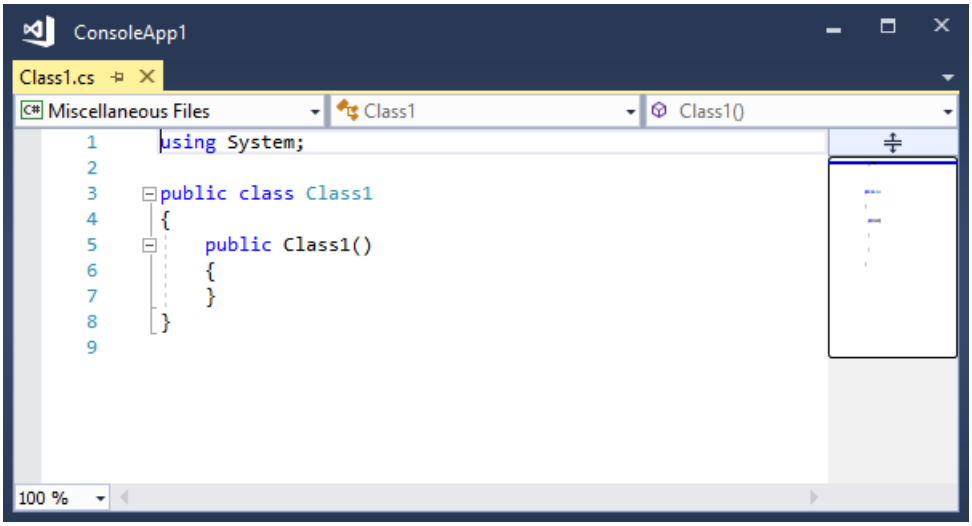
> **TIP**
> To follow along with this article, make sure you have the C# settings selected for Visual Studio. For information about selecting settings for the integrated development environment (IDE), see Select environment settings.

Create a new code file

Start by creating a new file and adding some code to it.

1. Open Visual Studio.
2. From the **File** menu on the menu bar, choose **New** > **File**, or press **Ctrl**+**N**.
3. In the **New File** dialog box, under the **General** category, choose **Visual C# Class**, and then choose **Open**.

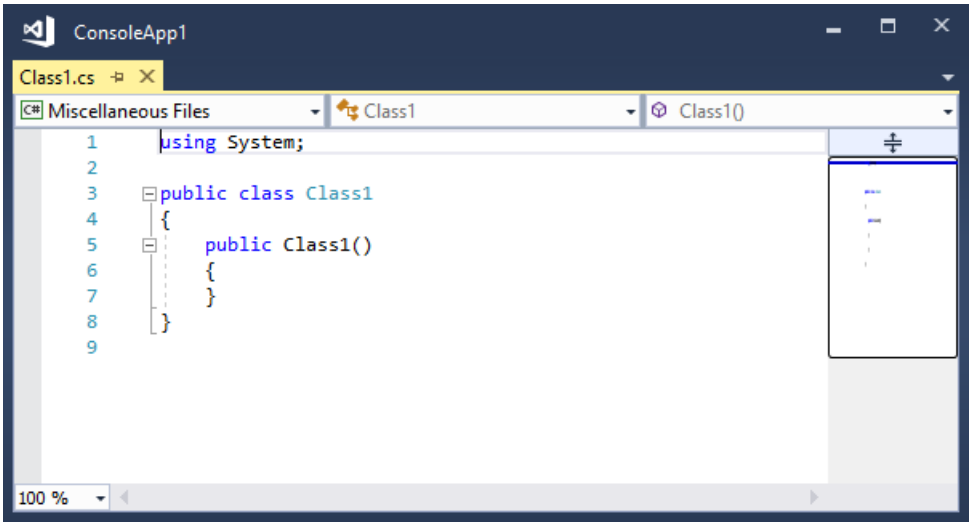
   A new file opens in the editor with the skeleton of a C# class. (Notice that we don't have to create a full Visual Studio project to gain some of the benefits that the code editor offers; all you need is a code file!)



1. Open Visual Studio. Press **Esc** or click **Continue without code** on the start window to open the development environment.

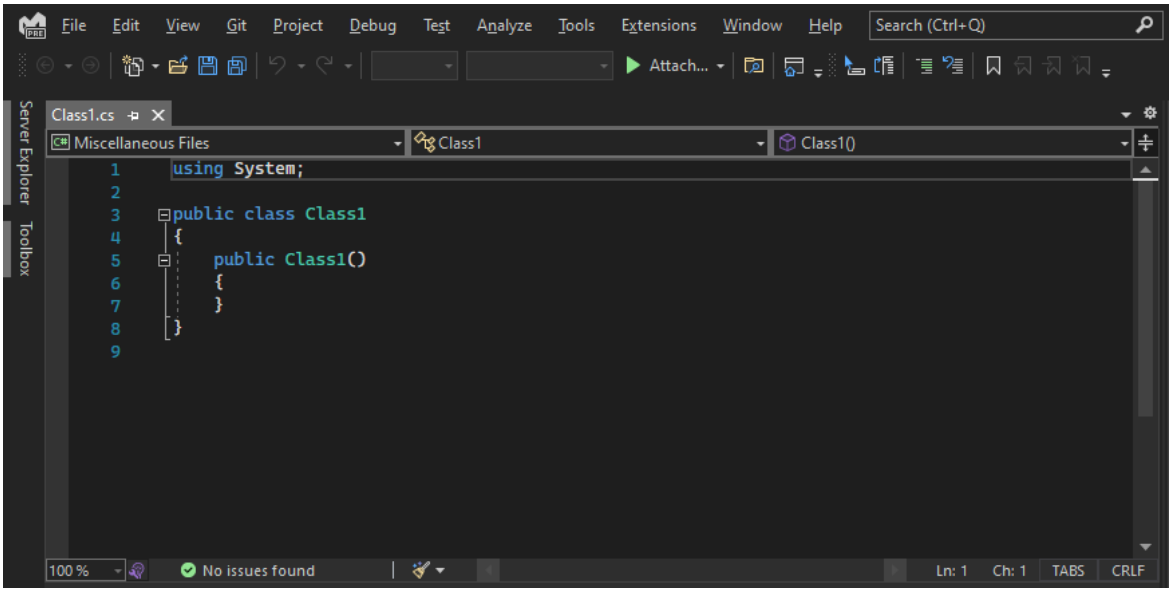
2. From the **File** menu on the menu bar, choose **New** > **File**, or press **Ctrl**+**N**.
3. In the **New File** dialog box, under the **General** category, choose **Visual C# Class**, and then choose **Open**.

   A new file opens in the editor with the skeleton of a C# class. (Notice that we don't have to create a full Visual Studio project to gain some of the benefits that the code editor offers; all you need is a code file!)



1. Open Visual Studio. Press **Esc**, or choose **Continue without code** on the start window, to open the development environment.
2. From the **File** menu on the menu bar, choose **New** > **File**, or press **Ctrl**+**N**.
3. In the **New File** dialog box, under the **General** category, choose **Visual C# Class**, and then choose **Open**.

   A new file opens in the editor with the skeleton of a C# class. You don't have to create a full Visual Studio project to gain some of the benefits that the code editor offers—all you need is a code file.



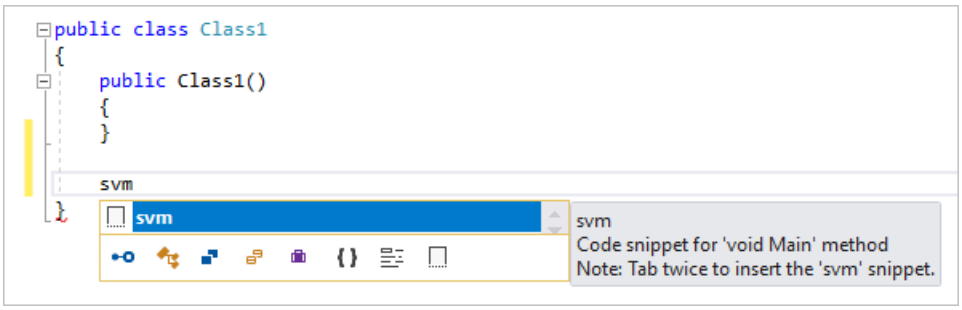
Use code snippets

Visual Studio provides useful *code snippets* that you can use to quickly and easily generate commonly used code blocks. Code snippets are available for different programming languages including C#, Visual Basic, and C++.

Let's add the C# `void Main` snippet to our file.

1. Place your cursor just above the final closing brace **}** in the file, and type the characters `svm` (which stands for `static void Main`—don't worry too much if you don't know what that means).

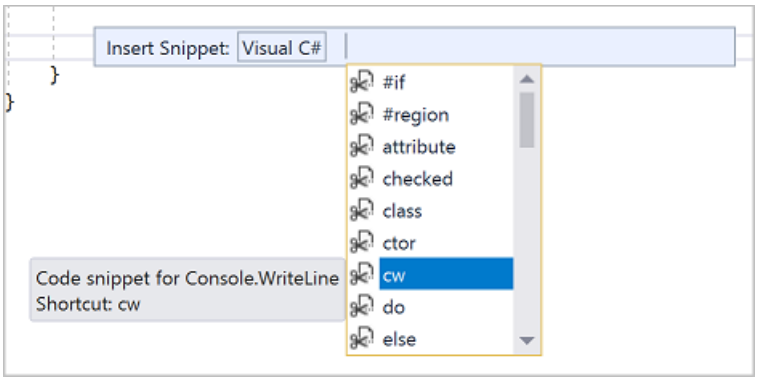
   A pop-up dialog box appears with information about the `svm` code snippet.



2. Press **Tab** twice to insert the code snippet.

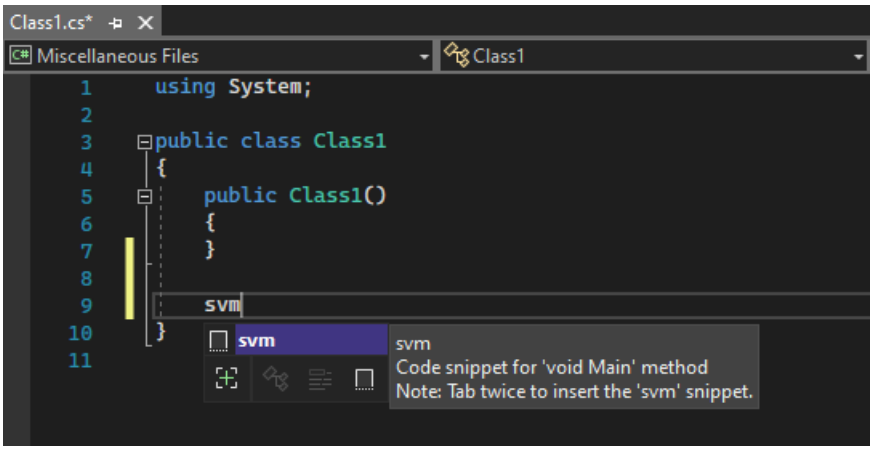
   You see the `static void Main()` method signature get added to the file. The Main() method is the entry point for C# applications.

The available code snippets vary for different programming languages. You can look at the available code snippets for your language by choosing **Edit** > **IntelliSense** > **Insert Snippet** or pressing **Ctrl**+**K**, **Ctrl**+**X**, and then choosing your language's folder. For C#, the list looks like this:



1. Place your cursor just above the final closing brace `}` in the file, and type the characters `svm`. `svm` stands for `static void Main`—don't worry if you don't know what that means yet.

   A pop-up dialog box appears with information about the `svm` code snippet.

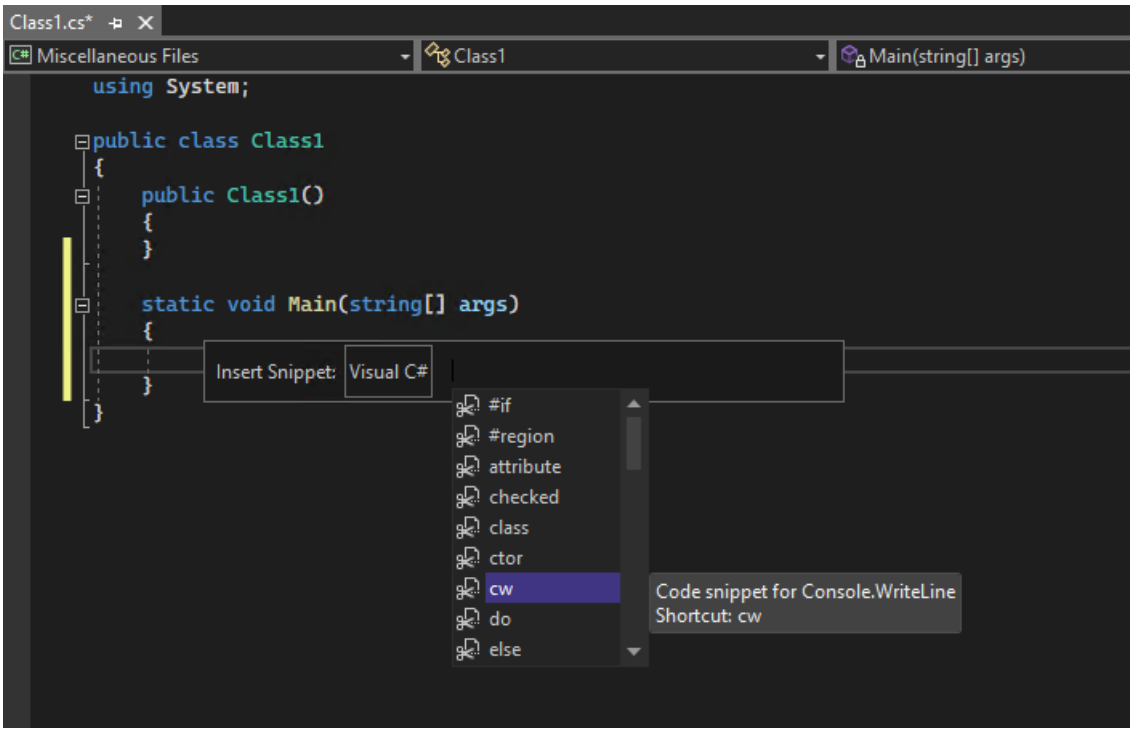


2. Press **Tab** twice to insert the code snippet.

   You'll see the `static void Main()` method signature get added to the file. The Main() method is the entry point for C# applications.

Available code snippets vary for different programming languages. You can look at the available code snippets

for your language by choosing **Edit** > **IntelliSense** > **Insert Snippet** or pressing **Ctrl**+**K**, **Ctrl**+**X**, and then choosing the folder for your programming language. For C#, the snippet list looks like this:



The list includes snippets for creating a class, a constructor, a for loop, an if or switch statement, and more.

Comment out code

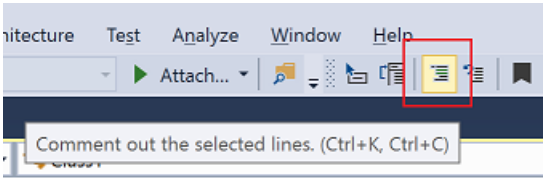
The toolbar, which is the row of buttons under the menu bar in Visual Studio, can help make you more productive as you code. For example, you can toggle IntelliSense completion mode (IntelliSense is a coding aid that displays a list of matching methods, amongst other things), increase or decrease a line indent, or comment out code that you don't want to compile. In this section, we'll comment out some code.

1. Paste the following code into the `Main()` method body.

```
// _words is a string array that we'll sort alphabetically
string[] _words = {
    "the",
    "quick",
    "brown",
    "fox",
    "jumps"
};

string[] morewords = {
    "over",
    "the",
    "lazy",
    "dog"
};

IEnumerable<string> query = from word in _words
                            orderby word.Length
                            select word;
```

2. We're not using the `morewords` variable, but we may use it later so we don't want to completely delete it.

Instead, let's comment out those lines. Select the entire definition of `morewords` to the closing semicolon, and then choose the **Comment out the selected lines** button on the toolbar. If you prefer to use the keyboard, press **Ctrl**+**K**, **Ctrl**+**C**.



The C# comment characters `//` are added to the beginning of each selected line to comment out the code.

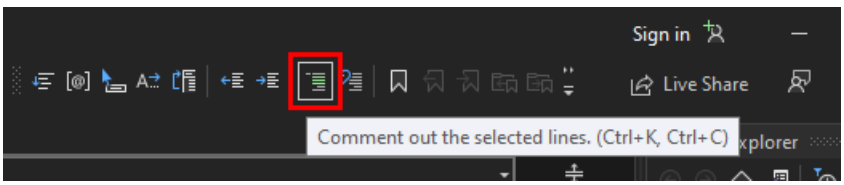
The toolbar, which is the row of buttons under the menu bar in Visual Studio, helps make you more productive as you code. For example, you can toggle IntelliSense completion mode, increase or decrease a line indent, or comment out code that you don't want to compile.

Let's comment out some code.

1. Paste the following code into the `Main()` method body.

```
// someWords is a string array.
string[] someWords = {
    "the",
    "quick",
    "brown",
    "fox",
    "jumps"
};

string[] moreWords = {
    "over",
    "the",
    "lazy",
    "dog"
};

// Alphabetically sort the words.
IEnumerable<string> query = from word in someWords
                            orderby word
                            select word;
```

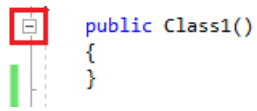
2. We're not using the `moreWords` variable, but we might use it later so we don't want to delete it. Instead, we'll comment out those lines. Select the entire definition of `moreWords` down to the closing semicolon, and then choose the **Comment out the selected lines** button on the toolbar. If you prefer to use the keyboard, press **Ctrl**+**E**, **Ctrl**+**C**.



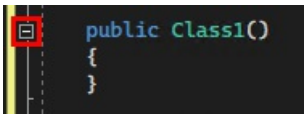
The C# comment characters `//` are added to the beginning of each selected line to comment out the code.

Collapse code blocks

We don't want to see the empty constructor that was generated for `Class1`, so to unclutter our view of the code, let's collapse it. Choose the small gray box with the minus sign inside it in the margin of the first line of the constructor. Or, if you prefer to use the keyboard, place the cursor anywhere in the constructor code and press **Ctrl+M**, **Ctrl+M**.



The code block collapses to just the first line, followed by an ellipsis (`...`). To expand the code block again, click the same gray box that now has a plus sign in it, or press **Ctrl+M**, **Ctrl+M** again. This feature is called Outlining and is especially useful when you're collapsing long methods or entire classes.



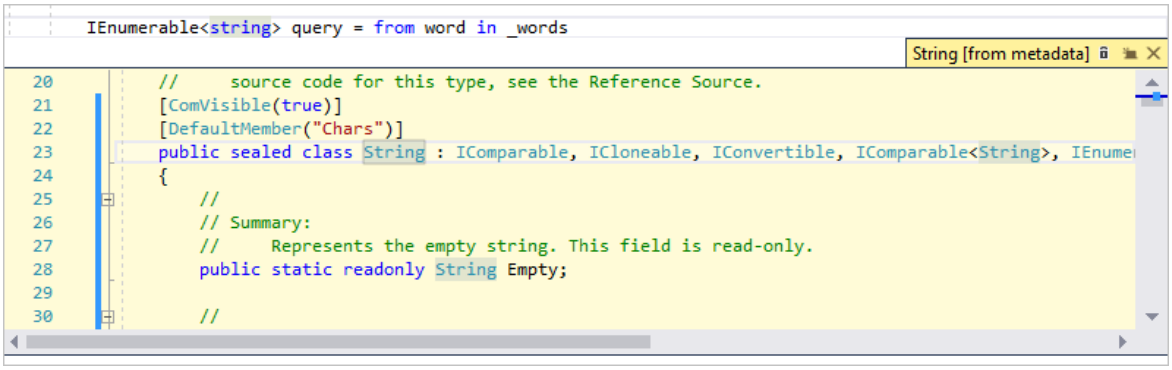
The code block collapses to just the first line, followed by an ellipsis (`...`). To expand the code block again, select the same gray box that now has a plus sign in it, or press **Ctrl+M**, **Ctrl+M** again. This feature is called Outlining and is especially useful when you're collapsing long methods or entire classes.

View symbol definitions

The Visual Studio editor makes it easy to inspect the definition of a type, method, etc. One way is to navigate to the file that contains the definition, for example by choosing **Go to Definition** or pressing **F12** anywhere the symbol is referenced. An even quicker way that doesn't move your focus away from the file you're working in is to use Peek Definition. Let's peek at the definition of the `string` type.

1. Right-click on any occurrence of `string` and choose **Peek Definition** from the content menu. Or, press **Alt+F12**.

   A pop-up window appears with the definition of the `String` class. You can scroll within the pop-up window, or even peek at the definition of another type from the peeked code.



2. Close the peeked definition window by choosing the small box with an "x" at the top right of the pop-up window.

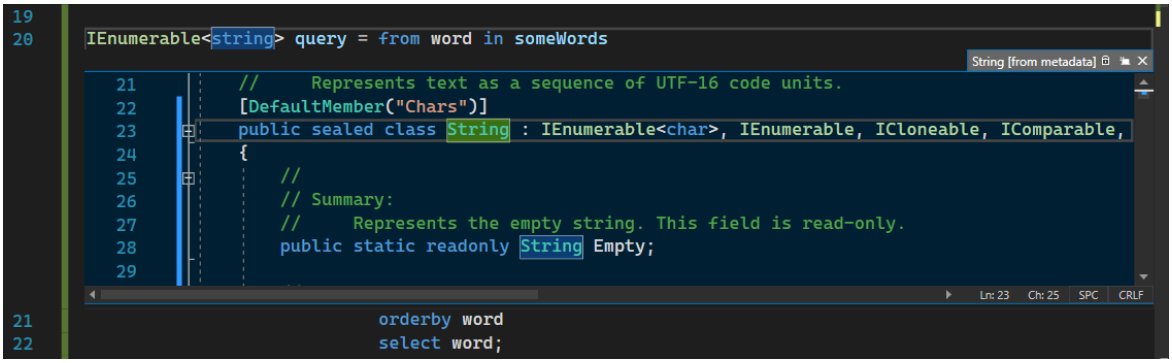
The Visual Studio editor makes it easy to inspect the definition of a type, method, or variable. One way is to go to the definition, in whichever file has it, by choosing **Go to Definition** or pressing **F12** anywhere a symbol is referenced. An even quicker way that doesn't move your focus away from the code you're working on is to use Peek Definition.

Let's peek at the definition of the `string` type.

1. Right-click on any occurrence of `string` and choose **Peek Definition** from the content menu. Or, press

Alt+F12.

A pop-up window appears with the definition of the `String` class. You can scroll within the pop-up window, or even peek at the definition of another type from the peeked code.



2. Close the peek definition window by choosing the small box with an "x" at the top right of the pop-up window.

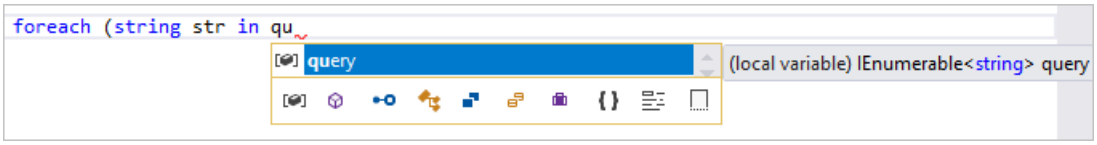
Use IntelliSense to complete words

IntelliSense is an invaluable resource when you're coding. It can show you information about available members of a type, or parameter details for different overloads of a method. You can also use IntelliSense to complete a word after you type enough characters to disambiguate it. Let's add a line of code to print out the ordered strings to the console window, which is the standard place for output from the program to go.

1. Below the `query` variable, start typing the following code:

```
foreach (string str in qu
```

You see IntelliSense show you **Quick Info** about the `query` symbol.



2. To insert the rest of the word `query` by using IntelliSense's word completion functionality, press **Tab**.

3. Finish off the code block to look like the following code. You can even practice using code snippets again by entering `cw` and then pressing **Tab** twice to generate the `Console.WriteLine` code.

```
foreach (string str in query)
{
   Console.WriteLine(str);
}
```

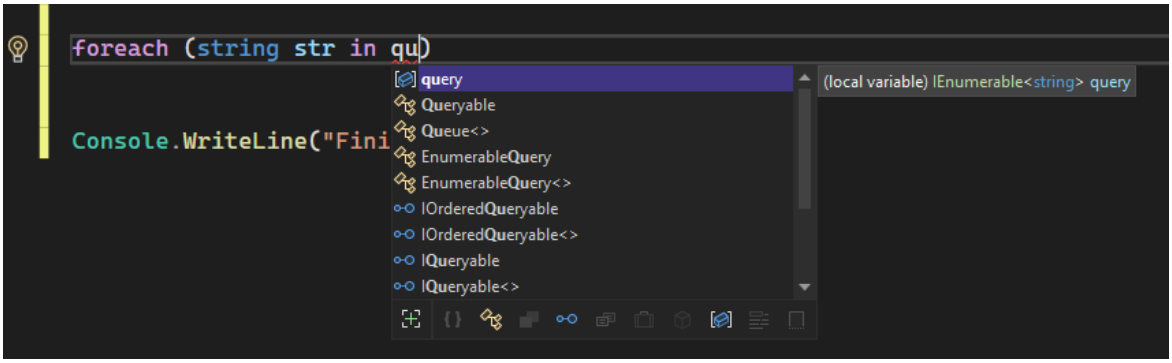
IntelliSense is an invaluable resource when you're coding. It can show you information about available members of a type, or parameter details for different overloads of a method. You can also use IntelliSense to complete a word after you type enough characters to disambiguate it.

Let's add a line of code to print out the ordered strings to the console window, which is the standard place for output from the program to go.

1. Below the `query` variable, start typing the following code:

```
foreach (string str in qu
```

You'll see an IntelliSense pop-up appear with information about the `query` symbol.



2. To insert the rest of the word `query` by using IntelliSense word completion, press **Tab**.

3. Finish off the code block to look like the following code. You can practice further with code snippets by entering `cw` and then pressing **Tab** twice to generate the `Console.WriteLine` statement.

```
foreach (string str in query)
{
   Console.WriteLine(str);
}
```

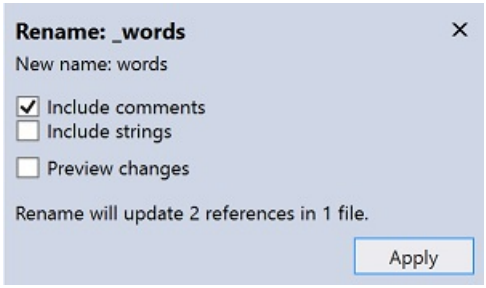
Refactor a name

Nobody gets code right the first time, and one of the things you might have to change is the name of a variable or method. Let's try out Visual Studio's refactor functionality to rename the `_words` variable to `words`.

1. Place your cursor over the definition of the `_words` variable, and choose **Rename** from the right-click or context menu, or press **Ctrl**+**R**, **Ctrl**+**R**.

   A pop-up **Rename** dialog box appears at the top right of the editor.

2. Enter the desired name **words**. Notice that the reference to `words` in the query is also automatically renamed. Before you press **Enter**, select the **Include comments** checkbox in the **Rename** pop-up box.



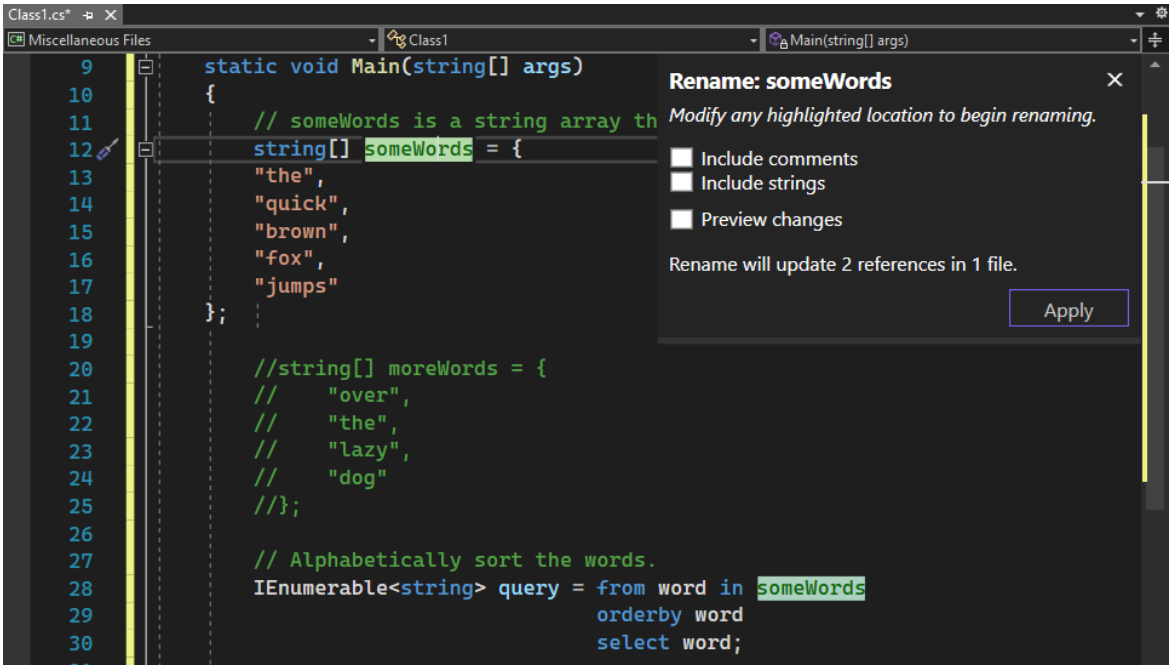
3. Press **Enter**.

   Both occurrences of `words` have been renamed, as well as the reference to `words` in the code comment.

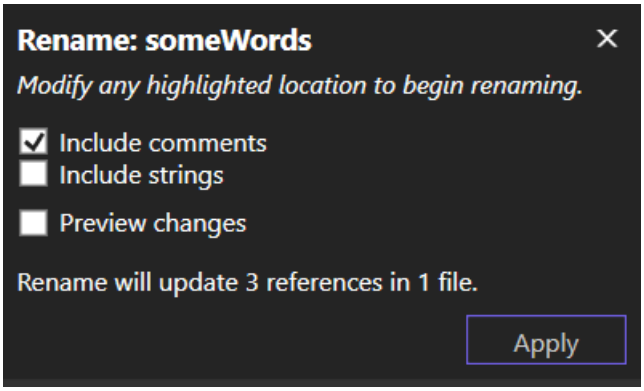
Nobody gets code right the first time, and one of the things you might have to change is the name of a variable or method. Let's try out Visual Studio's refactor functionality to rename the `someWords` variable to `unsortedWords`.

1. Place your cursor over the definition of the `someWords` variable, and choose **Rename** from the right-click or context menu, or press **F2**.

A **Rename** dialog box appears at the top right of the editor.



2. Enter the desired name **unsortedWords**. You'll see that the reference to `unsortedWords` in the `query` assignment statement is also automatically renamed. Before you press **Enter**, select the **Include comments** checkbox in the **Rename** pop-up box.



3. Press **Enter**, or choose **Apply** in the **Rename** dialog box.

   Both occurrences of `someWords` in your code have been renamed, as well as the text `someWords` in your code comment.

Next steps

Learn about projects and solutions

See also

- Code snippets
- Navigate code
- Outlining
- Go To Definition and Peek Definition
- Refactoring
- Use IntelliSense

Compile and build in Visual Studio

10/28/2021 • 2 minutes to read • Edit Online

For a first introduction to building within the IDE, see Walkthrough: Building an application.

You can use any of the following methods to build an application: the Visual Studio IDE, the MSBuild command-line tools, and Azure Pipelines:

BUILD METHOD	BENEFITS
IDE	- Create builds immediately and test them in a debugger. - Run multi-processor builds for C++ and C# projects. - Customize different aspects of the build system.
CMake	- Build projects using the CMake tool - Use the same build system across Linux and Windows platforms.
MSBuild command line	- Build projects without installing Visual Studio. - Run multi-processor builds for all project types. - Customize most areas of the build system.
Azure Pipelines	- Automate your build process as part of a continuous integration/continuous delivery pipeline. - Apply automated tests with every build. - Employ virtually unlimited cloud-based resources for build processes. - Modify the build workflow and create build activities to perform deeply customized tasks.

The documentation in this section goes into further details of the IDE-based build process. For more information on the other methods, see CMake, MSBuild and Azure Pipelines, respectively.

> **NOTE**
> This topic applies to Visual Studio on Windows. For Visual Studio for Mac, see Compile and build in Visual Studio for Mac.

Overview of building from the IDE

When you create a project, Visual Studio created default build configurations for the project and the solution that contains the project. These configurations define how the solutions and projects are built and deployed. Project configurations in particular are unique for a target platform (such as Windows or Linux) and build type (such as debug or release). You can edit these configurations however you like, and can also create your own configurations as needed.

For a first introduction to building within the IDE, see Walkthrough: Building an application.

Next, see Building and cleaning projects and solutions in Visual Studio to learn about the different customizations you can make to the process. Customizations include changing output directories, specifying custom build events, managing project dependencies, managing build log files, and suppressing compiler warnings.

From there, you can explore a variety of other tasks:

- Understand build configurations
- Understand build platforms
- Manage project and solution properties.
- Specify build events in C# and Visual Basic.
- Set build options
- Build multiple projects in parallel.

See also

- Building (compiling) website projects
- Compile and build (Visual Studio for Mac)
- CMake projects in Visual Studio

Tutorial: Learn to debug C# code using Visual Studio

10/28/2021 • 22 minutes to read • Edit Online

This article introduces the features of the Visual Studio debugger in a step-by-step walkthrough. If you want a higher-level view of the debugger features, see First look at the debugger. When you *debug your app*, it usually means that you are running your application with the debugger attached. When you do this, the debugger provides many ways to see what your code is doing while it runs. You can step through your code and look at the values stored in variables, you can set watches on variables to see when values change, you can examine the execution path of your code, see whether a branch of code is running, and so on. If this is the first time that you've tried to debug code, you might want to read Debugging for absolute beginners before going through this article.

Although the demo app is C#, most of the features are applicable to C++, Visual Basic, F#, Python, JavaScript, and other languages supported by Visual Studio (F# does not support Edit-and-continue. F# and JavaScript do not support the **Autos** window). The screenshots are in C#.

In this tutorial, you will:

- Start the debugger and hit breakpoints.
- Learn commands to step through code in the debugger
- Inspect variables in data tips and debugger windows
- Examine the call stack

Prerequisites

You must have Visual Studio 2022 RC installed and the **.NET desktop development** workload.

You must have Visual Studio 2019 installed and the **.NET Core cross-platform development** workload.

You must have Visual Studio 2017 installed and the **.NET Core cross-platform development** workload.

If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

If you need to install the workload but already have Visual Studio, go to **Tools** > **Get Tools and Features...**, which opens the Visual Studio Installer. The Visual Studio Installer launches. Choose the **.NET Core cross-platform development** workload, then choose **Modify**.

If you already have Visual Studio but the **.NET desktop development** workload isn't installed, go to **Tools** > **Get Tools and Features...**, which launches the Visual Studio Installer. In the Visual Studio Installer, choose the **.NET desktop development** workload, then choose **Modify**.

Create a project

First, you'll create a .NET Core console application project. The project type comes with all the template files you'll need, before you've even added anything!

1. Open Visual Studio 2017.
2. From the top menu bar, choose **File** > **New** > **Project**.

3. In the **New Project** dialog box in the left pane, expand **C#**, and then choose **.NET Core**. In the middle pane, choose **Console App (.NET Core)**. Then name the project *get-started-debugging*.

   If you don't see the **Console App (.NET Core)** project template, choose the **Open Visual Studio Installer** link in the left pane of the **New Project** dialog box.

   The Visual Studio Installer launches. Choose the **.NET Core cross-platform development** workload, and then choose **Modify**.

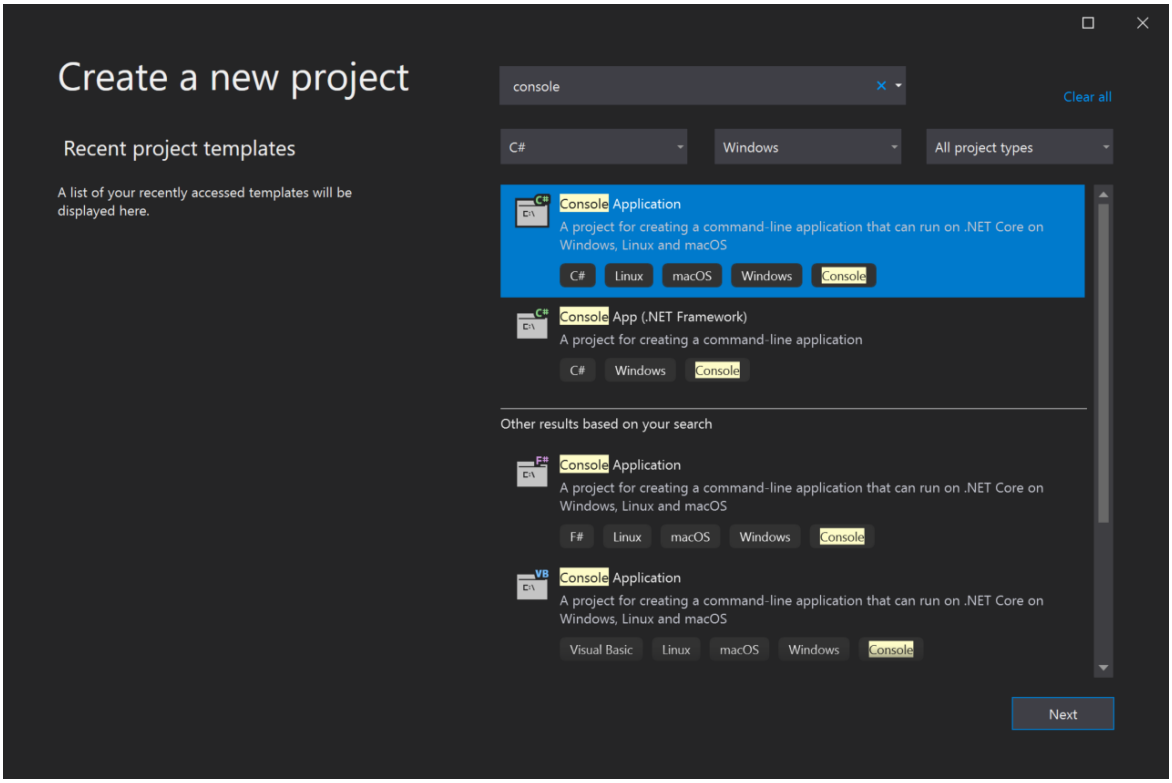
1. Open Visual Studio.

   If the start window is not open, choose **File** > **Start Window**.

2. On the start window, choose **Create a new project**.

3. On the **Create a new project** window, enter or type *console* in the search box. Next, choose **C#** from the Language list, and then choose **Windows** from the Platform list.

   After you apply the language and platform filters, choose the **Console App** template for .NET Core, and then choose **Next**.



> **NOTE**
>
> If you do not see the **Console App** template, you can install it from the **Create a new project** window. In the **Not finding what you're looking for?** message, choose the **Install more tools and features** link. Then, in the Visual Studio Installer, choose the **.NET Core cross-platform development** workload.

4. In the **Configure your new project** window, type or enter *GetStartedDebugging* in the **Project name** box. Then, choose **Next**.

5. Choose either the recommended target framework (.NET Core 3.1) or .NET 5, and then choose **Create**.

   Visual Studio opens your new project.

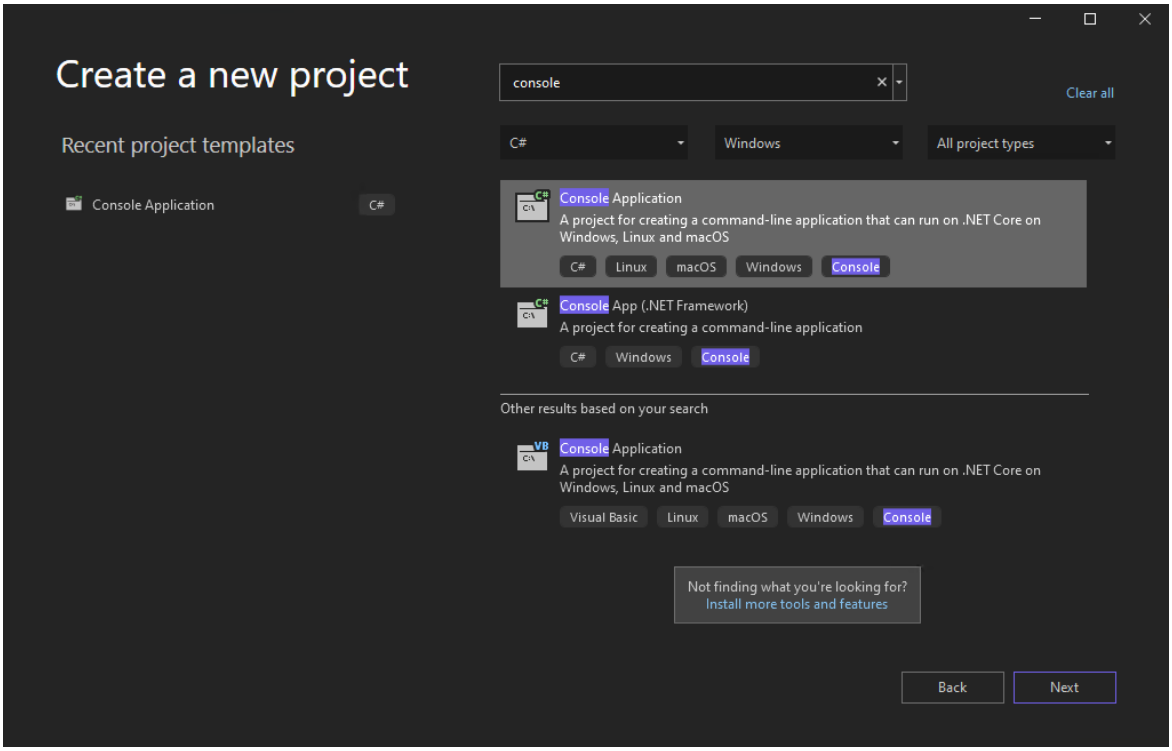
1. Open Visual Studio.

If the start window isn't open, choose **File** > **Start Window**.

2. On the start window, choose **Create a new project**.

3. On the **Create a new project** window, enter or type *console* in the search box. Next, choose **C#** from the Language list, and then choose **Windows** from the Platform list.

   After you apply the language and platform filters, choose the **Console Application** template, and then choose **Next**.



> **NOTE**
> If you don't see the **Console Application** template, you can install it from the **Create a new project** window. In the **Not finding what you're looking for?** message, choose the **Install more tools and features** link. Then, in the Visual Studio Installer, choose the **.NET desktop development** workload.

4. In the **Configure your new project** window, type or enter *GetStartedDebugging* in the **Project name** box. Then, choose **Next**.

5. In the **Additional information** window, ensure that **.NET 6.0 (Preview)** is selected in the **Framework** dropdown menu, and then choose **Create**.

   Visual Studio opens your new project.

Create the application

In *Program.cs*, replace all of the default code with the following code:

```
using System;

class ArrayExample
{
    static void Main()
    {
        char[] letters = { 'f', 'r', 'e', 'd', ' ', 's', 'm', 'i', 't', 'h'};
        string name = "";
        int[] a = new int[10];
        for (int i = 0; i < letters.Length; i++)
        {
            name += letters[i];
            a[i] = i + 1;
            SendMessage(name, a[i]);
        }
        Console.ReadKey();
    }

    static void SendMessage(string name, int msg)
    {
        Console.WriteLine("Hello, " + name + "! Count to " + msg);
    }
}
```

Start the debugger!

1. Press **F5** (**Debug > Start Debugging**) or the **Start Debugging** button in the Debug Toolbar.

   F5 starts the app with the debugger attached to the app process, but right now we haven't done anything special to examine the code. So the app just loads and you'll see this console output.

   ```
   Hello, f! Count to 1
   Hello, fr! Count to 2
   Hello, fre! Count to 3
   Hello, fred! Count to 4
   Hello, fred ! Count to 5
   Hello, fred s! Count to 6
   Hello, fred sm! Count to 7
   Hello, fred smi! Count to 8
   Hello, fred smit! Count to 9
   Hello, fred smith! Count to 10
   ```

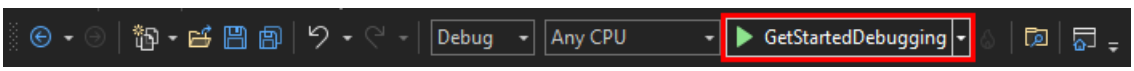
   In this tutorial, we'll take a closer look at this app using the debugger and get a look at the debugger features.

2. Stop the debugger by pressing the red stop button (**Shift** + **F5**).

3. In the console window, press a key to close the console window.

Mostly, we use keyboard shortcuts here, because it's a fast way to execute debugger commands. Equivalent commands, such as toolbar or menu commands, are also noted.

1. To start the debugger, select **F5**, or choose the **Debug Target** button in the Standard toolbar, or choose the **Start Debugging** button in the Debug toolbar, or choose **Debug** > **Start Debugging** from the menu bar.

   F5 starts the app with the debugger attached to the app process. Since we haven't done anything special to examine the code, the app runs to completion and you see the console output.

```
Hello, f! Count to 1
Hello, fr! Count to 2
Hello, fre! Count to 3
Hello, fred! Count to 4
Hello, fred ! Count to 5
Hello, fred s! Count to 6
Hello, fred sm! Count to 7
Hello, fred smi! Count to 8
Hello, fred smit! Count to 9
Hello, fred smith! Count to 10
```

2. To stop the debugger, select **Shift+F5**, or choose the **Stop Debugging** button in the Debug toolbar, or choose **Debug** > **Stop Debugging** from the menu bar.

3. In the console window, select any key to close the console window.

Set a breakpoint and start the debugger

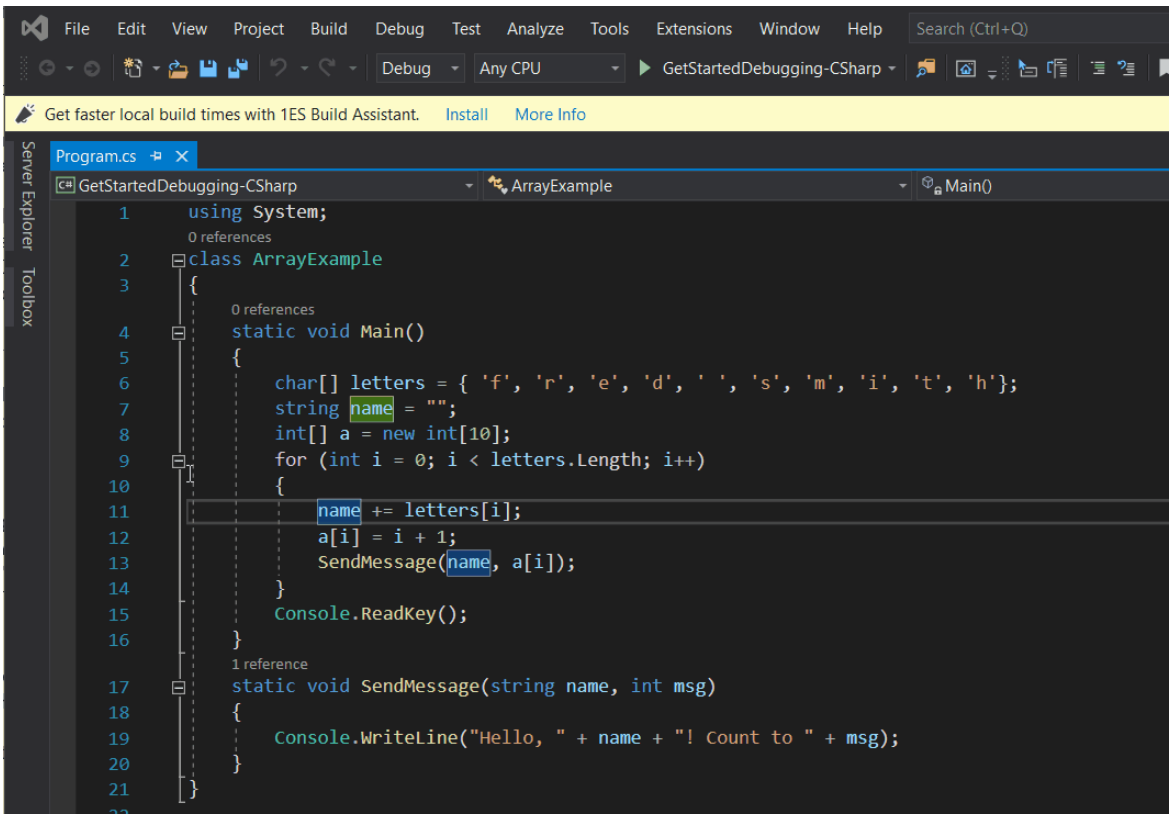
1. In the `for` loop of the `Main` function, set a breakpoint by clicking the left margin of the following line of code:

   ```
   name += letters[i];
   ```

   A red circle appears where you set the breakpoint.

   Breakpoints are one of the most basic and essential features of reliable debugging. A breakpoint indicates where Visual Studio should suspend your running code so you can take a look at the values of variables, or the behavior of memory, or whether or not a branch of code is getting run.

2. Press **F5** or the **Start Debugging** button, the app starts, and the debugger runs to the line of code where you set the breakpoint.



The yellow arrow represents the statement on which the debugger paused, which also suspends app

execution at the same point (this statement has not yet executed).

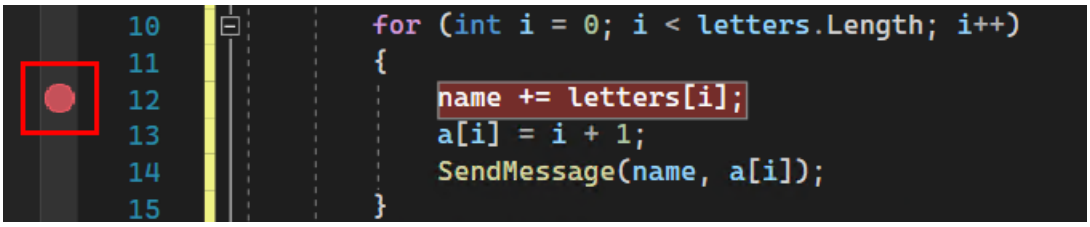
If the app is not yet running, **F5** starts the debugger and stops at the first breakpoint. Otherwise, **F5** continues running the app to the next breakpoint.

Breakpoints are a useful feature when you know the line of code or the section of code that you want to examine in detail. For information on the different types of breakpoints you can set, such as conditional breakpoints, see Using breakpoints.

1. In the `for` loop of the `Main` function, set a breakpoint by clicking the left margin of the following line of code:

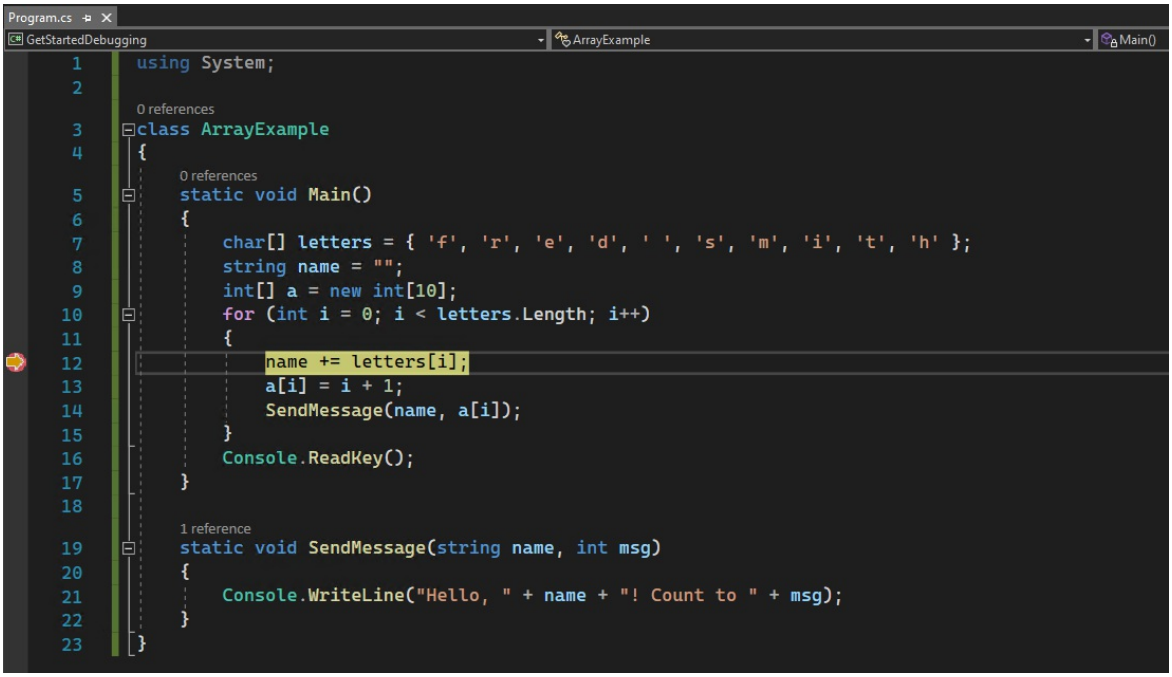
   ```
   name += letters[i];
   ```

   A red circle appears where you set the breakpoint.

   Breakpoints are an essential feature of reliable debugging. You can set breakpoints where you want Visual Studio to pause your running code so you can look at the values of variables or the behavior of memory, or know whether or not a branch of code is getting run.

2. To start debugging, select **F5**, or choose the **Debug Target** button in the Standard toolbar, or choose the **Start Debugging** button in the Debug toolbar, or choose **Debug** > **Start Debugging** from the menu bar. The app starts and the debugger runs to the line of code where you set the breakpoint.

   The yellow arrow points to the statement on which the debugger paused. App execution is paused at the same point, with the statement not yet executed.

   When the app isn't running, **F5** will start the debugger, which will run the app until it reaches the first breakpoint. If the app is paused at a breakpoint, then **F5** will continue running the app until it reaches the next breakpoint.

   Breakpoints are a useful feature when you know the line or section of code that you want to examine in

detail. For more about the different types of breakpoints you can set, such as conditional breakpoints, see Using breakpoints.

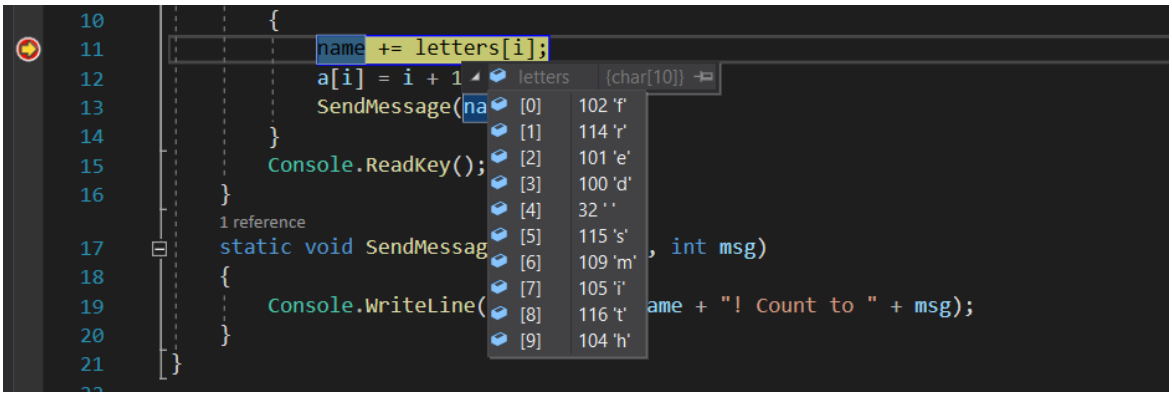
Navigate code and inspect data by using data tips

Mostly, we use the keyboard shortcuts here, because it's a good way to get fast at executing your app in the debugger (equivalent commands such as menu commands are shown in parentheses).

1. While paused on the `name += letters[i]` statement, hover over the `letters` variable and you see it's default value, the value of the first element in the array, `char[10]`.

   Features that allow you to inspect variables are one of the most useful features of the debugger, and there are different ways to do it. Often, when you try to debug an issue, you are attempting to find out whether variables are storing the values that you expect them to have at a particular time.

2. Expand the `letters` variable to see its properties, which include all the elements that the variable contains.

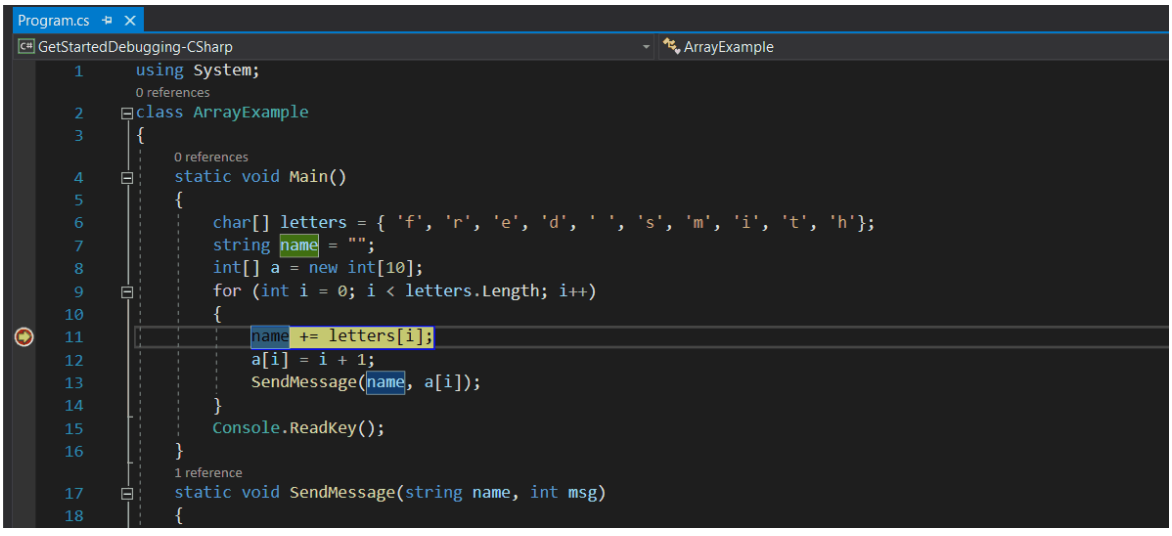


3. Next, hover over the `name` variable, and you see its current value, an empty string.

4. Press **F10** (or choose **Debug > Step Over**) twice to advance to the `SendMessage` method call, and then press **F10** one more time.

   F10 advances the debugger to the next statement without stepping into functions or methods in your app code (the code still executes). By pressing F10 on the `SendMessage` method call, we skipped over the implementation code for `SendMessage` (which maybe we're not interested in right now).

5. Press **F10** (or **Debug > Step Over**) a few times to iterate several times through the `for` loop, pausing again at the breakpoint, and hovering over the `name` variable each time to check its value.



   The value of the variable changes with each iteration of the `for` loop, showing values of `f`, then `fr`,

then `fre`, and so on. To advance the debugger through the loop faster in this scenario, you can press **F5** (or choose **Debug** > **Continue**) instead, which advances you to the breakpoint instead of the next statement.

Often, when debugging, you want a quick way to check property values on variables, to see whether they are storing the values that you expect them to store, and the data tips are a good way to do it.

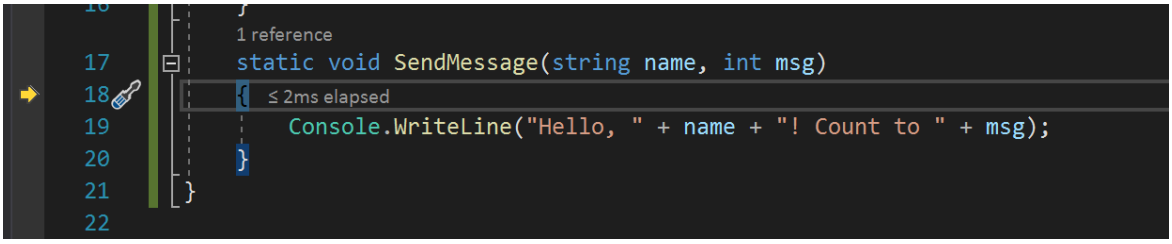
6. While still paused in the `for` loop in the `Main` method, press **F11** (or choose **Debug** > **Step Into**) until you pause at the `SendMessage` method call.

   You should be at this line of code:

   ```
   SendMessage(name, a[i]);
   ```

7. Press **F11** one more time to step into the `SendMessage` method.

   The yellow pointer advances into the `SendMessage` method.

   F11 is the **Step Into** command and advances the app execution one statement at a time. F11 is a good way to examine the execution flow in the most detail. By default, the debugger skips over non-user code (if you want more details, see Just My Code).

   Let's say that you are done examining the `SendMessage` method, and you want to get out of the method but stay in the debugger. You can do this using the **Step Out** command.

8. Press **Shift** + **F11** (or **Debug** > **Step Out**).

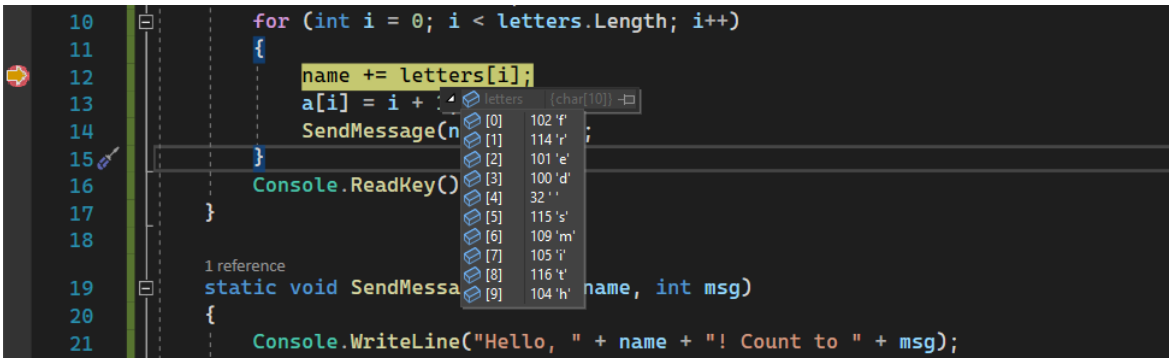
   This command resumes app execution (and advances the debugger) until the current method or function returns.

   You should be back in the `for` loop in the `Main` method, paused at the `SendMessage` method call. For more information on different ways to move through your code, see Navigate code in the debugger.

1. While paused on the `name += letters[i]` statement, hover over the `letters` variable to see a data tip showing the array size and element type, `char[10]`.

   > **NOTE**
   > One of the most useful features of the debugger is its ability to inspect a variable. Often, when you're trying to debug an issue, you're attempting to find out whether variables have values that you expect at a particular time. Viewing data tips is a good way to check that.

2. Expand the `letters` variable to view all its array elements and their values.

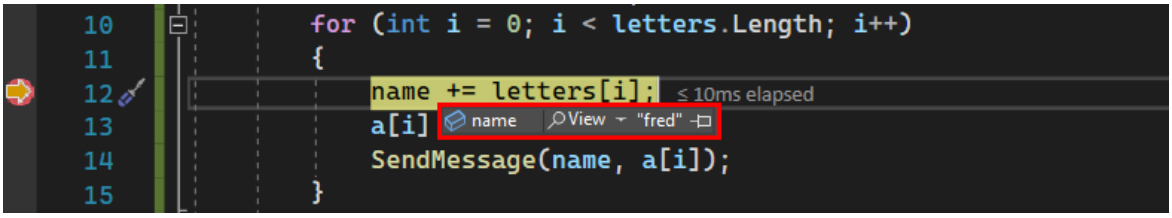


3. Hover over the `name` variable to see its current value, which is an empty string.

4. To advance the debugger to the next statement, select **F10**, or choose the **Step Over** button in the Debug toolbar, or choose **Debug** > **Step Over** from the menu bar. Select **F10** twice more to move past the `SendMessage` method call.

   F10 advances the debugger without stepping into function or methods, although their code still executes. In this way, we skipped debugging the code in the `SendMessage` method, which we're not interested in right now.

5. To iterate through the `for` loop a few times, select **F10** repeatedly. During each loop iteration, pause at the breakpoint, and then hover over the `name` variable to check its value in the data tip.



   The value of the variable changes with each iteration of the `for` loop, showing values of `f`, then `fr`, then `fre`, and so on. To advance the debugger through the loop faster, select **F5** instead, which advances to your breakpoint instead of the next statement.

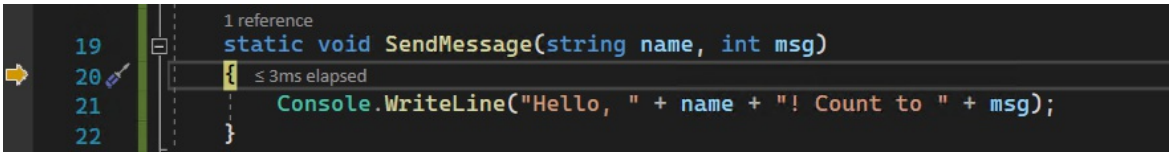
6. While paused in the `for` loop of the `Main` method, select **F11**, or choose the **Step Into** button from the Debug toolbar, or choose **Debug** > **Step Into** from the menu bar, until you reach the `SendMessage` method call.

   The debugger should be paused at this line of code:

   ```
   SendMessage(name, a[i]);
   ```

7. To step into the `SendMessage` method, select **F11** again.

   The yellow pointer advances into the `SendMessage` method.



   F11 helps you examine the execution flow of your code in more depth. To step into a method from a method call, select **F11**. By default, the debugger skips stepping into non-user methods. To learn about debugging non-user code, see Just My Code.

   Once you've finished debugging the `SendMessage` method, you're ready to return to the `for` loop of the `main` method.

8. To leave the `SendMessage` method, select **Shift+F11**, or choose the **Step Out** button in the Debug toolbar, or choose **Debug** > **Step Out** from the menu bar.

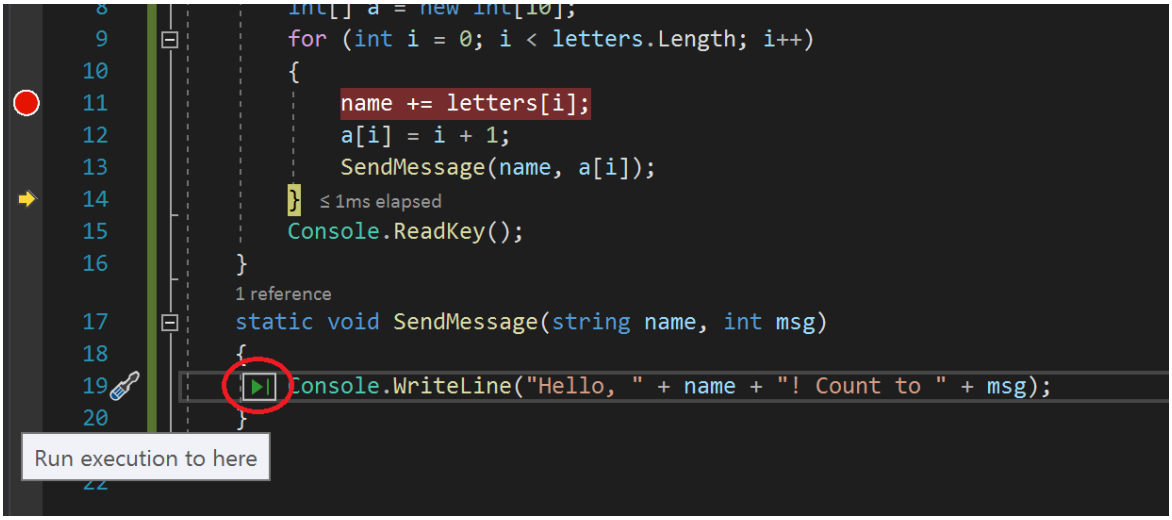
   Step Out resumes app execution and advances the debugger until the current method or function returns.

   You'll see the yellow pointer back in the `for` loop of the `Main` method, paused at the `SendMessage` method call. For more information on different ways to move through your code, see Navigate code in the debugger.

Navigate code using Run to Click

1. Select **F5** to advance to the breakpoint again.

2. In the code editor, scroll down and hover over the `Console.WriteLine` method in the `SendMessage` method until the green **Run to Click** button appears on the left. The tooltip for the button shows "Run execution to here".



> **NOTE**
>
> The **Run to Click** button is new in Visual Studio 2017. (If you don't see the green arrow button, use **F11** in this example instead to advance the debugger to the right place.)

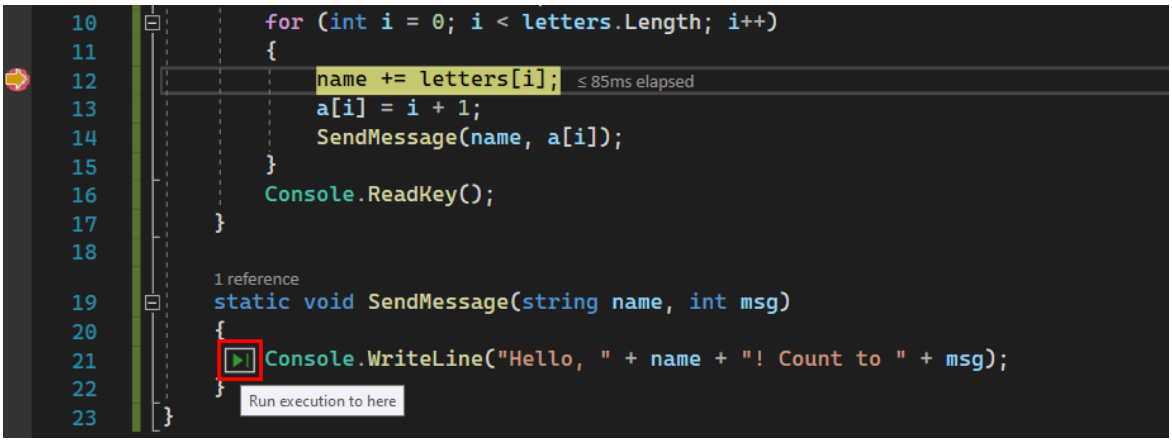
3. Click the **Run to Click** button.

   The debugger advances to the `Console.WriteLine` method.

   Using this button is similar to setting a temporary breakpoint. **Run to Click** is handy for getting around quickly within a visible region of app code (you can click in any open file).

1. Select **F5** to advance to the breakpoint again.

2. In the code editor, hover over the `Console.WriteLine` method call in the `SendMessage` method until the **Run to Click** button appears on the left. The tooltip for the button shows "Run execution to here".



3. Choose the **Run to Click** button. Alternatively, with your cursor at the `Console.WriteLine` statement, select **Ctrl+F10**. Or, right-click the `Console.WriteLine` method call, and choose **Run to Cursor** from the context menu.

   The debugger advances to the `Console.WriteLine` method call.

   Using the **Run to Click** button is similar to setting a temporary breakpoint, and is handy for getting around quickly within a visible region of your app code in an open file.

Restart your app quickly

Click the **Restart** button in the Debug Toolbar (**Ctrl + Shift + F5**).

When you press **Restart**, it saves time versus stopping the app and restarting the debugger. The debugger pauses at the first breakpoint that is hit by executing code.

The debugger stops again at the breakpoint you previously set inside the `for` loop.

To rerun your app from the beginning in the debugger, select **Ctrl+Shift+F5**, or choose the **Restart** button in the Debug toolbar, or choose **Debug** > **Restart** from the menu bar.

Restart stops the debugger and then restarts it, in one step. When the debugger restarts, it will run to the first breakpoint, which is the breakpoint you previously set inside the `for` loop, and then pause.

Inspect variables with the Autos and Locals windows

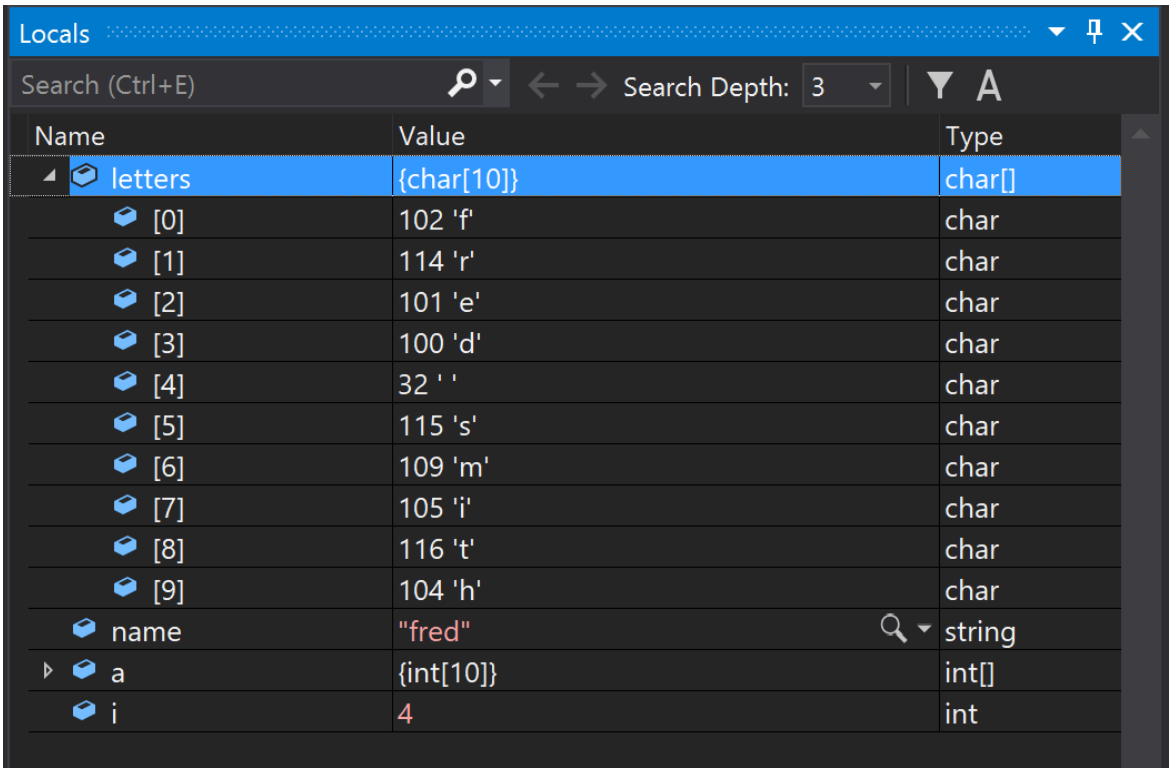
1. Look at the **Autos** window at the bottom of the code editor.

   If it is closed, open it while paused in the debugger by choosing **Debug** > **Windows** > **Autos**.

   In the **Autos** window, you see variables and their current value. The **Autos** window shows all variables used on the current line or the preceding line (Check documentation for language-specific behavior).

2. Next, look at the **Locals** window, in a tab next to the **Autos** window.

3. Expand the `letters` variable to show the elements that it contains.



The **Locals** window shows you the variables that are in the current scope, that is, the current execution context.

The **Autos** and **Locals** windows show variable values while you're debugging. The windows are only available during a debug session. The **Autos** window shows variables used on the current line that the debugger is at and the preceding line. The **Locals** window shows variables defined in the local scope, which is usually the current function or method.

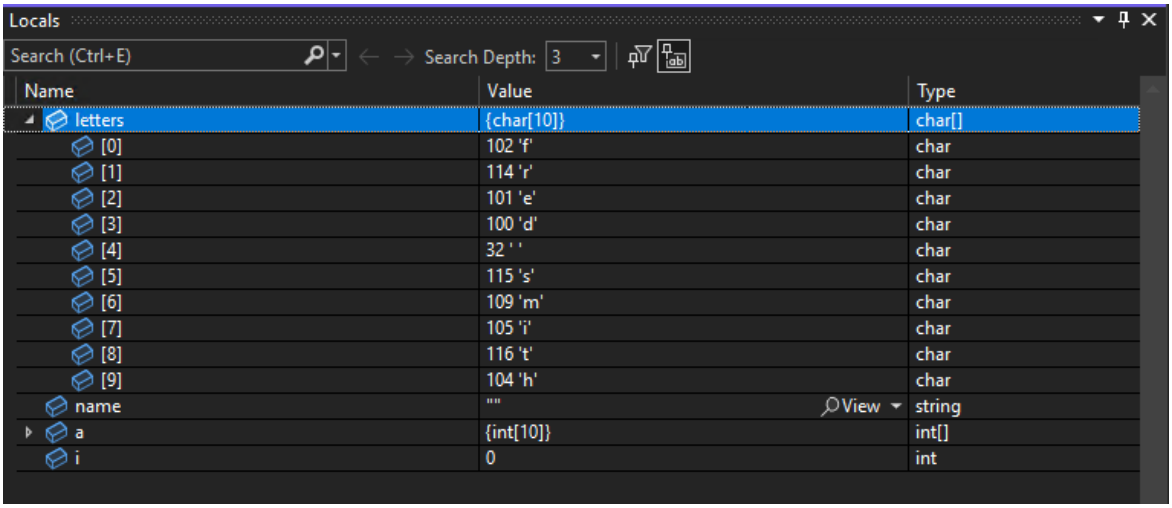
1. While the debugger is paused, view the **Autos** window at the bottom of the code editor.

   If the **Autos** window is closed, select **Ctrl+D, A**, or choose **Debug** > **Windows** > **Autos** from the menu bar.

2. With the debugger still paused, view the **Locals** window, in a tab next to the **Autos** window.

   If the **Locals** window is closed, select **Ctrl+D, L**, or choose **Debug** > **Windows** > **Locals**.

3. In the **Locals** window, expand the `letters` variable to see its array elements and their values.



For more about the **Autos** and **Locals** windows, see Inspect variables in the Autos and Locals windows.

Set a watch

1. In the main code editor window, right-click the `name` variable and choose **Add Watch**.

   The **Watch** window opens at the bottom of the code editor. You can use a **Watch** window to specify a variable (or an expression) that you want to keep an eye on.

   Now, you have a watch set on the `name` variable, and you can see its value change as you move through the debugger. Unlike the other variable windows, the **Watch** window always shows the variables that you are watching (they're grayed out when out of scope).

You can specify a variable, or an expression, that you want to keep an eye on as you step through code—by adding it to the **Watch** window.

1. While the debugger is paused, right-click the `name` variable and choose **Add Watch**.

   The **Watch** window opens by default at the bottom of the code editor.

2. Now that you've set a watch on the `name` variable, step through your code to see the value of the `name` variable change with each `for` loop iteration.

   Unlike the other variable windows, the **Watch** window always shows the variables that you're watching, though they'll be grayed out when they're out of scope.

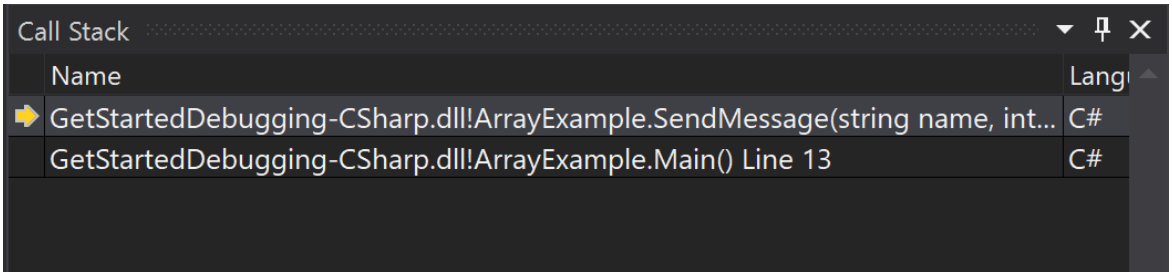
For more about the **Watch** window, see Watch variables with Watch windows.

Examine the call stack

1. While paused in the `for` loop, click the **Call Stack** window, which is by default open in the lower right pane.

   If it is closed, open it while paused in the debugger by choosing **Debug** > **Windows** > **Call Stack**.

2. Click **F11** a few times until you see the debugger pause in the `SendMessage` method. Look at the **Call Stack** window.

   The **Call Stack** window shows the order in which methods and functions are getting called. The top line shows the current function (the `SendMessage` method in this app). The second line shows that `SendMessage` was called from the `Main` method, and so on.

   > **NOTE**
   > The **Call Stack** window is similar to the Debug perspective in some IDEs like Eclipse.

   The call stack is a good way to examine and understand the execution flow of an app.

   You can double-click a line of code to go look at that source code and that also changes the current scope being inspected by the debugger. This action does not advance the debugger.

   You can also use right-click menus from the **Call Stack** window to do other things. For example, you can

insert breakpoints into specified functions, advance the debugger using **Run to Cursor**, and go examine source code. For more information, see How to: Examine the Call Stack.

The **Call Stack** can help you understand the execution flow of your app, by showing the order in which methods and functions are getting called.

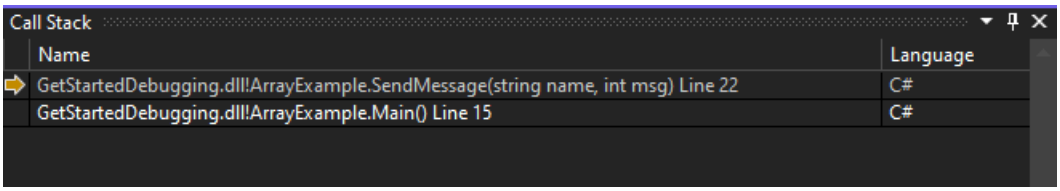
1. While the debugger is paused in the `for` loop, view the **Call Stack** window, which opens by default in the lower right pane of the code editor.

   If the **Call Stack** window is closed, select **Ctrl+D, C**, or choose **Debug** > **Windows** > **Call Stack** from the menu bar.

   In the **Call Stack** window, you'll see the yellow pointer at the current `Main` method.

2. Select **F11** a few times until you see the debugger pause in the `SendMessage` method.

   The top line of the **Call Stack** window shows the current function, which is the `SendMessage` method. The second line shows that the `SendMessage` method was called from the `Main` method.



> **NOTE**
> The **Call Stack** window is similar to the Debug perspective in some IDEs, like Eclipse.

   In the **Call Stack** window, you can double-click a line of code to go to that source code, which changes the current scope being inspected by the debugger. This action does not advance the debugger.

   You can also use right-click menus from the **Call Stack** window to do other things. For example, you can insert breakpoints into specified functions, advance the debugger by using **Run to Cursor**, or go to source code.

For more about the **Call Stack**, see How to: Examine the Call Stack.

Change the execution flow

1. Press **F11** twice to run the `Console.WriteLine` method.

2. With the debugger paused in the `SendMessage` method call, use the mouse to grab the yellow arrow (the execution pointer) on the left and move the yellow arrow up one line, back to `Console.WriteLine`.

3. Press **F11**.

   The debugger reruns the `Console.WriteLine` method (you see this in the console window output).

   By changing the execution flow, you can do things like test different code execution paths or rerun code without restarting the debugger.

> **WARNING**
> Often you need to be careful with this feature, and you see a warning in the tooltip. You may see other warnings, too. Moving the pointer cannot revert your application to an earlier app state.

4. Press **F5** to continue running the app.

   Congratulations on completing this tutorial!

You can move the execution pointer to change the flow of your app while debugging.

1. With the debugger paused at the `SendMessage` method call in the `for` loop, select **F11** three times to step into the `SendMessage` method and to move past the `Console.WriteLine` method after executing it.

   The debugger is now paused at the final closing brace of the `SendMessage` method.

2. Use the mouse to grab the yellow arrow or execution pointer (in the left margin), and then drag the pointer up one line.

   The debugger is now back on the `Console.WriteLine` statement.

3. Select **F11**.

   The debugger reruns the `Console.WriteLine` method, and you'll see a duplicate line in the console window output.

4. Select **F5** to continue running the app.

By changing the execution flow, you can do things like test different code execution paths or rerun code without restarting the debugger.

> **WARNING**
> Use this feature with care. You'll see a warning in the tooltip of the execution pointer about the possibility of unintended consequences. You might see other warnings, too. Moving the execution pointer can't revert your application to an earlier state.

For more about the changing the execution flow, see Move the pointer to change the execution flow.

Congratulations on completing this tutorial!

Next steps

In this tutorial, you've learned how to start the debugger, step through code, and inspect variables. You might want to get a high-level look at debugger features along with links to more information.

First look at the debugger

Get started with unit testing

10/28/2021 • 6 minutes to read • Edit Online

Use Visual Studio to define and run unit tests to maintain code health, ensure code coverage, and find errors and faults before your customers do. Run your unit tests frequently to make sure your code is working properly.

In this article, the code uses C# and C++, illustrations are in C#, but the concepts and features apply to .NET languages, C++, Python, JavaScript, and TypeScript.

Create unit tests

This section describes how to create a unit test project.

1. Open the project that you want to test in Visual Studio.

   For the purposes of demonstrating an example unit test, this article tests a simple "Hello World" C# or C++ Console project named **HelloWorld** (**HelloWorldCore** in C#). The sample code for such a project is as follows:

   - .NET
   - C++

   ```
   namespace HelloWorldCore

      public class Program
      {
         public static void Main()
         {
            Console.WriteLine("Hello World!");
         }
      }
   ```

2. In **Solution Explorer**, select the solution node. Then, from the top menu bar, select **File** > **Add** > **New Project**.

3. In the new project dialog box, find the unit test project to use.

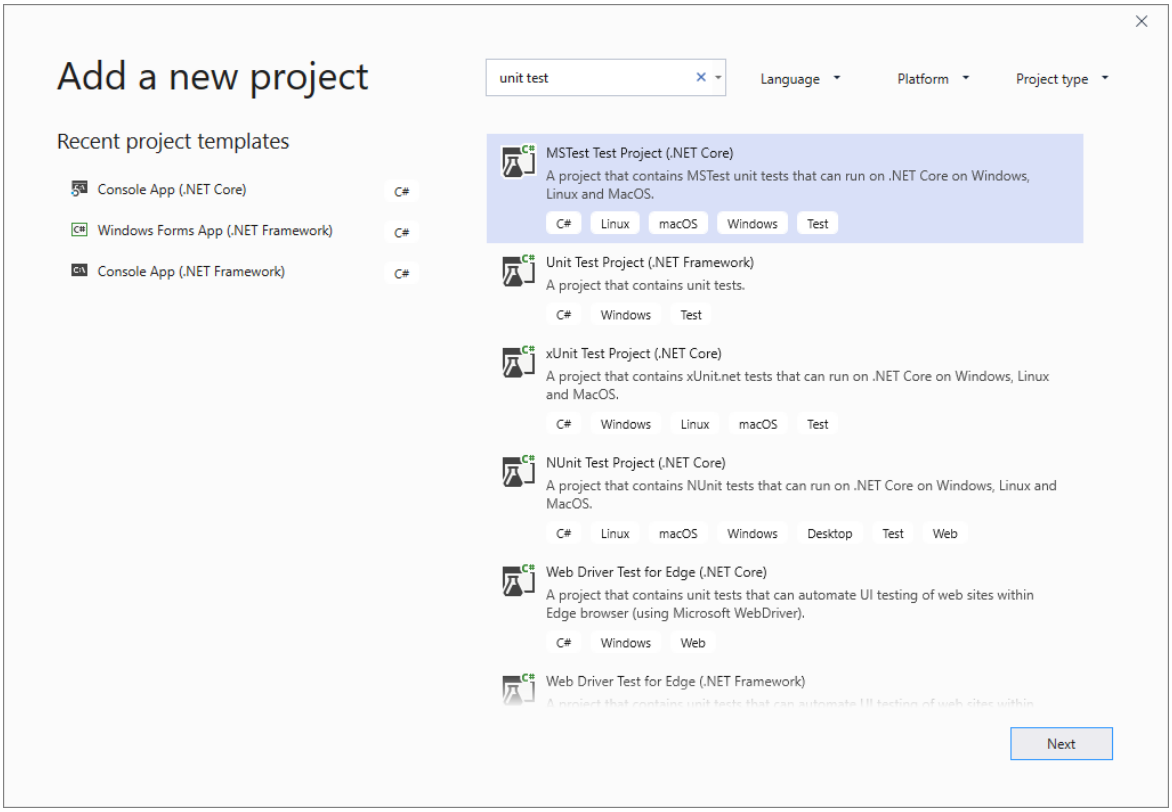
   Type **test** in the search box to find a unit test project template for the test framework you want to use, such as MSTest (C#) or the **Native Unit Test** project (C++), and select it.

   Expand the **Installed** node, choose the language that you want to use for your test project, and then choose **Test**.

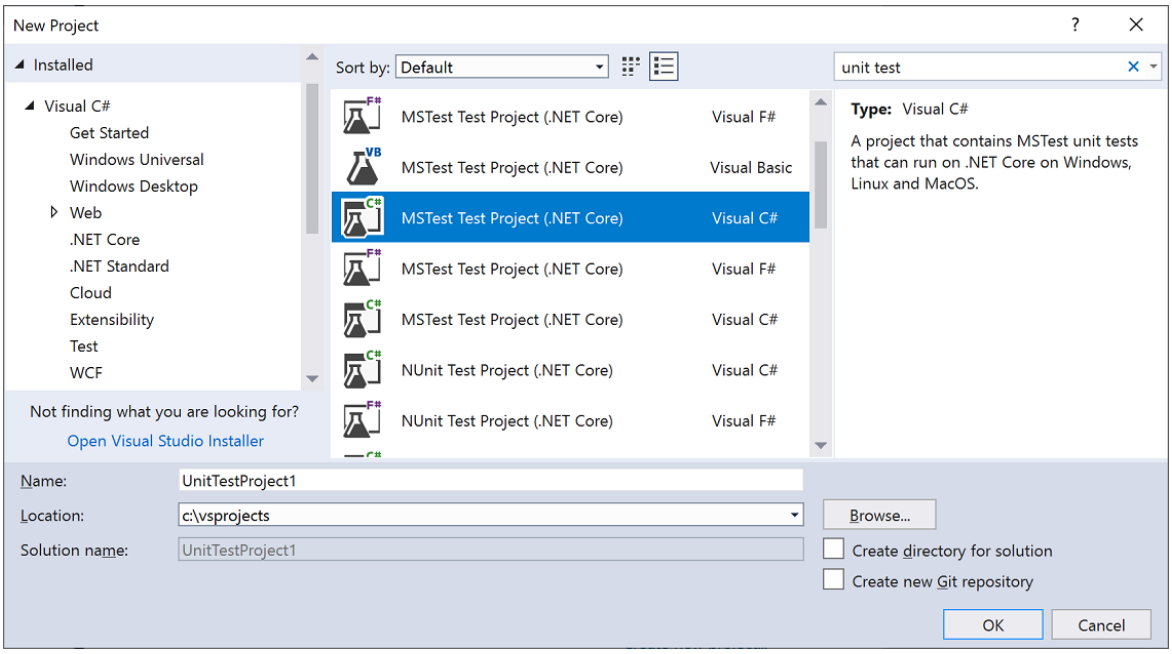
   Starting in Visual Studio 2017 version 14.8, the .NET languages include built-in templates for NUnit and xUnit. For C++, in this example select the **Native Unit Test** project, which uses Microsoft Native Unit Test Framework. (To use a different C++ test framework, see Writing unit tests for C/C++). For Python, see Set up unit testing in Python code to set up your test project.

> **TIP**
>
> For C# only, you can create unit test projects from code using a faster method. For more information, see Create unit test projects and test methods. To use this method with .NET Core or .NET Standard, Visual Studio 2019 is required.

The following illustration shows an MSTest unit test, which is supported in .NET.

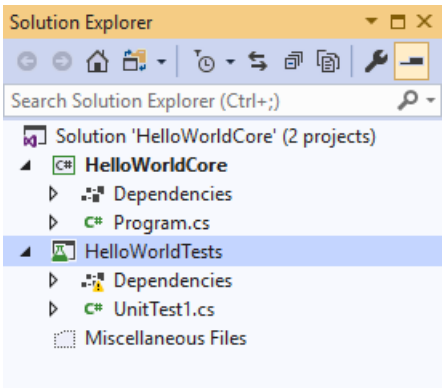


Click **Next**, choose a name for the test project, and then click **Create**.



Choose a name for the test project, such as HelloWorldTests, and then click **OK**.

The project is added to your solution.



4. In the unit test project, add a reference to the project you want to test by right-clicking on **References** or **Dependencies** and then choosing **Add Reference** or **Add Project Reference**.

5. Select the project that contains the code you'll test and click **OK**.

6. Add code to the unit test method.

   For example, you might use the following code by selecting the correct documentation tab that matches your test framework: MSTest, NUnit, or xUnit (supported on .NET only), or C++ Microsoft Native Unit Test Framework.

   - MSTest
   - NUnit
   - xUnit
   - Microsoft Native Unit Test Framework

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

namespace HelloWorldTests
{
   [TestClass]
   public class UnitTest1
   {
      private const string Expected = "Hello World!";
      [TestMethod]
      public void TestMethod1()
      {
         using (var sw = new StringWriter())
         {
            Console.SetOut(sw);
            HelloWorldCore.Program.Main();

            var result = sw.ToString().Trim();
            Assert.AreEqual(Expected, result);
         }
      }
   }
}
```

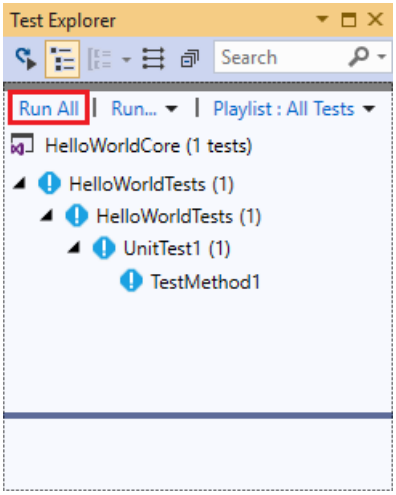
Run unit tests

1. Open Test Explorer.

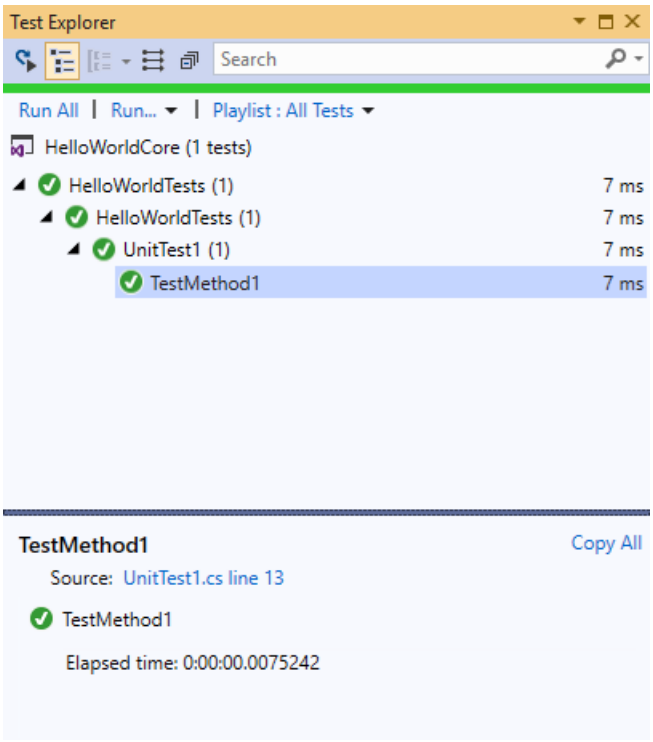
   To open Test Explorer, choose **Test** > **Test Explorer** from the top menu bar (or press **Ctrl** + **E**, **T**).

   To open Test Explorer, choose **Test** > **Windows** > **Test Explorer** from the top menu bar.

2. Run your unit tests by clicking **Run All** (or press **Ctrl** + **R**, **V**).



   After the tests have completed, a green check mark indicates that a test passed. A red "x" icon indicates that a test failed.



TIP

You can use Test Explorer to run unit tests from the built-in test framework (MSTest) or from third-party test frameworks. You can group tests into categories, filter the test list, and create, save, and run playlists of tests. You can also debug tests and analyze test performance and code coverage.

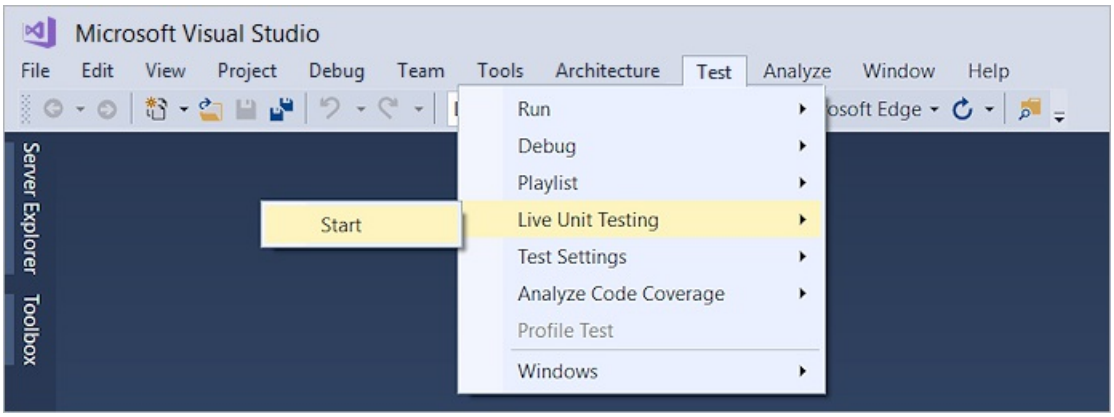
View live unit test results (Visual Studio Enterprise)

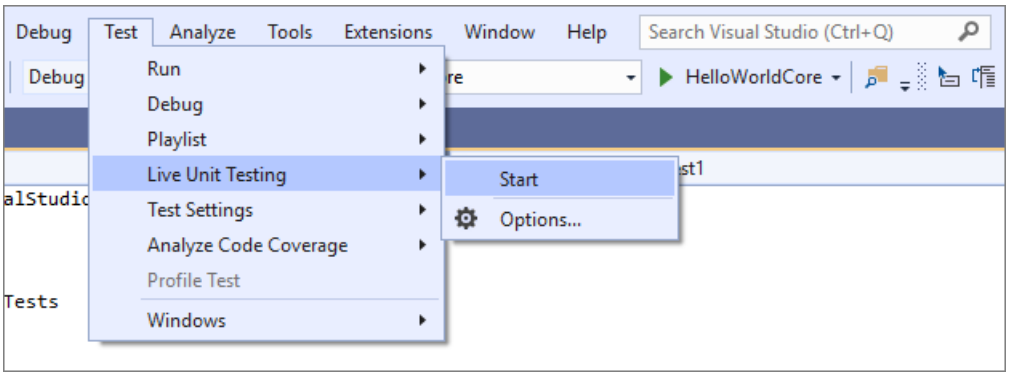
If you are using the MSTest, xUnit, or NUnit testing framework in Visual Studio 2017 or later, you can see live results of your unit tests.

NOTE

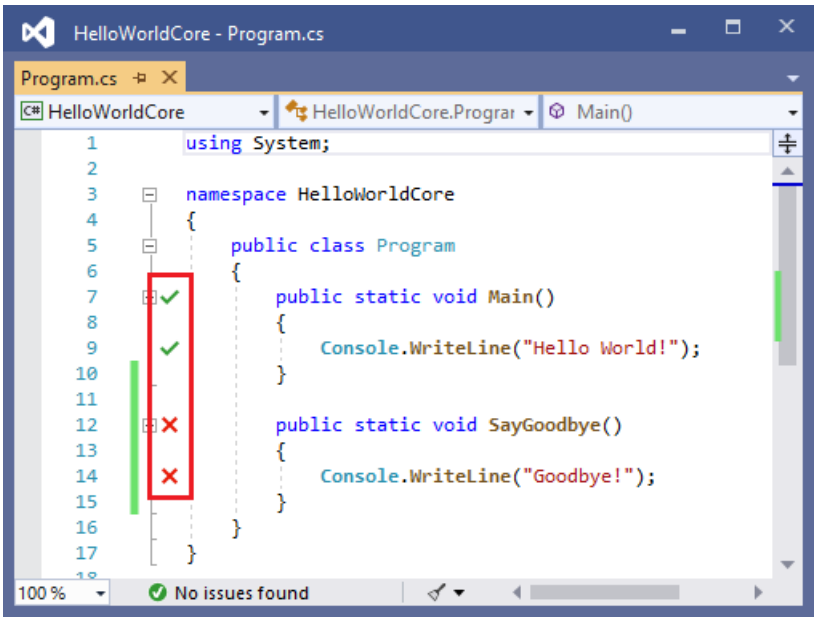
To follow these steps, Visual Studio Enterprise is required, along with .NET code and one of the following test frameworks: MSTest, xUnit, or NUnit.

1. Turn live unit testing from the **Test** menu by choosing **Test** > **Live Unit Testing** > **Start**.





2. View the results of the tests within the code editor window as you write and edit code.



3. Click a test result indicator to see more information, such as the names of the tests that cover that method.

For more information about live unit testing, see Live unit testing.

Use a third-party test framework

You can run unit tests in Visual Studio by using third-party test frameworks such as NUnit, Boost, or Google C++ Testing Framework, depending on your programming language. To use a third-party framework:

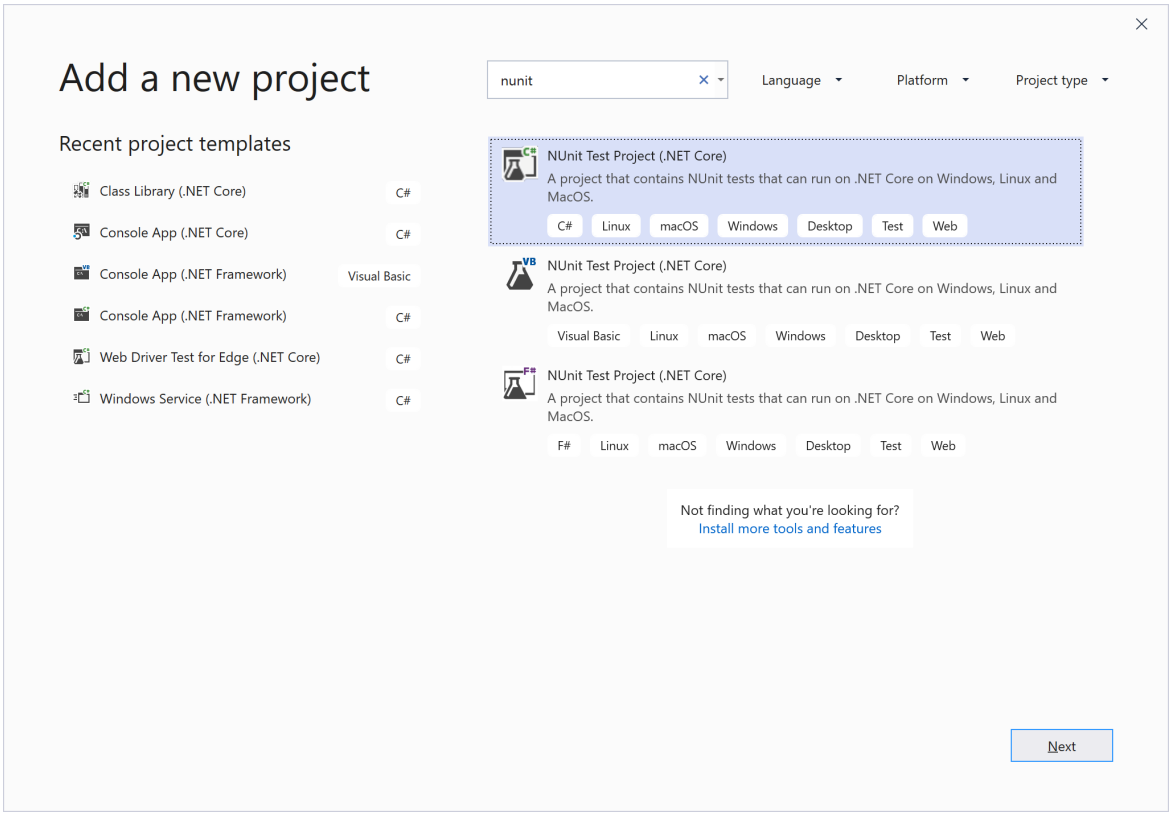
- Use the **NuGet Package Manager** to install the NuGet package for the framework of your choice.
- (.NET) Starting in Visual Studio 2017 version 14.6, Visual Studio includes pre-configured test project templates for NUnit and xUnit test frameworks. The templates also include the necessary NuGet packages to enable support.

- (C++) In Visual Studio 2017 and later versions, some frameworks like Google C++ Testing Framework are already included. For more information, see Write unit tests for C/C++ in Visual Studio.

To add a unit test project:

1. Open the solution that contains the code you want to test.
2. Right-click on the solution in **Solution Explorer** and choose **Add** > **New Project**.
3. Select a unit test project template.

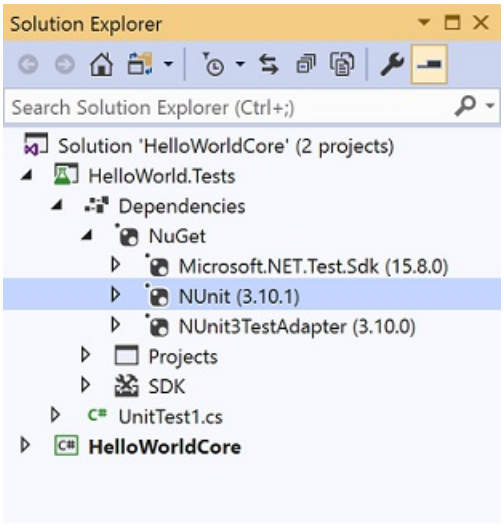
   In this example, select NUnit



   Click **Next**, name the project, and then click **Create**.

   Name the project, and then click **OK** to create it.

   The project template includes NuGet references to NUnit and NUnit3TestAdapter.

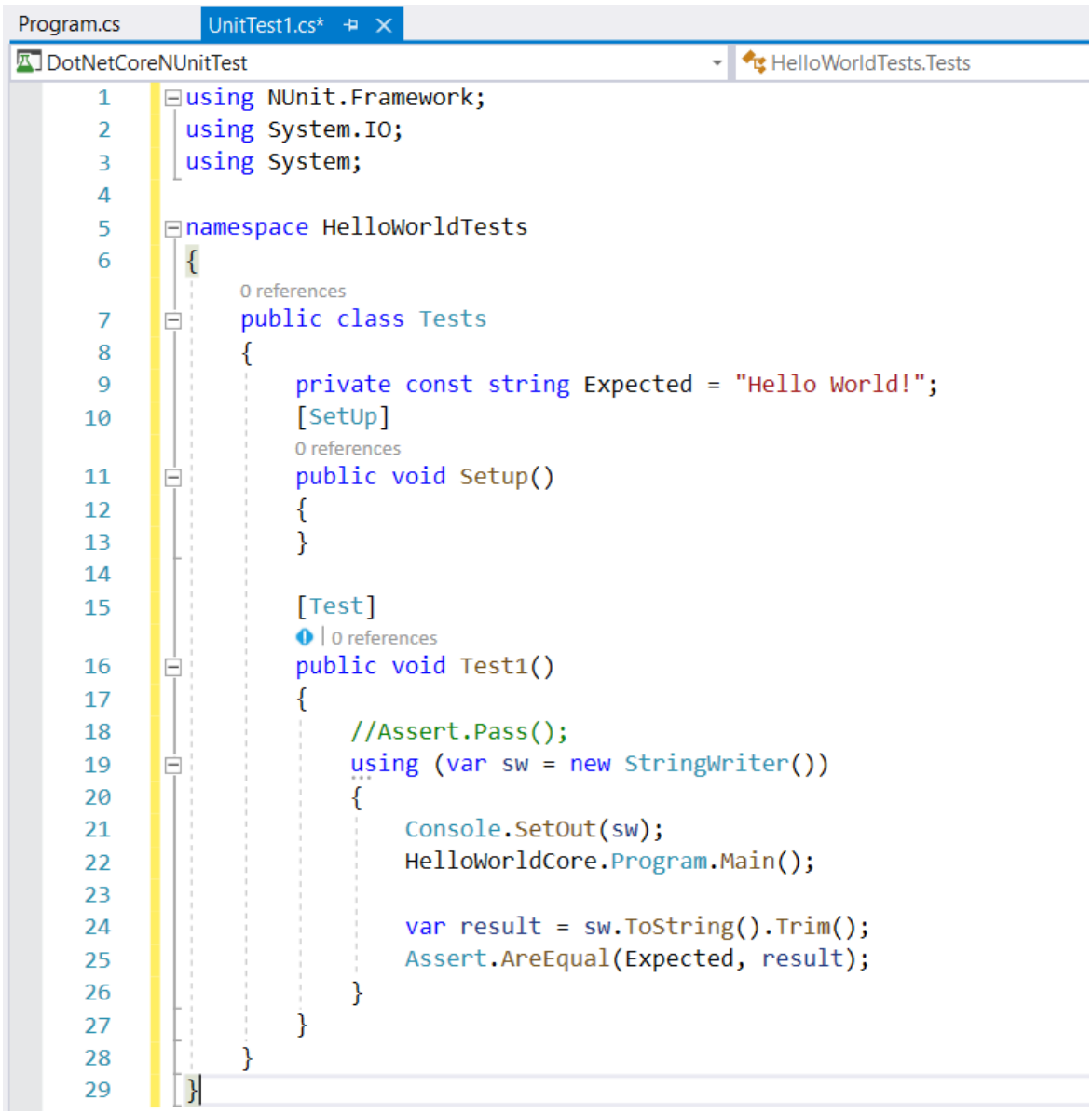


4. Add a reference from the test project to the project that contains the code you want to test.

   Right-click on the project in **Solution Explorer**, and then select **Add** > **Reference**. (You can also add a

reference from the right-click menu of the **References** or **Dependencies** node.)

5. Add code to your test method.



6. Run the test from **Test Explorer** or by right-clicking on the test code and choosing **Run Test(s)** (or **Ctrl** + **R**, **T**).

Next steps

Unit test basics

Create and run unit tests for managed code

Write unit tests for C/C++

Create a database and add tables in Visual Studio

10/28/2021 • 5 minutes to read • Edit Online

You can use Visual Studio to create and update a local database file in SQL Server Express LocalDB. You can also create a database by executing Transact-SQL statements in the **SQL Server Object Explorer** tool window in Visual Studio. In this topic, we'll create an *.mdf* file and add tables and keys by using the Table Designer.

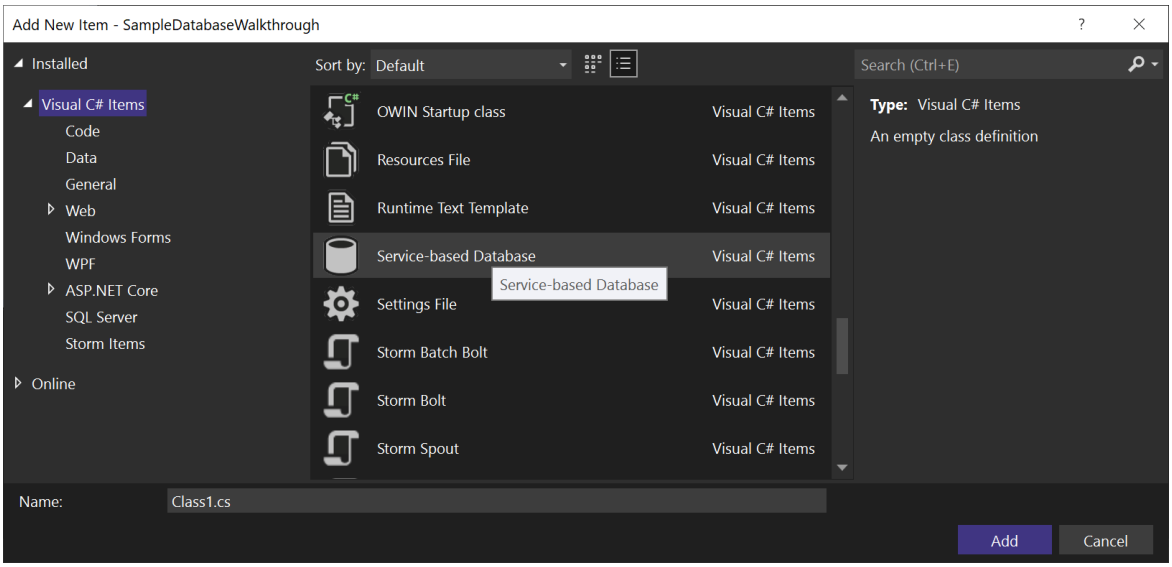
Prerequisites

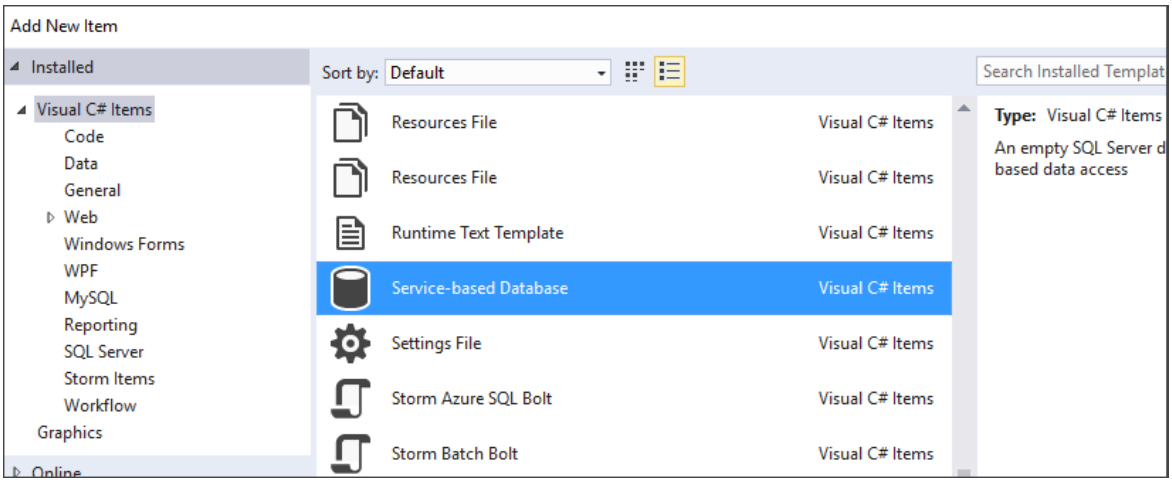
To complete this walkthrough, you'll need the **.NET desktop development** and **Data storage and processing** workloads installed in Visual Studio. To install them, open **Visual Studio Installer** and choose **Modify** (or **More** > **Modify**) next to the version of Visual Studio you want to modify.

> **NOTE**
>
> The procedures in this article apply only to .NET Framework Windows Forms projects, not to .NET Core Windows Forms projects.

Create a project and a local database file

1. Create a new **Windows Forms App (.NET Framework)** project and name it **SampleDatabaseWalkthrough**.
2. On the menu bar, select **Project** > **Add New Item**.
3. In the list of item templates, scroll down and select **Service-based Database**.



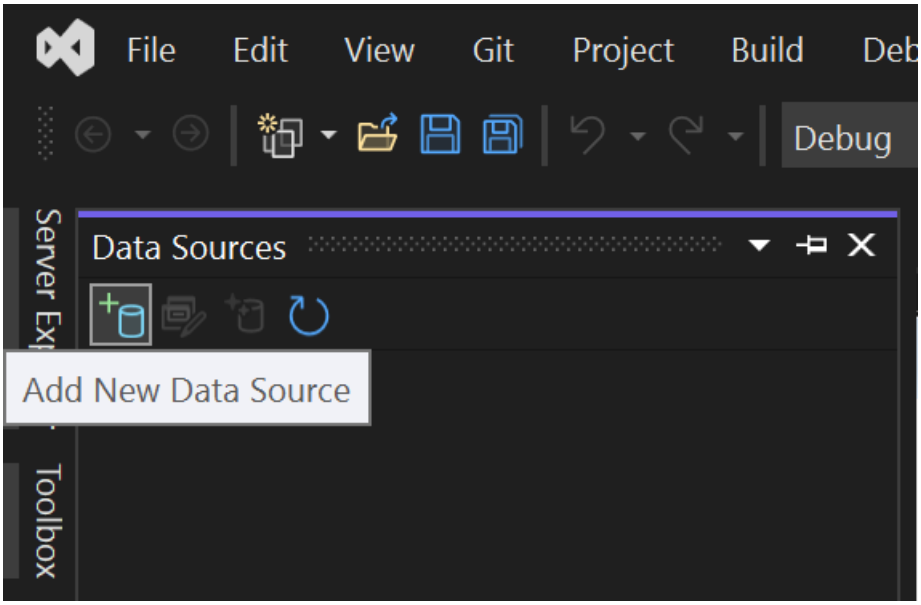


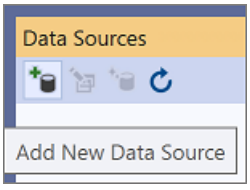
4. Name the database **SampleDatabase**, and then click **Add**.

Add a data source

1. If the **Data Sources** window isn't open, open it by pressing **Shift**+**Alt**+**D** or selecting **View** > **Other Windows** > **Data Sources** on the menu bar.

2. In the **Data Sources** window, select **Add New Data Source**.





The **Data Source Configuration Wizard** opens.

3. On the **Choose a Data Source Type** page, choose **Database** and then choose **Next**.

4. On the **Choose a Database Model** page, choose **Next** to accept the default (Dataset).

5. On the **Choose Your Data Connection** page, select the **SampleDatabase.mdf** file in the drop-down list, and then choose **Next**.

6. On the **Save the Connection String to the Application Configuration File** page, choose **Next**.

7. On the **Choose your Database Objects** page, you'll see a message that says the database doesn't contain any objects. Choose **Finish**.

View properties of the data connection

You can view the connection string for the *SampleDatabase.mdf* file by opening the Properties window of the data connection:

- Select **View** > **SQL Server Object Explorer** to open the **SQL Server Object Explorer** window. Expand **(localdb)\MSSQLLocalDB** > **Databases**, and then right-click on *SampleDatabase.mdf* and select **Properties**.
- Alternatively, you can select **View** > **Server Explorer**, if that window isn't already open. Open the Properties window by expanding the **Data Connections** node, right-clicking on *SampleDatabase.mdf*, and then selecting **Properties**.

> **TIP**
> If you can't expand the Data Connections node, or the SampleDatabase.mdf connection is not listed, select the **Connect to Database** button in the Server Explorer toolbar. In the **Add Connection** dialog box, make sure that **Microsoft SQL Server Database File** is selected under **Data source**, and then browse to and select the SampleDatabase.mdf file. Finish adding the connection by selecting **OK**.

Create tables and keys by using Table Designer

In this section, you'll create two tables, a primary key in each table, and a few rows of sample data. You'll also create a foreign key to specify how records in one table correspond to records in the other table.

Create the Customers table

1. In **Server Explorer**, expand the **Data Connections** node, and then expand the **SampleDatabase.mdf** node.

   If you can't expand the Data Connections node, or the SampleDatabase.mdf connection is not listed, select the **Connect to Database** button in the Server Explorer toolbar. In the **Add Connection** dialog box, make sure that **Microsoft SQL Server Database File** is selected under **Data source**, and then browse to and select the SampleDatabase.mdf file. Finish adding the connection by selecting **OK**.

2. Right-click on **Tables** and select **Add New Table**.

   The Table Designer opens and shows a grid with one default row, which represents a single column in the table that you're creating. By adding rows to the grid, you'll add columns in the table.

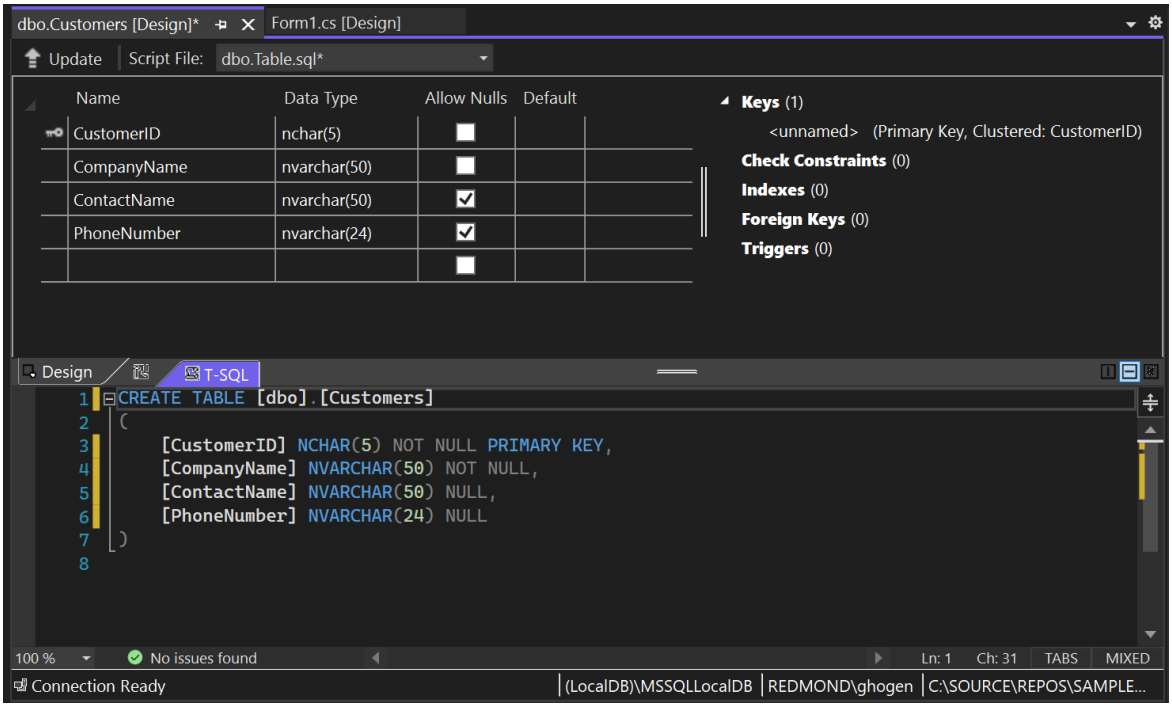
3. In the grid, add a row for each of the following entries:

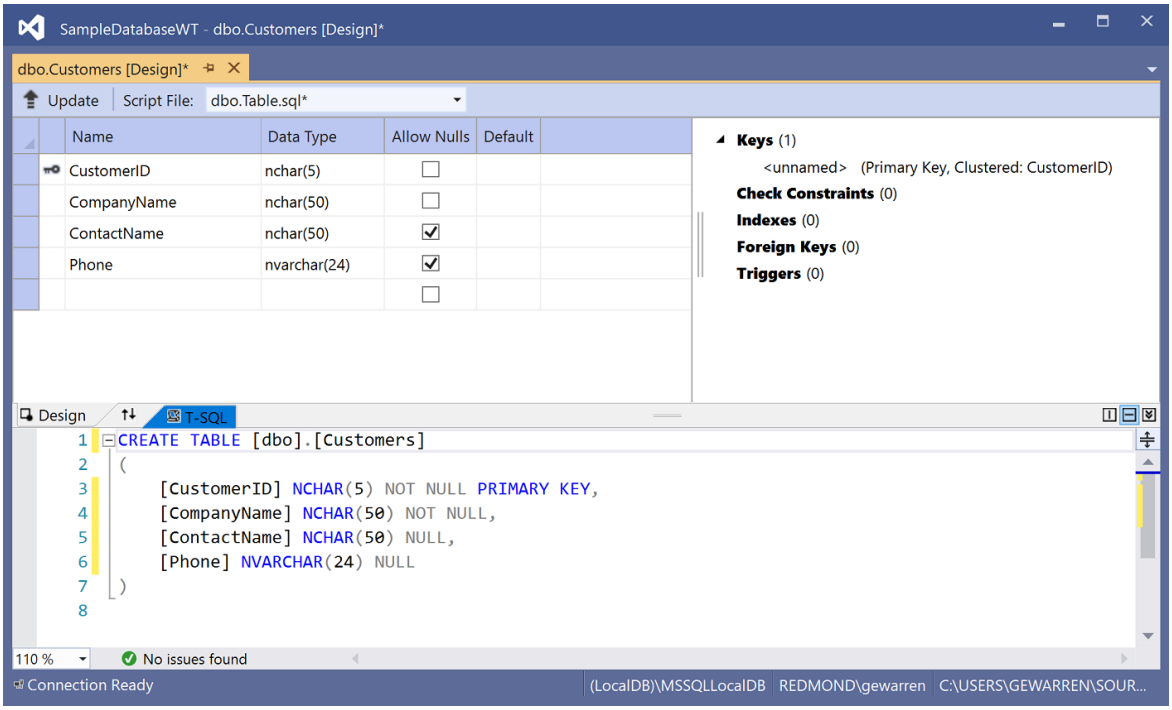
COLUMN NAME	DATA TYPE	ALLOW NULLS
`CustomerID`	`nchar(5)`	False (cleared)
`CompanyName`	`nvarchar(50)`	False (cleared)
`ContactName`	`nvarchar (50)`	True (selected)
`Phone`	`nvarchar (24)`	True (selected)

4. Right-click on the `CustomerID` row, and then select **Set Primary Key**.
5. Right-click on the default row (`Id`), and then select **Delete**.
6. Name the Customers table by updating the first line in the script pane to match the following sample:

```
CREATE TABLE [dbo].[Customers]
```

You should see something like this:





7. In the upper-left corner of **Table Designer**, select **Update**, or press **Shift+Alt+U**.
8. In the **Preview Database Updates** dialog box, select **Update Database**.

   The Customers table is created in the local database file.

Create the Orders table

1. Add another table, and then add a row for each entry in the following table:

COLUMN NAME	DATA TYPE	ALLOW NULLS
`OrderID`	`int`	False (cleared)

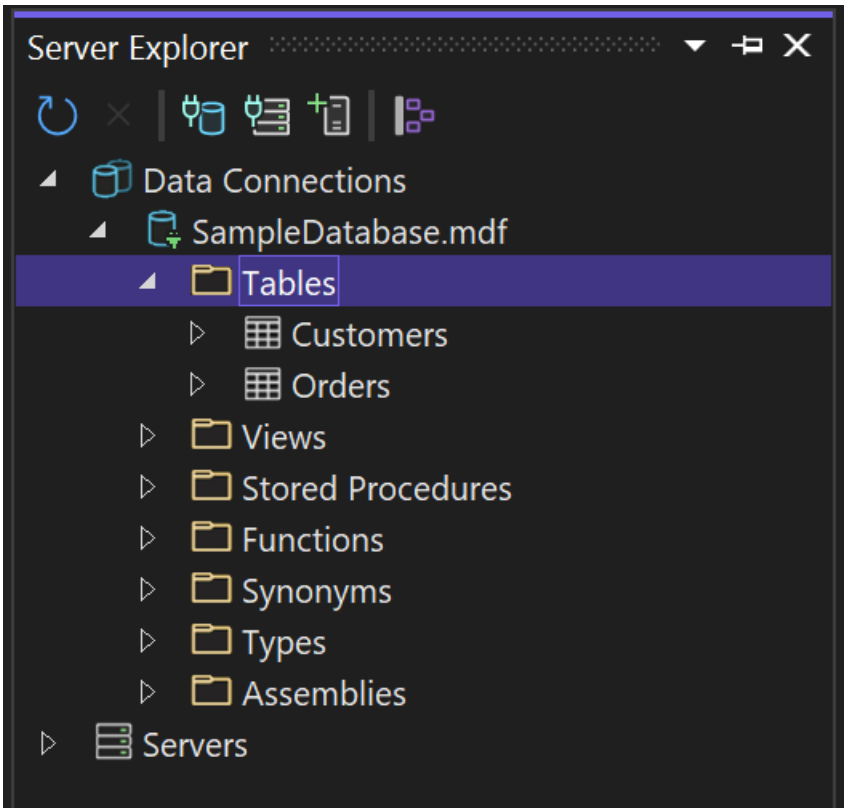
COLUMN NAME	DATA TYPE	ALLOW NULLS
`CustomerID`	`nchar(5)`	False (cleared)
`OrderDate`	`datetime`	True (selected)
`OrderQuantity`	`int`	True (selected)

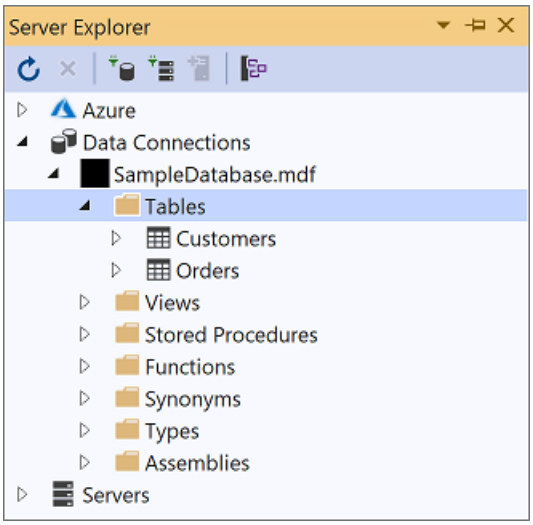
2. Set **OrderID** as the primary key, and then delete the default row.
3. Name the Orders table by updating the first line in the script pane to match the following sample:

```
CREATE TABLE [dbo].[Orders]
```

4. In the upper-left corner of the **Table Designer**, select **Update**, or press **Shift+Alt+U**..
5. In the **Preview Database Updates** dialog box, select **Update Database**.

   The Orders table is created in the local database file. If you expand the **Tables** node in Server Explorer, you see the two tables:

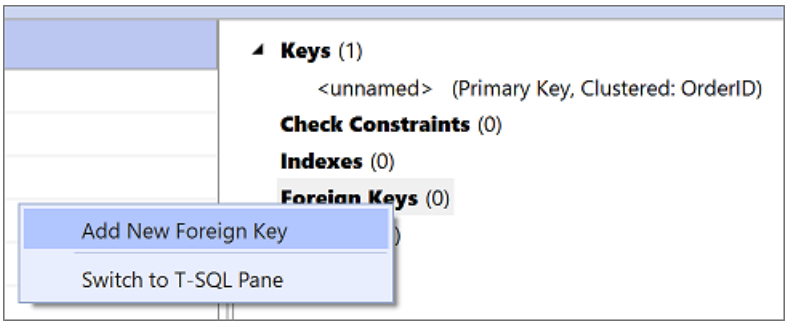




If you don't see it, hit the **Refresh** toolbar button.

Create a foreign key

1. In the context pane on the right side of the Table Designer grid for the Orders table, right-click on **Foreign Keys** and select **Add New Foreign Key**.



2. In the text box that appears, replace the text **ToTable** with **Customers**.
3. In the T-SQL pane, update the last line to match the following sample:

```
CONSTRAINT [FK_Orders_Customers] FOREIGN KEY ([CustomerID]) REFERENCES [Customers]([CustomerID])
```

4. In the upper-left corner of the **Table Designer**, select **Update** (**Shift**+**Alt**+**U**).
5. In the **Preview Database Updates** dialog box, select **Update Database**.

   The foreign key is created.

Populate the tables with data

1. In **Server Explorer** or **SQL Server Object Explorer**, expand the node for the sample database.
2. Open the shortcut menu for the **Tables** node, select **Refresh**, and then expand the **Tables** node.
3. Open the shortcut menu for the Customers table, and then select **View Data**.
4. Add whatever data you want for some customers.

   You can specify any five characters you want as the customer IDs, but choose at least one that you can remember for use later in this procedure.
5. Open the shortcut menu for the Orders table, and then select **Show Table Data**.
6. Add data for some orders. As you enter each row, it is saved in the database.

> **IMPORTANT**
>
> Make sure that all order IDs and order quantities are integers and that each customer ID matches a value that you specified in the **CustomerID** column of the Customers table.

Congratulations! You now know how to create tables, link them with a foreign key, and add data.

See also

- Accessing data in Visual Studio