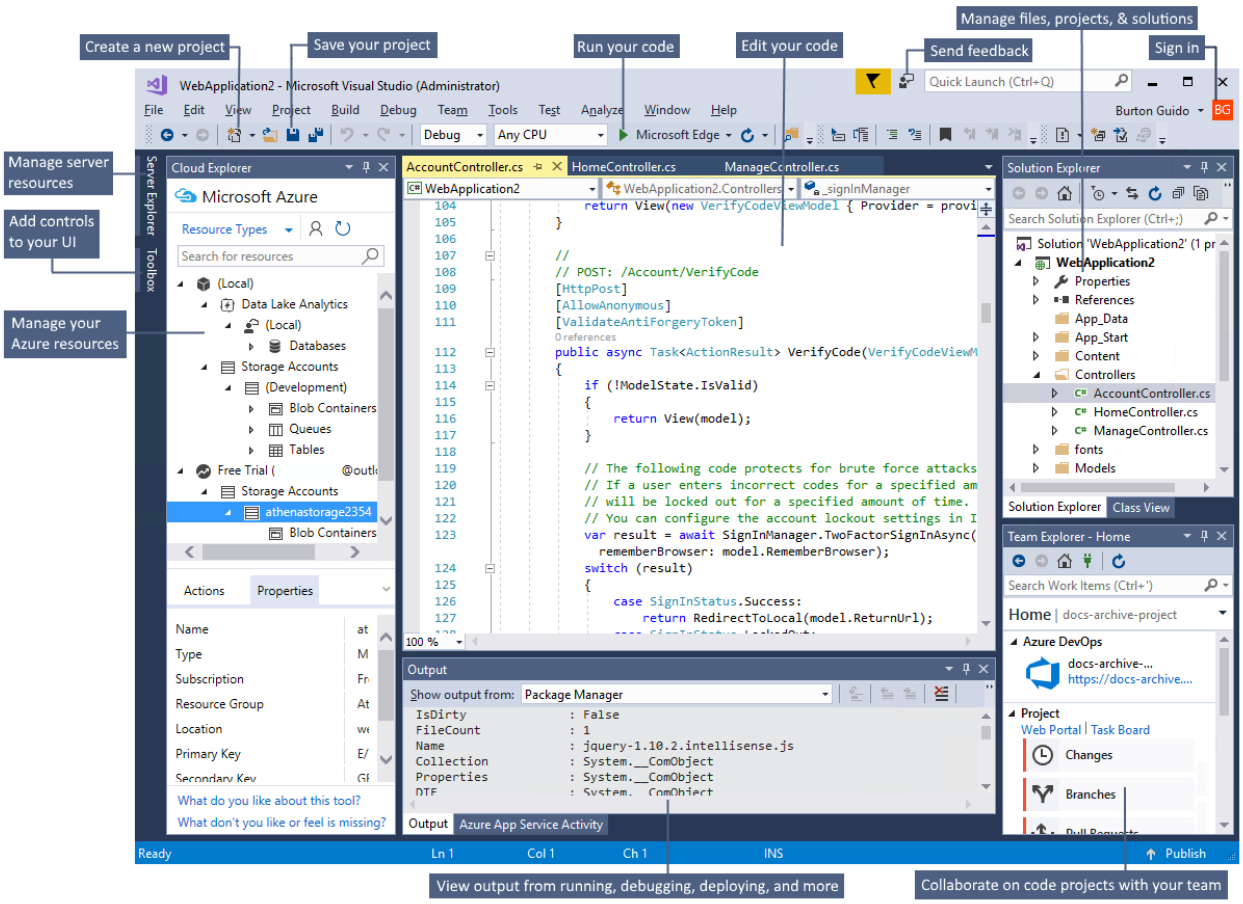
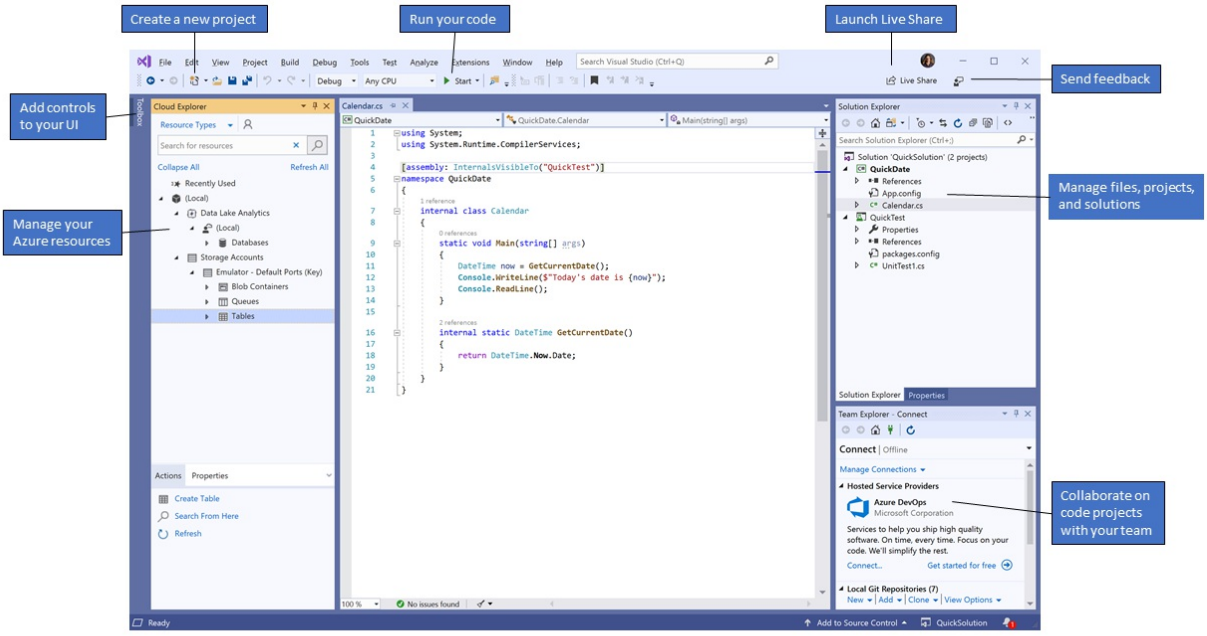
Contents

Welcome to the Visual Studio IDE | C#

1/1/2020 • 14 minutes to read • Edit Online

The Visual Studio *integrated development environment* is a creative launching pad that you can use to edit, debug, and build code, and then publish an app. An integrated development environment (IDE) is a feature-rich program that can be used for many aspects of software development. 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 ease the software development process.





This image shows Visual Studio with an open project and several key tool windows you'll likely use:

- Solution Explorer (top right) lets you view, navigate, and manage your code files. **Solution Explorer** can help organize your code by grouping the files into solutions and projects.
- The editor window (center), where you'll likely spend a majority of your time, displays file contents. This is where 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.
- Team Explorer (bottom right) lets you track work items and share code with others using version control technologies such as Git and Team Foundation Version Control (TFVC).

Editions

Visual Studio is available for Windows and Mac. Visual Studio for Mac has many of the same features as Visual Studio 2017, and is optimized for developing cross-platform and mobile apps. This article focuses on the Windows version of Visual Studio 2017.

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.

Visual Studio is available for Windows and Mac. Visual Studio for Mac has many of the same features as Visual Studio 2019, and is optimized for developing cross-platform and mobile apps. This article focuses on the Windows version of Visual Studio 2019.

There are three editions of Visual Studio 2019: Community, Professional, and Enterprise. See Compare Visual Studio editions to learn about which features are supported in each edition.

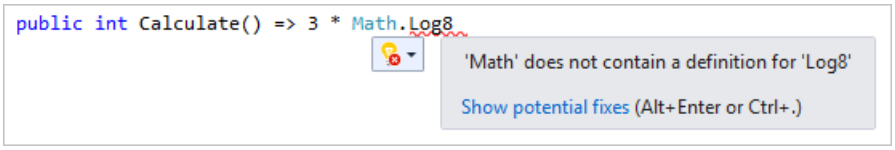
Popular productivity features

Some of the popular features in Visual Studio that help you to be more productive as you develop software include:

- Squiggles and Quick Actions

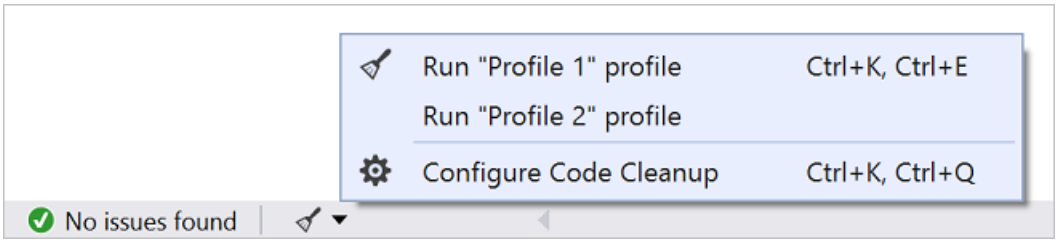
  Squiggles are wavy underlines that alert you to errors or potential problems in your code as you type. These

visual clues enable you to fix problems immediately without waiting for the error to be discovered during build or when you run the program. If you hover over a squiggle, you see additional information about the error. A light bulb may also appear in the left margin with actions, known as Quick Actions, to fix the error.



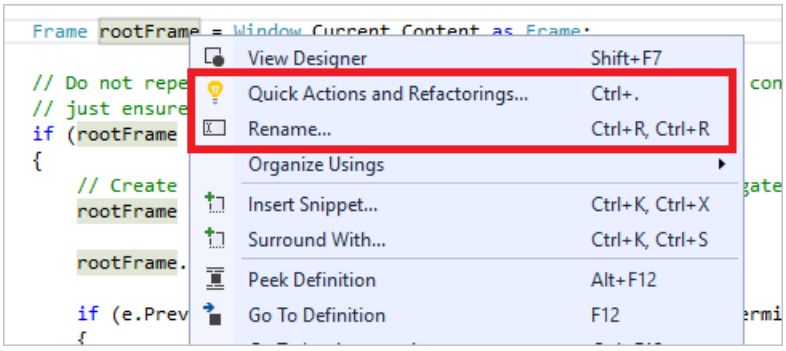
- Code Cleanup

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



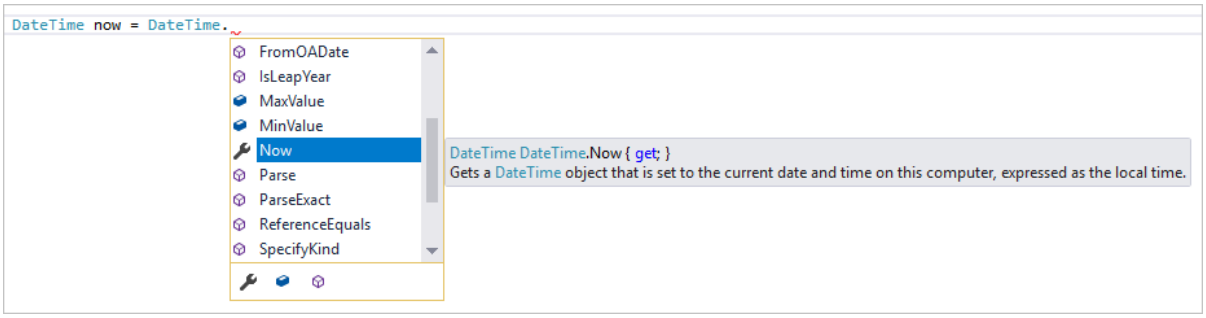
- Refactoring

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



- IntelliSense

  IntelliSense is a term for a set of features that displays 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, which saves you from having to look up type information elsewhere. IntelliSense features vary by language. For more information, see C# IntelliSense, Visual C++ IntelliSense, JavaScript IntelliSense, and Visual Basic IntelliSense. The following illustration shows how IntelliSense displays a member list for a type:

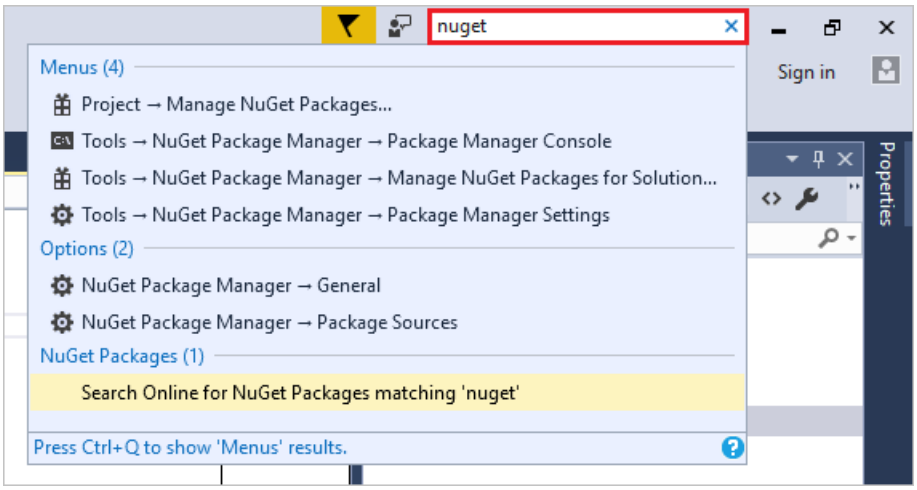


- Search box

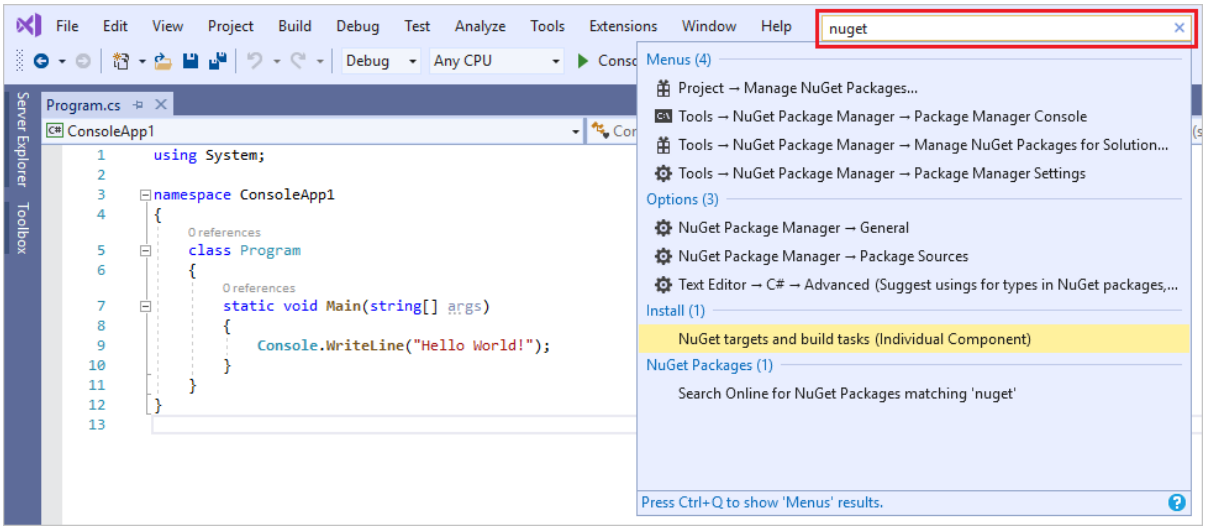
  Visual Studio can seem overwhelming at times with so many menus, options, and properties. The search box is a great way to rapidly find what you need in Visual Studio. When you start typing the name of something

you're looking for, Visual Studio lists results that take you exactly where you need to go. If you need to add functionality to Visual Studio, for example to add support for an additional programming language, the search box provides results that open Visual Studio Installer to install a workload or individual component.

> **TIP**
>
> Press **Ctrl+Q** as a shortcut to the search box.



For more information, see Quick Launch.

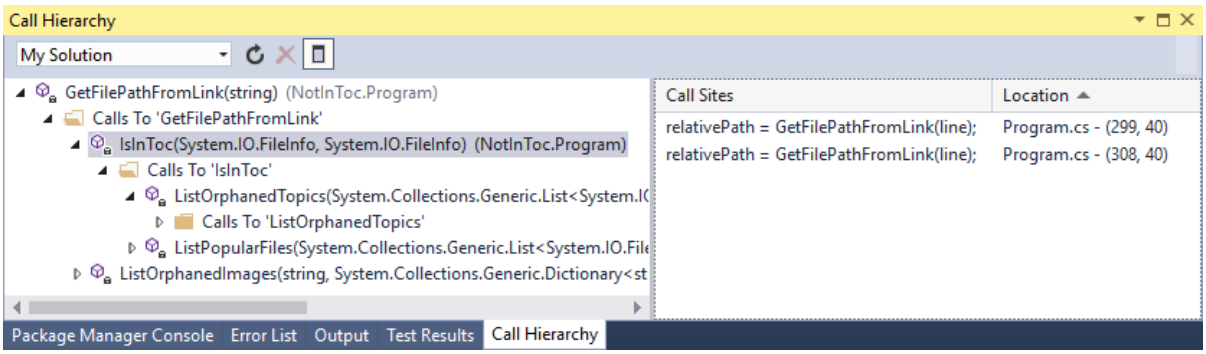


- Live Share

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

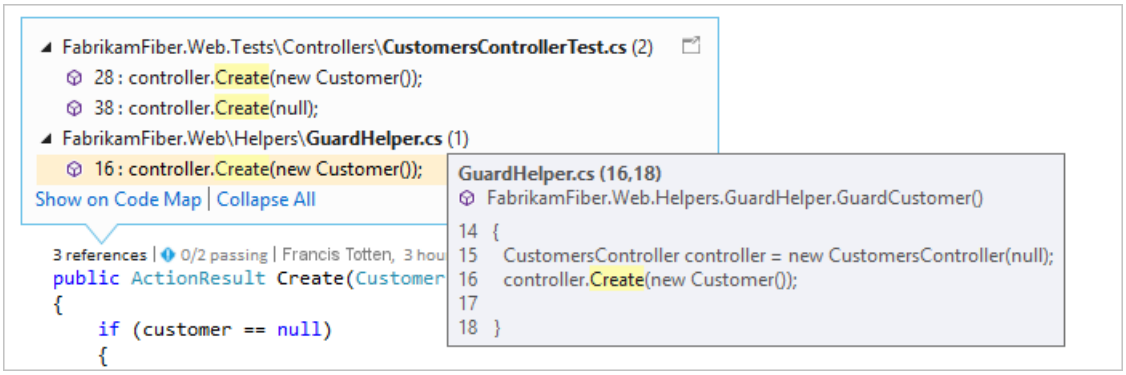
- Call Hierarchy

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



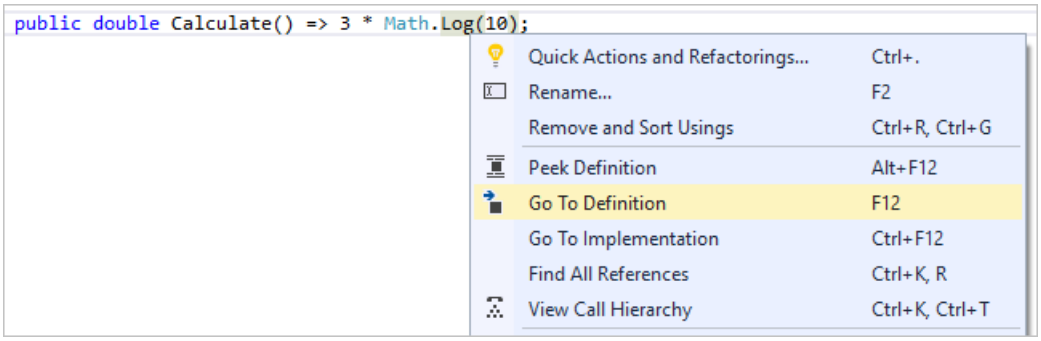
- CodeLens

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



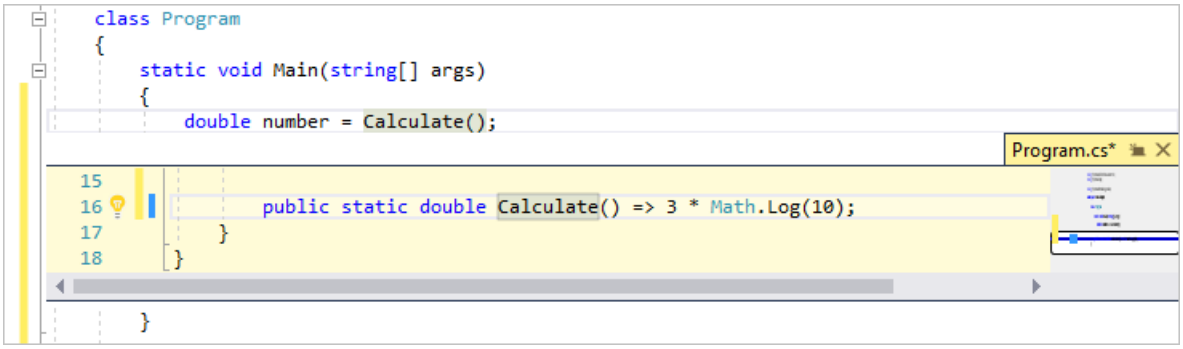
- Go To Definition

  The Go To Definition feature takes you directly to the location where a function or type is defined.



- Peek Definition

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



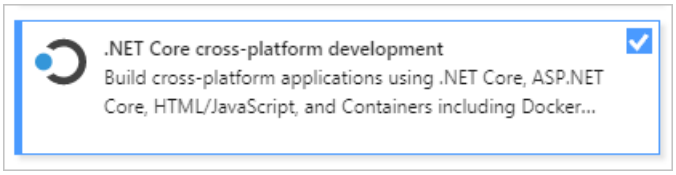
Install the Visual Studio IDE

In this section, you'll create a simple project to try out some of the things you can do with Visual Studio. You'll use

IntelliSense as a coding aid, debug an app to see the value of a variable during the program's 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 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.

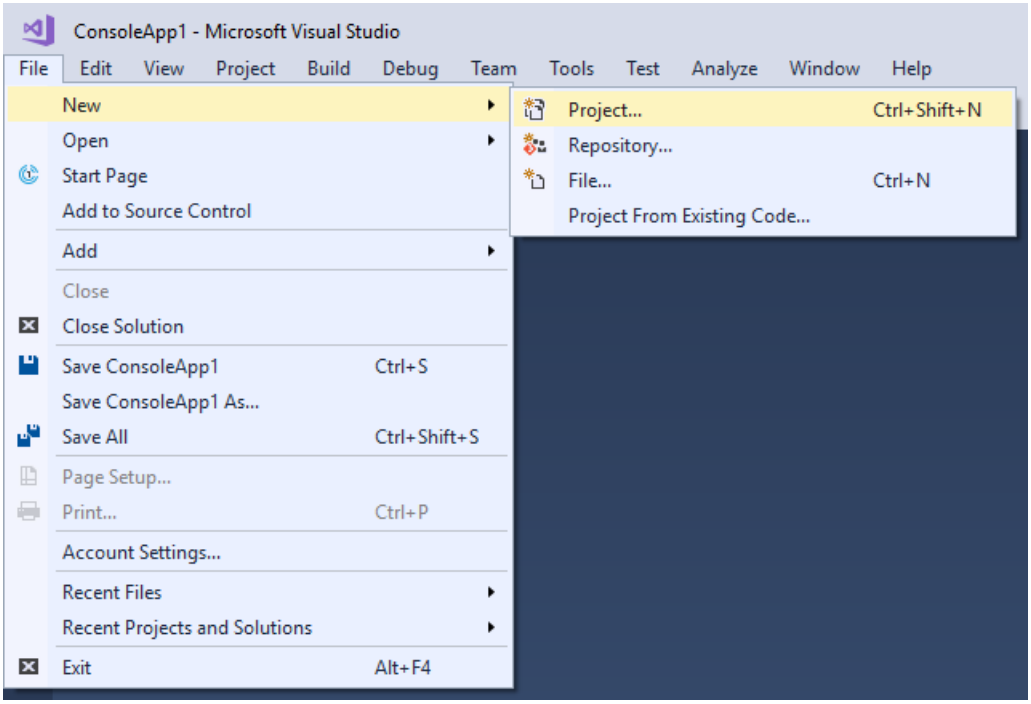


When you open Visual Studio for the first time, you can optionally sign in using your Microsoft account or your work or school account.

Create a program

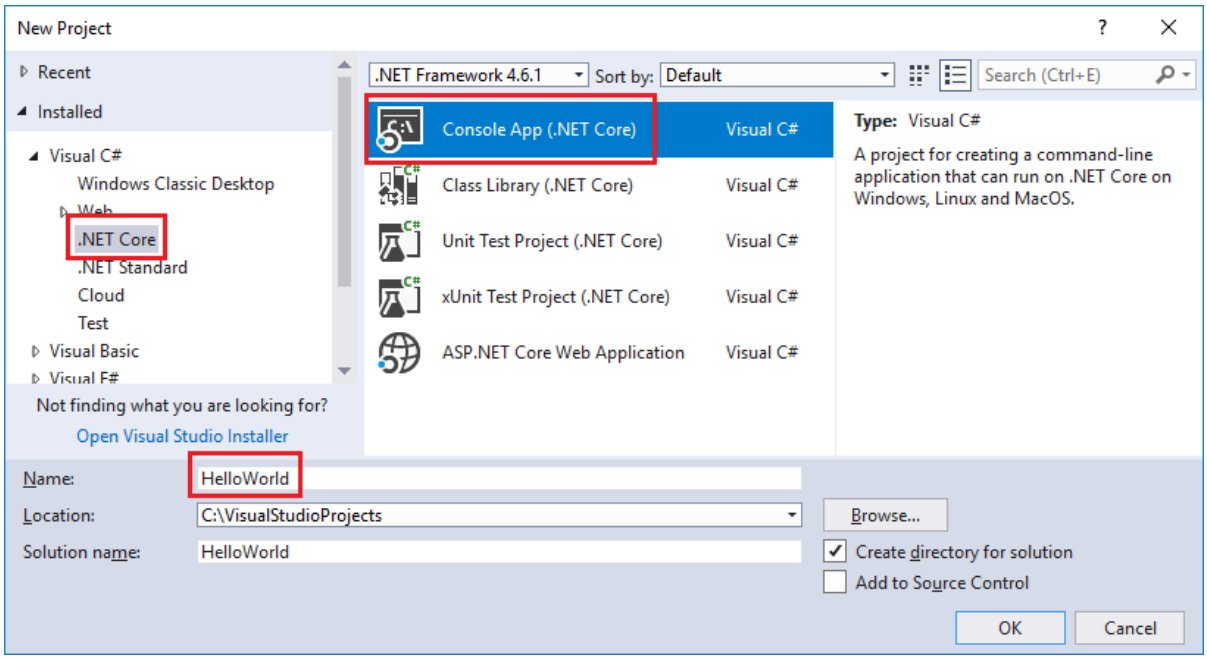
Let's dive in and create a simple program.

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.

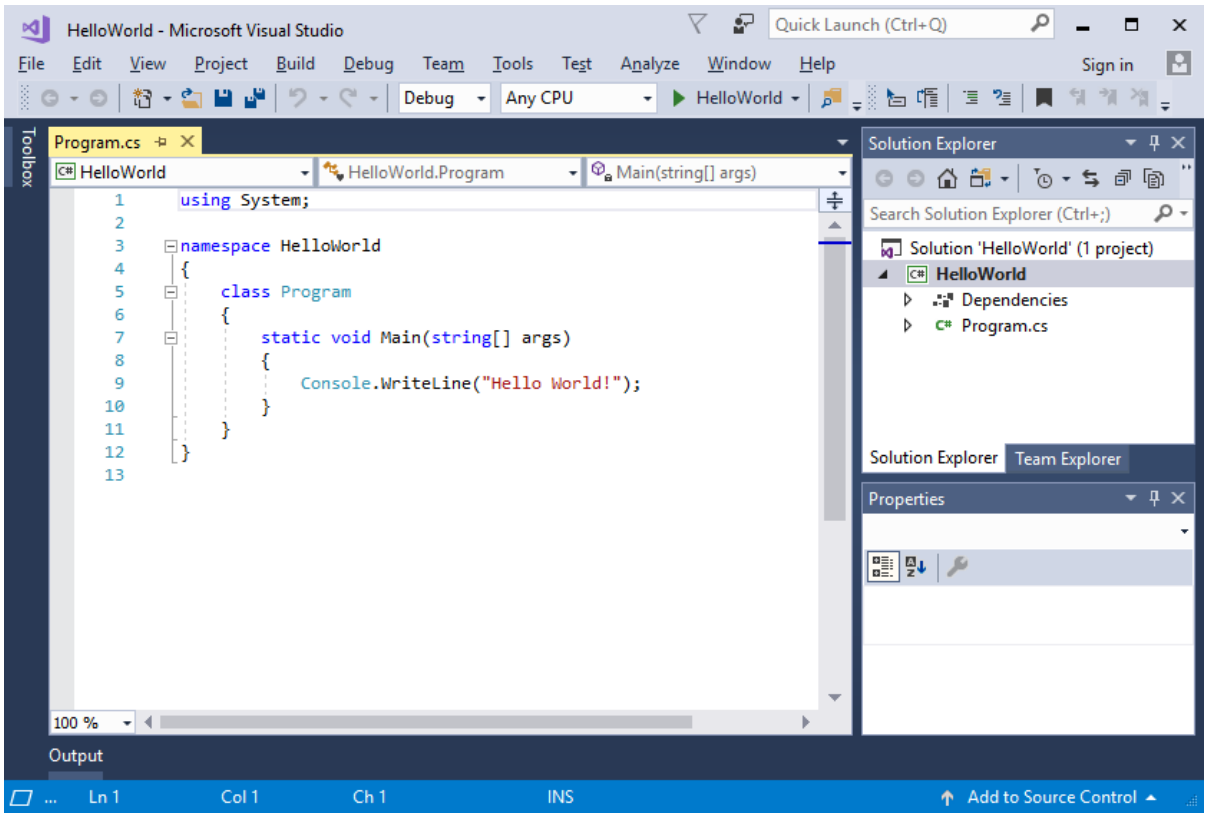


NOTE

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**.

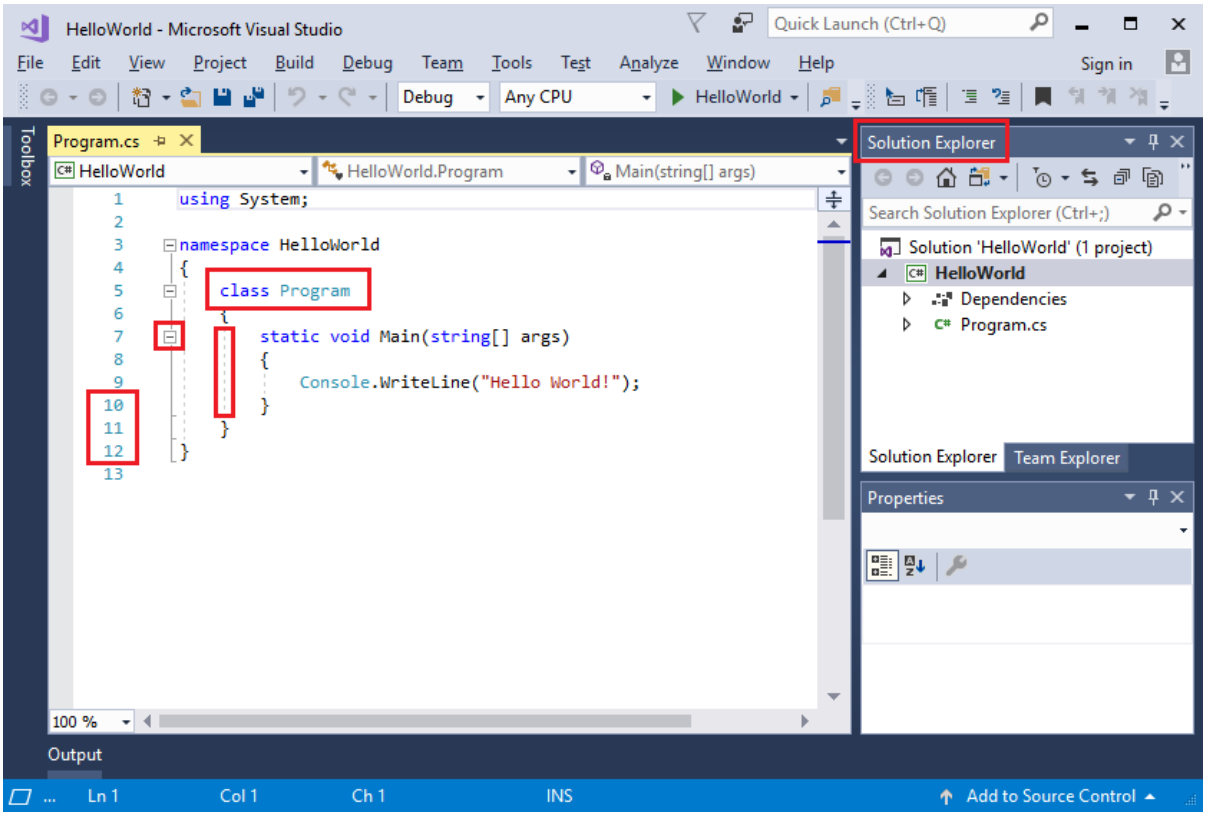
Visual Studio creates the project. It's a simple "Hello World" application that calls the Console.WriteLine() method to display the literal string "Hello World!" in the console (program output) window.

Shortly, you should see something like the following:



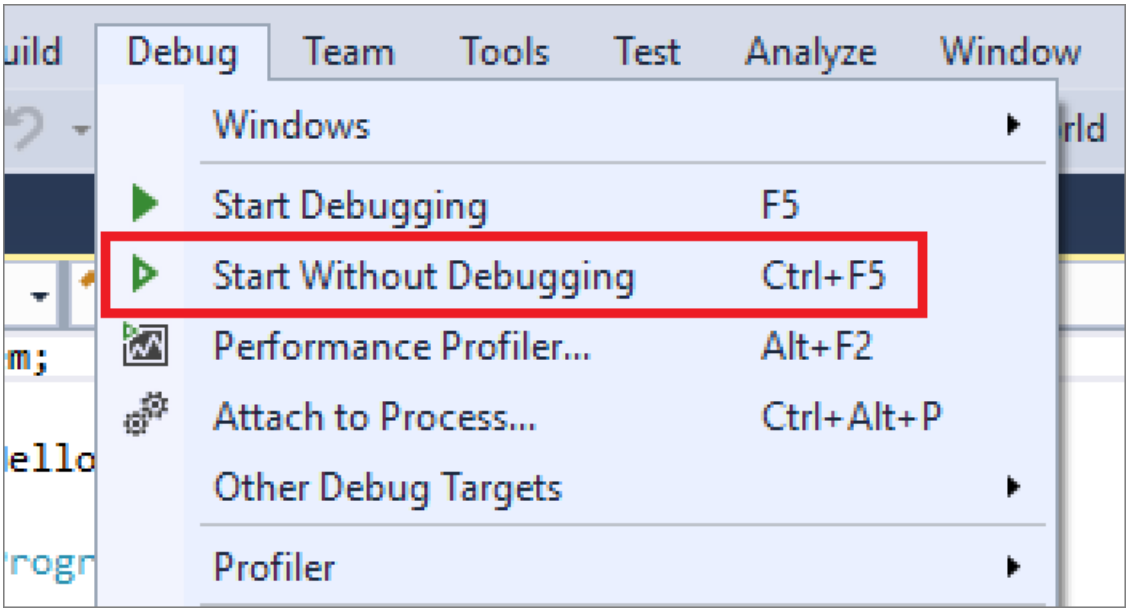
The C# code for your application shows in the editor window, which takes up most of the space. Notice that the text is automatically colorized to indicate different parts of the code, such as keywords and types. In

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**.

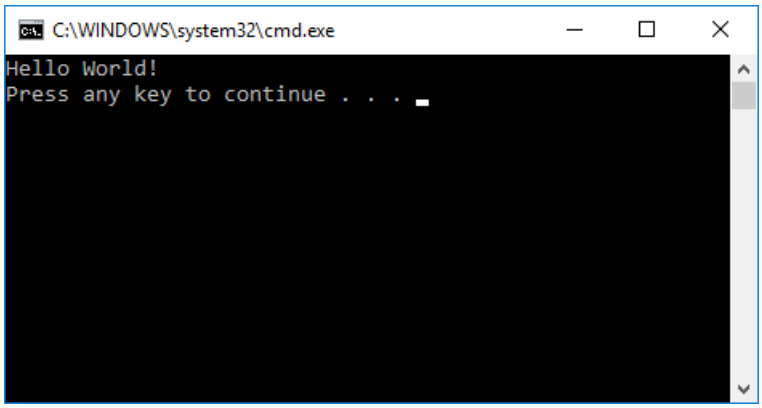


There are other menus and tool windows available, but let's move on for now.

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 additional 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.

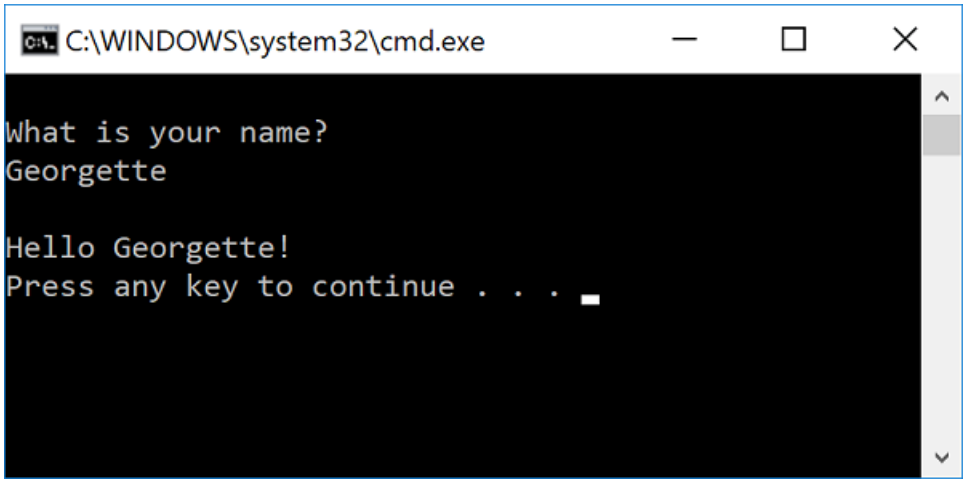
7. Change the line that says `Console.WriteLine("Hello World!");` to the following code:

   ```
   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**.

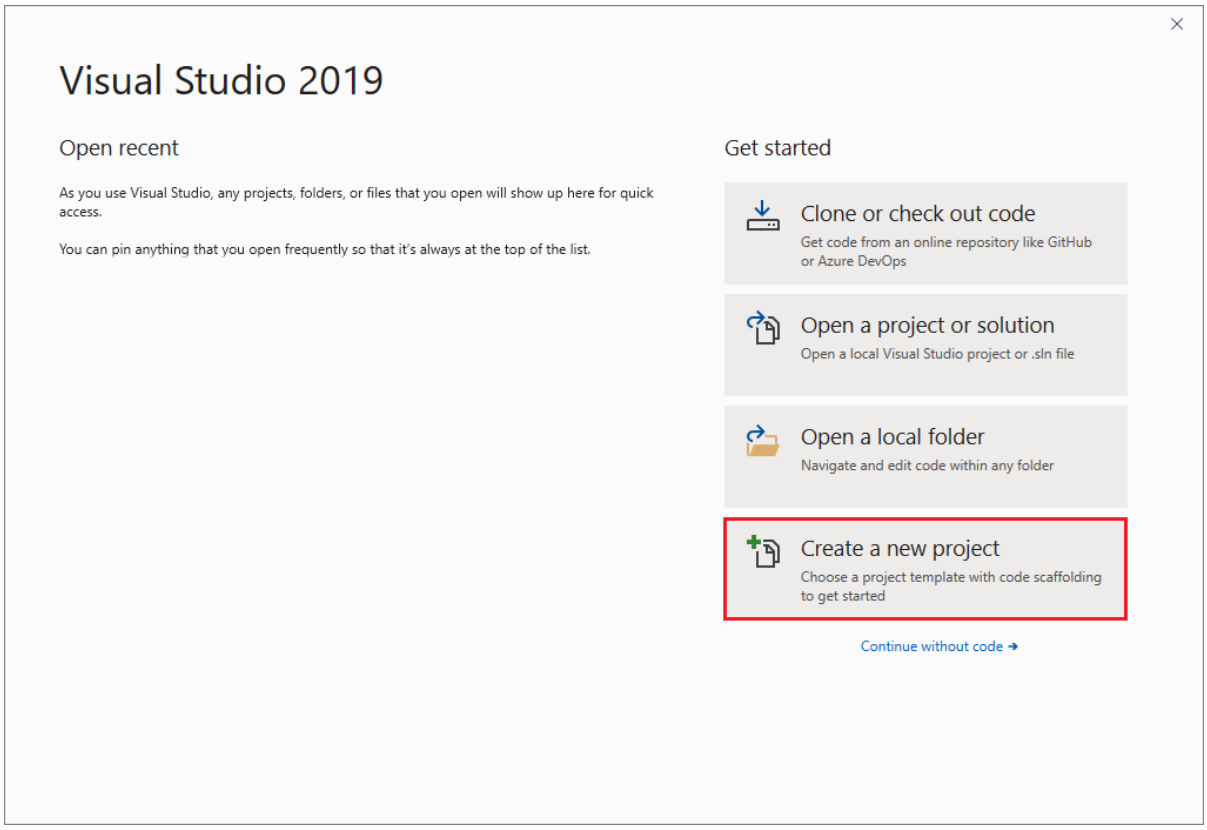


10. Press any key to close the console window and stop the running program.

1. Open Visual Studio.

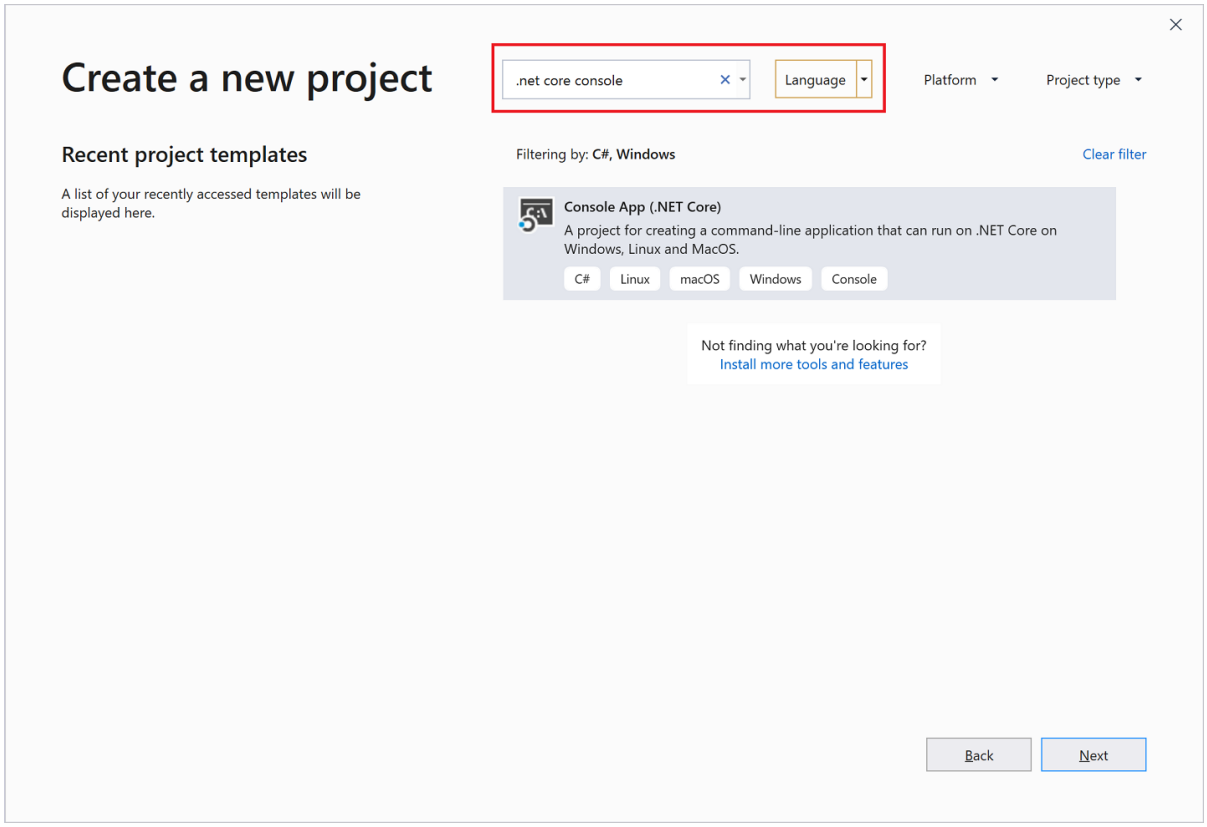
   The start window appears with various options for cloning a repo, opening a recent project, or creating a brand new project.

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

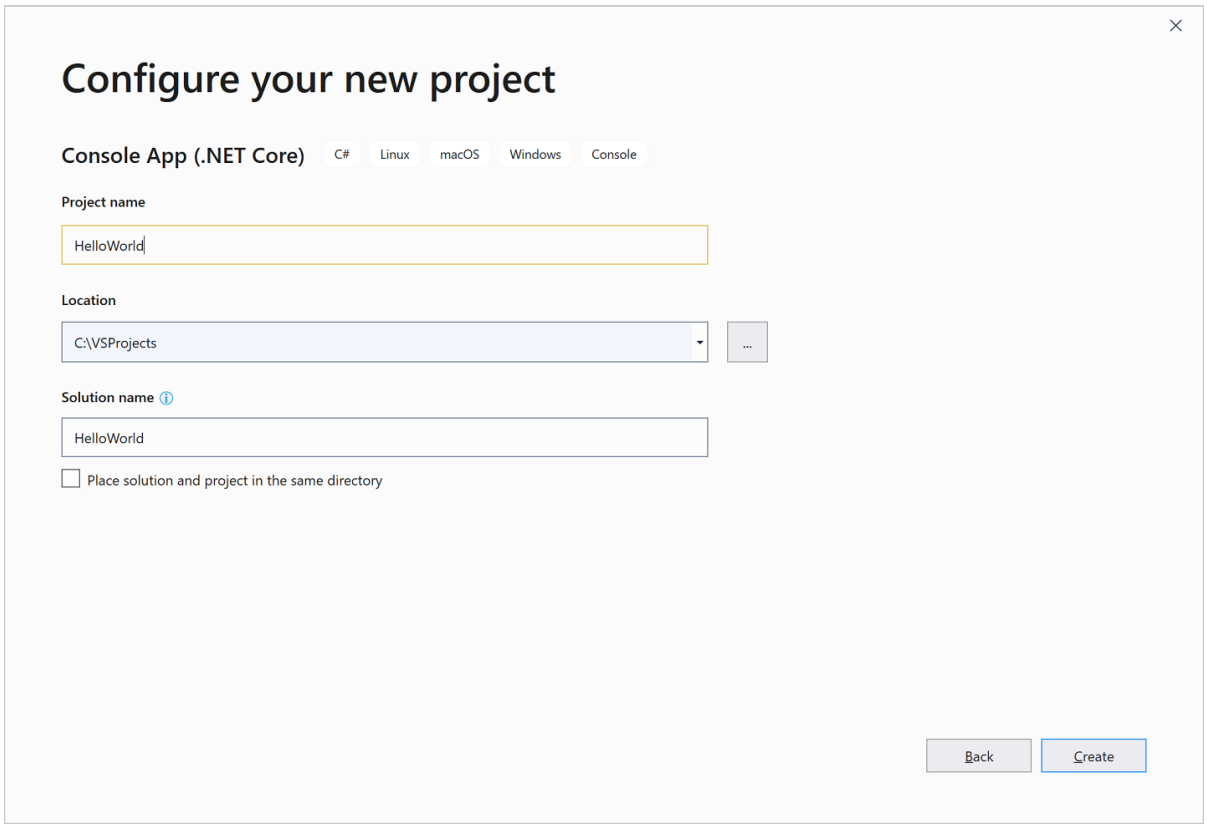


The **Create a new project** window opens and shows several project *templates*. A template contains the basic files and settings needed for a given project type.

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 **Language** drop-down list. Select the **Console App (.NET Core)** template, and then choose **Next**.

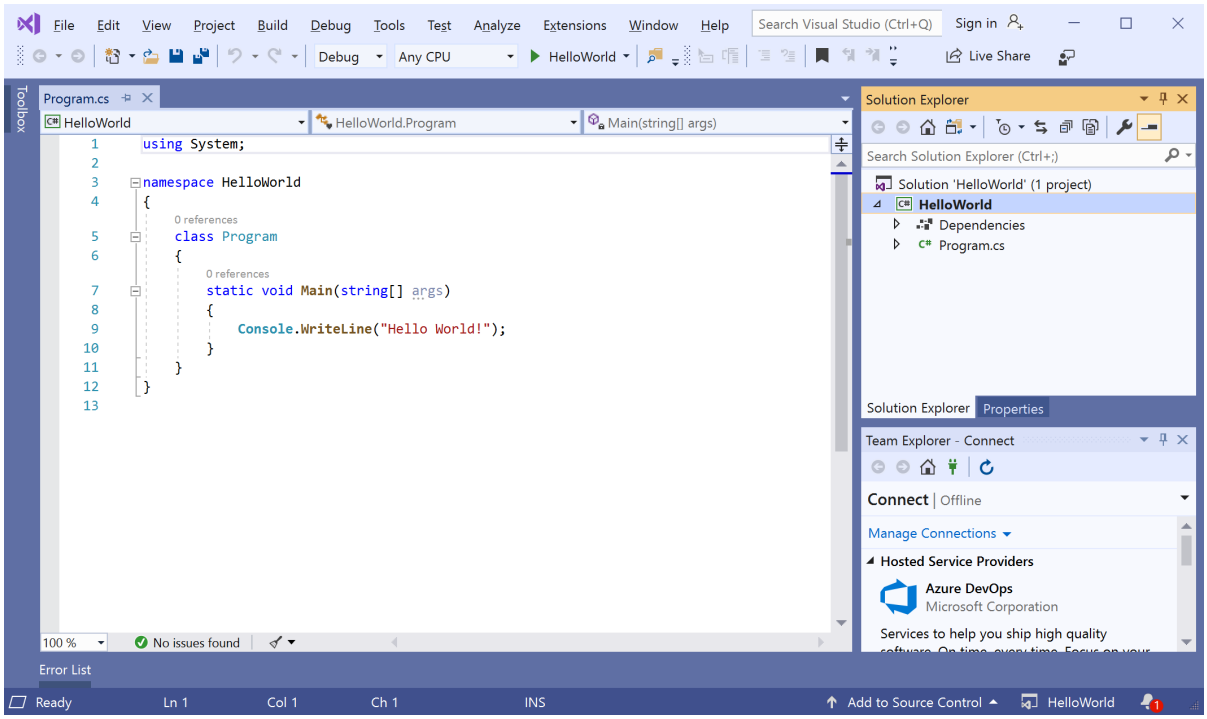


4. In the **Configure your new project** window, enter **HelloWorld** in the **Project name** box, optionally change the directory location for your project files, and then choose **Create**.

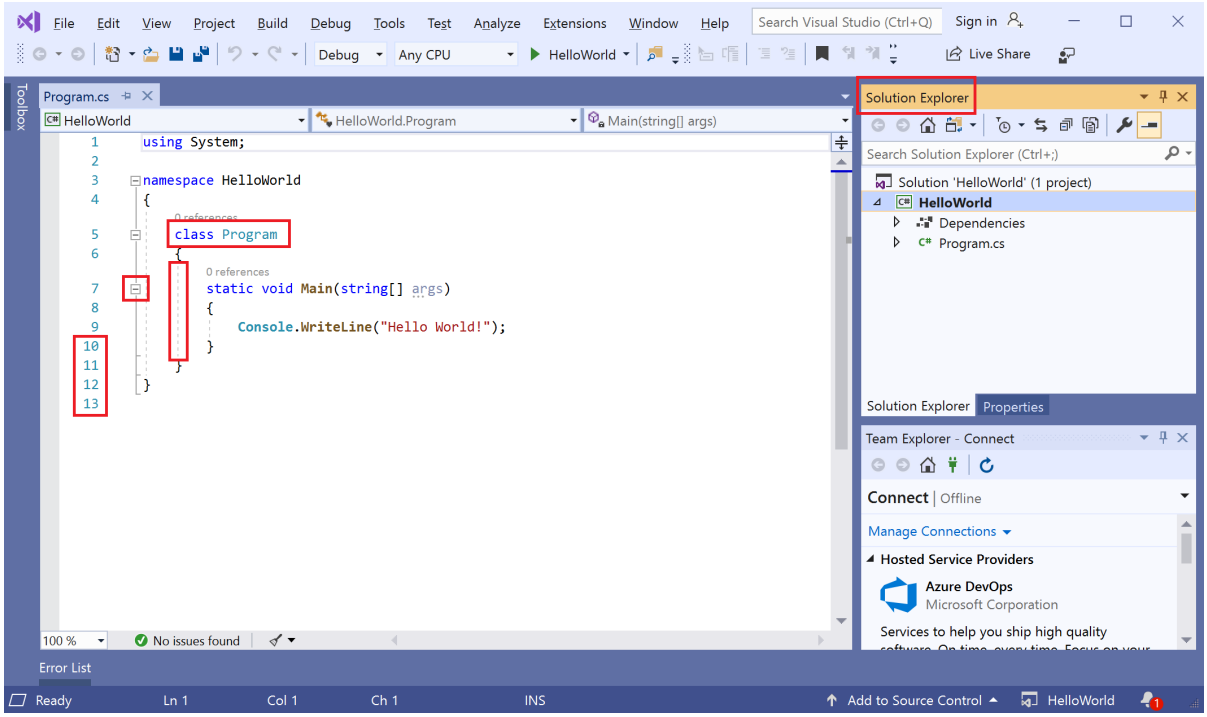


Visual Studio creates the project. It's a simple "Hello World" application that calls the Console.WriteLine() method to display the literal string "Hello World!" in the console (program output) window.

Shortly, you should see something like the following:

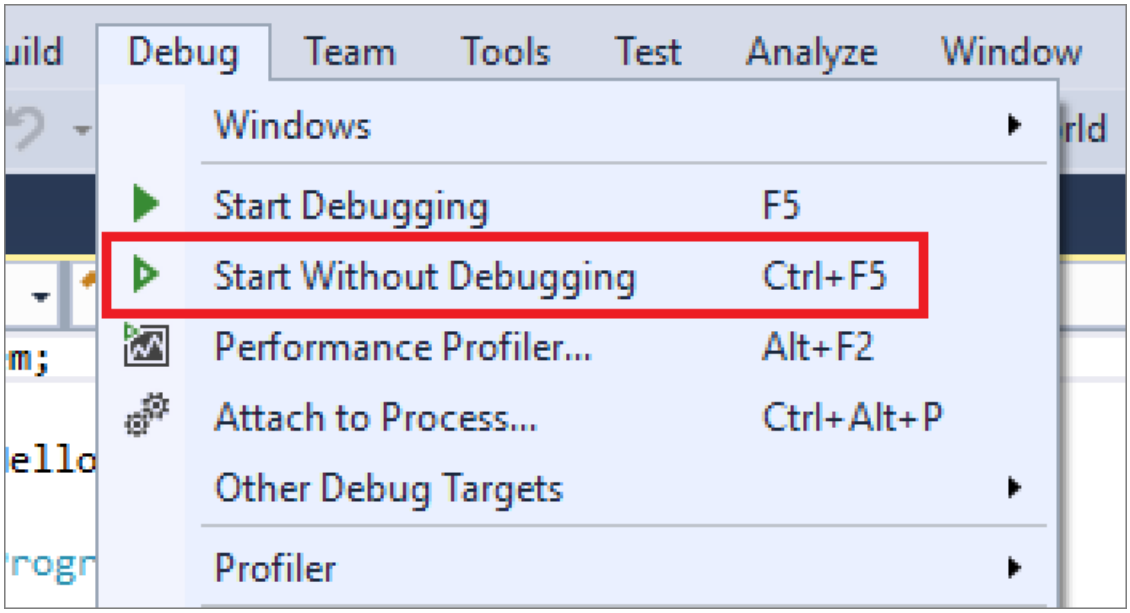


The C# code for your application shows in the editor window, which takes up most of the space. Notice that the text is automatically colorized to indicate different parts of the code, such as keywords and types. In addition, small, vertical dashed lines in the code indicate which braces match one another, 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**.

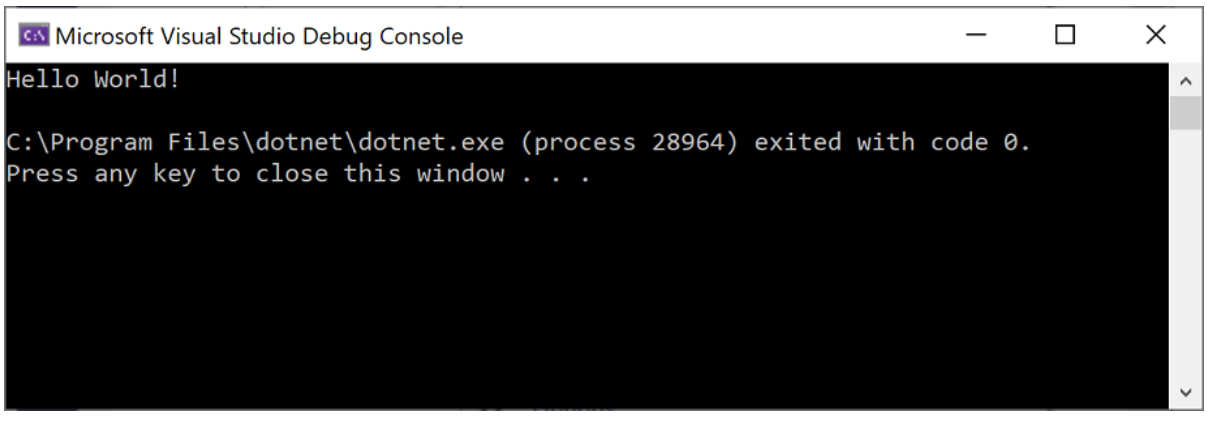


There are other menus and tool windows available, but let's move on for now.

5. 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!



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

7. Let's add some additional 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.

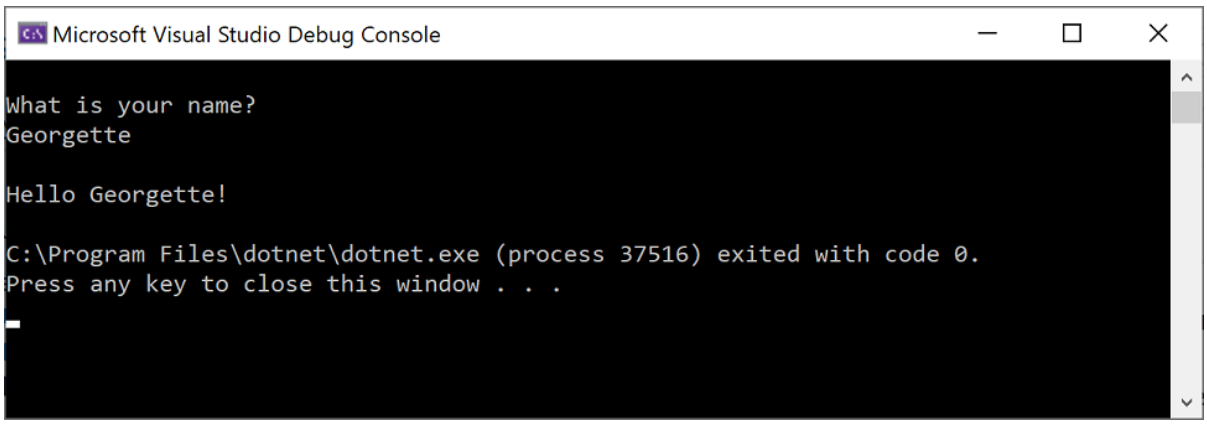
8. Change the line that says `Console.WriteLine("Hello World!");` to the following code:

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

9. 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.

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



11. 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.

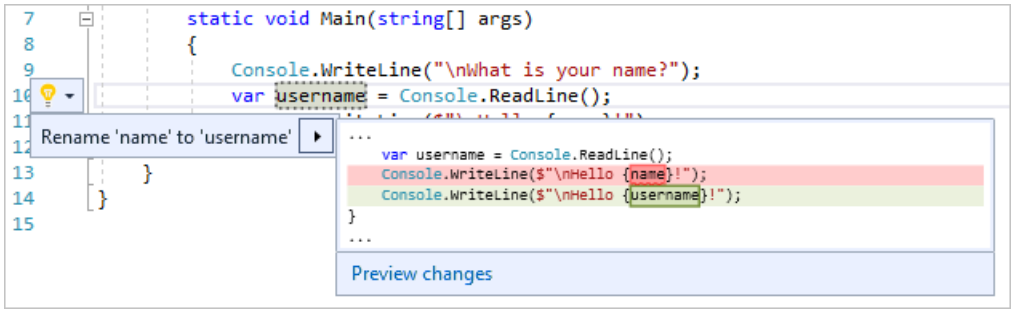
First, let's rename the `name` variable:

1. Double-click the `name` variable to select it.

2. Type in the new name for the variable, **username**.

   Notice that a gray box appears around the variable, and a light bulb appears in the margin.

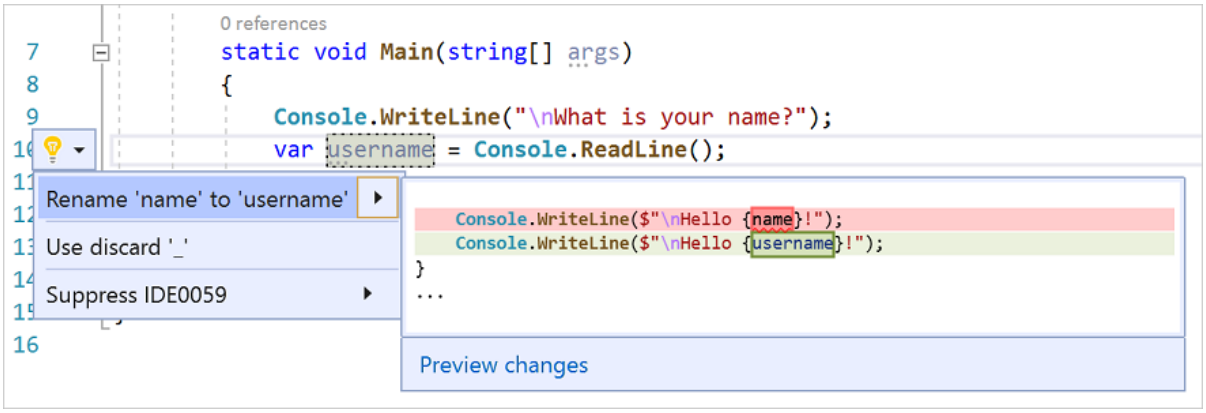
3. Select the light bulb icon to show the available Quick Actions. Select **Rename 'name' to 'username'**.



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}!");
        }
    }
}
```

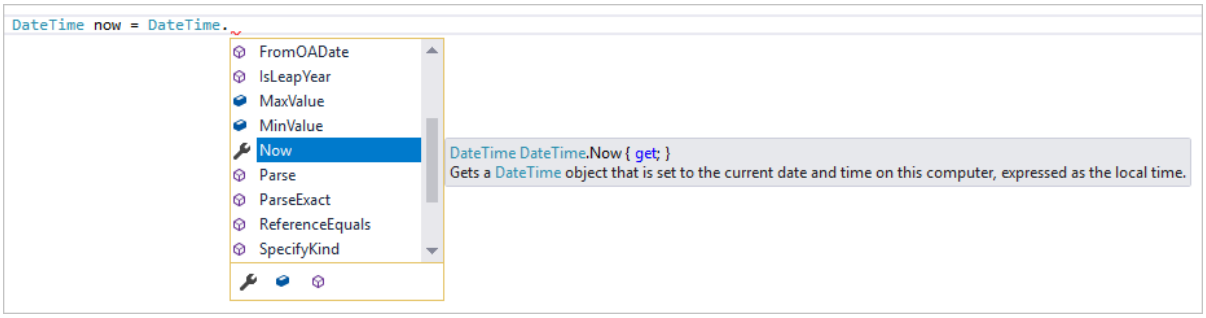
3. Select the light bulb icon to show the available Quick Actions. Select **Rename 'name' to 'username'**.



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

4. Now let's 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. In addition, the description of the currently selected member displays in a separate box.



5. Select the member named **Now**, which is a property of the class, by double-clicking on it or pressing **Tab**. Complete the line of code by adding a semi-colon to the end.

6. Below that, type in or paste the following lines of code:

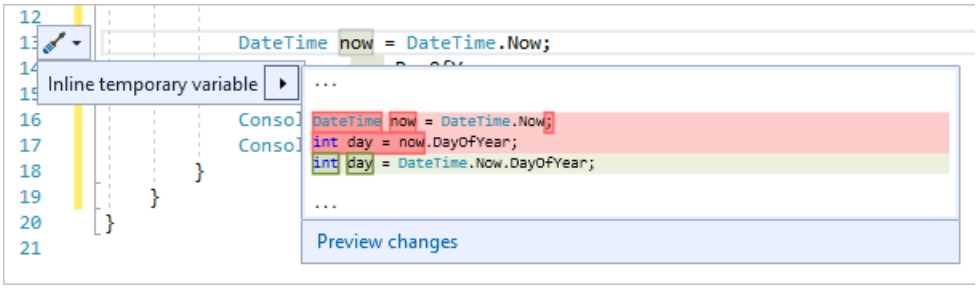
```
int dayOfYear = now.DayOfYear;

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

> **TIP**
>
> Console.Write is a little different to 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 description.

7. Next, we'll use refactoring again to make the code a little more concise. Click on the variable `now` in the line `DateTime now = DateTime.Now;`.

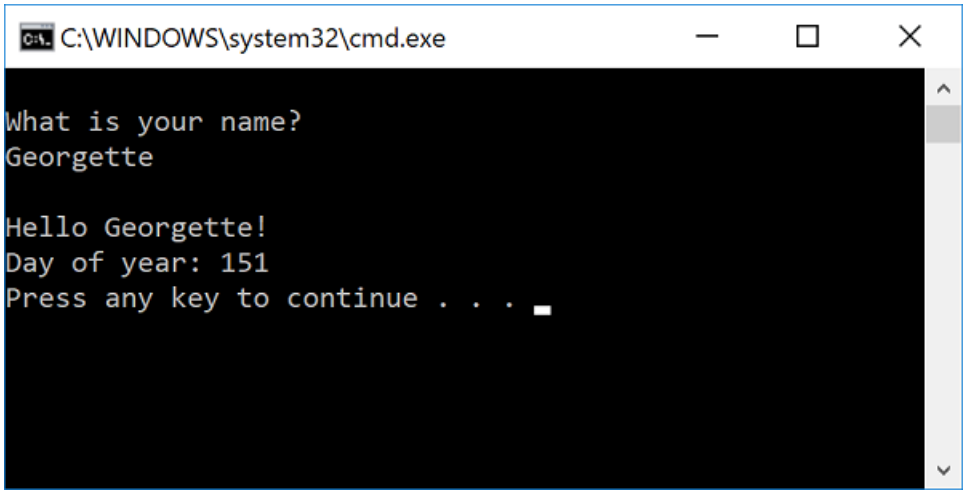
   Notice that a little screwdriver icon appears in the margin on that line.

8. Click the screwdriver icon to see what suggestions Visual Studio has available. In this case, it's showing the Inline temporary variable refactoring to remove a line of code without changing the overall behavior of the code:

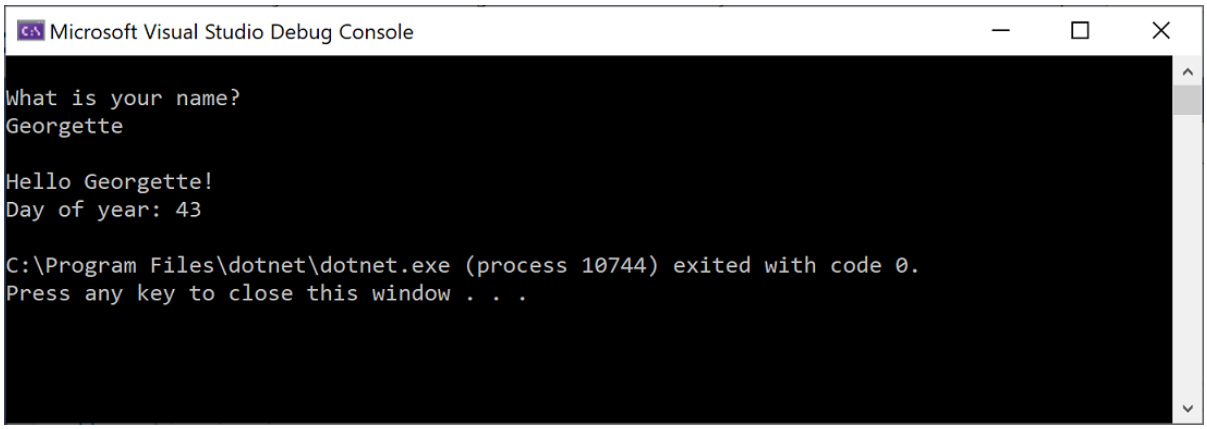


9. Click **Inline temporary variable** to refactor the code.

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



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



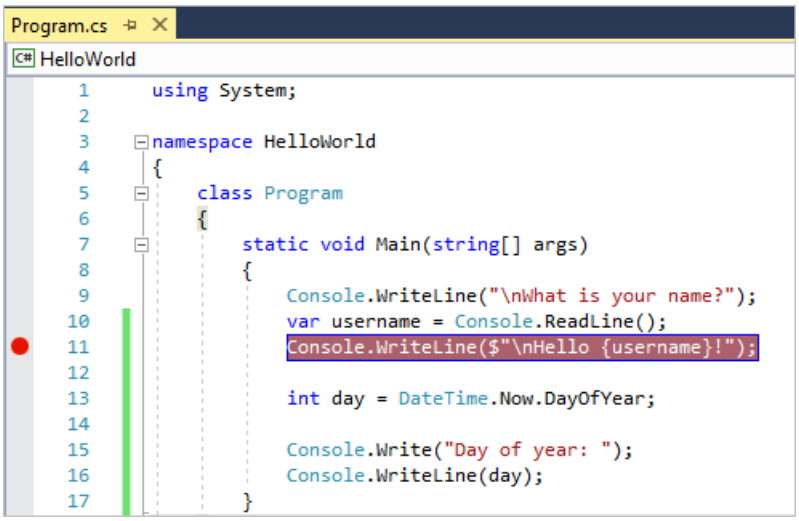
Debug code

When you write code, you need to 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. You can observe how the value of a variable changes as the code runs, and more.

Let's set a breakpoint to see the value of the `username` variable while the program is "in flight".

1. Find the line of code that says `Console.WriteLine($"\nHello {username}!");`. To set a breakpoint on this line of code, that is, to make the program pause execution at this line, click in the far left margin of the editor. You can also click anywhere on the line of code and then press **F9**.

   A red circle appears in the far left margin, and the code is highlighted in red.

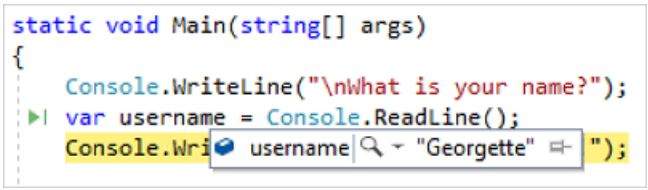


2. Start debugging by selecting **Debug** > **Start Debugging** or by pressing **F5**.

3. When the console window appears and asks for your name, type it in and press **Enter**.

   The focus returns to the Visual Studio code editor and the line of code with the breakpoint is highlighted in yellow. This signifies that it's the next line of code that the program will execute.

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



5. To let the program run to completion, press **F5** again.

To get more details about debugging in Visual Studio, see Debugger feature tour.

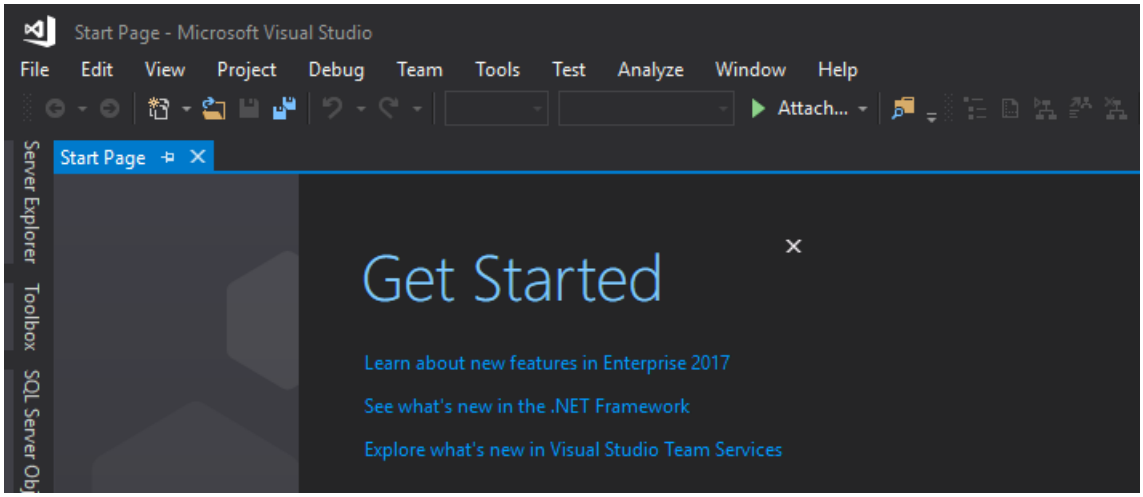
Customize Visual Studio

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

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

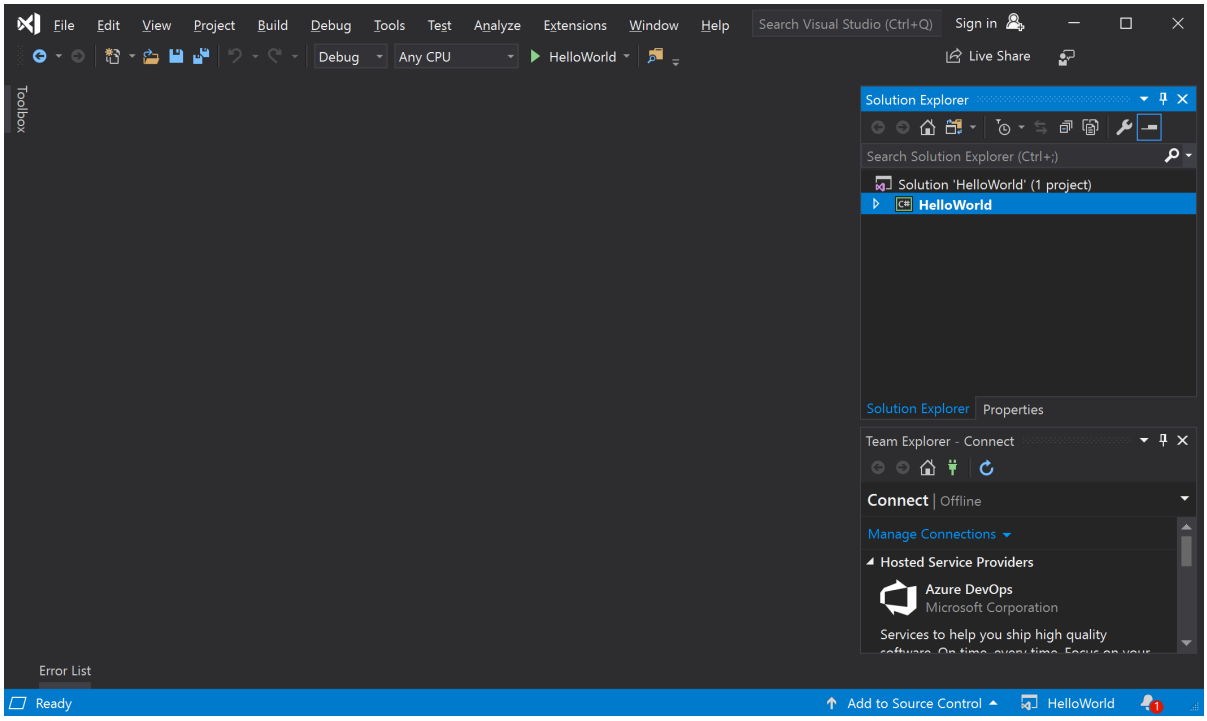
2. On the **Environment** > **General** options page, change the **Color theme** selection to **Dark**, and then choose **OK**.

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



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**.



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

Select environment settings

Let's 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** on the first page, and then choose **Next**.
3. On the **Save Current Settings** page, select an option to save your current settings or not, and then choose **Next**. (If you haven't customized any settings, select **No, just reset settings, overwriting my current settings**.)
4. On the **Choose a Default Collection of Settings** page, choose **Visual C#**, and then choose **Finish**.

5. On the **Reset Complete** page, choose **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

1/1/2020 • 6 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 code easier.

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

> **TIP**
> 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.

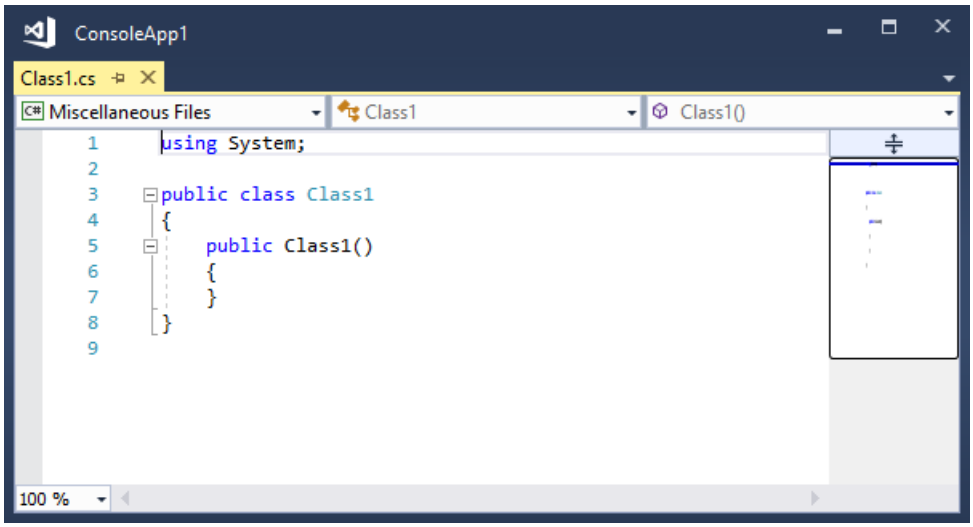
1. Open Visual Studio.

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!)

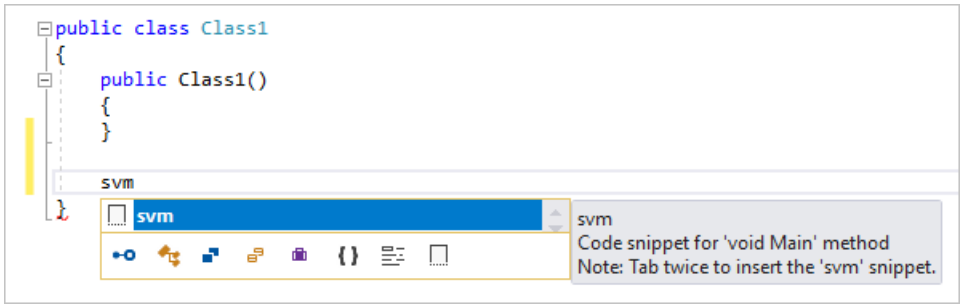


Use code snippets

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).

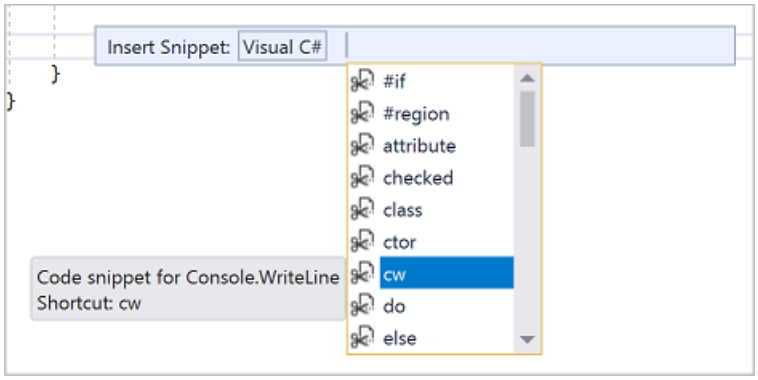
   A pop-up dialog box appears with information about the `svm` code snippet.



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:



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

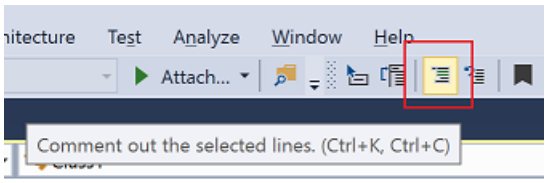
Comment out code

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;
```

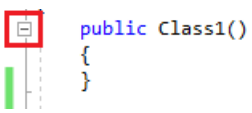
2. We're not using the `morewords` variable, but we may use it later so we don't want to completely delete it. Instead, let's comment out those lines. Select the entire definition of `morewords` to the closing semi-colon, and then choose the **Comment out the selected lines** button on the toolbar. If 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.

Collapse code blocks

We don't want to see the empty constructor for `Class1` that was generated, 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're a keyboard user, place the cursor anywhere in the constructor code and press **Ctrl**+**M**, **Ctrl**+**M**.



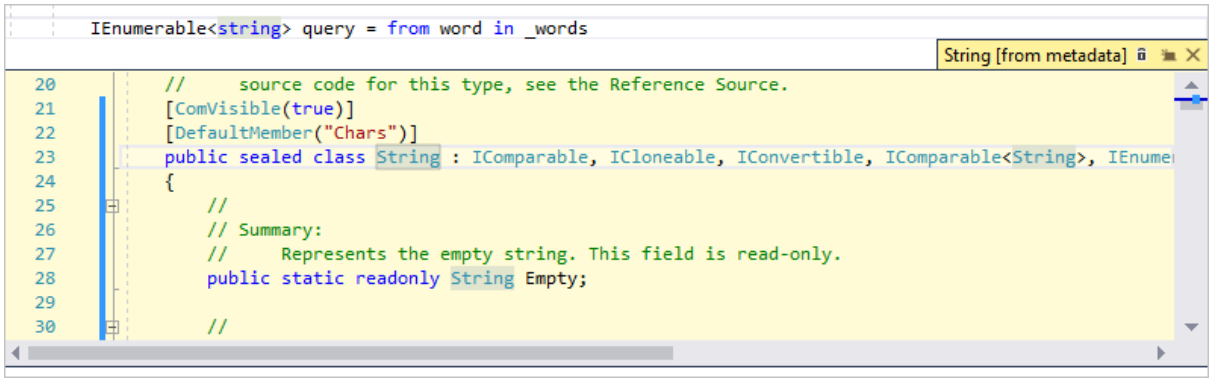
The code block collapses to just the first line, followed by an ellipsis (`...`). To expand the code block again, click the same gray box that now has a plus sign in it, or press **Ctrl**+**M**, **Ctrl**+**M** again. This feature is called Outlining and is especially useful when you're collapsing long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

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.

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);
}
```

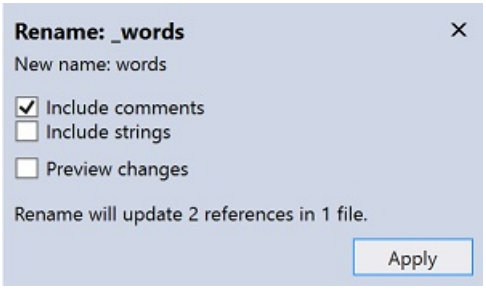
Refactor a name

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.

Next steps

Learn about projects and solutions

See also

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

Learn about projects and solutions

2/25/2020 • 8 minutes to read • Edit Online

In this introductory article, we'll explore what it means to create a *solution* and a *project* in Visual Studio. A solution is a container that's used to organize one or more related code projects, for example a class library project and a corresponding test project. We'll look at the properties of a project and some of the files it can contain. We'll also create a reference from one project to another.

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.

We'll construct a solution and project from scratch as an educational exercise to understand the concept of a project. In your general use of Visual Studio, you'll likely use some of the various project *templates* that Visual Studio offers when you create a new project.

> **NOTE**
> Solutions and projects aren't required to develop apps in Visual Studio. You can also just open a folder that contains code and start coding, building, and debugging. For example, if you clone a GitHub repo, it might not contain Visual Studio projects and solutions. For more information, see Develop code in Visual Studio without projects or solutions.

Solutions and projects

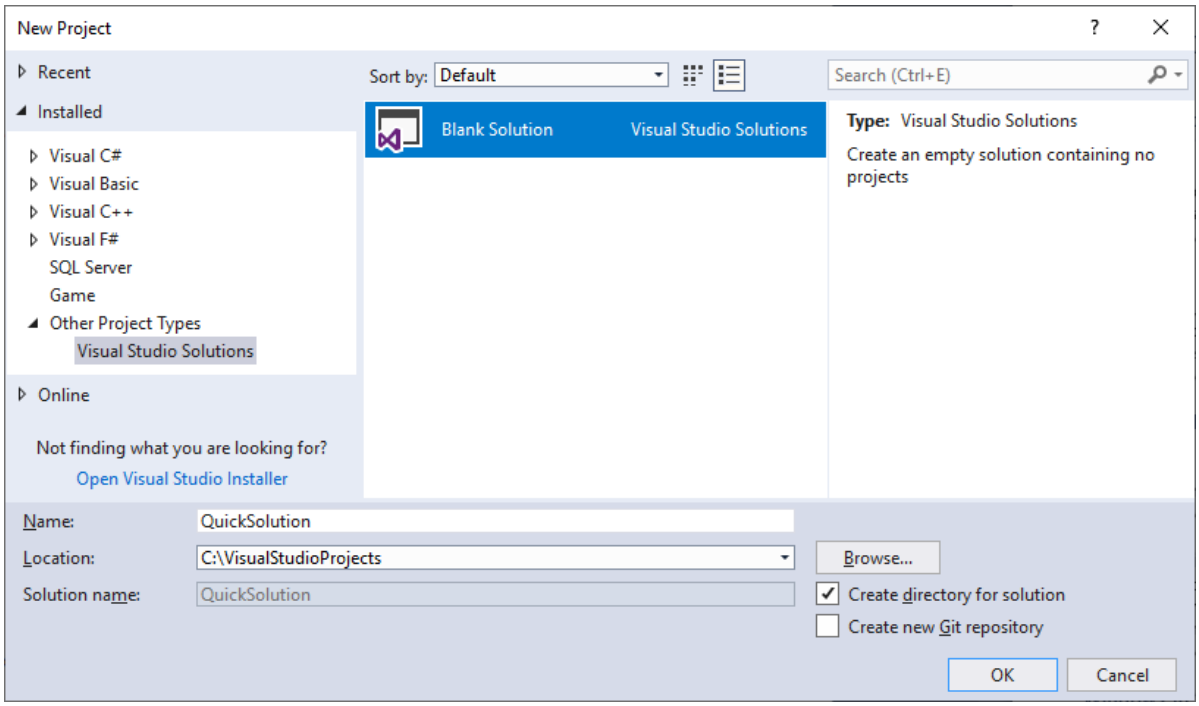
Despite its name, a solution is not an "answer". A solution is simply a container used by Visual Studio to organize one or more related projects. When you open a solution in Visual Studio, it automatically loads all the projects that the solution contains.

Create a solution

We'll start our exploration by creating an empty solution. After you get to know Visual Studio, you probably won't find yourself creating empty solutions very often. When you create a new project, Visual Studio automatically creates a solution to house the project if there's not a solution already open.

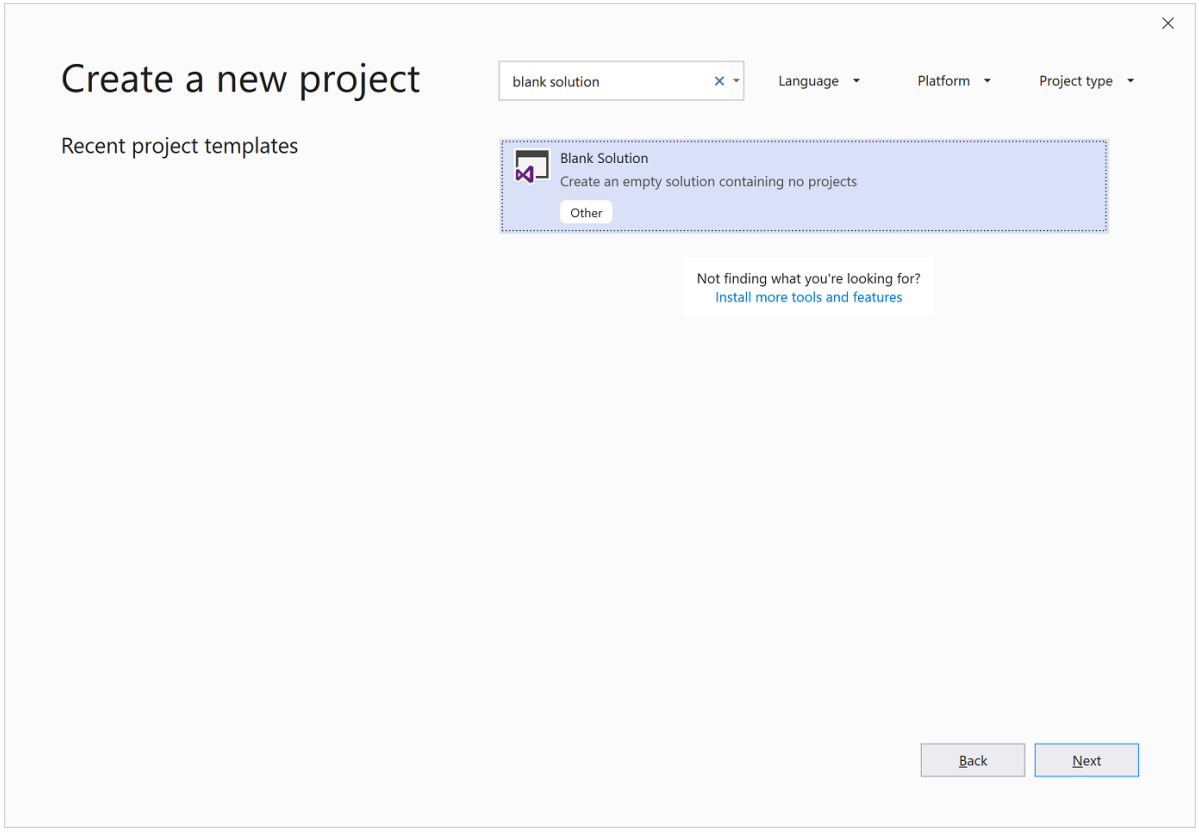
1. Open Visual Studio.
2. On the top menu bar, choose **File** > **New** > **Project**.

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



4. Name the solution **QuickSolution**, and then choose **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.

Add a project

Now let's add our first project to the solution. We'll start with an empty project and add the items we need to the project.

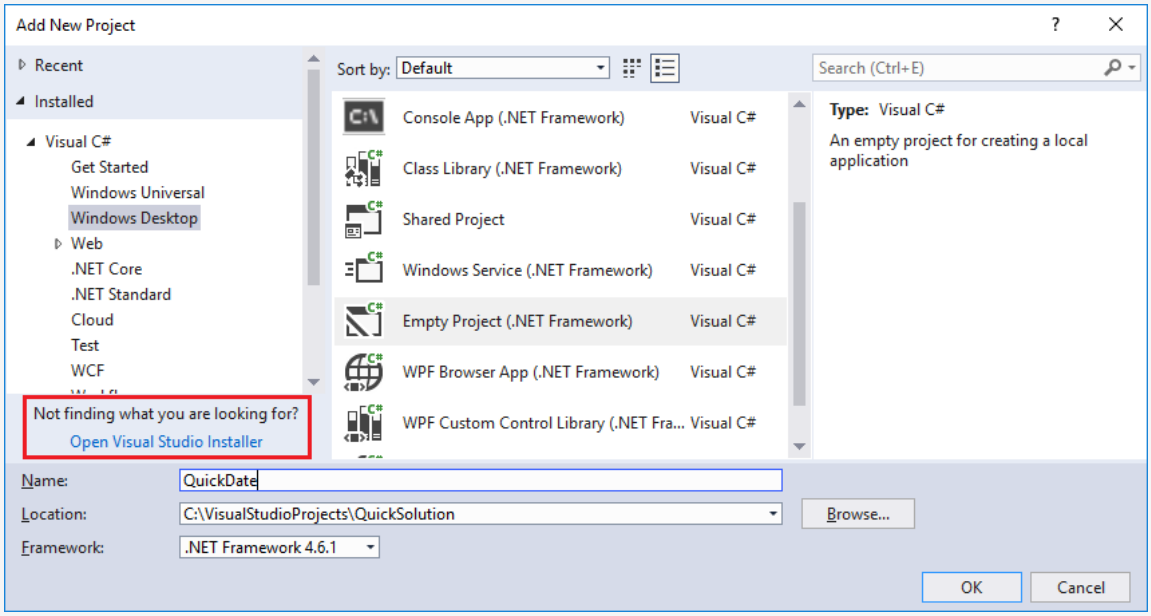
1. From the right-click or context menu of **Solution 'QuickSolution'** in **Solution Explorer**, choose **Add** > **New Project**.

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

2. In the left pane, expand **Visual C#** and choose **Windows Desktop**. Then, in the middle pane, choose the **Empty Project (.NET Framework)** template. Name the project **QuickDate**, then choose **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 choose the **Open Visual Studio Installer** link in the bottom left corner of the **Add New Project** dialog box. After Visual Studio Installer launches, choose the **.NET desktop development** workload and then the **Modify** button.



1. From the right-click or context menu of **Solution 'QuickSolution'** in **Solution Explorer**, choose **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**.

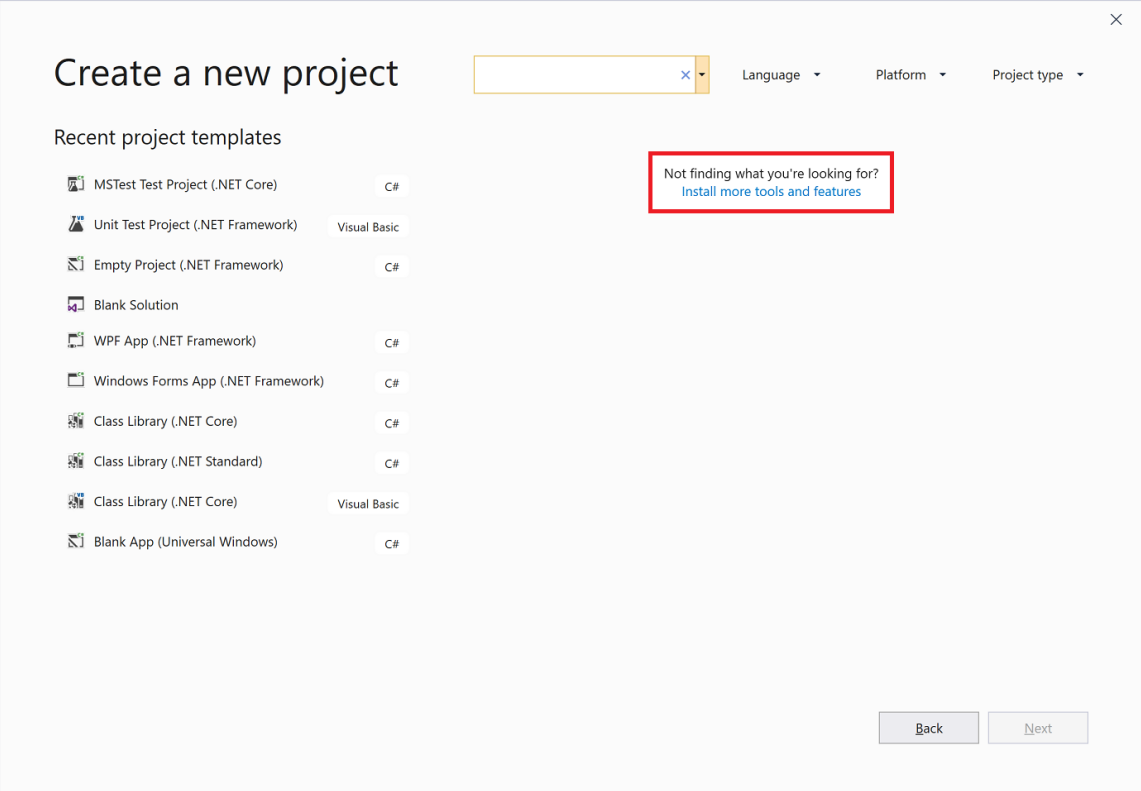
3. Select the **Empty Project (.NET Framework)** template, and then choose **Next**.

4. Name the project **QuickDate**, then choose **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 choose the **Install more tools and features** link under the text that says **Not finding what you're looking for?**. After Visual Studio Installer launches, choose the **.NET desktop development** workload and then the **Modify** button.



Add an item to the project

We have an empty project. Let's add a code file.

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

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

2. Expand **Visual C# Items**, then choose **Code**. In the middle pane, choose the **Class** item template. Name the class **Calendar**, and then choose the **Add** button.

   A file named *Calendar.cs* is added to the project. The *.cs* on the end is the file extension that is given to C# code files. The file appears in the visual project hierarchy in **Solution Explorer**, and its contents are opened 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 what the code does, but if you want, you can run the program by pressing **Ctrl+F5** and see that it prints today's date to the console (or standard output) window.

Add a second project

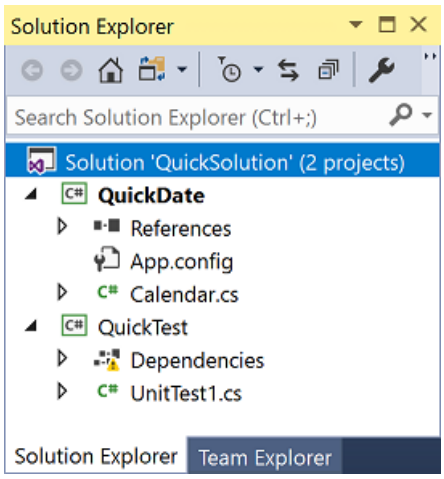
It is common for solutions to contain more than one project, and often these projects reference each other. Some projects in a solution might be class libraries, some executable applications, and some might be unit test projects or websites.

Let's add a unit test project to our solution. This time we'll start from a project template so we don't have to add an additional code file to the project.

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

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

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

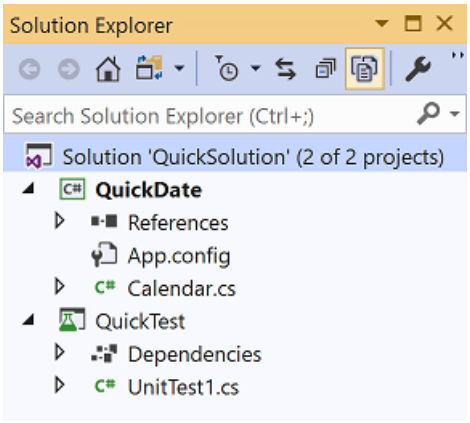


2. In the **Add a new project** dialog box, enter the text **unit test** into the search box at the top, and then select **C#** under **Language**.

3. Choose the **MSTest Test Project (.NET Core)** project template, and then choose **Next**.

4. Name the project **QuickTest**, and then choose **Create**.

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



Add a project reference

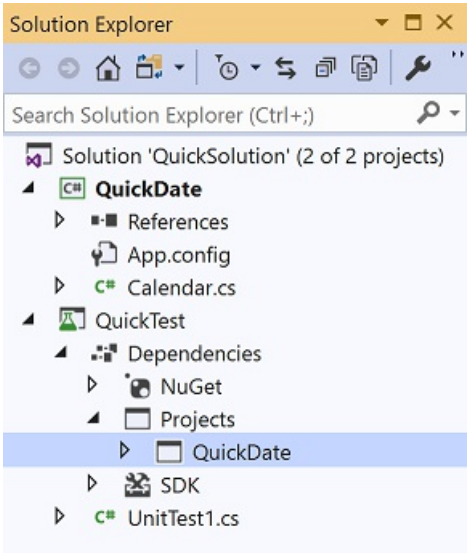
We're going to use the new unit test project to test our method in the **QuickDate** project, so we need to add a reference to that project. This creates a *build dependency* between the two projects, meaning that when you build the solution, **QuickDate** is built before **QuickTest**.

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

   The **Reference Manager** dialog box opens.

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

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



Add test code

1. Now we'll 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());
        }
    }
}
```

You'll see a red squiggle under some of the code. We'll fix this error by making the test project a friend assembly to the **QuickDate** project.

2. Back in the **QuickDate** project, open the *Calendar.cs* file if it's not already open. 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 code file should look like this:

```
1   using System;
2   using System.Runtime.CompilerServices;
3
4   [assembly: InternalsVisibleTo("QuickTest")]
5
6   namespace QuickDate
7   {
8       internal class Calendar
9       {
10          static void Main(string[] args)
11          {
12              DateTime now = GetCurrentDate();
13              Console.WriteLine($"Today's date is {now}");
14              Console.ReadLine();
15          }
16
17          internal static DateTime GetCurrentDate()
18          {
19              return DateTime.Now.Date;
20          }
21      }
22  }
23
```

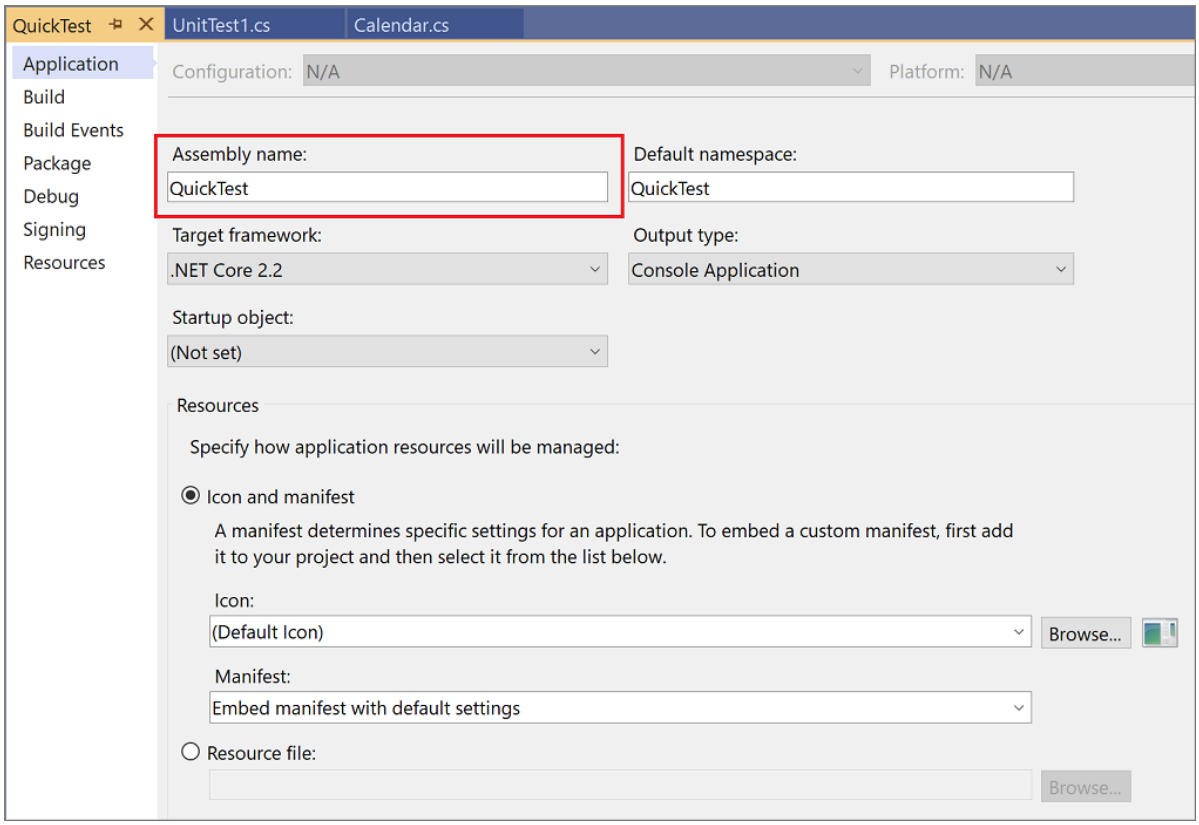
Project properties

The line in the *Calendar.cs* file that contains the InternalsVisibleToAttribute attribute references the assembly name (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, open the project properties.

1. In **Solution Explorer**, select the **QuickTest** project. From the right-click or context menu, select

Properties, or just press **Alt+Enter**.

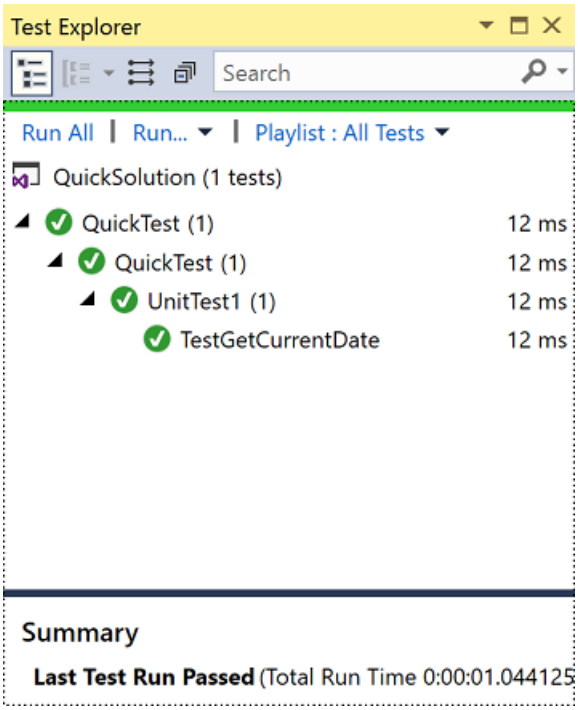
The *property pages* for the project open on the **Application** tab. The property pages contain various settings for the project. Notice that the assembly name of the **QuickTest** project is indeed "QuickTest". If you wanted to change it, this is where you'd do that. Then, when you build the test project, the name of the resulting binary file would change from *QuickTest.dll* to whatever you chose.



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.

Next steps

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.



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.

See also

- Create 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

3/3/2020 • 7 minutes to read • Edit Online

The Visual Studio IDE overview article gives a basic introduction to Visual Studio. This article describes features that might be more appropriate for experienced developers, or those developers who are already familiar with Visual Studio.

Modular installation

Visual Studio's modular installer enables you to choose and install *workloads*. Workloads are groups of features needed for the programming language or platform you prefer. This strategy helps to keep the footprint of the Visual Studio installation smaller, which means it installs and updates faster too.

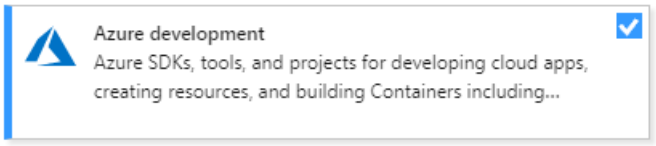
If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

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

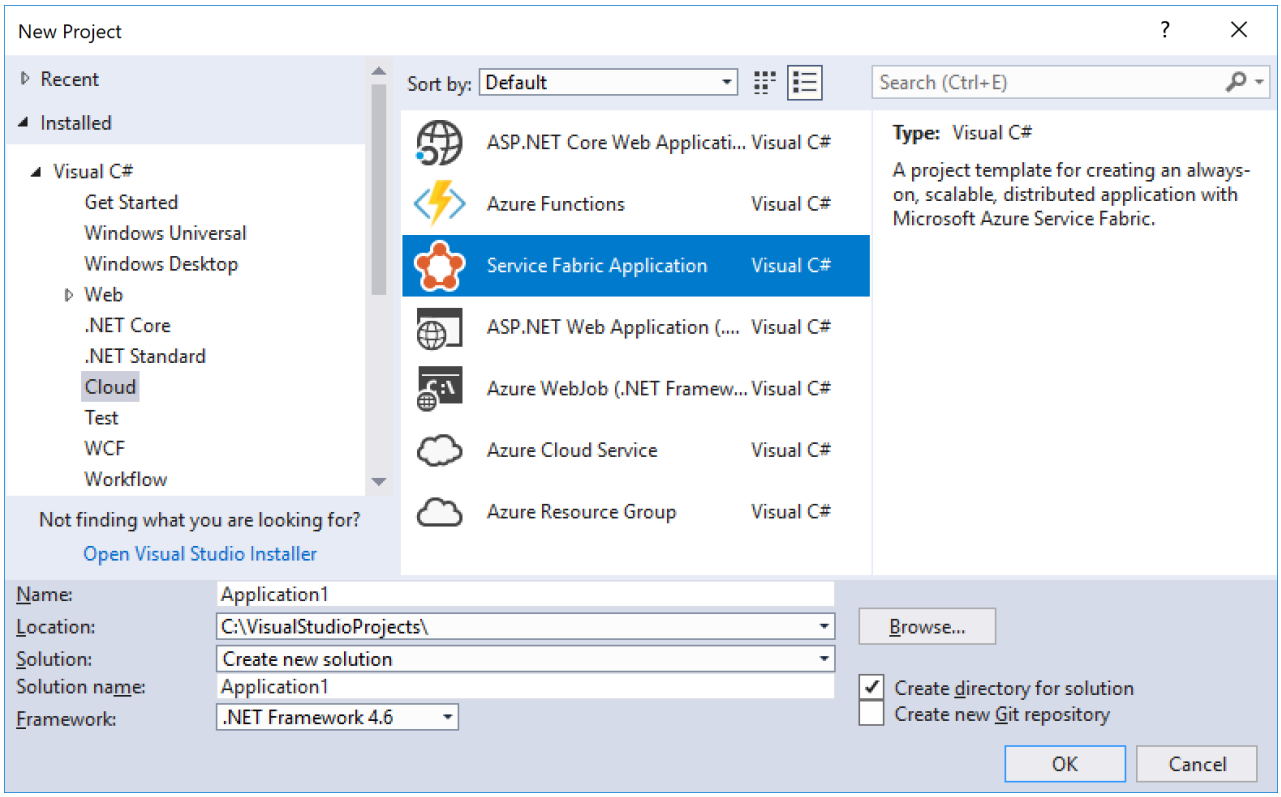
To learn more about setting up Visual Studio on your system, see Install Visual Studio.

Create cloud-enabled apps for Azure

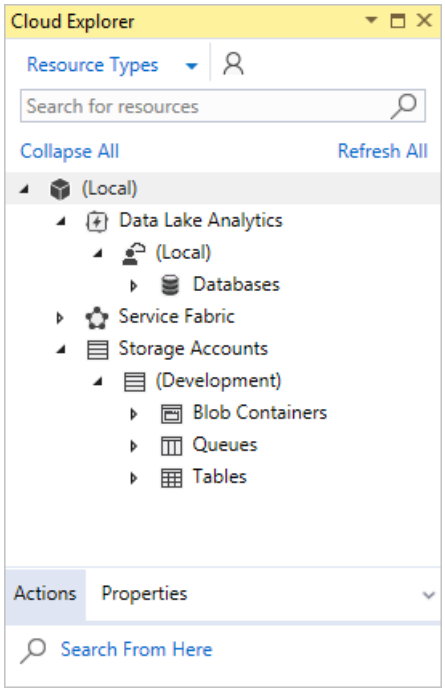
Visual Studio offers a suite of tools that enable you to easily create cloud-enabled applications powered by Microsoft Azure. You can configure, build, debug, package, and deploy applications and services on Microsoft Azure directly from the IDE. To get the Azure tools and project templates, select the **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:



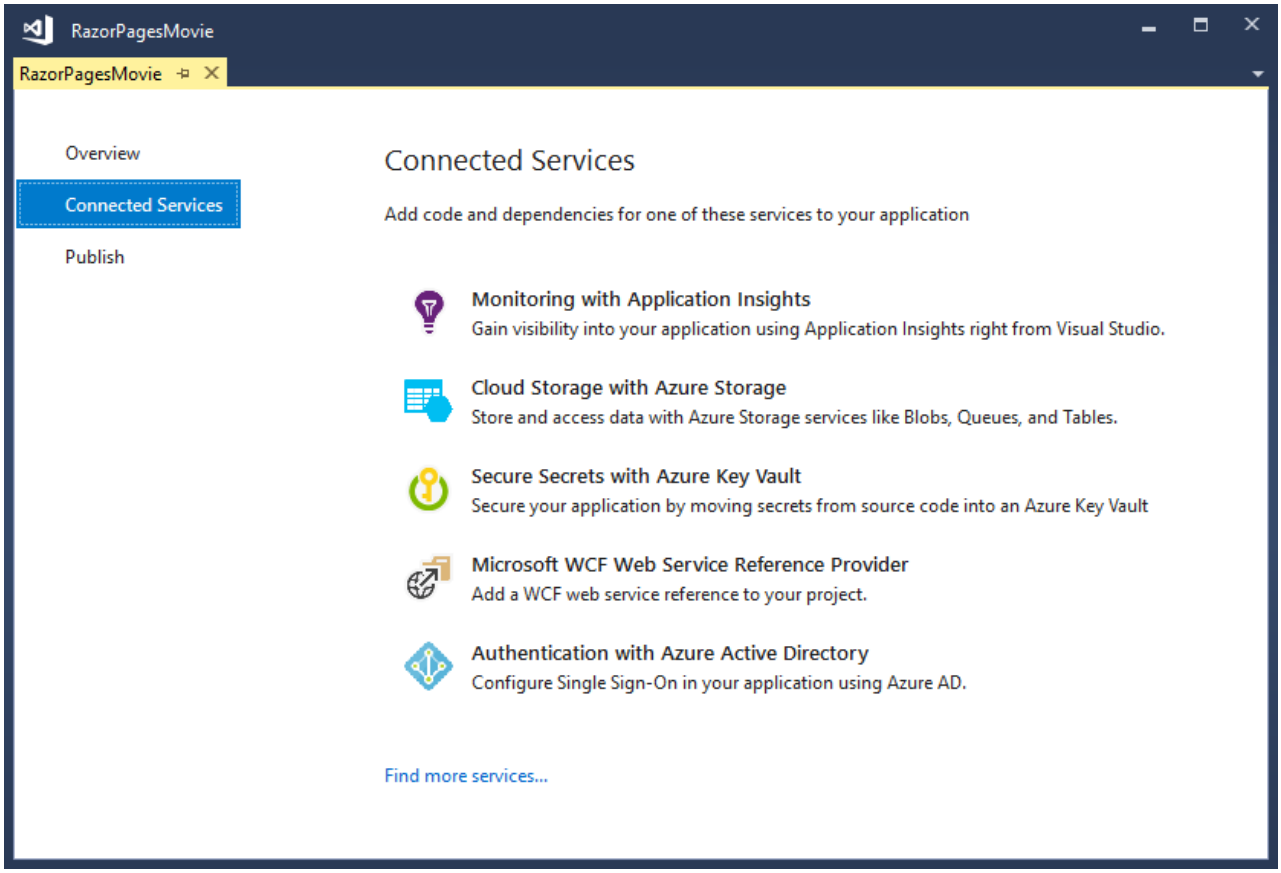
Visual Studio's Cloud Explorer lets you view and manage your Azure-based cloud resources within Visual Studio. These resources may include virtual machines, tables, SQL databases, and more. **Cloud Explorer** shows the Azure resources in all the accounts managed under the Azure subscription you're logged into. And if a particular operation requires the Azure portal, **Cloud Explorer** provides links that take you to the place in the portal where you need to go.



You can leverage Azure services for your apps using **Connected Services** such as:

- Active Directory connected service so users can use their accounts from Azure Active Directory 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 on the project in **Solution Explorer** and choosing **Add** > **Connected Service**.



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

Create apps for the web

The web drives our modern world, and Visual Studio can help you write apps for it. You can create web apps using ASP.NET, Node.js, Python, JavaScript, and TypeScript. Visual Studio understands web frameworks like Angular, jQuery, Express, and more. 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, and runs on Windows, Mac, and Linux. ASP.NET Core has been designed from the ground up to provide you with 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

You can use Visual Studio to build apps and games for macOS, Linux, and Windows, as well as for Android, iOS, and other mobile devices.

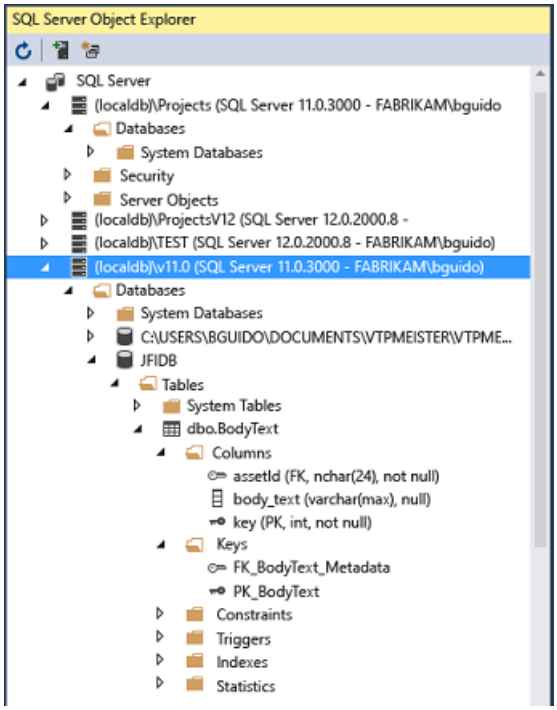
- Build .NET Core apps that run on Windows, macOS, and Linux.
- Build mobile apps for iOS, Android, and Windows in C# and F# by using Xamarin.
- Use standard web technologies—HTML, CSS, and JavaScript—to build mobile apps for iOS, Android, and Windows by using Apache Cordova.
- Build 2D and 3D games in C# by using Visual Studio Tools for Unity.
- Build native C++ apps for iOS, Android, and Windows devices. Share common code in libraries built for iOS, Android, and Windows, by using C++ for cross-platform development.
- Deploy, test, and debug Android apps with the Android emulator.

Connect to databases

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

SQL Server Data Tools (SSDT) is a powerful development environment for SQL Server, Azure SQL Database, and Azure SQL Data Warehouse. It enables you to build, debug, maintain, and refactor databases. You can work with a database project, or directly with a connected database instance on- or off-premises.

SQL Server Object Explorer in Visual Studio provides a view of your database objects similar to SQL Server Management Studio. SQL Server Object Explorer enables you to do light-duty database administration and design work. Work examples include editing table data, comparing schemas, executing queries by using contextual menus right from SQL Server Object Explorer, and more.



Debug, test, and improve your code

When you write code, you need to run it and test it for bugs and performance. Visual Studio's cutting-edge debugging system enables you to debug code running in your local project, on a remote device, or on a device emulator. You can step through code one statement at a time and inspect variables as you go. You can set breakpoints that are only hit when a specified condition is true. Debug options can be managed in the code editor itself, so that you don't have to leave your code. To get more details about debugging in Visual Studio, see First look at the debugger.

To learn more about improving the performance of your apps, checkout out Visual Studio's profiling feature.

For testing, Visual Studio offers unit testing, Live Unit Testing, IntelliTest, load and performance testing, and more. Visual Studio also has advanced code analysis capabilities to catch design, security, and other types of flaws.

Deploy your finished application

When your application is ready to deploy to users or customers, Visual Studio provides the tools to do that. Deployment options include to Microsoft Store, to a SharePoint site, or with InstallShield or Windows Installer technologies. It's all accessible through the IDE. For more information, see Deploy applications, services, and components.

Manage your source code and collaborate with others

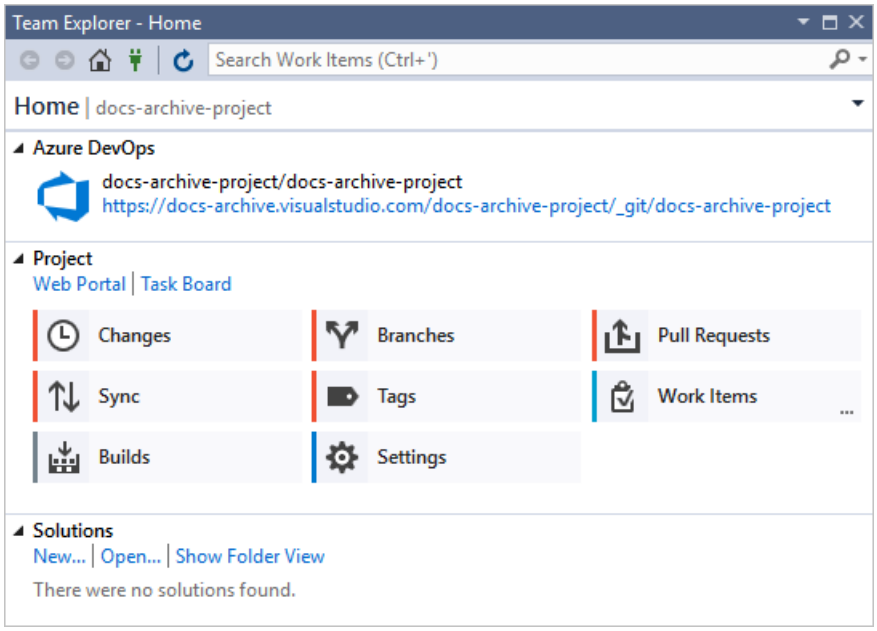
You can manage your source code in Git repos hosted by any provider, including GitHub. Or use Azure DevOps Services to manage code alongside bugs and work items for your whole project. See Get started with Git and Azure Repos to learn more about managing Git repos in Visual Studio using Team Explorer. Visual Studio also has other built-in source control features. To learn more about them, see New Git features in Visual Studio (blog).

Azure DevOps Services are cloud-based services to plan, host, automate, and deploy software and enable collaboration in teams. Azure DevOps Services support both Git repos (distributed version control) and Team Foundation Version Control (centralized version control). They support pipelines for continuous build and release (CI/CD) of code stored in version control systems. Azure DevOps Services also support Scrum, CMMI and Agile development methodologies.

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.

If you have an Azure DevOps organization or a Team Foundation Server on your network, you connect to it through the **Team Explorer** window in Visual Studio. From this window you can check code into or out of source control, manage work items, start builds, and access team rooms and workspaces. You can open **Team Explorer** from the search box, or on the main menu from **View** > **Team Explorer** or from **Team** > **Manage Connections**.

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



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

Extend Visual Studio

If Visual Studio doesn't have the exact functionality you need, you can add it! You can personalize the IDE based on your workflow and style, add support for external tools not yet integrated with Visual Studio, and modify existing functionality to increase your productivity. To find 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 as well as our 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

Tutorial: Create a simple C# console app in Visual Studio

5/5/2020 • 12 minutes to read • Edit Online

In this tutorial for C#, you'll use Visual Studio to create and run a console app and explore some features of the Visual Studio integrated development environment (IDE) while you do so.

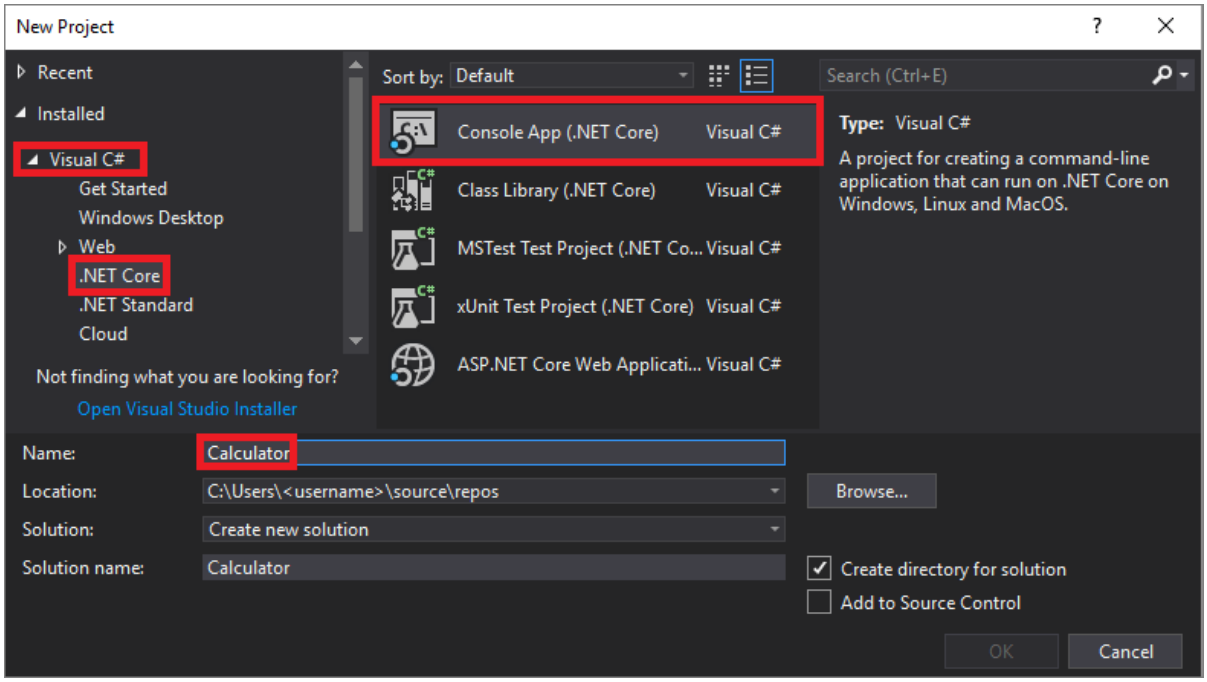
If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

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, we'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**. (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*.

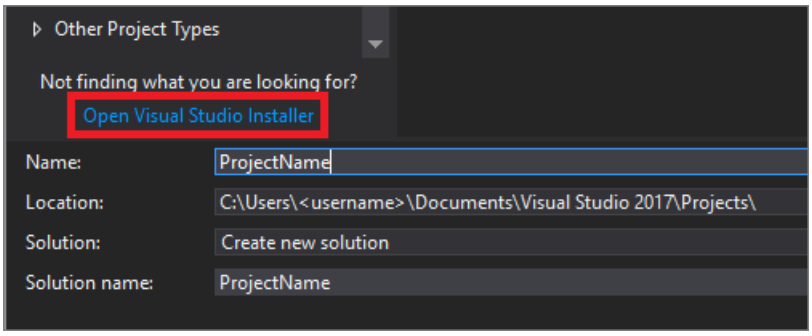


Add a workload (optional)

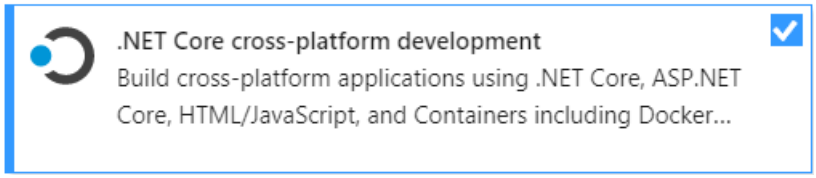
If you don't see the **Console App (.NET Core)** project template, you can get it by adding the **.NET Core cross-platform development** workload. Here's how.

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**.



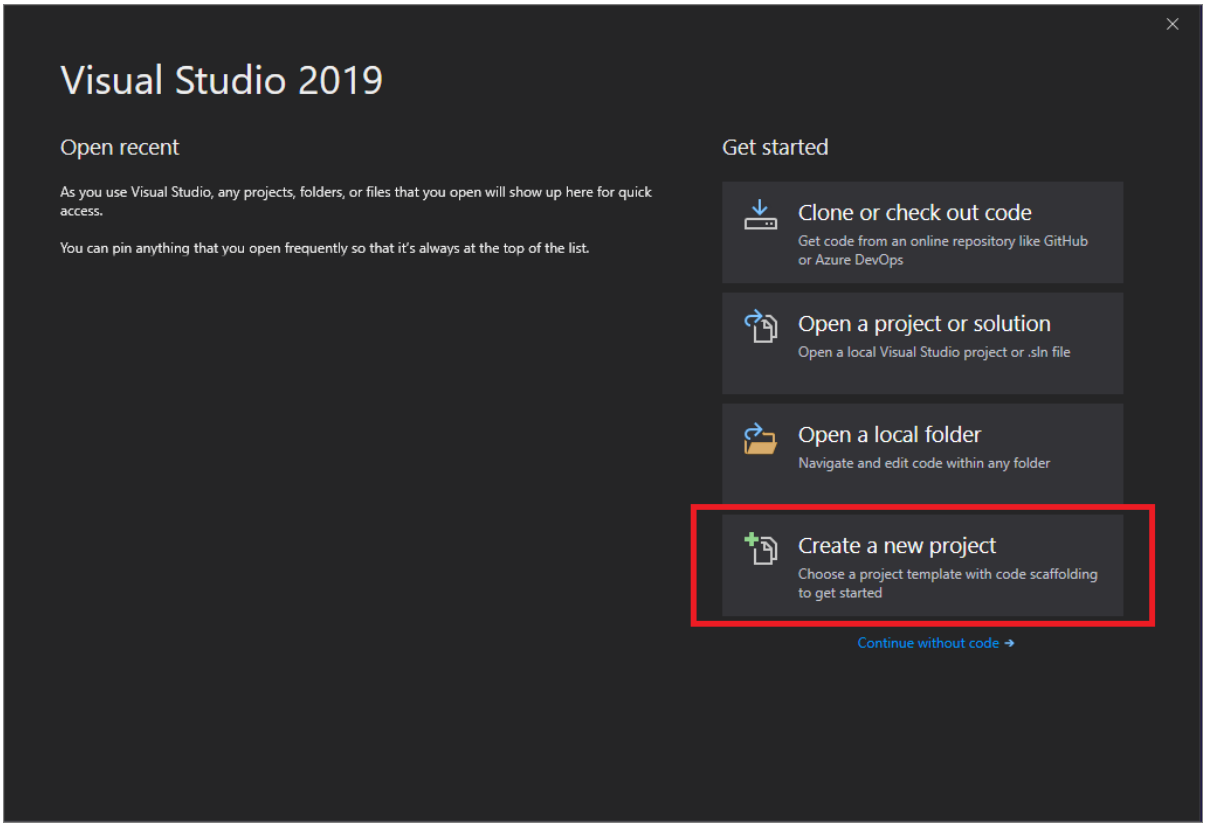
Option 2: Use the Tools menu bar

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**.

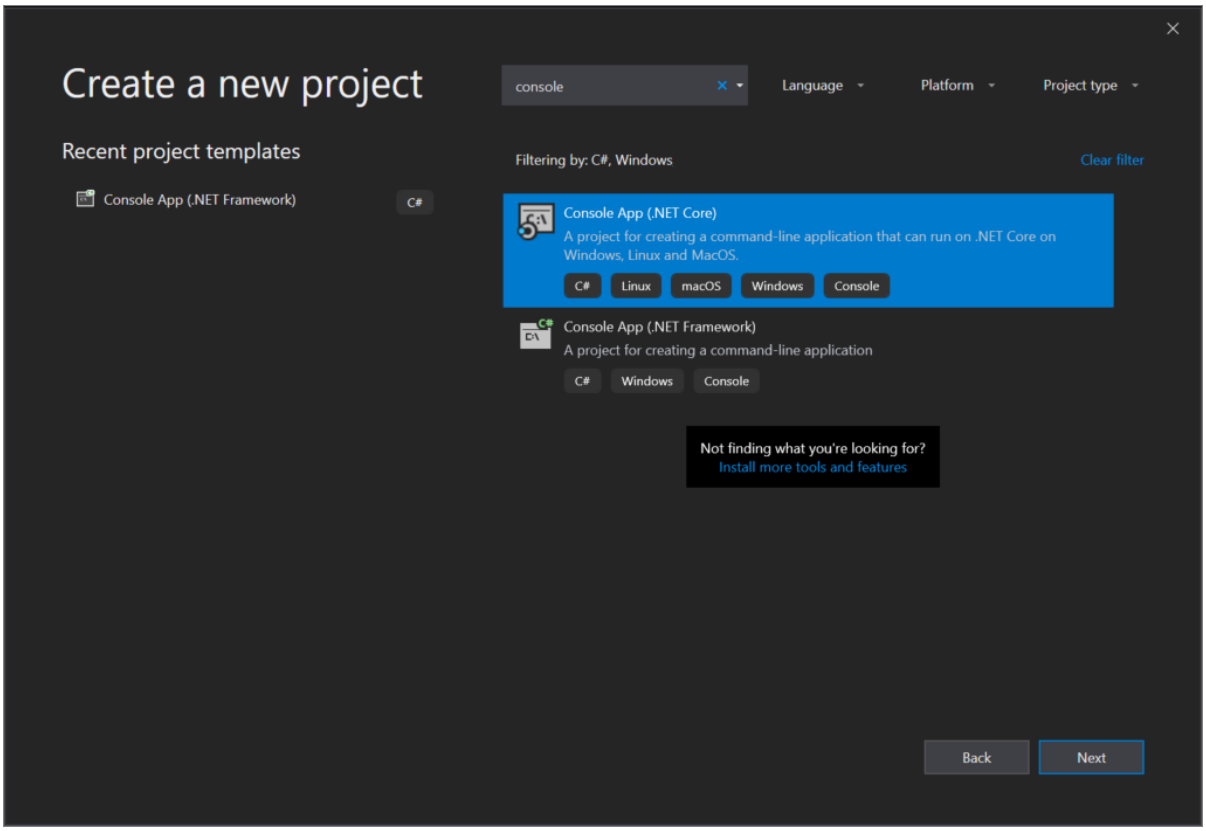
1. Open Visual Studio 2019.

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.

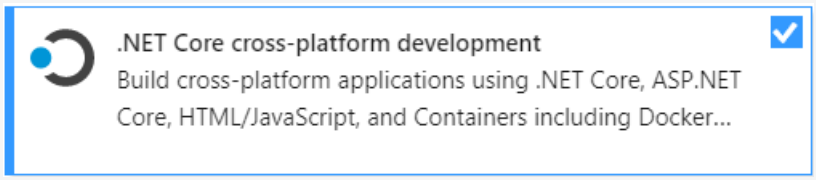
   After you apply the language and platform filters, choose the **Console App (.NET Core)** template, and then choose **Next**.



NOTE

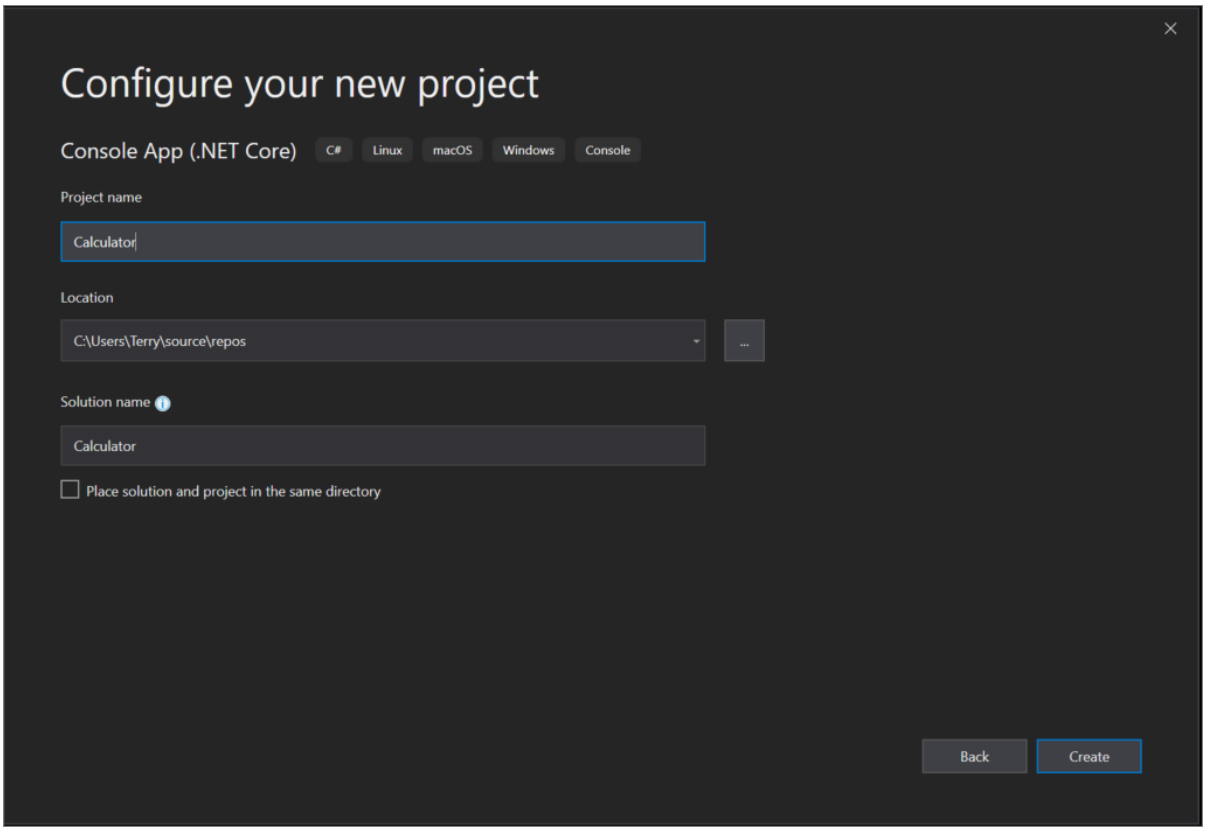
If you do not see the **Console App (.NET Core)** template, you can install it from the **Create a new project** window. In the **Not finding what you're looking for?** message, choose the **Install more tools and features** link.

Then, in the Visual Studio Installer, choose the **.NET Core cross-platform development** workload.



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 *Calculator* in the **Project name** box. Then, choose **Create**.



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

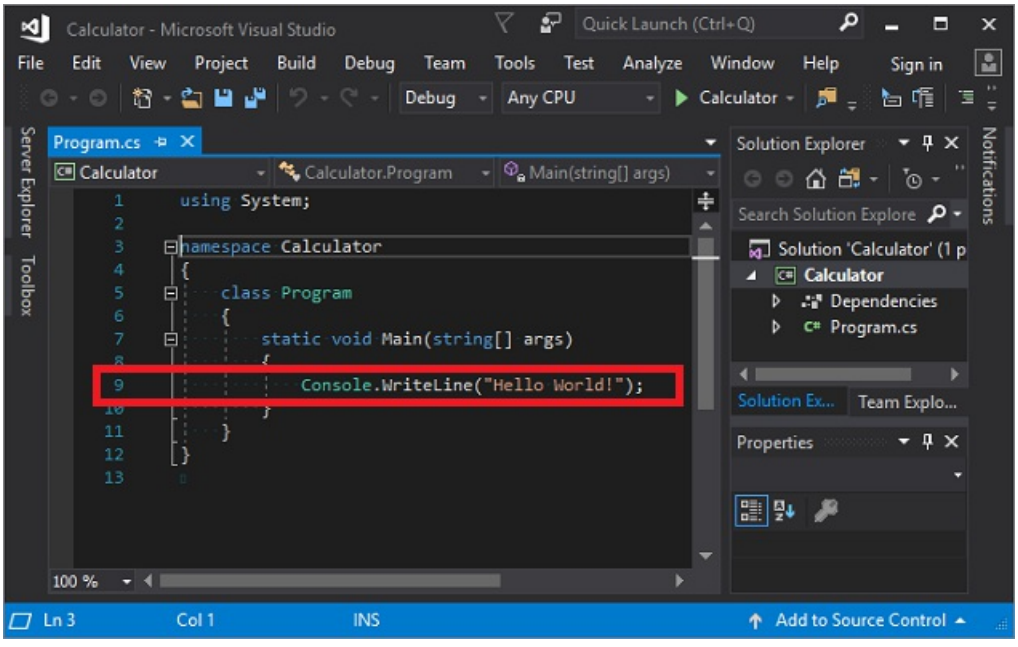
Create the app

First, we'll explore some basic integer math in C#. Then, we'll add code to create a basic calculator. After that, we'll debug the app to find and fix errors. And finally, we'll refine the code to make it more efficient.

Explore integer math

Let's start with some basic integer math in C#.

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



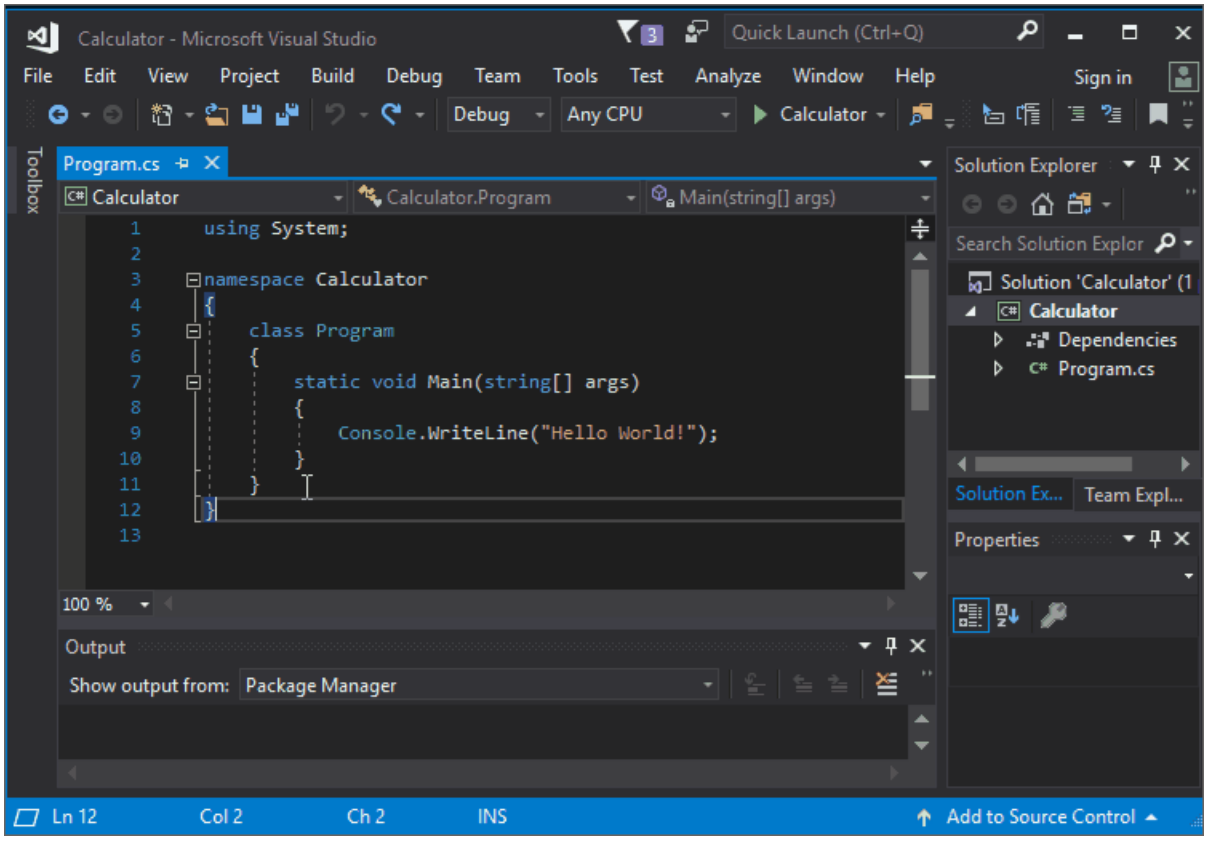
Specifically, delete the line that says, `Console.WriteLine("Hello World!");`.

2. In its place, type the following code:

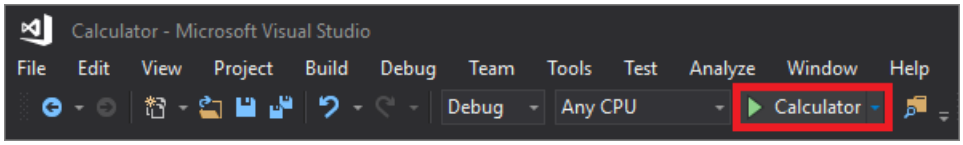
```
        int a = 42;
        int b = 119;
        int c = a + b;
        Console.WriteLine(c);
        Console.ReadKey();
```

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

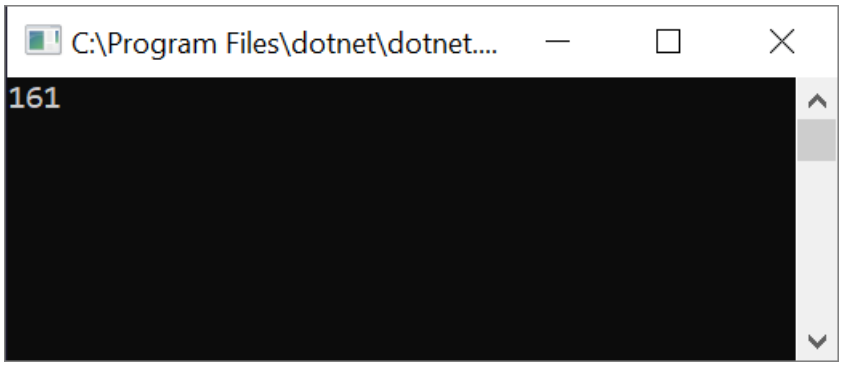
> **NOTE**
>
> The following animation isn't intended to duplicate the preceding code. It's intended only to show how the autocomplete feature works.



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.

Add code to create a calculator

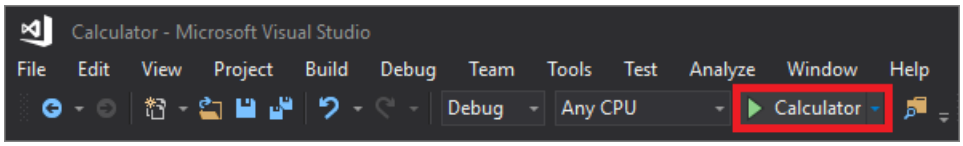
Let's continue by adding a more complex set of calculator code to your project.

1. Delete all the code you see in the code editor.

2. Enter or paste the following new code into the code editor:

```
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();
        }
    }
}
```

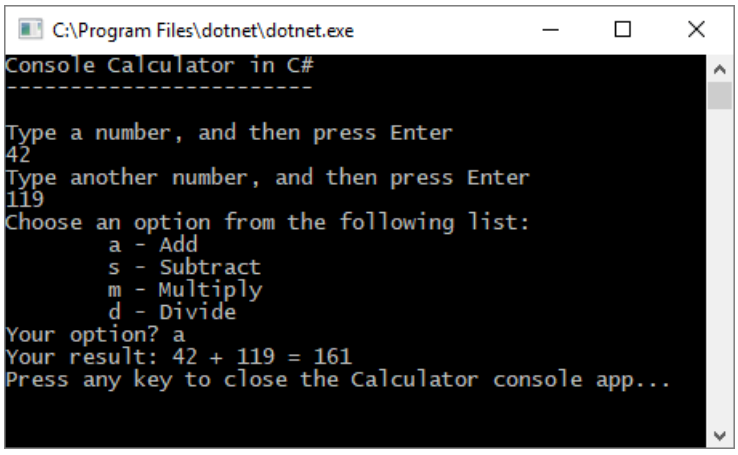
3. Choose **Calculator** to run your program, or press **F5**.



A console window opens.

4. View your app in the console window, and then follow the prompts to add the numbers **42** and **119**.

Your app should look similar to the following screenshot:



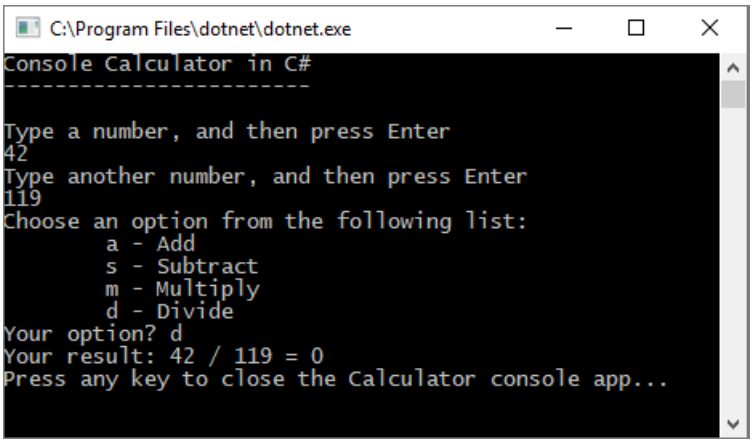
Add functionality to the calculator

Let's tweak the code to add further functionality.

Add decimals

The calculator app currently accepts and returns whole numbers. But, it will be more precise if we add code that allows for decimals.

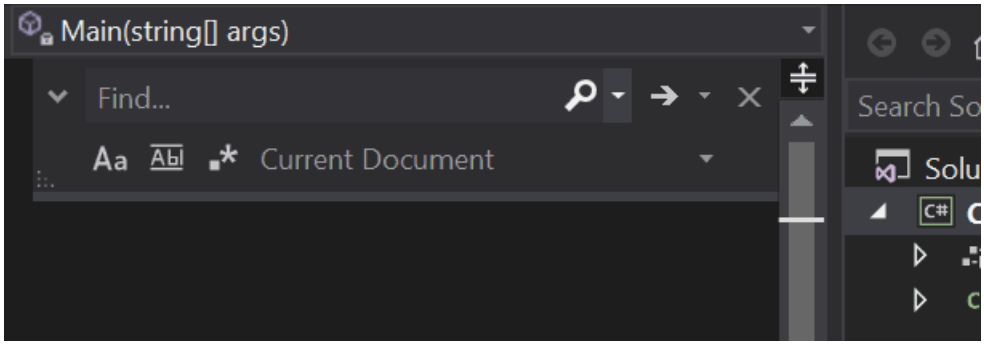
As in the following screenshot, if you run the app and divide number 42 by the number 119, your result is 0 (zero), which isn't exact.



Let's fix the code so that it handles decimals.

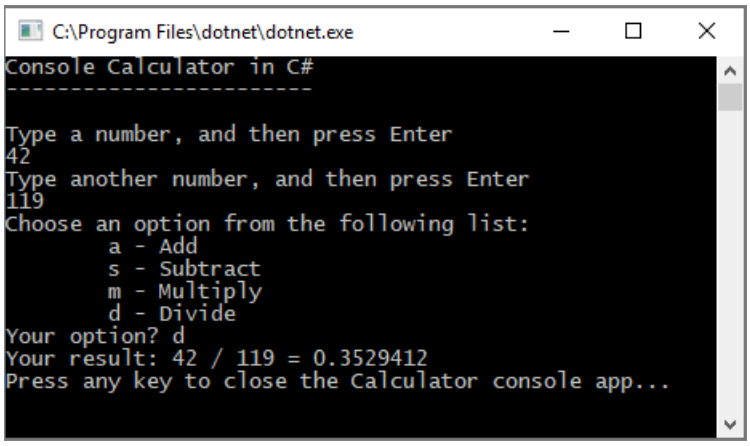
1. Press **Ctrl** + **H** to open the **Find and Replace** control.
2. Change each instance of the `int` variable to `float`.

   Make sure that you toggle **Match case** (**Alt**+**C**) and **Match whole word** (**Alt**+**W**) in the **Find and Replace** control.



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

   Notice that the app now returns a decimal numeral instead of zero.

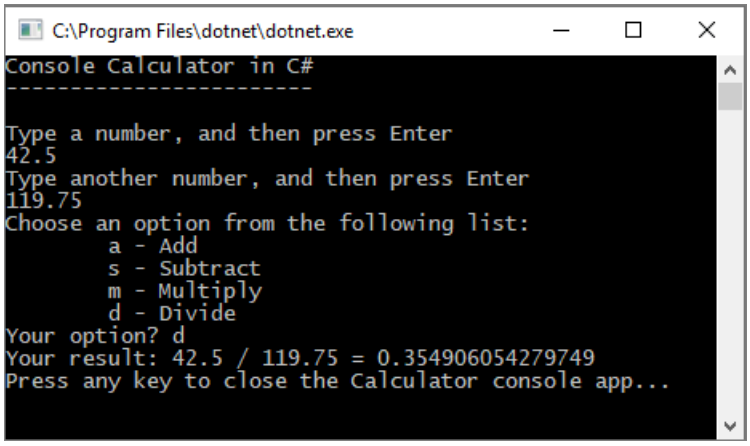


However, the app produces only a decimal result. Let's make a few more tweaks to the code so that the app can calculate decimals too.

1. Use the **Find and Replace** control (**Ctrl** + **H**) 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**.

   Notice that the app now accepts decimal values and returns a longer decimal numeral as its result.



   (We'll fix the number of decimal places in the Revise the code section.)

Debug the app

We've improved on our basic calculator app, but it doesn't yet have fail safes in place to handle exceptions, such as user input errors.

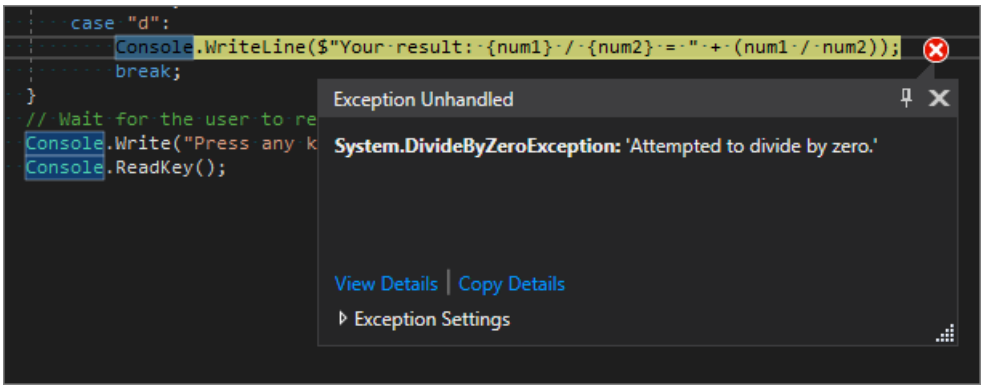
For example, if you try to divide a number by zero, or enter an alpha character when the app expects a numeric character (or vice versa), the app might stop working, return an error, or return an unexpected nonnumeric result.

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 the First look at the Visual Studio debugger page.

Fix the "divide by zero" error

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



> **NOTE**
>
> Sometimes, the app doesn't freeze and the debugger won'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.

Let's change the code to handle this error.

1. Delete the code that appears directly between `case "d":` and the comment that says `// Wait for the user to respond before closing`.

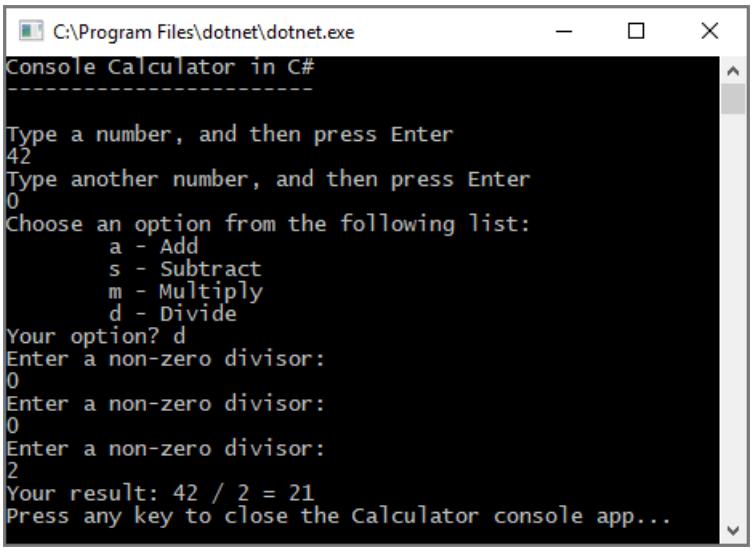
2. Replace it with the following code:

```
            // 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 add the code, the section with the `switch` statement should look similar to the following screenshot:

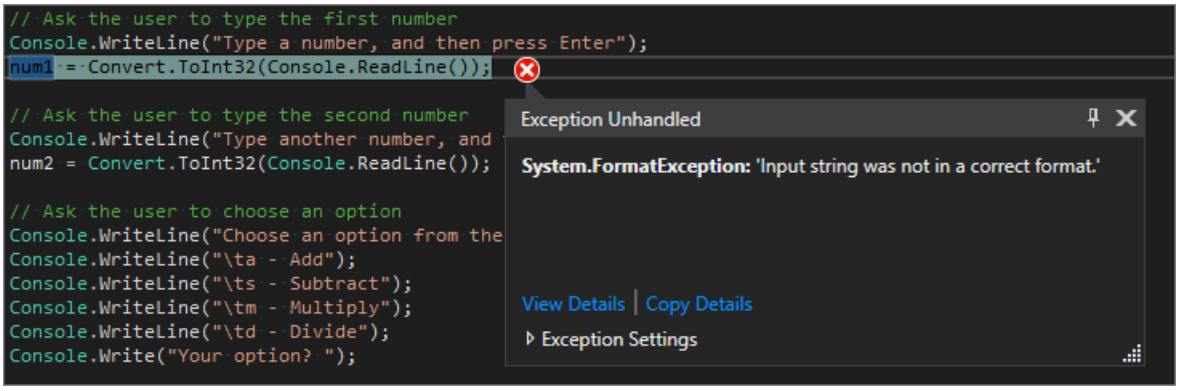
```
// Use a switch statement to do the math
switch (Console.ReadLine())
{
    case "a":
        Console.WriteLine($"Your result: {num1} + {num2} = " + (num1 + num2));
        break;
    case "s":
        Console.WriteLine($"Your result: {num1} - {num2} = " + (num1 - num2));
        break;
    case "m":
        Console.WriteLine($"Your result: {num1} * {num2} = " + (num1 * num2));
        break;
    case "d":
        // Ask the user to enter a non-zero divisor until they do so
        while (num2 == 0)
        {
            Console.WriteLine("Enter a non-zero divisor: ");
            num2 = Convert.ToInt32(Console.ReadLine());
        }
        Console.WriteLine($"Your result: {num1} / {num2} = " + (num1 / num2));
        break;
}
// Wait for the user to respond before closing
Console.Write("Press any key to close the Calculator console app...");
Console.ReadKey();
```

Now, when you divide any number by zero, the app will ask for another number. Even better: It won't stop asking until you provide a number other than zero.



Fix the "format" error

If you enter an alpha character when the app expects a numeric character (or vice versa), the console app freezes. Visual Studio then shows you what's wrong in the code editor.



To fix this error, we must refactor the code that we've previously entered.

Revise the code

Rather than rely on the `program` class to handle all the code, we'll divide our app into two classes: `Calculator` and `Program`.

The `Calculator` class will handle the bulk of the calculation work, and the `Program` class will handle the user interface and error-capturing work.

Let's get started.

1. Delete everything in the `Calculator` namespace between its opening and closing braces:

```
using System;

namespace Calculator
{

}
```

2. Next, add a new `Calculator` class, as follows:

```
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;
    }
}
```

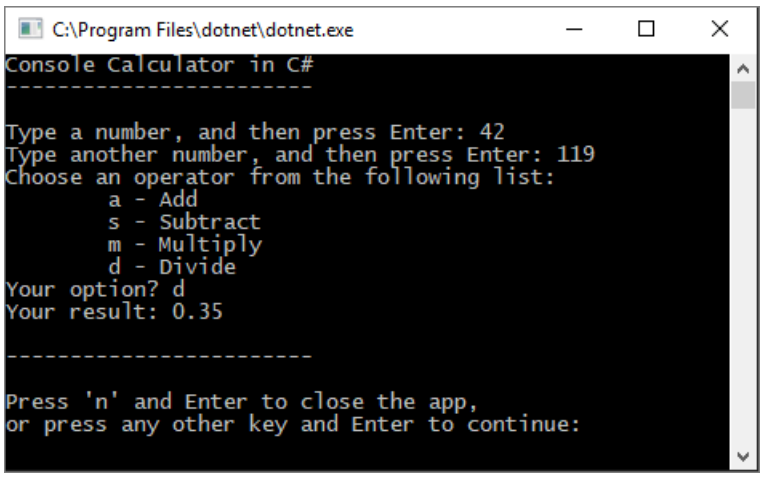
3. Then, add a new `Program` class, as follows:

```
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. Choose **Calculator** to run your program, or press **F5**.

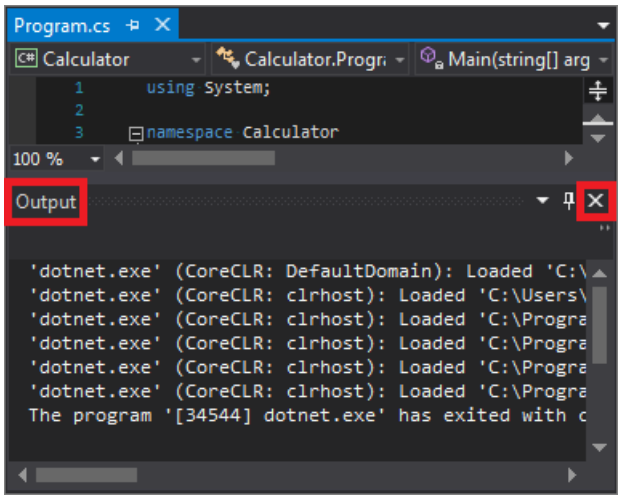
5. Follow the prompts and divide the number **42** by the number **119**. Your app should look similar to the following screenshot:



Notice that you have the option to enter more equations until you choose to close the console app. And, we've also reduced the number of decimal places in the result.

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.
4. Close Visual Studio.

Code complete

During this tutorial, we've made a lot of changes to the calculator app. The app now handles computing resources more efficiently, and it 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)
```

```
                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

Congratulations on completing this tutorial! To learn even more, continue with the following tutorials.

Continue with more C# tutorials

See also

- C# IntelliSense
- Learn to debug C# code in Visual Studio

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

2/25/2020 • 10 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.

Before you begin

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.

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.

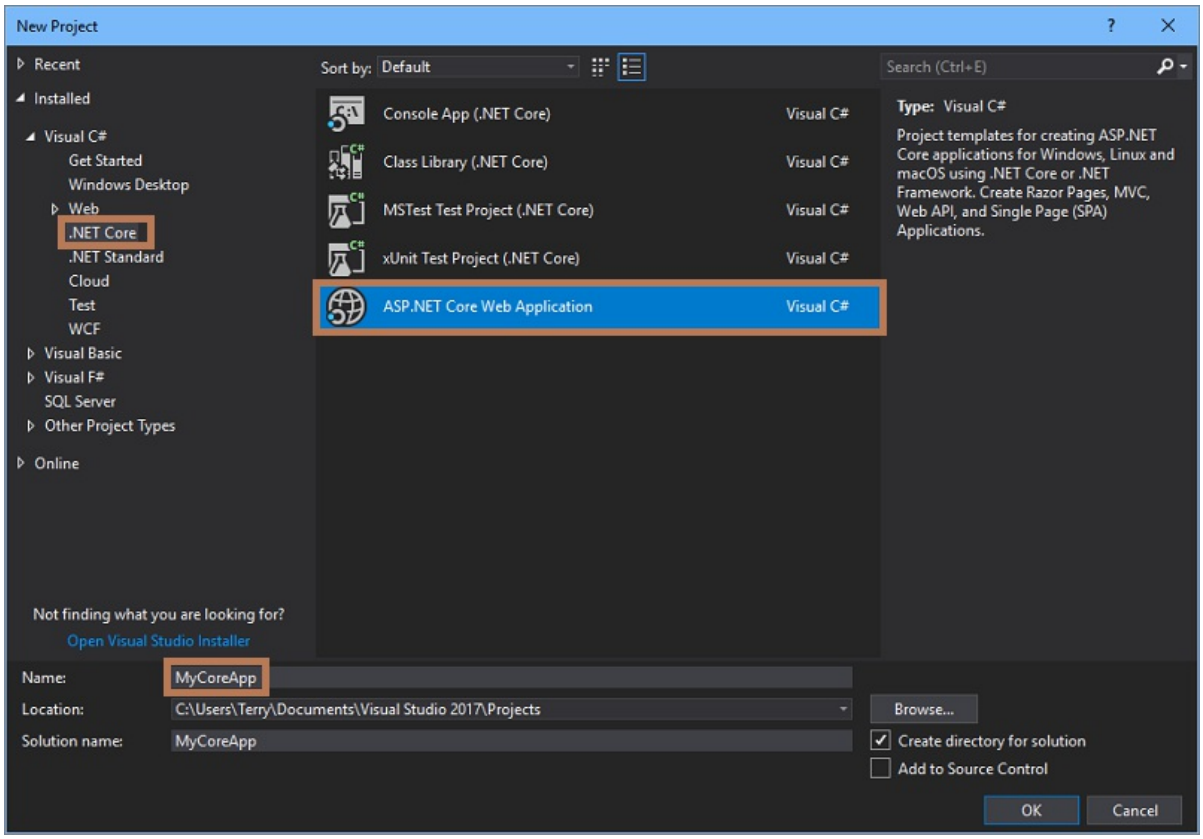
Choose your theme (optional)

This tutorial includes screenshots that 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 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**.

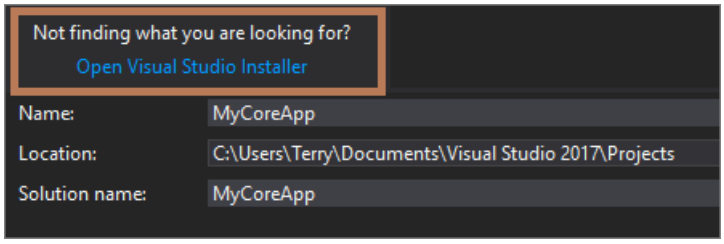


Add a workload (optional)

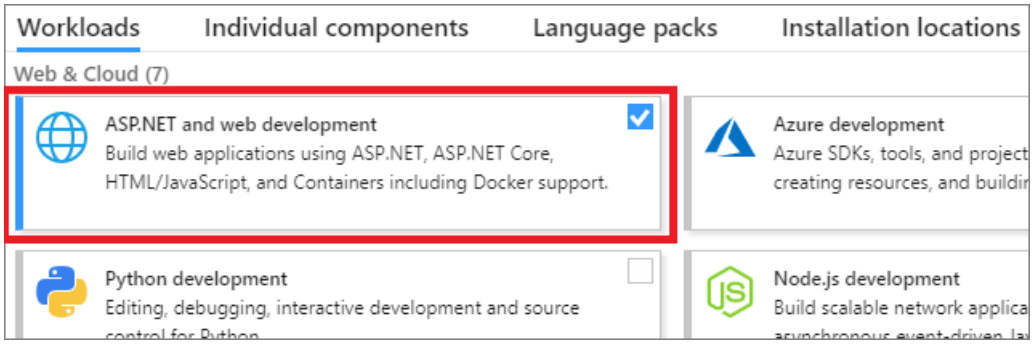
If you don't see the **ASP.NET Core Web Application** project template, you can get it by adding the **ASP.NET and web development** workload. You can add this workload in one of the two following ways, depending on which Visual Studio 2017 updates are installed on your machine.

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**

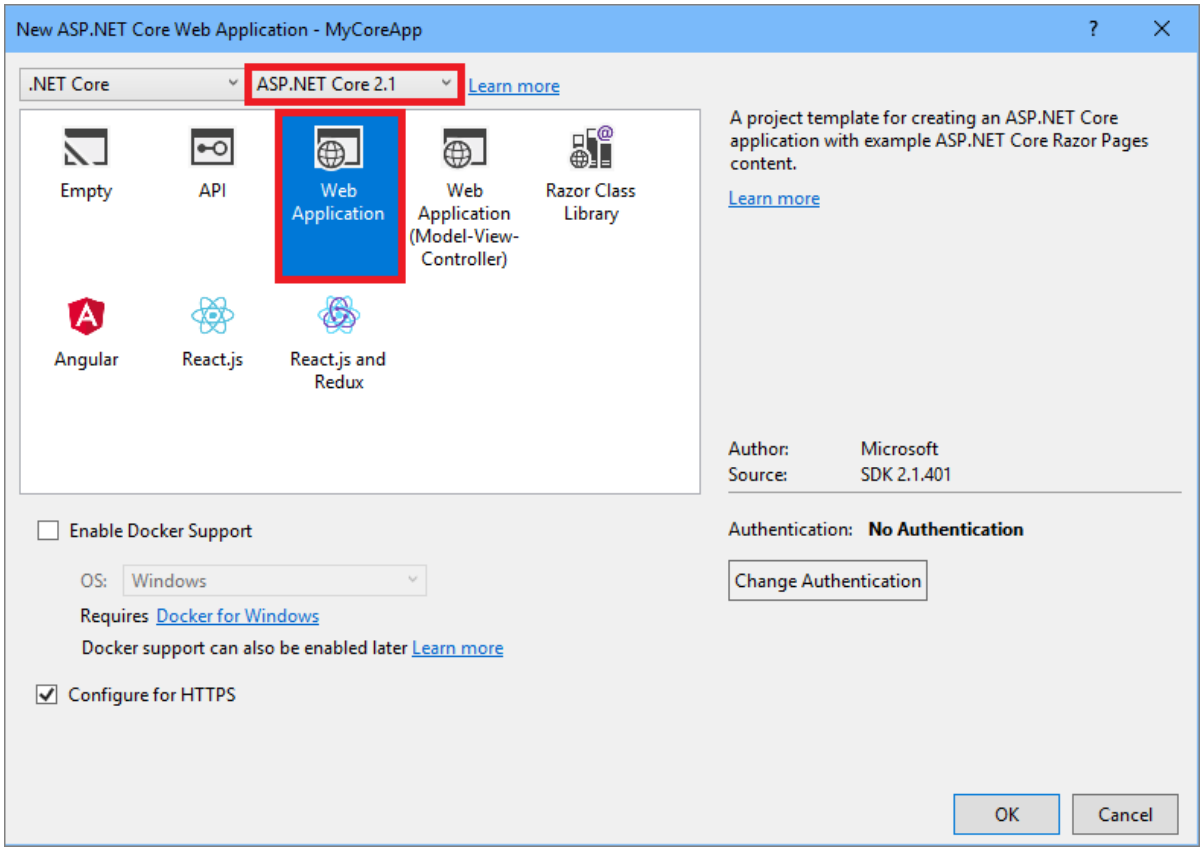
Features.

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.)

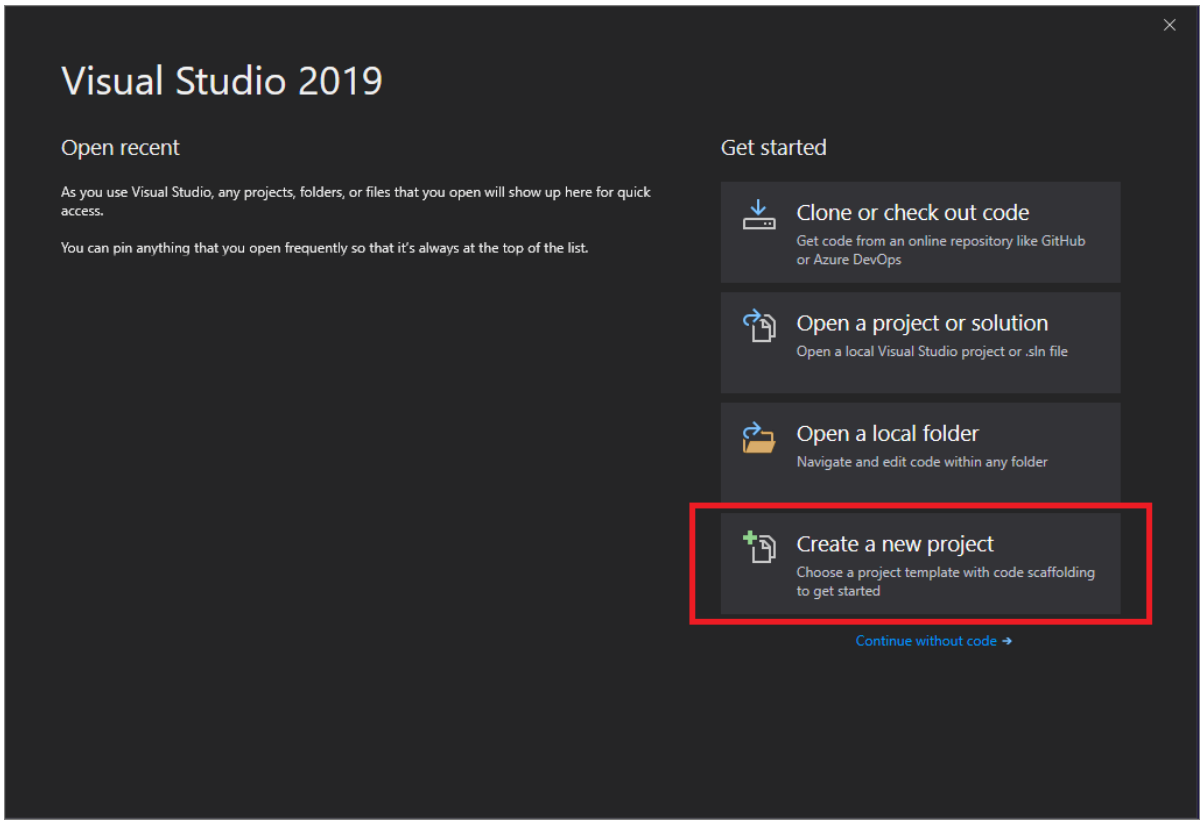
Add a project template

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**.



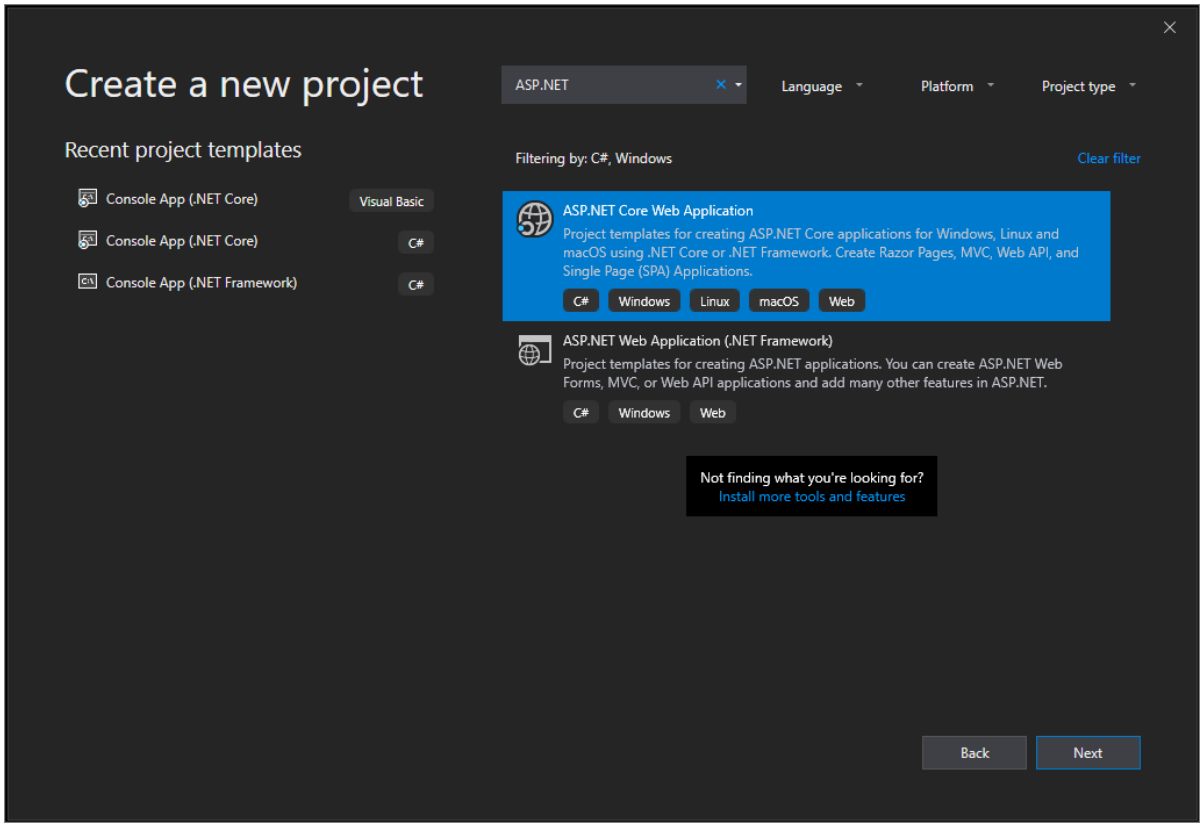
> **NOTE**
>
> If you don't see **ASP.NET Core 2.1** from the top drop-down menu, make sure that you are running the most recent release of Visual Studio. For more information about how to update your installation, see the Update Visual Studio to the most recent release page.

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



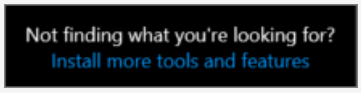
2. On the **Create a new project** window, enter or type *ASP.NET* in the search box. Next, choose **C#** from the Language list, and then choose **Windows** from the Platform list.

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

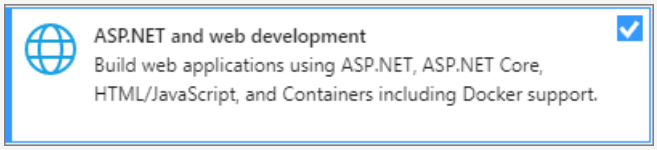


NOTE

If you don't see the **ASP.NET Core Web 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.

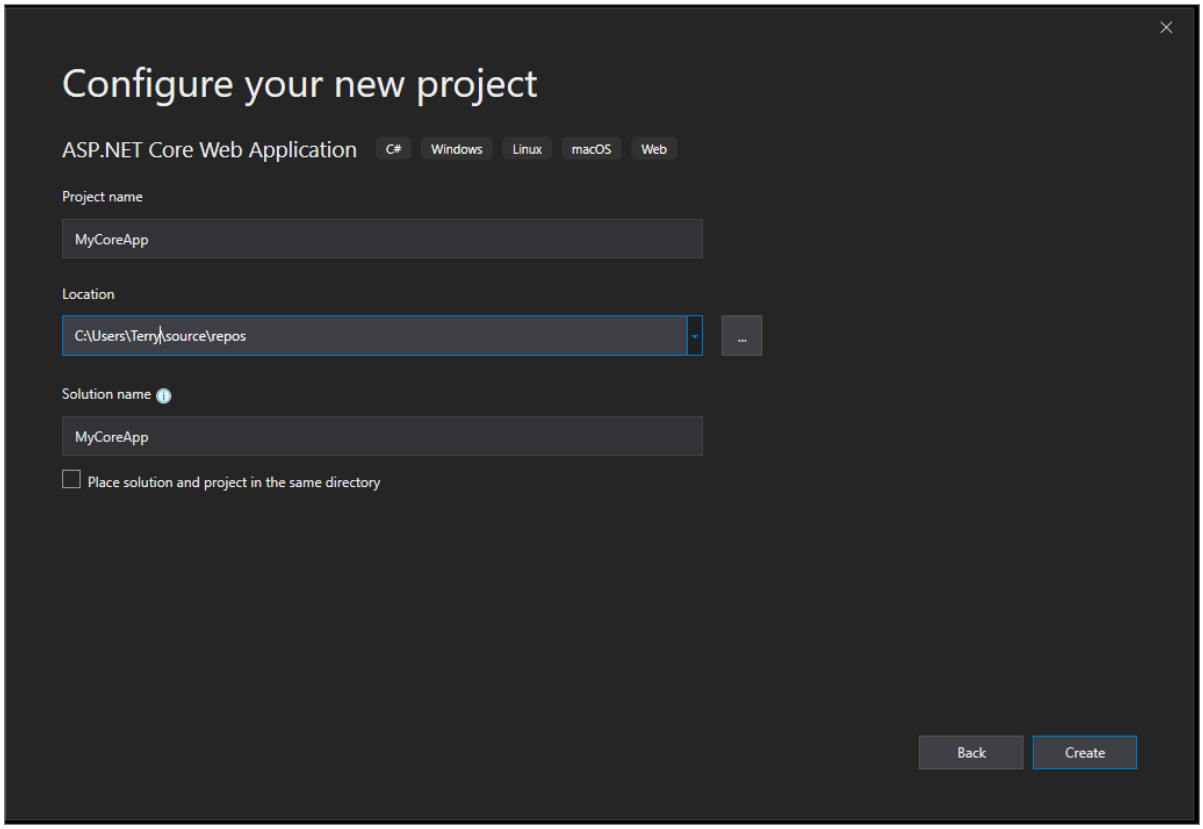


Then, in the Visual Studio Installer, choose the **ASP.NET and web development** workload.

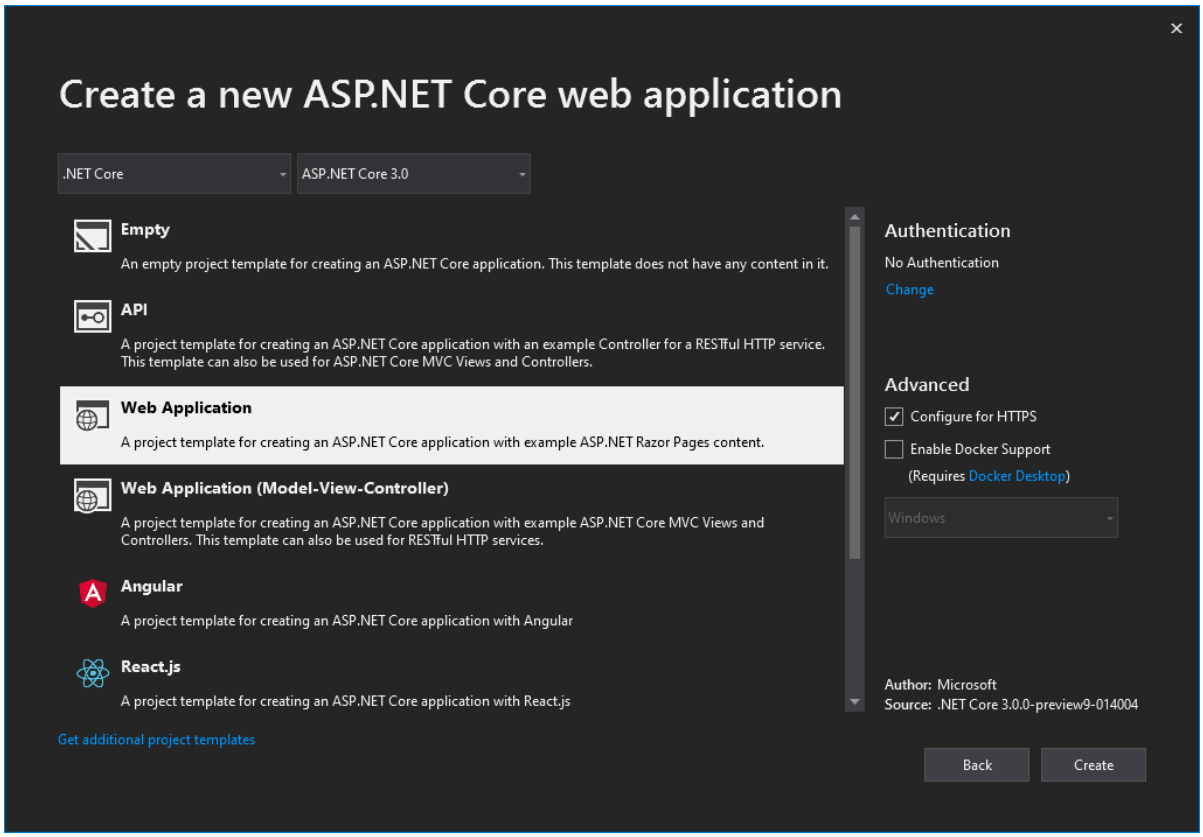


After that, choose the **Modify** button in the Visual Studio Installer. If you're prompted to save your work, do so. Next, choose **Continue** to install the workload. Then, return to step 2 in this "Create a project" procedure.

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



4. In the **Create a new ASP.NET Core Web Application** window, verify that **ASP.NET Core 3.0** appears in the top drop-down menu. Then, choose **Web Application**, which includes example Razor Pages. Next, choose **Create**.



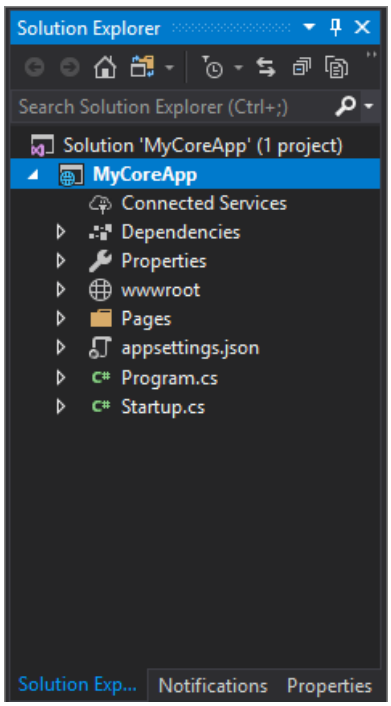
Visual Studio opens your new project.

About your solution

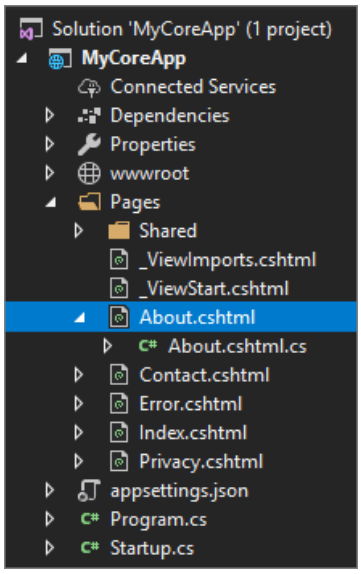
This solution follows the **Razor Page** design pattern. It's different than the Model-View-Controller (MVC) design pattern in that it's streamlined to include the model and controller code within the Razor Page itself.

Tour your solution

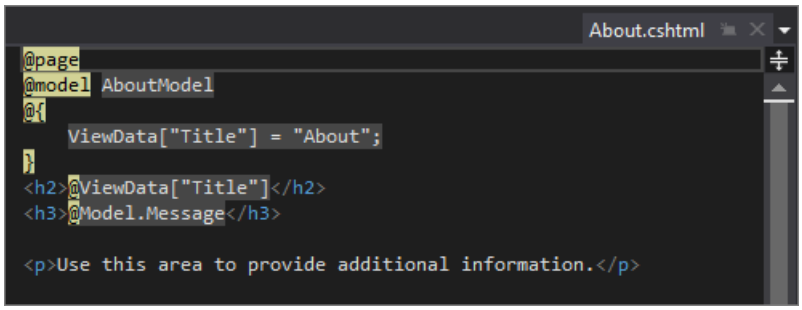
1. The project template creates a solution with a single ASP.NET Core project that is named *MyCoreApp*. Choose the **Solution Explorer** tab to view its contents.



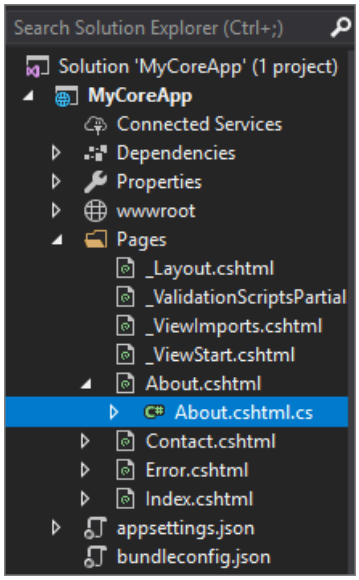
2. Expand the **Pages** folder, and then expand *About.cshtml*.



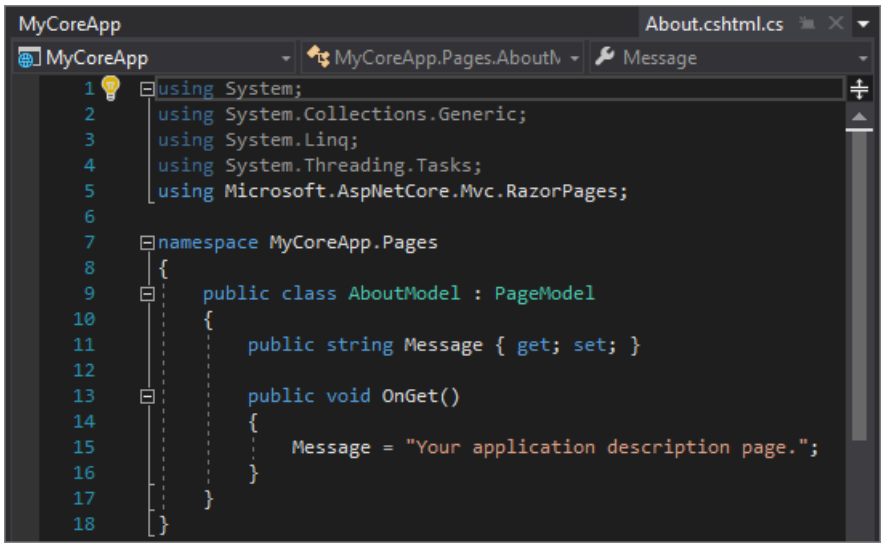
3. View the **About.cshtml** file in the code editor.



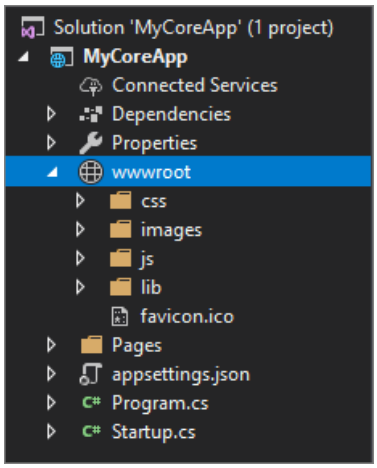
4. Choose the **About.cshtml.cs** file.



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

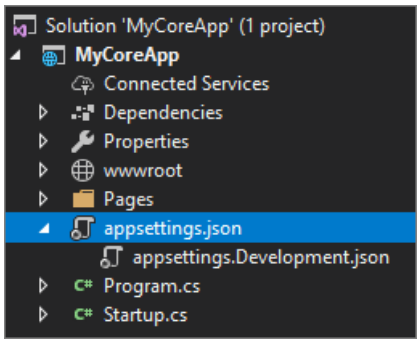


6. The project contains a **wwwroot** folder that is the root for your website. Expand the folder to view its contents.



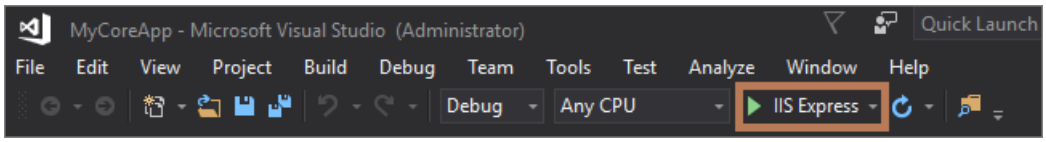
You can put static site content—such as CSS, images, and JavaScript libraries—directly in the paths where you want them.

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.



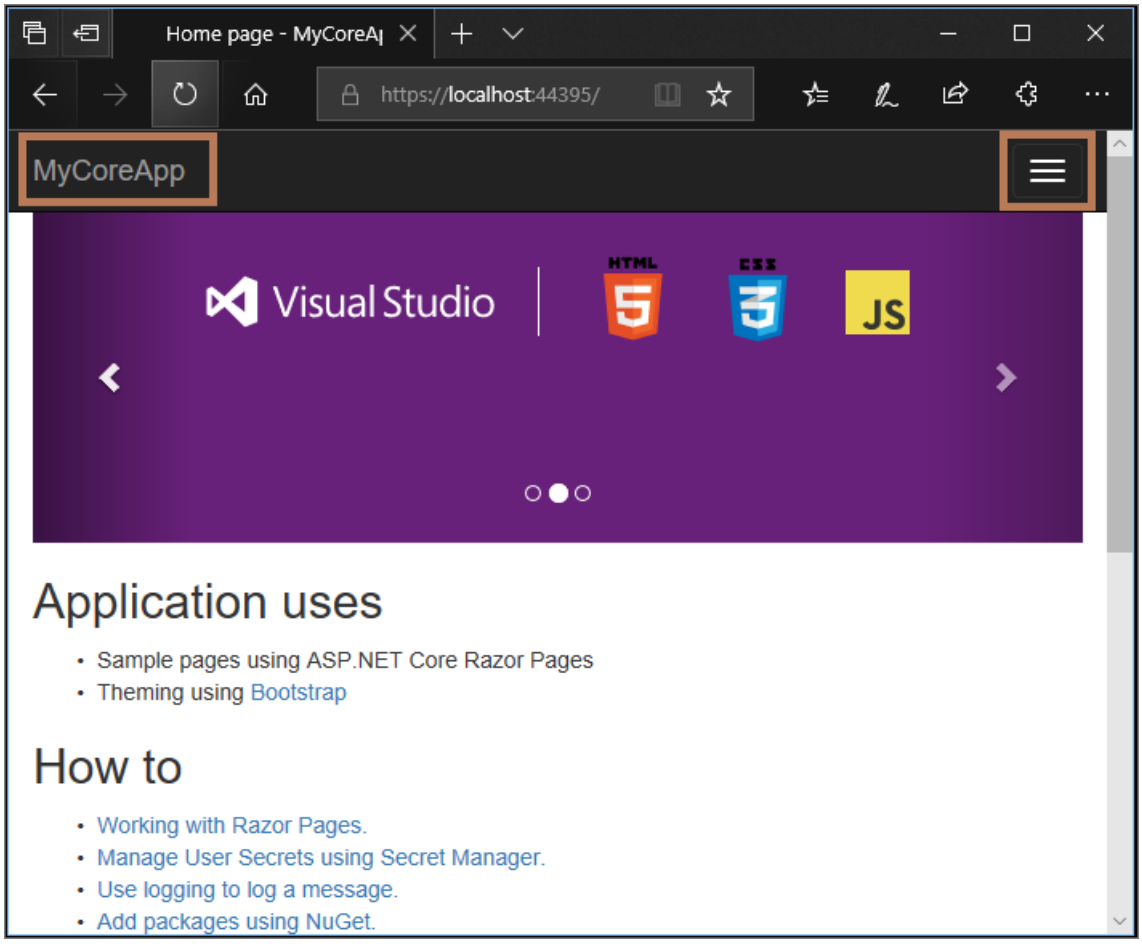
Run, debug, and make changes

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.)

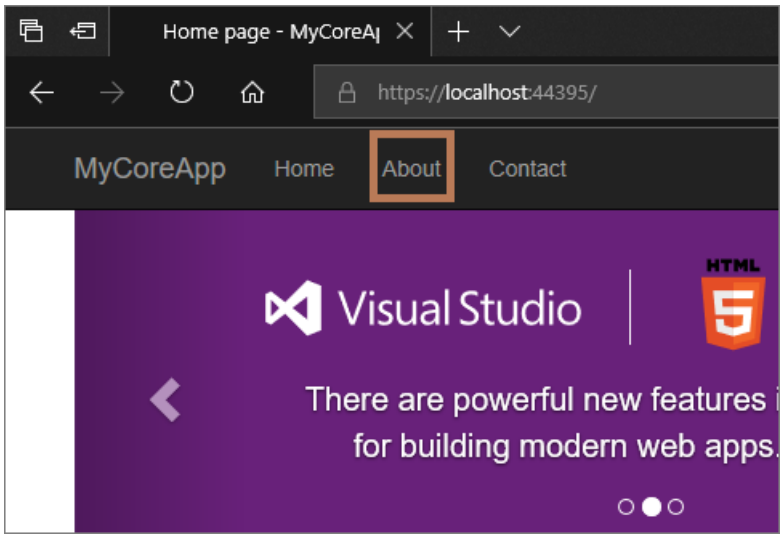


> **NOTE**
>
> If you get an error message that says **Unable to connect to web server 'IIS Express'**, close Visual Studio and then open it by using the **Run as administrator** option from the right-click or context menu. Then, run the application again.
>
> You might also get a message that asks if 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.

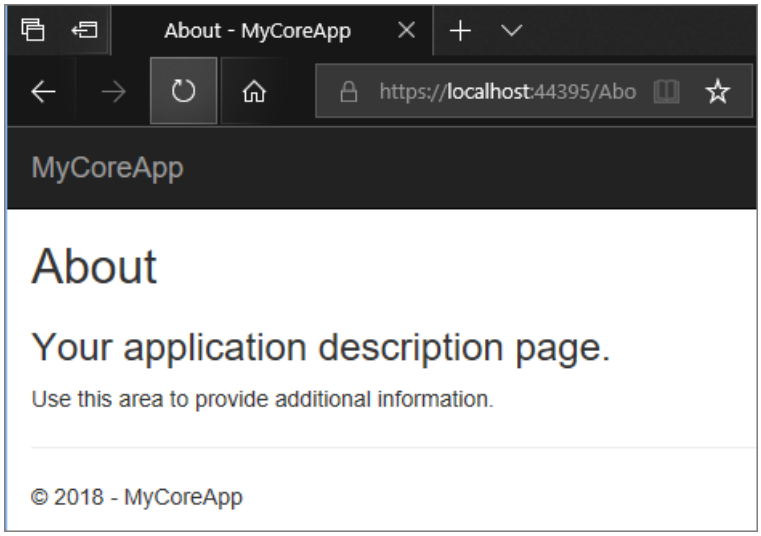
2. Visual Studio launches a browser window. You should then see **Home**, **About**, and **Contact** pages in the menu bar. (If you don't, choose the "hamburger" menu item to view them.)



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

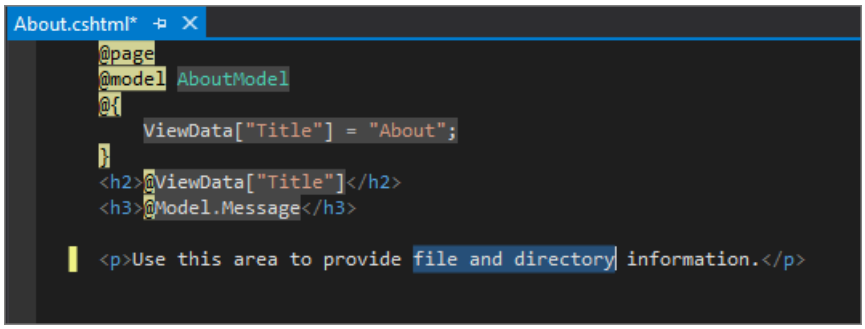


Among other things, the **About** page in the browser renders the text that is set in the *About.cshtml* file.



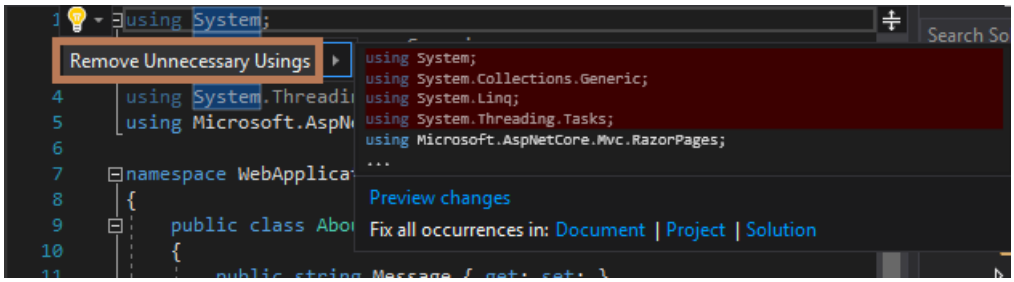
4. Return to Visual Studio, and then press **Shift+F5** to stop Debug mode. This also closes the project in the browser window.

5. In Visual Studio, choose **About.cshtml**. Then, delete the word *additional* 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**.

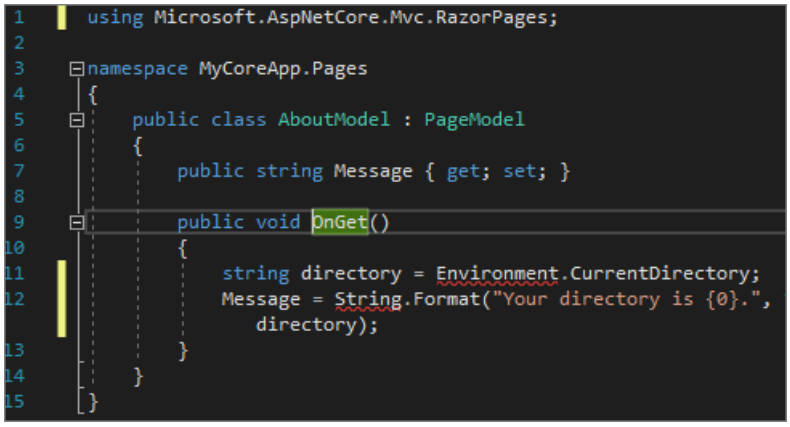


Visual Studio deletes the unnecessary `using` directives from the file.

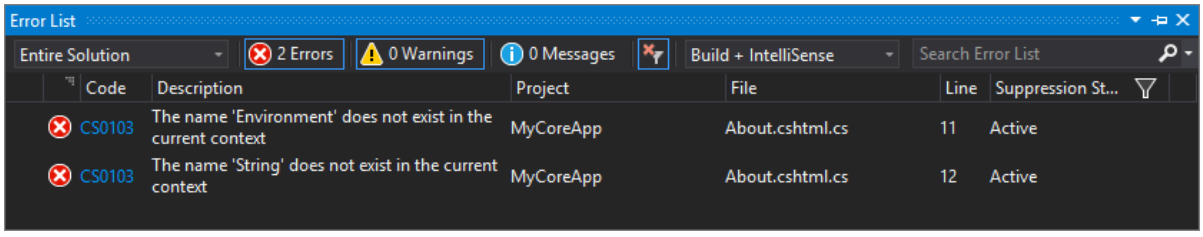
7. Next, in the `OnGet()` method, change the body to the following code:

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

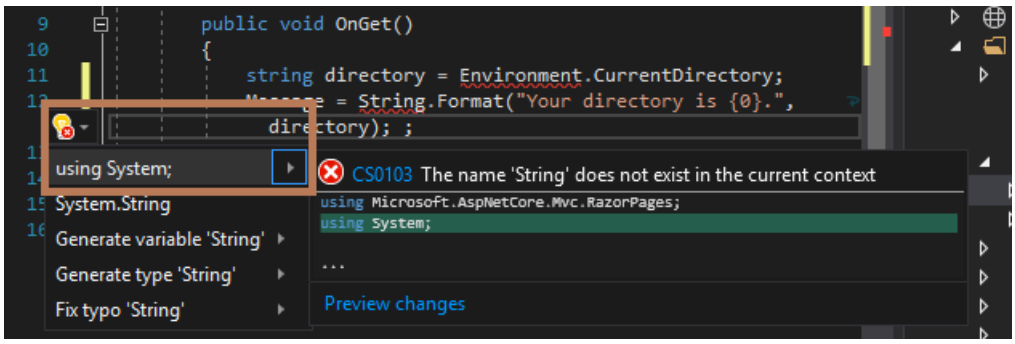
8. Notice that two wavy underlines appear under **Environment** and **String**. The wavy underlines appear because these types aren't in scope.



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.)

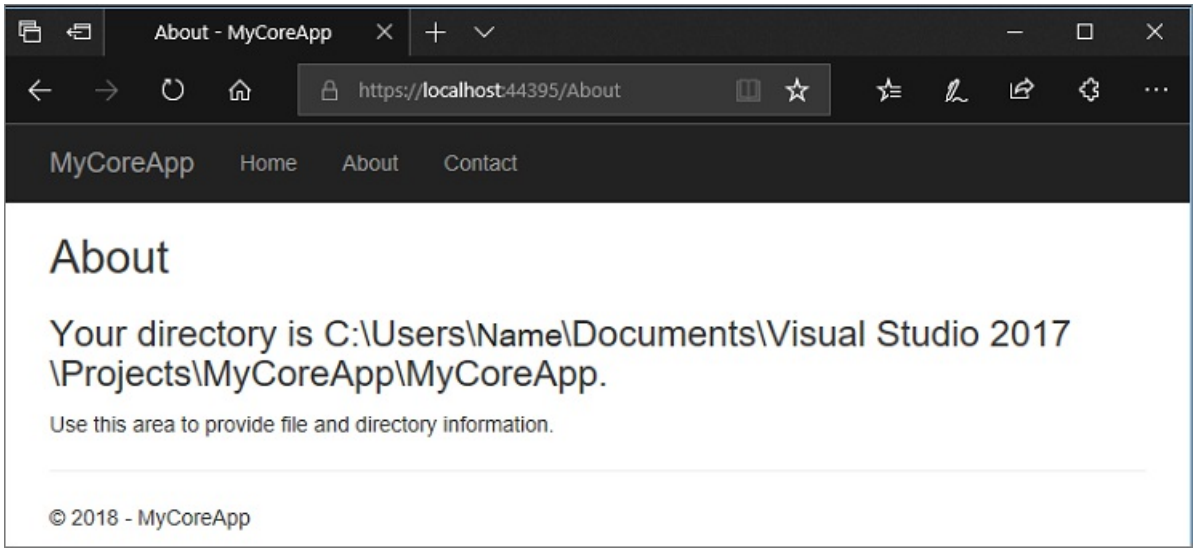


9. Let's fix this. In the code editor, place your cursor on either line that contains the error, and then choose the Quick Actions light bulb in the left margin. Then, from the drop-down menu, choose **using System;** to add this directive to the top of your file and resolve the errors.



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

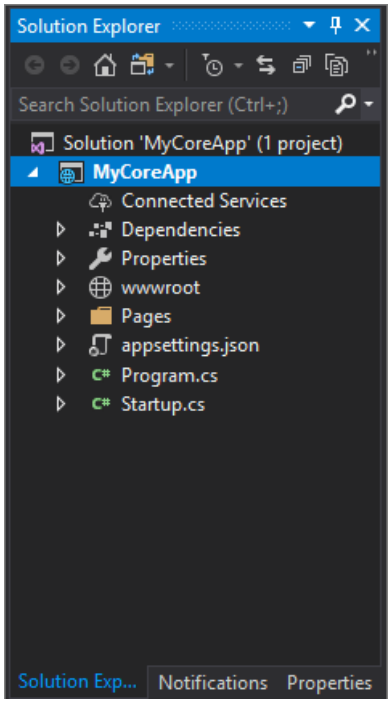
11. At the top of the web site, choose **About** to view your changes.



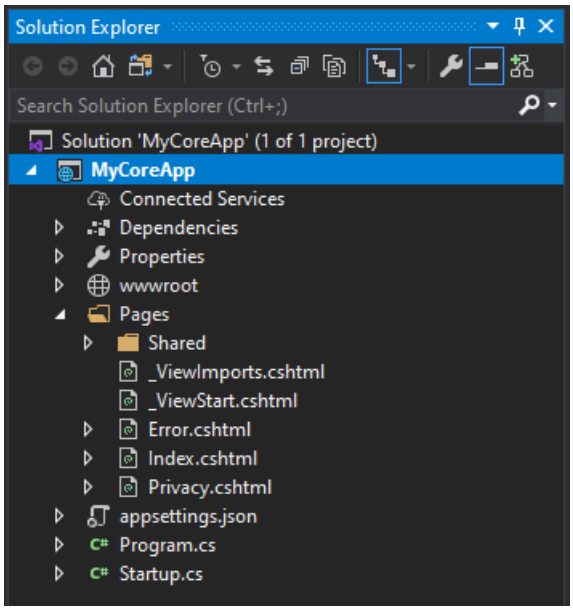
12. Close the web browser, press **Shift**+**F5** to stop Debug mode, and then close Visual Studio.

Tour your solution

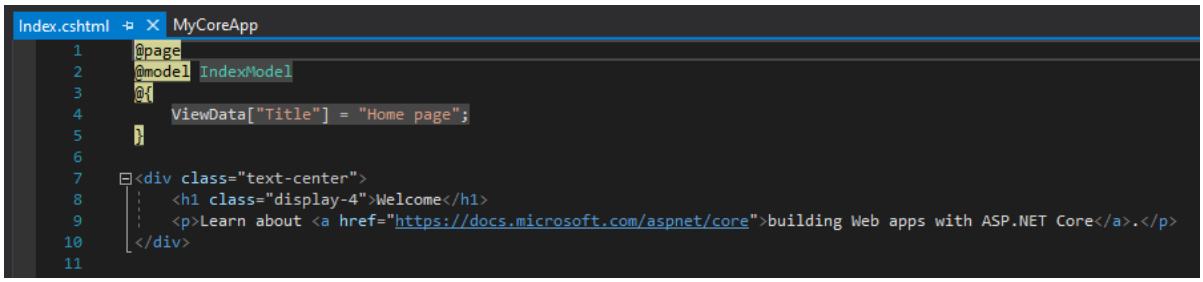
1. The project template creates a solution with a single ASP.NET Core project that is named *MyCoreApp*. Choose the **Solution Explorer** tab to view its contents.



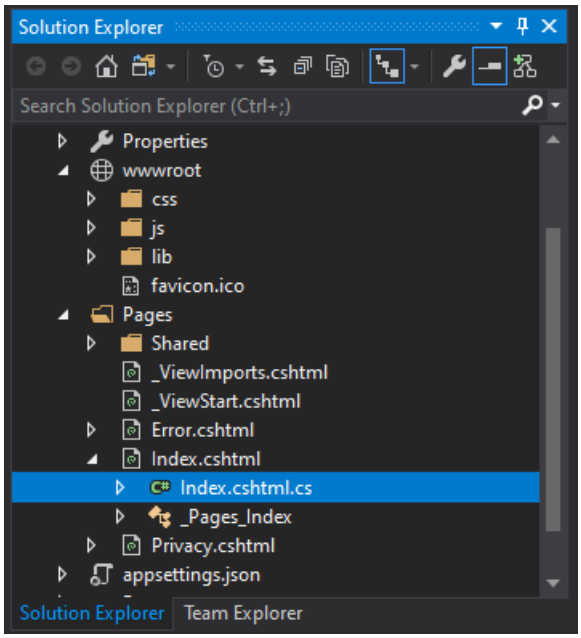
2. Expand the **Pages** folder.



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



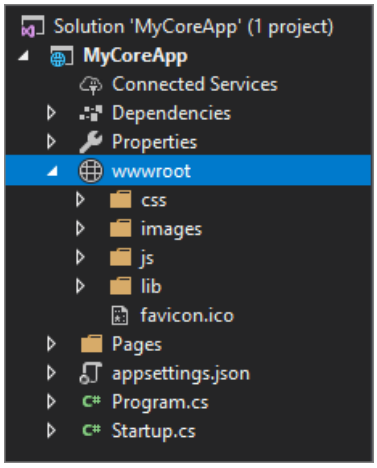
4. Each .cshtml file has an associated code file. To open the code file in the editor, expand the **Index.cshtml** node in Solution Explorer, and choose the **Index.cshtml.cs** file.



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

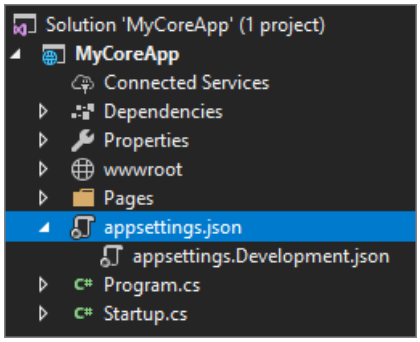
```
Index.cshtml.cs  Index.cshtml  MyCoreApp
MyCoreApp  MyCoreApp.Pages.IndexModel  _logger
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Threading.Tasks;
5  using Microsoft.AspNetCore.Mvc;
6  using Microsoft.AspNetCore.Mvc.RazorPages;
7  using Microsoft.Extensions.Logging;
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.



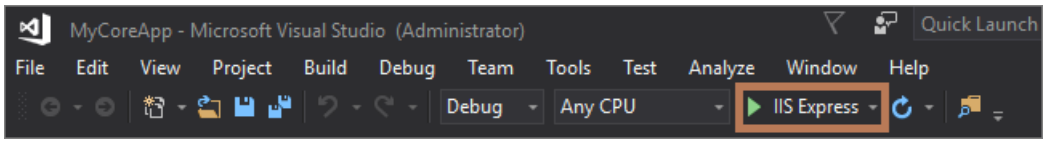
You can put static site content—such as CSS, images, and JavaScript libraries—directly in the paths where you want them.

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.)

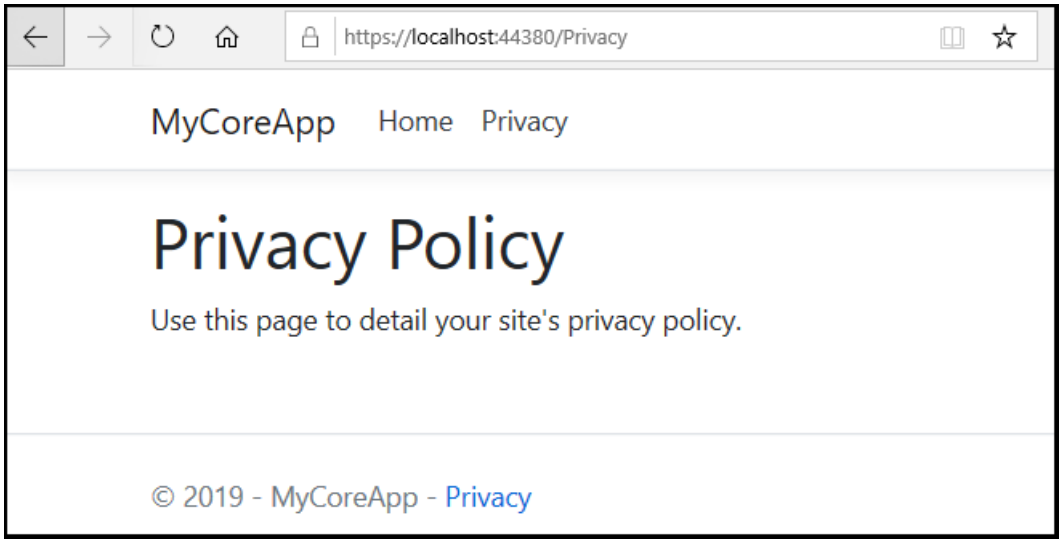


> **NOTE**
>
> If you get an error message that says **Unable to connect to web server 'IIS Express'**, close Visual Studio and then open it by using the **Run as administrator** option from the right-click or context menu. Then, run the application again.
>
> You might also get a message that asks if 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.



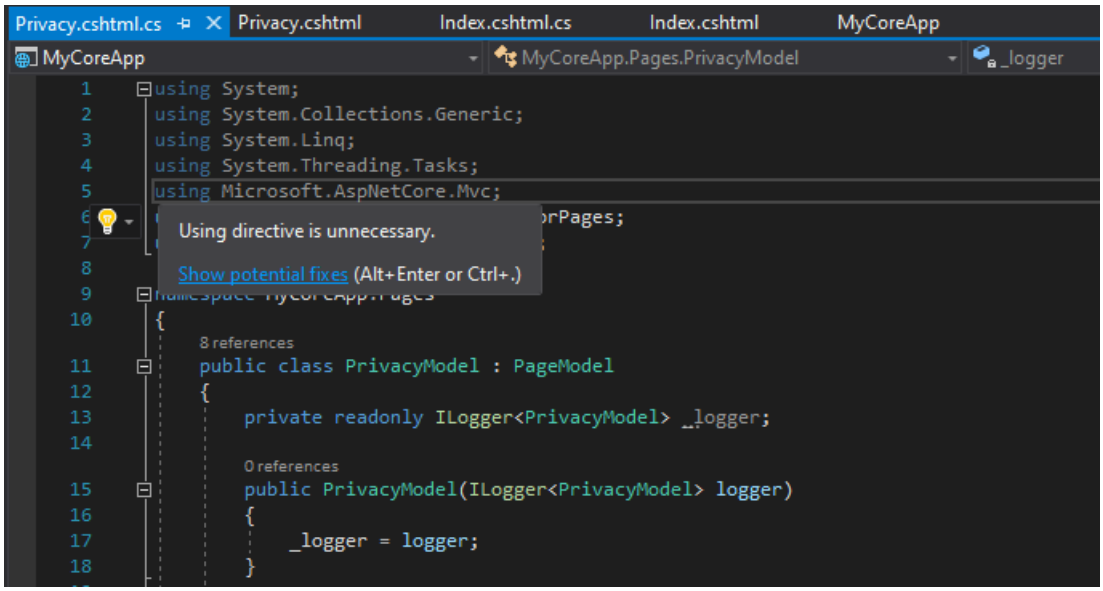
4. Return to Visual Studio, and then press **Shift+F5** to stop Debug mode. This also closes the project in the browser window.

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"]*.

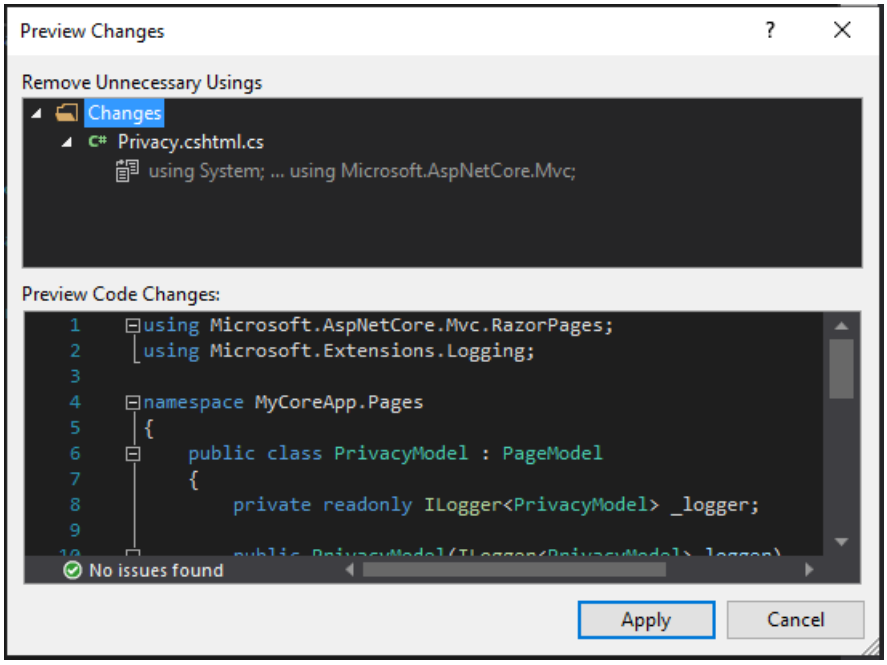
```
1   @page
2   @model PrivacyModel
3   @{
4       ViewData["Title"] = "Privacy Policy";
5   }
6   <h1>@ViewData["Title"]</h1>
7
8   <p>This page is under construction as of @ViewData["TimeStamp"].</p>
9
```

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.

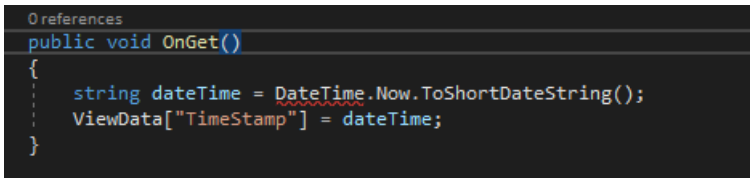


Choose **Apply**. Visual Studio deletes the unnecessary `using` directives from the file.

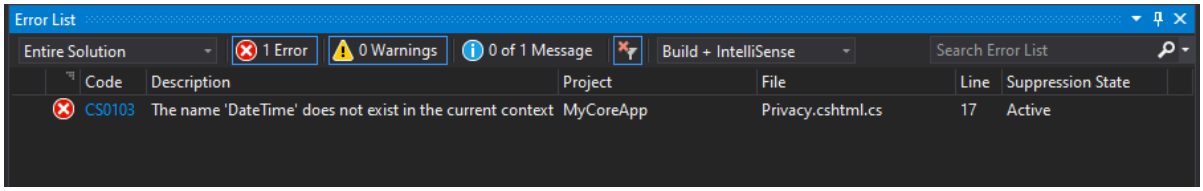
7. Next, in the `OnGet()` method, change the body to the following code:

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

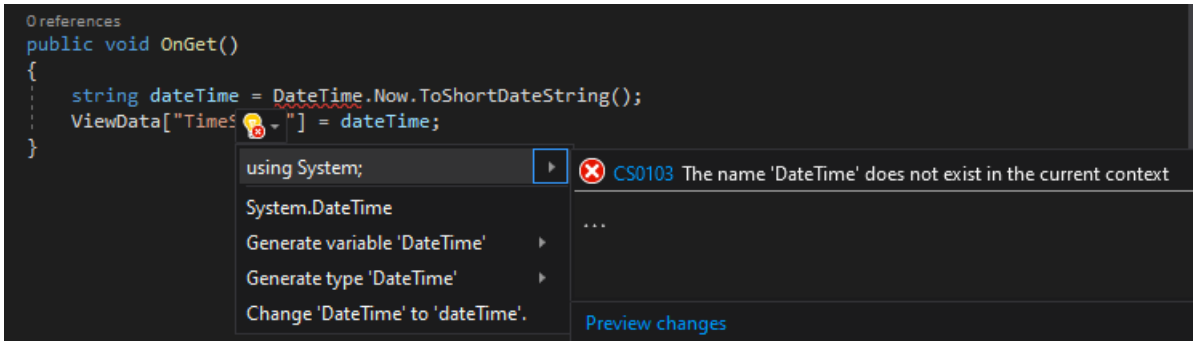
8. Notice that two wavy underlines appear under **DateTime**. The wavy underlines appear because these type isn't in scope.



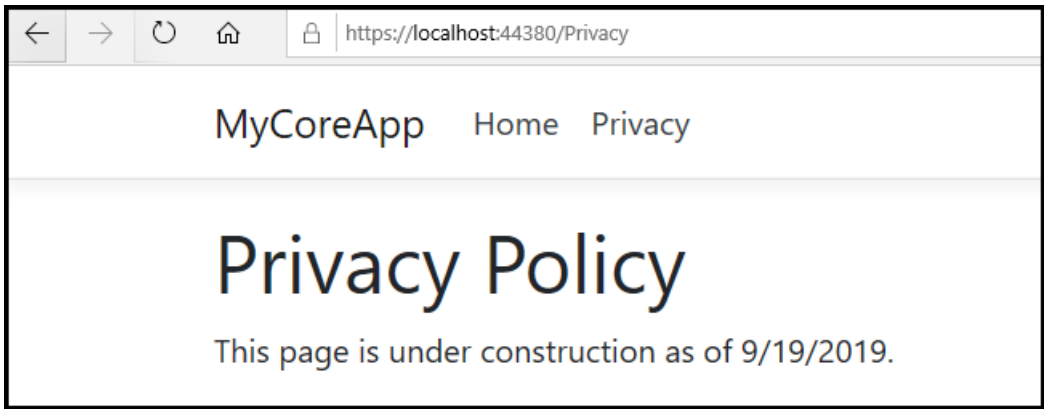
Open the **Error List** toolbar to see the same errors listed there. (If you don't see the **Error List** toolbar, choose **View** > **Error List** from the top menu bar.)



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.

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 tutorials:

Create a Razor Pages web app with ASP.NET Core

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#

4/1/2020 • 5 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 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.

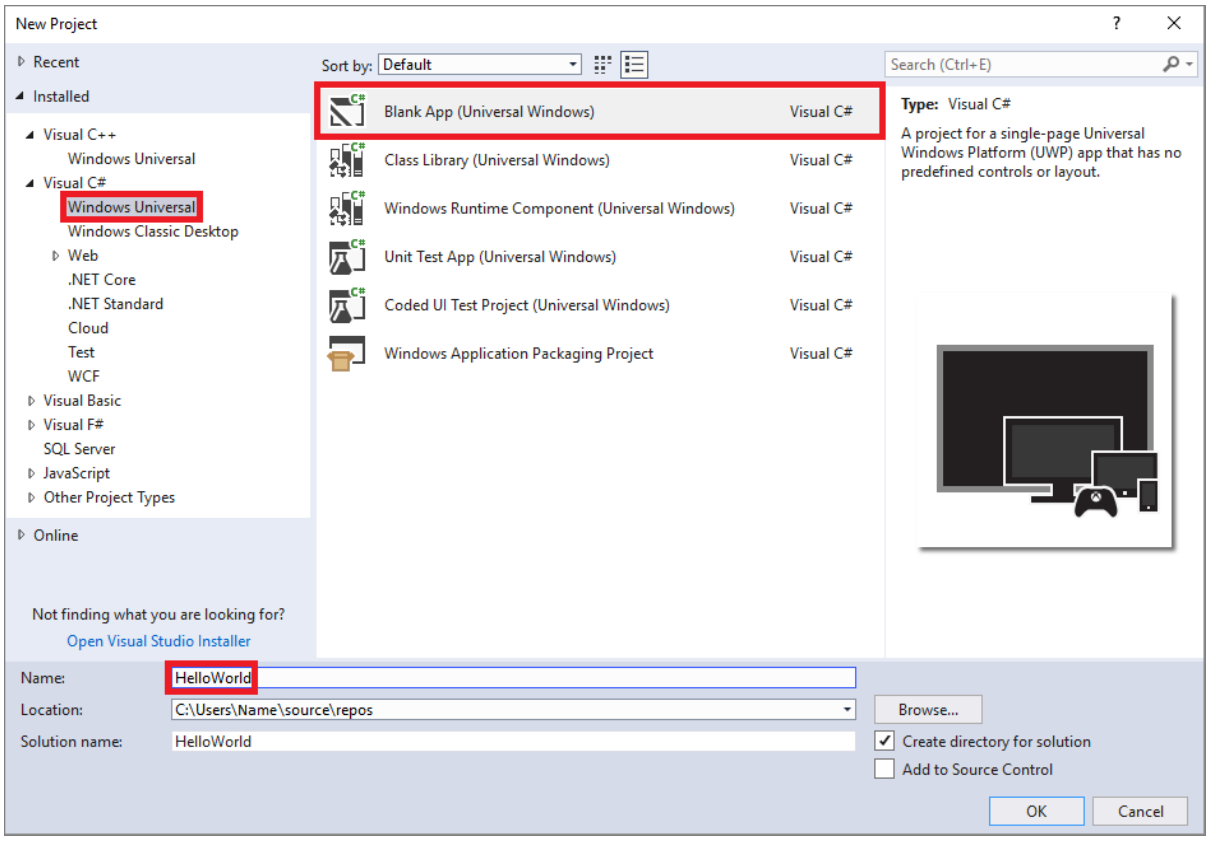
If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

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!

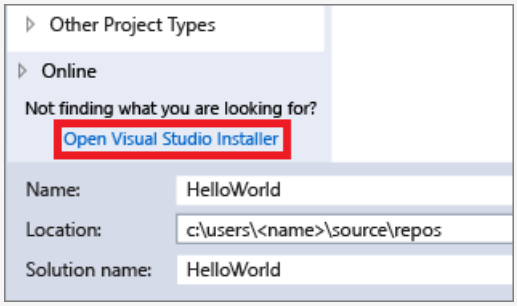
1. Open Visual Studio.
2. From the top menu bar, choose **File** > **New** > **Project**.
3. In the left pane of the **New Project** dialog box, expand **Visual C#**, and then choose **Windows Universal**. In the middle pane, choose **Blank App (Universal Windows)**. Then, name the project *HelloWorld*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

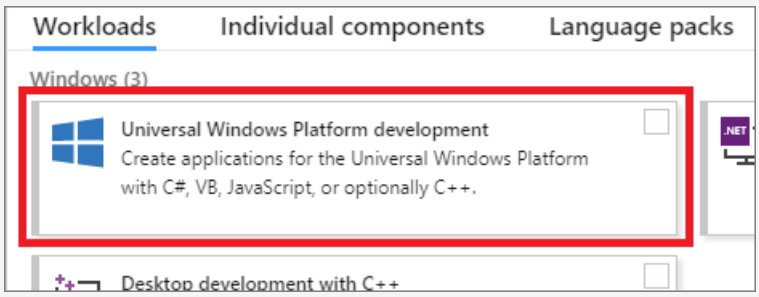


NOTE

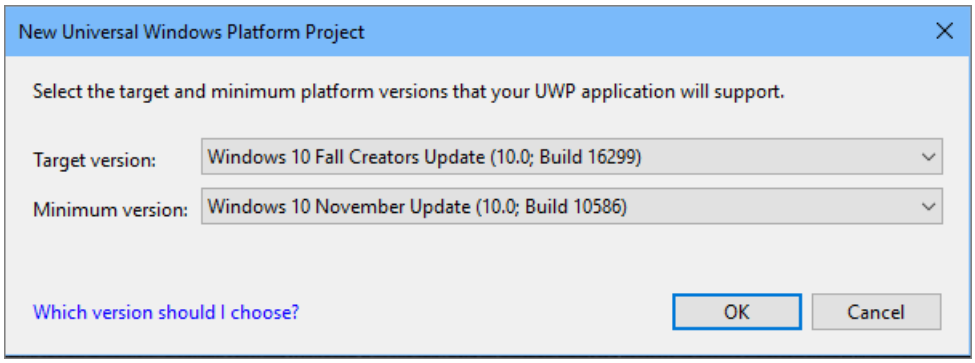
If you don't see the **Blank App (Universal Windows)** project template, click the **Open Visual Studio Installer** link in the left pane of the **New Project** dialog box.



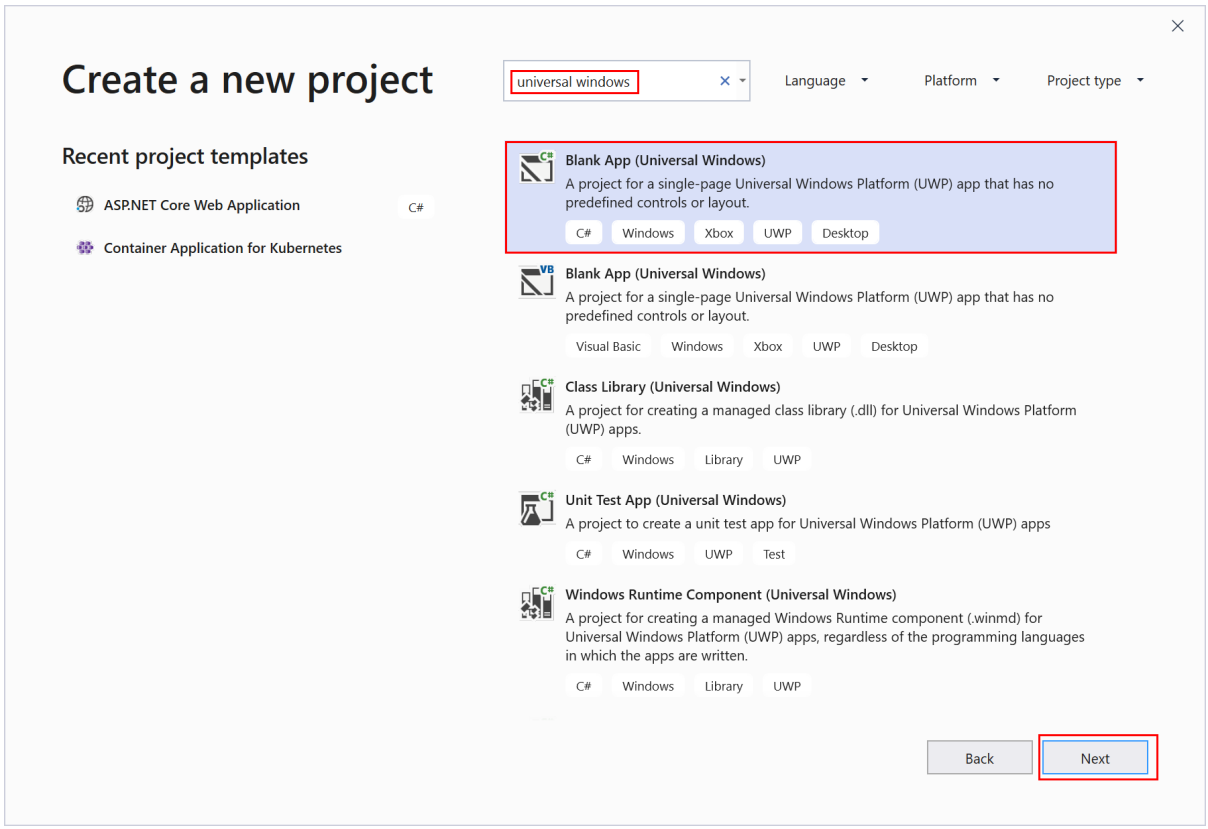
The Visual Studio Installer launches. Choose the **Universal Windows Platform development** workload, and then choose **Modify**.



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

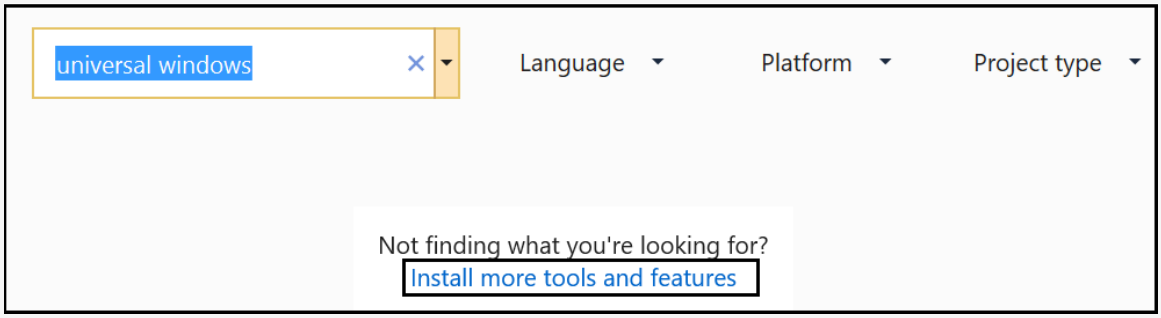


1. Open Visual Studio, and on the start window, choose **Create a new project**.
2. On the **Create a new project** screen, enter *Universal Windows* in the search box, choose the C# template for **Blank App (Universal Windows)**, and then choose **Next**.

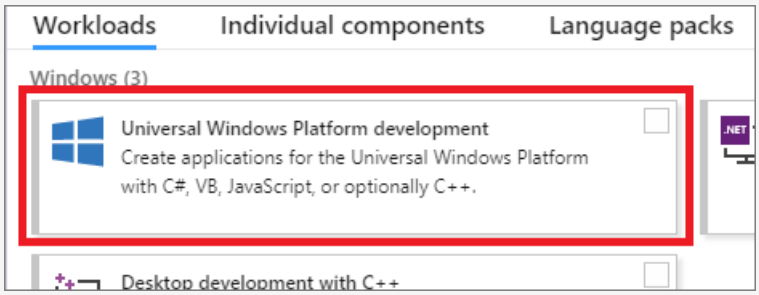


NOTE

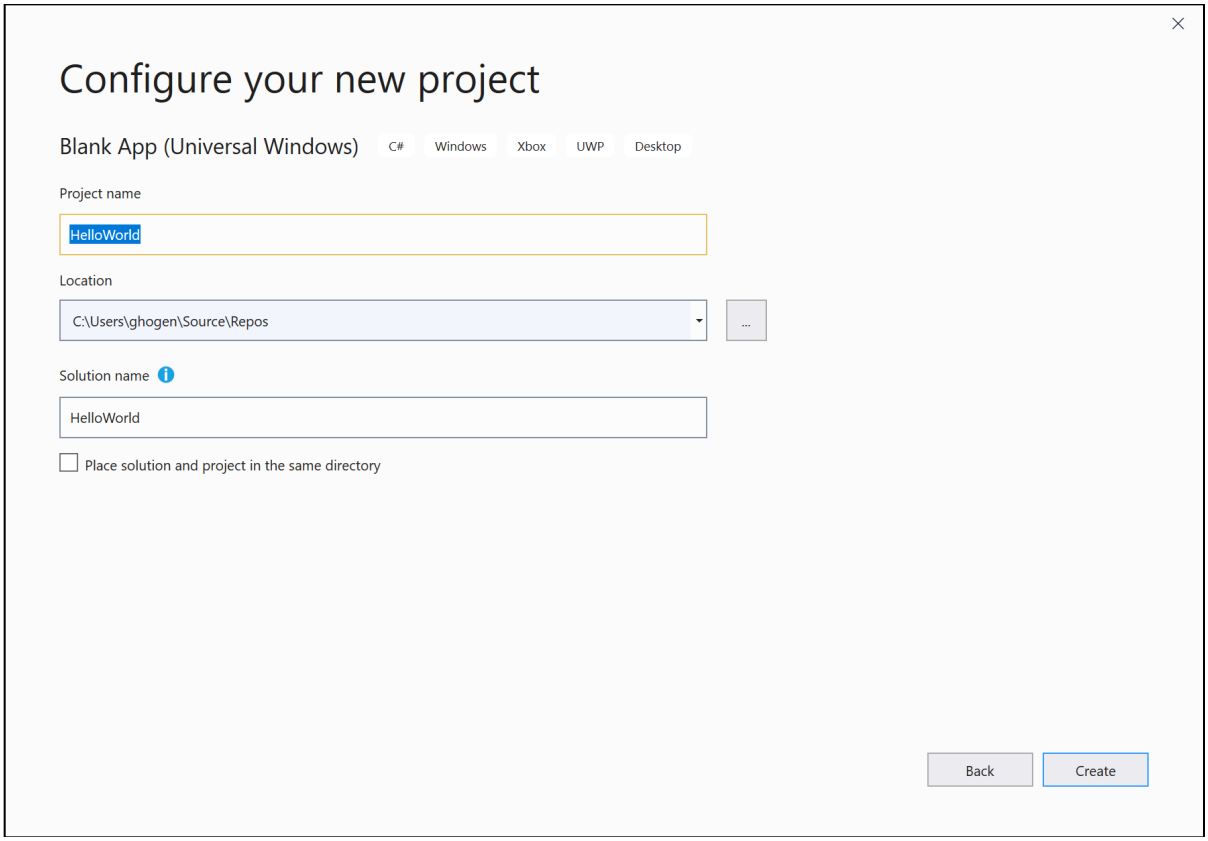
If you don't see the **Blank App (Universal Windows)** project template, click the **Install more tools and features** link.



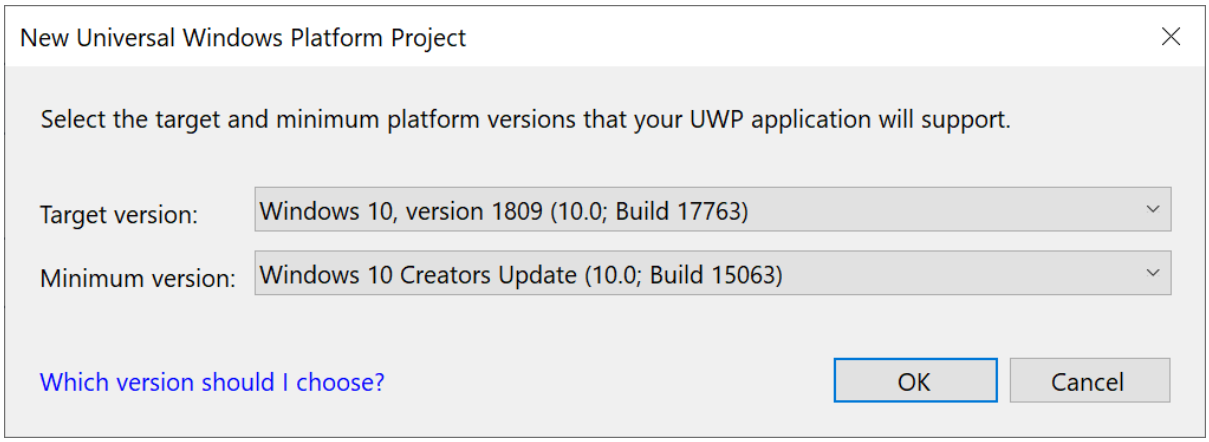
The Visual Studio Installer launches. Choose the **Universal Windows Platform development** workload, and then choose **Modify**.



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

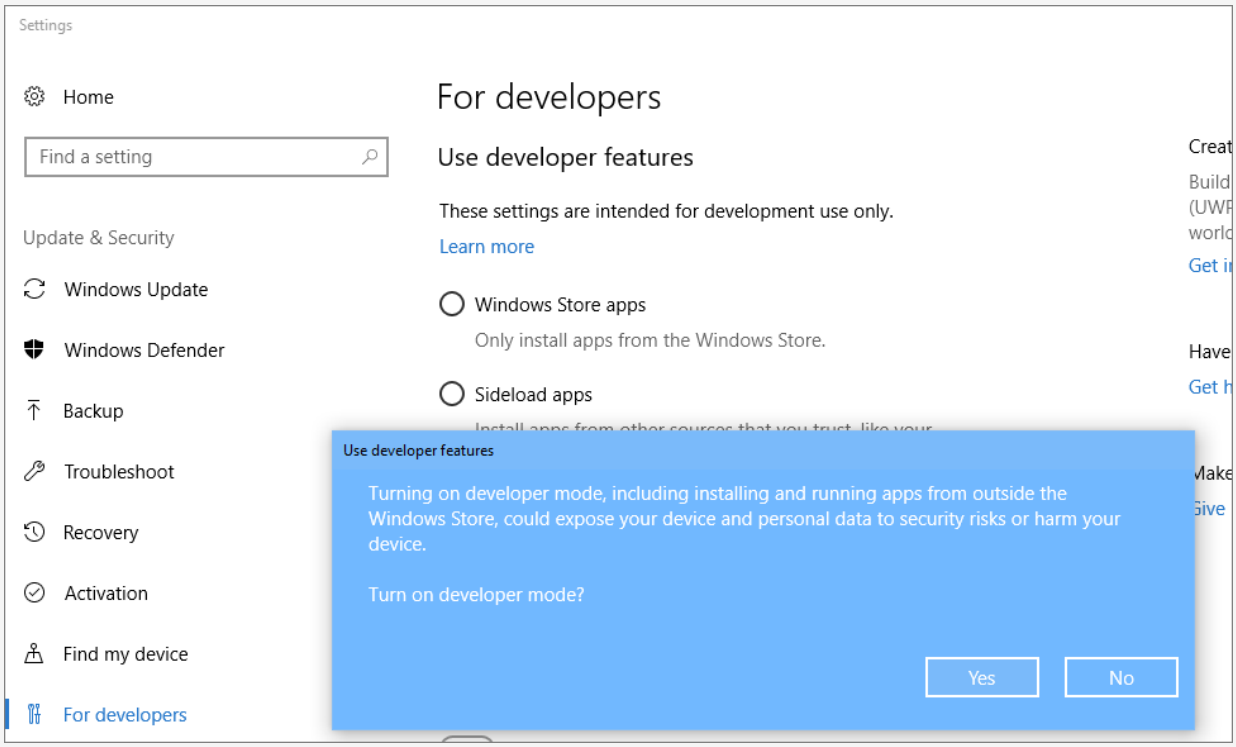


4. Accept the default **Target version** and **Minimum version** settings in the **New Universal Windows Platform Project** dialog box.



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**.



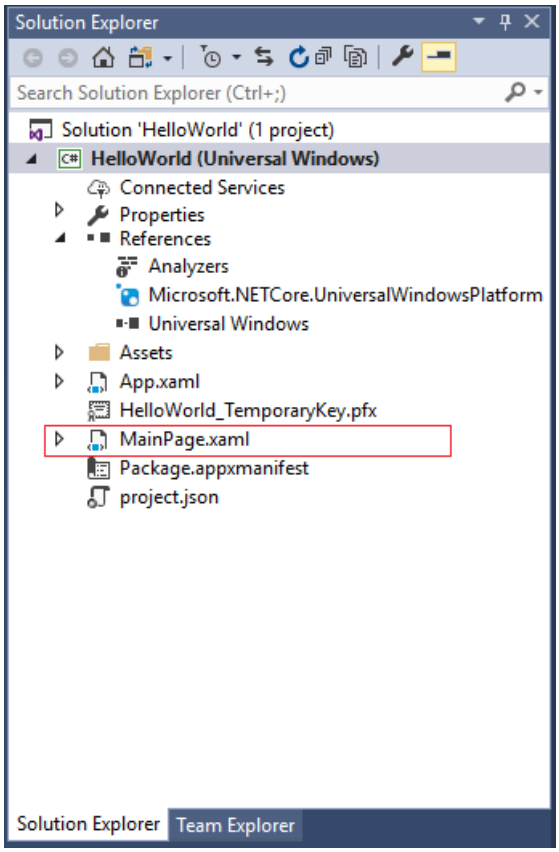
Visual Studio installs an additional Developer Mode package for you. When the package installation is complete, close the **Settings** dialog box.

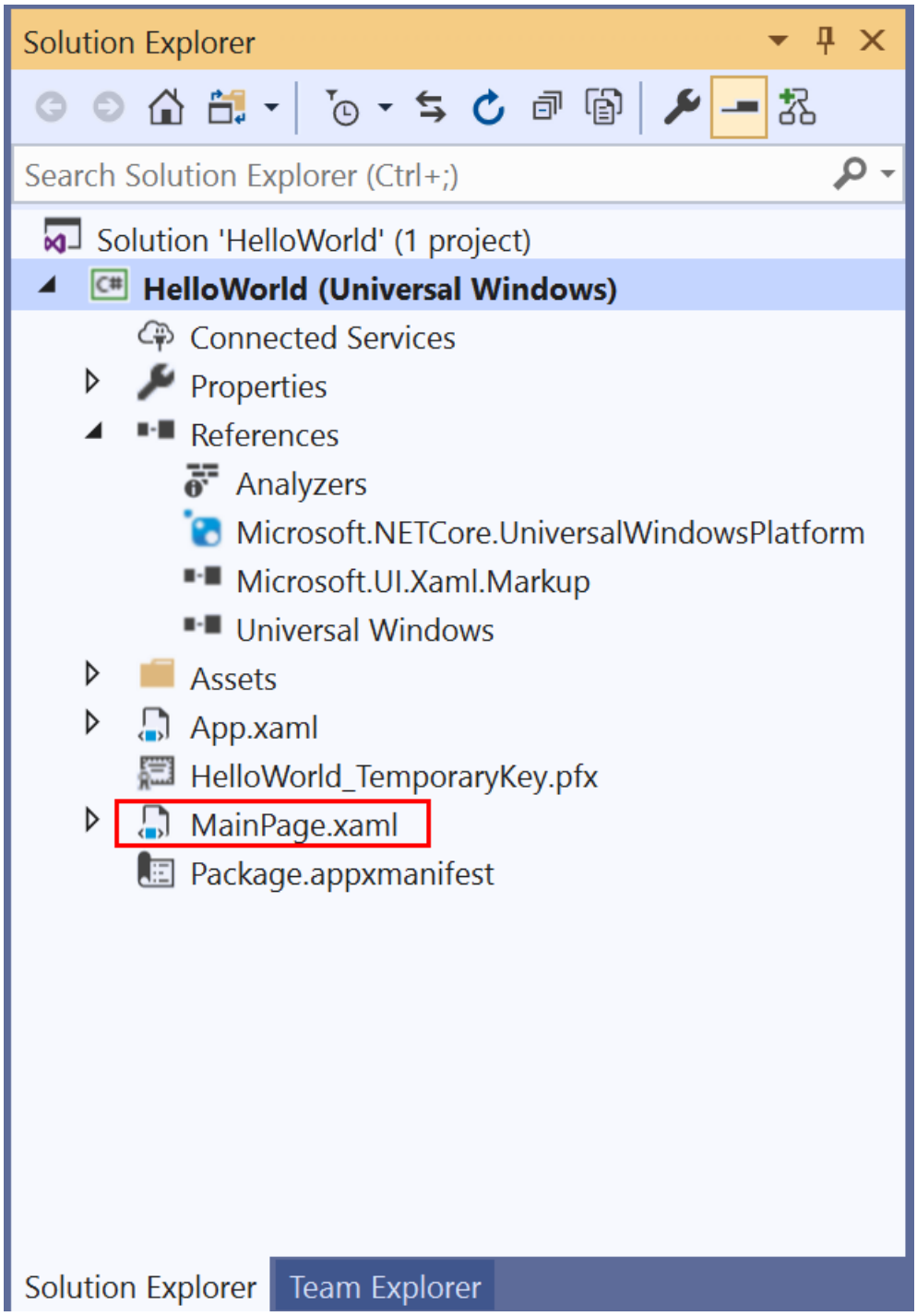
Create the application

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.

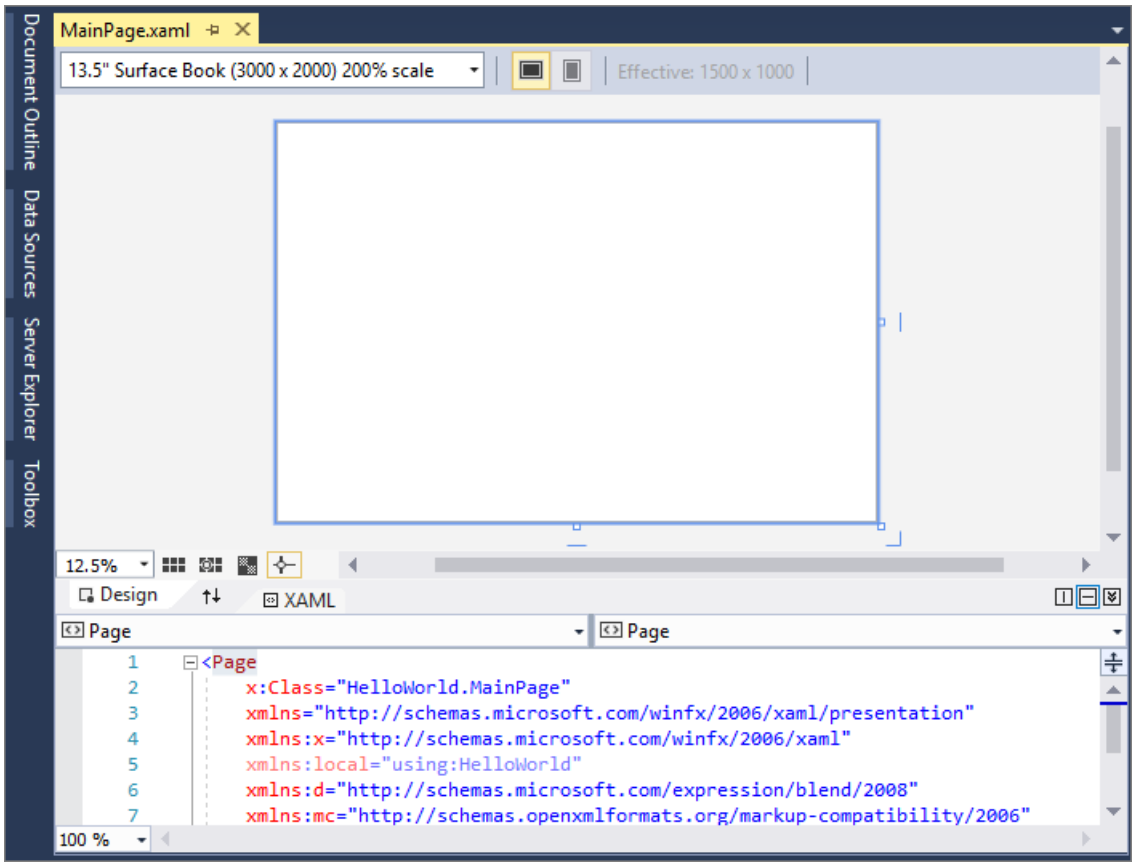
Add a button to the Design canvas

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

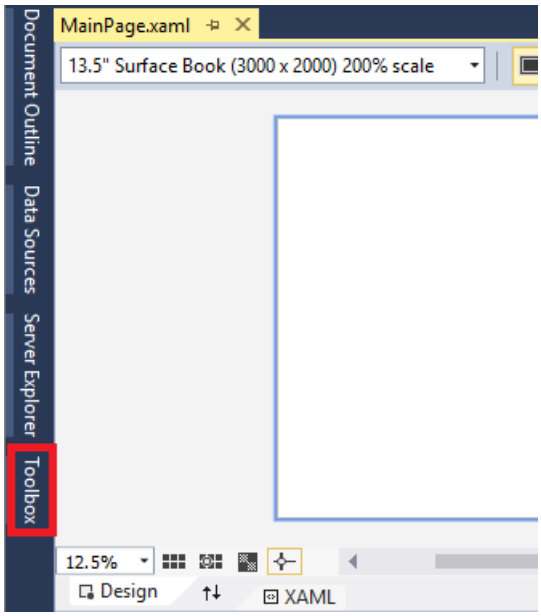

Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'HelloWorld' (1 project)
HelloWorld (Universal Windows)
Connected Services
Properties
References
Analyzers
Microsoft.NETCore.UniversalWindowsPlatform
Universal Windows
Assets
App.xaml
HelloWorld_TemporaryKey.pfx
MainPage.xaml
Package.appxmanifest
project.json
Solution Explorer
Team Explorer



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

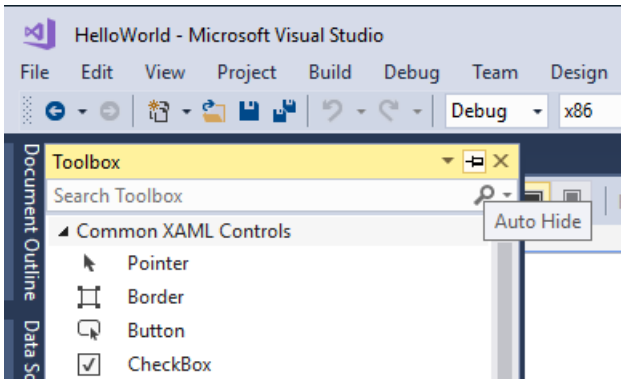


2. Choose **Toolbox** to open the Toolbox fly-out window.

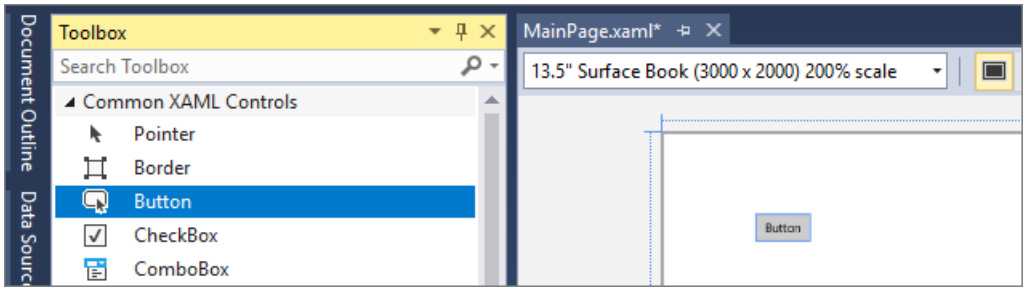


(If you don't see the **Toolbox** option, you can open it from the menu bar. To do so, choose **View** > **Toolbar**. Or, press **Ctrl**+**Alt**+**X**.)

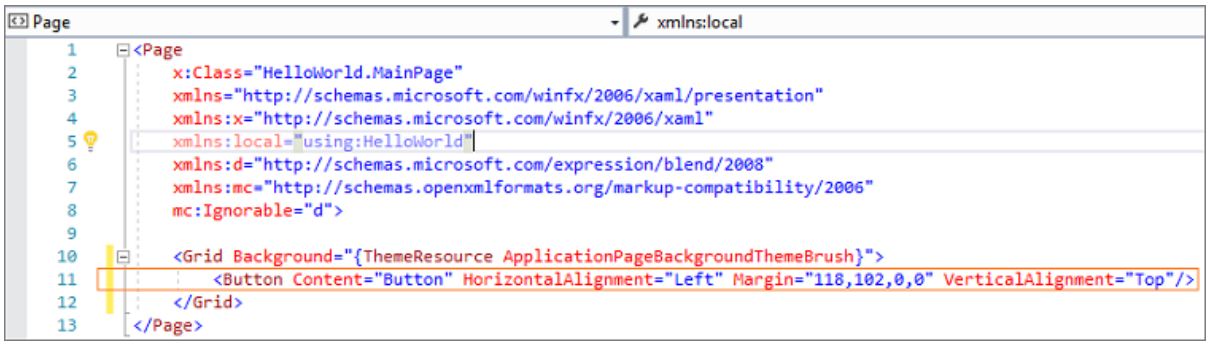
3. Click the **Pin** icon to dock the Toolbox window.



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

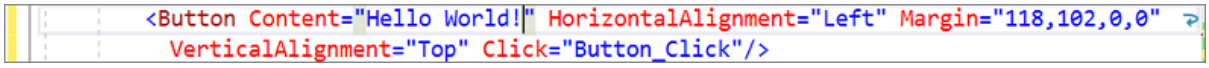


If you look at the code in the **XAML Editor**, you'll see that the Button has been added there, too:

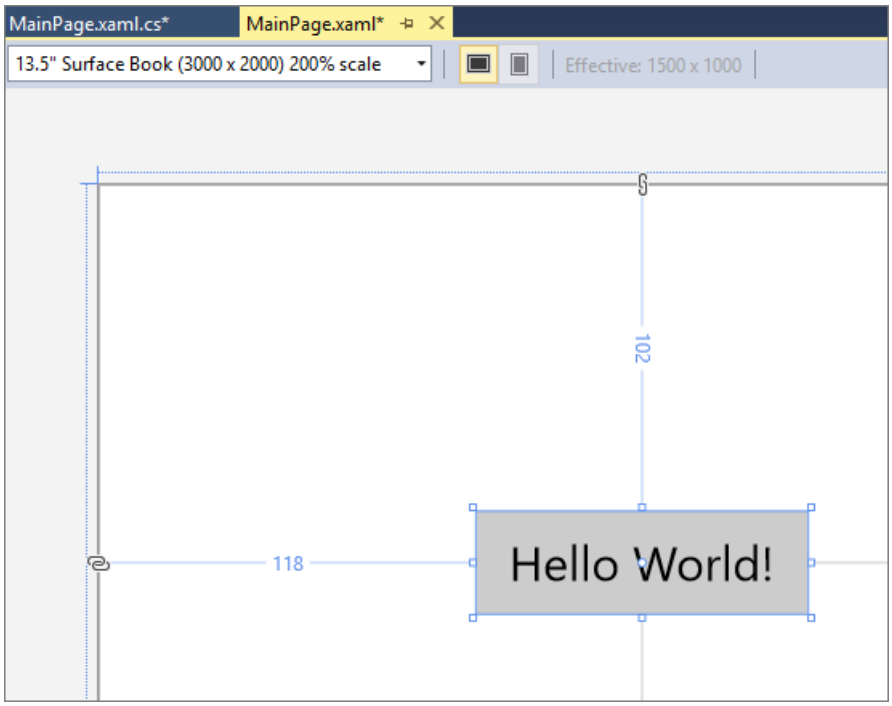


Add a label to the button

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.



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();
      }
```

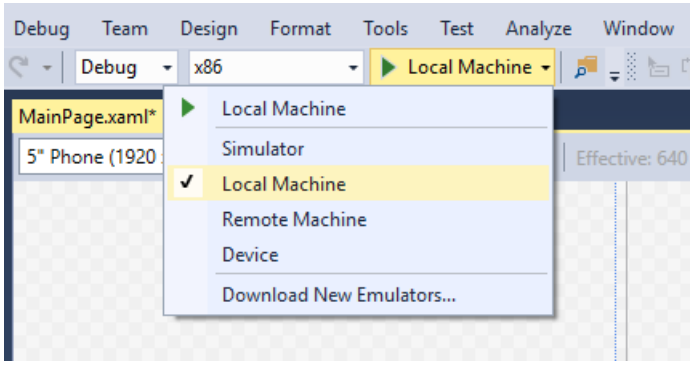
What did we just do?

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

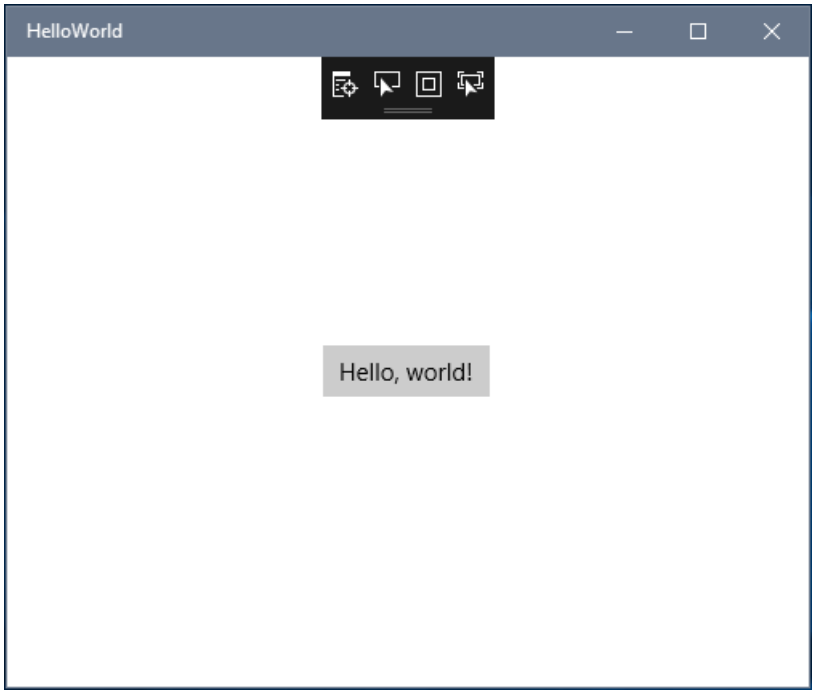
It's time to build, deploy, and launch the "Hello World" UWP app to see what it looks and sounds like. Here's how.

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:



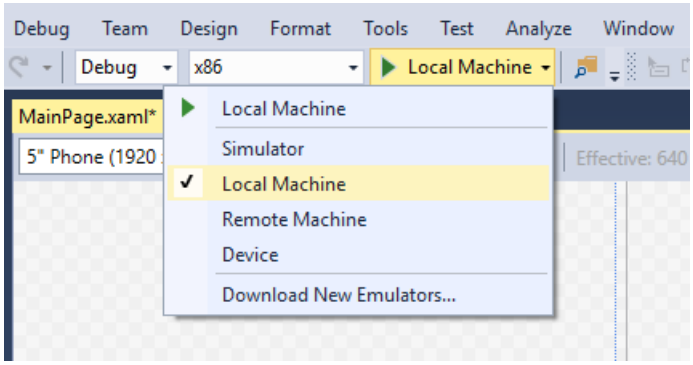
3. Click the **Hello World** button.

   Your Windows 10 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.)

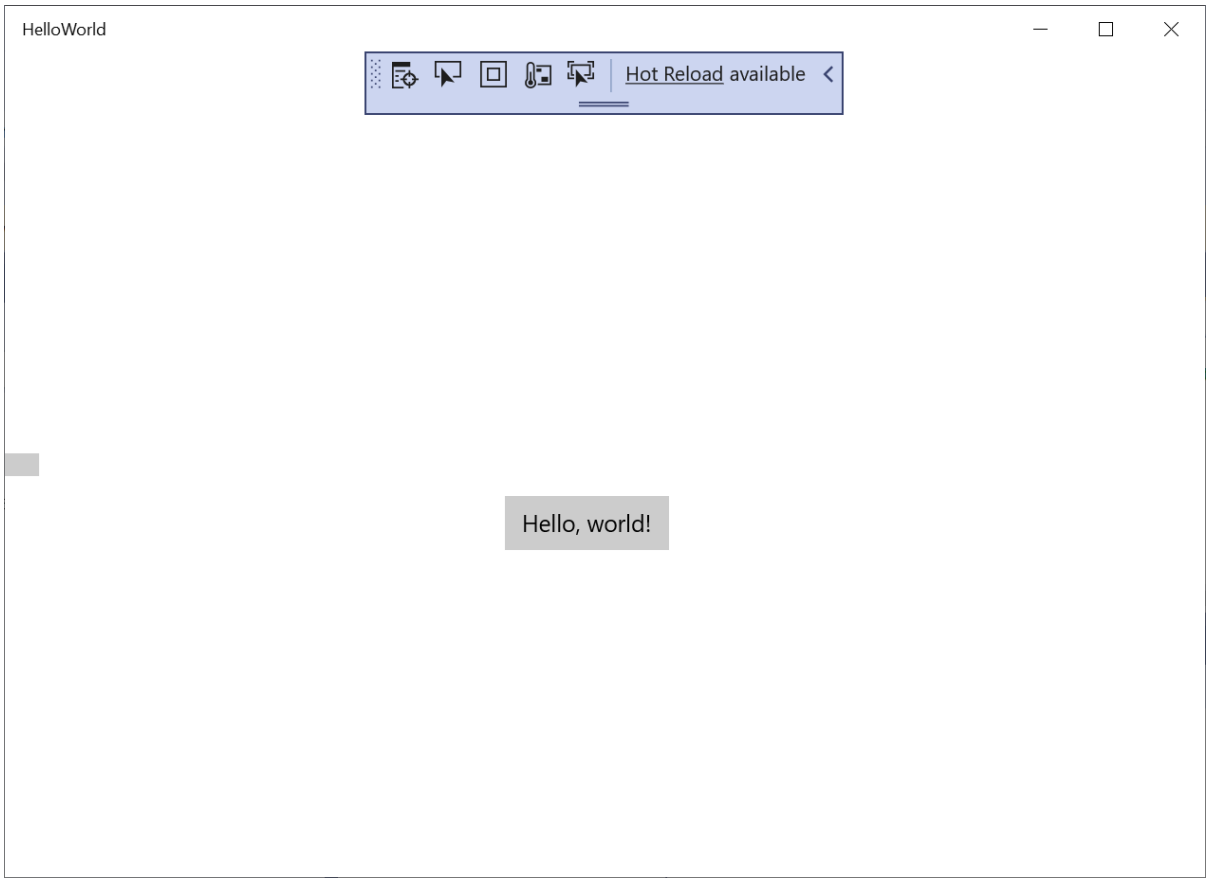
It's time to build, deploy, and launch the "Hello World" UWP app to see what it looks and sounds like. Here's how.

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 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.)

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#

2/25/2020 • 10 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).

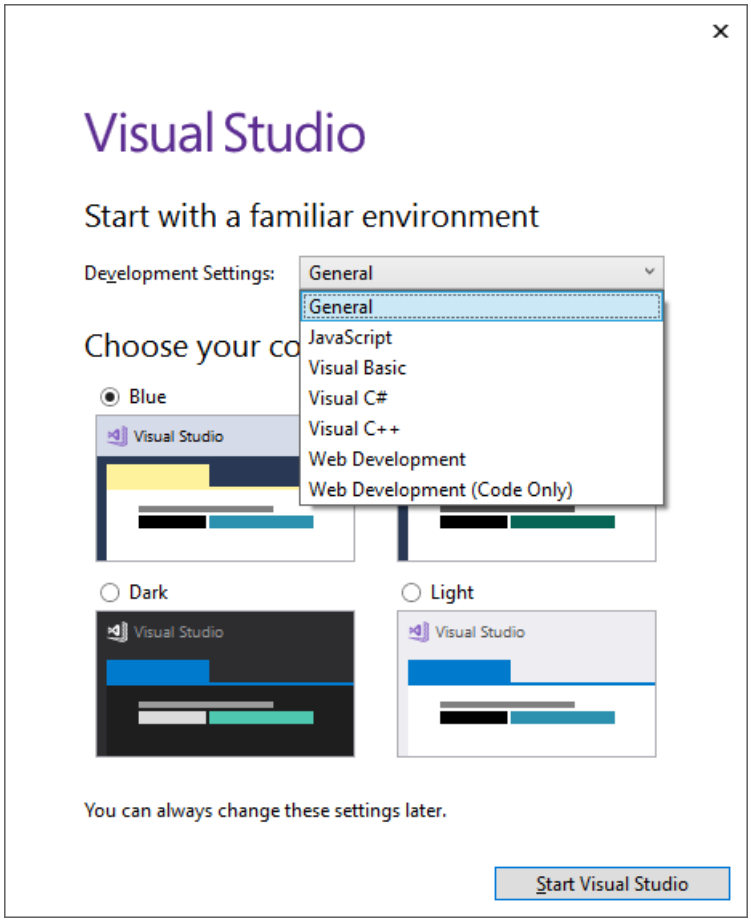
Prerequisites

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

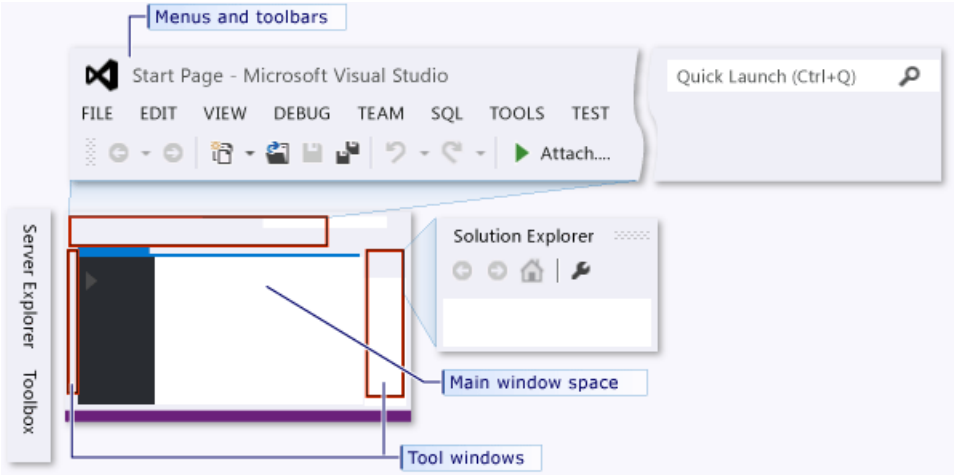
- If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.
- You can use either .NET Framework or .NET Core for this tutorial. .NET Core is the newer, more modern framework. .NET Core requires Visual Studio 2019 version 16.3 or later.

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.

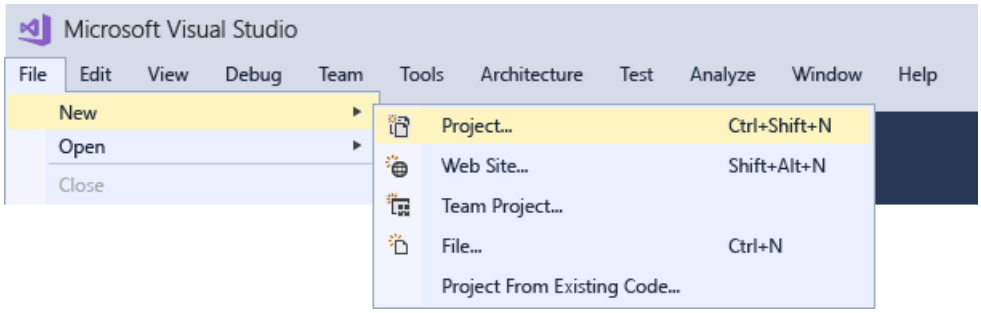


When you launch Visual Studio, the start window opens first. Select **Continue without code** to open the development environment. You'll see tool windows, the menus and toolbars, and the main window space. Tool windows are docked on the left and right sides of the application window, with a search box, the menu bar, and the standard toolbar 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.

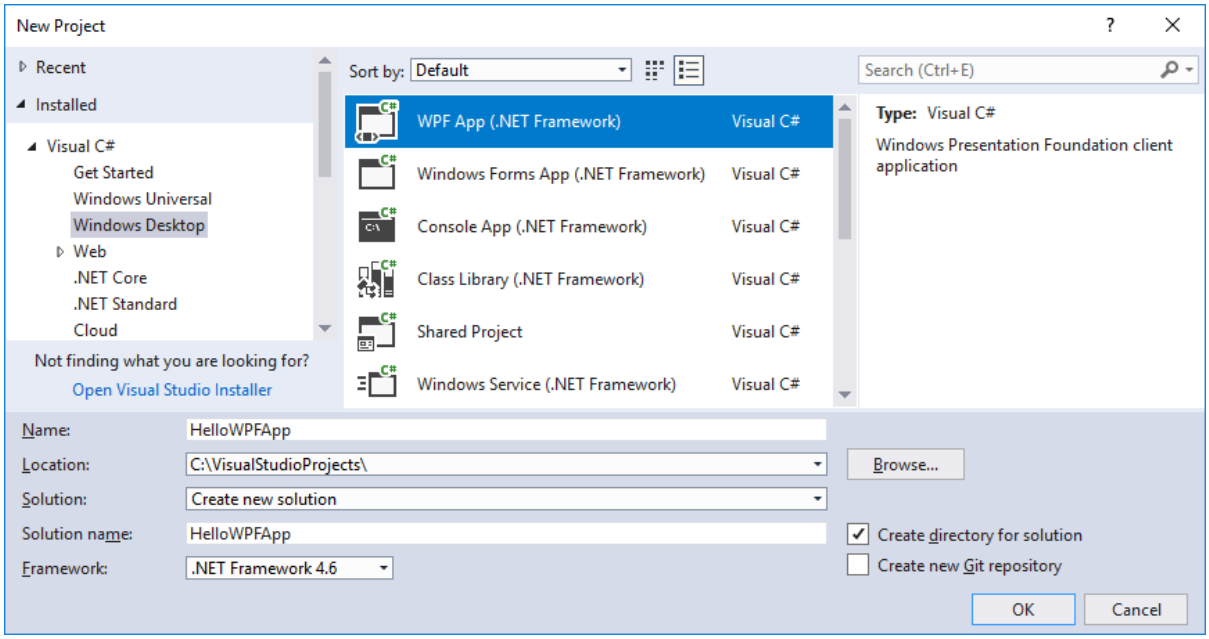
Create the project

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.

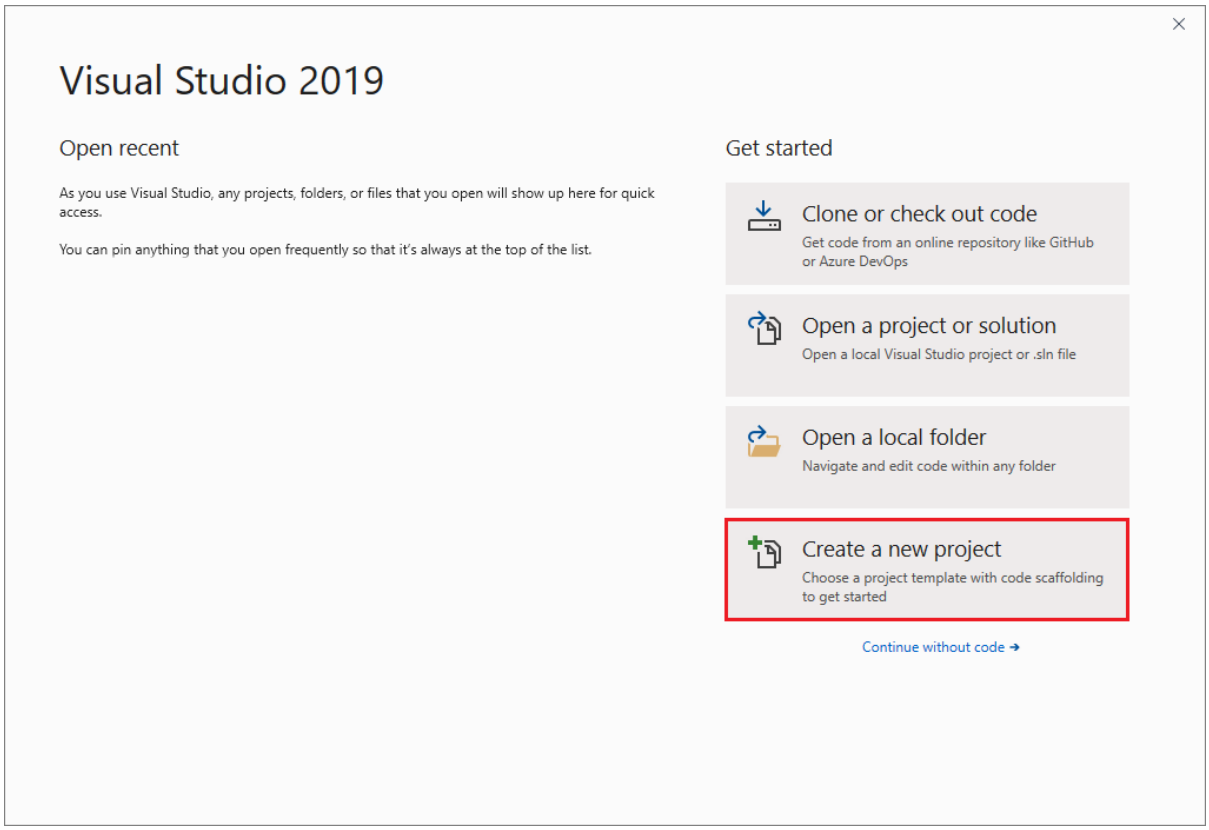
1. Create a new project. On the menu bar, select **File** > **New** > **Project**.



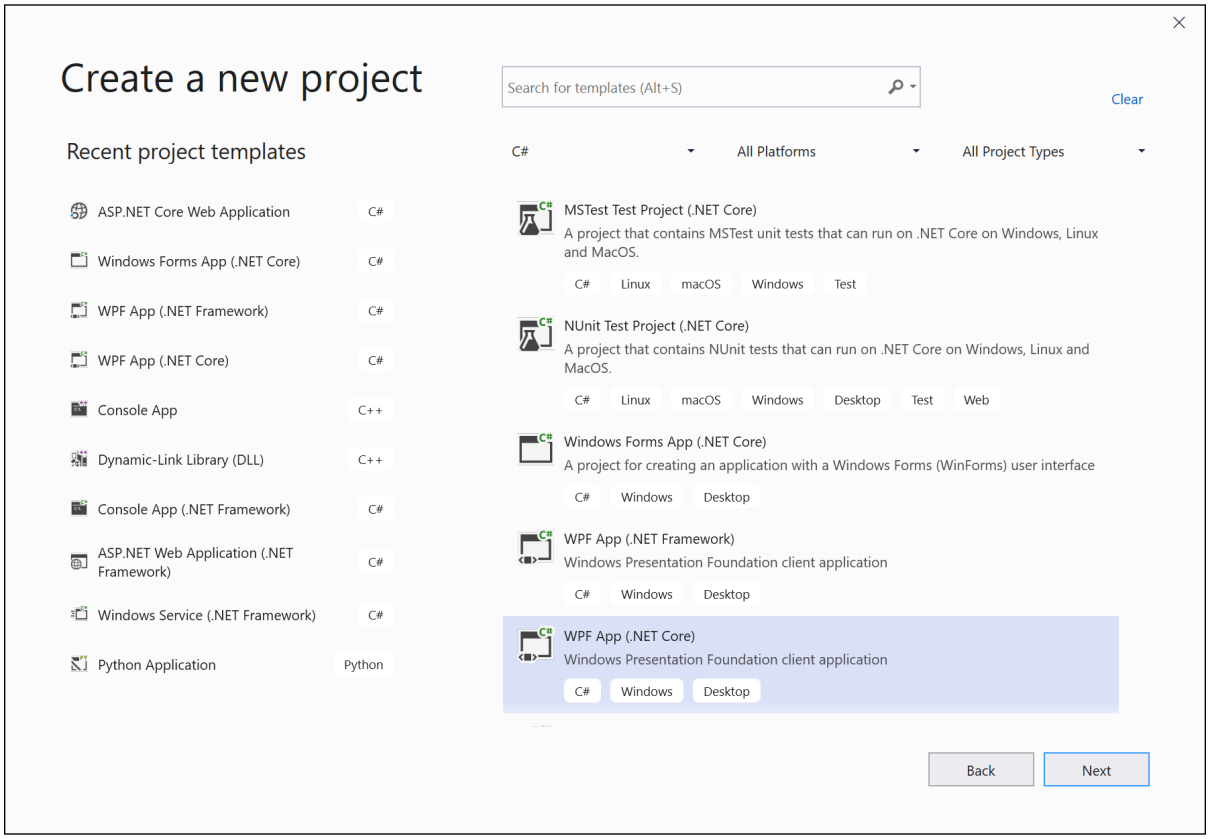
2. In the **New Project** dialog, select the **Installed** > **Visual C#** > **Windows Desktop** category, and then select the **WPF App (.NET Framework)** template. Name the project **HelloWPFApp**, and select **OK**.



1. Open Visual Studio 2019.
2. On the start window, choose **Create new project**.

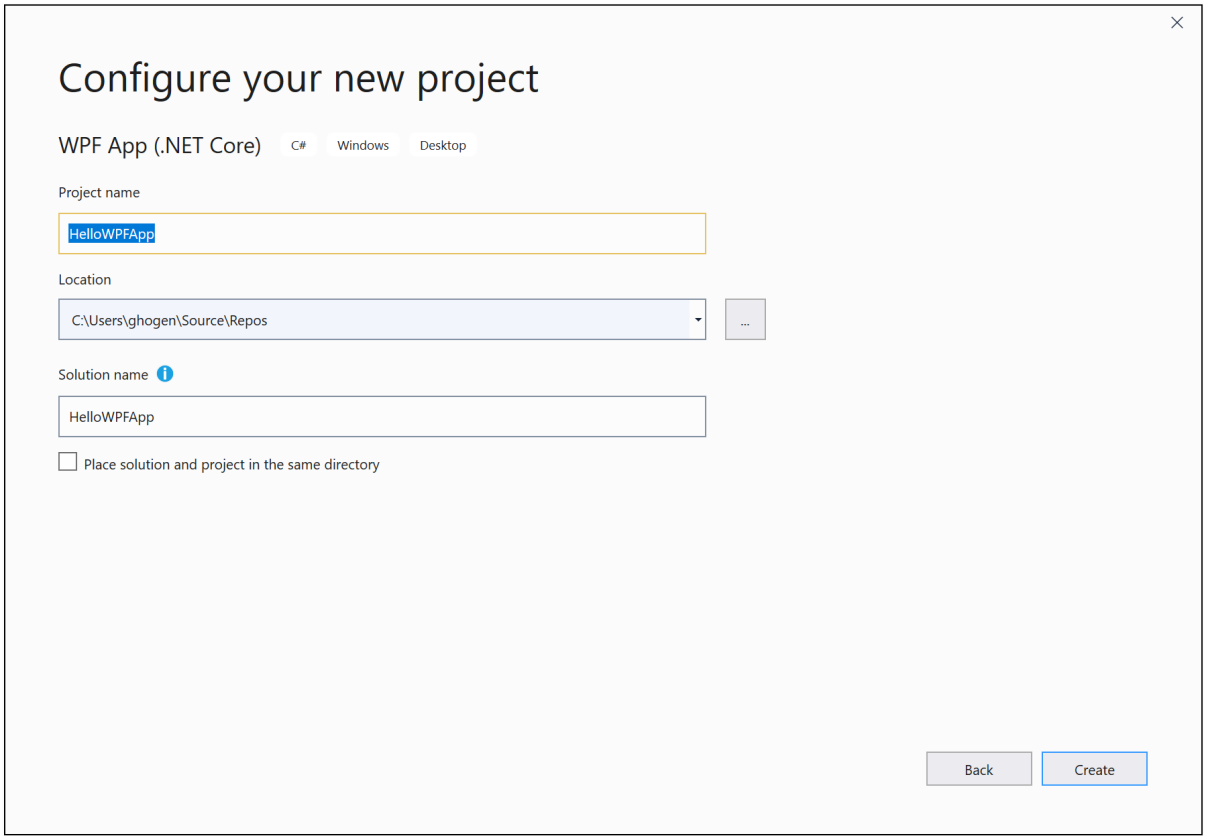


3. On the **Create a new project** screen, search for "WPF," choose **WPF App (.NET Core)**, and then choose **Next**.

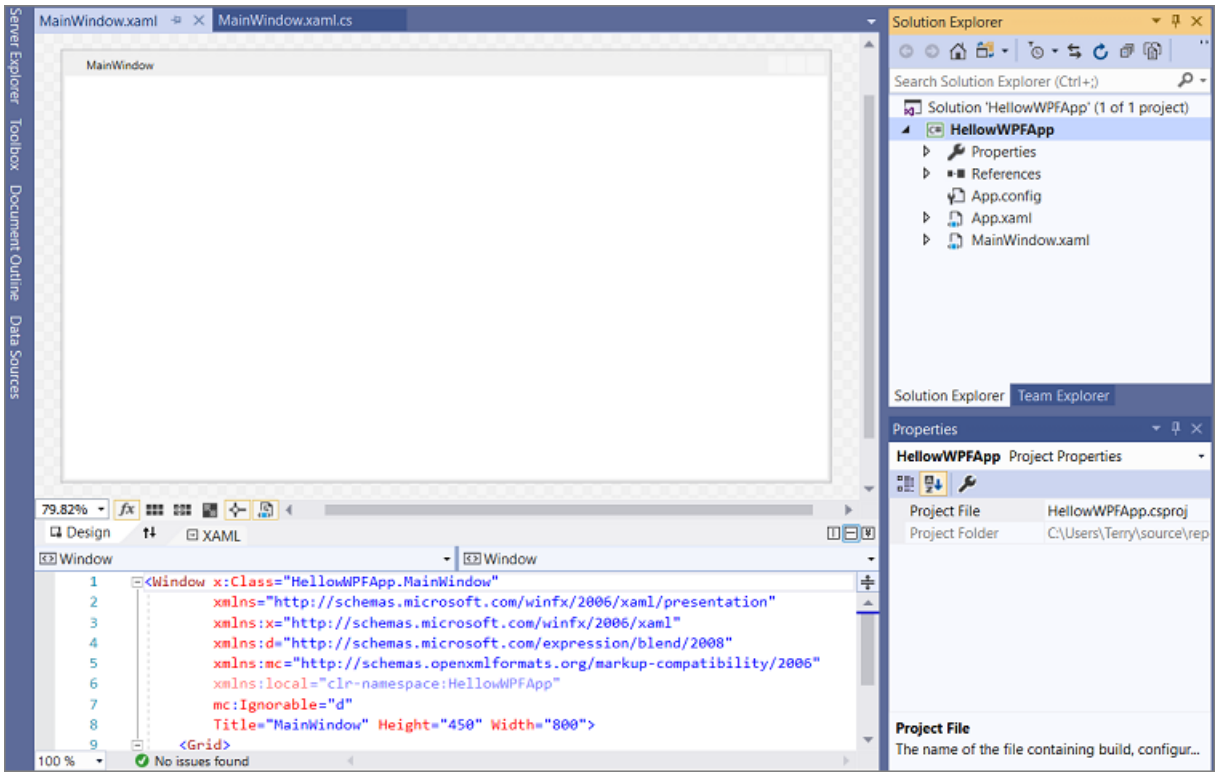


> **NOTE**
>
> You might find two WPF desktop templates, one for .NET Framework and another for .NET Core. The .NET Core template is available in Visual Studio 2019 version 16.3 and later. You can use either one for this tutorial, but we recommend .NET Core for new development.

4. At the next screen, give the project a name, **HelloWPFApp**, and choose **Create**.

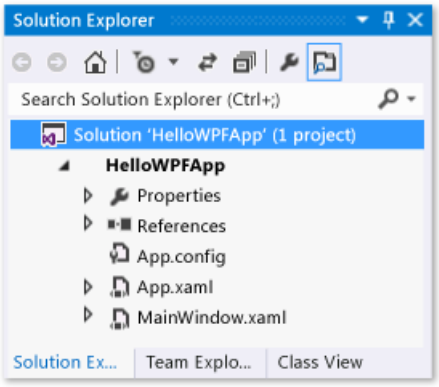


Visual Studio creates the HelloWPFApp project and solution, and **Solution Explorer** shows the various files. The **WPF Designer** shows a design view and a XAML view of *MainWindow.xaml* in a split view. You can slide the splitter to show more or less of either view. You can choose to see only the 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)

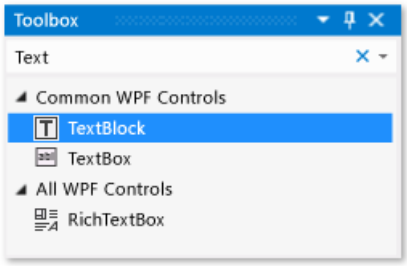
If the designer is not open, select *Greetings.xaml* and press **Shift+F7** to open the designer.

We'll add three types of controls to this application: a TextBlock control, two RadioButton controls, and a Button control.

Add a TextBlock control

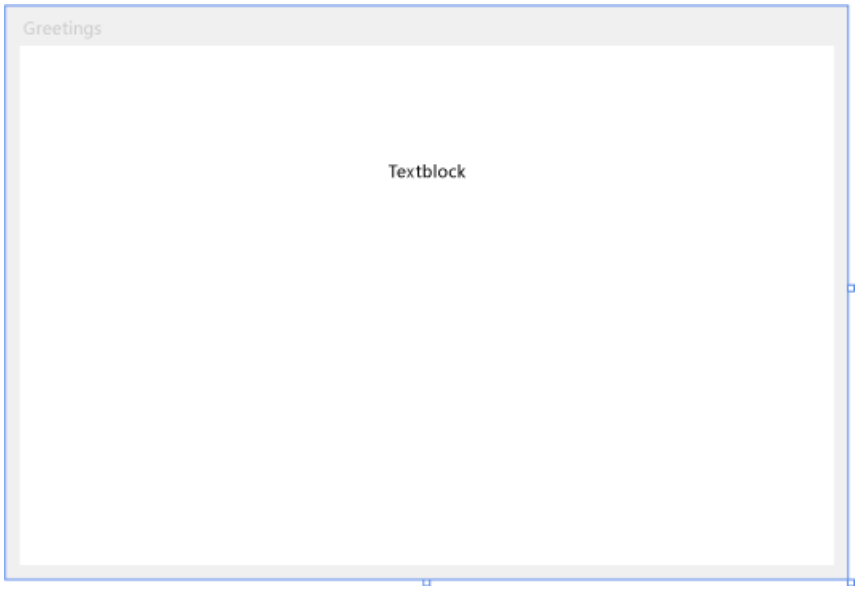
1. Press **Ctrl+Q** to activate the search box and type **Toolbox**. Choose **View > Toolbox** from the results list.

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>
```

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:

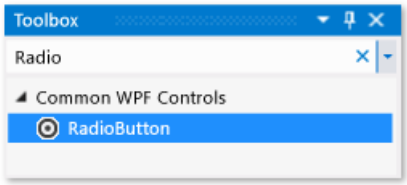
```
<Grid>
    <TextBlock HorizontalAlignment="Left" Margin="387,60,0,0" TextWrapping="Wrap" Text="Select a message
option and then choose the Display button." VerticalAlignment="Top"/>
</Grid>
```

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.

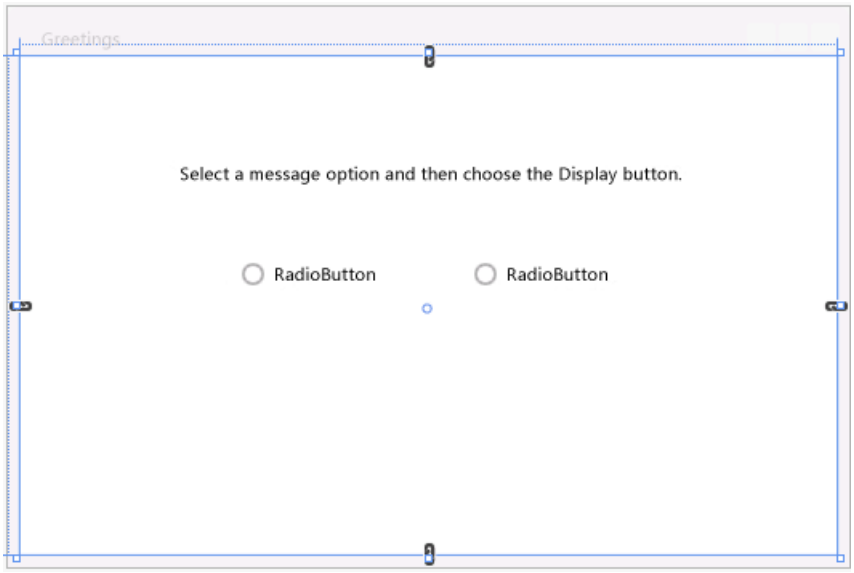
Add radio buttons

1. In the **Toolbox**, find the **RadioButton** control.

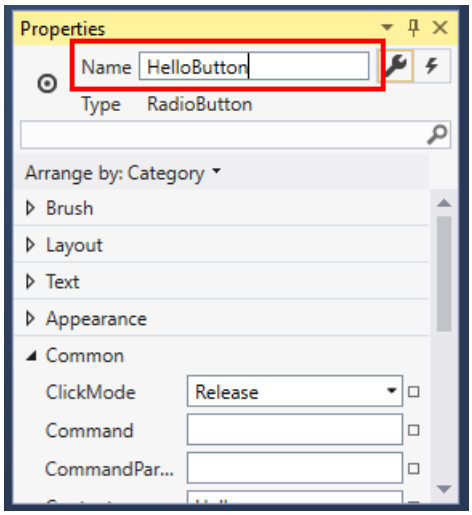


2. Add two RadioButton controls to the design surface by choosing the **RadioButton** item and dragging it to the window on the design surface. Move the buttons (by selecting them and using the arrow keys) so that the buttons appear side by side under the TextBlock control. Use the red guidelines to 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.

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:

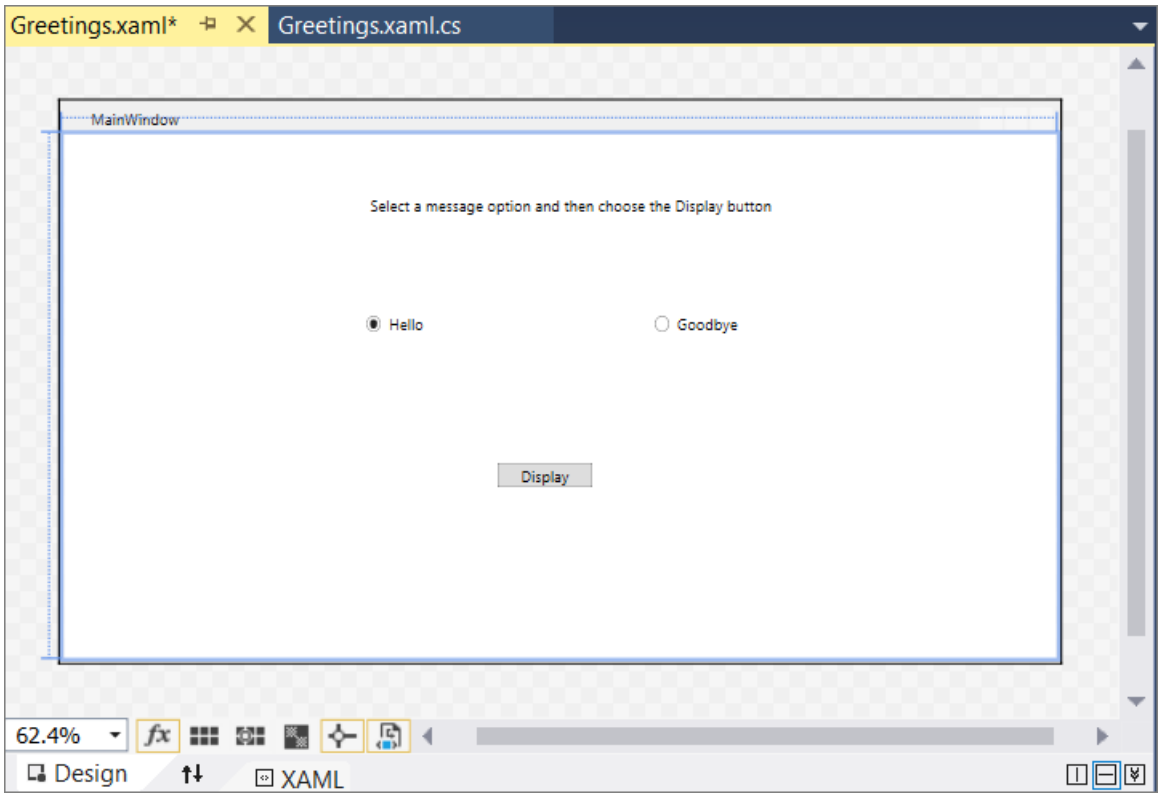
```
<Grid>
     <TextBlock HorizontalAlignment="Left" Margin="252,47,0,0" TextWrapping="Wrap" Text="Select a
message option and then choose the Display button." VerticalAlignment="Top"/>
     <RadioButton x:Name="HelloButton" Content="Hello" IsChecked="True" HorizontalAlignment="Left"
Margin="297,161,0,0" 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>
```

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.

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.

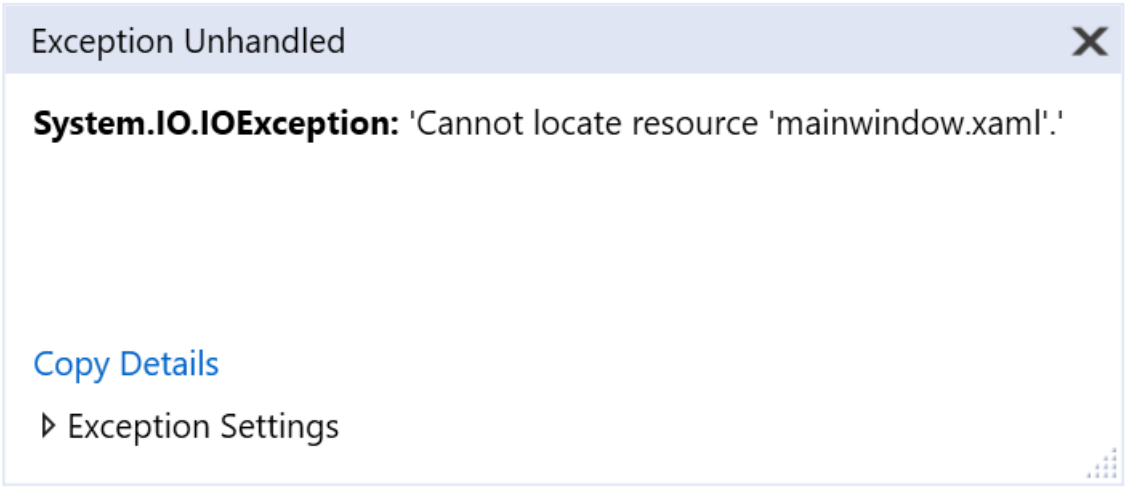
Find and fix errors

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'.



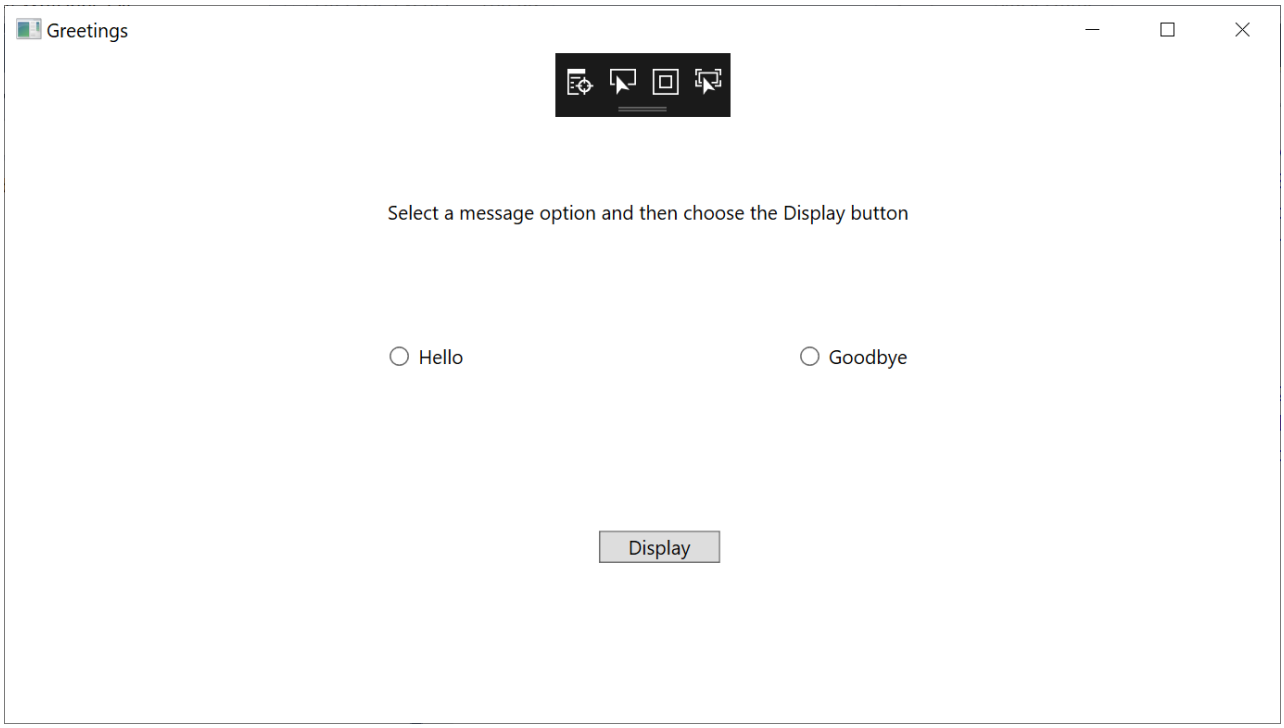
2. Stop the debugger by choosing **Debug** > **Stop Debugging**.

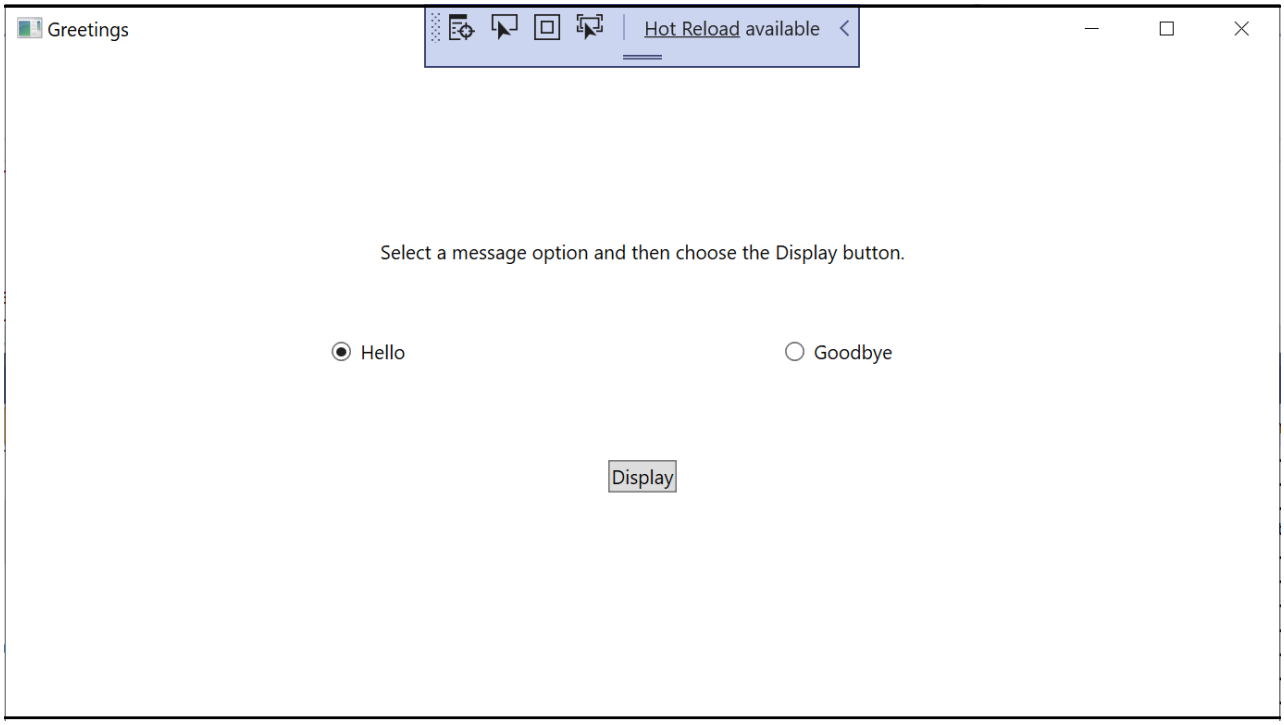
We renamed *MainWindow.xaml* to *Greetings.xaml* at the start of this tutorial, but the code still refers to *MainWindow.xaml* as the startup URI for the application, so the project can't start.

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 then save the changes.

Start the debugger again (press **F5**). You should see the **Greetings** window of the 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**.

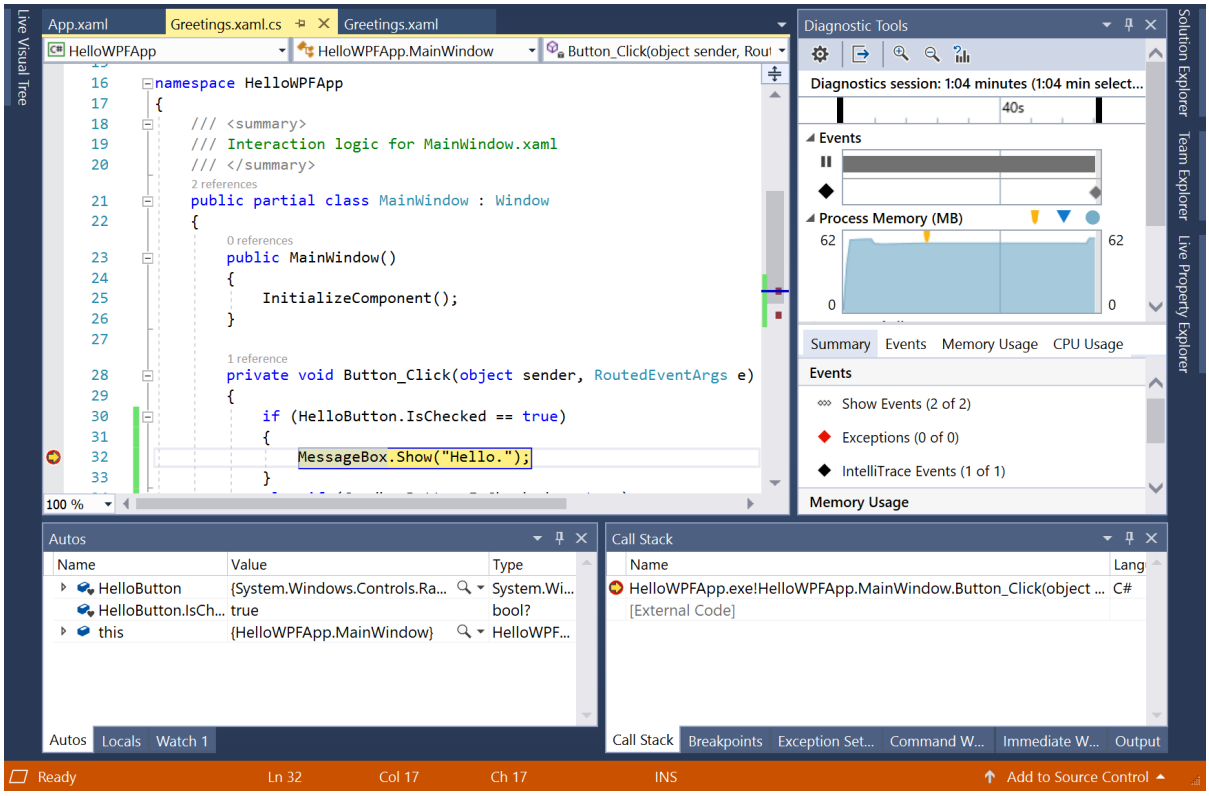
Add breakpoints

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.

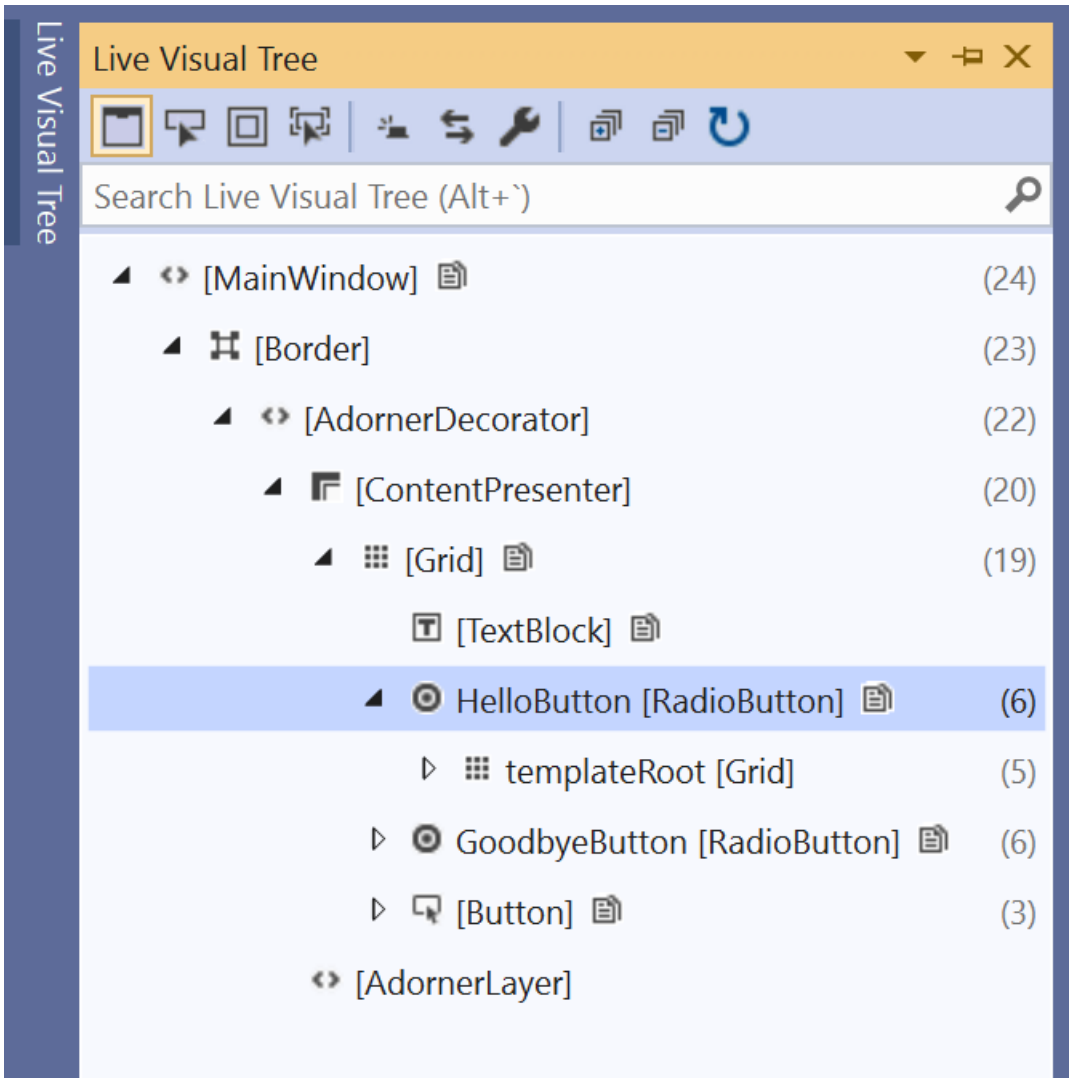
9. Choose the **F5** key to continue debugging. When the message box appears, choose the **OK** button on the message box to close it.

10. Close the application window to stop debugging.

11. On the menu bar, choose **Debug** > **Disable All Breakpoints**.

View a representation of the UI elements

In the running app, you should see a widget that appears at the top of your window. This is a runtime helper that provides quick access to some helpful debugging features. Click on 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 isn't necessary, 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#

4/13/2020 • 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.

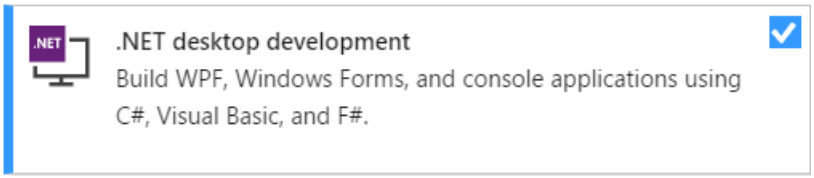
> **NOTE**
> Some of the screenshots in this tutorial use the dark theme. If you aren't using the dark theme but would like to, see the Personalize the Visual Studio IDE and Editor page to learn how.

Create a project

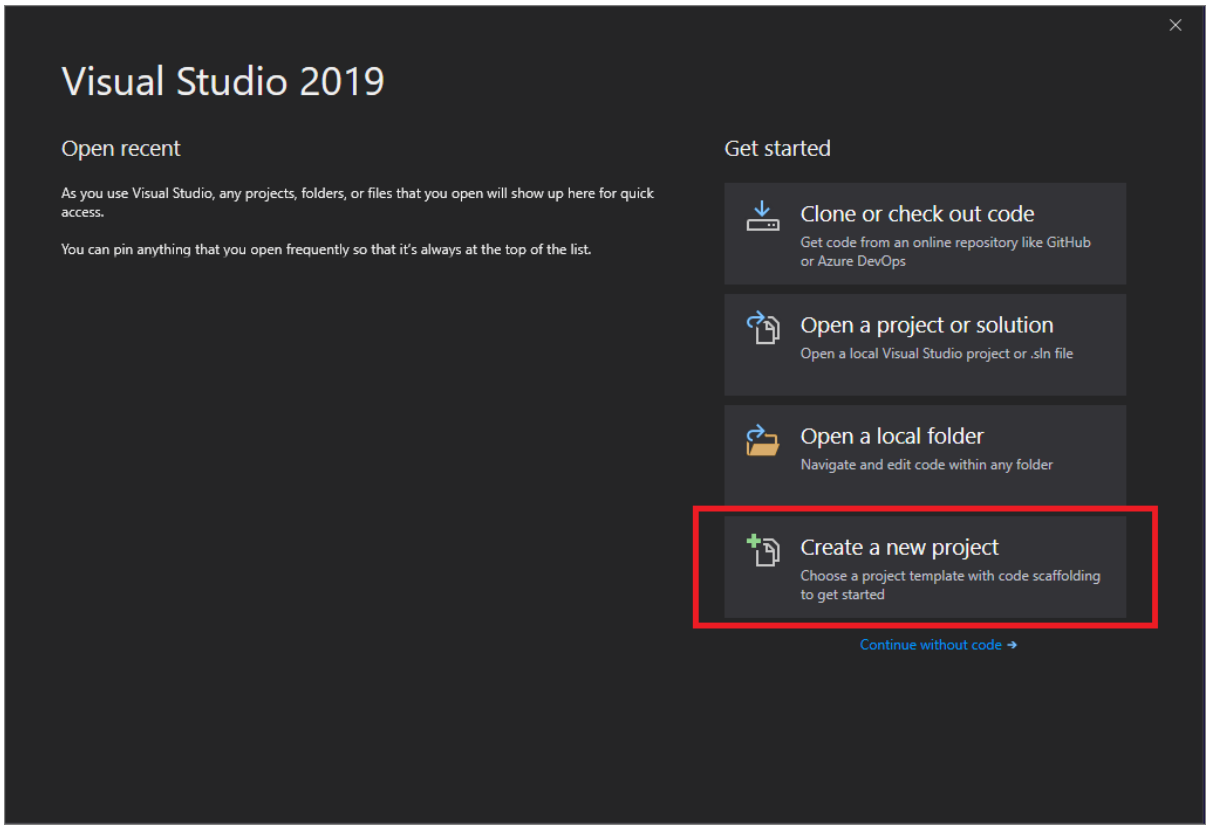
First, you'll create a C# application project. The project type comes with all the template files you'll need, before you've even added anything.

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**.

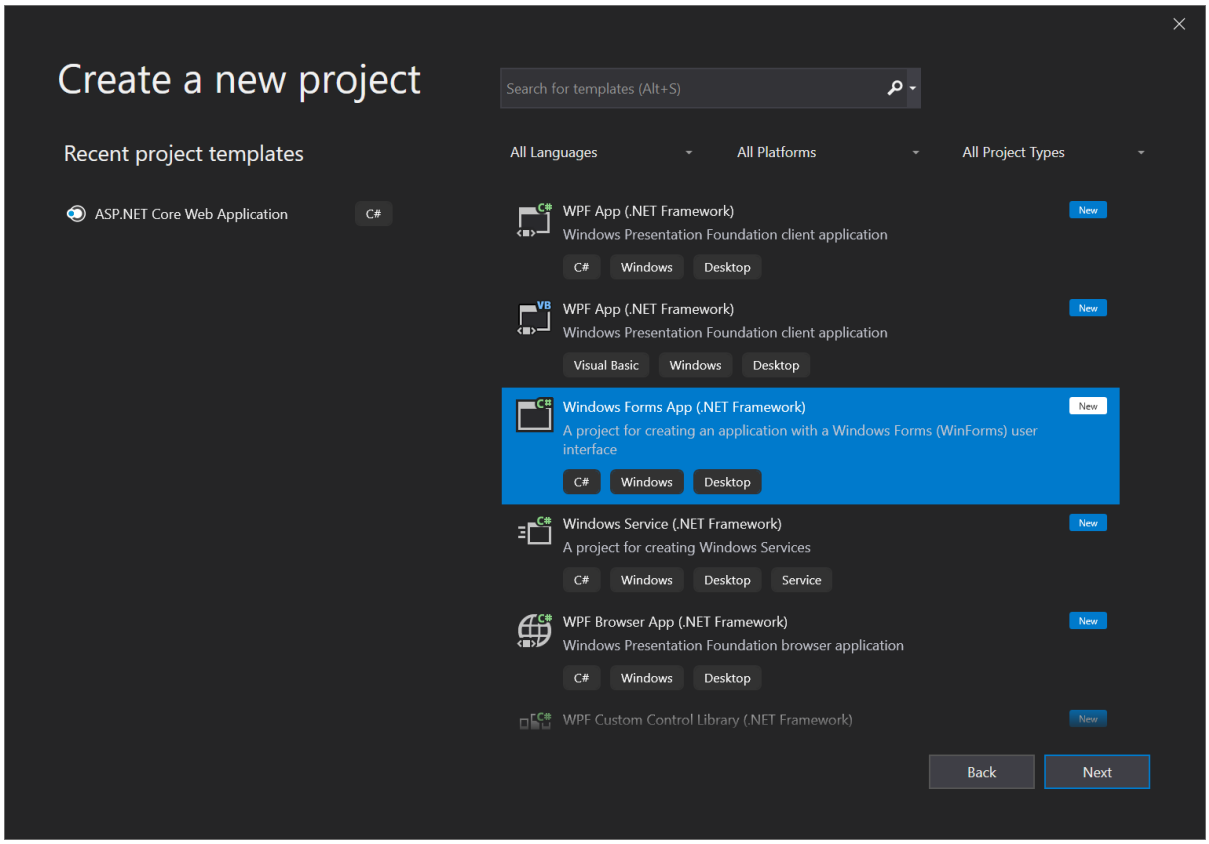


1. Open Visual Studio 2019.
2. On the start window, choose **Create a new project**.



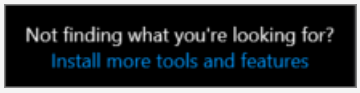
3. On the **Create a new project** window, choose the **Windows Forms App (.NET Framework)** template for C#.

   (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.)

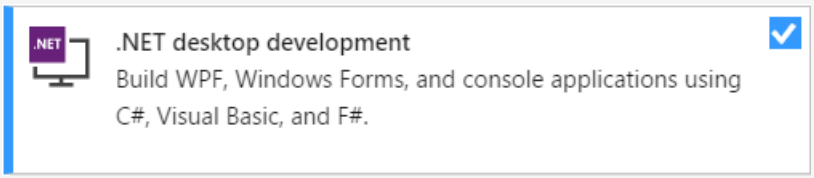


NOTE

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.

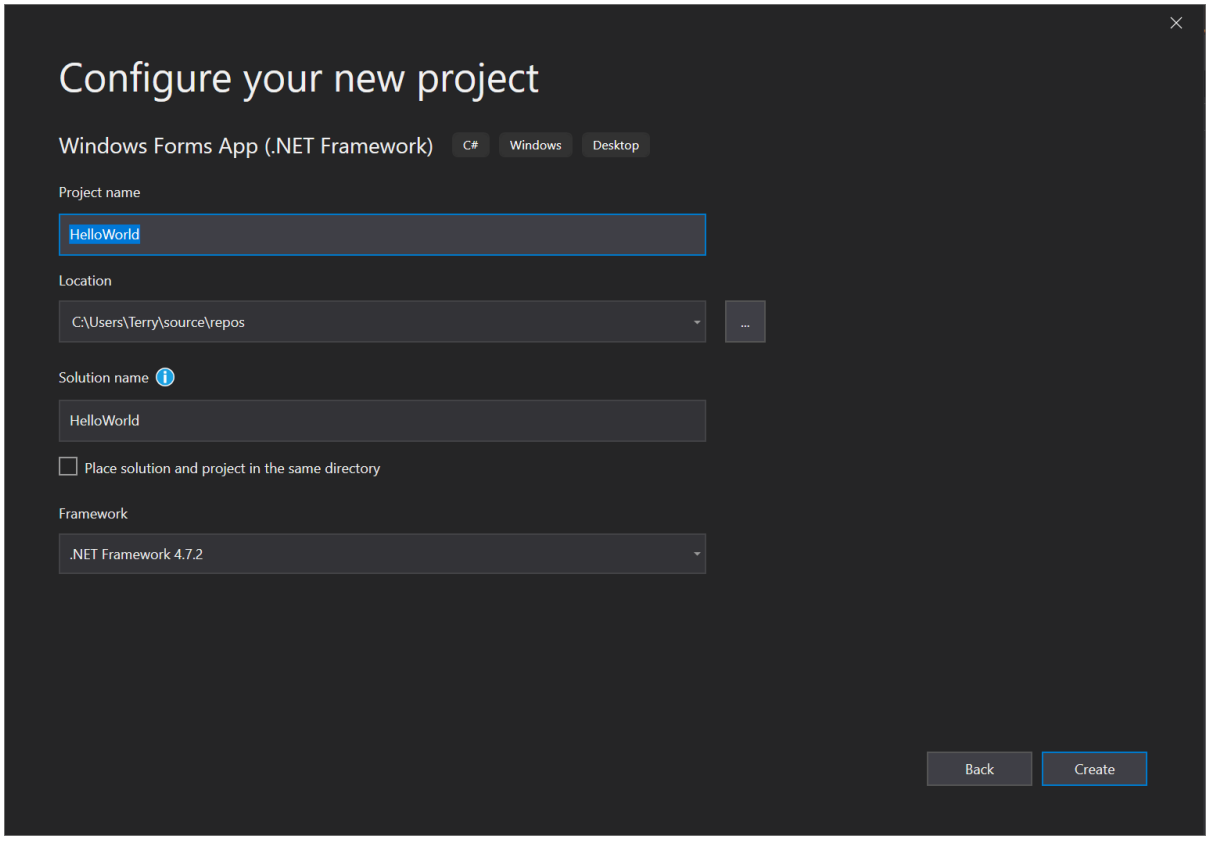


Next, in the Visual Studio Installer, choose the Choose the **.NET desktop development** workload.



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**.



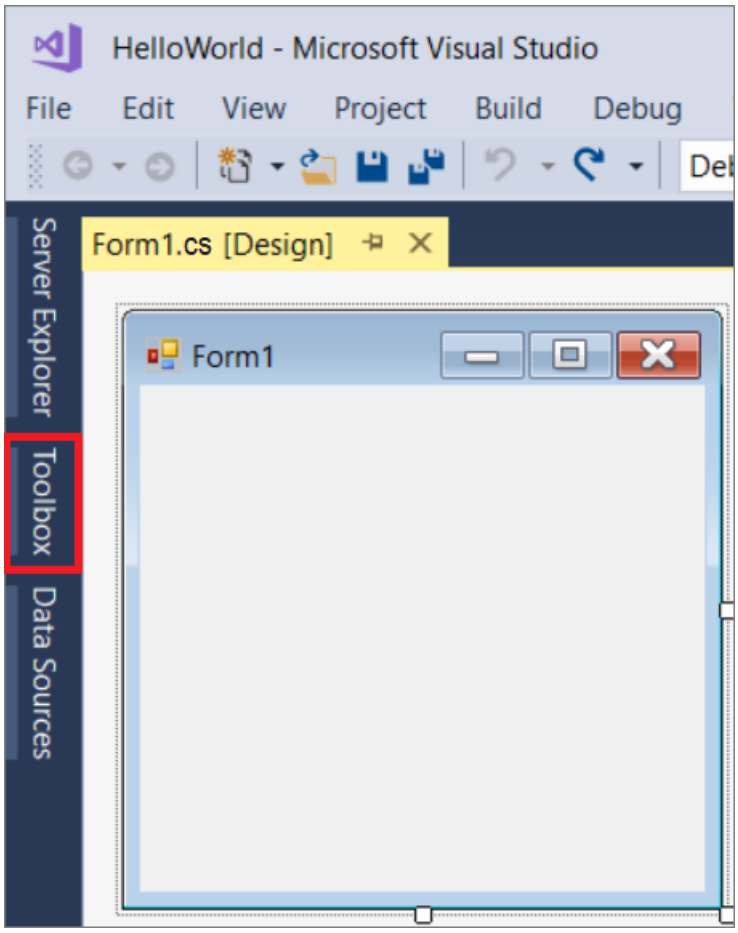
Visual Studio opens your new project.

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.

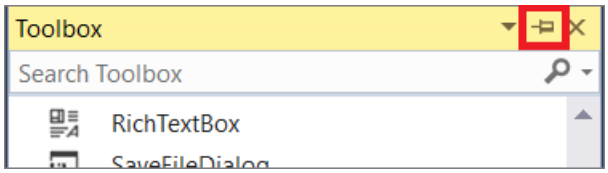
Add a button to the form

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

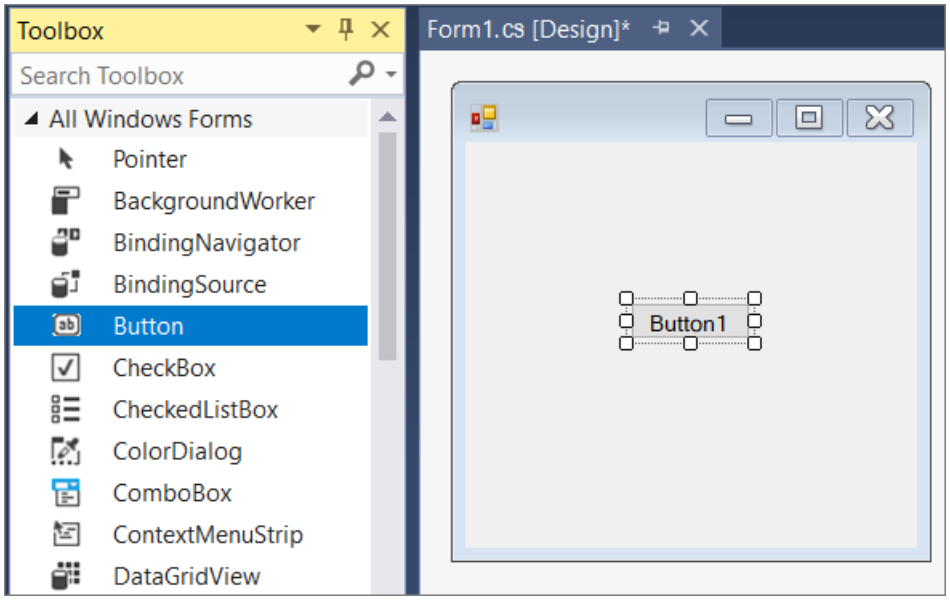


(If you don't see the **Toolbox** fly-out option, you can open it from the menu bar. To do so, **View** > **Toolbox**. Or, press **Ctrl**+**Alt**+**X**.)

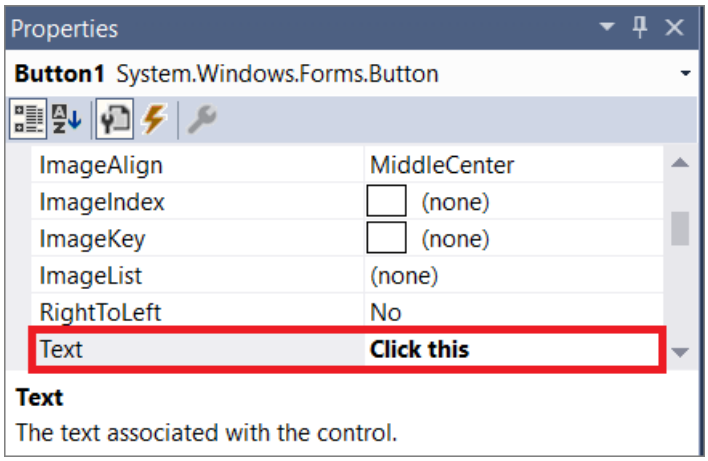
2. Choose the **Pin** icon to dock the **Toolbox** window.



3. Choose the **Button** control and then drag it onto the form.

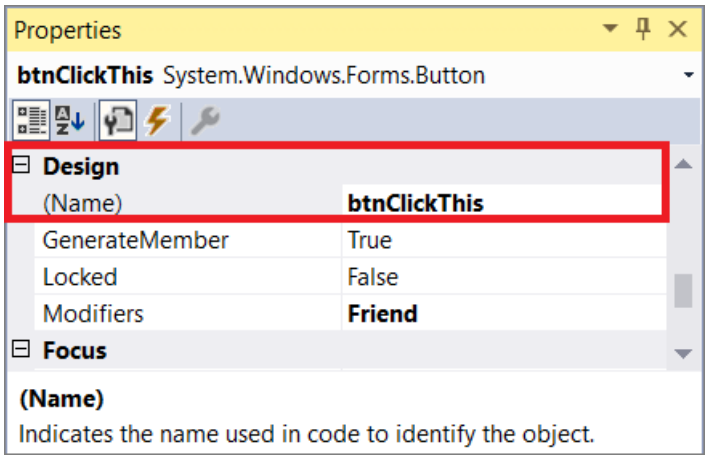


4. In the **Properties** window, locate **Text**, change the name from **Button1** to `Click this`, and then press **Enter**.



(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.

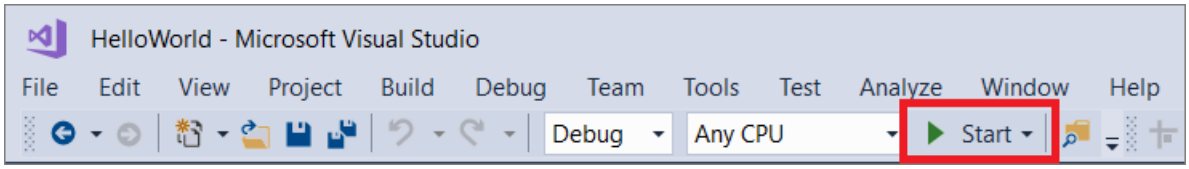
   (Alternatively, you can expand **Form1.cs** in **Solution Explorer**, and then choose **Form1**.)

2. In the **Form1.cs** window, after the **private void** line, type or enter `lblHelloWorld.Text = "Hello World!";` as shown in the following screenshot:

```
Form1.cs | Form1.cs [Design]
HelloWorld                                  HelloWorld.Form1
 1  using System;
 2  using System.Windows.Forms;
 3
 4  namespace HelloWorld
 5  {
 6      public partial class Form1 : Form
 7      {
 8          public Form1()
 9          {
10              InitializeComponent();
11          }
12
13          private void btnClickThis_Click(object sender, EventArgs e)
14          {
15              lblHelloWorld.Text = "Hello World!";
16          }
17      }
18  }
19
```

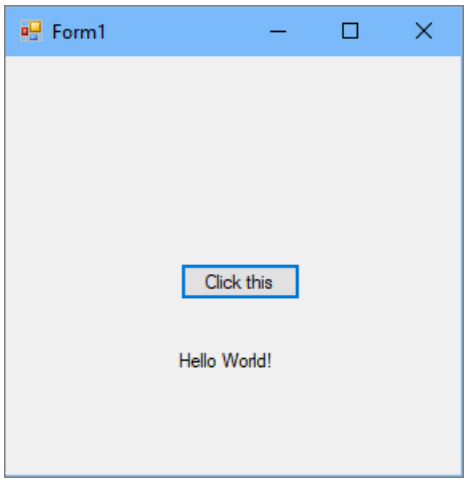
Run the application

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

Tutorial: Create your first ASP.NET Core App using Entity Framework with Visual Studio 2019

2/25/2020 • 2 minutes to read • Edit Online

In this tutorial, you'll create an ASP.NET Core web app that uses data, and deploy it to Azure. This tutorial consists of the following steps:

- Step 1: Install Visual Studio 2019
- Step 2: Create your first ASP.NET Core web app
- Step 3: Work with data using Entity Framework
- Step 4: Expose a web API from your ASP.NET Core app
- Step 5: Deploy your ASP.NET Core app to Azure

Step 1: Install Visual Studio 2019

Learn how to install Visual Studio 2019 with this video tutorial and step-by-step instructions. If you have already installed Visual Studio, skip ahead to Step 2: Create your first ASP.NET Core web app.

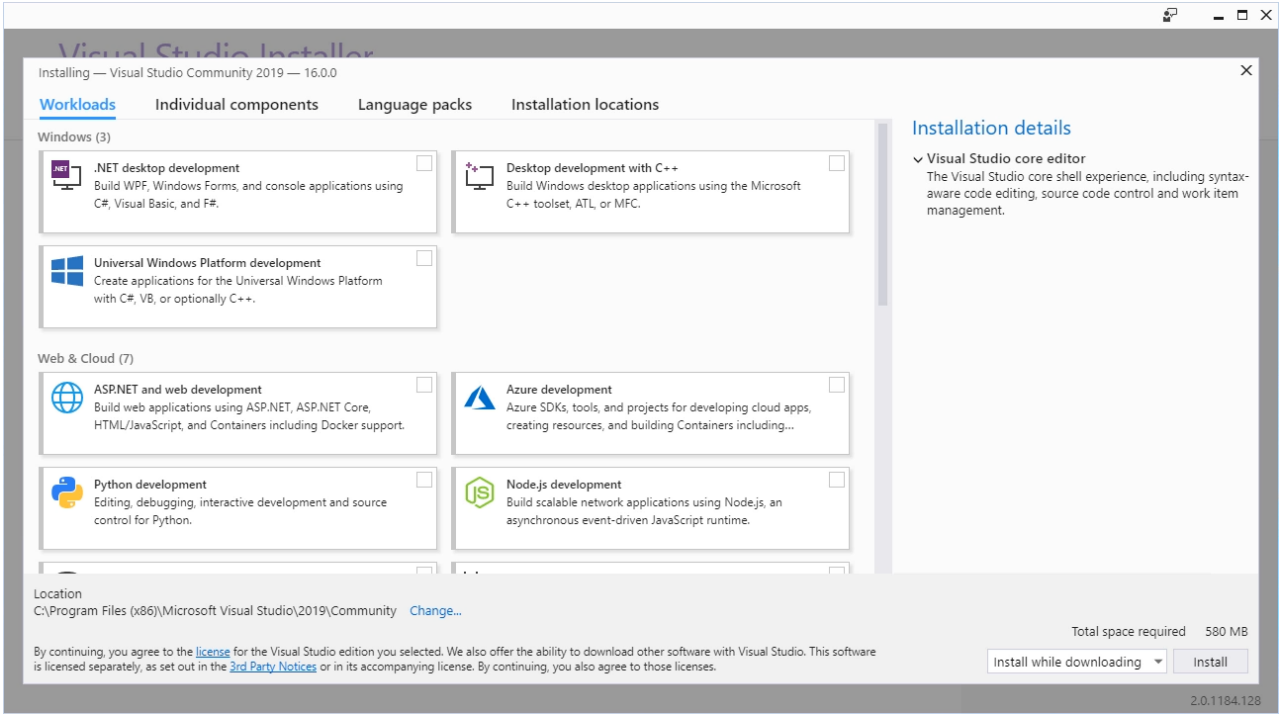
Watch this video and follow along to install Visual Studio and create your first ASP.NET Core app.

Download the Installer

Go to visualstudio.com to find the installer. Locate the Visual Studio 2019 link, and click it to start the download. For a free version of Visual Studio, choose Visual Studio Community.

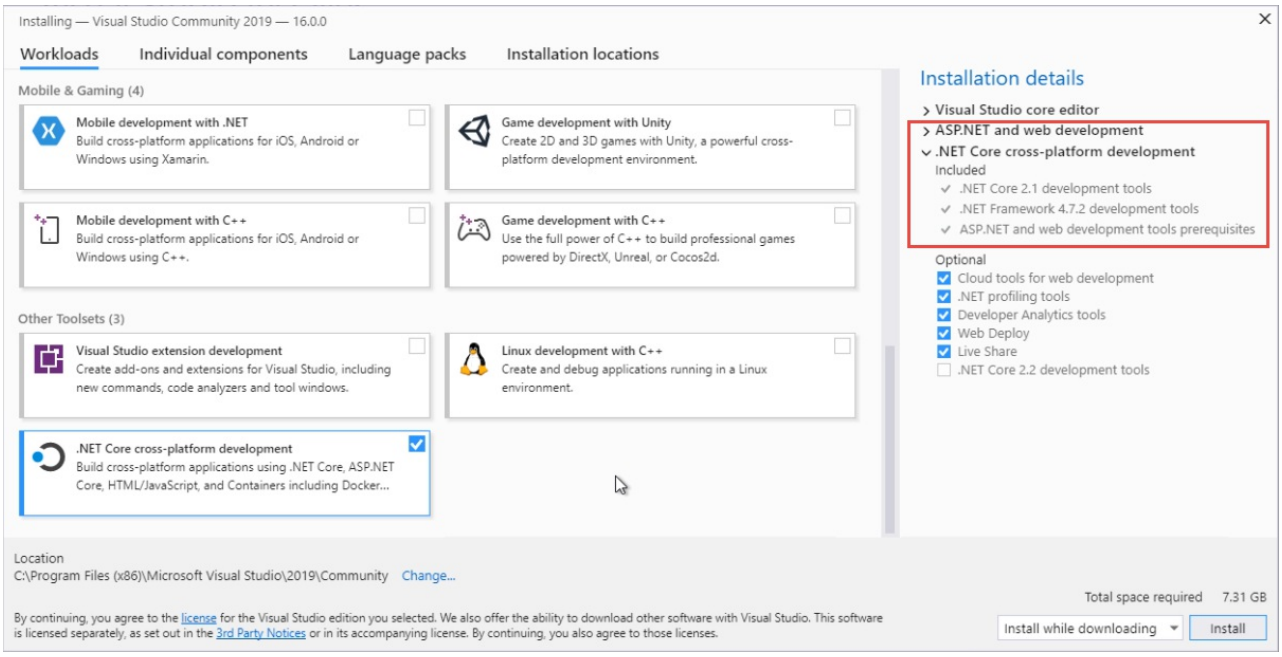
Start the Installer

Once the download has completed, click **Run** to start the installer.



Choose workloads

Visual Studio can be used for many different kinds of development, and workloads make it easy for you to download everything you need for the kind of apps you want to build. Choose **ASP.NET and Web Development** and **.NET Core cross-platform development** workloads for now. You can always relaunch the installer later to install additional workloads and components.

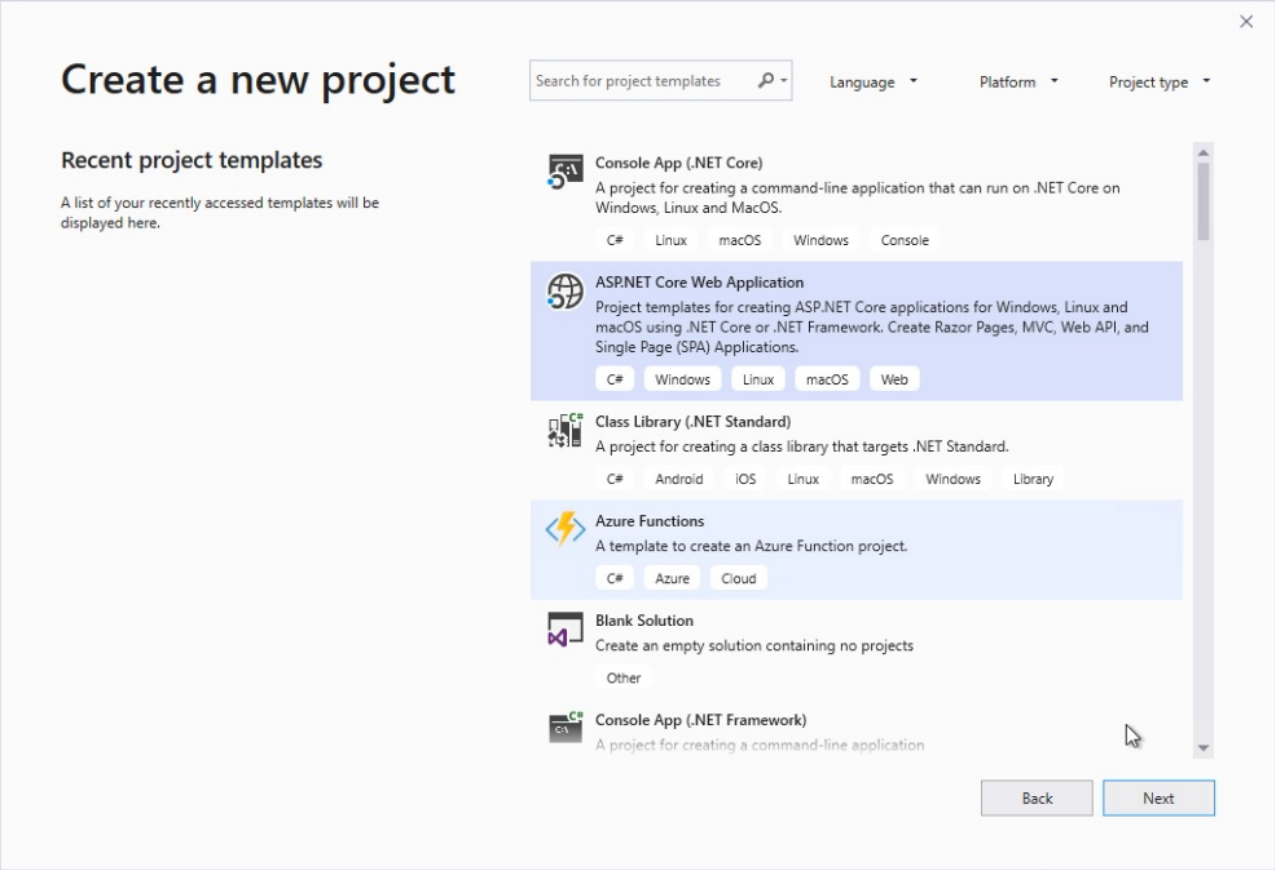


Install

Click **Install** and let the installer download and install Visual Studio.

Run Visual Studio for the first time

Visual Studio should launch automatically when the installer finishes. You may be prompted to sign in, which has some nice features associated with it, but for now you can choose to do so later. Next you can choose your theme and development settings. Once you've made these choices, you'll be ready to start your first project. Click **Create a new project** and then choose **ASP.NET Core Web Application**.



Explore ASP.NET Core project types

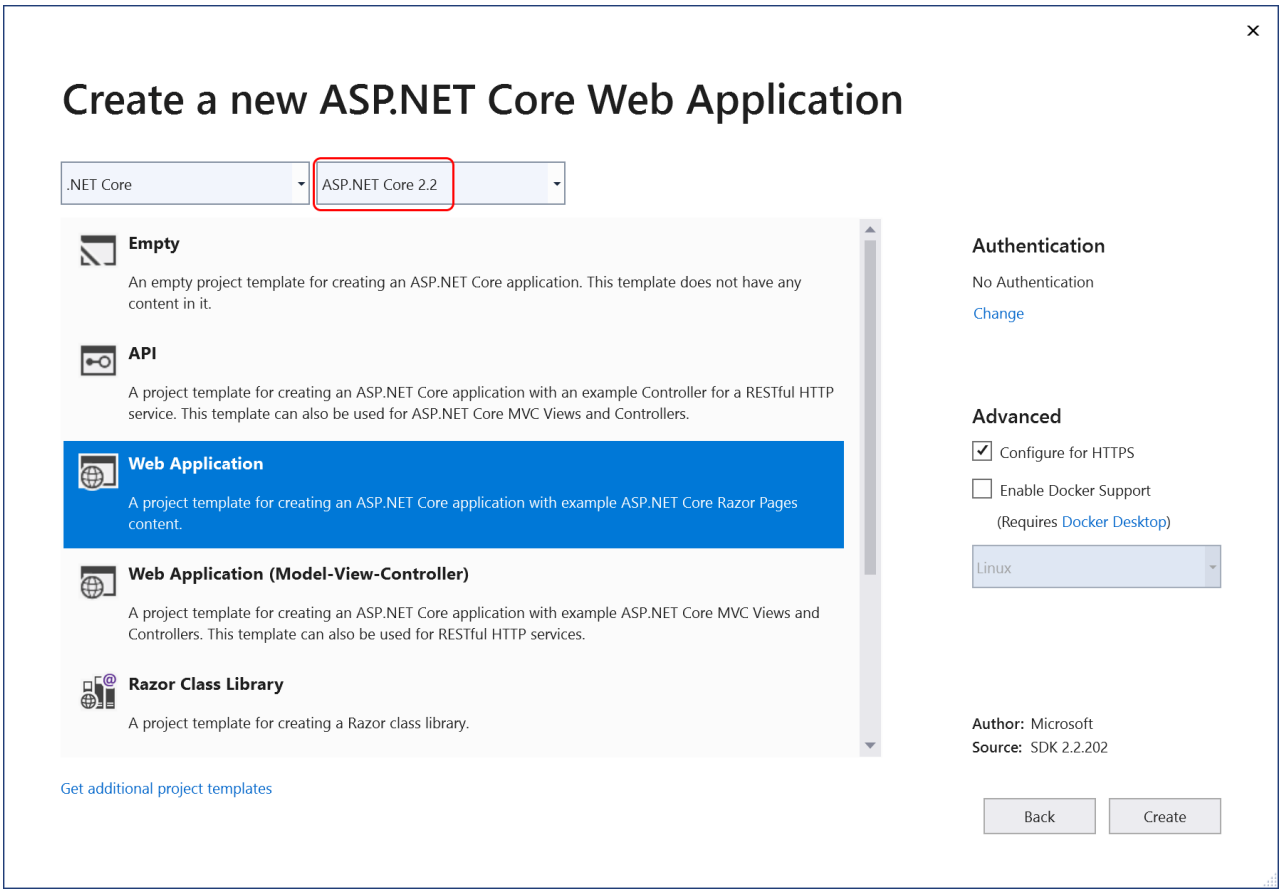
You can choose your project name and location, then pick **Create**. Now choose which template to use for your ASP.NET Core application. You can choose from the following options:

- Empty. An empty project template that lets you start from scratch.
- API. Best for web APIs.
- Web Application. A standard ASP.NET Core web application built with Razor Pages.
- Web Application (Model-View-Controller). A standard ASP.NET Core web application using the Model-View-Controller pattern.
- Angular.
- React.js.
- React.js / Redux.
- Razor Class Library. Used to share Razor assets between projects.

Note that for most of the project templates you can also choose to enable Docker support by checking a box. You can also add Authentication support by clicking the change Authentication button. From there you can choose from:

- No Authentication.
- Individual User Accounts. These are stored in a local or Azure-based database.
- Work or School Accounts. This option uses Active Directory, Azure AD, or Office 365 for authentication.
- Windows Authentication. Suitable for intranet applications.

Select the standard Web Application template with No Authentication and click **Create**.



Next steps

In the next video, you'll learn more about your first ASP.NET Core project.

Tutorial: Creating Your First ASP.NET Core Web App

See also

- Tutorial: Get started with C# and ASP.NET Core A more detailed tutorial without a video walkthrough

Step 2: Create your first ASP.NET Core web app

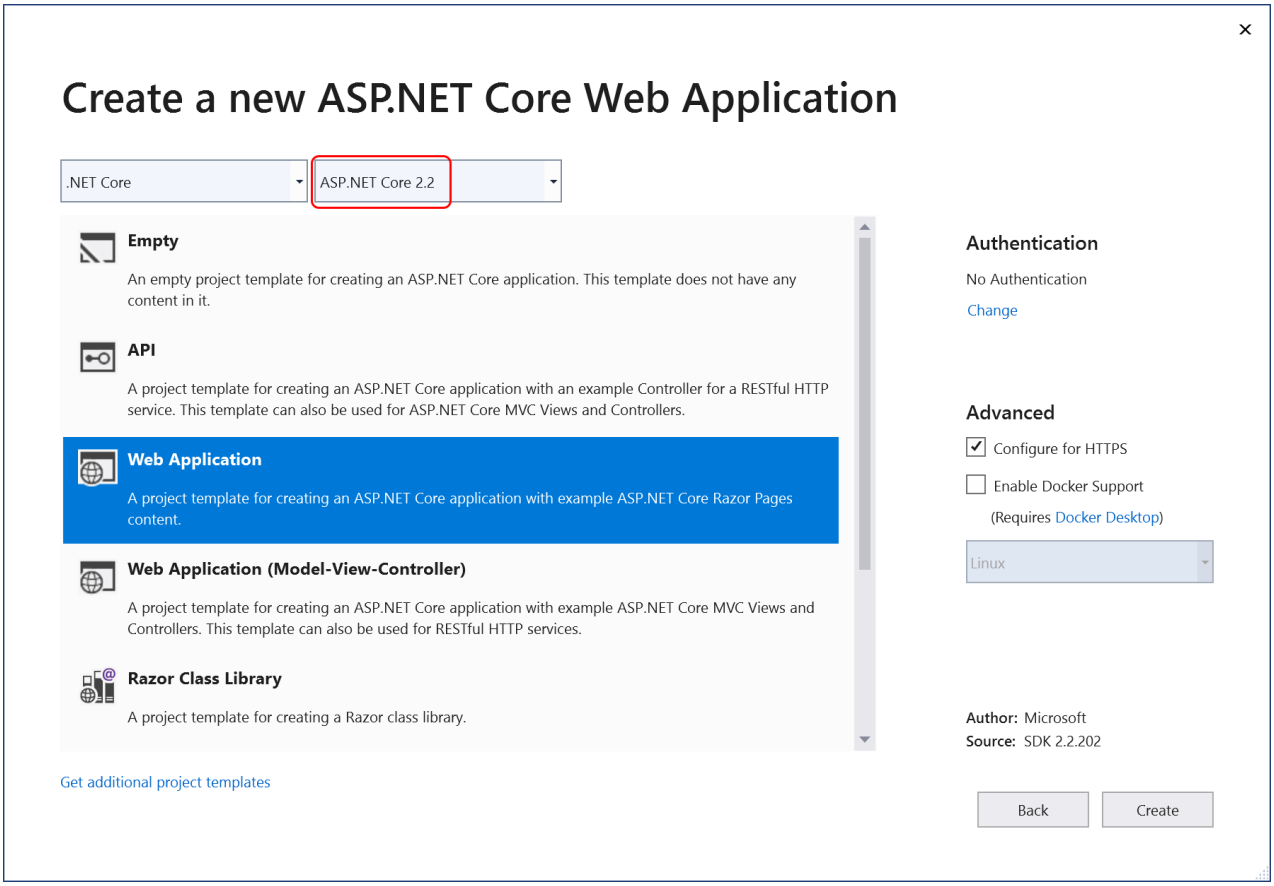
2/25/2020 • 3 minutes to read • Edit Online

Create your first ASP.NET Core Web App with this video tutorial and step-by-step instructions.

Watch this video and follow along to create your first ASP.NET Core app.

Start Visual Studio 2019 and create a new project

Start Visual Studio 2019 and click **Create new project**. Choose **ASP.NET Core Web Application**. Choose the **Web Application** template and keep the default project name and location. In the dropdown with the ASP.NET Core version, choose **ASP.NET Core 2.1** or **ASP.NET Core 2.2**. Click **Create**. For more detailed instructions, refer to the previous video in this tutorial series.

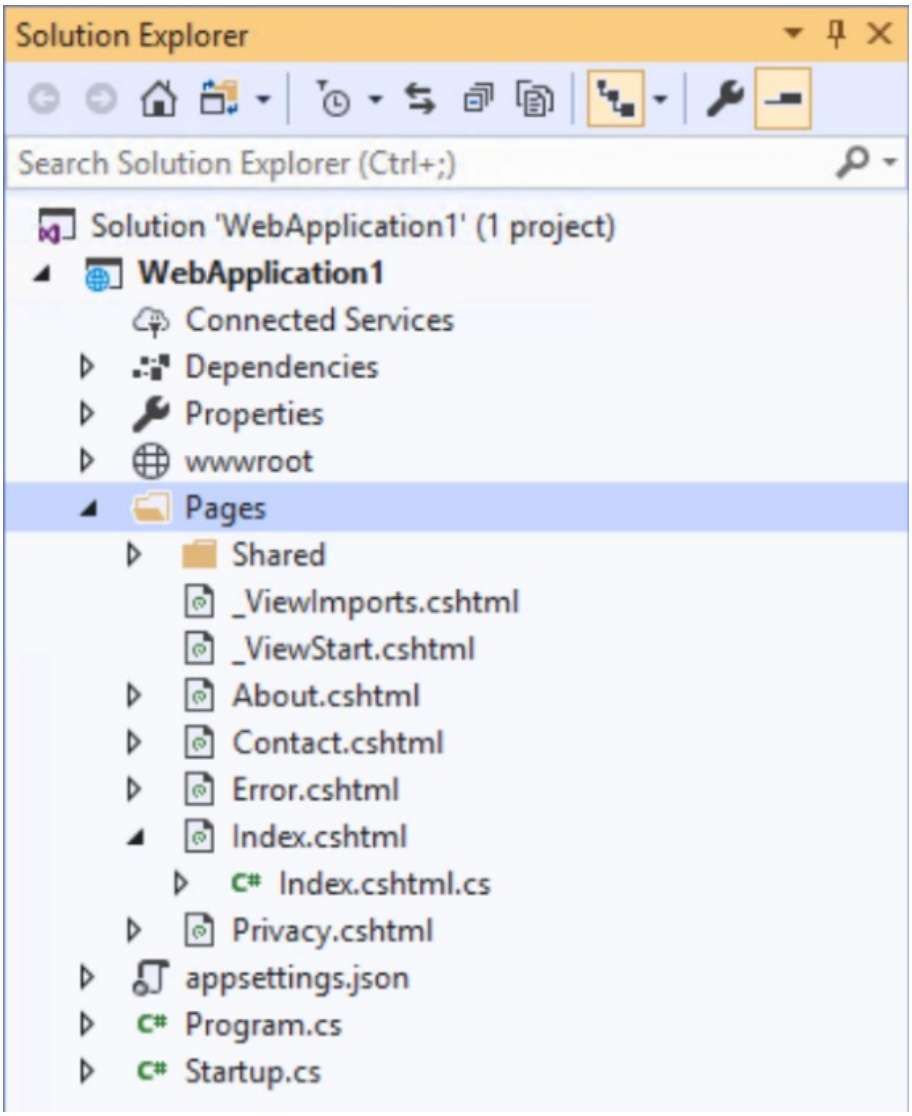


WARNING

Make sure you choose ASP .NET Core 2.1 or ASP.NET Core 2.2. This tutorial is not compatible with ASP.NET Core 3.x.

Explore the new project

In the solution explorer window on the right, you can view the contents of the new project. They're described here.



wwwroot

The *wwwroot* folder holds static files that will be publicly accessible from the web application. It typically holds stylesheets, client-side script files, and images.

Pages

The *Pages* folder holds the site's Razor Pages. The default template provides several pages, including the *Index.cshtml* page that is the application home page, as well as About, Contact, and so on.

appsettings.json

This file holds configuration settings for the site, in JSON format.

Program.cs

This file acts as the entry point for the application. When the app is run, its Main method is the first method that is run, and is responsible for creating the Web Host that will contain the application.

Startup.cs

The Web Host created in *Program.cs* references the Startup class and calls its methods to configure the application. The ConfigureServices method is responsible for setting up any services the app will use. The `Configure` method sets up the app's HTTP request pipeline. Each request goes through this pipeline, interacting with each piece of *middleware* as it does so.

Index.cshtml

The home page for the site includes some HTML markup and some server side Razor code. It uses Razor to specify the page model, `IndexModel`, which is located in the associated *Index.cshtml.cs* file. It also sets the page title by

setting a value in ViewData. This ViewData value is read in the *_Layout.cshtml* file, located in the Shared folder inside the Pages folder. The Layout file is shared by many Razor Pages and provides the common look and feel for the application. Each page's content is rendered within the Layout file's HTML.

Run the application

Now run the application and view it in the browser. You can run the application using **Ctrl+F5** or by choosing **Debug** > **Start Without Debugging** from Visual Studio's menu.

Customize the application

Add a property to the *Index.cshtml.cs* file and set its value to the current time in the `OnGet` handler:

```
public string Time { get; set; }
public void OnGet()
{
    Time = DateTime.Today.ToShortTimeString();
}
```

Replace the `<div>` content in *Index.cshtml* with this markup:

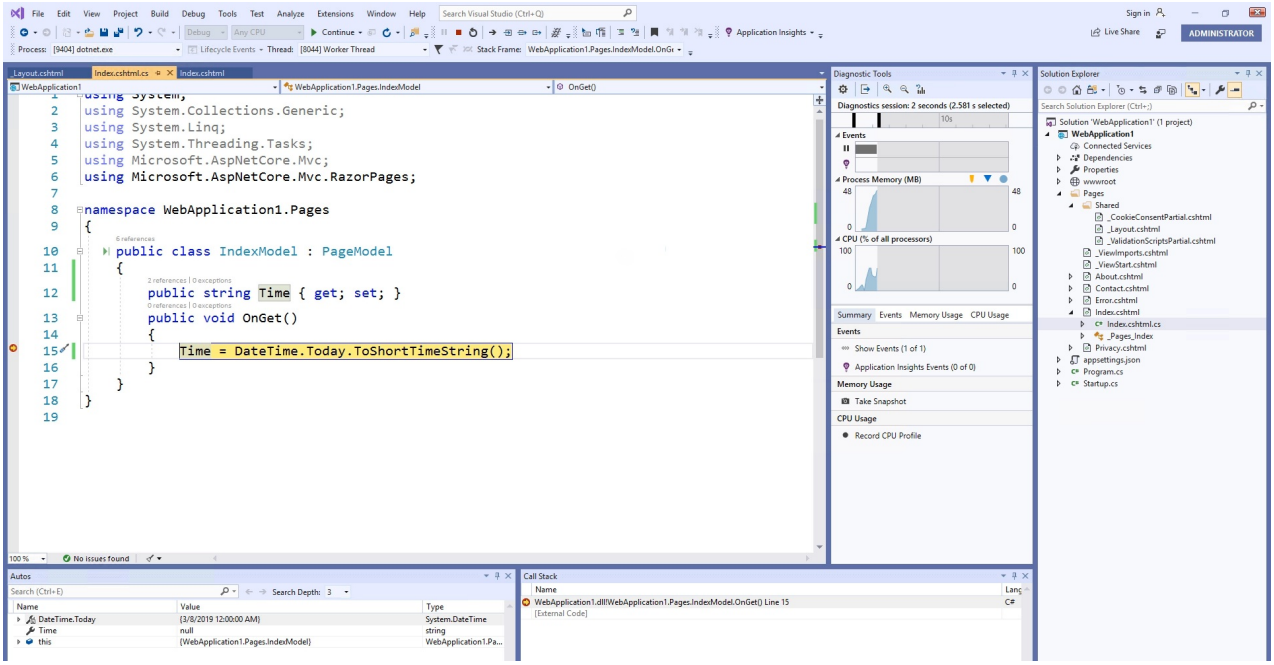
```
<h2>It's @Model.Time right now on the server!</h2>
```

Run the application again. You should see that the page now displays the current time, but it's always midnight! That's not right.

Debug the application

Add a breakpoint to the `OnGet` method where we're assigning a value to `Time` and this time start debugging the application.

Execution stops on the line, and you can see that `DateTime.Today` includes the date but the time is always midnight because it doesn't include time data.

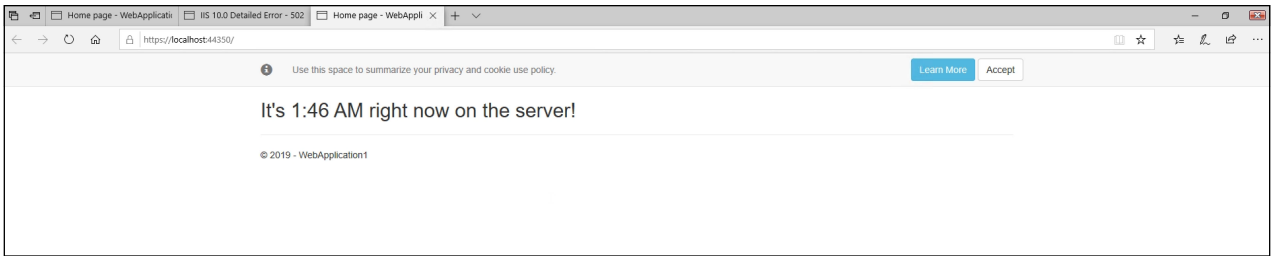


Change it to use `DateTime.Now` and continue executing. The new code for `OnGet` should be:

```
public void OnGet()
{
    Time = DateTime.Now.ToShortTimeString();
}
```

You should now see the actual server time in the browser when you navigate to the app.

> **NOTE**
>
> Your output might differ from the image, since the output format of ToShortDateTimeString depends on the current culture setting. See ToShortTimeString().



Next steps

In the next video, you'll learn how to add data support to your app.

Tutorial: Working with Data in Your ASP.NET Core App

See also

- Tutorial: Create a Razor Pages web app with ASP.NET Core

Step 3: Work with data using Entity Framework

2/25/2020 • 3 minutes to read • Edit Online

Follow these steps to start working with data using Entity Framework Core in your ASP.NET Core Web App.

Watch this video and follow along to add data to your first ASP.NET Core app.

Open your project

If you're following along with these videos, open the Web Application project you created in the previous section. If you're starting here, you need to create a new project and choose **ASP.NET Web Application** and then **Web Application**. Leave the rest of the options as defaults.

Add your model

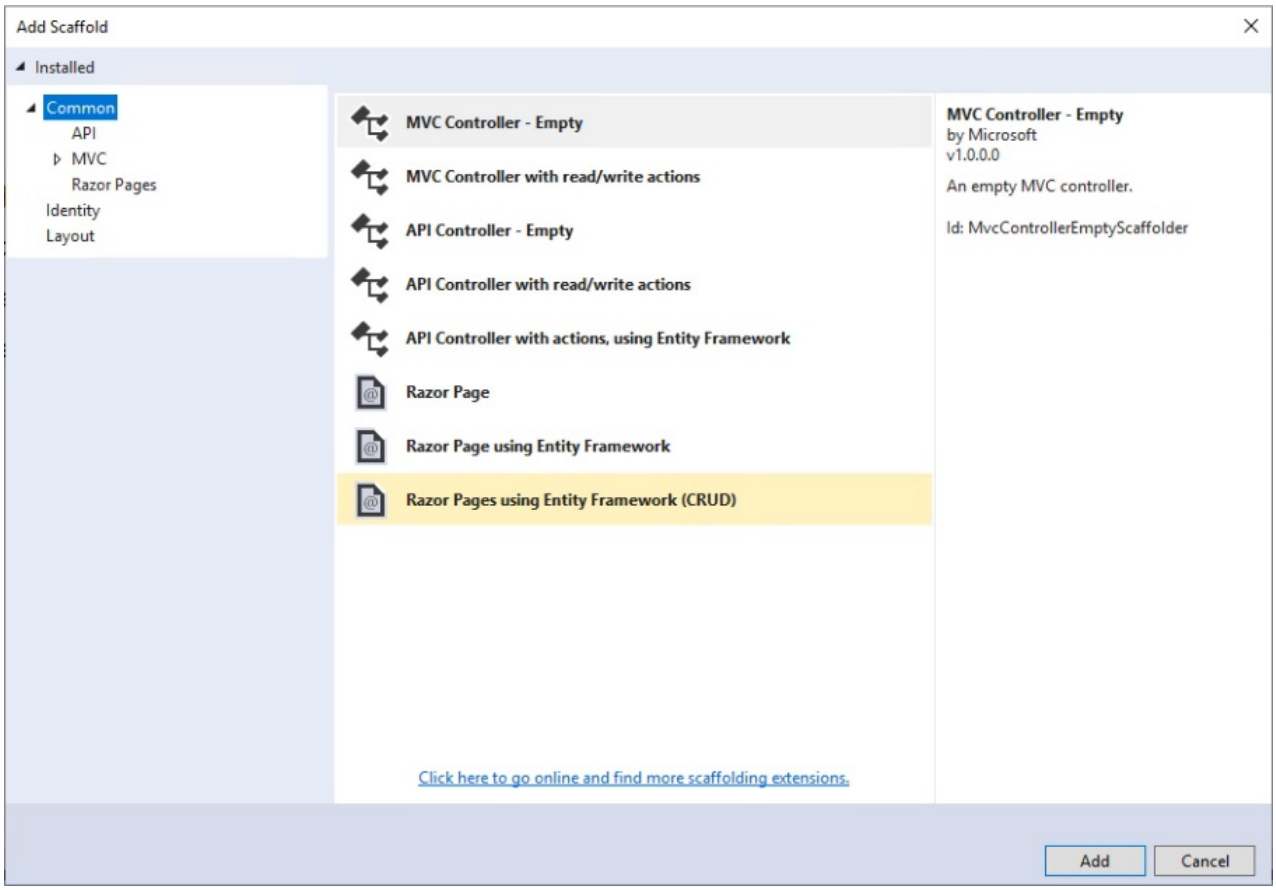
The first thing you need to do to work with data in your ASP.NET Core application is to describe what the data should look like. We call that creating a *model* of the things involved in the problem we're trying to solve. In real world applications, we'll add custom business logic to these models so they can behave in certain ways and automate tasks for us. For this sample, we're going to create a simple system for tracking board games. We need a class that represents a game, and includes some properties that we might want to record about that game, like how many players it can support. This class will go into a new folder we'll create in the root of the web project, called *Models*.

```
public class Game
{
    public int Id { get; set; }
    public string Title { get; set; }
    public int PublicationYear { get; set; }
    public int MinimumPlayers { get; set; }
    public int MaximumPlayers { get; set; }
}
```

Create the pages to manage your game library

Now we're ready to create the pages we'll use to manage our library of games. This might sound daunting but it's actually amazingly easy. First we need to decide where in our app this functionality should live. Open the Pages folder in the web project and add a new folder there. Call it *Games*.

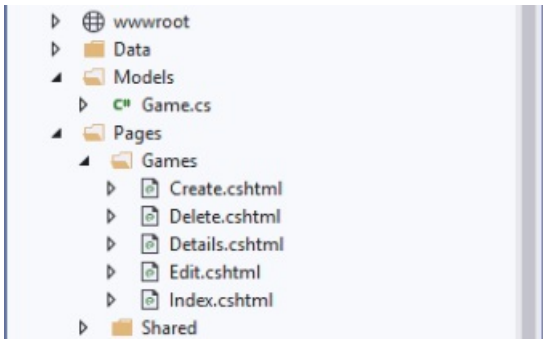
Now right click on Games and choose **Add** > **New Scaffolded Item**. Choose the Razor Pages using **Entity Framework (CRUD)** option. CRUD stands for "Create, Read, Update, Delete" and this template will create pages for each of these operations (including a "list all" page and a "view details of one item" page).



Select your Game model class and use the '+' icon to add a new Data context class. Name it `AppDbContext`. Leave the rest as defaults and click **Add**.

You will see the following Razor Pages added to your Games folder:

- Create.cshtml
- Delete.cshtml
- Details.cshtml
- Edit.cshtml
- Index.cshtml



In addition to adding pages in the *Games* folder, the scaffolding operation added code to my *Startup.cs* class. Looking in the `ConfigureServices` method in this class you will see this code has been added:

```
services.AddDbContext<AppDbContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("AppDbContext")));
```

You'll also find the `AppDbContext` connection string has been added to the project's *appsettings.json* file.

If you run the app now, it may fail because no database has been created, yet. You can configure the app to automatically create the database if needed by adding some code to Program.cs:

```
public static void Main(string[] args)
{
    var host = CreateWebHostBuilder(args).Build();

    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;

        try
        {
            var context = services.GetRequiredService<AppDbContext>();
            context.Database.EnsureCreated();
        }
        catch (Exception ex)
        {
            var logger = services.GetRequiredService<ILogger<Program>>();
            logger.LogError(ex, "An error occurred creating the DB.");
        }
    }

    host.Run();
}
```

To resolve the typenames in the preceding code, add the following using statements to *Program.cs* at the end of the existing block of using statements:

```
using Microsoft.Extensions.DependencyInjection;
using WebApplication1.Models;
```

Be sure to use your project name instead of WebApplication1 in your code.

Most of the code is just for error handling and to provide access to the EF Core `AppDbContext` before the app is running. The important line is the one that says `context.Database.EnsureCreated()`, which will create the database if it doesn't already exist. Now the app is ready to run.

Test it out

Run the application and navigate to `/Games` in the address bar. You will see an empty list page. Click **Create New** to add a new `Game` to the collection. Fill in the form and click **Create**. You should see it in the list view. Click on **Details** to see the details of a single record.

Add another record. You can click *Edit* to change the details of a record, or **Delete** to remove it, which will prompt you to confirm before it actually deletes the record.

That's all it took to start working with data in an ASP.NET Core app using EF Core and Visual Studio 2019.

Next steps

In the next video, you'll learn how to add web API support to your app.

Step 4: Exposing a web API From Your ASP.NET Core App

See also

- Razor Pages with Entity Framework Core in ASP.NET Core
- ASP.NET Core Razor Pages with EF Core

Step 4: Expose a web API from your ASP.NET Core app

2/25/2020 • 4 minutes to read • Edit Online

Follow these steps to add a web API to your existing ASP.NET Core app.

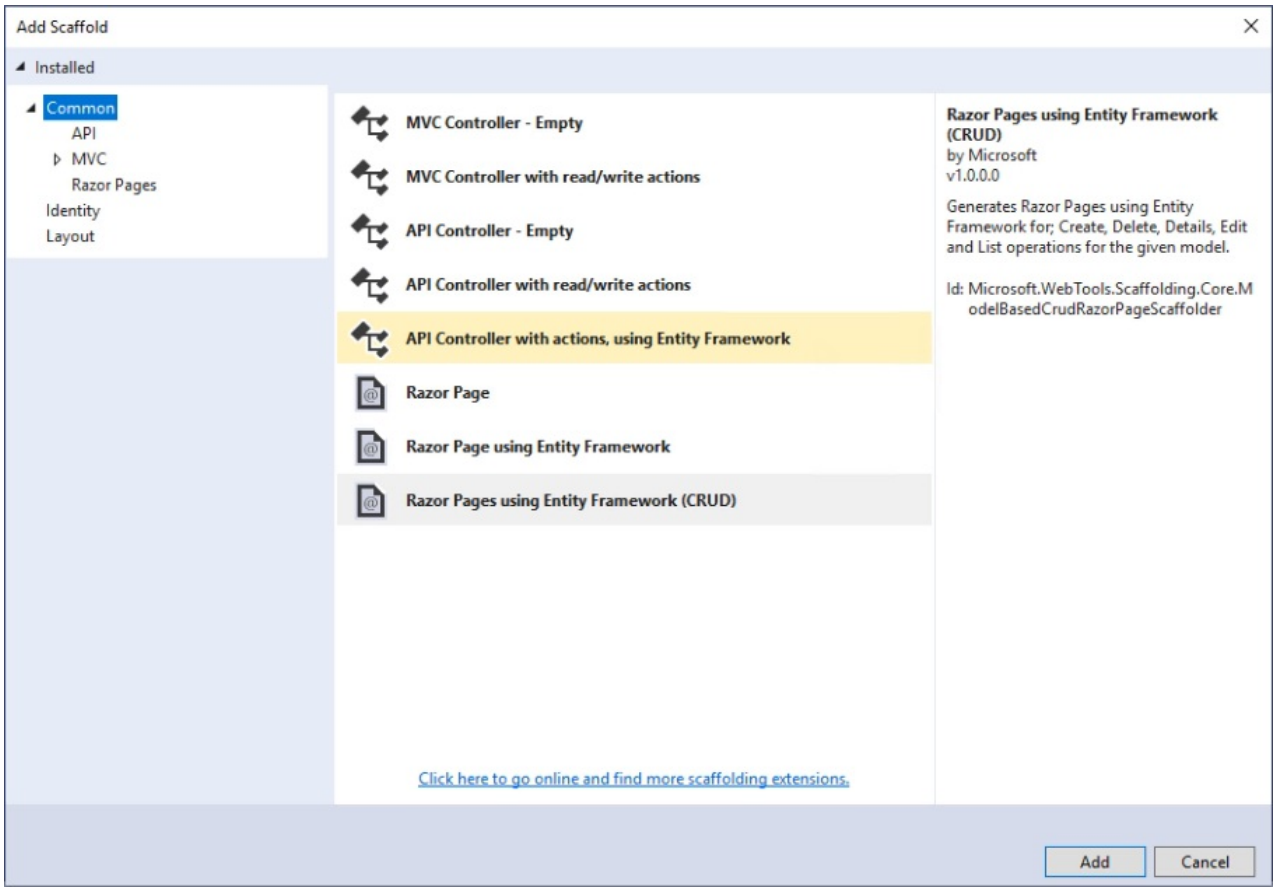
Watch this video and follow along to add web API support to your first ASP.NET Core app.

Open your project

Open your ASP.NET Core app in Visual Studio 2019. The app should already be using EF Core to manage your model types, as configured in step 3 of this tutorial series.

Add an API controller

Right click on the project and add a new folder called *Api*. Then, right click on this folder and choose **Add** > **New Scaffolded Item**. Choose **API Controller with actions, using Entity Framework**. Now choose an existing model class and click **Add**.



Reviewing the generated controller

The generated code includes a new controller class. At the top of the class definition are two attributes.

```
[Route("api/[controller]")]
[ApiController]
public class GamesController : ControllerBase
```

The first one specifies the route for actions in this controller as being `api/[controller]` which means if the controller is named `GamesController` the route will be `api/Games`.

The second attribute, `[ApiController]`, adds some useful validations to the class, such as ensuring every action method includes its own `[Route]` attribute.

```
public class GamesController : ControllerBase
{
    private readonly AppDbContext _context;

    public GamesController(AppDbContext context)
    {
        _context = context;
    }
```

The controller uses the existing `AppDbContext`, passed into its constructor. Each action will use this field to work with the application's data.

```
// GET: api/Games
[HttpGet]
public IEnumerable<Game> GetGame()
{
    return _context.Game;
}
```

The first method is a simple GET request, as specified using the `[HttpGet]` attribute. It takes no parameters and returns a list of all games in the database.

```
// GET: api/Games/5
[HttpGet("{id}")]
public async Task<IActionResult> GetGame([FromRoute] int id)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var game = await _context.Game.FindAsync(id);

    if (game == null)
    {
        return NotFound();
    }

    return Ok(game);
}
```

The next method specifies `{id}` in the route, which will be added to the route following a `/` so the full route will be something like `api/Games/5` as shown in the comment at the top. The `id` input is mapped to the `id` parameter on the method. Inside the method, if the model is invalid, a `BadRequest` result is returned. Otherwise, EF will attempt to find the record matching the provided `id`. If it can't a `NotFound` result is returned, otherwise the appropriate `Game` record is returned.

```
// PUT: api/Games/5
[HttpPut("{id}")]
public async Task<IActionResult> PutGame([FromRoute] int id, [FromBody] Game game)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    if (id != game.Id)
    {
        return BadRequest();
    }

    _context.Entry(game).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!GameExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}
```

Next, an `[HttpPut]` request made to the API is used to perform updates. The new `Game` record is provided in the body of the request. Some validation and error checking is performed, and if everything is successful the record in the database is updated with the values provided in the body of the request. Otherwise an appropriate error response is returned.

```
// POST: api/Games
[HttpPost]
public async Task<IActionResult> PostGame([FromBody] Game game)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    _context.Game.Add(game);
    await _context.SaveChangesAsync();

    return CreatedAtAction("GetGame", new { id = game.Id }, game);
}
```

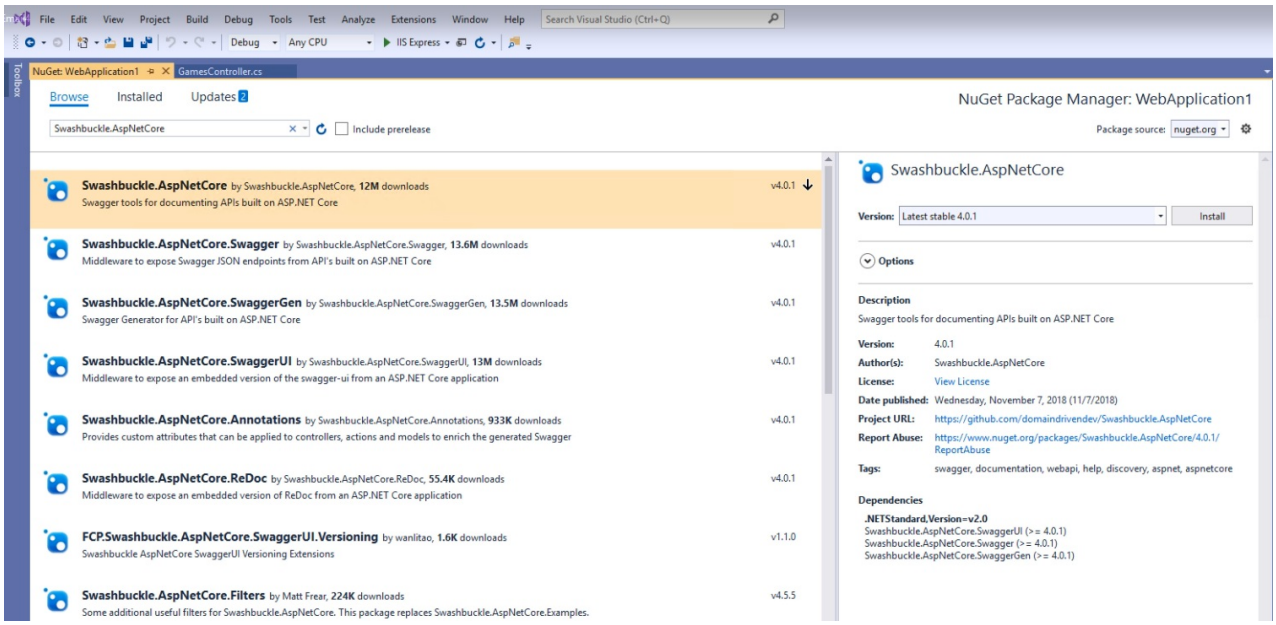
An `[HttpPost]` request is used to add new records to the system. As with the `[HttpPut]`, the record is added in the body of the request. If it's valid, EF Core adds the record to the database and the action returns the updated record (with its database generated ID) and a link to the record in the API.

```
// DELETE: api/Games/5
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteGame([FromRoute] int id)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var game = await _context.Game.FindAsync(id);
    if (game == null)
    {
        return NotFound();
    }

    _context.Game.Remove(game);
    await _context.SaveChangesAsync();

    return Ok(game);
}
```

Finally, an `[HttpDelete]` route is used with an ID to delete a record. If the request is valid and a record with the given ID exists, EF Core delete it from the database.

Adding Swagger

Swagger is an API documentation and testing tool that can be added as a set of services and middleware to an ASP.NET Core app. To do so, right-click on the project and choose **Manage NuGet Packages**. Then, click **Browse**, search for `Swashbuckle.AspNetCore`, and install the 4.0.1 version.



Once installed, open `Startup.cs` and add the following to the end of the `ConfigureServices` method:

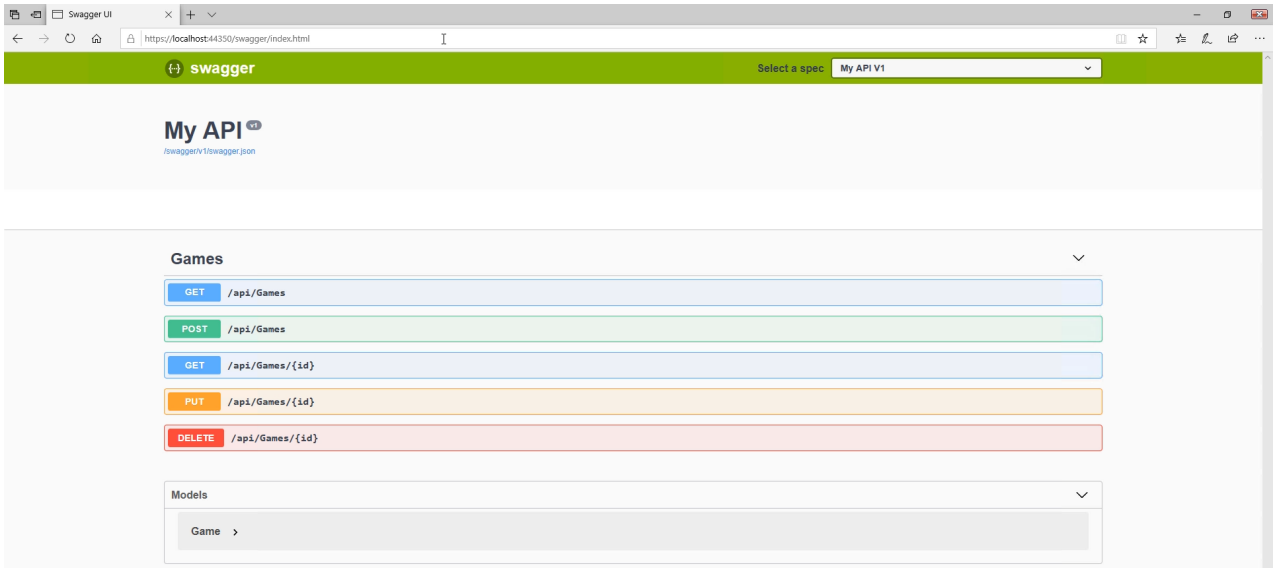
```
services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new Info { Title = "My API", Version = "v1" });
});
```

You'll also need to add `using Swashbuckle.AspNetCore.Swagger;` at the top of the file.

Next, add the following to the `Configure` method, just before `UseMvc`:

```
// Enable middleware to serve generated Swagger as a JSON endpoint.
app.UseSwagger();

// Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.),
// specifying the Swagger JSON endpoint.
app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
});
```

Now you should be able to build and run your app. In the browser, navigate to `/swagger` in the address bar. You should see a list of your app's API endpoints and models.



Click an endpoint under Games, then `Try it out` and `Execute` to see how the different endpoints behave.

Next steps

In the next video, you'll learn how to deploy your app to Azure.

Step 5: Deploying Your ASP.NET Core App to Azure

See also

- Getting Started with Swashbuckle and ASP.NET Core
- ASP.NET Core web API help pages with Swagger / OpenAPI

Step 5: Deploy your ASP.NET Core app to Azure

2/25/2020 • 2 minutes to read • Edit Online

Follow these steps to deploy your ASP.NET Core app and its database to Azure.

Watch this video and follow along to deploy your first ASP.NET Core app to Azure.

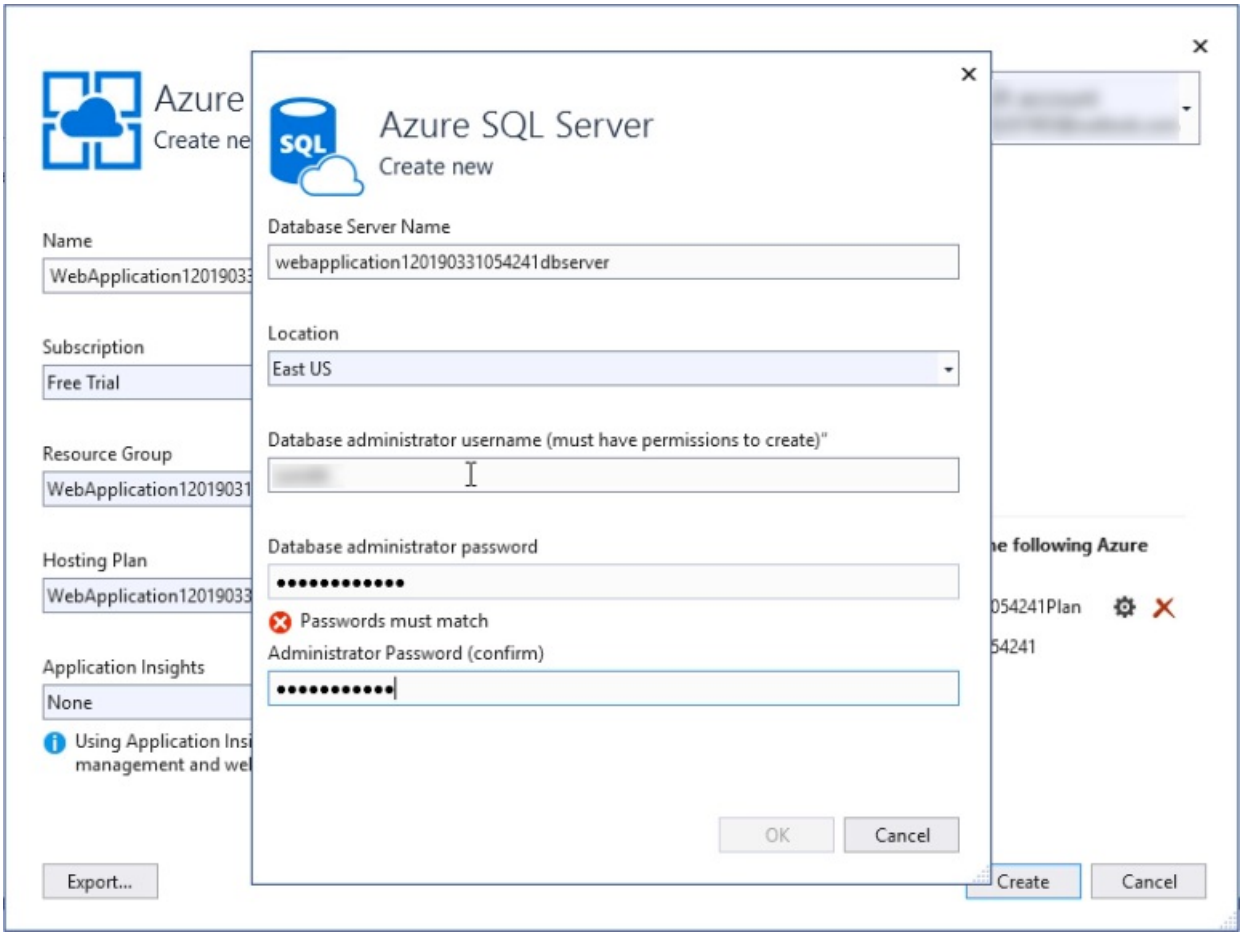
Open your project

Open your ASP.NET Core app in Visual Studio 2019. The app should already be using set up with EF Core and a working web API, as configured in step 4 of this tutorial series.

Publish to Azure App Service

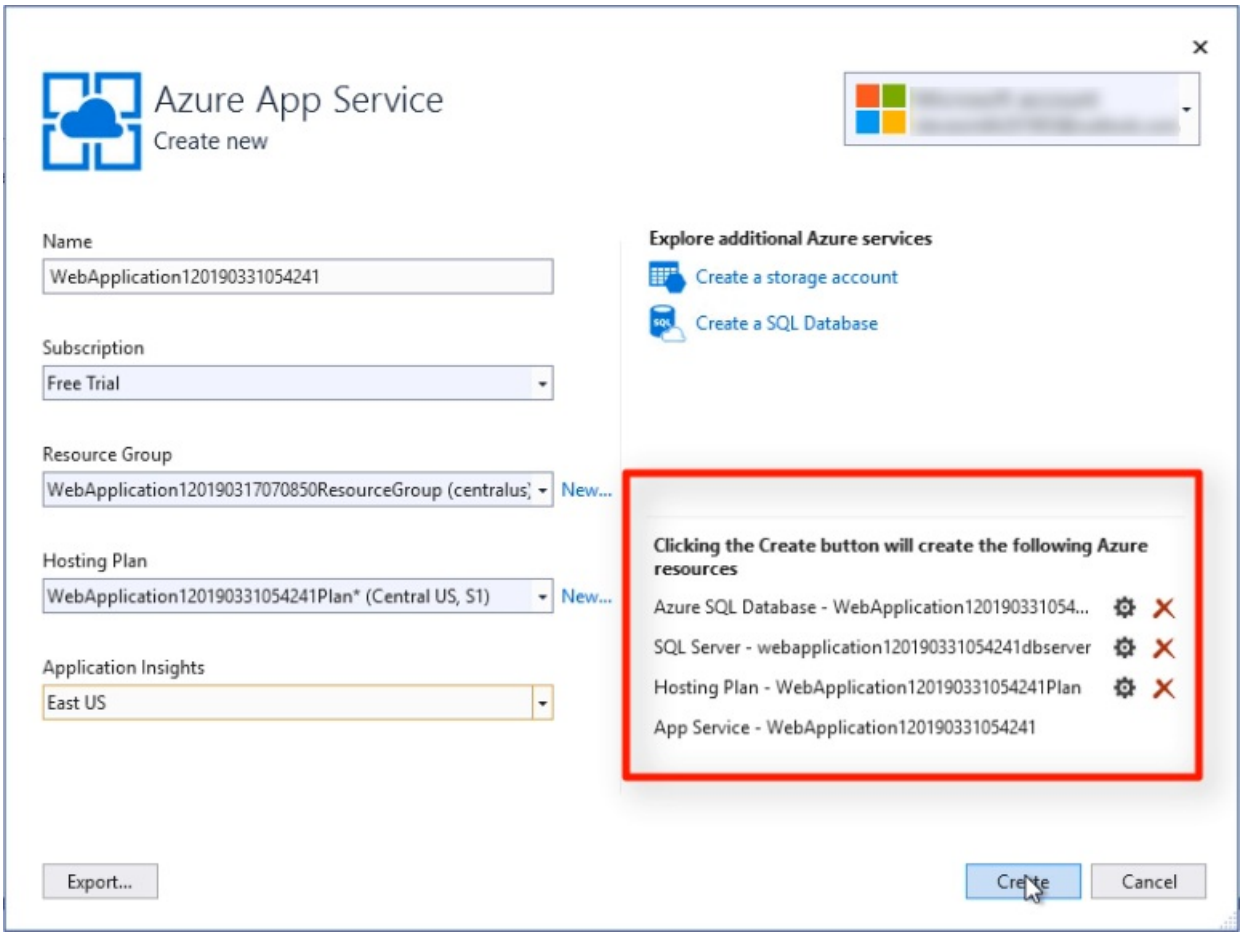
Right-click on the project in solution explorer and choose **Publish**. Leave the default settings of **App Service** and **Create New** and click the **Publish** button. If you don't already have an Azure account, click the **Create your Free Azure Account** and complete the brief registration process.

Add a SQL Server. Specify an administrator username and password.



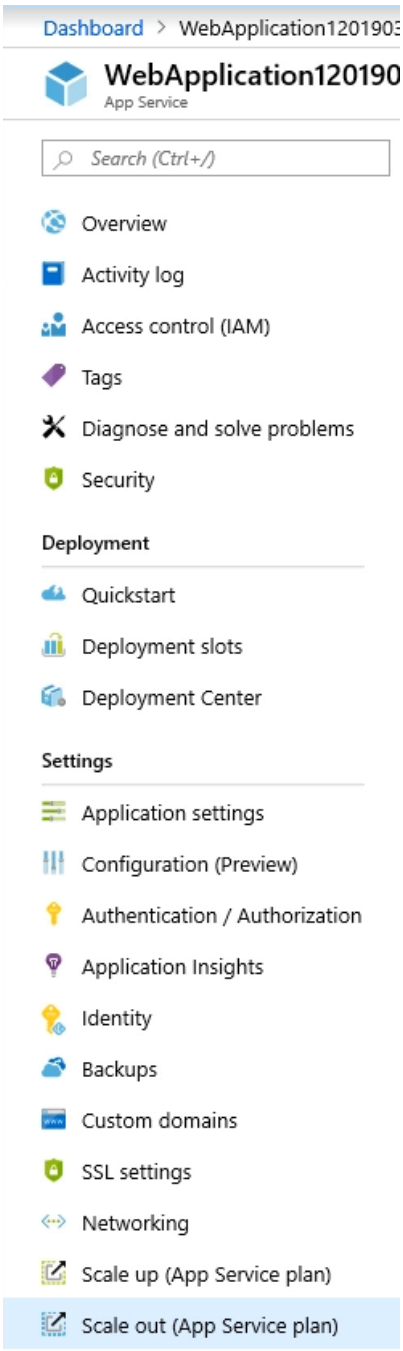
Add Application Insights.

Click the **Create** button to continue.



Exploring the Azure portal and your hosted app

Once the app service is created your site will launch in a browser. While it's loading you can also find the App Service in the Azure portal. Exploring the available options for your app service you'll find an **Overview** section where you can start and stop the app.



Scalability

You can examine the options to scale the app up as well as out. Scaling up refers to increasing the resources given to each instance hosting your app. Scaling out refers to increasing the number of instances hosting your app. You can configure autoscale for your app, which will automatically increase the number of instances used to host your app in response to load,and then reduce them once the load has decreased.

Security and compliance

Another benefit of hosting our app using Azure is security and compliance. Azure App Service provides ISO, SOC, and PCI compliance. We can choose to authenticate users with Azure Active Directory or social logins like Twitter, Facebook, Google,or Microsoft. We can create IP restrictions, manage service identities, add custom domains, and support SSL for the app, as well as configure backups with restorable archive copies of the app's content, configuration, and database. These features are accessed in the Authentication / Authorization, Identity, backups, and SSL Settings menu options.

Deployment slots

Frequently when you deploy an app, there's a small period of downtime while the app restarts. Deployment Slots avoid this issue by allowing you to deploy to a separate staging instance or set of instances and warm these up before swapping them into production. The swap is just an instant and seamless traffic redirection. If there are any

issues in production after the swap, you can always swap back to your last known good production state.

Update connection string

By default Azure expects a new app's connection to its new SQL Server database to use a connection string named `DefaultConnection`. Currently the app we created earlier in this tutorial series uses a connection string named `AppDbContext`. We need to change this in *appsettings.json* and *Startup.cs* and then redeploy the app.

Test the app running in Azure

Navigate to the */Games* path and you should be able to add a new game and see it listed. Next, navigate to the */swagger* path and you should be able to use the web API endpoints from there to confirm the app's API is working as well.

Congratulations! You've completed this video tutorial series!

Next steps

Learn more about how to architect ASP.NET Core applications with these free resources.

ASP.NET Core Application Architecture

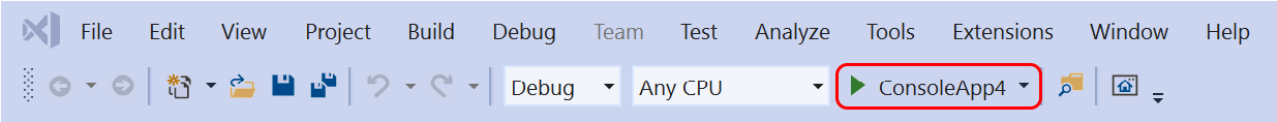
See also

- Publish an ASP.NET Core app to Azure with Visual Studio

How to: Run a C# program in Visual Studio

4/20/2020 • 6 minutes to read • Edit Online

What you need to do to run a program depends on what you're starting from, what type of program, app, or service it is, and whether you want to run it under the debugger or not. In the simplest case, when you have a project open in Visual Studio, build and run it by pressing **Ctrl+F5** (**Start without debugging**) or **F5** (**Start with debugging**), or press the green arrow (**Start Button**) on the main Visual Studio toolbar.

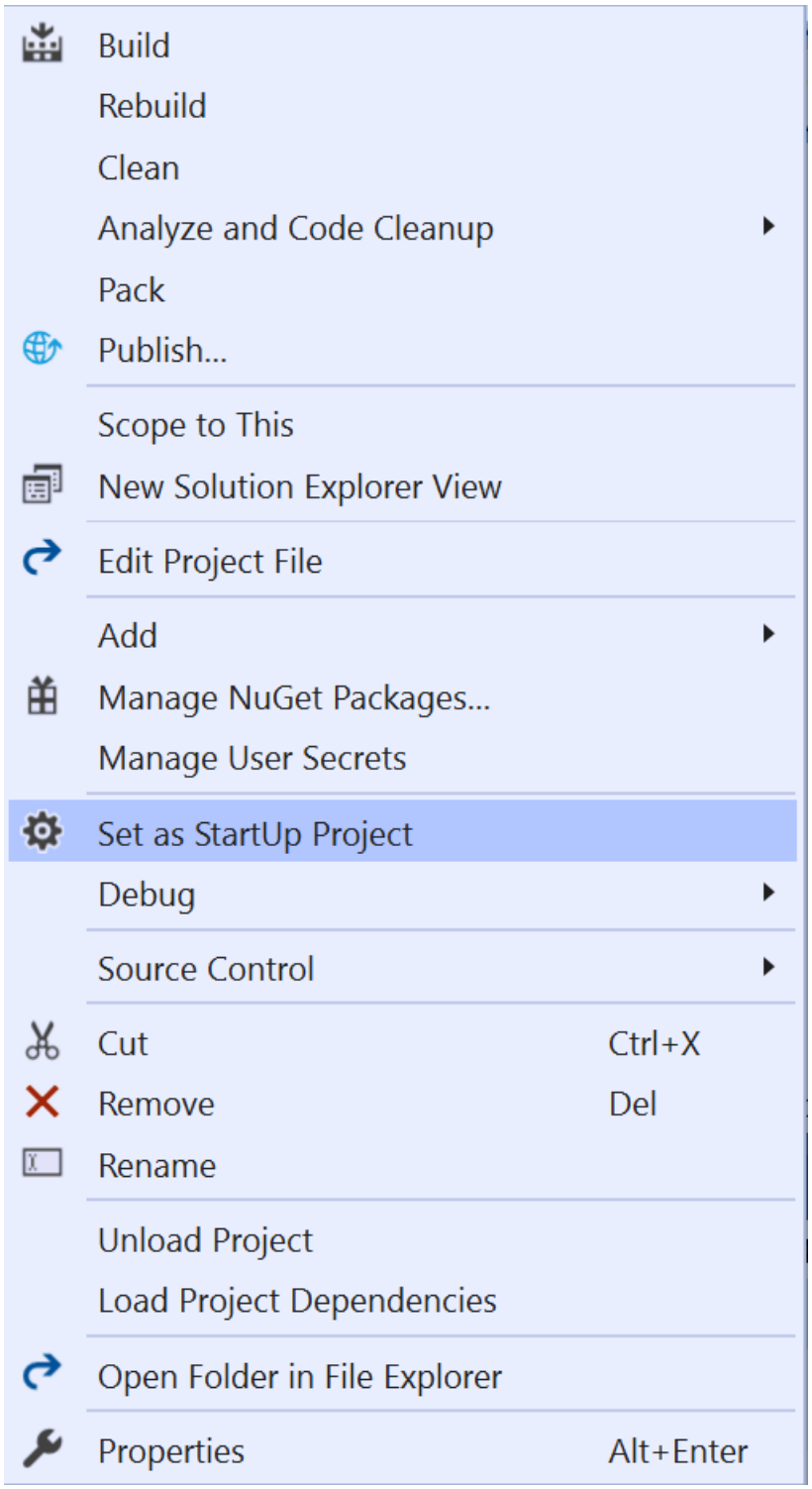


Starting from a project

If you have a C# project (*.csproj* file), then you can run it, if it is a runnable program. If a project contains a C# file with a `Main` method, and its output is an executable (EXE), then most likely it will run if it builds successfully.

If you already have the code for your program in a project in Visual Studio, open the project. To open the project, double-click or tap on the *.csproj* from the Windows File Explorer, or from Visual Studio, choose **Open a project**, browse to find the project (*.csproj*) file, and choose the project file.

After the projects loads in Visual Studio, press **Ctrl+F5** (**Start without debugging**) or use the green **Start** button on the Visual Studio toolbar to run the program. If there are multiple projects, the one with the `Main` method must be set as the startup project. To set the startup project, right-click on a project node, and choose **Set as startup project**.



Visual Studio attempts to build and run your project. If there are build errors, you see the build output in the **Output** window and the errors in the **Error List** window.

If the build succeeds, the app runs in a way that's appropriate for the type of project. Console apps run in a terminal window, Windows desktop apps start in a new window, web apps start in the browser (hosted by IIS Express), and so on.

Starting from code

If you're starting from a code listing, code file, or a small number of files, first make sure the code you want to run is from a trusted source and is a runnable program. If it has a `Main` method, it is likely intended as a runnable program that you can use the Console App template to create a project to work with it in Visual Studio.

Code listing for a single file

Start Visual Studio, open an empty C# console project, select all the code in the .cs file that's in the project already,

and delete it. Then, paste the contents of your code into the .cs file. When you paste the code, overwrite or delete the code that was there before. Rename the file to match the original code.

Code listings for a few files

Start Visual Studio, open an empty C# console project, select all the code in the .cs file that's in the project already, and delete it. Then, paste the contents of the first code file into the .cs file. Rename the file to match the original code.

For a second file, right-click on the project node in **Solution Explorer** to open the shortcut menu for the project, and choose **Add > Existing Item** (or use the key combination **Shift+Alt+A**), and select the code files.

Multiple files on disk

1. Create a new project of the appropriate type (use C# **Console App** if you're not sure).
2. Right-click on the project node, se **Add** > **Existing Item** to select the files and import them into your project.

Starting from a folder

When you're working with a folder of many files, first see if there's a project or solution. If the program was created with Visual Studio, you should find a project file or a solution file. Look for files with the *.csproj* extension or .sln extension and in the Windows File Explorer, double-click on one of them to open them in Visual Studio. See Starting from a Visual Studio solution or project.

If you don't have a project file, such as if the code was developed in another development environment, then open the top-level folder by using the **Open folder** method in Visual Studio. See Develop code without projects or solutions.

Starting from a GitHub or Azure DevOps repo

If the code you want to run is in GitHub or in an 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 the program, press the green arrow (**Start** button) on the main Visual Studio toolbar, or press **F5** or **Ctrl+F5** to run the program. When you use the **Start** button, it runs under the debugger. Visual Studio attempts to build the code in your project and run it. If that succeeds, great! But if not, continue reading for some ideas on how to get it to build successfully.

Troubleshooting

Your code might have errors, but if the code is correct, but just depends on some other assemblies or NuGet packages, or was written to target a different version of .NET, you might be able to easily fix it.

Add references

To build properly, the code must be correct and have the right references set up to libraries or other dependencies. You can look at the red squiggly lines and at the **Error List** to see if the program has any errors, even before you compile and run it. If you're seeing errors related to unresolved names, you probably need to add a reference or a using directive, or both. If the code references any assemblies or NuGet packages, you need to add those references in the project.

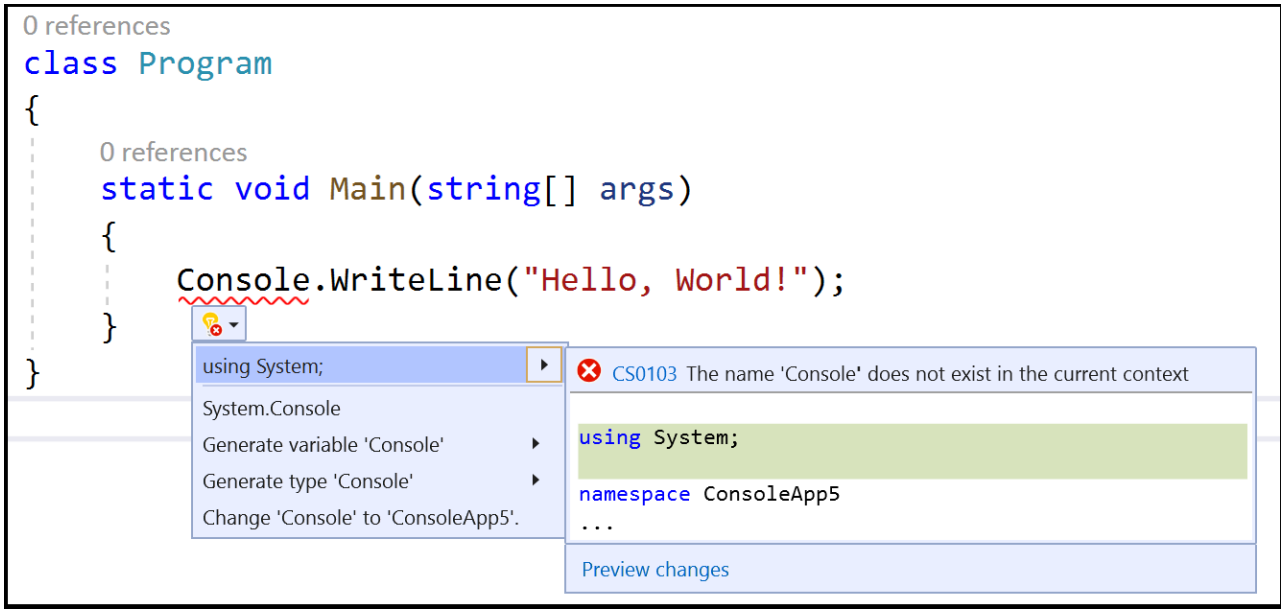
Visual Studio tries to help you identify missing references. When a name is unresolved, a light bulb icon appears in the editor. If you click the light bulb, you can see some suggestions on how to fix the issue. Fixes might be to:

- add a using directive
- add a reference to an assembly, or

- install a NuGet package.

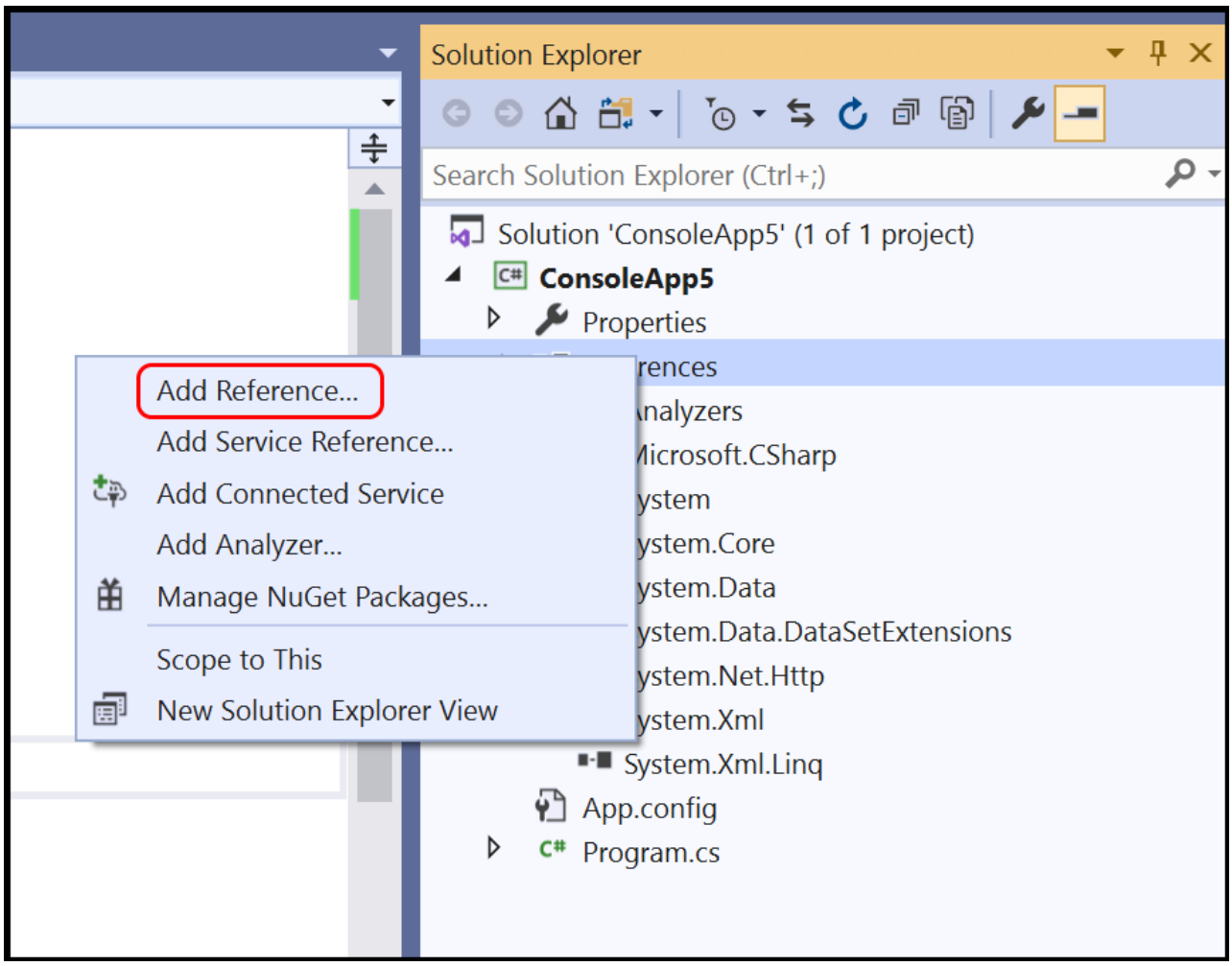
Missing using directive

For example, in the following screen, you can choose to add `using System;` to the start of the code file to resolve the unresolved name `Console`:



Missing assembly reference

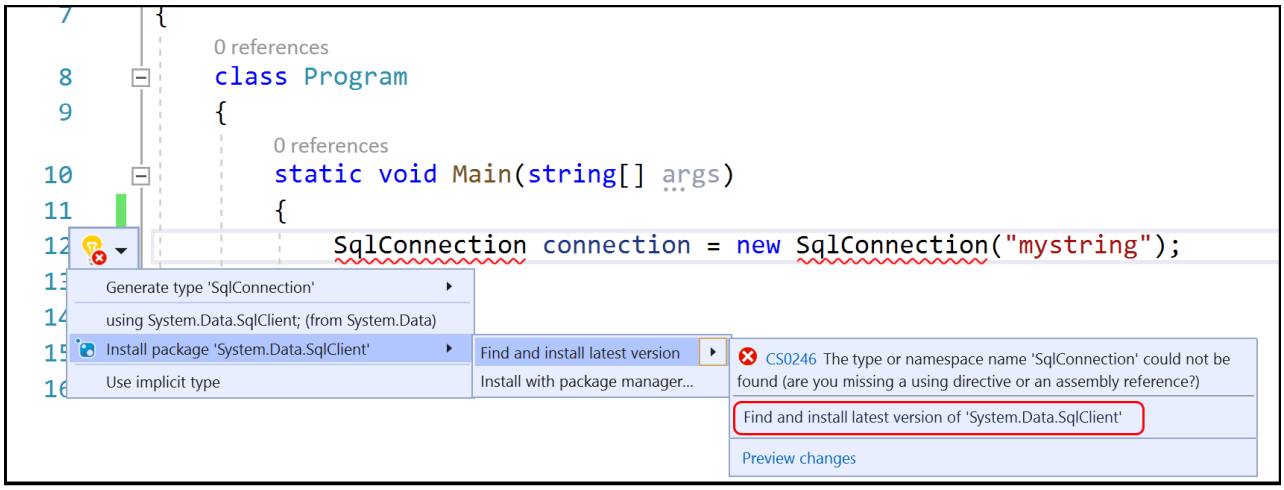
.NET references can be in the form of assemblies or NuGet packages. Usually, if you find source code, the publisher or author will explain what assemblies are required and what packages the code depends on. To add a reference to a project manually, right-click on the **References** node in the **Solution Explorer**, choose **Add Reference**, and locate the required assembly.



You can find assemblies and add references by following the instructions in Add or remove references by using the reference manager.

Missing NuGet package

If Visual Studio detects a missing NuGet package, a light bulb appears and gives you the option to install it:



If that doesn't solve the issue and Visual Studio can't locate the package, try searching for it 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 degree of backward compatibility, a newer framework might run code written for an older framework without any modifications. But, sometimes you need to target a specific framework. You might need to install a specific version of the .NET Framework or .NET Core, if it's not already installed. See Modify Visual Studio.

To change the target framework, 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.

See also

Create your first C# app

Tutorial: Open a project from a repo

8/30/2019 • 3 minutes to read • Edit Online

In this tutorial, you'll use Visual Studio to connect to a repository for the first time and then open a project from it.

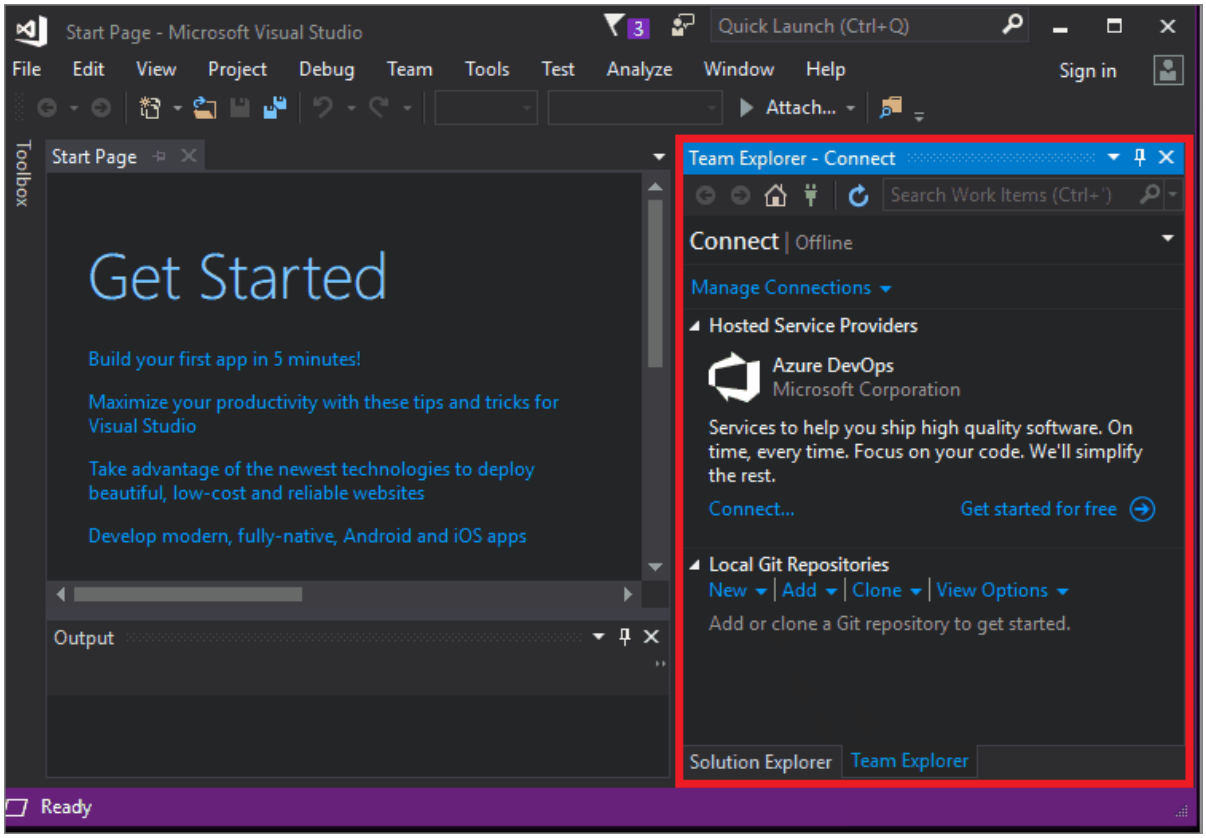
If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

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

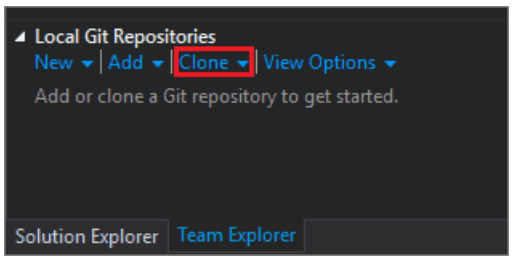
Open a project from a GitHub repo

1. Open Visual Studio 2017.
2. From the top menu bar, choose **File** > **Open** > **Open from Source Control**.

   The **Team Explorer - Connect** pane opens.



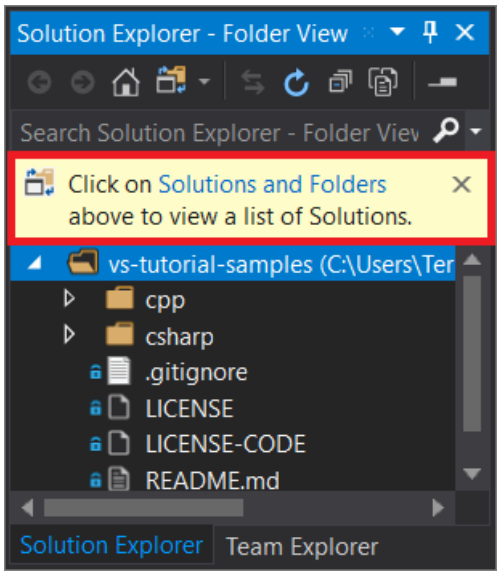
3. In the **Local Git Repositories** section, choose **Clone**.



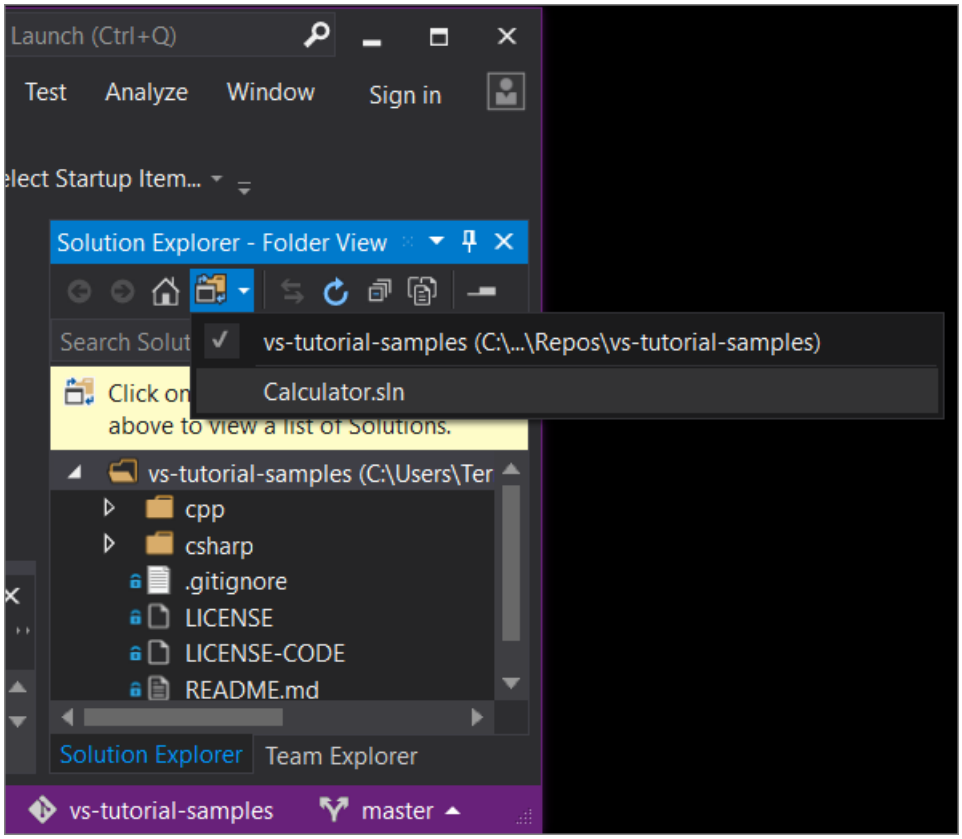
4. In the box that says ***Enter the URL of a Git repo to clone***, type or paste the URL for your repo, and then press **Enter**. (You might receive a prompt to sign in to GitHub; if so, do so.)

   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**.



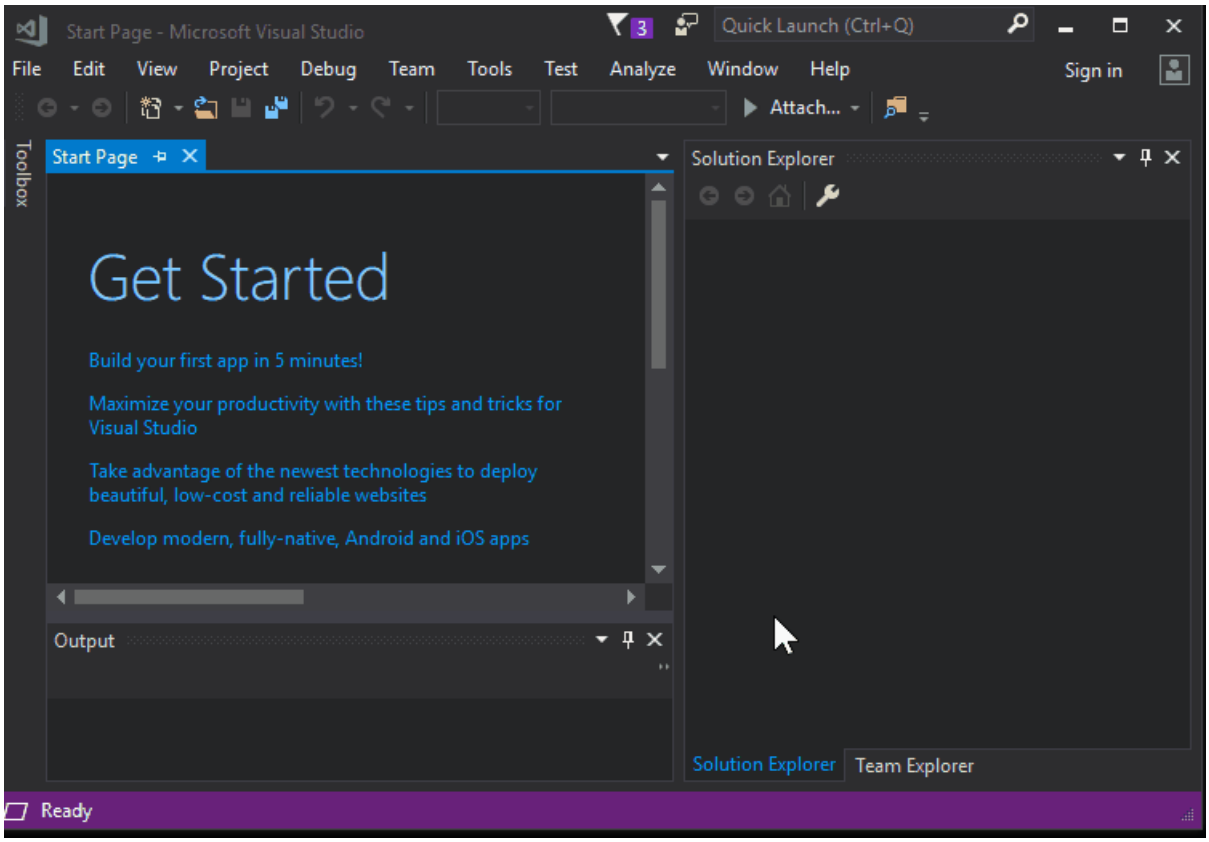
5. If you have a solution file available, it will appear in the "Solutions and Folders" fly-out menu. Choose it, and Visual Studio opens your solution.



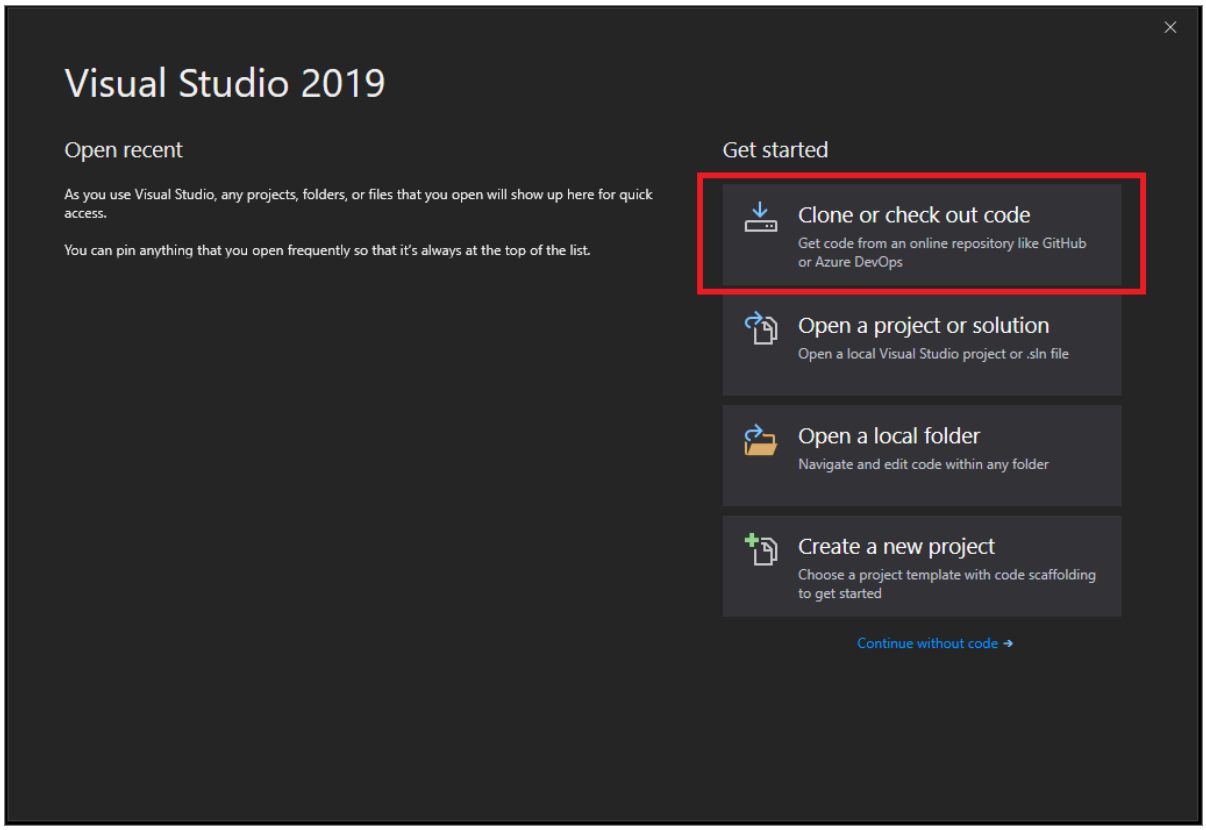
If you do not have a solution file (specifically, a .sln file) in your repo, the fly-out menu will say "No Solutions Found." However, you can double-click any file from the folder menu to open it in the Visual Studio code editor.

Review your work

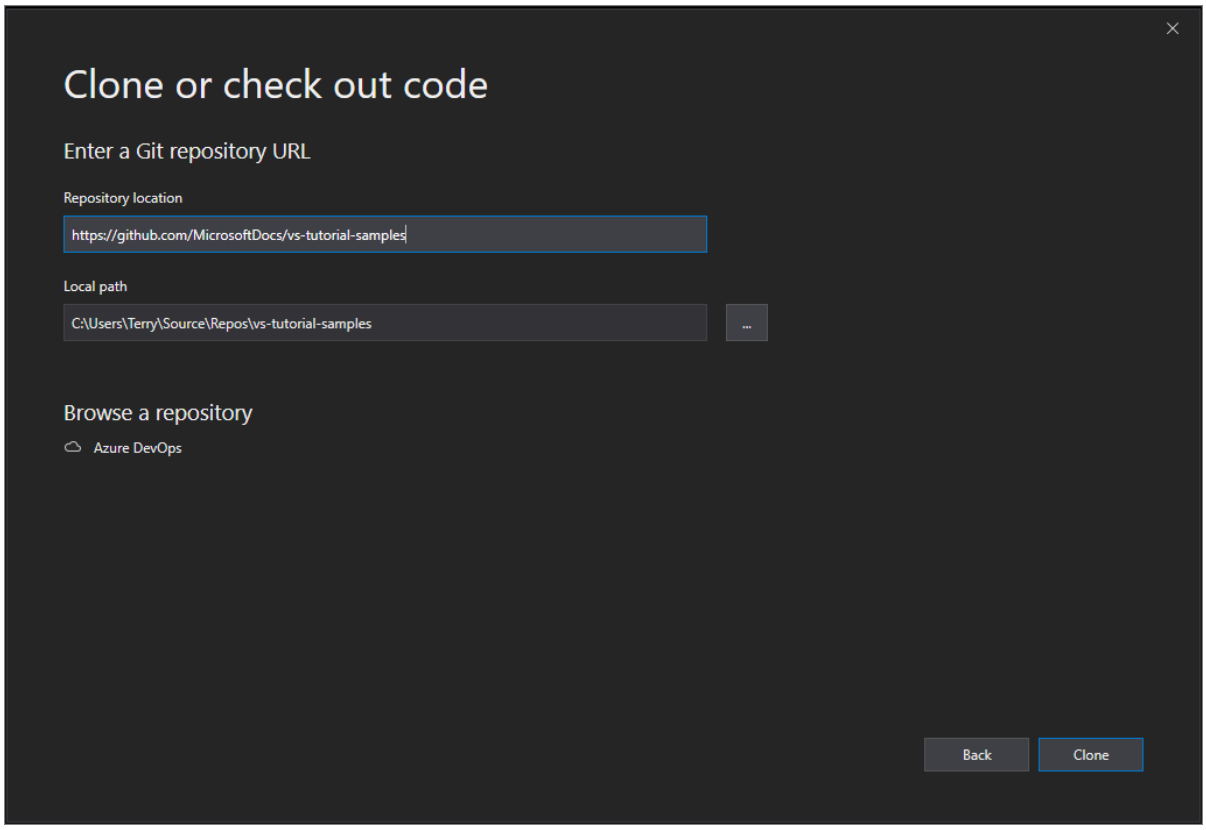
View the following animation to check the work that you completed in the previous section.



1. Open Visual Studio 2019.
2. On the start window, choose **Clone or check out code**.

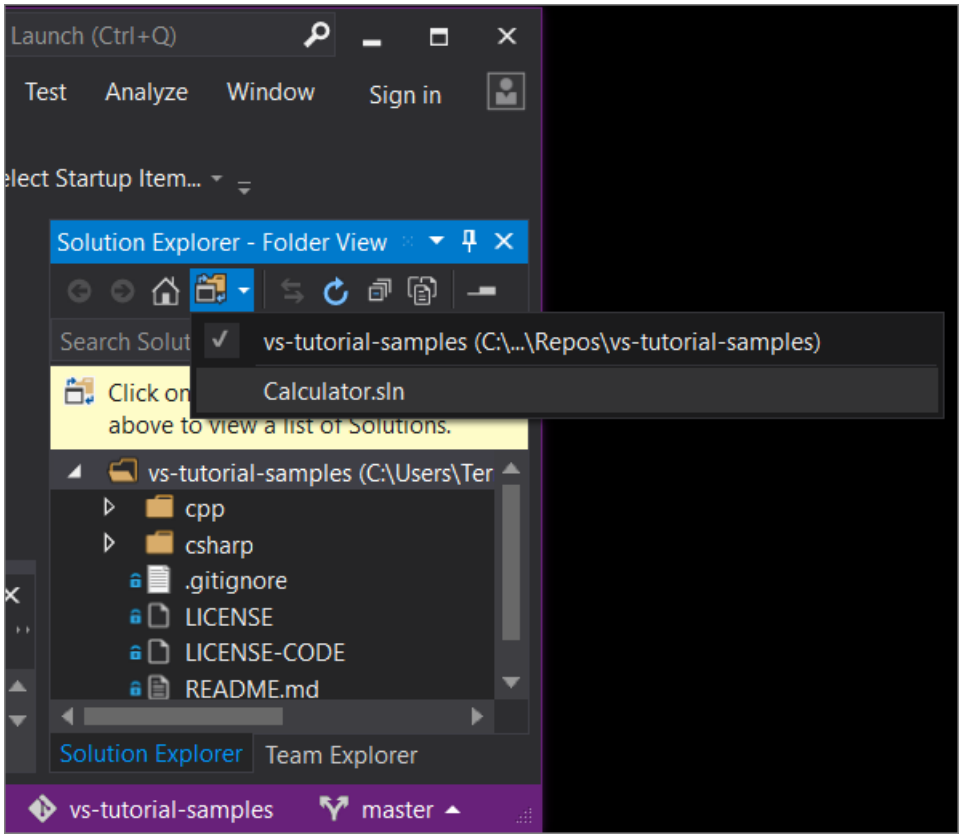


3. Enter or type the repository location, and then choose **Clone**.



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. Choose it, and Visual Studio opens your solution.

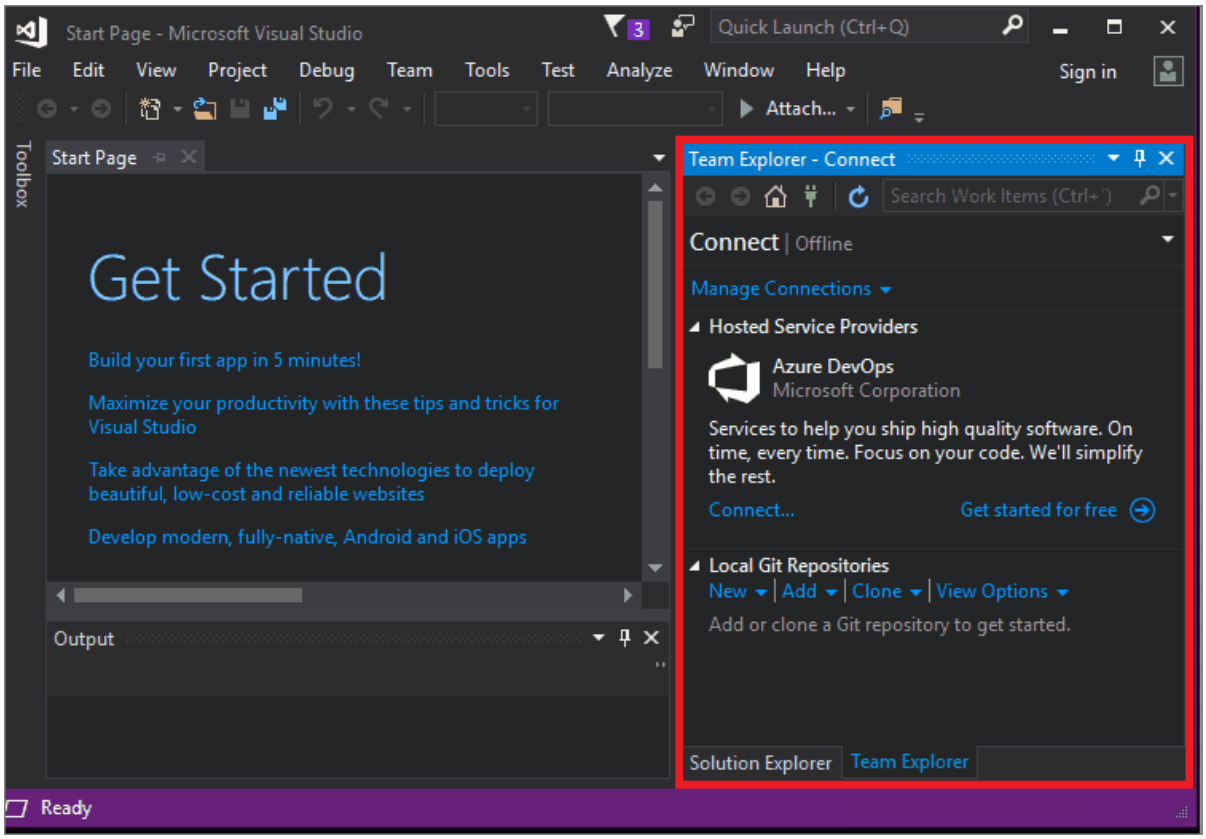


If you do not have a solution file (specifically, a .sln file) in your repo, the fly-out menu will say "No Solutions Found." However, you can double-click any file from the folder menu to open it in the Visual Studio code editor.

Open a project from an Azure DevOps repo

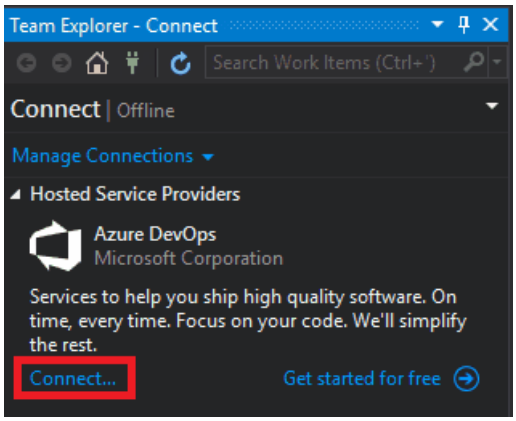
1. Open Visual Studio 2017.
2. From the top menu bar, choose **File** > **Open** > **Open from Source Control**.

   The **Team Explorer - Connect** pane opens.

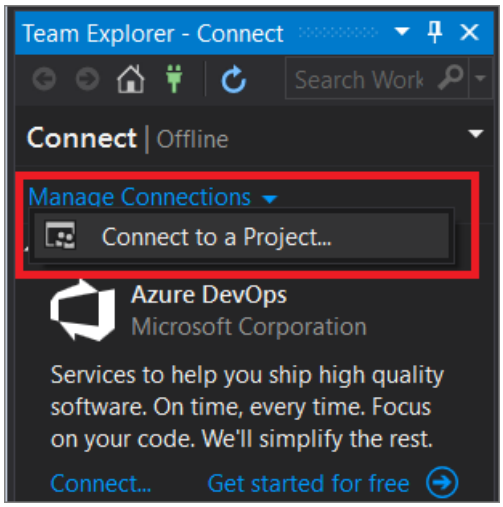


3. Here are two ways to connect to your Azure DevOps repo:

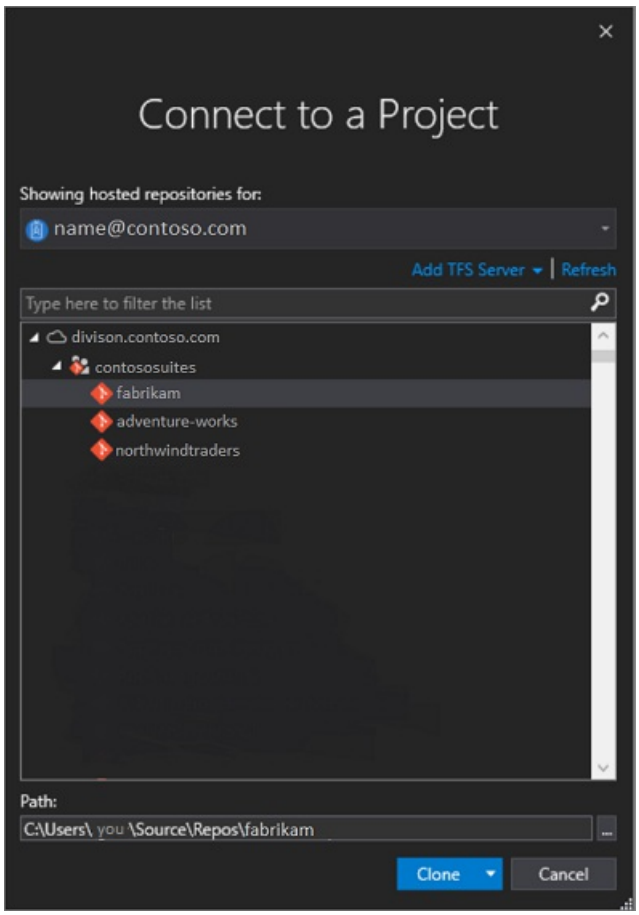
   - In the **Hosted Service Providers** section, choose **Connect...**.



   - In the **Manage Connections** drop-down list, choose **Connect to a Project...**.

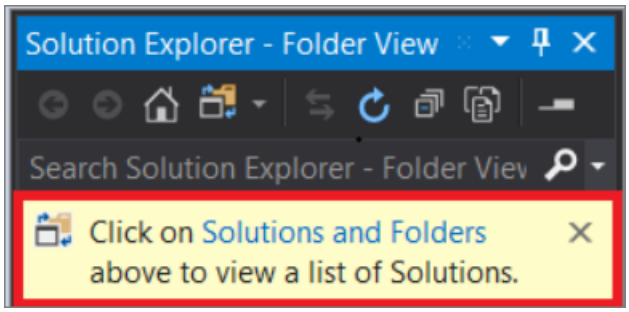


4. In the **Connect to a Project** dialog box, choose the repo that you want to connect to, and then choose **Clone**.



> **NOTE**
>
> What you see in the list box depends on the Azure DevOps repositories that you have access to.

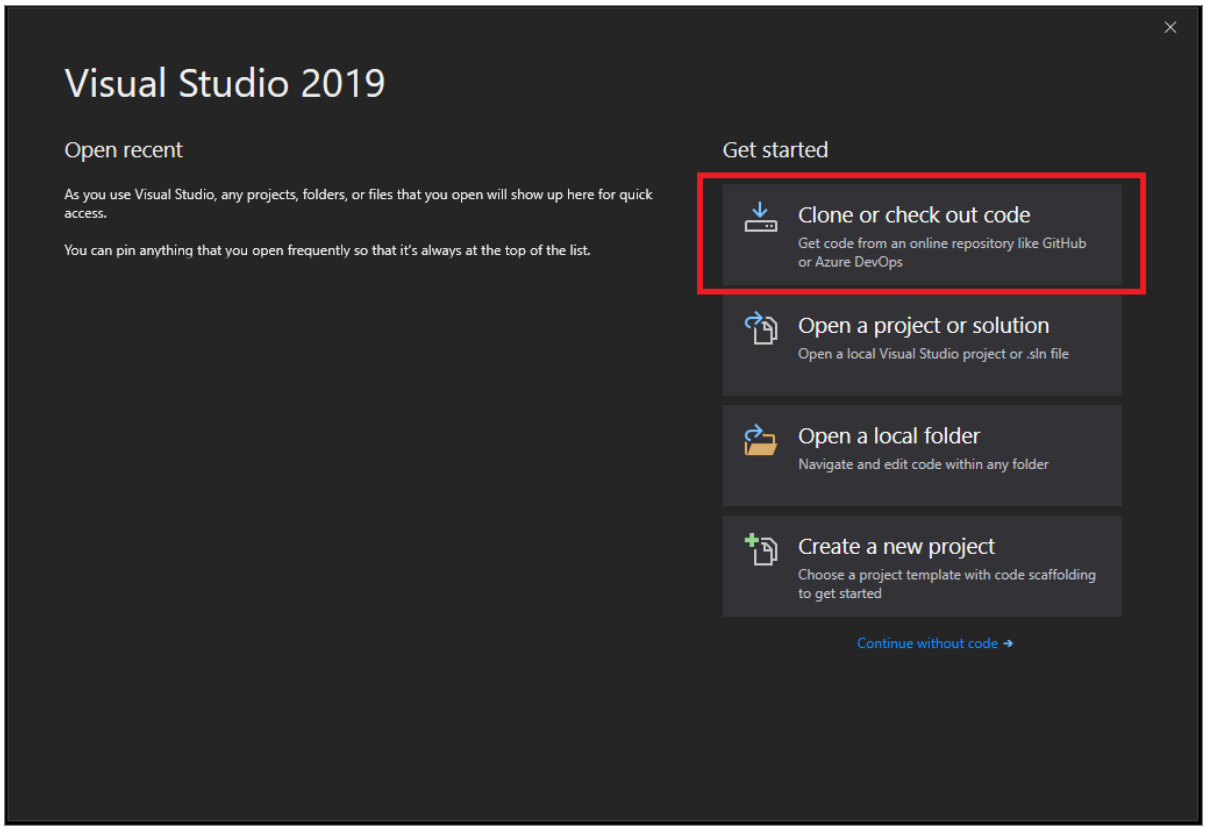
5. After Visual Studio clones your repo, Team Explorer closes and Solution Explorer opens. A message appears that says *Click on Solutions and Folders above to view a list of Solutions*. Choose **Solutions and Folders**.



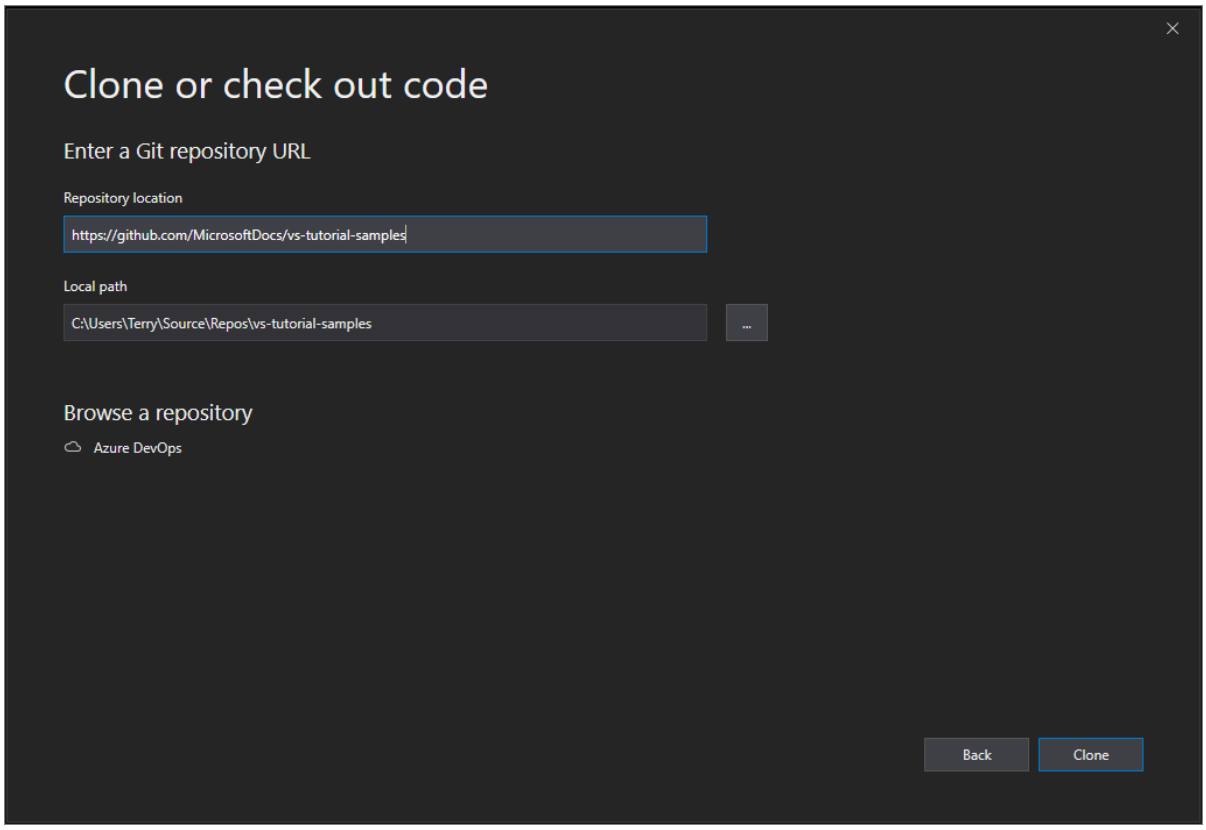
A solution file (specifically, a .sln file), will appear in the "Solutions and Folders" fly-out menu. Choose it, and Visual Studio opens your solution.

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.

1. Open Visual Studio 2019.
2. On the start window, choose **Clone or check out code**.

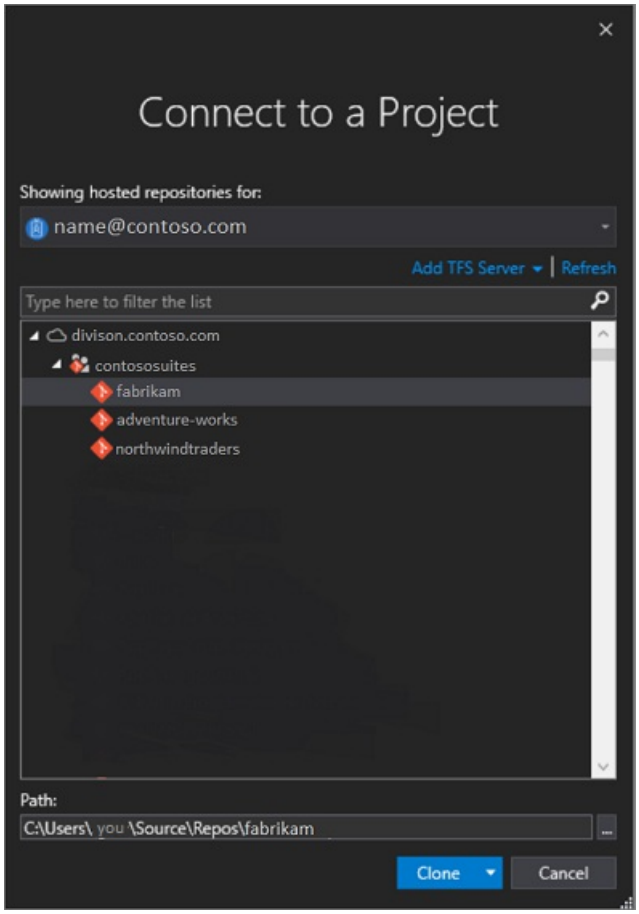


3. In the **Browse a repository** section, choose **Azure DevOps**.



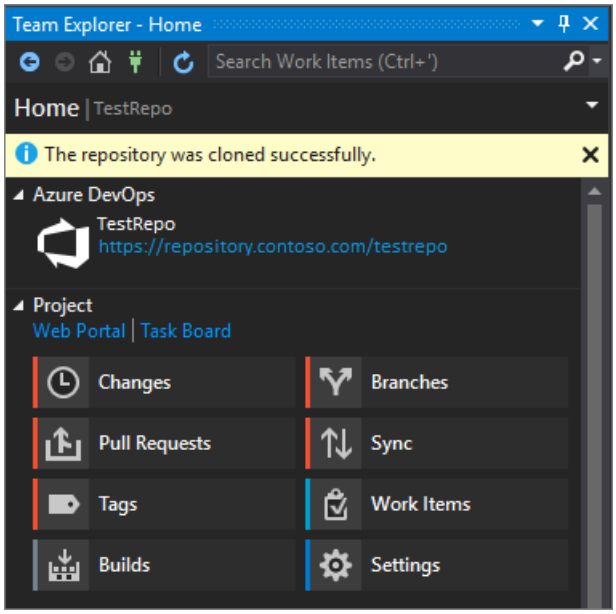
If you see a sign-in window, sign in to your account.

4. In the **Connect to a Project** dialog box, choose the repo that you want to connect to, and then choose **Clone**.

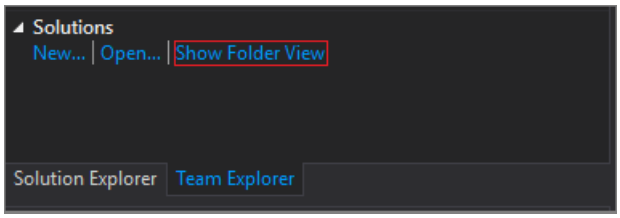


> **NOTE**
> What you see in the list box depends on the Azure DevOps repositories that you have access to.

Visual Studio opens **Team Explorer** and a notification appears when the clone is complete.

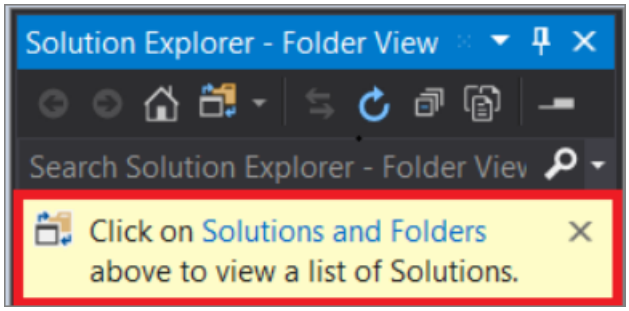


5. To view your folders and files, choose the **Show Folder View** link.



Visual Studio opens **Solution Explorer**.

6. Choose the **Solutions and Folders** link to search for a solution file (specifically, a .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.

Next steps

If you're ready to code with Visual Studio, 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

- Azure DevOps Services: Get started with Azure Repos and Visual Studio
- Microsoft Learn: Get started with Azure DevOps

Learn to use the code editor

1/1/2020 • 6 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 code easier.

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

> **TIP**
> 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.

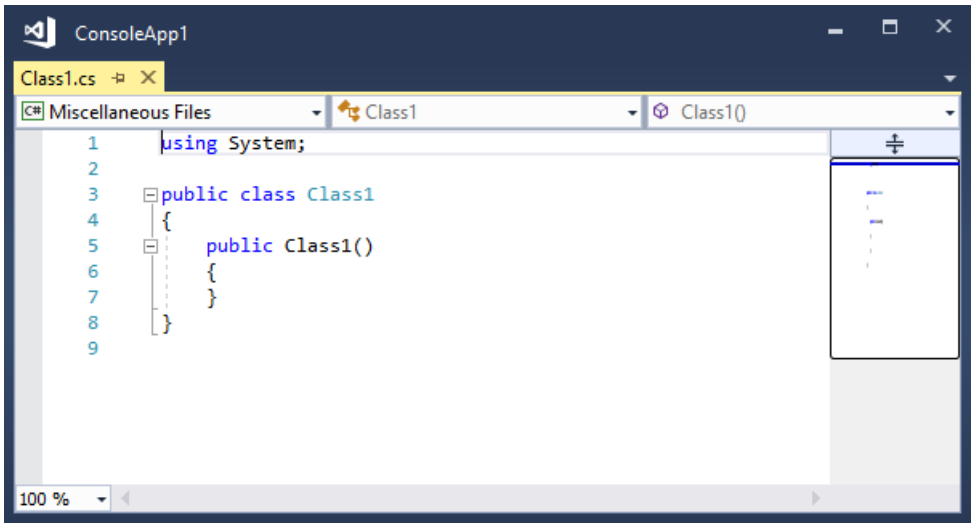
1. Open Visual Studio.

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!)

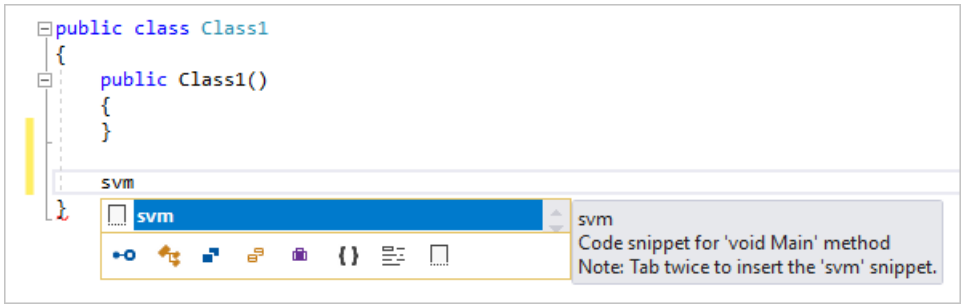


Use code snippets

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).

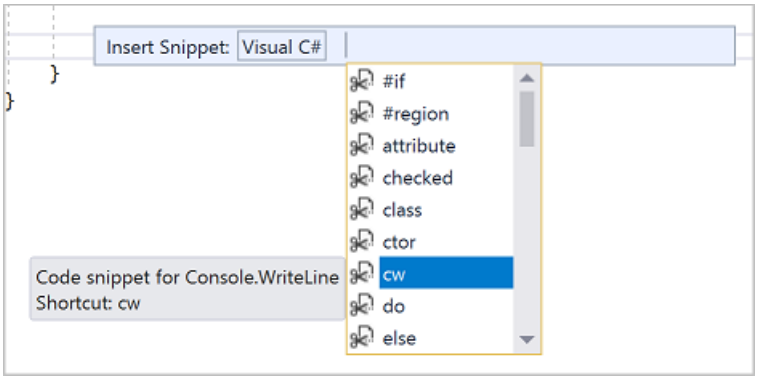
   A pop-up dialog box appears with information about the `svm` code snippet.



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:



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

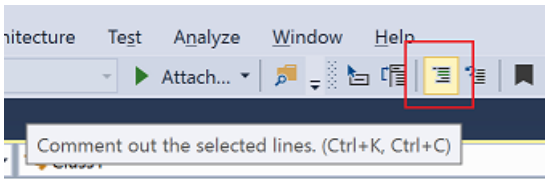
Comment out code

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;
```

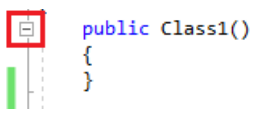
2. We're not using the `morewords` variable, but we may use it later so we don't want to completely delete it. Instead, let's comment out those lines. Select the entire definition of `morewords` to the closing semi-colon, and then choose the **Comment out the selected lines** button on the toolbar. If 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.

Collapse code blocks

We don't want to see the empty constructor for `Class1` that was generated, 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're a keyboard user, place the cursor anywhere in the constructor code and press **Ctrl**+**M**, **Ctrl**+**M**.



```
public Class1()
{
}
```

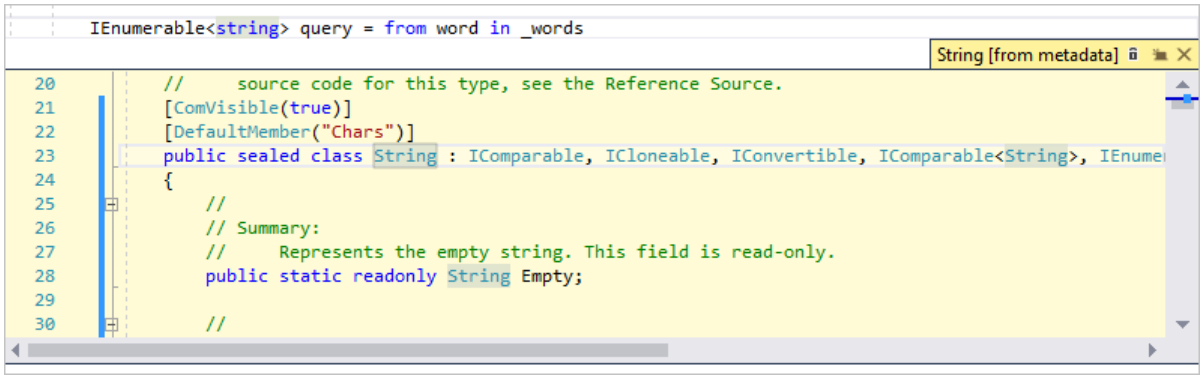
The code block collapses to just the first line, followed by an ellipsis (`...`). To expand the code block again, click the same gray box that now has a plus sign in it, or press **Ctrl**+**M**, **Ctrl**+**M** again. This feature is called Outlining and is especially useful when you're collapsing long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

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.

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);
}
```

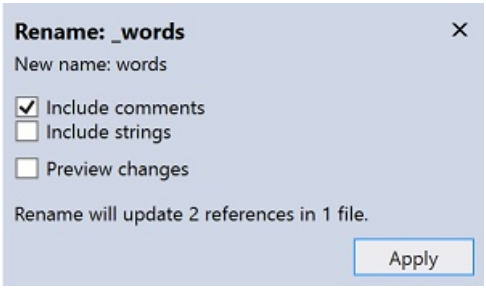
Refactor a name

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.

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

1/18/2020 • 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 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 aspects 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

4/25/2020 • 11 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 may 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 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**.

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**.

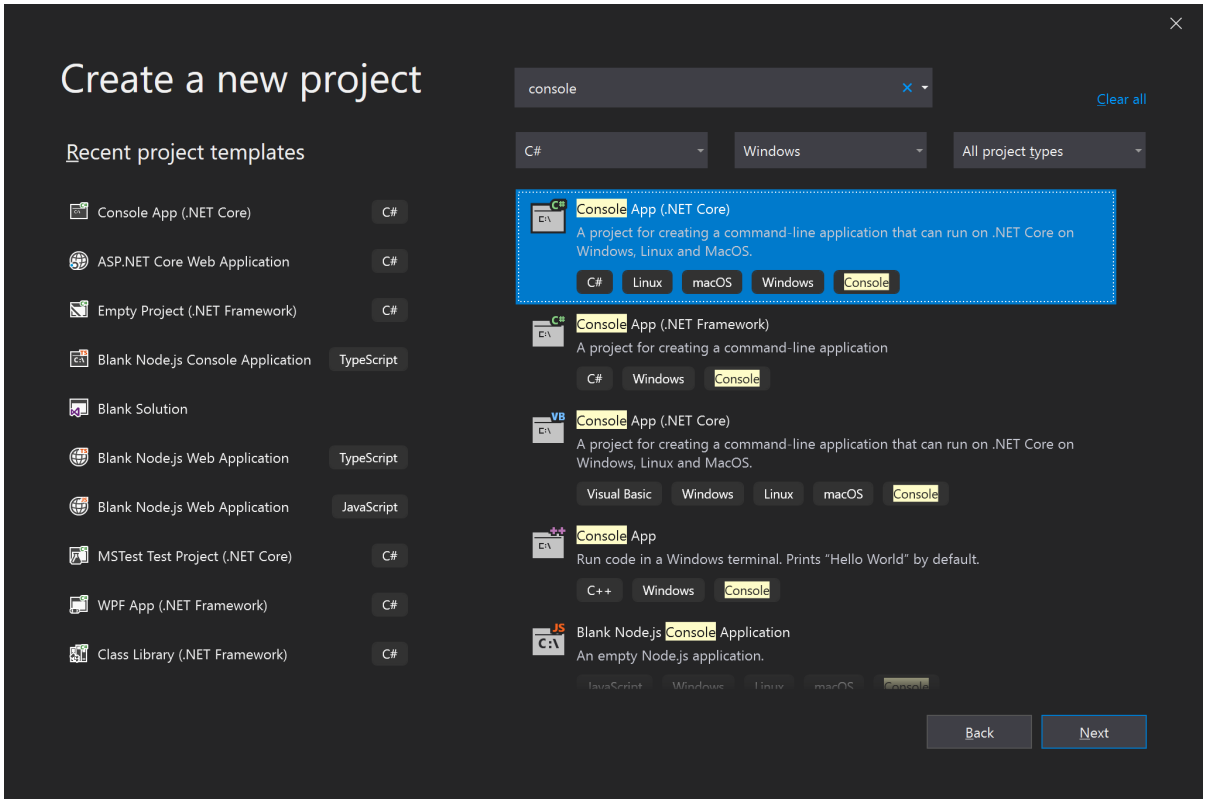
1. Open Visual Studio 2019.

   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 (.NET Core)** template, and then choose **Next**.



> **NOTE**
>
> If you do not see the **Console App (.NET Core)** 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 **Create**.

   Visual Studio opens your new project.

Create the application

1. In *Program.cs*, replace all of the default code with the following code instead:

```
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 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
   ```

   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.

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:

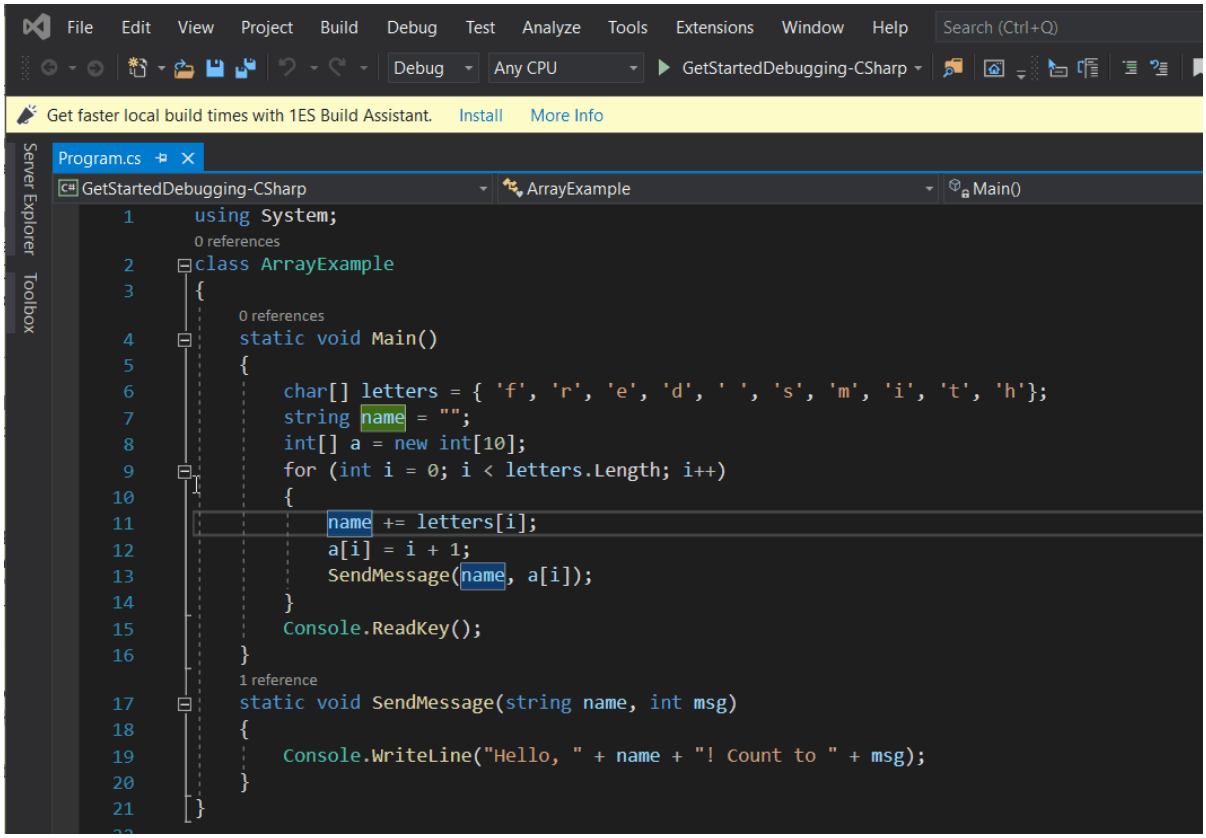
   ```
   name += letters[i];
   ```

   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.

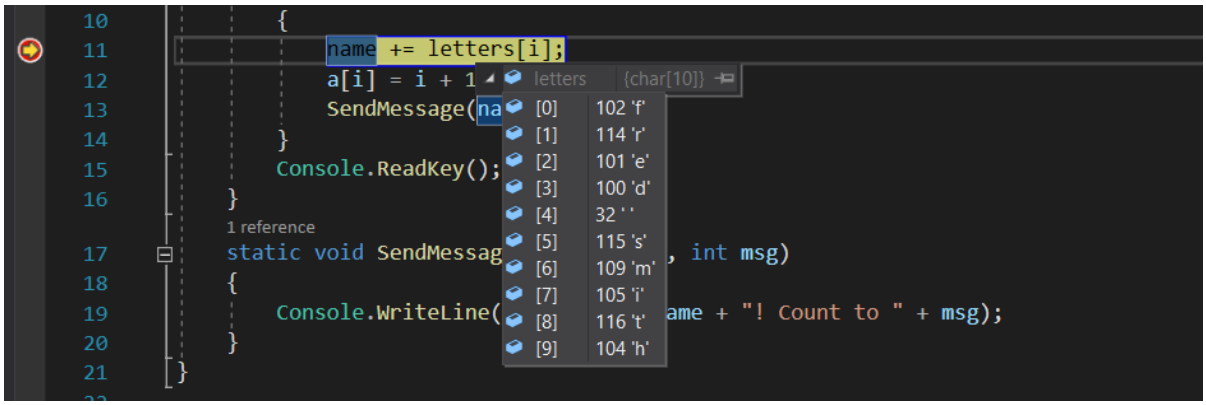
Navigate code and inspect data using data tips

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.

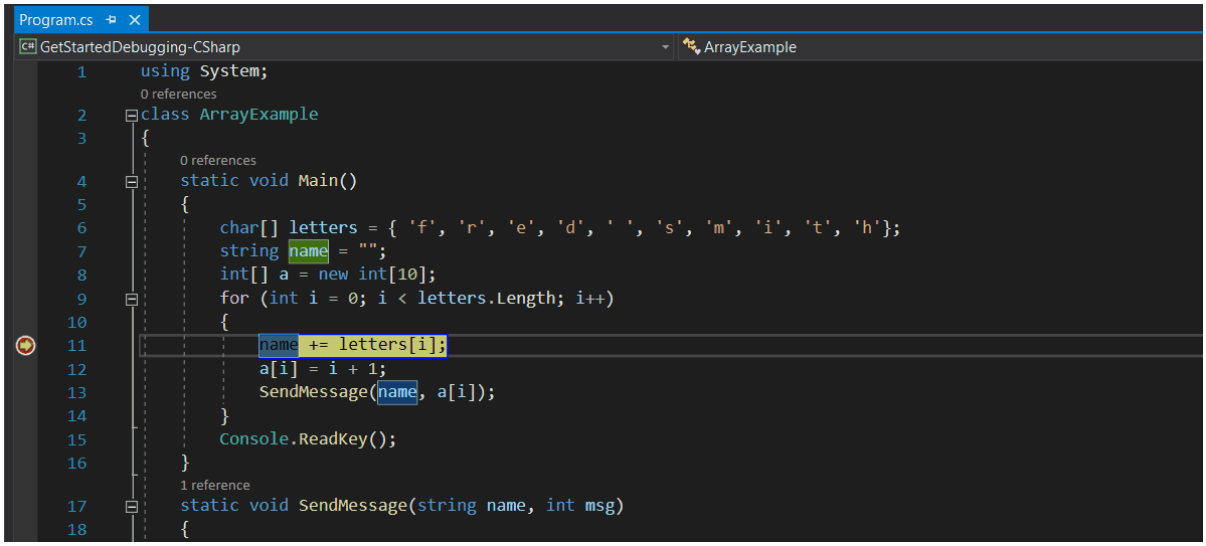


3. Next, hover over the `name` variable, and you see its current value, an empty string.

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.

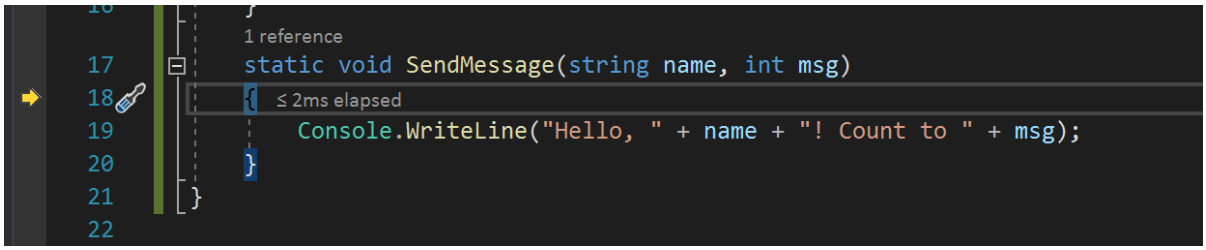
6. While still paused in the `for` loop in the `Main` method, press **F11** (or choose **Debug** > **Step Into**) until you pause at the `SendMessage` method call.

   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.

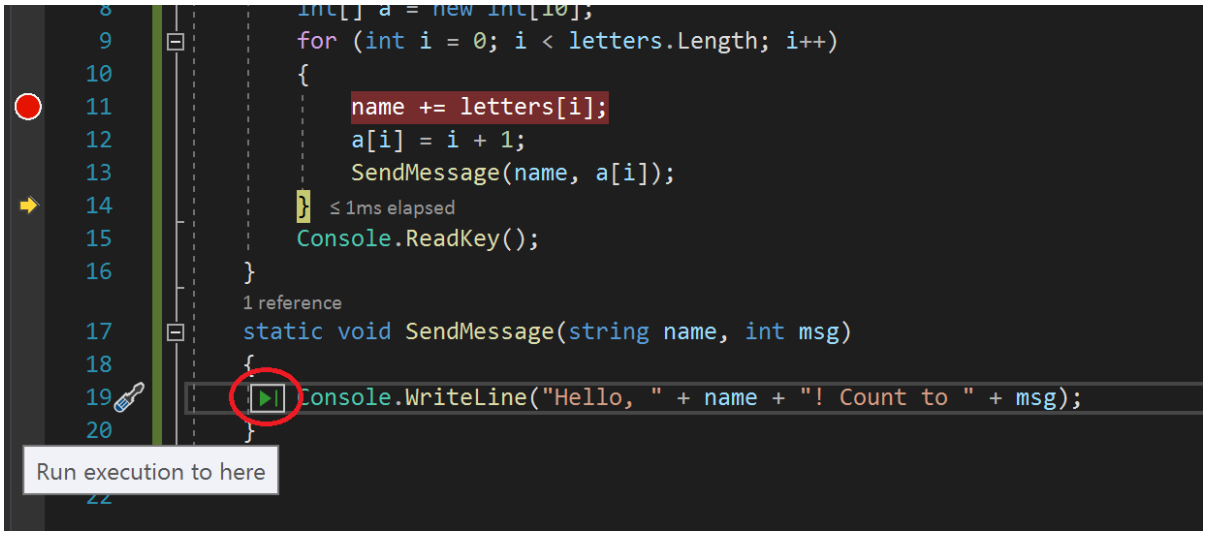
8. Press **Shift** + **F11** (or **Debug** > **Step Out**).

   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.

Navigate code using Run to Click

1. Press **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).

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.

Inspect variables with the Autos and Locals windows

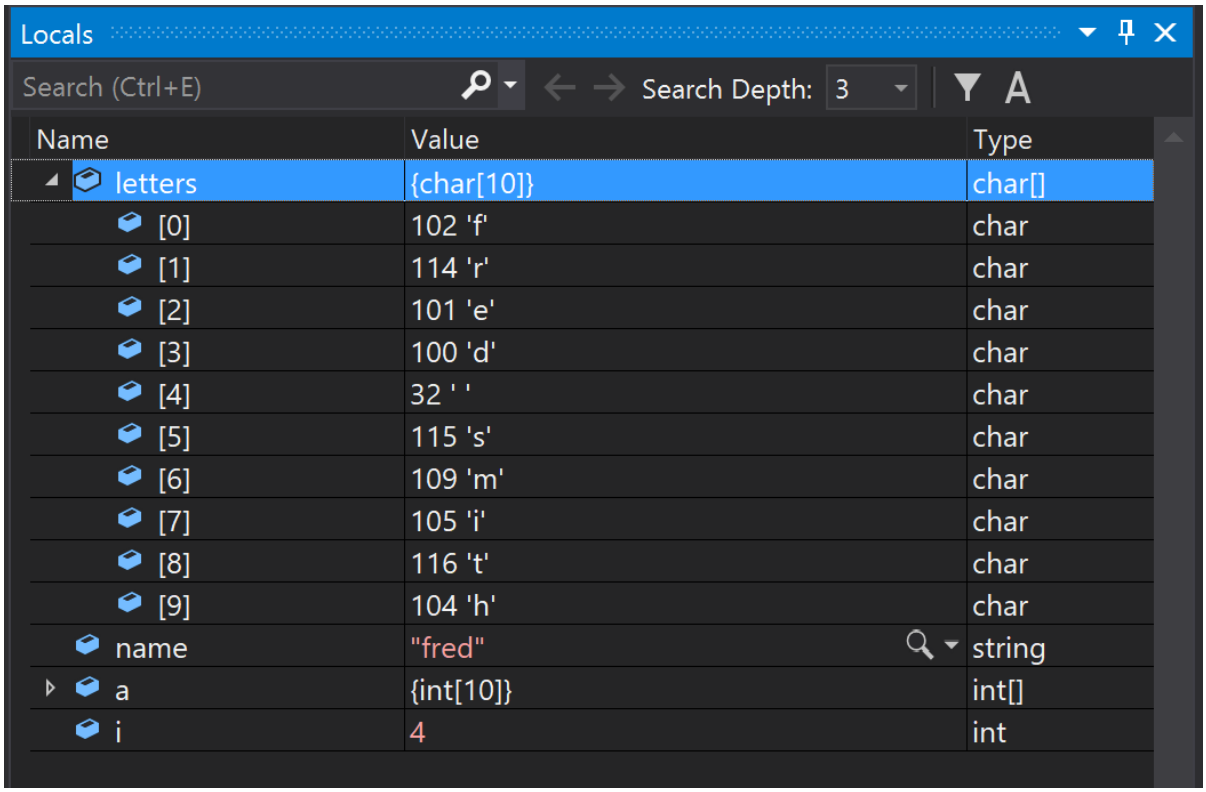
1. Look at the **Autos** window at the bottom of the code editor.

   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.

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.

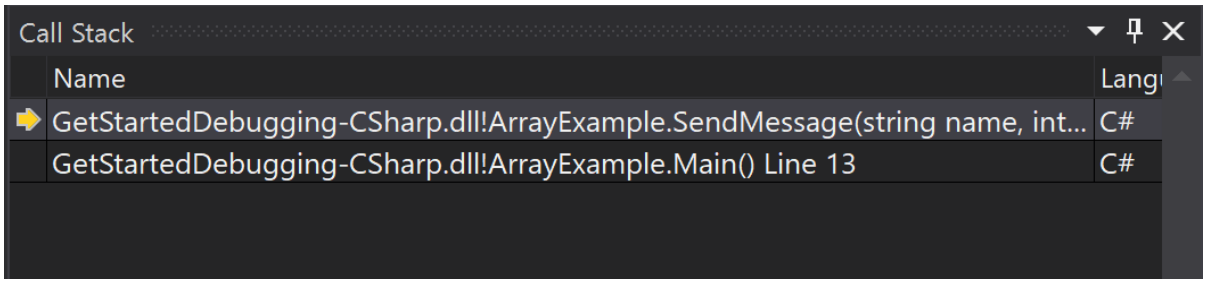
   Now, you have a watch set on the `name` variable, and you can see its value change as you move through the debugger. Unlike the other variable windows, the **Watch** window always shows the variables that you are watching (they're grayed out when out of scope).

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.

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!

Next steps

In this tutorial, you've learned how to start the debugger, step through code, and inspect variables. You may 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

4/25/2020 • 4 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.

Create unit tests

This section describes how to create a unit test project.

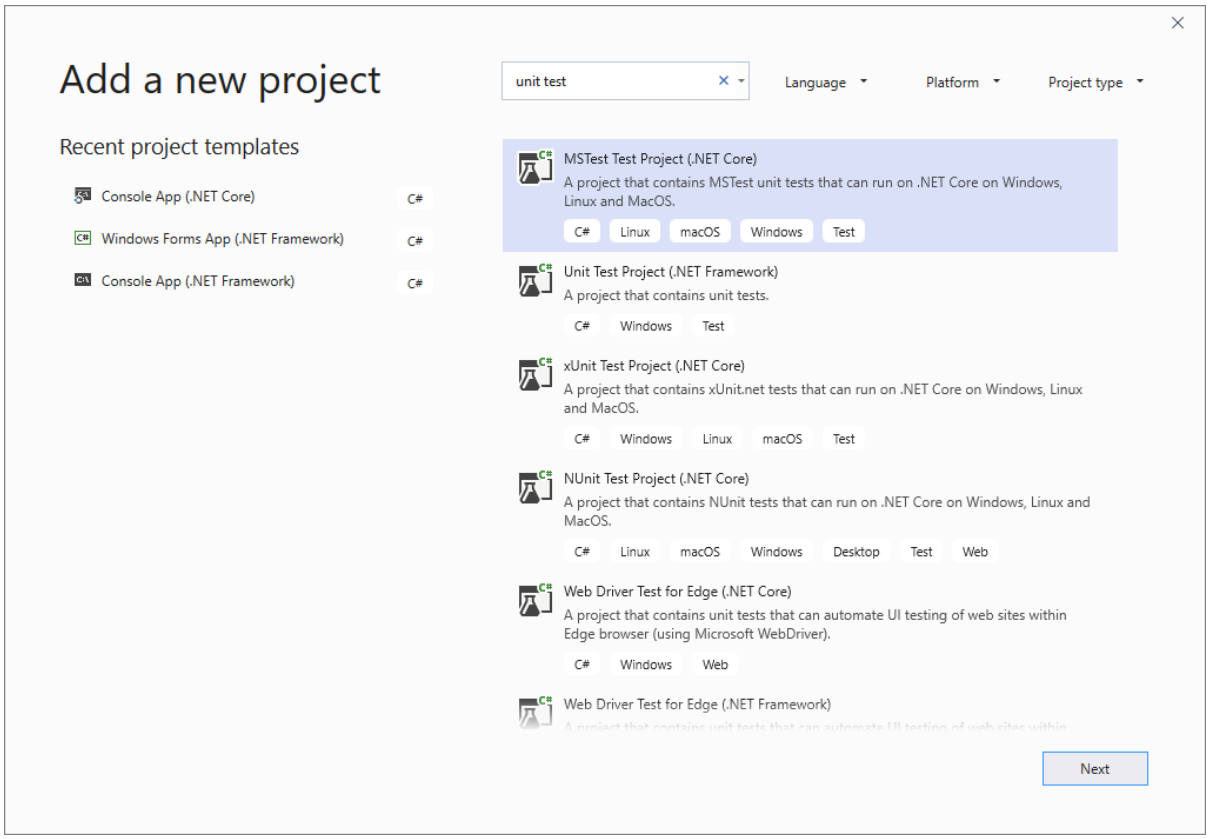
1. Open the project that you want to test in Visual Studio.

   For the purposes of demonstrating an example unit test, this article tests a simple "Hello World" project named **HelloWorldCore**. The sample code for such a project is as follows:

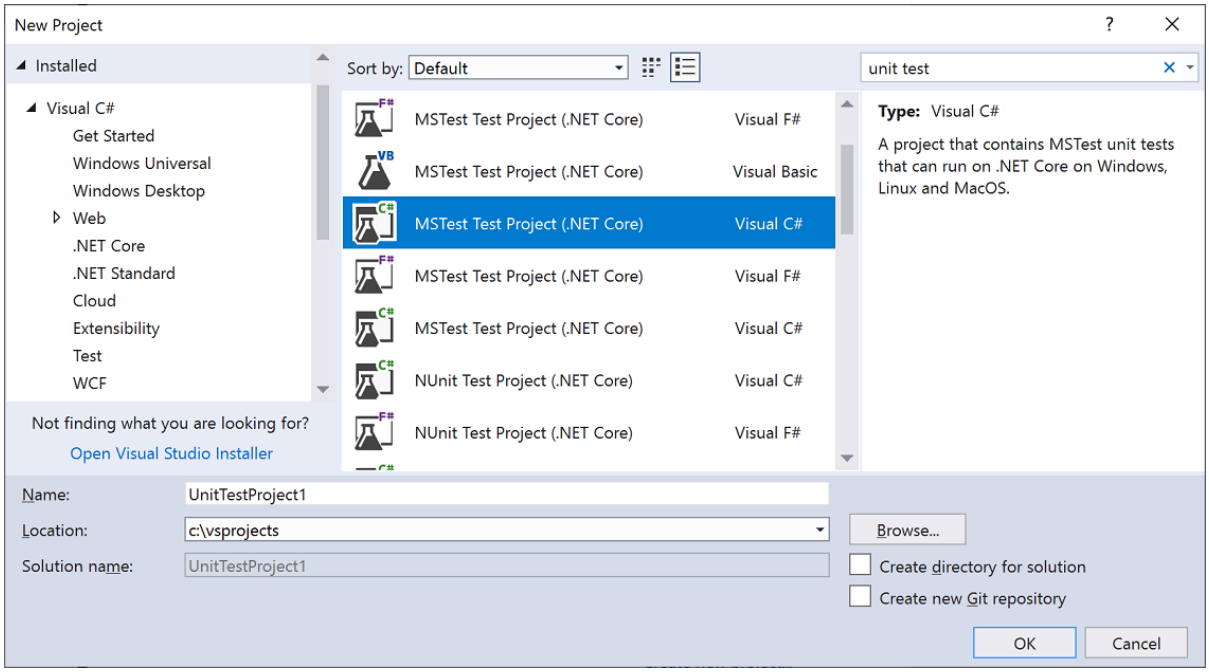
   ```
   namespace HelloWorldCore

      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 a unit test project template for the test framework you want to use and select it.

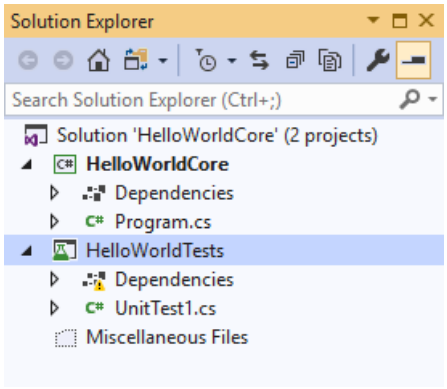


Click **Next**, choose a name for the test project, and then click **Create**.



Choose a name for the test project, 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**.

5. Select the project that contains the code you'll test and click **OK**.

6. Add code to the unit test method.

   For example, for an MSTest project, you might use the following code.

```
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);
         }
      }
   }
}
```

Or, for an NUnit project, you might use the following code.

```
using NUnit.Framework;
using System.IO;
using System;

namespace HelloWorldTests
{
   public class Tests
   {
      private const string Expected = "Hello World!";

      [SetUp]
      public void Setup()
      {
      }
      [Test]
      public void TestMethod1()
      {
         using (var sw = new StringWriter())
         {
            Console.SetOut(sw);
            HelloWorldCore.Program.Main();

            var result = sw.ToString().Trim();
            Assert.AreEqual(Expected, result);
         }
      }
   }
}
```

TIP

For more details about creating unit tests, see Create and run unit tests for managed code.

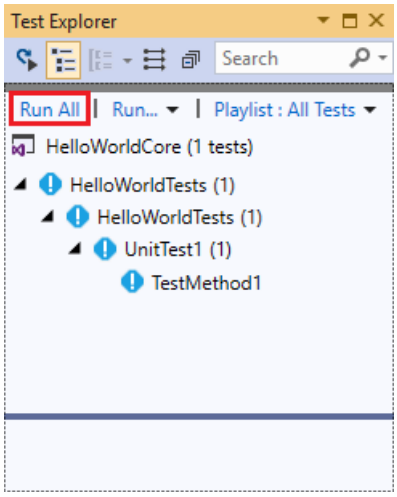
Run unit tests

1. Open Test Explorer.

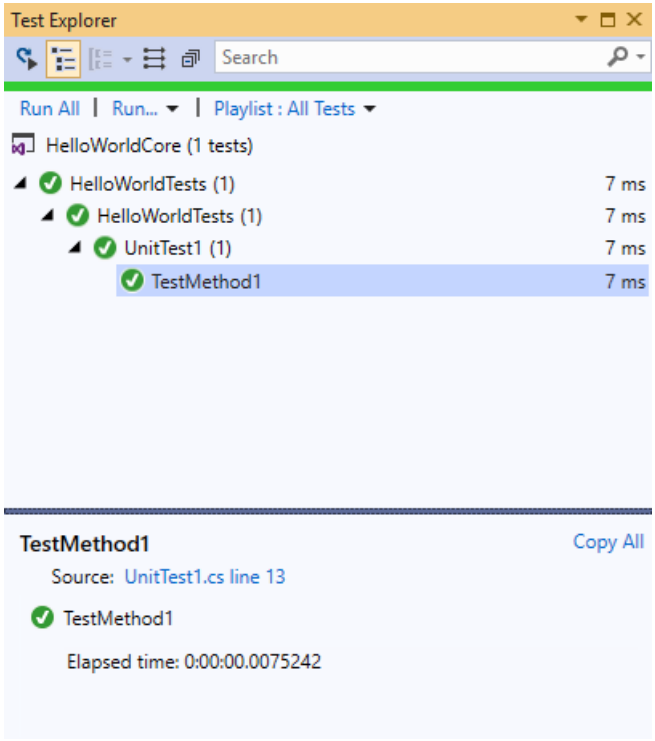
   To open Test Explorer, choose **Test** > **Test Explorer** from the top menu bar.

   To open Test Explorer, choose **Test** > **Windows** > **Test Explorer** from the top menu bar.

2. Run your unit tests by clicking **Run All**.



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.

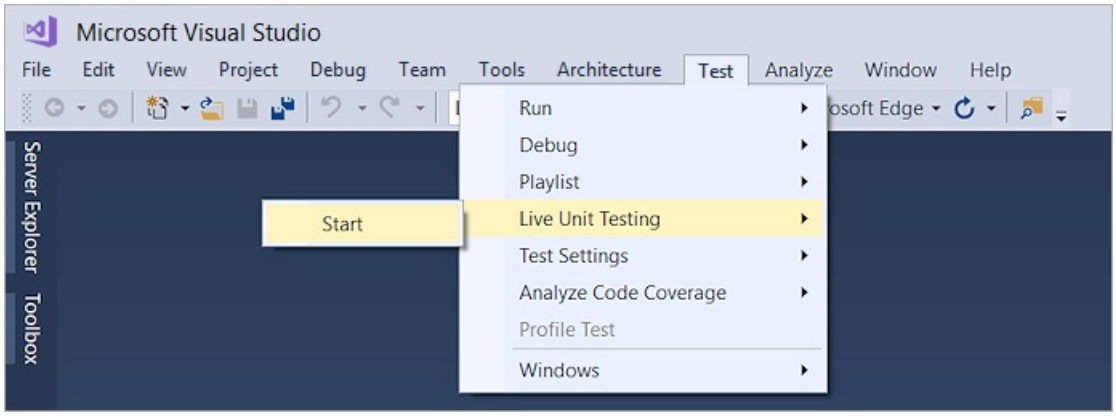
View live unit test results

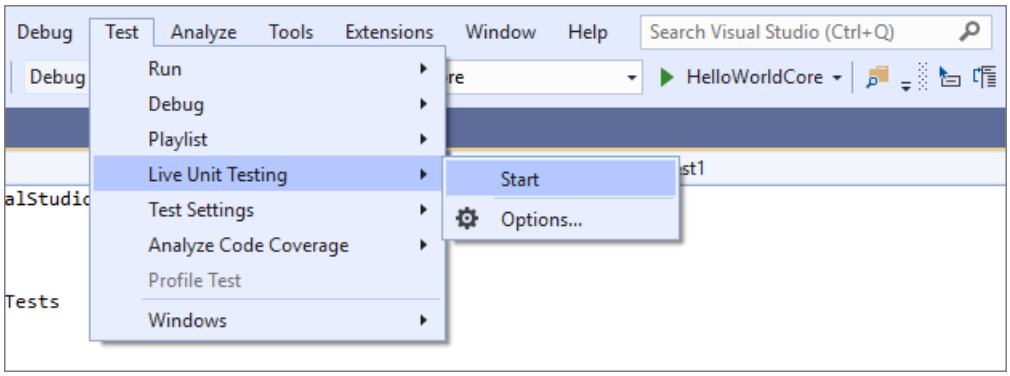
If you are using the MSTest, xUnit, or NUnit testing framework in Visual Studio 2017 or later, you can see live results of your unit tests.

NOTE

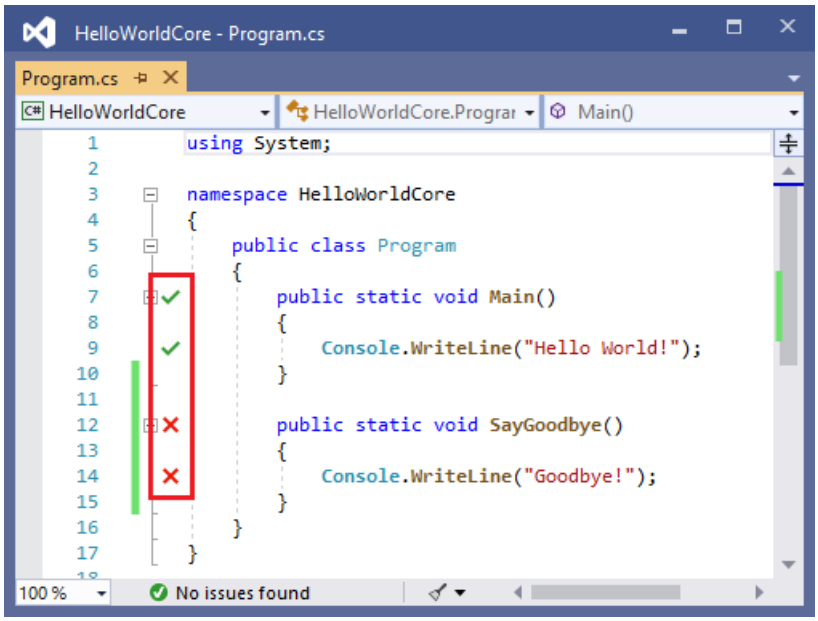
Live unit testing is available in Enterprise edition only.

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.



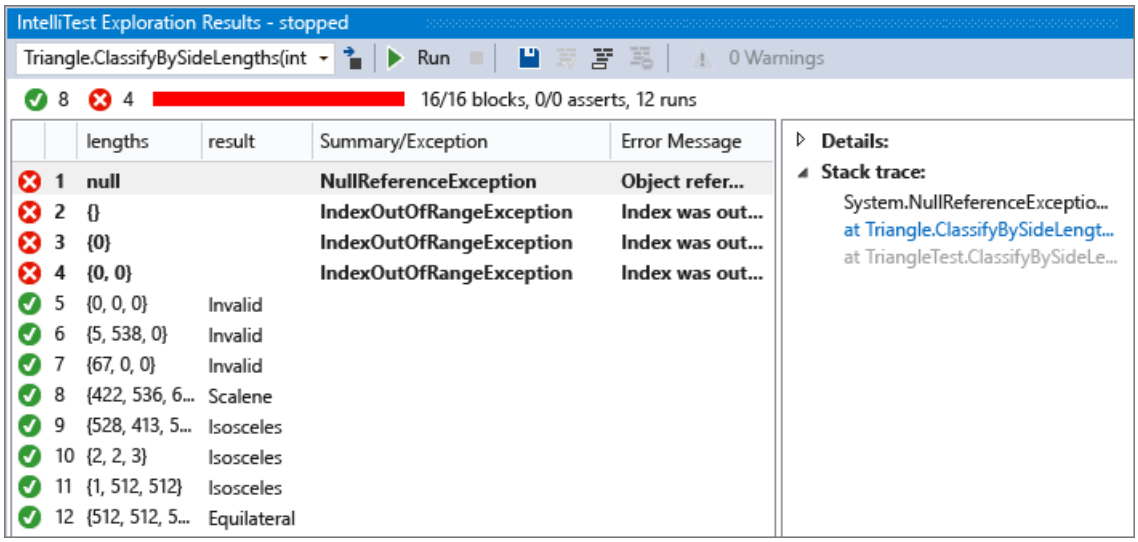
3. Click a test result indicator to see more information, such as the names of the tests that cover that method.

For more information about live unit testing, see Live unit testing.

Generate unit tests with IntelliTest

When you run IntelliTest, you can see which tests are failing and add any necessary code to fix them. You can select which of the generated tests to save into a test project to provide a regression suite. As you change your code, rerun IntelliTest to keep the generated tests in sync with your code changes. To learn how, see Generate unit tests for your code with IntelliTest.

> **TIP**
> IntelliTest is only available for managed code that targets the .NET Framework.



Analyze code coverage

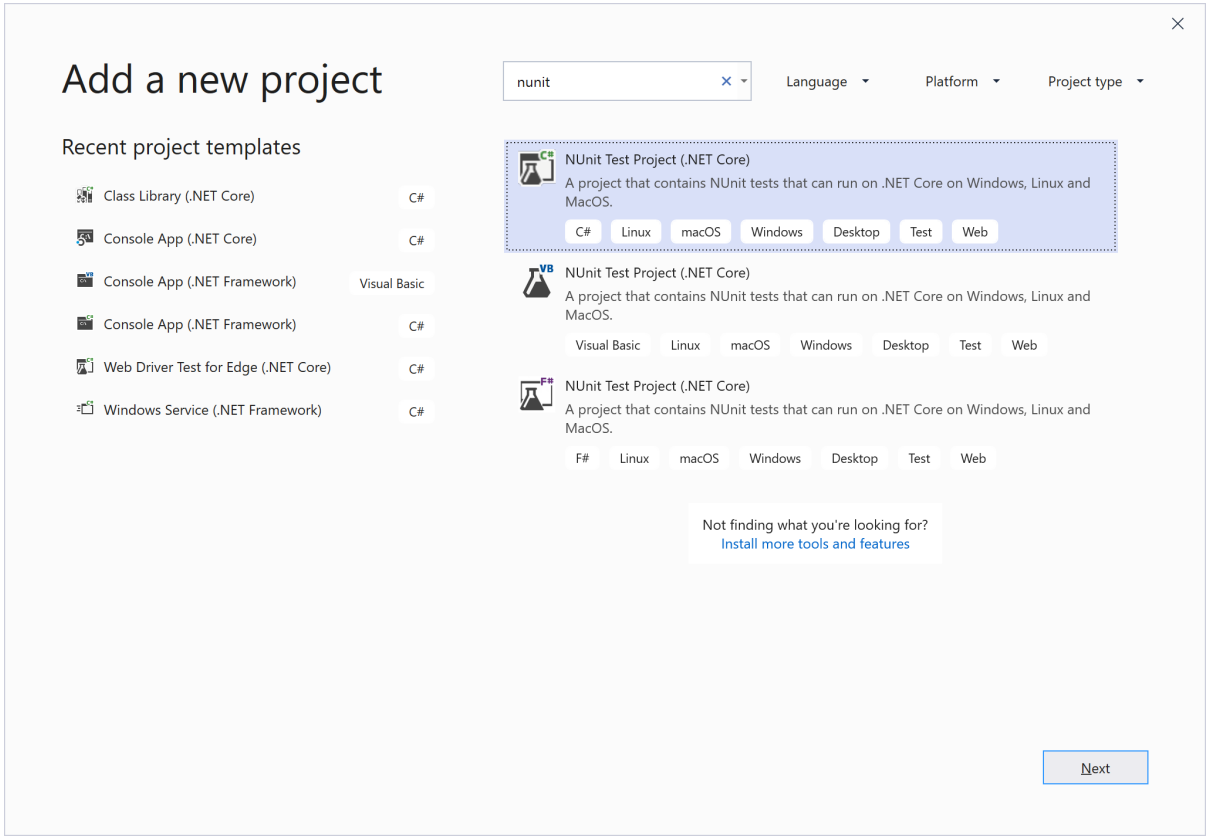
To determine what proportion of your project's code is actually being tested by coded tests such as unit tests, you can use the code coverage feature of Visual Studio. To guard effectively against bugs, your tests should exercise a large proportion of your code. To learn how, see Use code coverage to determine how much code is being tested.

Use a third-party test framework

You can run unit tests in Visual Studio by using third-party test frameworks such as Boost, Google, and NUnit. Use the **NuGet Package Manager** to install the NuGet package for the framework of your choice. Or, for the NUnit and xUnit test frameworks, Visual Studio includes preconfigured test project templates that include the necessary NuGet packages.

To create unit tests that use NUnit:

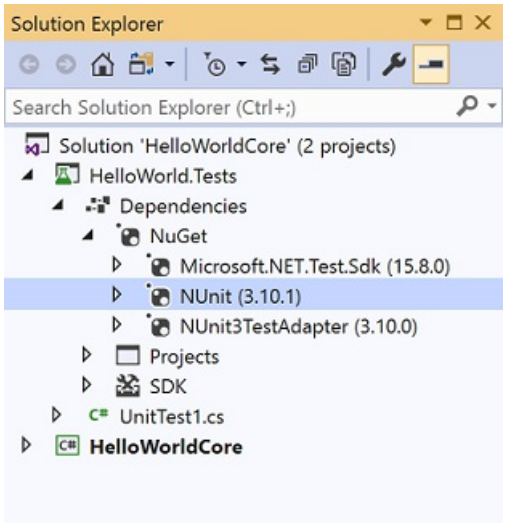
1. Open the solution that contains the code you want to test.
2. Right-click on the solution in **Solution Explorer** and choose **Add** > **New Project**.
3. Select the **NUnit Test Project** project template.



   Click **Next**, name the project, and then click **Create**.

   Name the project, and then click **OK** to create it.

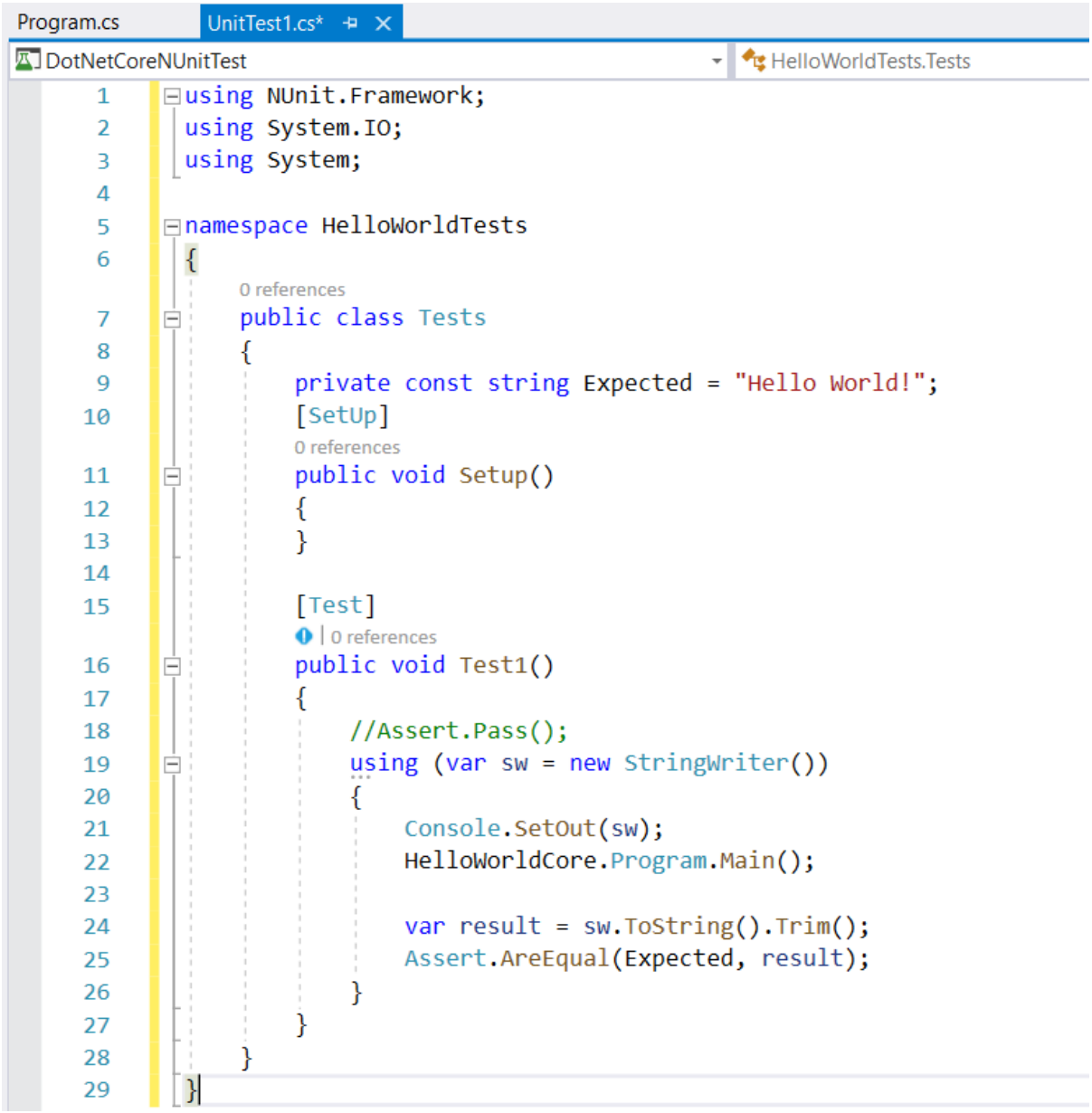
   The project template includes NuGet references to NUnit and NUnit3TestAdapter.



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)**.

See also

- Walkthrough: Create and run unit tests for managed code
- Create Unit Tests command
- Generate tests with IntelliTest
- Run tests with Test Explorer
- Analyze code coverage

Create a database and add tables in Visual Studio

6/16/2020 • 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.

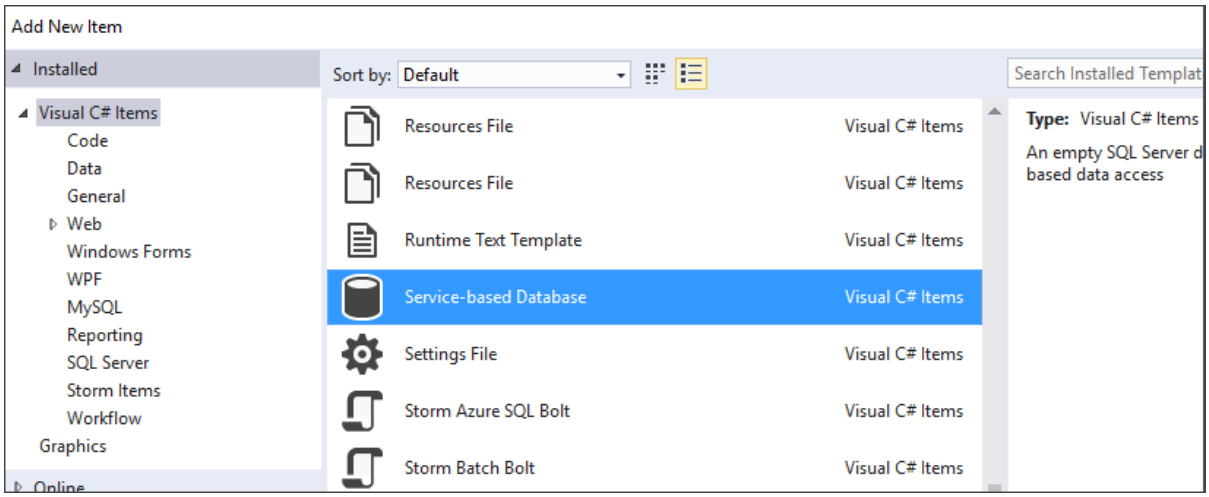
Prerequisites

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

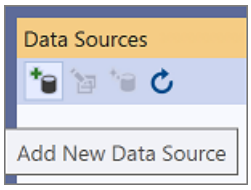
1. Create a new **Windows Forms App (.NET Framework)** project and name it **SampleDatabaseWalkthrough**.
2. On the menu bar, select **Project** > **Add New Item**.
3. In the list of item templates, scroll down and select **Service-based Database**.



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)

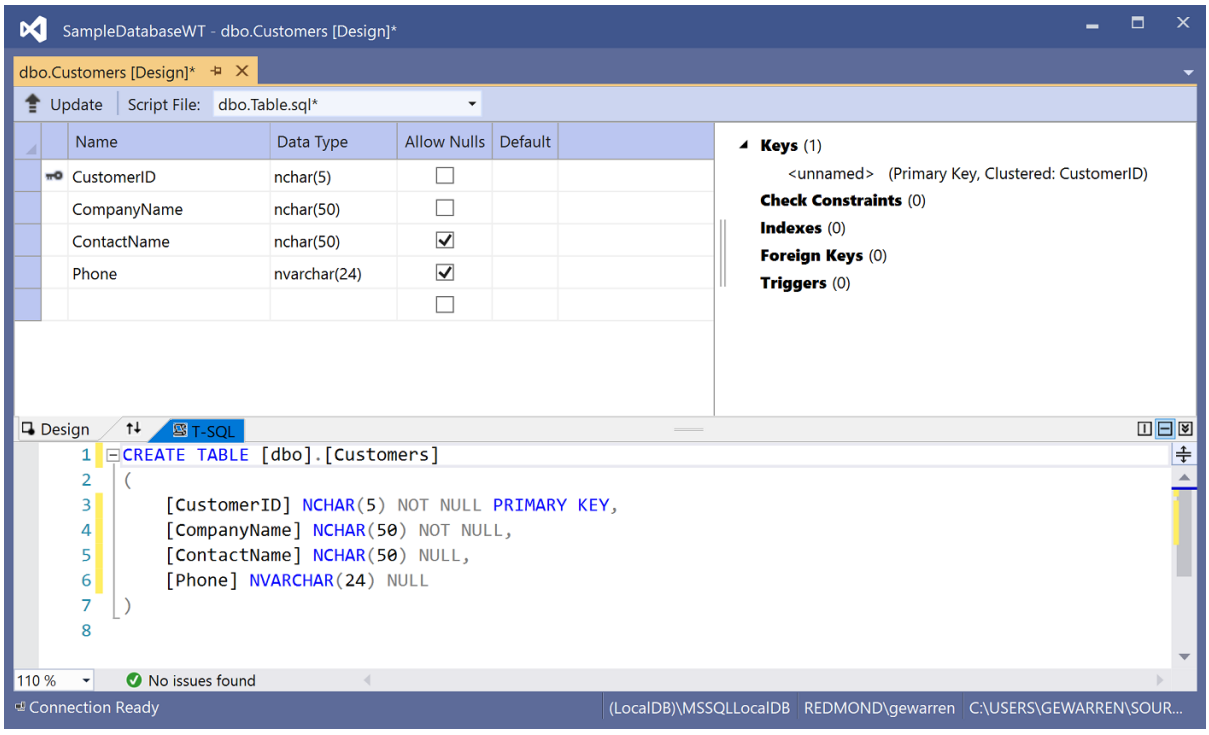
4. Right-click on the `CustomerID` row, and then select **Set Primary Key**.

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**.

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)

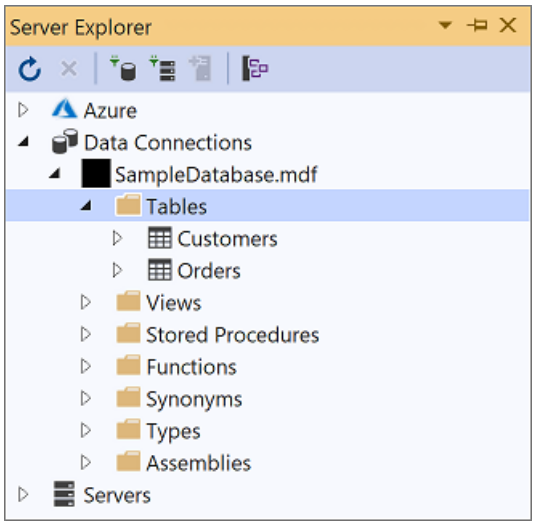
COLUMN NAME	DATA TYPE	ALLOW NULLS
`CustomerID`	`nchar(5)`	False (cleared)
`OrderDate`	`datetime`	True (selected)
`OrderQuantity`	`int`	True (selected)

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]
```

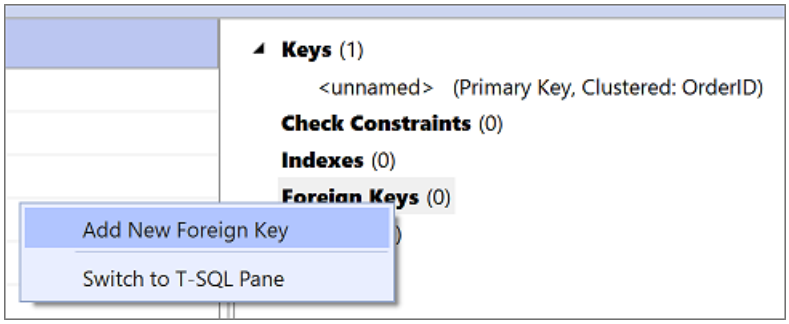
4. In the upper-left corner of the **Table Designer**, select **Update**.
5. In the **Preview Database Updates** dialog box, select **Update Database**.

   The Orders table is created in the local database file. If you expand the **Tables** node in Server Explorer, you see the two tables:



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**.
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 **Show Table 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.

   > **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.

7. On the menu bar, select **File** > **Save All**.

See also

- Accessing data in Visual Studio