Contents

JavaScript and TypeScript in Visual Studio

6/24/2022 • 4 minutes to read • Edit Online

Applies to: Visual Studio | Visual Studio for Mac | Visual Studio Code

Overview

Visual Studio 2022 provides rich support for JavaScript development, both using JavaScript directly, and also using theTypeScript programming language, which was developed to provide a more productive and enjoyable JavaScript development experience, especially when developing projects at scale. You can write JavaScript or TypeScript code in Visual Studio for many application types and services.

JavaScript Language Service

The JavaScript experience in Visual Studio 2022 is powered by the same engine that provides TypeScript support. This engine gives you better feature support, richness, and integration immediately out-of-the-box.

The option to restore to the legacy JavaScript language service is no longer available. Users have the new JavaScript language service out-of-the-box. The new language service is solely based on the TypeScript language service, which is powered by static analysis. This service enables us to provide you with better tooling, so your JavaScript code can benefit from richer IntelliSense based on type definitions. The new service is lightweight and consumes less memory than the legacy service, providing you with better performance as your code scales. We also improved performance of the language service to handle larger projects.

TypeScript support

By default, Visual Studio 2022 provides language support for JavaScript and TypeScript files to power IntelliSense without any specific project configuration.

For compiling TypeScript, Visual Studio gives you the flexibility to choose which version of TypeScript to use on a per-project basis.

In MSBuild compilation scenarios, the TypeScript NuGet package is the recommended method of adding TypeScript compilation support to your project. Visual Studio will give you the option to add this package the first time you add a TypeScript file to your project. This package is also available at any time through the NuGet package manager. When the NuGet package is used, the corresponding language service version will be used for language support in your project. Note: The minimum supported version of this package is 3.6.

Projects configured for npm can specify their own version of the TypeScript language service by adding the TypeScript npm package. You can specify the version using the npm manager in supported projects. Note: The minimum supported version of this package is 2.1.

The TypeScript SDK has been deprecated in Visual Studio 2022. Existing projects that rely on the SDK should be upgraded to using the NuGet package. For projects that cannot be upgraded immediately, the SDK is still available on the Visual Studio Marketplace and as an optional component in the Visual Studio installer.

> **TIP**
>
> For projects developed in Visual Studio 2022, we encourage you to use the TypeScript NuGet or the TypeScript npm package for greater portability across different platforms and environments. For more information, seeCompile TypeScript code using NuGetandCompile TypeScript code using tsc.

Project Templates

Starting in Visual Studio 2022, there is a new JavaScript/TypeScript project type (.esproj) that allows you to create standalone Angular, React, and Vue projects in Visual Studio. These front-end projects are created using the framework CLI tools you have installed on your local machine, so the version of the template is up to you.

Within these new projects, you can run JavaScript and TypeScript unit tests, easily add and connect ASP.NET Core API projects and download your npm modules using the npm manager. Check out some of the quickstarts and tutorials to get started. For more information, see Visual Studio tutorials | JavaScript and TypeScript.

Overview

Visual Studio 2019 provides rich support for JavaScript development, both using JavaScript directly, and also using the TypeScript programming language, which was developed to provide a more productive and enjoyable JavaScript development experience, especially when developing projects at scale. You can write JavaScript or TypeScript code in Visual Studio for many application types and services.

JavaScript Language Service

The JavaScript experience in Visual Studio 2019 is powered by the same engine that provides TypeScript support. This gives you better feature support, richness, and integration immediately out-of-the-box.

The option to restore to the legacy JavaScript language service is no longer available. Users now have the new JavaScript language service out-of-the-box. The new language service is solely based on the TypeScript language service, which is powered by static analysis. This enables us to provide you with better tooling, so your JavaScript code can benefit from richer IntelliSense based on type definitions. The new service is lightweight and consumes less memory than the legacy service, providing you with better performance as your code scales. We also improved performance of the language service to handle larger projects.

TypeScript support

Visual Studio 2019 provides several options for integrating TypeScript compilation into your project:

- The TypeScript NuGet package. When the NuGet package for TypeScript 3.2 or higher is installed into your project, the corresponding version of the TypeScript language service gets loaded in the editor.
- The TypeScript npm package. When the npm package for TypeScript 2.1 or higher is installed into your project, the corresponding version of the TypeScript language service gets loaded in the editor.
- The TypeScript SDK, available by default in the Visual Studio installer, as well as a standalone SDK download from the VS Marketplace.

> **TIP**
> For projects developed in Visual Studio 2019, we encourage you to use the TypeScript NuGet or the TypeScript npm package for greater portability across different platforms and environments. For more information, see Compile TypeScript code using NuGet and Compile TypeScript code using tsc.

Projects

UWP JavaScript apps are no longer supported in Visual Studio 2019. You cannot create or open JavaScript UWP projects (files with extension *.jsproj*). You can learn more using our documentation on creating Progressive Web Apps (PWAs) that run well on Windows.

First look at the Visual Studio IDE

6/24/2022 • 4 minutes to read • Edit Online

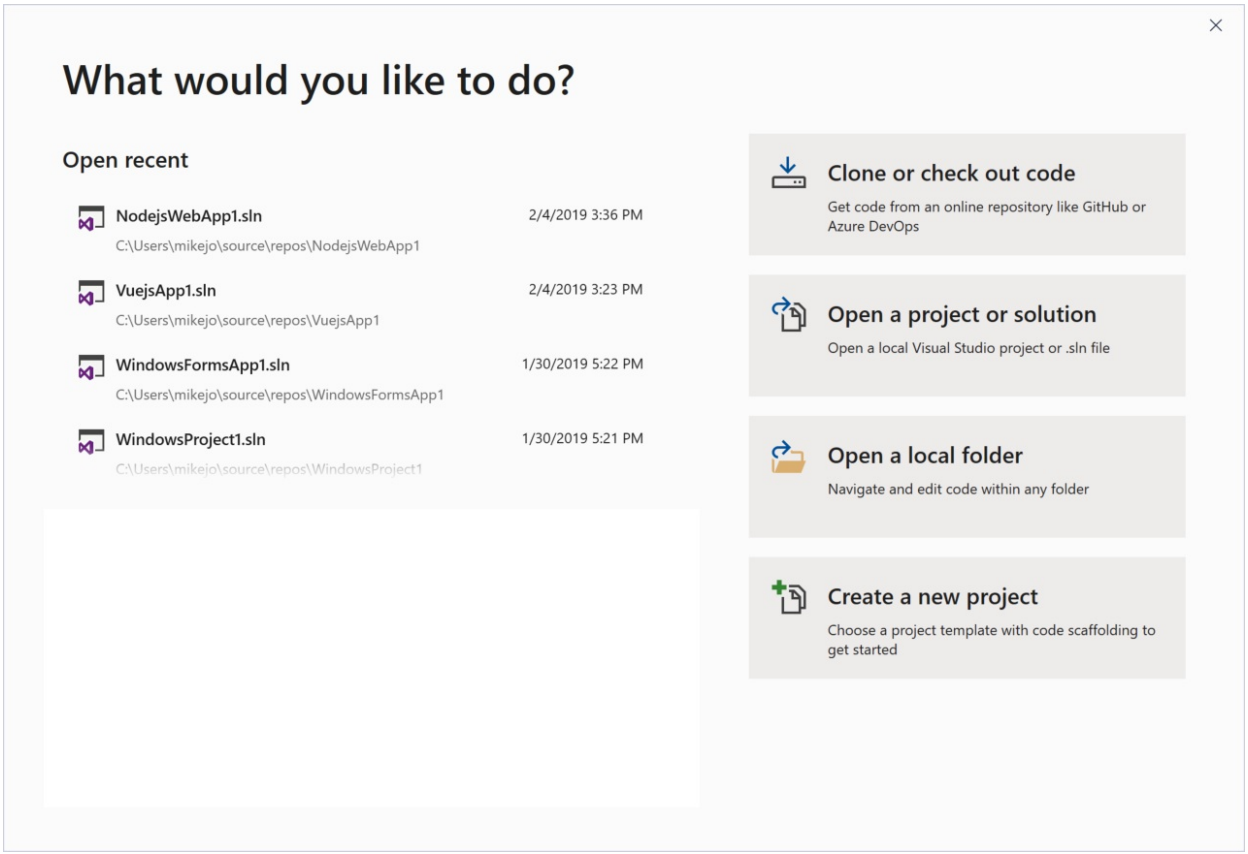
Applies to: Visual Studio | Visual Studio for Mac | Visual Studio Code

In this 5-10 minute introduction to the Visual Studio integrated development environment (IDE), we'll take a tour of some of the windows, menus, and other UI features.

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

Start window

The first thing you'll see after you launch Visual Studio is the start window. The start window is designed to help you "get to code" faster. It has options to close or check out code, open an existing project or solution, create a new project, or simply open a folder that contains some code files.



If this is the first time you're using Visual Studio, your recent projects list will be empty.

If you work with non-MSBuild based codebases, you'll use the **Open a local folder** option to open your code in Visual Studio. For more information, see Develop code in Visual Studio without projects or solutions. Otherwise, you can create a new project or clone a project from a source provider such as GitHub or Azure DevOps.

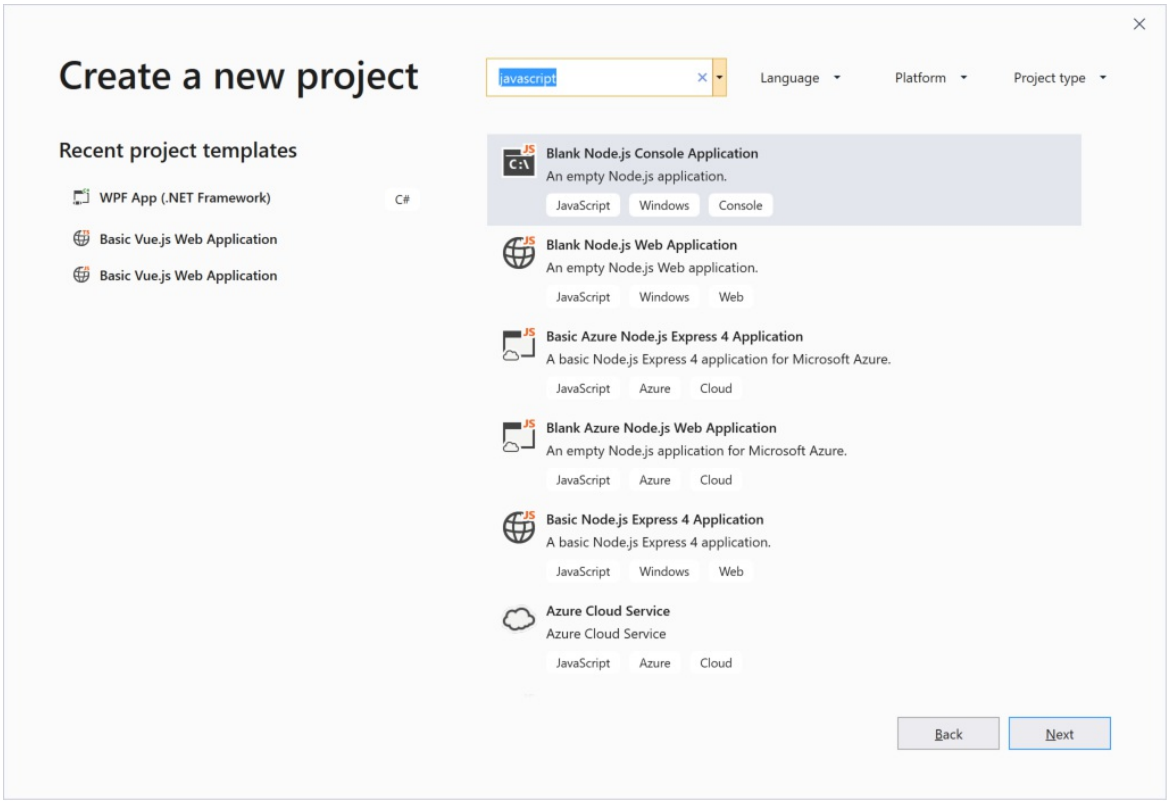
The **Continue without code** option simply opens the Visual Studio development environment without any specific project or code loaded. You might choose this option to join a Live Share session or attach to a process for debugging. You can also press **Esc** to close the start window and open the IDE.

Create a project

To continue exploring Visual Studio's features, let's create a new project.

1. On the start window, select **Create a new project**, and then in the search box type in **javascript** to filter the list of project types to those that contain "javascript" in their name or language type.

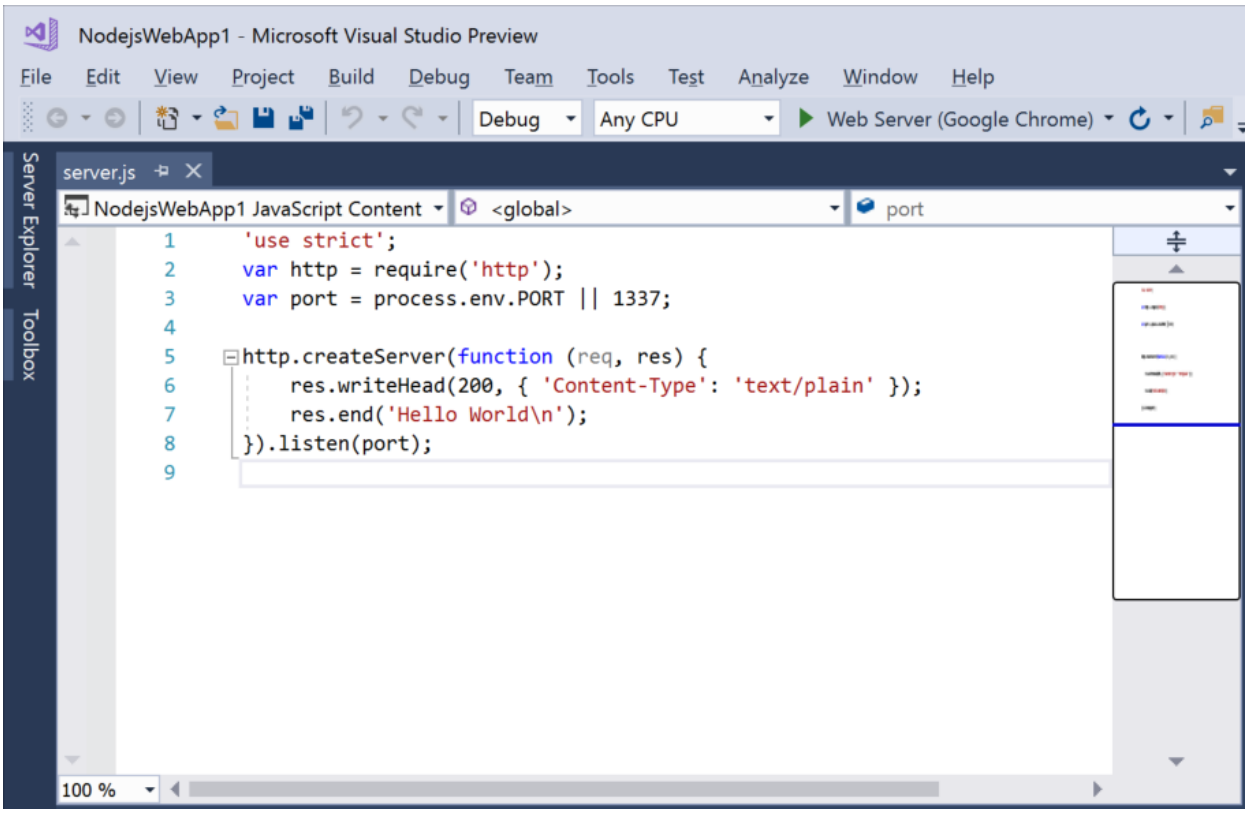
   Visual Studio provides various kinds of project templates that help you get started coding quickly. (Alternatively, if you're a TypeScript developer, feel free to create a project in that language. The UI we'll be looking at is similar for all programming languages.)



2. Choose a **Blank Node.js Web Application** project template and click **Next**.

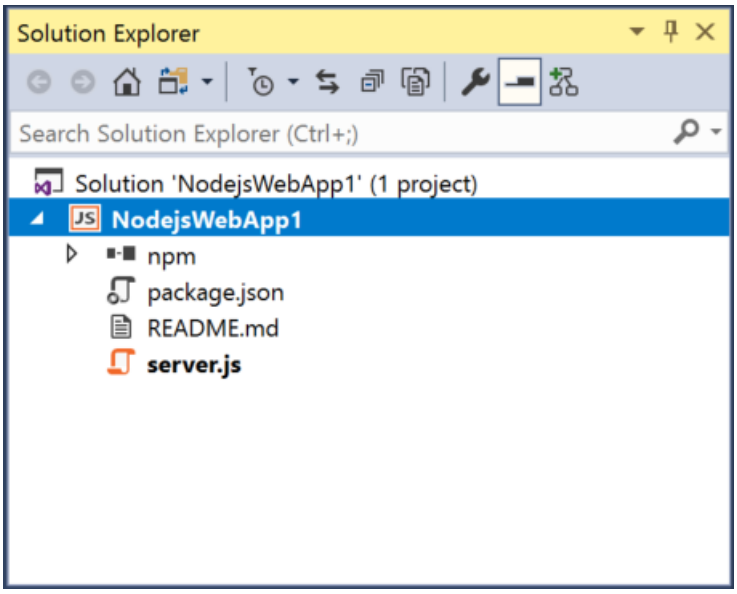
3. In the **Configure your new project** dialog box that appears, accept the default project name and choose **Create**.

The project is created and a file named *server.js* opens in the **Editor** window. The **Editor** shows the contents of files, and is where you'll do most of your coding work in Visual Studio.



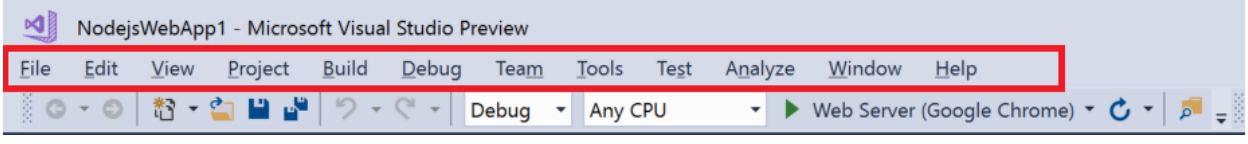
Solution Explorer

Solution Explorer, which is typically on the right-hand side of Visual Studio, shows you a graphical representation of the hierarchy of files and folders in your project, solution, or code folder. You can browse the hierarchy and navigate to a file in **Solution Explorer**.



Menus

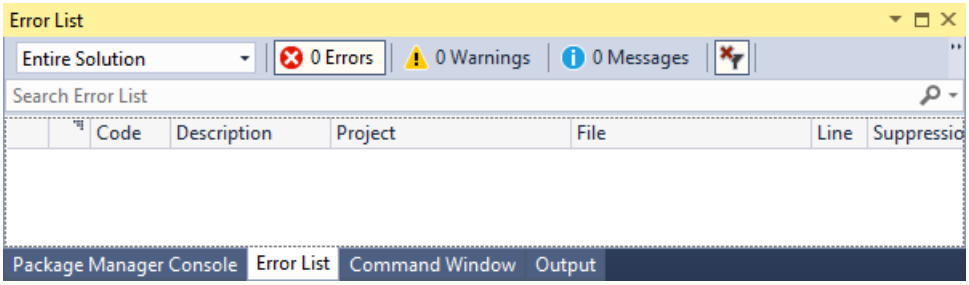
The menu bar along the top of Visual Studio groups commands into categories. For example, the **Project** menu contains commands related to the project you're working in. On the **Tools** menu, you can customize how Visual Studio behaves by selecting **Options**, or add features to your installation by selecting **Get Tools and Features**.



Let's open the **Error List** window by choosing the **View** menu, and then **Error List**.

Error List

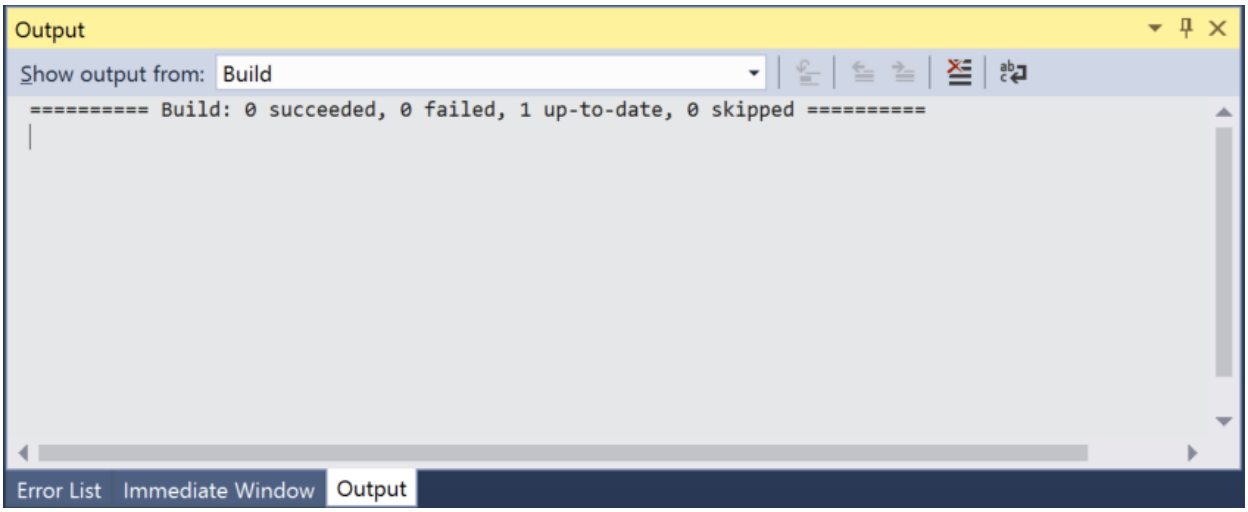
The **Error List** shows you errors, warning, and messages regarding the current state of your code. If there are any errors (such as a missing brace or semicolon) in your file, or anywhere in your project, they're listed here.



Output window

The **Output** window shows you output messages from building your project and from your source control provider.

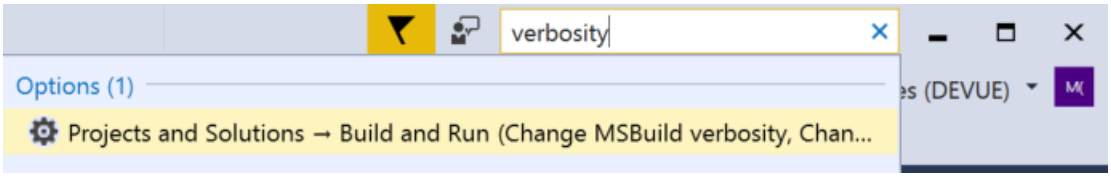
Let's build the project to see some build output. From the **Build** menu, choose **Build Solution**. The **Output** window automatically obtains focus and display a successful build message.



Search box

The search box is a quick and easy way to do pretty much anything in Visual Studio. You can enter some text related to what you want to do, and it'll show you a list of options that pertain to the text. For example, imagine you want to increase the build output's verbosity to display additional details about what exactly build is doing. Here's how you might do that:

1. Type **verbosity** into the search box. From the displayed results, choose **Projects and Solutions --> Build and Run** under the **Options** category.

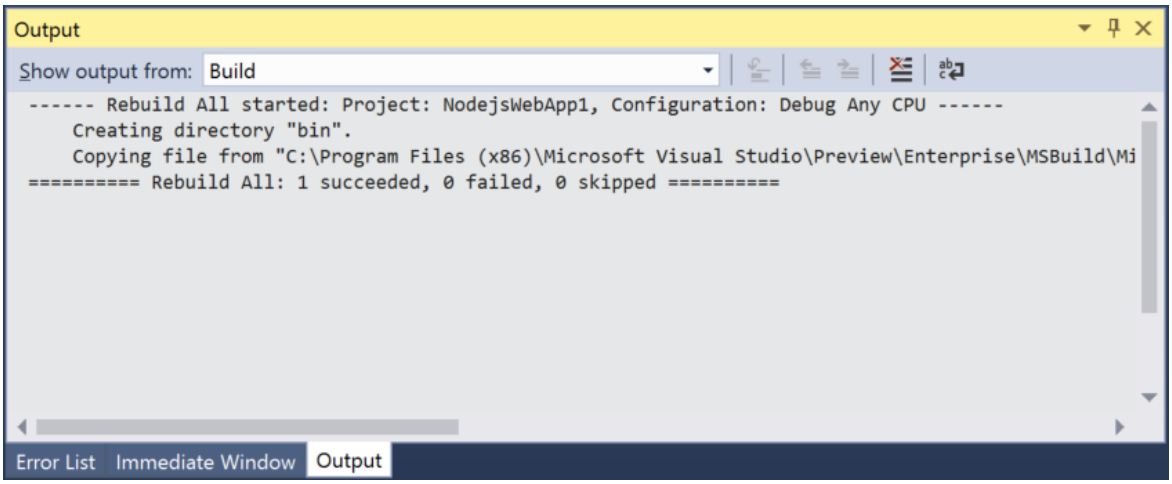


   The **Options** dialog box opens to the **Build and Run** options page.

2. Under **MSBuild project build output verbosity**, choose **Normal**, and then click **OK**.

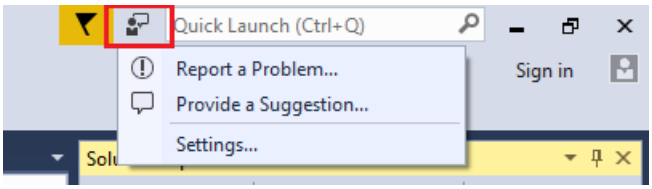
3. Build the project again by right-clicking on the **NodejsWebApp1** project in **Solution Explorer** and choosing **Rebuild** from the context menu.

   This time the **Output** window shows more verbose logging from the build process, including which files were copied where.



Send Feedback menu

Should you encounter any problems while you're using Visual Studio, or if you have suggestions for how to improve the product, you can use the **Send Feedback** menu at the top of the Visual Studio window.



Next steps

We've looked at just a few of the features of Visual Studio to get acquainted with the user interface. To explore further:

Learn about the code editor

Learn about projects and solutions

See also

- Overview of the Visual Studio IDE
- More features of Visual Studio 2017
- Change theme and font colors

Create a React app

6/24/2022 • 2 minutes to read • Edit Online

Applies to: Visual Studio ✗ Visual Studio for Mac ✗ Visual Studio Code

In this 5-10 minute introduction to the Visual Studio integrated development environment (IDE), you create and run a simple React frontend web application.

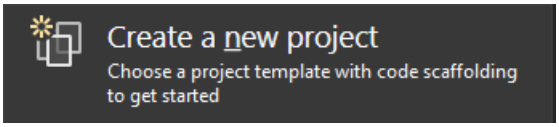
Prerequisites

Make sure to install the following:

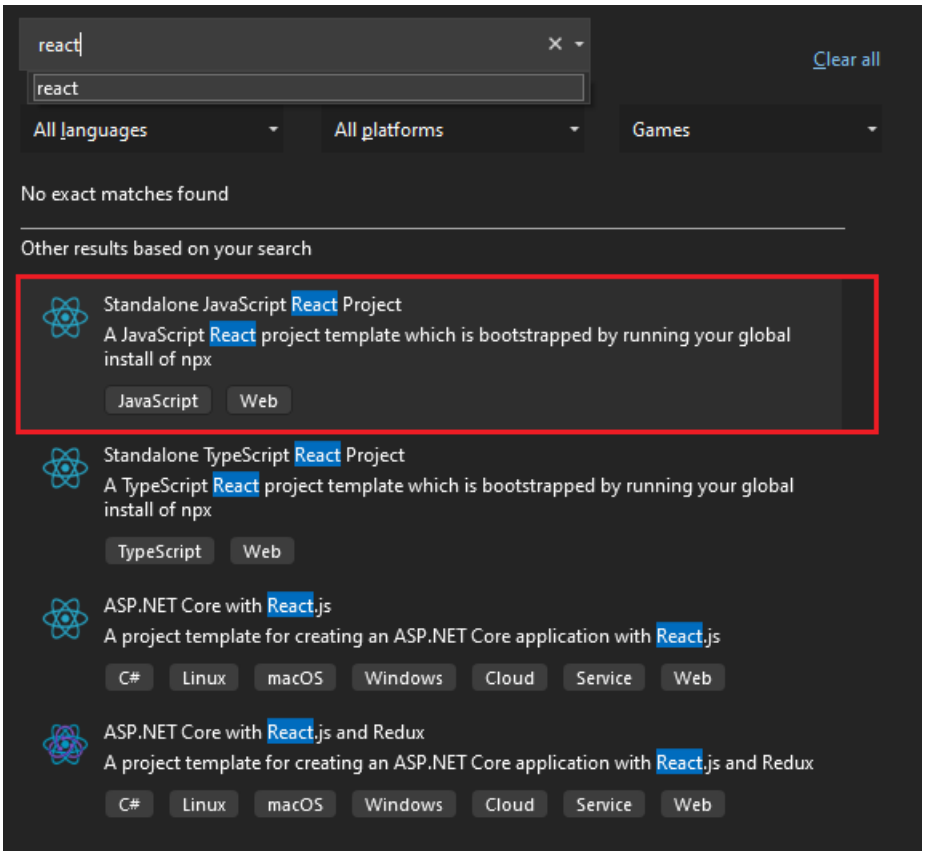
- Visual Studio 2022 or later. Go to the Visual Studio downloads page to install it for free.
- npm (https://www.npmjs.com/), which is included with Node.js
- npx (https://www.npmjs.com/package/npx)

Create your app

1. In the Start window (choose **File** > **Start Window** to open), select **Create a new project**.

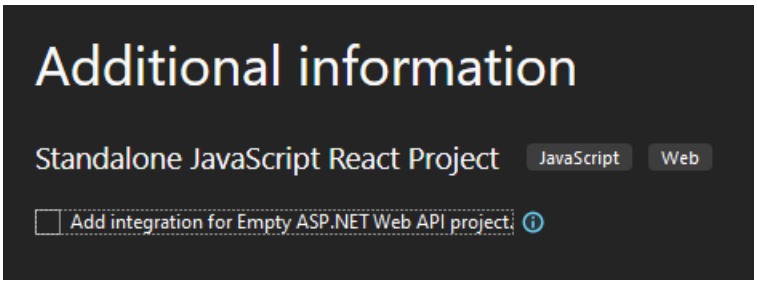


2. Search for React in the search bar at the top and then select **Standalone JavaScript React Template** or **Standalone TypeScript React Template**, based on your preference.



3. Give your project and solution a name.

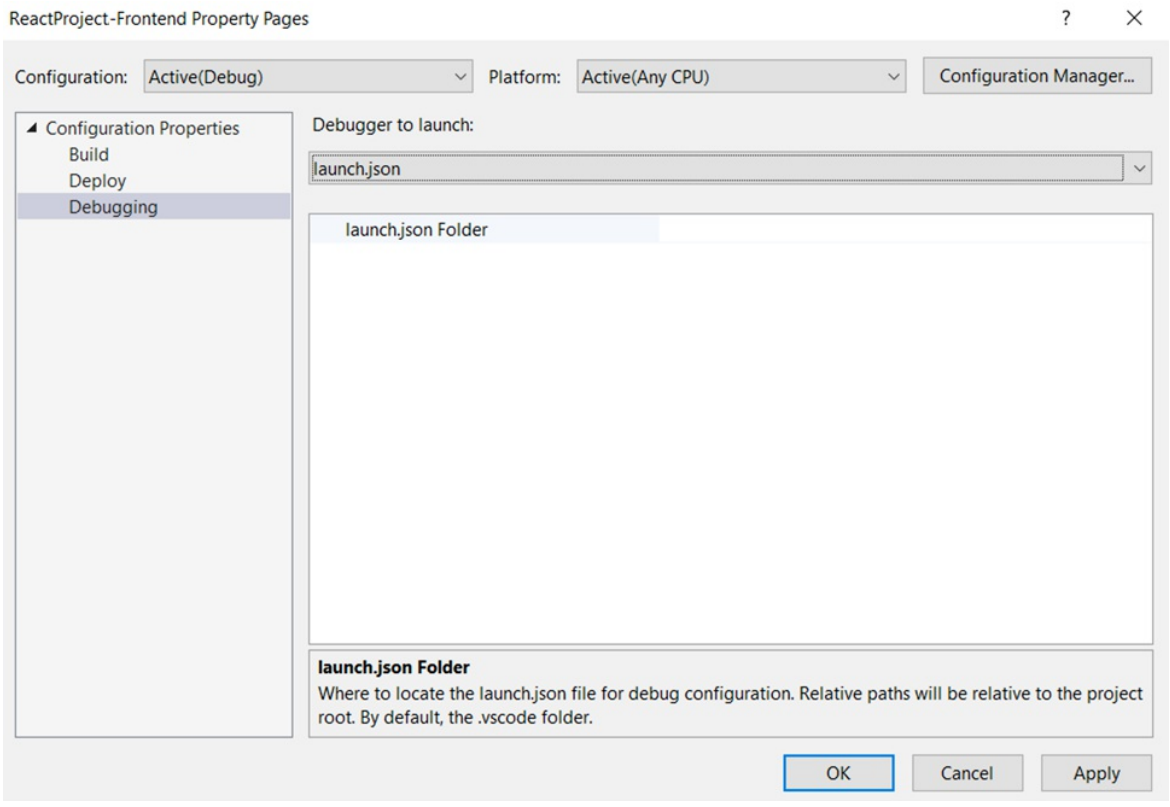
If you previously selected Standalone JavaScript React Template, when you get to the Additional information window be sure NOT to check the **Add integration for Empty ASP.NET Web API Project** option. This option adds files to your React template so that it can be hooked up with the ASP.NET Core project, if an ASP.NET Core project is added.



Please note that creation of the React project takes a moment because the create-react-app command that runs at this time also runs the npm install command

Set the project properties

1. In Solution Explorer, right-click the React project, select **Properties**, and then go the **Debugging** section.
2. Change the Debugger to launch to the **launch.json** option.



Build Your Project

Choose **Build** > **Build Solution** to build the project.

Start Your Project

Press **F5** or select the **Start** button at the top of the window, and you'll see a command prompt:

- npm running the react-scripts start command

> **NOTE**
>
> Check console output for messages, such as a message instructing you to update your version of Node.js.

Next, you should see the base React app appear!

Next steps

For ASP.NET Core integration:

Create an ASP.NET Core app with React

Create an Angular app

6/24/2022 • 2 minutes to read • Edit Online

Applies to: Visual Studio ✗ Visual Studio for Mac ✗ Visual Studio Code

In this 5-10 minute introduction to the Visual Studio integrated development environment (IDE), you create and run a simple Angular frontend web application.

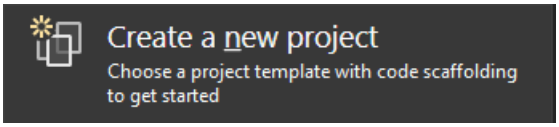
Prerequisites

Make sure to install the following:

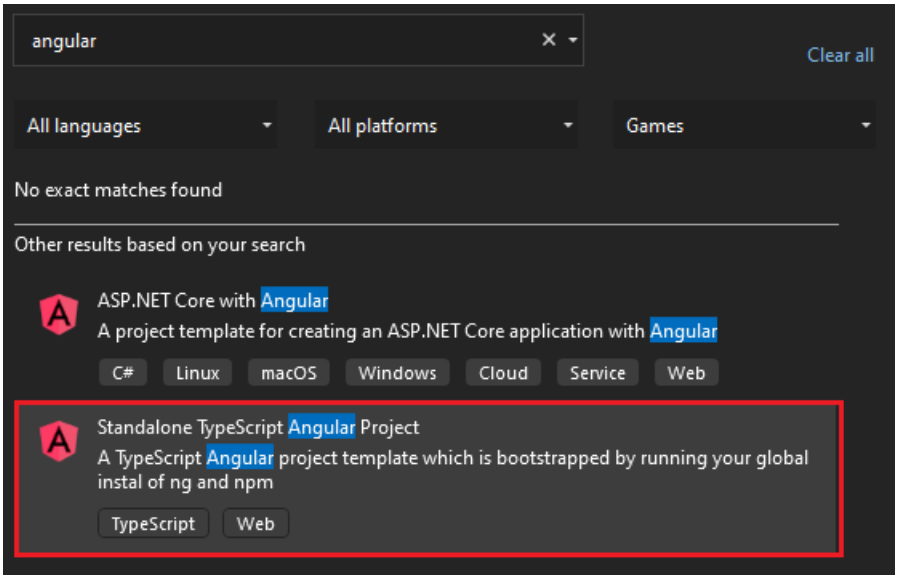
- Visual Studio 2022 Preview 2 or later. Go to the Visual Studio downloads page to install it for free.
- npm (https://www.npmjs.com/), which is included with Node.js
- Angular CLI (https://angular.io/cli) This can be the version of your choice

Create your app

1. In the Start window (choose **File** > **Start Window** to open), select **Create a new project**.



2. Search for Angular in the search bar at the top and then select **Standalone TypeScript Angular Template**.



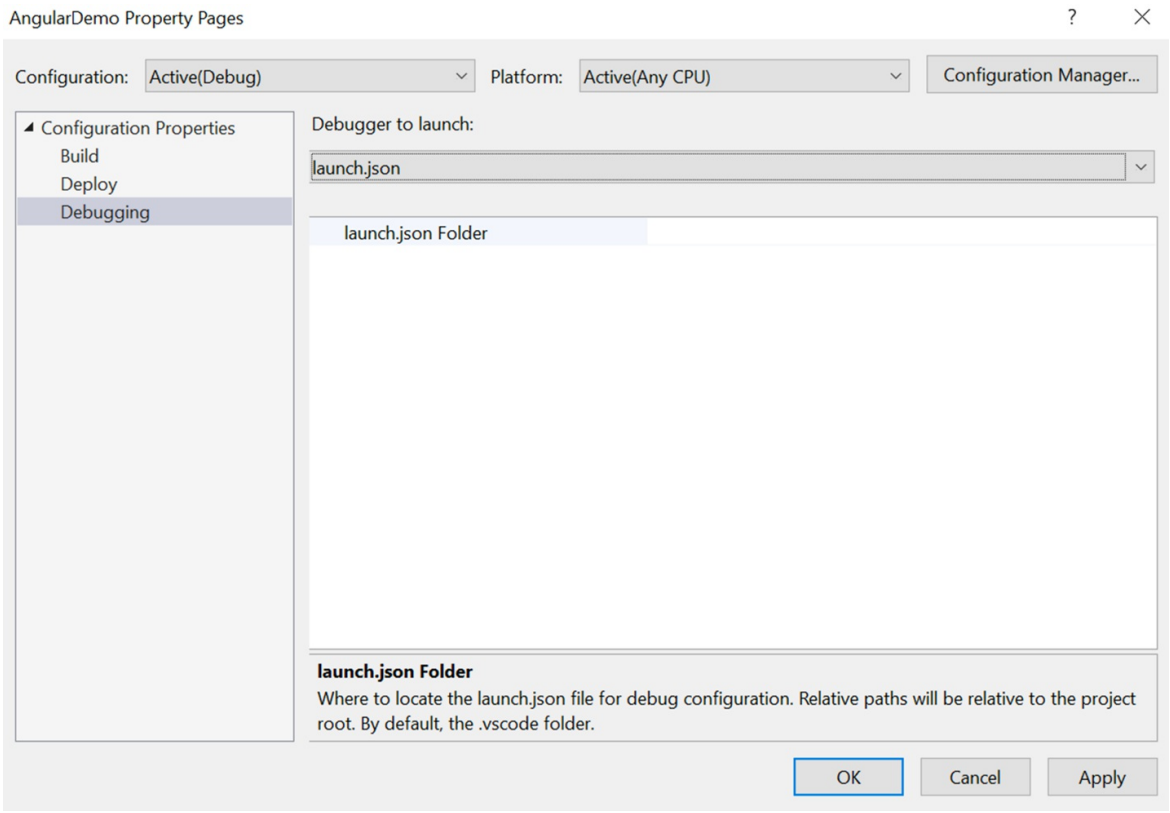
3. Give your project and solution a name.

   When you get to the Additional information window, be sure NOT to check the **Add integration for Empty ASP.NET Web API Project** option. This option adds files to your Angular template so that it can be hooked up with the ASP.NET Core project, if an ASP.NET Core project is added.

Set the project properties

1. In Solution Explorer, right-click the Angular project, select **Properties**, and then go the **Debugging** section.

2. Change the Debugger to launch to the **launch.json** option.



Build Your Project

Choose **Build** > **Build Solution** to build the project.

Note, the initial build may take a while, as the Angular CLI will run the npm install command.

Start Your Project

Press **F5** or select the **Start** button at the top of the window, and you'll see a command prompt:

- The Angular CLI running the ng start command

> **NOTE**
> Check console output for messages, such as a message instructing you to update your version of Node.js.

Next, you should see the base Angular apps appear!

Next steps

For ASP.NET Core integration:

Create an ASP.NET Core app with Angular

Create a Vue.js app

6/24/2022 • 2 minutes to read • Edit Online

Applies to: Visual Studio ✗ Visual Studio for Mac ✗ Visual Studio Code

In this 5-10 minute introduction to the Visual Studio integrated development environment (IDE), you create and run a simple Vue.js frontend web application.

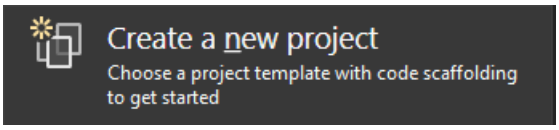
Prerequisites

Make sure to install the following:

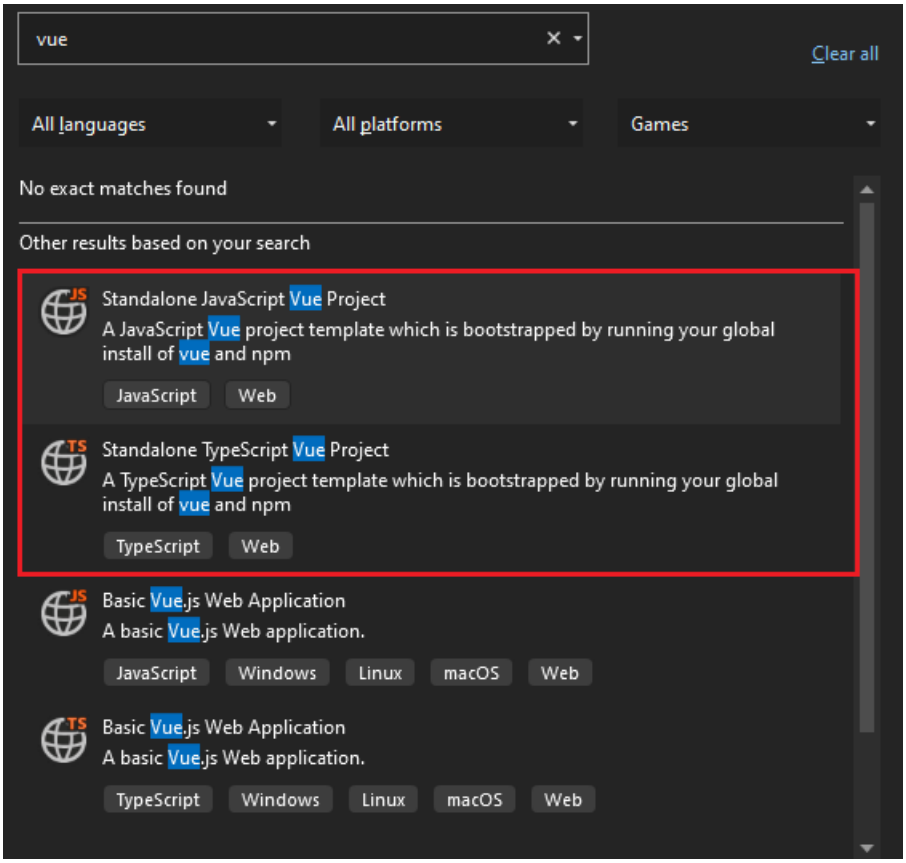
- Visual Studio 2022 Preview 2 or later. Go to the Visual Studio downloads page to install it for free.
- npm (https://www.npmjs.com/), which is included with Node.js
- Vue.js (Installation | Vue.js (vuejs.org))
- Vue.js CLI ((Installation | Vue.js (vuejs.org))

Create your app

1. In the Start window (choose **File** > **Start Window** to open), select **Create a new project**.



2. Search for Vue in the search bar at the top and then select **Standalone JavaScript Vue Template** or **Standalone TypeScript Vue Template**, based on your preference.

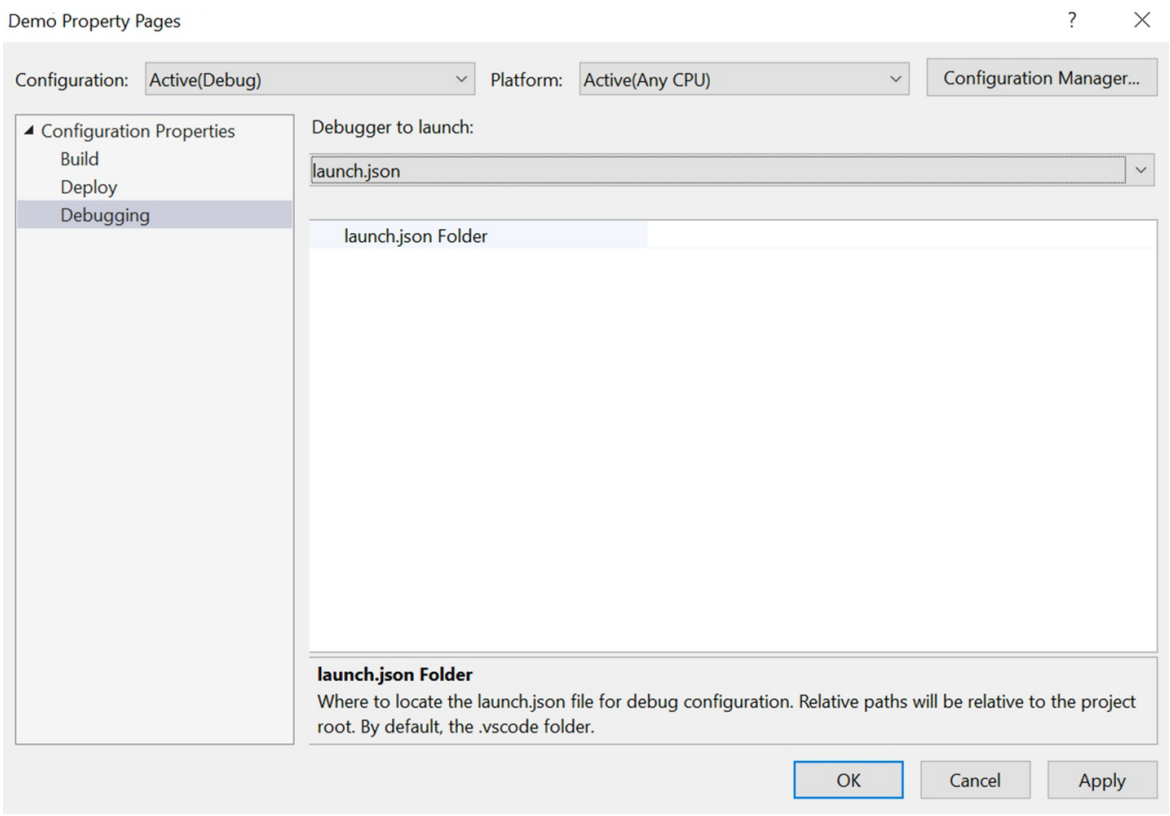


3. Give your project and solution a name.

Set the project properties

1. In Solution Explorer, right-click the Vue.js project, select **Properties**, and then go the **Debugging** section.

2. Change the Debugger to launch to the **launch.json** option.



Build Your Project

Choose **Build** > **Build Solution** to build the project.

Start Your Project

Press **F5** or select the **Start** button at the top of the window, and you'll see a command prompt:

- npm running the vue-cli-service start command

> **NOTE**
> Check console output for messages, such as a message instructing you to update your version of Node.js.

Next, you should see the base Vue.js app appear!

Next steps

For ASP.NET Core integration:

Create an ASP.NET Core app with Vue

Tutorial: Create a Node.js and Express app in Visual Studio

6/24/2022 • 11 minutes to read • Edit Online

Applies to: Visual Studio Visual Studio for Mac Visual Studio Code

This tutorial for Visual Studio development uses Node.js and Express. In this tutorial, you create a simple Node.js web app, add some code, explore some features of the IDE, and run the app.

In this tutorial, you learn how to:

- Create a Node.js project.
- Add some code.
- Use IntelliSense to edit code.
- Run the app.
- Hit a breakpoint in the debugger.

Before you begin, here's a quick FAQ to introduce you to some key concepts:

- **What is Node.js?**

  Node.js is a server-side JavaScript runtime environment that executes JavaScript code.

- **What is npm?**

  A package manager makes it easier to publish and share Node.js source code libraries. The default package manager for Node.js is npm. The npm package manager simplifies library installation, updating, and uninstallation.

- **What is Express?**

  Express is a server web application framework that Node.js uses to build web apps. With Express, you can use different front-end frameworks to create a user interface. This tutorial uses Pug, formerly called Jade, for its front-end framework.

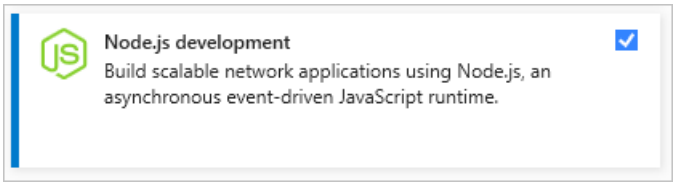
Prerequisites

This tutorial requires the following prerequisites:

- Visual Studio with the Node.js development workload installed.

  If you haven't yet installed Visual Studio:

  1. Go to the Visual Studio downloads page to install Visual Studio for free.
  2. In the Visual Studio Installer, select the **Node.js development** workload, and select **Install**.

  If you have Visual Studio installed already:

1. In Visual Studio, go to **Tools** > **Get Tools and Features**.
2. In the Visual Studio Installer, select the **Node.js development** workload, and select **Modify** to download and install the workload.

- The Node.js runtime installed:

  If you don't have the Node.js runtime installed, install the LTS version from the Node.js website. The LTS version has the best compatibility with other frameworks and libraries.

  The Node.js tools in the Visual Studio Node.js workload support both Node.js 32-bit and 64-bit architecture versions. Visual Studio requires only one version, and the Node.js installer only supports one version at a time.

  Visual Studio usually detects the installed Node.js runtime automatically. If not, you can configure your project to reference the installed runtime:

  1. After you create a project, right-click the project node and select **Properties**.
  2. In the **Properties** pane, set the **Node.exe path** to reference a global or local installation of Node.js. You can specify the path to a local interpreter in each of your Node.js projects.

This tutorial was tested with Node.js 16.14.0.

This tutorial was tested with Node.js 8.10.0.

Create a new Node.js project

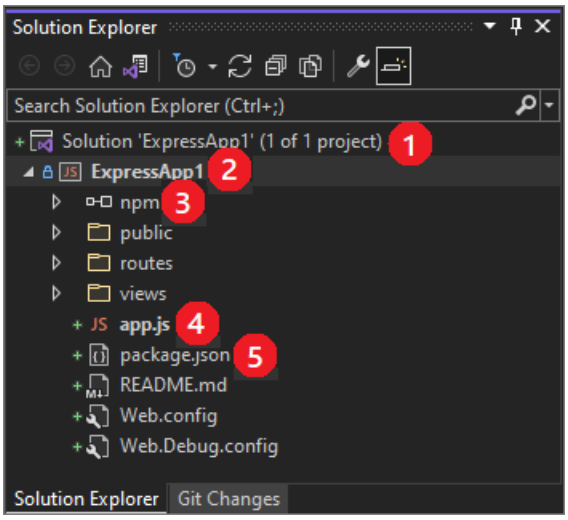
Visual Studio manages files for a single application in a *project*. The project includes source code, resources, and configuration files.

In this tutorial, you begin with a simple project that has code for a Node.js and Express app.

1. Open Visual Studio, and select **Esc** to close the start window.
2. Select **Ctrl+Q**, enter *node.js* in the search box, and then select **Basic Azure Node.js Express 4 Application - JavaScript** from the dropdown list.

   If you don't see the **Basic Azure Node.js Express 4 Application** choice, you need to install the Node.js development workload. For instructions, see Prerequisites.
3. In the **Configure your new project** dialog box, select **Create**.

   Visual Studio creates the new solution and project, and opens the project in the right pane. The *app.js* project file opens in the editor in the left pane.
4. Look at the project structure in **Solution Explorer** in the right pane.



- At the top level is the *solution* (**1**), which by default has the same name as your project. A solution, represented by a *.sln* file on disk, is a container for one or more related projects.
- Your project (**2**), using the name you gave in the **Configure your new project** dialog box, is highlighted in bold. In the file system, the project is a *.njsproj* file in your project folder.

  You can see and set project properties and environment variables by right-clicking the project and selecting **Properties** from the context menu. You can work with other development tools, because the project file doesn't make custom changes to the Node.js project source.
- The **npm** node (**3**) shows any installed npm packages. You can right-click the npm node to search for and install npm packages by using a dialog box.

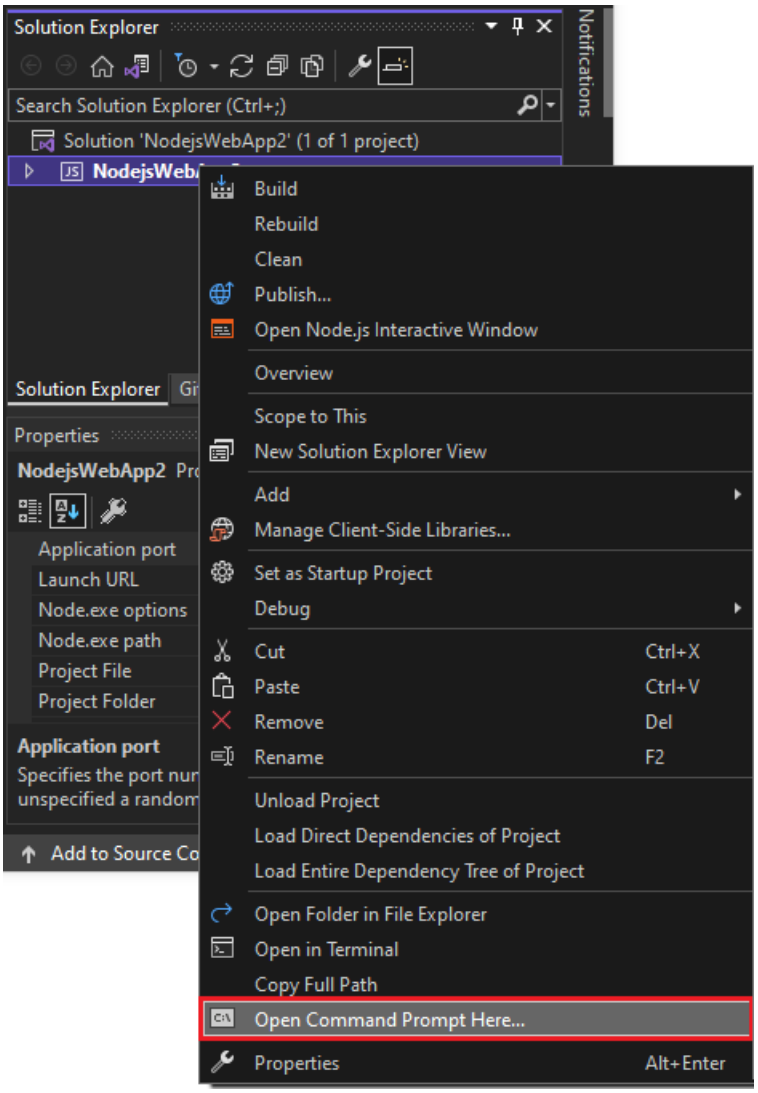
  You can install and update packages by using the settings in *package.json* and the right-click options in the **npm** node.
- Project files (**4**) appear under the project node. The project startup file, **app.js**, is bold.

  You can set the startup file by right-clicking a file in the project and selecting **Set as Node.js startup file**.
- Npm uses the **package.json** file (**5**) to manage dependencies and versions for locally installed packages. For more information, see Manage npm packages.

5. Open the **npm** node to make sure all the required npm packages are present.

   If any packages are listed as **(missing)**, right-click the **npm** node, select **Install npm Packages**, and install the missing packages.

   To install npm packages or Node.js commands from a command prompt, right-click the project node and select **Open Command Prompt Here**.

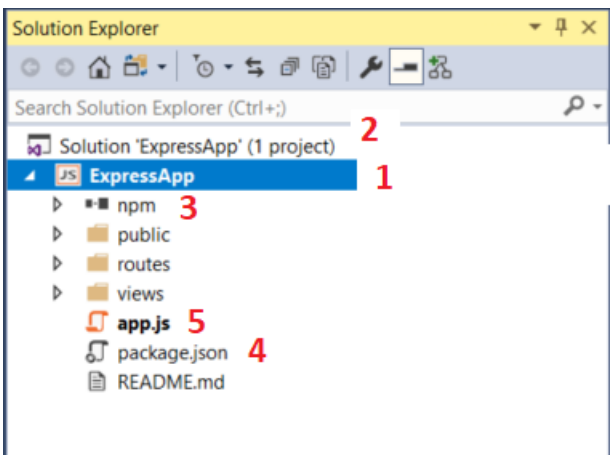


1. Open Visual Studio.
2. Create a new project.

   Select **Esc** to close the start window. Select **Ctrl + Q** to open the search box, enter **Node.js**, and then select **Create a new Basic Azure Node.js Express 4 application** (JavaScript). In the dialog box that appears, select **Create**.

   If you don't see the **Basic Azure Node.js Express 4 application** project template, you need to add the **Node.js development** workload. For instructions, see Prerequisites.

   Visual Studio creates the new solution and opens your project in the right pane. The *app.js* project file opens in the editor (left pane).



(1) Highlighted in **bold** is your project, using the name you gave in the **New Project** dialog box. In the file system, this project is represented by a *.njsproj* file in your project folder. You can set properties and environment variables associated with the project by right-clicking the project and choosing **Properties**. You can do round-tripping with other development tools, because the project file doesn't make custom changes to the Node.js project source.

(2) At the top level is a solution, which by default has the same name as your project. A solution, represented by a *.sln* file on disk, is a container for one or more related projects.

(3) The npm node shows any installed npm packages. You can right-click the npm node to search for and install npm packages by using a dialog box. You can also install and update packages by using the settings in *package.json* or the right-click options in the npm node.

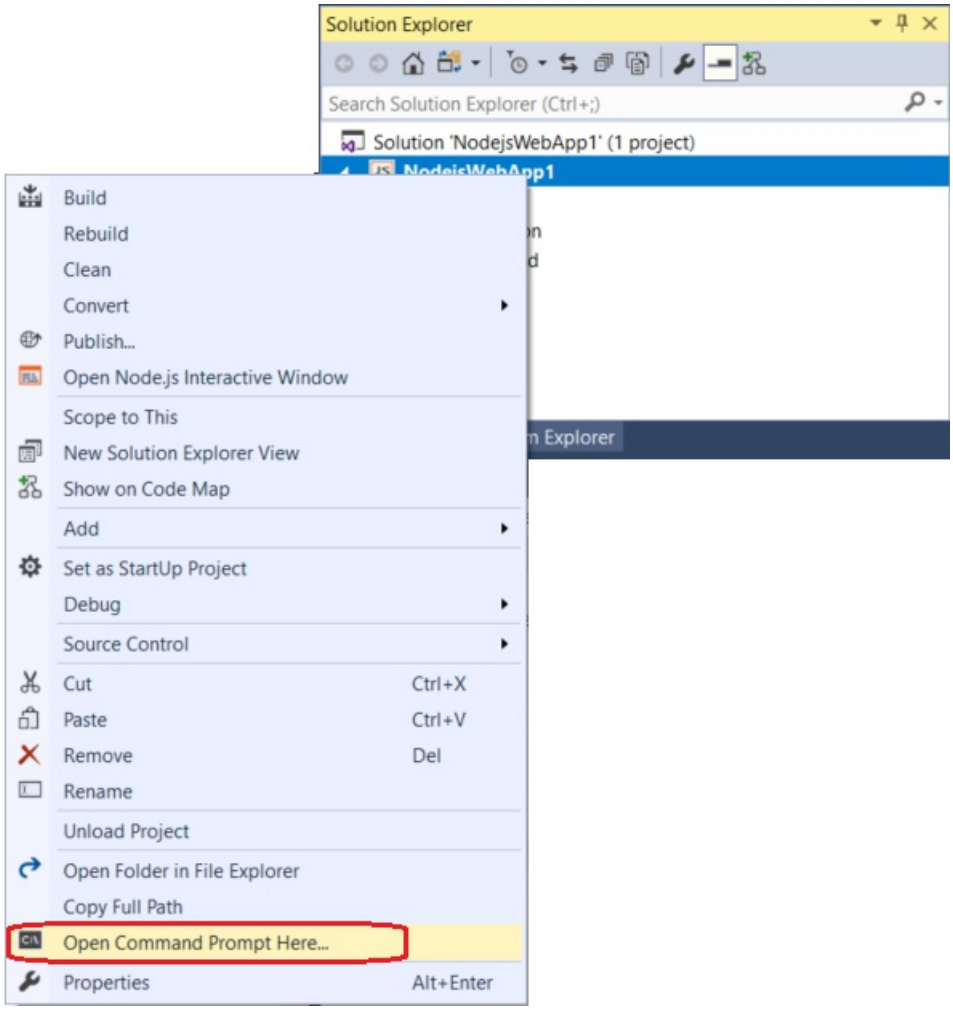
(4) *package.json* is a file used by npm to manage package dependencies and package versions for locally installed packages. For more information, see Manage npm packages.

(5) Project files such as *app.js* show up under the project node. *app.js* is the project startup file and that's why it's **bold**. You can set the startup file by right-clicking a file in the project and selecting **Set as Node.js startup file**.

3. Open the **npm** node and make sure that all the required npm packages are present.

   If a package is missing, its icon contains an exclamation point. To install any missing packages, you can right-click the **npm** node and select **Install npm Packages**.

   To install npm packages or Node.js commands from a command prompt, right-click the project node and select **Open Command Prompt Here** from the context menu.



Add some code

The application uses Pug for the front-end JavaScript framework. Pug uses simple markup code that compiles to HTML.

Pug is set as the view engine in *app.js*, with the code `app.set('view engine', 'pug');`.

1. In **Solution Explorer**, open the **views** folder, and then select **index.pug** to open the file.
2. Replace the file contents with the following markup.

```
extends layout

block content
  h1= title
  p Welcome to #{title}
  script.
    var f1 = function() { document.getElementById('myImage').src='#{data.item1}' }
  script.
    var f2 = function() { document.getElementById('myImage').src='#{data.item2}' }
  script.
    var f3 = function() { document.getElementById('myImage').src='#{data.item3}' }

  button(onclick='f1()') One!
  button(onclick='f2()') Two!
  button(onclick='f3()') Three!
  p
  a: img(id='myImage' height='300' width='300' src='')
```

   The preceding code dynamically generates an HTML page with a title and welcome message. The page also includes code to display an image that changes whenever you select a button.

3. In the **routes** folder, open **index.js**.
4. Add the following code before the `router.get` function call:

```
var getData = function () {
    var data = {
        'item1': 'https://images.unsplash.com/photo-1563422156298-c778a278f9a5',
        'item2': 'https://images.unsplash.com/photo-1620173834206-c029bf322dba',
        'item3': 'https://images.unsplash.com/photo-1602491673980-73aa38de027a'
    }
    return data;
}
```

   This code creates a data object that you pass to the dynamically generated HTML page.

5. Replace the `router.get` function call with the following code:

```
router.get('/', function (req, res) {
    res.render('index', { title: 'Express', "data" });
});
```

   The preceding code sets the current page using the Express router object and renders the page, passing the title and data object to the page. The code specifies the *index.pug* file as the page to load when *index.js* runs. The *app.js* code, not shown here, configures *index.js* as the default route.

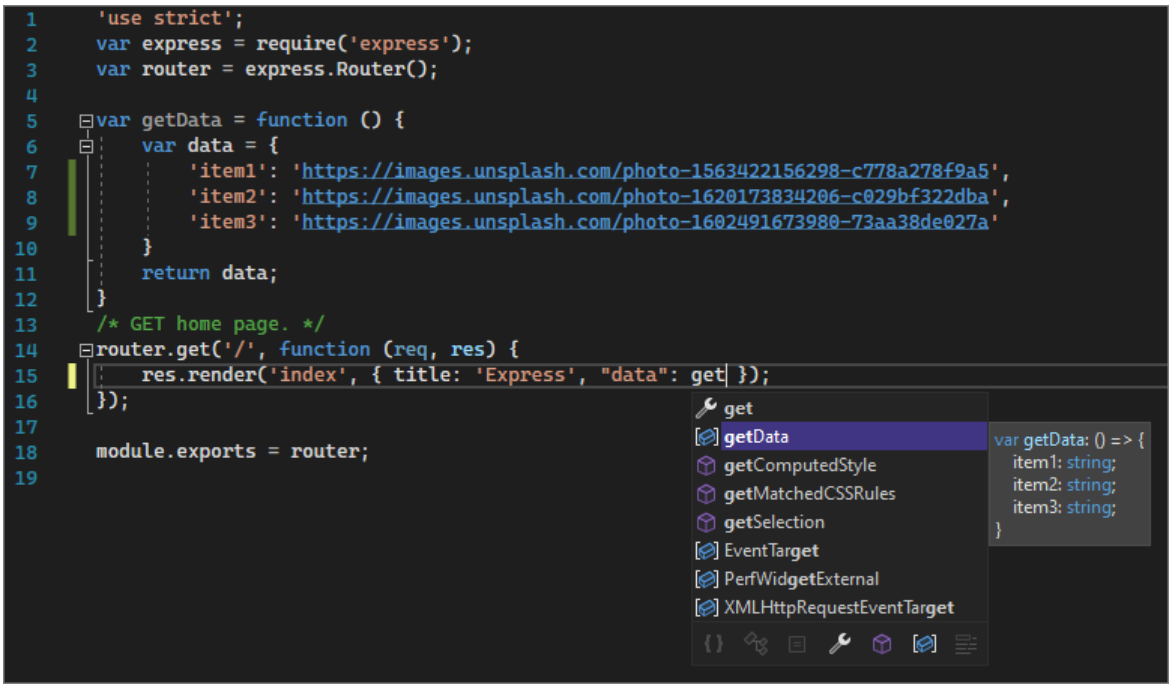
   To demonstrate several Visual Studio features, there's a deliberate error in the line of code that contains `res.render`. In the next section, IntelliSense helps you fix the error so the app can run.

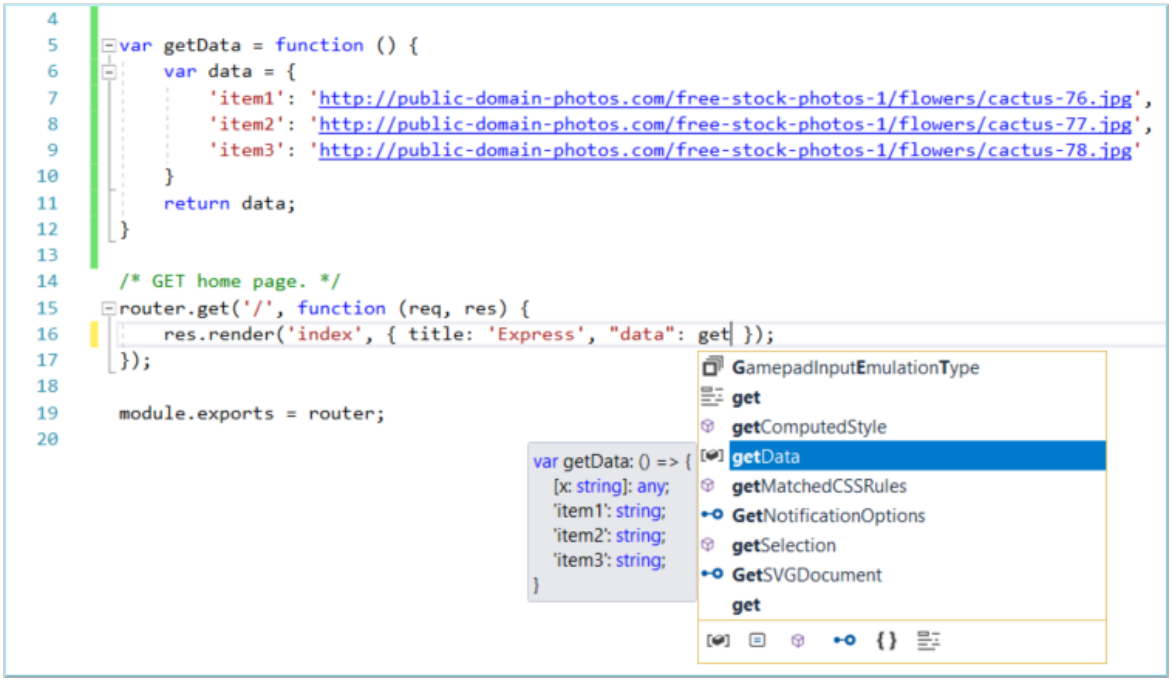
Use IntelliSense

IntelliSense is a Visual Studio tool that helps you as you write code.

1. In *index.js* in the Visual Studio code editor, go to the line of code that contains `res.render`.

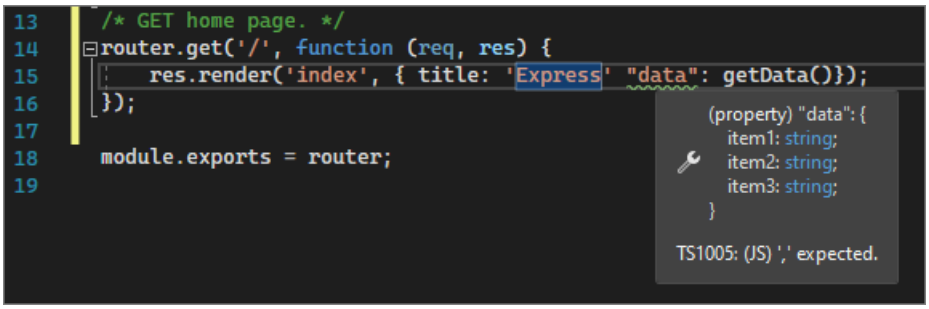
2. Put your cursor after the `"data"` string and type `: get`. IntelliSense displays the `getData` function you defined earlier in the code. Select `getData`.





3. Add parentheses to make the code a function call: `getData()`.

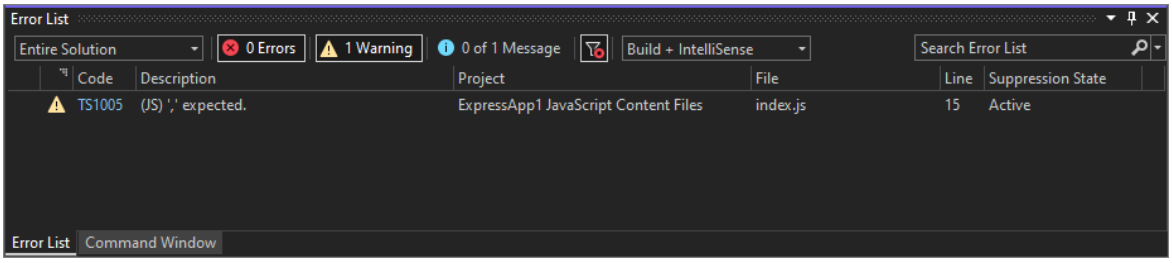
4. Remove the comma before `"data"`. Green syntax highlighting appears on the expression. Hover over the syntax highlighting.

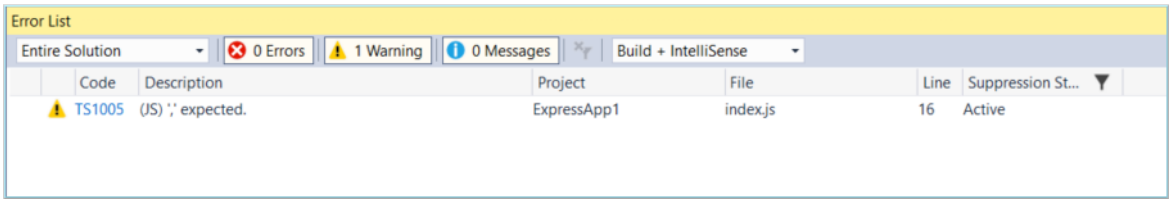


The last line of the message tells you that the JavaScript interpreter expected a comma.

5. In the lower pane, select the **Error List** tab, and select **Build + IntelliSense** from the dropdown list for the type of issues reported.

   The pane displays the warning and description along with the filename and line number.





6. Fix the code by replacing the comma before `"data"`.

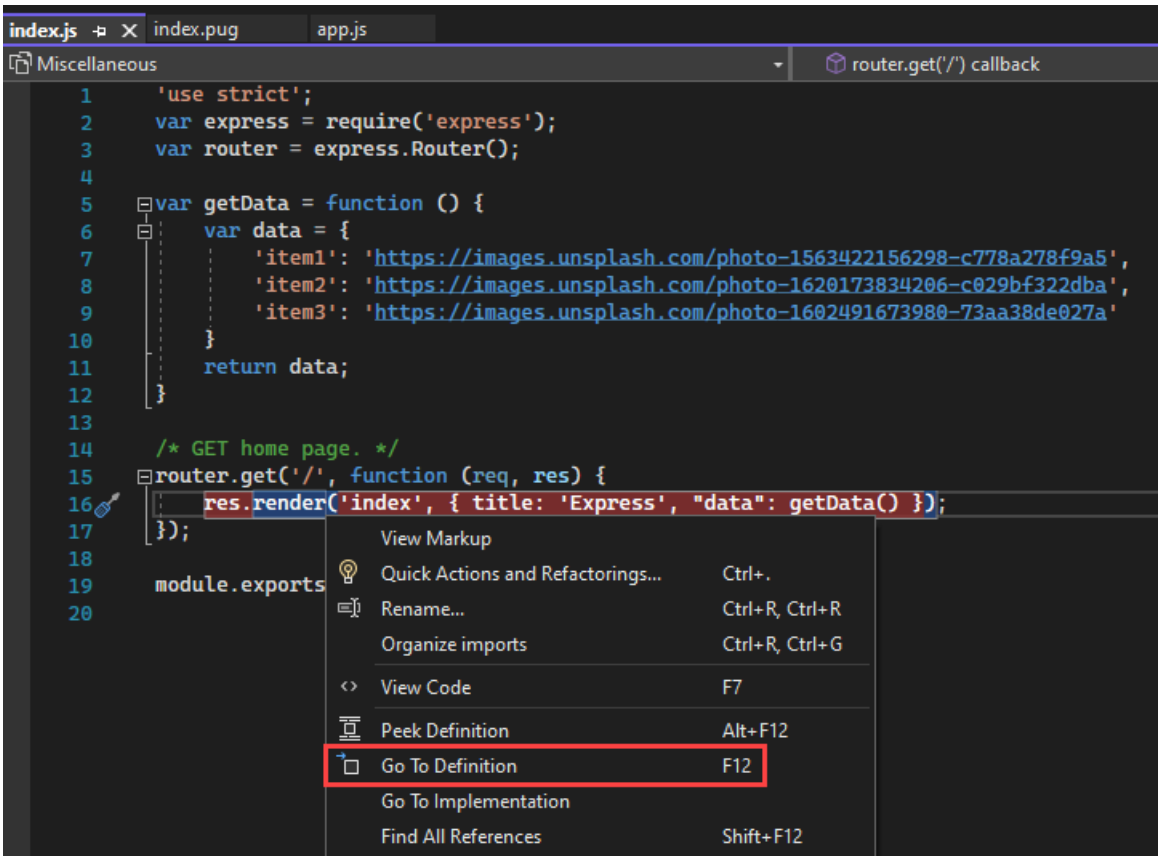
   The corrected line of code should look like this:

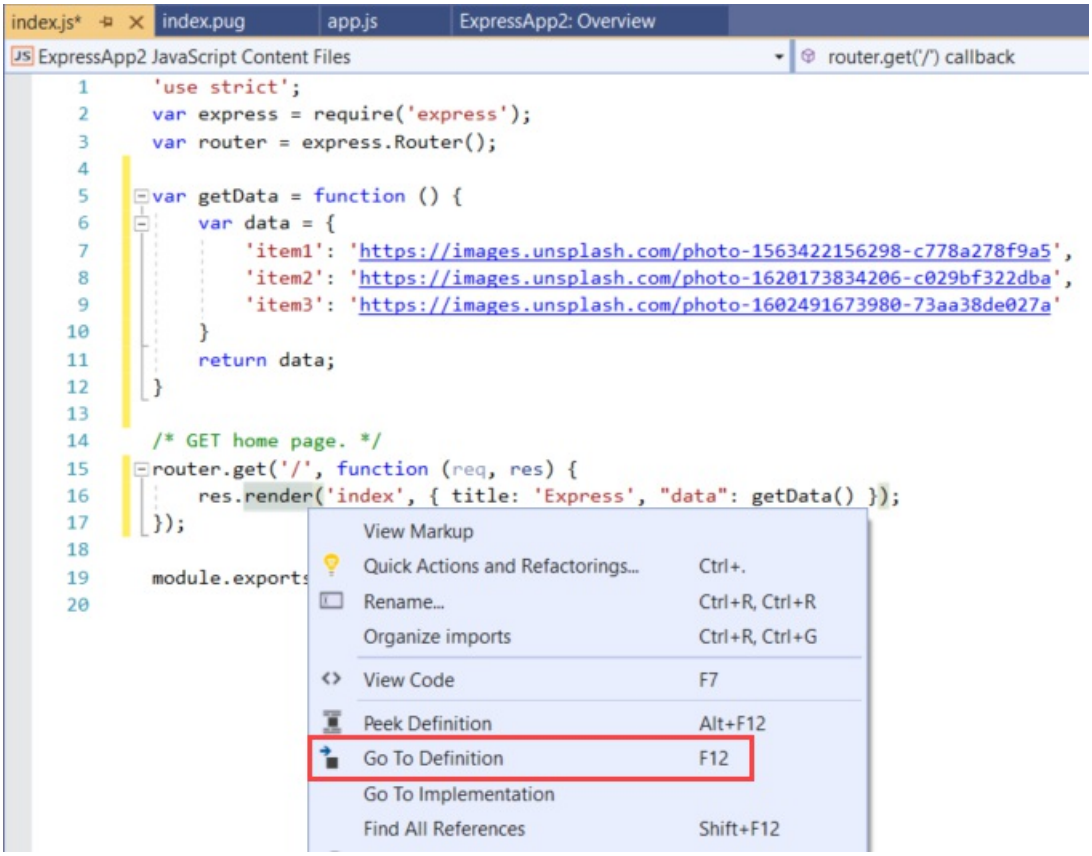
   ```
   res.render('index', { title: 'Express', "data": getData() });
   ```

7. If you'd like to navigate to the source code of `render`, use one of these options:

   - Select `render` and select **F12**.
   - Right-click `render` and select **Go To Definition** from the context menu.

   These commands take you to the definition of the `render` function in *index.d.ts*.





Run the app

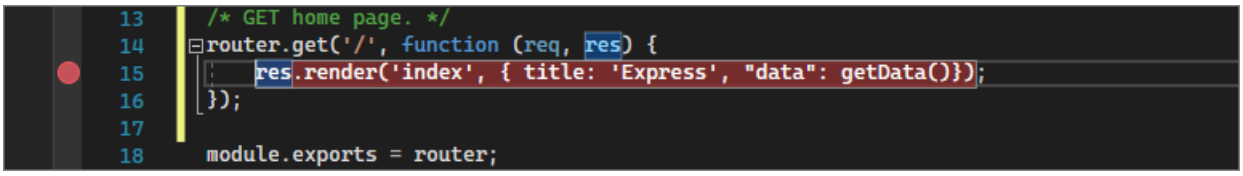
Next, run the app with the Visual Studio debugger attached. Before you do that, you need to set a breakpoint.

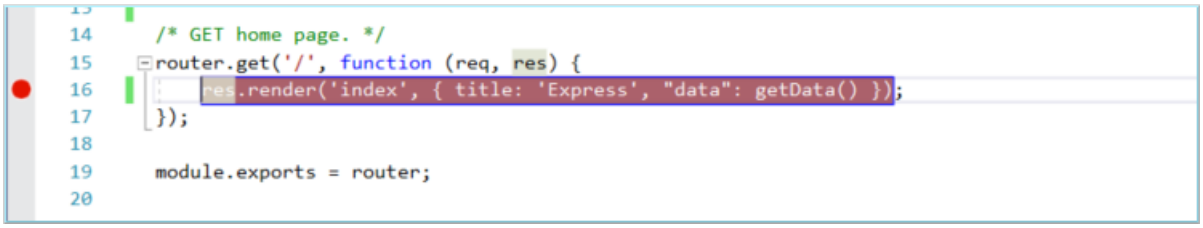
Set a breakpoint

Breakpoints are the most basic and essential feature of reliable debugging. A breakpoint indicates where Visual Studio should suspend your running code. You can then observe variable values, memory behavior, or whether a branch of code is running.

To set a breakpoint, in *index.js*, select the left gutter before the following line of code:

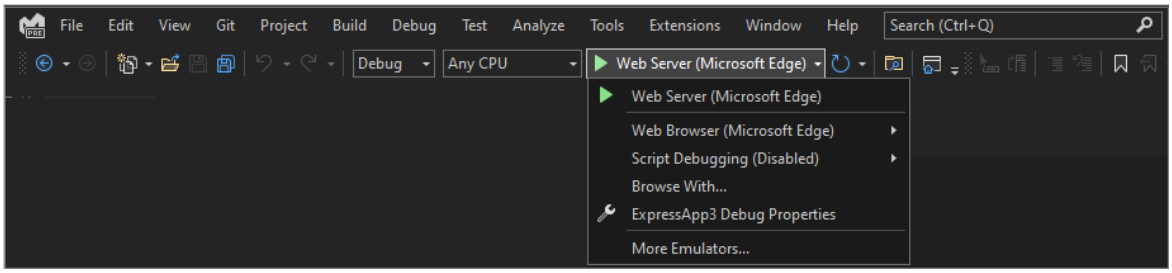
```
res.render('index', { title: 'Express', "data": getData() });
```

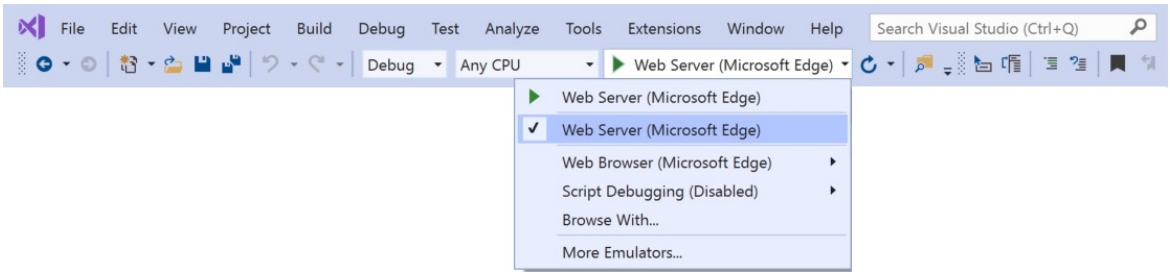




Run the app in Debug mode

1. Select the debug target in the **Debug** toolbar, such as **Web Server (Google Chrome)** or **Web Server (Microsoft Edge)**.



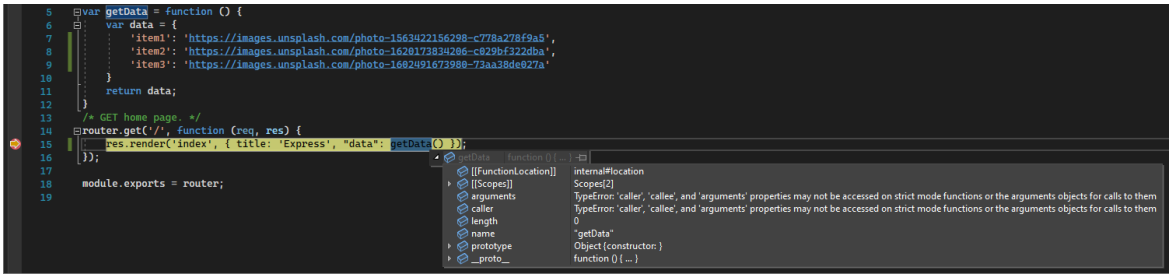


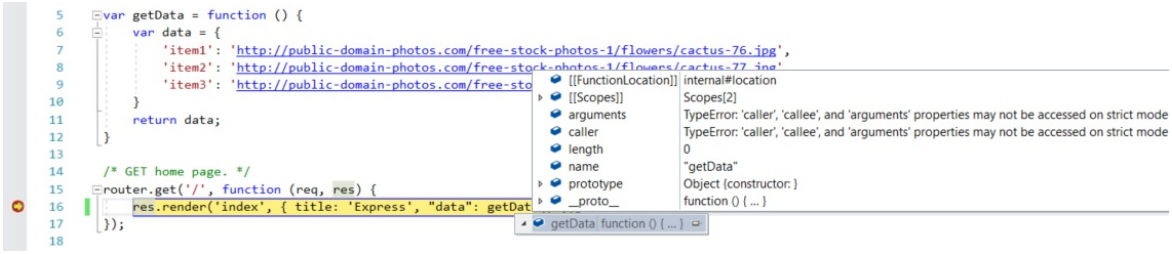
   If you know your preferred debug target is available on your machine, but it doesn't appear as an option, select **Browse With** from the debug target dropdown list. Select your default browser target in the list, and select **Set as Default**.

2. Select **F5** or select **Debug** > **Start Debugging** to run the app.

   The debugger pauses at the breakpoint you set, so you can inspect your app state.

3. Hover over `getData` to see its properties in a DataTip:

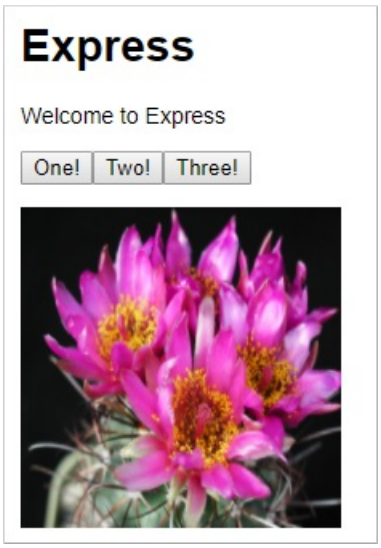




4. Select **F5** or select **Debug** > **Continue** to continue running the app.

   The app opens in a browser. In the browser window, you should see **Express** as the title and **Welcome to Express** as the first paragraph.

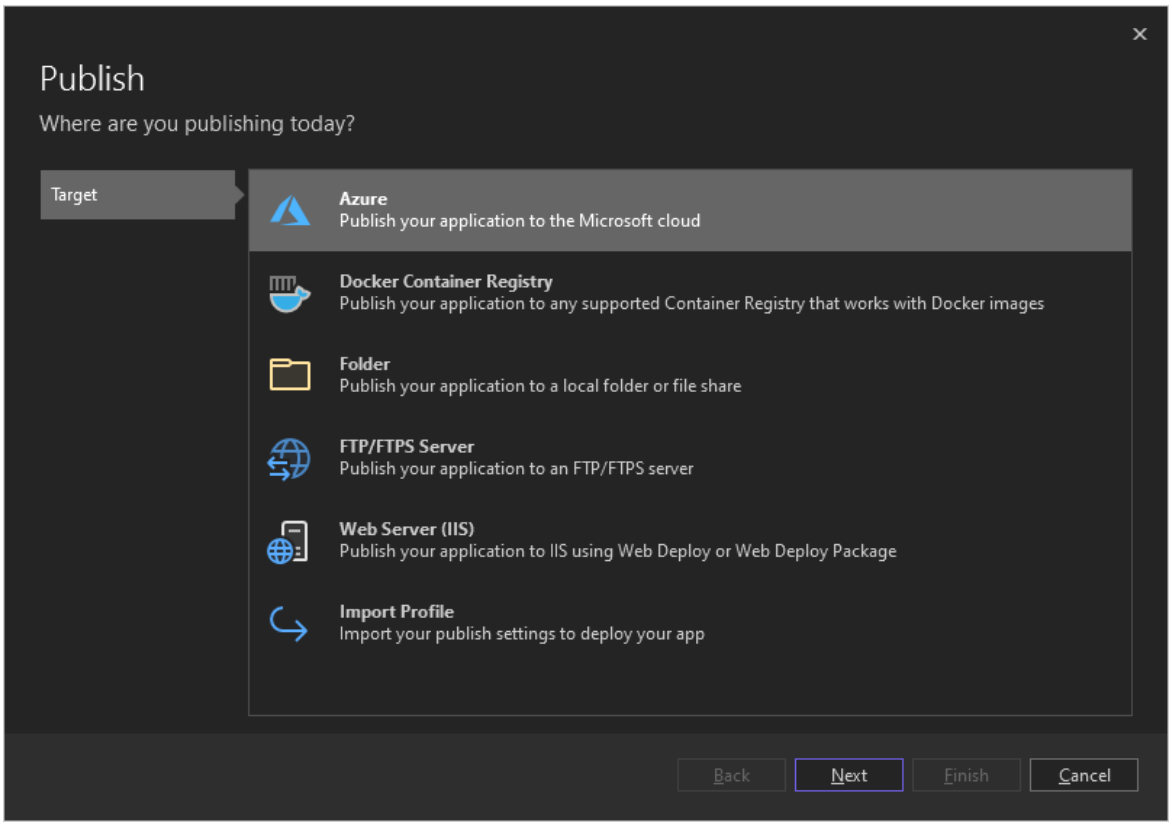
5. Select the **One!**, **Two!**, and **Three!** buttons to display different images.



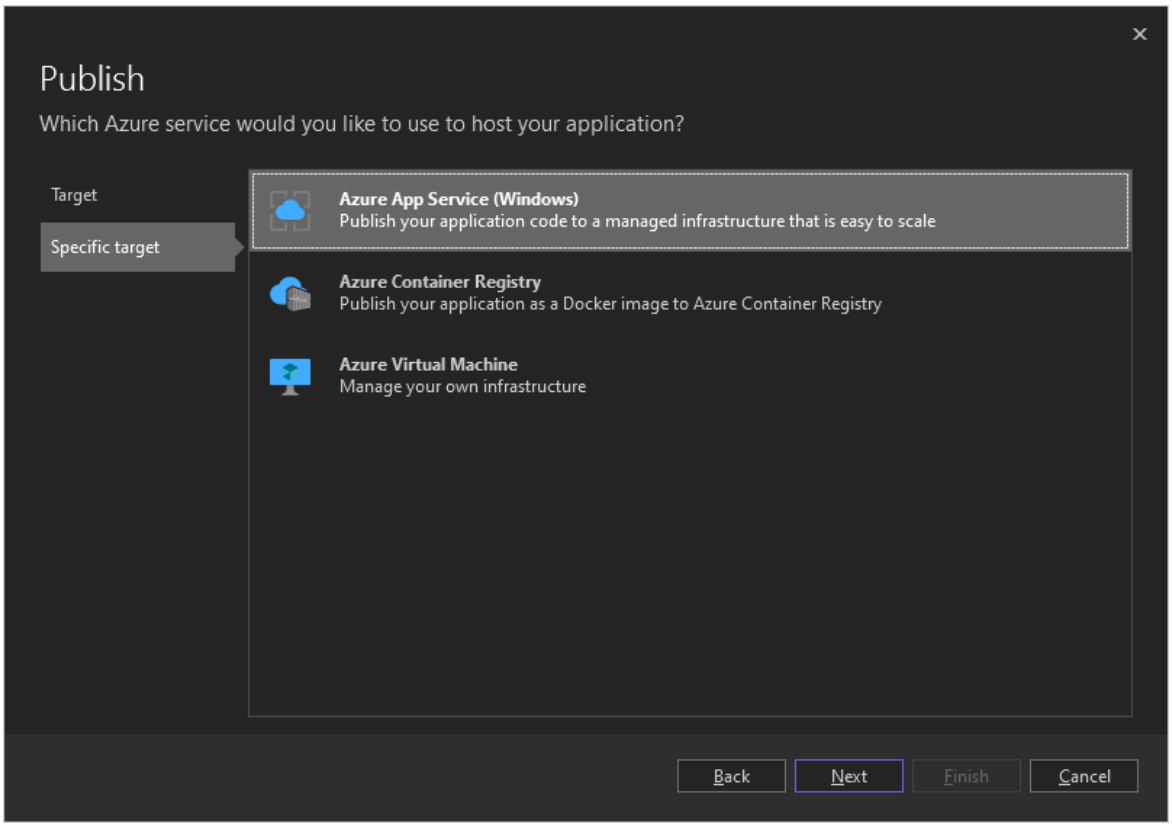
6. Close the web browser.

Publish to Azure App Service (optional)

1. In **Solution Explorer**, right-click the project and select **Publish**.
   - If prompted, select **Add a publish profile**.
   - If you're prompted to install Azure WebJob Tools, select **Install**.
2. On the first **Publish** screen, select **Azure**, and then select **Next**.

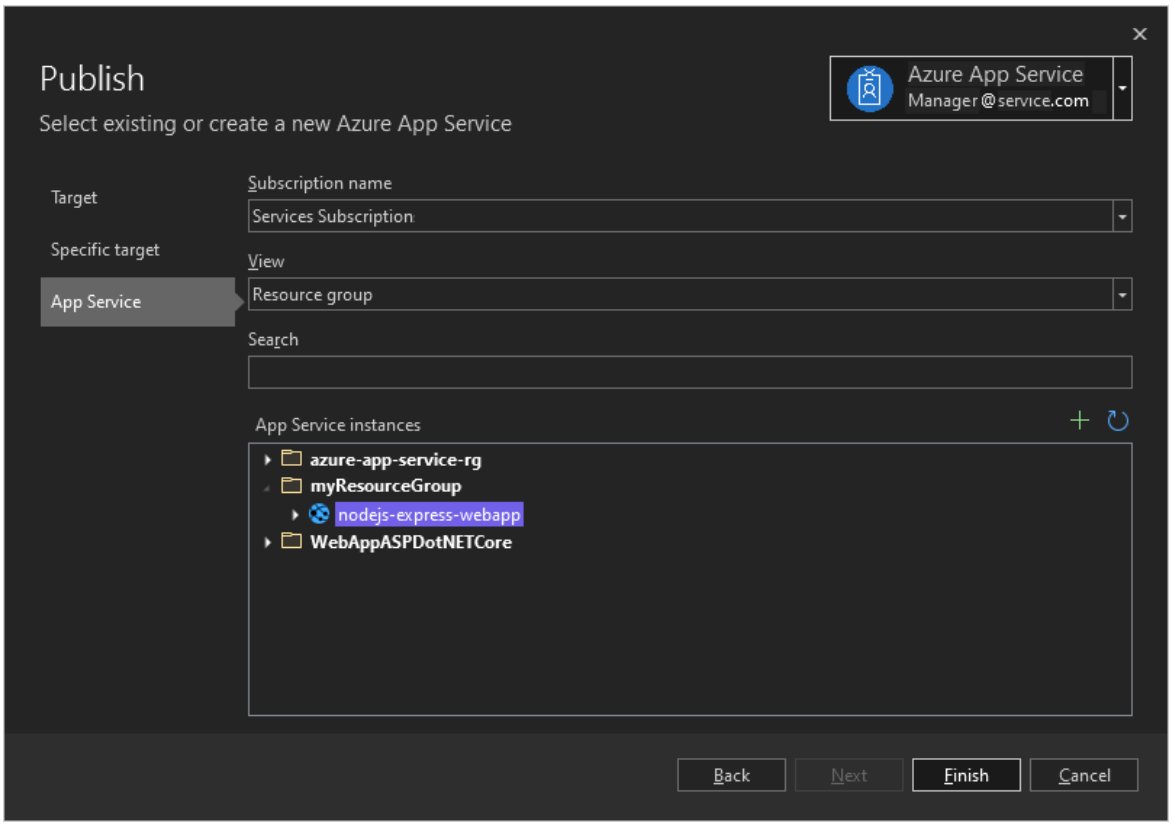


3. On the second **Publish** screen, select **Azure App Service (Windows)**, and then select **Next**.



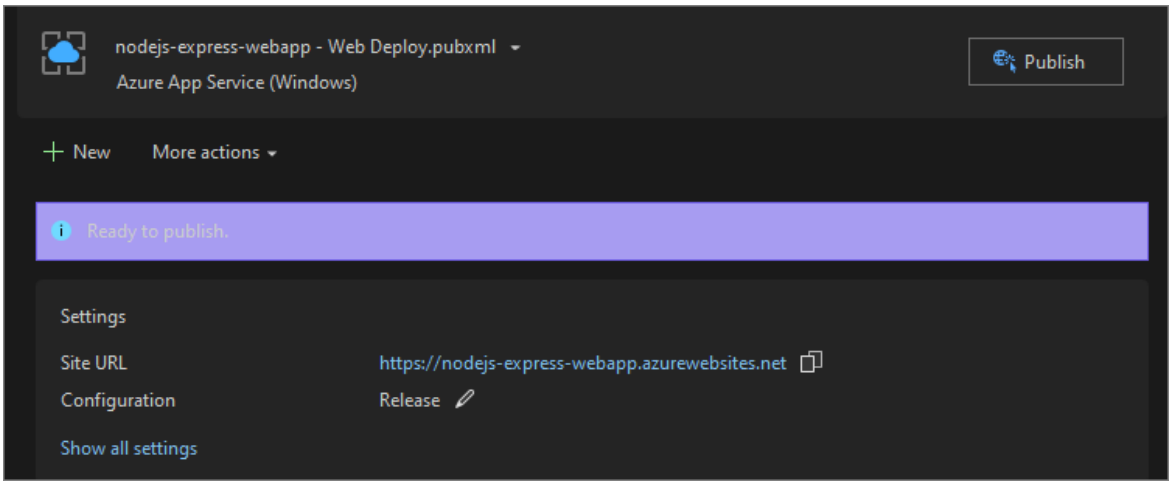
4. On the next screen, sign in to Azure if necessary. Select the Azure subscription, resource group, and App Service you want to publish to, and then select **Finish**.

   If you don't have an Azure subscription, resource group, or App Service, you can create them by following the prompts on this screen.



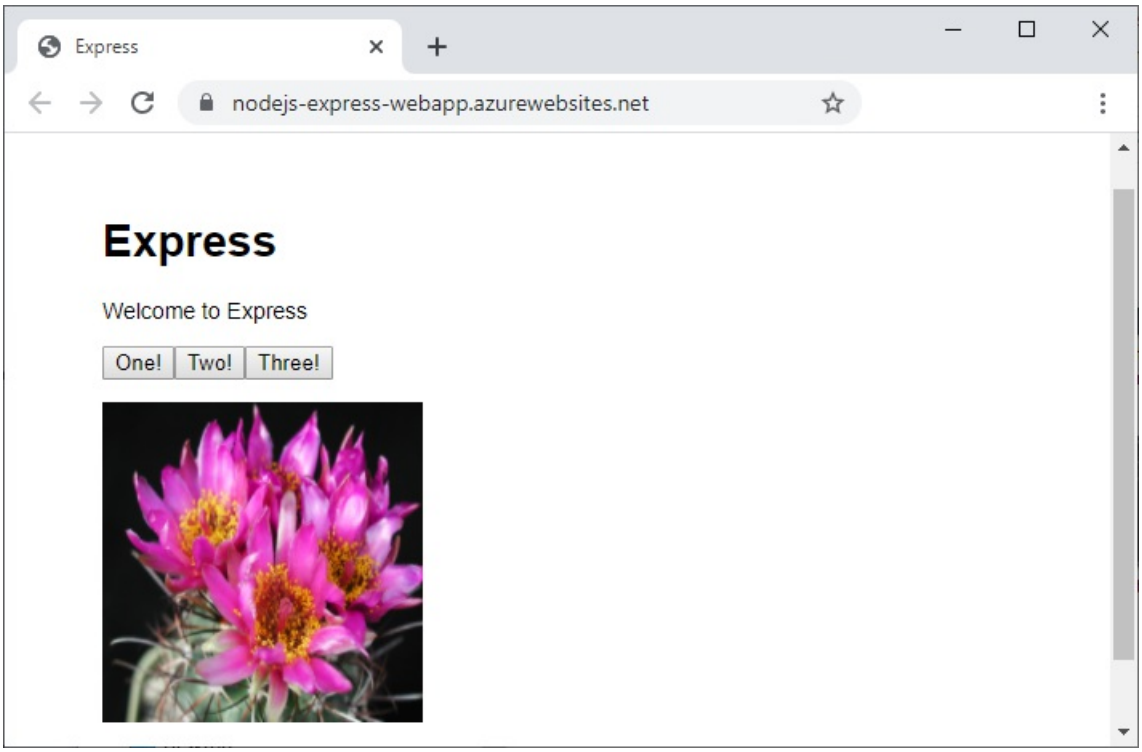
For more detailed instructions, see Publish to Azure website using web deploy.

5. Look over the publishing configuration, and then select **Publish**.

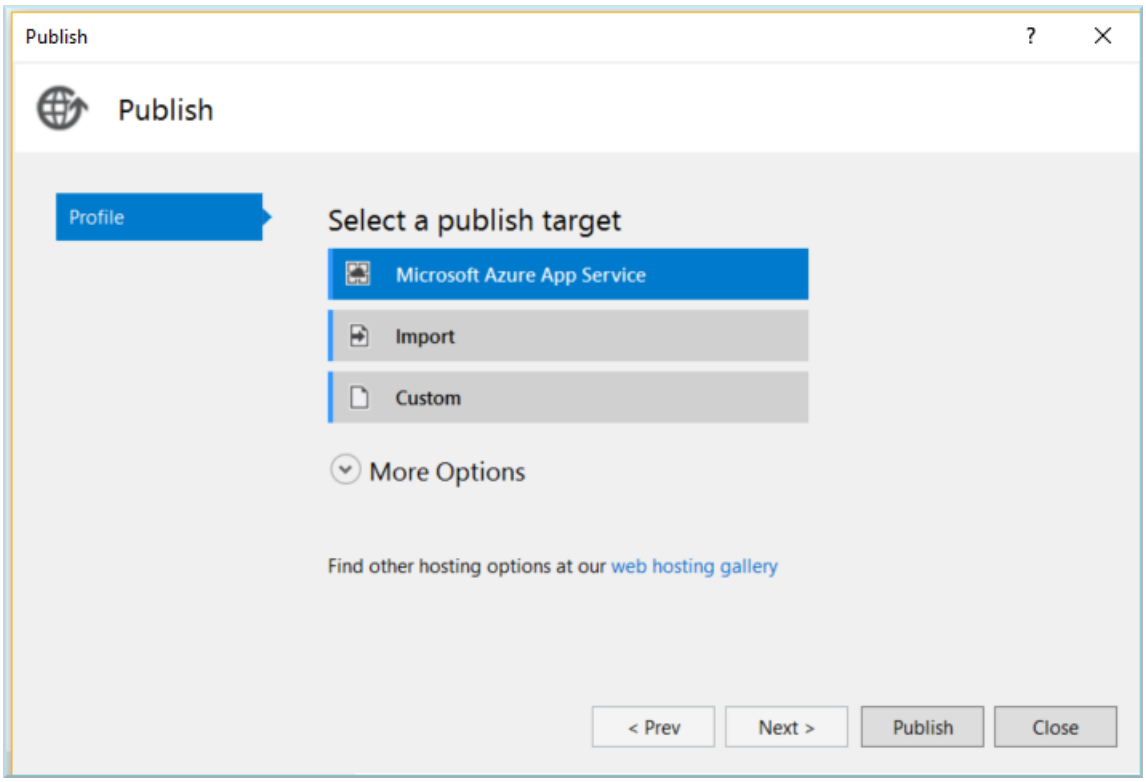


The Visual Studio **Output** window shows the Azure deployment progress.

6. On successful deployment, your app opens running in Azure App Service in a browser. Select a button to display an image.



1. In Solution Explorer, right-click the project and select **Publish**.



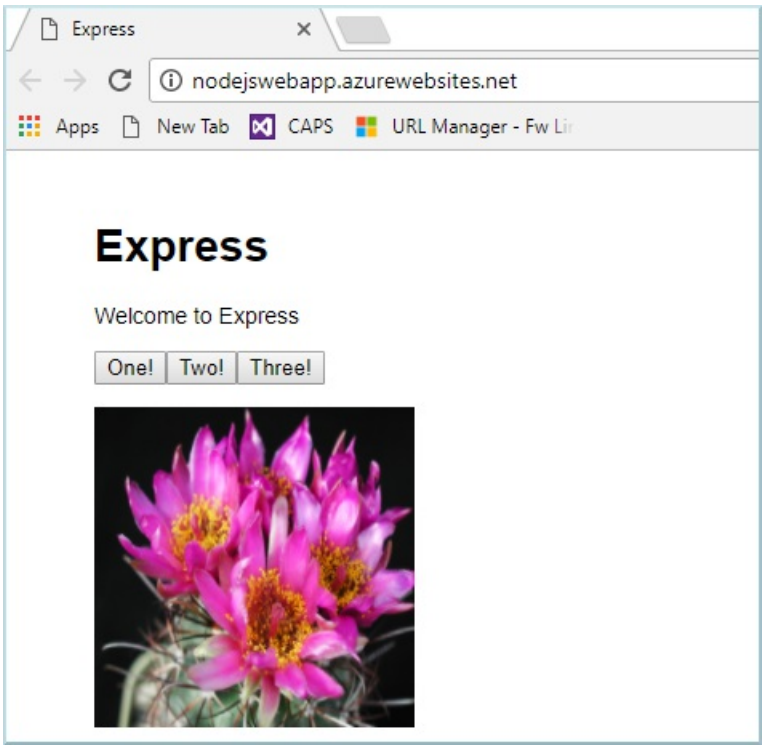
2. Select **Microsoft Azure App Service**.

   In the **App Service** dialog box, you can sign into your Azure account and connect to existing Azure subscriptions.

3. Follow the remaining steps to select a subscription, select or create a resource group, and select or create an app service plan. When prompted, follow the steps to publish to Azure. For more detailed instructions, see Publish to Azure website using web deploy.

4. The **Output** window shows the Azure deployment progress.

   On successful deployment, your app opens in a browser running in Azure App Service. Select a button to display an image.



Congratulations on completing this tutorial!

Next steps

Deploy the app to Linux App Service

AngularJS language service extension

Tutorial: Create a Node.js and React app in Visual Studio

6/24/2022 • 17 minutes to read • Edit Online

Applies to: Visual Studio Visual Studio for Mac Visual Studio Code

With Visual Studio, you can easily create a Node.js project and use IntelliSense and other built-in features that support Node.js. In this tutorial, you create a Node.js web app project from a Visual Studio template. Then, you create a simple app using React.

In this tutorial, you learn how to:

- Create a Node.js project
- Add npm packages
- Add React code to your app
- Transpile JSX
- Attach the debugger

> **IMPORTANT**
> Starting in Visual Studio 2022, you can alternatively create a React project using the new CLI-based project type. Some of the information in this article applies only to the Node.js project type (.njsproj).

Before you begin, here's a quick FAQ to introduce you to some key concepts:

- **What is Node.js?**

  Node.js is a server-side JavaScript runtime environment that executes JavaScript code.

- **What is npm?**

  The default package manager for Node.js is npm. A package manager makes it easier to publish and share Node.js source code libraries. The npm package manager simplifies library installation, updating, and uninstallation.

- **What is React?**

  React is a front-end framework for creating a user interface (UI).

- **What is JSX?**

  JSX is a JavaScript syntax extension typically used with React to describe UI elements. You must transpile JSX code to plain JavaScript before it can run in a browser.

- **What is webpack?**

  Webpack bundles JavaScript files so they can run in a browser, and can also transform or package other resources and assets. Webpack can specify a compiler, such as Babel or TypeScript, to transpile JSX or TypeScript code to plain JavaScript.

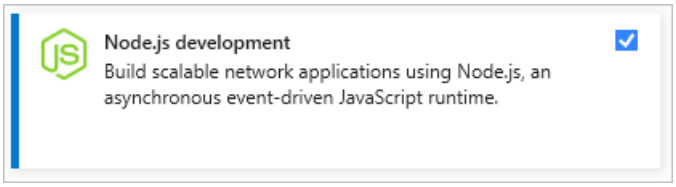
Prerequisites

This tutorial requires the following prerequisites:

- Visual Studio with the Node.js development workload installed.

  If you haven't yet installed Visual Studio:

  1. Go to the Visual Studio downloads page to install Visual Studio for free.
  2. In the Visual Studio Installer, select the **Node.js development** workload, and select **Install**.

  If you have Visual Studio installed but need the Node.js workload:

  1. In Visual Studio, go to **Tools** > **Get Tools and Features**.
  2. In the Visual Studio Installer, choose the **Node.js development** workload, and select **Modify** to download and install the workload.

- The Node.js runtime installed:

  If you don't have the Node.js runtime installed, install the LTS version from the Node.js website. The LTS version has the best compatibility with other frameworks and libraries.

  The Node.js tools in the Visual Studio Node.js workload support both Node.js 32-bit and 64-bit architecture versions. Visual Studio requires only one version, and the Node.js installer only supports one version at a time.

  Visual Studio usually detects the installed Node.js runtime automatically. If not, you can configure your project to reference the installed runtime:

  1. After you create a project, right-click the project node and select **Properties**.
  2. In the **Properties** pane, set the **Node.exe path** to reference a global or local installation of Node.js. You can specify the path to a local interpreter in each of your Node.js projects.

This tutorial was tested with Node.js 14.17.5.

This tutorial was tested with Node.js 12.6.2.

Create a project

First, create a Node.js web app project.

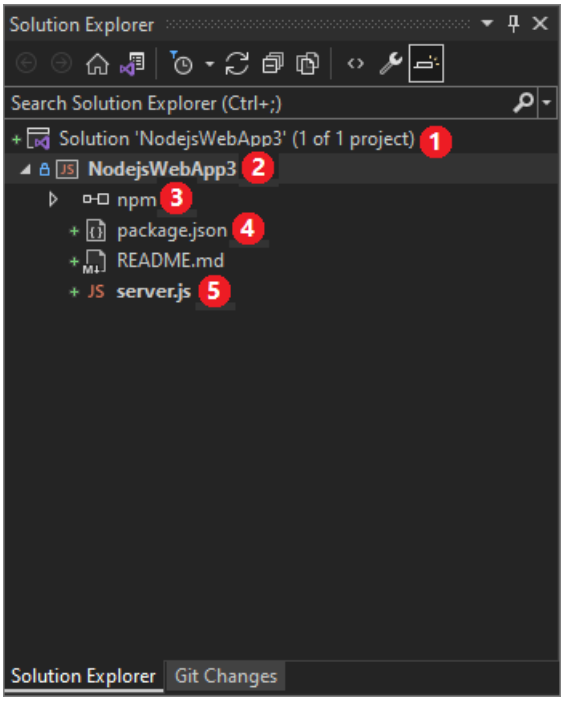
1. Open Visual Studio, and press **Esc** to close the start window.
2. Press **Ctrl+Q**, type *node.js* in the search box, and then choose **Blank Node.js Web Application - JavaScript** from the dropdown list.

   Although this tutorial uses the TypeScript compiler, the steps require that you start with the **JavaScript** template.

   If you don't see the **Blank Node.js Web Application** choice, you need to install the Node.js development workload. For instructions, see the Prerequisites.
3. In the **Configure your new project** dialog box, select **Create**.

   Visual Studio creates the new solution and project, and opens the project in the right pane. The *server.js* project file opens in the editor in the left pane.

4. Look at the project structure in **Solution Explorer** in the right pane.



   - At the top level is the *solution* (**1**), which by default has the same name as your project. A solution, represented by a *.sln* file on disk, is a container for one or more related projects.
   - Your project (**2**), using the name you gave in the **Configure your new project** dialog box, is highlighted in bold. In the file system, the project is a *.njsproj* file in your project folder.

     To see and set project properties and environment variables, press **Alt+Enter**, or right-click the project and select **Properties** from the context menu. You can work with other development tools, because the project file doesn't make custom changes to the Node.js project source.
   - The **npm** node (**3**) shows any installed npm packages.

     Right-click the **npm** node to search for and install npm packages. You can install and update packages by using the settings in *package.json* and the right-click options in the **npm** node.
   - Npm uses the *package.json* file (**4**) to manage dependencies and versions for locally installed packages. For more information, see Manage npm packages.
   - Project files (**5**) appear under the project node. The project startup file, *server.js*, shows in bold.

     You can set the startup file by right-clicking a file in the project and selecting **Set as Node.js startup file**.

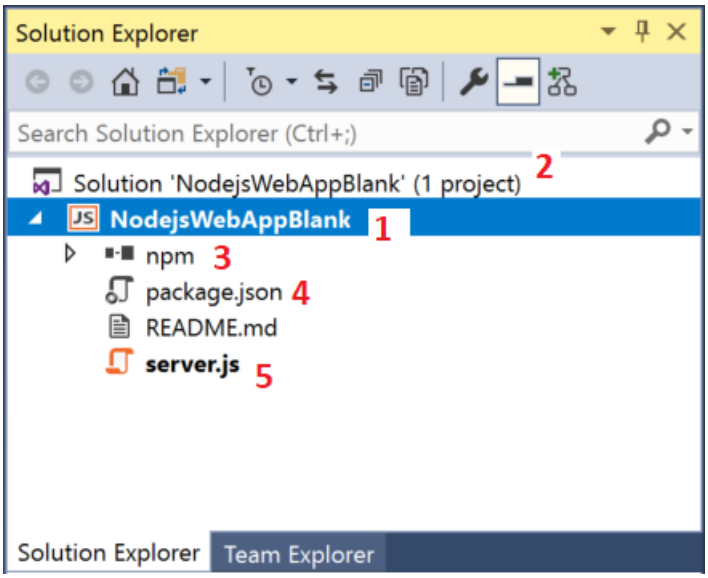
1. Open Visual Studio.
2. Create a new project.

   Press **Esc** to close the start window. Type **Ctrl + Q** to open the search box, type **Node.js**, then choose **Blank Node.js Web Application - JavaScript**. (Although this tutorial uses the TypeScript compiler, the steps require that you start with the **JavaScript** template.)

   In the dialog box that appears, choose **Create**.

   If you don't see the **Blank Node.js Web Application** project template, you must add the **Node.js development** workload. For detailed instructions, see the Prerequisites.

   Visual Studio creates the new solution and opens your project.



(1) Highlighted in **bold** is your project, using the name you gave in the **New Project** dialog box. In the file system, this project is represented by a *.njsproj* file in your project folder. You can set properties and environment variables associated with the project by right-clicking the project and choosing **Properties** (or press **Alt** + **Enter**). You can do round-tripping with other development tools, because the project file does not make custom changes to the Node.js project source.

(2) At the top level is a solution, which by default has the same name as your project. A solution, represented by a *.sln* file on disk, is a container for one or more related projects.

(3) The npm node shows any installed npm packages. You can right-click the npm node to search for and install npm packages using a dialog box or install and update packages using the settings in *package.json* and right-click options in the npm node.

(4) *package.json* is a file used by npm to manage package dependencies and package versions for locally installed packages. For more information, see Manage npm packages.

(5) Project files such as *server.js* show up under the project node. *server.js* is the project startup file and that is why it shows up in **bold**. You can set the startup file by right-clicking a file in the project and selecting **Set as Node.js startup file**.

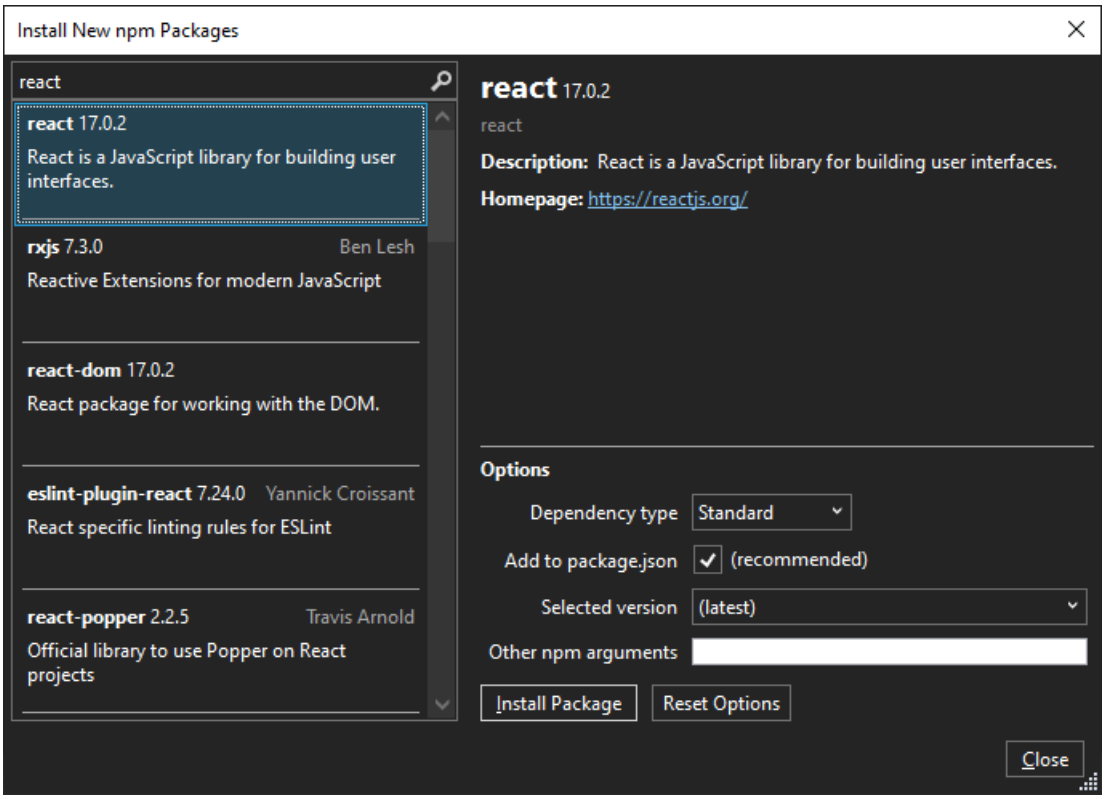
Add npm packages

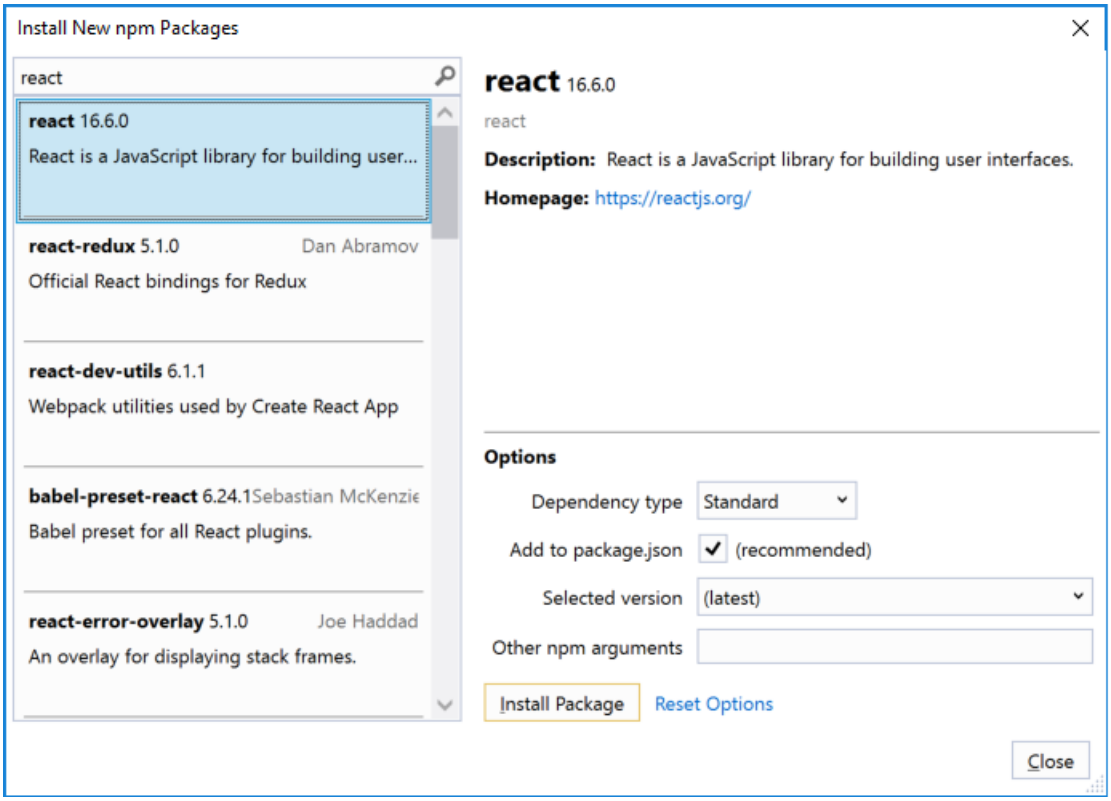
This app requires the following npm modules to run correctly:

- react
- react-dom
- express
- path
- ts-loader
- typescript
- webpack
- webpack-cli

To install a package:

1. In **Solution Explorer**, right-click the **npm** node and select **Install New npm Packages**.
2. In the **Install New npm Packages** dialog box, search for the **react** package, and select **Install Package** to install it.





In the **Install New npm Packages** dialog box, you can choose to install the most current package version or to specify a version. If you choose to install the current versions, but run into unexpected errors later, try installing the exact package versions listed in the next step.

The **Output** window in the Visual Studio lower pane shows package installation progress. Open the **Output** window by selecting **View** > **Output** or pressing **Ctrl**+**Alt**+**O**. In the **Show output from** field of the **Output** window, select **Npm**.

When installed, the **react** package appears under the **npm** node in **Solution Explorer**.

The project's *package.json* file updates with the new package information, including the package version.

Instead of using the UI to search for and add the rest of the packages one at a time, you can paste the required package code into *package.json*.

1. From **Solution Explorer**, open **package.json** in the Visual Studio editor. Add the following `dependencies` section before the end of the file:

   ```
   "dependencies": {
     "express": "~4.17.1",
     "path": "~0.12.7",
     "react": "~16.13.1",
     "react-dom": "~16.13.1",
     "ts-loader": "~7.0.1",
     "typescript": "~3.8.3",
     "webpack": "~4.42.1",
     "webpack-cli": "~3.3.11"
   }
   ```

   If the file already has a `dependencies` section, replace it with the preceding JSON code. For more information on using the *package.json* file, see package.json configuration.

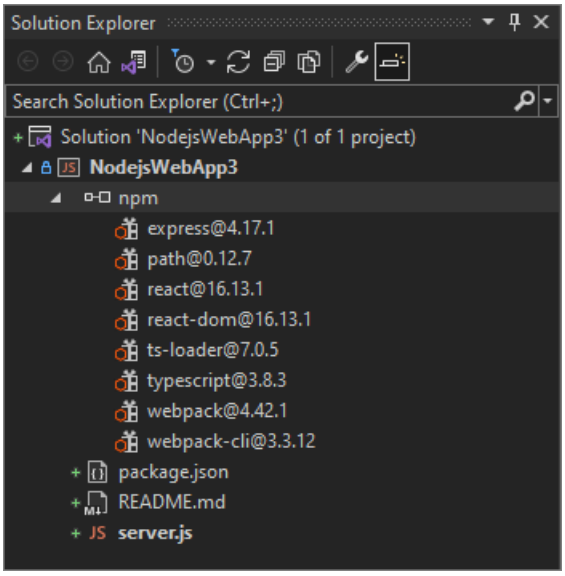
2. Press **Ctrl+S** or select **File** > **Save package.json** to save the changes.

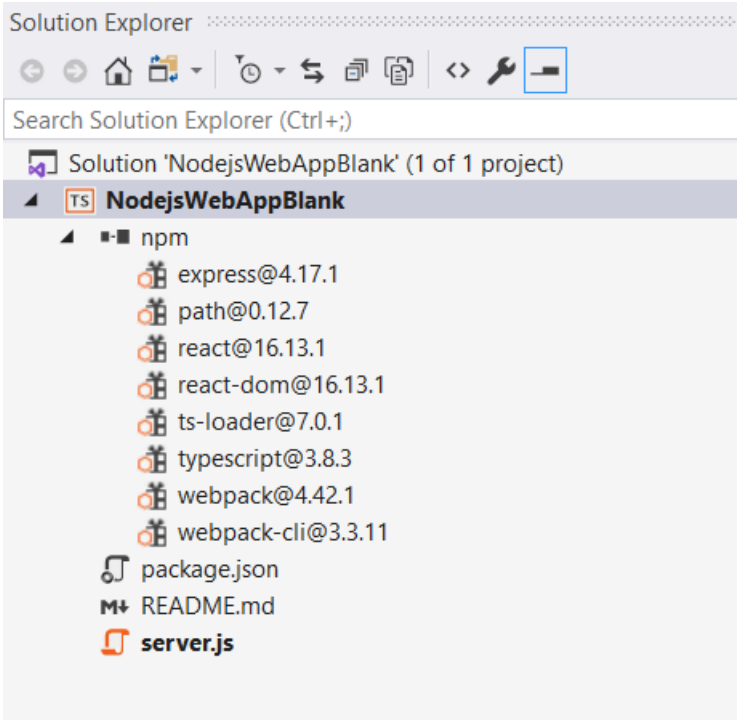
3. In **Solution Explorer**, right-click the **npm** node in your project and select **Install npm Packages**.

   This command runs the npm install command directly to install all the packages listed in *packages.json*.

   Select the **Output** window in the lower pane to see installation progress. Installation might take a few minutes, and you might not see results immediately. Make sure that you select **Npm** in the **Show output from** field in the **Output** window.

   After installation, the npm modules appear in the **npm** node in **Solution Explorer**.





> **NOTE**
>
> You can also install npm packages by using the command line. In **Solution Explorer**, right-click the project name and select **Open Command Prompt Here**. Use standard Node.js commands to install packages.

Add project files

Next, add four new files to your project.

- *app.tsx*
- *webpack-config.js*
- *index.html*
- *tsconfig.json*

For this simple app, you add the new project files in the project root. For most apps, you add the files to subfolders and adjust relative path references accordingly.

1. In **Solution Explorer**, select the project name and press **Ctrl**+**Shift**+**A**, or right-click the project name and select **Add** > **New Item**.
2. In the **Add New Item** dialog box, choose **TypeScript JSX File**, type the name *app.tsx*, and select **Add** or **OK**.
3. Repeat these steps to add a **JavaScript file** named *webpack-config.js*.
4. Repeat these steps to add an **HTML file** named *index.html*.
5. Repeat these steps to add a **TypeScript JSON Configuration File** named *tsconfig.json*.

Add app code

1. In **Solution Explorer**, open **server.js** and replace the existing code with the following code:

```
'use strict';
var path = require('path');
var express = require('express');

var app = express();

var staticPath = path.join(__dirname, '/');
app.use(express.static(staticPath));

// Allows you to set port in the project properties.
app.set('port', process.env.PORT || 3000);

var server = app.listen(app.get('port'), function() {
    console.log('listening');
});
```

The preceding code uses Express to start Node.js as your web application server. The code sets the port to the port number configured in the project properties, which by default is 1337. If you need to open the project properties, right-click the project name in **Solution Explorer** and select **Properties**.

2. Open **app.tsx** and add the following code:

```
declare var require: any

var React = require('react');
var ReactDOM = require('react-dom');

export class Hello extends React.Component {
    render() {
        return (
            <h1>Welcome to React!!</h1>
        );
    }
}

ReactDOM.render(<Hello />, document.getElementById('root'));
```

The preceding code uses JSX syntax and React to display a message.

3. Open **index.html** and replace the `body` section with the following code:

```
<body>
    <div id="root"></div>
    <!-- scripts -->
    <script src="./dist/app-bundle.js"></script>
</body>
```

This HTML page loads *app-bundle.js*, which contains the JSX and React code transpiled to plain JavaScript. Currently, *app-bundle.js* is an empty file. In the next section, you configure options to transpile the code.

Configure webpack and TypeScript compiler options

Next, you add webpack configuration code to *webpack-config.js*. You add a simple webpack configuration that specifies an input file, *app.tsx*, and an output file, *app-bundle.js*, for bundling and transpiling JSX to plain JavaScript. For transpiling, you also configure some TypeScript compiler options. This basic configuration code is an introduction to webpack and the TypeScript compiler.

1. In **Solution Explorer**, open **webpack-config.js** and add the following code.

```
module.exports = {
    devtool: 'source-map',
    entry: "./app.tsx",
    mode: "development",
    output: {
        filename: "./app-bundle.js"
    },
    resolve: {
        extensions: ['.Webpack.js', '.web.js', '.ts', '.js', '.jsx', '.tsx']
    },
    module: {
        rules: [
            {
                test: /\.tsx$/,
                exclude: /(node_modules|bower_components)/,
                use: {
                    loader: 'ts-loader'
                }
            }
        ]
    }
}
```

The webpack configuration code instructs webpack to use the TypeScript loader to transpile the JSX.

2. Open **tsconfig.json** and replace the contents with the following code, which specifies the TypeScript compiler options:

```
{
  "compilerOptions": {
    "noImplicitAny": false,
    "module": "commonjs",
    "noEmitOnError": true,
    "removeComments": false,
    "sourceMap": true,
    "target": "es5",
    "jsx": "react"
  },
  "exclude": [
    "node_modules"
  ],
  "files": [
    "app.tsx"
  ]
}
```

The code specifies *app.tsx* as the source file.

3. Press **Ctrl**+**Shift**+**S** or select **File** > **Save All** to save all changes.

Transpile the JSX

1. In **Solution Explorer**, right-click the project name and select **Open Command Prompt Here**.

2. In the command prompt, enter the following webpack command:

```
node_modules\.bin\webpack ./app.tsx --config webpack-config.js
```

The command prompt window shows the result.

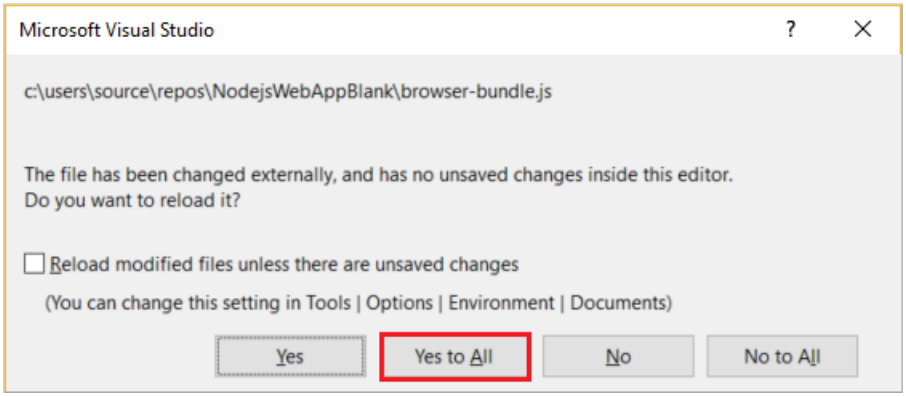
```
C:\Users\mik\source\repos\NodejsWebAppBlank\NodejsWebAppBlank>node_modules\.bin\webpack app.tsx --config webpack-conf
ig.js
Hash: 64dd2af4735c8b353923
Version: webpack 4.42.1
Time: 1460ms
Built at: 04/20/2020 12:27:51 PM
            Asset      Size  Chunks                    Chunk Names
    ./app-bundle.js   946 KiB    main  [emitted]         main
./app-bundle.js.map  1.08 MiB    main  [emitted] [dev]   main
Entrypoint main = ./app-bundle.js ./app-bundle.js.map
[./app.tsx] 1.18 KiB {main} [built]
    + 11 hidden modules

C:\Users\mik\source\repos\NodejsWebAppBlank\NodejsWebAppBlank>
```

If you see any errors instead of the preceding output, you must resolve them before your app will work. If your npm package versions are different than the versions this tutorial specifies, that can cause errors. One way to fix errors is to use the exact versions shown in the earlier step.

Also, if one or more package versions are deprecated and result in an error, you might need to install a more recent version to fix errors. For information on using *package.json* to control npm package versions, see package.json configuration.

3. In **Solution Explorer**, right-click the project node and select **Add** > **Existing Folder**.
4. Select the *dist* folder, and then select **Select Folder**.

   Visual Studio adds the *dist* folder, which contains *app-bundle.js* and *app-bundle.js.map*, to the project.
5. Open *app-bundle.js* to see the transpiled JavaScript code.
6. If prompted whether to reload externally modified files, select **Yes to All**.



Anytime you make changes to *app.tsx*, you must rerun the webpack command. To automate this step, you can add a build script to transpile the JSX.

Add a build script to transpile the JSX

Visual Studio versions starting with Visual Studio 2019 require a build script. Instead of transpiling JSX at the command line, as shown in the preceding section, you can transpile JSX when building from Visual Studio.

1. Open *package.json* and add the following section after the `dependencies` section:

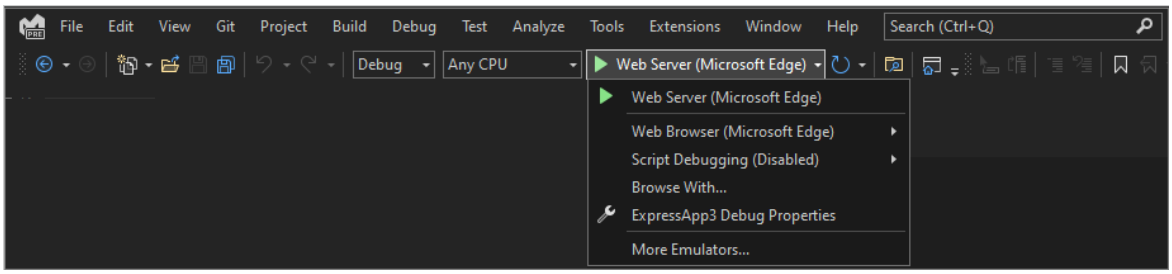
```
"scripts": {
 "build": "webpack-cli ./app.tsx --config webpack-config.js"
}
```

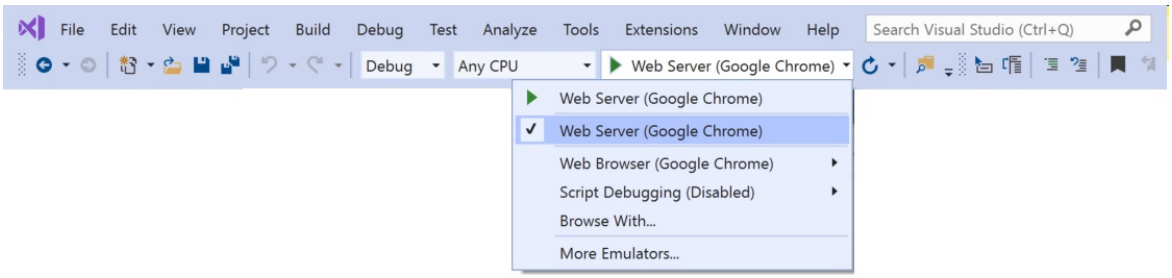
2. Save your changes.

Run the app

1. In the **Debug** toolbar, select either **Web Server (Microsoft Edge)** or **Web Server (Google Chrome)**

as the debug target.



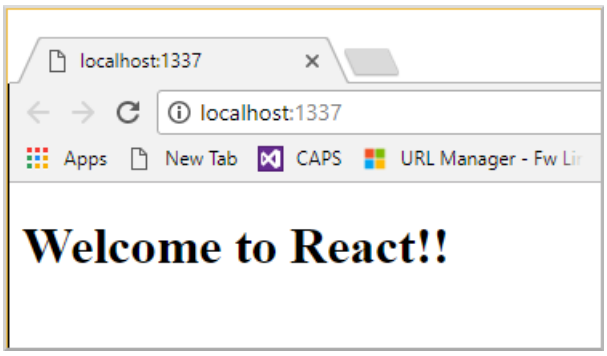


If you know your preferred debug target is available on your machine, but it doesn't appear as an option, select **Browse With** from the debug target dropdown list. Select your default browser target in the list, and select **Set as Default**.

2. To run the app, press **F5**, select the green arrow button, or select **Debug** > **Start Debugging**.

   A Node.js console window opens that shows the debugger listening port.

   Visual Studio starts the app by launching the startup file, *server.js*.



3. Close the browser and console windows.

Set a breakpoint and run the app

Breakpoints are the most basic and essential feature of reliable debugging. A breakpoint indicates where Visual Studio should suspend your running code. You can then observe variable values, memory behavior, or whether a branch of code is running.

1. In *server.js*, click in the gutter to the left of the `staticPath` declaration to set a breakpoint:

```
'use strict';
var path = require('path');
var express = require('express');

var app = express();

var staticPath = path.join(__dirname, '/');
app.use(express.static(staticPath));

// Allows you to set port in the project properties.
app.set('port', process.env.PORT || 3000);

var server = app.listen(app.get('port'), function () {
    console.log('listening');
});
```

```
var path = require('path');
var express = require('express');

var app = express();

var staticPath = path.join(__dirname, '/');
app.use(express.static(staticPath));

app.listen(1337, function () {
    console.log('listening');
});
```

2. To run the app, press **F5** or select **Debug** > **Start Debugging**.

   The debugger pauses at the breakpoint you set, with the current statement highlighted. Now, you can inspect your app state by hovering over variables that are currently in scope, using debugger windows like the **Locals** and **Watch** windows.

3. To continue running the app, press **F5**, select **Continue** in the **Debug** toolbar, or select **Debug** > **Continue**.

   If you want to use the Chrome Developer Tools or F12 Tools for Microsoft Edge, press **F12**. You can use these tools to examine the DOM and interact with the app by using the JavaScript Console.

4. Close the browser and console windows.

Set and hit a breakpoint in the client-side React code

In the preceding section, you attached the debugger to server-side Node.js code. To attach to and hit breakpoints in the client-side React code, you have to attach the debugger to the correct process. Here's one way to enable a browser and attach a process for debugging.

Enable the browser for debugging

You can use either Microsoft Edge or Google Chrome. Close all windows for the target browser. For Microsoft Edge, also shut down all instances of Chrome. Because both browsers share the Chromium code base, shutting down both browsers gives the best results.

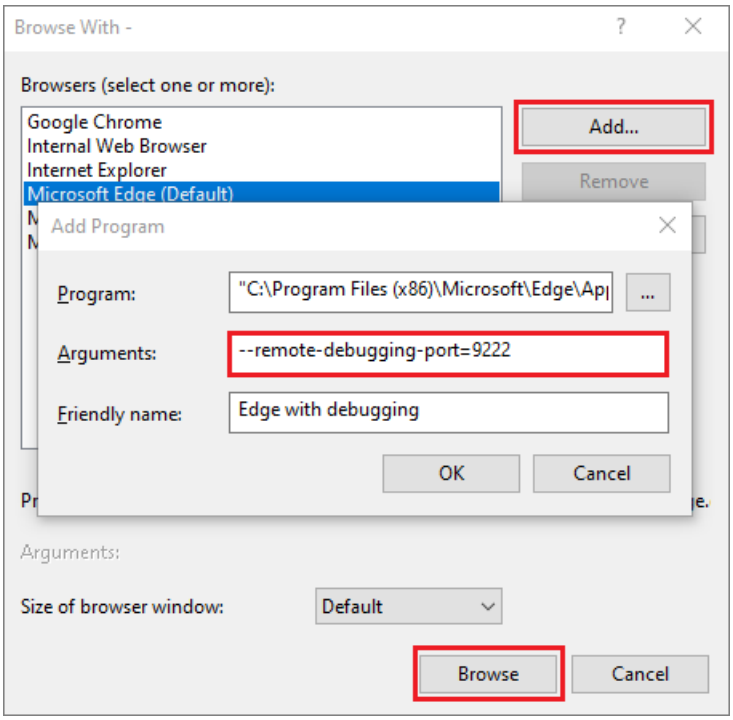
Other browser instances can prevent the browser from opening with debugging enabled. Browser extensions might prevent full debug mode. You might need to use Task Manager to find and end all running Chrome instances.

To start your browser with debugging enabled:

1. Select **Browse With** from the dropdown list in the **Debug** toolbar.
2. On the **Browse With** screen, with your preferred browser highlighted, select **Add**.
3. Enter the *--remote-debugging-port=9222* flag in the **Arguments** field.

4. Give the browser a new friendly name such as *Edge with debugging* or *Chrome with debugging*, and then select **OK**.

5. On the **Browse With** screen, select **Browse**.



- Alternatively, you can open the **Run** command by right-clicking the Windows **Start** button, and enter:

  ```
  msedge --remote-debugging-port=9222
  ```

  or

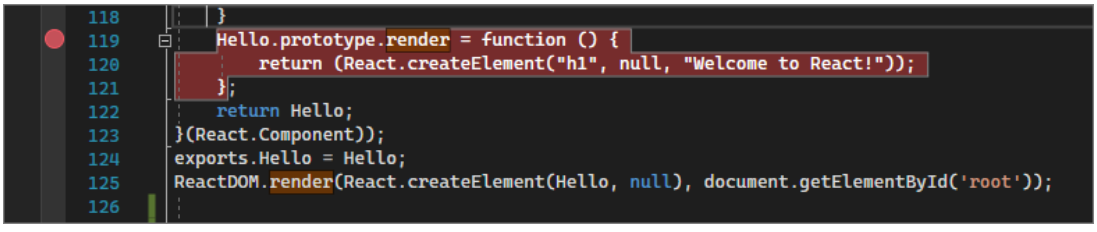
  ```
  chrome.exe --remote-debugging-port=9222
  ```

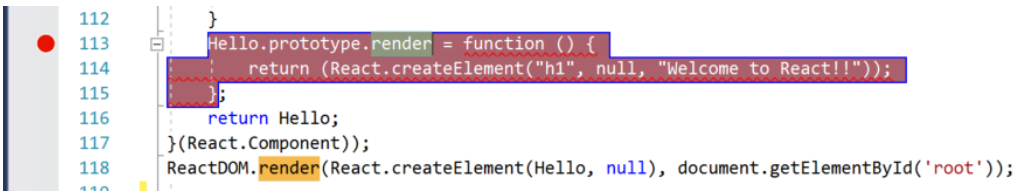
The browser starts with debugging enabled. The app isn't running yet, so the browser page is empty.

Attach the debugger to client-side script

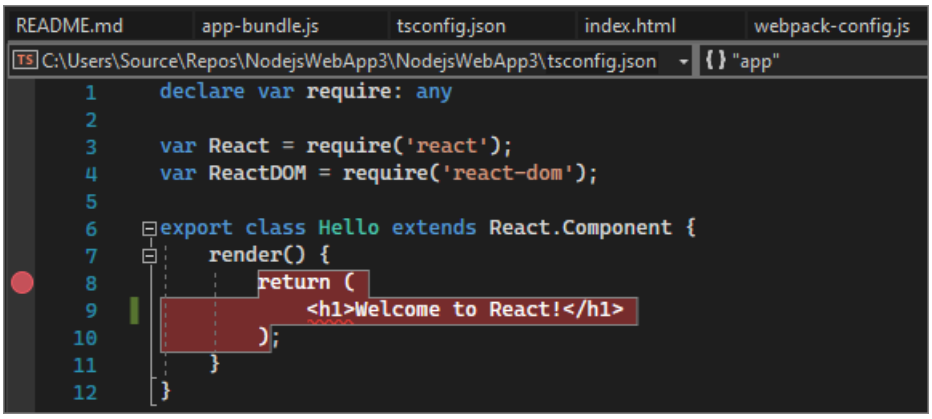
1. In the Visual Studio editor, set a breakpoint in either the *app-bundle.js* or *app.tsx* source code.

   - For *app-bundle.js*, set the breakpoint in the `render()` function. To find the `render()` function in the *app-bundle.js* file, press **Ctrl**+**F** or select **Edit** > **Find and Replace** > **Quick Find**, and enter *render* in the search field.





   - For *app.tsx*, set the breakpoint inside the `render()` function, on the `return` statement.



If you set the breakpoint in *app.tsx*, also update *webpack-config.js* to replace the following code, and save your changes.

Replace this code:

```
output: {
    filename: "./app-bundle.js",
},
```

With this code:

```
output: {
    filename: "./app-bundle.js",
    devtoolModuleFilenameTemplate: '[resource-path]'  // removes the webpack:/// prefix
},
```

This development-only setting enables debugging in Visual Studio. By default, webpack references in the source map file include the *webpack:///* prefix, which prevents Visual Studio from finding the source file *app.tsx*. This setting overrides the generated references in the source map file, *app-bundle.js.map*, when building the app. Specifically, this setting changes the reference to the source file from *webpack:///./app.tsx* to *./app.tsx*, which enables debugging.

2. Select your target browser as the debug target in Visual Studio, and then press **Ctrl+F5**, or select **Debug > Start Without Debugging**, to run the app in the browser.

   If you created a debugging-enabled browser configuration with a friendly name, choose that browser as your debug target.

   The app opens in a new browser tab.

3. Select **Debug > Attach to Process**, or press **Ctrl+Alt+P**.

> **TIP**
>
> Once you attach to the process the first time, you can quickly reattach to the same process by selecting **Debug** > **Reattach to Process** or pressing **Shift+Alt+P**.

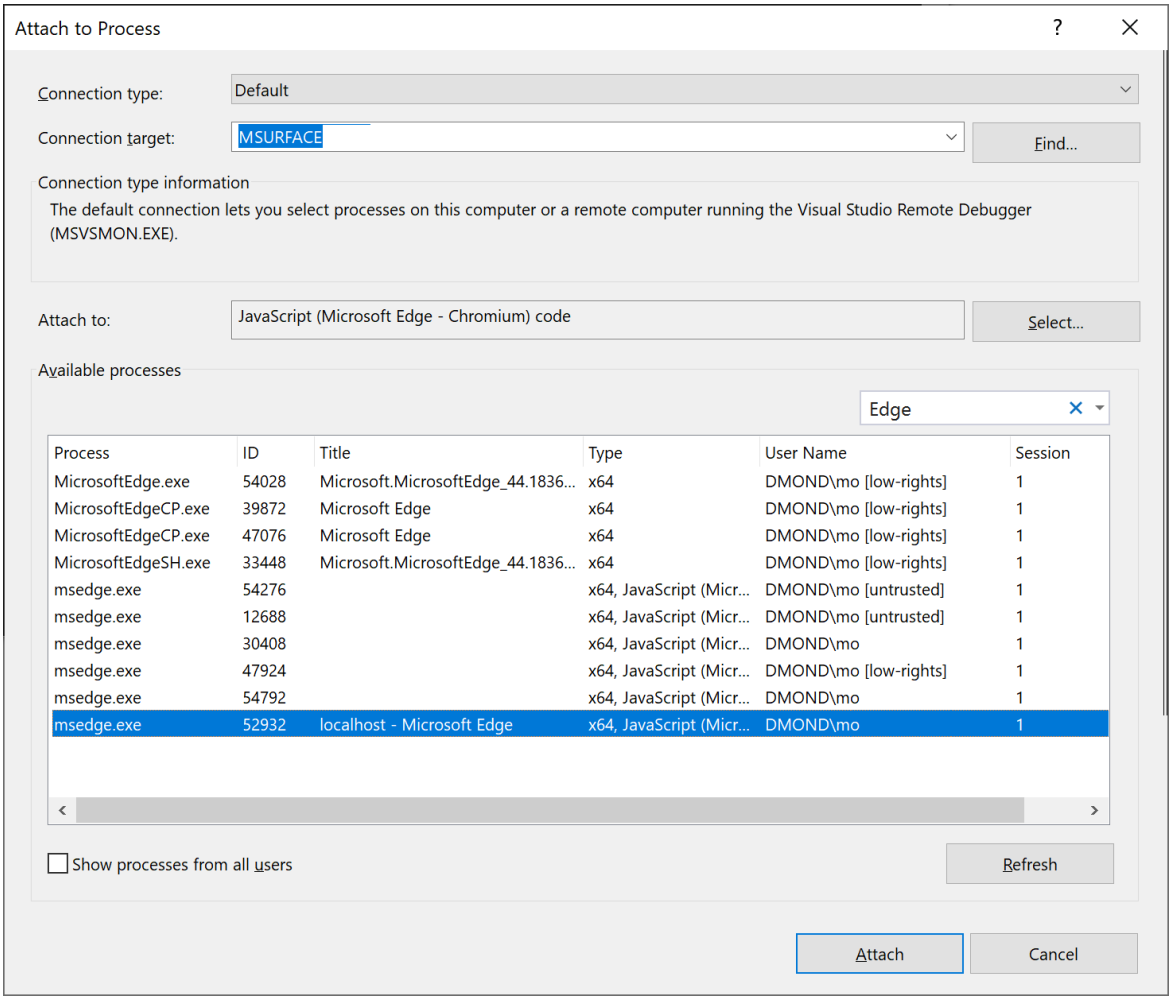
4. In the **Attach to Process** dialog box, get a filtered list of browser instances that you can attach to.

   Make sure the correct debugger for your target browser, **JavaScript (Chrome)** or **JavaScript (Microsoft Edge - Chromium)**, appears in the **Attach to** field. Type *chrome* or *edge* in the filter box to filter the results.

5. Select the browser process with the correct host port, **localhost** in this example. The port number **1337** or **localhost** might also appear in the **Title** field to help you select the correct process.

6. Select **Attach**.

   The following example shows an **Attach to Process** window for the Microsoft Edge browser.



> **TIP**
>
> If the debugger doesn't attach and you see the message **Unable to attach to the process. An operation is not legal in the current state.**, use Task Manager to close all instances of the target browser before starting the browser in debugging mode. Browser extensions may be running and preventing full debug mode.

7. Because the code with the breakpoint already executed, refresh your browser page to hit the breakpoint.

   Depending on your environment, browser state, and which steps you followed earlier, you might hit the breakpoint in *app-bundle.js* or its mapped location in *app.tsx*. Either way, you can step through code and

examine variables.

While the debugger is paused, you can examine your app state by hovering over variables and using debugger windows. To step through code, press **F11** or select **Debug** > **Step Into**, or press **F10** or select **Debug** > **Step Over**. To continue running the code, press **F5** or select **Continue**. For more information on basic debugging features, see First look at the debugger.

- If you can't break into code in *app.tsx*, retry using **Attach to Process** to attach the debugger as described in the previous steps. Make sure that your environment is set up correctly:
  - Close all browser instances, including Chrome extensions, by using the Task Manager. Make sure you start the browser in debug mode.
  - Make sure your source map file includes a reference to *./app.tsx* and not *webpack:///./app.tsx*, which prevents the Visual Studio debugger from locating *app.tsx*.

  Or, try using the `debugger;` statement in *app.tsx*, or set breakpoints in the Chrome Developer Tools or F12 Tools for Microsoft Edge instead.
- If you can't break into code in *app-bundle.js*, remove the source map file, *app-bundle.js.map*.

Next steps

Deploy the app to Linux App Service

Tutorial: Create an ASP.NET Core app with React in Visual Studio

6/24/2022 • 3 minutes to read • Edit Online

Applies to: Visual Studio Visual Studio for Mac Visual Studio Code

In this article, you learn how to build an ASP.NET Core project to act as an API backend and a React project to act as the UI.

Currently, Visual Studio includes ASP.NET Core Single Page Application (SPA) templates that support Angular and React. The templates provide a built in Client App folder in your ASP.NET Core projects that contains the base files and folders of each framework.

Starting in Visual Studio 2022 Preview 2, you can use the method described in this article to create ASP.NET Core Single Page Applications that:

- Put the client app in a separate project, outside from the ASP.NET Core project
- Create the client project based on the framework CLI installed on your computer

> **NOTE**
> Currently, the front-end project must be published manually (not currently supported with the Publish tool). For additional information, see https://github.com/MicrosoftDocs/visualstudio-docs/issues/7135.

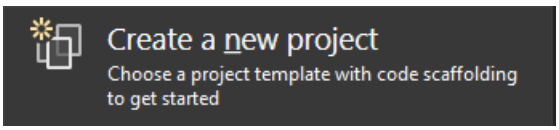
Prerequisites

Make sure to install the following:

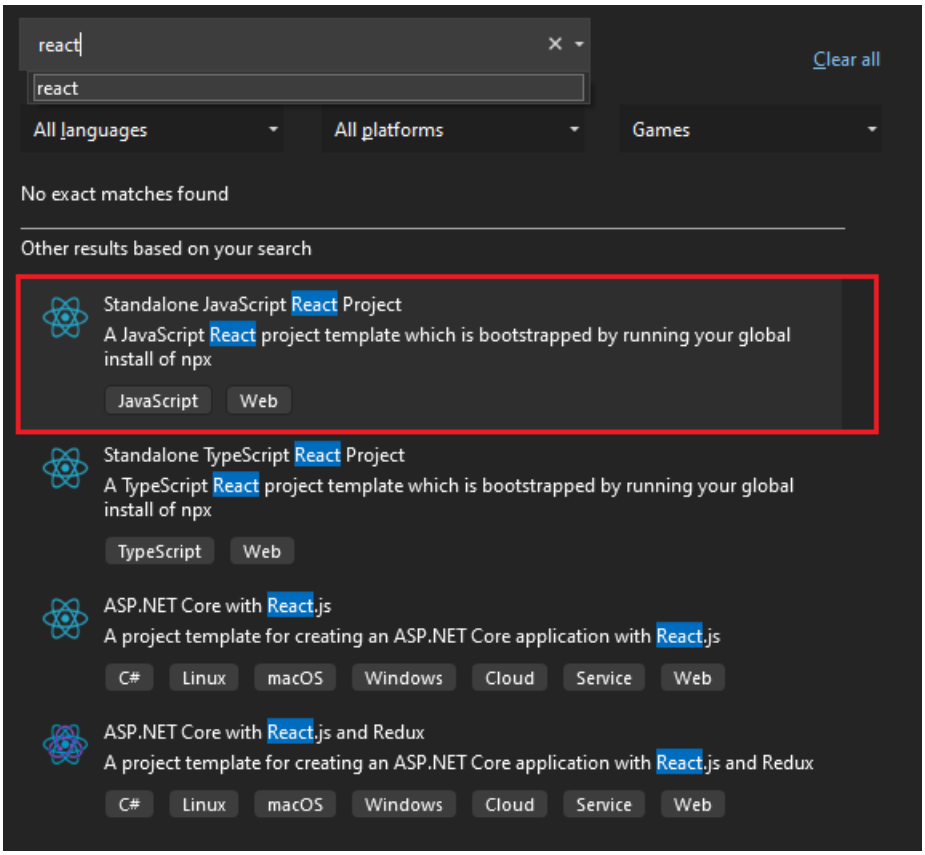
- Visual Studio 2022 or later with the **ASP.NET and web development** workload installed. Go to the Visual Studio downloads page to install it for free. If you need to install the workload and already have Visual Studio, go to **Tools** > **Get Tools and Features...**, which opens the Visual Studio Installer. Choose the **ASP.NET and web development** workload, then choose **Modify**.
- npm (https://www.npmjs.com/), which is included with Node.js
- npx (https://www.npmjs.com/package/npx)

Create the frontend app

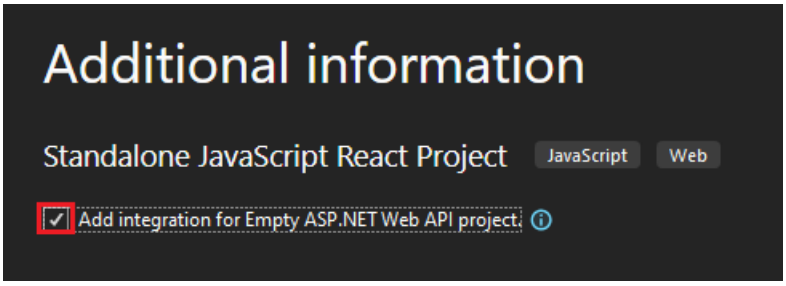
1. In the Start window (choose **File** > **Start Window** to open), select **Create a new project**.



2. Search for React in the search bar at the top and then select **Standalone JavaScript React Template**. (The standalone TypeScript React Template is not currently supported in this tutorial.)



3. Give your project and solution a name. When you get to the **Additional information** window, be sure to check the **Add integration for Empty ASP.NET Web API Project** option. This option adds files to your React template so that it can be hooked up later with the ASP.NET Core project.



   Once the project is created, you see some new and modified files:

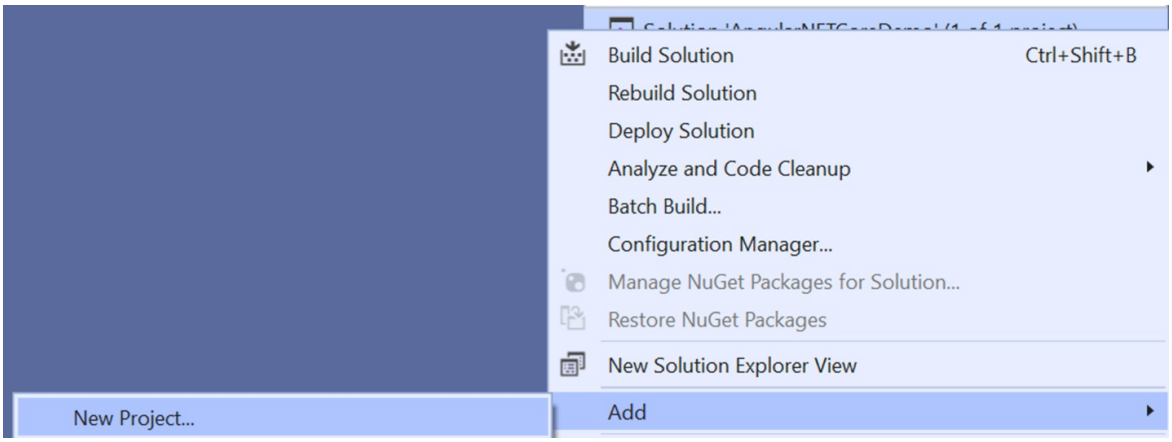
   - aspnetcore-https.js
   - aspnetcore-react.js
   - setupProxy.js
   - App.js (modified)
   - App.test.js (modified)

4. Select an installed browser from the Debug toolbar, such as Chrome or Microsoft Edge.

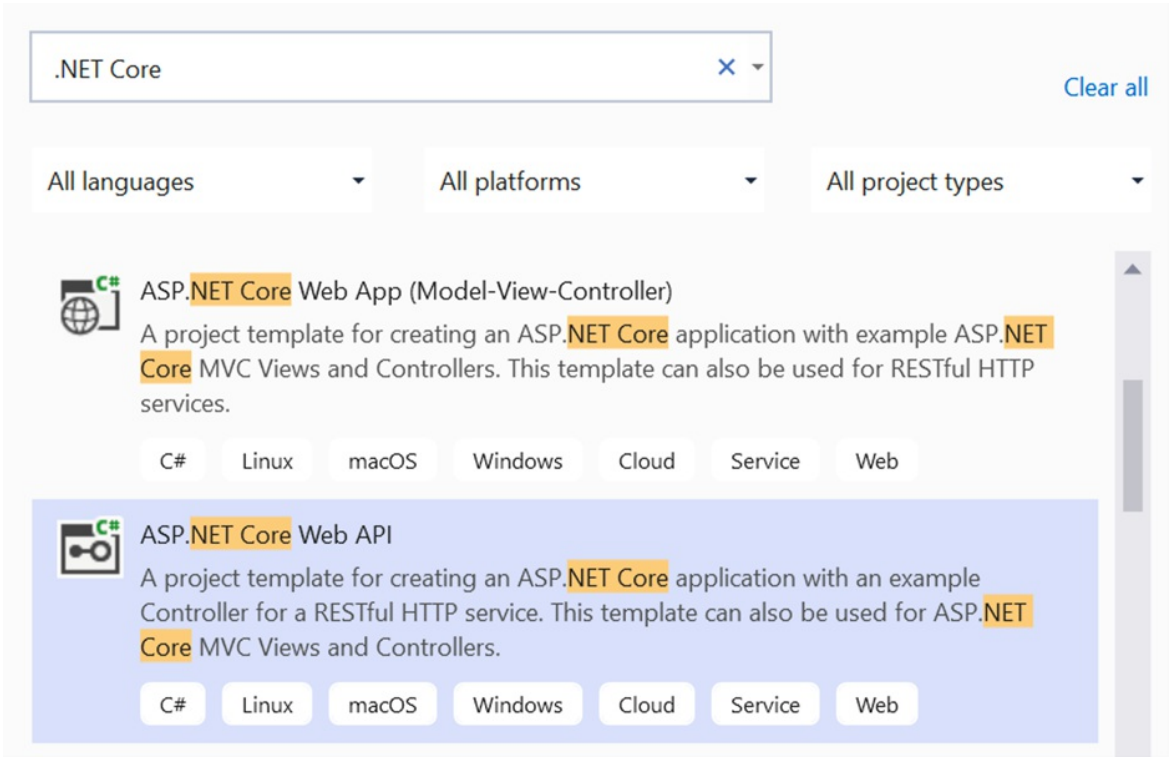
   If the browser you want is not yet installed, install the browser first, and then select it.

Create the backend app

1. In Solution Explorer, right-click the solution name, hover over **Add**, and then select **New Project**.

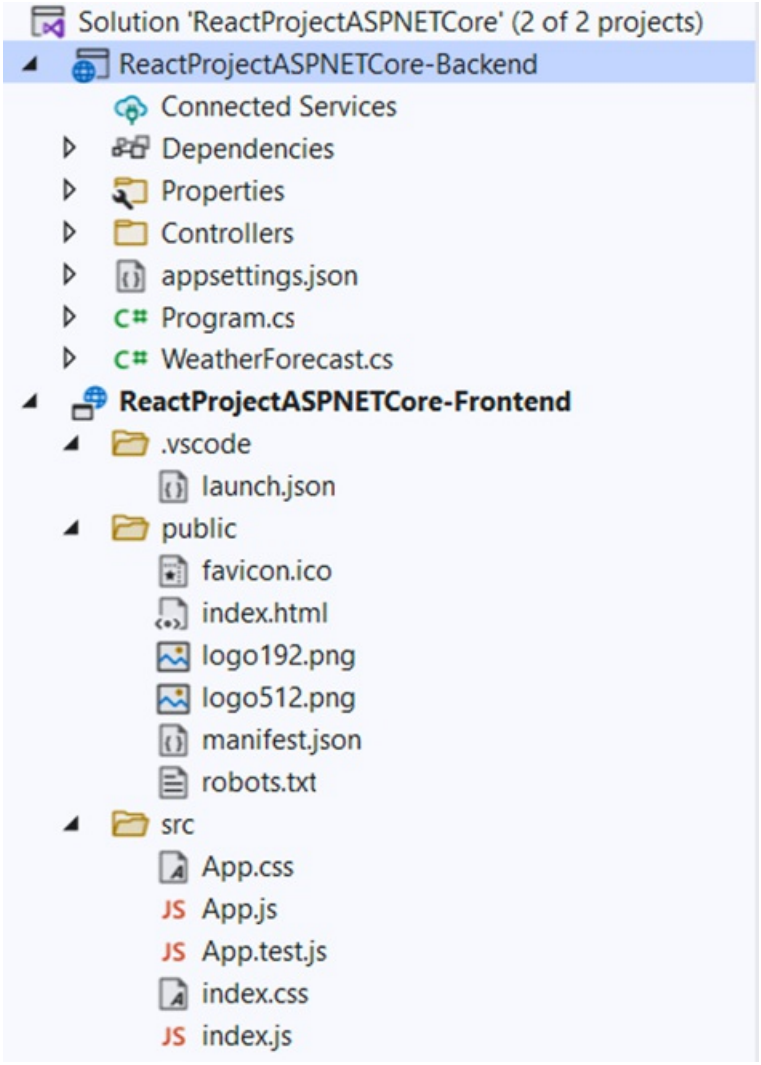


2. Search and select the ASP.NET Core Web API project.



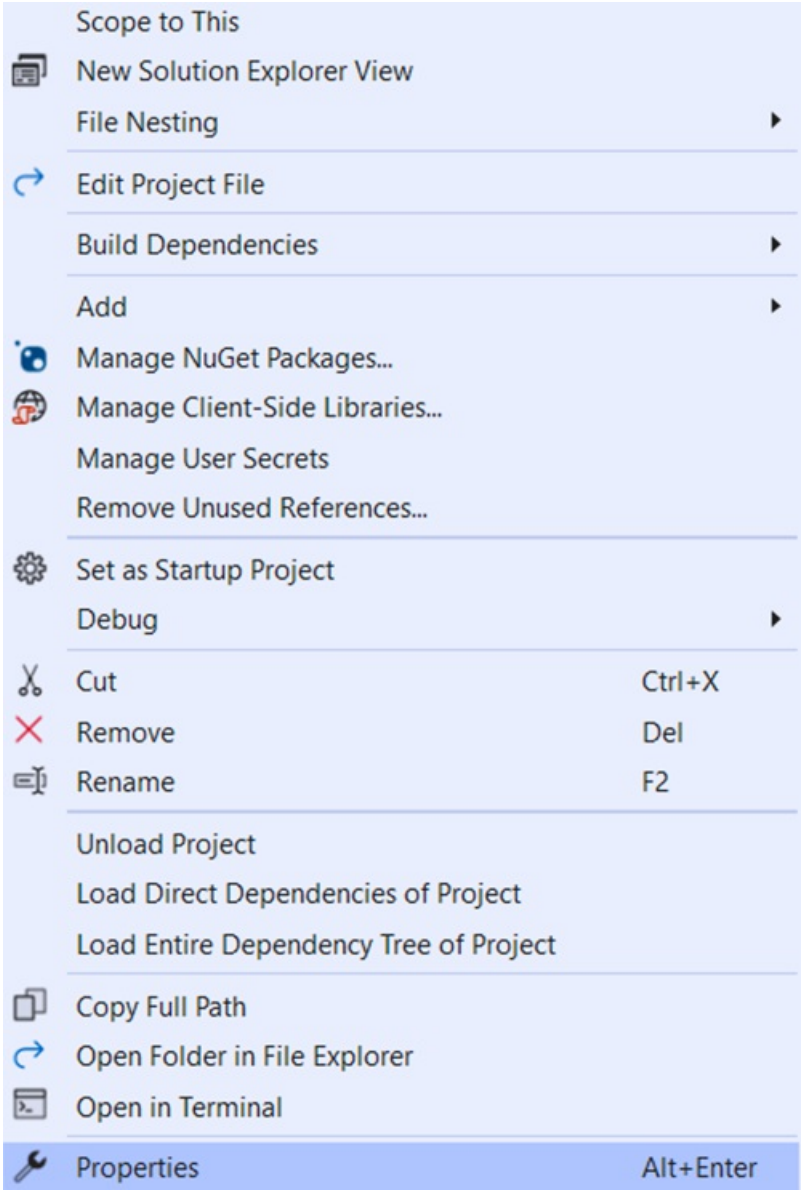
3. Give your project and solution a name. When you get to the **Additional information** window, select **.NET 6.0** as your target framework.

   Once the project is created, Solution Explorer should look like this:

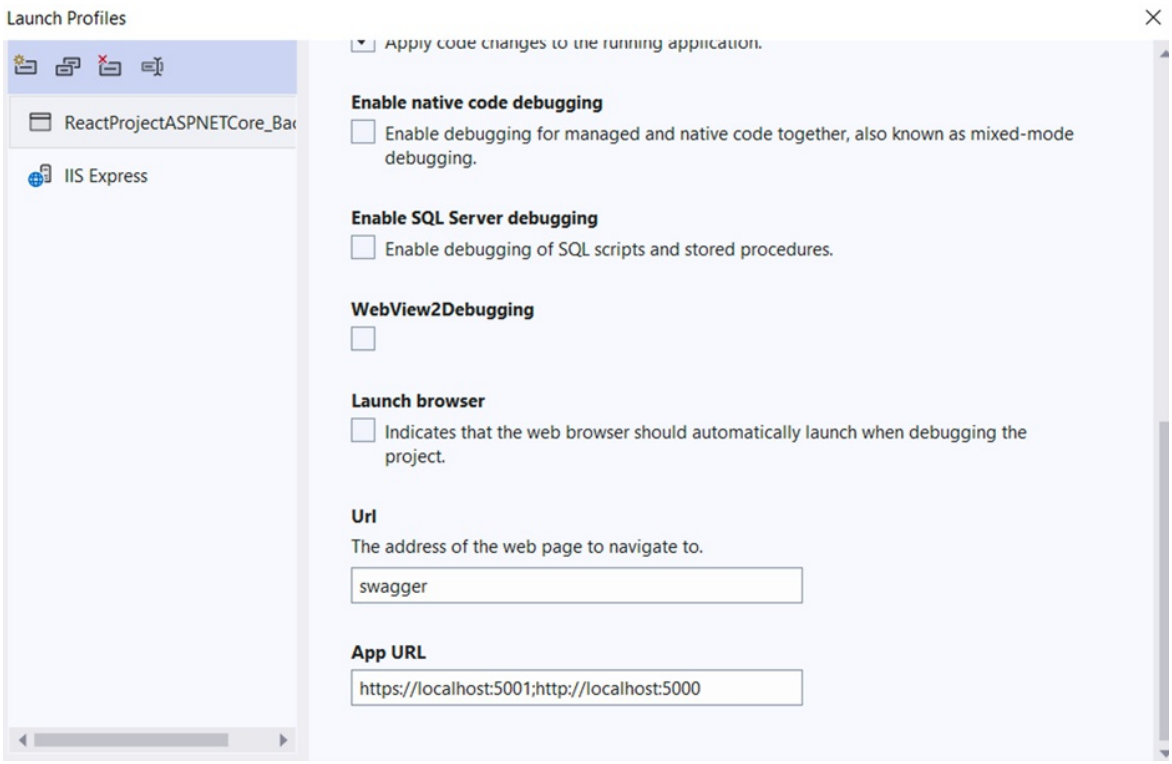


Set the project properties

1. In Solution Explorer, right-click the ASP.NET Core project and choose **Properties**.



2. Go to the Debug menu and select **Open debug launch profiles UI** option. Uncheck the **Launch Browser** option.

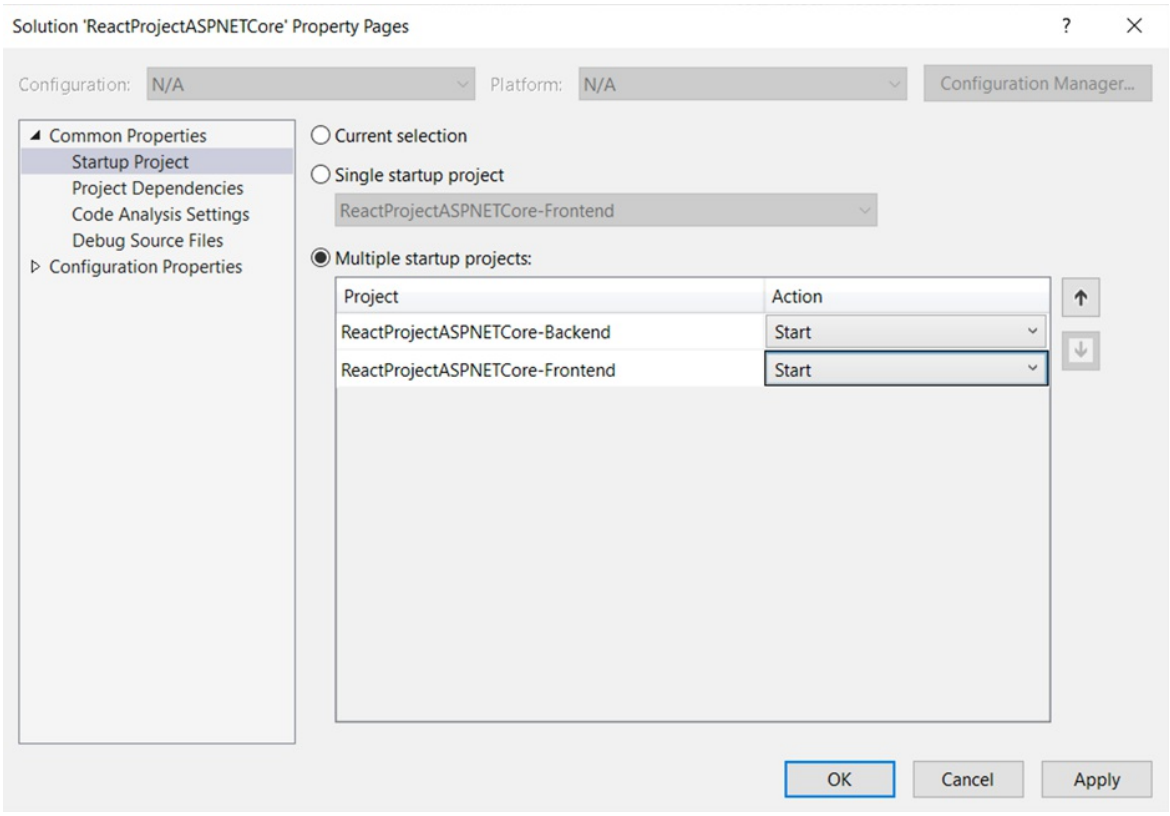


> **NOTE**
>
> Currently, *launch.json* must be located under the *.vscode* folder.

Set the startup project

1. In Solution Explorer, right-click the solution name and select **Set Startup Project**. Change the startup project from Single startup project to **Multiple startup projects**. Select **Start** for each project's action.

2. Next, select the backend project and move it above the frontend, so that it starts up first.



Start the project

1. Before you start the project, make sure that the port numbers match. Go to the *launchSettings.json* file in your ASP.NET Core project (in the *Properties* folder). Get the port number from the `applicationUrl` property.

   If there are multiple `applicationUrl` properties, look for one using an `https` endpoint. It should look similar to `https://localhost:7049`.

2. Then, go to the *setupProxy.js* file for your React project (look in the *src* folder). Update the target property to match the `applicationUrl` property in *launchSettings.json*. When you update it, that value should look similar to this:

   ```
   target: 'https://localhost:7049',
   ```

3. To start the project, press **F5** or select the **Start** button at the top of the window. You will see two command prompts appear:

   - The ASP.NET Core API project running
   - npm running the react-scripts start command

> **NOTE**
>
> Check console output for messages, such as a message instructing you to update your version of Node.js.

You should see an React app appear, that is populated via the API.

Troubleshooting

You may see the following error:

```
[HPM] Error occurred while trying to proxy request /weatherforecast from localhost:4200 to
https://localhost:5001 (ECONNREFUSED) (https://nodejs.org/api/errors.html#errors_common_system_errors)
```

If you see this issue, most likely the frontend started before the backend. Once you see the backend command prompt up and running, just refresh the React App in the browser.

Tutorial: Create an ASP.NET Core app with Angular in Visual Studio

6/24/2022 • 3 minutes to read • Edit Online

Applies to: ✔ Visual Studio ✘ Visual Studio for Mac ✘ Visual Studio Code

In this article, you learn how to build an ASP.NET Core project to act as an API backend and an Angular project to act as the UI.

Currently, Visual Studio includes ASP.NET Core Single Page Application (SPA) templates that support Angular and React. The templates provide a built in Client App folder in your ASP.NET Core projects that contains the base files and folders of each framework.

Starting in Visual Studio 2022 Preview 2, you can use the method described in this article to create ASP.NET Core Single Page Applications that:

- Put the client app in a separate project, outside from the ASP.NET Core project
- Create the client project based on the framework CLI installed on your computer

> **NOTE**
> Currently, the front-end project must be published manually (not currently supported with the Publish tool). For additional information, see https://github.com/MicrosoftDocs/visualstudio-docs/issues/7135.

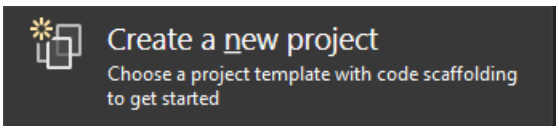
Prerequisites

Make sure to install the following:

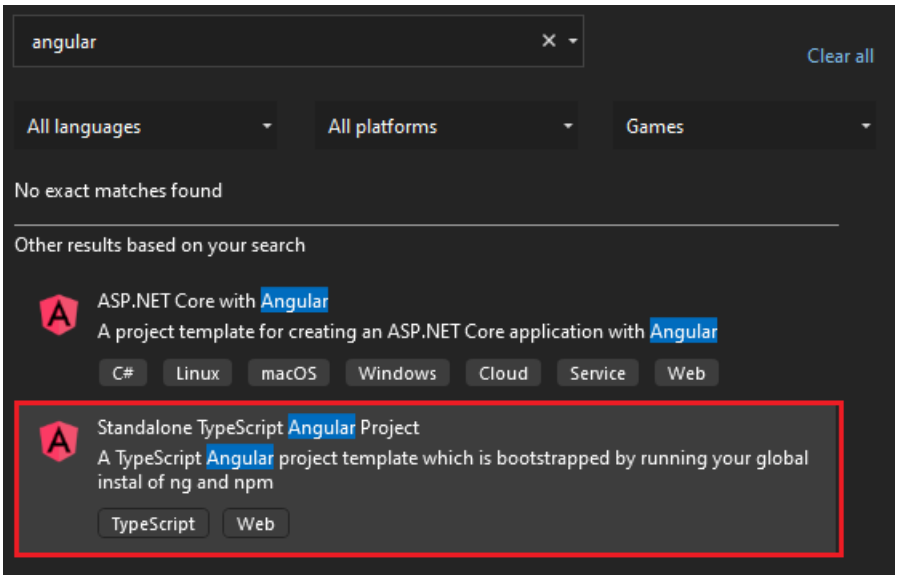
- Visual Studio 2022 Preview 2 or later with the **ASP.NET and web development** workload installed. Go to the Visual Studio downloads page to install it for free. If you need to install the workload and already have Visual Studio, go to **Tools** > **Get Tools and Features...**, which opens the Visual Studio Installer. Choose the **ASP.NET and web development** workload, then choose **Modify**.
- npm (https://www.npmjs.com/), which is included with Node.js
- Angular CLI (https://angular.io/cli) This can be the version of your choice

Create the frontend app

1. In the Start window (choose **File** > **Start Window** to open), select **Create a new project**.



2. Search for Angular in the search bar at the top and then select **Standalone TypeScript Angular Template**.



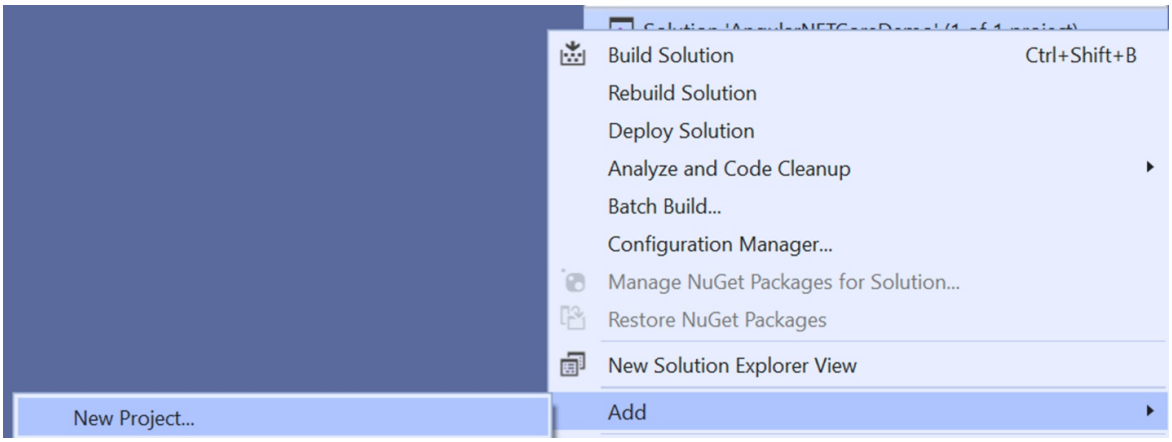
3. Give your project and solution a name. When you get to the **Additional information** window, be sure to check the **Add integration for Empty ASP.NET Web API Project** option. This option adds files to your Angular template so that it can be hooked up later with the ASP.NET Core project.

Once the project is created, you see some new and modified files:

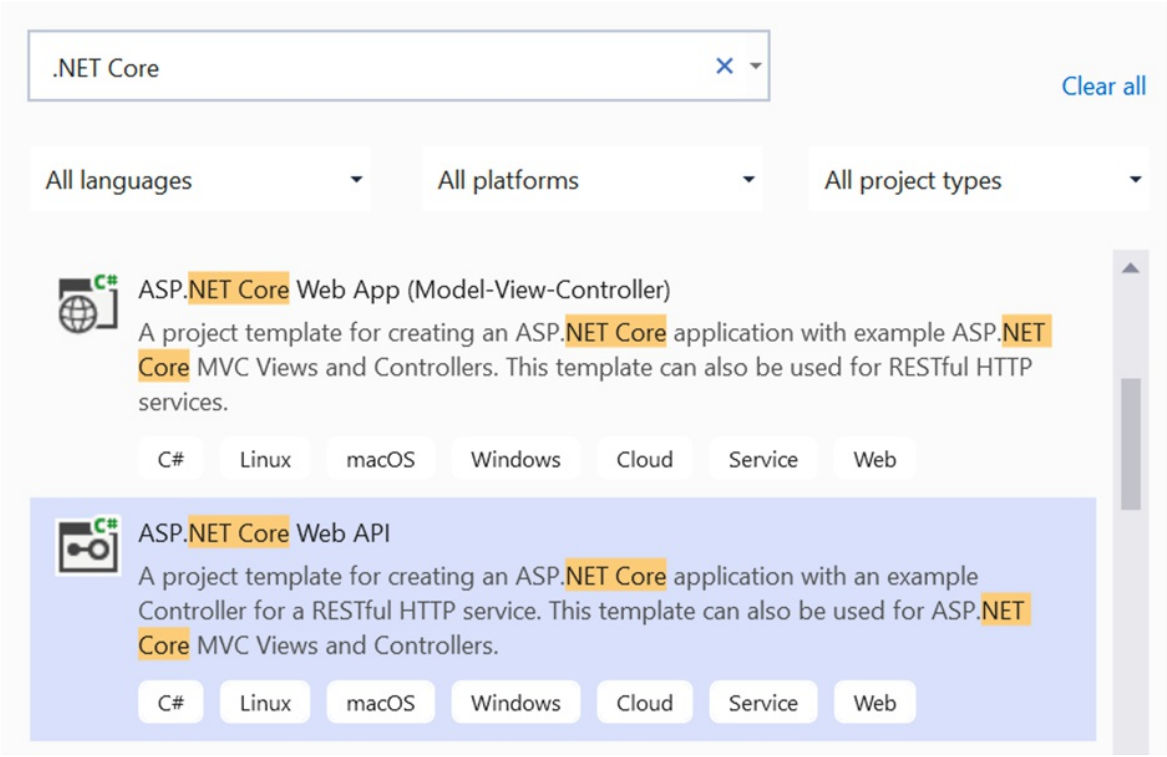
- aspnetcore-https.js
- proxy.js
- package.json(modified)
- angular.json(modified)
- app.components.ts
- app.module.ts

Create the backend app

1. In Solution Explorer, right-click the solution name, hover over **Add**, and then select **New Project**.

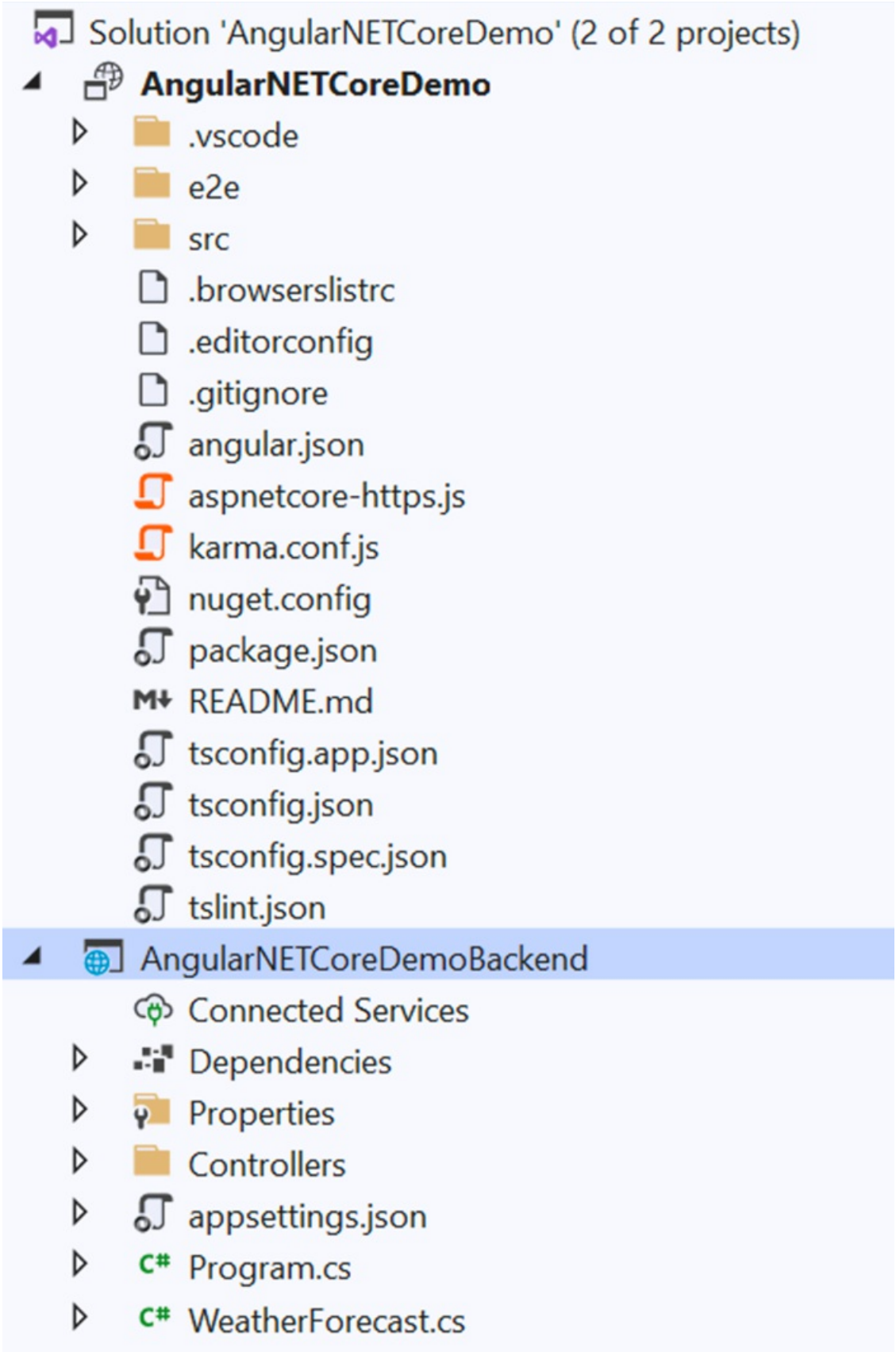


2. Search and select the ASP.NET Core Web API project.



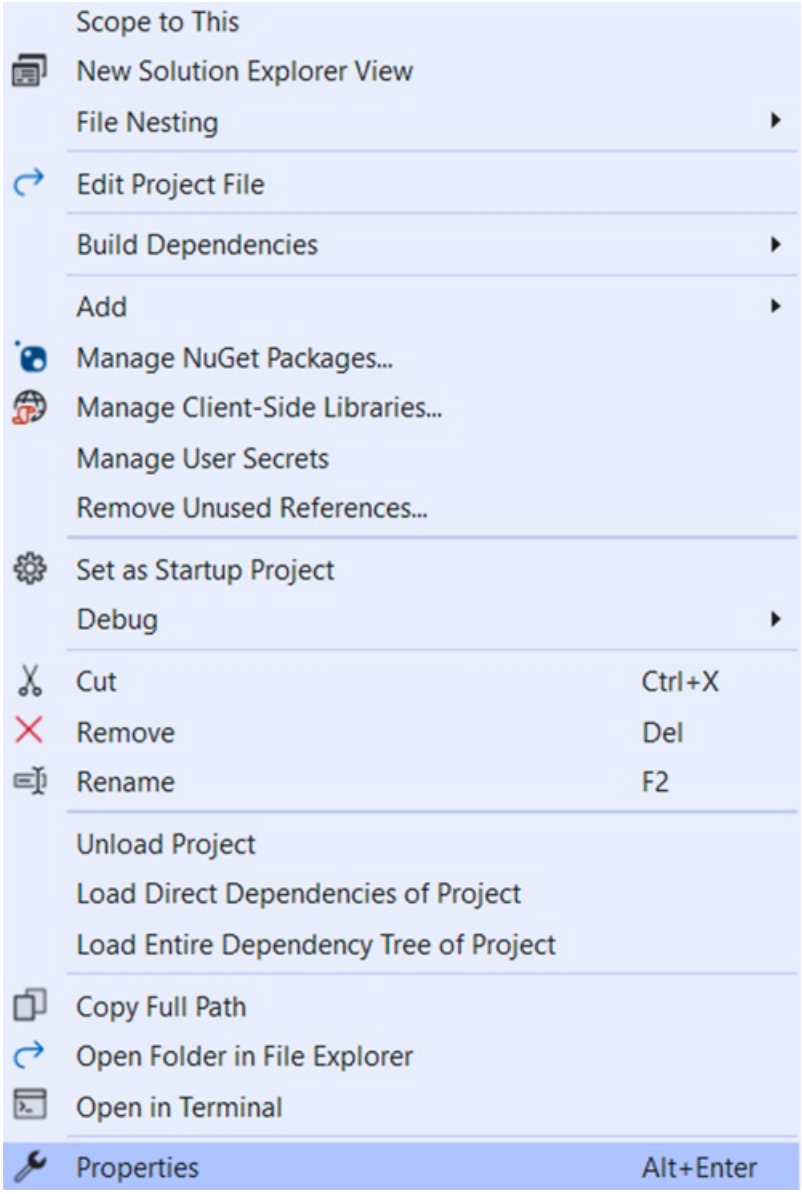
3. Give your project and solution a name. When you get to the **Additional information** window, select **.NET 6.0** as your target framework.

   Once the project is created, Solution Explorer should look like this:

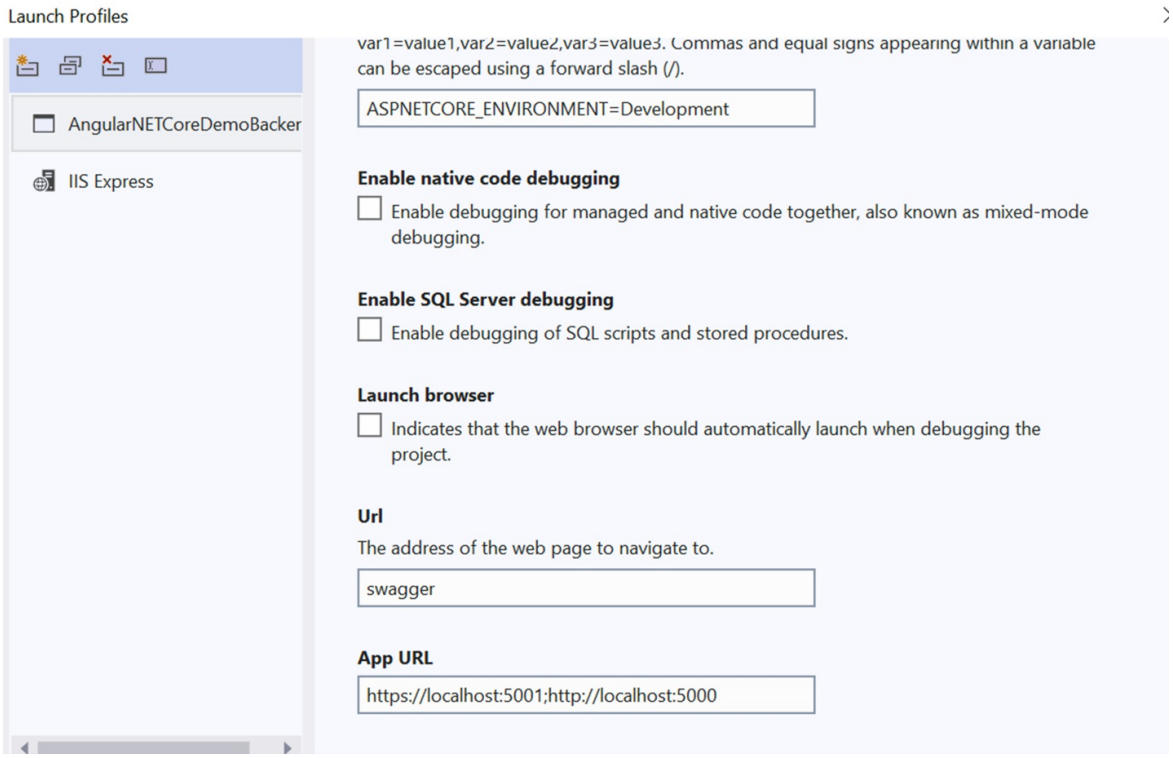


Set the project properties

1. In Solution Explorer, right-click the ASP.NET Core project and choose **Properties**.



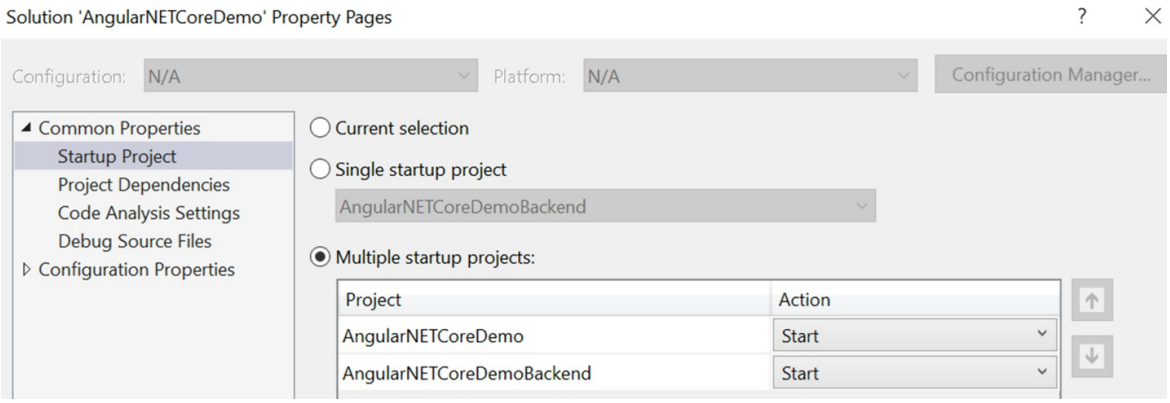
2. Go to the Debug menu and select **Open debug launch profiles UI** option. Uncheck the **Launch Browser** option.



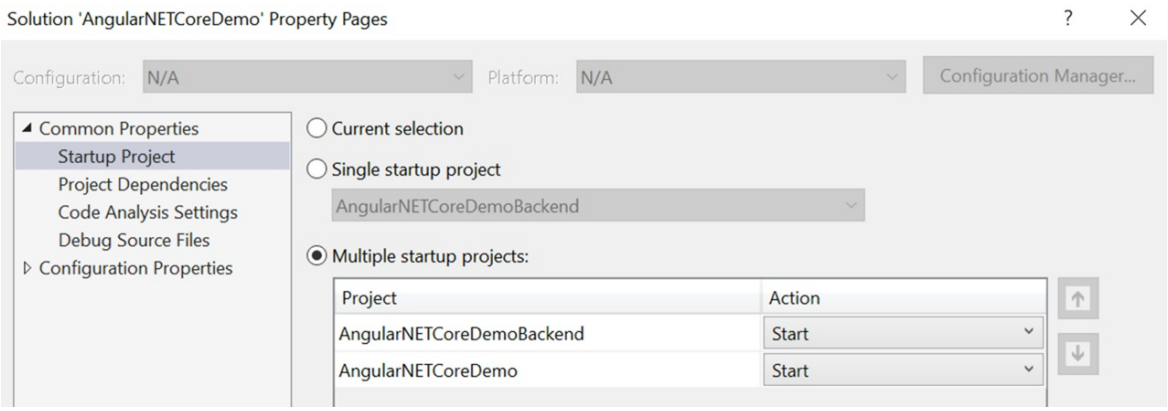
> **NOTE**
>
> Currently, *launch.json* must be located under the *.vscode* folder.

Set the startup project

1. In Solution Explorer, right-click the solution name and select **Set Startup Project**. Change the startup project from Single startup project to **Multiple startup projects**. Select **Start** for each project's action.



2. Next, select the backend project and move it above the frontend, so that it starts up first.



Start the project

Before you start the project, make sure that the port numbers match.

1. Go to the *launchSettings.json* file in your ASP.NET Core project (in the *Properties* folder). Get the port number from the `applicationUrl` property.

   If there are multiple `applicationUrl` properties, look for one using an `https` endpoint. It should look similar to `https://localhost:7049`.

2. Then, go to the *proxy.conf.js* file for your Angular project (look in the *src* folder). Update the target property to match the `applicationUrl` property in *launchSettings.json*. When you update it, that value should look similar to this:

   ```
   target: 'https://localhost:7049',
   ```

3. To start the project, press **F5** or select the **Start** button at the top of the window. You will see two command prompts appear:

   - The ASP.NET Core API project running

- The Angular CLI running the ng start command

> **NOTE**
>
> Check console output for messages, such as a message instructing you to update your version of Node.js.

You should see an Angular app appear, that is populated via the API.

Troubleshooting

You may see the following error:

```
[HPM] Error occurred while trying to proxy request /weatherforecast from localhost:4200 to
https://localhost:5001 (ECONNREFUSED) (https://nodejs.org/api/errors.html#errors_common_system_errors)
```

If you see this issue, most likely the frontend started before the backend. Once you see the backend command prompt up and running, just refresh the Angular App in the browser.

Tutorial: Create an ASP.NET Core app with Vue in Visual Studio

6/24/2022 • 4 minutes to read • Edit Online

Applies to: ✔ Visual Studio ✘ Visual Studio for Mac ✘ Visual Studio Code

In this article, you learn how to build an ASP.NET Core project to act as an API backend and a Vue project to act as the UI.

Currently, Visual Studio includes ASP.NET Core Single Page Application (SPA) templates that support Angular, React, and Vue. The templates provide a built in Client App folder in your ASP.NET Core projects that contains the base files and folders of each framework.

Starting in Visual Studio 2022 Preview 2, you can use the method described in this article to create ASP.NET Core Single Page Applications that:

- Put the client app in a separate project, outside from the ASP.NET Core project
- Create the client project based on the framework CLI installed on your computer

> **NOTE**
> Currently, the front-end project must be published manually (not currently supported with the Publish tool). For additional information, see https://github.com/MicrosoftDocs/visualstudio-docs/issues/7135.

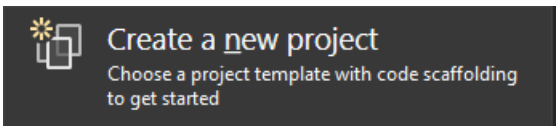
Prerequisites

Make sure to install the following:

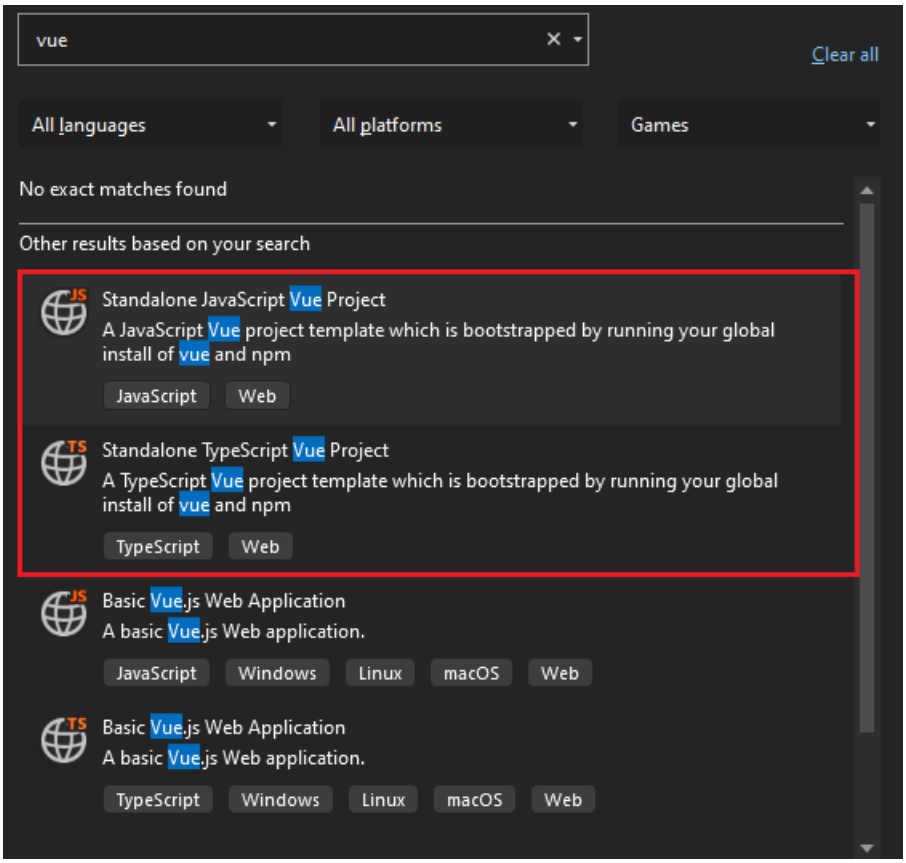
- Visual Studio 2022 Preview 2 or later with the **ASP.NET and web development** workload installed. Go to the Visual Studio downloads page to install it for free. If you need to install the workload and already have Visual Studio, go to **Tools** > **Get Tools and Features...**, which opens the Visual Studio Installer. Choose the **ASP.NET and web development** workload, then choose **Modify**.
- npm (https://www.npmjs.com/), which is included with Node.js
- Vue CLI (https://cli.vuejs.org/)

Create the frontend app

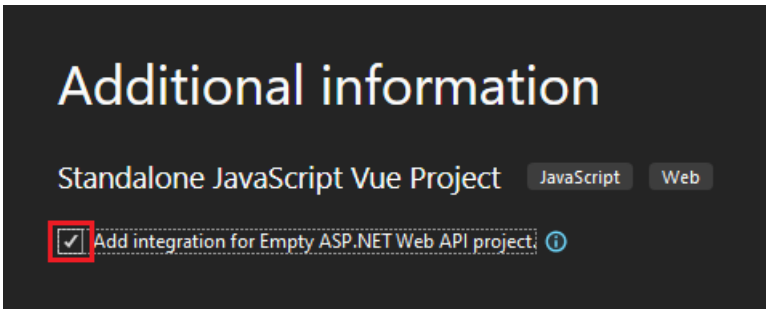
1. In the Start window (choose **File** > **Start Window** to open), select **Create a new project**.



2. Search for Vue in the search bar at the top and then select **Standalone JavaScript Vue Template** or **Standalone TypeScript Vue Template**.



3. Give your project and solution a name. When you get to the **Additional information** window, be sure to check the **Add integration for Empty ASP.NET Web API Project** option. This option adds files to your Vue template so that it can be hooked up later with the ASP.NET Core project.

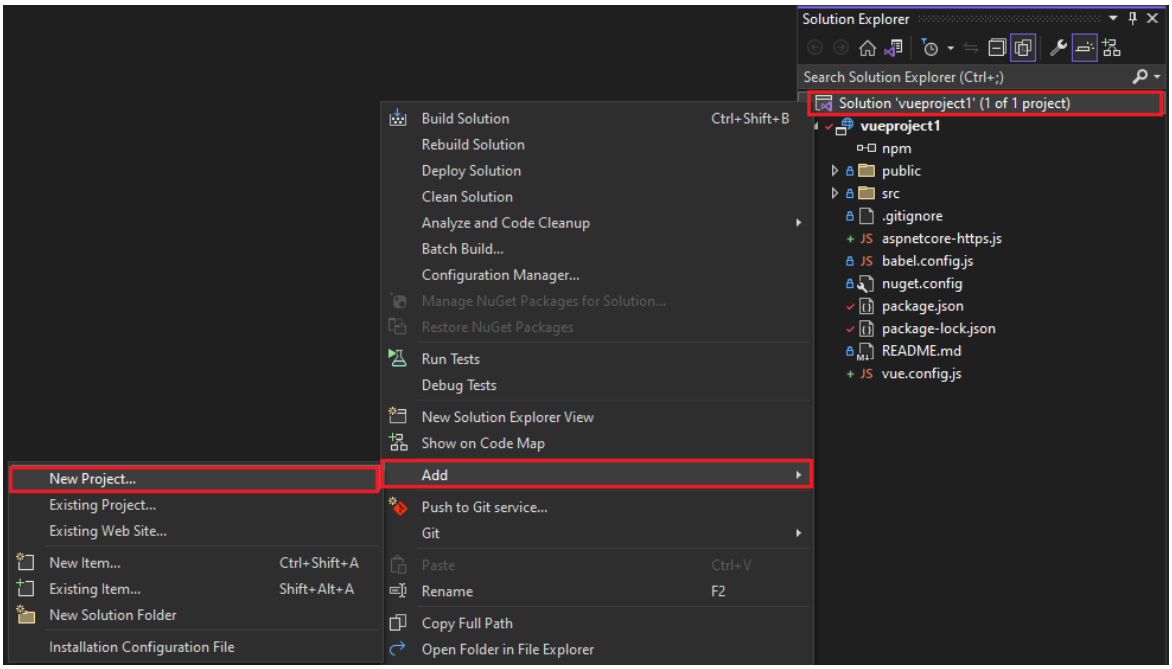


Once the project is created, you see some new and modified files:

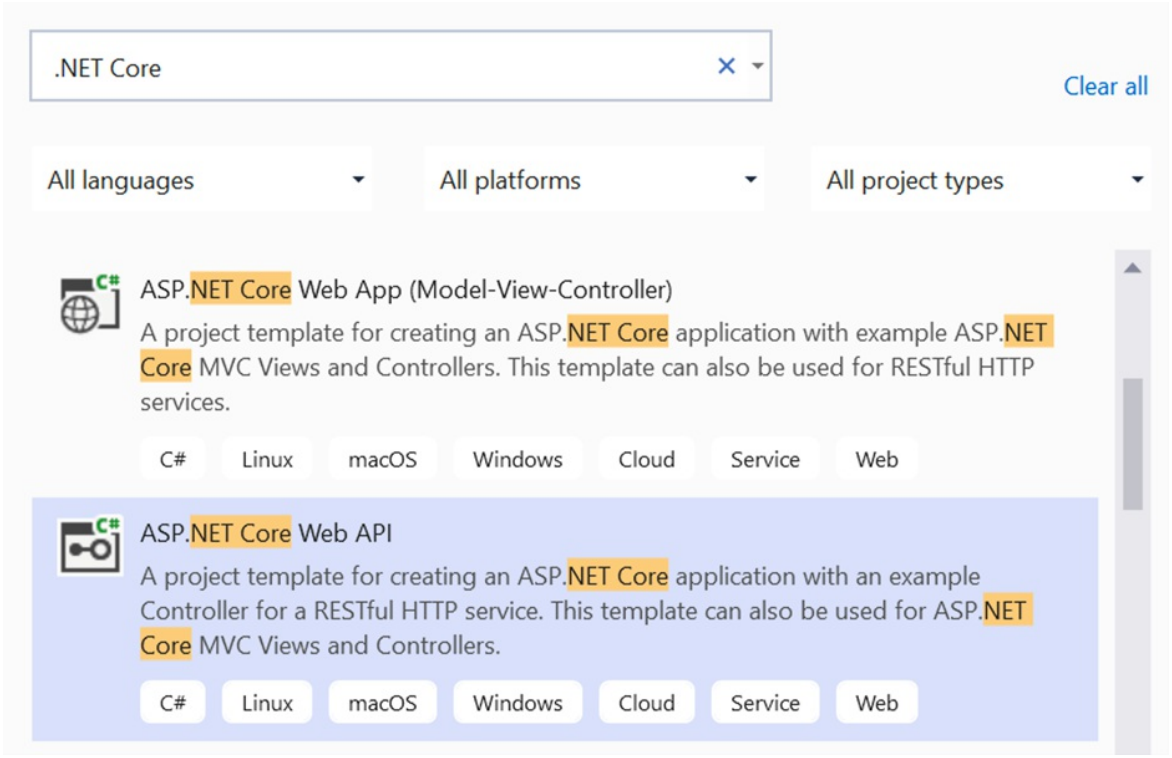
- aspnetcore-https.js
- vue.config.json (modified)
- HelloWorld.vue (modified)
- package.json (modified)

Create the backend app

1. In Solution Explorer, right-click the solution name, hover over **Add**, and then select **New Project**.

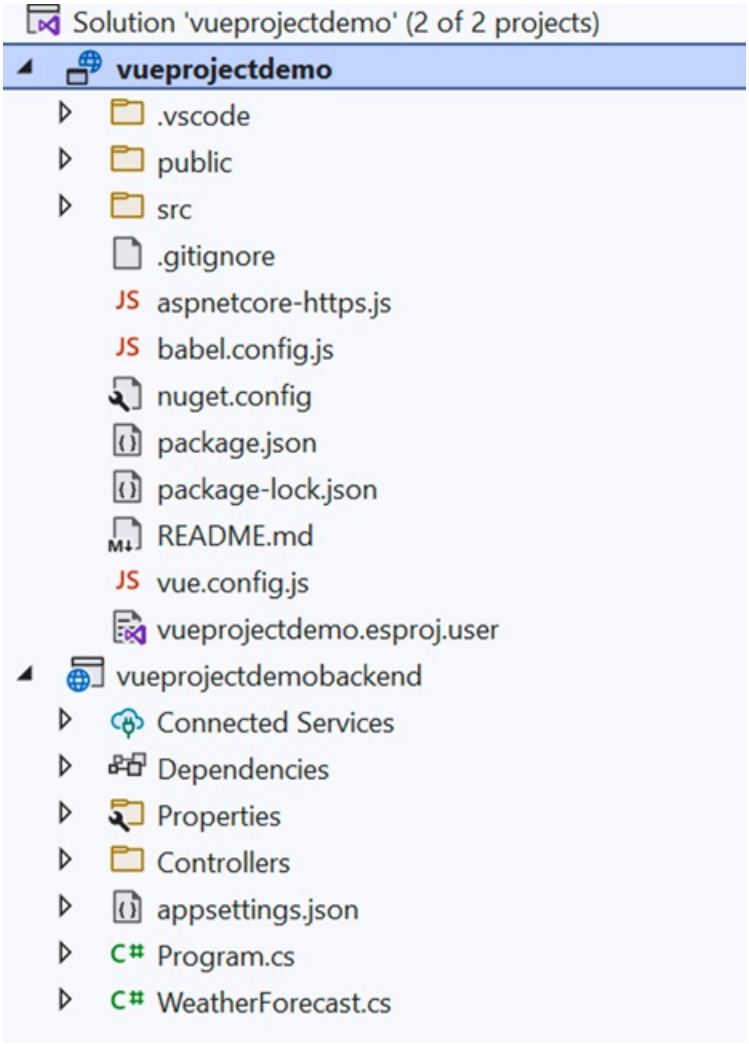


2. Search and select the ASP.NET Core Web API project.



3. Give your project and solution a name. When you get to the **Additional information** window, select **.NET 6.0** as your target framework.

   Once the project is created, Solution Explorer should look like this:

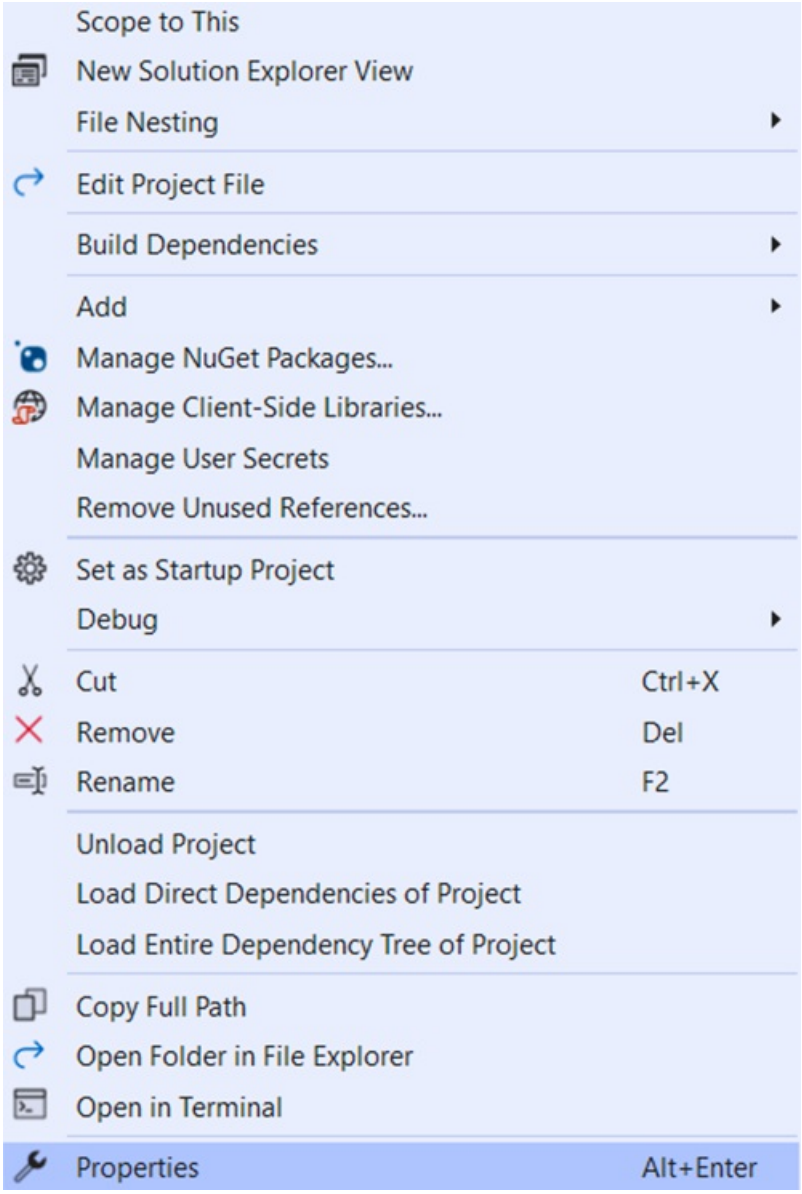


4. Open `launchSettings.json` from the **Properties** folder, and under the profiles section for the backend app, change the default ports to 5001 and 5003.

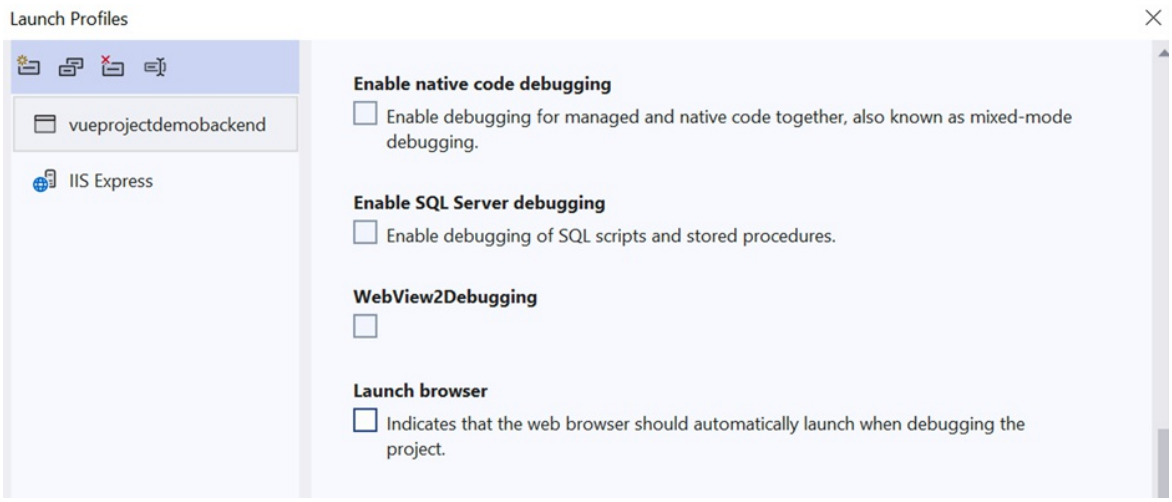
```
"profiles": {
  "yourbackendapp": {
    "commandName": "Project",
    "launchUrl": "swagger",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    },
    "applicationUrl": "https://localhost:5001;http://localhost:5003",
    "dotnetRunMessages": true
  },
```

Set the project properties

1. In Solution Explorer, right-click the ASP.NET Core project and choose **Properties**.



2. Go to the Debug menu and select **Open debug launch profiles UI** option. Clear the **Launch browser** option.

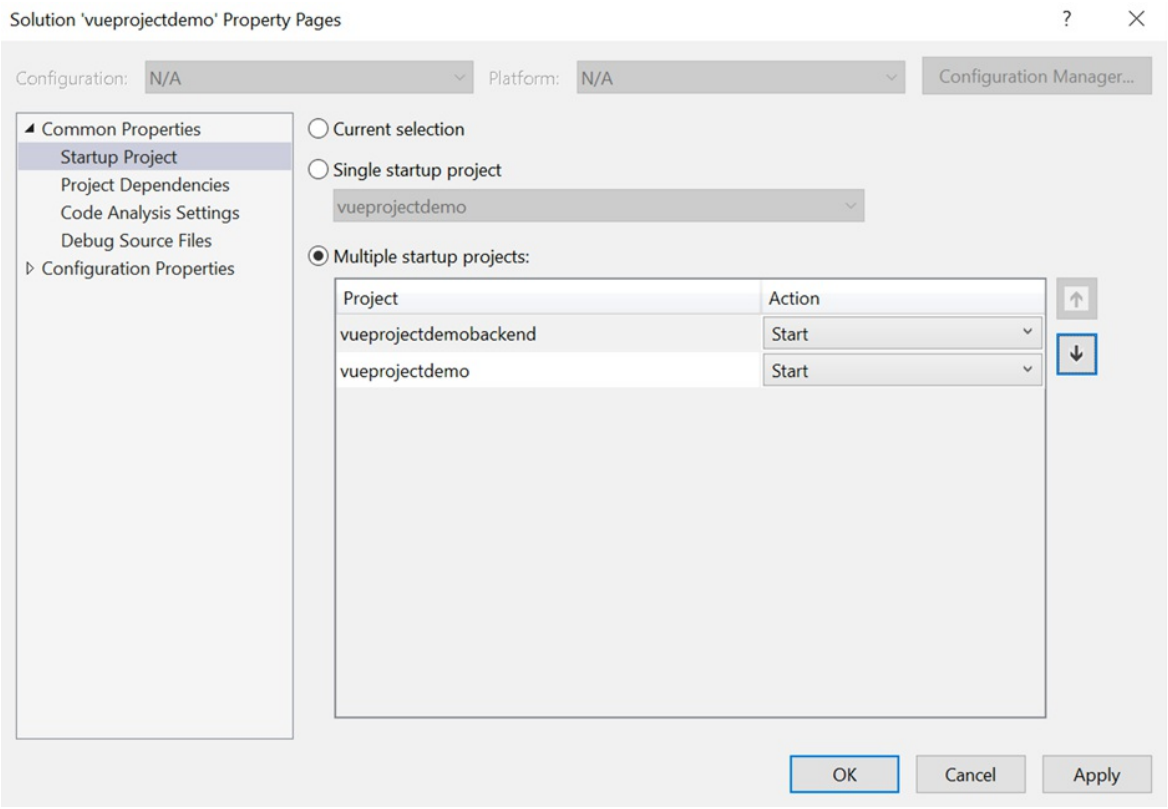


> **NOTE**
>
> Currently, *launch.json* must be located under the *.vscode* folder.

Set the startup project

1. In Solution Explorer, right-click the solution name and select **Set Startup Project**. Change the startup project from Single startup project to **Multiple startup projects**. Select **Start** for each project's action.

2. Next, select the backend project and move it above the frontend, so that it starts up first.



Start the project

1. Before you start the project, make sure that the port numbers match. Go to the *launchSettings.json* file in your ASP.NET Core project (in the *Properties* folder). Get the port number from the `applicationUrl` property.

   If there are multiple `applicationUrl` properties, look for one using an `https` endpoint. It should look similar to `https://localhost:5001`.

2. Then, go to the *vue.config.js* file for your Vue project. Update the target property to match the `applicationUrl` property in *launchSettings.json*. When you update it, that value should look similar to this:

   ```
   target: 'https://localhost:5001',
   ```

3. To start the project, press **F5** or select the **Start** button at the top of the window. You will see two command prompts appear:

   - The ASP.NET Core API project running
   - The Vue CLI running the vue-cli-service serve command

   > **NOTE**
   > Check console output for messages, such as a message instructing you to update your version of Node.js.

You should see the Vue app appear, that is populated via the API.

Troubleshooting

Proxy error

You may see the following error:

```
[HPM] Error occurred while trying to proxy request /weatherforecast from localhost:4200 to
https://localhost:5001 (ECONNREFUSED) (https://nodejs.org/api/errors.html#errors_common_system_errors)
```

If you see this issue, most likely the frontend started before the backend. Once you see the backend command prompt up and running, just refresh the Vue app in the browser.

Otherwise, if the port is in use, try 5002 in *launchSettings.json* and *vue.config.js*.

Outdated version of Vue

If you see the console message **Could not find the file 'C:\Users\Me\source\repos\vueprojectname\package.json'** when you create the project, you may need to update your version of the Vue CLI. After you update the Vue CLI, you may also need to delete the *.vuerc* file in *C:\Users\[yourprofilename]*.

Docker

If you enable Docker support while creating the web API project, the backend may start up using the Docker profile and not listen on the configured port 5001. To resolve:

Edit the Docker profile in the launchSettings.json by adding the following properties:

```
"httpPort": 5003,
"sslPort": 5001
```

Alternatively, reset using the following method:

1. In the Solution properties, set your backend app as the startup project.
2. In the Debug menu, switch the profile using the **Start** button drop-down menu to the profile for your backend app.
3. Next, in the Solution properties, reset to multiple startup projects.

Tutorial: Create an ASP.NET Core app with TypeScript in Visual Studio

6/24/2022 • 7 minutes to read • Edit Online

Applies to: ✔ Visual Studio ✘ Visual Studio for Mac ✘ Visual Studio Code

In this tutorial for Visual Studio development ASP.NET Core and TypeScript, you create a simple web application, add some TypeScript code, and then run the app.

Starting in Visual Studio 2022, if you want to use Angular or Vue with ASP.NET Core, it is recommended that you use the ASP.NET Core Single Page Application (SPA) templates to create an ASP.NET Core app with TypeScript. For more information, see the Visual Studio tutorials for Angular or Vue.

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

In this tutorial, you learn how to:

- Create an ASP.NET Core project
- Add the NuGet package for TypeScript support
- Add some TypeScript code
- Run the app
- Add a third-party library using npm

Prerequisites

- You must have Visual Studio installed and the ASP.NET web development workload.

  If you haven't already installed Visual Studio 2022, go to the Visual Studio downloads page to install it for free.

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

  If you need to install the workload but already have Visual Studio, go to **Tools** > **Get Tools and Features...**, which opens the Visual Studio Installer. Choose the **ASP.NET and web development** workload, then choose **Modify**.

Create a new ASP.NET Core MVC project

Visual Studio manages files for a single application in a *project*. The project includes source code, resources, and configuration files.

> **NOTE**
> To start with an empty ASP.NET Core project and add a TypeScript frontend, see ASP.NET Core with TypeScript instead.

In this tutorial, you begin with a simple project containing code for an ASP.NET Core MVC app.

1. Open Visual Studio.
2. Create a new project.

In Visual Studio 2022, choose **Create a new project** in the start window. If the start window is not open, choose **File** > **Start Window**. Type **web app**, choose **C#** as the language, then choose **ASP.NET Core Web Application (Model-View-Controller)**, and then choose **Next**. On the next screen, name the project, and then choose **Next**.

Choose either the recommended target framework or .NET 6, and then choose **Create**.

In Visual Studio 2019, choose **Create a new project** in the start window. If the start window is not open, choose **File** > **Start Window**. Type **web app**, choose **C#** as the language, then choose **ASP.NET Core Web Application (Model-View-Controller)**, and then choose **Next**. On the next screen, name the project, and then choose **Next**.

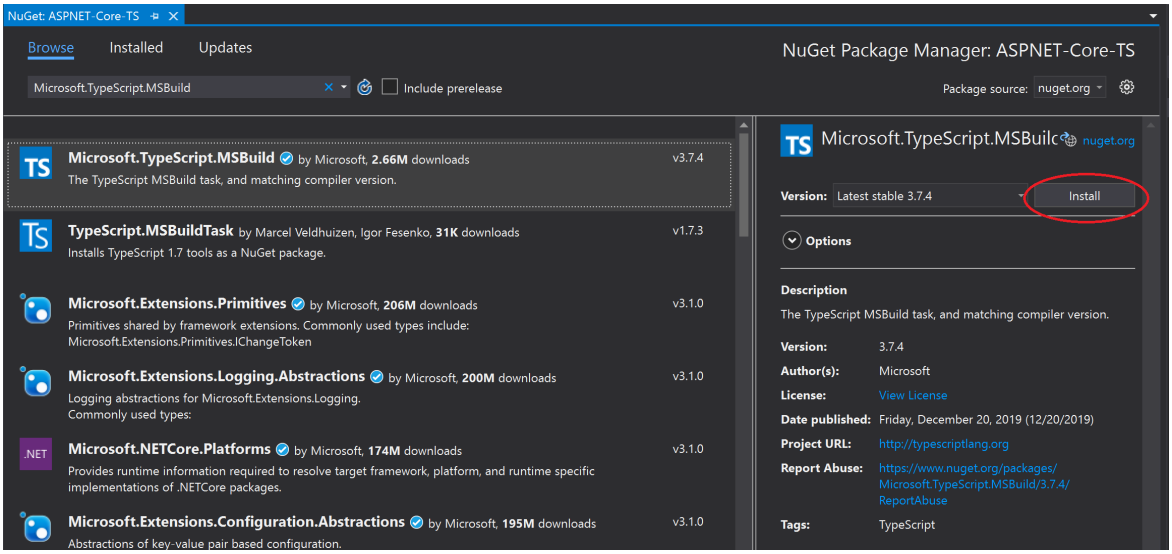
Choose either the recommended target framework or .NET 6, and then choose **Create**.

If you don't see the **ASP.NET Core Web App** project template, you must add the **ASP.NET and web development** workload. For detailed instructions, see the Prerequisites.

Visual Studio creates the new solution and opens your project in the right pane.

Add some code

1. In Solution Explorer (right pane). right-click the project node and choose **Manage NuGet Packages**. In the **Browse** tab, search for **Microsoft.TypeScript.MSBuild**, and then click **Install** on the right to install the package.



   Visual Studio adds the NuGet package under the **Dependencies** node in Solution Explorer.

2. Right-click the project node and choose **Add** > **New Item**. Choose the **TypeScript JSON Configuration File**, and then click **Add**.

   Visual Studio adds the *tsconfig.json* file to the project root. You can use this file to configure options for the TypeScript compiler.

3. Open *tsconfig.json* and replace the default code with the following code:

```
{
  "compileOnSave": true,
  "compilerOptions": {
    "noImplicitAny": false,
    "noEmitOnError": true,
    "removeComments": false,
    "sourceMap": true,
    "target": "es5",
    "outDir": "wwwroot/js"
  },
  "include": [
    "scripts/**/*"
  ]
}
```

The *outDir* option specifies the output folder for the plain JavaScript files that are transpiled by the TypeScript compiler.

This configuration provides a basic introduction to using TypeScript. In other scenarios, for example when using gulp or webpack, you may want a different intermediate location for the transpiled JavaScript files, depending on your tools and configuration preferences, instead of *wwwroot/js*.

4. In Solution Explorer, right-click the project node and choose **Add > New Folder**. Use the name *scripts* for the new folder.

5. Right-click the *scripts* folder and choose **Add > New Item**. Choose the **TypeScript File**, type the name *app.ts* for the filename, and then click **Add**.

   Visual Studio adds *app.ts* to the *scripts* folder.

6. Open *app.ts* and add the following TypeScript code.

```
function TSButton() {
   let name: string = "Fred";
   document.getElementById("ts-example").innerHTML = greeter(user);
}

class Student {
   fullName: string;
   constructor(public firstName: string, public middleInitial: string, public lastName: string) {
       this.fullName = firstName + " " + middleInitial + " " + lastName;
   }
}

interface Person {
   firstName: string;
   lastName: string;
}

function greeter(person: Person) {
   return "Hello, " + person.firstName + " " + person.lastName;
}

let user = new Student("Fred", "M.", "Smith");
```

Visual Studio provides IntelliSense support for your TypeScript code.

To test this, remove `.lastName` from the `greeter` function, then retype the ".", and you see IntelliSense.

Select `lastName` to add the last name back to the code.

7. Open the *Views/Home* folder, and then open *Index.cshtml*.

8. Add the following HTML code to the end of the file.

```
<div id="ts-example">
    <br />
    <button type="button" class="btn btn-primary btn-md" onclick="TSButton()">
        Click Me
    </button>
</div>
```

9. Open the *Views/Shared* folder, and then open *_Layout.cshtml*.

10. Add the following script reference before the call to `@RenderSection("Scripts", required: false)`:

```
<script src="~/js/app.js"></script>
```

Build the application

1. Choose **Build > Build Solution**.

   Although the app builds automatically when you run it, we want to take a look at something that happens during the build process.

2. Open the *wwwroot/js* folder, and you find two new files, *app.js* and the source map file, *app.js.map*. These files are generated by the TypeScript compiler.

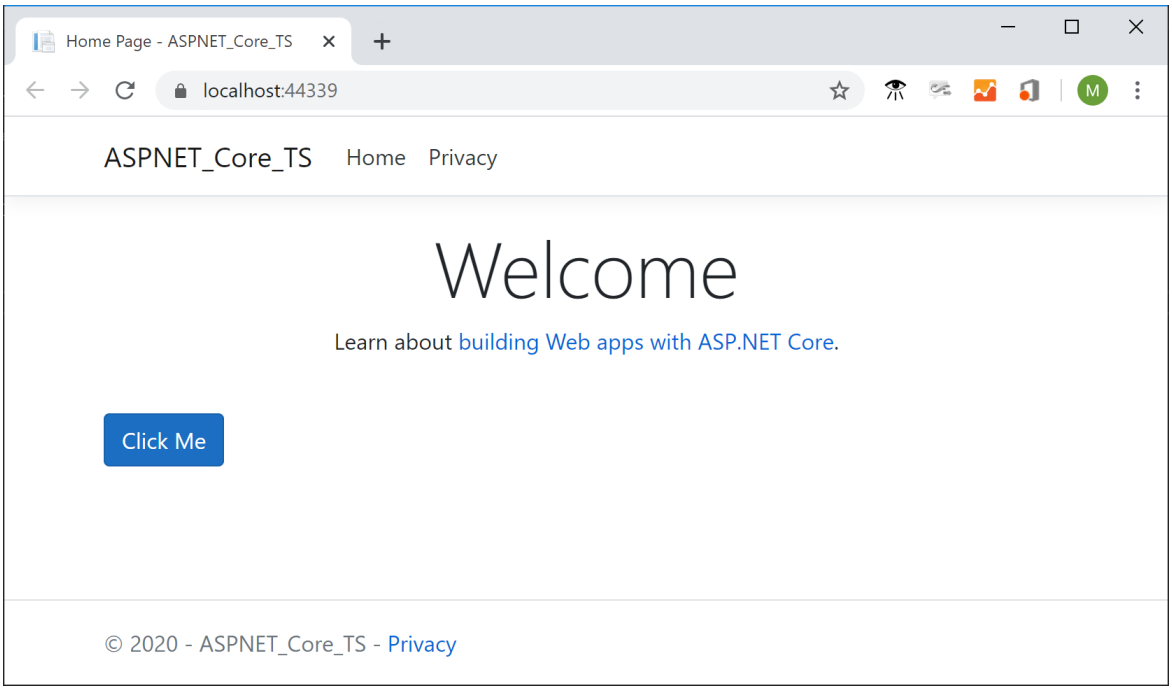
   Source map files are required for debugging.

Run the application

1. Press **F5** (**Debug > Start Debugging**) to run the application.

   The app opens in a browser.

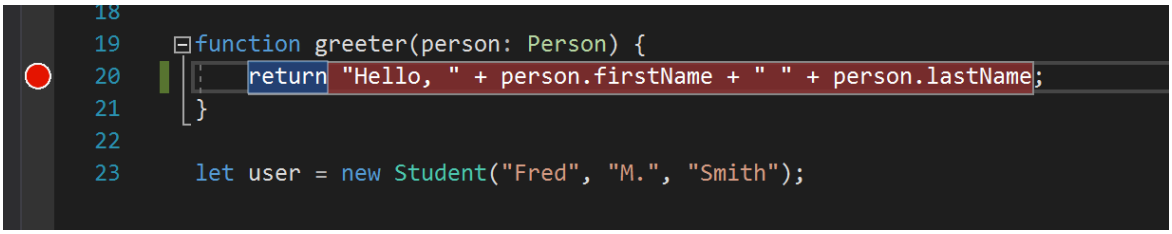
   In the browser window, you will see the **Welcome** heading and the **Click Me** button.



2. Click the button to display the message we specified in the TypeScript file.

Debug the application

1. Set a breakpoint in the `greeter` function in `app.ts` by clicking in the left margin in the code editor.



2. Press **F5** to run the application.

   You may need to respond to a message to enable script debugging.

   The application pauses at the breakpoint. Now, you can inspect variables and use debugger features.

Add TypeScript support for a third-party library

1. Follow instructions in npm package management to add a `package.json` file to your project. This adds npm support to your project.

   > **NOTE**
   > For ASP.NET Core projects, you can also use Library Manager or yarn instead of npm to install client-side JavaScript and CSS files.

2. In this example, add a TypeScript definition file for jQuery to your project. Include the following in your *package.json* file.

```
"devDependencies": {
   "@types/jquery": "3.3.33"
}
```

This adds TypeScript support for jQuery. The jQuery library itself is already included in the MVC project template (look under wwwroot/lib in Solution Explorer). If you are using a different template, you may need to include the jquery npm package as well.

3. If the package in Solution Explorer is not installed, right-click the npm node and choose **Restore Packages**.

> **NOTE**
>
> In some scenarios, Solution Explorer may indicate that an npm package is out of sync with *package.json* due to a known issue described here. For example, the package may appear as not installed when it is installed. In most cases, you can update Solution Explorer by deleting *package.json*, restarting Visual Studio, and re-adding the *package.json* file as described earlier in this article.

4. In Solution Explorer, right-click the scripts folder and choose **Add** > **New Item**.

5. Choose **TypeScript File**, type *library.ts*, and choose **Add**.

6. In *library.ts*, add the following code.

```
var jqtest = {
   showMsg: function (): void {
      let v: any = jQuery.fn.jquery.toString();
      let content: any = $("#ts-example-2")[0].innerHTML;
      alert(content.toString() + " " + v + "!!");
      $("#ts-example-2")[0].innerHTML = content + " " + v + "!!";
   }
};

jqtest.showMsg();
```

For simplicity, this code displays a message using jQuery and an alert.

With TypeScript type definitions for jQuery added, you get IntelliSense support on jQuery objects when you type a "." following a jQuery object, as shown here.

7. In _Layout.cshtml, update the script references to include `library.js`.

```
<script src="~/js/app.js"></script>
<script src="~/js/library.js"></script>
```

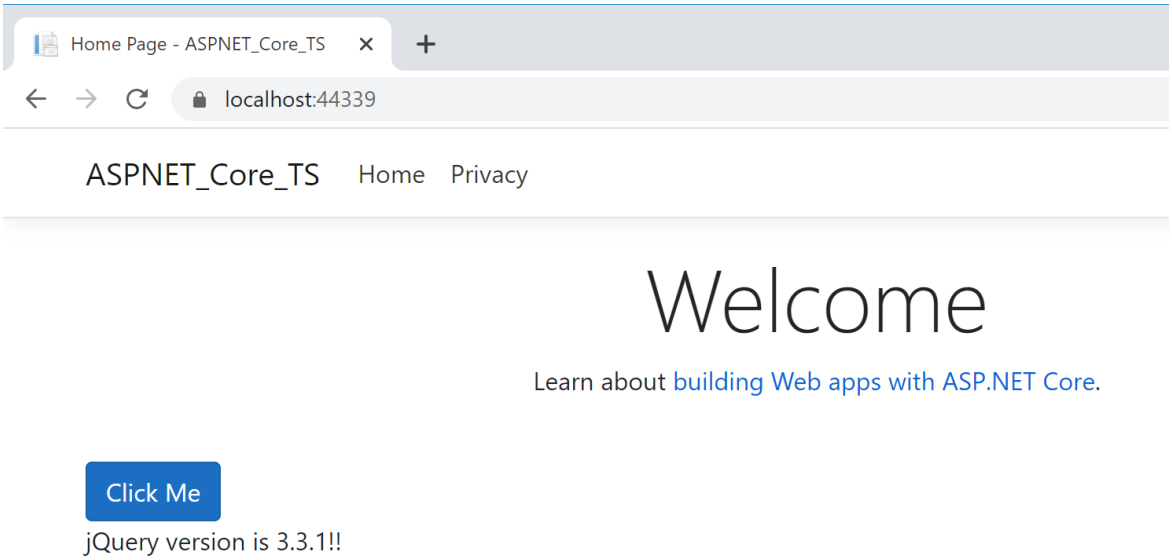
8. In Index.cshtml, add the following HTML to the end of the file.

```
<div>
   <p id="ts-example-2">jQuery version is:</p>
</div>
```

9. Press **F5** (**Debug** > **Start Debugging**) to run the application.

   The app opens in the browser.

   Click **OK** in the alert to see the page updated to **jQuery version is: 3.3.1!!**.



Next steps

You may want to learn more details about using TypeScript with ASP.NET Core. If you are interested in Angular programming in Visual Studio, you can use the Angular language service extension for Visual Studio.

ASP.NET Core and TypeScript

Angular language service extension

Publish a Node.js application to Azure (Linux App Service)

6/24/2022 • 6 minutes to read • Edit Online

Applies to: Visual Studio Visual Studio for Mac Visual Studio Code

This tutorial walks you through the task of creating a simple Node.js application and publishing it to Azure.

When publishing a Node.js application to Azure, there are several options. These include Azure App Service, a VM running an OS of your choosing, Azure Container Service (AKS) for management with Kubernetes, a Container Instance using Docker, and more. For more details on each of these options, see Compute.

For this tutorial, you deploy the app to Linux App Service. Linux App Service deploys a Linux Docker container to run the Node.js application (as opposed to the Windows App Service, which runs Node.js apps behind IIS on Windows).

This tutorial shows how to create a Node.js application starting from a template installed with the Node.js Tools for Visual Studio, push the code to a repository on GitHub, and then provision an Azure App Service via the Azure web portal so that you can deploy from the GitHub repository. To use the command-line to provision the Azure App Service and push the code from a local Git repository, see Create Node.js App.

In this tutorial, you learn how to:

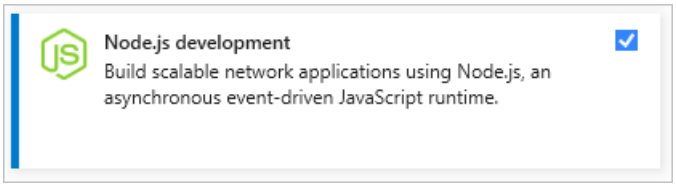
- Create a Node.js project
- Create a GitHub repository for the code
- Create a Linux App Service on Azure
- Deploy to Linux

Prerequisites

- You must have Visual Studio installed and the Node.js development workload.

  If you haven't already installed Visual Studio 2019, 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. Choose the **Node.js development** workload, then choose **Modify**.

- You must have the Node.js runtime installed.

  If you don't have it installed, install the LTS version from the Node.js website. In general, Visual Studio automatically detects the installed Node.js runtime. If it does not detect an installed runtime, you can configure your project to reference the installed runtime in the properties page (after you create a project, right-click the project node and choose **Properties**).

Create a Node.js project to run in Azure

1. Open Visual Studio.
2. Create a new TypeScript Express app.

   Press **Esc** to close the start window. Type **Ctrl + Q** to open the search box, type **Node.js**, then choose **Create new Basic Azure Node.js Express 4 application** (TypeScript). In the dialog box that appears, choose **Create**.

   If you don't see the **Basic Azure Node.js Express 4 application** project template, you must add the **Node.js development** workload. For detailed instructions, see the Prerequisites.

   Visual Studio creates the project and opens it in Solution Explorer (right pane).
3. Press **F5** to build and run the app, and make sure that everything is running as expected.
4. Select **File** > **Add to source control** to create a local Git repository for the project.

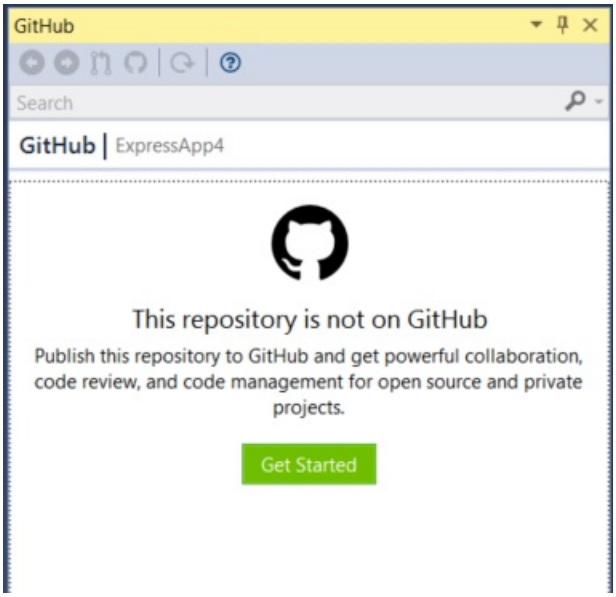
   At this point, a Node.js app using the Express framework and written in TypeScript is working and checked in to local source control.
5. Edit the project as desired before proceeding to the next steps.

Push code from Visual Studio to GitHub

To set up GitHub for Visual Studio:

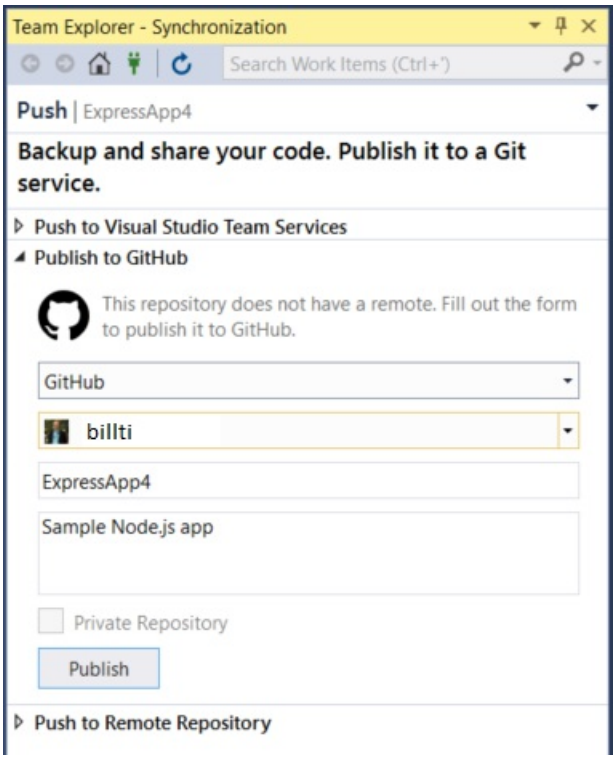
1. Make sure the GitHub Extension for Visual Studio is installed and enabled using the menu item **Tools** > **Extensions and Updates**.
2. From the menu select **View** > **Other Windows** > **GitHub**.

   The GitHub window opens.
3. If you don't see the **Get Started** button in the GitHub window, click **File** > **Add to Source Control** and wait for the UI to update.



4. Click **Get started**.

   If you are already connected to GitHub, the toolbox appears similar to the following illustration.



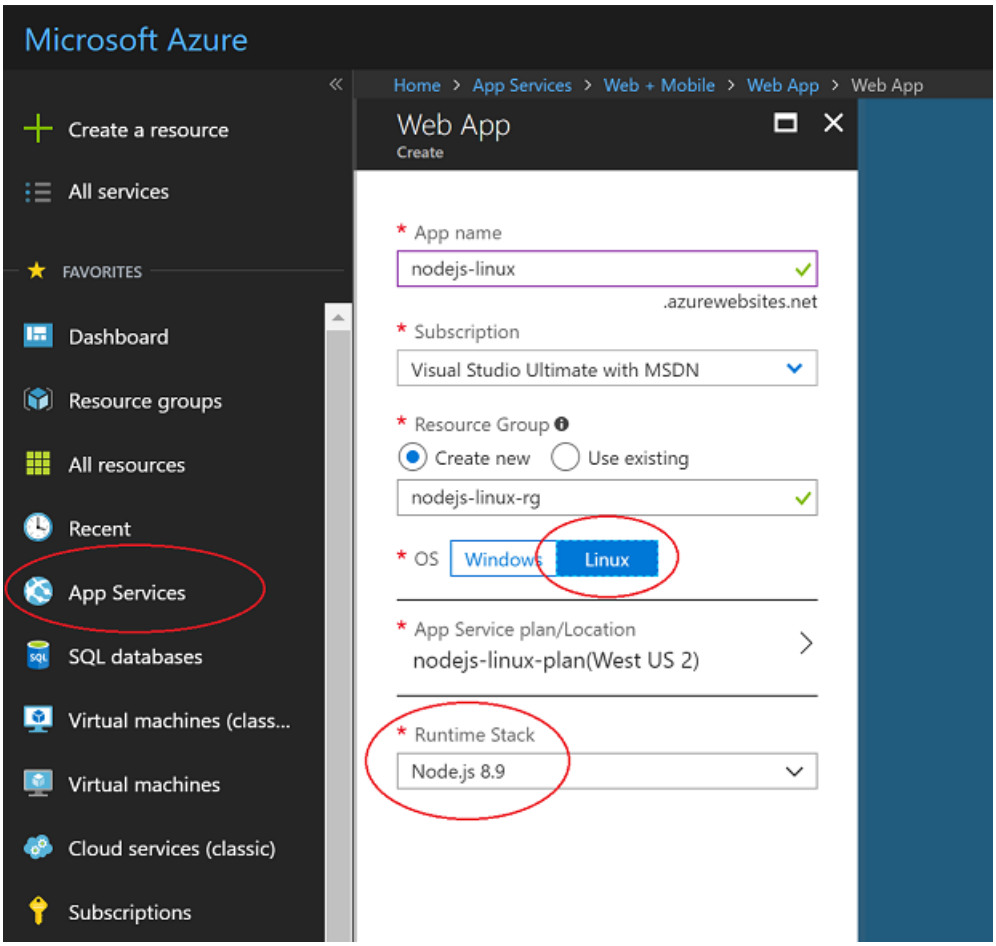
5. Complete the fields for the new repository to publish, and then click **Publish**.

   After a few moments, a banner stating "Repository created successfully" appears.

   In the next section, you learn how to publish from this repository to an Azure App Service on Linux.

Create a Linux App Service in Azure

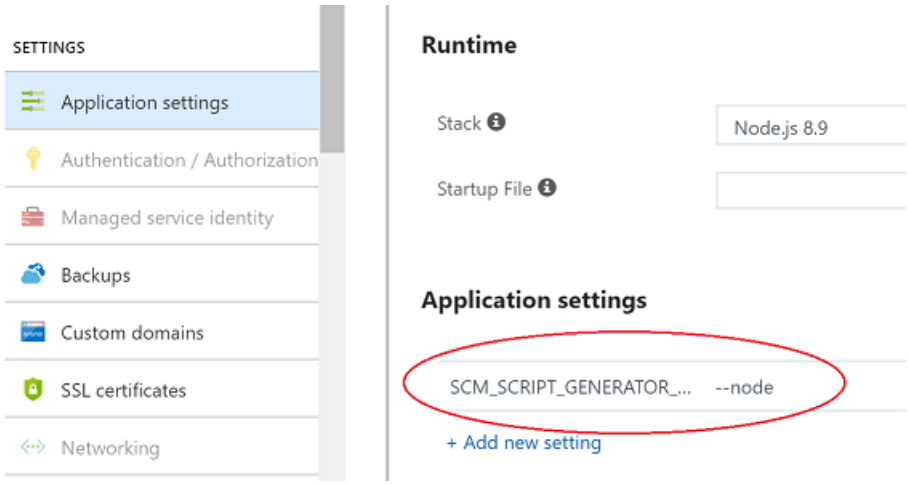
1. Sign in to the Azure portal.
2. Select **App Services** from the list of services on the left, and then click **Add**.
3. If required, create a new Resource Group and App Service plan to host the new app.
4. Make sure to set the **OS** to **Linux**, and set **Runtime Stack** to the required Node.js version, as shown in the illustration.



5. Click **Create** to create the App Service.

   It may take a few minutes to deploy.

6. After it is deployed, go to the **Application settings** section, and add a setting with a name of `SCM_SCRIPT_GENERATOR_ARGS` and a value of `--node`.

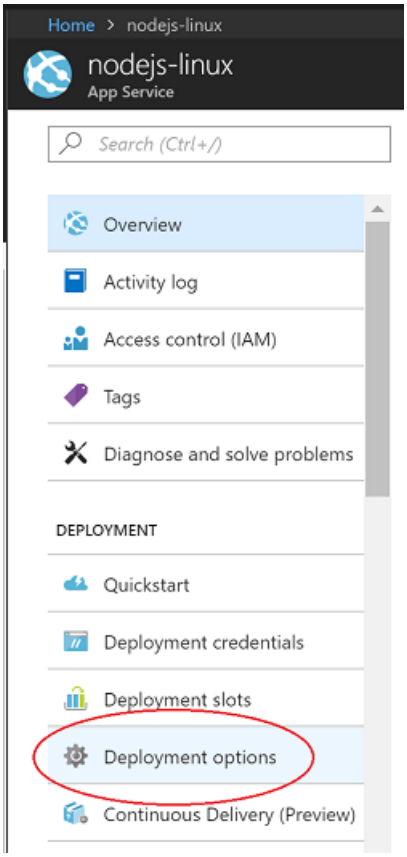


> **WARNING**
>
> The App Service deployment process uses a set of heuristics to determine which type of application to try and run. If a *.sln* file is detected in the deployed content, it will assume an MSBuild based project is being deployed. The setting added above overrides this logic and specifies explicitly that this is a Node.js application. Without this setting, the Node.js application will fail to deploy if the *.sln* file is part of the repository being deployed to the App Service.

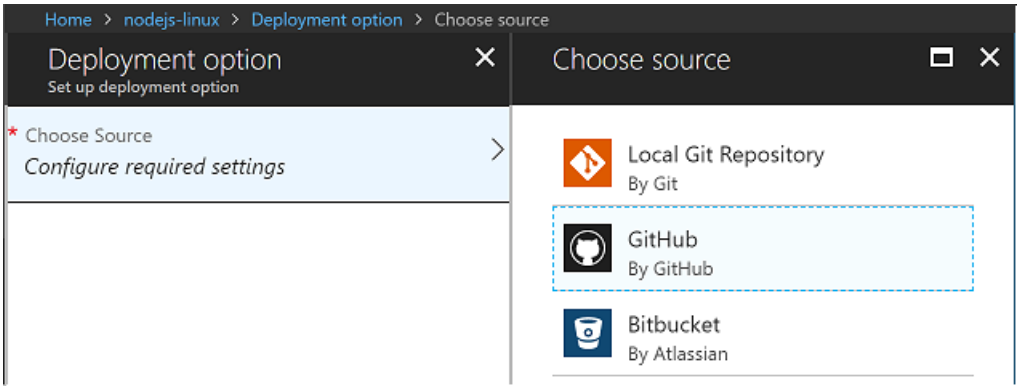
7. Under **Application settings**, add another setting with a name of `WEBSITE_NODE_DEFAULT_VERSION` and a

value of `8.9.0`.

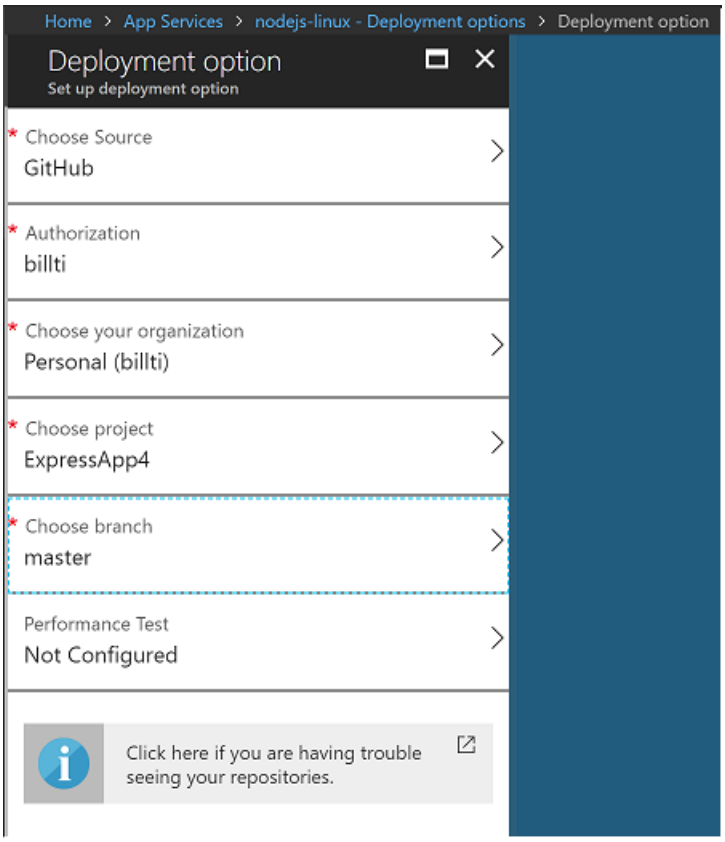
8. After it is deployed, open the App Service and select **Deployment options**.



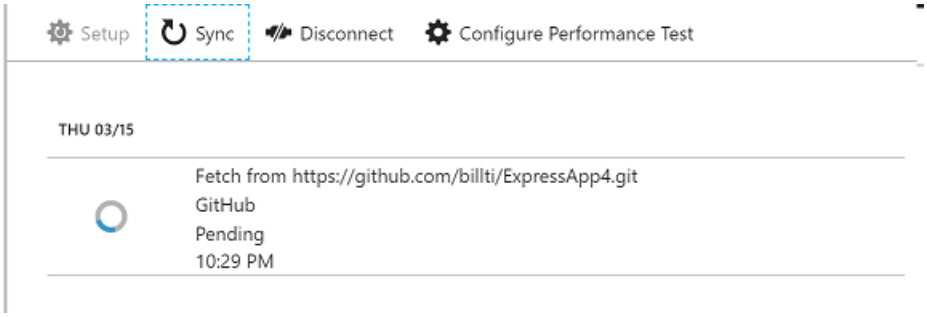
9. Click **Choose source**, and then choose **GitHub**, and then configure any required permissions.



10. Select the repository and branch to publish, and then select **OK**.



The **deployment options** page appears while syncing.



Once it is finished syncing, a check mark will appear.

The site is now running the Node.js application from the GitHub repository, and it is accessible at the URL created for the Azure App Service (by default the name given to the Azure App Service followed by ".azurewebsites.net").

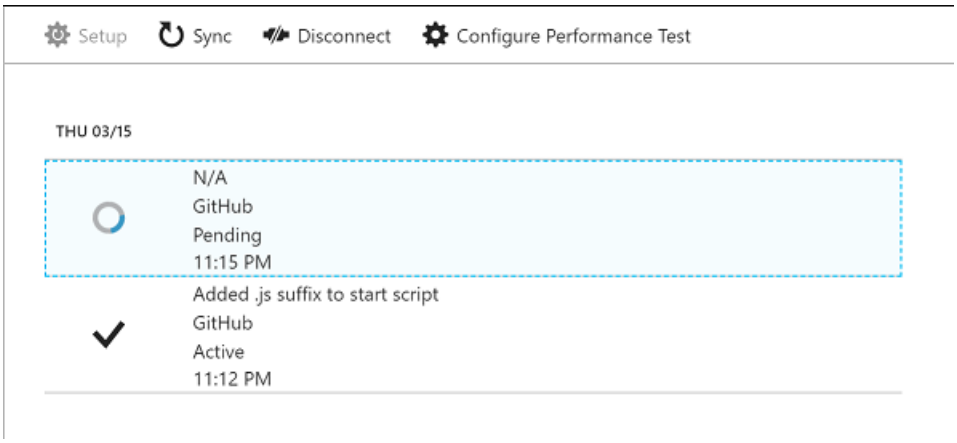
Modify your app and push changes

1. Add the code shown here in *app.ts* after the line `app.use('/users', users);`. This adds a REST API at the URL */api*.

```
app.use('/api', (req, res, next) => {
    res.json({"result": "success"});
});
```

2. Build the code and test it locally, then check it in and push to GitHub.

   In the Azure portal, it takes a few moments to detect changes in the GitHub repo, and then a new sync of the deployment starts. This looks similar to the following illustration.



3. Once deployment is complete, navigate to the public site and append */api* to the URL. The JSON response gets returned.

Troubleshooting

- If the node.exe process dies (that is, an unhandled exception occurs), the container restarts.
- When the container starts up, it runs through various heuristics to figure out how to start the Node.js process. Details of the implementation can be seen at generateStartupCommand.js.
- You can connect to the running container via SSH for investigations. This is easily done using the Azure portal. Select the App Service, and scroll down the list of tools until reaching **SSH** under the **Development Tools** section.
- To aid in troubleshooting, go to the **Diagnostics logs** settings for the App Service, and change the **Docker Container logging** setting from **Off** to **File System**. Logs are created in the container under */home/LogFiles/*_docker.log*, and can be accessed on the box using SSH or FTP(S).
- A custom domain name may be assigned to the site, rather than the *.azurewebsites.net URL assigned by default. For more details, see the topic Map Custom Domain.
- Deploying to a staging site for further testing before moving into production is a best practice. For details on how to configure this, see the topic Create staging environments.
- See the App Service on Linux FAQ for more commonly asked questions.

Next steps

In this tutorial, you learned how create a Linux App Service and deploy a Node.js application to the service. You may want to learn more about Linux App Service.

Linux App Service

Learn to use the code editor for JavaScript

6/24/2022 • 5 minutes to read • Edit Online

Applies to: Visual Studio Visual Studio for Mac Visual Studio Code

In this short introduction to the code editor in Visual Studio, we'll 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. Depending on the type of app development you're doing, you may need to install the **Node.js development workload** with Visual Studio. For more information in getting the language service for TypeScript, see TypeScript support.

This article assumes you're already familiar with JavaScript development. If you aren't, we suggest you look at a tutorial such as Create a Node.js and Express app first.

Add a new project file

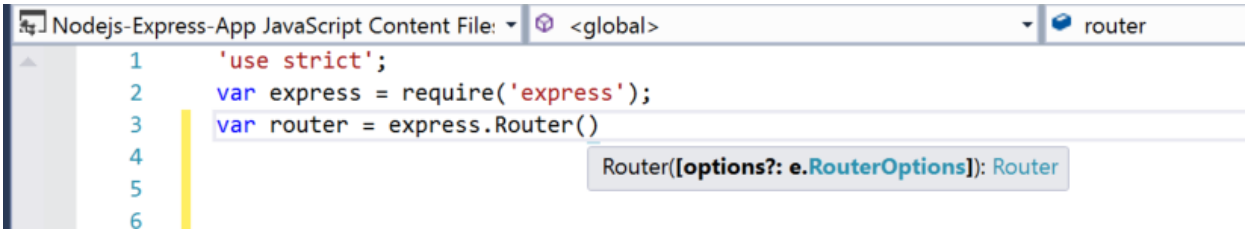
You can use the IDE to add new files to your project.

1. With your project open in Visual Studio, right-click on a folder or your project node in Solution Explorer (right pane), and choose **Add** > **New Item**.
2. In the **New File** dialog box, under the **General** category, choose the file type that you want to add, such as **JavaScript File**, and then choose **Open**.

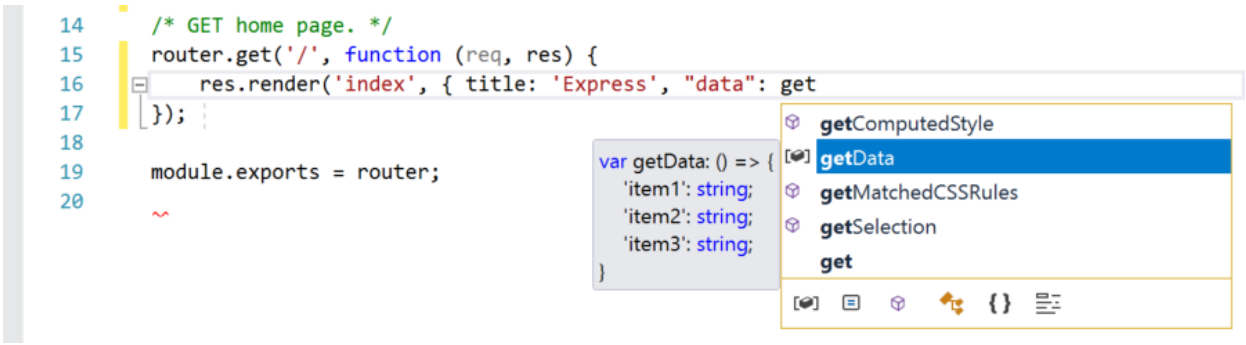
   The new file gets added to your project and it opens in the editor.

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. In the following code, when you type `Router()`, you see the argument types that you can pass. This is called signature help.



You can also use IntelliSense to complete a word after you type enough characters to disambiguate it. If you put your cursor after the `data` string in the following code and type `get`, IntelliSense will show you functions defined earlier in the code or defined in a third-party library that you've added to your project.



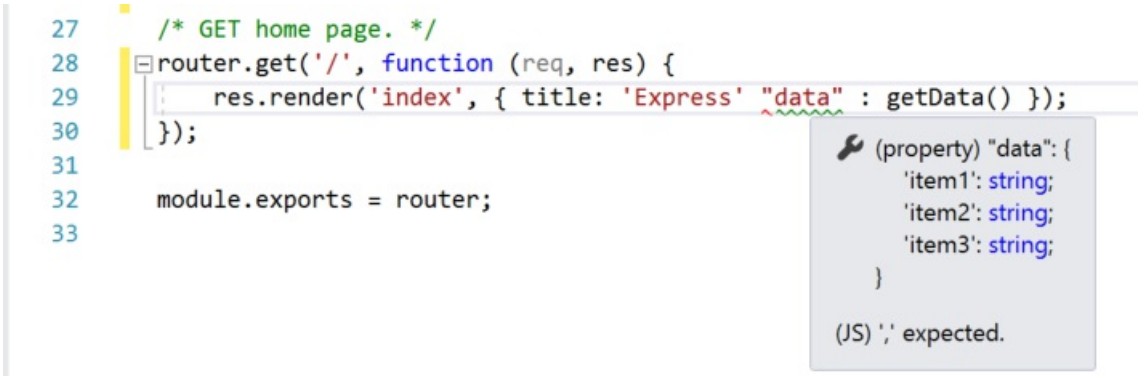
IntelliSense can also show you information about types when you hover over programming elements.

To provide IntelliSense information, the language service can use TypeScript *d.ts* files and JSDoc comments. For most common JavaScript libraries, *d.ts* files are automatically acquired. For more details about how IntelliSense information is acquired, see JavaScript IntelliSense. If you are interested in AngularJS programming in Visual Studio, you can use the AngularJS language service extension for Visual Studio to get IntelliSense.

Check syntax

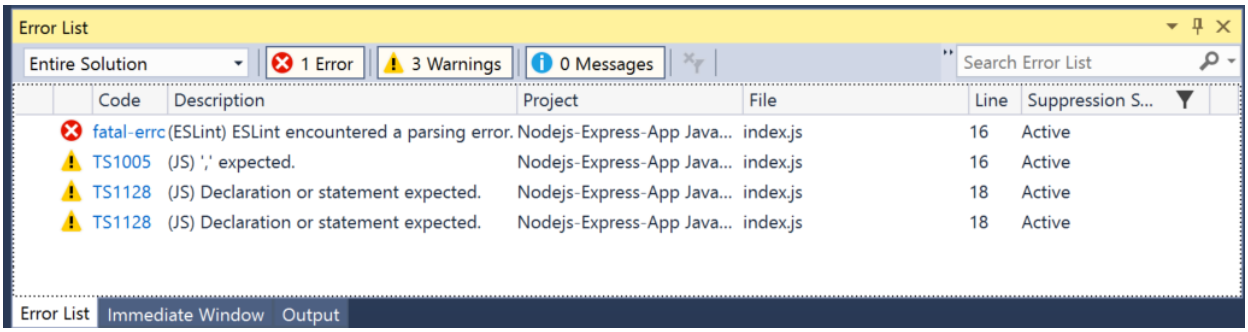
The language service uses ESLint to provide syntax checking and linting. If you need to set options for syntax checking in the editor, select **Tools** > **Options** > **JavaScript/TypeScript** > **Linting**. The linting options point you to the global ESLint configuration file.

In the following code, you see green syntax highlighting (green squiggles) on the expression. Hover over the syntax highlighting.



The last line of this message tells you that the language service expected a comma (`,`). The green squiggle indicates a warning. Red squiggles indicate an error.

In the lower pane, you can click the **Error List** tab to see the warning and description along with the filename and line number.



You can fix this code by adding the comma (`,`) before `"data"`.

For additional information on linting, see Linting.

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.

Select one or more lines of code in the editor 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 JavaScript comment characters `//` are added to the beginning of each selected line to comment out the code.

Collapse code blocks

If you need to unclutter your view of some regions of code, you can collapse it. Choose the small gray box with the minus sign inside it in the margin of the first line of a function. Or, if you're a keyboard user, place the cursor anywhere in the constructor code and press **Ctrl**+**M**, **Ctrl**+**M**.

```
18  var getData = function () {
19      var data = {
20          'item1': 'http://public-domain-photos.com/free-stock-photos-1/flowers/cactus-76.jpg',
21          'item2': 'http://public-domain-photos.com/free-stock-photos-1/flowers/cactus-77.jpg',
22          'item3': 'http://public-domain-photos.com/free-stock-photos-1/flowers/cactus-78.jpg'
23      }
24      return data;
25  }
```

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 functions or entire classes.

View definitions

The Visual Studio editor makes it easy to inspect the definition of a type, function, etc. One way is to navigate to the file that contains the definition, for example by choosing **Go to Definition** anywhere the programming element 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 `render` method in the example below.

Right-click on `render` and choose **Peek Definition** from the content menu. Or, press **Alt**+**F12**.

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

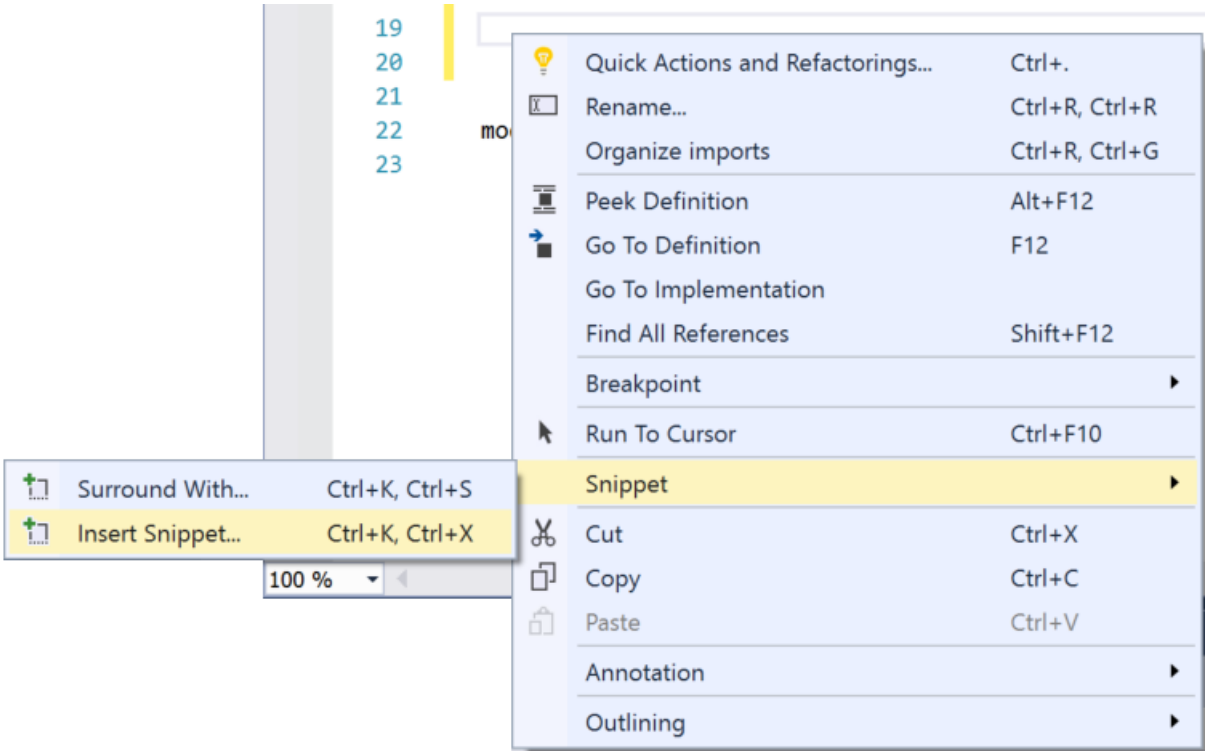
```
14  /* GET home page. */
15  router.get('/', function (req, res) {
16      res.render('index', { title: 'Express' "data" : getData() });
                                                              index.d.ts
        808          */
        809         render(view: string, options?: Object, callback?: (err: Error, html: string) => void): void;
        810         render(view: string, callback?: (err: Error, html: string) => void): void;
        811
        812         locals: any;
        813
        814         charset: string;
        815
        816         /**
        817          * Adds the field to the Vary response header, if it is not there already.
        818          * Examples:
17  });
```

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

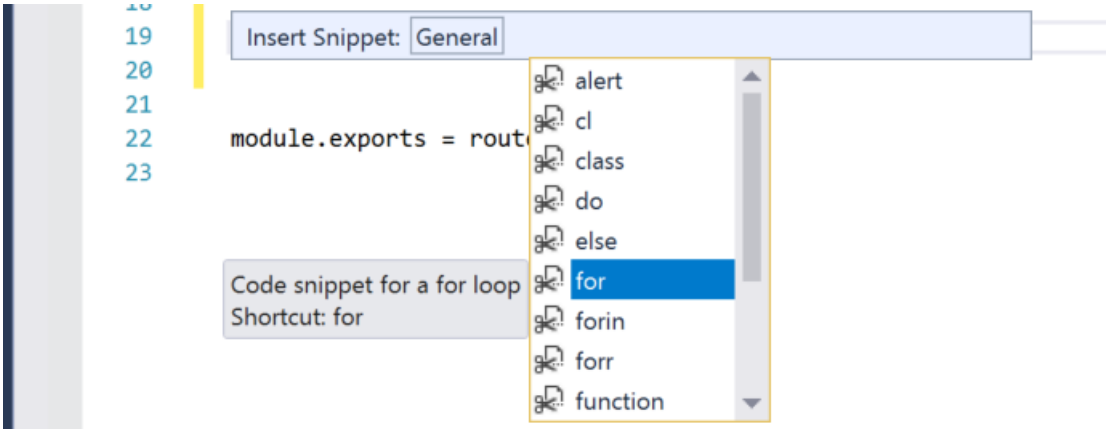
Use code snippets

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 JavaScript. Let's add a `for` loop to your code file.

Place your cursor where you want to insert the snippet, right-click and choose **Snippet** > **Insert Snippet**.



An **Insert Snippet** box appears in the editor. Choose **General** and then double-click the **for** item in the list.



This adds the `for` loop snippet to your code:

```
for (var i = 0; i < length; i++) {

}
```

You can look at the available code snippets for your language by choosing **Edit** > **IntelliSense** > **Insert Snippet**, and then choosing your language's folder.

See also

- Code snippets
- Navigate code

- Outlining
- Go To Definition and Peek Definition
- Refactoring
- Use IntelliSense

JavaScript IntelliSense

6/24/2022 • 4 minutes to read • Edit Online

Applies to: Visual Studio | Visual Studio for Mac | Visual Studio Code

Visual Studio provides a powerful JavaScript editing experience right out of the box. Powered by a TypeScript based language service, Visual Studio delivers richer IntelliSense, support for modern JavaScript features, and improved productivity features such as Go to Definition, refactoring, and more.

> **NOTE**
> Starting in Visual Studio 2017, the JavaScript language service uses a new engine for the language service (called "Salsa"). Details are included in this article, and you can also read this blog post. The new editing experience also mostly applies to Visual Studio Code. See the VS Code docs for more info.

For more information about the general IntelliSense functionality of Visual Studio, see Using IntelliSense.

What's new in the JavaScript language service in Visual Studio 2017

Starting in Visual Studio 2017, JavaScript IntelliSense displays a lot more information on parameter and member lists. This new information is provided by the TypeScript language service, which uses static analysis behind the scenes to better understand your code.

TypeScript uses several sources to build up this information:

- IntelliSense based on type inference
- IntelliSense based on JSDoc
- IntelliSense based on TypeScript declaration files
- Automatic acquisition of type definitions

IntelliSense based on type inference

In JavaScript, most of the time there is no explicit type information available. Luckily, it is usually fairly easy to figure out a type given the surrounding code context. This process is called type inference.

For a variable or property, the type is typically the type of the value used to initialize it or the most recent value assignment.

```
var nextItem = 10;
nextItem; // here we know nextItem is a number

nextItem = "box";
nextItem; // now we know nextItem is a string
```

For a function, the return type can be inferred from the return statements.

For function parameters, there is currently no inference, but there are ways to work around this using JSDoc or TypeScript *.d.ts* files (see later sections).

Additionally, there is special inference for the following:

- "ES3-style" classes, specified using a constructor function and assignments to the prototype property.

- CommonJS-style module patterns, specified as property assignments on the `exports` object, or assignments to the `module.exports` property.

```
function Foo(param1) {
    this.prop = param1;
}
Foo.prototype.getIt = function () { return this.prop; };
// Foo will appear as a class, and instances will have a 'prop' property and a 'getIt' method.

exports.Foo = Foo;
// This file will appear as an external module with a 'Foo' export.
// Note that assigning a value to "module.exports" is also supported.
```

IntelliSense based on JSDoc

Where type inference does not provide the desired type information (or to support documentation), type information may be provided explicitly via JSDoc annotations. For example, to give a partially declared object a specific type, you can use the `@type` tag as shown below:

```
/**
 * @type {{a: boolean, b: boolean, c: number}}
 */
var x = {a: true};
x.b = false;
x. // <- "x" is shown as having properties a, b, and c of the types specified
```

As mentioned, function parameters are never inferred. However, using the JSDoc `@param` tag you can add types to function parameters as well.

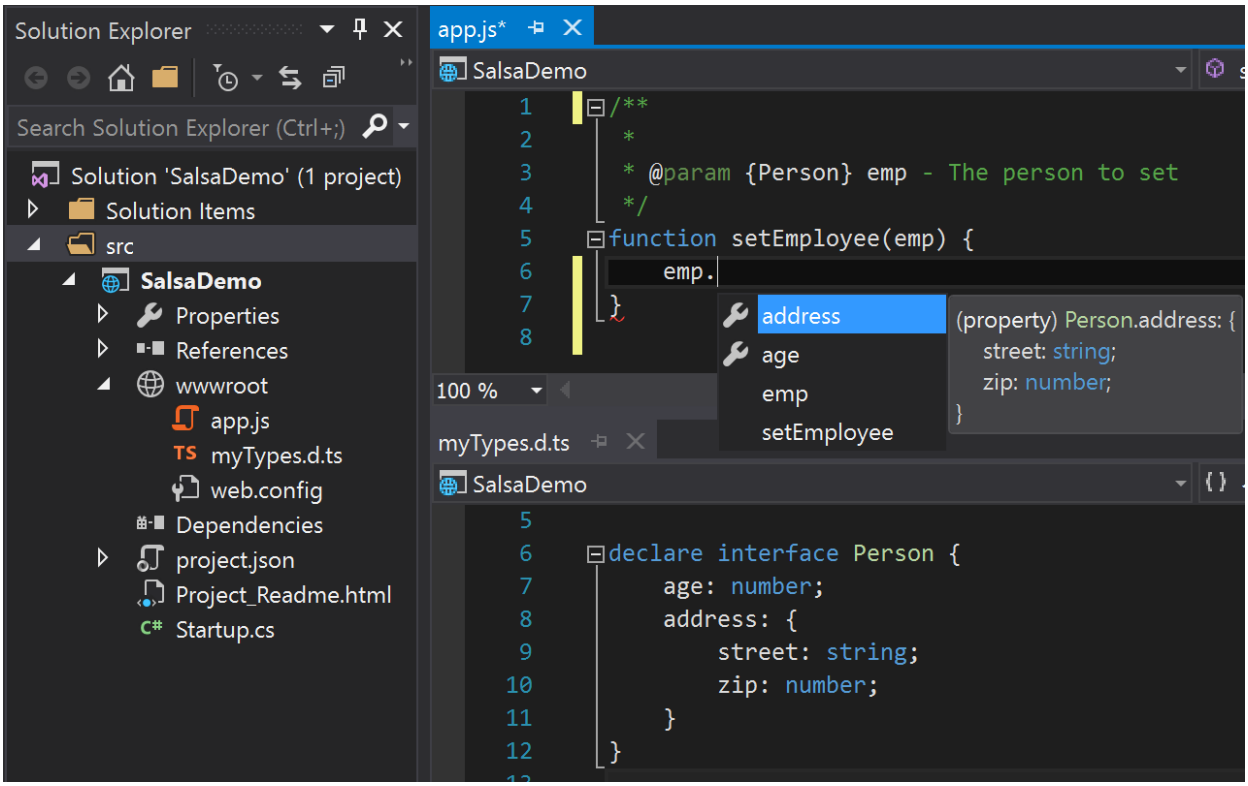
```
/**
 * @param {string} param1 - The first argument to this function
 */
function Foo(param1) {
    this.prop = param1; // "param1" (and thus "this.prop") are now of type "string".
}
```

See JSDoc support in JavaScript for the JsDoc annotations currently supported.

IntelliSense based on TypeScript declaration files

Because JavaScript and TypeScript are now based on the same language service, they are able to interact in a richer way. For example, JavaScript IntelliSense can be provided for values declared in a *.d.ts* file (see TypeScript documentation), and types such as interfaces and classes declared in TypeScript are available for use as types in JsDoc comments.

Below, we show a simple example of a TypeScript definition file providing such type information (via an interface) to a JavaScript file in the same project (using a `JsDoc` tag).



Automatic acquisition of type definitions

In the TypeScript world, most popular JavaScript libraries have their APIs described by *.d.ts* files, and the most common repository for such definitions is on DefinitelyTyped.

By default, the Salsa language service will try to detect which JavaScript libraries are in use and automatically download and reference the corresponding *.d.ts* file that describes the library in order to provide richer IntelliSense. The files are downloaded to a cache located under the user folder at *%LOCALAPPDATA%\Microsoft\TypeScript*.

> **NOTE**
> This feature is **disabled** by default if using a *tsconfig.json* configuration file, but may be set to enabled as outlined further below.

Currently auto-detection works for dependencies downloaded from npm (by reading the *package.json* file), Bower (by reading the *bower.json* file), and for loose files in your project that match a list of roughly the top 400 most popular JavaScript libraries. For example, if you have *jquery-1.10.min.js* in your project, the file *jquery.d.ts* will be fetched and loaded in order to provide a better editing experience. This *.d.ts* file will have no impact on your project.

See also

- Using IntelliSense
- JavaScript support (Visual Studio for Mac)

Compile TypeScript code (Node.js)

6/24/2022 • 3 minutes to read • Edit Online

Applies to: Visual Studio | Visual Studio for Mac | Visual Studio Code

You can add TypeScript support to your projects using the TypeScript SDK or by using the npm. The TypeScript SDK is available by default in the Visual Studio installer.

For projects developed in Visual Studio 2019, we encourage you to use the TypeScript npm package for greater portability across different platforms and environments.

For ASP.NET Core projects, it's recommended that you use the NuGet package instead.

Add TypeScript support using npm

The TypeScript npm package adds TypeScript support. When the npm package for TypeScript 2.1 or higher is installed into your project, the corresponding version of the TypeScript language service gets loaded in the editor.

1. Follow instructions to install the Node.js development workload and the Node.js runtime.

   For a simple Visual Studio integration, create your project using one of the Node.js TypeScript templates, such as the Blank Node.js Web Application template. Else, use either a Node.js JavaScript template included with Visual Studio and follow instructions here. Or, use an Open Folder project.

2. If your project doesn't already include it, install the TypeScript npm package.

   From Solution Explorer (right pane), open the *package.json* in the project root. The packages listed correspond to packages under the npm node in Solution Explorer. For more information, see Manage npm packages.

   For a Node.js project, you can install the TypeScript npm package using the command line or the IDE. To install using the IDE, right-click the npm node in Solution Explorer, choose **Install New npm package**, search for **TypeScript**, and install the package.

   Check the **npm** option in the **Output** window to see package installation progress. The installed package shows up under the **npm** node in Solution Explorer.

3. If your project doesn't already include it, add a *tsconfig.json* file to your project root. To add the file, right-click the project node and choose **Add > New Item**. Choose the **TypeScript JSON Configuration File**, and then click **Add**.

   Visual Studio adds the *tsconfig.json* file to the project root. You can use this file to configure options for the TypeScript compiler.

4. Open *tsconfig.json* and update to set the compiler options that you want.

   An example of a simple *tsconfig.json* file follows.

```
{
  "compilerOptions": {
    "noImplicitAny": false,
    "noEmitOnError": true,
    "removeComments": false,
    "sourceMap": true,
    "target": "es5",
    "outDir": "dist"
  },
  "include": [
    "scripts/**/*"
  ]
}
```

In this example:

- *include* tells the compiler where to find TypeScript (*.ts) files.
- *outDir* option specifies the output folder for the plain JavaScript files that are transpiled by the TypeScript compiler.
- *sourceMap* option indicates whether the compiler generates *sourceMap* files.

The previous configuration provides only a basic introduction to configuring TypeScript. For information on other options, see tsconfig.json.

Build the application

1. Add TypeScript (*.ts*) or TypeScript JSX (*.tsx*) files to your project, and then add TypeScript code. A simple example of TypeScript follows:

```
let message: string = 'Hello World';
console.log(message);
```

2. In *package.json*, add support for Visual Studio build and clean commands using the following scripts.

```
"scripts": {
  "build": "tsc --build",
  "clean": "tsc --build --clean"
},
```

To build using a third-party tool like webpack, you can add a command-line build script to your *package.json* file:

```
"scripts": {
   "build": "webpack-cli app.tsx --config webpack-config.js"
}
```

For an example of using webpack with React and a webpack configuration file, see Create a web app with Node.js and React.

For an example of using Vue.js with TypeScript, see Create a Vue.js application.

3. If you need to configure options such as the startup page, path to the Node.js runtime, application port, or runtime arguments, right-click the project node in Solution Explorer, and choose **Properties**.

> **NOTE**
>
> When configuring third-party tools, Node.js projects don't use the paths that are configured under **Tools** > **Options** > **Projects and solutions** > **Web Package Management** > **External Web Tools**. These settings are used for other project types.

4. Choose **Build > Build Solution**.

   The app builds automatically when you run it. However, the following might occur during the build process:

   If you generated source maps, open the folder specified in the *outDir* option and you find the generated *.js file(s) along with the generated *js.map file(s).

   Source map files are required for debugging.

Run the application

For instructions to run the app after you compile it, see Create a Node.js and Express app.

Automate build tasks

You can use Task Runner Explorer in Visual Studio to help automate tasks for third-party tools like npm and webpack.

- NPM Task Runner - Adds support for npm scripts defined in *package.json*. Supports yarn.
- Webpack Task Runner - Adds support for webpack.

Compile TypeScript code (ASP.NET Core)

6/24/2022 • 4 minutes to read • Edit Online

Applies to: Visual Studio | Visual Studio for Mac | Visual Studio Code

You can add TypeScript support to your projects using the TypeScript SDK, available by default in the Visual Studio installer or by using the NuGet package. For projects developed in Visual Studio 2019, we encourage you to use the TypeScript NuGet for greater portability across different platforms and environments.

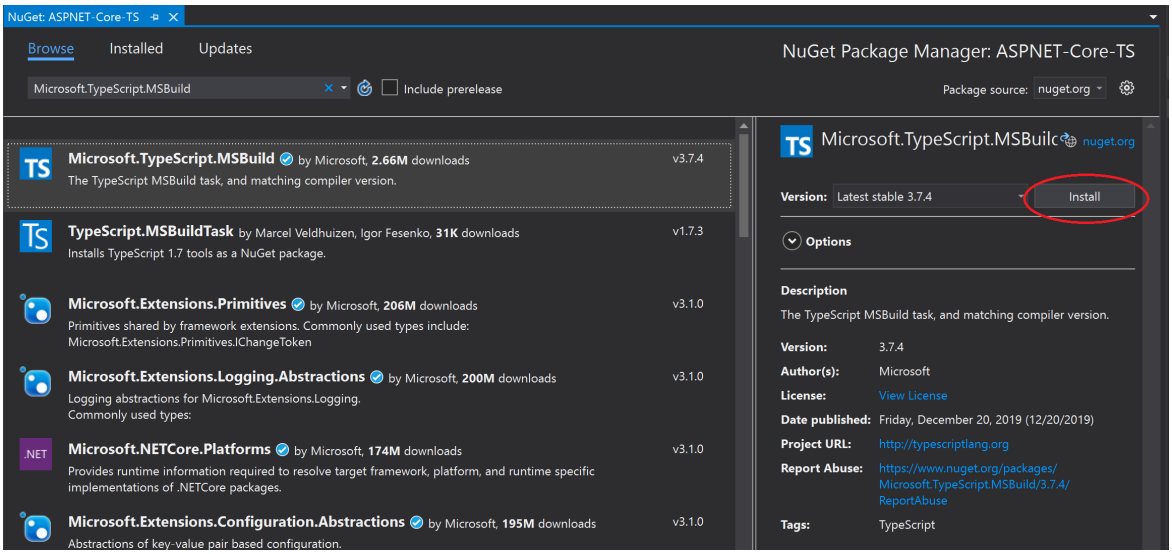
For ASP.NET Core projects, one common usage for the NuGet package is to compile TypeScript using the .NET Core CLI. Unless you manually edit your project file to import build targets from a TypeScript SDK installation, the NuGet package is the only way to enable TypeScript compilation using .NET Core CLI commands such as `dotnet build` and `dotnet publish`. Also, for MSBuild integration with ASP.NET Core and TypeScript, choose the NuGet package over the npm package.

Add TypeScript support with NuGet

The TypeScript NuGet package adds TypeScript support. When the NuGet package for TypeScript 3.2 or higher is installed into your project, the corresponding version of the TypeScript language service gets loaded in the editor.

If Visual Studio is installed, then the node.exe bundled with it will automatically be picked up by Visual Studio. If you don't have Node.js installed, we recommend you install the LTS version from the Node.js website.

1. Open your ASP.NET Core project in Visual Studio.
2. In Solution Explorer (right pane). right-click the project node and choose **Manage NuGet Packages**. In the **Browse** tab, search for **Microsoft.TypeScript.MSBuild**, and then click **Install** on the right to install the package.



Visual Studio adds the NuGet package under the **Dependencies** node in Solution Explorer. The following package reference gets added to your *.csproj file.

```
<PackageReference Include="Microsoft.TypeScript.MSBuild" Version="3.9.7">
   <PrivateAssets>all</PrivateAssets>
   <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
</PackageReference>
```

3. Right-click the project node and choose **Add > New Item**. Choose the **TypeScript JSON Configuration File**, and then click **Add**.

   Visual Studio adds the *tsconfig.json* file to the project root. You can use this file to configure options for the TypeScript compiler.

4. Open *tsconfig.json* and update to set the compiler options that you want.

   The following is an example of a simple *tsconfig.json* file.

   ```
   {
     "compilerOptions": {
       "noImplicitAny": false,
       "noEmitOnError": true,
       "removeComments": false,
       "sourceMap": true,
       "target": "es5",
       "outDir": "wwwroot/js"
     },
     "include": [
       "scripts/**/*"
     ]
   }
   ```

   In this example:

   - *include* tells the compiler where to find TypeScript (*.ts) files.
   - *outDir* option specifies the output folder for the plain JavaScript files that are transpiled by the TypeScript compiler.
   - *sourceMap* option indicates whether the compiler generates *sourceMap* files.

   The previous configuration provides only a basic introduction to configuring TypeScript. For information on other options, see tsconfig.json.

Build the application

1. Add TypeScript (*.ts*) or TypeScript JSX (*.tsx*) files to your project, and then add TypeScript code. For a simple example of TypeScript, use the following:

   ```
   let message: string = 'Hello World';
   console.log(message);
   ```

2. If you are using an older non-SDK style project, follow instructions in Remove default imports before building.

3. Choose **Build > Build Solution**.

   Although the app builds automatically when you run it, we want to take a look at something that happens during the build process:

   If you generated source maps, open the folder specified in the *outDir* option and you find the generated *.js file(s) along with the generated *js.map file(s).

   Source map files are required for debugging.

4. If you want to compile every time you save the project, use the *compileOnSave* option in *tsconfig.json*.

```
{
   "compileOnSave":  true,
   "compilerOptions": {
   }
}
```

For an example of using gulp with the Task Runner to build your app, see ASP.NET Core and TypeScript.

If you run into issues where Visual Studio is using a version of Node.js or a third-party tool that is different than what the version you expected, you may need to set the path for Visual Studio to use. Choose **Tools** > **Options**. Under **Projects and solutions**, choose **Web Package Management** > **External Web Tools**.

Run the application

For instructions to run the app after you compile it, see Create a Node.js and Express app.

NuGet package structure details

`Microsoft.TypeScript.MSBuild.nupkg` contains two main folders:

- *build* folder

  Two files are located in this folder. Both are entry points - for the main TypeScript target file and props file respectively.

  1. *Microsoft.TypeScript.MSBuild.targets*

     This file sets variables that specify the run-time platform, such as a path to *TypeScript.Tasks.dll*, before importing *Microsoft.TypeScript.targets* from the *tools* folder.

  2. *Microsoft.TypeScript.MSBuild.props*

     This file imports *Microsoft.TypeScript.Default.props* from the *tools* folder and sets properties indicating that the build has been initiated through NuGet.

- *tools* folder

  Package versions prior to 2.3 only contain a tsc folder. *Microsoft.TypeScript.targets* and *TypeScript.Tasks.dll* are located at the root level.

  In package versions 2.3 and later, the root level contains `Microsoft.TypeScript.targets` and `Microsoft.TypeScript.Default.props`. For more details on these files, see MSBuild Configuration.

  Additionally, the folder contains three subfolders:

  1. *net45*

     This folder contains `TypeScript.Tasks.dll` and other DLLs on which it depends. When building a project on a Windows platform, MSBuild uses the DLLs from this folder.

  2. *netstandard1.3*

     This folder contains another version of `TypeScript.Tasks.dll`, which is used when building projects on a non-Windows machine.

  3. *tsc*

     This folder contains `tsc.js`, `tsserver.js` and all dependency files required to run them as node scripts.

> **NOTE**
> If Visual Studio is installed, then the *node.exe* bundled with it will automatically be picked up. Otherwise Node.js must be installed on the machine.

Versions prior to 3.1 contained a `tsc.exe` executable to run the compilation. In version 3.1, this was removed in favor of using `node.exe`.

Remove default imports

In older ASP.NET Core projects that use the non-SDK-style format, you may need to remove some project file elements.

If you are using the NuGet package for MSBuild support for a project, the project file must not import `Microsoft.TypeScript.Default.props` or `Microsoft.TypeScript.targets`. The files get imported by the NuGet package, so including them separately may cause unintended behavior.

1. Right-click the project and choose **Unload Project**.
2. Right-click the project and choose **Edit <*project file name*>**.

   The project file opens.
3. Remove references to `Microsoft.TypeScript.Default.props` and `Microsoft.TypeScript.targets`.

   The imports to remove look similar to the following:

```
<Import

Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Microso
ft.TypeScript.Default.props"

Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScri
pt\Microsoft.TypeScript.Default.props')" />

<Import

Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScript\Microso
ft.TypeScript.targets"

Condition="Exists('$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStudioVersion)\TypeScri
pt\Microsoft.TypeScript.targets')" />
```

Manage npm packages in Visual Studio

6/24/2022 • 7 minutes to read • Edit Online

Applies to: Visual Studio ✗ Visual Studio for Mac ✗ Visual Studio Code

npm allows you to install and manage packages for use in both Node.js and ASP.NET Core applications. Visual Studio makes it easy to interact with npm and issue npm commands through the UI or directly. If you're unfamiliar with npm and want to learn more, go to the npm documentation.

Visual Studio integration with npm is different depending on your project type.

- CLI-based projects (.esproj)
- Node.js
- ASP.NET Core
- Open folder (Node.js)

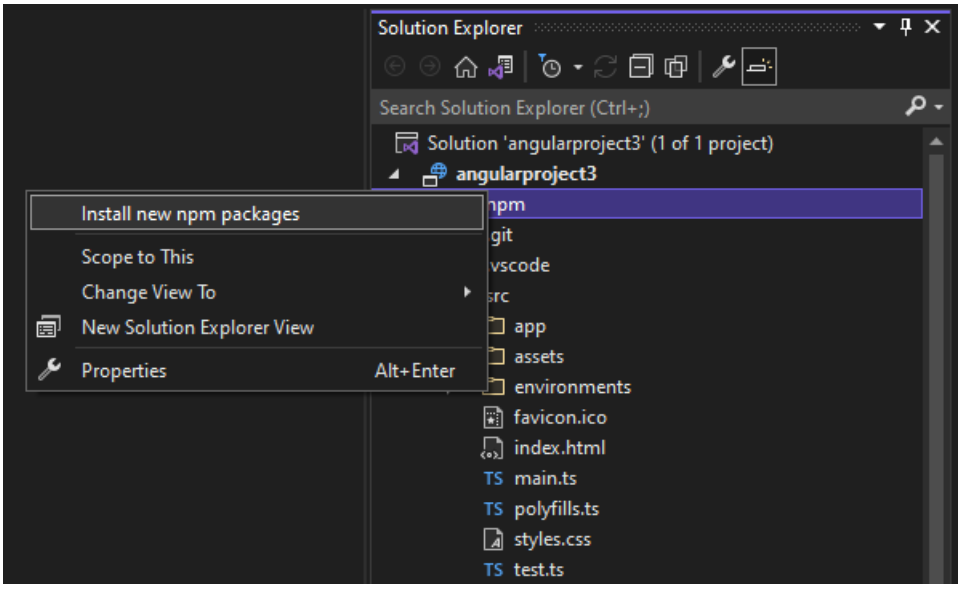
- Node.js
- ASP.NET Core
- Open folder (Node.js)

> **IMPORTANT**
> npm expects the *node_modules* folder and *package.json* in the project root. If your app's folder structure is different, you should modify your folder structure if you want to manage npm packages using Visual Studio.

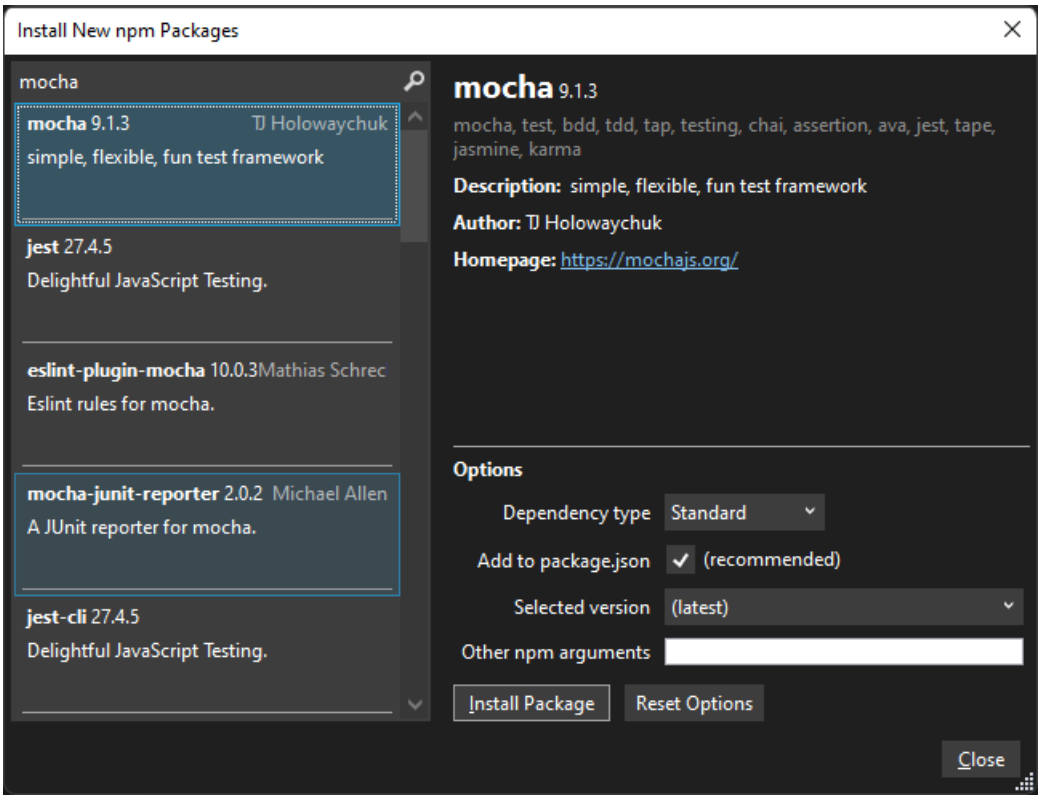
CLI-based project (.esproj)

Starting in Visual Studio 2022 Preview 4, the npm package manager is available for CLI-based projects, so you can now download npm modules similarly to the way you download NuGet packages for ASP.NET Core projects. Then you can use *package.json* to modify and delete packages.

To open the package manager, from Solution Explorer, right-click the **npm** node in your project.



Next, you can search for npm packages, select one, and install by selecting **Install Package**.



Node.js projects

For Node.js projects (.njsproj), you can perform the following tasks:

- Install packages from Solution Explorer
- Manage installed packages from Solution Explorer
- Use the `.npm` command in the Node.js Interactive Window

These features work together and synchronize with the project system and the *package.json* file in the project.

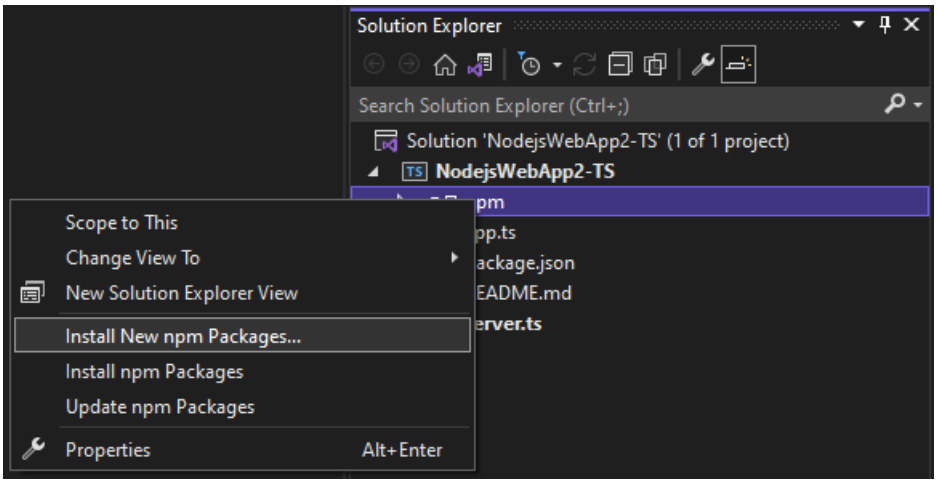
Prerequisites

You need the **Node.js development** workload and the Node.js runtime installed to add npm support to your project. For detailed steps, see Create a Node.js and Express app.

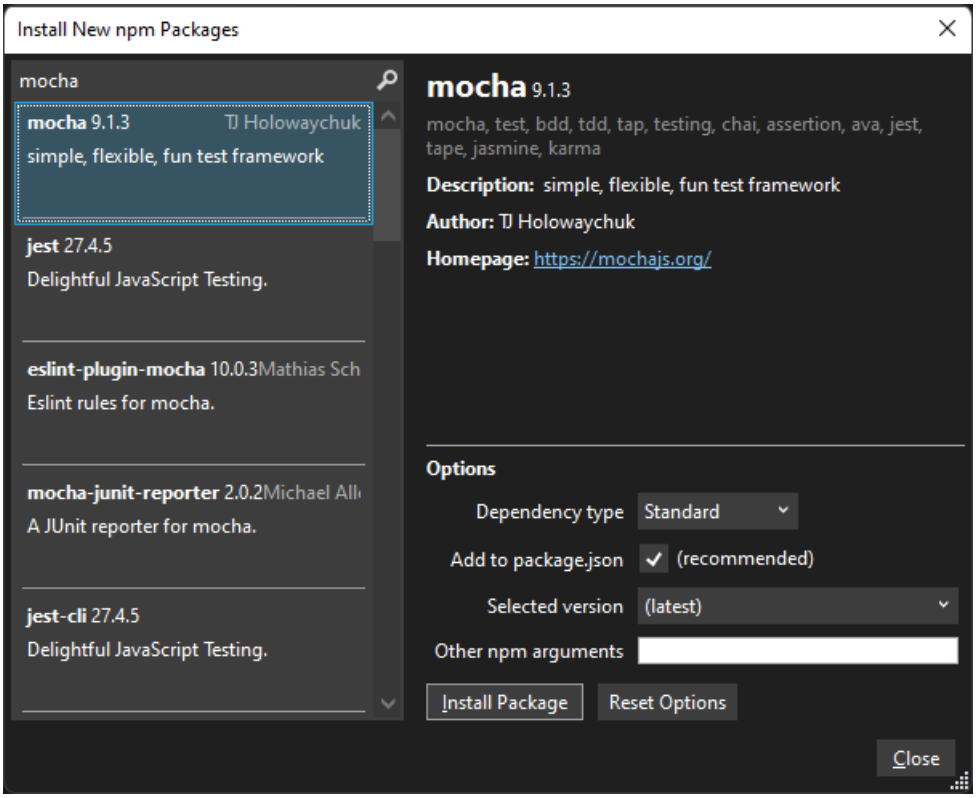
> **NOTE**
> For existing Node.js projects, use the **From existing Node.js code** solution template or the Open folder (Node.js) project type to enable npm in your project.

Install packages from Solution Explorer (Node.js)

For Node.js projects, the easiest way to install npm packages is through the npm package installation window. To access this window, right-click the **npm** node in the project and select **Install New npm Packages**.

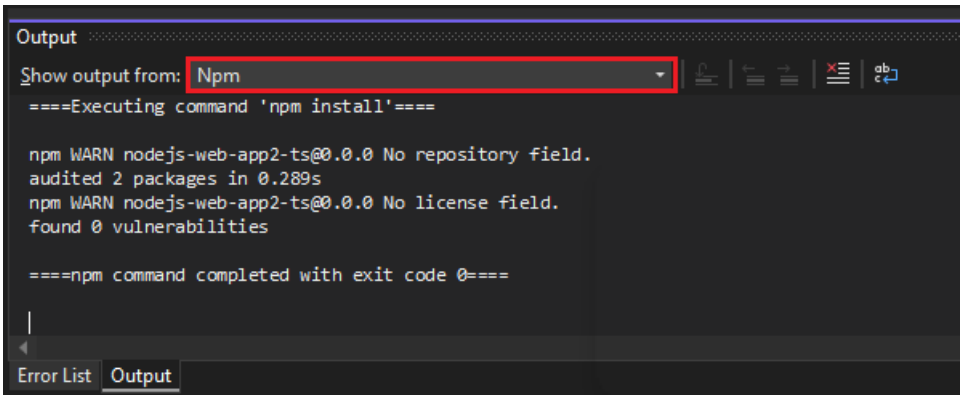


In this window you can search for a package, specify options, and install.



- **Dependency type** - Chose between **Standard**, **Development**, and **Optional** packages. Standard specifies that the package is a runtime dependency, whereas Development specifies that the package is only required during development.
- **Add to package.json** - Recommended. This configurable option is deprecated.
- **Selected version** - Select the version of the package you want to install.
- **Other npm arguments** - Specify other standard npm arguments. For example, you can enter a version value such as `@~0.8` to install a specific version that is not available in the versions list.

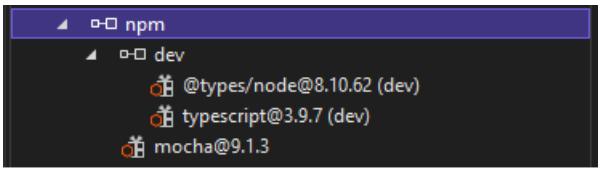
You can see the progress of the installation in the **npm** output in the **Output** window (to open the window, choose **View** > **Output** or press **Ctrl** + **Alt** + **O**). This may take some time.



> **TIP**
>
> You can search for scoped packages by prepending the search query with the scope you're interested in, for example, type `@types/mocha` to look for TypeScript definition files for mocha. Also, when installing type definitions for TypeScript, you can specify the TypeScript version you're targeting by adding `@ts2.6` in the npm argument field.

Manage installed packages in Solution Explorer (Node.js)

npm packages are shown in Solution Explorer. The entries under the **npm** node mimic the dependencies in the *package.json* file.



Package status

- - Installed and listed in package.json
- - Installed, but not explicitly listed in package.json
- - Not installed, but listed in package.json

Right-click the **npm** node to take one of the following actions:

- **Install New npm Packages** Opens the UI to install new packages.
- **Install npm Packages** Runs the npm install command to install all packages listed in *package.json*. (Runs `npm install`.)
- **Update npm Packages** Updates packages to the latest versions, according to the semantic versioning (SemVer) range specified in *package.json*. (Runs `npm update --save`.). SemVer ranges are typically specified using "~" or "^". For more information, package.json configuration.

Right-click a package node to take one of the following actions:

- **Install npm Package(s)** Runs the npm install command to install the package version listed in *package.json*. (Runs `npm install`.)
- **Update npm Package(s)** Updates the package to the latest version, according to the SemVer range specified in *package.json*. (Run `npm update --save`.) SemVer ranges are typically specified using "~" or "^".
- **Uninstall npm Package(s)** Uninstalls the package and removes it from *package.json* (Runs `npm uninstall --save`.)

> **NOTE**
>
> For help resolving issues with npm packages, see Troubleshooting.

Use the .npm command in the Node.js Interactive Window (Node.js)

You can also use the `.npm` command in the Node.js Interactive Window to execute npm commands. To open the window, right-click the project in Solution Explorer and choose **Open Node.js Interactive Window** (or press **Ctrl** + **K**, **N**).

In the window, you can use commands such as the following to install a package:

```
.npm install azure@4.2.3
```

> **TIP**
> By default, npm will execute in your project's home directory. If you have multiple projects in your solution specify the name or the path of the project in brackets. `.npm [MyProjectNameOrPath] install azure@4.2.3`

> **TIP**
> If your project doesn't contain a package.json file, use `.npm init -y` to create a new package.json file with default entries.

ASP.NET Core projects

For projects such as ASP.NET Core projects, you can integrate npm support in your project and use npm to install packages.

- Add npm support to a project
- Install packages using package.json

> **NOTE**
> For ASP.NET Core projects, you can also use Library Manager or yarn instead of npm to install client-side JavaScript and CSS files.

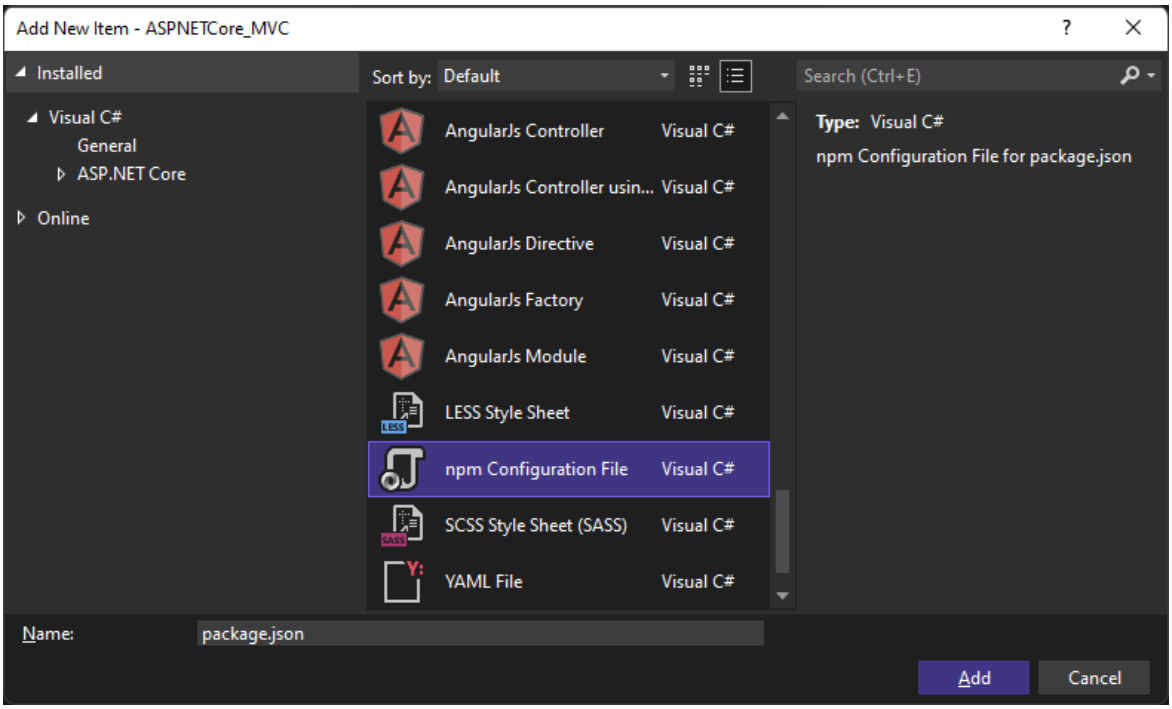
Add npm support to a project (ASP.NET Core)

If your project does not already include a *package.json* file, you can add one to enable npm support by adding a *package.json* file to the project.

1. If you don't have Node.js installed, we recommend you install the LTS version from the Node.js website for best compatibility with outside frameworks and libraries.

   npm requires Node.js.

2. To add the *package.json* file, right-click the project in Solution Explorer and choose **Add** > **New Item** (or press **Ctrl** + **SHIFT** + **A**). Use the search box to find the npm file, choose the **npm Configuration File**, use the default name, and click **Add**.



If you don't see the npm Configuration File listed, Node.js development tools are not installed. You can use the Visual Studio Installer to add the **Node.js development** workload. Then repeat the previous step.

3. Include one or more npm packages in the `dependencies` or `devDependencies` section of *package.json*. For example, you might add the following to the file:

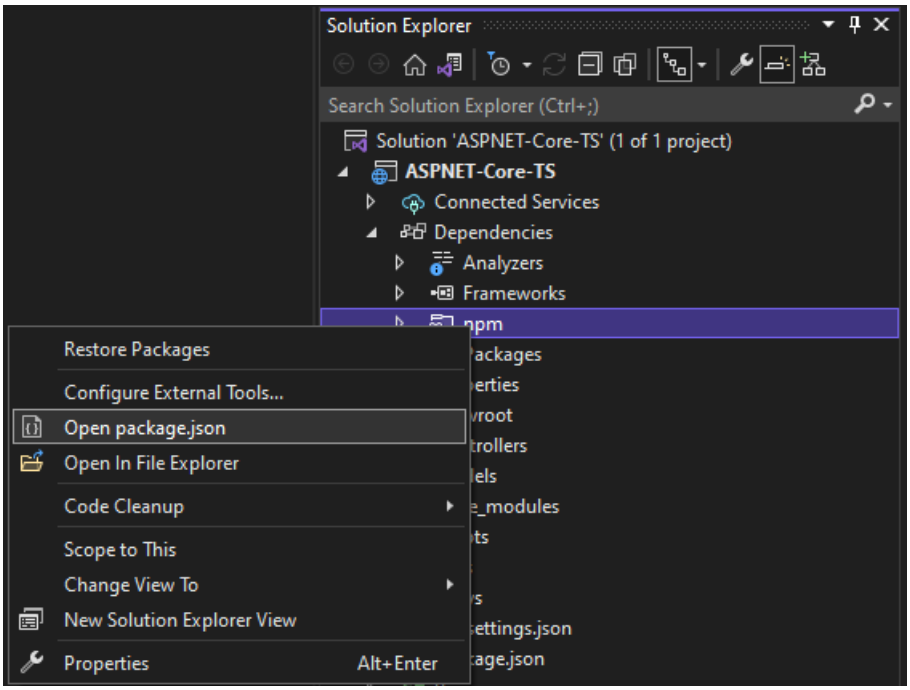
```
"devDependencies": {
   "gulp": "4.0.2",
   "@types/jquery": "3.3.33"
}
```

When you save the file, Visual Studio adds the package under the **Dependencies / npm** node in Solution Explorer. If you don't see the node, right-click **package.json** and choose **Restore Packages**.

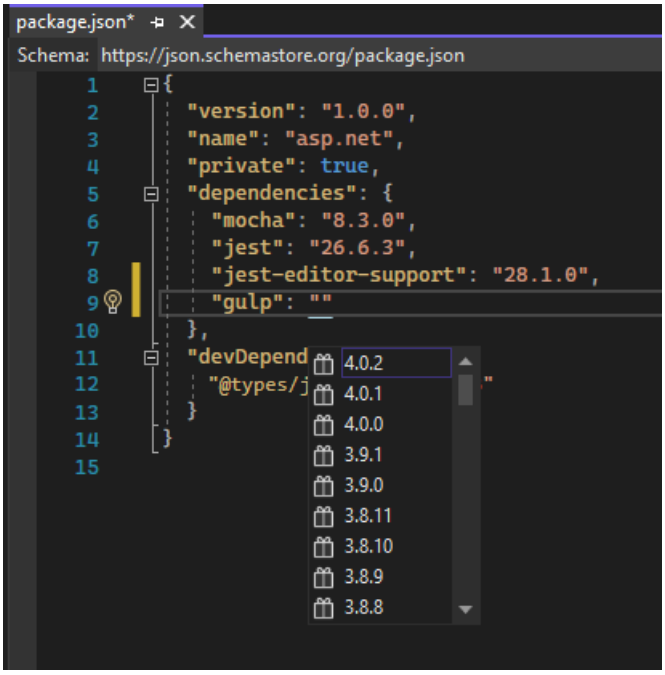
> **NOTE**
> In some scenarios, Solution Explorer may not show the correct status for installed npm packages. For more information, see Troubleshooting.

Install packages using package.json (ASP.NET Core)

For projects with npm included, you can configure npm packages using `package.json`. Either open `package.json` directly, or right-click the npm node in Solution Explorer and choose **Open package.json**.

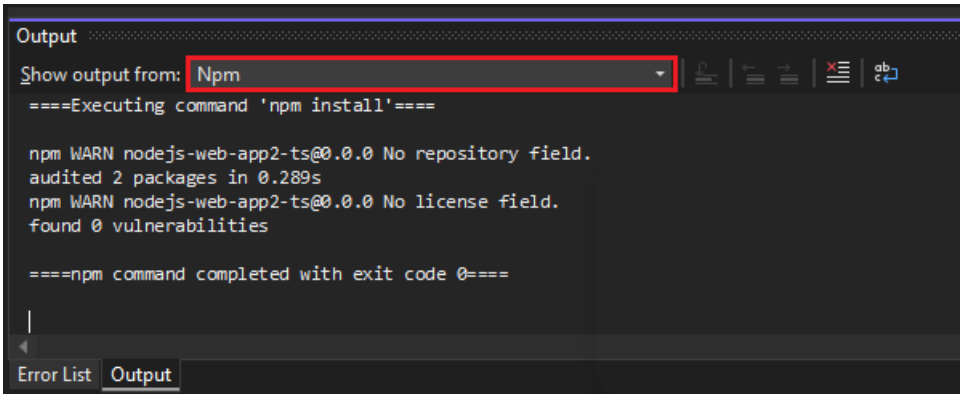


IntelliSense in *package.json* helps you select a particular version of an npm package.



When you save the file, Visual Studio adds the package under the **Dependencies / npm** node in Solution Explorer. If you don't see the node, right-click **package.json** and choose **Restore Packages**.

It may take several minutes to install a package. Check progress on package installation by switching to **npm** output in the **Output** window.



Troubleshooting npm packages

- npm requires Node.js. If you don't have Node.js installed, we recommend you install the LTS version from the Node.js website for best compatibility with outside frameworks and libraries.
- For Node.js projects, you must have the **Node.js development** workload installed for npm support.
- In some scenarios, Solution Explorer may not show the correct status for installed npm packages due to a known issue described here. For example, the package may appear as not installed when it is installed. In most cases, you can update Solution Explorer by deleting *package.json*, restarting Visual Studio, and re-adding the *package.json* file as described earlier in this article. Or, when installing packages, you can use the npm Output window to verify installation status.
- In some ASP.NET Core scenarios, the npm node in Solution Explorer may not be visible after you build the project. To make the node visible again, right-click the project node and choose **Unload Project.** Then right-click the project node and choose **Reload Project**.
- If you see any errors when building your app or transpiling TypeScript code, check for npm package incompatibilities as a potential source of errors. To help identify errors, check the npm Output window when installing the packages, as described previously in this article. For example, if one or more npm package versions has been deprecated and results in an error, you may need to install a more recent version to fix errors. For information on using *package.json* to control npm package versions, see package.json configuration.

Develop JavaScript and TypeScript code in Visual Studio without solutions or projects

6/24/2022 • 2 minutes to read • Edit Online

Applies to: Visual Studio Visual Studio for Mac Visual Studio Code

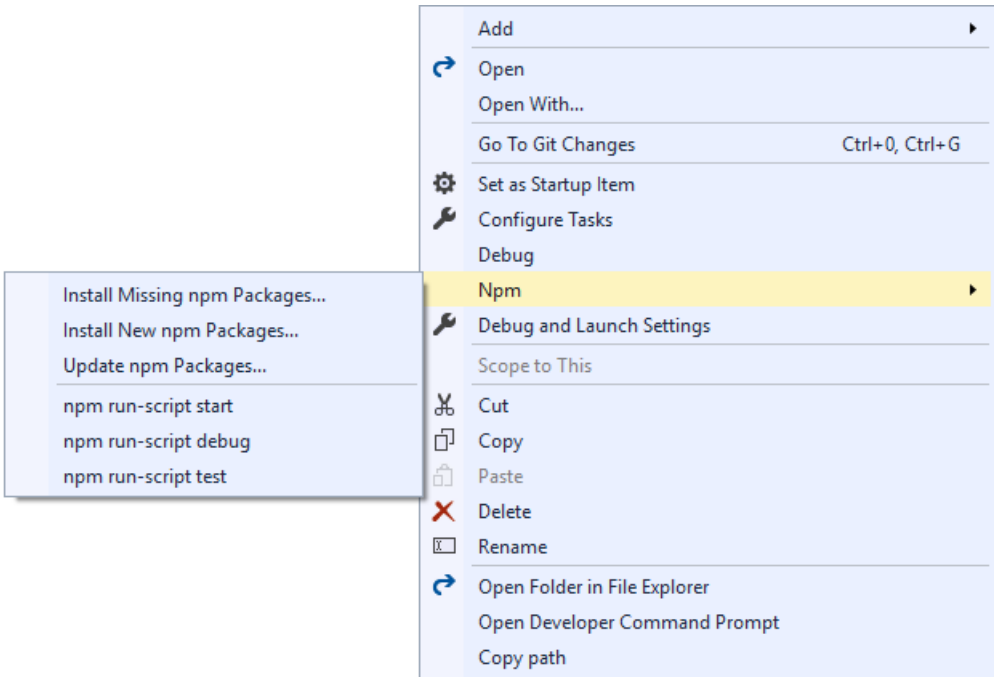
Starting in Visual Studio 2017, you can develop code without projects or solutions, which enables you to open a folder of code and immediately start working with rich editor support such as IntelliSense, search, refactoring, debugging, and more. In addition to these features, the Node.js Tools for Visual Studio adds support for building TypeScript files, managing npm packages, and running npm scripts.

To get started, select **File** > **Open** > **Folder** from the toolbar. Solution Explorer displays all the files in the folder, and you can open any of the files to begin editing. In the background, Visual Studio indexes the files to enable npm, build, and debug features.

> **IMPORTANT**
> Many of the features described in this article, including npm integration, require Visual Studio 2017 version 15.8 or later versions. The Visual Studio **Node.js development** workload must be installed.

npm integration

If the folder you open contains a *package.json* file, you can right-click *package.json* to show a context menu (shortcut menu) specific to npm.



In the shortcut menu, you can manage the packages installed by npm in the same way that you manage npm packages when using a project file.

In addition, the menu also allows you to run scripts defined in the `scripts` element in *package.json*. These scripts will use the version of Node.js available on the `PATH` environment variable. The scripts run in a new window. This is a great way to execute build or run scripts.

Build and debug

package.json

If the *package.json* in the folder specifies a `main` element, the **Debug** command will be available in the right-click shortcut menu for *package.json*. Clicking this will start *node.exe* with the specified script as its argument.

JavaScript files

You can debug JavaScript files by right-clicking a file and selecting **Debug** from the shortcut menu. This starts *node.exe* with that JavaScript file as its argument.

TypeScript files and tsconfig.json

If there is no *tsconfig.json* present in the folder, you can right-click a TypeScript file to see shortcut menu commands to build and debug that file. When you use these commands, you build or debug using *tsc.exe* with default options. (You need to build the file before you can debug.)

> **NOTE**
>
> When building TypeScript code, we use the newest version installed in `C:\Program Files (x86)\Microsoft SDKs\TypeScript`.

If there is a *tsconfig.json* file present in the folder, you can right-click a TypeScript file to see a menu command to debug that TypeScript file. The option appears only if there is no `outFile` specified in *tsconfig.json*. If an `outFile` is specified, you can debug that file by right-clicking *tsconfig.json* and selecting the correct option. The `tsconfig.json` file also gives you a build option to allow you to specify compiler options.

> **NOTE**
>
> You can find more information about *tsconfig.json* in the tsconfig.json TypeScript Handbook page.

Unit Tests

You can enable the unit test integration in Visual Studio by specifying a test root in your *package.json*:

```
{
    // ...
    "vsTest":{
        "testRoot": "./tests"
    }
    // ...
}
```

The test runner enumerates the locally installed packages to determine which test framework to use. If none of the supported frameworks are recognized, the test runner defaults to *ExportRunner*. The other supported frameworks are:

- Mocha (mochajs.org)
- Jasmine (Jasmine.github.io)
- Tape (github.com/substack/tape)
- Jest (jestjs.io)

After opening Test Explorer (choose **Test** > **Windows** > **Test Explorer**), Visual Studio discovers and displays tests.

NOTE

The test runner will only enumerate the JavaScript files in the test root, if your application is written in TypeScript you need to build those first.

Work with the Node.js interactive window

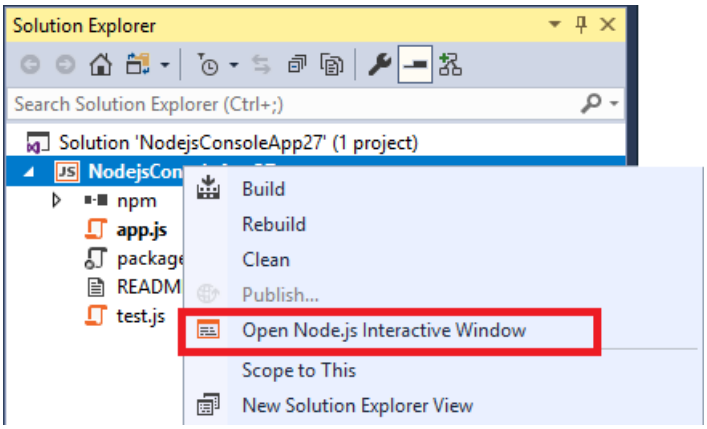
6/24/2022 • 2 minutes to read • Edit Online

Applies to: Visual Studio | Visual Studio for Mac | Visual Studio Code

Node.js Tools for Visual Studio include an interactive window for the installed Node.js runtime. This window allows you to enter JavaScript code and see the results immediately, as well as execute npm commands to interact with the current project. The interactive window is also known as a REPL (**R**ead/**E**valuate/**P**rint **L**oop).

Open the interactive window

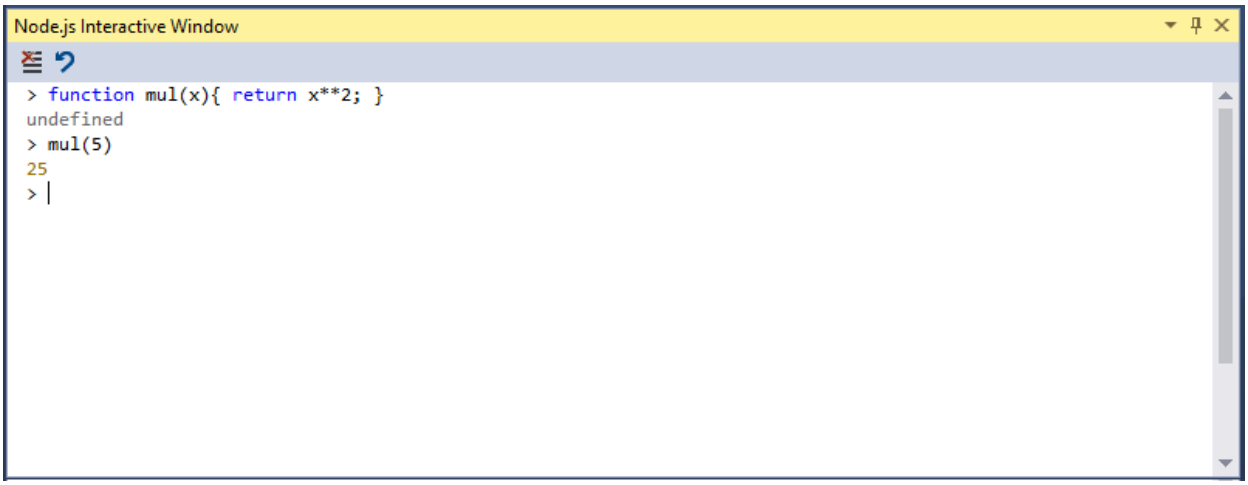
You can open the interactive window by right-clicking the Node.js project node in Solution Explorer and selecting **Open Node.js Interactive Window**.



The default short-cut keys to open the Node.js interactive window are **[CTRL] + K, N**. Or, you can open the window from the toolbar by choosing **View** > **Windows** > **Node.js Interactive Window**.

Use the REPL

Once opened, you can enter commands.



The interactive window has several built-in commands, which start with a dot prefix to distinguish them from any JavaScript function that you declare. The following commands are supported:

.cls, .clear Clears the contents of the editor window, leaving the history and execution context intact.

.help Displays help on the specified command, or on all available commands and key bindings if none is

specified.

.info Shows information about the current used Node.js executable.

.npm Runs an npm command. If the solution contains more than one project, specify the target project using `.npm [projectname] <npm arguments>`.

.reset Resets the execution environment to the initial state, keep history.

.save Saves the current REPL session to a file.

Debug a JavaScript or TypeScript app in Visual Studio

6/24/2022 • 12 minutes to read • Edit Online

Applies to: Visual Studio Visual Studio for Mac Visual Studio Code

You can debug JavaScript and TypeScript code using Visual Studio. You can set and hit breakpoints, attach the debugger, inspect variables, view the call stack, and use other debugging features.

> **TIP**
> If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free. Depending on the type of app development you're doing, you may need to install the **Node.js development workload** with Visual Studio.

Debug server-side script

1. With your project open in Visual Studio, open a server-side JavaScript file (such as *server.js*), click in the gutter to the left gutter to set a breakpoint:

   Breakpoints are the most basic and essential feature 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. To run your app, press **F5** (**Debug** > **Start Debugging**).

   The debugger pauses at the breakpoint you set (the current statement is marked in yellow). Now, you can inspect your app state by hovering over variables that are currently in scope, using debugger windows like the **Locals** and **Watch** windows.

3. Press **F5** to continue the app.

4. If you want to use the Chrome Developer Tools or F12 Tools, press **F12**. You can use these tools to examine the DOM and interact with the app using the JavaScript Console.

Debug client-side script

Visual Studio provides client-side debugging support for Chrome and Microsoft Edge (Chromium) only. In some scenarios, the debugger automatically hits breakpoints in JavaScript and TypeScript code and in embedded scripts on HTML files. For debugging client-side script in ASP.NET apps, see the blog post Debug JavaScript in Microsoft Edge and this post for Google Chrome. For debugging TypeScript in ASP.NET Core, also see Create an

ASP.NET Core app with TypeScript.

> **NOTE**
> For ASP.NET and ASP.NET Core, debugging embedded scripts in CSHTML files is not supported. JavaScript code must be in separate files to enable debugging.

For applications other than ASP.NET, follow the steps described here.

Prepare your app for debugging

If your source is minified or created by a transpiler like TypeScript or Babel, the use of source maps is required for the best debugging experience. Without source maps, you can still attach the debugger to a running client-side script. However, you may only be able to set and hit breakpoints in the minified or transpiled file, not in the original source file. For example, in a Vue.js app, minified script gets passed as a string to an `eval` statement, and there is no way to step through this code effectively using the Visual Studio debugger, unless you use source maps. In complex debugging scenarios, you might also use Chrome Developer Tools or F12 Tools for Microsoft Edge instead.

For help to generate source maps, see Generate source maps for debugging.

Prepare the browser for debugging

For this scenario, use either Microsoft Edge (Chromium), currently named **Microsoft Edge Beta** in the IDE, or Chrome.

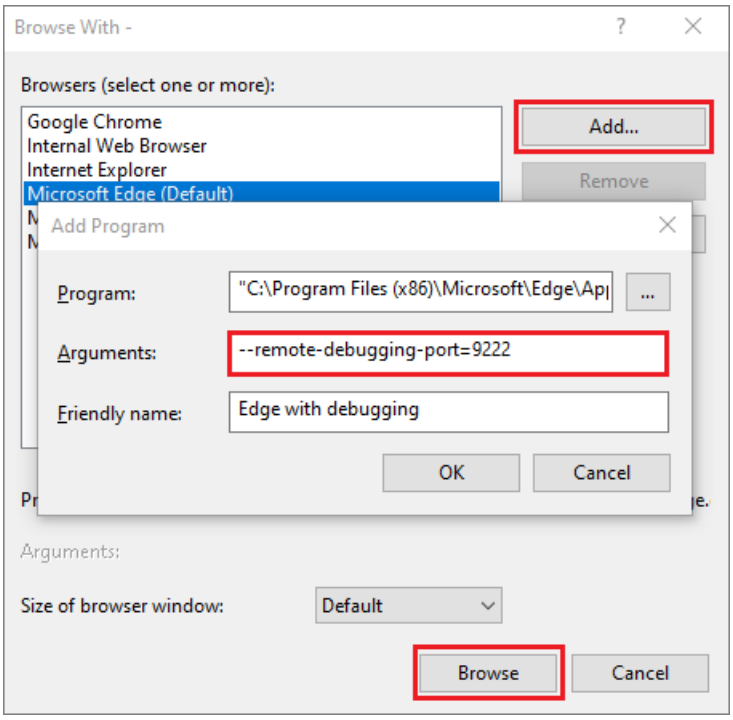
1. Close all windows for the target browser.

   Other browser instances can prevent the browser from opening with debugging enabled. (Browser extensions may be running and preventing full debug mode, so you may need to open Task Manager to find unexpected instances of Chrome.)

   For Microsoft Edge (Chromium), also shut down all instances of Chrome. Because both browsers use the chromium code base, this gives the best results.

2. Start your browser with debugging enabled.

   Starting in Visual Studio 2019, you can set the `--remote-debugging-port=9222` flag at browser launch by selecting **Browse With...** > from the **Debug** toolbar, then choosing **Add**, and then setting the flag in the **Arguments** field. Use a different friendly name for the browser such as **Edge with Debugging** or **Chrome with Debugging**. For details, see the Release Notes.



Alternatively, open the **Run** command from the Windows **Start** button (right-click and choose **Run**), and enter the following command:

```
msedge --remote-debugging-port=9222
```

or,

```
chrome.exe --remote-debugging-port=9222
```

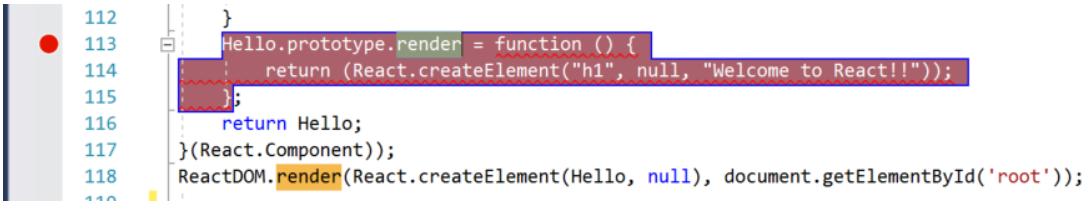
This starts your browser with debugging enabled.

The app is not yet running, so you get an empty browser page.

Attach the debugger to client-side script

To attach the debugger from Visual Studio and hit breakpoints in client-side code, the debugger needs help to identify the correct process. Here is one way to enable this.

1. Switch to Visual Studio and then set a breakpoint in your source code, which might be a JavaScript file, TypeScript file, or a JSX file. (Set the breakpoint in a line of code that allows breakpoints, such as a return statement or a var declaration.)



   To find the specific code in a transpiled file, use **Ctrl+F** (**Edit** > **Find and Replace** > **Quick Find**).

   For client-side code, to hit a breakpoint in a TypeScript file, *.vue*, or JSX file typically requires the use of source maps. A source map must be configured correctly to support debugging in Visual Studio.

2. Select your target browser as the debug target in Visual Studio, then press **Ctrl+F5** (**Debug** > **Start Without Debugging**) to run the app in the browser.

   If you created a browser configuration with a friendly name, choose that as your debug target.

   The app opens in a new browser tab.

3. Choose **Debug** > **Attach to Process**.

> **TIP**
>
> Starting in Visual Studio 2017, once you attach to the process the first time by following these steps, you can quickly reattach to the same process by choosing **Debug** > **Reattach to Process**.

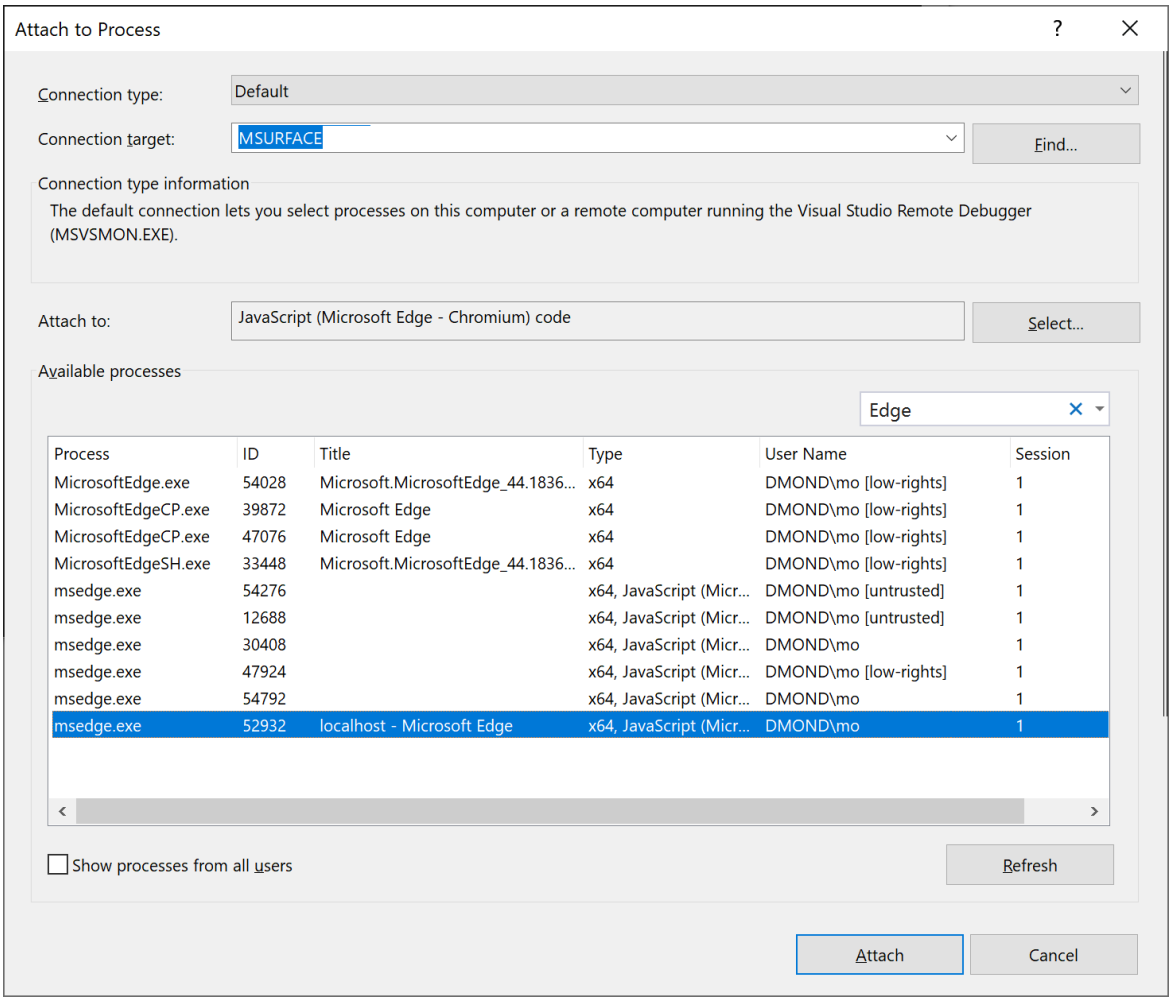
4. In the **Attach to Process** dialog box, get a filtered list of browser instances that you can attach to.

   In Visual Studio 2019, choose the correct debugger for your target browser, **JavaScript (Chrome)** or **JavaScript (Microsoft Edge - Chromium)** in the **Attach to** field, type **chrome** or **edge** in the filter box to filter the search results.

5. Select the browser process with the correct host port (localhost in this example), and select **Attach**.

   The port (for example, 1337) may also appear in the **Title** field to help you select the correct browser instance.

   The following example shows how this looks for the Microsoft Edge (Chromium) browser.



> **TIP**
>
> If the debugger does not attach and you see the message "Failed to launch debug adapter" or "Unable to attach to the process. An operation is not legal in the current state.", use the Windows Task Manager to close all instances of the target browser before starting the browser in debugging mode. Browser extensions may be running and preventing full debug mode.

6. Because the code with the breakpoint may have already executed, refresh your browser page. If

necessary, take action to cause the code with the breakpoint to execute.

While paused in the debugger, you can examine your app state by hovering over variables and using debugger windows. You can advance the debugger by stepping through code (**F5**, **F10**, and **F11**). For more information on basic debugging features, see First look at the debugger.

You may hit the breakpoint in either a transpiled *.js* file or source file, depending on your app type, which steps you followed previously, and other factors such as your browser state. Either way, you can step through code and examine variables.

- If you need to break into code in a TypeScript, JSX, or *.vue* source file and are unable to do it, make sure that your environment is set up correctly, as described in the Troubleshooting section.
- If you need to break into code in a transpiled JavaScript file (for example, *app-bundle.js*) and are unable to do it, remove the source map file, *filename.js.map*.

Troubleshooting breakpoints and source maps

If you need to break into code in a TypeScript or JSX source file and are unable to do it, use **Attach to Process** as described in the previous steps to attach the debugger. Make sure you that your environment is set up correctly:

- You closed all browser instances, including Chrome extensions (using the Task Manager), so that you can run the browser in debug mode.
- Make sure you start the browser in debug mode.
- Make sure that your source map file includes the correct relative path to your source file and that it doesn't include unsupported prefixes such as *webpack:///*, which prevents the Visual Studio debugger from locating a source file. For example, a reference like *webpack:///./app.tsx* might be corrected to *./app.tsx*. You can do this manually in the source map file (which is helpful for testing) or through a custom build configuration. For more information, see Generate source maps for debugging.

Alternatively, if you need to break into code in a source file (for example, *app.tsx*) and are unable to do it, try using the `debugger;` statement in the source file, or set breakpoints in the Chrome Developer Tools (or F12 Tools for Microsoft Edge) instead.

Generate source maps for debugging

Visual Studio has the capability to use and generate source maps on JavaScript source files. This is often required if your source is minified or created by a transpiler like TypeScript or Babel. The options available depend on the project type.

- A TypeScript project in Visual Studio generates source maps for you by default. For more information, see Configure source maps using a tsconfig.json file.
- In a JavaScript project, you can generate source maps using a bundler like webpack and a compiler like the TypeScript compiler (or Babel), which you can add to your project. For the TypeScript compiler, you must also add a *tsconfig.json* file and set the `sourceMap` compiler option. For an example that shows how to do this using a basic webpack configuration, see Create a Node.js app with React.

> **NOTE**
> If you are new to source maps, please read Introduction to JavaScript Source Maps before continuing.

To configure advanced settings for source maps, use either a *tsconfig.json* or the project settings in a TypeScript project, but not both.

To enable debugging using Visual Studio, you need to make sure that the reference(s) to your source file in the generated source map are correct (this may require testing). For example, if you are using webpack, references in the source map file include the *webpack:///* prefix, which prevents Visual Studio from finding a TypeScript or JSX source file. Specifically, when you correct this for debugging purposes, the reference to the source file (such as *app.tsx*), must be changed from something like *webpack:///./app.tsx* to something like *./app.tsx*, which enables debugging (the path is relative to your source file). The following example shows how you can configure source maps in webpack, which is one of the most common bundlers, so that they work with Visual Studio.

(Webpack only) If you are setting the breakpoint in a TypeScript of JSX file (rather than a transpiled JavaScript file), you need to update your webpack configuration. For example, in *webpack-config.js*, you might need to replace the following code:

```
  output: {
    filename: "./app-bundle.js", // This is an example of the filename in your project
  },
```

with this code:

```
  output: {
    filename: "./app-bundle.js", // Replace with the filename in your project
    devtoolModuleFilenameTemplate: '[resource-path]'  // Removes the webpack:/// prefix
  },
```

This is a development-only setting to enable debugging of client-side code in Visual Studio.

For complicated scenarios, the browser tools (**F12**) sometimes work best for debugging, because they don't require changes to custom prefixes.

Configure source maps using a tsconfig.json file

If you add a *tsconfig.json* file to your project, Visual Studio treats the directory root as a TypeScript project. To add the file, right-click your project in Solution Explorer, and then choose **Add > New Item > TypeScript JSON Configuration File**. A *tsconfig.json* file like the following gets added to your project.

```
{
  "compilerOptions": {
    "noImplicitAny": false,
    "noEmitOnError": true,
    "removeComments": false,
    "sourceMap": true,
    "target": "es5"
  },
  "exclude": [
    "node_modules",
    "wwwroot"
  ]
}
```

Compiler options for tsconfig.json

- **inlineSourceMap**: Emit a single file with source maps instead of creating a separate source map for each source file.
- **inlineSources**: Emit the source alongside the source maps within a single file; requires *inlineSourceMap* or *sourceMap* to be set.
- **mapRoot**: Specifies the location where the debugger should find source map (*.map*) files instead of the default location. Use this flag if the run-time *.map* files need to be in a different location than the *.js* files. The location specified is embedded in the source map to direct the debugger to the location of the *.map* files.
- **sourceMap**: Generates a corresponding *.map* file.

- **sourceRoot**: Specifies the location where the debugger should find TypeScript files instead of the source locations. Use this flag if the run-time sources need to be in a different location than the location at design-time. The location specified is embedded in the source map to direct the debugger to where the source files are located.

For more details about the compiler options, check the page Compiler Options on the TypeScript Handbook.

Configure source maps using project settings (TypeScript project)

You can also configure the source map settings using project properties by right-clicking the project and then choosing **Project > Properties > TypeScript Build > Debugging**.

These project settings are available.

- **Generate source maps** (equivalent to **sourceMap** in *tsconfig.json*): Generates corresponding *.map* file.
- **Specify root directory of source maps** (equivalent to **mapRoot** in *tsconfig.json*): Specifies the location where the debugger should find map files instead of the generated locations. Use this flag if the run-time *.map* files need to be located in a different location than the .js files. The location specified is embedded in the source map to direct the debugger to where the map files are located.
- **Specify root directory of TypeScript files** (equivalent to **sourceRoot** in *tsconfig.json*): Specifies the location where the debugger should find TypeScript files instead of source locations. Use this flag if the run-time source files need to be in a different location than the location at design-time. The location specified is embedded in the source map to direct the debugger to where the source files are located.

Debug JavaScript in dynamic files using Razor (ASP.NET)

Starting in Visual Studio 2019, Visual Studio provides debugging support for Chrome and Microsoft Edge (Chromium) only.

However, you cannot automatically hit breakpoints on files generated with Razor syntax (cshtml, vbhtml). There are two approaches you can use to debug this kind of file:

- **Place the `debugger;` statement where you want to break**: This causes the dynamic script to stop execution and start debugging immediately while it is being created.
- **Load the page and open the dynamic document on Visual Studio**: You'll need to open the dynamic file while debugging, set your breakpoint, and refresh the page for this method to work. Depending on whether you're using Chrome or Internet Explorer, you'll find the file using one of the following strategies:

  For Chrome, go to **Solution Explorer > Script Documents > YourPageName**.

  > **NOTE**
  > When using Chrome, you might get a message **no source is available between <script> tags**. This is OK, just continue debugging.

  For Microsoft Edge (Chromium), use the same procedure as Chrome.

For more information, see Client-side debugging of ASP.NET projects in Google Chrome.

Unit testing JavaScript and TypeScript in Visual Studio

6/24/2022 • 9 minutes to read • Edit Online

Applies to: Visual Studio Visual Studio for Mac Visual Studio Code

You can write and run unit tests in Visual Studio using some of the more popular JavaScript frameworks without the need to switch to a command prompt. Both Node.js and ASP.NET Core projects are supported.

The supported frameworks are:

- Mocha (mochajs.org)
- Jasmine (Jasmine.github.io)
- Tape (github.com/substack/tape)
- Jest (jestjs.io)
- Export Runner (this framework is specific to Node.js Tools for Visual Studio)

For ASP.NET Core and JavaScript or TypeScript, see Write unit tests for ASP.NET Core .

If your favorite framework is not supported, see Add support for a unit test framework for information on adding support.

Write unit tests for a CLI-based project (.esproj)

The CLI-based projects supported in Visual Studio 2022 work with Test Explorer. Jest is the built-in test framework for React and Vue projects, and Karma and Jasmine is used for Angular projects. By default, you will be able to run the default tests provided by each framework, as well as any additional tests you write. Just hit the **Run** button in Test Explorer. If you don't already have Test Explorer open, you can find it by selecting **Test** > **Test Explorer** in the menu bar.

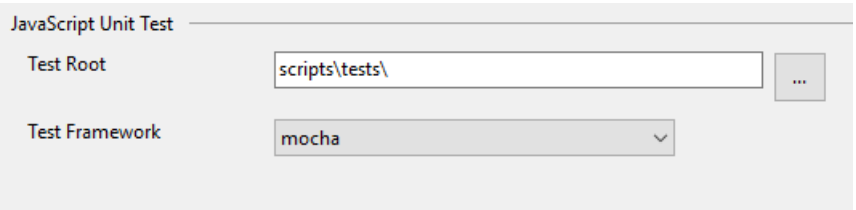
The Node.js development workload is required to support unit testing for CLI-based projects.

Mocha and Tape test libraries are also supported. To use one of these, simply change the default test library in package.json to the appropriate test library's package.

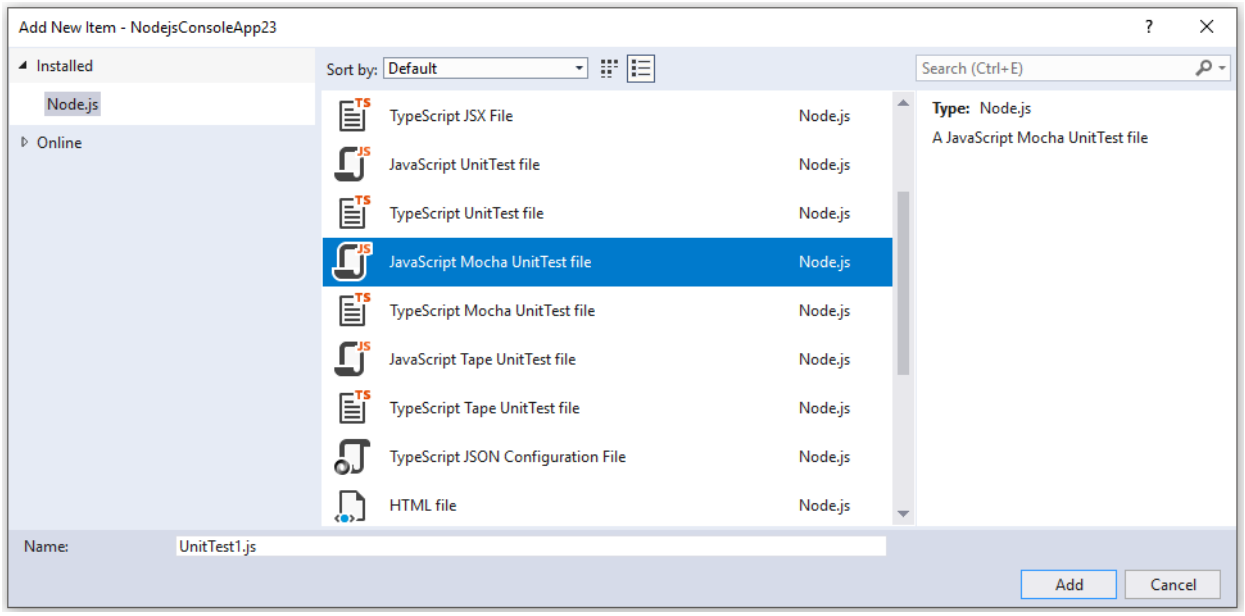
Write unit tests in a Node.js project (.njsproj)

For Node.js projects, before adding unit tests to your project, make sure the framework you plan to use is installed locally in your project. This is easy to do using the npm package installation window.

The preferred way to add unit tests to your project is by creating a *tests* folder in your project, and setting that as the test root in project properties. You also need to select the test framework you want to use.



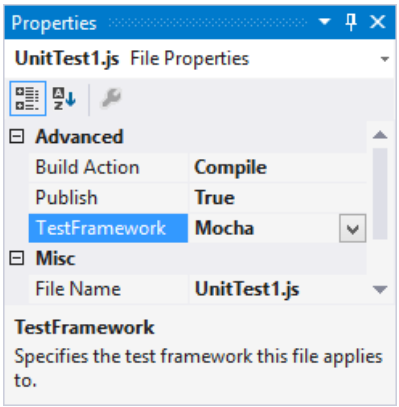
You can add simple blank tests to your project, using the **Add New Item** dialog box. Both JavaScript and TypeScript are supported in the same project.



For a Mocha unit test, use the following code:

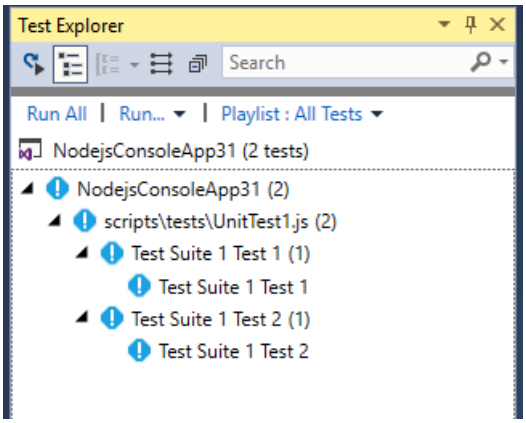
```
var assert = require('assert');

describe('Test Suite 1', function() {
    it('Test 1', function() {
        assert.ok(true, "This shouldn't fail");
    })

    it('Test 2', function() {
        assert.ok(1 === 1, "This shouldn't fail");
        assert.ok(false, "This should fail");
    })
})
```

If you haven't set the unit test options in the project properties, you must ensure the **Test Framework** property in the **Properties** window is set to the correct test framework for your unit test files. This is done automatically by the unit test file templates.



> **NOTE**
>
> The unit test options will take preference over the settings for individual files.

After opening Test Explorer (choose **Test** > **Windows** > **Test Explorer**), Visual Studio discovers and displays tests. If tests are not showing initially, then rebuild the project to refresh the list.



> **NOTE**
>
> For TypeScript, do not use the `outdir` or `outfile` option in *tsconfig.json*, because Test Explorer won't be able to find your unit tests.

Run tests (Node.js)

You can run tests in Visual Studio or from the command line.

Run tests in Visual Studio

You can run the tests by clicking the **Run All** link in Test Explorer. Or, you can run tests by selecting one or more tests or groups, right-clicking, and selecting **Run** from the shortcut menu. Tests run in the background, and Test Explorer automatically updates and shows the results. Furthermore, you can also debug selected tests by right-clicking and selecting **Debug**.

For TypeScript, unit tests are run against the generated JavaScript code.

> **NOTE**
>
> In most TypeScript scenarios, you can debug a unit test by setting a breakpoint in TypeScript code, right-clicking a test in Test Explorer, and choosing **Debug**. In more complex scenarios, such as some scenarios that use source maps, you may have difficulty hitting breakpoints in TypeScript code. As a workaround, try using the `debugger` keyword.

> **NOTE**
>
> We don't currently support profiling tests, or code coverage.

Run tests from the command line

You can run the tests from Developer Command Prompt for Visual Studio using the following command:

```
vstest.console.exe <path to project file>\NodejsConsoleApp23.njsproj /TestAdapterPath:
<VisualStudioFolder>\Common7\IDE\Extensions\Microsoft\NodeJsTools\TestAdapter
```

This command shows output similar to the following:

```
Microsoft (R) Test Execution Command Line Tool Version 15.5.0
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
Processing: NodejsConsoleApp23\NodejsConsoleApp23\UnitTest1.js
  Creating TestCase:NodejsConsoleApp23\NodejsConsoleApp23\UnitTest1.js::Test Suite 1 Test 1::mocha
  Creating TestCase:NodejsConsoleApp23\NodejsConsoleApp23\UnitTest1.js::Test Suite 1 Test 2::mocha
Processing finished for framework of Mocha
Passed   Test Suite 1 Test 1
Standard Output Messages:
 Using default Mocha settings
 1..2
 ok 1 Test Suite 1 Test 1

Failed   Test Suite 1 Test 2
Standard Output Messages:
 not ok 1 Test Suite 1 Test 2
   AssertionError [ERR_ASSERTION]: This should fail
       at Context.<anonymous> (NodejsConsoleApp23\NodejsConsoleApp23\UnitTest1.js:10:16)

Total tests: 2. Passed: 1. Failed: 1. Skipped: 0.
Test Run Failed.
Test execution time: 1.5731 Seconds
```

NOTE

If you get an error indicating that *vstest.console.exe* cannot be found, make sure you've opened the Developer Command Prompt and not a regular command prompt.

Write unit tests for ASP.NET Core

1. Create an ASP.NET Core project and add TypeScript support.

   For an example project, see Create an ASP.NET Core app with TypeScript. For unit testing support, we recommend you start with a standard ASP.NET Core project template.

   Use the NuGet package to add TypeScript support instead of the npm TypeScript package.

2. Install the NuGet package Microsoft.JavaScript.UnitTest

3. In Solution Explorer, right-click the project node and choose **Unload Project**.

   The *.csproj* file should open in Visual Studio.

4. Add the following elements to the *.csproj* file in the `PropertyGroup` element.

   This example specifies Mocha as the test framework. You could specify Jest, Tape, or Jasmine instead.

   ```
   <PropertyGroup>
      ...
      <JavaScriptTestRoot>tests\</JavaScriptTestRoot>
      <JavaScriptTestFramework>Mocha</JavaScriptTestFramework>
      <GenerateProgramFile>false</GenerateProgramFile>
   </PropertyGroup>
   ```

   The `JavaScriptTestRoot` element specifies that your unit tests will be in the *tests* folder of the project root.

5. In Solution Explorer, right-click the project node and choose **Reload Project**.

6. Add npm support as described in the npm package management article under ASP.NET Core projects.

   This requires installing the Node.js runtime for npm support and adding *package.json* in the project root.

7. In *package.json*, add the npm package you want under dependencies.

   For example, for mocha, you might use the following:

   ```
   "dependencies": {
     "mocha": "8.3.0",
   ```

   Some unit testing frameworks, such as Jest, require additional npm packages. For Jest, use the following JSON:

   ```
   "dependencies": {
     "jest": "26.6.3",
     "jest-editor-support": "28.1.0"
   ```

   > **NOTE**
   > In some scenarios, Solution Explorer may not show the npm node due to a known issue described here. If you need to see the npm node, you can unload the project (right-click the project and choose **Unload Project**) and then reload the project to make the npm node re-appear.

8. Add code to test.

   If you are using the example described in Create an ASP.NET Core app with TypeScript, add the following code at the end of the *library.ts* file, which is in the *scripts* folder.

   ```
   function getData(value) {
      if (value > 1) {
         return true;
      }
   }

   module.exports = getData;
   ```

   For TypeScript, unit tests are run against the generated JavaScript code.

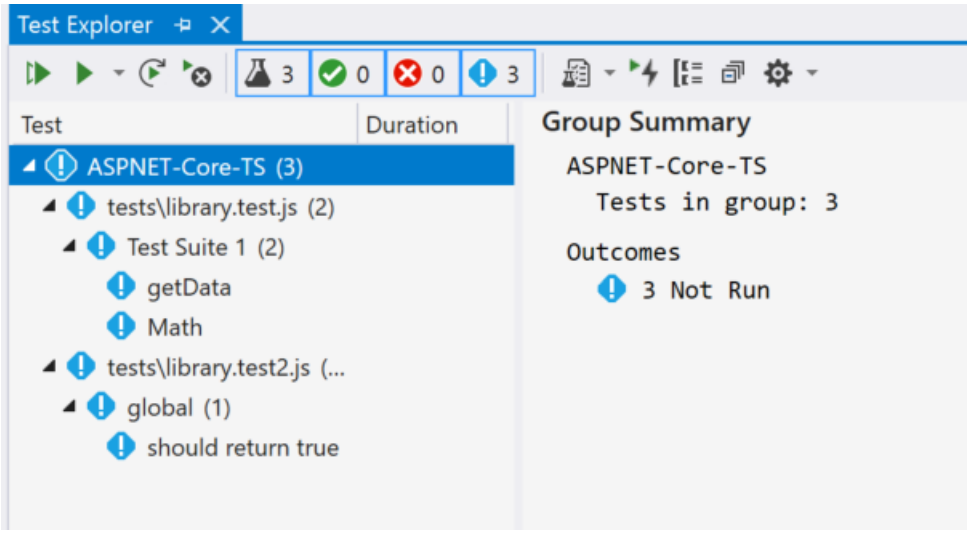
9. Add your unit tests to the *tests* folder in the project root.

   For example, you might use the following code by selecting the correct documentation tab that matches your test framework, in this example either Mocha or Jest. This code tests a function called `getData`.

   - Mocha
   - Jest

   ```
   const getData = require('../wwwroot/js/library.js');
   var assert = require('assert');

   describe('Test Suite 1', function () {
      it('getData', function () {
         assert.ok(true, getData(2));
      })
   })
   ```

10. Open Test Explorer (choose **Test** > **Windows** > **Test Explorer**) and Visual Studio discovers and displays tests. If tests are not showing initially, then rebuild the project to refresh the list.

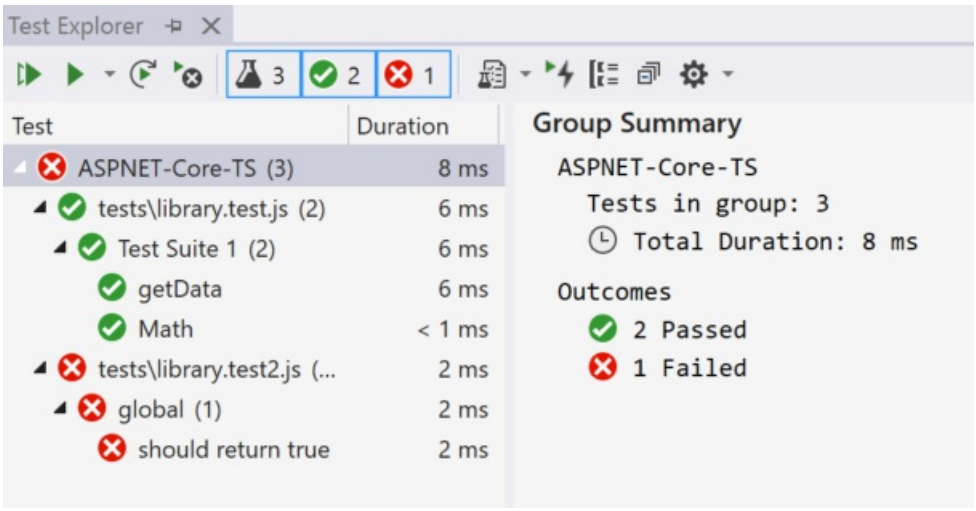


> **NOTE**
>
> For TypeScript, do not use the `outfile` option in *tsconfig.json*, because Test Explorer won't be able to find your unit tests. You can use the `outdir` option, but make sure that configuration files such as `package.json` and `tsconfig.json` are in the project root.

Run tests (ASP.NET Core)

You can run the tests by clicking the **Run All** link in Test Explorer. Or, you can run tests by selecting one or more tests or groups, right-clicking, and selecting **Run** from the shortcut menu. Tests run in the background, and Test Explorer automatically updates and shows the results. Furthermore, you can also debug selected tests by right-clicking and selecting **Debug**.

For TypeScript, unit tests are run against the generated JavaScript code.



> **NOTE**
>
> In most TypeScript scenarios, you can debug a unit test by setting a breakpoint in TypeScript code, right-clicking a test in Test Explorer, and choosing **Debug**. In more complex scenarios, such as some scenarios that use source maps, you may have difficulty hitting breakpoints in TypeScript code. As a workaround, try using the `debugger` keyword.

> **NOTE**
> We don't currently support profiling tests, or code coverage.

Add support for a unit test framework

You can add support for additional test frameworks by implementing the discovery and execution logic using JavaScript.

> **NOTE**
> For ASP.NET Core, add the NuGet package Microsoft.JavaScript.UnitTest to your project to add support.

You do this by adding a folder with the name of the test framework under:

```
<VisualStudioFolder>\Common7\IDE\Extensions\Microsoft\NodeJsTools\TestAdapter\TestFrameworks
```

If you don't see the `NodeJsTools` folder in an ASP.NET Core project, add the Node.js development workload using the Visual Studio Installer. This workload includes support for unit testing JavaScript and TypeScript.

This folder has to contain a JavaScript file with the same name which exports the following two functions:

- `find_tests`
- `run_tests`

For a good example of the `find_tests` and the `run_tests` implementations, see the implementation for the Mocha unit testing framework in:

```
<VisualStudioFolder>\Common7\IDE\Extensions\Microsoft\NodeJsTools\TestAdapter\TestFrameworks\mocha\mocha.js
```

Discovery of available test frameworks occurs at Visual Studio start. If a framework is added while Visual Studio is running, restart Visual Studio to detect the framework. However you don't need to restart when making changes to the implementation.

Unit tests in .NET Framework

You are not limited to writing unit tests in just your Node.js and ASP.NET Core projects. When you add the TestFramework and TestRoot properties to any C# or Visual Basic project, those tests will be enumerated and you can run them using the Test Explorer window.

To enable this, right-click the project node in the Solution Explorer, choose **Unload Project**, and then choose **Edit Project**. Then in the project file, add the following two elements to a property group.

> **IMPORTANT**
> Make sure that the property group you're adding the elements to doesn't have a condition specified. This can cause unexpected behavior.

```
<PropertyGroup>
    <JavaScriptTestRoot>tests\</JavaScriptTestRoot>
    <JavaScriptTestFramework>Tape</JavaScriptTestFramework>
</PropertyGroup>
```

Next, add your tests to the test root folder you specified, and they will be available to run in the Test Explorer

window. If they don't initially appear, you may need to rebuild the project.

Unit test .NET Core and .NET Standard

In addition to the preceding properties, you also need to install the NuGet package Microsoft.JavaScript.UnitTest and set the property:

```
<PropertyGroup>
    <GenerateProgramFile>false</GenerateProgramFile>
</PropertyGroup>
```

Some test frameworks may require additional npm packages for test detection. For example, jest requires the jest-editor-support npm package. If necessary, check the documentation for the specific framework.

package.json configuration

6/24/2022 • 2 minutes to read • Edit Online

Applies to: Visual Studio Visual Studio for Mac Visual Studio Code

If you are developing a Node.js app with a lot of npm packages, it's not uncommon to run into warnings or errors when you build your project if one or more packages has been updated. Sometimes, a version conflict results, or a package version has been deprecated. Here are a couple of quick tips to help you configure your package.json file and understand what is going on when you see warnings or errors. This is not a complete guide to *package.json* and is focused only on npm package versioning.

The npm package versioning system has strict rules. The version format follows here:

```
[major].[minor].[patch]
```

Let's say you have a package in your app with a version of 5.2.1. The major version is 5, the minor version is 2, and the patch is 1.

- In a major version update, the package includes new features that are backwards-incompatible, that is, breaking changes.
- In a minor version update, new features have been added to the package that are backwards-compatible with earlier package versions.
- In a patch update, one or more bug fixes are included. Bug fixes are always backwards-compatible.

It's worth noting that some npm package features have dependencies. For example, to use a new feature of the TypeScript compiler package (ts-loader) with webpack, it is possible you would also need to update the webpack npm package and the webpack-cli package.

To help manage package versioning, npm supports several notations that you can use in the *package.json*. You can use these notations to control the type of package updates that you want to accept in your app.

Let's say you are using React and need to include the **react** and **react-dom** npm package. You could specify that in several ways in your *package.json* file. For example, you can specify use of the exact version of a package as follows.

```
"dependencies": {
  "react": "16.4.2",
  "react-dom": "16.4.2",
},
```

Using the preceding notation, npm will always get the exact version specified, 16.4.2.

You can use a special notation to limit updates to patch updates (bug fixes). In this example:

```
"dependencies": {
  "react": "~16.4.2",
  "react-dom": "~16.4.2",
},
```

you use the tilde (~) character to tell npm to only update a package when it is patched. So, npm can update react 16.4.2 to 16.4.3 (or 16.4.4, etc.), but it will not accept an update to the major or minor version. So, 16.4.2 will not

get updated to 16.5.0.

You can also use the caret (^) symbol to specify that npm can update the minor version number.

```
"dependencies": {
  "react": "^16.4.2",
  "react-dom": "^16.4.2",
},
```

Using this notation, npm can update react 16.4.2 to 16.5.0 (or 16.5.1, 16.6.0, etc.), but it will not accept an update to the major version. So, 16.4.2 will not get updated to 17.0.0.

When npm updates packages, it generates a *package-lock.json* file, which lists the actual npm package versions used in your app, including all nested packages. While *package.json* controls the direct dependencies for your app, it does not control nested dependencies (other npm packages required by a particular npm package). You can use the *package-lock.json* file in your development cycle if you need to make sure that other developers and testers are using the exact packages that you are using, including nested packages. For more information, see package-lock.json in the npm documentation.

For Visual Studio, the *package-lock.json* file is not added to your project, but you can find it in the project folder.