Contents

JavaScript and TypeScript in Visual Studio 2019

4/17/2020 • 2 minutes to read • Edit Online

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.

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`.

Remove default imports (ASP.NET Core projects)

In older 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\Microsoft.
TypeScript.Default.props"

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

<Import

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

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

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.

JavaScript in Visual Studio 2017

4/20/2020 • 10 minutes to read • Edit Online

JavaScript is a first-class language in Visual Studio. You can use most or all of the standard editing aids (code snippets, IntelliSense, and so on) when you write JavaScript code in the Visual Studio IDE. You can write JavaScript code for many application types and services.

> **NOTE**
>
> We have joined the community-wide effort to make MDN web docs the web's one-stop, premiere development resource, by redirecting all (500+ pages) of Microsoft's JavaScript API reference from docs.microsoft.com to their MDN counterparts. For details, see this announcement.

Support for ECMAScript 2015 (ES6) and beyond

Visual Studio now supports syntax for ECMAScript language updates such as ECMAScript 2015/2016.

What is ECMAScript 2015?

JavaScript is still evolving as a programming language and TC39 is the committee responsible for making updates. ECMAScript 2015 is an update to the JavaScript language that brings helpful new syntax and functionality. For a deep dive on ES6 features, check out this reference site.

In addition to support for ECMAScript 2015, Visual Studio also supports ECMAScript 2016 and will have support for future versions of ECMAScript as they are released. To keep up with TC39 and the latest changes in ECMAScript, follow their work on github.

Transpile JavaScript

A common problem with JavaScript is that you want to use the latest ES6+ language features because they help you be more productive, but your runtime environments (often browsers) don't yet support these new features. This means that you either have to keep track of which browsers support what features (which can be tedious), or you need a way to convert your ES6+ code into a version that your target runtimes understand (usually ES5). Converting your code to a version that the runtime understands is commonly referred to as "transpiling".

One of the key features of TypeScript is the ability transpile ES6+ code to ES5 or ES3 so that you can write the code that makes you most productive, but still run your code on any platform. Because JavaScript in Visual Studio 2017 uses the same language service as TypeScript, it too can take advantage of ES6+ to ES5 transpilation.

Before transpilation can be set up, some understanding of the configuration options is required. TypeScript is configured via a `tsconfig.json` file. In the absence of such a file, some default values are used. For compatibility reasons, these defaults are different in a context where only JavaScript files (and optionally `.d.ts` files) are present. To compile JavaScript files, a `tsconfig.json` file must be added, and some of these options must be set explicitly.

The required settings for the tsconfig file are as follows:

- `allowJs`: This value must be set to `true` for JavaScript files to be recognized. The default value is `false`, because TypeScript compiles to JavaScript, and the compiler should not include files it just compiled.
- `outDir`: This value should be set to a location not included in the project, in order that the emitted JavaScript files are not detected and then included in the project (see `exclude`).
- `module`: If using modules, this setting tells the compiler which module format the emitted code should use (for example `commonjs` for Node, or bundlers such as Browserify).

- `exclude`: This setting states which folders not to include in the project. The output location, as well as non-project folders such as `node_modules` or `temp`, should be added to this setting.
- `enableAutoDiscovery`: This setting enables the automatic detection and download of definition files as outlined previously.
- `compileOnSave`: This setting tells the compiler if it should recompile any time a source file is saved in Visual Studio.
- `typeAcquisition`: This set of settings control the behavior of automatic type acquisition (further explain in this section)

In order to convert JavaScript files to CommonJS modules and place them in an `./out` folder, you could use the following `tsconfig.json` file:

```
{
  "compilerOptions": {
    "module": "commonjs",
    "allowJs": true,
    "outDir": "out"
  },
  "exclude": [
    "node_modules",
    "wwwroot",
    "out"
  ],
  "compileOnSave": true,
  "typeAcquisition": {
    "enable": true
  }
}
```

With the settings in place, if a source file (`./app.js`) existed and contained several ECMAScript 2015 language features as follows:

```
import {Subscription} from 'rxjs/Subscription';  // ES6 import

class Foo {  // ES6 Class
    sayHi(name) {
        return `Hi ${name}, welcome to Salsa!`;  // ES6 template string
    }
}

export let sqr = x => x * x;  //ES6 export, let, and arrow function
export default Subscription;  //ES6 default export
```

Then a file would be emitted to `./out/app.js` targeting ECMAScript 5 (the default) that looks something like the following:

```
"use strict";
var Subscription_1 = require('rxjs/Subscription');
var Foo = (function () {
    function Foo() {
    }
    Foo.prototype.sayHi = function (name) {
        return "Hi " + name + ", welcome to Salsa!";
    };
    return Foo;
}());
exports.sqr = function (x) { return x * x; };
Object.defineProperty(exports, "__esModule", { value: true });
exports.default = Subscription_1.Subscription;
```

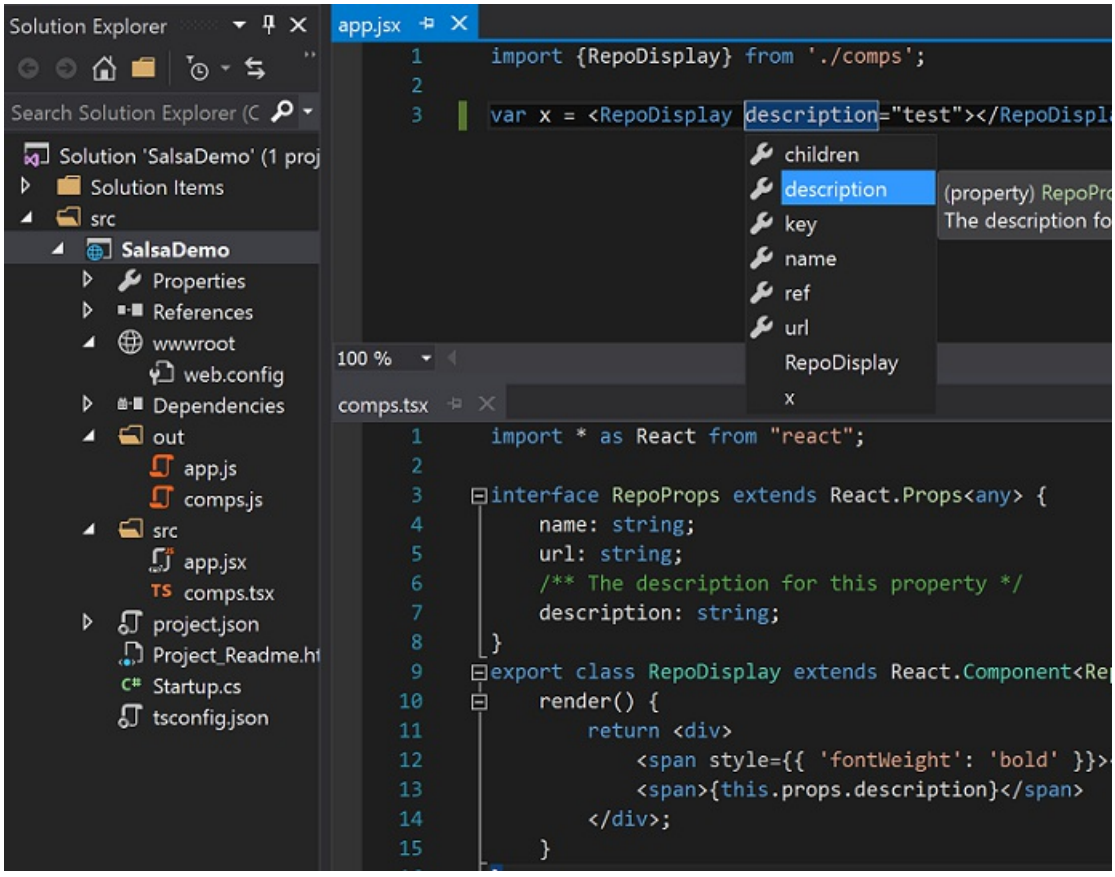
Better IntelliSense

JavaScript IntelliSense in Visual Studio 2017 will now display a lot more information on parameters 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. You can read more about the new IntelliSense experience and how it works here.

JSX syntax support

JavaScript in Visual Studio 2017 has rich support for the JSX syntax. JSX is a syntax set that allows HTML tags within JavaScript files.

The following illustration shows a React component defined in the `comps.tsx` TypeScript file, and then this component being used from the `app.jsx` file, complete with IntelliSense for completions and documentation within the JSX expressions. You don't need TypeScript here, this specific example just happens to contain some TypeScript code as well.



NOTE

To convert the JSX syntax to React calls, the setting `"jsx": "react"` must be added to the `compilerOptions` in the `tsconfig.json` file.

The JavaScript file created at `./out/app.js' upon build would contain the code:

```
"use strict";
var comps_1 = require('./comps');
var x = React.createElement(comps_1.RepoDisplay, {description: "test"});
```

Configure your JavaScript project

The language service is powered by static analysis, which means it analyzes your source code without actually executing it in order to return IntelliSense results and provide other editing features. Therefore, the larger the quantity and size of files that are included your project context, the more memory and CPU will be used during analysis. Because of this, there are a few default assumptions that are made about your project shape:

- `package.json` and `bower.json` list dependencies used by your project and by default are included in Automatic Type Acquisition (ATA)
- A top level `node_modules` folder contains library source code and its contents are excluded from the project context by default
- Every other `.js`, `.jsx`, `.ts`, and `.tsx` file is possibly one of *your own* source files and must be included in project context

In most cases, you will be able to just open your project and have great experience using the default project configuration. However, in projects that are large or have different folder structures, it may be desirable to further configure the language service to better focus only on your own source files.

Override defaults

You can override the default configuration by adding a `tsconfig.json` file to your project root. A `tsconfig.json` has several different options that can manipulate your project context. A few of them are listed below, but for a full set of all options available, see the schema store.

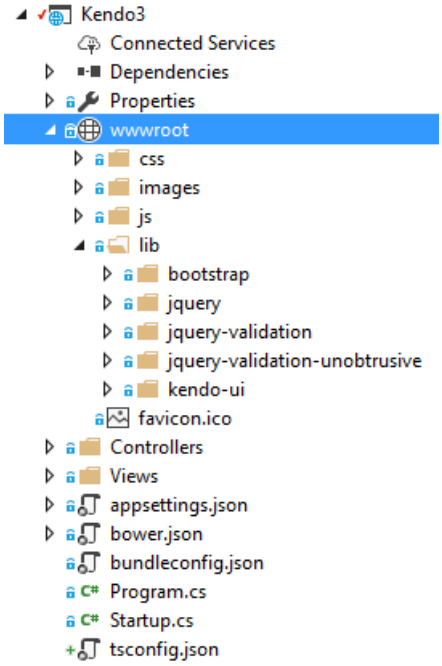
Important `tsconfig.json` options

```
{
  "compilerOptions": {
    "allowJs": true,          // include .js and .jsx in project context (defaults to only .ts and .tsx)
    "noEmit": true            // turns off downlevel compiler
  },
  "files": [],                // list of explicit files to include in the project context. Highest priority.
  "include": [],              // list of folders or glob patterns to include in the project context.
  "exclude": [],              // list of folders or glob patterns to exclude. Overridden by files array.
  "typeAcquisition": {
    "enable": true,           // Defaulted to "false" with a tsconfig. Enables better IntelliSense in JS.
    "include": [ "jquery" ],  // Specific libs to fetch .d.ts files that weren't picked up by ATA
    "exclude": [ "node" ]     // Specific libs to not fetch .d.ts files for
  }
}
```

Example project configuration

Given a project with the following setup:

- project's source files are in `wwwroot/js`
- project's lib files are in `wwwroot/lib`
- `bootstrap`, `jquery`, `jquery-validation`, and `jquery-validation-unobtrusive` are listed in the `bower.json`
- `kendo-ui` has been manually added to the lib folder



You could use the following `tsconfig.json` to make sure the language service only analyzes your source files in the `js` folder, but still fetches and uses `.d.ts` files for the libraries in your `lib` folder.

```
{
  "compilerOptions": {
    "allowJs": true,
    "noEmit": true
  },
  "exclude": ["wwwroot/lib"],  //ignore lib folders, we will get IntelliSense via ATA
  "typeAcquisition": {
    "enable": true,
    "include": [ "kendo-ui" ]  //kendo-ui wasn't added via bower, so we need to list it here
  }
}
```

Troubleshooting The JavaScript language service has been disabled for the following project(s)

When you open a JavaScript project that has a very large amount of content, you might get a message that reads "The JavaScript language service has been disabled for the following project(s)". The most common reason for having a very large amount of JavaScript source is due to including libraries with source code that exceeds a 20MB project limit.

A simple way to optimize your project is to add a `tsconfig.json` file in your project root to let the language service know which files are safe to ignore. Use the sample below to exclude the most common directories where libraries are stored:

```
{
  "compilerOptions": {
    "allowJs": true,
    "allowSyntheticDefaultImports": true,
    "maxNodeModuleJsDepth": 2,
    "noEmit": true,
    "skipLibCheck": true
  },
  "exclude": [
    "**/bin",
    "**/bower_components",
    "**/jspm_packages",
    "**/node_modules",
    "**/obj",
    "**/platforms"
  ]
}
```

Add more directories as you see fit. Some other examples include "vendor" or "wwwroot/lib" directories.

> **NOTE**
> The compiler property `disableSizeLimit` can be used as well to disable the 20MB check limit. Take special precautions when using this property because disabling the limit might crash the language service.

Notable Changes from Visual Studio 2015

As Visual Studio 2017 features a completely new language service, there are a few behaviors that will be different or absent from the previous experience. The most notable changes are the replacement of VSDoc with JSDoc, the removal of custom `.intellisense.js` extensions, and limited IntelliSense for specific code patterns.

No more `///<references/>` or `_references.js`

Previously it was fairly complicated to understand at any given moment which files were in your IntelliSense scope. Sometimes it was desirable to have all your files in scope and other times it wasn't, and this led to complex configurations involving manual reference management. Going forward you no longer need to think about reference management and so you don't need triple slash references comments or `_references.js` files.

See the JavaScript IntelliSense page for more info on how IntelliSense works.

VSDoc

XML documentation comments, sometimes referred to as VSDocs, could previously be used to decorate your source code with additional data that would be used to buff up IntelliSense results. VSDoc is no longer supported in favor of JSDoc which is easier to write and the accepted standard for JavaScript.

`.intellisense.js` extensions

Previously, you could author IntelliSense extensions which would allow you to add custom completion results for third-party libraries. These extensions were fairly difficult to write and installing and referencing them was cumbersome, so going forward the new language service won't support these files. As an easier alternative, you can write a TypeScript definition file to provide the same IntelliSense benefits as the old `.intellisense.js` extensions. You can learn more about declaration (`.d.ts`) file authoring here.

Unsupported patterns

Because the new language service is powered by static analysis rather than an execution engine (read this issue for information of the differences), there are a few JavaScript patterns that no longer can be detected. The most common pattern is the "expando" pattern. Currently the language service cannot provide IntelliSense on objects that have properties tacked on after declaration. For example:

```
var obj = {};
obj.a = 10;
obj.b = "hello world";
obj. // IntelliSense won't show properties a or b
```

You can get around this by declaring the properties during object creation:

```
var obj = {
    "a": 10,
    "b": "hello world"
}
obj. // IntelliSense shows properties a and b
```

You can also add a JSDoc comment as follows:

```
/**
 * @type {{a: number, b: string}}
 */
var obj = {};
obj.a = 10;
obj.b = "hello world";
obj. // IntelliSense shows properties a and b
```

First look at the Visual Studio IDE

4/1/2020 • 5 minutes to read • Edit Online

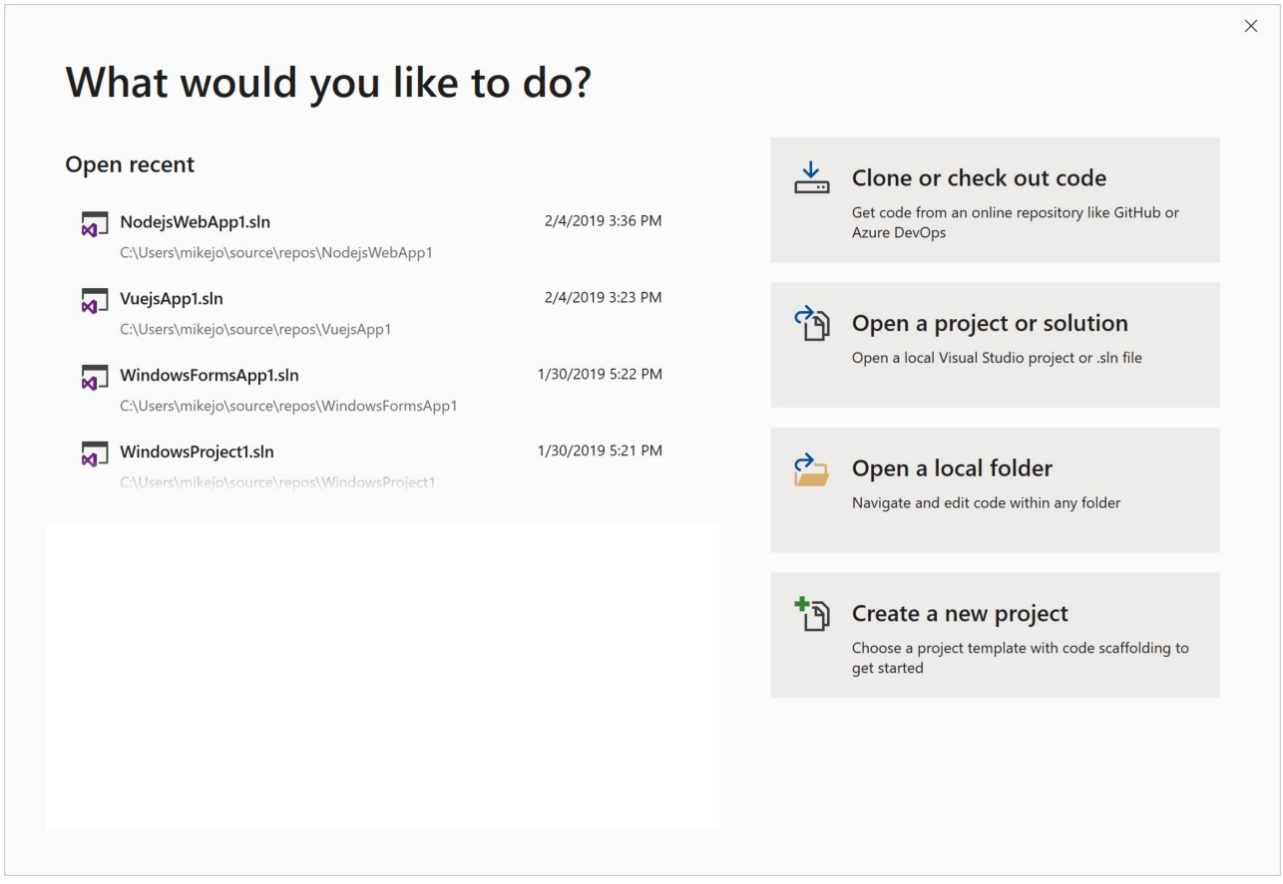
In this 5-10 minute introduction to the Visual Studio integrated development environment (IDE), we'll take a tour of some of the windows, menus, and other UI features.

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

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.



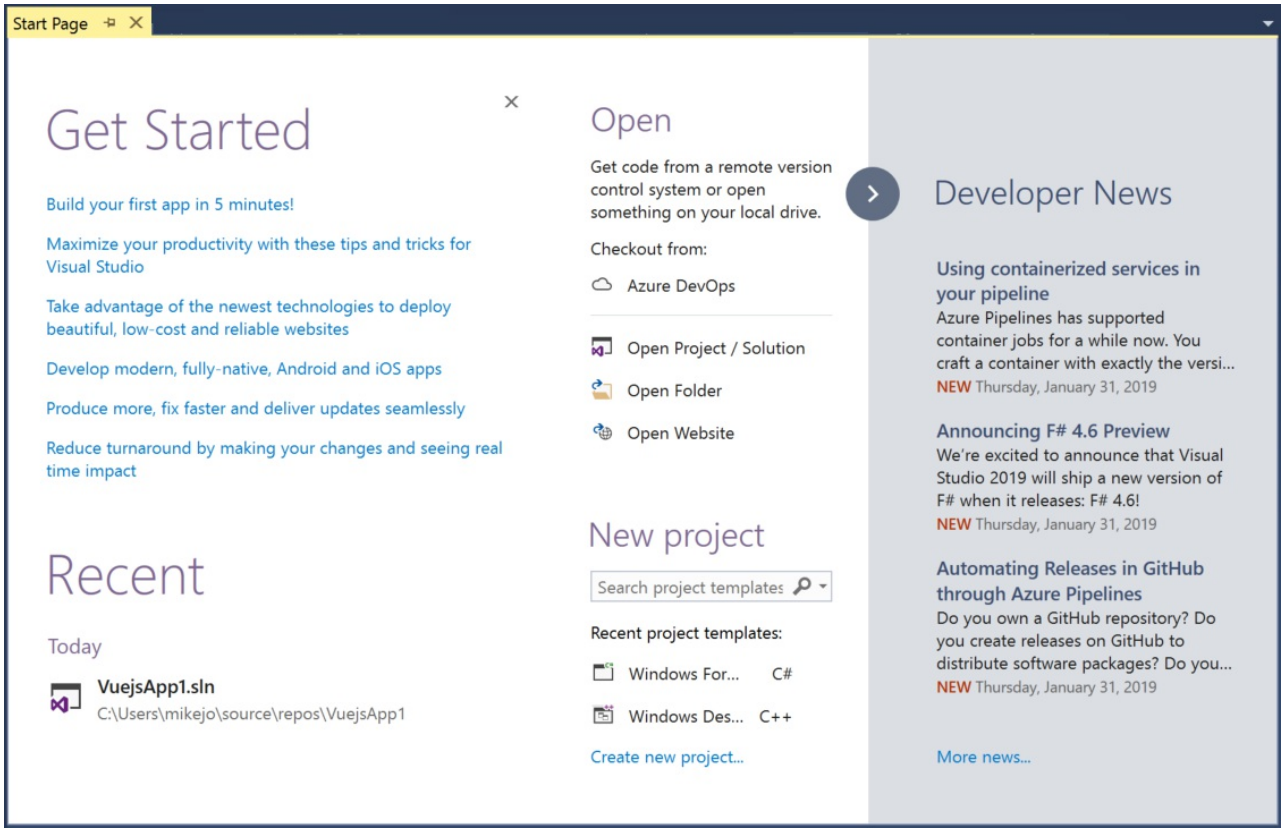
If this is the first time you're using Visual Studio, your recent projects list will be empty.

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.

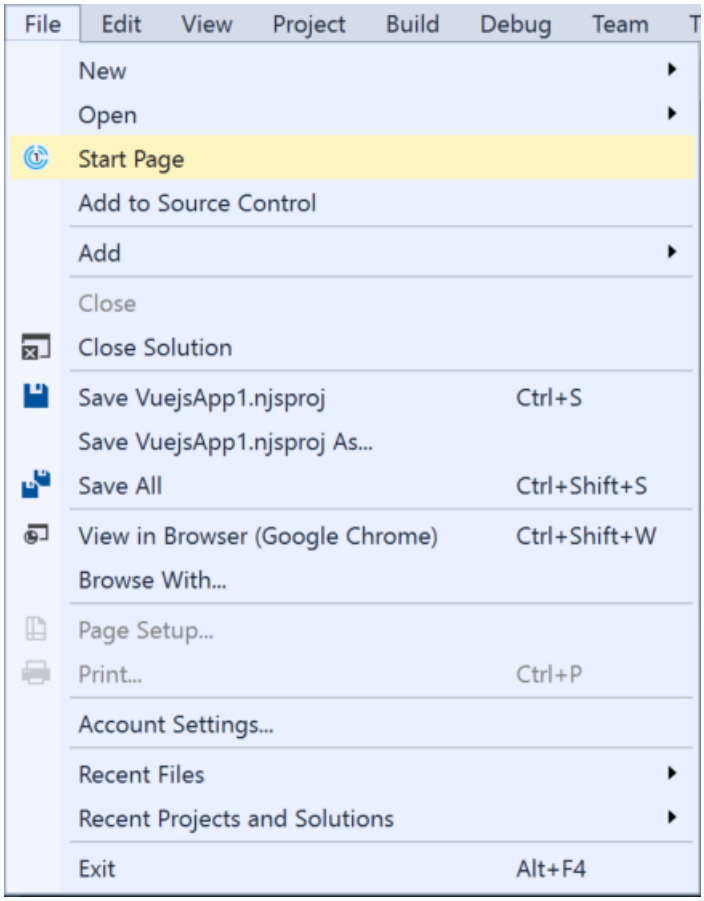
The **Continue without code** option simply opens the Visual Studio development environment without any specific project or code loaded. You might choose this option to join a Live Share session or attach to a process for debugging. You can also press **Esc** to close the start window and open the IDE.

Start Page

The first thing you'll see after you launch Visual Studio is most likely the **Start Page**. The **Start Page** is designed as a "hub" to help you find the commands and project files you need faster. The **Recent** section displays projects and folders you've worked on recently. Under **New project**, you can click a link to bring up the **New Project** dialog box, or under **Open**, you can open an existing code project or folder. On the right is a feed of the latest developer news.



If you close the **Start Page** and want to see it again, you can reopen it from the **File** menu.

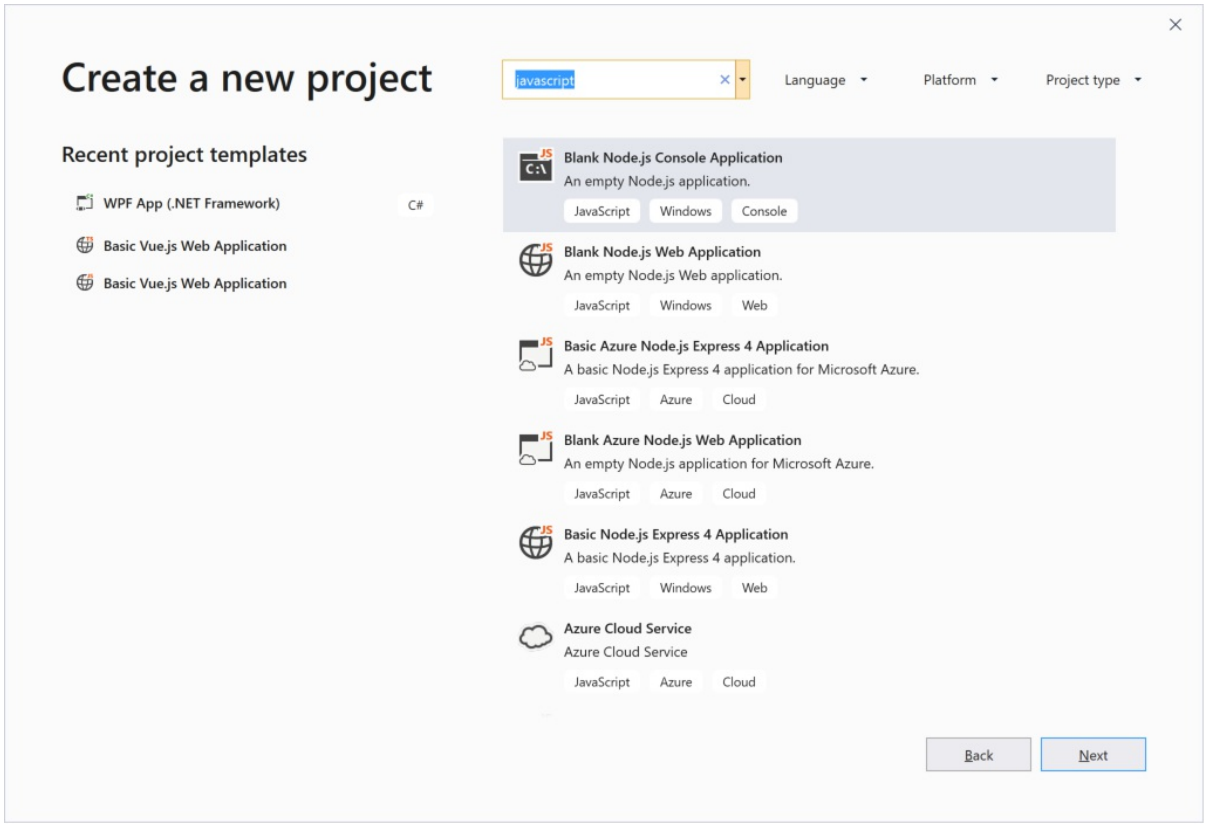


Create a project

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.

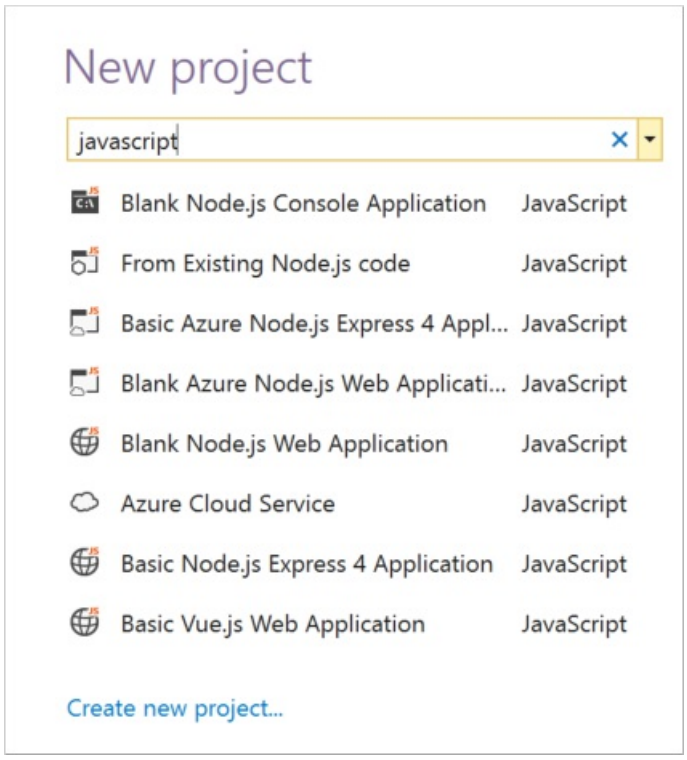
   Visual Studio provides various kinds of project templates that help you get started coding quickly. (Alternatively, if you're a TypeScript developer, feel free to create a project in that language. The UI we'll be looking at is similar for all programming languages.)



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**.

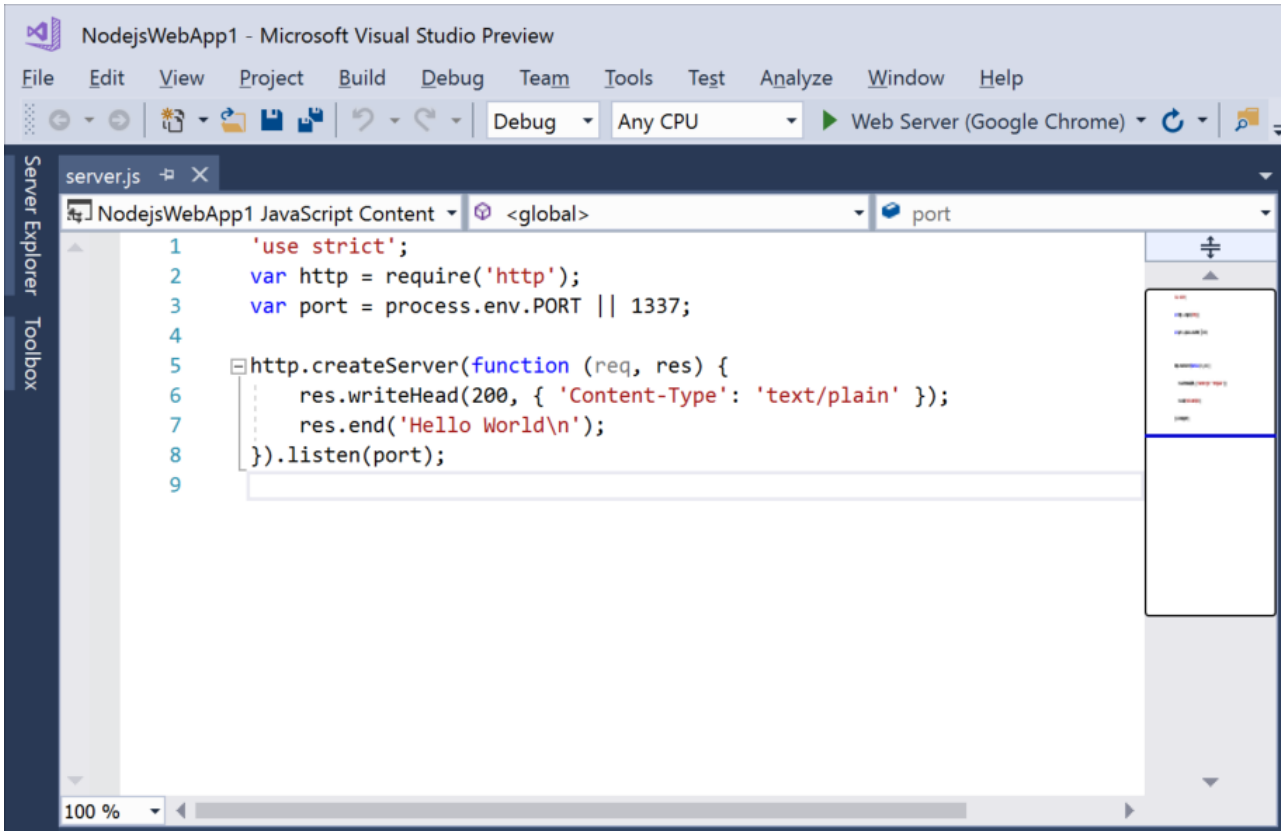
1. On the **Start Page**, in the search box under **New project**, type in **javascript** to filter the list of project types to those that contain "javascript" in their name or language type.



Visual Studio provides various kinds of project templates that help you get started coding quickly. Choose a **Blank Node.js Web Application** project template. (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. In the **New Project** dialog box that appears, accept the default project name and choose **OK**.

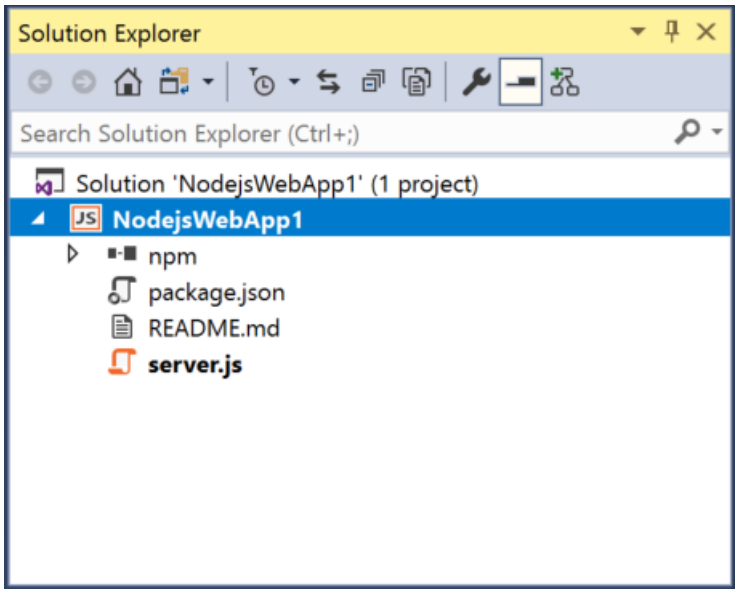
The project is created and a file named *server.js* opens in the **Editor** window. The **Editor** shows the contents of files, and is where you'll do most of your coding work in Visual Studio.



Solution Explorer

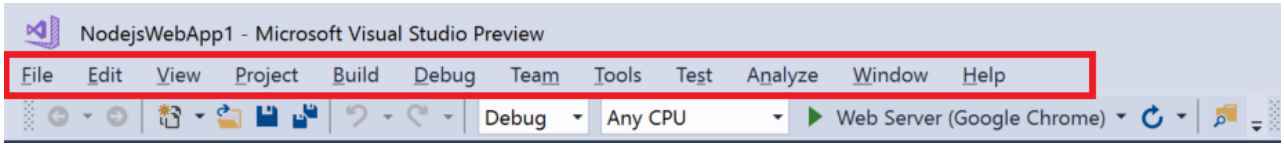
Solution Explorer, which is typically on the right-hand side of Visual Studio, shows you a graphical representation

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

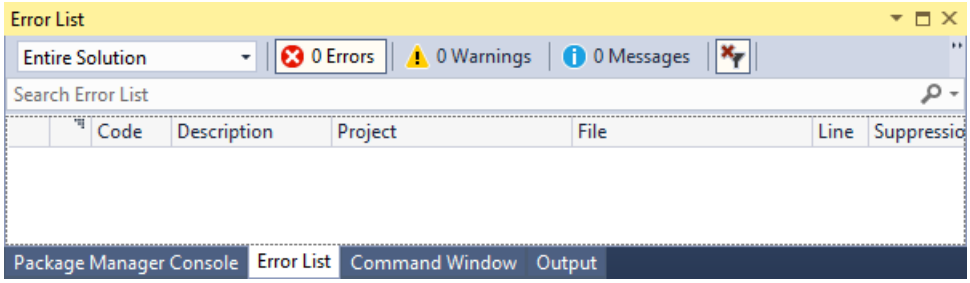
The menu bar along the top of Visual Studio groups commands into categories. For example, the **Project** menu contains commands related to the project you're working in. On the **Tools** menu, you can customize how Visual Studio behaves by selecting **Options**, or add features to your installation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



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

Error List

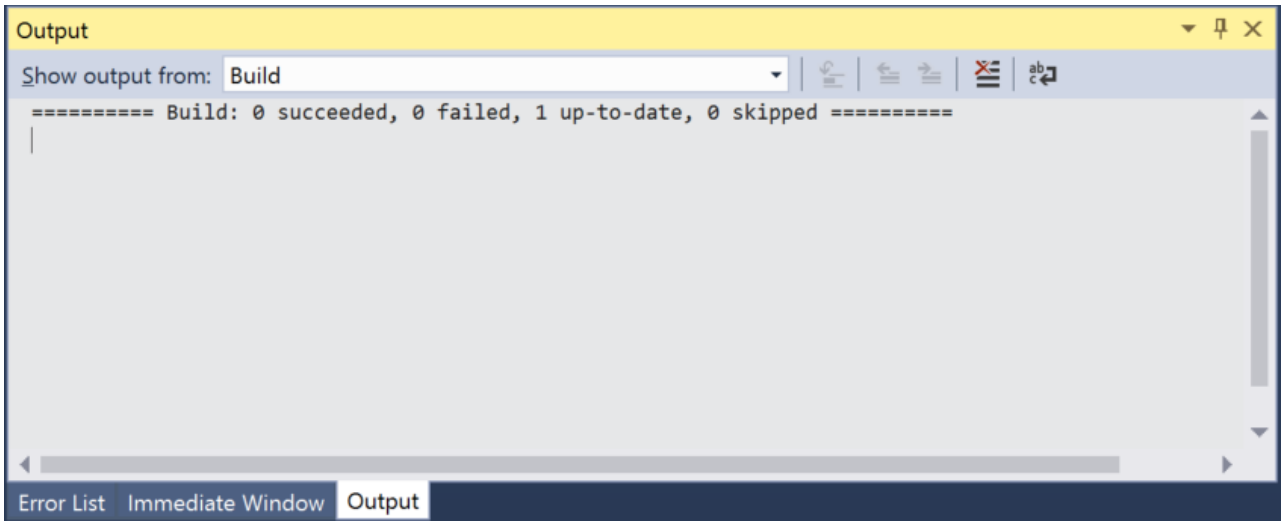
The **Error List** shows you errors, warning, and messages regarding the current state of your code. If there are any errors (such as a missing brace or semicolon) in your file, or anywhere in your project, they're listed here.



Output window

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

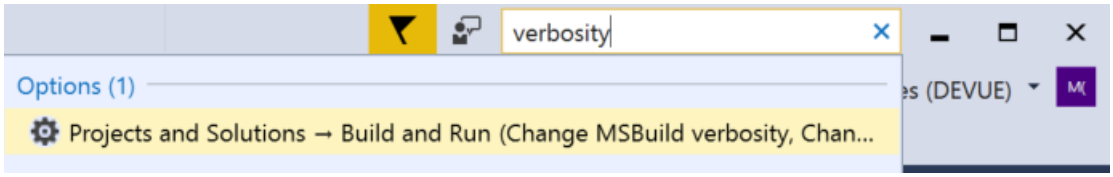
Let's build the project to see some build output. From the **Build** menu, choose **Build Solution**. The **Output** window automatically obtains focus and display a successful build message.



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.

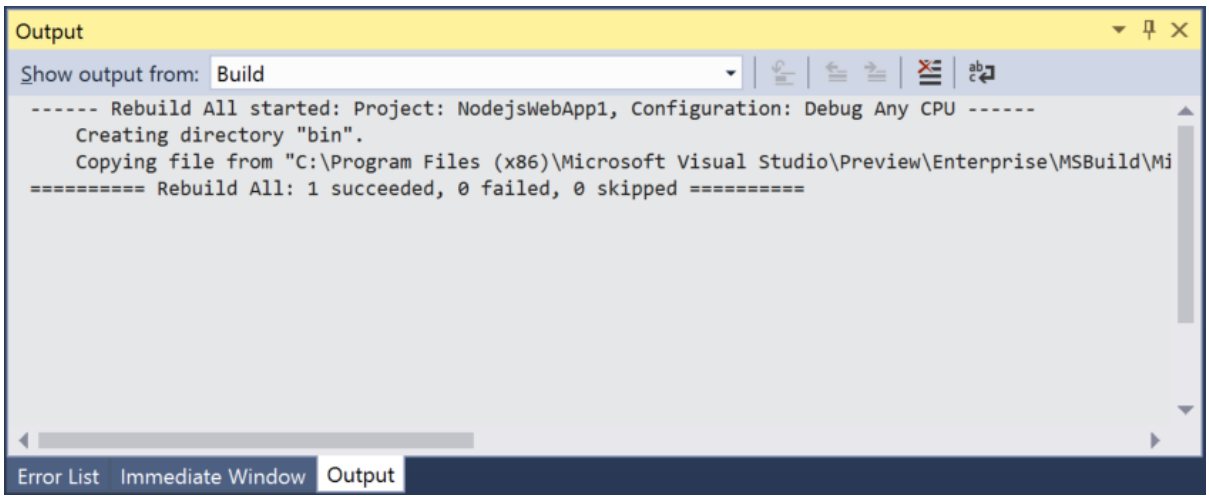
   

   The **Options** dialog box opens to the **Build and Run** options page.

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

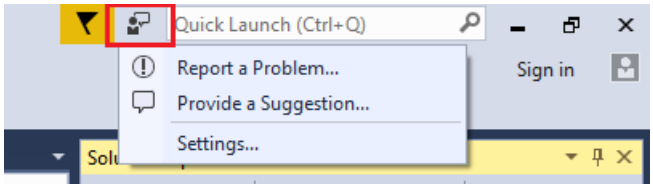
3. Build the project again by right-clicking on the **NodejsWebApp1** project in **Solution Explorer** and choosing **Rebuild** from the context menu.

   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

Quickstart: Use Visual Studio to create your first Vue.js app

4/21/2020 • 4 minutes to read • Edit Online

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

> **IMPORTANT**
> This article requires the Vue.js template, which is available starting in Visual Studio 2017 version 15.8.

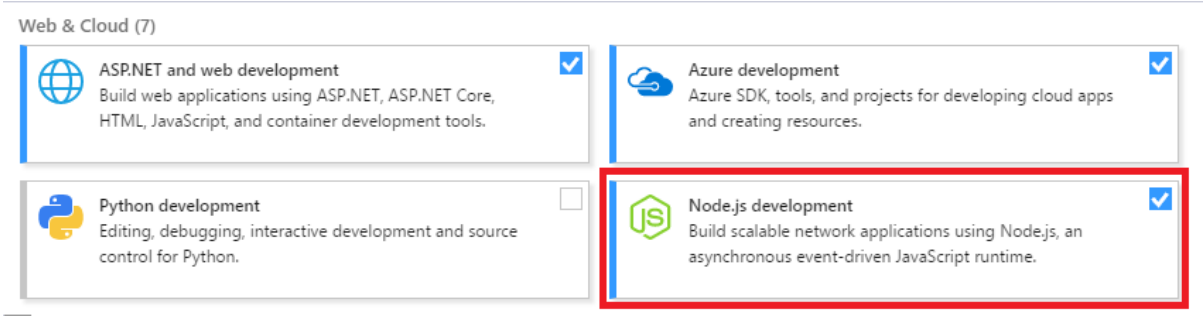
Prerequisites

- 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 haven't already installed Visual Studio 2017, 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, we recommend you install the LTS version from the Node.js website for best compatibility with outside frameworks and libraries. Node.js is built for 32-bit and 64-bit architectures. The Node.js tools in Visual Studio, included in the Node.js workload, support both versions. Only one is required and the Node.js installer only supports one being installed at a time.

  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, choose **Properties**, and set the **Node.exe path**). You can use a global installation of Node.js or you can specify the path to a local interpreter in each of your Node.js projects.

Create a project

First, you'll create a Vue.js web application project.

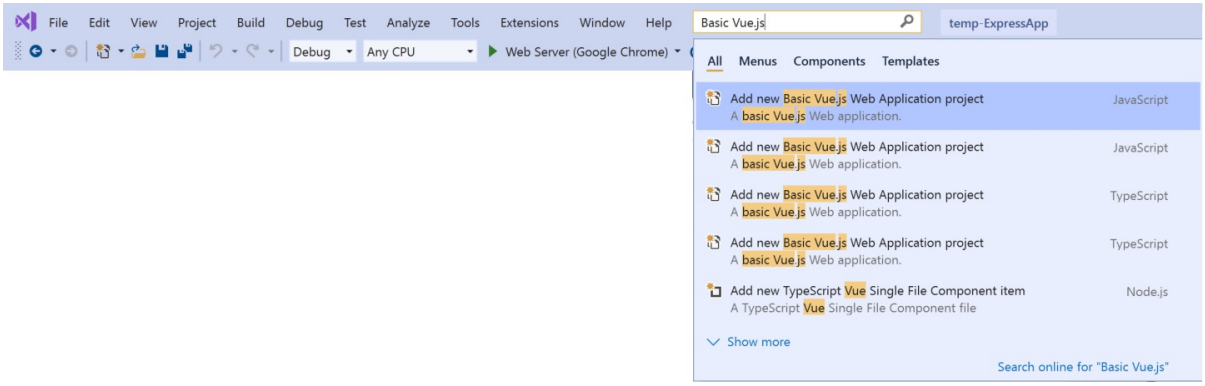
1. If you don't have the Node.js runtime already installed, install the LTS version from the Node.js website.

For more information, see the prerequisites.

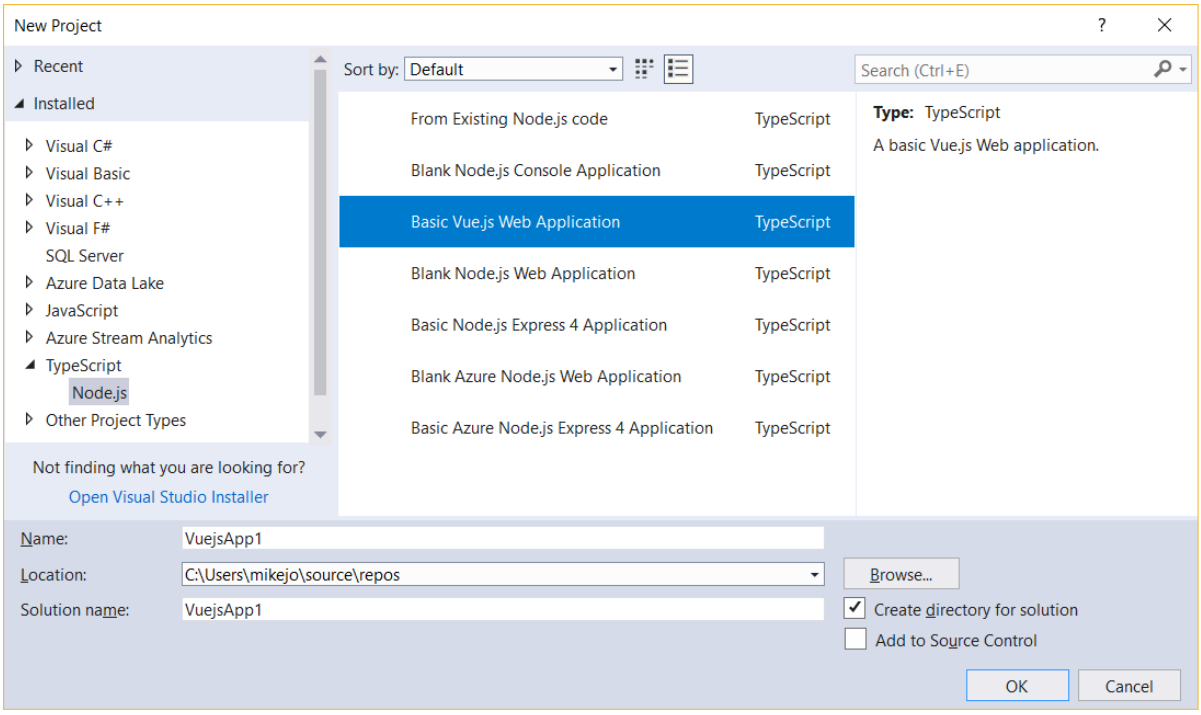
2. Open Visual Studio.

3. Create a new project.

   Press **Esc** to close the start window. Type **Ctrl + Q** to open the search box, type **Basic Vue.js**, then choose **Basic Vue.js Web application** (either JavaScript or TypeScript). In the dialog box that appears, type the name **basic-vuejs**, and then choose **Create**.



   From the top menu bar, choose **File** > **New** > **Project**. In the left pane of the **New Project** dialog box, expand **JavaScript** or **TypeScript**, then choose **Node.js**. In the middle pane, choose **Basic Vue.js Web application**, type the name **basic-vuejs**, and then choose **OK**.



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

   Visual Studio creates the new project. The new project opens in Solution Explorer (right pane).

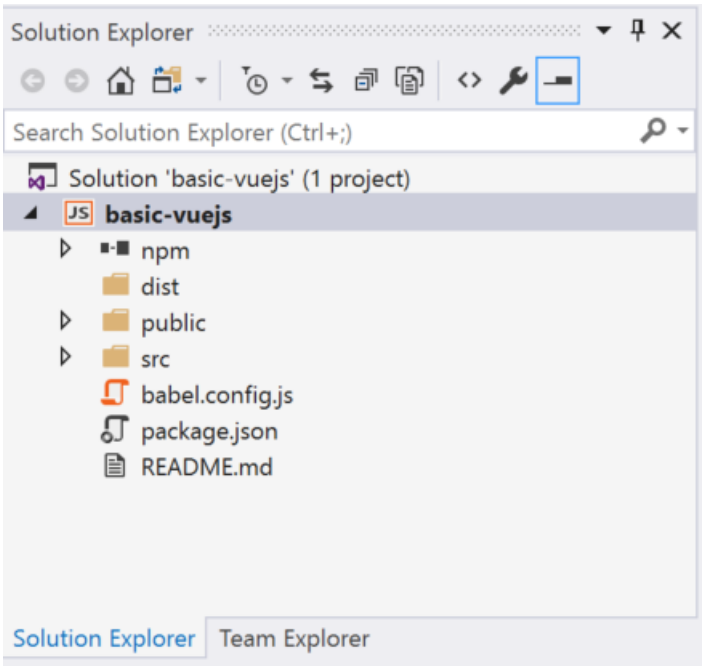
4. Check the Output window (lower pane) for progress on installing the npm packages required for the application.

5. In Solution Explorer, open the **npm** node and make sure that all the listed npm packages are installed.

   If any packages are missing (exclamation point icon), you can right-click the **npm** node and choose **Install Missing npm Packages**.

Explore the IDE

1. Take a look at **Solution Explorer** in the right pane.



   - Highlighted in bold is your project, using the name you gave in the **New Project** dialog box. On disk, this project is represented by a *.njsproj* file in your project folder.
   - 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.
   - 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.

2. If you want to install npm packages or run Node.js commands from a command prompt, right-click the project node and choose **Open Command Prompt Here**.

Add a .vue file to the project

1. In Solution Explorer, right-click any folder such as the *src/components* folder, and then choose **Add** > **New Item**.

2. Select either **JavaScript Vue Single File Component** or **TypeScript Vue Single File Component**, and then click **Add**.

   Visual Studio adds the new file to the project.

Build the project

1. Next, choose **Build** > **Build Solution** to build the project.

2. Check the **Output** window to see build results, and choose **Build** from the **Show output from** list.

1. (TypeScript project only) From Visual Studio, choose **Build** > **Clean Solution**.

2. Next, choose **Build** > **Build Solution** to build the project.

3. Check the **Output** window to see build results, and choose **Build** from the **Show output from** list.

The JavaScript Vue.js project template (and older versions of the TypeScript template) use the `build` npm script by configuring a post build event. If you want to modify this setting, open the project file (*<projectname>.njsproj*)

from Windows Explorer and locate this line of code:

```
<PostBuildEvent>npm run build</PostBuildEvent>
```

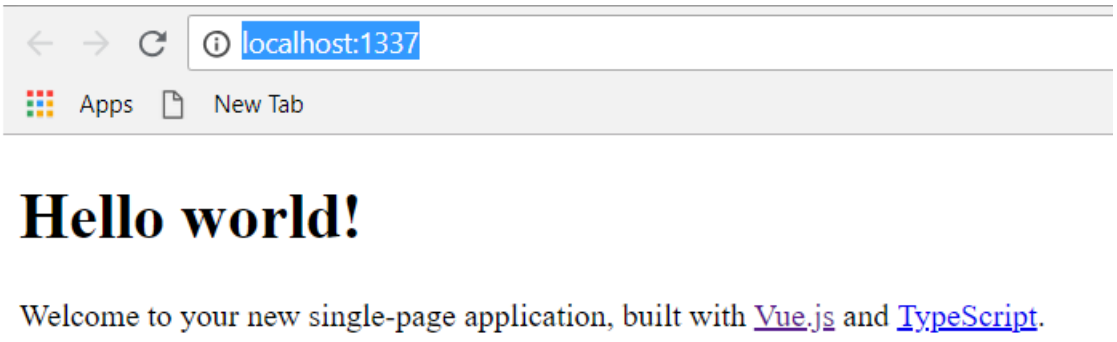
Run the application

1. Press **Ctrl+F5** (or **Debug > Start Without Debugging**) to run the application.

   In the console, you see a message *Starting Development Server*.

   Then, the app opens in a browser.

   If you don't see the running app, refresh the page.



2. Close the web browser.

Congratulations on completing this Quickstart! We hope you learned a little bit about using the Visual Studio IDE with Vue.js. If you'd like to delve deeper into its capabilities, continue with a tutorial in the **Tutorials** section of the table of contents.

Next steps

- Go through the article for Vue.js
- Go through the Tutorial for Node.js and Express
- Deploy the app to Linux App Service

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

4/21/2020 • 8 minutes to read • Edit Online

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

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

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

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 server-side.

What is npm?

npm is the default package manager for the Node.js. The package manager makes it easier for programmers to publish and share source code of Node.js libraries and is designed to simplify installation, updating, and uninstallation of libraries.

What is express?

Express is a web application framework, used as a server framework for Node.js to build web applications. Express allows you to choose different front-end frameworks to create a UI, such as Pug (formerly called Jade). Pug is used in this tutorial.

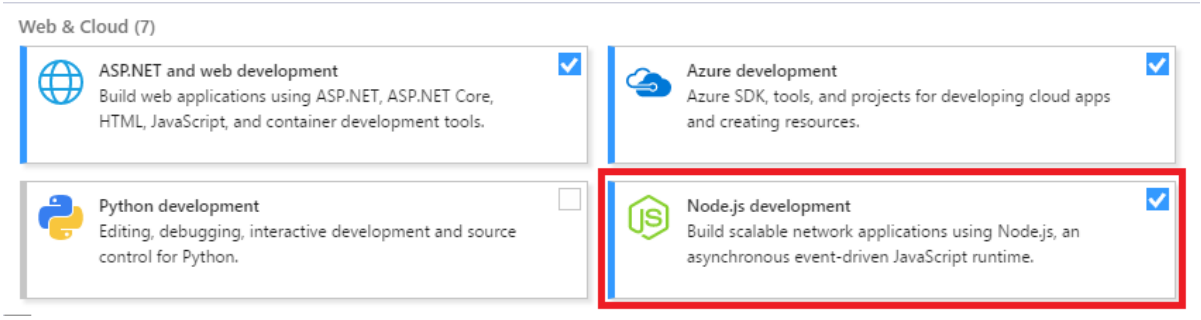
Prerequisites

- 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 haven't already installed Visual Studio 2017, 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, we recommend you install the LTS version from the Node.js website for best compatibility with outside frameworks and libraries. Node.js is built for 32-bit and 64-bit architectures. The Node.js tools in Visual Studio, included in the Node.js workload, support both versions. Only one is required and the Node.js installer only supports one being installed at a time.

  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, choose **Properties**, and set the **Node.exe path**). You can use a global installation of Node.js or you can specify the path to a local interpreter in each of your Node.js projects.

  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 containing code for a Node.js and express app.

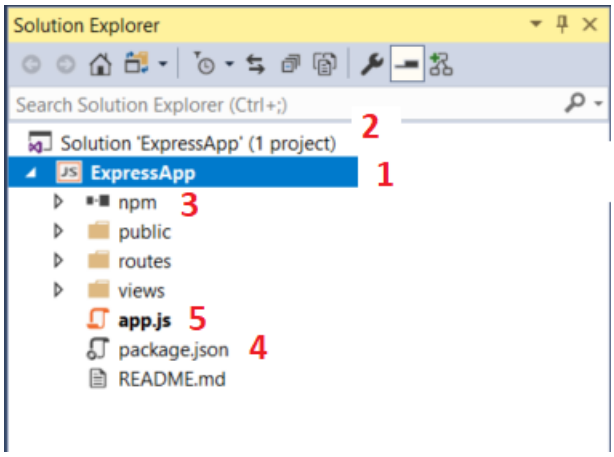
1. Open Visual Studio.
2. Create a new project.

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

   From the top menu bar, choose **File** > **New** > **Project**. In the left pane of the **New Project** dialog box, expand **JavaScript**, then choose **Node.js**. In the middle pane, choose **Basic Azure Node.js Express 4 application**, then choose **OK**.

   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 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 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 *app.js* show up under the project node. *app.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**.

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

   If any packages are missing (exclamation point icon), you can right-click the **npm** node and choose **Install npm Packages**.

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*. The code that sets the view engine in *app.js* is `app.set('view engine', 'pug');`.)

1. In Solution Explorer (right pane), open the views folder, then open *index.pug*.

2. Replace the content 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='200' width='200' src='')
```

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

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

```
var getData = function () {
    var data = {
        'item1': 'http://public-domain-photos.com/free-stock-photos-1/flowers/cactus-76.jpg',
        'item2': 'http://public-domain-photos.com/free-stock-photos-1/flowers/cactus-77.jpg',
        'item3': 'http://public-domain-photos.com/free-stock-photos-1/flowers/cactus-78.jpg'
    }
    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" });
});
```

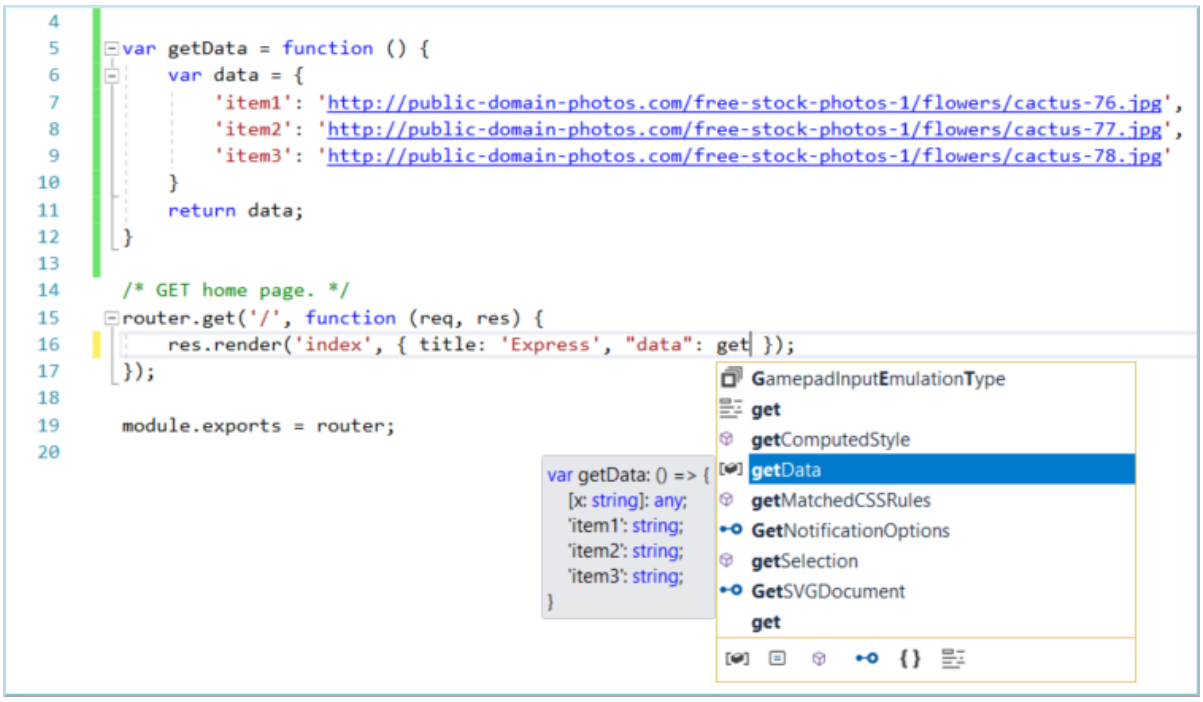
The preceding code sets the current page using the Express router object and renders the page, passing the title and data object to the page. The *index.pug* file is specified here as the page to load when *index.js* runs. *index.js* is configured as the default route in *app.js* code (not shown).

To demonstrate several features of Visual Studio, there's a deliberate error in the line of code containing `res.render`. You need to fix the error before the app can run, which you do in the next section.

Use IntelliSense

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

1. In *index.js*, go to the line of code containing `res.render`.
2. Put your cursor after the `data` string, type `: get` and IntelliSense will show you the `getData` function defined earlier in the code. Select `getData`.



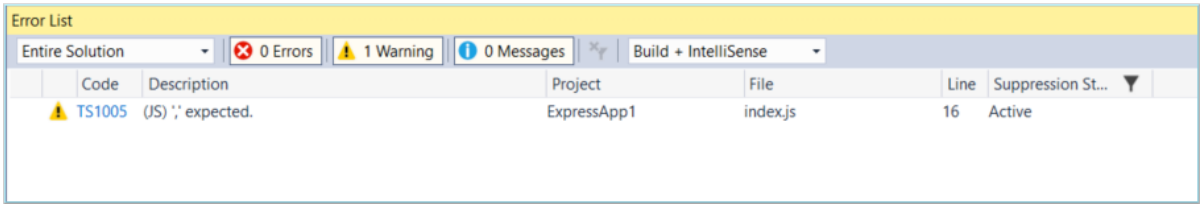
3. Add the parentheses to make it a function call, `getData()`.

4. Remove the comma (`,`) before `"data"` and you see green syntax highlighting on the expression. Hover over the syntax highlighting.

   The last line of this message tells you that the JavaScript interpreter expected a comma (`,`).

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

   You see the warning and description along with the filename and line number.



6. Fix the code by adding the comma (`,`) before `"data"`.

   When corrected, line of code should look like this:

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

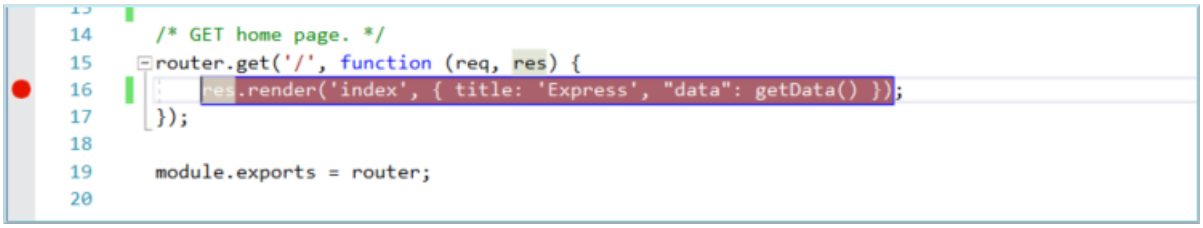
Set a breakpoint

You're next going to run the app with the Visual Studio debugger attached. Before doing that, you need to set a breakpoint.

1. In *index.js*, click in the left gutter before the following line of code to set a breakpoint:

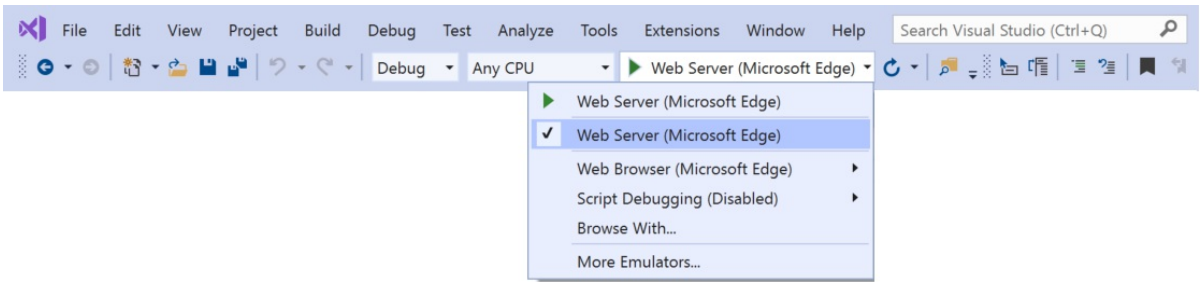
```
res.render('index', { title: 'Express', "data": getData() });
```

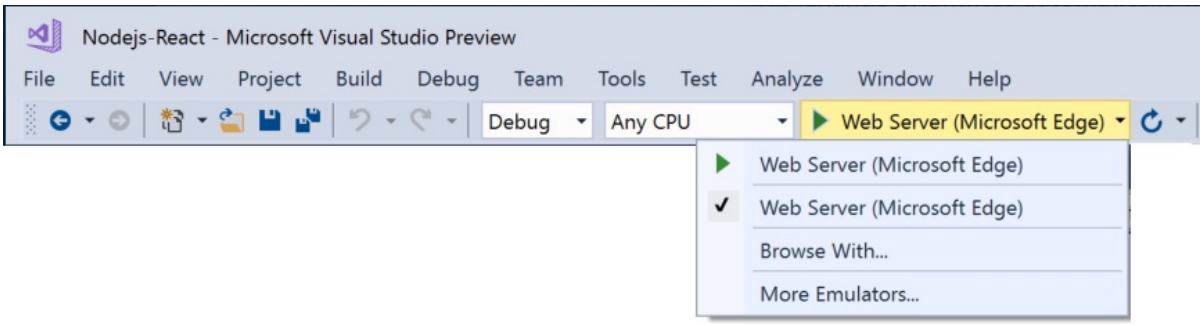
Breakpoints are the most basic and essential feature of reliable debugging. A breakpoint indicates where Visual Studio should suspend your running code so you can take a look at the values of variables, or the behavior of memory, or whether or not a branch of code is getting run.



Run the application

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



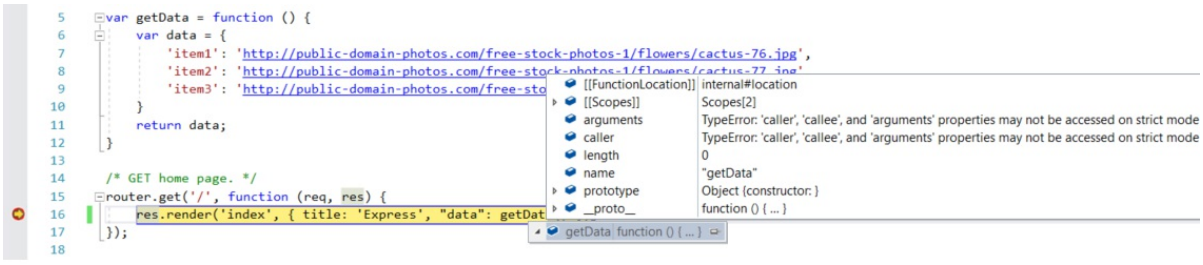


If Chrome is available on your machine, but does not show up as an option, choose **Browse With** from the debug target dropdown list, and select Chrome as the default browser target (choose **Set as Default**).

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

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

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

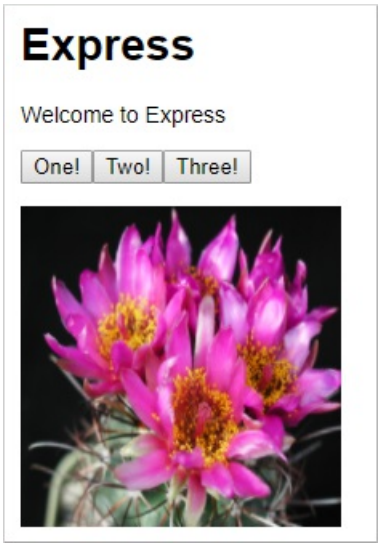


4. Press **F5** (**Debug** > **Continue**) to continue.

   The app opens in a browser.

   In the browser window, you will see "Express" as the title and "Welcome to Express" in the first paragraph.

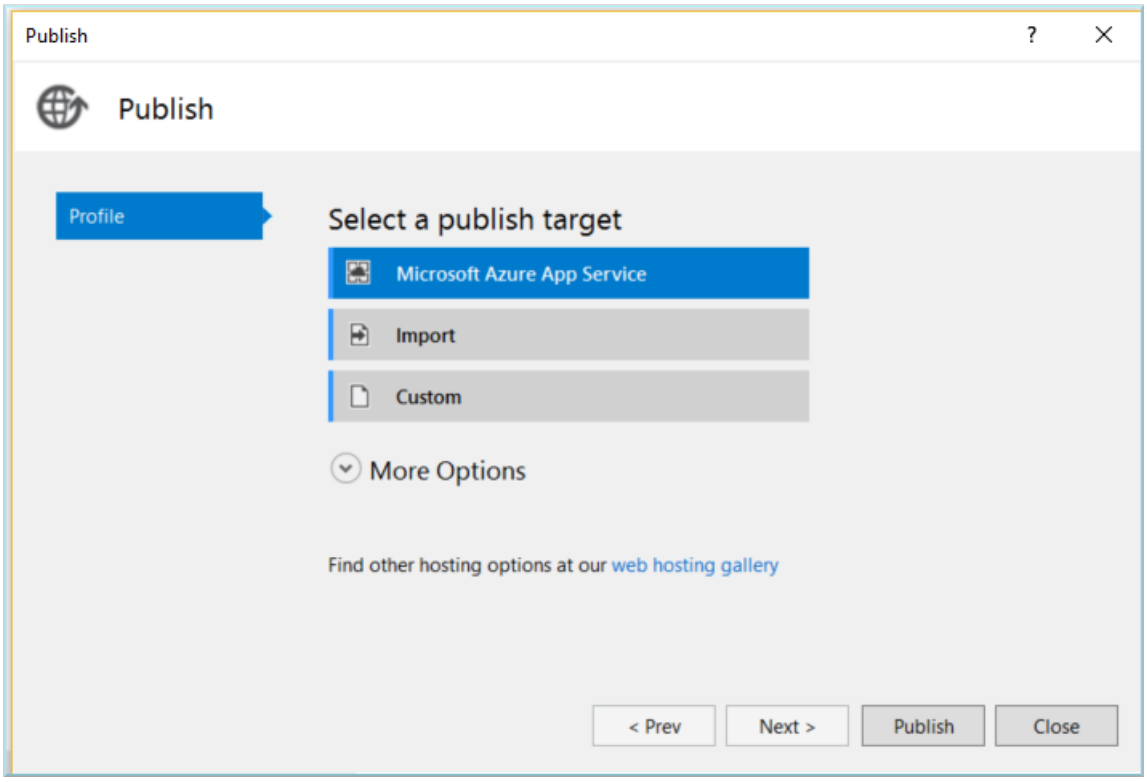
5. Click the buttons to display different images.



6. Close the web browser.

(Optional) Publish to Azure App Service

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



2. Choose **Microsoft Azure App Service**.

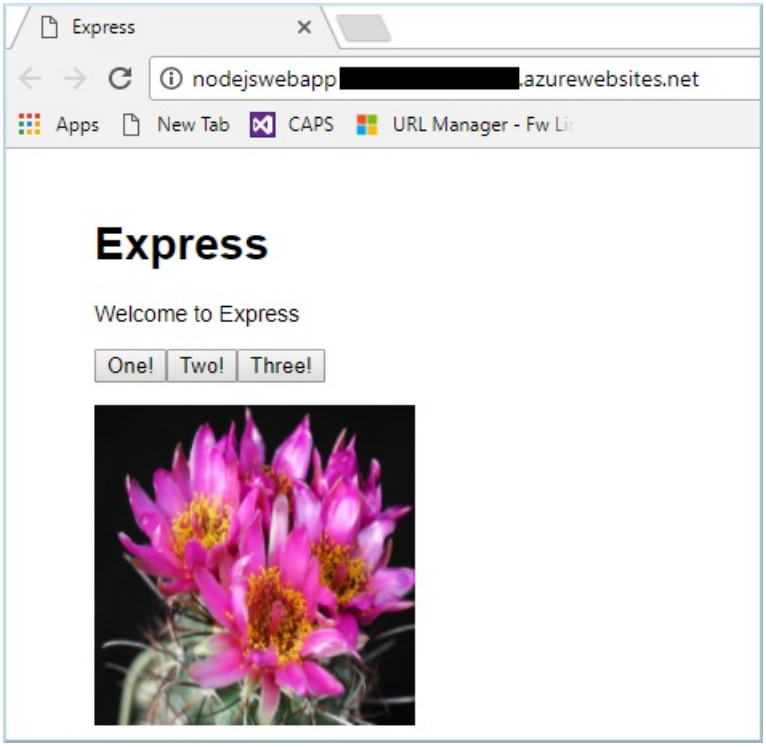
   In the **App Service** dialog box, you can sign into your Azure account and connect to existing Azure subscriptions.

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

4. The **Output** window shows progress on deploying to Azure.

   On successful deployment, your app opens in a browser running in Azure App Service. Click a button to

display an image.



Congratulations on completing this tutorial!

Next steps

Deploy the app to Linux App Service

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

4/21/2020 • 16 minutes to read • Edit Online

Visual Studio allows you to easily create a Node.js project and experience IntelliSense and other built-in features that support Node.js. In this tutorial for Visual Studio, you create a Node.js web application 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

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 server-side.

What is npm?

npm is the default package manager for the Node.js. The package manager makes it easier for programmers to publish and share source code of Node.js libraries and is designed to simplify installation, updating, and uninstallation of libraries.

What is React?

React is a front-end framework to create a UI.

What is JSX?

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

What is webpack?

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

Prerequisites

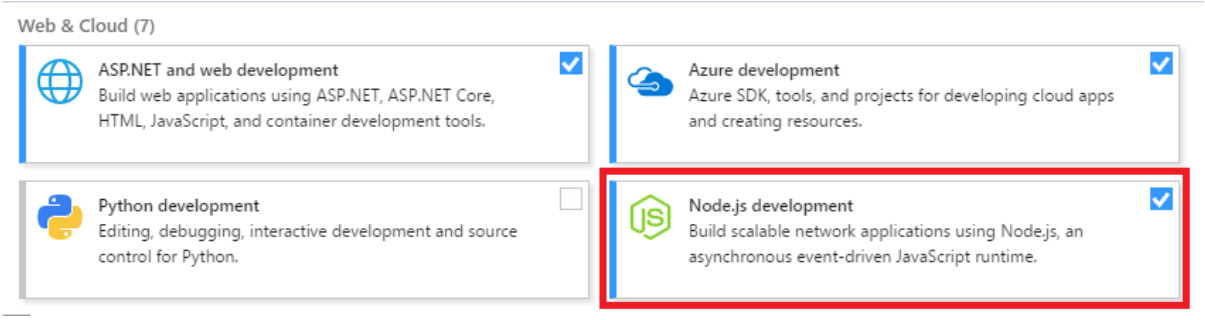
- You must have Visual Studio installed and the Node.js development workload.

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

  If you haven't already installed Visual Studio 2017, 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.

  This tutorial was tested with version 12.6.2.

  If you don't have it installed, we recommend you install the LTS version from the Node.js website for best compatibility with outside frameworks and libraries. Node.js is built for 32-bit and 64-bit architectures. The Node.js tools in Visual Studio, included in the Node.js workload, support both versions. Only one is required and the Node.js installer only supports one being installed at a time.

  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, choose **Properties**, and set the **Node.exe path**). You can use a global installation of Node.js or you can specify the path to a local interpreter in each of your Node.js projects.

Create a project

First, create a Node.js web application project.

1. Open Visual Studio.

2. Create a new project.

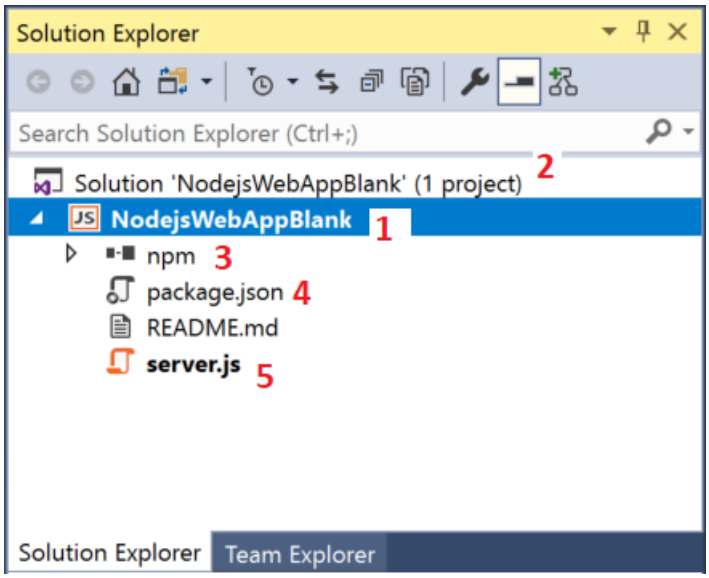
   Press **Esc** to close the start window. Type **Ctrl + Q** to open the search box, type **Node.js**, then choose **Blank Node.js Web Application - JavaScript**. (Although this tutorial uses the TypeScript compiler, the steps require that you start with the **JavaScript** template.)

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

   From the top menu bar, choose **File** > **New** > **Project**. In the left pane of the **New Project** dialog box, expand **JavaScript**, then choose **Node.js**. In the middle pane, choose **Blank Node.js Web Application**, type the name **NodejsWebAppBlank**, then choose **OK**.

   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**. 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

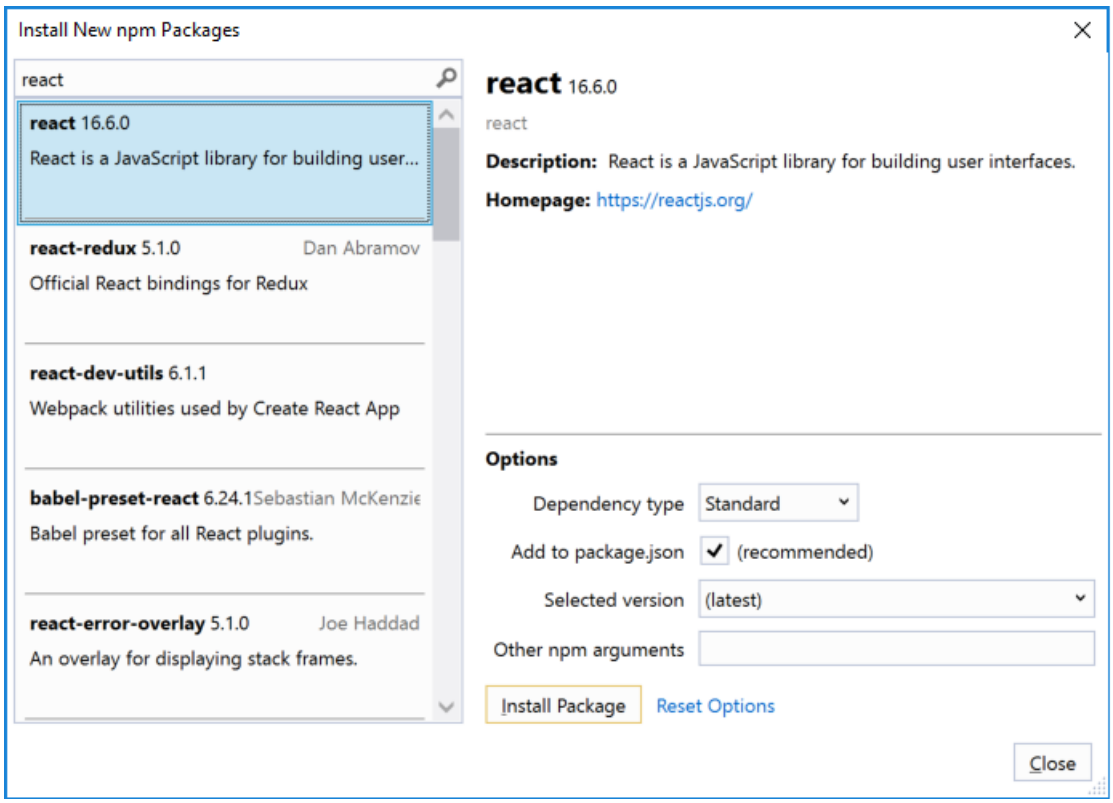
This app requires a number of npm modules to run correctly.

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

1. In Solution Explorer (right pane), right-click the **npm** node in the project and choose **Install New npm Packages**.

   In the **Install New npm Packages** dialog box, you can choose to install the most current package version or specify a version. If you choose to install the current version of these packages, but run into unexpected errors later, you may want to install the exact package versions described later in these steps.

2. In the **Install New npm Packages** dialog box, search for the react package, and select **Install Package** to install it.



   Select the **Output** window to see progress on installing the package (select **Npm** in the **Show output from** field). When installed, the package appears under the **npm** node.

   The project's *package.json* file is updated with the new package information including the package version.

3. Instead of using the UI to search for and add the rest of the packages one at a time, paste the following code into *package.json*. To do this, add a `dependencies` section with this code:

```
"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 there is already a `dependencies` section in your version of the blank template, just replace it with the preceding JSON code. For more information on use of this file, see package.json configuration.

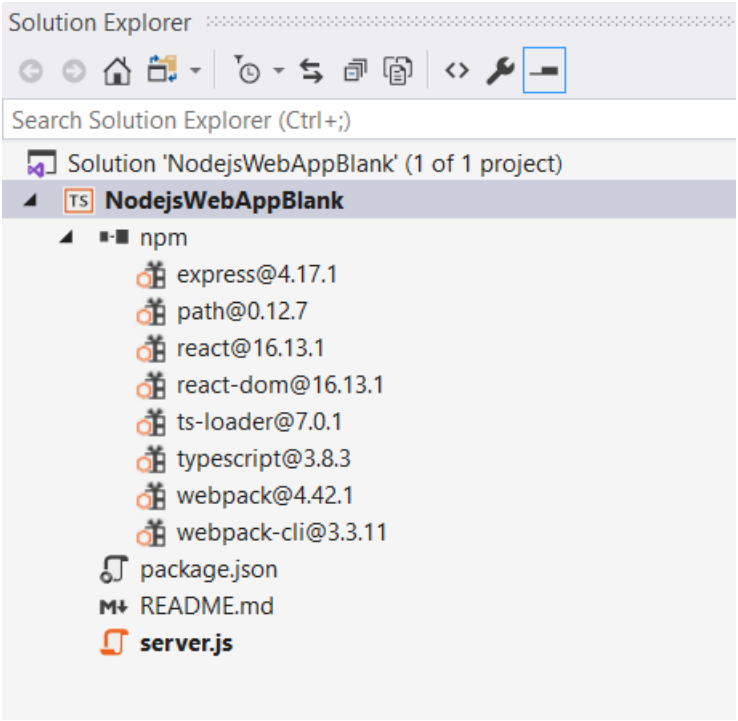
4. Save the changes.

5. Right-click **npm** node in your project and choose **Install npm Packages**.

   This command runs the npm install command directly.

   In the lower pane, select the **Output** window to see progress on installing the packages. Installation may take a few minutes and you may not see results immediately. To see the output, make sure that you select **Npm** in the **Show output from** field in the **Output** window.

   Here are the npm modules as they appear in Solution Explorer after they are installed.



> **NOTE**
>
> If you prefer to install npm packages using the command line, right-click the project node and choose **Open Command Prompt Here**. Use standard Node.js commands to install packages.

Add project files

In these steps, you 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. (In most apps, you typically add the files to subfolders and adjust relative path references accordingly.)

1. In Solution Explorer, right-click the project **NodejsWebAppBlank** and choose **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 *webpack-config.js*. Instead of a TypeScript JSX file, choose **JavaScript file**.
4. Repeat the same steps to add *index.html* to the project. Instead of a JavaScript file, choose **HTML file**.
5. Repeat the same steps to add *tsconfig.json* to the project. Instead of a JavaScript file, choose **TypeScript JSON Configuration file**.

Add app code

1. 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. This code sets the port to the port number configured in the project properties (by default, the port is configured to 1337 in the properties). To open the project properties, right-click the project in Solution Explorer and choose **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 simple 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

In the previous steps, you added *webpack-config.js* to the project. Next, you add webpack configuration code. You will 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 code is a basic configuration that is intended as 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 default code 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"
  ]
}
```

   app.tsx is specified as the source file.

Transpile the JSX

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

2. In the command prompt, type the following 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>
```

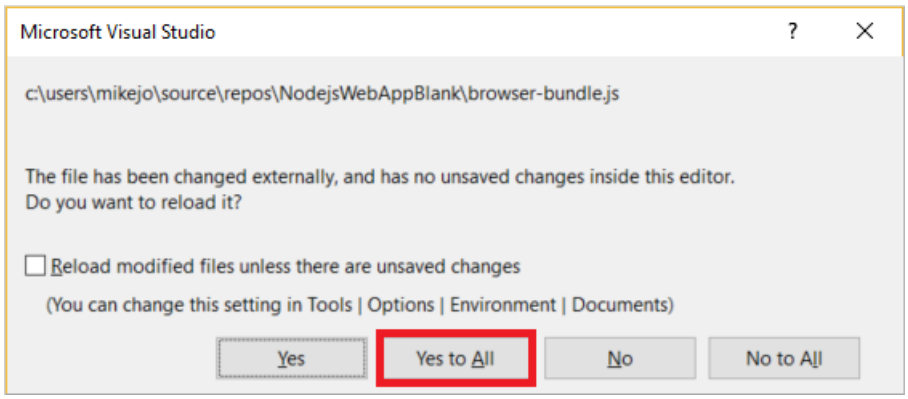
If you see any errors instead of the preceding output, you must resolve them before your app will work. If your npm package versions are different than the versions shown in this tutorial, that can be a source of errors. One way to fix errors is to use the exact versions shown in the earlier steps. Also, if one or more of these 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.

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

   Visual Studio adds the *dist* folder to the project, which contains *app-bundle.js* and *app-bundle.js.map*.

4. Open *app-bundle.js* to see the transpiled JavaScript code.

5. If prompted to reload externally modified files, select **Yes to All**.



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

Add a build script to transpile the JSX

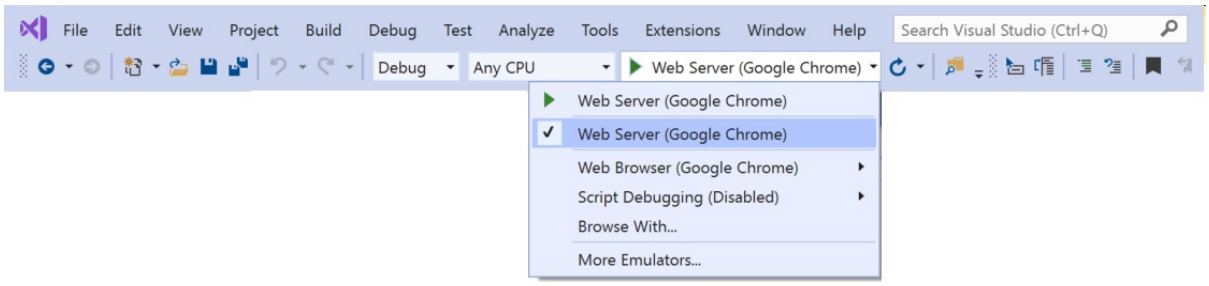
Starting in Visual Studio 2019, a build script is required. Instead of transpiling JSX at the command line (as shown in the preceding section), you can transpile JSX when building from Visual Studio.

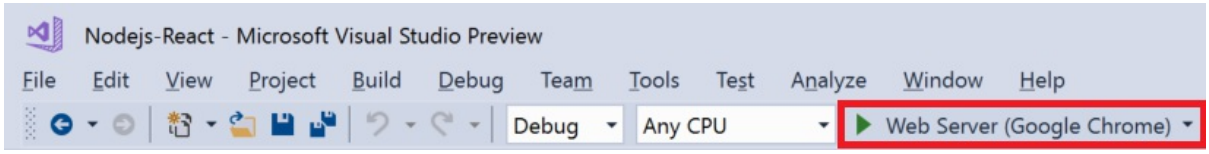
- Open *package.json* and add the following section after the `dependencies` section:

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

Run the app

1. Select either **Web Server (Google Chrome)** or **Web Server (Microsoft Edge)** as the current debug target.



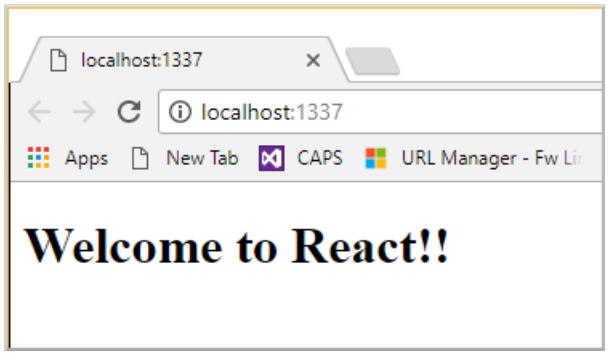


If Chrome is available on your machine, but does not show up as an option, choose **Browse With** from the debug target dropdown list, and select Chrome as the default browser target (choose **Set as Default**).

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

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

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



3. Close the browser window.

4. Close the console window.

Set a breakpoint and run the app

1. In *server.js*, click in the gutter to the left of the `staticPath` declaration 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 the 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 for Microsoft Edge, press **F12**. You can use these tools to examine the DOM and interact with the app using the JavaScript Console.
5. Close the web browser and the console.

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

Prepare the browser for debugging

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

For this scenario, use Chrome.

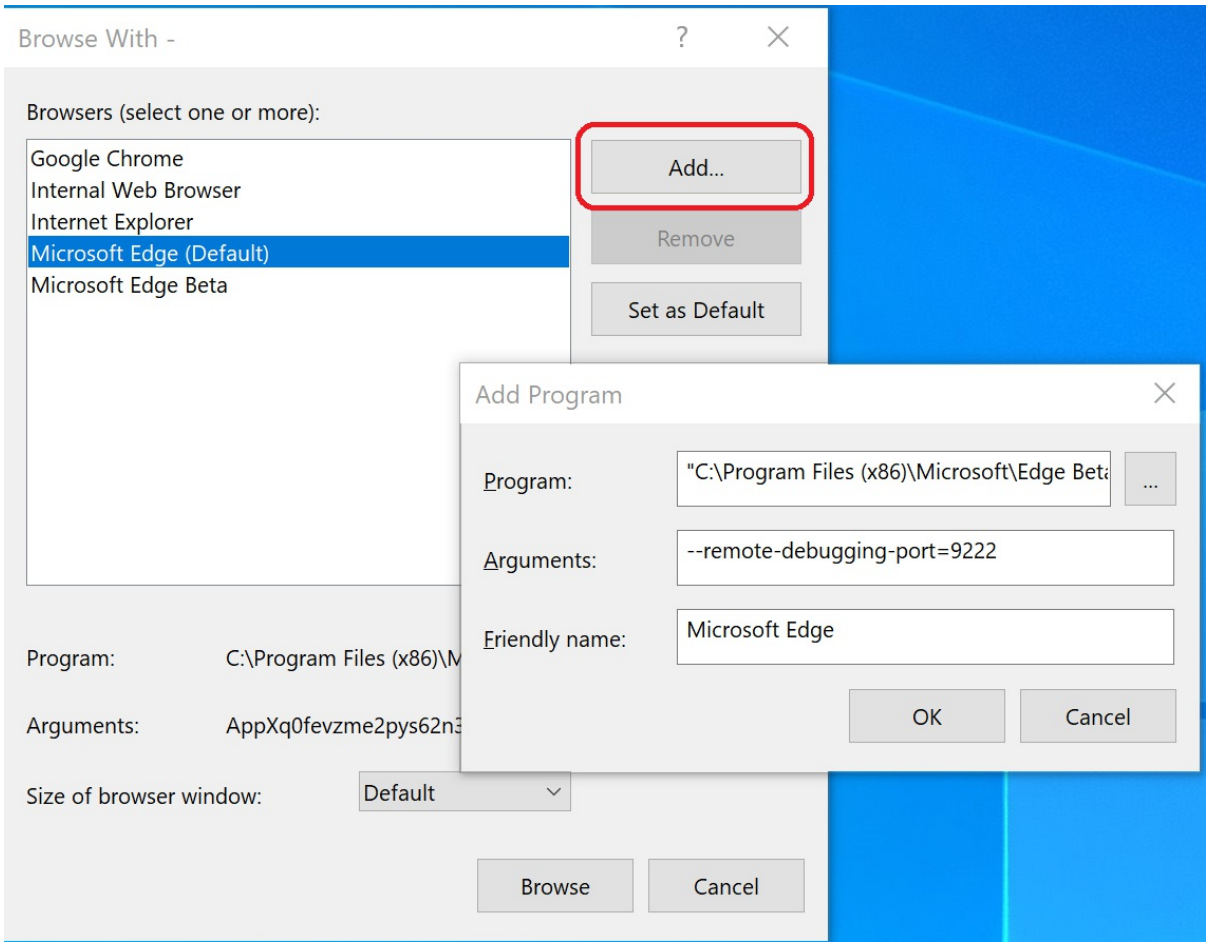
1. Close all windows for the target browser.

   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 share 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
```

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

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

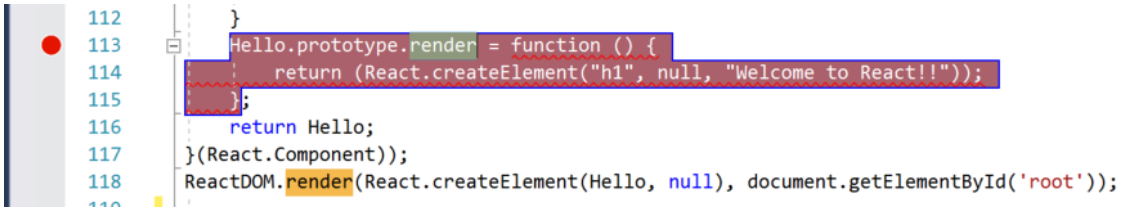
This starts your browser with debugging enabled.

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

Attach the debugger to client-side script

1. Switch to Visual Studio and then set a breakpoint in your source code, either *app-bundle.js* or *app.tsx*.

   For *app-bundle.js*, set the breakpoint in the `render()` function as shown in the following illustration:



   To find the `render()` function in the transpiled *app-bundle.js* file, use **Ctrl+F** (**Edit** > **Find and Replace** > **Quick Find**).

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

2. If you are setting the breakpoint in the *.tsx* file (rather than *app-bundle.js*), you need to update *webpack-config.js*. Replace the following code:

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

with this code:

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

This is a development-only setting to enable debugging in Visual Studio. This setting allows you to override the generated references in the source map file, *app-bundle.js.map*, when building the app. By default, webpack references in the source map file include the *webpack:///* prefix, which prevents Visual Studio from finding the source file, *app.tsx*. Specifically, when you make this change, the reference to the source file, *app.tsx*, gets changed from *webpack:///./app.tsx* to *./app.tsx*, which enables debugging.

3. 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.

4. 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**.

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

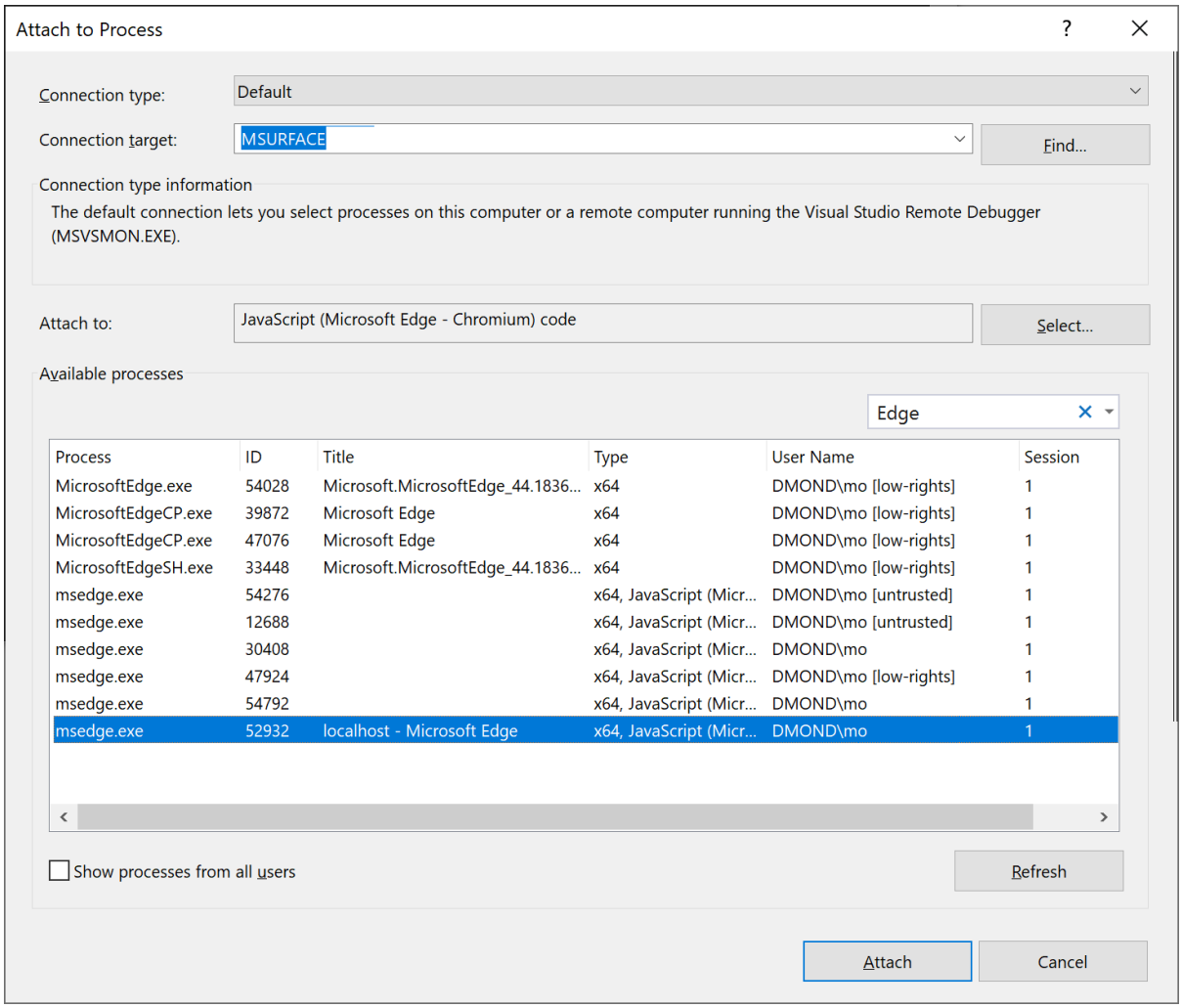
   In Visual Studio 2019, choose the correct debugger for your target browser, **JavaScript (Chrome)** or **JavaScript (Microsoft Edge - Chromium)** in the **Attach to** field, type **chrome** or **edge** in the filter box to filter the search results.

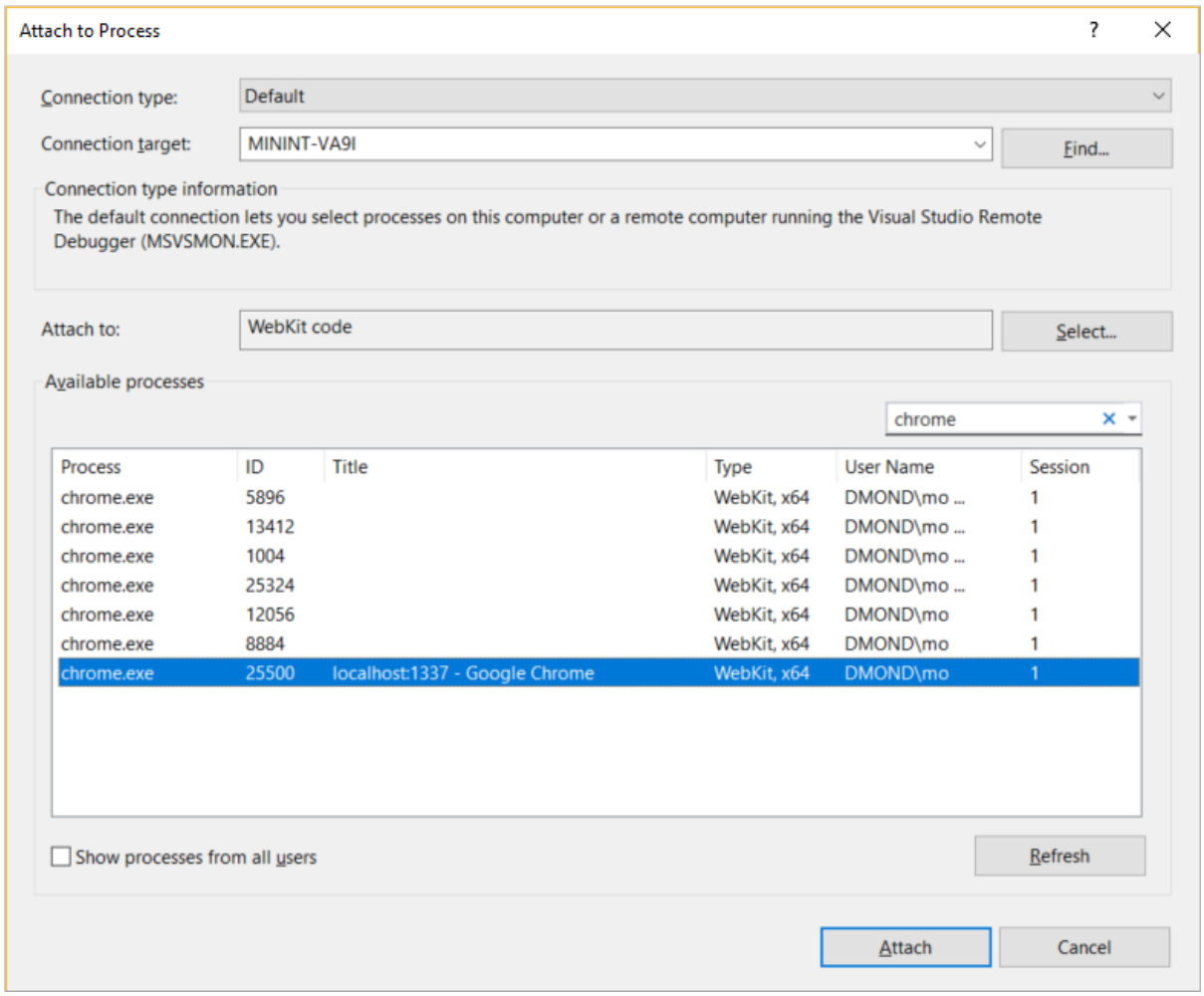
   In Visual Studio 2017, choose **Webkit code** in the **Attach to** field, type **chrome** in the filter box to filter the search results.

6. Select the browser process with the correct host port (localhost in this example), and select **Attach**.

   The port (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.





You know the debugger has attached correctly when the DOM Explorer and the JavaScript Console open in Visual Studio. These debugging tools are similar to Chrome Developer Tools and F12 Tools for Microsoft Edge.

> **TIP**
> If the debugger does not attach and you see the message "Unable to attach to the process. An operation is not legal in the current state.", use the 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.

   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 *app-bundle.js* or its mapped location in *app.tsx*, depending on which steps you followed previously, along with your environment and browser state. Either way, you can step through code and examine variables.

   - If you need to break into code in *app.tsx* 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 a reference to *./app.tsx* and not *webpack:///./app.tsx*, which prevents the Visual Studio debugger from locating *app.tsx*. Alternatively, if you need to break into code in *app.tsx* and are unable to do it, 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 need to break into code in *app-bundle.js* and are unable to do it, 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 TypeScript in Visual Studio

5/4/2020 • 6 minutes to read • Edit Online

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.

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

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

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 2019, go to the Visual Studio downloads page to install it for free.

  If you haven't already installed Visual Studio 2017, 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.

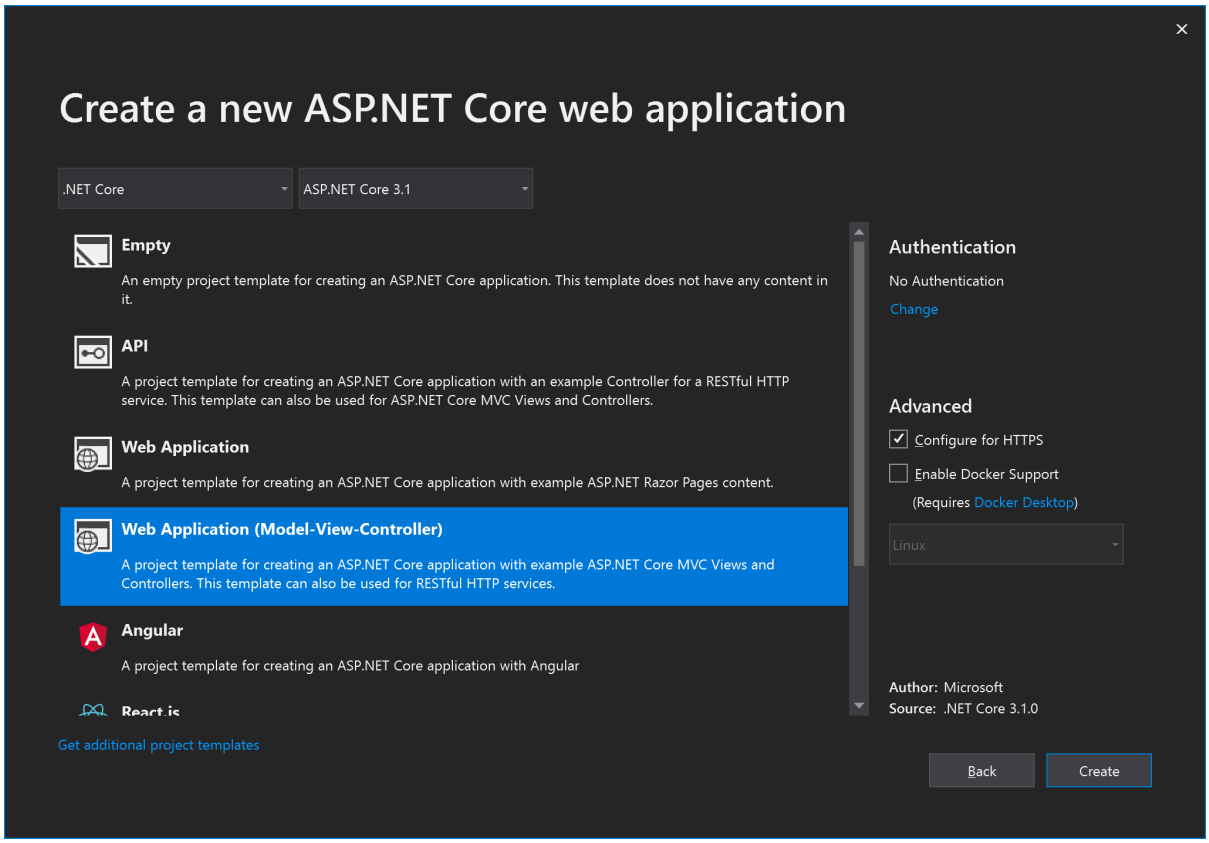
   If the start window is not open, choose **File** > **Start Window**. On the start window, choose **Create a new project**. In the language drop-down list, choose **C#**. In the search box, type **ASP.NET**, then choose **ASP.NET Core Web Application**. Choose **Next**.

Type a name for the project and choose **Create**.

From the top menu bar, choose **File** > **New** > **Project**. In the left pane of the **New Project** dialog box, expand **Visual C#**, then choose **.NET Core**. In the middle pane, choose **ASP.NET Core Web Application - C#**, then choose **OK**.

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

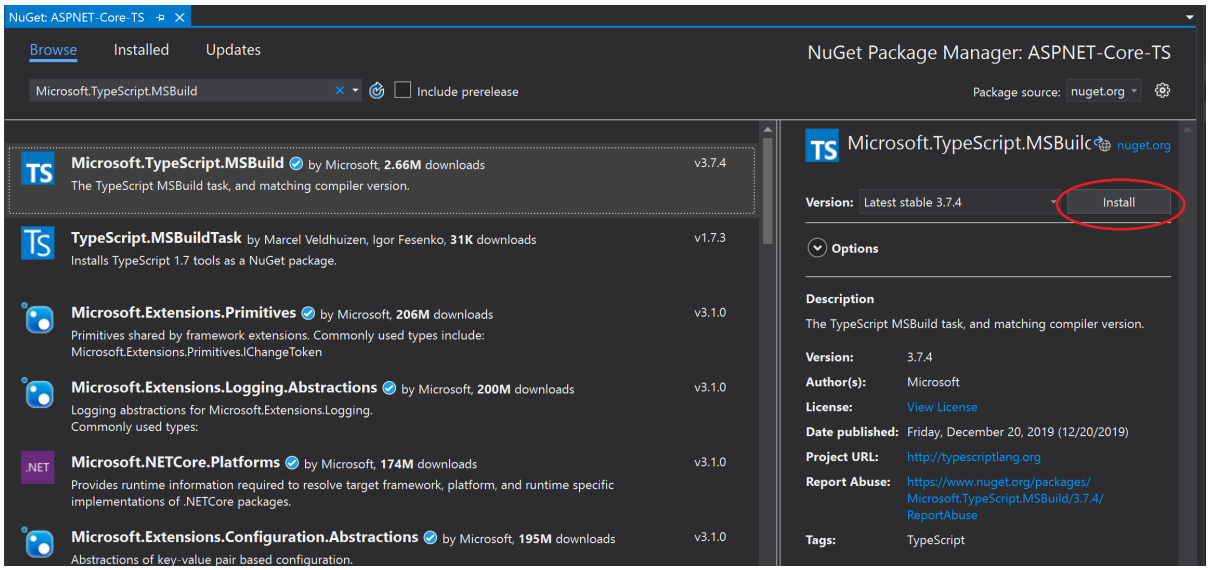
3. In the dialog box that appears, select **Web Application (Model-View-Controller)** in the dialog box, and then choose **Create** (or **OK**).



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.

> **NOTE**
>
> This tutorial requires the NuGet package. Alternatively, in your own apps, you may want to use the TypeScript npm package.

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:

```
{
  "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.html*.

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.

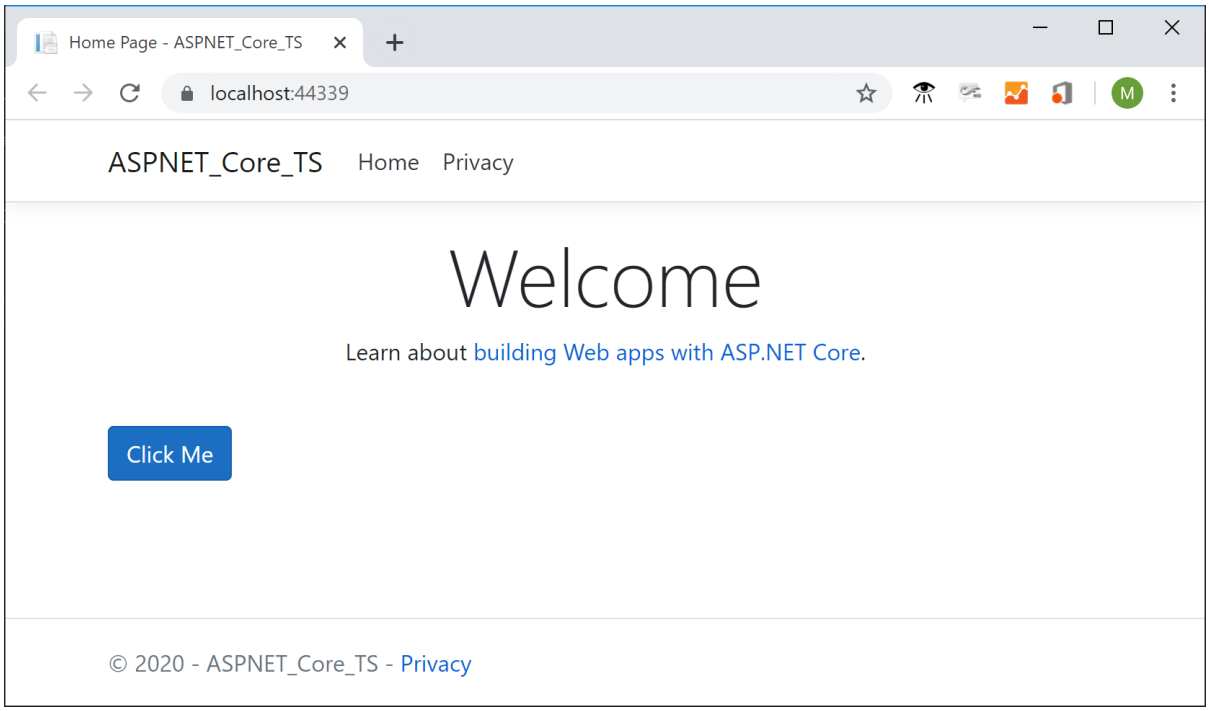
   Source map files are required for debugging.

Run the application

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

   The app opens in a browser.

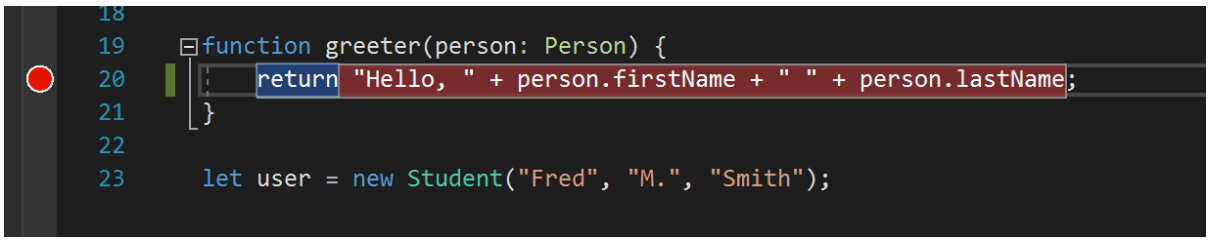
   In the browser window, you will see the **Welcome** heading and the **Click Me** button.



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());
      $("#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.

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

jqtest.showMsg();
```

```
★ innerHTML
★ appendChild
★ querySelector
★ style
★ firstChild
accessKey
accessKeyLabel
addEventListener
```

(property) InnerHTML.innerHTML: string

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

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

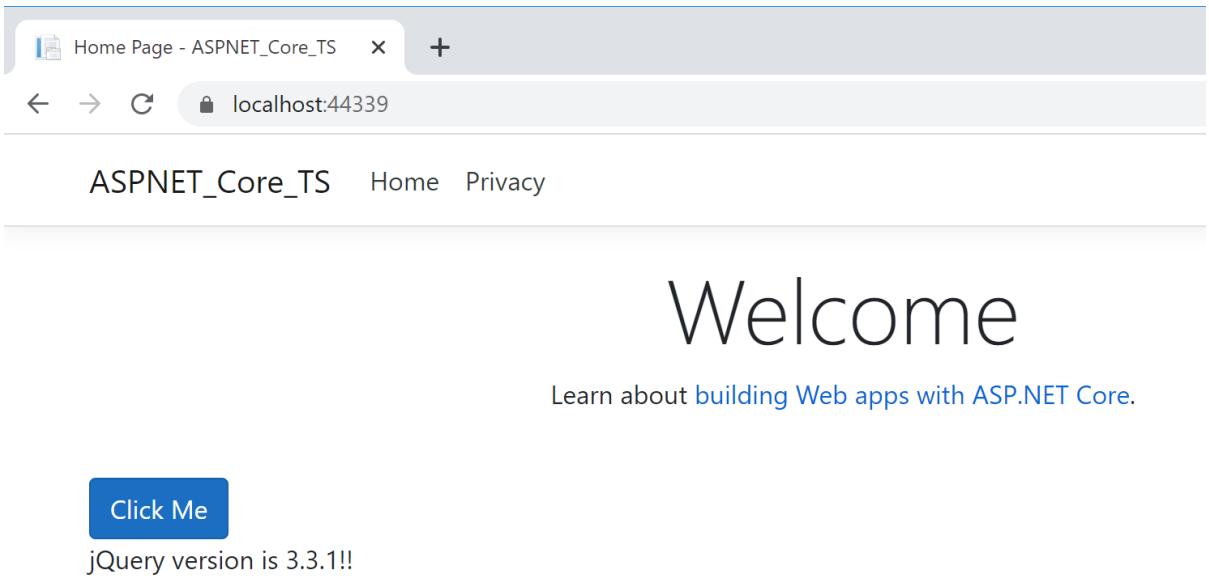
8. In Index.cshtml, add the following HTML to the end of the file.

```
<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.

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

4/16/2020 • 6 minutes to read • Edit Online

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

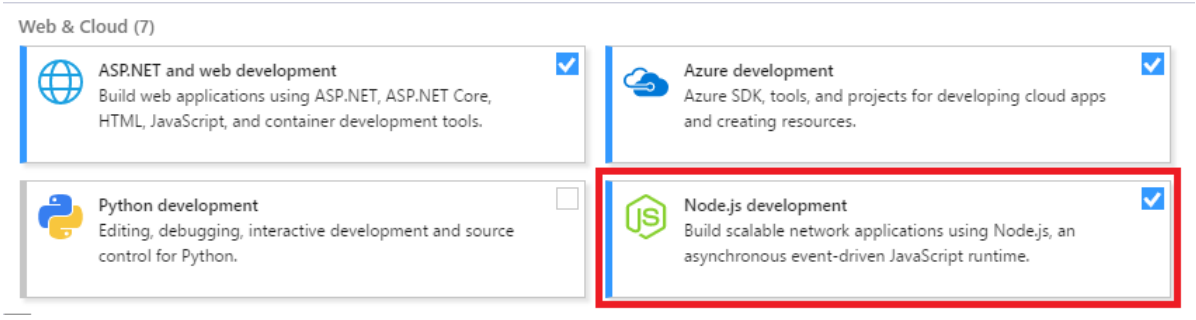
Prerequisites

- 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 haven't already installed Visual Studio 2017, 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**).

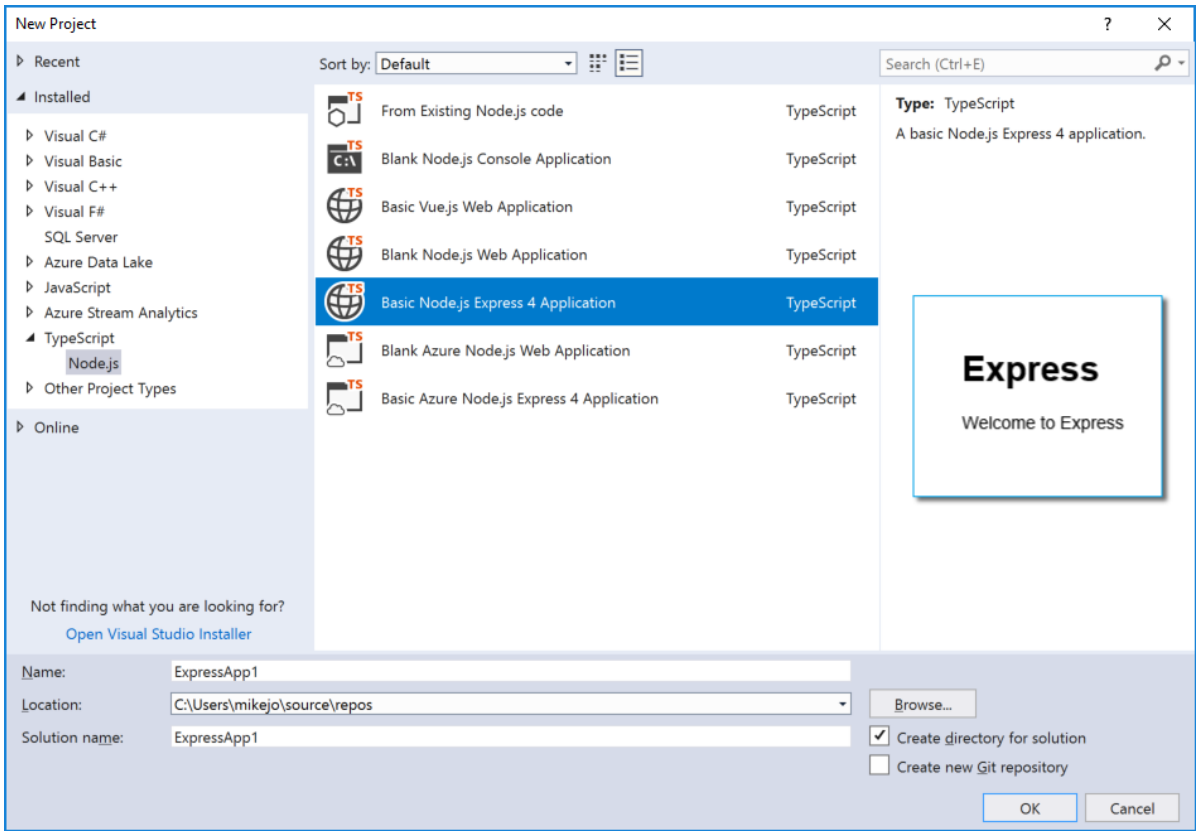
Create a Node.js project to run in Azure

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**.

   From the top menu bar, choose **File** > **New** > **Project**. In the left pane of the **New Project** dialog box, expand **TypeScript**, then choose **Node.js**. In the middle pane, choose **Basic Azure Node.js Express 4 application**, then choose **OK**.



   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

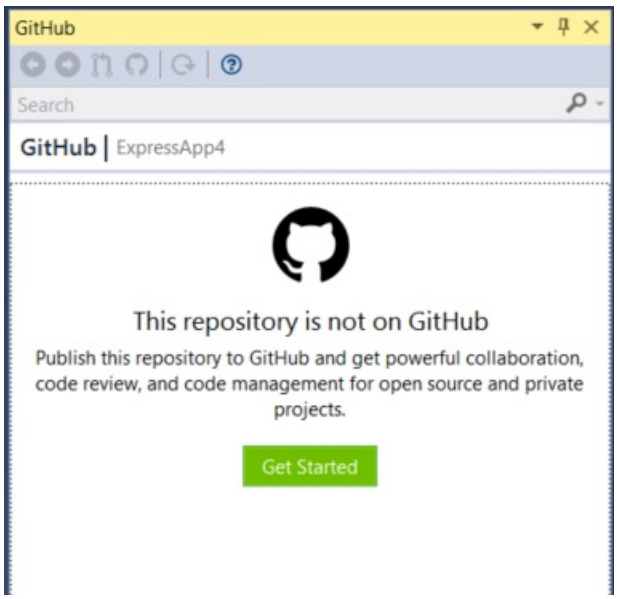
To set up GitHub for Visual Studio:

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**.

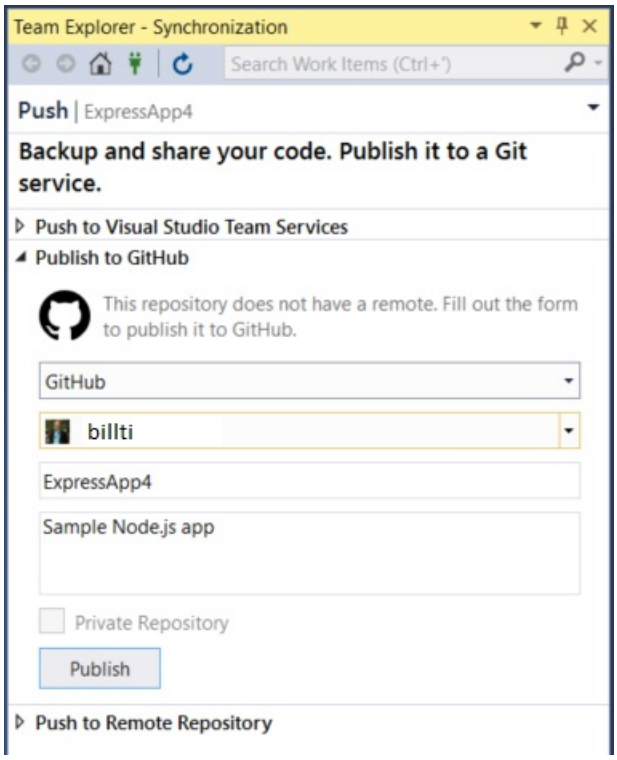
   The GitHub window opens.

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.



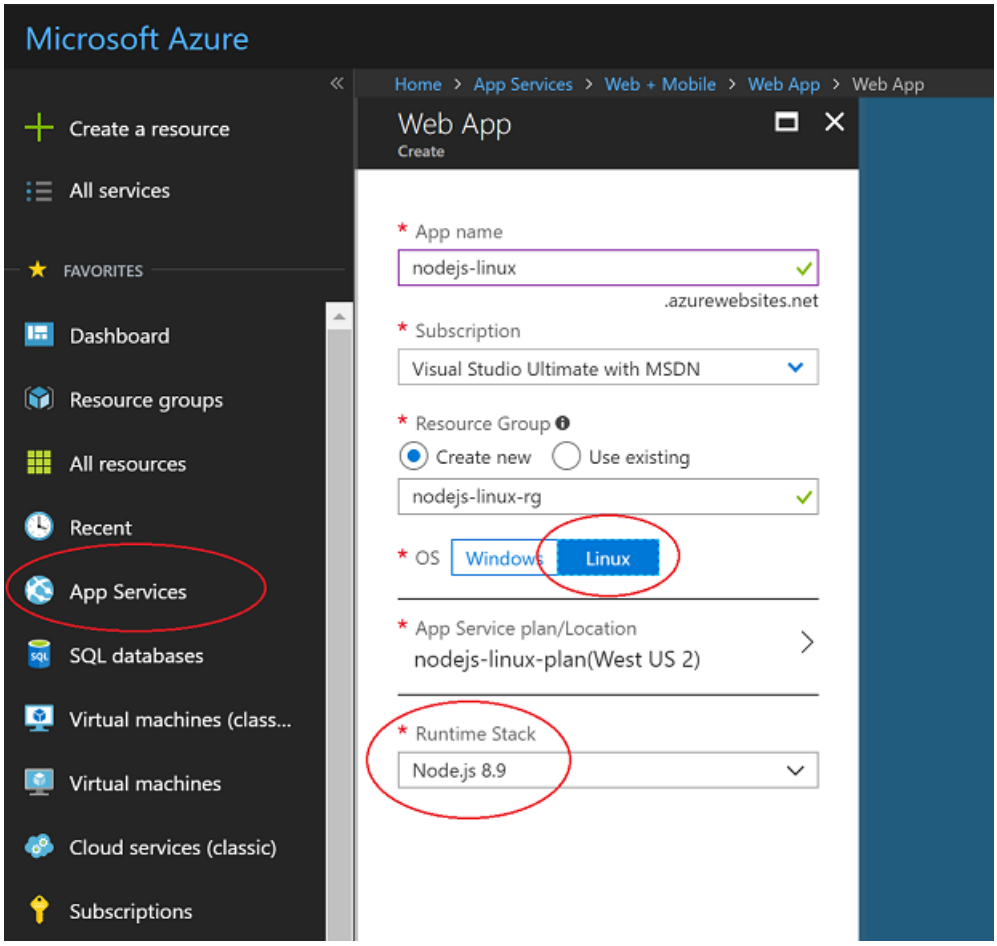
5. Complete the fields for the new repository to publish, and then click **Publish**.

   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

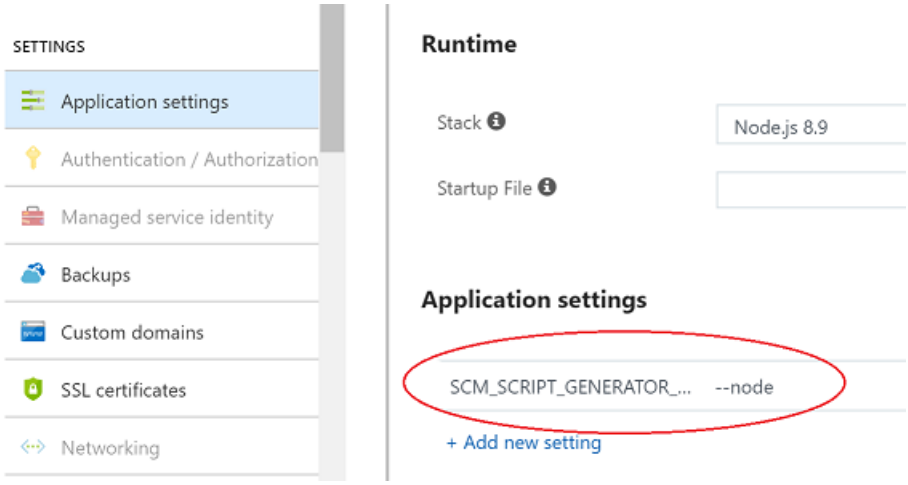
1. Sign in to the Azure portal.
2. Select **App Services** from the list of services on the left, and then click **Add**.
3. If required, create a new Resource Group and App Service plan to host the new app.
4. Make sure to set the **OS** to **Linux**, and set **Runtime Stack** to the required Node.js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



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

   It may take a few minutes to deploy.

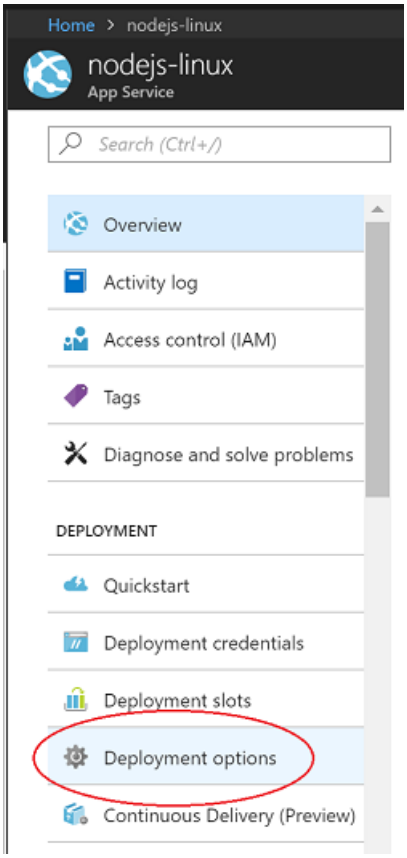
6. After it is deployed, go to the **Application settings** section, and add a setting with a name of `SCM_SCRIPT_GENERATOR_ARGS` and a value of `--node`.



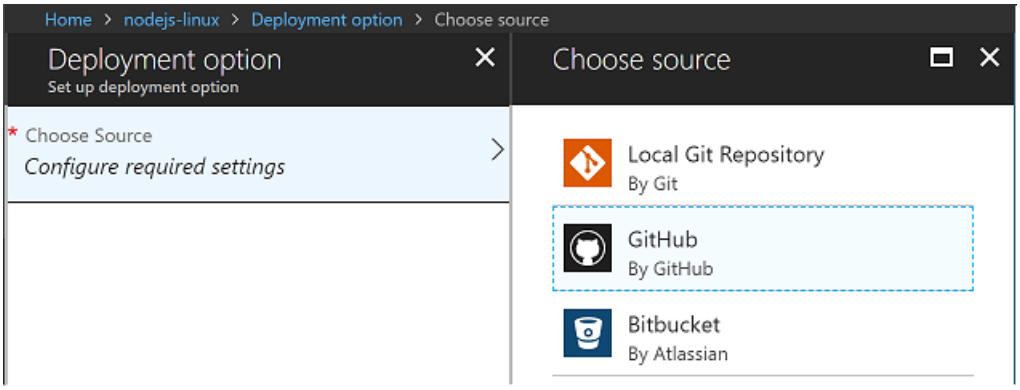
> **WARNING**
>
> The App Service deployment process uses a set of heuristics to determine which type of application to try and run. If a *.sln* file is detected in the deployed content, it will assume an MSBuild based project is being deployed. The setting added above overrides this logic and specifies 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`.

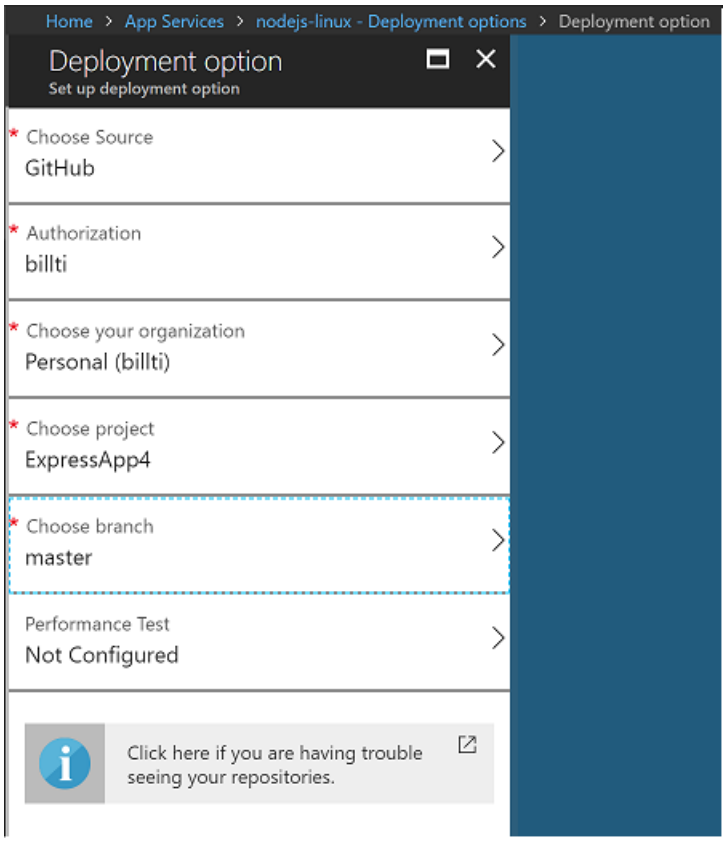
8. After it is deployed, open the App Service and select **Deployment options**.



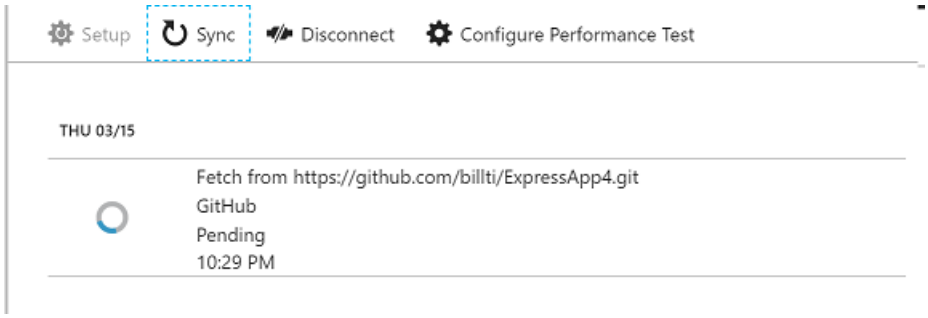
9. Click **Choose source**, and then choose **GitHub**, and then configure any required permissions.



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").

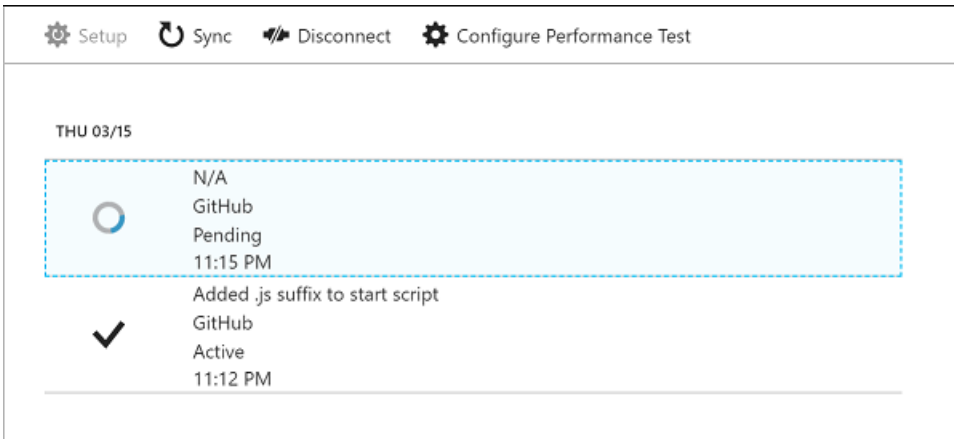
Modify your app and push changes

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

4/2/2019 • 4 minutes to read • Edit Online

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.

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

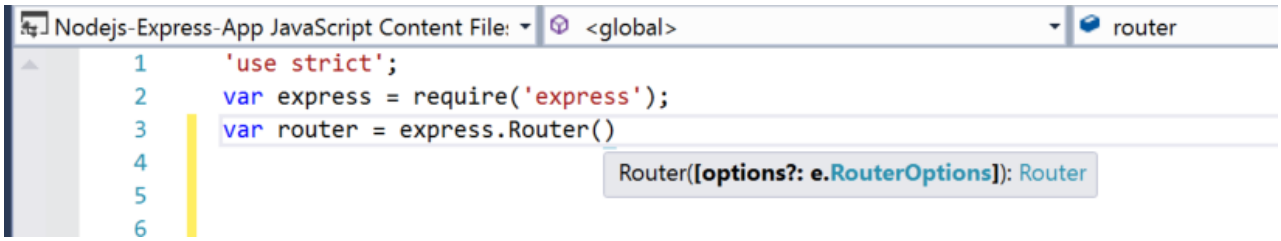
You can use the IDE to add new files to your project.

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**.

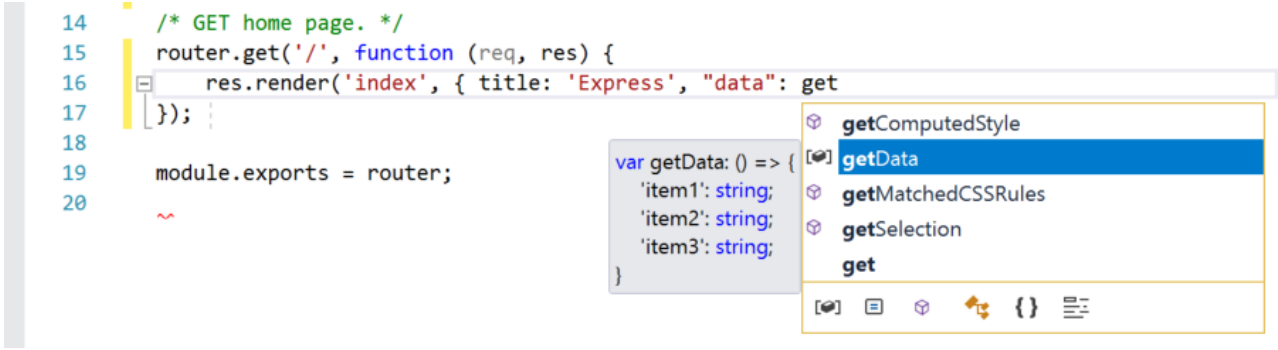
   The new file gets added to your project and it opens in the editor.

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.



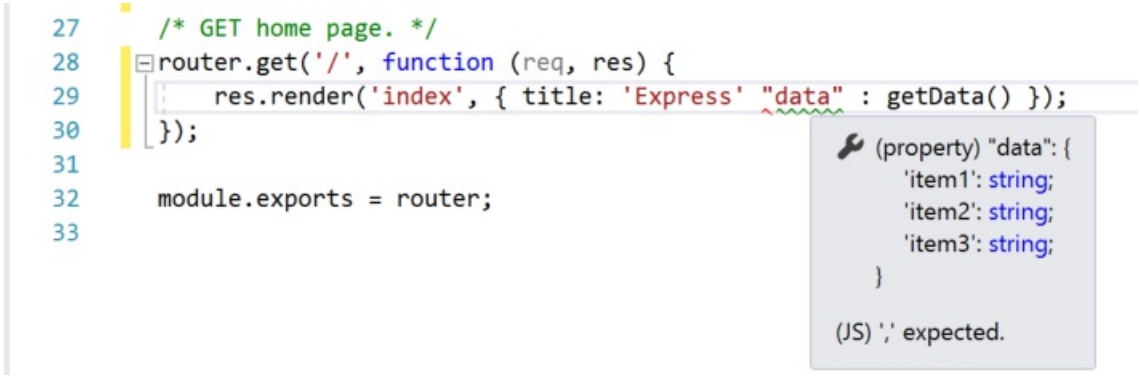
IntelliSense can also show you information about types when you hover over programming elements.

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

Check syntax

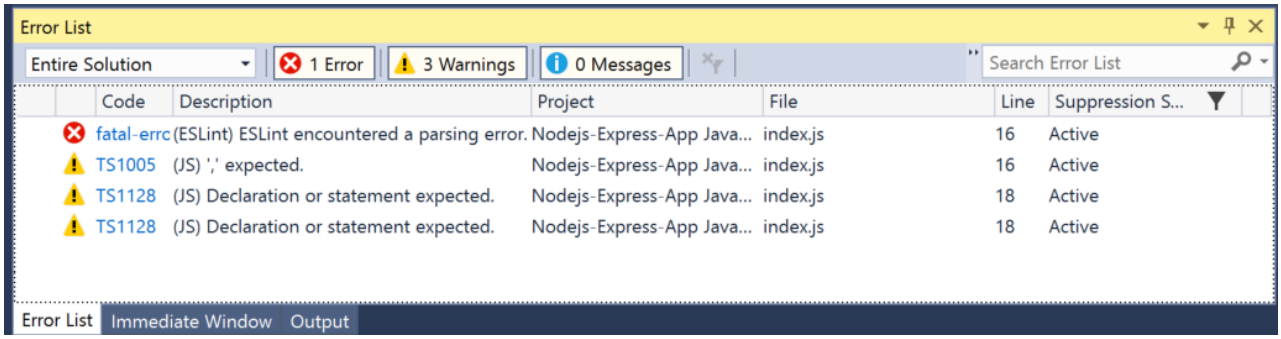
The language service uses ESLint to provide syntax checking and linting. If you need to set options for syntax checking in the editor, select **Tools** > **Options** > **JavaScript/TypeScript** > **Linting**. The linting options point you to the global ESLint configuration file.

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"`.

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**.

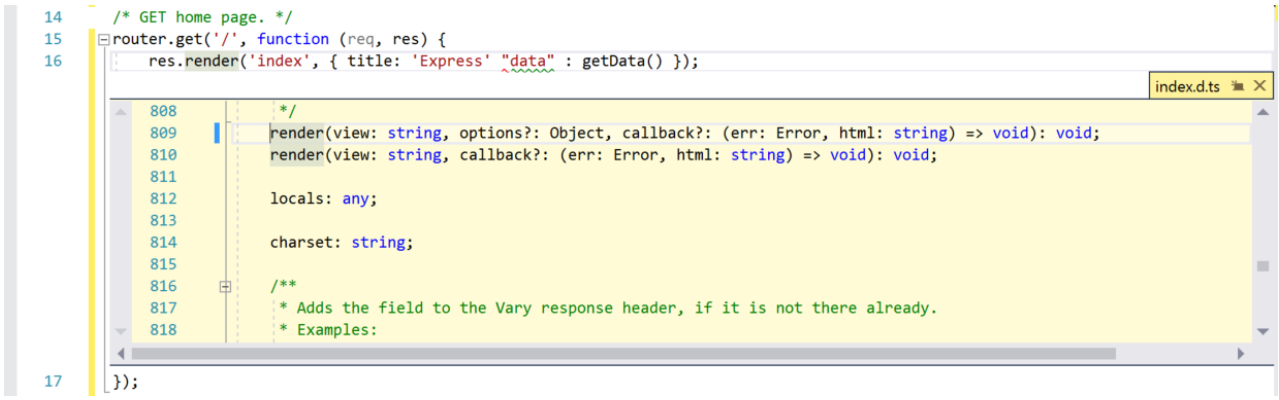
The code block collapses to just the first line, followed by an ellipsis (`...`). To expand the code block again, click the same gray box that now has a plus sign in it, or press **Ctrl**+**M**, **Ctrl**+**M** again. This feature is called Outlining and is especially useful when you're collapsing long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



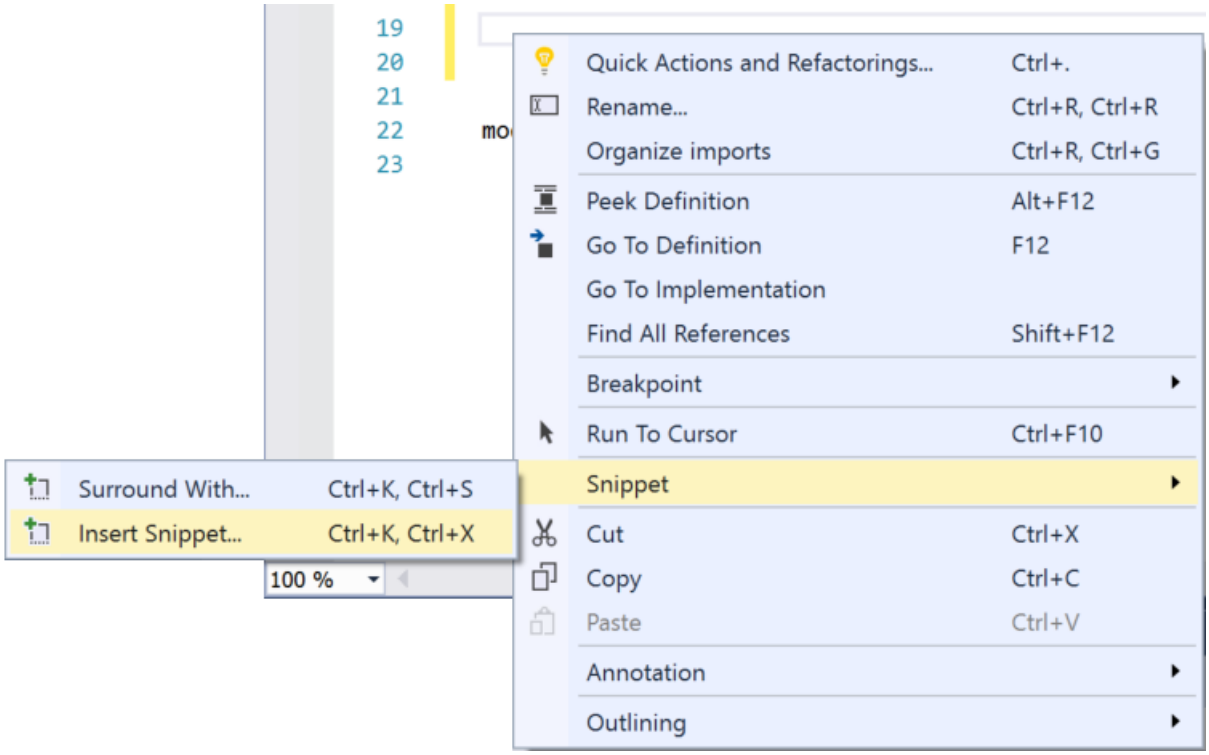
Close the peeked definition window by choosing the small box with an "x" at the top right of the pop-up window.

Use code snippets

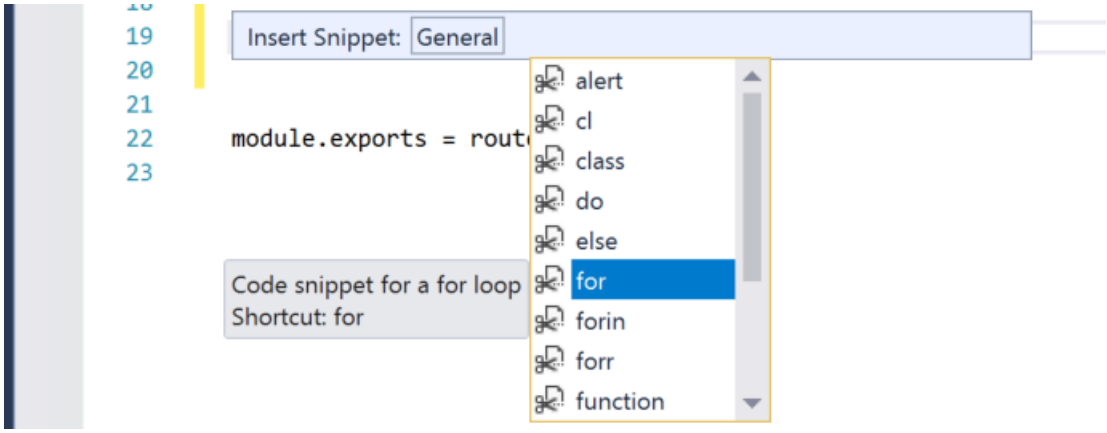
Visual Studio provides useful *code snippets* that you can use to quickly and easily generate commonly used code blocks. Code snippets are available for different programming languages including JavaScript. Let's add a `for`

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

Create a Vue.js application using Node.js Tools for Visual Studio

4/21/2020 • 5 minutes to read • Edit Online

Visual Studio supports app development with the Vue.js framework in either JavaScript or TypeScript.

The following new features support Vue.js application development in Visual Studio:

- Support for Script, Style, and Template blocks in *.vue* files
- Recognition of the `lang` attribute on *.vue* files
- Vue.js project and file templates

Prerequisites

- You must have Visual Studio 2017 version 15.8 or a later version installed and the **Node.js development** workload.

  > **IMPORTANT**
  > This article requires features that are only available starting in Visual Studio 2017 version 15.8.

  If a required version is not already installed, install Visual Studio 2019.

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

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

- To create the ASP.NET Core project, you must have the ASP.NET and web development and .NET Core cross-platform development workloads installed.

- 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 Vue.js project using Node.js

You can use the new Vue.js templates to create a new project. Use of the template is the easiest way to get started. For detailed steps, see Use Visual Studio to create your first Vue.js app.

Create a Vue.js project with ASP.NET Core and the Vue CLI

Vue.js provides an official CLI for quickly scaffolding projects. If you would like to use the CLI to create your application, follow the steps in this article to set up your development environment.

> **IMPORTANT**
>
> These steps assume that you already have some experience with the Vue.js framework. If not, please visit Vue.js to learn more about the framework.

Create a new ASP.NET Core project

For this example, you use an empty ASP.NET Core Application (C#). However, you can choose from a variety of projects and programming languages.

Create an Empty project

1. Open Visual Studio and create a new project.

   Press **Esc** to close the start window. Type **Ctrl + Q** to open the search box, type **asp.net**, then choose **Create a new ASP.NET Core Web Application**. In the dialog box that appears, type the name **client-app**, and then choose **Create**.

   From the top menu bar, choose **File** > **New** > **Project**. In the left pane of the **New Project** dialog box, expand **Visual C#**, then choose **Web**. In the middle pane, choose **ASP.NET Core Web Application**, type the name **client-app**, and then choose **OK**.

   If you don't see the **ASP.NET Core Web Application** project template, you must install the **ASP.NET and web development** workload and the **.NET Core** development workload first. To install the workload(s), click the **Open Visual Studio Installer** link in the left pane of the **New Project** dialog box (select **File** > **New** > **Project**). The Visual Studio Installer launches. Select the required workloads.

2. Select **Empty**, and then click **OK**.

   Visual Studio creates the project, which opens in Solution Explorer (right pane).

Configure the project startup file

- Open the file *./Startup.cs*, and add the following lines to the Configure method:

```
app.UseDefaultFiles(); // Enables default file mapping on the web root.
app.UseStaticFiles(); // Marks files on the web root as servable.
```

Install the vue CLI

To install the vue-cli npm module, open a command prompt and type `npm install --g vue-cli` or `npm install -g @vue/cli` for version 3.0 (currently in beta).

Scaffold a new client application using the vue CLI

1. Go to your command prompt and change the current directory to your project root folder.
2. Type `vue init webpack client-app` and follow steps when prompted to answer additional questions.

> **NOTE**
>
> For *.vue* files, you need to use WebPack or a similar framework with a loader to do the conversion. TypeScript and Visual Studio does not know how to compile *.vue* files. The same is true for bundling; TypeScript doesn't know how to convert ES2015 modules (that is, `import` and `export` statements) into a single final *.js* file to load in the browser. Again, WebPack is the best choice here. To drive this process from within Visual Studio using MSBuild, you need to do start from a Visual Studio template. At present, there is no ASP.NET template for Vue.js development in-the-box.

Modify the webpack configuration to output the built files to wwwroot

- Open the file *./client-app/config/index.js*, and change the `build.index` and `build.assetsRoot` to wwwroot path:

```
// Template for index.html
index: path.resolve(__dirname, '../../wwwroot/index.html'),

// Paths
assetsRoot: path.resolve(__dirname, '../../wwwroot'),
```

Indicate the project to build the client app each time that a build is triggered

1. In Visual Studio, go to **Project** > **Properties** > **Build Events**.
2. On **Pre-build event command line**, type `npm --prefix ./client-app run build`.

Configure webpack's output module names

- Open the file *./client-app/build/webpack.base.conf.js*, and add the following properties to the output property:

```
devtoolModuleFilenameTemplate: '[absolute-resource-path]',
devtoolFallbackModuleFilenameTemplate: '[absolute-resource-path]?[hash]'
```

Add TypeScript support with the Vue CLI

These steps require vue-cli 3.0, which is currently in beta.

1. Go to your command prompt and change the current directory to the project root folder.
2. Type `vue create client-app`, and then choose **Manually select features**.
3. Choose **Typescript**, and then select other desired options.
4. Follow the remaining steps and respond to the questions.

Configure a Vue.js project for TypeScript

1. Open the file *./client-app/tsconfig.json* and add `noEmit:true` to the compiler options.

   By setting this option, you avoid cluttering your project each time that you build in Visual Studio.

2. Next, create a *vue.config.js* file in *./client-app/* and add the following code.

```
module.exports = {
    outputDir: '../wwwroot',

    configureWebpack: {
        output: {
            devtoolModuleFilenameTemplate: '[absolute-resource-path]',
            devtoolFallbackModuleFilenameTemplate: '[absolute-resource-path]?[hash]'
        }
    }
};
```

   The preceding code configures webpack and sets the wwwroot folder.

Build with vue-cli 3.0

An unknown issue with the vue-cli 3.0 may prevent automating the build process. Each time that you try to refresh the wwwroot folder, you need to run the command `npm run build` on the client-app folder.

Alternatively, you can build the vue-cli 3.0 project as a pre-build event using the ASP.NET project properties. Right-click the project, choose **Properties**, and include the following commands in the **Build** tab, in the **Pre-build event command line** text box.

```
cd ./client-app
npm run build
cd ../
```

Limitations

- `lang` attribute only supports JavaScript and TypeScript languages. The accepted values are: js, jsx, ts, and tsx.
- `lang` attribute doesn't work with template or style tags.
- Debugging script blocks in *.vue* files isn't supported due to its preprocessed nature.
- TypeScript doesn't recognize *.vue* files as modules. You need a file that contains code such as the following to tell TypeScript what *.vue* files look like (vue-cli 3.0 template already includes this file).

  ```
  // ./client-app/vue-shims.d.ts
  declare module "*.vue" {
      import Vue from "vue";
      export default Vue;
  }
  ```

- Running the command `npm run build` as a pre-build event on the project properties doesn't work when using vue-cli 3.0.

See also

- Vue get started guide.
- Vue CLI project.
- Webpack configuration documentation.

Manage npm packages in Visual Studio

4/21/2020 • 6 minutes to read • Edit Online

npm allows you to install and manage packages for use in your Node.js 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.

- 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.

Node.js projects

For Node.js projects, 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.

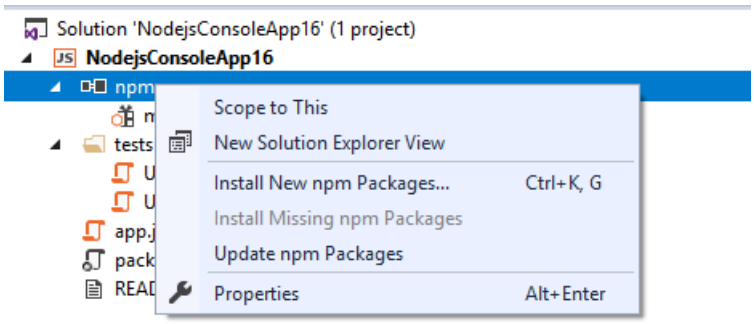
Prerequisites

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 project.

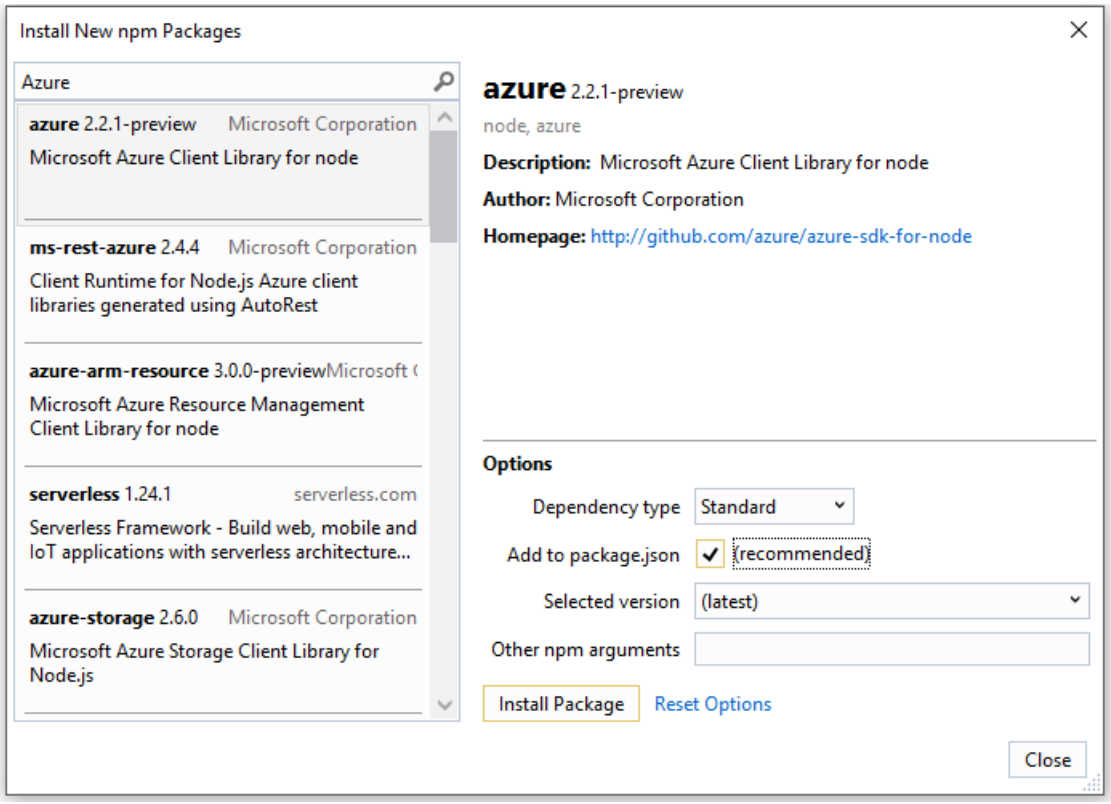
> **NOTE**
> For existing Node.js projects, use the **From existing Node.js code** solution template or the Open folder (Node.js) project type to enable npm in your project.

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**.

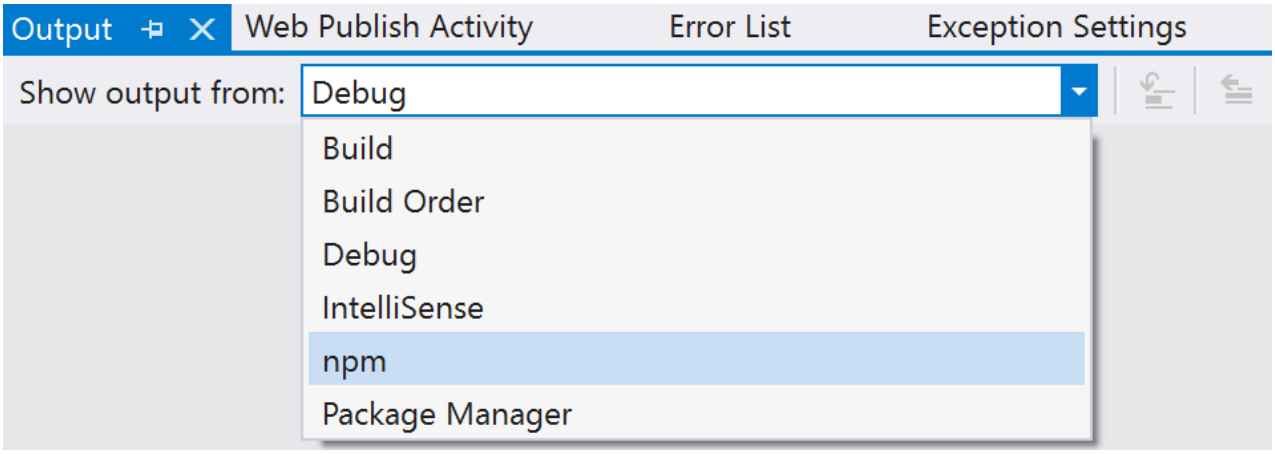


In this window you can search for a package, specify options, and install.



- **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. This may take some time.

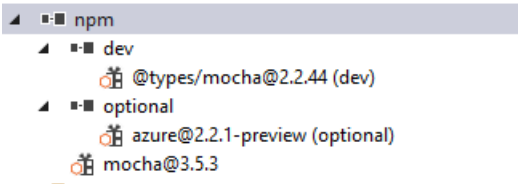


TIP

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 lastest versions, according to the 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 lastest 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`.)

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

- **Install missing packages** that are listed in *package.json*

- **Update npm packages** to the latest version
- **Uninstall a package** and remove from *package.json*

> **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**.

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.

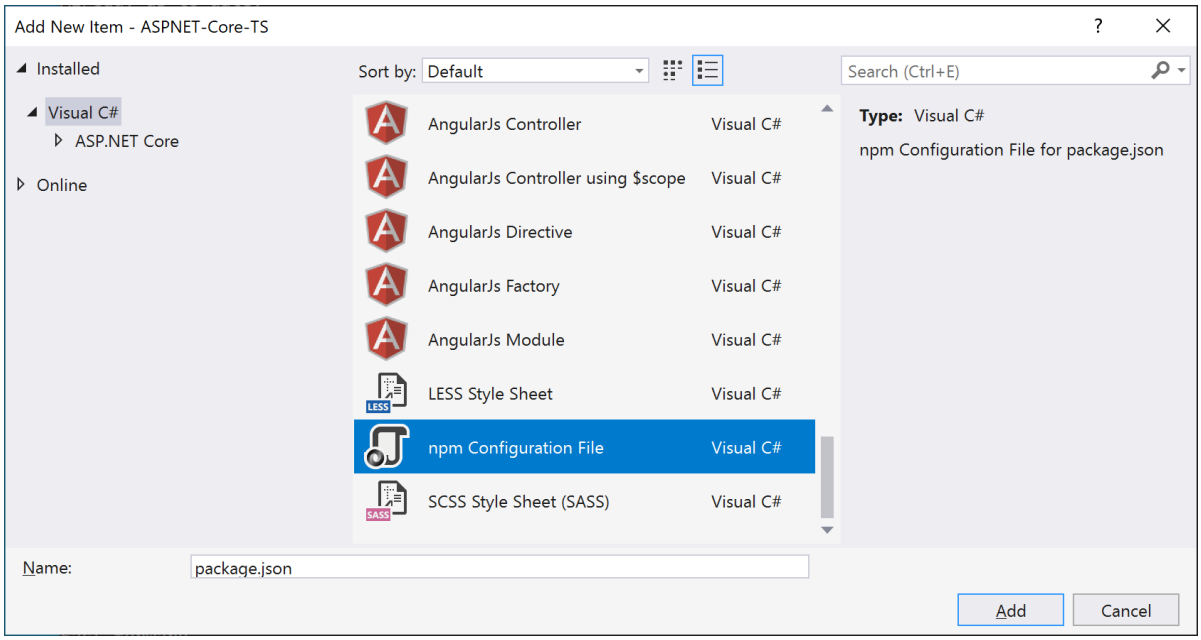
Add npm support to a project (ASP.NET Core)

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**. 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:

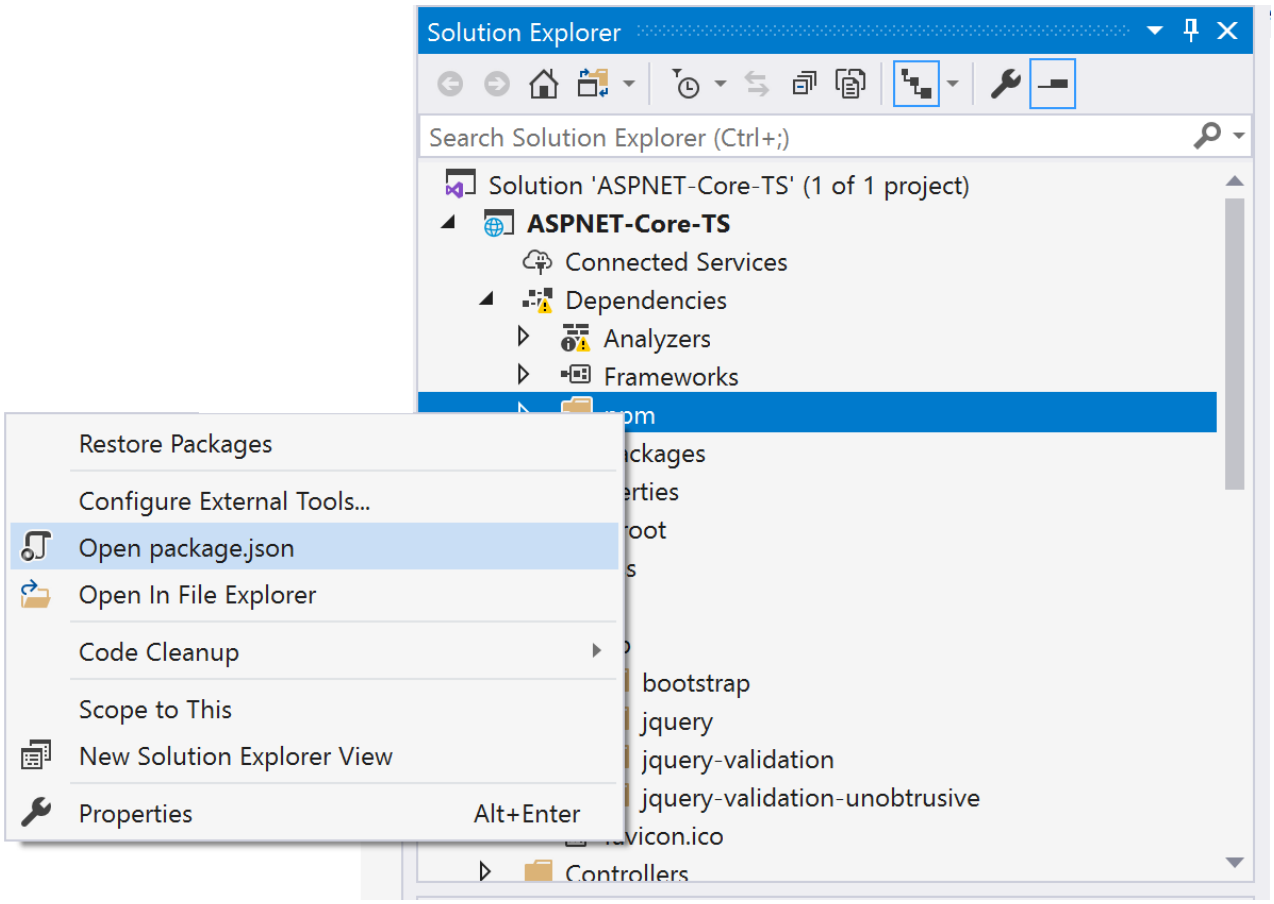
```
"devDependencies": {
   "gulp": "4.0.2",
   "@types/jquery": "3.3.33"
}
```

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**.

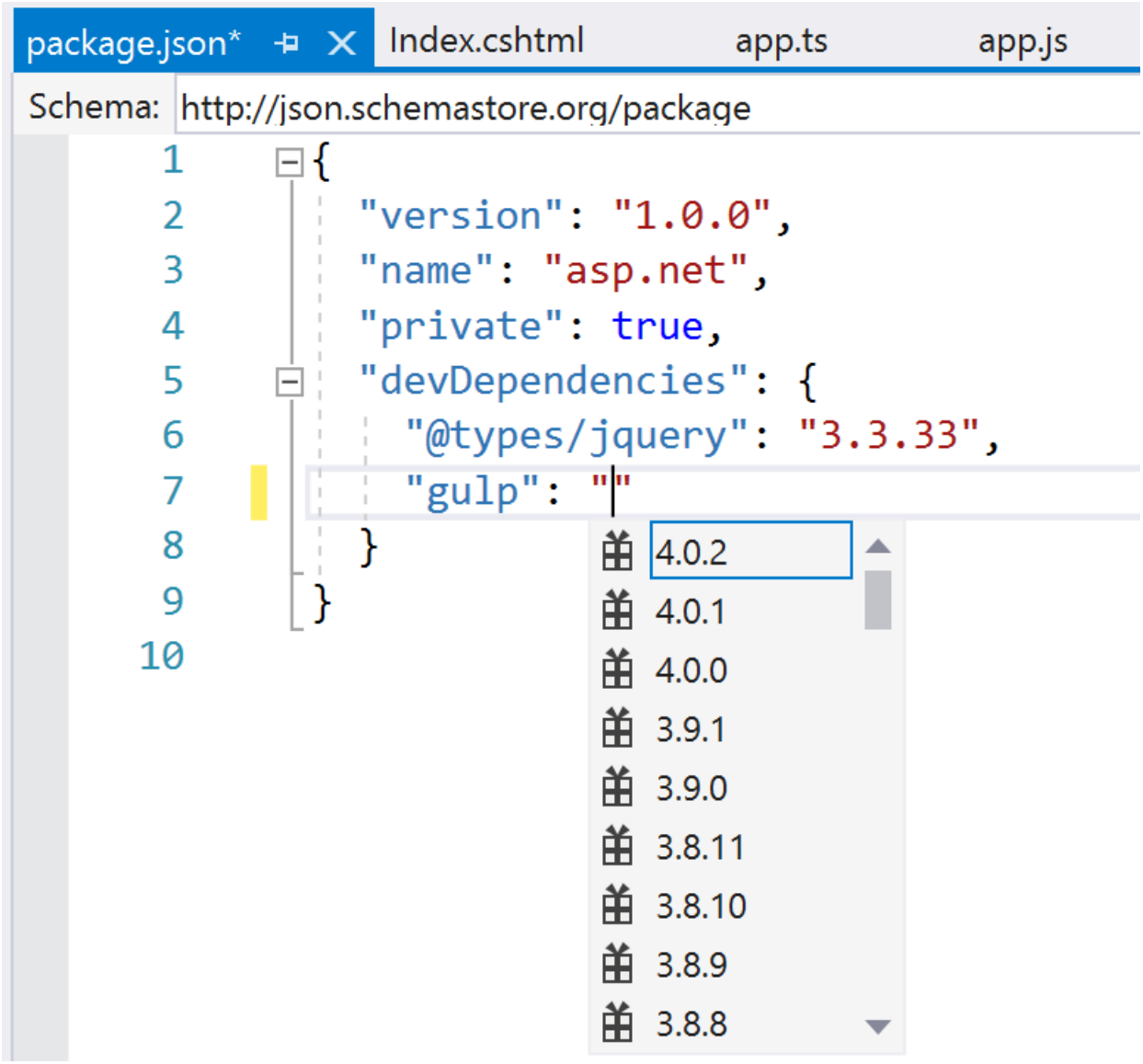
> **NOTE**
> In some scenarios, Solution Explorer may not show the correct status for installed npm packages. For more information, see Troubleshooting.

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

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

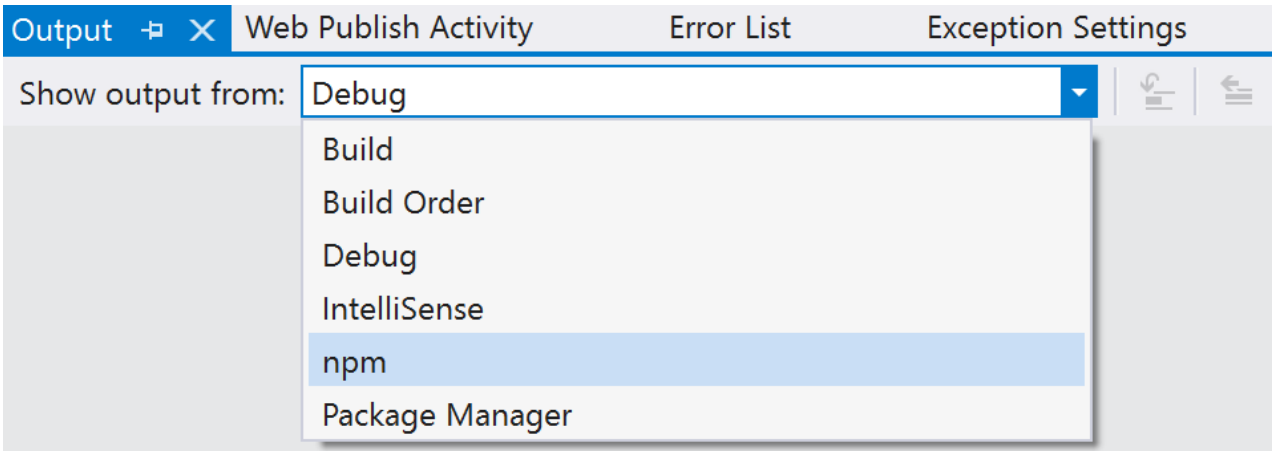


IntelliSense in *package.json* helps you select a particular version of an npm package.



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.
- 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

3/19/2020 • 2 minutes to read • Edit Online

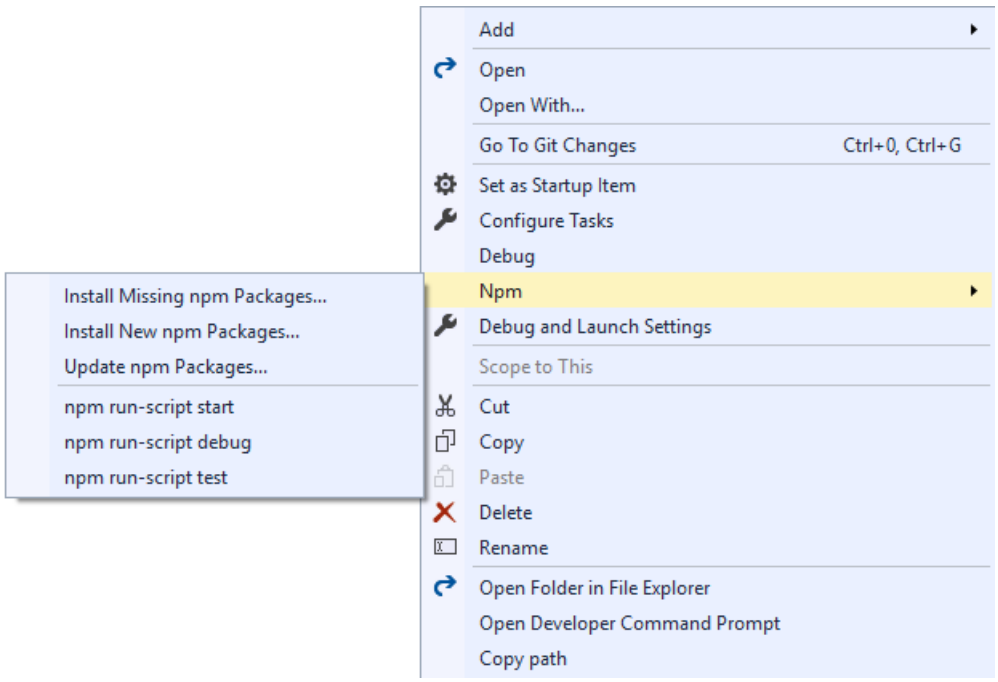
Starting in Visual Studio 2017, you can develop code without projects or solutions, which enables you to open a folder of code and immediately start working with rich editor support such as IntelliSense, search, refactoring, debugging, and more. In addition to these features, the Node.js Tools for Visual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

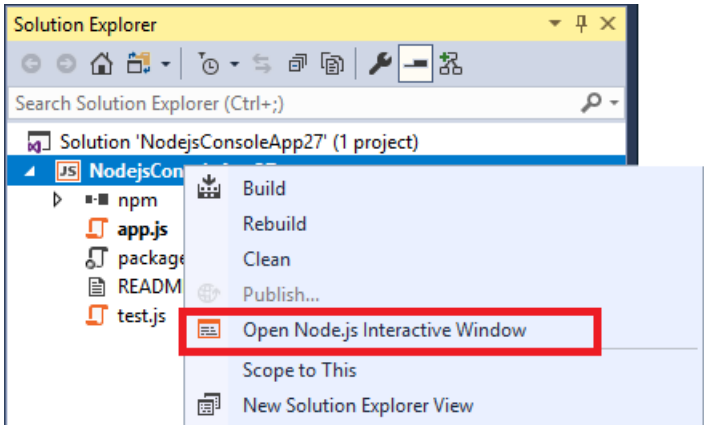
Work with the Node.js interactive window

1/25/2019 • 2 minutes to read • Edit Online

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

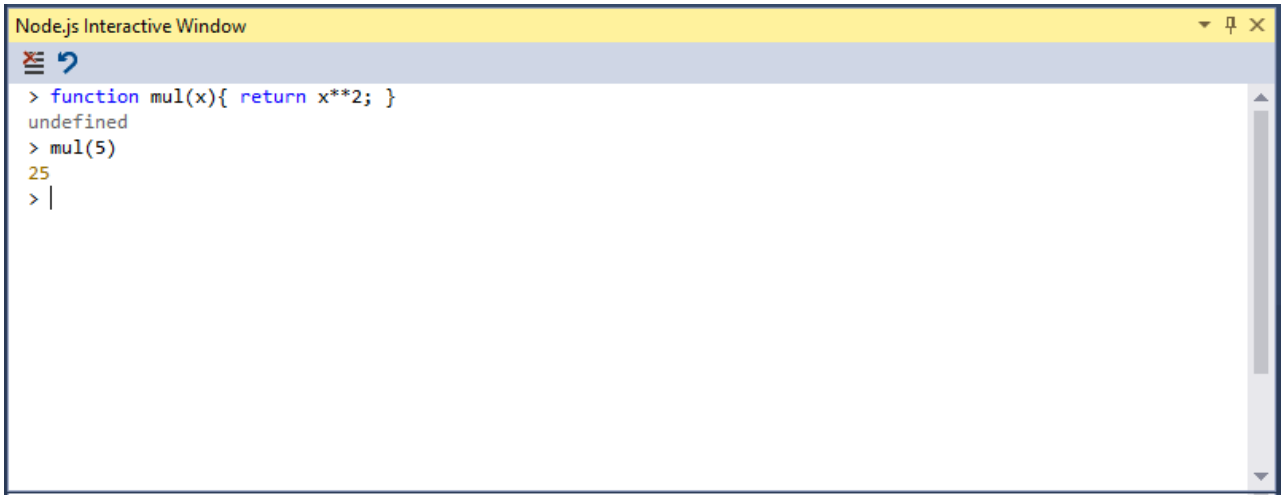
You can open the interactive window by right-clicking the Node.js project node in Solution Explorer and selecting **Open Node.js Interactive Window**.



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

1/6/2020 • 13 minutes to read • Edit Online

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.

Visual Studio provides client-side debugging support for Chrome and Internet Explorer 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 Client-side debugging of ASP.NET projects in

Google Chrome.

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.

For this scenario, use Chrome.

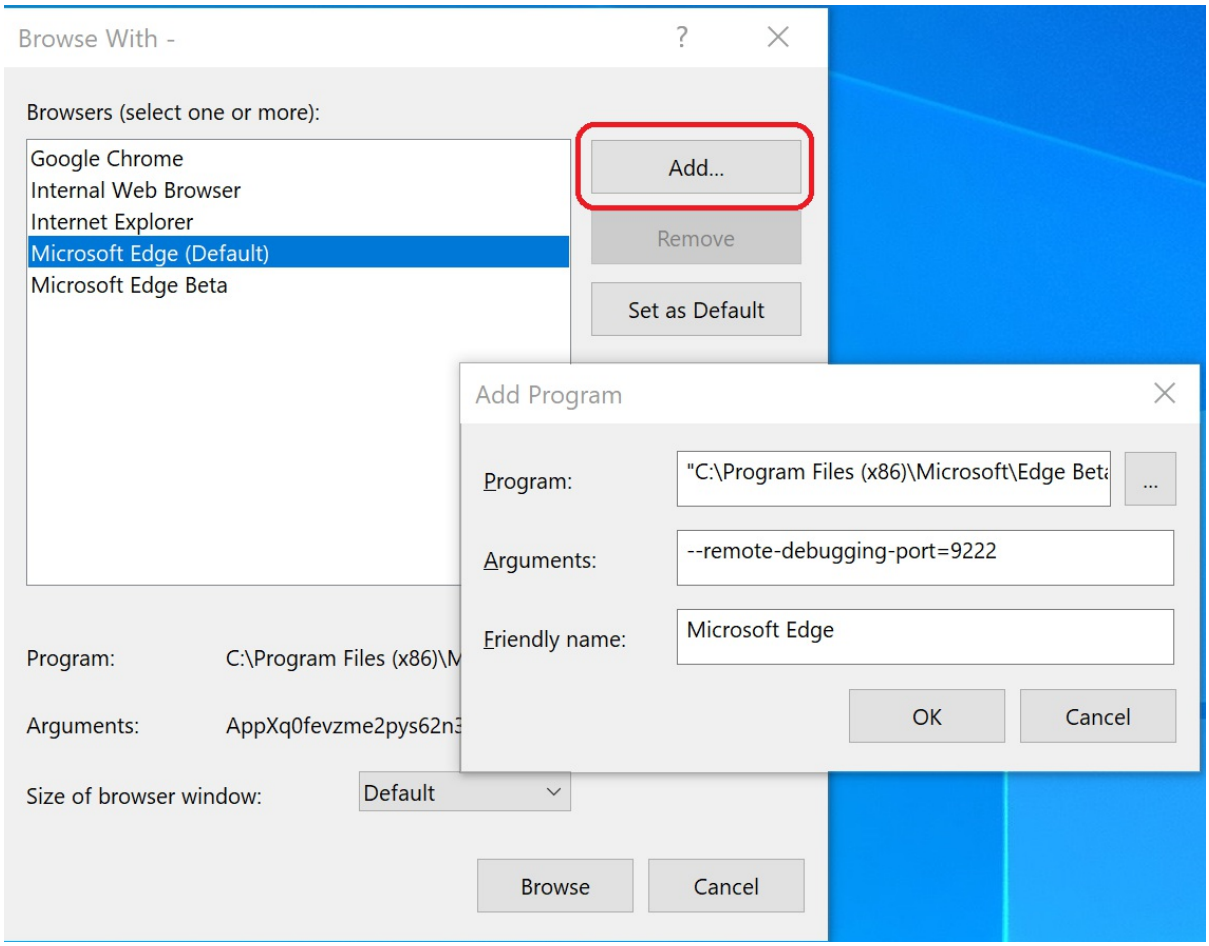
1. Close all windows for the target browser.

   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
```

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

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

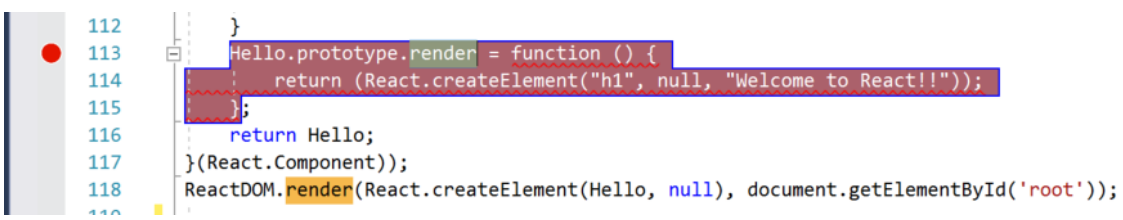
This starts your browser with debugging enabled.

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.

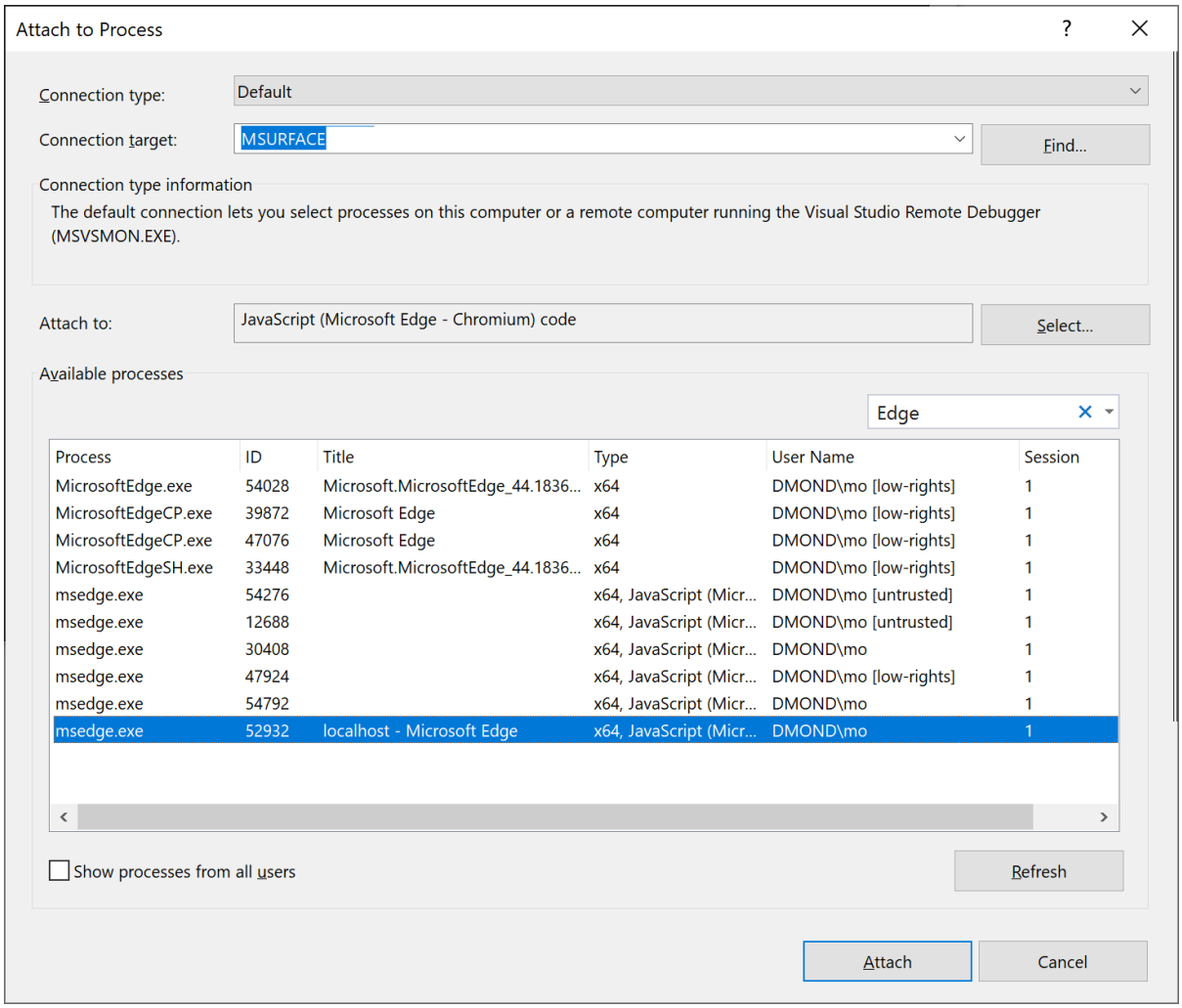
   In Visual Studio 2019, choose the correct debugger for your target browser, **JavaScript (Chrome)** or **JavaScript (Microsoft Edge - Chromium)** in the **Attach to** field, type **chrome** or **edge** in the filter box to filter the search results.

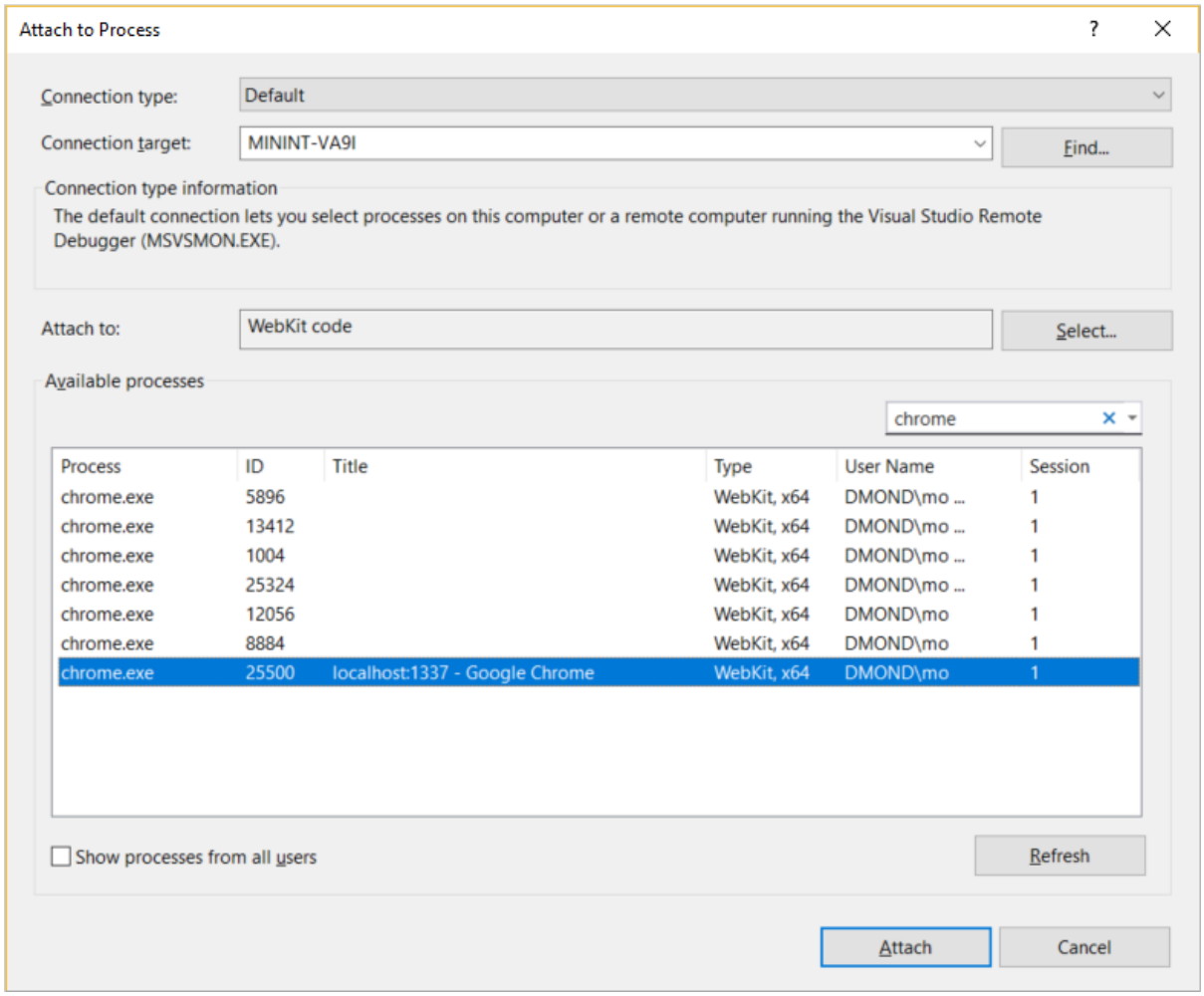
   In Visual Studio 2017, choose **Webkit code** in the **Attach to** field, type **chrome** in the filter box to filter the search results.

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.


Attach to Process
Connection type:
Default
Connection target:
MSURFACE
Find...
Connection type information
The default connection lets you select processes on this computer or a remote computer running the Visual Studio Remote Debugger (MSVSMON.EXE).
Attach to:
JavaScript (Microsoft Edge - Chromium) code
Select...
Available processes
Edge
Process ID Title Type User Name Session
MicrosoftEdge.exe 54028 Microsoft.MicrosoftEdge_44.1836... x64 DMOND\mo [low-rights] 1
MicrosoftEdgeCP.exe 39872 Microsoft Edge x64 DMOND\mo [low-rights] 1
MicrosoftEdgeCP.exe 47076 Microsoft Edge x64 DMOND\mo [low-rights] 1
MicrosoftEdgeSH.exe 33448 Microsoft.MicrosoftEdge_44.1836... x64 DMOND\mo [low-rights] 1
msedge.exe 54276 x64, JavaScript (Micr... DMOND\mo [untrusted] 1
msedge.exe 12688 x64, JavaScript (Micr... DMOND\mo [untrusted] 1
msedge.exe 30408 x64, JavaScript (Micr... DMOND\mo 1
msedge.exe 47924 x64, JavaScript (Micr... DMOND\mo [low-rights] 1
msedge.exe 54792 x64, JavaScript (Micr... DMOND\mo 1
mseдge.exe 52932 localhost - Microsoft Edge x64, JavaScript (Micr... DMOND\mo 1
Show processes from all users
Refresh
Attach
Cancel


Attach to Process
Connection type:
Default
Connection target:
MININT-VA9I
Find...
Connection type information
The default connection lets you select processes on this computer or a remote computer running the Visual Studio Remote Debugger (MSVSMON.EXE).
Attach to:
WebKit code
Select...
Available processes
chrome
Process ID Title Type User Name Session
chrome.exe 5896 WebKit, x64 DMOND\mo ... 1
chrome.exe 13412 WebKit, x64 DMOND\mo ... 1
chrome.exe 1004 WebKit, x64 DMOND\mo ... 1
chrome.exe 25324 WebKit, x64 DMOND\mo ... 1
chrome.exe 12056 WebKit, x64 DMOND\mo 1
chrome.exe 8884 WebKit, x64 DMOND\mo 1
chrome.exe 25500 localhost:1337 - Google Chrome WebKit, x64 DMOND\mo 1
Show processes from all users
Refresh
Attach
Cancel

You know the debugger has attached correctly when the DOM Explorer and the JavaScript Console open in Visual Studio. These debugging tools are similar to Chrome Developer Tools and F12 Tools for Microsoft Edge.

> **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.

Visual Studio provides debugging support for Chrome and Internet Explorer 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 Internet Explorer, go to **Solution Explorer** > **Script Documents** > **Windows Internet Explorer** > **YourPageName**.

For more information, see Client-side debugging of ASP.NET projects in Google Chrome.

Unit testing JavaScript and TypeScript in Visual Studio

2/7/2020 • 4 minutes to read • Edit Online

The Node.js Tools For Visual Studio allow you to write and run unit tests using some of the more popular JavaScript frameworks without the need to switch to a command prompt.

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)

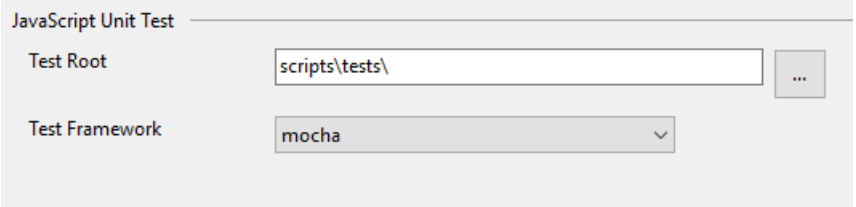
> **WARNING**
> An issue in Tape currently prevents Tape tests from running. If PR #361 is merged, the issue should be resolved.

If your favorite framework is not supported, see Add support for a unit test framework for information on adding support.

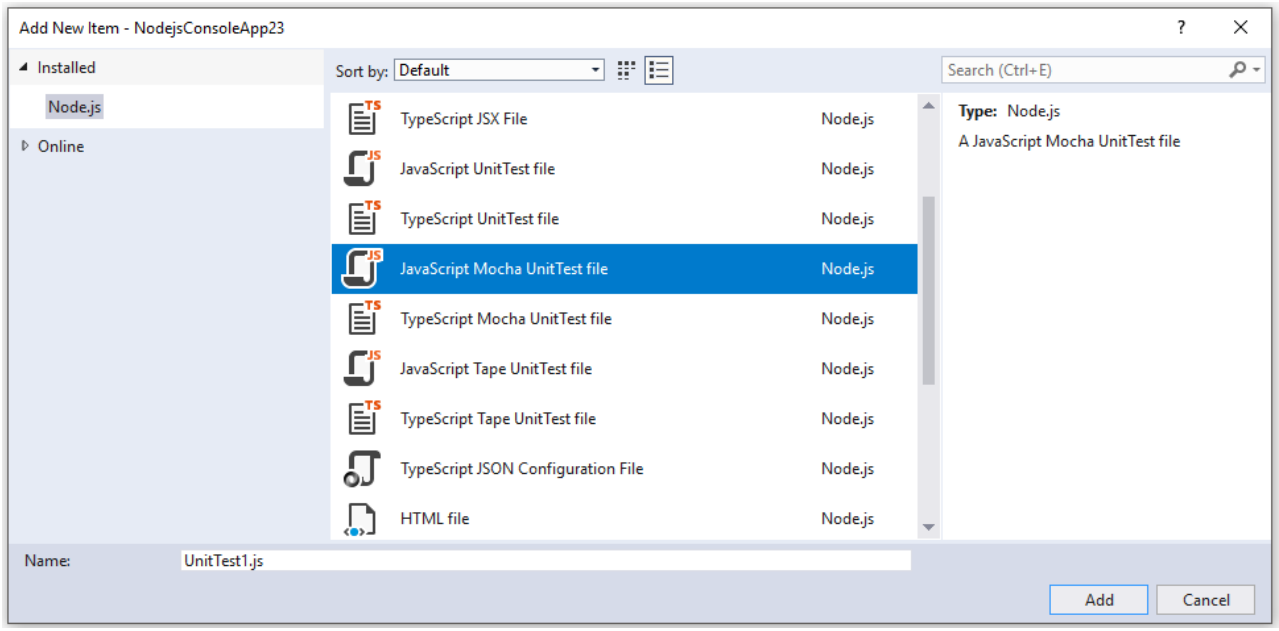
Write unit tests

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.



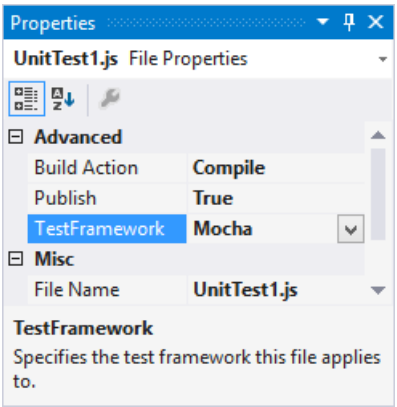
You can add simple blank tests to your project, using the **Add New Item** dialog box. Both JavaScript and TypeScript are supported in the same project.



For a Mocha unit test, use the following code:

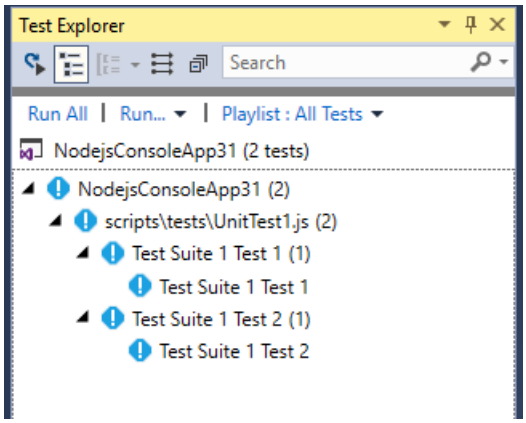
```
var assert = require('assert');

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**
>
> Do not use the `outdir` or `outfile` option in *tsconfig.json*, because Test Explorer won't be able to find your unit tests in TypeScript files.

Run tests

You can run tests in Visual Studio 2017 or from the command line.

Run tests in Visual Studio 2017

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 Selected Tests** 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 selecting **Debug Selected Tests**.

> **WARNING**
>
> Debugging unit tests using Node 8+ currently only works for JavaScript test files, TypeScript test files will fail to hit breakpoints. As a workaround use 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 the Developer Command Prompt for Visual Studio 2017 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.

Add support for a unit test framework

You can add support for additional test frameworks by implementing the discovery and execution logic using JavaScript. You do this by adding a folder with the name of the test framework under:

`<VisualStudioFolder>\Common7\IDE\Extensions\Microsoft\NodeJsTools\TestAdapter\TestFrameworks`

This folder has to contain a JavaScript file with the same name which exports the following two functions:

- `find_tests`
- `run_tests`

For good a 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 other project types

You are not limited to writing unit tests in just your Node.js 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.

> **NOTE**
>
> 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 properties above, you will 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

7/9/2019 • 2 minutes to read • Edit Online

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.