Visual Studio tutorials | JavaScript and TypeScript

Create JavaScript and TypeScript apps with Visual Studio

Get started

OVERVIEW

JavaScript and TypeScript in Visual Studio

HOW-TO GUIDE

Install Visual Studio

GET STARTED

Start a guided tour

Create JavaScript and TypeScript apps

QUICKSTART

Create an Angular App

Create a React App

Create a Vue App

TUTORIAL

Create a web app with Angular and ASP.NET Core

Create a web app with React and ASP.NET Core

Create a web app with Vue and ASP.NET Core

Create a web app with Node.js and Express

Add TypeScript to an ASP.NET Core app

Learn Visual Studio

HOW-TO GUIDE

Write and edit code

Linting JavaScript

Compile TypeScript code using tsc

Compile TypeScript code using NuGet

Manage npm packages

Debug your code

Develop code without projects or solutions

Write and run unit tests

JavaScript and TypeScript in Visual Studio

Article • 06/27/2024

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

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

TypeScript support

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

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

In MSBuild compilation scenarios such as ASP.NET Core, the TypeScript NuGet package is the recommended method of adding TypeScript compilation support to your project. Visual Studio will give you the option to add this package the first time you add a TypeScript file to your project. This package is also available at any time through the NuGet package manager. When the NuGet package is used, the corresponding

language service version will be used for language support in your project. Note: The minimum supported version of this package is 3.6.

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

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

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

Project templates

Starting in Visual Studio 2022, there is a new JavaScript/TypeScript project type (.esproj), called the JavaScript Project System (JSPS), which allows you to create standalone Angular, React, and Vue projects in Visual Studio. These front-end projects are created using the framework CLI tools you have installed on your local machine, so the version of the template is up to you. To migrate from existing Node.js projects to the new project system, see Migrate Node.js projects. For MSBuild information for the new project type, see MSBuild properties for JSPS

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

> **Note**
>
> A simplified, updated template is available starting in Visual Studio 2022 version 17.5. Compared to the ASP.NET SPA templates available in Visual Studio, the *.esproj*

SPA templates for ASP.NET Core provide better npm dependency management, and better build and publish support.

Feedback

Was this page helpful? Yes No

First look at the Visual Studio IDE

Article • 11/07/2023

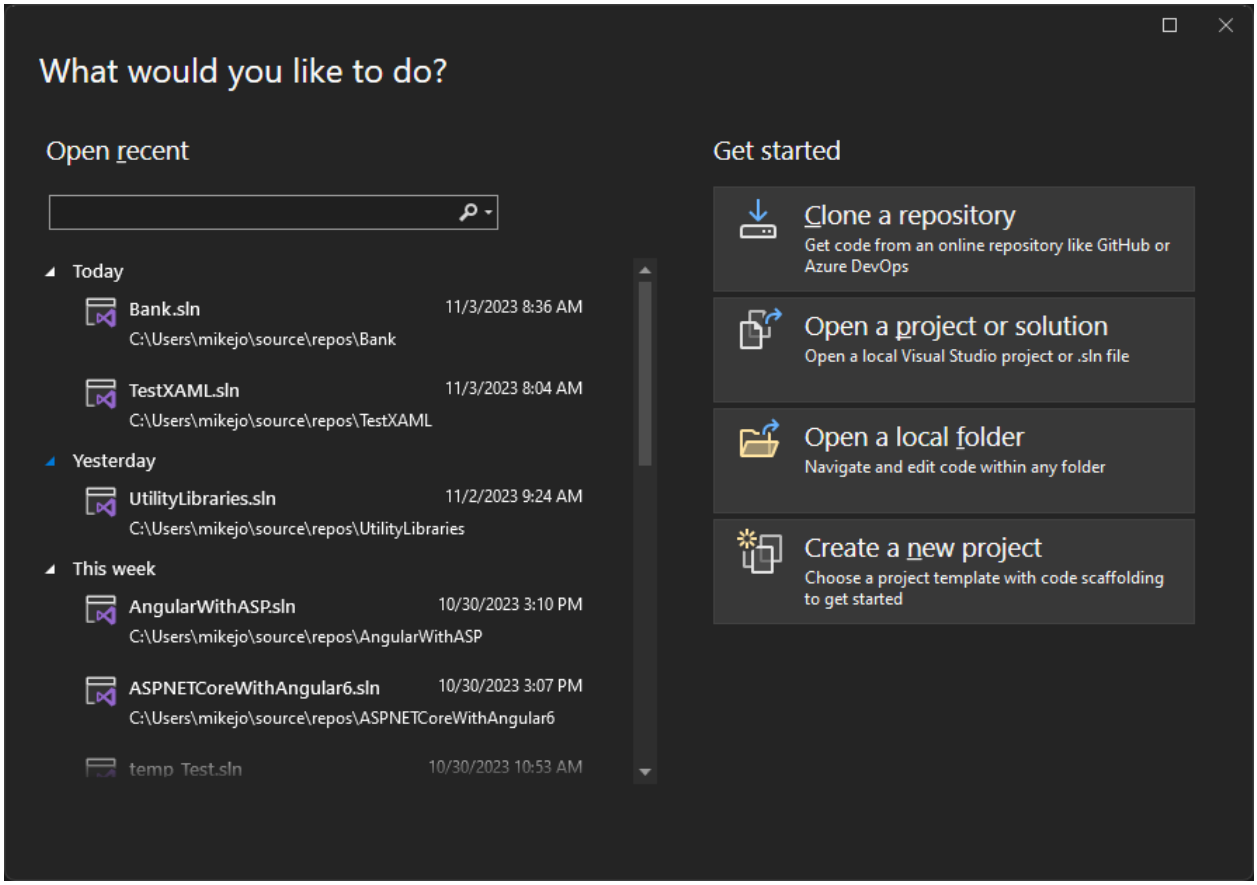
Applies to: Visual Studio | Visual Studio for Mac | Visual Studio Code

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

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

Start window

The first thing you 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, use the **Open a local folder** option to open your code in Visual Studio. For more information, see Develop code in Visual

Studio without projects or solutions. Otherwise, you can create a new project or clone a project from a source provider such as GitHub or Azure DevOps.

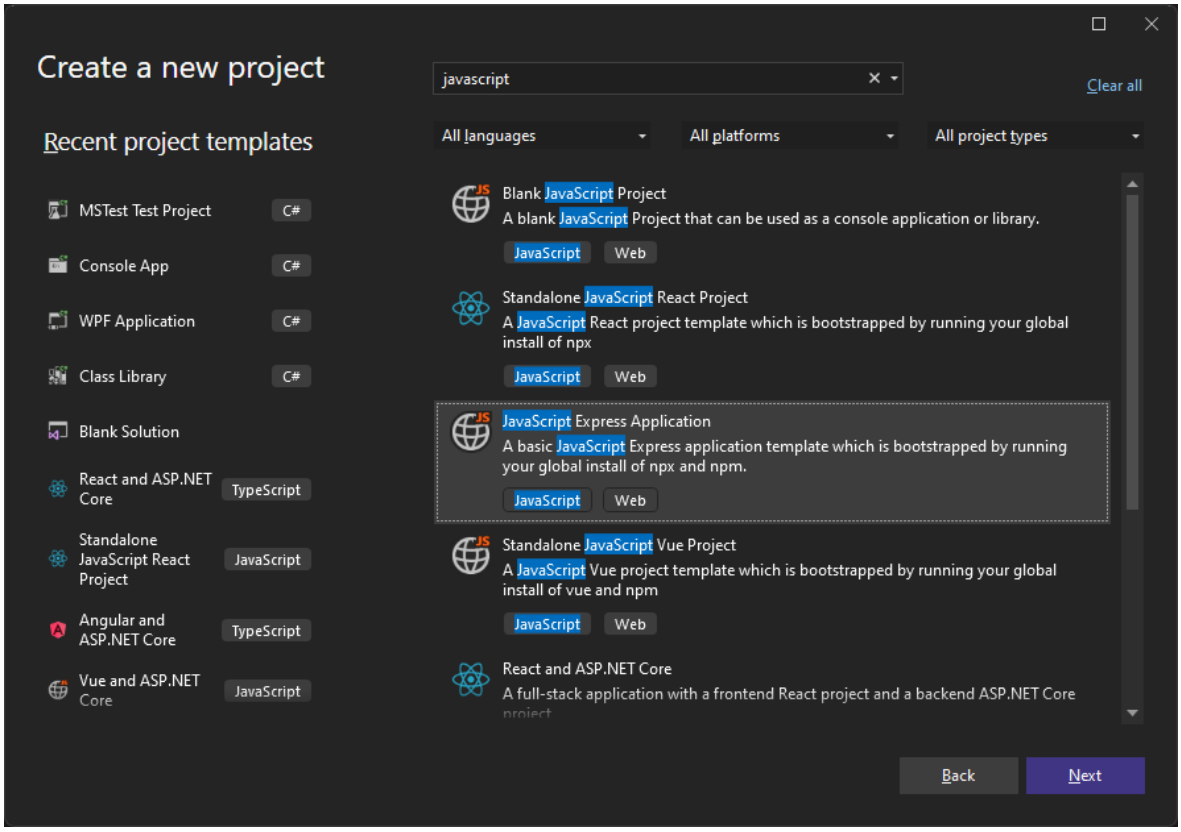
The **Continue without code** option simply opens the Visual Studio development environment without any specific project or code loaded. You might choose this option to join a Live Share session or attach to a process for debugging. You can also press **Esc** to close the start window and open the IDE.

Create a project

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

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

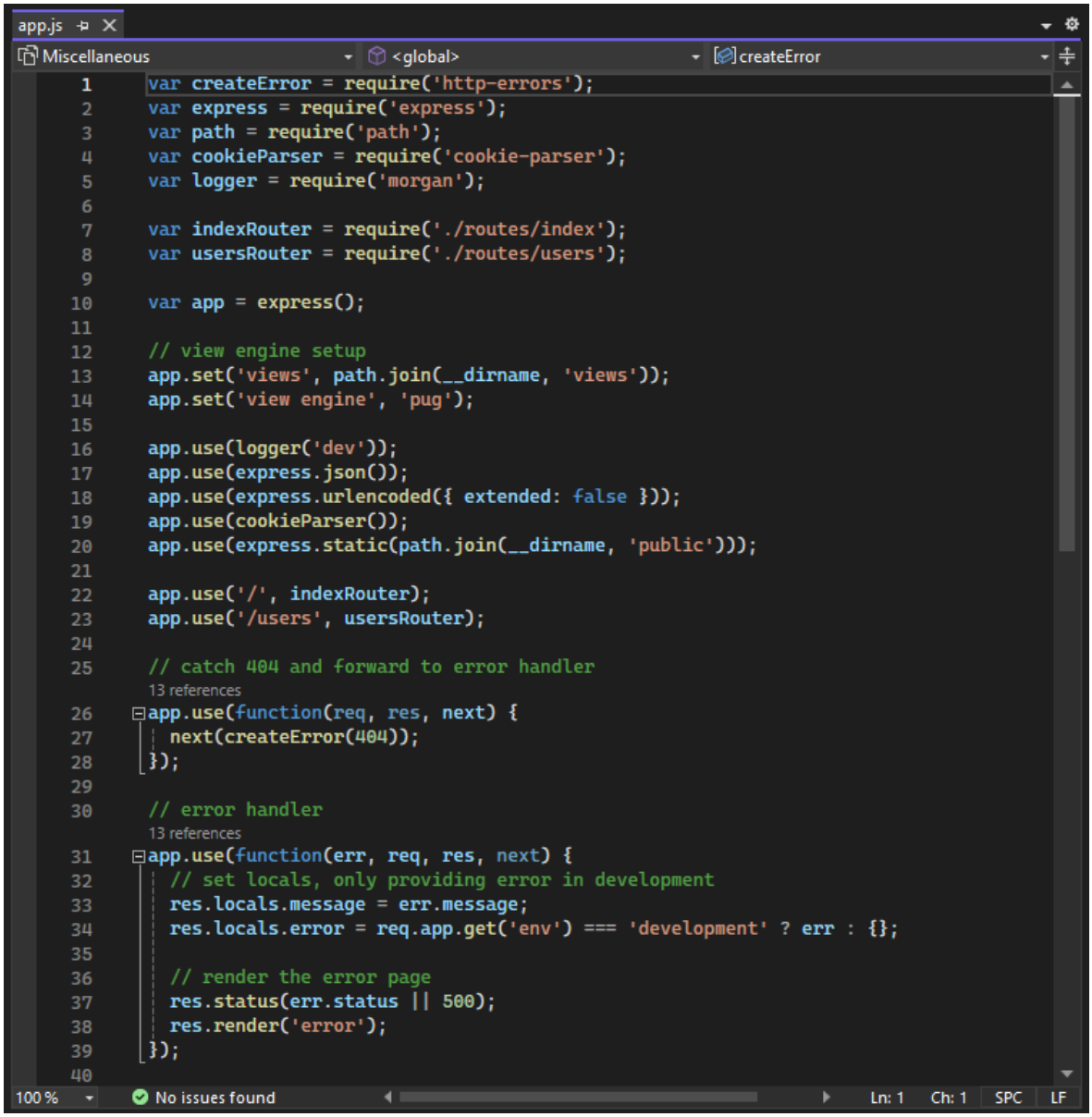
   Visual Studio provides various kinds of project templates that help you get started coding quickly.



2. Choose a **JavaScript Express Application** project template and click **Next**.

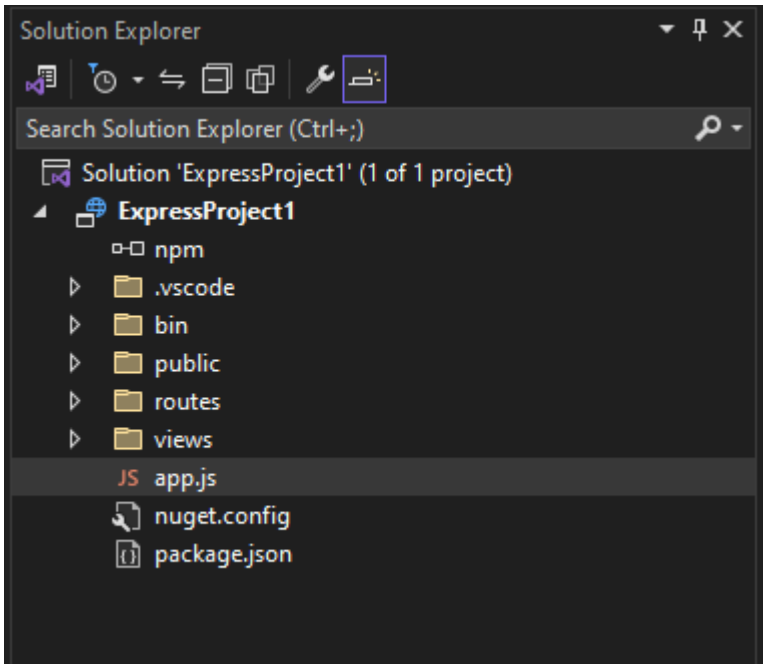
3. In the **Configure your new project** dialog box that appears, accept the default project name and choose **Create**.

The project is created. In the right pane, select *app.js* to open the file in the **Editor** window. The **Editor** shows the contents of files, and is where you do most of your coding work in Visual Studio.



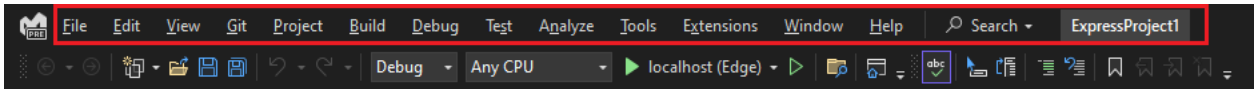
Solution Explorer

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

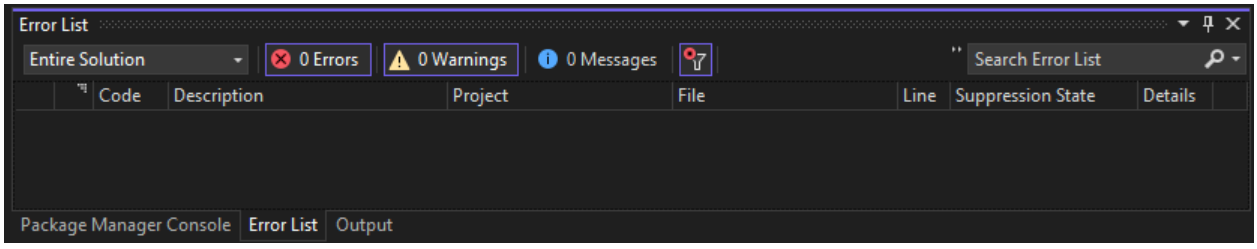
The menu bar along the top of Visual Studio groups commands into categories. For example, the **Project** menu contains commands related to the project you're working in. On the **Tools** menu, you can customize how Visual Studio behaves by selecting **Options**, or add features to your installation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



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

Error List

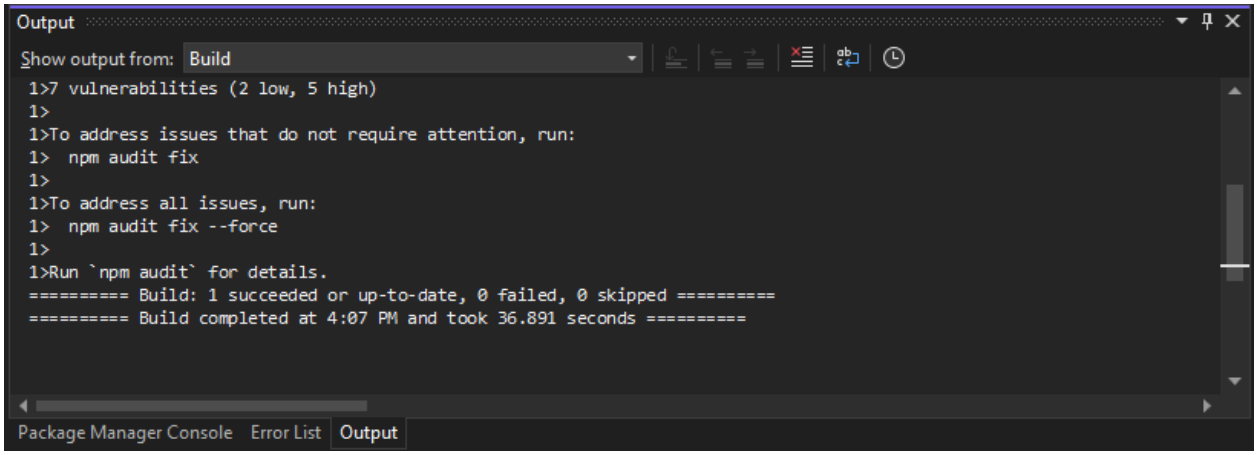
The **Error List** shows you errors, warning, and messages regarding the current state of your code. If there are any errors (such as a missing brace or semicolon) in your file, or anywhere in your project, they're listed here.



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. If you don't see the search box, press **Ctrl** + **Q** to open it.

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

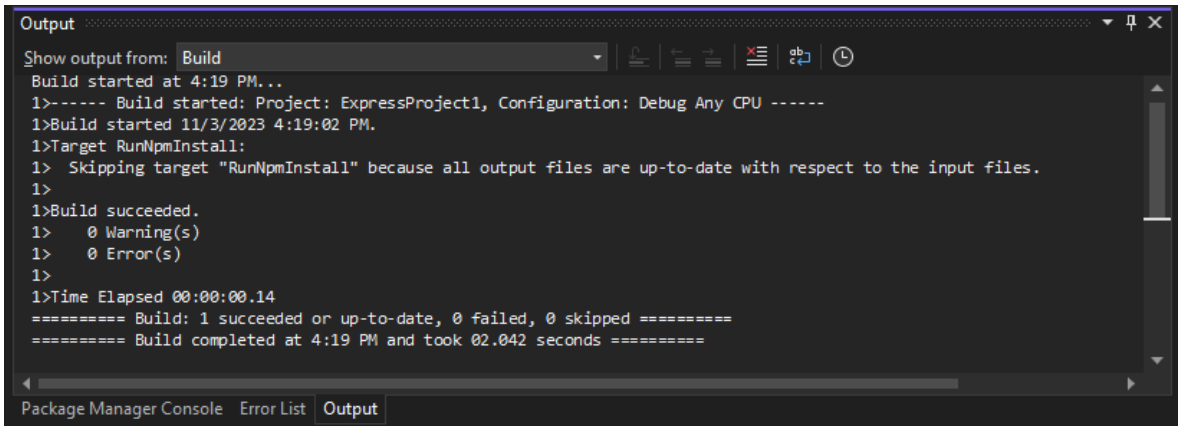
   

   The **Options** dialog box opens to the **Build and Run** options page.

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

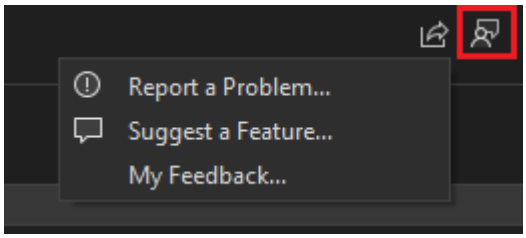
4. Build the project again by right-clicking the project in **Solution Explorer** and choosing **Rebuild** from the context menu.

   This time the **Output** window shows more verbose logging from the build process.



Send Feedback menu

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



Next steps

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

Learn about the code editor

Learn about projects and solutions

See also

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

Create a React app

Article • 05/23/2024

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

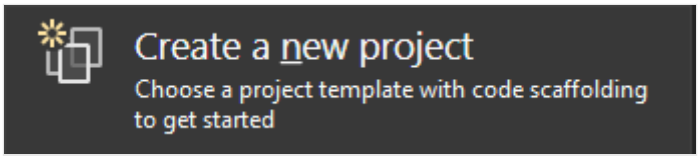
Prerequisites

Make sure to install the following:

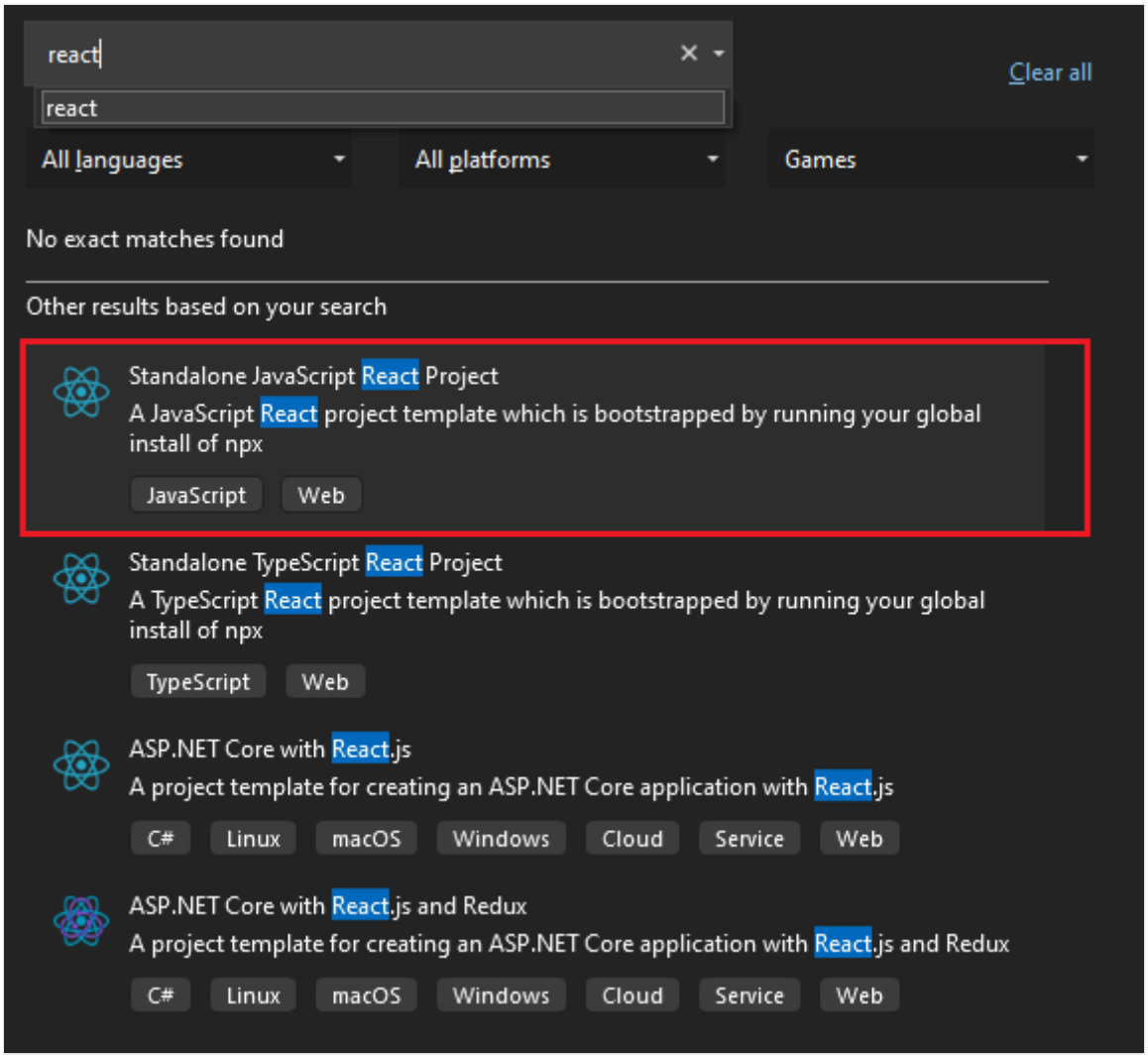
- Visual Studio 2022 or later. Go to the Visual Studio downloads page to install it for free.
- npm (https://www.npmjs.com/), which is included with Node.js
- npx (https://www.npmjs.com/package/npx)

Create your app

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

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



3. Give your project and solution a name.

4. Choose **Create**, and then wait for Visual Studio to create the project.

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

View the project properties

The default project settings allow you to build and debug the project. But, if you need to change settings, right-click the project in Solution Explorer, select **Properties**, and then go the **Build** or **Debugging** section.

> ⓘ **Note**
>
> `launch.json` stores the startup settings associated with the **Start** button in the Debug toolbar. Currently, `launch.json` must be located under the `.vscode` folder.

Build Your Project

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

Start Your Project

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

- VITE v4.4.9 ready in 780 ms

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

Next, you should see the base React app appear!

Next steps

For ASP.NET Core integration:

Create an ASP.NET Core app with React

Feedback

Was this page helpful? Yes No

Create an Angular app

Article • 05/23/2024

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

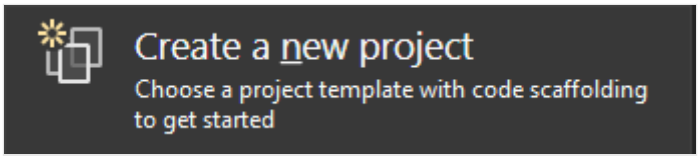
Prerequisites

Make sure to install the following:

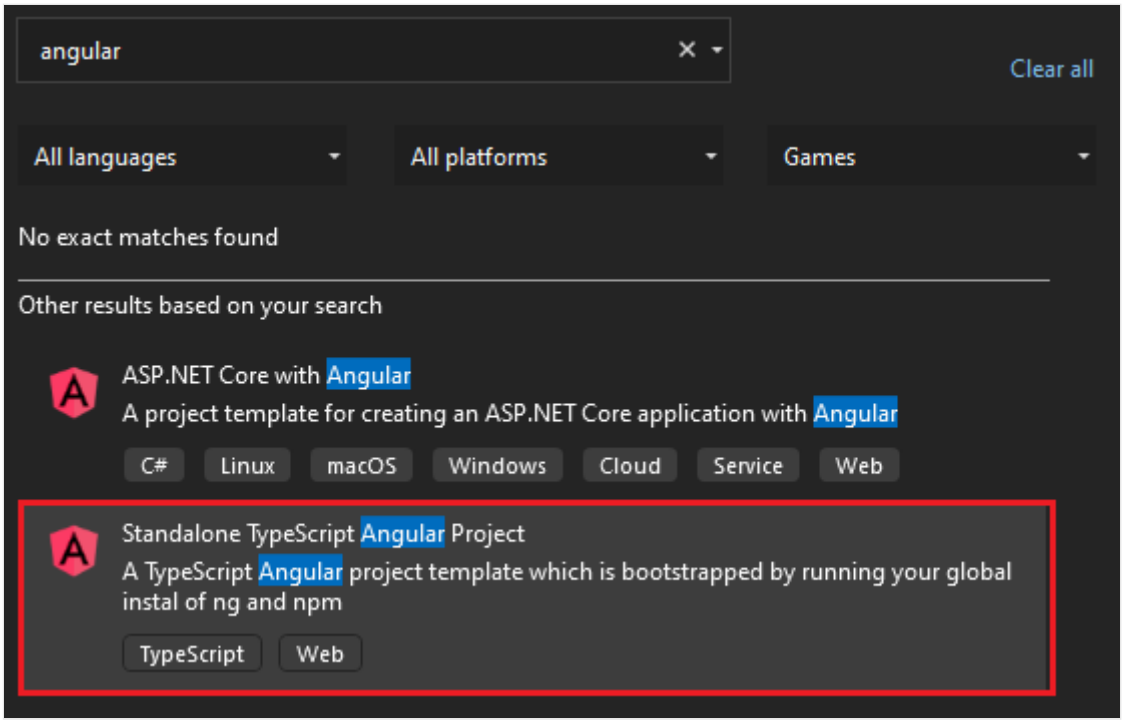
- Visual Studio 2022 or later. Go to the Visual Studio downloads page to install it for free.
- npm (https://www.npmjs.com/), which is included with Node.js
- Angular CLI (https://angular.io/cli) This can be the version of your choice

Create your app

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

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



3. Give your project and solution a name.

4. Choose **Create**, and then wait for Visual Studio to create the project.

View the project properties

The default project settings allow you to build and debug the project. But, if you need to change settings, right-click the project in Solution Explorer, select **Properties**, and then go the **Build** or **Debugging** section.

> ⓘ Note
>
> `launch.json` stores the startup settings associated with the **Start** button in the Debug toolbar. Currently, `launch.json` must be located under the `.vscode` folder.

Build Your Project

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

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

Start Your Project

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

- The Angular CLI running the ng start command

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

Next, you should see the base Angular apps appear!

Next steps

For ASP.NET Core integration:

Create an ASP.NET Core app with Angular

Feedback

Create a Vue.js app

Article • 05/23/2024

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

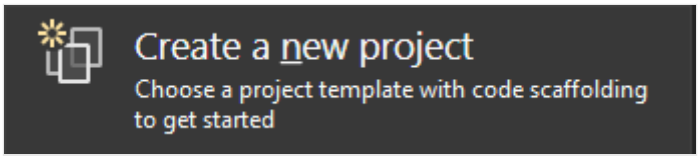
Prerequisites

Make sure to install the following:

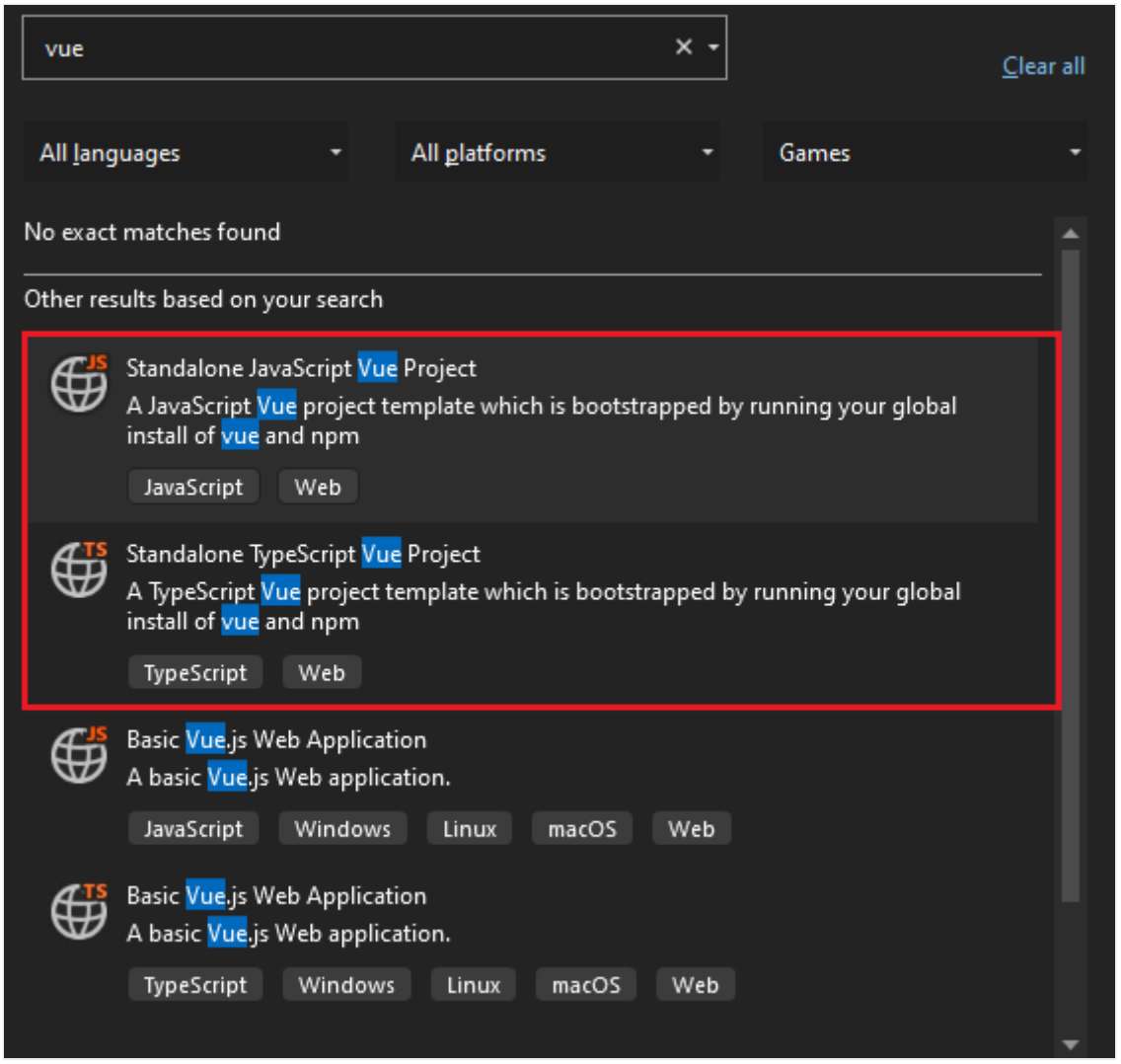
- Visual Studio 2022 or later. Go to the Visual Studio downloads page to install it for free.
- npm (https://www.npmjs.com/), which is included with Node.js
- Vue.js (Installation | Vue.js (vuejs.org))

Create your app

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

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



3. Give your project and solution a name, and then choose **Next**.

4. Choose **Create**, and then wait for Visual Studio to create the project.

View the project properties

The default project settings allow you to build and debug the project. But, if you need to change settings, right-click the project in Solution Explorer, select **Properties**, and then go the **Build** or **Debugging** section.

> ⓘ Note
>
> `launch.json` stores the startup settings associated with the **Start** button in the Debug toolbar. Currently, `launch.json` must be located under the `.vscode` folder.

Build Your Project

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

Start Your Project

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

- VITE v4.4.9 ready in 780 ms

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

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

Next steps

For ASP.NET Core integration:

Create an ASP.NET Core app with Vue

Feedback

Was this page helpful? Yes No

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

Article • 01/21/2023

In this article, you will learn how to use Visual Studio to build a simple Node.js web app that uses the Express framework.

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

- **What is Node.js?**

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

- **What is npm?**

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

- **What is Express?**

  Express is a server web application framework that Node.js uses to build web apps. With Express, there are many different ways to create a user interface. The implementation provided in this tutorial uses the Express application generator's default template engine, called Pug, to render the front-end.

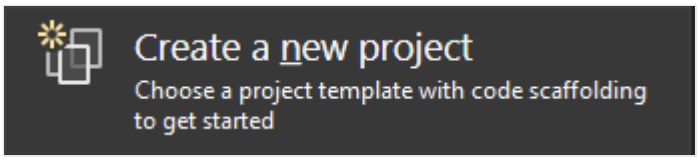
Prerequisites

Make sure to install the following:

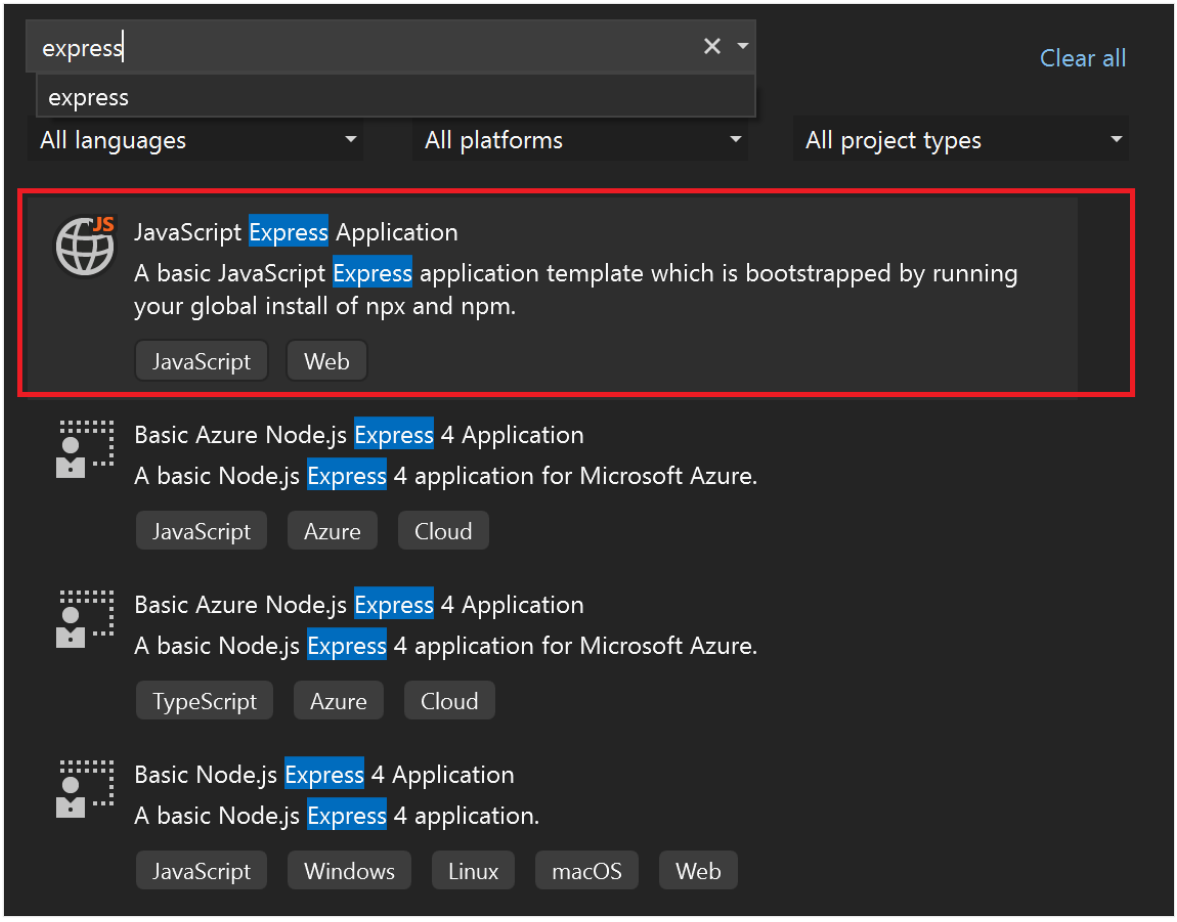
- Visual Studio 2022 **version 17.4** or later with the **ASP.NET and web development** workload installed. Go to the Visual Studio downloads page to install it for free. If you need to install the workload and already have Visual Studio, go to **Tools** > **Get Tools and Features...**, which opens the Visual Studio Installer. Choose the **ASP.NET and web development** workload, then choose **Modify**.
- npm (https://www.npmjs.com/), which is included with Node.js
- npx (https://www.npmjs.com/package/npx)

Create your app

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



2. Search for Express in the search bar at the top and then select **JavaScript Express Application**.



3. Give your project and solution a name.

View the project properties

The default project settings allow you to build and debug the project. But, if you need to change settings, right-click the project in Solution Explorer, select **Properties**, and then go the **Build** or **Debugging** section.

> **Note**
>
> `launch.json` stores the startup settings associated with the **Start** button in the Debug toolbar. Currently, `launch.json` must be located under the `.vscode` folder.

Build your project

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

Start your app

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

- npm running the node ./bin/www command

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

Next, you should see the base Express app appear!

Debug your app

We will now go through a couple of ways you can debug your app.

First, if your app is still running, press **Shift + F5** or select the red stop button at the top of the window in order to stop the current session. You might notice that stopping the session closes the browser showing your app, but leaves behind the command prompt window running the Node process. For now, go ahead and close any lingering command prompts. Later in this article, we describe why you might want to leave the Node process running.

Debugging the Node process

In the dropdown next to the **Start** button, you should see the following start options:

- localhost (Edge)
- localhost (Chrome)
- Debug Dev Env
- Launch Node and Browser

Go ahead and select the **Launch Node and Browser** option. Now, before pressing **F5** or selecting the **Start** button again, set a breakpoint in `index.js` (in the **routes** folder) by

selecting the left gutter before the following line of code: `res.render('index', { title: 'Express' });`

> **Tip**
>
> You can also put your cursor on a line of code and hit **F9** to toggle the breakpoint for that line.

Then, press **F5** or select **Debug** > **Start Debugging** to debug your app.

You should see the debugger pause at the breakpoint you just set. While it is paused, you can inspect your app state. Hovering over variables will let you examine their properties.

When you're finished inspecting the state, hit **F5** to continue, and your app should load as expected.

This time, if you hit stop, you will notice that both the browser and the command prompt windows close. To see why, take a closer look at the *launch.json*.

Understanding the *launch.json*

The `launch.json` is currently located in the `.vscode` folder. If you cannot see the `.vscode` folder in **Solution Explorer**, select **Show All Files**.

If you have worked with Visual Studio Code before, the `launch.json` file will look familiar. The `launch.json` here works in much the same way as it does in Visual Studio Code to denote launch configurations used for debugging. Each entry specifies one or more targets to be debugged.

The first two entries are browser entries, and they should look something like this:

JSON

```
    {
      "name": "localhost (Edge)",
      "type": "edge",
      "request": "launch",
      "url": "http://localhost:3000",
      "webRoot": "${workspaceFolder}\\public"
    },
    {
      "name": "localhost (Chrome)",
      "type": "chrome",
      "request": "launch",
```

```
    "url": "http://localhost:3000",
    "webRoot": "${workspaceFolder}\\public"
}
```

You can see in the above entries that the `type` is set to a browser type. If you launch with only a browser type as the sole debug target, Visual Studio will debug only the frontend browser process, and the Node process will be started without a debugger attached, meaning that any breakpoints that are set in the Node process **will not** bind.

Upon stopping the session, the Node process will also continue to run. It is intentionally left running when a browser is the debug target, because if work is solely being done on the frontend, having the backend process continuously running eases the development workflow.

At the start of this section, you closed the lingering command prompt window in order to set breakpoints in the Node process. For the Node process to be debuggable, it must be restarted with the debugger attached. If a non-debuggable Node process is left running, attempting to launch the Node process in debug mode (without reconfiguring the port) **will fail**.

> ⓘ **Note**
>
> Currently, `edge` and `chrome` are the only supported browser types for debugging.

The third entry in the `launch.json` specifies `node` as the debug type, and it should look something like this:

```
{
  "name": "Debug Dev Env",
  "type": "node",
  "request": "launch",
  "cwd": "${workspaceFolder}/bin",
  "program": "${workspaceFolder}/bin/www",
  "stopOnEntry": true
}
```

This entry will launch only the Node process in debug mode. No browser will be launched.

The fourth entry provided in the "launch.json* is the following compound launch configuration.

```
{
  "name": "Launch Node and Browser",
  "configurations": [
    "Debug Dev Env",
    "localhost (Edge)"
  ]
}
```

This compound configuration is the same as a vscode compound launch configuration, and selecting it allows you to debug both the frontend and backend. You can see that it simply references the individual launch configurations for the Node and browser processes.

There are many other attributes you can use in a launch configuration. For example, you can hide a configuration from the dropdown, but still have it be referenceable, by setting the `hidden` attribute in the `presentation` object to `true`.

```
{
  "name": "localhost (Chrome)",
  "type": "chrome",
  "request": "launch",
  "url": "http://localhost:3000",
  "webRoot": "${workspaceFolder}\\public",
  "presentation": {
    "hidden": true
  }
}
```

Click Options for a list of attributes you can use to enhance your debugging experience. Please note that at the moment, only **launch** configurations are supported. Any attempt to use an **attach** configuration will result in a deployment failure.

Feedback

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

Article • 05/14/2024

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

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

You can use the method described in this article to create ASP.NET Core Single Page Applications that:

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

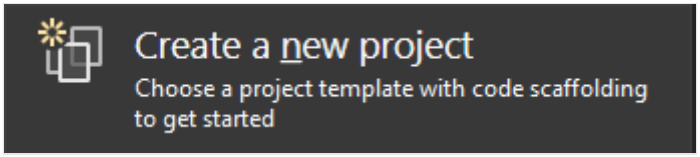
> **Note**
>
> This article describes the project creation process using the updated template in Visual Studio 2022 version 17.8, which uses the Vite CLI.

Prerequisites

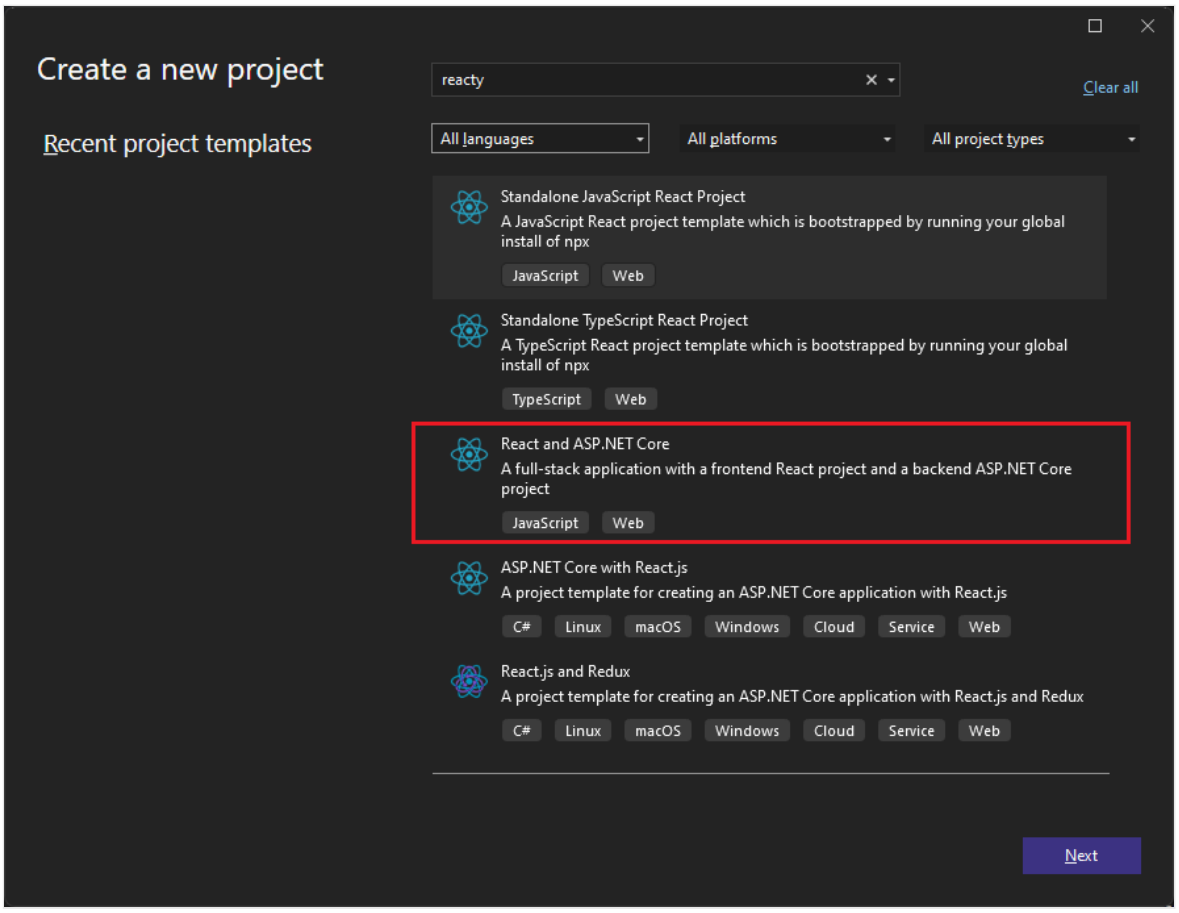
- Visual Studio 2022 version 17.8 or later with the **ASP.NET and web development** workload installed. Go to the Visual Studio downloads page to install it for free. If you need to install the workload and already have Visual Studio, go to **Tools** > **Get Tools and Features...**, which opens the Visual Studio Installer. Choose the **ASP.NET and web development** workload, then choose **Modify**.
- npm (https://www.npmjs.com/), which is included with Node.js
- npx (https://www.npmjs.com/package/npx)

Create the frontend app

1. In the Start window, select **Create a new project**.



2. Search for React in the search bar at the top and then select **React and ASP.NET Core**. This template is a JavaScript template.

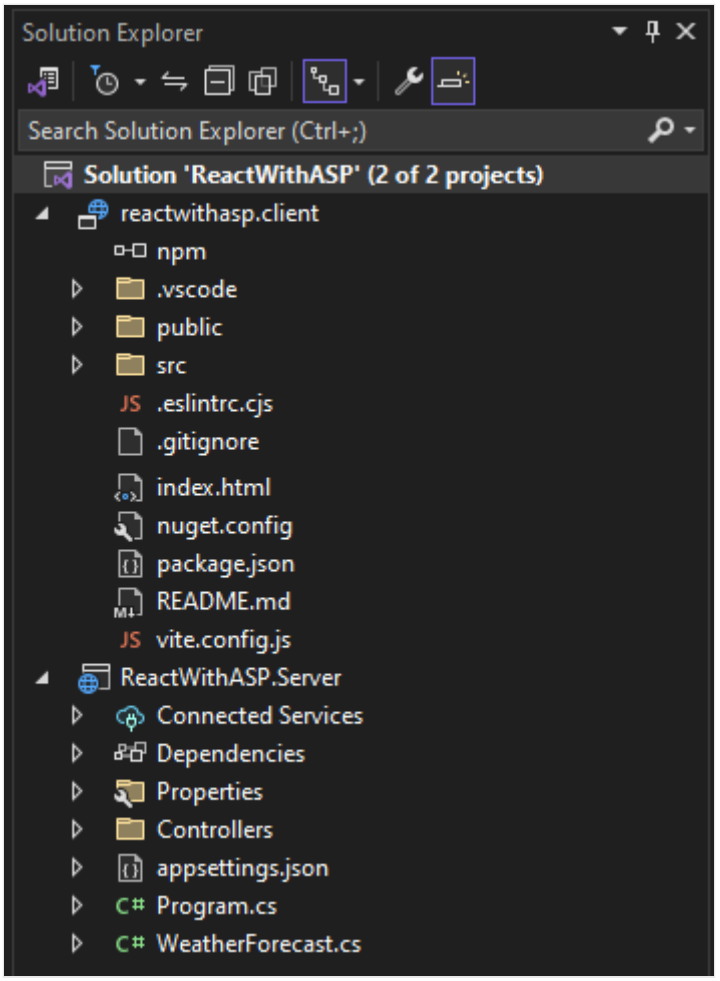


3. Name the project **ReactWithASP** and then select **Next**.

   In the Additional Information dialog, make sure that **Configure for HTTPS** is enabled. In most scenarios, leave the other settings at the default values.

4. Select **Create**.

   Solution Explorer shows the following project information:



Compared to the standalone React template, you see some new and modified files for integration with ASP.NET Core:

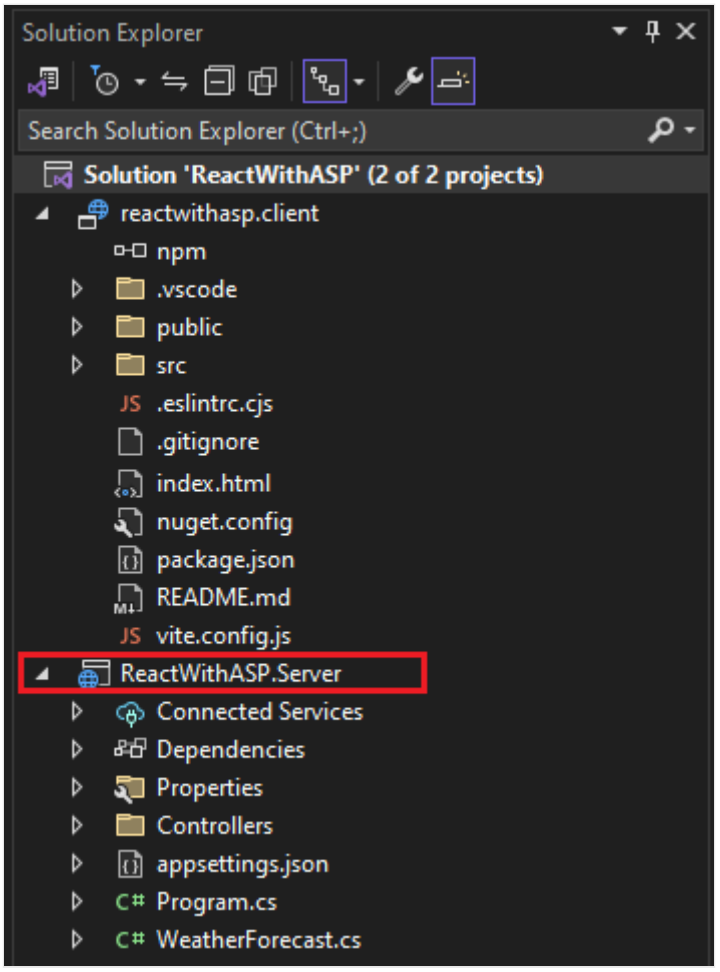
- vite.config.js
- App.js (modified)
- App.test.js (modified)

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

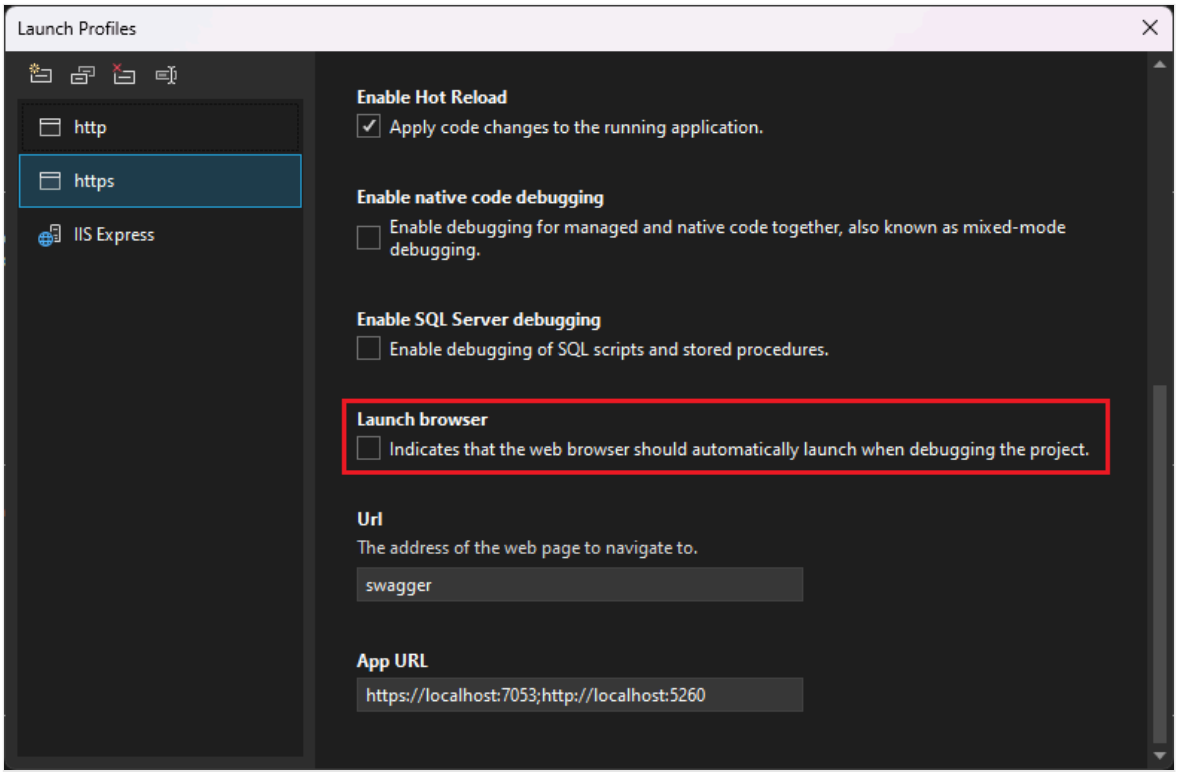
   If the browser you want is not yet installed, install the browser first, and then select it.

Set the project properties

1. In Solution Explorer, right-click the **ReactWithASP.Server** project and choose **Properties**.



2. In the Properties page, open the **Debug** tab and select **Open debug launch profiles UI** option. Uncheck the **Launch Browser** option for the **https** profile or the profile named after the ASP.NET Core project, if present.



This value prevents opening the web page with the source weather data.

> ⓘ Note
>
> In Visual Studio, `launch.json` stores the startup settings associated with the **Start** button in the Debug toolbar. Currently, `launch.json` must be located under the `.vscode` folder.

3. Right-click the solution in Solution Explorer and select **Properties**. Verify that the Startup project settings are set to **Multiple projects**, and that the Action for both projects is set to **Start**.

Start the project

Press **F5** or select the **Start** button at the top of the window to start the app. Two command prompts appear:

- The ASP.NET Core API project running
- The Vite CLI showing a message such as `VITE v4.4.9 ready in 780 ms`

> ⓘ Note
>
> Check console output for messages. For example there might be a message to update Node.js.

The React app appears and is populated via the API. If you don't see the app, see Troubleshooting.

Publish the project

1. In Solution Explorer, right-click the **ReactWithASP.Server** project and select **Add** > **Project Reference**.

   Make sure the **reactwithasp.client** project is selected.

2. Choose **OK**.

3. Right-click the ASP.NET Core project again and select **Edit Project File**.

   This opens the `.csproj` file for the project.

4. In the `.csproj` file, make sure the project reference includes a `<ReferenceOutputAssembly>` element with the value set to `false`.

   This reference should look like the following.

   XML

   ```
    <ProjectReference
   Include="..\reactwithasp.client\reactwithasp.client.esproj">
      <ReferenceOutputAssembly>false</ReferenceOutputAssembly>
    </ProjectReference>
   ```

5. Right-click the ASP.NET Core project and choose **Reload Project** if that option is available.

6. In *Program.cs*, make sure the following code is present.

   C#

   ```
   app.UseDefaultFiles();
   app.UseStaticFiles();

   // Configure the HTTP request pipeline.
   if (app.Environment.IsDevelopment())
   {
      app.UseSwagger();
      app.UseSwaggerUI();
   }
   ```

7. To publish, right click the ASP.NET Core project, choose **Publish**, and select options to match your desired publish scenario, such as Azure, publish to a folder, etc.

   The publish process takes more time than it does for just an ASP.NET Core project, since the `npm run build` command gets invoked when publishing. The BuildCommand runs `npm run build` by default.

Troubleshooting

Proxy error

You may see the following error:

Windows Command Prompt

```
[HPM] Error occurred while trying to proxy request /weatherforecast from
localhost:4200 to https://localhost:7183 (ECONNREFUSED)
(https://nodejs.org/api/errors.html#errors_common_system_errors)
```

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

Verify ports

If the weather data doesn't load correctly, you may also need to verify that your ports are correct.

1. Make sure that the port numbers match. Go to the `launchSettings.json` file in the ASP.NET Core **ReactWithASP.Server** project (in the *Properties* folder). Get the port number from the `applicationUrl` property.

   If there are multiple `applicationUrl` properties, look for one using an `https` endpoint. It looks similar to `https://localhost:7183`.

2. Open the `vite.config.js` file for the React project. Update the `target` property to match the `applicationUrl` property in *launchSettings.json*. The updated value looks similar to the following:

   JavaScript

   ```
   target: 'https://localhost:7183/',
   ```

Privacy error

You may see the following certificate error:

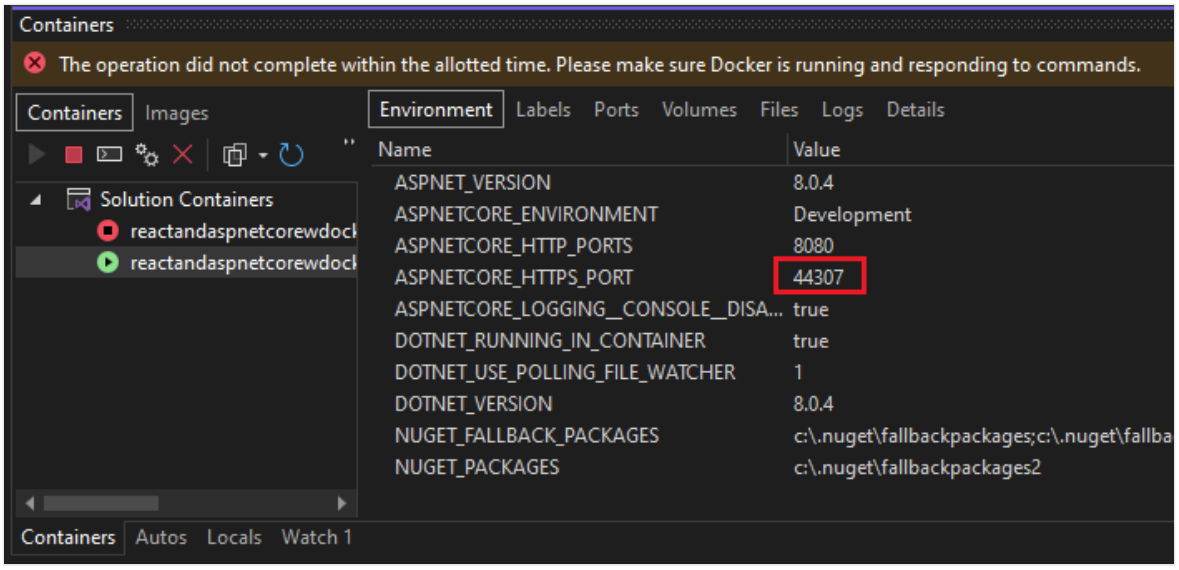
```
Your connection isn't private
```

Try deleting the React certificates from *%appdata%\local\asp.net\https* or *%appdata%\roaming\asp.net\https*, and then retry.

Docker

If you create the project with Docker support enabled, take the following steps:

1. After the app loads, get the Docker HTTPS port using the Containers window in Visual Studio. Check the **Environment** or **Ports** tab.



2. Open the `vite.config.js` file for the React project. Update the `target` variable to match the HTTPS port in the Containers window. For example, in the following code:

```
JavaScript
```

```javascript
const target = env.ASPNETCORE_HTTPS_PORT ?
`https://localhost:${env.ASPNETCORE_HTTPS_PORT}` :
   env.ASPNETCORE_URLS ? env.ASPNETCORE_URLS.split(';')[0] :
'https://localhost:7143';
```

   change `https://localhost:7143` to `https://localhost:44307`.

3. Restart the app.

Next steps

For more information about SPA applications in ASP.NET Core, see the React section under Developing Single Page Apps. The linked article provides additional context for project files such as *aspnetcore-https.js*, although details of the implementation are different based on the template differences. For example, instead of a ClientApp folder, the React files are contained in a separate project.

For MSBuild information specific to the client project, see MSBuild properties for JSPS.

Feedback

Was this page helpful? Yes No

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

Article • 06/19/2024

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

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

You can use the method described in this article to create ASP.NET Core Single Page Applications that:

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

> **Note**
>
> This article describes the project creation process using the updated template in Visual Studio 2022 version 17.8.

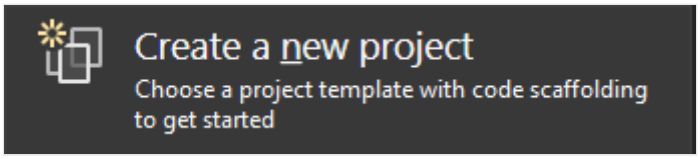
Prerequisites

Make sure to install the following:

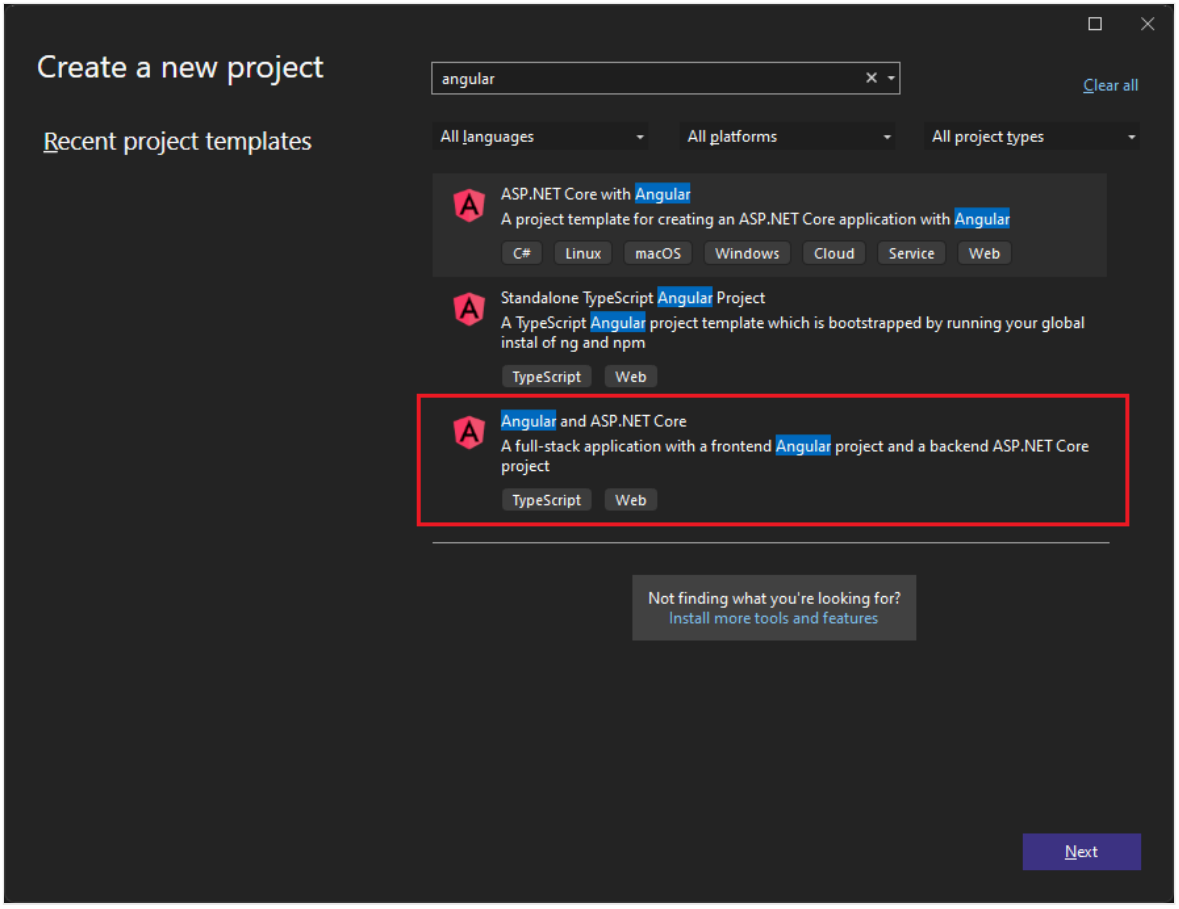
- Visual Studio 2022 version 17.8 or later with the **ASP.NET and web development** workload installed. Go to the Visual Studio downloads page to install it for free. If you need to install the workload and already have Visual Studio, go to **Tools** > **Get Tools and Features...**, which opens the Visual Studio Installer. Choose the **ASP.NET and web development** workload, then choose **Modify**.
- npm (https://www.npmjs.com/), which is included with Node.js
- Angular CLI (https://angular.io/cli), which can be the version of your choice.

Create the frontend app

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



2. Search for Angular in the search bar at the top and then select **Angular and ASP.NET Core**.

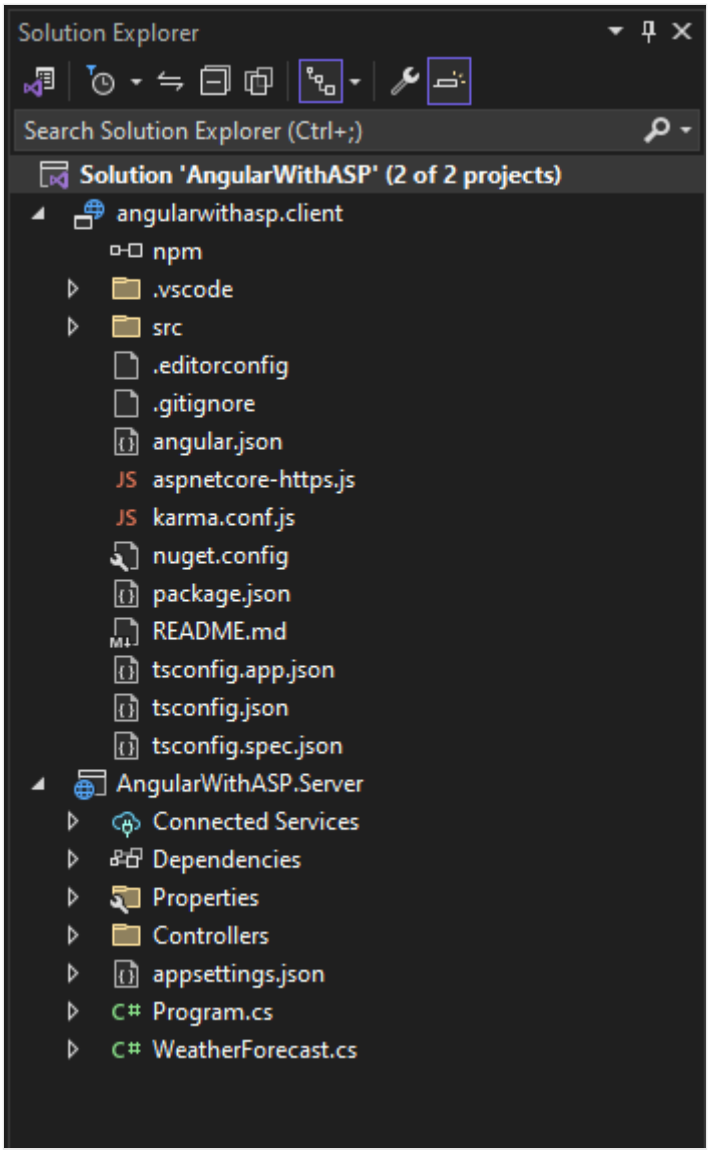


3. Name the project **AngularWithASP** and then select **Next**.

   In the Additional Information dialog, make sure that **Configure for HTTPS** is enabled. In most scenarios, leave the other settings at the default values.

4. Select **Create**.

   Solution Explorer shows the following::



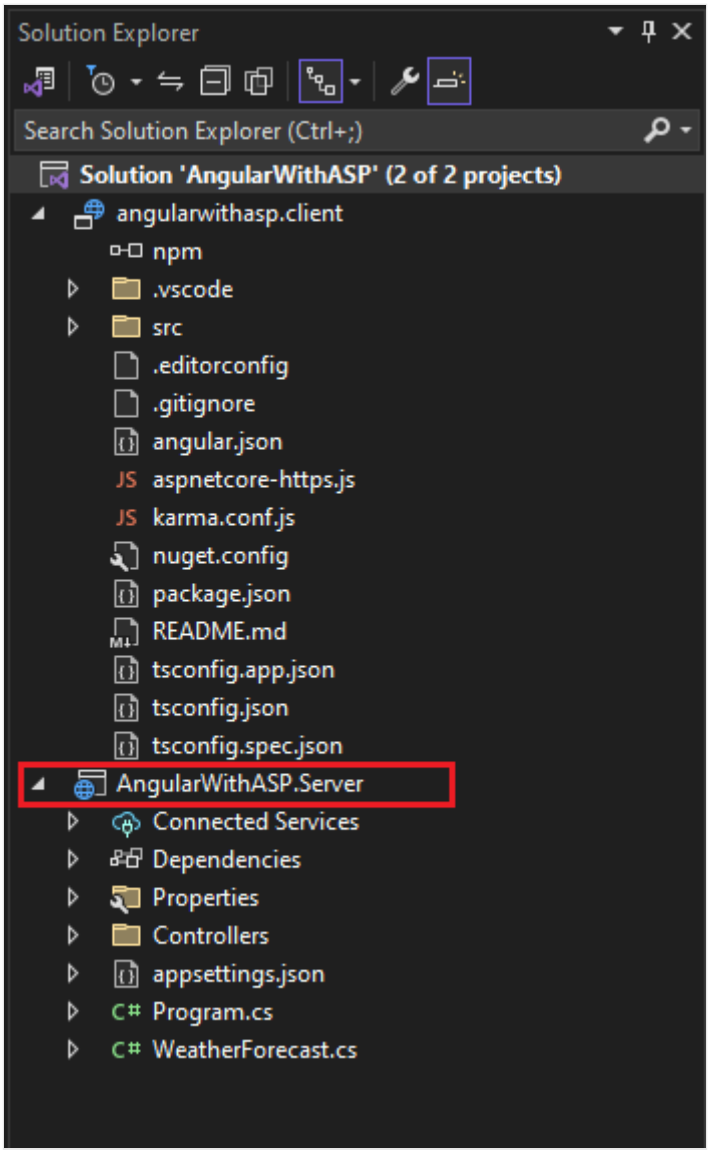
Compared to the standalone Angular template, you see some new and modified files for integration with ASP.NET Core:

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

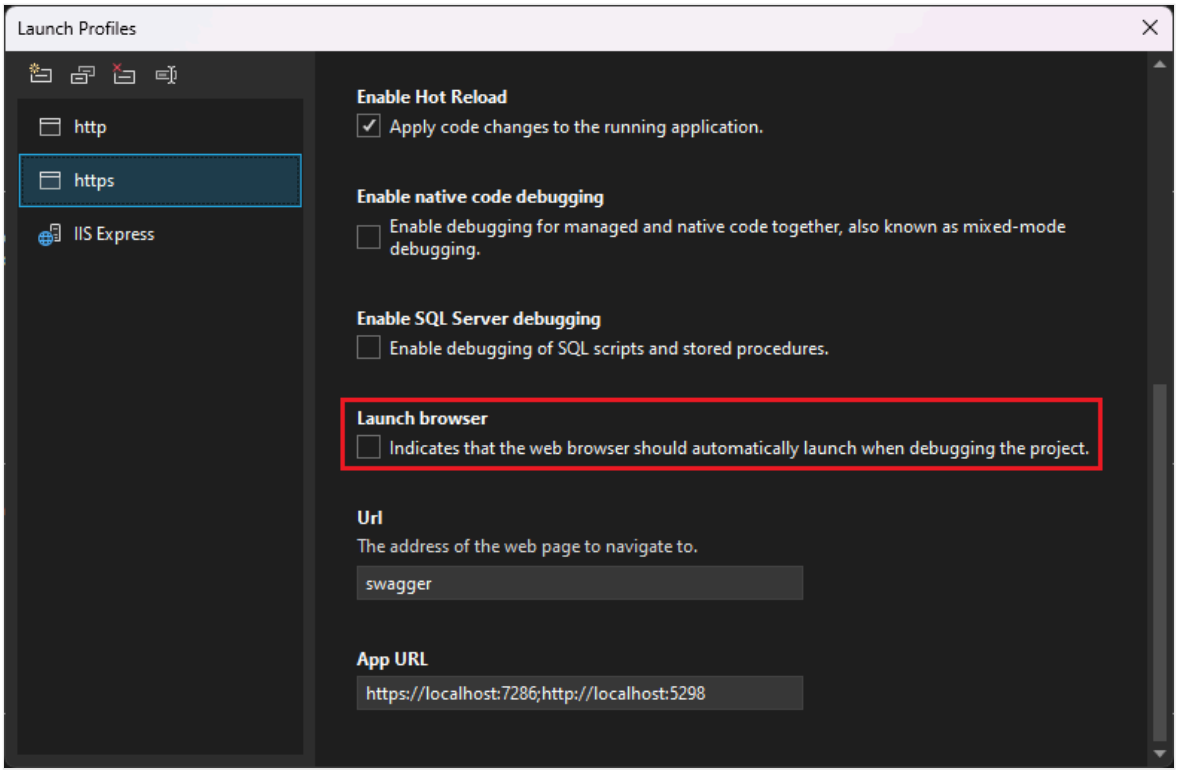
For more information on some of these project files, see Next steps.

Set the project properties

1. In Solution Explorer, right-click the **AngularWithASP.Server** project and choose **Properties**.



2. In the Properties page, open the **Debug** tab and select **Open debug launch profiles UI** option. Uncheck the **Launch Browser** option for the **https** profile or the profile named after the ASP.NET Core project, if present.



This value prevents opening the web page with the source weather data.

> ⓘ Note
>
> In Visual Studio, `launch.json` stores the startup settings associated with the **Start** button in the Debug toolbar. `launch.json` must be located under the `.vscode` folder.

3. Right-click the solution in Solution Explorer and select **Properties**. Verify that the Startup project settings are set to **Multiple projects**, and that the Action for both projects is set to **Start**.

Start the project

Press **F5** or select the **Start** button at the top of the window to start the app. Two command prompts appear:

- The ASP.NET Core API project running
- The Angular CLI running the ng start command

> ⓘ Note
>
> Check console output for messages. For example there might be a message to update Node.js.

The Angular app appears and is populated via the API. If you don't see the app, see Troubleshooting.

Publish the project

Starting in Visual Studio 2022 version 17.3, you can publish the integrated solution using the Visual Studio Publish tool.

> **Note**
>
> To use publish, create your JavaScript project using Visual Studio 2022 version 17.3 or later.

1. In Solution Explorer, right-click the **AngularWithASP.Server** project and select **Add** > **Project Reference**.

   Make sure the **angularwithasp.client** project is selected.

2. Choose **OK**.

3. Right-click the ASP.NET Core project again and select **Edit Project File**.

   This opens the `.csproj` file for the project.

4. In the `.csproj` file, make sure the project reference includes a `<ReferenceOutputAssembly>` element with the value set to `false`.

   This reference should look like the following.

   XML

   ```
    <ProjectReference
   Include="..\angularwithasp.client\angularwithasp.client.esproj">
      <ReferenceOutputAssembly>false</ReferenceOutputAssembly>
    </ProjectReference>
   ```

5. Right-click the ASP.NET Core project and choose **Reload Project** if that option is available.

6. In *Program.cs*, make sure the following code is present.

   C#

   ```
   app.UseDefaultFiles();
   app.UseStaticFiles();
   ```

```
// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
   app.UseSwagger();
   app.UseSwaggerUI();
}
```

7. To publish, right click the ASP.NET Core project, choose **Publish**, and select options to match your desired publish scenario, such as Azure, publish to a folder, etc.

   The publish process takes more time than it does for just an ASP.NET Core project, since the `npm run build` command gets invoked when publishing. The BuildCommand runs `npm run build` by default.

Troubleshooting

Proxy error

You may see the following error:

Windows Command Prompt

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

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

Verify port

If the weather data doesn't load correctly, you may also need to verify that your ports are correct.

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

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

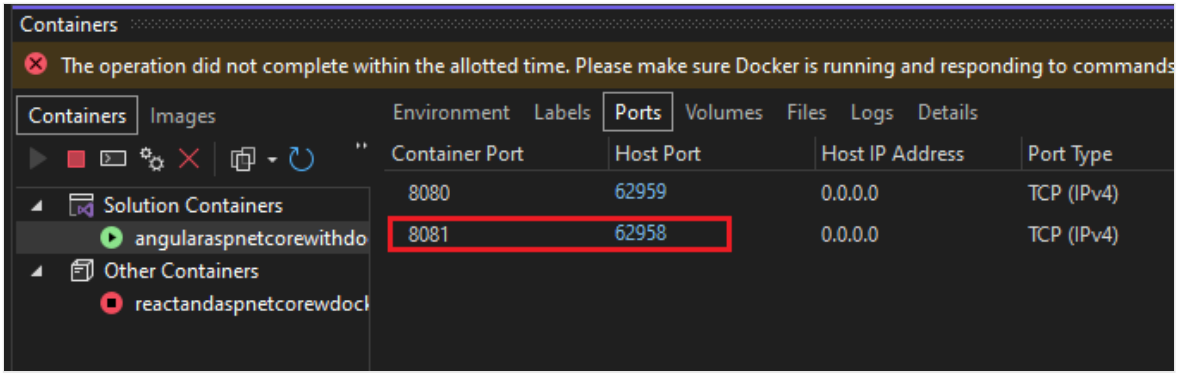
2. Then, go to the `proxy.conf.js` file for your Angular project (look in the *src* folder). Update the target property to match the `applicationUrl` property in *launchSettings.json*. When you update it, that value should look similar to this:

```
JavaScript

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

Docker

If you create the project with Docker support enabled, take the following steps:

1. After the app loads, get the Docker HTTPS port using the Containers window in Visual Studio. Check the **Environment** or **Ports** tab.



2. Open the `proxy.conf.js` file for the Angular project. Update the `target` variable to match the HTTPS port in the Containers window. For example, in the following code:

```
JavaScript

const target = env.ASPNETCORE_HTTPS_PORT ?
`https://localhost:${env.ASPNETCORE_HTTPS_PORT}` :
   env.ASPNETCORE_URLS ? env.ASPNETCORE_URLS.split(';')[0] :
'https://localhost:7209';
```

   change `https://localhost:7209` to `https://localhost:62958`.

3. Restart the app.

Next steps

For more information about SPA applications in ASP.NET Core, see the Angular section under Overview of Single-Page Apps (SPAs). The linked article provides additional

context for project files such as *aspnetcore-https.js* and *proxy.conf.js*, although details of the implementation are different due to project template differences. For example, instead of a ClientApp folder, the Angular files are contained in a separate project.

For MSBuild information specific to the client project, see MSBuild properties for JSPS.

Feedback

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

Article • 05/14/2024

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

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

You can use the method described in this article to create ASP.NET Core Single Page Applications that:

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

> **Note**
>
> This article describes the project creation process using the updated template in Visual Studio 2022 version 17.8, which uses the Vite CLI.

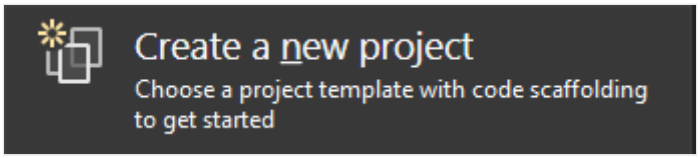
Prerequisites

Make sure to install the following:

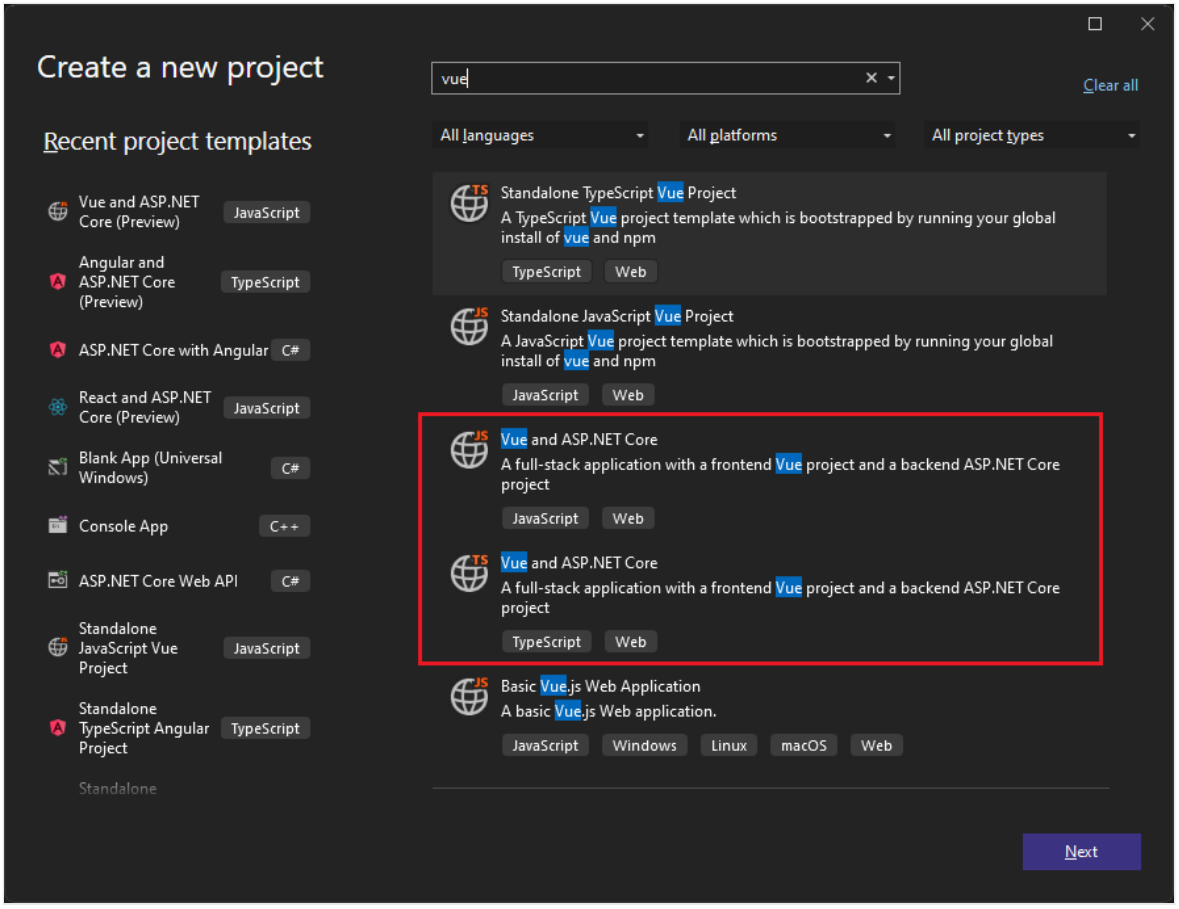
- Visual Studio 2022 version 17.8 or later with the **ASP.NET and web development** workload installed. Go to the Visual Studio downloads page to install it for free. If you need to install the workload and already have Visual Studio, go to **Tools** > **Get Tools and Features...**, which opens the Visual Studio Installer. Choose the **ASP.NET and web development** workload, then choose **Modify**.
- npm (https://www.npmjs.com/), which is included with Node.js.

Create the frontend app

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



2. Search for Vue in the search bar at the top and then select **Vue and ASP.NET Core** with either JavaScript or TypeScript as the selected language.

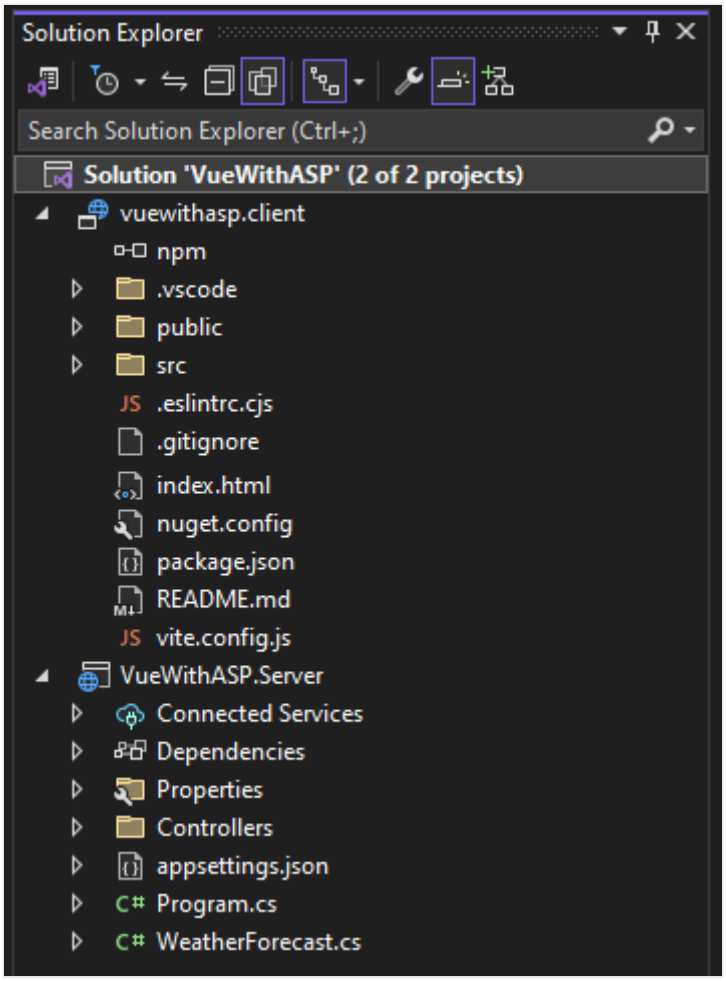


3. Name the project **VueWithASP** and then select **Next**.

   In the Additional Information dialog, make sure that **Configure for HTTPS** is enabled. In most scenarios, leave the other settings at the default values.

4. Select **Create**.

   Solution Explorer shows the following project information:

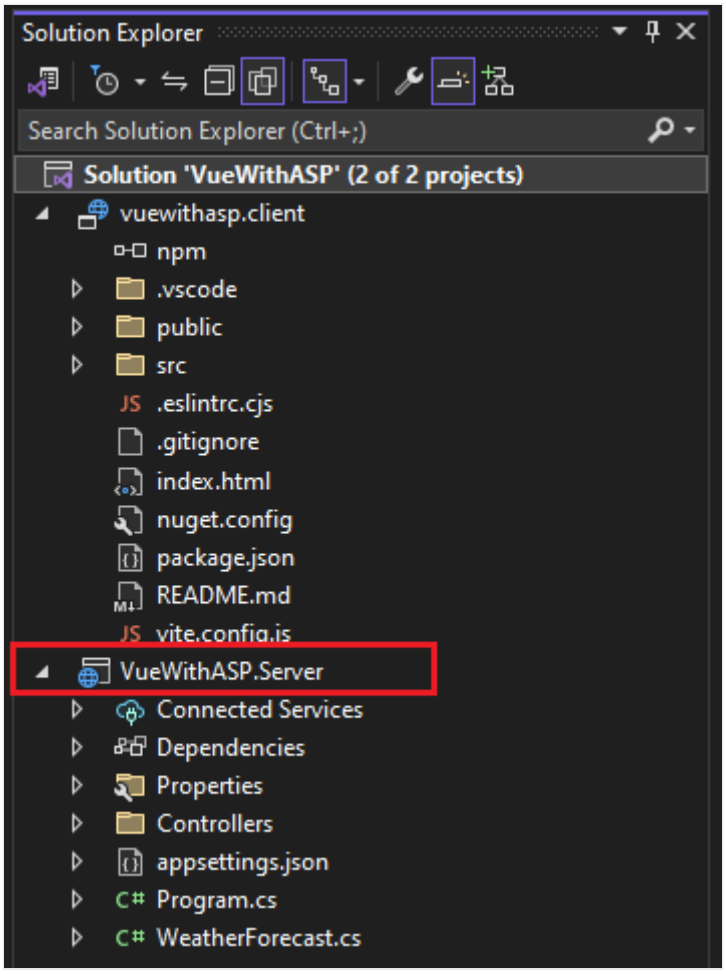


Compared to the standalone Vue template, you see some new and modified files for integration with ASP.NET Core:

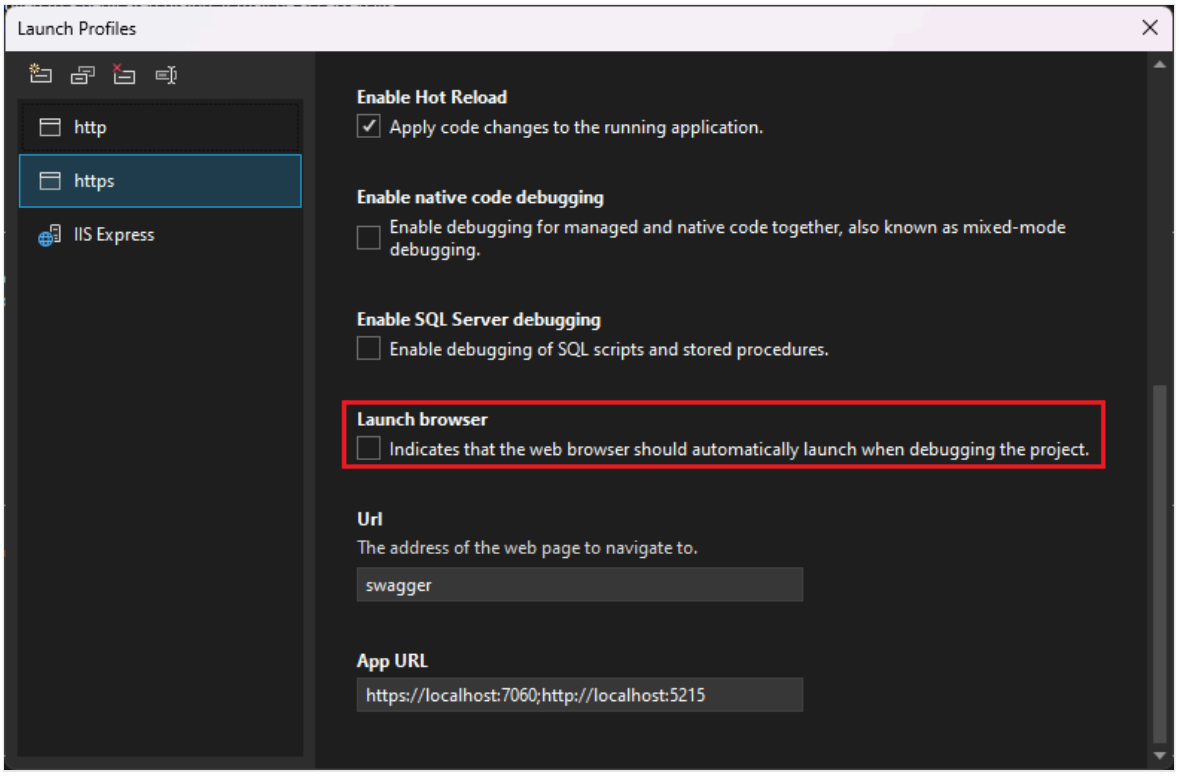
- vite.config.json (modified)
- HelloWorld.vue (modified)
- package.json (modified)

Set the project properties

1. In Solution Explorer, right-click the **VueWithASP.Server** and choose **Properties**.



2. In the Properties page, open the **Debug** tab and select **Open debug launch profiles UI** option. Uncheck the **Launch Browser** option for the **https** profile or the profile named after the ASP.NET Core project, if present.



This value prevents opening the web page with the source weather data.

> ⓘ Note
>
> In Visual Studio, `launch.json` stores the startup settings associated with the **Start** button in the Debug toolbar. Currently, `launch.json` must be located under the `.vscode` folder.

3. Right-click the solution in Solution Explorer and select **Properties**. Verify that the Startup project settings are set to **Multiple projects**, and that the Action for both projects is set to **Start**.

Start the project

Press **F5** or select the **Start** button at the top of the window to start the app. Two command prompts appear:

- The ASP.NET Core API project running
- The Vite CLI showing a message such as `VITE v4.4.9 ready in 780 ms`

> ⓘ Note
>
> Check console output for messages. For example there might be a message to update Node.js.

The Vue app appears and is populated via the API. If you don't see the app, see Troubleshooting.

Publish the project

Starting in Visual Studio 2022 version 17.3, you can publish the integrated solution using the Visual Studio Publish tool.

> ⓘ Note
>
> To use publish, create your JavaScript project using Visual Studio 2022 version 17.3 or later.

1. In Solution Explorer, right-click the **VueWithASP.Server** project and select **Add** > **Project Reference**.

Make sure the **vuewithasp.client** project is selected.

2. Choose **OK**.

3. Right-click the ASP.NET Core project again and select **Edit Project File**.

   This opens the `.csproj` file for the project.

4. In the `.csproj` file, make sure the project reference includes a `<ReferenceOutputAssembly>` element with the value set to `false`.

   This reference should look like the following.

   XML

   ```
    <ProjectReference
   Include="..\vuewithasp.client\vuewithasp.client.esproj">
      <ReferenceOutputAssembly>false</ReferenceOutputAssembly>
    </ProjectReference>
   ```

5. Right-click the ASP.NET Core project and choose **Reload Project** if that option is available.

6. In *Program.cs*, make sure the following code is present.

   C#

   ```
   app.UseDefaultFiles();
   app.UseStaticFiles();

   // Configure the HTTP request pipeline.
   if (app.Environment.IsDevelopment())
   {
      app.UseSwagger();
      app.UseSwaggerUI();
   }
   ```

7. To publish, right click the ASP.NET Core project, choose **Publish**, and select options to match your desired publish scenario, such as Azure, publish to a folder, etc.

   The publish process takes more time than it does for just an ASP.NET Core project, since the `npm run build` command gets invoked when publishing. The BuildCommand runs `npm run build` by default.

Troubleshooting

Proxy error

You may see the following error:

```
[HPM] Error occurred while trying to proxy request /weatherforecast from
localhost:4200 to https://localhost:5173 (ECONNREFUSED)
(https://nodejs.org/api/errors.html#errors_common_system_errors)
```

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

Otherwise, if the port is in use, try incrementing the port number by **1** in `launchSettings.json` and *vite.config.js*.

Privacy error

You may see the following certificate error:

```
Your connection isn't private
```

Try deleting the Vue certificates from *%appdata%\local\asp.net\https* or *%appdata%\roaming\asp.net\https*, and then retry.

Verify ports

If the weather data doesn't load correctly, you may also need to verify that your ports are correct.

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

   If there are multiple `applicationUrl` properties, look for one using an `https` endpoint. It should look similar to `https://localhost:7142`.

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

JavaScript

```
target: 'https://localhost:7142/',
```

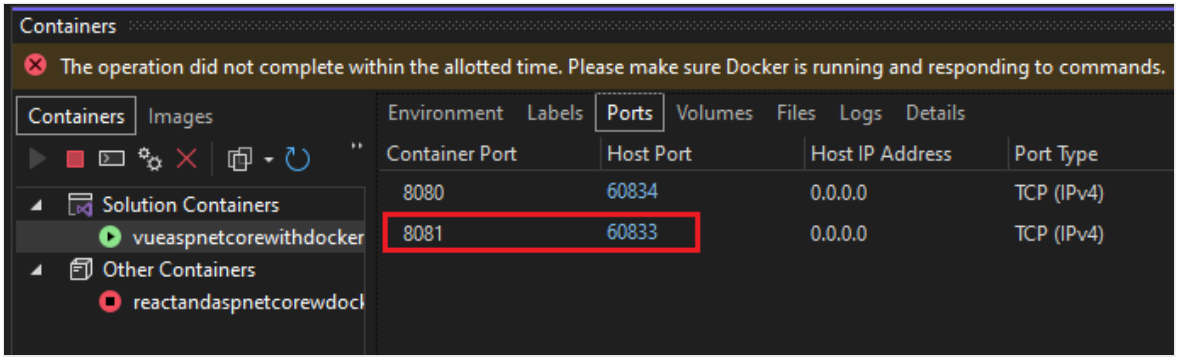
Outdated version of Vue

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

Docker

If you create the project with Docker support enabled, take the following steps:

1. After the app loads, get the Docker HTTPS port using the Containers window in Visual Studio. Check the **Environment** or **Ports** tab.



2. Open the `vite.config.js` file for the Vue project. Update the `target` variable to match the HTTPS port in the Containers window. For example, in the following code:

   JavaScript

   ```
   const target = env.ASPNETCORE_HTTPS_PORT ?
   `https://localhost:${env.ASPNETCORE_HTTPS_PORT}` :
      env.ASPNETCORE_URLS ? env.ASPNETCORE_URLS.split(';')[0] :
   'https://localhost:7163';
   ```

   change `https://localhost:7163` to `https://localhost:60833`.

3. Restart the app.

If you are using a Docker configuration created in older versions of Visual Studio, the backend may start up using the Docker profile and not listen on the configured port

5173. To resolve:

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

JSON

```
"httpPort": 5175,
"sslPort": 5173
```

Next steps

For more information about SPA applications in ASP.NET Core, see Developing Single Page Apps. The linked article provides additional context for project files such as *aspnetcore-https.js*, although details of the implementation are different due to differences between the project templates and the Vue.js framework vs. other frameworks. For example, instead of a ClientApp folder, the Vue files are contained in a separate project.

For MSBuild information specific to the client project, see MSBuild properties for JSPS.

Feedback

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

Article • 10/30/2023

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

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

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

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, 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...** to open the Visual Studio Installer. Choose the **ASP.NET and web development** workload, then select **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. If the start window isn't open, choose **File** > **Start Window**.
2. On the start window, choose **Create a new project**.
3. On the **Create a new project** window, enter *web app* in the search box. Next, choose **C#** as the language.

   After you apply the language filter, choose **ASP.NET Core Web Application (Model-View-Controller)**, and then select **Next**.

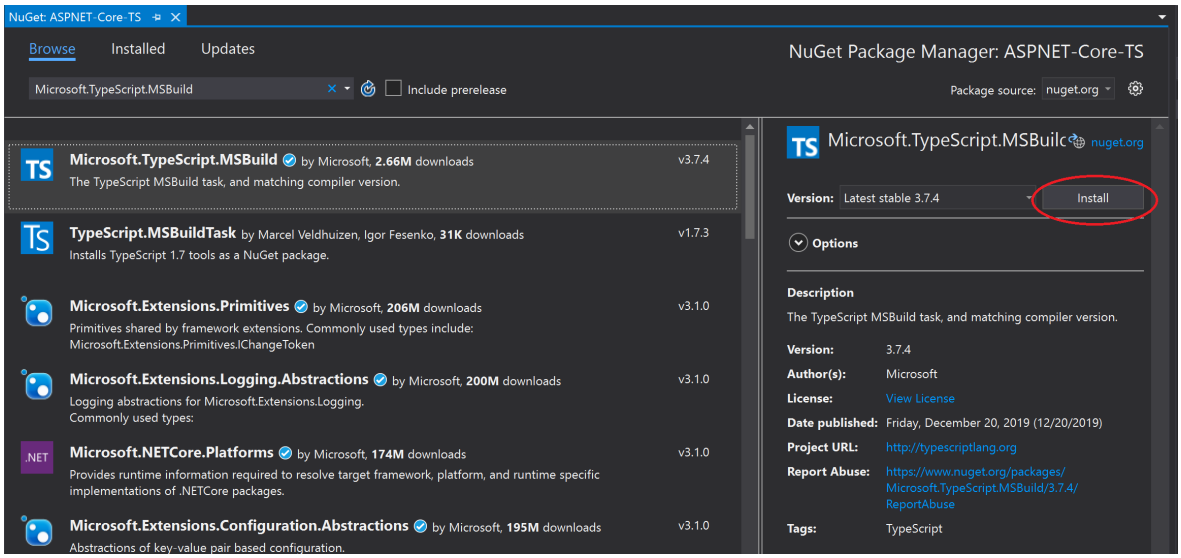
   > **Note**
   >
   > If you don't see the **ASP.NET Core Web Application** project template, you must add the **ASP.NET and web development** workload. For detailed instructions, see the **Prerequisites**.

4. In the **Configure your new project** window, enter a name for your project in the **Project name** box. Then, select **Next**.
5. In the **Additional information** window, ensure **.NET 8.0** is selected in the **Framework** dropdown menu, and then select **Create**.

Visual Studio opens your new project.

Add some code

1. In Solution Explorer (right pane), right-click the project node and select **Manage NuGet Packages for Solutions**.
2. In the **Browse** tab, search for **Microsoft.TypeScript.MSBuild**.
3. Select **Install** to install the package.



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

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

   If you don't see all the item templates, select **Show All Templates**, and then choose the item template.

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

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

   JSON

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

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

This configuration provides a basic introduction to using TypeScript. In other scenarios, such as when using gulp or Webpack, you might want a different intermediate location for the transpiled JavaScript files instead of *wwwroot/js*. The location depends on your tools and configuration preferences.

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

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

   If you don't see all the item templates, select **Show All Templates**, and then choose the item template.

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

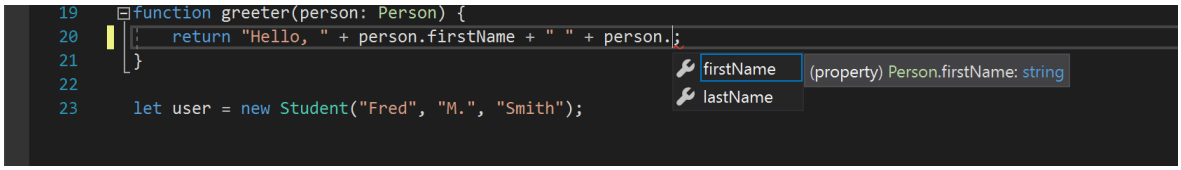
8. Open *app.ts* and add the following TypeScript code.

   ts

   ```ts
   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 try this feature, remove `.lastName` from the `greeter` function, reenter the period (.), and notice the IntelliSense updates.



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

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

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

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

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

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

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

13. Select **File** > **Save All** (**Ctrl** + **Shift** + **S**) to save your changes.

Build the application

1. Select **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 to see two new files: *app.js* and the source map file, *app.js.map*. The TypeScript compiler generates these files.

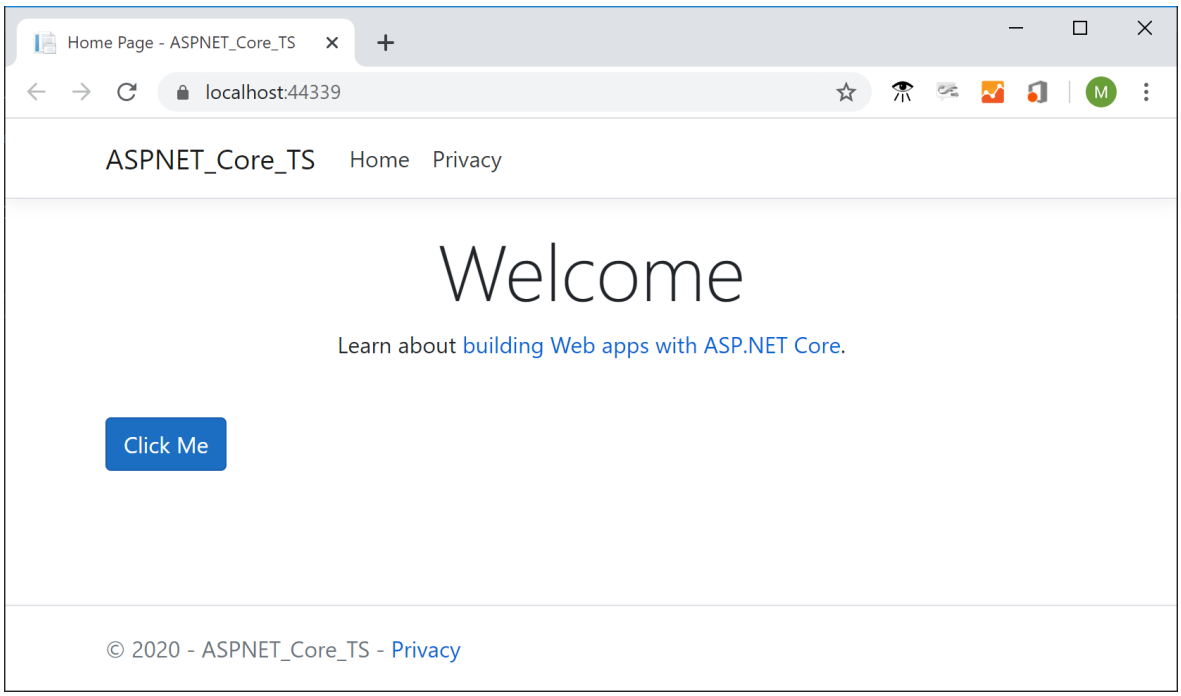
   Source map files are required for debugging.

Run the application

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

   The app opens in a browser.

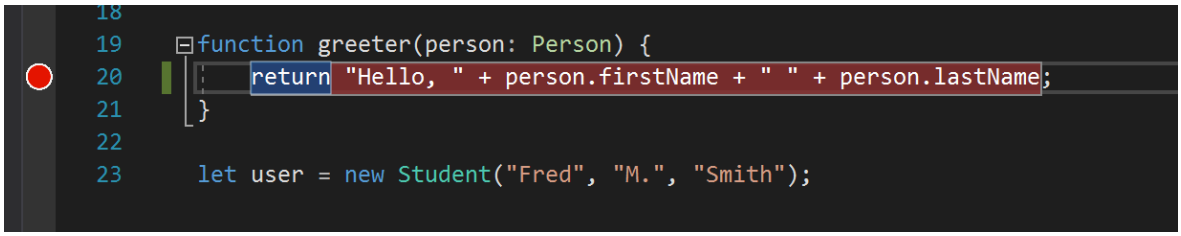
   In the browser window, you see the **Welcome** heading and the **Click Me** button.



2. Select 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 might need to respond to a message to enable script debugging.

> ⓘ **Note**
>
> Chrome or Edge is required for client-side script debugging.

3. When the page loads, press **Click Me**.

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 the instructions in npm package management to add a `package.json` file to your project. This task 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 code in your *package.json* file.

   ```JSON
   "devDependencies": {
     "@types/jquery": "3.5.1"
   }
   ```

   This code 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're using a different template, you might need to include the jquery npm package as well.

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

   > **Note**
   >
   > In some scenarios, Solution Explorer might indicate that an npm package is out of sync with *package.json* due to a known issue described **here**. For example, the package might 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**.

If you don't see all the item templates, choose **Show All Templates**, and then choose the item template.

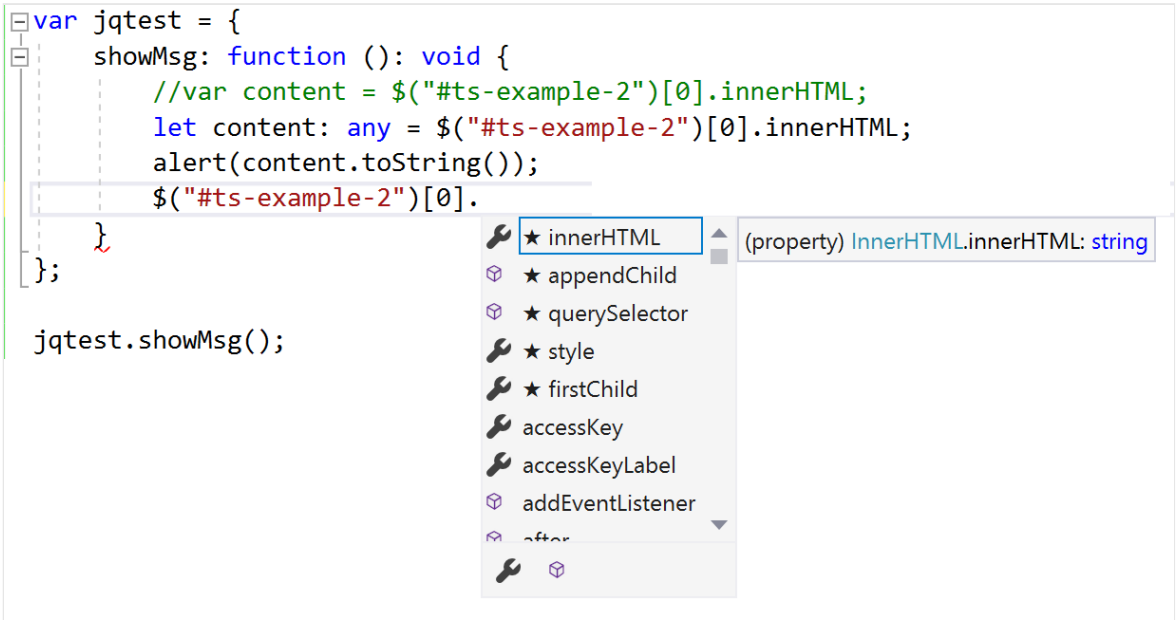
5. Choose **TypeScript File**, type *library.ts*, and choose **Add**.

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

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

jqtest.showMsg();
```

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

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



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

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

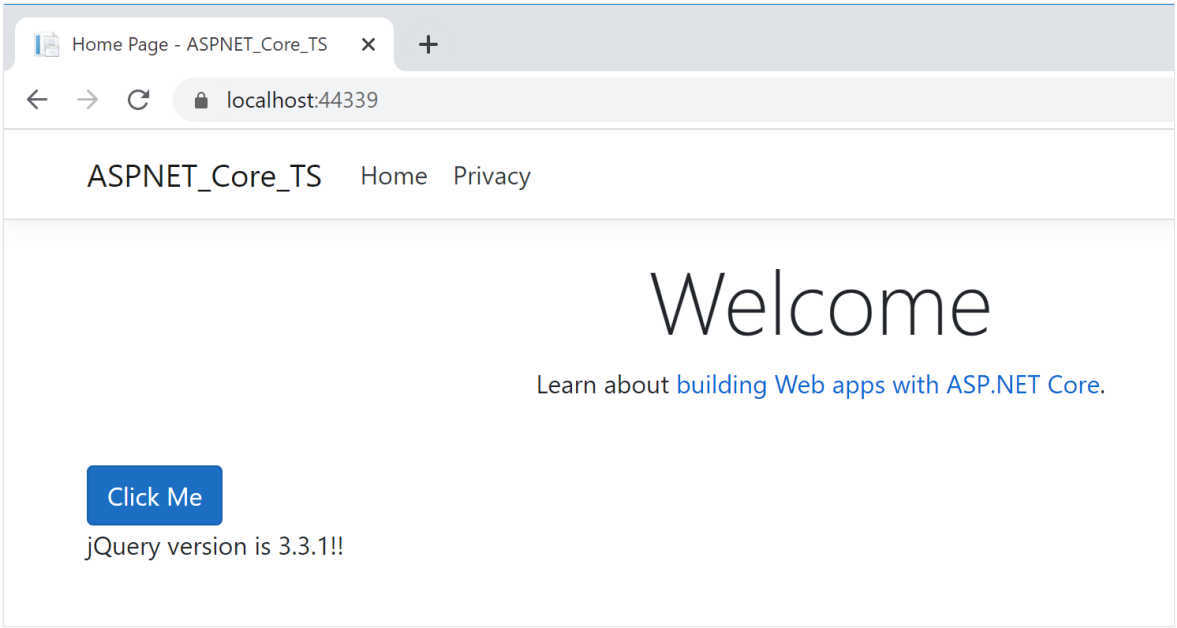
8. In *Index.cshtml*, add the following HTML to the end of the file.

```HTML
<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.

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



Next steps

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

ASP.NET Core and TypeScript

Angular language service extension

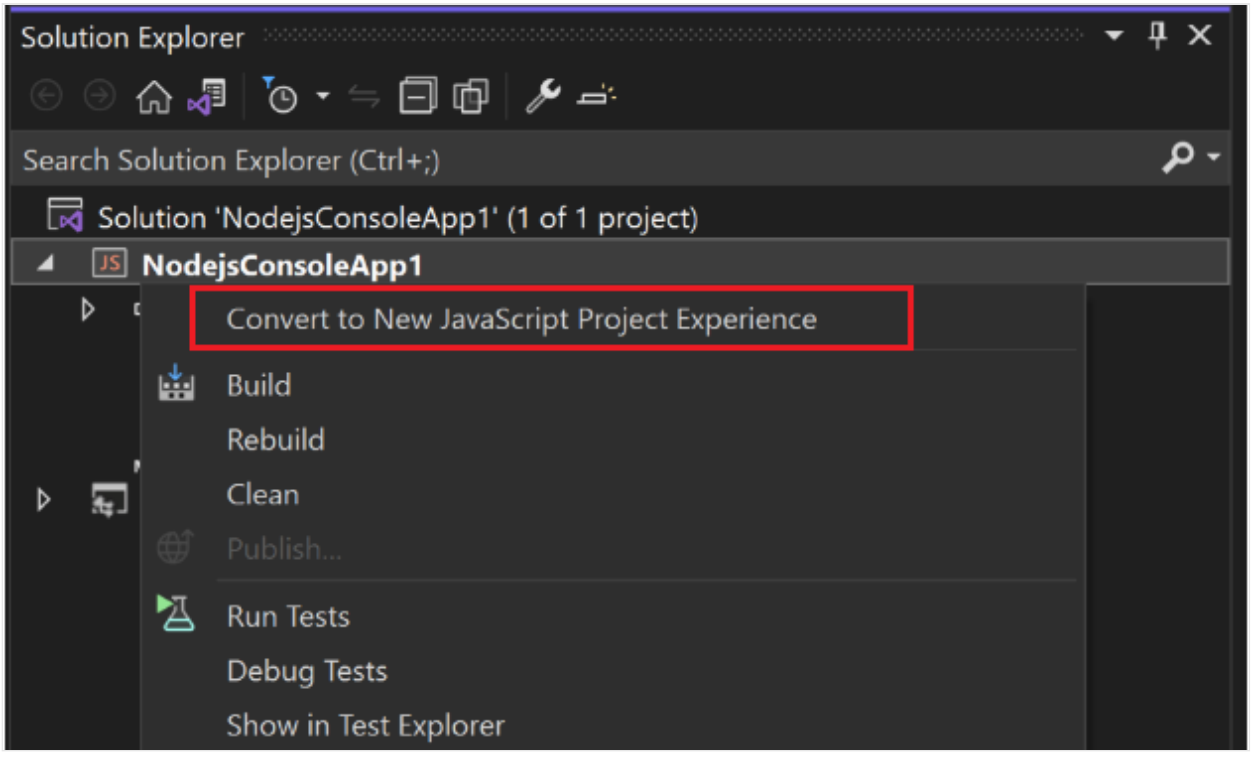
Migrate Node.js projects in Visual Studio

Article • 01/12/2024

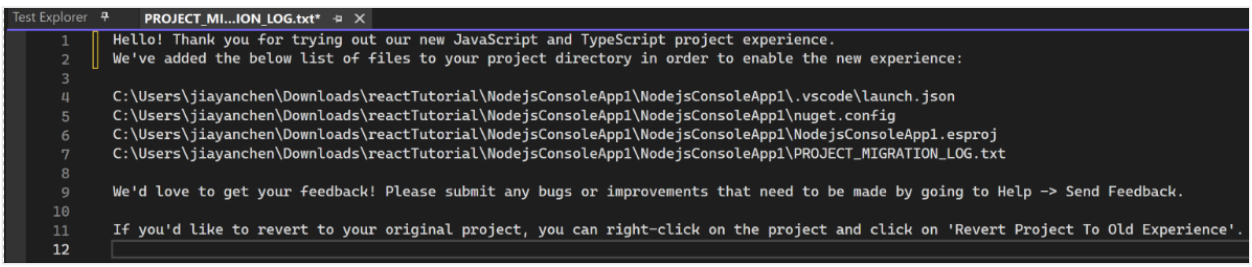
Starting in Visual Studio 2022 version 17.7 Preview 1, you can convert existing projects based on the older Node.js project system (*.njsproj*) to the new JavaScript project system (*.esproj*). By migrating the project, you can benefit from project system updates such as npm dependency management, unit testing support, and launch config settings.

To migrate from a Node.js project to a JavaScript project:

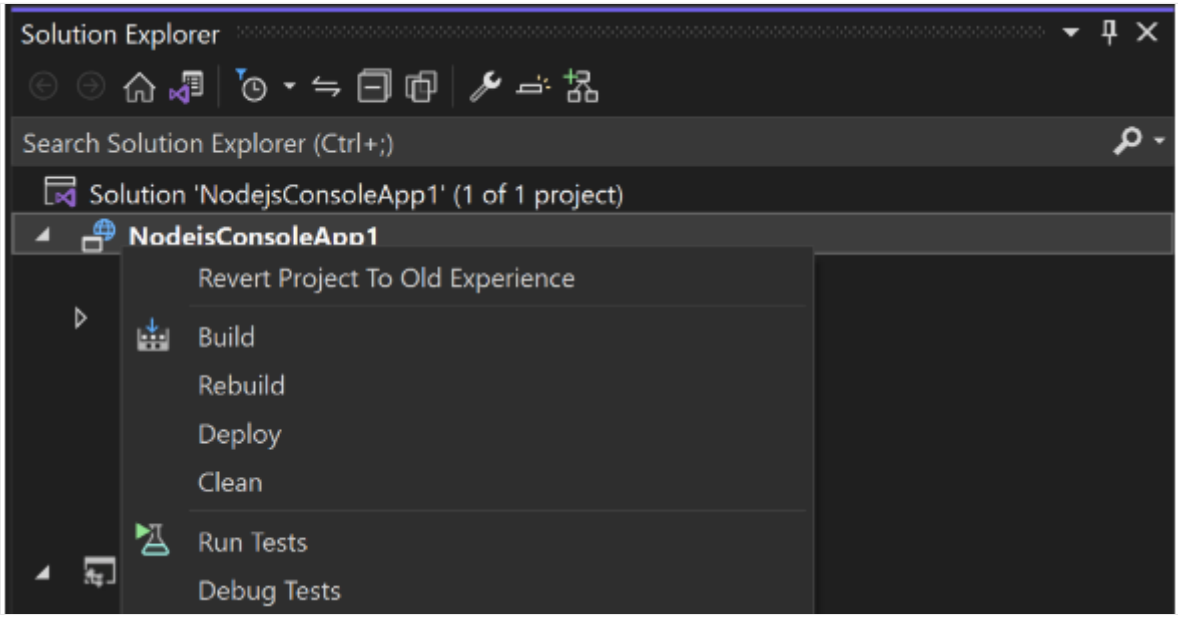
Right-click the project node for your Node.js project. You should see one of two options: **Convert to New JavaScript Project Experience** or **Convert to New TypeScript Project Experience**. Select the available option to migrate your project to the new project system.



After you choose **Convert**, a conversion log text file gets created and then it opens. The log file details the steps that occurred during the migration.



If anything goes wrong during the migration, you can choose the **Revert Project to Old Experience** option so that the conversion will be reverted. If you encounter any problem during this process, please the Report a problem feature in Visual Studio.



Learn to use the code editor for JavaScript

Article • 11/07/2023

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

In this short introduction to the code editor in Visual Studio, we'll look at some of the ways that Visual Studio makes writing, navigating, and understanding code easier.

> **Tip**
>
> If you haven't already installed Visual Studio, go to the **Visual Studio downloads** page to install it for free. For more information in getting the language service for TypeScript, see **TypeScript support**.

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

Add a new project file

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

1. With your project open in Visual Studio, right-click on a folder or your project node in Solution Explorer (right pane), and choose **Add** > **New Item**.

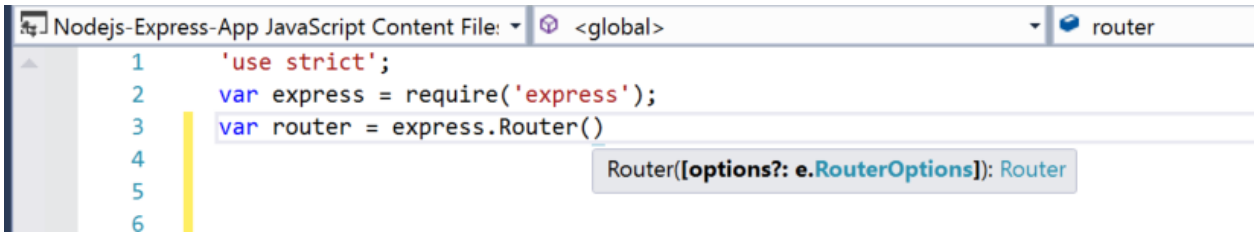
   If you don't see all the item templates, choose **Show All Templates**, and then choose the item template.

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**.

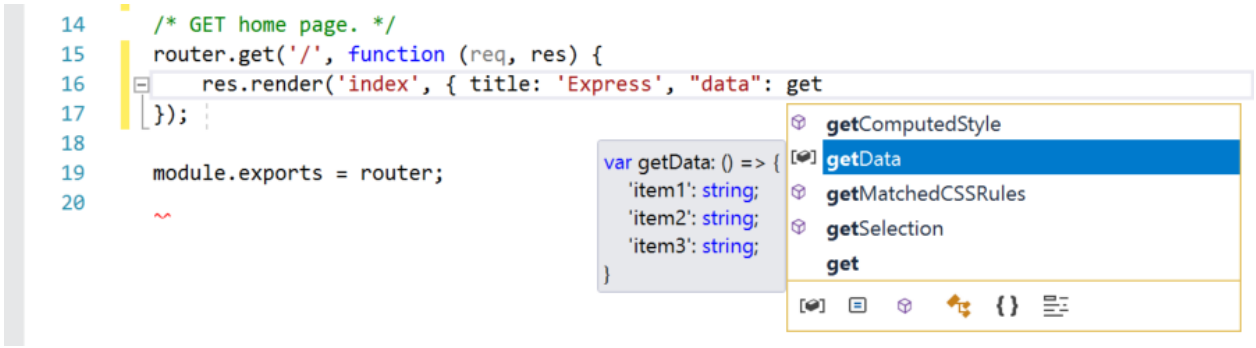
   The new file gets added to your project and it opens in the editor.

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.

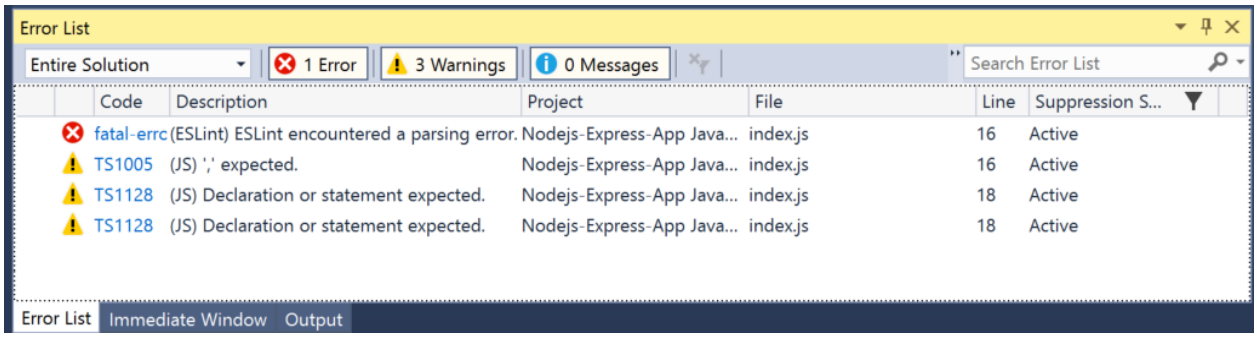
In the following code, you see green syntax highlighting (green squiggles) on the expression. Hover over the syntax highlighting.

```
27    /* GET home page. */
28    router.get('/', function (req, res) {
29        res.render('index', { title: 'Express' "data" : getData() });
30    });
31
32    module.exports = router;
33
```

```
(property) "data": {
    'item1': string;
    'item2': string;
    'item3': string;
}

(JS) ',' expected.
```

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

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



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

For additional information on linting, see Linting.

Comment out code

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

Select one or more lines of code in the editor and then choose the **Comment out the selected lines** button on the toolbar. If you prefer to use the keyboard, press **Ctrl**+**K**, **Ctrl**+**C**.

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

Collapse code blocks

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

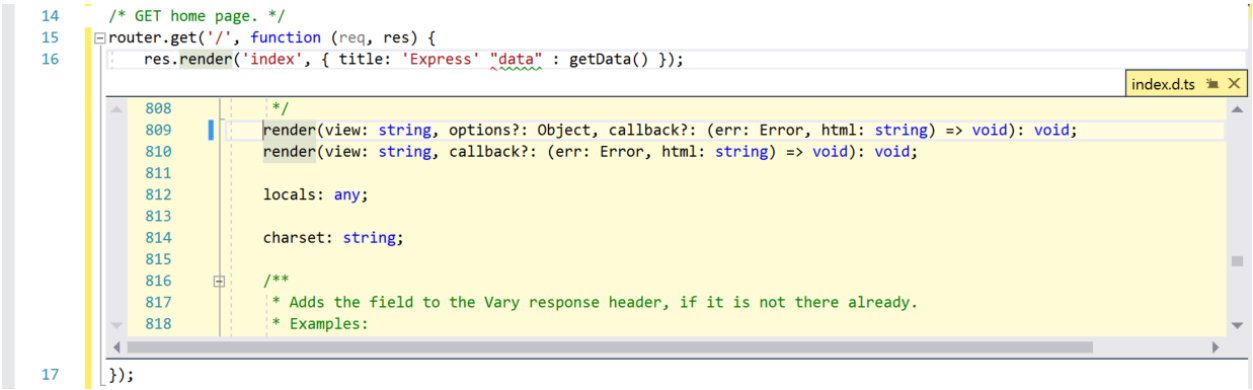
The code block collapses to just the first line, followed by an ellipsis (`...`). To expand the code block again, click the same gray box that now has a plus sign in it, or press **Ctrl**+**M**, **Ctrl**+**M** again. This feature is called Outlining and is especially useful when you're collapsing long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

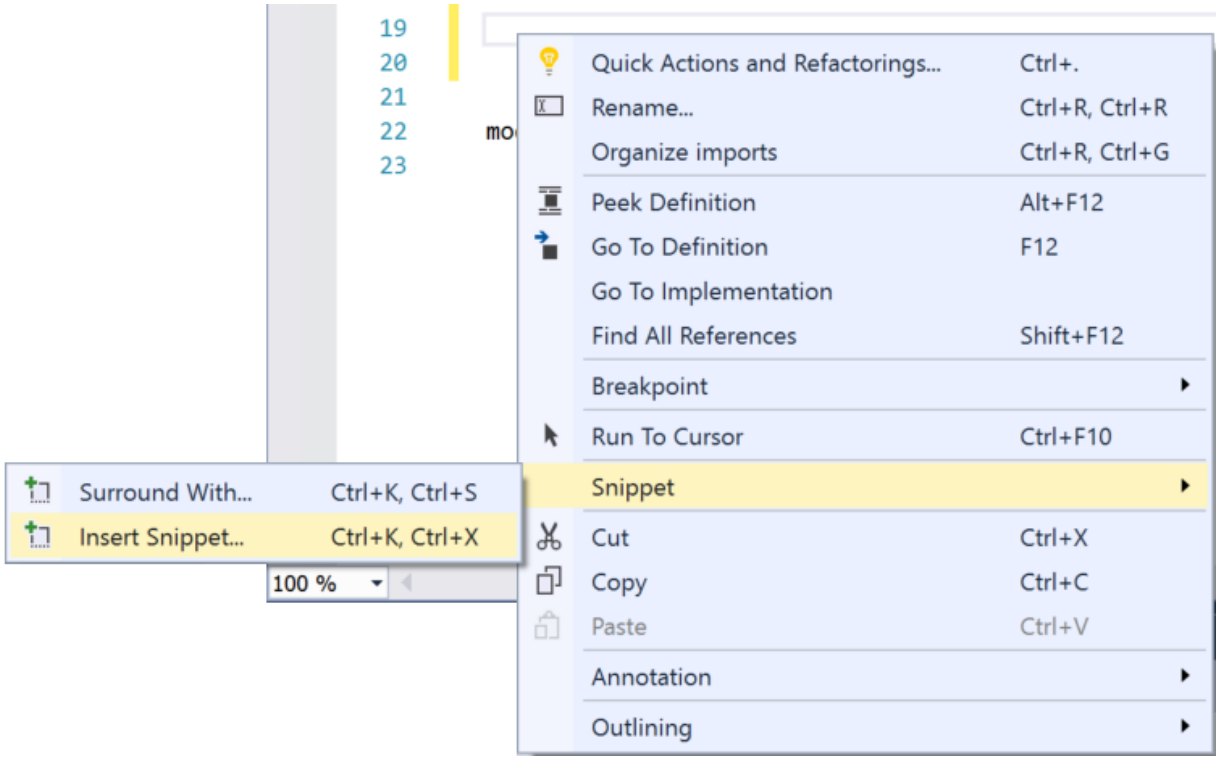


Close the peeked definition window by choosing the small box with an "x" at the top right of the pop-up window.

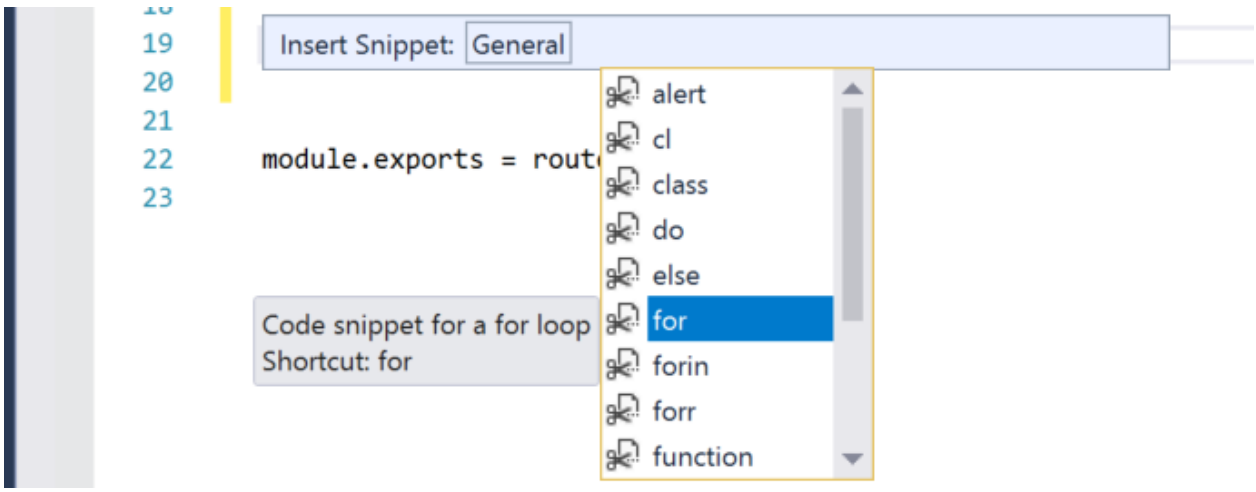
Use code snippets

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:

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

}
```

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

See also

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

JavaScript IntelliSense

Article • 02/22/2023

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

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

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

JavaScript IntelliSense displays information on parameter and member lists. This information is provided by the TypeScript language service, which uses static analysis behind the scenes to better understand your code.

TypeScript uses several sources to build up this information:

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

IntelliSense based on type inference

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

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

JavaScript

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

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

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

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

Additionally, there is special inference for the following:

- "ES3-style" classes, specified using a constructor function and assignments to the prototype property.
- CommonJS-style module patterns, specified as property assignments on the `exports` object, or assignments to the `module.exports` property.

JavaScript

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

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

IntelliSense based on JSDoc

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

JavaScript

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

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

JavaScript

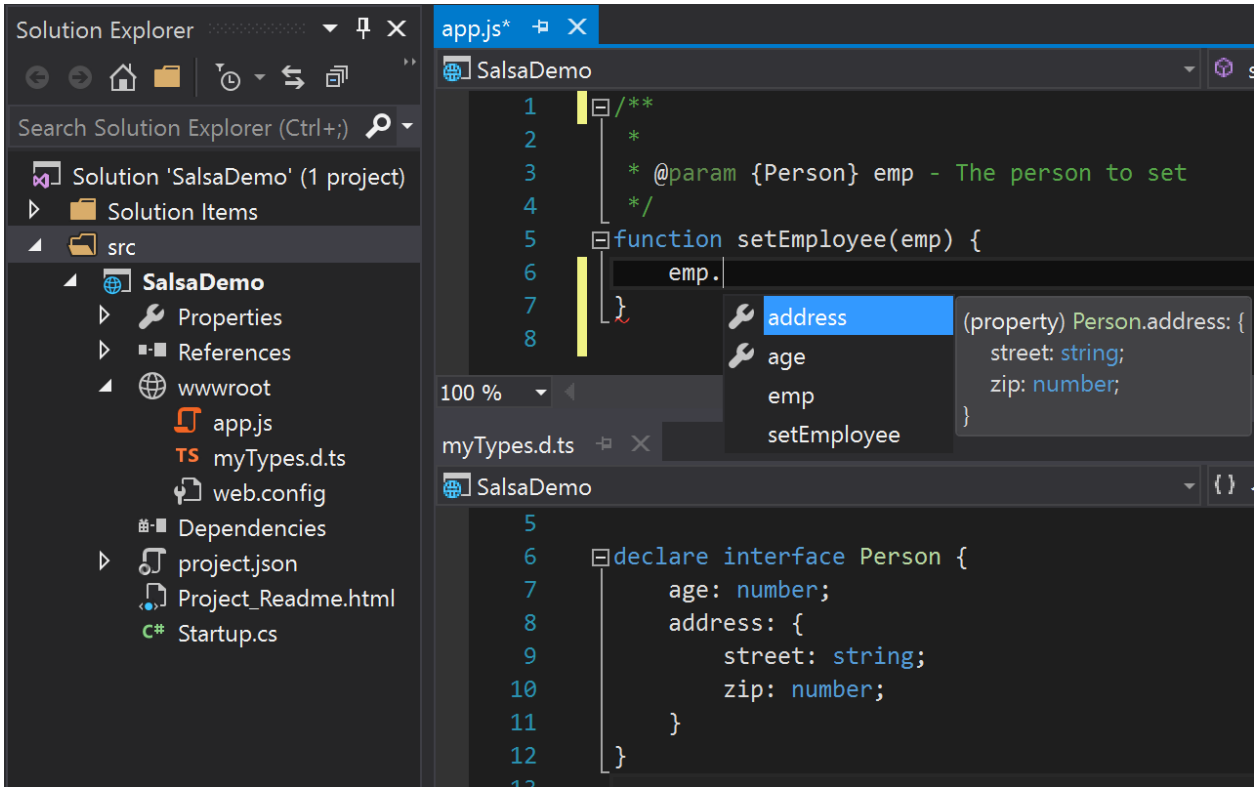
```
/**
 * @param {string} param1 - The first argument to this function
 */
function Foo(param1) {
    this.prop = param1; // "param1" (and thus "this.prop") are now of type
"string".
}
```

See the JsDoc information in Type Checking JavaScript Files for the JsDoc annotations currently supported.

IntelliSense based on TypeScript declaration files

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

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



Automatic acquisition of type definitions

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

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

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

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

See also

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

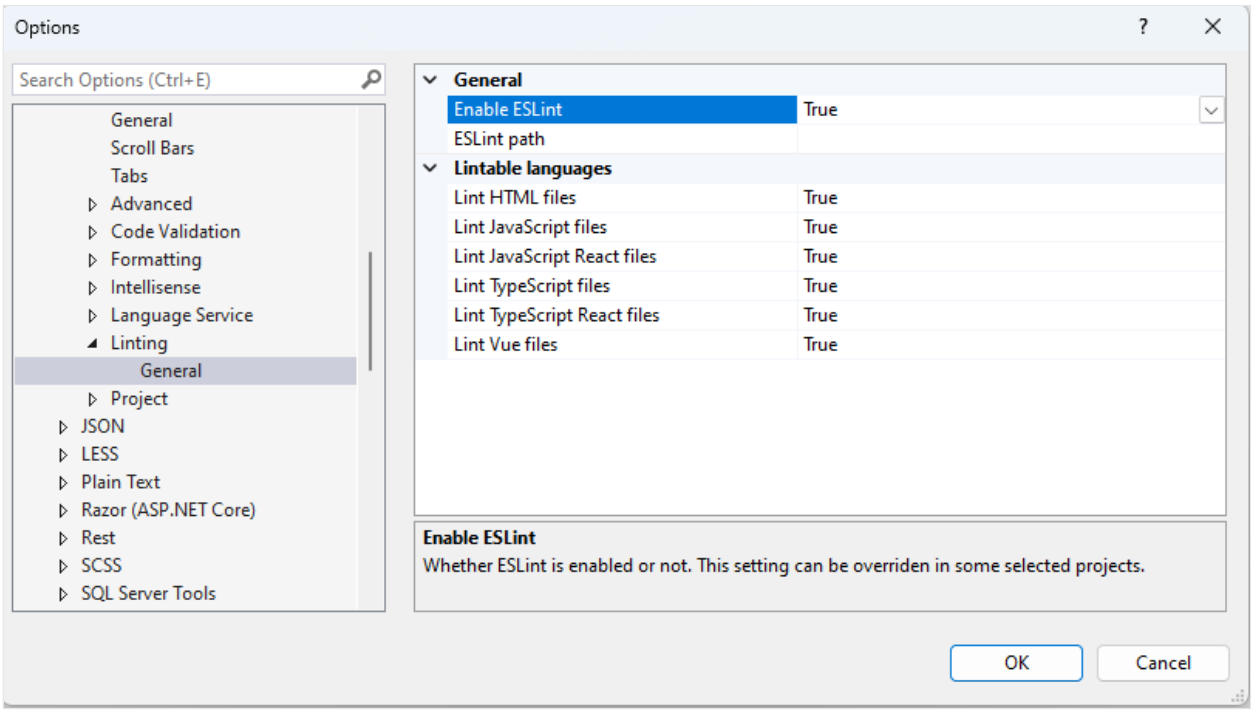
Linting JavaScript in Visual Studio

Article • 06/29/2023

Linting JavaScript and TypeScript in Visual Studio is powered by ESLint. If you're new to ESLint, you can begin by checking their documentation.

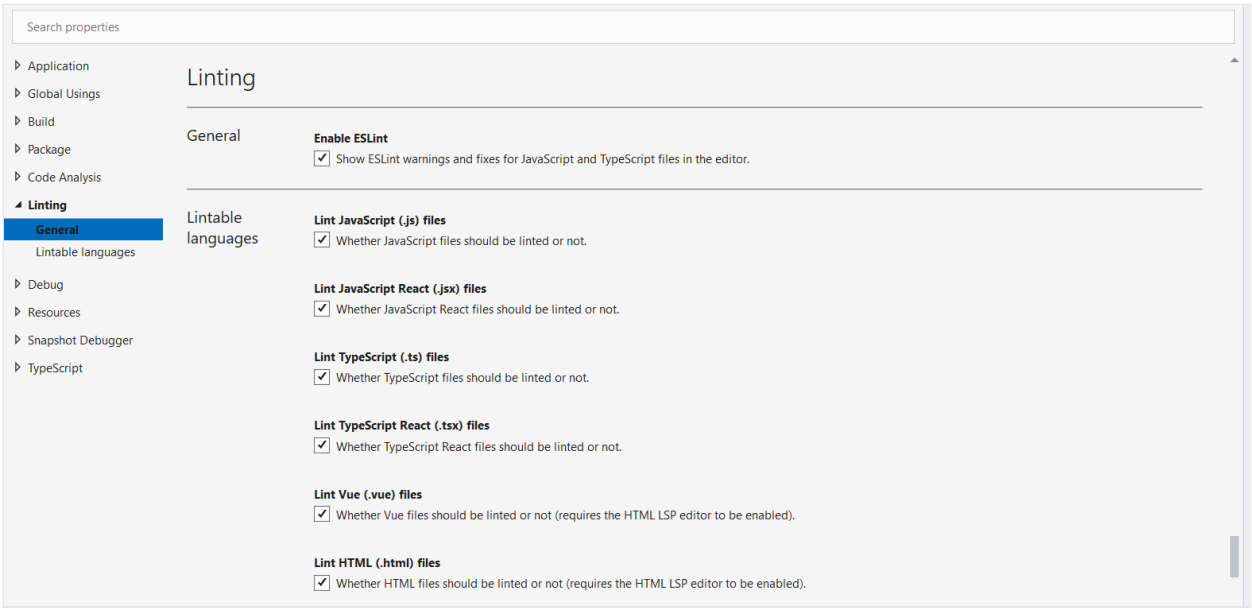
Enabling linting support

To enable linting support in Visual Studio 2022 or later versions, enable the **Enable ESLint** setting in **Tools** > **Options** > **Text Editor** > **JavaScript/TypeScript** > **Linting**.



In the options page, you can also modify the set of files that you want to lint. By default, all file extensions that can be linted (*.js*, *.jsx*, *.ts*, *.tsx*, *.vue*, *.html*) will be linted. The HTML LSP-based editor must be enabled for linting Vue and HTML files. The respective setting can be found in **Tools** > **Options** > **Environment** > **Preview Features**.

You can override these options in some project types, like the standalone React project templates. In these projects, you can override the settings from the **Tools** > **Options** page using project properties:



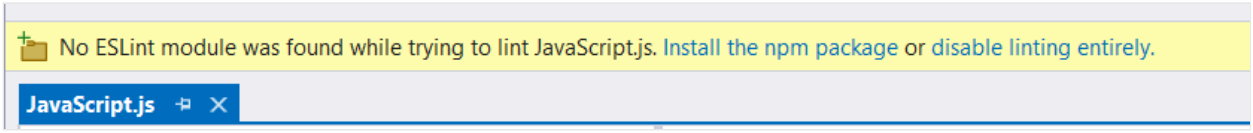
Installing ESLint dependencies

Once linting is enabled, the necessary dependencies need to be installed. Dependencies include the ESLint npm package and other plugins applicable to your project. This package can be installed locally in each project where you want to enable linting, or you can install it globally using `npm install -g eslint`. However, a global installation isn't recommended because plugins and shareable configs always need to be installed locally.

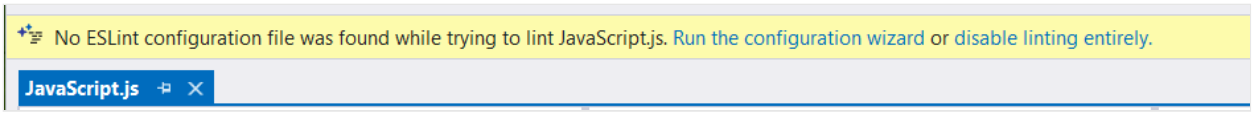
Starting in Visual Studio 2022 version 17.7 Preview 2, you can also use the **ESLint Path** setting in **Tools** > **Options** > **Text Editor** > **JavaScript/TypeScript** > **Linting** to specify a directory from which to load ESLint. This is useful when ESLint is installed globally, where you might set the corresponding path to *C:\Program Files\nodejs\node_modules*.

Depending on the files you want to lint, other ESLint plugins may be needed. For example, you may need TypeScript ESLint, which enables ESLint to run on TypeScript code and includes rules that are specific to the extra type information.

When ESLint is enabled but the ESLint `npm` package isn't found, a gold bar is displayed that allows you to install ESLint as a local `npm` development dependency.

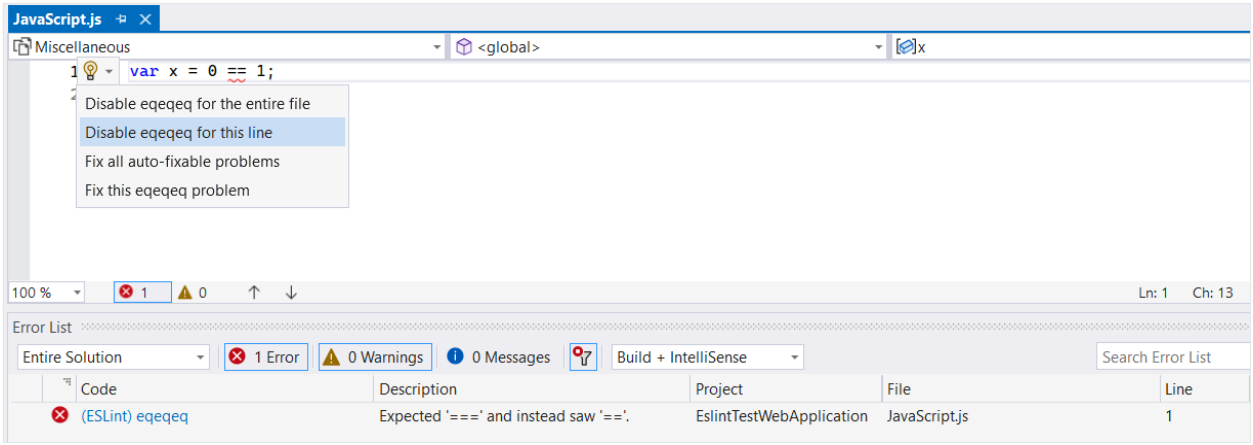


Similarly, when an *.eslintrc* file isn't found, a gold bar is displayed to run a configuration wizard that will install the plugins applicable to the current project.

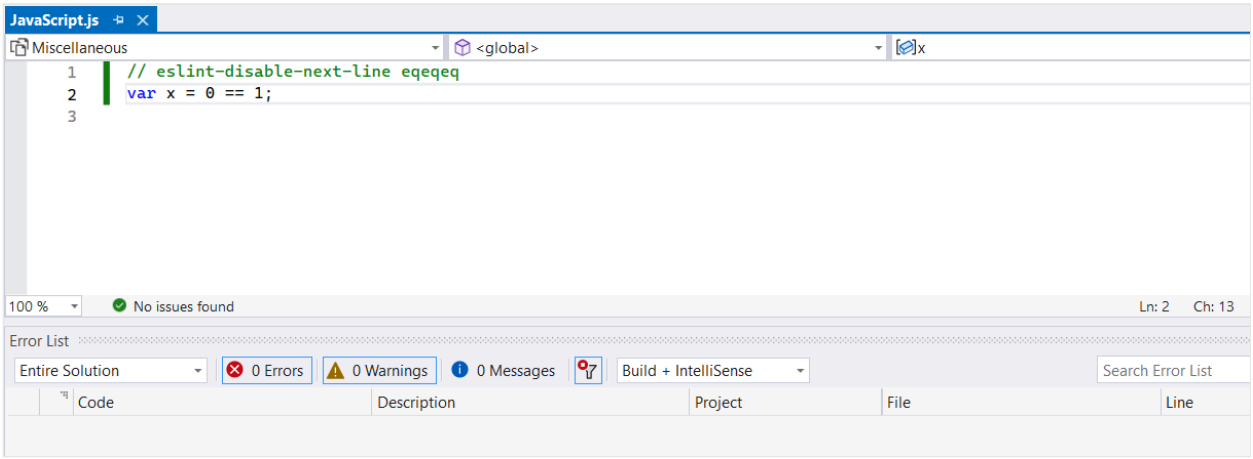


Disabling linting rules and auto-fixes

You can disable linting errors on a specific line or file. You can disable the errors by using the Quick Actions lightbulb menu:



The following illustration shows the result if you disable a linting error for the selected line of code.



In addition, auto-fix code actions allow you to apply an auto-fix to address the respective linting error.

Troubleshooting

You can open the **ESLint Language Extension** pane in the Output window to see any error messages or other logs that might explain the problem.

Compile TypeScript code (Node.js)

Article • 01/12/2024

Use the TypeScript npm package to add TypeScript support to projects based on the JavaScript Project System (JSPS), or *.esproj*. Starting in Visual Studio 2019, it is recommended that you use the npm package instead of the TypeScript SDK. The TypeScript npm package provides greater portability across different platforms and environments.

> **Important**
>
> For ASP.NET Core projects, use the **NuGet package** instead of npm to add TypeScript support.

Add TypeScript support using npm

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

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

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

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

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

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

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

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

   If you don't see all the item templates, choose **Show All Templates**, and then choose the item template.

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

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

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

   JSON

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

   In this example:

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

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

Build the application

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

   TypeScript

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

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

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

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

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

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

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

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

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

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

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

Source map files are required for debugging.

Run the application

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

Automate build tasks

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

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

Related content

Properties, React, Angular, Vue

Compile TypeScript code (ASP.NET Core)

Article • 10/23/2023

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

Use the TypeScript NuGet package to add TypeScript support to your ASP.NET Core projects. Starting in Visual Studio 2019, it is recommended that you use the NuGet package instead of the TypeScript SDK. The TypeScript NuGet package provides greater portability across different platforms and environments.

For ASP.NET Core projects, one common usage for the NuGet package is to compile TypeScript using the .NET Core CLI. In .NET scenarios, the NuGet package is the preferred option, and it's the only way to enable TypeScript compilation using .NET Core CLI commands such as `dotnet build` and `dotnet publish`. Also, for MSBuild integration with ASP.NET Core and TypeScript, choose the NuGet package.

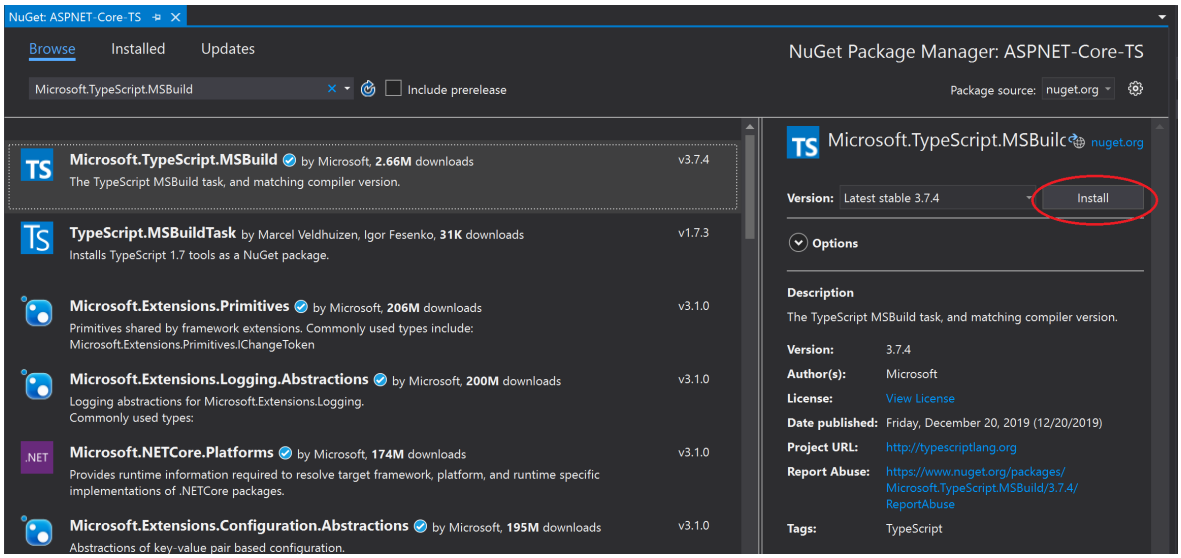
> **Important**
>
> For projects based on the JavaScript Project System (JSPS), or *.esproj* projects, use the **npm package** instead of NuGet to add TypeScript support.

Add TypeScript support with NuGet

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

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

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



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

XML

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

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

   If you don't see all the item templates, choose **Show All Templates**, and then choose the item template.

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

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

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

JSON

```
{
  "compilerOptions": {
    "noImplicitAny": false,
    "noEmitOnError": true,
    "removeComments": false,
    "sourceMap": true,
    "target": "es5",
    "outDir": "wwwroot/js"
```

```
  },
  "include": [
    "scripts/**/*"
  ]
}
```

In this example:

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

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

Build the application

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

   TypeScript

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

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

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

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

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

   Source map files are required for debugging.

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

   JSON

   ```
   {
      "compileOnSave":  true,
   ```

```
    "compilerOptions": {
    }
}
```

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

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

Run the application

Press **F5** or select the Start button at the top of the window.

NuGet package structure details

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

- *build* folder

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

  1. *Microsoft.TypeScript.MSBuild.targets*

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

  2. *Microsoft.TypeScript.MSBuild.props*

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

- *tools* folder

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

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

Additionally, the folder contains three subfolders:

1. *net45*

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

2. *netstandard1.3*

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

3. *tsc*

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

   > **Note**
   >
   > If Visual Studio is installed, then the NuGet package automatically picks up the version of *node.exe* bundled with Visual Studio. Otherwise, Node.js must be installed on the machine.

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

Remove default imports

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

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

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

   The project file opens.

3. Remove references to `Microsoft.TypeScript.Default.props` and `Microsoft.TypeScript.targets`.

   The imports to remove look similar to the following:

```xml
<Import

Project="$(MSBuildExtensionsPath32)\Microsoft\VisualStudio\v$(VisualStu
dioVersion)\TypeScript\Microsoft.TypeScript.Default.props"

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

<Import

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

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

Manage npm packages in Visual Studio

Article • 01/31/2024

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

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

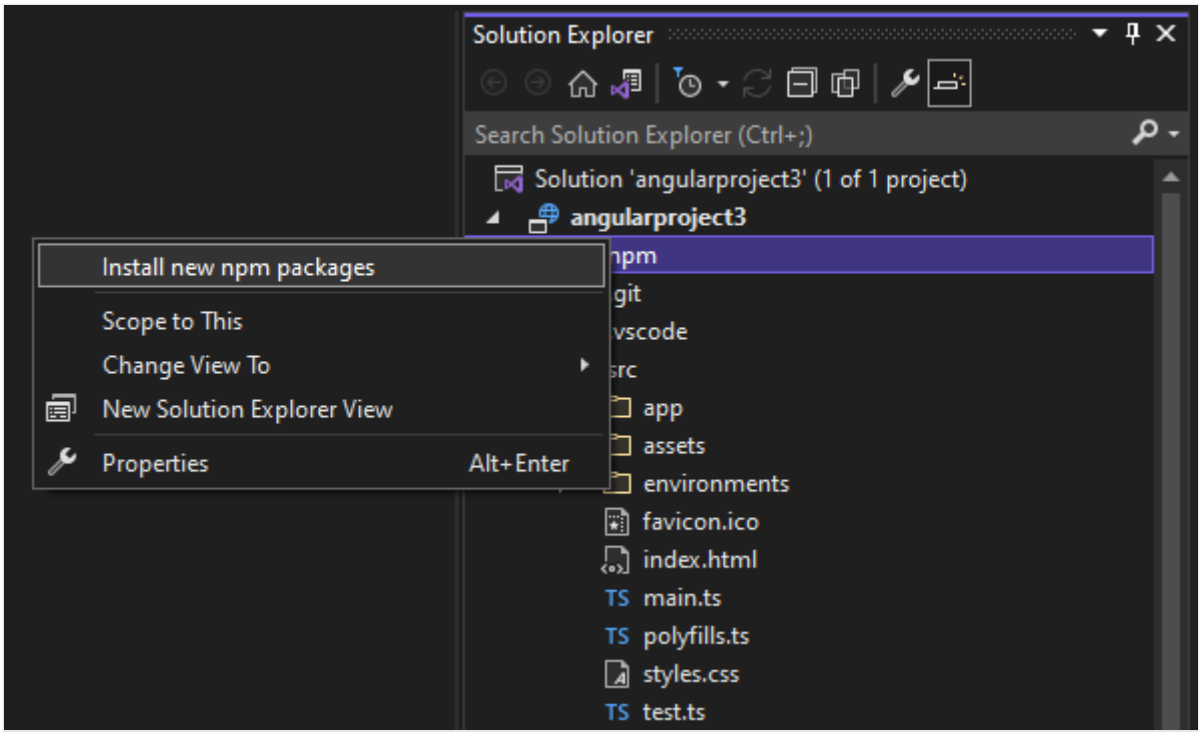
- CLI-based projects (.esproj)
- ASP.NET Core
- Open folder (Node.js)

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

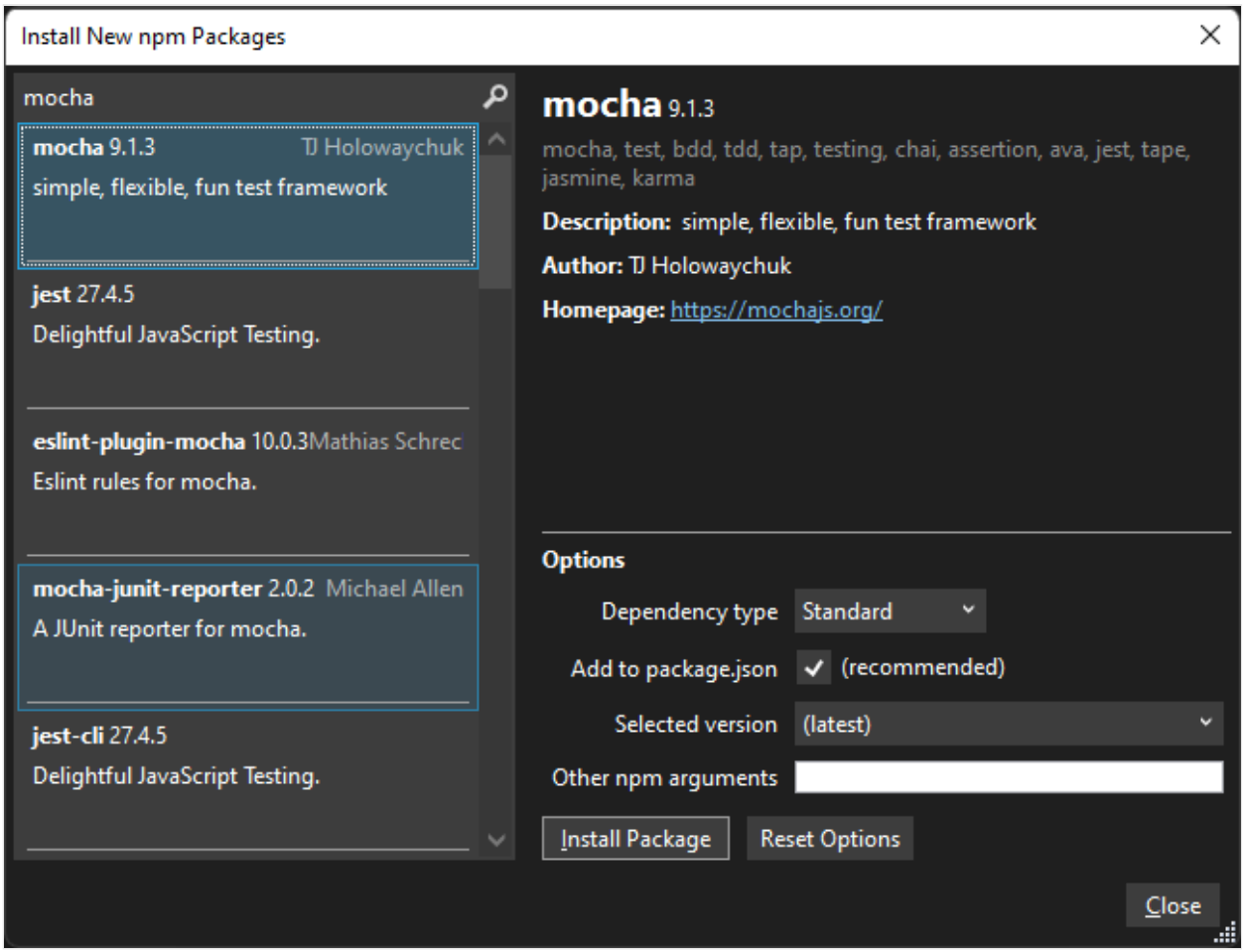
CLI-based project (.esproj)

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

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



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



ASP.NET Core projects

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

> **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. One of these options might be necessary if you require integration with MSBuild or the dotnet CLI for package management, which is not provided by npm.

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

2. 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:

   JSON

   ```
   "devDependencies": {
      "gulp": "4.0.2",
      "@types/jquery": "3.5.29"
   }
   ```

   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**. To view package installation status, select **npm** output in the Output window.

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

Troubleshooting npm packages

- 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 might need to install a more recent version to fix errors. For information on using *package.json* to control npm package versions, see package.json configuration.

- In some ASP.NET Core scenarios, Solution Explorer might not show the correct status for installed npm packages due to a known issue described here. For example, the package might appear as not installed when it is installed. In most cases, you can update Solution Explorer by deleting *package.json*, restarting Visual Studio, and re-adding the *package.json* file as described earlier in this article. Or, when installing packages, you can use the npm Output window to verify installation status.

- In some ASP.NET Core scenarios, the npm node in Solution Explorer might not be visible after you build the project. To make the node visible again, right-click the project node and choose **Unload Project.** Then right-click the project node and choose **Reload Project**.

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

Article • 01/12/2024

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.

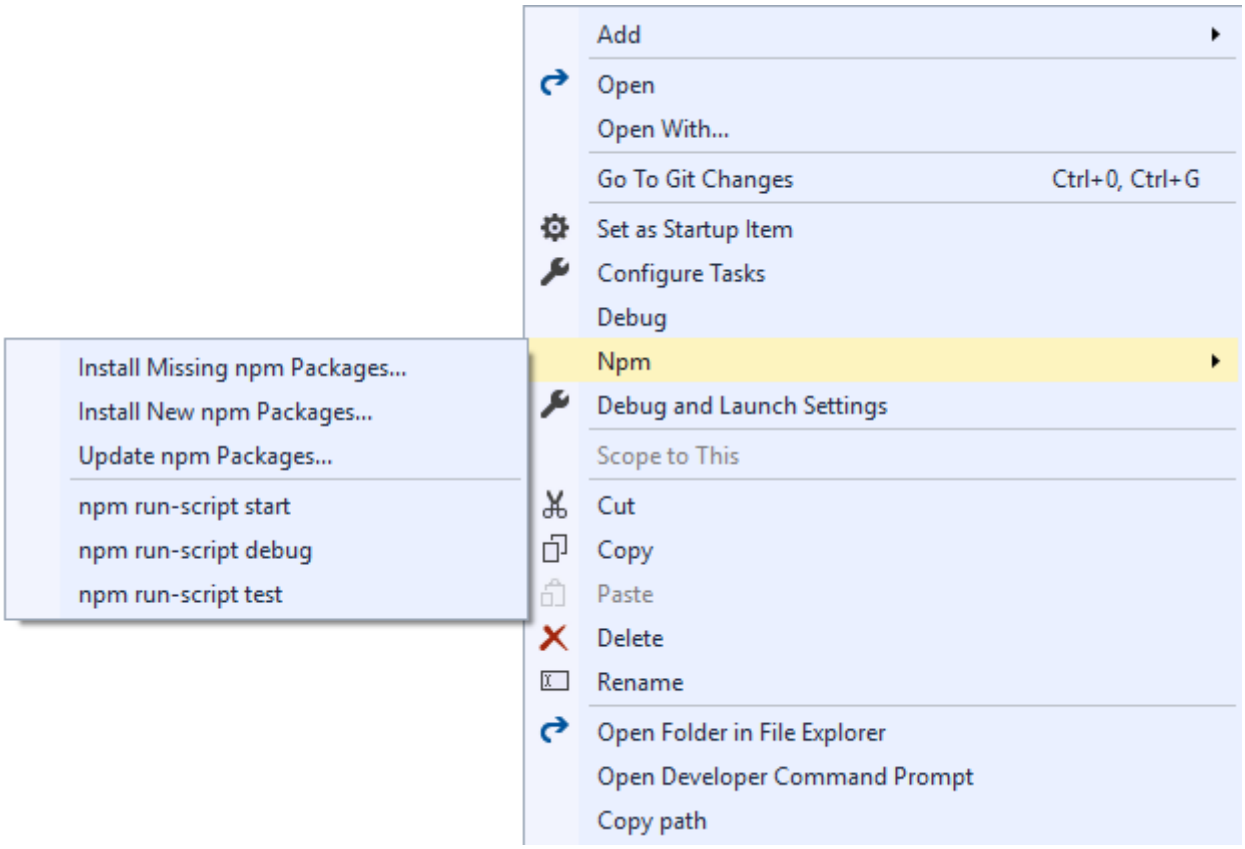
> **Tip**
>
> Before using an Open Folder project, try creating a solution from existing Node.js code. In some scenarios, this method provides better feature support in Visual Studio. To create the project, choose **File** > **New Project** > **JavaScript** > **From Existing Node.js code**, and then choose your project folder as the source.

Prerequisites

- Visual Studio 2017 version 15.8 or later versions
- 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.

> ⓘ **Note**

If you don't see the **Debug** menu option, you may need to create the project from existing Node.js code, as described previously.

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*:

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.

Debug a JavaScript or TypeScript app in Visual Studio

Article • 07/09/2024

You can debug JavaScript and TypeScript code using Visual Studio. You can 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.

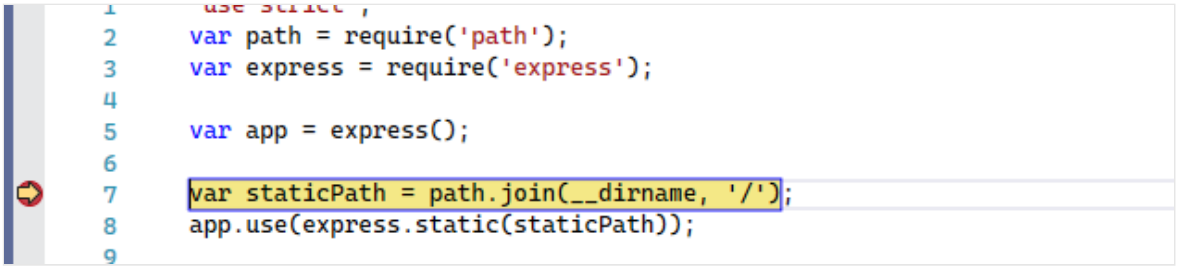
Debug server-side script

1. With your project open in Visual Studio, open a server-side JavaScript file (such as *server.js*), click in the gutter to 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 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 (IDE highlights the statement in the yellow background). Now, you can inspect your app state by hovering over variables 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, press **F12** in the Chrome browser. Using these tools, you can examine the DOM or interact with the app using the JavaScript Console.

Debug client-side script

Visual Studio provides client-side debugging support only for Chrome and Microsoft Edge. In some scenarios, the debugger automatically hits breakpoints in JavaScript and TypeScript code and embedded scripts on HTML files.

- For debugging client-side script in ASP.NET apps, choose **Tools** > **Options** > **Debugging**, and then select **Enable JavaScript Debugging for ASP.NET (Chrome, Edge, and IE)**.

  If you prefer to use Chrome Developer Tools or F12 Tools for Microsoft Edge to debug client-side script, you should disable this setting.

  For more detailed information, see this blog post for Google Chrome. For debugging TypeScript in ASP.NET Core, see Add TypeScript to an existing ASP.NET Core app.

- For Node.js applications and other JavaScript projects, follow the steps described here.

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

Prepare your app for debugging

If your source is minified or created by a transpiler like TypeScript or Babel, use source maps for the best debugging experience. You can even attach the debugger to a

running client-side script without the source maps. However, you may only be able to set and hit breakpoints in the minified or transpiled file, not in the source file. For example, in a Vue.js app, the minified script gets passed as a string to an `eval` statement, and there's no way to step through this code effectively using the Visual Studio debugger unless you use source maps. For complex debugging scenarios, you may want to use Chrome Developer Tools or F12 Tools for Microsoft Edge instead.

For help with generating source maps, see Generate source maps for debugging.

Prepare the browser for debugging

For this scenario, use either Microsoft Edge or Chrome.

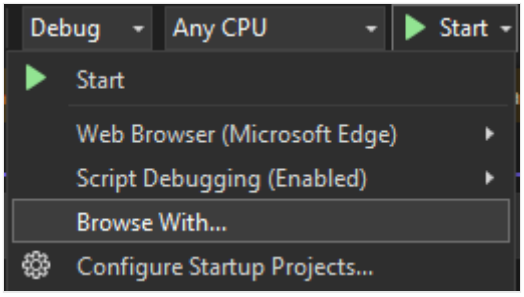
1. Close all windows for the target browser, either Microsoft Edge or Chrome instances.

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

   For best results, shut down all instances of Chrome, even if you're working with Microsoft Edge. Both the browsers use the same chromium code base.

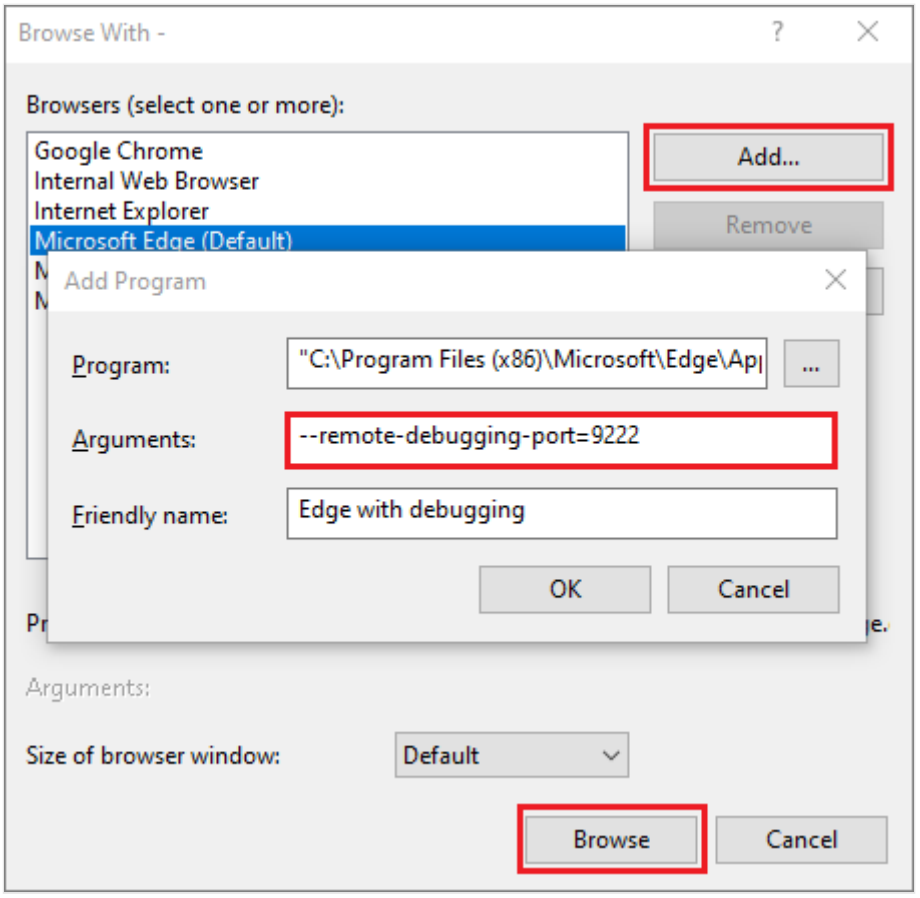
2. Start your browser with debugging enabled.

   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.

   If you don't see the **Browse With...** command in the **Debug** toolbar, select a different browser, and then retry.

   From the Browse With dialog box, choose **Add**, and then set the flag in the **Arguments** field. Use a different friendly name for the browser, like **Edge Debug Mode** or **Chrome Debug Mode**. For details, see the Release Notes.



Select **Browse** to start your app with the browser in debug mode.

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

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

or,

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

This starts your browser with debugging enabled.

The app isn't yet running, so you get an empty browser page. (If you start the browser using the Run command, you need to paste in the correct URL for your app instance.)

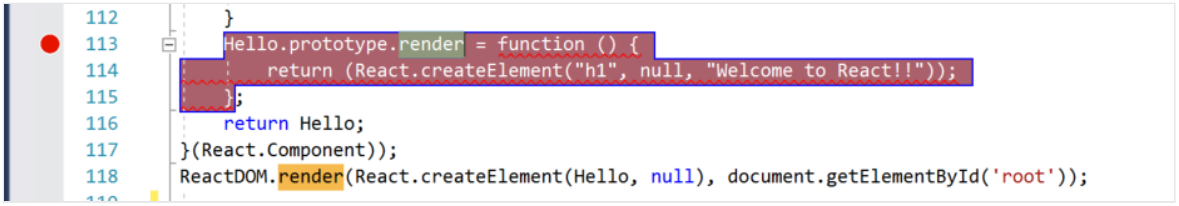
Attach the debugger to client-side script

To attach the debugger from Visual Studio and hit breakpoints in the client-side code, it needs help with identifying the correct process. Here's one way to enable it.

1. Make sure your app is running in the browser in debug mode, as described in the preceding section.

If you created a browser configuration with a friendly name, choose that as your debug target, and then press **Ctrl**+**F5** (**Debug** > **Start Without Debugging**) to run the app in the browser.

2. 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.

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

> **Tip**
>
> Starting in Visual Studio 2017, after 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**.

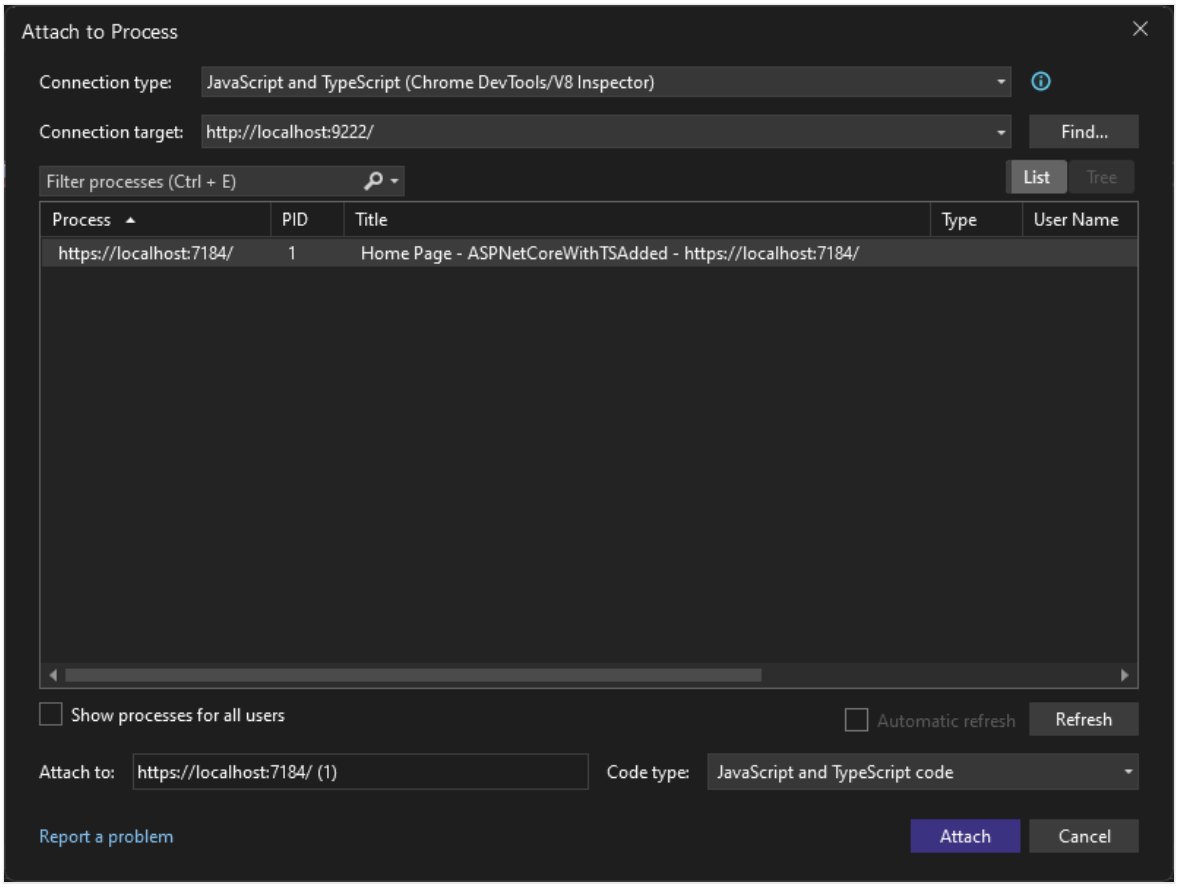
4. In the **Attach to Process** dialog, select **JavaScript and TypeScript (Chrome Dev Tools/V8 Inspector)** as the **Connection Type**.

   The debugger target, such as http://localhost:9222, should appear in the **Connection Target** field.

5. In the list of browser instances, select the browser process with the correct host port (`https://localhost:7184/` in this example), and select **Attach**.

   The port (for example, 7184) 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 browser.



> **Tip**
>
> If the debugger does not attach and you see the message "Failed to launch debug adapter" or "Unable to attach to the process. An operation is not legal in the current state.", use the Windows Task Manager to close all instances of the target browser before starting the browser in debugging mode. Browser extensions may be running and preventing full debug mode.

6. The code with the breakpoint may have already been 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 section to attach the debugger. Make sure that your environment is set up correctly:

- Close 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, read What are 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're 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're 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:

JavaScript

```
  output: {
    filename: "./app-bundle.js", // This is an example of the filename in
your project
  },
```

with this code:

JavaScript

```
  output: {
    filename: "./app-bundle.js", // Replace with the filename in your
project
    devtoolModuleFilenameTemplate: '[absolute-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.

JSON

```
{
  "compilerOptions": {
    "noImplicitAny": false,
    "module": "commonjs",
    "noEmitOnError": true,
    "removeComments": false,
    "sourceMap": true,
    "target": "es5"
  },
  "exclude": [
    "node_modules"
  ]
}
```

Compiler options for tsconfig.json file

- **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.

Debug JavaScript in dynamic files using Razor (ASP.NET)

In Visual Studio 2022, you can debug Razor pages using breakpoints. For more information, see Using Debugging Tools in Visual Studio.

Related content

Properties, React, Angular, Vue

Feedback

Unit testing JavaScript and TypeScript in Visual Studio

Article • 11/23/2023

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

You can write and run unit tests in Visual Studio using some of the more popular JavaScript frameworks without the need to switch to a command prompt. Both Node.js and ASP.NET Core projects are supported.

The supported frameworks are:

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

Write unit tests for a CLI-based project (.esproj)

The CLI-based projects supported in Visual Studio 2022 work with Test Explorer. Jest is the built-in test framework for React and Vue projects, and Karma and Jasmine is used for Angular projects. By default, you will be able to run the default tests provided by each framework, as well as any additional tests you write. Just hit the **Run** button in Test Explorer. If you don't already have Test Explorer open, you can find it by selecting **Test** > **Test Explorer** in the menu bar.

To run unit tests from the command-line, right-click the project in Solution Explorer, choose **Open in Terminal**, and run the command specific to the test type.

For information on setting up unit tests, see the following:

- Testing React with Jest
- Angular testing
- Testing Vue.js

A simple example is also provided here. However, use the preceding links for complete information.

Add a unit test (.esproj)

The following example is based on the TypeScript React project template provided in Visual Studio 2022 version 17.8 or later, which is the **Standalone TypeScript React Project** template. For Vue and Angular, the steps are similar.

1. In Solution Explorer, right-click the React project and choose **Edit Project File**.

2. Make sure that the following properties are present in the *.esproj* file with the values shown.

   XML

   ```
   <PropertyGroup>
     <JavaScriptTestRoot>src\</JavaScriptTestRoot>
     <JavaScriptTestFramework>Jest</JavaScriptTestFramework>
   </PropertyGroup>
   ```

   This example specifies Jest as the test framework. You could specify Mocha, Tape, or Jasmine instead.

   The `JavaScriptTestRoot` element specifies that your unit tests will be in the *src* folder of the project root.

3. In Solution Explorer, right-click the npm node and choose **Install new npm packages**.

   Use the npm package installation dialog to install the following npm packages:

   - jest
   - jest-editor-support

   These packages are added to the *package.json* file under dependencies.

4. In *package.json*, add the `test` section at the end of the `scripts` section:

   JSON

   ```
   "scripts": {
      ...
      "test": "jest"
   },
   ```

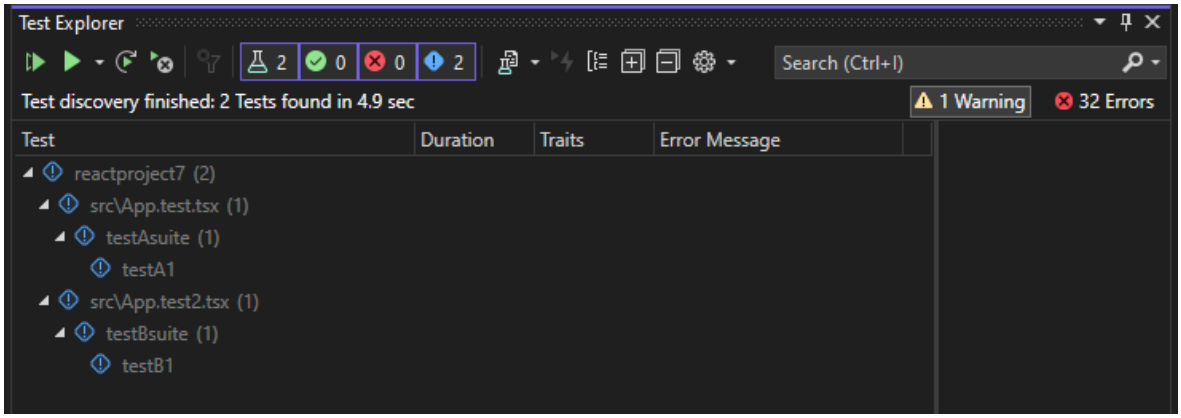
5. In Solution Explorer, right-click the src folder and choose **Add** > **New Item**, and then add a new file named *App.test.tsx*.

   This adds the new file under the src folder.

6. Add the following code to *App.test.tsx*.

```
JavaScript

describe('testAsuite', () => {
   it('testA1', async () => {
      expect(2).toBe(2);
   });
});
```

7. Open Test Explorer (choose **Test** > **Test Explorer**) and Visual Studio discovers and displays tests. If tests are not showing initially, then rebuild the project to refresh the list.

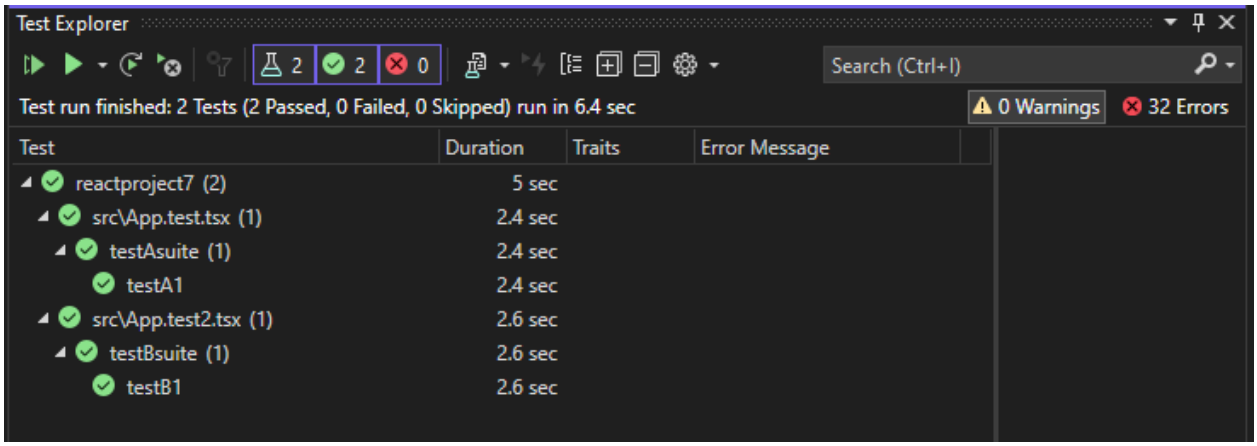


> **Note**
>
> For TypeScript, do not use the `outfile` option in *tsconfig.json*, because Test Explorer won't be able to find your unit tests. You can use the `outdir` option, but make sure that configuration files such as `package.json` and `tsconfig.json` are in the project root.

Run tests (.esproj)

You can run the tests by clicking the **Run All** link in Test Explorer. Or, you can run tests by selecting one or more tests or groups, right-clicking, and selecting **Run** from the shortcut menu. Tests run in the background, and Test Explorer automatically updates and shows the results. Furthermore, you can also debug selected tests by right-clicking and selecting **Debug**.

The following illustration shows the example with a second unit test added.



For some unit test frameworks, unit tests are typically run against the generated JavaScript code.

> **Note**
>
> In most TypeScript scenarios, you can debug a unit test by setting a breakpoint in TypeScript code, right-clicking a test in Test Explorer, and choosing **Debug**. In more complex scenarios, such as some scenarios that use source maps, you may have difficulty hitting breakpoints in TypeScript code. As a workaround, try using the `debugger` keyword.

> **Note**
>
> Profiling tests and code coverage are not currently supported.

Write unit tests for ASP.NET Core

To add support for unit testing of JavaScript and TypeScript in an ASP.NET Core project, you need to add TypeScript, Npm, and unit testing support to the project by including required NuGet packages.

Add a unit test (ASP.NET Core)

The following example is based on the ASP.NET Core Model-View-Controller project template, and includes adding a Jest or Mocha unit test.

1. Create an ASP.NET Core Model-View-Controller project.

   For an example project, see Create an ASP.NET Core app with TypeScript. For unit testing support, we recommend you start with a standard ASP.NET Core project

template.

2. In Solution Explorer (right pane), right-click the ASP.NET Core project node and select **Manage NuGet Packages for Solutions**.

3. In the **Browse** tab, search for the following packages and install each one:

   - Microsoft.TypeScript.MSBuild
   - Npm
   - Microsoft.JavaScript.UnitTest

   Use the NuGet package to add TypeScript support instead of the npm TypeScript package.

4. In Solution Explorer, right-click the project node and choose **Edit Project File**.

   The *.csproj* file opens in Visual Studio.

5. Add the following elements to the *.csproj* file in the `PropertyGroup` element.

   This example specifies Jest or Mocha as the test framework. You could specify Tape or Jasmine instead.

   Jest

   The `JavaScriptTestRoot` element specifies that your unit tests will be in the *tests* folder of the project root.

   XML

   ```
   <PropertyGroup>
      ...
      <JavaScriptTestRoot>tests\</JavaScriptTestRoot>
      <JavaScriptTestFramework>Jest</JavaScriptTestFramework>
      <GenerateProgramFile>false</GenerateProgramFile>
   </PropertyGroup>
   ```

6. In Solution Explorer, right-click the ASP.NET Core project node and select **Add** > **New Item**. Choose the **TypeScript JSON Configuration File**, and then select **Add**.

   If you don't see all the item templates, select **Show All Templates**, and then choose the item template.

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

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

Jest

JSON

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

For Jest, if you want to compile TypeScript tests to JavaScript, remove the *tests* folder from the *exclude* section.

The *scripts* folder is where you can put the TypeScript code for your app. For an example project that adds code, see Create an ASP.NET Core app with TypeScript.

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

   A *package.json* file is added to the project root.

9. In Solution Explorer, right-click the **npm** node under Dependencies and choose **Install new npm packages**.

> **Note**
>
> In some scenarios, Solution Explorer might not show the npm node due to a known issue described here. If you need to see the npm node, you can unload the project (right-click the project and choose **Unload Project**) and

then reload the project to make the npm node re-appear. Alternatively, you can add the package entries to *package.json* and install by building the project.

Use the npm package installation dialog to install the following npm packages:

Jest

- jest
- jest-editor-support
- @types/jest

These packages are added to the *package.json* file under devDependencies.

TypeScript

```
"@types/jest": "^29.5.8",
"jest": "^29.7.0",
"jest-editor-support": "^31.1.2"
```

10. In *package.json*, add the `test` section at the end of the `scripts` section:

Jest

JSON

```
"scripts": {
   ...
   "test": "jest"
},
```

11. In Solution Explorer, right-click the *test* folder and choose **Add** > **New Item**, and then add a new file named *App.test.tsx*.

    This adds the new file under the *test* folder.

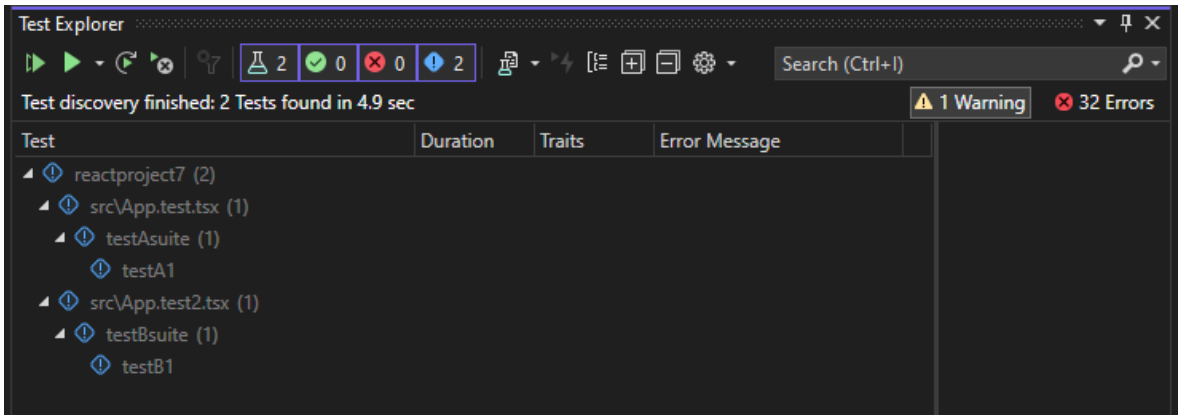
12. Add the following code to *App.test.tsx*.

Jest

JavaScript

```
describe('testAsuite', () => {
   it('testA1', async () => {
      expect(2).toBe(2);
   });
});
```

13. Open Test Explorer (choose **Test** > **Windows** > **Test Explorer**) and Visual Studio discovers and displays tests. If tests are not showing initially, then rebuild the project to refresh the list. The following illustration shows the Jest example, with two different unit test files.

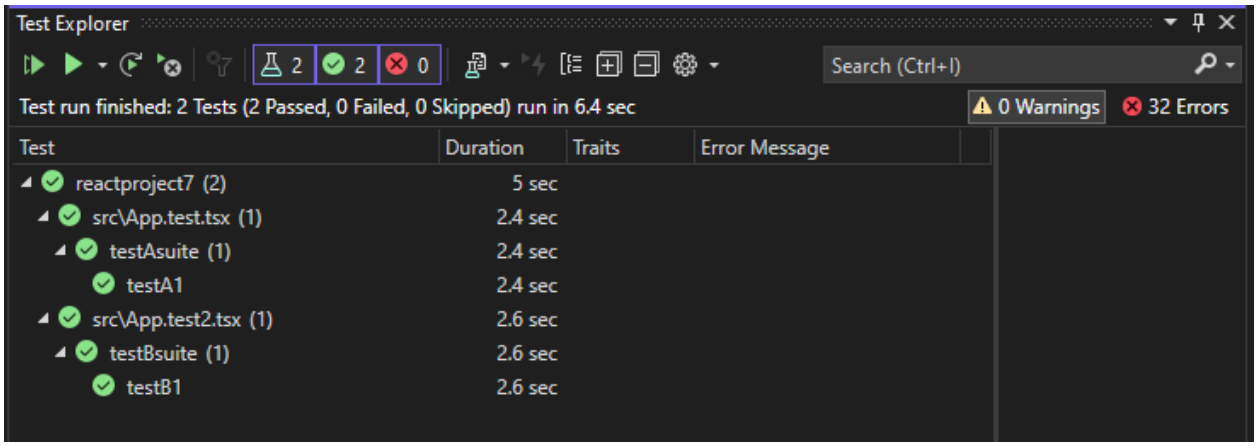


> **Note**
>
> For TypeScript, do not use the `outfile` option in *tsconfig.json*, because Test Explorer won't be able to find your unit tests. You can use the `outdir` option, but make sure that configuration files such as `package.json` and `tsconfig.json` are in the project root.

Run tests (ASP.NET Core)

You can run the tests by clicking the **Run All** link in Test Explorer. Or, you can run tests by selecting one or more tests or groups, right-clicking, and selecting **Run** from the shortcut menu. Tests run in the background, and Test Explorer automatically updates and shows the results. Furthermore, you can also debug selected tests by right-clicking and selecting **Debug**.

The following illustration shows the Jest example, with a second unit test added.



For some unit test frameworks, unit tests are typically run against the generated JavaScript code.

> **Note**
>
> In most TypeScript scenarios, you can debug a unit test by setting a breakpoint in TypeScript code, right-clicking a test in Test Explorer, and choosing **Debug**. In more complex scenarios, such as some scenarios that use source maps, you may have difficulty hitting breakpoints in TypeScript code. As a workaround, try using the `debugger` keyword.

> **Note**
>
> Profiling tests and code coverage are not currently supported.

Property pages for React, Angular, and Vue projects in Visual Studio

Article • 10/23/2023

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

This article applies to React, Angular, and Vue projects created in Visual Studio that use the JavaScript Project System (JSPS), which is the *.esproj* project format. This format is supported starting in Visual Studio 2022.

The **Property Pages** provides access to project settings. To open the property pages, select the project in **Solution Explorer** and then select the **Properties** icon, or right-click the project and select **Properties**.

> **Note**
>
> Your computer might show different names or locations for some of the Visual Studio user interface elements in this article. You may be using a different edition of Visual Studio or different environment settings. For more information, see Personalize the IDE.

The following pages and options appear in the **Property Pages**.

Build tab

Under the General tab, the following properties are available.

Build Command

Specifies the command to run when you build the project. (**Build** > **Build Solution**, or when you run the project.) If used, this value is typically an npm command. This property corresponds to the BuildCommand property in the project file.

Production Build Command

Specifies the command to run when you build the project, when the project is integrated with the ASP.NET Core Web API project. Generates production-ready files. By default, this command is `npm run build`.

Starting in Visual Studio 2022 version 17.5, this option is not present in the recommended project templates for React, Vue, and Angular.

Build Output Folder

Specifies the output folder for production build objects. For older projects, use this property when you use the **Production Build Command**.

Clean Command

Specifies the command to run when you clean the project. (**Build** > **Clean Solution**) If used, this value is typically an npm command. This property corresponds to the CleanCommand property in the project file.

Working Directory

Specifies the working directory for the build command. This value is the project root, by default.

Deploy tab

Startup Command

Specifies the command to execute when you start the project. For example, an Angular project uses `npm start` by default. This property corresponds to the StartupCommand property in the project file.

Working Directory

Specifies the working directory for the startup command. By default, this value is the project root. Relative paths are relative to the project root.

> **Note**
>
> In Visual Studio, *launch.json* stores the startup settings associated with the **Start** button in the Debug toolbar. *launch.json* must be located under the *.vscode* folder.

See also

JavaScript and TypeScript in Visual Studio

MSBuild reference for the JavaScript Project System

Article • 10/23/2023

This article provides reference information for the MSBuild properties and items that you can use to configure projects based on the JavaScript Project System (JSPS), which use the *.esproj* format.

> **Note**
>
> The properties described in this article extend the properties MSBuild provides by default. For a list of common MSBuild properties, see **Common MSBuild properties**.

ShouldRun properties

The following MSBuild properties are documented in this section:

- ShouldRunNpmInstall
- ShouldRunBuildScript

ShouldRunNpmInstall

The `ShouldRunNpmInstall` property specifies whether to run or not run `npm install` on Build and Restore commands. The default value for the property is `true` if unset.

XML

```
<PropertyGroup>
  <ShouldRunNpmInstall>false</ShouldRunNpmInstall>
</PropertyGroup>
```

Two common scenarios where not running `npm install` is desirable are:

1. When a non-npm package manager (such as yarn or pnpm) is used. In this scenario, the best solution is to create a target that runs before `BeforeRestore` to manually run the installation.
2. When a global package installation mechanism exists in the solution that makes running individual installations unnecessary.

ShouldRunBuildScript

The `ShouldRunBuildScript` property specifies whether or not to run `npm run build` on Build commands. The default value for the property is `true` if unset.

```xml
<PropertyGroup>
  <ShouldRunNpmBuildScript>false</ShouldRunNpmBuildScript>
</PropertyGroup>
```

For projects containing only JavaScript that do not require building, set this property to `false`. Newly created React, Vue, and Angular projects usually fall in this category. In this scenario, build is used for production and not for debugging. Note that the `Build` and `Publish` commands are separate in JSPS projects, and `Publish` still runs even if this property is set.

Command Properties

Command properties are properties intended to map common *package.json* scripts to MSBuild targets. Default values are supported for all of these properties, as described in this section.

Set these properties when using package managers other than npm, or scripting engines such as `gulp`.

The following MSBuild properties are described in this section:

- BuildCommand
- StartupCommand
- TestCommand
- CleanCommand
- PublishCommand

BuildCommand

The `BuildCommand` property specifies the behavior for the `build` target. If the associated *package.json* contains a `build` or `compile` script, the default `BuildCommand` value is already set to run them.

To modify the command, include `npm run` when using `npm`.

XML

```xml
<PropertyGroup>
  <BuildCommand>npm run build</BuildCommand>
</PropertyGroup>
```

StartupCommand

The `StartupCommand` property specifies the behavior for the `dotnet run` target. If the associated `package.json` contains a `start`, `server`, or `dev` script, the default `StartupCommand` value is already set to run these scripts.

If you modify the command, include `npm run` when using `npm`.

XML

```xml
<PropertyGroup>
  <StartupCommand>npm run serve</StartupCommand>
</PropertyGroup>
```

TestCommand

The `TestCommand` property specifies the behavior for the `test` target. If the associated `package.json` contains a `test` script, the default `TestCommand` value is already set to run this script.

If you modify the command, include `npm run` when using `npm`.

XML

```xml
<PropertyGroup>
  <TestCommand>ng test</TestCommand>
</PropertyGroup>
```

CleanCommand

The `CleanCommand` property specifies the behavior for the `clean` target. If the associated `package.json` contains a `clean` script, the default `CleanCommand` value is already set to run this script.

If you modify the command, include `npm run` when using `npm`.

XML

```
<PropertyGroup>
  <CleanCommand>npm run clean</CleanCommand>
</PropertyGroup>
```

PublishCommand

The `PublishCommand` property specifies the behavior for the `publish` target. If the associated `package.json` contains a `publish` script, the default pro`PublishCommand` value is already set to run this script. In npm, it is common to have pre- and post- publish scripts, which will also run.

If you modify the command, include `npm run` when using `npm`.

XML

```
<PropertyGroup>
  <PublishCommand>npm run publish</PublishCommand>
</PropertyGroup>
```

See also

- MSBuild schema reference
- Common MSBuild properties
- MSBuild properties for NuGet pack
- MSBuild properties for NuGet restore
- Customize a build

package.json configuration

Article • 04/30/2022

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

If you are developing a Node.js app with a lot of npm packages, it's not uncommon to run into warnings or errors when you build your project if one or more packages has been updated. Sometimes, a version conflict results, or a package version has been deprecated. Here are a couple of quick tips to help you configure your package.json file and understand what is going on when you see warnings or errors. This is not a complete guide to *package.json* and is focused only on npm package versioning.

The npm package versioning system has strict rules. The version format follows here:

```
[major].[minor].[patch]
```

Let's say you have a package in your app with a version of 5.2.1. The major version is 5, the minor version is 2, and the patch is 1.

- In a major version update, the package includes new features that are backwards-incompatible, that is, breaking changes.
- In a minor version update, new features have been added to the package that are backwards-compatible with earlier package versions.
- In a patch update, one or more bug fixes are included. Bug fixes are always backwards-compatible.

It's worth noting that some npm package features have dependencies. For example, to use a new feature of the TypeScript compiler package (ts-loader) with webpack, it is possible you would also need to update the webpack npm package and the webpack-cli package.

To help manage package versioning, npm supports several notations that you can use in the *package.json*. You can use these notations to control the type of package updates that you want to accept in your app.

Let's say you are using React and need to include the **react** and **react-dom** npm package. You could specify that in several ways in your *package.json* file. For example, you can specify use of the exact version of a package as follows.

JSON

```
"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:

JSON

```
"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.

JSON

```
"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.