

How to add Webpack to Asp.Net Core MVC

We will use tools and pipelines that are conventional among the front-end developers:

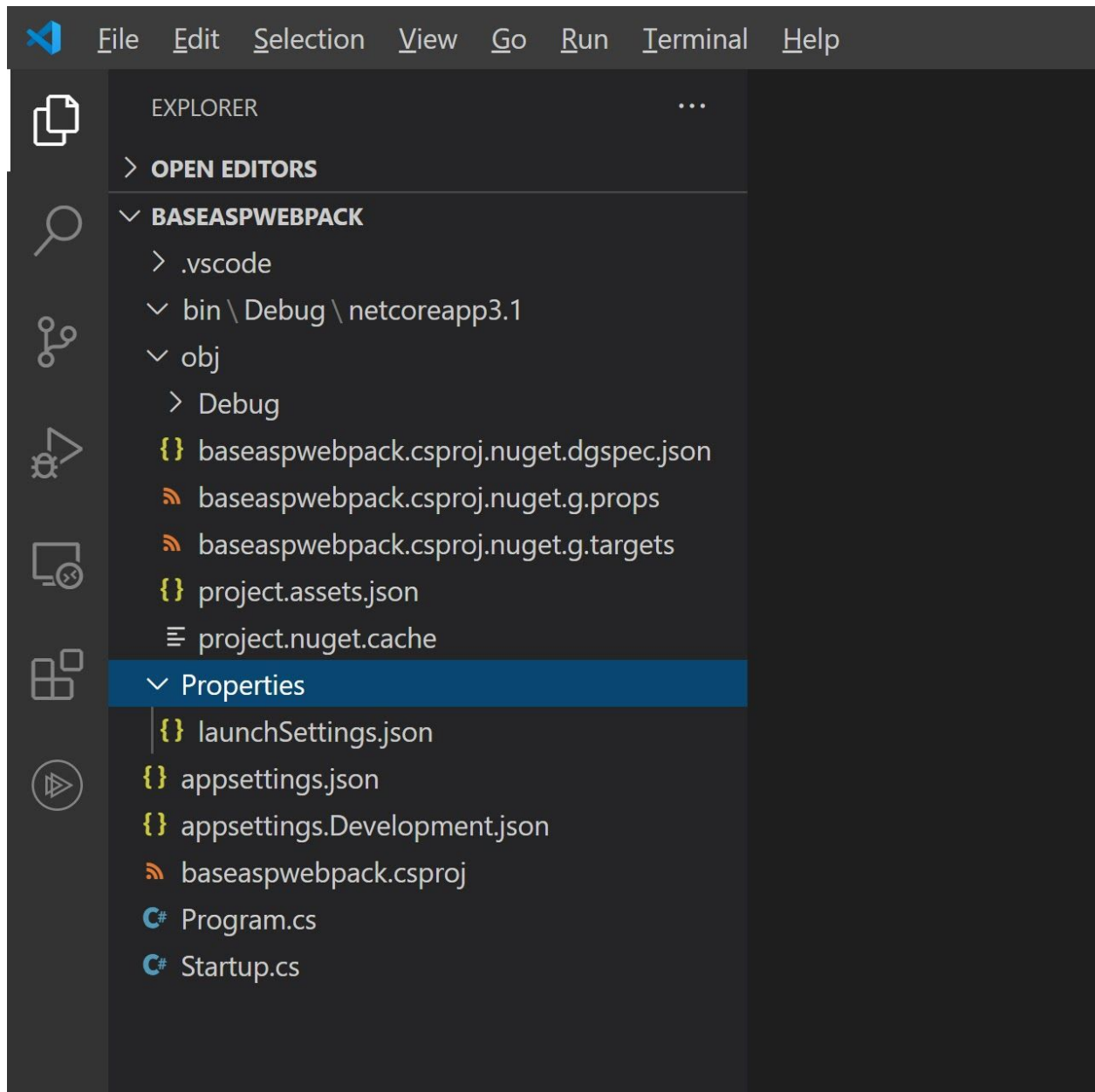
1. NPM package manager to manage Node.js packages and to run scripts.
2. Javascript ES6-module syntax in the JavaScript code.
3. SASS to process styles.
4. Webpack 4 to bundle all things together.
5. and Visual Studio Code as an IDE.

Create the empty ASP.NET Core MVC application

Using the command line, move to the folder where you want to create a new project. Then type in the next command:

```
dotnet new web
```

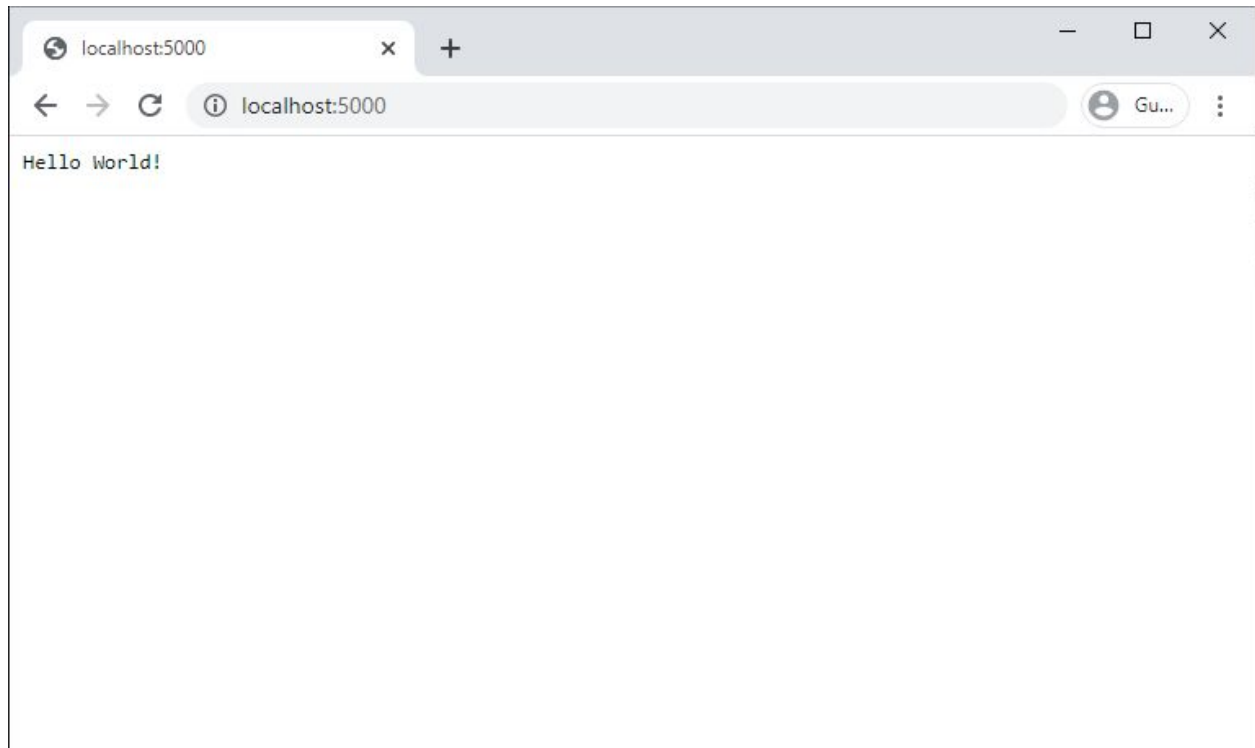
This command creates an empty ASP.NET Core application with the next structure:



Build and run it with the next .Net Core CLI commands:

```
dotnet build
dotnet run
```

At this stage, the application does not do much.



Let's add a controller and Views to convert it to the very simple MVC web app.

Add MVC functionality to the application

First, modify the `Startup.cs` class. It should look as listed below. The namespace name depends on your project name.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
namespace AspNetCore3Webpack4
```

```

{
    public class Startup
    {
        // This method gets called by the runtime. Use this method to
        // add services to the container.
        // For more information on how to configure your application,
        // visit https://go.microsoft.com/fwlink/?LinkID=398940
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllersWithViews();
        }
        // This method gets called by the runtime. Use this method to
        // configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            app.UseStaticFiles();
            app.UseRouting();
            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllerRoute(
                    name: "default",
                    pattern:
                    "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }
}

```

Create Controllers folder and add a file HomeController.cs with Index() action.

```

using Microsoft.AspNetCore.Mvc;
namespace AspNetCore3Webpack4.Controllers
{

```

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
}
```

Create a Views folder . Next, create Home and Shared sub-folders inside it.

Add the `_ViewStart.cshtml` file to the Views folder and add the following content:

```
@{
    Layout = "_Layout";
}
```

Add the `Index.cshtml` file to the `views/home` directory, with the following content:

```
@{
    ViewData["Title"] = "Home";
}

<h2>Home</h2>
```

Create the `_ViewImports.cshtml` file into the Views folder and place the next code inside:

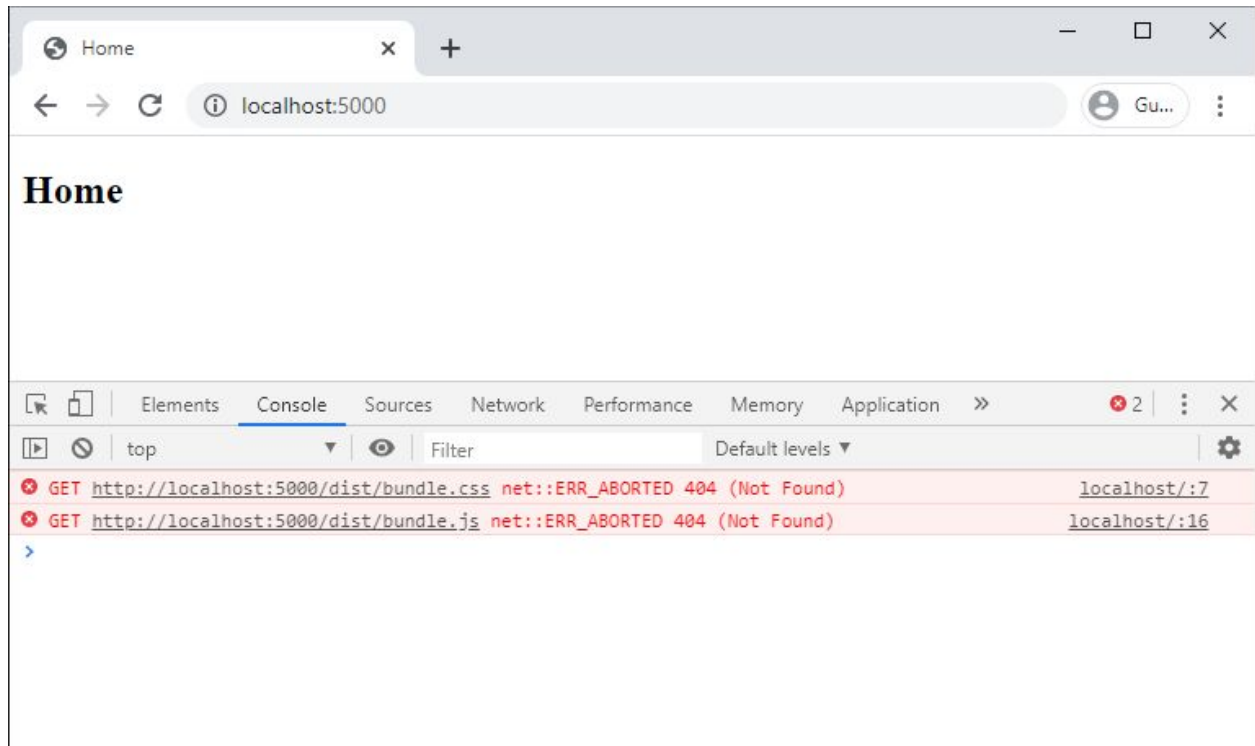
```
@using [YOU PROJECT NAME]
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Add the `_Layout.cshtml` file to the Shared folder. This file should include a common layout for all of the views i. e. navigation menu, footer, etc. So, put the next HTML markup inside.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width,
initial-scale=1.0" />
  <title>@ViewData["Title"]</title>
  <link rel="stylesheet" href="~/dist/bundle.css"
asp-append-version="true"/>
</head>
<body>
  <div>
    @RenderBody()
  </div>
  <script src="~/dist/bundle.js"
asp-append-version="true"></script>
</body>
</html>
```

And finally, add the `wwwroot` and `wwwroot/dist` folder. They will contain all static resources of our application.

At this point, the only thing our application can do is simply display the “Home” header :).

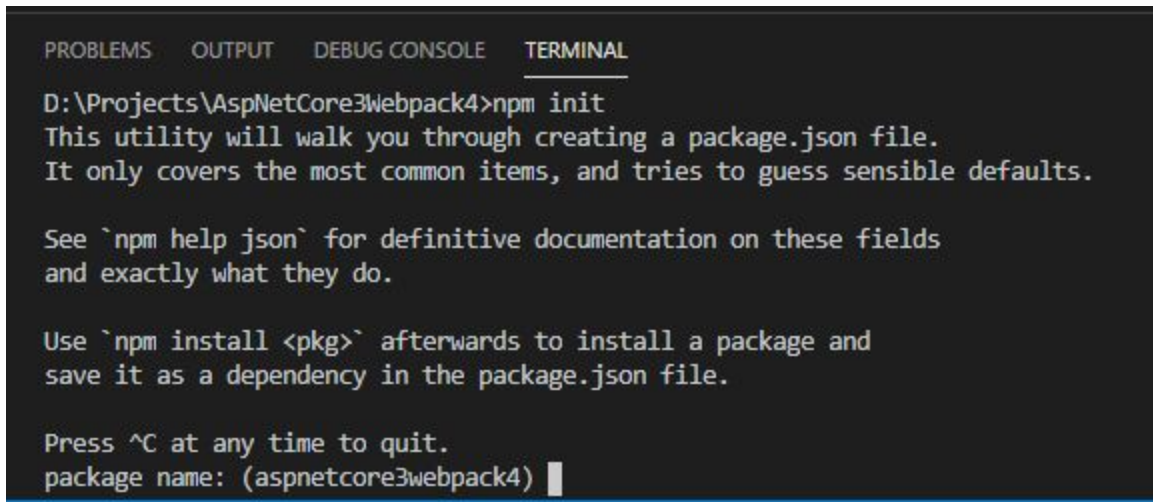


But as you can see there are a couple of errors in the browser console. That's because those files are not created yet. Let's correct this.

Add Node.js packages to the project

You should have Node.js installed to continue.

It's time to add NPM settings to the project. The simplest way to do this is to use `npm init` command from the command line. You can use your Visual Studio Code terminal.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
D:\Projects\AspNetCore3Webpack4>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (aspnetcore3webpack4) |
```

Since we will use NPM as a task runner, just press Enter for any step.

By performing this command, NPM will create the basic `package.json` file inside your project directory.

Install Webpack and other development dependencies

Type in (or simply copy and paste) the next command to the command line tool:

```
npm install --save-dev webpack webpack-cli style-loader sass-loader
postcss-loader node-sass mini-css-extract-plugin cssnano css-loader
clean-webpack-plugin
```

With this command, NPM installs Webpack, Webpack command line, and other node packages which are necessary to process JavaScript code and CSS styles into the bundles. All development dependencies will be installed into the `node_modules` directory.

You can install other development dependencies at any time you need.

At this step (and any time later), you can install any JavaScript libraries, frameworks, or any other requirements needed for your project. For example, let's use the [lodash](#) library.

```
npm install --save lodash
```

After these steps completed, you will get the package.json similar to listed below (packages versions will differ at yours systems and Github links may be omitted):

```
{
  "name": "baseaspwebpack",
  "version": "1.0.0",
  "description": "Base app for working with asp.net core and webpack.",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Michiel Oppenheimer",
  "license": "ISC",
  "devDependencies": {
    "clean-webpack-plugin": "^3.0.0",
    "css-loader": "^4.2.0",
    "cssnano": "^4.1.10",
    "mini-css-extract-plugin": "^0.9.0",
    "node-sass": "^4.14.1",
    "postcss-loader": "^3.0.0",
    "sass-loader": "^9.0.2",
    "style-loader": "^1.2.1",
    "webpack": "^4.44.1",
    "webpack-cli": "^3.3.12"
  },
  "dependencies": {
    "lodash": "^4.17.19"
  }
}
```

Add JavaScript and SASS sources

Because of using the ES6 standard of JavaScript and SASS to define CSS styles, we need the directory to store initial source codes. Those sources will be compiled into the production bundles by Webpack.

Inside the project directory, create the `./src` folder. Add the `index.js` file to this folder and put the next code inside:

```
import _ from 'lodash';
function createChild() {
  var element = document.createElement('div');
  element.innerHTML = _.join(['Hello', 'Webpack'], ' ');
  return element;
}
document.body.appendChild(createChild());
```

Here we add the new div-element with “Hello Webpack” text to our web-page. Nothing fancy.

Add the SASS sources

Add the `./src/sass` directory to the `./src` folder which was created at the previous step. Inside it, create two files: `index.scss` and `common.scss`.

The `common.scss` file intended to store SASS variables, which are common for our project. Let's put inside a single variable:

```
$color: #444;
```

The `index.scss` file should contain all necessary styles, so it has to import `common.scss`.

```
@import 'common.scss';
body{
  color: $color;
}
```

Configure the Webpack

Now we have some sources to process. It's time to hire a webpack to do its job.

In your project root folder create the `webpack.config.js` file and put the next code into it:

```
const path = require('path');
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const {CleanWebpackPlugin} = require('clean-webpack-plugin');
const bundleFileName = 'bundle';
const dirName = 'wwwroot/dist';
module.exports = (env, argv) => {
  return {
    mode: argv.mode === "production" ? "production" :
    "development",
    entry: ['./src/index.js', './src/sass/index.scss'],
    output: {
      filename: bundleFileName + '.js',
      path: path.resolve(__dirname, dirName)
    },
    module: {
      rules: [
        {
          test: /\.s[c|a]ss$/,
          use: [
            'style-loader',
            MiniCssExtractPlugin.loader,
            'css-loader',
            {
              loader: 'postcss-loader',
              options: {
                config: {
                  ctx: {
                    env: argv.mode
                  }
                }
              }
            }
          ]
        }
      ]
    }
  }
}
```

```

    },
    {
      loader: 'sass-loader',
      options: {}
    }
  ],
  plugins: [
    new CleanWebpackPlugin(),
    new MiniCssExtractPlugin({
      filename: bundleFileName + '.css'
    })
  ]
};
};

```

Here we use an arrow function to get webpack-cli arguments. This function returns the Webpack configuration object. We need the `-mode` argument from webpack-cli. This value is packed to the `argv.mode` parameter and is used in `mode` property to affect on final JavaScript code minification. Also, we use it to configure PostCSS loader via the `options.config.ctx.env` property. Additionally, the destination directory will be cleaned with the `CleanWebpackPlugin`.

Description of each loader you can find in the Webpack [loaders documentation](#).

Note, that 'postcss-loader' configuration object MUST BE USED BEFORE the 'sass-loader'!

To configure the *PostCSS* plugin, which used to minify final style-sheets, add the `postcss.config.js` file to the `./src/sass` directory. Add the next source code to the `postcss.config.js`:

```

module.exports = ({ options }) => {
  const plugins = [];
  if (options.env === 'production')
    plugins.push(require('cssnano'));
  return {

```

```
        plugins: plugins
    };
};
```

Here, we add the `cssnano` plugin to the `plugins` array only when the environment is set to production mode.

Add NPM scripts to run Webpack

After configuring, Webpack is able to create the bundles of JavaScript and CSS codes. But how do we instruct it to do that? The easiest way to do this is to add few commands to the `scripts` section of the `package.json` file:

```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "build": "webpack --progress --profile",
  "watch": "webpack --progress --profile --watch",
  "production": "webpack --progress --profile --mode production"
},
```

More about Webpack command line interface options you can find [here](#).

To create development bundles, which can be debugged in the browser development tools, use the next command in your command line, for example in the Visual Studio Code terminal:

```
npm run build
```

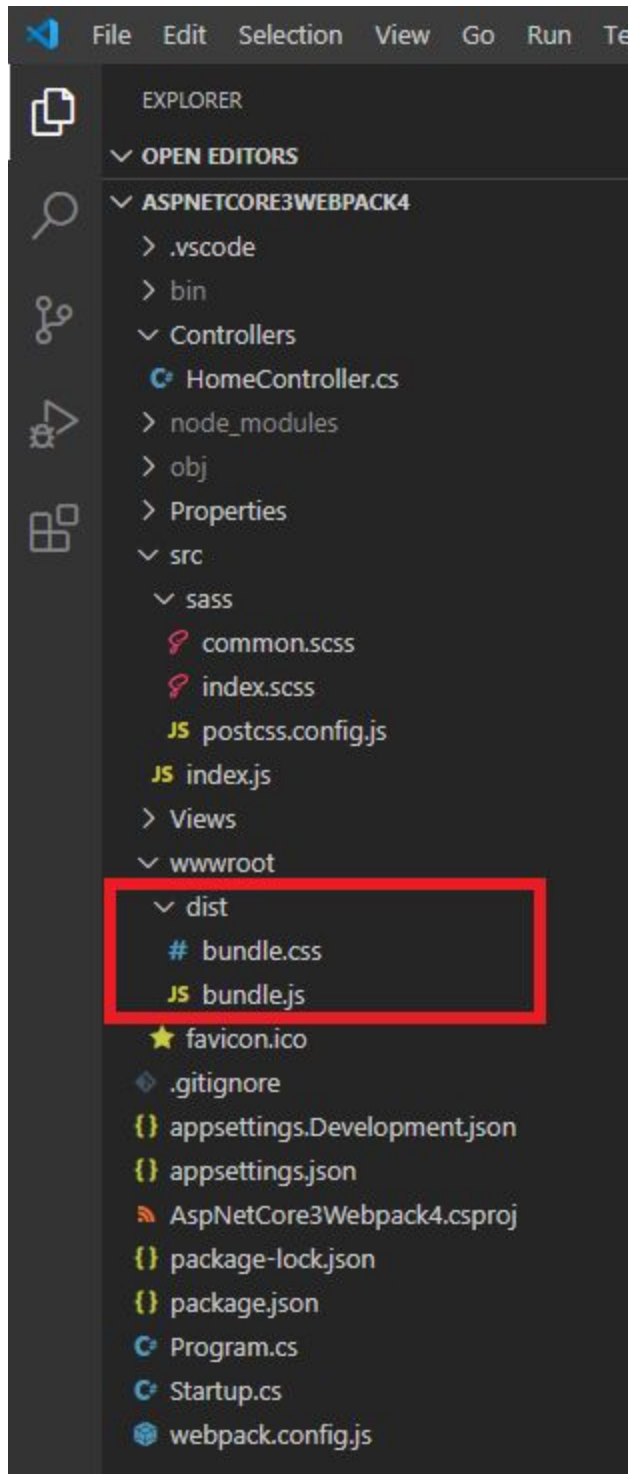
In the same way, you can run production minified bundles:

```
npm run production
```

When developing, it's convenient to update bundles when something changes in your source files. So, use this command:

```
npm run watch
```

All of these commands create two bundle files into the `wwwroot/dist` folder.



Results and conclusions

After performing all steps above, our application displays almost the classical “Hello Webpack” 😊 message, which is formed in the bundle.js file. And all CSS styles are composed in the bundle.css file. Both files are created by Webpack from different sources located in the ./src directory of our project. Depending on your needs, those bundles may contain the production or the development versions of our static resources.

