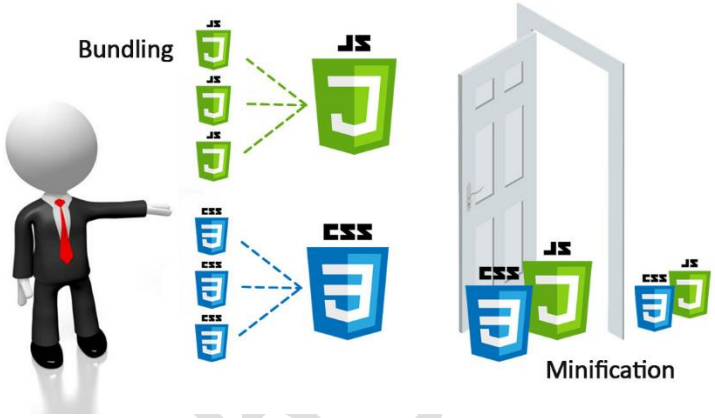
Bundling and Minification in ASP.NET Core

It is the dream of every web developer to build blazing fast and high-performance web applications but this is not easy to accomplish unless we implement some performance optimizations. Web pages have evolved from static HTML pages to complex and responsive pages with a lot of dynamic contents and plugins which require a large number of CSS and JavaScript files to be downloaded to the clients. To improve the initial page request load time, we normally apply two performance techniques called **bundling** and **minification**.

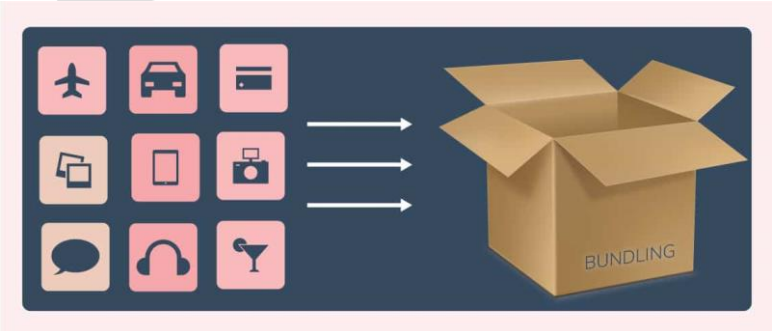
Introduction to Bundling and Minification:

Bundling and **minification** are two distinct performance optimizations we can use in our web apps to improve the performance of our apps and to reduce the number of requests from the browser to server and to reduce the size of data transferred from server to browser. If these two techniques are used together they improve the load time of our page on the first request. Once the page has been requested and loaded into the browser, the browser caches the static files e.g. JavaScript, CSS, and images so this improves the overall user experience.



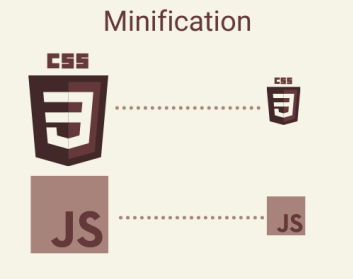
Bundling

Bundling is the process of combining multiple files into a single file. Bundling reduces the number of server requests that are necessary to render a web asset, such as a web page. You can create any number of individual bundles specifically for CSS, JavaScript, etc. Fewer files mean fewer HTTP requests from the browser to the server or from the service providing your application. This results in improved first page load performance. We can have separate bundles for CSS and JavaScript files and we can even have separate bundles for third-party plugins, scripts, and styles.



Minification

Minification is the process of removing unnecessary data without affecting any functionality. It removes **comments**, **white spaces**, **line breaks** and **converts large variable names to small names** (**shortening variable names to one character**). This means we can reduce the file size significantly which increases the file load time from server to client.



ASP.NET Core bundling and minification should be generally enabled only in the production environment and for the non-production environments used original files which makes it easier to debug.

Consider the following JavaScript function:

JavaScript

```
AddAltToImg = function (imageTagAndImageID, imageContext) {
 ///<signature>
 ///<summary> Adds an alt tab to the image
 // </summary>
 //<param name="imgElement" type="String">The image selector.</param>
 //<param name="ContextForImage" type="String">The image context.</param>
 ///</signature>
 var imageElement = $(imageTagAndImageID, imageContext);
 imageElement.attr('alt', imageElement.attr('id').replace(/ID/, ''));
}
```

Minification reduces the function to the following:

JavaScript

```
AddAltToImg=function(t,a){var r=$(t,a);r.attr("alt",r.attr("id").replace(/ID/,""))};
```

In addition to removing the comments and unnecessary whitespace, the following parameter and variable names were renamed as follows:

Original	Renamed
`imageTagAndImageID`	`t`
`imageContext`	`a`
`imageElement`	`r`

Impact of bundling and minification

The following table outlines differences between individually loading assets and using bundling and minification for a typical web app.

Action	Without B/M	With B/M	Reduction
File Requests	18	7	61%
Bytes Transferred (KB)	265	156	41%
Load Time (ms)	2360	885	63%

The load time improved, but this example ran locally. Greater performance gains are realized when using bundling and minification with assets transferred over a network.

The test app used to generate the figures in the preceding table demonstrates typical improvements that might not apply to a given app. We recommend testing an app to determine if bundling and minification yields an improved load time.

Bundling and Minification Strategies in ASP.NET Core

In old days, we were using a different technique to register all our bundles in **BundleConfig.cs** file available in **App_Start** folder. A method called **RegisterBundles** in ASP.NET was used to create, register, and configure bundles as follows:

```
public static void RegisterBundles(BundleCollection bundles)
{
    bundles.Add(new ScriptBundle("~/bundles/js").Include(
      "~/Scripts/bootstrap.js",
      "~/Scripts/jquery-3.3.1.js"));
}
```

We were including above bundles in our Web pages, Views and Layout Pages as follows:

```
<head>
    <title>@ViewBag.Title</title>

    @Scripts.Render("~/bundles/js")
</head>
```

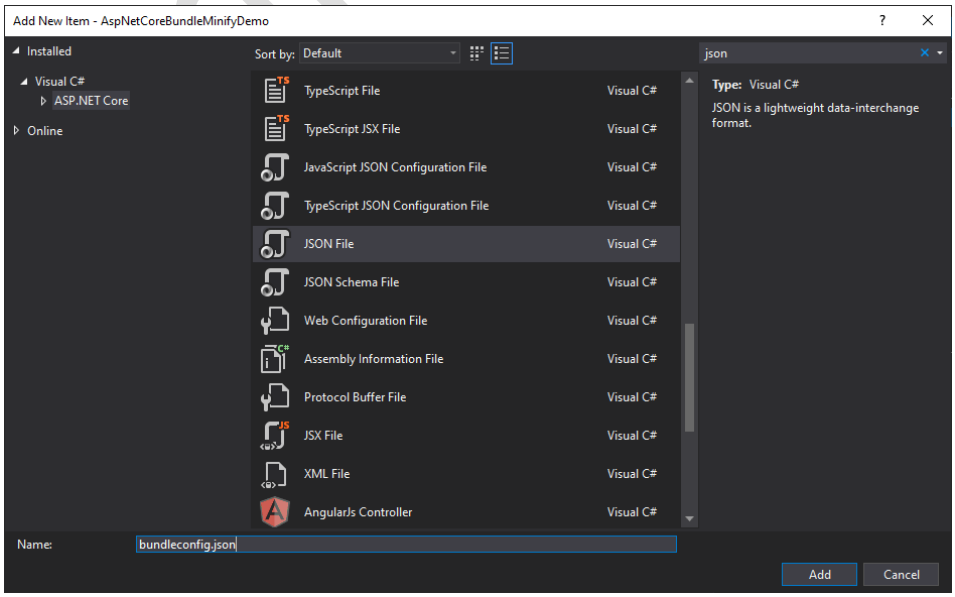
While this approach looks easy to use, it's no longer applicable in ASP.NET Core. The new ASP.NET Core project templates in Visual Studio provide a solution for bundling and minification using a JSON configuration file called **bundleconfig.json**. There are also some NuGet Packages and Visual Studio extensions that can help us in bundling and minification.

If you need a more advanced bundling and minification solution or your workflow requires processing beyond bundling and minification then you have the option of using third-party tools or task runners such as **Gulp**, **Grunt** or **Webpack**, etc.

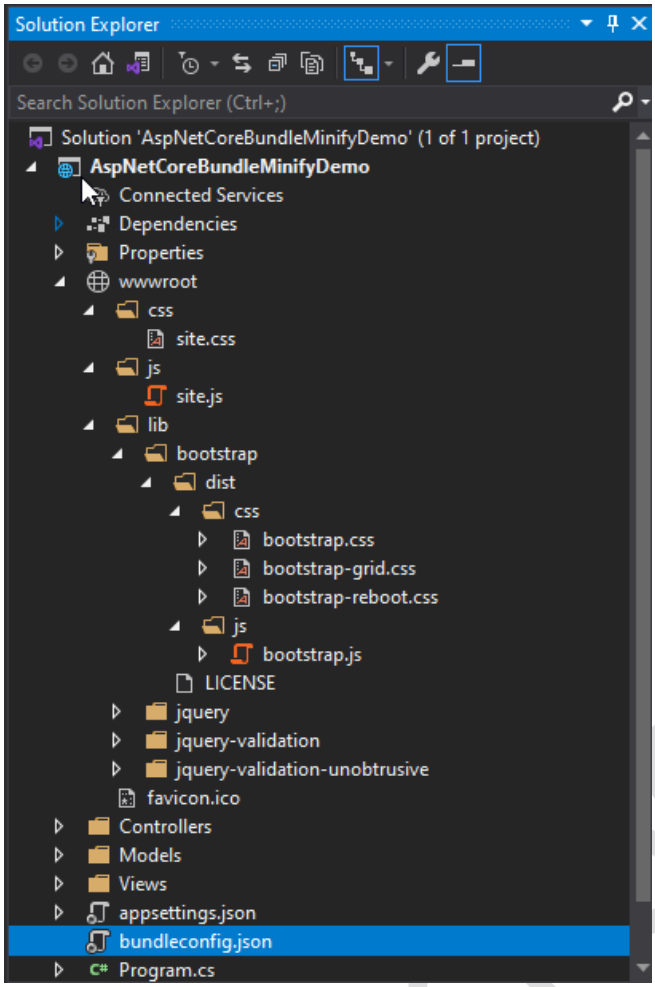
Another important aspect is to decide whether you want to do bundling and minification at design time or run time. Design time bundling and minification will allow you to create minified files before application deployment which will reduce server load time. However, the downside is that the design time bundling and minification only work with static files and it will also increase the build complexity.

Configure Bundles using bundleconfig.json File

In ASP.NET Core we typically add our static files in the wwwroot folder. There is a folder css for CSS files, a folder js for JavaScript files, and a third folder lib for external libraries such as JQuery, bootstrap, etc. To start bundling and minification, the first you need is a **bundleconfig.json** file so let's add this file using the standard Add New Item dialog in Visual Studio.



Typically, we add this file to our project root folder.



Here's an example of how the contents of **bundleconfig.json** look like in the default template:

```
[
  {
    "outputFileName": "wwwroot/css/site.min.css",
    "inputFiles": [
      "wwwroot/lib/bootstrap/dist/css/bootstrap.css",
      "wwwroot/css/site.css"
    ]
  },
  {
    "outputFileName": "wwwroot/js/site.min.js",
    "inputFiles": [
      "wwwroot/js/site.js"
    ],
    "minify": {
      "enabled": true,
      "renameLocals": true
    },
    "sourceMap": false
  }
]
```

It is not very difficult to understand the JSON available in the **bundleconfig.json** file. There is a JSON object for each bundle we want to configure and there are some options available related to each bundle e.g. **outputFileName**, **inputFiles** etc.

The default template has a single bundle configuration for custom JavaScript **wwwroot/js/site.js** and a single bundle configuration for custom stylesheet **wwwroot/css/site.css** files.

```
{
    "outputFileName": "wwwroot/css/site.min.css",
    "inputFiles": [
      "wwwroot/lib/bootstrap/dist/css/bootstrap.css",
      "wwwroot/css/site.css"
    ]
}
```

Some of the common options available are the following:

outputFileName: This is a required option and this specifies the name of the output bundle file. It can contain a relative path from the bundleconfig.json file.

inputFiles: This is an array of files we want to bundle together and it can also contain relative paths from the **bundleconfig.json** file. This option is not required so if you will not specify any input file, an empty output file will be generated.

minify: This is another optional setting and it specifies whether we want to enable minification or not. The default value is enabled: true.

sourceMap: This option indicates whether a source map for the bundled file should be generated or not. The default value is false.

sourceMapRootPath: The root path for storing the generated source map file.

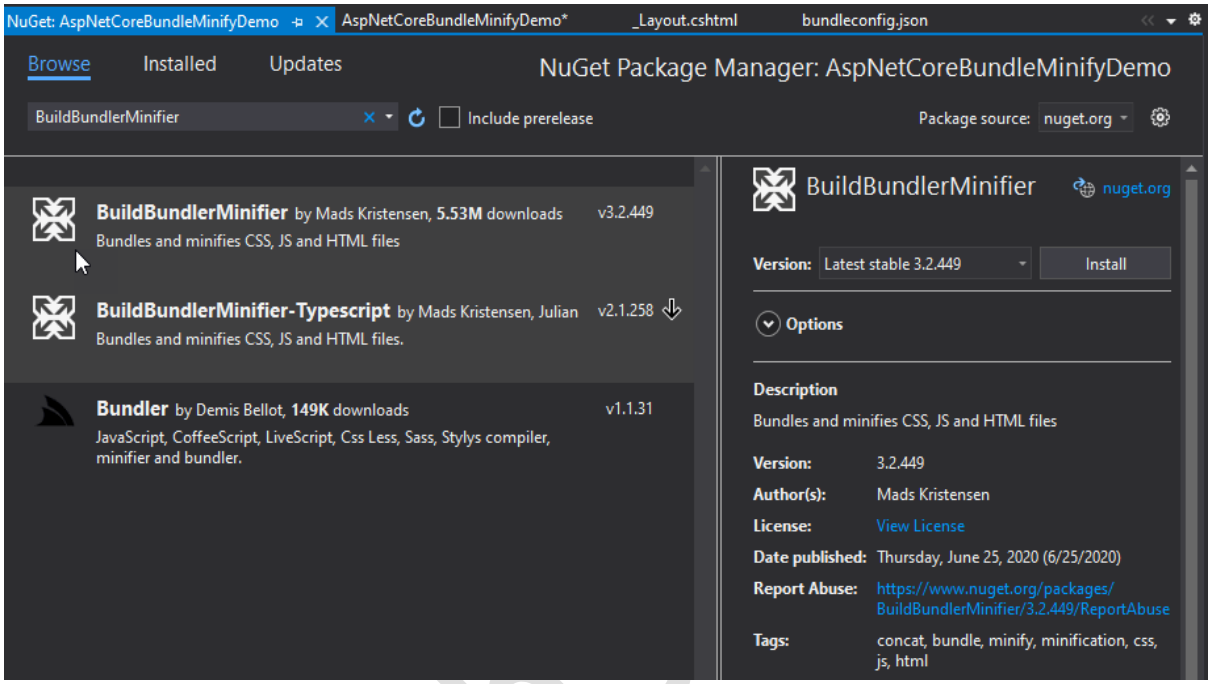
includeInProject: This option specifies whether to add generated files to the project file. The default value is false.

Let's modify the default template and include both bootstrap and jQuery into the generated bundle files. I have also renamed the output file name from **site** to **main**.

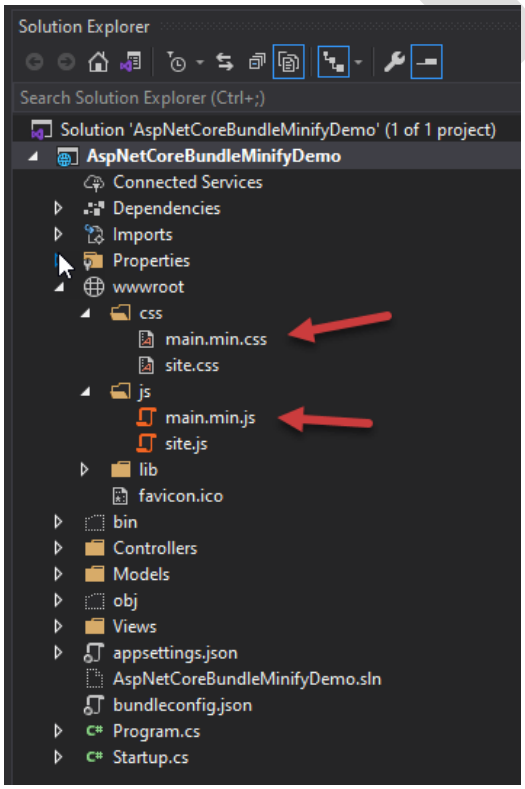
```
[
  {
    "outputFileName": "wwwroot/css/main.min.css",
    "inputFiles": [
      "wwwroot/lib/bootstrap/dist/css/bootstrap.css",
      "wwwroot/css/site.css"
    ]
  },
  {
    "outputFileName": "wwwroot/js/main.min.js",
    "inputFiles": [
      "wwwroot/lib/jquery/dist/jquery.js",
      "wwwroot/lib/bootstrap/dist/js/bootstrap.js",
      "wwwroot/js/site.js"
    ],
    "minify": {
      "enabled": true,
      "renameLocals": true
    },
    "sourceMap": false
  }
]
```

Bundling and Minification using BuildBundlerMinifier Package

In the previous section, we added the bundleconfig.json file and configure our bundles in this JSON file. If you will build your project in Visual Studio, nothing will happen and no bundles will be generated. This is because you need to install a NuGet package called **BuildBundlerMinifier** to let Visual Studio and .NET know that they need to build your bundles. You can install this package from the NuGet package manager available within Visual Studio.



Once the package is installed, build your project and you will now see your bundled and minified files generated in the folder specified in bundleconfig.json.



Environment-based Bundling and Minification

You may like to use the non-bundles and non-minified files in the development environment because they will make your life much easier during debugging. As a best practice, you should use the bundled files only in the production environment. You can specify which file to use in which environment by using Environment Tag Helper.

The Environment Tag Helper only renders its contents when the application is running in specific environments. The following code snippet shows how to use environment tag helper to render non-bundled files in the Development environment and bundled files when the environment is not Development such as Production or Staging. You can add the following environment tag helper in _Layout.cshtml file.

```
<environment include="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>

    <environment exclude="Development">
        <link rel="stylesheet" href="~/css/main.min.css" asp-append-version="true" />
    </environment>
```

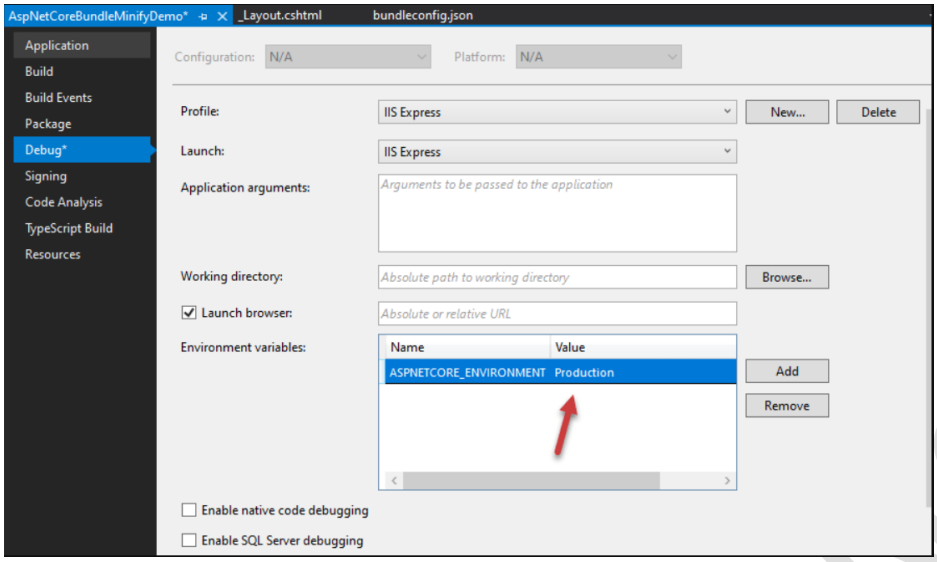
We can use same environment tag helper to render our JavaScript files as follows:

```
<environment include="Development">
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
</environment>

<environment exclude="Development">
    <link rel="stylesheet" href="~/js/main.min.js" asp-append-version="true" />
</environment>
```

If you will build and run your project now, you will see the following CSS and JavaScript files will be included on your website. This is because you are currently in the Development environment.

```
<link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css" />
<link rel="stylesheet" href="/css/site.css" />
...
...
...
<script src="/lib/jquery/dist/jquery.js"></script>
    <script src="/lib/bootstrap/dist/js/bootstrap.js"></script>
    <script src="/js/site.js?v=4q1jwFhaPaZgr8"></script>
```

You can change your environment from Development to Production using the project properties dialog as shown in the following screenshot.



If you will run your project again, you will see the following bundled and minified files are included on your web page.

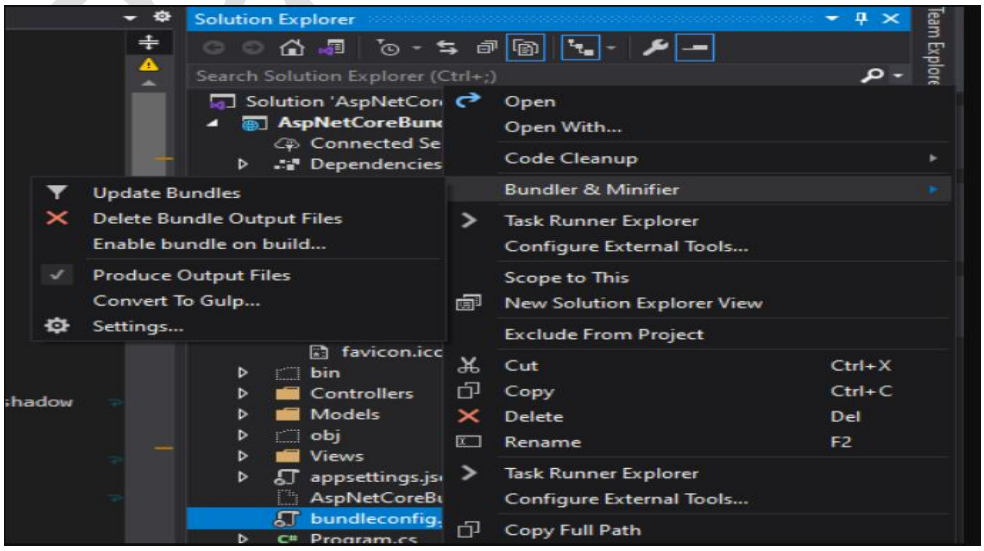
```
<link rel="stylesheet" href="/css/main.min.css?v=GEO9YOMPwE4" />
...
...
...
<link rel="stylesheet" href="/js/main.min.js?v=AHmIHyQd" />
```

Using Bundler & Minifier Visual Studio Extension

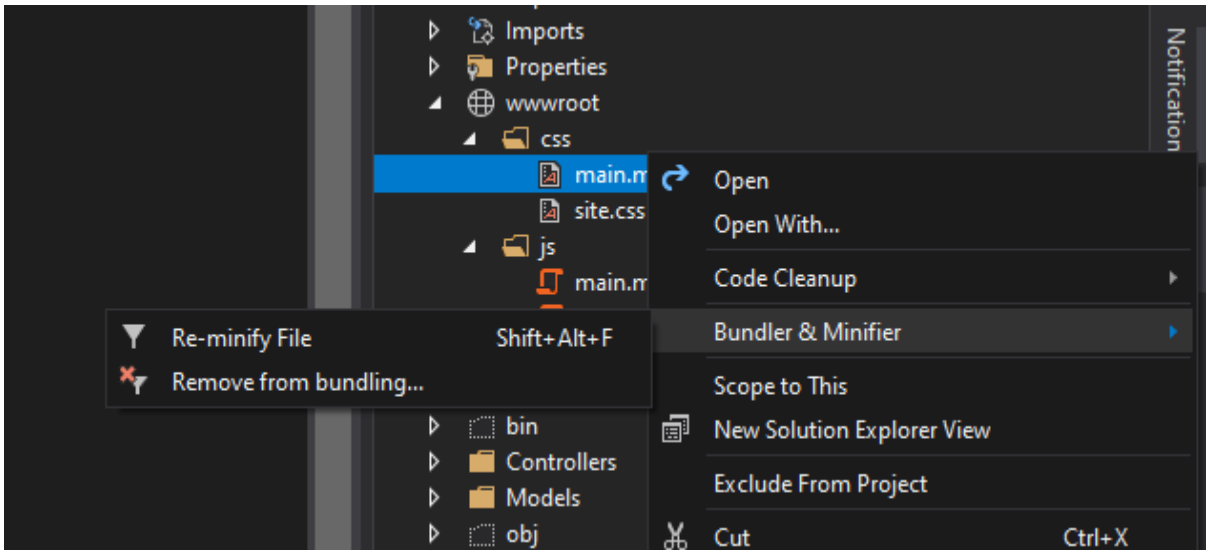
If you are using Visual Studio, it is a good idea to automate the bundling and minification process using a Visual Studio extension Bundler & Minifier developed by Mads Kristensen. This extension allows us to select and bundle files without writing a single line of code. You can download and install this extension from Visual Studio Marketplace. You may be asked to close Visual Studio and re-open it during the installation of the extension.

Please note that if you want to use this extension then you don't need the BuildBundlerMinifier NuGet package anymore. You can uninstall the BuildBundlerMinifier package from your project.

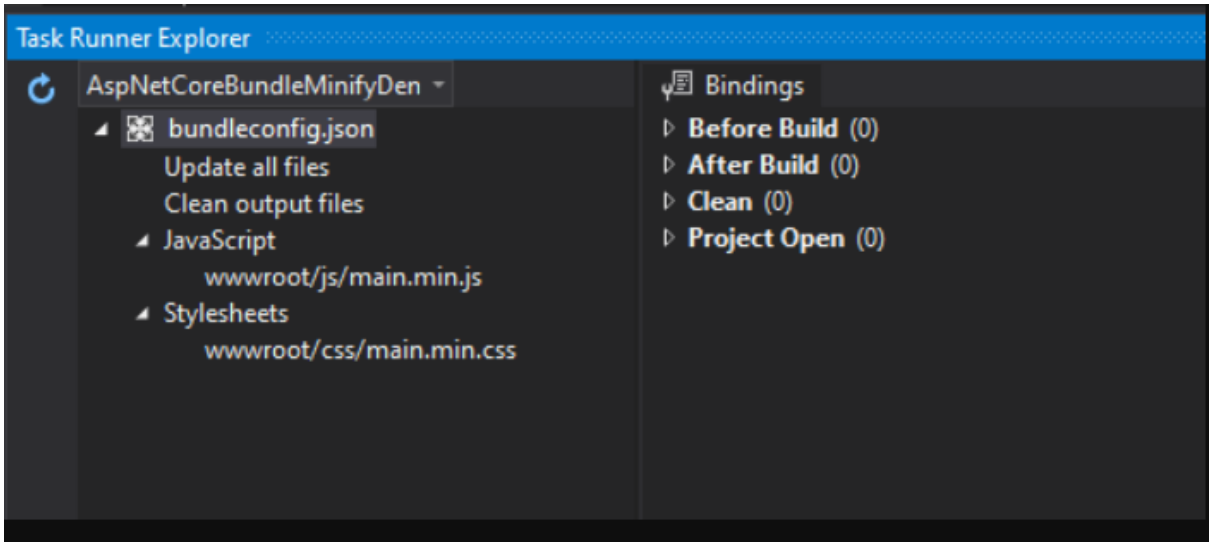
If you will right-click on the bundleconfig.json file, you will see some new "Bundler & Minifier" related options available to you. You can delete Bundle Output Files, Update Bundles, etc.



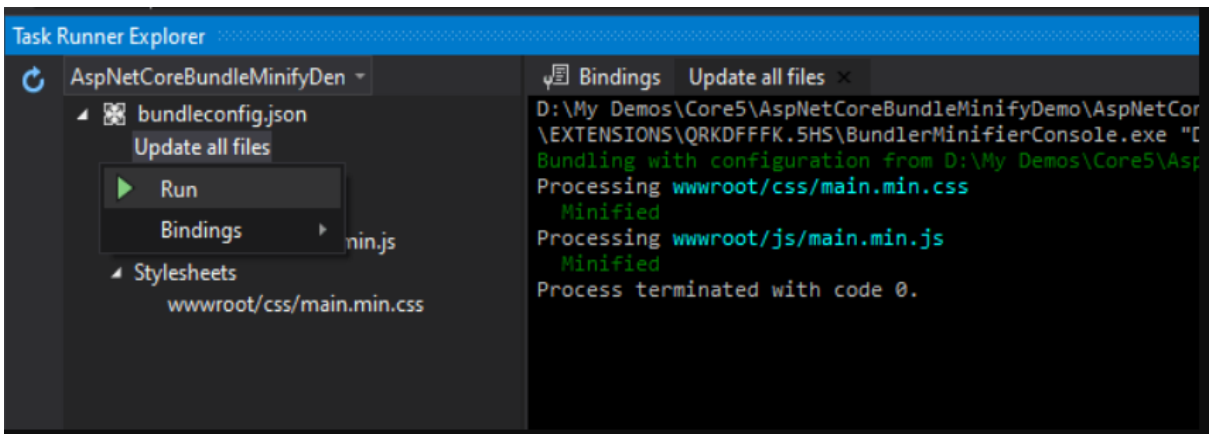
Similarly, if you will select and right-click on a single bundled file such as **main.min.css**, you should see the option to "**Re-minify File**" or an option to remove the file from bundling.



This extension will also make your bundleconfig.json file recognizable in Task Runner Explorer. This will allow us to run bundling from Task Runner Explorer manually or we can trigger bundling when certain Visual Studio events occur such as Before or After Build.



To execute the bundle from Task Runner Explorer, right-click on "**Update all files**" option and click "**Run**" option and you will see the bundled files are generated in your project Solution Explorer.



ASP.NET Core WebOptimizer

WebOptimizer is an ASP.NET Core middleware for bundling and minification of CSS and JavaScript files at runtime.

It supports full server-side and client-side caching to ensure high performance and takes away all the complicated build process for bundling and minifying. It sets up a pipeline for static files so they can be transformed (minified, bundled, etc.) before sent to the browser. This pipeline is flexible enough to combine many transformations to the same files. The pipeline is set up when the ASP.NET web application starts, but no output is being generated until the first time they are requested by the browser. The output is then being stored in memory and served very fast on all subsequent requests. This also means that no output files are being generated on disk.

Steps to configure ASP.NET Core Web Optimizer

Install the **LigerShark.WebOptimizer.Core** NuGet package.

Use the below command:

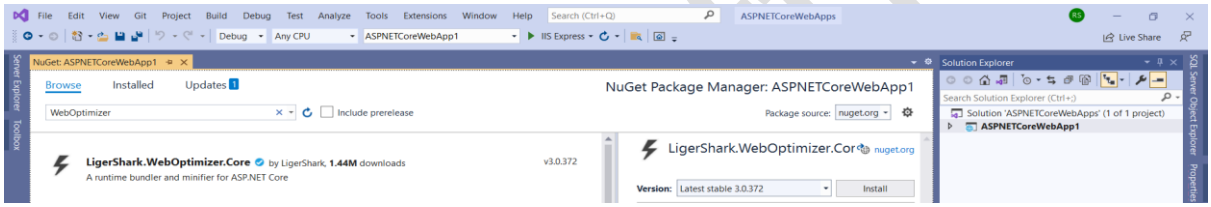
Package Manager Console:

```
Install-Package LigerShark.WebOptimizer.Core
```

.NET CLI:

```
dotnet add package LigerShark.WebOptimizer.Core
```

Or from the NuGet Package Manager window:



Then in **Startup.cs**, add app.UseWebOptimizer() to the Configure method anywhere before app.UseStaticFiles (if present), like so:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
   if (env.IsDevelopment())
   {
      app.UseDeveloperExceptionPage();
   }

   app.UseWebOptimizer();

   app.UseStaticFiles();

   ……
   ……
}
```

That sets up the middleware that handles the requests and transformation of the files.

And finally modify the *ConfigureServices* method by adding a call to services.AddWebOptimizer():

```
public void ConfigureServices(IServiceCollection services)
{
   ……..
   services.AddWebOptimizer();
}
```

The service contains all the configuration used by the middleware and allows your app to interact with it as well.

That's it. You have now enabled automatic CSS and JavaScript minification. No other code changes are needed for enabling this.

Try it by requesting one of your .css or .js files in the browser and see if it has been minified.

Disabling minification:

If you want to disable minification (e.g. in development), the following overload for AddWebOptimizer() can be used:

```
if (env.IsDevelopment())
{
   services.AddWebOptimizer(minifyJavaScript:false,minifyCss:false);
}
```

Minification

To control the minification in more detail, we must interact with the pipeline that manipulates the file content.

Minification is the process of removing all unnecessary characters from source code without changing its functionality in order to make it as small as possible.

For example, perhaps we only want a few certain JavaScript files to be minified automatically. Then we would write something like this:

```
public void ConfigureServices(IServiceCollection services)
{
   ……..
   services.AddWebOptimizer(pipeline =>
   {
      pipeline.MinifyJsFiles("js/Demo1.js", "js/Demo2.js", "js/Demo3.js");
   });
}
```

Notice that the paths to the .js files are relative to the wwwroot folder.

We can do the same for CSS, but this time we're using a globbing pattern to allow minification for all .css files in a particular folder and its sub folders:

```
pipeline.MinifyCssFiles("css/**/*.css");
```

When using globbing patterns, you still request the .css files on their relative path such as http://localhost:1234/css/site.css.

Setting up automatic minification like this doesn't require any other code changes in your web application to work.

Under the hood, WebOptimizer uses NUglify to minify JavaScript and CSS.

Bundling

To bundle multiple source file into a single output file couldn't be easier.

> Bundling is the process of taking multiple source files and combining them into a single output file. All CSS and JavaScript bundles are also being automatically minified.

Let's imagine we wanted to bundle /css/Demo1.css and /css/Demo2.css into a single output file and we want that output file to be located at http://localhost/css/bundle.css.

Then we would call the **AddCssBundle** method:

```
services.AddWebOptimizer(pipeline =>
{
   pipeline.AddCssBundle("/css/bundle.css", "css/Demo1.css", "css/Demo2.css");
});
```

The AddCssBundle method will combine the two source files in the order they are listed and then minify the resulting output file. The output file /css/bundle.css is created and kept in memory and not as a file on disk.

To bundle all files from a particular folder, we can use globbing patterns like this:

```
services.AddWebOptimizer(pipeline =>
{
   pipeline.AddCssBundle("/css/bundle.css", "css/**/*.css");
});
```

When using bundling, we have to update our <script> and <link> tags to point to the bundle route. It could look like this:

```
<link rel="stylesheet" href="/css/bundle.css" />
```

Content Root vs. Web Root

By default, all bundle source files are relative to the Web Root (*wwwroot*) folder, but you can change it to be relative to the Content Root instead.

The Content Root folder is usually the project root directory, which is the parent directory of *wwwroot*.

As an example, let's create a bundle of files found in a folder called styles that exist in the Content Root:

```
services.AddWebOptimizer(pipeline =>
{
   pipeline.AddCssBundle("/css/bundle.css", "styles/*.css")
           .UseContentRoot();
});
```

The UseContentRoot() method makes the bundle look for source files in the Content Root rather than in the Web Root.

To use a completely custom IFileProvider, you can use the UseFileProvider pipeline method.

```
services.AddWebOptimizer(pipeline =>
{
   var provider = new Microsoft.Extensions.FileProviders.PhysicalFileProvider(@"D:\folder");
   pipeline.AddJavaScriptBundle("/js/scripts.js", "Demo1.js", "Demo2.js")
           .UseFileProvider(provider);
});
```

Tag Helpers

WebOptimizer ships with a few Tag Helpers that helps with a few important tasks.

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files.

First, we need to register the TagHelpers defined in *LigerShark.WebOptimizer.Core* in our project.

To do that, go to _ViewImports.cshtml and register the Tag Helpers by adding @addTagHelper *, WebOptimizer.Core to the file.

```
@addTagHelper *, WebOptimizer.Core
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Cache busting

As soon as the Tag Helpers are registered in your project, you'll notice how the <script> and <link> tags starts to render a little differently when they are referencing a file or bundle.

NOTE: Unlike other ASP.NET Core Tag Helpers, <script> and <link> tags don't need an asp- attribute to be rendered as a Tag Helper.

They will get a version string added as a URL parameter:

<link rel="stylesheet" href="/css/bundle.css?v=UabjmJ2eJZ4ZDgkAMgLEuXCZEBI" />

This version string changes every time one or more of the source files are modified.

This technique is called *cache busting* and is a critical component to achieving high performance, since we cannot utilize browser caching of the CSS and JavaScript files without it. That is also why it cannot be disabled when using WebOptimizer.

Inlining content

We can also use Web Optimizer to inline the content of the files directly into the Razor page. This is useful for creating high-performance websites that inlines the above-the-fold CSS and lazy loads the rest later.

To do this, simply add the attribute inline to any <link> or <script> element like so:

<link rel="stylesheet" href="/css/bundle.css" inline />

<script src="/js/bundle.js" inline></script>

There is a Tag Helper that understands what the inline attribute means and handles the inlining automatically.

Compiling Scss

WebOptimizer can also compile Scss files into CSS.

For that you need to install the **LigerShark.WebOptimizer.Sass** NuGet package.

This package compiles Sass/Scss into CSS by hooking into the LigerShark.WebOptimizer pipeline.

Add the NuGet package LigerShark.WebOptimizer.Sass to any ASP.NET Core project supporting .NET Standard 2.0 or higher.

Install

Use the below command:

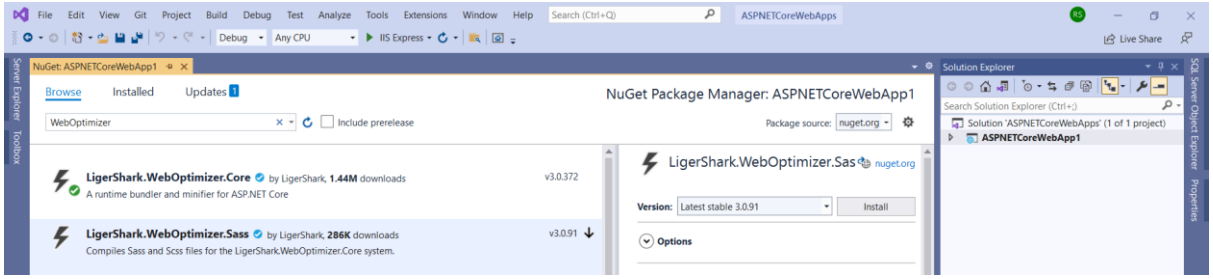
Package Manager Console:

```
Install-Package LigerShark.WebOptimizer.Sass
```

.NET CLI:

```
dotnet add package LigerShark.WebOptimizer.Sass
```

Or from the NuGet Package Manager window:



Usage

Here's an example of how to compile Demo1.scss and Demo2.scss from inside the wwwroot folder and bundle them into a single .css file called /Demo.css:

In Startup.cs, modify the ConfigureServices method:

```
public void ConfigureServices(IServiceCollection services)
{
  ……….
  services.AddWebOptimizer(pipeline =>
  {
    pipeline.AddScssBundle("/Demo.css", "Demo1.scss", "Demo2.scss");
  });
}
```

And add app.UseWebOptimizer() to the Configure method anywhere before app.UseStaticFiles as follows:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseWebOptimizer();

    app.UseStaticFiles();

    ……..
}
```

Now the path /Demo.css will return a compiled, bundled and minified CSS document based on the two source files.

You can also reference any .scss files directly in the browser (/Demo1.scss) and a compiled and minified CSS document will be served. To set that up, do this:

```
services.AddWebOptimizer(pipeline =>
{
  pipeline.CompileScssFiles();
});
```

Or if you just want to limit what .scss files will be compiled, do this:

```
services.AddWebOptimizer(pipeline =>
{
  pipeline.CompileScssFiles("/path/file1.scss", "/path/file2.scss");
});
```

Setup TagHelpers

In _ViewImports.cshtml register the TagHelpers by adding @addTagHelper *, WebOptimizer.Core to the file. It may look something like this:

@addTagHelper *, WebOptimizer.Core

@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers

ASP.NET Core Bundling and Minification Using Gulp

In a typical modern web application, the build process might:

- Bundle and minify JavaScript and CSS files.
- Run tools to call the bundling and minification tasks before each build.
- Compile LESS or SASS files to CSS.
- Compile CoffeeScript or TypeScript files to JavaScript.

A *task runner* is a tool which automates these routine development tasks and more. Visual Studio provides built-in support for two popular JavaScript-based task runners: Gulp and Grunt.

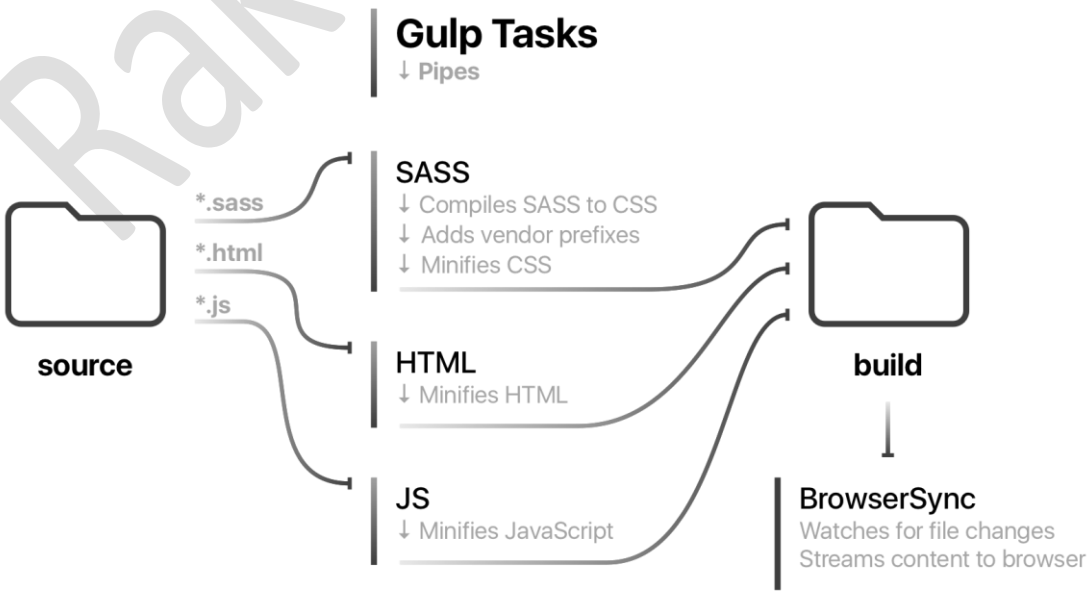
Here we will discuss how to use **Gulp** in Visual Studio and how to use Gulp tasks to automate bundling and minification in Visual Studio.

What is Gulp?

Gulp is a JavaScript-based streaming build toolkit for client-side code. It is commonly used to stream client-side files through a series of processes when a specific event is triggered in a build environment. Some advantages of using Gulp include the automation of common development tasks, the simplification of repetitive tasks, and a decrease in overall development time. For instance, Gulp can be used to automate bundling and minification or the cleansing of a development environment before a new build.

- **Automation** - gulp is a toolkit that helps you automate painful or time-consuming tasks in your development workflow.
- **Platform-agnostic** - Integrations are built into all major IDEs and people are using gulp with PHP, .NET, Node.js, Java, and other platforms.
- **Strong Ecosystem** - Use npm modules to do anything you want + over 3000 curated plugins for streaming file transformations.
- **Simple** - By providing only a minimal API surface, gulp is easy to learn and simple to use.

Here, we will use the latest version of Gulp 4.0 that is rewritten from the ground-up and will allow us to compose tasks using new **series()** and **parallel()** methods.



The gulp tasks we needed to perform bundling and minification are as follows:

1. We need a gulp task that will delete existing bundled or minified files.
2. We need a gulp task that will read JavaScript files from a source folder and will bundle them together, minifies the bundled file, and finally save it to a target folder.
3. We need a gulp task that will read CSS files from a source folder and will bundle them together, minifies the bundled file, and finally save it to a target folder.

We will use the following plugins to perform the above tasks.

- del – Deletes files and directories
- gulp-concat – Concatenates files
- gulp-cssmin – Minifies CSS files
- gulp-uglify – Minifies JavaScript files
- merge-streams – Merges multiple streams

There are thousands of plugins available to perform all types of routine tasks using gulp. You can search these plugins in npm packages repository.

What is Gulpfile.js?

So far, we have learned that we need to define gulp tasks using JavaScript and we have to use different plugins to perform different routine tasks now the last question is where we will define these tasks. This is where **Gulpfile.js** comes in. A Gulpfile.js or gulpfile.js is a file in your project directory that automatically loads when we run the **gulp** command. In this file, we often define tasks and use gulp APIs like src(), dest(), series(), parallel() etc. We will also create this file shortly to define our gulp tasks.

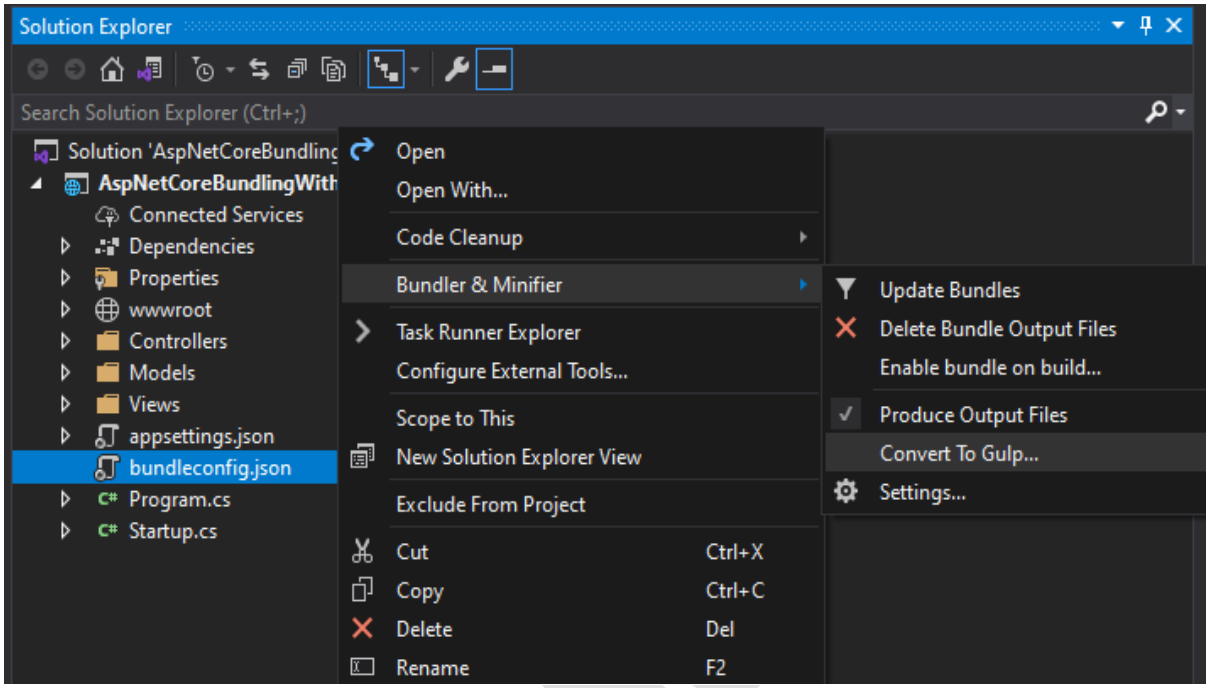
Converting bundleconfig.json to Gulp

In the earlier, we created a bundleconfig.json file for bundling and minification and we also installed a Visual Studio extension called **BundlerMinifier**. We can reuse the same bundleconfig.json file from Gulp but you have to remove some extra options from the bundleconfig.json file. The following bundleconfig.json file is a trim down version we need here. It is simply defining some CSS and JavaScript bundles with source and destination paths.

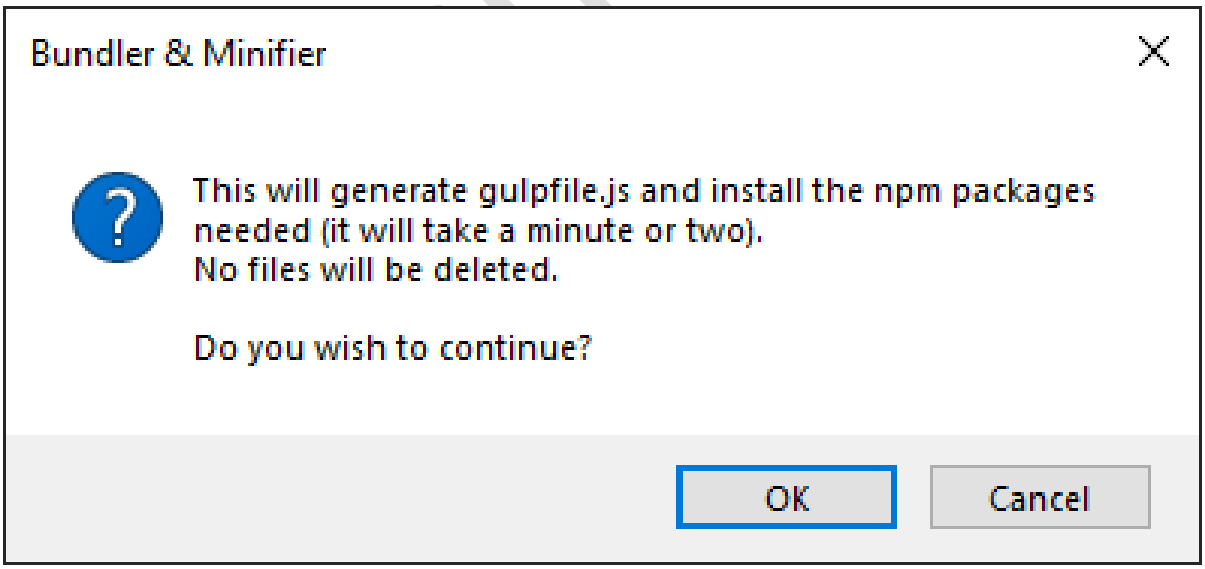
```
[
  {
    "outputFileName": "wwwroot/css/main.min.css",
    "inputFiles": [
      "wwwroot/lib/bootstrap/dist/css/bootstrap.css",
      "wwwroot/css/site.css"
    ]
  },
  {
    "outputFileName": "wwwroot/js/main.min.js",
    "inputFiles": [
      "wwwroot/lib/jquery/dist/jquery.js",
      "wwwroot/lib/bootstrap/dist/js/bootstrap.js",
      "wwwroot/js/site.js"
    ]
  }
]
```

The first step to start using Gulp in your ASP.NET Core project is to install Gulp and related npm modules and create a Gulpfile.js. We can do all this easily by using Bundler & Minifier "**Convert To Gulp**" option.

Right-click on the bundleconfig.json file, choose Bundler & Minifier, and then choose "Convert To Gulp…" option from the menu as shown below.

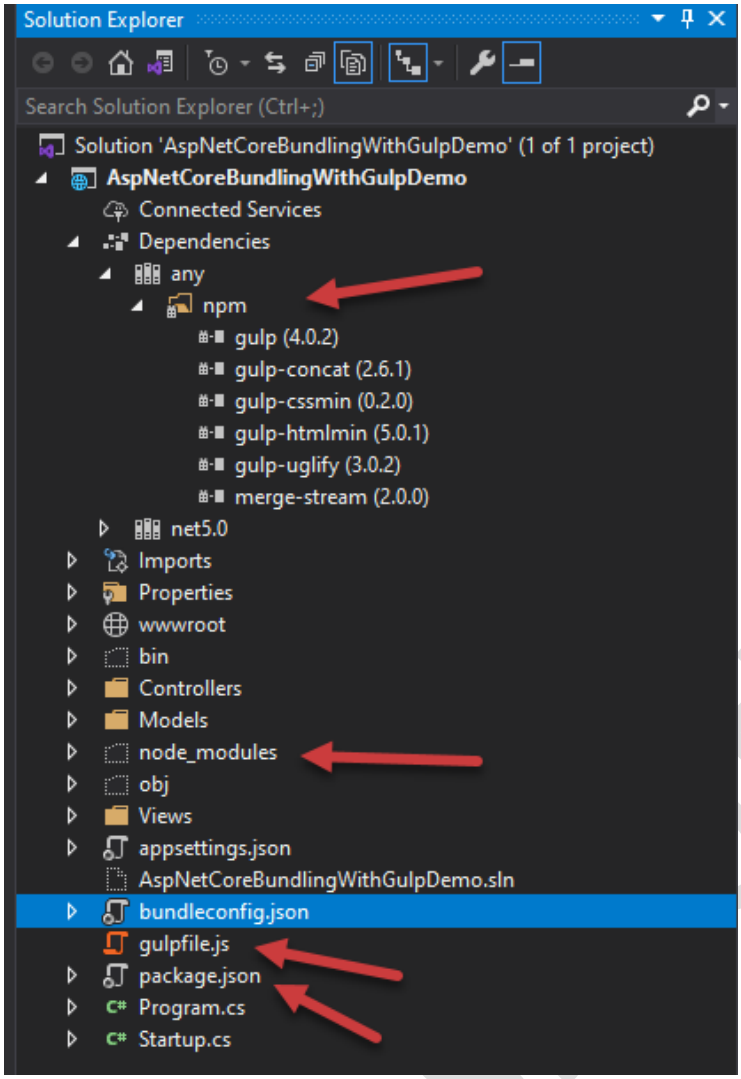


You will see the following confirmation message asking you about generating gulpfile.js and installing npm packages. You can press OK to proceed.



After few minutes of downloading, you will see the following new items in your solution explorer.

1. A gulpfile.js
2. A package.json
3. Some npm packages are added in Dependencies
4. A node_modules folder is added in which many node packages are downloaded and installed



A package.json file is a JSON file that is normally available at the root of a project and it contains the metadata information related to the project. It is used to manage the project's dependencies, scripts, versions, etc. If you will open the generated package.json file you will notice that it has project name, version, and some development related dependencies which are mainly npm packages we will use in gulpfile.js file shortly.

```
{
  "name": "app",
  "version": "1.0.0",
  "private": true,
  "devDependencies": {
    "del": "^6.0.0",
    "gulp": "^4.0.2",
    "gulp-concat": "^2.6.1",
    "gulp-cssmin": "^0.2.0",
    "gulp-htmlmin": "^5.0.1",
    "gulp-uglify": "^3.0.2",
    "merge-stream": "^2.0.0"
  }
}
```

How to Define Tasks in Gulp

If you will open gulpfile.js, you will notice that the Bundler & Minifier extension has already generated all the gulp tasks for you. The problem with this generated code is that it is not using Gulp 4.0 even though the package.json file is showing gulp version 4.0.2 in it. This means you have two options now.

1. You can downgrade your Gulp version in package.json and continue using the generated gulp tasks available in gulpfile.js file
2. You can use the latest Gulp version 4.0.2 and update gulpfile.js code to use the latest Gulp features.

I have chosen option 2 above as it's always a good option to use the latest version of any tool or framework.

It is now time to update gulpfile.js and learn more about gulp. First of all, we need to specify/import all the required npm packages. These packages can be loaded either by path or name and each of them will be assigned to a variable. This is similar to importing NuGet packages in C# files while working on the server-side.

We are loading the main '**gulp**' package that contains all gulp related methods and features. Next, we are loading all npm packages. Finally, we are saving the reference of bundleconfig.json file in a local variable bundleconfig so that we can read input and output file paths from bundleconfig.json file.

```
var gulp = require("gulp"),
    concat = require("gulp-concat"),
    cssmin = require("gulp-cssmin"),
    uglify = require("gulp-uglify"),
    merge = require("merge-stream"),
    del = require("del"),
    bundleconfig = require("./bundleconfig.json");
```

Next, we have a regex variable that will keep both css and js files regex which are used in **getBundles** function to filter and fetch files matching with a specific regex pattern.

```
var regex = {
    css: /\.css$/,
    js: /\.js$/
};

function getBundles(regexPattern) {
    return bundleconfig.filter(function (bundle) {
        return regexPattern.test(bundle.outputFileName);
    });
}
```

Clean Task

Next, we need to define the '**clean**' gulp task. The clean task will remove the existing minified files so that they can be regenerated with the updated code or functionality. Notice that the gulp tasks are simply JavaScript functions. We create JavaScript functions and then registered them by using **gulp.task()** method as we did with our clean function below.

```
function clean() {
    var files = bundleconfig.map(function (bundle) {
        return bundle.outputFileName;
    });

    return del(files);
}
gulp.task(clean);
```

Minify JS Task

To minify JavaScript files, we are creating minJs task. This task

1. Load all JavaScript files
2. Concatenates or bundle all files using '**concat**' module
3. Minify the bundled file using '**ugligy**' module
4. Place the resulting minified file at the root location

```
function minJs() {
    var tasks = getBundles(regex.js).map(function (bundle) {
        return gulp.src(bundle.inputFiles, { base: "." })
            .pipe(concat(bundle.outputFileName))
            .pipe(uglify())
            .pipe(gulp.dest("."));
    });
    return merge(tasks);
}
gulp.task(minJs);
```

Minify CSS Task

To minify CSS files, we are creating minCss task. This task

1. Load all CSS files
2. Concatenates or bundle all files using 'concat' module
3. Minify the bundled file using 'cssmin' module
4. Place the resulting minified file at the root location

```
function minCss() {
    var tasks = getBundles(regex.css).map(function (bundle) {
        return gulp.src(bundle.inputFiles, { base: "." })
            .pipe(concat(bundle.outputFileName))
            .pipe(cssmin())
            .pipe(gulp.dest("."));
    });
    return merge(tasks);
}
gulp.task(minCss);
```

At the end, we created a named task with the name 'min' and combine all task functions using gulp.series() method. The **gulp.series()** method executes multiple tasks one after another in a series. There is also another method called **gulp.parallel()** that can be used to execute multiple tasks simultaneously.

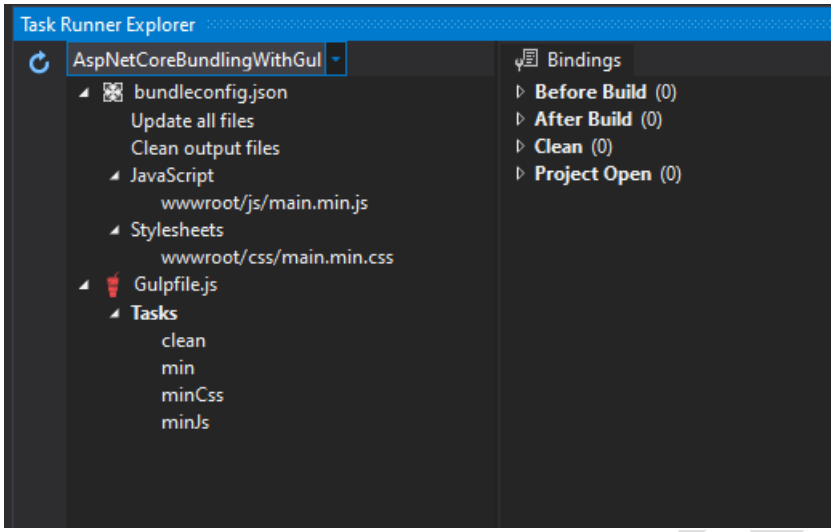
```
gulp.task("min", gulp.series(clean, minJs, minCss));
```

Following is the complete gulpfile.js file with all the tasks

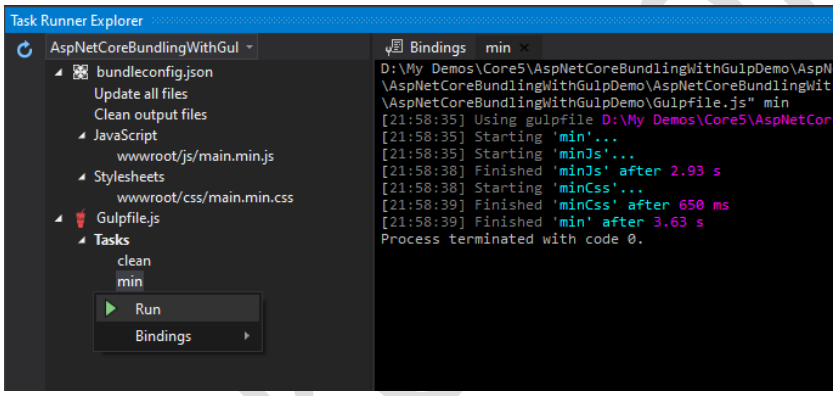
```
"use strict";

var gulp = require("gulp"),
    concat = require("gulp-concat"),
    cssmin = require("gulp-cssmin"),
    uglify = require("gulp-uglify"),
    merge = require("merge-stream"),
    del = require("del"),
    bundleconfig = require("./bundleconfig.json");

var regex = {
    css: /\.css$/,
    js: /\.js$/
};

function getBundles(regexPattern) {
    return bundleconfig.filter(function (bundle) {
        return regexPattern.test(bundle.outputFileName);
    });
}

function clean() {
    var files = bundleconfig.map(function (bundle) {
        return bundle.outputFileName;
    });

    return del(files);
}
gulp.task(clean);

function minJs() {
    var tasks = getBundles(regex.js).map(function (bundle) {
        return gulp.src(bundle.inputFiles, { base: "." })
            .pipe(concat(bundle.outputFileName))
            .pipe(uglify())
            .pipe(gulp.dest("."));
    });
    return merge(tasks);
}
gulp.task(minJs);

function minCss() {
    var tasks = getBundles(regex.css).map(function (bundle) {
        return gulp.src(bundle.inputFiles, { base: "." })
            .pipe(concat(bundle.outputFileName))
            .pipe(cssmin())
            .pipe(gulp.dest("."));
    });
    return merge(tasks);
}
gulp.task(minCss);

gulp.task("min", gulp.series(clean, minJs, minCss));
```

Running Gulp Tasks in Visual Studio

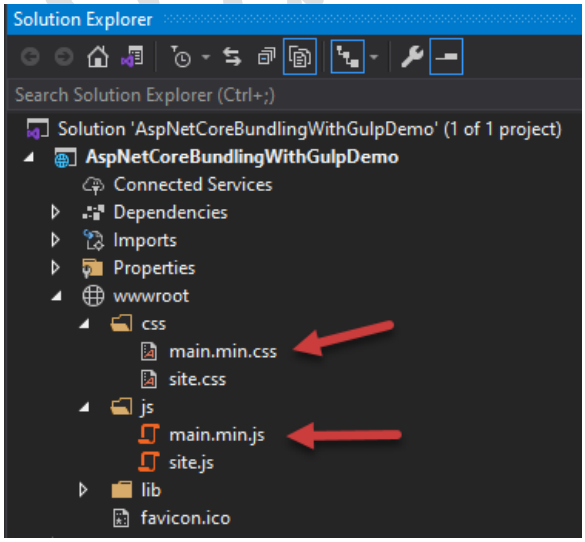
We are now ready to run our gulp tasks using Visual Studio. Open the Task Runner Explorer window and you will notice that it is reading Gulpfile.js and showing all gulp tasks.



You can run the '**min**' task by right-clicking it and selecting "**Run**". The 'min' task will be shown on the right-hand side and you can also view the logs which display when a particular task is finished and how much time a particular task took to complete.



You will see that the **main.min.css** and **main.min.js** files will be generated in Solution Explorer. You can also run individual tasks to perform a specific function. For example, you can run the 'clean' task to delete the minified files.



Binding Gulp Tasks with Visual Studio Events

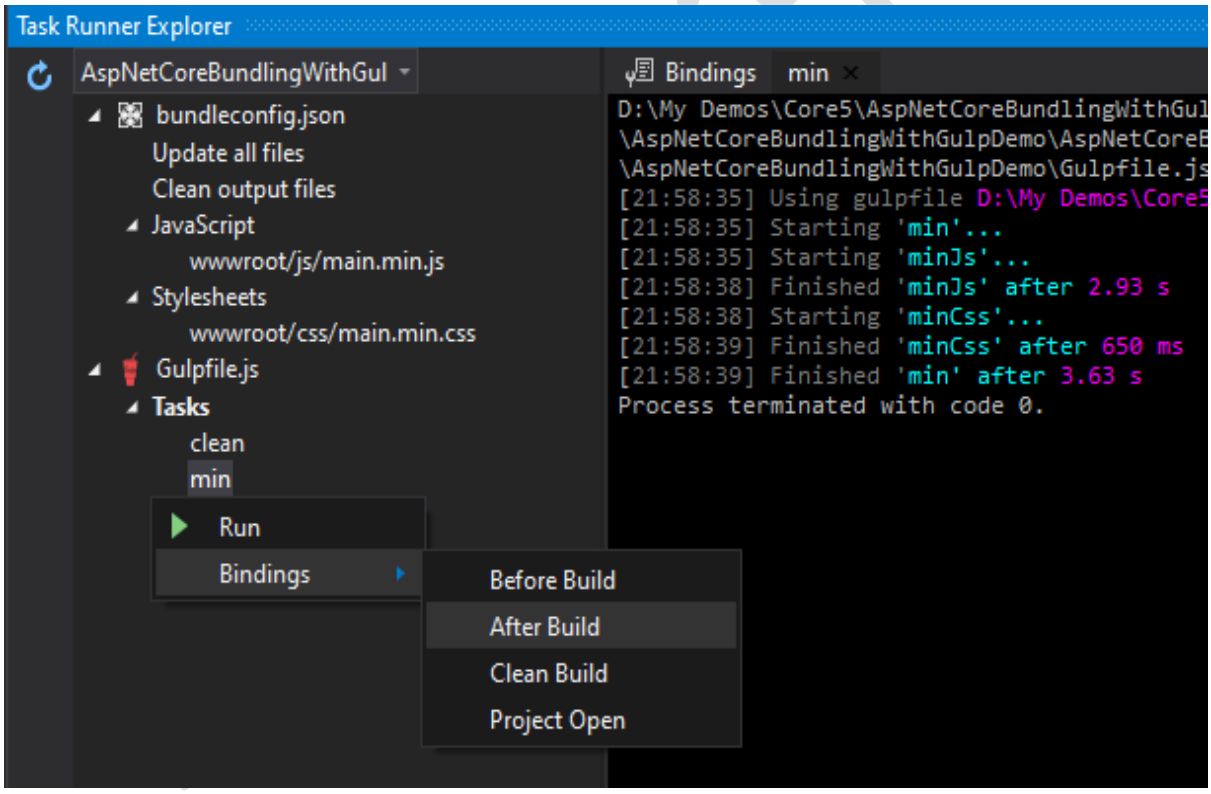
Running gulp tasks from Task Runner Explorer every time you make some changes in JavaScript or CSS files during development seems a non-productive method. You can automate this process by binding these tasks with certain Visual Studio events. You can

1. Bind any task with any event
2. Bind one task with multiple events
3. Bind multiple tasks with a single event

We can bind Gulpfile.js tasks with the following Visual Studio events:

- Before Build
- After Build
- Clean Build
- Project Open

To bind a task with an event, right-click on a task in Task Runner Explorer and choose any event from the child menu of the "Bindings" menu as shown below:



We can also define this binding directly in the gulpfile.js using the following syntax. The following line states that the '**min**' task should run when Visual Studio executes the **AfterBuild** event.

```
1    /// <binding AfterBuild='min' />
```

Working with Bundling & Minification using Gulp from Scratch:

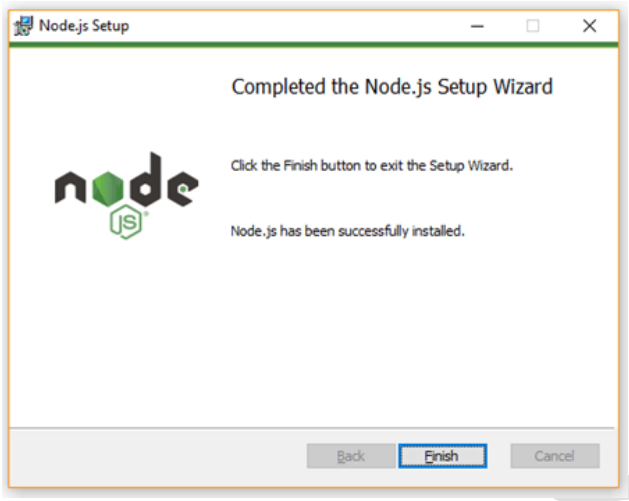
Gulp is a toolkit that helps web developers automate many development tasks such as bundling and minification. We can create tasks using NPM gulp dependencies.

Prerequisites

1. Install Node.js

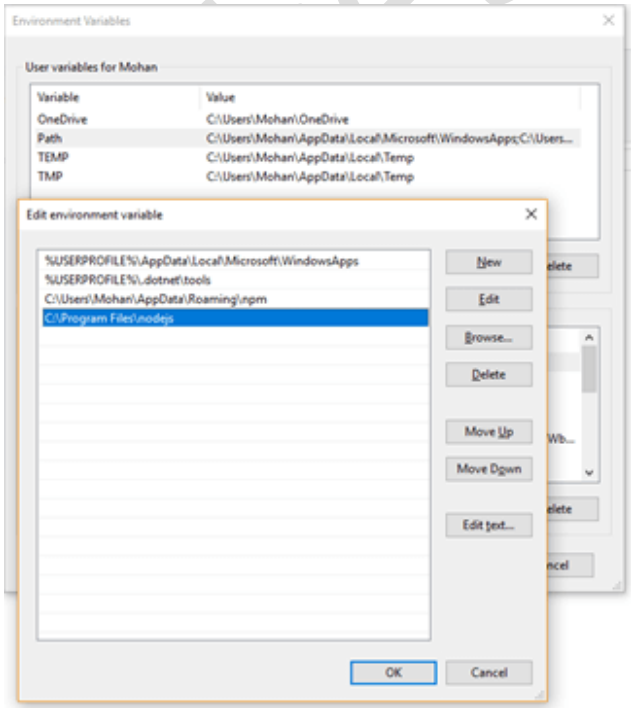
Node.js is an open-source and cross-platform runtime environment for creating server-side and networking apps. Node.js apps are written in JavaScript and can be run on macOS, Windows, and Linux.

We must install Node.js to execute our gulp tasks. So, download and install the recommended version of Node.js from its official website i.e. https://nodejs.org/en/ before implementing gulp.



Set Node.js path as an environment variable

After installing Node.js, set its installed location as a path in an environment variable. To do so, open Run (Win + R) and execute "**rundll32 sysdm.cpl, EditEnvironmentVariables**". This will open the environment variables dialog, in that open path by performing double-click on the **Path**. In the Edit environment variable dialog, click New and add the installed location of Node.js, which by default is **C:\Program Files\nodejs**.

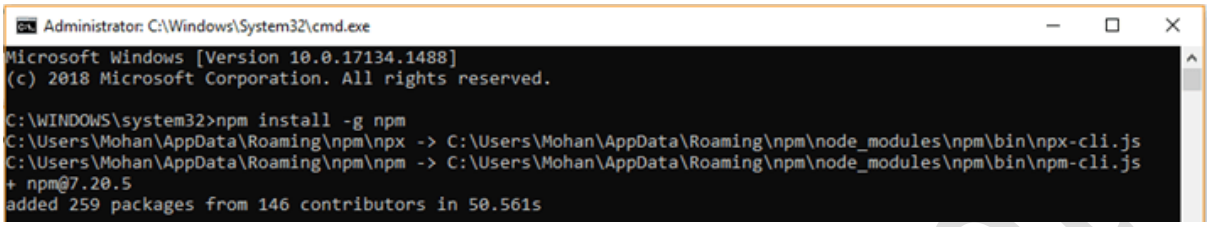


2. Install NPM

Node Package Manager (NPM) is the default package manager for the JavaScript runtime environment Node.js. It is a command-line utility for installing, updating, and uninstalling Node.js packages in your app.

Install NPM using the following command in the command prompt.

npm install -g npm



To check whether Node.js or NPM are already installed, execute the following commands.

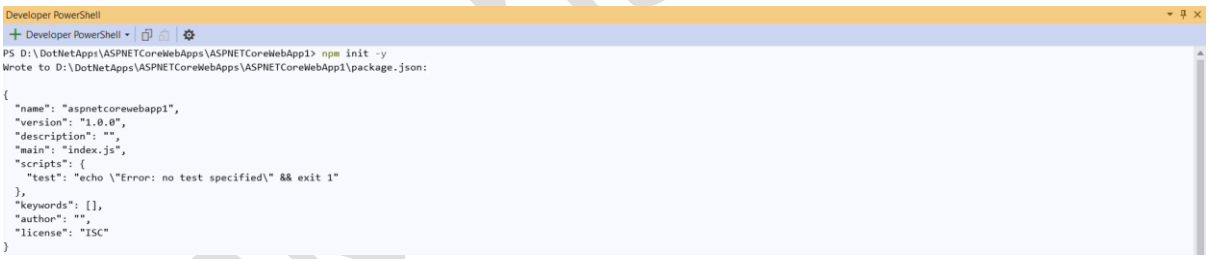
> Node.js: node --version
>
> NPM version: npm --version

3. Create package.json file

The package.json is a JSON file that contains information about the dependencies required in the project.

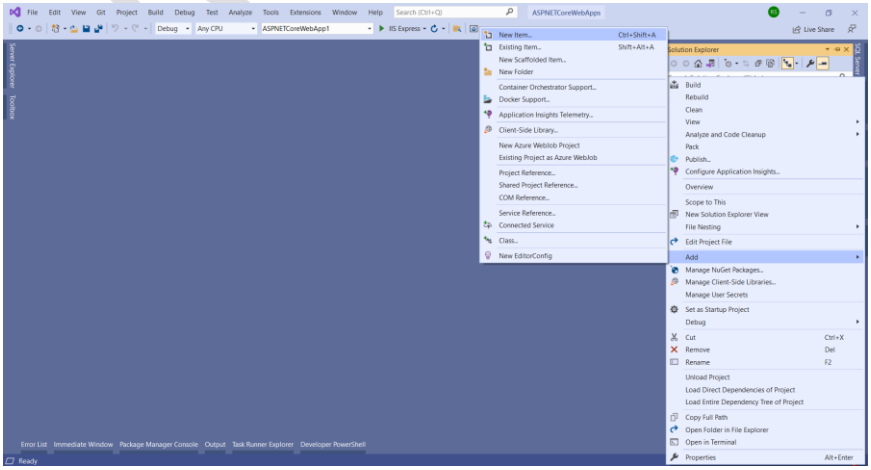
Create the package.json file by running the following command using the command prompt or terminal.

npm init –y

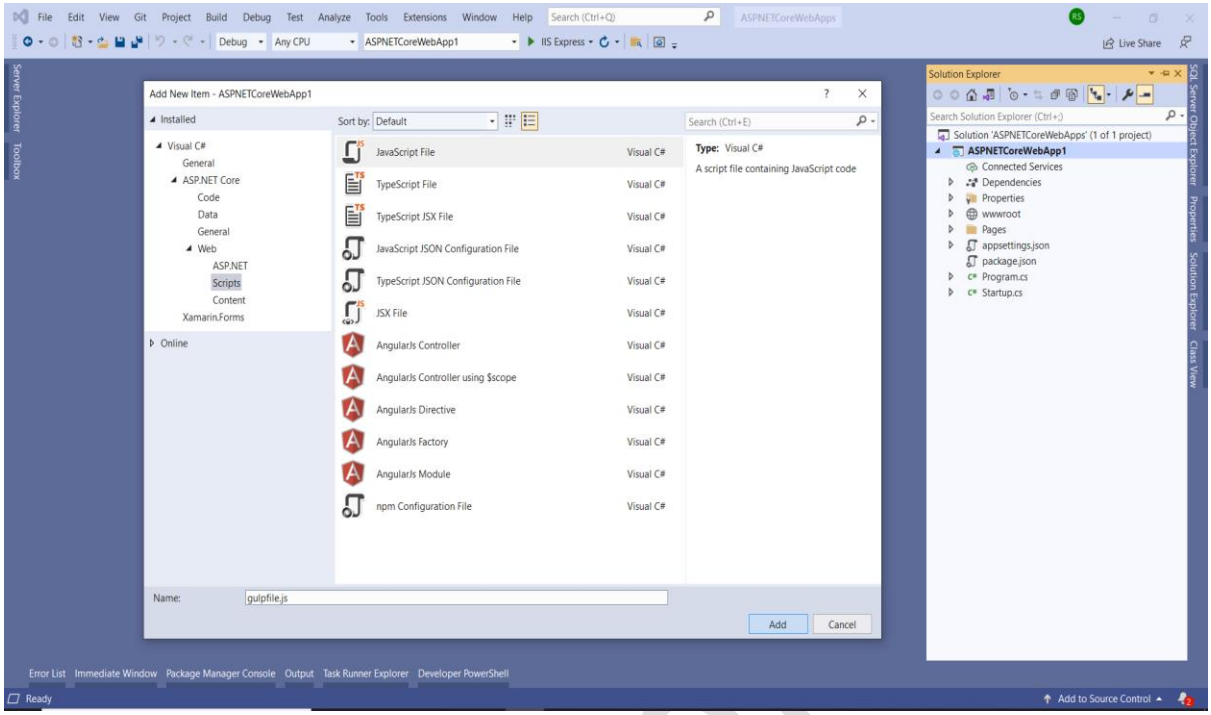


4. Add Gulp.js file

Open your app on Visual Studio and right-click on the startup project. Add a new item (Add -> New Item).



Choose the scripts option (**Installed -> Visual C# -> ASP.NET Core -> Web -> Scripts**), select **JavaScript File** and specify file name as **gulpfile.js** and finally click **Add** button to add it.



Note: The name of the gulp file must be gulpfile.js.

Install the Gulp CLI (command line utility)

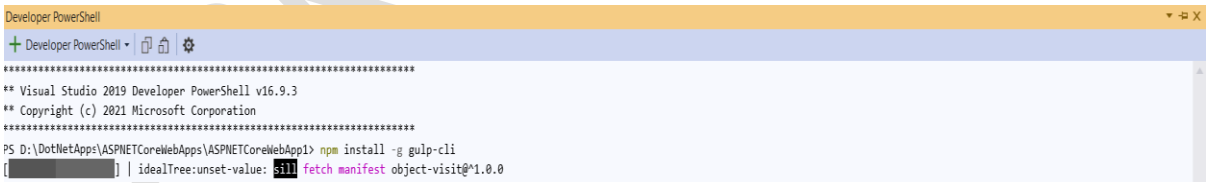
Gulp comes in two parts: the Gulp CLI (command line utility) that is installed globally on your computer, and then local Gulp packages installed in each individual project you might have.

This separation, introduced in Gulp v4 allows you to run different versions of Gulp in each local project.

Install Gulp CLI Globally:

To install the Gulp CLI globally, run the following command on the command prompt or terminal.

```
npm install –g gulp-cli
```

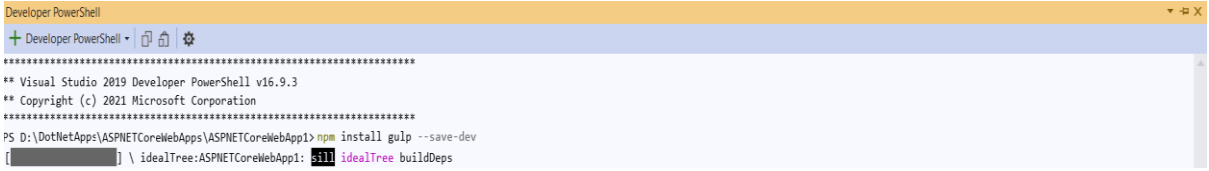


The -g flag means npm will install the package in the global npm directory, so you can run the gulp command from any directory.

Install Gulp into your local project:

To install gulp in your project, run the following command on the command prompt or terminal.

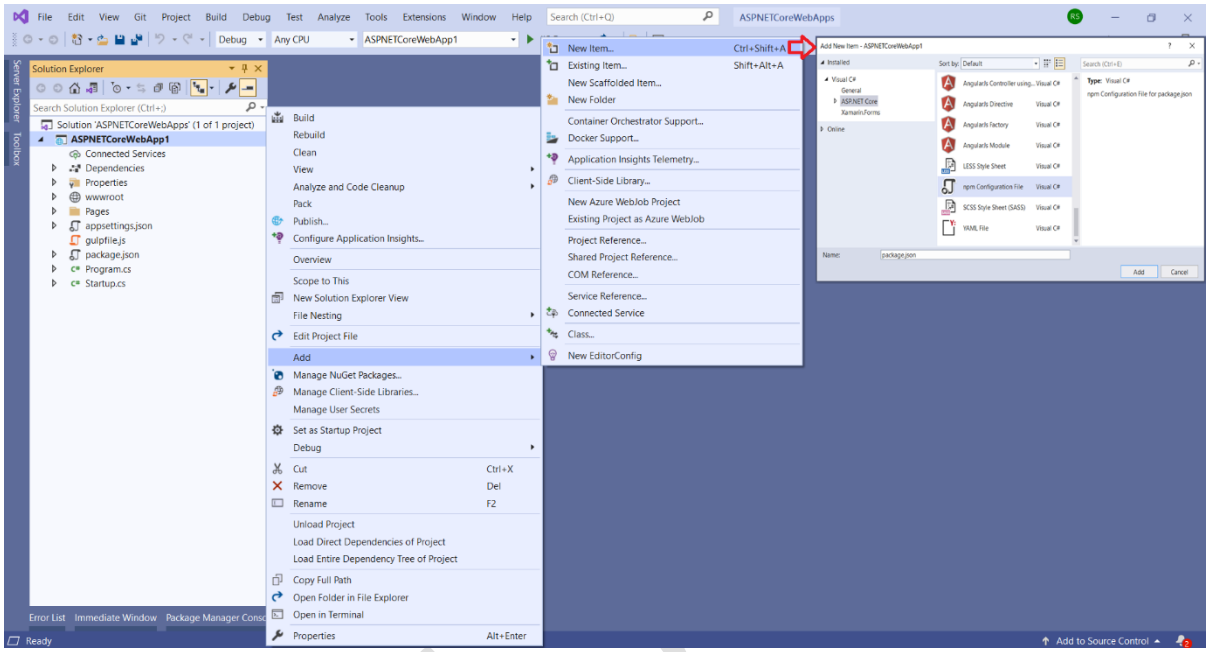
```
npm install gulp --save-dev
```



Bundling

Bundling is a process of combining multiple files into single file. It reduces the number of requests to the Server, which are required to load the page asset or resources. This only improves page load performance.

Using **gulp-concat** plugin, we can achieve Bundling. This plugin can be installed from npm (Node Package Manager). So, we need to add gulp-concat package to devDependencies section of our package.json file. If package.json file is not present in the project, we can add this file to the project.



Package.json

```
{
  "name": "aspnetcorewebapp1",
  "version": "1.0.0",
  "devDependencies": {
    "gulp": "^4.0.2",
    "gulp-concat": "2.6.1"
  }
}
```

Now, we need to run "npm install" command from command window, to install the specific version of packages. Visual Studio will install the missing packages automatically on build or when package.json modifies.

The next step is to import "**gulp-concat**" module to **gulpfile.js**. If **gulpfile.js** is not present in project, we can add it from new Item list.

```
var gulp = require("gulp"),
    concat = require("gulp-concat");
```

Now, we have to specify the files that we want to bundle. In this example, we have created "paths" array to specify all paths that need to be bundled.

```
var paths = {
  webroot: "./wwwroot/"
};

paths.js = paths.webroot + "js/**/*.js";
paths.concatJsDest = paths.webroot + "js/site.min.js";
paths.css = paths.webroot + "css/**/*.css";
paths.concatCssDest = paths.webroot + "css/site.min.css";
```

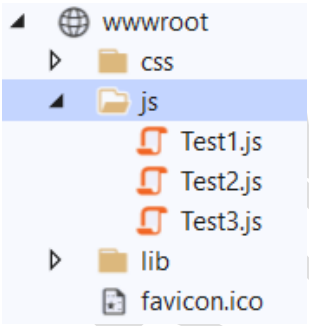
Next step is to define gulp tasks used for concatenating the desired files and create result file in destination folder.

```
gulp.task("bundleJS", function () {
   return gulp.src([paths.js])
      .pipe(concat(paths.concatJsDest))
      .pipe(gulp.dest("."));
});

gulp.task("bundleCSS", function () {
   return gulp.src([paths.css])
      .pipe(concat(paths.concatCssDest))
      .pipe(gulp.dest("."));
});
```

The gulp.src function emits a stream of files which are piped to gulp plugins. An array of globs is used to specify the files to emit using node-glob syntax. The glob which begins with "!" excludes matching files from the glob results.

```
gulp.task("bundleJS", function () {
   return gulp.src([paths.js] , "!" + paths.concatJsDest)
      .pipe(concat(paths.concatJsDest))
      .pipe(gulp.dest("."));
});

gulp.task("bundleCSS", function () {
   return gulp.src([paths.css] , "!" + paths.concatCssDest)
      .pipe(concat(paths.concatCssDest))
      .pipe(gulp.dest("."));
});
```

To test gulp task i.e. **bundleJS**, we have created three JavaScript files under "**wwwroot\js**" folder and each file contains single function.



Test1.js

```
function test1(name) {
   console.log("Your Name Is: " + name);
}
```

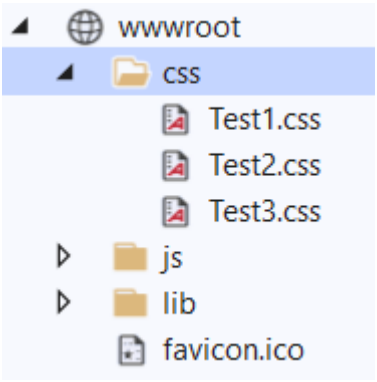
Test2.js

```
function test2(num1, num2) {
   var sum = num1 + num2;
   console.log("Sum of " + num1 + " + " + num2 + " = " + sum);
}
```

Test3.js

```
function test3(message) {
   console.log("Welcome to " + message);
}
```

To test gulp task i.e. **bundleCSS**, we have created three CSS files under "**wwwroot\css**" folder and each file contains single stylesheet.



Test1.css:

```
p {
    color: blue;
    font-size:15px;
}
```

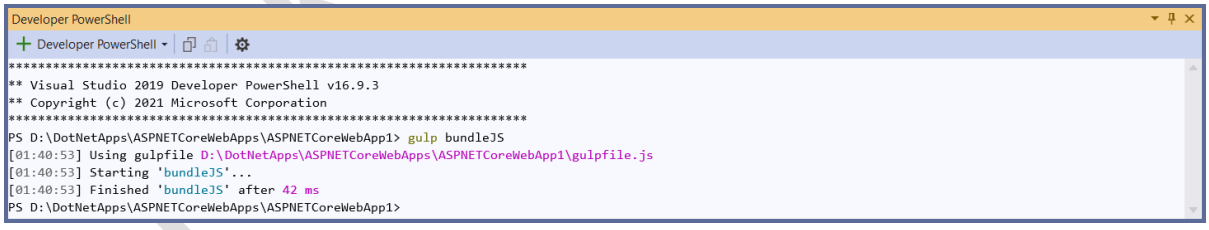
Test2.css:

```
div {
    border: 2px solid red;
    font-size: 15px;
    padding:5px;
}
```

Test3.css:

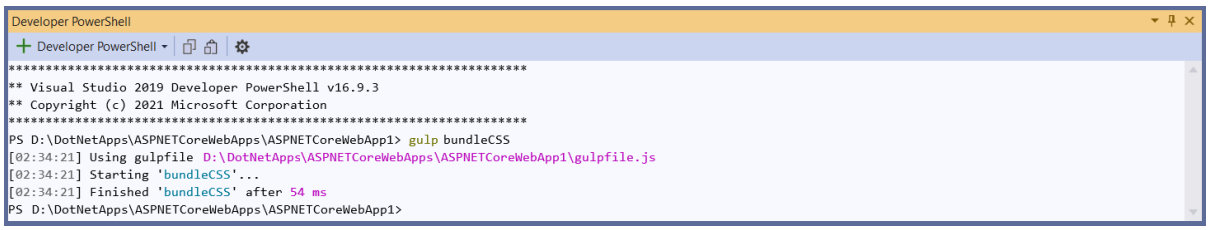
```
h1{
    color:brown;
    font-family:Arial
}
```

Now, for bundling the JavaScript, we need to run "gulp taskname" command from command prompt or terminal. Here, our task name is "bundleJS", so command becomes "gulp bundleJS".



As a result of the task, gulp is concatenating all three JavaScript files into **site.min.js** file.

Now, for bundling the CSS, we need to run "gulp taskname" command from command prompt or terminal. Here, our task name is "bundleCSS", so command becomes "gulp bundleCSS".



As a result of the task, gulp is concatenating all three CSS files into **site.min.css** file.

Minification

Minification is technique to reduce the size of the file by removing white spaces, commented code, line breaks and shortening variable names to one character. It helps to reduce the size of requested assets such as CSS, JavaScript which increases the load time from server to client.

To minify our JavaScript files, we can use the "**gulp-uglify**" plugin and for CSS, we can use "**gulp-cssmin**" plugin. So, first of all, these packages must be mentioned in package.json file.

Package.json

```
{
  "name": "aspnetcorewebapp1",
  "version": "1.0.0",
  "devDependencies": {
    "gulp": "^4.0.2",
    "gulp-concat": "2.6.1",
    "gulp-cssmin": "0.2.0",
    "gulp-uglify": "3.0.2"
  }
}
```

The next step is to import "**gulp-uglify**" and "**gulp-cssmin**" module to **gulpfile.js**.

```
var gulp = require("gulp"),
  concat = require("gulp-concat");
  cssmin = require("gulp-cssmin"),
  uglify = require("gulp-uglify");
```

Now, we have to specify the files that we want to minify. In this example, we have created "paths" array to specify all path that need to minify.

```
var paths = {
  webroot: "./wwwroot/"
};

paths.minJs = paths.webroot + "js/**/*.min.js";
paths.minCss = paths.webroot + "css/**/*.min.css";
```

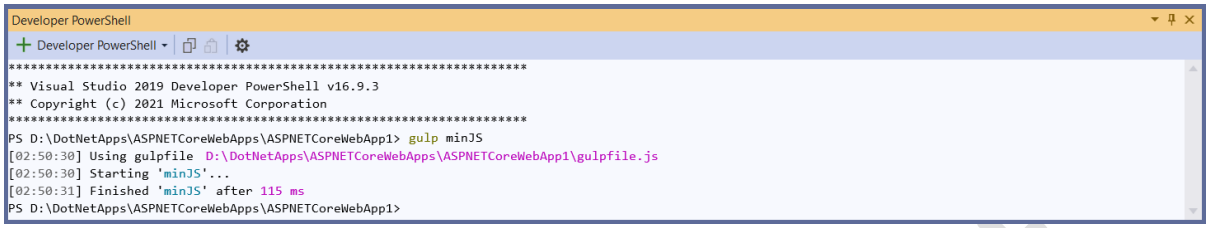
Now, we need to add task that perform minification. The function "**uglify()**" is used to minify the JavaScript file and "**cssmin()**" function is used to minify the CSS files.

```
gulp.task("minJS", function () {
  return gulp.src([paths.js, "!" + paths.minJs])
    .pipe(concat(paths.concatJsDest))
    .pipe(uglify())
    .pipe(gulp.dest("."));
});

gulp.task("minCSS", function () {
  return gulp.src([paths.css, "!" + paths.minCss])
    .pipe(concat(paths.concatCssDest))
    .pipe(cssmin())
    .pipe(gulp.dest("."));
});
```

Now, to **minify** the **JavaScript** and **CSS**, we need to run "**gulp taskname**" command from command prompt or terminal.

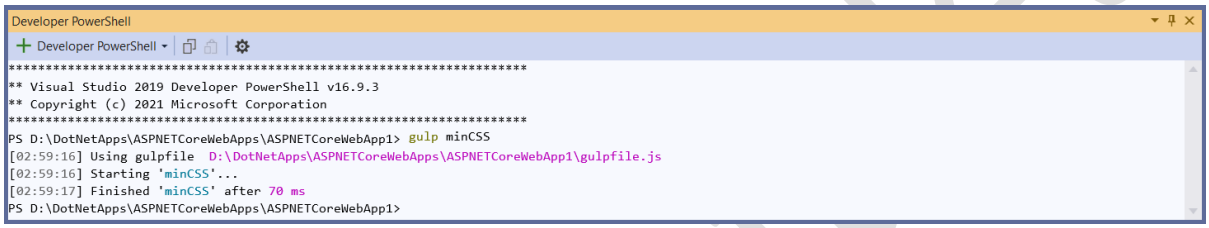
Here, our task name is "**minJS**" and "**minCSS**", so command becomes "**gulp minJS**" and "**gulp minCSS**".

gulp minJS



As a result of the run gulp minJS, gulp is minifying all three JS within same file i.e. **site.min.js** file.

gulp minCSS



As a result of the run gulp minCSS, gulp is minifying all three CSS within same file i.e. **site.min.css** file.

If you want to create a task for delete the created minified files then need to use the following module (plugin) to perform this task:

- del – Deletes files and directories

In package.json:

```
"devDependencies": {
        ……
        ……
        "del": "6.0.0"
}
```

In gulpfile.js:

```
var gulp = require("gulp"),
   concat = require("gulp-concat");
   cssmin = require("gulp-cssmin"),
   uglify = require("gulp-uglify");
   del = require("del");
……
……
gulp.task("cleanJS", function () {
   return del(paths.concatJsDest);
});
gulp.task("cleanCSS", function () {
   return del(paths.concatCssDest);
});
gulp.task("clean", gulp.series("cleanJS", "cleanCSS"));
```

Complete Example:

package.json:

```
{
  "name": "aspnetcorewebapp1",
  "version": "1.0.0",
  "devDependencies": {
    "gulp": "^4.0.2",
    "gulp-concat": "2.6.1",
    "gulp-cssmin": "0.2.0",
    "gulp-uglify": "3.0.2",
    "del": "6.0.0"
  }
}
```

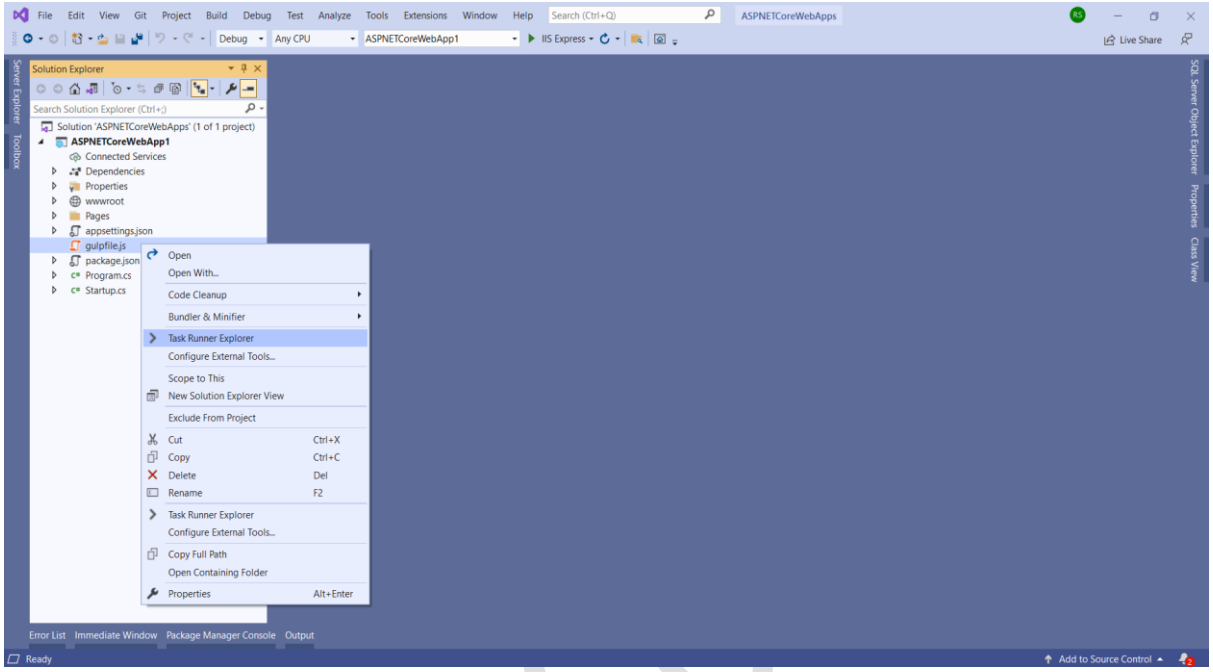
gulpfile.js

```
var gulp = require("gulp"),
    concat = require("gulp-concat");
    cssmin = require("gulp-cssmin"),
    uglify = require("gulp-uglify");
    del = require("del");

var paths = {
    webroot: "./wwwroot/"
};

paths.js = paths.webroot + "js/**/*.js";
paths.concatJsDest = paths.webroot + "js/site.min.js";
paths.css = paths.webroot + "css/**/*.css";
paths.concatCssDest = paths.webroot + "css/site.min.css";
paths.minJs = paths.webroot + "js/**/*.min.js";
paths.minCss = paths.webroot + "js/**/*.min.css";

gulp.task("bundleJS", function () {
    return gulp.src([paths.js], "!" + paths.concatJsDest)
        .pipe(concat(paths.concatJsDest))
        .pipe(gulp.dest("."));
});

gulp.task("bundleCSS", function () {
    return gulp.src([paths.css], "!" + paths.concatJsDest)
        .pipe(concat(paths.concatCssDest))
        .pipe(gulp.dest("."));
});

gulp.task("minJS", function () {
    return gulp.src([paths.js, "!" + paths.minJs])
        .pipe(concat(paths.concatJsDest))
        .pipe(uglify())
        .pipe(gulp.dest("."));
});

gulp.task("minCSS", function () {
    return gulp.src([paths.css, "!" + paths.minCss])
        .pipe(concat(paths.concatCssDest))
        .pipe(cssmin())
        .pipe(gulp.dest("."));
});

gulp.task("cleanJS", function () {
    return del(paths.concatJsDest);
});

gulp.task("cleanCSS", function () {
    return del(paths.concatCssDest);
});

gulp.task("clean", gulp.series("cleanJS", "cleanCSS"));

gulp.task("bundle", gulp.series("clean", "bundleJS", "bundleCSS"));

gulp.task("min", gulp.series("clean", "minJS", "minCSS"));
```

Execute the gulp task using Task Runner Explorer in Visual Studio:

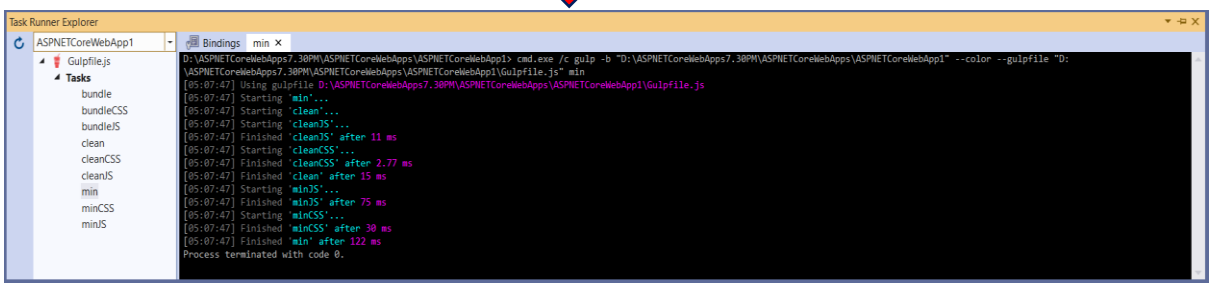
We are now ready to run our gulp tasks using Visual Studio.

Open the Task Runner Explorer window by right clicking on gulpfile.js and select “Task Runner Explorer” option from context menu.



You will notice that it is reading Gulpfile.js and showing all gulp tasks as shown below:

To run gulp task using Visual Studio Task Runner Explore, select task, right click on task and "Run".

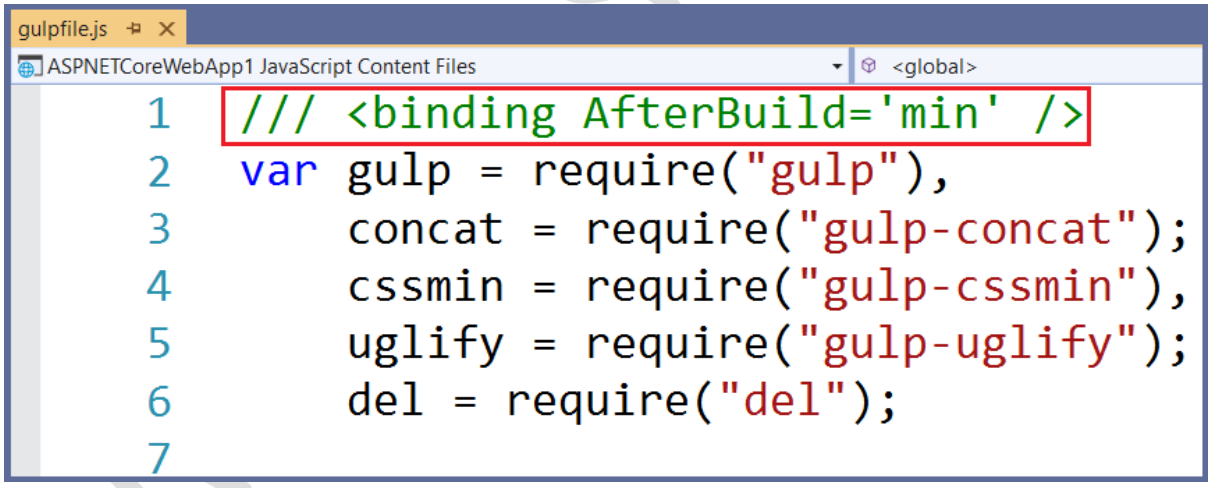


We can also bind these tasks to run during certain Visual Studio events. One task can be bound with multiple events and with any event. Following events can be bound with gulp tasks.

- Before Build
- After Build
- Clean
- Project Open

When we bind task with any event, it is reflected in gulpfile.js file. We can also do this directly by modifying the binding code in gulpfile.js file.

gulpfile.js file




```
1. /// <binding AfterBuild='min' />
```

The integration of gulp with Visual Studio makes it simple to manage JavaScript and CSS. We can create tasks for bundling and minification that can be integrated with any Visual Studio event.