**JavaScript, CSS, HTML & Other Static Files in ASP .NET Core 3.1**

By Shahed C on March 9, 2020

[4 Replies](https://wakeupandcode.com/javascript-css-html-other-static-files-in-asp-net-core-3-1/#comments)



This is the tenth of a new [series of posts](https://wakeupandcode.com/aspnetcore/#aspnetcore2020) on ASP .NET Core 3.1 for 2020. In this series, we’ll cover 26 topics over a span of 26 weeks from January through June 2020, titled **ASP .NET Core A-Z!** To differentiate from the [2019 series](https://wakeupandcode.com/aspnetcore/#aspnetcore2019), the 2020 series will mostly focus on a growing single codebase ([NetLearner!](https://wakeupandcode.com/netlearner-on-asp-net-core-3-1/)) instead of new unrelated code snippets week.

Previous post:

* [IIS Hosting for ASP .NET Core 3.1 Web Apps](https://wakeupandcode.com/iis-hosting-for-asp-net-core-3-1-web-apps/)

**NetLearner on GitHub**:

* Repository: <https://github.com/shahedc/NetLearnerApp>
* v0.10-alpha release: <https://github.com/shahedc/NetLearnerApp/releases/tag/v0.10-alpha>

**In this Article:**

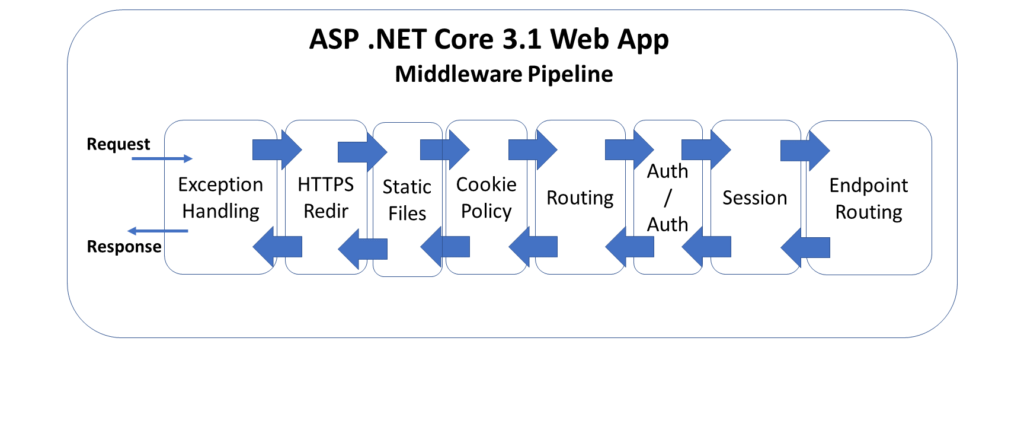
* J is for JavaScript, CSS, HTML & Other Static Files
* Configuring Static Files via Middleware
* Customizing Locations for Static Files
* Preserving CDN Integrity
* What About NPM or WebPack?
* References

**J is for JavaScript, CSS, HTML & Other Static Files**

**NOTE**: *this article will teach you how to include and customize the use of static files in ASP .NET Core web applications. It is not a tutorial on front-end web development.*

If your ASP .NET Core web app has a front end – whether it’s a collection of MVC Views or a Single-Page Application (SPA) – you will need to include static files in your application. This includes (but is not limited to): JavaScript, CSS, HTML and various image files.

When you create a new web app using one of the built-in templates (MVC, Razor Pages or Blazor), you should see a “wwwroot” folder in the Solution Explorer. This points to a physical folder in your file system that contains the same files seen from Visual Studio. However, this location can be configured, you can have multiple locations with static files, and you can enable/disable static files in your application if desired. In fact, you have to “opt in” to static files in your middleware pipeline.

ASP .NET Core 3.1 middleware pipeline diagram

You can browse the (template-generated) sample app (with static files) on GitHub at the following location:

* MVC – <https://github.com/shahedc/NetLearnerApp/tree/main/src/NetLearner.Mvc>
* Razor Pages – <https://github.com/shahedc/NetLearnerApp/tree/main/src/NetLearner.Pages>
* Blazor – <https://github.com/shahedc/NetLearnerApp/tree/main/src/NetLearner.Blazor>

**Configuring Static Files via Middleware**

Let’s start by observing the Startup.cs configuration file, from any of the web projects. We’ve seen this file several times throughout this blog series. In the **Configure**() method, you’ll find the familiar method call to enable the use of static files.

public void Configure(IApplicationBuilder app, IHostingEnvironment env)  
{  
 ...  
 app.UseStaticFiles();  
 ...  
}

This call to app.**UseStaticFiles**() ensures that static files can be served from the designated location, e.g. wwwroot. In fact, this line of code looks identical whether you look at the Startup.cs file in an [MVC](https://github.com/shahedc/NetLearnerApp/blob/main/src/NetLearner.Mvc/Startup.cs), [Razor Pages](https://github.com/shahedc/NetLearnerApp/blob/main/src/NetLearner.Pages/Startup.cs) or [Blazor](https://github.com/shahedc/NetLearnerApp/blob/main/src/NetLearner.Blazor/Startup.cs) project.

It’s useful to note the placement of this line of code. It appears *before* app.**UseRouting**() and app.**UseEndpoints**(), which is very important. This ensures that static file requests can be processed and sent back to the web browser without having to touch the application routing middleware. This becomes even more important when authentication is used.

Take a look at any of the web projects to observe the authentication added to them. In the code below, you can see the familiar call to app.**UseStaticFiles**() once again. However, there is also a call to app.**UseAuthentication**(). It’s important for the authentication call to appear *after* the call to use static files. This ensure that the authentication process isn’t triggered when it isn’t needed.

public void Configure(IApplicationBuilder app, IHostingEnvironment env)  
{  
 ...  
 app.UseStaticFiles();  
 ...  
 app.UseAuthentication();  
 ...  
 app.UseMvc(...);  
}

By using the middleware pipeline in this way, you can “short-circuit” the pipeline when a request has been fulfilled by a specific middleware layer. If a static file has been successfully served using the Static Files middleware, it prevents the next layers of middleware (i.e. authentication, routing, etc) from processing the request.

**NOTE**: if you need to secure any static files, e.g. private images, you can consider a cloud solution such as [Azure Blob Storage](https://wakeupandcode.com/azure-blob-storage-from-asp-net-core-file-upload/) to store the files. The files can then be served from within the application, instead of actual static files. You could also serve secure files (from outside the wwwroot location) as a [FileResult() object returned from an action method](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/static-files#static-file-authorization) that has an [Authorize] attribute.

**Customizing Locations for Static Files**

It may be convenient to have the default web templates create a location for your static files and also enable the use of those static files. As you’ve already seen, enabling static files isn’t magic. Removing the call to app.useStaticFiles() will disable static files from being served. In fact, the location for static files isn’t magic either.

In a [previous post](https://wakeupandcode.com/generic-host-builder-in-asp-net-core-3-1/), we had discussed how the Program.cs file includes a call to **CreateDefaultBuilder**() to set up your application:

public class Program  
{  
 ...  
 public static IHostBuilder CreateHostBuilder(string[] args) =>  
 Host.CreateDefaultBuilder(args)  
 .ConfigureWebHostDefaults(webBuilder =>   
 {  
 webBuilder.UseStartup<Startup>();  
 });  
}

Behind the scenes, this method call sets the “content root” to the current directory, which contains the “wwwroot” folder, your project’s “web root”. These can both be customized.

* To change the content root, you can [configure it while building the host](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/?view=aspnetcore-3.1&amp;tabs=windows#content-root), e.g.

Host.CreateDefaultBuilder(args).UseContentRoot("c:\\<content-root>")

* To change the web root within your content root, you can also configure it while building the host, e.g.

webBuilder.UseWebRoot("<content-root>")

You may also use the call to app.**UseStaticFiles**() to customize an alternate location to serve static files. This allows you to serve additional static files from a location outside of the designated web root.

...  
using Microsoft.Extensions.FileProviders;  
using System.IO;  
...  
public void Configure(IApplicationBuilder app)  
{  
 ...  
 app.UseStaticFiles(new StaticFileOptions  
 {  
 FileProvider = new PhysicalFileProvider(  
 Path.Combine(env.ContentRootPath, "AltStaticRoot")),  
 RequestPath = "/AltStaticFiles"  
 });  
}

***Wait a minute***… why does it look like there are *two* alternate locations for static files? There is a simple explanation:

* In the call to **Path**.**Combine**(), the “***AltStaticRoot***” is an actual folder in your current directory. This Path class and its Combine() method are available in the System.IO namespace.
* The “***AltStaticFiles***” value for **RequestPath**is used as a root-level “virtual folder” from which images can be served. The **PhysicalFileProvider** class is available in the Microsoft.Extensions.FileProviders namespace.

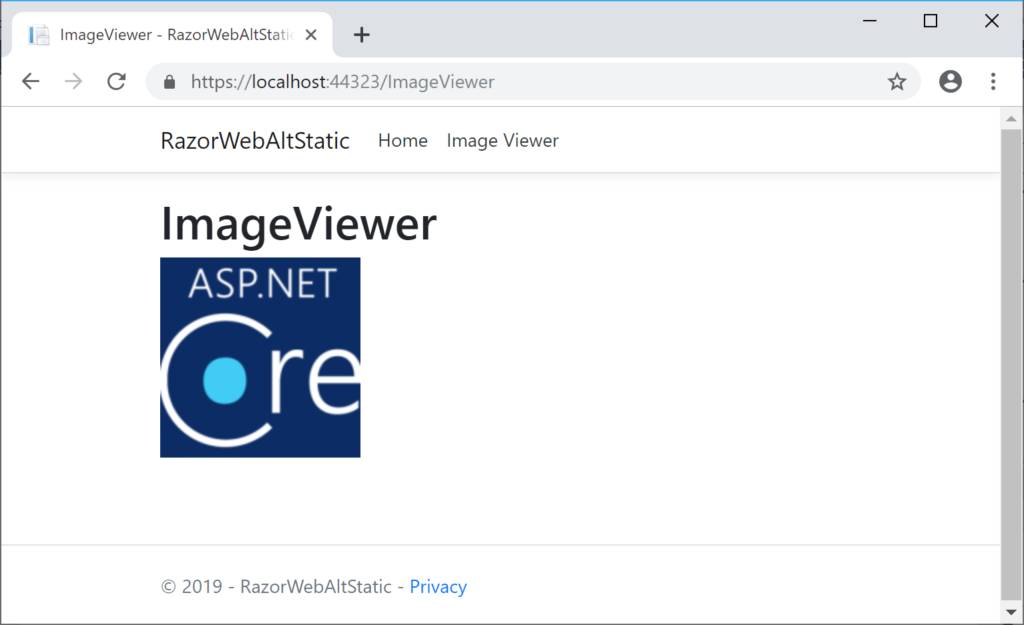
The following markup may be used in a .cshtml file to refer to an image, e.g. MyImage01.png:

<img src="~/AltStaticFiles/MyImages/MyImage01.png" />

The screenshot below shows an example of an image loaded from an alternate location.

Visual Studio showing image loaded from alternate location

The screenshot below shows a web browser displaying such an image.

Web Browser showing image loaded from alternate location

**Preserving CDN Integrity**

When you use a CDN (Content Delivery Network) to serve common CSS and JS files, you need to ensure that the integrity of the source code is reliable. You can rest assured that ASP .NET Core has already solved this problem for you in its built-in templates. Let’s take a look at the shared Identity pages, e.g. [\_ValidationScriptsPartial.cshtml](https://github.com/shahedc/NetLearnerApp/blob/main/src/NetLearner.Pages/Areas/Identity/Pages/_ValidationScriptsPartial.cshtml) in the Razor sample code.

<environment include="Development">  
 <script src="~/Identity/lib/jquery-validation/dist/jquery.validate.js"></script>   
</environment>  
  
<environment exclude="Development">  
 <script   
src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.17.0/jquery.validate.min.js"  
asp-fallback-src="~/Identity/lib/jquery-validation/dist/jquery.validate.min.js"  
asp-fallback-test="window.jQuery && window.jQuery.validator"  
crossorigin="anonymous"  
integrity="sha384-rZfj/ogBloos6wzLGpPkkOr/gpkBNLZ6b6yLy4o+ok+t/SAKlL5mvXLr0OXNi1Hp">  
 </script>   
</environment>

Right away, you’ll notice that there are two conditional <**environment**> blocks in the above markup. The first block is used only during development, in which the jQuery Validate file is obtained from your local copy. When *not* in development (e.g. staging, production, etc), the jQuery Validate file is obtained from a CDN.

You could use an automated hash-generation tool to generate the SRI ([Subresource Integrity](https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity)) hash values, but you would have to manually copy the value into your code. You can try out the newer LibMan (aka Library Manager) for easily adding and updating your client-side libraries.

**LibMan (aka Library Manager)**

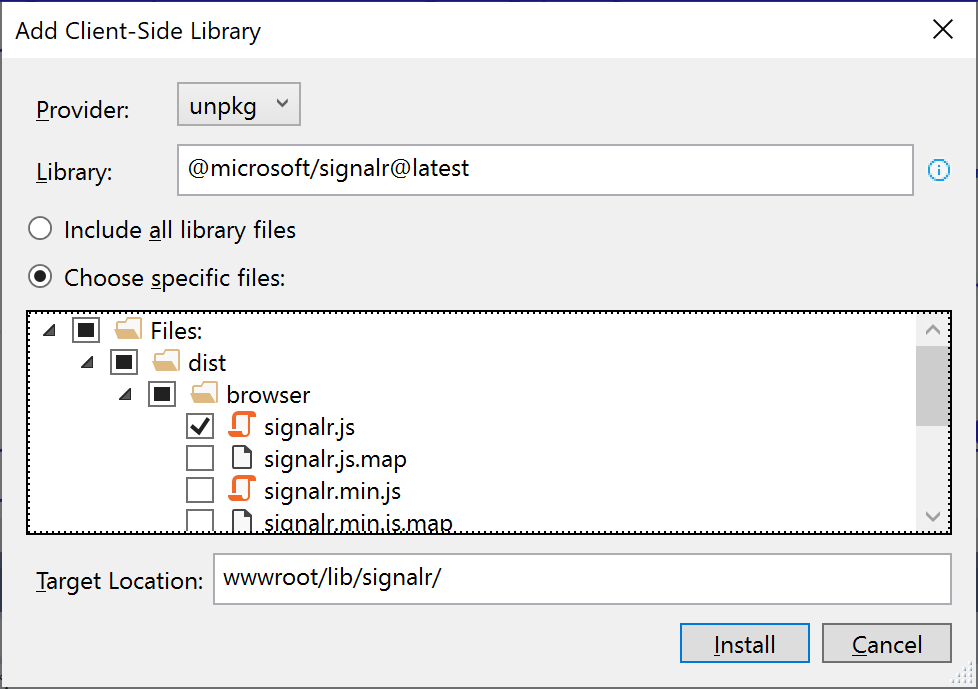
The easiest way to use LibMan is to use the built-in features available in Visual Studio. Using LibMan using the IDE is as easy as launching it from Solution Explorer. Specify the provider from the library you want, and any specific files you want from that library.

If you’ve already read my [SignalR article from my 2018 blog series](https://wakeupandcode.com/real-time-asp-net-core-web-apps-with-signalr/), you may recall the steps to add a client library via LibMan (aka Library Manager):

* Right-click project in Solution Explorer
* Select Add | Client-Side Library

In the popup that appears, select/enter the following:

* **Provider**: choose from cdnjs, filesystem, unpkg
* **Library** search term, e.g. @microsoft/signalr@latest to pick the latest stable version of the SignalR client-side library
* **Files**: At a minimum, choose specific files, e.g. signalr.js and/or its minified equivalent
* **Target Location**: modify the target location folder tree as desired

Adding a client-side library

For more on LibMan (using VS or CLI), check out the official docs:

* Use LibMan with ASP.NET Core in Visual Studio: <https://docs.microsoft.com/en-us/aspnet/core/client-side/libman/libman-vs>
* Use the LibMan command-line interface (CLI): <https://docs.microsoft.com/en-us/aspnet/core/client-side/libman/libman-cli>
* Library Manager: Client-side content manager for web apps: <https://devblogs.microsoft.com/aspnet/library-manager-client-side-content-manager-for-web-apps/>

In any case, using LibMan will auto-populate a “***libman.json”*** manifest file, which you can also inspect and edit manually. The aforementioned SignalR article also includes a [real-time polling web app sample](https://github.com/shahedc/SignalRCoreSamples/blob/master/SignalRPoll). You can view its [libman.json file](https://github.com/shahedc/SignalRCoreSamples/blob/master/SignalRPoll/libman.json) or the newer [NetLearner.Portal app’s libman.json file](https://github.com/shahedc/NetLearnerApp/blob/main/src/NetLearner.Portal/libman.json) to observe its syntax for using a SignalR client library.

{  
 "version": "1.0",  
 "defaultProvider": "unpkg",  
 "libraries": [  
 {  
 "library": "@microsoft/signalr@latest",  
 "destination": "wwwroot/lib/signalr/",  
 "files": [  
 "dist/browser/signalr.js"  
 ]  
 }  
 ]  
}

Every time you save the libman.json file, it will trigger LibMan’s restore process. This pulls down the necessary libraries from their specified source, and adds them to your local file system. If you want to trigger this restore process manually, you can always choose the “Restore Client-Side Libraries” option by right-clicking the libman.json file in Solution Explorer.

**What About NPM or WebPack?**

If you’ve gotten this far, you may be wondering: *“hey, what about NPM or WebPack?”*

It’s good to be aware that LibMan is a not a replacement for your existing package management systems. In fact, the Single-Page Application templates in Visual Studio (for Angular and React) currently use npm and WebPack. LibMan simply provides a lightweight mechanism to include client-side libraries from external locations.

For more information on WebPack in ASP .NET Core, I would recommend these 3rd-party articles:

* ReactJs Webpack and ASP.NET Core: <https://sensibledev.com/reactjs-webpack-and-asp-net-core/>
* Updating Your JavaScript Libraries in ASP.NET Core 2.2 *(includes LibMan, NPM, WebPack)*: <https://medium.com/@scottkuhl/updating-your-javascript-libraries-in-asp-net-core-2-2-3c2d985a491e>

**References**

* Static files in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/static-files>
* ASP.NET Core Middleware: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/>
* Library Manager (LibMan): <https://devblogs.microsoft.com/aspnet/library-manager-client-side-content-manager-for-web-apps/>
* Use LibMan with ASP.NET Core in Visual Studio: <https://docs.microsoft.com/en-us/aspnet/core/client-side/libman/libman-vs>
* Get started with ASP.NET Core SignalR: <https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr>
* Environment Tag Helper in ASP.NET Core: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/built-in/environment-tag-helper>
* Securing the CDN links in the ASP.NET Core 2.1 templates: <https://damienbod.com/2018/03/14/securing-the-cdn-links-in-the-asp-net-core-2-1-templates/>
* Pre-compressed static files with ASP.NET Core: <https://gunnarpeipman.com/aspnet/pre-compressed-files/>