



Andrew Lock

[Home](#) | [About](#) | [Subscribe](#) | [Dark](#)

Sponsored by **MailBee.NET Objects**—send, receive and process email in .NET apps. Supports Outlook file formats.



mindee
AI powered
document parsing
latest d
techno

November 01, 2016 in [ASP.NET CORE](#) [CACHING](#) [MIDDLEWARE](#) ~ 6 min read.

Adding Cache-Control headers to Static Files in ASP.NET Core

Share on: [f](#) [t](#) [r](#) [in](#)

Thanks to the ASP.NET Core middleware pipeline, it is relatively simple to add additional HTTP headers to your application by using [custom middleware](#). One common use case for this is to add caching headers.

Allowing clients and CDNs to cache your content can have [a massive effect on your application's performance](#). By allowing caching, your application never sees these additional requests and never has to allocate resources to process them, so it is more available for requests that cannot be cached.

In most cases you will find that a significant proportion of the requests to your site can be cached. A typical site serves both dynamically generated content (e.g. in ASP.NET Core, the HTML generated by your Razor templates) and static files (CSS stylesheets, JS, images etc). The static files are typically fixed at the time of publish, and so are perfect candidates for caching.

In this post I'll show how you can add headers to the files served by the `StaticFileMiddleware` to increase your site's performance. I'll also show how you can add a version tag to your file links, to ensure you don't inadvertently serve stale data.

Note that this is not the only way to add cache headers to your site. You can also use the `ResponseCacheAttribute` in MVC to decorate Controllers and Actions if you are returning data which is safe to cache.

You could also consider adding caching at the reverse proxy level (e.g. in IIS or Nginx), or use a third party provider like [CloudFlare](#).

Adding Caching to the StaticFileMiddleware

When you create a new ASP.NET Core project from the default template, you will find the `StaticFileMiddleware` is added early in the middleware pipeline, with a call to `AddStaticFiles()` in `Startup.Configure()`:

```
public void Configure(IApplicationBuilder app)
{
    // Logging and exception handler removed for clarity

    app.UseStaticFiles();

    app.UseMvc(routes =>
```



My new book *ASP.NET Core in Action, Second Edition* is available now! It supports .NET Core 5.0, and is available as a hardcover or paperback. You even get a free copy of the first edition of *ASP.NET Core in Action*.

ENJOY THIS BLOG?



```

1
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
    });
}

```

This enables serving files from the `wwwroot` folder in your application. The default template contains a number of static files (`site.css`, `bootstrap.css`, `banner1.svg`) which are all served by the middleware when running in development mode. It is these we wish to cache.

Don't we get caching by default?

Before we get to adding caching, let's investigate the default behaviour. The first time you load your application, your browser will fetch the default page, and will download all the linked assets. Assuming everything is configured correctly, these should all return a `200 - OK` response with the file data:

Name	Method	Status	Type	Size	Time	Timeline - Start Time
localhost	GET	200	document	7.4 KB	3 ms	
bootstrap.css	GET	200	stylesheet	143 KB	4 ms	
site.css	GET	200	stylesheet	954 B	3 ms	
jquery.js	GET	200	script	253 KB	3 ms	
bootstrap.js	GET	200	script	67.6 KB	2 ms	
site.js?v=Ynfdc1vuMNOWZiqT4...	GET	200	script	294 B	2 ms	
banner1.svg	GET	200	svg+xml	9.7 KB	2 ms	
banner2.svg	GET	200	svg+xml	8.4 KB	3 ms	
banner3.svg	GET	200	svg+xml	10.9 KB	3 ms	
banner4.svg	GET	200	svg+xml	12.3 KB	3 ms	
glyphicons-halflings-regular.woff2	GET	200	font	17.9 KB	3 ms	

15 requests | 531 KB transferred | Finish: 416 ms | DOMContentLoaded: 250 ms | Load: 262 ms

As well as the file data, by default the response header will contain `ETag` and `Last-Modified` values:

```

HTTP/1.1 200 OK
Date: Sat, 15 Oct 2016 14:15:52 GMT
Content-Type: image/svg+xml
Last-Modified: Sat, 15 Oct 2016 13:43:34 GMT
Accept-Ranges: bytes
ETag: "1d226ea1f827703"
Server: Kestrel

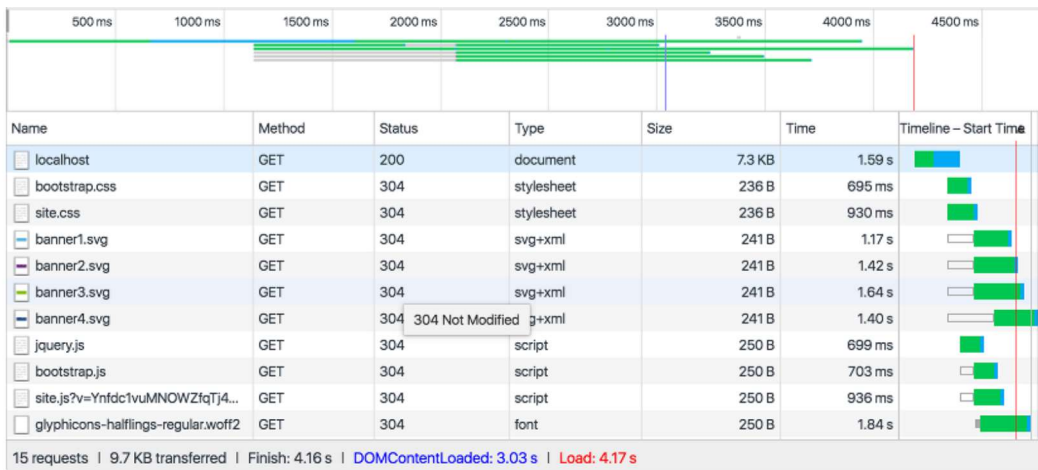
```

The second time a resource is requested from your site, your browser will send this `ETag` and `Last-Modified` value in the header as `If-None-Match` and `If-Modified-Since`. This tells the server that it doesn't need to send the data again if the file hasn't changed. If it hasn't changed, the server will send a `304 - Not Modified` response, and the browser will use the data it received previously instead.

This level of caching comes out-of-the-box with the `StaticFileMiddleware`, and gives improved performance by reducing the amount of bandwidth required. However it is important to note that the client is still sending a request to your server - the response has just been optimised. This becomes particularly noticeable with high latency connections or pages with many files - the browser still has to wait for the response to come back as `304`:

Name	Method	Status	Type	Size	Time	Timeline - Start Time
localhost	GET	200	document	7.4 KB	3 ms	
bootstrap.css	GET	200	stylesheet	143 KB	4 ms	
site.css	GET	200	stylesheet	954 B	3 ms	
jquery.js	GET	200	script	253 KB	3 ms	
bootstrap.js	GET	200	script	67.6 KB	2 ms	
site.js?v=Ynfdc1vuMNOWZiqT4...	GET	200	script	294 B	2 ms	
banner1.svg	GET	304	svg+xml	9.7 KB	2 ms	
banner2.svg	GET	304	svg+xml	8.4 KB	3 ms	
banner3.svg	GET	304	svg+xml	10.9 KB	3 ms	
banner4.svg	GET	304	svg+xml	12.3 KB	3 ms	
glyphicons-halflings-regular.woff2	GET	304	font	17.9 KB	3 ms	

15 requests | 531 KB transferred | Finish: 416 ms | DOMContentLoaded: 250 ms | Load: 262 ms



The image above uses Chrome's built in network throttling to emulate a GPRS connection with a very large latency of 500ms. You can see that the first *Index* page loads in 1.59s, after which the remaining static files are requested. Even though they all return 304 responses using only 250 bytes, the page doesn't actually finish loading until an additional 2.5s have passed!

Adding cache headers to static files

Rather than requiring the browser to always check if a file has changed, we now want it to assume that the file is the same, for a predetermined length of time. This is the purpose of the `Cache-Control` header.

In ASP.NET Core, you can easily add this header when you configure the `StaticFileMiddleware`:

```
using Microsoft.Net.Http.Headers;

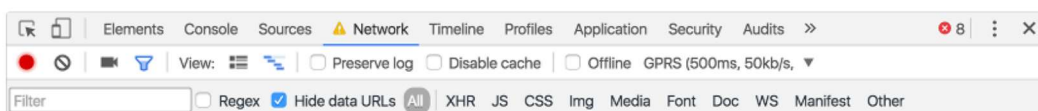
app.UseStaticFiles(new StaticFileOptions
{
    OnPrepareResponse = ctx =>
    {
        const int durationInSeconds = 60 * 60 * 24;
        ctx.Context.Response.Headers[HeaderNames.CacheControl] =
            "public,max-age=" + durationInSeconds;
    }
});
```

One of the overloads of `UseStaticFiles` takes a `StaticFileOptions` parameter, which contains the property `OnPrepareResponse`. This action can be used to specify any additional processing that should occur before a response is sent. It is passed a single parameter, a `StaticFileResponseContext`, which contains the current `HttpContext` and also an `FileInfo` property representing the current file.

If set, the `Action<StaticFileResponseContext>` is called before each successful response, whether a 200 or 304 response, but it won't be called if the file was not found (and instead returns a 404).

In the example provided above, we are setting the `Cache-Control` header (using the constant values defined in `Microsoft.Net.Http.Headers`) to cache our files for 24 hours. You can read up on the details of the various associated cache headers [here](#). In this case, we marked the response as `public` as we want intermediate caches between our server and the user to store the cached file too.

If we run our high-latency scenario again, we can see our results in action:



100000 ms	200000 ms	300000 ms	400000 ms	500000 ms	600000 ms	700000 ms	800000 ms	900000 ms	1000000 ms	1100000 ms	1200000 ms	1300000 ms
Name	Method	Status	Type	Size	Time	Timeline – Start Time						
localhost	GET	200	document	7.3 KB	1.54 s							
bootstrap.css	GET	200	stylesheet	(from cache)	3 ms							
site.css	GET	200	stylesheet	(from cache)	2 ms							
banner1.svg	GET	200	svg+xml	(from cache)	2 ms							
banner2.svg	GET	200	svg+xml	(from cache)	1 ms							
banner3.svg	GET	200	svg+xml	(from cache)	2 ms							
banner4.svg	GET	200	svg+xml	(from cache)	1 ms							
glyphicons-halflings-regular.woff2	GET	200	font	(from cache)	0 ms							
jquery.js	GET	200	script	(from cache)	4 ms							
bootstrap.js	GET	200	script	(from cache)	3 ms							
site.js?v=Ynfdc1vuMNOWZfqTj4...	GET	200	script	(from cache)	3 ms							

15 requests | 7.3 KB transferred | Finish: 1.75 s | DOMContentLoaded: 1.61 s | Load: 1.61 s

Our *index* page still takes 1.58s to load, but as you can see, all our static files are loaded from the cache, which means no requests to our server, and consequently no latency! We're all done in 1.61s instead of the 4.17s we had previously.

Once the `max-age` duration we specified has expired, or after the browser evicts the files from its cache, we'll be back to making requests to the server, but until then we can see a massive improvement. What's more, if we use a CDN or there are intermediate cache servers between the user's browser and our server, then they will also be able to serve the cached content, rather than the request having to make it all the way to your server.

Note: Chrome is a bit funny with respect to cache behaviour - if you reload a page using *F5* or the *Reload* button, it will generally not use cached assets. Instead it will pull them down fresh from the server. If you are struggling to see the fruits of your labour, navigate to a different page by clicking a link - you should see the correct caching behaviour then.

Cache busting for file changes

Before we added caching we saw that we return an `ETag` whenever we serve a static file. This is calculated based on the properties of the file such that if the file changes, the `ETag` will change. For those interested, this is the [snippet of code](#) that is used in ASP.NET Core:

```
_length = _fileInfo.Length;

DateTimeOffset last = _fileInfo.LastModified;
// Truncate to the second.
_lastModified = new DateTimeOffset(last.Year, last.Month, last.Day, last.Hour, last.Minute, last
    Second, last.Offset);

long etagHash = _lastModified.ToFileTime() ^ _length;
_etag = new EntityTagHeaderValue("'" + Convert.ToString(etagHash, 16) + "'");
```

This works great before we add caching - if the `ETag` hasn't changed we return a `304`, otherwise we return a `200` response with the new data.

Unfortunately, once we add caching, we are no longer making a request to the server. The file could have completely changed or have been deleted entirely, but if the browser doesn't ask, the server can't tell them!

One common solution around this is to append a querystring to the url when you reference the static file in your markup. As the browser determines uniqueness of requests including the querystring, it treats `https://localhost/css/site.css?v=1` as a different file to `https://localhost/css/site.css?v=2`. You can use this approach by updating any references to the file in your markup whenever you change the file.

While this works, it requires you to find every reference to your static file anywhere on your site whenever you change the file, so it can be a burden to manage. A simpler technique is to have the querystring be calculated based on the content of the file itself, much like an `ETag`. That way, when the file changes, the querystring will automatically change.

This is not a new technique - Mads Kristensen describes one method of achieving it with ASP.NET 4.X [here](#) but with ASP.NET Core we can use the link, script and image [Tag Helpers](#) to do the work for us.

It is highly likely that you are actually already using these tag helpers, as they are used in the default templates for exactly this purpose! For example, in `_Layout.cshtml`, you will find the following link:

```
<script src="~/js/site.js" asp-append-version="true"></script>
```

The Tag Helper is added with the markup `asp-append-version="true"` and ensures that when rendered, the link will be rendered with a hash of the file as a querystring:

```
<script src="/js/site.js?v=Ynfdc1vuMNOWZfqTj4N3SPcebaz0GXiIPgtfE-b2T04"></script>
```

If the file changes, the SHA256 hash will also change, and the cache will be automatically bypassed! You can add this Tag Helper to `img`, `script` and `link` elements, though there is obviously a degree of overhead as a hash of the file has to be calculated on first request. For files which are very unlikely to ever change (e.g. some images) it may not be worth the overhead to add the helper, but for others it will no doubt prevent quirky behaviour once you add caching!

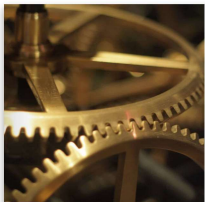
Summary

In this post we saw the built in caching using `ETag`s provided out of the box with the `StaticFileMiddleware`. I then showed how to add caching to the requests to prevent unnecessary requests to the server. Finally, I showed how to break out of the cache when the file changes, by using Tag Helpers to add a version querystring to the file request.

FOLLOW ME



ENJOY THIS BLOG?



PREVIOUS

Accessing services when configuring MvcOptions in ASP.NET Core

NEXT

Using dependency injection in a .Net Core console application

19 Comments Andrew Lock | .Net Escapades

Disqus' Privacy Policy

Login

Recommend 5 Tweet Share

Sort by Best



Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

**Jonathan Channon** • 3 years ago • edited

Have you tested Firefox.

Hit a static file, get 200

Reload the page you get a 304 from the server

Go to address bar and press enter you get 200 (cached). No server hit. Not sure how FF is deciding how to cache files as by default there is no Cache-Control, Age, Expires headers returned from Static middleware

13 ^ | v • Reply • Share ›

**Flavio Francisco** • 4 years ago

Awesome article! Thank you very much!

1 ^ | v • Reply • Share ›

**Muhammad Rehan Saeed** • 4 years ago

I saw your attempt at a PR. It's a shame it was not accepted. Performance should be a default, not an extra.

1 ^ | v • Reply • Share ›

**Sock** Mod → Muhammad Rehan Saeed • 4 years ago

Yeah, it's a shame, but I can see their concern in that it was maybe a bit too much complexity to run on every single request, even if you're not using the caching.

The main benefit I saw was the win in terms of setting up the cache headers properly - I'm sure a lot of people don't know to set the appropriate 'pragma' header etc in the required cases..

1 ^ | v • Reply • Share ›

**Muhammad Rehan Saeed** → Sock • 4 years agoI included a helper based on your PR in the [Boilerplate.AspNetCore] (<https://www.nuget.org/packages/Boilerplate.AspNetCore/>) NuGet package. The GitHub repo is [ASP-NET-MVC-Boilerplate/Framework] (<https://github.com/ASP-NET-MVC-Boilerplate/Framework>)

The Boilerplate project template will also use it shortly. Thanks Andrew!

1 ^ | v • Reply • Share ›

**Sock** Mod → Muhammad Rehan Saeed • 4 years ago

Excellent, more great stuff in the Boilerplate project:) Thanks!

^ | v • Reply • Share ›

**Wang Andrew** • a year ago

The point the article is, Chrome already has "cache" and cheat chrome the static files never change if the same day thus omit the request for server, right ? I have tried the "cache-control" as the article mentioned, but it still requests from server (although it returns 304)...

^ | v • Reply • Share ›

**Wang Andrew** → Wang Andrew • a year ago

Finally, the problem solved. Use asp-append-version="true", the static files must be in the directory "/wwwroot/images/". Thanks.


^ | v • Reply • Share ›

**Maq Said** • 3 years ago • editedHave you tried quickest browser cache bursting solution in .net core mvc 1.1/2.1. The tag helpers does all for images, style sheet and .js files. See this link <https://www.davepaquette.co...>


^ | v • Reply • Share ›

**Nirman** • 3 years ago


Very much informative and covered all aspects with utmost clarity.. Thanks for sharing




^ | v • Reply • Share ›




Ref C • 3 years ago
This was great, thanks!
^ | v • Reply • Share ›




Ben Niu • 3 years ago
I test this on Azure web app.
Added
`<script src=~\js/site.js" asp-append-version="true"></script>`
to client-website.
But seems on Chrome it still not going to refresh the cache.
^ | v • Reply • Share ›




Ross Huberty • 4 years ago
Highly educational piece, thanks!
^ | v • Reply • Share ›




arun prasad • 4 years ago • edited
Nice one
^ | v • Reply • Share ›




Manoj Kulkarni • 4 years ago
Nice article. Thank you very much
^ | v • Reply • Share ›




David Pine • 4 years ago
@Sock typo "looging"
^ | v • Reply • Share ›



Sock Mod ➔ **David Pine** • 4 years ago
Thanks, fixed
^ | v • Reply • Share ›



njy • 4 years ago
Good article! One thing though: when you say "Chrome is a bit funny with respect to cache behaviour" and suggest just changing the page I found out that if, instead of pressing F5 or hitting reload, you just go in the address bar (the entire url will auto-select) and just press enter, the result is the same.
This will avoid you from having to change the page twice to test the changes in the same page you are.
^ | v • Reply • Share ›



Sock Mod ➔ **njy** • 4 years ago
Yep, a good point, and something I've got into the habit of doing all the time now!
1 ^ | v • Reply • Share ›

☒ Subscribe

☐ Add Disqus to your site

☐ Add Disqus Add

☐ Do Not Sell My Data