A guide to caching in ASP.NET Core

This post looks at the various techniques available in ASP.NET Core for caching. We'll look at caching of data, partial pages and full pages at the server and client level and explain when to use each.

Why cache?

Caching adds some complexity to an application and as developers we must always be cautious when making our applications more complex so why bother with caching at all? A few reasons spring to mind:

- It saves you money (bandwidth costs, fewer servers required)
- It provides a better experience for your customers
- Faster sites make more money and rank better in Google

Just to emphasise how much of a speed benefit you can achieve through caching, if your page requires database calls to render, then caching the page can sometimes be an order of magnitude faster. Even completely static MVC views, without any external calls are faster when cached.

Given all these benefits and the fact that it can be very easy to add caching to your sites, it makes it a very attractive proposition. You do have to be careful though and later in the article, we will discuss some of the problems that caching can introduce.

What to cache?

Choosing what to cache is highly dependent on application. Generally speaking, to maximize performance, we want to cache at the highest level we can get away with.

For assets such as CSS, JS and images, we should be aggressively caching at the browser level and a cache duration of a year or more is fairly standard.

For relatively static, non-personalised pages, we can cache the entire page at both client and server level.

times.

At an even lower level, caching data to reduce calls to databases can be useful - particularly if the cached data is the result of multiple queries.

Typically, most applications will use a combinations of the above techniques for different areas.

Where to cache?

It makes sense to instruct the web browser to cache pages that rarely change but it is important to be aware of the limitations of caching at the client.

Unlike server side caching, it is not possible to instruct the browser to invalidate cache entries. This means that we need to be very careful with cache durations. The current best practice advice is to use very high cache durations for static assets and simply change the filename if an update is necessary. Obviously this is not an option for web pages. We can hardly change the URL of published pages every time an update is required. Instead, for web pages, we need to set a much lower cache duration at the client and rely on the server to efficiently handle requests.

Using a CDN is beyond the scope of this article, but is always worth considering, especially given how easy it is to set up today.

We also need to think about how server-side caching works once we move beyond a single server instance. When you have a single server, in-memory caching is easy to set up and generally works perfectly well. Most commercial sites however run on multiple load-balanced servers and dealing with caching becomes significantly more complex.

You have two high-level choices to consider. Do you:

- Maintain a discrete cache on every server
- Use a centralised cache that each server accesses

The first option, although slightly faster, has too many negatives to be recommended as a general solution:

- Discrepancies between caches can cause major headaches
- It is difficult to invalidate cache entries
- Wasted RAM filled with duplicated data

I strongly recommend you use a distributed cache solution, for all but the simplest of cases (i.e. caching static lookup data).

Caching data with IMemoryCache and IDistributedCache

The lowest level of caching in ASP.NET Core that we are going to discuss is the caching of data using IMemoryCache and IDistributedCache. These interfaces are the standard, in-built mechanisms for caching data in .NET Core. All other techniques that we discuss later in the article rely on IMemoryCache or IDistributedCache internally.

IMemoryCache

IMemoryCache is very similar to the System.Runtime.Caching.MemoryCache cache from .NET 4.

The interface itself is rather minimal:

```
public interface IMemoryCache : IDisposable
{
    bool TryGetValue(object key, out object value);
    ICacheEntry CreateEntry(object key);
    void Remove(object key);
}
```

This only tells half the story though because there are multiple extension methods available which make the API much richer and easier to use:

```
public static class CacheExtensions
{
    public static TItem Get<TItem>(this IMemoryCache cache, object key);

    public static TItem Set<TItem>(this IMemoryCache cache, object key, TItem value,
MemoryCacheEntryOptions options);

    public static bool TryGetValue<TItem>(this IMemoryCache cache, object key, out
TItem value);
    ...
}
```

You can register IMemoryCache in ConfigureServices using:

```
services.AddMemoryCache();
```

In either case, if you specify IMemoryCache in a component's constructor then it will get resolved along with the rest of the dependency graph.

The following code shows a naive example of using IMemoryCache to avoid hitting the database. The example returns a cached version of the data if available, else it queries the database, caches the data and returns the result:

```
public class BlahService
{
    private const string BlahCacheKey = "blah-cache-key";

    private readonly IMemoryCache _cache;
    private readonly IDatabase _db;

    public BlahService(IMemoryCache cache, IDatabase db)
    {
        _cache = cache;
        _db = db;
    }

    public async Task<IEnumerable<Blah>> GetBlahs()
    {
        if (_cache.TryGet(BlahCacheKey, out IEnumerable<Blah> blahs))
        {
            return blahs;
        }

        blahs = await _db.getAll<Blah>(...);

        _cache.Set(BlahCacheKey, blahs, ...);

        return blahs;
    }
}
```

When saving to IMemoryCache, MemoryCacheEntryOptions provides you with many ways to expire cache content. Options include absolute expiry (a fixed time), sliding expiry (time since last accessed) and expiry based on a token which is a powerful technique for creating dependencies between cache items. There are also overloads of Set which allow you to choose an expiration time directly. Here are a few examples:

```
//absolute expiration using TimeSpan
_cache.Set("key", item, TimeSpan.FromDays(1));

//absolute expiration using DateTime
```

```
//sliding expiration (evict if not accessed for 7 days)
_cache.Set("key", item, new MemoryCacheEntryOptions
{
    SlidingExpiration = TimeSpan.FromDays(7)
});

//use both absolute and sliding expiration
_cache.Set("key", item, new MemoryCacheEntryOptions
{
    AbsoluteExpirationRelativeToNow = TimeSpan.FromDays(30),
    SlidingExpiration = TimeSpan.FromDays(7)
});

// use a cancellation token
var tokenSource = new CancellationTokenSource();

var token = new CancellationChangeToken(tokenSource.Token);

_cache.Set("key", item, new MemoryCacheEntryOptions().AddExpirationToken(token));
```

When using cancellation tokens, you can store the CancellationTokenSource itself in the cache and access it when you need to. To evict all cache entries using tokens from a particular CancellationTokenSource, you can just call the Cancel method:

```
tokenSource.Cancel();
```

The documentation provides full details for all the options you can use with IMemoryCache.

IDistributedCache

For web farm scenarios, you will want to make use of IDistributedCache instead of IMemoryCache:

```
public interface IDistributedCache
{
    byte[] Get(string key);
    Task<byte[]> GetAsync(string key);

    void Set(string key, byte[] value, DistributedCacheEntryOptions options);
    Task SetAsync(string key, byte[] value, DistributedCacheEntryOptions options);

    void Refresh(string key);
    Task RefreshAsync(string key);

    void Remove(string key);
    Task RemoveAsync(string key);
}
```

- Additional async methods
- Refresh methods (which just reset sliding expirations without retrieving data as far as I can tell)
- Byte based rather than object based (though extension methods add the ability to use string values)

This last change means that you will need to serialize any objects being stored yourself. The example below uses Json.NET for this:

```
public async Task SaveToCache<T>(string key, T item, int expirationInHours)
{
    var json = JsonConvert.SerializeObject(item);

    await _cache.SetStringAsync(key, json, new DistributedCacheEntryOptions
    {
        AbsoluteExpirationRelativeToNow = TimeSpan.FromHours(expirationInHours)
    });
}

public async Task<T> RetrieveFromCache<T>(string key)
{
    var json = await _cache.GetStringAsync(key);

    return JsonConvert.DeserializeObject<T>(json);
}
```

DistributedCacheEntryOptions offers absolute and sliding expiration much like MemoryCacheEntryOptions but token based expiration is absent. This makes adding cache dependencies much more of a challenge and you will need to roll your own implementation if you need this functionality.

There are three Microsoft implementations of the IDistributedCache interface currently available. These include a local, in-memory version ideal for development, plus Redis and SQL Server versions. Given that SQL Server is disk based rather than in-memory, I can't imagine many people opting for this version.

The local in-memory version of IDistributedCache is part of Microsoft.Extensions.Caching.Memory so is already brought in by the MVC package. If you should need to manually add it, you can use:

```
services.AddDistributedMemoryCache();
```

referencing the NuGet, simply add the following to ConfigureServices:

```
services.AddDistributedRedisCache (option =>
{
    option.Configuration = "your connection string";
    option.InstanceName = "your instance name";
});
```

Caching partial pages with Tag Helpers

There is no doubt that caching of data can significantly speed up server responses but we can often go one step further and cache rendered page output rather than (or in addition to) raw data. Partial page caching is available using the built-in Caching Tag Helpers.

The cache tag helper

At it's simplest, you can wrap part of a view in cache tags to enable caching:

```
<cache>...</cache>
```

This results in that part of the page being cached for a default duration of 20 minutes.

Obviously caching an arbitrary part of the page is next to useless unless that part of the page is expensive to render. One obvious candidate for caching is a view component call. If you are not familiar with view components then you can think of them as more capable successors to child actions. They are very useful for secondary parts of the page such as sidebars. If your sidebar is dynamically generated from a database then caching the result can be extremely beneficial.

```
<cache expires-on="@TimeSpan.FromSeconds(600)">
    @await Component.InvokeAsync("BlogPosts", new { tag = "popular" })
</cache>
```

The above example caches the blog posts view component for 10 minutes.

The cache tag helper allows you to vary the cache (i.e. create multiple separate copies) by many different criteria including headers, queries, routes, cookies and even users.

```
<cache expires-sliding="@TimeSpan.FromHours(1)" vary-by-user="true">
    ...user specific content...
</cache>
```

certainly very feature rich. See the docs for full details.

The distributed-cache tag helper

As with IMemoryCache, the Cache tag helper has a sibling for use in web farm situations where a distributed solution is required.

```
<distributed-cache name="key">
    @await Component.InvokeAsync("BlogPosts", new { tag = "popular" })
</Cache>
```

In use, the distributed cache tag helper is very similar to the in memory version. Other than the tag name change from cache to distributed-cache, the only notable difference is the requirement of a name attribute for the distributed version. This value is used to generate a key for the cache entry.

Internally, the tag helper uses the IDistributedCache outlined in the previous section. If you do not configure an IDistributedCache implementation in ConfigureServices then the in-memory version is used without any configuration necessary.

Caching full pages with response caching

The highest level of caching that we can make use of is caching of the entire page. Caching full pages in the browser results in the very minimum of server load and caching fully rendered pages on the server can also hugely reduce load and response times.

In .NET core, these two techniques are closely related. We have a ResponseCache attribute which is used to set cache headers and we have a ResponseCaching piece of middleware which can optionally be used to enable server side caching.

Caching in the browser

You can set cache headers manually using Response.Headers but the preferred approach for caching actions is to make use of the ResponseCache attribute which can be applied to both actions and controllers.

```
[ResponseCache(Duration = 3600)]
```

seconds or one hour.

Instead of specifying values for each instance of the attribute, we can also configure one or more cache profiles. Cache profiles are configured in ConfiguresServices when you add the MVC middleware:

```
services.AddMvc(options =>
{
    options.CacheProfiles.Add("Hourly", new CacheProfile()
    {
        Duration = 60 * 60 // 1 hour
    });
    options.CacheProfiles.Add("Weekly", new CacheProfile()
    {
        Duration = 60 * 60 * 24 * 7 // 7 days
    });
});
```

You can then reference the cache profile names in the ResponseCache attributes

```
[ResponseCache(CacheProfileName = "Weekly")]
```

You can also explicitly disable any caching with the following:

```
[ResponseCache(Location = ResponseCacheLocation.None, NoStore = true)]
```

Note that both Location and NoStore are required. NoStore returns the standard "cache-control: no-store" header but some older proxies do not understand this so Location = ResponseCacheLocation.None adds "no-cache" values to cache-control and pragma headers.

Caching at the server

As mentioned above, caching at the server uses a piece of middleware which reads the values set by the ResponseCache attribute and caches the page appropriately.

The response caching middleware is in a separate NuGet package so you will need to add a package reference to Microsoft.AspNetCore.ResponseCaching in order to use it. Once installed, it can be added to the pipeline by adding the following to ConfigureServices:

```
services.AddResponseCaching();
```

page. One common use for this would be to vary by Accept-encoding header so you can cache both gzipped and non-gzipped responses, plus any other compression algorithms you are using. This of course assumes that you are compressing within your application rather than at the reverse proxy level.

As well as the standard VaryByHeader option, you can use the ResponseCache attribute to specify a VaryByQuery value which is used exclusively by the server side response caching middleware. As you would expect, this causes the middleware to store additional response copies based on the query string values specified.

Limitations

Server side response caching can provide astonishing gains in speed but it is important to be aware of the limitations of such an approach.

If you are caching fairly static content for anonymous users and the pages have no personalisation or forms then full-page caching is ideal. Unfortunately, this is rarely true and you will run into issues if you try to cache pages in these other situations.

In fact the built-in response caching middleware will not cache the page if any of the following are true:

- The response code is not 200
- The request method is not GET or HEAD
- An Authorization header is present
- A Set-Cookie header is present

In addition, using the Anti-CSRF features of MVC will override any explicit cache-control header you have set and replace it with "no-cache, no-store" resulting in your page not being cached. This is essential because this feature works by setting a cookie and embedding a form value so you do not want to cache these values.

Cache invalidation

One common approach to full page caching is to cache on the client for a short period of time and on the server for a much longer period. This technique relies on the fact that we can typically remove cache entries early on the server if necessary. This way we can effectively cache frequently accessed pages indefinitely and only invalidate the cache and store a new version if the page changes.

cache duration is used for both client and server caches. Secondly, currently there is no easy way to invalidate cache entries. This is a real shame and I hope it is something that will change. For now I ended up writing my own basic implementation which we will look at next time.

Caching JS, CSS, Images etc.

Images and other static files are served by adding the static files middleware. A typical registration in the Configure method of startup.cs looks like this:

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

This code will enable the serving of static files but not in the most efficient way. By default, no cache headers are used so the browser will request these files again and again, slowing your sites and putting more load on your server.

The good news is that it is very easy to change the static files registration code to enable browser caching. Here we set caching to a year:

```
app.UseStaticFiles(new StaticFileOptions
{
    OnPrepareResponse = (context) =>
    {
        var headers = context.Context.Response.GetTypedHeaders();

        headers.CacheControl = new CacheControlHeaderValue
        {
            Public = true,
            MaxAge = TimeSpan.FromDays(365)
        };
    }
});
```

Summary

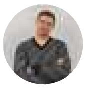
Caching can drastically reduce your costs and also provide a more responsive site for your customers. We looked at a number of different techniques from low level caching of data through to entire page caching at both the client and server. We discussed some gotchas to be aware of when adding caching to your applications and also explained about a few limitations when using the built-in response caching.

Next time, we'll take a look at writing our own response caching solution.

If you liked the article, I would really appreciate it if you could share it with your Twitter followers.

SHARE ON TWITTER

Comments



CALABONGA WROTE ON 19 JUL 2017

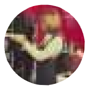
Thank you very much! This is great post!

VICENTE WROTE ON 28 AUG 2017

Nice article! I would appreciate if you could clarify the following questions:

How does the response caching middleware interacts with memory caching and distributed caching? Can it be used with both? Does it default to memory? Is it a mechanism independent to memory and distributed caching? Can it be adapted to store cached responses in a distributed cache?

Thanks



MALACHI WROTE ON 30 AUG 2017

Well written, easy to read, informative. Thank you for this

URIEL WROTE ON 25 SEP 2017

Can you add some information about session?
I read that session data lives in the cache, but its not clear in wich of all the caches.
Also, i try without any cache and session still works! So, im really confused!

Thanks!!

PAUL HILES WROTE ON 11 OCT 2017

@Vicente - unfortunately it seems as though the built-in ResponseCaching middleware is extremely limited at this time. As far as I can tell, there is currently no way to change the default in-memory caching mechanism. I am sure that this will be added in future versions but for now, writing your own implementation is fairly simple. See my next article for an explanation - https://www.devtrends.co.uk/blog/custom-response-caching-in-asp.net-core-with-cache-invalidation

@Uriel - I do not use session but it looks fairly simple to use a custom cache store in .NET core. If you do not register an IDistributedCache implementation then the default in-memory version will be used. To change this behaviour, you have two options. You can use one of the supported NuGet packages to add Redis or SQL Server support or alternatively, you can write your own implementation and register it in ConfigureServices. In my testing, the session code seems to use the last IDistributedCache implementation that has been registered in the case where you have multiple registered implementations.

For an introduction to session in .NET Core, see https://andrewlock.net/an-introduction-to-session-storage-in-asp-net-core/

URIEL WROTE ON 06 NOV 2017

Thanks for your reply!
Im courius, why you dont use session? if you dont use session how do you make things like a log in?

@Uriel - I prefer to keep my applications stateless whenever possible. This is primarily to allow easy scaling but also for simplicity, performance and user experience. There is no need for session for logging in. In MVC, the standard approach would be using a secure cookie. You can also use this to store small amounts of additional info if required, though you need to be aware that this is sent long with every request. You can also use HTML5 local storage etc for things like shopping baskets.

MICHAEL FREIDGEIM WROTE ON 10 DEC 2017

Link https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/built-in/cachetaghelper is broken

PAUL HILES WROTE ON 10 DEC 2017

@Michael - thanks. now updated to use the new URL

MARK DEMPSEY WROTE ON 13 DEC 2017

I am using IDistributedCache in an ASP.NET Core MVC controller - no Redis or Sql Server are currently configured for this so it should be defaulting to the in-memory version. When deploying to Dev/Test/Prod environments, if Redis or Sql isn't configured, will this work properly across servers in a web farm? I was looking for the most lightweight solution and was hoping to avoid Redis or Sql. Thanks in advance for your help.

PAUL HILES WROTE ON 13 DEC 2017

@Mark - it will cache fine but each server will maintain its own collection of items. This duplication is a bit of a waste of server memory but that may not be a concern. For strictly read-only data, this different servers might contain different versions of data, leading to inconsistencies and bugs that are very hard to diagnose. In addition, if you need to update a cache entry, there is no way to update all servers. Server affinity can go someway to mitigate some of these problems, but it is dangerous ground IMO.

ALEX WROTE ON 11 JAN 2018

How can I access IMemoryCache outside of the controller? Ex: from a middleware?

PAUL HILES WROTE ON 12 JAN 2018

@Alex - the DI system in .NET Core means that all you need to do is specify an IMemoryCache parameter in the constructor and it will automatically get resolved using the built-in container. You would typically then use the parameter to set a private member variable and access this from the rest of the class. With middleware, you also have the option of adding dependencies to the Invoke method directly instead of the constructor. This allows you to resolve components on a per request basis but this is probably not what you want when using IMemoryCache.

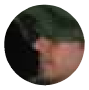
In either case, the MVC middleware will automatically register IMemoryCache with the container so there is no need for you to register it.

MICHAEL HAREN WROTE ON 29 JUN 2018

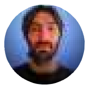
I suggest removing ".core" from the nuget package name for redis. It should be just "Microsoft.Extensions.Caching.Redis"

https://github.com/aspnet/Caching/issues/348#issuecomment-324222570

JESUS ESTEVEZ (@JECAESTEVEZ) WROTE ON 16 JUL 2018


JOAO PRADO WROTE ON 24 AUG 2018

Great article, thank you.


SEBASTIAN WROTE ON 12 NOV 2018

First of all: Great article. Thanks for sharing that.
I'm rather new on Asp.Net Core so I may be asking a newbie question.

I have implemented in my current Asp.Net Core, which is a REST API two cache techniques.
On the one hand, I have an IMemoryCache implemented for a specific controller in which I need to cache some data that rarely changes.
On the other hand, I'm using ResponseCache attribute in most of my controllers with several profiles.
The question is related to this specific controller in which I have IMemoryCache and ResponseCache attribute as well.
Is that possible or I'm doing something that I shouldn't?

And one last thing. I realize that when the same question (same resource in REST) is being asked but from different browsers, ResponseCache attribute is not usefully.
So, what would be the right thing to do to cache for the same information which I know up front it's going to be asked from several clients?
Thanks

PAUL HILES WROTE ON 12 NOV 2018

@Sebastian - there is nothing wrong with caching the whole response (via ResponseCache) and also caching fairly static data in a memory cache.

ResponseCache serves two quite different purposes. First, it sets cache headers so the browser

AddResponseCaching() middleware) to store the response on the server so the action does not need to execute again. The first method will stop an individual client (a browser) from making the request to the server unnecessarily but as you point out, it will not affect a different browser which will call the server itself before caching the data in its own cache). The second method will short-circuit the logic, database calls etc in the controller action and return the response immediately so although using a second browser will still result in an HTTP call (which cannot be avoided), the server will be doing much less work and will return much faster.

It is these two functions used together that can result in huge speed increases and a much smaller load on your servers.

RANGANATH WROTE ON 05 FEB 2019

Great article.

ATMANE WROTE ON 12 JUL 2019

Here's a more explicit article on the subject, thank you !

STEFAN WROTE ON 11 APR 2020

Fantastic post!!!!

WENDY WROTE ON 08 MAY 2020

With idistributedmemorycache, is there a way to clear the cache in demand, e.g through an endpoint if an api

Esclarecedor. Obrigado pelo artigo.
Thank you. Brazil

HEKURAN WROTE ON 26 AUG 2020

Great article - Thank you!

CHIENVX89 WROTE ON 19 DEC 2020

Thanks you, this guide is very clear.
Can you help me find solution for under question:
How to set cache (distribute cache) is never expire?

NAME

EMAIL

Used for Gravatar and never shared

COMMENT

About

DevTrends is owned by Paul Hiles who has over 15 years of experience developing Microsoft based applications. MORE »

Contact

Email me at **paul@devtrends.co.uk**

Search

enter search term

SEARCH

Latest Blog Posts

Fast Free Geolocation in .NET with freegeoip.net

3 Ways To Avoid An Anemic Domain Model In Entity Framework

Installing the ASP.NET Core 2.0 runtime store on Linux

Dependency Injection in action filters in ASP.NET Core

Custom response caching in ASP.NET Core (with cache invalidation)

A guide to caching in ASP.NET Core

Hashing, Encryption and Random in ASP.NET Core

Create a Free Private NuGet Server with Continuous Deployment using VSTS

Conditional Middleware based on request in ASP.NET Core

VIEW ALL BLOG POSTS »

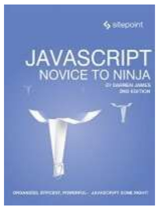
Subscribe

Sign up below and never miss a new article

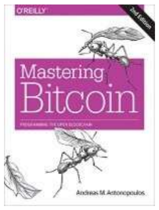
enter your email address

SUBSCRIBE

Reading List

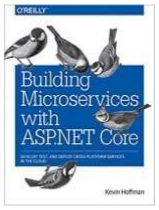


JavaScript: Novice to Ninja 2nd Edition (2017) - Darren Jones

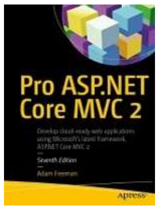


Mastering Bitcoin: Programming the Open Blockchain - Andreas M. Antonopoulos

Clean Architecture: A Craftsman's Guide to Software Structure and Design - Robert C. Martin



Building Microservices with ASP.NET Core - Kevin Hoffman



Pro ASP.NET Core MVC 2 - Adam Freeman