Cache in-memory in ASP.NET Core

04/24/2024

By Rick Anderson, John Luo, and Steve Smith

Caching can significantly improve the performance and scalability of an app by reducing the work required to generate content. Caching works best with data that changes infrequently and is expensive to generate. Caching makes a copy of data that can be returned much faster than from the source. Apps should be written and tested to never depend on cached data.

ASP.NET Core supports several different caches. The simplest cache is based on the IMemoryCache. `IMemoryCache` represents a cache stored in the memory of the web server. Apps running on a server farm (multiple servers) should ensure sessions are sticky when using the in-memory cache. Sticky sessions ensure that requests from a client all go to the same server. For example, Azure Web apps use Application Request Routing (ARR) to route all requests to the same server.

Non-sticky sessions in a web farm require a distributed cache to avoid cache consistency problems. For some apps, a distributed cache can support higher scale-out than an in-memory cache. Using a distributed cache offloads the cache memory to an external process.

The in-memory cache can store any object. The distributed cache interface is limited to `byte[]`. The in-memory and distributed cache store cache items as key-value pairs.

System.Runtime.Caching/MemoryCache

System.Runtime.Caching/MemoryCache (NuGet package) can be used with:

- .NET Standard 2.0 or later.
- Any .NET implementation that targets .NET Standard 2.0 or later. For example, ASP.NET Core 3.1 or later.
- .NET Framework 4.5 or later.

Microsoft.Extensions.Caching.Memory/`IMemoryCache` (described in this article) is recommended over `System.Runtime.Caching`/`MemoryCache` because it's better integrated into ASP.NET Core. For example, `IMemoryCache` works natively with ASP.NET Core dependency injection.

Use `System.Runtime.Caching`/`MemoryCache` as a compatibility bridge when porting code from ASP.NET 4.x to ASP.NET Core.

Cache guidelines

- Code should always have a fallback option to fetch data and not depend on a cached value being available.

- The cache uses a scarce resource, memory. Limit cache growth:
    - Do not insert external input into the cache. As an example, using arbitrary user-provided input as a cache key is not recommended since the input might consume an unpredictable amount of memory.
    - Use expirations to limit cache growth.
    - Use SetSize, Size, and SizeLimit to limit cache size. The ASP.NET Core runtime does not limit cache size based on memory pressure. It's up to the developer to limit cache size.

Use IMemoryCache

Warning

Using a *shared* memory cache from Dependency Injection and calling `SetSize`, `Size`, or `SizeLimit` to limit cache size can cause the app to fail. When a size limit is set on a cache, all entries must specify a size when being added. This can lead to issues since developers may not have full control on what uses the shared cache. When using `SetSize`, `Size`, or `SizeLimit` to limit cache, create a cache singleton for caching. For more information and an example, see Use SetSize, Size, and SizeLimit to limit cache size. A shared cache is one shared by other frameworks or libraries.

In-memory caching is a *service* that's referenced from an app using Dependency Injection. Request the `IMemoryCache` instance in the constructor:

C#

Copy

```
public class IndexModel : PageModel
{
    private readonly IMemoryCache _memoryCache;

    public IndexModel(IMemoryCache memoryCache) =>
        _memoryCache = memoryCache;

    // ...
```

The following code uses TryGetValue to check if a time is in the cache. If a time isn't cached, a new entry is created and added to the cache with Set:

C#

Copy

```
public void OnGet()
{
    CurrentDateTime = DateTime.Now;
```

```
    if (!_memoryCache.TryGetValue(CacheKeys.Entry, out DateTime
cacheValue))
    {
        cacheValue = CurrentDateTime;

        var cacheEntryOptions = new MemoryCacheEntryOptions()
            .SetSlidingExpiration(TimeSpan.FromSeconds(3));

        _memoryCache.Set(CacheKeys.Entry, cacheValue, cacheEntryOptions);
    }

    CacheCurrentDateTime = cacheValue;
}
```

In the preceding code, the cache entry is configured with a sliding expiration of three seconds. If the cache entry isn't accessed for more than three seconds, it gets evicted from the cache. Each time the cache entry is accessed, it remains in the cache for a further 3 seconds. The `CacheKeys` class is part of the download sample.

The current time and the cached time are displayed:

CSHTML

Copy

```
<ul>
    <li>Current Time: @Model.CurrentDateTime</li>
    <li>Cached Time: @Model.CacheCurrentDateTime</li>
</ul>
```

The following code uses the `Set` extension method to cache data for a relative time without `MemoryCacheEntryOptions`:

C#

Copy

```
_memoryCache.Set(CacheKeys.Entry, DateTime.Now, TimeSpan.FromDays(1));
```

In the preceding code, the cache entry is configured with a relative expiration of one day. The cache entry gets evicted from the cache after one day, even if it's accessed within this timeout period.

The following code uses GetOrCreate and GetOrCreateAsync to cache data.

C#

Copy

```
public void OnGetCacheGetOrCreate()
{
    var cachedValue = _memoryCache.GetOrCreate(
        CacheKeys.Entry,
        cacheEntry =>
        {
            cacheEntry.SlidingExpiration = TimeSpan.FromSeconds(3);
            return DateTime.Now;
        });

    // ...
}

public async Task OnGetCacheGetOrCreateAsync()
{
    var cachedValue = await _memoryCache.GetOrCreateAsync(
        CacheKeys.Entry,
        cacheEntry =>
        {
            cacheEntry.SlidingExpiration = TimeSpan.FromSeconds(3);
            return Task.FromResult(DateTime.Now);
        });

    // ...
}
```

The following code calls Get to fetch the cached time:

C#

Copy

```
var cacheEntry = _memoryCache.Get<DateTime?>(CacheKeys.Entry);
```

The following code gets or creates a cached item with absolute expiration:

C#

Copy

```
var cachedValue = _memoryCache.GetOrCreate(
    CacheKeys.Entry,
    cacheEntry =>
    {
        cacheEntry.AbsoluteExpirationRelativeToNow =
TimeSpan.FromSeconds(20);
        return DateTime.Now;
```

```
    });
```

A cached item set with only a sliding expiration is at risk of never expiring. If the cached item is repeatedly accessed within the sliding expiration interval, the item never expires. Combine a sliding expiration with an absolute expiration to guarantee the item expires. The absolute expiration sets an upper bound on how long the item can be cached while still allowing the item to expire earlier if it isn't requested within the sliding expiration interval. If either the sliding expiration interval *or* the absolute expiration time pass, the item is evicted from the cache.

The following code gets or creates a cached item with both sliding *and* absolute expiration:

C#

Copy

```
var cachedValue = _memoryCache.GetOrCreate(
    CacheKeys.CallbackEntry,
    cacheEntry =>
    {
        cacheEntry.SlidingExpiration = TimeSpan.FromSeconds(3);
        cacheEntry.AbsoluteExpirationRelativeToNow =
TimeSpan.FromSeconds(20);
        return DateTime.Now;
    });
```

The preceding code guarantees the data won't be cached longer than the absolute time.

GetOrCreate, GetOrCreateAsync, and Get are extension methods in the CacheExtensions class. These methods extend the capability of IMemoryCache.

MemoryCacheEntryOptions

The following example:

- Sets the cache priority to CacheItemPriority.NeverRemove.
- Sets a PostEvictionDelegate that gets called after the entry is evicted from the cache. The callback is run on a different thread from the code that removes the item from the cache.

C#

Copy

```
public void OnGetCacheRegisterPostEvictionCallback()
{
    var memoryCacheEntryOptions = new MemoryCacheEntryOptions()
```

```
        .SetPriority(CacheItemPriority.NeverRemove)
        .RegisterPostEvictionCallback(PostEvictionCallback, _memoryCache);

    _memoryCache.Set(CacheKeys.CallbackEntry, DateTime.Now,
memoryCacheEntryOptions);
}

private static void PostEvictionCallback(
    object cacheKey, object cacheValue, EvictionReason evictionReason,
object state)
{
    var memoryCache = (IMemoryCache)state;

    memoryCache.Set(
        CacheKeys.CallbackMessage,
        $"Entry {cacheKey} was evicted: {evictionReason}.");
}
```

Use SetSize, Size, and SizeLimit to limit cache size

A `MemoryCache` instance may optionally specify and enforce a size limit. The cache size limit doesn't have a defined unit of measure because the cache has no mechanism to measure the size of entries. If the cache size limit is set, all entries must specify size. The ASP.NET Core runtime doesn't limit cache size based on memory pressure. It's up to the developer to limit cache size. The size specified is in units the developer chooses.

For example:

- If the web app was primarily caching strings, each cache entry size could be the string length.
- The app could specify the size of all entries as 1, and the size limit is the count of entries.

If SizeLimit isn't set, the cache grows without bound. The ASP.NET Core runtime doesn't trim the cache when system memory is low. Apps must be architected to:

- Limit cache growth.
- Call Compact or Remove when available memory is limited.

The following code creates a unitless fixed size MemoryCache accessible by dependency injection:

C#

Copy

```
public class MyMemoryCache
{
```

```
    public MemoryCache Cache { get; } = new MemoryCache(
        new MemoryCacheOptions
        {
            SizeLimit = 1024
        });
}
```

`SizeLimit` doesn't have units. Cached entries must specify size in whatever units they consider most appropriate if the cache size limit has been set. All users of a cache instance should use the same unit system. An entry won't be cached if the sum of the cached entry sizes exceeds the value specified by `SizeLimit`. If no cache size limit is set, the cache size set on the entry is ignored.

The following code registers `MyMemoryCache` with the dependency injection container:

C#

Copy

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddSingleton<MyMemoryCache>();
```

`MyMemoryCache` is created as an independent memory cache for components that are aware of this size limited cache and know how to set cache entry size appropriately.

The size of the cache entry can be set using the SetSize extension method or the Size property:

C#

Copy

```
if (!_myMemoryCache.Cache.TryGetValue(CacheKeys.Entry, out DateTime
cacheValue))
{
    var cacheEntryOptions = new MemoryCacheEntryOptions()
        .SetSize(1);

    // cacheEntryOptions.Size = 1;

    _myMemoryCache.Cache.Set(CacheKeys.Entry, cacheValue,
cacheEntryOptions);
}
```

In the preceding code, the two highlighted lines achieve the same result of setting the size of the cache entry. `SetSize` is provided for convenience when chaining calls onto `new MemoryCacheOptions()`.

MemoryCache.Compact

`MemoryCache.Compact` attempts to remove the specified percentage of the cache in the following order:

- All expired items.
- Items by priority. Lowest priority items are removed first.
- Least recently used objects.
- Items with the earliest absolute expiration.
- Items with the earliest sliding expiration.

Pinned items with priority NeverRemove are never removed. The following code removes a cache item and calls `Compact` to remove 25% of cached entries:

C#

Copy

```
_myMemoryCache.Cache.Remove(CacheKeys.Entry);
_myMemoryCache.Cache.Compact(.25);
```

For more information, see the Compact source on GitHub.

Cache dependencies

The following sample shows how to expire a cache entry if a dependent entry expires. A CancellationChangeToken is added to the cached item. When `Cancel` is called on the `CancellationTokenSource`, both cache entries are evicted:

C#

Copy

```
public void OnGetCacheCreateDependent()
{
    var cancellationTokenSource = new CancellationTokenSource();

    _memoryCache.Set(
        CacheKeys.DependentCancellationTokenSource,
        cancellationTokenSource);

    using var parentCacheEntry =
_memoryCache.CreateEntry(CacheKeys.Parent);

    parentCacheEntry.Value = DateTime.Now;

    _memoryCache.Set(
        CacheKeys.Child,
```

```
        DateTime.Now,
        new CancellationChangeToken(cancellationTokenSource.Token));
}

public void OnGetCacheRemoveDependent()
{
    var cancellationTokenSource =
_memoryCache.Get<CancellationTokenSource>(
        CacheKeys.DependentCancellationTokenSource);

    cancellationTokenSource.Cancel();
}
```

Using a CancellationTokenSource allows multiple cache entries to be evicted as a group. With the `using` pattern in the code above, cache entries created inside the `using` scope inherit triggers and expiration settings.

Additional notes

- Expiration doesn't happen in the background. There's no timer that actively scans the cache for expired items. Any activity on the cache (`Get`, `TryGetValue`, `Set`, `Remove`) can trigger a background scan for expired items. A timer on the `CancellationTokenSource` (CancelAfter) also removes the entry and triggers a scan for expired items. The following example uses CancellationTokenSource(TimeSpan) for the registered token. When this token fires, it removes the entry immediately and fires the eviction callbacks:

- C#
- Copy

```
if (!_memoryCache.TryGetValue(CacheKeys.Entry, out DateTime cacheValue))
{
    cacheValue = DateTime.Now;

    var cancellationTokenSource = new CancellationTokenSource(
        TimeSpan.FromSeconds(10));

    var cacheEntryOptions = new MemoryCacheEntryOptions()
        .AddExpirationToken(
            new CancellationChangeToken(cancellationTokenSource.Token))
        .RegisterPostEvictionCallback((key, value, reason, state) =>
        {
            ((CancellationTokenSource)state).Dispose();
        }, cancellationTokenSource);

    _memoryCache.Set(CacheKeys.Entry, cacheValue, cacheEntryOptions);
```

```
}
```

- When using a callback to repopulate a cache item:
    - Multiple requests can find the cached key value empty because the callback hasn't completed.
    - This can result in several threads repopulating the cached item.
- When one cache entry is used to create another, the child copies the parent entry's expiration tokens and time-based expiration settings. The child isn't expired by manual removal or updating of the parent entry.
- Use PostEvictionCallbacks to set the callbacks that will be fired after the cache entry is evicted from the cache.
- For most apps, `IMemoryCache` is enabled. For example, calling `AddMvc`, `AddControllersWithViews`, `AddRazorPages`, `AddMvcCore().AddRazorViewEngine`, and many other `Add{Service}` methods in `Program.cs`, enables `IMemoryCache`. For apps that don't call one of the preceding `Add{Service}` methods, it may be necessary to call AddMemoryCache in `Program.cs`.

Background cache update

Use a background service such as IHostedService to update the cache. The background service can recompute the entries and then assign them to the cache only when they’re ready.