Cache in-memory in ASP.NET Core

02/02/2020 • 22 minutes to read •      +15

In this article

By Rick Anderson , John Luo , and Steve Smith

View or download sample code (how to download)

Caching basics

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 subsequent requests from a client all go to the same server. For example, Azure Web apps use Application Request Routing (ARR) to route all subsequent 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 2.0 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** use external input as cache keys.
  - 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. For example, Entity Framework Core uses the shared cache and does not specify a size. If an app sets a cache size limit and uses EF Core, the app throws an `InvalidOperationException`. 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. For example, EF Core uses the shared cache and does not specify a size.

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 HomeController : Controller
{
    private IMemoryCache _cache;

    public HomeController(IMemoryCache memoryCache)
    {
        _cache = 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. The `CacheKeys` class is part of the download sample.

C# Copy

```
public static class CacheKeys
{
    public static string Entry { get { return "_Entry"; } }
    public static string CallbackEntry { get { return "_Callback"; } }
    public static string CallbackMessage { get { return "_CallbackMessage"; } }
    public static string Parent { get { return "_Parent"; } }
    public static string Child { get { return "_Child"; } }
    public static string DependentMessage { get { return "_DependentMessage"; } }
    public static string DependentCTS { get { return "_DependentCTS"; } }
    public static string Ticks { get { return "_Ticks"; } }
    public static string CancelMsg { get { return "_CancelMsg"; } }
    public static string CancelTokenSource { get { return "_CancelTokenSource"; } }
}
```

C# Copy

```
public IActionResult CacheTryGetValueSet()
{
    DateTime cacheEntry;

    // Look for cache key.
    if (!_cache.TryGetValue(CacheKeys.Entry, out cacheEntry))
    {
        // Key not in cache, so get data.
        cacheEntry = DateTime.Now;

        // Set cache options.
        var cacheEntryOptions = new MemoryCacheEntryOptions()
            // Keep in cache for this time, reset time if accessed.
            .SetSlidingExpiration(TimeSpan.FromSeconds(3));


        // Save data in cache.
        _cache.Set(CacheKeys.Entry, cacheEntry, cacheEntryOptions);
    }

    return View("Cache", cacheEntry);
}
```

The current time and the cached time are displayed:

CSHTML Copy

```
@model DateTime?

<div>
    <h2>Actions</h2>
    <ul>
        <li><a asp-controller="Home" asp-action="CacheTryGetValueSet">TryGetValue and Set</a></li>
        <li><a asp-controller="Home" asp-action="CacheGet">Get</a></li>
        <li><a asp-controller="Home" asp-action="CacheGetOrCreate">GetOrCreate</a></li>
        <li><a asp-controller="Home" asp-
action="CacheGetOrCreateAsynchronous">CacheGetOrCreateAsynchronous</a></li>
        <li><a asp-controller="Home" asp-action="CacheRemove">Remove</a></li>
```

```
        <li><a asp-controller="Home" asp-action="CacheGetOrCreateAbs">CacheGetOrCreateAbs</a></li>
        <li><a asp-controller="Home" asp-action="CacheGetOrCreateAbsSliding">CacheGetOrCreateAbsSliding</a>
</li>

    </ul>
</div>

<h3>Current Time: @DateTime.Now.TimeOfDay.ToString()</h3>
<h3>Cached Time: @(Model == null ? "No cached entry found" : Model.Value.TimeOfDay.ToString())</h3>
```

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

C# Copy

```
public IActionResult SetCacheRelativeExpiration()
{
    DateTime cacheEntry;

    // Look for cache key.
    if (!_cache.TryGetValue(CacheKeys.Entry, out cacheEntry))
    {
        // Key not in cache, so get data.
        cacheEntry = DateTime.Now;

        // Save data in cache and set the relative expiration time to one day
        _cache.Set(CacheKeys.Entry, cacheEntry, TimeSpan.FromDays(1));
    }

    return View("Cache", cacheEntry);
}
```

The cached `DateTime` value remains in the cache while there are requests within the timeout period.

The following code uses GetOrCreate and GetOrCreateAsync to cache data

The following code uses GetOrCreate and GetOrCreateAsync to cache data:

C# Copy

```
public IActionResult CacheGetOrCreate()
{
    var cacheEntry = _cache.GetOrCreate(CacheKeys.Entry, entry =>
    {
        entry.SlidingExpiration = TimeSpan.FromSeconds(3);
        return DateTime.Now;
    });

    return View("Cache", cacheEntry);
}


public async Task<IActionResult> CacheGetOrCreateAsynchronous()
{
    var cacheEntry = await
        _cache.GetOrCreateAsync(CacheKeys.Entry, entry =>
        {
            entry.SlidingExpiration = TimeSpan.FromSeconds(3);
            return Task.FromResult(DateTime.Now);
        });

    return View("Cache", cacheEntry);
}
```

The following code calls Get to fetch the cached time:

C# Copy

```
public IActionResult CacheGet()
{
    var cacheEntry = _cache.Get<DateTime?>(CacheKeys.Entry);
    return View("Cache", cacheEntry);
}
```

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

C# Copy

```
public IActionResult CacheGetOrCreateAbs()
{
    var cacheEntry = _cache.GetOrCreate(CacheKeys.Entry, entry =>
    {
        entry.AbsoluteExpirationRelativeToNow = TimeSpan.FromSeconds(10);
        return DateTime.Now;
    });

    return View("Cache", cacheEntry);
}
```

A cached item set with a sliding expiration only is at risk of becoming stale. If it's accessed more frequently than the sliding expiration interval, the item will never expire. Combine a sliding expiration with an absolute expiration to guarantee that the item expires once its absolute expiration time passes. The absolute expiration sets an upper bound to 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. When both absolute and sliding expiration are specified, the expirations are logically ORed. 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

```
public IActionResult CacheGetOrCreateAbsSliding()
{
    var cacheEntry = _cache.GetOrCreate(CacheKeys.Entry, entry =>
    {
        entry.SetSlidingExpiration(TimeSpan.FromSeconds(3));
        entry.AbsoluteExpirationRelativeToNow = TimeSpan.FromSeconds(20);
        return DateTime.Now;
    });
```

```
    return View("Cache", cacheEntry);
}
```

The preceding code guarantees the data will not 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 sample:

- Sets a sliding expiration time. Requests that access this cached item will reset the sliding expiration clock.
- Sets the cache priority to CacheItemPriority.NeverRemove.
- Sets a PostEvictionDelegate that will be 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 IActionResult CreateCallbackEntry()
{
    var cacheEntryOptions = new MemoryCacheEntryOptions()
        // Pin to cache.
        .SetPriority(CacheItemPriority.NeverRemove)
        // Add eviction callback
        .RegisterPostEvictionCallback(callback: EvictionCallback, state: this);

    _cache.Set(CacheKeys.CallbackEntry, DateTime.Now, cacheEntryOptions);

    return RedirectToAction("GetCallbackEntry");
}

public IActionResult GetCallbackEntry()
{
```

```
    return View("Callback", new CallbackViewModel
    {
        CachedTime = _cache.Get<DateTime?>(CacheKeys.CallbackEntry),
        Message = _cache.Get<string>(CacheKeys.CallbackMessage)
    });
}

public IActionResult RemoveCallbackEntry()
{
    _cache.Remove(CacheKeys.CallbackEntry);
    return RedirectToAction("GetCallbackEntry");
}

private static void EvictionCallback(object key, object value,
    EvictionReason reason, object state)
{
    var message = $"Entry was evicted. Reason: {reason}.";
    ((HomeController)state)._cache.Set(CacheKeys.CallbackMessage, message);
}
```

Use SetSize, Size, and SizeLimit to limit cache size

A `MemoryCache` instance may optionally specify and enforce a size limit. The cache size limit does not 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 does not 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

```
// using Microsoft.Extensions.Caching.Memory;
public class MyMemoryCache
{
    public MemoryCache Cache { get; set; }
    public MyMemoryCache()
    {
        Cache = new MemoryCache(new MemoryCacheOptions
        {
            SizeLimit = 1024
        });
    }
}
```

`SizeLimit` does not have units. Cached entries must specify size in whatever units they deem most appropriate if the cache size limit has been set. All users of a cache instance should use the same unit system. An entry will not 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 will be ignored.

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

C# Copy

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages();
    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 following code uses `MyMemoryCache`:

C# Copy

```
public class SetSize : PageModel
{
    private MemoryCache _cache;
    public static readonly string MyKey = "_MyKey";

    public SetSize(MyMemoryCache memoryCache)
    {
        _cache = memoryCache.Cache;
    }

    [TempData]
    public string DateTime_Now { get; set; }

    public IActionResult OnGet()
    {
        if (!_cache.TryGetValue(MyKey, out string cacheEntry))
        {
            // Key not in cache, so get data.
            cacheEntry = DateTime.Now.TimeOfDay.ToString();

            var cacheEntryOptions = new MemoryCacheEntryOptions()
                // Set cache entry size by extension method.
                .SetSize(1)
                // Keep in cache for this time, reset time if accessed.
                .SetSlidingExpiration(TimeSpan.FromSeconds(3));

            // Set cache entry size via property.
            // cacheEntryOptions.Size = 1;
```

```
            // Save data in cache.
            _cache.Set(MyKey, cacheEntry, cacheEntryOptions);
        }

        DateTime_Now = cacheEntry;

        return RedirectToPage("./Index");
    }
}
```

The size of the cache entry can be set by Size or the SetSize extension methods:

C# Copy

```
public IActionResult OnGet()
{
    if (!_cache.TryGetValue(MyKey, out string cacheEntry))
    {
        // Key not in cache, so get data.
        cacheEntry = DateTime.Now.TimeOfDay.ToString();

        var cacheEntryOptions = new MemoryCacheEntryOptions()
            // Set cache entry size by extension method.
            .SetSize(1)
            // Keep in cache for this time, reset time if accessed.
            .SetSlidingExpiration(TimeSpan.FromSeconds(3));

        // Set cache entry size via property.
        // cacheEntryOptions.Size = 1;

        // Save data in cache.
        _cache.Set(MyKey, cacheEntry, cacheEntryOptions);
    }

    DateTime_Now = cacheEntry;

    return RedirectToPage("./Index");
```

```
}
```

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`:

C# Copy

```
_cache.Remove(MyKey);

// Remove 33% of cached items.
_cache.Compact(.33);
cache_size = _cache.Count;
```

See Compact source on GitHub for more information.

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 IActionResult CreateDependentEntries()
{
    var cts = new CancellationTokenSource();
    _cache.Set(CacheKeys.DependentCTS, cts);

    using (var entry = _cache.CreateEntry(CacheKeys.Parent))
    {
        // expire this entry if the dependant entry expires.
        entry.Value = DateTime.Now;
        entry.RegisterPostEvictionCallback(DependentEvictionCallback, this);

        _cache.Set(CacheKeys.Child,
            DateTime.Now,
            new CancellationChangeToken(cts.Token));
    }

    return RedirectToAction("GetDependentEntries");
}

public IActionResult GetDependentEntries()
{
    return View("Dependent", new DependentViewModel
    {
        ParentCachedTime = _cache.Get<DateTime?>(CacheKeys.Parent),
        ChildCachedTime = _cache.Get<DateTime?>(CacheKeys.Child),
        Message = _cache.Get<string>(CacheKeys.DependentMessage)
    });
}

public IActionResult RemoveChildEntry()
{
    _cache.Get<CancellationTokenSource>(CacheKeys.DependentCTS).Cancel();
    return RedirectToAction("GetDependentEntries");
}

private static void DependentEvictionCallback(object key, object value,
    EvictionReason reason, object state)
{
```

```
    var message = $"Parent entry was evicted. Reason: {reason}.";
    ((HomeController)state)._cache.Set(CacheKeys.DependentMessage, message);
}
```

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` block will inherit triggers and expiration settings.

Additional notes

- Expiration doesn't happen in the background. There is no timer that actively scans the cache for expired items. Any activity on the cache (`Get`, `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

```
public IActionResult CacheAutoExpiringTryGetValueSet()
{
    DateTime cacheEntry;

    if (!_cache.TryGetValue(CacheKeys.Entry, out cacheEntry))
    {
        cacheEntry = DateTime.Now;

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

        var cacheEntryOptions = new MemoryCacheEntryOptions()
            .AddExpirationToken(new CancellationChangeToken(cts.Token));

        _cache.Set(CacheKeys.Entry, cacheEntry, cacheEntryOptions);
    }
```

```
    return View("Cache", cacheEntry);
}
```

- 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 `ConfigureServices`, enables `IMemoryCache`. For apps that are not calling one of the preceding `Add{Service}` methods, it may be necessary to call AddMemoryCache in `ConfigureServices`.

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.

Additional resources

- Distributed caching in ASP.NET Core
- Detect changes with change tokens in ASP.NET Core
- Response caching in ASP.NET Core
- Response Caching Middleware in ASP.NET Core