In Memory Caching in ASP.NET Core

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. Caching makes a copy of data that can be returned much faster than from the original source. You should write and test your app to never depend on cached data.

ASP.NET Core supports several different caches. The simplest cache is based on the [IMemoryCache](https://docs.microsoft.com/aspnet/core/api/microsoft.extensions.caching.memory.imemorycache), which represents a cache stored in the memory of the web server. Apps which run on a server farm of multiple servers should ensure that 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](https://www.iis.net/learn/extensions/planning-for-arr) (ARR) to route all subsequent requests to the same server.

Non-sticky sessions in a web farm require a [distributed cache](https://docs.microsoft.com/en-us/aspnet/core/performance/caching/distributed) 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 IMemoryCache cache will evict cache entries under memory pressure unless the [cache priority](https://docs.microsoft.com/aspnet/core/api/microsoft.extensions.caching.memory.cacheitempriority)is set to CacheItemPriority.NeverRemove. You can set the CacheItemPriority to adjust the priority the cache evicts items under memory pressure.

The in-memory cache can store any object; the distributed cache interface is limited to byte[].

Using IMemoryCache

In-memory caching is a *service* that is referenced from your app using [Dependency Injection](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection). Call AddMemoryCache in ConfigureServices:

C#Copy

using Microsoft.AspNetCore.Builder;

using Microsoft.Extensions.DependencyInjection;

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

services.AddMemoryCache();

services.AddMvc();

}

public void Configure(IApplicationBuilder app)

{

app.UseMvcWithDefaultRoute();

}

}

Request the IMemoryCache instance in the constructor:

C#Copy

public class HomeController : Controller

{

private IMemoryCache \_cache;

public HomeController(IMemoryCache memoryCache)

{

\_cache = memoryCache;

}

IMemoryCache requires NuGet package "Microsoft.Extensions.Caching.Memory".

The following code uses [TryGetValue](https://docs.microsoft.com/aspnet/core/api/microsoft.extensions.caching.memory.imemorycache" \l "Microsoft_Extensions_Caching_Memory_IMemoryCache_TryGetValue_System_Object_System_Object__) to check if the current time is in the cache. If the item is not cached, a new entry is created and added to the cache with [Set](https://docs.microsoft.com/aspnet/core/api/microsoft.extensions.caching.memory.cacheextensions#Microsoft_Extensions_Caching_Memory_CacheExtensions_Set__1_Microsoft_Extensions_Caching_Memory_IMemoryCache_System_Object___0_).

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);

}

**CacheKeys.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

namespace CacheDemo

{

public static class CacheKeys

{

public static readonly string Entry = "\_Entry";

public static readonly string CallbackEntry = "\_Callback";

public static readonly string CallbackMessage = "\_CallbackMessage";

public static readonly string Parent = "\_Parent";

public static readonly string Child = "\_Child";

public static readonly string DependentMessage = "\_DependentMessage";

public static readonly string DependentCTS = "\_DependentCTS";

public static string Ticks { get { return "\_Ticks"; } }

public static string CancelMsg { get { return "\_CancelMsg"; } }

public static string CancelTokenSource { get { return "\_CancelTokenSource"; } }

}

}

The current time and the cached time is displayed:

htmlCopy

@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="CacheGetOrCreateAsync">GetOrCreateAsync</a></li>

<li><a asp-controller="Home" asp-action="CacheRemove">Remove</a></li>

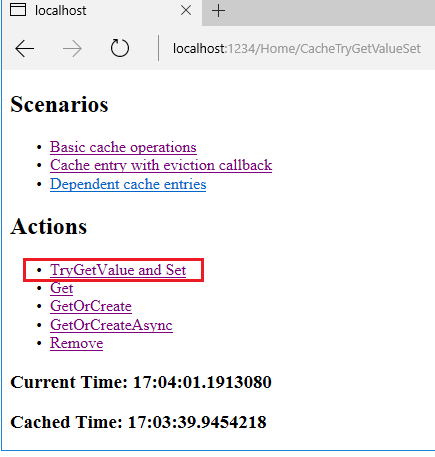
</ul>

</div>

<h3>Current Time: @DateTime.Now.TimeOfDay.ToString()</h3>

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

The cached DateTime value will remain in the cache while there are requests within the timeout period (and no eviction due to memory pressure). The image below shows the current time and an older time retrieved from cache:



The following code uses [GetOrCreate](https://docs.microsoft.com/aspnet/core/api/microsoft.extensions.caching.memory.cacheextensions" \l "Microsoft_Extensions_Caching_Memory_CacheExtensions_GetOrCreate__1_Microsoft_Extensions_Caching_Memory_IMemoryCache_System_Object_System_Func_Microsoft_Extensions_Caching_Memory_ICacheEntry___0__) and [GetOrCreateAsync](https://docs.microsoft.com/aspnet/core/api/microsoft.extensions.caching.memory.cacheextensions" \l "Microsoft_Extensions_Caching_Memory_CacheExtensions_GetOrCreateAsync__1_Microsoft_Extensions_Caching_Memory_IMemoryCache_System_Object_System_Func_Microsoft_Extensions_Caching_Memory_ICacheEntry_System_Threading_Tasks_Task___0___) 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> CacheGetOrCreateAsync()

{

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](https://docs.microsoft.com/aspnet/core/api/microsoft.extensions.caching.memory.cacheextensions#Microsoft_Extensions_Caching_Memory_CacheExtensions_Get__1_Microsoft_Extensions_Caching_Memory_IMemoryCache_System_Object_) to fetch the cached time:

C#Copy

public IActionResult CacheGet()

{

var cacheEntry = \_cache.Get<DateTime?>(CacheKeys.Entry);

return View("Cache", cacheEntry);

}

See [IMemoryCache methods](https://docs.microsoft.com/aspnet/core/api/microsoft.extensions.caching.memory.imemorycache) and [CacheExtensions methods](https://docs.microsoft.com/aspnet/core/api/microsoft.extensions.caching.memory.cacheextensions) for a description of the cache methods.

Using MemoryCacheEntryOptions

The following sample:

* Sets the absolute expiration time. This is the maximum time the entry can be cached and prevents the item from becoming too stale when the sliding expiration is continuously renewed.
* 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](https://docs.microsoft.com/aspnet/core/api/microsoft.extensions.caching.memory.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);

}

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

* 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 is not expired by manual removal or updating of the parent entry.