.NET Core In-Memory Cache

Here is a working method for using the In-Memory cache.

For a thread-safe cache update, see the example at the bottom.

Assimilate this into a working project:

In-Memory Caching in ASP.NET Core - Code Maze

Understanding & Implementing Caching in ASP.NET Core | Mitchel Sellers

Distributed caching in ASP.NET Core | Microsoft Learn

Response Caching Middleware in ASP.NET Core | Microsoft Learn

Cache in-memory in ASP.NET Core | Microsoft Learn

In-Use Example

Here’s an implementation of the Memory Cache, as it serves a lookup for user group memberships.

> NOTE: This example is not thread-safe for writes. See the bottom of this page for an example with thread-safe cache updates.

```
public string[] GetGroups_for_UserID(string userid)
{
    // Formulate a cache key for the user context...
    // We prepend the type of entry in the key, to ensure no overlap occurs with other cache entry types.
    string usecckey = "usec:" + userid;

    // First, check the cache for a hit...
    if (this._cache.TryGetValue(usecckey, out string[] data))
    {
```

```
        // We got a cache hit.
        // Will use that.
    }
    else
    {
        // Not in the cache.
        // We will look it up, and cache it for later.

        // Lookup the groups for the user...
        data = SomeRESTCLientCalltoLookupUSerSecContext(userid);

        // And, store it in the cache...
        var ceo = MOVETHISELSEWHERE.CachingOptions.Get_MemCacheOptions();
        this._cache.Set(usecckey, data, ceo);
    }

    // Return the group data we collected...
    return data;
}
```

The above method sits in a service that accepts a memory cache instance in its constructor, like this:

```
public class UserSecurityService : IUserSecurityService
{
    IMemoryCache _cache;

    public UserSecurityService(IMemoryCache cache)
    {
        this._cache = cache;
    }
```

As well, each cache entry add requires a set of options.
These are configured as such:

```
static public MemoryCacheEntryOptions Get_MemCacheOptions()
{
    var ceo = new MemoryCacheEntryOptions()
        .SetSlidingExpiration(TimeSpan.FromSeconds(60))
        .SetAbsoluteExpiration(TimeSpan.FromSeconds(3600))
        .SetPriority(CacheItemPriority.Normal)
```

```
        .SetSize(1024);

    return ceo;
}
```

NOTE: The above call is an easily static method that can be used anywhere, that a default cache policy needs.

And last, using the memory cache in a top-level binary does require DI registration of the memory class.

Here’s an example of how to add the memory cache to DI services:

```
// Since we do leverage Memory Cache in Security Directory and UserContext lookups,
//    we need to ensure the caching service is available...
// This includes services such as: DirectoryService_Adapter_for_USC and cUserSecurityService.
// And, we add it here, in case memory cache was not included in the choice of API level (above).
services.AddMemoryCache();
```

Call that in your startup.cs

Thread Safe Cache Updates

If your implementation requires thread-safe cache updates, as most real-world implementations do, then follow this variation of the `GetGroups_for_UserID` method example:

Here’s the source article that explains the reasons for the double check: Async-Lock Mechanism on Asynchronous Programing

```
public string[] GetGroups_for_UserID(string userid)
{
    // Formulate a cache key for the user context...
    // We prepend the type of entry in the key, to ensure no overlap occurs with other cache entry types.
    string usecckey = "usec:" + userid;

    // First, check the cache for a hit...
    // See this for memory cache implementation notes:
    //  https://oga.atlassian.net/wiki/spaces/~311198967/pages/93159425/NET+Core+In-Memory+Cache
    if (!this._cache.TryGetValue(usecckey, out string[] data))
    {
```

```
        // Key was not found in the cache.
        // We will enter an update lock, and check again, to ensure we are the only thread updating the entry.
        // The reason we do this check-lock-check again flow, is two-fold:
        //  For a Cache-Hit: The outer check doesn't impose a lock, and remains very fast.
        //  For a Cache-Miss: The outer check falls into our sync lock that will ensure that one-and-only-one thread gets to perform a true cache check and update.

        // Enter a sync lock before checking again...
        lock (this._updatelock)
        {
            // Inside the sync lock.

            // Check to see if still a cache miss...
            if (!this._cache.TryGetValue(usecckey, out data))
            {
                // Still a cache miss.
                // And, we are the only thread that can see that, and update it.
                // So, we will attempt to update the cache.

                // We will look it up, and cache it for later.

                // Lookup the groups for the user...

                // Populate a test set of groups...
                data = new string[] { "admin" };

                // And, store it in the cache...
                var ceo = MOVETHISELSEWHERE.CachingOptions.Get_ShortDuration_MemCacheOptions();
                this._cache.Set(usecckey, data, ceo);
            }
            else
            {
                // Got a cache-hit from inside the lock.
                // This means, another thread was inside this lock block, filling the cache, while we saw the first cache-miss.
                // And, that other thread (that filled the cache) left the sync lock block before we got in to check again.
                // So, we don't may not need to populate the cache.

                // We will let the logic flow fall out, below...
            }
```

```
        }
        // Bottom of the update lock.
    }
    else
    {
        // We got a cache hit.
        // We will return that value...
    }

    // Return the group data we collected...
    return data;
}
```

> NOTE: The above example does a cache check, twice.
> This is for performance reasons.

The first check is has no lock penalty, and if a cache-hit, moves as quickly as possible.

However. If a cache-miss was seen, the thread enters the update lock block, where it must do a second check for a cache miss.

This second cache check, being done from inside the sync lock, ensures that the cache is not being populated while being verified as a cache-miss.

Once confirmed (with the second check), the real data is requested, and pushed into the cache.

Then, the logic flow falls to the bottom to return the value.

Revision #2
Created 22 August 2025 20:19:46 by glwhite
Updated 22 August 2025 20:23:06 by glwhite