Agenda: Web Caching

- Cache Tag Helpers
- Memory Caching Introduction
- In-Memory Caching
- Response Cache

Cache Tag Helpers

The Cache tag helper is one of the more unique tag helpers in ASP.NET Core. It provides a flexible and convenient approach to caching the output of a portion of a page and can be a useful tool for improving the performance of MVC applications.

A. Cache Expiry

If no specific expiry is specified, the contents will be cached as long as the memory cache decides to hang on to the item which is likely the lifetime of the application. Chances are this is not the behavior you want. You will likely want to use one of the 3 options for expiring the cached contents for the Cache Tag Helper: **expires-after**, **expires-on** and **expires-sliding**.

1. expires-after:

Use the expires-after attribute to expire the cache entry after a specific amount of time has passed since it was added to the cache. This attribute expects a TimeSpan value.

For example, you expire an item 5 seconds after it was cached:

```
<cache expires-after="@TimeSpan.FromSeconds(5)">
    @DateTime.Now.ToLocalTime()
</cache>
```

2. expires-sliding:

Use the expires-sliding attribute to expire the cache entry after it has not been accessed for a specified amount of time. This attribute expects a TimeSpan value.

```
<cache expires-sliding="@TimeSpan.FromMinutes(5)">
    @DateTime.Now.ToLocalTime()
</cache>
```

3. expires-on

Use the expires-on attribute to expire the cache entry at a specific time. This attribute expects a DateTime value

Example: Imagine your system has some backend processing that you know will be updated by the end of each day. You could specify the cache to expire at the end of the day as follows:

```
<cache expires-on="@DateTime.Today.AddDays(1).AddTicks(-1)">
    @DateTime.Now.ToLocalTime()
</cache>
```

B. Vary-by / Complex Cache Keys

The cache tag helper builds cache keys by generating an id that is unique to the context of the cache tag.

1. vary-by-user:

Use this attribute to cache different contents for each logged in user. The username for the logged in user will be added to the cache key. This attribute expects a Boolean value.

```
<cache vary-by-user="true">
   @DateTime.Now. ToLocalTime()
</cache>
```

2. vary-by-route

Use this attribute to cache different contents based on a set of route data parameters. This attribute expects a comma-separated list of route data parameter names will be added to the cache key.

```
<cache vary-by-route="id">
   @DateTime.Now.ToLocalTime()
</cache>
```

C. Cache Priority:

The contents of a cache tag are stored in an **IMemoryCache** which is limited by the amount of available memory. If the host process starts to run out of memory, the memory cache might purge items from the cache to release memory. In cases like this, you can tell the memory cache which items are considered a lower priority using the priority attribute. For example, the following cache tag is specified as low priority:

```
<cache vary-by-user="true" priority="@CacheItemPriority.Low">
   @DateTime.Now.ToLocalTime()
</cache>
```

Note: Possible values for CacheItemPriority are **Low, Normal, High and NeverRemove**.

Setting this attribute to **NeverRemove** doesn't guarantee that the cache will always be retained.

In-Memory Caching - IMemoryCache

.NET Core came up with caching API under the Namespace **Microsoft.Extensions.Caching.Memory**.

- In-memory caching holds the copy of data in local server's memory.

Steps to Implement In-Memory Caching:

Step1: The In-Memory caching is a service called by dependency injection in the application, so we register it in the ConfigureServices method of Startup class, as per the following code snippet.

```
public void ConfigureServices(IServiceCollection services)
{
```

```
        services.AddMvc();
        services.AddMemoryCache();
}
```

Step2: In-Memory cache, we create a Controller named HomeController. This Controller holds the implementation of the In-Memory cache.

Now, we create ImemoryCache interface instance in the HomeController using constructor dependency injection.

```
private readonly IMemoryCache memoryCache;
public HomeController(IMemoryCache memoryCache)
{
    this.memoryCache = memoryCache;
}
```

Step3: we need to create an action method named Index. This action method sets data in the cache.

```
public IActionResult Index()
{
   string Time = null;
   if (memoryCache.TryGetValue("Time", out Time))
   {
      ViewBag.Time = Time;
   }
   else
   {
      Time = DateTime.Now.ToLocalTime().ToString();
      memoryCache.Set("Time", Time, new MemoryCacheEntryOptions()
      {
         AbsoluteExpirationRelativeToNow = (DateTime.Now.AddSeconds(10) - DateTime.Now)
      });
      ViewBag.Time = Time;
   }
   return View();
}
```

Index.cshtml:

```
@{
   ViewData["Title"] = "Index";
```

```
}
<h2>Cache Index</h2>
Current time: @ViewBag.Time
```

The other way to set the expire value as shown below.

```
this.memoryCache.Set("key","Value",
                        new MemoryCacheEntryOptions().SetAbsoluteExpiration(TimeSpan.FromMinutes(1)));
```

Setting Cache Priority:

By default, an instance of MemoryCache will automatically manage the items stored, removing entries when necessary in response to memory pressure in the app. You can influence the way cache entries are managed by setting their CacheItemPriority when adding the item to the cache. For example, if you have an item you want to keep in the cache unless you explicitly remove it, you can use the NeverRemove priority option.

```
new MemoryCacheEntryOptions().SetPriority(CacheItemPriority.NeverRemove))
```

Response caching

Response caching adds cache-related headers to responses. These headers specify how you want client, proxy and **middleware** to cache responses. It can drastically improve performance of your web application.

Step1: Include following NuGet package Microsoft.AspNetCore.ResponseCaching.

Step2: Once the package is restored, now we need to configure it. So open Startup.cs

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddResponseCaching();
    services.AddMvc();
}
```

Step3: Add this middleware to HTTP pipeline so add below code in the Configure method.

```
app.UseResponseCaching();
```

Step4: We are done with all configurations. To use it, you need to include ResponseCache attribute on controller's action method. So open HomeController.cs and add ResponseCache attribute to Contact method and set the duration to 20 seconds.

Controller:

```
[ResponseCache(Duration = 10)]
```

```
public IActionResult Index()
{
    ViewData["Message"] = "Current time: " + DateTime.Now.ToLocalTime();
    return View();
}
```

Index.cshtml:

```
@ViewData["Message"]
```

Step5: This attribute will set the **Cache-Control** header and set max-age to 10 seconds. The Cache-Control HTTP/1.1 general-header field is used to specify directives for caching mechanisms in both requests and responses. Use this header to define your caching policies with the variety of directives it provides. In our case, following header will be set.

```
Cache-Control:public,max-age=10
```

Public: Indicates that the response is allowed to cache by **any** caching mechanism. Mostly if the page is independent of user then we can make it publicly cacheable.

Private: The value of Cache-Control header is "private," which means it will not be cached by any intermediate cache server. We can specify "Location = ResponseCacheLocation.Client" along with ResponseCache to make it private.

```
[ResponseCache(Duration = 10, Location = ResponseCacheLocation.Client)]
public IActionResult Index()
{
      ViewData["Message"] = "Current time: " + DateTime.Now.ToLocalTime();
     return View();
}
```

No Cache: If we don't want to cache a response in any cache layer, we can specify the Location value as none.

```
[ResponseCache(Duration = 20, Location = ResponseCacheLocation.None)]
public IActionResult Index()
{
   ViewData["Message"] = "Current time: " + DateTime.Now.ToLocalTime();
      return View();
}
```

Properties of ResponseCache:

The ResponseCache attribute supports many parameters, in this section we will understand each one of them.

- **Duration:** The maximum time (in seconds) the response should be cached. Required unless NoStore is true.

- **Location:** The location where the response may be cached. May be Any, None, or Client. Default is Any. Based on this value, the value of the "Response-Cache" header changes.
- **NoStore**: Determines whether the value should be stored or not, and overrides other property values. When true, Duration is ignored and Location is ignored for values other than None.
- **VaryByHeader**: When set, a vary response header will be written with the response. When the value of the header will change, the cached version will be invalid.
- **CacheProfileName**: When set, determines the name of the cache profile to use. We can specify cache profile in Configuration section of ASP.NET 5 application.
- **Order:** The order of the filter (from IOrderedFilter).

Cache Profile: We can maintain the Cache profile in application level and use in action or controller level. The mechanism will reduce code repetition. In this example we have created one cache profile called private cache.

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc(options =>
    {
        options.CacheProfiles.Add("PrivateCache",
            new CacheProfile()
              {
                 Duration = 10,
                 Location = ResponseCacheLocation.Client
              });
          });
    }
}
```

Now, we can use the profile in controller/action level. Here is example code to implement same.

```
[ResponseCache(CacheProfileName = "PrivateCache")]
public ActionResult ResponseCache()
{
        return View("Index");
}
```