MICROSOFT ARCHITECT
By Joydip Kanjilal, Columnist, InfoWorld
OCT 9, 2017 3:00 AM PDT

How to use in-memory caching in ASP.NET Core

Take advantage of in-memory caching in ASP.NET Core to improve the performance and scalability of your application

ASP.NET Core is a lean and modular framework that can be used to build high-performance, modern web applications on Windows, Linux, or MacOS. Unlike legacy ASP.NET, ASP.NET Core doesn't have a `Cache` object. However, ASP.NET Core provides support for several different types of caching including in-memory caching, distributed caching, and response caching.

In this article, we'll look at how you can boost your ASP.NET Core application's performance and scalability by storing infrequently changing data in the in-memory cache. As always, I will include code examples to illustrate the concepts discussed.

How to enable in-memory caching in ASP.NET Core

The in-memory cache in ASP.NET Core is a service that you can incorporate into your application using dependency injection. Once you have created an ASP.NET Core project in Visual Studio, you can enable the in-memory cache in the `ConfigureServices` method in the `Startup` class as shown in the code snippet below.

[Also on InfoWorld: 5 signs your agile development process must change]

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddMemoryCache();
}
```

To work with the in-memory cache in ASP.NET Core, you need to use the `IMemoryCache` interface. Here is how it looks:

```
public interface IMemoryCache : IDisposable
{
    bool TryGetValue(object key, out object value);
    ICacheEntry CreateEntry(object key);
    void Remove(object key);
}
```

You can register the `IMemoryCache` in the `ConfigServices` method using the `AddMemoryCache` method we examined above. Then you should inject the cache object in the constructor of your controller class as shown in the code snippet below.

```
 private IMemoryCache cache;
 public IDGCacheController(IMemoryCache cache)
       {
           this.cache = cache;
       }
```

And that's all you need to do to set up support for in-memory caching in your ASP.NET Core application. In the section that follows, we will look at how we can work with the cache API in ASP.NET Core to store and retrieve objects.

How to store and retrieve objects using ASP.NET Core IMemoryCache

To store an object using the `IMemoryCache` interface you need to use the `Set<T>()` method as shown in the code snippet below. Note that the version of the `Set<T>()` method we have used in this example accepts two parameters. The first parameter is the name of the key and the second parameter is the value, i.e., the object that is to be stored in the cache that can be identified using the key.

```
[HttpGet]
        public string Get()
        {
            cache.Set("IDGKey", DateTime.Now.ToString());
            return "This is a test method...";
        }
```

To retrieve an item from the cache, you can take advantage of the `Get<T>()` method as shown below.

```
 [HttpGet("{key}")]
       public string Get(string key)
       {
           return cache.Get<string>(key);
       }
```

You can use the `TryGet()` method on the cache object to check if the specified key exists in the cache. Here is the modified version of our `Get` method that illustrates how this can be achieved.

```
[HttpGet]
       public string Get()
       {
           string key ="IDGKey";
           string obj;
           if (!cache.TryGetValue<string>(key, out obj))
           {
               obj = DateTime.Now.ToString();
               cache.Set<string>(key, obj);
           }
           return obj;
       }
```

There is another method, called `GetOrCreate`, that can be used to retrieve cached data based on the key provided. If the key doesn't exist, the method creates it.

```
[HttpGet]
       public string Get()
       {
           return cache.GetOrCreate<string>("IDGKey",
               cacheEntry => {
                           return DateTime.Now.ToString();
                        });
       }
```

Note that an asynchronous version of this method is available called `GetOrCreateAsync`. Here is the complete code listing of our `IDGCacheController` class for your reference.

```
using System;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Caching.Memory;

namespace InMemoryCaching.Controllers
{
    [Route("api/[controller]")]
    public class IDGCacheController : Controller
    {
        private IMemoryCache cache;
        public IDGCacheController(IMemoryCache cache)
        {
            this.cache = cache;
        }

        [HttpGet]
        public string Get()
        {
            return cache.GetOrCreate<string>("IDGKey",
                cacheEntry => {
                            return DateTime.Now.ToString();
                        });
        }
    }
}
```

How to set expiration policies on cached data in ASP.NET Core

Note that you can set absolute and sliding expiration policies on your cached data. While the former is used to specify the duration for which an object should reside in the cache, the latter is used to specify the duration for which an object will reside in the cache when there is no activity—i.e., the item will be removed from the cache when the specified duration of inactivity elapses.

To set expiration policies you use the `MemoryCacheEntryOptions` class as shown in the code snippet below.

```
MemoryCacheEntryOptions cacheExpirationOptions = new MemoryCacheEntryOptions();
cacheExpirationOptions.AbsoluteExpiration = DateTime.Now.AddMinutes(30);
cacheExpirationOptions.Priority = CacheItemPriority.Normal;
cache.Set<string>(“IDGKey”, DateTime.Now.ToString(), cacheExpirationOptions);
```

Note the usage of the `Priority` property on the `MemoryCacheEntryOptions` instance in the code snippet above. The `Priority` property specifies which objects (based on the priority already set) should be removed from the cache as part of a strategy of the runtime to reclaim memory whenever the web server runs out of memory space.

To set the priority, we used the `CacheItemPriority` enum. This can have one of these possible values: Low, Normal, High, and NeverRemove. The in-memory cache provider in ASP.NET Core will remove cache entries when under memory pressure unless you have set the cache priority to `CacheItemPriority.NeverRemove`.

You may also want to register a callback that will execute whenever an item is removed from the cache. The following code snippet illustrates how this can be achieved.

```
cacheExpirationOptions.RegisterPostEvictionCallback
(IDGCacheItemChangedHandler, this);
```

You can even set dependencies between the cached objects. As an example, you might want to remove certain items from the cache if some related item has been removed. We will explore this further and many other features of caching in ASP.NET Core in my future posts here. Until then, you may want to take a look at the relevant pages in Microsoft’s ASP.NET Core documentation.

How to do more in ASP.NET and ASP.NET Core:

- How to use in-memory caching in ASP.NET Core
- How to handle errors in ASP.NET Web API
- How to pass multiple parameters to Web API controller methods
- How to log request and response metadata in ASP.NET Web API
- How to work with HttpModules in ASP.NET

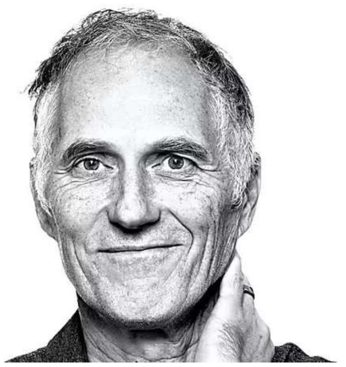
- Advanced versioning in ASP.NET Core Web API
- How to use dependency injection in ASP.NET Core
- How to work with sessions in ASP.NET
- How to work with HTTPHandlers in ASP.NET
- How to use IHostedService in ASP.NET Core
- How to consume a WCF SOAP service in ASP.NET Core
- How to improve the performance of ASP.NET Core applications
- How to consume an ASP.NET Core Web API using RestSharp
- How to work with logging in ASP.NET Core
- How to use MediatR in ASP.NET Core
- How to work with session state in ASP.NET Core
- How to use Nancy in ASP.NET Core
- Understand parameter binding in ASP.NET Web API
- How to upload files in ASP.NET Core MVC
- How to implement global exception handling in ASP.NET Core Web API
- How to implement health checks in ASP.NET Core
- Best practices in caching in ASP.NET
- How to use Apache Kafka messaging in .NET
- How to enable CORS on your Web API
- When to use WebClient vs. HttpClient vs. HttpWebRequest
- How to work with Redis Cache in .NET
- When to use Task.WaitAll vs. Task.WhenAll in .NET

Joydip Kanjilal is a Microsoft MVP in ASP.Net, as well as a speaker and author of several books and articles. He has more than 20 years of experience in IT including more than 16 years in Microsoft .Net and related technologies.

Follow

YOU MAY ALSO LIKE

Recommended by



Tim O'Reilly: the golden age of the programmer is over

7 best practices for remote development teams

Edge computing archetypes are emerging, and they are not pretty

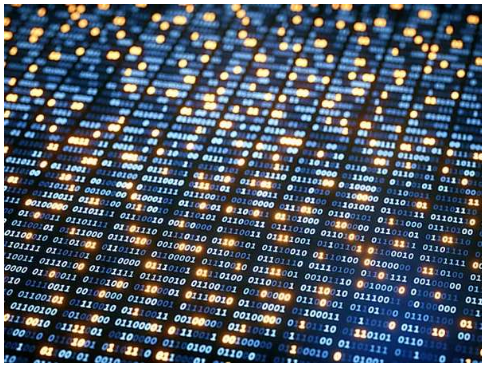
How IT priorities are shifting during the COVID-19 crisis

Make life easy with ssh_config

Microsoft Visual Studio 2022 preview is coming soon

Are industry clouds an opportunity or a

6 reasons to switch to managed Kubernetes



What is a computational storage drive? Much-

The 24 highest paying developer roles in 2020

The decline of Heroku