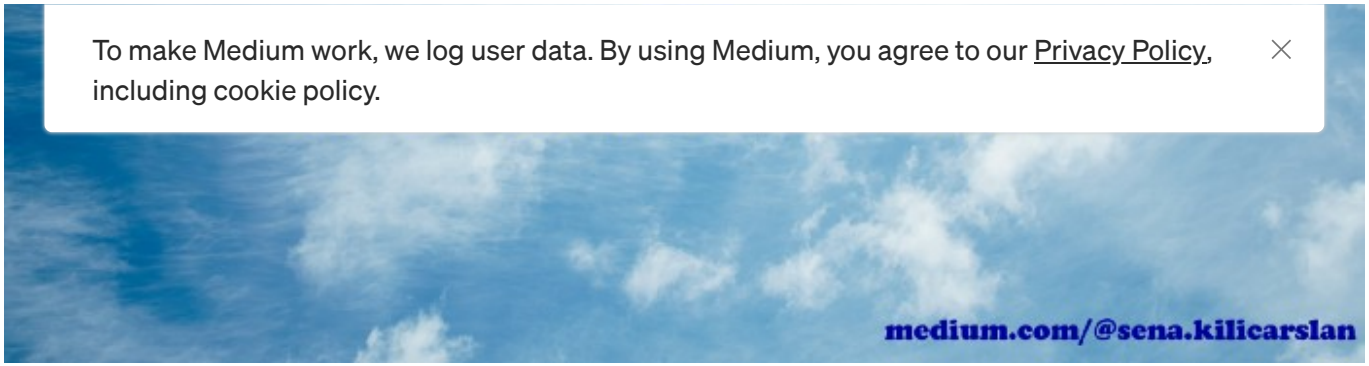
In-memory & Distributed (Redis) Caching in ASP.NET Core

Sena Kılıçarslan Follow
Jan 26, 2020 · 6 min read

In this post, I will demonstrate how to use in-memory caching and Redis based distributed caching in an ASP.NET Core Web API.



I will use the following tools & technologies:

- ASP.NET Core 3.1
- Visual Studio 2019
- Redis

Let's start with the definition of caching.

Caching

A **cache** is a hardware or software component that stores data so that future requests for that data can be served faster; the data stored in a cache might be the result of an earlier computation or a copy of data stored elsewhere. A ***cache hit*** occurs when the requested data can be found in a cache, while a ***cache miss*** occurs when it cannot. Cache hits are served by reading data from the cache, which is faster than recomputing a result or reading from a

sl
ca

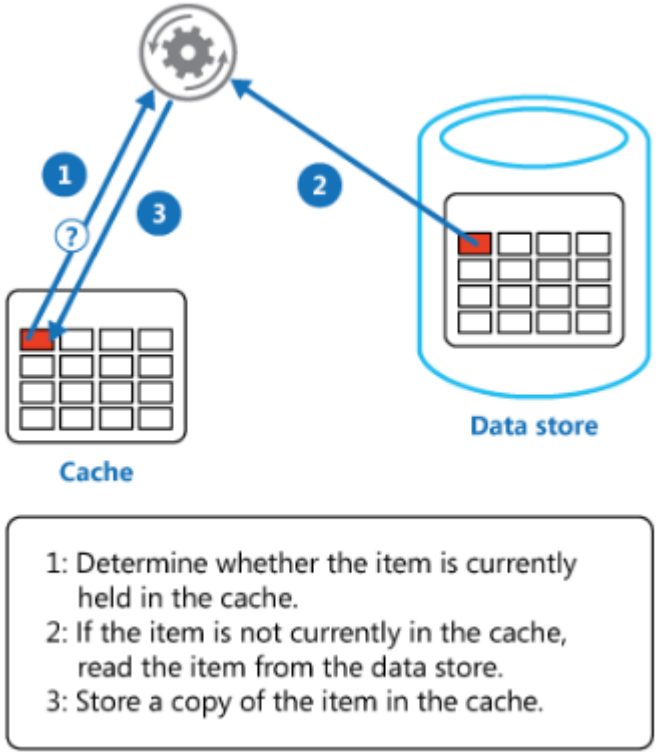


Image source

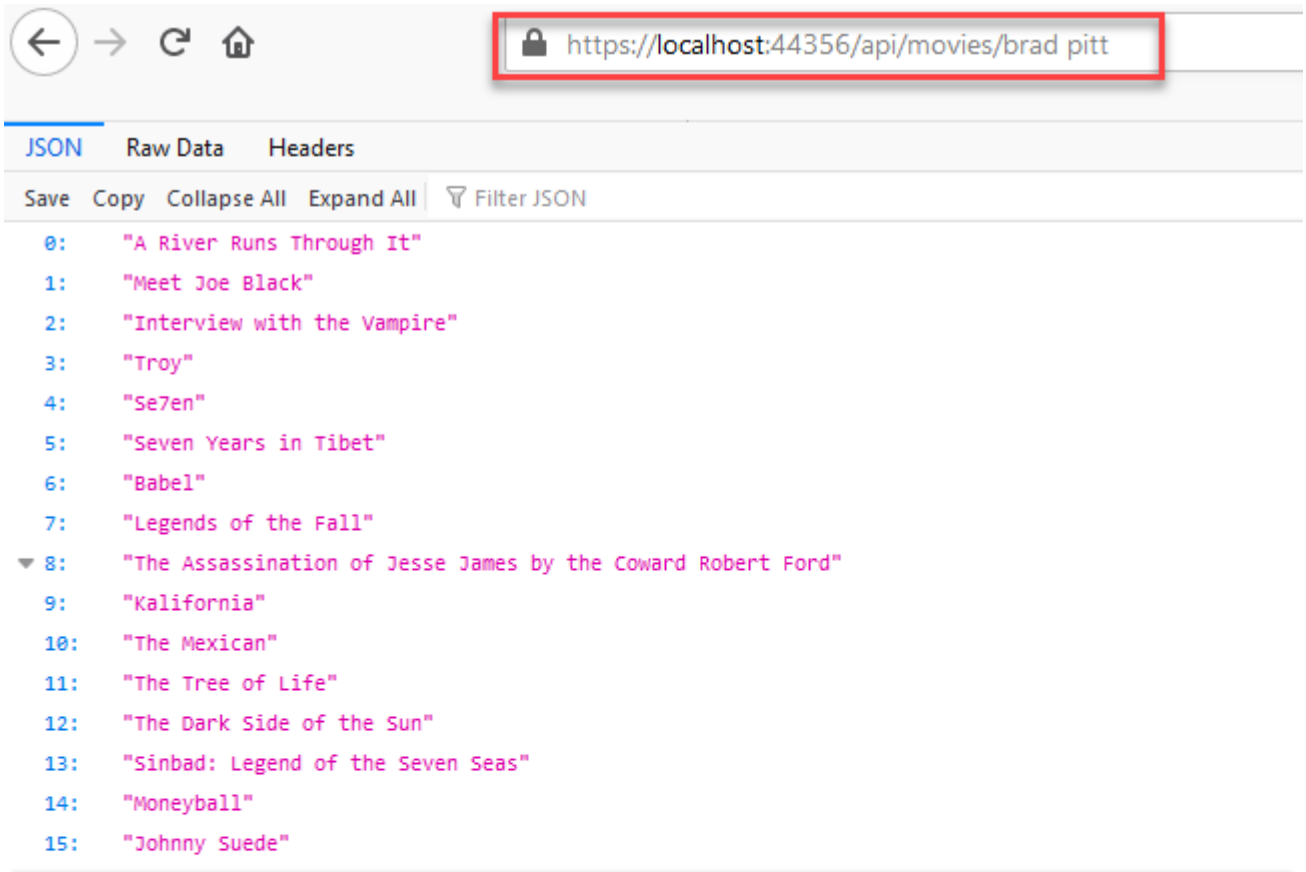
Caching can significantly improve the performance and scalability of an app by reducing the work required to generate content. Caching works best with data that changes infrequently **and** is expensive to generate. Caching makes

a c s
sh

Now, I will present the API in which we will use caching.

About the API

The API returns the movies of an actor/actress given as a parameter and looks like below:



```
19:   "Fury"
20:   "Hitting the Apex"
21:   "The Devil's Own"
22:   "Inglourious Basterds"
23:   "Allied"
24:   "War Machine"
25:   "Ad Astra"
26:   "Ocean's Eleven"
27:   "Ocean's Twelve"
28:   "Ocean's Thirteen"
```

It makes an external API call to the TMDb API and the controller code is shown below:

MoviesController.cs

AspNetCoreCaching | AspNetCoreCaching.Controllers.MoviesController | GetMovieList(string actorName)

```
9   namespace AspNetCoreCaching.Controllers
10  {
11      [Route("api/[controller]")]
12      [ApiController]
        0 references
13      public class MoviesController : ControllerBase
14      {
15          [HttpGet]
            0 references
16          public string Get()
17          {
18              return "Please enter the name of the actor/actress at the end of the URL";
19          }
20
21          [HttpGet("{actorName}")]
            0 references
22          public async Task<List<string>> Get(string actorName)
23          {
24              return await GetMovieList(actorName);
25          }
26
            1 reference
27          private async Task<List<string>> GetMovieList(string actorName)
```

```
29            retur
30        }
31    }
32
}
```


As you see, an external API call is made for every query. However, there will be cases like the same actor/actress is queried several times by the same or different users. So, we can increase the performance of this application by using caching mechanism instead of calling the external API again and again for the same actors/actresses.

First, let's see how we can implement in-memory caching in this API.

In-memory Caching

> 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

client all go to the same server.[3]

In-memory caching is a *service* that's referenced from an app using Dependency Injection. So, we first need to register this service to the built-in IoC container of ASP.NET Core by modifying `ConfigureServices` method of `Startup.cs` as below:

```
Startup.cs  MoviesController.cs
AspNetCoreCaching | AspNetCoreCaching.Startup | ConfigureServices(IServiceCollection services)

2 references
public class Startup
{
    0 references
    public Startup(IConfiguration configuration)...

    1 reference
    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services to the container.
    0 references
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers();

        //added to use in-memory cache
        services.AddMemoryCache();
    }
```

Then we inject `IMemoryCache` to the constructor:

```
Startup.cs  MoviesController.cs
AspNetCoreCaching | AspNetCoreCaching.Controllers.MoviesController | MoviesController(IMemoryCache memoryCache)
```

```
10  namespace AspNetC
11  {
12      [Route("api/[
13      [ApiControlle
        1 reference
14      public class MoviesController : ControllerBase
15      {
16
17          private readonly IMemoryCache memoryCache;
            0 references
18          public MoviesController(IMemoryCache memoryCache)
19          {
20              this.memoryCache = memoryCache;
21          }
```

And we change `GetMovieList` method as below:

Startup.cs | MoviesController.cs

AspNetCoreCaching | AspNetCoreCaching.Controllers.MoviesController | MoviesController(IMemoryCache memoryCache)

```
            1 reference
35          private async Task<List<string>> GetMovieList(string actorName)
36          {
37              var cacheKey = actorName.ToLower();
38
39              if (!memoryCache.TryGetValue(cacheKey, out List<string> movieList))
40              {
41                  movieList = await TmdbApiCall.GetMovieList(actorName);
42
43                  var cacheExpirationOptions =
44                      new MemoryCacheEntryOptions
45                      {
46                          AbsoluteExpiration = DateTime.Now.AddHours(6),
47                          Priority = CacheItemPriority.Normal,
48                          SlidingExpiration = TimeSpan.FromMinutes(5)
49                      };
50                  memoryCache.Set(cacheKey, movieList, cacheExpirationOptions);
51              }
52              return movieList;
53          }
54      }
```

As you see in the above code, we first check if the movie list exists in the cache and return from there if it does. Otherwise, we call the external API

an

Besides, we set the cache expiration options which are explained below:

`SlidingExpiration:` Gets or sets how long a cache entry can be inactive (e.g. not accessed) before it will be removed. This will not extend the entry lifetime beyond the absolute expiration (if set).

`AbsoluteExpiration:` Gets or sets an absolute expiration date for the cache entry.

> *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.* [4]

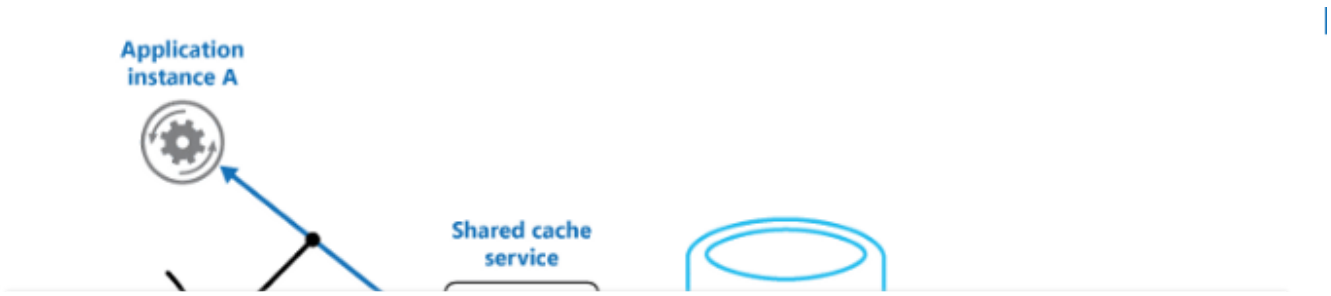
In the next section, we will implement distributed caching for the same controller.

Distributed Caching

> app servers, typically maintained as an external service to the app servers that access it. A distributed cache can improve the performance and scalability of an ASP.NET Core app, especially when the app is hosted by a cloud service or a server farm.

A distributed cache has several advantages over other caching scenarios where cached data is stored on individual app servers. When cached data is distributed, the data:

- Is *coherent* (consistent) across requests to multiple servers.
- Survives server restarts and app deployments.
- Doesn't use local memory.[5]



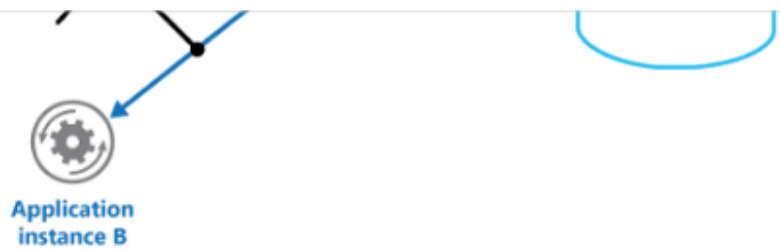


Figure 2: Using a shared cache.

Image source

There are two main disadvantages of the shared caching approach:

- The cache is slower to access because it is no longer held locally to each application instance.
- The requirement to implement a separate cache service might add complexity to the solution.[6]

We will use Redis-based distributed caching in our API. So let's continue with what Redis is and how to install it.

Redis

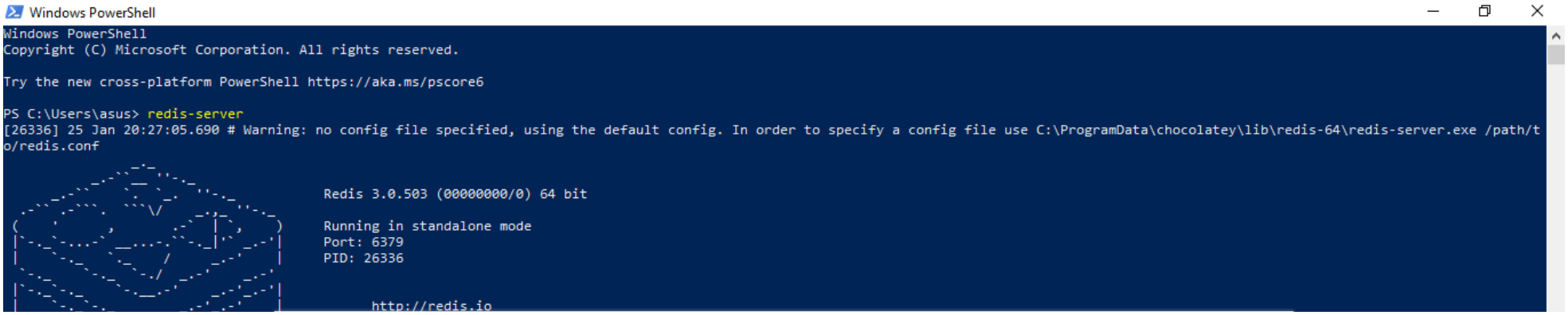
The name Redis means **RE**mote **DI**ctionary **S**erver.

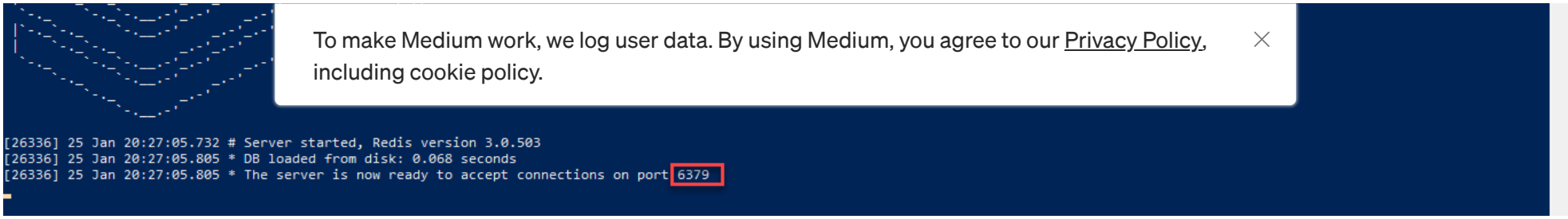
> data structure store, used as a database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs, geospatial indexes with radius queries and streams.[7]

Now, I will show how to install Redis in a Windows machine using Chocolatey. *(If you don't have Chocolatey you can install it from here.)*

First, install the Chocolatey Redis package.

Then run `redis-server` from a command prompt.





Now, the Redis server is up and running. We can test it using the `redis-cli` command.

Open a new command prompt and run `redis-cli` on it and try the following commands:

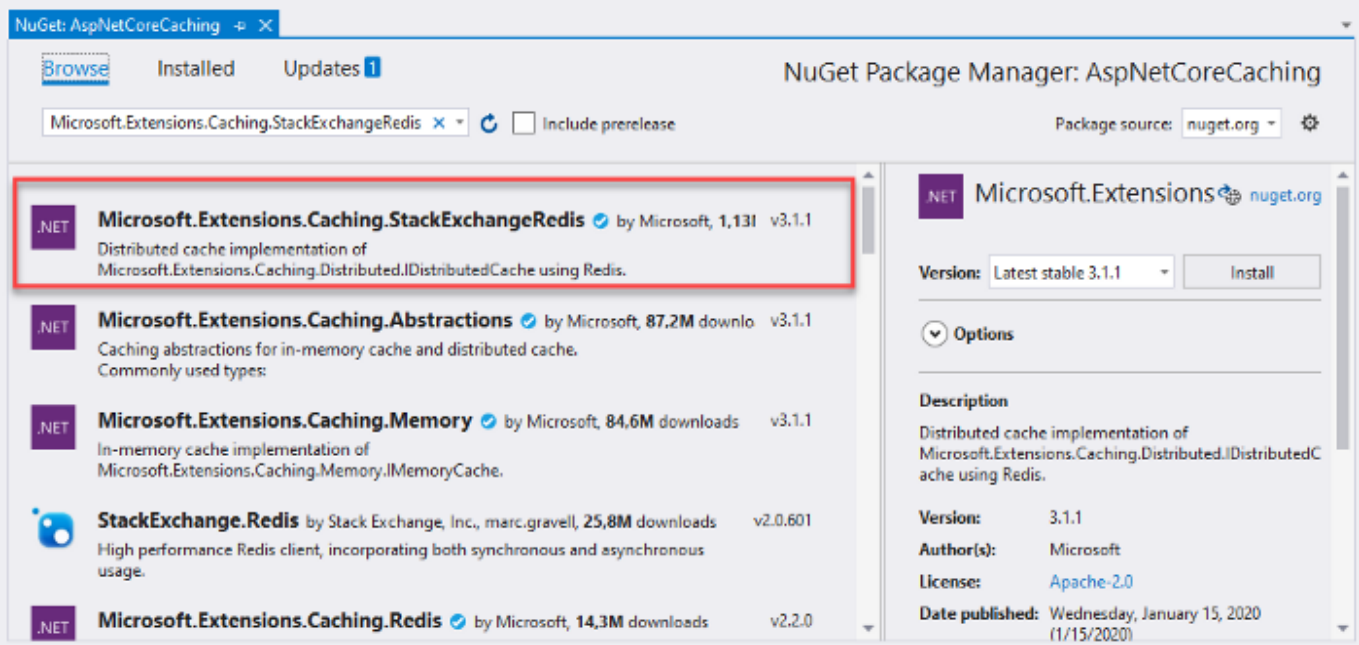
```
PS C:\Users\asus> redis-cli
127.0.0.1:6379> ping
PONG
127.0.0.1:6379>
127.0.0.1:6379> set name "Alice"
OK
127.0.0.1:6379>
127.0.0.1:6379> get name
"Alice"
127.0.0.1:6379>
```

Now that we installed the Redis-server and saw that it is working properly, we can modify the API to use Redis-based distributed caching.

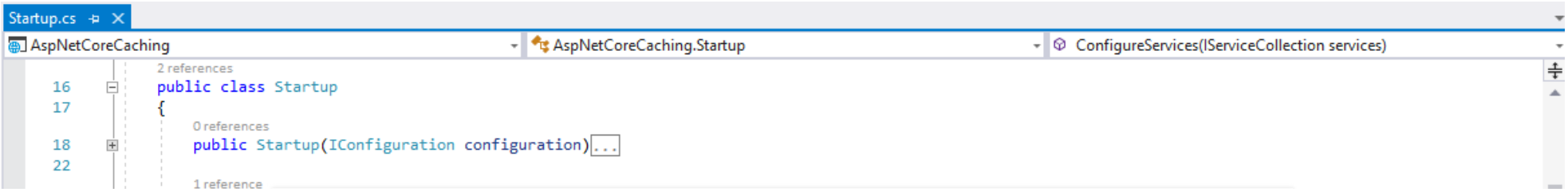
Modify the API

```
Microsoft.Extensions.Caching.Abstractions | Microsoft.Extensions.Caching.Distributed.IDistributedCache | Set(string key, byte[] value, DistributedCacheEntryOptions options)
1   Assembly Microsoft.Extensions.Caching.Abstractions, Version=3.1.1.0, Culture=neutral, PublicKeyToken=adb9793829ddae60
4
5   using ...
8
9   namespace Microsoft.Extensions.Caching.Distributed
10  {
11      public interface IDistributedCache
15      {
16          byte[] Get(string key);
27          Task<byte[]> GetAsync(string key, CancellationToken token = default);
43          void Refresh(string key);
52          Task RefreshAsync(string key, CancellationToken token = default);
68          void Remove(string key);
76          Task RemoveAsync(string key, CancellationToken token = default);
91          void Set(string key, byte[] value, DistributedCacheEntryOptions options);
105         Task SetAsync(string key, byte[] value, DistributedCacheEntryOptions options, CancellationToken token = default);
126     }
127 }
```

- `Get` , `GetAsync` : Accepts a string key and retrieves a cached item as a `byte[]` array if found in the cache.
- `Set` , `SetAsync` : Adds an item (as `byte[]` array) to the cache using a string key.
- `Refresh` , `RefreshAsync` : Refreshes an item in the cache based on its key, resetting its sliding expiration timeout (if any).
- `Remove` , `RemoveAsync` : Removes a cache item based on its string key.



Then we add the following line to `ConfigureServices` method of the `Startup.cs` for configuration the cache implementation using a RedisCache instance:



```
23          public IC
24
25          // This m
            0 references
26          public vo
27          {
28              services.AddControllers();
29
30              //added to use in-memory cache
31              services.AddMemoryCache();
32
33              //added to use Redis cache
34              services.AddStackExchangeRedisCache(options =>
35              {
36                  options.Configuration = "localhost:6379";
37              });
38          }
```

Our approach in modifying the controller will be very similar to what we did in the in-memory caching section.

First, we inject `IDistributedCache` to the controller:

```
MoviesController.cs
AspNetCoreCaching | AspNetCoreCaching.Controllers.MoviesController | distributedCache
11  namespace AspNetCoreCaching.Controllers
12  {
13      [Route("api/[controller]")]
14      [ApiController]
        1 reference
15      public class MoviesController : ControllerBase
16      {
17
18          private readonly IDistributedCache distributedCache;
            0 references
19          public MoviesController(IDistributedCache distributedCache)
20          {
21              this.distributedCache = distributedCache;
22          }
```

Th

MoviesController.cs

AspNetCoreCaching | AspNetCoreCaching.Controllers.MoviesController | GetMovieList(string actorName)

```
1 reference
private async Task<List<string>> GetMovieList(string actorName)
{
    var cacheKey = actorName.ToLower();

    List<string> moviesList;
    string serializedMovies;

    var encodedMovies = await distributedCache.GetAsync(cacheKey);

    if (encodedMovies != null)
    {
        serializedMovies = Encoding.UTF8.GetString(encodedMovies);
        moviesList = JsonConvert.DeserializeObject<List<string>>(serializedMovies);
    }
    else
    {
        moviesList = await TmdbApiCall.GetMovieList(actorName);
        serializedMovies = JsonConvert.SerializeObject(moviesList);
        encodedMovies = Encoding.UTF8.GetBytes(serializedMovies);
        var options = new DistributedCacheEntryOptions()
                        .SetSlidingExpiration(TimeSpan.FromMinutes(5))
                        .SetAbsoluteExpiration(DateTime.Now.AddHours(6));
        await distributedCache.SetAsync(cacheKey, encodedMovies, options);
    }
    return moviesList;
}
```

As mentioned above, `GetAsync` and `SetASync` methods work with `byte[]` arrays, so I encoded the serialized movies list to `byte[]` array. Also, I want to mention that there are extension methods where you do not need to convert the value to `byte[]` array and these are `GetStringAsync` and `SetStringAsync`.

Yo

That's the end of the post. I hope you found this post helpful and easy to follow. If you have any questions and/or comments, you can share them in the responses section below.

And if you liked this post, please clap your hands 👏👏👏

Bye!

Aspnetcore Caching Redis Rest Api