# Distributed caching in ASP.NET Core

02/07/2020 • 20 minutes to read •      +8

**In this article**



By Mohsin Nasir and Steve Smith

A distributed cache is a cache shared by multiple 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.

Distributed cache configuration is implementation specific. This article describes how to configure SQL Server and Redis distributed caches. Third party implementations are also available, such as NCache (NCache on GitHub). Regardless of which implementation is selected, the app interacts with the cache using the IDistributedCache interface.

View or download sample code (how to download)

## Prerequisites

To use a SQL Server distributed cache, add a package reference to the Microsoft.Extensions.Caching.SqlServer package.

To use a Redis distributed cache, add a package reference to the Microsoft.Extensions.Caching.StackExchangeRedis package.

To use NCache distributed cache, add a package reference to the NCache.Microsoft.Extensions.Caching.OpenSource package.

# IDistributedCache interface

The IDistributedCache interface provides the following methods to manipulate items in the distributed cache implementation:

- 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.

# Establish distributed caching services

Register an implementation of IDistributedCache in `Startup.ConfigureServices`. Framework-provided implementations described in this topic include:

- Distributed Memory Cache
- Distributed SQL Server cache
- Distributed Redis cache
- Distributed NCache cache

## Distributed Memory Cache

The Distributed Memory Cache (AddDistributedMemoryCache) is a framework-provided implementation of IDistributedCache that stores items in memory. The Distributed Memory Cache isn't an actual distributed cache. Cached items are stored by the app instance on the server where the app is running.

The Distributed Memory Cache is a useful implementation:

- In development and testing scenarios.

- When a single server is used in production and memory consumption isn't an issue. Implementing the Distributed Memory Cache abstracts cached data storage. It allows for implementing a true distributed caching solution in the future if multiple nodes or fault tolerance become necessary.

The sample app makes use of the Distributed Memory Cache when the app is run in the Development environment in `Startup.ConfigureServices`:

C# Copy

```
services.AddDistributedMemoryCache();
```

## Distributed SQL Server Cache

The Distributed SQL Server Cache implementation (AddDistributedSqlServerCache) allows the distributed cache to use a SQL Server database as its backing store. To create a SQL Server cached item table in a SQL Server instance, you can use the `sql-cache` tool. The tool creates a table with the name and schema that you specify.

Create a table in SQL Server by running the `sql-cache create` command. Provide the SQL Server instance (`Data Source`), database (`Initial Catalog`), schema (for example, `dbo`), and table name (for example, `TestCache`):

.NET Core CLI Copy

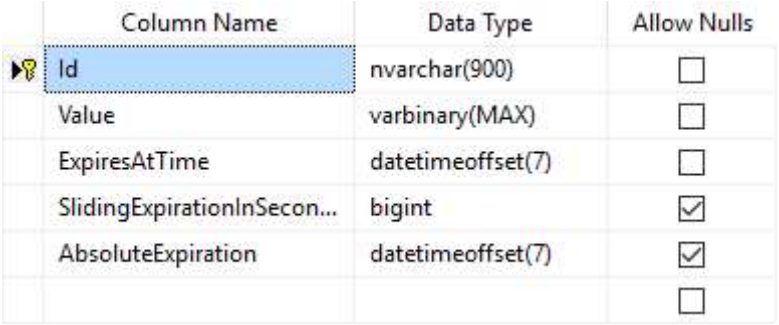
```
dotnet sql-cache create "Data Source=(localdb)\MSSQLLocalDB;Initial
Catalog=DistCache;Integrated Security=True;" dbo TestCache
```

A message is logged to indicate that the tool was successful:

Console Copy

```
Table and index were created successfully.
```

The table created by the `sql-cache` tool has the following schema:



| Column Name | Data Type | Allow Nulls |
|---|---|---|
| Id | nvarchar(900) | ☐ |
| Value | varbinary(MAX) | ☐ |
| ExpiresAtTime | datetimeoffset(7) | ☐ |
| SlidingExpirationInSecon... | bigint | ☑ |
| AbsoluteExpiration | datetimeoffset(7) | ☑ |
| | | ☐ |

> **Note**
>
> An app should manipulate cache values using an instance of **IDistributedCache**, not a **SqlServerCache**.

The sample app implements SqlServerCache in a non-Development environment in `Startup.ConfigureServices`:

C# Copy

```
services.AddDistributedSqlServerCache(options =>
{
    options.ConnectionString =
        _config["DistCache_ConnectionString"];
    options.SchemaName = "dbo";
    options.TableName = "TestCache";
});
```

> **Note**
>
> A **ConnectionString** (and optionally, **SchemaName** and **TableName**) are typically stored outside of source control (for example, stored by the **Secret Manager** or in *appsettings.json/appsettings.{ENVIRONMENT}.json* files). The connection string may contain credentials that should be kept out of source control systems.

## Distributed Redis Cache

Redis is an open source in-memory data store, which is often used as a distributed cache. You can configure an Azure Redis Cache for an Azure-hosted ASP.NET Core app, and use an Azure Redis Cache for local development.

An app configures the cache implementation using a RedisCache instance (AddStackExchangeRedisCache).

For more information, see Azure Cache for Redis.

See this GitHub issue for a discussion on alternative approaches to a local Redis cache.

## Distributed NCache Cache

NCache is an open source in-memory distributed cache developed natively in .NET and .NET Core. NCache works both locally and configured as a distributed cache cluster for an ASP.NET Core app running in Azure or on other hosting platforms.

To install and configure NCache on your local machine, see Getting Started Guide for Windows (.NET and .NET Core) .

To configure NCache:

1. Install NCache open source NuGet .
2. Configure the cache cluster in client.ncconf .
3. Add the following code to `Startup.ConfigureServices`:

C# Copy

```
services.AddNCacheDistributedCache(configuration =>
{
    configuration.CacheName = "demoClusteredCache";
    configuration.EnableLogs = true;
    configuration.ExceptionsEnabled = true;
});
```

# Use the distributed cache

To use the IDistributedCache interface, request an instance of IDistributedCache from any constructor in the app. The instance is provided by dependency injection (DI).

When the sample app starts, IDistributedCache is injected into `Startup.Configure`. The current time is cached using IHostApplicationLifetime (for more information, see Generic Host: IHostApplicationLifetime):

C# Copy

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
    IHostApplicationLifetime lifetime, IDistributedCache cache)
{
    lifetime.ApplicationStarted.Register(() =>
    {
        var currentTimeUTC = DateTime.UtcNow.ToString();
        byte[] encodedCurrentTimeUTC = Encoding.UTF8.GetBytes(currentTimeUTC);
        var options = new DistributedCacheEntryOptions()
            .SetSlidingExpiration(TimeSpan.FromSeconds(20));
        cache.Set("cachedTimeUTC", encodedCurrentTimeUTC, options);
    });
```

The sample app injects IDistributedCache into the `IndexModel` for use by the Index page.

Each time the Index page is loaded, the cache is checked for the cached time in `OnGetAsync`. If the cached time hasn't expired, the time is displayed. If 20 seconds have elapsed since the last time the cached time was accessed (the last time this page was loaded), the page displays *Cached Time Expired*.

Immediately update the cached time to the current time by selecting the **Reset Cached Time** button. The button triggers the `OnPostResetCachedTime` handler method.

C# Copy

```
public class IndexModel : PageModel
{
    private readonly IDistributedCache _cache;

    public IndexModel(IDistributedCache cache)
    {
        _cache = cache;
    }

    public string CachedTimeUTC { get; set; }

    public async Task OnGetAsync()
    {
        CachedTimeUTC = "Cached Time Expired";
        var encodedCachedTimeUTC = await _cache.GetAsync("cachedTimeUTC");

        if (encodedCachedTimeUTC != null)
        {
            CachedTimeUTC = Encoding.UTF8.GetString(encodedCachedTimeUTC);
        }
    }

    public async Task<IActionResult> OnPostResetCachedTime()
```

```
    {
        var currentTimeUTC = DateTime.UtcNow.ToString();
        byte[] encodedCurrentTimeUTC = Encoding.UTF8.GetBytes(currentTimeUTC);
        var options = new DistributedCacheEntryOptions()
            .SetSlidingExpiration(TimeSpan.FromSeconds(20));
        await _cache.SetAsync("cachedTimeUTC", encodedCurrentTimeUTC, options);

        return RedirectToPage();
    }
}
```

> **Note**
>
> There's no need to use a Singleton or Scoped lifetime for **IDistributedCache** instances (at least for the built-in implementations).
>
> You can also create an **IDistributedCache** instance wherever you might need one instead of using DI, but creating an instance in code can make your code harder to test and violates the **Explicit Dependencies Principle**.

# Recommendations

When deciding which implementation of IDistributedCache is best for your app, consider the following:

- Existing infrastructure
- Performance requirements
- Cost
- Team experience

Caching solutions usually rely on in-memory storage to provide fast retrieval of cached data, but memory is a limited resource and costly to expand. Only store commonly used data in a cache.

Generally, a Redis cache provides higher throughput and lower latency than a SQL Server cache. However, benchmarking is usually required to determine the performance characteristics of caching strategies.

When SQL Server is used as a distributed cache backing store, use of the same database for the cache and the app's ordinary data storage and retrieval can negatively impact the performance of both. We recommend using a dedicated SQL Server instance for the distributed cache backing store.

# Additional resources

- Redis Cache on Azure
- SQL Database on Azure
- ASP.NET Core IDistributedCache Provider for NCache in Web Farms (NCache on GitHub )
- Cache in-memory in ASP.NET Core
- Detect changes with change tokens in ASP.NET Core
- Response caching in ASP.NET Core
- Response Caching Middleware in ASP.NET Core
- Cache Tag Helper in ASP.NET Core MVC
- Distributed Cache Tag Helper in ASP.NET Core
- Host ASP.NET Core in a web farm

## Is this page helpful?

Yes No