August 25, 2016 in ASP.NET CORE SESSION STATE ~ 5 min read.

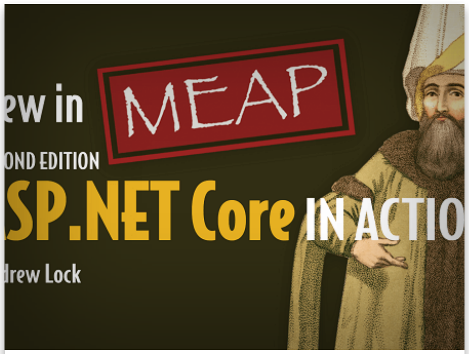
An introduction to Session storage in ASP.NET Core

Share on:

A common requirement of web applications is the need to store temporary state data. In this article I discuss the use of Session storage for storing data related to a particular user or browser session.

Options for storing application state

When building ASP.NET Core applications, there are a number of options available to you when you need to store data that is specific to a particular request or session.



One of the simplest methods is to use **querystring parameters** or **post data** to send state to subsequent requests. However doing so requires sending that data to the user's browser, which may not be desirable, especially for sensitive data. For that reason, extra care must be taken when using this approach.

Cookies can also be used to store small bits of data, though again, these make a roundtrip to the user's browser, so must be kept small, and if sensitive, must be secured.

For each request there exists a property `Items` on `HttpContext`. This is an `IDictionary<string, object>` which can be used to store arbitrary objects against a string key. The data stored here lasts for just a single request, so can be useful for communicating between middleware components and storing state related to just a single request.

Files and **database storage** can obviously be used to store state data, whether related to a particular user or the application in general. However they are typically slower to store and retrieve data than other available options.

Session state relies on a cookie identifier to identify a particular browser session, and stores data related to the session on the server. This article focuses on how and when to use Session in your ASP.NET Core application.

Session in ASP.NET Core

ASP.NET Core supports the concept of a Session out of the box - the `HttpContext` object contains a `Session` property of type `ISession`. The get and set portion of the interface is shown below (see the full interface here):

```
public interface ISession
{
    bool TryGetValue(string key, out byte[] value);
    void Set(string key, byte[] value);
    void Remove(string key);
}
```

As you can see, it provides a dictionary-like wrapper over the `byte[]` data, accessing state via `string` keys. Generally speaking, each user will have an individual session, so you can store data related to a single user in it. However you cannot technically consider the data secure as it may be possible to hijack another user's session, so it is not advisable to store user secrets in it. As the documentation states:

> *You can't necessarily assume that a session is restricted to a single user, so be careful what kind of information you store in Session.*

Another point to consider is that the session in ASP.NET Core is non-locking, so if multiple requests modify the session, the last action will win. This is an important point to consider, but should provide a significant performance increase over the locking session management used in the previous ASP.NET 4.X framework.

Under the hood, Session is built on top of `IDistributedCache`, which can be used as a more generalised cache in your application. ASP.NET Core ships with a number of `IDistributedCache` implementations, the simplest of which is an in-memory implementation, `MemoryCache`, which can be found in the *Microsoft.Extensions.Caching.Memory* package.

MVC also exposes a `TempData` property on a `Controller` which is an additional wrapper around Session. This can be used for storing transient data that only needs to be available for a single request after the current one.

Configuring your application to use Session

In order to be able to use Session storage in your application, you must configure

In order to be able to use Session storage in your application, you must configure the required Session services, the Session middleware, and an `IDistributedCache` implementation. In this example I will be using the in-memory distributed cache as it is simple to setup and use, but the documentation states that this should only be used for development and testing sites. I suspect this reticence is due it not actually being distributed and the fact that app restarts will clear the session.

First, add the `IDistributedCache` implementation and Session state packages to your *project.json*:

```
dependencies: {
  "Microsoft.Extensions.Caching.Memory" : "1.0.0",
  "Microsoft.AspNetCore.Session": "1.0.0"
}
```

Next, add the required services to `Startup` in `ConfigureServices`:

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

    services.AddDistributedMemoryCache();
    services.AddSession();
}
```

Finally, configure the session middleware in the `Startup.Configure` method. As with all middleware, order is important in this method, so you will need to enable the session before you try and access it, e.g. in your MVC middleware:

```
public void Configure(IApplicationBuilder app)
{
    app.UseStaticFiles();


    //enable session before MVC
    app.UseSession();

    app.UseMvc(routes =>
```

```
        {
            routes.MapRoute(
                name: "default",
                template: "{controller=Home}/{action=Index}/{id?}");
        });
    }
```

With all this in place, the Session object can be used to store our data.

Storing data in Session

As shown previously, objects must be stored in Session as a `byte[]`, which is obviously not overly convenient. To alleviate the need to work directly with byte arrays, a number of extensions exist for fetching and setting `int` and `string`. Storing more complex objects requires serialising the data.

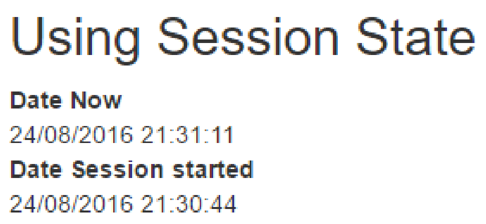
As an example, consider the simple usage of session below.

```
public IActionResult Index()
{
    const string sessionKey = "FirstSeen";
    DateTime dateFirstSeen;
    var value = HttpContext.Session.GetString(sessionKey);
    if (string.IsNullOrEmpty(value))
    {
        dateFirstSeen = DateTime.Now;
        var serialisedDate = JsonConvert.SerializeObject(dateFirstSeen);
        HttpContext.Session.SetString(sessionKey, serialisedDate);
    }
    else
    {
        dateFirstSeen = JsonConvert.DeserializeObject<DateTime>(value);
    }

    var model = new SessionStateViewModel
    {
        DateSessionStarted = dateFirstSeen,
        Now = DateTime.Now
    };

    return View(model);
    return View(model);
}
```

This action simply simply returns a view with a model that shows the current time, and the time the session was initialised.

First, the Session is queried using `GetString(key)`. If this is the first time that action has been called, the method will return null. In that case, we record the current date, serialise it to a string using *Newtonsoft.Json*, and store it in the session using `SetString(key, value)`.

On subsequent requests, the call to `GetString(key)` will return our serialised `DateTime` which we can set on our view model for display. After the first request to our action, the `DateSessionStarted` property will differ from the `Now` property on our model:


Using Session State

Date Now
24/08/2016 21:31:11
Date Session started
24/08/2016 21:30:44

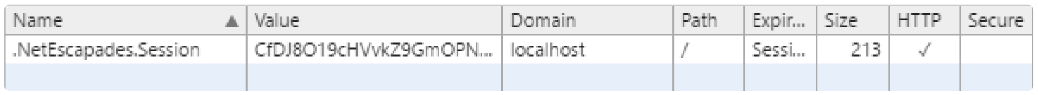
This was a very trivial example, but you can store any data that is serialisable to a `byte[]` in the Session. The JSON serialisation used here is an easy option as it is likely already used in your project. Obviously, serialising and deserialising large objects on every request could be a performance concern, so be sure to think about the implications of using Session storage in your application.

Customising Session configuration

When configuring your session in `Startup`, you can provide an instance of `StartupOptions` or a configuration lambda to either the `UseSession` or `AddSession` calls respectively. This allows you to customise details about the session cookie that is used to track the session in the browser. For example you can customise the cookie name, domain, path and how long the session may be idle before the session expires. You will likely not need to change the defaults, but it may be necessary in some cases:

```
services.AddSession(opts =>
    {
        opts.CookieName = ".NetEscapades.Session";
        opts.IdleTimeout = TimeSpan.FromMinutes(5);
    });
```

Note the cookie name is not the default `.AspNetCore.Session`:



Name ▲	Value	Domain	Path	Expir...	Size	HTTP	Secure
.NetEscapades.Session	CfDJ8O19cHVvkZ9GmOPN...	localhost	/	Sessi...	213	✓	

It's also worth noting that in ASP.NET Core 1.0, you cannot currently mark the cookie as Secure. This has been fixed here so should be in the 1.1.0 release (probably Q4 206/ Q1 2017).

Summary

In this post we saw an introduction to using Session storage in an ASP.NET Core application. We saw how to configure the required services and middleware, and to use it to store and retrieve simple strings to share state across requests.

As mentioned previously, it's important to not store sensitive user details in Session due to potential security issues, but otherwise it is a useful location for storage of serialisable data.

Further Reading

- https://docs.asp.net/en/latest/fundamentals/app-state.html

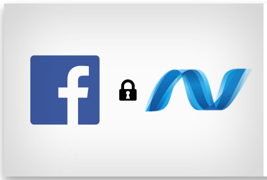
- https://docs.asp.net/en/latest/fundamentals/app-state.html
- https://docs.asp.net/en/latest/performance/caching/distributed.html
- https://github.com/aspnet/Caching
- https://github.com/aspnet/Session

FOLLOW ME

ENJOY THIS BLOG?

Buy me a coffee

PREVIOUS
A look behind the JWT bearer authentication middleware in ASP.NET Core



NEXT
An introduction to OAuth 2.0 using Facebook in ASP.NET Core

23 Comments Andrew Lock | .Net Escapades Disqus' Privacy Policy

1 Login

Recommend Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Jay • 3 years ago

IdleTimeout is currently not working for me on .NET Core 2. Got it set to an hour, but the session still expires in minutes?

22 | • Reply • Share ›

IMAGE CREDITS

“Big Data” by DARPA licesnsed under Public Domain work