Session and state management in ASP.NET Core

Article • 02/14/2023

By Rick Anderson , Kirk Larkin , and Diana LaRose

HTTP is a stateless protocol. By default, HTTP requests are independent messages that don't retain user values. This article describes several approaches to preserve user data between requests.

State management

State can be stored using several approaches. Each approach is described later in this article.

Storage approach	Storage mechanism
Cookies	HTTP cookies. May include data stored using server-side app code.
Session state	HTTP cookies and server-side app code
TempData	HTTP cookies or session state
Query strings	HTTP query strings
Hidden fields	HTTP form fields
HttpContext.Items	Server-side app code
Cache	Server-side app code

SignalR/Blazor Server and HTTP context-based state management

SignalR apps shouldn't use session state and other state management approaches that rely upon a stable HTTP context to store information. SignalR apps can store per-connection state in Context.Items in the hub. For more information and alternative state management approaches for Blazor Server apps, see ASP.NET Core Blazor state management.

Cookies

Cookies store data across requests. Because cookies are sent with every request, their size should be kept to a minimum. Ideally, only an identifier should be stored in a cookie with the data stored by the app. Most browsers restrict cookie size to 4096 bytes. Only a limited number of cookies are available for each domain.

Because cookies are subject to tampering, they must be validated by the app. Cookies can be deleted by users and expire on clients. However, cookies are generally the most durable form of data persistence on the client.

Cookies are often used for personalization, where content is customized for a known user. The user is only identified and not authenticated in most cases. The cookie can store the user's name, account name, or unique user ID such as a GUID. The cookie can be used to access the user's personalized settings, such as their preferred website background color.

See the European Union General Data Protection Regulations (GDPR) when issuing cookies and dealing with privacy concerns. For more information, see General Data Protection Regulation (GDPR) support in ASP.NET Core.

Session state

Session state is an ASP.NET Core scenario for storage of user data while the user browses a web app. Session state uses a store maintained by the app to persist data across requests from a client. The session data is backed by a cache and considered ephemeral data. The site should continue to function without the session data. Critical application data should be stored in the user database and cached in session only as a performance optimization.

Session isn't supported in SignalR apps because a SignalR Hub may execute independent of an HTTP context. For example, this can occur when a long polling request is held open by a hub beyond the lifetime of the request's HTTP context.

ASP.NET Core maintains session state by providing a cookie to the client that contains a session ID. The cookie session ID:

- Is sent to the app with each request.
- Is used by the app to fetch the session data.

Session state exhibits the following behaviors:

- The session cookie is specific to the browser. Sessions aren't shared across browsers.
- Session cookies are deleted when the browser session ends.

- If a cookie is received for an expired session, a new session is created that uses the same session cookie.
- Empty sessions aren't retained. The session must have at least one value set to persist the session across requests. When a session isn't retained, a new session ID is generated for each new request.
- The app retains a session for a limited time after the last request. The app either sets the session timeout or uses the default value of 20 minutes. Session state is ideal for storing user data:
  - That's specific to a particular session.
  - Where the data doesn't require permanent storage across sessions.
- Session data is deleted either when the ISession.Clear implementation is called or when the session expires.
- There's no default mechanism to inform app code that a client browser has been closed or when the session cookie is deleted or expired on the client.
- Session state cookies aren't marked essential by default. Session state isn't functional unless tracking is permitted by the site visitor. For more information, see General Data Protection Regulation (GDPR) support in ASP.NET Core.
- **Note**: There is no replacement for the cookieless session feature from the ASP.NET Framework because it's considered insecure and can lead to session fixation attacks.

> ⚠ **Warning**
>
> Don't store sensitive data in session state. The user might not close the browser and clear the session cookie. Some browsers maintain valid session cookies across browser windows. A session might not be restricted to a single user. The next user might continue to browse the app with the same session cookie.

The in-memory cache provider stores session data in the memory of the server where the app resides. In a server farm scenario:

- Use *sticky sessions* to tie each session to a specific app instance on an individual server. Azure App Service uses Application Request Routing (ARR) to enforce sticky sessions by default. However, sticky sessions can affect scalability and complicate web app updates. A better approach is to use a Redis or SQL Server distributed cache, which doesn't require sticky sessions. For more information, see Distributed caching in ASP.NET Core.
- The session cookie is encrypted via IDataProtector. Data Protection must be properly configured to read session cookies on each machine. For more information, see ASP.NET Core Data Protection Overview and Key storage providers.

Configure session state

The Microsoft.AspNetCore.Session package:

- Is included implicitly by the framework.
- Provides middleware for managing session state.

To enable the session middleware, `Program.cs` must contain:

- Any of the IDistributedCache memory caches. The `IDistributedCache` implementation is used as a backing store for session. For more information, see Distributed caching in ASP.NET Core.
- A call to AddSession
- A call to UseSession

The following code shows how to set up the in-memory session provider with a default in-memory implementation of `IDistributedCache`:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddDistributedMemoryCache();

builder.Services.AddSession(options =>
{
    options.IdleTimeout = TimeSpan.FromSeconds(10);
    options.Cookie.HttpOnly = true;
    options.Cookie.IsEssential = true;
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.UseSession();
```

```
app.MapRazorPages();
app.MapDefaultControllerRoute();

app.Run();
```

The preceding code sets a short timeout to simplify testing.

The order of middleware is important. Call `UseSession` after `UseRouting` and before `MapRazorPages` and `MapDefaultControllerRoute`. See Middleware Ordering.

HttpContext.Session is available after session state is configured.

`HttpContext.Session` can't be accessed before `UseSession` has been called.

A new session with a new session cookie can't be created after the app has begun writing to the response stream. The exception is recorded in the web server log and not displayed in the browser.

Load session state asynchronously

The default session provider in ASP.NET Core loads session records from the underlying IDistributedCache backing store asynchronously only if the ISession.LoadAsync method is explicitly called before the TryGetValue, Set, or Remove methods. If `LoadAsync` isn't called first, the underlying session record is loaded synchronously, which can incur a performance penalty at scale.

To have apps enforce this pattern, wrap the DistributedSessionStore and DistributedSession implementations with versions that throw an exception if the `LoadAsync` method isn't called before `TryGetValue`, `Set`, or `Remove`. Register the wrapped versions in the services container.

Session options

To override session defaults, use SessionOptions.

Option	Description
Cookie	Determines the settings used to create the cookie. Name defaults to SessionDefaults.CookieName (`.AspNetCore.Session`). Path defaults to SessionDefaults.CookiePath (`/`). SameSite defaults to SameSiteMode.Lax (`1`). HttpOnly defaults to `true`. IsEssential defaults to `false`.

Option	Description
IdleTimeout	The `IdleTimeout` indicates how long the session can be idle before its contents are abandoned. Each session access resets the timeout. This setting only applies to the content of the session, not the cookie. The default is 20 minutes.
IOTimeout	The maximum amount of time allowed to load a session from the store or to commit it back to the store. This setting may only apply to asynchronous operations. This timeout can be disabled using InfiniteTimeSpan. The default is 1 minute.

Session uses a cookie to track and identify requests from a single browser. By default, this cookie is named `.AspNetCore.Session`, and it uses a path of `/`. Because the cookie default doesn't specify a domain, it isn't made available to the client-side script on the page (because HttpOnly defaults to `true`).

To override cookie session defaults, use SessionOptions:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages();
builder.Services.AddControllersWithViews();

builder.Services.AddDistributedMemoryCache();

builder.Services.AddSession(options =>
{
    options.Cookie.Name = ".AdventureWorks.Session";
    options.IdleTimeout = TimeSpan.FromSeconds(10);
    options.Cookie.IsEssential = true;
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.UseSession();
```

```
app.MapRazorPages();
app.MapDefaultControllerRoute();

app.Run();
```

The app uses the IdleTimeout property to determine how long a session can be idle before its contents in the server's cache are abandoned. This property is independent of the cookie expiration. Each request that passes through the Session Middleware resets the timeout.

Session state is *non-locking*. If two requests simultaneously attempt to modify the contents of a session, the last request overrides the first. `Session` is implemented as a *coherent session*, which means that all the contents are stored together. When two requests seek to modify different session values, the last request may override session changes made by the first.

Set and get Session values

Session state is accessed from a Razor Pages PageModel class or MVC Controller class with HttpContext.Session. This property is an ISession implementation.

The `ISession` implementation provides several extension methods to set and retrieve integer and string values. The extension methods are in the Microsoft.AspNetCore.Http namespace.

`ISession` extension methods:

- Get(ISession, String)
- GetInt32(ISession, String)
- GetString(ISession, String)
- SetInt32(ISession, String, Int32)
- SetString(ISession, String, String)

The following example retrieves the session value for the `IndexModel.SessionKeyName` key (`_Name` in the sample app) in a Razor Pages page:

C#

```
@page
@using Microsoft.AspNetCore.Http
@model IndexModel

...
```

```
Name: @HttpContext.Session.GetString(IndexModel.SessionKeyName)
```

The following example shows how to set and get an integer and a string:

C#

```
public class IndexModel : PageModel
{
    public const string SessionKeyName = "_Name";
    public const string SessionKeyAge = "_Age";

    private readonly ILogger<IndexModel> _logger;

    public IndexModel(ILogger<IndexModel> logger)
    {
        _logger = logger;
    }

    public void OnGet()
    {
        if
(string.IsNullOrEmpty(HttpContext.Session.GetString(SessionKeyName)))
        {
            HttpContext.Session.SetString(SessionKeyName, "The
Doctor");
            HttpContext.Session.SetInt32(SessionKeyAge, 73);
        }
        var name = HttpContext.Session.GetString(SessionKeyName);
        var age =
HttpContext.Session.GetInt32(SessionKeyAge).ToString();

        _logger.LogInformation("Session Name: {Name}", name);
        _logger.LogInformation("Session Age: {Age}", age);
    }
}
```

The following markup displays the session values on a Razor Page:

CSHTML

```
@page
@model PrivacyModel
@{
    ViewData["Title"] = "Privacy Policy";
}
<h1>@ViewData["Title"]</h1>

<div class="text-center">
<p><b>Name:</b> @HttpContext.Session.GetString("_Name");<b>Age:
```

```
</b> @HttpContext.Session.GetInt32("_Age").ToString()</p>
</div>
```

All session data must be serialized to enable a distributed cache scenario, even when using the in-memory cache. String and integer serializers are provided by the extension methods of ISession. Complex types must be serialized by the user using another mechanism, such as JSON.

Use the following sample code to serialize objects:

C#

```
public static class SessionExtensions
{
    public static void Set<T>(this ISession session, string key, T
value)
    {
        session.SetString(key, JsonSerializer.Serialize(value));
    }

    public static T? Get<T>(this ISession session, string key)
    {
        var value = session.GetString(key);
        return value == null ? default :
JsonSerializer.Deserialize<T>(value);
    }
}
```

The following example shows how to set and get a serializable object with the `SessionExtensions` class:

C#

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using Web.Extensions;    // SessionExtensions

namespace SessionSample.Pages
{
    public class Index6Model : PageModel
    {
        const string SessionKeyTime = "_Time";
        public string? SessionInfo_SessionTime { get; private set; }
        private readonly ILogger<Index6Model> _logger;

        public Index6Model(ILogger<Index6Model> logger)
        {
            _logger = logger;
        }
```

```
        public void OnGet()
        {
            var currentTime = DateTime.Now;

            // Requires SessionExtensions from sample.
            if (HttpContext.Session.Get<DateTime>(SessionKeyTime) ==
default)
            {
                HttpContext.Session.Set<DateTime>(SessionKeyTime,
currentTime);
            }
            _logger.LogInformation("Current Time: {Time}", current-
Time);
            _logger.LogInformation("Session Time: {Time}",
                                   HttpContext.Session.Get<DateTime>
(SessionKeyTime));

        }
    }
}
```

> ⚠ **Warning**
>
> Storing a live object in the session should be used with caution, as there are many problems that can occur with serialized objects. For more information, see **Sessions should be allowed to store objects (dotnet/aspnetcore #18159)** .

TempData

ASP.NET Core exposes the Razor Pages TempData or Controller TempData. This property stores data until it's read in another request. The Keep(String) and Peek(string) methods can be used to examine the data without deletion at the end of the request. Keep marks all items in the dictionary for retention. `TempData` is:

- Useful for redirection when data is required for more than a single request.
- Implemented by `TempData` providers using either cookies or session state.

TempData samples

Consider the following page that creates a customer:

C#

```
public class CreateModel : PageModel
{
    private readonly RazorPagesContactsContext _context;

    public CreateModel(RazorPagesContactsContext context)
    {
        _context = context;
    }

    public IActionResult OnGet()
    {
        return Page();
    }

    [TempData]
    public string Message { get; set; }

    [BindProperty]
    public Customer Customer { get; set; }

    public async Task<IActionResult> OnPostAsync()
    {
        if (!ModelState.IsValid)
        {
            return Page();
        }

        _context.Customer.Add(Customer);
        await _context.SaveChangesAsync();
        Message = $"Customer {Customer.Name} added";

        return RedirectToPage("./IndexPeek");
    }
}
```

The following page displays `TempData["Message"]`:

CSHTML

```
@page
@model IndexModel

<h1>Peek Contacts</h1>

@{
    if (TempData.Peek("Message") != null)
    {
        <h3>Message: @TempData.Peek("Message")</h3>
    }
}
```

```
@*Content removed for brevity.*@
```

In the preceding markup, at the end of the request, `TempData["Message"]` is **not** deleted because `Peek` is used. Refreshing the page displays the contents of `TempData["Message"]`.

The following markup is similar to the preceding code, but uses `Keep` to preserve the data at the end of the request:

CSHTML

```
@page
@model IndexModel

<h1>Contacts Keep</h1>

@{
    if (TempData["Message"] != null)
    {
        <h3>Message: @TempData["Message"]</h3>
    }
    TempData.Keep("Message");
}

@*Content removed for brevity.*@
```

Navigating between the *IndexPeek* and *IndexKeep* pages won't delete `TempData["Message"]`.

The following code displays `TempData["Message"]`, but at the end of the request, `TempData["Message"]` is deleted:

CSHTML

```
@page
@model IndexModel

<h1>Index no Keep or Peek</h1>

@{
    if (TempData["Message"] != null)
    {
        <h3>Message: @TempData["Message"]</h3>
    }
}
```

```
@*Content removed for brevity.*@
```

TempData providers

The cookie-based TempData provider is used by default to store TempData in cookies.

The cookie data is encrypted using IDataProtector, encoded with Base64UrlTextEncoder, then chunked. The maximum cookie size is less than 4096 bytes due to encryption and chunking. The cookie data isn't compressed because compressing encrypted data can lead to security problems such as the CRIME and BREACH attacks. For more information on the cookie-based TempData provider, see CookieTempDataProvider.

Choose a TempData provider

Choosing a TempData provider involves several considerations, such as:

- Does the app already use session state? If so, using the session state TempData provider has no additional cost to the app beyond the size of the data.
- Does the app use TempData only sparingly for relatively small amounts of data, up to 500 bytes? If so, the cookie TempData provider adds a small cost to each request that carries TempData. If not, the session state TempData provider can be beneficial to avoid round-tripping a large amount of data in each request until the TempData is consumed.
- Does the app run in a server farm on multiple servers? If so, there's no additional configuration required to use the cookie TempData provider outside of Data Protection. For more information, see ASP.NET Core Data Protection Overview and Key storage providers.

Most web clients such as web browsers enforce limits on the maximum size of each cookie and the total number of cookies. When using the cookie TempData provider, verify the app won't exceed these limits. Consider the total size of the data. Account for increases in cookie size due to encryption and chunking.

Configure the TempData provider

The cookie-based TempData provider is enabled by default.

To enable the session-based TempData provider, use the AddSessionStateTempDataProvider extension method. Only one call to `AddSessionStateTempDataProvider` is required:

C#

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddRazorPages()
                    .AddSessionStateTempDataProvider();
builder.Services.AddControllersWithViews()
                    .AddSessionStateTempDataProvider();

builder.Services.AddSession();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.UseSession();

app.MapRazorPages();
app.MapDefaultControllerRoute();

app.Run();
```

Query strings

A limited amount of data can be passed from one request to another by adding it to the new request's query string. This is useful for capturing state in a persistent manner that allows links with embedded state to be shared through email or social networks. Because URL query strings are public, never use query strings for sensitive data.

In addition to unintended sharing, including data in query strings can expose the app to Cross-Site Request Forgery (CSRF) attacks. Any preserved session state must protect against CSRF attacks. For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

Hidden fields

Data can be saved in hidden form fields and posted back on the next request. This is common in multi-page forms. Because the client can potentially tamper with the data, the app must always revalidate the data stored in hidden fields.

`HttpContext.Items`

The HttpContext.Items collection is used to store data while processing a single request. The collection's contents are discarded after a request is processed. The `Items` collection is often used to allow components or middleware to communicate when they operate at different points in time during a request and have no direct way to pass parameters.

In the following example, middleware adds `isVerified` to the `Items` collection:

C#

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

ILogger logger = app.Logger;

app.Use(async (context, next) =>
{
    // context.Items["isVerified"] is null
    logger.LogInformation($"Before setting: Verified:
{context.Items["isVerified"]}");
    context.Items["isVerified"] = true;
    await next.Invoke();
});

app.Use(async (context, next) =>
{
    // context.Items["isVerified"] is true
    logger.LogInformation($"Next: Verified: {context.Items["isVeri-
fied"]}");
    await next.Invoke();
});

app.MapGet("/", async context =>
{
    await context.Response.WriteAsync($"Verified:
{context.Items["isVerified"]}");
});

app.Run();
```

For middleware that's only used in a single app, it's unlikely that using a fixed `string` key would cause a key collision. However, to avoid the possibility of a key collision

altogether, an `object` can be used as an item key. This approach is particularly useful for middleware that's shared between apps and also has the advantage of eliminating the use of key strings in the code. The following example shows how to use an `object` key defined in a middleware class:

C#

```
public class HttpContextItemsMiddleware
{
    private readonly RequestDelegate _next;
    public static readonly object HttpContextItemsMiddlewareKey =
new();

    public HttpContextItemsMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext httpContext)
    {
        httpContext.Items[HttpContextItemsMiddlewareKey] = "K-9";

        await _next(httpContext);
    }
}

public static class HttpContextItemsMiddlewareExtensions
{
    public static IApplicationBuilder
        UseHttpContextItemsMiddleware(this IApplicationBuilder app)
    {
        return app.UseMiddleware<HttpContextItemsMiddleware>();
    }
}
```

Other code can access the value stored in `HttpContext.Items` using the key exposed by the middleware class:

C#

```
public class Index2Model : PageModel
{
    private readonly ILogger<Index2Model> _logger;

    public Index2Model(ILogger<Index2Model> logger)
    {
        _logger = logger;
    }

    public void OnGet()
    {
```

```
        HttpContext.Items

.TryGetValue(HttpContextItemsMiddleware.HttpContextItemsMiddlewareKey
,
                out var middlewareSetValue);

        _logger.LogInformation("Middleware value {MV}",
            middlewareSetValue?.ToString() ?? "Middleware value not
set!");
    }
}
```

Cache

Caching is an efficient way to store and retrieve data. The app can control the lifetime of cached items. For more information, see Response caching in ASP.NET Core.

Cached data isn't associated with a specific request, user, or session. **Do not cache user-specific data that may be retrieved by other user requests.**

To cache application wide data, see Cache in-memory in ASP.NET Core.

Checking session state

ISession.IsAvailable is intended to check for transient failures. Calling `IsAvailable` before the session middleware runs throws an `InvalidOperationException`.

Libraries that need to test session availability can use `HttpContext.Features.Get<ISessionFeature>()?.Session != null`.

Common errors

- "Unable to resolve service for type 'Microsoft.Extensions.Caching.Distributed.IDistributedCache' while attempting to activate 'Microsoft.AspNetCore.Session.DistributedSessionStore'."

  This is typically caused by failing to configure at least one `IDistributedCache` implementation. For more information, see Distributed caching in ASP.NET Core and Cache in-memory in ASP.NET Core.

If the session middleware fails to persist a session:

- The middleware logs the exception and the request continues normally.

- This leads to unpredictable behavior.

The session middleware can fail to persist a session if the backing store isn't available. For example, a user stores a shopping cart in session. The user adds an item to the cart but the commit fails. The app doesn't know about the failure so it reports to the user that the item was added to their cart, which isn't true.

The recommended approach to check for errors is to call `await feature.Session.CommitAsync` when the app is done writing to the session. CommitAsync throws an exception if the backing store is unavailable. If `CommitAsync` fails, the app can process the exception. LoadAsync throws under the same conditions when the data store is unavailable.

Additional resources

View or download sample code (how to download)

Host ASP.NET Core in a web farm