**Sessions in ASP.NET Core MVC**

In this article, I will discuss **Sessions in ASP.NET Core MVC Applications** with Examples. Please read our previous article discussing [**Differences Between Persistent and Non-Persistent Cookies in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/persistent-vs-non-persistent-cookies-in-asp-net-core-mvc/) Applications. At the end of this article, you will understand the following pointers:

* **What are Sessions in Web Applications?**
* **How Does Session Work in Web Application?**
* **Types of Sessions in Web Applications**
* **How To Use Session in ASP.NET Core MVC?**
* **How to Set and Get Session Values in ASP.NET Core MVC?**
* **How do you Access the Session Object in a View?**
* **Use Cases of Sessions in ASP.NET Core MVC**
* **Limitations of Sessions in ASP.NET Core Web Application**

**What are Sessions in Web Applications?**

Sessions in Web Applications refer to the mechanism of storing user-specific data temporarily on the server side across multiple requests. Unlike cookies, which are stored on the client side, session data is stored on the server side, enhancing security and reducing the risk of exposing sensitive information. When a user interacts with a web application, a session allows the server to store information relevant to that user during the interaction period. This is useful in stateless HTTP protocols where each request is independent.

**How Does Session Work in Web Application?**

Sessions typically generate a unique identifier (session ID) for each user session upon their first interaction with the application. This identifier (session ID) is stored as a cookie on the client side (usually), and the corresponding data is stored on the Web Server. When the client makes subsequent requests, this session ID is sent in the Request header. The server uses this identifier to retrieve session-specific data stored in memory (Temporary Caching Mechanism), a database, or another persistent storage mechanism. This data persists until the session expires (due to user inactivity or logout) or is manually cleared. For a better understanding, please have a look at the following diagram.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=0d20d600-061c-4df7-6c44-a1f3cfaf5a2b&d_id=118560&imp_id=1428402288504328&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

Let’s understand how sessions work in web applications, step by step:

**Step 1: Client Makes an Initial Request**

When users first visit a web application, they send a request to the server (e.g., by entering a URL into their browser or clicking a link). If the user has no existing session, this is recognized as a new session request.

**Step 2: Server Creates a Session**

Upon receiving the request, the server checks if an existing session is associated with this client. Since this is the initial request, no session exists. The server creates a unique session identifier (session ID). This Session ID is important as it will track the user’s session across multiple requests.

**Step 3: Server Stores Session Data**

The session object can store various data, such as user preferences, login status, shopping cart contents, etc., and the session data is stored on the server side in a session store that maps the Session ID. The session store could be in memory, a database, or a distributed cache like Redis.

**Step 4: Server Sends Session ID to Client**

The server sends the session ID back to the client in the response. This is usually done by setting a cookie in the response header. The cookie contains the session ID and is typically configured with attributes like HttpOnly, Secure, etc.

**Step 5: Client Stores the Session ID**

The client’s browser stores the session ID cookie under the domain from which it came. This cookie will be sent with every subsequent request to the server, allowing the server to identify the session.

**Step 6: Client Makes Subsequent Requests**

For every subsequent request made by the user’s browser to the server, the browser sends the cookie containing the Session ID back to the server. This happens automatically with every request to the domain, provided the cookie has not expired or been deleted.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=0d20d600-061c-4df7-6c44-a1f3cfaf5a2b&d_id=118560&imp_id=7558984898542385&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

**Step 7: Server Retrieves Session Data**

With each incoming request that includes a Session ID, the server looks up the session ID in the session store. If the session ID is valid and not expired, the server retrieves the session data associated with that ID.

**Step 8: Server Processes the Request**

The server processes the request, using the session data as needed. For example, it might use the session data to confirm that the user is logged in or to retrieve items in the user’s shopping cart.

**Step 8: Server Updates Session Data (if needed)**

The server may update the session data based on the request. For example, if the user adds an item to their shopping cart, the server updates the session data to reflect this change under the user’s Session ID.

**Step 9: Server Sends Response**

The server sends a response back to the client, including updates to the session ID cookie (e.g., to extend its expiration time).

**Step 10: Idle Timeout and Expiration**

Sessions typically have an expiration (e.g., 20 minutes of inactivity) or an idle timeout. If the client does not make a request within this period, the session expires and is removed from the session store. On subsequent requests after expiration, the server will create a new session. This is a security measure and a way to free up resources.

**Step 11: User Logs Out**

If the application involves a user log-in, the user can typically log out. When this happens, the session can be explicitly ended by the server. The server then deletes the session data, and usually, the client is instructed to delete the session cookie.

**Step 12: Session End**

Whether a session ends due to logout, expiration, or the user’s manual cookie deletion, once the session ends, the server no longer retains the session data. Any subsequent request will be treated as a new session, repeating the process from Step 1.

Advertisements

**Note:** Each browser has a unique SessionId that is deleted at the end of the session and not shared with other browsers. The default session timeout is 20 minutes but can be configured based on business requirements.

**Types of Sessions in Web Applications**

There are primarily two types of sessions in web applications:

* **In-Memory or In-Proc Sessions:** Data is stored in memory on the web server. This is fast and suitable for small-scale applications but not suitable for large-scale applications or load-balancing environments.
* **Distributed or Out-Proc Sessions:** Data is stored outside the web server process, often using databases or external caching services like Redis or Memcached. This approach is ideal for load-balanced environments to maintain session data consistency.

**Note:** In this article, I will discuss Implementing **In-Memory or In-Proc Sessions in ASP.NET Core MVC** Applications. In our next article, I will discuss Implementing [**Distributed or Out-Proc Sessions in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/in-memory-vs-distributed-sessions-in-asp-net-core-mvc/) and the differences between them.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=0d20d600-061c-4df7-6c44-a1f3cfaf5a2b&d_id=118560&imp_id=4335074378564432&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

**How To Use Session in ASP.NET Core MVC?**

The **Microsoft.AspNetCore.Session** package offers a middleware component for managing sessions in ASP.NET Core Web Applications. The Microsoft.AspNetCore.Session Package is included implicitly by the Framework. To enable Session in ASP.NET Core Web Application, the Main method of the Program class must contain the following:

1. Any of the **IDistributedCache**Memory Cache Service. The IDistributedCache implementation stores user data in the server by creating a SessionId as the key. We will discuss **Distributed Caching** in detail in our upcoming articles.
2. A call to the **AddSession**Service method where we need to configure the session time out, whether the session is essential or not, and whether we want to access the session using HTTP only or not.
3. A call to the **UseSession** Middleware component to register the session into the application processing pipeline. It should be registered after the Routing and before the MVC Middleware component. This is because we want the session middleware component to be executed before executing the action method.

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

Here,

* **AddDistributedMemoryCache:** This method provides the necessary in-memory storage backend for storing session data. It configures IMemoryCache, allowing us to use memory caching to store session data.
* **AddSession:** Sets up the session services and configures the session behavior, such as timeout and cookie settings.
* **UseSession:** This function adds the session middleware to the application’s request pipeline, ensuring the session can be used throughout the application.

**Session Options in ASP.NET Core MVC:**

In ASP.NET Core MVC, the SessionOptions class provides various properties to configure session behavior. Each of these properties serves a specific purpose, enhancing the functionality and security of your session management. Please have a look at the following AddSessions service:

**Explanation of Session Options Properties:**

**IdleTimeout: options.IdleTimeout = TimeSpan.FromSeconds(10);**

This property sets the duration for which the session can remain idle (no requests made by the user) before it is abandoned (i.e., removed from the server). If the user does not make a request within this time frame, the session data will be lost. This property is set to TimeSpan.FromSeconds(10), meaning the session will expire if no requests are made within 10 seconds. This is useful for limiting the lifespan of sessions to enhance security and reduce server load.

Advertisements

**IOTimeout: options.IOTimeout = TimeSpan.FromSeconds(10);**

This property sets the maximum time allowed for the session middleware for input/output operations, i.e., to load a session from the store or save it back to the store. If the session data cannot be read from or written to the store within this time frame, an exception is thrown. This is important in scenarios where quick data retrieval is essential for performance. In our example, it’s set to TimeSpan.FromSeconds(10), indicating that I/O operations related to session state (like reading from or writing to session) should be completed within 10 seconds.

**Cookie.Name: options.Cookie.Name = “.MySampleMVCWeb.Session”;**

This property sets the name of the cookie used to store the session ID on the client’s browser. Here, the session cookie is named “.MySampleMVCWeb.Session”. A specific cookie name can help differentiate your session cookies from other applications on the same domain. By default, this cookie is named “.AspNetCore.Session”.

**Cookie.HttpOnly: options.Cookie.HttpOnly = true;**

This property determines if the cookie is accessible only through HTTP (not via client-side scripts like JavaScript). When set to true, this property ensures that the cookie is accessible only to the server side and not available through client-side scripts (e.g., JavaScript). This helps prevent cross-site scripting (XSS) attacks, where unauthorized scripts try to access the cookies.

**Cookie.IsEssential: options.Cookie.IsEssential = true;**

This property indicates whether the cookie is essential for the application to function correctly. When set to true, the essential cookies bypass consent checks under certain privacy regulations like the GDPR. This is useful for cookies that are necessary for user sessions and authentication.

**Cookie.Path: options.Cookie.Path = “/”;**

This property sets the path for which the cookie is valid. By setting the path to /, the cookie is available on all pages of the application. This is the default setting, but you can restrict it to a specific part of the application if needed. For example, options.Cookie.Path = “/Home”; (limits the cookie to the specified path).

**Cookie.SecurePolicy: options.Cookie.SecurePolicy = CookieSecurePolicy.Always;**

This property determines under what conditions the cookie should be sent over HTTPS. Setting it to Always ensures that the cookie is only sent over HTTPS, providing additional security by preventing the cookie from being transmitted over insecure connections. The options are as follows:

* **options.Cookie.SecurePolicy = CookieSecurePolicy.Always;** (cookie sent only over HTTPS).
* **options.Cookie.SecurePolicy = CookieSecurePolicy.None;** (cookie sent over HTTP or HTTPS).
* **options.Cookie.SecurePolicy = CookieSecurePolicy.SameAsRequest;** (cookie sent using the same protocol as the request).

**Configuring the Sessions in ASP.NET Core MVC:**

So, please modify the Main method of the Program class as follows to configure sessions for our application:

**namespace** *SampleMVCWeb*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**(string[]** args**)**

**{**

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

// Add services to the container.

builder.Services.AddControllersWithViews**()**;

//Configuring Session Services in ASP.NET Core

builder.Services.AddDistributedMemoryCache**()**;

builder.Services.AddSession**(**options =**>**

**{**

options.IdleTimeout = TimeSpan.FromSeconds**(**10**)**;

options.IOTimeout = TimeSpan.FromSeconds**(**10**)**;

options.Cookie.Name = ".MySampleMVCWeb.Session";

options.Cookie.HttpOnly = **true**;

options.Cookie.IsEssential = **true**;

options.Cookie.Path = "/";

options.Cookie.SecurePolicy = CookieSecurePolicy.Always;

**})**;

var app = builder.Build**()**;

// Configure the HTTP request pipeline.

**if** **(**!app.Environment.IsDevelopment**())**

**{**

app.UseExceptionHandler**(**"/Home/Error"**)**;

// The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.

app.UseHsts**()**;

**}**

app.UseHttpsRedirection**()**;

app.UseStaticFiles**()**;

app.UseRouting**()**;

app.UseAuthorization**()**;

//Configuring Session Middleware in ASP.NET Core

//It should and must be configured after UseRouting and before MapControllerRoute

app.UseSession**()**;

app.MapControllerRoute**(**

name: "default",

pattern: "{controller=Home}/{action=Index}/{id?}"**)**;

app.Run**()**;

**}**

**}**

**}**

**Note:** If two requests simultaneously attempt to modify the content of a session, the last request overrides the first.

**How to Set and Get Session Values in ASP.NET Core MVC**

In ASP.NET Core, the ISession interface provides methods to work with session data in a key-value pair format. These methods allow us to get and set session values easily. The following is the list of ISession extension methods:

* **Get(ISession, String):** Retrieves a value from the session as a byte array. If the key does not exist, it returns null.
* **GetInt32(ISession, String):** This method tries to read an integer value from the session. If the key is not found or the value cannot be converted to an int, it returns null.
* **GetString(ISession, String):** This method retrieves a string value from the session. If the key does not exist, it returns null.
* **SetInt32(ISession, String, Int32):** Sets an integer value in the session.
* **SetString(ISession, String, String):** Sets a string value in the session.

**Modifying the Home Controller:**

Next, modify the HomeController as follows. As you can see, we have created two constant variables to store the session keys. We store user data in the session object within the Index Action method. Further, if you notice, we are accessing the session data from the About action method.

**using** *Microsoft.AspNetCore.Mvc;*

**namespace** *SampleMVCWeb.Controllers*

**{**

**public** **class** HomeController : Controller

**{**

const **string** SessionUserId = "\_UserId";

const **string** SessionUserName = "\_UserName";

**public** IActionResult Index**()**

**{**

//Let assume the User is logged in and we need to store the user information in the session

//Storing Data into Session using SetString and SetInt32 method

HttpContext.Session.SetString**(**SessionUserName, "pranaya@dotnettutotials.net"**)**;

HttpContext.Session.SetInt32**(**SessionUserId, 1234567**)**;

**return** View**()**;

**}**

**public** **string** About**()**

**{**

//Accessing the Session Data inside a Method

**string**? UserName = HttpContext.Session.GetString**(**SessionUserName**)**;

**int**? UserId = HttpContext.Session.GetInt32**(**SessionUserId**)**;

**string** Message = $"UserName: {UserName}, UserId: {UserId}";

**return** Message;

**}**

**public** IActionResult Privacy**()**

**{**

**return** View**()**;

**}**

**}**

**}**

**Note:** After 10 seconds, the cookie expires, and you will not get the data once the cookie expires.

**How do you Access the Session Object in a View?**

To use the Session inside a View, we need to inject the **IHttpContextAccessor** object into our view and then use the IHttpContextAccessor object to get the HttpContext and Session. For this, we need to add the following two statements in our view. The first statement adds the namespace, and the second statement injects the IHttpContextAccessor. Using this HttpContextAccessor, we can access the HttpContext and Session object.  
**@using Microsoft.AspNetCore.Http;**  
**@inject IHttpContextAccessor HttpContextAccessor**

**Modifying the Index.cshtml view:**

@using Microsoft.AspNetCore.Http;

@inject IHttpContextAccessor HttpContextAccessor

@{

ViewData["Title"] = "Home Page";

}

**<div** class="text-left"**>**

**<b>**User Name:**</b>** @HttpContextAccessor?.HttpContext?.Session.GetString("\_UserName")

**<br** **/>**

**<b>**User Id:**</b>** @HttpContextAccessor?.HttpContext?.Session.GetInt32("\_UserId")

**</div>**

**Modifying the Privacy.cshtml view:**

@using Microsoft.AspNetCore.Http;

@inject IHttpContextAccessor HttpContextAccessor

@{

ViewData["Title"] = "Privacy Policy";

}

**<h1>**@ViewData["Title"]**</h1>**

**<div** class="text-left"**>**

**<b>**User Name:**</b>** @HttpContextAccessor?.HttpContext?.Session.GetString("\_UserName")

**<br** **/>**

**<b>**User Id:**</b>** @HttpContextAccessor?.HttpContext?.Session.GetInt32("\_UserId")

**</div>**

If you run the application with the above changes, you will get the following exception: We use the HttpContextAccessor service but have not yet configured it into the built-in IOC container.

Now, we need to configure the HttpContextAccessor service within the Program class’s main method. So, please add the following code to the Program class.

**builder.Services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();**

Now, run the application. By default, it will execute the Index Action Method of the Home Controller and set the session values. Then, it will render the Index view, where you can see the session data, as shown in the image below.

When it returns the response to the client (Web Browser), the Web Server also includes the generated SessionId in the response header using the Set-Cookie. To confirm this, open a different web browser, open the developer tool, and then issue a request to the Index action method. Then, go to the Network tab, inspect the request, and go to the Response header. You will see the Set-Cookie property where the web server sends the SessionId and its unique value. This SessionId value is going to be unique per session.

From the next HTTP Request onwards, the Client (Browser) will send the SessionId using the Cookie request header. Within 10 seconds, if you visit the Privacy page, it will also render the session data, as shown in the image below.

If you check the Network tab and inspect the request, you will see that one cookie is being sent from the client to the web server using the Request header, as shown in the image below. This happens in each and every request from the client to the server as long as the session has not expired.

Again, if you visit the About page within 10 seconds, it will also render the session data, as shown in the image below.

Now, if you visit the Privacy page (Not the Index page where we store the session data) after 10 seconds, you will not get the session data as shown in the image below.

In this case, it also sends the expired cookie in the HTTP request header. If you want, access the Index Action method and reset the Session timeout. Then, the same SessionId will be used.

**Use Cases of Sessions in ASP.NET Core MVC:**

The following are some of the situations where using sessions is appropriate:

**User Authentication:**

Store user login information after authentication to maintain the user’s login status across different pages. After a user logs in, their user ID or username is stored in the session. This allows the application to recognize the user on subsequent requests without requiring them to log in again.

**Shopping Cart:**

Track items added to a shopping cart until the user checks out or leaves the site. Items added to a shopping cart are stored in the session so users can navigate different pages and retain their cart contents.

**User Preferences:**

Store user preferences like language, theme, notification, or layout settings. Save the user’s preferred language in the session to provide a personalized experience across different pages.

**Temporary Data Storage:**

Store data temporarily between requests, such as data across different steps of a multi-step form. Store form data in the session so that it can be retrieved and used in subsequent steps of a multi-step process.

**Security Tokens:**

Store security tokens, such as CSRF tokens, to validate requests and enhance security. Generate and store a CSRF token in the session to validate forms and prevent cross-site request forgery attacks.

**Limitations of Sessions in ASP.NET Core Web Application**

The following are the Limitations of using Sessions in ASP.NET Core Web Application:

* **Scalability Issues:** In-memory session storage is limited to a single server instance. If your application is deployed across multiple servers or instances (e.g., in a load-balanced environment), in-memory sessions will not work as each server instance maintains its own session state. This can lead to inconsistencies and lost session data if requests are not routed to the same server.
* **Performance Overhead:** Storing and retrieving session data can introduce performance overhead. Accessing session data, especially when stored in a database or distributed cache, can add a performance penalty to each request. Frequent read/write operations can degrade application performance.
* **Data Storage Limitations:** Sessions store small amounts of temporary data. Storing large amounts of data in sessions can lead to increased memory usage and performance issues. It is not recommended to store large objects or large collections in sessions.
* **Security Concerns:** Sessions can be vulnerable to various security attacks if not managed properly. Session hijacking (stealing session IDs) and session fixation (forcing a user to use a known session ID) are common threats. Properly using secure cookies (HTTP-only, Secure flag) and regular session ID regeneration can handle some risks.
* **Session Timeout:** Sessions have a predefined timeout period after which they expire. If a user is inactive for longer than the session timeout period, their session data is lost, which can lead to a poor user experience. This can be problematic for applications requiring long periods of inactivity, such as filling out long forms.
* **Persistence:** Sessions are typically not persistent across browser sessions. When a user closes their browser or tab, the session data is typically lost. This can be a limitation for use cases requiring data persistence beyond a single browser session.
* **Resource Consumption:** Sessions consume server resources. Each session consumes memory or other resources on the server. In applications with a large number of concurrent users, this can lead to increased resource consumption and scalability issues.

In the next article, I will explain [**In-Memory or In-Proc** **and Distributed or Out-Proc Sessions in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/in-memory-vs-distributed-sessions-in-asp-net-core-mvc/) with Examples. In this article, I try to explain **Sessions in ASP.NET Core MVC** Applications with Examples. I hope you enjoy this Session in the ASP.NET Core MVC Application article.