**Cookies in ASP.NET Core MVC**

In this article, I will discuss **Cookies in ASP.NET Core MVC** Applications with Examples. Before understanding what cookies are and how to implement them, let’s first understand why we need Cookies and Sessions in Web Applications.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=ed184184-1d8f-40c2-5f1a-650afc213801&d_id=118560&imp_id=5664936854963758&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

**Why Do We Need Sessions and Cookies in Web Applications?**

The HTTP Protocol is a stateless protocol used to communicate between a server and a client on the Web. This means that information will not be automatically shared between two requests. To share the information between multiple requests, we need to use Cookies or Sessions. So, Sessions and Cookies are essential components in web applications for managing state and user data across multiple requests.

* **HTTP Protocol:** HTTP is stateless, meaning each request from a client to a server is independent, and the server does not retain any information about previous requests.
* **State Management:** To build functional web applications, we need a way to remember user interactions and data across multiple requests. This is where sessions and cookies come into play.

Nowadays, most server-side technologies, including the ASP.NET Core MVC framework, have built-in support for Cookies and Sessions. Let us proceed and understand Cookies in this article. In our upcoming articles, we will discuss [**Sessions**](https://dotnettutorials.net/lesson/sessions-in-asp-net-core-mvc/) in Detail.

**What is a Cookie?**

A cookie is a small piece of data that a server sends to the client (usually a web browser), which is then stored on the client’s machine. Cookies are primarily used to store information about the user or their interaction with the website, such as user ID, user name, login status, token, etc, and are sent back to the website with every subsequent request. Typically, it is used to tell the web server if two requests are coming from the same web browser.

Advertisements

Cookies are stored on the client side (user’s computer) in the form of name-value pairs and can be accessed by both the web browser and the web server. You can use the keys to read, write, or delete cookies. For a better understanding of how cookies work in a web application, please look at the following diagram.

**How Do Cookies Work in Web Applications?**

The following is a detailed step-by-step explanation of how cookies work in a web application involving interactions between the client (browser) and the server.

* **Client Makes an Initial Request:** When a client (browser) makes an initial request to a web application, it typically doesn’t have any cookies associated with the server.
* **Server Responds and Sets a Cookie:** The server processes the request and decides to set a cookie. The server sends a Set-Cookie header in the HTTP response. You can also set multiple cookies as per your requirement. For each cookie, it will add one Set-Cookie in the response header.
* **Client Stores the Cookie:** The client’s browser receives the response and stores the cookie according to the attributes specified (name, value, expiration, path, HttpOnly, etc.).
* **Client Makes Subsequent Requests with the Cookie:** For subsequent requests to the same domain and path, the browser automatically includes the cookie in the request headers (as long as the cookie is not expired and not deleted). The client will send only one Cookie in the request header containing all cookie data.
* **Server Receives and Reads the Cookie:** The server processes the request and reads the cookie from the Cookie header to perform actions based on the cookie’s value.
* **Server Uses the Cookie Data:** The server can use the data stored in the cookie to maintain state, authenticate the user, or provide personalized content based on the cookie’s value.
* **Server Updates or Deletes the Cookie (Optional):** The server can update the cookie by sending a new Set-Cookie header with the same name but a different value. It can also delete the cookie by setting its expiration date to a past date.

**Types of Cookies in Web Applications:**

There are two types of Cookies. They are as follows:

* **Persistent Cookies:**Persistent cookies are long-term cookies that can be stored across multiple sessions. They retain information such as login credentials, allowing users to have a personalized experience when they revisit a website.
* **Non-Persistent Cookies:**Non-Persistent Cookies are temporary cookies stored on the user’s computer while browsing a website. They are typically used to maintain a session state and are destroyed when the client closes their browser or navigates away from the web page.

**Note:** In this article, I will give an example of using persistent cookies. In our next article, I will discuss the example using **Non-Persistent Cookies** and then discuss the differences between them.

**How Do We Write, Read, and Delete Cookies in ASP.NET Core MVC?**

In ASP.NET Core MVC, working with cookies involves writing, reading, and deleting cookies using built-in methods and properties provided by the ASP.NET Core MVC Framework. Let us proceed and understand how to Read, Write, and Delete Cookies in an ASP.NET Core MVC Application:

Advertisements

**Writing a Cookie in ASP.NET Core MVC:**

To create a Cookie in ASP.NET Core MVC, we need to create an instance of the CookieOptions class. Then, we need to set the expiry date using the Expires property and add the cookie to **Response.Cookies** collection using the Append method with the Name, Value, and CookieOptions object as parameters as follows.

CookieOptions options = new CookieOptions

**{**

Domain = "example.com", // Set the domain for the cookie

Expires = DateTime.Now.AddDays**(**7**)**, // Set expiration date to 7 days from now

Path = "/", // Cookie is available within the entire application

Secure = **true**, // Ensure the cookie is only sent over HTTPS

HttpOnly = **true**, // Prevent client-side scripts from accessing the cookie

MaxAge = TimeSpan.FromDays**(**7**)**, // Another way to set the expiration time

IsEssential = **true** // Indicates the cookie is essential for the application to function

**}**;

Response.Cookies.Append**(**"UserId", "1234567", options**)**;

Response.Cookies.Append**(**"UserName", "pranaya@dotnettutotials.net", options**)**;

**Explanation of the CookieOptions Properties:**

The CookieOptions class provides various properties that can be used when creating a Cookie in ASP.NET Core MVC Application. The use of the properties are as follows:

* **Domain:** Specifies the domain for which the cookie is valid. Including this setting can make the cookie accessible across subdomains, depending on the specified domain.
* **Expires:** This property sets a specific date and time for when the cookie should expire. After this time, the client’s browser will automatically delete the cookie.
* **Path:** Defines the path under the domain for which the cookie is valid. If set to “/”, the cookie is available across the entire domain.
* **Secure:** When set to true, the cookie will only be sent over secure (HTTPS) connections. This is important for protecting the cookie from being intercepted by attackers.
* **HttpOnly:** When set to true, the cookie cannot be accessed through client-side scripts. This is useful for protecting cookies from access via XSS attacks.
* **MaxAge:** Specifies the duration for which the cookie will remain valid from the time it is created. It’s an alternative to Expires but defined in terms of a TimeSpan instead of a specific date and time.
* **IsEssential:** Indicates whether the cookie is essential for the application to function. If true, then consent policy checks may be bypassed. The default value is false.

**Reading a Cookie in ASP.NET Core MVC:**

To read a cookie value from an incoming HTTP request in ASP.NET Core MVC, we can use the **Request.Cookies** collection, and we need to pass the key as follows. If the cookie exists, it will retrieve its value. It’s a good programming practice to check if the cookie exists to prevent runtime exceptions.

**string? UserName = Request.Cookies[“UserName”];**  
**int? UserId = Convert.ToInt32(Request.Cookies[“UserId”]);**

**Deleting a Cookie in ASP.NET Core MVC:**

To delete a cookie, we need to use the Delete method on Response.Cookies collection in ASP.NET Core MVC, and we need to pass the Cookie key name as follows:

**Response.Cookies.Delete(“UserId”);**  
**Response.Cookies.Delete(“UserName”);**

**Modifying the Home Controller:**

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

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

**namespace** *SampleMVCWeb.Controllers*

**{**

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

**{**

const **string** CookieUserId = "UserId";

const **string** CookieUserName = "UserName";

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

**{**

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

CookieOptions options = new CookieOptions**()**

**{**

Domain = "localhost", // Set the domain for the cookie

Path = "/", // Cookie is available within the entire application

Expires = DateTime.Now.AddDays**(**7**)**, // Set cookie expiration to 7 days from now

Secure = **false**, // Ensure the cookie is only sent over HTTPS (set to false for local development)

HttpOnly = **true**, // Prevent client-side scripts from accessing the cookie

IsEssential = **true** // Indicates the cookie is essential for the application to function

**}**;

// Append UserId to the cookies

Response.Cookies.Append**(**CookieUserId, "1234567", options**)**;

// Append UserName to the cookies

Response.Cookies.Append**(**CookieUserName, "pranaya@dotnettutotials.net", options**)**;

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

**}**

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

**{**

// Accessing the Cookie Data inside a Method

// Try to get the UserName from the cookies, if it does not exist, default to null

**string**? UserName = Request.Cookies.ContainsKey**(**CookieUserName**)** ? Request.Cookies**[**CookieUserName**]** : **null**;

// Try to get the UserId from the cookies and convert it to an integer

// If it does not exist or conversion fails, default to null

**int**? UserId = **null**;

**if** **(**Request.Cookies.ContainsKey**(**CookieUserId**))**

**{**

**bool** isValidInt = **int**.TryParse**(**Request.Cookies**[**CookieUserId**]**, out **int** parsedUserId**)**;

**if** **(**isValidInt**)**

**{**

UserId = parsedUserId;

**}**

**}**

// Create a message with the UserName and UserId

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

**return** Message;

**}**

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

**{**

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

**}**

**public** **string** DeleteCookie**()**

**{**

// Create CookieOptions to match the domain and path used when setting the cookies

CookieOptions options = new CookieOptions

**{**

Domain = "localhost", // Match the domain used when setting the cookie

Path = "/" // Match the path used when setting the cookie

**}**;

// Delete the cookies from the browser

Response.Cookies.Delete**(**CookieUserId, options**)**;

Response.Cookies.Delete**(**CookieUserName, options**)**;

**return** "Cookies are Deleted";

**}**

**}**

**}**

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

To use Cookies inside a View, we need to inject the IHttpContextAccessor service into our view and use it to get the HttpContext and Request objects from that. 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?.Request.Cookies["UserName"]

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

**<b>**User Id:**</b>** @HttpContextAccessor?.HttpContext?.Request.Cookies["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?.Request.Cookies["UserName"]

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

**<b>**User Id:**</b>** @HttpContextAccessor?.HttpContext?.Request.Cookies["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.**

Advertisements

Now, we need to configure the HttpContextAccessor service within the Program class. The following code does the same.

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

So, modify the Main method of the Program class as follows:

**namespace** *SampleMVCWeb*

**{**

**public** **class** Program

**{**

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

**{**

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

// Add services to the container.

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

//Adding the IHttpContextAccessor servive to the Dependency Injection IOC Container

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

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**()**;

app.MapControllerRoute**(**

name: "default",

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

app.Run**()**;

**}**

**}**

**}**

Now, run the application, and by default, it will execute the Index Action Method of the Home controller. Then, it will render the Index view where you can see the cookie data, as shown in the image below.

**Inspecting the Cookies using Browser Developer Tool:**

Now, open the browser developer tool, go to the Network tab, and inspect the request. The server will send two cookies in the response header, as shown in the image below. The cookie contains the key name, value, path, domain, and expiration date. As you can see, cookies store user information in plain text, which is not good from a security point of view.

If you visit the Privacy page, it will render the cookie data, as shown in the image below.

Now, if you go to the Network tab and inspect the request using the browser developer tool, you will see that along with the request, the web browser is now sending the cookie in the request header, as shown in the image below. This will happen for each and every request sent from the client to the server until the cookie is not expired or deleted.

If you visit the About page, it will also render the session data, as shown in the image below.

If you visit the DeleteCookie page, it will delete the cookies, and you will get the following message, as shown in the image below.

Advertisements

Now, if you verify the response header, you will see that the cookie values are being deleted, as shown in the image below.

If you visit the About and Privacy page, you will not see the cookie data shown in the image below.

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

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

**Authentication and Authorization:**

Cookies play an important role in managing user authentication (verifying identity) and authorization (granting access rights) in web applications.

For example, after a user successfully logs in, the server generates a session token or authentication cookie. This cookie typically contains a session ID or a securely hashed token that identifies the user’s authenticated session. As the user navigates through the application, the server verifies their access rights based on the session cookie. For example, certain pages or actions may only be accessible to authenticated users with specific roles or permissions.

**Remember Me Functionality:**

Cookies enable the “Remember Me” feature, allowing users to remain authenticated across multiple sessions without needing to log in each time they visit the site.

When a user selects “Remember Me” at login, a long-lived authentication cookie is stored on their device. This cookie contains a token or identifier linked to the user’s account. On subsequent visits, the server checks for the presence of this cookie. If valid, the user is automatically logged in without entering credentials again, enhancing the user experience.

**Shopping Carts:**

Cookies are commonly used to manage shopping carts in e-commerce applications. They allow users to store items they intend to purchase across different pages or visits.

For example, each time a user adds an item to their cart, a cookie stores the cart’s contents (like product IDs and quantities). Even if the user navigates away from the shopping cart page or closes their browser, the cookie ensures that their selected items remain in the cart until they complete the purchase or manually remove items.

**Form Data Persistence:**

Cookies can store form data temporarily to retain user input across different pages or during a single session. For example, if a user fills out a multi-step form or navigates away from a page, cookies can store form field values (like name, address, or preferences). This allows users to resume where they left off without re-entering information, improving usability.

**Single Sign-On (SSO):**

Cookies facilitate single sign-on, enabling users to authenticate once and access multiple related applications or services without re-entering credentials.

For example, upon initial login, a session cookie or token is issued. This token can authenticate users across various applications or domains within the same SSO ecosystem. As the user moves between different SSO-enabled applications, cookies maintain their authenticated state, providing a good user experience.

**Limitations of Cookies in ASP.NET Core Web Application:**

The following are the Limitations of Cookies in the ASP.NET Core Web Application

**Data Size Limit:**

Cookies have a size limit of approximately 4KB per cookie. This limitation means you can only store limited data in each cookie, which may not be sufficient for storing large amounts of data. If you attempt to store large amounts of data in a cookie, it may exceed this limit, leading to truncation or unexpected behavior. Attempting to store too much information in a single cookie could lead to data loss.

**Security Concerns:**

Cookies can be vulnerable to security threats such as cross-site scripting (XSS) and cross-site request forgery (CSRF). If cookies are not properly secured, they can be intercepted or manipulated by malicious users. Setting cookies without secure flags can expose them to interception. Developers must use secure and HttpOnly flags appropriately to avoid these risks.

**Storage on the Client Side:**

Cookies are stored on the client’s device (user’s browser), which means they can be accessed or manipulated by users or malicious scripts. Sensitive information should not be stored in plain text within cookies to prevent unauthorized access. Data stored in cookies can be deleted by the user at any time. Relying on cookies for essential data that could be lost if a user clears their browser cache.

**Browser Settings:**

Not all web browsers handle cookies in the same way, and users can configure their browsers to reject cookies entirely or delete them periodically, which can affect the functionality of your web application. For example, code that assumes cookies are always enabled and fails when they are not:

**Performance Impact:**

Sending and receiving cookies with each HTTP request can add to the overhead, especially when large numbers of cookies are involved. This can impact the performance of web applications by increasing load time.

In the next article, I will discuss [**How to Encrypt Cookies in an ASP.NET Core MVC**](https://dotnettutorials.net/lesson/encrypt-cookies-in-asp-net-core-mvc/) Application with Example. In this article, I try to explain Cookies in ASP.NET Core MVC Applications with Examples. I hope you enjoy this Cookie in ASP.NET Core MVC Application article.