<https://www.c-sharpcorner.com/article/authentication-and-authorization-in-asp-net-core-mvc-using-cookie/>

Authentication and Authorization are two major aspects while thinking about securing your application. Security is the main concern of modern applications because anyone can steal your data if it is not secured. So, if you are going to create an application where the data security is a primary concern, then think about Authentication and Authorization.

Authentication is the process to validate an anonymous user based on some credentials and Authorization process happens just after that and grants resources to this validated user. So, we can say, it's two-step validating process before providing the access of the resources or data.

We have many techniques to validate the users, like Windows Authentication, JWT Authentication, and Cookie Authentication etc. Today, we will learn how to implement and make ASP.NET Core MVC applications more secure using Cookie-based authentication and authorization. So, let's start the demonstration and create a fresh ASP.NET Core MVC project. You can refer to the following for the step by step process of creating an ASP.NET Core MVC application.

* [First Application in ASP.NET Core MVC 2.0](https://www.c-sharpcorner.com/article/first-application-in-asp-net-core-mvc-2-0/)

Be sure that while creating the project, your template should be Web Application (Model-View-Controller) and change the authentication as ‘No Authentication’.

You can download the code from [here](https://github.com/mukeshkumartech/CookieAuthenticationInAsp.NetCore).

Here, you can choose the inbuilt Authentication functionality instead of ‘No Authentication’ and it will provide the readymade code. But we are choosing ‘No Authentication’ here because we are going to add our own Cookie-based authentication functionality in this demo and you will learn how to implement the Authentication and Authorization system from scratch.

We are choosing MVC template because we would like to see some Login and Logout functionality on UI along with Authentication and Authorization using Cookies. Now, click OK and it will take a few seconds and the project will be ready. Run it for checking if everything is working fine or not. Once everything is OK, you are ready to go.

Let’s move to the starting point of the ASP.NET Core application file which is “Startup.cs” where we configure the setting for the application like configuring the required services and configuring the middleware services etc. So, implementing the Authentication features, first, we have to add the authentication and then use it. So, let’s move to Startup.cs’s ConfigureService method and add the authentication feature using the following line of code, it will be just above services.AddMvc().

1. services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme).AddCookie();

Now move to Configure in the startup.cs method and use the authentication features using the following line of code, it will be just above  routing.

1. app.UseAuthentication();

Following is the whole code for adding the Authentication and using it.

1. using Microsoft.AspNetCore.Authentication.Cookies;
2. using Microsoft.AspNetCore.Builder;
3. using Microsoft.AspNetCore.Hosting;
4. using Microsoft.AspNetCore.Http;
5. using Microsoft.AspNetCore.Mvc;
6. using Microsoft.Extensions.Configuration;
7. using Microsoft.Extensions.DependencyInjection;
9. namespace CookieDemo
10. {
11. **public** **class** Startup
12. {
13. **public** Startup(IConfiguration configuration)
14. {
15. Configuration = configuration;
16. }
18. **public** IConfiguration Configuration { get; }
20. // This method gets called by the runtime. Use this method to add services to the container.
21. **public** **void** ConfigureServices(IServiceCollection services)
22. {
23. services.Configure<CookiePolicyOptions>(options =>
24. {
25. // This lambda determines whether user consent for non-essential cookies is needed for a given request.
26. options.CheckConsentNeeded = context => **true**;
27. options.MinimumSameSitePolicy = SameSiteMode.None;
28. });
30. services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme).AddCookie();
31. services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_1);
32. }
34. // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
35. **public** **void** Configure(IApplicationBuilder app, IHostingEnvironment env)
36. {
37. **if** (env.IsDevelopment())
38. {
39. app.UseDeveloperExceptionPage();
40. }
41. **else**
42. {
43. app.UseExceptionHandler("/Home/Error");
44. }
46. app.UseStaticFiles();
47. app.UseCookiePolicy();
48. app.UseAuthentication();
49. app.UseMvc(routes =>
50. {
51. routes.MapRoute(
52. name: "default",
53. template: "{controller=Home}/{action=Index}/{id?}");
54. });
55. }
56. }
57. }

We can implement Authentication through Login feature. In most of the applications today, Authorization is decided internally based on your role. So, now we are going to create account login and logout feature, so just create one more controller as ‘AccountController.cs’ inside the controllers folder and add two action methods, one for rendering the Login View and  the other one for posting user credentials data for logging in to the system. Here is the code for AccountController where we have implemented Login functionality.

1. using System;
2. using System.Collections.Generic;
3. using System.Linq;
4. using System.Security.Claims;
5. using System.Threading.Tasks;
6. using Microsoft.AspNetCore.Authentication;
7. using Microsoft.AspNetCore.Authentication.Cookies;
8. using Microsoft.AspNetCore.Mvc;
10. namespace CookieDemo.Controllers
11. {
12. **public** **class** AccountController : Controller
13. {
14. **public** IActionResult Login()
15. {
16. **return** View();
17. }
19. [HttpPost]
20. **public** IActionResult Login(string userName, string password)
21. {
22. **if**(!string.IsNullOrEmpty(userName) && string.IsNullOrEmpty(password))
23. {
24. **return** RedirectToAction("Login");
25. }
27. //Check the user name and password
28. //Here can be implemented checking logic from the database
30. **if**(userName=="Admin" && password == "password"){
32. //Create the identity for the user
33. **var** identity = **new** ClaimsIdentity(**new**[] {
34. **new** Claim(ClaimTypes.Name, userName)
35. }, CookieAuthenticationDefaults.AuthenticationScheme);
37. **var** principal = **new** ClaimsPrincipal(identity);
39. **var** login = HttpContext.SignInAsync(CookieAuthenticationDefaults.AuthenticationScheme, principal);
41. **return** RedirectToAction("Index", "Home");
42. }
44. **return** View();
45. }
46. }
47. }

The first login action method is rendering the UI for login page and once you fill the data required for Login as username and password then the second action method as Login will work and send the Post request to the server.

In this method, first, we will check whether username and password should not be empty then we will validate the username and password. Here, in this demonstration, we are checking the username and password with some dummy data. You can implement database login instead of this.

After validating the user information, if everything is correct then we create Identity for that user and create the cookie information for it. Based on this principal data, we try to Sign In using a generic function called "SignInAsync" and if everything goes in the right direction then we redirect to the Home page.

Now, let's create the Login view page from where we can give the functionality to the user to enter the username and password. So, right click on the Login action method and add view without a model. It will automatically create the Account folder inside the Views under that will create “login.cshtml” file. Just open it and create a container and add a form tag along with two textboxes for entering the username and password. Apart from this, create two separate buttons as “Submit” and “Reset”. Once you fill the data and click on the submit button, it will call to Login action method defined in Account Controller using POST call. So, modify the code of “login.cshtml” as follows.

1. @{
2. ViewData["Title"] = "Login";
3. }
5. <div **class**="container">
6. <div **class**="row">
7. <div **class**="col-md-3">
8. <h2><strong>Login Page </strong></h2><br />
9. <form asp-action="login" method="post">
10. <div **class**="form-group">
11. <label>User Name</label>
12. <input type="text" **class**="form-control" id="userName" name="userName" placeholder="Enter user name">
13. </div>
14. <div **class**="form-group">
15. <label>Password</label>
16. <input type="password" **class**="form-control" name="password" id="password" placeholder="Password">
17. </div>
18. <div **class**="form-check">
19. <button **class**="btn btn-info" type="reset">Reset</button>
20. <button type="submit" **class**="btn btn-primary">Submit</button>
21. </div>
22. </form>
23. </div>
24. </div>
25. </div>

So far we have implemented the Cookie-based Authentication functionality in Asp.Net Core MVC project. But what about Authorization. Authorization means, providing access to the authenticated user to access a resource based on role.

So, let's first understand how we can implement the Authorization in Asp.Net Core MVC. For now, if you will try to access the HOME page without sign in, you can access it. So, let’s prevent the anonymous user from accessing the HOME page directly, if someone wants to access the HOME page then they should have to go through the Authentication process and then they will be able to access it.

So, to accomplish this, let’s open the Home Controller and put the [Authorize] attribute just above to controller. You can place it at action level but here we would like to block the whole home controller functionality and if we want to access, just go and log in. So, just do something like below.

1. using CookieDemo.Models;
2. using Microsoft.AspNetCore.Authorization;
3. using Microsoft.AspNetCore.Mvc;
4. using System.Diagnostics;
6. namespace CookieDemo.Controllers
7. {
9. [Authorize]
10. **public** **class** HomeController : Controller
11. {
12. **public** IActionResult Index()
13. {
14. **return** View();
15. }
17. **public** IActionResult About()
18. {
19. ViewData["Message"] = "Your application description page.";
21. **return** View();
22. }
24. **public** IActionResult Contact()
25. {
26. ViewData["Message"] = "Your contact page.";
28. **return** View();
29. }
31. **public** IActionResult Privacy()
32. {
33. **return** View();
34. }
36. [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = **true**)]
37. **public** IActionResult Error()
38. {
39. **return** View(**new** ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });
40. }
41. }
42. }

**Note**Be sure you have cleared all cookies which have been  created based on your previous login. If you will not do this, you will be accessing the HOME page, it is because authenticated user cookie is available in browser memory.

So, let's check how it works. Run the application and try to access the Home page. You will see here that your application automatically redirects to Login page. Now let's try to provide the user information as username = "Admin" and password =" password". Once you will pass the correct credentials and login then you will redirect to HOME page. So, let'sadd the feature that shows the logged in username along with a logout button. If you click to the log out button, your cookie value will be deleted and you will redirect to login page.

So, let's open the Account Controller and add the following logout action method.

1. [HttpPost]
2. **public** IActionResult Logout()
3. {
4. **var** login = HttpContext.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);
5. **return** RedirectToAction("Login");
6. }

And now open the Index.cshtml file from the Home folder inside the Views and modify the code as follows. Here, first of all, we are trying to show the Logged In username using @User.Identity.Name and apart from this adding a link for logout.

1. @{
2. ViewData["Title"] = "Home Page";
3. }
5. <div **class**="container">
6. <div **class**="row">
7. <div **class**="col-md-12">
8. <h2><strong>Home Page </strong></h2><br /><br />
9. Hello @User.Identity.Name
10. <a asp-action="logout" asp-controller="account">
11. Logout
12. </a>
13. <br />
14. <br />
15. <h4>Welcome to Asp.Net Core Authentication and Authorization Demo!!</h4>
16. </div>
17. </div>
18. </div>

So far, we are able to understand how to implement Authentication in Asp.Net Core MVC and how to implement Authorization and give access to validate the users. Now, let's understand how to work with multiple roles. Here we are doing everything manually with some static value, but you can change the logic and connect to the database for validating the user. So, just modify the Login method as follows where we are providing two different kinds of roles; one is Admin role and another is User role. Based on these roles, we will provide access to some of the pages.

1. [HttpPost]
2. **public** IActionResult Login(string userName, string password)
3. {
4. **if** (!string.IsNullOrEmpty(userName) && string.IsNullOrEmpty(password))
5. {
6. **return** RedirectToAction("Login");
7. }
9. //Check the user name and password
10. //Here can be implemented checking logic from the database
11. ClaimsIdentity identity = **null**;
12. bool isAuthenticated = **false**;
14. **if** (userName == "Admin" && password == "password")
15. {
17. //Create the identity for the user
18. identity = **new** ClaimsIdentity(**new**[] {
19. **new** Claim(ClaimTypes.Name, userName),
20. **new** Claim(ClaimTypes.Role, "Admin")
21. }, CookieAuthenticationDefaults.AuthenticationScheme);
23. isAuthenticated = **true**;
24. }
26. **if** (userName == "Mukesh" && password == "password")
27. {
28. //Create the identity for the user
29. identity = **new** ClaimsIdentity(**new**[] {
30. **new** Claim(ClaimTypes.Name, userName),
31. **new** Claim(ClaimTypes.Role, "User")
32. }, CookieAuthenticationDefaults.AuthenticationScheme);
34. isAuthenticated = **true**;
35. }
37. **if** (isAuthenticated)
38. {
39. **var** principal = **new** ClaimsPrincipal(identity);
41. **var** login = HttpContext.SignInAsync(CookieAuthenticationDefaults.AuthenticationScheme, principal);
43. **return** RedirectToAction("Index", "Home");
44. }
45. **return** View();
46. }

Now let's move to HomeController and remove the [Authorize] attribute from the class level and put it in action level as follows. Here we have two different action methods which point to two different views. One is pointing to Index view and another one is pointing to the Setting page. Index page can be accessible to both type of roles, either it is Admin or User but the Setting page can be accessed only by Admin role.

1. **public** **class** HomeController : Controller
2. {
3. [Authorize(Roles ="Admin, User")]
4. **public** IActionResult Index()
5. {
6. **return** View();
7. }
9. [Authorize(Roles ="Admin")]
10. **public** IActionResult Setting()
11. {
12. **return** View();
14. }

Now modify the Index.cshtml file and add one thing as Role, just modify the code as follows.

1. @{
2. ViewData["Title"] = "Setting Page";
3. }
5. <div **class**="container">
6. <div **class**="row">
7. <div **class**="col-md-12">
8. <h2><strong>Setting Page </strong></h2><br /><br />
9. Hello @User.Identity.Name !, Role @User.FindFirst(claim=>claim.Type==System.Security.Claims.ClaimTypes.Role)?.Value
10. <a asp-action="logout" asp-controller="account">
11. Logout
12. </a>
13. <br />
14. <br />
15. <h4>Admin role user can only access **this** page!!</h4>
16. </div>
17. </div>
18. </div>

Now, we have added everything and it's time to run the application. So, just press F5 and it will run your application. First, go to "Login" page and login with "User" role.