Custom user roles and role-based authorization in ASP.NET core


Temi Lajumoke Ambassador

ASP.NET | ASP.NET MVC | Identity management | Role-based access control | Roles | Security

I've recently been trying to contribute to the developer community, in more ways than I previously have. One of the areas I decided to delve into, is spending more time on tech forums and resources like StackOverflow. Well not for asking questions or checking for answers, but more for answering as many questions as I possibly can. The aim is to provide answers primarily to questions budding developers are asking. so Let's hope I do a lot of that as time goes on. Hey, I might even be answering most the questions in detail here.

Lately, I have stumbled upon a lot of questions about role-based authorization and setting up custom user roles. I see a good number of developers have a bit of trouble setting this up on ASP.NET core, and the question keeps reoccurring. I'm going to try to explain it as simple as possible. So wish

me luck!

Introduction

One of the most important features in building robust web applications is ensuring that users have access to just the areas or web pages they need. This is usually achieved by creating different user roles and assigning users to the roles created. These roles define what a user can and cannot do in the web application. In this article, we'll walk through creating custom user roles in ASP.NET core and authorizing users based on these roles. We'll also take a look at how can grant access to or prevent a specific class of users from viewing certain pages or performing certain tasks in an ASP.NET core web application.

A bit about Identity and user roles

ASP.NET Core Identity is the membership or identity management system shipped with the ASP.NET Core web development stack, for building web applications. It includes membership, login, and management of user data. ASP.NET Core Identity allows you to add authentication features and customize data about the logged in user in your application. It works with OWIN and also has support for OAuth, and external login providers such as Facebook, Microsoft, and Google.In ASP.Net and ASP.NET core, when an identity is created, it may belong to one or more roles. For instance, a web application may have an administrator, a manager and of course, the user/member role, and any user can be assigned to these roles.

Creating user configuration settings

First off we'll be creating a super admin that can manage our web app. On application startup, we'll check if our super admin user has been created, and if not, we'll create one. As a standard practice, the super admin's account credentials should be read from configuration settings, as opposed to hard-coding it directly in the Startup Class, so we'll demo that in this article. By default, configuration settings are stored in the

"appsettings.json" file , or the "appsettings.{EnvironmentName}.json". The EnvironmentName could be Production, Development or any other environment you please. The configuration consists of name-value pairs and we basically create a "UserSettings" section with the name-value pairs we need; such as the user's email and password. Like so.

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Data Source=WebApplication.db"
  },
  "UserSettings":{
    "UserEmail": "johndoe@email.com",
    "UserPassword": "P@ssw0rd"
  }
}
```

Creating custom roles

To create the custom roles, we create an array of strings, containing each of the roles we intend to have in our application, then we create the roles by iterating through the elements in our string array, passing each element to the method in RoleManager class. This automatically creates the roles and seeds them to the database.

Next, we create our super admin user, reading the configuration from the configuration file we previously created, and then we assign the user to our preferred role. in this case, the "Admin" role. Like so

```
private async Task CreateRoles(IServiceProvider serviceProvider)
        {
            //adding custom roles
            var RoleManager = serviceProvider.GetRequiredService<RoleManager<IdentityRole>>();
            var UserManager = serviceProvider.GetRequiredService<UserManager<ApplicationUser>>();
```

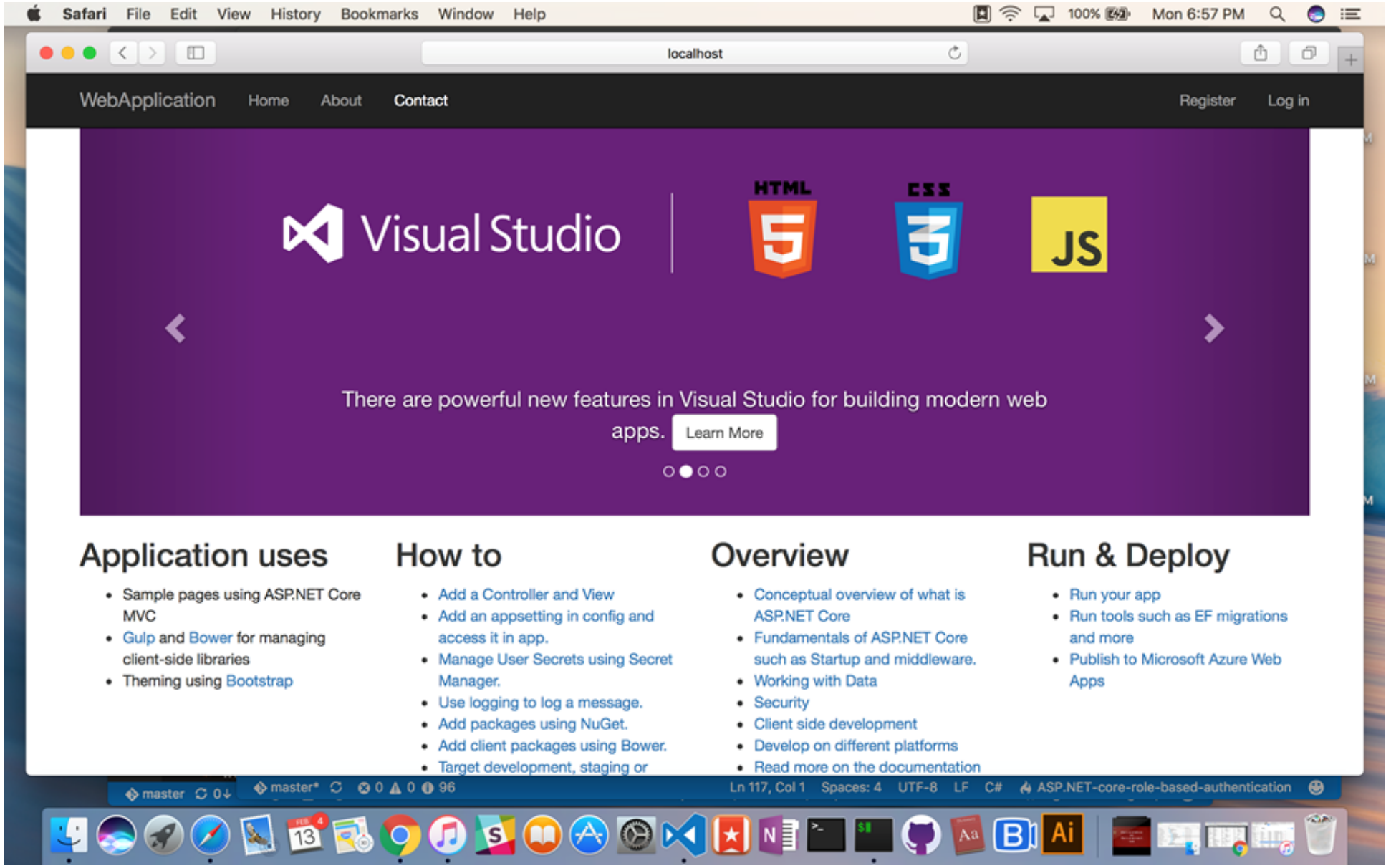
```
6             string[] roleNames = { "Admin", "Manager", "Member" };
7             IdentityResult roleResult;
8
9             foreach (var roleName in roleNames)
10            {
11                //creating the roles and seeding them to the database
12                var roleExist = await RoleManager.RoleExistsAsync(roleName);
13                if (!roleExist)
14                {
15                    roleResult = await RoleManager.CreateAsync(new IdentityRole(roleName));
16                }
17            }
18
19            //creating a super user who could maintain the web app
20            var poweruser = new ApplicationUser
21            {
22                UserName = Configuration.GetSection("UserSettings")["UserEmail"],
23                Email = Configuration.GetSection("UserSettings")["UserEmail"]
24            };
25
26            string UserPassword = Configuration.GetSection("UserSettings")["UserPassword"];
27            var _user = await UserManager.FindByEmailAsync(Configuration.GetSection("UserSettings")
28
29            if(_user == null)
30            {
31                    var createPowerUser = await UserManager.CreateAsync(poweruser, UserPassword);
32                    if (createPowerUser.Succeeded)
33                    {
```

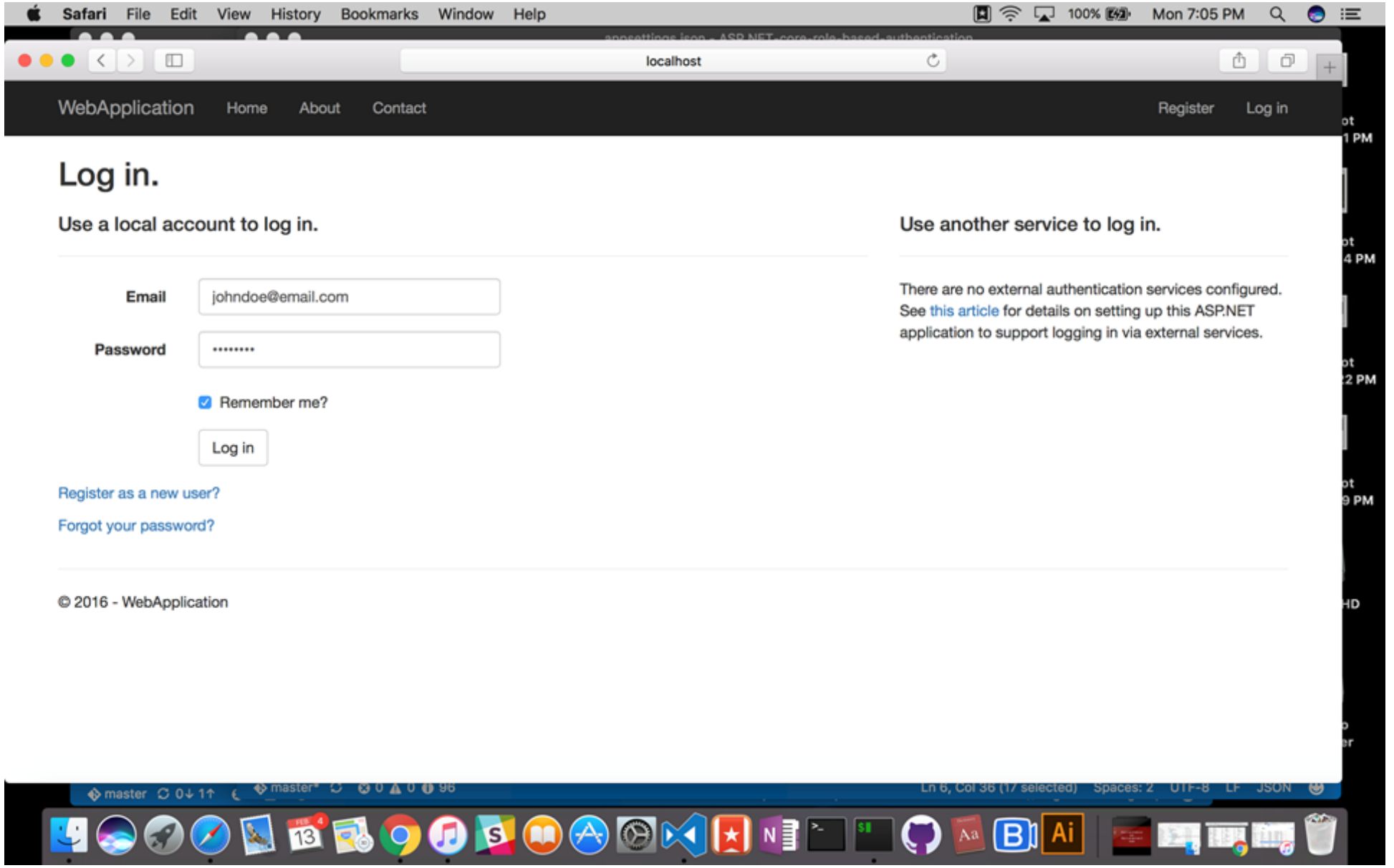
```
                    //here we tie the new user to the "Admin" role
                    await UserManager.AddToRoleAsync(poweruser, "Admin");

                }
            }
        }
```

And then, as you probably guessed, we call our CreateRoles() method from the Configure method.

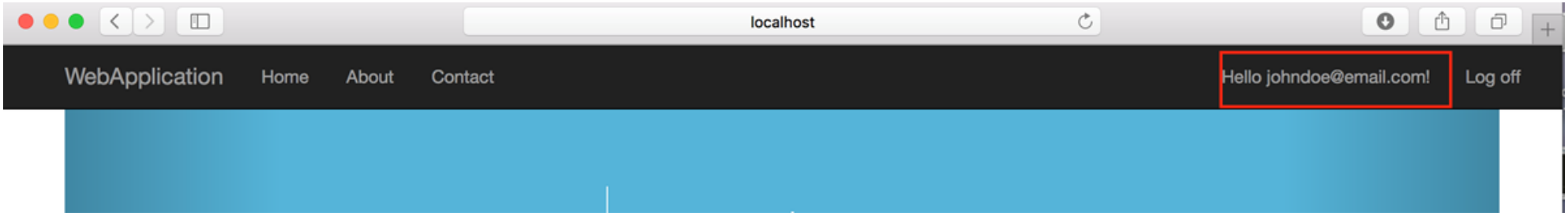
```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactor
{
  loggerFactory.AddConsole(Configuration.GetSection("Logging"));
  loggerFactory.AddDebug();

  app.UseStaticFiles();

  app.UseIdentity();
  app.UseMvc(routes =>
            {
    routes.MapRoute(
      name: "default",
      template: "{controller=Home}/{action=Index}/{id?}");
  });
  CreateRoles(serviceProvider).Wait();
}
```

et voila! we have just created custom roles, created a super admin user, and assigned a role to our super admin user in our web app. So, we can successfully run our application.





et voila, we have successfully created a new user, assigned a role to the user, and signed in with the account.



However, when you sign in, you may not see any major changes. So let's put our user roles to practice with role-base authentication.

Role based authentication

1. using role checks

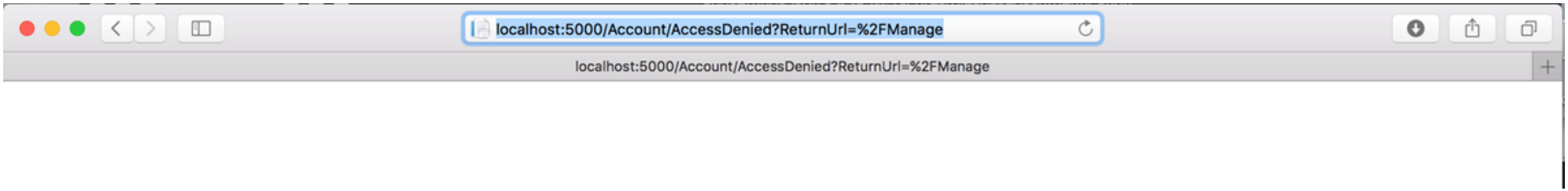
Perhaps the simplest way to authorize users based on roles in your ASP.NET core application is to use role checks. Role checks are embedded within your code, against a controller, or an action in a controller. Role checks simply specify the roles the current user must be a member of, to be able to access the requested page or to perform a particular action.

For instance, the following code would prevent any other user, apart from the those assigned "Manager" role from accessing any actions on the ManageController.

```
1 [Authorize(Roles="Manager")]
2 public class ManageController : Controller
3 {
4     //....
5 }
```

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using WebApplication.Models;
using WebApplication.Models.ManageViewModels;
using WebApplication.Services;

namespace WebApplication.Controllers
{
    [Authorize(Roles="Manager")]
    1 reference
    public class ManageController : Controller
    {
        19 references
        private readonly UserManager<ApplicationUser> _userManager;
        12 references
        private readonly SignInManager<ApplicationUser> _signInManager;
        1 reference
        private readonly IEmailSender _emailSender;
        2 references
        private readonly ISmsSender _smsSender;
        4 references
        private readonly ILogger _logger;

        0 references
        public ManageController(
        UserManager<ApplicationUser> userManager,
        SignInManager<ApplicationUser> signInManager,
        IEmailSender emailSender,
        ISmsSender smsSender,
        ILoggerFactory loggerFactory)
        {
            _userManager = userManager;
            _signInManager = signInManager;
```

If we run our application and navigate to any action in the ManageController, we get an "Access Denied" error because our super admin user is assigned only to the "Admin" role. Like so.



Of course, in an ideal situation, we may want to restrict access to just some actions in a controller, in this case, we can have role authorization checks against only the actions we intend to restrict users in other roles from.

We can also apply multiple role authorization attributes, in which an accessing user must be a member of the roles specified; the following sample requires that a user must be a member of the Admin and Manager roles.

```
1 [Authorize(Roles="Admin, Manager")]
2 public IActionResult Index()
3 {
4  /*
5  .....
6  */
7 }
```

2. using policy based role checks

With ASP.NET Core 1.0 came a new authorization abstraction called authorization policies. Using this feature, you can define authorization policies, and then apply the defined policies to controller actions. In the earlier example, we authorized users using the role checks, however, this has some drawbacks compared to using authorization policies. One major drawback is that roles are just a type of claim, and so using role checks, we check for only one type of claim. Policy based checks, on the other hand, can check for any type of claim. For instance (policy.RequireUserName("JohnDoe")); in this case, the current user must posses the user name "johnDoe" to access the requested page or action.

Let's say, hypothetically, that we have a controller we only want to give our super admin access to. We can apply an authorization policy using the Policy property on the Authorize attribute. To use policy based role checks, we register the policy at startup, normally in the ConfigureServices() method in our *Startup.cs* file.

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

    services.AddAuthorization(options =>
    {
        options.AddPolicy("RequireAdminRole", policy => policy.RequireRole("Admin"));
    });
}
```

```
[Authorize(Policy = "RequireAdminRole")]
public class AdminController : Controller
{
```

```
4     public IActionResult Index()
5     {
6       return View();
7     }
8 }
```

UPDATE

I've received a good number of emails and comments requesting for a dotnet core 2.0 version. Thus, I've updated the article and the GitHub repo to support both dotnet core 1.x and 2.x.

Setting default user roles

To set users to a single default role or multiple roles, we simply add them to these roles upon registration. In the Register action of the Account controller, Like so

```
1 [HttpPost]
2 [AllowAnonymous]
3 [ValidateAntiForgeryToken]
4 public async Task<IActionResult> Register(RegisterViewModel model, string returnUrl = null)
5 {
6     ViewData["ReturnUrl"] = returnUrl;
7     if (ModelState.IsValid)
8     {
9         var user = new ApplicationUser { UserName = model.Email, Email = model.Email };
10        var result = await _userManager.CreateAsync(user, model.Password);
11        if (result.Succeeded)
```

```
        {
            _logger.LogInformation("User created a new account with password.");

            // Add a user to the default role, or any role you prefer here
            await _userManager.AddToRoleAsync(user, "Member");

            await _signInManager.SignInAsync(user, isPersistent: false);
            _logger.LogInformation("User created a new account with password.");
            return RedirectToLocal(returnUrl);
        }
        AddErrors(result);
    }
    // If we got this far, something failed, redisplay form
    return View(model);
}
```

Restricting access to sections of the view

To restrict users from seeing navigation links, menu options, images, or anything really, in a razor view, ASP.NET core provides a simple "IsInRole()" Method. Which takes a role name string value, and checks if the current user is in that role

```
@if (User.IsInRole("Admin"))
{
  <li><a asp-area="" asp-controller="Admin" asp-action="Index">Admin</a></li>
}
else
{
```

```
7     <li><a asp-area="" asp-controller="Members" asp-action="Index">Admin</a></li>
8 }
9
```

And there you have it! You should now understand role-based authorization in ASP.NET/ ASP.NET core, and how to create user roles.

The entire code sample for this application can be found on my Github here. Good luck!

Share

Conversation

ABOUT GOOROO

Our story

Official Blog

Ambassador Program

Microsoft Alliance

Working @ Gooroo

Corporate

Support & FAQs

Contact Us

LEGALS

Privacy Policy

Terms of Use

Publishing Guidelines