

## Exercise 16a : Security Basics

### Objective

In this exercise you will authenticate your users, and then secure your application using the built-in authorisation functionality. You will explore how to use authorisation policies, and then finally you will customise the authentication system by adding extra data for each user.

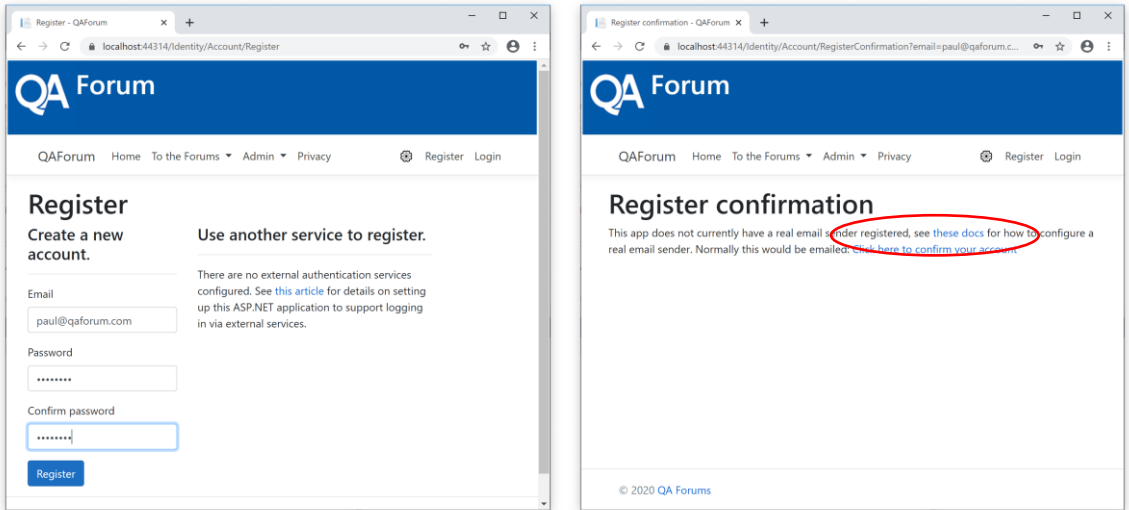
This exercise will take around 30 minutes.

### Referenced material

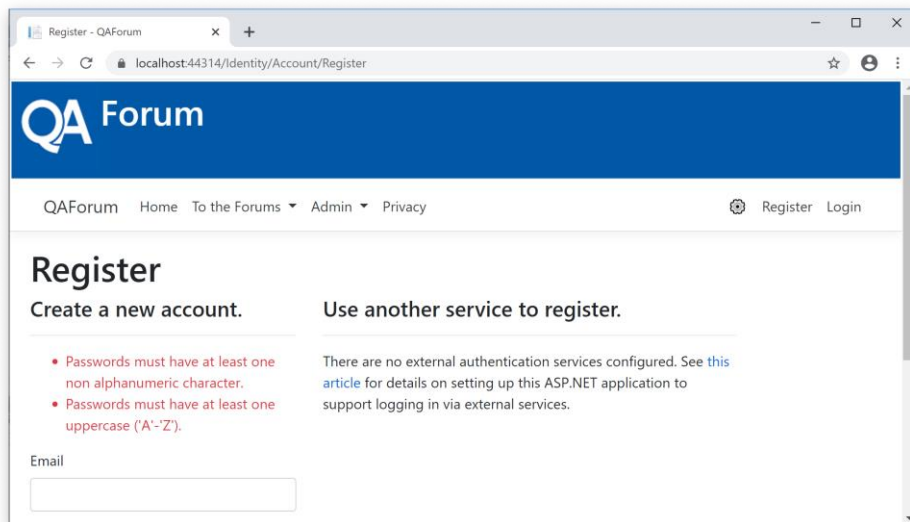
This exercise is based on material from the chapter "Security".

### Adding Basic Authorization

<ul style="list-style-type: none"><li>•</li></ul>	<p>Open the 'Begin' solution in Visual Studio and compile (Shift Ctrl+B). The 'Begin' solution is identical to the 'End' solution from the previous lab.</p>
<ul style="list-style-type: none"><li>•</li></ul>	<p>At the moment, everyone can access every single part of our website. The system has authentication built in (allowing users to identify themselves by logging in), but it does not make use of authorization.</p> <p>Let's start by restricting access to most of the application, so that it can only be used by registered users.</p> <p>Add the following attribute to the PostController class in /Areas/Users/Controllers:</p> <pre>[Area("Users")] [Authorize] public class PostController : Controller {</pre> <p>Do the same for the ThreadController too.</p>
<ul style="list-style-type: none"><li>•</li></ul>	<p>In the ForumController, although we want to restrict access to most of the controller, we would like users to be able to see a list of forums even if they are not signed in. Therefore, in addition to the [Authorize] attribute, there is a further attribute we need to add to allow this:</p> <pre>[Area("Users")] [Authorize] public class ForumController : Controller {     private readonly ForumDbContext context;      public ForumController(ForumDbContext context)     {         this.context = context;     }      // GET: Forum     [AllowAnonymous]     public ActionResult Index()     {</pre>
<ul style="list-style-type: none"><li>•</li></ul>	<p>Let's test our changes. Run the program, and ensure that you can still see the list</p>

	<p>of forums, but you're not able to see any of the other forum, thread or post pages. When you try to visit any of these disallowed pages, you are instead sent to the logon page.</p>
<ul style="list-style-type: none"> <li>Click the Register button in your application's navigation bar. Create a new user – enter any email address you like, and a password you will remember (we recommend using Pa\$\$w0rd as the password, because it meets all the password requirements and is easy to remember, but this is not essential).</li> </ul> <p>After you submit your registration details, you need to “confirm your email”. But since we haven't built an email confirmation system yet, instead you are shown a screen with a link to click which simulates confirming your email</p>	 <p>Once you complete this process, click Login, and re-enter the same username and password. Now, confirm that you can access the whole of the website once again.</p>
<ul style="list-style-type: none"> <li>Next, we're going to add authorisation to the Admin section of the website.</li> </ul> <p>Although it's possible to use the [Authorize] attribute on individual page models, the need to remember to do this for every single Razor page means that it's not the most common way of authorization access to Razor pages. Instead, it's common to create a convention for authorisation.</p> <p>So, open Startup.cs, and locate the line in the ConfigureServices() method which currently reads services.AddRazorPages(); Modify the line as shown here:</p>	<pre>services.AddRazorPages     (options =&gt; options.Conventions.AuthorizeAreaFolder("Admin", "/"));</pre>
<ul style="list-style-type: none"> <li>Check that this has worked as expected – you are now unable to view the two Admin pages unless you're logged in.</li> </ul>	
<ul style="list-style-type: none"> <li>Users are reporting that it's hard to find a password which meets the password restrictions. See if you agree – try to create a new user with the password</li> </ul>	

“password0”:



Let's ease those restrictions a little.

- Add the following into ConfigureServices() in Startup.cs:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<IdentityOptions>(options =>
    {
        options.Password.RequireNonAlphanumeric = false;
        options.Password.RequireUppercase = false;
        options.Password.RequireLowercase = false;
    });
}
```

Then, try adding a user with the same password as previous (“password0”), and confirm that it now works.

- One last change to make in the section: Look in /Data/ApplicationDbContext.cs – notice how the ApplicationDbContext class inherits from IdentityDbContext:

```
public class ApplicationDbContext : IdentityDbContext
```

This means that ApplicationDbContext will be a database context that contains all the Identity data. The class which contains the user data is called IdentityUser – this is the default class that is used for this purpose.

Although we're not quite ready to customise it yet, later on we will want to customise the data we store about each user. To do this, we need to make sure the user data is stored in our own class, not one which we don't have the source code for. So add the following class to the Data folder:

```
public class ApplicationUser : IdentityUser
{
}
}
```

And modify the ApplicationDbContext so that it uses our new class to hold its user data, instead of the built-in class:

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
```

	{
<ul style="list-style-type: none"> <li>Now, there are a couple of other places where we need to change the user data type.</li> </ul> <p>In Startup.cs, the line which adds the default identity needs changing:</p> <pre>services.AddDefaultIdentity&lt;ApplicationUser&gt;     (options =&gt; options.SignIn.RequireConfirmedAccount = true)     .AddEntityFrameworkStores&lt;ApplicationDbContext&gt;();</pre> <p>And in /Views/Shared/_LoginPartial.cshtml:</p> <pre>@using Microsoft.AspNetCore.Identity @using QAForum.Data @Inject SignInManager&lt;ApplicationUser&gt; SignInManager @Inject UserManager&lt;ApplicationUser&gt; UserManager</pre> <p>This change won't make any visible difference, but it will make later parts of this lab easier having done it early on. Run the program and check that everything still works.</p>	

## Authorisation Using Policies

Currently, any user has access to the admin pages. We need to change this so that only administrators have access.

To enable this, we are going to create one user who is an administrator. We will use the identity system to add a "Claim" to the user – the claim will indicate that the user is an Administrator.

Then, we will create a "Policy" – the policy being that the user must have the Administrator Claim. And we will then apply that Policy to the Admin section of the website.

<ul style="list-style-type: none"> <li>There is no built-in mechanism for seeding users. One common way to seed users is to create a method to add the users to the user manager, and to call that method from the ConfigureServices() method so that it gets run once when the program starts running.</li> </ul> <p>Add a class to the Data folder called Claims. This class will contain constants that can be used throughout the program:</p> <pre>public static class Claims {     public const string ADMIN_CLAIM = "Administrator"; }</pre>	
<ul style="list-style-type: none"> <li>Add the following class to the Data folder:</li> </ul> <pre>public class UserSeeder {     private readonly UserManager&lt;ApplicationUser&gt; userManager;</pre>	

```

public UserSeeder(UserManager<ApplicationUser> userManager)
{
    this.userManager = userManager;
}

public async Task SeedUsers()
{
    const string email = "admin@qaforum.com";
    const string password = "Pa$$w0rd";

    var existingUser = await userManager.FindByNameAsync(email);
    if (existingUser != null)
    {
        // The user already exists
        return;
    }

    var user = new ApplicationUser
    {
        UserName = email,
        Email = email,
        EmailConfirmed = true
    };
    var result = await userManager.CreateAsync(user, password);
    if (result != IdentityResult.Success)
    {
        throw new ApplicationException("Failed to create admin user");
    }

    result = await userManager.AddClaimAsync
        (user, new Claim(Claims.ADMIN_CLAIM, ""));
    if (result != IdentityResult.Success)
    {
        throw new ApplicationException("Failed to add Administrator claim");
    }
}
}

```

This class is responsible for seeding the admin user, if it doesn't already exist, and adding the Administrator claim.

- Register the class with the service provider by adding the following into Startup.cs:

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddScoped<UserSeeder>();
}

```

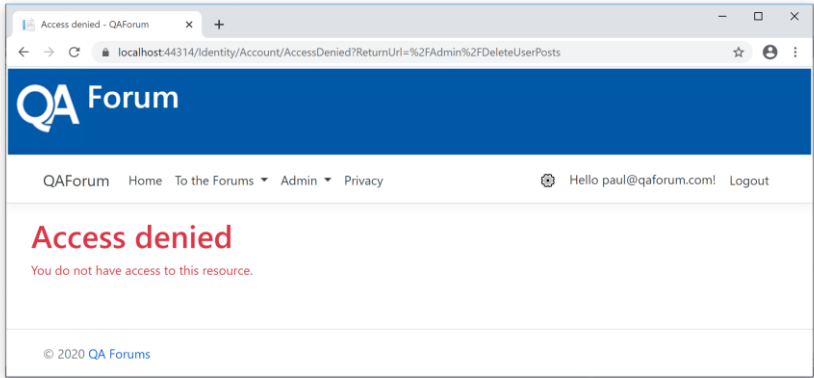
- Modify the Main() method in Program.cs to seed the data:

```

public static async Task Main(string[] args)
{
    IHost host = CreateHostBuilder(args).Build();

    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;
        var seeder = services.GetRequiredService<UserSeeder>();
    }
}

```

	<pre>         await seeder.SeedUsers();     }      host.Run(); } </pre>
<ul style="list-style-type: none"> <li>Policies are created in the Startup.cs file, in the ConfigureServices() method. Add the following to that method:</li> </ul>	<pre> public void ConfigureServices(IServiceCollection services) {     services.AddAuthorization(options =&gt;     {         options.AddPolicy("AdminOnly",             policy =&gt; policy.RequireClaim(Claims.ADMIN_CLAIM));     }); } </pre>
<ul style="list-style-type: none"> <li>Now, modify the Razor pages authorisation convention that we created earlier in the lab:</li> </ul>	<pre> services.AddRazorPages(options =&gt; options.Conventions.AuthorizeAreaFolder     ("Admin", "/", "AdminOnly")); </pre>
<ul style="list-style-type: none"> <li>Test your program again. Visit one of the admin pages when you are not logged in. Ensure that you get redirected to the login page. Log in as a non-administrator, and you should now see the following error page:</li> </ul>	 <p>Finally, log in as the admin user. The user id is admin@qaforum.com and the password is Pa\$\$w0rd (this is set in UserSeeder.cs). Now, you should be able to visit the admin pages.</p>
<ul style="list-style-type: none"> <li>Although everything is working as intended, it's not great to show menu options to the user that they can't access and are never going to be allowed to access. So let's remove the Admin options from the menu, unless the user is an administrator.</li> </ul> <p>We need to make these changes in the _Layout.cshtml file. At the top of the file:</p>	<pre> @using QAForum.Data @using Microsoft.AspNetCore.Mvc.Razor @using Microsoft.AspNetCore.Mvc.Infrastructure </pre>

And then, when we build the navigation bar:

```
@if (User.Identity.IsAuthenticated &&
    User.HasClaim(claim => claim.Type == Claims.ADMIN_CLAIM))
{
    <li class="dropdown">
        <a class="nav-link text-dark dropdown-toggle" href="#"
            data-toggle="dropdown" data-target="#admin-dropdown">
            Admin
        </a>
        <div class="dropdown-menu" id="admin-dropdown">
            <a class="nav-link text-dark dropdown-item" asp-area="Admin"
                asp-page="/DeleteUserPosts">Delete User Posts</a>
            <a class="nav-link text-dark dropdown-item" asp-area="Admin"
                asp-page="/PostsByDate">Count Posts</a>
        </div>
    </li>
}
```

Now, check that the Admin items do not appear on the navigation bar when logged out, nor when logged in as a non-admin user, but that they do appear when logged in as an admin user (and that they work, too!)

- The claims-based policy that we used to confirm administrator privileges took only one line to add to the service provider, because it's a common requirement that comes built into Asp.Net.

Sometimes, we need to base our policies on requirements that we build ourselves. In this case, we need to have a class which represents the requirements, and a class that represents a handler for that requirement.

Let's add a restriction on the creation of posts, so that posts can only be created on weekdays.

- Add a folder to the root of the project, called AuthorizationRequirements. Into that folder, add a class called DaysOfWeekRequirement. Include the enum in the same file as the class, since it's very closely related to the class:

```
[Flags]
public enum DaysOfWeek
{
    Sunday = 1,
    Monday = 2,
    Tuesday = 4,
    Wednesday = 8,
    Thursday = 16,
    Friday = 32,
    Saturday = 64,

    None = 0,

    Weekdays = Monday | Tuesday | Wednesday | Thursday | Friday,
    Weekends = Saturday | Sunday
}

public class DaysOfWeekRequirement:IAuthorizationRequirement
{
}
```

```

public DaysOfWeek DaysOfWeek { get; set; }

public DaysOfWeekRequirement(DaysOfWeek daysOfWeek)
{
    DaysOfWeek = daysOfWeek;
}
}

```

This very simple class allows us to indicate on which days an action is allowed. Note that it implements the `IAuthorizationRequirement` interface.

- In the same folder, add another class to handle the requirement, called `DaysOfWeekHandler`:

```

public class DaysOfWeekHandler : AuthorizationHandler<DaysOfWeekRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        DaysOfWeekRequirement requirement)
    {
        int currentSystemDay = (int)DateTime.Now.DayOfWeek;

        // Convert the System.DayOfWeek (a data type that does
        // not support representing a group of days, only a single
        // day) to our own enum
        DaysOfWeek currentDay = (DaysOfWeek)Math.Pow(2, currentSystemDay);

        // See if today is one of the days that's allowed
        if ((requirement.DaysOfWeek & currentDay) == currentDay)
        {
            context.Succeed(requirement);
        }

        return Task.CompletedTask;
    }
}

```

This class inherits from the abstract generic class `AuthorizationHandler`. It has to override the abstract method `HandleRequirementAsync()`. The return type from this method is a `Task`, because it is common to want to do asynchronous processing inside the method, but in our case, there is no asynchronous processing, so we simply return `Task.CompletedTask`.

- Now, we need to register our handler, and then add a policy based on the handler. In `Startup.cs`, add the following to `ConfigureServices()`:

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IAuthorizationHandler, DaysOfWeekHandler>();

    services.AddAuthorization(options =>
    {
        options.AddPolicy("AdminOnly",
            policy => policy.RequireClaim(Claims.ADMIN_CLAIM));
        options.AddPolicy("WeekdaysOnly",
            policy => policy.Requirements.Add(
                new DaysOfWeekRequirement(DaysOfWeek.Weekdays)));
    });
}

```



	});
<ul style="list-style-type: none"> <li>And the final stage is to apply an [Authorize] attribute where we want the policy applied. In /Areas/Users/Controllers/PostController.cs, add the attribute as shown: <div data-bbox="451 349 1461 808" data-label="Text"> <pre>// GET: Post/Create [Authorize("WeekdaysOnly")] public ActionResult Create() {     return View(PostWriteViewModel.WithThreadSelectListItems(context)); }  // POST: Post/Create [HttpPost] [ValidateAntiForgeryToken] [Authorize("WeekdaysOnly")] public ActionResult Create(PostWriteViewModel viewModel) {     try     {</pre> </div> </li> </ul>	
<ul style="list-style-type: none"> <li>Test your program, by seeing if you can add a post right now. It should be allowed. Then, change your system clock on your computer to be a weekend, and try again. You should find that you are no longer allowed to add a post.  (If the settings on your computer prevent you from changing the system clock, an alternative is to temporarily change the policy to something like this: <div data-bbox="418 1070 1414 1104" data-label="Text"> <pre>policy.Requirements.Add(new DaysOfWeekRequirement(DaysOfWeek.Saturday)</pre> </div> which only allows posts on Saturdays.) </li> </ul>	
<ul style="list-style-type: none"> <li>If you have multiple handlers for a single requirements, only one of the handlers needs to succeed in order for the requirement to be met.  Let's add another handler for the DaysOfWeek requirement, which will allow the requirement to be met if the user is an administrator (i.e. administrators are allowed to post on weekends).  Add the following class to the AuthorizationRequirements folder: <div data-bbox="338 1473 1482 1877" data-label="Text"> <pre>public class DaysOfWeekAdminHandler:AuthorizationHandler&lt;DaysOfWeekRequirement&gt; {     protected override Task HandleRequirementAsync         (AuthorizationHandlerContext context, DaysOfWeekRequirement requirement)     {         if (context.User.HasClaim(claim =&gt; claim.Type == Claims.ADMIN_CLAIM))         {             context.Succeed(requirement);         }          return Task.CompletedTask;     } }</pre> </div> </li> </ul>	
<ul style="list-style-type: none"> <li>Now register the new handler in ConfigureServices(): <div data-bbox="338 1980 1160 2013" data-label="Text"> <pre>public void ConfigureServices(IServiceCollection services)</pre> </div> </li> </ul>	

	<pre> {     services.AddSingleton&lt;IAuthorizationHandler, DaysOfWeekHandler&gt;();     services.AddSingleton&lt;IAuthorizationHandler, DaysOfWeekAdminHandler&gt;();      services.AddAuthorization(options =&gt;     {         options.AddPolicy("AdminOnly",             policy =&gt; policy.RequireClaim(Claims.ADMIN_CLAIM));         options.AddPolicy("WeekdaysOnly",             policy =&gt; policy.Requirements.Add                 (new DaysOfWeekRequirement(DaysOfWeek.Weekdays)));     }); </pre>
<ul style="list-style-type: none"> <li>•</li> </ul>	<p>Ensure that your system clock shows that it is a weekend. Then, check that regular users are not allowed to post, but administrators are.</p>

## Optional Challenge

Now that users have to be logged on before they can use most of the website's functionality, another change that would be useful would be to remove the option for users to specify the "user name" when they add a new post, and instead to make the system always use the user name of the user currently logged in. How could you make that change?

A related change would be to make it so that users (unless they are an administrator) cannot delete other users' posts. Unfortunately, this can't be done using a policy, because requirements handlers run before the route values (such as the post id) have been processed and parsed. Therefore, a requirement like this would need to be done inside the controller's actions, returning Unauthorized() if the user attempts to do something they're not allowed to do. How would you go about making this change?

You now have a good understanding of the fundamentals of how security is implemented in ASP.Net Core! In the next lab you will explore how to customise the security system to suit your exact needs.