

Exercise 16b : Customising Security

Objective

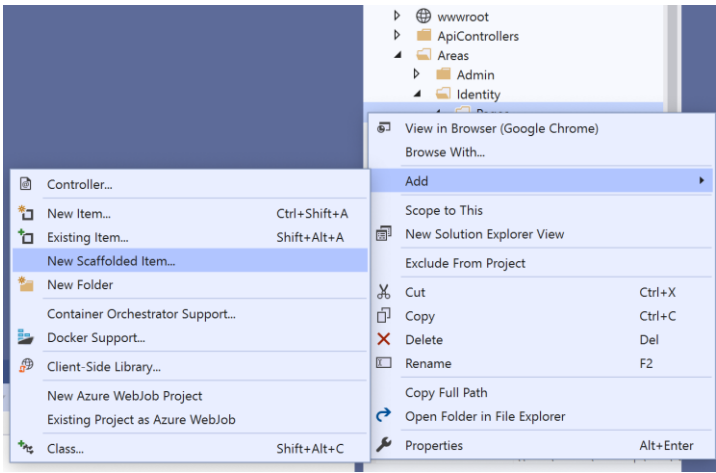
In this exercise you will customise the data which we hold about each user. Then you will configure the identity system to send confirmation emails to new users.

This exercise will take around 30 minutes.

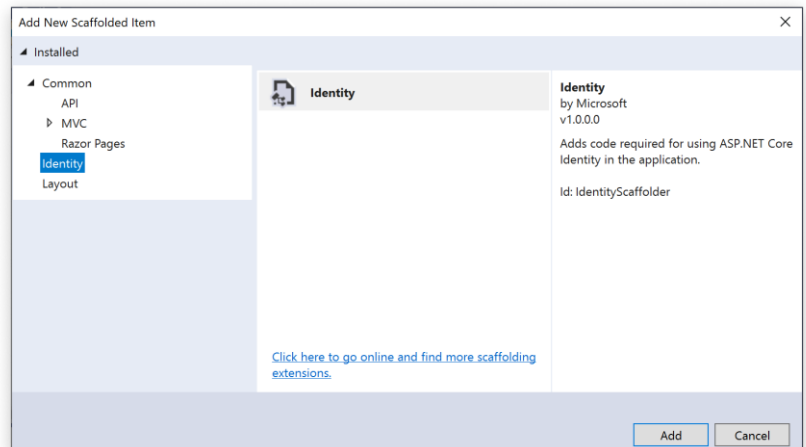
Referenced material

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

Adding Users' First and Last Names

<ul style="list-style-type: none">•	<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>
	<p>Take a look in /Areas/Identity/Pages. You will note that the folder is very nearly empty.</p> <p>There are many, many Razor pages which make up the identity system. But, by default, they are all imported from a library, which means that you can't see their source code.</p> <p>If we want to customise the security system, you will need to scaffold the pages that you want to customise – that is, you will need to add their source code to your project.</p>
<ul style="list-style-type: none">•	<p>Right-click on /Areas/Identity/Pages, and select Add, New Scaffolded Item:</p>  <p>In the dialogue box that appears, choose Identity from the left hand side, ensure that Identity is selected in the middle section (it should be selected by default</p>

because it's the only option), then press Add:

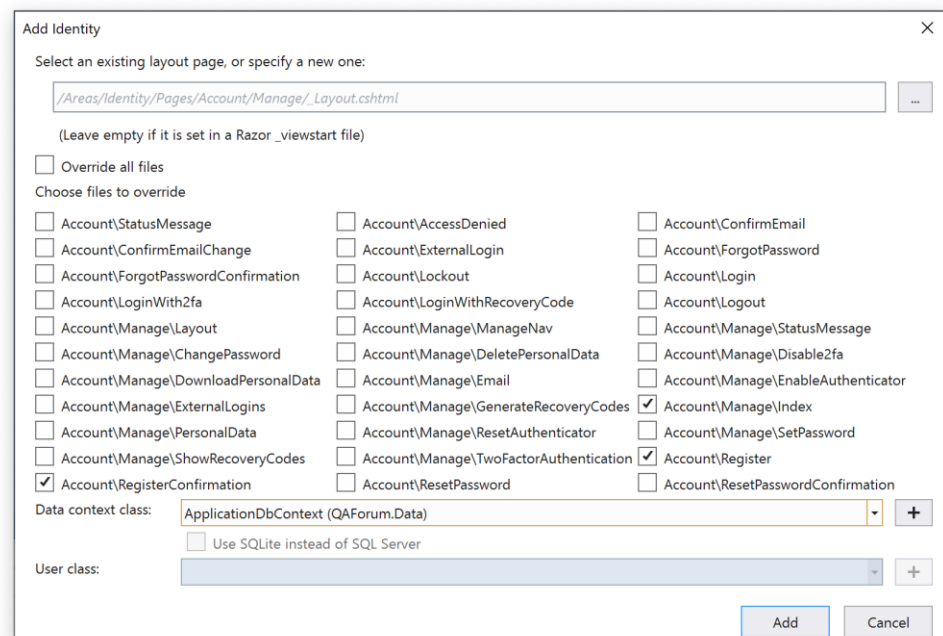


- The next dialogue shows all of the Razor pages that you can scaffold. We need to select all the pages that we need to edit (although if we change our mind later, we can always either add more, or delete any that we didn't need to alter).

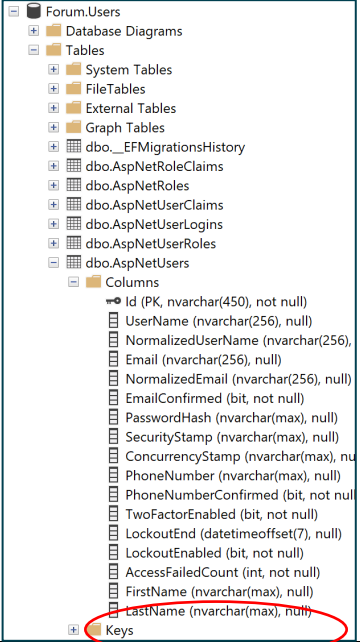
Put ticks in the following boxes:

- Account/Manager/Index
- Account/Register
- Account/RegisterConfirmation

Choose a Data Context Class of ApplicationDbContext. Then press Add.



Check that the requested pages have been added to the Solution Explorer. Ensure that the project builds correctly. (You may need to use Ctrl-dot to add a few namespaces.)

<ul style="list-style-type: none"> • 	<p>We are going to add the ability for users to tell us their name.</p> <p>Open /Data/ApplicationUser, and add the following properties:</p> <pre>public class ApplicationUser : IdentityUser { public string FirstName { get; set; } public string LastName { get; set; } }</pre>
<ul style="list-style-type: none"> • 	<p>Add two new columns to the database.</p> <p>Do this by going to the Tools menu, and choosing Nuget Package Manager, then Package Manager Console.</p> <p>Into the package manager, enter the following two commands:</p> <pre>add-migration AddUserName -context ApplicationDbContext update-database -context ApplicationDbContext</pre> <p>Use Sql Server Management Studio to confirm that the two columns have been added successfully:</p> 
<ul style="list-style-type: none"> • 	<p>Now we have columns in the database to hold the data, the next step is to prompt the user for that data.</p> <p>Open the file /Areas/Identity/Pages/Account/Register.cshtml.cs. Take a moment to look through the code and see how it works.</p> <p>In the file is a nested class called InputModel. This is bound to the Razor page, and will contain the data the user enters when registering for the website. Add two new properties to the class:</p> <pre>public class InputModel { [Required] [EmailAddress] [Display(Name = "Email")] public string Email { get; set; } [Required] [Display(Name = "First Name")] public string FirstName { get; set; } [Required] [Display(Name = "Last Name")] public string LastName { get; set; } }</pre>
<ul style="list-style-type: none"> • 	<p>Next, in the same file, find the OnPostAsync() method. This is the method which registers a user when they submit their details.</p>

Towards the top of that method is a line which creates an ApplicationUser object. (Note that it has picked up the correct class here – our own ApplicationUser, rather than the built-in IdentityUser. That's because when we scaffolded the page, we specified that ApplicationDbContext was the context class. The scaffolder saw this line:

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

From this line, it's possible to scaffold the identity pages to use the correct data type.)

Modify the creation of the ApplicationUser to ensure that we store the first name and last name:

```
public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    returnUrl = returnUrl ?? Url.Content("~/");
    ExternalLogins =
        (await _signInManager.GetExternalAuthenticationSchemesAsync()).ToList();
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser
        {
            UserName = Input.Email,
            Email = Input.Email,
            FirstName = Input.FirstName,
            LastName = Input.LastName
        };
        var result = await _userManager.CreateAsync(user, Input.Password);
    }
}
```

- Next, we need to change /Areas/Identity/Pages/Account/Register.cshtml, which is the Razor page for registering new users. Add two new form groups:

```
<div class="row">
  <div class="col-md-4">
    <form asp-route-returnUrl="@Model.ReturnUrl" method="post">
      <h4>Create a new account.</h4>
      <hr />
      <div asp-validation-summary="All" class="text-danger"></div>
      <div class="form-group">
        <label asp-for="Input.Email"></label>
        <input asp-for="Input.Email" class="form-control" />
        <span asp-validation-for="Input.Email"
              class="text-danger"></span>
      </div>
      <div class="form-group">
        <label asp-for="Input.FirstName"></label>
        <input asp-for="Input.FirstName" class="form-control" />
        <span asp-validation-for="Input.FirstName"
              class="text-danger"></span>
      </div>
      <div class="form-group">
        <label asp-for="Input.LastName"></label>
        <input asp-for="Input.LastName" class="form-control" />
        <span asp-validation-for="Input.LastName"
              class="text-danger"></span>
      </div>
      <div class="form-group">
        <label asp-for="Input.Password"></label>
        <input asp-for="Input.Password" class="form-control" />
      </div>
    </form>
  </div>
</div>
```

	<pre> </div> </pre>
<ul style="list-style-type: none"> Run your program, and register a new user. Now, the system ensures that you enter a first name and a last name when you complete the registration form! 	
<ul style="list-style-type: none"> It's nice that users now enter their name, but it's not that helpful if the name isn't used for anything. Let's greet the user by their name instead of their e-mail address. <p>We will do this in the file <code>/Views/Shared/_LoginPartial.cshtml</code>:</p> <pre> @using Microsoft.AspNetCore.Identity @using QAForum.Data @inject SignInManager<ApplicationUser> SignInManager @inject UserManager<ApplicationUser> UserManager <ul class="navbar-nav"> @if (SignInManager.IsSignedIn(User)) { var appUser = await UserManager.GetUserAsync(User); <li class="nav-item"> Hello @(appUser?.FirstName ?? User.Identity.Name)! } </pre> <p>Remember that, although new users are forced to add a first name and last name, existing users that registered before we made this change won't have their name in the database, so it's important that we fall back to display <code>User.Identity.Name</code> (which is actually the user's email address – that's what the identity system uses as the user name by default) if no first name is found.</p>	
<ul style="list-style-type: none"> Let's also change the Post Details view so that if a post is made by someone who is registered with the system, and has their name recorded, we show their name instead of their e-mail address. (The change will also apply to the Delete view, since that shares a view-model with the Details view.) <p>Make the following changes in the <code>PostDetailsViewModel</code> static method:</p> <pre> public static async Task<PostDetailsViewModel> FromPostAsync (Post post, UserManager<ApplicationUser> userManager) { var user = await userManager.FindByNameAsync(post.UserName); var name = (user?.FirstName != null user?.LastName != null) ? user.FirstName + " " + user.LastName : post.UserName; return new PostDetailsViewModel { PostId = post.PostId, Thread = post.Thread.Title, Title = post.Title, } } </pre>	

	<pre> UserName = name, PostBody = post.PostBody, PostDateTime = post.PostDateTime }; } </pre>
<ul style="list-style-type: none"> Now, change the PostController, so that it has a user manager injected into it: 	<pre> private readonly ForumDbContext context; private readonly IStateRepository state; private readonly UserManager<ApplicationUser> userManager; public PostController(ForumDbContext context, IStateRepository state, UserManager<ApplicationUser> userManager) { this.context = context; this.state = state; this.userManager = userManager; } </pre> <p>Modify the Details action to be asynchronous, and to pass the user manager to the view model:</p> <pre> public async Task<ActionResult> DetailsAsync(int id) { var post = context.Posts.Single(p => p.PostId == id); var recentPosts = state.GetRecentPosts(); recentPosts.AddRecentPost(RecentPostViewModel.FromPost(post)); state.SetRecentPosts(recentPosts); return View(await PostDetailsViewModel.FromPostAsync(post, userManager)); } </pre> <p>Modify both the Delete actions in exactly the same way, and ensure it all compiles.</p>
<ul style="list-style-type: none"> Test your changes. 	<p>Run the application, log on, and go to the Posts list view. Pick one of the posts, and click Details. The user shown on the details screen will probably be an e-mail address. (If not, pick a different post and try again!) Note that e-mail address.</p> <p>Log out of the application, and register a new user. Use the e-mail address that you just picked from the Post. Enter a first name, surname and password for the user, and submit the details. Then, log back into the system (you do not need to log in with the user id you just created – any user id will work). Go back to the Posts list view, view the details of the same post again, and confirm that the user's name is now shown instead of their e-mail address.</p>
<ul style="list-style-type: none"> Next, we're going to make a similar change to the Posts Index view. 	<p>Open the PostViewModel class. We're going to take a different approach here, because we need to get the details of many users (all the users who have made posts) all at the same time. So rather than make many queries to the database, we're going to make just one query, by using the userManager.Users property.</p> <p>Delete the two static methods in PostViewModel, and replace them with this one method. The rather scary-looking piece of Linq is really just doing a left-join</p>

	<p>between posts and users (to get every single post, even those without corresponding users, alongside the post's user if there is one) and then working out the correct name to show.</p> <pre> public static IEnumerable<PostViewModel> FromPosts (IEnumerable<Post> posts, UserManager<ApplicationUser> userManager) { return from post in posts join user in userManager.Users on userManager.NormalizeName(post.UserName) equals user.NormalizedUserName into postsUsers from userOrNull in postsUsers.DefaultIfEmpty() let firstName = userOrNull?.FirstName let lastName = userOrNull?.LastName let name = (firstName != null lastName != null) ? firstName + " " + lastName : post.UserName select new PostViewModel { PostId = post.PostId, Thread = post.Thread.Title, Title = post.Title, UserName = name }; } </pre>
<ul style="list-style-type: none"> • 	<p>Modify the PostController's TagSearch method so that it passes the user manager into the view model:</p> <pre> public ActionResult TagSearch(string tag) { return View(PostViewModel.FromPosts(posts, userManager)); } </pre>
<ul style="list-style-type: none"> • 	<p>Finally, modify the PostListViewComponent.</p> <p>Inject a user manager, in the same way as we did in the PostController.</p> <p>Then, pass the user manager to the PostViewModel.FromPosts method, again mimicking what we just did in the Post Controller.</p>
<ul style="list-style-type: none"> • 	<p>Now, test your changes. Run the program, and check that the Posts list view includes the name of the user you added earlier in this lab, and not their e-mail address. Note a word in the post title, and use the tag search feature to search for the post. Check that it includes the name in this view too.</p>

Here's something to think about. Does it seem wrong that the PostController and the PostListViewComponent both have to have a UserManager injected into them, just so they can pass it onto the view model?

Maybe a better way of designing our view models, instead of having static methods to do the conversions, would be to have a service that did the conversions. Then, the service could be registered in ConfigureServices, and injected into the PostController and PostListViewComponent. If the service

needs a UserManager, it can then be included in its constructor, without having to alter any of the classes that use that service.

That's the benefit of dependency injection. Here, we took a bit of a shortcut must earlier in the project, and it's making life a little more difficult than it needs to be now.

Optional Challenge

If you feel up for a challenge, have a go at the next step. If not, simply move on to the following section.

	<p>It would be nice if users could modify their name after they have registered with the system. Have a look at /Areas/Identity/Pages/Account/Manage/Index.cshtml.cs – the page model for the user management page.</p> <p>Take a moment to understand how the page and its page view work. Then, see if you can modify them so that users can amend their names.</p> <p>Hint: after updating an ApplicationUser, you can use the following line to save the changes to the database:</p> <pre>await _userManager.UpdateAsync(user);</pre> <p>To test your changes, log in and then click on the message that says "Hello name!" or "Hello email!" in the navigation bar.</p>
--	---

Sending Account Confirmation Emails

When a new user signs up to our website, because we don't have an e-mail service registered, we have to show them a fake page which has a link they can click to "confirm" their e-mail.

For a real-life system this is obviously not appropriate, so here we're going to fix that. We're not going to send a real e-mail, but we're going to save the e-mail to the web server's C:\Users\Public folder so that we can see its contents. This folder can be accessed by anyone who is logged on to the computer (including, when we come to deploy our web site, the user under which the web server will run). When implementing a real system, it is recommended to use an e-mail marketing company such as <https://sendgrid.com/>, and to use their API to send e-mails.

<ul style="list-style-type: none">•	Add a folder to the root of the project called Services
<ul style="list-style-type: none">•	<p>In the Services folder, add a new class called DemoEmailSender:</p> <pre>public class DemoEmailSender : IEmailSender { public async Task SendEmailAsync (string email, string subject, string htmlMessage) { var message = @\$"<div>To: {email}</div> <div>Subject: {subject}</div> <hr /> {htmlMessage}"; var filename = @\$"C:\Users\Public\QAForumEmail{"</pre>

	<pre> DateTime.Now.ToString("yyyyMMddHHmmss").html"; using (var writer = new StreamWriter(filename)) { await writer.WriteAsync(message); } } } </pre> <p>This class implements the IEmailSender interface, which is the only requirement it has to be an e-mail sender.</p>
<ul style="list-style-type: none"> Now, register the new class in Startup.cs, in ConfigureServices(): 	<pre> public void ConfigureServices(IServiceCollection services) { services.AddTransient<IEmailSender, DemoEmailSender>(); } </pre> <p>Also, in the same file, note the following line, which requires users to have a confirmed e-mail address. This line is already current, no change is required:</p> <pre> services.AddDefaultIdentity<ApplicationUser> (options => options.SignIn.RequireConfirmedAccount = true) .AddEntityFrameworkStores<ApplicationDbContext>(); </pre>
<ul style="list-style-type: none"> Open /Areas/Identity/Pages/Account/Register.cshtml.cs. Have a look at the OnPostAsync() method, where, about half way down, you'll find the line that sends the e-mail. Modify this line to customise the message a little: 	<pre> await _emailSender.SendEmailAsync(Input.Email, "Confirm your email", \$"Welcome to QA Forums! Please confirm your account by " + \$"clicking here."); if (_userManager.Options.SignIn.RequireConfirmedAccount) { return RedirectToPage("RegisterConfirmation", new { email = Input.Email }); } else </pre>
<ul style="list-style-type: none"> To prevent the user from being shown the temporary page where they can confirm their e-mail address without having received an e-mail, we need to make a change in /Areas/Identity/Pages/Account/RegisterConfirmation.cshtml.cs: 	<pre> Email = email; // Once you add a real email sender, you should remove this code that lets you confirm the account DisplayConfirmAccountLink = false; if (DisplayConfirmAccountLink) { </pre>
<ul style="list-style-type: none"> Now, run the program, and register a new account! 	<p>Once you register, you should find that you can't log in as the new user. However, in the C:\Users\Public folder of your computer is a file starting QAForumEmail... Open this file, and click on the link contained within it. Now you should be able to log in.</p>

Congratulations, you now know how to customise the Identity system to meet your needs!