CHAPTER 9

Creating, Deleting, and Locking Accounts

In this chapter, I create workflows for creating accounts, deleting accounts, and managing account lockouts. As part of the account creation process, I explain how confirmation tokens are used to create workflows that allow the user to confirm their account. Table 9-1 puts the features described in this chapter in context.

Table 9-1. *Putting the API Features for Creating, Deleting and Locking Accounts in Context*

Question	Answer
What are they?	These features are used to create and delete accounts from the user store and to temporarily prevent a user from signing in.
Why are they useful?	Account creation is required to enable users to sign into the application. Account deletion is useful when a user no longer wants or is no longer allowed to use the application. Lockouts prevent a user from signing in following multiple failed attempts or at the action of the administrator.
How are they used?	These features are provided through a combination of user manager and sign-in manager methods and the properties of the user class.
Are there any pitfalls or limitations?	Because the Identity user store is consulted only when the user signs in, the user can continue to use the application even after they have been locked out or their account has been deleted because the authentication cookie issues to their browser remains valid. This can be minimized by enabling validation for the cookie, but this requires additional user store queries.
Are there any alternatives?	No.

Table 9-2 summarizes the chapter.

A. Freeman, *Pro ASP.NET Core Identity*, https://doi.org/10.1007/978-1-4842-6858-2_9

Table 9-2. *Chapter Summary*

Problem	Solution	Listing
Create a new account	Create a new `IdentityUser` object and pass it to the user manager's `CreateAsync` method. Generate a token that can be validated to let the user select a password.	1–9
Confirm new accounts	Generate a token using the `GenerateEmailConfirmationTokenAsync` method and send it to the user. Validate the token and confirm the account with the `ConfirmEmailAsync` method.	10–16
Determine if an account has been confirmed	Read the `IdentityUser.EmailConfirmed` property or call the user manager's `IsEmailConfirmedAsync` method.	17
Locking accounts	Call the user manager's `SetLockoutEnabledAsync` and `SetLockoutEndDateAsync` methods to enable lockouts for an account and to specify the end date for a lockout.	18–20
Force immediate sign-outs	Enable authentication cookie validation.	21–23
Deleting an account	Call the user manager's `DeleteAsync` method.	24–29

Preparing for This Chapter

This chapter uses the `IdentityApp` project from Chapter 8. Open a new PowerShell command prompt and run the commands shown in Listing 9-1 to reset the application and Identity databases.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from `https://github.com/Apress/pro-asp.net-core-identity`. See Chapter 1 for how to get help if you have problems running the examples.

Listing 9-1. Resetting the Databases

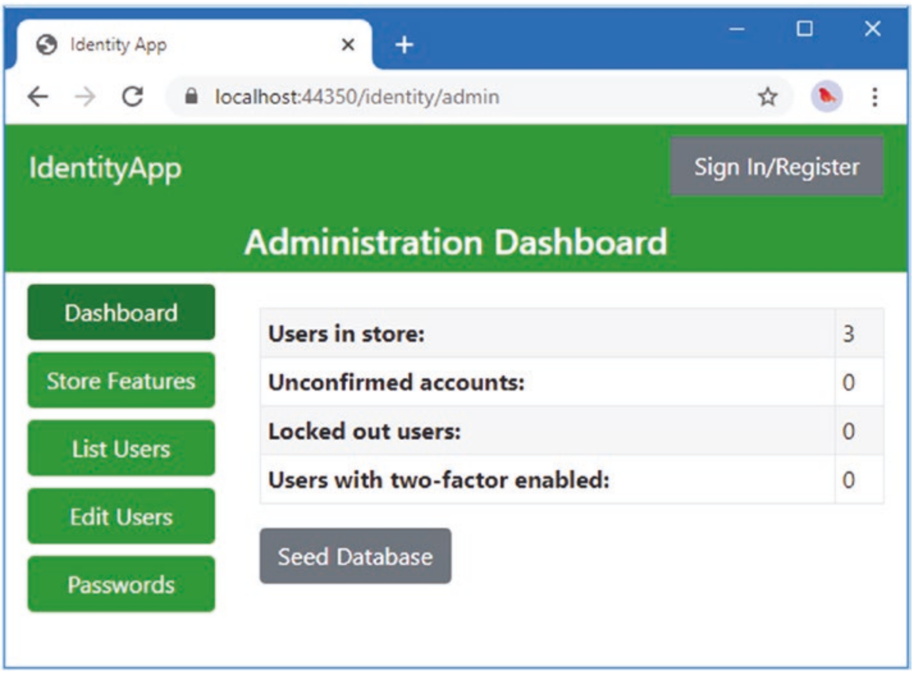
```
dotnet ef database drop --force --context ProductDbContext
dotnet ef database drop --force --context IdentityDbContext
dotnet ef database update --context ProductDbContext
dotnet ef database update --context IdentityDbContext
```

Use the PowerShell prompt to run the command shown in Listing 9-2 in the `IdentityApp` folder to start the application.

Listing 9-2. Running the Example Application

```
dotnet run
```

Open a web browser and request `https://localhost:44350/Identity/Admin`, which will show the administration dashboard. Click the Seed Database button to add the test accounts to the user store, as shown in Figure 9-1.



Administration Dashboard

Users in store:	3
Unconfirmed accounts:	0
Locked out users:	0
Users with two-factor enabled:	0

Figure 9-1. *Running the example application*

Creating User Accounts

In applications that don't support self-registration, creating accounts is one of the most important workflows. For this chapter, I am going to create a workflow that allows the administrator to create an account but that will require the user to choose their password. See Part 2 for an example of a more conventional approach where the administrator provides all the data for the new account, including the password.

Add a Razor Page named `Create.cshtml` to the `Pages/Identity/Admin` folder with the content shown in Listing 9-3.

Listing 9-3. The Contents of the Create.cshtml File in the Pages/Identity/Admin Folder

```
@page
@model IdentityApp.Pages.Identity.Admin.CreateModel
@{
    ViewBag.Workflow = "Create";
}

<div asp-validation-summary="All" class="text-danger m-2"></div>
```

```
@if (TempData.ContainsKey("message")) {
    <div class="alert alert-success">@TempData["message"]</div>
}

<form method="post">
    <div class="form-group">
        <label>Email</label>
        <input class="form-control" name="email" />
    </div>
    <div>
        <button type="submit" class="btn btn-success">Create</button>
        <a asp-page="Dashboard" class="btn btn-secondary">Cancel</a>
    </div>
</form>
```

The view part of the page is a simple form that captures the email address for the new account. To implement the page model, add the code shown in Listing 9-4 to the `Create.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 9-4. The Contents of the Create.cshtml.cs File in the Pages/Identity/Admin Folder

```
using IdentityApp.Services;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;

namespace IdentityApp.Pages.Identity.Admin {

    public class CreateModel : AdminPageModel {

        public CreateModel(UserManager<IdentityUser> mgr,
            IdentityEmailService emailService) {
            UserManager = mgr;
            EmailService = emailService;
        }

        public UserManager<IdentityUser> UserManager { get; set; }
        public IdentityEmailService EmailService { get; set; }

        [BindProperty(SupportsGet = true)]
        [EmailAddress]
        public string Email { get; set; }

        public async Task<IActionResult> OnPostAsync() {
            if (ModelState.IsValid) {
                IdentityUser user = new IdentityUser {
                    UserName = Email,
                    Email = Email,
                    EmailConfirmed = true
                };
                IdentityResult result = await UserManager.CreateAsync(user);
                if (result.Process(ModelState)) {
                    await EmailService.SendPasswordRecoveryEmail(user,
                        "/Identity/UserAccountComplete");
                    TempData["message"] = "Account Created";
                    return RedirectToPage();
                }
            }
            return Page();
        }
    }
}
```

The page model constructor declares dependencies on the UserManager<IdentityUser> class, which is used to add an IdentityUser object to the user store through the CreateAsync method.

```
...
IdentityUser user = new IdentityUser {
    UserName = email,
    Email = email,
    EmailConfirmed = true
};
IdentityResult result = await UserManager.CreateAsync(user);
...
```

I set the UserName, Email, and EmailConfirmed properties of a newly created IdentityUser object and use it as the argument to the CreateAsync method. As part of the storage process, other values for other properties will be generated, such as the Id, NormalizedUserName, and NormalizedEmail properties.

The CreateAsync method returns an IdentityAsync object, which indicates the outcome of the operation. If the new object was successfully added to the store, I use the IdentityEmailService to send the user an email to the user containing a link. I have used the same method defined in Chapter 8 for password recovery.

```
...
await EmailService.SendPasswordRecoveryEmail(user, "/Identity/UserAccountComplete");
...
```

The email sent to the user will contain a link with a confirmation token. To allow the user to validate the token and complete their account setup, add a Razor Page named UserAccountComplete.cshtml to the Pages/Identity folder with the content shown in Listing 9-5.

Listing 9-5. The Contents of the UserAccountComplete.cshtml File in the Pages/Identity Folder

```
@page "{email?}/{token?}"
@model IdentityApp.Pages.Identity.UserAccountCompleteModel
@{
    ViewData["showNav"] = false;
    ViewData["banner"] = "Complete Account";
}
```

```
@if (string.IsNullOrEmpty(Model.Token) || string.IsNullOrEmpty(Model.Email)) {
    <div class="h6 text-center">
        <div class="p-2">
            Check your inbox for a confirmation email and click the link it contains.
        </div>
    </div>
} else {
    <div asp-validation-summary="All" class="text-danger m-2"></div>
    <form method="post">
        <input type="hidden" asp-for="Token" />
        <div class="form-group">
            <label>Email</label>
            <input class="form-control" asp-for="Email" readonly />
        </div>
        <div class="form-group">
            <label>Password</label>
            <input class="form-control" type="password" name="password" />
        </div>
        <div class="form-group">
            <label>Confirm Password</label>
            <input class="form-control" type="password" name="confirmpassword" />
        </div>
        <button class="btn btn-primary" type="submit">Finish and Sign In</button>
    </form>
}
```

The view part of the page follows the approach I took for password recovery, although I have defined a separate Razor Page so that the account completion process can be easily customized. The user is prompted to enter a password and is presented with a Finish and Sign In button that will update the user store and redirect the browser to the sign-in page.

To define the page model class, add the code shown in Listing 9-6 to the `UserAccountComplete.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 9-6. The Contents of the UserAccountComplete.cshtml.cs File in the Pages/Identity Folder

```
using IdentityApp.Services;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;

namespace IdentityApp.Pages.Identity {

    [AllowAnonymous]
    public class UserAccountCompleteModel : UserPageModel {

        public UserAccountCompleteModel(UserManager<IdentityUser> usrMgr,
                TokenUrlEncoderService tokenUrlEncoder) {
            UserManager = usrMgr;
            TokenUrlEncoder = tokenUrlEncoder;
        }
        public UserManager<IdentityUser> UserManager { get; set; }
        public TokenUrlEncoderService TokenUrlEncoder { get; set; }

        [BindProperty(SupportsGet = true)]
        public string Email { get; set; }

        [BindProperty(SupportsGet = true)]
        public string Token { get; set; }

        [BindProperty]
        [Required]
        public string Password { get; set; }

        [BindProperty]
        [Required]
        [Compare(nameof(Password))]
        public string ConfirmPassword { get; set; }

        public async Task<IActionResult> OnPostAsync() {
            if (ModelState.IsValid) {
                IdentityUser user = await UserManager.FindByEmailAsync(Email);
                string decodedToken = TokenUrlEncoder.DecodeToken(Token);
                IdentityResult result = await UserManager.ResetPasswordAsync(user,
                    decodedToken, Password);
                if (result.Process(ModelState)) {
                    return RedirectToPage("SignIn", new { });
                }
            }
            return Page();
        }
    }
}
```

The POST handler method decodes the token and uses it with the user manager's `ResetPasswordAsync` method to set the password, after which the user is redirected to the `SignIn` page so they can sign into the application. For quick reference, Table 9-3 describes the user manager methods used to create and complete an account in this workflow.

Table 9-3. *The UserManager<IdentityUser> Methods Used to Create a New Account*

Name	Description
`CreateAsync(user)`	This method adds an `IdentityUser` object to the store.
`GeneratePasswordResetTokenAsync(user)`	This method generates a token that can be validated by the `ResetPasswordAsync` method. The token is securely sent to the user so that possession of the token establishes the identity of the user.
`ResetPasswordAsync(user, token, password)`	This method validates the token provided by the user and changes the stored password if it matches the one generated by the `GeneratePasswordResetTokenAsync` method.

To complete the workflow, add the element shown in Listing 9-7 to create a button in the administration navigation view.

Listing 9-7. Adding Navigation in the _AdminWorkflows.cshtml File in the Pages/Identity/Admin Folder

```
@model (string workflow, string theme)

@{
    Func<string, string> getClass = (string feature) =>
        feature != null && feature.Equals(Model.workflow) ? "active" : "";
}

<a class="btn btn-@Model.theme btn-block @getClass("Dashboard")"
        asp-page="Dashboard">
    Dashboard
</a>
<a class="btn btn-@Model.theme btn-block @getClass("Features")" asp-page="Features">
    Store Features
</a>
<a class="btn btn-success btn-block @getClass("List")" asp-page="View"
        asp-route-id="">
    List Users
</a>
<a class="btn btn-success btn-block @getClass("Create")" asp-page="Create">
    Create Account
</a>
<a class="btn btn-success btn-block @getClass("Edit")" asp-page="Edit"
        asp-route-id="">
    Edit Users
</a>
<a class="btn btn-success btn-block @getClass("Passwords")" asp-page="Passwords"
        asp-route-id="">
    Passwords
</a>
```

Restart ASP.NET Core, request `https://localhost:44350/Identity/Admin` and click the Create Account button. Enter dora@example.com into the text field and click the Create button, as shown in Figure 9-2.

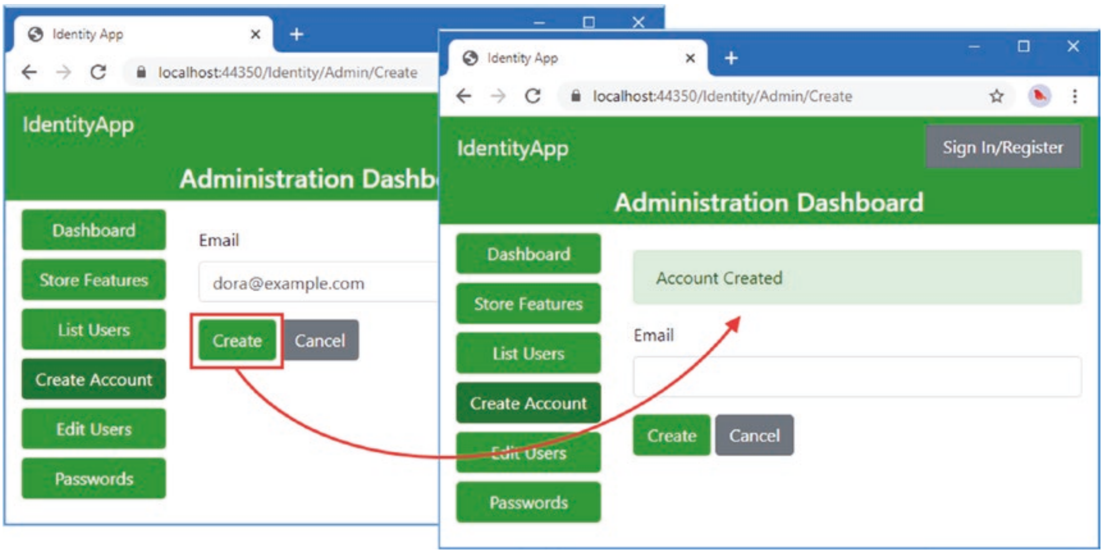


Figure 9-2. Creating a new account

Look at the ASP.NET Core console output, and you will see a message like this one, albeit with a different validation token:

```
---New Email----
To: dora@example.com
Subject: Set Your Password
Please set your password by <a href=https://localhost:44350/Identity/UserAccountComplete/
dora@example.com/Q2ZESjh>
    clicking here
</a>.
-------
```

This simulated email is produced by the `IEmailSender` service, which I created for the Identity UI package. The encoded token is a long string, which I have shorted here for brevity.

Copy the URL from the message displayed by your application (and not the one shown earlier), and you will be prompted to select a password. Enter mysecret into the Password and Confirm Password fields and click the Finish and Sign In button. The user store will be updated, and you will be redirected to the sign-in page. Use `dora@example.com` with the password `mysecret` to sign into the application, as shown in Figure 9-3.

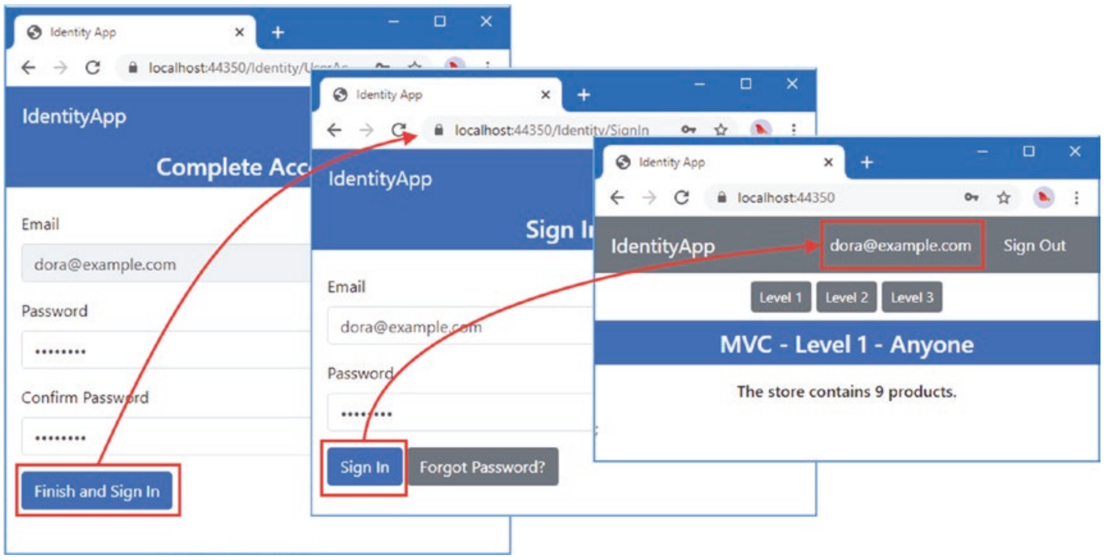


Figure 9-3. *Choosing a password and signing into the application*

Performing Self-Service Registration

Self-registration uses the basic approach as the previous example, with the exception that the email address provided by the user should not be trusted until it is confirmed until the user provides a confirmation token that has been sent securely to them. Listing 9-8 adds a method to the `IdentityEmailService` class that will send the user an email that contains a token for confirming their email address.

Listing 9-8. Adding a Method in the IdentityEmailService.cs File in the Services Folder

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.UI.Services;
using Microsoft.AspNetCore.Routing;
using System.Threading.Tasks;

namespace IdentityApp.Services {

    public class IdentityEmailService {

        // ...methods omitted for brevity...

        public async Task SendAccountConfirmEmail(IdentityUser user,
                string confirmationPage) {
            string token =
                await UserManager.GenerateEmailConfirmationTokenAsync(user);
            string url = GetUrl(user.Email, token, confirmationPage);
            await EmailSender.SendEmailAsync(user.Email,
                "Complete Your Account Setup",
                $"Please set up your account by <a href={url}>clicking here</a>.");
        }
    }
}
```

The new email will include a token created with the user manager's GenerateEmailConfirmationTokenAsync method, which generates a token that can be used to confirm email addresses.

Add a Razor Page named SignUp.cshtml to the Pages/Identity folder with the content shown in Listing 9-9.

Listing 9-9. The Contents of the SignUp.cshtml File in the Pages/Identity Folder

```
@page
@model IdentityApp.Pages.Identity.SignUpModel
@{
    ViewData["showNav"] = false;
    ViewData["banner"] = "Sign Up";
}

<div asp-validation-summary="All" class="text-danger m-2"></div>

<form method="post" class="m-4">
    <div class="form-group">
        <label>Email</label>
        <input class="form-control" asp-for="Email" />
    </div>
    <div class="form-group">
        <label>Password</label>
        <input class="form-control" type="password" asp-for="Password" />
    </div>
    <button class="btn btn-primary">Sign Up</button>
</form>
```

The view part of the page displays a form that asks the user for their email address and password. For the page model class, add the code shown in Listing 9-10 to the SignUp.cshtml.cs file. (You will have to create this file if you are using Visual Studio Code.)

Listing 9-10. The Contents of the SignUp.cshtml.cs File in the Pages/Identity Folder

```
using IdentityApp.Services;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;

namespace IdentityApp.Pages.Identity {

    [AllowAnonymous]
    public class SignUpModel : UserPageModel {
        public SignUpModel(UserManager<IdentityUser> usrMgr,
                IdentityEmailService emailService) {
            UserManager = usrMgr;
            EmailService = emailService;
        }

        public UserManager<IdentityUser> UserManager { get; set; }
        public IdentityEmailService EmailService { get; set; }

        [BindProperty]
        [Required]
        [EmailAddress]
        public string Email { get; set; }

        [BindProperty]
        [Required]
        public string Password { get; set; }

        public async Task<IActionResult> OnPostAsync() {
            if (ModelState.IsValid) {
                IdentityUser user = await UserManager.FindByEmailAsync(Email);
                if (user != null && !await UserManager.IsEmailConfirmedAsync(user)) {
                    return RedirectToPage("SignUpConfirm");
                }
                user = new IdentityUser {
                    UserName = Email,
                    Email = Email
                };
                IdentityResult result = await UserManager.CreateAsync(user);
                if (result.Process(ModelState)) {
                    result = await UserManager.AddPasswordAsync(user, Password);
                    if (result.Process(ModelState)) {
                        await EmailService.SendAccountConfirmEmail(user,
                            "SignUpConfirm");
                        return RedirectToPage("SignUpConfirm");
                    } else {
                        await UserManager.DeleteAsync(user);
                    }
                }
            }
            return Page();
        }
    }
}
```

The page model class defines a POST handler that receives the email address and password provided by the user. The email address is used to create an `IdentityUser` object that is added to the user store with the `CreateAsync` method. The password is set using the `AddPasswordAsyc` method, which presents a potential problem because passwords are validated before they are stored, to ensure they conform to the policy settings I described in Chapter 5. If a password doesn't pass validation, an error will be reported after the

IdentityUser has been stored, which will produce an account that cannot be used but which is associated with the user's email address. The simplest way to solve this problem is to use the DeleteAsync method to remove the IdentityUser object from the store if there is a problem with the password.

If the IdentityUser object is stored successfully, a confirmation email is sent. Once the email has been sent, the browser is redirected to a Razor Page named SignUpConfirm, which I create in the next section. This is also the Razor Page that will receive the request when the user clicks the link in the email they receive.

A common problem arises when the user forgets to confirm the account and subsequently repeats the signup process using the same email address. This will cause an error because the unconfirmed account is already in the user store, essentially trapping the user. To avoid this problem, I check the user store to see if it contains an unconfirmed account and perform a redirection if it does.

```
...
IdentityUser user = await UserManager.FindByEmailAsync(Email);
if (user != null && !await UserManager.IsEmailConfirmedAsync(user)) {
    return RedirectToPage("SignUpConfirm");
}
...
```

The redirection allows the user to complete the sign-up process, resending the account confirmation email if required.

Confirming Self-Registered Accounts

To allow the user to confirm their account, add a Razor Page named SignUpConfirm.cshtml to the Pages/Identity folder with the content shown in Listing 9-11.

Listing 9-11. The Contents of the SignUpConfirm.cshtml File in the Pages/Identity Folder

```
@page "{email?}/{token?}"
@model IdentityApp.Pages.Identity.SignUpConfirmModel
@{
    ViewData["showNav"] = false;
    ViewData["banner"] = "Account Confirmation";
}

@if (Model.ShowConfirmedMessage) {
    <div class="text-center">
        <h6 class="p-2">Account confirmed</h6>
        <a asp-page="SignIn" class="btn btn-primary">Sign In</a>
    </div>
} else {
    <div class="text-center">
        <h6 class="p-2">
            Check your inbox for a confirmation email and
            click the link it contains.
        </h6>
        <a asp-page="SignUpResend" class="btn btn-primary">Resend Email</a>
    </div>
}
```

This Razor Page will be shown when the user has been sent an email and will receive the request when they click the link in the email. The view section of the page alters the message it displays to suit these situations. Add the code shown in Listing 9-12 to the `SignUpConfirm.cshtml.cs` file to define the page model class. (You will have to create this file if you are using Visual Studio Code.)

Listing 9-12. The Contents of the SignUpConfirm.cshtml.cs File in the Pages/Identity Folder

```
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.WebUtilities;
using System.Text;
using System.Threading.Tasks;

namespace IdentityApp.Pages.Identity {

    [AllowAnonymous]
    public class SignUpConfirmModel : UserPageModel {

        public SignUpConfirmModel(UserManager<IdentityUser> usrMgr)
            => UserManager = usrMgr;

        public UserManager<IdentityUser> UserManager { get; set; }

        [BindProperty(SupportsGet = true)]
        public string Email { get; set; }

        [BindProperty(SupportsGet = true)]
        public string Token { get; set; }

        public bool ShowConfirmedMessage { get; set; } = false;

        public async Task<IActionResult> OnGetAsync() {
            if (!string.IsNullOrEmpty(Email) && !string.IsNullOrEmpty(Token)) {
                IdentityUser user = await UserManager.FindByEmailAsync(Email);
                if (user != null) {
                    string decodedToken = Encoding.UTF8.GetString(
                        WebEncoders.Base64UrlDecode(Token));
                    IdentityResult result =
                        await UserManager.ConfirmEmailAsync(user, decodedToken);
                    if (result.Process(ModelState)) {
                        ShowConfirmedMessage = true;
                    }
                }
            }
            return Page();
        }
    }
}
```

Confirming an account is done by retrieving the `IdentityUser` object from the store and passing it to the `ConfirmEmailAsync` method, along with the token that was provided by the user. If the confirmation is successful, then the confirmation message is shown.

Resending Confirmation Emails

For self-service applications, it is important to allow the user to request the confirmation emails are sent again if the original isn't delivered. Add a Razor Page named `SignUpResend.cshtml` to the `Pages/Identity` folder with the content shown in Listing 9-13.

Listing 9-13. The Contents of the SignUpResend.cshtml File in the Pages/Identity Folder

```
@page
@model IdentityApp.Pages.Identity.SignUpResendModel
@{
    ViewData["showNav"] = false;
    ViewData["banner"] = "Account Confirmation";
}

<div asp-validation-summary="All" class="text-danger m-2"></div>

@if (TempData.ContainsKey("message")) {
    <div class="alert alert-success">@TempData["message"]</div>
}

<form method="post">
    <div class="form-group">
        <label>Your email address</label>
        <input class="form-control" asp-for="Email" />
    </div>
    <button type="submit" class="btn btn-primary">Resend Email</button>
</form>
```

The view part of the page contains a form that the user can submit to request that the confirmation email be reset. To create the page model class, add the code shown in Listing 9-14 to the `SignUpResend.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 9-14. The Contents of the SignUpResend.cshtml.cs File in the Pages/Identity Folder

```
using IdentityApp.Services;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;

namespace IdentityApp.Pages.Identity {

    [AllowAnonymous]
    public class SignUpResendModel : UserPageModel {
        public SignUpResendModel(UserManager<IdentityUser> usrMgr,
            IdentityEmailService emailService) {
            UserManager = usrMgr;
            EmailService = emailService;
        }

        public UserManager<IdentityUser> UserManager { get; set; }
        public IdentityEmailService EmailService { get; set; }

        [EmailAddress]
        [BindProperty(SupportsGet = true)]
        public string Email { get; set; }

        public async Task<IActionResult> OnPostAsync() {
            if (ModelState.IsValid) {
                IdentityUser user = await UserManager.FindByEmailAsync(Email);
                if (user != null && !await UserManager.IsEmailConfirmedAsync(user)) {
                    await EmailService.SendAccountConfirmEmail(user,
                        "SignUpConfirm");
                }
                TempData["message"] = "Confirmation email sent. Check your inbox.";
                return RedirectToPage(new { Email });
            }
            return Page();
        }
    }
}
```

The POST handler method receives the email address provided by the user and retrieves the `IdentityUser` object from the store. If there is an `IdentityUser` object and the user manager's `IsEmailConfirmedAsync` method reports that the email address is unconfirmed, then a new email is sent using the `IdentityEmailService.SendAccountConfirmEmail` method.

A temp data value is set that will display a message to the user. This is displayed regardless of whether the email is sent to prevent this page from being used to determine which accounts exist and if they are awaiting confirmation.

Handling Unconfirmed Sign-ins

The user may attempt to sign in to the application after creating an account but before clicking the email link. To avoid confusion, add the code shown in Listing 9-15 to the page model class for the `SignIn` page that will notify the user that account confirmation is required.

■ **Note** Accounts with unconfirmed email addresses are prevented from signing in only if the `SignIn.RequireConfirmedAccount` configuration option is `true`, as described in Chapter 5.

Listing 9-15. Handling Unconfirmed Accounts in the SignIn.cshtml.cs File in the Pages/Identity Folder

```
using System.ComponentModel.DataAnnotations;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using SignInResult = Microsoft.AspNetCore.Identity.SignInResult;
using Microsoft.AspNetCore.Authorization;

namespace IdentityApp.Pages.Identity {

    [AllowAnonymous]
    public class SignInModel : UserPageModel {

        public SignInModel(SignInManager<IdentityUser> signMgr,
                UserManager<IdentityUser> usrMgr) {
            SignInManager = signMgr;
            UserManager = usrMgr;
        }

        public SignInManager<IdentityUser> SignInManager { get; set; }
        public UserManager<IdentityUser> UserManager { get; set; }

        [Required]
        [EmailAddress]
        [BindProperty]
        public string Email { get; set; }

        [Required]
        [BindProperty]
        public string Password { get; set; }

        [BindProperty(SupportsGet = true)]
        public string ReturnUrl { get; set; }

        public async Task<IActionResult> OnPostAsync() {
            if (ModelState.IsValid) {
                SignInResult result = await SignInManager.PasswordSignInAsync(Email,
                    Password, true, true);
                if (result.Succeeded) {
                    return Redirect(ReturnUrl ?? "/");
                } else if (result.IsLockedOut) {
                    TempData["message"] = "Account Locked";
                } else if (result.IsNotAllowed) {
                    IdentityUser user = await UserManager.FindByEmailAsync(Email);
                    if (user != null &&
                           !await UserManager.IsEmailConfirmedAsync(user)) {
                        return RedirectToPage("SignUpConfirm");
                    }
                    TempData["message"] = "Sign In Not Allowed";
                } else if (result.RequiresTwoFactor) {
                    return RedirectToPage("SignInTwoFactor", new { ReturnUrl });
                } else {
                    TempData["message"] = "Sign In Failed";
                }
            }
            return Page();
        }
    }
}
```

When a user signs in with the correct password but has an unconfirmed account, the sign-in manager will return a `SignInResult` whose `IsNotAllowed` property is `true`. The new code in Listing 9-15 retrieves the `IdentityUser` object for the user's email address, determines if the account has a confirmed email address with the `IsEmailConfirmedAsync` method, and performs a redirection to the `SignUpConfirm` page. For quick reference, Table 9-4 describes the user manager methods used for self-service registration.

Table 9-4. *The UserManager<IdentityUser> Methods for Self-Registration*

Name	Description
`CreateAsync(user)`	This method adds an `IdentityUser` object to the store.
`AddPasswordAsync(user, password)`	This method sets a password for the specified `IdentityUser` object.
`GenerateEmailConfirmationTokenAsync(user)`	This method generates a token that can be sent to the user to confirm their email address.
`ConfirmEmailAsync(user, token)`	This method validates a token generated by the `GenerateEmailConfirmationTokenAsync` method to confirm an email address.
`IsEmailConfirmedAsync(user)`	This method returns `true` if the email address for the specified `IdentityUser` object has been confirmed.

Integrating Self-Service Registration

To integrate the self-service feature into the rest of the application, add the element shown in Listing 9-16 to the `SignIn.cshtml` file.

Listing 9-16. Adding an Element in the SignIn.cshtml File in the Pages/Identity Folder

```
@page "{returnUrl?}"
@model IdentityApp.Pages.Identity.SignInModel
@{
    ViewData["showNav"] = false;
    ViewData["banner"] = "Sign In";
}

<div asp-validation-summary="All" class="text-danger m-2"></div>

@if (TempData.ContainsKey("message")) {
    <div class="alert alert-danger">@TempData["message"]</div>
}
```

```
<form method="post">
    <div class="form-group">
        <label>Email</label>
        <input class="form-control" name="email" />
    </div>
    <div class="form-group">
        <label>Password</label>
        <input class="form-control" type="password" name="password" />
    </div>
    <button type="submit" class="btn btn-primary">
        Sign In
    </button>
    <a asp-page="SignUp" class="btn btn-primary">Register</a>
    <a asp-page="UserPasswordRecovery" class="btn btn-secondary">Forgot Password?</a>
</form>
```

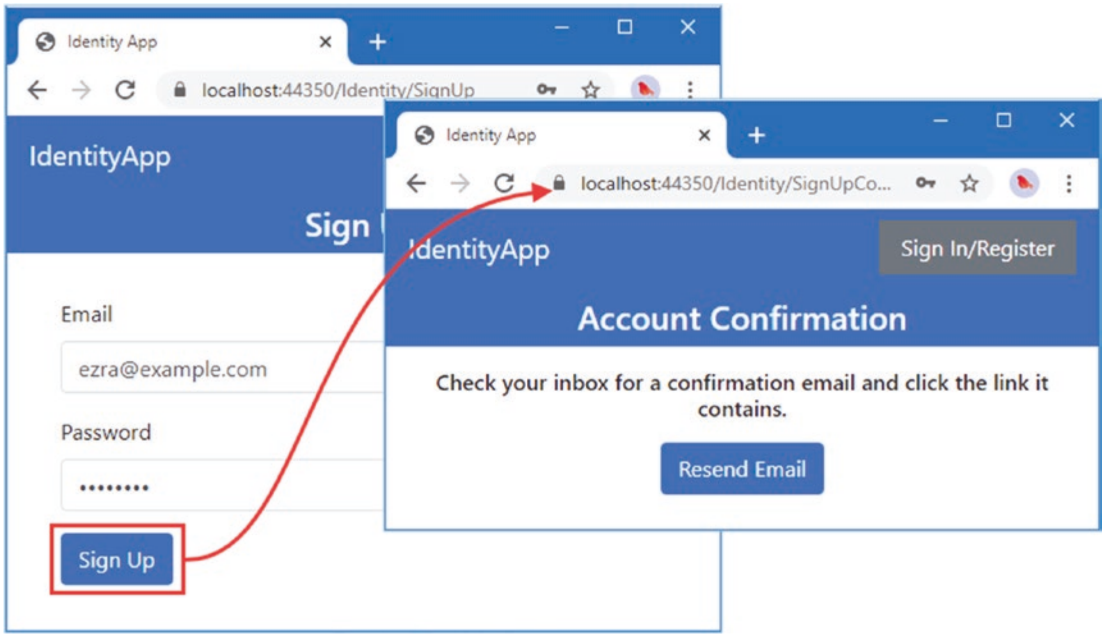
The final change is to display the number of unconfirmed accounts in the administrator dashboard, as shown in Listing 9-17.

Listing 9-17. Displaying Accounts in the Dashboard.cshtml.cs File in the Pages/Identity/Admin Folder

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;
using System.Linq;

namespace IdentityApp.Pages.Identity.Admin {

    public class DashboardModel : AdminPageModel {

        public DashboardModel(UserManager<IdentityUser> userMgr)
            => UserManager = userMgr;

        public UserManager<IdentityUser> UserManager { get; set; }

        public int UsersCount { get; set; } = 0;
        public int UsersUnconfirmed { get; set; } = 0;
        public int UsersLockedout { get; set; } = 0;
        public int UsersTwoFactor { get; set; } = 0;

        private readonly string[] emails = {
            "alice@example.com", "bob@example.com", "charlie@example.com"
        };

        public void OnGet() {
            UsersCount = UserManager.Users.Count();
            UsersUnconfirmed = UserManager.Users
                .Where(u => !u.EmailConfirmed).Count();
        }
        public async Task<IActionResult> OnPostAsync() {
            // ...statements omitted for brevity...
        }
    }
}
```

■ **Tip** Although I have been relying on the user manager's `IsEmailConfirmedAsync` method in earlier listings, the LINQ query in Listing 9-17 operates directly on the properties defined by the `IdentityUser` class. This ensures that the query can be executed by the database server, which isn't possible with the `IsEmailConfirmedAsync` method, which cannot be easily transformed into a SQL query and which requires client execution.

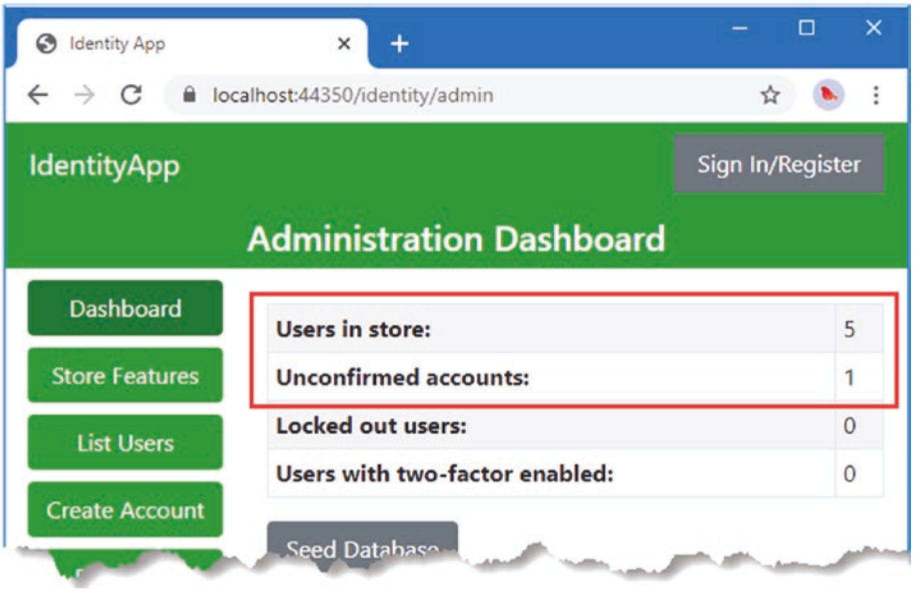
Restart ASP.NET Core and request `https://localhost:44350/Identity/SignUp`. Enter ezra@example.com into the Email field and mysecret into the Password field. Click the Sign Up button, and you will be prompted to check your inbox, as shown in Figure 9-4.



Check your inbox for a confirmation email and click the link it contains.

Figure 9-4. *Self-registration*

The account is created but not yet confirmed, which you can see by requesting `https://localhost:44350/Identity/admin`. The dashboard overview shows that there are now five accounts in the user store, one of which is unconfirmed, as shown in Figure 9-5.



Users in store:	5
Unconfirmed accounts:	1
Locked out users:	0
Users with two-factor enabled:	0

Figure 9-5. *The unconfirmed account shown in the administration dashboard*

Examine the ASP.NET Core console output, and you will see a message similar to this one:

```
---New Email----
To: ezra@example.com
Subject: Complete Your Account Setup
Please set up your account by <a href=https://localhost:44350/Identity/SignUpConfirm/ezra@
example.com/Q2ZESj>
    clicking here
</a>.
-------
```

I have shortened the confirmation token for brevity. Copy the URL from the email generated by your application (which will be different to the one I received because each confirmation token is unique) and paste it into a browser window to confirm the account.

Click the Sign In button, and you will be presented with the sign in page. Use `ezra@example.com` as the email address and `mysecret` as the password to sign into the application, as shown in Figure 9-6.

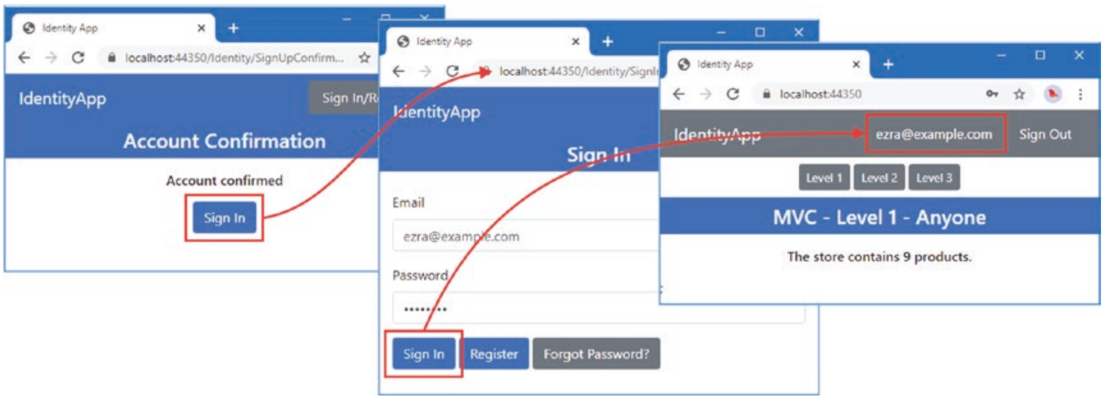


Figure 9-6. *Completing the self-registration process*

Locking Out Accounts

ASP.NET Core Identity can be configured to lock out accounts after a specified number of failed sign-ins. When an application supports lockouts, it is a good idea to create workflows that let administrators lock and unlock accounts. Locking an account is useful as a temporary measure, such as when suspicious behavior is observed. Unlocking an account is useful when a senior executive forgets their password, demands that it be changed, and won't wait for the lockout to expire. Add a Razor Page named `Lockouts.cshtml` to the `Pages/Identity/Admin` folder with the content shown in Listing 9-18.

UNDERSTANDING LOCKOUTS

For an account to be locked out, the `IdentityUser.LockoutEnabled` property must be `true`, and the final argument to the `SignInManager<IdentityUser>.PasswordSignInAsync` must be `true` when the user's credentials are checked. See Chapter 6 for the configuration options that control `LockoutEnabled`, and see Chapter 8 for a description of the `PasswordSignInAsync` method. Lockouts are described in detail in Chapter 20, including an explanation of how they are implemented in the user store.

Listing 9-18. The Contents of the Lockouts.cshtml File in the Pages/Identity/Admin Folder

```
@page
@model IdentityApp.Pages.Identity.Admin.LockoutsModel
@{
    ViewBag.Workflow = "Lockouts";
}

<table class="table table-sm table-striped table-bordered">
    <thead>
        <tr><th class="text-center py-2" colspan="3">Locked Out Users</th></tr>
    </thead>
    <tbody>
        @if (Model.LockedOutUsers.Count() == 0) {
            <tr>
                <td colspan="3" class="py-2 text-center">
                    No locked out users
                </td>
            </tr>
        } else {
            <tr><th>Email</th><th>Lockout Remaining</th><th/></tr>
            @foreach (IdentityUser user in Model.LockedOutUsers) {
                TimeSpan timeLeft = await Model.TimeLeft(user);
                <tr>
                    <td>@user.Email</td>
                    <td>
                        @timeLeft.Days days, @timeLeft.Hours hours,
                        @timeLeft.Minutes min, @timeLeft.Seconds secs
                    </td>
                    <td>
                        <form method="post" asp-page-handler="unlock">
                            <input type="hidden" name="id" value="@user.Id" />
                            <button type="submit" class="btn btn-sm btn-success">
                                Unlock Now
                            </button>
                        </form>
                    </td>
                </tr>
            }
        }
    </tbody>
</table>

<table class="table table-sm table-striped table-bordered">
    <thead>
        <tr><th class="text-center py-2" colspan="2">Other Users</th></tr>
    </thead>
    <tbody>
        @if (Model.OtherUsers.Count() == 0) {
            <tr>
                <th colspan="2" class="py-2 text-center">
                    All users locked out
                </th>
            </tr>
        } else {
            <tr><th>Email</th><th/></tr>
            @foreach (IdentityUser user in Model.OtherUsers) {
                <tr>
                    <td>@user.Email</td>
                    <td>
                        <form method="post" asp-page-handler="lock">
                            <input type="hidden" name="id" value="@user.Id" />
                            <button type="submit" class="btn btn-sm btn-success">
```

```
                                Lock Out
                            </button>
                        </form>
                    </td>
                </tr>
            }
        }
    </tbody>
</table>
```

The view part of the page displays two tables. The first table lists the locked-out users, showing when the lockout expires, and provides buttons to unlock accounts immediately. The second table lists the remaining users, with buttons to start lockouts. To define the page model class, add the code shown in Listing 9-19 to the `Lockouts.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 9-19. The Contents of the Lockouts.cshtml.cs File in the Pages/Identity/Admin Folder

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace IdentityApp.Pages.Identity.Admin {

    public class LockoutsModel : AdminPageModel {

        public LockoutsModel(UserManager<IdentityUser> usrMgr)
            => UserManager = usrMgr;

        public UserManager<IdentityUser> UserManager { get; set; }

        public IEnumerable<IdentityUser> LockedOutUsers { get; set; }
        public IEnumerable<IdentityUser> OtherUsers { get; set; }

        public async Task<TimeSpan> TimeLeft(IdentityUser user)
            => (await UserManager.GetLockoutEndDateAsync(user))
                .GetValueOrDefault().Subtract(DateTimeOffset.Now);

        public void OnGet() {
            LockedOutUsers = UserManager.Users.Where(user => user.LockoutEnd.HasValue
                    && user.LockoutEnd.Value > DateTimeOffset.Now)
                .OrderBy(user => user.Email).ToList();
            OtherUsers = UserManager.Users.Where(user => !user.LockoutEnd.HasValue
                    || user.LockoutEnd.Value <= DateTimeOffset.Now)
                 .OrderBy(user => user.Email).ToList();
        }

        public async Task<IActionResult> OnPostLockAsync(string id) {
            IdentityUser user = await UserManager.FindByIdAsync(id);
            await UserManager.SetLockoutEnabledAsync(user, true);
            await UserManager.SetLockoutEndDateAsync(user,
                DateTimeOffset.Now.AddDays(5));
            return RedirectToPage();
        }

        public async Task<IActionResult> OnPostUnlockAsync(string id) {
            IdentityUser user = await UserManager.FindByIdAsync(id);
            await UserManager.SetLockoutEndDateAsync(user, null);
            return RedirectToPage();
        }
    }
}
```

To determine if a user is locked out, the user manager's GetLockoutEndDateAsync method is called. If the result has a value and that value specifies a time in the future, then the user is locked out. If there is no value or the time has passed, then the user is not locked out.

To lock and unlock the user account, the SetLockoutEnabledAsync method is used to enable lockouts, and a time is specified using the SetLockoutEndDateAsync method. The user won't be allowed to sign in until the specified time has passed. Lockouts can be disabled by calling the SetLockoutEndDateAsync with a null argument.

This is an awkwardly implemented feature, which is made worse by the need to work directly with the LockoutEnd property defined by the IdentityUser class, which is used by the GetLockoutEndDateAsync and SetLockoutEndDateAsync methods. Using this property directly makes it possible to query the database effectively using LINQ. This isn't a huge concern for the example application but becomes an issue to watch out for in projects with large numbers of users.

I have set the lockout period to five days, which effectively suspends an account. I generally use multiday intervals for lockouts applied by an administrator so that an account that has attracted attention doesn't become active again while investigations are ongoing. By contrast, the default lockout period applied by Identity for repeated failed sign-ins is five minutes.

To display the number of lockouts in the administrator dashboard, add the statement shown in Listing 9-20 to the Dashboard.cshtml.cs file.

Listing 9-20. Displaying Lockouts in the Dashboard.cshtml.cs File in the Pages/Identity/Admin Folder

```
...
public void OnGet() {
    UsersCount = UserManager.Users.Count();
    UsersUnconfirmed = UserManager.Users
        .Where(u => !u.EmailConfirmed).Count();
    UsersLockedout = UserManager.Users
        .Where(u => u.LockoutEnabled && u.LockoutEnd > System.DateTimeOffset.Now)
        .Count();
}
...
```

This statement uses LINQ to count the number of locked-out users by checking the LockoutEnabled and LockoutEnd properties. For quick reference, Table 9-5 describes the user manager methods for managing lockouts.

Table 9-5. *The UserManager<IdentityUser> Methods for Managing Lockouts*

Name	Description
`GetLockoutEndDateAsync(user)`	This method returns a `DateTimeOffet?` object. If this object has a value and represents a time in the future, then the account is locked out.
`SetLockoutEnabledAsync(user, enabled)`	This method sets whether lockouts are enabled for the user account. A lockout cannot be set unless lockouts are enabled.
`SetLockoutEndDateAsync(user, time)`	This method sets a lockout that will prevent the user from signing into the application until the specified time, which is expressed as a `DateTimeOffset?` value. Using a null value for the time unlocks the account.

Forcing Immediate Sign-Outs

When a user signs into the application, a cookie is added to the response. The cookie is included in subsequent HTTP requests and is used to authenticate the user without requiring them to provide their credentials again.

When an account is logged out, the user is unable to sign in to the application, but any cookies that have already been created remain valid, which means the user won't be affected by the lockout if they were already signed into the application.

A common requirement is to terminate any existing sessions when an account is logged out, which will prevent the user from using the application even if they had signed in before the lock started. The key to implementing this feature is the security stamp, which is a random string that is changed every time an alternation to the user's security data is made. The first step is to configure Identity so that it periodically validates the cookies presented by the user to see if the security stamp has changed, as shown in Listing 9-21.

Listing 9-21. Configuring the Application in the Startup.cs File in the IdentityApp Folder

```
...
services.AddIdentity<IdentityUser, IdentityRole>(opts => {
    opts.Password.RequiredLength = 8;
    opts.Password.RequireDigit = false;
    opts.Password.RequireLowercase = false;
    opts.Password.RequireUppercase = false;
    opts.Password.RequireNonAlphanumeric = false;
    opts.SignIn.RequireConfirmedAccount = true;
}).AddEntityFrameworkStores<IdentityDbContext>()
    .AddDefaultTokenProviders();

services.Configure<SecurityStampValidatorOptions>(opts => {
    opts.ValidationInterval = System.TimeSpan.FromMinutes(1);
});

services.AddScoped<TokenUrlEncoderService>();
services.AddScoped<IdentityEmailService>();
...
```

The validation feature is enabled by using the options pattern to assign an interval to the ValidationInterval property defined by the SecurityStampValidatorOptions class. I have chosen one minute for this example, but it is important to select an appropriate value for each project. Validation requires data to be retrieved from the user store, and if you set the interval too short, you will generate a large number of additional database queries, especially in applications with substantial concurrent users. On the other hand, setting the interval too long will extend the period a signed-in user will be able to continue using the application after their account is locked out.

UNDERSTANDING THE IMMEDIATE SIGN-OUT PITFALLS

There are two potential pitfalls when enabling the cookie validation feature for immediate sign-outs. First, the term *immediate* is used from the perspective of the ASP.NET Core application. From the user's perspective, they won't know they have been signed out until the next time their browser sends a request to the application, which may occur some time after the account has been locked. This isn't an issue for most projects, but if you use short lockout periods, the lockout may expire before the user next sends a request, which means the user is signed out of the application without understanding why.

The second pitfall is that any operation that changes the user's security stamp will cause validation to fail the next time the user sends an HTTP request, signing the user out of the application. Many of the methods provided by the user manager class update the security stamp, as described in Part 2, which will trigger unexpected user sign out. For self-service operations, you can prevent this issue by signing the user into the application again, as I demonstrate in Chapter 11, when I set up two-factor authentication. There is no such solution available for operations performed by an administrator, and you should bear in mind that the user may be signed out when you update their account.

The second step is to change the security stamp when locking out an account, as shown in Listing 9-22.

Listing 9-22. Changing a Security Stamp in the Lockouts.cshtml.cs File in the Pages/Identity/Admin Folder

```
...
public async Task<IActionResult> OnPostLockAsync(string id) {
    IdentityUser user = await UserManager.FindByIdAsync(id);
    await UserManager.SetLockoutEnabledAsync(user, true);
    await UserManager.SetLockoutEndDateAsync(user,
        DateTimeOffset.Now.AddDays(5));
    await UserManager.UpdateSecurityStampAsync(user);
    return RedirectToPage();
}
...
```

The UpdateSecurityStampAsync method creates a new security stamp for the specified user account, which will cause the user to be signed out the next time a cookie is validated. For quick reference, Table 9-6 describes the user manager method I used to set the security stamp.

Table 9-6. *The UserManager<IdentityUser> Method for Changing Security Stamps*

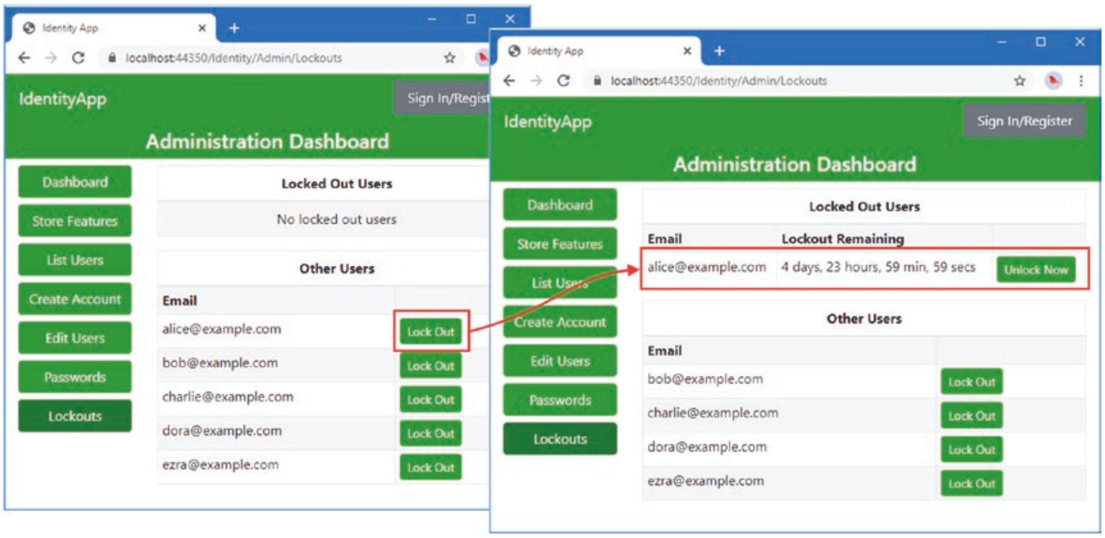
Name	Description
UpdateSecurityStampAsync(user)	This method generates a new security stamp for the specified identityUser.

Add the element shown in Listing 9-23 to the navigation partial view to integrate the new Razor Page into the administration layout.

Listing 9-23. Adding Navigation in the _AdminWorkflows.cshtml File in the Pages/Identity/Admin Folder

```
@model (string workflow, string theme)

@{
    Func<string, string> getClass = (string feature) =>
        feature != null && feature.Equals(Model.workflow) ? "active" : "";
}

<a class="btn btn-@Model.theme btn-block @getClass("Dashboard")"
        asp-page="Dashboard">
    Dashboard
</a>
<a class="btn btn-@Model.theme btn-block @getClass("Features")" asp-page="Features">
    Store Features
</a>
<a class="btn btn-success btn-block @getClass("List")" asp-page="View"
        asp-route-id="">
    List Users
</a>
<a class="btn btn-success btn-block @getClass("Create")" asp-page="Create">
    Create Account
</a>
<a class="btn btn-success btn-block @getClass("Edit")" asp-page="Edit"
        asp-route-id="">
    Edit Users
</a>
<a class="btn btn-success btn-block @getClass("Passwords")" asp-page="Passwords"
        asp-route-id="">
    Passwords
</a>
<a class="btn btn-success btn-block @getClass("Lockouts")" asp-page="Lockouts" >
    Lockouts
</a>
```

Restart ASP.NET Core, request `https://localhost:44350/Identity/Admin`, and click the Lockouts button. You will be presented with a list of users. Click the Lockout button for the `alice@example.com` account, and the account will be locked, as shown in Figure 9-7.



Locked Out Users	
No locked out users	

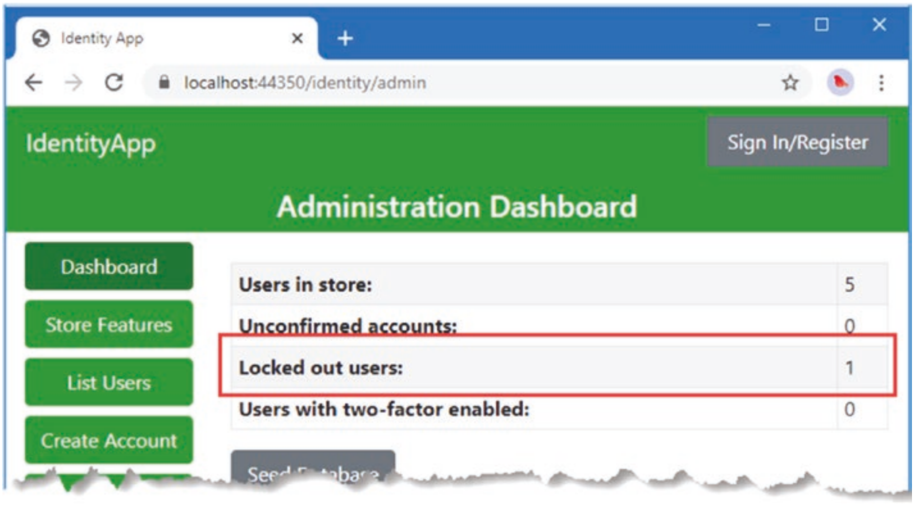
Other Users	
Email	
alice@example.com	Lock Out
bob@example.com	Lock Out
charlie@example.com	Lock Out
dora@example.com	Lock Out
ezra@example.com	Lock Out

Locked Out Users		
Email	**Lockout Remaining**	
alice@example.com	4 days, 23 hours, 59 min, 59 secs	Unlock Now

Other Users	
Email	
bob@example.com	Lock Out
charlie@example.com	Lock Out
dora@example.com	Lock Out
ezra@example.com	Lock Out

Figure 9-7. *Locking out an account*

Click the Dashboard button, and the overview will show that one account is locked out, as shown in Figure 9-8.



Users in store:	5
Unconfirmed accounts:	0
Locked out users:	1
Users with two-factor enabled:	0

Figure 9-8. *Locked accounts in the dashboard*

Click the Sign In/Register link at the top of the page and sign in using `alice@example.com` as the email address and `mysecret` as the password. Even though these are the correct credentials, you won't be able to sign in to the application, as shown in Figure 9-9.

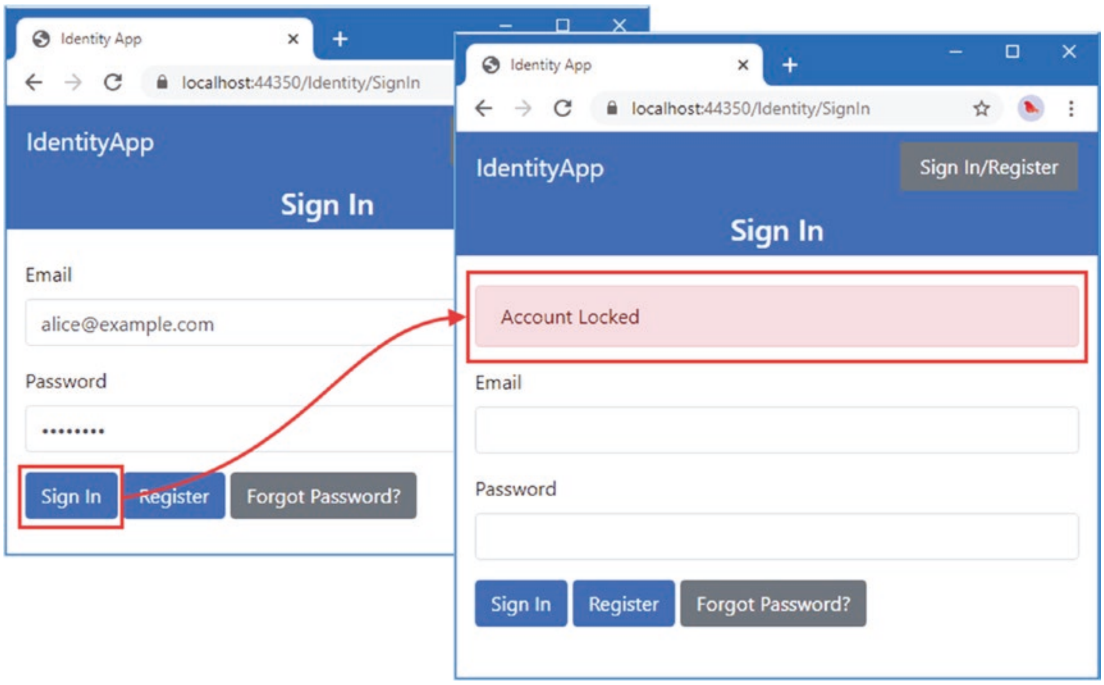


Figure 9-9. *Attempting to sign in with a locked account*

■ **Tip** Browsers share cookies between tabs, which makes it difficult to test the forced sign-out process. The best approach is to use the guest browsing features to open a window that has its own cookies and sign into the application. Use the main browser window to lock out the account. You can then continue using the application in the guest window until the next time the cookie is validated, at which point you will be signed out of the application.

Sign in to the application using `bob@example.com` as the email address and `mysecret` as the password. Use your browser's private/guest browsing feature to request `https://localhost:44350/Identity/Admin`, click the Lockouts button, and lock out `bob@example.com`. Wait for a couple of minutes and then reload the original browser window. You will see that you have been signed out of the application, as shown in Figure 9-10.

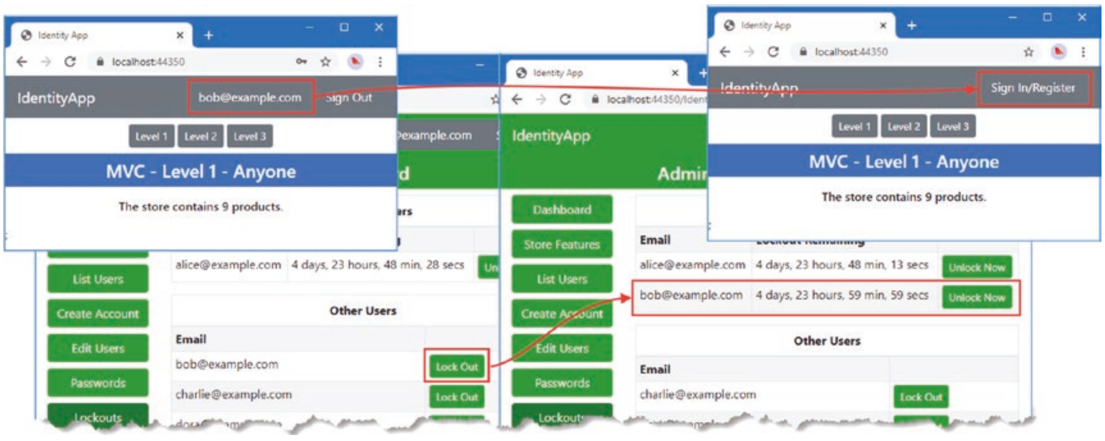


Figure 9-10. *Signing out users when locking an account*

Deleting Accounts

Accounts can be deleted for a range of reasons. For corporate applications, the most common reason is that an employee has left the organization and should no longer be able to access the application. For self-service applications, accounts are deleted because the user no longer wants to use the application.

To support administrator account deletion, add a Razor Page named Delete.cshtml to the Pages/Identity/Admin folder with the content shown in Listing 9-24.

Listing 9-24. The Contents of the Delete.cshtml File in the Pages/Identity/Admin Folder

```
@page "{id?}"
@model IdentityApp.Pages.Identity.Admin.DeleteModel
@{
    ViewBag.Workflow = "Delete";
}

<div asp-validation-summary="All" class="text-danger m-2"></div>

<form method="post">

    <h3 class="bg-danger text-white text-center p-2">Caution</h3>

    <h5 class="text-center m-2">
        Delete @Model.IdentityUser.Email?
    </h5>
    <input type="hidden" name="id" value="@Model.IdentityUser.Id" />
    <div class="text-center p-2">
        <button type="submit" class="btn btn-danger">Delete</button>
        <a asp-page="Dashboard" class="btn btn-secondary">Cancel</a>
    </div>
</form>
```

The view part of the page contains a simple form that will delete a selected account when submitted. To define the page model class, add the code shown in Listing 9-25 to the Delete.cshtml.cs file. (You will have to create this file if you are using Visual Studio Code.)

Listing 9-25. The Contents of the Delete.cshtml.cs File in the Pages/Identity/Admin Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace IdentityApp.Pages.Identity.Admin {

    public class DeleteModel : AdminPageModel {

        public DeleteModel(UserManager<IdentityUser> mgr) => UserManager = mgr;

        public UserManager<IdentityUser> UserManager { get; set; }

        public IdentityUser IdentityUser { get; set; }

        [BindProperty(SupportsGet = true)]
        public string Id { get; set; }

        public async Task<IActionResult> OnGetAsync() {
            if (string.IsNullOrEmpty(Id)) {
                return RedirectToPage("Selectuser",
                    new { Label = "Delete", Callback = "Delete" });
            }
            IdentityUser = await UserManager.FindByIdAsync(Id);
            return Page();
        }

        public async Task<IActionResult> OnPostAsync() {
            IdentityUser = await UserManager.FindByIdAsync(Id);
            IdentityResult result = await UserManager.DeleteAsync(IdentityUser);
            if (result.Process(ModelState)) {
                return RedirectToPage("Dashboard");
            }
            return Page();
        }
    }
}
```

The user manager defines the DeleteAsync method, which removes an IdentityUser object from the store. The FindByIdAsync method is used to locate the selected account, and there is a redirection to the SelectUser page if no selection has been made. For quick reference, Table 9-7 describes the method used to delete objects from the user store.

■ **Tip** Notice that I don't need to update the security stamp to force an immediate sign-out in Listing 9-25 because the `IdentityUser` data—including the security stamp—is removed from the user store. If cookie validation is enabled, as shown in Listing 9-21, the user will be automatically signed out the next time validation is performed.

Table 9-7. *The UserManager<IdentityUser> Method for Deleting Objects*

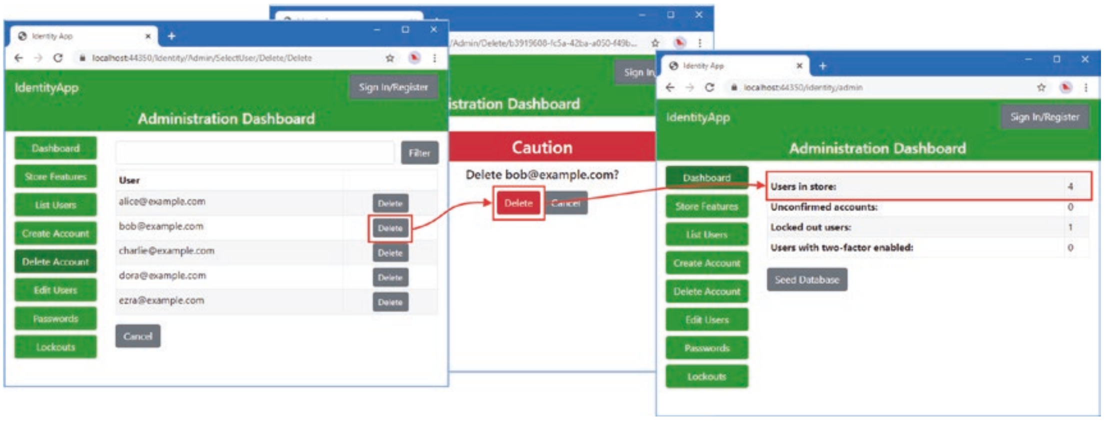
Name	Description
`DeleteAsync(user)`	This method removes the specified `IdentityUser` object from the user store.

Add the element shown in Listing 9-26 to enable navigation to the new Razor Page.

Listing 9-26. Adding Navigation in the _AdminWorkflows.cshtml File in the Pages/Identity/Admin Folder

```
@model (string workflow, string theme)

@{
    Func<string, string> getClass = (string feature) =>
        feature != null && feature.Equals(Model.workflow) ? "active" : "";
}

<a class="btn btn-@Model.theme btn-block @getClass("Dashboard")"
        asp-page="Dashboard">
    Dashboard
</a>
<a class="btn btn-@Model.theme btn-block @getClass("Features")" asp-page="Features">
    Store Features
</a>
<a class="btn btn-success btn-block @getClass("List")" asp-page="View"
        asp-route-id="">
    List Users
</a>
<a class="btn btn-success btn-block @getClass("Create")" asp-page="Create">
    Create Account
</a>
<a class="btn btn-success btn-block @getClass("Delete")" asp-page="Delete">
    Delete Account
</a>
<a class="btn btn-success btn-block @getClass("Edit")" asp-page="Edit"
        asp-route-id="">
    Edit Users
</a>
<a class="btn btn-success btn-block @getClass("Passwords")" asp-page="Passwords"
        asp-route-id="">
    Passwords
</a>
<a class="btn btn-success btn-block @getClass("Lockouts")" asp-page="Lockouts" >
    Lockouts
</a>
```

Restart ASP.NET Core and request `https://localhost:44350/identity/admin`. Click the Delete Account button and click the Delete button for the `bob@example.com` account. You will be prompted to confirm the deletion. Click Delete, and the account will be removed from the user store, as shown in Figure 9-11.



User	
alice@example.com	Delete
bob@example.com	Delete
charlie@example.com	Delete
dora@example.com	Delete
ezra@example.com	Delete

Delete bob@example.com?

Users in store:	4
Unconfirmed accounts:	0
Locked out users:	1
Users with two-factor enabled:	0

Figure 9-11. *Deleting an account*

Performing Self-Service Account Deletion

Self-service account deletion uses the same methods, and the only difference from the administrator workflow is that the `IdentityUser` object is obtained using the current `ClaimsPrincipal`, rather than querying the user store. Add a Razor Page named `UserDelete.cshtml` to the `Pages/Identity` folder, with the content shown in Listing 9-27.

Listing 9-27. The Contents of the UserDelete.cshtml File in the Pages/Identity Folder

```
@page
@model IdentityApp.Pages.Identity.UserDeleteModel
@{
    ViewBag.Workflow = "Delete";
}

<div asp-validation-summary="All" class="text-danger m-2"></div>

<form method="post">

    <h3 class="bg-danger text-white text-center p-2">Caution</h3>

    <h5 class="text-center m-2">
        Do you want to delete your account?
    </h5>
    <div class="text-center p-2">
        <button type="submit" class="btn btn-danger">Delete</button>
        <a asp-page="Dashboard" class="btn btn-secondary">Cancel</a>
    </div>
</form>
```

The view part of the page displays a simple warning and contains a form that sends a POST request. To define the page model class, add the code shown in Listing 9-28 to the UserDelete.cshtml.cs file. (You will have to create this file if you are using Visual Studio Code.)

Listing 9-28. The Contents of the UserDelete.cshtml.cs File in the Pages/Identity Folder

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using System.Threading.Tasks;

namespace IdentityApp.Pages.Identity {

    public class UserDeleteModel : UserPageModel {

        public UserDeleteModel(UserManager<IdentityUser> usrMgr,
                SignInManager<IdentityUser> signMgr) {
            UserManager = usrMgr;
            SignInManager = signMgr;
        }

        public UserManager<IdentityUser> UserManager { get; set; }
        public SignInManager<IdentityUser> SignInManager{ get; set; }

        public async Task<IActionResult> OnPostAsync() {
            IdentityUser idUser = await UserManager.GetUserAsync(User);
            IdentityResult result = await UserManager.DeleteAsync(idUser);
            if (result.Process(ModelState)) {
                await SignInManager.SignOutAsync();
                return Challenge();
            }
            return Page();
        }
    }
}
```

The account is deleted using the DeleteAsync method, just as in the administrator workflow with the addition that I use the SignOutAsync method defined by the SignInManager<IdentityUser> class to sign the user out of the application. This method can't be used to sign out other users in administration workflows, but it can be used for self-service workflows because it removes the authentication cookie from the response sent to the user.

Add the element shown in Listing 9-29 to create navigation for the new workflow.

Listing 9-29. Adding Navigation in the _Workflows.cshtml File in the Pages/Identity Folder

```
@model (string workflow, string theme)
@inject UserManager<IdentityUser> UserManager
@{
    Func<string, string> getClass = (string feature) =>
        feature != null && feature.Equals(Model.workflow) ? "active" : "";
    IdentityUser identityUser
        = await UserManager.GetUserAsync(User) ?? new IdentityUser();
}

<a class="btn btn-@Model.theme btn-block @getClass("Overview")" asp-page="Index">
    Overview
</a>

@if (await UserManager.HasPasswordAsync(identityUser)) {
    <a class="btn btn-@Model.theme btn-block @getClass("PasswordChange")"
            asp-page="UserPasswordChange">
        Change Password
    </a>
}
<a class="btn btn-@Model.theme btn-block @getClass("UserDelete")"
        asp-page="UserDelete">
    Delete Account
</a>
```

Restart ASP.NET Core, request `https://localhost:44350/Identity/Account/Login`, and sign in using the email `dora@example.com` and the password `mysecret`.

Click the email address shown in the header to navigate to the user dashboard and then click the Delete Account button. Click the Delete button, and the account will be removed from the user store, as shown in Figure 9-12. You can confirm the result by examining the administrator dashboard, which will show one fewer account in the user store.

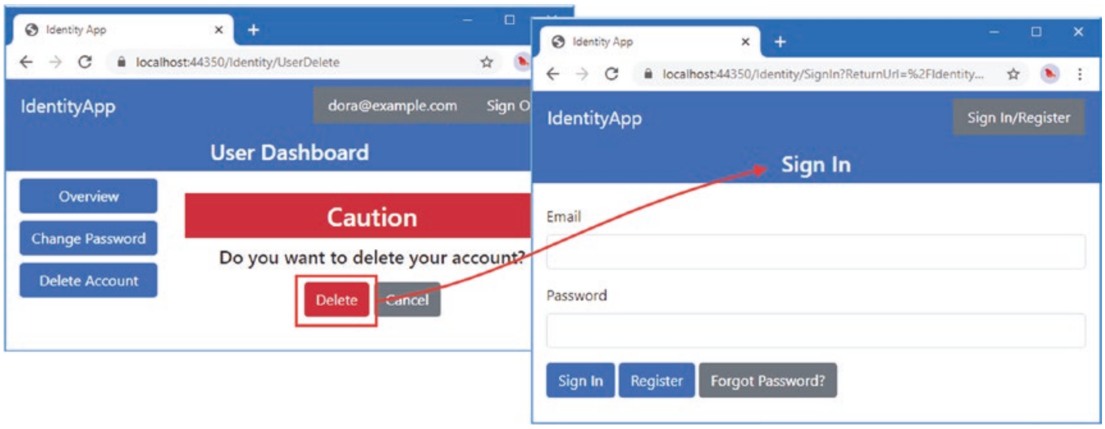


Figure 9-12. *Self-service account deletion*

Summary

In this chapter, I demonstrated administrator and self-service workflows for creating and deleting accounts. I also explained how lockouts work and how to force immediate sign-out by revalidating the cookie used to authenticate requests. In the next chapter, I describe the Identity API features for roles and claims.