CHAPTER 8

Signing In and Out and Managing Passwords

In this chapter, I describe the Identity API features for signing in and out of applications. Identity supports different ways of signing in, and I start with passwords in this chapter, including creating the workflows for managing passwords. In later chapters, I describe the other ways in which users can authenticate themselves. Table 8-1 puts the features for signing in and out and managing passwords in context.

Table 8-1. *Putting the Features for Signing In and Out and Managing Passwords in Context*

Question	Answer
What are they?	These API features are used to create workflows for signing the user into the application with a password and signing them out again when they have finished their session. These features are also used to manage passwords, both to set passwords administratively and to perform self-service password changes and password recovery.
Why are they useful?	Passwords are not the only way to authenticate with an Identity application, but they are the most widely used and are required by most projects.
How are they used?	Passwords are managed using methods provided by the `UserManager<IdentityUser>` class, which allows passwords to be added and removed from a user account. Users sign into and out of the application using methods defined by the sign-in manager class, `SignInManager<T>`.
Are there any pitfalls or limitations?	The sign-in process can be complex, especially if the project supports two-factor authentication and external authentication.
Are there any alternatives?	These features are built on the underlying ASP.NET Core platform, which you could use directly to achieve the same results. However, doing so would undermine the purpose of using Identity to manage users.

A. Freeman, *Pro ASP.NET Core Identity*, https://doi.org/10.1007/978-1-4842-6858-2_8

Table 8-2 summarizes the chapter.

Table 8-2. *Chapter Summary*

Problem	Solution	Listing
Manage passwords for a user account	Use the `HasPasswordAsync`, `RemovePasswordAsync`, and `HasPasswordAsync` methods defined by the user manager class.	1–3
Sign a user into the application with a password	Use the `PasswordSignInAsync` method defined by the sign-in manager class.	4, 5
Sign a user out of the application	Use the `SignOutAsync` method defined by the sign-in manager class.	6, 7
Configure ASP.NET Core to use the custom workflows for signing users in and out	Use the `ConfigureApplicationCookie` method to configure the `LoginPath`, `LogoutPath`, and `AccessDeniedPath` properties.	8–10
Get the `IdentityUser` object for the signed in user	Call the user manager's `GetUserAsync` method.	11
Support self-service password change	Call the user manager's `ChangePasswordAsync` method.	15–17
Support self-service passwords recovery	Call the user manager's `GeneratePasswordResetTokenAsync` to get a token is sent to the user. Validate the token and change the password with the `ResetPasswordAsync` method.	12–14, 18–23
Support administrator password changes	Use the `HasPasswordAsync`, `RemovePasswordAsync`, and `HasPasswordAsync` methosd defined by the user manager class or use the `GeneratePasswordResetTokenAsync` and `ResetPasswordAsync` methods to allow the user to select a new password.	24–26
Restrict access to resources to signed-in users	Apply the `Authorize` attribute, creating exceptions with the `AllowAnonymous` attribute.	27–32

Preparing for This Chapter

This chapter uses the `IdentityApp` project from Chapter 7. No changes are required for this chapter. Open a new PowerShell command prompt and run the commands shown in Listing 8-1 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 8-1. Resetting the Databases

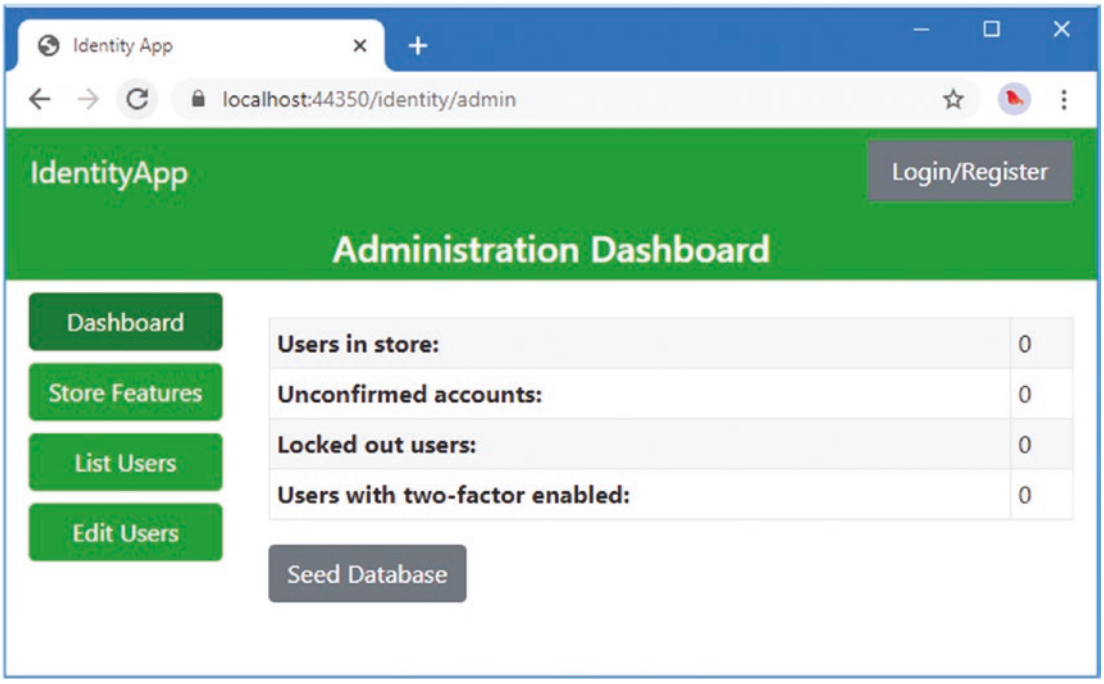
```
dotnet ef database drop --force --context ProductDbContext
dotnet ef database drop --force --context IdentityDbContext
dotnet ef database update --context ProductDbContext
dotnet ef database update --context IdentityDbContext
```

Use the PowerShell prompt to run the command shown in Listing 8-2 in the `IdentityApp` folder to start the application.

Listing 8-2. Running the Example Application

```
dotnet run
```

Open a web browser and request `https://localhost:44350/Identity/Admin`, which will show the administration dashboard, as shown in Figure 8-1.



Administration Dashboard

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

Figure 8-1. *Running the example application*

Adding Passwords to the Seed Data

The user accounts used to seed the user store won't be able to sign into the application because they have no credentials. Identity supports a range of authentication mechanisms and doesn't require `IdentityUser` objects to be created with any specific authentication data. The basic authentication model uses a password, and that is where I will start in this chapter before introducing other options in later chapters. Passwords are assigned using methods defined by the user class, as described in Table 8-3.

Table 8-3. *The UserManager<IdentityUser> Methods for Managing Passwords*

Name	Description
`HasPasswordAsync(user)`	This method returns `true` if the specific `IdentityUser` object has been assigned a password.
`AddPasswordAsync(user, password)`	This method adds a password to the store for the specified `IdentityUser`.
`RemovePasswordAsync(user)`	This method removes the password stored for the specified `IdentityUser` object.

In Listing 8-3, I have used the `AddPasswordAsync` method to set the same password for all the accounts used to seed the user store.

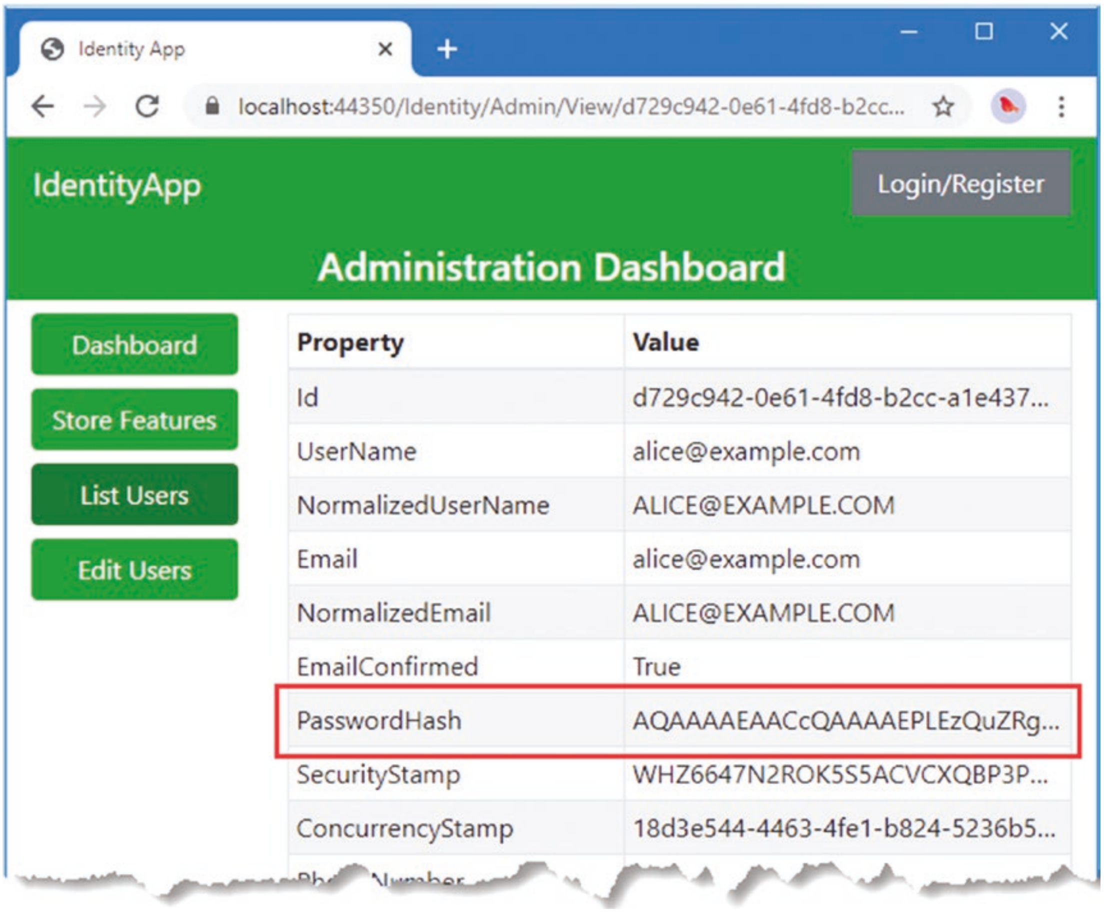
Listing 8-3. Setting Passwords in the Dashboard.cshtml.cs File in the Pages/Identity/Admin Folder

```
...
public async Task<IActionResult> OnPostAsync() {
    foreach (IdentityUser existingUser in UserManager.Users.ToList()) {
        IdentityResult result = await UserManager.DeleteAsync(existingUser);
        result.Process(ModelState);
    }
    foreach (string email in emails) {
        IdentityUser userObject = new IdentityUser {
            UserName = email,
            Email = email,
            EmailConfirmed = true
        };
        IdentityResult result = await UserManager.CreateAsync(userObject);
        if (result.Process(ModelState)) {
            result = await UserManager.AddPasswordAsync(userObject, "mysecret");
            result.Process(ModelState);
        }
    }
    if (ModelState.IsValid) {
        return RedirectToPage();
    }
    return Page();
}
...
```

I use the `AddPasswordAsync` method to give all of the test accounts the same password: `mysecret`. Later in this chapter, I create workflows for changing passwords, but this is enough to get started.

Restart ASP.NET Core and request `https://localhost:44350/identity/admin`. Click the Seed Database button, and the test accounts will be added to the user store, including a password.

Passwords are stored as hash codes, which means that having access to the database doesn't reveal a user's password. I describe the process by which passwords are converted into hash codes in detail in Part 2, but you can see the result by clicking the List Users button and clicking the View button for the alice@example.com account. The `PasswordHash` property has been assigned a value, as shown in Figure 8-2.



Property	Value
Id	d729c942-0e61-4fd8-b2cc-a1e437...
UserName	alice@example.com
NormalizedUserName	ALICE@EXAMPLE.COM
Email	alice@example.com
NormalizedEmail	ALICE@EXAMPLE.COM
EmailConfirmed	True
PasswordHash	AQAAAAEAACcQAAAAEPLEzQuZRg...
SecurityStamp	WHZ6647N2ROK5S5ACVCXQBP3P...
ConcurrencyStamp	18d3e544-4463-4fe1-b824-5236b5...

Figure 8-2. *Storing passwords*

Signing In, Signing Out, and Denying Access

The three most fundamental user-facing features are the ability to sign in to the application, the ability to sign out again, and the ability to display an error message when the user requests content for which they are not authorized. In the sections that follow, I create these essential workflows and configure the application to make use of them.

Signing into the Application

When signing into the application, the user provides credentials that are compared to the data in the user store. If the credentials match the stored data, then a cookie is added to the response that securely identifies the user in subsequent requests.

ASP.NET Core uses the `ClaimsPrincipal` class to represent the signed-in user. I describe this class in more detail in later chapters, but for now, it is enough to know that the sign-in process obtains an `IdentityUser` object from the store and uses the data that it contains to create a `ClaimsPrincipal` object that can be used by ASP.NET Core.

Evaluating the user's credentials and creating the `ClaimsPrincipal` object are the responsibilities of the sign-in manager class, `SignInManager<IdentityUser>`, which is configured as a service when Identity is configured.

Add a Razor Page named `SignIn.cshtml` to the `Pages/Identity` folder with the content shown in Listing 8-4.

Listing 8-4. The Contents of 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>
</form>
```

The user is prompted to enter their email address and password into a form that sends a POST request. There are elements to display model validation errors, and I have added a `div` element styled as an alert so I can display messages using the ASP.NET Core temp data feature.

As you have seen in earlier chapters, when an unauthenticated user requests content that is protected, ASP.NET Core sends a challenge response, which leads to the user being prompted to sign in. As part of this process, the URL that the user requested is provided as a query string parameter named `returnUrl`, which allows the browser to be redirected back to the content after a successful sign-in. The page directive in Listing 8-4 defines a route parameter named `returnUrl` to capture this URL.

To define the page model class, add the code shown in Listing 8-5 to the `SignIn.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 8-5. The Contents of 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;

namespace IdentityApp.Pages.Identity {

    public class SignInModel : UserPageModel {

        public SignInModel(SignInManager<IdentityUser> signMgr)
            => SignInManager = signMgr;

        public SignInManager<IdentityUser> SignInManager { 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) {
                    TempData["message"] = "Sign In Not Allowed";
                } else if (result.RequiresTwoFactor) {
                    return RedirectToPage("SignInTwoFactor", new { ReturnUrl });
                } else {
                    TempData["message"] = "Sign In Failed";
                }
            }
            return Page();
        }
    }
}
```

The page model receives a `SignInManager<IdentityUser>` object through its constructor and uses the `PasswordSignInAsync` method to sign the user into the application. For quick reference, Table 8-4 describes this method.

Table 8-4. The SignInManager<IdentityUser> Method for Password SignIns

Name	Description
`PasswordSignInAsync(username, password, persist, lockout)`	This method signs the user into the application with the specified username and password. The `persist` argument specifies whether the authentication cookie persists after the browser is closed. The `lockout` argument specifies whether a failed sign-in attempt counts toward a lockout, as described in Chapter 9. There is also a version of this method that accepts an `IdentityUser` object instead of a username.

The result from the `PasswordSignInAsync` is an instance of the `SignInResult` class defined in the `Microsoft.AspNetCore.Identity` namespace. There is also a `SignInResult` class defined in the `Microsoft.AspNetCore.Mvc` namespace, so disambiguation is required, like this:

```
...
using SignInResult = Microsoft.AspNetCore.Identity.SignInResult;
...
```

The `SignInResult` describes the sign attempt using the properties described in Table 8-5.

Table 8-5. The SignInResult Properties

Name	Description
`Succeeded`	This property returns `true` if the user has been successfully signed into the application.
`IsLockedOut`	This property returns `true` if the user is currently locked out. See Chapter 9 for details of how lockouts work.
`RequiresTwoFactor`	This property returns `true` if two-factor authentication is required. See Chapter 11 for details of creating workflows for two-factor authentication.
`IsNotAllowed`	This property returns `true` if the user is not allowed to sign in. This is most commonly the case when the email address has not been confirmed. I create a workflow that supports confirmations in Chapter 9.

If the `Succeeded` property is `false`, you can check the other properties to see if any follow-up action is required, such as prompting the user for two-factor authentication. If all the `SignInResult` properties are `false`, then the user hasn't provided a valid email address or password.

If the `Succeeded` property is `true`, then the user has been signed into the application and can be redirected back to the URL that triggered the challenge response, like this:

```
...
if (result.Succeeded) {
    return RedirectToPage(ReturnUrl ?? "/");
} else if (result.IsLockedOut) {
...
```

The other outcomes are handled by displaying an error message to the user, except when the user requires two-factor authentication. For this outcome, I perform a redirection to a Razor Page named `SignInTwoFactor` that I will create in Chapter 11.

Signing Out of the Application

Signing out of the application allows the user to explicitly terminate their session. Add a Razor Page named `SignOut.cshtml` to the `Pages/Identity` folder with the content shown in Listing 8-6.

Listing 8-6. The Contents of the SignOut.cshtml File in the Pages/Identity Folder

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

@if (User.Identity.IsAuthenticated) {
    <form method="post">
        <div class="text-center">
            <h6>Click the button to sign out of the application</h6>
            <button type="submit" class="btn btn-secondary">
                Sign Out
            </button>
        </div>
    </form>
} else {
    <div class="text-center">
        <h6>You are signed out of the application</h6>
        <a asp-page="SignIn" asp-route-returnUrl="" class="btn btn-secondary">
            OK
        </a>
    </div>
}
```

The view part of the page checks to see if there is an authenticated user by reading the `User.Identity.IsAuthenticated` property. I explain how this property works in Part 2, but for now, it is enough to know that it returns `true` if the current request is authenticated.

If there is an authenticated user, then the view presents a page that contains a form, allowing the user to sign out. If there is no authenticated user, then a message indicating the user is signed out is displayed. This is important because the browser should be redirected during the sign-out process to ensure that the cookie that authenticates the user is removed.

To define the page model class, add the code shown in Listing 8-7 to the `SignOut.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 8-7. The Contents of the SignOut.cshtml.cs File in the Pages/Identity Folder

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

namespace IdentityApp.Pages.Identity {

    public class SignOutModel : UserPageModel    {

        public SignOutModel(SignInManager<IdentityUser> signMgr)
            => SignInManager = signMgr;

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

        public async Task<IActionResult> OnPostAsync() {
            await SignInManager.SignOutAsync();
            return RedirectToPage();
        }
    }
}
```

Signing out of the application is done using the sign-in manager's SignOutAsync method. Once the user is signed out, I call the RedirectToPage method that will cause the browser to send a GET request to the SignOut page and to display the message confirming the user is signed out. This is the redirection that ensures the authentication cookie is deleted. For quick reference, Table 8-6 describes the method used to sign users out of the application.

Table 8-6. *The SignInManager<IdentityUser> Method for Signing Out*

Name	Description
SignOutAsync()	This method signs the current user out of the application.

Creating the Forbidden Page

Add a page named Forbidden.cshtml to the Pages/Identity folder with the content shown in Listing 8-8. This is the page that will be displayed when the user requests a URL for which they are not authorized.

Listing 8-8. The Contents of the Forbidden.cshtml File in the Pages/Identity Folder

```
@page
@model IdentityApp.Pages.Identity.ForbiddenModel
@{
    ViewData["showNav"] = false;
    ViewData["banner"] = "Access Denied";
}

<h6 class="text-center">You do not have access to this content</h6>
```

No page model code is required for this page, which just displays the message to the user.

Configuring the Application

Now that I have custom pages for signing in and out, I need to make some configuration changes to integrate them into the application. Listing 8-9 updates the navigation links displayed in the layout header to use the new pages. (When updating this file, take care to remove the `asp-area` attributes, which were used to select pages from the ASP.NET Core area created by the Identity UI package.)

Listing 8-9. Updating Links in the _LoginPartial.cshtml File in the Views/Shared Folder

```
<nav class="nav">
    @if (User.Identity.IsAuthenticated) {
        <a asp-page="/Identity/Index" class="nav-link bg-secondary text-white">
                @User.Identity.Name
        </a>
        <a asp-page="/Identity/SignOut" class="nav-link bg-secondary text-white">
            Sign Out
        </a>
    } else {
        <a asp-page="/Identity/SignIn" class="nav-link bg-secondary text-white">
            Sign In/Register
        </a>
    }
</nav>
```

I also need to update the ASP.NET Core configuration. By default, ASP.NET Core will use the `/Account/Login` and `/Account/Logout` URLs for signing in and out of the application. I could have used the routing system to ensure that my new Razor Pages will receive requests to these URLs, but I have chosen to change the URLs that ASP.NET Core uses instead, as shown in Listing 8-10.

Listing 8-10. Configuring URLs in the Startup.cs File in the ExampleApp Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddControllersWithViews();
    services.AddRazorPages();
    services.AddDbContext<ProductDbContext>(opts => {
        opts.UseSqlServer(
            Configuration["ConnectionStrings:AppDataConnection"]);
    });

    services.AddHttpsRedirection(opts => {
        opts.HttpsPort = 44350;
    });

    services.AddDbContext<IdentityDbContext>(opts => {
        opts.UseSqlServer(
            Configuration["ConnectionStrings:IdentityConnection"],
            opts => opts.MigrationsAssembly("IdentityApp")
        );
    });
    services.AddScoped<IEmailSender, ConsoleEmailSender>();

    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>();

    services.AddAuthentication()
        .AddFacebook(opts => {
            opts.AppId = Configuration["Facebook:AppId"];
            opts.AppSecret = Configuration["Facebook:AppSecret"];
        })
        .AddGoogle(opts => {
            opts.ClientId = Configuration["Google:ClientId"];
            opts.ClientSecret = Configuration["Google:ClientSecret"];
        })
        .AddTwitter(opts => {
            opts.ConsumerKey = Configuration["Twitter:ApiKey"];
            opts.ConsumerSecret = Configuration["Twitter:ApiSecret"];
        });

    services.ConfigureApplicationCookie(opts => {
        opts.LoginPath = "/Identity/SignIn";
        opts.LogoutPath = "/Identity/SignOut";
        opts.AccessDeniedPath = "/Identity/Forbidden";
    });
}
...
```

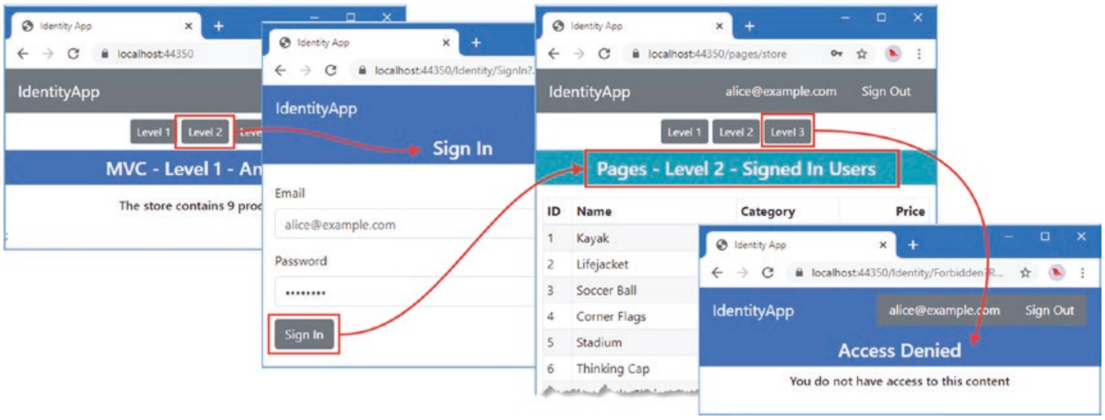
The ConfigureApplicationCookie extension method is provided by Identity and can be used to override the default settings by assigning new values to the properties defined by the CookieAuthenticationOptions class. The properties used in Listing 8-10 are described in Table 8-7, and I explain the role of the CookieAuthenticationOptions class in Chapter 14.

Table 8-7. *The CookieAuthenticationOptions Used to Select URLs*

Name	Description
LoginPath	This property is used to specify the URL to which the browser is directed following a challenge response so the user can sign into the application.
LogoutPath	This property is used to specify the URL to which the browser is directed so the user can sign into the application.
AccessDeniedPath	This property is used to specify the URL to which the browser is directed following a forbidden response, indicating that the user does not have access to the requested content.

Restart ASP.NET Core and request `https://localhost:44350`. Click the Level 2 button to request content that is available only for authenticated users. This produces a challenge response that leads to the sign-in page, as shown in Figure 8-3. Sign in to the application using `alice@example.com` as the email address and `mysecret` as the password.

Click the Sign In button, and you will be signed into the application and redirected to the protected content. Click the Level 3 button to request content that is only available to users who have been assigned to a specific role. This type of request will lead to a forbidden response until I add support for roles in Chapter 10, producing the access denied message, as shown in Figure 8-3.


Identity App
localhost:44350
IdentityApp
Level 1
Level 2
MVC - Level 1 - An
The store contains 9 proc
localhost:44350/Identity/SignIn?...
Sign In
Email
alice@example.com
Password
••••••••
Sign In
localhost:44350/pages/store
alice@example.com
Sign Out
Level 3
Pages - Level 2 - Signed In Users
localhost:44350/Identity/Forbidden?R...
Access Denied
You do not have access to this content

ID	Name	Category	Price
1	Kayak		
2	Lifejacket		
3	Soccer Ball		
4	Corner Flags		
5	Stadium		
6	Thinking Cap		

Figure 8-3. *Signing into the application with a password and triggering a forbidden response*

Click the Sign Out link in the header, click the Sign Out button, and you will be signed out of the application. Click the OK button, and you will navigate back to the sign-in page, as shown in Figure 8-4.

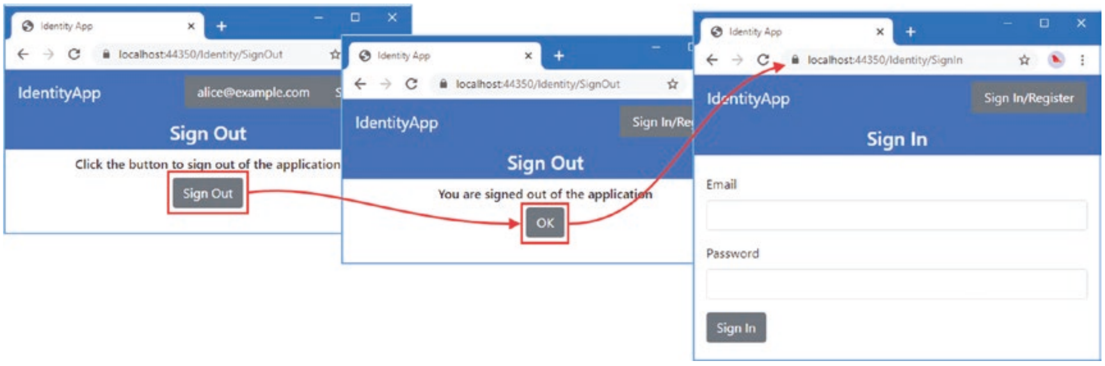

Identity App
localhost:44350/Identity/SignOut
IdentityApp
alice@example.com
Sign Out
Click the button to sign out of the application
Sign Out
Sign In/Re
Sign Out
You are signed out of the application
OK
localhost:44350/Identity/SignIn
Sign In/Register
Sign In
Email
Password
Sign In

Figure 8-4. *Signing out of the application*

Completing the User Dashboard

Now that users can sign into the application, I can return to the user dashboard and complete the initial features. Listing 8-11 shows the changes required to the `Index` page model class to display data for the current user.

Listing 8-11. Displaying Data in the Index.cshtml File in the Pages/Identity Folder

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

namespace IdentityApp.Pages.Identity {

    public class IndexModel : UserPageModel {

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

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

        public string Email { get; set; }
        public string Phone { get; set; }

        public async Task OnGetAsync() {
            IdentityUser CurrentUser = await UserManager.GetUserAsync(User);
            Email = CurrentUser?.Email ?? "(No Value)";
            Phone = CurrentUser?.PhoneNumber ?? "(No Value)";
        }
    }
}
```

As noted earlier, ASP.NET Core uses a `ClaimsPrincipal` object to represent the currently authenticated user, which is accessed through the `User` property provided by the base classes for page models and controllers. The user manager class provides the `GetUserAsync` method, which obtains the `IdentityUser` object from the user store that represents the signed-in user, like this:

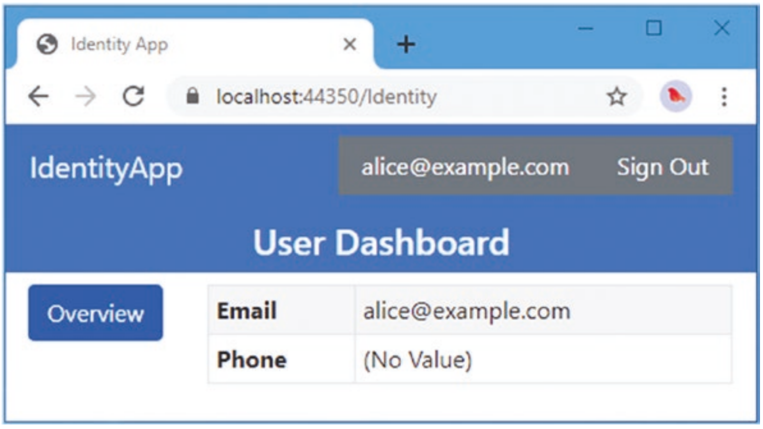
```
...
IdentityUser CurrentUser = await UserManager.GetUserAsync(User);
...
```

This provides a helpful bridge between the objects that the ASP.NET Core platform and ASP.NET Core Identity use to represent users and is useful for self-service workflows. Table 8-8 summarizes the user manager method used in Listing 8-11 for quick reference.

Table 8-8. *The UserManager<IdentityUser> Method for Obtaining the Current User Object*

Name	Description
`GetUserAsync(principal)`	This method returns the `IdentityUser` object that has been stored for the specified `ClaimsPrincipal` object, which is most often obtained through the `User` property defined by the base classes for page models and controllers.

Restart ASP.NET Core, and make sure you are signed in as `alice@example.com`. Click the email address displayed in the header, and you will see the user dashboard, which has been populated with the data for the signed-in user, as shown in Figure 8-5.



Email	alice@example.com
Phone	(No Value)

Figure 8-5. *Displaying data for the currently signed-in user*

Managing Passwords

If an application supports signing in with passwords, then workflows are required to manage them. In the sections that follow, I show you how to support the administrator setting a password, the administrator forcing the user to choose a new password, and the user changing their password.

Preparing the Email Confirmation Service

Some workflows require the user to click a link they receive in an email, either to confirm they control the email address during registration or to confirm they initiated an operation, such as password changes. Dealing with these confirmation emails is easier if they are created using a service because it means that confirmations can be sent from any Razor Page or controller and that all of the emails will be consistent.

The emails sent to the user contain confirmation tokens, and these tokens must be encoded so they can be safely sent as part of a URL and decoded again when the user clicks the link in the email. Add a file named `TokenUrlEncoderService.cs` to the `Services` folder and add the code shown in Listing 8-12.

Listing 8-12. The Contents of the TokenUrlEncoderService.cs File in the Services Folder

```
using Microsoft.AspNetCore.WebUtilities;
using System.Text;

namespace IdentityApp.Services {

    public class TokenUrlEncoderService {

        public virtual string EncodeToken(string token)
            => WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(token));

        public virtual string DecodeToken(string urlToken)
            => Encoding.UTF8.GetString(WebEncoders.Base64UrlDecode(urlToken));
    }
}
```

I have used Base64 encoding, which ensures that tokens can be encoded so they contain only characters that are allowed in URLs. Next, add a file named `IdentityEmailService.cs` to the `Services` folder and add the code shown in Listing 8-13.

Listing 8-13. The Contents of 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;

namespace IdentityApp.Services {

    public class IdentityEmailService {

        public IdentityEmailService(IEmailSender sender,
                UserManager<IdentityUser> userMgr,
                IHttpContextAccessor contextAccessor,
                LinkGenerator generator,
                TokenUrlEncoderService encoder) {
            EmailSender = sender;
            UserManager = userMgr;
            ContextAccessor = contextAccessor;
            LinkGenerator = generator;
            TokenEncoder = encoder;
        }

        public IEmailSender EmailSender { get; set; }
        public UserManager<IdentityUser> UserManager { get; set;}
        public IHttpContextAccessor ContextAccessor { get; set; }
        public LinkGenerator LinkGenerator { get; set; }
        public TokenUrlEncoderService TokenEncoder { get; set; }

        private string GetUrl(string emailAddress, string token, string page) {
            string safeToken = TokenEncoder.EncodeToken(token);
            return LinkGenerator.GetUriByPage(ContextAccessor.HttpContext, page,
                null, new { email = emailAddress, token = safeToken});
        }
    }
}
```

The `IdentityEmailService` class declares dependencies on the services it requires: the `TokenUrlEncoderService` defined in Listing 8-13, the user manager class, the `IHttpContextAccessor` service (which is used to access the `HttpContext` object outside of a Razor Page or controller), and the `IEmailSender` service, which is used to send emails.

The `GetUrl` method accepts an email address, a confirmation token, and a page, and it returns a URL that the user can click. I will add methods to the `IdentityEmailService` class to send specific emails as I create workflows that require confirmations. Add the statements shown in Listing 8-14 to the `Startup` class to set up the classes defined in Listing 8-12 and Listing 8-13 as services.

Listing 8-14. Configuring Services in the Startup.cs File in the IdentityApp Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddControllersWithViews();
    services.AddRazorPages();
    services.AddDbContext<ProductDbContext>(opts => {
        opts.UseSqlServer(
            Configuration["ConnectionStrings:AppDataConnection"]);
    });

    services.AddHttpsRedirection(opts => {
        opts.HttpsPort = 44350;
    });

    services.AddDbContext<IdentityDbContext>(opts => {
        opts.UseSqlServer(
            Configuration["ConnectionStrings:IdentityConnection"],
            opts => opts.MigrationsAssembly("IdentityApp")
        );
    });

    services.AddScoped<IEmailSender, ConsoleEmailSender>();

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

    // ...statements omitted for brevity...
}
...
```

Listing 8-14 also adds the `AddDefaultTokenProviders` method to the chain of calls that set up Identity. This method sets up the services that are used to generate the confirmation tokens sent to users, which I describe in detail in Part 2.

Performing Self-Service Password Changes

The simplest password workflow is a self-service change, where the user provides their current password and a new password. Add a Razor Page named `UserPasswordChange.cshtml` to the `Pages/Identity` folder with the content shown in Listing 8-15.

Listing 8-15. The Contents of the UserPasswordChange.cshtml File in the Pages/Identity Folder

```
@page
@model IdentityApp.Pages.Identity.UserPasswordChangeModel
@{
    ViewBag.Workflow = "PasswordChange";
}

<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>Current Password</label>
        <input class="form-control" name="current" type="password" />
    </div>
    <div class="form-group">
        <label>New Password</label>
        <input class="form-control" name="newpassword" type="password" />
    </div>
        <div class="form-group">
        <label>Confirm New Password</label>
        <input class="form-control" name="confirmpassword" type="password" />
    </div>
    <button class="btn btn-primary">Change Password</button>
    <a asp-page="Index" class="btn btn-secondary">Cancel</a>
</form>
```

The view part of the page displays a form into which the user enters their current and new passwords. There is an `@if` expression that will display a `div` element if there is a temp data property named `message`, which I will use to indicate the password has been successfully changed. To define the page model class, add the code shown in Listing 8-16 to the `UserPasswordChange.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 8-16. The Contents of the UserPasswordChange.cshtml.cs File in the Pages/Identity Folder

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

namespace IdentityApp.Pages.Identity {

    public class PasswordChangeBindingTarget {
        [Required]
        public string Current { get; set; }

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

    public class UserPasswordChangeModel : UserPageModel {

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

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

        public async Task<IActionResult> OnPostAsync(
                PasswordChangeBindingTarget data) {
            if (ModelState.IsValid) {
                IdentityUser user = await UserManager.GetUserAsync(User);
                IdentityResult result = await UserManager.ChangePasswordAsync(user,
                    data.Current, data.NewPassword);
                if (result.Process(ModelState)) {
                    TempData["message"] = "Password changed";
                    return RedirectToPage();
                }
            }
            return Page();
        }
    }
}
```

Password changes are done using the user manager's ChangePasswordAsync method, which accepts an IdentityUser object, the current password, and the new password. The new password is validated against the password options described in Chapter 5, and if they match, the new password is stored. Since this is a self-service feature, I get the IdentityUser object using the GetUserAsync. The outcome of the password change is described using an IdentityResult object, which is processed to add errors to the model validation dictionary. If the password is successfully changed, a message is added to the temp data dictionary so it can be displayed to the user.

Listing 8-17 shows the changes required to integrate the new feature into the user dashboard.

Listing 8-17. 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>
}
```

Not all users sign into the application using a password, so I use dependency injection to obtain a user manager object so that I can use the `HasPasswordAsync` method to determine if a navigation button for the password change page should be shown.

For quick reference, Table 8-9 describes the user manager methods used to implement self-service password changes.

Table 8-9. *The UserManager<IdentityUser> Methods for Self-Service Password Changes*

Name	Description
`ChangePasswordAsync(user, current, new)`	This method changes the password for the specified `IdentityUser` object. This method requires the existing password. If the user cannot provide the password, then password recovery is required, as described in the next section.
`GetUserAsync(principal)`	This method returns the `IdentityUser` object that has been stored for the specified `ClaimsPrincipal` object, which is most often obtained through the `User` property defined by the base classes for page models and controllers.
`HasPasswordAsync(user)`	This method returns `true` if the user store contains a password for the specified `IdentityUser` object and is used to ensure that only users with passwords are provided with the tools to change them.

Restart ASP.NET Core and make sure you are signed in to the application as `alice@example.com` using the password `mysecret`. Request `https://localhost:44350/Identity` and click the Change Password button. Enter mysecret into the password field and mysecret2 into the New Password and Confirm New Password fields. Click the Change Password button; the user store will be updated, and a confirmation message will be displayed, as shown in Figure 8-6.

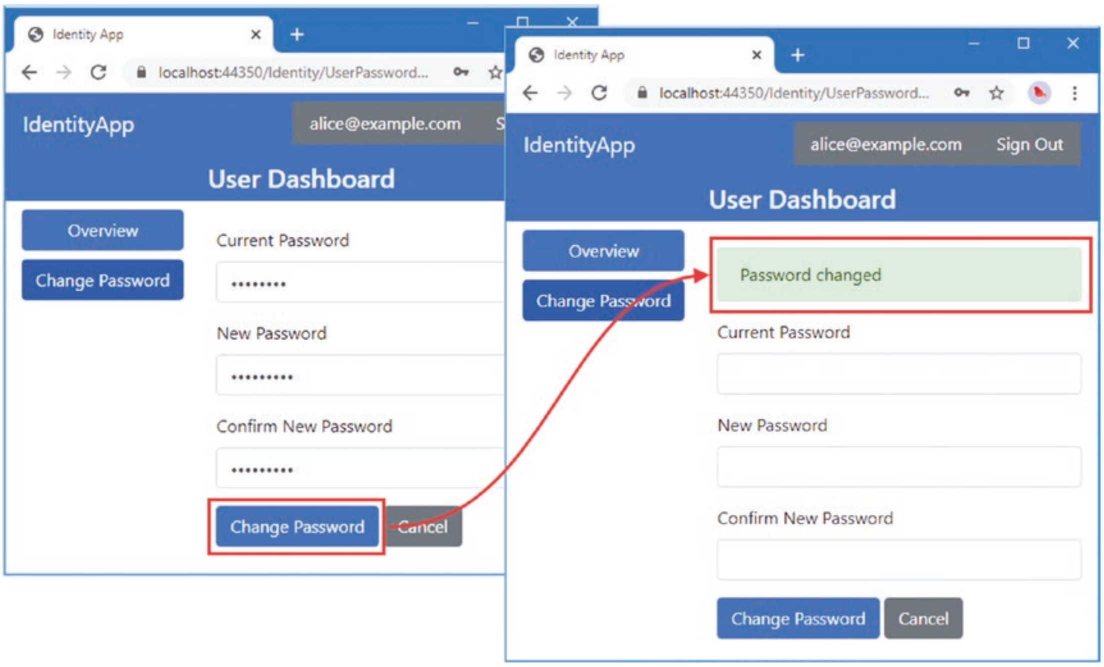


Figure 8-6. *Performing a self-service password change*

Performing Self-Service Password Recovery

Password recovery is used when the user has forgotten their password. This workflow requires the user to click a link that is emailed to them to confirm they are the owner of the account. To define the email that the user will receive, add the method shown in Listing 8-18 to the IdentityEmailService class.

Listing 8-18. 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 {

        // ...statements omitted for brevity...

        private string GetUrl(string emailAddress, string token, string page) {
            string safeToken = TokenEncoder.EncodeToken(token);
            return LinkGenerator.GetUriByPage(ContextAccessor.HttpContext, page,
                null, new { email = emailAddress, token = safeToken});
        }
        public async Task SendPasswordRecoveryEmail(IdentityUser user,
                string confirmationPage) {
            string token = await UserManager.GeneratePasswordResetTokenAsync(user);
            string url = GetUrl(user.Email, token, confirmationPage);
            await EmailSender.SendEmailAsync(user.Email, "Set Your Password",
                $"Please set your password by <a href={url}>clicking here</a>.");
        }
    }
}
```

The link sent to the user contains a confirmation token, which is used by Identity to prevent other users from recovering an account password. Confirmation tokens are generated by methods defined by the user manager class, and the `GeneratePasswordResetTokenAsync` method generates a token that can be used in password recovery workflows. I describe how tokens are generated and validated in Part 2, but for now, it is enough to know that tokens are generated for a specific purpose, such as password recovery, to prevent them from being misused.

Add a Razor Page named `UserPasswordRecovery.cshtml` to the `Pages/Identity` folder with the content shown in Listing 8-19.

Listing 8-19. The Contents of the UserPasswordRecovery.cshtml File in the Pages/Identity Folder

```
@page
@model IdentityApp.Pages.Identity.UserPasswordRecoveryModel
@{
    ViewData["showNav"] = false;
    ViewData["banner"] = "Password Recovery";
}

<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 Address</label>
        <input class="form-control" name="email" />
    </div>
    <button class="btn btn-primary">Send Recovery Email</button>
    <a asp-page="SignIn" class="btn btn-secondary">Cancel</a>
</form>
```

The user is presented with a form into which they are able to enter their email address so that a confirmation email can be sent. To create the page model class, add the code shown in Listing 8-20 to the `UserPasswordRecovery.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 8-20. The Contents of the UserPasswordRecovery.cshtml.cs File in the Pages/Identity Folder

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

namespace IdentityApp.Pages.Identity {

    public class UserPasswordRecoveryModel : UserPageModel {

        public UserPasswordRecoveryModel(UserManager<IdentityUser> usrMgr,
                IdentityEmailService emailService) {
            UserManager = usrMgr;
            EmailService = emailService;
        }

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

        public async Task<IActionResult> OnPostAsync([Required]string email) {
            if (ModelState.IsValid) {
                IdentityUser user = await UserManager.FindByEmailAsync(email);
                if (user != null) {
                    await EmailService.SendPasswordRecoveryEmail(user,
                        "UserPasswordRecoveryConfirm");
                }
                TempData["message"] = "We have sent you an email. "
                    + " Click the link it contains to choose a new password.";
                return RedirectToPage();
            }
            return Page();
        }
    }
}
```

The POST handler method receives the user's email address and retrieves the IdentityUser object from the user store, which is then passed to the IdentityEmailService object received through the constructor so that an email can be sent.

To define the page that will receive the request when the user clicks the email link, add a Razor Page named UserPasswordRecoveryConfirm.cshtml to the Pages/Identity folder with the content shown in Listing 8-21.

Listing 8-21. The Contents of the UserPasswordRecoveryConfirm.cshtml File in the Pages/Identity Folder

```
@page "{email?}/{token?}"
@model IdentityApp.Pages.Identity.UserPasswordRecoveryConfirmModel
@{
    ViewData["showNav"] = false;
    ViewData["banner"] = "Password Recovery";
}
```

```
@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>
        <a asp-page="UserPasswordRecovery" class="btn btn-primary">Resend Email</a>
    </div>
} else {
    <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">
        <input type="hidden" asp-for="Token" />
        <div class="form-group">
            <label>Email</label>
            <input class="form-control" asp-for="Email" />
        </div>
        <div class="form-group">
            <label>New 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">Set Password</button>
    </form>
}
```

The view part of the page displays a form that allows the user to select a new password. The URL contained in the email will contain `email` and `token` routing parameters, and there is a message that tells the user to check their email that is displayed if either parameter is missing, which will be the case if the user requests the page directly rather than clicking the email link. To define the page model class, add the code shown in Listing 8-22 to the `UserPasswordRecoveryConfirm.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 8-22. The Contents of the UserPasswordRecoveryConfirm.cshtml.cs File in the Pages/Identity Folder

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

namespace IdentityApp.Pages.Identity {

    public class UserPasswordRecoveryConfirmModel : UserPageModel {

        public UserPasswordRecoveryConfirmModel(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)) {
                    TempData["message"] = "Password changed";
                    return RedirectToPage();
                }
            }
            return Page();
        }
    }
}
```

The email and token included in the confirmation email link are assigned to the `Email` and `Token` properties with the `BindProperty` attribute. When the POST handler method is invoked, it decodes the token from its URL-safe form and passes it to the user manager's `ResetPasswordAsync` method, along with the `IdentityUser` object that represents the user and the user's new choice of password. The outcome of this method is described with an `IdentityResult` object, and a message is displayed to the user if the password recovery operation worked. Table 8-10 describes the user manager methods used in password recovery.

Table 8-10. *The UserManager<IdentityUser> Methods for Password Recovery*

Name	Description
`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.

Add the link shown in Listing 8-23 to the `SignIn` page to present the password recovery feature to the user.

Listing 8-23. Adding Navigation in the SignIn.cshtml File in the Pages/Identity Folder

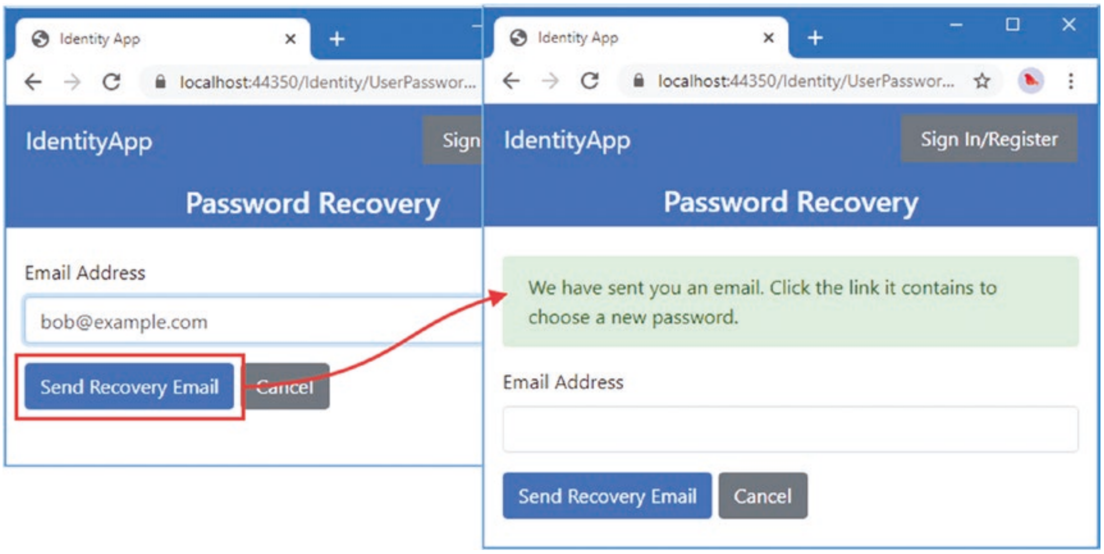
```
@page "{returnUrl?}"
@model IdentityApp.Pages.Identity.SignInModel
@{
    ViewData["showNav"] = false;
    ViewData["banner"] = "Sign In";
}

<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="UserPasswordRecovery" class="btn btn-secondary">Forgot Password?</a>
</form>
```

Restart ASP.NET Core and request `https://localhost:44350/Identity/SignIn`. Click the Forgot Password? button, enter bob@example.com into the text field, and click the Send Recovery Email button. You will receive a message confirming that a recovery email has been sent, as shown in Figure 8-7.



We have sent you an email. Click the link it contains to choose a new password.

Figure 8-7. *Starting the password recovery process*

The application is configured to use the implementation of the IEmailSender interface I created for the Identity UI package, which simulates sending emails by writing messages to the console. Examine the ASP.NET Core output, and you will see a message similar to this one:

```
---New Email----
To: bob@example.com
Subject: Set Your Password
Please set your password by <a href=https://localhost:44350/Identity/UserPasswordRecovery
Confirm/bob@example.com/Q2ZESjhBMVB3bFFBQ3gxRmhXTERPQzZ1UTVoTVQvWEFJZHMxbOhpS1N3RlA1ZmpX
YXZvd3k3QU03UENrS3dSbDMyU2ZQem41NWRJbOV5d3FWWHRqQ2YrR2dzWHBkaW81a3NsOXdlWnJnRko5Zo
d3OWE5ZTBYYlZOSzZtTEowYTc3UFNZTVI1VFN4WkhLdWYOOTRCQ2ooS2ROY2hrRWFMdU1BTXRBSDUxZ3lxcjFH
dXZaNSsrTFFOZ1NFREtIQTlwZ1hxcHFIMWO4L3VObGYxSzg3TnBabmhocExSM1NhOTNJaO5IODVSbWtscEYyRTl
4>clicking here</a>.
-------
```

Copy the URL from the message (which will have a different token to the one shown here), and you will be prompted to select a new password. Enter mynewpassword into both text fields and click the Set Password button. A message will be displayed confirming the password change, as shown in Figure 8-8. Click the SignIn/Register link at the top of the page, and you will be able to sign in to the application with the new password.

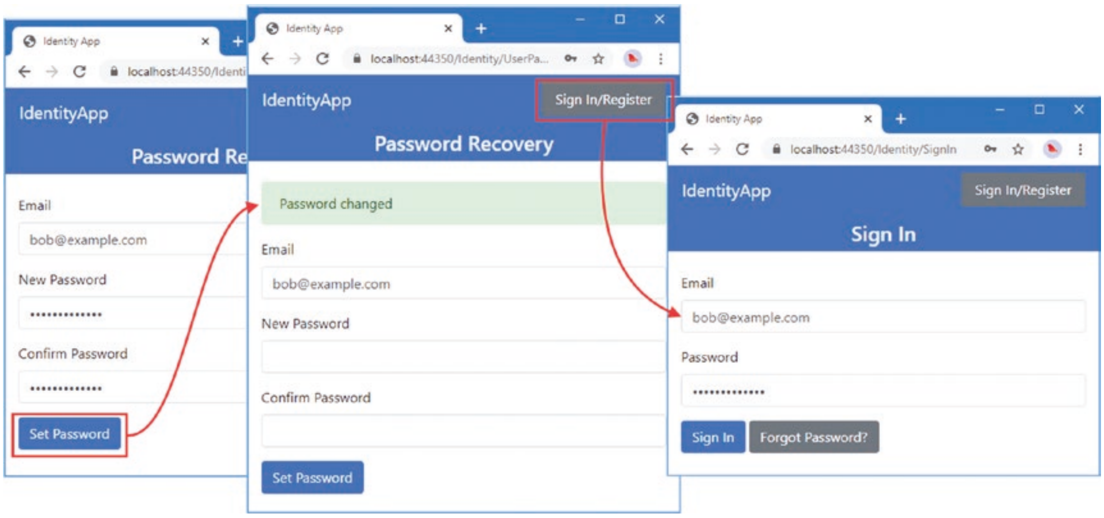


Figure 8-8. Completing password recovery

Performing Administrator Password Changes

The features used for self-service password management can also be used to provide administrator workflows. Add a Razor Page `Passwords.cshtml` to the `Pages/Identity/Admin` folder with the content shown in Listing 8-24.

Listing 8-24. The Contents of the Passwords.cshtml File in the Pages/Identity/Admin Folder

```
@page "{Id?}"
@model IdentityApp.Pages.Identity.Admin.PasswordsModel
@{
    ViewBag.Workflow = "Passwords";
}

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

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

<div class="container-fluid">
    <div class="row">
        <div class="col p-1">
            <div asp-validation-summary="All" class="text-danger m-2"></div>
            <form method="post" asp-page-handler="setPassword" class="pb-2">
                <input type="hidden" asp-for="Id" />
                <div class="form-group">
                    <label>New Password</label>
                    <input class="form-control" name="password" type="password" />
                </div>
                <div class="form-group">
                    <label>Confirm Password</label>
                    <input class="form-control" name="confirmation"
                            type="password" />
                </div>
                <button class="btn btn-secondary">Set Password</button>
            </form>
            @if (await Model.UserManager.IsEmailConfirmedAsync(Model.IdentityUser)) {
                <form method="post" asp-page-handler="userChange">
                    <input type="hidden" asp-for="Id" />
                    <button class="btn btn-secondary mt-2">
                        Send User Reset Email
                    </button>
                </form>
            }
        </div>
    </div>
</div>
```

The view part of the page displays two forms. The first allows a new password to be specified directly, which is the conventional approach for setting passwords, but which requires the administrator to communicate the new password to the user. The second form is used to send a password reset email to the user, which will allow them to choose their password, but this can be done only if the user has a confirmed email address. All the test accounts used to seed the user store have confirmed email addresses, but I create a workflow for confirming email addresses in Chapter 9.

To create the page model class, add the code shown in Listing 8-25 to the `Passwords.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 8-25. The Contents of the Passwords.cshtml.cs File in the Pages/Identity/Admin Folder

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

namespace IdentityApp.Pages.Identity.Admin {

    public class PasswordsModel : AdminPageModel {

        public PasswordsModel(UserManager<IdentityUser> usrMgr,
                IdentityEmailService emailService) {
            UserManager = usrMgr;
            EmailService = emailService;
        }

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

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

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

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

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

        public async Task<IActionResult> OnPostSetPasswordAsync() {
            if (ModelState.IsValid) {
                IdentityUser = await UserManager.FindByIdAsync(Id);
                if (await UserManager.HasPasswordAsync(IdentityUser)) {
                    await UserManager.RemovePasswordAsync(IdentityUser);
                }
                IdentityResult result =
                    await UserManager.AddPasswordAsync(IdentityUser, Password);
                if (result.Process(ModelState)) {
                    TempData["message"] = "Password Changed";
                    return RedirectToPage();
                }
            }
            return Page();
        }

        public async Task<IActionResult> OnPostUserChangeAsync() {
            IdentityUser = await UserManager.FindByIdAsync(Id);
            await UserManager.RemovePasswordAsync(IdentityUser);
            await EmailService.SendPasswordRecoveryEmail(IdentityUser,
                "/Identity/UserPasswordRecoveryConfirm");
            TempData["message"] = "Email Sent to User";
            return RedirectToPage();
        }
    }
}
```

There is no user manager method to change a password in a single step that doesn't require the existing password or a confirmation token. Instead, the `HasPasswordAsync` method is used to determine if there is a password in the user store and, if there is, the `RemovePasswordAsync` method is used to remove it. The new password is stored with the `AddPasswordAsync` method.

The recovery email is sent to the user using the same approach as for self-service password recovery. Before sending the email, I call the RemovePasswordAsync method to remove the existing password to prevent the user from signing in until a new password is chosen. For quick reference, Table 8-11 describes the user manager methods used for administrator password changes.

Table 8-11. *The UserManager<IdentityUser> Methods for Administrator Password Changes*

Name	Description
`HasPasswordAsync(user)`	This method returns `true` if there is a stored password for the user.
`RemovePasswordAsync(user)`	This method removes the stored password for the specified user.
`AddPasswordAsync(user, password)`	This method stores a password for the specified user.
`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.
`IsEmailConfirmedAsync(user)`	This method returns `true` if the user has a confirmed email address. This method is used to determine if the user can receive a password reset email.

To integrate the password functionality into the administrator dashboard, add the element shown in Listing 8-26 to the _AdminWorkflows partial view.

Listing 8-26. 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("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 Passwords button in the navigation panel to display a list of users. Click the Passwords button for any of the accounts, and you will be presented with the features that allow passwords to be changed, as shown in Figure 8-9.

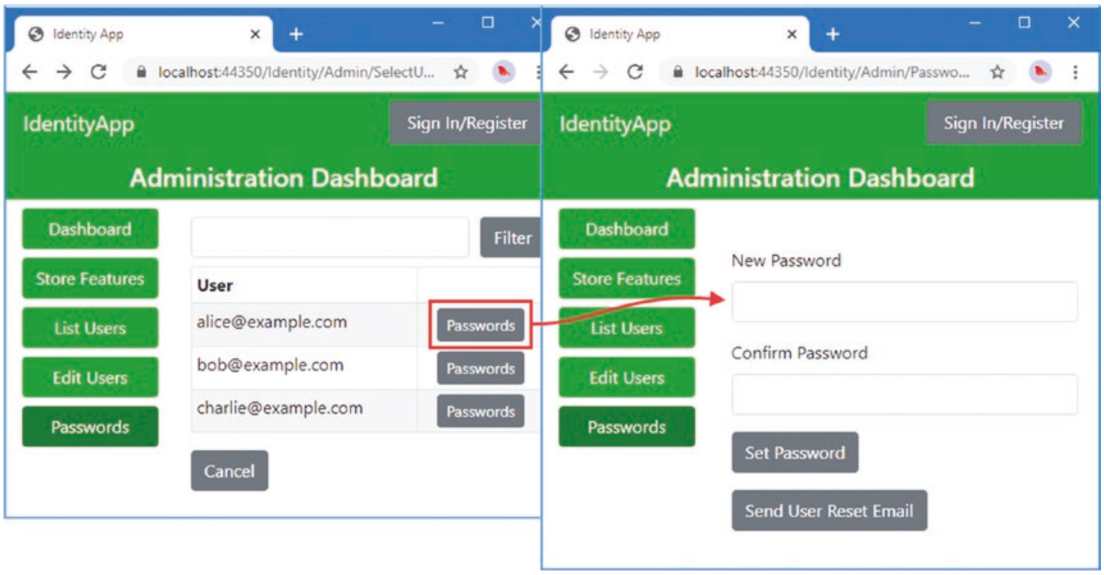


Figure 8-9. *The administrator password change feature*

Restricting Access to the Custom Workflow Razor Pages

There is enough functionality in place to restrict access to the Razor Pages that provide the custom workflows. Add the attribute shown in Listing 8-27 to the `UserPageModel` class, which is the base for the page model classes.

Listing 8-27. Restricting Access in the UserPageModel.cs File in the Pages/Identity Folder

```
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Authorization;

namespace IdentityApp.Pages.Identity {
    [Authorize]
    public class UserPageModel : PageModel {

        // no methods or properties required
    }
}
```

The Authorize attribute restricts access to any authenticated user, which is ideal for the features provided by the self-management dashboard. An exception has to be made for the SignIn page, as shown in Listing 8-28.

Listing 8-28. Granting Access 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 {

        // ...statements omitted for brevity...
    }
}
```

The AllowAnonymous attribute allows anyone to access a page. At this point in the project, more pages require the AllowAnonymous attribute than not, and you may be tempted to simply use the standard page model base class instead of applying attributes individually. My preference is to make authorization the default requirement and then explicitly grant the exceptions so that the intention for each page is obvious and consistent.

The AllowAnonymous attribute must also be applied to the SignOut page model class, as shown in Listing 8-29, since it displays content to users after they have signed out.

Listing 8-29. Granting Access in the SignOut.cshtml.cs File in the Pages/Identity Folder

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

namespace IdentityApp.Pages.Identity {

    [AllowAnonymous]
    public class SignOutModel : UserPageModel {

        public SignOutModel(SignInManager<IdentityUser> signMgr)
            => SignInManager = signMgr;
        public SignInManager<IdentityUser> SignInManager { get; set; }

        public async Task<IActionResult> OnPostAsync() {
            await SignInManager.SignOutAsync();
            return RedirectToPage();
        }
    }
}
```

Finally, the same attribute must be added to the pages that deal with password recovery. Listing 8-30 applies `AllowAnonymous` to the page model for the `UserPasswordRecovery` page.

Listing 8-30. Granting Access in the UserPasswordRecovery.cshtml.cs File in the Pages/Identity Folder

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

namespace IdentityApp.Pages.Identity {

    [AllowAnonymous]
    public class UserPasswordRecoveryModel : UserPageModel {

        // ...statements omitted for brevity...
    }
}
```

Listing 8-31 applies the attribute to the page model class for the `UserPasswordRecoveryConfirm` page.

Listing 8-31. Granting Access in the UserPasswordRecoveryConfirm.cshtml.cs File in the Pages/Identity Folder

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

namespace IdentityApp.Pages.Identity {

    [AllowAnonymous]
    public class UserPasswordRecoveryConfirmModel : UserPageModel {

        public UserPasswordRecoveryConfirmModel(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)) {
                    TempData["message"] = "Password changed";
                    return RedirectToPage();
                }
            }
            return Page();
        }
    }
}
```

The default policy of restricting access to signed-in users also applies to the administration dashboard, which will be a problem the next time the database is reset: the button that seeds the user store will be accessible only to signed-in users, but no sign ins are possible because the user store will be empty.

I'll resolve this properly in Chapter 10 when I create workflows for managing roles. Until then I am going to allow anyone to access the administration features, even if there is no signed-in user. Add the attribute shown in Listing 8-32 to the `AdminUserPage` class.

Listing 8-32. Disabling Authorization in the AdminPageModel.cs File in the Pages/Identity/Admin Folder

```
using Microsoft.AspNetCore.Authorization;

namespace IdentityApp.Pages.Identity.Admin {

    [AllowAnonymous]
    public class AdminPageModel : UserPageModel {

        // no methods or properties required
    }
}
```

Restart ASP.NET Core, make sure you are signed out of the application, and then request `https://localhost:44350/Identity`. Now that the user dashboard is restricted to signed-in users, you will be prompted for credentials and then redirected once you have signed in, as shown in Figure 8-10.

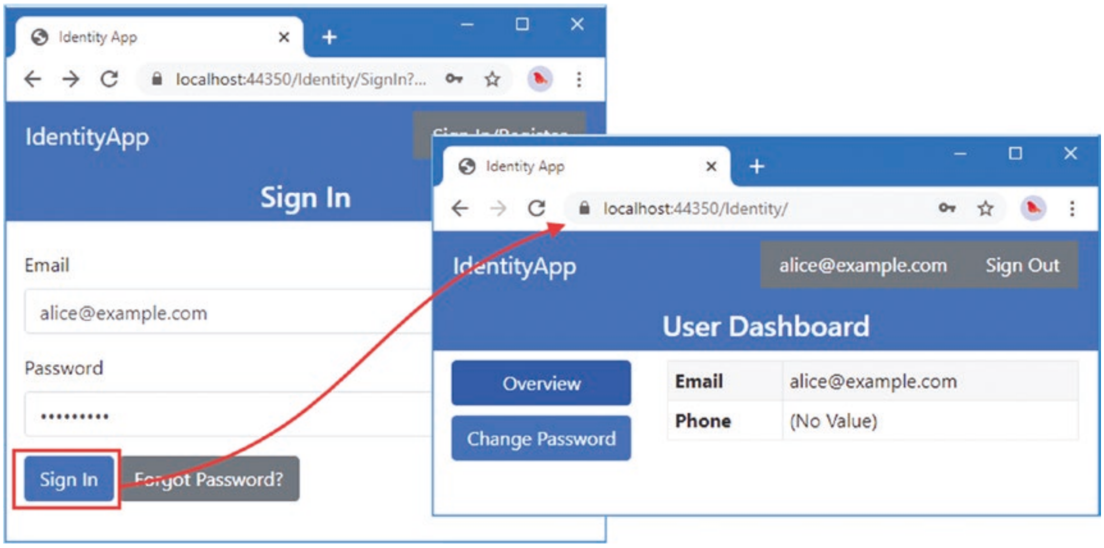


Figure 8-10. *Restricting access to the user dashboard*

Summary

In this chapter, I introduced the Identity features for signing users into an application with a password and signing them out again. I also created the workflows required to manage passwords, both for self-service applications and those that require an administrator. I finished the chapter by using the ASP.NET Core authorization features to restrict access to the user and dashboard so that it is available only to signed-in users. In the next chapter, I create workflows for creating and deleting user accounts.