CHAPTER 11

Two-Factor and External Authentication

Identity supports two-factor authentication, where the user provides additional information alongside their password. In Chapter 17, I demonstrate how to create an SMS two-factor workflow, but for this chapter, I am going to focus on support for an authenticator, which is a more secure approach but requires users to have access to an app. I also create workflows for signing in with third-party services from Facebook, Google, and Twitter, using the same configuration settings created in Chapter 5. Table 11-1 puts the features described in this chapter in context.

Table 11-1. *Putting Two-Factor Authentication and External Services in Context*

Question	Answer
What are they?	Two-factor authentication requires the user to provide an additional credential to sign in to the application. External authentication services allow a user to authenticate themselves with a third party, such as Google or Facebook.
Why are they useful?	Two-factor authentication increases the security of a user's account. External authentication allows a user to sign in with credentials they have already established, which means they don't have to manage another account and allows ASP.NET Core applications to benefit from more advanced security options that are not directly supported by ASP.NET Core Identity.
How are they used?	An authenticator app is configured to generate codes every 30 seconds. The user provides the current code, in addition to their password, when they sign in. For external authentication, the ASP.NET Core application redirects the user to the third-party service, where they are authenticated, before being redirected back to the ASP.NET Core application, where they are signed in.
Are there any pitfalls or limitations?	The default mechanism for two-factor authentication is an authenticator application, which requires the user to go through an initial configuration process and have access to a device—typically a smartphone—when they sign in. Users won't lose or forget devices, and some users don't have them. External authentication is effective but complex and can be frustrating to configure.
Are there any alternatives?	These are optional features and are not required, although they do increase the security of a user account.

Table 11-2 summarizes the chapter.

Table 11-2. *Chapter Summary*

Problem	Solution	Listing
Determine if a user has two-factor authentication enabled	Call the user manager's `GetTwoFactorEnabledAsync` method or read the `IdentityUser.TwoFactorEnabled` property	3–4, 9, 10
Generate a token that can be used to set up an authenticator	Call the user manager's `GetAuthenticatorKeyAsync` or `ResetAuthenticatorKeyAsync` methods.	5, 6
Validate a token provided by the user during setup	Call the user manager's `VerifyTwoFactorTokenAsync` method.	5, 6
Use two-factor authentication to sign into the application	If the sign-in manager's `PasswordSignInAsync` returns a `SignInResult` whose `RequiresTwoFactor` property is `true`, then validate a token and sign the user into the application using the `TwoFactorAuthenticatorSignInAsync` method.	11, 12
Generate a set of recovery codes	Call the user manager's `GenerateNewTwoFactorRecoveryCodesAsync` method.	6–8, 13–14
Use a recovery code to sign a user into the application	Call the sign-in manager's `TwoFactorRecoveryCodeSignInAsync` method.	12
Determine the external authentication services that have been configured	Call the sign-in manager's `GetExternalAuthenticationSchemesAsync` method.	16, 17
Sign in or register with an external provider	Call the `ConfigureExternalAuthenticationProperties` method to select the provider and return a challenge response. Receive the callback and get the external authentication details using the `GetExternalLoginInfoAsync` method. Sign the user into the application with the `ExternalLoginSignInAsync` method.	18–24

Preparing for This Chapter

This chapter uses the `IdentityApp` project from Chapter 10. Open a new PowerShell command prompt and run the commands shown in Listing 11-1 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 11-1. Resetting the Databases

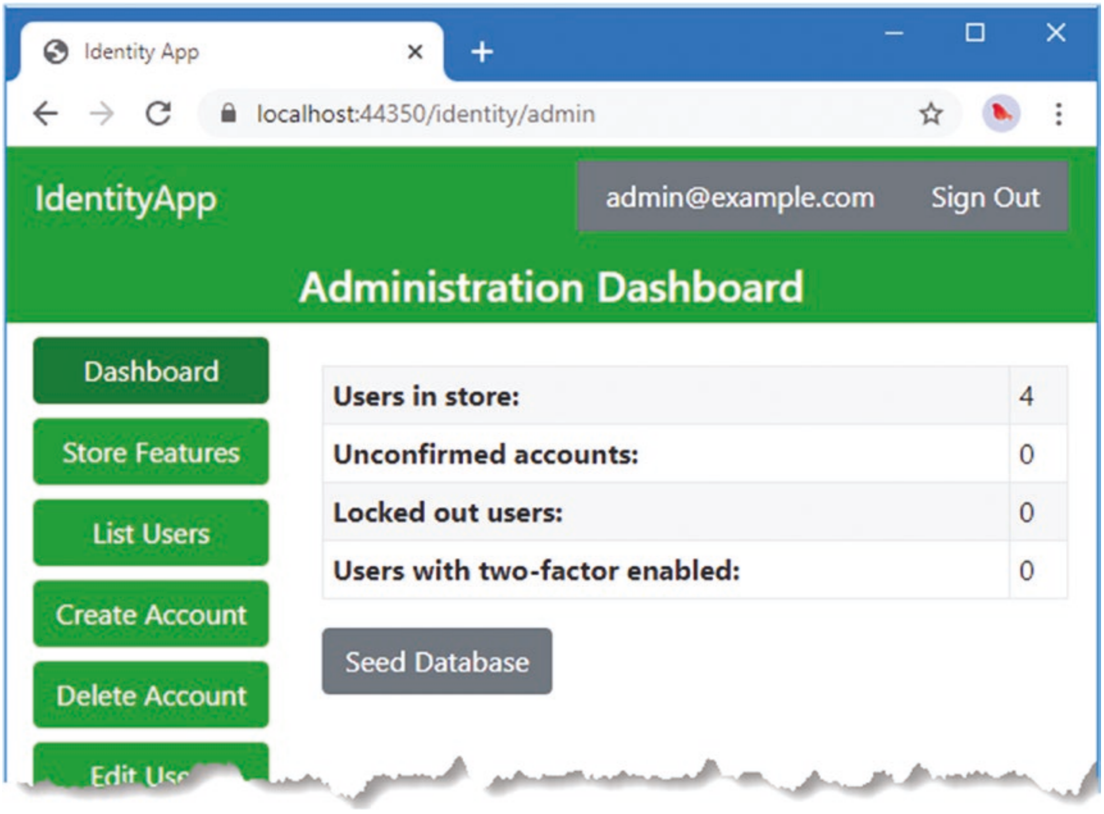
```
dotnet ef database drop --force --context ProductDbContext
dotnet ef database drop --force --context IdentityDbContext
dotnet ef database update --context ProductDbContext
dotnet ef database update --context IdentityDbContext
```

Use the PowerShell prompt to run the command shown in Listing 11-2 in the `IdentityApp` folder to start the application.

Listing 11-2. Running the Example Application

```
dotnet run
```

Open a web browser, request `https://localhost:44350/Identity/Admin`, and sign in as `admin@example.com` using `mysecret` at the password. When you sign in, you will be redirected to the administration dashboard. Click Seed Database, which will update the dashboard to indicate there are four users in the store, as shown in Figure 11-1.



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

Figure 11-1. *Running the example application*

Supporting Two-Factor Authentication

Two-factor authentication requires the user to prove their identity with two pieces of information, typically a password and a token that is given to them securely. In Part 2, I explain how two-factor authentication works in detail and demonstrate how to use SMS messages to send the user a token. In this chapter, I am going to add support for an authenticator app, which is the same approach taken by the Identity UI package. Setting up an authenticator is a self-service operation because it requires the user to configure an app, typically on a smartphone.

Create the Two-Factor Overview Page

The first step is to create a Razor Page that will display the current account status and allow the user to enable or disable an authenticator. Add a Razor Page named `UserTwoFactorManage.cshtml` to the `Pages/Identity` folder with the content shown in Listing 11-3.

Listing 11-3. The Contents of the UserTwoFactorManage.cshtml File in the Pages/Identity Folder

```
@page
@model IdentityApp.Pages.Identity.UserTwoFactorManageModel
@{
    ViewBag.Workflow = "TwoFactor";
}

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

@if (await Model.IsTwoFactorEnabled()) {
    <div class="text-center">
        <div class="h6 m-2">Your account is configured to use an authenticator</div>
        <div>
            <form method="post">
                <button type="submit" class="btn btn-primary m-1"
                        asp-page-handler="GenerateCodes">
                    Generate New Recovery Codes
                </button>
                <button type="submit" class="btn btn-warning m-1"
                        asp-page-handler="Disable">
                    Disable Authenticator and Sign Out
                </button>
            </form>
        </div>
    </div>

} else {
    <h6>Your account is not configured to use an authenticator.</h6>
    <a asp-page="UserTwoFactorSetup" class="btn btn-primary">Enable Authenticator</a>
}
```

The content produced by the view part of the page presents a button that allows the user to set up an authenticator by navigating to a page named `UserTwoFactorSetup`, which I will create shortly. If there is an authenticator, then the user is presented with a form that allows the authenticator to be disabled or a new set of recovery codes to be generated. Recovery codes are one-time passwords that can be used when the authenticator isn't available. Applications don't have to support recovery codes, but users forget or lose phones and won't always have the authenticator to hand.

To create the page model class, add the code shown in Listing 11-4 to the `UserTwoFactorManage.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 11-4. The Contents of the UserTwoFactorManage.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 UserTwoFactorManageModel : UserPageModel {

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

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

        public IdentityUser IdentityUser { get; set; }

        public async Task<bool> IsTwoFactorEnabled()
            => await UserManager.GetTwoFactorEnabledAsync(IdentityUser);

        public async Task OnGetAsync() {
            IdentityUser = await UserManager.GetUserAsync(User);
        }

        public async Task<IActionResult> OnPostDisable() {
            IdentityUser = await UserManager.GetUserAsync(User);
            IdentityResult result = await
                UserManager.SetTwoFactorEnabledAsync(IdentityUser, false);
            if (result.Process(ModelState)) {
                await SignInManager.SignOutAsync();
                return RedirectToPage("Index", new { });
            }
            return Page();
        }

        public async Task<IActionResult> OnPostGenerateCodes() {
            IdentityUser = await UserManager.GetUserAsync(User);
            TempData["RecoveryCodes"] =
                await UserManager.GenerateNewTwoFactorRecoveryCodesAsync(
                    IdentityUser, 10);
            return RedirectToPage("UserRecoveryCodes");
        }
    }
}
```

The user manager's GetTwoFactorEnabledAsync method is used to determine if the user has configured an authenticator. Applications can support multiple forms of two-factor authentication, as I demonstrate in Part 2, but this example uses only authenticators, so I can assume that an account with two-factor authentication enabled has been configured with an authenticator.

The SetTwoFactorEnabledAsync method is used to enable and disable two-factor authentication. In this class, I only need to disable two-factor authentication in the Disable POST handler method.

The GenerateCodes POST handler method generates a new set of recovery codes. These codes are shown to the user only once and are consumed when they are used, so it is important to ensure that the user can create a new set, either when they use up the codes or forget them. (I am not a fan of security codes that cannot be viewed, and I show you how to change this behavior in Part 2 so the user can see their unused codes.)

Creating the Authenticator Setup Page

As part of the setup process, the user must enter a secret key generated by Identity into their authenticator. To present the user with the key, add a Razor Page named UserTwoFactorSetup.cshtml to the Pages/Identity folder with the content shown in Listing 11-5.

Listing 11-5. The Contents of the UserTwoFactorSetup.cshtml File in the Pages/Identity Folder

```
@page
@model IdentityApp.Pages.Identity.UserTwoFactorSetupModel
@{
    ViewBag.Workflow = "TwoFactor";
}

<div class="container-fluid">
    <div class="row">
        <div class="col">
            <h6>Step 1:</h6>
            Scan the QR Code or enter the following key into your authenticator:
            <div><kbd>@Model.AuthenticatorKey</kbd> </div>
        </div>
        <div class="col-auto p-2">
            <div id="qrCode"></div>
            <div id="qrCodeData" data-url="@Html.Raw(@Model.QrCodeUrl)"></div>
        </div>
    </div>
    <div>
        <div class="row">
            <div class="col">
                <div asp-validation-summary="All" class="text-danger m-2"></div>
                <form method="post" asp-page-handler="confirm">
                    <h6>Step 2:</h6>
                    Enter the code shown by your authenticator into the
                    text field and click the Confirm button
                    <input name="confirm" placeholder="Enter code"
                        class="form-control my-2" />
                    <button class="btn btn-primary" type="submit">Confirm</button>
                </form>
            </div>
        </div>
    </div>
</div>

<script type="text/javascript" src="/lib/qrcode/qrcode.min.js"></script>
    <script type="text/javascript">
        new QRCode(document.getElementById("qrCode"), {
            text: document.getElementById("qrCodeData").getAttribute("data-url"),
            width: 150, height: 150
        });
</script>
```

The view displays the key to the user. Most authenticators are apps on smartphones, and displaying a QR code provides the user with an easier configuration path. The view displays a QR code using the same JavaScript package added to the example project to support the Identity UI package. See Chapter 4 for the Library Manager (`libman`) command required to install the JavaScript package for generating QR codes.

The view prompts the user to enter a code generated by the authenticator app. A new code will be displayed every 30 seconds, and asking the user for the current code is a sensible way of ensuring that the authenticator is working before updating the account to require two-factor authentication. To define the page model class, add the code shown in Listing 11-6 to the `UserTwoFactorSetup.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 11-6. The Contents of the UserTwoFactorSetup.cshtml.cs File in the Pages/Identity Folder

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

namespace IdentityApp.Pages.Identity {

    public class UserTwoFactorSetupModel : UserPageModel {

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

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

        public IdentityUser IdentityUser { get; set; }

        public string AuthenticatorKey { get; set; }

        public string QrCodeUrl { get; set; }

        public async Task<IActionResult> OnGet() {
            await LoadAuthenticatorKeys();
            if (await UserManager.GetTwoFactorEnabledAsync(IdentityUser)) {
                return RedirectToPage("UserTwoFactorManage");
            }
            return Page();
        }

        public async Task<IActionResult> OnPostConfirm([Required] string confirm) {
            await LoadAuthenticatorKeys();
            if (ModelState.IsValid) {
                string token = Regex.Replace(confirm, @"\s", "");
                bool codeValid = await
                        UserManager.VerifyTwoFactorTokenAsync(IdentityUser,
                    UserManager.Options.Tokens.AuthenticatorTokenProvider, token);
                if (codeValid) {
                    TempData["RecoveryCodes"] = await UserManager
                        .GenerateNewTwoFactorRecoveryCodesAsync(IdentityUser, 10);
                    await UserManager.SetTwoFactorEnabledAsync(IdentityUser, true);
                    await SignInManager.RefreshSignInAsync(IdentityUser);
                    return RedirectToPage("UserRecoveryCodes");
                } else {
                    ModelState.AddModelError(string.Empty,
                        "Confirmation code invalid");
                }
            }
            return Page();
        }

        private async Task LoadAuthenticatorKeys() {
            IdentityUser = await UserManager.GetUserAsync(User);
            AuthenticatorKey =
                await UserManager.GetAuthenticatorKeyAsync(IdentityUser);
            if (AuthenticatorKey == null) {
                await UserManager.ResetAuthenticatorKeyAsync(IdentityUser);
                AuthenticatorKey =
                    await UserManager.GetAuthenticatorKeyAsync(IdentityUser);
                await SignInManager.RefreshSignInAsync(IdentityUser);
            }
            QrCodeUrl = $"otpauth://totp/ExampleApp:{IdentityUser.Email}"
                        + $"?secret={AuthenticatorKey}";
        }
    }
}
```

When the user requests the page, the GET handler uses the `GetTwoFactorEnabledAsync` method to see if the user is already configured for two-factor authentication. If the response is `true`, then a redirection to the `UserTwoFactorManage` page is performed.

For users without two-factor authentication enabled, the GET handler method is responsible for presenting the secret key to the user. Authenticator keys are persistent and are used to validate the codes generated by the authenticator every 60 seconds. If the stored key doesn't match the one used by the authenticator, the user won't be able to sign in.

The user manager's GetAuthenticatorKeyAsync method retrieves the secret key from the user store. If the method returns null, then no key has been stored. A new key is created and stored using the ResetAuthenticatorKeyAsync method, and the GetAuthenticatorKeyAsync method is called again to retrieve the key from the store. The key is presented to the user directly and formatted in a URL that can be displayed as a QR code. (The format of these URLs is described in Chapter 21.)

The POST handler receives the code displayed by the user's authenticator app, which is processed to remove whitespace. Some authenticators, such as the Authy app I use in this book, display tokens in groups of digits separated by a space. These must be removed before the token can be validated, which I do with a regular expression.

```
...
string token = Regex.Replace(confirm, @"\s", "");
...
```

The token is validated using the VerifyTwoFactorTokenAsync method, like this:

```
...
UserManager.VerifyTwoFactorTokenAsync(IdentityUser,
   UserManager.Options.Tokens.AuthenticatorTokenProvider, token);
...
```

The arguments are the user object, the name of the token provider class, and the code to validate. In Listing 11-6, I read the user manager's Options property to get the value of the AuthenticatorTokenProvider property, which specifies the name of the provider. The provider is configured by the AddDefaultTokenProviders extension method added to the Identity configuration in Chapter 8. This method sets up token generators that are suitable for most applications, but I describe how tokens are generated in detail in Part 2 if your application has specific requirements.

If the code provided by the user is valid, I generate a new set of recovery codes using the GenerateNewTwoFactorRecoveryCodesAsync method, like this:

```
...
TempData["RecoveryCodes"] =
    await UserManager.GenerateNewTwoFactorRecoveryCodesAsync(IdentityUser, 10);
...
```

I specified that I require 10 codes, which I store as temp data before redirecting the browser to the UserRecoveryCodes page so that I can display them to the user.

In Chapter 9, I enabled authentication cookie validation to effectively sign the user out of the application when their security stamp changes. As I explained in that chapter, this can cause the user to be signed out when other changes are performed, and this includes setting up an authenticator. The ResetAuthenticatorKeyAsync and SetTwoFactorEnabledAsync methods both update the security stamp. To prevent signing the user out of the application, I use the SignInManager<IdentityUser>.RefreshSignInAsync method to refresh the authentication cookie after these methods are called:

```
...
await UserManager.SetTwoFactorEnabledAsync(IdentityUser, true);
await SignInManager.RefreshSignInAsync(IdentityUser);
...
```

The effect of the ResetAuthenticatorKeyAsync method can be especially problematic because this method is called during the authenticator setup state, which means the user can be signed out of the application as they are configuring their authenticator, which can be confusing.

To display the recovery codes, add a Razor Page named UserRecoveryCodes.cshtml to the Pages/Identity folder with the content shown in Listing 11-7.

Listing 11-7. The Contents of the UserRecoveryCodes.cshtml File in the Pages/Identity Folder

```
@page
@model IdentityApp.Pages.Identity.UserRecoveryCodesModel
@{
    ViewBag.Workflow = "TwoFactor";
}

<h4 class="text-center">Recovery Codes</h4>

<h6>
    These recovery codes can be used to sign in if you don't have your authenticator.
    Store these codes in a safe place. You won't be able to view them again.
    Each code can only be used once.
</h6>

<table class="table table-sm table-striped">
    <tbody>
        @for (int i = 0; i < Model.RecoveryCodes.Length; i +=2 ) {
            <tr>
                <td><code>@Model.RecoveryCodes[i]</code></td>
                <td><code>@Model.RecoveryCodes[i + 1]</code></td>
            </tr>
        }
    </tbody>
</table>
<a asp-page="UserTwoFactorManage" class="btn btn-primary">OK</a>
```

The recovery codes are displayed in a table, along with a message explaining to the user that the codes cannot be viewed again and must be stored safely. To define the page model class, add the code shown in Listing 11-8 to the UserRecoveryCodes.cshtml.cs file. (You will have to create this file if you are using Visual Studio Code.)

Listing 11-8. The Contents of the UserRecoveryCodes.cshtml.cs File in the Pages/Identity Folder

```
using Microsoft.AspNetCore.Mvc;

namespace IdentityApp.Pages.Identity {

    public class UserRecoveryCodesModel : UserPageModel {

        [TempData]
        public string[] RecoveryCodes { get; set; }

        public IActionResult OnGet() {
            if (RecoveryCodes == null || RecoveryCodes.Length == 0) {
                return RedirectToPage("UserTwoFactorManage");
            }
            return Page();
        }
    }
}
```

The TempData attribute is used to set the value of the RecoveryCodes property, and the GET handler will perform a redirection to the management page if there are no recovery codes.

Updating the User and Administrator Dashboards

Add the element shown in Listing 11-9 to provide the user with navigation to the two-factor feature.

Listing 11-9. 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("UserTwoFactor")"
            asp-page="UserTwoFactorManage">
        Authenticator
    </a>
}
<a class="btn btn-@Model.theme btn-block @getClass("UserDelete")"
        asp-page="UserDelete">
    Delete Account
</a>
```

The navigation element will be shown only to users who have password, which will prevent users who sign in with external services from using an authenticator.

The administrator dashboard overview displays the number of users with two-factor authentication. To set the value, add the statement shown in Listing 11-10 to the Dashboard.cshtml.cs file in the Pages/Identity/Admin folder.

Listing 11-10. Counting Users in the Dashboard.cshtml.cs File in the Pages/Identity/Admin Folder

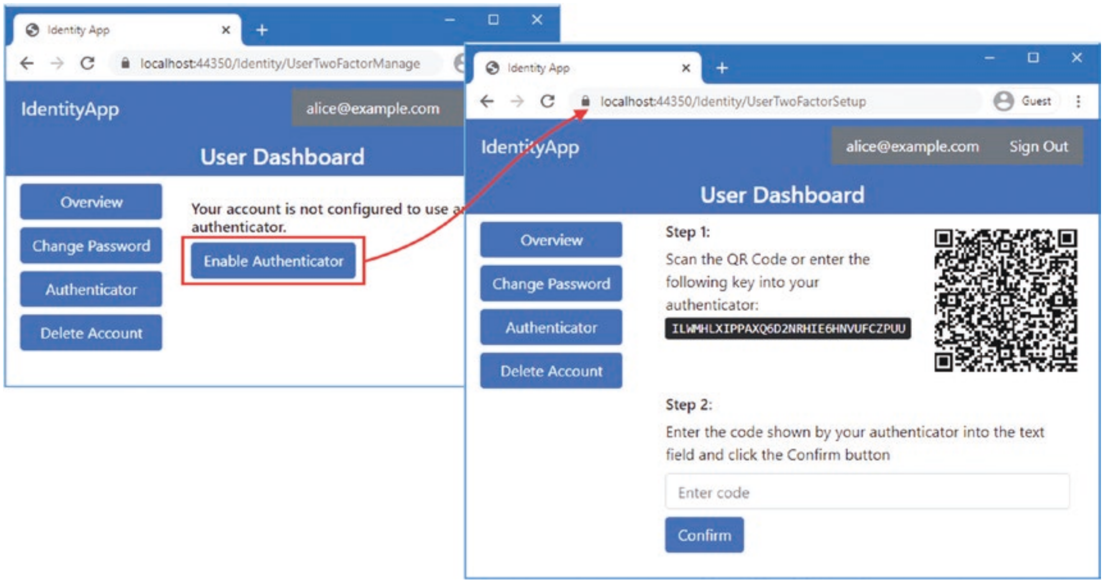
```
...
public void OnGet() {
    UsersCount = UserManager.Users.Count();
    UsersUnconfirmed = UserManager.Users
        .Where(u => !u.EmailConfirmed).Count();
    UsersLockedout = UserManager.Users
        .Where(u => u.LockoutEnabled && u.LockoutEnd > System.DateTimeOffset.Now)
        .Count();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

The user manager's `GetTwoFactorEnabledAsync` method can't be evaluated by the database server, which means that counting users with this method would require retrieving all of the stored `IdentityUser` objects, calling the method on each of them, and counting the results. Instead, I have chosen to use the `TwoFactorEnabled` property defined by the `IdentityUser` class.

Restart ASP.NET Core, sign into the application as `alice@example.com` with password `mysecret`, and request `https://localhost:44350/Identity`. Click the Authenticator button, and you will see the message shown in the first screenshot in Figure 11-2.

Click the Enable Authenticator button, and you will be shown a secret key used to set up the authenticator and the QR code, as shown in the second screenshot in Figure 11-2.

■ **Note** If you do not see the QR code, then you may not have installed the required JavaScript package. See Chapter 4 for instructions.



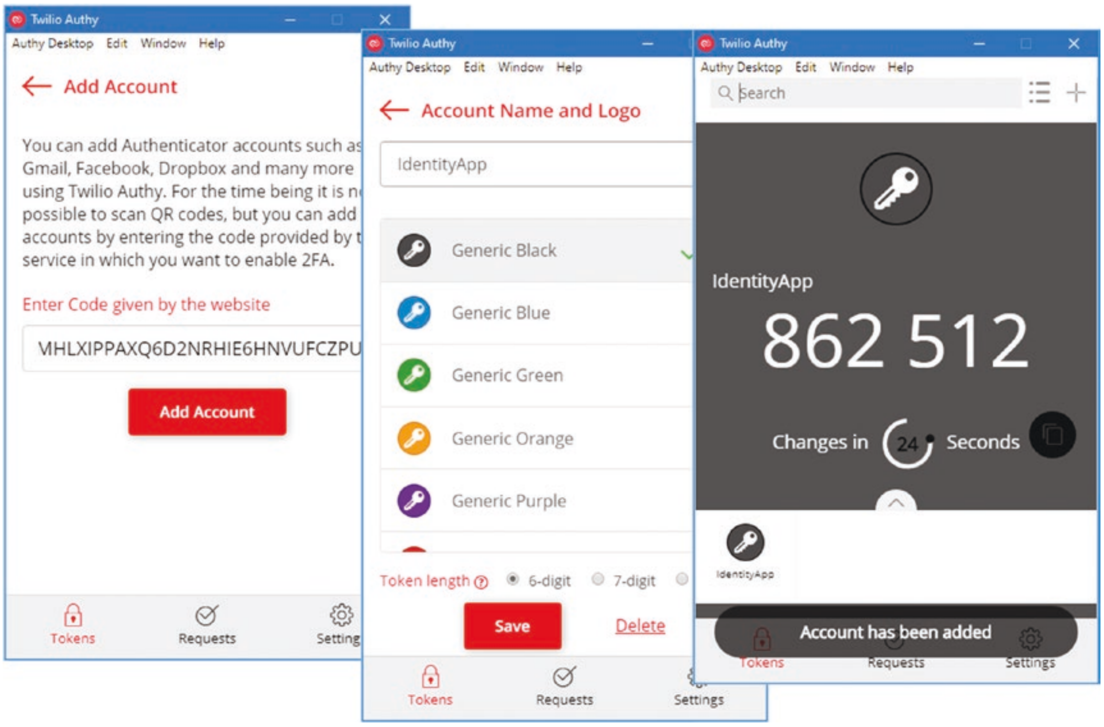
Your account is not configured to use a
authenticator.

Step 1:
Scan the QR Code or enter the following key into your authenticator:
ILWMHLXIPPAXQ6D2NRHIE6HNVUFCZPUU

Step 2:
Enter the code shown by your authenticator into the text field and click the Confirm button

Figure 11-2. *Setting up two-factor authentication*

Enter the key into your authenticator or scan the QR code. I use the Authy app (`authy.com`) for the examples in this book because there is a Windows client, but there are alternatives from Google and Microsoft that run on mobile devices available in the iOS and Android app stores. (There is also a good tool at `https://totp.danhersam.com` that I use during development, into which you can paste a key and start receiving codes without any additional configuration, which is helpful if you are repeatedly testing a workflow.) The authenticator will start generating tokens once it has been set up, as shown in Figure 11-3.



You can add Authenticator accounts such as Gmail, Facebook, Dropbox and many more using Twilio Authy. For the time being it is n possible to scan QR codes, but you can add accounts by entering the code provided by t service in which you want to enable 2FA.

MHLXIPPAXQ6D2NRHIE6HNVUFCZPU

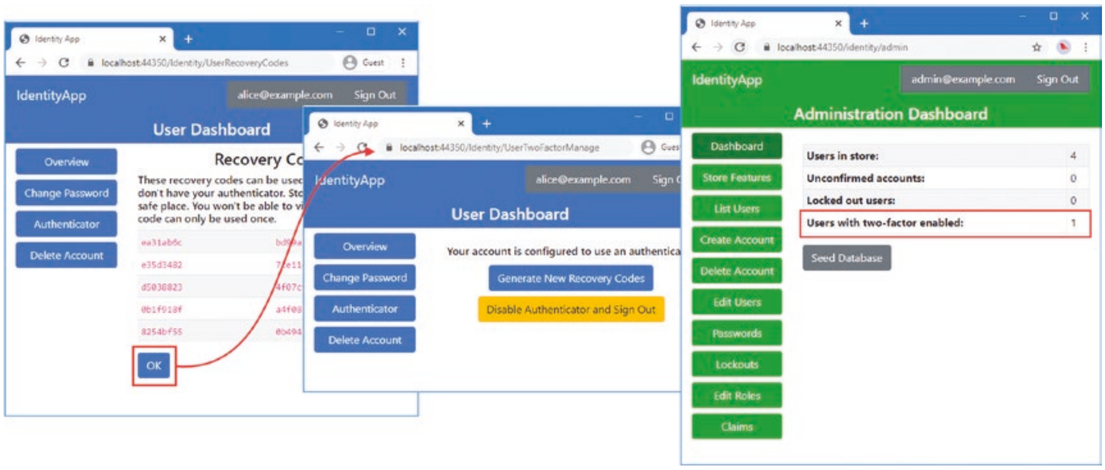
IdentityApp

862 512

Account has been added

Figure 11-3. Setting up the authenticator

Enter the current token into the text field displayed in the browser, and click the Confirm button. You will be presented with a set of recovery codes, as shown in Figure 11-4. Click the OK button to return to the `UserTwoFactorManage` page, where you will be presented with buttons that generate a new set of recovery codes and disable the authenticator.



These recovery codes can be used
don't have your authenticator. Sto
safe place. You won't be able to vi
code can only be used once.

Your account is configured to use an authentica

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

Figure 11-4. Completing the authenticator setup

As Figure 11-4 also shows, the administrator dashboard shows that there is now one user set up for two-factor authentication. For quick reference, Table 11-3 describes the user manager methods used to manage two-factor authentication with an authenticator.

Table 11-3. The UserManager<IdentityUser> Methods for Two-Factor Authenticator Authentication

Name	Description
`GetTwoFactorEnabledAsync(user)`	This method returns `true` if the user account has been configured for two-factor authentication.
`SetTwoFactorEnabledAsync(user, enabled)`	This method is used to enable or disable two-factor authentication. Care must be taken to configure the authenticator before calling this method.
`GetAuthenticatorKeyAsync(user)`	This method returns the user's authenticator secret key from the user store or `null` if no key has been stored.
`ResetAuthenticatorKeyAsync(user)`	This method is used to generate a new authenticator secret key and add it to the store.
`VerifyTwoFactorTokenAsync(user, code)`	This method verifies an authenticator code using the secret key in the user store. Verifying the code does not sign the user into the application.
`GenerateNewTwoFactorRecoveryCodesAsync(user, count)`	This method generates a new set of recovery codes.

Table 11-4 describes the `SignInManager<IdentityUser>` method I use to prevent the user from being signed out of the application while setting up an authenticator.

Table 11-4. *The SignInManager<IdentityUser> Method for Preventing Sign Out*

Name	Description
`RefreshSignInAsync(user)`	This method signs the user into the application using the existing authentication settings, refreshing the authentication cookie using the current security stamp.

Signing In with an Authenticator

When I created the `SignIn` page, I examined the properties of the `SignInResult` object to determine the outcome of the sign in, like this:

```
...
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 has set up an authenticator, the result of the `PasswordSignInAsync` method is a `SignInResult` object whose `RequiresTwoFactor` property is `true` (assuming that they have provided the correct password, of course). For this outcome, I send a redirection to a page named `SignInTwoFactor` so the user can complete the sign-in process. Add a Razor Page named `SignInTwoFactor.cshtml` to the `Pages/Identity` folder with the content shown in Listing 11-11.

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

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

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

<form method="post">
    <div class="form-group">
        <label>Authenticator Token or Recovery Code:</label>
        <input class="form-control" asp-for="Token" />
    </div>
    <div class="form-check">
        <input class="form-check-input" type="checkbox" asp-for="RememberMe" />
        <label class="form-check-label" >Remember Me</label>
    </div>
    <button type="submit" class="btn btn-primary mt-2">Sign In</button>
</form>
```

The view part of the page presents the user with an `input` element into which an authenticator token or a recovery code can be entered. There is also a checkbox that will be used to set a cookie so the user can sign in using the same browser with just a password.

Add the code shown in Listing 11-12 to the `SignInTwoFactor.cshtml.cs` file to define the page model class. (You will have to create this file if you are using Visual Studio Code.)

Listing 11-12. The Contents of the SignInTwoFactor.cshtml.cs File in the Pages/Identity Folder

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

namespace IdentityApp.Pages.Identity {

    [AllowAnonymous]
    public class SignInTwoFactorModel : UserPageModel {

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

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

        [BindProperty]
        public string ReturnUrl { get; set; }

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

        [BindProperty]
        public bool RememberMe { get; set; }

        public async Task<IActionResult> OnPostAsync() {
            if (ModelState.IsValid) {
                IdentityUser user = await
                    SignInManager.GetTwoFactorAuthenticationUserAsync();
                if (user != null) {
                    string token = Regex.Replace(Token, @"\s", "");
                    SignInResult result = await
                         SignInManager.TwoFactorAuthenticatorSignInAsync(token, true,
                            RememberMe);
                    if (!result.Succeeded) {
                        result = await
                            SignInManager.TwoFactorRecoveryCodeSignInAsync(token);
                    }
                    if (result.Succeeded) {
                        if (await UserManager.CountRecoveryCodesAsync(user) <= 3) {
                            return RedirectToPage("SignInCodesWarning");
                        }
                        return Redirect(ReturnUrl ?? "/");
                    }
                }
                ModelState.AddModelError("", "Invalid token or recovery code");
            }
            return Page();
        }
    }
}
```

The first step is to make sure the user has provided a valid password in the first stage of the sign-in process, like this:

```
...
IdentityUser user = await SignInManager.GetTwoFactorAuthenticationUserAsync();
...
```

The sign-in manager's GetTwoFactorAuthenticationUserAsync method retrieves the IdentityUser object associated with the email address and password provided in the previous step. If this method returns null, then the user has not provided a password, and the signing-in process should be stopped.

In Part 2, I handle authenticator tokens and recovery codes separately, but, for this chapter, I first try the string provided by the user as an authenticator token and then fall back to using it as a recovery code.

The sign-in manager's TwoFactorAuthenticatorSignInAsync method is used to sign in with an authenticator token. The arguments are the token, a bool indicating whether the authentication cookie should be persistent, and a bool indicating whether a cookie should be created that will allow the user to sign in without the authenticator from the same browser.

```
...
await SignInManager.TwoFactorAuthenticatorSignInAsync(token, true, RememberMe);
...
```

If the user has provided a valid token, they are signed into the application. If the token is not valid, then I try and use it as a recovery code, like this:

```
...
result = await SignInManager.TwoFactorRecoveryCodeSignInAsync(token);
...
```

The user is signed in if the recovery code is valid. Codes can be used only once, and it is important to warn the user if they are running out of codes. I use the user manager's `CountRecoveryCodesAsync` method to check how many are remaining and redirect the user to a warning page if there three or fewer codes left.

```
...
if (await UserManager.CountRecoveryCodesAsync(user) <= 3) {
...
```

To define the warning page, add a Razor Page named `SignInCodesWarning.cshtml` to the `Pages/Identity` folder with the content shown in Listing 11-13.

Listing 11-13. The Contents of the SignInCodesWarning.cshtml File in the Pages/Identity Folder

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

<div class="h6">
    You are running out of recovery codes.
    Generate new codes using the user dashboard.
</div>
<a href="@Model.ReturnUrl" class="btn btn-primary my-2">Continue to application</a>
<a asp-page="UserTwoFactorManage" class="btn btn-primary m-2">Go to dashboard</a>
```

To define the page model class, add the code shown in Listing 11-14 to the `SignInCodesWarning.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 11-14. The Contents of the SignInCodesWarning.cshtml.cs File in the Pages/Identity Folder

```
using Microsoft.AspNetCore.Mvc;

namespace IdentityApp.Pages.Identity {

    public class SignInCodesWarningModel : UserPageModel {

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

For quick reference, Table 11-5 shows the sign-in manager methods used for two-factor sign-in with an authenticator.

Table 11-5. *The SignInManager<IdentityUser> Methods for Authenticator Two-Factor Sign-Ins*

Name	Description
`GetTwoFactorAuthenticationUserAsync()`	This method returns the `IdentityUser` object associated with the username/email address provided at the password stage. You should stop the sign-in process if this method returns `null`.
`TwoFactorAuthenticatorSignInAsync(token, persist, remember)`	This method validates an authenticator token and signs the user into the application.
`TwoFactorRecoveryCodeSignInAsync(code)`	This method validates a recovery code and signs the user into the application. The code is removed from the store so that it cannot be reused.

Table 11-6 describes the user manager method I used to count the unused recovery codes.

Table 11-6. *The UserManager<IdentityUser> Method for Counting Recovery Codes*

Name	Description
`CountRecoveryCodesAsync(user)`	This method returns the number of unused recovery codes for the specific `IdentityUser` object.

Restart ASP.NET Core, make sure you are signed out of the application, and request `https://localhost:44350/Identity/SignIn`. Enter `alice@example.com` into the email field and use `mysecret` as the password. Click the Sign In button, enter the current token displayed by your authenticator, and click the Sign In button to complete the sign-in process, as shown in Figure 11-5.

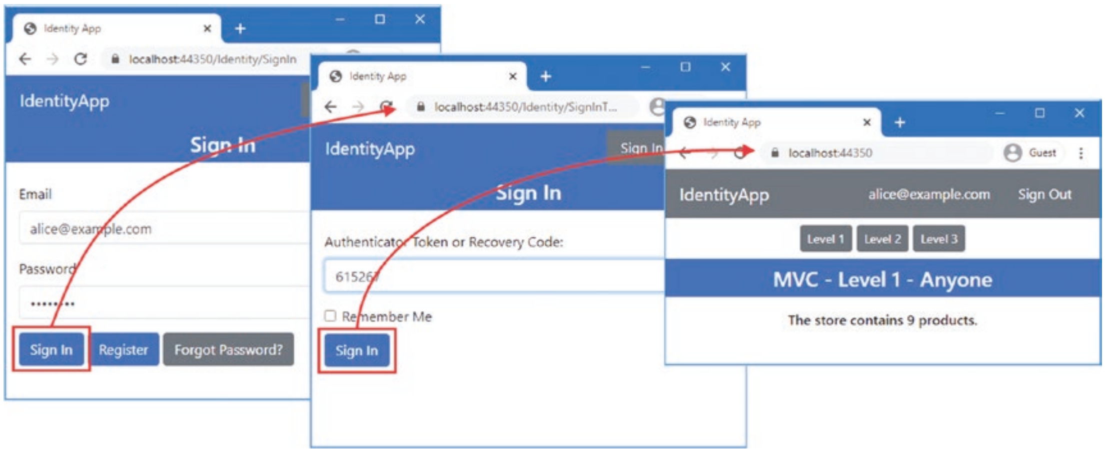
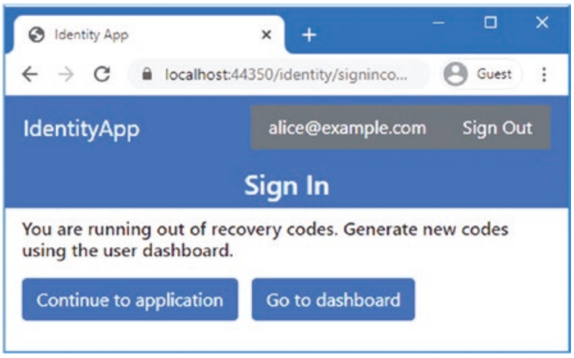


***Figure 11-5.** Signing in with an authenticator*

If you select the Remember Me option when signing in, then you won't be prompted for an authenticator token the next time you sign in as the same user with the same browser. You will need to delete the browser's cookies—or wait until the cookie expires—to be prompted for a token once again. If you are close to exhausting the set of recovery codes when you sign into the application, you will see the warning shown in Figure 11-6.



You are running out of recovery codes. Generate new codes using the user dashboard.

***Figure 11-6.** Warning the user about recovery codes*

Supporting External Authentication Services

In Chapter 5, I explained how to configure ASP.NET Core for authentication with third-party services, such as Google and Facebook. There are several ways in which you can support external authentication services. The `IdentityUI` package, for example, allows users to have a local password and external authentication and move freely between them. The idea is that the user can still sign in, even when the external service is down. My preference is to have the user choose one or the other, on the basis that the external services provided by Google or Facebook are generally reliable and that presenting users with unnecessary options will lead to additional support requests.

Configuring the Application

I configured Identity to use the built-in providers for Google, Facebook, and Twitter in Chapter 5. One change is required for this chapter, to adjust the configuration of the Twitter provider so that it requests user details from the external service, as shown in Listing 11-15. These details include the user's email address, which I will use to create accounts in the user store.

Listing 11-15. Changing the Provider Configuration in the Startup.cs File in the IdentityApp Folder

```
...
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"];
        opts.RetrieveUserDetails = true;
    });
...
```

Supporting Self-Service Registration an External Service

The first step is to display the option to create an account alongside the local authentication option that is already supported. Add the content shown in Listing 11-16 to the `SignUp.cshtml` file in the `Pages/Identity` folder.

Listing 11-16. Displaying External Services in the SignUp.cshtml File in the Pages/Identity Folder

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

<link href="/lib/font-awesome/css/all.min.css" rel="stylesheet" />

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

<div class="container-fluid">
    <div class="row">
        <div class="col-6">
            <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>
        </div>
        <div class="col-auto text-center">
            <h6>Sign Up with a Social Media Account</h6>
            <form method="post" asp-page="SignUpExternal">
                @foreach (var scheme in Model.ExternalSchemes) {
                    <partial name="_ExternalButtonPartial" model="scheme" />
                }
            </form>
        </div>
    </div>
</div>
```

The link element includes the CSS stylesheet for the Font Awesome package I added to the project in Chapter 5 to display icons for the external services. The remaining additions create a grid layout that will display buttons for each of the configured external services in the application, alongside the traditional email/password sign-up option. The button for each service is displayed using the _ExternalButtonPartial partial view created for the Identity UI package. When the user clicks one of the buttons, they will submit a form to a page named SignUpExternal, which I will create shortly.

Listing 11-17 shows the corresponding changes to the page model class to provide the view with the list of external services.

Listing 11-17. Supporting External Services in 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;
using System.Collections.Generic;
using Microsoft.AspNetCore.Authentication;

namespace IdentityApp.Pages.Identity {

    [AllowAnonymous]
    public class SignUpModel : UserPageModel {

        public SignUpModel(UserManager<IdentityUser> usrMgr,
                IdentityEmailService emailService,
                SignInManager<IdentityUser> signMgr) {
            UserManager = usrMgr;
            EmailService = emailService;
            SignInManager = signMgr;
        }
        public UserManager<IdentityUser> UserManager { get; set; }
        public IdentityEmailService EmailService { get; set; }
        public SignInManager<IdentityUser> SignInManager { get; set; }

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

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

        public IEnumerable<AuthenticationScheme> ExternalSchemes { get; set; }

        public async Task OnGetAsync() {
            ExternalSchemes = await
                SignInManager.GetExternalAuthenticationSchemesAsync();
        }

        public async Task<IActionResult> OnPostAsync() {
            // ...statements omitted for brevity...
        }
    }
}
```

The sign-in manager's `GetExternalAuthenticationSchemesAsync` method returns a sequence of `AuthenticationScheme` objects, each of which describes one of the external services configured in the application. Add a Razor Page named `SignUpExternal.cshtml` to the `Pages/Identity` folder with the content shown in Listing 11-18.

Listing 11-18. The Contents of the SignUpExternal.cshtml File in the Pages/Identity Folder

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

@if (TempData["errorMessage"] != null) {
        <div class="alert alert-danger">@TempData["errorMessage"]</div>
        <a asp-page="SignUp" class="btn btn-danger">Cancel Sign Up</a>
} else {
    <div class="text-center">
        <h6>
            An account for <code>@Model.IdentityUser.Email</code> has been created.
        </h6>
        <h6>Click the OK button and sign in using the
            @(await Model.ExternalProvider()) button.
        </h6>
        <a asp-page="SignIn" class="btn btn-primary">OK</a>
    </div>
}
```

The view part of this page only has to display a confirmation message when an account is set up or an error message if something goes wrong. The complexity of the process is in the page model class, which is responsible for dealing with the external service and creating an account. Add the code shown in Listing 11-19 to the `SignUpExternal.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 11-19. The Contents of the SignUpExternal.cshtml.cs File in the Pages/Identity Folder

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using System.Security.Claims;
using System.Threading.Tasks;

namespace IdentityApp.Pages.Identity {

    [AllowAnonymous]
    public class SignUpExternalModel : UserPageModel {

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

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

        public IdentityUser IdentityUser { get; set; }

        public async Task<string> ExternalProvider() =>
            (await UserManager.GetLoginsAsync(IdentityUser))
            .FirstOrDefault()?.ProviderDisplayName;

        public IActionResult OnPost(string provider) {
            string callbackUrl = Url.Page("SignUpExternal", "Callback");
             AuthenticationProperties props =
               SignInManager.ConfigureExternalAuthenticationProperties(
                   provider, callbackUrl);
            return new ChallengeResult(provider, props);
        }

        public async Task<IActionResult> OnGetCallbackAsync() {
            ExternalLoginInfo info = await SignInManager.GetExternalLoginInfoAsync();

            string email = info?.Principal?.FindFirst(ClaimTypes.Email)?.Value;
            if (string.IsNullOrEmpty(email)) {
                return Error("External service has not provided an email address.");
            } else if ((await UserManager.FindByEmailAsync(email)) != null ) {
                return Error("An account already exists with your email address.");
            }

            IdentityUser identUser = new IdentityUser {
                UserName = email,
                Email = email,
                EmailConfirmed = true
            };
            IdentityResult result = await UserManager.CreateAsync(identUser);
            if (result.Succeeded) {
                identUser = await UserManager.FindByEmailAsync(email);
                result = await UserManager.AddLoginAsync(identUser, info);
                return RedirectToPage(new { id = identUser.Id });
            }
            return Error("An account could not be created.");
        }

        public async Task<IActionResult> OnGetAsync(string id) {
            if (id == null) {
                return RedirectToPage("SignUp");
            } else {
                IdentityUser = await UserManager.FindByIdAsync(id);
                if (IdentityUser == null) {
                    return RedirectToPage("SignUp");
                }
            }
            return Page();
        }

        private IActionResult Error(string err) {
            TempData["errorMessage"] = err;
            return RedirectToPage();
        }
    }
}
```

I have defined the handler methods in the order they will be used to try to help simplify a complex process. The OnPost handler method will be called when the user clicks one of the external authentication buttons shown by the SignUp page.

The sign-in manager’s ConfigureExternalAuthenticationProperties method is called to create an AuthenticationProperties object that will authenticate the user with the selected external provider. These properties are configured with a callback URL, which will be called when the user has authenticated themselves with their chosen service. The AuthenticationProperties object is used to create a challenge response, which will start the authentication process and redirect the user to the selected service.

When the user has been authenticated, the OnGetCallbackAsync method will be invoked. The user's external information is obtained using the sign-in manager's GetExternalLoginInfoAsync method, which returns an ExternalLoginInfo object. The ExternalLoginInfo.Principal property returns a ClaimsPrincipal object that contains the user's account information, expressed as a series of claims. The user's email address is obtained by finding the first claim with the ClaimType.Email type, like this:

```
...
ExternalLoginInfo info = await SignInManager.GetExternalLoginInfoAsync();
string email = info?.Principal?.FindFirst(ClaimTypes.Email)?.Value;
...
```

Once I have an email address, I use it to create a new IdentityUser object and add it to the user store. I then use the FindByEmailAsync method so that I am working with the stored version of the object, including the properties that are generated automatically during the storage process. I call the user manager's AddLoginAsync method to store details of the external login in the store, so they can be used to sign into the application.

A redirection is performed that invokes the OnGetAsync method, which sets the property required to display the confirmation message to the user. For quick reference, Table 11-7 describes the sign-in methods used to create an account with an external login.

Table 11-7. *The SignInManager<IdentityUser> Methods for Registration with an External Login*

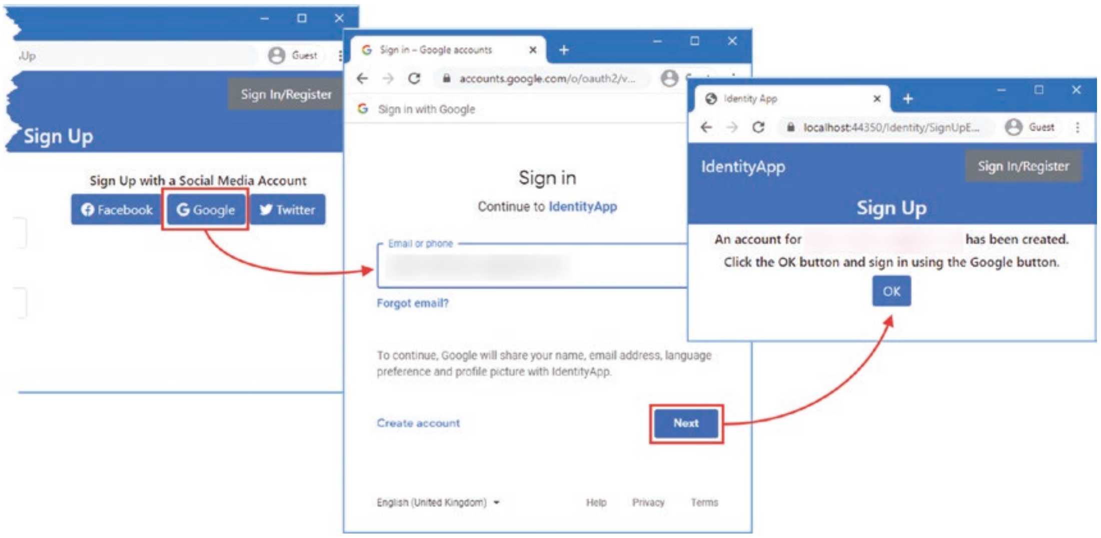
Name	Description
ConfigureExternalAuthenticationProperties (provider, url)	This method creates an AuthenticationProperties object for the specified provider and callback URL, which can then be used to create a challenge response.
GetExternalLoginInfoAsync()	This method returns a ExternalLoginInfo object that represents the user data provided by the external authentication service.

Table 11-8 describes the user manager method used to store the external login.

Table 11-8. *The UserManager<IdentityUser> Method for Storing an External Login*

Name	Description
AddLoginAsync(user, info)	This method stores an ExternalLoginInfo object for the specified IdentityUser.

Restart ASP.NET Core and request https://localhost:44350/Identity/SignUp; you will be presented with buttons for creating an account using Facebook, Google, and Twitter. Creating an account requires a real account with one of these providers, so click the button for a provider for which you have an account, and you will be redirected to a login page. Go through the sign-in process, and you will be redirected back to the example application, which will display a confirmation message. Figure 11-7 shows a basic sequence for a Google account, although you will see additional steps for passwords, authenticator tokens, SMS messages, or other factors based on your account.



To continue, Google will share your name, email address, language preference and profile picture with IdentityApp.

An account for [redacted] has been created.

Click the OK button and sign in using the Google button.

Figure 11-7. *Creating an account using an external authentication service*

Supporting Administrator Registration with an External Service

An external authentication service can be chosen by the user instead of a password when an account is created by an administrator. To start, make the changes shown in Listing 11-20 to the `UserAccountComplete` page to offer the user a choice between a password and an external authentication service. Just for variety, I have made all the changes in the view part of the page, rather than updating the page model class as well.

Listing 11-20. External Options in the UserAccountComplete.cshtml File in the Pages/Identity Folder

```
@page "{email?}/{token?}"
@model IdentityApp.Pages.Identity.UserAccountCompleteModel
@inject SignInManager<IdentityUser> SignInManager
@{
    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>
    <div class="container-fluid">
        <div class="row">
            <div class="col mb-3">
                <div class="form-group">
                    <label>Email</label>
                    <input class="form-control" asp-for="Email" readonly />
                </div>
            </div>
        </div>
        <div class="row">
            <div class="col-6">
                <h6>Sign In with a Password</h6>
                <form method="post">
                    <input type="hidden" asp-for="Token" />
                    <input type="hidden" asp-for="Email" />
                    <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>
            </div>
            <div class="col-auto">
                <h6>Sign In with a Social Media Account</h6>
                <form method="post" asp-page="UserAccountCompleteExternal">
                    <input type="hidden" asp-for="Email" />
                    <input type="hidden" asp-for="Token" />
                    @foreach (var scheme in await
                            SignInManager.GetExternalAuthenticationSchemesAsync()) {
                        <partial name="_ExternalButtonPartial" model="scheme" />
                    }
                </form>
            </div>
        </div>
    </div>
}
```

To handle using an external service to complete account setup, add a Razor Page named `UserAccountCompleteExternal.cshtml` to the `Pages/Identity` folder with the content shown in Listing 11-21.

Listing 11-21. The Contents of the UserAccountCompleteExternal.cshtml File in the Pages/Identity Folder

```
@page "{id?}"
@model IdentityApp.Pages.Identity.UserAccountCompleteExternalModel
@{
    ViewData["showNav"] = false;
    ViewData["banner"] = "Complete Account";
}
```

```
@if (TempData["errorMessage"] != null) {
        <div class="alert alert-danger">@TempData["errorMessage"]</div>
        <a asp-page="SignUp" class="btn btn-danger">Cancel Sign Up</a>
} else {
    <div class="text-center">
        <h6>
            Your account has been completed.
        </h6>
        <h6>Click the OK button and sign in using the
            @(await Model.ExternalProvider()) button.
        </h6>
        <a asp-page="SignIn" class="btn btn-primary">OK</a>
    </div>
}
```

This page is similar to the one defined in the previous section, but I have not attempted to reduce the duplication because most applications will require only one approach and it is easier to incorporate a workflow into a real project if it is self-contained. To define the page model class, add the code shown in Listing 11-22 to the `UserAccountCompleteExternal.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 11-22. The Contents of the UserAccountCompleteExternal.cshtml.cs File in the Pages/Identity Folder

```
using IdentityApp.Services;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using System.Security.Claims;
using System.Threading.Tasks;

namespace IdentityApp.Pages.Identity {

    [AllowAnonymous]
    public class UserAccountCompleteExternalModel : UserPageModel {

        public UserAccountCompleteExternalModel(
                UserManager<IdentityUser> usrMgr,
                SignInManager<IdentityUser> signMgr,
                TokenUrlEncoderService encoder) {
            UserManager = usrMgr;
            SignInManager = signMgr;
            TokenUrlEncoder = encoder;
        }

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

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

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

        public IdentityUser IdentityUser { get; set; }

        public async Task<string> ExternalProvider() =>
            (await UserManager.GetLoginsAsync(IdentityUser))
                .FirstOrDefault()?.ProviderDisplayName;

        public async Task<IActionResult> OnPostAsync(string provider) {
            IdentityUser = await UserManager.FindByEmailAsync(Email);
            string decodedToken = TokenUrlEncoder.DecodeToken(Token);
            bool valid = await UserManager.VerifyUserTokenAsync(IdentityUser,
                UserManager.Options.Tokens.PasswordResetTokenProvider,
                UserManager<IdentityUser>.ResetPasswordTokenPurpose, decodedToken);
            if (!valid) {
                return Error("Invalid token");
            }
            string callbackUrl = Url.Page("UserAccountCompleteExternal",
                "Callback", new { Email, Token });
            AuthenticationProperties props =
               SignInManager.ConfigureExternalAuthenticationProperties(
                   provider, callbackUrl);
            return new ChallengeResult(provider, props);
        }

        public async Task<IActionResult> OnGetCallbackAsync() {
            ExternalLoginInfo info = await SignInManager.GetExternalLoginInfoAsync();
            string email = info?.Principal?.FindFirst(ClaimTypes.Email)?.Value;
            if (string.IsNullOrEmpty(email)) {
                return Error("External service has not provided an email address.");
            } else if ((IdentityUser =
                    await UserManager.FindByEmailAsync(email)) == null) {
                return Error("Your email address doesn't match.");
            }
            IdentityResult result
                = await UserManager.AddLoginAsync(IdentityUser, info);
            if (!result.Succeeded) {
                return Error("Cannot store external login.");
            }
            return RedirectToPage(new { id = IdentityUser.Id });
        }

        public async Task<IActionResult> OnGetAsync(string id) {
            if ((id == null
                || (IdentityUser = await UserManager.FindByIdAsync(id)) == null)
                && !TempData.ContainsKey("errorMessage")) {
                return RedirectToPage("SignIn");
            }
            return Page();
        }

        private IActionResult Error(string err) {
            TempData["errorMessage"] = err;
            return RedirectToPage();
        }
    }
}
```

The basic approach is the same as for self-registration, but some changes are required to fit into the administrator-led model. First, this page has to validate the token that was sent to the user when the account was created. The token is intended for use with the method that resets a password but can be validated on its own, albeit awkwardly, with the user manager's `VerifyUserTokenAsync` method.

```
...
bool valid = await UserManager.VerifyUserTokenAsync(IdentityUser,
    UserManager.Options.Tokens.PasswordResetTokenProvider,
    UserManager<IdentityUser>.ResetPasswordTokenPurpose, decodedToken);
...
```

The arguments are the user object, the name of the token generator, the purpose for which the token was created, and, finally, the token. The name of the generator is obtained through the `UserManager.Options.Tokens.PasswordResetTokenProvider` property, and the `UserManager<IdentityUser>` class defines a set of constant values that describe the different reasons for which tokens are created. The result is that the token is validated without setting a password, which would undermine the purpose of the workflow.

The remaining differences arise because the external login is being added to an `IdentityUser` object that has already been stored by the administrator. The email address provided by the external service is checked to ensure it matches the one used by the administrator. For quick reference, Table 11-9 describes the method I used to validate the token without resetting the password.

Table 11-9. *The UserManager<IdentityUser> Method for Validating Tokens*

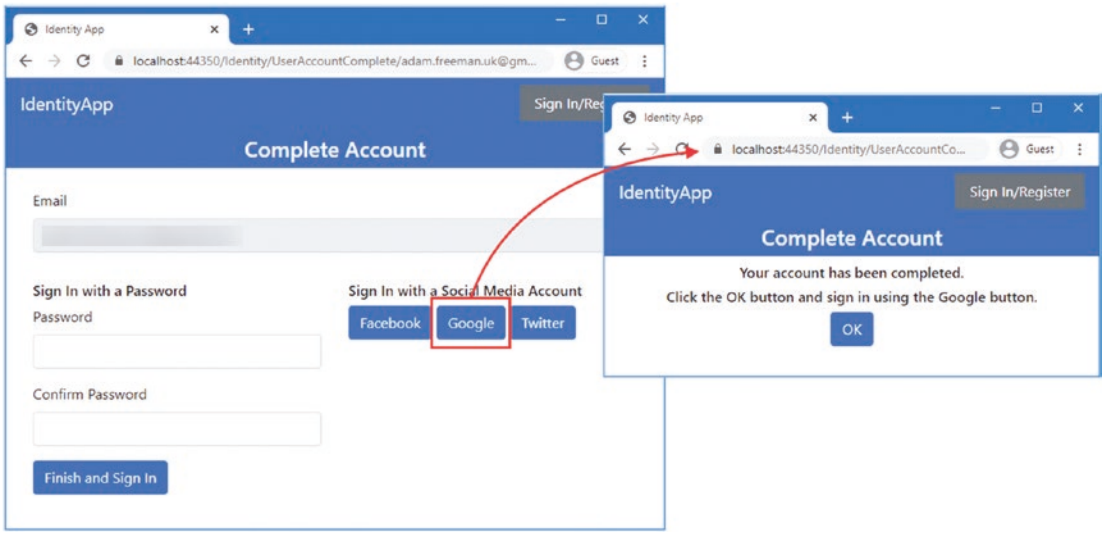
Name	Description
`VerifyUserTokenAsync(user, provider, purpose, token)`	This method returns `true` if the specified token is valid for the specified user and purpose and was created using the specified generator.

Restart ASP.NET Core and request `https://localhost:44350/Identity/Admin`. Unless you have a second external account available, click the Seed Database button to remove the account you created in the last section. Click the Create Account button, enter your external email address into the text field, and click the Create button. Examine the ASP.NET Core console output, and you will see a message similar to this one, but with your email address and a unique validation code:

```
---New Email----
To: user@domain
Subject: Set Your Password
Please set your password by
```

```
<a href=https://localhost:44350/Identity/UserAccountComplete/user@domain/Q2ZESjhBM>
    clicking here
</a>.
-------
```

Copy and paste the link from your message into a browser, and you will be prompted to either choose a password or use an external provider. After you have been authenticated by the external service, you will receive a confirmation message, as shown in Figure 11-8. You may go through an abbreviated authentication process if your browser stored cookies from the previous example. You may need to clear the cookies before you can authenticate as a different user.



Your account has been completed.
Click the OK button and sign in using the Google button.

Figure 11-8. *Using an external service to complete an account created by an administrator*

Supporting Signing In with an External Service

To allow users to sign in with an external account, start by making the changes shown in Listing 11-23 to present the user with the set of configured external services.

Listing 11-23. Displaying External Services 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>
}
```

```
<div class="container-fluid">
    <div class="row">
        <div class="col-6>">
            <h6>Sign In with a Password</h6>
            <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>
        </div>
        <div class="col-auto">
            <h6>Sign In with a Social Media Account</h6>
                <form method="post" asp-page="SignIn" asp-page-handler="External">
                    @foreach (var scheme in await
                       Model.SignInManager.GetExternalAuthenticationSchemesAsync()) {
                        <partial name="_ExternalButtonPartial" model="scheme" />
                    }
                </form>
        </div>
    </div>
</div>
```

To allow the user to sign in, make the changes shown in Listing 11-24 to the `SignIn.cshtml.cs` file.

Listing 11-24. Supporting External Services 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;
using Microsoft.AspNetCore.Authentication;
using System.Net;

namespace IdentityApp.Pages.Identity {

    [AllowAnonymous]
    public class SignInModel : UserPageModel {
        // ...methods and properties omitted for brevity...

        public IActionResult OnPostExternalAsync(string provider) {
            string callbackUrl = Url.Page("SignIn", "Callback", new { ReturnUrl });
            AuthenticationProperties props =
               SignInManager.ConfigureExternalAuthenticationProperties(
                   provider, callbackUrl);
            return new ChallengeResult(provider, props);
        }

        public async Task<IActionResult> OnGetCallbackAsync() {
            ExternalLoginInfo info = await SignInManager.GetExternalLoginInfoAsync();
            SignInResult result = await SignInManager.ExternalLoginSignInAsync(
                info.LoginProvider, info.ProviderKey, true);
            if (result.Succeeded) {
                return Redirect(WebUtility.UrlDecode(ReturnUrl ?? "/"));
            } else if (result.IsLockedOut) {
                TempData["message"] = "Account Locked";
            } else if (result.IsNotAllowed) {
                TempData["message"] = "Sign In Not Allowed";
            } else {
                TempData["message"] = "Sign In Failed";
            }
            return RedirectToPage();
        }
    }
}
```

The OnPostExternalAsync handler is called when the user clicks one of the external service buttons and produces the same type of challenge result shown in earlier examples. The OnGetCallbackAsync method is called once the user has been authenticated. The user's details from the external service are obtained using the GetExternalLoginInfoAsync method, which was also used in earlier examples. The sign-in manager's ExternalLoginSignInAsync method is used to sign the user into the application using the stored external login. For quick reference, Table 11-10 describes these methods.

Table 11-10. *The SignInManager<IdentityUser> Methods for External Signing In*

Name	Description
GetExternalLoginInfoAsync()	This method returns a ExternalLoginInfo object that represents the user data provided by the external authentication service.
ExternalLoginSignInAsync(provider, key, persist)	This method signs the user into the application using a previously stored external login. The arguments are taken from the ExternalLoginInfo object returned by the GetExternalLoginInfoAsync method.

Restart ASP.NET Core, request `https://localhost:44350/Identity/SignIn`, and click the external service button you used to create an account in one of the previous sections. Once you are authenticated, you will be signed into the application, as shown in Figure 11-9. You may not be prompted to enter any credentials if your browser has previously received cookies from the external service.

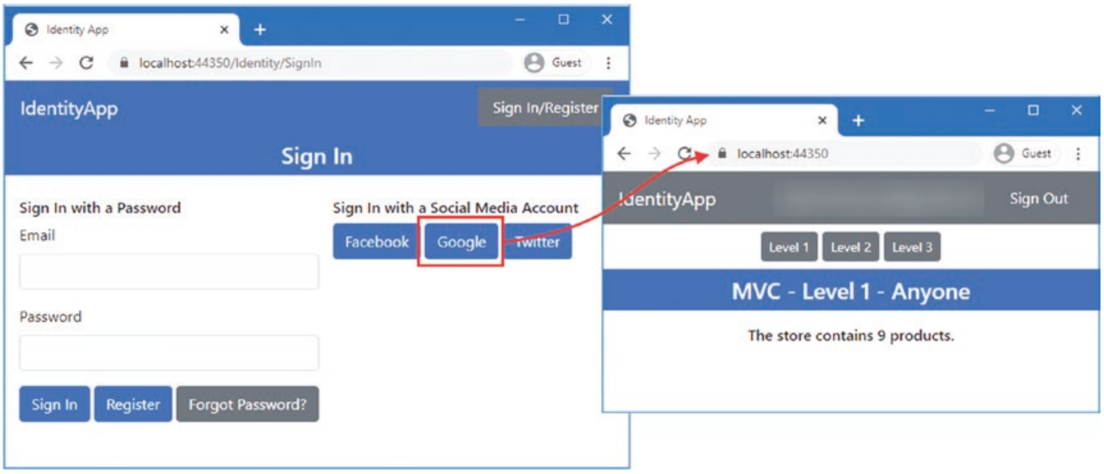


Figure 11-9. Signing into the application with an external service

Summary

In this chapter, I created custom workflows that support two-factor authentication with an authenticator and for registering and signing in with a third-party authentication. In the next chapter, I explain how to provide authentication services for API clients.