CHAPTER 6

Adapting Identity UI

The Identity UI package has been designed for self-service applications that support authenticators for two-factor authentication and external authentication with third-party services. There is some flexibility within that design so that individual features can be altered or disabled, and new features can be introduced. In this chapter, I explain the process for adapting the Identity UI package, showing the different ways it can be customized.

There are limits to the extent of the changes, however, and in Chapters 7 to 12, I show you how to work directly with the API that Identity provides to create completely custom workflows that replace the Identity UI package. Table 6-1 puts adapting Identity UI in context.

Table 6-1. *Putting Identity UI Adaptations in Context*

Question	Answer
What are they?	Adaptations allow the files in the Identity UI package to be added to the project so they can be modified, allowing features to be created, customized, or disabled.
Why are they useful?	If your project almost fits into the general model expected by the Identity UI package, adaptations can customize Identity UI to make it fit your needs exactly.
How are they used?	Razor Pages, views, and other files are added to the project using a process known as *scaffolding*. Scaffolded files are given precedence over those in the Identity UI package, which means that changes made to the scaffolded files become part of the content presented to the user.
Are there any pitfalls or limitations?	Although you can customize individual features, you cannot adjust the underlying approach taken by the Identity UI package, which means there are limits to extent of the changes you can make. Some scaffolding operations overwrite files even if they have been changed, requiring an awkward shuffling of files to protected customizations.
Are there any alternatives?	Identity provides an API that can be used to create custom workflows, as described in Chapters 7 to 12.

© Adam Freeman 2021
A. Freeman, *Pro ASP.NET Core Identity*, https://doi.org/10.1007/978-1-4842-6858-2_6

Table 6-2 summarizes the chapter.

Table 6-2. *Chapter Summary*

Problem	Solution	Listing
Scaffold an Identity UI file	Install the code generator global tool package, add the Identity UI scaffolding package to the project, and use the dotnet `aspnet-codegenerator` command to scaffold the files you require.	1–13
Scaffold the account management features or appearance	Use the scaffolding command to select files whose name starts with `Account.Manage`, which corresponds to the `Areas/Identity/Pages/Account/Manage` folder.	14–16, 20, 21
Add a new feature the Identity UI	Scaffold the navigation class and Razor Page and add links to a Razor Page in which you implement the new feature.	17–19
Scaffold new files after changing shared views or classes	Move the modified files into a safe location and scaffold additional files without causing the project to be compiled.	22–26
Disable an Identity UI feature	Scaffold the file that defines the feature and replace the handler methods in the page model class. Scaffold the files that contain navigation links to the disabled feature and remove them.	27–29

Preparing for This Chapter

This chapter uses the IdentityApp project from Chapter 5. Open a new PowerShell command prompt and run the commands shown in Listing 6-1 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 6-1. 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 6-2 in the `IdentityApp` folder to start the application.

Listing 6-2. Running the Example Application

```
dotnet run
```

Open a web browser and request `https://localhost:44350`, which will show the output from the `Home` controller, and `https://localhost:44350/pages`, which will show the output from the `Landing` Razor Page, as shown in Figure 6-1.

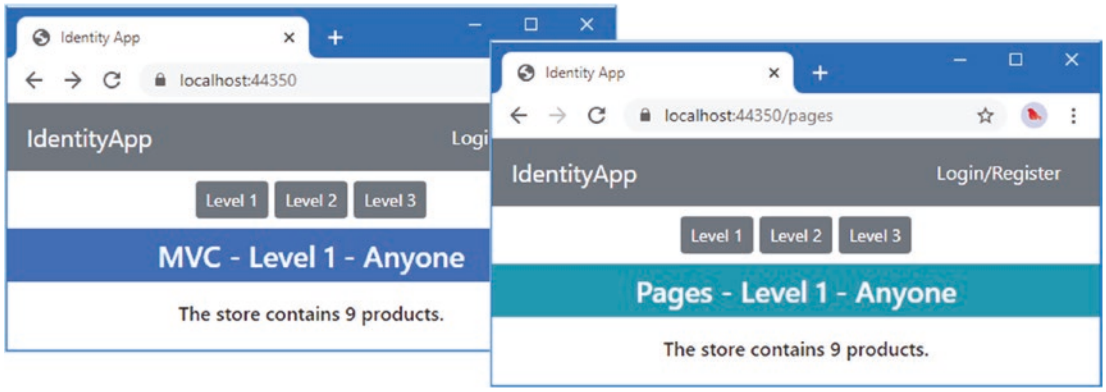


Figure 6-1. *Running the example application*

Click the Login/Register link, click Register as a New User, and create a user account with the details shown in Table 6-3.

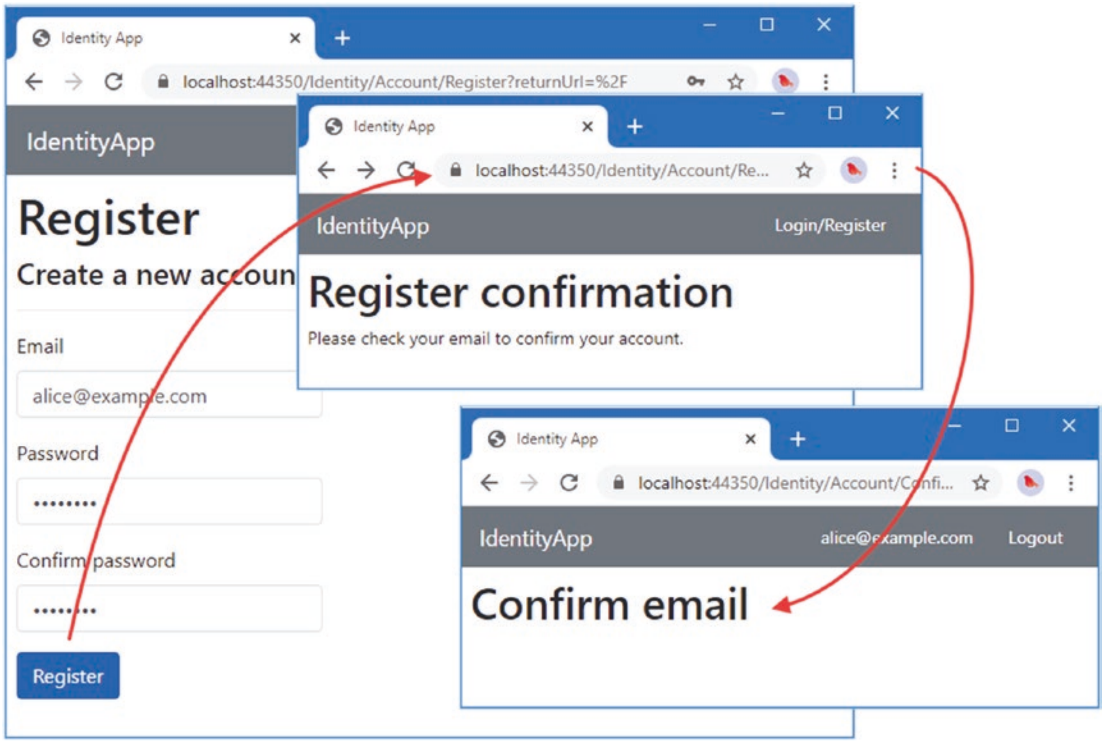
Table 6-3. *Account Details*

Field	Value
Email	`alice@example.com`
Password	`mysecret`

Click the Register button to create the account. Check the ASP.NET Core console output, and you will see a confirmation email, similar to this one:

```
---New Email----
To: alice@example.com
Subject: Confirm your email
Please confirm your account by <a href='https://localhost:44350
/Identity/Account/ConfirmEmail?userId=395a955f &returnUrl=%2F'>
    clicking here
</a>.
-------
```

Copy the URL into a browser to confirm the account so that you can sign in again, as shown in Figure 6-2.



localhost:44350/Identity/Account/Register?returnUrl=%2F

Register

Create a new accoun

alice@example.com

localhost:44350/Identity/Account/Re...

Register confirmation

Please check your email to confirm your account.

localhost:44350/Identity/Account/Confi...

Confirm email

Figure 6-2. *Creating and confirming an account*

Understanding Identity UI Scaffolding

The Identity UI package uses the ASP.NET Core areas feature, which allows Razor Pages defined in the project to override those in the UI package, just as long as they are in a specific folder, which is `Areas/Identity/Pages`. This folder already exists in the project, and it contains the Razor View Start file I created to enforce a consistent layout.

You can simply create Razor Pages whose names match those described in the workflows in Chapter 4, and they will override the pages in the Identity UI package, but this means you are responsible for re-creating the HTML content and C# code that provides the features of the original page. To make it easy to build on the existing features, the Identity UI package will create replacement Razor Pages in your project, so you can modify features rather than re-creating them completely. This process is known as *scaffolding*.

Preparing for Identity UI Scaffolding

The scaffolding process relies on a global .NET tool package. Use a PowerShell command prompt to run the commands shown in Listing 6-3 to remove any existing version of the package and install the version required for this chapter. (You may receive an error from the first command if you have not previously installed this package.)

Listing 6-3. Installing the Scaffolding Tool Package

```
dotnet tool uninstall --global dotnet-aspnet-codegenerator
dotnet tool install --global dotnet-aspnet-codegenerator --version 5.0.0
```

An additional package must be added to the project to provide the global tool with the templates it needs to create new items. Use the PowerShell prompt to run the command shown in Listing 6-4 in the `IdentityApp` folder.

Listing 6-4. Adding the Scaffolding Package

```
dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design --version 5.0.0
```

Listing the Identity UI Pages for Scaffolding

The command-line scaffolding tool allows individual pages to be scaffolded. You can see a complete list of the Razor Pages available for scaffolding by running the command shown in Listing 6-5 in the `IdentityApp` folder.

OTHER USES FOR THE IDENTITY SCAFFOLDING TOOL

The scaffolding tool can also be used to add Identity to a project, including generating database context classes and adding statements to the `Startup` class. I do not describe these features because it is important to understand how Identity is configured to ensure you get the features you expect, even when using the Identity UI package to provide a user experience. You can see the list of options by running the `dotnet aspnet-codegenerator identity` command in a project that has been configured for use with the Identity scaffold tool, but my advice is to examine the results carefully to make sure you produced the effect you expected.

Listing 6-5. Listing the Pages Available for Scaffolding

```
dotnet aspnet-codegenerator identity --listFiles
```

Here is the first part of the output, which corresponds to the Razor Pages described in Chapter 5:

```
...
Account._StatusMessage
Account.AccessDenied
Account.ConfirmEmail
Account.ConfirmEmailChange
Account.ExternalLogin
Account.ForgotPassword
Account.ForgotPasswordConfirmation
Account.Lockout
Account.Login
Account.LoginWith2fa
...
```

To use the scaffolding feature, determine which pages relate to the features that interest you from the workflows described in Chapter 5 and use one of the techniques described in the following sections to disable, change, or add features.

Using the Identity UI Scaffolding

In the sections that follow, I show you how to use the scaffolding feature to change the Razor Pages that Identity UI uses.

Using Scaffolding to Change HTML

The simplest way to use the scaffolding feature is to change only the HTML produced by a Razor UI page. In this section, I will use this technique to improve the appearance of the buttons that allow signing in with external services, such as Google and Facebook. To prepare, run the command shown in Listing 6-6 in the `IdentityApp` folder to install the Font Awesome package, which includes company logos for the main platforms.

Listing 6-6. Installing the Font Awesome Package

```
libman install font-awesome@5.15.1 -d wwwroot/lib/font-awesome
```

Add the element shown in Listing 6-7 to the layout used by the Identity UI package to incorporate the Font Awesome styles into the HTML sent to the browser.

Listing 6-7. Adding an Element in the _CustomIdentityLayout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Identity App</title>
    <link rel="stylesheet" href="/Identity/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="/Identity/css/site.css" />
    <script src="/Identity/lib/jquery/dist/jquery.js"></script>
    <script src="/Identity/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    <script src="/Identity/js/site.js" asp-append-version="true"></script>
    <script type="text/javascript" src="/lib/qrcode/qrcode.min.js"></script>
    <link href="/lib/font-awesome/css/all.min.css" rel="stylesheet" />
</head>
```

```
<body>
    <nav class="navbar navbar-dark bg-secondary">
        <a class="navbar-brand text-white">IdentityApp</a>
        <div class="text-white"><partial name="_LoginPartial" /></div>
    </nav>
    <div class="m-2">
        @RenderBody()
        @await RenderSectionAsync("Scripts", required: false)
    </div>
    <script type="text/javascript">
        var element = document.getElementById("qrCode");
        if (element !== null) {
            new QRCode(element, {
                text: document.getElementById("qrCodeData").getAttribute("data-url"),
                width: 150, height: 150
            });
            element.previousElementSibling?.remove();
        }
    </script>
</body>
</html>
```

Add a Razor View named `_ExternalButtonPartial.cshtml` to the `Views/Shared` folder and use it to define the partial view shown in Listing 6-8.

Listing 6-8. The Contents of the _ExternalButtonPartial.cshtml File in the Views/Shared Folder

```
@model Microsoft.AspNetCore.Authentication.AuthenticationScheme

<button type="submit"
        class="btn btn-primary" name="provider" value="@Model.Name">
    <i class="@($"fab fa-{Model.Name.ToLower()}")"></i>
    @Model.DisplayName
</button>
```

The model for the partial view is an `AuthenticationScheme` object, which is how ASP.NET Core describes an authentication option, with `Name` and `DisplayName` properties. (The `AuthenticationScheme` class—and many, many other classes—are described in Part 2.)

The partial view renders an HTML button that has an icon from the Font Awesome package, which is added with the `i` element.

Now scaffold the page that displays the login buttons to the user by running the command shown in Listing 6-9 in the `IdentityApp` folder.

Listing 6-9. Scaffolding an Identity UI Razor Page

```
dotnet aspnet-codegenerator identity --dbContext Microsoft.AspNetCore.Identity.
EntityFrameworkCore.IdentityDbContext --files Account.Login
```

The dotnet `aspnet-codegenerator identity` selects the Identity UI scaffolding tool. The `--dbContext` argument is used to specify the Entity Framework Core database context class. This argument must specify the complete name, including the namespace, of the context class used by the application. If the name does not match exactly, the scaffolding tool will create a new database context class, which will lead to inconsistent results later. The `--files` argument specifies the files that should be scaffolded, using one or more names from the list produced in the previous section, separated by semicolons. I have selected the `Account.Login` page, which is responsible for presenting users with the external authentication buttons.

The command in Listing 6-9 adds several files to the project. The `Areas/Identity/IdentityHostingStartup.cs` file is used to set up features that are specific to the Identity UI package but that should contain only an empty `ConfigureServices` method for this chapter, like this:

```
using Microsoft.AspNetCore.Hosting;

[assembly: HostingStartup(typeof(IdentityApp.Areas.Identity.IdentityHostingStartup))]

namespace IdentityApp.Areas.Identity {

    public class IdentityHostingStartup : IHostingStartup {

        public void Configure(IWebHostBuilder builder) {
            builder.ConfigureServices((context, services) => {
            });
        }
    }
}
```

If you did not correctly specify the context class when running the command in Listing 6-9, you will see statements in this class that register a newly created context. If that happens, the simplest approach is to delete the entire `Areas` folder, re-create it with the Razor View Start file, and run the command in Listing 6-9 again.

The command also creates view import files, a partial view containing validation script references, and, of course, the specified Razor Page and its page model class. Edit the contents of the `Login.cshtml` file in the `Areas/Identity/Pages/Account` folder to replace the `button` element with a `partial` element that applies the partial view, as shown in Listing 6-10.

Listing 6-10. Applying a Partial in the Login.cshtml File in the Areas/Identity/Pages/Account Folder

```
...
<section>
    <h4>Use another service to log in.</h4>
    <hr />
    @{
       if ((Model.ExternalLogins?.Count ?? 0) == 0) {
           <div>
               <p>There are no external authentication services configured. See
               <a href="https://go.microsoft.com/fwlink/?LinkID=532715">
                    this article</a>
                for details on setting up this ASP.NET application to support
                logging in via external services.
                </p>
            </div>
        } else {
            <form id="external-account" asp-page="./ExternalLogin"
                 asp-route-returnUrl="@Model.ReturnUrl" method="post"
                 class="form-horizontal">
                <div>
                    <p>
                        @foreach (var provider in Model.ExternalLogins) {
                            <partial name="_ExternalButtonPartial"
                               model="provider" />
                        }
                    </p>
                </div>
            </form>
        }
    }
</section>
...
```

The content in the Login Razor Page is verbose, so I have shown only the part that should be changed. It can take a little effort to figure out which section of HTML relates to a specific feature, but just as long as you are careful to preserve the attributes and tag helpers on elements, changes are generally easy to make.

Some changes have to be applied in multiple places. In the case of the external authentication buttons, I also need to change the `Account/Register` page, which also presents buttons for the configured services. Run the command shown in Listing 6-11 in the `IdentityApp` to scaffold the page.

Listing 6-11. Scaffolding an Identity UI Razor Page

```
dotnet aspnet-codegenerator identity --dbContext Microsoft.AspNetCore.Identity.
EntityFrameworkCore.IdentityDbContext --files Account.Register
```

Edit the contents of the `Register.cshtml` file in the `Areas/Identity/Pages/Account` folder to replace the `button` element with a `partial` element that applies the partial view, as shown in Listing 6-12. This is the same change I applied to the `Login` page because the Identity UI package duplicates the markup instead of using a partial view.

Listing 6-12. Applying a Partial View in the Register.cshtml File in the Areas/Identity/Pages/Account Folder

```
...
<section>
    <h4>Use another service to register.</h4>
    <hr />
    @{
        if ((Model.ExternalLogins?.Count ?? 0) == 0) {
            <div>
                <p>
                    There are no external authentication services configured.
                    See <a href="https://go.microsoft.com/fwlink/?LinkID=532715">
                    this article</a>
                    for details on setting up this ASP.NET application to support
                    logging in via external services.
                </p>
            </div>
        } else {
            <form id="external-account" asp-page="./ExternalLogin"
                 asp-route-returnUrl="@Model.ReturnUrl" method="post"
                 class="form-horizontal">
                <div>
                    <p>
                        @foreach (var provider in Model.ExternalLogins) {
                            <partial name="_ExternalButtonPartial"
                                 model="provider" />
                        }
                    </p>
                </div>
            </form>
        }
    }
</section>
...
```

Restart ASP.NET Core and `https://localhost:44350/Identity/Account/Login`. The scaffolded page takes precedence over the one in the Identity UI package, and the external authentication buttons are displayed with appropriate icons. The same icons are displayed if you click the Register As a New User link, as shown in Figure 6-3.

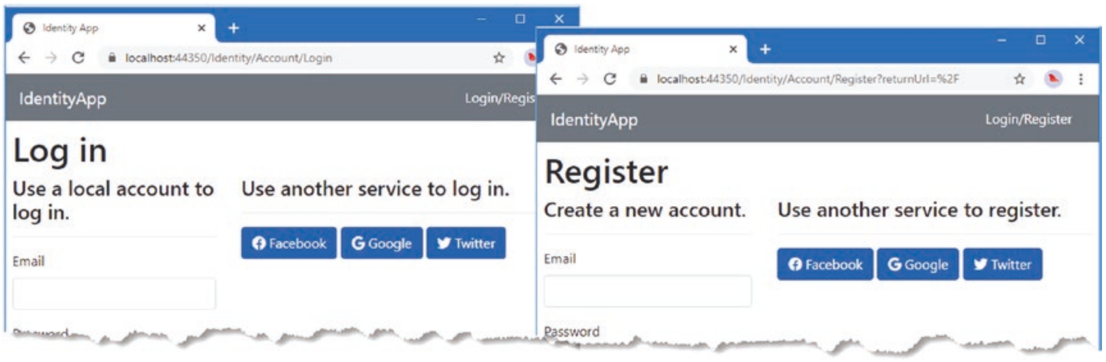

Identity App
localhost:44350/Identity/Account/Login
IdentityApp
Login/Regis
Log in
Use a local account to log in.
Use another service to log in.
Facebook
Google
Twitter
Email
Password
Identity App
localhost:44350/Identity/Account/Register?returnUrl=%2F
IdentityApp
Login/Register
Register
Create a new account.
Use another service to register.
Facebook
Google
Twitter
Email
Password

Figure 6-3. *Using scaffolding to change HTML*

Using Scaffolding to Modify C# Code

Scaffolding doesn't just override the view part of a Razor Page. It also creates a page model class containing the C# code that implements the features presented by the page. The command that scaffolded the `Login` page created a `Login.cshtml.cs` file in the `Areas/Identity/Pages/Account` folder, and changing the code in this file will alter how users are signed into the application.

Making changes to the page model class often requires knowledge of the Identity API, and care must be taken to understand the impact of the changes you intend to make. I describe the Identity API features in Chapters 7–12 and describe almost every aspect of the API in detail in Part 2, but for now, there is one change that can be made without needing to get into details. Locate the `OnPostAsync` method in the `Login.cshtml.cs` file and change the final argument to the `PasswordSignInAsync` method, as shown in Listing 6-13.

Listing 6-13. Changing an Argument in the Login.cshtml.cs File in the Areas/Identity/Pages/Account Folder

```
...
public async Task<IActionResult> OnPostAsync(string returnUrl = null) {
    returnUrl ??= Url.Content("~/");

    ExternalLogins = (await _signInManager.GetExternalAuthenticationSchemesAsync())
                    .ToList();

    if (ModelState.IsValid) {
        // This doesn't count login failures towards account lockout
        // To enable password failures to trigger account lockout,
        // set lockoutOnFailure: true
        var result = await _signInManager.PasswordSignInAsync(Input.Email,
            Input.Password, Input.RememberMe, lockoutOnFailure: true);
        if (result.Succeeded) {
            _logger.LogInformation("User logged in.");
            return LocalRedirect(returnUrl);
        }
        if (result.RequiresTwoFactor) {
            return RedirectToPage("./LoginWith2fa", new { ReturnUrl = returnUrl,
                RememberMe = Input.RememberMe });
        }
        if (result.IsLockedOut) {
            _logger.LogWarning("User account locked out.");
            return RedirectToPage("./Lockout");
        } else {
            ModelState.AddModelError(string.Empty, "Invalid login attempt.");
            return Page();
        }
    }

    // If we got this far, something failed, redisplay form
    return Page();
}
...
```

By default, the `Login` page signs users into the application so that failed attempts do not lead to lockouts. The change in Listing 6-13 alters this behavior so that failed attempts count toward a lockout, based on the configuration options described in Chapter 5.

Restart ASP.NET Core and request `https://localhost:44350/Identity/Account/Login`. Enter alice@example.com into the email field and notmypassword into the password field. Click the Log In button, and you will receive an error message telling you that the login has failed. Repeat this process, attempting to sign in using `notmypassword` each time.

After five failed attempts, the account will be locked out, and the browser is redirected to the `Lockout` page, as shown in Figure 6-4. Further attempts to sign in, even with the correct password, will fail until the lockout expires.

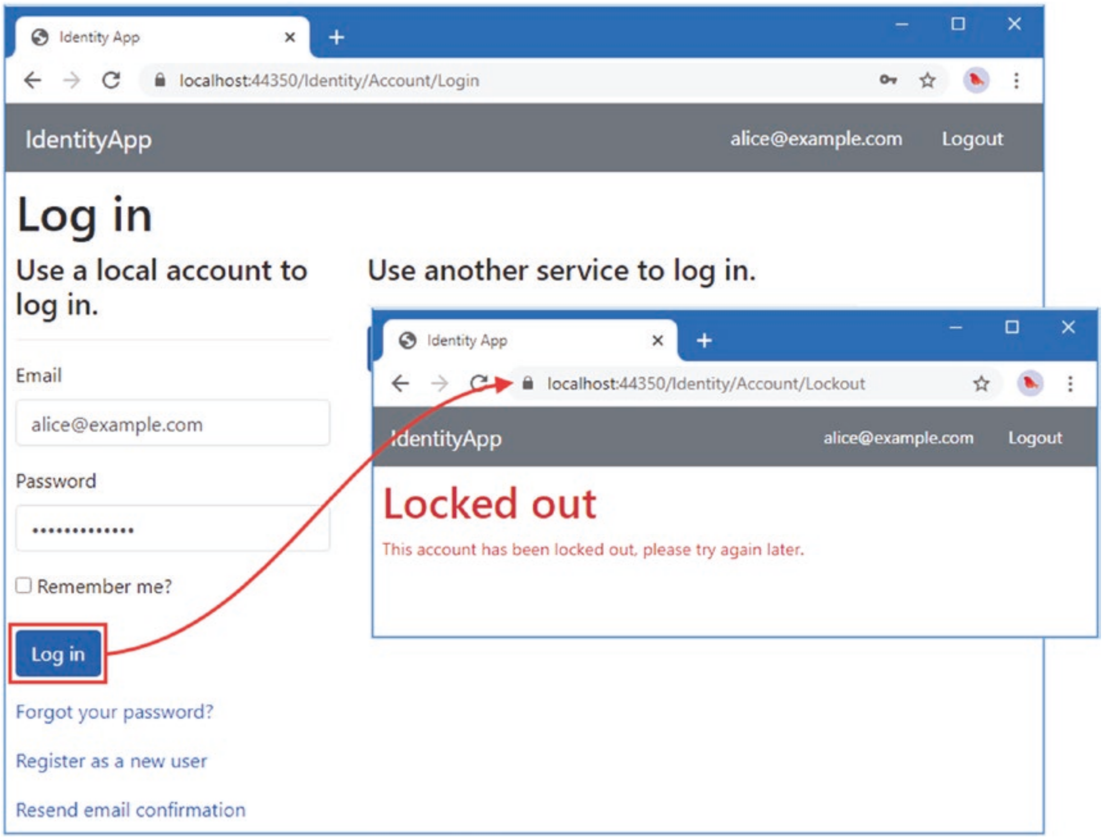

Identity App
localhost:44350/Identity/Account/Login
IdentityApp
alice@example.com
Logout
Log in
Use a local account to log in.
Use another service to log in.
Email
alice@example.com
Password
Remember me?
Log in
Forgot your password?
Register as a new user
Resend email confirmation
Identity App
localhost:44350/Identity/Account/Lockout
IdentityApp
alice@example.com
Logout
Locked out
This account has been locked out, please try again later.

Figure 6-4. *Locking out an account*

Configuring the Account Management Pages

The Identity UI package uses a layout and partial view to present the navigation links for the self-management features. To see the default layout, shown in Figure 6-5, sign in to the application using `alice@example.com` as the email address and `mysecret` as the password and click the email address shown in the header. (You may have to wait until the lockout from the previous section has expired. The lockout lasts for 5 minutes.)

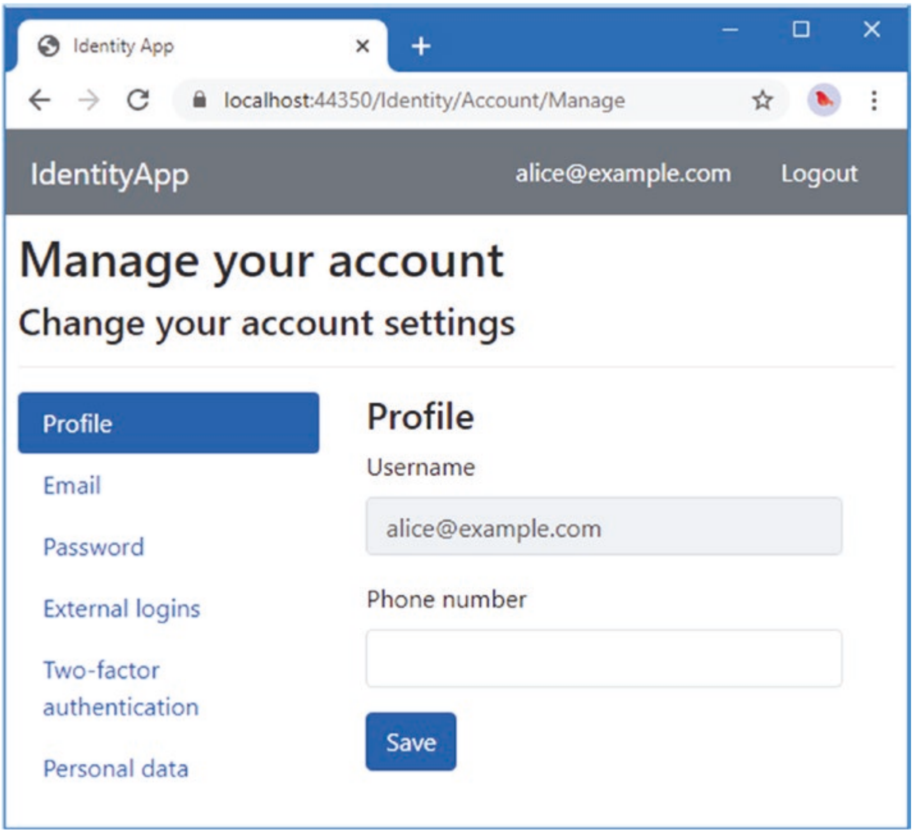


Figure 6-5. *The account self-management features*

The Identity UI Razor Pages for account management are defined in the `Areas/Identity/Pages/Account/Manage` folder and have the `Account.Manage` prefix when listed with the scaffolding tool. Use a PowerShell command prompt to run the command shown in Listing 6-14 in the `IdentityApp` folder to display only the management pages.

Listing 6-14. Listing the Management Pages

```
dotnet aspnet-codegenerator identity --listFiles | Where-Object {$_ -like '*Manage*'}
```

This command produces the following output:

```
Account.Manage._Layout
Account.Manage._ManageNav
Account.Manage._StatusMessage
Account.Manage.ChangePassword
Account.Manage.DeletePersonalData
Account.Manage.Disable2fa
Account.Manage.DownloadPersonalData
Account.Manage.Email
Account.Manage.EnableAuthenticator
Account.Manage.ExternalLogins
Account.Manage.GenerateRecoveryCodes
Account.Manage.Index
Account.Manage.PersonalData
Account.Manage.ResetAuthenticator
Account.Manage.SetPassword
Account.Manage.ShowRecoveryCodes
Account.Manage.TwoFactorAuthentication
```

In addition to the Razor Pages for specific features, the list contains two files that are useful in their own right: `Account.Manage._Layout` and `Account.Manage._ManageNav`. The `_Layout` file is the Razor Layout used by the management Razor Pages. The `_ManageNav` file is a partial view that generates the links on the left of the layout shown in Figure 6-6.

Run the command shown in Listing 6-15 to scaffold these two files. As with previous examples, take particular care with the name of the database context class.

Listing 6-15. Scaffolding the Management Layout Files

```
dotnet aspnet-codegenerator identity --dbContext
Microsoft.AspNetCore.Identity.EntityFrameworkCore.IdentityDbContext --files
"Account.Manage._Layout;Account.Manage._ManageNav"
```

This command creates the `Areas/Identity/Pages/Account/Manage` folder and populates it with the `_Layout.cshtml`, `_ManageNav.cshtml`, and `ManageNavPages.cs` files. There is also a Razor View Imports file, which is used to import the classes used by the Razor Pages in the `Manage` folder, but which isn't needed for this chapter.

Changing the Management Layout

The `_Layout.cshtml` file created by the command in Listing 6-15 is used to present a consistent layout for all the account management Razor Pages. Listing 6-16 replaces the header text, just to make an easily discernible change.

Listing 6-16. Changing Text in the _Layout.cshtml File in the Areas/Identity/Pages/Account/Manage Folder

```
@{
    if (ViewData.TryGetValue("ParentLayout", out var parentLayout)) {
        Layout = (string)parentLayout;
    } else {
        Layout = "/Areas/Identity/Pages/_Layout.cshtml";
    }
}
<h2>Account Self-Management</h2>
<div>
    <h4>Change your account settings</h4>
    <hr />
    <div class="row">
```

```
        <div class="col-md-3"><partial name="_ManageNav" /></div>
        <div class="col-md-9">@RenderBody()</div>
    </div>
</div>
@section Scripts {
    @RenderSection("Scripts", required: false)
}
```

This layout displays the contents of the _ManageNav.cshtml partial view, which I describe in the next section, and uses the RenderBody method to display the content generated by the Razor Page. The code block at the top of the layout allows the base layout to be selected dynamically using view data, also described in the next section. Restart ASP.NET Core, sign into the application as alice@example.com with the password mysecret, and click the email address displayed in the header to see the modified content, as shown in Figure 6-6.

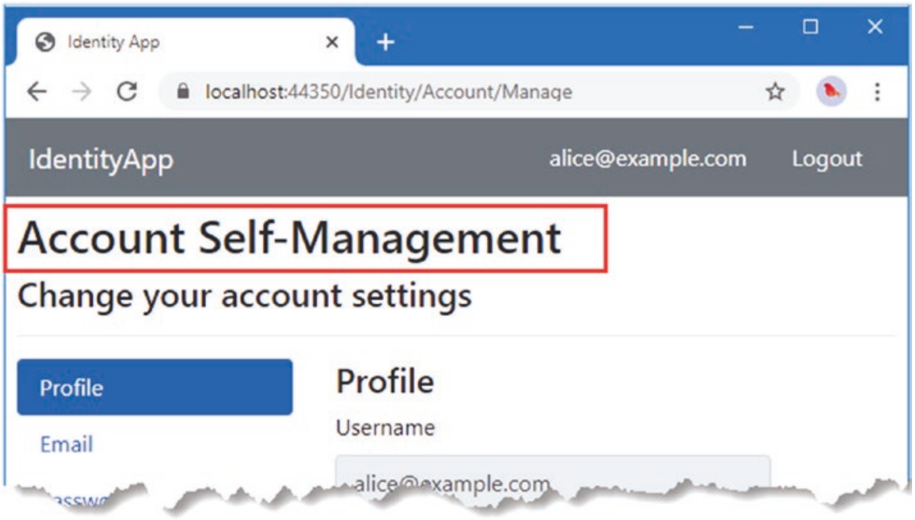


Figure 6-6. *Modifying the self-management layout*

Adding an Account Management Page

In this section, I am going to demonstrate the process of adding a new management Razor Page, which requires a little additional effort to integrate it into the rest of the management layout.

Preparing the Navigation Link

To add a new page to the management interface, the first step is to modify the ManageNavPages class, which is used to keep track of the selected page so that the appropriate link is highlighted in the layout. Make the changes shown in Listing 6-17 to prepare for a new page that will be called StoreData.

Listing 6-17. Preparing for a New Page in the ManageNavPages.cs File in the Areas/Identity/Pages/Account/Manage Folder

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System;

namespace IdentityApp.Areas.Identity.Pages.Account.Manage {

    public static class ManageNavPages {
        public static string Index => "Index";

        public static string Email => "Email";

        public static string ChangePassword => "ChangePassword";

        public static string DownloadPersonalData => "DownloadPersonalData";

        public static string DeletePersonalData => "DeletePersonalData";

        public static string ExternalLogins => "ExternalLogins";

        public static string PersonalData => "PersonalData";

        public static string TwoFactorAuthentication => "TwoFactorAuthentication";

        public static string StoreData => "StoreData";

        public static string IndexNavClass(ViewContext viewContext)
            => PageNavClass(viewContext, Index);

        public static string EmailNavClass(ViewContext viewContext)
            => PageNavClass(viewContext, Email);

        public static string ChangePasswordNavClass(ViewContext viewContext)
            => PageNavClass(viewContext, ChangePassword);

        public static string DownloadPersonalDataNavClass(ViewContext viewContext)
            => PageNavClass(viewContext, DownloadPersonalData);

        public static string DeletePersonalDataNavClass(ViewContext viewContext)
           => PageNavClass(viewContext, DeletePersonalData);

        public static string ExternalLoginsNavClass(ViewContext viewContext)
           => PageNavClass(viewContext, ExternalLogins);

        public static string PersonalDataNavClass(ViewContext viewContext)
           => PageNavClass(viewContext, PersonalData);

        public static string TwoFactorAuthenticationNavClass(ViewContext viewContext)
           => PageNavClass(viewContext, TwoFactorAuthentication);
        public static string StoreDataNavClass(ViewContext viewContext)
            => PageNavClass(viewContext, StoreData);

        private static string PageNavClass(ViewContext viewContext, string page) {
            var activePage = viewContext.ViewData["ActivePage"] as string
                ?? System.IO.Path.GetFileNameWithoutExtension(
                       viewContext.ActionDescriptor.DisplayName);
            return string.Equals(activePage, page,
                StringComparison.OrdinalIgnoreCase) ? "active" : null;
        }
    }
}
```

The first part of the ManageNavPages class is a set of read-only string properties for each of the Razor Pages for which links are displayed. These properties make it easy to replace the default pages without breaking the way the links are displayed.

The next section is a set of methods used by the _ManageNav partial to set the classes for the link elements for each page. These methods used the private PageNavClass method to return the string active if the page they represent has been selected, which is determined by reading a view data property named ActivePage.

Adding the Navigation Link

The _ManageNav partial view that was scaffolded by the command in Listing 6-17 presents the navigation links for the individual management Razor Pages. The Index page in the Areas/Identity/Pages/Account/Manage folder is presented by default, and there are links for changing email address, changing password, managing external authentication, configuring two-factor authentication, and managing personal data. The next step is to add a link to the _ManageNav.cshtml partial view for the new Razor Page, as shown in Listing 6-18.

■ **Note** The _ManageNav partial uses the GetExternalAuthenticationSchemesAsync method defined by the SignInManager<IdentityUser> class to determine whether the current user has logged in using an external authentication service. I describe this method in Chapter 11.

Listing 6-18. Adding a Link in the _ManageNav.cshtml File in the Areas/Identity/Pages/Account/Manage Folder

```
@inject SignInManager<IdentityUser> SignInManager
@{
    var hasExternalLogins = (await SignInManager
        .GetExternalAuthenticationSchemesAsync()).Any();
}
<ul class="nav nav-pills flex-column">
    <li class="nav-item">
        <a class="nav-link @ManageNavPages.IndexNavClass(ViewContext)"
            id="profile" asp-page="./Index">Profile</a>
    </li>
    <li class="nav-item">
        <a class="nav-link @ManageNavPages.StoreDataNavClass(ViewContext)"
             id="personal-data" asp-page="./StoreData">Store Data</a>
    </li>
    <li class="nav-item">
        <a class="nav-link @ManageNavPages.EmailNavClass(ViewContext)"
            id="email" asp-page="./Email">Email</a>
    </li>
    <li class="nav-item">
        <a class="nav-link @ManageNavPages.ChangePasswordNavClass(ViewContext)"
             id="change-password" asp-page="./ChangePassword">Password</a>
    </li>
    @if (hasExternalLogins) {
        <li id="external-logins" class="nav-item">
            <a id="external-login" class="nav-link
                 @ManageNavPages.ExternalLoginsNavClass(ViewContext)"
                 asp-page="./ExternalLogins">External logins</a>
        </li>
    }
    <li class="nav-item">
         <a class="nav-link
             @ManageNavPages.TwoFactorAuthenticationNavClass(ViewContext)"
             id="two-factor" asp-page="./TwoFactorAuthentication">
                  Two-factor authentication</a>
    </li>
    <li class="nav-item">
        <a class="nav-link @ManageNavPages.PersonalDataNavClass(ViewContext)"
             id="personal-data" asp-page="./PersonalData">Personal data</a>
    </li>
</ul>
```

The anchor element provides a link to the StoreData page, and its class attribute is set using the StoreDataNavClass method added to the ManageNavPages class earlier. This method returns active if the StoreData page has been selected, which is the Bootstrap CSS class for active links.

Defining the New Razor Page

Add a Razor Page named StoreData.cshtml to the Areas/Identity/Pages/Account/Manage folder with the content shown in Listing 6-19.

Listing 6-19. The Contents of the StoreData.cshtml File in the Areas/Identity/Pages/Account/Manage Folder

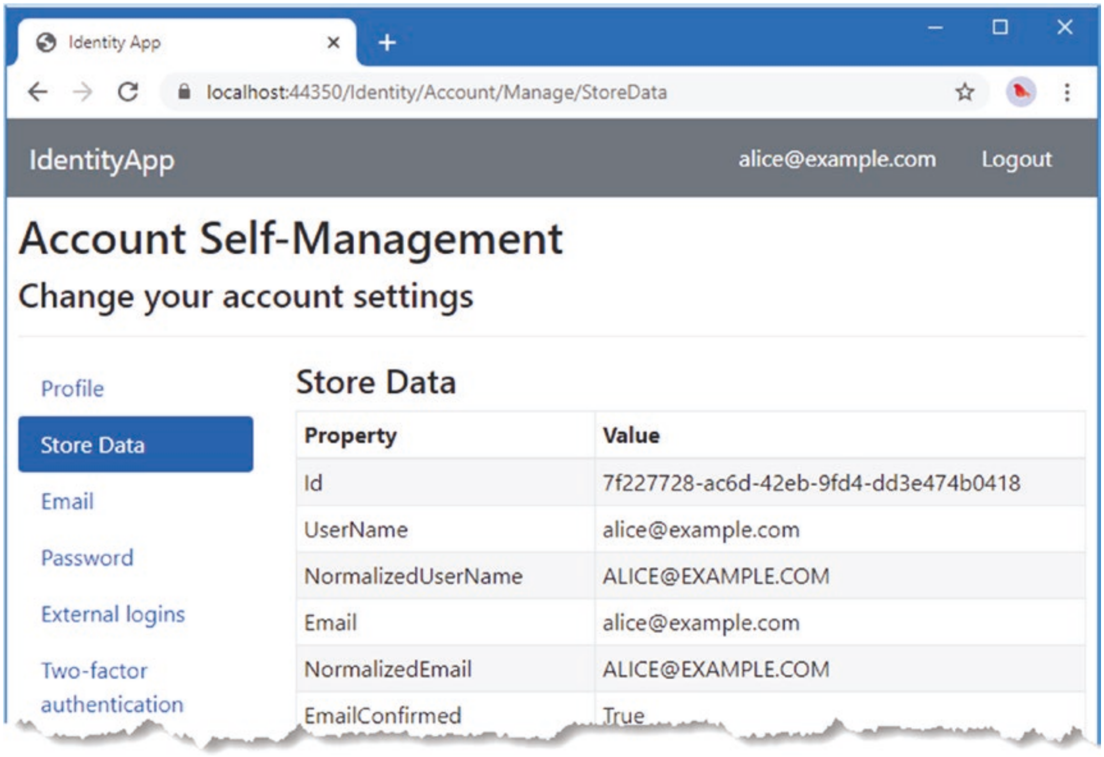
```
@page
@inject UserManager<IdentityUser> UserManager
@{
    ViewData["ActivePage"] = ManageNavPages.StoreData;
    IdentityUser user = await UserManager.GetUserAsync(User);
}

<h4>Store Data</h4>
```

```
<table class="table table-sm table-bordered table-striped">
    <thead>
        <tr><th>Property</th><th>Value</th></tr>
    </thead>
    <tbody>
        <tr>
            <td>Id</td><td>@user.Id</td>
        </tr>
        @foreach (var prop in typeof(IdentityUser).GetProperties()) {
            if (prop.Name != "Id") {
                <tr>
                    <td>@prop.Name</td>
                    <td class="text-truncate" style="max-width:250px">
                        @prop.GetValue(user)
                    </td>
                </tr>
            }
        }
    </tbody>
</table>
```

This Razor Page uses the `UserManager<IdentityUser>` class, which provides access to the data in the Identity user store. I describe this class in more detail in Chapter 7 to 12, but this statement in the code block at the start of the page gets an object that describes the signed-in user:

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

The user is represented by a `IdentityUser` object, and I use the standard .NET reflection features to generate an HTML table containing each property defined by the `IdentityUser` class. Restart ASP.NET Core, sign in as `alice@example.com` with the password `mysecret`, and click the Store Data link in the self-management section of the application, which produces the output shown in Figure 6-7. (You will see different values for some property values, which are generated dynamically.)



Account Self-Management

Change your account settings

Store Data

Property	Value
Id	7f227728-ac6d-42eb-9fd4-dd3e474b0418
UserName	alice@example.com
NormalizedUserName	ALICE@EXAMPLE.COM
Email	alice@example.com
NormalizedEmail	ALICE@EXAMPLE.COM
EmailConfirmed	True

Figure 6-7. *Adding an account management page*

Notice the first statement in the code block defined by the StoreData page:

```
...
ViewData["ActivePage"] = ManageNavPages.StoreData;
...
```

This statement sets the ActivePage view data property that the ManageNavPages.PageNavClass method uses to determine which page has been selected. This works with the preparations made at the start of this section to highlight the link for the StoreData page, as shown in the figure.

Overriding the Default Layout in an Account Management Page

The Razor layout that is scaffolded for the account management pages allows Razor Pages to override the default layout by setting a view data property named ParentLayout. Add a Razor Layout named _InfoLayout.cshtml to the Views/Shared folder and add the content shown in Listing 6-20.

Listing 6-20. The Contents of the _InfoLayout.cshtml File in the Views/Shared Folder

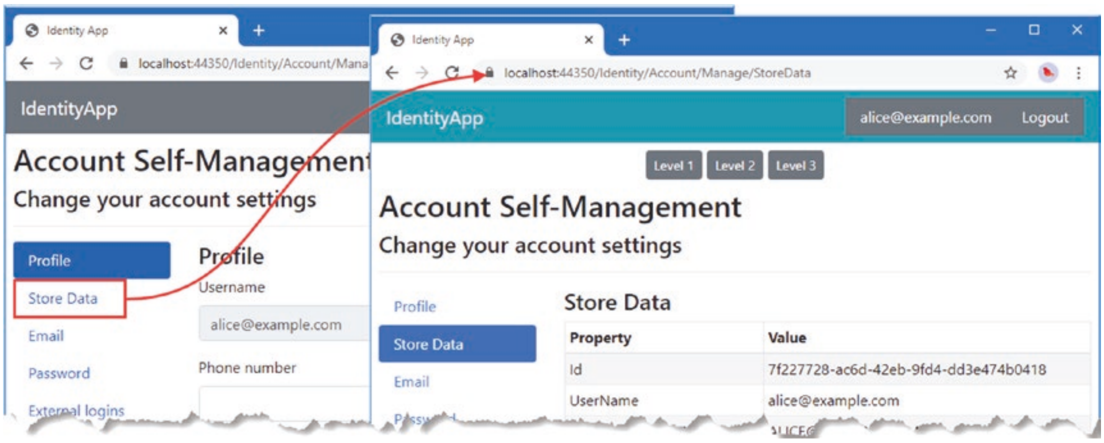
```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Identity App</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <nav class="navbar navbar-dark bg-info">
        <a class="navbar-brand text-white">IdentityApp</a>
        <div class="text-white"><partial name="_LoginPartial" /></div>
    </nav>
    <partial name="_NavigationPartial" />
    <div class="m-2">
        @RenderBody()
    </div>
    @RenderSection("Scripts", false)
</body>
</html>
```

To select the view, add the statement shown in Listing 6-21 to the code block defined by the `StoreData` page.

Listing 6-21. Selecting a Top-Level View in the StoreData.cshtml File in the Areas/Identity/Pages/Account/Manage Folder

```
@page
@inject UserManager<IdentityUser> UserManager
@{
    ViewData["ActivePage"] = ManageNavPages.StoreData;
    ViewData["ParentLayout"] = "_InfoLayout";
    IdentityUser user = await UserManager.GetUserAsync(User);
}

<h4>Store Data</h4>

<table class="table table-sm table-bordered table-striped">
    <thead>
        <tr><th>Property</th><th>Value</th></tr>
    </thead>
    <tbody>
        <tr>
            <td>Id</td><td>@user.Id</td>
        </tr>
        @foreach (var prop in typeof(IdentityUser).GetProperties()) {
            if (prop.Name != "Id") {
                <tr>
                    <td>@prop.Name</td>
                    <td class="text-truncate" style="max-width:250px">
                        @prop.GetValue(user)
                    </td>
                </tr>
            }
        }
    </tbody>
</table>
```

To see the effect, restart ASP.NET Core, navigate to the account management features, and click the Store Data link. The top-level view selected in Listing 6-21 is used, presenting the header in a different color, as shown in Figure 6-8.



Property	Value
Id	7f227728-ac6d-42eb-9fd4-dd3e474b0418
UserName	alice@example.com

Figure 6-8. *Overriding the top-level header*

Tidying Up the QR Code Support

In Chapter 4, I added support for displaying QR codes for authenticators by using JavaScript in the layout used for the Identity UI Razor Pages and by using the DOM API to inspect the HTML document and change it if it contained specific elements. This worked, but it requires the JavaScript code to be included in every page that uses the layout, even though all but one of them doesn't display a QR code. In this section, I am going to scaffold the Razor Page that displays the QR code and modify it to remove the placeholder content and include the JavaScript files that are required.

Doing the Scaffold File Shuffle

When you scaffold the account management files, the scaffolding tool tries to create the `ManageNavPages.cs` and `_ManageNav.cshtml` files, even though these files already exist. The result is an awkward file shuffle, which is most easily accomplished using two PowerShell command prompts.

Open the first command prompt, navigate to the `IdentityApp\Areas\Identity\Pages\Account\Manage` folder, and run the command shown in Listing 6-22. These commands move the files that the scaffolding tool will try to create.

Listing 6-22. Moving Files Before Scaffolding

```
Move-Item -Path ManageNavPages.cs -Destination ManageNavPages.cs.safe
Move-Item -Path _ManageNav.cshtml -Destination _ManageNav.cshtml.safe
```

Open the second command prompt and run the command shown in Listing 6-23 in the `IdentityApp` folder. This command scaffolds the Razor Page that displays the authenticator setup details. Pay close attention to the name of the Entity Framework Core database context class, which must match exactly to prevent the scaffolding tool from generating a new class.

Listing 6-23. Scaffolding the Identity UI Page for Authenticator Setup

```
dotnet aspnet-codegenerator identity --dbContext
Microsoft.AspNetCore.Identity.EntityFrameworkCore.IdentityDbContext --files
Account.Manage.EnableAuthenticator --no-build
```

The .NET command-line tools build the project before they run, which is usually helpful because it ensures the latest code changes are used. But this causes a problem in this situation because the `ManageNavPages` class that was moved in Listing 6-22 is referred to in other files, which breaks the build process. To avoid this issue, the command in Listing 6-23 includes the `--no-build` argument, which prevents the project from being built before the file is scaffolded.

Return to the first command prompt and run the commands shown in Listing 6-24 in the `IdentityApp\Areas\Identity\Pages\Account\Manage` folder to copy the modified `ManageNavPages.cs` and `_ManageNav.cshtml` back into place and overwrite the files created by the scaffolding process.

Listing 6-24. Moving Files After Scaffolding

```
Move-Item -Path ManageNavPages.cs.safe -Destination ManageNavPages.cs -Force
Move-Item -Path _ManageNav.cshtml.safe -Destination _ManageNav.cshtml -Force
```

The result of the file shuffle is that the `EnableAuthenticator` Razor Page has been scaffolded, and the changes made to the `ManageNavPages` class and the `_ManageNav` partial view have been preserved.

Modifying the Razor Page

Replace the contents of the `EnableAuthenticator.cshtml` file with those shown in Listing 6-25 to incorporate the QR code directly in the content produced by the page. This is a simplification of the HTML in the original page, with additional `script` elements to generate the QR code.

Listing 6-25. Replacing the Contents of the EnableAuthenticator.cshtml File in the Areas/Identity/Pages/Account/Manage Folder

```
@page
@model EnableAuthenticatorModel
@{
    ViewData["Title"] = "Configure authenticator app";
    ViewData["ActivePage"] = ManageNavPages.TwoFactorAuthentication;
}

<partial name="_StatusMessage" for="StatusMessage" />
<h4>@ViewData["Title"]</h4>
<div>
    <p>To use an authenticator app go through the following steps:</p>
    <ol class="list">
```

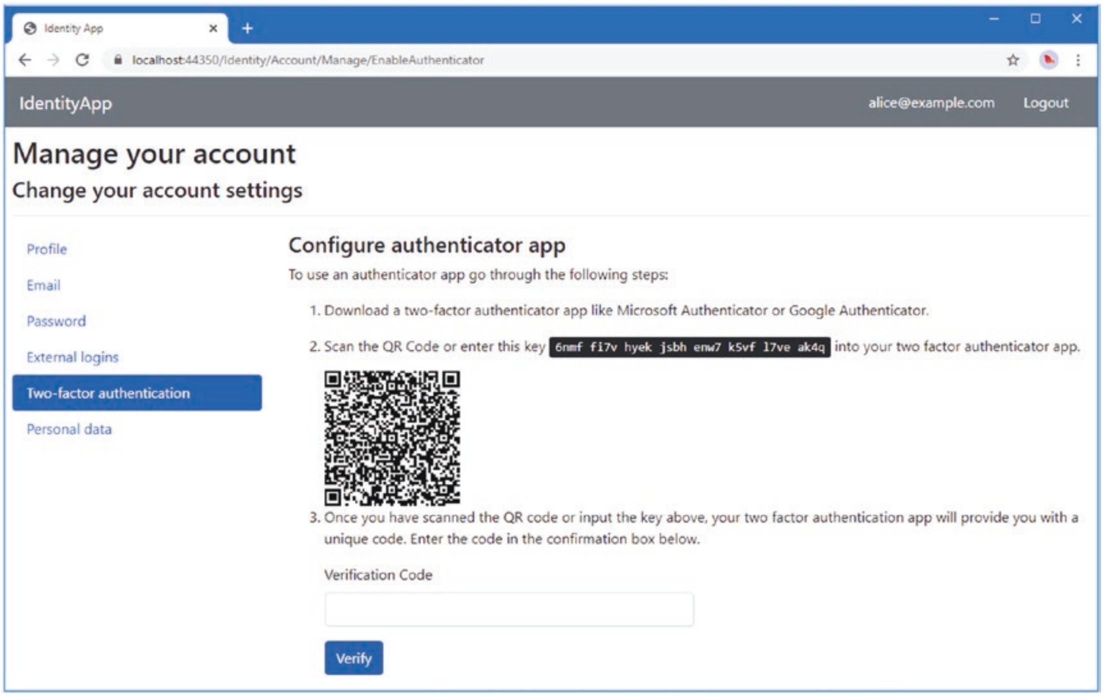
```
        <li>
            <p>
                Download a two-factor authenticator app like
                Microsoft Authenticator or Google Authenticator.
            </p>
        </li>
        <li>
            <p>Scan the QR Code or enter this key <kbd>@Model.SharedKey</kbd>
                 into your two factor authenticator app. Spaces
                 and casing do not matter.
            </p>
            <div id="qrCode"></div>
            <div id="qrCodeData" data-url="@Html.Raw(@Model.AuthenticatorUri)"></div>
        </li>
        <li>
            <p>
                Once you have scanned the QR code or input the key above,
                your two factor authentication app will provide you
                with a unique code. Enter the code in the confirmation box below.
            </p>
            <div class="row">
                <div class="col-md-6">
                    <form id="send-code" method="post">
                        <div class="form-group">
                            <label asp-for="Input.Code" class="control-label">
                                Verification Code
                            </label>
                            <input asp-for="Input.Code" class="form-control"
                                autocomplete="off" />
                            <span asp-validation-for="Input.Code"
                                class="text-danger"></span>
                        </div>
                        <button type="submit" class="btn btn-primary">Verify</button>
                        <div asp-validation-summary="ModelOnly" class="text-danger">
                        </div>
                    </form>
                </div>
            </div>
        </li>
    </ol>
</div>

@section Scripts {
    <partial name="_ValidationScriptsPartial" />
    <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>
}
```

Moving the JavaScript code into the Razor Page means that it can be removed from the shared layout, as shown in Listing 6-26.

Listing 6-26. Removing JavaScript Code from the _CustomIdentityLayout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Identity App</title>
    <link rel="stylesheet" href="/Identity/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="/Identity/css/site.css" />
    <script src="/Identity/lib/jquery/dist/jquery.js"></script>
    <script src="/Identity/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>
    <script src="/Identity/js/site.js" asp-append-version="true"></script>
    @*<script type="text/javascript" src="/lib/qrcode/qrcode.min.js"></script>*@
    <link href="/lib/font-awesome/css/all.min.css" rel="stylesheet" />
</head>
<body>
    <nav class="navbar navbar-dark bg-secondary">
        <a class="navbar-brand text-white">IdentityApp</a>
        <div class="text-white"><partial name="_LoginPartial" /></div>
    </nav>
    <div class="m-2">
        @RenderBody()
        @await RenderSectionAsync("Scripts", required: false)
    </div>
    @*<script type="text/javascript">
        var element = document.getElementById("qrCode");
        if (element !== null) {
            new QRCode(element, {
                text: document.getElementById("qrCodeData").getAttribute("data-url"),
                width: 150, height: 150
            });
            element.previousElementSibling?.remove();
        }
    </script>*@
</body>
</html>
```

Restart ASP.NET Core, request `https://localhost:44350/Identity/Account/Login`, and sign in to the application using the email address `alice@example.com` and the password `mysecret`. Request `https://localhost:44350/Identity/Account/Manage/TwoFactorAuthentication` and click Set up Authenticator App to see the modified content, which is shown in Figure 6-9.



Manage your account

Change your account settings

Configure authenticator app

To use an authenticator app go through the following steps:

1. Download a two-factor authenticator app like Microsoft Authenticator or Google Authenticator.
2. Scan the QR Code or enter this key `6nmf fi7v hyek jsbh enw7 k5vf l7ve ak4q` into your two factor authenticator app.
3. Once you have scanned the QR code or input the key above, your two factor authentication app will provide you with a unique code. Enter the code in the confirmation box below.

Verification Code

Figure 6-9. *Tidying up QR code generation*

Using Scaffolding to Disable Features

Scaffolding can also be used to disable features, which can be a useful way to benefit from the useful parts of the Identity UI package without offering workflows that don't suit your project. In this section, I demonstrate the process of disabling the password recovery feature, which may not be required in corporate environments where credentials are managed centrally, for example.

In Listing 6-27, I have changed the link for password recovery in the Login page.

Listing 6-27. Removing a Link in the Login.cshtml File in the Areas/Identity/Pages/Account Folder

```
...
<div class="form-group">
    @*<p>
        <a id="forgot-password" asp-page="./ForgotPassword">Forgot your password?</a>
    </p>*@
    <p>
        <a asp-page="./Register" asp-route-returnUrl="@Model.ReturnUrl">
             Register as a new user
         </a>
    </p>
    <p>
        <a id="resend-confirmation" asp-page="./ResendEmailConfirmation">
            Resend email confirmation
        </a>
    </p>
</div>
...
```

It isn't enough to just disable a link because the user can navigate directly to the URL for the password recovery pages. It isn't possible to delete pages from the Identity UI package, so the next best approach is to scaffold the files and redefine the page model class handler methods, which has the effect of safety disabling the page.

Use a PowerShell command prompt to run the command shown in Listing 6-28 in the IdentityApp folder. As with earlier commands, you must take care to specify the name of the database context class correctly to prevent a new class from being generated.

Listing 6-28. Scaffolding the Password Recovery Pages

```
dotnet aspnet-codegenerator identity --dbContext
Microsoft.AspNetCore.Identity.EntityFrameworkCore.IdentityDbContext --files
Account.ForgotPassword
```

Once you have scaffolded the page, modify the page model class to remove the OnPostAsync method and add GET and POST handler methods that redirect the browser to the Login page, as shown in Listing 6-29.

Listing 6-29. Redefining Methods in the ForgotPassword.cshtml.cs File in the Areas/Identity/Pages/Account Folder

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Text.Encodings.Web;
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.UI.Services;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.WebUtilities;

namespace IdentityApp.Areas.Identity.Pages.Account {
    [AllowAnonymous]
    public class ForgotPasswordModel : PageModel {
        private readonly UserManager<IdentityUser> _userManager;
        private readonly IEmailSender _emailSender;
        public ForgotPasswordModel(UserManager<IdentityUser> userManager,
                IEmailSender emailSender) {
            _userManager = userManager;
            _emailSender = emailSender;
        }

        [BindProperty]
        public InputModel Input { get; set; }

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

        //public async Task<IActionResult> OnPostAsync() {
        //    if (ModelState.IsValid) {
        //        var user = await _userManager.FindByEmailAsync(Input.Email);
        //        if (user == null || !(await _userManager.
        //                IsEmailConfirmedAsync(user))) {
        //            return RedirectToPage("./ForgotPasswordConfirmation");
        //        }
        //
        //        var code = await _userManager.
        //            GeneratePasswordResetTokenAsync(user);
        //        code = WebEncoders.Base64UrlEncode(Encoding.UTF8.GetBytes(code));
        //        var callbackUrl = Url.Page(
        //            "/Account/ResetPassword",
        //            pageHandler: null,
        //            values: new { area = "Identity", code },
        //            protocol: Request.Scheme);
        //        await _emailSender.SendEmailAsync(
        //            Input.Email,
        //            "Reset Password",
        //            $"Please reset your password by <a href='{HtmlEncoder.
        //                Default.Encode(callbackUrl)}'>clicking here</a>.");
        //        return RedirectToPage("./ForgotPasswordConfirmation");
        //    }
        //    return Page();
        //}

        public IActionResult OnGet() => RedirectToPage("./Login");
        public IActionResult OnPost() => RedirectToPage("./Login");
    }
}
```

The GET handler prevents the page from rendering content. I like to add a POST handler as well so that attempts to submit a POST request also result in a redirection. Restart ASP.NET Core and request `https://localhost:44350/Identity/Account/Login`. You will see no link for password recovery, as shown in Figure 6-10. If you attempt to navigate to `https://localhost:44350/Identity/Account/ForgotPassword`, you will be redirected back to the `Login` page.

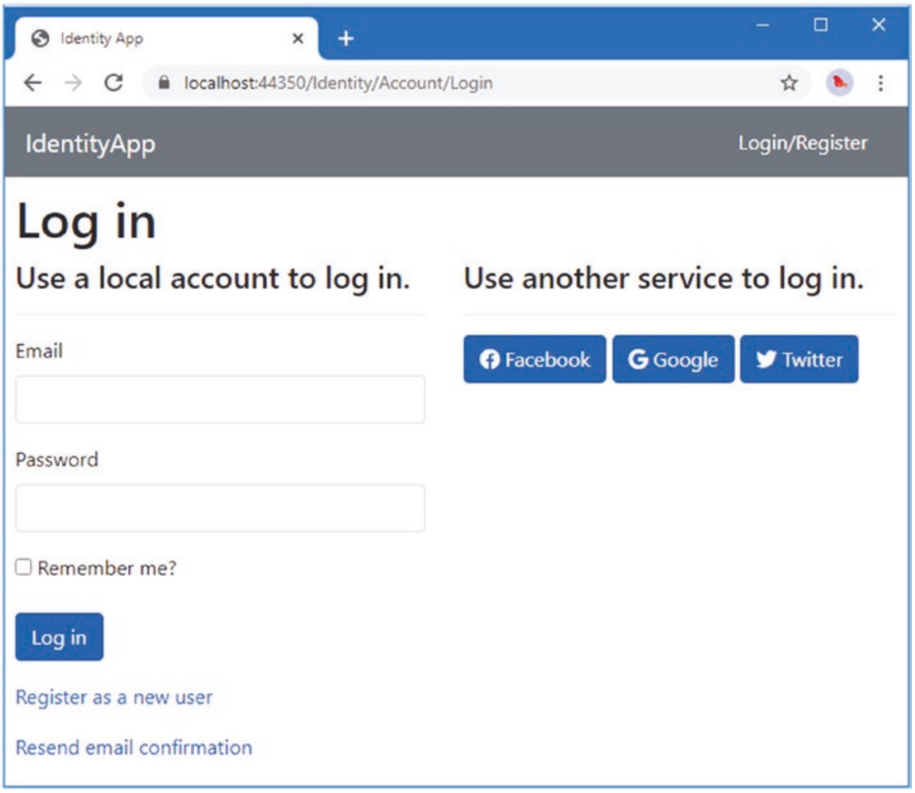

Identity App
localhost:44350/Identity/Account/Login
IdentityApp
Login/Register
Log in
Use a local account to log in.
Use another service to log in.
Email
Password
Remember me?
Log in
Facebook
Google
Twitter
Register as a new user
Resend email confirmation

Figure 6-10. *Disabling a feature*

Summary

In this chapter, I explained the different ways in which the Identity UI package can be adapted to suit the needs of a project. I demonstrated how the scaffolding process can be used to bring Razor Pages into the application where they take precedence over the default pages in the Identity UI package. This feature can be used to modify the features that the Identity UI package provides, create new features, or remove features entirely. Although the adaptations provide flexibility, there are limits to the customizations that can be made. In the next chapter, I start to describe the API that Identity provides for creating custom workflows that can replace the Identity UI package.