CHAPTER 4

Using the Identity UI Package

Microsoft provides a build user interface for Identity, known as *Identity UI*, which makes it possible to get up and running quickly. In this chapter, I add Identity to the example project created in Chapter 3 and explain the features that the Identity UI package provides. In Chapter 6, I explain how to adapt those features to suit different project types.

Even with the adaptations that I describe in Chapter 6, the Identity UI package is only suitable for applications with a specific set of characteristics. I describe the restrictions those characteristics lead to and, starting in Chapter 7, explain how to create entirely custom workflows as an alternative to using the Identity UI package.

But, as this chapter will demonstrate, if your application does meet the requirements, you can get a lot of benefit from the Identity UI package with little effort. Table 4-1 puts the Identity UI package in context.

Table 4-1. *Putting the Identity UI Package in Context*

Question	Answer
What is it?	The Identity UI package is a set of Razor Pages and supporting classes provided by Microsoft to jump-start the use of ASP.NET Core Identity in ASP.NET Core projects.
Why is it useful?	The Identity UI package provides all the workflows required for basic user management, including creating accounts and signing in with passwords, authenticators, and third-party services.
How is it used?	The Identity UI package is added to projects as a NuGet package and enabled with the `AddDefaultIdentity` extension method.
Are there any pitfalls or limitations?	The approach that Identity UI takes doesn't suit all projects. This can be remedied either by adapting the features it provides or by working directly with the Identity API to create custom alternatives.
Are there any alternatives?	Identity provides an API that can be used to create custom alternatives to the Identity UI package, which I describe in Chapters 7 to 11.

Table 4-2 summarizes the chapter.

Table 4-2. *Chapter Summary*

Problem	Solution	Listing
Add Identity and the Identity UI package to a project	Add the NuGet packages to the project and configure them using the `AddDefaultIdentity` method in the `Startup` class. Create a database migration and use it to prepare a database for storing user data.	1–7
Present the user with the registration or sign-in links	Create a shared partial view named `_LoginPartial.cshtml`.	8, 9
Create a consistent layout for the application and the Identity UI package	Define a Razor Layout and refer to it in a Razor View Start created in the `Areas/Identity/Pages` folder.	10–12
Add support for confirmations	Create an implementation of the `IEmailSender` interface and register it as a service in the `Startup` class.	13, 14
Display QR codes for configuring authenticator applications	Add the `qrcodejs` JavaScript package to the project and create a `script` element that applies it to the URL produced by the Identity UI package.	15, 16

Preparing for This Chapter

This chapter uses the `IdentityApp` project created in Chapter 3. No changes are required to prepare for this chapter. Open a PowerShell command prompt, navigate to the `IdentityApp` folder, and run the commands shown in Listing 4-1 to delete and then re-create the database the application uses.

■ **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 4-1. Resetting the Application Database

```
dotnet ef database drop --force
dotnet ef database update
```

Use the PowerShell prompt to run the command shown in Listing 4-2 in the `IdentityApp` folder to start the application.

Listing 4-2. 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 4-1. Clicking the Level 2 or Level 3 button produces an exception because ASP.NET Core hasn't been provided with the services it needs to authenticate requests and enforce the restrictions applied with the `Authorize` attribute.

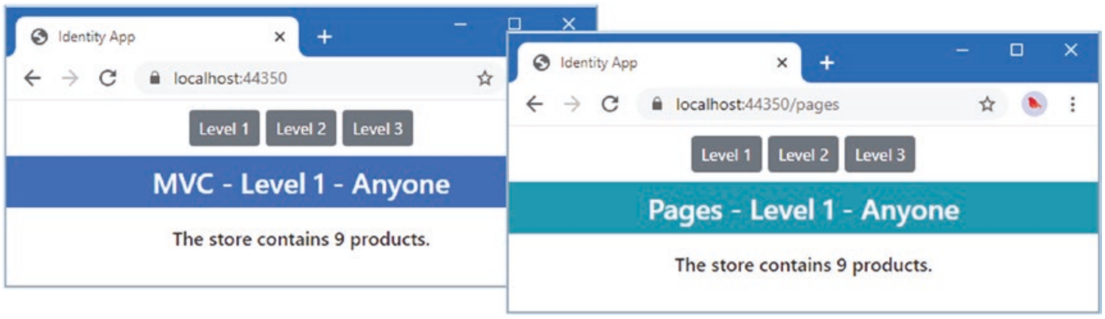


Figure 4-1. *Running the example application*

Adding ASP.NET Core Identity to the Project

Use a PowerShell command prompt to run the commands shown in Listing 4-3 in the `IdentityApp` folder to add the ASP.NET Core Identity packages to the project.

Listing 4-3. Adding the ASP.NET Core Identity Packages

```
dotnet add package Microsoft.Extensions.Identity.Core --version 5.0.0
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore --version 5.0.0
```

The first package contains the core Identity features. The second package contains the features required to store data in a database using Entity Framework Core.

Adding the Identity UI Package to the Project

Use PowerShell to run the command shown in Listing 4-4 in the `IdentityApp` folder to install the Identity UI package.

Listing 4-4. Installing the Identity UI Package

```
dotnet add package Microsoft.AspNetCore.Identity.UI --version 5.0.0
```

Defining the Database Connection String

The easiest way to store Identity data is in a database, and Microsoft provides built-in support for doing this with Entity Framework Core. Although you can use a single database for the application's domain data and the Identity data, I recommend you keep everything separate so that you can manage the schemas independently.

To define the connection string that Entity Framework Core will use for the Identity database, add the configuration item shown in Listing 4-5 to the `appsettings.json` file, which specifies a LocalDB database named `IdentityAppUserData` using a connection string named `IdentityConnection`. (Connection strings should be on a single unbroken line but are too long to show this way on the printed page.)

Listing 4-5. Adding a Connection String in the appsettings.json File in the IdentityApp Folder

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "AppDataConnection": "Server=(localdb)\\MSSQLLocalDB;Database=IdentityAppData;
    MultipleActiveResultSets=true",
    "IdentityConnection": "Server=(localdb)\\MSSQLLocalDB;Database=IdentityAppUserData;
    MultipleActiveResultSets=true"
  }
}
```

Configuring the Application

The next step is to configure the application to set up the database that will be used to store user data and to configure ASP.NET Core Identity. Add the statements shown in Listing 4-6 to the `Startup` class.

Listing 4-6. Configuring the Application in the Startup.cs File in the IdentityApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using IdentityApp.Models;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
```

```
namespace IdentityApp {

    public class Startup {

        public Startup(IConfiguration config) => Configuration = config;

        private IConfiguration Configuration { get; set; }

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

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

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

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();
            app.UseRouting();

            app.UseAuthentication();
            app.UseAuthorization();

            app.UseEndpoints(endpoints => {
                endpoints.MapDefaultControllerRoute();
                endpoints.MapRazorPages();
            });
        }
    }
}
```

The AddDbContext method is used to set up an Entity Framework Core database context for Identity. The database context class is IdentityDbContext, which is included in the Identity packages and includes details of the schema that will be used to store identity data. You can create a custom database context class if you prefer—and this is the approach taken by the project template I used in Chapter 2—but there is no good reason to do so, and it is just another class to add to the project. (And, as you will see by the end of this part of the book, using Identity can add a lot of files to a project.)

Because the IdentityDbContext class is defined in a different assembly, I have to tell Entity Framework Core to create database migrations in the IdentityApp project, like this:

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

The other new statement in Listing 4-6 sets up ASP.NET Core Identity. The first part calls the AddDefaultIdentity method, like this:

```
...
services.AddDefaultIdentity<IdentityUser>()
    .AddEntityFrameworkStores<IdentityDbContext>();
...
```

The reason that ASP.NET Core threw exceptions for requests to restricted URLs in Chapter 3 was that no services had been registered to authentication requests. The AddDefaultIdentity method sets up those services using sensible default values. The generic type argument specifies the class Identity will use to represent users. The default class is IdentityUser, which is included in the Identity package.

IdentityUser is known as the *user class* and is used by Identity to represent users. IdentityUser is the default user class provided by Microsoft. In Part 2, I create a custom user class, but IdentityUser is suitable for almost every project. The second part of this statement sets up the Identity datastore:

```
...
services.AddDefaultIdentity<IdentityUser>()
    .AddEntityFrameworkStores<IdentityDbContext>();
...
```

The AddEntityFrameworkStores method sets up data storage using Entity Framework Core, and the generic type argument specifies the database context that will be used. Identity uses two kinds of datastore: the *user store* and the *role store*. The user store is the heart of Identity and is used to store all of the user data, including email addresses, passwords, and so on. Confusingly, membership of roles is kept in the user store. The role store contains additional information about roles that are used only in complex applications. I explain every aspect of the user store and role store in Part 2, but you don't typically need to get into the detail of either store when using Identity, other than to know that they exist and to check they support all of the features you require, which I demonstrate in Chapter 7.

Creating the Database

Entity Framework Core requires a database migration, which will be used to create the database for Identity data. Run the commands shown in Listing 4-7 to create and then apply a migration for Identity. These commands require the `--context` argument because there are two database context classes set up in the `Startup` class: one for Identity and one for the application data.

Listing 4-7. Creating and Applying a Migration for ASP.NET Core Identity

```
dotnet ef migrations add IdentityInitial --context IdentityDbContext
dotnet ef database drop --force --context IdentityDbContext
dotnet ef database update --context IdentityDbContext
```

The result of these commands is that a new migration will be added to the `IdentityApp` folder, which is then used to create a new database. The `database drop` command ensures that any existing database named `IdentityAppUserData` is deleted.

Preparing the Login Partial View

The Identity UI package requires a partial view named `_LoginPartial`, which is displayed at the top of every page. This approach means the same partial view can be used by the rest of the application, presenting the user with a consistent user interface. Add a Razor View named `_LoginPartial.cshtml` to the `Views/Shared` folder with the content shown in Listing 4-8.

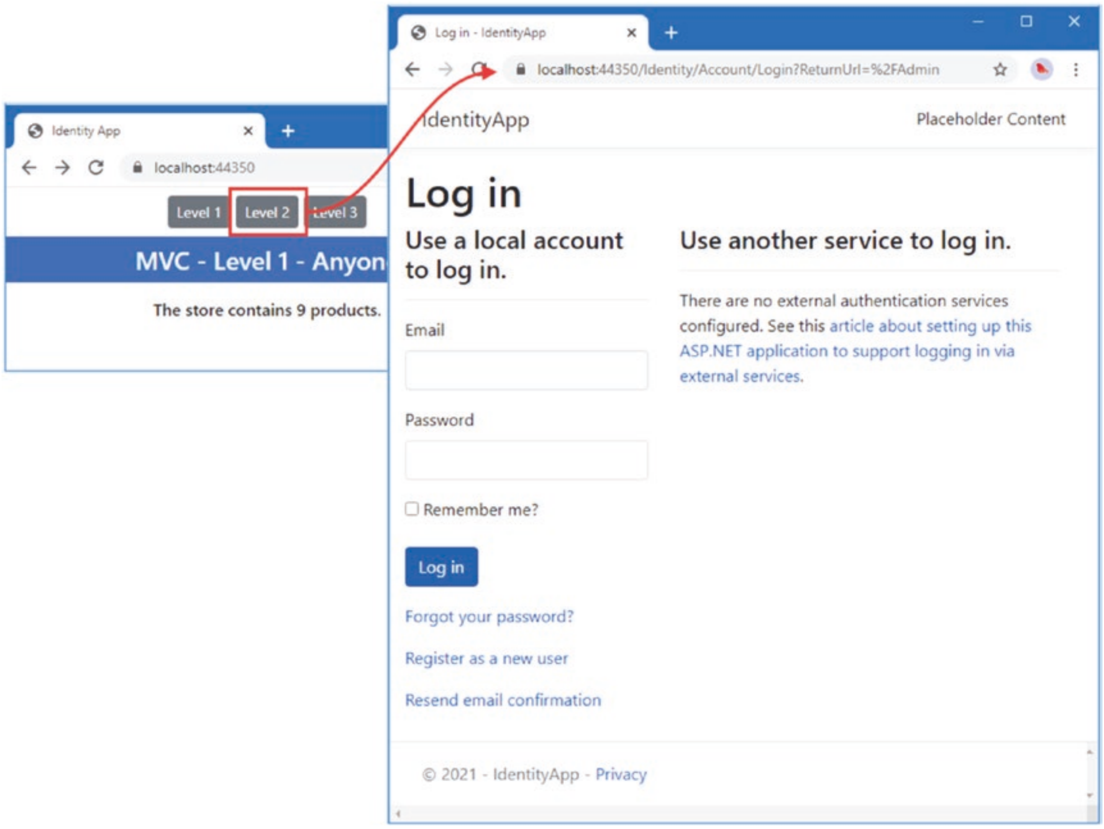
Listing 4-8. The Contents of the _LoginPartial.cshtml File in the Views/Shared Folder

```
<div>Placeholder Content</div>
```

I return to this listing later to make it more useful. For now, it is enough to know that you can't use the Identity UI package without creating this partial view.

Testing the Application with Identity

Restart ASP.NET Core and request `https://localhost:44350` or `https://localhost:44350/pages`, which will present the content that is always accessible. Click the Level 2 button, and you will see the content shown in Figure 4-2.


Identity App

localhost:44350

Level 1 Level 2 Level 3

MVC - Level 1 - Anyon

The store contains 9 products.

Log in - IdentityApp

localhost:44350/Identity/Account/Login?ReturnUrl=%2FAdmin

IdentityApp Placeholder Content

Log in

Use a local account to log in.

Email

Password

Remember me?

Log in

Forgot your password?

Register as a new user

Resend email confirmation

Use another service to log in.

There are no external authentication services configured. See this article about setting up this ASP.NET application to support logging in via external services.

© 2021 - IdentityApp - Privacy

Figure 4-2. *The Identity challenge response*

Clicking the Level 2 button sends a request to ASP.NET Core that requires authorization. The request doesn't provide ASP.NET Core with any information about the user, which triggers the response shown in the figure, called the *challenge response*. The term *challenge* is used because the user is challenged to identify themselves in a modern-day equivalent of asking "who goes there?"

The challenge response is a redirection to the `Identity/Account/Login` URL. The Identity UI package contains a set of Razor Pages in a separate area, named `Identity`, which keeps them isolated from the rest of the application.

SIGNING IN OR LOGGING IN?

I have tried to be consistent throughout this book and use the term *sign in* to refer to the process of a user identifying themselves to an application. You will, however, see references to *logging in*, including in the default content provided by the Identity UI package, as shown in Figure 4-2.

Sign in and *log in* mean the same thing. Some studies show users find the terms *sign up* and *sign in* easier to understand, but I am skeptical and suspect there is little difference in practice. I have chosen *sign in* for consistency and because that is the term that has been adopted by the Identity API, even though it isn't used by the Identity UI package, which uses the Identity API behind the scenes.

It doesn't matter, and you should use whichever terms suit your projects and are likely to be understood by your target users.

Creating a New User Account

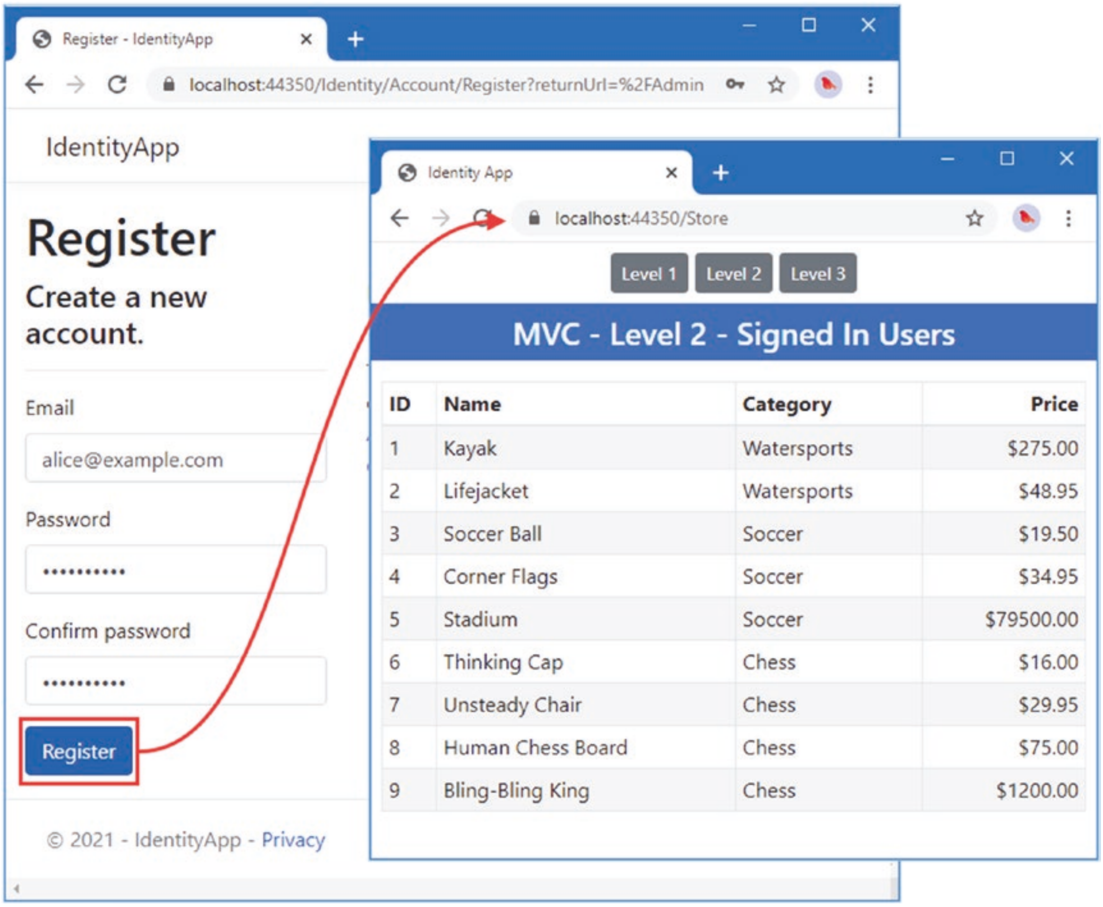
In a self-service application, it is the responsibility of the user to create an account. Identity supports using third-party services, such as Google and Facebook, to create accounts, which I demonstrate in Chapter 11. By default, no external services are configured, and users can only create local accounts, which means that users must authenticate themselves with a password, which is compared to data stored in the Identity database.

Click the Register link at the top right of the page, and you will be presented with a registration page. Use the form fields to enter the data values shown in Table 4-3. (Identity has a password policy that requires a mix of different types, which I describe in Chapter 5.)

Table 4-3. *Account Registration Details*

Field	Value
`Email`	`alice@example.com`
`Password`	`MySecret1$`
`ConfirmPassword`	`MySecret1$`

Click the Register button, and a new user account will be created. You will be signed into the application and redirected to the URL you requested that triggered the challenge response, as shown in Figure 4-3.

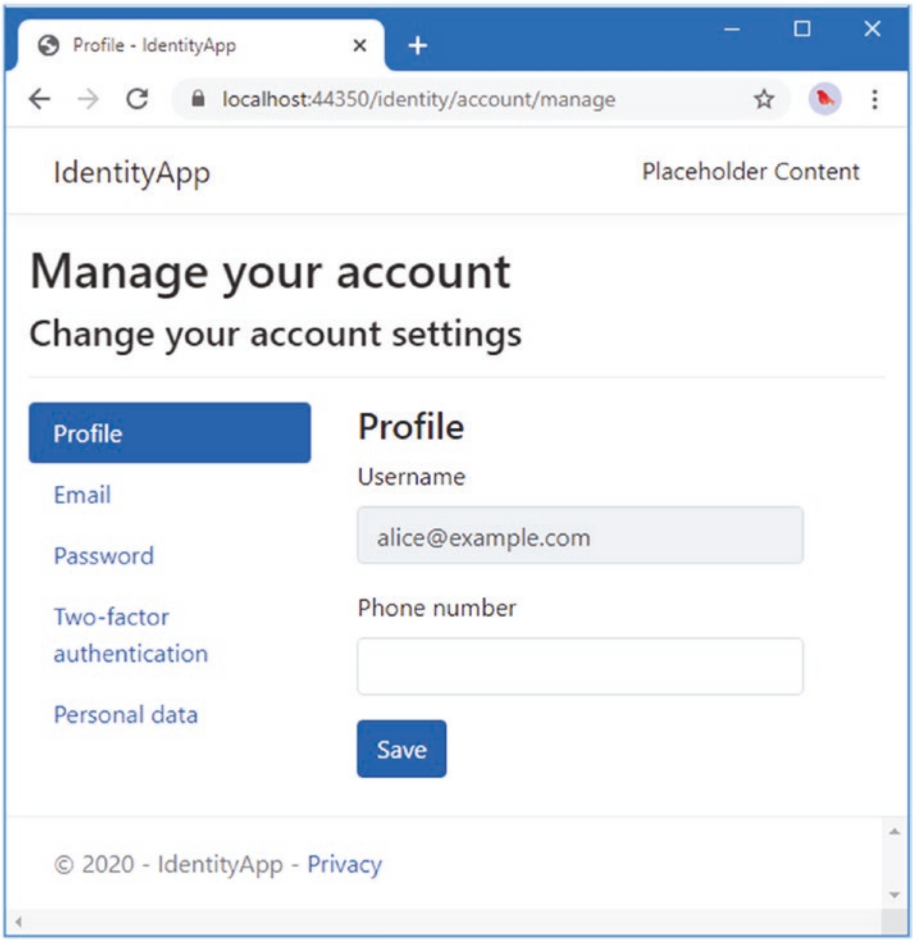


ID	Name	Category	Price
1	Kayak	Watersports	$275.00
2	Lifejacket	Watersports	$48.95
3	Soccer Ball	Soccer	$19.50
4	Corner Flags	Soccer	$34.95
5	Stadium	Soccer	$79500.00
6	Thinking Cap	Chess	$16.00
7	Unsteady Chair	Chess	$29.95
8	Human Chess Board	Chess	$75.00
9	Bling-Bling King	Chess	$1200.00

Figure 4-3. *Creating a new account*

Managing an Account

Request `https://localhost:44350/identity/account/manage`, and you will be presented with the account self-management features provided by the Identity UI package, as shown in Figure 4-4.


Profile - IdentityApp
localhost:44350/identity/account/manage
IdentityApp
Placeholder Content
Profile
Email
Password
Two-factor authentication
Personal data
Save
© 2020 - IdentityApp - Privacy

Manage your account

Change your account settings

Profile

Username

alice@example.com

Phone number

Figure 4-4. *The Identity UI account self-management features*

Additional work is required to build out the example project, as explained in the next section, so not all of the features work fully. But the basics are in place, and users can update their details, change their password, and increase the security of their account by adding a second factor. I revisit each of these features later in the chapter and explain how they work in detail.

Completing the Application Setup

The basic configuration is complete, but several features require additional work before they function correctly. In the sections that follow, I will go through the process of completing the setup process for the self-service UI.

Displaying Login Information

Earlier in the chapter, I created a partial view named `_LoginPartial.cshtml`, which the Identity UI requires. The purpose of this partial view is to present the user with links to Identity UI pages, making it easy to navigate to the sign-in page or the account self-management management features. Replace the placeholder content in the partial view with the content in Listing 4-9.

Listing 4-9. Replacing the Contents of the _LoginPartial.cshtml File in the Pages/Shared Folder

```
<nav class="nav">
    @if (User.Identity.IsAuthenticated) {
        <a asp-area="Identity" asp-page="/Account/Manage/Index"
            class="nav-link bg-secondary text-white">
                @User.Identity.Name
        </a>
        <a asp-area="Identity" asp-page="/Account/Logout"
            class="nav-link bg-secondary text-white">
                Logout
        </a>
    } else {
        <a asp-area="Identity" asp-page="/Account/Login"
                class="nav-link bg-secondary text-white">
            Login/Register
        </a>
    }
</nav>
```

The `@if` expression in the partial view determines whether there is a signed-in user by reading the `User.Identity.IsAuthenticated` property. ASP.NET Core represents users with the `ClaimsPrincipal` class and a `ClaimsPrincipal` object for the current user is available through the `User` property defined by the `Controller` and `RazorPageBase` classes, which means the same features are available for the MVC Framework and Razor Pages. (The `nav` elements and the classes to which the elements are assigned apply styles from the Bootstrap CSS framework and are not related to Identity.)

If there is a signed-in user, the partial view displays two anchor (`a`) elements, which will navigate to the Identity UI pages for managing an account or logging out. The anchor elements are configured using tag helpers and specify the `Identity` area that contains the Identity UI Razor Pages.

```
...
 <a asp-area="Identity" asp-page="/Account/Manage/Index"
        class="nav-link bg-secondary text-white">
    @User.Identity.Name
</a>
...
```

The `asp-area` and `asp-page` tags work together to create a link for the Index Razor Page in the `Account/Manage` folder of the Identity UI package.

One drawback of Identity UI is you need to know the names of the pages that provide key features, such as signing in, signing out, and managing an account. I detail the URLs for each feature in the "Using the Identity UI Workflows" section, later in the chapter.

Restart ASP.NET Core, and request `https://localhost:44350/identity/account/login`. If you are still logged in from the previous section, you will see a link to manage the account and a logout button. If you have logged out—or your session has expired—you will see a prompt to sign in or register, as shown in Figure 4-5.

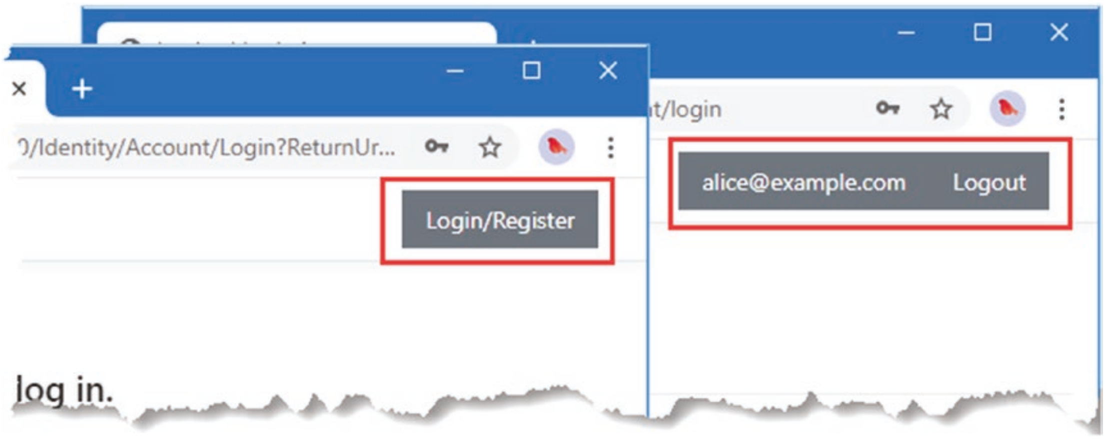


***Figure 4-5.** Displaying login information*

Creating a Consistent Layout

The Identity UI package is a collection of Razor Pages set up in a separate ASP.NET Core area. This means a project can override individual files from the Identity UI package by creating Razor Pages with the same names. I show you how this can be used to adapt the Identity UI functionality in later examples, but the simplest use of this feature is to provide a consistent layout that will be used for both the application's content and the Identity UI package.

Add a Razor Layout named `_CustomIdentityLayout.cshtml` to the `Pages/Shared` folder with the contents shown in Listing 4-10.

Listing 4-10. The Contents of the _CustomIdentityLayout.cshtml File in the Pages/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>
</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>
</body>
</html>
```

This layout contains includes content rendered by the `_LoginPartial` view, as part of a larger navigation bar.

■ **Tip** I looked at the layout in the Identity UI package to determine the `link` and `script` elements required in Listing 4-10. You can see the contents of the package at `https://github.com/dotnet/aspnetcore/tree/master/src/Identity/UI/src/Areas/Identity/Pages/V4`.

To use the new view, create the `Areas/Identity/Pages` folder and add to it a Razor View Start file named `_ViewStart.cshtml` with the content shown in Listing 4-11. The location of this file overrides the Razor View Start file in the Identity UI package.

Listing 4-11. The Contents of the _ViewStart.cshtml File in the Areas/Identity/Pages Folder

```
@{
    Layout = "_CustomIdentityLayout";
}
```

The final step is to update the layout used by the rest of the application to display the same header, as shown in Listing 4-12.

Listing 4-12. Adding a Header in the _Layout.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-secondary">
        <a class="navbar-brand text-white">IdentityApp</a>
        <div class="text-white"><partial name="_LoginPartial" /></div>
    </nav>
    <partial name="_NavigationPartial" />
    @RenderBody()
</body>
</html>
```

Restart ASP.NET Core, and request `https://localhost:44350`. You will see the new header at the top of the page, which is also shown if you click one of the links presented by the login partial view, as shown in Figure 4-6.

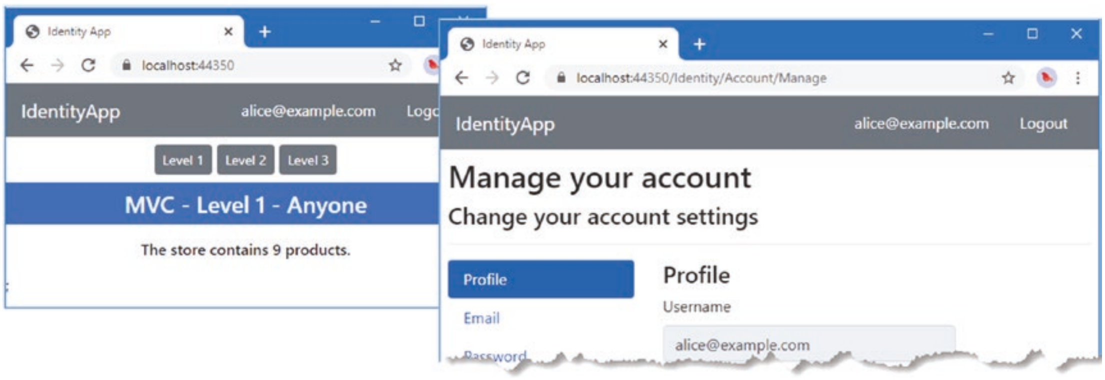


Figure 4-6. *Creating a consistent layout*

Configuring Confirmations

A confirmation is an email message that asks the user to click a link to confirm an action, such as creating an account or changing a password. The Identity support for confirmations is described in detail in Part 2, but the Identity UI package provides a simplified confirmation process that requires an implementation of the `IEmailSender` interface, which is defined in the `Microsoft.AspNetCore.Identity.UI.Services` namespace. The `IEmailSender` interface defines one method, which is described in Table 4-4.

Table 4-4. *The Method Defined by the IEmailSender Interface*

Name	Description
`SendEmailAsync(emailAddress, subject, htmlMessage)`	This method sends an email using the specified address, subject, and HTML message body.

The Identity UI package includes an implementation of the interface whose `SendEmailAsync` method does nothing. I am going to create a dummy email service in this chapter because the process of setting up and integrating with a real service is beyond the scope of the book. In Chapter 17, where I describe the confirmation process in depth, I provide suggestions for commercial messaging platforms, but, in this book, I demonstrate the Identity support for confirmations by writing messages to the .NET console. Create the `IdentityApp/Services` folder and add to it a class file named `ConsoleEmailSender.cs` with the code shown in Listing 4-13.

Listing 4-13. The Contents of the ConsoleEmailSender.cs File in the Services Folder

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

```
namespace IdentityApp.Services {
    public class ConsoleEmailSender : IEmailSender {

        public Task SendEmailAsync(string emailAddress,
                string subject, string htmlMessage) {
            System.Console.WriteLine("---New Email----");
            System.Console.WriteLine($"To: {emailAddress}");
            System.Console.WriteLine($"Subject: {subject}");
            System.Console.WriteLine(HttpUtility.HtmlDecode(htmlMessage));
            System.Console.WriteLine("-------");
            return Task.CompletedTask;
        }
    }
}
```

In Listing 4-14, I have registered the `ConsoleEmailSender` class as the implementation of the `IEmailSender` that will be used for dependency injection.

Listing 4-14. Registering the Email Sender Service in the Startup.cs File in the IdentityApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using IdentityApp.Models;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.AspNetCore.Identity.UI.Services;
using IdentityApp.Services;

namespace IdentityApp {

    public class Startup {

        public Startup(IConfiguration config) => Configuration = config;

        private IConfiguration Configuration { get; set; }

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

            services.AddHttpsRedirection(opts => {
                opts.HttpsPort = 44350;
            });
            services.AddDbContext<IdentityDbContext>(opts => {
                opts.UseSqlServer(
                    Configuration["ConnectionStrings:IdentityConnection"],
                    opts => opts.MigrationsAssembly("IdentityApp")
                );
            });

            services.AddScoped<IEmailSender, ConsoleEmailSender>();

            services.AddDefaultIdentity<IdentityUser>()
                .AddEntityFrameworkStores<IdentityDbContext>();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();
            app.UseRouting();

            app.UseAuthentication();
            app.UseAuthorization();

            app.UseEndpoints(endpoints => {
                endpoints.MapDefaultControllerRoute();
                endpoints.MapRazorPages();
            });
        }
    }
}
```

Notice that I have registered the email service before the call to the AddDefaultIdentity method so that my custom service takes precedence over the placeholder implementation in the Identity UI package.

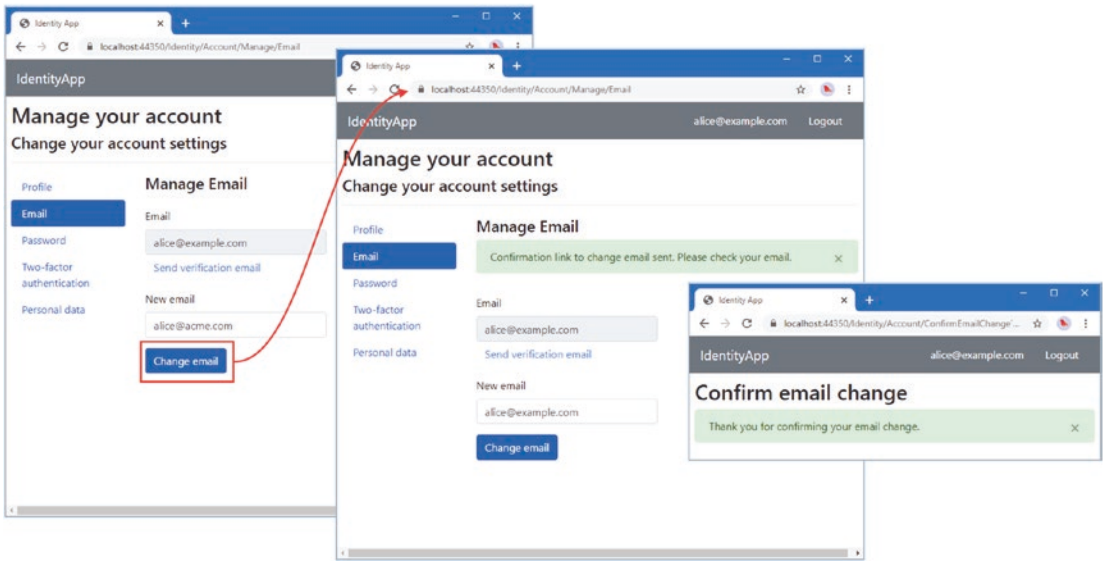
To test the confirmation process, restart ASP.NET Core. If you are not already signed to the application from earlier examples, request https://localhost:44350/identity/account/login and sign into the application as alice@example.com, with the password MySecret1$. Click the alice@example.com email address in the header to request the self-management features and click the Email link.

Enter alice@acme.com into the New Email field and click the Change Email button. Examine the console output from ASP.NET Core, and you will see the email message that the Identity UI has sent to the user, which will look like this:

```
---New Email----
To: alice@acme.com
Subject: Confirm your email
Please confirm your account by <a href='https://localhost:44350/Identity/
Account/ConfirmEmailChange?userId=cb55600e-9e03-43b8-a7b4-4e347c9d3943&email=alice@acme.com&
code=Q2ZESjhBMVB3bFFBQ3g'>clicking here</a>.
-------
```

The message contains a link for the user to click to confirm their new address. I have shortened the URL, but the query string contains a long security token used to validate the request securely. I explain how these tokens are created and validated in detail in Part 2, but you don't need to know how they work to use them.

Use your browser to navigate to the URL specified in the email message, and you will receive a response confirming the change of email address, as shown in Figure 4-7.


Identity App
localhost:44350/Identity/Account/Manage/Email
IdentityApp
Profile
Email
Password
Two-factor authentication
Personal data
Send verification email
Change email
alice@example.com
Logout
localhost:44350/Identity/Account/ConfirmEmailChange'...

Manage your account

Change your account settings

Manage Email

Email

alice@example.com

New email

alice@acme.com

Confirmation link to change email sent. Please check your email.

Confirm email change

Thank you for confirming your email change.

Figure 4-7. Changing an email address

Displaying QR Codes

Identity provides support for two-factor authentication, where the user has to present additional credentials to sign into the application. The Identity UI package supports a specific type of additional credential, which is a code generated by an authenticator application. An authenticator application is set up once and then generates authentication codes that can be validated by the application. To complete the setup for authenticators with Identity UI, a third-party JavaScript library named `qrcodejs` is required to generate QR codes that can be scanned by mobile devices to simplify the initial setup process.

Use a PowerShell command prompt and run the command shown in Listing 4-15 in the `IdentityApp` folder to install the package that Microsoft recommends for generating QR codes.

Listing 4-15. Adding a JavaScript Package

```
libman install qrcodejs@1.0.0 -d wwwroot/lib/qrcode
```

I explain how to customize the Identity UI package in Chapter 6, so for this chapter I am going to take a shortcut to display QR codes without needing to use the customization features. Add the `script` elements shown in Listing 4-16 to the `_CustomIdentityLayout.cshtml` file in the `Views/Shared` folder.

Listing 4-16. Adding Script Elements 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>
</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>
```

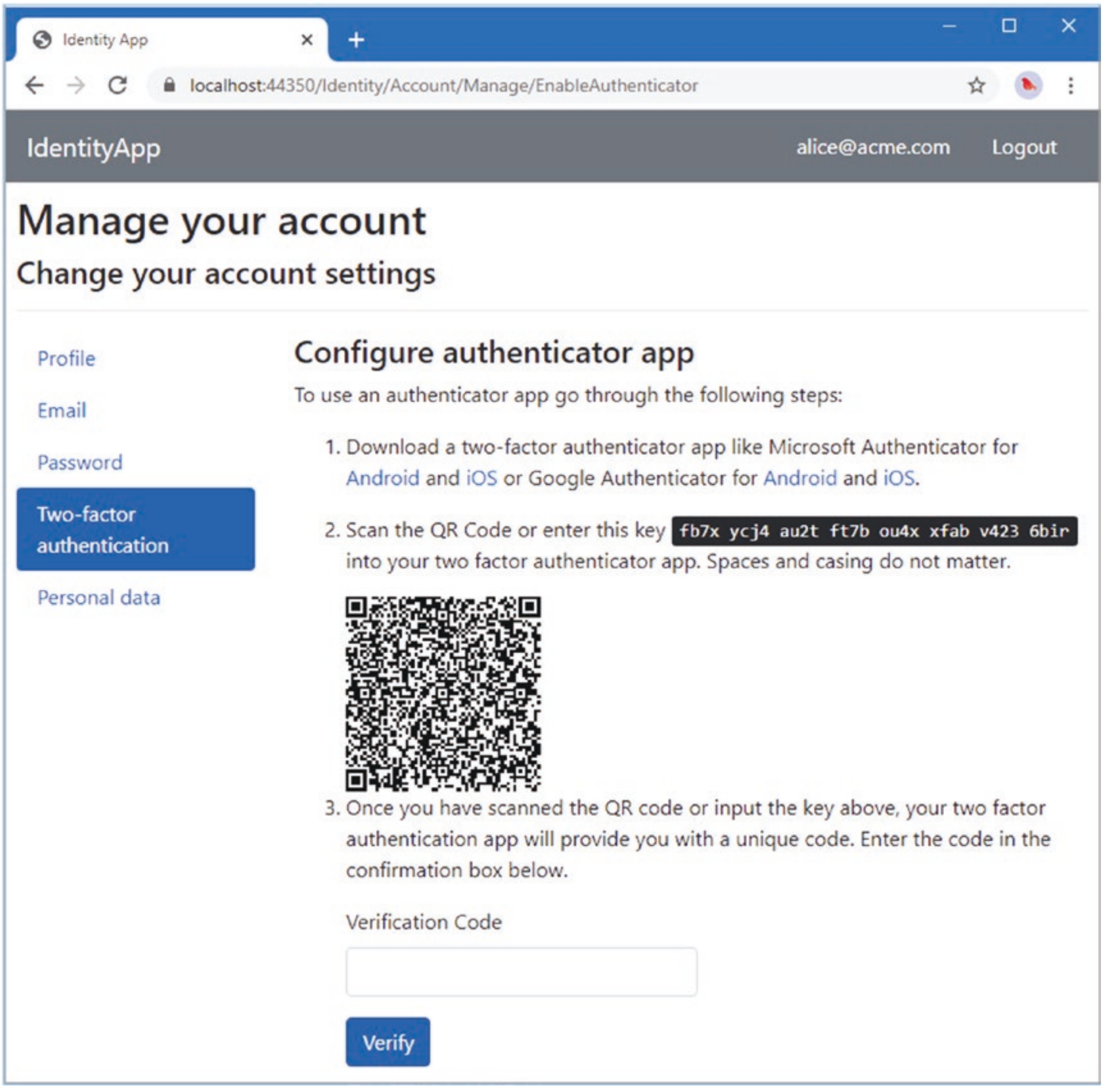
The first script element includes the JavaScript file from the qrcodejs package in the layout used for the Identity UI Razor Pages. The second script element looks for an HTML element with an ID of qrcode. If such an element exists, it is used to create a QR code image using the qrcodejs package, with the data used to generate the QR code obtained from the data-url attribute of an HTML element with an ID of qrCodeData. Finally, the element that occurs before the qrcode element is removed. This may seem like an oddly specific sequence of actions until you learn that the HTML produced by the Razor Page used to set up authenticators contains these elements:

```
...
<div class="alert alert-info">Learn how to
    <a href="https://go.microsoft.com/fwlink/?Linkid=852423">
         enable QR code generation
    </a>.
</div>
```

```
<div id="qrCode"></div>
<div id="qrCodeData" data-url="@Model.AuthenticatorUri"></div>
...
```

I explain how I knew the Razor Page contained these elements in Chapter 6, where I explain how to customize the Identity UI package.

Restart ASP.NET Core, ensure you are signed in using `alice@acme.com` with `MySecret1$` as the password. Click the email address on the right of the header and click Two-Factor Authentication. Click the Add Authenticator App button, and you will be presented with instructions for configuring an authenticator, as shown in Figure 4-8, including a QR code that can be scanned by devices with a camera.


Identity App
localhost:44350/Identity/Account/Manage/EnableAuthenticator
IdentityApp
alice@acme.com
Logout
Profile
Email
Password
Two-factor authentication
Personal data
Verify

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 for Android and iOS or Google Authenticator for Android and iOS.
2. Scan the QR Code or enter this key `fb7x ycj4 au2t ft7b ou4x xfab v423 6bir` into your two factor authenticator app. Spaces and casing do not matter.
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 4-8. *Displaying a QR code*

Using the Identity UI Workflows

The basic configuration of the Identity UI package is complete. In Chapter 6, I explain how to customize the Identity UI package, but before doing that, I describe the features the Identity UI package provides by default and detail the Razor Pages that each relies on, which is useful when it comes to customization.

I use the term *workflow* in this book to refer to the processes that can be performed using Identity. Each workflow combines multiple features to support a task, such as creating a new user account or changing a password.

Registration

The Identity UI package supports self-registration, which means that anyone can create a new account and then use it to sign into the application. There is one additional feature enabled by the configuration changes in the previous section, which is that a confirmation email is sent when a new account is created. Restart ASP.NET Core, request `https://localhost:44350/Identity/Account/Register`, and use the values in Table 4-5 to create a new account.

Table 4-5. *Account Registration Details*

Field	Value
`Email`	`bob@example.com`
`Password`	`MySecret1$`
`ConfirmPassword`	`MySecret1$`

Click the Register button, and you will see an email like this one displayed in the console output:

```
---New Email----
To: bob@example.com
Subject: Confirm your email
Please confirm your account by <a href='https://localhost:44350/Identity/Account/
ConfirmEmail?userId=9e8c0aed-3990-4806-9d8d-19b9b543d7c2&code=Q2ZESjhBMVB9&returnUrl=%2FStore'>
clicking here</a>.
-------
```

Identity can be configured to require the user to click the confirmation link before signing into the application, as I explain in Chapter 9. Table 4-6 lists the Identity UI Razor Pages used in the registration process.

Table 4-6. *The Identity UI Pages for Registration*

Page	Description
`Account/Register`	This page prompts the user to create a new account.
`Account/RegisterConfirmation`	This is the page that handles the URLs sent in confirmation emails.
`Account/ResendEmailConfirmation`	This page allows the user to request another confirmation email.
`Account/ConfirmEmail`	This is the page that handles the URLs sent in the emails when the user requests a confirmation be reset.

Signing In and Out of the Application

One of the most important features provided by the Identity UI package is to sign users in and out of the application, establishing their identity with which authorization policies can be evaluated.

You didn't need to explicitly sign into the application earlier in the chapter because the Identity UI package signs in new accounts automatically. To explicitly sign in, request `https://localhost:44350/Identity/Account/Login` and sign in using `alice@acme.com` as the email address and `MySecret1$` as the password, as shown in Figure 4-9.

You can log out of the application by requesting `https://localhost:44350/Identity/Account/Logout` and clicking the link to log out, also shown in Figure 4-9.

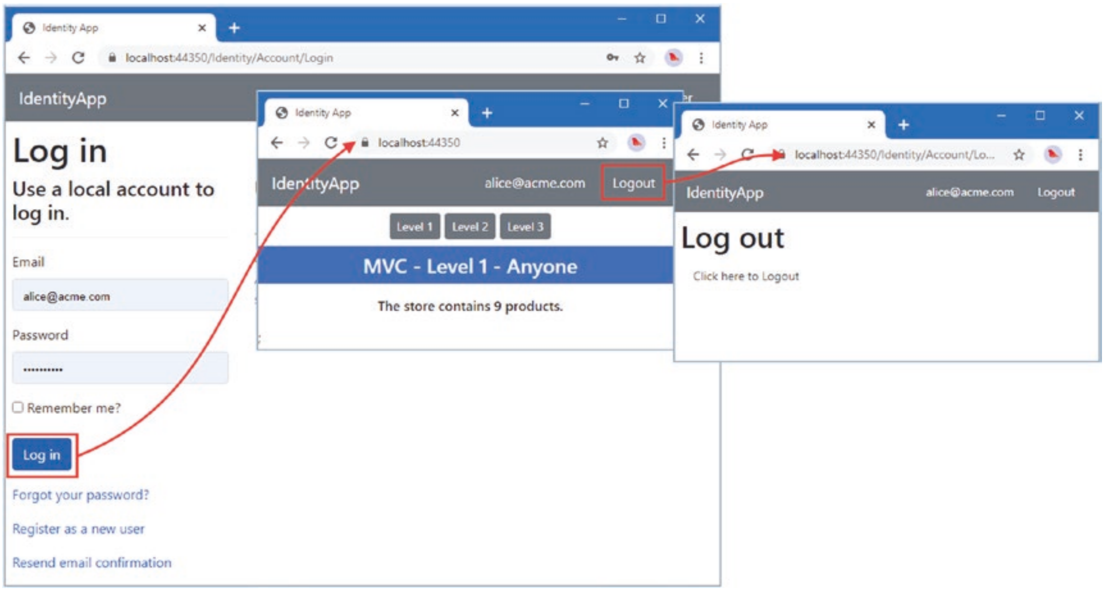


Figure 4-9. *The basic login sequence*

Table 4-7 lists the Identity UI Razor Pages for signing in and out of an application.

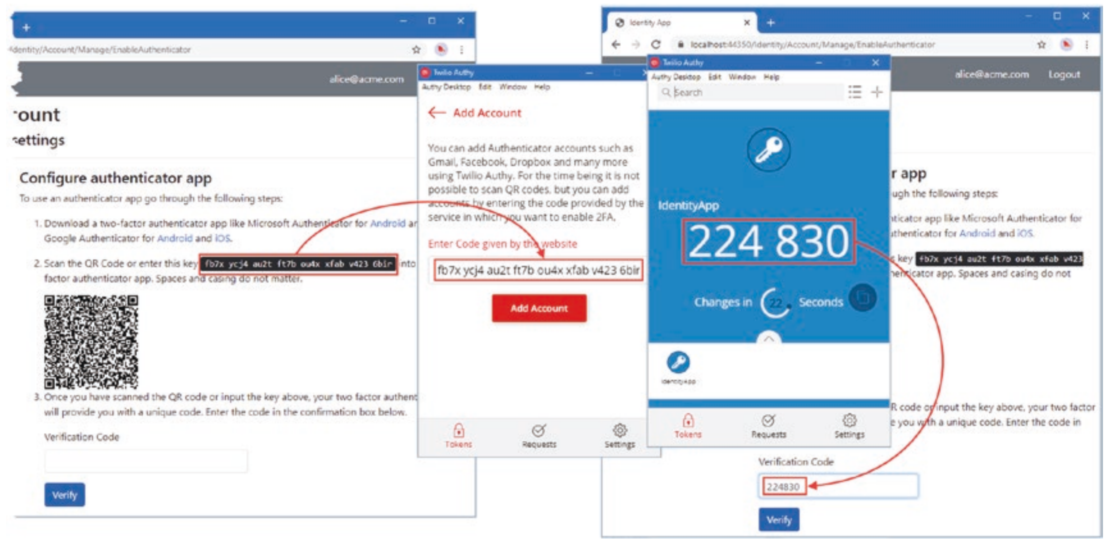
Table 4-7. *The Identity UI Pages for Signing In and Signing Out*

Page	Description
`Account/Login`	This page asks the user for their credentials or, if configured, to choose an external authentication service (which I describe in Chapter 11).
`Account/ExternalLogin`	This page is displayed after the user has signed into the application using an external authentication service, as described in Chapter 11.
`Account/SetPassword`	This page is used when an account has been created with an external authentication provider but the user wants to be able to sign in with a local password.
`Account/Logout`	This page allows the user to sign out of the application.
`Account/Lockout`	This page is displayed when the account is locked out following a series of failed sign-ins. I explain how to configure the lockout feature in Chapter 9.

Using Two-Factor Authentication

As noted earlier, Identity supports a range of two-factor authentication options, one of which—authenticators—is available through the Identity UI package. To explore this workflow, you will need an authenticator application. I use the Authy app (`authy.com`) for the examples in this book because there is a Windows client, but there are popular options from Google and Microsoft that run on mobile devices.

Request `https://localhost:44350/Identity/Account/Login` and sign in using `alice@acme.com` as the email address and `MySecret1$` as the password. Navigate to the self-management feature by clicking the email address in the header, click Two-Factor Authentication, and then the Set Up Authenticator App button. You will be presented with a setup key and a QR code that can be used to set up an authenticator app. Scan the QR code or type in the setup key, and the authenticator will start generating codes every 30 seconds. Enter the current code into the Verification Code text field to complete the two-factor authentication setup, as shown in Figure 4-10.

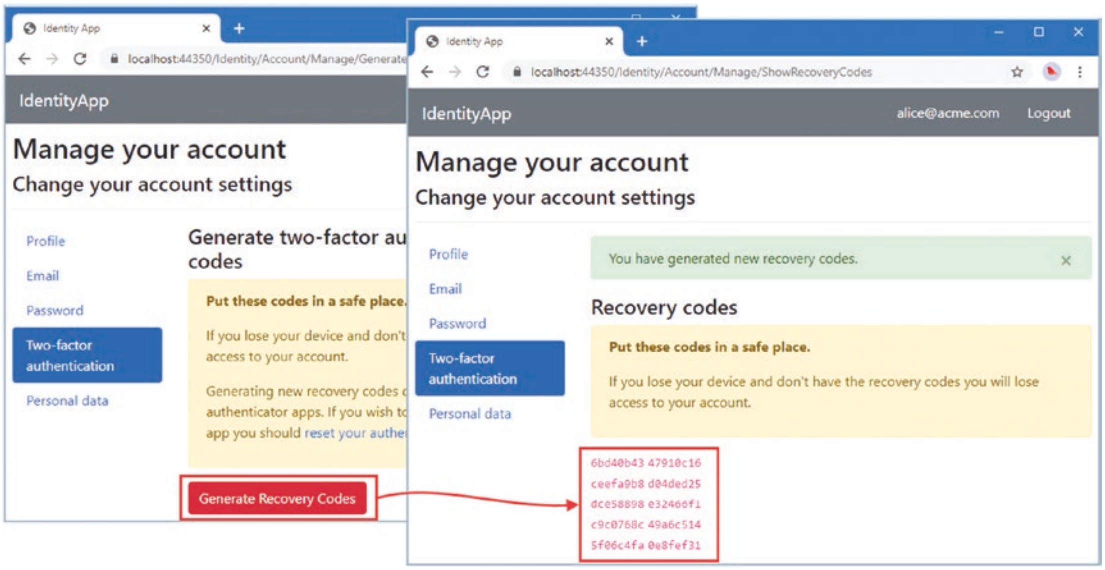


alice@acme.com
Configure authenticator app
To use an authenticator app go through the following steps:
1. Download a two-factor authenticator app like Microsoft Authenticator for Android and Google Authenticator for Android and iOS.
2. Scan the QR Code or enter this key fb7x ycj4 au2t ft7b ou4x xfab v423 6bir into your two factor authenticator app. Spaces and casing do not matter.
3. Once you have scanned the QR code or input the key above, your two factor authenticator app will provide you with a unique code. Enter the code in the confirmation box below.
Verification Code
Verify
Twilio Authy
Authy Desktop Edit Window Help
Add Account
You can add Authenticator accounts such as Gmail, Facebook, Dropbox and many more using Twilio Authy. For the time being it is not possible to scan QR codes, but you can add accounts by entering the code provided by the service in which you want to enable 2FA.
Enter Code given by the website
fb7x ycj4 au2t ft7b ou4x xfab v423 6bir
Add Account
Tokens
Requests
Settings
Identity App
localhost:44350/Identity/Account/Manage/EnableAuthenticator
Search
IdentityApp
224 830
Changes in 22 Seconds
alice@acme.com
Logout
224830
Verify

Figure 4-10. *Setting up two-factor authentication*

Once you have set up an authenticator, you will be redirected to the TwoFactorAuthentication page, which presents buttons for different management tasks. The Reset Recovery Codes button is used to generate single-use codes that can be used to sign in if the authenticator app is unavailable (such as when a mobile device has been lost or stolen).

Click the button, and you will be presented with a set of recovery codes, as shown in Figure 4-11. It is not obvious, but each line shows two recovery codes, separated by a space. Each code can be used only once, after which it is invalidated. The Identity UI package doesn't allow the remaining codes to be inspected by the user, although this is possible when using Identity directly, as I demonstrate in Part 2. Make a note of the first code you generated, which will be different from the ones shown in the figure.


Identity App
localhost:44350/Identity/Account/Manage/Generate
IdentityApp
Manage your account
Change your account settings
Profile
Email
Password
Two-factor authentication
Personal data
Generate two-factor au
codes
Put these codes in a safe place.
If you lose your device and don't
access to your account.
Generating new recovery codes
authenticator apps. If you wish to
app you should reset your authe
Generate Recovery Codes

Identity App
localhost:44350/Identity/Account/Manage/ShowRecoveryCodes
IdentityApp
alice@acme.com
Logout
Manage your account
Change your account settings
Profile
Email
Password
Two-factor authentication
Personal data
You have generated new recovery codes.
Recovery codes
Put these codes in a safe place.
If you lose your device and don't have the recovery codes you will lose access to your account.

```
6bd40b43 47910c16
ceefa9b8 d04ded25
dce58898 e32466f1
c9c0768c 49a6c514
5f06c4fa 0e8fef31
```

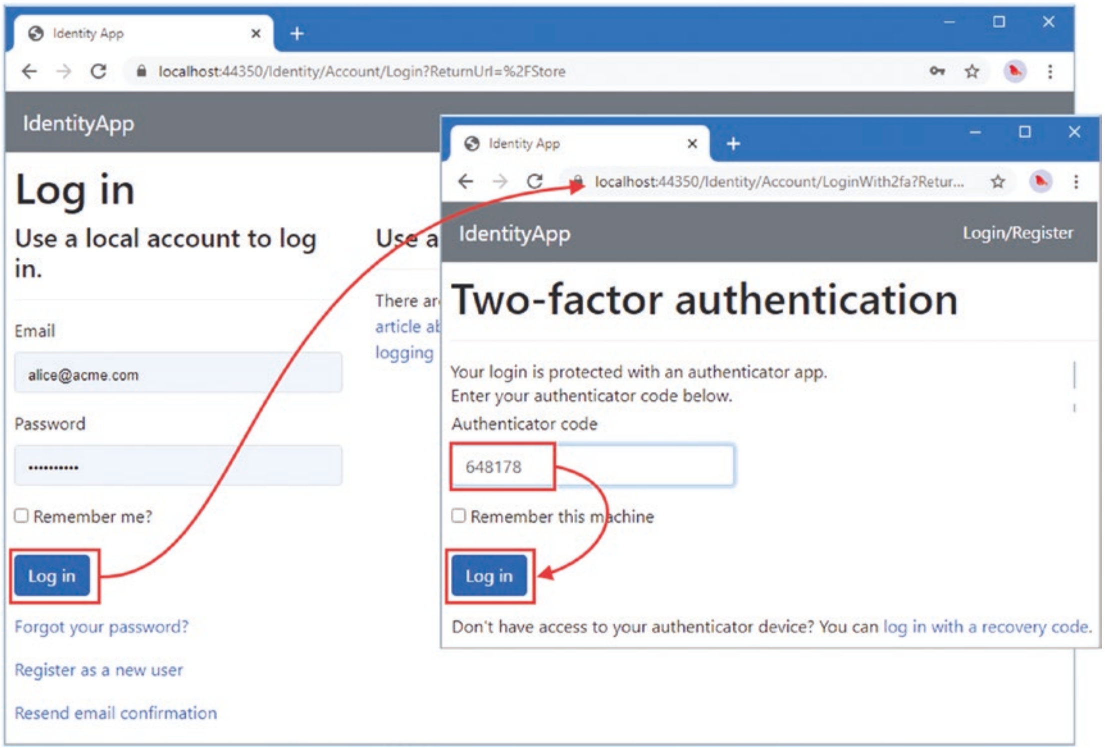
Figure 4-11. *Generating recovery codes*

Table 4-8 describes the Identity UI Razor Pages used to configure an authenticator and generate recovery codes.

Table 4-8. *The Identity UI Pages for Managing an Authenticator*

Page	Description
`Account/Manage/TwoFactorAuthentication`	This is the page displayed when the user clicks the Two-Factor Authentication link in the self-management feature. It links to other pages that handle individual authenticator tasks.
`Account/Manage/EnableAuthenticator`	This page displays the QR code and setup key required to configure an authenticator.
`Account/Manage/ResetAuthenticator`	This page allows the user to generate a new authenticator setup code, which will invalidate the existing authenticator and allow a new one to be set up, which is done by the `EnableAuthenticator` page.
`Account/Manage/GenerateRecoveryCodes`	This page generates a new set of recovery codes and then redirects to the `ShowRecoveryCodes` page to display them.
`Account/Manage/ShowRecoveryCodes`	This page displays a newly generated set of recovery codes.
`Account/Manage/Disable2fa`	This page allows the user to disable the authenticator and return to signing into the application with just a password.

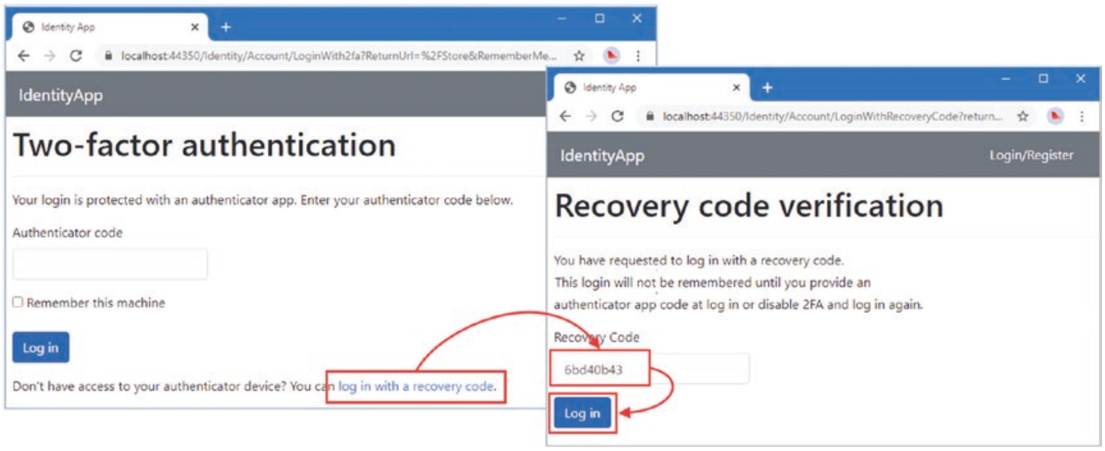
Click Logout in the header and sign into the application again, using `alice@acme.com` as the email address and `MySecret1$` as the password. Once the password has been checked, you will be prompted to enter the current code displayed by the authenticator app. Enter the code and click the Log In button, as shown in Figure 4-12. (You can also select the "Remember this machine" option, which creates a cookie that allows sign-in without needing the authenticator, as explained in Chapter 11.)


Identity App
localhost:44350/Identity/Account/Login?ReturnUrl=%2FStore
IdentityApp
Log in
Use a local account to log in.
Email
alice@acme.com
Password
••••••••••
Remember me?
Log in
Forgot your password?
Register as a new user
Resend email confirmation

Identity App
localhost:44350/Identity/Account/LoginWith2fa?Retur...
IdentityApp
Login/Register
Two-factor authentication
Your login is protected with an authenticator app. Enter your authenticator code below.
Authenticator code
648178
Remember this machine
Log in
Don't have access to your authenticator device? You can log in with a recovery code.

Figure 4-12. *Signing in using an authenticator code*

Repeat the process, but click the Log In with a Recovery Code link instead of entering the authenticator code. You will be prompted to enter one of the recovery codes you generated earlier, as shown in Figure 4-13.



Identity App

localhost:44350/Identity/Account/LoginWith2fa?ReturnUrl=%2FStore&RememberMe...

IdentityApp

Two-factor authentication

Your login is protected with an authenticator app. Enter your authenticator code below.

Authenticator code

Remember this machine

Log in

Don't have access to your authenticator device? You can log in with a recovery code.

Identity App

localhost:44350/Identity/Account/LoginWithRecoveryCode?return...

IdentityApp Login/Register

Recovery code verification

You have requested to log in with a recovery code.
This login will not be remembered until you provide an authenticator app code at log in or disable 2FA and log in again.

Recovery Code

6bd40b43

Log in

Figure 4-13. *Signing in with a recovery code*

Table 4-9 describes the Razor Pages used when signing into the application using an authenticator or with a recovery code.

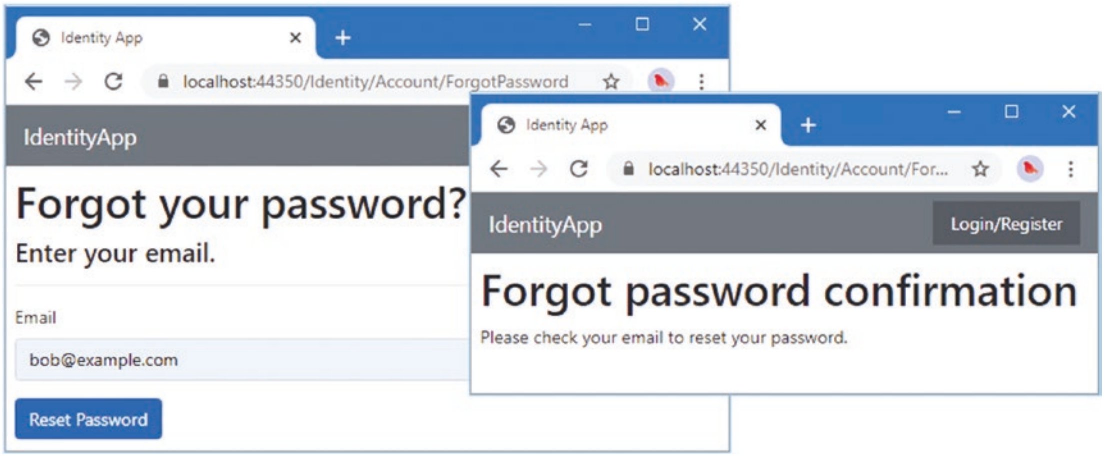
Table 4-9. *The Identity UI Pages for Two-Factor Authentication*

Name	Description
`Account/LoginWith2fa`	This page prompts the user to enter an authenticator code.
`Account/LoginWithRecoveryCode`	This page prompts the user to enter a recovery code.

Recovering a Password

If a user has forgotten their password, they can go through a recovery process to generate a new one. Password recovery works only if a user confirmed their email address following registration—the Identity UI package won't send the recovery password email if a user hasn't confirmed their email address. To see the recovery workflow, request `https://localhost:44350/Identity/Account/Login` and click the Forgotten Your Password? link. Enter bob@example.com into the text field and click the Reset Password button, as shown in Figure 4-14.

■ **Note** The `bob@example.com` account was created after email configurations were configured earlier in the chapter. If you want to test recovery for the `alice@acme.com` account, then click the Resend Email Confirmation link displayed by the `Account/Login` page and paste the link from the confirmation email into a browser window.


Identity App
localhost:44350/Identity/Account/ForgotPassword
IdentityApp
Reset Password
Identity App
localhost:44350/Identity/Account/For...
IdentityApp
Login/Register

Forgot your password?

Enter your email.

Email

bob@example.com

Forgot password confirmation

Please check your email to reset your password.

Figure 4-14. *Requesting password recovery*

Examine the console output from ASP.NET Core, and you will see a password recovery email, like this (although I have shortened the security code for brevity):

```
---New Email----
To: bob@example.com
Subject: Reset Password
Please reset your password by
<a href='https://localhost:44350/Identity/Account/ResetPassword?code=Q2ZESjhBM'>
    clicking here
</a>.
-------
```

Copy the link from the email into a browser window, and you will be prompted to reenter the email address and choose a new password, as shown in Figure 4-15. Enter bob@example.com into the email field and enter MySecret2$ as the new password.

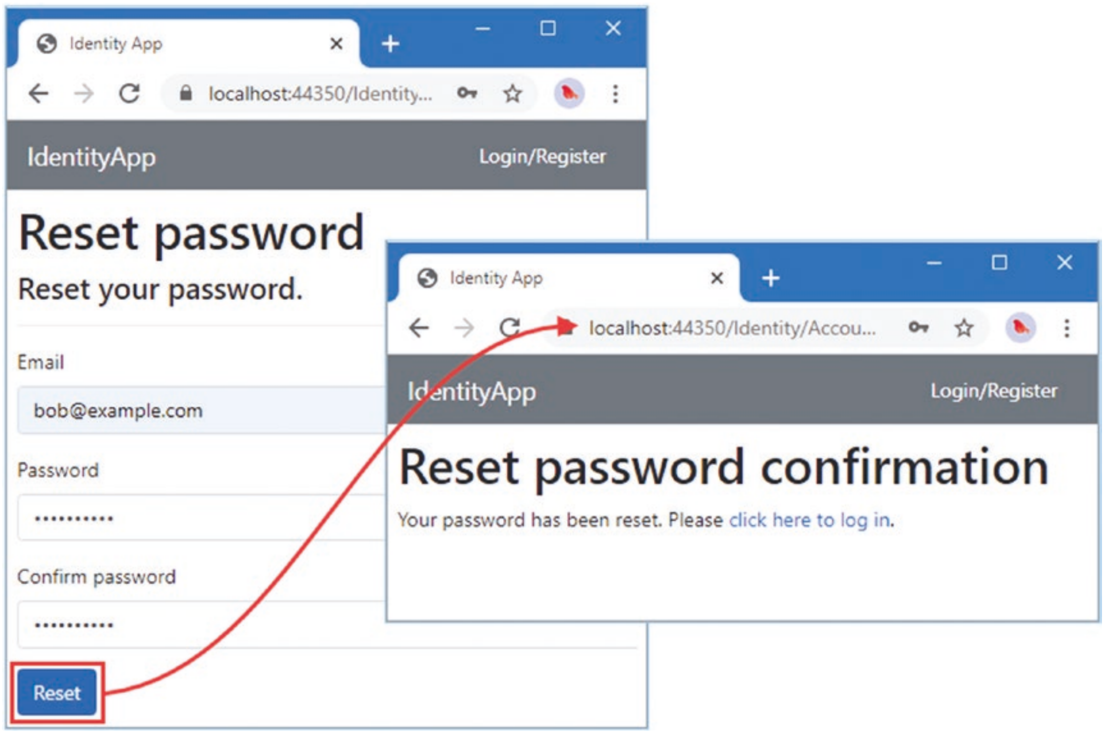

Identity App
localhost:44350/Identity...
IdentityApp
Login/Register
Reset password
Reset your password.
Email
bob@example.com
Password
Confirm password
Reset
localhost:44350/Identity/Accou...
Reset password confirmation
Your password has been reset. Please click here to log in.

Figure 4-15. *Choosing a new password*

When you click the Reset button, the password stored in the Identity user store will be updated, and you can sign into the application using the new password. Table 4-10 describes the Identity UI Razor Pages that support the password recovery process.

Table 4-10. *The Identity UI Pages for Password Recovery*

Name	Description
`Account/ForgotPassword`	This page prompts the user for their email address and sends the confirmation email.
`Account/ForgotPasswordConfirmation`	This page is displayed once the confirmation email has been sent.
`Account/ResetPassword`	This page is targeted by the URL sent in the confirmation email. It prompts the user for their email address and a new password.
`Account/ResetPasswordConfirmation`	This page is displayed once the password has been changed and provides the user with confirmation that the process has been completed.

Changing Account Details

The self-management features include support for changing the user's details, including the phone number, email address, and password. Request `https://localhost:44350/Identity/Account/Login` and sign in using `bob@example.com` as the email address and `MySecret2$` as the password. Click the email address in the header, and you will be presented with the default self-management page, which allows the user's phone number to be set or changed.

Click the Email link, and you will be presented with the page that allows a new email address to be specified. Click the Password link, and you will be prompted for the existing password and a new password. Figure 4-16 shows the phone number and password pages.

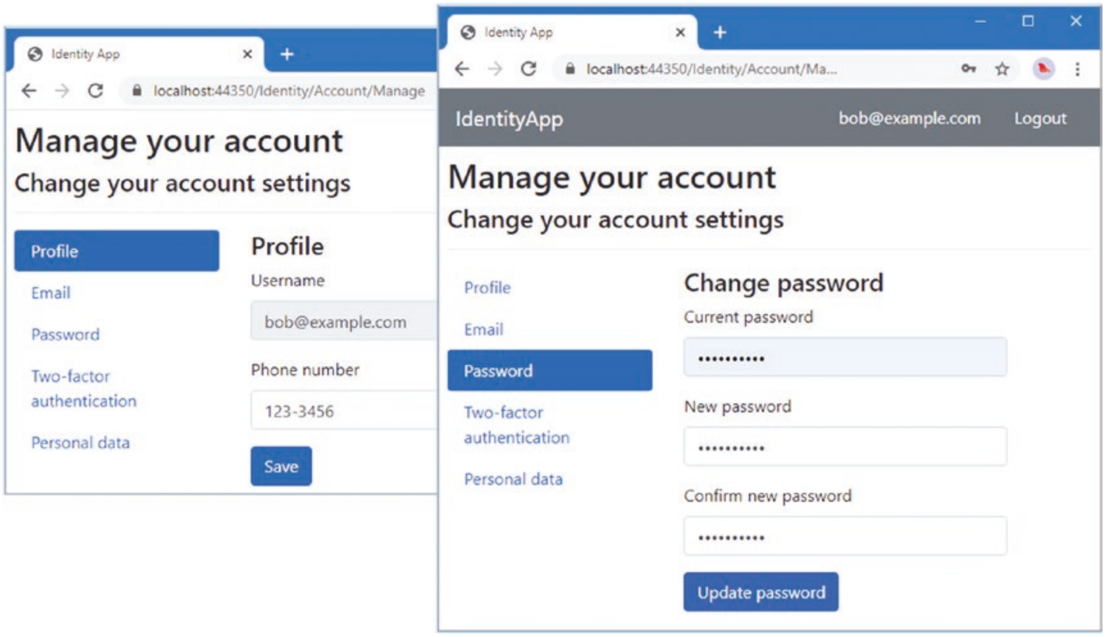


Figure 4-16. *The phone and password change pages*

Table 4-11 describes the Identity UI Razor Pages that support changing account details.

Table 4-11. *The Identity UI Pages for Changing Account Details*

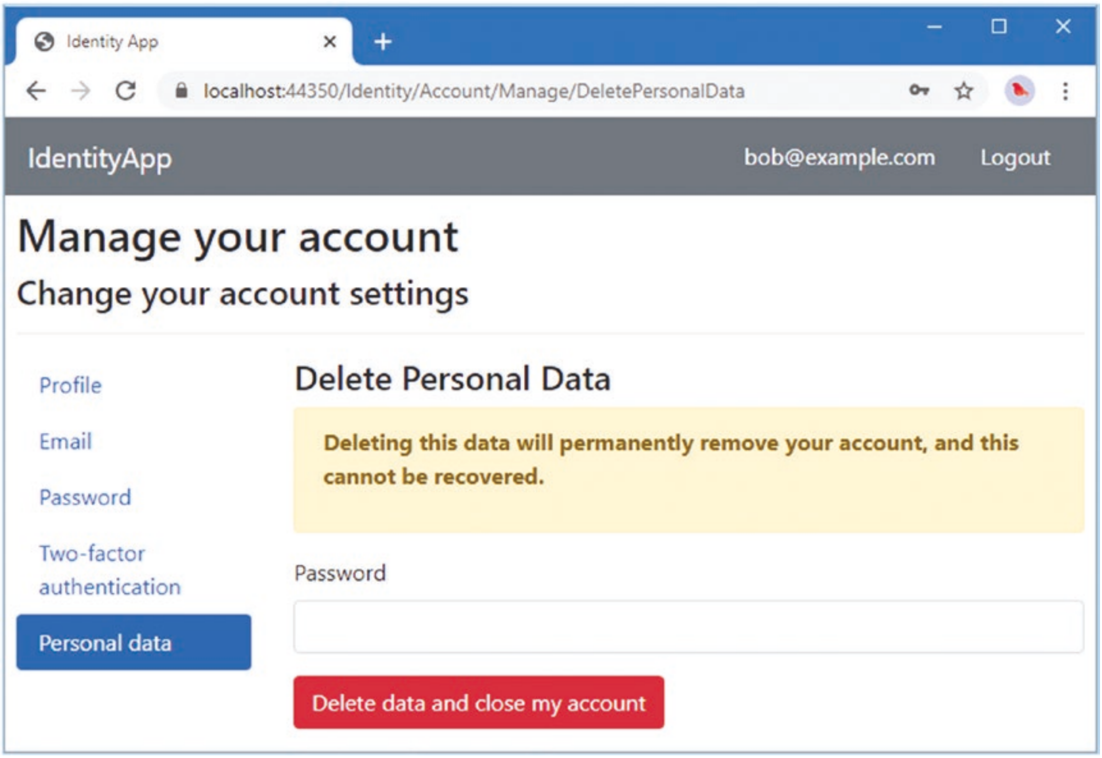
Name	Description
`Account/Manage/Index`	This page allows the user to set a phone number
`Account/Manage/ChangePassword`	This page allows a new password to be chosen.
`Account/Manage/Email`	This page allows a new email address to be chosen and sends a confirmation email to the user.
`Account/ConfirmEmailChange`	This page is targeted by the URL in the confirmation email and updates the user store with the new email address.

Managing Personal Data

In some regions, users have a right to personal data and to request that their data is deleted. The Identity UI package provides a generic personal data feature that provides access to the data in the user store and allows the user to delete their account.

■ **Caution** Don't assume that the features provided by the Identity UI package are sufficient to comply with any specific data access and retention regulation. You must ensure your application complies with the regulations in every region in which you have users.

Request `https://localhost:44350/Identity/Account/Login` and sign in using `bob@example.com` as the email address and `MySecret2$` as the password. Click the email address in the header to navigate to the self-management page and then click the Personal Data link. Clicking the Download button generates a JSON document containing the data from the user store. Clicking the Delete button prompts the user for their password before deleting their account, as shown in Figure 4-17.


Identity App
localhost:44350/Identity/Account/Manage/DeletePersonalData
IdentityApp
bob@example.com
Logout
Profile
Email
Password
Two-factor authentication
Personal data
Delete data and close my account

Manage your account

Change your account settings

Delete Personal Data

Deleting this data will permanently remove your account, and this cannot be recovered.

Password

***Figure 4-17.** Managing personal data*

Table 4-12 describes the Identity UI Razor Pages that support the personal data features.

Table 4-12. *The Identity UI Pages for Managing Personal Data*

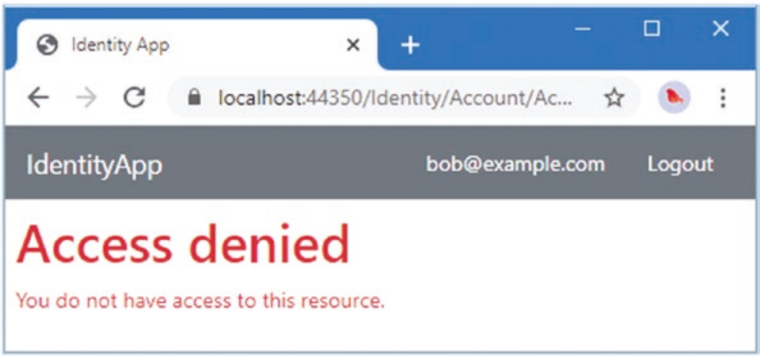
Name	Description
`Account/Manage/PersonalData`	This is the page that presents the user with the buttons for downloading or deleting data.
`Account/Manage/DownloadPersonalData`	This is the page that generates the JDON document containing the user's data.
`Account/Manage/DeletePersonalData`	This is the page that prompts the user for their password and deletes the account.

Denying Access

The final workflow is used when the user is denied access to an action or Razor Page. This is known as the *forbidden response*, and it is the counterpart to the challenge response that prompts for user credentials. (The difference is that the challenge response is used when authorization is required but no user is signed in. The forbidden response is used when authorization is required but the signed-in user is not allowed access.)

One limitation of the Identity UI package is that it doesn't cater for users being assigned to roles, which is the authorization requirement I specified for the most restricted part of the example application. This means that no user account created using the Identity UI package will be granted access, which you can confirm by signing in and clicking the Level 3 button. The application sends the Identity UI package's forbidden response, as shown in Figure 4-18.

■ **Tip** I explain how to use the Identity API to add users to roles in Chapter 10.



Access denied

You do not have access to this resource.

Figure 4-18. *The forbidden response*

For completeness, Table 4-13 describes the Identity UI Razor Page used for the forbidden response.

Table 4-13. *The Identity UI Page for the Forbidden Response*

Name	Description
`Account/AccessDenied`	This page displays a warning to the user.

Summary

In this chapter, I explained how to add Identity and the Identity UI package to a project. I showed you how to prepare the Identity database, I explained how to override individual files to create a consistent layout, and I described the default workflows that the Identity UI package provides. In the next chapter, I show you how to configure ASP.NET Core Identity.