CHAPTER 7

Using the Identity API

As I explained in Chapter 6, there are limits to the customizations that can be made to the Identity UI package. Minor changes can be achieved using scaffolding, but if your application doesn't fit into the self-service model that Identity UI expects, then you won't be able to adapt its features to suit your project.

In this chapter—and for the rest of this part of the book—I describe the API that ASP.NET Core Identity provides, which can be used to create completely custom workflows. This is the same API that Identity UI uses, but using it directly means you can create any combination of features you require and implement them exactly as needed. In this chapter, I describe the basic features that the API provides. In later chapters, I explain more advanced features and create administrator and self-service workflows for every major Identity feature. Table 7-1 puts the Identity API in context.

Table 7-1. *Putting the Identity API in Context*

Question	Answer
What is it?	The Identity API provides access to all of the Identity features.
Why is it useful?	The API allows custom workflows to be created that perfectly match the requirements of a project, which may not be what the Identity UI package provides.
How is it used?	Key classes are provided as services that are available through the standard ASP.NET Core dependency injection feature.
Are there any pitfalls or limitations?	The API can be complex, and creating custom workflows requires a commitment of time and effort. It is also important to think through the workflows you create to ensure that you are creating a secure application.
Are there any alternatives?	You can use and adapt the Identity UI package if your project fits into the model it is designed for.

Table 7-2 summarizes the chapter.

Table 7-2. *Chapter Summary*

Problem	Solution	Listing
Create a user account	Create a new instance of the `IdentityUser` class and pass it to the user manager's `CreateAsync` method.	1–15
Determine the outcome from a user manager operation	Read the properties defined by the `IdentityResult` class, which indicate the outcome and describe any errors.	16–18
Query the user store	Formulate a LINQ query with the user manager's `Users` property.	19–22
Display user details	Enumerate the properties of an `IdentityUser` object or call the corresponding user manager methods.	23–26
Update user details	Set new `IdentityUser` property values or call the corresponding user manager methods and then call the `UpdateAsync` method to store the changes.	27–30
Determine the features supported by the user store	Access the user store through the user manager's `Store` property and read the properties that are defined for each feature.	31–33
Enable support for roles in the default user store	Use the `AddIdentity` method to set up Identity, with generic type arguments to specify the user and role classes.	34

Preparing for This Chapter

This chapter uses the `IdentityApp` project from Chapter 6. Open a new PowerShell command prompt and run the commands shown in Listing 7-1 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 7-1. Resetting the Databases

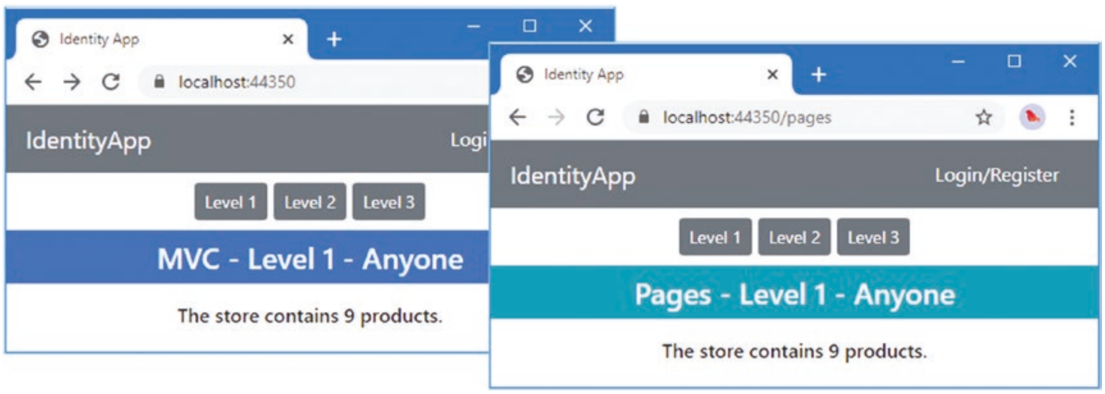
```
dotnet ef database drop --force --context ProductDbContext
dotnet ef database drop --force --context IdentityDbContext
dotnet ef database update --context ProductDbContext
dotnet ef database update --context IdentityDbContext
```

Use the PowerShell prompt to run the command shown in Listing 7-2 in the `IdentityApp` folder to start the application.

Listing 7-2. 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 7-1.



MVC - Level 1 - Anyone

The store contains 9 products.

Pages - Level 1 - Anyone

The store contains 9 products.

Figure 7-1. *Running the example application*

Creating the User and Administrator Dashboards

For this part of the book, I am going to create custom workflows for the operations commonly required by most applications, in versions that can be used by administrators and, where appropriate, by self-service users.

I am going to create two "dashboard"-style layouts, one for administrator functions and one for self-service. As I develop individual workflows, I will add navigation elements to the dashboards so that the new features can be easily accessed.

These layouts will be color-coded so that it is obvious when an example is intended for an administrator or a user. It will also give the custom workflows an appearance that is distinct from the Identity UI examples shown in earlier examples. Initially, the new workflows will exist alongside Identity UI, but at the end of this chapter, once I have some basic features established, I change the Identity configuration to disable Identity UI.

Create the `IdentityApp/Pages/Identity` folder and add to it a Razor Layout named `_Layout.cshtml`, with the contents shown in Listing 7-3.

USING RAZOR PAGES OR THE MVC FRAMEWORK

I use Razor Pages throughout this book for the examples where I consume the Identity API, following the same basic pattern as the Identity UI package from Microsoft. The development style of Razor Pages fits nicely with the nature of Identity, where features are generally self-contained. You don't have to use Razor Pages in your projects, and every service, method, and property that I use in a Razor Page can be accessed in the same way using the MVC Framework.

Listing 7-3. The Contents of the _Layout.cshtml File in the Pages/Identity Folder

```
@{
    string theme = ViewData["theme"] as string ?? "primary";
    bool showNav = ViewData["showNav"] as bool? ?? true;
    string navPartial = ViewData["navPartial"] as string ?? "_Workflows";
    string workflow = ViewData["workflow"] as string;
    string banner =  ViewData["banner"] as string ?? "User Dashboard";
    bool showHeader = ViewData["showHeader"] as bool? ?? true;
}

<!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>
    @if (showHeader) {
        <nav class="navbar navbar-dark bg-@theme">
            <a class="navbar-brand text-white">IdentityApp</a>
            <div class="text-white"><partial name="_LoginPartial" /></div>
        </nav>
    }
    <h4 class="bg-@theme text-center text-white p-2">@banner</h4>
    <div class="my-2">
        <div class="container-fluid">
            <div class="row">
                @if (showNav) {
                    <div class="col-auto">
                        <partial name="@navPartial" model="@((workflow, theme))" />
                    </div>
                }
                <div class="col">
                    @RenderBody()
                </div>
            </div>
        </div>
    </div>
</body>
</html>
```

I use ViewData values to select the Bootstrap theme to determine whether to display navigation content and to display a banner.

Next, add a Razor View named _Workflows.cshtml to the Pages/Identity folder with the content shown in Listing 7-4. This partial view will display navigation buttons that will lead to different workflows.

Listing 7-4. The Contents of the _Workflows.cshtml File in the Pages/Identity Folder

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

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

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

Next, create the `IdentityApp/Pages/Identity/Admin` folder and add to it a Razor Layout, called `_AdminLayout.cshtml`, with the content shown in Listing 7-5. This layout builds on the one in Listing 7-4 and will denote the administration workflows.

Listing 7-5. The Contents of the _AdminLayout.cshtml File in the Pages/Identity/Admin Folder

```
@{
    Layout = "../_Layout";
    ViewData["theme"] = "success";
    ViewData["banner"] = "Administration Dashboard";
    ViewData["navPartial"] = "_AdminWorkflows";
}

@RenderBody()
```

This may seem odd for a layout, but I only need to set the view data properties to differentiate the administration view from the user view, as you will see once the parts start to come together. Add a Razor View named `_AdminWorkflows.cshtml` to the `Pages/Identity/Admin` folder with the content shown in Listing 7-6. This partial view will contain the navigation elements for the administration workflows.

Listing 7-6. The Contents of the _AdminWorkflows.cshtml File in the Pages/Identity/Admin Folder

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

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

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

Add a Razor View Start file named `_ViewStart.cshtml` to the `Pages/Identity/Admin` folder with the content shown in Listing 7-7. I have kept the names of the user and administration files distinct to make it easier to follow the examples, and the Razor View Start file ensures that the `_AdminLayout.cshtml` file will be used as the default layout for pages in the `Admin` folder.

Listing 7-7. The Contents of the _ViewStart.cshtml File in the Pages/Identity/Admin folder

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

Creating the Custom Base Classes

As I start to define workflows, they will be publicly accessible to make development easier. Once I get enough functionality in place, I will use the ASP.NET Core authorization features to restrict access so that user features are only available for signed-in users and administration features are only available to designated administrators. Applying the authorization policy is simpler when all the related Razor Pages share a common page model base class. To define the base class for user features, add a class file named `UserPageModel.cs` to the `Pages/Identity` folder with the content shown in Listing 7-8.

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

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

namespace IdentityApp.Pages.Identity {

    public class UserPageModel : PageModel {

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

To create the common base class for the administration features, add a class file named `AdminPageModel.cs` to the `Pages/Identity/Admin` folder with the content shown in Listing 7-9.

Listing 7-9. The Contents of the AdminPageModel.cs File in the Pages/Identity/Admin Folder

```
namespace IdentityApp.Pages.Identity.Admin {

    public class AdminPageModel: UserPageModel {

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

I return to these classes when I apply the authorization policy that will restrict access to the custom Identity workflows.

Creating the Overview and Dashboard Pages

Add a Razor Page named `Index.cshtml` to the `Pages/Identity` folder with the content shown in Listing 7-10.

Listing 7-10. The Contents of the Index.cshtml File in the Pages/Identity Folder

```
@page
@model IdentityApp.Pages.Identity.IndexModel
@{
    ViewBag.Workflow = "Overview";
}

<table class="table table-sm table-striped table-bordered">
    <tbody>
        <tr><th>Email</th><td>@Model.Email</td></tr>
        <tr><th>Phone</th><td>@Model.Phone</td></tr>
    </tbody>
</table>
```

The view part of the page displays the user's email address and phone number. To define the page model class, add the code shown in Listing 7-11 to the `Index.cshtml.cs` file in the `Pages/Identity` folder. (You will have to create this file if you are using Visual Studio Code.)

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

```
namespace IdentityApp.Pages.Identity {

    public class IndexModel : UserPageModel {

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

The page model class defines the properties required by its view. I'll add the code to retrieve the data from the store later in this chapter.

Next, add a Razor Page named `Dashboard.cshtml` to the `Pages/Identity/Admin` folder with the content shown in Listing 7-12.

Listing 7-12. The Contents of the Dashboard.cshtml File in the Pages/Identity/Admin Folder

```
@page "/identity/admin"
@model IdentityApp.Pages.Identity.Admin.DashboardModel
@{
    ViewBag.Workflow = "Dashboard";
}

<table class="table table-sm table-striped table-bordered">
    <tbody>
        <tr><th>Users in store:</th><td>@Model.UsersCount</td></tr>
        <tr><th>Unconfirmed accounts:</th><td>@Model.UsersUnconfirmed</td></tr>
        <tr><th>Locked out users:</th><td>@Model.UsersLockedout</td></tr>
        <tr>
            <th>Users with two-factor enabled:</th>
            <td>@Model.UsersTwoFactor</td>
        </tr>
    </tbody>
</table>
```

The view part of this page displays a table with some useful overview data that will be useful in later chapters. To define the page model class, add the code shown in Listing 7-13 to the `Dashboard.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 7-13. The Contents of the Dashboard.cshtml.cs File in the Pages/Identity/Admin Folder

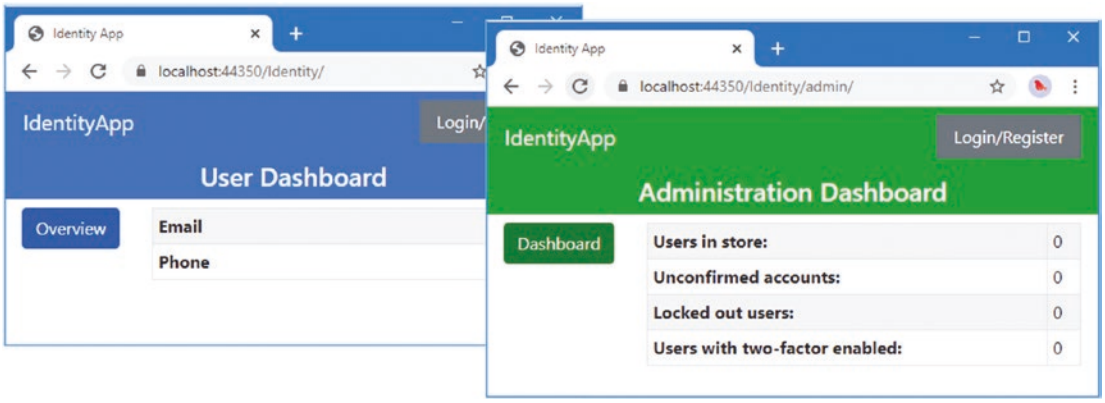
```
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace IdentityApp.Pages.Identity.Admin {

    public class DashboardModel : AdminPageModel {

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

The final change is to update the link that allows users to manage their accounts, as shown in Listing 7-14.

Listing 7-14. Changing URLs in the _LoginPartial.cshtml File in the Views/Shared Folder

```
@inject SignInManager<IdentityUser> SignInManager

<nav class="nav">
    @if (User.Identity.IsAuthenticated) {
        <a asp-page="/Identity/Index" class="nav-link bg-secondary text-white">
                @User.Identity.Name
        </a>
        <a asp-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>
```

To check the preparations for the administration workflows, restart ASP.NET Core and request `https://localhost:44350/identity`, which will show the overview that will be provided to the user. Next, request `https://localhost:44350/identity/admin`, which will show the dashboard for administrators. Both are shown in Figure 7-2.



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

Figure 7-2. *Preparing the user and administration dashboards*

Using the Identity API

Two of the most important parts of the Identity API are the user manager and the user class. The user manager provides access to the data that Identity manages, and the user class describes the data that Identity manages for a single user account.

The best approach is to jump in and write some code that uses the API. I am going to start with a Razor Page that will create instances of the user class and ask the user manager to store them in the database. Add the code shown in Listing 7-15 to the `Dashboard.cshtml.cs` file.

Listing 7-15. Using the Identity API in the Dashboard.cshtml.cs File in the Pages/Identity/Admin Folder

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

namespace IdentityApp.Pages.Identity.Admin {

    public class DashboardModel : AdminPageModel {

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

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

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

        private readonly string[] emails = {
            "alice@example.com", "bob@example.com", "charlie@example.com"
        };
        public async Task<IActionResult> OnPostAsync() {
            foreach (string email in emails) {
                IdentityUser userObject = new IdentityUser {
                    UserName = email,
                    Email = email,
                    EmailConfirmed = true
                };
                await UserManager.CreateAsync(userObject);
            }
            return RedirectToPage();
        }
    }
}
```

There are only a few statements, but there is a lot to understand because this is the first code that deals directly with the Identity API. Let's start with the user class. Identity is agnostic about the user class, and you can create a custom user class, which I demonstrate in Part 2. There is a default class, named `IdentityUser`, which is the class I have used in the example application and which should be used unless you are creating a custom user store.

The user class is declared when configuring Identity in the `Startup` class. Here is the statement that sets up Identity in the example application:

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

The type of the user class is specified using the generic type argument to the `AddDefaultIdentity` method. The user class defines a set of properties that describe the user account and provide the data values that Identity needs to implement its features. To create a new user object, I create a new instance of the `IdentityUser` class and set three of these properties, like this:

```
...
IdentityUser userObject = new IdentityUser {
    UserName = email,
    Email = email,
    EmailConfirmed = true
};
...
```

I describe all the properties defined by the `IdentityUser` class later in this chapter, but these properties are enough to get started. As the property names suggest, the `UserName` and `Email` properties store the user's account name and email address. The `EmailConfirmed` is used to indicate if the user's control of the email address has been confirmed. I describe the process for confirmation in Chapter 9, but I have set this property to `true` for the test accounts.

I am going to follow the policy used by the Identity UI package and use the user's email address for both the username and email address. (You can see examples that handle them separately in Part 2.)

The second key class is the user manager, which is `UserManager<T>`. The generic type argument, `T`, is used to specify the user class. Since the example application uses the built-in user class, the user manager will be `UserManager<IdentityUser>`.

REFERRING TO A SPECIFIC USER CLASS

Identity can work with just about any user class, but most projects end up using the default `IdentityUser` class because there are few good reasons to use anything else, especially if you are using Entity Framework Core to store the Identity data. For this reason, I am going to refer to `UserManage<IdentityUser>` instead of "`UserManager<T>`, where `T` is the user class." You will encounter "where `T` is the user class" a lot in in Part 2, and it is probably the phrase I repeat most often. For the rest of this part of the book, however, I am going to assume that the user class is `IdentityUser` for the sake of simplicity and brevity.

The user manager is configured as a service through the ASP.NET Core dependency injection feature. To access the user manager in the Razor Page model class, I added a constructor with a `UserManager<IdentityUser>` parameter, like this:

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

When the page model class is instantiated, the constructor receives a `UserManager<IdentityUser>` object, which I assign to a property named `UserManager` so that it can be used by the page handler methods or accessed from the view part of the page.

The user manager class has a lot of methods, but there are three that are used to manage the stored data, as described in Table 7-3. These methods are all asynchronous, as are many of the methods defined by the Identity API.

Table 7-3. *The UserManager<IdentityUser> Methods for Managing Stored Data*

Name	Description
`CreateAsync(user)`	This method stores a new instance of the user class.
`UpdateAsync(user)`	This method updates a stored instance of the user class.
`DeleteAsync(user)`	This method removes a stored instance of the user class.

To store the test `IdentityUser` objects, I call the user manager's `CreateAsync` method.

```
...
foreach (string email in emails) {
    IdentityUser userObject = new IdentityUser {
        UserName = email,
        Email = email,
        EmailConfirmed = true
    };
    await UserManager.CreateAsync(userObject);
}
...
```

The CreateAsync method stores the IdentityUser objects I create, seeding the database with accounts.

Processing Identity Results

My use of the CreateAsync method in Listing 7-15 assumes that everything works as expected, which is a level of optimism that is rarely warranted in software development. The methods in Table 7-3 return IdentityResult objects that describe the outcome of operations using the properties described in Table 7-4.

Table 7-4. *The IdentityResult Properties*

Name	Description
Succeeded	This property returns true if the operation is successful and false otherwise.
Errors	This property returns an IEnumerable<IdentityError> object containing an IdentityError object for each error that has occurred. The IdentityError class defines Code and Description properties.

If the Succeeded property is true, then an operation has worked. If it is false, then the Errors property can be enumerated to understand the problems that have arisen. When writing custom Identity workflows, a common requirement is to handle errors from the IdentityResult object by adding validation errors to the ASP.NET Core model state. Add a class file named IdentityExtensions.cs to the IdentityApp/Pages/Identity folder and use it to define the extension method shown in Listing 7-16.

Listing 7-16. The Contents of the IdentityExtensions.cs File in the IdentityApp/Pages/Identity Folder

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc.ModelBinding;
using System.Linq;

namespace IdentityApp.Pages.Identity {
    public static class IdentityExtensions {

        public static bool Process(this IdentityResult result,
                ModelStateDictionary modelState) {
            foreach (IdentityError err in result.Errors
                    ?? Enumerable.Empty<IdentityError>()) {
                modelState.AddModelError(string.Empty, err.Description);
            }
            return result.Succeeded;
        }
    }
}
```

Each IdentityError object in the sequence returned by the IdentityResult.Errors property defines a Code property and a Description property. The Code property is used to unambiguously identify the error and is intended to be consumed by the application. I am interested in the Description property, which describes an error that can be presented to the user. I use the foreach keyword to add the value from each IdentityError.Description property and add it to the set of validation errors that ASP.NET Core will handle.

```
...
modelState.AddModelError(string.Empty, err.Description);
...
```

In Listing 7-17, I have used the new extension method to process the results from the CreateAsync method.

Listing 7-17. Handling Results in the Dashboard.cshtml.cs File in the Pages/Identity/Admin Folder

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

namespace IdentityApp.Pages.Identity.Admin {

    public class DashboardModel : AdminPageModel {

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

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

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

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

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

This is not the most elegant code because it forces together two different error handling approaches, while also signaling whether there are any errors to handle at all. But it is a helpful approach to ensure that errors are surfaced from the Identity API to the user.

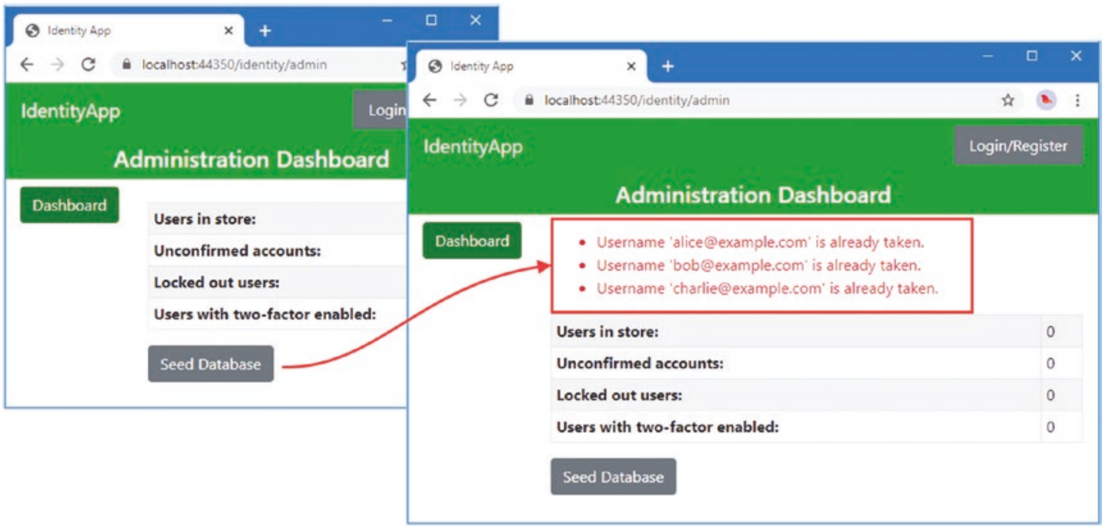
The final step to get the basic features working is to add HTML elements to the view part of the `Dashboard` page, as shown in Listing 7-18.

Listing 7-18. Adding HTML Elements to the Dashboard.cshtml File in the Pages/Identity/Admin Folder

```
@page "/identity/admin"
@model IdentityApp.Pages.Identity.Admin.DashboardModel
@{
    ViewBag.Workflow = "Dashboard";
}

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

<table class="table table-sm table-striped table-bordered">
    <tbody>
        <tr><th>Users in store:</th><td>@Model.UsersCount</td></tr>
        <tr><th>Unconfirmed accounts:</th><td>@Model.UsersUnconfirmed</td></tr>
        <tr><th>Locked out users:</th><td>@Model.UsersLockedout</td></tr>
        <tr>
            <th>Users with two-factor enabled:</th>
            <td>@Model.UsersTwoFactor</td>
        </tr>
    </tbody>
</table>

<form method="post">
    <button class="btn btn-secondary" type="submit">Seed Database</button>
</form>
```

The new elements display validation errors and define a form that is submitted to seed the database.

There is still work to do, but there is enough functionality for a simple test. Restart ASP.NET Core and request `https://localhost:44350/identity/admin`. Click the Seed Database button to create the test accounts. Nothing will happen because the Dashboard page isn't yet reading data from the database, but if you click the button again, you can confirm that the error handling features are working, as shown in Figure 7-3.



- Username 'alice@example.com' is already taken.
- Username 'bob@example.com' is already taken.
- Username 'charlie@example.com' is already taken.

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

Figure 7-3. *Testing the error handling code*

On the second attempt to seed the database, the calls to the `CreateAsync` method produce errors because the first seeding stored `IdentityUser` objects with the same `UserName` values. Identity requires unique usernames and produces error that are displayed using the ASP.NET Core model validation features.

Querying the User Data

The problem with the code in Listing 7-17 is that it doesn't clear out existing data before storing the new `IdentityUser` objects. To provide access to the existing data, the user manager class defines a property named `Users`, which can be used to enumerate the stored `IdentityUser` objects and which can be used with LINQ to perform queries. Table 7-5 describes this property for future quick reference.

Table 7-5. *The UserManager<T> Property for Reading Data*

Name	Description
`Users`	This property returns an `IQueryable<IdentityUser>` object that can be used to enumerate the stored `IdentityUser` objects and can be used with LINQ to perform queries.

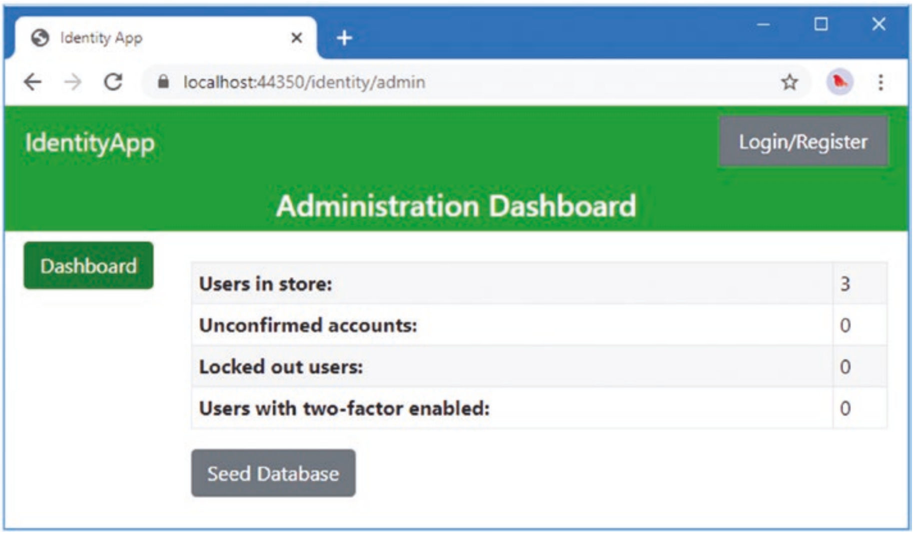
In Listing 7-19, I have added to statements the `Dashboard` page model's POST handler method to enumerate the stored `IdentityUser` objects and pass them to the `DeleteAsync` method, which will remove them from the database. I have also added a GET handler method that sets one of the properties displayed by the view part of the page.

Listing 7-19. Reading/Deleting Data in the Dashboard.cshtml.cs File in the Identity/Pages/Admin Folder

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

namespace IdentityApp.Pages.Identity.Admin {

    public class DashboardModel : AdminPageModel {

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

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

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

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

        public void OnGet() {
            UsersCount = UserManager.Users.Count();
        }

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

The Users property returns an IQueryable<IdentityUser> object that can be enumerated or used in a LINQ query. In the GET handler, I use the LINQ Count method to determine how many IdentityUser objects have been stored. In the POST handler, I use the foreach keyword to enumerate the stored IdentityUser objects so I can delete them. (Notice that I use the ToList method to force evaluation in the foreach loop. This ensures I don’t cause an error by deleting objects from the sequence that I am enumerating.)

Restart ASP.NET Core and request `https://localhost:44350/identity/admin`. The changes in Listing 7-19 display the number of stored `IdentityUser` objects, as shown in Figure 7-4, and clicking the Seed Database button no longer generates errors.



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

Figure 7-4. *Reading (and deleting) stored data*

Displaying a List of Users

There is now some insight into the stored data, but not enough to be useful. It would be helpful to see a list of user accounts, which can be obtained through the user manager's `Users` property. And, since the `Users` property returns an `IQueryable<IdentityUser>` object, it is easy to create a list that can be filtered. Selecting a user account is the first step in many custom Identity workflows, and this is a feature that I will reuse throughout this part of the book. Add a Razor Page named `SelectUser.cshtml` to the `Pages/Identity/Admin` folder with the content shown in Listing 7-20.

Listing 7-20. The Contents of the SelectUser.cshtml File in the Pages/Identity/Admin Folder

```
@page "{label?}/{callback?}"
@model IdentityApp.Pages.Identity.Admin.SelectUserModel
@{
    ViewBag.Workflow = Model.Callback ?? Model.Label ?? "List";
}

<form method="post" class="my-2">
    <div class="form-row">
        <div class="col">
            <div class="input-group">
                <input asp-for="Filter" class="form-control" />
            </div>
        </div>
        <div class="col-auto">
            <button class="btn btn-secondary">Filter</button>
        </div>
    </div>
</form>

<table class="table table-sm table-striped table-bordered">
    <thead>
        <tr>
            <th>User</th>
            @if (!string.IsNullOrEmpty(Model.Callback)) {
                <th/>
            }
        </tr>
    </thead>
    <tbody>
        @if (Model.Users.Count() == 0) {
            <tr><td colspan="2">No matches</td></tr>
        } else {
            @foreach (IdentityUser user in Model.Users) {
                <tr>
                    <td>@user.Email</td>
                    @if (!string.IsNullOrEmpty(Model.Callback)) {
                        <td class="text-center">
                            <a asp-page="@Model.Callback"
                               asp-route-id="@user.Id"
                               class="btn btn-sm btn-secondary">
                                @Model.Callback
                            </a>
                        </td>
                    }
                </tr>
            }
        }
    </tbody>
</table>

@if (!string.IsNullOrEmpty(Model.Callback)) {
    <a asp-page="Dashboard" class="btn btn-secondary">Cancel</a>
}
```

The view part of the page displays a table with a text box that can be used to filter the user store by searching email addresses. When I start using this page as part of a custom workflow, each user will be displayed with a button that requests the page that is specified in the URL, including the `Id` value of the selected user. This will allow users to be selected and trigger a redirection back to the page that handles the workflow.

To define the page model class, add the code shown in Listing 7-21 to the `SelectUser.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 7-21. The Contents of the SelectUser.cshtml.cs File in the Pages/Identity/Admin Folder

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

namespace IdentityApp.Pages.Identity.Admin {

    public class SelectUserModel : AdminPageModel {

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

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

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

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

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

        public void OnGet() {
            Users = UserManager.Users
                .Where(u => Filter == null || u.Email.Contains(Filter))
                .OrderBy(u => u.Email).ToList();
        }

        public IActionResult OnPost() => RedirectToPage(new { Filter, Callback });
    }
}
```

The page model uses the `UserManager<IdentityUser>` object it receives via dependency injection to query the user store with LINQ. I use the `ToList` method to force the evaluation of the query so that I can operate on the results without triggering repeated user store searches.

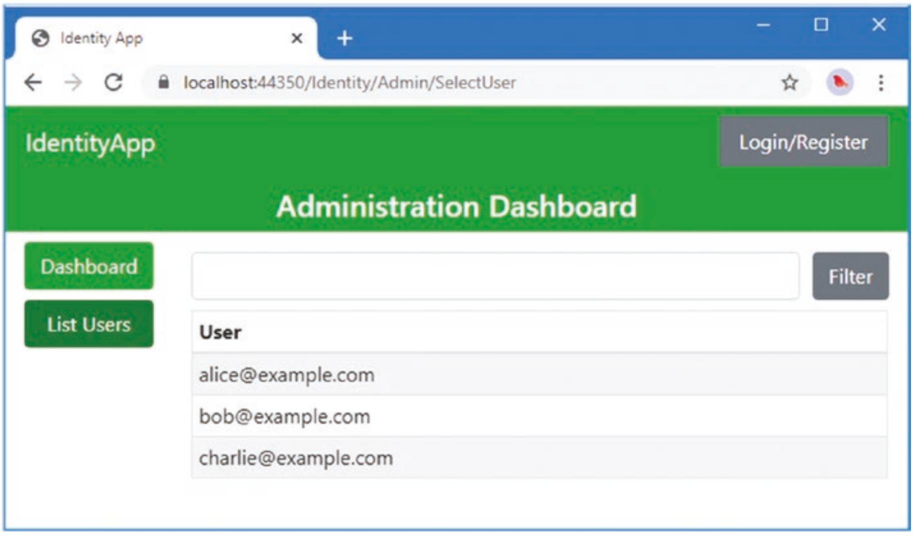
The final step is to add a navigation button for the new Razor Page, as shown in Listing 7-22.

Listing 7-22. Adding Navigation in the _AdminWorkflows.cshtml File in the Pages/Identity/Admin Folder

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

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

```
<a class="btn btn-@Model.theme btn-block @getClass("Dashboard")"
        asp-page="Dashboard">
    Dashboard
</a>
<a class="btn btn-success btn-block @getClass("List")" asp-page="SelectUser">
    List Users
</a>
```

To see the list of users, restart ASP.NET Core, request `https://localhost:44350/Identity/Admin`, and click the List Users button, which will produce the results shown in Figure 7-5. You can enter text into the field and filter the list of users, but the only stored data is the account created by the seeding process.



User
alice@example.com
bob@example.com
charlie@example.com

Figure 7-5. *Displaying a list of stored user accounts*

Viewing and Editing User Details

The `IdentityUser` class defines a set of properties that provide access to the stored data values for a user account. For example, there is an `Email` property that provides access to the user's email address. Some properties can be accessed through methods defined by the `UserManager<IdentityUser>` class so that the `GetEmailAsync` and `SetEmailAsync` methods get and set the value of the `Email` property.

Some properties are read-only, so the user manager class only provides methods that read the property value. For example, the `IdentityUser.Id` property provides access to the unique ID for an `IdentityUser` object that cannot be changed, so there is a `UserManager<IdentityUser>.GetUserIdAsync` method to read the value but no corresponding method to change the value.

Some `IdentityUser` properties do not have corresponding user manager methods at all. The `NormalizedUserName` method, for example, is automatically updated when a new `UserName` value is stored and there are no user manager methods for this property.

Some user manager methods do more than directly update a user object property. The `UserManager<IdentityUser>.SetEmailAsync` method updates the `Identity.Email` property but also sets the `IdentityUser.EmailConfirmed` property to `false`, which indicates that the new email address has not been confirmed.

Working directly with user object properties is the simplest approach and feels natural in C# development, especially when combined with the ASP.NET Core model binding features, which make it easy to receive form data. But the additional work that the user manager methods perform can be useful in ensuring that the data in the user store is consistent. In Part 2 of this book, I describe every `UserManager<T>` method in detail so you can understand when additional work is performed.

For quick reference, Table 7-6 describes the properties defined by the `IdentityUser` class, along with the most important `UserManager<IdentityUser>` methods that relate to them, and provides a reference to the detailed descriptions later in this book. Some of the properties in the table are part of complex features that require additional user manager methods, as noted in the table. You may find the properties and methods approach confusing at first, but it is simpler than it appears and quickly becomes second nature.

Table 7-6. *User Class Properties and User Manager Class Methods*

Property Name	Description	User Manager Methods
`Id`	This property stores the unique ID for the user and cannot be changed.	`GetUserIdAsync`
`UserName`	This property stores the user's username.	`GetUserNameAsync` `SetUserNameAsync`
`NormalizedUserName`	This property stores a normalized representation of the username that is used when searching for users.	This property is updated automatically when the object is stored.
`Email`	This property stores the user's email address.	`GetEmailAsync` `SetEmailAsync`
`NormalizedEmail`	This property stores a normalized representation of the email address that is used when searching for users.	This property is updated automatically when the object is stored.
`EmailConfirmed`	This property indicates whether the user's email address has been confirmed.	`IsEmailConfirmedAsync`, with additional methods described in Chapter 17.
`PasswordHash`	This property stores a hashed representation of the user's password.	`HasPasswordAsync`, with additional methods described in Chapter 18.
`PhoneNumber`	This property stores the user's phone number.	`GetPhoneNumberAsync` `SetPhoneNumberAsync`
`PhoneNumberConfirmed`	This property indicates whether the user's phone number has been confirmed.	`IsPhoneNumberConfirmedAsync`, with additional methods described in Chapter 17.
`TwoFactorEnabled`	This property indicates whether the user has configured two-factor access.	`GetTwoFactorEnabledAsync`, `SetTwoFactorEnabledAsync`, with additional methods described in Chapter 20.
`LockoutEnabled`	This property indicates whether the user account can be locked out following failed sign-ins.	`GetLockoutEnabledAsync` `SetLockoutEnabledAsync` `IsLockedOutAsync`
`AccessFailedCount`	This property is used to keep track of the number of failed sign-ins.	This property is not used directly. See Chapter 20 for details.

(*continued*)

Table 7-6. (*continued*)

Property Name	Description	User Manager Methods
`LockoutEnd`	This property is used to store the time when the account will be permitted to sign in again.	This property is not used directly.
`SecurityStamp`	This property stores a random value that is changed when the user's credentials are updated.	This property is not used directly and is updated automatically.
`ConcurrencyStamp`	This property stores a random value that is updated whenever the user data is stored.	This property is not used directly.

Add a Razor Page named `View.cshtml` to the `Pages/Identity/Admin` folder with the content shown in Listing 7-23.

Listing 7-23. The Contents of the View.cshtml File in the Pages/Identity/Admin Folder

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

<table class="table table-sm table-striped table-bordered">
    <thead>
        <tr><th>Property</th><th>Value</th></tr>
    </thead>
    <tbody>
        @foreach (string name in Model.PropertyNames) {
            <tr>
                <td>@name</td>
                <td class="text-truncate" style="max-width:250px">
                    @Model.GetValue(name)
                </td>
            </tr>
        }
    </tbody>
</table>

<a asp-page="Edit" asp-route-id="@Model.Id" class="btn btn-secondary">
    Edit
</a>
<a asp-page="View" asp-route-id="" class="btn btn-secondary">Back</a>
```

The view part of the page displays a table containing the properties defined by the `IdentityUser` class and the values for the selected object. To implement the page model class, add the code shown in Listing 7-24 to the `View.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 7-24. The Contents of the View.cshtml.cs File in the Pages/Identity/Admin Folder

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

namespace IdentityApp.Pages.Identity.Admin {

    public class ViewModel : AdminPageModel {

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

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

        public IdentityUser IdentityUser { get; set; }

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

        public IEnumerable<string> PropertyNames
            => typeof(IdentityUser).GetProperties()
                .Select(prop => prop.Name);

        public string GetValue(string name) =>
            typeof(IdentityUser).GetProperty(name)
                 .GetValue(IdentityUser)?.ToString();

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

To identify the user account to be edited, the GET handler page will perform a redirection to the `SelectUser` page if the request URL doesn't include an `Id` value. If there is an `Id`, then it is used to search the user store, using one of the methods that the `UserManager<IdentityUser>` class provides, as described in Table 7-7.

Table 7-7. *The UserManager<T> Members for Searching the Store*

Name	Description
`FindByIdAsync(id)`	This method returns an `IdentityUser` object representing the user with the specified unique ID.
`FindByNameAsync(name)`	This method returns an `IdentityUser` object representing the user with the specified name.
`FindByEmailAsync(email)`	This method returns an `IdentityUser` object representing the user with the specified email address.

These methods locate a single `IdentityUser` object using an ID, username, or email address or return `null` if there is no match.

If the GET handler method in Listing 7-24 is invoked by a URL that contains an ID value, then the `FindByIdAsync` method is used to search the user store and assign the result to a property named `IdentityUser` that can be accessed in the view.

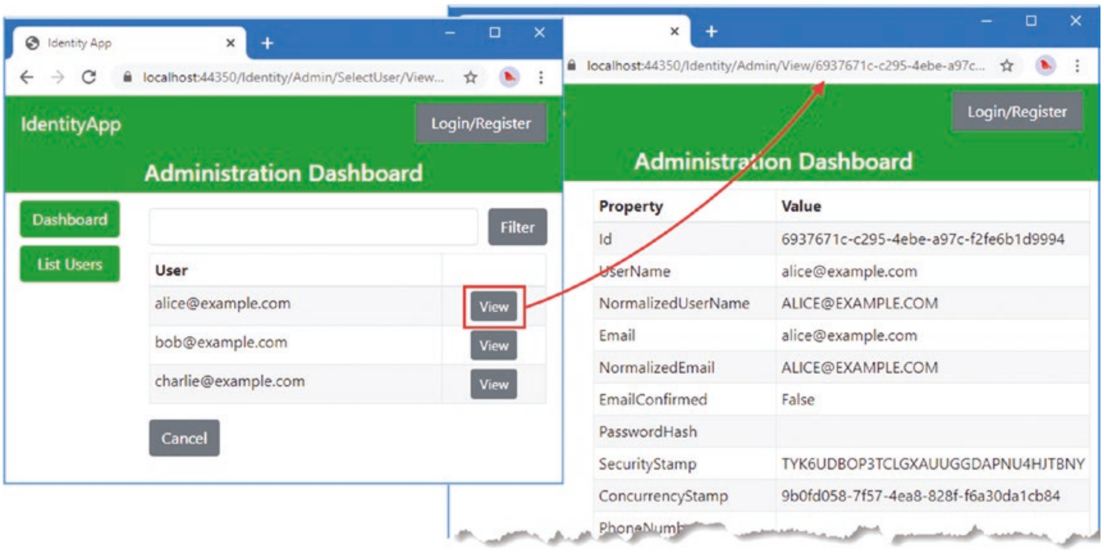
```
...
IdentityUser = await UserManager.FindByIdAsync(Id);
...
```

The list of properties is obtained from the `IdentityUser` type using the .NET reflection feature. In Listing 7-25, I have changed the page selected by the List Users button so that the `View` page is requested.

Listing 7-25. Changing Navigation in the _AdminWorkflows.cshtml File in the Pages/Identity/Admin Folder

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

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

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

Restart ASP.NET Core, request `https://localhost:44350/identity/admin`, and click the List Users button. Click the View button for the `alice@example.com` account, and you will be presented with a table containing the `IdentityUser` properties and values, as shown in Figure 7-6. Some properties are assigned values automatically, while others are not assigned until the feature to which they relate is used.


Identity App
localhost:44350/Identity/Admin/SelectUser/View...
IdentityApp
Login/Register
Administration Dashboard
Dashboard
List Users
Filter
View
Cancel
localhost:44350/Identity/Admin/View/6937671c-c295-4ebe-a97c...

User	
alice@example.com	View
bob@example.com	View
charlie@example.com	View

Property	Value
Id	6937671c-c295-4ebe-a97c-f2fe6b1d9994
UserName	alice@example.com
NormalizedUserName	ALICE@EXAMPLE.COM
Email	alice@example.com
NormalizedEmail	ALICE@EXAMPLE.COM
EmailConfirmed	False
PasswordHash	
SecurityStamp	TYK6UDBOP3TCLGXAUUGGDAPNU4HJTBNY
ConcurrencyStamp	9b0fd058-7f57-4ea8-828f-f6a30da1cb84
PhoneNumb	

Figure 7-6. *Displaying IdentityUser properties*

Editing User Details

The previous examples demonstrated the basic operations for creating, reading, and deleting `IdentityUser` objects, which are three of the four classic data operations. I demonstrate how these features can be used to create custom workflows in later chapters, but there is one additional operation that creates the classic set: performing an update.

Add a Razor Page named `Edit.cshtml` to the `Pages/Identity/Admin` folder with the content shown in Listing 7-26.

Listing 7-26. The Contents of the Edit.cshtml File in the Pages/Identity/Admin Folder

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

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

<form method="post">
    <input type="hidden" asp-for="Id" />
    <div class="form-group">
        <label>Username</label>
        <input class="form-control" asp-for="IdentityUser.UserName" />
    </div>
    <div class="form-group">
        <label>Normalized Username</label>
        <input class="form-control" asp-for="IdentityUser.NormalizedUserName"
            readonly />
    </div>
    <div class="form-group">
        <label>Email</label>
        <input class="form-control" asp-for="IdentityUser.Email" />
    </div>
    <div class="form-group">
        <label>Normalized Email</label>
        <input class="form-control"
            asp-for="IdentityUser.NormalizedEmail" readonly />
    </div>
    <div class="form-group">
        <label>Phone Number</label>
        <input class="form-control" asp-for="IdentityUser.PhoneNumber" />
    </div>
    <div>
        <button type="submit" class="btn btn-success">Save</button>
        <a asp-page="Dashboard" class="btn btn-secondary">Cancel</a>
    </div>
</form>
```

The view part of the page displays an HTML form with fields for editing the `UserName`, `Email`, and `PhoneNumber` properties of an `IdentityUser` object. There is also a hidden value for the `Id` property, so that the user object to be modified can be identified, and read-only fields for the `NormalizedUserName` and `NormalizedEmail` properties, which I have included to demonstrate how some properties are updated automatically.

To create the page model, add the code shown in Listing 7-27 to the `Edit.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 7-27. The Contents of the Edit.cshtml.cs File in the Pages/Identity/Admin Folder

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

namespace IdentityApp.Pages.Identity.Admin {

    public class EditBindingTarget {
        [Required]
        public string Username { get; set; }
        [Required]
        [EmailAddress]
        public string Email { get; set; }
        [Phone]
        public string PhoneNumber { get; set; }
    }

    public class EditModel : AdminPageModel {

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

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

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

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

        public async Task<IActionResult> OnPostAsync(
                [FromForm(Name = "IdentityUser")] EditBindingTarget userData) {
            if (!string.IsNullOrEmpty(Id) && ModelState.IsValid) {
                IdentityUser user = await UserManager.FindByIdAsync(Id);
                if (user != null) {
                    user.UserName = userData.Username;
                    user.Email = userData.Email;
                    user.EmailConfirmed = true;
                    if (!string.IsNullOrEmpty(userData.PhoneNumber)) {
                        user.PhoneNumber = userData.PhoneNumber;
                    }
                }
                IdentityResult result = await UserManager.UpdateAsync(user);
                if (result.Process(ModelState)) {
                    return RedirectToPage();
                }
            }
            IdentityUser = await UserManager.FindByIdAsync(Id);
            return Page();
        }
    }
}
```

When the user submits the form, the ASP.NET Core model binding feature assigns the form values to the properties to an instance of the EditBindingTarget class, which is defined as follows:

```
...
public class EditBindingTarget {
    [Required]
    public string Username { get; set; }
    [Required]
    [EmailAddress]
    public string Email { get; set; }
    [Phone]
    public string PhoneNumber { get; set; }
}
...
```

As I demonstrate in Part 2, you can use model binding directly with an instance of the user class, but this approach allows me to be selective about the fields that I am interested in and apply validation attributes without needing to subclass IdentityUser or declare and decorate individual handler method parameters.

The POST handler method uses the Id property, which is set using model binding, to search the user store with the FindByIdAsync and update the properties of the resulting IdentityUser object with the form data values, like this:

```
...
IdentityUser user = await UserManager.FindByIdAsync(Id);
if (user != null) {
    user.UserName = userData.Username;
    user.Email = userData.Email;
    user.EmailConfirmed = true;
    if (!string.IsNullOrEmpty(userData.PhoneNumber)) {
        user.PhoneNumber = userData.PhoneNumber;
    }
}
...
```

I could have used the methods provided by the UserManager<IdentityUser> class, but, as I explained earlier, I generally prefer to update IdentityUser properties directly.

Notice that I set the EmailConfirmed property to true in Listing 7-28. As a general rule, email addresses set by administrators should not require the user to go through the confirmation process, so I explicitly set the EmailConfirmed property. I create a workflow that does require user confirmation for email addresses in Chapter 9.

Changes to an IdentityUser object are not added to the store until the user manager's UpdateAsync method is called, like this:

```
...
IdentityResult result = await UserManager.UpdateAsync(user);
...
```

This method updates the user store and returns an IdentityResult object. To integrate the new workflow, add the element shown in Listing 7-28 to the navigation partial view.

Listing 7-28. Adding Navigation in the _AdminWorkflows.cshtml File in the Pages/Identity/Admin Folder

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

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

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

```
<a class="btn btn-success btn-block @getClass("List")" asp-page="View"
        asp-route-id="">
    List Users
</a>
<a class="btn btn-success btn-block @getClass("Edit")" asp-page="Edit"
        asp-route-id="">
    Edit Users
</a>
```

Restart ASP.NET Core, request `https://localhost:44350/Identity/Admin`, click Edit Users, and click the Edit button for the `alice@example.com` account. Change the username to `alice.smith@example.com` and click the Save button. The user store will be updated, as shown in Figure 7-7.

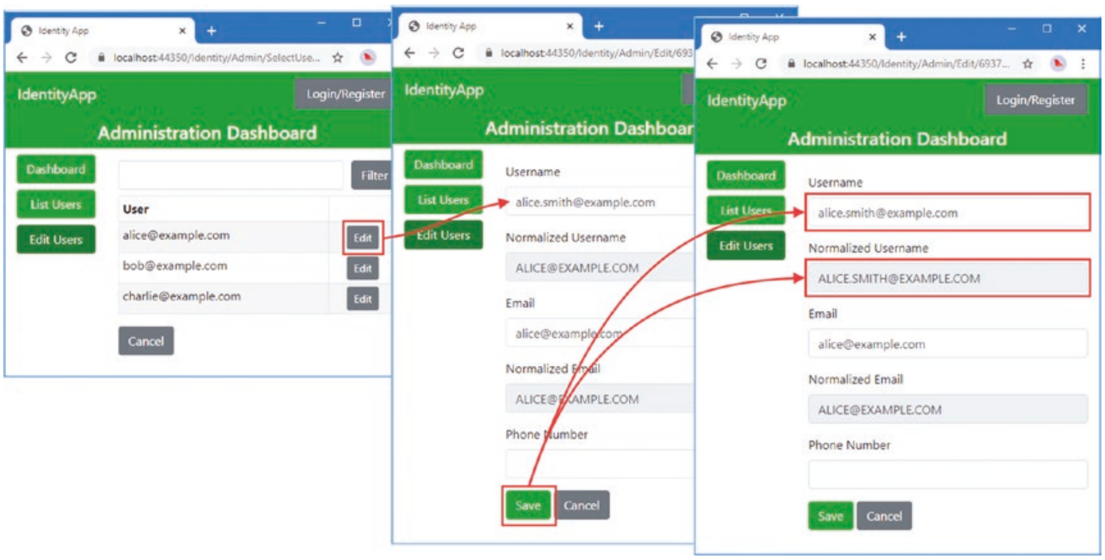
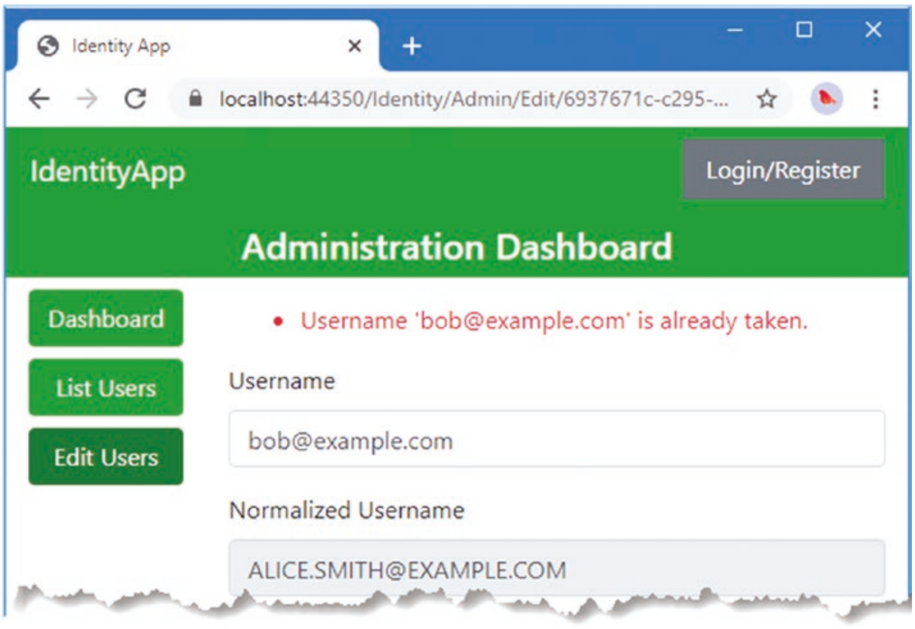


Figure 7-7. *Editing User Details*

Notice that the normalized username property is automatically updated. This is done by the user manager's `UpdateAsync` method, and it helps ensure consistency in the user store.

The normalization process is described in detail in Part 2, but what's important for this chapter is that it is updated automatically when data is stored.

The `UpdateAsyc` method also performs validation, which you can see by editing the Alice account and entering **bob@example.com** into the Username field. When you click the Save button, you will see the error message shown in Figure 7-8, which was included in the `IdentityResult` object returned by the `UpdateAsync` method. The validation process is described in Part 2.



• Username 'bob@example.com' is already taken.

Username

bob@example.com

Normalized Username

ALICE.SMITH@EXAMPLE.COM

Figure 7-8. *Validating user details*

Fixing the Username and Email Problem

Identity supports different values for a user's username and email address, which are stored using the IdentityUser class UserName and Email properties. This means a user can sign in with a username that is not their email address, which is a common approach taken in corporate environments, especially in larger companies where email is ring-fenced from other services.

For other applications, it makes more sense to use the email address as the username and keep both IdentityUser fields synchronized, which is also the approach taken by the Identity UI package.

In Listing 7-29, I have updated the view part of the Edit page so that the Username text field to readonly. There is no need to display both UserName and Email fields when they are being updated with the same values, but I like to see the change happen.

Listing 7-29. Making a Field Read-Only in the Edit.cshtml File in the Pages/Identity/Admin Folder

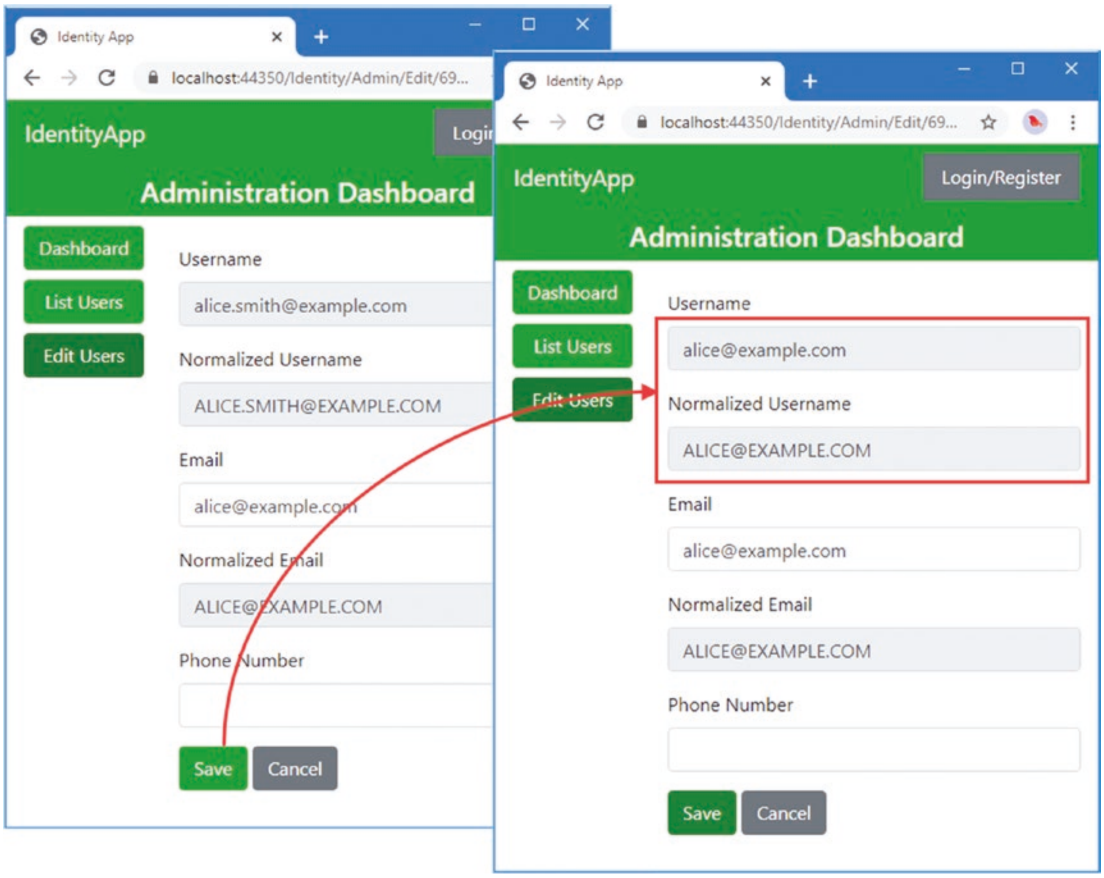
```
@page "{Id?}"
@model IdentityApp.Pages.Identity.Admin.EditModel
@{
    ViewBag.Workflow = "Edit";
}

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

<form method="post">
    <input type="hidden" asp-for="Id" />
    <div class="form-group">
        <label>Username</label>
        <input class="form-control" asp-for="IdentityUser.UserName" readonly />
    </div>
    <div class="form-group">
        <label>Normalized Username</label>
        <input class="form-control" asp-for="IdentityUser.NormalizedUserName"
            readonly />
    </div>
    <div class="form-group">
        <label>Email</label>
        <input class="form-control" asp-for="IdentityUser.Email" />
    </div>
    <div class="form-group">
        <label>Normalized Email</label>
        <input class="form-control"
            asp-for="IdentityUser.NormalizedEmail" readonly />
    </div>
    <div class="form-group">
        <label>Phone Number</label>
        <input class="form-control" asp-for="IdentityUser.PhoneNumber" />
    </div>
    <div>
        <button type="submit" class="btn btn-success">Save</button>
        <a asp-page="Dashboard" class="btn btn-secondary">Cancel</a>
    </div>
</form>
```

In Listing 7-30, I have modified the statements that apply the form data to the IdentityUser object so that the Email value is used for the Username property.

Listing 7-30. Setting Properties in the Edit.cshtml.cs File in the Pages/Identity/Admin Folder

```
...
public async Task<IActionResult> OnPostAsync(
        [FromForm(Name = "IdentityUser")] EditBindingTarget userData) {
    if (!string.IsNullOrEmpty(Id) && ModelState.IsValid) {
        IdentityUser user = await UserManager.FindByIdAsync(Id);
        if (user != null) {
            user.UserName = userData.Email;
            user.Email = userData.Email;
            user.EmailConfirmed = true;
            if (!string.IsNullOrEmpty(userData.PhoneNumber)) {
                user.PhoneNumber = userData.PhoneNumber;
            }
        }
        IdentityResult result = await UserManager.UpdateAsync(user);
        if (result.Process(ModelState)) {
            return RedirectToPage();
        }
    }
    IdentityUser = await UserManager.FindByIdAsync(Id);
    return Page();
}
...
```

Restart ASP.NET Core, request `https://localhost:44350/Identity/Admin`, click Edit Users, and click the Edit button for the Alice account. Click the Save button, and the username and email address will be updated to the same value, as shown in Figure 7-9. Notice that the normalized username is also updated.



Username: alice.smith@example.com
Normalized Username: ALICE.SMITH@EXAMPLE.COM
Email: alice@example.com
Normalized Email: ALICE@EXAMPLE.COM
Phone Number:

Username: alice@example.com
Normalized Username: ALICE@EXAMPLE.COM
Email: alice@example.com
Normalized Email: ALICE@EXAMPLE.COM
Phone Number:

***Figure 7-9.** Updating the username and email address consistently*

Understanding the User Store

The `UserManager<IdentityUser>` class doesn't store data itself. Instead, it depends on dependency injection to obtain an implementation of the `IUserStore<IdentityUser>` interface. The interface defines the operations required to store and retrieve `IdentityUser` data, and there are additional interfaces that can be implemented by user stores that support additional features. In Part 2, I create a custom user store and explain how each method and optional interface is implemented, but that is a level of detail that is rarely required in most projects.

Instead, you just have to know how to set up the user store to understand which features it implements. The user store in the example application is the one that Microsoft provides for storing data in a SQL database using Entity Framework Core.

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

The AddEntityFrameworkStores method sets up the user store, and the generic type argument specifies the Entity Framework Core context that will be used to access the database.

Separating the user manager from the user store means it is relatively simple to change the user store if the needs of the project change. As long as the new user store can work with your chosen user class, there should be little difficulty in moving from one user store to another.

Caution is required because not all user stores support all of the Identity features, which is indicated by the set of optional interfaces that the user store has implemented. It is important to check that all the features you require are supported when you start a new project and when you change user store.

You don't have to inspect the user store directly because the user manager class defines a set of properties you can read to check feature support, as described in Table 7-8. You will see these features used in the workflows created in this part of the book and described in detail in Part 2.

Table 7-8. *The User Manager Properties for Checking User Store Capabilities*

Name	Description
SupportsQueryableUsers	This property returns true if the user store supports queries via LINQ. The query feature is demonstrated in the querying the user data section and described in depth in Part 2.
SupportsUserEmail	This property returns true if the user store supports email addresses, allows searching the store using an email address, and can keep track of whether an email address has been confirmed. (The actual confirmation process is handled largely by the user manager class, as described in Part 2.)
SupportsUserPhoneNumber	This property returns true if the user store supports phone numbers and keeps track of whether a phone number has been confirmed. (The actual confirmation process is handled largely by the user manager class, as described in Part 2.)
SupportsUserPassword	This property returns true if the user store supports hashed passwords for a user. The management of passwords is demonstrated in Chapter 8 and described in depth in Part 2.
SupportsUserRole	This property returns true if the user store supports roles. The roles feature is demonstrated in Chapter 10 and described in depth in Part 2.
SupportsUserClaim	This property returns true if the user store supports claims. The claims feature is demonstrated in Chapter 10 and described in depth in Part 2.

(*continued*)

Table 7-8. *(continued)*

Name	Description
`SupportsUserLockout`	This property returns `true` if the user store supports user lockouts. This feature is demonstrated in Chapter 8 and described in depth in Part 2.
`SupportsUserTwoFactor`	This property returns `true` if the user store can keep track of whether a user requires two-factor authentication. This feature is demonstrated in Chapter 11 and described in depth in Part 2.
`SupportsUserTwoFactorRecoveryCodes`	This property returns `true` if the store can manage recovery codes, which allow two-factor authentication to be bypassed. This feature is described in depth in Part 2.
`SupportsUserLogin`	This property returns `true` if the store can manage details of signing in with external authentication services. This feature is demonstrated in Chapter 11 and described in depth in Part 2.
`SupportsUserAuthenticatorKey`	This property returns `true` if the store can manage keys for signing in with an authenticator app. This feature is demonstrated in Chapter 11 and described in depth in Part 2.
`SupportsUserAuthenticationTokens`	This property returns `true` if the store can manage tokens used to access APIs provided by third parties, as described in Part 2.
`SupportsUserSecurityStamp`	This property returns true if the store can manage security stamps, which are random values that change when the user's credentials are altered. Security stamps are used indirectly by many Identity workflows and are described in detail in Part 2.

This is a long list of features, but you will understand all of them in detail by the end of this book, and not all of them are required by every application. It is always a good idea to confirm you have access to the features you expect because the user manager class will throw exceptions if you try to perform an operation that relies on a feature that your user store does not support.

Add a Razor Page named `Features.cshtml` to the `Pages/Identity/Admin` folder with the content shown in Listing 7-31.

Listing 7-31. The Contents of the Features.cshtml File in the Pages/Identity/Admin Folder

```
@page
@model IdentityApp.Pages.Identity.Admin.FeaturesModel
@inject UserManager<IdentityUser> UserManager
@{
    ViewBag.Workflow = "Features";
}

<table class="table table-sm table-striped table-bordered">
    <thead><tr><th>Property</th><th>Supported</th></tr></thead>
    <tbody>
        @foreach ((string prop, string val) in Model.Features) {
            <tr>
                <td>@prop</td>
                <td class="@(val == "True" ? "bg-success" : "bg-danger") text-white">
```

```
                    @val
                </td>
            </tr>
        }
    </tbody>
</table>
```

The view section of the page displays a table that is populated with a set of tuples that contain each feature property and its value. To implement the page model class, add the code shown in Listing 7-32 to the `Features.cshtml.cs` file. (You will have to create this file if you are using Visual Studio Code.)

Listing 7-32. The Contents of the Features.cshtml.cs File in the Pages/Identity/Admin Folder

```
using Microsoft.AspNetCore.Identity;
using System.Collections.Generic;
using System.Linq;

namespace IdentityApp.Pages.Identity.Admin {
    public class FeaturesModel : AdminPageModel {

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

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

        public IEnumerable<(string, string)> Features { get; set; }

        public void OnGet() {
            Features = UserManager.GetType().GetProperties()
                .Where(prop => prop.Name.StartsWith("Supports"))
                .OrderBy(p => p.Name)
                .Select(prop => (prop.Name, prop.GetValue(UserManager)
                .ToString()));
        }
    }
}
```

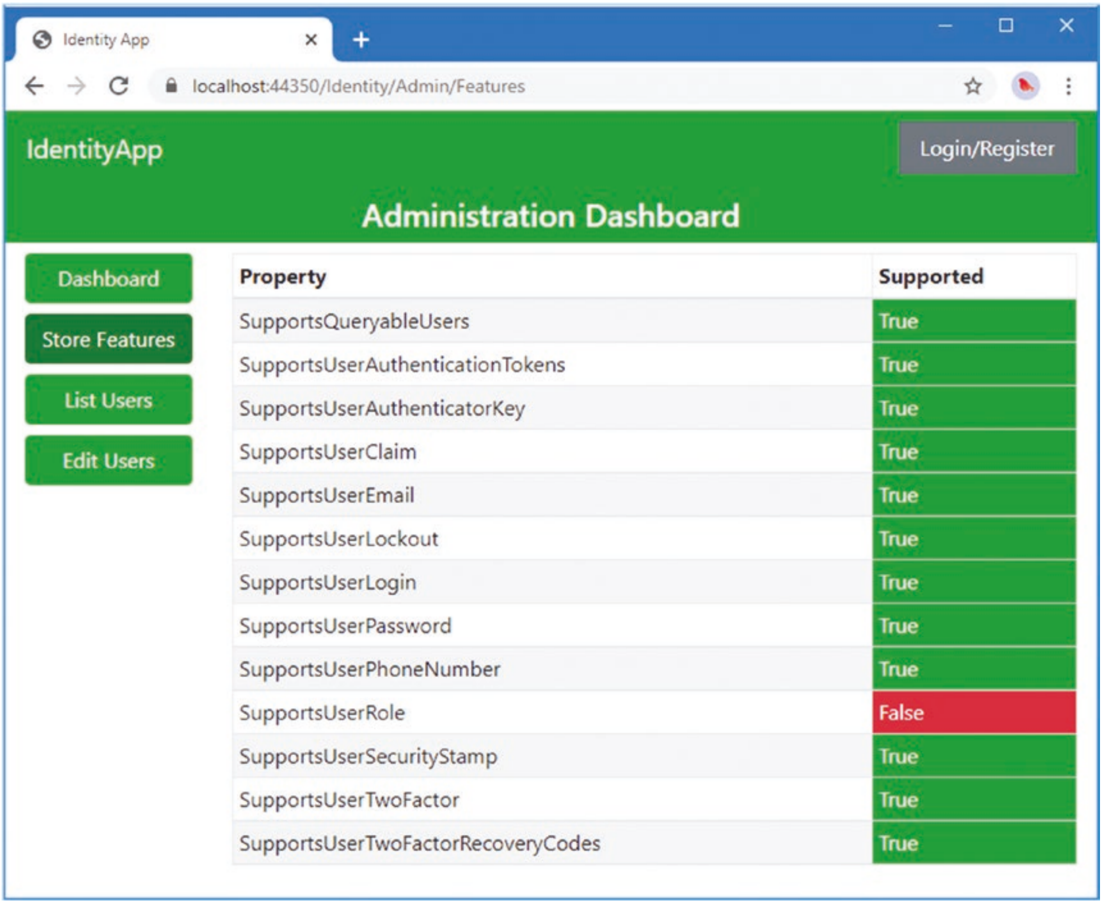
The page model declares a dependency on the `UserManager<IdentityUser>` class by defining a constructor parameter, which is resolved by the ASP.NET Core dependency injection feature. This allows me to use .NET reflection and LINQ to get the set of properties whose names start with `Supports` and create a sequence of tuples containing their names and values. The final step is to add a navigation button for the new page, as shown in Listing 7-33.

Listing 7-33. Adding a Button in the _AdminWorkflows.cshtml File in the Pages/Identity/Admin Folder

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

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

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

Restart ASP.NET Core, request `https://localhost:44350/Identity/Admin`, and click the Store Features button, which will produce the output shown in Figure 7-10.



Property	Supported
SupportsQueryableUsers	True
SupportsUserAuthenticationTokens	True
SupportsUserAuthenticatorKey	True
SupportsUserClaim	True
SupportsUserEmail	True
SupportsUserLockout	True
SupportsUserLogin	True
SupportsUserPassword	True
SupportsUserPhoneNumber	True
SupportsUserRole	False
SupportsUserSecurityStamp	True
SupportsUserTwoFactor	True
SupportsUserTwoFactorRecoveryCodes	True

***Figure 7-10.** Checking the features supported by the user store*

Changing the Identity Configuration

You will notice that the user store doesn't support roles. This is because the Identity UI package doesn't support roles, and the method used to set up Identity and Identity UI doesn't include the configuration information for role support.

I am going to need the role feature in Chapter 10, so I have changed the method used to set up Identity in the `Startup` class, as shown in Listing 7-34, which also has the effect of disabling the Identity UI package.

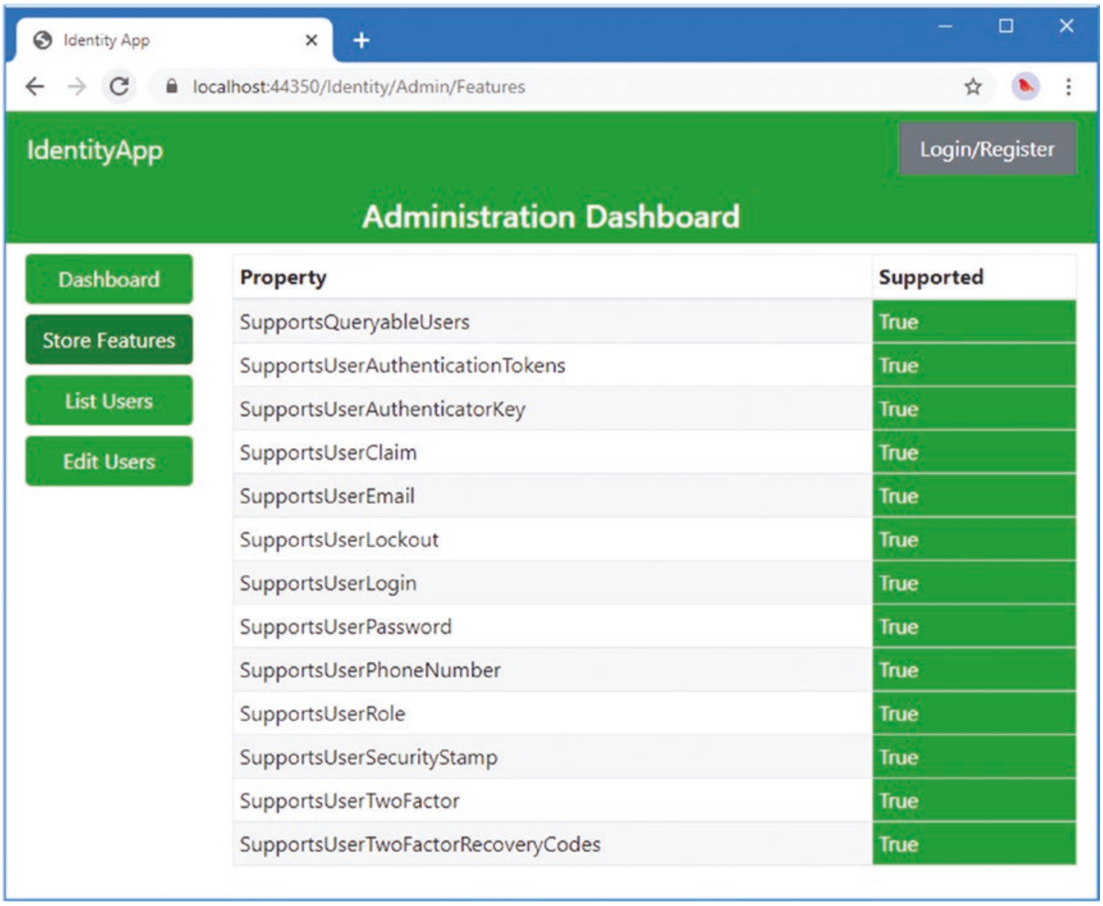
Listing 7-34. Changing the Identity Configuration in the Startup.cs File in the IdentityApp Folder

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

The user store set up by the `AddEntityFrameworkStores` method does support roles but only when a role class has been selected, which isn't possible with the `AddDefaultIdentity` method used previously.

I have replaced the `AddDefaultIdentity` method the `AddIdentity` method. The `AddIdentity` method defines an additional generic type parameter that is used to specify the role class, which enables role support in the user store.

Restart ASP.NET Core, request `https://localhost:44350/Identity/Admin`, and click the Store Features button, and you will see that the user store now supports all of the Identity features, as shown in Figure 7-11.



Property	Supported
SupportsQueryableUsers	True
SupportsUserAuthenticationTokens	True
SupportsUserAuthenticatorKey	True
SupportsUserClaim	True
SupportsUserEmail	True
SupportsUserLockout	True
SupportsUserLogin	True
SupportsUserPassword	True
SupportsUserPhoneNumber	True
SupportsUserRole	True
SupportsUserSecurityStamp	True
SupportsUserTwoFactor	True
SupportsUserTwoFactorRecoveryCodes	True

Figure 7-11. *Enabling user store features*

Summary

In this chapter, I started to describe the Identity API and used its basic features to lay the foundation for future chapters by creating administrator and self-service dashboards. In the next chapter, I create workflows for signing into the application and managing passwords.