CHAPTER 18

Signing In with Identity

In this chapter, I extend the user store to support passwords and create the services that support user sign-ins. I explain how passwords are stored, how they are validated, and how the entire process can be customized. Table 18-1 puts the features described in this chapter in context.

Table 18-1. *Putting Signing In in Context*

Question	Answer
What is it?	Signing in is the process by which the user identifies themselves to the application. Passwords are the conventional means with which users sign in and that prove their identity because they are the only person who knows the password.
Why is it useful?	Signing in generates a token, usually a cookie, that is included in subsequent requests so that the user can interact with the application without needing to provide further proof of their identity.
How is it used?	The user store is extended to store passwords, and a service is defined that hashes passwords so they can be stored securely. As part of the sign-in process, the data in the user store is transformed into a series of claims that are provided to ASP.NET Core through a `ClaimsPrincipal` object.
Are there any pitfalls or limitations?	Users tend to pick poor passwords and will subvert onerous password policies. Applications that manage sensitive data or operations should combine passwords with additional forms of identification, as described in Chapters 20 and 21.
Are there any alternatives?	Identity can be configured to support external authentication where responsibility for identifying users is delegated to a third party. See Chapters 22 and 23.

Table 18-2 summarizes the chapter.

A. Freeman, *Pro ASP.NET Core Identity*, https://doi.org/10.1007/978-1-4842-6858-2_18

Table 18-2. *Chapter Summary*

Problem	Solution	Listing
Provide ASP.NET Core with the data in the user store when a user signs in	Create an implementation of the `IUserClaimsPrincipalFactory<T>` interface.	2
Sign users into an application	Use the `SignInManager<T>` class.	3–4, 10, 11
Stwore passwords in the user store	Implement the `IUserPasswordStore<T>` interface.	5, 7–9
Change or recovery passwords	Use the methods provided by the `UserManager<T>` class.	12–19,23–25
Restrict the passwords that a user can choose	Create an implementation of the `IPasswordValidator<T>` interface.	20–22

Preparing for This Chapter

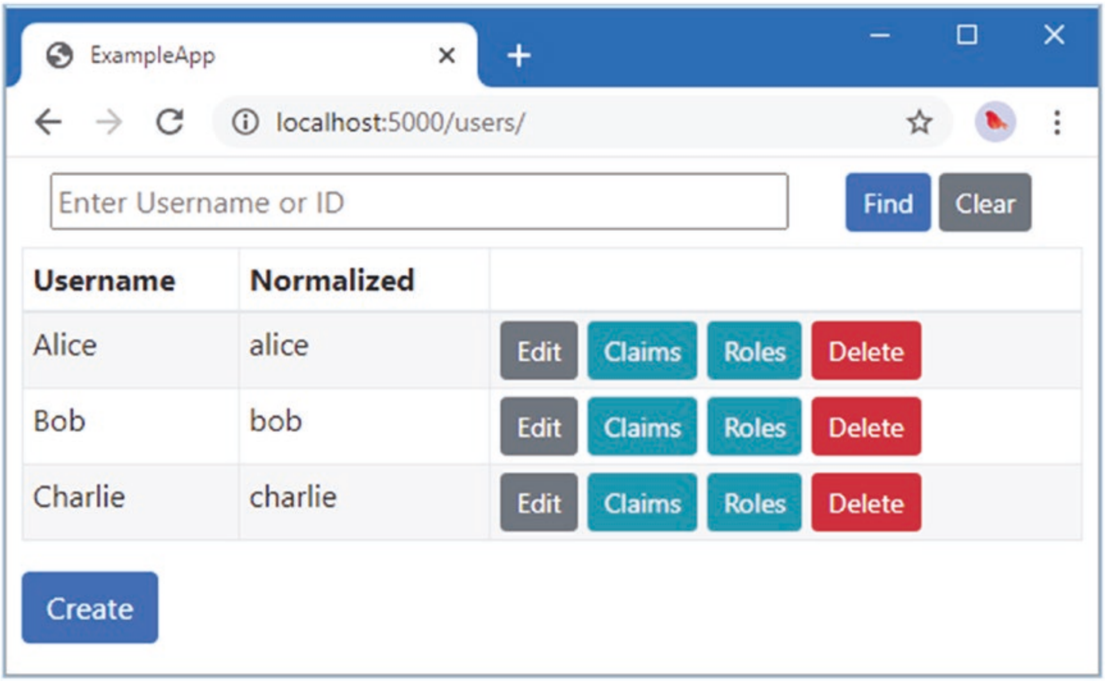
This chapter uses the `ExampleApp` project from Chapter 17. No changes are required to prepare for this chapter. Open a new command prompt, navigate to the `ExampleApp` folder, and run the command shown in Listing 18-1 to start ASP.NET Core.

■ **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 18-1. Running the Example Application

```
dotnet run
```

Open a new browser window and request `http://localhost:5000/users`. You will be presented with the user data shown in Figure 18-1. The data is stored only in memory, and changes will be lost when ASP.NET Core is stopped.



Username	Normalized	
Alice	alice	Edit Claims Roles Delete
Bob	bob	Edit Claims Roles Delete
Charlie	charlie	Edit Claims Roles Delete

Figure 18-1. *Running the example application*

Signing Users In

The user store has taken shape, along with the tools that use it through the `UserManager<T>` class. In this section, I am going to demonstrate how Identity is used to sign users into the application and, once signed in, how requests are authenticated.

Creating the Claims Principal Factory

The `AppUser` user class is currently unrelated to the `ClaimsPrincipal` objects that represented users in earlier chapters. To bridge between these two ways of representing users, Identity uses the `IUserClaimsPrincipal Factory<T>` interface, where `T` is the user class, which defines the single method described in Table 18-3.

Table 18-3. *The IUserClaimsPrincipalFactory<T> Method*

Name	Description
`CreateAsync(user)`	This method creates a `ClaimsPrincipal` object that represents the specified user.

The flexibility that is available in the implementation of the user store is possible because of this interface, which connects the bespoke user class and its many features with the classes that ASP.NET Core uses for signing in, authentication, and authorization. Identity includes a default implementation of this interface, which creates a `ClaimsPrincipal` object using the `UserManager<T>` methods.

To create an implementation of the IUserClaimsPrincipalFactory<T> interface that is tailored to the AppUser class, add a class file named AppUserClaimsPrincipalFactory.cs to the ExampleApp/Identity folder and use it to define the class shown in Listing 18-2.

Listing 18-2. The Contents of the AppUserClaimsPrincipalFactory.cs File in the Identity Folder

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

namespace ExampleApp.Identity {
    public class AppUserClaimsPrincipalFactory :
        IUserClaimsPrincipalFactory<AppUser> {

        public Task<ClaimsPrincipal> CreateAsync(AppUser user) {
            ClaimsIdentity identity
                = new ClaimsIdentity(IdentityConstants.ApplicationScheme);
            identity.AddClaims(new [] {
                new Claim(ClaimTypes.NameIdentifier, user.Id),
                new Claim(ClaimTypes.Name, user.UserName),
                new Claim(ClaimTypes.Email, user.EmailAddress)
            });
            if (!string.IsNullOrEmpty(user.Hobby)) {
                identity.AddClaim(new Claim("Hobby", user.Hobby));
            }
            if (!string.IsNullOrEmpty(user.FavoriteFood)) {
                identity.AddClaim(new Claim("FavoriteFood", user.FavoriteFood));
            }
            if (user.Claims != null) {
                identity.AddClaims(user.Claims);
            }
            return Task.FromResult(new ClaimsPrincipal(identity));
        }
    }
}
```

The class creates a ClaimsIdentity object that is populated with claims from the AppUser Id, UserName, EmailAddress, Hobby, and FavoriteFood properties, as well as any additional Claim objects that have been stored for the user.

■ **Tip** Notice that I have used the IdentityConstants.ApplicationScheme value when creating the ClaimsIdentity object in Listing 18-2. This scheme name indicates that a user has been authenticated by Identity and is expected by some Identity features. I describe additional IdentityConstants values in later chapters.

GETTING USER OBJECTS FROM CLAIMSPRINCIPAL OBJECTS

The `UserManager<T>` class defines the `GetUserAsync` method, which accepts a `ClaimsPrincipal` object and returns the associated user object. It does this by using the value of the `NameIdentifier` claim as the argument to the `FindByIdAsync` method, querying the user store. The method returns `null` if the `ClaimsPrincipal` object doesn't have a `NameIdentifier` claim or if the user store doesn't contain a user object for the claim value.

Signing Users In

Identity provides the `SignInManager<T>` class to handle signing users into an application. Table 18-4 describes the basic methods provided by the `SignInManager<T>` class that are used to sign users in and out. I describe further methods as I explain more advanced features.

Table 18-4. *The Basic SignInManager<T> Methods*

Name	Description
`SignInAsync(user, persistent)`	This method signs the specified user into the application. The persistent argument specifies whether the authentication cookie will persist after the browser has closed.
`SignInWithClaimsAsync(user, persistent, claims)`	This method signs the user into the application and adds the specified claims to the principal identity.
`SignOutAsync()`	This method signs out the user.

In Listing 18-3, I have updated the page model for the `SignIn` Razor Page to use the `UserManager<T>` and `SignInManager<T>` classes.

Listing 18-3. Using Identity in the SignIn.cshtml.cs File in the Pages Folder

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using System.Linq;
using ExampleApp.Identity;

namespace ExampleApp.Pages {
    public class SignInModel : PageModel {

        public SignInModel(UserManager<AppUser> userManager,
                SignInManager<AppUser> signInManager) {
```

```
            UserManager = userManager;
            SignInManager = signInManager;
        }

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

        public SelectList Users => new SelectList(
            UserManager.Users.OrderBy(u => u.EmailAddress),
                "EmailAddress", "EmailAddress");

        public string Username { get; set; }

        public int? Code { get; set; }

        public void OnGet(int? code) {
            Code = code;
            Username = User.Identity.Name ?? "(No Signed In User)";
        }

        public async Task<ActionResult> OnPost(string username,
                [FromQuery]string returnUrl) {
            //Claim claim = new Claim(ClaimTypes.Name, username);
            //ClaimsIdentity ident = new ClaimsIdentity("simpleform");
            //ident.AddClaim(claim);
            //await HttpContext.SignInAsync(new ClaimsPrincipal(ident));
            AppUser user = await UserManager.FindByEmailAsync(username);
            await SignInManager.SignInAsync(user, false);
            return Redirect(returnUrl ?? "/signin");
        }
    }
}
```

Identity is designed to integrate easily into ASP.NET Core. I have added a constructor to receive UserManager<T> and SignInManager<T> objects through dependency injection. This allows me to present a list of users, which I have listed using their email address, just to demonstrate that the data is coming from the store. Once a user has been selected, I get an AppUser object through the UserManager<T>.FindByEmailAsync method and sign that user into the application using the SignInManager<T>.SignInAsync method.

Configuring the Application

The final step is to register the claims principal factory class so that it will be used by Identity and set up the SignInManager<T> service, as shown in Listing 18-4.

Listing 18-4. Configuring the Application in the Startup.cs File in the ExampleApp Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using ExampleApp.Custom;
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using ExampleApp.Identity;
using ExampleApp.Identity.Store;
using ExampleApp.Services;

namespace ExampleApp {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<ILookupNormalizer, Normalizer>();
            services.AddSingleton<IUserStore<AppUser>, UserStore>();
            services.AddSingleton<IEmailSender, ConsoleEmailSender>();
            services.AddSingleton<ISMSSender, ConsoleSMSSender>();
            services.AddSingleton<IUserClaimsPrincipalFactory<AppUser>,
                AppUserClaimsPrincipalFactory>();

            services.AddIdentityCore<AppUser>(opts => {
                opts.Tokens.EmailConfirmationTokenProvider = "SimpleEmail";
                opts.Tokens.ChangeEmailTokenProvider = "SimpleEmail";
            })
           .AddTokenProvider<EmailConfirmationTokenGenerator>("SimpleEmail")
           .AddTokenProvider<PhoneConfirmationTokenGenerator>
                (TokenOptions.DefaultPhoneProvider)
           .AddSignInManager();

            services.AddSingleton<IUserValidator<AppUser>, EmailValidator>();

            services.AddAuthentication(opts => {
                opts.DefaultScheme = IdentityConstants.ApplicationScheme;
            }).AddCookie(IdentityConstants.ApplicationScheme, opts => {
                opts.LoginPath = "/signin";
                opts.AccessDeniedPath = "/signin/403";
            });
            services.AddAuthorization(opts => {
                AuthorizationPolicies.AddPolicies(opts);
            });
            services.AddRazorPages();
            services.AddControllersWithViews();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

            app.UseStaticFiles();
            app.UseAuthentication();
            app.UseRouting();
            //app.UseMiddleware<RoleMemberships>();
            app.UseAuthorization();
            app.UseEndpoints(endpoints => {
                endpoints.MapRazorPages();
                endpoints.MapDefaultControllerRoute();
                endpoints.MapFallbackToPage("/Secret");
            });
        }
    }
}
```

I used the `AddSingleton` method to register the `AppUserClaimsPrincipalFactory` class as the implementation to be used for the `IUserClaimsPrincipalFactory<AppUser>` interface.

The `AddSignInManager` method is added to the chain of methods used to set up Identity and sets up the `SignInManager<T>` service. The other changes alter the name of the authentication scheme, reflecting the use of Identity to sign users into the application and authenticate requests.

I have also commented out the statement that applies the custom role middleware so that the only source of claims and roles is the user store.

Restart ASP.NET Core and request `http://localhost:5000/signin`. Choose `alice@example.com` from the drop-down list and click the Sign In button to sign into the application. Request `http://localhost:5000/secret`, and the authorization policy applied to the Secret Razor Page will be evaluated using the claims generated from the `AppUser` object by the factory class. Alice's claims meet the policy and access is granted, as shown in Figure 18-2.

■ **Tip** If signing in has no effect, you may find that you are already authenticated using a cookie from a previous chapter. If that's the case, request `http://localhost:5000/signout` and click the Sign Out button. This will clear the existing cookie and allow you to sign in again using Identity. If that doesn't work, clear your browser history and try again.

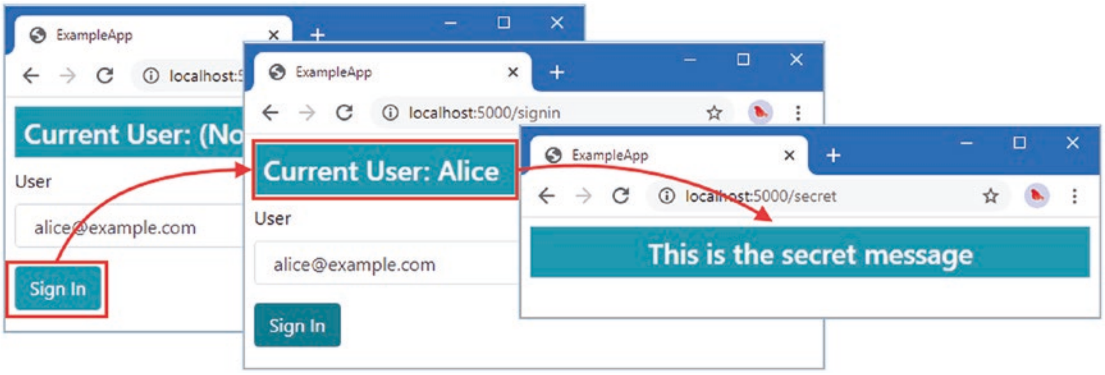


Figure 18-2. *Using Identity to sign in to the application*

Repeat the process and select `bob@example.com` to sign in as Bob. Request `http://localhost:5000/secret`; access will be denied because Bob's claims do not meet the policy requirements, as shown in Figure 18-3.

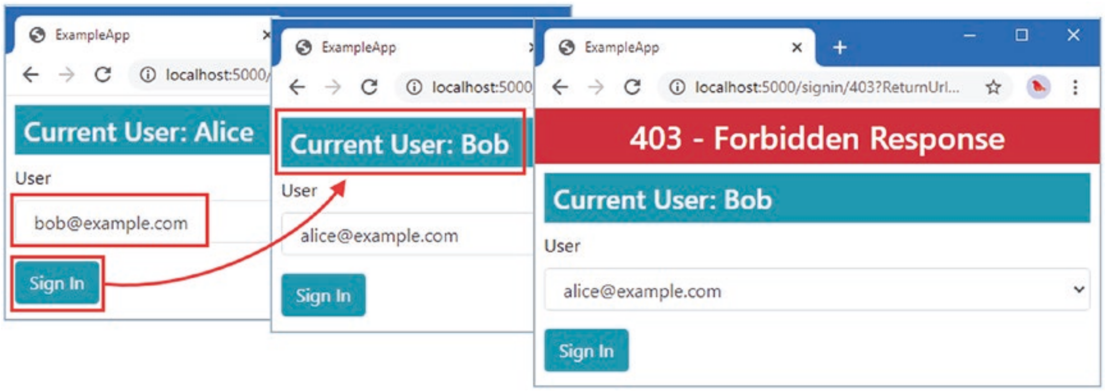


***Figure 18-3.** Enforcing authorization policies in the example application*

Signing In Users with Passwords

Trusting users to pick their account from a drop-down list has been useful, but it is time to introduce passwords. In the sections that follow, I explain how Identity stores and uses passwords and how they are used in the sign-in process.

Updating the User Class

Passwords are not stored directly. Instead, a hash code is created from a user's password and stored as a proxy for the password itself. When the user presents a password during sign-in, a hash code is created from the candidate password and compared to the hash code in the user store. If the user has supplied the right password, the two hash codes will be the same. In Listing 18-5, I have added a property to the AppUser class for a user's password.

Listing 18-5. Adding a Property in the AppUser.cs File in the Identity Folder

```
using System;
using System.Collections.Generic;
using System.Security.Claims;

namespace ExampleApp.Identity {
    public class AppUser {

        public string Id { get; set; } = Guid.NewGuid().ToString();

        public string UserName { get; set; }

        public string NormalizedUserName { get; set; }

        public string EmailAddress { get; set; }
        public string NormalizedEmailAddress { get; set; }
        public bool EmailAddressConfirmed { get; set; }
        public string PhoneNumber { get; set; }
        public bool PhoneNumberConfirmed { get; set; }

        public string FavoriteFood { get; set; }
        public string Hobby { get; set; }

        public IList<Claim> Claims { get; set; }

        public string SecurityStamp { get; set; }
        public string PasswordHash { get; set; }
    }
}
```

There are more complex options for signing in, which I describe in later examples, but for now, a simple string password is a good place to start.

Creating the Password Hasher

Identity relies on implementations of the `IPasswordHasher<T>` interface to generate hash codes from passwords, where `<T>` is the user class. The interface defines the methods shown in Table 18-5.

Table 18-5. *The IPasswordHasher<T> Methods*

Name	Description
`HashPassword(user, password)`	This method returns a hashed representation of a password for the specified user.
`VerifyHashedPassword(user, storedHash, password)`	This method verifies that a password provided by the specified user matches the supplied hashed representation. The result is expressed using a value from the `PasswordVerificationResult` enum: `Failed`, `Success`, or `SuccessRehashNeeded`. (This last value is used to indicate that the algorithm used by the application has changed and the password hash code should be updated.)

To create a custom implementation of the interface, add a class file named `SimplePasswordHasher.cs` to the `ExampleApp/Identity` folder and use it to define the class shown in Listing 18-6.

Listing 18-6. The Contents of the SimplePasswordHasher.cs File in the Identity Folder

```
using Microsoft.AspNetCore.Identity;
using System;
using System.Security.Cryptography;
using System.Text;

namespace ExampleApp.Identity {
    public class SimplePasswordHasher : IPasswordHasher<AppUser> {

        public SimplePasswordHasher(ILookupNormalizer normalizer)
            => Normalizer = normalizer;

        private ILookupNormalizer Normalizer { get; set; }
        public string HashPassword(AppUser user, string password) {
            HMACSHA256 hashAlgorithm =
                new HMACSHA256(Encoding.UTF8.GetBytes(user.Id));
            return BitConverter.ToString(hashAlgorithm.ComputeHash(
                    Encoding.UTF8.GetBytes(password)));
        }

        public PasswordVerificationResult VerifyHashedPassword(AppUser user,
            string storedHash, string password)
                => HashPassword(user, password).Equals(storedHash)
                    ? PasswordVerificationResult.Success
                    : PasswordVerificationResult.Failed;
    }
}
```

The built-in password hasher that Identity provides generates robust hashes from passwords, using a good cryptographic algorithm that is applied carefully and thoroughly. To demonstrate, I have adopted an approach that is simpler and should not be used in real projects. The SimplePasswordHasher class represents passwords by simply combining using the HMACSHA256 hashing algorithm with the user's ID as the key. Listing 18-7 registers the SimplePasswordHasher class, so it will be used to resolve dependencies on the IPasswordHasher<AppUser> interface.

Listing 18-7. Registering a Service in the Startup.cs File in the ExampleApp Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddSingleton<ILookupNormalizer, Normalizer>();
    services.AddSingleton<IUserStore<AppUser>, UserStore>();
    services.AddSingleton<IEmailSender, ConsoleEmailSender>();
    services.AddSingleton<ISMSSender, ConsoleSMSSender>();
    services.AddSingleton<IUserClaimsPrincipalFactory<AppUser>,
        AppUserClaimsPrincipalFactory>();
    services.AddSingleton<IPasswordHasher<AppUser>, SimplePasswordHasher>();

    services.AddIdentityCore<AppUser>(opts => {
        opts.Tokens.EmailConfirmationTokenProvider = "SimpleEmail";
        opts.Tokens.ChangeEmailTokenProvider = "SimpleEmail";
    })
    .AddTokenProvider<EmailConfirmationTokenGenerator>("SimpleEmail")
    .AddTokenProvider<PhoneConfirmationTokenGenerator>
        (TokenOptions.DefaultPhoneProvider)
    .AddSignInManager();

    services.AddSingleton<IUserValidator<AppUser>, EmailValidator>();

    services.AddAuthentication(opts => {
        opts.DefaultScheme = IdentityConstants.ApplicationScheme;
    }).AddCookie(IdentityConstants.ApplicationScheme, opts => {
        opts.LoginPath = "/signin";
        opts.AccessDeniedPath = "/signin/403";
    });
    services.AddAuthorization(opts => {
        AuthorizationPolicies.AddPolicies(opts);
    });
    services.AddRazorPages();
    services.AddControllersWithViews();
}
...
```

Storing Password Hashes in the User Store

When an application manages its own passwords, it needs a user store that implements the `IUserPasswordStore<T>` interface, where `T` is the user class. This interface defines the methods described in Table 18-6. All these methods define the `token` parameter that receives a `CancellationToken` object used to receive notifications when asynchronous operations are canceled.

Table 18-6. *The IUserPasswordStore<T> Methods*

Name	Description
`HasPasswordAsync(user, token)`	This method returns `true` if the specified user has a password.
`GetPasswordHashAsync(user, token)`	This method returns the stored password data for the specified user.
`SetPasswordHashAsync(user, passwordHash, token)`	This method stores a new password hash for the specified user.

To add support for storing passwords, add a class file named `UserStorePasswords.cs` to the `Identity/Store` folder and use it to define the partial class shown in Listing 18-8.

Listing 18-8. The Contents of the UserStorePasswords.cs File in the Identity/Store Folder

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

namespace ExampleApp.Identity.Store {

    public partial class UserStore : IUserPasswordStore<AppUser> {

        public Task<string> GetPasswordHashAsync(AppUser user,
            CancellationToken token) => Task.FromResult(user.PasswordHash);

        public Task<bool> HasPasswordAsync(AppUser user, CancellationToken token)
            => Task.FromResult(!string.IsNullOrEmpty(user.PasswordHash));

        public Task SetPasswordHashAsync(AppUser user, string passwordHash,
                CancellationToken token) {
            user.PasswordHash = passwordHash;
            return Task.CompletedTask;
        }
    }
}
```

The new statements shown in Listing 18-9 add passwords to the seed data in the user store. Since passwords are stored as hashes, I declared a dependency on the password hasher, which I use to generate the stored values.

Listing 18-9. Adding Passwords in the UserStore.cs File in the Identity/Store Folder

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

namespace ExampleApp.Identity.Store {

    public partial class UserStore {

        public ILookupNormalizer Normalizer { get; set; }

        public IPasswordHasher<AppUser> PasswordHasher { get; set; }

        public UserStore(ILookupNormalizer normalizer,
                IPasswordHasher<AppUser> passwordHasher) {
            Normalizer = normalizer;
            PasswordHasher = passwordHasher;
            SeedStore();
        }

        private void SeedStore() {

            var customData = new Dictionary<string, (string food, string hobby)> {
                { "Alice", ("Pizza", "Running") },
                { "Bob", ("Ice Cream", "Cinema") },
                { "Charlie", ("Burgers", "Cooking") }
            };

            int idCounter = 0;

            string EmailFromName(string name) => $"{name.ToLower()}@example.com";

            foreach (string name in UsersAndClaims.Users) {
                AppUser user = new AppUser {
                    Id = (++idCounter).ToString(),
                    UserName = name,
                    NormalizedUserName = Normalizer.NormalizeName(name),
                    EmailAddress = EmailFromName(name),
                    NormalizedEmailAddress =
                        Normalizer.NormalizeEmail(EmailFromName(name)),
                    EmailAddressConfirmed = true,
                    PhoneNumber = "123-4567",
                    PhoneNumberConfirmed = true,
                    FavoriteFood = customData[name].food,
                    Hobby = customData[name].hobby,
                    SecurityStamp = "InitialStamp"
                };
                user.Claims =  UsersAndClaims.UserData[user.UserName]
                    .Select(role => new Claim(ClaimTypes.Role, role)).ToList();
                user.PasswordHash = PasswordHasher.HashPassword(user, "MySecret1$");
                users.TryAdd(user.Id, user);
            }
        }
    }
}
```

For simplicity, I have set all the users' passwords to the same value: MySecret1$. I show you how to create a password change workflow in the "Changing and Recovering Passwords" section.

Signing In to the Application with Passwords

The SignInManager<T> class defines methods that accept a password when signing a user in, as described in Table 18-7.

Table 18-7. *The SignInManager<T> Methods for Signing In with Passwords*

Name	Description
PasswordSignInAsync(user, password, persistent, lockout)	This method signs in the user with the specified password. The persistent argument specifies whether the cookie remains valid when the browser is restarted, and the lockout argument specifies whether failed logins lock out the user, as explained in Chapter 20.
PasswordSignInAsync(username, password, persistent, lockout)	This is a convenience method that retrieves the user objects from the store using the specified username and attempts to sign in with a password.

The result produced by the methods in Table 18-7 is an instance of the SignInResult class, which defines the properties described in Table 18-8.

Table 18-8. *SignInResult Properties*

Name	Description
Succeeded	This property returns true if the user was successfully signed into the application and false otherwise.
IsLockedOut	This property returns true if the sign in attempt was unsuccessful because the user is locked out. I explain lockouts in Chapter 20.
IsNotAllowed	This property returns true if the user is not allowed to sign into the application, which I explain in Chapter 20.
RequiresTwoFactor	This property returns true if the user is required to provide additional credentials, which I explain in Chapter 20.

The most important SignInResult property is Suceeded, which returns true or false to indicate the outcome of the sign-in attempt. The other properties are used to provide additional detail about why an attempt was refused, all of which rely on features that are described in other chapters.

In Listing 18-10, I have updated the SignIn Razor Page to prompt for a password that is used to sign the user into the application.

Listing 18-10. Using a Password in the SignIn.cshtml File in the Pages Folder

```
@page "{code:int?}"
@model ExampleApp.Pages.SignInModel
@using Microsoft.AspNetCore.Http

@if (Model.Code == StatusCodes.Status401Unauthorized) {
    <h3 class="bg-warning text-white text-center p-2">
        401 - Challenge Response
    </h3>
} else if (Model.Code == StatusCodes.Status403Forbidden) {
    <h3 class="bg-danger text-white text-center p-2">
        403 - Forbidden Response
    </h3>
}
<h4 class="bg-info text-white m-2 p-2">
    Current User: @Model.Username
</h4>

<div class="m-2">
    <form method="post">
        <div class="form-group">
            <label>User</label>
            <select class="form-control"
                    asp-for="Username" asp-items="@Model.Users">
            </select>
        </div>
        <div class="form-group">
            <label>Password</label>
            <input class="form-control" type="password" name="password" />
        </div>
        <button class="btn btn-info" type="submit">Sign In</button>
    </form>
</div>
```

The user is presented with an input element into which a password is entered. To process the password, I made the changes shown in Listing 18-11 to the page model class.

Listing 18-11. Using Passwords in the SignIn.cshtml.cs File in the Pages Folder

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Security.Claims;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Identity;
using System.Linq;
using ExampleApp.Identity;
using SignInResult = Microsoft.AspNetCore.Identity.SignInResult;

namespace ExampleApp.Pages {
    public class SignInModel : PageModel {

        public SignInModel(UserManager<AppUser> userManager,
        SignInManager<AppUser> signInManager) {
            UserManager = userManager;
            SignInManager = signInManager;
        }

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

        public SelectList Users => new SelectList(
            UserManager.Users.OrderBy(u => u.EmailAddress),
                "EmailAddress", "NormalizedEmailAddress");

        public string Username { get; set; }

        public int? Code { get; set; }

        public void OnGet(int? code) {
            Code = code;
            Username = User.Identity.Name ?? "(No Signed In User)";
        }

        public async Task<ActionResult> OnPost(string username,
                string password, [FromQuery]string returnUrl) {
            SignInResult result = SignInResult.Failed;
            AppUser user = await UserManager.FindByEmailAsync(username);
            if (user != null && !string.IsNullOrEmpty(password)) {
                result = await SignInManager.PasswordSignInAsync(user, password,
                    false, true);
            }
            if (!result.Succeeded) {
                Code = StatusCodes.Status401Unauthorized;
                return Page();
            }
            return Redirect(returnUrl ?? "/signin");
        }
    }
}
```

This value is used as an argument to the PasswordSignInAsync method to sign the user into the application:

```
...
result = await SignInManager.PasswordSignInAsync(user, password, false, true);
...
```

In addition to the user and password, the arguments passed to the `PasswordSignInAsync` method tell Identity that the cookie used to authenticate requests should not persist when the browser is closed and that repeated failed attempts should result in an account lockout, which I implement in Chapter 20.

I use the existing features of the Razor Page to display an error message when the user doesn't supply a password or when the password doesn't match the one in the user store.

There are two classes named `SignInResult` in the packages used in ASP.NET Core, and the C# compiler cannot tell which one is required, which is why I have used this `using` statement:

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

This tells the compiler that the Identity class named `SignInResult` should be used, resolving the ambiguity so I can use the `SignInResult` type in the code without qualifying It with a namespace.

Restart ASP.NET Core and request `http://localhost:5000/signin`. Select `alice@example.com` from the list and enter MySecret1$ into the password field. Click Sign In; the password will be validated, and Alice will be signed into the application, as shown in Figure 18-4. You will see the 401 – Challenge response if you omit the password or enter a different password, which is also shown in Figure 18-4.

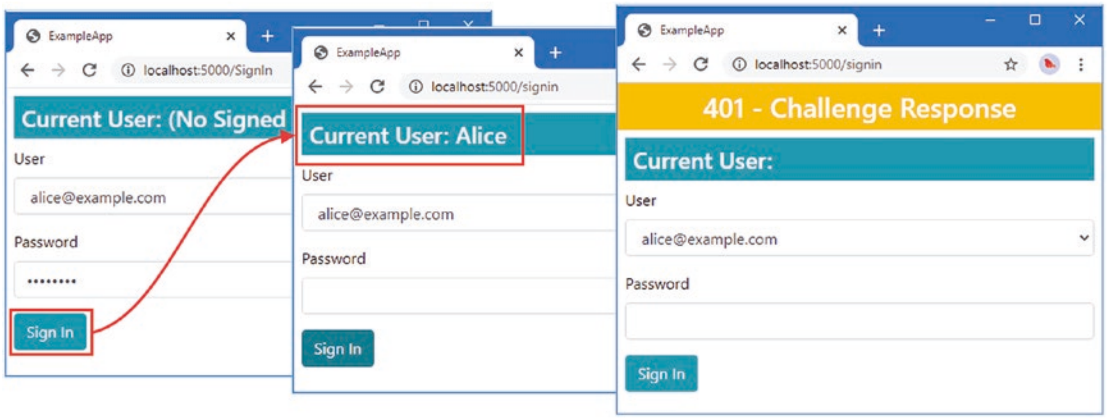


Figure 18-4. *Signing into the application with a password*

Managing Passwords

Passwords require several management workflows. The `UserManager<T>` class defines members that support the management of passwords, which I introduce in the sections that follow.

Changing and Recovering Passwords

The fundamental task for passwords is to allow a new one to be selected. This can be an intended change, where the existing password is known but a new one is required, or unintended, where the existing password has been forgotten and a new one must be defined so the user can sign in to the application. The `UserManager<T>` class defines the methods shown in Table 18-9 for changing passwords.

Table 18-9. *The UserManager<T> Methods for Changing Passwords*

Name	Description
`HasPasswordAsync(user)`	This method returns `true` if the user has a password, using the user store's `HasPasswordAsync` method.
`AddPasswordAsync(user, password)`	This method adds a password to the store for the specified user. The password is validated (described following the table) and hashed before it is passed to the user store's `SetPasswordHashAsync` method, a new security stamp is generated, and the user manager's update sequence is performed. An exception will be thrown if the user store's `GetPasswordHashAsync` method returns anything other than `null`.
`RemovePasswordAsync(user)`	This method sets the password for the specified user to `null` in the user store, generates a new security stamp, and performs the user manager's update sequence.
`ChangePasswordAsync(user, oldPassword, newPassword)`	This method performs a password change for the specified user. The old password is checked, and the new password is validated (described after the table) and hashed before being passed to the user store's `SetPasswordHashAsync` method. A new security stamp is generated, and the user manager's update sequence is performed.
`GeneratePasswordResetToken Async(user)`	This method generates a token that can be used to confirm a password reset.
`ResetPasswordAsync(user, token, newPassword)`	This method checks a password reset token and, if it is valid, stores the specified password for the user. The password hashed and passed to the user store's `SetPasswordHashAsync` method, a new security stamp is generated, and the user manager's update sequence is performed.

To allow a password to be changed, add a Razor Page named `PasswordChange.cshtml` to the `Pages/Store` folder with the content shown in Listing 18-12.

Listing 18-12. The Contents of the PasswordChange.cshtml File in the Pages/Store Folder

```
@page "/password/change/{success:bool?}"
@model ExampleApp.Pages.Store.PasswordChangeModel

<h4 class="bg-primary text-white text-center p-2">Change Password</h4>

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

@if (Model.Success) {
    <h5 class="bg-success text-white text-center p-2">Password Changed</h5>
}

<div class="m-2">
    <form method="post">
        <table class="table table-sm table-striped">
            <tbody>
                <tr><th>Your Username</th>
                    <td>@HttpContext.User.Identity.Name</td></tr>
                <tr>
                    <th>Existing Password</th>
                    <td><input class="w-100" type="password" name="oldPassword" />
                    </td>
                </tr>
                <tr>
                    <th>New Password</th>
                    <td><input class="w-100" type="password" name="newPassword" />
                    </td>
                </tr>
            </tbody>
        </table>
        <div class="text-center">
            <button class="btn btn-primary">Change</button>
            <a class="btn btn-secondary" asp-page="/SignIn">Back</a>
        </div>
    </form>
</div>
```

The Razor Page presents the user with input elements for the existing and new passwords. Use the `PasswordChange.cshtml.cs` file to define the page model class shown in Listing 18-13. (You will need to create this file if you are using Visual Studio Code.)

Listing 18-13. The Contents of the PasswordChange.cshtml.cs File in the Pages/Store Folder

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

namespace ExampleApp.Pages.Store {
    public class PasswordChangeModel : PageModel {

        public PasswordChangeModel(UserManager<AppUser> manager) =>
            UserManager = manager;

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

        [BindProperty(SupportsGet = true)]
        public bool Success { get; set; } = false;

        public async Task<IActionResult> OnPost(string oldPassword,
                string newPassword) {
            string username = HttpContext.User.Identity.Name;
            if (username != null) {
                AppUser user = await UserManager.FindByNameAsync(username);
                if (user != null && !string.IsNullOrEmpty(oldPassword)
                        && !string.IsNullOrEmpty(newPassword)) {
                    IdentityResult result = await UserManager.ChangePasswordAsync(
                        user, oldPassword, newPassword);
                    if (result.Succeeded) {
                        Success = true;
                    } else {
                        foreach (IdentityError err in result.Errors) {
                            ModelState.AddModelError("", err.Description);
                        }
                    }
                }
            }
            return Page();
        }
    }
}
```

In this example, I rely on the ASP.NET Core `HttpContext.User` property to provide me with details of the signed-in user, which I can then use to get the `AppUser` object that represents the user from the store. I call the `ChangePasswordAsync` method to change the password using the values provided by the user. The result of the `ChangePasswordAsync` method is an `IdentityResult` object that indicates whether the operation was successful and provides error details if it was not. If there are errors, I add them to the ASP.NET Core model state so they can be displayed to the user.

Resetting Passwords

The process for changing a password is simple because we can be confident that only the user knows their existing password, which is the same standard of security the application requires for signing in.

Performing a password reset is more complex because the user has forgotten or lost the password. Instead, a confirmation step is required, which generates a token and is sent to the user outside of the application, most often as a link sent by email or a code sent by SMS. The user clicks the link or enters the code into an HTML form and selects a new password.

I do not cover the selection and configuration of email and SMS services in this book, so I will simulate sending the confirmation token by writing messages to the console.

I can use the token generator used earlier to confirm phone number address changes. Listing 18-14 registers the token generator so it will be used for password resets.

Listing 18-14. Registering a Token Generator in the Startup.cs File in the ExampleApp Folder

```
...
services.AddIdentityCore<AppUser>(opts => {
    opts.Tokens.EmailConfirmationTokenProvider = "SimpleEmail";
    opts.Tokens.ChangeEmailTokenProvider = "SimpleEmail";
    opts.Tokens.PasswordResetTokenProvider = TokenOptions.DefaultPhoneProvider;
})
.AddTokenProvider<EmailConfirmationTokenGenerator>("SimpleEmail")
.AddTokenProvider<PhoneConfirmationTokenGenerator>(TokenOptions.DefaultPhoneProvider)
.AddSignInManager();
...
```

The options pattern is used to set the `IdentityOptions.Tokens.PasswordResetTokenProvider` property, whose value specifies the name of the password reset token generator. Next, add a Razor Page named `PasswordReset.cshtml` to the `Pages/Store` folder with the content shown in Listing 18-15.

Listing 18-15. The Contents of the PasswordReset.cshtml File in the Pages/Store Folder

```
@page "/password/reset"
@model ExampleApp.Pages.Store.PasswordResetModel

<h4 class="bg-primary text-white text-center p-2">Password Reset</h4>

<div class="m-2">
    <form method="post">
        <div class="form-group">
            <label>Email Address</label>
            <input class="form-control" name="email"  />
        </div>
        <button class="btn btn-primary mt-2" type="submit">Reset Password</button>
    </form>
</div>
```

The Razor Page displays a simple form that allows the user to enter their email address. To implement the page model class, add the code shown in Listing 18-16 to the `PasswordReset.cshtml.cs` file in the `Pages/Store` folder. (You will have to create this file if you are using Visual Studio Code.)

Listing 18-16. The Contents of the PasswordReset.cshtml.cs File in the Pages/Store Folder

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

namespace ExampleApp.Pages.Store {

    public class PasswordResetModel : PageModel {

        public PasswordResetModel(UserManager<AppUser> manager,
                ISMSSender sender) {
            UserManager = manager;
            SMSSender = sender;
        }

        public UserManager<AppUser> UserManager { get; set; }
        public ISMSSender SMSSender { get; set; }

        public async Task<IActionResult> OnPost(string email) {
            AppUser user  = await UserManager.FindByEmailAsync(email);
            if (user != null) {
                string token =
                    await UserManager.GeneratePasswordResetTokenAsync(user);
                SMSSender.SendMessage(user, $"Your password reset token is {token}");
            }
            return RedirectToPage("PasswordResetConfirm", new { email = email });
        }
    }
}
```

The handler method queries the user store for the AppUser object and uses the GeneratePasswordResetTokenAsync method to produce a token, which is sent to the user via (simulated) SMS, after which the client is redirected.

To allow the user to provide a new password and the confirmation token, add a Razor Page named PasswordResetConfirm.cshtml to the Pages/Store folder with the content shown in Listing 18-17.

Listing 18-17. The Contents of the PasswordResetConfirm.cshtml File in the Pages/Store Folder

```
@page "/password/reset/{email}"
@model ExampleApp.Pages.Store.PasswordResetConfirmModel

<h4 class="bg-primary text-white text-center p-2">Password Reset</h4>

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

@if (Model.Changed) {
    <h5 class="bg-success text-white text-center p-2">Password Changed</h5>
    <div class="text-center">
        <a href="/signin" class="btn btn-primary m-2">OK</a>
    </div>
} else {
    <div class="m-2">
        <form method="post">
            <div class="form-group">
                <label>Email Address</label>
                <input class="form-control" name="email" value="@Model.Email" />
            </div>
            <div class="form-group">
                <label>New Password</label>
                <input class="form-control" name="password" type="password" />
            </div>
            <div class="form-group">
                <label>Confirmation Token</label>
                <input class="form-control" name="token"  />
            </div>
            <button class="btn btn-primary mt-2" type="submit">
                Reset Password
            </button>
        </form>
    </div>
}
```

The view section of the page displays either a form that allows the password and code to be entered or a summary that confirms the password has been changed. To define the page model class, add the code shown in Listing 18-18 to the PasswordResetConfirm.cshtml.cs file in the Pages/Store folder. (You will have to create this file if you are using Visual Studio Code.)

Listing 18-18. The Contents of the PasswordResetConfirm.cshtml.cs File in the Pages/Store Folder

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

namespace ExampleApp.Pages.Store {

    public class PasswordResetConfirmModel : PageModel {

        public PasswordResetConfirmModel(UserManager<AppUser> manager)
            => UserManager = manager;

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

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

        [BindProperty(SupportsGet = true)]
        public bool Changed { get; set; } = false;

        public async Task<IActionResult> OnPostAsync(string password, string token) {
            AppUser user = await UserManager.FindByEmailAsync(Email);
            if (user != null) {
                IdentityResult result = await UserManager.ResetPasswordAsync(user,
                    token, password);
                if (result.Succeeded) {
                    return RedirectToPage(new { Changed = true });
                } else {
                    foreach (IdentityError err in result.Errors) {
                        ModelState.AddModelError("", err.Description);
                    }
                }
            } else {
                ModelState.AddModelError("", "Password Change Error");
            }
            return Page();
        }
    }
}
```

The page handler method uses the ResetPasswordAsync method to change the password and adds an error to the model state if the password change fails or if the email address provided by the user can't be located in the user store.

Integrating the Password Features

To integrate password changes with the rest of the application, add the elements shown in Listing 18-19 to the SignIn Razor Page.

Listing 18-19. Integrating Password Features in the SignIn.cshtml File in the Pages Folder

```
@page "{code:int?}"
@model ExampleApp.Pages.SignInModel
@using Microsoft.AspNetCore.Http

@if (Model.Code == StatusCodes.Status401Unauthorized) {
    <h3 class="bg-warning text-white text-center p-2">
        401 - Challenge Response
    </h3>
} else if (Model.Code == StatusCodes.Status403Forbidden) {
    <h3 class="bg-danger text-white text-center p-2">
        403 - Forbidden Response
    </h3>
}
<h4 class="bg-info text-white m-2 p-2">
    Current User: @Model.Username
</h4>

<div class="m-2">
    <form method="post">
        <div class="form-group">
            <label>User</label>
            <select class="form-control"
                    asp-for="Username" asp-items="@Model.Users">
            </select>
        </div>
        <div class="form-group">
            <label>Password</label>
            <input class="form-control" type="password" name="password" />
        </div>
        <button class="btn btn-info" type="submit">Sign In</button>
        @if (User.Identity.IsAuthenticated) {
            <a asp-page="/Store/PasswordChange" class="btn btn-secondary"
                asp-route-id="@Model.User?
                        .FindFirst(ClaimTypes.NameIdentifier)?.Value">
                    Change Password
            </a>
        } else {
          <a class="btn btn-secondary" href="/password/reset">Reset Password</a>
        }
    </form>
</div>
```

The button the user sees depends on whether the user has signed in. If they are signed in, then the application assumes they know their password and can perform a password change without requiring confirmation. If there is no signed-in user, then the user is presented with a button to perform a password reset. Restart ASP.NET Core and request `http://localhost:5000/signin`.

Sign in as `alice@example.com` with the password `MySecret1$`. Once you are signed in, you will see the Change Password button, which will lead you to the password change feature for the signed-in user, as shown in Figure 18-5.

Enter MySecret1$ into the Existing Password field and MySecret2$ into the New Password field. Click Change and the user's password will be updated.

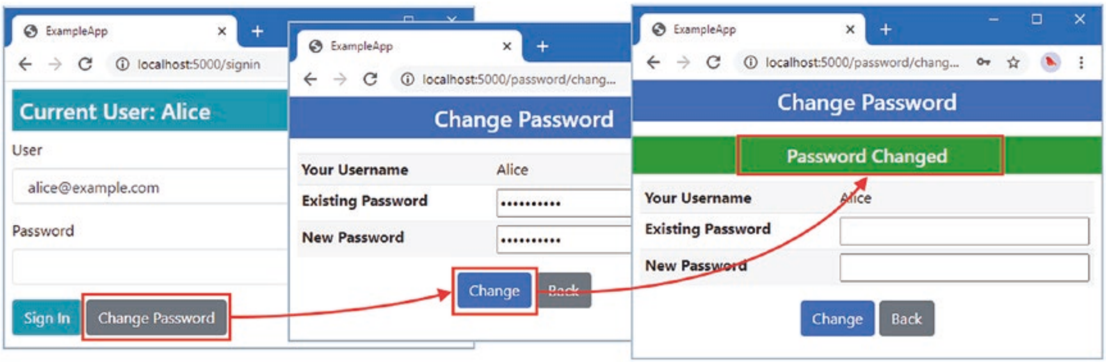


Figure 18-5. *Performing a password change*

To test the password reset feature, request `http://localhost:5000/signout` and sign the existing user out of the application. Next, request `http://localhost:5000/signin` and click the Reset Password button. Enter bob@example.com into the text field and click the Reset Password button. The display will change to prompt you for a new password and to render the validation code, as shown in Figure 18-6.

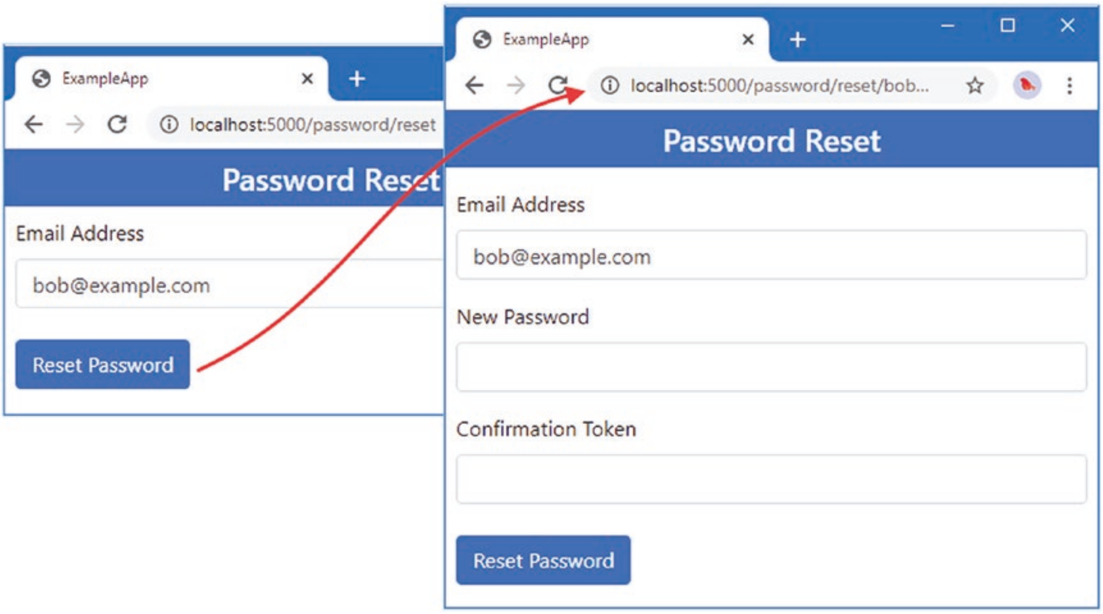


Figure 18-6. *Performing a password reset*

Examine the console output from ASP.NET Core, and you will see a simulated SMS message like this one:

```
--- SMS Starts ---
To: 123-4567
Your password reset token is C91430
--- SMS Ends ---
```

Enter MySecret2$ into the New Password field, enter C91430 into the Confirmation Token field, and click the Reset Password button. The password will be reset, as shown in Figure 18-7. You can now sign into the application using the new password.

■ **Note** The confirmation code is `C91430` because the seed data uses a fixed value for the security stamp when it adds users to the store. Subsequent changes generate different security stamps and, as a consequence, different confirmation tokens.

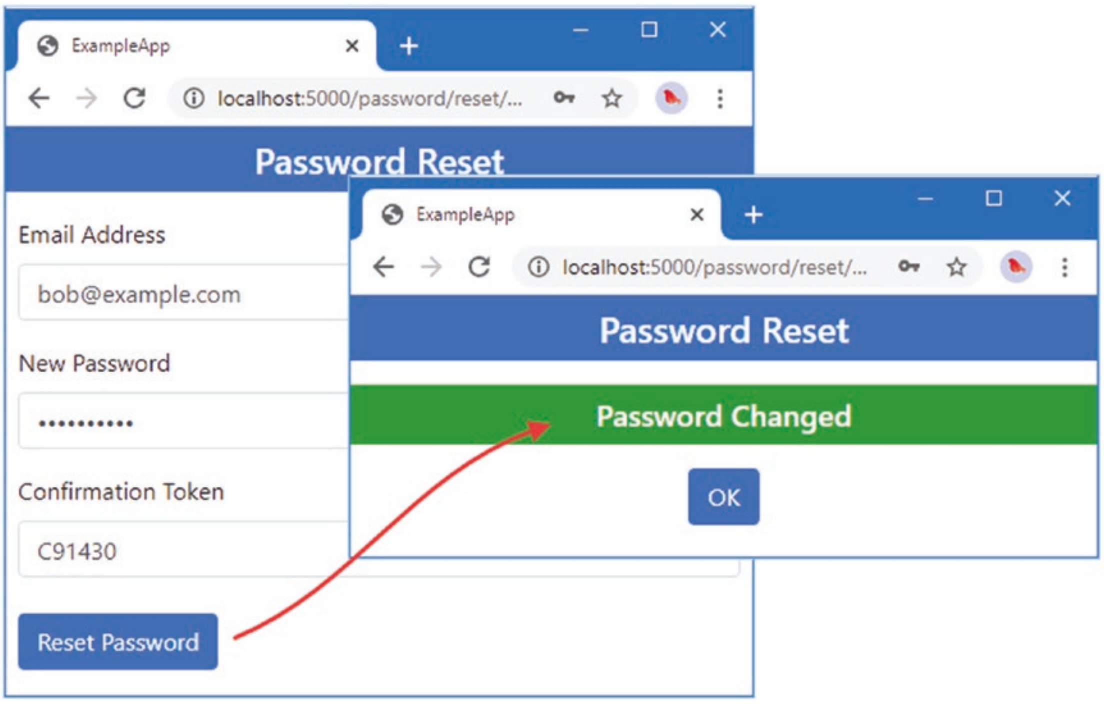
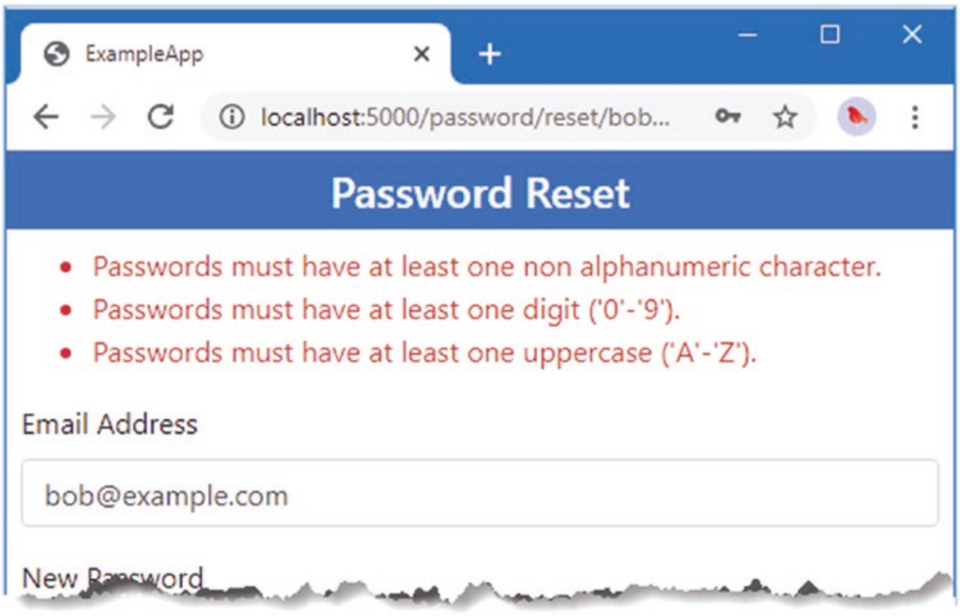


Figure 18-7. Completing the password reset

Validating Passwords

I was specific about the passwords used to test new features because Identity applies a password validation policy by default. Attempt a password change or reset using `mysecret` as the new password, for example, and you will see a series of error messages, as shown in Figure 18-8.



Password Reset

- Passwords must have at least one non alphanumeric character.
- Passwords must have at least one digit ('0'-'9').
- Passwords must have at least one uppercase ('A'-'Z').

Email Address

bob@example.com

New Password

Figure 18-8. *Identity password validation*

The validation policy is configured using the options pattern. The `IdentityOptions.Password` property returns an instance of the `PasswordOptions` class, which defines the properties described in Table 18-10.

Table 18-10. *The PasswordOptions Properties*

Name	Description
`RequiredLength`	This property specifies the minimum length for a password. The default value is 6.
`RequiredUniqueChars`	This property specifies the minimum number of unique characters in a password. The default value is 1.
`RequireNonAlphanumeric`	This property specifies whether passwords must contain a nonalphanumeric character, such as a punctuation mark. The default value is `true`.
`RequireLowercase`	This property specifies whether passwords must contain at least one lowercase character. The default value is `true`.
`RequireUppercase`	This property specifies whether passwords must contain at least one uppercase character. The default value is `true`.
`RequireDigit`	This property specifies whether passwords must contain at least one numeric digit. The default value is `true`.

The combined effect of the default values is that passwords must contain at least six characters, which must be a mix of uppercase and lowercase, numbers, and punctuation. Listing 18-20 uses the options pattern to change the password validation policy.

Listing 18-20. Changing the Password Validation Policy in the Startup.cs File in the ExampleApp Folder

```
...
services.AddIdentityCore<AppUser>(opts => {
    opts.Tokens.EmailConfirmationTokenProvider = "SimpleEmail";
    opts.Tokens.ChangeEmailTokenProvider = "SimpleEmail";
    opts.Tokens.PasswordResetTokenProvider = TokenOptions.DefaultPhoneProvider;
    opts.Password.RequireNonAlphanumeric = false;
    opts.Password.RequireLowercase = false;
    opts.Password.RequireUppercase = false;
    opts.Password.RequireDigit = false;
    opts.Password.RequiredLength = 8;
})
.AddTokenProvider<EmailConfirmationTokenGenerator>("SimpleEmail")
.AddTokenProvider<PhoneConfirmationTokenGenerator>(TokenOptions.DefaultPhoneProvider)
.AddSignInManager();
...
```

In this listing, I have disabled the requirements for mixing types of character and increased the minimum length requirement.

CHOOSING A SENSIBLE PASSWORD VALIDATION POLICY

Most password validation policies are intended to force the user to pick strong passwords but rarely work as intended. The only effect of restrictive password validation is an increase in the effort users will apply to bypass them.

The best practice has changed in recent years to focus on enforcing only a minimum length for passwords (usually eight characters), which should be supplemented by two-factor authentication (which I describe in Chapter 20) and features such as account lockout (also described in Chapter 20) and blocking commonly weak passwords (demonstrated in the next section).

There are some good sources for describing modern password restrictions, including Microsoft (`https://docs.microsoft.com/en-us/microsoft-365/admin/misc/password-policy-recommendations`) and NIST (`https://pages.nist.gov/800-63-3/sp800-63b.html`) and Wikipedia (`https://en.wikipedia.org/wiki/Password_policy`).

Adding Custom Password Validation

The `UserManager<T>` class performs password validation by using dependency injection to receive the set of services for the `IPasswordValidator<T>` interface, which defines the method described in Table 18-11. Each registered `IPasswordValidator<T>` implementation is asked to validate passwords before they are hashed and stored.

Table 18-11. *The IPasswordValidator<T> Method*

Name	Description
`ValidateAsync(manager, user, password`	This method validates a password for the specified user, with access to the user store provided through the specified `UserManager<T>`. The result is an `IdentityResult` object.

To implement a custom password validator, I am going to perform two checks. The first check is to block the 20 most common passwords. There are many lists of common passwords available, and I have chosen 20 as a manageable number for this example.

The second check is to exclude passwords that have been exposed publicly using the excellent `api.pwnedpasswords.com` web service provided by `haveibeenpwned.com`, which is documented at `https://haveibeenpwned.com/API/v3#PwnedPasswords`. The web service accepts the first five characters from a SHA1 hash generated from a password and returns a list of all the matching hashes from the extensive database of publicly exposed passwords. This allows clients to check passwords without sending the complete hash code, and if the hash of the password to validate appears in the response from the web service, it will not pass validation. The web service helpful includes the number of times a password has been exposed.

■ **Note** The point of this example is to demonstrate the use of a remote web service for password validation, which can be useful because the data sets for bad passwords are too large and too volatile to include directly in projects. But that's not to say that you should use this specific web service or that you should automatically invalidate any password that has been compromised. The more passwords you exclude, the more frustrated your users will become when they choose a password.

Add a class named `PasswordValidator.cs` to the `ExampleApp/Identity` folder and use it to define the class shown in Listing 18-21.

Listing 18-21. The Contents of the PasswordValidator.cs File in the Identity Folder

```
using Microsoft.AspNetCore.Identity;
using System;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;
using System.Security.Cryptography;
using System.Text;
using System.Collections.Generic;

namespace ExampleApp.Identity {
    public class PasswordValidator : IPasswordValidator<AppUser> {
        // set this field to false to disable the web service check
        private const bool remoteCheck = true;

        public async Task<IdentityResult> ValidateAsync(UserManager<AppUser> manager,
                AppUser user, string password) {
            IEnumerable<IdentityError> errors = CheckTop20(password);
            if (remoteCheck) {
                errors = errors.Concat(await CheckHaveIBeenPwned(password));
            }
            return errors.Count() == 0
                ? IdentityResult.Success : IdentityResult.Failed(errors.ToArray());
        }

        private async Task<IEnumerable<IdentityError>> CheckHaveIBeenPwned(
                string password) {
            string hash = BitConverter.ToString(SHA1.Create()
                .ComputeHash(Encoding.UTF8.GetBytes(password)))
                .Replace("-", string.Empty);
            string firstSection = hash[0..5];
            string secondSection = hash[5..];
            HttpResponseMessage response = await new HttpClient()
                .GetAsync($"https://api.pwnedpasswords.com/range/{firstSection}");
            string matchingHashes = await response.Content.ReadAsStringAsync();
            string[] matches = matchingHashes.Split("\n",
                StringSplitOptions.RemoveEmptyEntries);
            string match = matches.FirstOrDefault(match =>
                match.StartsWith(secondSection,
                    StringComparison.CurrentCultureIgnoreCase));
            if (match == null) {
                return Enumerable.Empty<IdentityError>();
            } else {
                long count = long.Parse(match.Split(":")[1]);
                return new[] {new IdentityError {
                    Description = $"Password has been compromised {count:N0} times"
                }};
            }
        }

        private IEnumerable<IdentityError> CheckTop20(string password) {
            if (commonPasswords.Any(commonPassword =>
                string.Equals(commonPassword, password,
                     StringComparison.CurrentCultureIgnoreCase))) {
                return new [] {
                    new IdentityError {
                        Description = "The top 20 passwords cannot be used"
                    }
                };
            }
            return Enumerable.Empty<IdentityError>();
        }

        private static string[] commonPasswords = new[] {
            "123456", "123456789", "qwerty", "password", "1111111", "12345678",
            "abc123", "1234567", "password1", "12345", "1234567890", "123123",
            "000000", "Iloveyou", "1234", "1q2w3e4r5t", "Qwertyuiop", "123",
            "Monkey", "Dragon"};
    }
}
```

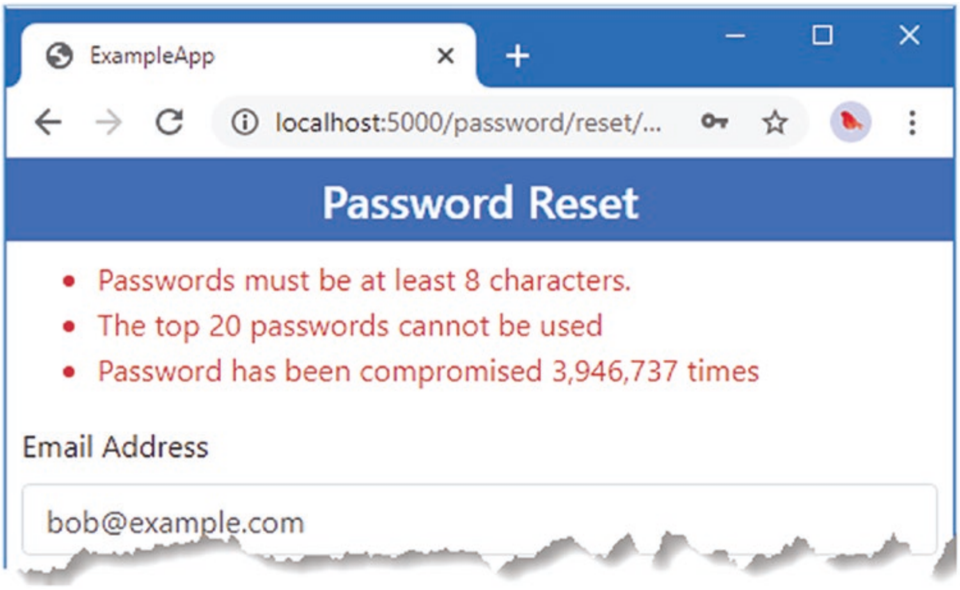
The validator checks to see if the password selected by the user is on a statically defined list of passwords and sends a request to the web service. The result of both checks is combined into an `IdentityResult`, which Identity will send back to the Razor Page that has requested a password change. Listing 18-22 registers the validator as a service. As with the user validator I defined in Chapter 16, the service is defined after the `AddIdentityCore` method is called so that the new validator supplements the built-in one. If I had defined the service before the `AddIdentityCore` method, then the new validator would replace the built-in one.

■ **Note** Readers who are following the examples offline can set the `remoteCheck` field defined in Listing 18-21 to `false`. This will disable the web service check but still invalidate passwords from the statically defined top 20 list.

Listing 18-22. Defining a Service in the Startup.cs File in the ExampleApp Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddSingleton<ILookupNormalizer, Normalizer>();
    services.AddSingleton<IUserStore<AppUser>, UserStore>();
    services.AddSingleton<IEmailSender, ConsoleEmailSender>();
    services.AddSingleton<ISMSSender, ConsoleSMSSender>();
    services.AddSingleton<IUserClaimsPrincipalFactory<AppUser>,
        AppUserClaimsPrincipalFactory>();
    services.AddSingleton<IPasswordHasher<AppUser>, SimplePasswordHasher>();

    services.AddIdentityCore<AppUser>(opts => {
        opts.Tokens.EmailConfirmationTokenProvider = "SimpleEmail";
        opts.Tokens.ChangeEmailTokenProvider = "SimpleEmail";
        opts.Tokens.PasswordResetTokenProvider = TokenOptions.DefaultPhoneProvider;

        opts.Password.RequireNonAlphanumeric = false;
        opts.Password.RequireLowercase = false;
        opts.Password.RequireUppercase = false;
        opts.Password.RequireDigit = false;
        opts.Password.RequiredLength = 8;
    })
    .AddTokenProvider<EmailConfirmationTokenGenerator>("SimpleEmail")
    .AddTokenProvider<PhoneConfirmationTokenGenerator>
        (TokenOptions.DefaultPhoneProvider)
    .AddSignInManager();

    services.AddSingleton<IUserValidator<AppUser>, EmailValidator>();
    services.AddSingleton<IPasswordValidator<AppUser>, PasswordValidator>();

    services.AddAuthentication(opts => {
        opts.DefaultScheme
            = IdentityConstants.ApplicationScheme;
    }).AddCookie(IdentityConstants.ApplicationScheme, opts => {
        opts.LoginPath = "/signin";
        opts.AccessDeniedPath = "/signin/403";
    });
    services.AddAuthorization(opts => {
        AuthorizationPolicies.AddPolicies(opts);
    });
    services.AddRazorPages();
    services.AddControllersWithViews();
}
...
```

Restart ASP.NET Core request `http://localhost:5000/signin` and click the Reset Password or Change Password button, depending on whether a user is signed into the application. Try changing the password to `qwerty`, and you will see validation errors from both the built-in and custom password validators, as shown in Figure 18-9.



- Passwords must be at least 8 characters.
- The top 20 passwords cannot be used
- Password has been compromised 3,946,737 times

Email Address

bob@example.com

Figure 18-9. *Validating passwords*

Setting Passwords Administratively

The `UserManager<T>` class defines a convenience method for creating users and assigning an initial password, as shown in Table 18-12 for future quick reference.

Table 18-12. *The UserManager<T> Method for Creating Users with Passwords*

Name	Description
`CreateAsync(user, password)`	This method validates the password, creates a new password hash, and calls the user's store `SetPasswordHashAsync` method. A new security stamp is generated, and the user manager's `CreateAsync` method is called.

Administrative password resets require a different approach because the administrator doesn't know the user's password and there is no method for changing a password without either the existing password or a confirmation code. I find the most reliable way to let an administrator change a password is to remove the existing password and add a new one.

To add support for setting passwords, add a Razor View named `_EditUserPassword.cshtml` to the `Pages/Store` folder with the content shown in Listing 18-23.

Listing 18-23. The Contents of the _EditUserPassword.cshtml File in the Pages/Store Folder

```
@model AppUser
@inject UserManager<AppUser> UserManager

@if (UserManager.SupportsUserPassword) {
    <tr>
        <td>New Password</td>
        <td>
            <input class="w-00" name="newPassword" />
        </td>
    </tr>
}
```

The view renders an `input` element whose name doesn't correspond to any of the properties defined by the `AppUser` class, which will allow me to handle password changes without the model binding process trying to assign a raw password string when a hashed password is required.

Add the element shown in Listing 18-24 to the `EditUser.cshtml` file to incorporate the new partial view into the application.

Listing 18-24. Incorporating a Partial View in the EditUser.cshtml File in the Pages/Store Folder

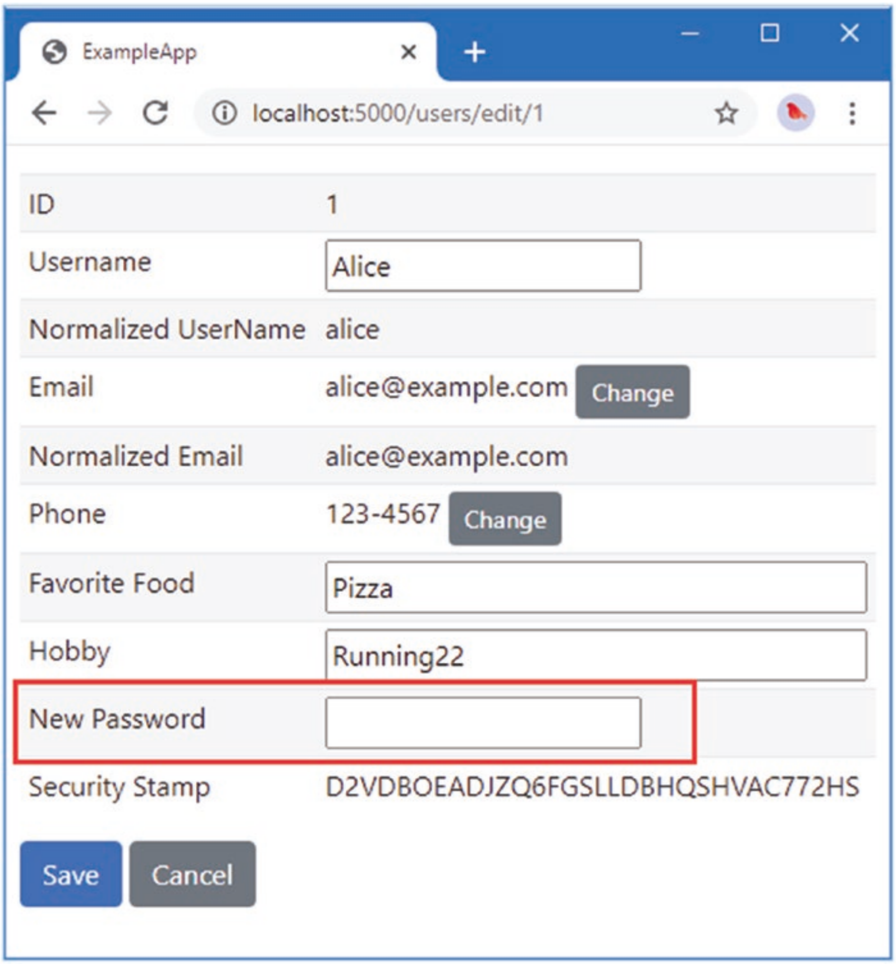
```
@page "/users/edit/{id?}"
@model ExampleApp.Pages.Store.UsersModel

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

<div class="m-2">
    <form method="post">
        <input type="hidden" name="id" value="@Model.AppUserObject.Id" />
        <table class="table table-sm table-striped">
            <tbody>
                <partial name="_EditUserBasic" model="@Model.AppUserObject" />
                <partial name="_EditUserEmail" model="@Model.AppUserObject" />
                <partial name="_EditUserPhone" model="@Model.AppUserObject" />
                <partial name="_EditUserCustom" model="@Model.AppUserObject" />
                <partial name="_EditUserPassword" model="@Model.AppUserObject" />
                <partial name="_EditUserSecurityStamp"
                         model="@Model.AppUserObject" />
            </tbody>
        </table>
        <div>
            <button type="submit" class="btn btn-primary">Save</button>
            <a asp-page="users" class="btn btn-secondary">Cancel</a>
        </div>
    </form>
</div>
```

In Listing 18-25, I have updated the EditUser page model class to change passwords.

Listing 18-25. Changing Passwords in the EditUser.cshtml.cs File in the Pages/Store Folder

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

namespace ExampleApp.Pages.Store {

    public class UsersModel : PageModel {

        public UsersModel(UserManager<AppUser> userMgr) => UserManager = userMgr;

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

        public AppUser AppUserObject { get; set; } = new AppUser();

        public async Task OnGetAsync(string id) {
            if (id != null) {
                AppUserObject = await UserManager.FindByIdAsync(id) ?? new AppUser();
            }
        }

        public async Task<IActionResult> OnPost(AppUser user, string newPassword) {
            IdentityResult result = IdentityResult.Success;
            AppUser storeUser = await UserManager.FindByIdAsync(user.Id);
            if (storeUser == null) {
                if (string.IsNullOrEmpty(newPassword)) {
                    ModelState.AddModelError("", "Password Required");
                    return Page();
                }
                result = await UserManager.CreateAsync(user, newPassword);
            } else {
                storeUser.UpdateFrom(user, out bool changed);
                if (newPassword != null) {
                    if (await UserManager.HasPasswordAsync(storeUser)) {
                        await UserManager.RemovePasswordAsync(storeUser);
                    }
                    result = await UserManager.AddPasswordAsync(storeUser,
                        newPassword);
                }
                if (changed && UserManager.SupportsUserSecurityStamp) {
                    await UserManager.UpdateSecurityStampAsync(storeUser);
                }
                if (result != null && result.Succeeded) {
                    result = await UserManager.UpdateAsync(storeUser);
                }
            }
            if (result.Succeeded) {
                return RedirectToPage("users", new { searchname = user.Id });
            } else {
                foreach (IdentityError err in result.Errors) {
                    ModelState.AddModelError("", err.Description ?? "Error");
                }
                AppUserObject = user;
                return Page();
            }
        }
    }
}
```

Restart ASP.NET Core, request `http://localhost:5000/users`, and click the Edit button for one of the users. You will see the new text field, which can be used to select a new password, as shown in Figure 18-10.



ID	1
Username	Alice
Normalized UserName	alice
Email	alice@example.com
Normalized Email	alice@example.com
Phone	123-4567
Favorite Food	Pizza
Hobby	Running22
New Password	
Security Stamp	D2VDBOEADJZQ6FGSLLDBHQSHVAC772HS

Figure 18-10. *Changing a password*

Summary

In this chapter, I extended the user store so that it can manage passwords and created the workflows for signing the user into the application, changing passwords, and recovering passwords. I also explained how passwords are validated and demonstrated custom password validation. In the next chapter, I create a role store.