CHAPTER 21

Authenticators and Recovery Codes

In this chapter, I describe how ASP.NET Core Identity supports authenticators for two-factor authentication and how to provide users with recovery codes for emergency access when their second factor isn't available. Table 21-1 puts these features in context.

Table 21-1. *Putting Authenticators and Recovery Codes in Context*

Question	Answer
What are they?	Authenticators are apps that are configured with a secret key, after which they generate codes that can be used as part of a two-factor authentication process. Recovery codes are single-use codes that can be used as second factors, typically when an authenticator app has been lost.
Why are they useful?	Authenticators offer a more secure sign-in process than SMS. Recovery codes are useful because they allow users to sign in even when their second factor is unavailable.
How are they used?	Authenticators are set up with an initial key generated using Identity. During sign-in, the current code displayed by the authenticator is provided by the user and validated by Identity using the key. Recovery codes are entered as part of the sign-in process.
Are there any pitfalls or limitations?	Authenticators are typically smartphone apps, which not all users have access to. Recovery codes require users to prepare in advance, remember the codes when they need them, and generate new codes when they run out.
Are there any alternatives?	Alternative second factors can be used. Authentication can be delegated to third parties using the external authentication feature described in Chapter 22. Recovery codes are optional but should be used whenever two-factor authentication is enabled.

A. Freeman, *Pro ASP.NET Core Identity*, https://doi.org/10.1007/978-1-4842-6858-2_21

Table 21-2 summarizes the chapter.

Table 21-2. *Chapter Summary*

Problem	Solution	Listing
Support authenticators in two-factor sign-ins	Extend the user store by implementing the `IUserAuthenticatorKeyStore<T>` interface and manage the keys with the user manager methods.	3–8
Sign in users with an authenticator	Use the `TwoFactorAuthenticatorSignInAsync` method provided by the sign-in manager class.	9–12
Support recovery codes in two-factor sign-ins	Extend the user store by implementing the `IUserTwoFactorRecoveryCodeStore<T>` interface and manage the codes with the user manager methods.	13–18
Sign in users with a recovery code	Use the `TwoFactorRecoveryCodeSignInAsync` method provided by the sign-in manager class.	19–21

Preparing for This Chapter

This chapter uses the ExampleApp project from Chapter 20. To prepare for this chapter, disable the custom user confirmation service, as shown in Listing 21-1.

Listing 21-1. Disabling 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.AddSingleton<IRoleStore<AppRole>, RoleStore>();
    //services.AddSingleton<IUserConfirmation<AppUser>, UserConfirmation>();
...
```

Open a new command prompt, navigate to the `ExampleApp` folder, and run the command shown in Listing 21-2 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 21-2. 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 21-1. The data is stored only in memory, and changes will be lost when ASP.NET Core is stopped.

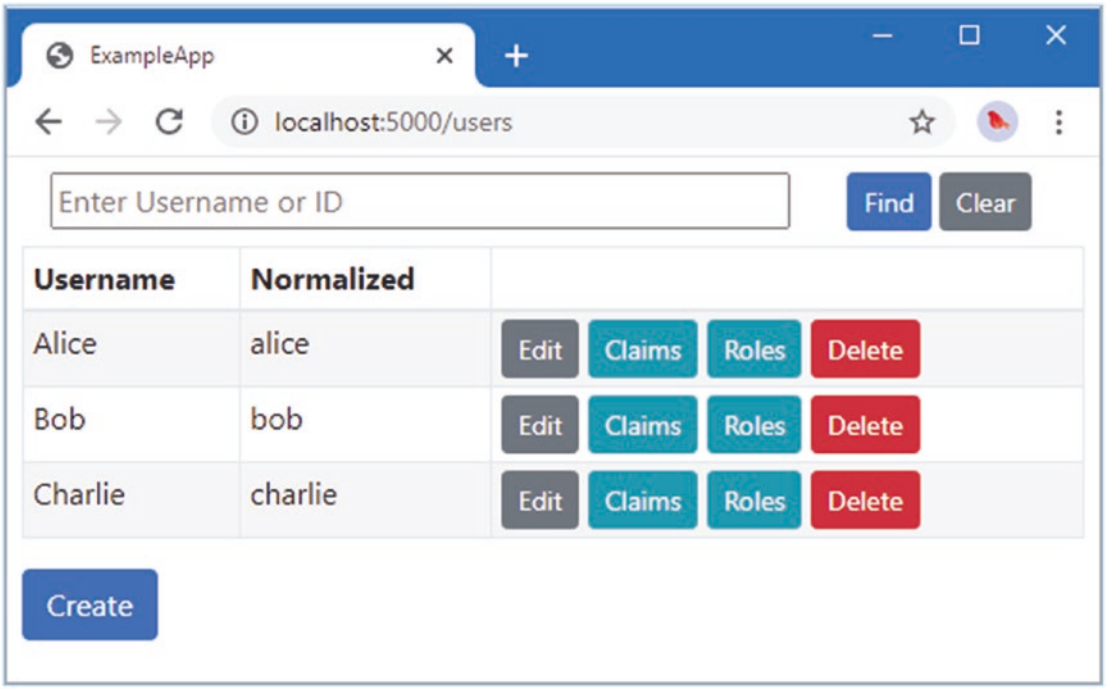

Figure 21-1. *Running the example application*

Using an Authenticator

One of the drawbacks of using SMS for two-factor authentication is that the user must have cell coverage to receive the security codes. One way to solve this problem is to use an *authenticator*, which is an app specifically intended to generate security codes while offline.

Authenticators have a setup phase and a code generation phase. During the setup phase, the ASP.NET Core Identity creates a key, also called a *seed*, which is shared with the authenticator app and stored securely. The key can be shared by copying and pasting a string of characters or, if the authenticator is a smartphone app, scanning a QR code.

During the code generation phase, the authenticator and Identity both use the same algorithm to generate a code using the shared key plus a modifier (also known as a *moving factor*) that ensures keys are different and cannot be intercepted and reused by an attacker. The authenticator displays the code to the user, which presents it to the application just as they would a code received via SMS. The user is signed in if the code generated by the authenticator matches the one generated by Identity.

To generate the same code, the authenticator and Identity need to use the same key and modifier. The key is shared when the authenticator is set up, but the modifier must be different for each sign-in. There are two standard ways to select a modifier: using a counter or using the current time. Codes that are generated using a counter are called *HMAC-based one-time passwords* (HOTPs), and codes generated using the time are called *time-based one-time passwords* (TOTPs). The challenge is that the authenticator and the application must be able to select the same modifier without being able to communicate.

When a counter is used, the authenticator and Identity both keep track of a counter. The authenticator increments the counter when the user presses a button and Identity increments the counter when the user is signed in. The two counters can drift apart if the user presses the button but doesn't use the code, so Identity must check a range of possible counter values during validation. A manual resynchronization process is required if the authenticator counter is incremented outside of the range that Identity checks.

The time-based approach works out how many intervals of a fixed duration have occurred since a specific time. The authenticator and Identity will produce the same modifier if they count the number of three-minute intervals that have occurred since the January 1, 1970, UTC, for example, just as long as the clocks they use are roughly synchronized. The clocks don't have to be completely synchronized—being accurate within a single time interval is close enough. The user won't be able to sign in if the clocks are further adrift because the authenticator and Identity will use different modifiers to produce the security codes.

TOTPs are more widely used because they are simpler to validate and don't require a resynchronization process. I generate TOTPs in the examples in this chapter.

Extending the User Class

The first step toward supporting authenticators is to extend the user class to keep track of the key. Add the properties shown in Listing 21-3 to the `AppUser` class.

Listing 21-3. Adding Properties 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; }

        public bool CanUserBeLockedout { get; set; } = true;
        public int FailedSignInCount { get; set; }
        public DateTimeOffset? LockoutEnd { get; set; }

        public bool TwoFactorEnabled { get; set; }
        public bool AuthenticatorEnabled { get; set; }
        public string AuthenticatorKey { get; set; }
    }
}
```

The new properties will be used to denote the user has an authenticator and to store the shared key that will be used to generate security codes. Each user has a key, and changing the key will prevent the user from signing in until the key has been shared with the authenticator.

Storing Authenticator Keys in the User Store

The `IUserAuthenticatorKeyStore<T>` interface is implemented by user stores that can manage authenticator keys. The interface defines the methods described in Table 21-3. As with the other user store interfaces described in this part of the book, the methods in Table 21-3 define a `CancellationToken` parameter named `token`, which is used to receive notifications when a task is canceled.

Table 21-3. *The IUserAuthenticatorKeyStore<T> Methods*

Name	Description
`SetAuthenticatorKeyAsync(user, key, token)`	This method sets the authenticator key for the specified user.
`GetAuthenticatorKeyAsync(user, token)`	This method gets the authenticator key for the specified user.

To add support for authenticator keys to the user store, add a class file named `UserStoreAuthenticatorKeys.cs` to the `Identity/Store` folder and use it to define the partial class shown in Listing 21-4.

Listing 21-4. The Contents of the UserStoreAuthenticatorKeys.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 : IUserAuthenticatorKeyStore<AppUser> {

        public Task<string> GetAuthenticatorKeyAsync(AppUser user,
                CancellationToken cancellationToken)
            => Task.FromResult(user.AuthenticatorKey);
```

```
        public Task SetAuthenticatorKeyAsync(AppUser user, string key,
                CancellationToken cancellationToken) {
            user.AuthenticatorKey = key;
            return Task.CompletedTask;
        }
    }
}
```

The implementation of the interface maps the methods onto the `AppUser` properties. The user store isn't responsible for generating the keys or tokens.

Managing Authenticator Keys

Authenticators require a setup phase so that the key can be shared. The `UserManager<T>` class provides the members described in Table 21-4 for working with authenticators.

Table 21-4. *The UserManager<T> Members for Authenticators*

Name	Description
`SupportsUserAuthenticatorKey`	This property returns true if the store implements the `IUserAuthenticatorKeyStore<T>` interface.
`GetAuthenticatorKeyAsync(user)`	This method gets the user's authenticator key by calling the store method with the same name.
`GenerateNewAuthenticatorKey()`	This method generates a new key.
`ResetAuthenticatorKeyAsync(user)`	This method resets the authenticator key for the specified user. The `GenerateNewAuthenticatorKey` method is used to generate a new key, which is assigned to the user by calling the store's `SetAuthenticatorKeyAsync` method. A new security stamp is generated, and the user manager's update sequence is performed to save the changes.

During the setup phase, the authenticator must be given the key generated by the application. This can be done by having the user copying and pasting the key string, but QR codes provide an elegant alternative for authenticators running on smartphones. ASP.NET Core Identity doesn't include the ability to generate QR codes directly, but Microsoft suggests the use of the QRCode.js JavaScript package. Run the command shown in Listing 21-5 in the `ExampleApp` folder to add the QRCode.js package to the project.

Listing 21-5. Adding a JavaScript Package

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

To generate a key and display it to the user, add a Razor Page named `AuthenticatorSetup.cshtml` to the `Pages` folder with the contents shown in Listing 21-6.

Listing 21-6. The Contents of the AuthenticatorSetup.cshtml File in the Pages Folder

```
@page "{id}"
@model ExampleApp.Pages.AuthenticatorSetupModel

<h4 class="bg-primary text-white text-center p-2">Authenticator Key</h4>

<div class="m-2">
    <table class="table table-sm table-bordered">
        <tbody>
            <tr>
                <th>User</th>
                <td>@Model.AppUser.UserName</td>
                @if (Model.AuthenticatorUrl != null) {
                    <td rowspan="3">
                        <div id="qrcode"></div>
                        <script type="text/javascript"
                             src="~/lib/qrcode/qrcode.min.js"></script>
                        <script type="text/javascript">
                            new QRCode(document.getElementById("qrcode"), {
                                text: "@Model.AuthenticatorUrl",
                                width: 150,
                                height: 150
                            });
                        </script>
                    </td>
                }
            </tr>
            <tr>
                <th>Authenticator Key</th>
                <td>@(Model.AppUser.AuthenticatorKey ?? "(No Key)")</td>
            </tr>
            <tr>
                <th>Authenticator Enabled</th>
                <td>
                    <span class="font-weight-bold
                          @(Model.AppUser.AuthenticatorEnabled
                              ? "text-success": "text-danger")">
                        @Model.AppUser.AuthenticatorEnabled
                    </span>
                </td>
            </tr>
        </tbody>
    </table>
</div>

<form method="post" class="m-1">
    <input type="hidden" name="Id" value="@Model.Id" />
    <div class="mt-2">
        <button class="btn btn-primary m-1" asp-route-task="enable">
            Enable Authenticator
        </button>
```

```
        <button class="btn btn-secondary m-1" asp-route-task="disable">
            Disable Authenticator
        </button>
        <button class="btn btn-info m-1">Generate New Key</button>
        <a href="/users" class="btn btn-secondary">Back</a>
    </div>
</form>
```

The page displays the current key for a user and whether the use of the authenticator is enabled. There are buttons that toggle the user of authenticators and generate new keys. A QR code is displayed using a page model property named `AuthenticatorUrl`.

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

Listing 21-7. The Contents of the AuthenticatorSetup.cshtml.cs File in the Pages Folder

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

namespace ExampleApp.Pages {

    public class AuthenticatorSetupModel : PageModel {

        public AuthenticatorSetupModel(UserManager<AppUser> userManager) =>
            UserManager = userManager;

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

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

        public AppUser AppUser { get; set; }

        public string AuthenticatorUrl { get; set; }

        public async Task OnGetAsync() {
            AppUser = await UserManager.FindByIdAsync(Id);
            if (AppUser != null) {
                if (AppUser.AuthenticatorKey != null) {
                    AuthenticatorUrl =
                        $"otpauth://totp/ExampleApp:{AppUser.EmailAddress}"
                        + $"?secret={AppUser.AuthenticatorKey}";
                }
            }
        }

        public async Task<IActionResult> OnPostAsync(string task) {
            AppUser = await UserManager.FindByIdAsync(Id);
```

```
            if (AppUser != null) {
                switch (task) {
                    case "enable":
                        AppUser.AuthenticatorEnabled = true;
                        AppUser.TwoFactorEnabled = true;
                        break;
                    case "disable":
                        AppUser.AuthenticatorEnabled = false;
                        AppUser.TwoFactorEnabled = false;
                        break;
                    default:
                        await UserManager.ResetAuthenticatorKeyAsync(AppUser);
                        break;
                }
                await UserManager.UpdateAsync(AppUser);
            }
            return RedirectToPage();
        }
    }
}
```

The GET handler method uses the `UserManager<T>.GetUserAsync` to get the `AppUser` object that represents the signed-in user and reads the value of the `AuthenticatorKey` and `AuthenticatorEnabled` properties. The POST handler method sets the `AuthenticatorEnabled` property and, if required, uses the `UserManager<T>.ResetAuthenticatorKeyAsync` method to generate a new key.

Authenticators that can scan a QR code expect to receive a URL in the format `otpauth://totp/<label>?secret=<key>`, where `<label>` identifies the user account and where `<key>` is the secret key.

■ **Tip** See https://github.com/google/google-authenticator/wiki/Key-Uri-Format for full details of the URL format used for authenticator QR codes.

To incorporate the new Razor Page into the application, add the element shown in Listing 21-8 to the `_UserTableRow` partial view.

Listing 21-8. Adding a Feature in the _UserTableRow.cshtml File in the Pages/Store Folder

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

<a asp-page="edituser" asp-route-id="@Model" class="btn btn-sm btn-secondary">
    Edit
</a>
@if (UserManager.SupportsUserClaim) {
    <a asp-page="claims" asp-route-id="@Model" class="btn btn-sm btn-info">
        Claims
    </a>
}
```

```
@if (UserManager.SupportsUserRole) {
    <a asp-page="userroles" asp-route-id="@Model" class="btn btn-sm btn-info">
        Roles
    </a>
}
@if (UserManager.SupportsUserAuthenticatorKey) {
    <a asp-page="/authenticatorsetup" asp-route-id="@Model"
       class="btn btn-sm btn-success">
        Authenticator
    </a>
}
```

Enabling Authenticators in Two-Factor Sign-Ins

The authenticator will replace the security code sent via SMS in the two-factor sign-in process. The `SignInManager<T>` class provides the method described in Table 21-5 that supports working with authenticators.

Table 21-5. *The SignInManager<T> Method for Authenticators*

Name	Description
`TwoFactorAuthenticatorSignInAsync(code, persistent, remember)`	This method works the same way as the `TwoFactorSignInAsync` method, except the code is validated by the authenticator token provider.

In Listing 21-9, I updated the `SignInTwoFactor` page to determine if the user is set up for an authenticator and, if so, display an appropriate message.

Listing 21-9. Supporting Authenticators in the SignInTwoFactor.cshtml File in the Pages Folder

```
@page
@model ExampleApp.Pages.SignInTwoFactorModel

<h4 class="bg-info text-white m-2 p-2">Two Factor Sign In</h4>

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

@if (Model.AuthenticatorEnabled) {
    <span class="m-2"> Enter the code displayed by your authenticator. </span>
} else {
    <span class="m-2"> We have sent a security code to your phone. </span>
    <a asp-page="/SignInTwoFactor" class="btn btn-sm btn-secondary">Resend Code</a>
}

<div class="m-2">
    <form method="post">
        <div class="form-group">
```

```
            <label>Enter security Code:</label>
            <input class="form-control" name="code"/>
        </div>
        <div class="form-check">
            <input class="form-check-input" type="checkbox" name="rememberMe" />
            <label class="form-check-label">Remember Me</label>
        </div>
        <div class="mt-2">
            <button class="btn btn-info" type="submit"
                    disabled="@(!ModelState.IsValid)">Sign In</button>
            <a asp-page="/Signin" class="btn btn-secondary">Cancel</a >
        </div>
    </form>
</div>
```

In Listing 21-10, I have updated the page model class to support the changes in the view and to validate authenticator security codes.

Listing 21-10. Supporting Authenticators in the SignInTwoFactor.cshtml.cs File in the Pages Folder

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

namespace ExampleApp.Pages {

    public class SignInTwoFactorModel : PageModel {

        public SignInTwoFactorModel(UserManager<AppUser> userManager,
                    SignInManager<AppUser> signInManager,
                    ISMSSender sender) {
            UserManager = userManager;
            SignInManager = signInManager;
            SMSSender = sender;
        }

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

        public bool AuthenticatorEnabled { get; set; }

        public async Task OnGet() {
            AppUser user = await SignInManager.GetTwoFactorAuthenticationUserAsync();
            if (user != null) {
                AuthenticatorEnabled = user.AuthenticatorEnabled;
                if (!AuthenticatorEnabled) {
                    await UserManager.UpdateSecurityStampAsync(user);
                    string token = await UserManager.GenerateTwoFactorTokenAsync(
```

```
                    user, IdentityConstants.TwoFactorUserIdScheme);
                SMSSender.SendMessage(user, $"Your security code is {token}");
            }
        }
    }

    public async Task<IActionResult> OnPost(string code, string rememberMe,
            [FromQuery] string returnUrl) {
        AppUser user = await SignInManager.GetTwoFactorAuthenticationUserAsync();
        if (user != null && !string.IsNullOrEmpty(code)) {
            SignInResult result = SignInResult.Failed;
            AuthenticatorEnabled = user.AuthenticatorEnabled;
            bool rememberClient = !string.IsNullOrEmpty(rememberMe);
            if (AuthenticatorEnabled) {
                string authCode = code.Replace(" ", string.Empty);
                result = await SignInManager.TwoFactorAuthenticatorSignInAsync(
                    authCode, false, rememberClient);
            } else {
                result = await SignInManager.TwoFactorSignInAsync(
                    IdentityConstants.TwoFactorUserIdScheme, code,
                    true, rememberClient);
            }
            if (result.Succeeded) {
                return Redirect(returnUrl ?? "/");
            } else if (result.IsLockedOut) {
                ModelState.AddModelError("", "Locked out");
            } else if (result.IsNotAllowed) {
                ModelState.AddModelError("", "Not allowed");
            } else {
                ModelState.AddModelError("", "Authentication failed");
            }
        }
        return Page();
    }
}
```

Some authenticators display security codes as groups of three digits, which means users will often enter codes with spaces. To prevent errors, I remove any spaces from the code the user provides and use the `SignInManager<T>.TwoFactorAuthenticatorSignInAsync` method to sign the user into the application.

Configuring the Application

The token generator used for authenticators is set using the `TokenOptions.DefaultAuthenticatorProvider` property, as shown in Listing 21-11.

Listing 21-11. Configuring the Application 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;
    opts.Lockout.MaxFailedAccessAttempts = 3;
    opts.SignIn.RequireConfirmedAccount = true;
})
.AddTokenProvider<EmailConfirmationTokenGenerator>("SimpleEmail")
.AddTokenProvider<PhoneConfirmationTokenGenerator>
    (TokenOptions.DefaultPhoneProvider)
.AddTokenProvider<TwoFactorSignInTokenGenerator>
    (IdentityConstants.TwoFactorUserIdScheme)
.AddTokenProvider<AuthenticatorTokenProvider<AppUser>>
    (TokenOptions.DefaultAuthenticatorProvider)
.AddSignInManager()
.AddRoles<AppRole>();
...
```

Authenticators work because the application and the app are using the same algorithm and key to generate security codes, which means I can't create a custom token generator unless I also create a custom authenticator app. Fortunately, Identity provides the `AuthenticatorTokenProvider<T>` class, which generates TOTP tokens for `T`, the user class. In Listing 21-11, I used the `AddTokenProvider` method to register `AuthenticatorTokenProvider<AppUser>` as the token provider for authenticators.

Creating the Seed Data

The final step is to add some seed data to make the authenticator support easier to test, as shown in Listing 21-12.

Listing 21-12. Adding Seed Data 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") }
            };
            var twoFactorUsers = new[] { "Alice", "Charlie" };
            var authenticatorKeys = new Dictionary<string, string> {
                {"Alice", "A4GG2BNKJNKKFOKGZRGBVUYIAJCUHEW7" }
            };
            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",
                    TwoFactorEnabled = twoFactorUsers.Any(tfName => tfName == name)
                };
                user.Claims =  UsersAndClaims.UserData[user.UserName]
                    .Select(role => new Claim(ClaimTypes.Role, role)).ToList();
                user.PasswordHash = PasswordHasher.HashPassword(user, "MySecret1$");
                if (authenticatorKeys.ContainsKey(name)) {
                    user.AuthenticatorKey = authenticatorKeys[name];
                    user.AuthenticatorEnabled = true;
                }
                users.TryAdd(user.Id, user);
            }
        }
    }
}
```

The changes in Listing 21-12 set an authenticator key for Alice but not the other users.

Setting Up an Authenticator

You can set up an authenticator on behalf of a user and email them the key if they have a confirmed email address. Or you can choose to let the user set up the authenticator on their own, either during self-service account creation or after they have signed into the application.

Restart ASP.NET Core, request `http://localhost:5000/users`, and click the Authenticator button for the `Alice` user. You will see the authenticator configuration for Alice, which shows her secret key as a string and a QR code, as shown in Figure 21-2.

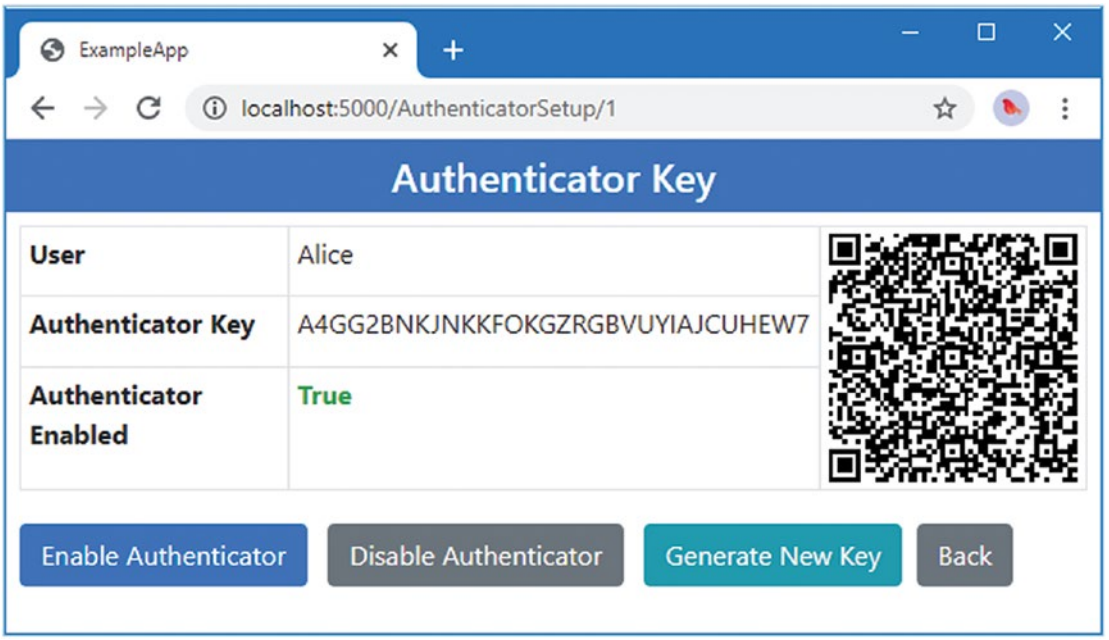

Figure 21-2. *Enabling the authenticator*

CHOOSING AN AUTHENTICATOR APP

It doesn't matter which authenticator app you use, just as long as it can generate standard TOTP codes. The main choices are the smartphone authenticator apps from Google or Microsoft (available for free in the app stores for iOS and Android). I like Authy (authy.com), which is produced by Twilio and has desktop versions alongside the usual smartphone apps. I used the Windows version of Authy for the screenshots in this chapter because the smartphone apps do not allow screenshots to be taken (to stop other apps from obtaining security codes). If you don't want to install an app, you can use the JavaScript-based authenticator available at `https://totp.danhersam.com`, which will generate TOTP codes for a specified key.

This is the key I added to the seed data in Listing 21-12. Using your authenticator app, set up a new account, which can be done by copying the key as a string or scanning the QR code. Follow your app's process for completing the process, which may ask you to select a nickname, choose an icon, and so on. Figure 21-3 shows the process using the Windows version of the Authy app. At the end of the process, your authenticator will start showing you security codes, which change every 30 seconds.

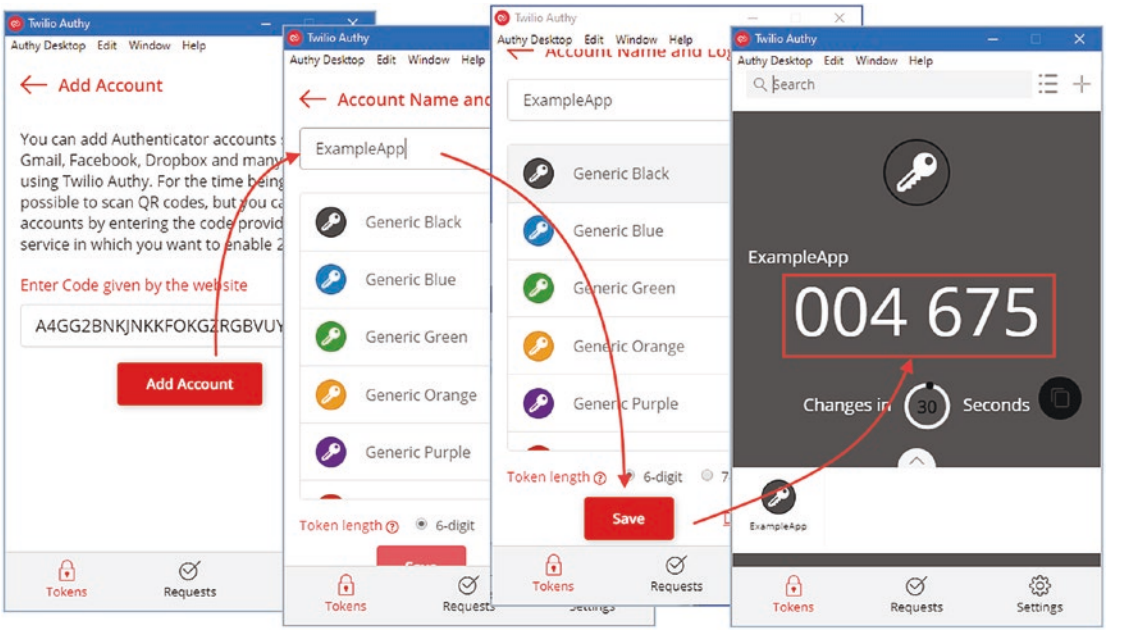

Figure 21-3. *Setting up the authenticator*

Using an Authenticator to Sign In

To test signing in with an authenticator, request `http://localhost:5000/signout`, ensure the Forget Me option is checked, and click the Sign Out button.

Next, request `http://localhost:5000/signin`, select `alice@example.com` from the list, enter MySecret1$ as the password, and click the Sign In button. Enter the code currently displayed by your authenticator app into the text field and click the Sign In button. You will be signed into the application if the code you entered matches the code generated by Identity, as shown in Figure 21-4.

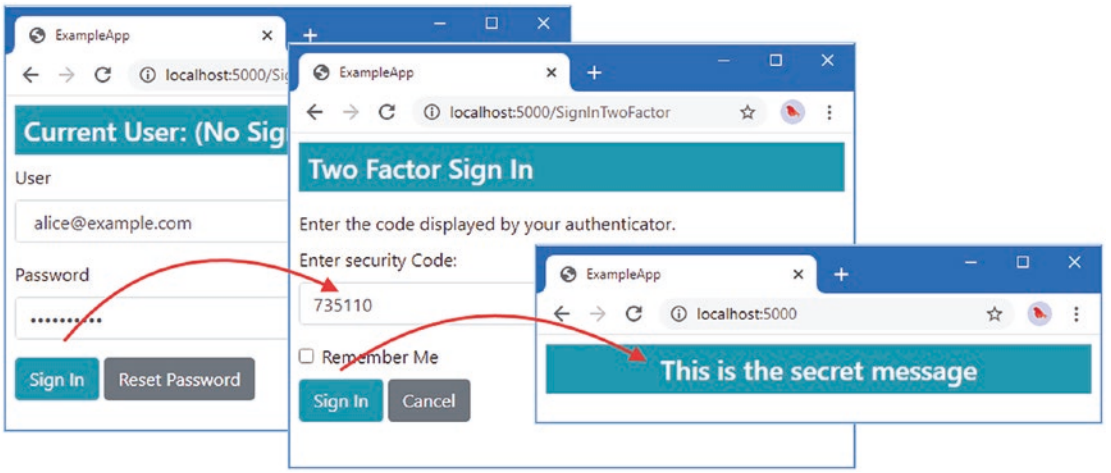

Figure 21-4. *Signing in with an authenticator*

Using Recovery Codes

Recovery codes are used to sign in when the user is unable to receive security codes or access their authenticator app. Ahead of time, the user is given a set of codes that can be redeemed during sign-in. Each recovery code can be redeemed once to sign in, after which it is invalid. In the sections that follow, I explain how ASP.NET Core Identity supports recovery codes and demonstrate their use.

Storing Recovery Codes

User stores that can manage recovery codes implement the `IUserTwoFactorRecoveryCodeStore<T>` interface, where `T` is the user class. The interface defines the methods described in Table 21-6. As with earlier user store interfaces, these methods define a `CancellationToken` parameter named `token` that is used to receive notifications when an asynchronous task is canceled.

Table 21-6. *The IUserTwoFactorRecoveryCodeStore<T> Methods*

Name	Description
`ReplaceCodesAsync(user, codes, token)`	This method replaces the existing set of recovery codes with a new set, expressed as an `IEnumerable<string>` object.
`RedeemCodeAsync(user, code, token)`	This method redeems a code for the specified user. The method returns `true` if the code is value and `false` otherwise. The code must be invalidated after it is redeemed.
`CountCodesAsync(user, token)`	This method returns the number of valid recovery codes available for the user.

To get started, I added a class file named `RecoveryCode.cs` to the `ExampleApp/Identity/Store` folder and used it to define the class shown in Listing 21-13, which will represent a single recovery code.

Listing 21-13. The Contents of the RecoveryCode.cs File in the Identity/Store Folder

```
namespace ExampleApp.Identity.Store {

    public class RecoveryCode {

        public string Code { get; set; }
        public bool Redeemed { get; set; }
    }
}
```

To extend the user store, add a class file named `UserStoreRecoveryCodes.cs` to the `ExampleApp/Identity/Store` folder and use it to define the partial class shown in Listing 21-14.

Listing 21-14. The Contents of the UserStoreRecoveryCodes.cs File in the Identity/Store Folder

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

namespace ExampleApp.Identity.Store {

    public interface IReadableUserTwoFactorRecoveryCodeStore
            : IUserTwoFactorRecoveryCodeStore<AppUser> {
        Task<IEnumerable<RecoveryCode>> GetCodesAsync(AppUser user);
    }

    public partial class UserStore : IReadableUserTwoFactorRecoveryCodeStore {
        private IDictionary<string, IEnumerable<RecoveryCode>> recoveryCodes
            = new Dictionary<string, IEnumerable<RecoveryCode>>();

        public async Task<int> CountCodesAsync(AppUser user, CancellationToken token)
            => (await GetCodesAsync(user)).Where(code => !code.Redeemed).Count();

        public Task<IEnumerable<RecoveryCode>> GetCodesAsync(AppUser user) =>
            Task.FromResult(recoveryCodes.ContainsKey(user.Id)
                ? recoveryCodes[user.Id] : Enumerable.Empty<RecoveryCode>());

        public async Task<bool> RedeemCodeAsync(AppUser user, string code,
                CancellationToken token) {
            RecoveryCode rc = (await GetCodesAsync(user))
                .FirstOrDefault(rc => rc.Code == code && !rc.Redeemed);
            if (rc != null) {
                rc.Redeemed = true;
                return true;
            }
            return false;
        }
```

```
        public Task ReplaceCodesAsync(AppUser user, IEnumerable<string>
                recoveryCodes, CancellationToken token) {
            this.recoveryCodes[user.Id] = recoveryCodes
                .Select(rc => new RecoveryCode { Code = rc, Redeemed = false });
            return Task.CompletedTask;
        }
    }
}
```

The interface provided by Identity doesn't provide a means to read the user's recovery codes once they have been stored. I find that users are more likely to use recovery codes if they can inspect the codes and see which ones have already been redeemed, so I defined the `IReadableUserTwoFactorRecoveryCodeStore` interface, which extends `IUserTwoFactorRecoveryCodeStore<AppUser>` by adding a method that allows the codes associated with the user to be retrieved.

Seeding the Data Store

Recovery codes can be any string that is known only by the application and the user. In Listing 21-15, I have added a set of codes to the seed data added to the store when it is created, which will make it easier to demonstrate the recovery code feature.

Listing 21-15. Adding Recovery Codes 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") }
            };
            var twoFactorUsers = new[] { "Alice", "Charlie" };
```

```
            var authenticatorKeys = new Dictionary<string, string> {
                {"Alice", "A4GG2BNKJNKKFOKGZRGBVUYIAJCUHEW7" }
            };
            var codes = new[] { "abcd1234", "abcd5678" };
            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",
                    TwoFactorEnabled = twoFactorUsers.Any(tfName => tfName == name)
                };
                user.Claims =  UsersAndClaims.UserData[user.UserName]
                    .Select(role => new Claim(ClaimTypes.Role, role)).ToList();
                user.PasswordHash = PasswordHasher.HashPassword(user, "MySecret1$");
                if (authenticatorKeys.ContainsKey(name)) {
                    user.AuthenticatorKey = authenticatorKeys[name];
                    user.AuthenticatorEnabled = true;
                }
                users.TryAdd(user.Id, user);
                recoveryCodes.Add(user.Id, codes.Select(c =>
                    new RecoveryCode() { Code = c }).ToArray());
            }
        }
    }
}
```

Recovery codes should be random, which they are when creating using the methods provided by the `UserManager<T>` class, which I describe in the next section. For ease of testing, however, I have defined two codes that are assigned to allow any user to sign in.

Managing Recovery Codes

The `UserManager<T>` class provides the methods described in Table 21-7 for managing recovery codes.

Table 21-7. *The UserManager<T> Methods for Recovery Codes*

Name	Description
`GenerateNewTwoFactorRecoveryCodesAsync(user, number)`	This method generates the specified number of recovery codes for a user. The news codes are passed to the store's `ReplaceCodesAsync` method, after which the user manager's update sequence is performed. Fewer codes may be generated than the number specified.
`RedeemTwoFactorRecoveryCodeAsync(user, code)`	This method redeems a recovery code for the specified user by calling the store's `RedeemCodeAsync` method. If the code is valid, the user manager's update sequence is performed. The result of this method is an `IdentiyResult`, which is used to indicate whether the code was successfully redeemed.
`CountRecoveryCodesAsync(user)`	This method returns the number of unredeemed codes available for the specified user.

Add a Razor Page named `RecoveryCodes.cshtml` to the `Pages/Store` folder with the content shown in Listing 21-16.

Listing 21-16. The Contents of the RecoveryCodes.cshtml File in the Pages/Store Folder

```
@page "{id}"
@model ExampleApp.Pages.Store.RecoveryCodesModel

<h4 class="bg-primary text-white text-center p-2">Recovery Codes</h4>

<h6 class="text-center">There are @Model.RemainingCodes codes remaining</h6>

<div class="mx-5">
    <table class="table table-sm table-striped table-bordered">
        <tbody>
            @for (int row = 0; row < Model.Codes.Length; row += 2) {
                <tr>
                    @for (int index = row; index < row + 2; index++) {
                        var rc = Model.Codes[index];
                        <td class="text-d">
                            @if (rc.Redeemed) {
                                <del>@rc.Code</del>
                            } else {
                                @rc.Code
                            }
                        </td>
                    }
                </tr>
            }
        </tbody>
    </table>
</div>
<div class="m-2 text-center">
    <form method="post">
```

```
        <input type="hidden" name="id" value="@Model.AppUser.Id" />
        <button class="btn btn-primary">Generate New Codes</button>
        <a href="@($"/users/edit/{Model.AppUser.Id}")" class="btn btn-secondary">
            Cancel
        </a>
    </form>
</div>
```

The view part of the page displays the recovery codes for a user, whose ID is received through the routing system. All of the user's codes are displayed, but those that are redeemed are struck through. Add the code shown in Listing 21-17 to the `RecoveryCodes.cshtml.cs` file to define the page model class. (You will have to create this file if you are using Visual Studio Code.)

Listing 21-17. The Contents of the RecoveryCodes.cshtml.cs File in the Pages/Store Folder

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

namespace ExampleApp.Pages.Store {

    public class RecoveryCodesModel : PageModel {

        public RecoveryCodesModel(UserManager<AppUser> manager,
                IUserStore<AppUser> store) {
            UserManager = manager;
            UserStore = store;
        }

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

        public AppUser AppUser { get; set; }

        public RecoveryCode[] Codes { get; set; }
        public int RemainingCodes { get; set;}

        public async Task OnGetAsync(string id) {
            AppUser = await UserManager.FindByIdAsync(id);
            if (AppUser != null) {
                Codes = (await GetCodes()).OrderBy(c => c.Code).ToArray();
                RemainingCodes = await UserManager.CountRecoveryCodesAsync(AppUser);
            }
        }
```

```
        public async Task<IActionResult> OnPostAsync(string id) {
            AppUser = await UserManager.FindByIdAsync(id);
            await UserManager.GenerateNewTwoFactorRecoveryCodesAsync(AppUser, 10);
            return RedirectToPage();
        }

        private async Task<IEnumerable<RecoveryCode>> GetCodes() {
            if (UserStore is IReadableUserTwoFactorRecoveryCodeStore) {
                return await (UserStore as
                    IReadableUserTwoFactorRecoveryCodeStore).GetCodesAsync(AppUser);
            }
            return Enumerable.Empty<RecoveryCode>();
        }
    }
}
```

To obtain the user's recovery codes, the page model class declares a constructor dependency on `IUserStore<AppUser>`, which provides access to the store and which is then cast to an implementation of the `IReadableUserTwoFactorRecoveryCodeStore` interface. This allows the GET handler method to populate the `Codes` property so its contents can be displayed to the user. The POST handler method is called when a new set of codes is required and calls the user manager's `GenerateNewTwoFactorRecoveryCodesAsync` method.

To incorporate the recovery codes into the rest of the application, add the elements shown in Listing 21-18 to the `_EditUserTwoFactor.cshtml` file in the `Pages/Store` folder.

Listing 21-18. Adding Content in the _EditUserTwoFactor.cshtml File in the Pages/Store Folder

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

@if (UserManager.SupportsUserTwoFactor) {
    <tr>
        <td>Two-Factor</td>
        <td><input asp-for="TwoFactorEnabled"/></td>
    </tr>
}
@if (UserManager.SupportsUserTwoFactorRecoveryCodes) {
    <tr>
        <td>Recovery Codes</td>
        <td>
            @(await UserManager.CountRecoveryCodesAsync(Model)) codes remaining
                <a asp-page="RecoveryCodes" asp-route-id="@Model.Id"
                   class="btn btn-sm btn-secondary align-top">Change</a>
        </td>
    </tr>
}
```

Restart ASP.NET Core, request `http://localhost:5000/users`, and click the Edit button for Alice. You will see a row in the table that shows how many codes are available. Click the Change button, and you will see the user's recovery codes. Click the Generate New Codes button, and a new set of codes will be produced, as shown in Figure 21-5. The new codes will be lost when ASP.NET Core restarts because the user store is memory based.

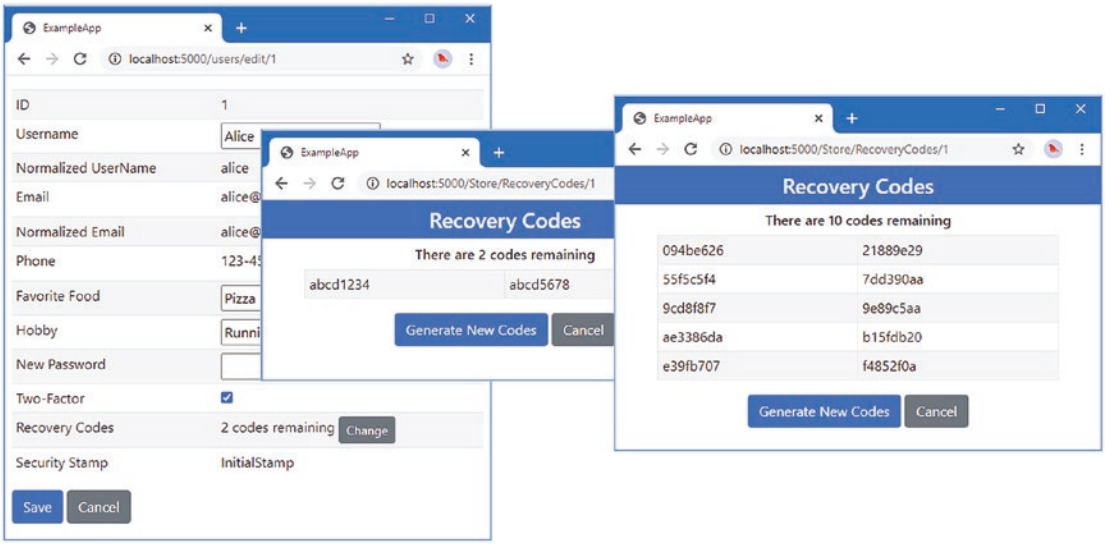

Figure 21-5. *Managing recovery codes*

Using Recovery Codes to Sign In

Recovery codes are redeemed when the user is unable to receive a security code, either through SMS/email or an authenticator. The `SignInManager<T>` class provides the method described in Table 21-8 for signing in with a recovery code.

Table 21-8. *The SignInManager<T> Recovery Code Method*

Name	Description
`TwoFactorRecovery CodeSignInAsync(code)`	This method completes the two-factor sign-in process using the specified code. The `RetrieveTwoFactorInfoAsync` method is used to retrieve details of the user signing in, and the user manager's `RedeemTwoFactorRecoveryCodeAsync` method is used to validate the code. If the code is valid, the user is signed into the application. The option to remember the client is disabled for sign-ins performed with a recovery code.

Add a Razor Page named `SignInRecoveryCode.cshtml` to the `Pages` folder and add the content shown in Listing 21-19.

Listing 21-19. The Contents of the SignInRecoveryCode.cshtml File in the Pages Folder

```
@page
@model ExampleApp.Pages.SignInRecoveryCodeModel

<h4 class="bg-info text-white m-2 p-2">Two Factor Sign In</h4>

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

<div class="m-2">
```

```
    <form method="post">
        <div class="form-group">
            <label>Enter Recovery Code:</label>
            <input class="form-control" name="code"/>
        </div>
        <div class="mt-2">
            <button class="btn btn-info" type="submit">Sign In</button>
            <a asp-page="/Signin" class="btn btn-secondary">Cancel</a>
        </div>
    </form>
</div>
```

The view part of the page displays an input element into which the user can enter one of their recovery codes. To define the page model class, add the code shown in Listing 21-20 to the SignInRecoveryCode.cshtml.cs file in the Pages folder. (You will have to create this file if you are using Visual Studio Code.)

Listing 21-20. The Contents of the SignInRecoveryCode.cshtml.cs File in the Pages Folder

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

namespace ExampleApp.Pages {

    public class SignInRecoveryCodeModel : PageModel {

        public SignInRecoveryCodeModel(SignInManager<AppUser> manager)
            => SignInManager = manager;

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

        public async Task<IActionResult> OnPostAsync (string code,
                string returnUrl) {
            if (string.IsNullOrEmpty(code)) {
                ModelState.AddModelError("", "Code required");
            } else {
                SignInResult result =
                    await SignInManager.TwoFactorRecoveryCodeSignInAsync(code);
                if (result.Succeeded) {
                    return Redirect(returnUrl ?? "/");
                } else {
                    ModelState.AddModelError("", "Sign In Failed");
                }
            }
            return Page();
        }
    }
}
```

The page model class uses the SignInManager<T>.TwoFactorRecoveryCodeSignInAsync method to try to complete the two-factor sign in process using the recovery code provided by the user. A validation error is displayed if the user has not provided a code or the code is invalid.

To allow the user to provide a recovery code, add the element shown in Listing 21-21 to the SignInTwoFactor Razor Page.

Listing 21-21. Adding an Element in the SignInTwoFactor.cshtml File in the Pages Folder

```
@page
@model ExampleApp.Pages.SignInTwoFactorModel

<h4 class="bg-info text-white m-2 p-2">Two Factor Sign In</h4>

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

@if (Model.AuthenticatorEnabled) {
    <span class="m-2"> Enter the code displayed by your authenticator. </span>
} else {
    <span class="m-2"> We have sent a security code to your phone. </span>
    <a asp-page="/SignInTwoFactor" class="btn btn-sm btn-secondary">Resend Code</a>
}

<div class="m-2">
    <form method="post">
        <div class="form-group">
            <label>Enter security Code:</label>
            <input class="form-control" name="code"/>
        </div>
        <div class="form-check">
            <input class="form-check-input" type="checkbox" name="rememberMe" />
            <label class="form-check-label">Remember Me</label>
            <a asp-page="SigninRecoveryCode" class="btn btn-sm btn-secondary ml-3"
               asp-route-returnurl="@HttpContext.Request.Query["returnUrl"]">
                    Use Recovery Code
            </a>
        </div>
        <div class="mt-2">
            <button class="btn btn-info" type="submit"
                    disabled="@(!ModelState.IsValid)">Sign In</button>
            <a asp-page="/Signin" class="btn btn-secondary">Cancel</a >
        </div>
    </form>
</div>
```

Restart ASP.NET Core, request http://localhost:5000/signout, select the Forget Me option, and click the Sign Out button. This ensures that no user is signed in and the client will not be remembered.

Request http://localhost:5000/secret to trigger the challenge response. Select alice@example.com from the list and enter MySecret1$ into the password field. Click the Sign In button, and you will advance to the second stage of the sign-in process. Click the Use Recovery Code button, enter abcd1234 into the text field, and click the Sign In button. You will be signed into the application and redirected to the /secret URL, as shown in Figure 21-6.

Figure 21-6. *Using a recovery code*

Request `http://localhost:5000/store/recoverycodes/1`, and you will see the recovery codes for the Alice user. The code you used to sign in is shown as redeemed, as shown in Figure 21-7.

Figure 21-7. *The effect of using a recovery code*

Summary

In this chapter, I described the Identity support for authenticators as part of a two-factor sign-in. Authenticators are set up using a secret key, which is used to generate tokens that the user can present to the application and that can be validated by Identity using the same key. I also described the support for recovery codes, which provide an important means for signing into the application when the second factor in a two-factor process is unavailable. In the next chapter, I describe external authentication, where the user authentication process is handled by a third party.